FORMAL LANGUAGES AND AUTOMATA THEORY

B Tech III Year I Sem

Course Code	Category	Hours/Week			Credits	Maximum Marks		
		L	T	P	C	CIE	SEE	TOTAL
23CY503	Professional Core	3	0	0	3	40	60	100
Contact Classes: 48	Tutorial Classes: Nil	Practical Classes: -				Total Classes: 48		
Prerequisite: A Course on Discrete Mathematics								

Course Objectives

- To provide an introduction to some of the central ideas of theoretical computer science from the perspective of formal languages.
- To introduce the fundamental concepts of formal languages, grammars, and automata theory.
- To classify machines by their power to recognize languages.
- To employ finite state machines to solve problems in computing.
- To understand deterministic and non-deterministic machines.
- To understand the differences between decidability and undecidability.

Course Outcomes

- Able to understand the concept of abstract machines and their power to recognize the languages.
- Able to employ finite state machines for modeling and solving computing problems.
- Able to design context-free grammars for formal languages.
- Able to distinguish between decidability and un decidability.
- Able to gain proficiency with mathematical tools and formal methods.

MODULE-I

Introduction to Finite Automata: Structural Representations, Automata and Complexity, the Central Concepts of Automata Theory-Alphabets, Strings, Languages, Problems. Nondeterministic Finite Automata: Formal Definition, an application, Text Search, Finite Automata with Epsilon-Transitions. **Deterministic Finite Automata:** Definition of DFA, How a DFA Process Strings, The language of DFA, Conversion of NFA with €-transitions to NFA without -transitions.

MODULE-II

Regular Expressions: Finite Automata and Regular Expressions, Applications of Regular Expressions, Algebraic Laws for Regular Expressions, Conversion of Finite Automata to Regular Expressions. Pumping Lemma for Regular Languages: Statement of the pumping lemma, Applications of the Pumping Lemma., Closure Properties of Regular Languages: Closure properties of Regular languages, Decision, Properties of Regular Languages, Equivalence and Minimization of Automata.

MODULE- III

Context-Free Grammars: Definition of Context- Free Grammars, Derivations Using a Grammar, Left most and Right most Derivations, the Language of a Grammar, Sentential Forms, Parse Tress, Applications Grammars and Languages. of Context-Free Grammars, Ambiguity in
Push Down Automata: Definition of the Push down Automaton, the Languages of a PDA, Equivalence of PDA's and CFG's, Acceptance by final state, Acceptance by empty stack, Deterministic Pushdown Automata. From CFG to PDA, From PDA to CFG.

MODULE- IV

Normal Forms for Context-Free Grammars: Eliminating useless symbols, Eliminating €-Productions. Chomsky Normal form, Griebech Normal form. Pumping Lemma for Context-Free Languages: Statement of pumping lemma, Applications.
Closure Properties of Context-Free Languages: Closure properties of CFL's, Decision Properties of CFL's Turing Machines: Introduction to Turing Machine, Formal Description, Instantaneous description, The language of a Turing machine

MODULE- V

Types of Turing machine: Turing machines and halting
Undesirability: Undesirability, A Language that is Not Recursively Enumerable, An Undividable Problem That is RE, Undividable Problems about Turing Machines, Recursive languages, Properties of recursive languages, Post's Correspondence Problem, Modified Post Correspondence problem, Other Undividable Problems, Counter machines.

TEXT BOOKS:
1. Introduction to Automata Theory, Languages and Computation, 3rd Edition, John E.Hopcroft, Rajeev Motwani, Jeffrey D.Ullman, Pearson Education.
2. Theory of Computer Science-Automata languages and computation, Mishra and Chandrashekaran, 2nd edition, PHI.

REFERENCE BOOKS:
1. Introduction to Languages and the Theory of Computation, John C Martin, TMH.
2. Introduction to Computer Theory, Daniel I.A. Cohen, John Wiley.
3. A Textbook on Automata Theory, P.K.Srimani, NasirS.F.B, Cambridge University Press.
4. Introduction to the Theory of Computation, Michael Sipser, 3rd edition, Cengage Learning.
5. Introduction to Formal languages Automata Theory and Computation Kamala Krithivasan, Rama R, Pearson

FORMAL LANGUAGE AUTOMATA THEORY

UNIT - I

Automata = an abstract computing devices.

→ Automata theory is the study of computation.
↓
Processing
(processes input to output)

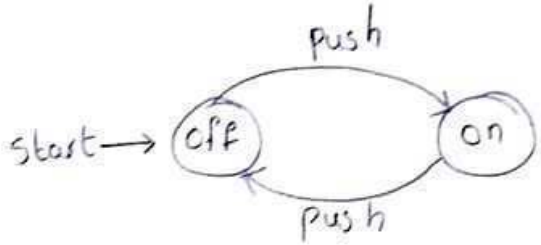
→ Automata theory deals with logic of computation with respect to simple machines, referred to as automata.

→ Through automata, we understand how machines compute functions and solve problems.

→ Alan Turing (1912-1954) is the father of modern computer, had developed a machine called Turing m/c.

→ Turing m/c is the basic model that describes the entire functionality of all the machines (i.e., set of i/p's, set of outputs and the processor).

Example. Finite Automata model for the switch is



→ For all finite automata states are represented by circles. In this example we have two states 'on' and 'off'. Arcs blw states are labeled by 'inputs', which represent external influences on the system.

Structural Representation:

→ There are two important notations that play an important role in the study of automata and their applications.

① Grammars

② Regular Expressions

→ Grammars are useful models when designing software that processes data with recursive structure.

Ex: A grammatical rule like $E \Longrightarrow E+E$ states that an expression is formed by taking any two expressions and connecting them by plus sign.

→ Regular Expressions: R.E denote the structure of data, especially text strings.

UNIX style R.E is '[A-Z][a-z]* []][A-Z][A-Z]' represent capitalized words followed by a space and two capital letters. This expression represents the patterns in text that could be a city and state Hyderabad TE.

Automata and complexity:

→ Automata is the study of limits of computation.

There are two important issues:

① What can a computer do at all? This study is called "decidability".

② what can a computer do efficiently? This study is called "intractability". An the problems tha can be solved by computer using no more time is called 'tractable".

The Central concepts of Automata Theory :

Three things used to design a machine :

① Alphabet (set of symbols)

② Strings (list of symbols from an alphabet)

③ Language (a set of strings from the same alphabet).

Alphabet:

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

→ Representation : Σ (sigma)

common alphabets include

- $\Sigma = \{0, 1\}$, the binary alphabet.
- $\Sigma = \{a, b, \ldots z\}$, set of all lower case letters.
- $\Sigma = \{0, 1, 2, \ldots 9\}$, set of all ASCII characters or digits.
- $\Sigma = \{a\text{-}z, A\text{-}z, 0\text{-}9\}$ Alphanumeric

Strings:

→ A string or word is a finite collection of symbols selected from the alphabets (Σ).

Ex: 01101 is a string from binary alphabet $\Sigma = \{0, 1\}$.

→ The Empty string is the string with zero occurences of symbols. Represented by ϵ ('Epsilon').

Length of string:

→ The "length" of the string is denoted by $|w|$ and it is the number of positions for the symbol in the string.

Ex: $w = 01101$ has length = 5 i.e, $|w| = 5$

Powers of an alphabet :

→ If Σ is an alphabet, we can express the set of all strings of certain length from that alphabet by using an exponential notation. 'Σ^k' represent set of string of length K.

Ex $\Sigma^0 = \epsilon$

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

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

$\Sigma^2 = \{aa, ab, ac, ba, bb, cc, \}$

$\Sigma^3 = \{aaa, abc, aab, bbb, aac, \ldots\}$

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

$\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \ldots$ is set of non empty strings

$\Sigma^* = \Sigma^+ \cup \{\epsilon\}$.

Concatenation of strings:

→ Let x & y be two strings. Then xy denotes the concatenation of x & y i.e, the string formed by making a copy of x and following it by a copy of y.

Ex: $x = ab$ & $y = cd$ then $xy = abcd$.

Reverse of string:

→ Reverse of the string can be achieved by simply interchanging over last symbols.

Ex $w = abc$ the $w^R = cba$

Languages:

→ A set of strings which are chosen from some Σ^*, where Σ is a particular alphabet is called a language.

Ex - The language of all strings consisting of n 0's followed by n 1's for some $n \geq 0$. $\{\epsilon, 01, 0011, 000111, \cdots\}$

- The set of strings of 0's and 1's with equal number of each: $\{\epsilon, 01, 10, 0101, 0011, 1001, \cdots\}$
- Set of binary numbers whose value is prime.

$$\{10, 11, 101, 111, 1011, \cdots\}$$

- Σ^* is a language for any alphabet Σ.
- $\emptyset$, the empty language, over any alphabet.

Problems:

→ Problems is the question of deciding whether a given string is a member of some particular language

→ If Σ is an alphabet, and L is a language over Σ, then the problem L is:

- Given a string w in Σ^*, decide whether or not w is in L.

Finite Automata.

① DFA (Deterministic Finite Automata)

② NFA (Non-deterministic Finite automata)

Non-Deterministic Finite Automata:

→ F.A are mainly used for pattern recognition.
→ It takes string of symbol as i/p and changes its state accordingly.
→ when desired symbol is found, then transition occurs.
→ A the time of transition, the automata can either move to the next state or stay in same state.
→ Finite automata has two states, Accept state or Reject state.

Formal Description of FA

A FA is a collection of 5-tuple $(Q, \Sigma, q_0, F, \delta)$ where:

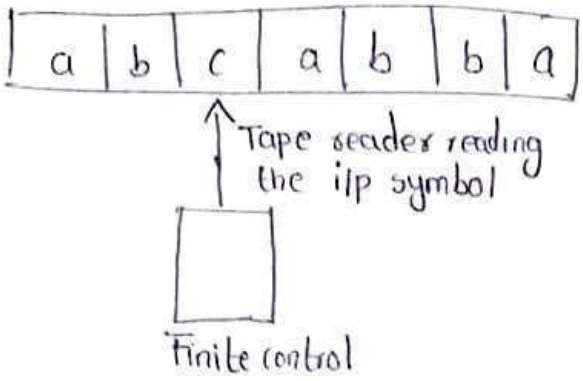
1. Q: finite set of states
2. Σ: Finite set of the input symbol.
3. q_0: Initial state.
4. F: Final state.
5. δ: Transition function. (movement of string from one state to another state)

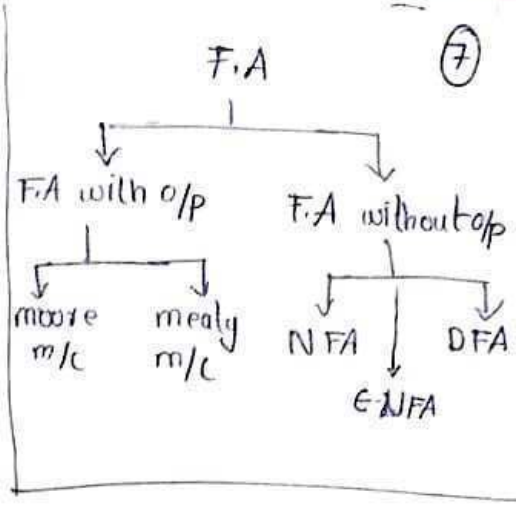
Finite Automata model

F.A can be represented by input tape and finite control.

Input tape: It is a linear tape having some number of cells. Each input symbol is placed in each cell.

Finite control: The finite control decides the next state on receiving particular input from input tape. The tape reader reads the cells one by one from left to right, and at a time only one input symbol is read.





Deterministic Finite Automata (DFA)

→ DFA is in a single state after reading any sequence of inputs.

→ The term 'deterministic' refers to fact that on each input there is one and only one state to which the automaton can transition from its current state.

→ DFA does not accept the null move i.e., DFA cannot change state without any input characters.

Definition of a DFA:

→ A DFA is a collection of 5-tuples

$$F.A = \{Q, \Sigma, q_0, F, \delta\}$$

1. Q : Finite set of states.
2. Σ : Finite set of the input symbol.
3. q_0 : initial state.
4. F : final state.
5. δ : Transition function that takes as arguments a state and an input symbol and returns a state.

$\delta : Q \times \Sigma \longrightarrow Q$ | $\delta(q, a) \longrightarrow p$

How a DFA Processes strings:

→ First understand how DFA decides whether or not to "accept" a sequence of input symbols.

→ Suppose $a_1 a_2 a_3 \ldots a_n$ is a sequence of input symbols, we start with start state q_0, transition function δ is

$$\delta(q_0, a_1) = q_1$$

Then we process next input symbol a_2 by evaluating $\delta(q_1, a_2)$ which enters state q_2.

→ we continue in this manner finding states $q_3, q_4 \ldots q_n$ such that $\delta(q_{i-1}, a_i) = q_i$ for each i.

If q_n is a member of F, then the input $a_1 a_2 \ldots a_n$ is accepted, if not it is "rejected".

example: specify a DFA that accepts all and only the strings of 0's and 1's that have the sequence 01 somewhere in the string.

Language L is as follows

$\{w \mid w$ is of the form $x01y$ for some strings x and y consisting of 0's and 1's only$\}$

(or)

$\{x01y \mid x$ and y are any strings of 0's and 1's$\}$

→ strings in this language include 01, 0001, 11010, ...

and strings not in this language include ϵ, 0, 111000, ...

Simpler Notations for DFA's:

There are two preferred notations for describing automata:

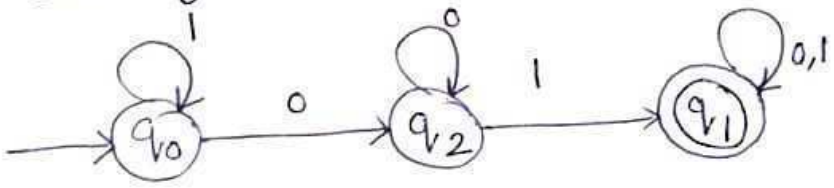
1. Transition diagram (which is a graph)
2. Transition Table (which is a tabular listing of δ function, which by implication tells us the set of states and the input alphabet)

Transition diagram:

→ A transition diagram for a DFA $A=(Q, \Sigma, \delta, q_0, F)$ is defined as

- For each state in Q there is a node.
- For each state q in Q and each input symbol a in Σ let $\delta(q,a)=P$, then transition diagram has an arc from node q to node P labeled a.
- There is an arrow into the start state q_0, labeled 'start'
- Nodes corresponding to accepting states are marked by a double circle. states not in F have a single circle.

Ex. Transition diagram for DFA accepting all strings with a substring 01.



Transition Tables:

→ Transition table is a tabular representation of a function like δ that takes two arguments and returns a value.

→ Rows correspond to states and columns correspond to the inputs.

Ex: Transition table for DFA accepting all strings with a substring 01.

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

→ Extended transition function ($\hat{\delta}$) is a function that takes a state q and a string w and returns a state p - the state the automaton reaches when starting in state q and process the input w.

Ex: suppose w is a string of the form xa, a is the last symbol of w. If $w = 1101$ is broken in to $x = 110$ & $a = 1$

then $\hat{\delta}(q, w) = \delta(\hat{\delta}(q, x), a)$

i/p: 1101

$\hat{\delta}(q_0, \epsilon) = q_0$

$\hat{\delta}(q_0, 1) = \delta(\hat{\delta}(q_0, \epsilon), 1) = \delta(q_0, 1) = q_0$

$\hat{\delta}(q_0, 11) = \delta(\hat{\delta}(q_0, 1), 1) = \delta(q_0, 1) = q_0$

$\hat{\delta}(q_0, 110) = \delta(\hat{\delta}(q_0, 11), 0) = \delta(q_0, 0) = q_2$

$\hat{\delta}(q_0, 1101) = \delta(\hat{\delta}(q_0, 110), 1) = \delta(q_2, 1) = q_1$

The language of DFA:

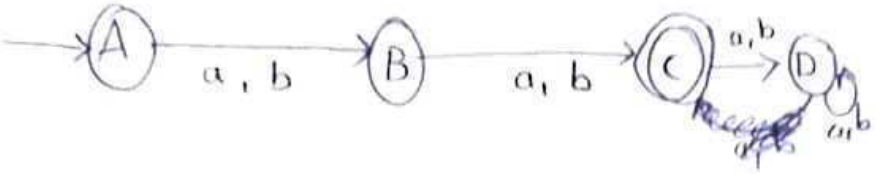
→ The language of DFA $A = (Q, \Sigma, \delta, q_0, F)$ is denoted by $L(A)$, is defined by

$$L(A) = \{ w \mid \hat{\delta}(q_0, w) \text{ is in } F \}$$

i.e., the language of A is the set of strings w that take the start state q_0 to one of the accepting states.

Example 1: construct a DFA over $\Sigma = \{a, b\}$ where the length is equal to 2.

Solution: The language will be $L = \{aa, ab, ba, bb\}$

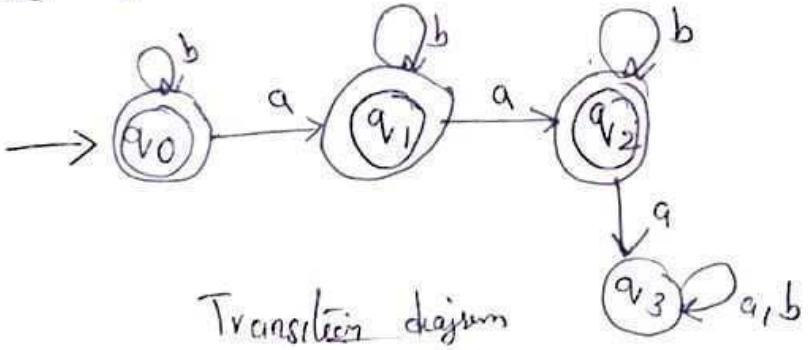


→ First write down lang.
→ Try to accept all the strings
→ Maintain DFA restrictions

Example 2.

Design DFA which accepts all the strings not having more than two a's over $\Sigma = \{a, b\}$.

Solution: In this DFA maximum two a's are accepted. If we try to accept third a then it should not lead us to final state.



Transition diagram

	a	b
→ q_0	q_1	q_0
q_1	q_2	q_1
q_2	q_3	q_2
q_3	q_3	q_3

Example : Design DFA for the following languages shown below $\Sigma = \{a, b\}$

(a) $L = \{w \mid w$ does not contains the substring $ab\}$

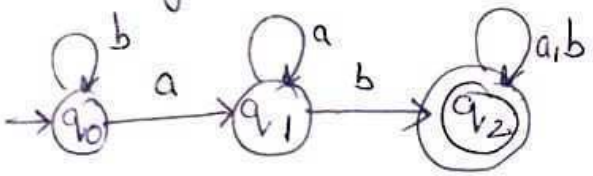
(b) $L = \{w \mid w$ contains neither the substring ab nor $ba\}$

(c) L $\{w \mid w$ is any string that does not contain exactly two a's$\}$

(d) $L = \{w \mid w$ is any string except a & $b\}$

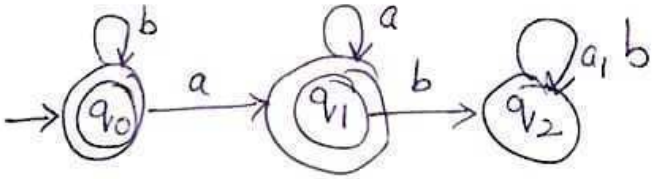
Solution:

(a) For designing the DFA specified by language L we will first design DFA that contain substring ab.

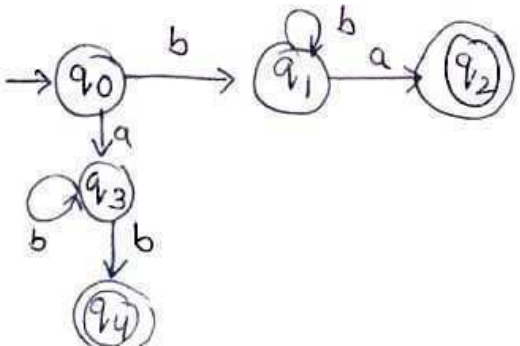


Now, For DFA that accepts a language L, which do not contain substring ab, we will simply change the accept state (ie., final state to non final state and non final states will be made final states.

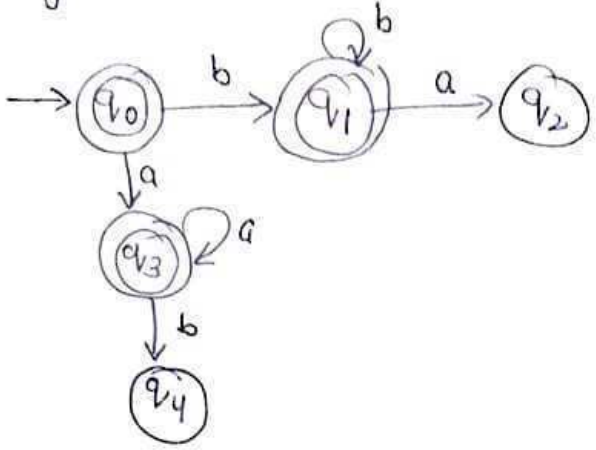
The DFA will then be as follows.



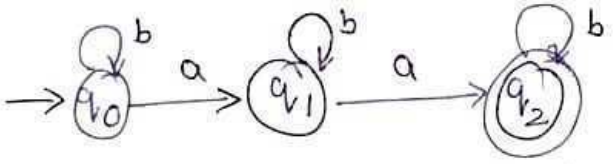
(b) The DFA that contains ab or ba is



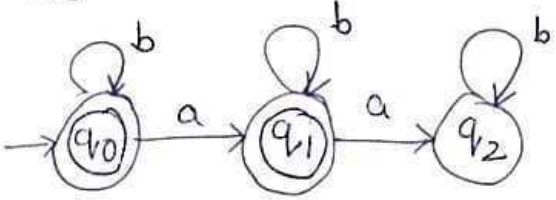
Now if we change non-final state to final and final state to non final state then the DFA is as follows. This DFA accepts strings which do not contain the substring ab or ba.



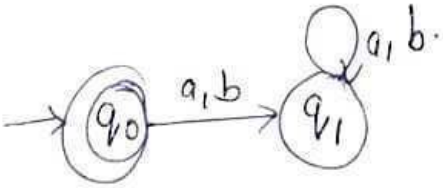
(c) The DFA that contains a language having two a's exactly:



Now we will exchange the final and non-final states and then the DFA which will be obtained will represent a language that doesn't contain exactly two a's.

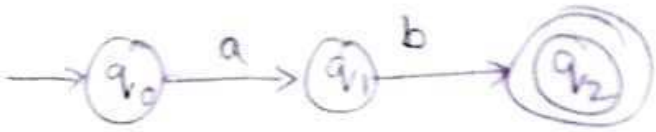


(d) The DFA that does not contain a, b is



Ex: Obtain a DFA to accept strings of a's & b's starting with the string ab

Solution: It is clear that the string should start with 'ab' and so the minimum string ab, we need three states and the machine can be written as

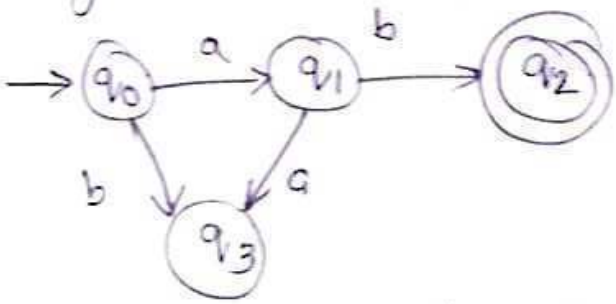


q_0 - starting / initial state

q_2 - Final / accepting state

→ since the starting char shouldn't be 'b' the input symbol 'b' is pointed to the rejected / dead state q_3 from q_0.

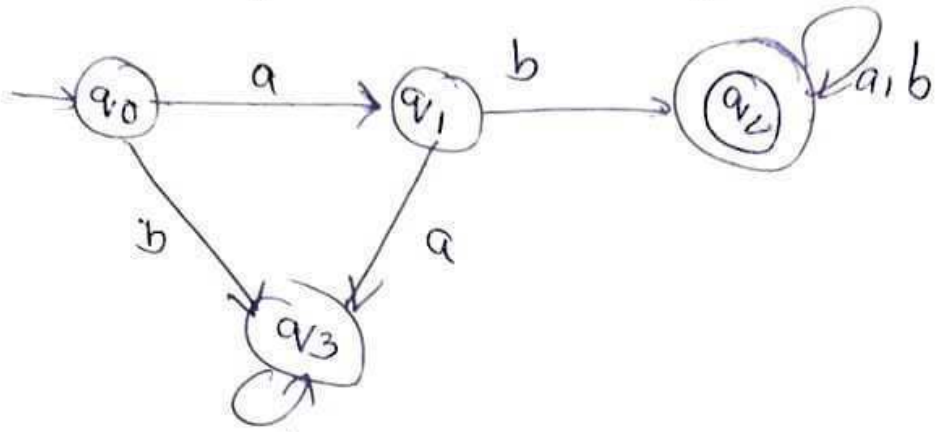
→ since the second char should'nt be 'a' the I/p symbol 'a' is pointed to dead state q_3 from q_1



→ whenever the string is not starting with ab, the machine will be in state q_3 which is dead state

→ After the string ab, the string containing any combination of a's & b's can be accepted and so remain in the state q_2 only

→ The complete m/c to accept the string of a's & b's starting with the string ab is as follows



The language accepted by DFA can be represented as $L = \{ ab(a+b)^n / n \geq 0 \}$ (or) $L = \{ ab(a+b)^* \}$

$Q - \{q_0, q_1, q_2, q_3\}$

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

$q_0 - \{q_0\}$

$F - \{q_2\}$

Transition table :

states	a	b
→ q_0	q_1	q_3
q_1	q_3	q_2
(*) q_2	q_2	q_2
q_3	q_3	q_3

(2) Ex: Draw a DFA to accept string of 0's and 1's ending with the string '011'. (16)

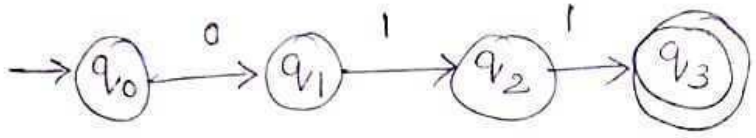
Solution: Given $\Sigma = \{0, 1\}$

L is All the strings ending with string 011.

So

$L = \{011, 0011, 111 \ldots 011, 01011001 \ldots 011, \ldots\}$

→ The basic string is 011, so draw FA for this using $Q = \{q_0, q_1, q_2, q_3\}$



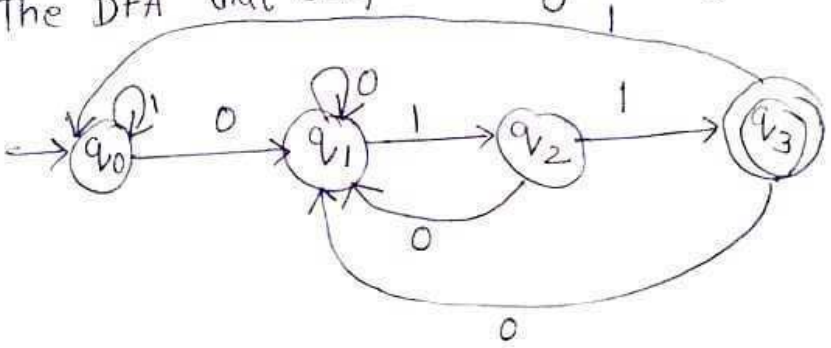
→ Since the starting symbol can be anything the input symbol '1' on q_0 is self looped.

→ Since the ending must be 011 the input '0' on q_1 is self-looped.

→ The input '0' on q_2 is pointed to q_1 so that it can end as 011.

→ '0' and '1' input on q_3 is pointed to q_1 and q_0 respectively.

→ The DFA that accepts strings ending with 011 is as follows



Transition table

states	symbols 0	1
→ q_0	q_1	q_0
q_1	q_1	q_2
q_2	q_1	q_3
(q_3)	q_1	q_0

Non - Deterministic Finite Automata (NFA):

→ The transition from a state can be to multiple next states for each input symbol.

→ NFA Permits empty string transitions.

A NFA is represented essentially like a DFA

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

where ① Q is a finite set of states.

② Σ is a finite set of input symbols

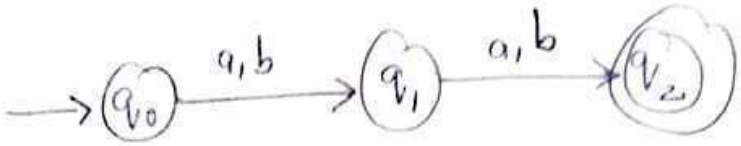
③ q_0, a member of Q, is the start state.

④ F, a subset of Q, is the set of final or accepting states

⑤ δ, the transition function that takes a state in Q and an input symbol in Σ as arguments and returns a subset of Q.

Example: construct NFA over $\Sigma = \{a, b\}$ where the length is equal to 2.

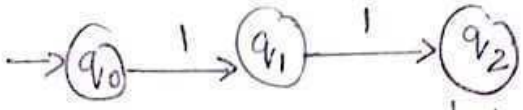
Solution: The language will be $L = \{aa, ab, ba, bb\}$



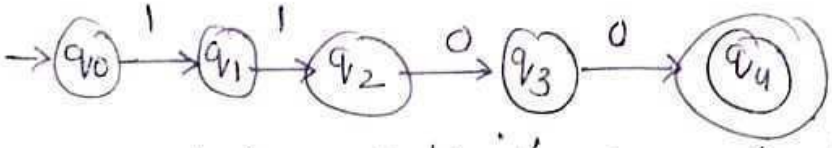
Example:

Design an NFA with $\Sigma = \{0, 1\}$ in which double '1' is followed by double 'zero'.

Sol: The FA with double '1' is as follows

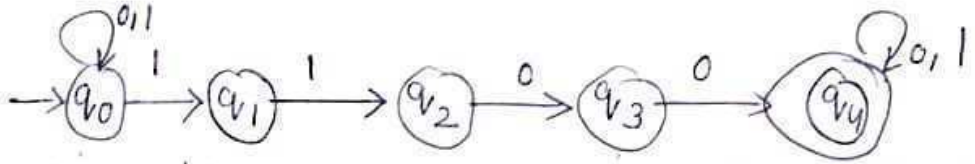


→ It should be immediately followed by '0'



→ Now before double '1' there can be any string of 0's & 1's, similarly after double '0' there can be any string of 0's & 1's.

→ Hence NFA becomes

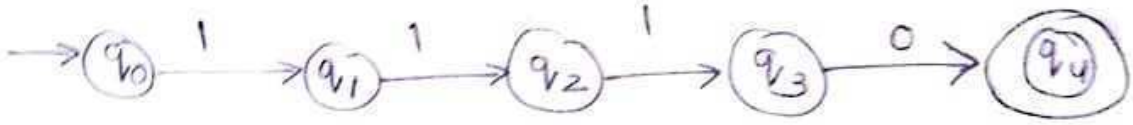


Transition table is as follows:

	0	1
→q_0	q_0	$\{q_1, q_0\}$
q_1	$\emptyset$	q_2
q_2	q_3	$\emptyset$
q_3	q_4	$\emptyset$
*q_4	q_4	q_4

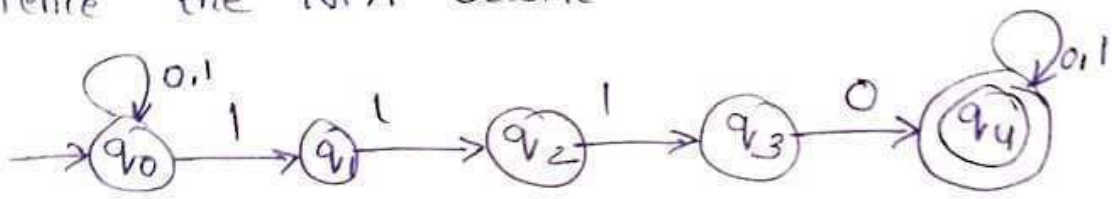
Example(2) Design a NFA in which all the string contain a substring 1110.

Solution The language consists of all the strings containing 1110. The partial transition diagram can be:



→ Now as 1110 could be the substring Hence we will add the inputs 0's and 1's so, that the substring 1110 of the language can be maintained.

→ Hence the NFA becomes:



Transition table:

States / input	0	1
→ q_0	q_0	$\{q_0, q_1\}$
q_1	ϕ	q_2
q_2	ϕ	q_3
q_3	q_4	ϕ
* q_4	q_4	q_4

consider the string 111010

$\delta(q_0, 111010) = \delta(q_0, 11010)$

$= \delta(q_0, 1010)$

$= \delta(q_2, 1010)$

$= \delta(q_3, 010)$

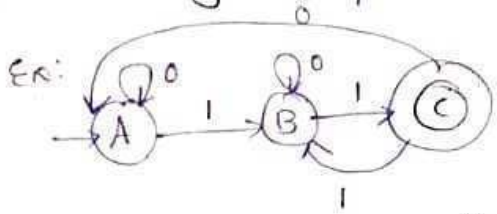
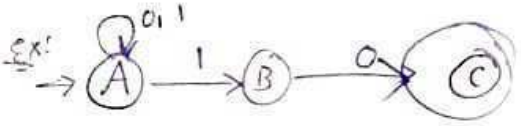
$= \delta(q_4, 10)$

$= \delta(q_4, 0)$

$= \delta(q_4, \epsilon)$

→ Hence q_4 is the acceptance state, the string is acceptable.

Differences b/w DFA & NFA

DFA	NFA
1) Deterministic Finite Automata	1) Non-Deterministic ~~finite Automata~~
2) DFA is complete system	2) It may/maynot be complete.
3) Every DFA is NFA	3) Every NFA may/maynot be DFA. But NFA can be converted to DFA
4) For every input string there exists only one path. Ex: 	4) For every input string there exist 0 or 1 more no of paths Ex: 
5) DFA is as powerful as NFA	5) NFA is as powerful as DFA
6) DFA is more efficient.	6) NFA is comparatively less efficient.
7) $\delta : Q \times \Sigma \rightarrow Q$	7) $\delta : Q \times \Sigma \rightarrow 2^Q$
8) Here we specify what is required to us and what not is required to us	8) Here we specify what is required only.
9) Design complexity is more	9) comparatively less.
10) Dead state may be required.	10) Dead state is not required.
11) DFA can be understood as one machine	11) NFA can be understood as multiple little machines computing at same time.

Applications of Finite Automata :

→ The applications of Finite Automata are :-

1) A finite automata is highly useful to design lexical Analysers.
2) It is useful to design text editors.
3) It is highly useful to design spell checkers.
4) It is useful to design sequential circuit design (transducers).
5) Software for scanning large bodies of text (Eg: web pages) for pattern finding.
6) Software for verifying systems of all types, that have a finite number of states.

(Eg: Stock market transaction, communication / network protocol)

NFA for Text search :

→ NFA are mostly used for performing text search from group of words usually called as Keywords. To determine the occurence of any keyword with in a text.

→ For this reason a NFA is designed. This automata is designed in such a way that it send signals upon seeing a keyword in accepting state.

→ Here the document compressing text is sent from time to time to NFA.

→ So that it finds occurences of the keyword in the text.

NFA to search text is designed as follows:

1) The FA consist of a start state with a transition to every input symbol.
2) There exist K states for the keywords $m_1, m_2, \ldots m_K$
3) A transition from the start state of q_1 on symbol m_1 will exist where in a transition from q_1 to q_2 exist on symbol m_2 and on forth q_K state represents that it is an accepting state and signifies, it has found all the keywords $m_1, m_2, \ldots m_K$.

NFA with Epsilion Transitions:

ϵ – Empty string / Null string

→ In this type of machines changing of state is possible without reading the strings.

→ Such transitions are labeled with ϵ-transition.

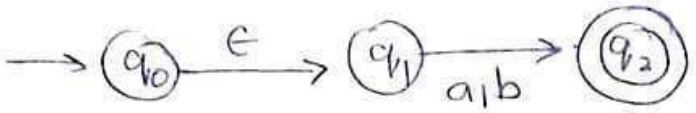
→ The symbol ϵ does not belong to any alphabet.

