2076/1/16

UNIT: 1

BASIC FOUNDATIONS

AUTOMATA, COMPUTABILITY AND COMPLEXITY

Theory of computation

Complexity Theory:

Complexity theory is the central topic in theoretical computer science. It has direct application to computability theory and uses computation models such as turing machines to help test complexity. Complexity theory helps computer scientists relate and group problems together into complexity classes. Sometimes if one problem can be solved it opens a way to solve other problems in its complexity class. Complexity helps determine the difficulty of a problem, often measured by how much time an space it takes to solve a particular problem. For example: Some problems can be solved in polynomial amounts of time others take exponential amount of time with respect to the input size.

Complexity theory has real world implications too, particularly with algorithm design and analysis. An algorithm can be analysed in terms of its complexity, this is often described in big oh (O) notation. Often times, programmer want to write efficient algorithms and being able to tell if an algorithm can be analysed and runs in polynomial time versus exponential time and can tell programmer if his/her algorithm is the best choice or not. Both in theory and practice, complexity theory helps computer scientists determine the limits of computers what they can do and cannot do.

Thus, complexity theory is branch of computer scienc and theory of computations that deals with the majors of efficiency of an algorithm such as in terms of time and space complexity.

Complexity and Algorithm.

Complexity is used to describe an algorithm. Complexity is also used to describe resource used in algorithms. In general the resources of concern are time and space. The time complexity of an algorithm represent the number of steps it has to take to complete. The space complexity of an algorithm represents the amount of memory the algorithm needs inorder to work.

The time complexity of an algorithm describes how many steps an algorithm needs to take with respect to the input. For example: If each of the 'n' input elements is only update operated once, this algorithm would take $O(n)$.
For example: If each of the 'n' inputed elements is only operated once or for some constant steps no matter the size of the input, this is a constant time i.e $O(1)$.

If an algorithm does 'n' operations for each one of 'n' elements inputed to the algorithm, then this algorithm runs in $O(n^2)$.

In algorithm design and analysis, there are three types of complexity that computer scientist thinks about. They are: Best case, worst case and average case

Let's say you are sorting a list of numbers. If the input list is already sorted, your algorithm probably have very little work to do, this could be considered as best case input and would have very fast running time.

Another case is, let's take the same sorting algorithm and give it an input list that is entirely backward and every element is out of list place. This case could be considered as worst case input and would have very slow running time. Now say you have a random input that is somewhat ordered and somewhat disordered. i.e an average input. This would be the average case running time. Therefore best case complexity gives lower bound on the running time of the algorithm for any instance of input. This indicates that the algorithm can never have lower running time then best case for particul class or problem.

Worst case complexity gives upper bound on th running time of the algorithm for any instance of inpu This ensures that no input can overcome the runnin time limit posed by worst case complexity.

Average case complexity gives average numbe of steps required on any instances of the input.

The time complexity and space complexity i also called the efficiency of the algorithm.

Computability Theory:

Computability Theory is the branch of the Theory of Computations that studies which problems are computationally solveable using different models of computation. A central question of computer science is to address the limits of computing devices by understanding the problems we can use computers to solve.

Computability theory is also known as recursion theory. which is the branch of mathematical logic of computer science and theory of Computation. that originated in the 1930s with the study of computable functions and turing machine. It includes the study of generalised computability and defineability. Basic questions addressed by recursion theory includes:

1) What does it mean for a function on natural numbers to be computable?
2) How can non computable functions be classified into a hierarchy based on their ~~label~~ level of compubality

Automata Theory:

Automata Theory is the study of abstract machines and automata as well as the computational problems that can be solved using them. It is theory in theoretical Computer Science and Discrete Mathematics.

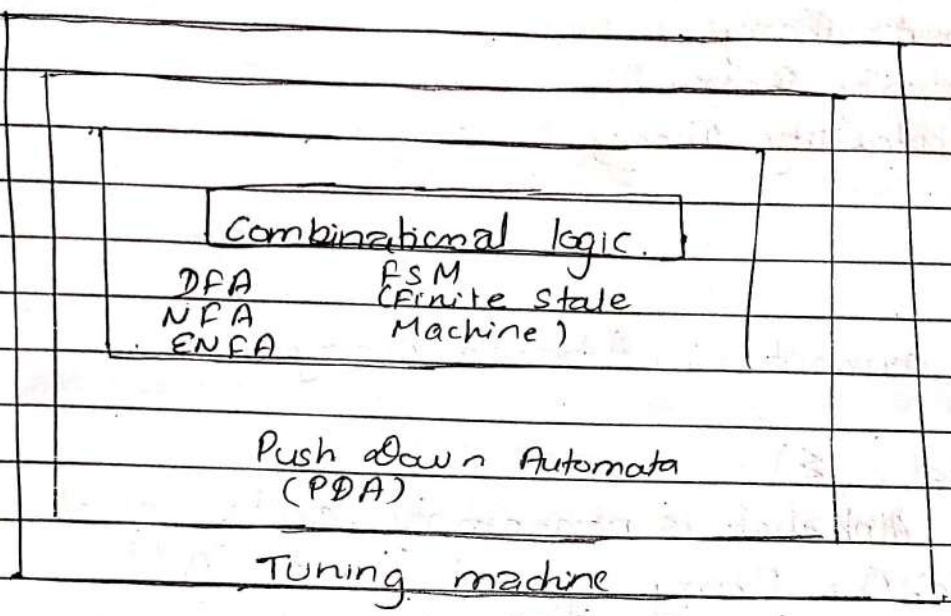


fig:- Classes of Automata

An automaton consists of finite numbers of states is called finite automaton or finite state machine. Automations are abstract modles of machines that perform computations on an input by moving through a series of states or configuration. At each state of the computation, a transition function determines the next configuration on the basis of a finite portion of the present configuration. As a result once the computation receives an accepting configuration it accepts the input. The most general and powerful automata is the turing machine.

Therefore in theoretical computer Science and Mathematics, the theory of computation is the branch that deals with how problems can be efficiently solve on a model of computation using an algorithm. The major field therefore in the theory of computation are

1) Automata Theory
2) Complexity Theory
3) Computability Theory

Basic concept of Automata Theory:

1) Alphabet: (Σ)

Alphabet is non-empty finite set of symbols.
Example: The Binary symbol. i.e $\Sigma = \{0, 1\}$
Example: Set of all small letters i.e $\Sigma = \{a, b, c, \ldots z\}$
Alphabets are denoted by capital Sigma (Σ).

2) Strings:

Strings is a finite sequence of symbols taken from some alphabets. Example: 0101001 is a string taken from $\Sigma = \{0, 1\}$.

3) Empty String:

Empty string is the string with zero occurence of symbols and denoted by ϵ.

4) length of string:

Length of string of the given string is the total no. of occurence of symbol. Example: 01010 has length 5. i.e $|01010| = 5$.

Length of string 'w' is denoted by $|w|$.
Example: |element| = 7 , $|\epsilon| = 0$

5) Power of alphabet (Σ^k):

Σ^k over alphabet sigma is the set of all available strings of length k. Let $\Sigma = \{0, 1\}$ be the alphabet then

$\Sigma^0 = \{\{\epsilon\}\}$, $\Sigma^1 = \{0, 1\}$, $\Sigma^2 = \{00, 01, 10, 11\}$

$\Sigma^3 = \{000, 001, 010, 100 \ldots 111\}$

$\Sigma^4 = \{0000, 0001, 0010, \ldots 1111\}$

6) Kleen closer (Σ^*):

Kleen closer is defined as $\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \ldots$

= set of all possible strings taken from Σ.

7) Language:

Language 'L' of an alphabet Σ is the set of string choosen from Σ^* i.e $L \subseteq \Sigma^*$.

eg: L = set of all binary strings with n 0's followed by n nos of 1's. for all $n \geq 0$.

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

$L = \{\epsilon, 01, 0011, 000111, 00001111, \ldots\}$

eg: L = set of all binary with equal number of 0's & 1's

$L = \{\epsilon, 01, 10, 1100, 0011, 0101, 1010, \ldots\}$

8) Positive closure : Σ^+

The set of all the strings over an alphabet Σ except the ~~entr~~ empty string is called positive closure and is denoted by Σ^+. i.e $\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \ldots\ldots$

9) Concatenation of strings:

Let x and y be strings. Then xy denotes the concatenation of x and y. More precisely, if x is the string composed of 'i' symbols, $x = a_1, a_2 \ldots\ldots a_i$ and y is the string composed of 'j' symbols, $y = b_1, b_2, \ldots b_j$, then xy is the string of length i+j and $xy = a_1 a_2 \ldots a_i\ b_1 b_2 \ldots b_j$

Note $xy \neq yx$.

10) Suffix of a string.

A string 's' is called a suffix of a string 'w' if it is obtained by zero or more leading symbols in 'w'. for example, w = abcd, s = bcd then 's' is suffix of 'w'. and s is proper suffix if $s \neq w$.

11) Prefix of a string.

A string 's' is called a prefix of a string 'w'. if it is obtained by removing zero or more trailing symbols of 'w'. for example: w = ~~a~~abcd, s = abc then s is prefix of w. S is proper prefix if $s \neq w$.

12) Substring of a string

A string 's' is called substring of a string 'w' if it is obtained by removing zero or more leading or trailing symbols in 'w'. It is proper substring if $s \neq w$.

UNIT- 2

INTRODUCTION TO FINITE AUTOMATA

INTRODUCTION TO FINITE AUTOMATA

Finite state machine:

A finite automaton is a Mathematical (model) abstract machine that has a set of states whose controls move from state to state in response to external ~~stat~~ input.

The control may be either deterministic meaning that the automation cannot be in more than one state at any one time, or non-deterministic meaning that it may be in several states at once.

This distinguishes the class of automata as deterministic finite automata (DFA) or non-deterministic finite automata (N-FA)

The DFA cannot be in more than one state at any time. The NFA can be in more than one state at a time.

The finite automaton may be generating output or it may not be.

The finite state machines are used in computer science and data networking as for example: Finite state machine are basis for programs for spell checking, grammar checking, speech recognition, transferring text.

The finite state machines can be represented with state transition diagram or state transition table.

A state transition table showing what state finite state machine will move to based on the current state

and other inputs. The row in the table indicates input symbol and the cells in the table indicate the next state if an event happen.

The state transition diagram in the directed graph with label edges. In this diagram each state is represented by circle, arrows label with the input and/or output pair are shown for each transition in between the state represented by the circle.

example:

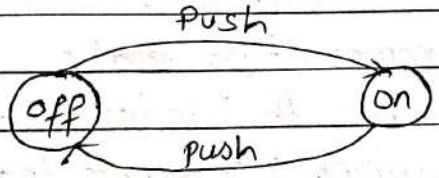


fig:- State transition diagram showing finite automata modeling of an on/off switch.

	Push
off	ON
ON	off

fig:- State transition table of above FSM.

DETERMINISTIC FINITE AUTOMATA (DFA)

A DFA is defined by a quitaple (5-taple) as $(Q, \Sigma, \delta, q_0, F)$

where:- Q = Finite set of states

Σ = finite set of symbols.

δ = A transition function that maps $Q \times \Sigma \to Q$.

q_0 = start state; $q_0 \in Q$

F = set of final states; $F \subseteq Q$.

General Notations of DFA

There are two preferred notation for describing DFA.

- Transition table (Tabular representation).
- Transition diagram (Graphical representation).

Transition table:

In tabular representation, each DFA is represented by table with rows representing states and column representing inputs symbols. The intersection of rows and columns is called cell which represents the next state. Starting state is represented by '→' and all the accepting states are represented by '*' (star) as shown in figure below:

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

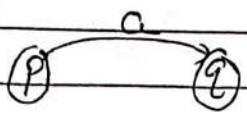
fig: Tabular representation of DFA accepting all strings taken from $\Sigma = \{0,1\}$ having 00 at the end.

Transition diagram:

Every DFA can be represented by graph with nodes representing states of the DFA and arcs represents the transition function.

Example: If there is a transition function $\delta(p,a) = q$ in a given DFA, then the graphical representation of that DFA contains

two nodes with label P&Q & three is an arc from P to q with label a, as shown in figure.



Rules to construct graphical representation of DFA.

1) For each state in the DFA there must be a node in graphical representation.
2) For each transition function $\delta(P, a) = q$, there is a arc from state P to q with a label 'a'.
3) There must be an input arc to starting state with label "START".
4) Accepting states are represented by double circle.

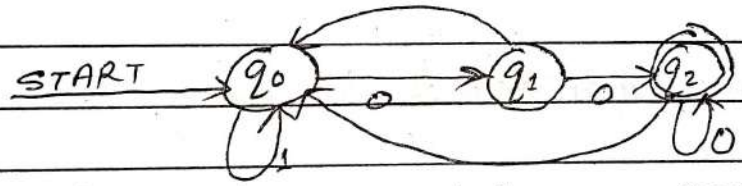


fig: Graphical representation of DFA accepting all the strings taken from $\Sigma = \{0, 1\}$ that ends with 00.

Some Examples:

⇒ Define a DFA over $\{0, 1\}$ accepting $\{1, 01\}$

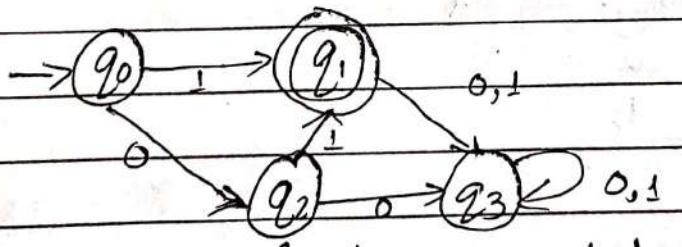


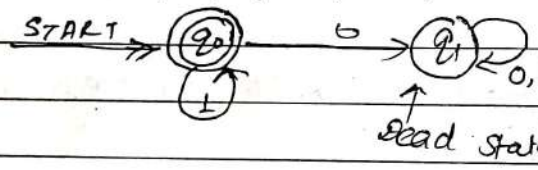
fig:- Graphical representation.

	0	1
$\rightarrow q_0$	q_2	q_1
$*q_1$	q_3	q_3
q_2	q_3	q_1
q_3	q_3	q_3

fig:- Tabular representation.

Define a DFA accepting zero or more consecutive is
i.e $L(M) = \{1^n / n = 0, 1, 2, \ldots\}$

$L = \{\epsilon, 1, 11, 111, 1111, \ldots\}$, $L' = \{0, 01, 10, 00, 000, 001 \ldots\}$



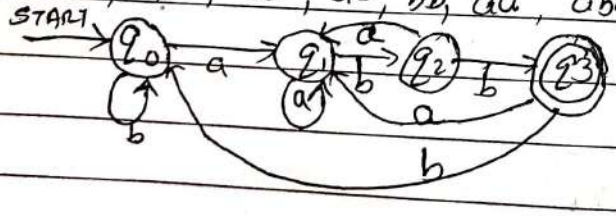
Define DFA over $\{a, b\}$ that accepts the strings ending with abb.

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

$L(M) = \{wabb / w$ is a string over $\{a, b\}\}$

$L = \{abb, aabb, aaabb, aaaabb \ldots\ babb, bbabb, ababb, ababbb, \ldots\}$

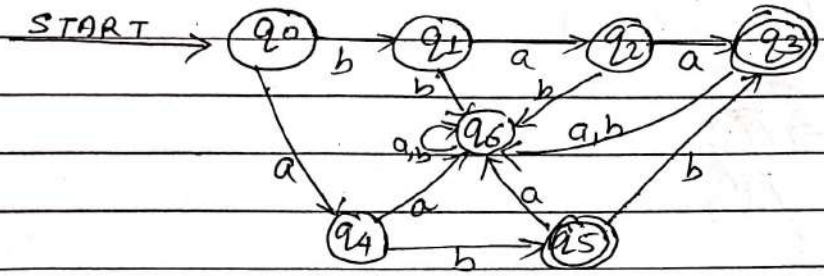
$L' = \{\epsilon, a, b, ba, ab, bb, aa, abab, \ldots\}$



Define a DFA over {a, b} accepting {baa, ab, abb}.

$L = \{baa, ab, abb\}$

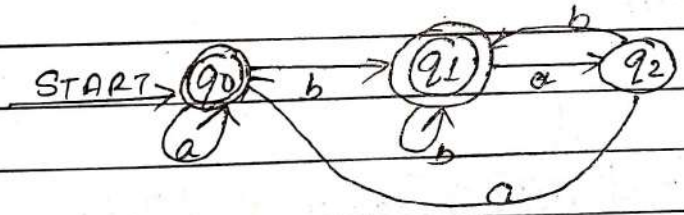
$L' = \{\epsilon, a, b, ba, baa, \ldots\ldots\}$



Define a DFA that accepts all the string over $\Sigma = \{a, b\}$ that do not end with {ba}

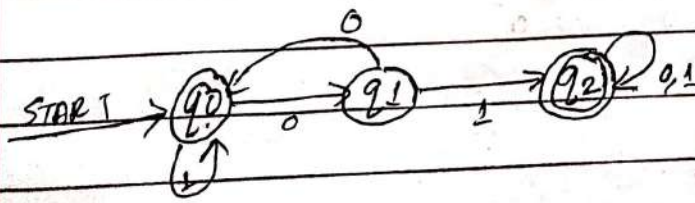
$L = \{\epsilon, a, b, ab, abb, baab, \ldots\ldots\}$

$L' = \{aba, bba, bbba, aba, aaba, \ldots\ldots\}$

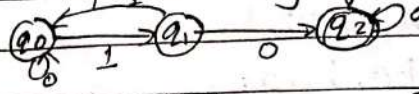


Create a DFA that accepts a language of all strings over $\Sigma = \{0, 1\}$ which contain {01} as substrings.

$L = \{01, 001, 101, 000111, 00101001\}$

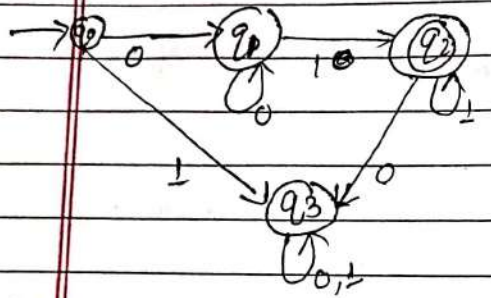


- Define a DFA to accept a string that contain all the strings from $\{0,1\}$ with (0,0) as sub string.

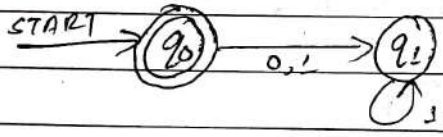


- Create a DFA for $L = \{0^{n,m} / n>0, m>0\}$

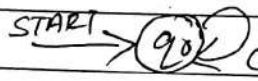
$L = \{01, 001, 011, \ldots\}$



Design a DFA for $L = \{\epsilon\}$ defined over $\Sigma = \{0,1\}$



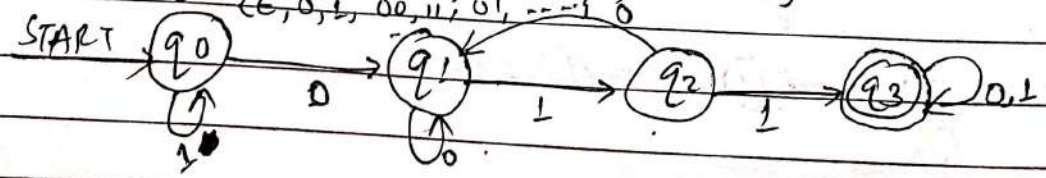
Design a DFA for $L = \phi$ defined over $\Sigma = \{0,1\}$



Define a DFA to accept the language that contains all the strings from $\Sigma = \{0,1\}$ with 0,11 as substring.

$L = \{011, 0011, 01100, 001100, \ldots\}$

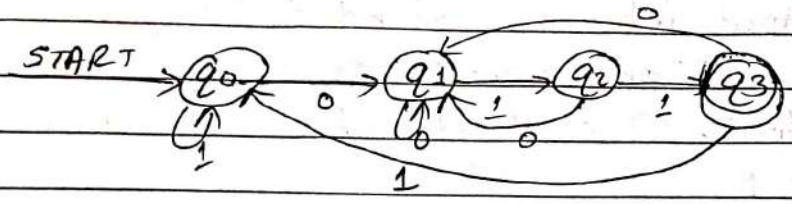
$L' = \{\epsilon, 0, 1, 00, 11, 01, \ldots\}$



Design a DFA to accept a language that contain all the strings from $\Sigma = \{0, 1\}$ and ends with 011.

$L = \{011, 0011, 00011, 1011, 11011, \ldots\}$

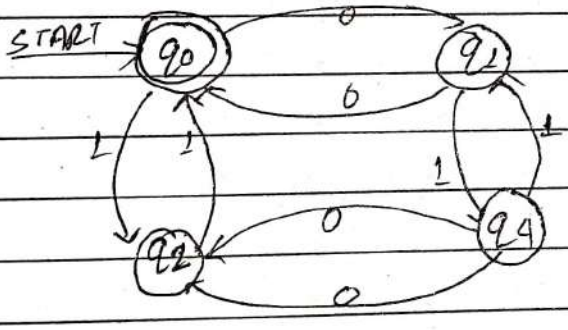
$L' = \{\epsilon, 0, 1, 0100, 0110, 1101 \ldots\}$



Create a DFA to accept the language that contains string from $\Sigma = \{0, 1\}$ made up of even number of zeros and even number of 1.

$L = \{\epsilon, 00, 11, 0101, 0011, 00001111, 1100, 1010, \ldots\}$

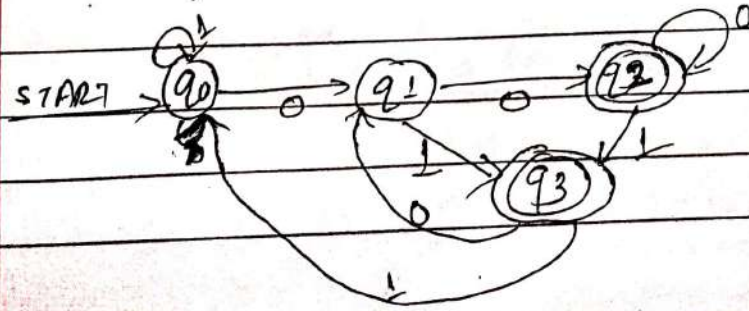
$L' = \{0, 1, 101, 10, 01, 0001, 111, 000, 100, \ldots\}$



Define a DFA to accept the language of type 'w0a' where w is a string and a is either 0 or 1.

$L = \{$string have 0 at second last position & 0 or 1 at last position$\}$

$= \{00, 01, 000, 001, 0100, 0101, 1001, \ldots\}$

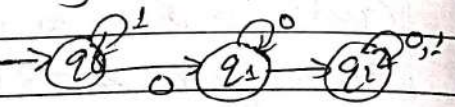


EXTENDED TRANSITION FUNCTION OF D.F.A.

Normal transition function denoted by "δ" takes a state and an input symbol as input and produce a state as output. whereas extended transition function takes a state and an input string as input and produce a state as output. It is denoted by "$\hat{\delta}$" and is defined as $\hat{\delta}(q, w) = p$

where, q is a state, w is a string & p is also a state.

Example:



$\hat{\delta}(q_0, 01101) = q_2 \in F$

$= \hat{\delta}(\delta(q_0, 0), 1101)$

$= \hat{\delta}(\delta, 1101)$

$= \hat{\delta}(\delta(q_1, 1), 101)$

$= \hat{\delta}(q_2, 101)$

$= \hat{\delta}(\delta(q_2, 1), 01)$

$= \hat{\delta}(q_2, 01)$

$= \hat{\delta}(\delta(q_2, 0), 1)$

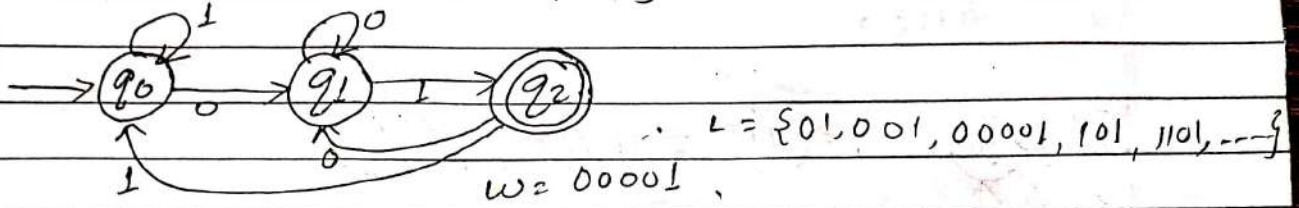
$= \delta(q_2, 1)$

$= q_2$

Language of DFA

Let $D = (Q, \Sigma, \delta, q_0, F)$ be a DFA then language of DFA denoted by $L(D)$ can be defined as:

$L(D) = \{w / \hat{\delta}(q_0, w) = p \in F\}$



$\therefore L = \{01, 001, 00001, 101, 1101, \ldots\}$

$w = 00001$.

$L = \{w / \hat{\delta}(q_0, w) \in F\}$

$L = \{00001 / \hat{\delta}(q_0, 00001) = q_2 \in F\}$

Non-deterministic finite Automata

In NFA, the control from state to state with respect to some input as non-deterministic i.e The transition function might take a control from a state to several states (including 0) at once. Both DFA and NFA accepts exactly same language known as regular language.

The transition function in case of NFA takes a state and input symbol as input and returns a set of zero, one, or more than one states (rather than returning exactly one state like DFA.

Formally, a NFA 'N' consists of $A = (Q, \Sigma, \delta, q_0, F)$ where Q = finite set of states.

