Mathematical Preliminaries

SETS

- A set is a well defined collection of objects
- First we specify a common property among things and then we gather up all the things that have this common property.
- It is simply things grouped together with a certain property in common.
- Eg:
  - Set of all students in a college
  - Types of fingers
  - Natural number's set. . . etc
- The individual objects are called members or elements of the set.
- We use capital letters like A, B, C. . . etc to denote a set.
- Also we use small letters like a, b, c, d to denote elements of a set.
- When a is an element of set A, we denote '$a \in A$'
- When a not an element of set A we denote '$a \notin A$.'

* Ways of describing a set

① By listing its elements

- In this we write all elements of the set, without repetition.
- Enclose them with in braces.
- Eg: • Set of all positive integers divisible by 15 and less than 100

⇒ {15, 30, 45, 60, 75, 90}

• Set of all letters from a to g

⇒ {a, b, c, d, e, f, g}

② By describing the properties of elements of the set.

- The description of property is called predicate
- The set is implicitly specified.
- Eg: • {15, 30, 45} can be represented as:

{n | n is a positive integer divisble by 15 and less than 50}

• {a, b, c, d}

as:

{n | n is an alphabet from a to d}

③ By recursion

- In this method we define elements by computational rule for calculating the elements.
- Eg: • Set of all natural numbers leaving a reminder 1 when divided by 3 can be described as

$$\{a_n \mid a_0 = 1, a_{n+1} = a_n + 3\}$$

* Subset

→ Set A is said to be a subset of B if every element of A also element of B

→ Represented as $A \subseteq B$

→ Eg: $A = \{1, 2, 3, 4, 5\}$

$B = \{1, 3, 4\}$

$B \subseteq A$ (because every element of B also element of A)

* Equal Sets

→ 2 sets A and B are said to be equal if all elements of A is equal to that of B or their members are same.

→ If $A = B$, then

$A \subseteq B$

$B \subseteq A$

→ Eg: $A = \{1, 2, 3\}$

$B = \{2, 1, 3\}$

$A = B$ (elements are same).

*Null set

→ A set with no element

→ Also called as null set / void set

→ denoted by ϕ

*Operations on set

- Union

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

→ Eg: let $A = \{1, 2\}$

$B = \{3, 4, 1\}$,

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

- Intersection

→ $A \cap B = \{x \mid x \in A \text{ and } x \in B\}$

→ Eg: let $A = \{a, b, c, d\}$

$B = \{b, d, e, a, f\}$

$A \cap B = \{a, b, d\}$

(common elements in A & B)

- **Complement**

→ $A - B = \{x \mid x \in A \text{ and } x \notin B\}$

→ U is universal set (set of all elements that are under consideration for a particular situation. For example the U set of numbers is set of all integers, natural numbers, negative numbers, fractional numbers and decimal numbers).

→ A complement of a set A is the set of elements that are in U but that are not in A.

→ Eg: $U = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$

$A = \{1, 4, 7, 8\}$

$A^c = A' = \{2, 3, 5, 6, 8, 9, 10\}$

ie, $\boxed{A^c = A' = U - A}$

* Power Set

→ Set of all subsets of a set A

→ Denoted by 2^A or $P(A)$

→ Eg:

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

$P(A) = \{\phi, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}\}$

* Ordered pair

→ Let A and B be 2 sets, the $A \times B$ is defined as $\{(a, b) \mid a \in A \text{ and } b \in B\}$

→ (a, b) is called ordered pair

→ Here (a, b) is diffnt from (b, a)

* Partition of Set

→ Let S be a set with elements $(A_1, A_2, A_3 \ldots A_n)$. Subsets of S is called partition if $A_i \cap A_j = \phi \ (i \neq j)$ and $S = \bigcup_{i=1}^{n} A_i$ ie, $(A_1 \cup A_2 \cup \ldots \cup A_n) = S$

→ Eg:

Let $S = \{1, 2, 3, 4, 5, 6, 7, 8\}$

$I = \{1, 3, 5, 7\}$

$J = \{2, 4, 6, 8\}$

here, I and J are partition sets of S because $i \neq j$ and $i \cap j = \phi$

also $I \cup J = S$

ie, there is no common elements in I and J

Also union of elements of I and J equal to set S.

Sets with one binary operation

→ A binary operation * on a set of S is a rule which assigns to every ordered pair (a,b) of elements from S, a unique element denoted by a * b.

→ Eg: Addition on set of all integers is a binary operation

⇒ let * ⇒ +

a = 2, b = 3

a * b ⇒ a+b

= 2+3 = 5

* Postulates of binary operation

① Closure

→ If a and b are in S, then a * b also in S.

→ Eg: S is a set of Integers

* be addition

let a = 2, b = 3

the a * b ⇒ a+b = 2+3 = 5

5 belongs to set S.

So this satisfies closure property

② Associativity

→ If a, b, c are in S, ~~then (a*b)*~~

then, $a * (b * c) = (a * b) * c$

→ Eg: $a = 2,\ b = 1,\ c = 3$

let * be addition

$\Rightarrow a*(b*c) = (a*b)*c$

$\Rightarrow 2+(1+3) = (2+1)+3$

$\Rightarrow 2+4 = 3+3$

$\Rightarrow \underline{6} = \underline{6}$

③ Identity Element

→ There exist a unique element called identity element e in S such that for any element x in S,

$x * e = e * x = x \quad ; x \in S$

→ Eg: let Z be set of integers and + be the operation

ie, (Z, +)

Now, additive identity of Z

⇒ we have set $Z = \{0, 1, 2, 3, 4 \cdots\}$

∴ take a number, let it be 2, and operation is +

we have $2 + e = e + 2 = 2$

$\Rightarrow e = 0$ (identity element w.r.t +)

ie, additive identity of set z is 0

⇒ Also we can find out multiplicative identity.

→ Eg: Let z be set of integers, and * be x

ie, $(z, *) \Rightarrow (z, x)$

$z = \{0, 1, 2, 3, 4 \cdots\}$

let $\overset{x}{a} = 2$, and operation is x.

$\therefore x * e = e * x = x$

$\Rightarrow 2 * e = e * 2 = 2$

$\Rightarrow 2 \times e = e \times 2 = 2$

$\Rightarrow e = 1$

$\therefore$ Multiplicative identity of z is 1

④ Inverse

→ For every element x in S there exist a unique element x' in S such that

$$x * x' = x' * x = e$$

→ x' is the inverse element of x with respect to '*'.

→ Eg:

Let $\mathbb{Z}$ be set of all real numbers and we have operation * addition

ie, $(\mathbb{Z}, +)$

⇒ let $x = 2$

$e = 0$ (identity element)(additive identity)

$x + x' = 0$

$\Rightarrow 2 + x' = 0$

$\Rightarrow 2 + (-2) = 0$

$\therefore \quad x' = -2$ ie, inverse of x is -2

⇒ -2 is additive inverse

⇒ let the operation be $(\mathbb{Z}, \times)$

⇒ let $x = 2$

$e = 1$ (multiplicative identity)

Now, $x \times x' = 1$ $(\because x * x' = e = x' * x)$

or $x' \times x = 1$

$\Rightarrow x' \times 2 = 1$

$\Rightarrow x' = 1/2$ (multiplicative inverse)

⑤ Commutativity

→ If $a, b \in S$, then $a * b = b * a$

→ Eg: Let the binary operation be $(Z, +)$

Now let

$a = 1, \; b = 2$

then

$a * b = a + b$

$b * a = b + a$

$\Rightarrow 1 + 2 = 2 + 1$

$\Rightarrow 3 = 3$

$\therefore (Z, +)$ satisfies commutative property

Semi Group

→ A set S with a binary operation $*$ is said to be semigroup if $(S, *)$ satisfies the following postulates

(1) Closure

(2) Associativity

→ Eg: Let Z be set of all integers and binary operation be $\times$.

then

(i) $a, b \in Z$ and $a \times b \in Z$

Eg: $1, 2 \in Z$ and $(1 \times 2) \in Z$

$\therefore$ closure property satisfied

(ii) Let

$a * (b * c) = (a * b) * c$

$\Rightarrow a \times (b \times c) = (a \times b) \times c$

$a = 1, \quad b = 2, \quad c = 3$

$\Rightarrow 1 \times (2 \times 3) = (1 \times 2) \times 3$

$\Rightarrow 1 \times 6 = 2 \times 3$

$\Rightarrow 6 = 6$

$\therefore (Z, \times)$ satisfies associativity property

→ Since $(Z, \times)$ satisfy postulates 1 and 2 (closure and associativity), it is semi group.

Monoid

→ A set S with a binary operation $*$ is called a monoid if the postulates 1, and 3 are satified.

ie, they satisfies closure associativity and exist an identity element. ie,

(1) closure
(2) associativity } semigroup
(3) Identity element.

→ Eg: Take the example of $(z, +)$

- first we check whether $(z, +)$ satisfies the 3 postulates

- let $a = 1$; $b = 2$ $(a, b \in z)$

- $a + b = 1 + 2 = 3$, $3 \in z$

∴ closure property satisfied

- Now,

$a = 1, b = 2, c = 3$

$a * (b * c) = (a * b) * c$

$\Rightarrow 1 + (2 + 3) = (1 + 2) + 3$

$\Rightarrow 1 + 5 = 3 + 3$

$\Rightarrow \underline{6} = \underline{6}$

∴ associativity property satisfied.

- Lastly,

lets find out the identity element e such that $x * e = e * x = x$

$\Rightarrow x + e = e + x = x$

$\Rightarrow$ let $x = 2$,

$x + e = 2 + e = 2$

$\Rightarrow \underline{e = 0}$ ∴ identity element exist

Abelian ~~Monoid~~ Semigroup

→ A set S with binary operation * is called abelian monoid if the below postulates are satisfied.

(1) closure
(2) ~~Semi group~~ Associativity } Semigroup
(3) Commutativity.

Abelian monoid

→ A set S with binary operation * is called abelian monoid if it satisfies:

(1) closure
(2) Associativity
(3) Identity element
(4) Commutativity

Group

→ A set S with binary operation * is called a group if it satisfies

(1) closure
(2) Associativity
(3) ~~Monoid~~ Identity element } Monoid
(4) Inverse element.

Abelian Group

→ A set S with binary operation * is called abelian group if it satisfies

(1) closure
(2) Associativity
(3) Identity element
(4) Inverse element
(5) Commutativity.

→ Eg: $(\mathbb{Z}, +)$

Sets with 2 binary Operations

→ We come across sets with 2 binary operations defined on them in some cases.

→ Let S be a set with 2 binary operations "*" and "•"

→ We have 11 postulates:

① Closure for *
② Associativity for *
③ Identity element for *
④ Inverse element for *
⑤ Commutativity for *
⑥ closure for •
⑦ Associativity for •

⑧ Identity element for '·'

⑨ Inverse element for '·'

ie, If S under * satisfies the postulates 1-5 then for every $x \neq e$, there exists a unique element x' in S such that $x' \cdot x = x \cdot x' = e'$, where e' is the identity element corresponding to '·'.

⑩ Commutativity with respect to ·

⑪ Distributivity

→ For every a, b, c in S

$$a \cdot (b * c) = (a \cdot b) * (a \cdot c)$$

⇔ Algebraic system:

→ A set with one or more binary operation is called an algebraic system

→ Example for algebraic system with one binary operation are:

- Groups
- Monoids
- Semigroups
- Abelian groups ...etc

→ Example for algebraic group with 2 binary operation are:

- Ring
- Field ..etc.

Ring

→ A set with 2 binary operation * and • is called a ring if:

(1) it is an abelian group w.r.t *

(2) • satisfies closure, associativity and distributivity postulates.

→ A ring is called commutative ring if commutativity postulate is satisfied by •

→ If a commutative ring satisfies identity postulate, it is called commutative ring with unity

Field

→ It is a set of 2 binary operations * and • if it satisfies the 11 postulates.

→ ie, it is an abelian group and satisfies closure, associativity, commut identity element and inverse w.r.t • and also satisfies distributive property

Eg:

(i) $\mathbb{Z}$ with addition and multiplication is a commutative ring with identity

Proof:

→ We have to prove $(\mathbb{Z}, +, \times)$ is a commutative ring with identity.

→ To prove this, we have to show that $(\mathbb{Z}, +)$ is an abelian group and $(\mathbb{Z}, \times)$ satisfies closure, associativity, identity element and commutativity properties. Also $(\mathbb{Z}, +, \times)$ satisfies distributive property.

→ Now consider $(\mathbb{Z}, +)$

- let $a = 2$, $b = 3$

$a + b = 2 + 3 = 5$

$a \in \mathbb{Z}$
$b \in \mathbb{Z}$
$a * b \in \mathbb{Z}$

∴ closure property satisfied for $(\mathbb{Z}, +)$

- let $a = 2$
  $b = 3$
  $c = 1$

$(a * b) * c) = a * (b * c)$

$\Rightarrow (2 + 3) + 1 = 2 + (3 + 1)$

$\Rightarrow 6 = 6$.

∴ $(\mathbb{Z}, +)$ is associative

- let $x = 2 \in z$

$2 * e = e * 2 = 2 \quad (x * e = e * x = x)$

$2 + e = e + 2 = 2$

$\Rightarrow e = 0$

$\therefore$ Identity element of $(z, +)$ is 0

- Now, let $x = 5 \in z$

$x * x' = x' * x = e$

$\Rightarrow 5 + x' = x' + 5 = 0 \quad (\text{since } e = 0)$

$\Rightarrow x' = -5$

$\therefore$ Inverse element of 5 w.r.t $(z, +)$ is -5

- To show $(z, +)$ is commutative

$a, b \in z$

then $a * b = b * a$

let $a = 2$

$b = 3$

$2 + 3 = 3 + 2 = 5$

$\therefore (z, +)$ is commutative

→ All this shows that $(z, +)$ is an abelian group.

→ Next, consider (Z, x)

- let $a = 2 \in Z$, $b = 5 \in Z$
  $a \in Z$, $b \in Z$, $a * b \in Z$ } closure postulate

  $a * b = a \times b$

  $a \times b = 2 \times 5 \in Z$

  $= \underline{10} \in Z$

  ∴ closure property satisfied for (Z, x)

- Now
  $a = 2$
  $b = 5$
  $c = 1$

  $(a * b) * c = a * (b * c)$ (associativity postulate)

  $\Rightarrow (a \times b) \times c = a \times (b \times c)$

  $\Rightarrow (2 \times 5) \times 1 = 2 \times (5 \times 1) = \underline{10}$

  ∴ Associativity postulate satisfied for (Z, x)

- We have
  $a = 2$
  $b = 3$

  $a \times b = b \times a$ (commutativity postulate)

  $\Rightarrow 2 \times 3 = 3 \times 2 = 6$

  ∴ Commutativity satisfied for (Z, x)

=

- To check whether identity element exist for $(z, *)$

$$x * e = e * x = x$$

$$\Rightarrow x \times e = e \times x = x$$

Let $x = 10$

$$\Rightarrow 10 \times e = e \times 10 = 10$$

$$\Rightarrow e = 1$$

- Now, to check distributive property

$$\Rightarrow a \cdot (b * c) = (a \cdot b) * (a \cdot c)$$

$$\Rightarrow \text{let } a = 1,\ b = 2,\ c = 3$$

$$\Rightarrow a \times (b + c) = (a \times b) + (a \times c)$$

$$\Rightarrow 1 \times (2 + 3) = (1 \times 2) + (1 \times 3)$$

$$\Rightarrow 5 = 5$$

also

$$\Rightarrow (a + b) \times c = (a \times c) + (b \times c)$$

$$\Rightarrow (1 + 2) \times 3 = (1 \times 3) + (2 \times 3)$$

$$\Rightarrow 3 \times 3 = 3 + 6$$

$$\Rightarrow 9 = 9$$

$\therefore$ distributive property satisfied

Hence $(z, +, \times)$ is a commutative ring with identity

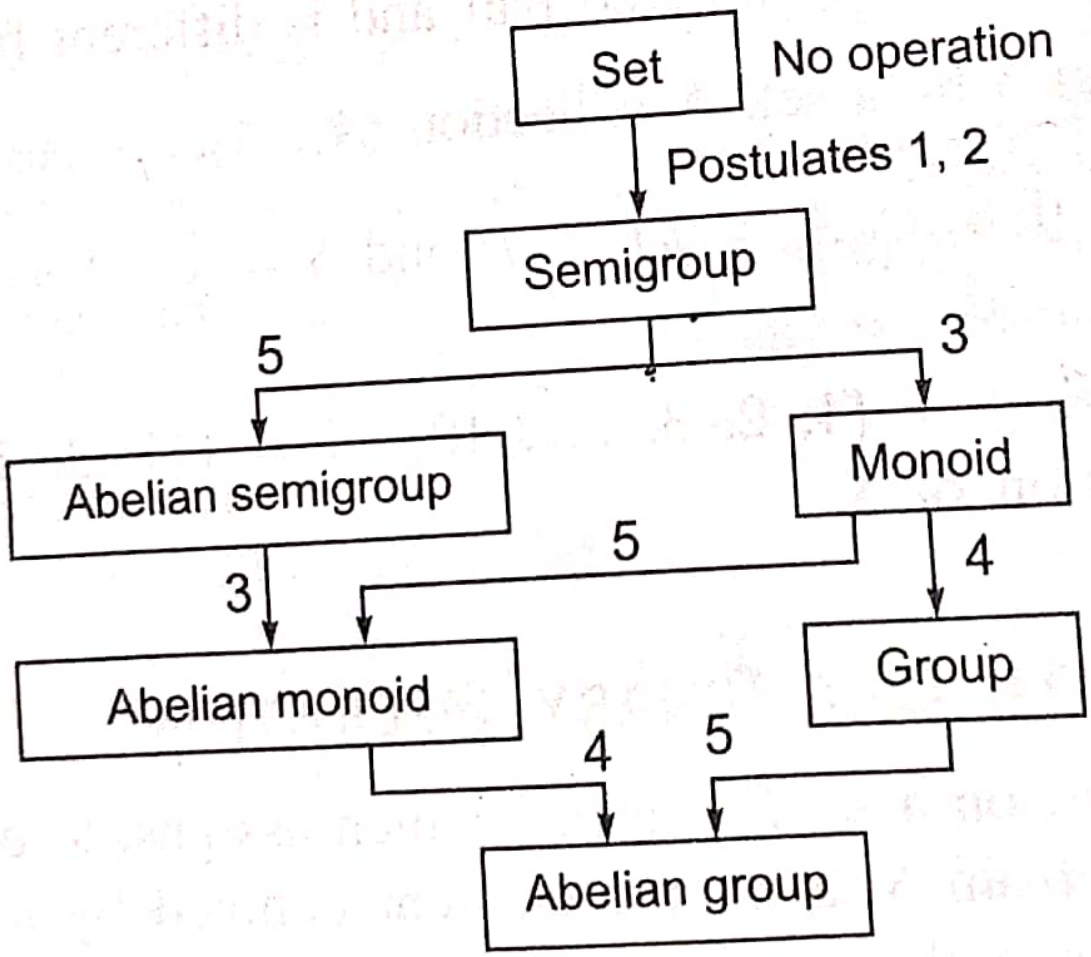


Fig. 2.1 Sets with one binary operation.

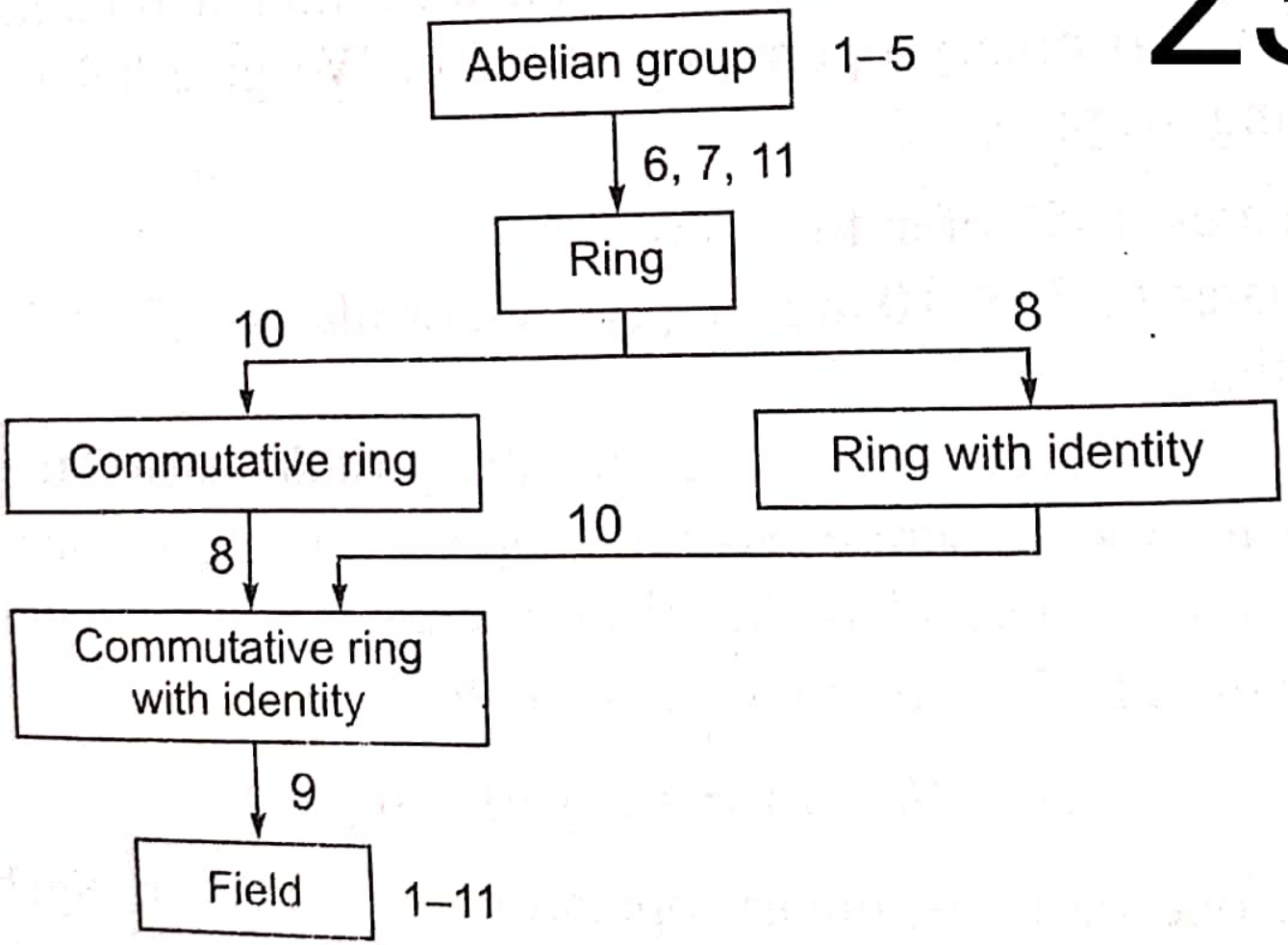


Fig. 2.2 Sets with two binary operations.

Relations

→ This concept is basic in computer science as well as in real life.

→ It arises when we consider a pair of objects and compare one with other.

→ Eg: 'being daughter of' is a relation b/w 2 persons.
We can represent this as ordered pair (a, b), ie a is daughter of b.

→ In computer science, this relation concept arises often in case of data structures.

→ A relation R in a set S is a collection of ordered pairs of elements in (ie, a subset of S×S). When (x, y) is in R, we write xRy. when (x, y) not in R, we write $x R^1 y$.

