UNIT – I
Fundamentals

Mathematical Notions:

- **Set:** It is a group of objects represented as a unit. Objects are called as elements.
  eg:{1,2,3,4,5,.} or{ a, b, c, d,...}
  - Є set membership (belongs to)
  - ∉ set non-membership(not belongs to)
  - ⊂ subset
  - ⊆ subset or equal to
    - A subset, A B, means that every element of A is also an element of B:
    - If x Є A, then x Є B.
  - empty set (ϕ)(epsilon)

- **Set Operations :** There are 4 basic set operations: **union, intersection, complement**, and **difference**.
  - **Union:**
    - $A \cup B$ is the set that contains all the elements in either A or B or both: $A \cup B = \{x \mid x \in A \text{ or } x \in B\}$.
    - For example, if A = { 1, 2, 3} and B ={ 3, 4, 5}, then $A \cup B$ = { 1, 2, 3, 4, 5}
  - **Intersection:**
    - $A \cap B$ is the set that contains all the elements that are in both A and B: $A \cap B = \{x \mid x \in A \text{ and } x \in B\}$.
    - For example, if A = { 1, 2, 3} and B ={ 3, 4, 5}, then $A \cap B$ = { 3 }
  - **Complement:**
    - A' is the set that contains everything in the universal set that is not in A: $A' = \{x \mid x \in \boldsymbol{U} \text{ and } x \notin A\}$.
    - For example, if $\boldsymbol{U}$= { 2, 4, 6, 8, 10, 12} and A = { 2, 4}, then A' ={ 6, 8, 10, 12}
  - **Difference:**
    - $A - B$ is the set that contains all the elements that are in A but not in B: $A - B = \{x \mid x \in A \text{ and not } x \notin B\}$.
    - For example, if A = { 1, 2, 3} and B ={ 3, 4, 5}, then $A - B$ = { 1, 2 }

- **Symbols:** An abstract entity that has no meaning by itself, Letters from various alphabets, digits and special characters are the most commonly used symbols.

- **Alphabet:** A finite set of symbols. An alphabet is often denoted by sigma(∑), yet can be given any name.

  A={a, b, c, d,...z} Says A is an alphabet that represent English letters.

B = {0, 1} Says B is an alphabet of two symbols, 0 and 1.
C = {a, b, c} Says C is an alphabet of three symbols, a, b and c.
D={0,1,2,3,4,5,6,7,8,9} Says D is an alphabet represent digits

- **Powers of an alphabet:** If Σ is an alphabet, then $\Sigma^0 = \varepsilon$, no matter what the alphabet Σ is. In other words ε is the only string of length 0.
  - If Σ = {a, b, c} then Σ^1 = {a, b, c}, Σ^2 = {aa, ab, ac, ba, bb, bc, ca, cb, cc}, Σ^3 = {aaa, aab, aac, aba, abb, abc, aca, acb, acc, baa, bab, bac, bba,bbb, bbc, bca, bcb, bcc, caa, cab, cac, cba, cbb, cbc, cca, ccb, ccc}
  - The set of all strings over an alphabet Σ is conventionally denoted by Σ*. For instance {0, 1}* = {ε, 0, 1, 00, 01, 10, 11, 000,… }. Another way is $\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \ldots$
  - The set of nonempty strings of an alphabet Σ is denoted as Σ+. And the two appropriate equivalences are :$\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \ldots$ $\Sigma^* = \Sigma^+ \cup \{\varepsilon\}$

- A String over an alphabet Σ is a finite sequence of symbols of Σ.

String over an alphabet is a finite sequence of symbols from the alphabet, usually written next to one another and not separated by commas.

- If Σ_1 = {0, 1} then 01001 is a string over the alphabet Σ_1.
- If Σ_2 = {a, b, c,…, z} then abracadabra is a string over Σ_2.
- If w is a string over Σ then the **length** of w is written as **|w|** which is the number of symbols that it contains.
- The string of length zero is called the **empty string**. It is written as **ε(Epsilon)**.
- The empty string plays the role of 0 in a number system. If **w** has length n then we can write **w =w1, w2, …, w_s** where each $\mathbf{w_i} \in \Sigma$.
- The **reverse** of **w** written as $\mathbf{w^R}$ is the string obtained by writing b in the opposite order (i.e. $w_n, w_{n-1}, \ldots, w_1$).

Operations on strings:

- ***Concatenation of strings***

  The concatenation of two strings is the string formed by writing the first, followed by the second, with no intervening space. Concatenation of strings is denoted by ◦.

  That is, if w and x are strings, then wx is the concatenation of these two strings.

  Example:

  The concatenation of dog and house is doghouse.

  Let x=0100101 and y= 1111 then x ◦ y=01001011111

- ***String Reversal***

  Reversing a string means writing the string backwards.

  It is denoted by w^R

  Example:

  Reverse of the string abcd is dcba.

  If $w= w^R$, then that string is called palindrome.

- ***Substring***

A substring is a part of a string.

Example:

If abcd is string then possible substrings are **ε**,a,b,c,d,ab,bc,cd,abc,bcd are proper substrings for the given string

A ***prefix*** of a string is any number of leading symbols of that string.

A ***suffix*** of a string is any number of trailing symbols.

Example:

String abc has prefixes **ε**, a, ab, and abc; its suffixes are **ε**, c, bc, and abc.

A prefix or suffix of a string, other than the string itself, is called a ***proper prefix or suffix***.

- A **language** is a set of strings.
  - A set of strings all of which are chosen from Σ*, where Σ is a particular alphabet, is called language.
  - **Examples:**
    - The language of all strings consisting of n 0's followed by n 1's for some n≥0; { ε, 01, 0011, 000111, . . . }.
    - The set of strings of 0's and 1's with an equal number of each : {ε, 01, 10, 0011, 0101, 1001, . . .}
    - The set of binary values whose value is prime: {10, 11, 101, 111, 1011,. . .}
    - Σ* is a language for any alphabet Σ.
    - ϕ the empty language is a language over any alphabet.
    - {ε} the language consisting of only the empty string, is also a language over any alphabet.

Operations on languages:

- ***Union***

  If L1 and L2 are two languages over an alphabet ∑.Then the union of L1 and L2 is denoted by L1 U L2.

  Example:

  L1={0,01,011} and L2={001},then L1 U L2={0,01,011,001}

- ***Intersection***

  If L1 and L2 are two languages over an alphabet ∑.Then the intersection of L1 and L2 is denoted by L1 ∩ L2.

  Example:

  L1= {0, 01,011} and L2= {01}, then L1 ∩ L2= {01}

- ***Complementation***

  L is a language over an alphabet ∑, then the complement of L denoted by L^{-} ,is the language consisting of those strings that are not in L over the alphabet.

  Example:

  If ∑={a,b} and L={a,b,aa} then L^{-}= ∑* - L={ **ε,a,b,aa,bb,ab.....}-{a,b,aa}={ε,bb,ab,ba.....}**

- ***Concatenation***

  Concatenation of two languages L1 and L2 is the language L1 o L2 ,each element of which is a string formed by combining one string of L1 with another string of L2.

  Example:

  L1={bc,bcc,cc}andL2={cc,ccc},thenL1oL2={bccc,bcccc,bccccc,cccc,ccccc}

- ***Reversal***

  If L is language, then L^R is obtained by reversing the corresponding string in L. This operation is similar to the reversal of a string.

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

  Example:

  If L= {0, 011, 0111}, then L^R = {0, 110, 1110}

- ***Kleene Closure***

  The Kleene closure (or just closure) of L, denoted L*, is the set and the positive closure of L, denoted L^+ , is the set

  $$L^* = \bigcup_{i=0}^{\infty} L^i$$

  $$L^+ = \bigcup_{i=1}^{\infty} L^i$$

  That is, L* denotes words constructed by concatenating any number of words from L. L+ is the same, but the case of zero words, whose "concatenation" is defined to be **ε**, is excluded. Note that L+ contains **ε** if and only if L does.

  Example:

  Let L1 = {10, 1}

  Then **L *** = L0 U L1 U L2..... = {ε, 1, 10, 11, 111, 1111 ,.....}

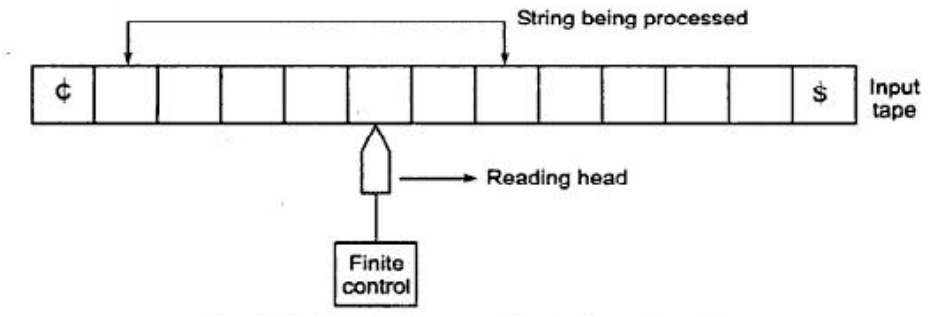
  L $^+$ = L1 U L2 U L3.....= {1, 10, 11, 111, 1111.....}

FINITE STATE MACHINE

- A finite-state machine (FSM) or simply a state machine is a mathematical model of computation used to design both computer programs and sequential logic circuits.
- It is conceived as an abstract machine that can be in one of a finite number of states. The machine is in only one state at a time; the state it is in at any given time is called the current state.

- It can change from one state to another when initiated by a triggering event or condition, this is called a transition. A particular FSM is defined by a list of its states, and the triggering condition for each transition.
- Finite State machine is also termed as finite automaton.
- Finite automata are good models for computers with an extremely limited amount of memory.
- The controller for an automatic door is one example of such a device which is often found at supermarket entrances and exits.
- Also automatic doors swing open when sensing that a person is approaching. It has a pad in front to detect the presence of a person about to walk through the doorway.
- This controller is a computer that has just a single bit of memory, capable of recording which of the two states the controller is in.
- Other common devices have controllers with somewhat larger memories. In an elevator controller a state may represent the floor the elevator is on and the inputs might be the signals received the buttons.
- This computer might need several bits to keep track of the information. Controllers for various household appliances like dishwashers, electronic thermostats, as well as parts of digital watches and calculators are additional examples of computers with limited memories.
- The design of this kind of devices requires keeping the methodology and terminology of finite automata in mind.
- Finite automata and their probabilistic counterpart Markov Chains are useful tools when we are attempting to recognize patterns in data. These devices are used in speech processing and in optical character recognition.

Finite Automaton Model:



Block diagram of a finite automaton

The various components are explained as follows:

(i) ***Input tape:***

- The input tape is divided into squares, each square containing a single symbol from the input alphabet ∑.
- The end squares of the tape contain the endmarker ¢ at the left end and the endmarker $ at the right end.

- The absence of endmarkers indicates that the tape is of infinite length. The left-to-right sequence of symbols between the two endmarkers is the input string to be processed.

(ii) ***Reading head:***

- The head examines only one square at a time and can move one square either to the left or to the right.
- For further analysis, we restrict the movement of the R-head only to the right side.

(iii) ***Finite control:*** The input to the finite control will usually be the symbol under the R-head, say a, and the present state of the machine, say q, to give the following outputs:

(a) A motion of R-head along the tape to the next square (in some a null move, i.e. the R-head remaining to the same square is permitted)

(b) The next state of the finite state machine given by δ(q, a).

ACCEPTANCE OF STRINGS AND LANGUAGES:

ACCEPTABILITY OF STRING BY FINITE AUTOMATON

- Def: A string x is accepted by a finite automaton M = (Q, Σ, δ, q0, F) if δ(q0, x) =q for some q Є F. This is basically the acceptability of a string by the final state.
- Note: A final state is also called an accepting state.
- If A is the set of all strings that machine M accepts then we say that A is the language of machine M and written as L(M) = A. we say that M recognizes A or M accepts A.

Representation of FA:

Finite automaton can be represented by using two ways

1. Transition Table
2. Transition Diagram

Transition Tables:

- A transition table is a conventional tabular representation of a function like δ that takes two arguments and returns a value.
- The rows of the table correspond to the states and columns correspond to the inputs. The entry for the row corresponding to state q and the column corresponding the input a is the state δ(q, a).

Example:

- The following figure shows the transition table, where the start state is marked with an arrow, and the accepting states are marked with a star.
- Since we can deduce the sets of states and input symbols by looking at row and column heads, we can now read from the transition table all the information we need to specify the finite automaton uniquely.

	0	1
→ q0	q2	q0
*q1	q1	q1
q2	q2	q1

Transition Diagram:

A transition diagram for DFA A = (Q, Σ, δ, q0, F) is a graph defined as follows:

a) For each state in Q there is a node

b) For each state q in Q and each input symbol a in Σ, let δ(q, a) = p. then the transition diagram has an arc from node q to node p labeled a. If there are several input symbols that cause transitions from q to p then the transition diagram can have one arc labeled by the list of these symbols

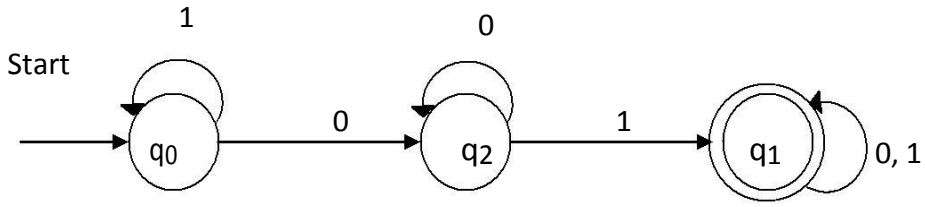
c) There is an arrow into the start state q0, labeled start. This arrow does not originate at any node.

d) Nodes corresponding to accepting state (those in F) are marked by a double circle. States not in F have a single circle.

Example:

The following figure shows the transition diagram for the DFA that we designed in three states. There is a start arrow entering the start state q0, and the one accepting state q1, is represented by double circle.

- Out of each state is one arc labeled 0 and one arc labeled 1, although the two arcs are combined into the one with a double label in the case q1.



Types of Finite Automata

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 $(Q, \sum, \delta, q_0, F)$ where −

- **Q** is a finite set of states.
- $\sum$ is a finite set of symbols called the alphabet.
- **δ** is the transition function where $\delta: Q \times \sum \rightarrow Q$
- $\mathbf{q_0}$ is the initial state from where any input is processed $(q0 \in Q)$.
- **F** is a set of final state/states of Q $(F \subseteq Q)$.

Non-Deterministic Finite Automaton (NFA or NDFA):

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, ∑, δ, 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 δ: Q × {∑ ∪ ε} → 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 ∈ Q).
- **F** is a set of final state/states of Q (F ⊆ Q).

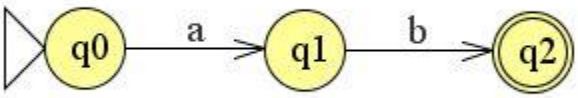
Examples:

1. **Design NFA over an alphabet {a,b} which accepts ab as substring**

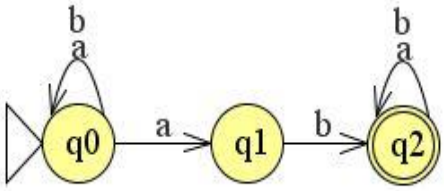
   Step 1: Understand the language and write the strings generated by given language

   L={ab,aab,abb,aabab,aaaabbbbb,.....}

   Step 2: Construct the automaton for the minimum string in the language



   Step 3: Put the possible transitions on the each state with each input symbol with respect to the given condition.



   Step 4: Tuple representation

   M=(Q,∑, δ ,q0,F) where

   Q={q0,q1,q2}

   ∑={a,b}

   q0=initial state=q0

F={q2}

δ=transition function

Step 5: String Verification(*If required)

δ(q0,aabba)= δ(q0,abba)

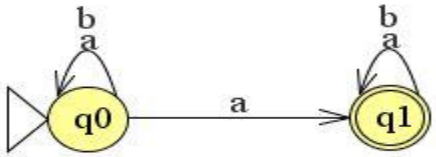
= δ(q1,bba)

= δ(q2,ba)

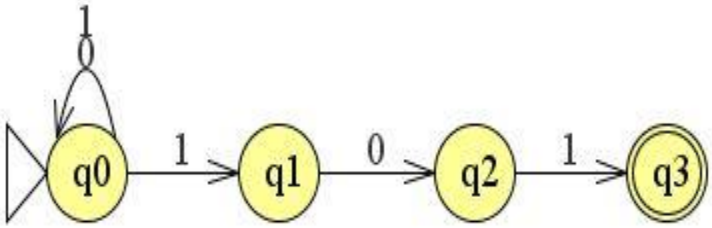
= δ(q2,a)

= q2 --> **Final state (accepted)**

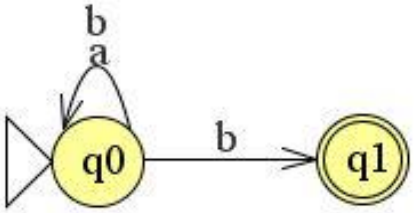
2. **Design NFA over an alphabet {a,b} which contains a.**



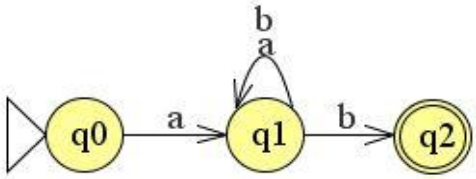
3. **Design NFA which can accept strings over the alphabet {0,1} ending with 101**



4. **Design NFA which can accept strings over the alphabet {a,b} ending with b**



5. **Design NFA which can accept strings over the alphabet {a,b} starting with a ending with b**

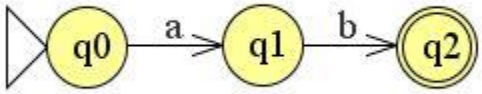


6. Design DFA over an alphabet {a,b} which accepts ab as substring

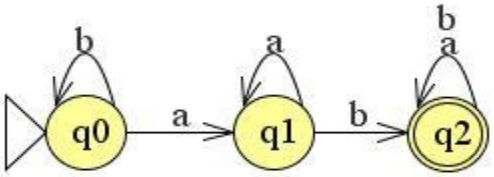
Step 1: Understand the language for which the DFA has to be designed and write the language for the set of strings starting with minimum string that is accepted by FA.