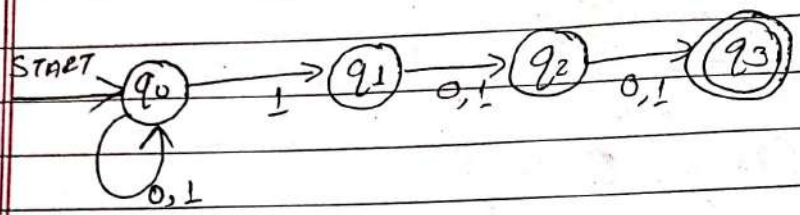
Σ = finite set of input symbols.

δ = a transition function that takes a state and input symbol as input and produce a set of states as output. The output set might be ϕ.

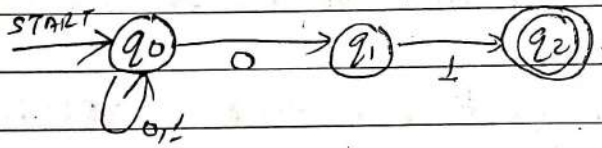
q_0: a start state

F: a set of final states $F \subseteq Q$

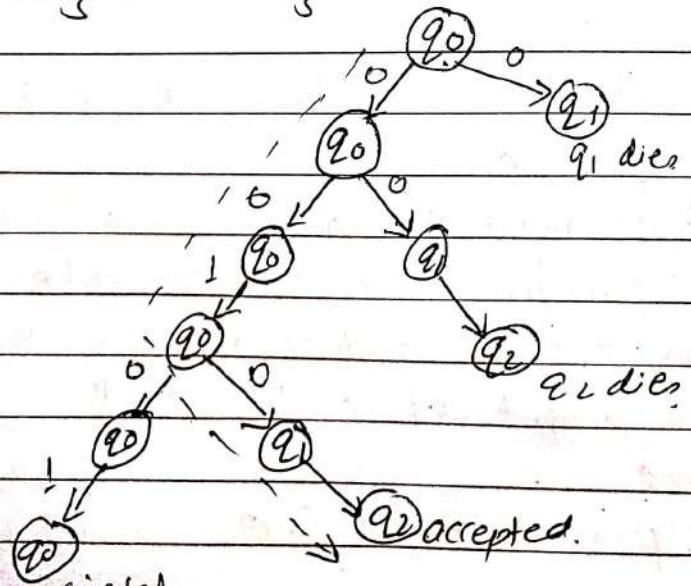
Some example:
Create a NFA accepting all the strings over $\Sigma = \{0,1\}$ that contains 1 at third last position.
$L = \{0100, 100, 1100, 1101, 110, 101, 1111, 111, 00100, 11100, 01100, \ldots\}$



Create a NFA accepting all the strings over $\Sigma = \{0,1\}$ that end in 0.1 and show that how it accept a string $w = 00101$.
$L = \{101, 1101, 0001, 1001, 001, 0101 \ldots\}$



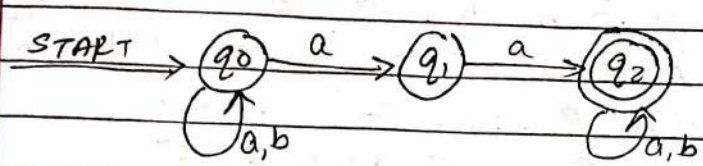
Using tree diagram.



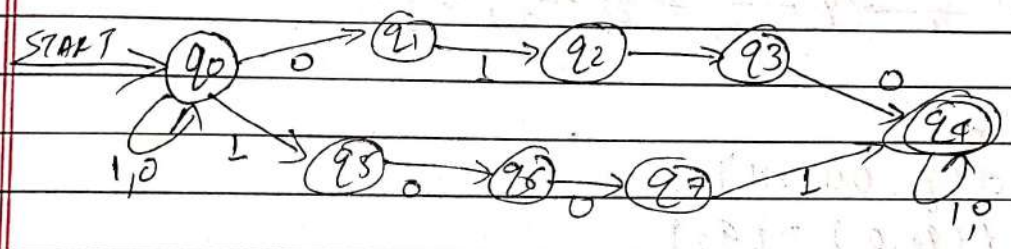
Equivalent Transition table

	0	1
→ q0	{q0, q1}	{q0}
q1	ϕ	q2
* q2	ϕ	ϕ

Construct NFA over $\Sigma = \{a, b\}$ that accepts strings having aa as substring.

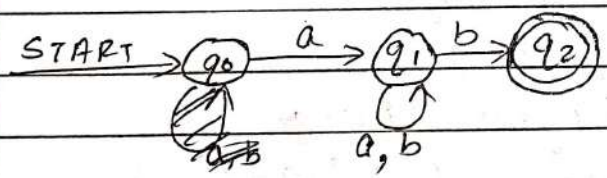


Construct NFA over $\Sigma = \{0, 1\}$ that accepts all the strings containing 0110 or 1001 as sub string.



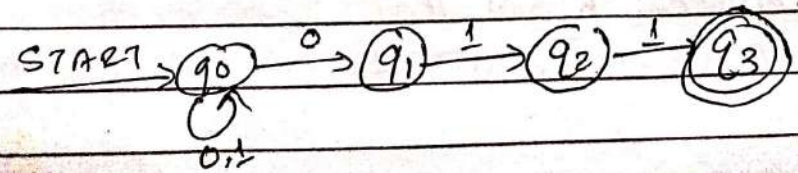
Construct a NFA over $\{a, b\}$ that accepts strings with 'a' and ending with b.

$L = \{aab, ab, abab, abb, \dots\}$



Construct NFA over $\{0, 1\}$ to accept the language that contains all strings with 011 at end.

$L = \{011, 0011, 1011, 11011, \dots\}$



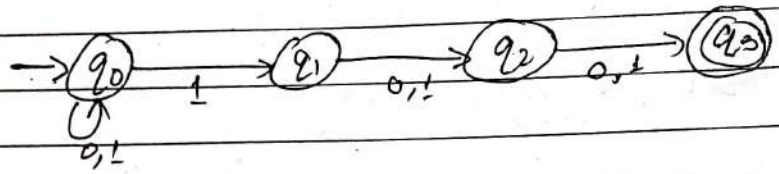
Extended Transition function of NFA.

Let $N = (Q, \Sigma, \delta, q_0, F)$ be a NFA then extended transition function takes a state and input string as input and produce a set of states as output.

i.e $\hat{\delta}(q, w) = \{p_1, p_2, p_3, \ldots p_n\}$ where,

$p_1, p_2, p_3, \ldots p_n \in Q$ and $w \in \Sigma^*$.

Given NFA find $\hat{\delta}(q_0, 001011)$.



Soln $\hat{\delta}(q_0, 001011)$

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

$= \hat{\delta}(q_0, 00) = \{q_0\}$

$= \hat{\delta}(q_0, 001) = \{q_0, q_1\}$

$= \hat{\delta}(q_0, 0010) = \delta(q_0, 0) \cup (q_2, 0)$

$= q_0 \cup q_2$

$= \{q_0, q_2\}$

$= \hat{\delta}(q_0, 00101) = \delta(q_0, 1) \cup \delta(q_2, 1)$

$= \{q_0, q_1\} \cup \{q_3\}$

$= \{q_0, q_1, q_3\}$

$= \hat{\delta}(q_0, 001011) = \delta(q_0, 1) \cup \delta(q_1, 1) \cup \delta(q_3, 1)$

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

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

∴ The string is rejected by NFA

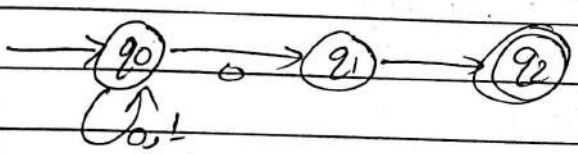
Language of NFA

Let $N = (Q, \Sigma, \delta, q_0, F)$ be a NFA then language of N. $L(N)$ is defined as:

$$L(N) = \{ w / \hat{\delta}(q_0, w) = \{p_1, p_2, \ldots p_k\} \cap F \neq \phi \}$$

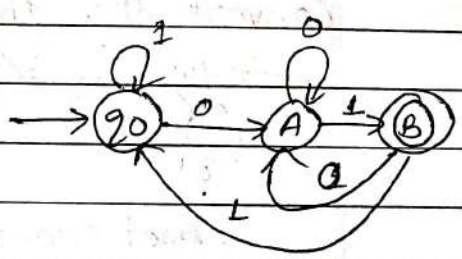
where $p_1, p_2, \ldots p_k \in Q$ & $w \subseteq \Sigma^*$

Conversion of NFA into DFA (using subset construction method)



	0	1
q_0	$\{q_0, q_1\} = A$	q_0
$\{q_0, q_1\} = A$	$\{q_0, q_1\} = A$	$\{q_0, q_2\} = B$
$\{q_0, q_2\} = B$	$\{q_0, q_1\} = A$	$\{q_0\}$

⇒



Every language that can be described by some NFA can also be described by some DFA. Moreover the DFA in practice has about as many states as the NFA although it often has more transitions. In worst case however, the smallest NFA for same language has only 'n' states. The proof that DFAs can do whatever NFAs can do involves an important construction called the subset construction because it involves constructing the subset of the set of states of the NFA.

Let $N = (Q_N, \Sigma, \delta_N, q_0, F_N)$ be a given NFA than the equivalent DFA $D = (Q_D, \Sigma, \delta_D, q_0, F_D)$ can be constructed by subset construction method as

Q_D = set of all subsets of Q_N.

Σ = Same on both.

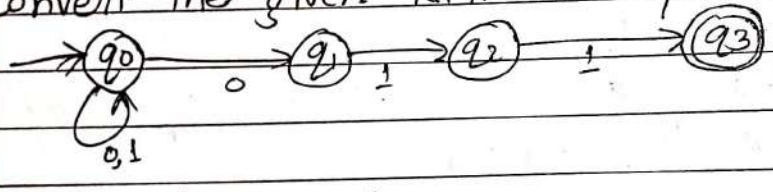
F_D = set of all subset of Q_N such that can be defined as $S \cap F_N \neq \phi$.

Let $S \in Q_N$, a subset of Q_N where

$S = \{p_1, p_2, p_3, \ldots p_k\}$

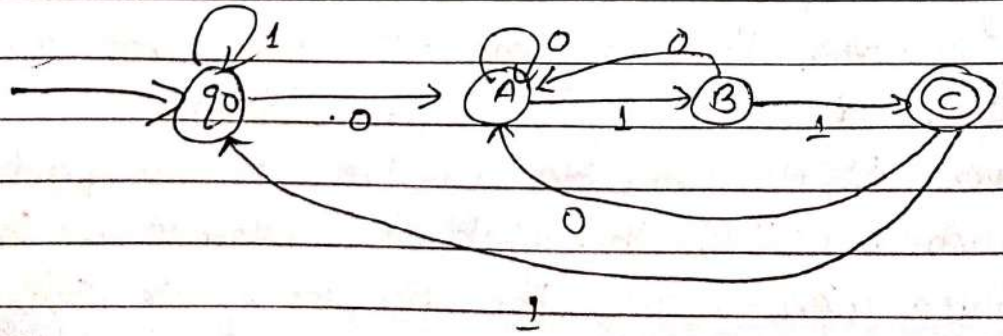
Then $\delta_N(S, a) = \bigcup_{i=1}^{k} \delta_N(p_i, a)$.

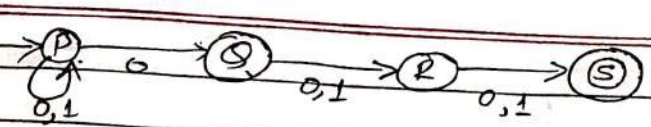
Convert the given NFA into Equivalent DFA.



Subset construction.

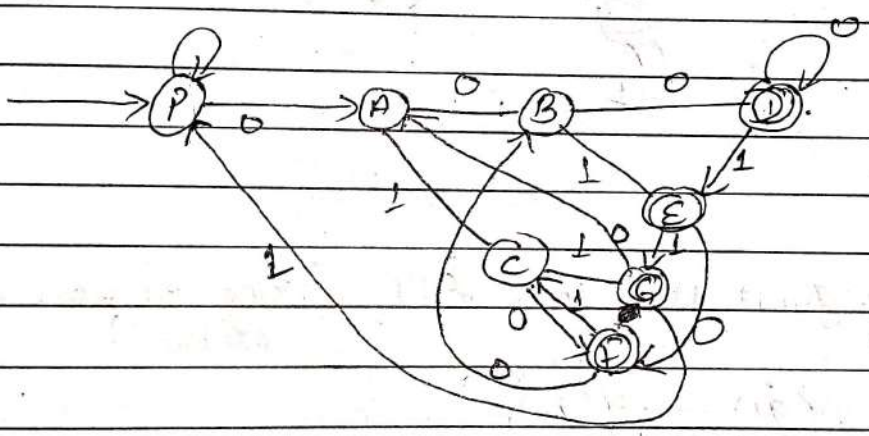
~~F_D~~	0	1
~~$\{q_0\}$ $\{q_0\}$ = A~~	$\{q_0, q_1\} = A$	q_0
$\{q_0, q_1\} = A$	$\{q_0, q_1\} = A$	$\{q_0, q_2\} = B$
$\{q_0, q_2\} = B$	$\{q_0, q_1\} = A$	$\{q_0, q_3\} = C$
$\{q_0, q_3\} = C$	$\{q_0, q_1\} = A$	$\{q_0\}$



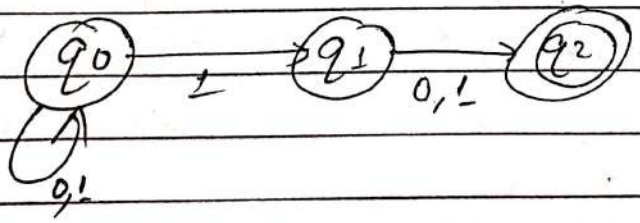


Subset construction

	0	1
P	{P,Q} = A	{P}
{P,Q} = A	{P,Q,R} = B	{P,R} = C
{P,Q,R} = B	{P,Q,R,S} = *D	{P,R,S} = *E
{P,R} = C	{P,Q,S} = *F	{P,S} = *G
{P,Q,R,S} = *D	{P,Q,R,S} = *D	{P,R,S} = *E
{P,R,S} = *E	{P,Q,S} = *F	{P,S} = *G
{P,Q,S} = *F	{P,Q,R} = B	{P,R} = C
{P,S} = *G	{P,Q} = A	{P} =

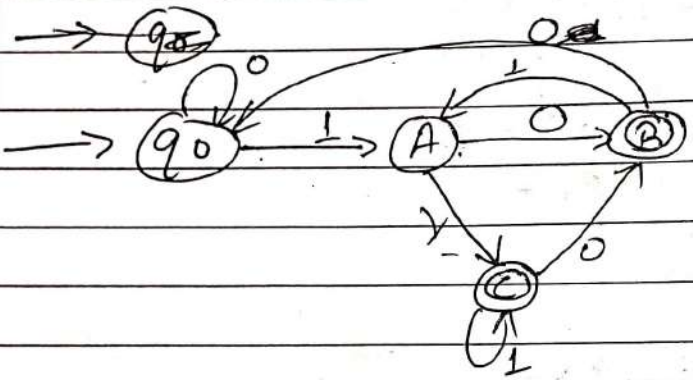


Convert the given NFA into equivalent DFA using subset construction method

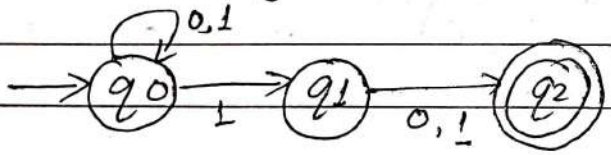


Subset construction

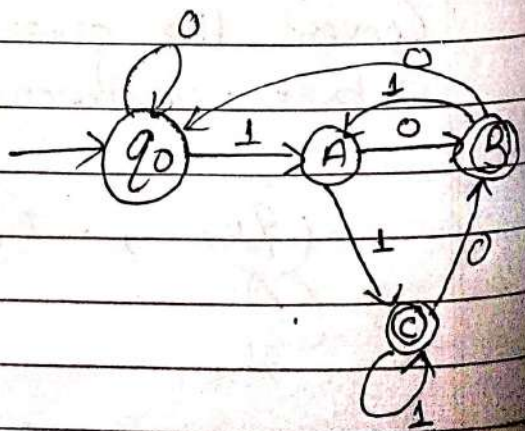
	0	1
q_0	q_0	$\{q_0, q_1\} = A$
$\{q_0, q_1\} = A$	$\{q_0, q_2\} = B$	$\{q_0, q_1, q_2\} = B^*C$
$\{q_0, q_2\} = B$	$\{q_0\}$	$\{q_0, q_1\} = A$
$\{q_0, q_1, q_2\} = C$	$\{q_0, q_2\} = B$	$\{q_0, q_1, q_2\} = C$



Convert the given NFA into DFA. (Using subset construction method)



	0	1
q_0	q_0	$\{q_0, q_1\} = A$
$\{q_0, q_1\} = A$	$\{q_0, q_2\} = B$	$\{q_0, q_1, q_2\} = C$
$\{q_0, q_2\} = B$	q_0	$\{q_0, q_1\} = A$
$\{q_0, q_1, q_2\} = C$	$\{q_0, q_2\} = B$	$\{q_0, q_1, q_2\} = C$



Theorem:

Let $D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$ be a DFA constructed from a NFA $N = (Q_N, \Sigma, \delta_N, q_0, F_N)$ by subset construction the $L(D) = L(N)$.

Proof:

To prove $L(D) = L(N)$, it is sufficient to prove $\hat{\delta}_D(q_0, w) = \hat{\delta}_N(q_0, w)$ $\forall$ $w \in \Sigma^*$

Using mathematical induction on length of w.

BASIS: let $|w| = 0$

Then $w = \epsilon$

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

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

$\therefore$ at basis step $\hat{\delta}_N(q_0, \epsilon) = \hat{\delta}_D(q_0, \epsilon)$

i.e $L(D) = L(N)$

INDUCTION

Let $w = xa$ where x is a string without last symbol & a is the last symbol.

Let w is the length $n+1$.

By induction hypothesis:

$\hat{\delta}_D(q_0, x) = \hat{\delta}_N(q_0, x) = \{p_1, p_2, p_3, \dots p_k\}$.

The inductive step:

$$\hat{\delta}_N(q_0, x_a) = \bigcup_{i=1}^{k} \delta_N(p_i, a) \quad \text{---} (1)$$

The subset construction on the other hand tells that

$\hat{\delta_D}(\{P_1, P_2, P_3 \ldots P_k\}, a) = \bigcup_{i=1}^{k} \delta_N(P_i, a)$ — (11)

Since $\hat{\delta_D}(q_0, xa) = \delta_D(\hat{\delta_D}(q_0, x), a)$

$= \delta_D(\{P_1, P_2, P_3, \ldots P_k\}, a)$

$\therefore \hat{\delta_D}(q_0, xa) = \bigcup_{i=1}^{k} (\delta_N(P_i, a)$ — eqn (11)

$\therefore \hat{\delta_N}(q_0, xa) = \hat{\delta_N}(q_0, xa)$ — eqn (11)

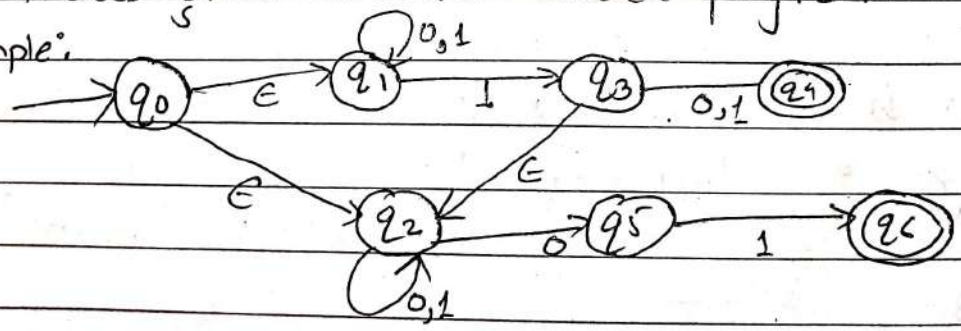
$\therefore L(D) = L(N)$ Hence proved

Epsilon NFA (ϵ-NFA)

Epsilon NFA is extended form of NFA. The new feature is that it allows a transition on Epsilon, the empty string. In effect an NFA is allowed to make a transition spontaneously without receiving an input symbol.

This new capability doesnot expand the class of languages that can be accepted by finite automata but it does gives us some added "programming convenience".

example:



Formally:

An ϵ-NFA $E = (Q, \Sigma, \delta, q_0, F)$ consists of

Q = a set of finite states

Σ = a set of finite input symbol.

q_0 = a starting state $q_0 \in Q$

F = set of accepting states $F \subseteq Q$

δ = a transition function, that takes a state and a symbol from $\Sigma \cup \{\epsilon\}$ as input and ~~procedure~~ produce a set of states as output.

i.e $\delta(q, a)$ $\{q_1, q_2, q_3 \ldots q_k\}$ where $q_1, q_2 \ldots q_k \in Q$ & $a \in \Sigma \cup \{\epsilon\}$

ϵ-closure of a state [ECLOSE)

Let $E = (Q, \Sigma, \delta, q_0, F)$ be an ϵ NFA.

If $q \in Q$ be a state then ECLOSE (q) can be recursive defined as.

BASIS: $q \in$ ECLOSE (q)

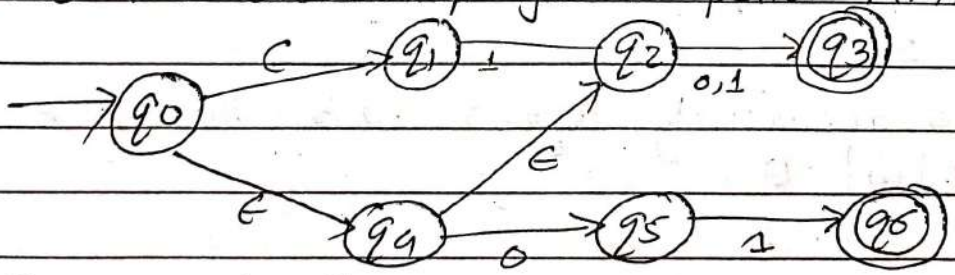
i.e ECLOSE $(q) = \{q\}$

INDUCTIVE:

if $P \in$ ECLOSE (q) & $\delta(P, \epsilon)$ contain r.

then $r \in$ ECLOSE (q)

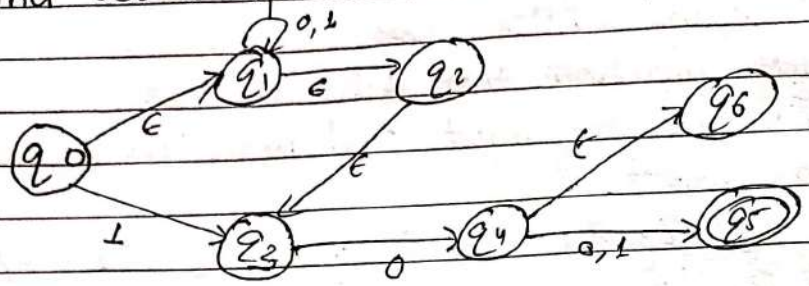
Find the ECLOSE $^{(q_0)}$ of given Epsilon NFA



ECLOSE $(q_0) = \{q_0, q_1, q_4, q_2\}$

ECLOSE $(q_6) = \{q_6\}$

Q. Find ECLOSE of all the states of given ENFA.



$\varepsilon CLOSE(q_0) = \{q_0, q_1, q_2, q_3\}$
$\varepsilon CLOSE(q_1) = \{q_1, q_2, q_3\}$
$\varepsilon CLOSE(q_2) = \{q_1, q_3\}$
$\varepsilon CLOSE(q_3) = \{q_3\}$
$\varepsilon CLOSE(q_4) = \{q_4, q_6\}$
$\varepsilon CLOSE(q_5) = \{q_5\}$
$\varepsilon CLOSE(q_6) = \{q_6\}$

Extended Transition Function of ENFA. ($\hat{\delta}$)

The extended transition function ($\hat{\delta}$) takes a state and an input string as input and produce a set of state as output. i.e

i.e $\hat{\delta}(P, w) = \{P_1, P_2, P_3, P_k\}$, where $P_1, P_2, P_3 \ldots, P_k \in Q$ & $w \in \Sigma^*$

$\therefore \hat{\delta}$ can be recursively defined as on the length of w as

Basis: if $|w| = 0$

then, $w = \varepsilon$

$\hat{\delta}(q_0, w) = \hat{\delta}(q_0, \varepsilon) = \varepsilon CLOSE(q)$

Inductive:

let $w = xa$, x is all the string except last symbol and a is last symbol.

