INTRODUCTION:-

Set:-

→ Set is a group of objects represented as unit.
→ objects in set are called its elements or members
→ is described by listing its elements inside braces.
e.g: $A = \{2, 6, 11, 15\}$
→ symbol $\in$ for set membership.
→ symbol $\notin$ for set non-membership.
e.g:
$2 \in \{2, 6, 11, 15\}$
$3 \notin \{2, 6, 11, 15\}$
→ neither the order of describing a set nor repetition of its elements matter.
i.e: $\{2, 11, 15, 6\}$
$\{2, 11, 2, 15, 6, 11, 2\}$
→ two sets are equal if and only if they have same elements.
→ A set with only one element is called singleton
i.e: $\{2\}$
→ A set with no elements at all is called empty set, denoted by ϕ.
→ A set that contain infinitely many elements is called an infinite set. $A = \{2, 7, 8, \ldots\}$
→ finite set is the one that is not infinite.

Sub-set:-

→ for any two sets A & B, A is subset of B if every member of A is also member of B.
→ written as $A \subseteq B$
e.g: $A = \{2, 5, 7\}$, $B = \{1, 5, 9, 3, 7, 2\}$
$\therefore A \subseteq B$

Proper subset:-

→ for two A & B, A is proper subset of B, if A is subset of B and not equal to B.
→ written as $A \subset B$
e.g: $A=\{1,3,5\}$, $B=\{1,5\}$, $C=\{1,5,3\}$, $D=\{1,6\}$
here,
$B \subset A$, $C \subseteq A$, $D \not\subset A$
→ any set is subset of itself.
→ empty set is subset of every set, including empty set itself. i.e: for set A,
$\phi \subset A$
→ sets containing elements can be described according to some rule.
for this,
$\{n \mid$ rule about $n\}$
or
: (such that)
e.g: 1) $\{n \mid n=n^2$ for $m \in N\}$
(it results set of perfect square)
e.g: 2) $A=\{n^4-4 : n$ is an integer and $0 \le n \le 20\}$
$\Rightarrow A=\{-4,-3,0,5,\dots\}$
e.g: 3) the set of odd natural numbers.
$B=\{x : x \in N$ & x is not divisible by $2\}$

SET OPERATIONS:-

(i) Union:-
- for two sets A & B, the union A & B, is the set containing all the elements in A & B into a single set. $A \cup B=\{x : x \in A$ or $x \in B\}$
e.g: $A=\{6,7,3\}$ Then,
$B=\{5,4,3\}$ $A \cup B=\{6,7,3,5,4\}$

(ii) Intersection:-
- for two sets A & B, the intersection of A & B, is the set of the elements that are both in A & B.
$A \cap B=\{x : x \in A$ & $x \in B\}$
e.g: 1) $A \cap B=\{3\}$
2) $A=\{6,2,3\}$, $C=\{4,9,5\}$
$\therefore A \cap C=\phi$

(iii) Complement:-
- for given set A, complement of A, is the set of all elements that are not in A.
- written then as $\overline{A}=\{x : x \notin A\}$

(iv) Difference:-
- for two sets A & B, the difference of A & B is the set of all elements of A but not in B.
$A-B=\{x : x \in A$ & $x \notin B\}$
e.g:
$A=\{6,7,3\}$
$B=\{5,4,3\}$
$A-B=\{6,7\}$

Set Operations

Let A, B & C are three arbitrary sets, U represents universal set and ϕ is empty set.

Then,

1) **Idempotency law**: $A \cup A = A$
   $A \cap A = A$

2) **Commutative law**: $A \cup B = B \cup A$
   $A \cap B = B \cap A$

3) **Absorptive law**: $A \cup (A \cap B) = A$
   $A \cap (A \cup B) = A$

4) **Associative law**:- $A \cup (B \cup C) = (A \cup B) \cup C$
   $A \cap (B \cap C) = (A \cap B) \cap C$

5) **Distributive law**:-
   $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$
   $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$

6) **De-Morgan's law**:-
   $(A \cup B)' = A' \cap B'$
   $(A \cap B)' = A' \cup B'$

7) **Others laws involving complements**:-
   $(A')' = A$
   $A \cap A' = \phi$
   $A \cup A' = U$

8) **Laws involving empty set**:-
   $A \cup \phi = A$, $A \cap \phi = \phi$

9) **Laws involving universal set**:-
   $A \cup U = U$
   $A \cap U = A$

Disjoint Set:

→ Two sets are disjoint if they have no elements in common i.e their intersection is empty.

Power Set:

→ For given set A, power set of A represented as 2^A is the collection of all subsets of set A.

Let, $A = \{1, 2, 3\}$

$2^A = \{\phi, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}\}$

Cartesian Product:

→ for two sets A & B, cartesian product is defined as

$A \times B = \{(x, y) : x \in A \ \& \ y \in B\}$

e.g:

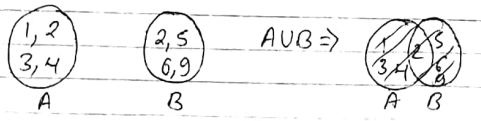
$A = \{2, 4, 6\}$ $B = \{1, 3\}$

$A \times B = \{\{2, 1\}, \{4, 1\}, \{6, 1\}, \{2, 3\}, \{4, 3\}, \{6, 3\}\}$

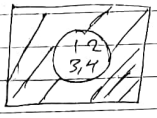
Venn-Diagram:

→ Set diagram that shows all possible logical relation betⁿ finite collection of sets.

e.g: $A = \{1, 2, 3, 4\}$ $B = \{2, 5, 6, 9\}$



$A \cap B \Rightarrow$ 

$A' \Rightarrow$ 

$A - B \Rightarrow$ 

Sequence and Tuples:

→ Sequence of objects is a list of objects in some order.

(2, 5, 9, 11)

here,

$(2,5,9,11) \neq (2,9,5,11)$

$(2,5,9,11) \neq (2,5,9,5,11,11)$

→ As set, sequences may be finite or infinite.
→ finite sequences often are called tuples.
→ A sequence with k elements is called k-tuple.
→ A 2 tuple is called a pair.

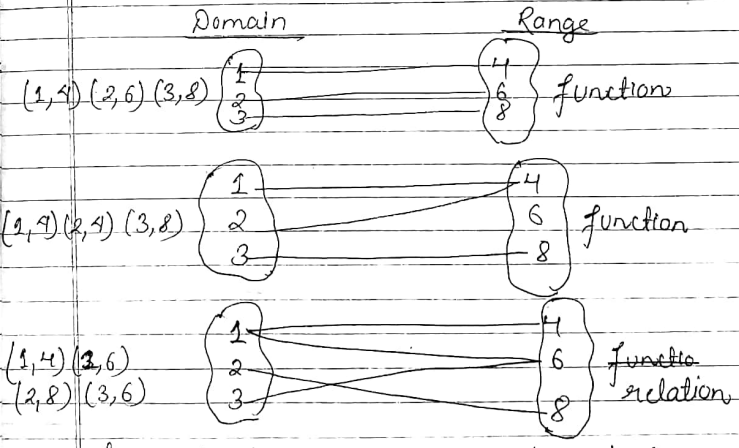
Function and Relation:

→ Relations is just association or relationship between set of informations.
→ ordered pair is way of grouping objects.
(2,4) is ordered pair of objects 2 & 4 where 2 & 4 are called components.
→ Functions is an object that sets up an input-output relation.
i.e: it takes input & produce output.
→ if f is a function with input a, and output then $f(a)=b$
→ function is also ~~termed~~ termed as mapping
i.e: $f(a)=b$; f maps a to b.

→ The set of possible inputs to the function is called its domain.
→ The output of function is called Range.
→ $f: D \to R \Rightarrow f$ is a function with domain D & range R.
→ function is a well behaved relation

→ in-case of function:-
for (x, y)
- each x has only one y.

→ in relation:-
- one, some or all ~~x is~~ x's can have more than one y.



here, all functions are relations but not all relations are functions.

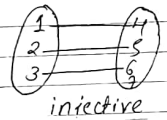
→ A function f from A to B, denoted by $f: A$ is a binary relation R, having the property that for each elements $a \in A$, there is exactly are ordered pair in R, whose first component is a.

→ the set A is called Domain of f & the set $\{b \in B$: there is an $a \in A$ with $f(a)=b\}$ is range of f.

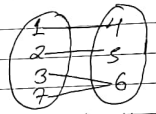
→ A function $f: A \to B$ is one to one (or injective) if for any two distinct elements a and a' in A, we have $f(a) \neq f(a')$

→ The function f is onto (or subjective) if for element $b \in B$, there exists an element $a \in A$, such that $f(a)=b$ i.e: range of f is equal to B.

→ The function f is bijection, if f is both injective & subjective.

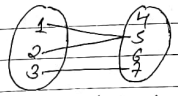


injective



subjective

Bijective



Non injective & non subjective

RELATION'S TYPES

1: **Transistive Relation**

- R is transistive if for all a, b & c in A $(a,b) \in R$ and $(b,c) \in R$ then also $(a,c) \in R$

e.g: let $A=\{1,2,3\}$

then $R=\{(1,2),(2,3),(1,3)\}$ is a transitive rel^n

2: **Equivalence Relation**

- Relation $R \subseteq A \times A$ is an equivalence rel^n, if it is reflective symmetric & transitive.

e.g: let $A=\{1,2,3\}$

then

$R=\{(1,1),(2,2),(3,3),(1,2),(1,3),(2,3),(2,1)$
$(3,1),(3,2)\}$

3: **Partial ~~on~~ order Relation**

- Let R be a relation defined on set A then R is partial order relation if it is reflexive, anti-symmetric & transitive.

e.g: let $A=\{1,2,3\}$

then $R=\{(1,1),(2,2),(3,3),(1,2),(2,3),(1,3)\}$

4: **Total order Relation**

- A partial order relation is total order if for all a & b in A,

either $(a,b) \in R$ or $(b,a) \in R$

e.g: let $A=\{1,2,3\}$

$R=\{(1,1),(2,2),(3,3),(2,1),(2,3),(1,3)\}$

5: **Reflexive Relation**

- R is reflexive if for every element in $a \in A$, we have $(a,a) \in R$.

e.g: let $A=\{1,2,3\}$
then $R=\{(1,1),(2,2),(3,3)\}$ is reflexive relation defined on set A.

6: Symmetric/Anti-symmetric Relation:
-R is symmetric for all a & b in A. if
If $(a,b)\in R$ then also $(b,a)\in R$
e.g: let $A=\{1,2,3\}$
then
$R=\{(1,2),(2,1),(1,3),(3,1)\}$ is symmetric relation on set A.
[* if the relⁿ is not symmetric then it in anti-symmetric].

Equivalence Class
-Let 'R' be an equivalence relation defined on set A then equivalence class is
$[a]=\{x: a\in A, (a,x)\in R\}$
e.g: let $A=\{1,2,3\}$
$R=\{(1,1),(2,2),(3,3),(2,1),(1,2)\}$
Then equivalence classes be
~~$[1]=\{(1,1),(2,1)\}$~~
~~$[2]=\{(2,2),(2,1)\}$~~
$[1]=\{1,2\}$
$[2]=\{1,2\}$
$[3]=\{3\}$

QUOTIENT SET
-is the collection of equivalence classes, represented by A/R
e.g: $A/R=\{\{1,2\},\{3\}\}$

RELATION
→relation is the subsets of cartesian product for two sets A and B, and relation R then, $R\subseteq A\times B$

Q: Let A be a set of first n ~~partial~~ positive integers given by $A=\{1,2,3,\dots\}$ & R be a relⁿ defined on set A as
$R=\{(a,b): a+b \text{ is divisible by } 2\}$
find equivalence classes & quotient set.

soln
$A=\{1,2,3,\dots\}$
$R=\{(1,1),(1,3),(1,5)\dots(2,2),(2,4),(2,6)\dots$
$(3,1),(3,3),(3,5)\dots\}$

Equivalence class
$[1]=\{1,3,5\dots\}$
$[2]=\{2,4,6\dots\}$
$[3]=\{1,3,5,\dots\}$

Quotient set:-
$A/R=\{\{1,3,5\dots\},\{2,4,6,\dots\}\dots\}$

Q: Let $A=\{1,2,3,4\}$
R be a relation defined as '≤' on set A now test whether relation R is partial or not.

ALPHABETS

- an alphabet is a finite, non-empty set, whose elements are called symbols.
- it is denoted by Σ

e.g: $\Sigma = \{0,1\}$

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

STRING

- a string over an alphabet Σ is a finite sequence of symbols, where each symbol is element of Σ.
- it is denoted by w.
- the length of a string w, denoted by $|w|$ is the no. of symbols contained in w.
- the empty string denoted by Σ or e is string having length zero.

e.g: let $\Sigma = \{0,1\}$ be alphabet

then

10, 1000, 0, 101 and e are

- the set of all strings over an alphabet denoted by Σ^*

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

$\Sigma^* = \{e, 0, 1, 01, 10, 001, \cdots\}$

- If we exclude Σ or e, then we write Σ^+ (excluding empty string)

$\Sigma^* = \Sigma^+ \cup \{\Sigma\}$ (including empty string)

e.g: let, $\Sigma = \{0,1\}$ be alphabet then

10, 1000, 0, 101 and e are strings over Σ, having length 2, 4, 1, 3, 0 respectively.

POWER OF ALPHABET

- the set of all strings of certain length k from an alphabet is the k^{th} power of alphabet i.e: $\Sigma^k = \{w : |w| = k\}$

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

$\Sigma^0 = \{0\}$

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

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

CONCATENATION OF STRING

- two strings over the same alphabet can be combined to form third by operation of concatenation.
- concatenation of two strings x and y, denoted by $x \cdot y$ or xy is the string x followed by string y.
- formally,

$w = xy$ if and only if

$|w| = |x| + |y|$

$w(i) = x(i)$ for $i = 1 \cdots |x|$

$w(|x| + j) = y(j)$ for $j = 1 \cdots |y|$

e.g: if $x = 001$ & $y = 101$ then

$w = x \cdot y = 001101$.

SUBSTRING

- A string V is substring of a string w if and only if there are strings x and y such that $w = xvy$
- Both x and y could be ϵ
- A string is, substring of string.

-if $w = xv$ for some x, v is suffix of w
-if $w = yvy$ for some y, v is prefix of w

Kleen Closure:-
- the set of all strings over an alphabet Σ is called Kleen closure of Σ.
- is denoted by Σ^*
$\therefore \Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \ldots$

Positive Closure:-
- the set of all strings over an alphabet Σ except empty string is called positive clo
- denoted by Σ^+
$\therefore \Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \ldots$

Reversal of String:-
→ for any string w, its reversal denoted w^R, is a string spelled backward.
e.g:
w = neeal
w^R = Leen

Language:-
→ any set of strings over an alphabet i.e: any subset of Σ^*, is called language.
→ $L \subseteq \Sigma^*$
e.g:
$L = \{w \in \{a, b\}^* : w$ has odd no.s of $a\}$
here,
$L = \{a, ab, aaab, aaba, \ldots\}$

Regular Expression
-is a formula for representing a language in terms of a form combined using 3 operations union, concatenation and Kleen closure
-describes a language exclusively by means of single symbol and ϕ, combined perhaps with symbols $\cup$ and $*$, possibly with help of parenthesis.
-the regular expression over an alphabet Σ^* are all strings over the alphabet $\Sigma \cup \{(,), \phi, \cup, *\}$ that can be obtained as follows:-
i) ϕ and each member of Σ is regular expression.
ii) if α and β are regular expressions then so is $(\alpha\beta)$
iii) if α and β are regular expressions then so is $(\alpha \cup \beta)$
iv) if α is regular expression then so is α^*
v) nothing is regular expression unless it follows from i) through iv)

1) $L = \{w \in \{a, b\}^* : w$ has even no. of a followed by odd no. of $b\}$
$L = \{b, aab, aabbb, aaabb \ldots\}$

$\boxed{(aa)^*(bb)^*b}$

2) $L = \{w \in \{0, 1\}^* : w$ has strings of 0s and 1's ending in $00\}$

$L = \{0100, 110100, 100 \ldots\}$ $\boxed{(0 \cup 1)^* 00}$

3) $L = \{w \in \{0,1\}^* : w$ has strings of 0's and 1's begining with 0 and ending with 1 $\}$

$L = \{01, 001, 0011, \ldots\}$

$\boxed{0(0 \cup 1)^* 1}$

- the relation between regular expression and the language they represent is established by a function L, such that if α is any regular expression then $L(\alpha)$ is language represented by α.
- the function L is defined as follow:-
  1) $L(\phi) = \phi$ and $L(a) = \{a\}$ for each $a \in \Sigma$
  2) if α and β are regular expressions then $L((\alpha \cup \beta)) = L(\alpha) \cup L(\beta)$
  3) if α and β are regular expressions then $L((\alpha\beta)) = L(\alpha) L(\beta)$
  4) if α is regular expression then $L(\alpha^*) =$

e.g:
What language is represented by following regular expression?

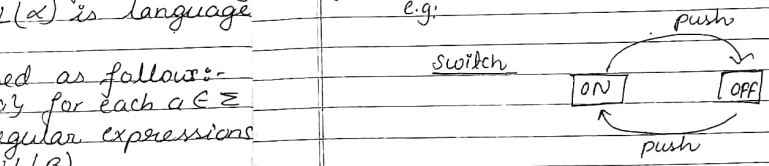
$L((L(a \cup b)^* a))$
$= L((a \cup b)^*) L(a)$
$= L((a \cup b)^*) \{a\}$
$= L((a \cup b))^* \{a\}$
$= (L(a) \cup L(b))^* \{a\}$
$= (\{a\} \cup \{b\})^* \{a\}$
$= \{a, b\}^* \{a\}$
$= \{w \in \{a,b\}^* : w$ end with an $a\}$

Chapter 2: Finite Automata / Finite State Machine

Introduction

→ Finite automata or finite state machine is the model of computation that accepts / rejects regular languages.
→ It captures all possible states and transition that a machine can take while responding to a sequence of input symbols.
→ It has finite set of states, edge that lead from one state to another and edge labelled with a symbol.

e.g:



Block-diagram:-

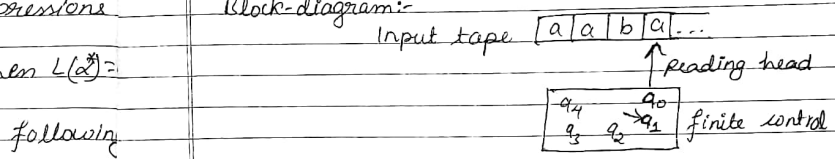


fig:- finite automata

→ here, input tape is device where strings are fed.
→ finite control is main part with finite no of states.
→ finite control can sense what symbol is written at any position on the input tape by means of moveable reading head.