A ϵ-NFA is a $A = (Q, \Sigma, \delta, q_0, F)$

- Q is set of states.
- Σ is the alphabet of input symbols.
- q_0 is initial state.
- F is the set of final states.
- $\delta : Q \times \Sigma\epsilon \rightarrow P(Q)$ is the transition function.

ϵ-closure: ϵ-closure for a given state "Q" means a set of states which can be reached from state "Q" with only ϵ (null) move including the state "Q" itself.

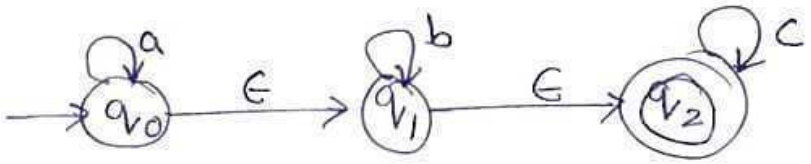
Example: A NFA accepting a language having length exactly 1 over $\Sigma = \{a, b\}$.



Example: Construct NFA with 'ϵ' which accepts a language consisting the strings of any number of a's followed by any number of b's. Followed by any number of c's.

Solution: Here any number of a's or b's or c's means zero or more in number. That means there can be zero or more a's followed by zero or more b's followed by zero or more c's.

Hence NFA with ϵ can be



The transition table is:

state \ input	a	b	c
→ q_0	q_0	ϕ	ϕ
q_1	ϕ	q_1	ϕ
*q_2	ϕ	ϕ	q_2

ϵ-closure$(q_0) = \{q_0, q_1, q_2\}$

ϵ-closure$(q_1) = \{q_1, q_2\}$

ϵ-closure$(q_2) = \{q_2\}$

we can parse the string aabbcc as follows:

$\delta(q_0, aabcc) - \delta(q_0, abcc)$

$- \delta(q_0, bcc)$

$- \delta(q_0, \epsilon bcc)$

$- \delta(q_1, bcc)$

$- \delta(q_1, cc)$

$- \delta(q_1, \epsilon cc)$

$- \delta(q_2, c)$

$- \delta(q_2, c)$

$- \delta(q_2, \epsilon)$

$= q_2$

Conversion of NFA with ϵ to NFA without ϵ.

In this method we try to remove all the ϵ transitions from the given NFA. The method will be.

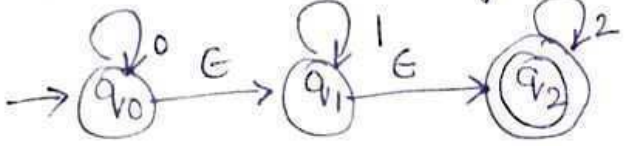
1. Find out all the ϵ-transitions from each state from Q. This will be called as ϵ-closure$\{q_i\}$ where $q_i \in Q$.
2. Then δ' transitions can be obtained. The δ' transitions means an ϵ-closure on δ moves.
3. Step 2 is repeated for each input symbol and for each state of given NFA.
4. Using the resultant states the transition table for equivalent NFA without ϵ can be built.

Rule for conversion

$$\delta'(q, a) = \epsilon\text{-closure}\left(\hat{\delta}(\hat{\delta}(q, \epsilon), a)\right)$$

where $\hat{\delta}(q, \epsilon) = \epsilon\text{-closure}(q)$

Example: convert the given NFA with ϵ to NFA without ϵ.



Solution: we will first obtain ϵ-closure of each state. i.e. we will find out ϵ reachable states from current state.

Hence $\epsilon\text{-closure}(q_0) = \{q_0, q_1, q_2\}$

$\epsilon\text{-closure}(q_1) = \{q_1, q_2\}$

$\epsilon\text{-closure}(q_2) = \{q_2\}$

$\delta'(q_0,0) = \epsilon\text{-closure}(\delta(\hat{\delta}(q_0,\epsilon),0))$

$= \epsilon\text{-closure}(\delta(\epsilon\text{-closure}(q_0),0))$

$= \epsilon\text{-closure}(\delta(q_0,q_1,q_2),0)$

$= \epsilon\text{-closure}(\delta(q_0,0) \cup \delta(q_1,0) \cup \delta(q_2,0))$

$= \epsilon\text{-closure}(q_0 \cup \phi \cup \phi)$

$= \epsilon\text{-closure}(q_0) = \{q_0, q_1, q_2\}$

$\delta'(q_1,0) = \epsilon\text{-closure}(\delta(\hat{\delta}(q_1,\epsilon),0))$

$= \epsilon\text{-closure}(\delta(q_1,q_2),0))$

$= \epsilon\text{-closure}(\delta(q_1,0) \cup \delta(q_2,0))$

$= \epsilon\text{-closure}(\phi \cup \phi) = \phi$

$\delta'(q_1,1) = \epsilon\text{-closure}(\delta(\hat{\delta}(q_1,\epsilon),1))$

$= \epsilon\text{-closure}(\delta(\epsilon\text{-closure}(q_1),1)$

$= \epsilon\text{-closure}(\delta(q_1,q_2),1)$

$= \epsilon\text{-closure}(\delta(q_1,1) \cup \delta(q_2,1)$

$= \epsilon\text{-closure}(q_1 \cup \phi) = \epsilon\text{-closure}(q_1) = \{q_1, q_2\}$

$\delta'(q_{10},1) = \epsilon\text{-closure}(\delta(\hat{\delta}(q_0,\epsilon),1))$

$= \epsilon\text{-closure}(\delta(q_{0,}q_1,q_2),1))$

$= \epsilon\text{-closure}(\delta(q_0,1) \cup \delta(q_1,1) \cup \delta(q_2,1))$

$= \epsilon\text{-closure}(\phi \cup q_1 \cup \phi) = \epsilon\text{-closure}(q_1) = \{q_1, q_2\}$

$$\delta'(q_2,0) = \epsilon\text{-closure}(\delta(\hat{\delta}(q_2,\epsilon),0))$$
$$= \epsilon\text{-closure}(\delta(\epsilon\text{-closure}(q_2),0)$$
$$= \epsilon\text{-closure}(\delta(q_2,0))$$
$$= \epsilon\text{-closure}(\phi) = \phi$$

$$\delta'(q_2,1) = \epsilon\text{-closure}(\delta(\delta'(q_2,\epsilon),1))$$
$$= \epsilon\text{-closure}(\delta(\epsilon\text{-closure}(q_2),1))$$
$$= \epsilon\text{-closure}(\delta(q_2,1)) = \epsilon\text{-closure}(\phi) = \phi$$

$$\delta'(q_0,2) = \epsilon\text{-closure}(\delta(\delta'(q_0,\epsilon),2))$$
$$= \epsilon\text{-closure}(\delta(\epsilon\text{-closure}(q_0),2)) = \epsilon\text{-closure}(\delta(q_0,q_1,q_2),2))$$
$$= \epsilon\text{-closure}(\delta(q_0,2) \cup \delta(q_1,2) \cup \delta(q_2,2))$$
$$= \epsilon\text{-closure}(\phi \cup \phi \cup q_2) = \epsilon\text{-closure}(q_2) = \{q_2\}$$

$$\delta'(q_1,2) = \epsilon\text{-closure}(\delta(\delta'(q_1,\epsilon),2)$$
$$= \epsilon\text{-closure}(\delta(\epsilon\text{-closure}(q_1),2))$$
$$= \epsilon\text{-closure}(\delta(q_1,q_2),2) = \epsilon\text{-closure}(\delta(q_1,2) \cup \delta(q_2,2))$$
$$= \epsilon\text{-closure}(\phi \cup q_2) = \epsilon\text{-closure}(q_2) = \{q_2\}$$

$$\delta'(q_2,2) = \epsilon\text{-closure}(\delta(\delta'(q_2,\epsilon),2))$$
$$= \epsilon\text{-closure}(\delta(\epsilon\text{-closure}(q_2),2))$$
$$= \epsilon\text{-closure}(\delta(q_2,2)) = \epsilon\text{-closure}(q_2) = \{q_2\}$$

Finally:

$\delta'(q_0,0) = \{q_0,q_1,q_2\}$
$\delta'(q_1,0) = \phi$
$\delta'(q_2,0) = \phi$

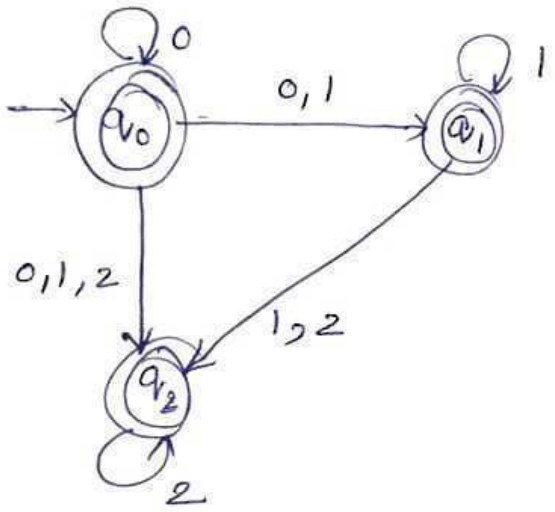
$\delta'(q_0,1) = \{q_1,q_2\}$
$\delta'(q_1,1) = \{q_1,q_2\}$
$\delta'(q_2,1) = \phi$

$\delta'(q_0,2) = \{q_2\}$
$\delta'(q_1,2) = \{q_2\}$
$\delta'(q_2,2) = \{q_2\}$

From this Transition table is

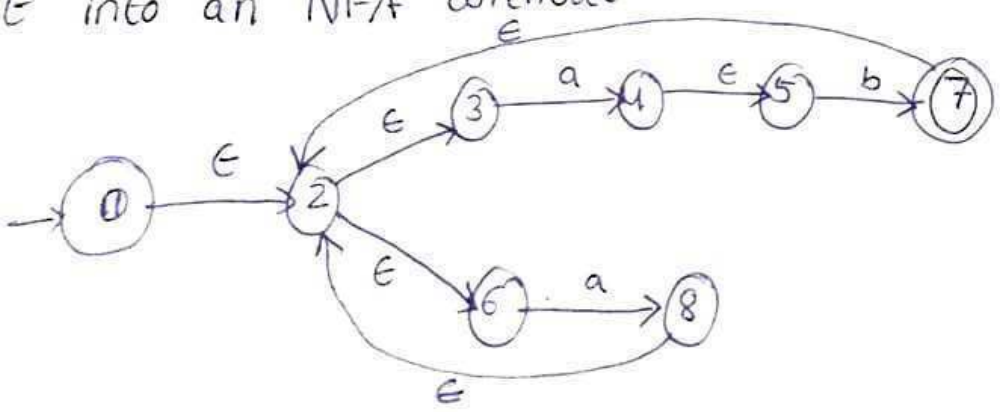
State / input	0	1	2
q_0	$\{q_0, q_1, q_2\}$	$\{q_1, q_2\}$	$\{q_2\}$
q_1	ϕ	$\{q_1, q_2\}$	$\{q_2\}$
q_2	ϕ	ϕ	$\{q_2\}$

NFA will be



Hence q_0, q_1, q_2 is a final state because ϵ-closures of q_0, q_1, q_2 contains final state.

Example (2) convert the following NFA with ϵ moves convert it into an NFA without ϵ-moves.



Solution : we will first find out ∈-closure of states

∈-closure (1) = {1, 2, 3, 6}

∈-closure (2) = {2, 3, 6}

∈-closure (3) = {3}

∈-closure (4) = {4, 5}

∈-closure (5) = {5}

∈-closure (6) = {6}

∈-closure (7) = {2, 3, 6, 7}

∈-closure (8) = {2, 3, 6, 8}

Now apply input transitions on states.

$\delta'(1,a)$ = ∈-closure (δ (δ'(1, ∈), a))

= ∈-closure(δ (∈-closure(1), a))

= ∈-closure(δ(1, 2, 3, 6), a))

= ∈-closure(δ(1,a) ∪ δ(2,a) ∪ δ(3,a) ∪ δ(6,a))

= ∈-closure (4, 8) = {2, 3, 4, 5, 6, 8}

$\delta'(1,b)$ = ∈-closure (δ(δ'(1, ∈), b))

= ∈-closure(δ (1, 2, 3, 6), b)

= ∈-closure (δ(1,b) ∪ δ(2,b) ∪ δ(3,b) ∪ δ(6,b))

= ∈-closure (∅) = ∅

$\delta'(2,a)$ = {2, 3, 4, 5, 6, 8}

$\delta'(2,b)$ = {∅}

$\delta'(3,a)$ = {4, 5}

$\delta'(3,b)$ = ∅

$\delta'(4,a)$ = ∅

$\delta'(4,b)$ = {2, 3, 6, 7}

$\delta'(5,a)$ = ∅

$\delta'(5,b)$ = {2, 3, 6, 7}

$\delta'(6,a)$ = {2, 3, 6, 8}

$\delta'(6,b)$ = ∅

$\delta'(7,a)$ = {2, 3, 4, 5, 6, 8}

$\delta'(7,b)$ = ∅

$\delta'(8,a)$ = {2, 3, 4, 5, 6, 8}

$\delta'(8,b)$ = ∅

Transition table is

state \ input	a	b
1	{2,3,4,5,6,8}	ϕ
2	{2,3,4,5,6,8}	ϕ
3	{4,5}	ϕ
4	ϕ	{2,3,6,7}
5	ϕ	{2,3,6,7}
6	{2,3,6,8}	ϕ
7	{2,3,4,5,6,8}	ϕ
8	{2,3,4,5,6,8}	ϕ

Conversion from NFA to DFA:

Let $M = (Q, \Sigma, \delta, q_0, F)$ is a NFA which accepts the language $L(M)$. There should be equivalent DFA denoted by $M' = (Q', \Sigma', \delta', q_0', F')$ such that $L(M) = L(M')$.

The conversion method will follow following steps:

① The start state of NFA M will be the start for DFA M'. Hence add q_0 of NFA to Q'. Then find the transitions from this start state.

② For each state $[q_1, q_2, q_3, \ldots q_i]$ in Q' the transitions for each input symbol can be obtained as,

(i) $\delta'[q_1, q_2, q_3, \ldots q_i], a) = \delta(q_1, a) \cup \delta(q_2, a) \cup \delta(q_3, a) \ldots \cup \delta(q_i, a)$

$= [q_1, q_2, \ldots q_k]$ may be some state.

(ii) Add the state $[q_1, q_2, \ldots q_k]$ to DFA if it is not already added in Q'.

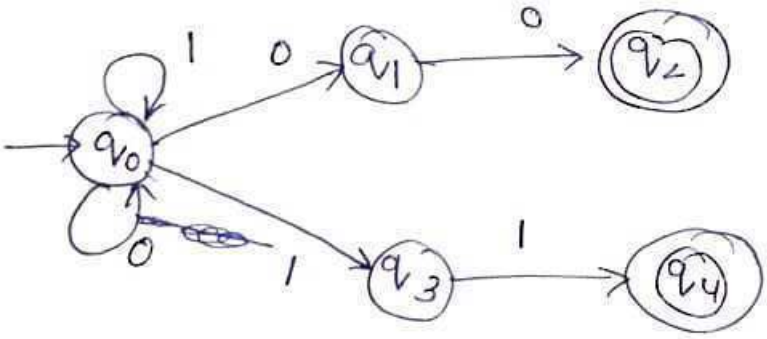
(iii) Then find the transitions for every input symbol from Σ for state $[q_1, q_2, \ldots q_k]$. If we get some state $[q_1, q_2, \ldots q_n]$ which is not in Q' of DFA then add this state to Q'.

(iv) If there is no new state generating then stop the process after finding all the transitions.

3) For the state $[q_1, q_2, \ldots q_n] \in Q'$ of DFA if any one state q_i is a final state of NFA then $[q_1, q_2, \ldots q_n]$ becomes final state. Thus the set of all the final states $\in F'$ of DFA.

Example:

1. Convert the give NFA to its equivalent DFA.



Solution we will first design transition table from given transition diagram.

state \ input	0	1
→ q_0	$\{q_0, q_1\}$	$\{q_0, q_3\}$
q_1	$\{q_2\}$	ϕ
* q_2	ϕ	ϕ
q_3	ϕ	$\{q_4\}$
* q_4	ϕ	ϕ

→ As we got a new state $[q_0, q_1]$ we will compute δ' transitions for input 0 & 1.

$$\delta'([q_0,q_1],0) = \delta(q_0,0) \cup \delta(q_1,0)$$
$$= \{q_0,q_1\} \cup \{q_2\}$$
$$= \{q_0,q_1,q_2\}$$

$$\delta'([q_0,q_1],1) = \delta(q_0,1) \cup \delta(q_1,1)$$
$$= \{q_0,q_3\} \cup \{\phi\}$$
$$= \{q_0,q_3\}$$

→ The new state $[q_0, q_3]$ is obtained, we will process it

$$\delta'([q_0,q_3],0) = \delta(q_0,0) \cup \delta(q_3,0)$$
$$= \{q_0,q_1\} \cup \{\phi\}$$
$$= \{q_0,q_1\}$$

$$\delta'([q_0,q_3],1) = \delta(q_0,1) \cup \delta(q_3,1)$$
$$= \{q_0,q_3\} \cup \{q_4\} = \{q_0,q_3,q_4\}$$

→ Now we process newly generated states

$[q_0, q_1, q_2]$ & $[q_0, q_3, q_4]$

$\delta'([q_0, q_1, q_2], 0) = \delta(q_0, 0) \cup \delta(q_1, 0) \cup \delta(q_2, 0)$

$= \{q_0, q_1, q_2\}$

$\delta'([q_0, q_1, q_2], 1) = \delta(q_0, 1) \cup \delta(q_1, 1) \cup \delta(q_2, 1)$

$= \{q_0, q_3\}$

$\delta'([q_0, q_3, q_4], 0) = \{q_0, q_1\}$

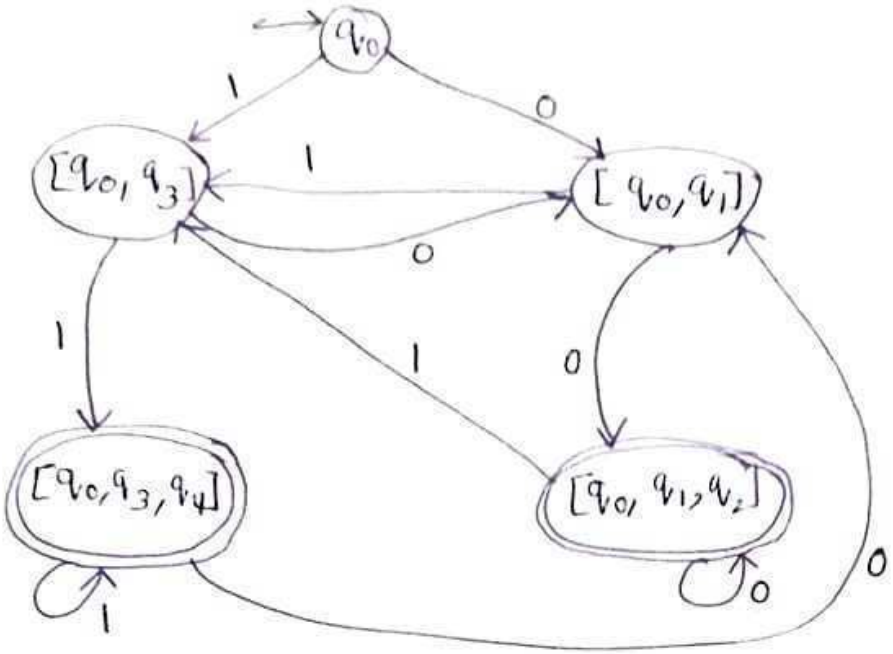
$\delta'([q_0, q_3, q_4], 1) = \{q_0, q_3, q_4\}$

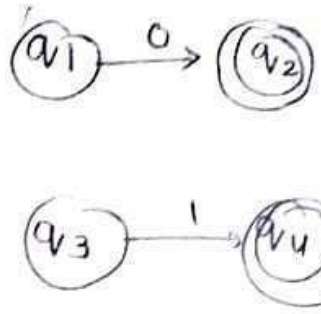
⇒ There are no further new states

Transition table is

state \ input	0	1
→ q_0	$\{q_0, q_1\}$	$\{q_0, q_3\}$
q_1	$\{q_2\}$	ϕ
*q_2	ϕ	ϕ
q_3	ϕ	$\{q_4\}$
*q_4	ϕ	ϕ
$[q_0, q_1]$	$\{q_0, q_1, q_2\}$	$\{q_0, q_3\}$
$[q_0, q_3]$	$\{q_0, q_1\}$	$\{q_0, q_3, q_4\}$
* $[q_0, q_1, q_2]$	$\{q_0, q_1, q_2\}$	$\{q_0, q_3\}$
* $[q_0, q_3, q_4]$	$\{q_0, q_1\}$	$\{q_0, q_3, q_4\}$

The DFA can be drawn as





Not connected part.

→ Finally Above is the final DFA.

Example:

(2) Let $M = (\{q_0, q_1\}, \{0,1\}, \delta, q_0, \{q_1\})$ be NFA where $\delta(q_0,0) = \{q_0, q_1\}$, $\delta(q_0,1) = \{q_1\}$ $\delta(q_1,0) = \emptyset$ & $\delta(q_1,1) = \{q_0, q_1\}$ construct its equivalent DFA.

Solution: Let the DFA $M' = (Q', \Sigma, \delta', q_0', F')$

Now the δ' function can be computed as

$$\delta(q_0,0) = \{q_0, q_1\}$$

$$\delta'([q_0], 0) = [q_0, q_1].$$

As in NFA the initial state is q_0, the DFA will also contain the initial state $[q_0]$.

let us draw the transition table for δ function for given NFA.

state \ input	0	1
→ q_0	$\{q_0, q_1\}$	$\{q_1\}$
(q_1)	ϕ	$\{q_0, q_1\}$

δ function for NFA

From the transition table we can compute that there are $[q_0]$, $[q_1]$, $[q_0, q_1]$ states for equivalent DFA. we need to compute the transition from state $[q_0, q_1]$

$$\delta'([q_0,q_1],0) = \delta(q_0,0) \cup \delta(q_1,0)$$
$$= \{q_0,q_1\} \cup \phi$$
$$= \{q_0,q_1\}$$

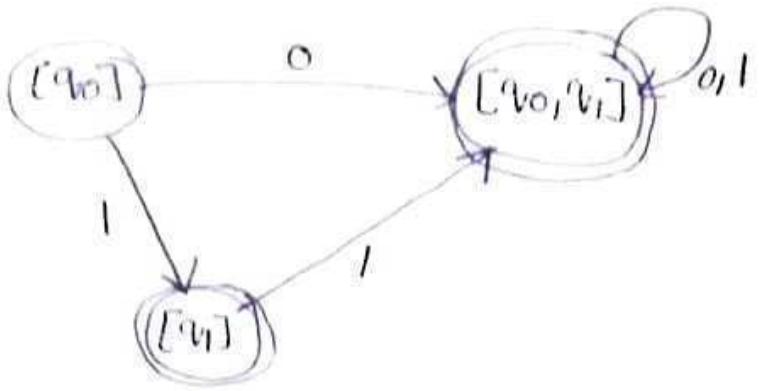
$$\delta'([q_0,q_1],1) = \delta(q_0,1) \cup \delta(q_1,1)$$
$$= \{q_1\} \cup \{q_0,q_1\}$$
$$= \{q_0,q_1\}$$

As in given NFA q_1 is the final state, then in DFA wherever q_1 exists that state becomes a final state. Hence in DFA final states are $[q_1]$ & $[q_0\ q_1]$. Therefore set of final states are $F = \{[q_1], [q_0,q_1]\}$

Equivalent DFA is

state \ input	0	1
→ q_0	$[q_0,q_1]$	$[q_1]$
* $[q_1]$	ϕ	$[q_0,q_1]$
* $[q_0,q_1]$	$[q_0,q_1]$	$[q_0,q_1]$

Transition diagram for DFA is



Conversion of NFA with ∈ to DFA:

① Consider $M = (Q, \Sigma, \delta, q_0, F)$ is NFA with ∈. we have to convert this NFA with ∈ to equivalent DFA denoted by

$$M_D = (Q_D, \Sigma, \delta_D, q_0, F_D),$$

Then obtain

$\in$-closure$(q_0) = \{P_1, P_2, \cdots P_n\}$ then $[P_1, P_2, \cdots P_n]$ becomes a start state of DFA.

Now $[P_1, P_2, P_3 \cdots P_n] \in Q_D$

② we will obtain δ transitions on $[P_1, P_2, P_3 \cdots P_n]$ for each input.

$$\delta_D([P_1, P_2, \cdots P_n], a) = \in\text{-closure}(\delta[P_1, a) \cup \delta(P_2, a) \cup \cdots \delta(P_n, a))$$

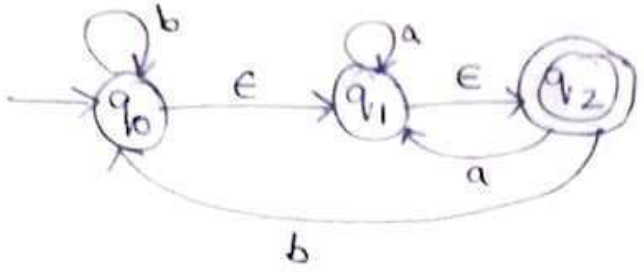
$$= \bigcup_{i=1}^{n} \in\text{-closure}(P_i, a)$$

where a is input $\in \Sigma$.

③ The states obtained $[P_1, P_2, \cdots P_n] \in Q_D$. The states containing final states in P_i is a final state in DFA.

Example

① Convert the following NFA with ϵ to DFA



Solution:

To convert this NFA we will first find ϵ-closures

ϵ-closure$(q_0) = \{q_0, q_1, q_2\}$

ϵ-closure$(q_1) = \{q_1, q_2\}$

ϵ-closure$(q_2) = \{q_2\}$

Now let us start from ϵ-closure of start state.

ϵ-closure$\{q_0\} = \{q_0, q_1, q_2\}$ we will call this state as A

Now let us find transitions on A with every input symbol

$\delta'(A, a) = \epsilon$-closure$(\delta(A, a))$

$= \epsilon$-closure$(\delta(q_0, q_1, q_2\}, a)$

$= \epsilon$-closure$(\delta(q_0, a) \cup \delta(q_1, a) \cup \delta(q_2, a))$

$= \epsilon$-closure$(\phi \cup q_1 \cup q_1)$

$= \epsilon$-closure$(q_1) = \{q_1, q_2\}$. let us call it as state B.

$\delta'(A, b) = \epsilon$-closure$(\delta(A, b))$

$= \epsilon$-closure$(\delta(q_0, q_1, q_2), b)$

$= \epsilon$-closure$(\delta(q_0, b) \cup \delta(q_1, b) \cup \delta(q_2, b))$

$= \epsilon$-closure$(q_0 \cup \phi \cup q_0)$

$= \epsilon$-closure$(q_0) = \{q_0, q_1, q_2\}$ i.e., A

Hence we can state that

$\delta'(A, a) = B$

$\delta'(A, b) = A$

Now let us find transitions for state $B - \{q_1, q_2\}$

$\delta'(B, a) = \delta(\epsilon\text{-closure}(\delta(q_1, q_2), a)$

$= \epsilon\text{-closure}(\delta(q_1, a) \cup \delta(q_2, a))$

$= \epsilon\text{-closure}(q_1 \cup q_1) = \epsilon\text{-closure}(q_1)$

$= \{q_1, q_2\}$ ie., B

$\delta'(B, b) = \epsilon\text{-closure}(\delta(q_1, q_2), b)$

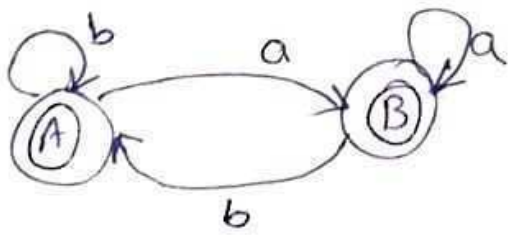
$= \epsilon\text{-closure}(\delta(q_1, b) \cup \delta(q_2, b))$

$= \epsilon\text{-closure}(\phi \cup q_0)$

$= \epsilon\text{-closure}(q_0) = \{q_0, q_1, q_2\}$ ie., A

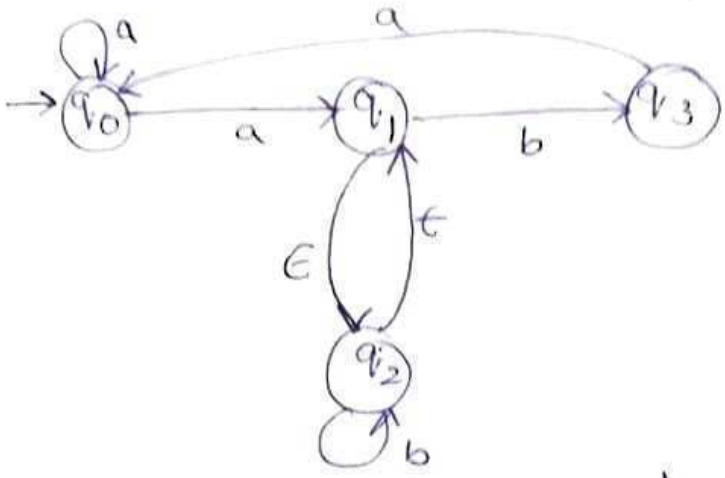
Hence the generated DFA is

	a	b
→ (A)	B	A
(B)	B	A



Example 2

Construct DFA for the following NFA - ε



Solution: we will first write ε-closure of start state q_0.

ε-closure $(q_0) = \{q_0\}$ call it as state A

Now obtain δ transitions for input a & b for state A

$\delta'(A,a)$ = ε-closure $(\delta(A,a))$

= ε-closure $(\delta(q_0,a))$

= ε-closure (q_0, q_1)

= ε-closure $(q_0) \cup$ ε-closure (q_1)

= $\{q_0\} \cup \{q_1, q_2\}$

= $\{q_0, q_1, q_2\}$ — call it as state B.

$\delta'(A,b)$ = ε-closure $(\delta(A,b))$

= ε-closure $(\delta(q_0,b))$

= ε-closure $(\phi) = \phi$

Now consider state B ie, $\{q_0, q_1, q_2\}$

$\delta'(B,a)$ = ε-closure $(\delta(B,a))$

= ε-closure $(\delta(q_0,q_1,q_2),a)$

= ε-closure $(\delta(q_0,a) \cup \delta(q_1,a) \cup \delta(q_2,a))$

$= \epsilon\text{-closure}(\{q_0, q_1\} \cup \phi \cup \phi)$

$= \epsilon\text{-closure}(q_0, q_1) = \epsilon\text{-closure}(q_0) \cup \epsilon\text{-closure}(q_1)$

$= \{q_0, q_1, q_2\}$ i.e., state B.

$\delta'(B,b) = \epsilon\text{-closure}(\delta(B,b))$

$= \epsilon\text{-closure}(\delta(q_0, q_1, q_2), b))$

$= \epsilon\text{-closure}(\delta(q_0,b) \cup \delta(q_1,b) \cup \delta(q_2,b))$

$= \epsilon\text{-closure}(\phi \cup q_3 \cup q_2)$

$= \epsilon\text{-closure}(q_2, q_3) = \epsilon\text{-closure}(q_2) \cup \epsilon\text{-closure}(q_3)$

$= \{q_1, q_2\} \cup \{q_3\}$

$= \{q_1, q_2, q_3\}$ — call is as state C

The δ transitions on C are obtained as:

$\delta'(C,a) = \epsilon\text{-closure}(\delta(C,a))$

$= \epsilon\text{-closure}(\delta(q_1, q_2, q_3), a))$

$= \epsilon\text{-closure}(\delta(q_1,a) \cup \delta(q_2,a) \cup \delta(q_3,a))$

$= \epsilon\text{-closure}(\phi \cup \phi \cup q_0)$

$= \epsilon\text{-closure}(q_0) = q_0$ i.e., A

$\delta'(C,b) = \epsilon\text{-closure}(\delta(C,b))$

$= \epsilon\text{-closure}(\delta(q_1, q_2, q_3), b))$

$= \epsilon\text{-closure}(\delta(q_1,b) \cup \delta(q_2,b) \cup \delta(q_3,b))$

$= \epsilon\text{-closure}(q_3 \cup q_2 \cup \phi)$

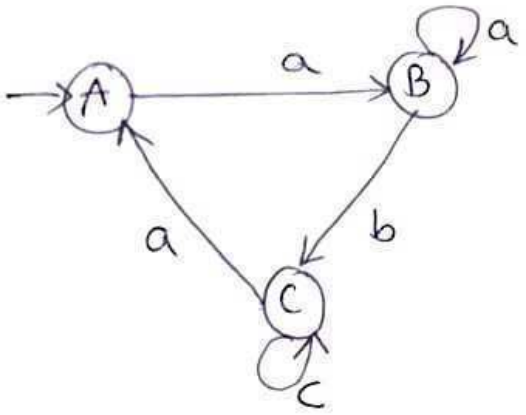
$= \epsilon\text{-closure}(q_2, q_3) = \epsilon\text{-closure}(q_2) \cup \epsilon\text{-closure}(q_3)$

$= \{q_1, q_2\} \cup \{q_3\}$

$= \{q_1, q_2, q_3\}$ i.e., state C

Now no new state is generated. Hence we will write transition table for above computations.

state \ inputs	a	b
→ A	B	∅
B	B	C
C	A	C



Finite Automata with output:

There are two types of FA with output and those are:

1. Moore machine.
2. Mealy machine.

Moore machine:

→ Moore machine is a finite state machine in which the next state is decided by current state and current input symbol.

→ The output symbol at a given time depends only on the present state of the machine.

Moore machine is a six tuple representation $(Q, \Sigma, \Delta, \delta, \lambda, q_0)$ where

Q — finite set of states

Σ — finite set of input symbols

Δ — an output alphabet

δ — Transition function such that $Q \times \Sigma \rightarrow Q$

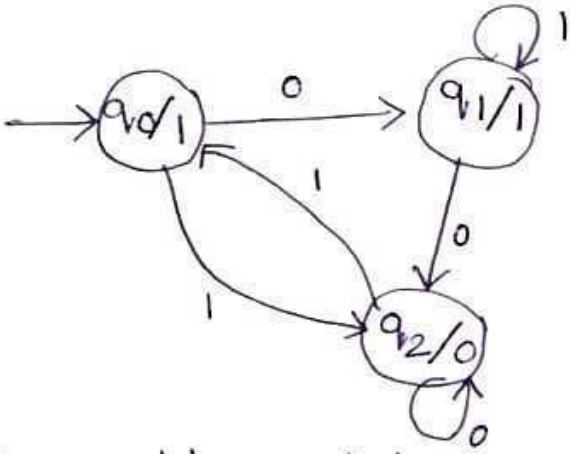
λ — output function $Q \rightarrow \Delta$ (also known as machine function).

q_0 — initial state of machine.

→ In moore machine output is associated with every state. If the length of the input string is 'n' then output string has length 'n+1'.

Example:

Consider the moore machine given below



Transition table will be

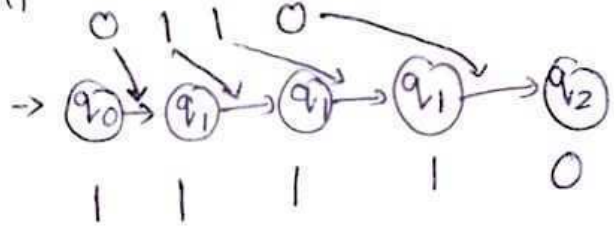
Current state	Next state (δ)		Output (λ)
	0	1	
→q_0	q_1	q_2	1
q_1	q_2	q_1	1
q_2	q_2	q_0	0

→ In moore machine output is associated with every state.

→ In the above given moore machine when machine is in q_0 state the output will be 1.

For the string 0110 then output will be 11110 ie.,

Acceptance of string is



= 11110

Mealy machine:

Mealy machine is a machine in which output symbol depends upon the present input symbol and present state of the machine. The mealy machine can be defined as a six tuple notation

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

Q - finite set of states.

Σ - finite set of input symbols.