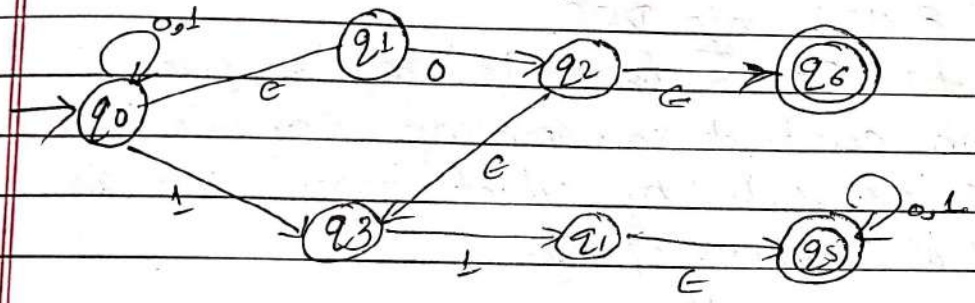
let,

$\hat{\delta}(q, x) = \{P_1, P_2, P_3, \ldots, P_k\}$

& $\hat{\delta}(P_i, a) = \{r_1, r_2, r_3, \ldots r_m\}$

$\therefore \hat{\delta}(q, w) = \bigcup_{i=1}^{m} \text{ECLOSE}(r_i)$

Eg: from given ENFA find $\hat{\delta}(q_0, 101)$



$\hat{\delta}(q_0, 101) = \hat{\delta}(\delta(q_0, 1), 01)$

$= \hat{\delta}(\text{ECLOSE}(\delta(\text{ECLOSE}(q_0), 1), 01)$

$= \hat{\delta}(\text{ECLOSE}(\delta(\{q_0, q_1\}, 1), 01)$

$= \hat{\delta}(\text{ECLOSE}(q_0, q_3), 01)$

$= \hat{\delta}(\{q_0, q_1, q_3\}, 01)$

$= \hat{\delta}(\delta(q_0, q_1, q_3), 0), 1)$

$= \hat{\delta}\,\text{ECLOSE}(\delta\,\text{ECLOSE}(q_0, q_1, q_3), 0), 1)$

$= \hat{\delta}(\text{ECLOSE}(\{q_0, q_2\}, 1)$

$= \hat{\delta}(\text{ECLOSE}(q_0, q_3, q_4), 1)$

$= \hat{\delta}(\{q_0, q_1, q_3, q_4, q_5\}, 1)$

$= \text{ECLOSE}(\delta(q_0, q_1, q_3, q_6), \ldots)$

$= \text{ECLOSE}(\delta(q_0, q_1, q_2, q_3, q_6, \ldots), 1)$

$= \text{ECLOSE}(q_0, q_3, q_4, \ldots)$

$= (q_0, q_1, q_3, q_4, q_5)$.

Language of ε-NFA

Let $E = (Q, \Sigma, \delta, q_0, F)$ be a given ε-NFA then language of ε-NFA $L(E)$ is defined as

$L(E) = \{ w \mid \hat{\delta}(q_0, w) = \{p_1, p_2, \ldots p_k\} \cap F \neq \phi \}$

i.e the language of E is the set of strings w that take the START state to at least 1 accepting state.

Converting ε-NFA into DFA.

Let $E = (Q_E, \Sigma, \delta_E, q_0, F_E)$ be the given ε-NFA then we can construct corresponding equivalent DFA $D = (Q_D, \Sigma, \delta_D, q_0, F_D)$ by subset construction as

Q_D = set of subsets of Q_E

Σ is same on both

$q_0 = ECLOSE(q_0)$:

δ_D is defined as

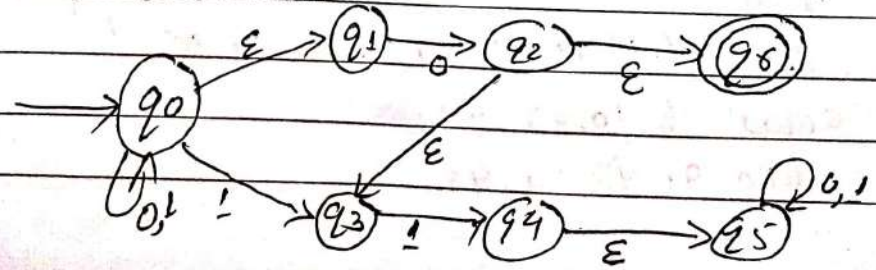
let $\{P_1, P_2, \ldots P_k\} \in Q_D$.

$\delta_D(S, a)$ where $a \in \Sigma$ can be constructed as

$\bigcup_{i=1}^{k} \delta_E(P_i, a) = (r_1, r_2, \ldots r_m)$

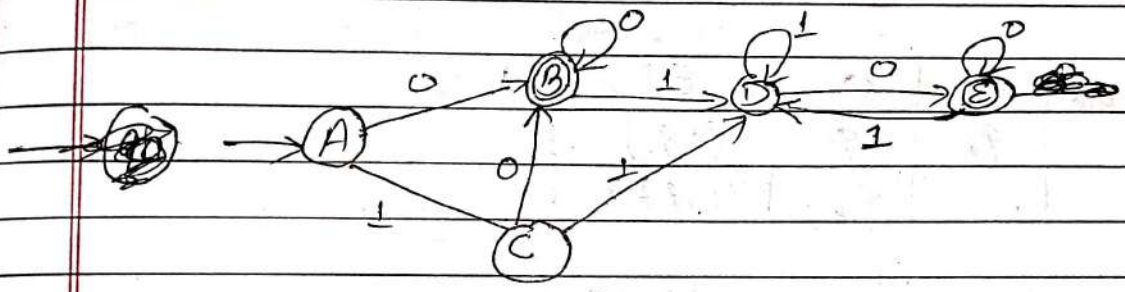
$\therefore \hat{\delta}_D(S, a) = \bigcup_{i=1}^{m} ECLOSE(r_i)$.

Convert the given ε-NFA into equivalent DFA using subset construction



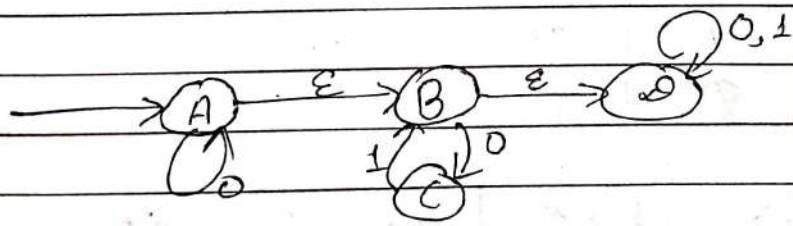
Subset construction.

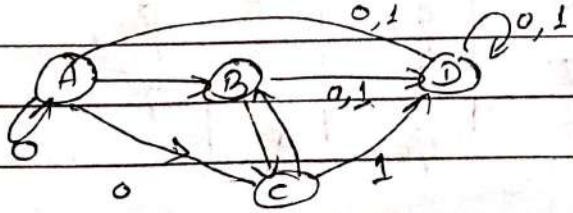
	0	1
$\{q_0, q_1\} = A$	$\{q_0, q_1, q_2, q_3, q_6\} = B$	$\{q_0, q_1, q_3\} = C$
$\{q_0, q_1, q_2, q_3, q_6\} = B$	$\{q_0, q_1, q_2, q_3, q_6\} = B$	$\{q_0, q_1, q_3, q_4, q_5\} = D$
$\{q_0, q_1, q_3\} = C$	$\{q_0, q_1, q_2, q_3, q_6\} = B$	$\{q_0, q_1, q_3, q_4, q_5\} = D$
$\{q_0, q_1, q_3, q_4, q_5\} = D$	$\{q_0, q_1, q_2, q_3, q_5, q_6\} = E$	$\{q_0, q_1, q_3, q_4, q_5\} = D$
$\{q_0, q_1, q_2, q_3, q_5, q_6\} = E$	$\{q_0, q_1, q_2, q_3, q_6, q_5\} = E$	$\{q_0, q_1, q_3, q_4, q_5\} = D$

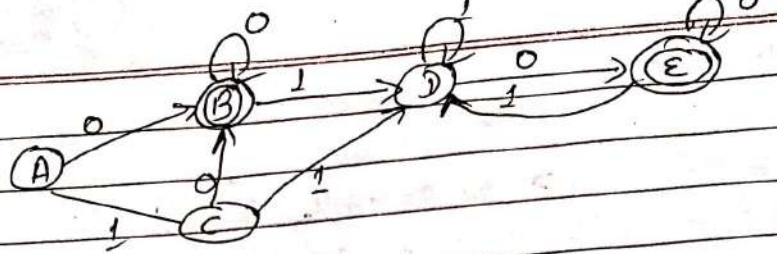


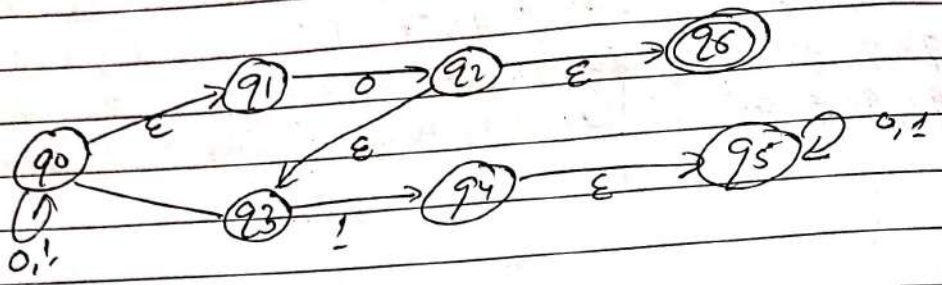
Elimination of ϵ-transition using the concept of ϵ-closure. (Convert ϵNFA to NFA)

Convert the following ϵNFA into equivalent NFA









(q_0)	ε^*	0	ε^*
	q_0	q_0	$\{q_0, q_1\}$
	q_1	q_2	$\{q_2, q_3, q_6\}$

$(q_0, 1)$	ε^*	1	ε^*
	q_0	q_0, q_3	$\{q_0, q_1\}$
	q_1		$\{q_3\}$

$(q_1, 0)$	ε^*	0	ε^*
	q_1	q_2	q_2, q_3, q_6

$(q_1, 1)$	ε^*	1	ε^*
	q_1	ϕ	ϕ

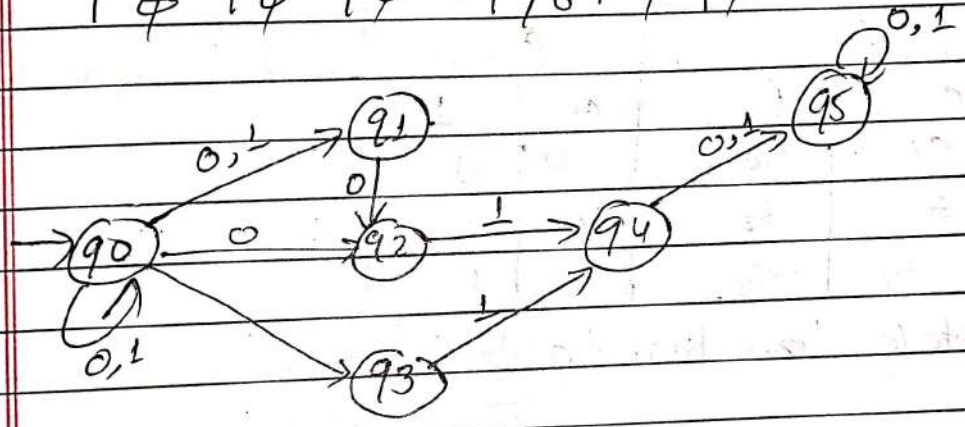
$(q_2, 0)$	ε^*	0	ε^*	ε^*	1	ε^*
	q_2	ϕ	ϕ	q_2	ϕ	q_4,
	q_3	ϕ	ϕ	q_3	q_4	q_5
	q_6	ϕ	ϕ	q_6	ϕ	

q_3	ε^*	0	ε^*	ε^*	1	ε^*
	q_3	ϕ	ϕ	q_3	q_4	q_4, q_5

q_4	ε^*	0	ε^*	ε^*	1	ε^*
	q_4	ϕ	ϕq_5	q_4	ϕ	ϕq_5
	q_5	q_5		q_5	q_5	

q_5	ε^*	0	ε^*	ε^*	1	ε^*
	q_5	q_5	q_5	q_5	q_5	q_5.

q_6	ε^*	0	ε^*	ε^*	1	ε^*
	ϕ	ϕ	ϕ	q_6	ϕ	ϕ



finite state machine with output:
Moore Machine and Mealy Machines.

Example 1:
Consider the following automatam with output as defined below.

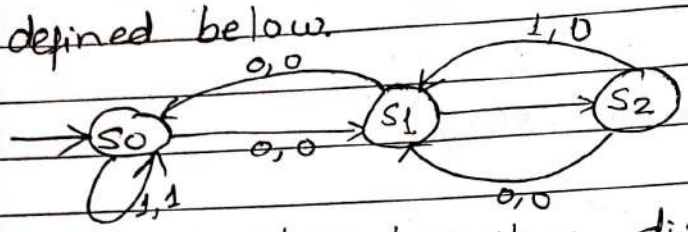


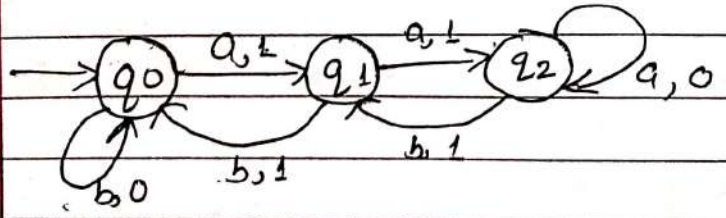
fig:- state transition diagram.

	δ		f	
	0	1	0	1
S_0	S_1	S_0	0	1
S_1	S_0	S_2	0	1
S_2	S_1	S_2	0	1

fig:- state transition table.

In above FSM the output generated from the input string 01110 is 01010 and the state transition sequence is $S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow S_1 \rightarrow S_2 \rightarrow S_1$.

Example 2:
Consider another example:



	δ		f	
	a	b	a	b
q_0	q_1	q_0	1	0
q_1	q_2	q_0	1	1
q_2	q_2	q_1	0	1

fig: State transition table.

In above example for the input string baaa the automaton generates the output 0110 and the transition sequence is $q_0 \to q_0 \to q_1 \to q_2 \to q_2$.

Both of the machine are FSM with output and categorised as Mealy machine.

Example 3:

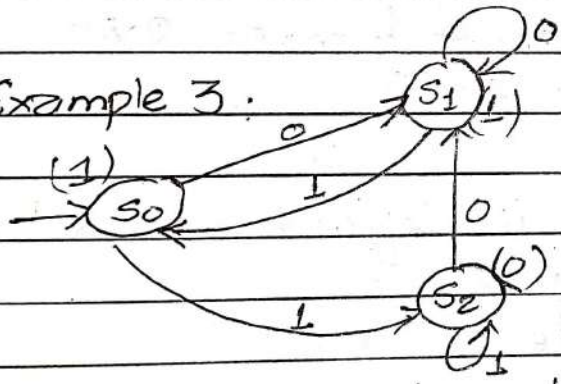


fig:- State transition diagram.

	δ		f
	0	1	
S_0	S_1	S_2	1
S_1	S_1	S_0	1
S_2	S_1	S_2	0

The above machine is a FSM with output which is categorised as Moore Machine.

Mealy machine is the finite state machine whose output values are determined by its current states and the current inputs.

Moore machine are finite state machine with output whose output values are solely determined by its current state.

Formally, Mealy machine is defined as six tuples $(Q, \Sigma, O, \delta, f, q_0)$ where;

Q = Finite set of states.
Σ = finite set of input alphabet.
O = Finite set of output alphabet.
δ = A transition function which maps $Q \times \Sigma \to Q$.
f = An output function that maps $Q \times \Sigma \to O$.
q_0 = A start state. $q_0 \in Q$.

Formally, Moore machine is defined as six tuples $(Q, \Sigma, O, \delta, f, q_0)$ where;

Q = Finite set of states
Σ = finite set of input symbols
O = finite set of output alphabet
δ = A transition function which maps $Q \times \Sigma \to Q$
f = An output function that maps $Q \to O$.
q_0 = A start state. $q_0 \in Q$

UNIT: 3

REGULAR EXPRESSION

Regular language:

An algebraic notation to represent the language is known as Regular expression.

Regular expression is an alternative to find an automata.

Language represented by regular expression is always regular. ie Regular expression defines same language as finite automata does. However regular expression offer something that automata don't. i.e A declarative way to express the strings we want to accept for many systems that process strings.

Examples:

$RE = 01^* + 10^*$

= set of strings that either starts with 0 followed by any number of 1's or starts with 1 & followed by any number of 0's.

$RE = 01^* + 10^*$

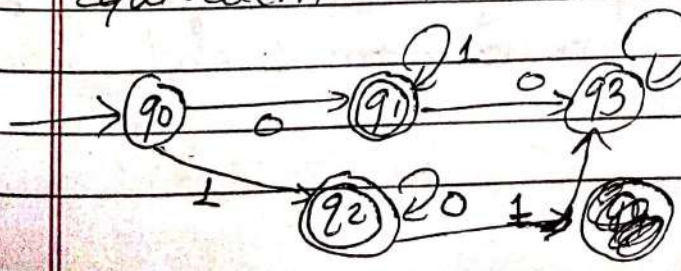
$= L(0) . L(1^*) + L(1) . L(0^*)$

$= \{0\} . \{\epsilon, 1, 11, 111, 1111 \ldots\} + \{1\} . \{\epsilon, 0, 00, 000, 0000, \ldots\}$

$= \{0, 01, 011, 0111, 01111 \ldots\} + \{1, 10, 100, 1000, 10000, \ldots\}$

$= \{0, 01, 011, 0111, 01111, 1, 10, 100, 1000, 10000, \ldots\}$

Equivalent DFA is.



Operators Associated with Language

1) Union:

Let 'L' and 'M' be two languages then union of 'L' and 'M' denoted by 'L ∪ M' or 'L + M'. If the set of strings that is either taken from 'L' or 'M'.

eg: Let $L = \{0, 11\}$

$M = \{\epsilon, 110, 11\}$

$L \cup M = \{\epsilon, 0, 11, 110\}$

2) Concatenation (dot) operator:

Let 'L' & 'M' be two languages then ~~cont~~ concatenation of 'L' & 'M' denoted by L.M or LM is set of strings formed by taking any string from 'L' & concatenating it with any string from 'M'.

i.e $L.M = \{xy / x \in L \ \& \ y \in M\}$

eg: If $L = \{0, 11\}$

$M = \{\epsilon, 110\}$

then $L.M = \{0, 11, 0110, 11110.\}$

3) Kleen Closure:

let L be a language, then Kleen Closure of 'L' denoted by L^* is the set of strings that are formed by first taking any number of strings from L and concatenating them.

eg: $L = \{0, 11\}$

$L^* = \{\epsilon, 011, 00, 01, 110, 1111, 000, 0011, 0110, 01111 \ldots\}$

Note: $\phi^* = \{\epsilon\}$

$\phi^0 = \{\epsilon\}$

$\epsilon^* = \{\epsilon\}$

Recursive defination of R.E.

Regular expression is an algebraic notation to represent the regular language, which is made up of constants and variables.

The defination of regular expression can be defined recursively as

Basis:

I: ϵ and ϕ are regular expression representing the language $\{\epsilon\}$ and ϕ itself.

Q. a is a regular expression representing the language $\{a\}$

INDUCTIVE:

i) Let E & F are regular expressions than E+F is a regular expression denoting the language:

$L(E+F) = L(E) + L(F) = L(E) \cup L(F)$

ii) Let E & F are regular expression than E.F is regular expression representing the language:

$L(E.F) = L(E) \cdot L(F) \neq L(F) \cdot L(E)$

iii) Let E be the regular expression than E^* is also a regular expression representing the language

$L(E^*) = (L(E))^*$

1) Find the language represented by 10^*.

solution:

$= L(1) \cdot L(0^*)$

$= \{1\}, \{\epsilon, 0, 00, 000, 0000 \ldots\}$

$= \{1, 10, 100, 1000, 10000, \ldots\}$

= All the strings that starts with 1 followed by any number of 0's.

2. Find the language of RE $(01)^*$

$= (1(01))^* = L(01^*)$

$= \{\epsilon, 01, 0101, 010101, 01010101, \ldots\}$

= All the strings that ~~starts with~~ contains any number of 01.

Q. $(1+\epsilon) \cdot 01 \cdot (0+1)^*$

$= L(1+\epsilon) \cdot L(01) \cdot L(0+1)^*$

$= \{1, \epsilon\}, \{01\} \cdot \{\epsilon, 0, 01, 001, 101, \ldots\}$

$= \{101, 01, 1010, 010, 10101, \ldots\}$

= Starting with 101 or 01 and followed any combination of 0 & 1.

Q. Regular expression $= (01 + 11)(01)^*$

$= L(01)+L(11) \cdot L((01)^*)$

$= \{01\}+\{11\} \cdot \{L(01)^*\}$

$= \{01, 11\} \cdot \{\epsilon, 01, 0101, 010101 \ldots\}$

$= 01, 11, 0101, 1101, 010101, 110101, 01010101, 11010101 \ldots$

Starting with 01 or 11 & followed by any combination of 01.

Q. Create a regular expression that represent the language of strings that contains 1 at 5th position from the right end.

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

Q. Create a regular expression that starts with 0 & ends with 1.

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

Create a regular expression from $\Sigma = \{0, 1\}$ that ends with ~~abb~~ 011

$= (0+1)^* \; 011$

Create a regular expression

Basic Rules To create Equivalent Regular Expression.

1) Identity and Annihilator rule:
   → $L + \phi = \phi + L$, ϕ is identity for union.
   → $L\varepsilon = \varepsilon L$, ε is identity for concatenation
   $L\phi = \phi = \phi L$, ϕ is annihilator for concatenation.

ii) Commutative law:
   $L + M = M + L$
   $LM \neq ML$

iii) Associative law
   → $(L+M) + N = L + (M+N)$
   → $(LM)N = L(MN)$

iv) Distributive law
   $L(M+N) = LM + LN$
   $(L+M)N = LN + MN$

v) Idempotent Rule.
   $L + L = L$

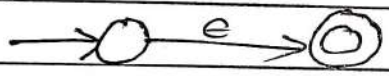
vi) Rule for closure.
   $(L^*)^* = L^*$
   $L^+ = L \cdot L^*$
   $0^* = \varepsilon$
   $\varepsilon^* = \varepsilon$
   $L^{+*} = L^* + \varepsilon$
   $L^? = \varepsilon + L$

Conversion of regular expression to ε-NFA.

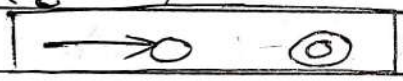
$(a+b)^* abb$

→ We can show that every language 'L' i.e. L(R) for some regular expression R, is also L for some ε-NFA, E. We start by showing here to construct automata for the basic expression, single symbols, ε & ϕ, we then show how to comb this automata into larger automata that accept union, concatenation or closure of the language accepted by smaller automata.

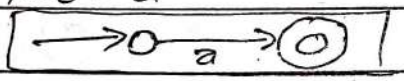
i) R.E = ε



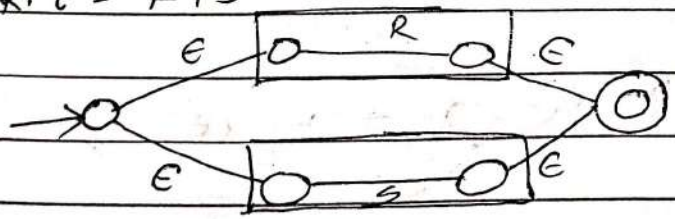
ii) R.E = ϕ



iii) R.E = a

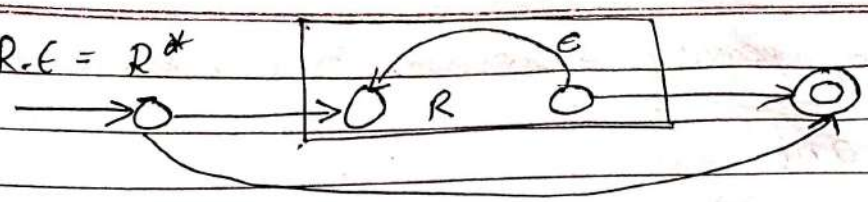


iv) R.E = R+S



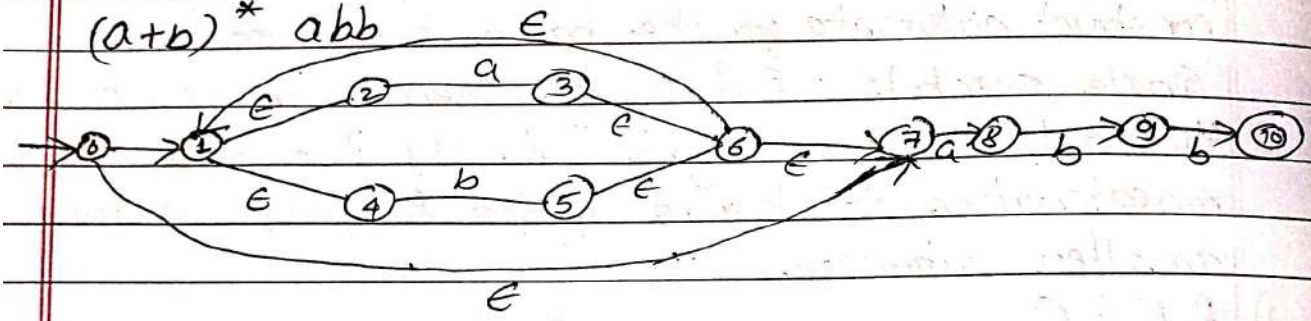
iv) R.E = R.S

v) $R.\epsilon = R^*$



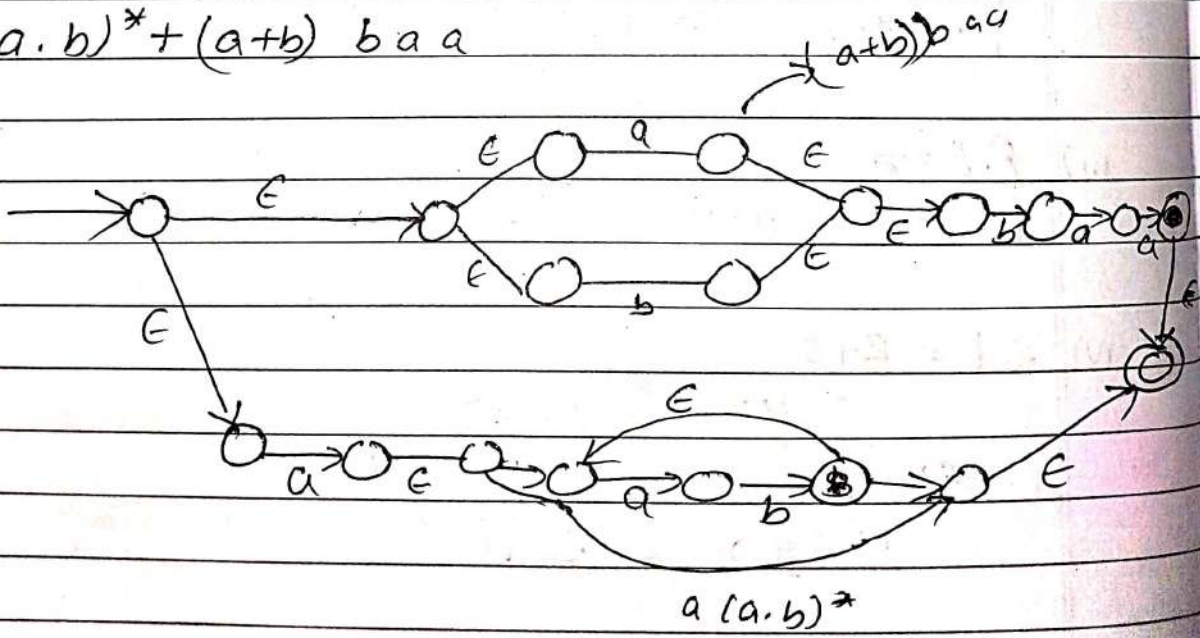
Design ~~gio~~ ε-NFA from given regular expression.