→

→ finite automata is of two types:-

(i) Deterministic finite Automata (DFA)
→ can exist in only one state at any given

(ii) Non-Deterministic finite Automata (NDFA)
→ can exist in mulliple states at same time

(i) Deterministic Finite Automata (DFA)

→ DFA is defined by 5 tuples $(Q, \Sigma, \delta, q_0, F)$
where,

$Q \rightarrow$ a finite set of ~~tuple~~ states
$\Sigma \rightarrow$ a finite set of input symbols (alphabets)
$q_0 \rightarrow$ a start state
$F \rightarrow$ set of final state
$\delta \rightarrow$ a transition function that takes a st and a symbol as argument and returns state.

here,
$q_0 \in Q$
$F \subseteq Q$
$f \Rightarrow Q \times \Sigma \rightarrow Q$
here, $Q \times \Sigma$ is set of 2 tuples (q, a) with $q \in Q$ $a \in \Sigma$

DFA's

~~DATA~~

→ are often represented by state or transition diagra
→ transition function can also be represented by table called transition table.

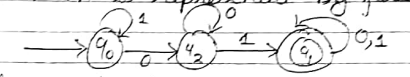
e.g: $Q = \{q_0, q_1, q_2\}$
$\Sigma = \{0, 1\}$
$q_0 = q_0$
$F = \{q_2\}$

δ is as follows:-

Q	0	1
q_0	q_2	q_0
q_1	q_1	q_1
q_2	q_2	q_1

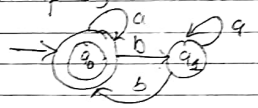
$\delta(q_0, 0) \rightarrow q_2$
$\delta(q_0, 1) \rightarrow q_0$
$\delta(q_1, 0) \rightarrow q_1$
$\delta(q_1, 1) \rightarrow q_1$
$\delta(q_2, 0) \rightarrow q_2$
$\delta(q_2, 1) \rightarrow q_1$

Now, DFA is represented by following state diagram,



This automata accept string 1010 and reject 1110.

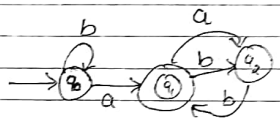
Q: Design a DFA that accepts the language given by
$L = \{w \in \{a, b\}^* : w$ has even no. of $b\}$
$L = \{abb, bbbb, bbbbbb\}$



$L = (a)^*(bb)^*$

→

$Q = \{q_0, q_1\}$
$\Sigma = \{a, b\}$
$q_0 = q_0$
$F = \{q_0\}$



δ:

Q	a	b
q_0	q_0	q_1
q_1	q_1	q_0

• Q. Design a DFA that accepts the language given:

$L = \{w \in \{a,b\}^* : w$ doesn't contain three consecutive $a\}$

$L = \{ab, abb, aabb, \ldots\}$

→ $Q = \{q_0, q_1, q_2, q_3\}$

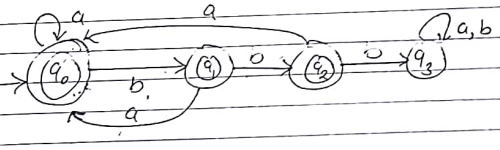
$\Sigma = \{a, b\}$

$q_0 = q_0$

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

δ:

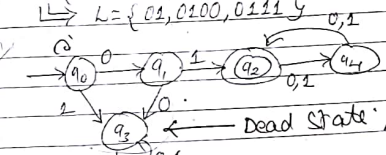
Q	a	b
q_0	q_0	q_1
q_1	q_0	q_2
q_2	q_0	q_3
q_3	q_3	q_3



• Q. Design a DFA that accepts the language

$L = \{w \in \{0,1\}^* : w$ starts with 01 having even length$\}$

↳ $L = \{01, 0100, 0111\}$



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

when, $Q = \{q_0, q_1, q_2, q_3, q_4\}$

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

$q_0 = q_0$

$F = \{q_2\}$

δ:- transition table

Q	0	1
q_0	q_1	q_3
q_1	q_3	q_2
q_2	q_4	q_4
q_3	q_3	q_3
q_4	q_2	q_2

0111011

$\Rightarrow \delta(q_0, 0111011)$

$\vdash (q_1, 111011)$

$\vdash (q_2, 11011)$

$\vdash (q_4, 1011)$

$\vdash (q_2, 011)$

$\vdash (q_4, 11)$

$\vdash (q_2, 1)$

$\vdash (q_4, \epsilon)$

$\therefore q_4$ is not in set of final state so, DFA rejects.

010110

$\Rightarrow \delta(q_0, 010110)$

$\vdash (q_1, 10110)$

$\vdash (q_2, 0110)$

$\vdash (q_4, 110)$

$\vdash (q_2, 10)$

$\vdash (q_4, 0)$

$\vdash (q_2, \epsilon)$

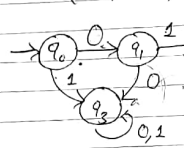
$\therefore q_2$ lies in final state so, DFA accepts.

Q: Design a DFA that accepts the language $L = \{w \in \{0,1\}^* : w \text{ starts with } 01\}$

$\Downarrow$

$L = \{01, 0100, 010, 0111, \cancel{10}\}$

n+1 | 2+1 = 3



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

when,

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

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

$q_0 = q_0$

$F = \{q_2\}$

δ :-

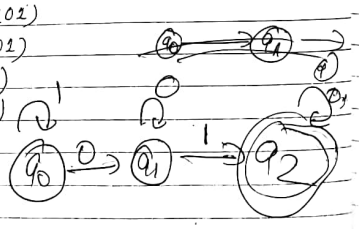
Q	0	1
q_0	q_1	q_3
q_1	q_2	q_2
q_2	q_2	q_2
q_3	q_3	q_3

$(q_0, 0) \to q_1$
$(q_0, 1) \to q_3$
$(q_1, 0) \to q_2$
$(q_1, 1) \to q_2$

0! = n+1 = 3

e.g: (i) 00101 ✗

$\delta(q_0, 00101)$
$\delta(q_1, 0101)$
$\delta(q_3, 101)$
$\delta(q_3, 01)$
$\delta(q_3, 1)$
$\delta(q_3, \epsilon)$



DFA...

→ A string X is accepted by a DFA $(Q, \Sigma, q_0, \delta, F)$ if $\delta(q, x) = P \in F$

→ The language of DFA $M = (Q, \Sigma, q_0, \delta, F)$ denoted by $L(M)$ is a set of string over Σ^* that are accepted by M. i.e: $L(M) = \{w : \delta(q_0, w) = P \in F\}$

Non-Deterministic Finite Automata (NDFA or NFA):-

→ It is a finite automata where several possible next states for a given combination of current state and input symbol could exists.

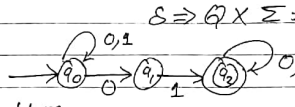
→ it reads the input string and may choose at each step to go to any of legal states.

→ here, choice is not determined by anything in model, so non-deterministic.

→ NFA is defined by 5 tuple $(Q, \Sigma, \delta, q_0, F)$

where, Q = finite set of states
Σ = finite set of input symbols (alphabets)
q_0 = a start state
F = set of final state
δ = transition function,
where,
$\delta \Rightarrow Q \times \Sigma$ = subset of Q.

e.g:



Here,

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

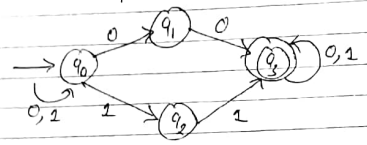
$\Sigma = \{0,1\}$

$q_0 = q_0$

$F = \{q_2\}$

δ: Q	0	1
q_0	$\{q_0, q_1\}$	q_0
q_1	ϕ	q_2
q_2	q_2	q_2

e.g:



subset = {0011}

Here,

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

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

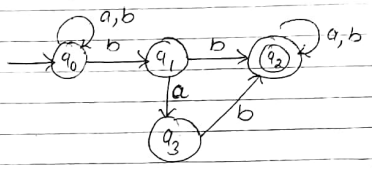
$q_0 = q_0$

$F = \{q_3\}$

Q	0	1
q_0	$\{q_0, q_1\}$	$\{q_0, q_2\}$
q_1	q_3	ϕ
q_2	ϕ	q_3
q_3	q_3	q_3

• Q: Design a NDFA that accepts the set of strings containing occurrence of pattern bb or bab.

⇒



Here,

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

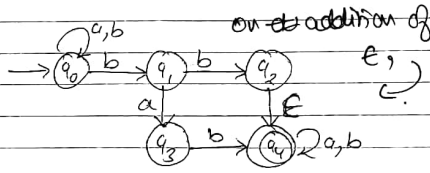
$\Sigma = \{a, b\}$

$q_0 = q_0$

$F = \{q_2\}$

Q	a	b
q_0	q_0	$\{q_0, q_1\}$
q_1	q_3	q_2
q_2	q_2	q_2
q_3	ϕ	q_2

⇒

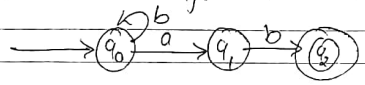


On addition of ε,

ϕ ⇒ Dead configuration, if this. No more we put

• Q: Design a NDFA for $R = b^*ab$

⇒



Here,

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

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

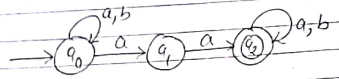
$q_0 = q_0$

$F = \{q_2\}$

Q	a	b
q_0	q_1	q_0
q_1	ϕ	q_2
q_2	ϕ	ϕ

• Q: Design a NDFA over {a,b} that accepts strings having aa as substring

⇒



Here, $Q = \{q_0, q_1, q_2\}$
$\Sigma = \{a, b\}$
$q_0 = q_0$
$F = \{q_2\}$

~~Q. Design a~~ Transition function:-

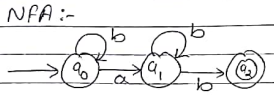
Q	a	b
q_0	$\{q_0, q_1\}$	q_0
q_1	q_2	ϕ
q_2	q_2	q_2

EQUIVALENCE OF DFA & NFA:-

- For each NDFA, there is an equivalent deterministic finite automata.
- Conversion steps:-
  (i) Observe all the transitions from starting state q_0, for every symbol in Σ.
  (ii) For new states from (i), check all transition for every Σ.
  (iii) Repeat step (ii) till new states are appeared.

e.g: ①

• Given NFA:-



Step 1: initial state q_0, for new DFA $\{q_0\}$

now,

$\delta'(\{q_0\}, a) \Rightarrow \delta(q_0, a) = \{q_1\}$ - new state

$\delta'(\{q_0\}, b) \Rightarrow \delta(q_0, b) = \{q_0\}$

Step 2: for new state $\{q_1\}$

$\delta'(\{q_1\}, a) \Rightarrow \delta(q_1, a) = \phi$

$\delta'(\{q_1\}, b) \Rightarrow \delta(q_1, b) = \{q_1, q_2\}$ - new state

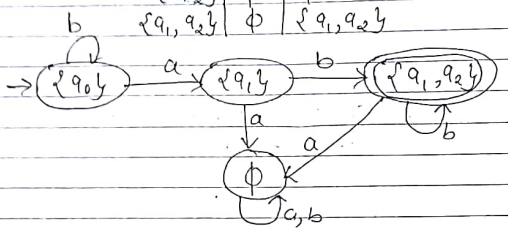
Step 3: for new state $\{q_1, q_2\}$

$\delta'(\{q_1, q_2\}, a) \Rightarrow \delta(q_1, a) \cup \delta(q_2, a)$
$\Rightarrow \phi \cup \phi \Rightarrow \phi$

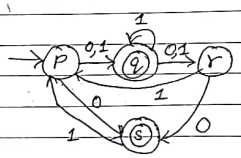
$\delta'(\{q_1, q_2\}, b) \Rightarrow \delta(q_1, b) \cup \delta(q_2, b)$
$= \{q_1, q_2\} \cup \phi$
$= \{q_1, q_2\}$.

Now, transition table:-

Q	a	b
$\{q_0\}$	$\{q_1\}$	$\{q_0\}$
$\{q_1\}$	ϕ	$\{q_1, q_2\}$
~~$\{q_2\}$~~	ϕ	ϕ
$\{q_1, q_2\}$	ϕ	$\{q_1, q_2\}$



e.g. ②



Here,
initial state of NFA P.
For, DFA initial state $\{P\}$

Now,

Step 1:-

$\delta'(\{P\}, 0) \Rightarrow \{q, s\}$, new state

$\delta'(\{P\}, 1) \Rightarrow \delta(P, 1) \Rightarrow \{q\}$, new state

Step 2:-

for $\{q, s\}$

$\delta'(\{q, s\}, 0) \Rightarrow \delta(q, 0) \cup \delta(s, 0)$

$\Rightarrow \{r\} \cup \phi$

$\Rightarrow \{r\}$, new state

$\delta'(\{q, s\}, 1) \Rightarrow \delta(q, 1) \cup \delta(s, 1)$

$\Rightarrow \{q, r\} \cup \{P\}$

$\Rightarrow \{q, r, P\}$; new state

Step 3:-

for $\{q\}$

$\delta'(\{q\}, 0) \Rightarrow \delta(q, 0) \Rightarrow \{r\}$

$\delta'(\{q\}, 1) \Rightarrow \delta(q, 1)$

$\Rightarrow \{q, r\}$, new state

Step 4:-

for $\{r\}$

$\delta'(\{r\}, 0) \Rightarrow \delta(r, 0) \Rightarrow \{s\}$, new state

$\delta'(\{r\}, 1) \Rightarrow \delta(r, 1) \Rightarrow \{P\}$

Step 5:-

for $\{q, r, P\}$

$\delta'(\{q, r, P\}, 0) \Rightarrow \delta(q, 0) \cup \delta(r, 0) \cup \delta(P, 0)$

$\Rightarrow \{r\} \cup \{s\} \cup \{q, s\}$

$\Rightarrow \{r, s, q\}$; new state

$\delta'(\{q, r, P\}, 1) \Rightarrow \delta(q, 1) \cup \delta(r, 1) \cup \delta(P, 1)$

$\Rightarrow \{q, r\} \cup \{P\} \cup \{q\}$

$\Rightarrow \{q, r, P\}$

Step 6:-

for $\{q, r\}$

$\delta'(\{q, r\}, 0) \Rightarrow \delta(q, 0) \cup \delta(r, 0)$

$\Rightarrow \{r\} \cup \{s\}$

$\Rightarrow \{r, s\}$; new state

$\delta'(\{q, r\}, 1) = \delta(q, r) \cup \delta(r, 1)$

$= \delta(q, r) \cup \{P\}$

$= \{q, r, P\}$

Step 7:-

for {r,s,q}

$\delta'(\{r,s,q\},0) \Rightarrow \delta(r,0) \cup \delta(s,0) \cup \delta(q,0)$

$\Rightarrow \{s\} \cup \phi \cup \{r\}$

$= \{r,s\}$

$\delta'(\{r,s,q\},1) \Rightarrow \delta(r,1) \cup \delta(s,1) \cup \delta(q,1)$

$\Rightarrow \{p\} \cup \{p\} \cup \{q,r\}$

$\Rightarrow \{q,r,p\}$

Step 8:-

for {r,s}

$\delta'(\{r,s\},0) \Rightarrow \delta(r,0) \cup \delta(s,0)$

$\Rightarrow \{s\} \cup \phi \Rightarrow \{s\}$

$\delta'(\{r,s\},1) \Rightarrow \delta(r,1) \cup \delta(s,1)$

$\Rightarrow \{p\} \cup \{p\} = \{p\}$

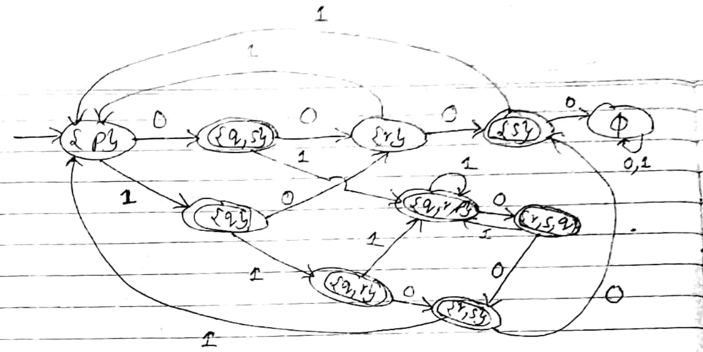
Step 9:-

for {s}

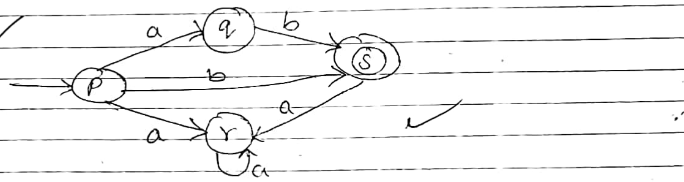
$\delta'(\{s\},0) \Rightarrow \delta(s,0) \Rightarrow \phi$

$\delta'(\{s\},1) \Rightarrow \delta(s,1) \Rightarrow \{p\}$

Q	0	1
{p}	{q,s}	{q}
{q,s}	{r}	{q,r,p}
{q}	{r}	{q,r}
{r}	{s}	{p}
{q,r,p}	{r,s,q}	{q,r,p}
{q,r}	{r,s}	{q,r,p}
{r,s,q}	{r,s}	{q,r,p}
{r,s}	{s}	{p}
{s}	ϕ	{p}



Q:



Converting E-NFA to DFA

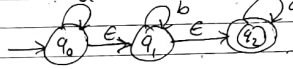
Steps:-

1) Find out the E-closure of starting st
and calculate union of E-closure of
transition for every symbol of Σ.
2) Repeat same process till new states
are approved.

here,

E-closure of state q can be obtained
following all transitions out of q that
labelled ε.

e.g:



here,

E-closure of q_0 is $E(q_0)$; $E(q_0) = \{q_0, q_1, q_2\}$
$E(q_1) = \{q_1, q_2\}$, $E(q_2) = \{q_2\}$

Now,

Step 1: $\delta'(\{q_0, q_1, q_2\}, a) = E(q_0) \cup \phi \cup E(q_2)$
$= \{q_0, q_1, q_2\} \cup \phi \cup \{q_2\}$
$= \{q_0, q_1, q_2\}$

$\delta'(\{q_0, q_1, q_2\}, b) = \phi \cup E(q_1) \cup \phi$
$= \{q_1, q_2\}$; new state.