→ Eg:

⊙ Construct a relation from the set $S = \{1, 2, 3, 4\}$ defined by xRy if $x > y$

⇒ $R = \{(2,1)(3,2)(4,1)(4,2)(4,3),(3,1)\}$

② Construct a relation R from given set

$A = \{1, 3, 5, 7\}$

$B = \{2, 4, 6\}$

defined by set of $\{(x, y) \in A \times B \mid x + y = 9\}$

$\Rightarrow R = \{(3,6)(5,4)(7,2)\}$

Properties of Relation

① Reflexive

→ A relation R in S is reflexive if xRx for every x in S.

→ Eg:

$S = \{1, 2, 3, 4\}$

$R = \{(1,1)(2,2)(3,3)(4,4)\}$

② Symmetric

→ A relation R in S is symmetric if for x, y in S, yRx whenever xRy

→ Eg:

$S = \{1, 2, 3, 4\}$

$R = \{(1,1)(1,2)(2,1)(3,4)(4,3)\}$

③ Transitive

→ A relation R in S is transitive if for x, y and z in S, xRz whenever xRy and yRz

→ Eg:

$S = \{1, 2, 3, 4\}$

$R = \{(1,2)(2,3)(1,3)(3,4)$ ~~$(4,4)(2,2)$~~

$(4,2)(3,2)\}$

Q) Check the given relation R on set S is symmetric, reflexive or transitive

$S = \{1, 2, 3, 4, 5, 6\}$

$R = \{(1,2)(2,3)(3,4)(4,4)(4,5)\}$

A)→ Since there is no reflexive image for the elements 1, 2, 3 and 5 the given relation is not reflexive. ie, elements are not related to itself.

→ Since there no relation yRx whenever xRy, it is also not symmetric

→ This relation is not transitive as for every xRy & yRz, no relation xRz.

→ So the given relation R for set S is not reflexive, not symmetric, and not transitive.

Types of relations

1. Reflexive relation
2. Symmetric relation
3. Transitive relation
4. Equivalence relation

→ A relation is said to be equivalence if it is reflexive, symmetric and transitive.

→ Eg: $S = \{1, 2, 3\}$

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

5. Empty relation / void relation

→ A relation R on set S is called empty if there is no relation b/w any elements of a set.

→ Eg: $S = \{1, 2, 3, 4\}$

$R = \{(x, y) \mid |x - y| = 8\}$

→ In this example there is no element satisfies the relation R where $x - y = 8$.

② Symmetric closure

- let R be a relation on set A, reflexive closure of relation R on set A, let R^{-1} be the inverse of R
- The symmetric closure of relation R on set A is

$$R \cup R^{-1}$$

- Eg: $A = \{1, 2, 3, 4\}$

$R = \{(1,1)(1,4)(2,3)(3,1)(3,4)\}$

$R^{-1} = \{(1,1)(1,3)(3,2)(4,1)(4,3)\}$

$R \cup R^{-1} = \{(1,1)(1,3)(1,4)(2,3)(3,1)$
$(3,2)(3,4)(4,1)(4,3)\}$

③ Transitive closure

- Let R be a relation on set A

$$R^+ = \bigcup_{n=1}^{\infty} R^n$$

- Transitive closure of R is R^+

ie, $R^* = R^1 \cup R^2 \cup R^3 \cdots \cup R^{n-1}$

- Eg: $A = \{1, 2, 3, 4\}$

$R = \{(1,1)(1,4)(2,3)(3,1)(3,4)\}$

$R^1 = \{(1,1)(1,4)(2,3)(3,1)(3,4)\}$

$R^2 = \{(1,1)(1,4)(2,1)(2,4)(3,1)(3,4)\}$

$R^3 = \{$ " " " $\}$

⑥ Antisymmetric relation

→ A relation R on set S is called anti-symmetric if $(a,b) \in R$ and $(b,a) \in R$ then $a = b$.

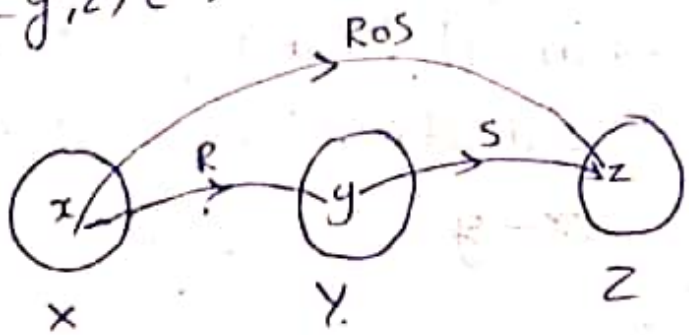
ie, $R = \{(a,b) \to R \mid a \leq b\}$

$\Rightarrow a \leq b, b \leq a \Rightarrow a = b$

⑦ Composite relation

→ Suppose, we have 3 sets A, B and C, a relation R from set A to B and a relation S defined from B to C then we can define a new relation known as composition of S (RoS)

→ Let R be a relation from x to y and S be a relation from y to z, then composite relation of R and S is the relation consisting of ordered pair (x,z) where $x \in X$ and $z \in Z$ and there exist an element $y \in Y$ such that $(x,y) \in R$ and $(y,z) \in S$



⑧ Universal relation

⑨ Asymmetric relation

Closure of relations

→ A given relation may not be reflexive symmetric or transitive, by adding more ordered pairs to R, we can make it reflexive, symmetric or transitive

→ For eg: let $S = \{1, 2, 3\}$

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

R is not reflexive as $3 R' 3$ i.e. $(3,3)$

by adding $(3,3)$ to R,

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

R became reflexive.

→ There are mainly 3 types of closure

① Reflexive closure

→ $\Delta = \{(a,a) \mid a \in A\}$ is the diagonal relation on set A

- Reflexive closure of relation R on set A is $R \cup \Delta$

- Eg: $A = \{1, 2, 3, 4\}$

$R = \{(1,1)(1,4)(2,3)(3,1)(3,4)\}$

$\Delta = \{(1,1)(2,2)(3,3)(4,4)\}$

$R \cup \Delta = \{(1,1)(2,2)(1,4)(2,3)(3,1)$
$(3,3)(3,4)(4,4)\}$

- Since $R^2 = R^3$ we stop process.
- $R^* = R^1 \cup R^2 = \{(1,1)(1,4)(2,1)(2,3)$
  $(2,4)(3,1)(3,4)\}$

---

* Definition:

Let R be an equivalence relation on set S, Let $a \in S$ the C_a (Equivalence class containing a) is defined as

$$C_a = \{b \in S \mid aRb\}$$

Functions

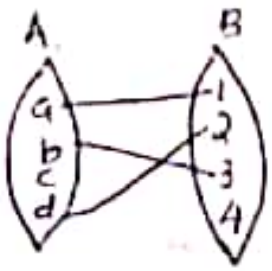
→ Functions can be one-to-one relations or many to one relations.

- Domain:
  - Set of values to which the rule is applied.
  - It is the complete set of possible values of independent variable
  - Or all the possible x-values which will make function work.

- Co domain
  - Set of all y values.

- Range
  - It is the complete set of all possible resulting values of domain or the set of values that function can produce

→ Eg:



here,
Range = {1, 2, 3}
Domain = {a, b, d}
Codomain = {1, 2, 3, 4} = {B}

→ A function or map f from a set X to a set Y is a rule which associates to every element x in X a unique element in Y, which is denoted by $f(x)$. The element $f(x)$ is called image of x under f. And the function is denoted by $f : X \to Y$

→ Example of a function

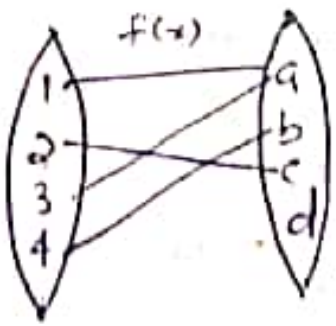
$f : \{1, 2, 3, 4\} \to \{a, b, c\}$

$f(1) = a$

$f(2) = b, c$

$f(3) = a$

$f(4) = b$



* Types of functions:

① One - to - one / Injective function

→ $f: X \to Y$ is said to be one to one or an injective function if different elements in X have different images

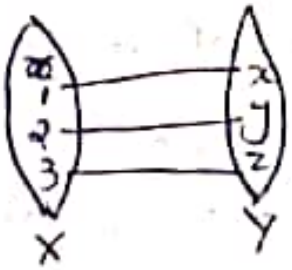
ie, $f(x_1) \neq f(x_2)$ when $x_1 \neq x_2$

→ To prove f is one to one, we assume $f(x_1) = f(x_2)$ and show that $x_1 = x_2$.

→ Example of injective function

$f: \{1, 2, 3\} \to \{x, y, z\}$

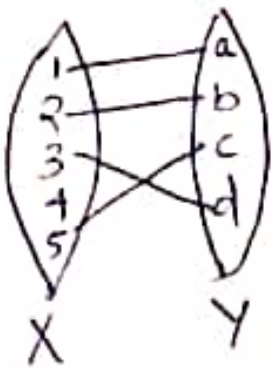
$f(1) = x$, $f(2) = y$, $f(3) = z$



② Onto / surjective function

→ $f: X \to Y$ is onto if every element y in Y is the image of some element x in X.

→ Eg:



$f: \{1,2,3,4,5\} \to \{a, b, c, d\}$

$f(1) = a$, $f(2) = b$, $f(3) = d$

$f(5) = c$

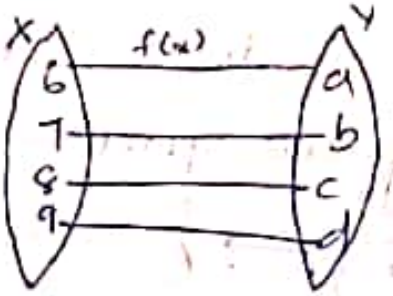
③ Bijective function

→ $f: X \to Y$ is said to be one to one correspondence or bijection if f is both injective and surjective or one to one and on to function

→ Eg: $f: \{6, 7, 8, 9\} \longrightarrow \{a, b, c, d\}$

$f(6) = a,\ f(7) = b,\ f(8) = c,\ f(9) = d$

⇒



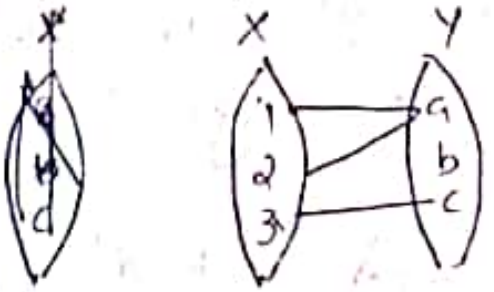
④ Many to one function

→ A function $f: X \to Y$ is said to be many to one if there are y values that have more than one x value mapped onto them

→ Eg: $f: \{1, 2, 3\} \longrightarrow \{a, b, c\}$ ⇒ $f(1) = a$

$f(2) = a$

$f(3) = c$



Many to one

⑤ One to many function

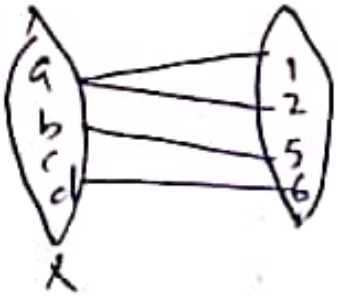
→ A function is said to be one to many if if there are x values that have more than one y value mapped.

→ Eg: $f: \{a, b, c, d\} \longrightarrow \{1, 2, 5, 6\} \Rightarrow f(a) = 1$

$f(a) = 2$

$f(b) = 5$

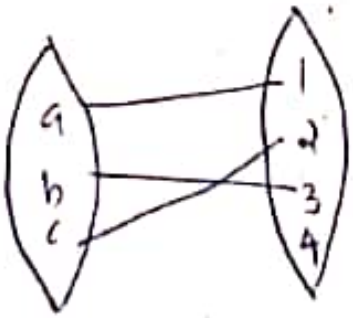
$f(d) = 6$



⑥ Into function

→ A function $f: A \to B$ is said to be into if there exist even a single element in B having no pre-image in A, then f is said to be an into function.

→ Eg.



→ There are many other functions like polynomial function, quadratic function, algebraic function, lineare function .. etc.

Pigeonhole Principle

If n objects are distributed over m places and $n > m$, then some place receives at least 2 objects.

→ Example

(1) Suppose a postman distributes 51 letters in 50 mailboxes (pigeonholes). Then it is evident that some mailbox will contain at least 2 letters. This enunciated as a mathematical principle called the pigeonhole principle.

(2) Suppose that a flock of 20 pigeons flies into a set of 19 pigeonholes. Since there are 20 pigeons and only 19 pigeonholes, at least one of these 19 pigeonholes must have atleast 2 pigeons in it.

→ In the case of function, we can say as if $n+1$ objects are put into n boxes, then at least one box contain 2 or more objects.

- The abstract formulation of the principle :

let X and Y be finite sets and let
f : A → B be a function

→ If X has more elements than Y, then f is not one to one

→ If X and Y have the same no: of elements and f is onto, then f is one to one.

→ If X and Y have the same no: of elements and f is one to one, then f is onto.

Graph

→ Graph theory is widely applied in many areas of computer science like formal languages, compiler writing, artificial intelligence (AI) .. etc.

→ A graph is a mathematical structure used to model pairwise relations b/w objects which is made up of vertices and edges.

→ A graph is an ordered pair
G = (V, E)

It comprises,

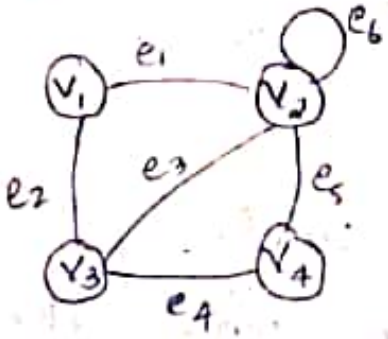
- V → a set of vertices / nodes / point

- E → Set of edges / links / lines.
- Loop → It is an edge that joins a vertex to itself.

* Un-directed graph

→ It is represented by a diagram where the vertices are represented by points or small circles and edges by arcs joining the vertices of associated pair

→ Eg:



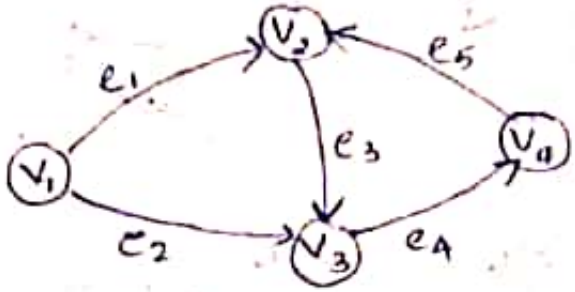
- $\{V_1, V_2\}$ associated with e_1
- $\{V_2, V_2\}$ associated with e_6 (self loop) ...ac

* Directed graph.

→ A directed graph or 'digraph' consist of

- A non empty set V (set of vertices)
- A set of E (edges)
- A map ψ which assign to every edge a unique ordered pair of vertices.

→ Eg:



* Successor : If (v_i, v_j) is associated with an edge e, then v_i and v_j are called end vertice of e and v_j is the successor of v_i.

* Predecessor : v_i is the predecessor of v_j

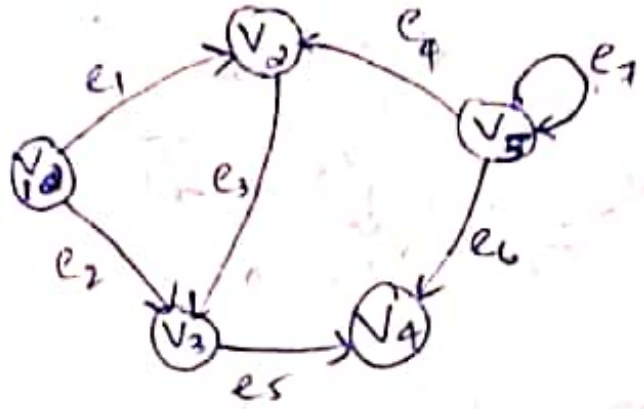
: In the above example,
v_2 is the successor of v_1
v_3 is the predecessor of v_4 ..et

* degree of vertex.

→ It is the no: of edges with v as an end vertex. A self loop is counted twice while calculating the degree.

→ The no: of vertices of odd degree in any graph is even.

→ Eg:



here degree of $V_2 = 3$

$V_3 = 3$

$V_5 = 4$

* Path in a graph

→ It is an alternating sequence of vertices and edges of the form $v_1 e_1 v_2 e_2 \ldots v_{n-1} e_{n-1} v_n$

→ It is the path from v_1 to v_n.

* Connected graph

→ If there is a path b/w every pair of vertices, graph is called connected

* Circuit in a graph

→ It is an alternating sequence $v_1 e_1 v_2 e \ldots e_{n-1} v_1$ of vertices and edges starting and ending at same vertex such that e_1 has v_i and v_{i+1} as the end vertices and no edge or vertex other than v_1 is repeated.

~~Eg: for above graph;~~

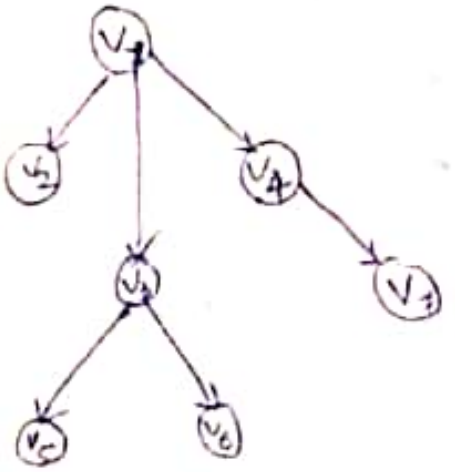
5 If a connected graph with n vertices and has n-1 edges, then it is a tree

6 If a graph with no circuits has n vertices and n-1 edges, then it is a tree.

* Leaf of a tree
  → It can be defined as a vertex of degree one.

* Internal vertices
  → Vertices other than leaves are called internal vertices.

→ Eg:



v_2, v_5, v_6, v_7 – leaf

v_1, v_3, v_4 – Internal vertices.

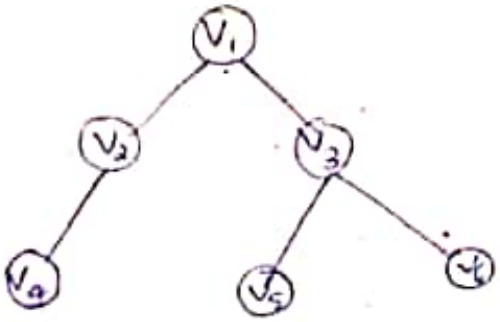
* Ordered directed tree:
  → It is a digraph which satisfies the following conditions:

  T_1: There is one vertex called root of tree which is distinguished from all other vertices and has no predecessor

T_2 : There is a directed path from root to every other vertex

T_3 : Every vertex except the root has exactly one predecessor

T_4 : Succesors of each vertex are ordered from the left.



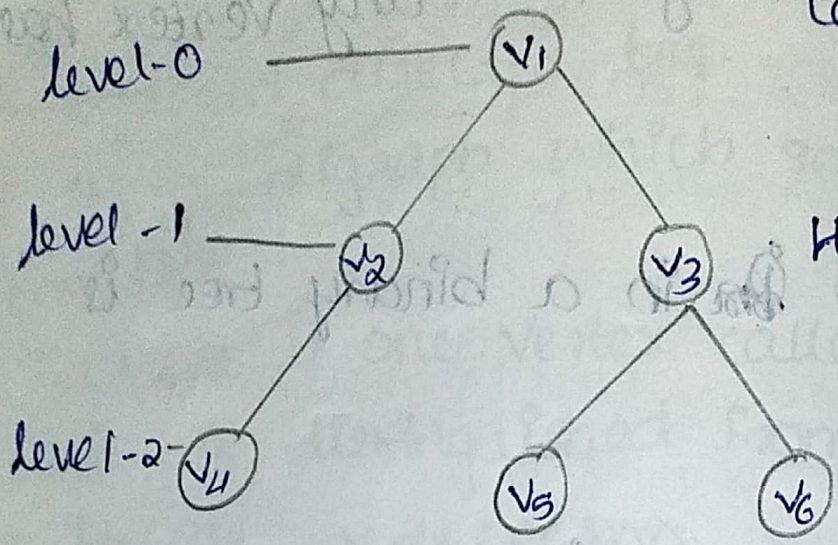
* Binary tree
  → It is a tree in, which the degree of root is 2 and remaining vertices are of degree one or three
  → In a binary tree, any vertex has atmost 2 successors.

• Theorem:
  → The no: of vertices in a binary tree is odd.
  → Proof :
  - Let n be the no: of vertices
  - root has degree 2
  - Remaining n-1 vertices are of odd degree.

we have the theorum that no. of vertices of odd degree in any graph is even. since (n-1) is even and the by above theorum n is odd.

—some more terminologies.

* A son of a vertex v is successor of v.
* The father of v is predecessor of v.
* If there is a directed path from v_1 to v_2, v_1 is called the ancestor of v_2 & v_2 is called the descendt & of v_1.
* No. of edges in a path is called length of path.
* Height of tree is the length of longes past from the root.



Length of $v_1 - v_4$ = 2

Height of tree = 2

1) Prove that no. of leaves in a binary tree T is $(\frac{n+1}{2})$ where n is the no. of vertices?

Let 'm' be no. of leaves in a tree

no. of leaves = m

no. of vertices = n

Degree of root = 2 (since its binary tree)

remaining vertices has (n-m-1)

degree = 3

No. of edges = n-1

(From theorem that if a binary tree has n vertices then it has n-1 edges)

Edges are counted twice while calculating degree of its end vertices i.e,

$2(n-1)$.

$2(n-1) = 2+m+3(n-3-1)$

$2n-2 = 2+m+3n-3m-3$

$-n = 1-2m$

$m = \left(\frac{n+1}{2}\right)$

hence proved

— strings.

A string over an alphabet set 'Σ' is a finite sequence of symbols from 'Σ'.

$\Sigma \Rightarrow$ set of symbols called alphabets. It includes letters, numbers, special characters ....etc.

eg: $\{0,1\}$ → binary alphabet

$\{a,b,c\}$

A string is generally notes as 'w'

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

$w_1 = 0000$

$w_2 = 0001$

$w_3 = 0010$ etc

- Length of a string.

Its denoted by $|W|$ & is defined as the no. of positions for symbols in the string.

eg: $W = 010011$

$|W| = 6$

- Empty string.

It denoted by zero occurance of symbols & is represented as ϵ or λ

- Language / L

It is the set of string all of which are chosen from Σ^* where Σ is a particular alphabet.

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

$L = \{01, 001, 11, 00, 110\}$

- Closure of string.

* Klean closure (Σ^*)

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

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

$\Sigma^* = \{\epsilon, 0, 1, 11, 00, 111, 100, 1100 \ldots\}$

L is the subset of Σ^*

- positive or + closure (Σ^+)

The set of all strings from an alphabet Σ except ϵ

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

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

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

- operations on strings.

The basic operation for strings is binary concatenation.

* concatenation.

Its the process of joining strings. Let W_1 & W_2 be a string in Σ^*. Now, $W_1 \cdot W_2$ = concatenated result of W_1 & W_2.

eg; $W_1 = abc$

$W_2 = efg$

$W_1 \cdot W_2 = abcefg$

- properties of concatenation.