$(a+b)^*\ abb$



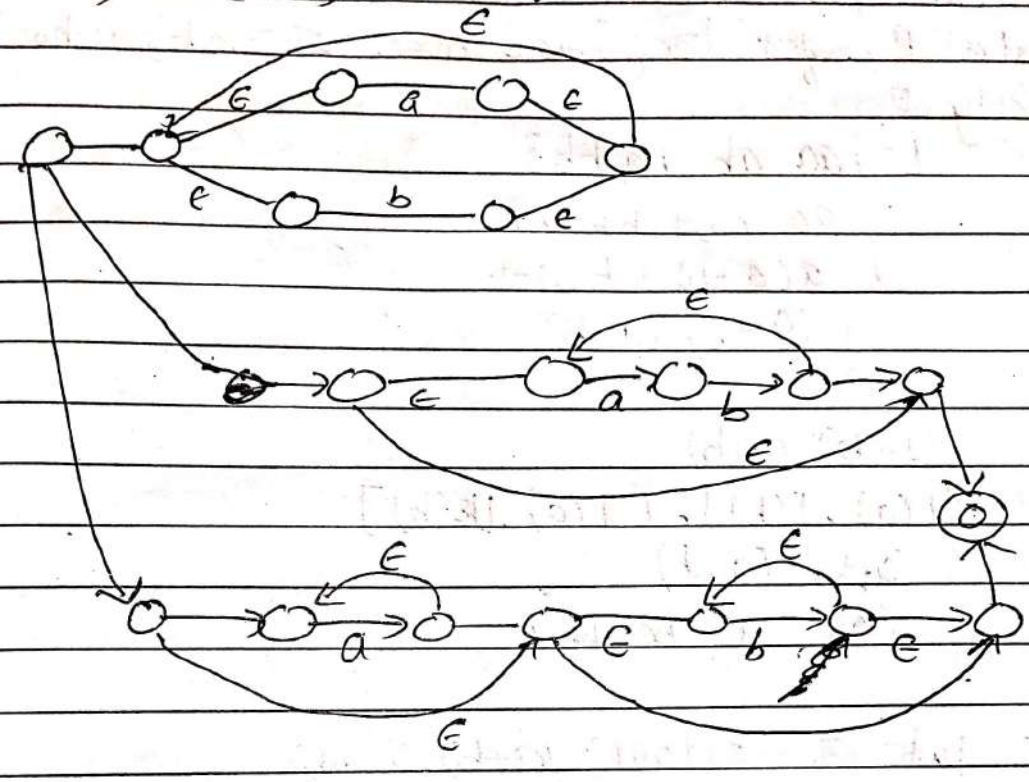
Convert the following RE into ε-NFA.

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



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

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



Questions for Creating R.E.

1) Create R.E for language over $\Sigma = \{a, b\}$ whose length is exactly 2.

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

$= aa + ab + ba + bb$

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

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

R.E $= (a+b)(a+b)$

$= [L(a), L(b)] \cdot [L(a), L(b)]$

$= (a, b)(a \cdot b)$

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

Hint: $|w| = 3 = (a+b)(a+b)(a+b)$

$|w| = 4 = (a+b)(a+b)(a+b)(a+b)$

2) Regular language where $|w|$ = at least 2.

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

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

3) RE for language where $|w|$ = at most 2.

$\{\epsilon, a, b, aa, ab, ba, bb\}$

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

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

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

$=$

4) R.E for language where |w| = even.
⇒ L = { ∈, aa, ab, ba, bb, aaaa, abab, baba, bbbb, aaaaaa, ababab, bababa, bbbbbb .... }
= ((a+b).(a+b))*

5) R.E for language where |w| = odd length strings.
⇒ L = { a, b, aaa, bbb, abb, aab, aabbb, ... }
= ((a+b).(a+b))*+1.

6) R.E for language where |w| = length is divisible by 3.
= L = ((a+b).(a+b).(a+b))*

7) R.E for language where |w| = 2 mod 3
(a+b)(a+b)((a+b)(a+b)(a+b))* + ((a+b)(a+b)(a+b))* (a+b)(a+b)
OR (a+b)(a+b)((a+b)(a+b)(a+b))*

e) Starting and ending with different symbols.
= a(a+b)*b + b(a+b)*a

g) Set of all string that contains no two edges together
= (b+ab)* + (b+ab)*a + a(b+ab)* + (b+ba)*

All th
Language = All the strings that contains no two a's and no two b's
= 

	starts with	ends with	
a, aba, abababa	a	a	a(ba)*
ab, abab, ababab	a	b	(ab)* a(ba)*b
ba, baba, bababa	b	a	(ba)* b(ab)*a
b, b(ab)*	b	b	b(ab)*

R.E = $a(ba)^* + (ab)^* + (ba)^* + b(ab)^*$

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

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

Minimization of the DFA: (imp)

1) Equivalent states:

Two states p and q are said to be equivalent if for every string, $w \in \Sigma^*$.

$\hat{\delta}(p,w)$ & $\hat{\delta}(q,w)$ are both either accepting states or non-accepting states

2) Distinguish states:

Two states p and q are said to be distinguish states if they are not equivalent. i.e. $\hat{\delta}(p,w)$ & $\hat{\delta}(q,w)$ are of opposite types

Table Filling Method [Myhill Nerode theorem]

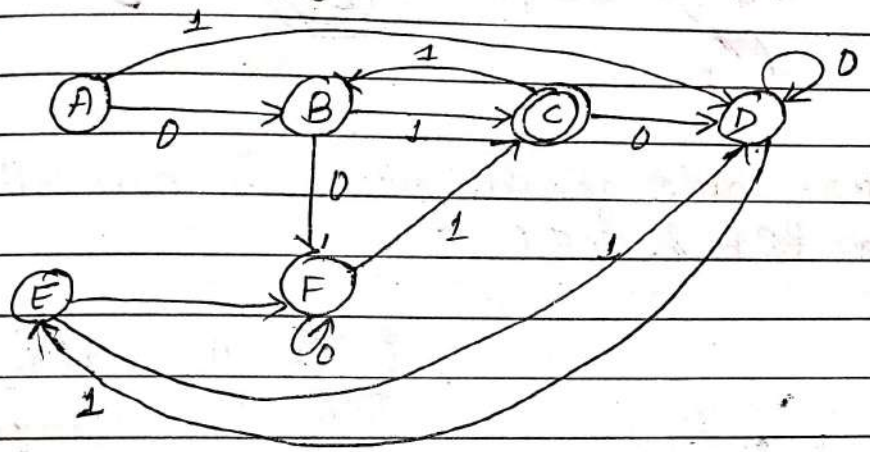
Algorithm:

1) Draw table for all pairs of states (P, Q)
2) Mark all pairs where $P \in F$ & $Q \notin F$
3) If there are any unmarked pair (P,Q) s.t $[\delta(P,x), \delta(Q,x)]$

marked then mark (p, q) where x is an input symbol. Repeat until no markings can be done.

4) Combine the unmarked pairs & make them a single state.

Q. Minimize the given DFA.



~~Steps~~ Minimised DFA is:

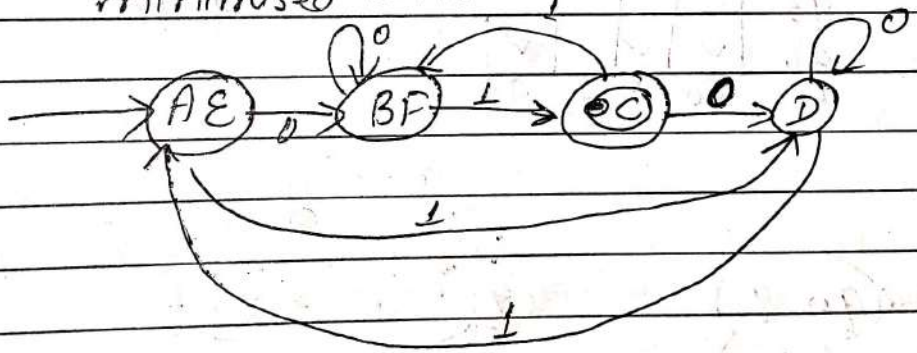
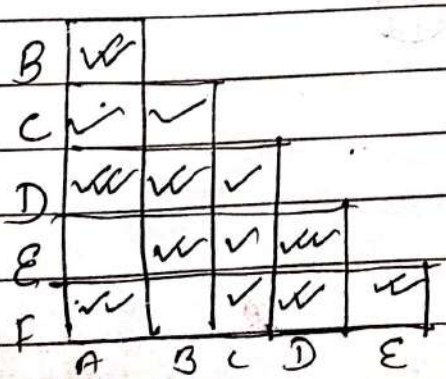
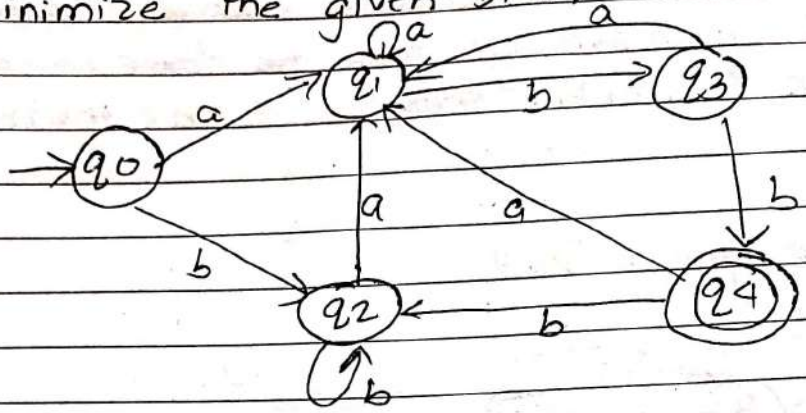


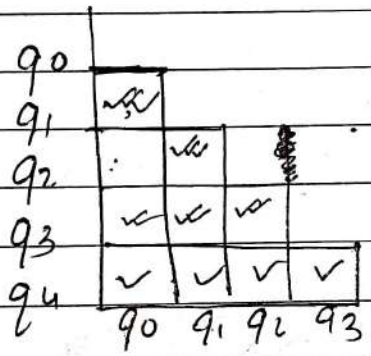
Table construction: Construct the table for all pairs & mark all pairs where P∈F & Q∉F



Q. Minimize the given DFA.



Step1: Construct table for all pairs & mark all pairs where $P \in F$ & $Q \notin F$.



$q_0 \to q_1 = a$ $\quad q_0 \to q_2 = b$

$q_1 \to q_1 = a$ $\quad q_2 \to q_2 = b$

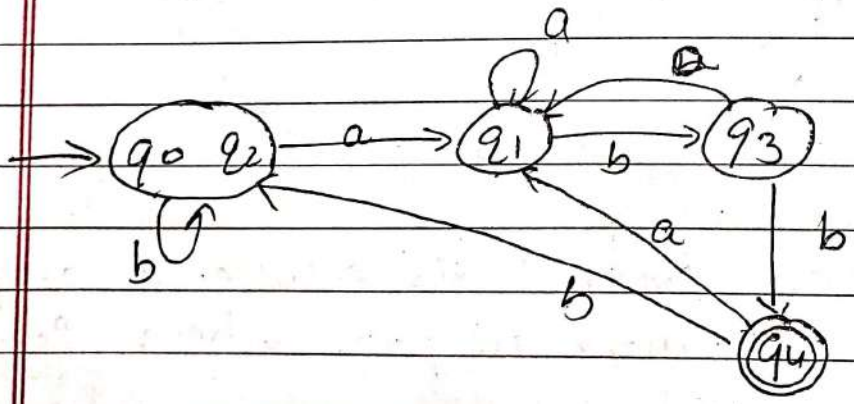
$q_2 \to q_1 = a$

$q_0 \to q_3 = x$ $\quad q_3 \to q_1 = a$

$q_1 \to q_3 = b$

$q_3 \to q_2 = x$

$q_2 \to q_3 = x$



Partitioning Method.

Step1: Remove all the unreachable states from q (start)

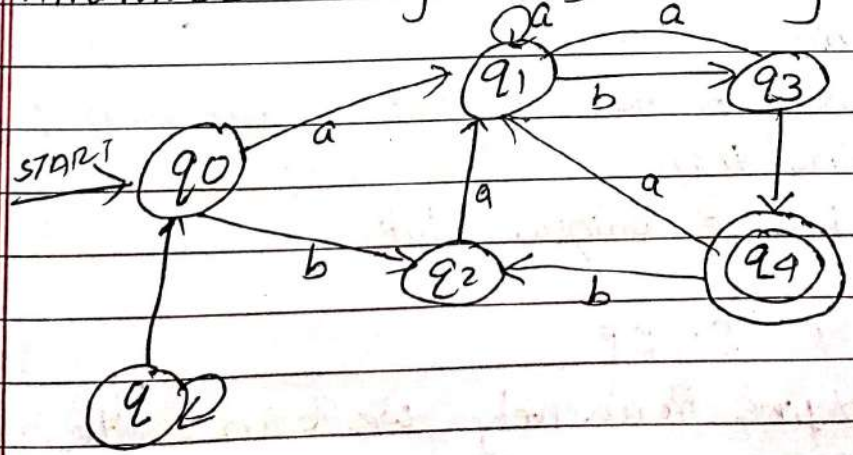
Step2: Construct state transition table for given DFA

Step3: Partition the states using 0-equivalence. 0-equivalence can be defined as:
- Two disjoint sets P & Q such that P contains only non-accepting states & Q contains only accepting states.

Step4: Again partition 0-equivalence into 1 equivalence can be defined as 'Let P be a 0-equivalence set and q & p are arbitary states in the set P.
If $\delta(q,x)$ & $\delta(p,x)$ ∈ same set in 0-equivalence then q & p are 1-equivalent else partition.

Step 5: Similarly compute 2-equivalent & 3-equivalent and so on. Repeat the process until no partition can be done.

Minimize the given DFA using partition method.



Step1: Remove q_5 because it is unreachable state.

Step 2: Construct ST.T.

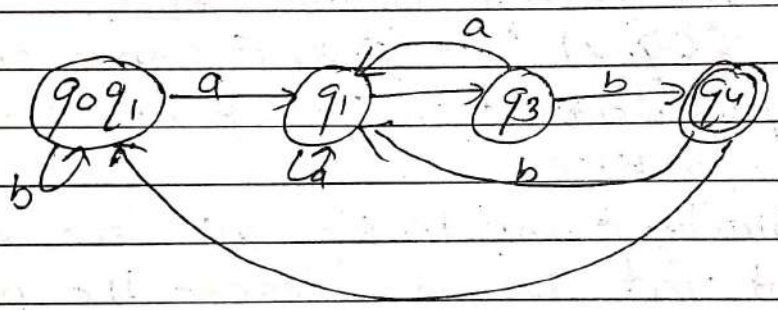
	a	b
$\rightarrow q_0$	q_1	q_2
q_1	q_1	q_3
q_2	q_1	q_2
q_3	q_1	$*q_4$
$*q_4$	q_1	q_2

Step3: 0 equivalent : $[q_0, q_1, q_2, q_3][q_4]$

1 - equivalent : $[q_0, q_1, q_2], [q_3][q_4]$

2 - equivalent : $[q_0, q_2][q_1][q_3][q_4]$

3 - equivalent : $[q_0, q_2][q_1][q_3][q_4]$



Conversion of Finite Automata To Regular Expression

Arden's Theorem:

Let P & Q be two regular expression if P doesn't contain null string then,

$R = Q + RP$ has a unique solution.

$R = QP^*$

Proof: Given: $R = Q + RP$

As R is define recursively we can write,

$R = Q + (Q + RP)\,P$

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

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

$= Q + QP + QP^2 + (Q + RP)\,P^3$

$= Q + QP + QP^2 + QP^3 + QP^4 - \cdots$

$= Q(\varepsilon + P + P^2 + P^3 + P^4 - \cdots)$

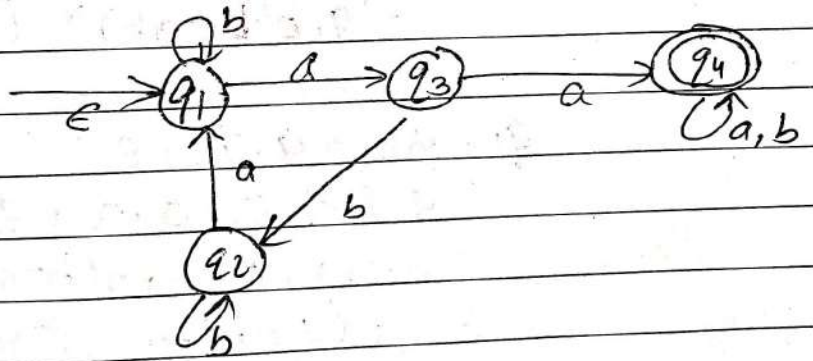
$= QP^*$

Hence proved

Conversion of Finite Automata to Recursive Expression

Method I :

Step 1 : Create equation as the following form for all the states of NFA / DFA having n states with initial state q_1.



$q_1 = bq_1 + a \cdot q_2$

$q_1 = q_1R_{11} + q_2R_{12} + \cdots + q_nR_{1n} + \varepsilon$

$q_2 = q_1R_{21} + q_2R_{22} + \cdots + q_nR_{2n}$

$\vdots$

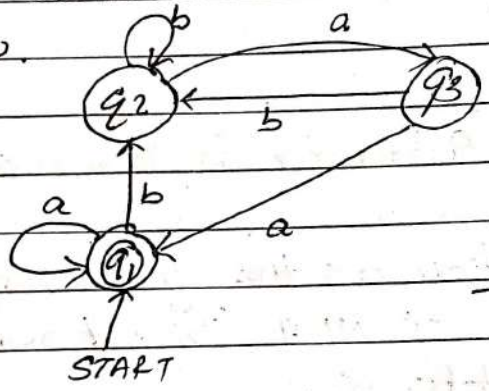
$q_n = q_1R_{n1} + q_2R_{n2} + \cdots + q_nR_{nn}$

Rij represents the set of labels of edges from q_i to q_j if no such edge exists the Rij = ϕ.

Step 2: Solve these equations to get the equations for the final state in terms of Rij.

Problem:

Construct R.E corresponding to the automata given below.



$q_1 = a \cdot q_1 + a \cdot q_3 + \epsilon$

$q_2 = q_1 b + q_2 b + q_3 \cdot b$

$q_3 = q_2 \cdot a$

Soln

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

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

$= q_1 b (b + ab)^*$ [Ardens Theorem]

$q_1 = a \cdot q_1 + a \cdot q_3 + \epsilon$

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

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

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

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

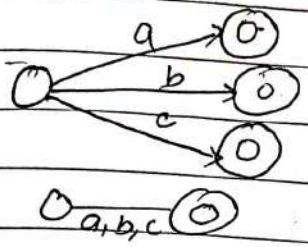
$= \epsilon (a + b(b+ab)^* aa)^*$

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

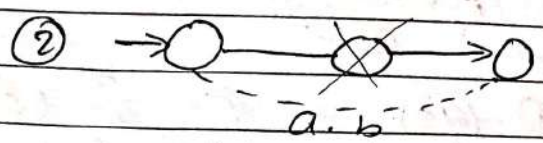
Method 2:

Step elimination Method

Rules:

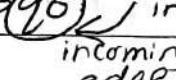
1) 

R.E = a+b+c.

2) 

a.b

R.E = a.b

3) 

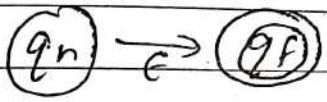
incoming edge in the start state.

incoming edge

add new start state.
a ε transition from new to old.

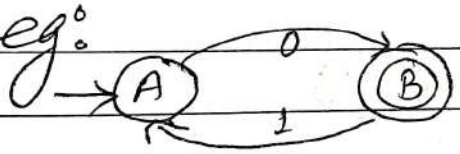
4) 

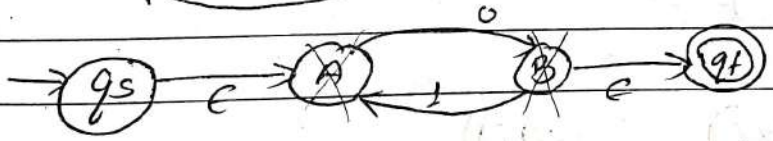
outgoing state.

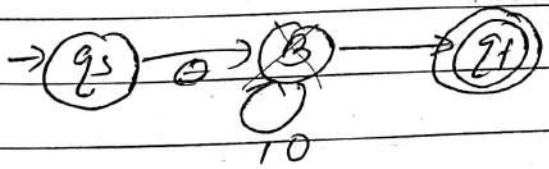


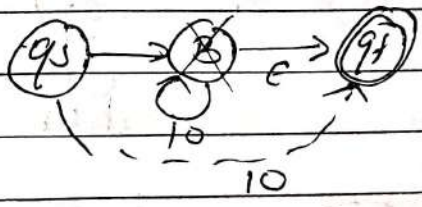
add new final state & ε transition from old to new also convert previous accepting state to non-accepting state.

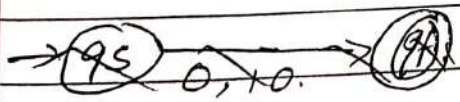
eg:





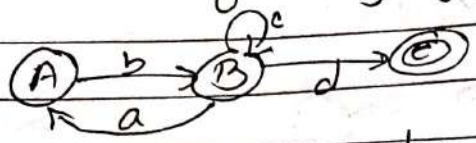


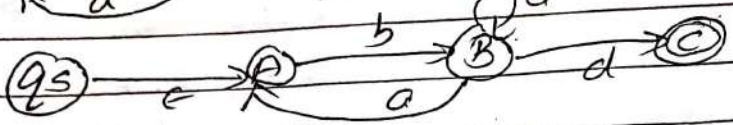


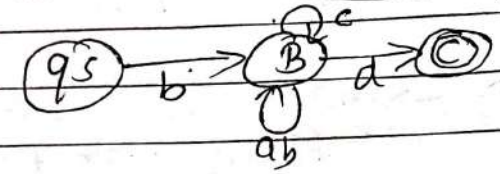


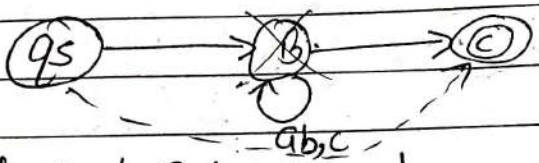
R.E = 0.(10)*

Convert the following finite automata into regular expression

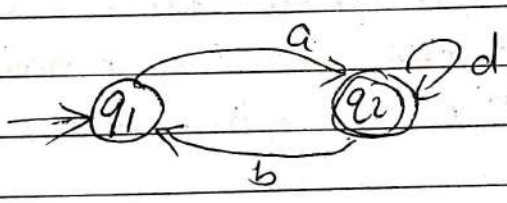


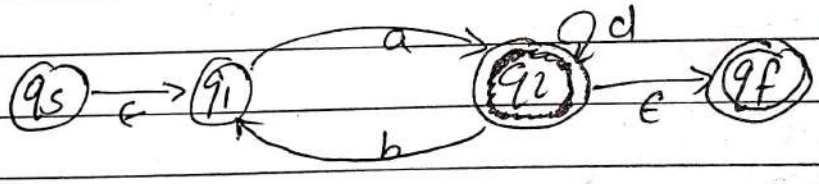
(1) 

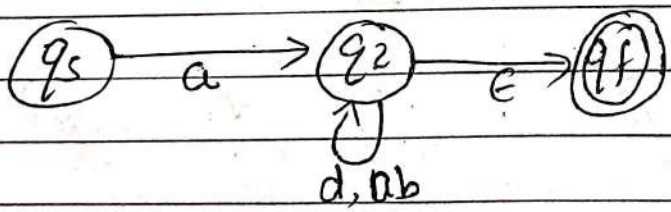
(2) 

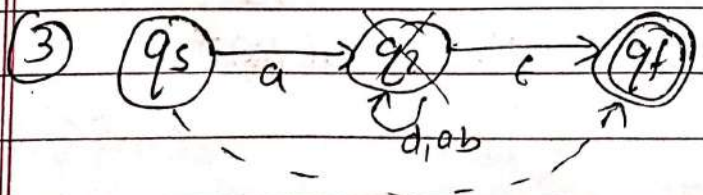
(3) 

RE ⇒ $b(ab+c)^*.d$



1) 

2) 

3) 

RE ⇒ $a(ba+a)^*$

Closure properties of Regular language.

i) Union of two regular language is regular.