L={ab,aab,aaababa,baab,.....}

Step 2: Draw transition diagram for the minimum length string.



Step 3 : Obtain the possible transitions to be made for each state on each input symbol.



Step 4: Tuple Representation:-

Q=finite set of states={q0,q1,q2 }

∑=finite set of input alphabet={a,b}

q0=initial state=q0

F=set of final states={q2}

Step 5: String Verification(*If required)

δ(q0,aabba)= δ(q1,abba)

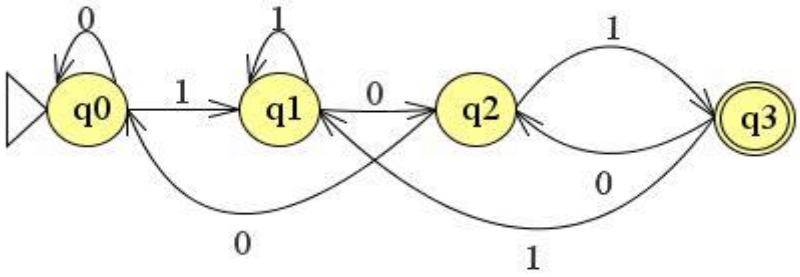
= δ(q1,bba)

= δ(q2,ba)

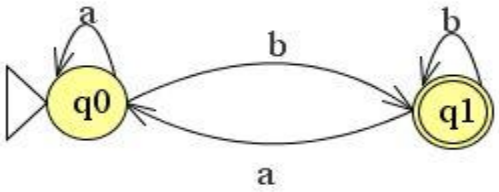
= δ(q2,a)

= q2 --> **Final state (accepted)**

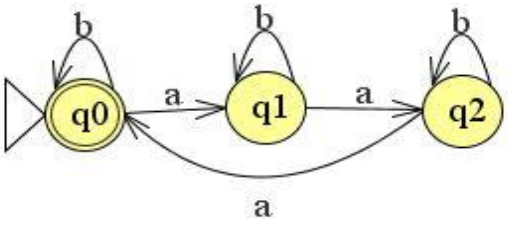
7. **Construct a transition system which can accept strings over the alphabet {0,1} ending with 101(DFA)**



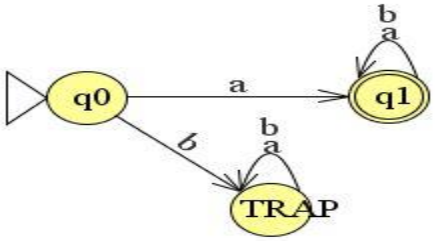
8. **Construct DFA which can accept strings over the alphabet {a,b} ending with b**



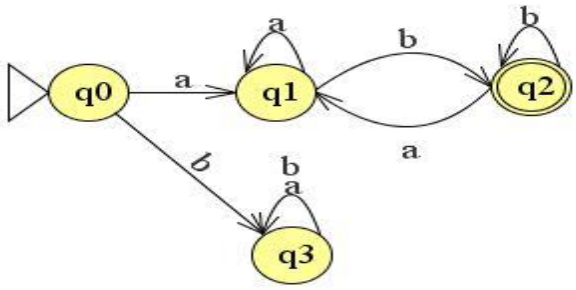
9. **Construct DFA which can accept strings over the alphabet {a,b} number of a's divisible by 3**



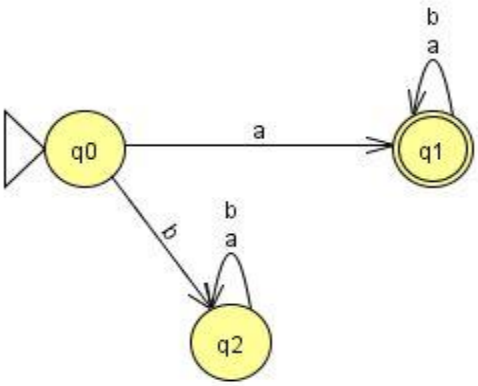
10. **Construct DFA which can accept strings over the alphabet {a,b} starting with a.**



11. Construct a transition system which can accept strings over the alphabet {a,b} starting with a ending with b



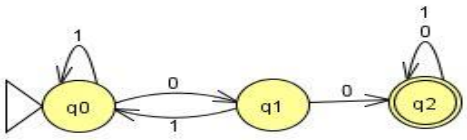
12. Design DFA that accepts all strings which starts with 'a' over the alphabet {a,b}



Transition table

State	Input	
	a	b
q_0	q_2	q_1
q_1	q_1	q_1
q_2	q_2	q_2

13. Design DFA that accepts all strings which contains '00' as substring over the alphabet {0,1}



Tuple Representation:

M(Q, ∑, δ, q_0, F) where

Q= finite set of states={ q0,q1,q2}
∑=finite set of inputs={0,1}
δ=transition function maps
q_0 =initial state=q0
F=set of final states={q2}

Language recognizers:

A language recognizer is a device that accepts valid strings produced in a given language. Finite state automata are formalized types of language recognizers.
The language accepted by Finite Automata M designated L(M) is the set {x | δ(q0,x) is in F}.

Applications of FA:

- Used in Lexical analysis phase of a compiler to recognize tokens.
- Used in text editors for string matching.

Differences between NFA and DFA:

S. No	NFA	DFA
1	A nondeterministic finite automaton is a 5-tuple **M= (Q, ∑, δ, q_0, F)**, where **δ**: Q x ∑ into 2^Q.	A deterministic finite automaton can be represented by a 5-tuple **M= (Q, ∑, δ, q_0, F),** where **δ**: Q x ∑ to Q.
2	NFA is the one in which there exists many paths for a specific input from current state to next state.	DFA is a FA in which there is only one path for a specific input from current state to next state.
3	NFA is easier to construct.	DFA is more difficult to construct.
4	NFA requires less space.	DFA requires more space.
5	Time required for executing an input string is more.	Time required for executing an input string is less.
6	Transitions could be non deterministic	All transitions are deterministic

7	A transition could lead to subset of states.	Each transition leads to exactly one state
8	For each state not all symbols necessarily have to defined in the transition functions	For each state, transition on all possible symbols should be defined
9	Accepts input if one of the last states is in F	Accepts input if the last state is in F
10	Generally easier than a DFA to construct	Sometimes harder to construct because of the number of states.

Regular Grammars:

Grammar:

A grammar is defined as **G = (V, T, P, S)**. Where

- **V** is a finite nonempty set whose elements are called ***variables***.
- **T** is a finite nonempty set whose elements are called ***Terminals***.
- **P** is a finite set whose elements are α→β, where α and β are strings on **V U T**. And α has at least one symbol from **V**. Elements of **P** are called ***productions*** or ***production rules*** or ***rewriting rules***.
- **S** is a special variable i.e. an element of **V** called the ***start symbol*** and **V ∩T = φ**

We observe the following regarding the production rules:

- Reverse substitutions is not permitted. For example if S →AB is a production then we can place S by AB but we cannot replace AB by S.
- No inversion operation is permitted. For example if S →AB is a production then, it is not necessary that AB→S is a production.

Example:

G = (V, T, P, S) is a grammar

Where V = {<sentence>, <noun>, <verb>, <adverb>} T = {Ram, Sam, ate, sang, well}

S = <sentence>

Then **P** consists of the following productions:

<sentence>→<noun> <verb>
<sentence>→<noun> <verb> <adverb>
<noun> →Ram
<noun>→Sam
<verb>→ate
<verb>→sang
<adverb>→well

Note: we use comma "," as a separator to separate multiple productions and alternation "|" to put several productions together.

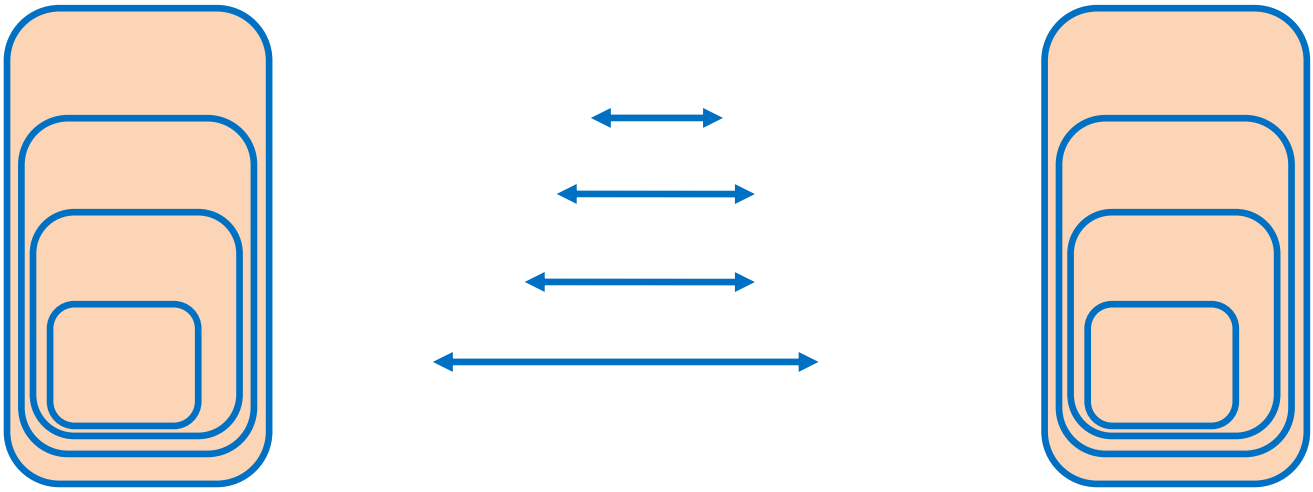
- **Derivation:** Derivation is an ordered tree which is defined as sequence of replacements of a substring in a sentential form. Productions are used to derive one string over V U T from another string.
  - The formal definition of derivation is as follows:

- If α→β is a production in a grammar G and ϒ, δ are any two strings on V U T, then we say
- ϒαδ directly derives ϒβδ in G. this process is called as one step derivation.

- **For example**
  - G = ({S}, {0, 1}, {S→0S1, S→01} has the productions S→0S1. So ***S*** in 0***S***1 can be replaced by ***0S1*** i.e. now S→0***0S1***1.

Chomsky Hierarchy of Languages:

Chomsky classified the grammars into four types in terms of productions (types 0-3) which is as shown in the table below:



Grammar	Languages	Automaton	Production Rules
Type 0	Recursively enumerable/ Phrase Structured	Turing machines	α→β
Type 1	Context Sensitive Language	Linear Bounded Automata	α→β | α |<=| β |
Type 2	Context Free Languages	Push Down Automata	A→α
Type 3	Regular Languages	Finite Automata	A →w A→wB A→Bw

Type 0 Grammar:-

This grammar is also called as phase structured grammar. In this grammar the Right Hand Side of production are free from any restriction. This is also called as unrestricted grammar. The language generated by Type 0 grammar is Recursively Enumerable Language.

Type 0 Grammar:-

It is also called as context sensitive grammar. the production should be of the p form **α→β** Such that |α|≤|β| the grammar which contains all type 1 productions such grammar is called Type 1 grammar or context sensitive grammar.
Language generated by the context sensitive grammar (CSG) is called Context Sensitive Language(CSL).

Example:- 1. S-->aS, S-->AB,S-->Aab
2. AS-->a (not valid)

Type 2 Grammar:-

A production of the form **α→β** , such that **β** ∈ (V U T)*. Then it is called Type 2 production. If a grammar contains all type 2 productions. That is called as type 2 grammar. This is also called as Context Free Grammar. The language generated by the Context Free Grammar is called as **Context Free Language(CFL)**

Example:
S-->aA
S-->BAD

Type 3 Grammar:-

Type-3 grammars (regular grammars) generate the regular languages. Such a grammar restricts its rule to a single non-terminal on the left-hand side and a right-hand side consisting of a single terminal, possibly followed or precedes, but not both in the same grammar by a single non- terminal. A production S→ε is allowed in Type-3 grammar, but in this case S does not appear on the right hand side of any production.

Example:- S-->a, S-->b, S-->aA, A-->a

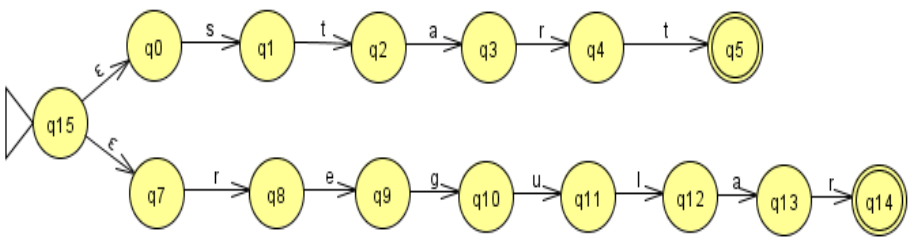
UNIT – II
Finite Automata
Conversions and Equivalence

Nondeterministic Finite Automata With ε-Transitions:

- In the automata theory, a nondeterministic finite automaton (NFA) or nondeterministic finite state machine is a finite state machine where from each state and a given input symbol the automaton may jump into several possible next states.
- This distinguishes it from the deterministic finite automaton (DFA), where the next possible state is uniquely determined.
- Although the DFA and NFA have distinct definitions, a NFA can be translated to equivalent DFA using power set construction, i.e., the constructed DFA and the NFA recognize the same formal language.
- Both types of automata recognize only regular languages. NFAs are sometimes studied by the name subshifts of finite type. NFAs have been generalized multiple ways, e.g., nondeterministic finite automaton with ε-moves, pushdown automaton.
- In the automata theory, a nondeterministic finite automaton with ε-moves (NFA-ε) is an extension of nondeterministic finite automaton (NFA), which allows a transformation to a new state without consuming or reading any input symbols.
- The transitions without consuming an input symbol are called ε-transitions. In the state diagrams, they are usually labeled with the Greek letter ε.

Use of ε-transitions

- To build an NFA that recognizes a set of keywords, the strategy can be simplified further if we allow the ε-transitions.
- For example consider the NFA recognizing the keywords **state** and **regular**, which is implemented with ε-transitions as in the figure below:



- In general we construct a complete sequence of states for each keyword, as if it were the only word the automaton needed to recognize.
- Then we add a new start state (state 15 in above figure) with ε-transitions to the start-states of the automata for each of the keywords.

Formal Notation for an ε-NFA

- We may represent an ε-NFA exactly as we do an NFA, with one exception: the transition function must include information about transitions on ε.
- Formally we represent an ε-NFA A by A = (Q, Σ, , q_0,F), where all components have their same interpretations as for an NFA, except that is now a function that takes as arguments
  1. A state in Q, and
  2. A number of Σ ∪ ,ε-, i.e. either an input or the symbol ε. We require that ε, the symbol for the empty string, cannot be a member of the alphabet Σ, so no confusion results.
- In the above ε-NFA A
  1. Q = Finite set of states
  2. Σ =input Alphabet
  3. δ=Transition function (δ: Q × (Σ ∪,ε-) → P(Q))
  4. q_0= start state in Q
  5. F = a set of final states F ⊆ Q.

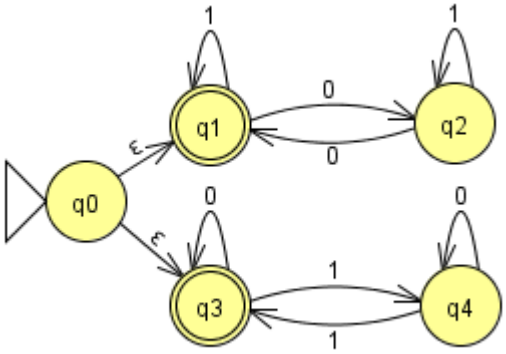
Equivalence between NFA with and without ε transitions:

- It can be shown that ordinary NFA and NFA-ε are equivalent, in that, given either one, one can construct the other, which recognizes the same language.

Example:

- Let *M*be a NFA-ε, with a binary alphabet, that determines if the input contains an even number of 0s or an even number of 1s.
- Note that 0 occurrences is an even number of occurrences as well. In formal notation, let *M* =({q_0, q1, q_2, q_3,q_4}, {0, 1}, δ,q_0, {q1,q_3}) where the transitionrelation can be defined by thestate transition tablegiven below:

Q/ Σ	0	1
q0	-	-
q1	q2	q1
q2	q1	q2
q3	q3	q4
q4	q4	q3



- M can be viewed as the union of two DFAs: one with states {q1, q2} and the other with states {q3, q4}.
- The language of M can be described by the regular language given by this regular expression (1*(01*01*)*) ∪ (0*(10*10*)*). We define M using ε-moves but M can be defined without using ε-moves.

Epsilon – Closure

- ε-closure of a state q is the set of states that can be reached from q along a path in which all arcs are labeled with ε.
- If q is in ε-closure then it is denoted as ε-closure(q). If p is in ε-closure(q) and there is an ε transition from p to r, then r is in ε-closure(q).
- NFA's are said to be closed under the following operations:
    1. Union
    2. Intersection
    3. Concatenation
    4. Negation
    5. Kleene closure
- Kleene star or Kleene operator or Kleene closure is a unary operation, performed either on sets of strings or on sets of symbols or characters. The application of the Kleene star to a set
- **V** is written as **V***.

It is widely used for regular expressions to characterize certain automaton where it means zero or more.

ACCEPTANCE OF LANGUAGES

- Like DFA the transition function on an NFA (A) is uniquely determined by A. According to the formal definition of NFA, it is a 5-tuple consisting of A = (Q, Σ, , q_0, F)
- Let us consider a string **w** over an Σ.
- **w** is accepted by Aif there is an accept state qЄF such that q is reachable from a start stateunder input string **w** i.e. q Є (q_0,**w**)
- The language that is accepted by A is denoted by L(A), is the set of all the strings accepted by A. i.e. L(A) = {**w** | **w** Є Σ*}
- Let M denotes DFA and N denotes NFA. Then Two finite automata M and N are said to be equivalent if L(M) = L(N).
- Under such definition every DFA i.e. M =(Q, Σ, δ, q_0, F) is equivalent to an NFA i.e. N = (Q, Σ, , q_0,F) where (q, a) = ,δ(q, a)} for every state q and input a.