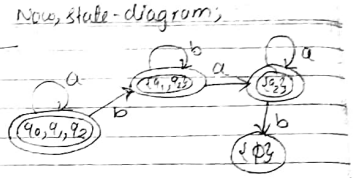
Step 2: $\delta'(\{q_1, q_2\}, a) \Rightarrow \phi \cup E(q_2)$
$\Rightarrow \{q_2\}$, new state

$\delta'(\{q_1, q_2\}, b) \Rightarrow E(q_1) \cup \phi$
$= \{q_1, q_2\}$

Step 3: $\delta'(\{q_2\}, a) \Rightarrow E(q_2) \Rightarrow \{q_2\}$
$\delta'(\{q_2\}, b) \Rightarrow \phi$

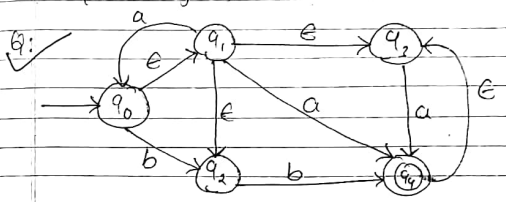
Now, state-diagram;

Q	a	b
$\{q_0, q_1, q_2\}$	$\{q_0, q_1, q_2\}$	$\{q_1, q_2\}$
$\{q_1, q_2\}$	$\{q_2\}$	$\{q_1, q_2\}$
$\{q_2\}$	$\{q_2\}$	ϕ

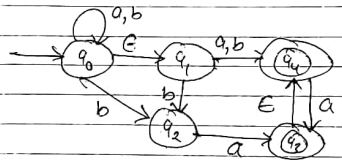


~~Now, state-diagram;~~

Q:



Q:



REGULAR EXPRESSION AND LANGUAGES:-

→ the languages accepted by finite automata is called regular language.

→ the algebric expression that represent regu languages are called regular expression.

→ Let Σ be an alphabet, the class of regular language over Σ is defined inductively
(i) ϕ is regular language representing emp language.
(ii) $\{\epsilon\}$ is regular language representing la of empty string.
(iii) for each $a \in \Sigma$, $\{a\}$ is a regular long.
(iv) if $L_1, L_2 \dots L_n$ are regular languages then is $L_1 \cup L_2 \cup \dots L_n$.
(v) if $L_1, L_2 \dots L_n$ are regular languages then s $L_1, L_2 \dots L_n$
(vi) if L is a regular language then so is L

Application of Regular Language:-

(i) Validation:-
-determining that a string complie with a set of formatting constraints like email validation, password validation, etc.

(ii) Search and Selection:-
-identifying a subset of items from larger set on the basis of a pattern match.

(iii) Tokenization:-
-converting a sequence of characters words, tokens for later interpretation.

Algebric Laws for Regular Expression:-

(i) Commutativity:-
-if r and s are regular expressions representing languages $L(r)$ and $L(s)$ then,
$r \cup s = s \cup r$
$r.s \neq s.r$

(ii) Associativity:-
-if r, s and t are regular expressions representing languages $L(r), L(s), L(t)$ then
$r \cup (s \cup t) = (r \cup s) \cup t$
$r.(s.t) = (r.s).t$

(iii) Distributive:-
- $r(s \cup t) = rs \cup rt \rightarrow$ left distribution
- $(s \cup t)r = sr \cup tr \rightarrow$ right distribution

(iv) Identity:-
$r \cup \phi = \phi \cup r = r$
$r\epsilon = \epsilon . r = r$

(v) Annihilator:-
$\phi . r = r . \phi = \phi$

(vi) Idempotent:-
$r \cup r = r$

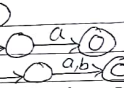
FINITE AUTOMATA AND REGULAR EXPRESSION:-

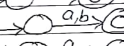
→ We can convert each of regular expressio into finite automata.

⇒ also, from given finite automata we can f out regular expressions.

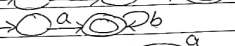
→ From R.E. to F.A.

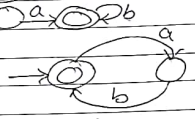
(i) $\phi \Rightarrow$ 

(ii) $a \Rightarrow$ 

(iii) $a+b \Rightarrow$ 

(iv) $ab \Rightarrow$ 

(v) $ab^* \Rightarrow$ 

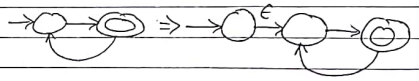
(vi) $(ab)^* \Rightarrow$ 

(vii) $(ab+ba)^* \Rightarrow$ 

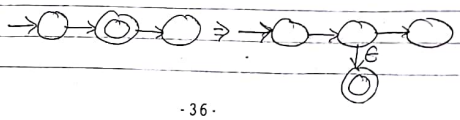
→ From F.A. to R.E.

(i) Using State Elimination Method:-

a: initial state should not have incoming part:-

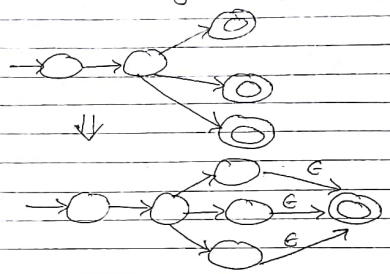


b: final state should not have outgoing par



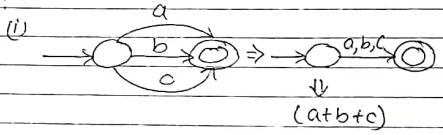
( ) ⇒ Union
⇒ ∪
ie (a+b)C
a+b ∪C

c: should have only one final state:-

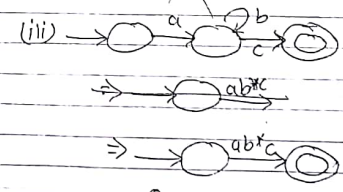


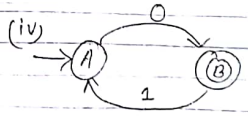
d: other than initial & final state, eliminate other state one by one.

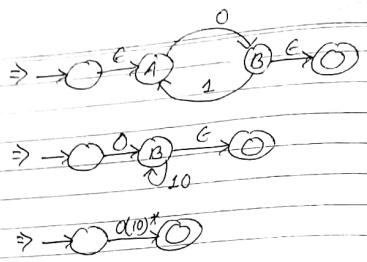
e.g:

(i) 

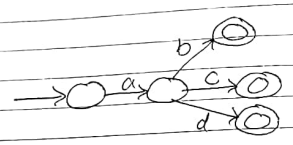
(ii) 

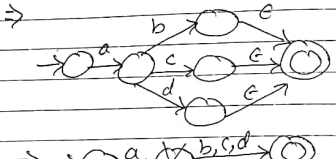
(iii) 

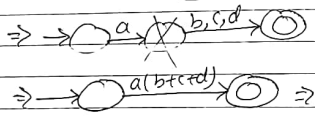
(iv) 



(v)

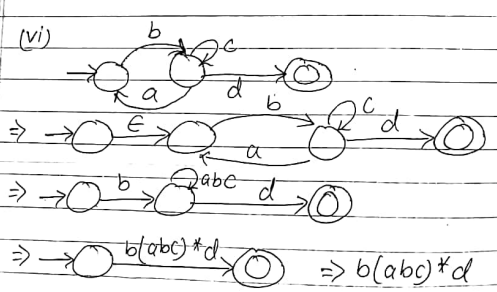






$\Rightarrow a(b+c+d)$

(vi)



$\Rightarrow b(abc)^*d$

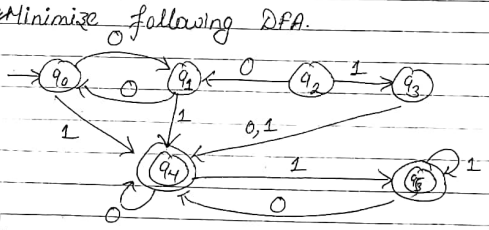
STATE MINIMIZATION

It is the process of minimizing the given DFA by removing unreachable states and by merging similar equivalent states.

Steps:-

1: Remove unreachable states from given DFA.
2: Construct a transition table for rest of the states.
3: Divide the transition table of step 2 into two parts as:-
   a: rows that start with final states.
   b: rows that start with non-final states.
4: Eliminate equivalent/similar states from table 3(a) & 3(b) separately.
5: Repeat step 4: until equivalent states are appeared.
6: Combine/merge updated tables.
7: Draw minimized diagram.

eg: Minimize following DFA.



sol^n

a: here q_2 & q_5 are unreachable states, so remove them.

b:

b: transition table for rest of states:-

Q	0	1
q_0	q_1	q_4
q_1	q_0	q_4
q_4	q_4	q_5
q_5	q_4	q_5

c: non-final

Q	0	1
q_0	q_1	q_4
q_1	q_0	q_4

Final

Q	0	1
q_4	q_4	q_5
q_5	q_4	q_5

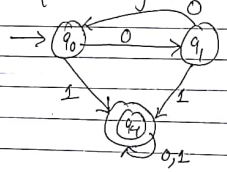
d:

Q	0	1
q_4	q_4	q_4

e:

Q	0	1
q_0	q_1	q_4
q_1	q_0	q_4
q_4	q_4	q_4

f: state-diagram:-

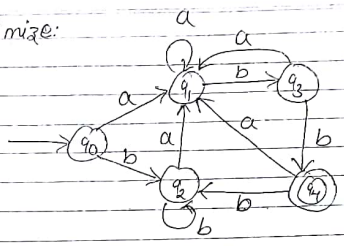


* 001001

$\Rightarrow \delta(q_0, 001001)$
$\vdash (q_1, 01001)$
$\vdash (q_0, 1001)$
$\vdash (q_4, 001)$
$\vdash (q_4, 01)$
$\vdash (q_4, 1)$
$\vdash (q_4, \epsilon)$

$\therefore q_4$ is in final state
so, DFA accepts.

• Q: Minimize:



Soln

a: Transition table:-

Q	a	b
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

b: non-final

Q	a	b
q_0	q_1	q_2
q_1	q_1	q_3
q_2	q_1	q_2
q_3	q_1	q_4

Final

Q	a	b
q_4	q_1	q_2

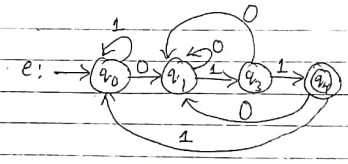
c:

Q	a	b
q_0	q_1	q_0

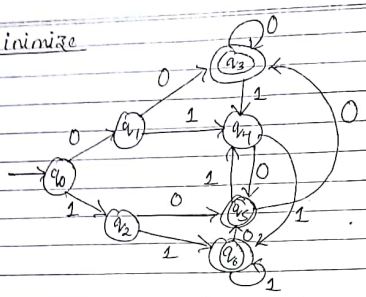
d:

Q	0	1
q_0	q_1	q_0
q_1	q_1	q_3
q_3	q_1	q_4
q_4	q_1	q_0

e:



Q: Minimize



⇒ a: Transition Table:-

Q	0	1
q_0	q_1	q_2
q_1	q_3	q_4
q_2	q_5	q_6
q_3	q_3	q_4
q_4	q_5	q_6
q_5	q_3	q_4
q_6	q_5	q_6

b: non-final

Q	0	1
q_0	q_1	q_2
q_1	q_3	q_4
q_2	q_5	q_6
q_4	q_5	q_6

final

Q	0	1
q_3	q_3	q_4
q_5	q_3	q_4
q_6	q_5	q_6

$q_4 = q_2$ $\quad$ $q_3 = q_5$

c: non-final

Q	0	1
q_0	q_1	q_2
q_1	q_3	q_2
q_2	q_5	q_6

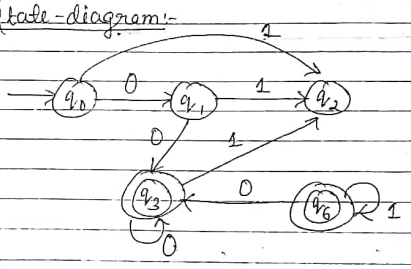
final

Q	0	1
q_3	q_3	q_4
q_6	q_3	q_6

d:

Q	0	1
q_0	q_1	q_2
q_1	q_3	q_2
q_2	q_3	q_6
q_3	q_3	q_2
q_6	q_3	q_6

e: State-diagram:-



1: PROOF BY CONTRADICTION

- establishes the truth or validity of proposition showing that the proposition's being false would imply a contradiction.

2: PIGEON HOLE PRINCIPLE

- if n items are put into ~~or~~ m containers, with $n > m$, then at least one container must contain more than one item.
- if there are n pigeon holes and n+1 pi then at least one pigeon hole is occupied two or more pigeons.

-Mathematically,

Let A & B are two finite sets with |A| then one to one function can't exists.

PUMPING LEMMA FOR REGULAR LANGUAGE:

→ informally, it says that all sufficient long words in a regular language may be pump i.e. have a middle section of word repeated arbitrary no. of times to produce new word lies with in same language.

→ Pumping lemma is powerful technique to s that language is not regular.

Statement:-

- Let L be a regular language.
- w be any string. $w \in L$ with $|w| \geq n$; n a integer con
- There are strings x, y, z such that

$w = xyz$ where $|xy| \leq n$

$|y| > 0$

Then,

$xy^iz \in L$ for all $i \geq 0$.

here,

y is substring that can be pumped.

PROOF:-

→ Suppose L is a regular language, then L is accepted by some DFA 'M'.

→ Let M has n states.

→ Also, L is ∞, so M accepts some string w of length n or greater.

→ Let, length of w, $|w| = m$ where $m \geq n$.

Here, $w = a_1, a_2, \ldots a_m$ where $m \geq n$, each $a_i \in \Sigma$ be an input symbol to M.

For, $j = 1 \ldots n$, q_j be states of M.

Then,

$\delta(q_0, w) = \delta(q_0, q_1, q_2 \ldots q_n$

$= \delta$

$\delta(q_0, w) = \delta(q_0, a_1, a_2, \ldots a_m)$

$= \delta(q_1, a_2 \ldots a_m)$

$= \ldots$

$= \ldots$

$= \delta(q_n, \epsilon)$

Since,

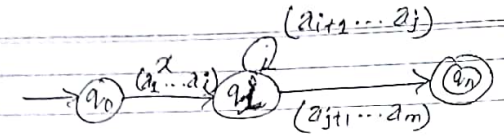
$m \geq n$ and DFA M has only n states, by pigeonhole principle, there exists some i & j with $0 \leq i < j \leq n$, such that $q_i = q_j$.

We can break, $w = xyz$ as follows,

$x = a_1, a_2, \ldots a_i$

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

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



i.e: string $a_{i+1} \cdots a_j$ takes M from state q_i back to itself since $q_i = q_j$

so, M accepts $(a_1, a_2 \cdots a_i)(a_{i+1} \cdots a_j)^k (a_{j+1} \cdots a_m)$ for all $k \geq 0$

$\therefore xy^kz \in L$ for all $k \geq 0$.

V.V.Imp

e.g: Show that $L = \{a^i b^i, i \geq 1\}$ is not regular using pumping lemma for regular language.

Proof:-

→ Let L be any regular language

→ n be a pumping constant.

→ w be any string; $w \in L$ with $|w| \geq n$

→ w can be rewritten as

$w = xyz$ such that

$|xy| \leq n$

$|y| \geq 0$

then;

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

For pumping lemma we can write

$w = xyz = a^n b^n$

it can be written as;

$w = xyz = \underbrace{aa\cdots a}_{x}\underbrace{aa\cdots a}_{y}\underbrace{ab\cdots bb\cdots bb}_{z}$

here,

$y = a^k$; $k \geq 1$

Now, from pumping lemma

for $i = 0$;

$w = xz = a^{n-k} b^n \notin L$

for $i = 2$;

$w = xy^2z = xyyz = a^n a^k b^n$

$= a^{n+k} b^n \notin L$

Here is contradiction with theorem;

so, given language

$L = \{a^i b^i; i \geq 1\}$ is not regular language.

DECISION PROPERTIES OF REGULAR LANGUAGE.

→ Here, we consider some of the fundamental question about languages.

1: Is the language described empty?

2: Is a particular string w in described language?

3: Do two descriptions of a language actually describe the same language?

→ we can answer the above questions as follows:-

1: Testing emptiness of regular language

→ If our representation is any kind of finite automata, the emptiness question is whether there is any path whatsoever from start state to some accepting state.

→ If yes then language is non-empty;

→ If the accepting states are separated from start state, then language is empty.

2: Testing membership in a regular language:-

→ Let L be any regular language and w is any string. We have to test whether $w \in L$

or not.
→ We can represent L by some finite aut
Then simulate the automata processing the
string of input symbols w, begining in the
state.
→ if automata ends in accepting state, th
answer is yes otherwise no.

3: Converting among representation:-
(i) Convert NFA's to DFA and vice-versa.
(ii) Convert automata to regular expression &
versa.

CLOSURE PROPERTIES OF REGULAR LANGUAGE:

- The class of language accepted by fini
automata is closed under
(i) Union
(ii) Concatenation
(iii) Kleene star

(i) Union

→ Let A_1 and A_2 are two regular langua
now prove $A_1 \cup A_2$ is regular.
→ Here, idea is, take two NFA's i.e. N_1 & N_2 f
A_1 and A_2 and combine them to one New
N; such that it accepts input i.e. either acce
by N_1 or N_2.