Δ - an output alphabet

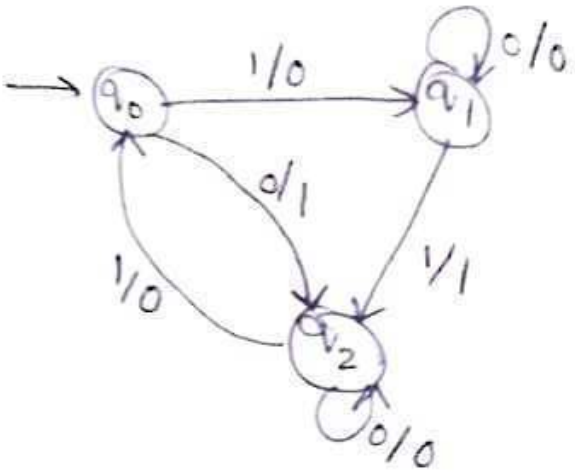
δ - state transition function such that $Q \times \Sigma \to Q$

λ - machine function such that $Q \times \Sigma \to \Delta$

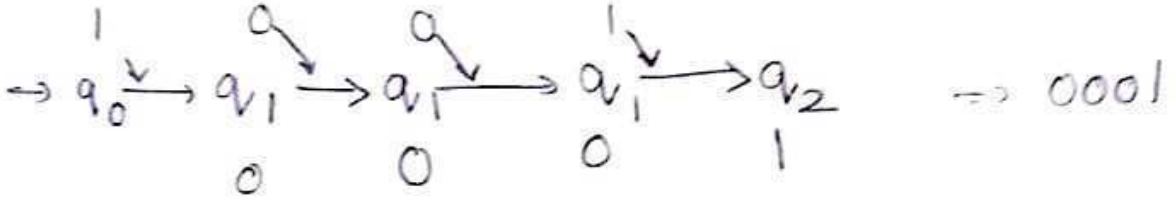
q_0 - initial state.

→ In mealy machine the length of input string is equal to length of output string.

For example:



For input string 1001



Conversion of moore to mealy:

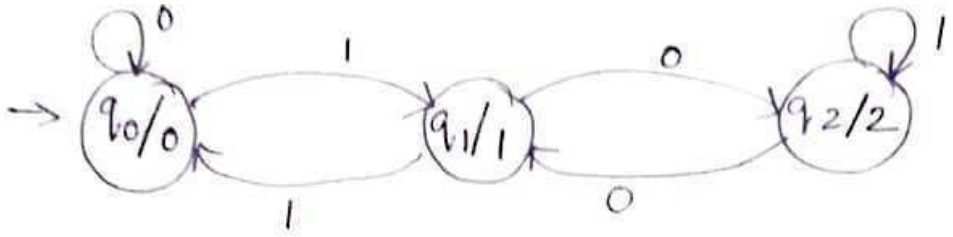
Let $M = (Q, \Sigma, \delta, \lambda, q_0)$ be moore m/c, its equivalent mealy machine can be represented as $M' = (Q, \Sigma, \delta, \lambda', q_0)$.

The output function $\lambda'(q, a) = \lambda(\delta(q, a))$

Example: convert the following moore m/c into its equivalent mealy m/c.

q_i \ Σ	0	1	output (λ)
q_0	q_0	q_1	0
q_1	q_2	q_0	1
q_2	q_1	q_2	2

Solution



The output function λ' can be obtained using following rule.

$$\lambda'(q_i, a) = \lambda(\delta(q_i, a))$$

Here we will obtain output for every transition corresponding to input symbol

$\lambda'(q_0, 0) = \lambda(\delta(q_0, 0))$
$= \lambda(q_0)$ ie., output of q_0
$= 0$

$\lambda'(q_0, 1) = \lambda(\delta(q_0, 1))$
$= \lambda(q_1)$
$= 1$

$\lambda'(q_1, 0) = \lambda(\delta(q_1, 0))$
$= \lambda(q_2)$
$= 2$

$\lambda'(q_1, 1) = \lambda(\delta(q_1, 1))$
$= \lambda(q_0)$
$= 0$

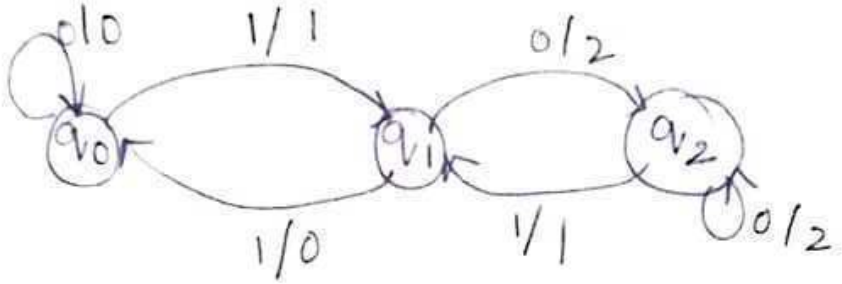
$\lambda'(q_2, 0) = \lambda(\delta(q_2, 0))$
$= \lambda(q_1)$
$= 1$

$\lambda'(q_2, 1) = \lambda(\delta(q_2, 1))$
$= \lambda(q_2)$
$= 2$

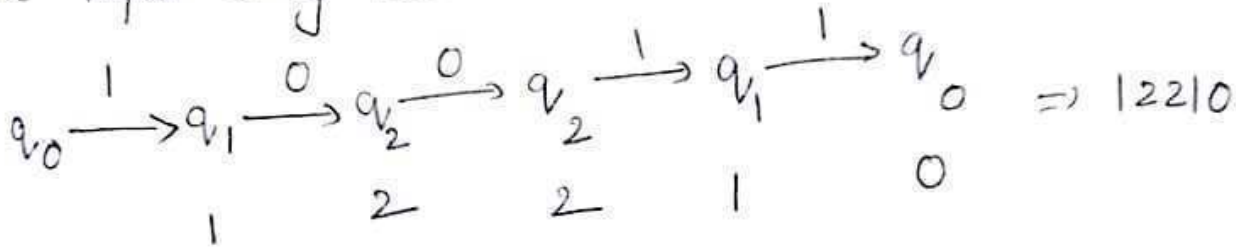
Hence transition table for mealy machine is

Q \ Σ	input = 0		input = 1	
	state	output	state	output
q_0	q_0	0	q_1	1
q_1	q_2	2	q_0	0
q_2	q_1	1	q_2	2

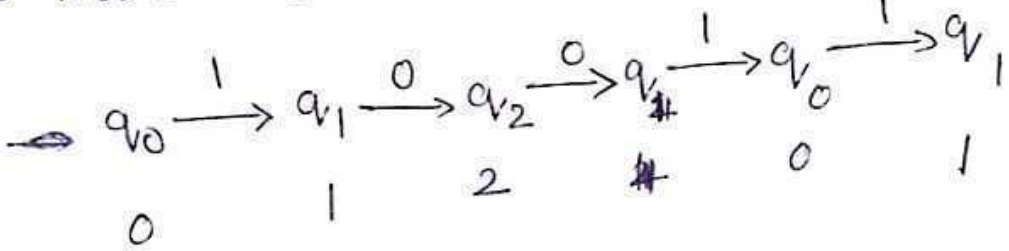
Transition diagram of mealy machine is



The input string 10011 then output for mealy m/c is



For moore m/c



conversion of mealy to moore :

step1 : If transition diagram is given, then convert the transition diagram to transition table.

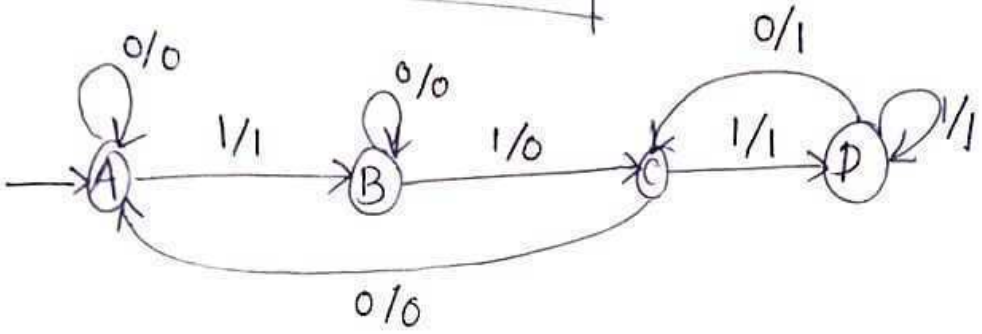
step2: From transition table of mealy machine, find out such kind of states which are showing more than 1 outputs

step3 Then in such cases, divide those states in to more states depending on the outputs associated with them.

Ex Convert the following mealy machine in to its equivalent moore machine. (16)

mealy Transition table.

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



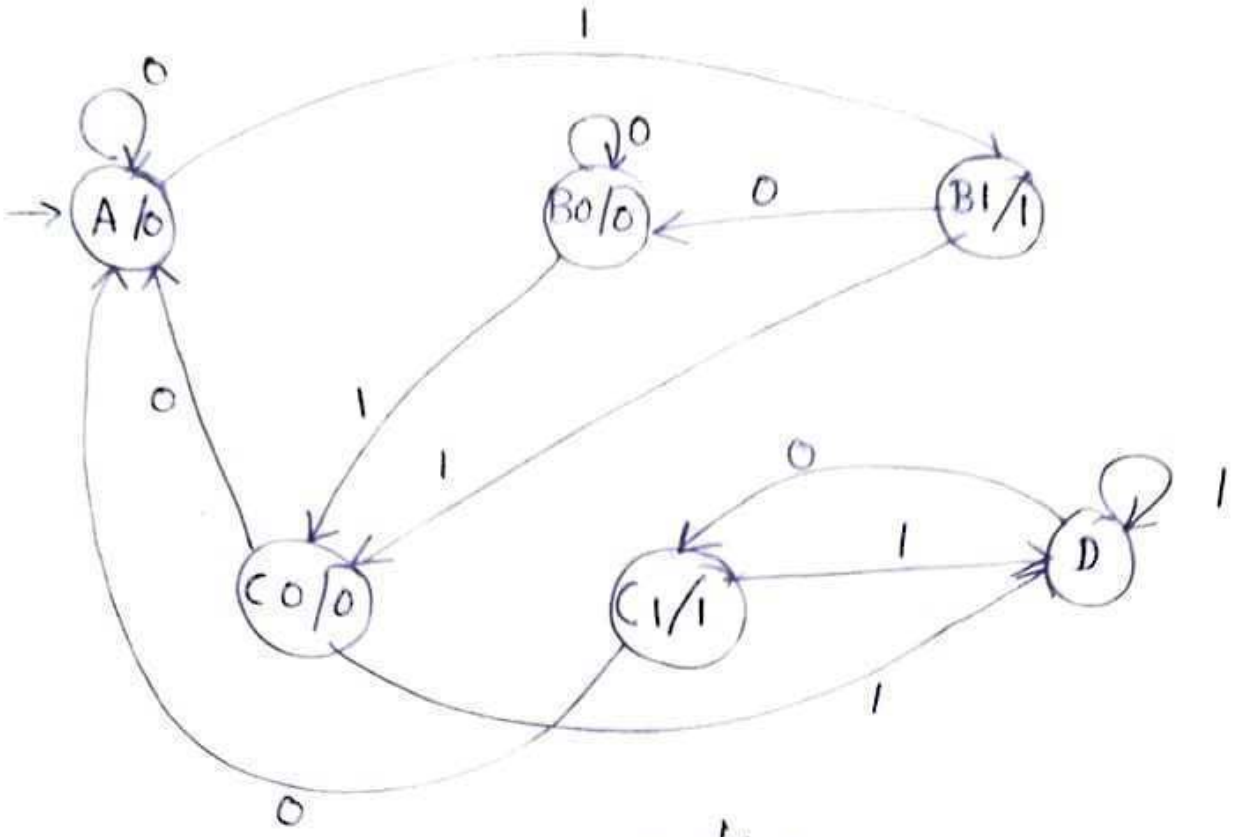
mealy states A B C D

outputs 0 | 0 1 | 0 1 | 1

moore states A B_0 B_1 C_0 C_1 D_1

moore transition table

states	output		output
	0	1	
→ A	A	B_1	0
B_0	B_0	C_0	0
B_1	B_0	C_0	1
C_0	A	D	0
C_1	A	D	1
D	C_1	D	1



moore machine.

UNIT - II (FLAT)

Finite Automata and Regular Expressions :

Regular expression :

→ The language accepted by FA can be easily described by simple expression called Regular Expressions.

→ It is most effective way to represent any language.

→ Regular expressions are used to match character combination in strings.

→ string searching algorithm used this pattern to find the operations on a string.

→ contains three operators

1. $\emptyset$ is a regular expression which denotes the empty set.
2. $\{\epsilon\}$ is a regular expression denotes null string.
3. For each a in Σ 'a' is a regular expression and denotes the set $\{a\}$.
4. If 'r' and 's' are regular expressions denoting L1 & L2 then $r+s$

Applications of Regular Expressions:

- operating systems (unix)
- compilation (lexical Analysis)
- programming languages (Java)
- finding patterns in text.

Regular expressions in unix:

→ Basically Regular expressions are used in search tools.
→ Text file search in unix (tool : egrep).
→ The egrep command searches for a text pattern in the file and list the words containing the pattern.

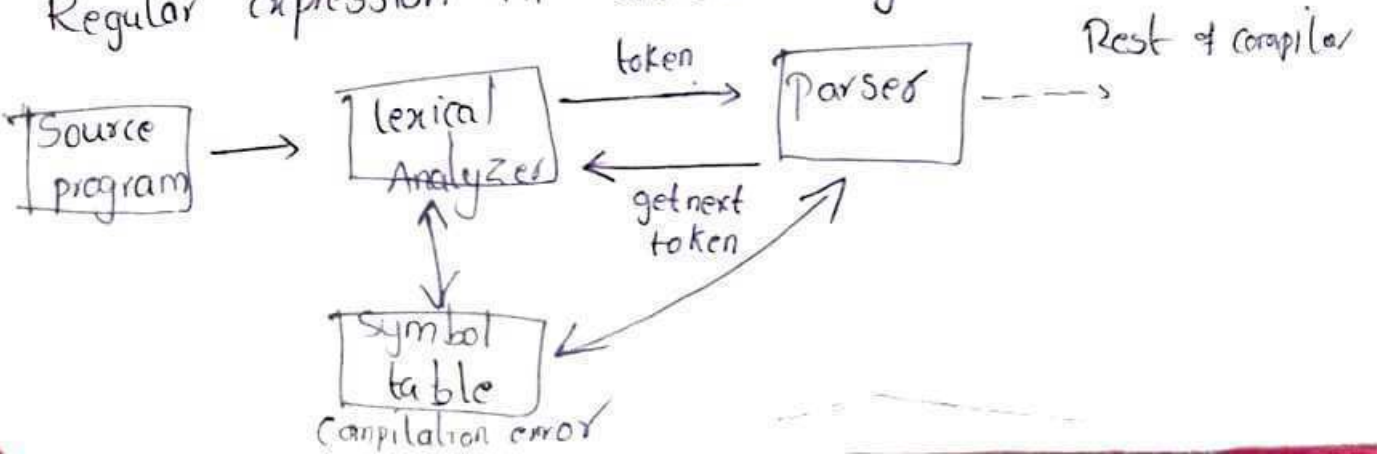
Example: File : xyz
sam
Dexter
John
Raman

command: */* egrep 'n' xyz → searching for 'n' in the contents of the file.

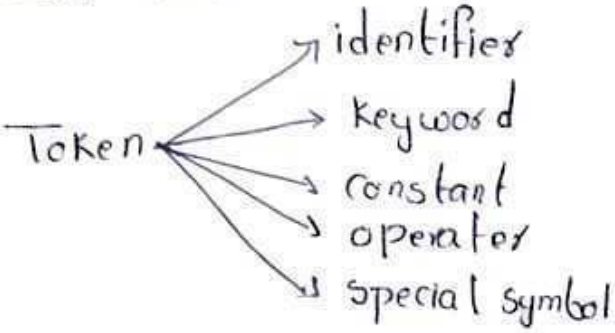
O/P: John
Raman.

Compilation:

Regular expression in lexical Analysis:



→ Lexical Analyzer takes the source program contents and converts them into tokens. Tokens are the individual units.



→ The role of the regular expressions in the lexical Analyzer is it specifies the patterns of the tokens.

Example: identifier

$$([A-Z]|[a-Z])([A-Z]|[a-z]|[0-9])^*$$

In most of the programming languages the identifier should start with alphabet and then it can contain the combination of alphabet & digit or ϵ.

Programming languages (Java):

Regular Expressions are a language of string patterns built in to most moder programming languages including Java 1.4 on wards used for searching, extracting and modifying text.

Java.util.regex contains classes for working with regular expressions in Java.

Finding Patterns in Text:

By using regular expressions notation, it becomes easy to describe the patterns at high level.

→ A "compiler" for R.E is useful to turn the regular expressions in to executable code.

Example: "find me a restaurant within 10 minutes drive of where I am now"

ex focus on recognizing street addresses in particular. Some people live in "Avenues" or "Roads" or "streets"

Thus we might use R.E like

Street | st\. | Avenue | Ave\. | Road | Rd\.

Above expression is unix style notation with vertical bar rather than +, as union operator.

[0-9]+ [A-Z]? [A-Z][a-z]* ([A-Z][a-z]*)* (Street|st\.|Avenue|Ave\.|Road|Rd\.)

. → Any characters ex: "123A Main st"

? → zero or one characters

+ → one or more digits

Algebraic laws for Regular Expressions:

Like arithmetic expressions, the R.E have a number of laws that work for them. Union as addition & concatenation as multiplication.

Associativity & Commutativity: → switch the order of operands → regroup the operands when operator is applied twice.

→ L+M = M+L (commutative law for union)

→ (L+M)+N = L+(M+N) (Associative law for union)

→ (LM)N = L(MN) (Associative law for concatenation)

Identities & Annihilators:

→ An identity for an operator is a value such that when the operator is applied to identity and some other value, the result is other value.

→ An annihilator for an operator is a value such that when the operator is applied to the annihilator & some other value, the result is annihilator.

⇒ $\emptyset + L = L + \emptyset = L$ ($\emptyset$ is identity for union)

⇒ $\epsilon L = L\epsilon = L$ (ϵ is identity for concatenation)

⇒ $\emptyset L = L\emptyset = \emptyset$ ($\emptyset$ is annihilator for concatenation)

Distributive Laws:

⇒ $L(M+N) = LM + LN$ (left distributive law of concatenation)

→ $(M+N)L = ML + NL$ (right distributive law of concatenation)

Idempotent law:

→ The result of applying it to two of the same values as arguments is that value is called as idempotent law.

* $L + L = L$ (idempotent law for union)

Laws involving closure:

① $(L^*)^* = L^*$

② $\emptyset^* = \epsilon$

③ $\epsilon^* = \epsilon$

④ $L^+ = LL^* = L^*L$

⑤ $L^* = L^+ + \epsilon$

⑥ $L^? = \epsilon + L$

⑦ $(L+M)^* = (L^*M^*)^* = (L^* + M^*)^*$

⑧ $L + LL^* = LL^* + + L = L^*$

⑨ $(PQ)^*P = P(PQ)^*$

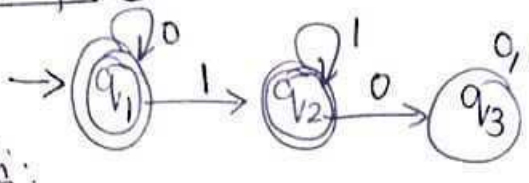
⑩ $(P+Q)R = PR + QR$

⑪ $R(P+Q) = RP + RQ$.

Conversion of FA to Regular expression

Arden theorem

Example: (1) Construct RE for the given DFA



Soln:

Equation for $q_1 = q_1 0 + \epsilon$

$q_2 = q_1 1 + q_2 1$

$q_3 = q_2 0 + q_3(0+1)$

we solve q_1 & q_2 as they are final states

$q_1 = q_1 0 + \epsilon$ $(R = Q + RP)$

$R = q_1$, $Q = \epsilon$, $P = 0$ $(R = QP^*)$

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

$\boxed{q_1 = 0^*}$

$q_2 = q_1 1 + q_2 1$

$q_2 = 0^* 1 + q_2 1$

$R = Q + RP$

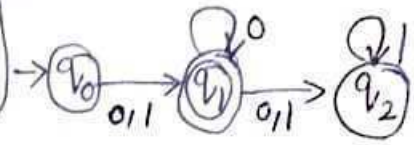
$R = q_2$ $Q = 0^*1$ $P = 1$

$R = QP^*$

$q_2 = 0^* 1 1^*$ $(RR^* = R^+)$

$\boxed{q_2 = 0^* 1^+}$

(2) Find R.E for the NFA



Solution: equation for

$q_0 = \epsilon$

$q_1 = q_0(0+1) + q_1 0$

$q_2 = q_1(0+1) + q_2 1$

As q_1 is final state, we solve for q_1

$q_1 = q_0(0+1) + q_1 0$

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

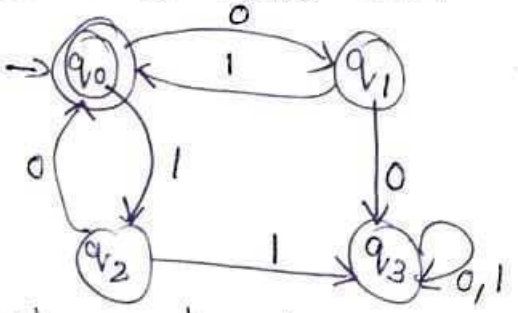
$R = q_1$ $Q = \epsilon(0+1)$ $P = 0$

$R = QP^*$

$q_1 = \epsilon(0+1)0^*$ $[\epsilon R = R]$

$\boxed{q_1 = (0+1)0^*}$

(3) construct R.E for DFA



Solution: Equations for

$q_0 = q_1 1 + q_2 0 + \epsilon$

$q_1 = q_0 0$ $q_2 = q_0 1$

$q_3 = q_1 0 + q_2 1 + q_3(0+1)$

← As q_0 is final state solve q_0

$q_0 = q_1 1 + q_2 0 + \epsilon$

$= q_0 01 + q_0 10 + \epsilon$

$= q_0(01+10) + \epsilon$ $[R = QP^*]$

$\boxed{q_0 = \epsilon(01+10)^*}$

Pumping lemma for Regular languages:

→ Pumping lemma is a powerful technique used for proving certain languages are not regular.

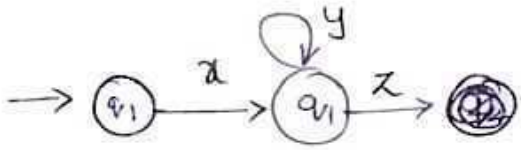
Theorem:

- Let L be a regular language → for a DFA
- Then there exists a constant 'n' → number of states of DFA such that for every string w in L, $|w| \geq n$

we can break w in to 3 strings, $w = xyz$ such that

1. $y \neq \epsilon$ (or) $|y| > 0$
2. $|xy| \leq n$
3. For all $K \geq 0$, the string xy^Kz is also in L.

Example: $n = 3$ $\quad w = abc$



Prove that $L = \{0^n 1^n \mid n \geq 1\}$ is not regular.

→ Assume L is regular.

- Let n be a constant.

Let $w = 0^n 1^n$ $\quad |w| \geq n$

Split $w = xyz$ such that

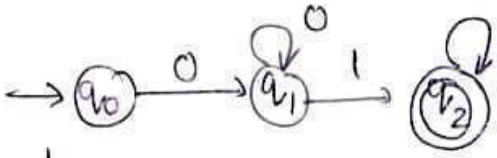
1. $y \neq \epsilon$
2. $|xy| \leq n$
3. For all $K \geq 0$, $xy^Kz \in L$

For given language the strings are 01, 0011, 000111, ...

- All strings having equal number of 1's followed by equal number of 1's.

Finite automata has limited memory, it cannot remember whether it read 'n' no. of zeros or 'm' no. of zeros

- consider DFA for given string



- consider string three zero's, after reading these the state q_1 is reached, now starts reading 1, but it cannot remember whether it has read 3, or 4, or 5 zero's. It cannot match number of zero's and 1's.

- Above DFA is acceptable for language

$$L = \{0^n 1^m \mid n \geq 1, m \geq 1\}$$

But the above DFA is not acceptable for language $L = \{0^n 1^n \mid n \geq 1\}$. For this language we cannot construct DFA. To prove this formally we need pumping lemma.

- Consider the string $w = 0^n 1^n$ & $n = 2$ then $w = 0011$.

- split $w = xyz$ $\quad w = \underline{00}11$

$xy = 00$, $y = 0$, $x = 0$ and $z = 11$

Assume $k = 2$ then $xy^2z = 00011 \notin L$ as no. of 0's is not equal to no. of 1's.

Hence $L = \{0^n 1^n \mid n \geq 1\}$ is not regular

Example: 2 show that $L = \{ b^{i^2} \mid i > 1 \}$ is not regular.

Proof: Assume that b^{i^2} is a regular language.

b^{i^2} length of the string $|w| = n^2$

Consider the pumping length is n

$n^2 > n$

Regular language for $L = \{ b^{i^2} \mid i > 1 \}$ is $\{ b^{2^2}, b^{3^2}, b^{4^2}, \ldots \}$

$= \{ b^4, b^9, b^{16}, \ldots \} = \{ bbbb, bbbbbbbbb, \text{---} \}$

condition 1: for $i \geq 0$, $xy^iz \in L$

$\underset{x}{b}\,\underset{y}{bbb}\,\underset{z}{b}$

$x = b$, $y = bb$, $z = b$

Pumping y take $i = 0$, $(bb)^0 = \phi \Rightarrow bb$

take $i = 1$, $(bb)^1 = bb \Rightarrow bbbb$

take $i = 2$ $(bb)^2 = bbbb \Rightarrow bbbbbb$.

for $i = 0$ & $i = 2$ it is clear that our assumption is wrong.

Hence $L = \{ b^{i^2} \mid i > 1 \}$ is not regular.

Example 3: show that $\{ a^n b^{2n} \mid n > 0 \}$ is not regular.

Assume that L is regular & $L = \{ abb, aabbbb, aaabbbbbb, \ldots \}$

Length of the language is $n + 2n = 3n = |w|$

length of pumping lemma $= n$

$|w| > n$

take $n = 2$

$a^n b^{2n} = a^2 b^4 \Rightarrow \underset{x}{aa}\,\underset{y}{bb}\,\underset{z}{bb}$

case(i) y in b

$x = aa \quad y = bb \quad z = bb$

for $i \geq 0 \quad xy^iz \in L$

for $i = 0$

$aa(bb)^0bb = aabb \notin L$

case (ii) y in a

$\underbrace{aa}_{x}\underbrace{a}_{y}\underbrace{bbbb}_{z}$

$x = a, \; y = a \; \& \; z = bbbb$

- for $i = 0$

$xy^iz = a(a)^0bbbb = abbbb \notin L$

- for $i = 2$

$xy^2z = aaabbbb \notin L$

case (iii) y in a & b

$\underbrace{a}_{x}\underbrace{abbb}_{y}\underbrace{b}_{z}$

$x = a \quad y = abbb \quad z = bb$

for $i = 0$ ~~abbb~~ $ab \notin L$

$i = 2 \quad aabbbabbbb \notin L$

Since the language $\{a^nb^{2n} \mid n > 0\}$ is not regular.

Example 4:

Show that the language $L = \{a^ib^{2i} \mid i > 0\}$ is not regular.

Solution: The set of strings accepted by language L is

$L = \{abb, aabbbb, aaabbbbbb, \ldots\}$

Applying pumping lemma for any of the strings above.

Take the string abb

It is of the form xyz.

where $|xy| \leq i$, $|y| \geq 1$

To find i such that $xy^i z \notin L$

Take $i = 2$ here, then

$xyz = abb$

$xy^2z = a(bb)b$

$= abbb$

Hence $xy^2z = abbb \notin L$

Since abbb is not present in the strings of L.

$\therefore$ L is not regular.

Closure Properties of Regular languages:

→ closure properties on regular languages are defined as certain operations on regular language which are guaranted to produce regular language.

→ closure refers to some language resulting in a new languages that is of same "type" as originals operated i.e., regular.

consider L_1 & L_2 are regular languages, then

① The union of two regular languages $(L_1 \cup L_2)$ is regular.

② The intersection of two regular languages $(L_1 \cap L_2)$ is regular.

③ The complement of two regular languages is regular,

L complement $(\bar{L})$ is $\Sigma^* - L$.

④ Reversal of a regular language is regular i.e. L^R.

$L = \{0, 01, 100\}$ $L^R = \{0, 10, 001\}$

(5) Kleen closure: R is a regular expression from the languages L & M then R^* is a regular expression whose language is L^*.

(6) Positive closure: R is a regular expression from the language L & M, then R^+ is a regular expression whose language is L^+.

(7) set difference operator:
$L_1 - L_2$ is regular language, strings in L_1 but not in L_2.

(8) Homomorphism:
A homomorphism on an alphabet is a function that gives a string for each symbol in that alphabet.

ex: $h(0) = ab$, $h(1) = \epsilon$ extend to strings by

$h(a_1 \ldots a_n) = h(a_1)\, h(a_2) \ldots h(a_n)$

ex: $h(01010) = abababab$

if L is a regular language, and h is homomorphism on its alphabet then $h(L) = \{h(w) \mid w \text{ is in } L\}$ is also a regular language.

(9) Inverse homomorphism:

$h^{-1}(L) = \{w \mid h(w) \text{ is in } L\}$

Inverse homomorphism of a language raise to regular language.

Decision Properties of Regular languages:

These are the algorithms that define the formal description of a regular language.

(i) Emptiness
(ii) Non-emptiness
(iii) Finiteness
(iv) Infiniteness
(v) Membership
(vi) Equality

Emptiness and Non-emptiness:

step 1: The states that cannot be reached from initial states delete them (ie; delete unreachable states)
step 2: If the finite automata contains any final state the language is not empty.
step 3: If the finite automata contains null final states the regular languages is empty.

Finite & Infiniteness:

step 1: Delete unreachable states.
step 2: Delete dead states as we cannot move from the dead state to the final state.
step 3: If the finite automata contains any loops then Regular language is infinite.
step 4: If the finite automata cannot contain any loops then the regular language is finite.

Membership:

If the string over an alphabet is accepted by the machine then we say that the string is member of the regular language.

Let us say the string from a language is w, it starts from the begining state and after certain transition if it reaches to final state then the string is accepted.

Equality:

let the automatas M_1 & M_2 are said to be equal if they have same regular languages.

The automatas are minimized to derive unique properties in both of them.

Minimization of Automata:

DFA minimization stands for converting a given DFA to its equivalent DFA with minimum number of states.

5 tuple notation for DFA $<Q, \Sigma, q_0, \delta, F>$, after minimization are $<Q', \Sigma, q_0, \delta', F'>$.

Step 1: we will divide Q (set of states) in to two sets, one set contains all final states and other contains non-final states.

This partition is called P_0.

Step 2: Initialize $K = 1$

Step 3: Find P_K by partitioning the different sets of P_{K-1}. we will take all possible pair of

states If two states are distinguishable, we will split the sets into different sets in P_K.

Step4: Stop when $P_K = P_{K-1}$ (No change in partition)

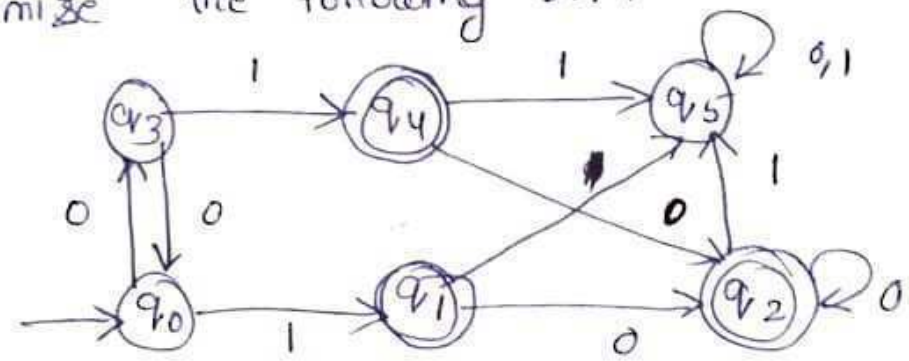
Step5 And states of one set are merged into one. Number of states in minimized DFA will be equal to number of sets in P_K.

Distinguishable. If two states are not equivalent, then we say they are distinguishable i.e, two states P & q are distinguishable if there is atleast one string w such that one of $\hat{\delta}(P,w)$ & $\hat{\delta}(q,w)$ is accepting and other is not accepting.

How to find two states are distinguishable:

Two states (q_i, q_j) are distinguishable in partition P_K if for any input a, $\delta(q_i, a)$ and $\delta(q_j, a)$ are in different sets in partition P_{K-1}.

Ex: Minimize the following DFA



Step1: $P_0 = \{\{q_1, q_2, q_4\}, \{q_0, q_3, q_5\}\}$

Step2: To calculate P_1, we will check whether sets of partition P_0 can be partitioned or not.

(i) for set $\{q_1, q_2, q_4\}$ pairs are

(q_1, q_2) (q_1, q_4) (q_2, q_4)

$(q_1, q_2) \Rightarrow \left.\begin{array}{l} \delta(q_1, 0) = q_2 \\ \delta(q_2, 0) = q_2 \end{array}\right\}$ not distinguishable (i.e, equivalent)

$\left.\begin{array}{l} \delta(q_1, 1) = q_5 \\ \delta(q_2, 1) = q_5 \end{array}\right\}$ not distinguishable.

$(q_1, q_4) \Rightarrow \left.\begin{array}{l} \delta(q_1, 0) = q_2 \\ \delta(q_4, 0) = q_2 \end{array}\right\}$

$\left.\begin{array}{l} \delta(q_1, 1) = q_5 \\ \delta(q_4, 1) = q_5 \end{array}\right\}$ not distinguishable

$(q_2, q_4) \Rightarrow \left.\begin{array}{l} \delta(q_2, 0) = q_2 \\ \delta(q_4, 0) = q_2 \end{array}\right\}$

$\left.\begin{array}{l} \delta(q_2, 1) = q_5 \\ \delta(q_4, 1) = q_5 \end{array}\right\}$ not distinguishable.

(ii) for set $\{q_0, q_3, q_5\}$ pairs are

(q_0, q_3) (q_0, q_5) (q_3, q_5)

$(q_0, q_3) \Rightarrow \left.\begin{array}{l} \delta(q_0, 0) = q_3 \\ \delta(q_3, 0) = q_0 \end{array}\right\}$ not distinguishable.

$\left.\begin{array}{l} \delta(q_0, 1) = q_1 \\ \delta(q_3, 1) = q_4 \end{array}\right\}$ not distinguishable

$(q_0, q_5) \Rightarrow \left.\begin{array}{l} \delta(q_0, 0) = q_3 \\ \delta(q_5, 0) = q_5 \end{array}\right\}$ not distinguishable.