CONVERSIONS AND EQUIVALENCES

Conversion of ε-NFA into NFA

Procedure:

Step 1: Find the ε-closure for all states in the given ε-NFA.

ε-closure (q) denotes the set of all states p such that there is a path from q to p labeled ε.

Step 2: Find the extended transition function for all states on all input symbols for the given ε-NFA. **δ' (q,a)= ε-closure(δ (ε-closure (q),a))**

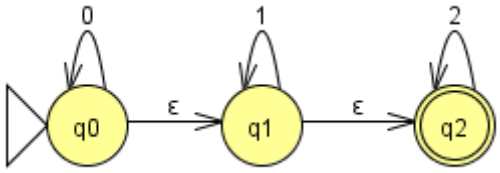
Step 3: Draw the transition table or diagram from the extended transition function (NFA)

Step 4: F is the set of final states of NFA, whose ε -closure contains the final state of ε -NFA.

Step 5: To check the equivalence of ε -NFA and NFA, the string accepted by **ε** -NFA should be accepted by NFA.

Example:

1. **Convert the following ε-NFA into NFA**



Step 1: Find the ε-closure for all states in the given ε-NFA.

ε -closure(q_0)= { q_0,q1, q_2}
ε -closure(q1)= { q1, q_2}
ε -closure(q_0)= { q_2}

Step 2: Find the extended transition function for all states on all input symbols for the given ε-NFA.

δ' (q_0,0) = ε-closure(δ(ε –closure(q_0)),0)
= ε-closure(δ {q_0,q1, q_2},0)
= ε-closure(δ(q_0, 0) U δ (q1,0) Uδ (q_2,0))
= ε-closure(q_0 U Ø U Ø)
={ q_0,q1, q_2}

δ' (q_0,1) = ε-closure(δ (ε –closure(q_0)),1))
= ε-closure(δ {q_0,q1, q_2},1)
= ε-closure(δ (q_0,1) Uδ (q1,1) Uδ (q_2,1))

= ε-closure(Ø U q1U Ø)
=$\{ q1, q_2\}$

$\delta'(q_0,2)$ = ε-closure(δ (ε –closure(q_0)),2))
= ε-closure($\delta \{ q_0, q1, q_2\}$,2)
= ε-closure($\delta (q_0,2) \cup \delta (q1,2) \cup \delta (q_2,2)$)
= ε-closure($q_2 \cup Ø$)=$\{ q_2\}$

$\delta'(q1,0)$ = ε-closure(δ (ε –closure(q1)),0))
= ε-closure($\delta \{q1, q_2\}$,0)
= ε-closure($\delta (q1,0) \cup \delta (q_2,0)$)
= ε-closure(Ø)
=$\{ Ø\}$

$\delta'(q1,1)$ = ε-closure(δ(ε –closure(q1)),,1))
= ε-closure($\delta \{q1, q_2\}$,1)
= ε-closure($\delta (q1,1) \cup \delta (q_2,1)$)
= ε-closure(q1)
=$\{ q1, q_2 \}$

$\delta (q1,2)$ = ε-closure(δ(ε –closure(q1)),2))
= ε-closure($\delta \{q1, q_2\}$,2)
= ε-closure($\delta (q1,2) \cup \delta (q_2,2)$)
= ε-closure(q_2)
=$\{ q_2\}$

$\delta (q_2,0)$ = ε-closure(δ(ε –closure(q_2)),0))
= ε-closure($\delta (q_2,2)$)
= ε-closure(Ø)
=$\{ Ø\}$

$\delta (q_2,1)$ = ε-closure(δ(ε –closure(q_2)),1))
= ε-closure($\delta (q_2,1)$)
= ε-closure(Ø)
=$\{ Ø \}$

$\delta (q_2,2)$ = ε-closure(δ(ε –closure(q_2)),2))
= ε-closure($\delta (q_2,2)$)
= ε-closure(q_2)
=$\{ q_2\}$

Step 3: Draw the transition table or diagram from the extended transition function (NFA)

State	**Inputs**

	0	1	2
q_0	$\{q_0, b, q_2\}$	$\{q1, q_2\}$	q_2
q1	Ø	$\{q1, q_2\}$	q_2
q_2	Ø	Ø	q_2

Step 4: F is the set of final states of NFA, whose ε -closure contains the final state of ε -NFA.

State	Inputs		
	0	**1**	**2**
q_0	$\{q_0, q1, q_2\}$	$\{q1, q_2\}$	q_2
q_1	Ø	$\{q1, q_2\}$	q_2
q_2	Ø	Ø	q_2

NFA to DFA Conversion:

Step 1: First take the starting state of NFA as the starting state of DFA.

Step 2: Apply the inputs on initial state and represent the corresponding states in the transition table.

Step 3: For each newly generated state, apply the inputs and represent the corresponding states in the transition table.

Step 4: Repeat step 3 until no more new states are generated.

Step 5: The states which contain any of the final states of the NFA are the final states of the equivalent DFA.

Step 6: Represent the transition diagram from the constructed table.

Step7: To check the equivalence of NFA and DFA, the string accepted by NFA should be accepted by DFA.

Step 8: Write the tuple representation for the obtained DFA.

Note: If the NFA has n states, the resulting DFA may have up to 2^n states, an exponentially larger number, which sometimes makes the construction impractical for large NFAs.

Example:

1. Construct DFA equivalent to the NFA M=({q_0,q1},{0,1}, δ,q_0,{q1})
where δ(q_0,0) = {q_0,b} δ(q_0,1) = {q1} δ(q1,0) = Ø δ(q1,1) = {q_0,q1}

Step 1: First take the starting state of NFA as the starting state of DFA

Q/∑	**0**	**1**
[q_0]		

Step 2: Apply the inputs on initial state and represent the corresponding states in the transition table.

Q/∑	**0**	**1**
[q_0]	[q_0,q1]	[q1]

Step 3: For each newly generated state, apply the inputs and represent the corresponding states in the transition table.

Q/∑	**0**	**1**
[q_0]	[q_0,q1]	[q1]
[q_0,q1]	[q_0,q1]	[q_0,q1]
[q1]	Ø	[q_0,q1]

Step 4: Stop the procedure as there are no more new states being generated.

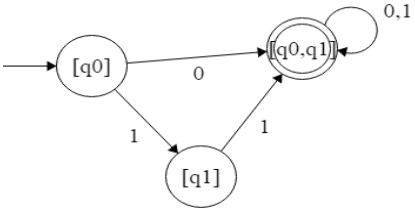
Step 5: The states which contain any of the final states of the NFA are the final states of the equivalent DFA.

q1 is the final state in NFA. q1 is included in the state [q_0,q1] and [b]. So [q_0,q1] and [q1] are the final states of the DFA.

Q/∑	**0**	**1**
[q_0]	[q_0,q1]	[q1]

[q_0,q_1]	[q_0,q1]	[q_0,q1]
[q_1]	Ø	[q_0,q1]

Step 6: Represent the transition diagram from the constructed table.



Step 7: To check the equivalence of NFA and DFA, the string accepted by NFA should be accepted by DFA.

Let **w=1110**be the string accepted by NFA.

Acceptability by NFA:

					1	q1	0	Ø
			1	q1			0	q0
q0	1	q1			1	q0		
			1	q0			0	**q1**
					1	q1		
							0	Ø

Acceptability by DFA:

	1	1	1	0
δ([q_0],1110) = δ([q1],110)	[q0]	[q1]	[q0,q1]	[q0,q1]

[q0,q1]

= δ([q_0,q1],10)

= δ([q_0,q1],0)

= [q_0,q1] ϵ F

Step 8: Write the tuple representation from the obtained DFA.

DFA M' = (Q,∑, δ,q_0,F)

Where Q = {[q_0], [q_0,q1], [bq1] }

∑ = {0, 1}

δ- transition function

[q_0] --- initial state

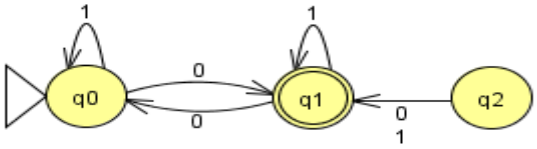
F = {[q_1], [q_0,q1]}

MINIMIZATION OF DFA's:

- **Dead State.** Al those non final states which transit to itself for al input symbols in ∑, are called Dead state.
  - Let ∑={a,b} the q is said to be dead state. If q is not the member of F and

    δ(q,a)=q
    δ(q,b)=q

- **Inaccessible states:** All those states which can never be reached from initial state are called Inaccessible states.



In the above diagram state q2 is the inaccessible state

- **Indistinguishable States:** Two States q_0 and b of DFA are called indistinguishable if

  $\delta(q_0,w) \in F \Rightarrow \delta(q1,w) \in F$,
  $\delta(q_0,w) \notin F \Rightarrow \delta(q1,w) \notin F$ for all $w \in \sum^*$

- **Distinguishable States:** Two States q_0 and b of DFA are called distinguishable if $\delta(q_0,w) \in F \Rightarrow \delta(q1,w) \notin F$ or vice versa for all $w \in \sum^*$

Procedure for Minimization of DFA:

Step 1: Remove the unreachable states from the given DFA

Step 2: (Construction of π_0)· By definition of 0-equivalence, π_0 ={B^0, $Q_2{}^0$} where $B^0$is the set of all final states and $Q_2{}^0$=Q - B^0.

Step 3: (Construction of π_{k+1} from π_k).

- Let $Q_i{}^k$be any subset in π_k . If bandq2are in $Q_i{}^k$, they are (k + 1)-equivalent provided δ (q1,a) and δ(q2,a) are k-equivalent.
- Find out whether δ (q1, a) and δ (q2, a) are in the same equivalence class in $\pi_k$for every a ∈ ∑. If so band q2 are (k + 1)-equivalent.

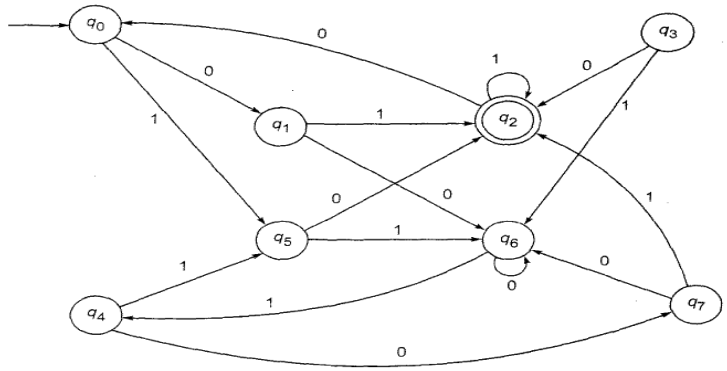
- In this way, Q_i^k is further divided into (k + 1)-equivalence classes. Repeat this for every Q_i^k in π_k to get all the elements of π_{k+1}.

Step 4: Construct π_n for n = 1, 2, ... until $\pi_n = \pi_{n+1}$.

Step 5: (Construction of minimum automaton). For the required minimum state automaton, the states are the equivalence classes obtained in step 3. i.e. the elements of π_n The state table is obtained by replacing a state q by the corresponding equivalence class [q].

Example:

Construct a minimum state automaton equivalent to the finite automaton.



Solution:

It will be easier if we construct the transition table.

State/Σ	0	1
→q_0	q_1	q_5
q_1	q_6	q_2
(q_2)	q_0	q_2
q_3	q_2	q_6
q_4	q_7	q_5
q_5	q_2	q_6
q_6	q_6	q_4
q_7	q_6	q_2

Step 1: Construction of π_0

$\pi_0 = \{B^0, Q_2^0\}$

Where $B^0 = F = \{q2\}$ $Q_2^0 = Q - B^0$

$\pi_0 = \{\{q2\}, \{q0,q1,q3,q4,q5,q6,q7\}\}$

Step 2 : The {q2} in π_0 cannot be further partitioned. So, $B^1 = \{q2\}$. Compare q_0 with $q1$, q_3,q_4,q_5,q_6 and q_7.

Consider qoand bϵ Q_2^0.

- The entries under the 0- column corresponding to qoand bare band q6; they lie in Q_2^0.
- The entries under the 1-column are $q_5$and q_2. q2 ϵ $B^0$and q5 ϵ Q_2^0. Therefore qoand q1are not 1- equivalent.

Q/∑	0	1
q_0	q1	q_5
q1	q_6	q_2

Consider q0 and q3

Q/∑	0	1
q_0	q1	q_5
q_3	q_2	q_6

The entries under the 0- column corresponding to qoand q3 are q1and q2; q1ϵ $Q_2^0$and q2 ϵ B^0. The entries under the 1-column are q5 and q6; they lie in Q_2^0. Therefore qoand q3 are not 1- equivalent

Similarly, qois not 1-equivalent to q5 and q7.

Consider q0 and q4

Q/∑	0	1
q0	q1	q5
q4	q7	q5

- The entries under the 0- column corresponding to qoand q4 are band q7; they lie in Q_2^0.

- The entries under the 1-column are q5 and q5; they lie in Q_2^0. Therefore qoand bare 1- equivalent.

  Similarly, qois 1-equivalent to q6.

  {qo. q4, q6} is a subset in π_1.
  So, Q_2^1 = {q0,q4,q6}

- Repeat the construction by considering band anyone of the state's q3, q5, q7. Now, q1is not 1-equivalent to q3 or q5 but 1-equivalent to q7.
  Hence, Q_3^1 = {q1,q7}.
- The elements left over in $Q_2^0$are q3 and q5. By considering the entries under the 0-column and the 1-column, we see that q3 and q5 are 1-equivalent.
  So Q_4^1 = {q3, q5}.
  Therefore, π_1 = {{q2}. {qo, q4, q6}. {q1, q7}, {q3, q5}}

Step 3: Construct π_n for n = 1, 2, ... until $\pi_n = \pi_{n+1}$.
Calculate 2-equivalent, π_2.

π_2= {{q2}, {qo,q4}, {q6}, {q1,q7}, {q3,q5}}

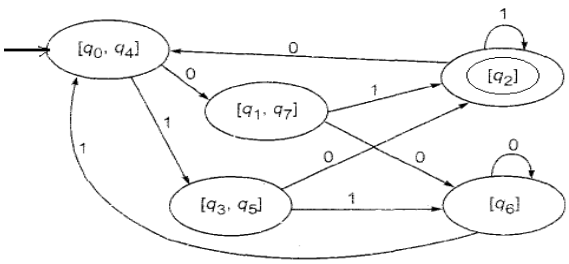
Similarly calculate 3-equivalent, π_3.

π_3 = {{q2}, {qo,q4}, {q6}, {q1,q7},{q3,q5}}

As π_2= π_3, π_2 gives us the equivalence classes.

Step 4: Construction of minimum automaton.

M' = (Q',{0,1},δ',q_0',F')
Where
Q' = {[q_2]. [q_0, q_4], [q_6], [q1, q_7], [q_3, q_5]}
q_0' = [q0, q4]
F' = [q2]
and δ' is given by



State/Σ	0	1
[q_0, q_4]	[q_1, q_7]	[q_3, q_5]
[q_1, q_7]	[q_6]	[q_2]
[q_2]	[q_0, q_4]	[q_2]
[q_3, q_5]	[q_2]	[q_6]
[q_6]	[q_6]	[q_0, q_4]

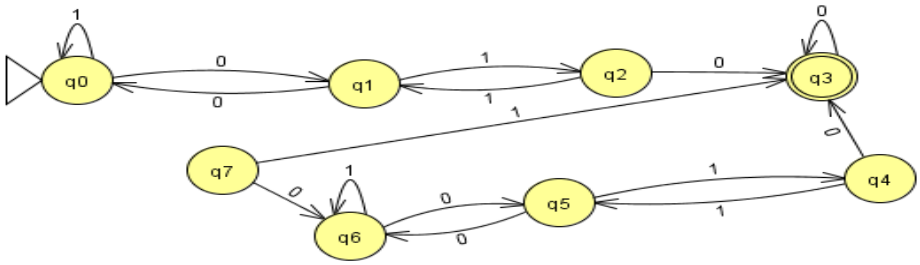
Minimization of DFA using Myhill-Nerode Theorem:

Algorithm for finding distinguishable states:

- For each pair [p,q] where p∈ F & q ∈ {Q-F} mark (p,q)=X
- For each pair of distinct states [p,q] in FXF or (Q-F) X ((Q-F) do.
  - If for some input symbol a δ([p,q],a)=[r,s]
  - If [r,s] = X then mark [p,q] = X
    Recursively mark all unmarked pairs which lead to [p,q] on input ∀a ∈
    else
    ∀ input symbols a do.
  - Put [p,q] on the list for δ([p,q],a) unless δ([p,q],a)=[r,r]
- For each pair [p,q] which is unmarked are the states which are equivalent

Example:

Find the minimum-state automaton equivalent to the transition diagram given below



Solution: The distinguishable states are marked with symbol X. The relation of all states are represented as a matrix of size n Xn. Since if p is distinguishable to q it implies that q is distinguishable to p. Therefore it is sufficient to have a lower matrix to represent the relation of one state with al other states.