* Associative

concatenation on a set Σ^* is associative

since for each x, y, z in Σ^*

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

$x = a$

$y = b, b$

$z = b, a$

$(xy)z = abbba$

$x(yz) = abbba$

$\Rightarrow (xy)z = x(yz)$

* Identity element

The set Σ^* has an identity element $\in$ with respect to binary operation of concatenation as $x \in = \in x = x$ for every x in Σ^*.

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

$x = ab$

$ab\in = \in ab = ab$

* Left cancellation & right cancellation property.

For x, yz in Σ^*, it has left cancellation & right cancellation property.

eg: $zx = zy \Rightarrow x = y$ (Left cancellatn)

$xz = yz \Rightarrow x = y$ (right cancellatn)

* For x, y in Σ^* we have,

$|xy| = |x| + |y|$

That is length of xy ($|xy|$) is equal to sum of length of x ($|x|$) & y ($|y|$).

— principle of induction

The process of reasoning from general observatns to specific truths is called inductn.

The following properties apply to the set N. of natural no. and the principle of inductn:

i) zero is a natural no.

ii) The successor of any natural no. is also a natural no.

iii) zero is not the succer of any natural no.

iv) No 2 natural no. have the same succer.

v) Let a prpsty $P(n)$ be defined for every natural no. n. If

1) $P(0)$ is true & (2) P(successor of n) is true whenever $P(n)$ is true, then $P(n)$ is true for

all n.

— Method of proof by induction.

This method consists of three basic steps.

Step 1: Prove P(n) for n = 0/1. This is called the proof for the basis.

Step 2: Assume the result / Properties for P(n). This is called the induction hypothesis.

Step 3: Prove P(n+1) using the induction hypothesis.

1) P.T $1+3+5+\ldots+r = n^2$. For all $n \geq 0$. Where 'r' is an odd & n is the no. of items in the sum (note: $r = 2n-1$).?

* a) Proof for the basis

for n = I, L.H.S = 1

& RHS = $1^2 = 1$

Hence the result is true for n = 1

b) By induction hypothesis we have

$1+3+5+\ldots r = n^2$. As $r = 2n-1$

LHS = $1+3+5+\ldots+2(n-1) = n^2$

c) we have to P.T $1+3+5+\ldots r+r+2 = (n+1)^2$

LHS = $(1+3+5+\cdots+n)+(n+2)$.

$= n^2+n+2 = n^2+2n-1+2 = (n+1)^2 = RHS$

— proof by contradictn.

suppose we want to prove a property 'P' under certain ~~contains. The~~ conditions. The method of proof by contradictn is as follows:

Assume that property P isn't true. By logical reasoning to get a conclusion which is either absurd or ~~contradiction~~ contradicts the given conditns.

MOD-2 FORMAL LANGUAGES

- Formal languages.

The theory of formal language finds its applicability extensively in the fields of C.S. Noam chomsky gave a maths model of grammar in 1956 which is effective for writing computer language.

— Grammar.

A grammar G can be formally written as a 4-tuple (N, T, S, P) where:

* N or Vn is set of variables or non-terminal symbols.
* T or Σ is a set of terminal symbols.
* S is a special variable called the start symbol $S \in N$
* P is production rules for terminals & Non-terminals.

A production rule has the form

$\alpha \rightarrow \beta$, where α & β are strings on Vn

$\cup \in$ & least one symbol of α belongs to V_N.

eg: 1) Grammar G_1:

$(\{S, A, B\}, \{a, b\}, S, \{S \to AB, A \to a, B \to b\})$

Here,

S, A & B are non-terminal symbols.

a & b are terminal symbols.

S is the start symbol $S \in N$

Productns $P: S \to AB, A \to a, B \to b$.

2) Grammar G_2:

$(\{S, A\}, \{a, b\}, S, \{S \to aAb, aA \to aaAb, A \to \varepsilon\})$

Here,

S & A are non-terminal symbols

a & b are terminals symbols.

ε is an empty string.

S in the start symbol. $S \in N$

Productn $P: S \to aAb. aA \to aaAb, A \to \varepsilon$.

— Derivatn from a Grammar.

Strings may be derived from other strings using the productns in grammar. If grammar 'G' has a productn $\alpha \to \beta$, we can say that

$x\alpha y$ derives $x\beta y$ in G. This derivation is written as.

$$x\alpha y \Rightarrow_G x\beta y$$

eg: Let us consider the grammar

$G_2 = (\{S, A\}, \{a, b\}, S, \{S \to aAb,\ aA \to aaAb,\ A \to \varepsilon\})$

some of the strings that can be derived are.

$S \Rightarrow \underline{aA}b$ using production $S \to aAb$

$\Rightarrow aa\underline{A}bb$ using production $aA \to aAb$

$\Rightarrow aaa\underline{A}bbb$ using production $aA \to aAb$

$\Rightarrow aaabbb$ using production $A \to \varepsilon$

The set of all strings that can be derived from a grammar is said to be the language generated from that grammar G is a subset formally defined by.

$$L(G) = \{W / W \in \Sigma^*, S \Rightarrow_G W\}$$

If $L(G_1) = L(G_2)$, the grammar G_1 is equivalent to the grammar G_2.

eg: If there is a grammar.

$G: N = \{S, A, B\}$ $T = \{a, b\}$ $P = \{S \to AB, A \to a, B \to b\}$

Here S produces AB, & we can replace A by a & B by b. Here the only accepted string is ab

i.e, $L(G) = \{ab\}$

eg: suppose we have the following grammar

$G: N = \{S, A, B\}$ $T = \{a, b\}$ $P = \{S \to AB, A \to aA, B \to bB | b\}$

The language generated by this grammar:

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

$= \{a^m b^n : m \geq 1 \text{ & } n \geq 1\}$

- construction of a grammar generating a language.

We'll consider some languages & convert it into a grammar G which produces those languages.

i) suppose, $L(G) = \{a^m b^n : m \geq 0 \text{ & } n > 0\}$

we have to find out the Grammar G which Produces $L(G)$?

since $L(G) = \{a^m b^n : m \geq 0 \text{ & } n > 0\}$

the set of strings accepted can be rewritten

as:

$L(a) = \{b, ab, bb, aab, abb, \ldots\}$

Here the start symbol has to take atleast one 'b' preceded by any no. of 'a' to accept the string set $\{b, ab, bb, aab, abb, \ldots\}$ we have taken the productns:

* $S \to aS$, $S \to B$, $B \to b$ & $B \to bB$
* $S \to B \to b$ (accepted)
* $S \to B \to bB \to bb$ (accepted)
* $S \to aS \to aB \to ab$ (accepted)
* $S \to aS \to aaS \to aaB \to aab$ (accepted)
* $S \to aS \to aB \to abB \to abb$ (accepted)

Thus, we can prove single string in $L(a)$ is accepted by the language generated by the production set. Hence the grammar.

$G = (\{S, A, B\}, \{a, b\}, S, \{S \to aS \mid B, B \to b \mid bB\})$

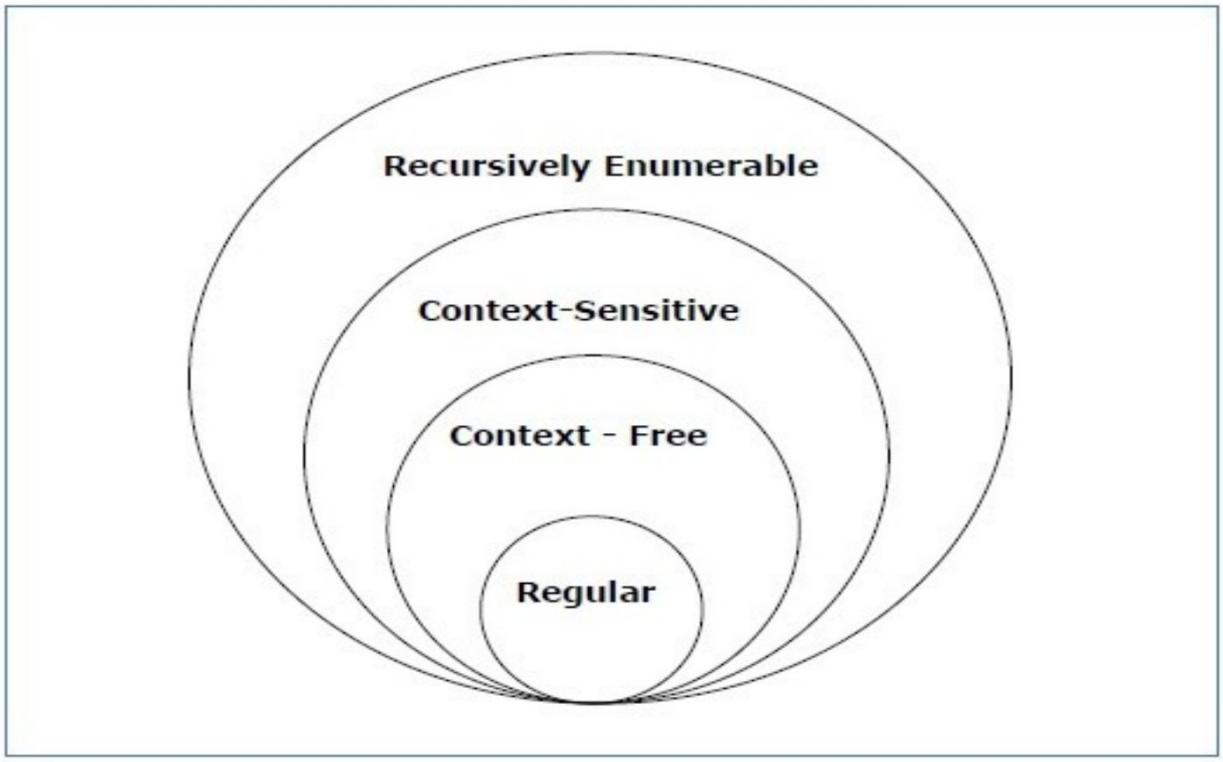
CHOMSKY CLASSIFICATION OF LANGUAGE

Chomsky Classification of Grammars

According to Noam Chomosky, there are four types of grammars − Type 0, Type 1, Type 2, and Type 3. The following table shows how they differ from each other

Grammar Type	Grammar Accepted	Language Accepted	Automaton
Type 0	Unrestricted grammar	Recursively enumerable language	Turing Machine
Type 1	Context-sensitive grammar	Context-sensitive language	Linear-bounded automaton
Type 2	Context-free grammar	Context-free language	Pushdown automaton
Type 3	Regular grammar	Regular language	Finite state automaton

Take a look at the following illustration. It shows the scope of each type of grammar –



Type - 3 Grammar

Type-3 grammars generate regular languages. Type-3 grammars must have a single non-terminal on the left-hand side and a right-hand side consisting of a single terminal or single terminal followed by a single non-terminal. The productions must be in the form **X → a or X → aY**

where **X, Y ∈ N** (Non terminal) and **a T** (Terminal)

The rule **S → ε** is allowed if **S** does not appear on the right side of any rule.

Example

$$X \rightarrow \varepsilon$$

$$X \rightarrow a \mid aY$$

$$Y \rightarrow b$$

Type - 2 Grammar

Type-2 grammars generate context-free languages. The productions must be in the form

$$A \rightarrow \gamma$$

where $A \in N$ (Non terminal) and $\gamma \in (T \cup N)^*$ (String of terminals and non-terminals).

These languages generated by these grammars are be recognized by a non-deterministic pushdown automaton

Example

$$S \rightarrow X\ a$$

$$X \rightarrow a$$

$$X \rightarrow aX$$

$$X \rightarrow abc$$

$$X \rightarrow \varepsilon$$

Type - 1 Grammar

Type-1 grammars generate context-sensitive languages. The productions must be in the form

$$\alpha A \beta \rightarrow \alpha \gamma \beta$$

where $A \in N$ (Non-terminal) and $\alpha, \beta, \gamma \in (T \cup N)^*$ (Strings of terminals and non-terminals)

The strings α and β may be empty, but γ must be non-empty.

The rule $S \rightarrow \varepsilon$ is allowed if S does not appear on the right side of any rule. The language generated by these grammars are recognized by a linear bounded automaton.

Example

$$AB \rightarrow AbBc$$

$$A \rightarrow bcA$$

$$B \rightarrow b$$

Type - 0 Grammar

Type-0 grammars generate recursively enumerable languages. The productions have no restrictions. They are any phase structure grammar including all formal grammars. They generate the languages that are recognized by a Turing machine. The productions can be in the form of $\alpha \rightarrow \beta$ where α is a string of terminals and non terminals with at least one non terminal and α cannot be null. β is a string of terminals and non-terminals.

Example

$$S \rightarrow ACaB$$

$$Bc \rightarrow acB$$

$$CB \rightarrow DB$$

$$aD \rightarrow Db$$

- Language & their Relation Recursive & recursively enumerable sets, Language & Automata

- Language & their relation.

Let L_0, L_{cb}, L_{CFL} & L_{γ_1} denoted the language.

- property-1

From the definition it follows that

$L_{\gamma_1} \subseteq L_{CFL}$, $L_{CS_1} \subseteq L_0$

- property-2

$L_{CFL} \subseteq L_{CS_1}$

- context free grammer G with product^n of the form.

$A \to \epsilon$ is equivalent to context free grammar G1. which has no product^n of the form $s \to \epsilon$. (s doesn't appear on the right side of any product^n) so G is context sensitive. This proven language of context free language subset of context sensitive language.

- Property:

* Recursive & Recursive enumerable set

For defining recursive set we need the defini^n of a procedure & algorithm.

- procedure:

A procedure for solving a problem is a finate sequence of instruct^n which can be mechanically carried out any type of i/p.

- Algorithm:

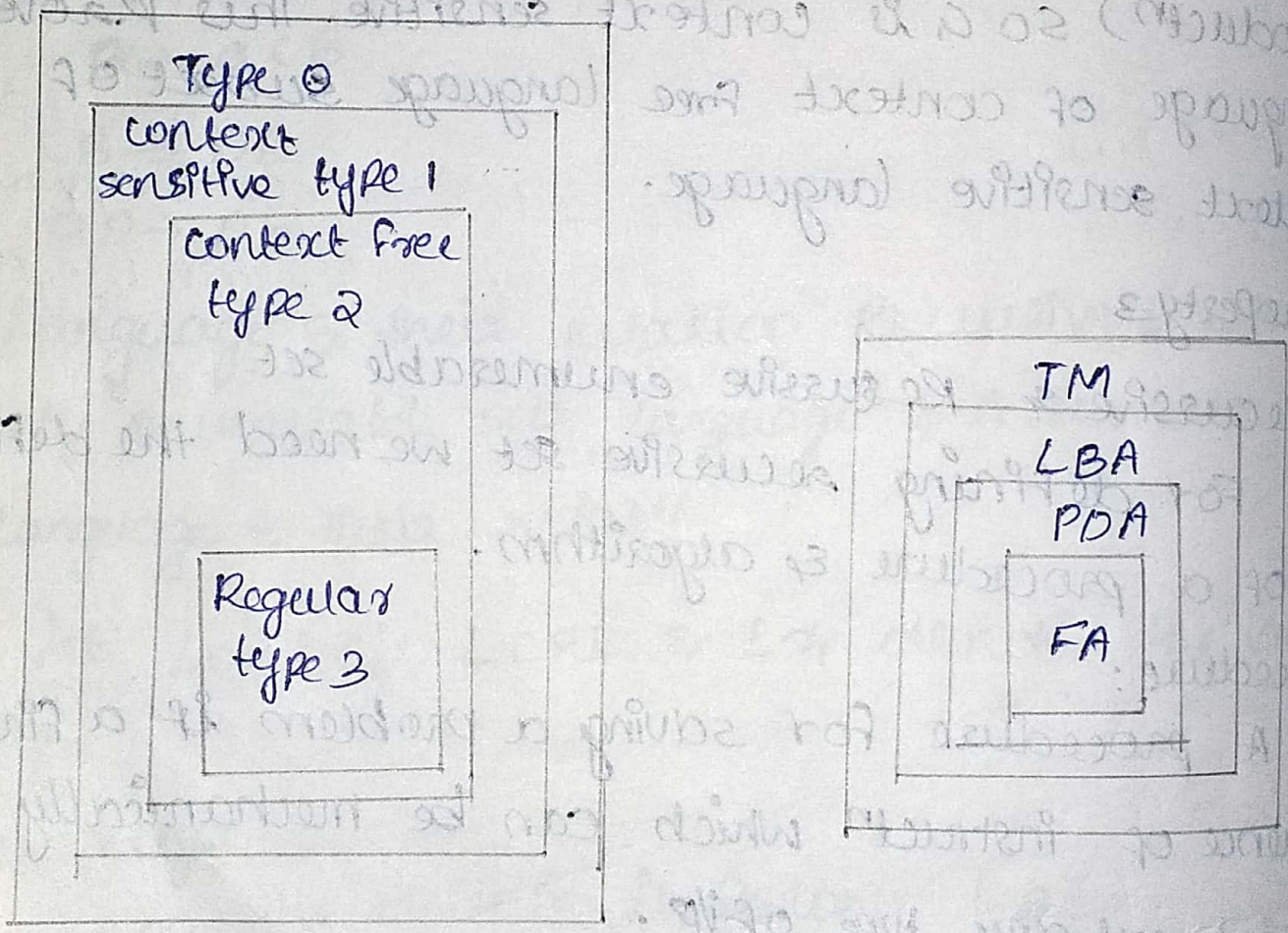
An Algorithm is a procedure that terminates After a finate no. of steps for any i/p.

A set x is recursive if we have an algorithm to determine whether a element belo

ngs to x or not.

A recursively enumerable set is a set x for which we have a procedure to determine whether a given element belongs to x or not.

—Language & Automata.



MOD-3 Theory of Automata.

Automata.

The term "Automata" which means "self-acting". An automaton (automata in plural) is an abstract self propelled computing device which follows a predetermined sequence of operatns automatically.

An automaton with finate no. of states is called a Finate Automata (FA) or Finate state machine (FSM).

Formal definitn of a Finate automatn. An automaton can be represented by a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where.

* Q is a finite set of states.
* Σ is a finite set of symbols, called the alphabet of the automatn.
* δ is the transitn fun^n.
* q_0 is the initial state from where any input is processed ($q_0 \in Q$).
* F is a set of finate state/states of Q ($F \subseteq Q$)

Alphabet

- **Definition** – An **alphabet** is any finite set of symbols.
- **Example** – ∑ = {a, b, c, d} is an **alphabet set** where 'a', 'b', 'c', and 'd' are **symbols**.

String

- **Definition** – A **string** is a finite sequence of symbols taken from ∑.
- **Example** – 'cabcad' is a valid string on the alphabet set ∑ = {a, b, c, d}

Length of a String

- **Definition** – It is the number of symbols present in a string. (Denoted by **|S|**).
- **Examples** –
  - If S = 'cabcad', |S|= 6
  - If |S|= 0, it is called an **empty string** (Denoted by **λ** or **ε**)

Kleene Star

- **Definition** – The Kleene star, **∑***, is a unary operator on a set of symbols or strings, **∑**, that gives the infinite set of all possible strings of all possible lengths over **∑** including **λ**

2

- **Representation** – $\sum^* = \sum_0 \cup \sum_1 \cup \sum_2 \cup \ldots\ldots$ where $\sum_p$ is the set of all possible strings of length p.
- **Example** – If $\sum = \{a, b\}$, $\sum^* = \{\lambda, a, b, aa, ab, ba, bb, \ldots\ldots\ldots\}$

Kleene Closure / Plus

- **Definition** – The set $\boldsymbol{\sum^+}$ is the infinite set of all possible strings of all possible lengths over $\sum$ excluding λ.
- **Representation** – $\sum^+ = \sum_1 \cup \sum_2 \cup \sum_3 \cup \ldots\ldots$

$$\sum^+ = \sum^* - \{ \lambda \}$$

- **Example** – If $\sum = \{ a, b \}$, $\sum^+ = \{ a, b, aa, ab, ba, bb, \ldots\ldots\ldots\}$

Language

- **Definition** – A language is a subset of $\sum^*$ for some alphabet $\sum$. It can be finite or infinite.
- **Example** – If the language takes all possible strings of length 2 over $\sum = \{a, b\}$, then $L = \{ ab, aa, ba, bb \}$

Finite Automaton can be classified into two types –

3

- Deterministic Finite Automaton (DFA)
- Non-deterministic Finite Automaton (NDFA / NFA)

Deterministic Finite Automaton (DFA)

In DFA, for each input symbol, one can determine the state to which the machine will move. Hence, it is called **Deterministic Automaton**. As it has a finite number of states, the machine is called **Deterministic Finite Machine** or **Deterministic Finite Automaton.**

Formal Definition of a DFA

A DFA can be represented by a 5-tuple (Q, ∑, δ, q_0, F) where –

- **Q** is a finite set of states.
- **∑** is a finite set of symbols called the alphabet.
- **δ** is the transition function where δ: Q × ∑ → Q
- **q_0** is the initial state from where any input is processed ($q_0 \in Q$).
- **F** is a set of final state/states of Q ($F \subseteq Q$).

Graphical Representation of a DFA

A DFA is represented by digraphs called **state diagram**.

- The vertices represent the states.
- The arcs labeled with an input alphabet show the transitions.
- The initial state is denoted by an empty single incoming arc.
- The final state is indicated by double circles.

Example

Let a deterministic finite automaton be →

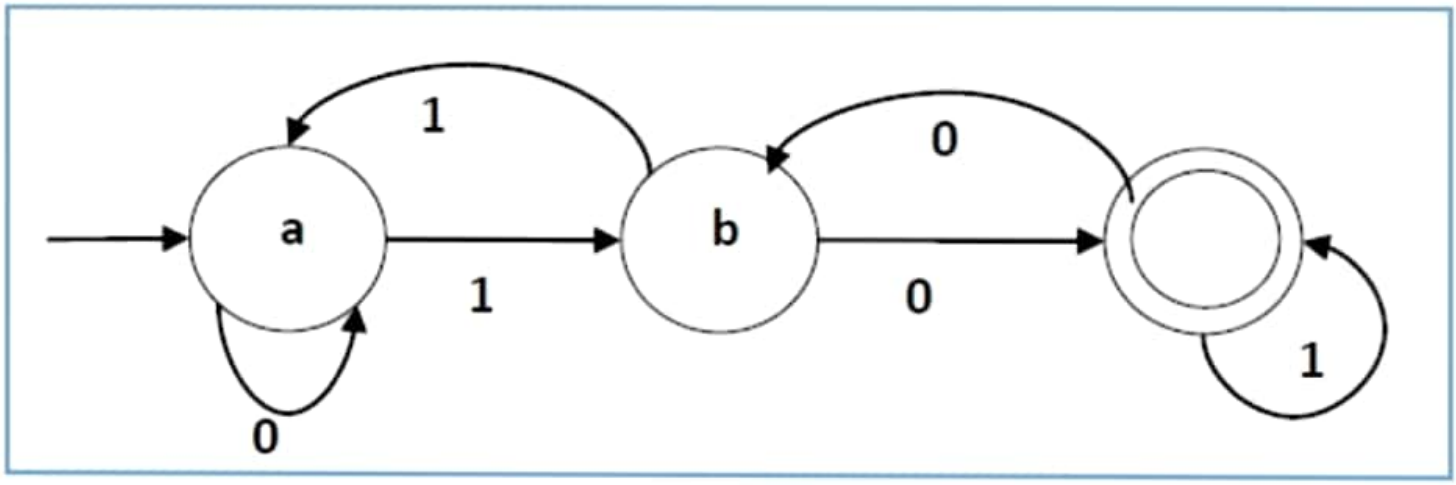
- Q = {a, b, c},
- ∑ = {0, 1},
- q_0 = {a},
- F = {c}, and

Transition function δ as shown by the following table −

Present State	Next State for Input 0	Next State for Input 1
a	a	b
b	c	a
c	b	c

Its graphical representation would be as follows –

5



In NDFA, for a particular input symbol, the machine can move to any combination of the states in the machine. In other words, the exact state to which the machine moves cannot be determined. Hence, it is called **Non-deterministic Automaton**. As it has finite number of states, the machine is called **Non-deterministic Finite Machine** or **Non-deterministic Finite Automaton**.