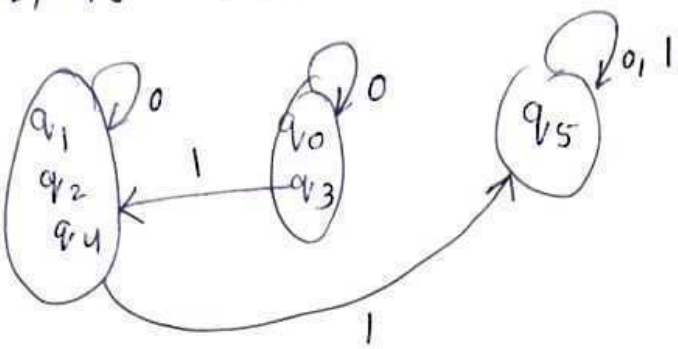
$\delta(q_0, 1) = q_1$
$\delta(q_5, 1) = q_5$ } distinguishable

We can partition the set $\{q_0, q_3, q_5\} \Rightarrow \{q_0, q_3\}, \{q_5\}$

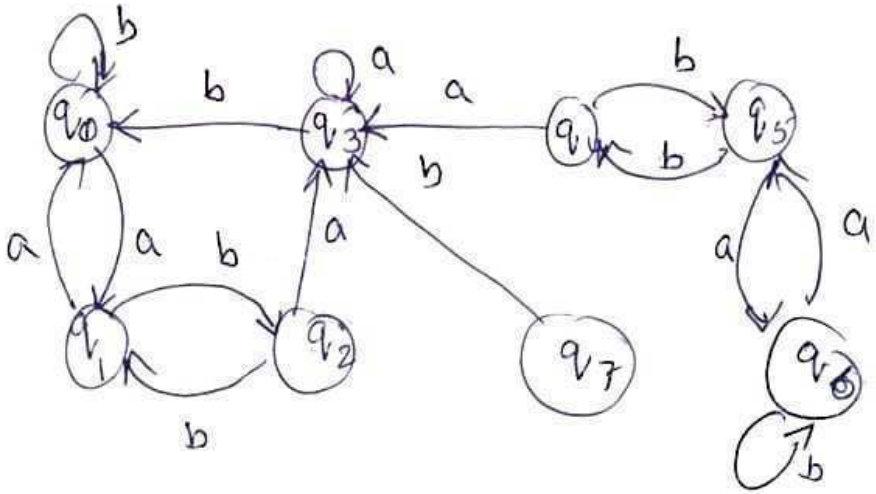
$\{q_3, q_5\}$ = $\delta(q_3, 0) = q_0$
$\delta(q_5, 0) = q_5$ } not distinguishable

$\delta(q_3, 1) = q_4$
$\delta(q_5, 1) = q_5$ } not distinguishable

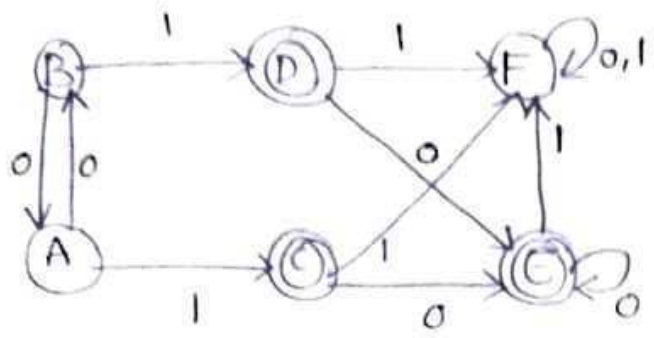
$\{q_1, q_2, q_4\}$ $\{q_0, q_3\}$ $\{q_5\}$



Example 2 minimize the following DFA



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



steps:

(1) Draw a table for all pairs of states (P, Q)

(2) Mark all pairs where $P \in F$ and $Q \notin F$ (ie., one state belongs to final state & other belongs to non-final state).

(3) If there are any unmarked pairs (P, Q) such that $\delta(P, w)$, $\delta(Q, w)$ is marked, then mark [P, Q], where 'w' is input symbol.

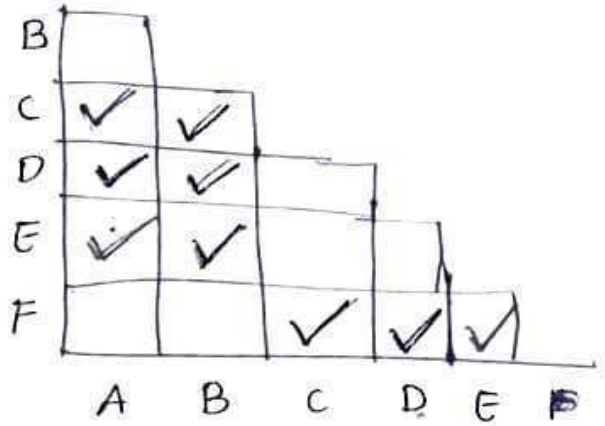
(4) Repeat this until no more markings can be made.

(5) Combine all the unmarked pairs and make them a single state in the minimized DFA.

Follow above steps for given F.A above.

step1 & 2

steps:



Step 3:

Now take the pairs that are unmarked ie, [B, A], [F, A], [F, B], [D, C], [E, C] [E, D]

→ [B, A] - $\delta(B,0) = A$, $\delta(A,0) = B$ } pair is not marked
$\delta(B,1) = D$, $\delta(A,1) = C$ } pair is not marked.

→ [F, A] - $\delta(F,0) = F$, $\delta(A,0) = B$ } Pair is unmarked
$\delta(F,1) = F$, $\delta(A,1) = C$ } Pair is marked

Since [F, C] is marked mark [F, A] in table

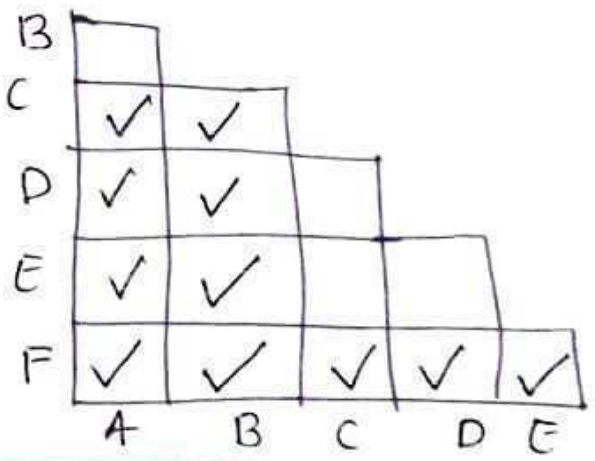
→ [F, B] - $\delta(F,0) = F$, $\delta(B,0) = A$ } Pair is marked mark [F, B]
$\delta(F,1) = F$, $\delta(B,1) = D$ } pair is marked then mark [F, B]

→ [D, C] - $\delta(D,0) = E$, $\delta(C,0) = E$ } not marked
$\delta(D,1) = F$, $\delta(C,1) = F$ } unmarked

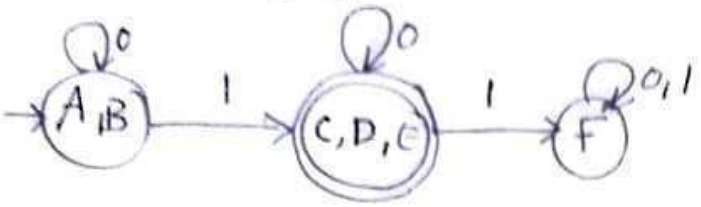
→ [E, C] - $\delta(E,0) = E$, $\delta(C,0) = E$ } not marked
$\delta(E,1) = F$, $\delta(C,1) = F$ } not marked

→ [E, D] = $\delta(E,0) = E$, $\delta(D,0) = E$ } not marked
$\delta(E,1) = F$, $\delta(D,1) = F$ } not marked

Finally the table after marking new pairs [F, A] & [F, B]



Step 4 combine all the unmarked pairs and make them a single state i.e., unmarked pairs are [A, B], [C, D], [C, E] [D, E]



is the minimized DFA

3.2 REGULAR EXPRESSIONS

The languages accepted by FA are regular languages and these languages are easily described by simple expressions called regular expressions. We have some algebraic notations to represent the regular expressions.

Regular expressions are means to represent certain sets of strings in some algebraic manner and regular expressions describe the language accepted by FA.

If Σ is an alphabet then regular expression(s) over this can be described by following rules.

1. Any symbol from $\Sigma, \in$ *and* ϕ are regular expressions.
2. If r_1 and r_2 are two regular expressions then *union* of these represented as $r_1 \cup r_2$ or $r_1 + r_2$ is also a regular expression
3. If r_1 and r_2 are two regular expressions then *concatenation* of these represented as $r_1 r_2$ is also a regular expression.
4. The Kleene closure of a regular expression r is denoted by r^* is also a regular expression.
5. If r is a regular expression then (r) is also a regular expression.
6. The regular expressions obtained by applying rules 1 to 5 once or more than once are also regular expressions.

Examples :

(1) If $\Sigma = \{a, b\}$, then

(a) a is a regular expression (Using rule 1)
(b) b is a regular expression (Using rule 1)
(c) $a + b$ is a regular expression (Using rule 2)
(d) b^* is a regular expression (Using rule 4)
(e) ab is a regular expression (Using rule 3)
(f) $ab + b^*$ is a regular expression (Using rule 6)

(2) Find regular expression for the following

(a) A language consists of all the words over $\{a, b\}$ ending in b.

(b) A language consists of all the words over $\{a, b\}$ ending in bb.

(c) A language consists of all the words over $\{a, b\}$ starting with a and ending in b.

(d) A language consists of all the words over $\{a, b\}$ having bb as a substring.

(e) A language consists of all the words over $\{a, b\}$ ending in aab.

Solution : Let $\Sigma = \{a, b\}$, and

All the words over $\Sigma = \{\in, a, b, aa, bb, ab, ba, aaa, \ldots\} = \Sigma^*$ *or* $(a + b)^*$ *or* $(a \cup b)^*$

Now let us compute for final state, which denotes the regular expression.

r_{12}^1 will be computed, because there are total 2 states and final state is q_1 whose start state is q_0.

$$r_{12}^2 = (r_{12}^1)(r_{22}^1)^*(r_{22}^1) + (r_{12}^1)$$

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

$$= 0 + 0$$

$r_{11}^1 = 0$ which is a final regular expression.

3.6.1 Arden's Method for Converting DFA to RE

As we have seen the Arden's theorem is useful for checking the equivalence of two regular expressions, we will also see its use in conversion of DFA to RE.

Following algorithm is used to build the r. e. from given DFA.

1. Let q_0 be the initial state.
2. There are $q_1, q_2, q_3, q_4, \ldots q_n$ number of states. The final state may be some q_j where $j \leq n$.
3. Let α_{ij} represents the transition from q_i to q_j.
4. Calculate q_i such that

$$q_i = \alpha_{ij} \cdot q_j$$

If q_i is a start state

$$q_i = \alpha_{ij} \cdot q_j + \epsilon$$

5. Similarly compute the final state which ultimately gives the regular expression r.

Example 1 : Construct RE for the given DFA.

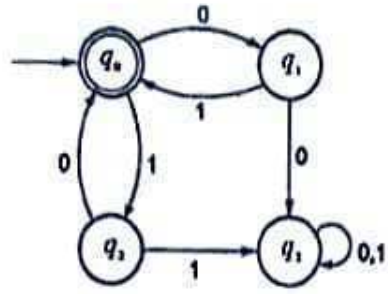
Solution :

Since there is only one state in the finite automata let us solve for q_0 only.

$$q_0 = q_0 0 + q_0 1 + \epsilon$$

$$q_0 = q_0 (0 + 1) + \epsilon$$

Example 3 : Construct RE for the DFA given in below figure.



Solution : Let us see the equations

$$q_0 = q_1 1 + q_2 0 + \epsilon$$
$$q_1 = q_0 0$$
$$q_2 = q_0 1$$
$$q_3 = q_1 0 + q_2 1 + q_3 (0+1)$$

Let us solve q_0 first,

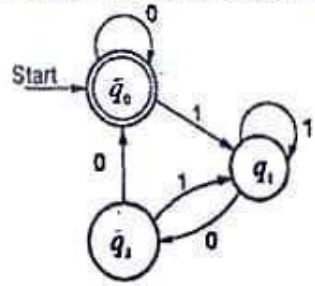
$$q_0 = q_1 1 + q_2 0 + \epsilon$$
$$q_0 = q_0 01 + q_0 10 + \epsilon$$
$$q_0 = q_0 (01+10) + \epsilon \qquad \because R = Q + RP$$
$$q_0 = \epsilon (01+10)^* \qquad \Rightarrow QP^* \text{ where}$$
$$q_0 = (01+10)^* \qquad R = q_0, Q = \epsilon, P = (01+10)$$

Thus the regular expression will be

$$r = (01+10)^*$$

Since q_0 is a final state, we are interested in q_0 only.

Example 4 : Find out the regular expression from given DFA.



$= (\{\epsilon, a, b, aa, bb, \ldots\})^*$

$= \{\epsilon, a, b, aa, bb, ab, ba, aaa, bbb, abb, baa, aabb, \ldots\}$

= {All the words over $\{a, b\}$}

$= (a + b)^*$

So, $(a^* + b^*)^* = (a + b)^*$

3.3 IDENTITIES FOR REs

The two regular expressions P and Q are equivalent (denoted as P = Q) if and only if P represents the same set of strings as Q does. For showing this equivalence of regular expressions we need to show some identities of regular expressions.

Let P, Q and R are regular expressions then the identity rules are as given below

1. $\epsilon R = R\epsilon = R$
2. $\epsilon^* = \epsilon$ ϵ is null string
3. $(\phi)^* = \epsilon$ ϕ is empty string.
4. $\phi R = R\phi = \phi$
5. $\phi + = R = R$
6. $R + R = R$
7. $RR^* = R^*R = R^+$
8. $(R^*)^* = R^*$
9. $\epsilon + RR^* = R^*$
10. $(P + Q)R = PR + QR$
11. $(P + Q)^* = (P^*Q^*) = (P^* + Q^*)^*$
12. $R^*(\epsilon + R) = (\epsilon + R)R^* = R^*$
13. $(R + \epsilon)^* = R^*$
14. $\epsilon + R^+ = R^*$
15. $(PQ)^*P = P(QP)^*$
16. $R^*R + R = R^*R$

3.3.1 Equivalence of two REs

Let us see one important theorem named Arden's Theorem which helps in checking the equivalence of two regular expressions.

Ardon's Theorem : Let P and Q be the two regular expressions over the input set Σ. The regular expression R is given as

$$R = Q + RP$$

Which has a unique solution as $R = QP^*$

Proof : Let, P and Q are two regular expressions over the input string Σ.
If P does not contain ϵ then there exists R such that

$$R = Q + RP \qquad \ldots (1)$$

We will replace R by QP^* in equation 1.
Consider R. H. S. of equation 1.

$$= Q + QP^*P$$

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

$$= QP^* \qquad \because \epsilon + R^*R = R^*$$

Thus $R = QP^*$

is proved. To prove that $R = QP^*$ is a unique solution, we will now replace L.H.S. of equation 1 by Q + RP. Then it becomes

$$Q + RP$$

But again R can be replaced by Q + RP.

$$\therefore \quad Q + RP = Q + (Q + RP)\,P$$

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

Again replace R by Q + RP.

$$= Q + QP + (Q + RP)\,P^2$$

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

Thus if we go on replacing R by Q + RP then we get,

$$Q + RP = Q + QP + QP^2 + \ldots\ldots + QP^i + RP^{i+1}$$

$$= Q(\epsilon + P + P^2 + \ldots\ldots P^i) + RP^{i+1}$$

From equation 1,

$$R = Q(\epsilon + P + P^2 + \ldots\ldots + P^i) + RP^{i+1} \qquad \ldots (2)$$

Where $i \geq 0$

Consider equation 2,

$$R = \underbrace{Q(\epsilon + P + P^2 + \ldots\ldots + P^i)}_{P^*} + RP^{i+1}$$

$$\therefore \quad R = QP^* + RP^{i+1}$$

Let w be a string of length i.

$= \{\epsilon, 0, 00, 1, 11, 111, 01, 10, \ldots\ldots\}$

$=$ { ϵ, any combination of 0's, any combination of 1's, any combination of 0 and 1 }

Hence, L. H. S. = R. H. S. is proved.

3.4 RELATIONSHIP BETWEEN FA AND RE

There is a close relationship between a finite automata and the regular expression we can show this relation in below figure.

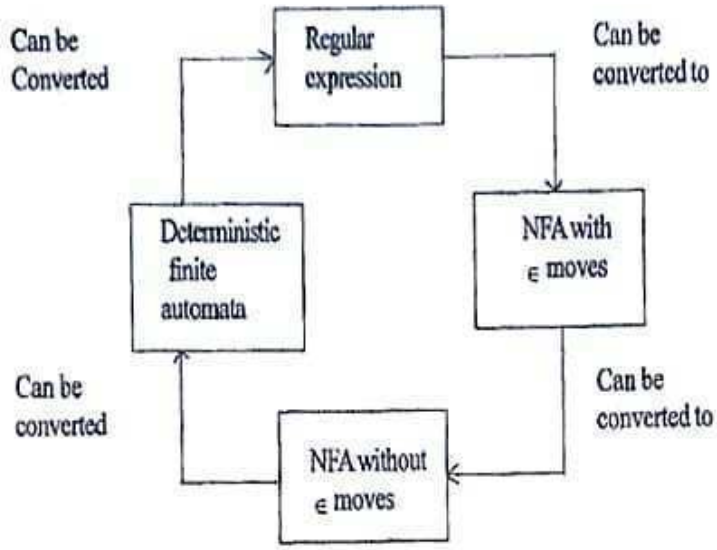


FIGURE : Relationship between FA and regular expression

The above figure shows that it is convenient to convert the regular expression to NFA with ϵ moves. Let us see the theorem based on this conversion.

3.5 CONSTRUCTING FA FOR A GIVEN REs

Theorem : If r be a regular expression then there exists a NFA with ϵ - moves, which accepts $L(r)$.

Proof : First we will discuss the construction of NFA M with ϵ - moves for regular expression r and then we prove that $L(M) = L(r)$.

Let r be the regular expression over the alphabet Σ.

Construction of NFA with ϵ - moves

Case 1 :

(i) $r = \phi$

NFA $M = (\{s, f\}, \{\ \}\delta, s, \{f\})$ as shown in Figure1 (a)

(No path from initial state s to reach the final state f.)

Figure 1 (a)

(ii) $r = \in$

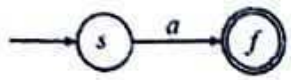
NFA $M = (\{s\}, \{\ \}, \delta, s, \{s\})$ as shown in Figure 1 (b)

(The initial state s is the final state)

Figure 1 (b)

(iii) $r = a$, for all $a \in \Sigma$,

NFA $M = (\{s, f\}, \Sigma, \delta, s, \{f\})$



(One path is there from initial state s to reach the final state f with label a.)

Figure 1 (c)

Case 2 : $|r| \geq 1$

Let r_1 and r_2 be the two regular expressions over Σ_1, Σ_2 and N_1 and N_2 are two NFA for r_1 and r_2 respectively as shown in Figure 2 (a).

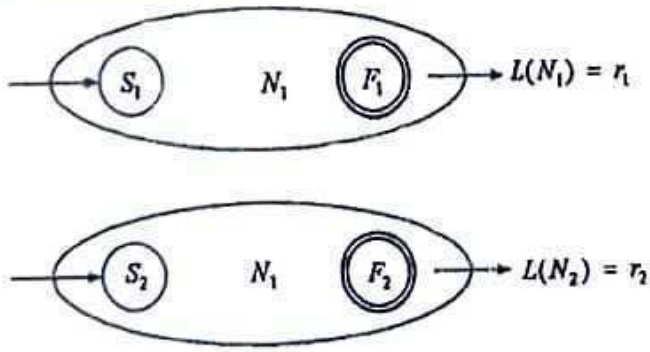


Figure 2 (a) NFA for regular expression r_1 and r_2

Now let us compute for final state, which denotes the regular expression.

r_{12}^1 will be computed, because there are total 2 states and final state is q_1, whose start state is q_0.

$$r_{12}^2 = (r_{12}^1)(r_{22}^1)^*(r_{22}^1) + (r_{12}^1)$$
$$= 0(\in)^*(\in) + 0$$
$$= 0 + 0$$

$r_{12}^2 = 0$ which is a final regular expression.

3.6.1 Arden's Method for Converting DFA to RE

As we have seen the Arden's theorem is useful for checking the equivalence of two regular expressions, we will also see its use in conversion of DFA to RE.

Following algorithm is used to build the r. e. from given DFA.

1. Let q_0 be the initial state.
2. There are $q_1, q_2, q_3, q_4, \ldots q_n$ number of states. The final state may be some q_j where $j \leq n$.
3. Let α_{ij} represents the transition from q_i to q_j.
4. Calculate q_i such that

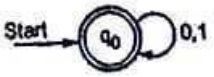
$$q_i = \alpha_{ij} \cdot q_j$$

If q_i is a start state

$$q_i = \alpha_{ij} \cdot q_j + \in$$

5. Similarly compute the final state which ultimately gives the regular expression r.

Example 1 : Construct RE for the given DFA.

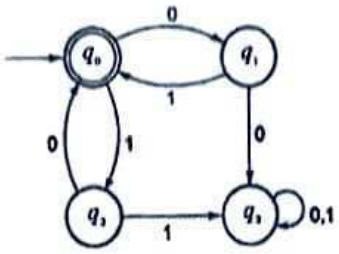


Solution :

Since there is only one state in the finite automata let us solve for q_0 only.

$$q_0 = q_0 0 + q_0 1 + \in$$
$$q_0 = q_0 (0 + 1) + \in$$

Example 3 : Construct RE for the DFA given in below figure.



Solution : Let us see the equations

$$q_0 = q_1 1 + q_2 0 + \epsilon$$
$$q_1 = q_0 0$$
$$q_2 = q_0 1$$
$$q_3 = q_1 0 + q_2 1 + q_3 (0+1)$$

Let us solve q_0 first,

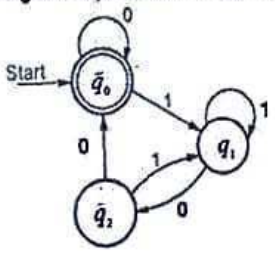
$$q_0 = q_1 1 + q_2 0 + \epsilon$$
$$q_0 = q_0 01 + q_0 10 + \epsilon$$
$$q_0 = q_0 (01+10) + \epsilon \qquad \because R = Q + RP$$
$$q_0 = \epsilon (01+10)^* \qquad \Rightarrow QP^* \text{ where}$$
$$q_0 = (01+10)^* \qquad R = q_0, Q = \epsilon, P = (01+10)$$

Thus the regular expression will be

$$r = (01+10)^*$$

Since q_0 is a final state, we are interested in q_0 only.

Example 4 : Find out the regular expression from given DFA.



Example 8 : Show that the language $L = \{a^i b^n | i > 0\}$ is not regular.

Solution : The set of strings accepted by language L is,

$L = \{abb, aabbbb, aaabbbbbb, aaaabbbbbbbb...\}$

Applying Pumping lemma for any of the strings above.

Take the string abb.

It is of the form uvw.

Where, $|uv| \leq i, |v| \geq 1$

To find i such that $uv^i w \notin L$

Take i = 2 here, then

$uv^2w = a(bb)b$

$= abbb$

Hence $uv^2w = abbb \notin L$

Since abbb is not present in the strings of L.

$\therefore$ L is not regular.

Example 9 : Show that $L = \{0^n | n$ is a perfect square $\}$ is not regular.

Solution :

Step 1 : Let L is regular by Pumping lemma. Let n be number of states of FA accepting L.

Step 2 : Let $z = 0^n$ then $|z| = n \geq 2$.

Therefore, we can write z = uvw; Where $|uv| \leq n, |v| \geq 1$.

Take any string of the language L = { 00, 0000, 000000 .... }

Take 0000 as string, here u = 0, v = 0, w = 00 to find i such that $uv^i w \notin L$.

Take i = 2 here, then

$uv^2w = 0(0)^2 00$

$= 00000$

This string 00000 is not present in strings of language L. So $uv^i w \notin L$.

$\therefore$ It is a contradiction.

3.9 PROPERTIES OF REGULAR SETS

Regular sets are closed under following properties.

1. Union
2. Concatenation

3. Kleene Closure
4. Complementation
5. Transpose
6. Intersection

1. **Union :** If R_1 and R_2 are two regular sets, then union of these denoted by $R_1 + R_2$ or $R_1 \cup R_2$ is also a regular set.

Proof : Let R_1 and R_2 be recognized by NFA N_1 and N_2 respectively as shown in Figure1(a) and Figure1(b).

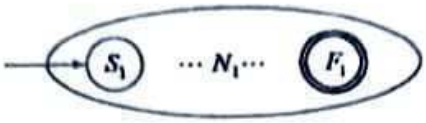


FIGURE 1(a) NFA for regular set R_1

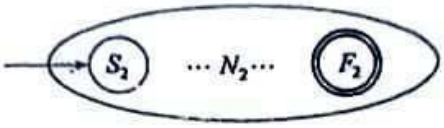


FIGURE 1(b) NFA for regular set R_2

We construct a new NFA N based on union of N_1 and N_2 as shown in Figure 1 (c)

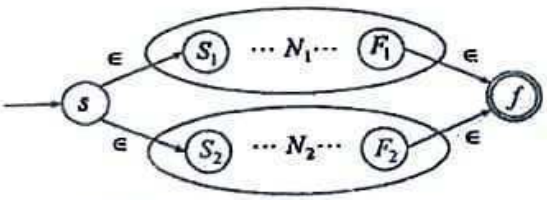


FIGURE 1(c) NFA for $N_1 + N_2$

Now,

$$L(N) = \in L(N_1) \in + \in L(N_2) \in$$
$$= \in R_1 \in + \in R_2 \in$$
$$= R_1 + R_2$$

Since, N is FA, hence $L(N)$ is a regular set (language). Therefore, $R_1 + R_2$ is a regular set.

2. **Concatenation :** If R_1 and R_2 are two regular sets, then concatenation of these denoted by R_1R_2 is also a regular set.

Proof : Let R_1 and R_2 be recognized by NFA N_1 and N_2 respectively as shown in Figure 2(a) and Figure 2(b).

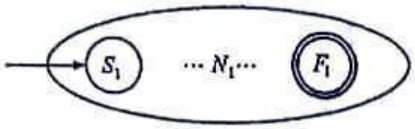


FIGURE 2(a) NFA for regular set R_1

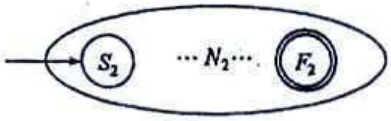


FIGURE 2(b) NFA for regular set R_2

We construct a new NFA N based on concatenation of N_1 and N_2 as shown in Figure2(c).

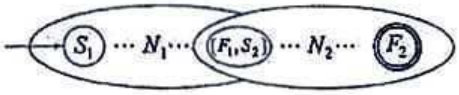


FIGURE 2(c) NFA for regular set R_1R_2

Now,

$L(N)$ = Regular set accepted by N_1 followed by regular set accepted by $N_2 = R_1R_2$

Since, $L(N)$ is a regular set, hence R_1R_2 is also a regular set.

3. **Kleene Closure :** If R is a regular set, then Kleene closure of this denoted by R^* is also a regular set.

Proof : Let R is accepted by NFA N shown in Figure 3(a).

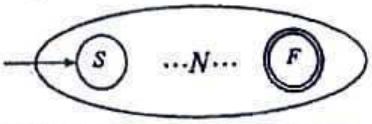
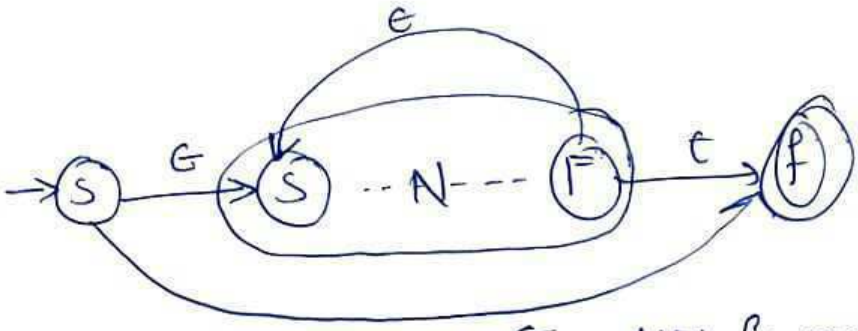


FIGURE 3(a) NFA for regular set R



NFA for regular expression R^*

Unit - III

CONTEXT FREE GRAMMARS

Context free grammar:

A grammar 'G' is said to be CFG if every production of the form '$\alpha \rightarrow \beta$' satisfies the following rules.

Rule 1: LHS should contain only one non-terminal.

$\therefore \alpha \in N^*$ and $|\alpha| = 1$

Rule 2: RHS can be anything, Non-terminal or terminals.

$\therefore \beta \in (N+T)^*$

→ It is denoted by 4 tuple Notation.

$G = \{V, T, P, S\}$

where, V - Set of variables or non-terminals

T = set of terminals

S = starting symbol of the grammar.

P = set of finite no. of productions or rules.

Example: Consider $E \rightarrow E+T \mid T$

$T \rightarrow F$

$F \rightarrow id$

In the above CFG

V or non-terminals = $\{E, T, F\}$

Terminals $T = \{+, id\}$

$S = \{E\}$

$P = \{E \rightarrow E+T,\ E \rightarrow T,\ T \rightarrow F,\ F \rightarrow id\}$

Derivations using a Grammar:

It is a step by step process of replacing the non-terminals with the appropriate productions.

→ There are two types of derivations.

(a) left most derivation (LMD)

(b) Right most derivation (RMD)

left most derivation (LMD):

In each and every step of the derivation, left most non-terminal is replaced with the production.

Ex: consider the following productions.

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

Derive id+id*id using LMD.

Solution:

$E \Rightarrow E+T$ ($\because E \rightarrow E+T$)

$\Rightarrow T+T$ ($\because E \rightarrow T$)

$\rightarrow F+T$ ($\because T \rightarrow F$)

$\Rightarrow id+T$ ($\because F \rightarrow id$)

$\Rightarrow id+T*F$ ($T \rightarrow T*F$)

$\Rightarrow id+F*F$ ($\because T \rightarrow F$)

$\Rightarrow id+id*F$ ($\because F \rightarrow id$)

$\Rightarrow id+id*id$ ($\because F \rightarrow id$)

$E \overset{*}{\Rightarrow} id+id*id$.

Right most derivation:

In each and every step of the derivation right most non-terminal is replaced with the production.

Example:

Consider the following productions.

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

Derived id + id * id using RMD

Solution: $E \Rightarrow E+T$ ($\because E \rightarrow E+T$)

$\Rightarrow E+T*F$ ($\because T \rightarrow T*F$)

$\Rightarrow E+T*id$ ($\because F \rightarrow id$)

$\Rightarrow E+F*id$ ($\because T \rightarrow F$)

$\Rightarrow E+id*id$ ($\because F \rightarrow id$)

$\Rightarrow T+id*id$ ($\because E \rightarrow T$)

$\Rightarrow F+id*id$ ($\because T \rightarrow F$)

$\Rightarrow id+id*id$ ($\because F \rightarrow id$)

$E \overset{*}{\Rightarrow} id+id*id$.

The language of grammar:

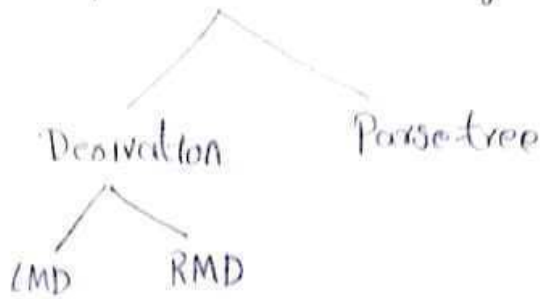
Every grammar generates a language such as regular grammar generates regular language, context free grammar generates context free language.

→ The language generated by a grammar is nothing but the strings accepted by the grammar.

→ A string can be accepted or a string is a member of a grammar can be analyzed by 2 ways

(a) Derivation (b) Parse-tree.

Acceptance of a string



Parse tree

Also known as syntax tree or derivation tree.

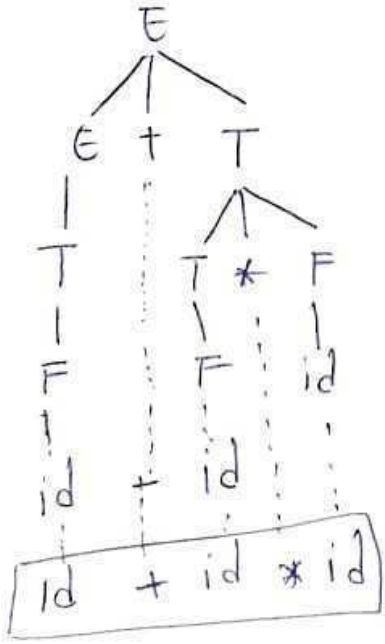
→ It is an hierarchial representation of a derivation

→ It consists of
- a root : starting : Non-terminal
- a leaf : Terminals
- Internals : Non-terminals.

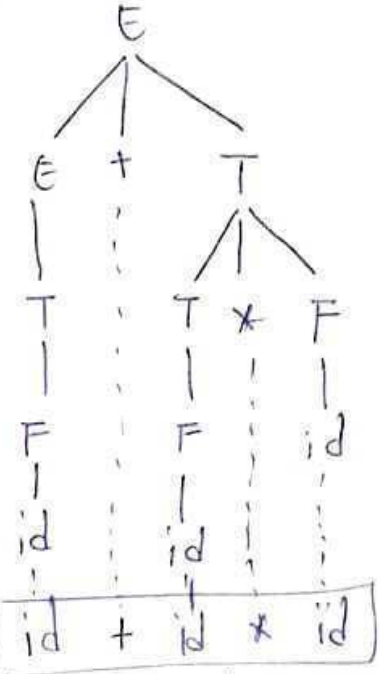
Ex: Consider a string 'w' = id + id * id from the above example in LMD / RMD

Deriving 'w' using parse tree :

LMD



RMD



We can observe that the parse tree of both LMD & RMD are same for w = id + id * id

Applications of CFG:

1. It is the most widely used grammars.
2. CFG's are used in text processing and compilers.
3. The parsers makes use of CFG to check the syntax of the source program, written in any language. Every statement of the source program is converted to its equivalent CFG and then to syntax tree by the parser.
4. CFG is also used in markup languages.
5. syntax analysis phase uses CFG to specify the syntactic rules to any programming languages.
6. semantic anlysis phase uses CFG to specify the type checking rules.

→ context free grammars were originally conceived by N.chomsky to describe natural languages.

→ CFG's are used to describe programming languages, and development of XML (Extensible Markup language), an essential part of XML is the Document type definition (DTD).

Parsers:

1. YACC - Parser-generator
2. XML and Document-Type definitions.

Parsers: Many aspects of a programming language have a structure that may be described by regular expressions. Typical languages use parentheses and/or brackets in nested and balanced fashion. We must be able to

match some left parenthesis against right parenthesis that appears immediately to its right, remove both of them, and repeat. If we eventually eliminate all the parentheses, then string was balanced and if we cannot match parentheses, then it is unbalanced.

Examples of strings of balanced parenthesis are (()), ()(), (()()) and ϵ, while)(and (() are not.

A Grammar $G_{bal} = (\{B\}, \{(,)\}, P, B)$ generates all and only the strings of balanced parentheses, where P consists of the productions.

$$B \longrightarrow BB \mid (B) \mid \epsilon$$

- First production $B \rightarrow BB$ says concatenation of two strings of balanced parenthesis is balanced.
- Second Production $B \rightarrow (B)$, says if we replace a pair of parenthesis around a balanced string, then the result is balanced.
- Third production $B \rightarrow \epsilon$ is the basis it says that the empty string is balanced.

YACC - Parser Generator:

The generation of a parser (function that creates parse trees from source programs) has been institutionalized in the YACC command that appears in all unix systems

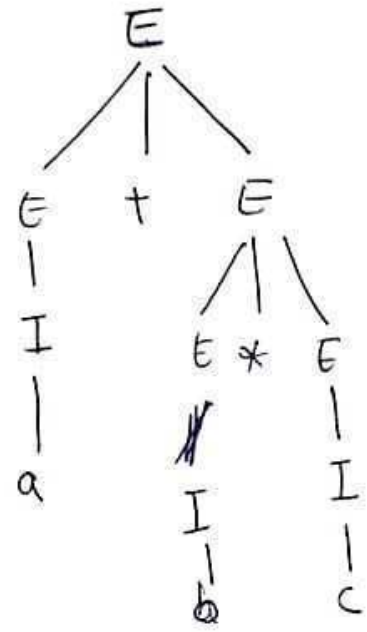
Ambiguity in Grammars & languages.

A CFG is said to be ambiguous if there exists some $w \in L(G)$ which has two distinct derivation trees. Alternatively, ambiguity implies the existence of two or more left most or right most derivations.