Step 1: First mark for all states [p,q] where p is final state and q is non final state

[q3,q0]=X [q3,q1]= X [q3,q2] = X [q3,q4] = X [q3,q5]=X [q3,q6]=X [q3,q7]=X

q1							
q2							
q3	X	X	X				
q4				X			
q5				X			
q6				X			
q7				X			
	q0	q1	q2	q3	q4	q5	q6

Step 2: Find the states are that are distinguishable with q0

δ([q0,q1],0)=[q1,q0] δ([q0,q1],1)=[q0,q2]

δ([q0,q2],0)=[q1,q3] δ([q0,q2],1)=[q0,q1]⇒ mark[q0,q2]= X as [q1,q3] = X

⇒mark [q0,q1]=X as [q0,q2] = X

δ([q0,q4],0)=[q1,q3] δ([q0,q4],1)=[q0,q5]⇒ mark[q0,q4]= X as [q1,q3] = X
δ([q0,q5],0)=[q1,q6] δ([q0,q5],1)=[q0,q1]⇒ mark[q0,q5]= X as [q1,q3]= X
δ([q0,q6],0)=[q1,q5] δ([q0,q6],1)=[q0,q6]
δ([q0,q7],0)=[q1,q6] δ([q0,q7],1)=[q0,q3]⇒ mark[q0,q7]= X as [q0,q3] = X

	q0	q1	q2	q3	q4	q5	q6
q1	X						
q2	X						
q3	X	X	X				
q4	X			X			
q5	X			X			
q6				X			
q7	X			X			

Find the states are that are distinguishable with q1

δ([q1,q2],0)=[q0,q3] δ([q1,q2],1)=[q2,q1]⇒ mark[q1,q2]= X as [q0,q3] = X
δ([q1,q4],0)=[q0,q3] δ([q1,q4],1)=[q2,q5]⇒ mark[q1,q4]= X as [q0,q3] = X
δ([q1,q5],0)=[q0,q5] δ([q1,q5],1)=[q2,q4]
δ([q1,q6],0)=[q0,q5] δ([q1,q6],1)=[q2,q6]⇒ mark[q1,q6]= X as [q0,q6] = X
δ([q1,q7],0)=[q0,q6] δ([q1,q7],1)=[q2,q3]⇒ mark[q1,q7]= X as [q2,q3] = X

	q0	q1	q2	q3	q4	q5	q6
q1	X						
q2	X	X					
q3	X	X	X				
q4	X	X		X			
q5	X			X			
q6		X		X			
q7	X	X		X			

Find the states are that are distinguishable with q2

δ([q2,q4],0)=[q3,q3] δ([q2,q4],1)=[q1,q5]
δ([q2,q5],0)=[q0,q5] δ([q2,q5],1)=[q1,q4]⇒ mark[q2,q5]= X as [q4,q6] = X
δ([q2,q6],0)=[q0,q6] δ([q2,q6],1)=[q2,q6]⇒ mark[q2,q6]= X as [q4,q5] = X
δ([q2,q7],0)=[q0,q7] δ([q2,q7],1)=[q2,q3]⇒ mark[q2,q7]= X as [q4,q6] = X

	q0	q1	q2	q3	q4	q5	q6
q1	X						
q2	X	X					
q3	X	X	X				
q4	X	X		X			
q5	X		X	X			
q6		X	X	X			
q7	X	X	X	X			

Find the states are that are distinguishable with q4

δ([q4,q5],0)=[q3,q6] δ([q4,q5],1)=[q5,q4]⇒ mark[q4,q5]= X as [q3,q6] = X
δ([q4,q6],0)=[q3,q5] δ([q4,q6],1)=[q5,q6]⇒ mark[q4,q6]= X as [q3,q5] = X
δ([q4,q7],0)=[q3,q6] δ([q4,q7],1)=[q5,q3]⇒ mark[q4,q7]= X as [q3,q6] = X

q1	X						
q2	X	X					
q3	X	X	X				
q4	X	X		X			
q5	X		X	X	X		
q6		X	X	X	X		
q7	X	X	X	X	X		
	q0	q1	q2	q3	q4	q5	q6

Find the states are that are distinguishable with q6
δ([q5,q6],0)=[q6,q5] δ([q5,q6],1)=[q4,q6]⇒ mark[q5,q6]= X as [q4,q6] = X
δ([q5,q7],0)=[q6,q6] δ([q5,q7],1)=[q4,q4]⇒ mark[q5,q7]= X as [q4,q3] = X

q1	X						
q2	X	X					
q3	X	X	X				
q4	X	X		X			
q5	X		X	X	X		
q6		X	X	X	X	X	
q7	X	X	X	X	X	X	
	q0	q1	q2	q3	q4	q5	q6

Find the states are that are distinguishable with q7
δ([q6,q7],0)=[q5,q6] δ([q6,q7],1)=[q6,q3]⇒ mark[q6,q7]= X as [q5,q4] = X

q1	X						
q2	X	X					
q3	X	X	X				
q4	X	X		X			
q5	X		X	X	X		
q6		X	X	X	X	X	
q7	X	X	X	X	X	X	X
	q0	q1	q2	q3	q4	q5	q6

From the table it is clear that state [q0,q6] ,[q1,q5] and [q2,q4] belongs to same class. These states can be merged and the minimized DFA is

Q/∑	0	1
→[q0,q6]	[b,q5]	[q0,q6]
[q1,q5]	[q2,q4]	[q0,q6]
[q2,q4]	[q3]	[q1,q5]
[q3]	[q3]	[q0,q6]
[q7]	[q0,q6]	[q3]

Equivalence of Two Finite Automata:

- Two finite automata over ∑ are equivalent if they accept the same set of strings over ∑
- When the two finite automata are not equivalent there is some string w over ∑ satisfying the following
    - One automaton reaches a final state on application of 'w' whereas the other automaton reaches a non-final state.
- Comparison method is used to test the equivalence of two finite automata over ∑

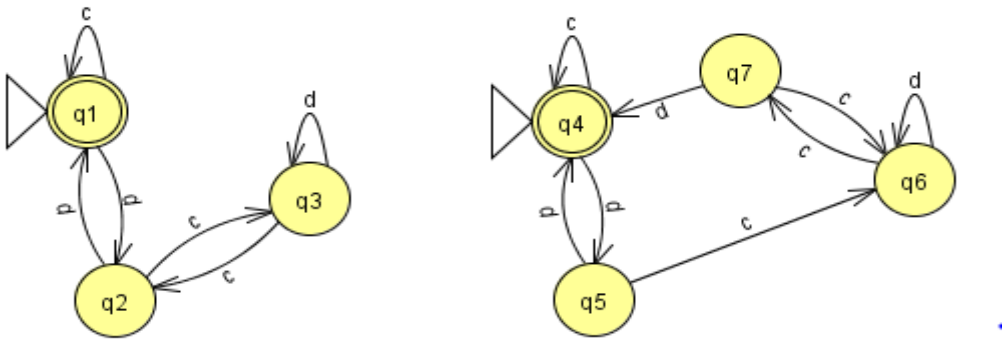
Comparison Method:

1. Let M and M' are two finite automata over ∑ we construct comparison table consisting of n+1 columns where n is the number of input symbols
2. The first column consists of pair of vertices of the form (q,q') where q ∈ M and q' M'. if(q,q') appear in some row of the first column then corresponding entry in the a-column(a ∈∑) is (q_a,q'_a), where q_a and $q'_a$are reachable from q,q' respectively on application of a.
3. The comparison table is constructed by starting with the pair of initial vertices q_{in}, q'_{in} of M,M' in the first column.
4. The first elements in the subsequent columns are (q_a,q'_a), where q_a and $q'_a$are reachable from q_{in}, q'_{in}respectively.
5. We repeat the construction by considering the pairs in the second and subsequent columns which are not in the first column
   The row wise construction is repeated.
   There are two cases:
   Case 1: if we reach a pair (q,q') such that q is a final state of M, and q' is a non final state of M' or vice versa, we terminate the construction and conclude that M and M' are not equivalent.
   Case 2: Here the construction is terminated when no new element appears in the second and subsequent columns which are not in the first column.

In this case we conclude that M and M' equivalent.

Example:

Test whether the following DFA's are equal or not



Comparison Method:

(q,q')	(q_c,q'_c)	(q_d,q'_d)
(q_1,q_4)	(q_1,q_4)	(q_2,q_5)
(q_2,q_5)	(q_3,q_6)	(q_1,q_4)
(q_3,q_6)	(q_2,q_7)	(q_3,q_6)
(q_2,q_7)	(q_3,q_6)	(q_1,q_4)

The initial state in M and M' are q_1 and q_4 respectively. Hence the first element of the first column in the comparison table must be (q_1,q_4). The first q_1 andq_4 are c-reachable from the respective initial states.

As we do not get a pair (q,q') where q is a final state and q' is a non final state at every row, we proceed until all the elements in the second and third columns are also in the first column

Therefore M and M' are equivalent

Finite Automata with output:

Moore Machine:

It is the finite automata in which output is associated with each state. In moore machine every state of this finite automata has a fix output. Mathematically moore machine is a six-tuple machine and defined as
$M=(Q,\sum, \Delta,\delta,q0,\lambda)$ where
Q= non empty set of states
$\sum$=non empty set of input smbols
Δ=non empty finite set of outputs

δ=it is transition function which takes two arguments as in finite automata, one is input state and another input symbol. The output of this function is a single state

q0= is the initial state

λ=is output function which maps Q to Δ

Representation of Moore Machine:

Moore machine can be represented by transition diagram as well as transition table same as finite automata.

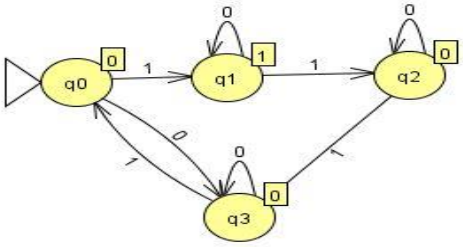


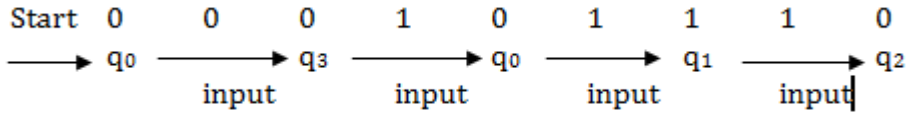
Fig: Transition diagram representation of Moore Machine

Present State	Next State at input		output
	a=0	a=1	
q0	q3	q1	0
q1	q1	q2	1
q2	q2	q3	0
q3	q3	q0	0

Fig: Transition table representation of Moore Machine

String Processing through Moore Machine:

Let us process the string w=0111 by the Moore machine.



The output string will be w'= λ(q0)λ(q3)λ(q0)λ(q1)λ(q2)

=00010

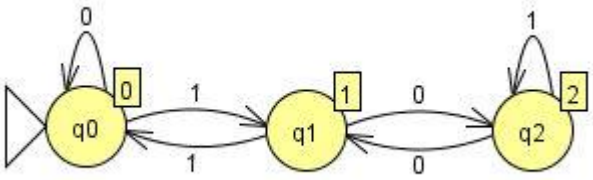
|w|=4

|w'|=5

- The length of input string is n then the length of output string will be n+1

Example:

Design a Moore machine which determines the residue mod-3 for each binary string treated as a binary integer



Mealy Machine:

It is the finite automata in which output is associated with each transition. In Mealy machine every transition for a particular input symbol has a fix output.

Mathematically Mealy machine is a six-tuple machine and defined as

M=(Q,∑, Δ,δ,q0,λ) where

Q= non empty set of states

∑=non empty set of input smbols

Δ=non empty finite set of outputs

δ=it is transition function which takes two arguments as in finite automata, one is input state and another input symbol. The output of this function is a single state

q0= is the initial state

λ=is output function which maps Q X ∑→Δ

Representation of Mealy Machine:

Mealy machine can be represented by transition diagram as well as transition table same as finite automata.

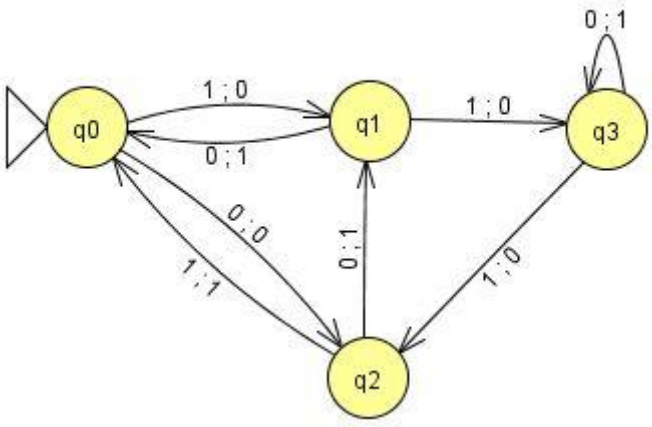


Fig: Transition diagram representation of Mealy Machine

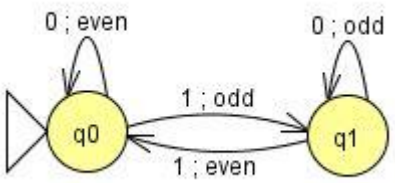
Present State	For input a=0		For input a=1	
	state	output	state	output
q0	q2	0	q1	0

q1	q0	1	q3	0
q2	q1	1	q0	1
q3	q3	1	q2	0

Fig: Transition table representation of Mealy Machine

Example:

Construct a mealy machine which can output even, odd according as the total number of 1's encountered is even or odd. The input symbols are 0 and 1



Conversion from Moore machine to 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 Qi output is m, copy it into the output columns of the Mealy Machine state table wherever Qi appears in the next state

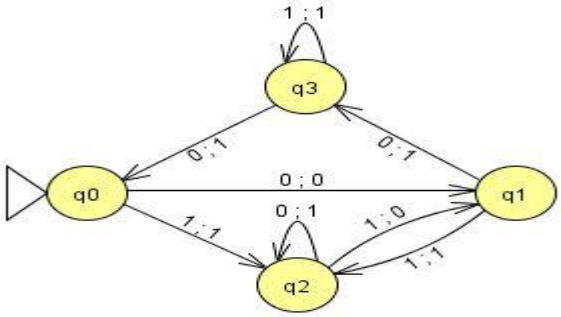
Example:
Construct a Mealy machine which is equivalent to the Moore machine defined in the following Table.

Present State	Next State at input		output
	a=0	a=1	
q0	q1	q2	1
q1	q3	q2	0
q2	q2	q1	1
q3	q0	q3	1

Transition table for Required Mealy Machine:

Present	For input a=0	For input a=1

State	state	output	state	output
q0	q1	0	q2	1
q1	q3	1	q2	1
q2	q2	1	q1	0
q3	q0	1	q3	1



Conversion fromMealy machine toMoore Machine:

Step 1: Determine the number of different output associated with qi in the next state column.

Step 2: we split qi into different states according to different output associated with it. for ex. suppose in the next state column of the above sample transition table of mealy machine, the output associated with q1 is "0" in the first next state column and "1" in the second next state column. so we split q1 into q10 and q11 states. similarly check others and split them**.**

Example:

Construct the Moore machine equivalent to the following Mealy Machine.

Present State	For input a=0		For input a=1	
	state	output	state	output
q1	q3	0	q2	0
q2	q1	1	q4	0
q3	q2	1	q1	1
q4	q4	1	q3	0

Step 1: Identify the States those are having different output. In given Mealy Machine q2 and a4 have two different outputs. So split q2,q4 states as q_{20},q_{21} and q_{40},q_{41}

Step 2: Write the transitions with new states

Transition table for Required Moore Machine

Present State	Next State at input		output
	a=0	a=1	
q1	q3	q20	1
q20	q1	q40	0
q21	q1	q40	1
q3	q0	q3	0
q40	q41	q3	0
q41	q41	q3	1

Applications of Finite Automata in Real World:

- Vending Machines
- Traffic Lights
- Video Games
- Text Processing
- Regular Expression Matching
- CPU controllers
- Protocol Analysis
- Natural Language Processing
- Speech Recognition

UNIT – III
Regular Languages

Regular Grammar:

A regular grammar is a formal grammar that describes the regular language. Where a formal grammar is defined as a set of rules for rewriting the strings, along with a start symbol from which the rewriting must start.

Therefore, a grammar is usually thought of as a language generator. However, it can also sometimes be used as the basis for a "recognizer"—a function in computing that determines whether a given string belongs to the language or is grammatically incorrect.

To describe such recognizers, formal language theory uses separate formalisms, known as automata theory. One of the interesting results of automata theory is that it is not possible to design a recognizer for certain formal languages. Parsing is the process of recognizing an utterance i.e. expression or word (a string in natural languages) by breaking it down to a set of symbols and analyzing each one against the grammar of the language. The regular grammars are of two types:

1) left regular grammars and
2) right regular grammars.

Right Regular Grammar:

Right regular grammar is also called as right linear grammar which is a formal grammar **(V, Σ, P, S)** such that all the productions or production rules in P are of one of the following forms:

- B →a (Where B is a variable or non-terminal in V and *a* is terminal in Σ)
- B →aC (Where B and C are in a Variable V and a is terminal in Σ)
- B →ε (Where B is a variable in V and ε denotes the empty string i.e. the string of length 0).

Left Regular Grammar:-

Left regular grammar is also called as left linear grammar which is a formal grammar **(V, Σ, P, S)** such that all the productions or production rules in P are of one of the following forms

- A → *a* d(Where A is a non-terminal in V and *a* is a terminal in Σ)
- A → B*a* (Where A and B are in V and *a* is in Σ)
- A → ε (Where A is a variable in V and ε is the empty string)

Example:

An example of a right regular grammar G with N = {S, A}, Σ = {a, b, c}, P consists of the following rules:

S → *a*S
S → bA
A → ε
A → cA

And S is the start symbol. This grammar describes the same language as the regular expression a*bc*

Extended Regular Grammars

- An extended right regular grammar is one in which all rules obey one of the following:
  - B → *a* (Where B is a non-terminal in V and *a* is a terminal in Σ)
  - A → wB (Where A and B are in V and w is in Σ*)
  - A → ε (Where A is in V and ε is the empty string)
- An extended right regular grammar is also called as strictly right regular grammar.
- An extended left regular grammar is one in which all rules obey one of the following:
  - A → *a* (Where A is a non-terminal in V and *a* is a terminal in Σ)
  - A → Bw (Where A and B are in V and w is in Σ*)
  - A → ε (Where A is in V and ε is the empty string)
- An extended left regular grammar is also called as strictly left regular grammar.

Examples:

Let us consider the grammar G=({S},{a, b},P,S), whereP=

S →abS
S → λ
S → Sab

EQUIVALENCE BETWEEN REGULAR LINEAR GRAMMAR AND FINITE AUTOMATA, INTERCONVERSION

The equivalence exists between regular grammar and finite automata in accepting languages.