Formal Definition of an NDFA

An NDFA can be represented by a 5-tuple $(Q, \sum, \delta, q_0, F)$ where –

- **Q** is a finite set of states.
- **∑** is a finite set of symbols called the alphabets.
- **δ** is the transition function where $\delta: Q \times \sum \rightarrow 2^Q$

  (Here the power set of Q (2^Q) has been taken because in case of NDFA, from a state, transition can occur to any combination of Q states)
- **q_0** is the initial state from where any input is processed ($q_0 \in Q$).
- **F** is a set of final state/states of Q ($F \subseteq Q$).

Graphical Representation of an NDFA: (same as DFA)

An NDFA is represented by digraphs called state diagram.

- The vertices represent the states.
- The arcs labeled with an input alphabet show the transitions.
- The initial state is denoted by an empty single incoming arc.
- The final state is indicated by double circles.

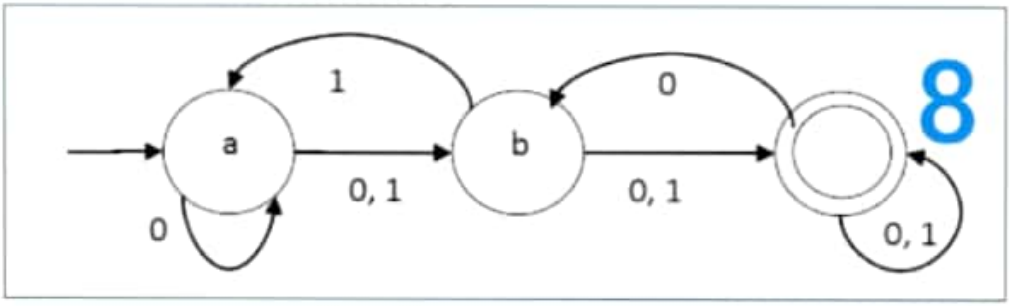
Example

Let a non-deterministic finite automaton be →

- Q = {a, b, c}
- ∑ = {0, 1}
- q_0 = {a}
- F = {c}

The transition function δ as shown below –

Present State	Next State for Input 0	Next State for Input 1
a	a, b	b
b	c	a, c
c	b, c	c



DFA vs NDFA

The following table lists the differences between DFA and NDFA.

DFA	NDFA
The transition from a state is to a single particular next state for each input symbol. Hence it is called *deterministic*.	The transition from a state can be to multiple next states for each input symbol. Hence it is called *non-deterministic*.
Empty string transitions are not seen in DFA.	NDFA permits empty string transitions.
Backtracking is allowed in DFA	In NDFA, backtracking is not always possible.
Requires more space.	Requires less space.
A string is accepted by a DFA, if it transits to a final state.	A string is accepted by a NDFA, if at least one of all possible transitions ends in a final state.

Acceptors, Classifiers, and Transducers

Acceptor (Recognizer)

An automaton that computes a Boolean function is called an **acceptor**. All the states of an acceptor is either accepting or rejecting the inputs given to it.

Classifier

A **classifier** has more than two final states and it gives a single output when it terminates.

Transducer

An automaton that produces outputs based on current input and/or previous state is called a **transducer**. Transducers can be of two types −

- **Mealy Machine** − The output depends both on the current state and the current input.
- **Moore Machine** − The output depends only on the current state.

Acceptability by DFA and NDFA

A string is accepted by a DFA/NDFA iff the DFA/NDFA starting at the initial state ends in an accepting state (any of the final states) after reading the string wholly.

A string S is accepted by a DFA/NDFA $(Q, \sum, \delta, q_0, F)$, iff

$$\delta^*(q_0, S) \in F$$

The language **L** accepted by DFA/NDFA is

$$\{S \mid S \in \sum^* \text{ and } \delta^*(q_0, S) \in F\}$$

A string S' is not accepted by a DFA/NDFA $(Q, \sum, \delta, q_0, F)$, iff

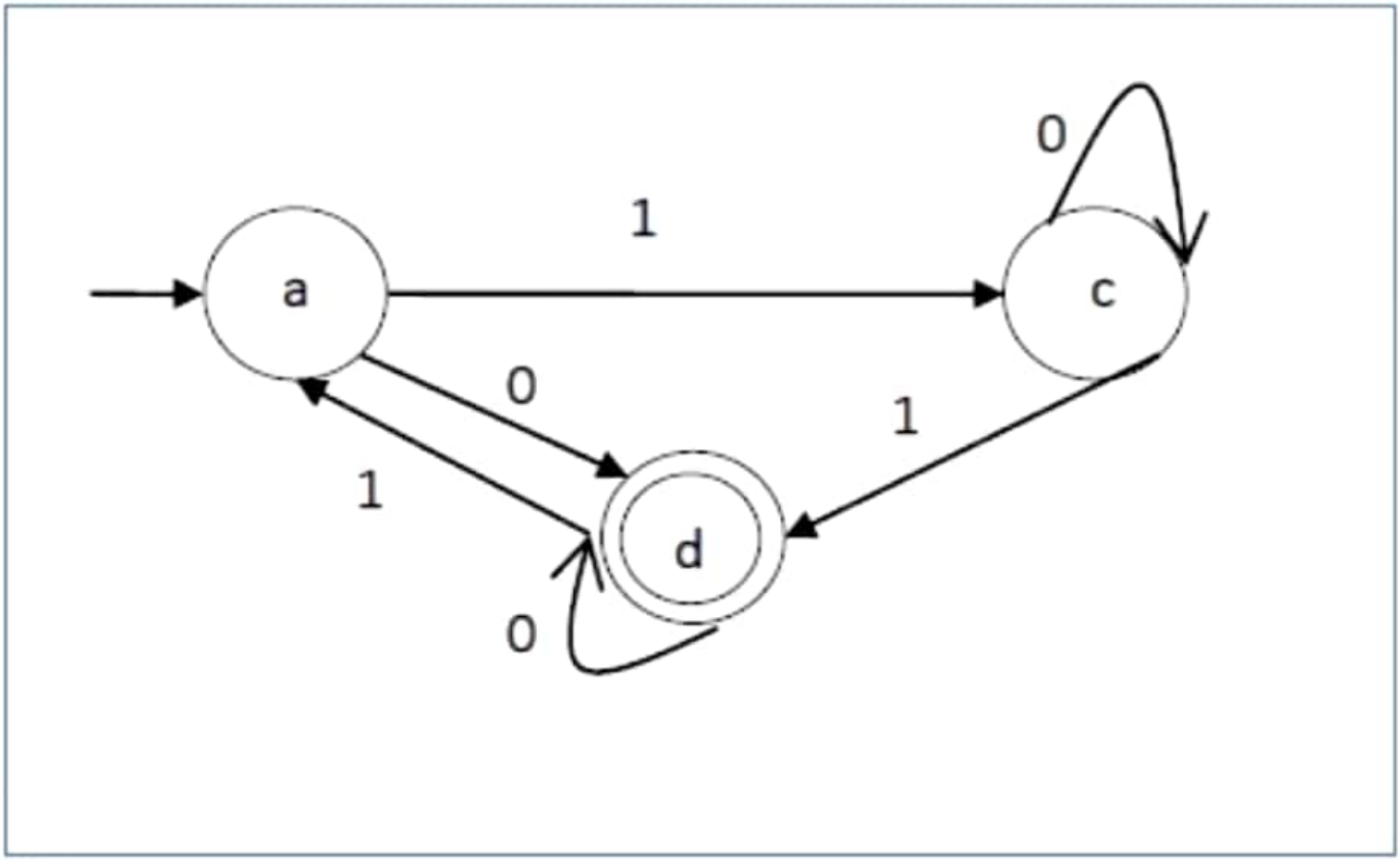
$$\delta^*(q_0, S') \notin F$$

The language L' not accepted by DFA/NDFA (Complement of accepted language L) is

$$\{S \mid S \in \sum^* \text{ and } \delta^*(q_0, S) \notin F\}$$

Example

Let us consider the DFA shown in Figure 1.3. From the DFA, the acceptable strings can be derived.



Strings accepted by the above DFA: {0, 00, 11, 010, 101, ...}

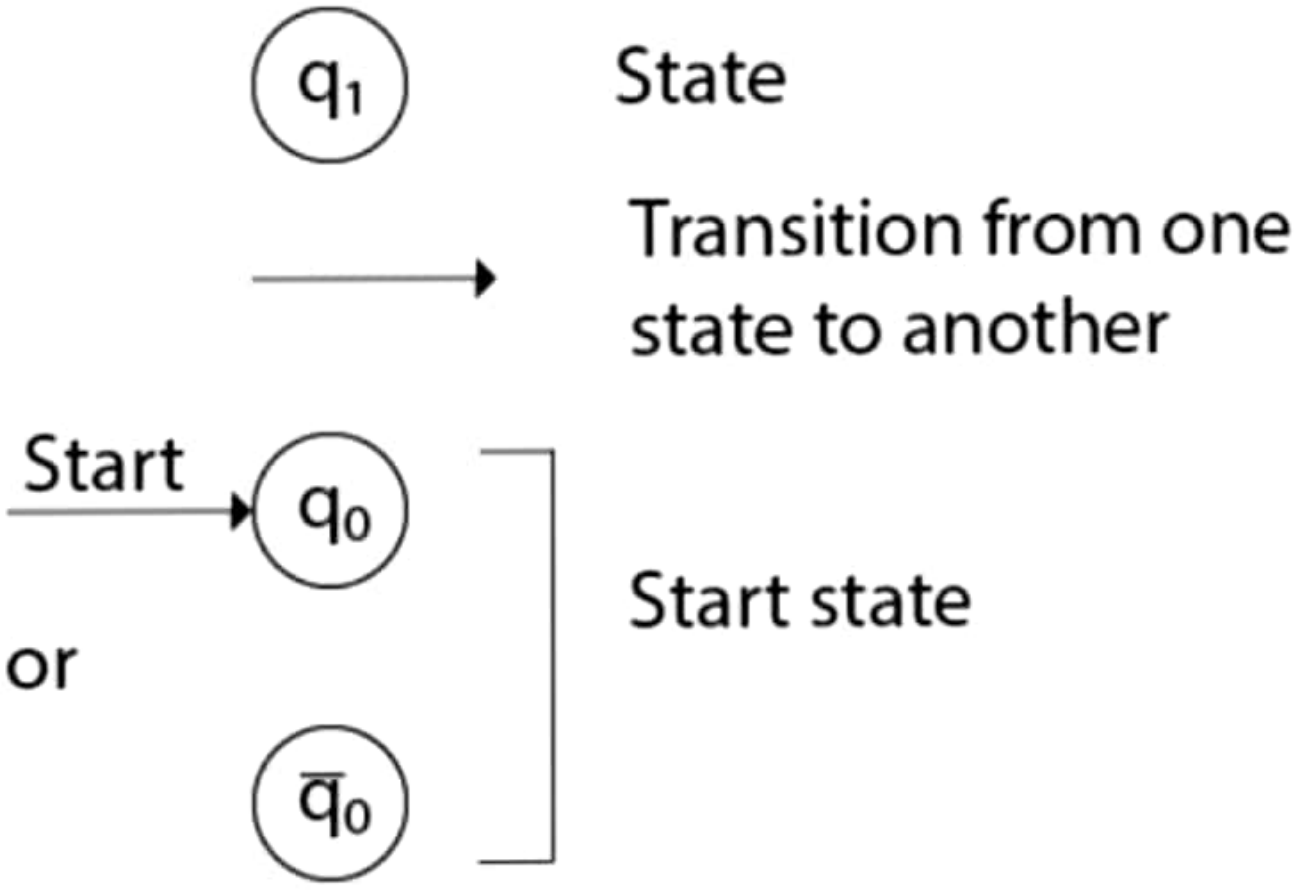
Strings not accepted by the above DFA: {1, 011, 111, ...}

Transition Diagram 12

A transition diagram or state transition diagram is a directed graph which can be constructed as follows:

- There is a node for each state in Q, which is represented by the circle.
- There is a directed edge from node q to node p labeled a if $\delta(q, a) = p$.
- In the start state, there is an arrow with no source.
- Accepting states or final states are indicating by a double circle.

Some Notations that are used in the transition diagram:



1. In DFA, the input to the automata can be any string. Now, put a pointer to the start state q and read the input string w from left to right and move the pointer according to the transition function, δ. We can read one symbol at a time. If the next symbol of string w is a and the pointer is on state p, move the pointer to δ(p, a). When the end of the input string w is encountered, then the pointer is on some state F.

2. The string w is said to be accepted by the DFA if $r \in F$ that means the input string w is processed successfully and the automata reached its final state. The string is said to be rejected by DFA if $r \notin F$.

DFA with ∑ = {0, 1} accepts all strings starting with 1.

Solution:

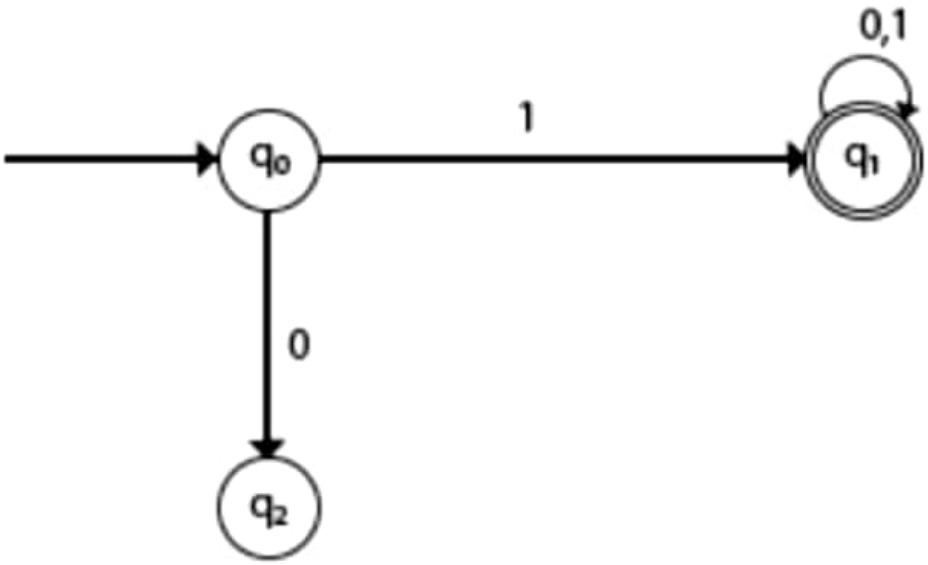


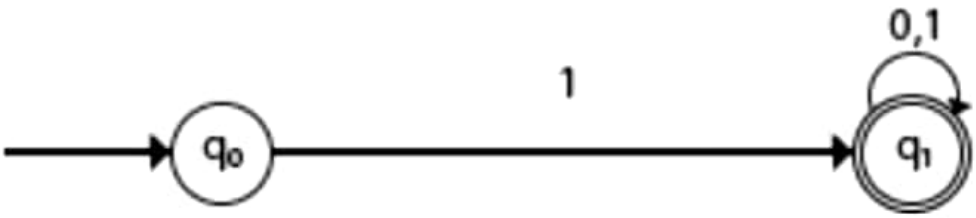
Fig: Transition diagram

The finite automata can be represented using a transition graph. In the above diagram, the machine initially is in start state q0 then on receiving input 1 the machine changes its state to q1. From q0 on receiving 0, the machine changes its state to q2, which is the dead state. From q1 on receiving input 0, 1 the machine changes its state to q1, which is the final state. The possible input strings that can be generated are 10, 11, 110, 101, 111....., that means all string starts with 1.

Example 2:

NFA with ∑ = {0, 1} accepts all strings starting with 1.

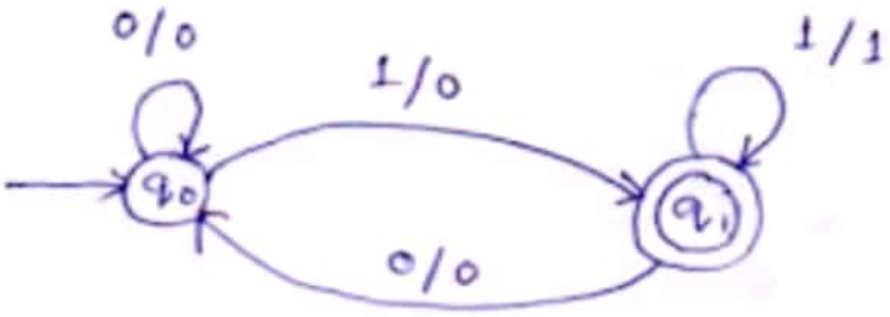
Solution:



The NFA can be represented using a transition graph. In the above diagram, the machine initially is in start state q0 then on receiving input 1 the machine changes its state to q1. From q1 on receiving input 0, 1 the machine changes its state to q1. The possible input string that can be generated is 10, 11, 110, 101, 111....., that means all string starts with 1.

Transition System & Transition Table

Transition System(Or Graph) :

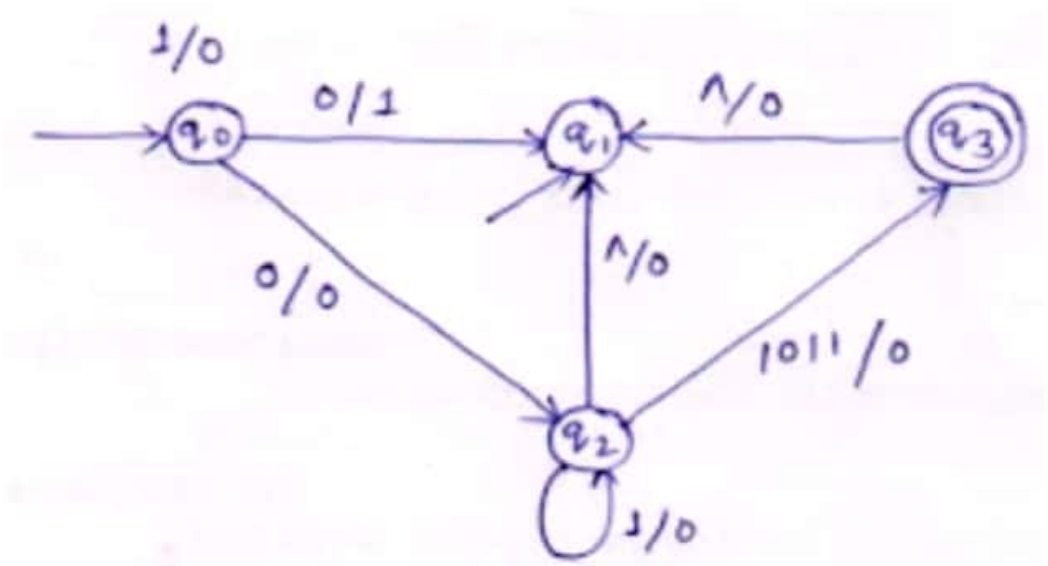


A Transition System

A transition system or transition graph is a finite directed labelled graph in which each vertex represents a state and the directed edges indicate of a state and the edges are labelled with input/output.

These diagrams shows a transition system and initial and final states.

Numerical : Consider the transition system given in figure below



Determine the initial states, final states and the acceptability of 101011,111010.

Solution: initial states are q0 and q1.

Final states are q3.

For string 101011, the path value is q0 q0 q2 q3. since q3 is the final state so this string is accepted by above transition system.

for string 111010, there is no path value. so this string is not accepted by above transition system.

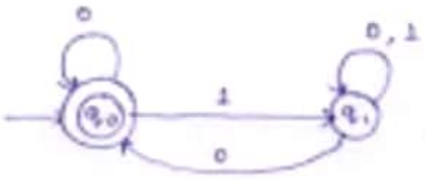
[**Trick : for checking the string acceptability, we start the string from initial state. if we reach the final state after completing the string, then we say that this string is accepted by transition system or not.**]

Transition Table :

State	0	1
$\rightarrow q_0$	q_0	q_1
q_1	q_1	q_0, q_1

Here q_0 is initial state as well as final state.

Transition diagram of the above transition table is



The above diagram shows that the transition table and the transition graph corresponding to the transition table.

Transition table is nothing but the tabular representation of the tarnsition graph or transition system.

We now give the (analytical) [illegible]

Definition 3.2 A transition system is a 5-tuple $(Q, \Sigma, \delta, Q_0, F)$, where

(i) Q, Σ and F are the finite nonempty set of states, the input alphabet, and the set of final states, respectively, as in the case of finite automata;

(ii) $Q_0 \subseteq Q$, and Q_0 is nonempty; and

(iii) δ is a finite subset of $Q \times \Sigma^* \times Q$.

In other words, if (q_1, w, q_2) is in δ, it means that the graph starts at the vertex q_1, goes along a set of edges, and reaches the vertex q_2. The concatenation of the label of all the edges thus encountered is w.

Definition 3.3 A transition system accepts a string w in Σ^* if

(i) there exists a path which originates from some initial state, goes along the arrows, and terminates at some final state; and

(ii) the path value obtained by concatenation of all edge-labels of the path is equal to w.

Example 3.2

Consider the transition system given in Fig. 3.6.

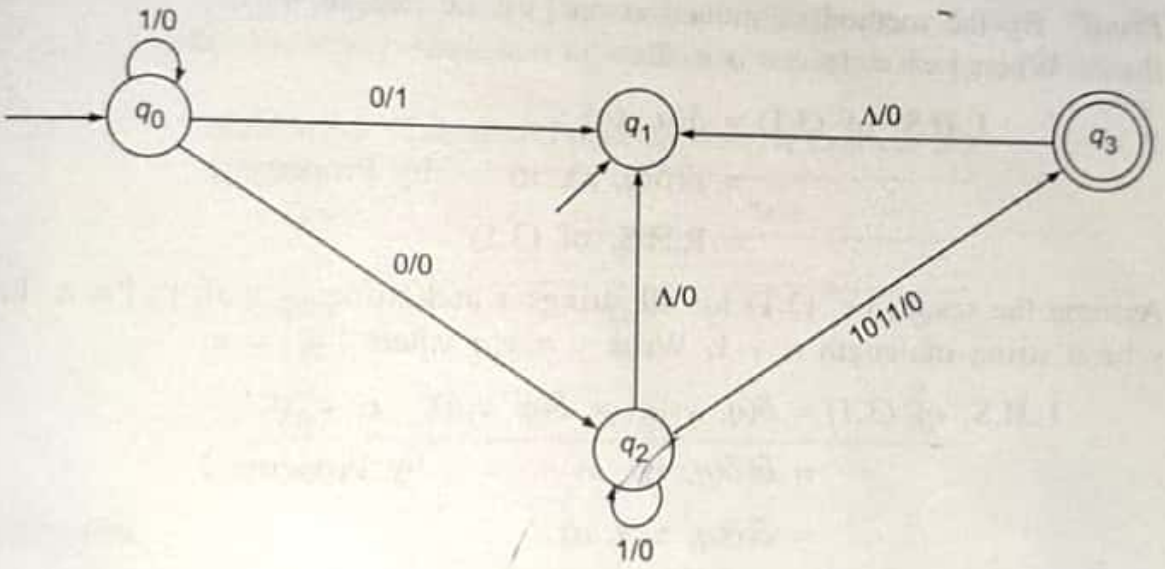


Fig. 3.6 Transition system for Example 3.2.