Proof:

Let R and S be two regular languages than there must exist regular expression r & s such that $L(r) = R$ & $L(s) = S$.

Then clearly $r+s$ is a regular expression defining the language $L(r+s)$ which is equal to $L(r) \cup L(s)$

i.e $L(r+s) = L(r) \cup L(s)$.

$= R \cup S$

Hence proved.

ii) Complement of a regular language is also regular

Proof:

Let L be a regular language then there must exist some DFA (D) that accept the language L. If we convert all accepting state of DFA 'D' to non accepting and all the non-accepting state of DFA 'D' to accepting states then the resulting DFA, say D' must accept all the strings that are rejected by D and vice versa.

Then clearly D' is a DFA that accepts the complement of language L. Therefore $\bar{L}$ is regular.

iii) The intersection of two regular language is also regular.

Proof:

Let L and M are regular languages then the intersection of L and M represented by $L \cap M$ is given by the identity $L \cap M = \overline{\overline{L} \cup \overline{M}}$ (According to De-Morgan's)

∴ As we know that complement of regular language is regular again and union of regular language is also regular.

∴ $L \cap M$ is regular.

iv) The difference of two regular language is regular

Proof:

Let L and M are regular language than the difference of L and M denoted by L-M, according to set theory can be defined in terms of intersection and complement as $L-M = L \cap \overline{M}$. Here L is regular, $\overline{M}$ is regular and intersection of L and $\overline{M}$ is also regular. So L-M is regular.

v) Reversal of a regular language is also regular.

Proof:

Defination of Reverse of a regular language:

Let $W = a_1 a_2 \ldots a_n$ be a string in a language L. The W^R can be written as

$W^R = a_n a_{n-1} \ldots a_2 a_1$ is reversal string of language L.

The reversal of language L denoted by L^R can be

written as : $L^R = \{w^R / w \in L\}$

Proof:

Let L be a regular language than clearly there exists some finite automata (DFA, NFA, ENFA) A that accept language 'L' then we can create a finite automata A' from A that accept L^R. i.e $A' \rightarrow L^R$ such that $L^R \rightarrow L$, such that

1) Make the start state of A to only accepting state of A'.
2) Convert the direction of arcs (transition) to opposite direction.
3) Create a start state q_0 in A' along with ϵ-transition from q_0 to all the accepting state of A.
4) The result is an automatom that simulate A in reverse order and therefore accept a string w iff A accepts w^R.

Pumping lemma Theorem for regular language.

Let 'L' be a regular language than there exist a constant 'n' (which depends on L) such that for every string 'w' in L such that every string length of $|w| \geq n$ we can break w into three strings ; $w = xyz$ such that: (i) $y \neq \epsilon$

(ii) $|xy| \leq n$

(iii) $\forall k \geq 0$, the string $xy^kz \in L$.

Proof:

Let 'L' be a regular language than there must exist some DFA $D = (Q, \Sigma, \delta, q_0, F)$ of 'n' states. Now suppose, $w = a_1 a_2 \ldots a_m$ be a string in the language L where $w \geq n$. and each a_i is an input symbol.

Let $\hat{\delta}(q_0, a_1 a_2 \ldots a_i) = p_i \quad \forall \; 0 \leq i \leq n$

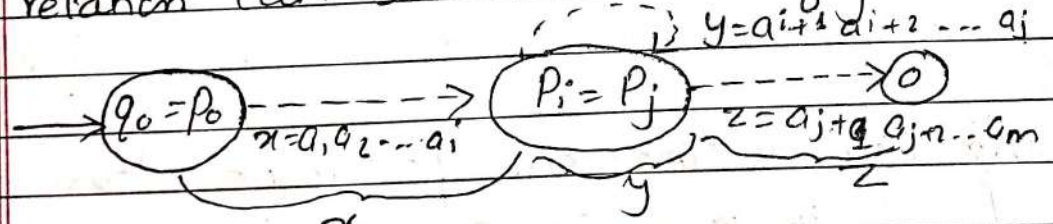
By the Pigeonhole principle, it is not possible for the 'n+1' different p_i's for $i = 0, 1, 2 \ldots n$ to be distinct, Since there are only 'n' different states.

Thus we can find two integers 'i' & 'j' such that $0 \leq i < j \leq n$ such that $p_i = p_j$. Now we can break $w = xyz$ as $x = a_1 a_2 \ldots a_i$

$y = a_{i+1}, a_{i+2} \ldots a_j$

$z = a_{j+1} a_{j+2} \ldots a_m$

i.e., x takes us to p_i once, y takes us from p_i back to p_i and z is the balance of w. This relation can be shown as figure below:



Thus:

i) $y \neq \epsilon$ because $i < j$

ii) $|xy| \leq n$ because $j \leq n$.

iii) $xy^k z \in L$ because $p_i = p_j$ for every $k \geq 0$.

Q. Show that $L = \{ w \mid w$ contains equal no. of 0's followed by equal no. of 1's $\}$ i.e $L = \{ 0^n 1^n \mid n \geq 1 \}$ is not regular.

Proof:

Let 'L' is a regular language then there must exist some 'n'. Let $w = 0^n 1^n$ be a string in language L where $|w| \geq n$.

Let us break $0^n 1^n$ into xyz & y contains more number of 0 & less number of 1.

Then according to Pumping Lemma Theorem, xy^0z must belong to language 'L' if it is regular. but xy^0z donot belong to language 'L'.

This centradicts our asumption.

∴ The language is not regular.

UNIT 4:
CONTEXT FREE GRAMMAR

Context free Grammar is a way to define a language (larger class of language than regular language) by using recursive rule also known as production. The language represented by context free grammar is known as context free language (CFL).

eg: Language of palindrome is CFL.
: $L = \{0^n 1^n \mid n \geq 1\}$ is a CFL}

Formal Defination:

A CFG, $G = (V, T, P, S)$ consists of

V = a set of variables.
T = a set of Terminal symbols
P = a set of Production rules
S = a start symbol

Example:

How a CFG represents the language of Palindrome

Soln Let $L = \{w \mid w = w^R$ be a CFL for palindrome.

By using Recursion, the language of palindrome can be defined as BASIS: ϵ, 0, 1 are palindrome.

INDUCTIVE: if x is a palindrome,

0x0 is also palindrome

& 1x1 is also palindrome

CFG for this language can be given as

V = {P}

T = {ϵ, 0, 1}

$P \to \epsilon$

$P \to 0$

$P \to 1$

$P \to 0P0$ $P \to 1P1$

Here, $V = \{P\}$

$T = \{\epsilon, 0, 1\}$

$P = \{P \to \epsilon, P \to 0, P \to 1, P \to 0P0, P \to 1P1\}$

$S = \{P\}$.

Write a CFG that represents the valid expression used in C-programming.

Note:

Let C support + and * operater only. Let identifier in C must begin with either a or b and than contains any combination of either a or b or 0 or 1. i.e

valid identifier {a, b, aa, ab, ba, ... a0, a1, b0, b1..}

Invalid identifier {0, 1, 01, 10, 1a, 1b, 1aa, 1bb ...}

Soln.

The CFG of above language:

Expression (E) → Expression + Expression.

$E \to E * E$

$E \to (E)$

$E \to I$

$I \to a$

$I \to b$

$I \to Ia$

$I \to Ib$

$I \to I0$

$I \to I1$

Check whether the following expression is in the language of grammar given below.

expression : $a * (b0 + a11)$

Grammar :

$E \to E * E \mid E + E \mid (E) \mid I$

$I \to a \mid b \mid I0 \mid Ib \mid I0 \mid I1$.

$E \to E * E$

$\to I * E \quad E \to I$

$\to a * E \quad I \to a$

$\to a * (E) \quad E \to (E)$

$\to a * (E + E) \quad E \to E + E$

$\to a * (I0 + E)$

$\to a * (b0 + E)$

$\to a * (b0 + I)$

$\to a * (b0 + I1)$

$\to a * (b0 + a11)$

$\in L(G)$

Derivation using Grammar.

We apply the production of a CFG to enter the certain string are in the language of a certain variable. There are two approaches to this inference. They are : (i) Head to Body (Derivative inference)

(ii) Body to Head (Recursive inference)

Head to Body (Derivative inference)

This is an approach for defining the language of a Grammar in which production from

head to body is used. We expand the start symbol using one of its productions (i.e using a production whose head is the start symbol.) We further expand the resulting string of the variables by the body of one of its productions and so on until we derive a string consisting entirely of terminals. The language of Grammar is all string of terminals that we can obtain in this way.

Example: Check $w = a*(b00+a1)$ is in language of following Grammar using head to body method.

Grammar: $E \to E*E \mid E+E \mid (E) \mid I$

$I \to a \mid b \mid Ia \mid Ib \mid I0 \mid I1$.

$E \to E*E$

$\to I*E$ $\quad E \to I$

$\to a*E$

$\to a*(E)$

$\to a*(E+E)$

$\to a*(I0+E)$

$\to a*$

$E \to E*E$

$\to I*E$ $\quad (E \to I)$

$\to a*E$ $\quad (I \to a)$

$\to a*(E)$ $\quad (E \to (E))$

$\to a*(E+E)$ $\quad (E \to E+E)$

$\to a*(I+E)$ $\quad (E \to I)$

$\to a*(I0+E)$ $\quad (I \to I0)$

$\to a*(I00+I)$ $\quad (I \to I0, E \to I)$

$\to a*(b00+I1)$ $\quad (I \to b, I \to I1)$

$\to a*(b00+a1)$ $\quad (I \to a)$

$\therefore \in L(G)$

Body to Head (Recursive inference)

This rule is more conventional to use. In this method we take string known to be in the language of each of the variable of the body concatinate them in the proper order, with any terminal appearing in the body and infer that the resulting string is in the language of the variable in the head.

eg: Check $w = a*(b00+a1)$ is in language of following grammar using Body to Head method.

Grammar: $E \to E*E \mid E+E \mid (E) \mid I$

$I \to a \mid b \mid Ia \mid Ib \mid I0 \mid I1$

$w = a*(b00+a1)$

$= I*(b00+a1)$ $\quad I \to a$

$= E*(I00+a1)$ $\quad E \to I$

$= E*(I0+a1)$ $\quad I \to I0$

$= E*(I+a1)$ $\quad I \to I0$

$= E*(E+a1)$ $\quad E \to I$

$= E*(E+I1)$ $\quad I \to a$

$= E*(E+I)$ $\quad I \to a,$

$= E*(E+E)$

$= E*(E)$

$= E*E$

$= E.$

Leftmost Derivation.

In this derivation at each ~~state~~ step we replace left most variable by one of its production bodies. We indicate that a derivation is leftmost by using the relation $\underset{lm}{\Rightarrow}$ or $\overset{*}{\underset{lm}{\Rightarrow}}$ for one or many steps respectively.

Example:

Use leftmost derivation to derive $w = a*(b00+a1)$ using following grammar.

$E \to E+E \mid E*E \mid (E) \mid I$.

$I \to a \mid b \mid Ia \mid Ib \mid I0 \mid I1$

$E \underset{lm}{\Rightarrow} E*E \longrightarrow E \to E*E$

$\underset{lm}{\Rightarrow} I*E \longrightarrow E \to I$

$\underset{lm}{\Rightarrow} a*E \longrightarrow I \to a$

$\underset{lm}{\Rightarrow} a*(E) \longrightarrow E \to (E)$

$\underset{lm}{\Rightarrow} a*(E+E) \longrightarrow E \to E+E$

$\underset{lm}{\Rightarrow} a*(I+E) \longrightarrow E \to I$

$\underset{lm}{\Rightarrow} a*(I0+E) \longrightarrow I \to I0$

$\underset{lm}{\Rightarrow} a*(I00+E) \longrightarrow I \to I0$

$\underset{lm}{\Rightarrow} a*(b00+E) \longrightarrow I \to b$.

$\underset{lm}{\Rightarrow} a*(b00+I) \longrightarrow E \to I$

$\underset{lm}{\Rightarrow} a*(b00+I1) \longrightarrow I \to I1$

$\underset{lm}{\Rightarrow} a*(b00+a1) \longrightarrow I \to a$

$\underset{lm}{\Rightarrow}$

Right most Derivation.

In this derivation we replace right most variable by one of its production bodies in each step. The symbol $\Rightarrow_{rm}$ or $\overset{*}{\Rightarrow}_{rm}$ are used to indicate one or many right most derivation steps respectively.

eg:

Use RMD to derive $W = a * (b00 + a1)$ using following grammar.

$E \to E * E \mid E + E \mid (E) \mid I$

$I \to a \mid b \mid Ia \mid Ib \mid I0 \mid I1$

$E \Rightarrow_{rm} E * E$ | $E \to E * E$

$E \Rightarrow_{rm} E * (E)$ | $E \to (E)$

$E \Rightarrow_{rm} E * (E + E)$ | $E \to E + E$

$E \Rightarrow_{rm} E * (E + I)$ | $E \to I$

$E \Rightarrow_{rm} E * (E + I1)$ | $I \to I1$

$E \Rightarrow_{rm} E * (E + a1)$ | $I \to a$

$E \Rightarrow_{rm} E * (I + a1)$ | $E \to I$

$E \Rightarrow_{rm} E * (I0 + a1)$ | $I \to I0$

$E \Rightarrow_{rm} E * (I00 + a1)$ | $I \to I0$

$E \Rightarrow_{rm} E * (b00 + a1)$ | $I \to b$

$E \Rightarrow_{rm} I * (b00 + a1)$ | $E \to I$

$E \Rightarrow_{rm} a * (b00 + a1)$ | $I \to a$.

Language of a CFG.

Let $G = (V, G, P, S)$ be a context free grammar than language of grammar G denoted by L(G) is the set of all strings (made up of terminal symbols) which can be derived from the starting variable of the grammar.

i.e. $L = \{ w \mid w \in T^* \text{ & } S \overset{*}{\Rightarrow} w \}$.

Ambiguity in Grammar

Parse tree:

Parse tree is simply the tree to represent that the given string can be derived from the given variable where:

i) Each interior node of the tree is a symbol from the set of variable

ii) If a internal node A contains childs like $x_1, x_2, x_3, \ldots, x_n$ than $A \to x_1, x_2, x_3, \ldots x_m$ must be a production rule in the grammar.

iii) Leaf node might contain variable or terminal symbol or Epsilon (ϵ). If child is ϵ it is the only child of its parent.

iv) Root node is the start variable.

Example:

Create a parse tree for the string a + b * a using given grammar.

$E \Rightarrow E+E \mid E*E \mid (E) \mid I$

$I \to a \mid b \mid Ia \mid Ib \mid I0 \mid I1$

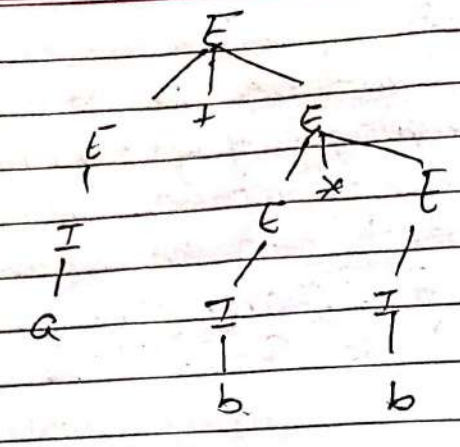


fig:- Top down parsing.

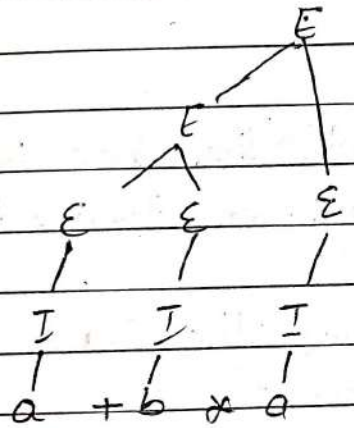
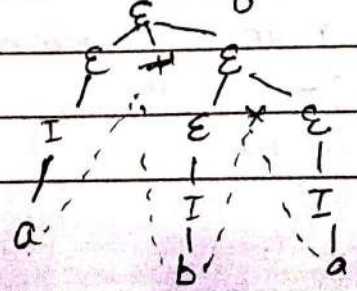


fig:- bottom up parsing.

Yield of parse tree.

If we concatinate leaves of any parse tree from left we get a string called the yield of the parse tree. Therefore yield of the parse tree is always a string that is derived from the root variable.



Ambiguity in Grammar

We assume that a grammar uniquely determine a structure for each string in its language. However not every grammar does provide unique structures which creates ambiguity.

A given grammar for the language is said to be ~~ambi~~ ambiguous if every string in the language of the grammar contains two different parse tree.

Example:

$E \rightarrow E+E \mid E*E \mid (E) \mid I$

$I \rightarrow a \mid b \mid Ia \mid Ib \mid I0 \mid I1$

This is ambiguous grammar because for the string $a+a \times a$ has two different parse tree as shown below.

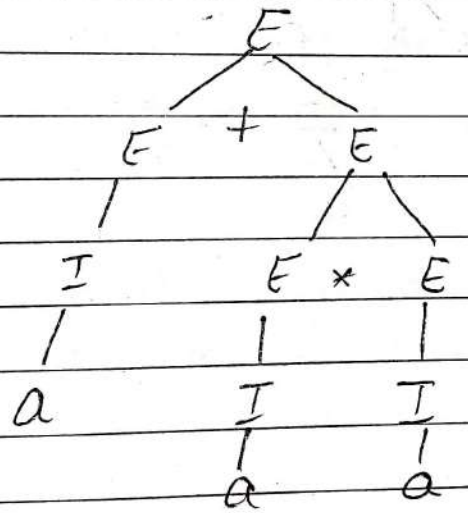


fig:- Parse tree 1

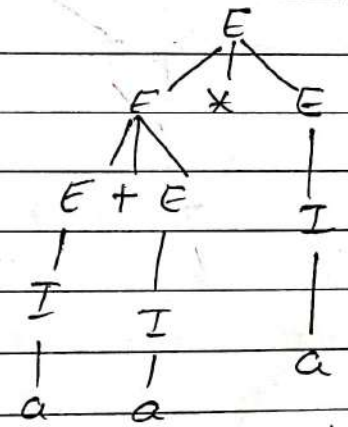


fig:- parse tree 2.

In above figure, derivation 1 says that 2nd & 3rd expressions are multiplied and result is added to the 1st expression, while derivation 2 says that 1st & 2nd expressions are added and result is multiplied to the 3rd expression.

Since this grammar is two different structures to any string of terminals so we need to modify it to use the expression grammar in a compiler. The unambigious grammar of above grammar is given below.

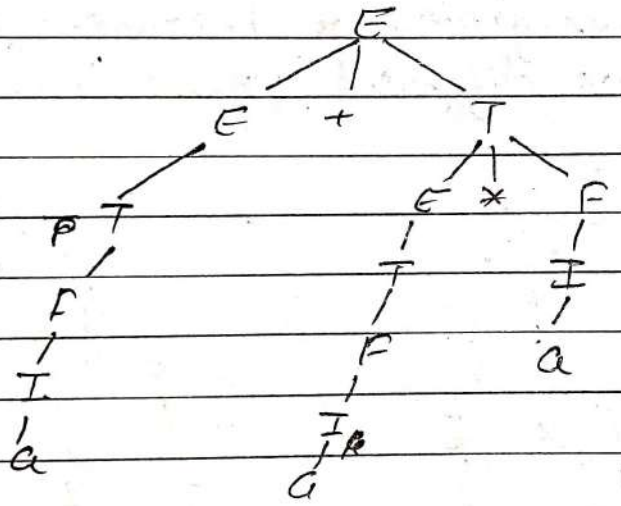
$E \rightarrow E + T \mid T$

$T \rightarrow E * F \mid F$

$F \rightarrow (E) \mid I$

$I \rightarrow a \mid b \mid Ia \mid Ib \mid I1 \mid I0$

This grammar allows only one parse tree for the string $a+a*a$ as shown below



Inherent Ambiguity.

A context free language is said to be inherent ambiguous if all its grammar are ambiguous.

Simplification of Grammar

There are three way of simplifying Grammar.

1) Eliminate useless symbols.
2) Eliminate ϵ- productions
3) Eliminate unit - production.

Eliminating useless symbols:

A symbol is said to be useless if it doesnot appear in any derivation of grammar from start symbol to the terminal symbol. or, which cannot produce terminal symbol.

eg: $S \Rightarrow \ldots\ldots \Rightarrow \alpha X \beta \Rightarrow \ldots\ldots \Rightarrow w$, then X is a useful symbol for grammar $G = (V, T, P, S)$, where $\alpha, \beta \in (V \cup T)^*$ & $X \in (V \cup T)$, $w \in T^*$

There are two different types of useless symbols

a) Non-generating useless symbol
b) Non-reachable useless symbol.

a) Non-generating useless symbol.

A symbol 'A' is said to be generating if there is a production $A \overset{*}{\Longrightarrow} w$ in the grammar where $w \in T^*$

In the grammar below;

eg: $S \rightarrow AB/b$

$A \rightarrow a$

generating symbols $\{S, A, a, b\}$

Non-generating $\{B\}$

$S \rightarrow b$

Recursive way to find out non-generating symbol.

Let $G = (V, T, P, S)$ be a grammar.

BASIS : All the terminal symbols are generating

INDUCTION: A symbol A is said to be generating if every symbol in α of derivation.

$A \to ABC / Ab / Bb / CC$ $\quad A \overset{*}{\Rightarrow} \alpha$ are generating.

eg: $S \to AB / a / ABC$

$A \to aAB / aa$

$B \to bB / b$

$C \to cD$

generating : {S, A, B, a, b}

non-generating : {C, D}

In the above grammar by recursive way {a, b, A, B, S} are generating and {D, C} are non-generating.

∴ the grammar after eliminating such non-generating symbol is given below.

$S \to AB / a$

$A \to aAB / aa$

$B \to bB / b$

Eliminating Non-reachable useless symbol.

A symbol 'S' is said to be reachable in the grammar $G = \{V, T, P, S\}$ if there is a derivation from the start symbol variable as

$S \Rightarrow \ldots\ldots \Rightarrow \alpha X \beta$

or, $S \overset{*}{\Rightarrow} \alpha X \beta$

Recursive way to find all reachable symbols.

Let $G = \{V, T, P, S\}$ be a grammar

Then:

BASIS: The symbol 'S' is reachable because $S \overset{*}{\Rightarrow} S$

INDUCTION: If there is a production or derivation of the form $A \overset{*}{\Rightarrow} \alpha$ and A is reachable then all the symbols in α are reachable.

eg: $S \to b$

$A \to aB$

$B \to BB/b$

reachable symbols $\{S, b\}$

Non-reachable symbols $\{A, a, B\}$

∴ the simplified form is: $S \to b$.

Remove useless symbol from following grammar

$S \to aAa / aBC$

$A \to aS / bD$

$B \to aBa / b$

$C \to abb / DD$

$D \to aDa$

generating: $\{S, B, C, A\}$

non-generating: $\{D\}$

The grammar after eliminating non-generating symbol is:

$S \to aAa / aBC$

$A \to aS$

$B \to aBa / b$

$C \to abb$

Reachable symbol: { S, a, A, B, C, b }
This is not simplified grammar also.

2. Eliminate ε-productions.

A production of the form A → ε is known as ε-production. During simplification of the grammar we must remove all ε-productions of course without the production that has an ε-body, it is impossible to generate empty string as a member of the language. Thus, if a language 'L' has a CFG then L = {ε} has a CFG without ε-productions.

Let G = {V, T, P, S} be a grammar then a symbol A is said to be nullable if there is a derivation in the grammar $A \xRightarrow{*} \epsilon$.

Recursive way to find all nullable symbol in the grammar.

BASIS: A symbol A is said to be nullable if there is a production A → ε in the grammar.

INDUCTION: If there is a derivation $A \xRightarrow{*} \alpha$ in the grammar and all the symbols in α are nullable, then A is also nullable.

Method to eliminate ε-production.

1) Find all the nullable symbols in the grammar
2) Let $A \to X_1 X_2 \dots X_m$ is a production in the grammar and X_i's are nullable then we replace the production

$A \to X_1 X_2 \ldots X_m$ by two or more production of all combination of absent or present of nullable symbol.

Example: Consider the grammar $S \to ABC$, $A \to BC/a$, $B \to b/BB/\epsilon$, $C \to \epsilon$.

Nullable Symbols: $\{B, C, A, S\}$

$S \to BC/AC/AB/ABC/A/B/C$.

$A \to BC/B/C/a$

$B \to b/BB/BB$

3) Eliminating Unit production

A production in the grammar is said to be unit production if it is in the form of $A \to B$, where A & B both are variables.