CONSTRUCTION OF REGULAR GRAMMAR GENERATING T(M) FOR A GIVEN DFA M

Let $M = (\{q_0, \ldots, q_n\}, \Sigma, \delta, q_0, F)$. if w is a string in the language of machine M i.e. T(M), then it is obtained by concatenating the labels corresponding to several transitions, the first from q_0 and the last terminating at some final state.

So for the grammar G to be constructed, productions should correspond to transitions. Also, there should be provision for terminating the derivation tree once a transition terminating at some final state is encountered.
With these ideas we construct G as:
$G = (\{A_0, A_1, \ldots, A_n\}, \Sigma, P, A_0)$
P is defined by the following rules:
(1) $A_i \rightarrow aA_i$ is included in P if $\delta(q_i, a) = q_j$ F
(2) $A_i \rightarrow aA_j$ and $A_i \rightarrow a$ are included in P if $\delta(q_i, a) = q_j \in F$.

We can now show that L(G) = T(M) by using the construction of P. Such a construction gives:

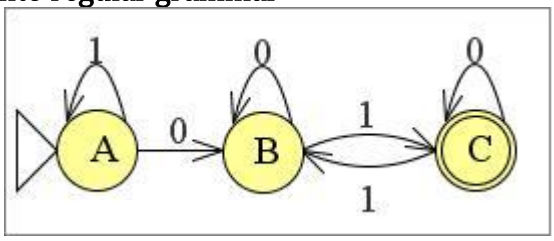
$A_i \rightarrow aA_j$ iff $\delta(qi, a) = qj$
$A_i \rightarrow a$ iff $\delta(qi, a) = F$

So $A_0 \rightarrow a_1A_1 \rightarrow a_1a_2A_2 \rightarrow \ldots \rightarrow a_1 \ldots \rightarrow a_{k-1}A_k \rightarrow a_1a_2 \ldots a_k$ iff
$\delta(q_0, a_1) = q_1, \quad \delta(q_1, a_2) = q_2, \ldots, \delta(q_k, a_k) \in F$

This proves that $w = a1 \ldots ak \in L(G)$ iff $\delta(q_0, a_1 \ldots a_k) \in F$, i.e. iff $w \in T(M)$.

Example Problems:

1. **Convert following DFA into regular grammar**

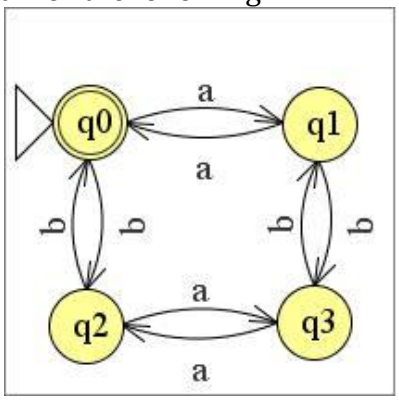


A→1A [∵δ(A,1) = A]
A→0B
B→0B|1C|1 [∵δ(B,1) = C C ∈ F]
C→0C
C→1B
C→0

Tuple Representation:

G(V,T,P,S) where V=set of Variables={A,B,C}
T = set of Terminals={0,1}
S = Start symbol= A
P = set of productions(shown in above)

2. Construct right linear grammar for the following DFA



$q_0 \rightarrow aq_1$
$q_0 \rightarrow bq_2$
$q_1 \rightarrow aq_0$
$q_1 \rightarrow a$
$q_1 \rightarrow bq_3$
$q_2 \rightarrow aq_3$
$q_2 \rightarrow bq_0$
$q_2 \rightarrow b$
$q_3 \rightarrow aq_2$
$q_3 \rightarrow bq_1$

Tuple Representation:

G(V,T,P,S) where V=set of Variables={q_0, q_1, q_2, q_3}
T = set of Terminals={a,b}
S = Start symbol= q_0
P = set of productions(shown in above)

CONSTRUCTION OF TRANSITION SYSTEM M ACCEPTING L(G) FOR A GIVEN REGULAR GRAMMAR G

- Let us consider the grammar G = ({A_0, A_1, … ,A_n}, Σ, P, A_0). We construct the transition system M whose
  (a) States corresponds to variables
  (b) Initial state corresponds to A_0
  (c) Transition in M corresponds to productions in P.
- As the last production applied in any derivation is of the form $A_i \rightarrow a$, the corresponding transition terminates at a new state and this is the unique final state.

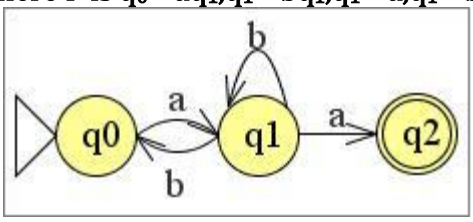
Now

We define M as ({q_0, … ,q_n}, Σ, δ, q_0, q_f). δ is defined as follows:
(1) Each production $A_i \rightarrow aA_j$ induces a transition from q_i to q_j with label *a*.
(2) Each production $A_k \rightarrow a$ induces a transition from q_k to q_f with label *a*.

- From the construction it is easy to see that $A_0 \rightarrow a_1A_1 \rightarrow a_1a_2A_2 \rightarrow \ldots \rightarrow a_1 \ldots \rightarrow a_{n-1}A_{n-1} \rightarrow a_1 \ldots a_n$ is a derivation iff there is a path in M starting from q_0 and terminating in q_f with path values $a_1a_2 \ldots a_n$. Therefore L(G) = T(M)

Examples:

1. **Convert following regular grammar into Finite Automata.**
   G=({q0,q1},{a,b},P,q0) where P is $q_0 \rightarrow aq_1, q_1 \rightarrow bq_1, q_1 \rightarrow a, q_1 \rightarrow bq_0$



Tuple Representation:
M=(Q,∑, δ,q_0,F) where
Q=set of states={ q_0,q_1, q_2}
∑=set of input symbols={a,b}
δ=transition function= QX∑→Q
q0=initial state=q0
F=Set of final states={q2}

Context Free Grammar:

Context free grammars have played a central role in compiler technology since the 1960's; they turned implementation of parsers (functions that discover the structure of a program) from a time consuming, ad-hoc implementation task into a routine job.

DEFINITION OF CONTEXT FREE GRAMMAR

- **Formally** context free grammar is a 4-tuple i.e. (*V*, *Σ*, *P*, *S*), where
  - *V* is a finite set called the **variables** also called as **nonterminals** or **syntactic categories.**
    - Each variable represents a language i.e. set of strings.
  - *Σ* is a finite set of symbols that form the strings of the language being

defined. We call this alphabet the **terminals** or **terminal symbols.**

- ***P*** is a finite set of **productions** or **rules** that represents the recursive definition of a language. Each production consists of :
  - A variable that is being (partially) defined by the production. This variable is often called the head of the production.
  - A production symbol→.
  - A string of zero or more terminals and variables. This string called the body of the production represents one way to form strings in the language of the variable of the head.
- One of the variables (***S***) represents the language being defined; it is called the **start symbol**. Other variables represent the auxiliary classes of strings that are used to help define the language of the start symbol. In our example P the only variable, is the start symbol.

- **Note:** By convention, the start variable is the variable on the left hand side of the first rule.

- **Find the language generated by the given grammar.**

  S→SS
  S→a

  Sol: V= set of variables={S}
  T=set of terminals={a}
  P=set of productions={S→SS/a}
  S= start symbol=S .
  S→aS
  S→aSS
  S→aSSS
  S→aaSS
  S→aaSSS
  S→aaaSS
  S→aaaaS
  S→aaaaa

  Language generated by G is L(G)= { a^i / i>=2}.
  In the given productions we have production in the form of S→a . So we have single a ϵ L(G).
  hence the language generated by given language G id L(G)={a^i / i>=1}.

- **Find the language generated by the given grammar.**

  S→SS
  S→aa
  S→ ϵ

  Sol: V= set of variables={S}
  T=set of terminals={a}
  P=set of productions={S→SS/aa/ ϵ }

S= start symbol=S .
S→SS
S→SSS
S→aaSS
S→aaSSS
S→aaaaSS
.
.
.
.
S→$(aa)^n$

∴ L(G)={$(aa)^n$/n>=1} or {a^{2n} / n>=1}

But we have production S→ ϵ. Such that **ε** ϵ L(G).So the language generated by given grammar G is L(G)= { a^{2n}/n>=0} or {$(aa)^n$ />=0}.

- **Generate grammar for the language L= {$b^n a^n$/n>=0}**

  Strings generated by the given language
  L={ ϵ, ba, bbaa, bbbaaa……}
  We have ϵ in the string. So we have production as S→ ϵ
  Then by observing set of strings generated by language. Equal number of b's followed by equal number of a's.
  So we have productions like S→bS
  S→ba
  The required grammar generated by the given language is S→bSa|ba| ϵ.

  Tuple representation:

  G(V,T,P,S)
  V={S}
  T={b,a}
  P={ S→bSa|ba| ϵ}
  S→S.

- **Find the grammar for the language L= {$a^n b^{2n}$/n>=1}**

  L= {$a^n b^{2n}$/n>=1}
  Strings generated by the given language
  L={ ϵ,abb,aabbbb,aaabbbbbb……}
  When n=1, abb
  N=2, aabbbb.
  Language generates the strings in the form above so that produces grammar like following productions.
  S→abb

S→aSbb.

Tuple representation:

G(V,T,P,S)
V={S}
T={b,a}
P={ S→abb|aSbb}
S→S.

- **Find the grammar for the language L= {$a^n c^m d^m b^n$/n,m >=1}**

Strings generated by the given language
L={ acdb, accddb, aacdbb, aaaccddbbb……}

So The required grammar have fillowing productions.

S→aSb
S→aAb
S→cAd
S→cd

Tuple representation:

G(V,T,P,S)
V={S}
T={b,a}
P={ S→aSb|aAb|,A→cAd|cd}
S→S.

Left Most Derivation: let G is a context free grammar and w ϵ L(G). Then the derivation S ≐> w is called left most derivation. If and only if all the steps involved in the derivation have the replacement of Left Most Derivation.

Example:

Consider the grammar S→S+S, S→S*S, S→a, S→b.
Derive the string w=a*a+b.
S→S*S
S→a*S
S→a*S+S
S→a*a+S
S→a*a+b

Right Most Derivation: let G is a context free grammar and w ϵ L(G). Then the derivation S ≐> w is called right most derivation. If and only if all the steps involved in the derivation have the replacement of Right Most Derivation.

Example:

Consider the grammar S→S+S, S→S*S, S→a, S→b.
Derive the string w=a*a+b.
S→S+S

S→S+b
S→S*S+b
S→S*a+b
S→a*a+b

Sentential Form:

Sentential Form and Partial Derivation Tree

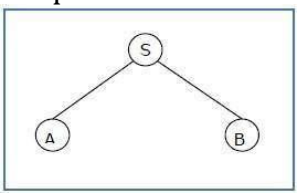
A partial derivation tree is a sub-tree of a derivation tree/parse tree such that either all of its children are in the sub-tree or none of them are in the sub-tree.

Example

If in any CFG the productions are –

S → AB, A → aaA | ε, B → Bb| ε

the partial derivation tree can be the following –



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

Regular Expression:

Regular expressions are useful for representing certain set of strings in an algebraic fashion. Regular expression describes the language accepted by the finite state machine.

- Any terminal symbol or element of Σ is a regular expression.
  Eg:- Ø, ε,a in Σ
  - Ø is a regular expression and denotes the empty set.
  - ε is a regular expression and denotes the set { ε}
  - a is a regular expression which denotes the set {a}
- Union of two regular expressions R1, R2 is a regular expression R1+R2
  Eg:- Let a be a regular expression R1, b be a regular expression R2
  (a+b) is also a regular expression
- Concatenation of two regular expressions R1,R2 is a regular expression R1.R2
  Eg:- Let a and b are two regular expressions then ab is also a regular expression
- Closure of a regular expression R written as R*. It is also a Regular expression.

Regular Set:

Any set represented by a regular expression is called a regular set.

- If a and b are the elements of Σ then regular expression a denotes the set {a}
- a+b denotes the set {a,b}
- ab denotes the set {ab}
- a* denotes the set { ε,a,aa,aaa,aaaa,aaaaaa,……..}
- (a+b)* denotes the set { ε,a,b,aa,bb,ab,aab,abb,aabb,aaab,abbb,…….}

1. **Write the regular expression for all strings of 0's and 1's.**
   The set represented by given language L={ε,0,1,00,11,01,10,010,100,……}
   The required regular expression is **(0+1)*.**
2. **Write the regular expression for all strings of 0's and 1's end in 00.**

The set represented by given language L={ 00,000,100,0000,1100,0100,1000,....}
The required regular expression is **(0+1)*00.**

3. **Write the regular expression for set of all strings 0's and 1's begins with 0 and ends with 1**
   The set represented by given language L={01,001,011,0011,00111,01111,……..}
   The required regular expression is **0(0+1)*1.**
4. **Write the regular expression for set of all strings having even number of 1's Σ={1}**
   The set represented by given language L={ ε,11,1111,111111,11111111,…….}
   The required regular expression is **(11)*.**
5. **Write the regular expression for set of all strings having odd number of 1's Σ={1}**
   The set represented by given language L={ 1,111,11111,1111111,…….}
   The required regular expression is **1(11)* or (11)*1.**
6. **Write the regular expression for set of all strings of 0's and 1's with at least two consecutive 0's**
   The set represented by given language L={ 00,000,100,001,0000,0011,…….}
   The required regular expression is **(0+1)*00(0+1)*.**

7. **Write a regular expression for all strings 0's and 1's beginning with 1 or 0 and not having two consecutive 0's.**
   The set represented by given language L={0,1,10,01,11,101,110,....}
   The required regular expression is **(0+ ε)(1+10)*.**
8. **Write a regular expression for set of all strings with at least one 0,one 1 and one 2 respectively .**
   The set represented by given language L={012,001122,000111222,…….}
   The required regular expression is $0^+1^+2^+$ or 00*11*22* .
9. **Write a regular expression for set of all strings of 0's and 1's whose last two symbols are the same.**
   The set represented by given language L={00,11,000,111,100,01011,0100,....}
   The required regular expression is **(0+1)*(00+11).**
10. **Write a regular expression for set of all strings in which every 0 is immediately followed by at least two 1's**
    The set represented by given language L={ **ε,011,1,011011,111** ,....}
    The required regular expression is **(1+011)*.**

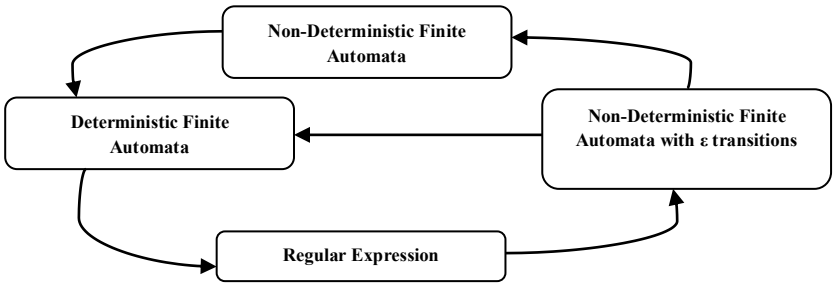
Identity Rules for Regular Expression:

a. Ø+R= R+Ø=R
b. εR=Rε=R
c. R+R=R
d. $RR^*=R^*R=R^+$
e. $\varepsilon+RR^*=R^*$
f. ØR=RØ=Ø
g. $\varepsilon^*=\varepsilon$, $Ø^*=\varepsilon$

h. (R*)*=R*
i. (PQ)*P=P(QP)*
j. (P+Q)R=PR+QR
k. R(P+Q)=RP+RQ
l. (P+Q)*=(P*Q*)*=(P*+Q*)*
m. (ε+R)*=R*
n. P+Q=Q+P

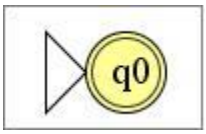
Example:
prove that ε +a*(b)*(a*(b)*)*=(a + b)*
solution:
Let P= a*(b)*
the given expression is represented as ε + PP*=P*
Gence P*= a*(b)*
Let P=a and Q=b . then the expression can be represented as (P*Q*)*
=(P+Q)* using the identity rule (P+Q)*=(P*Q*)*=(P*+Q*)*
=(a + b)*
Hence it is proved that both are same.

Equivalence of Finite Automata with Regular Expression

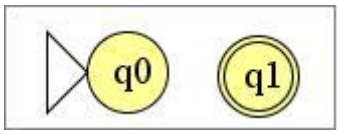


Every regular expression there exist a deterministic finite automata. So we can say that regular languages, regular expression and finite automata are all different representation of the same thing. Therefore we convert Finite Automata to Regular Expression, Regular Expression to Finite Automata.
Let r be a regular expression. Then there exists an NFA with ε -transitions that accept L(r).

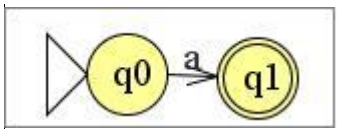
The expression r must be ε, Ø, or a for some a in Σ. The NFA's are



a)r=ε



b) r= Ø



c)r=a

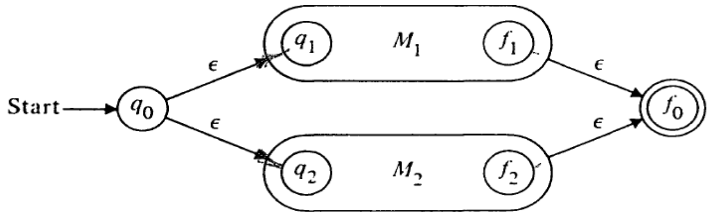
Construction of regular expression into Ɛ-NFA

- Ɛ-transitions provide a convenient way of modeling the system whose states sre not precisely known
- Ɛ-transitions do not add any extra capacity of recognizing formal languages
- Ɛ-NFA and NFA recognize same class of languages namely regular languages.
- NFA with Ɛ-transitions are defined because certain properties can be move easily proved on them as compared to NFA.

Let r have i operators. There are three cases depending on the form of r.