Determine the initial states, the final states, and the acceptability of 101011, 111010.

Solution

The initial states are q_0 and q_1. There is only one final state, namely q_3.

The path-value of $q_0q_0q_2q_3$ is 101011. As q_3 is the final state, 101011 is accepted by the transition system. But, 111010 is not accepted by the transition system as there is no path with path value 111010.

Note: Every finite automaton $(Q, \Sigma, \delta, q_0, F)$ can be viewed as a transition system $(Q, \Sigma, \delta' Q_0, F)$ if we take $Q_0 = \{q_0\}$ and $\delta' = \{(q, w, \delta(q, w)) | q \in Q, w \in \Sigma^*\}$. But a transition system need not be a finite automaton. For example, a transition system may contain more than one initial state.

3.4 PROPERTIES OF TRANSITION FUNCTIONS

Property 1 $\delta(q, \Lambda) = q$ is a finite automaton. This means that the state of the system can be changed only by an input symbol.

Property 2 For all strings w and input symbols a,

$$\delta(q, aw) = \delta(\delta(q, a), w)$$

$$\delta(q, wa) = \delta(\delta(q, w), a)$$

22

This property gives the state after the automaton consumes or reads the first symbol of a string aw and the state after the automaton consumes a prefix of the string wa.

EXAMPLE 3.3

Prove that for any transition function δ and for any two input strings x and y,

$$\delta(q, xy) = \delta(\delta(q, x), y) \tag{3.1}$$

Proof By the method of induction on $|y|$, i.e. length of y.

Basis: When $|y| = 1$, $y = a \in \Sigma$

$$\text{L.H.S. of (3.1)} = \delta(q, xa)$$

$$= \delta(\delta(q, x), a) \quad \text{by Property 2}$$

$$= \text{R.H.S. of (3.1)}$$

Assume the result, i.e. (3.1) for all strings x and strings y with $|y| = n$. Let y be a string of length $n + 1$. Write $y = y_1 a$ where $|y_1| = n$.

$$\text{L.H.S. of (3.1)} = \delta(q, xy_1a) = \delta(q, x_1a), \quad x_1 = xy_1$$

$$= \delta(\delta(q, x_1), a) \quad \text{by Property 2}$$

$$= \delta(\delta(q, xy_1), a)$$

$$= \delta(\delta(\delta(q, x), y_1), a) \quad \text{by induction hypothesis}$$

$$\text{R.H.S. of (3.1)} = \delta(\delta(q, x), y_1a)$$

$$= \delta(\delta(\delta(q, x), y_1), a) \quad \text{by Property 2}$$

Hence, L.H.S. = R.H.S. This proves (3.1) for any string y of length $n + 1$. By the principle of induction, (3.1) is true for all strings. ∎

NFA with ε-moves 23

Nondeterministic finite automaton with ε-moves (NFA-ε) is a further generalization to NFA. This automaton replaces the transition function with the one that allows the empty string ε as a possible input. The transitions without consuming an input symbol are called ε-transitions. In the state diagrams, they are usually labeled with the Greek letter ε. ε-transitions provide a convenient way of modeling the systems whose current states are not precisely known: i.e., if we are modeling a system and it is not clear whether the current state (after processing some input string) should be q or q', then we can add an ε-transition between these two states, thus putting the automaton in both states simultaneously.

Formal definition

An *NFA-ε* is represented formally by a 5-tuple, $(Q, \Sigma, \Delta, q_0, F)$, consisting of

- a finite set of states Q
- a finite set of input symbols called the alphabet Σ
- a transition function $\Delta : Q \times (\Sigma \cup \{\epsilon\}) \to P(Q)$
- an *initial* (or *start*) state $q_0 \in Q$

- an *initial* (or *start*) state $q_0 \in Q$

24

- a set of states F distinguished as *accepting* (or *final*) *states* $F \subseteq Q$.

Here, $P(Q)$ denotes the power set of Q and ε denotes empty string.

ε-closure of a state or set of states

For a state $q \in Q$, let $E(q)$ denote the set of states that are reachable from q by following ε-transitions in the transition function Δ, i.e., $p \in E(q)$ if there is a sequence of states $q_1, \ldots, q_k$ such that

- $q_1 = q$,
- $q_{i+1} \in \Delta(q_i, \varepsilon)$ for each $1 \leq i < k$, and
- $q_k = p$.

$E(q)$ is known as the **ε-closure** of q.

ε-closure is also defined for a set of states. The ε-closure of a set of states, P, of an NFA is defined as the set of states reachable from any state in P following ε-transitions. Formally, for $P \subseteq Q$, define $E(P) = \bigcup_{q \in P} E(q)$.

NFAs are said to be closed under a (binary/unary) operator if NFAs recognize the languages that are obtained by applying the operation on the NFA recognizable languages. The NFAs are closed under the following operations.

25

- Union (cf. picture)
- Intersection
- Concatenation
- Negation
- Kleene closure

Since NFAs are equivalent to nondeterministic finite automaton with ε-moves (NFA-ε), the above closures are proved using closure properties of NFA-ε. The above closure properties imply that NFAs only recognize regular languages.

NFAs can be constructed from any regular expression using Thompson's construction algorithm.

Steps for converting NFA with ε to DFA: 26

Step 1: We will take the ε-closure for the starting state of NFA as a starting state of DFA.

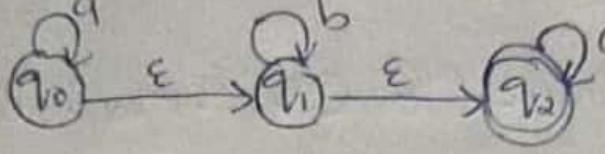
Step 2: Find the states for each input symbol that can be traversed from the present. That means the union of transition value and their closures for each state of NFA present in the current state of DFA.

Step 3: If we found a new state, take it as current state and repeat step 2.

Step 4: Repeat Step 2 and Step 3 until there is no new state present in the transition table of DFA.

Step 5: Mark the states of DFA as a final state which contains the final state of NFA.

→ Example for converting ε-NFA to DFA



Step I

→ Consider the starting state of NFA as the initial state of required DFA. Let it be state q_0 and we name it as 'A' in DFA

Now find out the ε-closure of q_0

ie,

A → starting state of DFA

$A = \varepsilon \text{ closure } (q_0)$

$= \{q_0, q_1, q_2\}$ —①

Step 2

→ Find out each transition on each input

ie, $\delta(A, a) = \varepsilon \text{ closure } (\delta(q_0, q_1, q_2), a)$

$= \varepsilon \text{ closure } (\delta(q_0, a) \cup \delta(q_1, a) \cup \delta(q_2, a))$

(28) $= \varepsilon$ closure $(q_0) = \{q_0, q_1, q_2\} = \boxed{A}$
- (from ①)

$\delta(A,b) = \varepsilon$ closure $(\delta(q_0, q_1, q_2), b)$

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

$= \varepsilon$ closure $(q_1) = \{q_1, q_2\} = \boxed{B}$

↓ (new state)

$\delta(A,c) = \varepsilon$ closure $(\delta(q_0, q_1, q_2), c)$

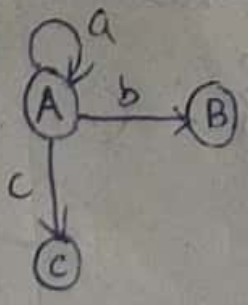
$= \varepsilon$ closure $(\delta(q_0,c) \cup \delta(q_1,c) \cup \delta(q_2,c)$

$= \varepsilon$ closure $(q_2) = \{q_2\} = \boxed{C}$

↓ new state

~~Steps~~

→ Now draw the DFA with above data



Step 3 & 4

→ Since we got new states B and C, so repeat step 2 for B and C

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

$\delta(B,a) = \varepsilon$ closure $(\delta(q_1,q_2)a)$

$= \varepsilon$ closure (ϕ) (since no value)

$= \boxed{D}$ → we introduce a dummy state for null value

$\delta(B,b) = \varepsilon$ closure $(\delta(q_1,q_2)\,b)$

$= \varepsilon$ closure $(\delta(q_1,b) \cup \delta(q_2,b))$

$= \varepsilon$ closure $\{q_1\} = B$

$\delta(B,c) = \varepsilon$ closure $(\delta(q_1,q_2),c)$

$= \varepsilon$ closure $\{q_2\} = C$

~~$\delta(C,a)$ & closure~~

Now,

$C = \{q_2\}$

$\delta(C,a) = \varepsilon$ closure $(\delta(q_2,a))$

$= \varepsilon$ closure $\phi = D$

$\delta(C,b) = \varepsilon$ closure $(\delta(q_2,b))$

$= \varepsilon$ closure $\phi = D$

$\delta(C,c) = \varepsilon$ closure $(\delta(q_2,c))$

$= \varepsilon$ closure $\{q_2\} = C$

Now,

$\delta(D,a) = \delta(D,b) = \delta(D,c) = D$

Step 5

Now we find out every transitions on all new states.

Finally, we mark final state of DFA as the final state of NFA.

→ Required DFA can be constructed using state table

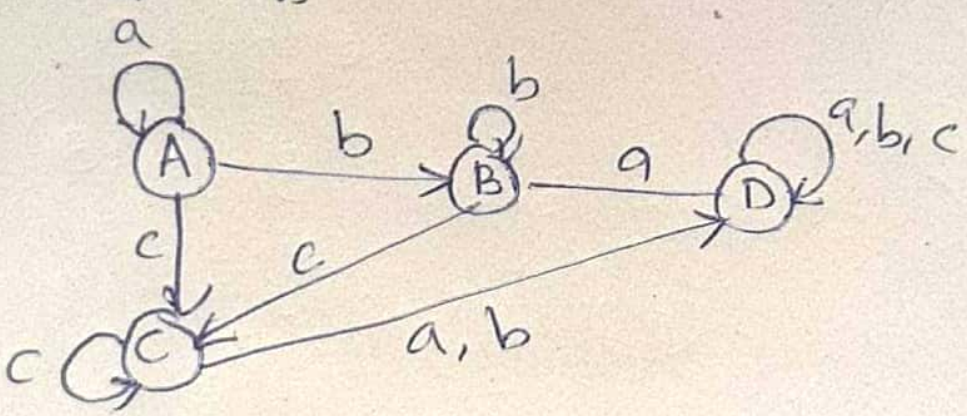
- State table of NFA

i/p / States	a	b	c	ε
q_0	q_0	–	–	q_1
q_1	–	q_1	–	q_2
q_2	–	–	q_2	–

- Required DFA state table is

i/p / States	a	b	c
A	A	B	C
B	D	B	C
C	D	D	C
D	D	D	D

DFA is



Conversion of Epsilon-NFA to NFA

Non-deterministic Finite Automata (NFA) is a finite automata having zero, one or more than one moves from a given state on a given input symbol. Epsilon NFA is the NFA which contains epsilon move(s)/Null move(s). To remove the epsilon move/Null move from epsilon-NFA and to convert it into NFA, we follow the steps mentioned below.

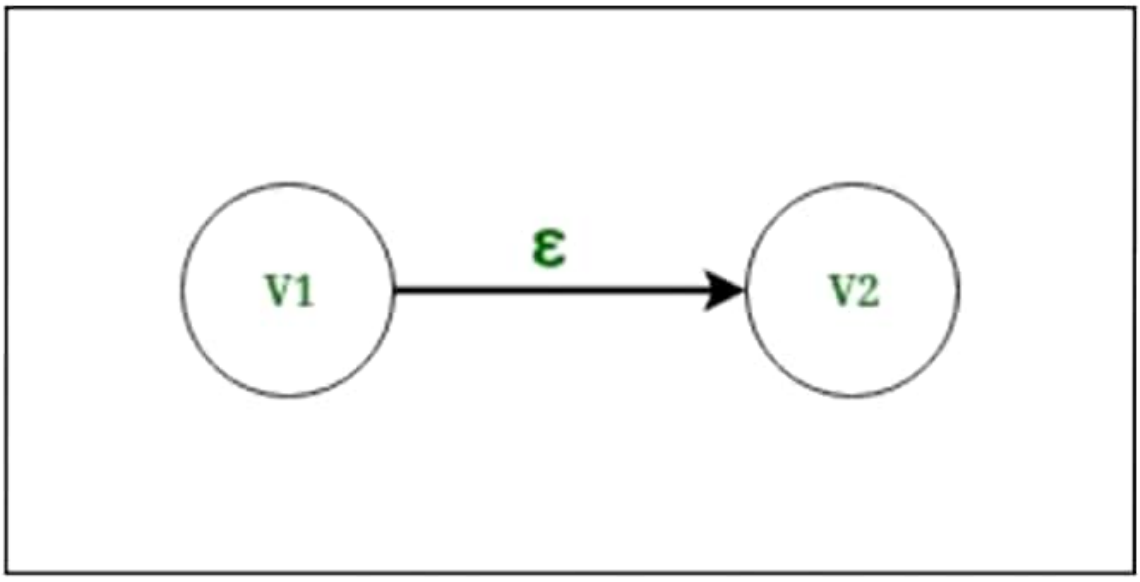
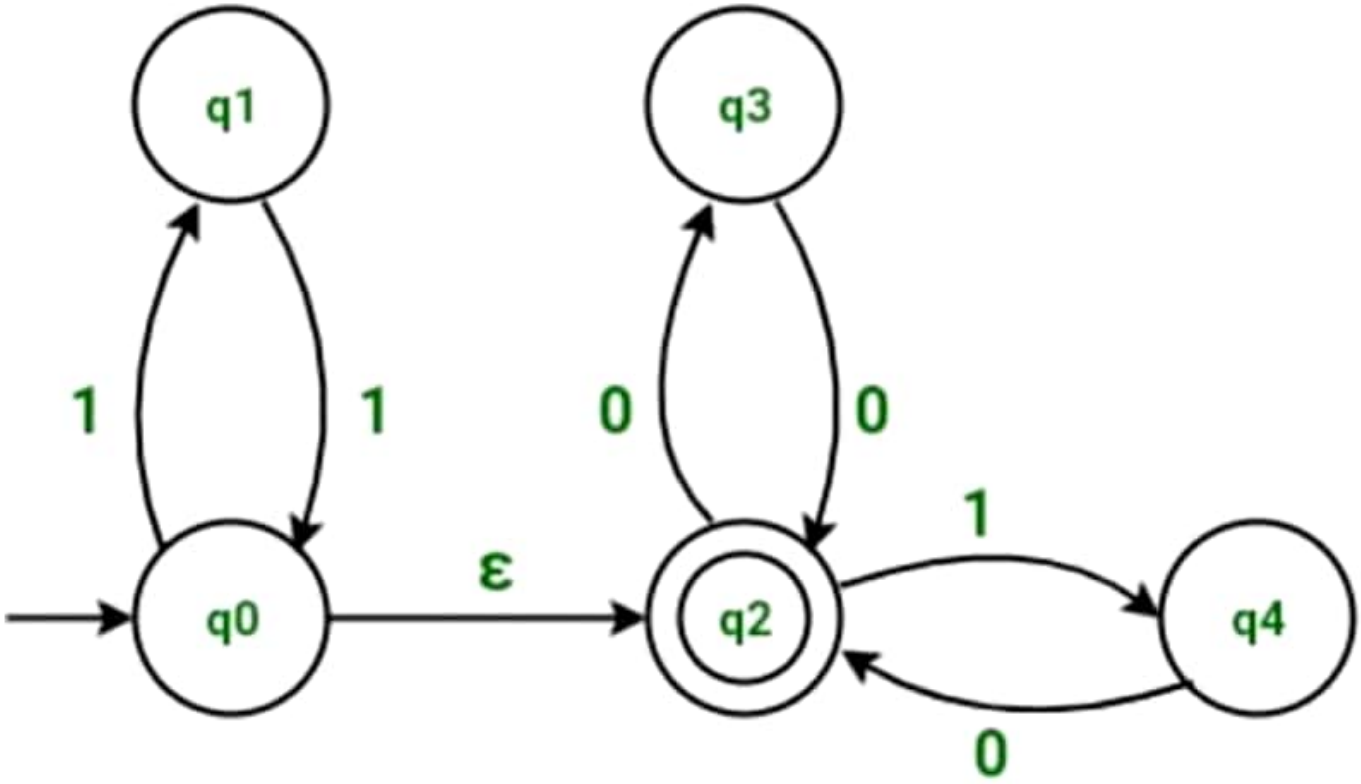


Figure – Vertex v1 and Vertex v2 having an epsilon move

Example: Convert epsilon-NFA to NFA. Consider the example having states q0, q1, q2, q3, and q4.



In the above example, we have 5 states named as q0, q1, q2, q3 and q4. Initially, we have q0 as start state and q2 as final state. We have q1, q3 and q4 as intermediate states.

Transition table for the above NFA is:

States/Input	Input 0	Input 1	Input epsilon
q0	–	q1	q2
q1	–	q0	–
q2	q3	q4	–
q3	q2	–	–
q4	q2	–	–

34

According to the transition table above,

- state q0 on getting input 1 goes to state q1.
- State q0 on getting input as a null move *(i.e. an epsilon move)* goes to state q2.
- State q1 on getting input 1 goes to state q0.
- Similarly, state q2 on getting input 0 goes to state q3, state q2 on getting input 1 goes to state q4.
- Similarly, state q3 on getting input 0 goes to state q2.
- Similarly, state q4 on getting input 0 goes to state q2.

We can see that we have an epsilon move from state q0 to state q2, which is to be removed.

Step-1:

Considering the epsilon move from state q0 to state q2. Consider the state q0 as vertex v1 and state q2 as vertex v2.

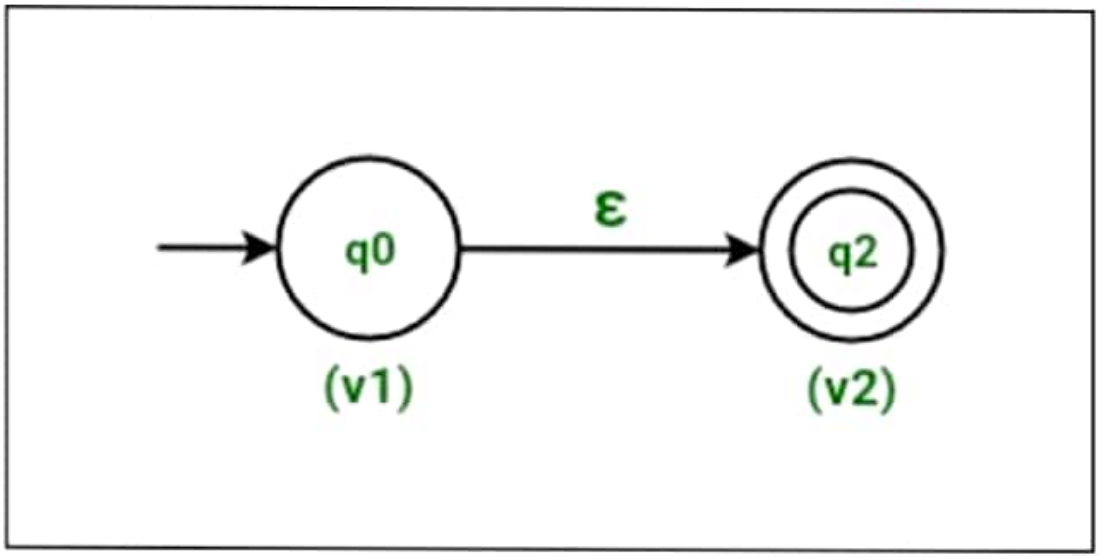


Figure – State q0 as vertex v1 and state q2 as vertex v2

Step-2:

Now find all the moves that starts from vertex v2 *(i.e. state q2)*.

After finding the moves, duplicate all the moves that start from vertex v2 *(i.e state q2)* with the same input to start from vertex v1 *(i.e. state q0)* and remove the epsilon move from vertex v1 *(i.e. state q0)* to vertex v2 *(i.e. state q2)*.

Since state q2 on getting input 0 goes to state q3.
Hence on duplicating the move, we will have state q0 on getting input 0 also to go to state q3.

Similarly state q2 on getting input 1 goes to state q4.
Hence on duplicating the move, we will have state q0 on getting input 1 also to go to state q4.

So, NFA after duplicating the moves is:

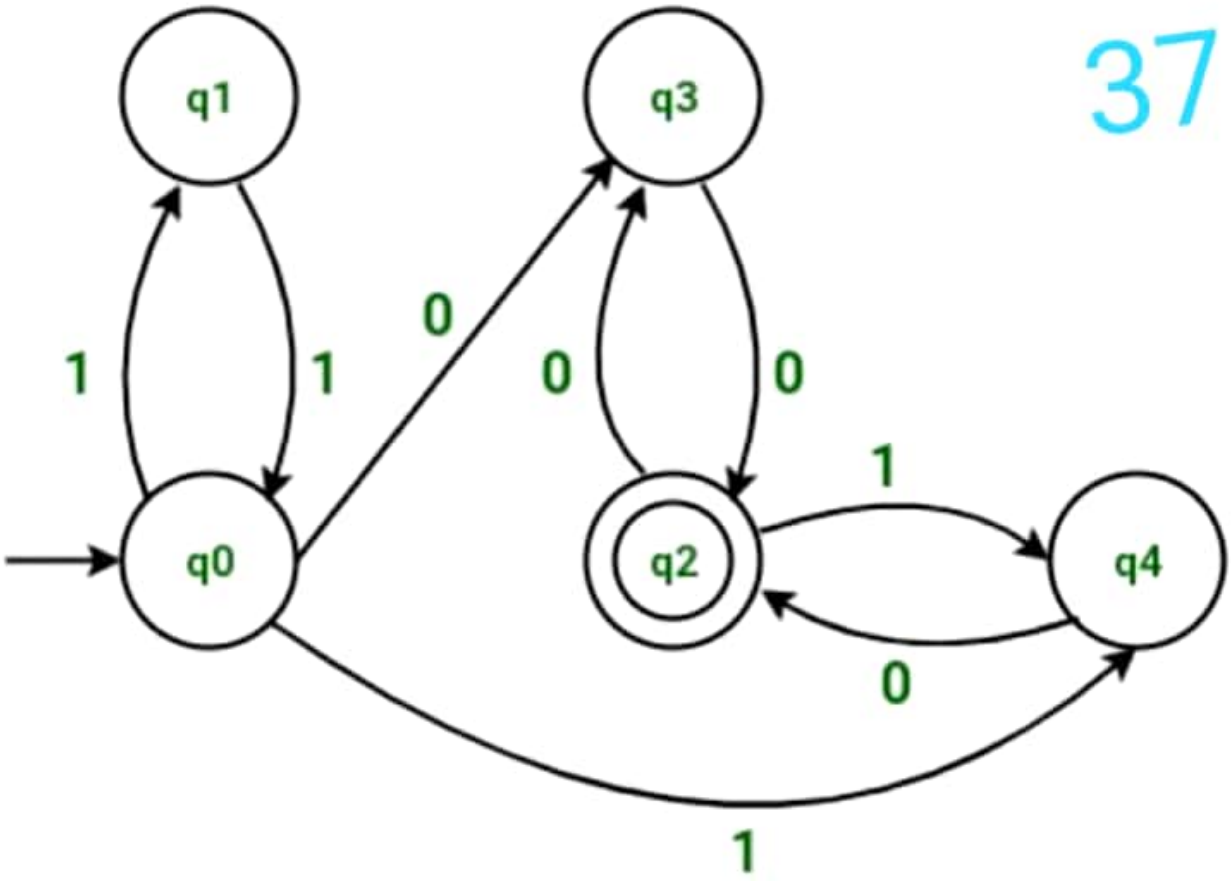


Figure – NFA on duplicating moves

Step-3:

Since vertex v1 *(i.e. state qo)* is a start state. Hence we will also make vertex v2 *(i.e. state q2)* as a start state.