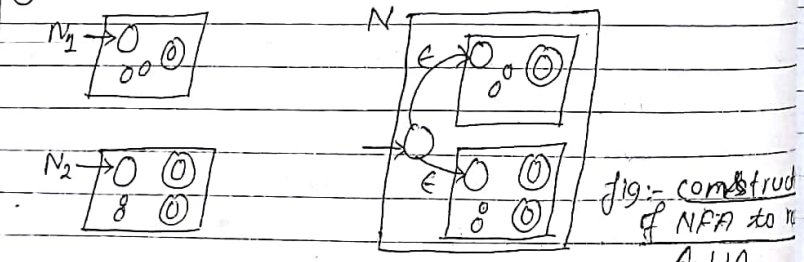


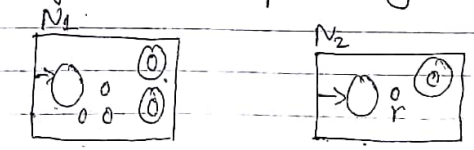
fig:- construc
of NFA to
$A_1 \cup A_2$

Let,
$N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ recognizes A_1
$N_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ recognizes A_2
Now,
$N = (Q, \Sigma, \delta, q_0, F)$ is such that $L(N) = A_1 \cup A_2$
here,
$Q = Q_1 \cup Q_2 \cup \{q_0\}$
q_0 = start state of N
$F = F_1 \cup F_2$

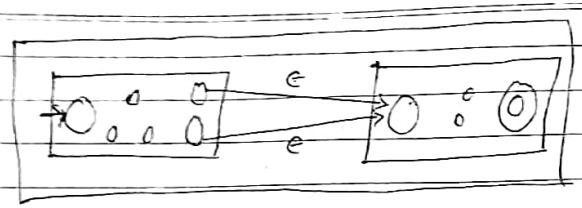
$$\delta(q,a) = \begin{cases} \delta_1(q,a) & q \in Q_1 \\ \delta_2(q,a) & q \in Q_2 \\ \{q_1, q_2\} & q = q_0 \text{ and } q = \epsilon \\ \phi & q = q_0 \text{ and } q \neq \epsilon \end{cases}$$

(ii) Concatenation

→ Let A_1 and A_2 are two regular language, now prove $A_1 - A_2$ is regular.
→ Here, idea is, take two NFA's N_1 and N_2 for A_1 and A_2 and combine them to one New NFA N;
⇒ → Assign N's starting state to be start state of N_1.
MAIN ⇒ → The accept states of N_1 have additional ϵ arrows that non-deterministically allow branching to N_2 whenever N_1 is in accept state.
⇒ → The accept states of N are that of N_2 only.
→ So, N accepts when input is split into two parts where first is accepted by N_1 and second by N_2.



N



Let

$N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ recognizes A_1

$N_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ recognizes A_2

Now,

$N = (Q, \Sigma, \delta, q_1, F_2)$ is such that $L(N) = A_1 \cdot A_2$

here,

$Q = Q_1 \cup Q_2$

q_1 = start state of N same as start state of N_1

F_2 = accept state of N same as accept state of N_2.

$$\delta(q,a) = \begin{cases} \delta_1(q,a) & q \in Q_1 \text{ and } q \notin F_1 \\ \delta_1(q,a) & q \in F_1 \text{ and } a \neq \epsilon \\ \delta_1(q,a) \cup \{q_1\} & q \in F_1 \text{ and } a = \epsilon \\ \delta_2(q,a) & q \in Q_2 \end{cases}$$

* Kleene star

→ For regular language A_1, prove A_1^* is a regular.

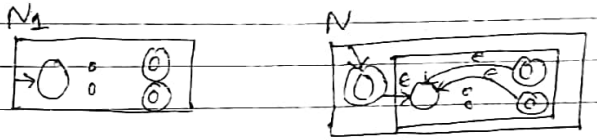
→ take an existing NFA N_1 for A_1 and make a new N for A_1^*.

→ N will accept its input whenever it can broken into several pieces and N_1 accepts each piece.

→ here, construct N like N_1 with additional ϵ arrow returning to start state from accept state

→ Modify N so that it accepts ϵ, which is not any is member of A_1^* so, here add new start state which is also accept state and has ϵ arrow to old start state.

N_1 N



Let, $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ recognizes A_1

Now,

$N = (Q, \Sigma, \delta, q_0, F)$ recognizes A_1^*

here,

$Q = Q_1 \cup \{q_0\}$

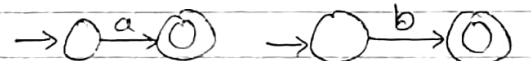
q_0 = start state of N

$F = F_1 \cup \{q_0\}$

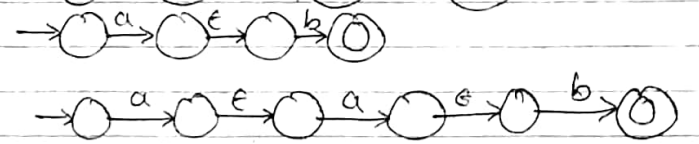
$$\delta(q,a) = \begin{cases} \delta_1(q,a) & q \in Q_1 \text{ and } q \notin F_1 \\ \delta_1(q,a) & q \in F_1 \text{ and } a \neq \epsilon \\ \delta_1(q,a) \cup \{q_1\} & q \in F_1 \text{ and } a = \epsilon \\ \{q_1\} & q = q_0 \text{ and } a = \epsilon \\ \phi & q = q_0 \text{ and } a \neq \epsilon \end{cases}$$

Q: Construct a NFA for reg. expression $(abUaab)^*$

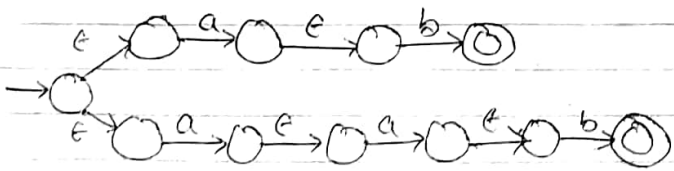
⇒ a : b



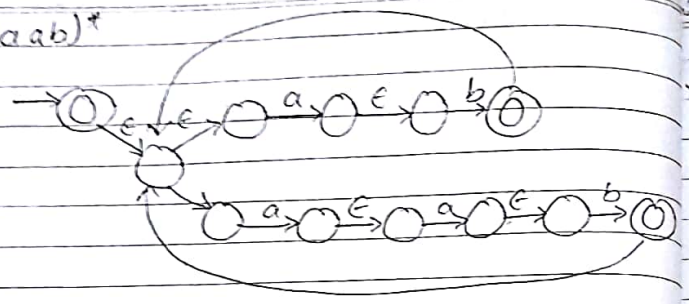
ab : aab



abUaab



$(ab \cup aab)^*$



Chapter-3: Context Free Language

CONTEXT FREE LANGUAGE GRAMMAR:-

P → a more powerful method of describing language.
E → it can describe certain features that have a recursive structure.
A → an important application of context free grammar occurs in specification and compilation of programming languages.
C → the collection of languages associated with context free grammars are called CFL.
G → CFL → Context free grammar is a language generator on some set of rules.

e.g: $A \rightarrow 0A1$
$A \rightarrow B$
$B \rightarrow \#$

here,
→ grammar consists of collection of substitution rules called production.
→ each rule comprise a symbol and string separated by arrow.
→ symbol is called variable.
→ strings that consists of variables and other symbols are called terminals.
→ one variable is a ~~short~~ start variable that usually occurs on the left handside of the topmost rule.
→ For given e.g; grammar contains 3 rules. Variables A & B. It is a start variable. Its terminals are 0, 1, #.
→ here above grammar can generate string 000#111 as follows:-

$A \to 0A1$
$00A11$
$000A111$
$000B111$
$000\#111$

[from: $A \to 0A1$, $A \to B$, $B \to \#$]

→We can abbreviate several rules with left hand variable. like $A \to 0A1 | B$.

Formal definition:- a context free gramma a four tuple (V, Σ, R, S); where;
V= is a finite set called variable
Σ= is a finite set, disjoint from V, called t
R= is a finite set of rules.
S= is a start variable as $S \in V$.

Some terminologies:-

(i) Derivation:-
→ Let $G = (V, \Sigma, R, S)$ be a context free [illegible] $w_1, w_2, \ldots w_n$ are strings over variable V such that $w_1 \Rightarrow w_2 \Rightarrow \ldots \Rightarrow w_n$ then say w_n is derivable.
→The sequence of substitution to obt a string is called derivation. Say w_i deri w_n i.e: $w_i \overset{*}{\Rightarrow} w_n$. Then, the sequence of ste to obtain w_n from w_i is called derivatio

(ii) Language of CFG ($L(G)$):-
If $G = (V, \Sigma, R, S)$ is a CFG, then the l of G denoted by $L(G)$ is the set of termin strings that have derivations from start i.e: $L(G) = \{ w \in \Sigma^* : S \overset{*}{\Rightarrow} w \}$

(iii) Sentential form:-
→derivations from the start symbol produce strings that have a special rule. This is called sentential form.
i.e: if $G = (V, \Sigma, R, S)$ is a CFG then any string α is in $((V-\Sigma) \cup \Sigma)^*$ such that $S \overset{*}{\Rightarrow} \alpha$ is a sentential form.

if $S \underset{lm}{\overset{*}{\Rightarrow}} \alpha$, then α is a left sentential form.

if $S \underset{rm}{\overset{*}{\Rightarrow}} \alpha$, then α is a right sentential form

e.g:-
$E \to I$
$E \to E+E | E*E | (E)$
$I \to a | b | Ia | Ib | I0 | I1$

Now,
$E*(I+E)$ is a sentential form as there is a derivation.
$E \Rightarrow E*E$
$\Rightarrow E*(E)$
$\Rightarrow E*(E+E)$
$\Rightarrow E*(I+E)$

(i) Consider $a*E$
$E \underset{lm}{\Rightarrow} E*E$
$\underset{lm}{\Rightarrow} I*E$
$\underset{lm}{\Rightarrow} a*E$
∴ $a*E$ is a left sentential form.

(ii) Consider E*(E+E)

$$E \underset{rm}{\Rightarrow} E*E$$
$$\underset{rm}{\Rightarrow} E*(E)$$
$$\underset{rm}{\Rightarrow} E*(E+E)$$

∴ E*(E+E) is right sentential.

Q.1 Let $G=(V,\Sigma,R,S)$ where
$V=\{S\}$
$\Sigma=\{a,b\}$
$R=\{S\to aSb$
$S\to ab\}$ now find $L(G)$

Soln

(i) $S\to ab$
(ii) $S\to asb$
$\to aabb$
(iii) $S\to asb$
$\to aasbb$
$\to aaasbbb$
⋮
$\to a^{n-1}sb^{n-1}$
$\to a^{n-1}abb^{n-1}$
$\to a^nb^n$

hence, language of given grammar is
$L(G)=\{a^nb^n : n\geq 1\}$

Q.2 Let $G=(V,\Sigma,R,S)$ where
$V=\{S\}$
$\Sigma=\{a,b\}$
$R=\{S\to aaS$
$S\to b\}$ now find $L(G)$.

Soln using given rules recursively;

(i) $S\to b$
(ii) $S\to aaS$
$\to aab$
(iii) $S\to aaS$
$\to aaaaSb$
$\to (aa)^2b$
(iv) $S\to aaS$
$\to aaaaS$
⋮
$\to (aa)^nb$

hence, language of given grammar is
$L=\{(aa)^nb : n\geq 0\}$

Q.3 Let $G=(V,\Sigma,R,S)$ where
$V=\{S,A\}$
$\Sigma=\{a,b\}$
$R=\{S\to aA\mid \epsilon$
$A\to bS\}$
Now find $L(G)$.

Soln

Using given rules recursively,

(i) $S\to\epsilon$
(ii) $S\to aA$
$\to abS$
$\to ab\epsilon$
$\to ab$
(iii) $S\to aA$
$\to abS$
$\to abaA$
$\to ababS$
$\to abab\epsilon$
$\to abab$
$\to (ab)^2$

(iv) $S \to aA$
$\to aba$
$\to aAbabs$
$\to aabaabs$
$\to (ab)n$

Hence, language of given grammar is $L(G) = \{(ab)^n : n \geq 0\}$ (Equivalent to given)

Q: Write a CFG for the language given by $L = \{a^m b^n : m \geq n\}$ = $\{a, ab, aab, aaab, aaaab$

Sol^n

Let, $G = (V, \Sigma, R, S)$ be required CFG. where

$V = \{S\}$
$\Sigma = \{a, b, \epsilon\}$
$R = \{S \to aSb$
$S \to \epsilon$
$S \to aS\}$

formula

$S \to \epsilon$ if question is no. then b

$S \to aS$ if question is a must br

$S = aSb$

$aaaab$
$S \to aSb$
$\to aaSb$
$\to aaaSb$
$\to aaaa\epsilon b$
$\to aaaab$

Q: Write a CFG for the language given by $L = \{a^m b^n : m \geq 2, n \geq 1\}$

Sol^n

Let, $G = (V, \Sigma, R, S)$ be required CFG where,

$V = \{S, A, B\}$
$\Sigma = \{a, b\}$
$R = \{S \to AB$
$A \to aA$
$A \to a$
$B \to bB$
$B \to b\}$

$S = \{SB$

$aabbb$
$S \to AB$
$aABB$
$aabBB$
$aabbb$

Q: Write

Q: Write a CFG for the language given by;

$L = \{w \in \{a,b\}^* : \text{there is equal no. of a and b}\}$

⇒ Let $G = (V, \Sigma, R, S)$ be required CFG.

where,

$V = \{S\}$

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

$R = \{S \to asb \quad S \to bsa \quad S \to \in \quad S \to sa \quad S \to sb\}$ ~~(crossed out)~~

$S = \{S\}$

use this:

s -> aSbS / bSaS / empty

abaababb

$S \to asb$
$\to asbb$
$\to asabb$
$\to asbabb \to abbabbabb$
$\to asaabbabb \to abaabbabb$
$\to asbaabbabb \to abaabbabb$
$\to a\in baabbabb \to abaabbabb$
$\to abaabbabb$

Derivative Tree / Parse Tree

→ is a tree representation of strings of terminals using the productions difinid by the grammar.

→ it pictorically shows how the start symbol of a grammar derives a strings in language.

→ is an ordered tree that shows the essentials of a derivation.

→ formally, given CFG $G = (V, \Sigma, R, S)$, a parse tree has following properties:-

(i) the root is labelled by start symbol.

(ii) each interior nodes of parse tree are variables.

(iii) each leaf node is labelled by terminal symbol or $\in$.

(iv) if an interior node is labelled with non-terminal A and its children are $X_1, X_2, \ldots X_n$ from left to right then there is a production P as:

$A \to X_1 X_2 \cdots X_n$ for each $X_i \in \Sigma$.

e.g: consider a grammar where

$S \to asa \mid a \mid b \mid \in$

Now,

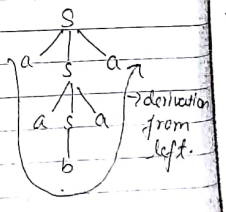
for string aabaa

we have,

$S \to asa$
$\to aasaa$
$\to aabaa$

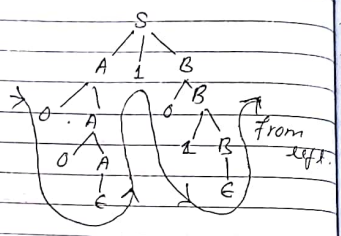
Now,

Its parse tree is;



Q: Consider the grammar is
$S \rightarrow A1B$
$A \rightarrow 0A \mid \epsilon$
$B \rightarrow 0B \mid 1B \mid \epsilon$
Construct parse tree for (i) 00101
(ii) 1001
(iii) 00011

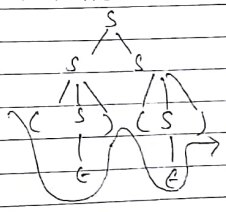
(i) $\Rightarrow$ 00101
$S \rightarrow A1B$
$\rightarrow 0A1B$
$\rightarrow 00A1B$
$\rightarrow 00A10B$
$\rightarrow 00\epsilon 10B$
$\rightarrow 00101B$
$\rightarrow 00101\epsilon$
$\rightarrow 00101$



Q: Let $G = (V, \Sigma, R, S)$ where
$V = \{S\}$, $\Sigma = \{(,)\}$
$R = \{ S \rightarrow SS$
$\rightarrow (S)$
$\rightarrow \epsilon$
$\}$

Now,
~~parse tree~~ ( ) ( )
$S \rightarrow SS$
$\rightarrow (S)S$
$\rightarrow (S)(S)$
$\rightarrow (\epsilon)(S)$
$\rightarrow ()(\epsilon)$
$\rightarrow ()()$

Parse tree:-



Leftmost and Rightmost Derivation:-

→ In leftmost derivation, leftmost variable is replaced first.

→ In rightmost derivation, rightmost variable is replaced first.

* Consider a grammar:-

$S \rightarrow S+S$ ~~A10~~

$S \rightarrow S/S$

$S \rightarrow (S)$

$S \rightarrow S-S$

$S \rightarrow S*S$

$S \rightarrow a$

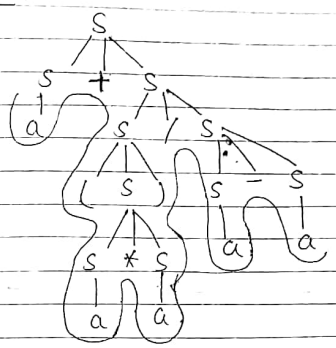
Now, derive string

a+(a*a)/a-a with L.M.D. & draw parse tree

~~tt~~ also " R.M.D. & " "

Leftmost derivation:-	Rightmost derivation:-
$S \rightarrow S+S$	$S \rightarrow S+S$
$\rightarrow a+S$	$\rightarrow S+S/S$
$\rightarrow a+S/S$	$\rightarrow S+S/S-S$
$\rightarrow a+(S)/S$	$\rightarrow S+S/S-a$
$\rightarrow a+(S*S)/S$	$\rightarrow S+S/a-a$
$\rightarrow a+(a*S)/S$	$\rightarrow S+(S)/a-a$
$\rightarrow a+(a*a)/S$	$\rightarrow S+(S*S)/a-a$
$\rightarrow a+(a*a)/S-S$	$\rightarrow S+(S*a)/a-a$
$\rightarrow a+(a*a)/a-S$	$\rightarrow S+(a*a)/a-a$
$\rightarrow a+(a*a)/a-a$	$\rightarrow a+(a*a)/a-a$

Left-most :-



Right-most:-

Ambiguous Grammar:-

→ A grammar $G=(V,\Sigma,R,S)$ is said to be ambiguous if there is a string $w \in L(G)$ for which we can derive two or more distinct derivation tree; rooted at S and yielding w.