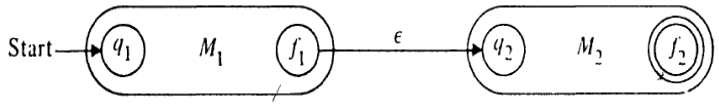
Case 1: Union (r = rl + r2.)

L(M) = L(M1) ∪ L(M2)

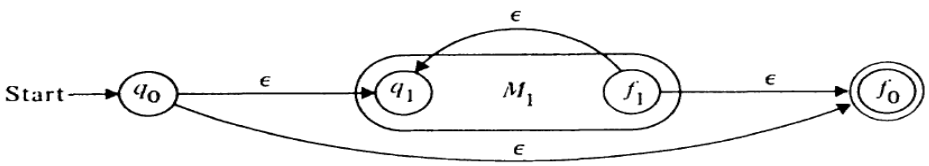


Case 2: Concatenation (r = r1 r2).

L(M) = L(M1)L(M2)



Case 3: Closure (r = r1*)

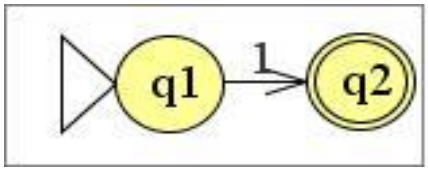


Examples:

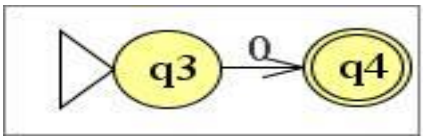
1. Construct Ɛ-NFA for the regular expression 1+01*

Regular expression is of the form r1 + r2, where r1 =1 and r2 = 01*.

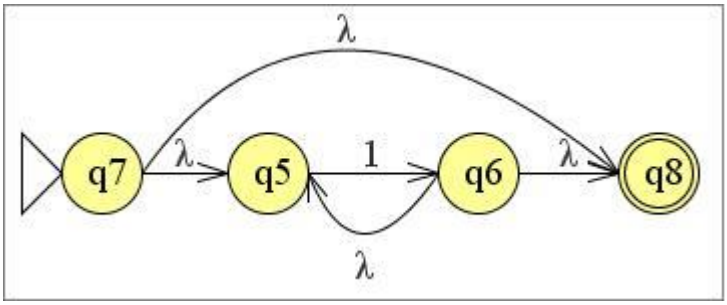
For 1 the regular expression is



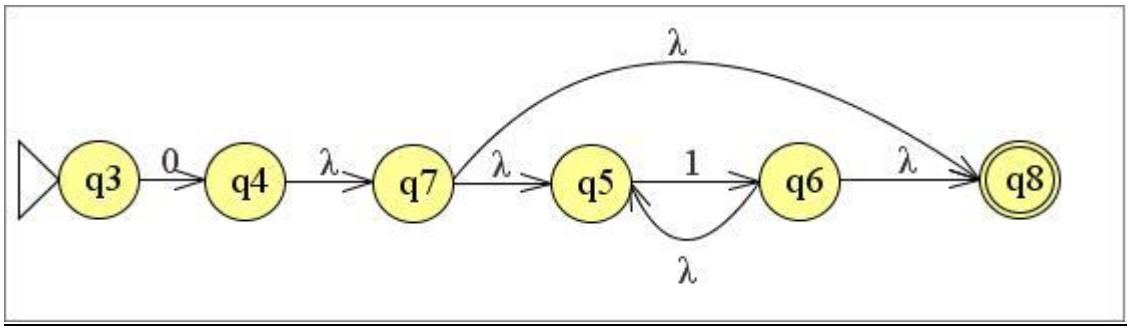
For 0 the regular expression is



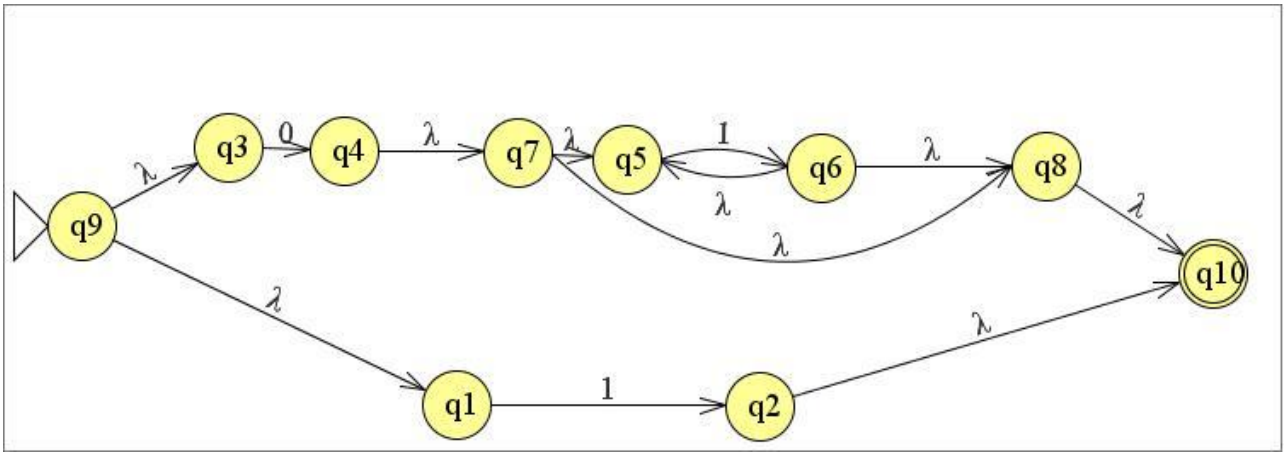
For 1* the regular expression is



For 01* the regular expression is



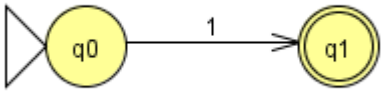
Finally, use the construction of union to find the NFA for r = r1 + r2



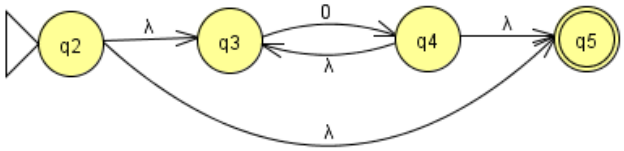
2. Consider regular expression 10*+01* and construct equivalent NFA.

The given regular expression is 10*+01* and Ɛ={0,1}

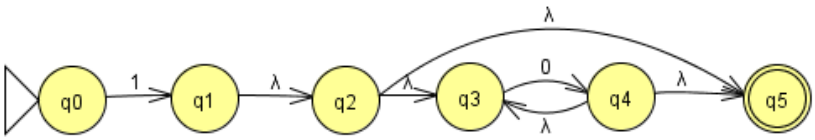
The following is a NFA accepting 1



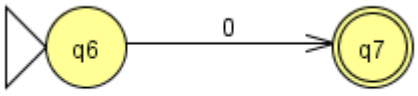
The following is a NFA accepting 0*



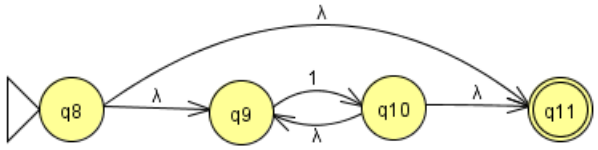
The following is a NFA accepting 10*



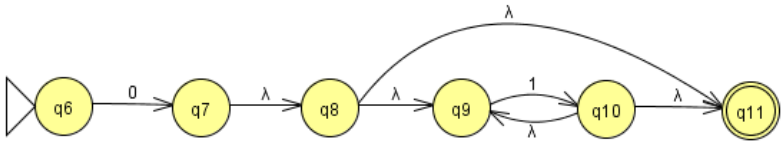
The following is a NFA accepting 0



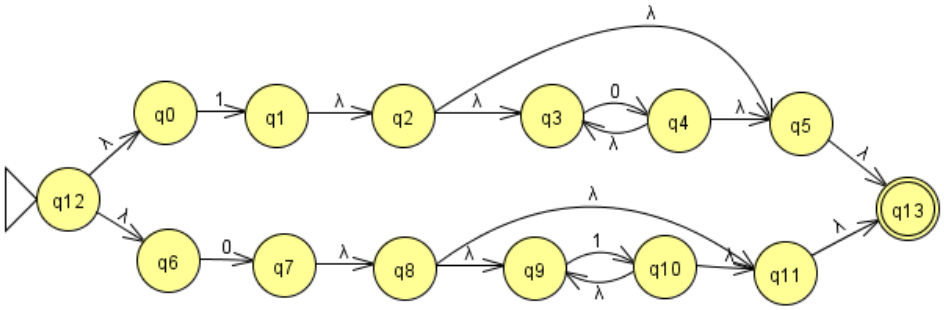
The following is a NFA accepting 1*



The following is a NFA accepting 01*



The following is a NFA accepting 10*+01*



Construction of Regular Expression from Finite Automata:

To convert Finite Automata into regular expression we need to know about Arden's Theorem.

Arden's Theorem:

Let P and Q be two regular expressions over alphabet Σ. If P does not contain **ε(null string)** then **R=Q+RP** has unique solution that is **R=QP***

Proof: Put the value of R in the R.H.S

$R=Q+(Q+RP)P=Q+QP+QP^2$

When we put the value of R again and again we get the following equation.

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

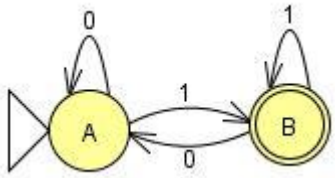
$R=Q+(Q+RP)P=Q(1+P+P^2+P^{3+}\ldots\ldots\ldots\ldots$

The second part of the product on the L.H.S can be replaced with the Kleene Closure. So the equation becomes R=QP*

Steps involved to Construct Regular Expression From DFA

Step 1: To get the regular expression from the automata we first create the equations for each state in presenting in the finite automata transition diagram.
Note: Consider the incoming edges only to a state in transition diagram to construct the equation of each state.
The state equation in the form of
$q_1 = q_1w_{11} + q_2w_{21} + \ldots\ldots + \varepsilon$ (q1 is the initial state so we have to add **ε** to the equation)
$q_2 = q_1W_{12} + q_2W_{22} + \ldots\ldots\ldots + q_nW_{n2}$
$q_n = q_1W_{1n} + q_2W_{2n} + \ldots\ldots q_nW_{nn}$
Where, w_{11} is the regular expression representing the set of labels of edges from q_i to q_j.
Note: For parallel edges there will be that many expressions for that state in the expression.
Step 2: Then we solve these equations to get the equation for q_i in terms of w_{ij} and that expression is the required solution, where qi is a final state.
Step 3: Repeatedly applying substitutions method and Arden's theorem over the state equations and we can express qi in terms of W_{ij}'s.
Step 4. For getting the set of string recognized by the transition system, we have to take the 'union' of all Ws corresponding to final states.

1. **Construct the regular expression for the following DFA.**



A=A0+B0+ ε ----------1
B=B1+A1 ----------2
B=B1+A1 From Arden's Theorem R=Q+RP then R=QP*
B=A11* ----------3
Substitute 3 in 1
A=A0+A11*0+ ε
A=A(0+11*0)+ ε
A=(0+11*0)*----------4 From Arden's Theorem
Substitute 4 in 3
B=(0+11*0)*11*

Required Regular expression is
(0+11*0)*11*

Regular Language:

Regular language is the set of all languages that can be represented by a regular expression.

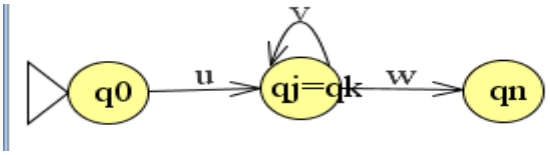
Pumping Lemma for Regular Languages:

Theorem: For any regular language L there exists an integer, such that for all x Є L with |x|>=n there exist u,v,w Є **Σ*** such that

i) x=uvw
ii) |uv|<=n
iii) |v|>=1
iv) for all k>=0 : $uv^kw \in L$

Proof:-

Let M be a DFA for L taken n be the number of states of M. Take any x Є L with |x|>=n consider the path in M that corresponds to x. The length of this path is |x|>=n.
Since M has at most n-1 states some state must be visited twice or more in the first n steps of the path.



Application of Pumping Lemma:

This theorem can be used to prove that certain sets are not regular
Step 1: Assume that L is regular. Let n be the number of states in corresponding finite automaton.
Step 2: Choose a string w such that |w| >=n. Use pumping lemma to write w=xyz, with |xy|<=n and |y|>0
Step 3: Find a suitable integer I such that xyz Є L. This contradicts our assumption. Hence L is not regular.

Example:

Show that $L=\{a^nb^n|n>=0\}$ is not regular.

Step 1: Assume L is regular.
Step 2: Let $w=a^2b^2 \in L$ for n=2
by using pumping lemma
w=aabb Є L
x=a,y=a,z=bb
pump y
w=a aa bb
again pump y
w=a aaa bb
$w=a^4b^2 \in L$
Step 3: There is a contradiction. So our assumption is wrong
Hence L is not regular

Closure Properties of Regular Sets:

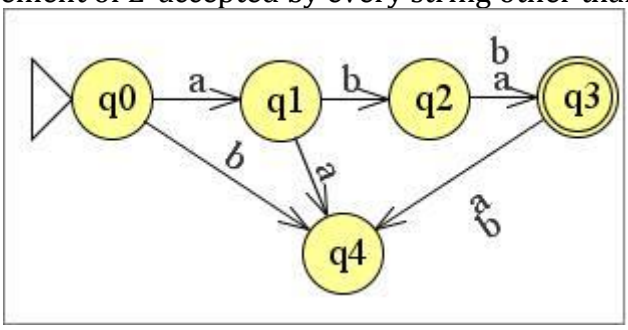
Theorem: Closure Properties of regular languages.

Statement: The class of languages accepted by Finite automata is closed under

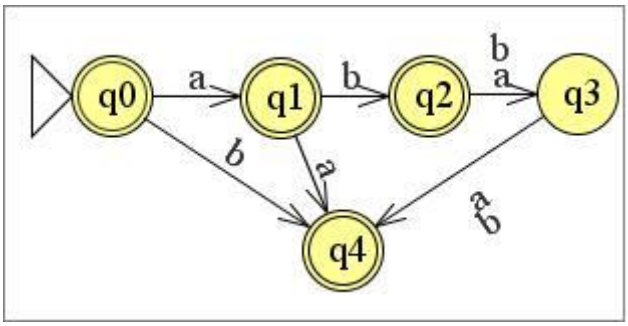
1. Union
2. Concatenation
3. Kleene Closure
4. Complementation

5. Intersection
6. Difference
7. Reversal
8. Homomorphism
9. Inverse Homomorphism

Example: An FA that accepts only the strings aba and abb is shown in below figure, find Fa for the complement of L' accepted by every string other than aba and abb



To find the complement of the given DFA make all non-final states as final and final as non-final.



Reversal: The reversal of string $a_1a_2a_3a_4.....a_n$ is the string written backwards, that is $a_n\ a_{n-1}\ a_{n-2}.....\ a_3\ a_2\ a_1$. We use W^r for the reversal of string W.

Example: 1010 is the reversal of 0101

Example: Let L be defined the regular expression (0+1)0* then find L^R

L=(0+1)0*

Solution: $L^R = (0^*)^R(0+1)^R$

Homomorphism: A String homomorphism is a function on strings that works by substituting a particular string for each symbol.

Suppose Σ and Σ' are alphabets. Then function h: Σ--> Σ' is called a "homomorphism". In other words, a homomorphism is a substitution in which a single letter is replaced with a string. The domain of the function h is extended to string in an obvious fashion if $w=x_1x_2.....x_n$ then $h(w)= h(x_1)\ h(x_2)\ h(x_3).....\ h(x_n)$

If L is a language on Σ, then its homomorphic image is defined as

$h(L)=\{h(w):w \in L\}$

Example: Let Σ={0,1} and Σ'={0,1,2} and defined n by h(0)=01 ,h(1)=112 then find h(010) and hommorphic image of {00,010}

Solution : Given Σ={0,1} , Σ'={0,1,2}

h is defined as h(0)=01,h(1)=112

So H(010)=0111201

The homomorphic image of L={00,010} is the language h(L)={0101,0111201}

UNIT – IV
Context Free Grammars

Derivation Tree: If W is a string in the context free grammar G .W Є L(G).Then the derivation of W is represented a tree is called derivation tree or parse tree.

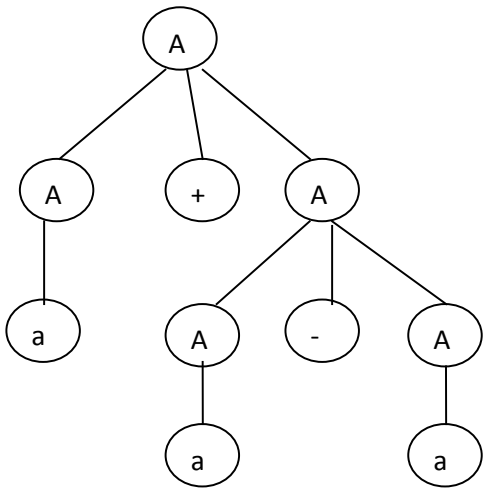
In the derivation tree we have

1. The starting sumbol S is the root.
2. All the internal nodes in the parse tree are variable.
3. All the leaf nodes are terminals.
4. If we write all the leaf nodes of the tree from left to right we get a string , that string is called "Derivation " or "Yield" of that tree.

Ambiguous Grammar: A Grammar G is said to be ambiguous if there exist a string W belongs to L(G) has two or more left derivations or two or more right derivations such grammar is called ambiguous grammar.

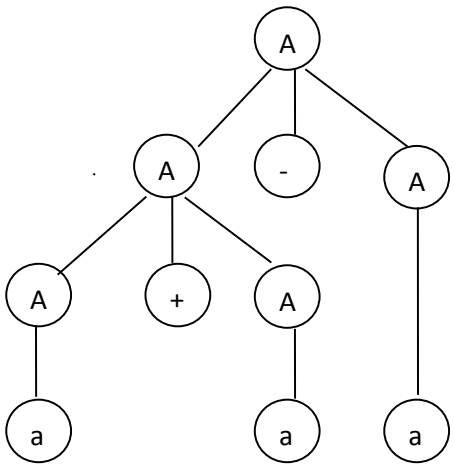
Example: A→A+A|A-A|a W=a+a-a

Left Most Derivation (LMD) -1 : A→A+A
A→A+a
A→a+A-A
A→a+a-A
A→a+a-a



Parse Tree for the above derivation:

Left Most Derivation(LMD) -2 : A→A-A
A→A+A-A
A→a+A-A
A→a+a-A
A→a+a-a



Parse Tree for the above derivation:

There exist two left derivations , So the given grammar is ambiguous.