Note that state q2 will also remain as a final state as we had initially.

NFA after making state q2 also as a start state is:

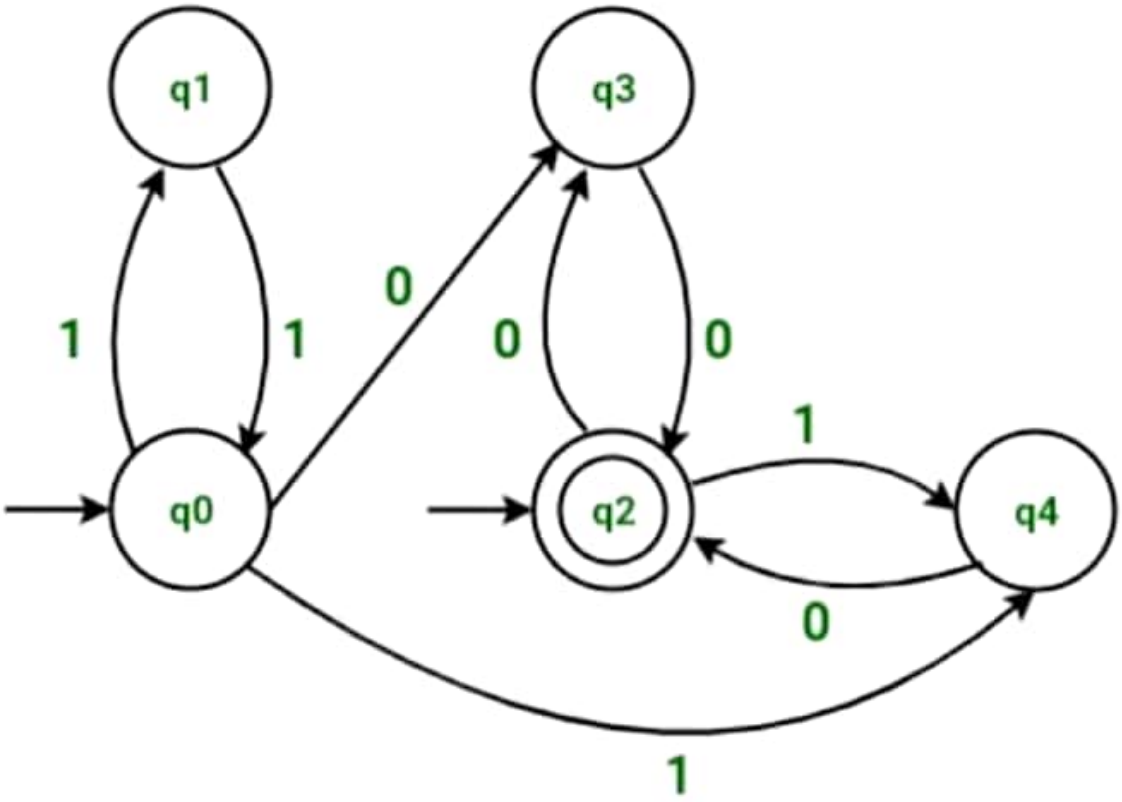


Figure – NFA after making state q2 as a start state

Step-4:

Since vertex v2 *(i.e. state q2)* is a final state. Hence we will also make vertex v1 *(i.e. state q0)* as a final state.

Note that state q0 will also remain as a start state as we had initially.

After making state q0 also as a final state, the resulting NFA is:

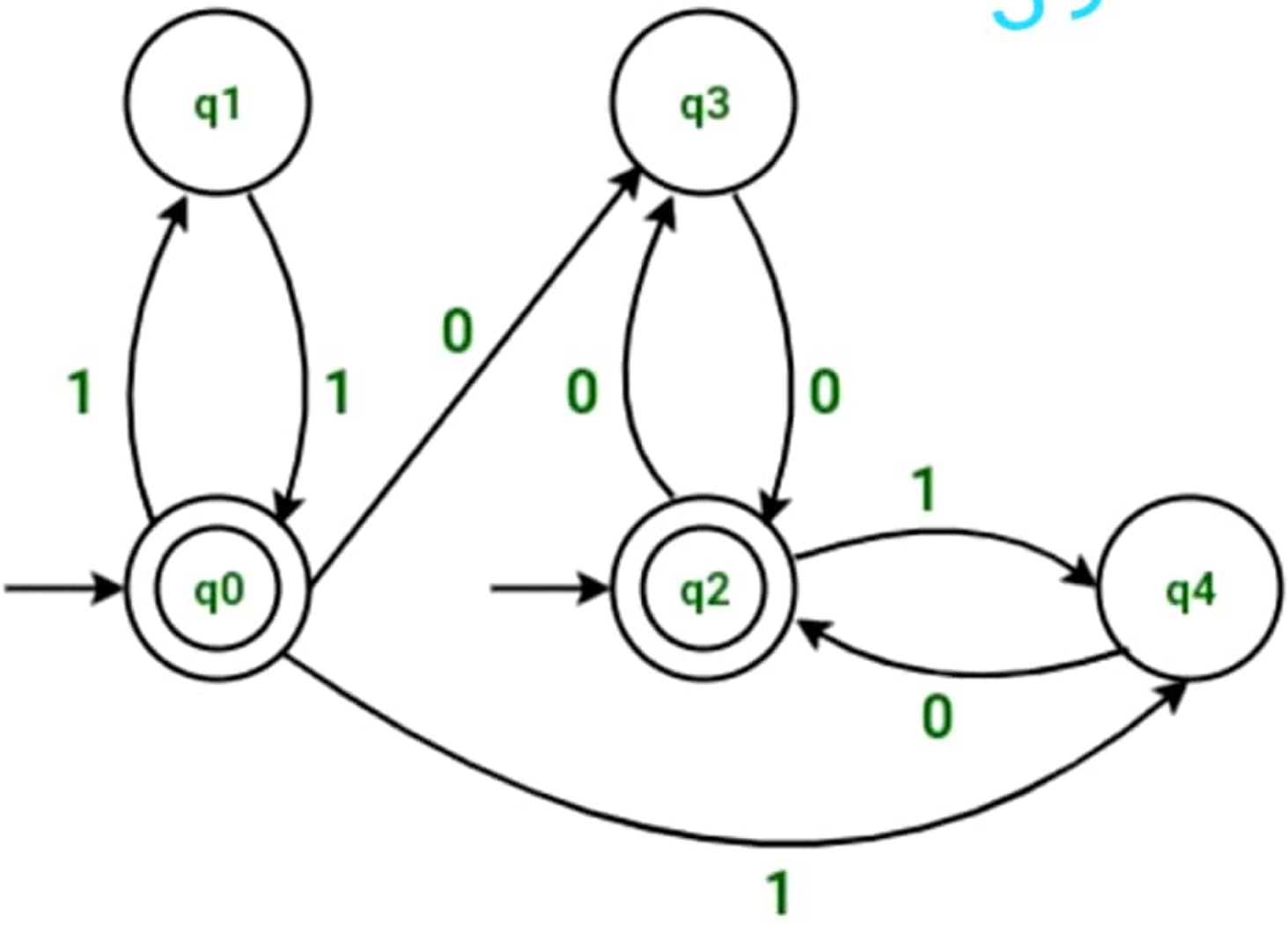


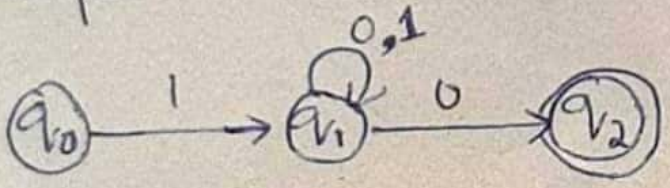
Figure – Resulting NFA (state q0 as a final state)

The transition table for the above resulting NFA is:

40

States/Input	Input 0	Input 1
q0	q3	q1,q4
q1	–	q0
q2	q3	q4
q3	q2	–
q4	q2	–

Example



– NFA

Step 1

State table for NFA

I/P / State	0	1
q_0	–	q_1
q_1	$\{q_1, q_2\}$	q_1
q_2	–	–

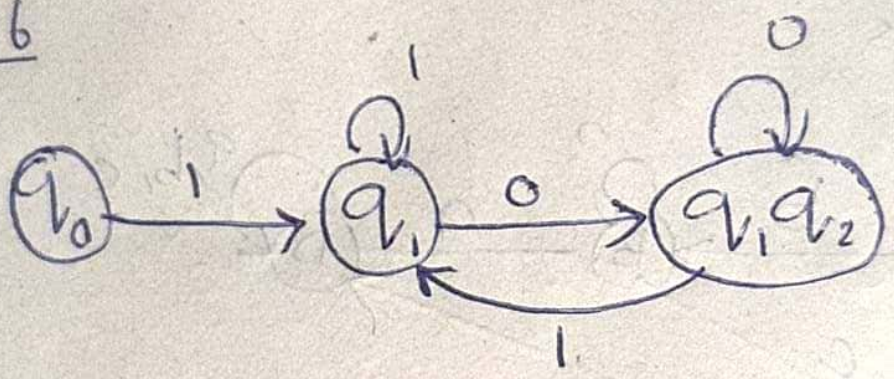
Step 2, 3, 4, 5

	0	1
q_0	–	q_1
q_1	$[q_1 q_2]$	q_1
$[q_1 q_2]$	$[q_1 q_2]$	q_1

→ Consider $\{q_1, q_2\}$ as a single state

→ Now take $[q_1 q_2]$ as next state

Step 6



Finite automata may have outputs corresponding to each transition. There are two types of finite state machines that generate output –

- Mealy Machine
- Moore machine

Mealy Machine

A Mealy Machine is an FSM whose output depends on the present state as well as the present input.

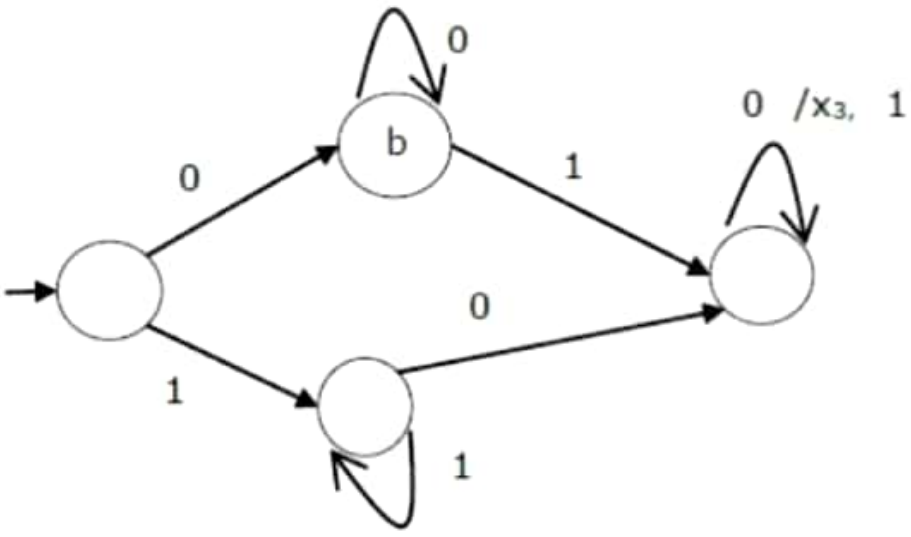
It can be described by a 6 tuple $(Q, \sum, O, \delta, X, q_0)$ where –

- **Q** is a finite set of states.
- **∑** is a finite set of symbols called the input alphabet.
- **O** is a finite set of symbols called the output alphabet.
- **δ** is the input transition function where $\delta: Q \times \sum \rightarrow Q$
- **X** is the output transition function where $X: Q \times \sum \rightarrow O$
- **q_0** is the initial state from where any input is processed ($q_0 \in Q$).

The state table of a Mealy Machine is shown below –

Present state	Next state			
	input = 0		input = 1	
	State	Output	State	Output
→ a	b	x_1	c	x_1
b	b	x_2	d	x_3
c	d	x_3	c	x_1
d	d	x_3	d	x_2

The state diagram of the above Mealy Machine is –



Moore Machine

Moore machine is an FSM whose outputs depend on only the present state.

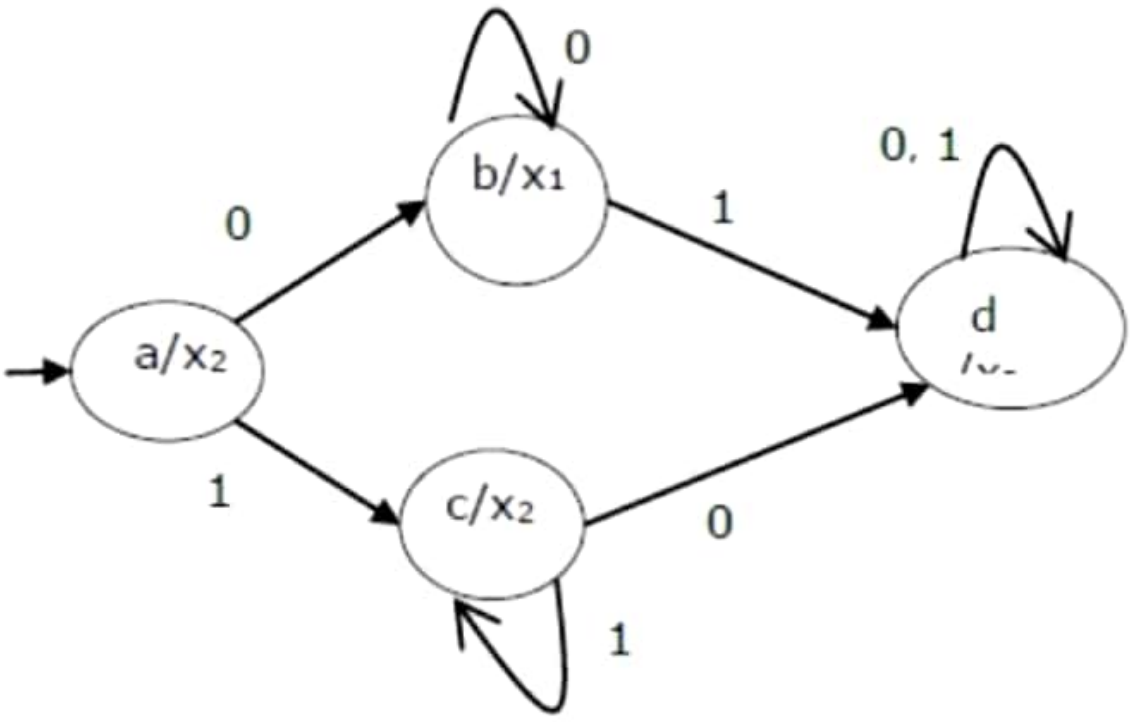
A Moore machine can be described by a 6 tuple $(Q, \sum, O, \delta, X, q_0)$ where –

- **Q** is a finite set of states.
- **∑** is a finite set of symbols called the input alphabet.
- **O** is a finite set of symbols called the output alphabet.
- **δ** is the input transition function where $\delta: Q \times \sum \rightarrow Q$
- **X** is the output transition function where $X: Q \rightarrow O$
- **q_0** is the initial state from where any input is processed ($q_0 \in Q$).

The state table of a Moore Machine is shown below –

Present state	Next State		Output
	Input = 0	Input = 1	
→ a	b	c	x_2
b	b	d	x_1
c	c	d	x_2
d	d	d	x_3

The state diagram of the above Moore Machine is –



Mealy Machine vs. Moore Machine

The following table highlights the points that differentiate a Mealy Machine from a Moore Machine.

Mealy Machine	Moore Machine
Output depends both upon the present state and the present input	Output depends only upon the present state.
Generally, it has fewer states than Moore Machine.	Generally, it has more states than Mealy Machine.
The value of the output function is a function of the transitions and the changes, when the input logic on the present state is done.	The value of the output function is a function of the current state and the changes at the clock edges, whenever state changes occur.
Mealy machines react faster to inputs. They generally react in the same clock cycle.	In Moore machines, more logic is required to decode the outputs resulting in more circuit delays. They generally react one clock cycle later.

Moore Machine to Mealy Machine

Algorithm 4

Input – Moore Machine

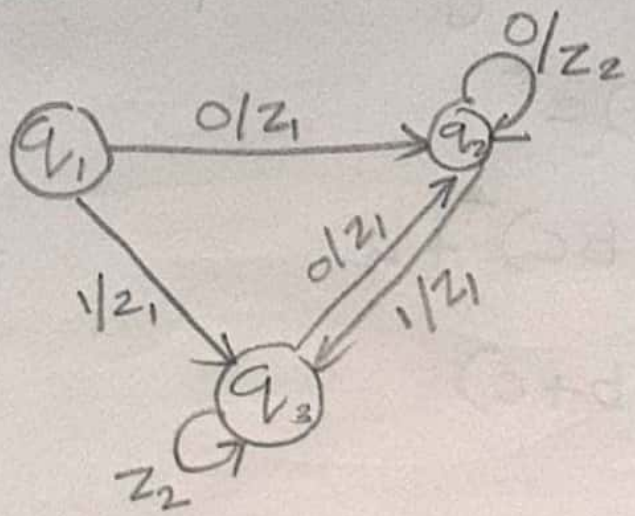
Output – Mealy Machine

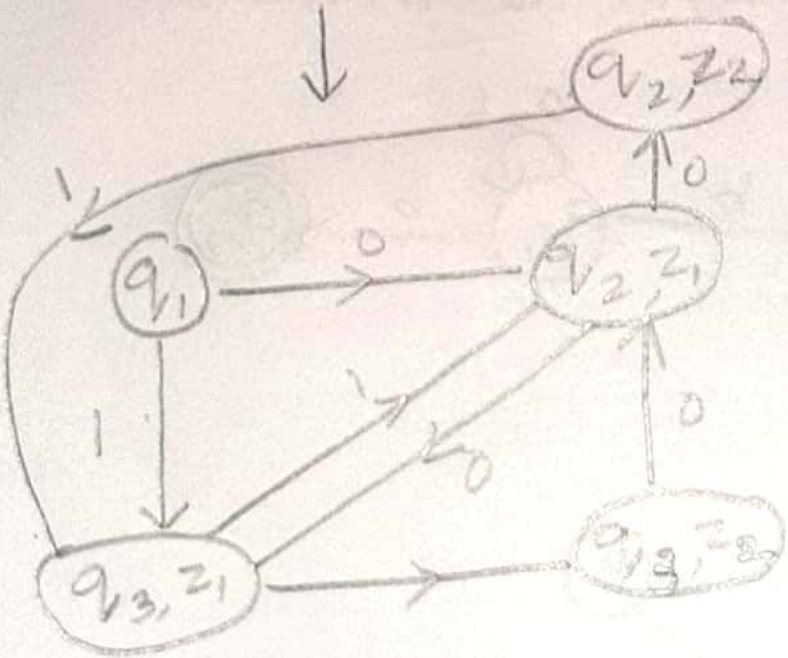
Step 1 – Take a blank Mealy Machine transition table format.

Step 2 – Copy all the Moore Machine transition states into this table format.

Step 3 – Check the present states and their corresponding outputs in the Moore Machine state table; if for a state Q_i output is m, copy it into the output columns of the Mealy Machine state table wherever Q_i appears in the next state.

* Convert Mealy machine to moore machine





Conversion from Mealy machine to Moore Machine

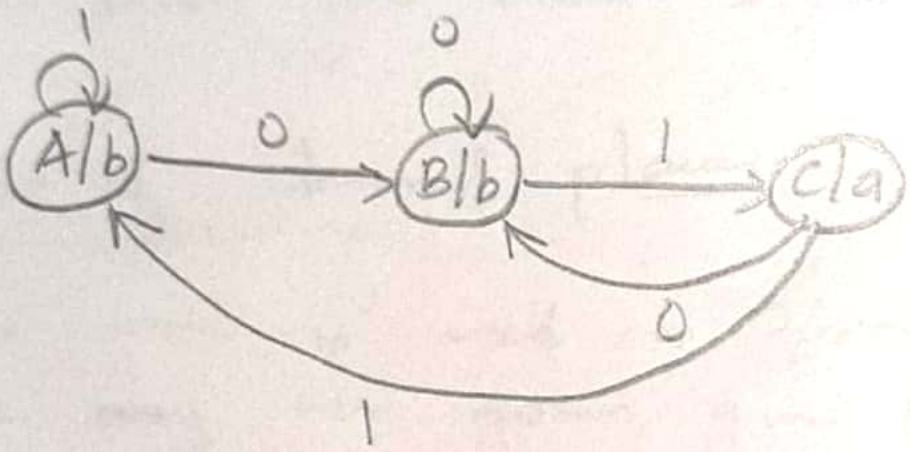
In Moore machine, the output is associated with every state, and in Mealy machine, the output is given along the edge with input symbol. To convert Moore machine to Mealy machine, state output symbols are distributed to input symbol paths. But while converting the Mealy machine to Moore machine, we will create a separate state for every new output symbol and according to incoming and outgoing edges are distributed.

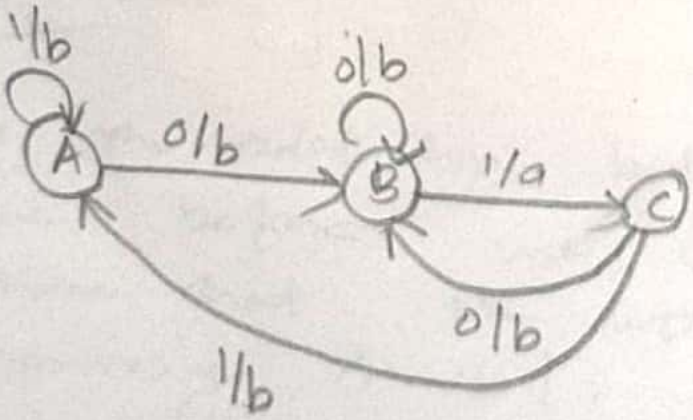
The following steps are used for converting Mealy machine to the Moore machine:

Step 1: For each state(Qi), calculate the number of different outputs that are available in the transition table of the Mealy machine.

Step 2: Copy state Qi, if all the outputs of Qi are the same. Break qi into n states as Qin, if it has n distinct outputs where n = 0, 1, 2....

→ Convert moore machine to mealy machine





MOD-4 CONTEXT FREE LANGUAGE

- Context Free Grammars

Context-Free grammars (CFGs) are used to describe context-Free languages. A context-Free grammar is a set of recursive rules used to generate patterns of strings. A context-Free grammar can describe all regular languages and more, but they can't describe all possible languages.

Context-free grammars are studied in field of theoretical computer science, compiler design, & linguistics. CFG's are used to describe programming languages & parser programs in compilers can be generated automatically from context-Free grammars.