→ In other words, a grammar is ambiguous if it can produce more than one leftmost or more than one rightmost derivation for same string in language of grammar.

e.g:

$S \to AB/aaB$
$A \to a/Aa$
$B \to b$

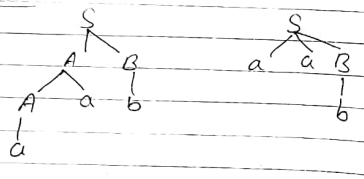
Now, for string aab

we have two leftmost derivation as;

① $S \Rightarrow AB$
$\Rightarrow AaB$
$\Rightarrow aaB$
$\Rightarrow aab$

② $S \Rightarrow aaB$
$\Rightarrow aab$

Parse tree:-



Simplification of CFG:-

- (i) We must eliminate useless symbols.
- (ii) We must eliminate Empty (ε) production.
- (iii) " " " unit production.

(i) Eliminating Useless Symbols:-

→ Useless symbols are those variables or terminals that do not appear in any derivation of a terminal string from the start symbol.

→ For a grammar $G=(V,\Sigma,R,S)$, a symbol 'x' is useful, if there is some derivation of the form $S \overset{*}{\Rightarrow} \alpha x \beta$.

→ Elimination of useless symbols include identifying whether or not the symbol is 'generating' and 'reachable'.

→ a symbol x is generating if $x \overset{*}{\Rightarrow} w$ for some terminal string w.

→ a symbol x is reachable if there is derivation $S \overset{*}{\Rightarrow} \alpha x \beta$ for some α and β.

→ Thus we eliminate non-generating and then non reachable symbols.

e.g: Consider a Grammar:-

$S \to aB/bX$
$A \to Bad/bsX/a$
$B \to aSB/bBX$
$X \to SBd/aBX/ad$

here, A and X can generate terminals so A and X are generating symbols as $A \to a$ and $X \to ad$

-also S is generating symbol as S can generate terminal strings such as $S \to bX$ and x generate terminal strings.

-But, B ~~do~~ can't produce any terminal symbol so it is non generating.

∴ after eliminating it;

$S \to bX$
$A \to bsX/a$
$X \to ad$

Because S ⇏ (doesn't come)

Again,
A is unreachable as there is no any derivation of form $S \overset{*}{\Rightarrow} \alpha A \beta$ in grammar.
∴ After eliminating it
$S \to bX$
$X \to ad$
∴ This is simplified grammar

✓ (ii) Eliminating Empty (ϵ) production:-
→ A grammar is said to have empty production if there is a production of the form $A \to \epsilon$
→ here, we have to discover nullable variable
→ a variable 'A' is nullable if $A \overset{*}{\Rightarrow} \epsilon$
→ if $A \to \epsilon$ is a production to be eliminated then we look for all productions whose right side contain A and replace each occurence of A in each of those production to obtain non empty productions
→ Now, those resultant non empty productions must be added to grammar to keep the language generated to be same.

e.g: $S \to aA$
$A \to b \mid \epsilon$

Sol^n
here, $A \to \epsilon$ ∴ A is nullable
→ Then, find all production whose right side contains A. $S \to aA$
→ replace A with ϵ
→ we get $S \to a$
→ finally, we get,
$S \to aA$
$A \to b$ or, $S \to aA \mid a$
$S \to a$ $A \to b$

e.g: $S \to ABAC$
$A \to aA \mid \epsilon$
$B \to bB \mid \epsilon$
$C \to c$

Sol^n
Here, we have two ϵ productions to remove
$A \to \epsilon$
$B \to \epsilon$
∴ A and B are both nullable.

To eliminate $A \to \epsilon$
Productions which contain A in right side are; $S \to ABAC$
$A \to aA$
here;
(i) $S \to ABAC$
⇓
$S \to BAC \mid ABC \mid BC$
(ii) $A \to aA$
⇓
$A \to a$

Now, we get;
$S \to ABAC \mid BAC \mid ABC \mid BC$
$A \to aA \mid a$
$B \to bB \mid \epsilon$
$C \to c$

To eliminate $B \to \epsilon$:
productions which contain B at right side are (i) $S \to ABAC$ (ii) $S \to BAC$ (iii) $S \to ABC$
⇓ ⇓ ⇓
$S \to AAC$ $S \to AC$ $S \to AC$

(iv) $S \to BC$ ⇓ $S \to C$ (v) $B \to bB$ ⇓ $B \to b$

finally, we get $S \to ABAC|BAC|ABC|BC|AAC|AC|C$
$A \to aA|a$
$B \to bB|b$
$C \to C$

(iii) Eliminating Unit productions:-
- a unit production is a production of the form $A \to B$, where A and B are both variables.

Algorithm:-
while (there exists unit production $A \to B$)
{
(i) Select a unit production $A \to B$, such that there exists a production $B \to \alpha$, where α is terminal.
(ii) for (every non-unit production, $B \to \alpha$)
- add production $A \to \alpha$ to grammar
iii) eliminate $A \to B$ from grammar.
}

e.g: $A \to B$
$B \to a$
$B \to c$

sol^n

$A \to a$
$A \to c$
$B \to a$
$B \to c$

∵ $A \to B$
$B \to a$ } $B \to \alpha$ } $\to A \to a$
$B \to c$ } $B \to \alpha$ } $A \to c$

e.g:
$S \to AB$
$A \to a$
$B \to C|b$
$C \to D$
$D \to E$
$E \to a$

sol^n
here, we have 3 unit productions,
$B \to C$, $C \to D$, $D \to E$
for $B \to C$, we can't remove this ∵ No $C \to \alpha$
" $C \to D$, " " " " ∵ No $D \to \alpha$,
" $D \to E$, " can " " ∵ we have $E \to a$
So,
$S \to AB$
$A \to a$
$B \to C|b$
$C \to D$
$D \to a$
$E \to a$
Now, for $C \to D$, we have $D \to a$
∴ We get,
$S \to AB$
$A \to a$
$B \to C|b$
$C \to a$
$D \to a$
$E \to a$
Similarly, for $B \to C$, we have $C \to a$
∴ Remaining $B \to C$ and
adding $C \to a$ we get,

S → AB
A → a
B → a
B → b
C → a
D → a
E → a

⇒ S → AB
A → a
B → a
B → b

Chomsky Normal Form (CNF):-

→ a CFG $G=(V,\Sigma,R,S)$ is said to be in Chomsky Normal Form (CNF) if every production in G are in one of two forms.

$A \to BC$

$A \to a$ where $A,B,C \in V$ and $a \in \Sigma$

→ So, a grammar in CNF is one which shouldnot have:

(i) ϵ production
(ii) Unit ,,
(iii) Useless symbols.

e.g: convert the following CFG into CNF

$G=(V,\Sigma,R,S)$ where

$V=\{S,A\}$

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

$R=\{$
$S \to aAB|AaB|B$
$A \to aA|\epsilon$
$B \to ab|bA$
$\}$

Soln

- To convert CFG into CNF, we have to simplify CFG.
- in given grammar, there are not any unit

symbols.
- Now, eliminating ϵ production;

$S \to aAB|AaB|aA|B$
$A \to aA|a$
$B \to ab|bA|b$

→ Again, eliminating Unit production;

$S \to aAB|AaB|aB|ab|bA|b$
$A \to aA|a$
$B \to ab|bA|b$

Now,
converting this simplified form to get CNF.

$S \to CAB|ACB|CB|CD|DA|b$
$C \to a$
$D \to b$
$A \to CA|a$
$B \to CD|DA|b$

here,
again $S \to CAB$ and $S \to ACB$ are not in required form.
So, we can perform,

$S \to EB|FB|CB|CD|DA|b$
$E \to CA$
$F \to AC$
$C \to a$
$D \to b$
$A \to CA|a$
$B \to CD|DA|b$

This is required CNF.

Q: Convert following grammar into CNF.

$S \rightarrow AAC$

$A \rightarrow aAb \mid \epsilon$

$C \rightarrow aC \mid a$

Sol^n

Non-empty;

$A \rightarrow \epsilon$

$S \rightarrow AAC$ $\quad S \rightarrow AC$

$A \rightarrow aAb$ $\quad (S \rightarrow C)$

$(A \rightarrow ab)$

Then,

Greibach Normal Form (GNF):-

→ a grammar $G = (V, \Sigma, R, S)$ is said to be in Greibach Normal Form if all the production of grammar are of the form:-

$A \rightarrow a\alpha$; where a is a terminal and α is string of zero or more variables.

eg: Convert the following CFG into GNF.

$S \rightarrow AB \mid BC$

$A \rightarrow aB \mid bA \mid a$

$B \rightarrow bC \mid cC \mid a$

Sol^n

here production $S \rightarrow AB \mid BC$ is not in required form;

-now applying substitution rule;

$S \rightarrow aBB \mid bAB \mid aB \mid bCa \mid cCa \mid bC$

$A \rightarrow aB \mid bA \mid a$

$B \rightarrow bC \mid cC \mid b$

$C \rightarrow c$

Chapter 4

Push-down Automata (PDA)

→ is automata that defines context free language, ie. it can capture context sensitive grammars.
→ it can be taught as a ε-NFA with addition of stack.
→ so, it remember infinite amount of information.
→ but, it can only access information in stack as last in first-out.

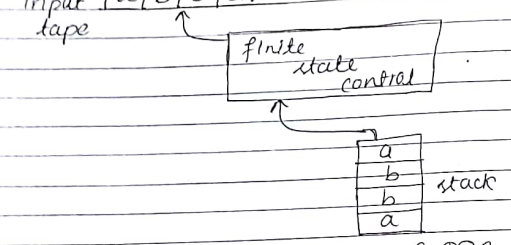


fig:- block-diagram of PDA

→ PDA is abstract machine determined by following three things:-
(i) input
(ii) finite state control
(iii) stack

→ each move of machine is determined by three things:-
(i) current state
(ii) Next input symbol
(iii) symbol on top of stack.

→ The move consists of:-
(a) changing state / stay on same
(b) replace stack top by string of symbols.

→ move of machine contains only one stack operation either push or pop.
→ popping the top symbol of the stack means replacing it by ε.
→ putting Y on the stack means replacing stack top say x by Yx.
→ A PDA can write symbols on the stack and read them back later.

FORMAL DEFINITION OF PDA:-

→ A PDA is defined by seven tuple $(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ where,

$Q \rightarrow$ finite set of states
$\Sigma \rightarrow$ finite set of input symbols
$\Gamma \rightarrow$ finite set of stack alphabets
$q_0 \rightarrow$ start state of PDA, $q_0 \in Q$
$Z_0 \rightarrow$ initial stack symbol, $Z_0 \in \Gamma$
$F \rightarrow$ set of a final states, $F \subseteq Q$
$\delta \rightarrow$ it takes as argument a triple (q, a, x) where,
q is a state
a is either an input symbol in Σ or t.
x is stack symbol.

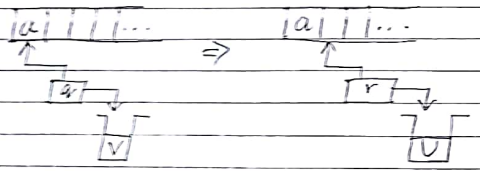
→ the output of δ is a finite set of pairs (p, γ) where p is a new state and γ is the string on stack after transition.

→ transition function δ maps
$\delta : Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \to Q \times \Gamma^*$

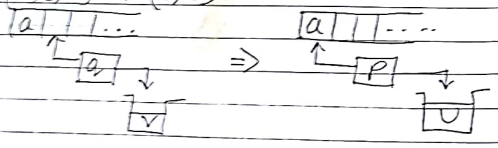
GRAPHICAL NOTATION OF PDA-

→ We can use transition diagram to represent a PDA where
- any state is represented by a node in the graph.
- any arc labelled with start indicates to the state and doubly circled states are accepting or final states
- the arc correspond to transition of PDA and are labelled as a, X/α means the transition $\delta(q, a, X) = (P, \alpha)$
- for arc from state q to P, this arc label tells what input is used and also gives the old and new helps of stack

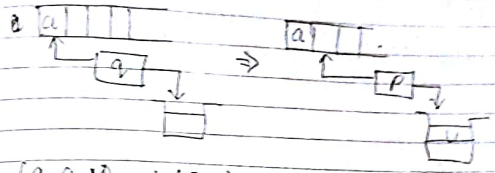
(i) $\delta(q, a, V) \to (P, U)$



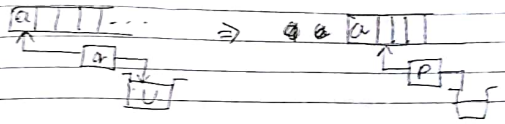
(ii) $\delta(q, \epsilon, V) \to (P, U)$



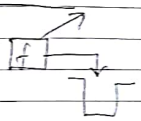
(iii) $\delta(q, a, \epsilon) \to (P, U)$



(iv) $\delta(q, a, V) \to (P, \epsilon)$



(v) X



i.e. when head gets off the tape, PDA stops an input string X is accepted by PDA if PDA stop at final state and stack is empty otherwise input string is rejected.

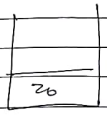
Symbol: stack symbol = push
" " = pop

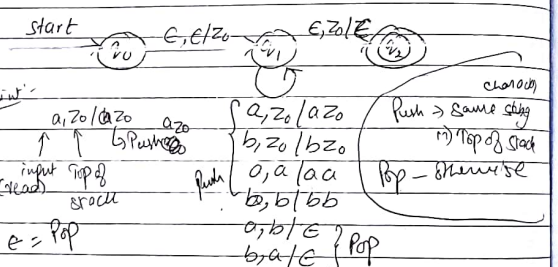
Eg: 1:
- A PDA accepting a string over {a, b} such that numbers of a's and b's are equal.
i.e: $L = \{w \mid w \in \{a,b\}^*$ and a's and b's are equal$\}$

→ Here, the idea is that PDA should push the input symbol if the top of stack symbol is same as it or Z_0 otherwise pop the stack.
→ Let us construct a PDA as:

$$P = \{Q, \Sigma, \Gamma, \delta, q_0, Z_0, F\}$$

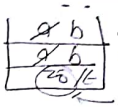
where,
$Q = \{q_0, q_1, q_2\}$
$\Sigma = \{a, b\}$
$\Gamma = \{a, b, Z_0\}$
$q_0 = q_0$
$Z_0 = Z_0$
$F = \{q_2\}$





(i) $\delta(q_0, \epsilon, \epsilon) \Rightarrow (q_1, Z_0)$
(ii) $\delta(q_1, a, Z_0) \Rightarrow (q_1, aZ_0)$
(iii) $\delta(q_1, b, Z_0) \Rightarrow (q_1, bZ_0)$
(iv) $\delta(q_1, a, a) \Rightarrow (q_1, aa)$
(v) $\delta(q_1, b, b) \Rightarrow (q_1, bb)$
(vi) $\delta(q_1, a, b) \Rightarrow (q_1, \epsilon)$
(vii) $\delta(q_1, b, a) \Rightarrow (q_1, \epsilon)$
(viii) $\delta(q_1, \epsilon, Z_0) \Rightarrow (q_2, \epsilon)$

Suppose
aabb



* $w = abbaab$

step	state	string	stack	transition No.
1	q_0	abbaab	ϵ	-
2	q_1	abbaab	Z_0	1
3	q_1	bbaab	aZ_0	2
4	q_1	baab	Z_0	7
5	q_1	aab	bZ_0	3
6	q_1	ab	aZ_0	6
7	q_1	b	aZ_0	2
8	q_1	ϵ	Z_0	7
9	q_2	ϵ	ϵ	8

* $w = ababb$

step	state	string	stack	transition No.
1	q_0	ababb	ϵ	-
2	q_1	ababb	Z_0	1
3	q_1	babb	aZ_0	-
4	q_1	abb	Z_0	-
5	q_1	bb	aZ_0	-
6	q_1	b	Z_0	-
7	q_1	6	bZ_0	-

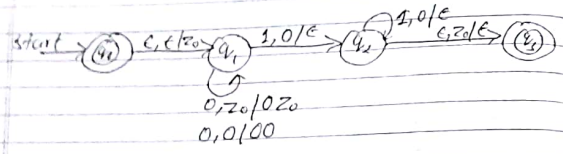
Q: Design a PDA that accepts $L = \{0^n 1^n \mid n \geq 0\}$
Ans/n

Here,
idea is; should push input symbol if it is 0 if it is 1 then pop from stack.
Now,

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

where,
$Q = \{q_0, q_1, q_2, q_3\}$
$\Sigma = \{0, 1\}$
$\Gamma = \{0, 1, Z_0\}$
$q_0 = q_0$
$Z_0 = Z_0$

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



$\delta \Rightarrow$ 1: $\delta(q_0, \epsilon, \epsilon) \Rightarrow (q_1, z_0)$
2: $\delta(q_1, 0, z_0) \Rightarrow (q_1, 0z_0)$
3: $\delta(q_1, 0, 0) \Rightarrow (q_1, 00)$
4: $\delta(q_1, 1, 0) \Rightarrow (q_2, \epsilon)$
5: $\delta(q_2, 1, 0) \Rightarrow (q_2, \epsilon)$
6: $\delta(q_2, \epsilon, z_0) \Rightarrow (q_f, \epsilon)$

$\omega = 000111$

Step	State	string	stack	transition no.
1:	q_0	000111	ϵ	—
2:	q_1	000111	z_0	1
3:	q_1	00111	$0z_0$	2
4:	q_1	0111	$00z_0$	3
5:	q_2	111	$000z_0$	3
6:	q_2	11	$00z_0$	4
7:	q_2	1	$0z_0$	5
8	q_2	ϵ	z_0	5
9	q_f	ϵ	ϵ	6

∴ Given string is accepted.