Left Recursion: A grammar is said to be Left recursive if and only if it is of the form A→Aα such that A is a variable, and α Є (VUT)*.

Example: S→S+S
S→Sa

Elimination of Left Recursion: Consider the grammar G is in the form

A→A α1|A α2....A αn|β1| β2... βm where β1, β2... βn doesn't starts with A. It is said to be Left Recursive production. This can be eliminated by introducing the following productions.

$$A \rightarrow \beta 1A^1 | \beta 2A^1 ..... | \beta mA^1$$

$$A^1 \rightarrow \alpha 1\ A^1 | \alpha 2A^1 ..... | \alpha nA^1 | Є$$

Examples:

1. **Consider the CFG S→S+S|S*S|a|b eliminate the left recursion if any .**
   Sol: S→S+S|S*S|a|b is in the form of A→A α1|A α2....A αn|β1| β2... βm so it is in left recursion.
   After eliminating left recursion, the required grammar is
   $S \rightarrow aS^1 | bS^1$
   $S^1 \rightarrow +SS^1 | *SS^1 | Є$
2. **Consider the S→Aa|b, A→Ac|Sd| Є eliminate the left recursion if any .**
   Sol: S→Aa|b
   A→Ac|Sd| Є
   In the production of A replace S with S→Aa|b then
   A→Ac|Aad|bd| Є
   S→Aa|b
   In production A we have Left recursion , so eliminate Left recursion.
   $A \rightarrow bdA^1 | A^1$
   $A^1 \rightarrow cA^1 | adA^1 | Є$
   After elimination of left recursion the required Grammar is
   S→Aa|b
   $A \rightarrow bdA^1 | A^1$
   $A^1 \rightarrow cA^1 | adA^1 | Є$

Left Factoring: Two or more productions of a variable A of the grammar G is said to be left factoring if all the productions of the form

A→α β1| α β2|..... α βn where βi Є(VUT)* and βi doesn't starts with α then all the productions are said to have common left factor α example

S→ab|ac|ad Here a is called common left factor.

Elimination of Left Factoring: Consider the A productions which are having the left factoring as follows

A→α β1| α β2|..... α βn| γ1| γ2 .....| γm

γ 1, γ2 ..... γm doesn't starts with α

we can eliminate left factoring in the following way

A→ αA¹| γ1| γ2 .....| γm

A¹→β1|β2.....βm.

Examples:

1. **Consider the CFG S→aSa|aa|b eliminate left factoring.**

Sol: In the given grammar there exist left factoring . a is common left factoring. We can eliminate left factoring in following way.

S→aA¹|b

A¹→ Sa|a

2. **Consider the grammar E→E*T|F|E+T|T**

T→F

F→id **eliminate left factoring if any.**

Sol: In the given grammar there exist left factoring . 'id' is common left factoring.
To eliminate that left factoring we can write the productions as follows.

E→idE¹

E¹→*T|+T|id

T→id

Simplification of Grammars:

Grammar may contain some extra symbols , these will increase the length of the grammar. Elimination of these unnecessary symbols is called simplification of CFGs.
Simplification of grammars generally includes

a. Elimination of useless symbols.
b. Elimination of Є productions.
c. Elimination of unit productions of the form A→B.

a. **Elimination of useless symbols:** A symbol is useless if it can not derive a terminal or it is not reachable from start symbol.

Examples:

1. **Eliminate useless symbols and productions from the following grammar.**

S→ABa|BC, A→aC|BCC, C→a, B→bcc, D→E, E→d, F→e

Sol: In the given grammar the non terminals D,E,F are not reachable from the start symbol 'S', so we can eliminate them And the simplified grammar is

S→ABa|BC, A→aC|BCC, C→a, B→bcc,

2. **Eliminate useless symbols in G.**
   S→AB|CA, S→BC|AB, A→a, C→aB|b.

Sol: In the given grammar there is no production for B. So we have to eliminate the productions which contains B.

The simplified grammar is

S→CA
A→a
C→b.

3. **Eliminate useless symbols in G. S→aAa, A→bBB, B→ab, C→ab.**

Sol: In the Given grammar the variable C is not derived from the start symbol 'S'. So 'C' is useless.

The simplified grammar is
S→aAa,
A→bBB,
B→ab.

b. Elimination of Є productions:

If some CFL contains the word Є then CFG must have a Є-production. However if a CFG has a Є-production then the CFL doesn't necessarily contain Є.

Example: S→aX
X→Є
CFL={a}

Nullable Variables: In a given Context free grammar a non terminal X is nullable if

1. There is a production X→Є
2. There is a derivation that starts at X and leads to Є
   i.e X→-----→Є

Procedure for eliminating Є – productions:

Step 1: Construct set Vn of all nullable variables.
Step 2: For each production B→A, if A is nullable variable, replace nullable variable by Є and add with all possible combinations on the RHS.
Step 3: Do not add the production A→ε .

Examples:

1. **Eliminate Є-productions from the following grammar G.**
   S→ABaC, A→BC, B→b|ε, C→D|ε, D→d.

Sol: nullable variable are Vn = {B,C,A}
Because B, C are having Є-productions and production A leads to Є.
Now we have to replace the nullable variable with Є.

S→ABaC|AaC|ABa|aC|a|Aa|Ba
A→BC|B|C
C→D
B→b
C→d.

2. **Eliminate Є-productions from the following grammar G.**
   S→aA, A→BB, B→aBb|Є

Sol: nullable set Vn= {B}

S→aA|a
A→BB|B
B→aBb|ab.

c. **Elimination of unit productions**: A production which is of the form A→B where A, B are variables is said to be unit productions.

For each pair of non-terminals A and B such that there is a production A→B and the non-unit productions from B are B→S1|S2|...Sn
Where Si Є(TUV)* are strings of terminals and non-terminals then create new productions as
A→S1|S2|...Sn
Do the same for all such pairs A and B simultaneously.

Examples:

1. **Eliminate the unit productions in the grammar**
   S→A|bb, A→B|b, B→S|a

Sol: In the given grammar we have following unit productions

S→A, A→B, B→S.

After eliminating the above unit productions, the required grammar is as follows.

S→b|bb|a
A→b|bb|a
B→a|bb|b

2. **Eliminate the unit productions in the grammar**
   S→Aa|B
   B→A|bb
   A→a|bc|B

Sol: In the given grammar we have following unit productions

S→B, B→A, A→B.

After eliminating the above unit productions, the required grammar is as follows.

S→Aa|a|bc|bb

B→bb|a|bc

A→a|bc|bb.

Problems:

*** Simplify the grammar S→aA|aBB, A→aAA|Є,B→bB|bbC, C→B.**

Sol: Removing Є-productions gives resulting grammar as

S→aA|a|aBB

A→aAA|aA|a

B→bB|bbc

C→B

Eliminating unit productions we get the resulting grammar as

S→aA|a|aBB

A→aAA|aA|a

B→bB|bbc

C→bB|bbC

B and C are identified as useless symbols. Eliminate these we get

S→aA|a

A→aAA|aA|a

Finally the reduced grammar is S→aA|a, A→aAA|aA|a which defines any number of a's

Normal Forms:

If G is a Context free grammar and the production of G satisfy certain properties then G is said to be in a normal form. There are two types of normal forms.

1. Chomsky Normal Form (CNF)
2. Greibach Normal Form (GNF)

1. Chomsky Normal Form (CNF): A grammar G is said to be in a normal form if all the productions of the form

<variable> → <terminal> (A→a)

<variable> → <variable><variable> (S→AB)

Note: CNF allows only a single terminal or two variables on RHS of the each production.

Procedure for Converting CNF:

1. Eliminate null productions and unit productions .

2. Include productions of the form
<variable>→<variable><variable>
<variable>→<terminal>.
3. Eliminate string of terminals on the right hand side of the productions if it exceeds one as follows,
for example if we have S→a1a2a3 where a_1,a_2,a_3 are terminals
then introduce a non terminal 'Ca_i' for terminal a_i as

$Ca_1 \rightarrow a_1$
$C\ a_2 \rightarrow a_2$
.
.
.
$Ca_n \rightarrow a_n$

4. To restrict number of variables in the right hand side introduce new variable and separate them as follows, suppose we have the production with N non terminals as shown below

$Y \rightarrow X_1X_2X_3X_4X_5$

Add N-2 new productions using n-2 new non terminals and modify the productions as shown in below.

$Y \rightarrow X_1R_1$
$R_1 \rightarrow X_2R_2$
$R_2 \rightarrow X_3R_3$
$R_3 \rightarrow X_4X_5$
Where R_i are new non terminals.

Note: the languages generated by the new context free grammar is same as the original context free grammar

Problems:

1. **Convert the following CFG to CNF S→AB|aB , A→aab|Є , B→bbA**

Sol: In the given grammar there is Є-production , So we have to eliminate null production (Є-production) as follows.

S→AB|aB|B
A→aab
B→bbA|bb.

After the elimination of Є-productions we get a unit production S→B , So we have to eliminate the unit production.

S→ AB|aB|bbA|bb
A→aab
B→bbA|bb

Then S→AB already in CNF form.

S→bbA
$S \rightarrow C_bC_bA$
$D \rightarrow C_bC_b$
$C_b \rightarrow b$
S→DA

$S \rightarrow C_bC_b$
$S \rightarrow C_aB$
$C_a \rightarrow a$
$A \rightarrow C_aC_aC_b$
$E \rightarrow C_aC_a$
$A \rightarrow EC_b$
$B \rightarrow DA$
$B \rightarrow C_bC_b$

Then the final production in CNF is

$S \rightarrow AB$
$S \rightarrow DA$
$S \rightarrow C_bC_b$
$S \rightarrow C_aB$
$D \rightarrow C_bC_b$
$A \rightarrow EC_b$
$E \rightarrow C_aC_a$
$B \rightarrow DA$
$B \rightarrow C_bC_b$
$C_a \rightarrow a$
$C_b \rightarrow b$

Tuple representation for the grammar:

V=Set of variables={S,A,B,Ca,Cb,D,E}
T= Set of terminals={a,b}
P= productions={ $S \rightarrow AB, S \rightarrow DA, S \rightarrow CbCb, S \rightarrow C_aB, D \rightarrow C_bC_b, A \rightarrow EC_b, E \rightarrow C_aC_a, , B \rightarrow DA, B \rightarrow C_bC_b , C_a \rightarrow a, C_b \rightarrow b$}
S= start symbol= S.

2. **Greibach Normal Form (GNF):**

Let G be the context free grammar, if all the production G of the form $A \rightarrow a\alpha$. Where $\alpha \in V^*$ i.e. A GNF grammar allows a single terminal and any number of variables on R.H.S at the productions.

Example: (i) $A \rightarrow aA$----valid
(ii) $A \rightarrow aAB$---valid
(iii) $A \rightarrow aAA$---valid
(iv) $A \rightarrow abAA$---not valid

Procedure for Converting GNF:

1. Eliminate null productions , unit productions and useless symbols and construct CNF.
2. Rename variables as $A_1, A_2, \ldots$..with $S=A_1$
3. For each production of the form $A_i \rightarrow A_j\alpha$ apply the following
   (a) If j>i leave the production as it is.
   (b) If j=i eliminate left recursion

(c) If j<I apply substitution rule.

4. For each production of the form $A_i \rightarrow A_j\alpha$ where j>I apply substitution rule if A_j is in GNF, to bring Ai to GNF.

For converting a given grammar to GNF, we need two Lemmas.

Lemma 1: Substitution Rule: Let $A \rightarrow B\alpha$ be a production in P and B is $B \rightarrow \beta_1|\beta_2|\beta_3|\beta_4$…. The equivalent grammar can be obtained by substituting B in A then resulting grammar is $A \rightarrow \beta_1\alpha|\beta_2\alpha|\beta_3\alpha$….

Lemma 2: Elimination of Left Recursion: Consider the grammar G is in the form

$A \rightarrow A\ \alpha 1|A\ \alpha 2....A\ \alpha n|\beta 1|\ \beta 2...\ \beta m$ where β1, β2... βn doesn't starts with A. It is said to be Left Recursive production. This can be eliminated by introducing the following productions.

$$A \rightarrow \beta A^1$$
$$A^1 \rightarrow \alpha\ A^1 \mid \epsilon$$

If we eliminate Є production, then we get

$$A \rightarrow \beta A^1|\beta$$
$$A^1 \rightarrow \alpha A^1 \mid \alpha$$

Examples on GNF:

1. **Convert the following CFG to GNF**
   S→AA|a, A→SS|b.

Sol:

STEP 1: There is no Є-productions, unit productions, null productions, already the grammar is in CNF

STEP 2: Rename variables as A_1,A_2...i.e S=A_1, A=A_2 then the production will be

$A_1 \rightarrow A_2A_2|a$----------1

$A_2 \rightarrow A_1A_1|b$----------2

STEP 3:

Take eq 1 $A_1 \rightarrow A_2A_2|a$ here j >i so, leave the production as it is

Take eq 2 $A_2 \rightarrow A_1A_1|b$ here j < i so, apply substitution rule i.e substitute A_1 production in A_2 production.

$A_2 \rightarrow A_2A2A_1|aA_1|b$ here j = i so, apply left recursion.

$A_2 \rightarrow aA_1Z|bZ|aA_1|b$

$Z \rightarrow A_2A_1Z|A_2A_1$

Substitute A_2 in Z then

$Z \rightarrow aA_1ZA_1Z|bA_1Z|aA_1A_1Z|bA_1Z|aA_1ZA_1|bZA_1|aA_1A_1|bA_1$

STEP 4: Substitute A_2 in A1 then

$A_1 \rightarrow aA_1ZA_2|bZA_2|aA_1A_2|bA_2|a$

$A_2 \rightarrow aA_1Z|bZ|aA_1|b$

Tuple Representation: G(V, T, P, S)

V= {A_1, A_2, Z}

T= {a,b}

P={ A_1→$aA_1ZA_2|bZA_2|aA_1A_2|bA_2|a$, A_2→$aA_1Z|bZ|aA_1|b$,

Z→$aA_1ZA_1Z|bA_1Z|aA_1A_1Z|bA_1Z|aA_1ZA_1|bZA_1|aA_1A_1|bA_1$}.

S=A1.

More problems: 1. Convert the following CFG to GNF S→ABA, A→aA|Є, B→bB|Є

2. Convert the following S→AB, A→BS|b, B→SA|a into GNF.
3. Convert the following S→S+S|S*S|a|b into GNF.
4. Convert the following S→XA|BB, B→b|SB, X→b, A→a.

Pumping Lemma for CFL:

Let L be a CFL, then we find a natural number n such that.

(i) Every ZЄL with |Z|>=n can be written as "UVWXY" for some strings U, V, W, X, Y.

(ii) |VX|>=1

(iii) |VWX|<=n

(iv) UV^KWX^KY Є L for all K>=0.

Problems:

1. **Prove that L={$0^n1^n2^n$ | n>=1} is not in CFL.**

Assume L is in Context free language. Assume L=2. Then

Z=$0^21^22^2$=001122 Є L

00 1 1 2 2

U V WXY

For i=2 Pump V,X

$UV^2WX^2Y = 001^212^22$

$=0^21^32^3$ does not belongs to L.

There is a contradiction. So our assumption is wrong. Hence given grammar is not Context free language.

2. **Prove that L = {ww|w is bit string} is not in CFL.**

Assume L is in Context free language.

$Z=0^n1^n0^n1^n$ Є L

UV^iWX^iY Є L

$Z=\underline{0}^n\ \underline{0}^n\ \underline{1}^n\ \underline{0}^n\ \underline{1}^n$

U V W X Y

Pump V, X

$Z= 0^n\ 0^{n+1}\ 1^n\ 0^{n+1}\ 1^n$ does not belongs to L.

Enumeration properties of CFL:

- Context free languages are closed under substitution , union, concatenation, closure, positive closure, reversal homomorphism, inverse homomorphism,
- Context free languages are not closed under intersection, difference, and complement.

Applications of Context free languages:

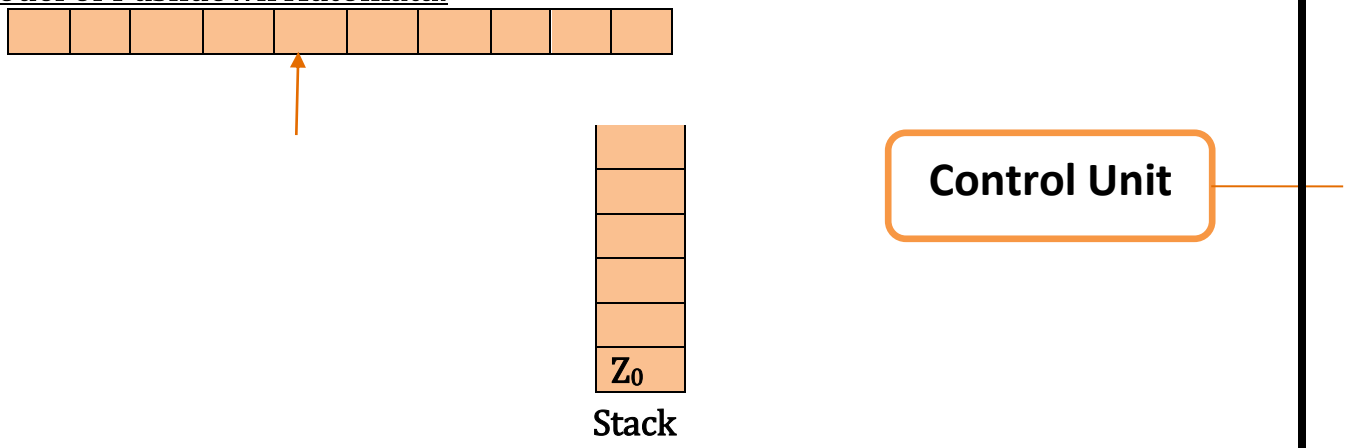
- Grammars are useful in specifying syntax of programming languages. They are mainly used in design of programming languages.
- They are used in natural language processing.
- CFGs are used in speech recognitions also in processing the spoken word.
- The expressive power of CFG is too limited to adequately capture all natural language phenomena. Therefore extensions of CFG are of interest for computational linguistics.