To remove all unit productions from the grammar, 1st we have to find all the unit pairs of the grammar.

Recursive technique to find all unit pairs in the grammar.

Let $G = \{V, T, P, S\}$ be a given grammar then (A, B) unit pair can be derived from the grammar as:

BASIS: Let A be the variable then (A, A) is a unit pair.

INDUCTION: Let (A, B) be a unit pair of variables and $B \to C$ be a production in the grammar then (A, C) is also unit pair.

Method to eliminate all unit productions.

Let $G = \{V, T, P, S\}$ be a given grammar. Then, we can construct equivalent grammar, $G' = \{V, T, P', S\}$ without any unit production such that language of grammar $L(G) = L(G')$.

i) First find all the unit pairs in the grammar.

ii) Let (A, B) is a unit pair in the grammar G, then add all productions $A \to \alpha$ in the grammar G' where $B \to \alpha$ are non-unit production in G. [Here $\alpha \in (V \cup T)^*$].

eg: Eliminate unit production from given grammar

$S \to SS / A / a / ad / AB$

$A \to B / d / CC$

$B \to C / e / DD$

$C \to SC / AB / ab$

$D \to AB$

BASIS: $(S, S), (A, A), (B, B), (C, C), (D, D)$.

INDUCTION:

(S, A) / $\because S \to A$

(A, B) / $\because A \to B$

(S, B) / $\because S \to A, A \to B$

(B, C) / $\because B \to C$.

(A, C) / $\because A \to B, B \to C$.

(S, C) / $\because S \to A, A \to B, B \to C$.

Now the equivalent grammar after removing the unit production is

$S \to SS / d / CC / a / ad / AB / e / DD / SC / ab$

A → e/DD/d/cc/SC/AB/ab
B → SC/AB/ab/e/DD
C → SC/AB/ab
D → AB

Remove all unit production from given grammar.
E → T/E+T
T → F/T+F
F → I/(E)
I → a/b/Ia/Ib/I0/I1
BASIS: (E,E), (T,T), (F,F), (I,I)
INDUCTION:
~~(E,T): T+F/E+T~~ (E,T) /∵ E → T
~~(T,F): (E)/T+F~~ (E,F) /∵ T → F
~~(E,F): (E)/E+T~~ (E,F) /∵ E → T, T → F
~~(F,I): a/b/Ia/Ib/I0/I1/(E)~~ (F,I)
~~(E,I):~~ (E,I)
(T,I)

After removing the unit pair the grammar is:
E → T+F/(E)/a/b/Ia/Ib/I0/I1/E+T
T → (E)/a/b/Ia/Ib/I0/I1/T+F
F → a/b/Ia/Ib/I0/I1/(E)
I → a/b/Ia/Ib/I0/I1

Chomsky Normal Form.

A grammar $G=(V,T,P,S)$ is said to be Chomsky Normal Form (CNF) if all of its productions are either of the form (i) $A \to a$ (ii) $A \to BC$ where $A, B, C \in V$ & $a \in T$. Further more G has no useless symbols.

Converting the given grammar into CNF:

Let $G=(V,T,P,S)$ be an arbitrary grammar representing the language $L(G)$. To convert G into CNF we follow the following steps:

1) Simplify the grammar in the order
   - Eliminate ϵ- productions.
   - Eliminate unit production
   - Eliminate useless symbols.
     - ↳ eliminate non-generating symbols
     - ↳ eliminate non-reachable symbols

2) Then the resulting grammar contain the production in the form either $A \to a$ or the body of the production contains two or more symbols from $(V \cup T)^*$.

$A \to a$

$A \to \alpha$

Here all the productions having body length greater or two or more are replaced by corresponding productions with variables only in body of production

3) Now all the productions in the grammar having length two or more variables only. If we replace these productions

by cascading productions, we get the resulting grammar in CNF.

Convert the following grammar into CNF.

$E \to I \mid E+T$

$T \to F \mid T*F$

$F \to I \mid (E)$

$I \to a \mid b \mid Ia \mid Ib \mid I0 \mid I1$

1) Simplifying the grammar

a) No ε-production.

b) Grammar after removing unit production

$E \to E+T \mid T*F \mid (E) \mid a \mid b \mid Ia \mid Ib \mid I0 \mid I1$

$T \to T*F \mid (E) \mid a \mid b \mid Ia \mid Ib \mid I0 \mid I1$

$F \to (E) \mid a \mid b \mid Ia \mid Ib \mid I0 \mid I1$

$I \to a \mid b \mid Ia \mid Ib \mid I0 \mid I1$

Since there are 8 terminals each of which appear in the body that is not single terminal. Thus we must introduce 8 new variables corresponding to these terminals and 8 productions in which the new variables is replaced by its terminal.

Let: $A \to a$, $B \to b$, $Z \to 0$, $O \to 1$

$P \to +$, $M \to *$, $L \to ($, $R \to)$

Now replacing each terminal in the body, we get:

$E \to EPT \mid TMF \mid LER \mid a \mid b \mid IA \mid IB \mid IZ \mid IO$

$T \to TMF \mid LER \mid a \mid b \mid IA \mid IB \mid IZ \mid IO$

$F \to LER \mid a \mid b \mid IA \mid IB \mid IZ \mid IO$

$I \to a \mid b \mid IA \mid IB \mid IZ \mid IO$

$A \to a$, $B \to b$, $Z \to 0$, $O \to 1$, $P \to +$, $M \to *$, $L \to ($, $R \to)$.

3) Replacing the body with more than two length by a new variable.

Let: $D \to PT$, $J \to MF$, $K \to ER$.

Then we get,

$E \to ED \mid TJ \mid LK \mid a \mid b \mid IA \mid IB \mid IZ \mid IO$

$T \to TJ \mid LK \mid a \mid b \mid IA \mid IB \mid IZ \mid IO$

$F \to LK \mid a \mid b \mid IA \mid IB \mid IZ \mid IO$

$I \to a \mid b \mid IA \mid IB \mid IZ \mid IO$

$D \to PT$

$J \to MF$

$K \to ER$

$A \to a$

$B \to b$

$Z \to 0$

$O \to 1$

$P \to +$

$M \to *$

$L \to ($

$R \to)$

∴ The grammar is in CNF.

Left Recursion.

A grammar is left recursive if it has non terminal 'A' such that there is a derivation $A \Rightarrow A\alpha$ for some string α. i.e $\alpha \in (V \cup T)^*$

Top down parsing method cannot handle left recursion so a transformation is needed to eliminate left recursion. In general, left recursive pair of

model $A \to A\alpha / \beta$ can be replaced by the non-left recursive production as:

$A \to \beta A'$

$A' \to \alpha A' / \varepsilon$

$A \to A\alpha / \beta_1 / \beta_2$

$A \to \beta A' / \beta_2 A'$

$A' \to \alpha A' / \varepsilon$

Q. Eliminate left recursion from given grammar.

$E \to E+T / T$

$T \to T * F / F$

$F \to (E) / id$

⇒ The non-left recursive grammar of above grammar is:

$E \to TE'$

$E' \to +TE' / \varepsilon$

$T \to FT'$

$T' \to *FT' / \varepsilon$

$F \to (E) / id.$

$A_4 \to b / bA_3A_4 / A_4A_4A_4$

$\Rightarrow A_4 \to \underbrace{A_4A_4A_4}_{A\alpha} / \underbrace{b}_{\beta_1} / \underbrace{bA_3A_4}_{\beta_2}$

The non-left recursive grammar of above grammar is:

$A_4 \to bZ / bA_3A_4Z$

$Z \to A_4A_4Z / \varepsilon$

Simplify the following grammar. $A_4 \to bZ / bA_3A_4Z$

$Z \to A_4A_4Z / \varepsilon$

$\Rightarrow A_4 \to b / bZ / bA_3A_4 / bA_3A_4Z$

$Z \to A_4A_4 / A_4A_4Z$

Greibach Normal form (GNF)

A CFG is GNF if the productions are in the form $A \to b$

$A \to bC_1 C_2 C_3 \dots C_n$

where $A, C_1, C_2, \dots, C_n$ are non-terminals & b is a terminal

Steps for converting a given CFG into GNF.

1) Check if the Given Grammar is in CNF or not if not then convert it into CNF.
2) Change the names of the Non-terminal symbol into some A_i in the ascending order of 'i'.
3) Alter the rules so that the non terminals are in ascending order such that if the production is of the form $A_i \to A_j x$, then $i < j$ & should never be $i \geq j$.
4) Eliminate left recursion.
5) If ϵ-production appears after eliminating left recursion, eliminate ϵ-productions.

Example: Convert the given grammar into GNF.

$S \to CA / BB$

$B \to b / SB$

$C \to b$

$A \to a$

Step 1: Given Grammar is in CNF.

Step 2: Replace S with A_1

C with A_2

A with A_3

B with A_4

then the Grammar Appears as:

$A_1 \rightarrow A_2A_3 \mid A_4A_4$

$A_4 \rightarrow b \mid A_1A_4$

$A_2 \rightarrow b$

$A_3 \rightarrow a$

Step3: In the production $A_4 \rightarrow A_1A_4$ while comparing with $A_i \rightarrow A_jx$ this $i > j$

∴ we need to alter their rule.

∴ $A_4 \rightarrow b \mid A_2A_3A_4 \mid A_4A_4A_4$ ∵ $A \rightarrow A_2A_3 \mid A_4A_4$.

$A_4 \rightarrow b \mid bA_3A_4 \mid A_4A_4A_4 \mid A_2 \rightarrow b$

After altering the rules the grammar is

$A_1 \rightarrow A_2A_3 \mid A_4A_4$

$A_4 \rightarrow b \mid bA_3A_4 \mid A_4A_4A_4$

$A_2 \rightarrow b$

$A_3 \rightarrow a$

Step4: Eliminate left recursion.

The production $A_4 \rightarrow A_4A_4A_4$ is left recursive.

$\underset{A}{A_4} \rightarrow \underset{A}{A_4}\underset{\alpha}{A_4A_4} \mid \underset{\beta_1}{b} \mid \underset{\beta_2}{bA_3A_4}$ $[A \rightarrow A\alpha \mid \beta]$

The non left recursive production equivalent to above production

$A_4 \rightarrow bZ \mid bA_3A_4Z$

$Z \rightarrow A_4A_4Z \mid \epsilon$

Now Simplifying Above grammar.

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

$Z \rightarrow A_4A_4 \mid A_4A_4Z$

Now the Grammar is

$A_1 \rightarrow A_2A_3 / A_4A_4$

$A_4 \rightarrow bbZ / bA_3A_4 / bA_3A_4Z$

$Z \rightarrow A_4A_4 / A_4A_4Z$

$A_2 \rightarrow b$

$A_3 \rightarrow a$

Now again manipulating the symbols of the Grammar

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

$A_4 \rightarrow b / bz / bA_3A_4 / bA_3A_4Z$

$Z \rightarrow bA_4 / bZA_4 / bA_3A_4A_4 / bA_3A_4ZA_4 / bA_4Z / bZA_4$
$/ bA_3A_4A_4Z / bA_3A_4ZA_4Z$

$A_2 \rightarrow b$

$A_3 \rightarrow a$

Constructing Context free Grammar

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

$L = \{w / |w| = 2\}$

$S \rightarrow aa / ab / ba / bb$

R.E $= (a+b)(a+b)$

$S \rightarrow AA$

$A \rightarrow a / b$

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

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

$a^* \Rightarrow S \to aS \mid \epsilon$

Q. $(a+b)^*$

$S \to bS$
$S \to aS$
$S \to \epsilon$

$\Rightarrow S \to aS \mid bS \mid \epsilon$.

Q. $L = \{w \mid \cdot \mid w \mid \geq 2\}$.

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

$A \to a \mid b$

$B \to ab \mid bB \mid \epsilon$

$S \to AAB$

Q. $L = \{w \mid |w| \geq 2\}$

R.E = $(a+b+\epsilon)(a+b+\epsilon)$

$L = \{\epsilon, a, b, aa, ab, ba, bb\}$

$S \to AA$

$A \to a \mid b \mid \epsilon$.

Q. $L = \{awb \mid w \in (a+b)^*\}$

$L = \{ab, aab, abb, aabbabb\}$

R.E. $a(a+b)^* b$ $\quad S \to aAb \quad A \to aA \mid bA \mid \epsilon$

Q. R.E = $a(a+b)^* b + b(a+b)^* a$.

$S \to aAb \mid bAa$

$A \to aA \mid bA \mid \epsilon$.

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

$A \to aA / bA / \varepsilon$

$S \to aAa / bAb$

Construct CFG for following language.

$L = \{a^n b^n / n \geq 1\}$ [Not regular]

$\Rightarrow S \to asb / ab$

Q. $L = \{ww^R \cup waw^R \cup wbw^R\}$

$S \to asa / a / b / \in / bSb$

Q) $L = \{a^n b^n c^m / m, n \geq 1\}$

$S \to AC.$

$A \to aAb / ab$

$C \to cC / c.$

$L = a^n b^n c^n /$ is not CFL
No CFG exist
Since $a^n b^n c^n$ is CSL.

Q) $L = \{a^n c^m b^n / n, m \geq 1\}$

$S \to aSb / aCb$

$C \to cC / c.$

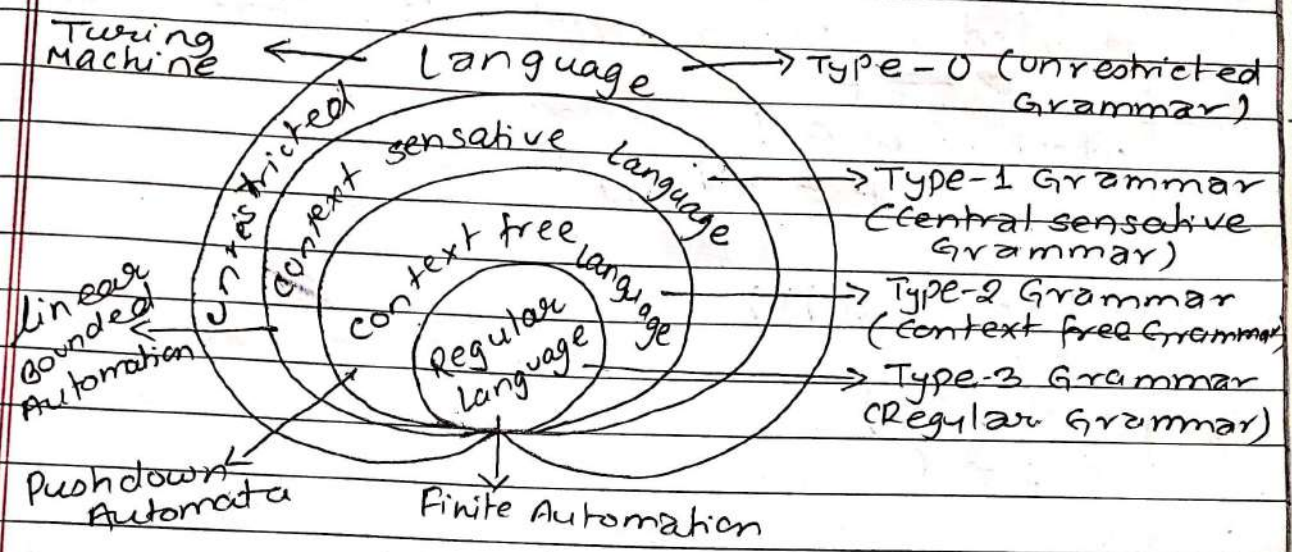
Q. $L = \{a^n b^n c^m d^m / m, n \geq 1\}$

$S \to AB$

$A \to aAb / ab$

$B \to cBd / cd.$

Chomsky's Hierarchy



Type-3 - Grammar (Regular Grammar)

This grammar generates regular language which is accepted by Finite Automata and has the form $A \to a / Ba$ or $A \to a / aB$.

left linear right linear

where $A, B \in V$ & $|A| - |B| = 1$ $a \in T$.

RE = a * b c*, RG = ?

$S \to aS$

$S \to bA$

$A \to cA$

$A \to \varepsilon$.

Type 2 Grammar (Context free Grammar)

This Grammar is used to generate CFL & accepted by pushdown automata. & has form.

$\alpha \rightarrow \beta$

where, $\alpha \in V$ & $|\alpha| = 1$

$\beta \in (VUT)^*$

- $L = \{((((\,)\,))\,)\}$

$S \rightarrow SS$

$S \rightarrow (S)$

$S \rightarrow ()$

Type 1 Grammar (Context Sensative Grammar / length increasing Grammar / Non-Contracting Grammar / Context Sensative ~~Enumerable~~ ~~Recursively~~ language (CSL) (~~REL~~) which is accepted by linear bounded Automata. (LBA)

It has form

$\alpha \rightarrow \beta$

where $\alpha \in (TUV)^* V (TUV)^*$

$\beta \in (VUT)^+$

& $|\alpha| \leq |\beta|$

$L = \{a^n b^n c^n \mid n >= 1\} \Rightarrow$ $S \rightarrow aSBC \mid aBC$

$bB \rightarrow bb$

$bC \rightarrow bc$

$cC \rightarrow cc$

$CB \rightarrow HB$

$HB \rightarrow HC$

$HC \rightarrow BC$

$bB \rightarrow bb$

$bC \rightarrow bC$

$cC \rightarrow cc$

Pumping Lemma for CFL: (without proof)

Let L be CFL. Then there exist a constant n st for all string $w \in L$ where $|w| \geq n$. We can break w as $w = uvzxy$ st.

i) $|vzx| \leq n$

ii) $vx \neq \epsilon$ or $|vx| \neq 0$

iii) $uv^k zx^k y \in L$.

Steps to prove/show given language is not context free language

Given: $L = \{$

1) Assume L is CFL & there exist n where $|w| \geq n$.

$w = uvzxy$

$|vx| \geq 1$.

2) Try to show that $w \notin L$, which contradicts our assumption. $\therefore$ L is not CFL.

Q. Show that $L = \{a^n b^n c^n / n \geq 1$ is not CFL.

Soln

1) Assume that L is CFL.

It means L must have pumping length P st.

$w = a^P b^P c^P$

2) we divide w into parts $uvzxy \in L$.

Let $P = 4$

Then $w = a^4 b^4 c^4$

Case I: v & x each contains only one type of symbol.
· Let v only contains a & x only contains b

let $w = \underbrace{a}_{u}\underbrace{aaa}_{v}\ \underbrace{bb}_{z}\ \underbrace{bb}_{x}\ \underbrace{cccc}_{y}$

Let $k = 2$.

$uv^kzx^ky = a\,aaaaaa\,bb\,bbbb\,cccc \notin L$.

We are done with this case.

Case II: Either v or x contains different symbols.

Here, $w = aaaa\ bbbb\ cccc$

let v contains some a & some b & x contains only b as:

$w = \underbrace{aa}_{u}\ \underbrace{aa\ bb}_{v}\ \underbrace{bb}_{z}\ \underbrace{bb}_{x}\ \underbrace{cccc}_{y}$

let $k = 2$ then;

$uv^kzx^ky = aa\ aabbaa\ bbbb\ bb\ aaaa\ bb\ aa\ bbbb\ bb\ cccc$

$\therefore L$ is not CFL.

UNIT: 5

PUSH DOWN AUTOMATA (PDA)

Push Down Automata (PDA):

The context free language have a type of automaton that defines them. This automaton called a Push Down Automata (PDA) is an extension of the non-determinastic finite automata (NFA) with ϵ-transition. The PDA is essentially an ϵ-NFA with the addition of the stack. The stack can read, pushed and poped, only at the top just like the stack data structure. The presence of the stack means that unlike the finite automaton, the push down automata (PDA) can remember an infinite amount of information. However, unlike of general purpose computer which also has the ability to remember arbitrary large amount of information, the PDA can only access the information on its stack in a last-in-first-out way.

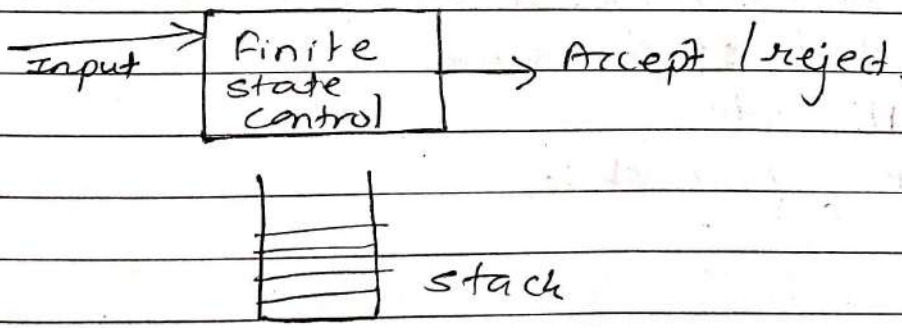
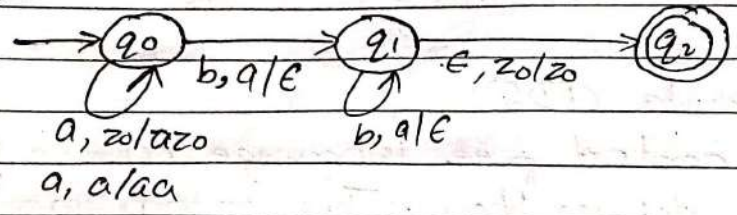


fig: Block Diagram of PDA

eg: Design a PDA for $L=\{a^n b^n / n \geq 1\}$



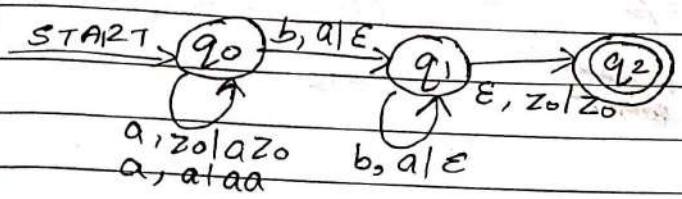
Formal Defination.

A Push down Automata PDA,
$P=\{Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$ consists of
Q = Finite set of states.
Σ = a finite set of input symbols
Γ = a finite set of stack symbol.
δ = a transaction function of the form
$\delta(q, a, \alpha) = (p, \beta)$

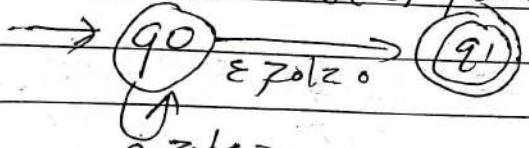
Where, q, p are states.
a is input symbol from Σ,
α is top of the stack
p is the action on the top of the stack.
z_0 is initial stack symbol
q_0 is start state.
F is set of final state.

Constructing Push Down Automata.

Q. $L = \{a^n b^n \mid n \geq 1\}$

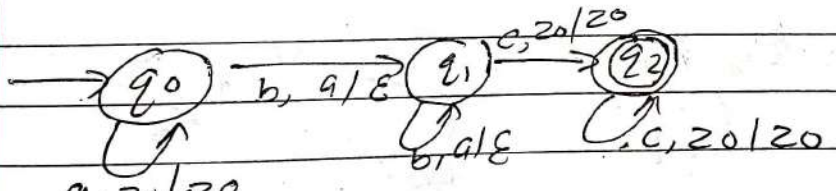


Q. $L = \{n_a(\omega) = n_b(\omega) \mid \omega \in (a,b)^*\}$

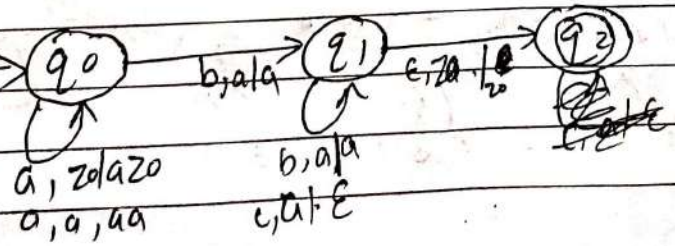


a, z0/az0
b, z0/bz0
a, a/aa
b, b/bb
a, b/ε
b, a/ε

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

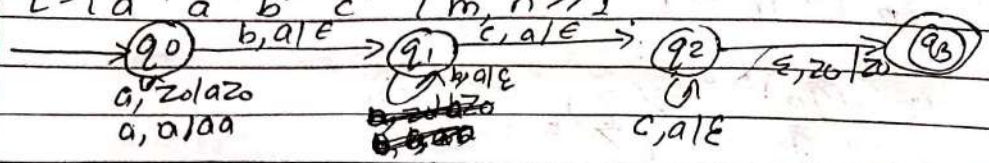


Q. $L = \{a^n b^m c^n \mid m, n \geq 1\}$



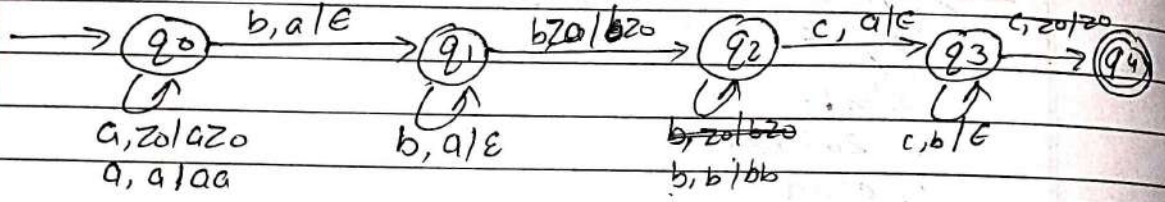
$L = \{ a^{m+n} b^m c^n \mid m, n \geq 1 \}$

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



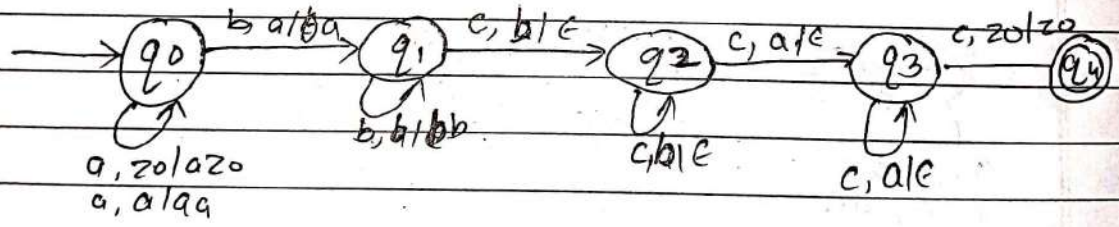
$L = \{ a^n b^{m+n} c^m \mid m, n \geq 1 \}$

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

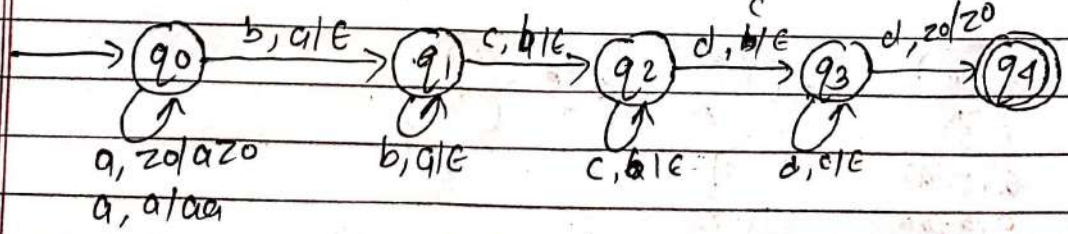


Construct PDA for following languages.