Context-Free grammars can generate context-Free languages. They do this by taking a set of variables which are defined recursively, it terms of one another, by a set of production rules. context-free grammars are named as such bez any of the production in the grammar can be applied regardless of context- it doesn't depend on any other symbols that may or may

not be asocund a given symbol that is having a rule applied to it.

- context - free grammars have the following components:

* A set of terminal symbols which are the characters that appear in the language / strings generated by the grammar. Terminal symbols never appear on the left-hand side of the production rule & are always on the right-hand side.

* A set of non terminal symbols (or variables) which are placeholders for ~~non terminal~~ patterns of terminal symbol that can be generated by the non terminal symbols. These are the symbols that will always appear on left-hand side of the production rule though they can be included on the right-hand side. The strings that a CFG produces will contain only symbols from the set of non terminal symbols.

* A set of production rule which are the rules for replacing non terminal symbols. production rules have the following form: variable → string of variables & terminals.

* A start symbol which is a special non terminal symbol that appears in the initial string generated by the grammar.

- Definition: A context-free grammar (CFG) consisting of a finite set of grammar rules is a quadruple (NTPS) where.

* N is a set of non-terminal symbols.
* T is a set of terminals when $N \cap T = NULL$.
* P is a set of rules, $P: N \rightarrow (N \cup T)^*$, i.e, the left-hand side of the production rule p does have any right context or left context
* S is the start symbol.

eg: * The grammar $(\{A, \{a, b, c\}, P, A)$ P: A $\rightarrow aA, A \rightarrow abc$.

* The grammar $(\{S, a, b\}, \{a, b\}, P, S\}$, P: $S \rightarrow asa, S \rightarrow bsb, S \rightarrow \varepsilon$

* The grammar $(\{S, F\}, \{0, 1\}, P, S)$, P: $S \rightarrow 00S \mid 11F, F \rightarrow 00F \mid \varepsilon$.

- Generation of Derivation Tree

A derivation tree or parse tree is an ordered rooted tree that graphically represents the semantic information a string derived from a context-free grammar.

- Representation Technique.

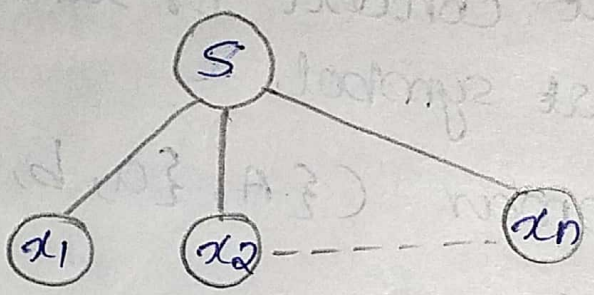
* Root vertex: Must be labeled by the start

symbol.

* Vertex: labeled by a non-terminal symbol.

* Leaves: Labeled by a terminal symbol or ε.

If $S \rightarrow x_1 x_2 \dots x_n$ is a product^n rule in CFG, then the parse tree / derivat^n tree will be as follows:



- There are 2 different approaches to draw a derivat^n tree:

- * Top down Approach:
  * starts with the starting symbol 'S'.
  * goes down to tree leaves using products.

- Bottom-up Approach:
  * starts from tree leaves.
  * proceeds upward to the root which is the starting symbol S.

- Derivat^n or Yield of a Tree.

The derivat^n or the yield of a parse tree is the final string obtained by concatenating the.

labels of the leaves of the tree from left to right, ignoring the nulls. However, if all the leaves are Null, derivation is Null.

eg: Let CFG $\{N, T, P, S\}$ be

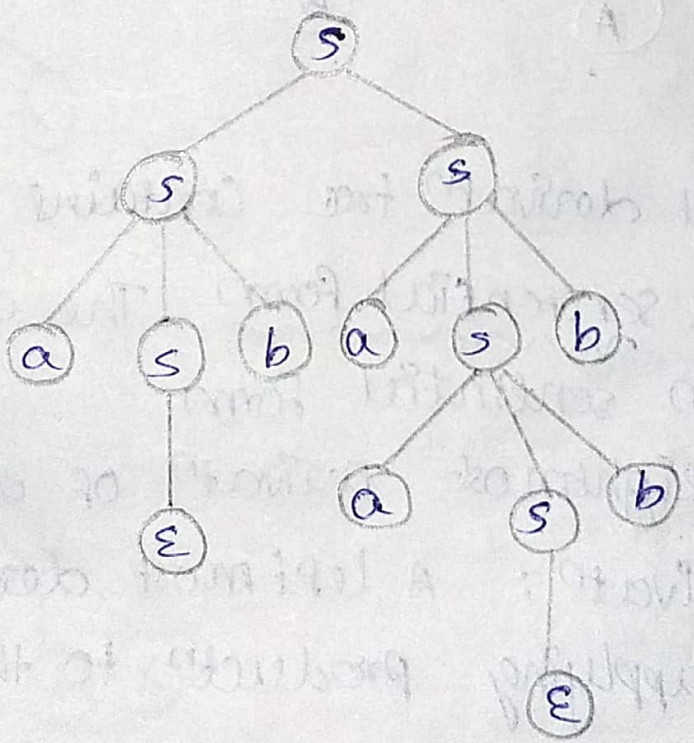
$N = \{S\}$, $T = \{a, b\}$, starting symbol = S; P

$= S \to SS \mid asb \mid \varepsilon$

one derivation from the above CFG is 'abaabb'

$S \to SS \to asbs \to abs \to abasb \to abaasbb \to$

abaabb.



Sentential Form & partial Derivation Tree

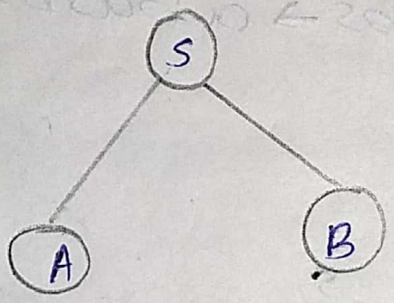
A partial derivation tree is a sub-tree

of a derivatn tree / parse tree such that either all of its children are in the sub-tree or none of them are in the sub-tree

eg: If in any CFG the productns are:

$S \to AB$, $A \to aaA \mid \varepsilon$, $B \to Bb \mid \varepsilon$

the partial derivatn tree can be the following:



If a partial derivatn tree contains the root S, its called a sentential form. The above sub-tree is also in sentential form.

Left most & Right most derivatn of a string.

* Left most derivatn: A left most derivatn is obtained by applying productn to the left most variable in each step.
* Right most derivatn: A right most derivatn is obtained by applying productn to the right most variable in each step.

eg: Let any set of producth rules in a CFG be

$$X \rightarrow X+X \mid X*X \mid X \mid a$$

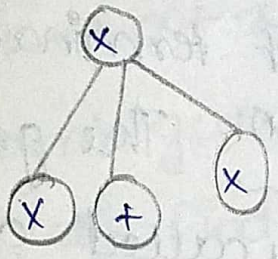
over an alphabet $\{a\}$.

The leftmost derivath for the string "a+a* a" may be:

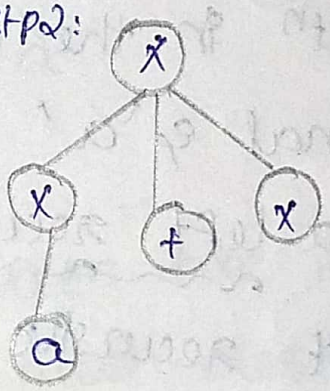
$$X \rightarrow X+X \rightarrow a+X \rightarrow a+X*X \rightarrow a+a*X \rightarrow a+a*a$$

The stepwise derivath of the above string is shown as below:

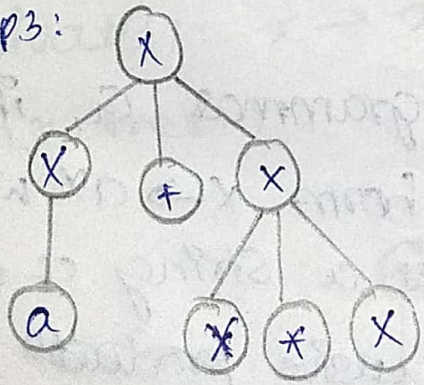
step1:



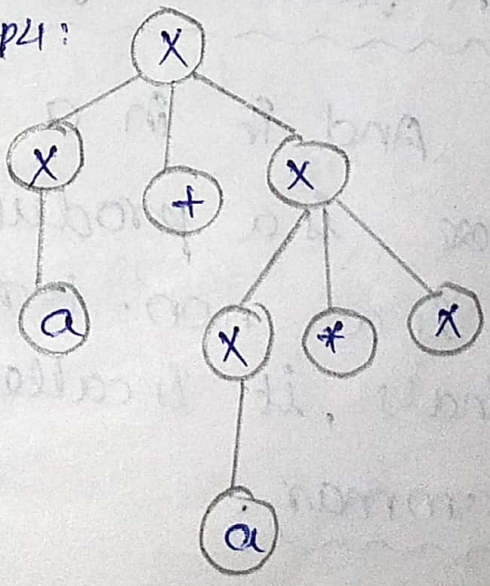
step2:



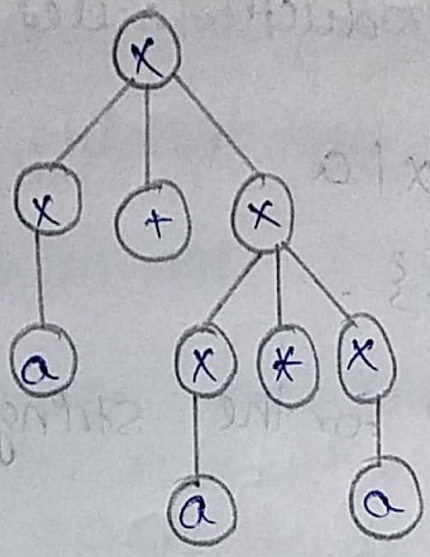
step3:



step4:



Step 5:



Left & Right Recursive Grammars

In a context-free grammar G, if there is a product^{n} in the form $X \rightarrow Xa$ where X is a non-terminal & 'a' is a string of terminals, its called a left recursive product^{n}. The gramma having a left recursive product^{n} is called a left recursive grammar.

And if in a context-free grammar G, if there is a product^{n} is in the form $X \rightarrow aX$ where X is a non-terminal & 'a' is a string of terminals, it is called a right recursive grammar.

- Ambiguity in context-free grammar

If a CFG 'G' has more than one derivation tree for some string $W \in L(G)$, it called an ambiguous grammar. There exist multiple right-most or left-most derivations for some string generated from that grammar.

- problem

1) Check whether the grammar G with produ ctn rules:

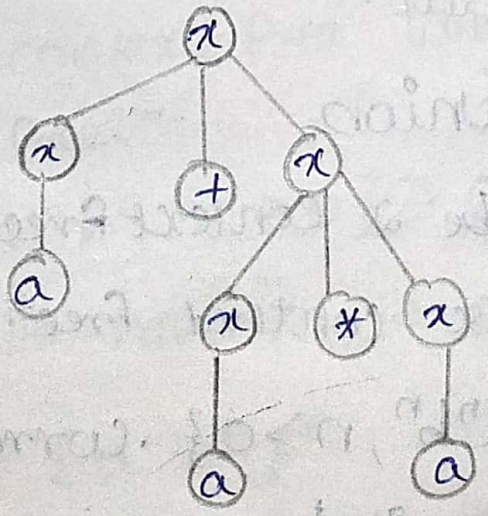
$x \rightarrow x+x \mid x*x \mid x \mid a$

is ambiguous or not

An: Let's find out the derivation tree for the string "a+a*a". It has 2 left most derivations.

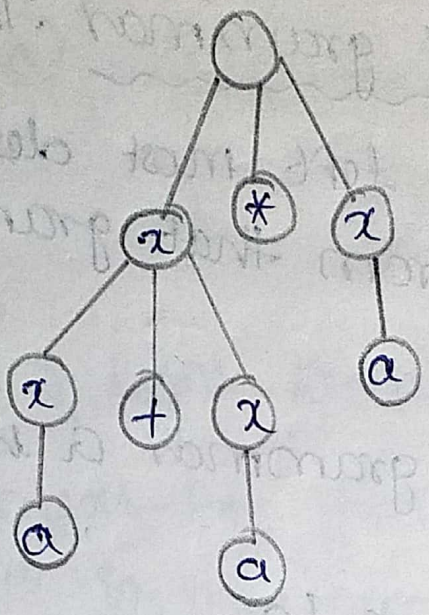
Derivation 1: $x \rightarrow x+x \rightarrow a+x \rightarrow a+x*x$

Parse tree 1:



Derivatn 2: $x \to x * x \to x + x * x \to a + x * x \to a + a * x \to a + a * a$

Parse tree 2:



Since there are 2 parse tree for a single string "a+a*a", the grammar G is ambiguous.

- context-Free grammar languages are closed under:
  * Union
  * concatenation
  * Kleene star operatn
- Union

Let L_1 & L_2 be 2 context Free languages. Then $L_1 \cup L_2$ is also context free.

eg: Let $L_1 = \{a^n b^n, n > 0\}$. Corresponding grammar G_1 will have $p: S_1 \to aAb$ lab

Let $L_2 = \{c^m d^m, m \geq 0\}$. Corresponding grammar G_2 will have $P: S_2 \rightarrow cBb \mid \varepsilon$

Union of L_1 & L_2 $L = L_1 \cup L_2 = \{a^n b^n\} \cup \{c^m d^m\}$

The corresponding grammar G will have the additional production $S \rightarrow S_1 \mid S_2$

- concatenation

If L_1 & L_2 are context free languages, then $L_1 L_2$ is also context free.

eg: Union of the languages L_1 & L_2, $L = L_1 L_2 = \{a^n b^n c^m d^m\}$

The corresponding grammar G will have the additional production $S \rightarrow S_1 S_2$

- Kleene star.

If L is a context free language, the L^* is also context free.

eg: Let $L = \{a^n b^n, n \geq 0\}$. Corresponding grammar G will have $P: S \rightarrow aAb \mid \varepsilon$

Kleene star $L_1 = \{a^n b^n\}^*$

The corresponding grammar G_1 will have

additional productns $S_1 \rightarrow SS_1 \mid \varepsilon$

- Context-free languages are not closed under:

* Intersectn: If L_1 & L_2 are context free languages, then $L_1 \cap L_2$ is not necessarily context free.
* Intersectn with Regular Language: If L_1 is a regular language & L_2 is a context free language, then $L_1 \cap L_2$ is a context free language.
* complement: If L_1 is a context free language, then L_1' may not be context free.

In a CFG, it may happen that all the productn rules & symbols aren't needed for the derivatn of strings. Besides, there may be some null productns & unit productns. Eliminatn of these productns & symbols is called simplification of CFGs. simplificatn essentially comprises of the following steps:

* Reductn of CFG
* Removal of unit productns.

* Removal of Null productns:

- Reductn of CFG.

CFGs are reduced in 2 phases:

*phase 1: Derivatn of an equivalent grammar, G', from the CFG, G, such that each variable derives some terminal string.

- Derivatn procedure:

stp 1: Include all symbols, W_i, that derive some terminal & initialize $i=1$.

stp 2: include all symbols, W_{i+1}, that derive W_i.

stp 3: Increment i & repeat stp 2, until $W_{i+1} = W_i$.

stp 4: Include all productn rules that have W_i in it.

- phase 2: Derivatn of an equivalent grammar "G", from the CFG, G', such that each symbol appears in a sentential form.

- Derivation procedure:

stp 1: Include the start symbol in Y_1 & initialize $i=1$.

stp 2: Include all symbols Y_{i+1}, that can be derived from Y_i & include all productn rules that have been applied.

Step3: Increment i & repeat step 2, until

$Y_{i+1} = Y_i$

- problem.

Find a reduced grammar equivalent to the grammar G, having Productn rules, P: S → AC | B, A → a, C → C | BC, E → aA | e

An: T = {a, c, e}

W_1 = {A, C, E} from rules A → a, C → C & E → aA

W_2 = {A, C, E} ∪ {S} from rule S → AC

W_3 = {A, C, E, S} ∪ ∅

since $W_2 = W_3$, we can derive G' as – G' = {{A, C, E, S}, {a, c, e}, P{S}}

where P: S → AC, A → a, C → C, E → aA | e

Phase 2:

Y_1 = {S}

Y_2 = {S, A, C} from rule S → AC

Y_3 = {S, A, C, a, C} from rules A → a & C → C

Y_4 = {S, A, C, a, e}

Since $Y_3 = Y_4$, we can derive G'' as -

G'' { { A, C, S } , {a, c} , P, {S} }

where P : S → AC, A → a, C → c

- Removal of Unit productn

Any productn rule in the form A → B where A, B ∈ Non-terminal is called Unit productn.

* Removal procedure:

Step 1: To remove A → B, add productn A → x to the grammar rule whenever B → x occurs in the grammar. [x ∈ Terminal, x can be null]

Step 2: Delete A → B from the grammar.

Step 3: Repeat from step 1 until all unit productns are removed.

- problem:

Remove unit productn from the following

S → xy, x → a, Y → z | b, z → M, M → N, N → a

An: There are 3 unit productns in the grammar.

Y → z, z → M & M → N

At first, we will remove M → N

as N → a, we add M → a & M → N is removed.

The Production set becomes

$S \to XY, X \to a, Y \to Z \mid b, Z \to M, M \to a, N \to a$

Now we will remove $Z \to M$.

As $M \to a$, we add $Z \to a$, & $Z \to M$ is removed.

The Production set becomes

$S \to XY, X \to a, Y \to Z \mid b, Z \to a, M \to a, N \to a$

Now we will remove $Y \to Z$.

As $Z \to a$, we add $Y \to a$, & $Y \to Z$ is removed.

The Production set becomes

$S \to XY, X \to a, Y \to a \mid b, Z \to a, M \to a, N \to a$

Now Z, M & N are unreachable, hence we can remove those.

The final CFG is unit production free-

$S \to XY, X \to a, Y \to a \mid b$

Removal of Null Productions

In a CFG, a non-terminal symbol 'A' is a nullable variable if there is a production $A \to \varepsilon$ or there is a derivation that starts at A & finally ends with

ε: $A \to \ldots\ldots \to \varepsilon$

-Removal Productions

Step1: Find out nullable non-terminal variables which derive ε.

Step2: For each Production $A \rightarrow a$, construct all productions $A \rightarrow x$ where x is obtained from 'a' by removing one or multiple non-terminals from Step1.

Step3: combine the original Productions with the result of Step2 & remove ε-Productions.

-problem:

Remove null production from the following.

$S \rightarrow ASA \mid aB \mid b$, $A \rightarrow B$, $B \rightarrow b \mid \epsilon$

An: There are 2 nullable variables - A & B

At first, we will remove $B \rightarrow \epsilon$.

After removing $B \rightarrow \epsilon$, the production set becomes-

$S \rightarrow ASA \mid aB \mid b \mid a$, $A \rightarrow \epsilon \mid B \mid b$ & epsilon, $B \rightarrow b$

Now we will remove $A \rightarrow \epsilon$.

After removing $A \rightarrow \epsilon$, the production set becomes

$S \rightarrow ASA \mid aB \mid b \mid a \mid SA \mid AS \mid S$, $A \rightarrow B \mid b$, $B \rightarrow b$

This is the final production set without null transitn.

— chomsky Normal Formal

A CFG is in chomsky Normal Form if the Productns are in the following forms —

* $A \to a$
* $A \to BC$
* $S \to \varepsilon$

Where A, B & C are non terminals & a is terminal

— Algorithm to convert into chomsky Normal Form.

Stp 1: If the start symbol s occurs on some right side Create a new start symbol 'S' & a new productn $S' \to S$.

Stp 2: Remove Null productns. (using the Null productn removal algorithm discussed earlier)

Stp 3: Remove unit productns. (using the unit Productn removal algorithm discussed earlier)

Stp 4: Replace each productn $A \to B_1 \dots B_n$ When $n > 2$ with $A \to B_1 C$ where $C \to B_2 \dots B_n$. Repeat this step for all productns having 2 or more symbols in the right side.

Step 5: If the right side of any productn is in the form $A \to aB$ where is a terminal & A, B are non-terminal, then the productn is replaced by

$A \to XB$ & $X \to a$. Repeat this step for every prodn which is in the form $A \to aB$

-problem

convert the following CFG into CNF

$S \to ASA \mid aB, A \to B \mid S, B \to b \mid \varepsilon$

Ans:

1) since S appears in RHS, we add a new state S_0 & $S_0 \to S$ is added to the production set & it becomes -

$S_0 \to S, S \to ASA \mid aB, A \to B \mid S, B \to b \mid \in$

2) NOW we will remove the null productions.

$B \to \in$ and $A \to \in$

After removing $B \to \varepsilon$, the production set becomes -

$S_0 \to S, S \to ASA \mid aB \mid a, A \to B \mid S \mid \in, B \to b$

After removing $A \to \in$, the production set becomes -

$S_0 \to S, S \to ASA \mid aB \mid a \mid AS \mid SA \mid S, A \to B \mid S, B \to b$

3) NOW we will remove the unit productions.

After removing $S \to S$, the production set becomes.

$S_0 \to S, S \to ASA \mid aB \mid a \mid AS \mid SA, A \to B \mid S, B \to b$.

After removing $S_0 \to S$, the production set becomes

~~After removing~~

$S_0 \to ASA \mid aB \mid a \mid As \mid SA$, $S \to ASA \mid aB \mid a \mid As \mid SA$

$A \to B \mid S$, $B \to b$

After removing $A \to B$, the production set becomes-

$S_0 \to ASA \mid aB \mid a \mid As \mid SA$, $S \to ASA \mid aB \mid a \mid As \mid SA$

$A \to S \mid b$

$B \to b$

After removing $A \to S$, the production set becomes-

$S_0 \to ASA \mid aB \mid a \mid As \mid SA$, $S \to ASA \mid aB \mid a \mid As \mid SA$

$A \to b \mid ASA \mid aB \mid a \mid As \mid SA$, $B \to b$

4) Now we will find out more than 2 variables in the R.H.S

Here, $S0 \to ASA$, $S \to ASA$, $A \to ASA$ violates 2 Non-terminals in R.H.S.

Hence we will apply step 4 & step 5 to get the following final production set which is in CNF-

$S_0 \to AX \mid aB \mid a \mid As \mid SA$

$S \to AX \mid aB \mid a \mid As \mid SA$

$A \to b \mid AX \mid aB \mid a \mid As \mid SA$

$B \to b$

$X \to SA$

5) we have to change the productions $S_0 \to aB$, $S \to aB$, $A \to aB$ and the final production set becomes-

$S_0 \rightarrow AX|YB|a|AS|SA$

$S \rightarrow AX|YB|a|AS|SA$

$A \rightarrow bA \rightarrow b|AX|YB|a|AS|SA$

$B \rightarrow b$

$X \rightarrow SA$

$Y \rightarrow a$

- Advantisments

A CFG is in Greibach Normal Form if the prodⁿs are in the following form -

$A \rightarrow b$

$A \rightarrow bD_1 \ldots D_n$

$S \rightarrow \varepsilon$

Where $A, D_1, \ldots, D_n$ are non-terminals & b is a terminal.

- Algorithm to convert a CFG into Greibach Normal Form.

Step 1: If the start symbol 'S' occurs on some right side, create a new start symbol S' & a new productⁿ $S' \rightarrow S$.

Step 2: Remove null productⁿs. (Using the null productⁿ removal algorithm discussed earlier).