2017 (D): $L = \{x^i y^j z^k : i, j, k >= 0$ and $i = k$ or $j = k\}$.

Here; $i = k$ or $j = k$. $L = \{x^n y^n z^n : n \geq 0\}$

$i = k = j$ (n suppose)

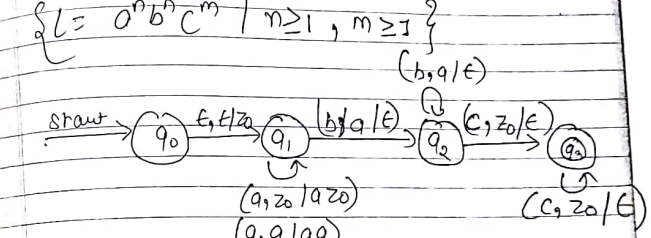
$0101 \Rightarrow$

State	String	Stack	transition no.
q_0	0101	ϵ	—
q_1	0101	z_0	1
q_1	0101	$0z_0$	2
q_2	01	z_0	4

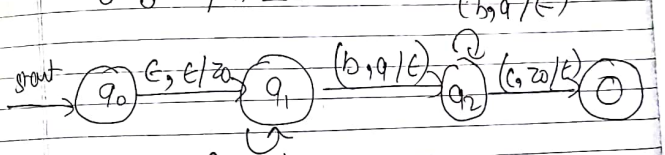
Q: Design a PDA that accepts proper parenthesis.
(i.e. (), ()(), (()), ((())), ...)

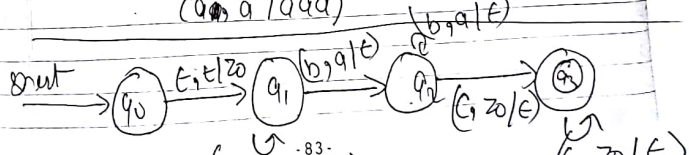
unequal power

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



$a^n b^{2n} \mid n \geq 1$





Instantaneous Description of PDA (ID):-

→ Description of any PDA by a triplet (q, w, γ) where q is the state, w is the remaining input and γ is the stack contents is called as Instantaneous description of PDA (ID).

→ Notation of Instantaneous description for PDA is used as to describe changes in state, input and stack.

→ Let $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ be a PDA.

→ Define a relation $\vdash$

→ Suppose $\delta(q, a, X)$ contains (p, α). Then, for all string w in Σ^* and β in Γ^*

$(q, aw, X\beta) \vdash (p, w, \alpha\beta)$

This reflects the idea that by consuming a from input and replacing X on top of stack by α, we can go from state q to p.

Note: What remains on the input w and what is below the top of stack β do not influence the action of PDA.

→ We can also use symbol $\vdash^*$ when PDA is understood to represent zero or more moves of PDA.

for e.g.:

$(q_0, aabbbaab, Z_0) \vdash^* (q_2, \epsilon, \epsilon)$

→ we can define acceptance of any strings by PDA in two ways:-

(i) acceptance by final state

-given PDA P, the language accepted by final state, $L(P)$ is

$\{w \mid (q, w, z_0) \vdash^* (P, \epsilon, \gamma)\}$ where $P \in F$, $\gamma \in \Gamma^*$

(ii) acceptance by empty stack

$\{w \mid (q, w, z_0) \vdash^* (P, \epsilon, \epsilon)\}$ where $P \in Q$.

Equivalence of PDA and CFG:-

→ to show the languages defined by PDA are exactly the context free language.

→ main goal is to prove that CFG and PDA accept the same class of language.

(i) from CFG to PDA:-

→ Given CFG, $G = (V, T, P, S)$, we construct a PDA M that accepts language generated by G

i.e: $L(M) = L(G)$

(variable, terminal, production, start symbol)

→ the machine can be defined as

$M = (\{q\}, T, V \cup T, \delta, q, S, \phi)$

where,

$Q = \{q\}$

$\Sigma = T$

$\Gamma = V \cup T$

$Z_0 = S$

$F = \phi$

$q_0 = q$

δ can be defined as

(i) $\delta(q, \epsilon, A) = (q, \alpha)$ here, $A \to \alpha$ is production P of G.

(ii) $\delta(q, a, a) = (q, \epsilon)$

here transition occurs in only one state q.

alternately
→ define two states p & q
→ p is start state
→ let initially stack symbol is ϵ.
→ PDA starts with p, read ϵ and inserts it into stack and then changes to state q.
→ then all transition occurs in q.

Now,
PDA can be defined as
$M = (\{p, q\}, T, \text{VUT}, \delta, p, \epsilon, \{q\})$
then
$\delta(p, \epsilon, \epsilon) = (q, S)$
$\delta(q, \epsilon, A) = (q, \alpha)$
$\delta(q, a, a) = (q, \epsilon)$ ϵ=pop

e.g: Let $G = (V, T, P, S)$ where P is
$E \to T \mid E+T$
$T \to F \mid T*F$
$F \to a \mid (E)$

Now,
equivalent PDA is
$M = (\{q_0\}, \{a, *, +, (,)\}, \{a, *, +, (,), E, T, F\}, S, q_0, \epsilon, \phi)$
where δ can be defined as
$\delta(q_0, \epsilon, E) \to \{(q_0, T), (q_0, E+T)\}$
$\delta(q_0, \epsilon, T) \to \{(q_0, F), (q_0, T*F)\}$
$\delta(q_0, \epsilon, F) \to \{(q_0, a), (q_0, (E))\}$
$\delta(q_0, a, a) \to (q_0, \epsilon)$
$\delta(q_0, *, *) \to (q_0, \epsilon)$
$\delta(q_0, +, +) \to (q_0, \epsilon)$
$\delta(q_0, (, () \to (q_0, \epsilon)$
$\delta(q_0,),)) \to (q_0, \epsilon)$

Now, Trace for a+(a*a)
using CFG,
$E \to E+T$
$\to T+T$
$\to F+T$
$\to a+T$
$\to a+F$
$\to a+(E)$
$\to a+(T)$
$\to a+(T*F)$
$\to a+(F*F)$
$\to a+(a+F)$
$\to a+(a*a)$

using PDA,
$(q_0, a+(a*a), E)$
$\vdash (q_0, a+(a*a), E+T)$
$\vdash (q_0, a+(a*a), T+T)$
$\vdash (q_0, a+(a*a), F+T)$
$\vdash (q_0, a+(a*a), a+T)$
$\vdash (q_0, +(a*a), +T)$
$\vdash (q_0, (a*a), T)$
$\vdash (q_0, (a*a), F)$
$\vdash (q_0, (a*a), (E))$
$\vdash (q_0, a*a), E))$
$\vdash (q_0, a*a), T))$
$\vdash (q_0, a*a), T*F))$
$\vdash (q_0, a*a), F*F))$
$\vdash (q_0, a*a), a*F))$
$\vdash (q_0, *a), *F))$
$\vdash (q_0, a), F))$
$\vdash (q_0, a), a))$
$\vdash (q_0,),))$ $\vdash (q_0, \epsilon, \epsilon)$
↳ accept by empty stack

Closure properties of Context Free Language:-

- Context free languages are closed under
  (a) Union
  (b) Concatenation
  (c) Kleene star
- Let $G_1=(V_1, \Sigma_1, R_1, S_1)$ and $G_2=(V_2, \Sigma_2, R_2, S_2)$ be two context free grammers.
- Let us assume that they have <u>disjoint set of non-terminals</u> i.e. $(V_1-\Sigma_1)$ and $(V_2-\Sigma_2)$ are disjoint.

(a) Union

→ Let S be new symbol not in G_1 and G_2
→ Construct a new grammar $G=(V, \Sigma, R, S)$ where,

$$V=V_1 \cup V_2 \cup \{S\}$$
$$\Sigma=\Sigma_1 \cup \Sigma_2$$
$$R=R_1 \cup R_2 \cup \{S \to S_1, S \to S_2\}$$

→ here, G is clearly a context free grammar because two new rules added are also of the correct form.
Now, we claim,

$$L(G)=L(G_1) \cup L(G_2)$$

∴ Context free languages are closed under union.

(b) Concatenation

→ Construct a new grammar $G=(V, \Sigma, R, S)$ where,

$$V=V_1 \cup V_2 \cup \{S\}$$
$$\Sigma=\Sigma_1 \cup \Sigma_2$$
$$R=R_1 \cup R_2 \cup \{S \to S_1 S_2\}$$

here,

if $S_1 \overset{*}{\underset{G_1}{\Rightarrow}} w_1$ & $S_2 \overset{*}{\underset{G_2}{\Rightarrow}} w_2$

→ we can claim,

$$S \overset{*}{\underset{G}{\Rightarrow}} w_1 w_2 \quad \because S \to S_1 S_2$$

Here,

$$L(G)=L(G_1)\cdot L(G_2)$$

i.e: CFL are closed under concatenation.

(c) Kleene Star

→ Concatenation
→ Construct a new grammar $G=(V, \Sigma, R, S)$ where

$$V=V_1 \cup \{S\}$$
$$\Sigma=\Sigma_1$$
$$R=R_1 \cup \{S \to SS_1, S \to \epsilon\}$$

here,

if $S_1 \overset{*}{\underset{G_1}{\Rightarrow}} w_1$ then $S \overset{*}{\underset{G}{\Rightarrow}} w_1^*$ $\because S \to SS_1$ & $S \to \epsilon$

hence, $L(G)=L(G_1)^*$

i.e: CFL are closed under Kleene star.

CONTEXT FREE GRAMMARS ARE NOT CLOSED UNDER INTERSECTION:-

→ Define two CFL L_1 and L_2 as

$$L_1=\{a^n b^n c^m : n \geq 1, m \geq 1\}$$
$$L_2=\{a^m b^n c^n : n \geq 1, m \geq 1\}$$

here,

a grammar for L_1 is
$S \to AB$
$A \to aAb \mid ab$
$B \to cB \mid c$

grammar for L_2 is
$S \to AB$
$A \to aA \mid a$
$B \to bBc \mid bc$

Now, the language $L = L_1 \cap L_2 = \{a^n b^n c^n : n \geq 1\}$ is not a CFL.

because,

L_1 requires equal no. of a and b, L_2 requires equal no. of b and c. a string in both must have equal no. of all ~~there~~ three symbols and thus to be in L.

But, as L is not a CFL, we can say $L_1 \cap L_2$ is not a CFL i.e. CFL's are not closed under ~~condition~~ intersection.

→ assume, CFL are NOT closed under Complementation

CFL's are closed under complementation.

Let L_1 and L_2 are two CFLs so, $\overline{L_1}$ and $\overline{L_2}$ are also CFL's.

As we know, CFL's are closed under union.

We can say,

$\overline{L_1} \cup \overline{L_2}$ are context free.

From De-Morgan Laws,

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

This contradicts the theorem that CFL's are closed under intersection.

Hence, CFL's are not closed under complementation

Pumping Lemma for CFL:-

→ It says that in any sufficiently long string in a CFL, it is possible to find at most two short near by substrings that we can pump in tandem (one behind another) i.e. we may ~~respect~~ repeat both of strings for i times and the resulting string will be still in the language.

→ Here string is divided into five parts so that the second and fourth parts may be repeated together

any no. of time and resulting string still remain in language.

→ in pumping lemma for CFL, we have to examine size of parse tree.

→ we will use grammar in CNF as it turn parse tree into binary trees.

Theorem:-

Let a terminal string w be yield of parse tree generated by a grammar $G = (V, \Sigma, R, S)$ in CNF. If length of path is n, then $|w| \leq 2^{n-1}$.

Proof:-

Basic step:-

Let n=1, then the tree consists of only the root and a leaf with terminal. String w be a simple terminal.

Thus,

$|w| = 1 \Rightarrow 2^{1-1} = 2^0 = 1$ which is true.

Inductive step:-

→ Let n be the length of path and $n > 1$. Then the root of the tree uses a production of the form $A \to BC$ since $n > 1$.

→ No path in the subtree rooted at B & C can have length greater than n-1, since edges from root 'A' to child B & C are excluded.

→ Thus, by inductive hypothesis, these two subtrees yields of length atmost 2^{n-2}

→ The yields of entire tree is concatenation of those two yields and so length in at most

$2^{n-2} + 2^{n-2} = 2^{n-1}$ $\therefore |w| \leq 2^{n-1}$ proved

Abiral Sangroula

Pumping Lemma statement:

→ Let L be a CFL and w be any string in L
→ Let n be a positive integer or pumping constant with $|w| \geq n$.
→ Then we can write $w = uvxyz$ such that
(i) $|vxy| \leq n$
(ii) $|vy| \geq 0$
(iii) for all $i \geq 0$, $uv^ixy^iz \in L$

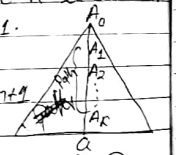
i.e. two strings v and y can be pumped any no. of times and the resulting string will still be in L

Proof:-

→ Consider a CNF grammar $G = (V, \Sigma, R, S)$ such that $L(G) = L - \{\epsilon\}$
→ Let G has m variables. Choose $n = 2^m$.
→ Suppose $z \in L$ is of length at least n i.e. $|z| \geq n$
→ any parse tree for z must have height $m+1$ if it is less than $m+1$ say m then by theorem of size of parse tree.

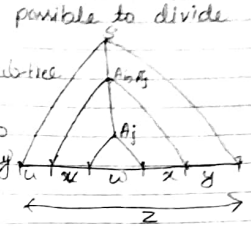
$$|z| = 2^{m-1} = 2^m/2 = n/2$$

which is contradiction so, height should be $m+1$
→ Let us consider a path of max^m length in tree for z as shown in fig ①. where k is at least of length m and the path is of length $k+1$.

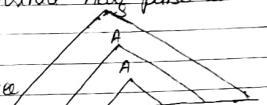


→ since $k \geq m$, there are at least $m+1$ occurence of variable $A_0 A_1 \cdots A_k$ on the path.
→ But there are only m variable in grammar so at least two of last $m+1$, variables on path must be same (by pigeonhole principle)

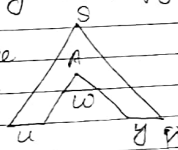
→ suppose $A_i = A_j$ then it is possible to divide tree as below in fig ②
→ here, string w is yield of subtree rooted at A_j.
→ string v & x are strings to the left and right respectively of w, is yield of larger subtree rooted at A_i.



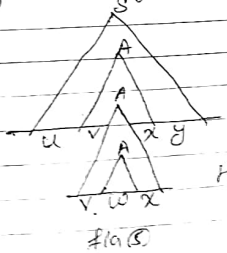
→ u and y represent beginning and end of tree rooted at S.
→ since there are no unit production (removing) v & x both couldn't be empty, although one (remaining) could be. So, $|vx| \geq 0$
→ if $A_i = A_j = A$ then we can construct new parse tree as in fig ③



→ Now, we may replace subtree rooted at A_i which has yield vwx by subtree rooted at A_j which has yield w. The resulting tree is fig ④ that yields uv^0wx^0y



→ Next we may replace subtree rooted at A_j by entire subtree rooted at A_i, the resulting parse tree is as shown in fig ⑤ that yield pattern uv^2wx^2y.



Hence, we can find any yield of form $uv^iwx^iy \in L$ for any $i \geq 0$

Pumping Algorithm for CFL

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

→ Let L be a CFL and ω be any string in L, where $|\omega| \geq m$ and m be pumping constant with $|uyz| \leq m$. Then ω can be written as $\omega = uvxyz = a^m b^m c^m$

such that;

$|vxy| \leq m$ & $|vy| > 0$

If a given language L is CFL, then it must satisfy $uv^i xy^i z \in L$ for $i \geq 0$

Case 1: vxy contains both as and bs

Now, $\omega = uvxyz$

$= \underbrace{aa\ldots a}_{u} \underbrace{a\ldots ab\ldots b}_{vxy} \underbrace{b\ldots bc\ldots cc\ldots c}_{z}$

Let,

$v = a^p, x = a^q, y = b^r ; p+r \geq 1$

then for $i=2$;

$uv^2xy^2z = a^{m+p+q+r} b^{m+r} c^m \notin L$

i.e: it contradicts our assumption.

$\therefore uvxyz \rightarrow a^m b^m c^m$

$a^{m+p} b^{m+r} c^m$

Case 2: vxy contains both bs and cs

Hence,

$\omega = uvxyz$

$= \underbrace{aa\ldots aa\ldots ab\ldots b}_{u} \underbrace{b\ldots bc\ldots c}_{vxy} \underbrace{c\ldots cc}_{z}$

Let,
$v=b^p, x=b^q, y=c^r$; $p+r \geq 1$
Then for $i=2$;
$uv^2xy^2z = a^m b^{m+p} c^{m+r} \notin L$
i.e. it contradicts our assumption.

Case 3: vxy is in a^m.

here,
$w = uvxyz$
$= \underbrace{a\cdots a}_{u}\ \underbrace{a\cdots a}_{vxy}\ \underbrace{a\cdots abb\cdots bbc\cdots c\cdots ccc}_{z}$

Let,
$v=a^p$ $y=a^r$; $p+r \geq 1$
for $i=2$;
$uv^2xy^2z = \underbrace{a\cdots a}_{u}\ \underbrace{a\cdots a}_{v^2}\ \underbrace{a\cdots a}_{x}\ \underbrace{a\cdots a}_{y^2}\ \underbrace{a\cdots abb\cdots bbc\cdots c}_{z}$
$= a^{m+p+r} b^m c^m \notin L$
i.e. it contradicts our assumption.