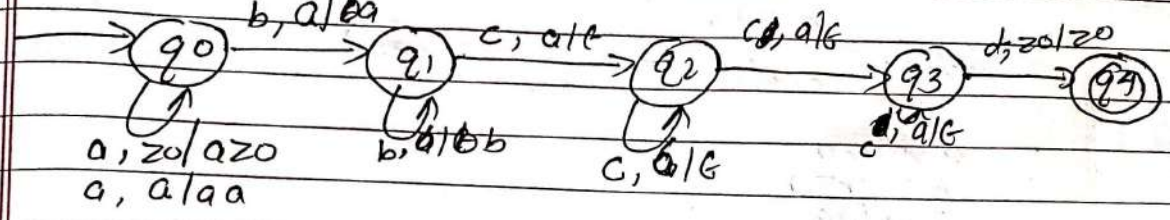
1) $L = \{ a^n b^m c^{m+n} \mid m, n >= 1 \}$



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

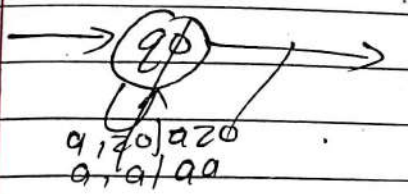


3] $L = \{a^n b^m c^m d^n \mid m, n \geq 1\}$.

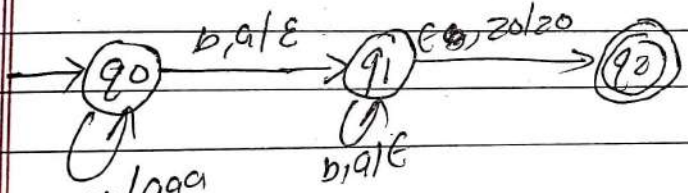


4] $L = \{a^n b^m c^n d^m \mid m, n \geq 1\}$

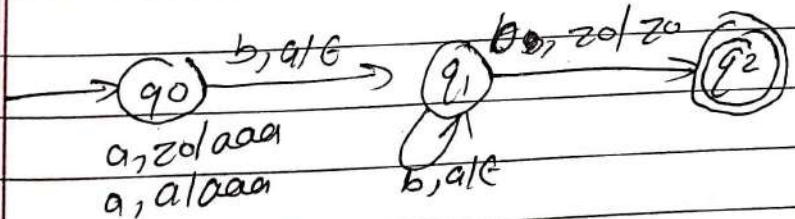
It is not context free language so PDA cannot be constructed.



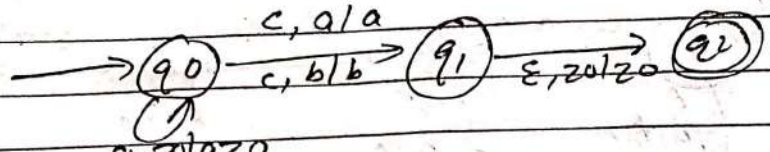
5] $L = \{a^n b^{2n} \mid n \geq 1\}$.



6] $L = \{a^n b^{2n+1} \mid n \geq 1\}$.



12 $L = \{ W c W^R / W \in (a,b)^+ \}$



a, z0/az0
b, z0/bz0
a, a/aa
b, b/bb
a, b/ab
b, a/ba

$L = \{ W W^R / W \in (a,b)^+ \}$

The language of PDA:

There are two approach by which PDA accepts any input string.

→ Acceptance by final state.

→ Acceptance by Empty state.

1) Acceptance by final state.

Let $P = (Q, \Sigma, \Gamma, \delta, q_0, F, Z_0)$ be a PDA then, $L(P)$ the language accepted by P by final state is: $L(P) = \{ w / (q_0, w, Z_0) \vdash^* (q, \epsilon, \alpha) \}$

where, $q \in F$

& α is any stack string.

2) Acceptance by Empty stack.

Let $P = (Q, \Sigma, \Gamma, \delta, Z_0, F)$ be a PDA then, $N(P)$, the language accepted by P by empty stack is:

$N(P) = \{ w / (q_0, w, Z_0) \vdash^* (q, \epsilon, \epsilon) \}$

where ~~$q \in Q$~~ $q \in Q$.

From empty stack to Final state.

From final state to empty stack.

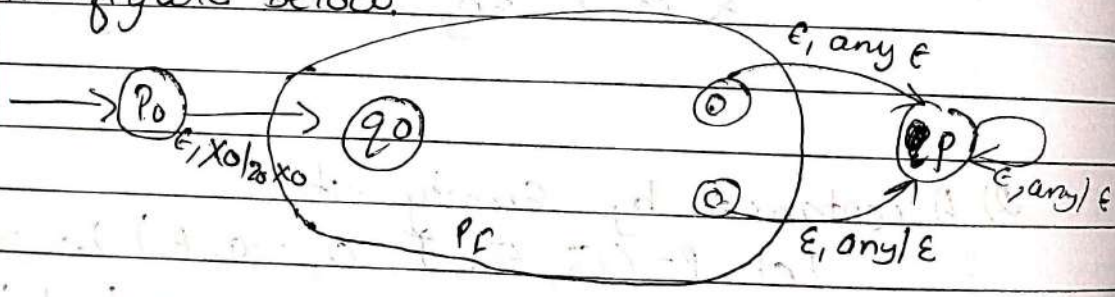
1) From Final state to empty stack.

Theorem:

Let L be $L(P_F)$ for some PDA $P(F) = (Q, \Sigma, I, \delta_F, q_0, Z_0, F)$ by a Final state then there is a PDA P_N by empty stack such that $L = N(P_N)$. i.e $L(P_F) = N(P_N)$.

Proof:

Take a PDA P_F that accepts a language L by final state and construct another PDA P_N that accepts L by empty stack as shown in figure below.



For each accepting state of P_F, at a transition on ε to a new state P. When in P, P_N pops its stack and doesnot consume any input. Thus whenever P_F enters an accepting state after consuming input 'w' completely, after that P_N will empty its stack after consuming 'w'

To avoid the simulating a situation where, P_F accidentally emptys its stack without

accepting, P_N must allow use of marker X_0 on the bottom of this stack. This marker is P_N's start symbol and P_N must start in a new state P_0, whose function is to push the start symbol of P_F on the stack and go to the state q_0.

Now the new specification is:

$P_N = (Q \cup \{P_0, P\}, \Sigma, T \cup \{X_0\}, \delta_N, P_0, X_0)$

Where, δ_N is defined as:

1) $\delta_N(P_0, \epsilon, X_0) = \{(q_0, Z_0 X_0)\}$

2) $\forall q \in Q$, & $a \in \Sigma$ or $a = \epsilon$, & $Y \in T$

$\delta_N(q, a, Y) = \delta_F(q, a, Y)$

i.e. P_N simulates P_F.

3) $\forall q \in F$ & $Y \in T$ or $Y = X_0$

$\delta(q, \epsilon, Y) = (P, \epsilon)$

4) $\delta(P, \epsilon, Y) = (P, \epsilon)$

Now to prove w is in $N(P_N)$:

If part: Suppose $(q_0, w, Z_0) \vdash^* (q, \epsilon, \alpha)$ for some $q \in F$ & stack string α.

Using the fact that every transition of P_F is a move of P_N, we can write,

$(P_0, w, X_0) \vdash (q_0, w, Z_0 X_0) \vdash^* (q, \epsilon, \alpha X_0)$

$P_F \vdash (P, \epsilon, \epsilon)$

P_N

The first move is by rule no 1. of the construction of P_N while the last sequence of move is by rule no 3 & rule no 4. Thus w is accepted by P_N by empty stack.

Only if:

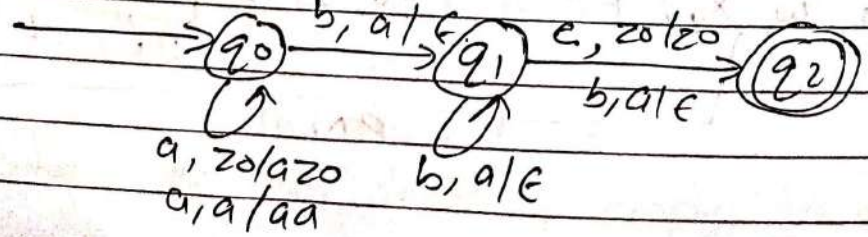
The only way P_N can empty its stack is by entering to state P_f since X_0 is sitting at the bottom of the stack and X_0 is not any symbol on which P_F has any moves. The only way P_N can enter state P is if the simulated P_F enters an accepting state

∴ The every accepting computation of P_N looks like $(P_0, w, X_0) \vdash (q_0, w, Z_0, X_0) \vdash^* (q, \epsilon, \gamma, X_0) \vdash (P, \epsilon, \epsilon)$

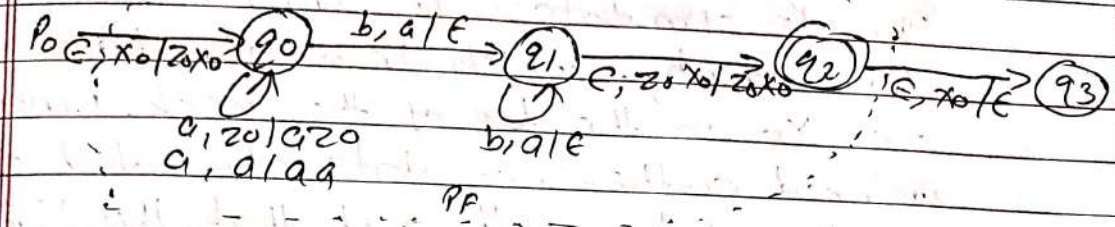
where $q \in F$ & of P_F.

$\therefore L(P_N) = L(P_D)$

Q. Construct a PDA by empty stack from given PDA.



Soln



2) From Empty Stack to Final state.

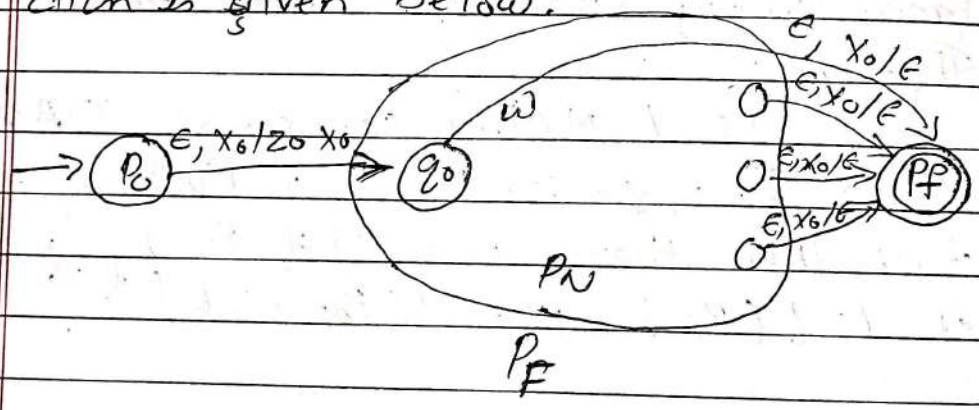
Theorem:

If $L = N(P_N)$ for some PDA $P_N = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ is a PDA by empty stack then there is a PDA P_F by final state st $L = L(P_F)$ i.e $L(P_N) = L(P_F)$.

Proof:

We use a new symbol X_0 which must not be a symbol of Γ such that X_0 is both the start symbol of P_F and a marker on the bottom of the stack that lets us to know when P_N has ~~least~~ reached an empty stack that is if P_F ~~is~~ sees X_0 on top of its stack then it knows that P_N would be empty its stack on the same input. We also need a new stack state P_0 whose ~~sole~~ sole function is to push Z_0, the start symbol of P_N, ~~from~~ on the top of stack and

enter to state q_0, the start state of P_N. The P_F simulates P_N until the stack of P_N is empty, which P_F detects because it sees X_0 on the top of the stack. Finally we need another new state, P_f which is the accepting state of P_F such that this PDA transfers to state P_f whenever it discovers that P_N could have emptied its stack. The construction is given below.



Now, the new specification of P_F is

$P_F = (Q \cup \{q_0, P_0\}, \Sigma, T \cup \{X_0\}, \delta_F, P_0, X_0, P_f)$

where, δ is defined as:

1) $\delta(P_0, \varepsilon, X_0) = (q_0, Z_0 X_0)$
2) $\forall q \in Q$ & input $a \in \Sigma$ or $a = \varepsilon$ & stack symbol $Y \in T$.
   $\delta_N(q, a, \alpha) = \delta_N(q, a, Y)$
3) $\delta_F(q, \varepsilon, X_0) = (P_f, \varepsilon)$ $\forall q \in Q$

If part:

We are given that $(q_0, w, z_0) \vdash_{P_N}^{*} (q, \epsilon, \epsilon)$. For some state q.

Let us insert X_0 at the bottom of the stack and conclude $(q_0, w, z_0X_0) \vdash^{*} (q, \epsilon, X_0)$, Since by rule 2 P_F has all the moves of P_N the above configuration exist for P_F also. i.e $(q_0, w, z_0, X_0) \vdash_{P_F}^{*} (q, \epsilon, X_0)$. Now if we put the sequence of moves together with the initial & final moves from rule no 1 & 3 we get,

$(P_0, w, X_0) \vdash_{P_F} (q_0, w, z_0X_0) \vdash_{P_F}^{*} (q, \epsilon, X_0) \vdash_{P_F} (P_F, \epsilon, \epsilon)$

Thus, P_F accepts w by final state.

Only If part:

The converse requires only that we observe the additional transitions of rule 1 & 3 gives us very limited ways to accept w by final state. We must use rule 3 at the last step and we can only use that rule if the stack of P_F contains only X_0. No X_0 appears on the stack except at the bottom most position.

Thus any computation of P_F that accept w must look like the sequence

$(P_0, w, X_0) \vdash_{P_F} (q_0, w, z_0X_0) \vdash_{P_F}^{*} (q, \epsilon, X_0) \vdash_{P_F} (P_F, \epsilon, \epsilon)$

At the middle of the completation, P_N must be with X_0 at the last before entering to the P_F and X_0 cannot be exposed or the completation would end at the next state. Hence we conclude that $(q_0, w, z_0) \vdash^*_{P_N} (q, \epsilon, \epsilon)$ i.e $w \in N(P_N)$

$\therefore L(P_F) = N(P_N)$

$\therefore N(P_N) = L(P_{EP})$

Construct a PDA by final state from the given PDA.

0, Z0/AZ0
1, Z0/BZ0
0, A/AA
1, B/BB
0, B/ϵ
1, A/ϵ
ϵ, Z0/ϵ

Sol:

$L = \{n(a) = n(b)$

0, Z0/AZ0
1, Z0/BZ0
0, A/AA
1, B/BB
0, B/ϵ
1, A/ϵ
ϵ, Z0/ϵ.

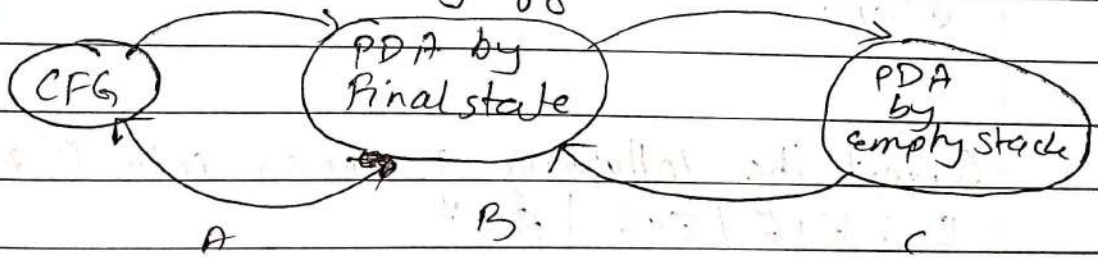
Equivalence of PDA and CFG

We demonstrate that the language defined by PDA's are exactly CFL.

The goal is to prove that the following ~~by CFGs~~ classes of language.

1) CFL: the language defined by CFGs.
2) The language that are accepted by PDA by final state
3) The language that are accepted by PDA by empty stack are the same

We have already shown that 2 & 3 are same. Now we show that 1 & 3 are same which turns out 1 & 2 are also same. This can be shown by fig below.



From Grammar to PDA:

Let $G = (V, D, P, S)$ be a CFG to construct the PDA P that accept language of ~~from~~ Grammar ~~G~~ L(a) by empty stack; we define the PDA as follows.

$P = \{(Q), \Sigma, \Gamma, \delta, q_0, z_0\}$ where δ is

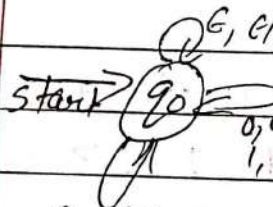
defined as:

1) For each variable A
$\delta(q, \epsilon, A) = \{q, \beta \mid A \to \beta$ is a production in the Grammar

2) $\forall a \in T \quad \delta(q, a, a) = \{q, \epsilon\}$

Convert the following grammar into corresponding PDA.

$S \to SOS1 \mid SOS \mid SOSOS1S \mid S1SOSOS \mid \epsilon$

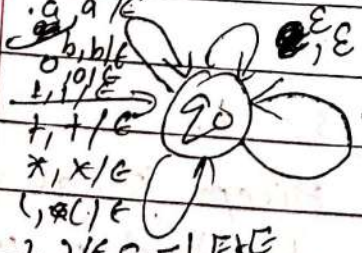


ε, S/SOS1SOS
ε, S/S1SOSOS
ε, S/ε

Convert the following grammar into PDA.

$E \to E+E \mid E \times E \mid (E) \mid I$

$I \to a \mid b \mid Ia \mid Ib \mid I0 \mid I1$



ε, I/a
ε, I/b
ε, I/I0
ε, I/I1
ε, I/Ia
ε, I/Ib
ε, E/E+E
ε, E/E×E
ε, E/(E)
ε, E/I

2) From PDA to Grammar

The production in P are introduced by moves of PDA as follows;

i) S production are given by $S \rightarrow [q_0, z_0, q]$ $\forall q \in Q$.

ii) Each Erasing moves $\delta(q, a, z) = (q', \epsilon)$
$[q, z, q'] \rightarrow a$

iii) Each non-erasing are move $\delta(q, a, z) = (q_1, z_1, z_2, z_3, \ldots z_n)$
$[q, z, q'] \rightarrow a\,[q_1, z_1, q_2]\,[q_2, z_2, q_3] \ldots [q_n, z_n, q']$

Convert the given PDA into Grammar.

1) $\delta(q_0, 0, z) = (q_0, A z)$
2) $\delta(q_0, 1, A) = (q_0, AA)$
3) $\delta(q_0, 0, A) = (q_1, \epsilon)$

1) $\delta(q_0, 0, z) = (q_0, A z)$
$[q_0, z, q_0] \rightarrow 0\,[q_0, A, q_0]\,[q_0, z, q_0]$
$[q_0, z, q_0] \rightarrow 0\,[q_0, A, q_1]\,[q_1, z, q_0]$
$[q_0, z, q_1] \rightarrow 0\,[q_0, A, q_0]\,[q_0, z, q_1]$
$[q_0, z, q_1] \rightarrow 0\,[q_0, A, q_1]\,[q_1, z, q_1]$

$S \rightarrow [q_0, z_0, q_0]$
$S \rightarrow [q_0, z_0, q_1]$

2) $\delta(q_0, 1, A) = (q_0, AA)$

$[q_0, A, q_0] \rightarrow 1\,[q_0, A, q_0]\,[q_0, A, q_0]$

$[q_0, A, q_0] \rightarrow 1\,[q_0, A, q_1]\,[q_1, A, q_0]$

$[q_0, A, q_1] \rightarrow 1\,[q_0, A, q_0]\,[q_0, A, q_1]$

$[q_0, A, q_1] \rightarrow 1\,[q_0, A, q_1]\,[q_1, A, q_1]$

3) $\delta(q_0, 0, A) = (q_1, \varepsilon)$

$[q_0, A, q_1] \rightarrow 0$

$[q_0$

$[q_0$

$[q_0$

4) Convert the given ~~grammar~~ PDA into grammar

$\delta(q_0, b, z_0) = (q_0, z z_0)$

$\delta(q_0, \varepsilon, z_0) = (q_1, \varepsilon)$

$\delta(q_0, b, z) = (q_0, z z)$

$\delta(q_0, a, z) = (q_1, z)$

$\delta(q_1, b, z) = (q_1, \varepsilon)$

$\delta(q_1, a, z_0) = (q_0, z_0)$

P_1: $S \rightarrow [q_0, z_0, q_0]$

$S \rightarrow [q_0, z_0, q_1]$

1) $\delta(q_0, b, z_0) = (q_0, z z_0)$

$= [q_0, z_0, q_0] \Rightarrow b\,[q_0, z, q_0]\,[q_0, z_0, q_0]$

$[q_0, z_0, q_0] \Rightarrow b\,[q_0, z, q_1]\,[q_1, z_0, q_0]$

$[q_0, z_0, q_1] \Rightarrow b\,[q_0, z, q_0]\,[q_0, z_0, q_1]$

$[q_0, z_0, q_1] \Rightarrow b\,[q_0, z, q_1]\,[q_1, z_0, q_1]$

2) $\delta(q_0, \epsilon, z_0) = (q_1, \epsilon)$

$[q_0, z_0, q_1] \rightarrow \epsilon$

3) $\delta(q_0, b, z) = (q_0, zz)$

$[q_0, z, q_0] \Rightarrow b\,[q_0, z, q_0]\,[q_0, z, q_0]$

$[q_0, z, q_0] \rightarrow b\,[q_0, z, q_1]\,[q_1, z, q_0]$

$[q_0, z, q_1] \rightarrow b\,[q_0, z, q_0]\,[q_0, z, q_1]$

$[q_0, z, q_1] \Rightarrow b\,[q_0, z, q_1]\,[q_1, z, q_1]$

4) $\delta(q_0, a, z) = (q_1, z)$

$[q_0, z, q_0] \rightarrow a\,[q_0, z, q_0]$

$[q_0, z, q_1] \rightarrow a\,[q_0, z, q_1]$

5) $\delta(q_1, b, z) = (q_1, \epsilon)$

$[q_1, z, q_1] \rightarrow \epsilon$

6) $\delta(q_1, a, z_0) = (q_0, z_0)$

$[q_1, z_0, q_0] = a\,[q_0, z_0, q_0]$

$[q_1, z_0, q_1] = a\,[q_0, z_0, q_1]$

UNIT: 6

TURING MACHINE.

Turing machine: (Introduction)

The computational models such as DFA, NFA, ε-NFA, R.E, CFG, PDA et are limited in certain kind of problems such as Text searching, ~~fra~~ Parsing programs etc.

Now we are going to move to search the answer of questions like what language can be defined by any computational model available today.

We would like to find the answer of the questions like what present computer can do and what they cannot do?

Problems that computer cannot solve, anyway are called "undecidable".

Some examples are:

1) Fermat's last theorem.

For any integer, the computer will never find a triple of positive integer to satisfy $x^n + y^n = z^n$.

iii) A program that could examine any program P and input I for P and tell whether P, run for I as input produce some output.

There is another class of problem beside undecidable. Those problem that are decidable but took large amount of time to solve them are called interactable problem.

Thus, Alan turing in 1936 creates a simple tool allows us to prove whether a given problem is solvable, interactable or undecidable. The device is a finite automata having a single tapes (memory) known as turing machine.

Church - Turing Thesis:

Church's hypothesis:

All models of computation (computers and all other logical or physical device) have the same power as turing machine. i.e any language that could be accepted by any model of computation, could also be accepted by corresponding Turing Machine (TM).

The logician A church create an assumption that

"Any general way to ~~create~~ compute will allow us to compute only the partial recursive functions or what TM or modern computer can compute".

The Turing Machine:

Consists of a finite control which can be in any finite set of states, and a tape divided into cells which can hold any finite number of symbols as shown below.