Step 3: Remove unit productⁿ. (Using the unit productⁿ removal algorithm discussed

earlier)

Step 4: Remove all direct & indirect left-recursion.

Step 5: Do proper substitutions of productions to convert it into the proper form of GNF.

—problem.

Convert the following CFG into CNF

$S \rightarrow XY \mid Xn \mid p$

$X \rightarrow mX \mid m$

$Y \rightarrow Xn \mid o$

Ans: Here, S doesn't appear on the right side of any production & there are no unit or null productions in the production rule set. So, we can skip step 1 to step 3

step 4: Now after replacing.

X in $S \rightarrow XY \mid Xo \mid p$

with

$mX \mid m$

we obtain

$S \rightarrow mXY \mid mY \mid mXo \mid mo \mid p.$

And after replacing

X in $Y \rightarrow Xn \mid o$

with the right side of

$X \rightarrow mx \mid m$

we obtain

$Y \rightarrow mxn \mid mn \mid o.$

Two new production $O \rightarrow o$ & $P \rightarrow p$ are added to the production set & then we came to the final CNF as the following-

$S \rightarrow mxy \mid my \mid mxc \mid mc \mid p$

$X \rightarrow mx \mid m$

$Y \rightarrow mxo \mid mo \mid o$

$O \rightarrow o$

$P \rightarrow p$

MOD-5 PUSH DOWN AUTOMATA

— Basic structure of PDA

A PDA is a way to implement a CFG in a similar way we design DFA for a regular grammar. A DFA can remember a finite amount of information, but a PDA can remember an infinite amount of information.

Basically a pushdown automaton is:

Finite machine + stack.

— A pushdown automaton has 3 components:

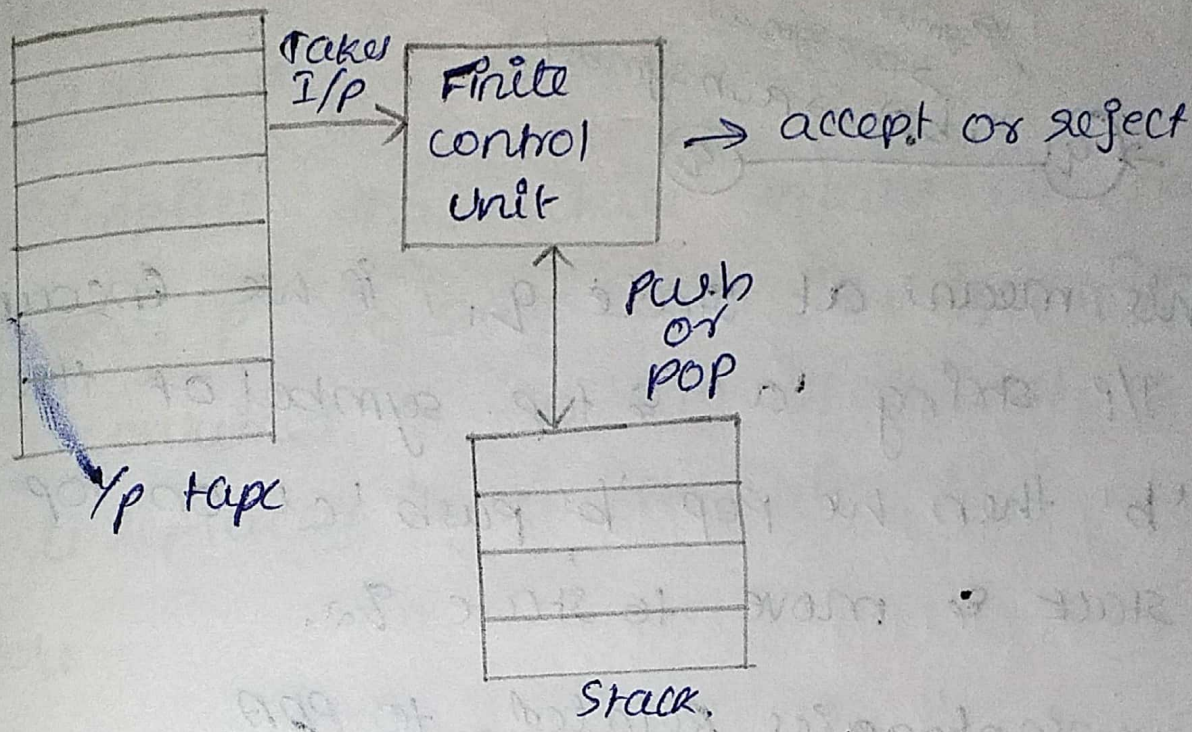
* Input tape
* a controlunit
* a stack with infinite size.

The stack head scans the top symbol of the stack.

A stack does 2 operation:

* Push: a new symbol is added at the top.
* Pop: The top symbol is read & removed.

A PDA may or may not read an I/p symbol but has to read the top of the stack in every transitn.

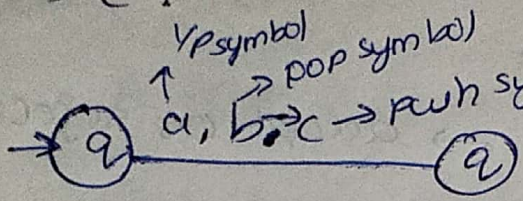


A PDA can be formally described as a 7 tuple $(Q, \Sigma, S, \delta, q_0, I, F)$:

* Q = the Finite no. of states.
* Σ is the I/p alphabet.
* S = stack symbol.
* δ is the transitn fun^n: $Q \times (\Sigma \cup \{\in\}) \times S \times Q \times S^*$
* q_0 is the initial stack ($q_0 \in Q$)
* I is the initial stack top symbol ($I \in S$)

* F is a set of accepting states ($F \in Q$)

— The following diagram shows a transition in a PDA from a state q_1 to state q_2, labeled as $a, b \to c$.



This means at state q_1, if we encounter an I/P string 'a' & top symbol of the stack is 'b' then we pop 'b', push 'c' on top of the stack & move to state q_2.

— Terminologgies Related to PDA

* Instantaneous Descriptn (ID)

It is represented by a triplet (q, w, s) where:

* q is the state
* w is unconsumed I/P
* s is the stack contents.

— Transitive notatn.

It is used for connecting pairs of ID's that represent one or many moves of a PDA. The process of transitn is denoted by the

turnstile symbol "⊢". Consider a PDA (Q, Σ, S, δ, q0, I, F). A transition can be mathematically represented by the following turnstile notation.

(p, a w, T β) ⊢ (q, w, a b)

This implies that while taking a transition from state 'p' to state 'q' the I/p symbol 'a' is consumed & the top of the stack 'T' is replaced by a new string 'a'.

- Note:

If we want zero or more moves of a PDA we have to use the symbol '⊢*'. for it.

- Pushdown Automata Acceptance.

There are 2 different ways to define PDA acceptability.

* Final state acceptability.

A PDA accepts a string when, after reading the entire string, the PDA is in a final state. From the starting state, we can

make moves that end up in a final state with any stack values. The stack values are irrelevant as long as we end up in a final state

For a PDA $(Q, \Sigma, S, \delta, q_0 I, F)$ the language accepted by the set of final states F is:

$L(PDA) = \{ W / (q_0 W, I) \vdash^* (q, \varepsilon, x), q \in F \}$.

for any input stack string x.

* Empty stack acceptability

Here a PDA accepts a string when, after reading the entire string, the PDA has emptied its stack.

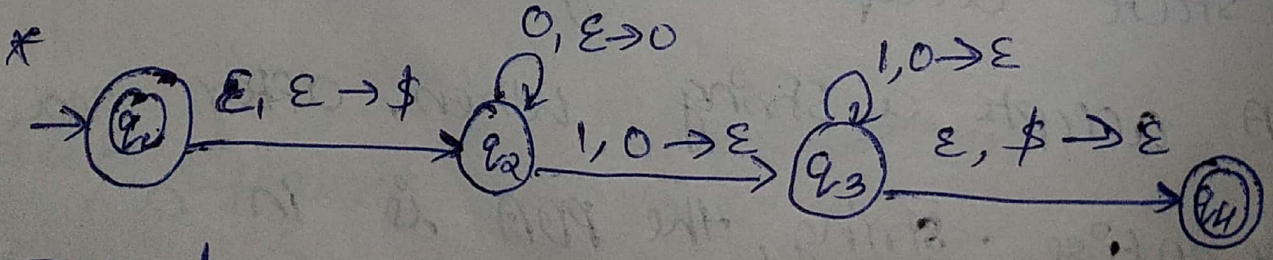
For a PDA $(Q, \Sigma, S, \delta, q_0 I, F)$ the language accepted by the empty stack is

$L(PDA) = \{ W / (q_0, W, I) \vdash^* (q, \varepsilon, \varepsilon), q \in Q \}$

eg:

1) Construct a PDA that accepts.

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



This language accepts

Here in this eg. the no. of 'a' & 'b' have to be same.

* Initially we put a special symbol '$' into the empty stack:

* Then at state q_2, if we encounter i/p 0 & tops is NULL, we push 0 into stack. This may iterate. And if we encounter i/p 1 & top is 0. we pop this 0.

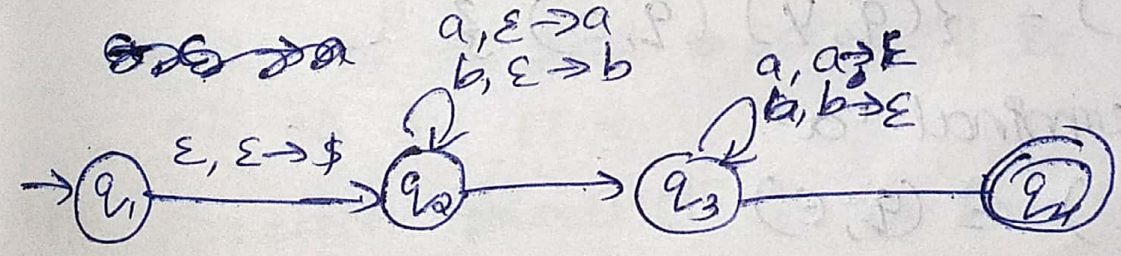
* Then at state q_3, if we encounter i/p 1 & top is 0, we pop the 0. This may also iterate.

& if we encounter i/p 1 & top is 0. we pop the top element.

* If the special symbol symbol '$' is encountered at top of the stack, it is popped out & it finally goes to the accepting state q_4

eg: construct a PDA that accepts

$$L = \{ w w^R / w = a+b^* \}$$



initially we put a special symbol '$' into the

empty stack. At state q_2, the w is being read.

In state q_3 each 0 or 1 is popped when it matches the i/p. If any other i/p is given, the PDA will go to a dead state. When we reach that special symbol '$' we go the accepting state q_4.

2) Construct a PDA that accept $L = \{0^n 1^n / n \geq 0\}$?

An: In this case these are 2 possibilities for accepting a string.

* The string reaches at final state
* The stack will be empty.

→ equivalence of PDA

We can construct a PDA. from CFG.

* Rules.

1) starting symbol.

$$\delta(q, \epsilon, z_0) = (q, S, z_0)$$

2) $X \rightarrow Y/Z$ (For all variables)

$$\delta(q, \epsilon, X) = \{(q, Y)(q, Z)\}$$

3) For all terminal 'a'

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

4) Final state

$$\delta(q, \epsilon, z_0) = (q, \epsilon)$$

PDA & CFG

If a grammar 'G' is CF, we can build an equivalent nondeterministic PDA which accepts the language that is produced by the CFG 'G'. A parser can be built for the grammar 'G'. Also, if 'P' is a PDA an equivalent CFG 'G' can be constructed where

$L(G) = L(P)$.

Algorithm to find PDA corresponding to a given CFG.

I/P = A CFG, $G = (V, T, P, S)$

O/P = Equivalent PDA, $P = (Q, \Sigma, S, \delta, q_0, I, F)$

Step 1: Convert the productions of the CFG into GNF

Step 2: The PDA will have only one state $\{q\}$

Step 3: The start symbol of CFG will be the start symbol in the PDA.

Step 4: All non-terminals of the CFG will be the stack symbols of the PDA & all the terminals of the CFG will be the I/P symbols of the PDA.

Step 5: For each production in the form $A \rightarrow ax$ where a is terminal & A, X are combination of terminal & non terminals make a transition

$\delta(q, a, A)$

1) Construct a PDA from the following CFG

$G = (\{S, X\}, \{a, b\}, P, S)$

where the productⁿs are.

$S \to XS \mid \varepsilon$, $A \to aXb \mid ab \mid ab$

Ans: Let the equivalent PDA

$P = (\{q\}, \{a, b\}, \{a, b, X, S\}, \delta, q, S)$

where δ -

$\delta(q, \varepsilon, S) = \{(q, XS), (q, \varepsilon)\}$

$\delta(q, \varepsilon, X) = \{(q, aXb), (q, Xb), (q, ab)\}$

$\delta(q, a, a) = \{(q, \varepsilon)\}$

$\delta(q, b, b) = \{(q, \varepsilon)\}$

- Algorithm to find CFG corresponding to a given PDA

I/p = A CFG, $G = (V, T, P, S)$

O/p = Equivalent PDA, $P = (Q, \Sigma, S, \delta, q_0, I, F)$.

such that the non-terminals of the grammar G will be $\{XWX \mid W, X \in Q\}$ and the start state will be Aq_0F.

Step1: For every $w, x, y, z \in Q$ mes & $a, b \in \Sigma$, if

$\delta(w, a, \varepsilon)$ contains (y, m) & (z, b, m) contain (x, ε) add the production rule $xwx \to a x_y z b$ in grammar G.

step2: For every $w, x, y, z \in Q$, add the production rule $xwx \to xwy\, xyx$ in grammar G.

step3: For $w \in Q$, add the production rule

$xww \to \varepsilon$ in grammar G.

→ The given grammar $S \to aSb \mid ab$, construct a PDA from this grammar?

Ans: $V = \{S\}$

$T = \{a, b\}$

$\delta(q, \epsilon, S) = (q, aSb)$

$\delta(q, \epsilon, S) = (q, ab)$

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

$\delta(q, b, b) = (q, \epsilon)$

$S \to aSb$

$aabb$

$S \to aSb$
$aabb$

$\delta(q, aabb, S)$

$\vdash (q, aabb, aSb)$

$\vdash (q, abb, Sb)$

$\vdash (q, abb, sb)$

$\vdash (q, abb, abb)$

$\vdash (q, bb, bb)$

$\vdash (q, b, b)$

$\vdash (q, \epsilon, \epsilon)$

1) The CFG is $S \to aBbb$ construct a PDA

From CFG $B \to a$.

$S \to aBbb$
$\to aabb$

$V = \{S, B\}$, $T = \{a, b\}$

$\delta(q, \epsilon S) = (q, aBbb)$

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

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

$\delta(q, b, b) = (q, \epsilon)$

$S \to aBbb$
$\to aabb$

$\delta(q, aabb, S)$

$\vdash (q, aabb, aBbb)$

$\vdash (q, abb, Bbb)$

$\vdash (q, abbabb)$

$\vdash (q, bb, bb)$

$\vdash (q, b, b)$
$\vdash (q, \epsilon, \epsilon)$

2) $S \to aSbc / b$

$S \to aSbc$
$\to abbc$

$V = \{S\}$ • $T = \{a, b, c\}$

$\delta(q, \epsilon, S) = (q, aSbc)$

$\delta(q, \epsilon, S) = (q, b)$

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

$\delta(q, b, b) = (q, \epsilon)$

$\delta(q, c, c) = q \cdot \epsilon$

$S \to abbc$

$\delta(q, abbc, S)$

$\delta(q, abbc, aSbc)$

$\delta(q, bbc, Sbc)$

$\delta(q, bbc, bbc)$

$\delta(q, bc, bc)$

$\delta(q, c, c)$

$\delta(q, \epsilon, \epsilon)$

3) $S \to OBB$

$B \to OS / 1S / 0$

test whether $01D^4$?

An: $V = \{S, B\}$ $T = \{0, 1\}$

string = 010000

$\delta(q, \epsilon, S) = \delta(q, 0BB)$

$\delta(q, \epsilon, B) = \delta(q, 0S)$

$\delta(q, \epsilon, B) = \delta(q, 1S)$

$\delta(q, \epsilon, B) = \delta(q, 0)$

$\delta(q, 0, 0) = \delta(q, \epsilon)$

$\delta(q, 1, 1) = \delta(q, \epsilon)$

$\delta(q, 01000, S)$

$\vdash (q, 01000, 0BB)$

$\vdash (q, 1000, BB)$

$\vdash (q, 000, SB)$

$\vdash (q, 000, 0BBB)$

$\vdash (q, 00, BBB)$

$\vdash (q, 00, 0BB)$

$\vdash (q, 0, 0B)$

$\vdash (q, 0, B)$

$\vdash (q, 0, 0)$

$\vdash (q, \epsilon, \epsilon)$

— Parsing.

Parsing is used to derive a string using the produ

th rule of a grammar. It is used to check the acceptability of a string compiler is used to check whether or not a string is syntaxically correct. A parse take the i/p & build a parse tree.

- Two type of parsing.

* TOP down parsing.

It is start from the top with start symbol & derive a string using a parse tree.

- Designing of Topdown parse

* POP the non terminal on the L.H.S of the production at the top of the stack & push its right hand side string.
* If the top symbol of the stack matches with the i/p symbol the pop it.
* push the start symbol s into the stack.
* If the i/p string is fully read & the stack is empty then it go to the final state f.

eg: $(\{S, A, B\}, \{a, b\}, P, S)$

$S \rightarrow aAB$

$S \rightarrow bBA$

$A \rightarrow bS$

$A \rightarrow a$

$B \rightarrow aS$

B → b

W = abbbaab

S → aAB

⇒ abSB

⇒ abbBAB

⇒ abbbAB

⇒ abbbaB

⇒ abbbaab

* Bottom up passing.

Bottom up passing start from the bottom with the string & comes to the start symbol with a parse tree.

— Designing of a bottom up parse

* push the current i/p symbol into the stack.
* Replace the R.H.S of the product^{n} at the top of the stack with its L.H.S
* If the top of the stack element matches with the current i/p symbol then pop it.
* If the i/p string is fully read & only if start symbol remains in the stack. Then pop it & go to final state F.

eg: $S \to aABe$ the i/p string abbcde
$A \to Abc \mid b$
$B \to a$

aAbcde ($A \to b$)
aAde ($A \to Abc$)
aABe ($A \to Abc$)
aABe ($B \to b$)
S ($S \to aABe$)

$S \to aABe$
$\to aAde$
$\to aAbcde$
$\to abbcde$

- Turing machine.

A.T.M is an accepting device which accepts the language (recursively enumerable set) generated by type 0 grammar. It was Invented in 1936 by Alan Turing.

- Definition.

A T.M is a mathematical model which consist of an infinale length tape divided into cells on which i/p is given. It can consists of a head which reads the i/p tape. A state register stores the state of the turing machine. After reading an i/p symbol it is replaced with another symbol. Its internal state is changed, & it moves from one cell to the right to left. If the T.M reaches the final state the

I/p string is accepted otherwise rejected.

A T.M can be formally described as a 7-tuple $(Q, X, \Sigma, \delta, q_0, B, F)$ where.

* Q is a finate set of states
* X is the tape alphabet
* Σ is the I/p alphabet
* δ is a transition fun^n; $\delta: Q \times X \rightarrow Q \times X \times \{\text{left-shift}, \text{Right-shift}\}$
* q_0 is the initial state
* B is the black symbol.
* F is the set of final states

The following table shows a comparison of how a T.M differs from finate automaton & PDA

Machine	stack data structure	Deterministic?
F.A	N.A	Yes
PDA	LIFO	ND
T.M.	Infinate tape	Yes

eg: T.M $M = (Q, X, \Sigma, \delta, q_0, B, F)$ with.

* $Q = \{q_0, q_1, q_2, q_f\}$
* $X = \{a, b\}$
* $\Sigma = \{1\}$

* $q_0 = \{q_0\}$
* B = Blank symbol.
* $F = \{q_F\}$
* δ is given by

Tape Alphabet symbol	Present state 'q0'	Present State 'q1'	Present state 'q2'
a	$1Rq_1$	$1Lq_0$	$1Lq_F$
b	$1Lq_2$	$1Rq_1$	$1Rq_F$

Here the transit^n $1Rq_1$ implies that the the ~~next state is~~ q_1, write symbol is 1. The tape move right, & the next state is q_1. Similarly, the transit^n $1Lq_2$ implies that the write symbol is 1. The tape moves left & next state is q_2.

– Time & space complexity of a T.M.

For a T.M, the time complexity refers to the measure of the no. of times the tape moves when the machine is initialised for some i/p symbol & the space complexity is the no. of cells of the tape written.

Time complexity all reasonable fun^{ns}:

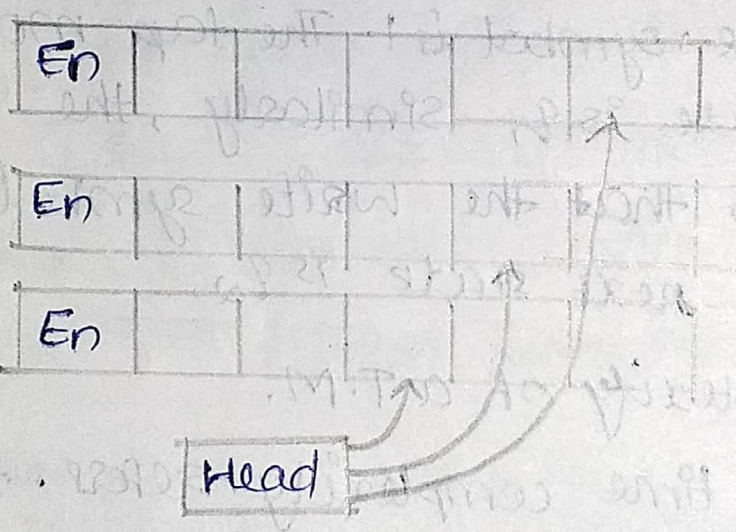
$T(n) = O(n \log n)$

T.M's space complexity:

$S(n) = O(n)$

- Multitape turing machine.

Multitape T.M have multiple tapes where each tape is accessed with a seperate head. Each head can move independently of the other heads. Initially the i/p is on tape1 & others are blank. At first, the 1st tape is occupied by the i/p & the other tapes are kept blank. Next the machine reads consecutive symbols under its heads & the T.M prints heads.



A multitape T.M can be formally described as a 7 tuple $(Q, X, B, \delta, q_0, F)$

Where

* Q = is a finate set of states
* X is the tape alphabet
* B is the blank symbol.
* δ is a relatn on state & symbols where

$\delta : Q \times X^K \to Q \times (X \times \{\text{Left shift, Right shift, No-shift}\})^K$

where there is 'K' no. of tapes.

* q_0 is the initial state
* F is the final state

~~Multi-track~~

— Multi track T.M

Multi track T.M a specific types of multitape tracks but just one tape head reads & writes on all tracks Here, a single tape head reads n symbols from n tracks at one step. It accepts recursively enumerable languages like a normal single-track single-tape. T.M accepts

A multitrack T.M can be formally described as a 6 tuple $(Q, X, \Sigma, \delta, q_0, F)$

Where:

* Q - finate set of states.
* X - tape alphabet
* Σ is the i/p alphabet
* δ is a relatn on states & symbols.

Where.

$\delta(Q_i, [a_1, a_2, a_3, \ldots]) = (Q_j, [b_1, b_2, b_3 \ldots], \text{left shift or Right-shift})$

* q_0 is the initial state.