Hence,
given language;
$L = \{a^m b^m c^m : n \geq 1\}$ is not CFL.

Turing Machine

TURING MACHINE

→ is an abstract machine developed by english mathematician Alan Turing in 1936.
→ is the theoretical foundation of modern computers.
→ TM will have:-
(i) finite set of alphabets
(ii) finite set of states.
(iii) linear tape which is potentially infinite in both end.
→ each square or cell of tape can hold one symbol from alphabet.
→ if no symbol then it contains blanks.
→ reading and writing is done by tape-head.
→ Tape serves as:- (i) input device
(ii) memory available for use during computation
(iii) output device.
→ TM is a much more accurate model of general purpose computer.
→ it can do everything that a real computer can do.
→ it consists of following:-
(i) there are K tapes, for some fixed $K \geq 1$. Each tape is divided into cells, and is ∞ both to the left and to the right.
(ii) Each cell stores a symbol belonging to a finite set of tape alphabet. The tape alphabet contains blank symbol □ if cell contains □, it means cell is actually empty.

(ii) each tape has a tape-head which can move along the tape, one cell per move. It can also read the cell. It currently scans and replace the symbol in the cell by another symbol.

(iii) there is a state control, which can be in any one of finite no of states. The finite set of states, Q, contains three special states: start state, accept state & reject state.

→ TM performs a sequence of computation steps. In one step it does following:-

(i) a: Immediately before the computation step, TM is in a state r of Q and each of k tape heads is on certain cell. (r may be start, accept, reject).

b: depending on the current state r and k symbols that are read by tape-head.

(ii) TM switches to state r' of Q (which may be equal to r).

(iii) Each tape head writes a symbol of tape alphabet in the cell it is currently scanning.

(iv) each tape head either moves one cell to the left or to the right or stays at current cell.

FORMAL DEFINITION:-

→ TM is a 7 tuple :-

$$M = (\Sigma, \Gamma, Q, \delta, q, q_{accept}, q_{reject})$$

Constant; q_0, B, F → Final state; B → Blank state; q_0 → initial state

where,

Σ is a finite set of i/p alphabet; $\square \notin \Sigma$

Γ " " " " " tape " ; $\square \in \Gamma, \Sigma \subseteq \Gamma$

Q " " " " " state

q " " start state

q_{accept} is accept state

q_{reject} is reject state

δ is a transition function

where, $\delta : Q \times \Gamma^k \longrightarrow Q \times \Gamma^k \times \{L, R, N\}^k$

here,

transition function δ is the program of TM that tells what machine can do in one computation step

Let,

$r \in Q, a_1, a_2, \dots a_k \in \Gamma$

$r' \in Q, a_1', a_2', \dots a_k' \in \Gamma$

and, $\sigma_1, \sigma_2, \sigma_k \in \{L, R, N\}$

~~Now,~~

~~$\delta(r, a_1, a_2 \dots a_k) = (r', a_1', a_2' \dots a_k', \sigma_1, \sigma_2, \dots \sigma_k)$~~

Then,

$\delta(r, a_1, a_2 \dots a_k) = (r_1', a_2', a_2' \dots a_k', t, t_2 \dots t_k)$

Now,

$\delta(r, a_1, a_2 \dots a_k) = (r_1', a_1', a_2' \dots a_k', \sigma_1, \sigma_2 \dots \sigma_k)$

This means,

→ The TM is in state r

→ the head of i^{th} tape reads the symbol $a_i, 1 \le i \le k$ then

→ TM switch to state r'

→ The head of i^{th} tape replaces the scanned symbol a_i by symbol a_i', $1 \le i \le k$

if $\delta = L$, then tape head moves one cell to left
$\delta = R$, then it moves one cell to right
$\delta = N$, then it doesn't move.

→ the computation terminates at the moment when the TM enters the q_{accept} state or q_{reject} state.
→ TM accepts input string if the computation terminates in q_{accept}.
→ TM rejects input string if computation terminate in q_{reject}.
→ L(M) is language accepted by TM i.e: set of ~~set~~ all string in Σ^*

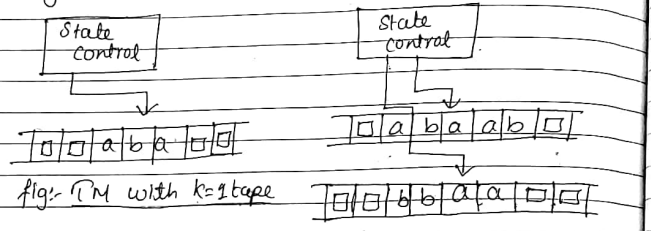


fig:- TM with k=1 tape

fig:- TM with k=2 tapes

Instantaneous Description for Turing Machine:-

→ A string $x_1, x_2 \ldots x_{i-1}\ q\ x_i, x_{i+1} \ldots x_n$ represents instantaneous description of TM where,
q is state of TM
→ q is state of TM
→ The tape head is scanning the i^{th} symbol from left.
→ $x_i \ldots x_n$ is the portion of tape between left-most and right-most non-blank.

Moves of TM:-

The mass of TM, $M = (Q, \Sigma, \tau, \delta, q_0, B, F)$ is described by the notation $\vdash$ for single move and $\vdash^*$ for zero or more moves.

① for $\delta(q, x_i) = (p, Y, L)$ i.e: next move is leftward. Then,

$$x_1, x_2 \ldots x_{i-1}\ q\ x_i, x_{i+1} \ldots x_n \vdash_M x_1 x_2 \ldots x_{i-2}\ p\ x_{i-1}\ x_{i+1} \ldots x_n$$

for $i = 1$

$$q\ x_1 x_2 \ldots x_n \vdash_M p B Y x_2 \ldots x_n$$

for $i = n$, $Y = B$

$$x_1 x_2 \ldots x_{n-1}\ q\ x_n \vdash x_1 x_2 \ldots x_{n-2}\ p\ x_{n-1}$$

② for $\delta(q, x_1) = (p, Y, R)$ i.e: next move is rightward then,

$$x_1, x_2 \ldots x_{i-1}\ q\ x_i \ldots x_n \vdash x_1 x_2 \ldots x_{i-1}\ Y\ p\ x_{i+1} \ldots x_n$$

for $i = n$

$$x_1, x_2 \ldots x_{n-1}\ q\ x_n \vdash x_1 x_2 \ldots x_{n-1}\ Y\ p\ B$$

for $i = 1$, $Y = B$

$$q\ x_1 x_2 \ldots x_n \vdash p\ x_2 \ldots x_n$$

Sample: Design a TM accepting $\{0^n 1^n, n \geq 1\}$

Sol:

Given finite sequence of 0's and 1's on its tape preceded and followed by blanks.

→ TM will change 0 to X and then 1 to Y until all 0's and 1's matches.

→ Starting at left end of input, it repeatedly changes 0 to X and moves over whatever 0's and Y's it sees until comes to q_1.

→ it changes 1 to Y and moves left over Y's and 0's until it find an X.

→ At that point it looks for 0 immediate to the right.

→ if it finds one 0 then changes it to X and repeats process changing a matching 1 and Y.

Now,

TM is $M = (\{q_0, q_1, q_2, q_3, q_4\}, \{0, 1\}, \{0, 1, X, Y, B\}, \delta, q_0, B, \{q_4\})$

Transition-table:-

	0	1	X	Y	B
q_0	(q_1, X, R)	—	—	(q_3, Y, R)	—
q_1	$(q_1, 0, R)$	(q_2, Y, L)	—	(q_1, Y, R)	—
q_2	$(q_2, 0, L)$	—	(q_0, X, R)	(q_2, Y, L)	—
q_3	—	—	—	(q_3, Y, R)	(q_4, B, R)
q_4	—	—	—	—	—

① $\{0^n 1^n 2^n ; n \geq 1\}$ N.V.V.
② $\{0^n b^n c^n ; n \geq 1\}$ imp

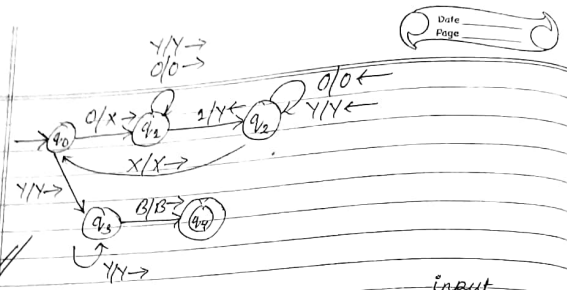
Test for string 0011

$q_0 0011$
$\vdash X q_1 011$
$\vdash X 0 q_1 11$
$\vdash X q_2 0 Y 1$
$\vdash q_2 X 0 Y 1$
$\vdash q_2 X 0 Y 1$
$\vdash X q_0 0 Y 1$
$\vdash X X q_2 Y 1$
$\vdash X X Y q_1 1$
$\vdash X X q_2 Y Y$
$\vdash X q_2 X Y Y$
$\vdash X X q_0 Y Y$
$\vdash X X Y q_3 Y$
$\vdash X X Y Y q_3$
$\vdash X X Y Y q_4$

Transition Diagram for TM:-

→ it consists of
- set of nodes representing states of TM.
- an arc from any state q to p is labelled by the items of the form X|YD where X and Y are tape symbols and D is a direction either L or R.
  i.e: $\delta(q, X) = (P, Y, D)$

here,
label X|YD on arc from q to p.
D is ← for left and → for right.



Duplicate

Q. TM to decide whether or not any string input $w \in \{a,b\}^*$ is a palindrome.

→ Initially tape consists of input string w, tape head is on leftmost symbol of w and TM is in start state q_0.

→ The tape head reads leftmost symbol of w, replace it by B. Then tape head moves to the rightmost symbol and test whether it is equal to leftmost symbol.

(i) If they are equal, then rightmost symbol is deleted, the tape head moves to new leftmost symbol and whole process is repeated.

(ii) If they are not equal, T.M. enters the reject state and computation terminates.

→ TM enters accept state as soon as string currently stored on tape is empty.

Now,

TM $M = (\{q_0, q_a, q_b, q_a', q_b', q_L, q_{accept}, q_{reject}\}, \{a,b\}, \{a,b,B\}, \delta, q_0, q_{accept}, q_{reject})$

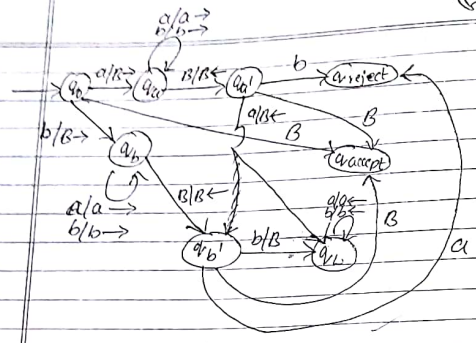
Transition table:-

	a	b	B
q_0	(q_a, B, R)	(q_b, B, R)	q_{accept}
q_a	(q_a, a, R)	(q_a, b, R)	(q_a', B, L)
q_b	(q_b, a, R)	(q_b, b, R)	(q_b', B, L)
q_a'	(q_L, B, L)	q_{reject}	q_{accept}
q_b'	q_{reject}	(q_L, B, L)	q_{accept}
q_L	(q_L, a, L)	(q_L, b, L)	(q_0, B, R)
q_{reject}			
q_{accept}	—	—	—
q_{reject}	—	—	—

Test for string: abba

$q_0 abba$
$\vdash B q_a bba$
$\vdash Bb q_a ba$
$\vdash Bbb q_a a$
$\vdash Bbba q_a$
$\vdash Bbb q_a' a$
$\vdash Bb q_L bB$
$\vdash B q_L bbB$
$\vdash q_L BbbB$
$\vdash B q_0 bbB$
$\vdash BB q_b bB$
$\vdash BBb q_b B$
$\vdash BB q_b' bB$
$\vdash B q_L BbB$
$\vdash BB q_0 bB$
$\vdash BBB q_0 B$
$\vdash BB q_b' BB$
$\vdash BB q_{accept} BB$

$\therefore$ T.M. accepts abba.



Test for

The Language of TM:-

→ if $M=(Q, \Sigma, \Gamma, \delta, q_0, B, F)$ is a TM, then language of TM $L(M)$ is the set of strings w in Σ^* such that $q_0 w \vdash^* \alpha p \beta$ where $p \in F$ and α, β are any tape string.

→ The set of languages accepted using TM are called "recursively Enumerable languages".

→ TM is designed to perform at least following three roles:-

(i) as a language recognizer:- TM can be used for accepting a language like FA and PDA.

(ii) as a computer of function:- TM represents particular function; initial input is treated as representing an argument of function and the final string on tape when TM enters halt state will be treated as representative of value obtained

by application of function to argument represented by initial string.

(iii) as an enumerator of string of language:- it outputs the string of language one at a time in some systematic order that is as a list.

TM for computing a function:- (How can you represent TM for computing a function?)

→ a TM can be used to compute function.

→ for this, adopt following policy to input any strings to TM which is input to computation function:-

(i) string w is presented into from BwB, where B is blank symbol, and placed on the tape. the head is positioned at a blank symbol which immediately follows the string w.

(ii) the symbol to the current position of tape head can be shown as (q, BwB) or presented by T.D. of TM as $Bwq\overset{\downarrow}{B}B$.

(iii) TM is said to halt on input w if we can reach to halting state after performing some operation.

i.e: for $M=(Q, \Sigma, \Gamma, \delta, q_0, B, \{q_a\})$

TM is said to halt on input w if and only if $BwqB$ yields to $B\alpha q_a aB$ for some $\alpha \in \Gamma^*$

$(q, BwB) \vdash (q_a, B\alpha B)$

Formal Definition:-

- a function $f(x)=y$ is said to be computable by TM defined as $(Q, \Sigma, \Gamma, \delta, q_0, B, \{q_a\})$ if $(q_0, B\alpha B) \vdash^* (q_0, ByB)$

-It means that if we give input x to TM, it gives output as string if it computes the function, $f(x)=y$.

e.g. Design a TM which computes the function $f(x)=x+1$ for each x belonging to set of natural numbers.

soln

Given function $f(x)=x+1$, we represent x on tape by 1's on tape

i.e. for $x=1$, input tape will be B1B
for $x=2$, " " " " B11B
and so on.

→ Similarly, output can be seen by number of 1's on tape when machine halts.

here, let TM $=(Q, \Sigma, \tau, \delta, q_0, B, \{q_f\})$

where,

$Q=\{q_0, q_f\}$

$\tau=\{1, B\}$

$\Sigma=\{1\}$

δ can be

	1	B
q_0	$(q_f, 1, R)$	$(q_0, 1, N)$

for input $x=3$,

input tape is B111B

Then,

$(q_0, B111B) \vdash (q_0, B111)$
$\vdash (q_f, B1111B)$

Here, o/p 4 accepted & TM halts.

Turing Machine and Halting:-

→ acceptance by halting

→ TM halts if it enters a state q, scanning a tape symbol X & there is no more in this situation i.e. $\delta(q, X)$ is undefined.

→ We assume a TM always halts when it is in accepting state i.e. without changing language accepted we can make $\delta(q, X)$ undefined whenever q is in accepting state.

Turing Machine with storage in the State:-

→ In TM, generally, any state represents the position in computation. But, the state can also be used to hold finite amount of data.

→ We can use the finite control not only to represent a position in computation but also to hold a finite amount of data.

→ hence, a ~~bit~~ state is considered on a tuple (state, data)

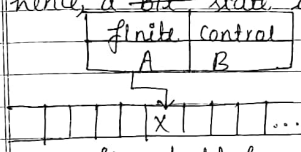


fig:- hold for storage

In this model of computation;
δ is defined by
$\delta([q, A], X)=([q_1, X], Y, R)$.

→ it means that q is the state and data is A; the symbol scanned on the tape is copied to data portion of state and moves right entering state q_1 and replacing symbol by Y.

e.g. a model of TM that recognize a language $L=\{w \in \{0,1\}^* / w$ where it sees a first symbol either 0 or 1 and then checks that it doesn't appear elsewhere in input.

→ here, state is thought of a pair with two components:-

(i) a control portion q_0 or q_1 where q_0 indicates that TM has not yet read its first symbol. While q_1 indicates that it has read symbol & is checking that it doesn't appear elsewhere by moving right and hoping to reach blank cell.

(ii) a data portion which remembers the first symbol seen, which must be 0 or 1. The symbol B in this component means no symbol has been read.

here,

$\delta \rightarrow$

1) $\delta([q_0, B], a) \rightarrow ([q_1, a], a, R)$: for $a=0$ or 1.

2) $\delta([q_1, a], a') \rightarrow ([q_1, a], a', R)$: for $a=0$ if $a=1$, $a'=1$ if $a=0$

3) $\delta([q_1, a], B) \rightarrow ([q_1, B], B, R)$: or q_{accept}

4) $\delta([q_1, a], a) \rightarrow q_{reject}$

→ if TM reaches first blank it enters accepting state.

→ if TM reaches or encounters second occurence of same symbol it halts without accepting.

Now,

$Q=\{q_0, q_1, q_{accept}, q_{reject}\}$

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

$\Gamma=\{0,1,B\}$

$\therefore M=(\{q_0, q_1, q_{accept}, q_{reject}\}, \{0,1\}, \{0,1,B\}, \delta, q_0, q_{accept}, q_{reject})$

Recursively Enumerable Language:- (Def: there is a TM already and the data languages)

→ the set of RE languages are precisely those language that can be accepted by TM.

→ strings that are not in the language may be rejected or may cause TM to go into infinite loop.

Recursive Language:- (Def: if there is a TM, then that TM checks the string)

→ a language is recursive if there exists a TM that accepts every string of the language and rejects strings that are not in language.

→ recursive language always halts TM.

→ recursive language is subset of RE language.

e.g:

Let $L=\{w \in \{a,b\}^* : w$ contains at least one 'a'$\}$

here,

Def: TM scans to the right to find one a. If no a is found, it goes forever never halting. It halts only if there is at least one a.

→ This language is RE.

Turing Recognizable Language:-

→ a language L is Turing recognizable if there exists a TM such that for all strings w;
- if $w \in L$, eventually M enters qaccept.
- if $w \notin L$, either M enters qreject or M never terminates.