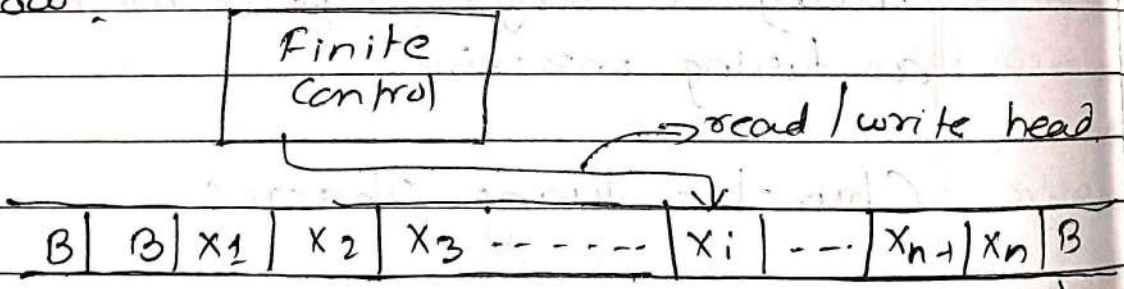


fig: Turing Machine.

Basic Assumption:

i) Initially, tape of TM contain input string surrounded by Blank symbol.

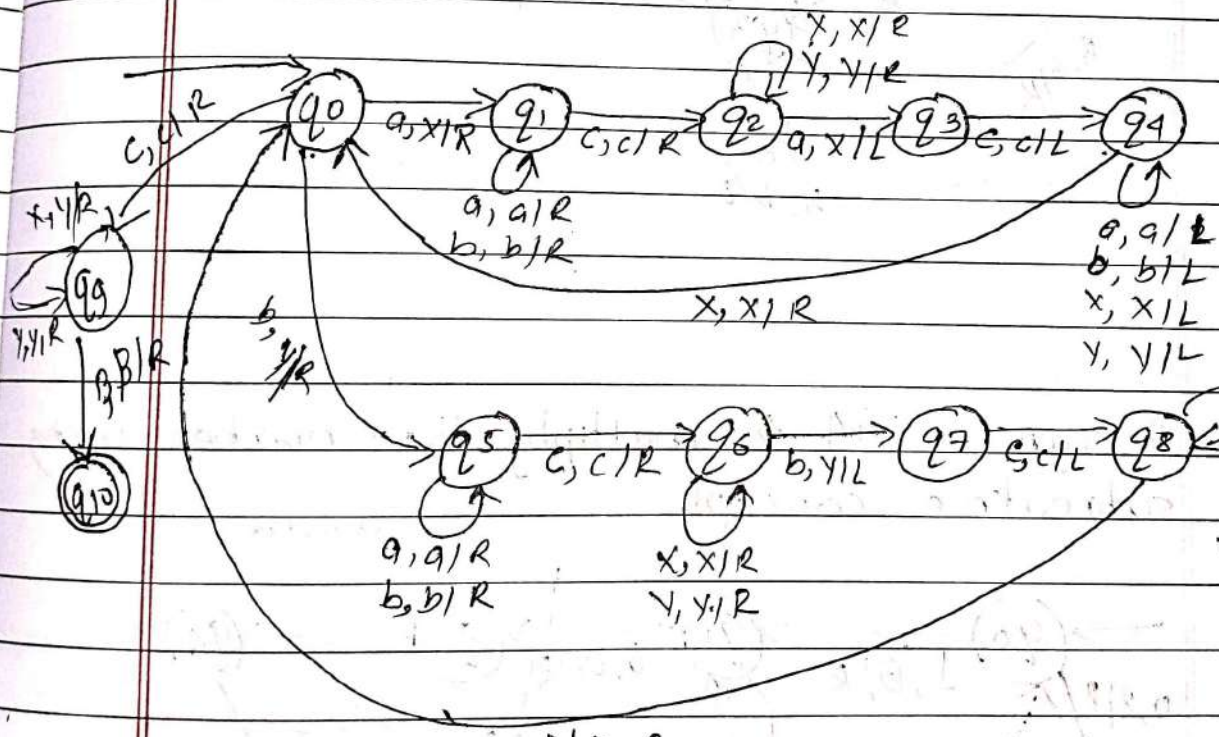
ii) Blank is a tape symbol but not a input symbol.

iii) There may be other tape symbols besides input & blank symbol.

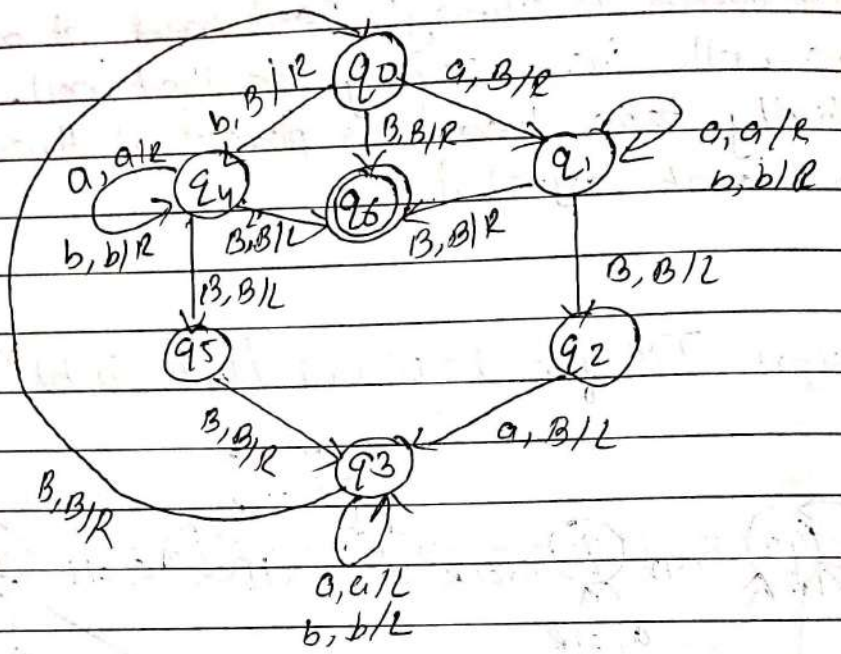
iv) Tape head is always positioned at one of the tape cells i.e scanning that cell.

v) Initially tape head is placed at the leftmost non-blank symbol.

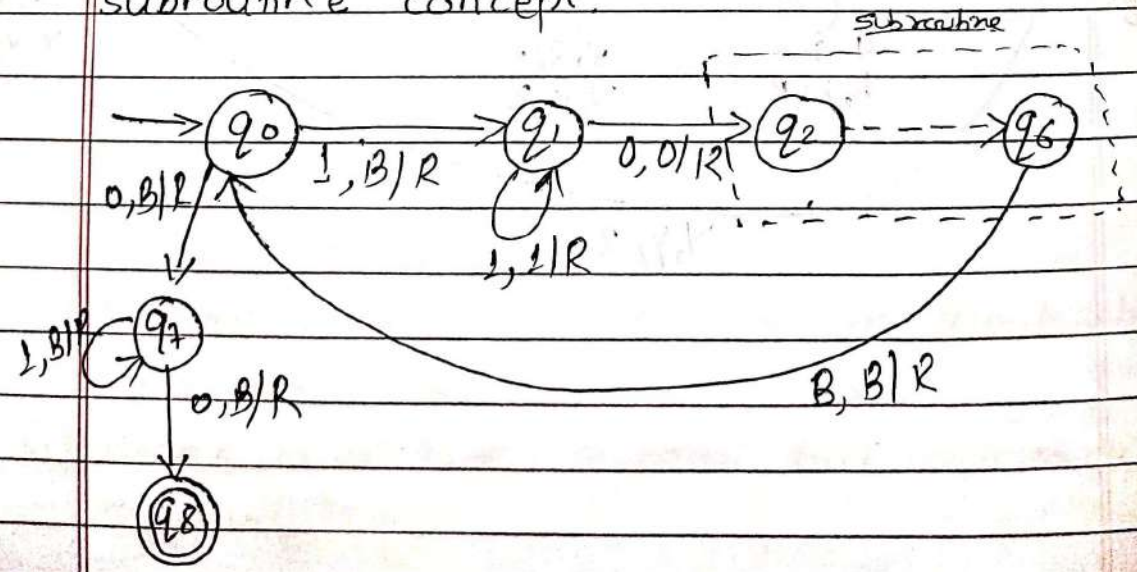
Q) Design TM for $L = wcw \mid w \in (a,b)^*$.



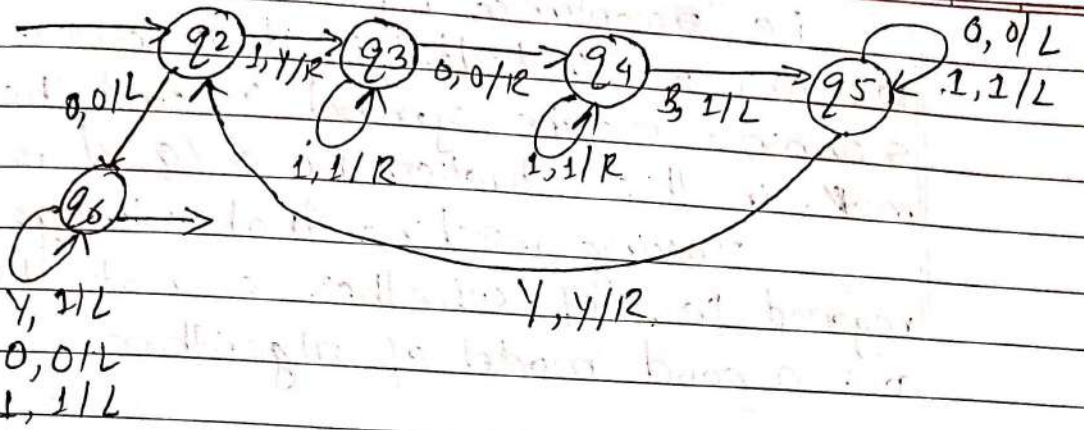
Design TM for language of palindrome.



Imp Design a TM to multiply two numbers using subroutine concept.



Subroutine:



The language of TM.

Let $M = (Q, \Sigma, T, \delta, q_0, B, F)$ be a TM then $L(M)$ is the set of string w in Σ^* such that

$(q_0, w) \vdash^* (\alpha, P, \beta)$

where, P is some state belongs to F & α & β are tape strings.

The set of language which can be accepted by TM is called Recursively Enumerable (RE) language.

Halting:

There is another notation of "acceptance" that is commonly used for TM.

i.e "acceptance by Halting".
we say a TM halts if it enters a state q_i scanning a tape symbol X_j and there is no move in this situation i.e $\delta(q, X)$ is undefined.

Turing machine that always halts, regardless of whether or not they accept are a good model of algorithm.

Programming Technique for Turing Machine.

Let's give some additional tricks which enhance the programming capability of a TM. But these trick donot control the basic model of the turing machine. These extended tricks are:

1) Storage in state.
2) Multiple tracks.
3) Sub routines.

1) Storage in state

We can use finite control not only to represent a position in the program of turing machine but also to hold a finite amount of data as memory as shown in figure below,

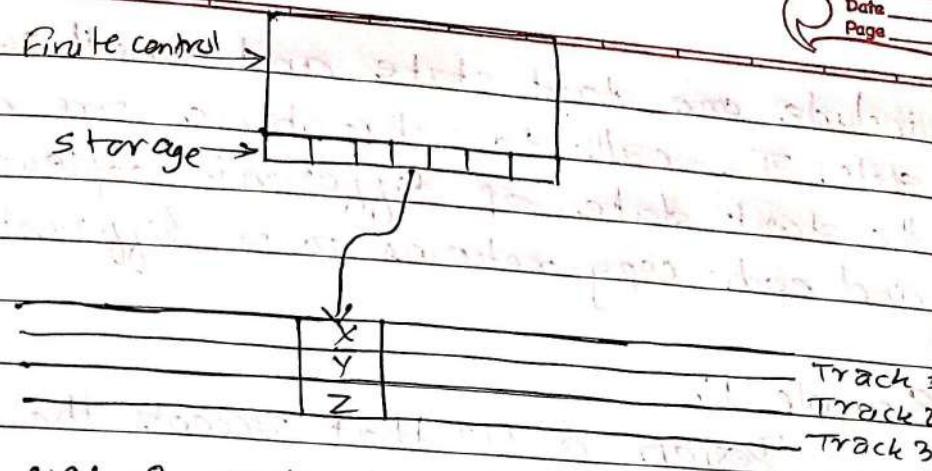


fig: A TM viewed as having finite control storage & multiple tracks.

2) Multiple tracks.

We can think of a tape as consisting of multiple tracks each can hold one symbols, and the tape alphabet of the turing machine consist of tuple with one component for each track.

Example: Cell scanned by the tape head in figure above contain the tape symbol [X, Y, Z].

3) Sub routines.

Just like programs in computer, turing machine can be considered as consists of subroutines. A turing machine subroutine is a set of states that performs some useful process. This subroutine consists of set of states which

include one start state and another returning state. The calls to subroutines are made to the start state of different copies of subroutines and each copy returns to a different state.

Example 1:

Design a TM that accepts the language $01^* + 10^*$ (use concept of storage in state)

Example 2:

Design a TM that accept $L=\{wcw/w\in(a+b)^*$ (use multiple track concept).

Example: 3

Design a TM that implement the function multiplication (use subroutine concept).

Variants of the TM (Extensions to the basic TM).

There are certain computation models related to turing machine and have the same language recognizing power as the basic model of a TM. One simple example is multitape turing machine, which is used to simulate the real computer, where extra tape in the TM at no power to the model as far as ability to accept language is concerned.

Another example is non determinastic turing machine that allow to make any of the finite set of moves in a given situation. This extension simply make programming easier in some situation but donot add any power to the basic model.

1) ~~Multip~~

1) Multitape TM.

The graphical representation of a multitape TM is shown as below.

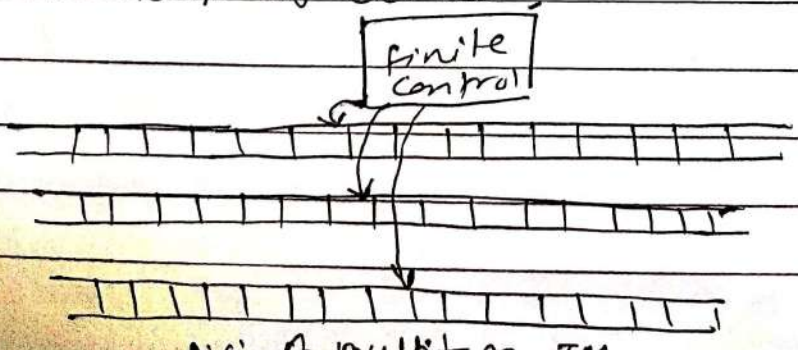


Fig: A multitape TM.

The device (MTM) has a finite control (state) and some finite number of tapes. Each tape is divided into cells, and each cell can hold any symbol of the finite tape symbol. There is a unique head associated with each tape scanning a particular cell.

Initially.

1) First, tape consists of finite sequence of input signal symbol and head of the first tape is at the left end of the input.
2) All other cells of all the tapes hold the blank.
3) All other tape heads, except first one are at some arbitrary cell.
4) The finite control is in the initial state.

The move of a multitape TM depends on the state and the symbol scanned by each of the tape head. In one move the multitape turing machine does the following:

1) The control enters to a new state (may be same state).
2) On each tape, a new tape symbol is written on the cell scanned.
3) Each of the tape heads makes a move, which can be either left, right or stationary. The heads

are independent so different heads may move in different directions and some may not move at all.

Multitape TM's like one tape TM's accepts the language by entering an accepting state.

Non-deterministic TM:

A Non-deterministic TM (NTM) differs from the deterministic by having a transition function δ such that for each state q and tape symbol X.

$\delta(q, X) = \{(q_1, X_1, D_1), (q_2, X_2, D_2) \ldots (q_n, X_n, D_n)\}$

where, n is a positive integer.

The NTM accept no language not accepted by a DTM. The NTM 'M' accepts an input 'w' if there is any sequence of choices of move that leads from the initial id with 'w' as input to an id with an accepting state.

Turing Machines & Computers:

If we see the capability of the computer, it has computing caliber exactly like Turing machine. They both can accept exactly the same language which is called recursively enumerable language. The above fact about similarity between computer and turing machine can be proved by dividing the problem into two category.

i) A computer can simulate a Turing machine.
ii) A turing machine can simulate a computer.

TM = C.

1) A computer can simulate a Turing machine.

C = TM.

Given a turing machine 'M', we must write a program that ack like 'M'.
We have to consider three aspect related with turing machine.

1) It's finite control.
2) It's tape.
3) It's finite number of transition rule.

Since there are finite number of states our program can encode states as character string and program maintains a table for representing transitio transactions. like wise tape symbols & input

alphabets can be encoded as character strings of a fixed length. We might face a little bit difficulty to simulate the TM tape in our program. Because the tape can grow infinitely long but computer's memory like RAM, hard-disk etc are finite.

A serious question here is, "can we simulate infinite tape with a fixed memory?" If

If there is no opportunity to replace the storage devices then in fact we cannot. However, today's computer have removable storage devices.

Thus we can arrange that the disks are placed into two stacks as suggested below.

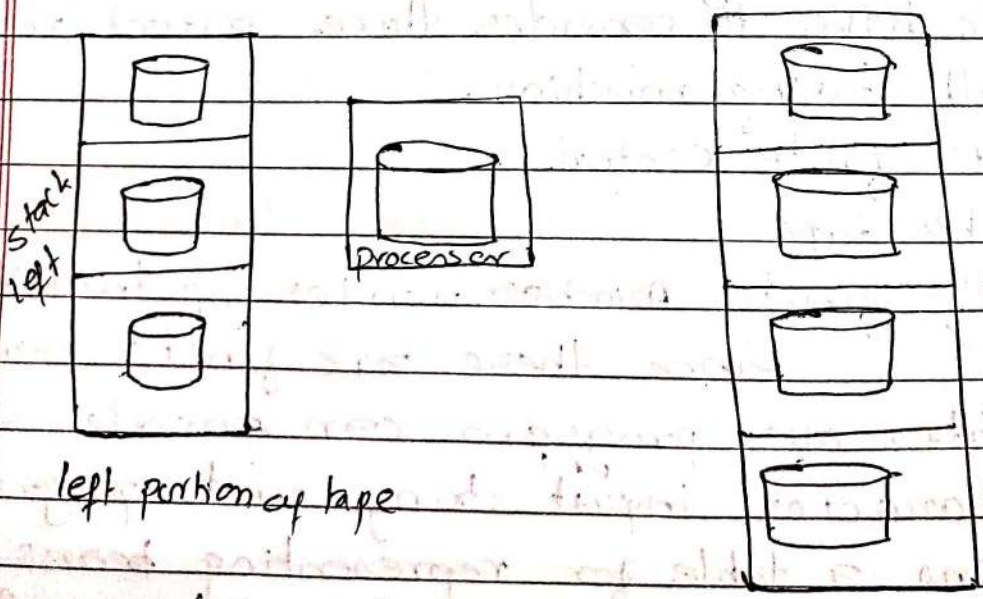


fig: Simulating TM using Computer.

Two stacks of disk are maintain by human operator in our program. If our head reach left most position on the tape on the current disk and again want to move left then program prints "swap left". And human operator push current disk in right stack and mount the CPU with disk at the top of the left stack.

Similar action is implemented if program prints "swap right" message. If either stack is empty when the computer ask that the disk from that stack be mounted then turing machine can be simulated by mounting the fresh copy of the disk to the stack.

Thus, we completely simulated TM using computer.

2) A turing machine can simulate a computer

$TM = C$

Before beginning the study of simulation of a computer by a TM lets list some simple assumptions about computer.

a) Suppose a computer consist of an indefinitely long sequence of words (may by 32 or

64 bit long) each with address. Assume addresses as integer 0, 1, 2, ... and so on.

b) Assume that the program is stored in some of the words of this memory.

c) Assume that each instruction involves limited number of words.

d) Rather than using registers as a fast memory we assume registers are the part of the memory as well.

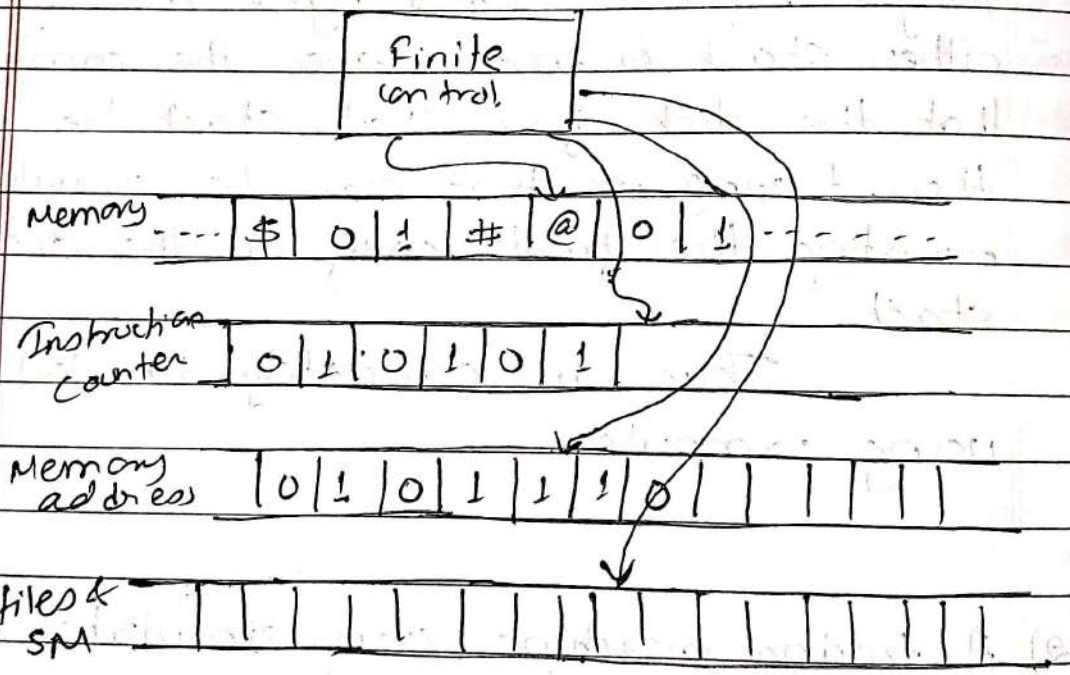


fig: A TM simulating computer

The simulated TM uses several tapes. The first tape represents the entire memory of computer. Special marker's symbols *, # &, $ are used to represent the end of address and content.

The second tape is the instruction counter and holds one integer in binary to represent the address of next instruction. The third tape holds a memory address or the content of address after the address has been located on tape 1. Now, our turing machine will simulate the instruction cycle of the computer as:

1) ~~Scans~~ search the first tape for an address on tape 2.
2) If instruction address found examine its value. Ex
Example: Find action of instruction. (copy, add, branch etc) and remaining bits code represent address that involve the action.
3) If instruction require value of some addresses then put that address into the third tape & save the current instruction on second track of the first tape.
4) Execute the instruction.

Equivalence of one tape & Multi tape TM.

Language defining capability of one tape TM and multitape TM are same i.e Multitape TM accepts all the recursively enumerable languages and no other languages that are not recursively enumerable are not accepted by multitape turing machine.

Since one tape TM is a multitape TM, it accept all the RE languages. But to prove that all multitape TM simulates only RE languages we have to prove the theorem below.

Theorem:

Simulation of multitape TM by one tape TM.

Every language accepted by multitape TM is recursively enumerable (RE).

Proof:

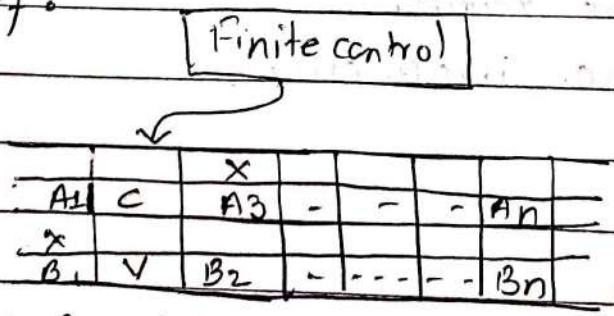


Fig: Simulation of Two-tape TM by one tape TM.

Suppose, Language L is accepted by a 'K' tape TM 'M'. We simulate 'M' with a one tape TM 'N' whose tape we think of as having 2K tracks. Half of 2K tracks holds the tapes of 'M' & half of the tracks is hold only a single marker that indicates the head position for the corresponding tape of 'M'.

Let K = 2 for a ~~multipe~~ multitape TM 'M'. Second and fourth ~~fo~~ track holds the content of M., track 1 holds the position of the head of tape 1 and track 3 holds the position of the 2nd tape head.

To simulate a move of 'M', N's head must visit the K head marker. After visiting each head marker and storing them in a component of its finite control, 'N' knows the state of 'M' & tape symbol at each of M's head.

'N' now re-visits each of the head marker's on its tape, changes the symbol in the track, representing the ~~cod~~ corresponding tapes of 'M', and move the head marker left or right if necessary. Finally 'N' changes the state of 'M' as recorded in its own finite control.

At this point, 'N' exactly simulates

the one move of 'M'. If we make M's accepting states as accepting state in 'N' then whenever the 'M' accepts 'N' also accept. & 'N' doesnot accept otherwise.

Hence, both Turing Machine accepts the same language i.e recursively enumerable language.