Ex consider the grammar $G_1 = (V, T, P, S)$ with $V = \{E, I\}$ $T = \{a, b, c, +, *, (,)\}$ and productions:

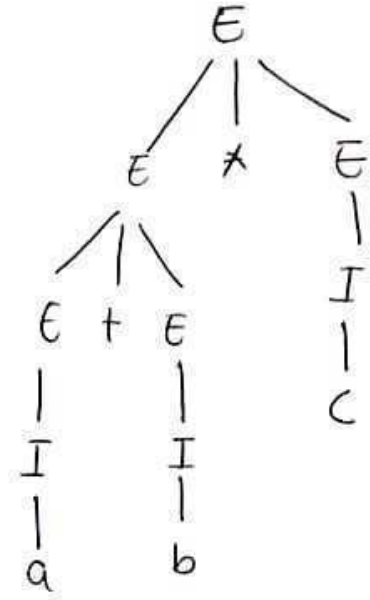
$E \rightarrow I$
$E \rightarrow E + E$
$E \rightarrow E * E$
$E \rightarrow (E)$
$I \rightarrow a \mid b \mid c$

consider two derivation trees for a+b*c



If a=5 b=6 c=7

value of tree will be 47



If a=5 b=6 c=7

value of tree will be 77

Inherently ambiguous

A context free language L is said to be inherently ambiguous if all its grammars are ambiguous.

Ex consider the grammar for string aabbccdd

$S \rightarrow AB \mid C$

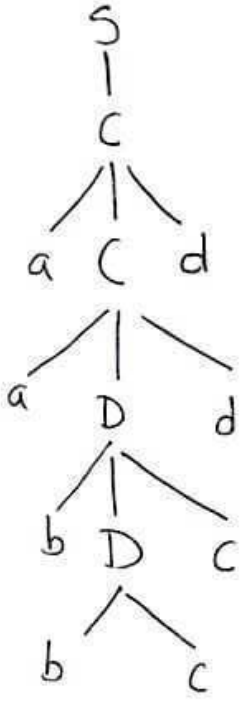
$A \rightarrow aAb \mid ab$

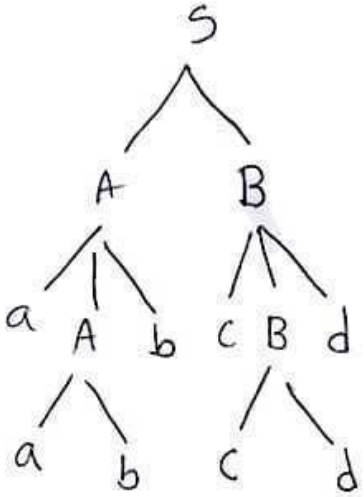
$B \rightarrow cBd \mid cd$

$C \rightarrow aCd \mid aDd$

$D \rightarrow bDc \mid bc$

Parse tree for string aabbccdd





PUSH DOWN AUTOMATA

A Finite automata with a stack memory can be viewed as push down automata. (PDA)

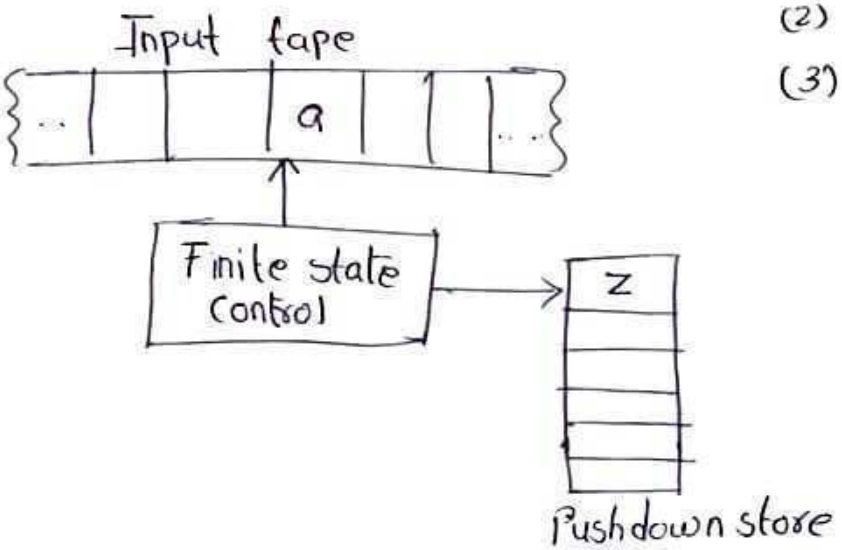
→ It can be defined as the automata or machine that processes all the context free languages.

→ FA cannot recognize every context free language and some context free languages are not regular.

→ some extra features are added to F.A inorder to accept any type of CFL.

→ The PDA performs transitions by observing symbols at top of the stack, present state and on the input symbol.

PDA consists of three components
(1) Input tape
(2) control unit
(3) stack structure



→ Input tape consists of linear configuration of cells each of which contains a character from an alphabet. The tape can be moved one cell at a time to the left.

→ The stack is also a sequential structure that has a first element.

→ The control unit contains both tape heads

Formal definition of PDA.

Formally PDA is defined as a seven tuple machine.

$$M = (Q, \Sigma, \Gamma, \delta, q_0, Z, F)$$

where Q → Finite set of states

Σ → set of input symbols

$\delta \rightarrow Q \times \Sigma \times \Gamma \rightarrow Q \times \Gamma^*$

Γ → Finite set of symbols to push on the stack called "stack alphabet"

q_0 → Initial state of control unit $\in Q$.

Z → stack start symbol

F → set of final states $F \subseteq Q$.

The arguments of δ are current state of control unit, the current input symbol and current stack symbol on the top of the stack.

Instantaneous description of a PDA:

Instantantaneous description of a PDA is a triplet (q, w, u)

where q = current state of the automaton

w = unread part of the input string.

u = stack contents (written as a string, with leftmost symbol on top of the stack).

Let the symbol '$\vdash$' denote move of the PDA and suppose that $\delta(q_1, a, x) = \{(q_2, y), \ldots\}$ then following is possible.

$$(q_1, aw, xz) \vdash (q_2, w, yz)$$

This notation tells that in moving from state q_1 to state q_2, an 'a' is consumed from the input string 'aw' and the x at the top of the stack 'xz' is replaced with y, leaving yz on the stack.

Example:

Construct a Push down automata for a language

$$L = \{a^n b^n \mid n \geq 0\}$$

Solution: Firstly note down the basic strings generated by the following language.

$$L = \{\epsilon, ab, aabb, aaabbb, \ldots\}$$

→ which means the input symbol 'a' is followed by the input symbol 'b' and no. of input symbols 'a' is equal to the no. of input symbols 'b'.

→ The main idea here is to push all the a's in the stack until the control reaches 'b' in the string, then pop 'a' for each 'b'.

→ Repeat the above step until the stack is empty and we reach the end point of the string 'ϵ'.

Note: If we reach the end point of the string and still find 'a' in the stack then it means that number of

'a's are more than b's.

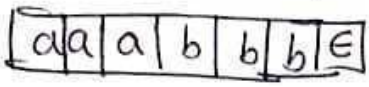
∴ Not acceptable / Rejected.

→ If the stack is empty and there are still b's left in the string, then it means that b's are more than a's.

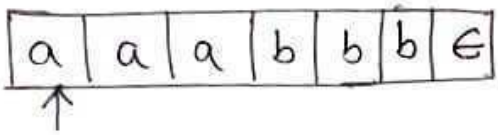
∴ Not acceptable / Rejected.

→ consider the string generated by the language

string

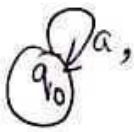


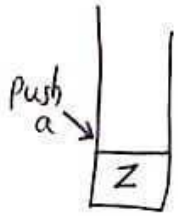
step1 : Read the first input symbol



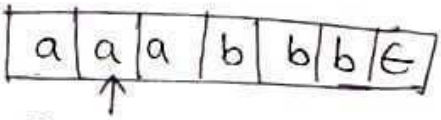
current state	i/p	Top of stack	next state	operation
q_0	a	Z	q_0	push

$(q_0, a, Z) \vdash \{q_0, aZ\}$

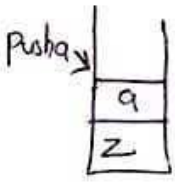




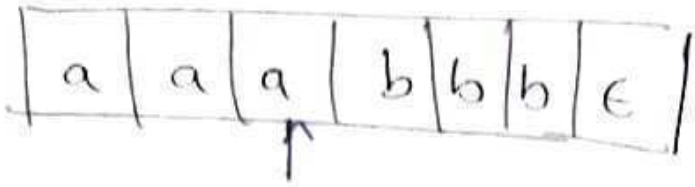
step2

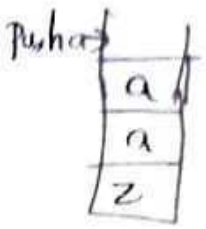


$(q_0, a, a) = \{q_0, aa\}$



step3 ;

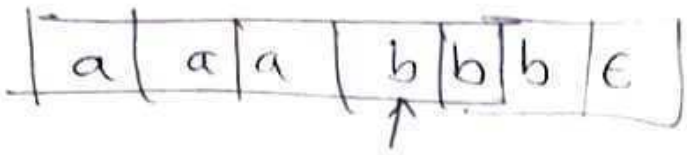


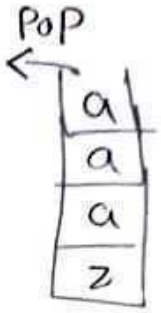


$(q_0, a, a) = \{q_0, aa\}$

repeat step 2

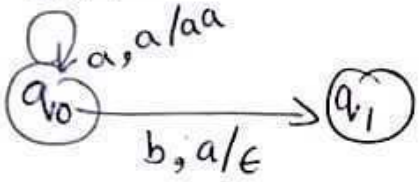
step 3 :



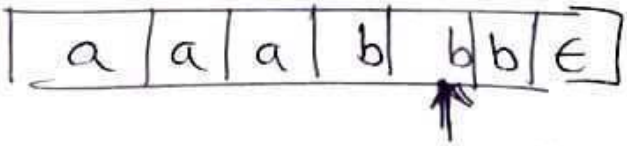


$(q_0, b, a) \vdash \{q_1, \epsilon\}$

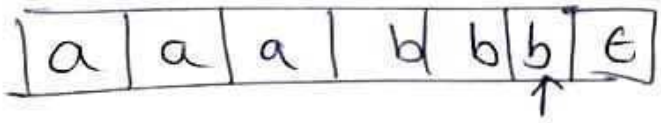
→ Pop operation
→ change the state

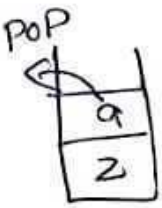


step 4:

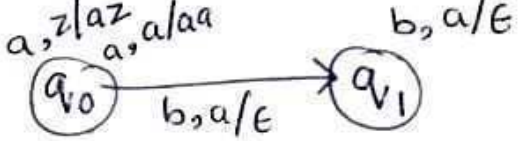


$(q_1, b, a) \vdash \{q_1, \epsilon\}$

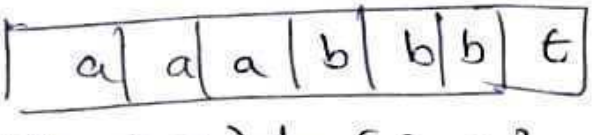




repeat the step 4



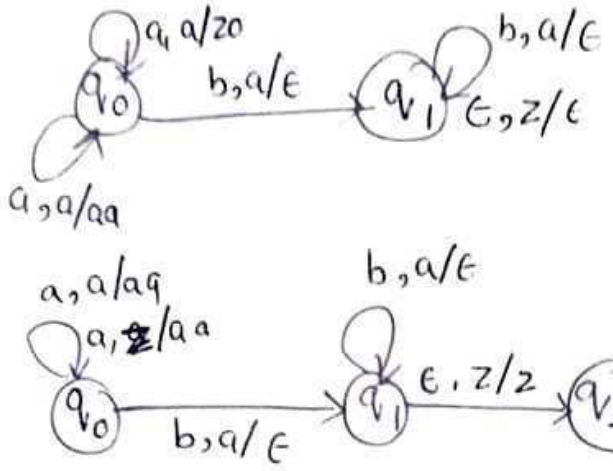
step 5 :



$(q_1, \epsilon, z) \vdash \{q_1, \epsilon\}$

∴ Stack is empty reached the end of the string.

∴ The final PDA (acceptance by empty stack)



Acceptance by final state

- Introduce a new state from q_1 to q_f

$$(q_1, \epsilon, Z_0) \vdash (q_f, z)$$

Instantaneous description is

$$\delta(q_0, a, z) \vdash (q_0, az)$$

$$\delta(q_0, a, a) \vdash (q_0, aa)$$

$$\delta(q_0, b, a) \vdash (q_1, \epsilon)$$

$$\delta(q_1, b, a) \vdash (q_1, \epsilon)$$

$$\delta(q_1, \epsilon, z_0) \vdash (q_1, \epsilon)$$

$$\delta(q_1, \epsilon, z_0) \vdash (q_f, z_0)$$

The simulation of the string is as follows

$$\delta(q_0, aaabbb, z_0) \Rightarrow \delta(q_0, aabbb, az_0)$$

$$\Rightarrow \delta(q_0, abbb, aaz_0)$$

$\Rightarrow \delta(q_0, bbb, aaaz_0)$

$\Rightarrow \delta(q_1, bbb, aaz_0)$

$\Rightarrow \delta(q_1, bb, az_0)$

$\Rightarrow \delta(q_1, b, z_0)$

$\Rightarrow \delta(q_1, \epsilon, z_0)$

$\Rightarrow \delta(q_2, \epsilon)$ Accepted.

The Languages of PDA:

There are two ways to define the language of a PDA $P = (Q, \Sigma, T, \delta, q_0, z_0, F)$ $(L(P) \subseteq \Sigma^*)$ because there are two notions of acceptance.

i) Acceptance by final state.
ii) Acceptance by empty stack.

Acceptance by final state:

A Push down automata is defined by $P=(Q, \Sigma, T, \delta, q_0, z_0, F)$ accepts a given language by generating the strings $w \in \Sigma^*$, by entering the final state such that

$(q_0, w, z_0) \Rightarrow (q_f, \epsilon, \alpha)$ where $q_f \subseteq F$ and

$\alpha \subset T^*$

ie., A PDA can accept strings through final state based on the following conditions.

(a) If string is completely traversed (i.e, it comes to an end)

(b) If the PDA has visited its final state

i.e. $\delta(q_i, \epsilon, z_0) = (q_f, z_0)$

Acceptance by Empty stack:

A PDA is defined by $P = (Q, \Sigma, T, \delta, q_0, z_0, F)$ accepts a given language by generating the set of strings. $w \in \Sigma^*$ by making its stack empty such that

$$(q_0, w, z_0) \Longrightarrow (q_0, \epsilon, \epsilon)$$

→ A PDA can accept strings by empty stack based on the following conditions.

(a) If the string is completely traversed (i.e, it comes to one end)

(b) If the PDA has its stack empty.

Thus if a string of PDA are accepted by empty stack then the final state should be of type.

$$\delta(q_i, \epsilon, z_0) \Rightarrow (q_f, \epsilon).$$

Example 2 PDA for

$L = \{a^n b^m c^n \mid n \geq 1, m \geq 1\}$

Sol: Strings generated by this language is

$L = \{abc, aabcc, abbbc, \ldots\}$

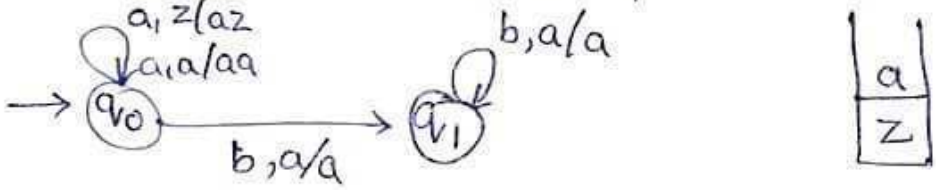
ie. equal no. of a's & equal number of c's and any number of b's between a's & c's.

→ The concept here is all a's are pushed on to the stack, when b will come 'no operation' and for each 'c' pop one 'a' then stack will be empty.

steps are as follows.

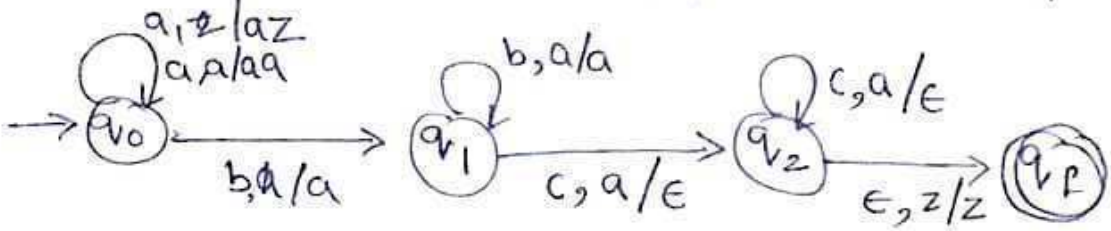
(i) First push 'a' on to the stack.

(ii) when b is read as input.



Here any number of b's may be read, but no operation is performed.

(iii) when 'c' is read then pop 'a' from top of stack ie.,



For each 'c' pop 'a' then stack will be empty then reaches final state. This language is deterministic PDA (DPDA).

Consider any string 'aabbbcc' generated by given language

$(q_0, aabbbcc, z) \vdash (q_0, abbbcc, az)$ (∵ 'a' pushed on to stack)

$\vdash (q_0, bbbcc, aaz)$ (second 'a' is also pushed on to stack)

$\vdash (q_0, bbcc, aaz)$ (∵ when 'b' is read stack top remains same as no operation is performed)

$\vdash (q_1, bcc, aaz)$ (∵ No operation)

$\vdash (q_1, cc, aaz)$

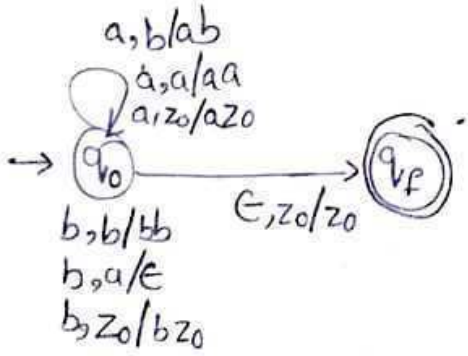
$\vdash (q_2, c, az)$ (∵ when 'c' is read change state and pop 'a')

$\vdash (q_2, \epsilon, z)$

Therefore when ϵ is read then state reaches the acceptance / final state.

Example: PDA for $L = \{w \in (a,b)^* \mid na = nb\}$

Soln: $L = \{\epsilon, ab, ba, abab, baba, \ldots\}$



Push down Automata.

(1) DPDA (Deterministic push down automata)

(2) NPDA (Non-deterministic " ")

DPDA :

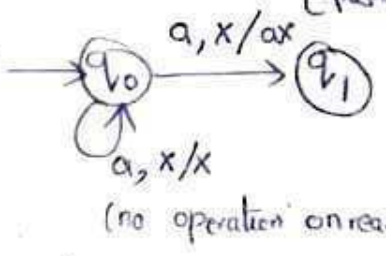
(1) Center symbol is known (ie. up to which element we need to perform push and at which element we need to pop ex: $a^n b^n$, wcw^R.

(2). There is only one moves in every situation

$$\delta(q_0, a, Z_0) \Rightarrow (q_0, aZ_0)$$

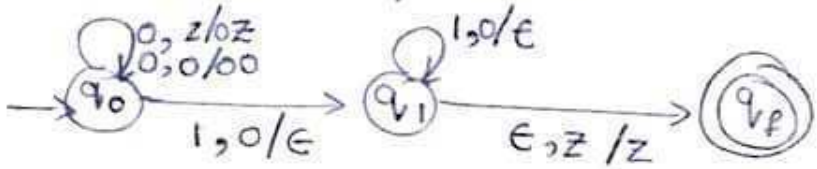
If M is deterministic then it should satisfy, for any $q \in Q$, $a \in \Sigma$ and $X \in T$ then (q, a, X) has atmost one transition, and $(q, \epsilon, X) \neq \emptyset$ then $(q, a, X) \neq \emptyset$ for every $a \in \Sigma$.

Ex:



Here two operations are performed for same input symbol for same state, this is not allowed in DPDA.

→ The grammar $L = \{0^n 1^n / n \geq 1\}$, ther string 0011 has the PDA



The ID's here are

$(q_0, 0, z) \vdash (q_0, 0z)$

$(q_0, 0, 0) \vdash (q_0, 00)$

$(q_0, 1, 0) \vdash (q_1, \epsilon)$

$(q_1, 1, 0) \vdash (q_1, \epsilon)$

$(q_1, \epsilon, z) \vdash (q_f, z)$

This is a deterministic PDA because in first two steps of ID input is same and stack top is different

NPDA

(1) Center is not known

i. ww^R suppose aaaa is string, we donot know for which we need to push or pop.

(2) There are multiple moves in every situation

$\delta(q_0, a, a) \vdash (q_0, aa)$ (or) (q_1, a) (or) (q_1, ϵ)

Examples of DPDA & NPDA are

① construct PDA for $L = \{ ww^R \mid w \in (a,b)^* \}$

solution: Language $L = \{ aa, bb, abba, abaaba, \ldots \}$

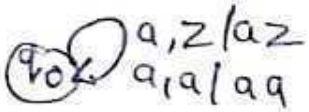
$\underbrace{a}_{w}\ \underbrace{a}_{w^R}$ $\quad \underbrace{ab}_{w}\ \underbrace{ba}_{w^R}$ re

b	b
a	a
ab	ba
ba	ab
abb	bba
w	w^R

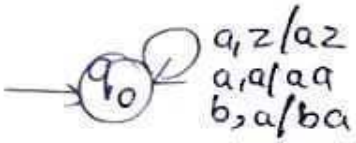
→ The concept here is for each symbol of 'w' we need to perform push operation and for w^R perform pop operation, then stack will be empty.

→ some times we are not clear where to stop push operation and start pop operation, so this cannot be a deterministic PDA.

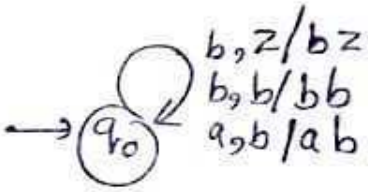
→ when ~~we~~ we read input a then



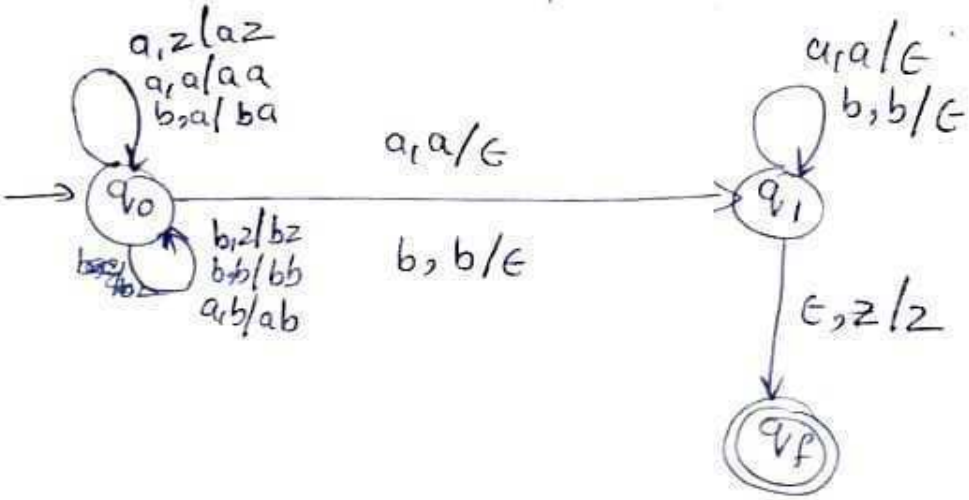
→ when we read input b then (ie. $w = aab$)



→ Suppos 'b' is read on empty stack (ie $w = bba$)



→ To go to q_1 state



This is a Non deterministic PDA because for

$(q_0, a, a) \Rightarrow (q_0, aa)$ – push operation

$(q_0, a, a) \Rightarrow (q_1, G)$ – pop operation

Here for same i/p & same stack top different operations are performed (that's why this is 'NPDA'.

→ This is a CFL because only NPDA can have CFL language, but not DPDA.

For string abba

$(q_0, w, z) \vdash (q_0, abba, z)$

(q_0, bba, az)

(q_0, ba, baz)

(q_1, a, az)

(q_1, ϵ, z)

Example

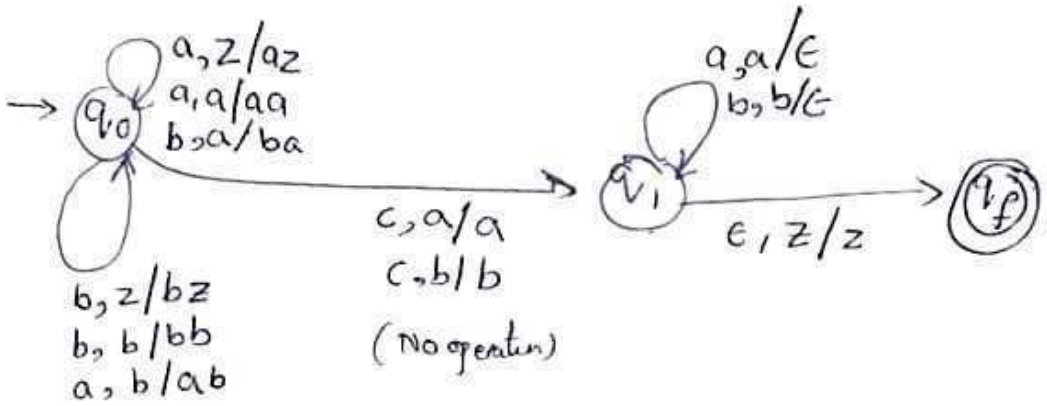
$L = \{ wcw^R \mid w \in (a,b)^* \text{ and } c \text{ is input alphabet} \}$

Solution consider strings aca

bcb

abcba

w w^R



This is deterministic PDA.

Consider string abcba

$(q_0, abcba, Z) \vdash (q_0, bcba, aZ)$

$\vdash (q_0, cba, baZ)$

$\vdash (q_1, ba, baZ)$

$\vdash (q_1, a, aZ)$

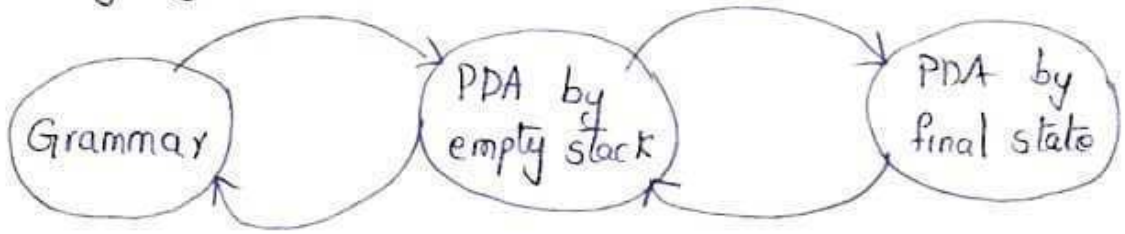
$\vdash (q_1, \epsilon, Z)$

$\vdash (q_f, Z)$

→ This language is deterministic context free language.

Equivalence of CFG & PDA:

Languages defined by PDA's are exactly the context free language.



Equivalence of three ways of defining the CFL's

The goal is to prove that the following three classes of languages.

1. The context free languages ie, the languages defined by CFG's.
2. The languages that are accepted by final state by some PDA.
3. The languages that are accepted by empty stack by some PDA.

Conversion from grammars to Push down automata:

$G = \{V, T, P, S\}$ $\quad M = \{Q, \Sigma, T, \delta, q_0, Z_0, F\}$

$A \rightarrow \alpha$

$A \in V$ & $\alpha \in (VUT)^*$

Then Push down automata P that accepts $L(G)$ by empty stack as follows:

$P = (\{q\}, T, VUT, \delta, q, S)$

$\{q\}$ – no of states

T – i/p symbol is replaced by T.

T – is replaced by VUT

where transition function δ is defined by:

1. For each variable A,

$$\delta(q, \epsilon, A) = \{(q, \beta) \mid A \to \beta \text{ is a production of } P\}$$

(ie, push operation)

2. For each terminal a,

$$\delta(q, a, a) = \{(q, \epsilon)\} \quad \text{(ie, pop operation).}$$

Example

① construct a PDA equivalent to the grammar

$$S \to aAA$$
$$A \to aS \mid bS \mid a.$$

Solution: First identify set of Variables & Terminals

$$T = \{a, b\}$$
$$V = \{S, A\}$$

For every variable

$$\delta(q, \epsilon, A) = \{(q, aAA)\}$$

$$\delta(q, \epsilon, S) = \{(q, aS), (q, bS), (q, a)\}$$

For each terminal

$$\delta(q, a, a) = \{(q, \epsilon)\}$$
$$\delta(q, b, b) = \{(q, \epsilon)\}$$

These four transition functions are used for PDA

Consider string aabb is acceptable or not

$$\delta(q, aabb, S) \vdash$$

Ex 2 Construct a PDA for the following CFG and test whether "abbabb" is in N(P).

$G = (\{S, A\}, \{a, b\}, P, S)$

$R = \{S \to AA \mid a,\ A \to SA \mid b\}$

Soln Identify set of Variables & Terminals

$V = \{S, A\}$ & $T = \{a, b\}$

For each Variable

$\delta(q, \epsilon, S) = \{(q, AA), (q, a)\}$

$\delta(q, \epsilon, A) = \{(q, SA), (q, b)\}$

For each terminal

$\delta(q, a, a) = \{(q, \epsilon)\}$

$\delta(q, b, b) = \{(q, \epsilon)\}$

$\delta(q, abbabb, S) \vdash (q, abbabb, AA)$

$\vdash (q, abbabb, SAA)$

$\vdash (q, \underline{a}bbabb, \underline{a}AA)$

$\vdash (q, bbabb, AA)$

$\vdash (q, \underline{b}babb, \underline{b}A)$

$\vdash (q, babb, A)$

$\vdash (q, babb, SA)$

$\vdash (q, babb, AAA)$

$\vdash (q, \underline{b}abb, \underline{b}AA)$

$\vdash (q, abb, AA)$

$\vdash (q, abb, SAA)$

$\vdash (q, \underline{a}bb, \underline{a}AA)$

$\vdash (q, bb, AA)$

$\vdash (q, \underline{b}b, \underline{b}A)$

$\vdash (q, b, A)$

$\vdash (q, \underline{b}, \underline{b})$

$\vdash (q, \epsilon)$

Finally stack is empty, string is accepted

Conversion of PDA to CFG:

To convert the PDA to CFG, we use the following rules:

R1: The productions from startsymbol S are given by

$S \longrightarrow [q_0, Z_0, q]$ for some state q in Q.

R2: Each move that Pops a symbol from stack with transitions.

$$\delta(q, a, Z_i) = (q, \epsilon)$$

includes a production as

$[q\ Z_i\ q_1] \longrightarrow a$ for q_1 in Q

R3: Each move that doesnot pop any symbol from stack with transition as

(i) $\delta(q, a, Z) = (q_1, Z_1 Z_2 Z_3 \ldots)$

includes a production as

$[q_1\ Z_0\ q_m] \longrightarrow a\,[q_1 Z_2 q_2]\,[q_2 Z_3 q_3]\,[q_3 Z_4 q_4] \ldots [q_{m-1} Z_m q_m]$

for each q_i in Q wher $1 \le i \le m$

(ii) $\delta(q, a, Z) = (q_1, \epsilon)$
then $[q\ Z\ q_1] \longrightarrow a$

(iii) $\delta(q, \epsilon, Z) \longrightarrow (q_1, \epsilon)$
then $[q\ Z\ q_1] \longrightarrow \epsilon$.

After defining the rules apply simplification of grammar to get reduced grammar.

Example: convert the following PDA to CFG.

$\delta(q_0, a, Z_0) = (q_0, aZ_0)$
$\delta(q_0, a, a) = (q_0, aa)$
$\delta(q_0, b, a) = (q_1, a)$
$\delta(q_1, b, a) = (q_1, a)$
$\delta(q_1, a, a) = (q_1, \epsilon)$
$\delta(q_1, \epsilon, Z_0) = (q_1, \epsilon)$

To convert to CFG first note set of variable, $V = \{S$ and seven tuple notation of PDA ie.

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

$P = (\{q_0, q_1\}, \{a, b\}, \{a, Z_0\}, \delta, q_0, Z_0, F)$

$V = \{S, [q_0, a, q_0], [q_0 Z_0 q_0], [q_0 a q_1], [q_0 Z_0 q_1],$

$[q_1 a q_0], [q_1 Z_0 q_0] [q_1 a q_1] [q_1 Z q_1] \}$

Now writing productions for states by considering RI.

(i) $S \longrightarrow [q_0 Z_0 q_0]$

$S \longrightarrow [q_0 Z_0 q_1]$

} $[q_0, Z_0, -]$ is common then followed by state $\{q_0, q_1\}$

(ii) $\delta(q_0, a, Z_0) = (q_0, aZ_0)$ (∵ As there are two symbol in stack $2^2 = 4$ productions are possible)

As 2^2 → power is 2 (in right side has two derivations).

$[q_0 Z_0 q_0] \longrightarrow a [q_0 a q_0] [q_0 Z_0 q_0]$

$[q_0 Z_0 q_0] \longrightarrow a [q_0 a q_1] [q_1 Z_0 q_0]$

$[q_0 Z_0 q_1] \longrightarrow a [q_0 a q_0] [q_0 Z_0 q_1]$

$[q_0 Z_0 q_1] \longrightarrow a [q_0 a q_1] [q_1 Z_0 q_1]$

states are written alternatively

(iii) $\delta(q_0, a, a) = (q_0, aa)$

$[q_0 a q_0] \longrightarrow a [q_0 a q_0] [q_0 a q_0]$

$[q_0 a q_0] \longrightarrow a [q_0 a q_1] [q_1 a q_0]$

$[q_0 a q_1] \longrightarrow a [q_0 a q_0] [q_0 a q_1]$

$[q_0 a q_1] \longrightarrow a [q_0 a q_1] [q_1 a q_1]$

(iii) $\delta(q_0, b, a) = (q_1, a)$ (stack contains only one symbol)

two productions are possible

$2^1 - 2$ and power is 1 then right side is only one terminal

$[q_0\ a\ q_0] \longrightarrow b[q_1\ a\ q_0]$

$[q_0\ a\ q_1] \longrightarrow b[q_1\ a\ q_1]$

states

(iv) $\delta(q_1, b, a) = (q_1, a)$

$[q_1\ a\ q_0] \longrightarrow b\ [q_1\ a\ q_0]$

$[q_1\ a\ q_1] \longrightarrow b\ [q_1\ a\ q_1]$

(v) $\delta(q_1, a, a) = (q_1, \epsilon)$ [∵ stack is empty]

$[q_1\ a\ q_1] \longrightarrow a$

(vi) $\delta(q_1, \epsilon, z_0) = (q_1, \epsilon)$

$[q_1\ z_0\ q_1] \longrightarrow \epsilon$

These are the set of productions for CFG1

Ex 2 Convert the following PDA in to CFG1.

$P = (\{q, p\}, \{0, 1\}, \{X, Z\}, \delta, q, Z)$

$\delta(q, 1, Z) = (q, XZ)$

$\delta(q, 1, X) = (q, XX)$

$\delta(q, \epsilon, X) = (q, \epsilon)$

$\delta(q, 0, X) = (p, X)$

$\delta(p, 1, X) = (p, \epsilon)$

$\delta(p, 0, Z) = (q, Z)$

FORMAL LANGUAGES AND AUTOMATA THEORY 10CS56

The Language of a Grammar

If G(V,T,P,S) is a CFG, the language of G, denoted by L(G), is the set of terminal strings that have derivations from the start symbol.

L(G) = {w in T | S → w}

Sentential Forms

Derivations from the start symbol produce strings that have a special role called "sentential forms". That is if G = (V, T, P, S) is a CFG, then any string in (V T)* such that S → is a sentential form. If S → , then is a left – sentential form, and if S → , then is a right – sentential form. Note that the language L(G) is those sentential forms that are in T*; that is they consist solely of terminals.