Turing Decidable Language:-

→ a language L is turing decidable if there exists a TM M such that for all string w,
- if $w \in L$, M enters qaccept.
- if $w \notin L$, M enters qreject.

hence,
decidable language always terminates.
recognizable language can run forever without deciding.

Extensions of TM:-

1) Multiple TM:-

→ Multiple TM consists of a finite control & finite no. of tapes.
→ each tape is divided into cells and each cell can hold any symbol of finite tape alphabets.
→ the set of tape symbols include a blank and input symbols.
→ in multiple TM initially,
(i) the input w is placed on the first tape.
(ii) all other cells of the tapes hold blank.
(iii) is in initial state q_0.

(iv) the head of first ~~place~~ tape is at left end of i/p.
(v) all other tape heads are at some arbitrary cells.

→ A move of multiple TM depends on the state & the symbol scanned by each of tapehead.
→ In one move, the multiple TM does following:-
(i) the control enters in a new state, which may be same previous state.
(ii) on each step, a new symbol is written on the cell scanned, these symbols may be same as the symbols previously there.
(iii) each of the tape head make a move either left or right or remain stationary. Different head may have different direction independently i.e: if head of first tape moves leftward, at same time other head can move another directions or remain stationary.

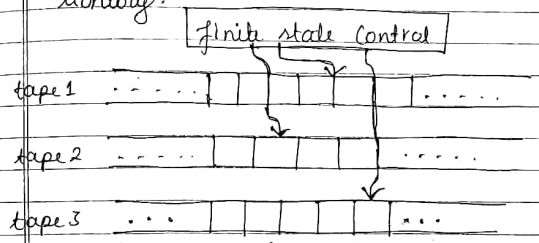


fig:- Multi-tape TM

→ for any fixed integer $K \geq 1$, a k tape TM is a TM equipped as above with K-tapes and corresponding heads.

Formally,
K-tape TM is a 7 tuple $(Q, \Sigma, \Gamma, \delta, q_0, B, F)$
here,
$\delta : Q \times \Gamma^K \longrightarrow Q \times \Gamma^K \times \{L, R, N\}^K$ where K is no. of tapes.

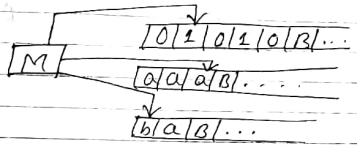
the expression $\delta(q_i, a_1 \dots a_K) = (q_j, b_1 \dots b_K, L, R, \dots L)$ means that if the machine is in state q_i and heads 1 through K are reading symbols a_1 through a_K, the machine goes to state q_j writes symbol b_1 through b_K and directs each head to move left or right or to stay as specified.

THEOREM:- (For any 'K' tape heads TM it can be converted into a single Turing machine)

Let $K \geq 1$ be an integer. Any K-tape TM can be converted to an equivalent one tape TM.
here,
→ convert a multitape TM, M to an equivalent single tape TM 'S'.
→ key idea is to show how to ~~stimulate~~ simulate M with S.
→ let M has K-tapes, then S simulates the effect of K-tapes by storing their information on its single tape.
→ it uses the new symbol # as a delimiter to separate the contents of the different tapes.
→ in addition to the contents of these tapes, S must keep track of the locations of the heads by writing a tape symbol with a dot (•) above it

to mask the place where the head on that tape would be.

→ following fig:- illustrates how one tape can be used to represent three tapes.



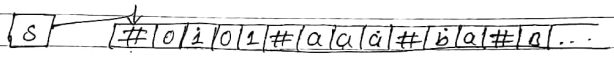


fig:- representing 3 tapes with one.

→ on input $w = w_1 \dots w_n$
1) first S puts its tape into the format that represents all K-tapes of M. The formatted tape contains;
$\# w_1 w_2 \dots w_n \# B \# B \# \dots \#$
2) To simulate a single move, S scans its tape from the first #, which makes lefthand end, to the (K+1)#, which makes righthand end, in order to determine the symbols under virtual heads. Then, S makes a second pass to update the tape according to ways that M's transition function dictate.
3) If at any point S moves one of the virtual head to the right onto a #, this action signifies that M has ~~moded the~~ moved the corresponding head onto a previously unread the blank portion of that tape. So, S write a blank symbol on this tape cell and (always place the blank at last)

Types of TM:- i) Multiple TM
ii) Non-Deterministic TM
iii) TM with infinite tapes
iv) TM with multiple heads

shifts the tape contents, from this cell until right most # one unit to the right. Then it continues the simulation as before.

Non-Deterministic TM:- (DTM)

→ is the one which at any point in a computation may proceed according to several possibilities.
→ the transition function δ are subsets rather than single element of set $Q \times \Gamma \times \{R, L, S\}$.
→ here, transition function δ is such that for each state q and tape symbol X, $\delta(q, X)$ is set of triples $\{(q_1, Y_1, D_1), (q_2, Y_2, D_2) \ldots (q_k, Y_k, D_k)\}$ where k is finite integer.
→ the NTM can choose, at each step, any of the triples to be the ~~most~~ next move but it can't however pick a state from one, a tape symbol from other and direction from yet another.

→ the computation of a NTM is a tree whose branches correspond to different possibilities for the machine.
→ if some branch of the computation leads to the accept state, the machine accepts its input.

* If M_N is a non-deterministic TM, then there is a Deterministic TM M_D such that $L(M_N) = L(M_D)$

v) One tape TM
vi) TM with two tapes

Unrestricted Grammar:-

→ is a grammar that generates class of recursively enumerable language.
→ is a grammar $G = (V, \Sigma, R, S)$ where;
V = set of symbols, both terminals & non-terminals
$\Sigma \subseteq V$ is set of terminals
$R \subseteq (V^*(V-\Sigma)V^* \times V^*)$ is set of rules.
$S \in (V-\Sigma)$ is start symbol.
→ here, the left hand side of rule contains a string of terminals and non-terminals, at least one of which must be a non-terminal.
→ Rules are applied just like CFG.
→ To generate a string with an ~~unrealized~~ unrestricted grammar:-
→ starts with initial symbol.
→ while the string contains at least one non-terminal
→ find a substring that matches the L.H.S. of some rule.
→ Replace that substring with RHS of rule.

e.g. Grammar for $L = \{a^n b^n c^n : n > 0\}$

$G = (V, \Sigma, R, S)$ where,

$V = \{S, a, b, c, A, B, C, T_a, T_b, T_c\}$

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

$R = \{$
$S \to ABCS$
$S \to T_c$
$CA \to AC$
$BA \to AB$
$CB \to BC$
$CT_c \to T_c C$
$T_c \to T_B$
$BT_B \to T_B b$
$T_B \to T_A$
$AT_A \to T_A a$
$T_A \to \in$ $\}$

Now, for "aabbcc"

$S \to ABCS$
$\to ABCABCS$
$\to ABCACBCS$
$\to AABCBCS$
$\to AABCBCCS$
$\to AABBCCTc$ $\to AT_Aabbcc$
$\to AABBCTcC$
$\to AABBTcCC$
$\to AABBT_bccCC$
$\to AABT_bbccCC$
$\to AAT_bbbccCC$
$\to AATAbbCCcc$
$\to ATAabbCCcc$
$\to TAaabbCCcc$
$\to \epsilon aabbCCcc$
$\to aabbCCcc$

Chapter:-6

Undecidability:-

Church Turing Thesis:- (SAMAT-R-IBUM)

→ It states that every computation or algorithm can be carried out by a Turing Machine.
→ This statement was first formulated by Alonzo Church
→ It is not a mathematically precise statement so, unprovable.
→ However, it is now-generally assumed to be true
→ the thesis might be replaced on saying that the notation of effective or mathematical metho

Chapter:-6

Undecidability

in logic and mathematics is captured by TM.
→ It is generally assumed that such methods must satisfy the following requirements:-
(a) the method consists of a finite set of simple and precise instructions that are described with a finite no. of symbols.
(b) the method will always produce the result in a finite no. of steps.
(c) the method can in principle be carried out by human being with only paper and pencil.
(d) the execution of the method requires no intelligence of human being except that which is needed to understand and execute instruction.
→ Invention of TM has accumulated enough evidence to except that which is need to understand to execute instruction.
→ The Invention of TM has accumulated enough evidence to adopt this hypothesis also understand to execute instruction
→ it is believed that there is no functions that can be defined by human, whose calculation can be described:-
by any well-defined mathematical algorithm that cannot be computed by Turing Machine

Turing Machine

→ TM is believed to be ultimate calculating mechanis
→ after adapting Church Turing thesis, we can give precise meaning of term as
"An algorithm is a procedure that can be executed on TM".

Universal Turing Machine: (SC◆DAR)

→ if a TM really is a sound model of computati it should be possible to demonstrate that it ca act as a stored program machine, where the pro gram is regarded as an input rather than hard wired.

→ we shall construct a TM M_U that takes as i/p as description of TM M and an i/p word x and simulates the computation of M on i/p x.

→ A machine such as M_U that can simulate the behaviour of an arbitary TM is called an Universal Turing Machine.

→ Thus, we can describe universal turing machine T_U as a TM that on i/p $\langle M, W \rangle$; where M is a and W is string, simulates computation of M on i/p

→ T_U accepts $\langle M, W \rangle$ iff M accepts W

→ T_U rejects $\langle M, W \rangle$ iff M rejects W.

Encoding of TM (PDUST-ROCCC)

→ Formulate a notational system where we can encode both an arbitary TM T_1 and an i/p string x over an arbitary alphabet as string $e(T_1)$ and $e(x)$ over some fixed alphabet.

→ This encoding must not destroy any information i.e. we must be able to reconstruct TM T_1 and string x.

→ For encoding of TM, use alphabet {0,1}, although TM may have much larger alphabet.

→ start by assigning positive integers to each state, each tape symbol and each of three directions in TM T_1 we want to encode.

→ Here,
we assume two fixed infinite sets
$Q = \{q_1, q_2 \cdots\}$ and $S = \{a_1, a_2 \cdots\}$ so that for any TM $T = \{Q, \Sigma, \Gamma, \delta, q_0, B, F\}$; $Q_1 \subseteq Q$
$\Gamma \subseteq S$

→ we can represent a state or a symbol by a string of 0's of appropriate length. Here 1's are used as separators.

→ Once, we have established an integer to represent each state, symbol and direction, we can encode now the transition function δ.

→ let one transition rule is
~~$\delta(q_i, q_j) = (q_k, q_l$~~
$\delta(q_i, a_j) = (q_k, a_l, D_m)$ for some integer i, j, k, l, m.

→ Then, we can code this rule by string
$\delta(q_i)1\,\delta(a_j)1\,\delta(q_k)1\,\delta(a_l)1\,\delta(D_m)$ say m_1;
where δ is encoding function.

→ A code for entire TM T_1 consists of all codes for transition in some order, separated by pair of 1's like $m_1 11 m_2 11 \cdots m_n$

→ Now, code for TM and I/p string x will be formed by separating them by three consecutive 1's. e(TM) 111 e(x)

Encoding function S:-

→ First associate a string of 0's to each state, each tape symbol and each of three directions.

→ Let, function S is defined as;

$S(B) = 0$

$S(a_i) = 0^{i+1}$ for each $a_i \in S$

$S(q_i) = 0^{i+2}$ for each $q_i \in Q$

$S(;) = 0$

$S(L) = 00$

$S(R) = 000$

Consider an example where TM T is defined as

$T = (\{q_1, q_2, q_3\}, \{a, b\}, \{a, b, B\}, \delta, q_1, B, F)$

where δ is defined as; $\delta(q_1, b) = (q_3, a, R)$

$\delta(q_3, a) = (q_2, b, R)$

$\delta(q_2, b) = (q_2, a, R)$

$\delta(q_3, B) = (q_1, b, L)$

Now, using encoding function S as defined above

$S(q_1) = 000$

$S(q_2) = 0000$

$S(q_3) = 00000$

$S(a_1) = 00$: say $a_1 = a$

$S(a_2) = 000$ $a_2 = b$

$S(B) = 0$

$S(R) = 000$

$S(L) = 00$

$S(;) = 0$

Now, for m_1 :

$e(m_1) = S(q_1)\,1\,S(b)\,1\,S(q_3)\,1\,S(a)\,1\,S(R)$

$= 0001000100000100 1000$

for m2

$e(m_2) = S(q_1)\,1\,S(a)\,1\,S(q_1)\,1\,S(b)\,1\,S(R)$

$= 0000010010001000 1000$

for m3

$e(m_3) = S(q_3)\,1\,S(b)\,1\,S(q_2)\,1\,S(a)\,1\,S(R)$

$= 0000010001000001000$

for m4

$e(m_4) = S(q_3)\,1\,S(B)\,1\,S(q_3)\,1\,S(b)\,1\,S(L)$

$= 000001 01 000001000100$

Now,

Code for TM T

$e(T) = e(m_1)\,11\,e(m_2)\,11\,e(m_3)\,11\,e(m_4)$

$= 0001000100000100 1000 11$

$0000010010001000 100011$

$0000010001000001 000011$

$000001010000010001 00$

Now,

for T and any i/p string x where x = ab

code will be = $e(T)\,111\,e(x)$; where $e(x) = S(a)\,1\,S(b)$

$= 001000$

$= 0001000100000100 1000 11 0000010010001$

$0001000111000001000100001000110000010100000 1$

000100111001000

The halting Problem: (LCNPMDU)

→ There are limits to the ~~paper~~ power of TM.
→ A TM continues until it reaches accept state or reject state where it will halt.
→ if it never reaches one, then it continues computation forever.
→ there exists problems that TM can't solve.
→ the best known problem i.e: unsolvable by a TM is the halting problem
→ Halting problem is:-
"Given an arbitrary TM T as i/p and ~~equal~~ arbitrary tape t, to decide whether T halts on t"
"To determine for an arbitrary given TM Tm and i/p w, whether Tm will eventually halts on i/p w"
→ the halting problem is not solvable by any TM.

Undecidable Problem about Turing Machine:-

→ The problems for which no algorithm exists are called undecidable or unsolvable.
e.g: halting problem
→ following problems about TM are ~~d~~ undecidable:-
(i) Given a TM M and i/p string w, does M halts on input w?
(ii) Given a TM M, does M halt on ~~the~~ empty tape?
(iii) Given a TM M, is there any string at all on which M halts?
(iv) Given a TM M, does M halts on every i/p string?
(v) Given two TMs M_1 and M_2, do they halt on same input string?

(vi) Given a TM M, is the language that M semidecides regular? Is it context free? Is it recursive?

Unsolvable Problems about Grammar:-

→ for a given grammar (G) and string (w), to determine whether $w \in L(G)$.
→ for a given G_1, to determine whether $\epsilon \in L(G_1)$.
→ for two grammars G_1 & G_2, to determine whether $L(G_1) = L(G_2)$
→ for an arbitrary grammar G_1, to determine whether $L(G_1) = \phi$.

Properties of Recursive Language:-

1) The complement of recursive language is recursive:-
→ Let L be a recursive language.
→ T be a turing machine that ~~halts~~ on all inputs and accepts L.
→ let us construct a T' from T so that if T enters a final state on input w, then T' halts without accepting.
→ if T halts without accepting, T' enters final state.
→ so, L(T') is the language accepted by T' is the complement of L.
→ hence, the complement of recursive language is recursive

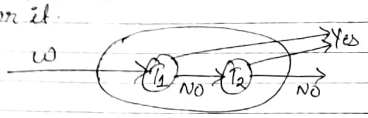
2) The union of two recursive language is also recursive:-

→ Let L_1 and L_2 be two recursive languages accepted by Turing Machines T_1 and T_2.

→ Construct a TM T that first simulates T_1.
- if T_1 accepts then T accepts.
- if T_1 rejects then T simulates T_2 and accepts if and only if T_2 accept.

→ here, T is guaranteed to halt because both T_1 and T_2 are algorithms. i.e: T accepts $L_1 \cup L_2$.

→ hence, $L_1 \cup L_2$ is also recursive since there exists TM for it.

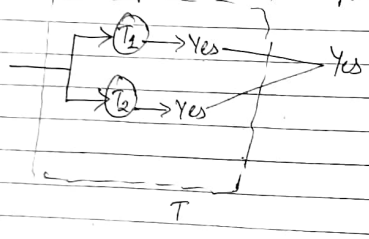


3) The union of two recursively enumerable language is recursively enumerable:-

→ Let L_1 and L_2 are recursively enumerable languages and their enumerative TM are T_1 and T_2.

→ T be a TM that can simulate T_1 and T_2 simultaneously on separate tapes.

→ if either T_1 or T_2 accepts then T accepts as follows:-



4) If a language L and its component $\bar{L}$ are both recursively enumerable then L (and $\bar{L}$) is recursive:-

→ Let T_1 and T_2 accepts L and $\bar{L}$ respectively.

→ construct a TM T that simulates T_1 and T_2 simultaneously.

→ T accepts w if T_1 accepts and rejects w if T_2 ~~will~~ accepts.

→ Thus, T will say either yes or no but not both.

→ Since, T is an algorithm that accepts L, it follows that L is recursive (hence $\bar{L}$ also).

Chapter 7

Computational Complexity

→ Computational complexity theory is a branch of the theory of computation in computer science that focus on classifying computational problems according to their inherent difficulty and relating these classes to each other.

→ it involves classifying problems according to their inherent tractability or intractability i.e. whether they are easy or hard to solve.

→ It deals with resources required during computation to solve a given problem:-
(i) time (how many steps it takes to solve)
(ii) space (how much memory it takes)

* Lambda transition is the empty transition.

Space Complexity :-

⇒ Space Complexity of a program is the amt. of memory consumed (apart from i/p & o/p, if required its specification) until it completes its execution.

⇒ The way in which the amt. of storage space required by an algorithm varies with the size of the problem to be solved.

Time Complexity

⇒ Time Complexity of an algorithm is the amt. of time (no. of steps) needed by a program to complete its task (to execute a particular algorithm).

⇒ The way in which the number of steps required by an algorithm varies with the size of the problem it is solving. The time taken for an algorithm is comprised of two times:- (i) compilation time (ii) Run time