For example, E * (I + E) is a sentential form, since there is a
derivation E → E * E → E * (E) → E * (E + E) → E * (I + E)
However this derivation is neither leftmost nor rightmost, since at the last step, the middle E is replaced.
As an example of a left - sentential form, consider a * E, with the leftmost derivation. E → E * E → I * E → a * E
Additionally, the derivation
E → E * E → E * (E) → E * (E +
E) Shows that
E * (E + E) is a right - sentential form.

: Applications of Context – Free Grammars

- Parsers
- The YACC Parser Generator
- Markup Languages
- XML and Document typr definitions

The YACC Parser Generator

E→ E+E | E*E |
(E)|id %{ #include
<stdio.h> %}
%token ID id
%%
Exp : id { $$ = $1 ; printf ("result is %d\n", $1);}
| Exp „+" Exp {$$ = $1 + $3;}
| Exp „*" Exp {$$ = $1 * $3; }
| „(„ Exp „)" {$$ = $2; }
;

```
%%

int main (void) {
return yyparse ( );
}
void yyerror (char *s) {
fprintf (stderr, "%s\n", s);
}
%{
#include "y.tab.h"
%}
%%
[0-9]+          {yylval.ID = atoi(yytext); return id;}
[ \t \n]          ;
[+ * ( )]                 {return yytext[0];}
.                         {ECHO; yyerror ("unexpected character");}
%%
```

Example 2:

```
%{
 #include <stdio.h>
%}
%start line
%token <a_number> number
%type <a_number> exp term factor
%%
line : exp ';' {printf ("result is %d\n", $1);}
;
exp : term {$$ = $1;}
        | exp '+' term {$$ = $1 + $3;}
        | exp '-' term {$$ = $1 - $3;}
term : factor {$$ = $1;}
        | term '*' factor {$$ = $1 * $3;}
        | term '/' factor {$$ = $1 / $3;}
;
factor : number {$$ = $1;}
| '(' exp ')' {$$ = $2;}
;
 %%
 int main (void) {
return yyparse ( );
}
void yyerror (char *s) {
fprintf (stderr, "%s\n", s);
}
%{
#include "y.tab.h"
%}
%%
```

```
[0-9]+ {yylval.a_number = atoi(yytext); return number;}
[ \t\n] ;
[-+*/();] {return yytext[0];}
.              {ECHO; yyerror ("unexpected character");}
%%
```

Markup Languages

Functions

- Creating links between documents
- Describing the format of the document

Example

The Things I *hate*

1. Moldy bread
2. People who drive too slow In the fast lane

HTML Source

```
<P> The things I
<EM>hate</EM>: <OL>
<LI> Moldy bread
<LI>People who drive too
slow In the fast lane
</OL>
```

HTML Grammar

- Char a | A | ...
- Text e | Char Text
- Doc e | Element Doc
- Element Text |
  <EM> Doc </EM>|
  <p> Doc |
  <OL> List </OL>| ...

5. List-Item <LI> Doc
6. List e | List-Item List Start symbol

XML and Document type definitions.

1. A→E1,E2.
   A→BC
   B→E1
   C→E2
2. A→E1 | E2.
   A→E1
   A→E2
3. A→(E1)*
   A→BA
   A→ ∈
   B→E1
4. A→(E1)+
   A→BA
   A→B
   B→E1
5. A→(E1)?
   A→ ∈
   A→E1

4.4:Ambiguity

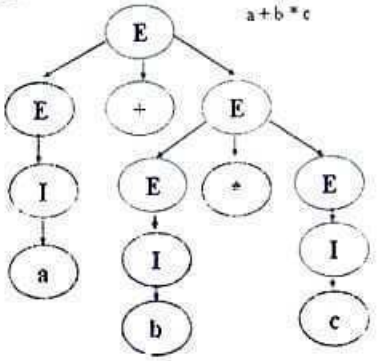
A context - free grammar G is said to be ambiguous if there exists some w L(G) which has at least two distinct derivation trees. Alternatively, ambiguity implies the existence of two or more left most or rightmost derivations.

Ex:-

Consider the grammar G=(V,T,E,P) with V={E,I}, T={a,b,c,+,*,(,)}, and productions. E→I,
E→E+E,
E→E*E,
E→(E),
I→a|b|c

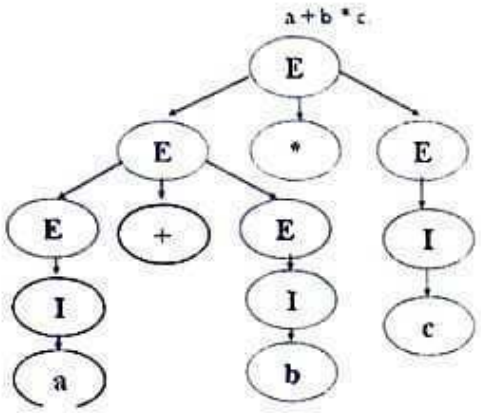
Consider two derivation trees for a + b * c.

Tree I
Let a=5, b=6, c=7
The value for Tree I will be 47



Tree II

Let a=5, b=6, c=7
The value for Tree II
will be 77



Now unambiguous grammar for the above
Example:

E→T, T→F, F→I, E→E+T, T→T*F,
F→(E), I→a|b|c

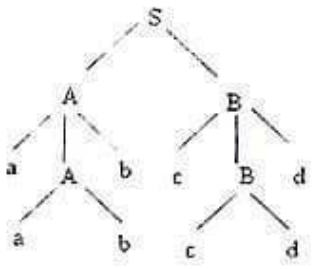
Inherent Ambiguity

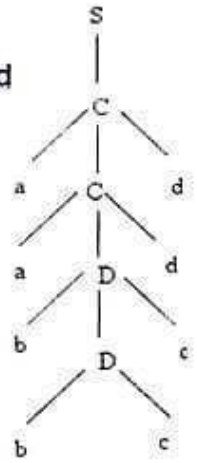
A CFL L is said to be inherently ambiguous if all its grammars are ambiguous Example:
Condider the Grammar for string

aabbccdd S→AB | C
A→ aAb | ab
B→cBd | cd
C→ aCd | aDd
D->bDc | bc

Parse tree for string aabbccdd

Parse tree for string aabbccdd





ASSIGNMENT QUESTIONS

1) The following grammar generates the language of RE
   $0^*1(0+1)^*$
   $S \rightarrow A|B$
   $A \rightarrow 0A|$
   $B \rightarrow 0B|1B|$
   Give leftmost and rightmost derivations of the following strings
   a) 00101 b) 1001 c) 00011

2) Consider the grammar
   $S \rightarrow aS|aSbS|$
   Show that deviation for the string aab is ambiguous

3) The following grammar generates the language of RE
   $0^*1(0+1)^*$
   $S \rightarrow A|B$
   $A \rightarrow 0A|$
   $B \rightarrow 0B|1B|$
   Give leftmost and rightmost derivations of the following strings
   a) 00101 b) 1001 c) 00011

4) Consider the grammar
   $S \rightarrow aS|aSbS|S$

   Show that deviation for the string aab is ambiguous

UNIT - IV

NORMAL FORMS FOR CONTEXT - FREE GRAMMARS

Minimization of context free grammar (CFG):

The CFG may contain some extra non terminals which are useless. This causes the length of grammar to be increased. Hence we delete some useless symbols which is then called as simplification of grammar.

The properties of reduced grammar are given below:

1) Each terminal and non-terminal of grammar appears in the derivation of some word in the language.

2) There should not be any productions as $x \rightarrow y$ where x and y are non-terminals.

3) If ϵ is not in the language 'L' then there need not be the production $x \rightarrow \epsilon$.

Elimination of useless symbols (or) unreachable symbols:

→ A symbol which is not involved in any string generation by the grammar is called as useless symbol.

(or)

→ A symbol which is not reachable from the starting symbol is called as unreachable symbols.

$S \overset{*}{\Rightarrow} \alpha p \beta$ } Here p is said to be useful symbol

$\alpha p \beta \overset{*}{\Rightarrow} w$

Ex: ① $A \longrightarrow aB \mid b$

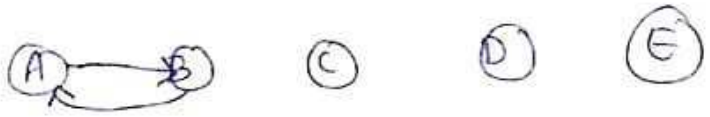
$B \longrightarrow bA \mid b$

$C \longrightarrow ec \mid f$

$D \longrightarrow d \mid dE$

$E \longrightarrow aE$

Soln: Here, the production 'c' is useless as it is unreachable from starting symbol 'A', and the production $E \longrightarrow aE$ cannot be terminated. So, it is also useless. Hence we can delete the useless symbols.



② $G_1 = (V, T, P, S)$ where $V = \{S, T, X\}$ $T = \{0, 1\}$

$S \longrightarrow 0T \mid 1T \mid x \mid 0 \mid 1$

$T \longrightarrow 00$

Soln: Here X is not having any further derivations Hence we delete X and associated if exists any

$\therefore S \longrightarrow 0T \mid 1T \mid x \mid 0 \mid 1$ $\quad S \longrightarrow 0T \mid 1T \mid 0 \mid 1$

$T \longrightarrow 00$ $\quad \Longrightarrow \quad T \longrightarrow 00$

③ Consider the CFG $G = (V, T, P, S)$ where $V = \{S, A, B\}$ $T = \{0, 1\}$, $P = \{$

$S \longrightarrow A11B \mid 11A$

$S \longrightarrow 1B \mid 11$

$A \longrightarrow 0$

$B \longrightarrow BB \}$

Soln Here B doesnot derive any terminal symbols so, we delete B and associated strings i.e., $B \to BB$ and $S \to A11B$

The resultant will be $P = \{ S \to 11A \mid S \to 11 \mid A \to 0 \} \Rightarrow \{ S \to 11A/11 \mid n \to 0 \}$

Elimination of ϵ productions from grammar:

If the grammar contains 'ϵ' which means it has no value, we remove it from the production rules.

→ 'ϵ' productions for a non-terminal is replaced by introducing a new production for the given N.T without ϵ Non Terminal.

[Note: only one 'ϵ' can be eliminated at a time]

Ex Remove 'ϵ' production from the following CFG, by preserving meaning of it.

$S \to aA \mid bBb \mid ab$

$A \to aA \mid \epsilon$

$B \to bBa \mid \epsilon$

Soln: $S \to aA \mid bBb \mid ab$

$A \to aA \mid \epsilon$

$B \to bBa \mid \epsilon$

$\xrightarrow{\text{Removing } \epsilon \text{ from } A}$

$S \to aA \mid bBb \mid ab \mid a$

$A \to aA \mid a$

$B \to bBa \mid \epsilon$

$\Downarrow$ Removing 'ϵ' from B

$S \to aA \mid bBb \mid ab \mid a \mid bb$

$A \to aA \mid a$

$B \to bBa \mid ba$

Ex (2) $A \longrightarrow 0B1 \mid 1B1$

$B \longrightarrow 0B \mid 1B \mid \epsilon$

Solution:

$A \longrightarrow 0B1 \mid 1B1$

$B \longrightarrow 0B \mid 1B \mid a \mid 1$

Ex.3 $S \longrightarrow aSa$

$S \longrightarrow bSb$

$S \longrightarrow \epsilon$

Sol: $S \longrightarrow aSa \mid bSb \mid aa \mid bb$

Elimination of unit productions:

A production is said to be unit if a non terminal derives another non-terminal alone. Ex: $X \longrightarrow Y$.

→ A unit production can be eliminated by replacing R.H.S non terminal with all its productions.

Ex

$E \longrightarrow E+T \mid T$

$T \longrightarrow T*F \mid F$

$F \longrightarrow (E) \mid id$

Solution:

$E \longrightarrow E+T$

$E \longrightarrow T$

$T \longrightarrow T*F$

$T \longrightarrow F$

$F \longrightarrow (E)$

$F \longrightarrow id$

Here $E \longrightarrow T$ and $T \longrightarrow F$ are unit productions

$\Longrightarrow$

$E \longrightarrow E+T$

$E \longrightarrow T$

$T \longrightarrow T*F$

$T \longrightarrow (E) \mid id$

$F \longrightarrow (E)$

$F \longrightarrow id$

Here $T \longrightarrow F$ is eliminated by replacing F productions

$\Longrightarrow$

$E \longrightarrow E+T$

$E \longrightarrow T*F \mid (E) \mid id$

$T \longrightarrow T*F$

$T \longrightarrow (E) \mid id$

$F \longrightarrow (E)$

$F \longrightarrow id$

Here $E \longrightarrow T$ is eliminated by replacing T productions

Normal Forms:

At the right hand of production rules we have different combinations of terminal & non-terminal. we need to arrange them in a specific format ie, fixed number of terminals. This process is known as "normalization"

→ These are 2 types of Normal Forms They are :-

(1) Chomsky Normal Form (CNF)

(2) Greibach Normal Form (GNF)

1) Chomsky Normal Form:

A Grammar is said to be in CNF if every production of the form is $A \longrightarrow XY$ or $A \longrightarrow a$ ie, In right hand side either two non-terminals or only one terminal is allowed.

Non-terminal ⟶ Nonterminal . Non terminal
Non-terminal ⟶ Terminal .

Conversion of context Free grammar to chomsky's Normal Form:

Before converting the CFG to CNF we reduce the gramma by removing useless symbols and ϵ productions and removing unit productions.

Step 1: Remove ϵ productions.

Step 2: Remove unit productions.

Step 3 Remove useless symbols

Step4: Replace terminal symbol with new variable.

Step5: Break the symbols having length more than 2.

Ex: Convert the following CFG to CNF

$S \longrightarrow AaBb \mid abA$

$A \longrightarrow ab \mid Bb$

$B \longrightarrow bA \mid b$

Solution: Given grammar is

$S \longrightarrow AaBb$

$S \longrightarrow abA$

$A \longrightarrow ab$

$A \longrightarrow Bb$

$B \longrightarrow bA$

$B \longrightarrow b$

The given CFG does not contain any useless symbols, ϵ & unit productions. Now replace $X \to a$ & $Y \to b$

$S \longrightarrow AXBY$

$S \longrightarrow XYA$

$A \longrightarrow XY$

$A \longrightarrow BY$

$B \longrightarrow YA$

$B \longrightarrow b$

$\Longrightarrow$

Replace

$AX \longrightarrow P$

$BY \longrightarrow Q$

$XY \longrightarrow R$

$S \longrightarrow PQ$

$P \longrightarrow AX$

$Q \longrightarrow BY$

$S \longrightarrow RA$

$R \longrightarrow XY$

$A \longrightarrow XY$

$A \longrightarrow BY$

$B \longrightarrow YA$

$B \longrightarrow b$

$X \longrightarrow a$

$Y \longrightarrow b$

Ex:2 Convert the following CFG to CNF

$S \rightarrow ABA$

$A \rightarrow aA \mid \epsilon$

$B \rightarrow bB \mid \epsilon$

Solution The given CFG contains 'ϵ' productions. so, remove 'ϵ' productions from the grammar.

→ Eliminating 'ϵ' from A

$S \rightarrow ABA \mid BA \mid AB \mid B$

$A \rightarrow aA \mid a$

$B \rightarrow bB \mid \epsilon$

→ Eliminating 'ϵ' from B

$S \rightarrow ABA \mid BA \mid AB \mid B \mid AA \mid A$

$A \rightarrow aA \mid a$

$B \rightarrow bB \mid b$

Now, the grammar contain unit productions. so eliminate them by replacing R.H.S N.T with all the productions

$S \rightarrow ABA \mid BA \mid AB \mid aA \mid a \mid b \mid bB$

$A \rightarrow aA \mid a$

$B \rightarrow bB \mid b$

Now convert it into CNF

replace $X \rightarrow AB$

$A_1 \rightarrow a$

$B_1 \rightarrow b$

$S \longrightarrow XA$

$S \longrightarrow BA$

$S \longrightarrow AB$

$S \longrightarrow A_1A$

$S \longrightarrow a$

$S \longrightarrow B_1B$

$S \longrightarrow b$

$A \longrightarrow A_1A$

$A \longrightarrow a$

$B \longrightarrow B_1B$

$B \longrightarrow b$

$X \longrightarrow AB$

$A_1 \longrightarrow a$

$B_1 \longrightarrow b$

Greibech Normal Form

A Grammar is said to be in GNF if every production of the form $A \longrightarrow a\alpha$ ie, a terminal followed by any number of non-terminals.

steps to convert CFG to GNF:

step1: check if the given CFG has any unit productions or Null productions and useless symbols, remove if there are any.

step2: check whether the CFG is already in CNF and convert it to CNF if it is not.

step 3: change the names of the non-terminals symbols in to some A_i in ascending order of i

step 4: After the rules so that the non-terminals are in ascending order, such that if the production is of the form $A_i \longrightarrow A_jX$ then $i < j$ and should never be $i \geq j$ [left recursive grammar: A production of the form $A \longrightarrow A\alpha$ is called left recursive as the left hand side variable appears as first symbol on right hand side]

step 5: Remove left recursion, introduce a new variable to remove the left recursion.

Example: Convert the following CFG to GNF

$S \longrightarrow CA \mid BB$

$B \longrightarrow b \mid SB$

$C \longrightarrow b$

$A \longrightarrow a$

Solution: changing the non terminal symbols in Ascending order 4.9

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

we get ⟹

$A_1 \longrightarrow A_2A_3 \mid A_4A_4$
$A_4 \longrightarrow b \mid A_1A_4$
$A_2 \longrightarrow b$
$A_3 \longrightarrow a$

Here $A_4 \longrightarrow b \mid A_1A_4$ is violating the rule-4 $[A_4 > A_1]$ i.e., $i > j$. so replace A_1 with its productions.

$A_1 \longrightarrow A_2A_3 \mid A_4A_4$
$A_4 \longrightarrow b \mid A_2A_3A_4 \mid A_4A_4A_4$
$A_2 \longrightarrow b$
$A_3 \longrightarrow a$

→ Againe A_4 production is violating the rule. so replace it with A_2 productions.

$A_4 \longrightarrow b \mid bA_3A_4 \mid \underbrace{A_4A_4A_4}$
left recursion.

→ Introduce a new variable.

$Z \longrightarrow A_4A_4Z \mid A_4A_4$ then

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

The gramma is

$A_1 \longrightarrow A_2A_3 \mid A_4A_4$
$A_4 \longrightarrow b \mid bA_3A_4 \mid bZ \mid bA_3A_4Z$
$Z \longrightarrow A_4A_4 \mid A_4A_4Z$
$A_2 \longrightarrow b$
$A_3 \longrightarrow a$.

→ Here A_1 production is not in the GNF. So replace with their respective productions.

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

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

$Z \longrightarrow bA_4 \mid bA_3A_4A_4 \mid bZA_4 \mid bA_3A_4ZA_4 \mid bA_4Z \mid bA_3A_4A_4Z \mid$

$bZA_4Z \mid bA_3A_4ZA_4Z$

$A_2 \longrightarrow b$

$A_3 \longrightarrow a$

Ex:② Conversion of CFG to GNF

$S \longrightarrow AA \mid a$

$A \longrightarrow SS \mid b$

Solution Given grammar is already in simplified form. There are no null productions, no unit productions & no useless symbols in the grammar.

step1: Now we proceed for renaming of variables. Variables S & A are renamed as A_1 & A_2.

So productions are.

$A_1 \longrightarrow A_2A_2 \mid a$

$A_2 \longrightarrow A_1A_1 \mid b$.

step2. Now every production should be of the form

$A_i \longrightarrow A_j\alpha \quad (i \not< j)$

'A_2' production does not satisfy the rule, hence substitute A_2 with its production.

Therefore

$A_2 \longrightarrow A_2A_2A_1 \mid aA_1 \mid b$

The resulting productions are

$A_1 \longrightarrow A_2A_2 \mid a$

$A_2 \longrightarrow A_2A_2A_1 \mid aA_1 \mid b$

step3: Removing left recursion

A_2 production of the form $A_2 \longrightarrow A_2A_2A_1 \mid aA_1 \mid b$ contains left recursion. So remove by introducing a new non terminal B

Therefore

$A_2 \longrightarrow A_2A_2A_1 \mid aA_1 \mid b$

can be written as

$A_2 \longrightarrow aA_1B \mid bB \mid aA_1 \mid b$

$B \longrightarrow A_2A_1B \mid A_2A_1$

[∵ If left recursive grammar is
$A \longrightarrow A\alpha \mid b$
can be written as
$A \longrightarrow b \mid bB$
$B \longrightarrow \alpha \mid \alpha B$]

Resulting productions are

$A_1 \longrightarrow A_2A_2 \mid a$

$A_2 \longrightarrow aA_1B \mid bB \mid aA_1 \mid b$

$B \longrightarrow A_2AB \mid A_2A_1$

Step 4: A_2 productions are in GNF, A_1 & B productions need to be converted to GNF

$A_1 \longrightarrow A_2A_2 \mid a$ replaced A_2 with its production

$A_1 \longrightarrow aA_1BA_2 \mid bBA_2 \mid aA_1A_2 \mid bA_2 \mid a$

Now $B \longrightarrow A_2A_1B \mid A_2A_1$ replace A_2 with its productions

$B \longrightarrow aA_1BA_1B \mid bBA_1B \mid aA_1A_1B \mid bA_1B \mid aA_1BA_1 \mid bBA_1 \mid aA_1A_1 \mid bA_1$

Final grammar in GNF is

$A_1 \longrightarrow aA_1BA_2 \mid bBA_2 \mid aA_1A_2 \mid bA_2 \mid a$

$A_2 \longrightarrow aA_1B \mid bB \mid aA_1 \mid b$

$B \longrightarrow aA_1BA_1B \mid bBA_1B \mid aA_1A_1B \mid bA_1B \mid aA_1BA_1 \mid bBA_1 \mid aA_1A_1 \mid bA_1$.

Pumping lemma for context free languages:

⇒ To prove certain languages are not context free

- let L be a CFL.
- let n be a constant.
- Any string z in L, $|z| \geq n$
- split $z = uvwxy$ such that
  - (i) $|vwx| \leq n$
  - (ii) $vx \neq \epsilon$ or $|vx| \geq 1$
  - (iii) for all $i \geq 0$, $uv^iwx^iy \in L$

Ex: Show that $L = \{a^n b^n c^n \mid n \geq 1\}$ is not a CFL.

Proof: - let L be a CFL
- n be a constant
- let $z = a^n b^n c^n$ $|z| \geq n$
- split L, hence we get

$z = uvwxy$

$u = a^n \quad vwx = b^n \quad y = c^n$

(i) $|vwx| \leq n$

(ii) $vwx = b^n$

$|vx| = b^{n-m}$

for all $i \geq 0 \quad uv^iwx^iy$

$uv^iwx^iy = uv^{i-1}wx\,x^{i-1}y$

$= uvw\,(vx)^{i-1}xy$

$= uvw\,(b^{n-m})^{i-1}xy$

$= a^n b^n (b^{n-m})^{i-1} c^n$

$= a^n b^{ni-mi-n+m} c^n$

$= a^n b^m c^n$

not in CFL

∴ Hence language L is not CFL.

Eg: Show that $L = \{a^n b^n c^n \mid n \geq 0\}$ is not context free.

Solution:

we divide string in to parts $uvwxy$

(we(i) let $n = 4 \quad S = a^4 b^4 c^4$

v & x contain one type of symbol.

$\underbrace{aaaa}_{u}\underbrace{bbbb}_{v}\underbrace{cccc}_{w\,x\,y}$

let $k = 2 \quad uv^2wx^2y$

$\Rightarrow a(aa)^2 abbbbcc^2c$

$\Rightarrow aaaaaa\,bbbb\,ccccc$

$\Rightarrow a^6 b^4 c^5$

The resultant string is not satisfying the condition

$\therefore a^6 b^4 c^5 \notin L$

Given language is not CFL.

Ex 2: Prove that the following language is CFL or not. $L = \{a^i b^i c^i \mid i \geq 0\}$.

Solution: $L = \{\epsilon, abc, aabbcc, aaabbbccc, \ldots\}$

$n = 4$

$z = \underbrace{aa}_{uv}\underbrace{bb}_{w}\underbrace{cc}_{x\,y} \qquad |z| = 4 \geq n$

$|aabb| \leq n$

$$z = \underbrace{a}_{u}\underbrace{a}_{v}\underbrace{bb}_{w}\underbrace{c}_{x}\underbrace{c}_{y}$$

For $i=1 \Rightarrow z = uv^1wx^1y = aabbcc \in L$

For $i=2 \Rightarrow z = uv^2wx^2y$

$= uvvwxxy$

$= aaabbccc \notin L$

∴ Hence given language is not CFL.

Closure properties of context free languages:

→ Regular languages are closed under every properties whereas CFL are not closed under CID (Complement Intersection, Difference).

→ CFL are closed under union, concatenation, Kleen closure, Reverse, Homomorphism, Inverse homomorphism, substitution.

union: CFL are said to be closed under union if L_1 & L_2 are two C.F.L then $L_1 \cup L_2$ must be a CFL.

Ex: $L_1 = \{a^n b^n c^m \mid m>=0 \ \& \ n>=0\}$ and

$L_2 = \{a^n b^m c^m \mid n>=0 \ \& \ m>=0\}$

$L_3 = L_1 \cup L_2 = \{a^n b^n c^m \cup a^n b^m c^m \mid n>=0, m>=0\}$ is CFL.

∴ CFL is closed under union.

Concatenation: CFL are said to be closed under concatenation if L_1 & L_2 are two CFL's then $L_1 \cdot L_2$ must be a CFL.

Ex: $L_1 = \{ a^n b^n \mid n \geq 0 \}$ and $L_2 = \{ c^m d^m \mid m \geq 0 \}$

$L_3 = L_1 \cdot L_2 = \{ a^n b^n c^m d^m \mid m \geq 0 \text{ \& } n \geq 0 \}$ is also CFL.

∴ CFL is closed under concatenation.

Kleen closure: If L_1 is CFL, its kleen closure L_1^* is also CFL.

$L_1 = \{ a^n b^n \mid n >= 0 \}$

$L_1^* = \{ a^n b^n \mid n \geq 0 \}^*$ is also kleen closure.

Intersection: If L_1 & L_2 are CFL then $L_1 \cap L_2$ need not to be CFL.

$L_1 = \{ a^n b^n c^m \mid n \geq 0 \text{ \& } m \geq 0 \}$

$L_2 = \{ a^m b^n c^n \mid n \geq 0 \text{ \& } m \geq 0 \}$

$L_3 = L_1 \cap L_2 = \{ a^n b^n c^n \mid n >= 0 \}$ need not be CFL.

complement: complementation of CFL need not to be in CFL.

Ex: $\Sigma^* - L_1$

Hence, CFL is not closed under intersection and complementation.

Deterministic context free language: DCFL are closed under complementation & inverse Homomorphism.

Decision properties of CFL:

Membership:

→ unlike finite automatas, we can't just run the string through the machine and see where it goes since PDA's are non-deterministic.

→ Must consider all possible paths.

→ Instead, start with your grammar in CNF.

→ The proof of pumping lemma states that the longest derivation path of a string of size n will be $2n-1$.

→ systematically, generate all the strings, if we derive the string is in the language.

Emptyness:

→ Remove useless symbols & productions.

→ If S is useless, then L(G) is empty.

Finiteness:

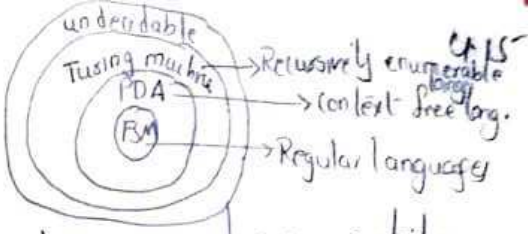
→ The language is infinite if there is a string x with length between n & 2n.

→ CFL $n = 2^{P+1}$ P is the non-terminal.

(n: no. of states.)

→ If Language L is passed through membership algorithm is infinite else L is finite.

Turing machine



Introduction to Turing machine:

(1) It has an external memory which remembers arbitrary long sequence of input (where as F.A reads only current symbol).

(2) It has unlimited memory capability

(3) The model has facility by which the input at the left or right on the tape can be read easily.

(4) The machine can produce a certain output based on its input.

Formal definition of Turing machine:

The formal notation we use for Turing machine (TM) is similar to that used for finite automata. we represent TM by 7-tuple.

$$M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$$

Q - Finite set of states of the finite control.

Σ - Finite set of input symbols.

T - Tape symbols.

δ - Transition function (or) mapping function.

$\delta(q, X)$ is defined as triple (P, Y, D) where

P - next state in Q.

Y - is symbol in T

D - is direction either L or R ie, left or right direction in which head moves

q_0 - The start state.

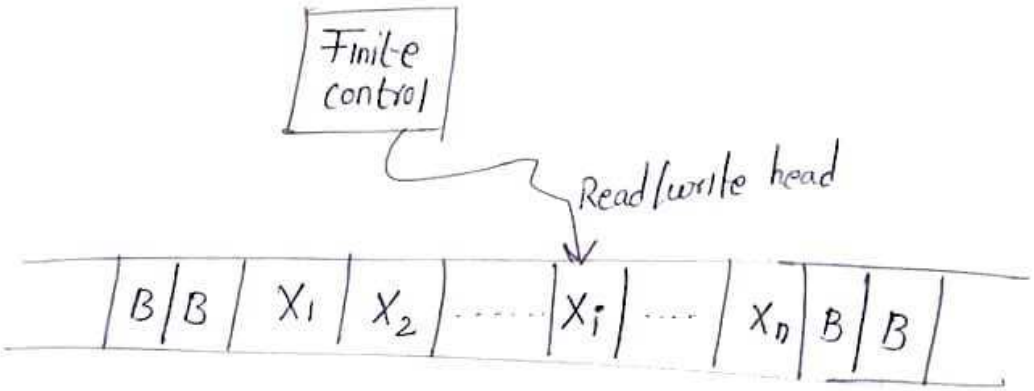
B - The blank symbol, this symbol is in T, but not in Σ.

F - Set of final or accepting states.

Block diagram of TM:

→ Machine consists of a finite control, which can be in any of the finite set of states.

→ A tape divided in to squares (or) cells, each cell hold any one of the finite numbers of symbols.



Turing machine

→ The tape is infinite length on the left side as well as right side.

→ The tape is divided in to certain cells, it can hold input symbol and also blank.

→ The reading head always points on the left most cell of the tape.

→ Reading If the TM reaches the final state, the input string is accepted, otherwise rejected.

Instantaneous descriptions for Turing machines

→ An instantaneous description is used to show the acceptance of some string by automata.

→ An ID of a TM is a string of the form $\alpha_1 q \alpha_2$ where α_1 & $\alpha_2 \in \Gamma^*$ and $q \in K$.

→ We use the string $X_1 X_2 \cdots X_{i-1} q X_i X_{i+1} \cdots X_n$ to represent an ID in which

- q is the state of the Turing machine.
- The tape head is scanning the i^{th} symbol from the left
- $X_1 X_2 \cdots X_n$ is the portion of the tape between the leftmost & the rightmost nonblank.

→ We describe instantaneous description by using tanstyle notation ($\vdash$), $\vdash^*$ represents 0 or more moves.

case 1: leftmove

⇒ Suppose $\delta(q, X_i) = (p, Y, L)$ (current state, current tape symbol) i.e., the next move is leftward then

$$X_1 X_2 \cdots X_{i-1} q X_i X_{i+1} \cdots X_n \vdash_M X_1 X_2 \cdots X_{i-2} p X_{i-1} Y X_{i+1} \cdots X_n$$

(∵ Here q is present state & X_i is the symbol currently scanned by tape head means we need to change q to p and X_i is replaced by Y & move is left mode

(i) if $i=1$ then M moves to the blank to the left of X_1.

So $X_1 X_2 \cdots X_n \vdash_M pBY X_2 \cdots X_n$

(i) if $i = n$ & $Y = B$ then the Symbol B written over X_n joins the infinite sequence of trailing blanks and does not appear in next ID.

then $X_1 X_2 \cdots X_{n-1} q X_n \vdash X_1 X_2 \cdots X_{n-2} P X_{n-1}$

case 2 : right move :

$\delta(q, X_i) = (P, Y, R)$ the next move is rightward.

then

$X_1 X_2 \cdots X_{i-1} q X_i X_{i+1} \cdots X_n \vdash_M X_1 X_2 \cdots X_{i-1} Y P X_{i+1} \cdots X_n$

(∵ Now tape head is scanning X_i, here move is right move so it has to scan X_{i+1})

Again there are two important exceptions.

(i) If $i = n$ then $i+1$ st cell holds a blank, and that cell was not part of the previous ID, Thus we instead have $X_1 X_2 \cdots X_{n-1} q X_n \vdash_M X_1 X_2 \cdots X_{n-1} Y P B$

(ii) If $i = 1$ & $Y = B$ then the symbol B written over X_1 joins the infinite sequence of leading blanks and does not appear in the next ID. Thus

$q X_1 X_2 \cdots X_n \vdash_M P X_2 \cdots X_n$

Ex: Design a turing machine for $L = \{a^n b^n / n \geq 1\}$

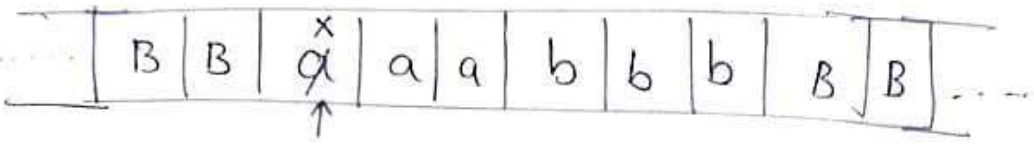
Soln: Given language $L = \{ab, aabb, aaabbb, \ldots\}$

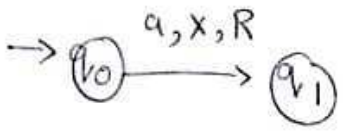
→ Here the concept is if we read 'a', then change a to 'X', then move head towards right till we find 'b' (ie. a a b b, X Y) then change b to Y.

→ Next move towards left till we get 'X', then move the head towards right applying the same concept.

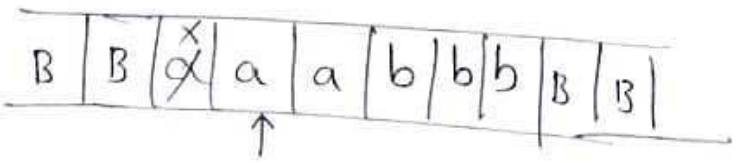
ie., a a b b
X X Y Y

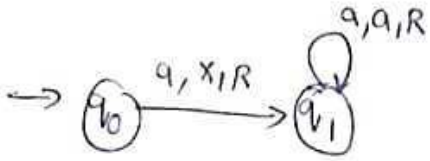
→ Let the string is



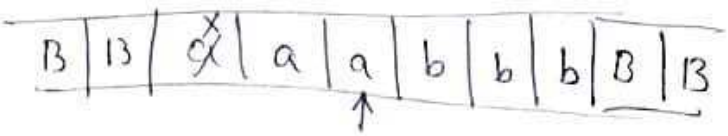


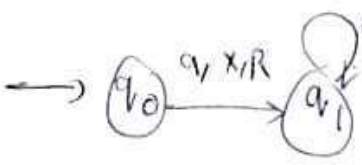
Step2





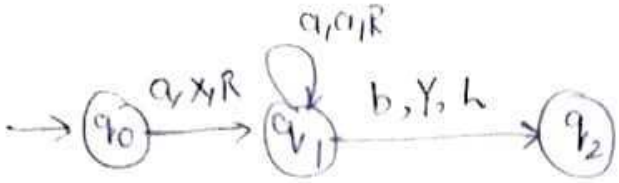
Step3





step4.

B	B	X	a	a	Y	b	b	B	B
				↑					

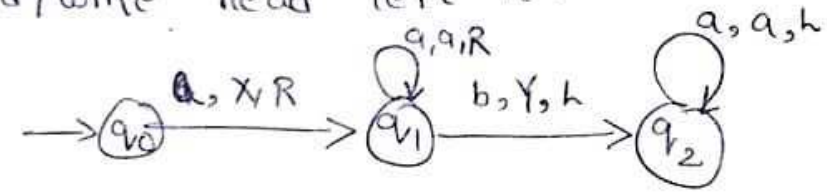


Here on reading b, then replace it with Y and move the read/write head towards left till we get X.

step5:

B	B	X	a	a	Y	b	b	B	B
				↑					

Now on reading a, no need to change state and move read/write head left ie.



step6:

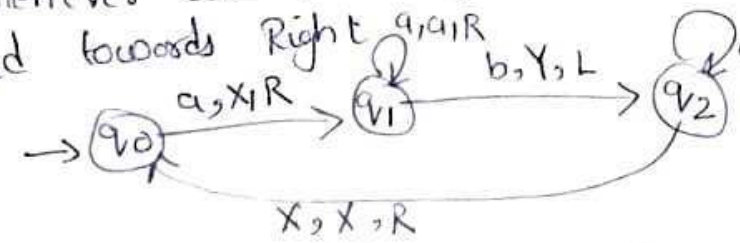
B	B	X	a	a	Y	b	b	B	B
			↑						

Now also on reading a, no need to change state, just move towards left.

step7.

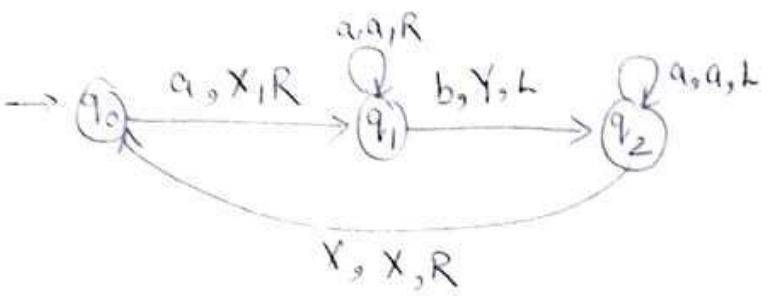
B	B	X	a	a	Y	b	b	B	B
		↑							

whenever we read X, no need to change X, but move read/write head towards Right



steps

B	B	X	a	a	Y	b	b	B	B
			↑						



Here a is applied on q_0, so 'a' is replaced with X and move towards Right

Step 9

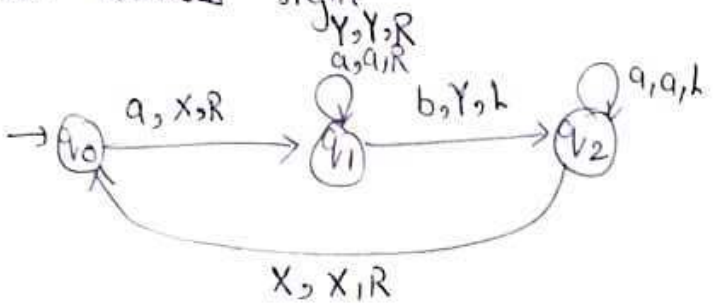
B	B	X	a	X	a	Y	b	b	B	B
				↑						

Now in state q_1 on reading a, no need to change state just move towards right.

Step 10:

B	B	X	a	X	a	Y	b	b	B	B
					↑					

Now on reading Y & a, no need to change Y, just move read/write head towards right.

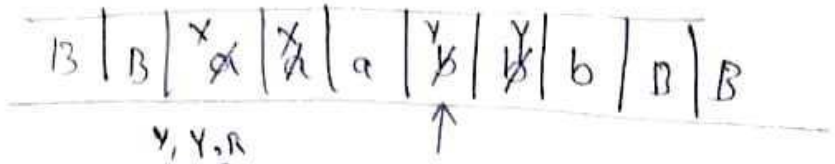


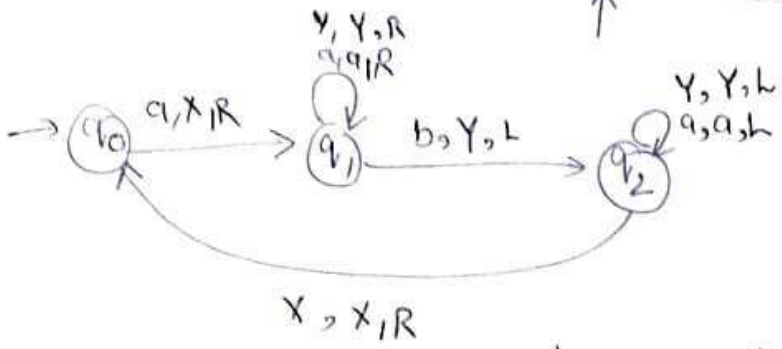
Step 11:

B	B	X	X	a	Y	Y	b	B	B
						↑			

Now at state q_1 on reading b, change state to Y and move read/write head towards L.

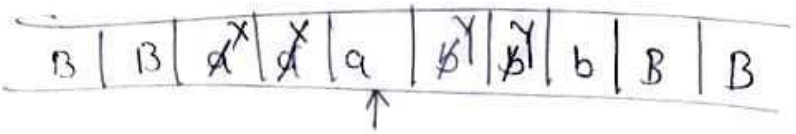
step 12:





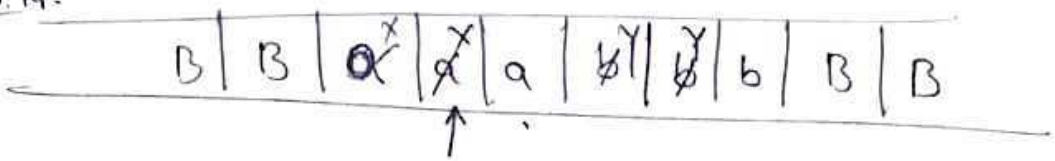
Here at state q_2 on reading Y, dont change state, just move towards left.

step 13.



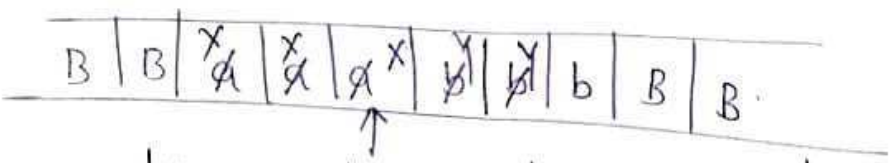
Here at state q_2 on read a, don't change charge state just move towards left.

step 14:



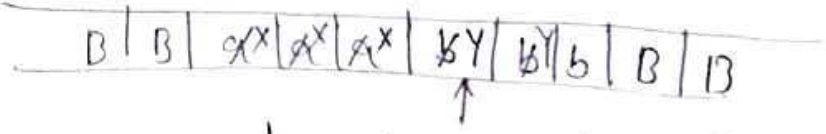
If we read X on q_2, do'nt change X, but move read/write head towards R, leading to q_0.

step 15.



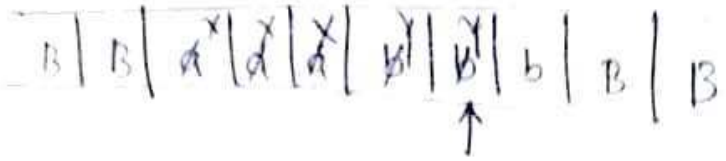
At q_0 on reading a, change content to X and move read/write head to words right

step 16:



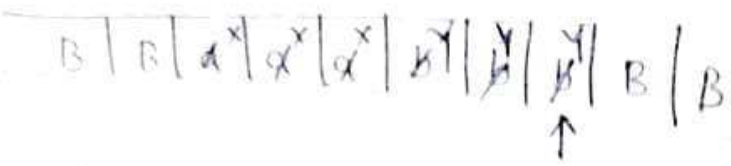
Now at q_1 on reading Y, dont change the content, Just move read/write head towards right.

step 17



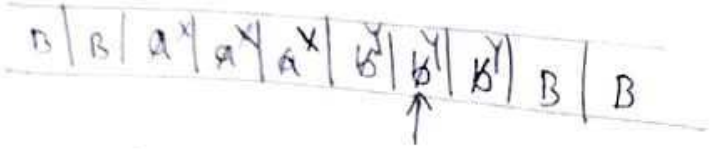
Same as before step.

step 18



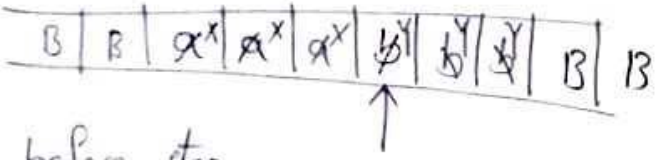
Now at q_1 on reading b, change the content to Y and state is changed to q_2 and move towards L.

step 19



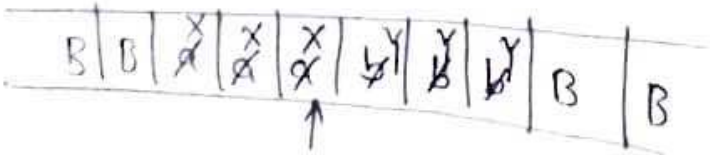
Same as before step

step 20

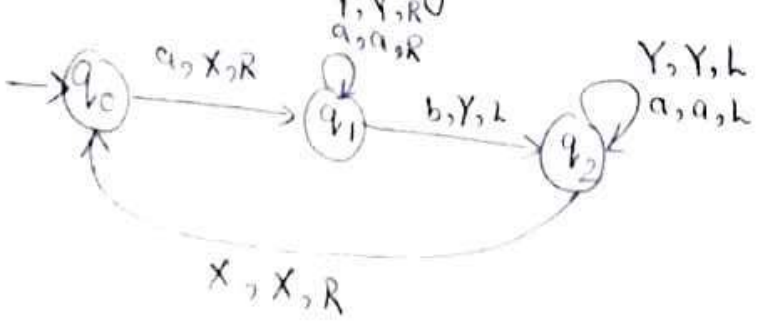


Same as before step

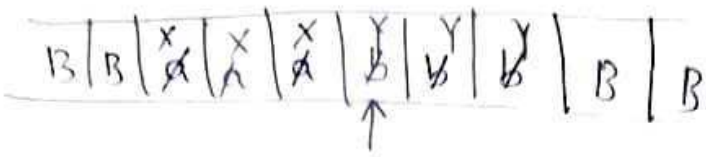
step 21



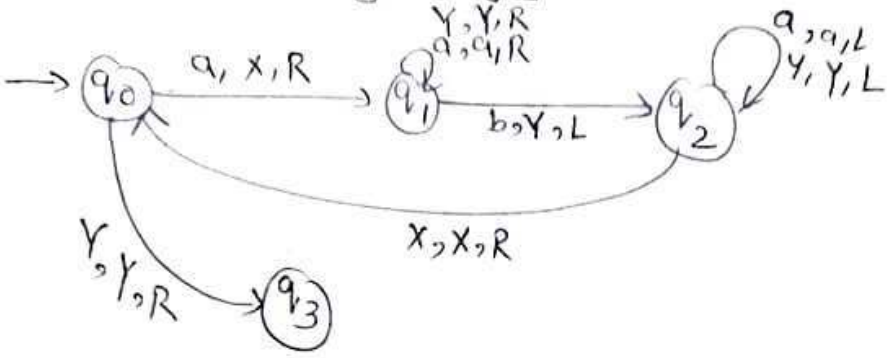
Now at q_2 on reading X move to q_0, and move towards right.



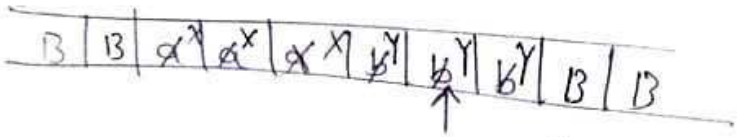
Step 22



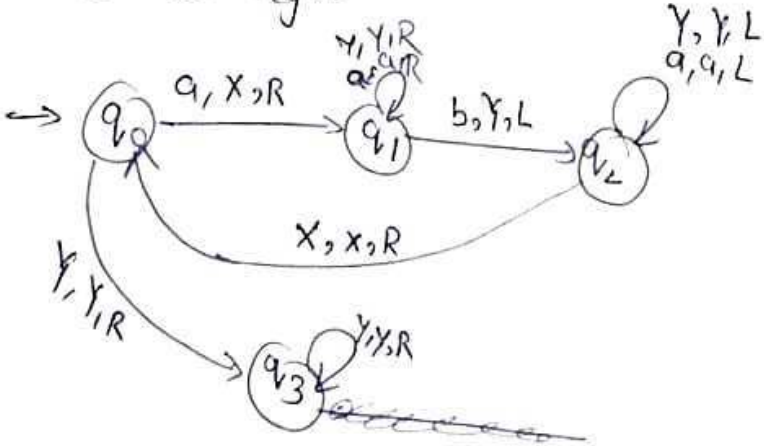
Now at q_0, on seeing Y, don't change Y, just move towards right by changing state to q_3.



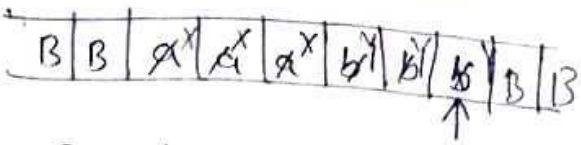
step 23



Now at q_3, on seeing Y, don't change Y, just move read/write head towards right.



step 24

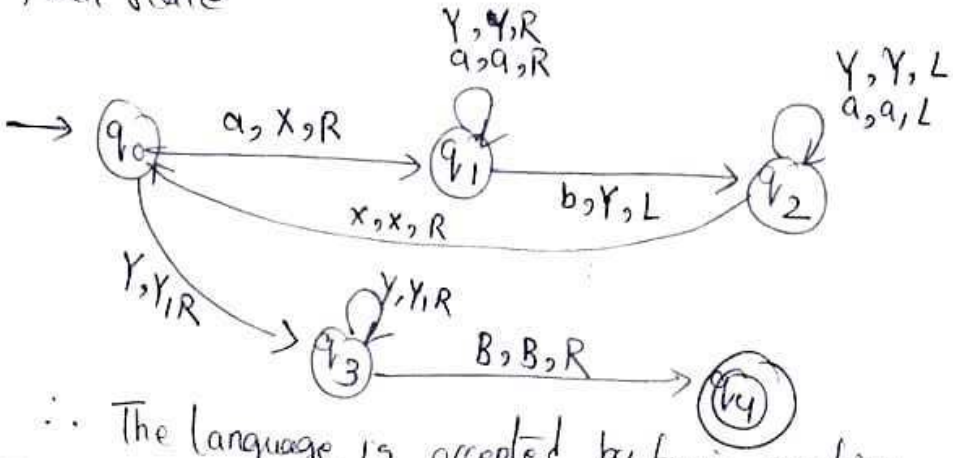


Repeat before step and move read/write head towards right.

step25

B	B	$\not{a}^X$	$\not{a}^X$	$\not{a}^X$	$\not{b}^Y$	$\not{b}^Y$	$\not{b}^Y$	B	B

Now at q_3, when we see blank symbol then it should be replaced by blank symbol only and leads to final state



∴ The language is accepted by turing machine.

Transition table is as follows, rows correspond to states and columns correspond to tape symbols.

states	a	b	X	Y	B
q_0	(q_1, X, R)	$\not\emptyset$	$\not\emptyset$	(q_3, Y, R)	$\not\emptyset$
q_1	(q_1, a, R)	(q_2, Y, L)	$\not\emptyset$	(q_1, Y, R)	$\not\emptyset$
q_2	(q_2, a, L)	$\not\emptyset$	(q_0, X, R)	(q_2, Y, L)	$\not\emptyset$
q_3	$\not\emptyset$	$\not\emptyset$	$\not\emptyset$	(q_3, Y, R)	(q_4, B, R)
q_4	$\not\emptyset$	$\not\emptyset$	$\not\emptyset$	$\not\emptyset$	$\not\emptyset$

Formal representation of turing machine is

$M = (\{q_0, q_1, q_2, q_3, q_4\}, \{a, b\}, \{a, b, X, Y, B\}, \delta, q_0, B, \{q_4\})$

Transition functions are

1. $\delta(q_0, a) = (q_1, X, R)$
2. $\delta(q_1, a) = (q_1, a, R)$
3. $\delta(q_1, b) = (q_2, Y, L)$
4. $\delta(q_2, a) = (q_2, a, L)$
5. $\delta(q_2, X) = (q_0, X, R)$
6. $\delta(q_1, Y) = (q_1, Y, R)$
7. $\delta(q_2, Y) = (q_2, Y, L)$
8. $\delta(q_0, Y) = (q_3, Y, R)$
9. $\delta(q_3, Y) = (q_3, Y, R)$
10. $\delta(q_3, B) = (q_4, H, -)$ ↓ halts.

Ex 2: Design turing m/c for $L = \{a^{2n} b^n \mid n \geq 1\}$

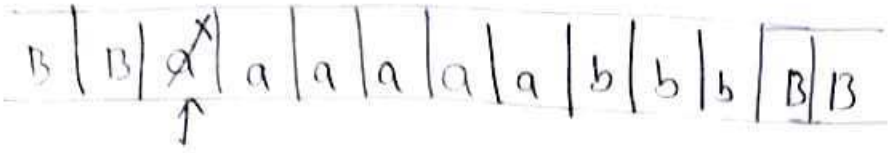
Soln: Language $L = \{aab, aaaabb, aaaaaabbb, \ldots\}$

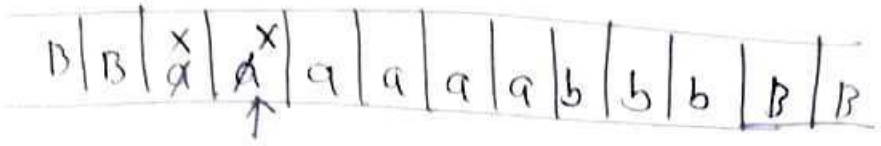
→ Here the logic is, if we read two conserative a's then make them as two x's and move right till we read b and replace b with Y ie., aaaabb (XX Y).

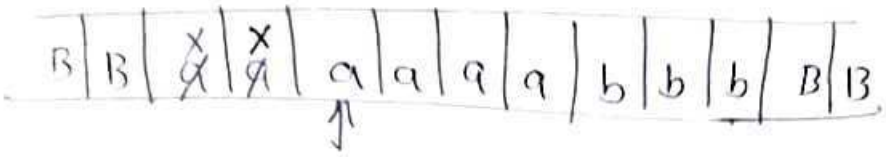
→ Then move head towards left, till we get X, then make two a's to two x's, then move towards right and replace b with Y. ie., aaaabb (XXXY Y).

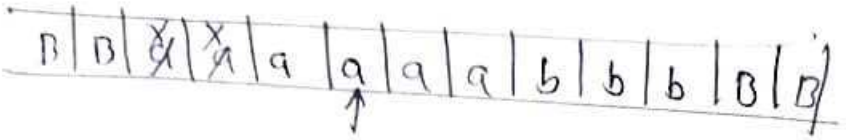
1. B
2. B B
3. B B
4. B B
5. B B
6. B B
7. B B
8. B B

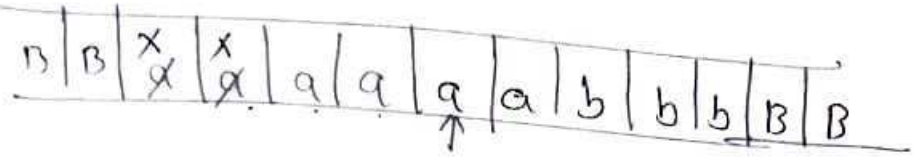
→ Then same
right and
→ Transition

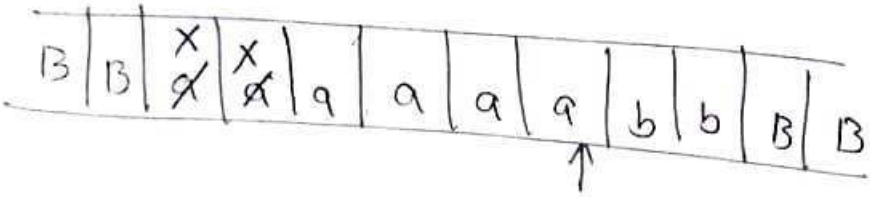
1. 

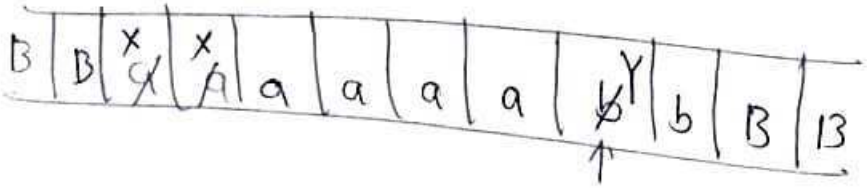
2. 

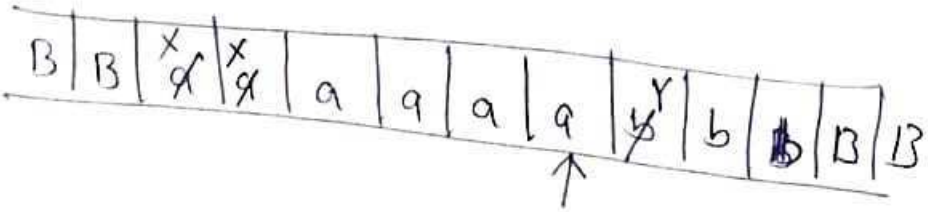
3. 

4. 

5. 

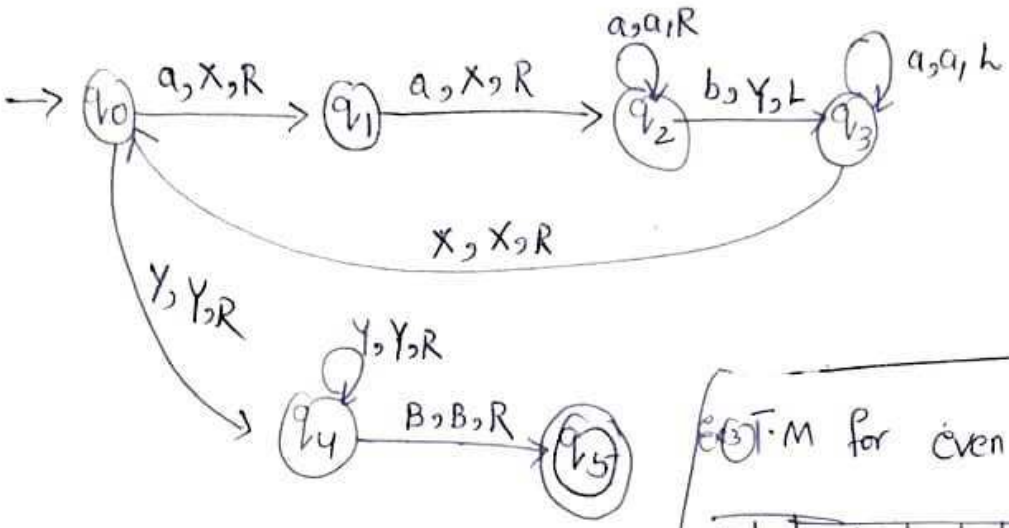
6. 

7. 

8. 

→ Then same process continues till we get X, then move right and repeat the same process.

→ Transition diagram is as follows.



Transition functions are

$\delta(q_0, a) = (q_1, X, R)$

$\delta(q_1, a) = (q_2, X, R)$

$\delta(q_2, a) = (q_2, a, R)$

$\delta(q_2, b) = (q_3, a, L)$

$\delta(q_3, a) = (q_3, a, L)$

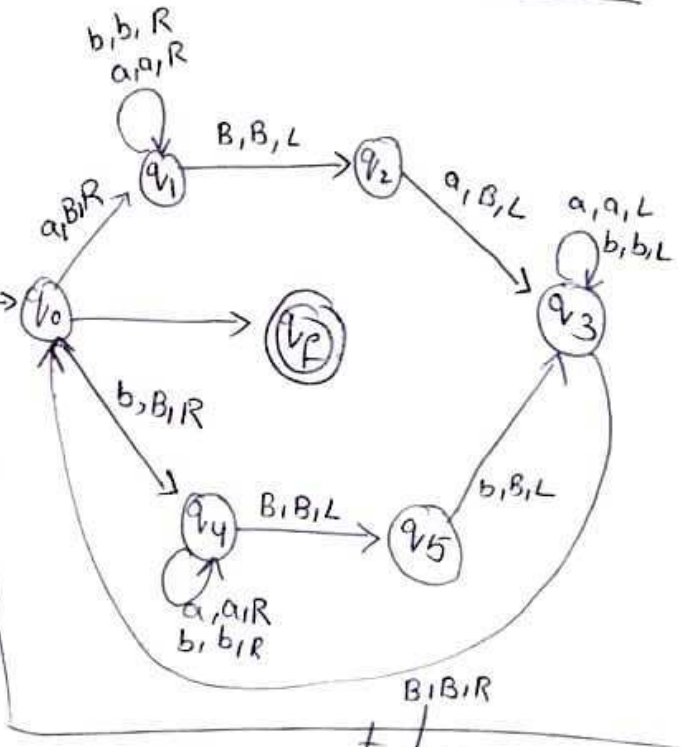
$\delta(q_3, X) = (q_0, X, R)$

$\delta(q_0, Y) = (q_4, Y, R)$

$\delta(q_4, Y) = (q_4, Y, R)$

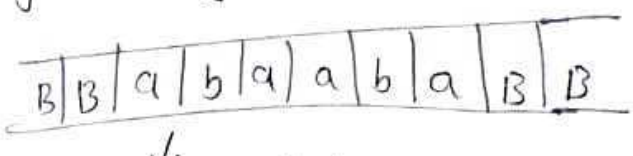
$\delta(q_4, B) = (q_5, B, -)$. String is accepted.

T.M for even palindrome

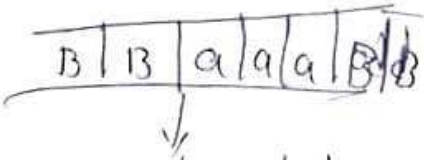


Ex3: Design turing machine for ~~odd~~ palindrome.

Sol^n:



even palind



odd palind

Ex: Construct a T.M which accepts the language over $\Sigma = \{a, b\}$ of subtraction in unary notation.

Soln:

$$f(m,n) = \begin{cases} m-n & \text{if } m > n \\ 0 & \text{if } m \leq n \end{cases}$$

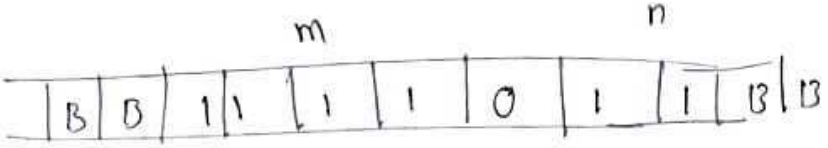
Let $m = 4$ $n = 2$

Represented by $(111) - (11) = 1$

m n

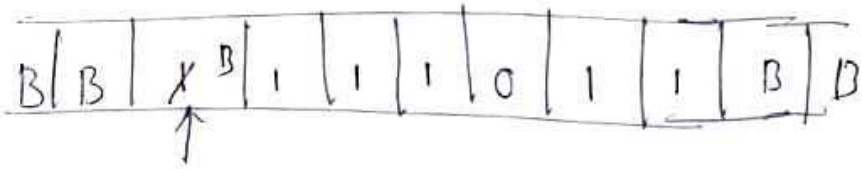
If $m = 2$ $n = 4$

$m - n = 0$



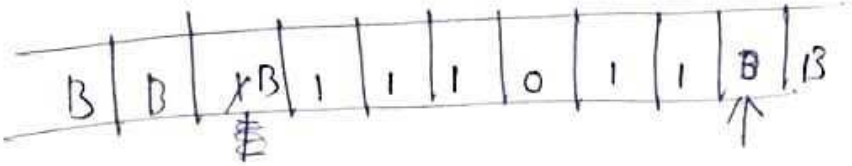
Here m & n are seperated by using seperator '0'

Step1:



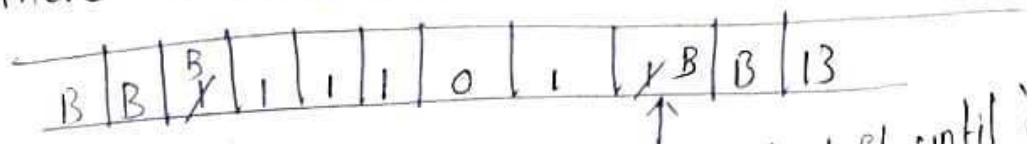
Replace 1 with B and move towards right without changing any symbol; till Blank symbol

Step2:



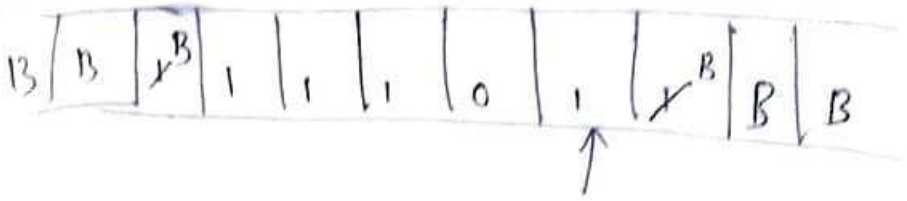
Now move towards left

Step3:

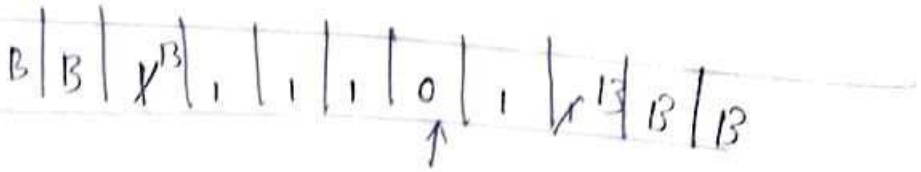


Now replace '1' with B and move towards left until 'B' is read.

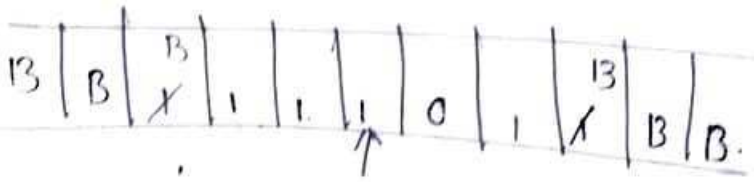
step4·



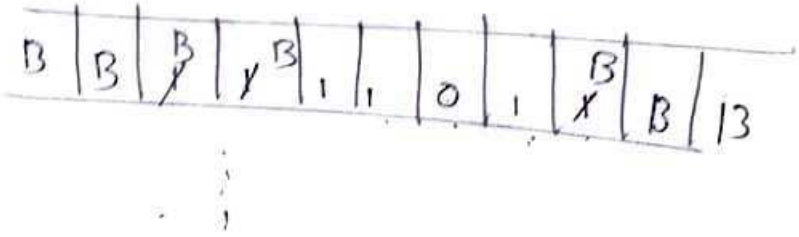
step5·



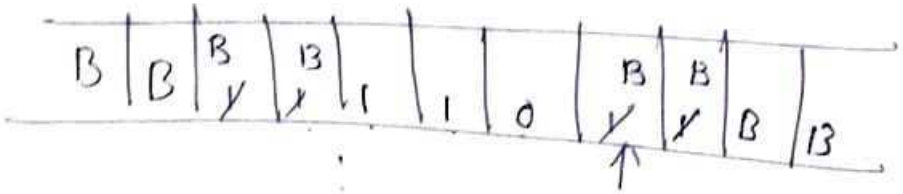
step6·



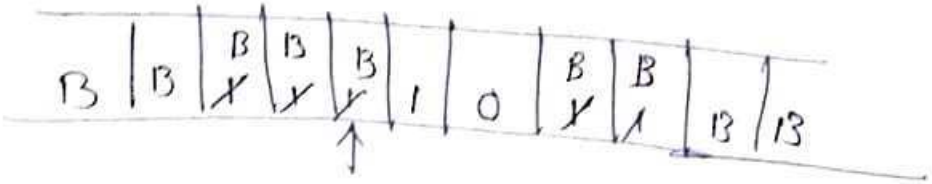
step10·



step16·



step21



Types of Turing machine

Various Types of TM's are

1. Turing machine with two dimensional tapes
2. Turing machine with multiple tapes
3. Turing machine with multiple heads
4. Turing machine with finite tape
5. Non deterministic turing machine.

It is observed that computationally all thes TM are equally powerful. That means one type can compute the same that other can. However the efficiency of computation may vary.

1) TM with two-dimensional tapes

This type of TM has two finite controls

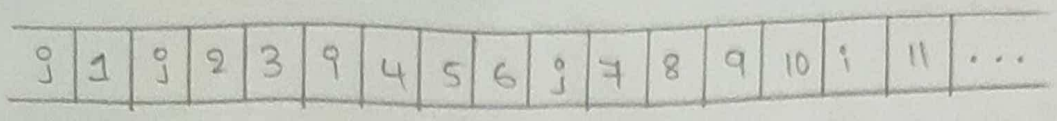
1) Read / Write head
2) Two dimensional tape

This tape has infinite extension to right and down. It is divided into small squares formed due to Corresponding rows and columns.

→ TM with one dimensional tape is equally powerful to that of two dimensional tape.

1	2	6	7	15	16	-
3	5	8	14	17	26	-
4	9	13	18	25	-	-
10	12	19	24	-	-	-
11	20	23	-	-	-	-
21	22	-	-	-	-	-
-	-	-	-	-	-	-

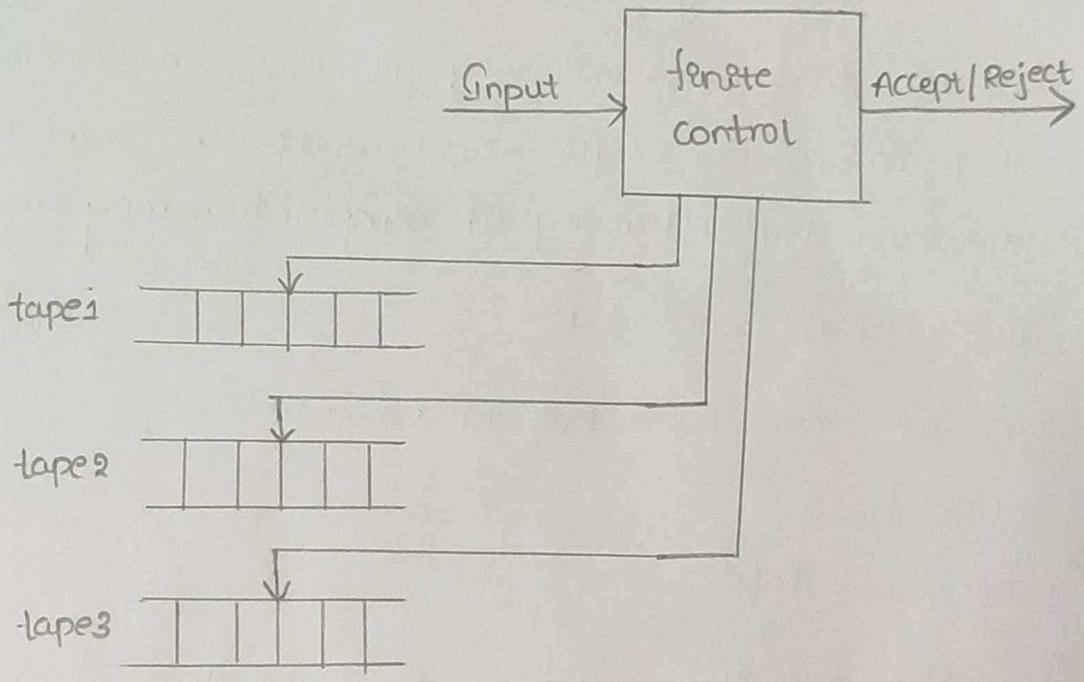
Two dimensional Tape



The head of Two dimensional tape moves one square up, down, left or right.

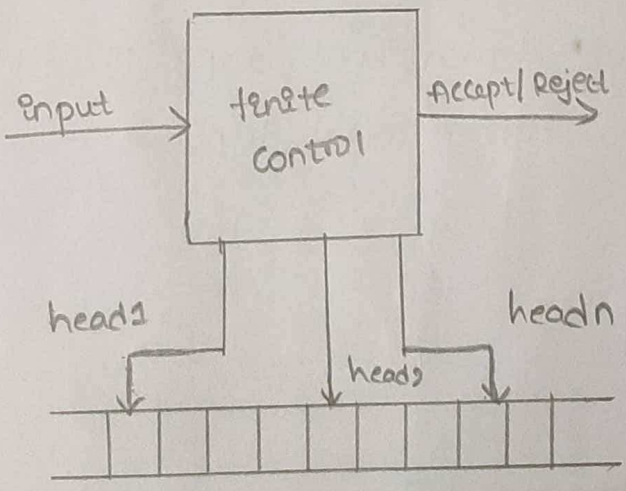
2) TM with multiple tapes

This is a kind of TM with one finite control and with more than one tape having its own read / write heads.



3) QM with multiple heads

The Tm with multiple heads can be shown below

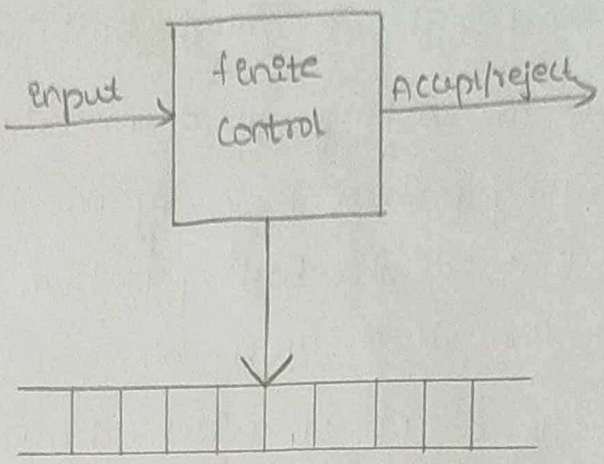


TM with multiple heads

There are n heads, but in any state, only one head can move. This type of TM are as powerful as one tape TM

4) Turing machine with infinite tape.

This is a TM that have one finite control and one tape which extends infinitely in both directions.



This type of TM's are as powerful as one tape TM's whose tape has a left end.

5) Non-determenistic turing machine

The concept of non-deterministic TM is semelar to the NFA.

→ for any state and any i/p symbol it can take any action from a set rather than a definite predetermined action.

for example, the language like $L = \{ww \mid w \in (a+b)^*\}$ can be shown by non determenistic TM.

→ The non determenistic TM is as powerful as determenistic TM.

HALTING PROBLEM

→ This is a famous undecidable problem of TM. To state halting problem we will consider the given configuration of a TM.

→ The o/p of TM can be

(i) Halt: The machine starting at this configuration will halt after a finite number of states.

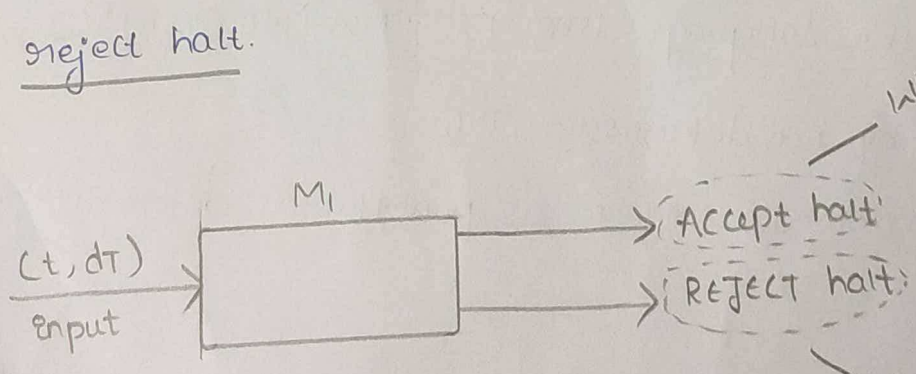
(ii) No Halt: - The machine starting at this configuration never

reaches a halt state, no matter how long it runs.

Now the question arises based on these two observations, given any functional matrix, i/p data tape and initial configuration, then is it possible to determine whether the process will ever halt? This is called halting problem.

That means we are asking for a procedure which enable us to solve the halting problem for every pair (machine, tape). The answer is "no". That is the halting problem is unsolvable. Now we will prove how it is unsolvable.

→ Let, there exists a TM M_1 which decides whether or not any computation by a TM T will ever halt when a description d_T of T and tape t of T is given. [That means input to machine M_1 will be (machine, tape)]. Then for every input (t, d_T) to M_1 if T halt for input t, M_1 also halts which is called accept halt. Similarly if T does not halt for i/p t then M_1 will halt which is called reject halt.

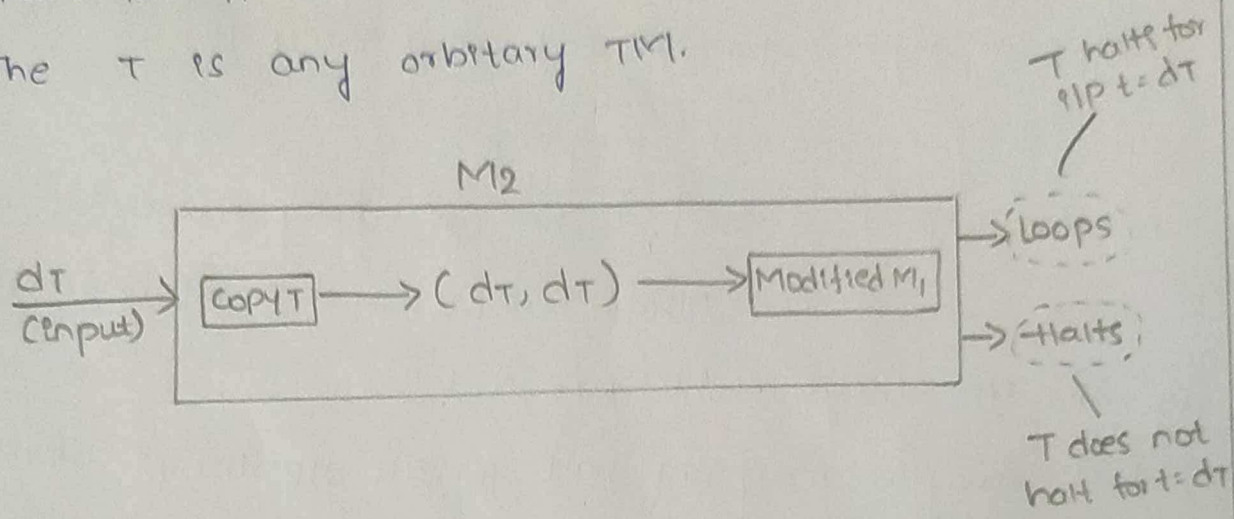


→ Now, we will consider another TM M_2 which takes an i/p d_T.

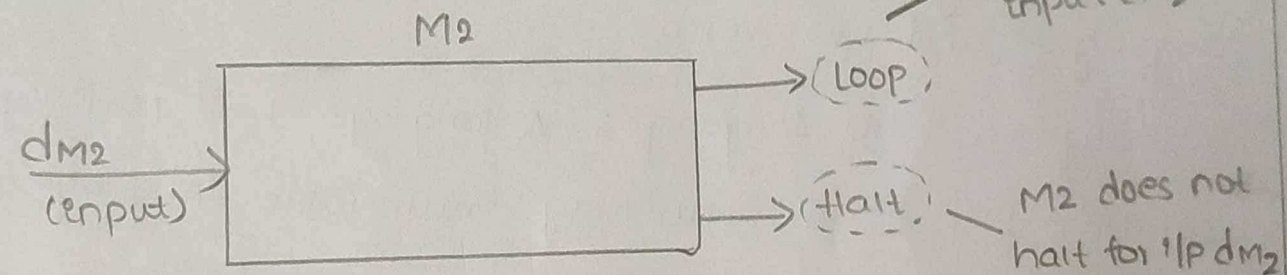
It first occupies d_T and duplicate d_T on its tape and then this duplicated tape information is given as i/p to Machine M_1.

But machine M_1 is a modified machine with the modification that whenever M_1 is supposed to reach an accept halt, M_2 loops forever. Hence behaviour of M_2 is as given. It loops if T halts for i/p $t = d_T$ and halts if T does not halt for $T = d_T$.

→ The T is any arbitary TM.



As M_2 itself is one TM we will take $M_2 = T$. That means we will replace T by M_2 from above given machine



Thus machine M_2 halts for i/p d_{M_2} if M_2 does not halt for i/p d_{M_2}. This is a contradiction. That means a machine M_1 which can tell whether any other TM will halt on particular i/p does not exist. Hence halting problem is unsolvable.

Recursive Language: A language 'L' is said to be recursive if there exists a TM which accept all the strings in 'L' and reject all the strings not in 'L'.

→ The TM will halt every time and give an answer (accept or reject) for each and every string i/p.

Recursively Enumerable language:

→ A language 'L' is said to be recursively Enumerable language if there exists a TM which accepts (and therefore halt) for all the i/p strings which are in 'L'

→ But may or may not halt for all i/p 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 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 TM for that language.

Undecidable problem about Turing Machine

1) Reduction
2) Empty and non-Empty language
3) Rice's Theorem.

Reduction: It is a technique in which if a problem P1 is reduced to a problem P2 Then any solution of P2 solves P1.

→ In general we have an algorithm to convert an instance of a problem P1 to instance of a problem P2 that have the same answer then it is called P1 reduces P2.

→ Hence if P1 is not Recursive, then P2 is also not recursive

→ if P1 is not recursively enumarable, then P2 is also not recursively enumerable.

Theorem: if P1 is reduced to P2 then,

i) if P1 is undecidable then P2 is also undecidable

ii) if P1 is non-RE then P2 is also non-RE

Proof: (i) consider an instance W of P1. Then construct an algorithm such that the algorithm takes instance W as i/p and convert it into another instance x of P2.

→ Then apply that algorithm to check whether x is in P2, if algorithm answers "yes" then that means x is in P2.

→ By we can also say that W is in P1.

→ Since we have obtained P2 after reduction of P1. Similarly if algorithem answers "no" then x is not in P2, that also means W is not in P1.

This proves that if P1 is undecidable then P2 is also undecidable

ii) We assume that P1 is non-RE but P2 is RE. Now construct an algorithm to reduce P1 to P2, but by this algorithm P2 will be recognized. That means there will be a TM says that "yes" if the i/p is P2 but may or may not halt for the i/p which is not in P2.

→ Apply a TM to check whether x is in P2. If x is accepted that also means w is accepted.

→ This procedure describes a TM whose language is P1 if w is in P1 then x is also in P2 and if w is not in P1 then x is also not in P2.

This proves that if P1 is non-RE then P2 is also non-RE.

(iii) Empty and non empty language:

There are two types of languages empty and non-empty Let Le denotes an empty language and Lne denotes non-empty language.

Let w be a binary string and M_i be a TM.

If $L(M_j) \neq \phi$ then M_i does not accept any i/p then w is in Le.

Similarly if $L(M_i)$ is not the Empty language then w is in Lne. Thus we can say that,

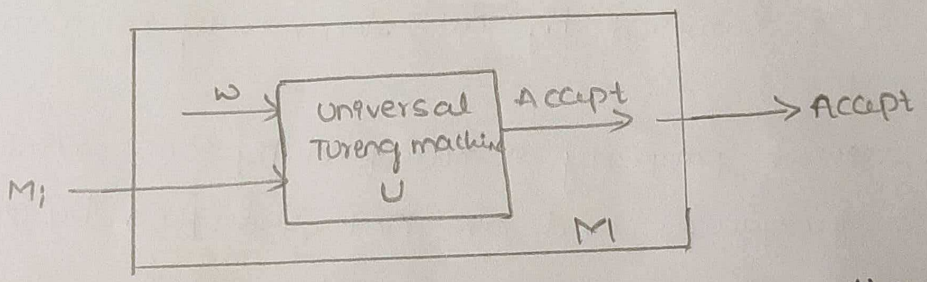
$Le = \{M \mid L(M) = \phi\}$ $Lne = \{M \mid L(M) \neq \phi\}$

Both Le and Lne are compliment of one another

Theorem : Lne is recursively enumerable.

proof :- To prove this, we simply need to prove that there exists some TM which accepts Lne. So do this we need to construct non deterministic TM M that can be converted to deterministic TM.

The TM M accepts another TM M_i as i/p. With the non-deterministic capability M can guess the i/p w that can be accepted by M_i. If M_i accepts w then M also accepts the input M_i



Thus if at all M_i accepts even one i/p, the M will guess that streng and will accept M_i. But if $L(M_i) = \phi$ that means M can make no guess about the i/p streng and therefore can not accept M_i.

This proves that there should be such a TM whose language $L(M) \neq \phi$, thus $L(M) = Lne$.

Rice's Theorem :-

Theorem: Every non trivial property of Rt language is undecidable

proof: Rice theorem states that any non-trival Semantic property of a language which is recognized by a TM

is undecidable. A property, P is the language of all TM that satisfy that property.

formal definition: If P is a non-trivial property, and the language holding the language of all TMs that satisfy the property. property L_P, is recognized by TM M, then $L_P = \{\langle M \rangle | L(M) \in P\}$ is undecidable.

→ property of language P is simply a set of languages. If any language belongs to P ($L \in P$), it is said that L satisfies the property P.

→ A property is called to be trivial if either it is not satisfied by all recursively Enumerable languages.

→ (or) if it is satisfied by all recursively Enumerable language.

→ A non-trivial property is satisfied by some recursively Enumerable languages and are not satisfied by others, formally speaking, in a non-trivial property, where $L \in P$, both the following properties hold:

property 1 - There exists TMs M_1 and M_2 that recognize the same language, i.e., either ($\langle M_1 \rangle, \langle M_2 \in L$)

or ($\langle M_1 \rangle, \langle M_2 \rangle \notin L$)

property 2 - There exists Turing machines M_1 and M_2 where M_1 recognizes the language while M_2 does not, i.e, $\langle M_1 \rangle \in L$ and $\langle M_2 \rangle \notin L$.

Proof : Suppose a property P is non-trivial and $\varphi \in P$.

Since, P is non trivial, at least one language satisfies P,

i.e., $L(M_0) \in P$, $\exists$ Turing machine M_0.

Let, w be the input in a particular instant and N is a Turing machine which follows -

on input x

→ Run M on w

→ if M does not accept (or doesn't halt), then do not accept x (or do not halt)

→ if M accepts w then run M_0 on x. if M_0 accepts x, then accept x.

A function that maps an instance ~~ATM = <w, m>~~

$A_{TM} = \{<M, w> \mid M \text{ accepts input } w\}$ to a N

Such that

→ if M accepts w and N accepts the same language as M_0, then $L(M) = L(M_0) \in P$

→ if M does not accept w and N accepts φ, Then $L(N) = \varphi \notin P$

Since A_{TM} is undecidable and it can be reduced to L_P, L_P is also undecidable.

Decidable properties of Formal language

Decidable properties	RL	CFL	CSL	RCL	REL
1 Membership $w \in L(m)$	✓	✓	✓	✓	×
2. Emptyness $L = \phi$?	✓	✓	×	×	×
3. finiteness $\|L\| \leq n$	✓	✓	×	×	×
4. Completness $L = \Sigma^*$	✓	✓	×	×	×
5. Equalness $L_1 \stackrel{?}{=} L_2$	✓	×	×	×	×
6. Complement $L^c = \Sigma^* - L$	✓	×	×	×	×

✓ — decidable

× — undecidable.

post's correspondence problem (pcp)

The undecidability of strings is determined with the help of post's correspondence problem (pcp).

→ "The post's correspondence problem consists of two lists of strings that are equal length over the input Σ.

The two lists are $A = w_1, w_2, w_3, \ldots\ldots w_n$

$B = x_1, x_2, x_3, \ldots\ldots x_n$

then there exists a non-empty set of strings $i_1, i_2, i_3, \ldots\ldots i_n$ such that

$w_{i_1}, w_{i_2}, w_{i_3} \ldots\ldots w_{i_n} = x_1, x_2, x_3 \ldots\ldots x_n$".

To solve the pcp we try all the combination of $i_1, i_2, i_3 \ldots i_n$ to find the $w_{i_i} = x_i$ then we say that pcp has a solution.

Ex: consider the correspondence system as given below

$A = (1, 0, 010, 11)$ and $B = (10, 10, 01, 1)$. The i/p set is $\Sigma = \{0, 1\}$. find the solution.

Sol: A solution is 1 2 1 3 3 4. that means

$w_1, w_2 w_1, w_3 w_3 w_4 = x_1 x_2 x_1 x_3 x_3 x_4$

$10101001011 = 10101001011$

$|A| = 4 \cong |B| = 4$

Ex(2): Obtain the solution for the following system of post's correspondence problem

$A = \{100, 0, 1\}$ $B = \{1, 100, 00\}$

The input symbols of A and B are same

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

$|A| = 3 \cong |B| = 3$

The solution is 1 3 1 1 3 2 2.

w_1	w_3	w_1	w_1	w_3	w_2	w_2
100	1	100	100	1	0	0

x_1	x_3	x_1	x_1	x_3	x_2	x_2
1	00	1	1	00	100	100

$w_1 w_3 w_1 w_1 w_3 w_2 w_2 = x_1 x_3 x_1 x_1 x_3 x_2 x_2$

$100\,1\,100\,100\,1\,00 = 1001100100100$

ex③: Obtain the solution for the following system of post's correspondence problem $A = \{ba, abb, bab\}$

$B = \{bab, bb, abb\}$

Now to consider 1, 3, 2 the streng bababbb from set A and bababbb from set B thus the two strengs obtained are not equal. As we can try various combinations from both the sets to find the unique sequence but we could not get such a sequence. Hence there is no soln for this system.

ex④: Obtain soln for the following correspondence system

$A = \{ba, ab, a, baa, b\}$, $B = \{bab, baa, ba, a, aba\}$

The i/p set is $\{a, b\}$

$w_1 w_5 w_2 w_3 w_4 w_4 w_3 w_4 = x_1 x_5 x_2 x_3 x_4 x_4 x_3 x_4$

The soln give a unique string

bababaabaaaabaa.

ex⑤: Does pcp with two lists $x = (b, bb^3, ba)$ and $y = (b^3, ba, a)$ have a soln?

Now we have to find out such a sequence that strings formed by and y are identical. Such a sequence is 2, 1, 1, 3. Hence from x and y list

$w_2 w_1 w_1 w_3 = x_2 x_4 x_4 x_3$

$bab^3 b\ b\ ba \qquad ba\ b^3 b^3 a$

ex⑥: find whether the pcp $P = \{(10, 101), (011, 11), (101, 011)\}$ has a match. Give the soln.

Let $P = \{(10, 101), (011, 11), (101, 011)\}$

$w_1 = \{10, 011, 101\}$

$w_2 = \{101, 11, 011\}$

The pcp has a solution if $w_{1i} = w_{2j}$,

$w_{12} = w_{23}$, $w_{13} = w_{21}$

but $w_{11} \neq w_{22}$

Hence we can not find any string $w_{1i} = w_{2j}$.

Hence thes pcp has no solution.

Modified post correspondence problem (MPCP):

The Modified post correspondence problem (MPCP) is just like PCP except that we specify both the set of tiles and also a special tile. Matches for MPCP have to start with the special tile.

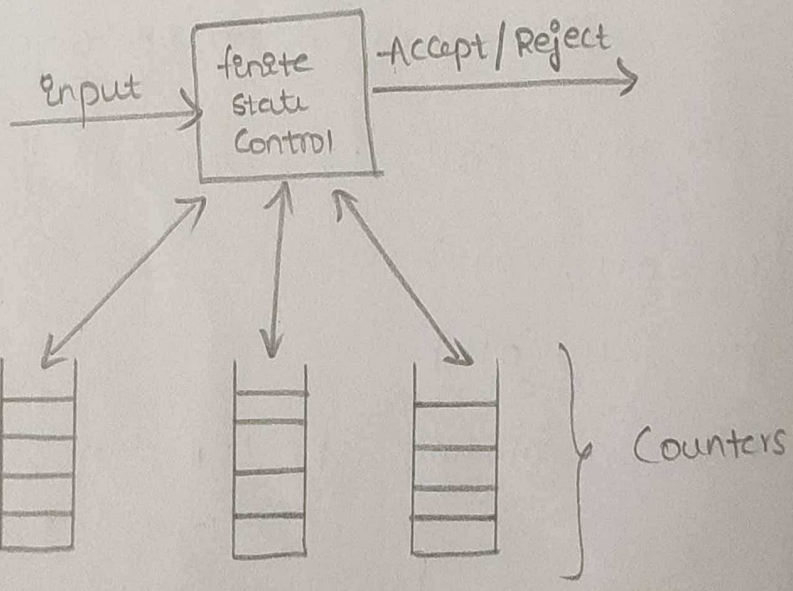
(or)

In MPCP the solution must starts from 1.

Ex:- 123, 113 .. etc.

Counter Machine

A Counter machine can be represented by following model-



Counter machine

There are two ways of representing Counter machine.

1) The Counter machine is similar to a multistack TM, But the only difference blw them is that in place of each stack there is a Counter. The Counter obtain non negative integers. Each move of Counter machine

depends on its state, input symbol. In one move Counter machine can:

i) Change state

ii) Add or subtract 1 from any of its counters. The negative counters are not allowed at all.

2) The Counter machine is similar to restricted multi stack machine. These restrictions are -

i) There are only two stack symbols: Z_0 and X

ii) The Z_0 is the bottom of stack marker. It is initially on each stack.

iii) Replace Z_0 only by string of the form $X^i Z_0$ where $i \geq 0$.

iv) Replace X only by X^i for $i \geq 0$. i.e. The Z_0 appears on the bottom of each ~~stack~~ and all other stack symbols are X.

There are two important observations about the Counter machine.

1) Every language accepted by a Counter machine is recursively enumerable.

2) Every language accepted by one Counter machine is a Context free language

Non-Recursive enumerable language

Universal Turing machine:

→ A universal Turing machine, Mu is an automata that, given as input the discription of any TM M and a string w can simulate the computation of M for i/p w.

→ To construct such Mu, we consider a turing machine without loss of generality assume that

$$M = (Q, \{0,1\}, \{0,1,B\}, \delta, q_1, B, q_2)$$

where $Q = \{q_1, q_2, \ldots q_n\}$

q_1 = initial state

q_2 = final state

$\{0,1,B\} \in \Gamma$ are represented as $\{a_1, a_2, a_3\}$

→ Directions left or right are represented as D_1 and D_2 respectively.

→ The transitions of TM are encoded in special binary representation where each symbol is represented by 1

for ex: if there is a transition

$$\delta(q_i, a_j) = (q_k, a_l, D_m)$$

then the binary representation for the transition is $0^i 1 0^j 1 0^k 1 0^l 1 0^m$

transitions $t_1, t_2 \ldots t_n$

$$111\ t_1\ 11\ t_2\ 11\ t_3\ 11 \ldots 11\ t_n\ 111$$

Note: transitions need not be in any particular order.

→ If a string has to be verified then the problem is represented as a tuple $\langle M, w \rangle$ where M is defenation of TM and w is i/p string.

Conside an example:

Let $M = [\{q_1, q_2, q_3\}, \{0,1\}, \{0,1,B\}, \delta, q_1, B, q_2\}$

have moves defined as

$$\delta(q_1, 1) = (q_3, 0, R)$$
$$\delta(q_3, 0) = (q_1, 1, R)$$
$$\delta(q_3, 1) = (q_2, 0, R)$$
$$\delta(q_3, B) = \{q_3, 1, L)$$

Give the problem representation for the string w=1011

Sol: let the binary representation for

State : $\{q_1, q_2, q_3\}$ be $\{0, 00, 000\}$

alphabet : $\{0, 1, B\}$ be $\{0, 00, 000\}$

directions : $\{L, R\}$ be $\{0, 00\}$

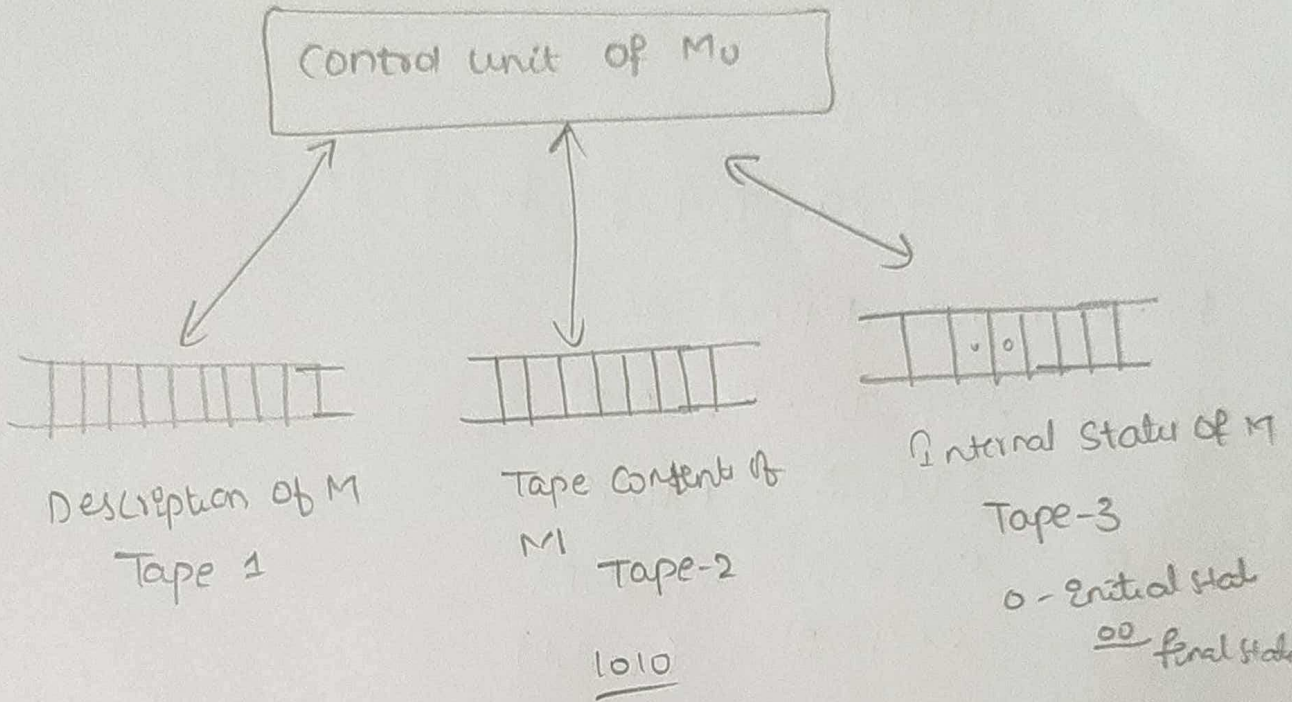
Transistions are represented as follows.

	Binary representation
$\delta(q_1, 1) = (q_3, 0, R)$	0100100010100
$\delta(q_3, 0) = (q_1, 1, R)$	0001010100100
$\delta(q_3, 1) = (q_2, 0, R)$	0001001001010 0
$\delta(q_3, B) = (q_3, 1, L)$	0001000100010010

∴ The problem representation $\langle M, 1011 \rangle$ is

(111) 0100100 0101001 1 0001010 1001001 1 000100 1001010011 000100100010 (111) 1011

The following diagram shows organizations of UTM which has a control unit and three tapes



non Recursively Enumerable language:-

If a language is not represented by Turing machine with / without halting then it is called non-REL

Ex:- Diagonalization

→ The language that is not accepted by any TM this proves the diagonolization. is not Recursively Enumerable

Σ^* is countable 2^{Σ^+} is uncountable.

	∈	a	b	aa	ab	ba	bb	aaa	aab . . . .
1)	1	1	0	0	0	0	0	0	0
2)	0	1	1	1	0	0	0	0	0
3)	1	1	0	1	0	0	0	0	0

110

001 → This language is should be Countable as we assumed but it will not present in any of machine so it is Contraction, so it can be said to uncountable

$L_1 = \{\epsilon, a\}$

$L_2 = \{a, b\}, L_3 = \{\epsilon, a, aa\}$

undecidable of problems

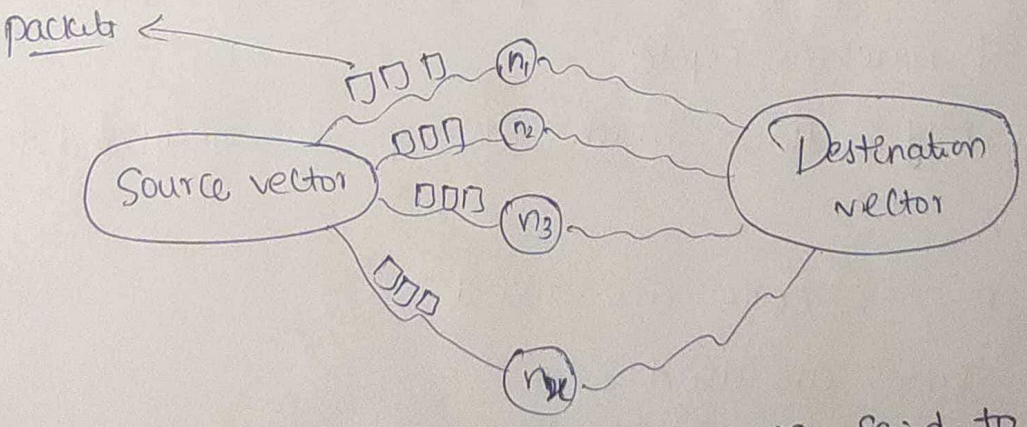
A problem p is said to be decidable if there exists a tureng machine to represent the problem or if there exists an algorethm to find the solution to the problem.

1) ambiguity in CFG
2) post correspondence problem (PCP)
3) Vertex vesting in a path
4) Code optimezation

Code optimization :- It is an undecidable problem we cannot say that a particular piece of code optimized code (in terms of cost space time)

Vertex vesting problem :-

In a N/w of computer's a packet travelling from source to Destination may or may not visit a particular node in a path



ambiguity in CFG :- A Grammer G is said to be ambigeous Grammer if it generates more than one different parse for any string.

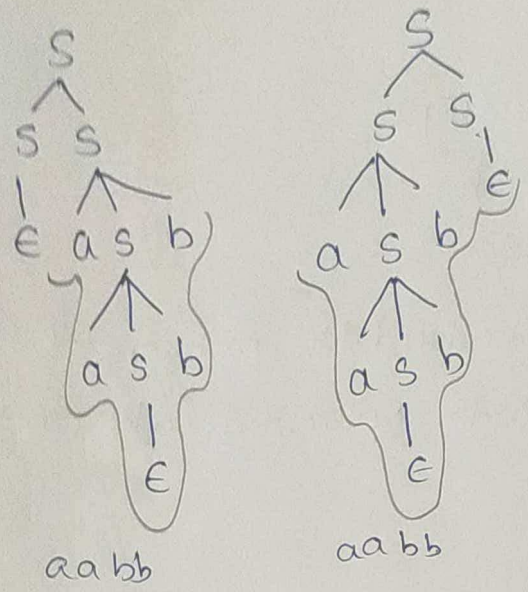
→ It is a trial and error method.
→ There exists no particular algorithm or tureng machine to prove the ambiguity in a grammer (undecidable problem)

Ex: $S \rightarrow asb \mid ss$

$S \rightarrow \epsilon$

for string "aabb" the above grammer generates two

Parse tree



Complexity classes

→ Time complexity :- How long computation takes to execute. In TM this could be measured as no. of moves which are required to perform computation.

→ Number of machine cycle.

Space complexity :- how much storage is required for computation.

→ In TM, no. of cells are used.

→ no. of bytes are used.

Types of Complexity classes

1) P-class: set of decision problem is solvable in polynomial time or in the class P.

If there exists an algorithm A such that

→ A. takes instances of D as i/p

→ A always o/p's the correct answer "yes" or "No".

→ There exists a polynomial p such that the execution of A on I/P of size n always terminate on p(n) steps

Eg:- The minimum spanning tree problem is in class p
Kruskal's algorithm

The class p is often considered as synonymous with the class of computationally feasible problem although in practice this is somewhat unrealistic.

NP class :-

A decision problem is non deterministically polynomial time solvable or in the class NP if there exists an Algorithm A such that -

→ There exists a polynomial p such that for each potential witness of each instance of size n of D. there exists an algorithm A takes at most p(n) steps

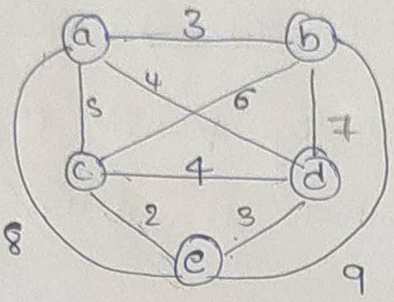
→ Think of a non-deterministic computes as a computer that magically guesses a solution, then has to verify that it is correct.

- If soln exist, computer always guesses it
- One way to imagine it: a parallel computer that can freely spawn an infinite no. of processes
  → One process work on each possible solution
  → All processes attempt to verify that their solution works
  → If a process finds it has a working solution o

So NP - problem verifable in polynomial time.

→ Every problem in this class can be solvable in exponential time using exhaustive search.

Eg:- Travelling sales man problem.

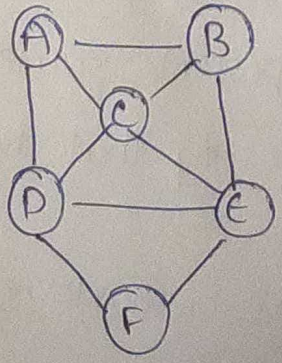


Routes	distance
a b c d e a	24
a b c e d a	19
a b d c e a	24
a b d e c a	16

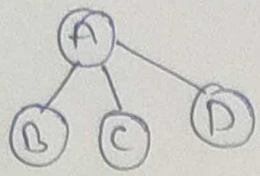
NP complete : NP complete are the subset of the class NP.

It is the set of all decision problem whose solution can be solved & verified in polynomial time on a non deterministic TM is called NP complete.

Ex:- hamelton cycle problem (Starting from starting point and writting adjacent vertices as brandes.

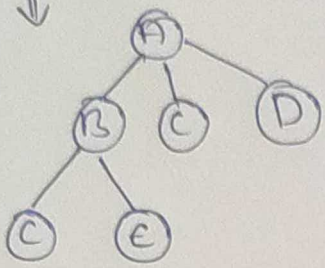


A ⇒

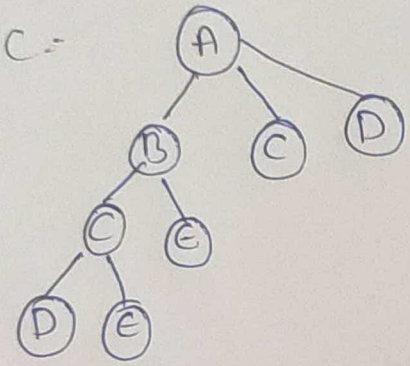


(∵ for A adjacent vertices)

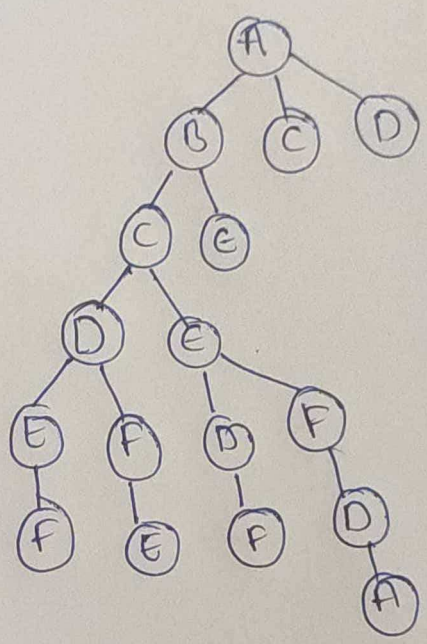
B ⇒



C:-

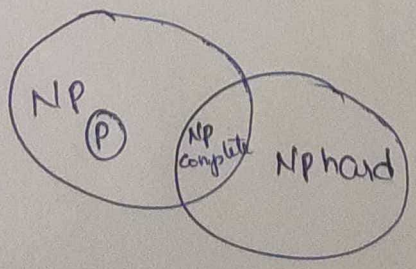


Similiary for D & E



← HC

NP hard:- The concept of NP hardness plays an vital role on the discussion about the relation between P & NP



Eg:- Travelling sales man problem