UNIT – V
Push Down Automata

Push Down Automata:

- The concept of push down automata is useful in the design of parsers or syntax analyzers.
- The parser verifies the syntax of the text.
- Parsing is a part of the compilation process.

Model of Pushdown Automata:



The PDA consists of three components.

1. A input tape
2. A finite control
3. A stack structure

- ❖ An input tape consists of a linear configuration of cells each of its contain a character from the input alphabet.
- ❖ The control unit has some pointer (reading head). This points to the current symbol which is to be read. The head position over the current stack element can read and write special stack character from that position.
- ❖ The stack is also a sequential structure that has a first element and close in either from the other end.

Formal Definition of PDA:

A finite state push down automata is a seven Tuple $M(Q,\sum, \delta,\Gamma,F,Z_0,q_0)$ where

Q=Finite set of states.
$\sum$=Finite set of Input alphabet
Γ=Finite set of stack alphabet
q_0=Initial state
F=Finite set of final state $F \subseteq Q$
Z_0=Initial stack symbol $Z_0 \in \Gamma$
δ=Transition function $\delta: Q \times (\sum \cup \{\varepsilon\}) \times \Gamma \dashrightarrow Q \times \Gamma^*$

1. PDA has two alphabets
    1. an input alphabet $\sum$ (input string)

2. a stack alphabet Γ(stored on the stack)

2. A move on PDA may indicate
   1. an element may added to stack
   2. an element may be deleted from
      $(q, a, Z_0)=(q,\varepsilon)$
      q=current state
      a=input symbol
      Z_0=stack symbol
   3. they may or may not be change state.

3. They may or may not be change of state.

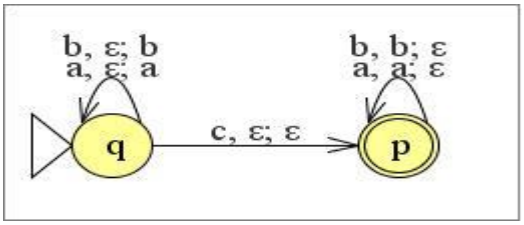
Example:

1. $\delta(q_0,a, Z_0)=(q_0, aZ_0)$ indicates that in the state q_0 on seeing a, a is pushed onto the stack. There is no change of state.
2. $\delta(q_0,a, Z_0)=(q_0, \varepsilon)$ indicates that in the state q_0 on seeing a, the current top symbol Z_0 is deleted from stack. There is no change of state.
3. $\delta(q_0,a, Z_0)=(q_1, aZ_0)$ indicates that a is pushed onto the stack and state is changed to q_1

Graphical Representation of PDA:

Let $M(Q,\sum, \delta,\Gamma,F,Z_0,q_0)$ be a PDA where $Q=\{p,q\}$, $\sum=\{a,b,c\}$, $F=\{a,b\}$, $q_0=q$, $F=\{p\}$ and δ is given as follows:

$\delta(q,a,\varepsilon)=(q,a)$, $\delta(q,a,a)=(q,aa)$
$\delta(q,b,\varepsilon)=(q,b)$ $\delta(q,b,b)=(q,bb)$
$\delta(q,c,\varepsilon)=(p,\varepsilon)$
$\delta(p,a,a)=(p,\varepsilon)$
$\delta(p,b,b)=(p,\varepsilon)$



Instantaneous Description of PDA-String Processing :

W=ababcbab

$\delta(q,ababcbab,\varepsilon)$ $\Rightarrow \delta(q,babcbab, a)$
$\Rightarrow \delta(q,abcbab,ba)$
$\Rightarrow \delta(q,bcbab,aba)$
$\Rightarrow \delta(q,cbab,baba)$
$\Rightarrow \delta(p,bab,baba)$
$\Rightarrow \delta(p,ab.aba)$
$\Rightarrow \delta(p,b,ba)$
$\Rightarrow \delta(p,\varepsilon,a)$

At this point the input string exhausted and the computation stops. We can not accepts the original string. Even we are in accept state because stack is not empty.

Language accepted by PDA:

A language can be accepted by a pushdown automata using two approaches

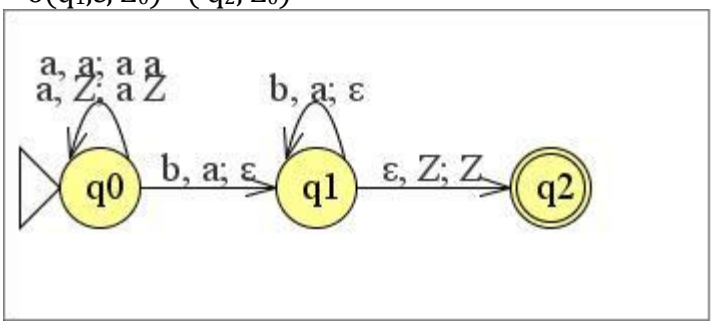
1. Acceptance by Final State: The PDA accepts its input by consuming it and finally it enters in the final state.
2. Acceptance by empty stack: On reading the input string from the initial configuration for some PDA the stack of PDA becomes empty.(Empty input & stack empty)

Design of Push Down Automata:

1. **Design a PDA which accepts $L=\{a^n b^n \mid n>=1\}$**

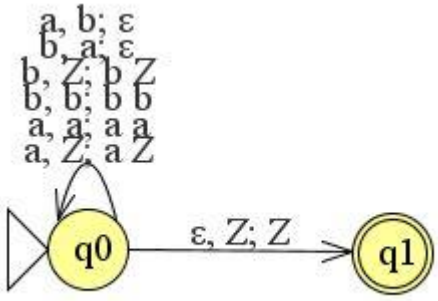
   Strings generated by language L={ab,aabb,aaabbb,aaaabbbb,.....}

   $\delta(q_0,a,Z_0)=(q_0,aZ_0)$
   $\delta(q_0,a,a)=(q_0,aa)$
   $\delta(q_0,b,a)=(q_1,\varepsilon)$
   $\delta(q_1,b,a)=(q_1,\varepsilon)$
   $\delta(q_1,\varepsilon,Z_0)=(q_2,Z_0)$



2. **Design PDA which accepts equal number of a's and b's over ∑={a,b}**

   $\delta(q_0,a,Z_0)=(q_0,aZ_0)$
   $\delta(q_0,a,a)=(q_0,aa)$
   $\delta(q_0,b,Z_0)=(q_0,bZ_0)$
   $\delta(q_0,b,a)=(q_0,\varepsilon)$
   $\delta(q_0,a,b)=(q_0,\varepsilon)$
   $\delta(q_0,b,a)=(q_0,\varepsilon)$
   $\delta(q_0,\varepsilon,Z_0)=(q_1,Z_0)$



Deterministic PDA:

- The PDA that has one choice of move in any state is called deterministic PDA.

A PDA P={ Q,∑, δ,Γ,F,Z,q_0} is deterministic if and only if

1. δ(q,a,X) has atmost one number of q ∈a, a∈ Z or a=ε and X∈Γ.
2. if δ(q,a,X) is not empty for some a∈∑ then δ(q,ε,X) must be empty

- DPDA is less powerful than NPDA
- The context free languages could be recognized by NPDA.

Deterministic Context Free Languages:

- The class of language accepted by DPDA is in between that of regular language and CFL.
- This language is called deterministic context free languages and it is that subset of language accepted by NPDA.
- DCFLs are closed under union, concatenation, Kleene closure and complement.

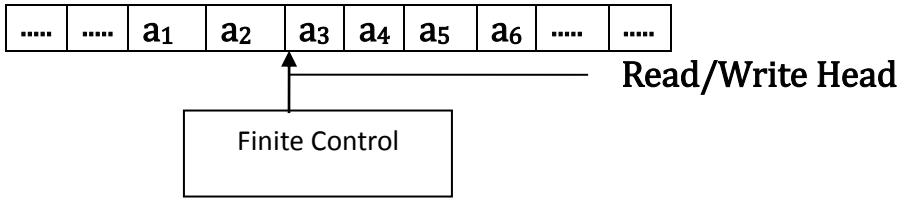
Note: CFL's are not closed under complementation But DCFL's are closed . Hence if the CFL's is formed to be closed under complement then it is DCFL.

UNIT – VI
Turing Machine

Turing Machine:

- Turing Machines first described by Alan Turing.
- Turing Machines are simple abstract computational devices intended to help investigate the extent and limitations of what can be computed.
- Some simple languages like $\{a^n b^n c^n\} n>=0\}$ is not recognized by either PDA or Finite Automata.
- Turing Machine is a device that will recognize this language and many more complicated languages.
- Turing machines are used as language recognizers, languages enumerator and as a computing machine.

Model of Turing Machine:



- Turing machine can be thought of as finite control connected to a Read/Write head.
- It has one tape which is divided into a number of cells.
- Each cell can store only one symbol.
- The input to and the output from the finite state automaton are affected by the Read/Write head which can examine one cell at a time.
- In one move the machine examine the present symbol under the R/W head on the tape and the present state of an automaton to determine.
  - ❖ A new symbol to be written on the tape in the cell under the R/W head.
  - ❖ A motion of the R/W head along the tape either the head moves one cell left(L), or cell right(R).
  - ❖ the next state of the automaton and
  - ❖ whether to halt or not.

Formal Definition: A Turing machine M is a 7 Tuple, namely $(Q,\sum, \delta,\Gamma,F,Z_0,q_0)$ where

Q=Finite set of states.
$\sum$=Finite set of Input alphabet
Γ=Finite set of tape alphabet
q_0=Initial state
F=Finite set of final state $F \subseteq Q$
B=Blank symbol $B \in \Gamma$

δ=Transition function δ: (q X Γ) --> Q X Γ*

Acceptance by Turing Machine:

The acceptability of a string is decided by the reachability from the initial state to some final state. So the final states are also called the accepting states.

Representation of Turing Machine:

Turing machine employing can be desired by:

1. Instantaneous description using move relation
2. Transition table.
3. Transition Diagram.

ID of Turing Machine:

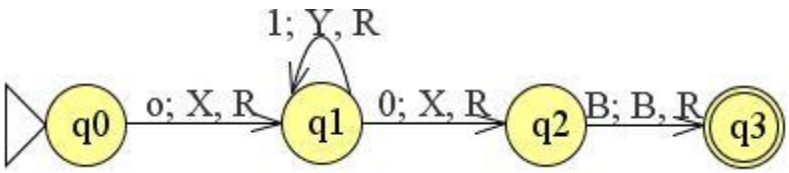
An ID of Turing machine M is a string αβγ, where β is the present state of M, the entire input string is split as αγ, the first symbol of γ is the current symbol α under the R/W head and γ has all the subsequent symbols of the input string formed by all the symbols to the left of α.

Transition Table:

We give the definition of δ in the form of a table called the transition table if δ(q, α)=(γ, α,β) we write αβγ under the α-column and in q-row.

α-current cell
β-movement -Right or Left
γ-next state or new state

Transition Diagram:



Designing of Turing Machines:

1. Design a Turing Machine that accepts L={$a^n b^n$|n>=1}

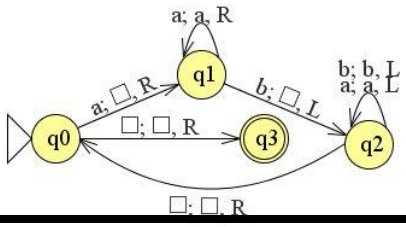
Solution:

We require the following moves:

L={ab,aabb,aaabbb,aaaabbbb,.....}

a) If the left most symbol in the given input string W is a, replace it by B and move right till we encounter a leftmost 1 in W. Change it to B and move backwards until we encounter B(Blank symbol).
b) Repeat (a) with the leftmost a. If we move back and forth and no a or b remains, move to a final state.
c) For the string in the form $a^n b^n$, the resulting state has to be non-final.

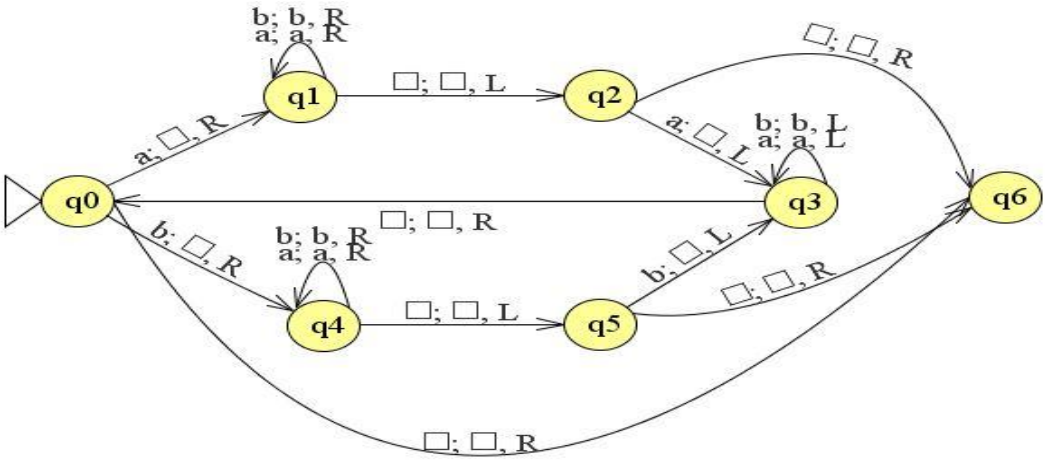
Note:□ndicates Blank symbol(B)



Tuple Representation:

Q=Finite set of states={q_0,q_1,q_2,q_3}
∑=Finite set of Input alphabet={a,b}
Γ=Finite set of tape alphabet ={a,b,B(□) }
q_0=Initial state= q_0
F=Finite set of final state ={ q_3}
B=Blank symbol B∈ Γ
δ=Transition function

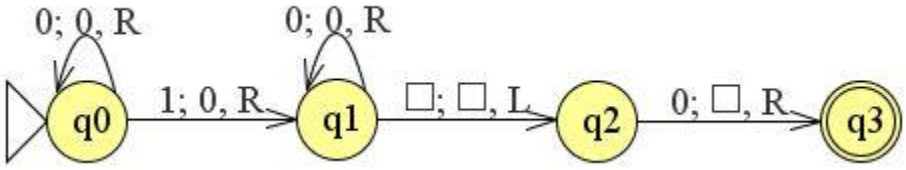
2. Design a Turing Machine which accepts all the palindrome Strings over the alphabet {a,b}



Turing Machine As Computational Machine:

A TM M compute a function f if, when given input w in the domain of f, the machine halts in its accept state with f(w) written the tape. To use Turing machine as a computation machine it is required to place the integer numbers as 0^m. Suppose it is required to add two numbers, i.e f(m,n)=m+n then the numbers m and n are to be placed on the tape as 0^m10^n where 1 is the separator for the numbers m and n. Once processing is completed and the Turing machine halts then tape would have the contents as $0^{(m+n)}$ which is the required result of computation.

3. Design a TM to add two numbers a and b.

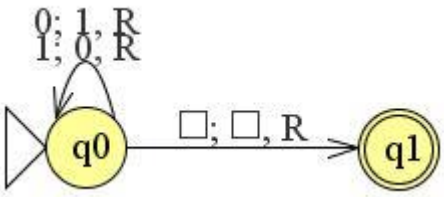


Transitions are as follows:

δ(q0,0)=(q0,0,R)
δ(q0,1)=(q1,0,R)

δ(q1,0)=(q1,0,R)
δ(q1,B)=(q2,B,L)
δ(q2,0)=(q3,B,R)

4.Design a TM for finding 1's complement of a given binary number.



Types of Turing Machines:

There are several types of Turing Machines. They are

1. Nondeterministic Turing Machine
2. Turing Machine with Two-Dimensional Tapes.
3. Turing Machines with Multiple Tapes
4. Turing Machines with Multiple Heads.
5. Turing Machines with Infinite Tape.

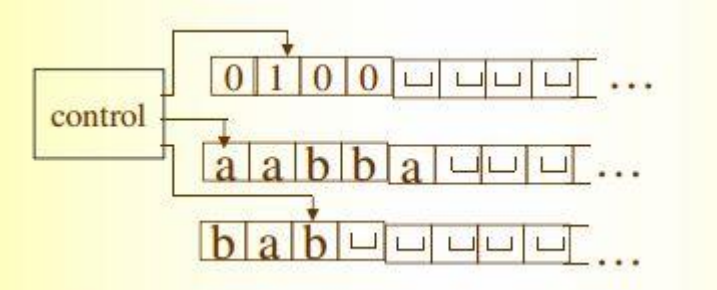
1. **Nondeterministic Turing Machine:**
   A Nondeterministic Turing Machine is a Turing machine which like NFA, at any current state and for the tape symbol it is reading there may be different possible actions to be performed.
   One action could be just change of state without modifying the cell content, ir change of the state with change of the cell content or change both state and cell content. In all action means the combination of writing a symbol on the tape, moving the tape head going to the next state.
   Example L={WW|W∈{a,b}}
2. **Turing Machine with Two-Dimensional Tapes**.
   - Turing machine with two dimensional tape is a kind of TM that has one finite control, one read-write head and one two dimensional tape.
   - The cell in the tape is two dimensional that is the tape has the top end and the left en but extends indefinitely to the right and down
   - It is divided into rows of small squares.

v	v	v	v	v	v	v	. . .	. . .
h	1	2	6	7	15	16	. . .	. . .
h	3	5	8	14	17	26	. . .	. . .
h	4	9	13	18	25	. . .	. . .	. . .
h	10	12	19	24	. . .	. . .	. . .	. . .
h	11	20	23	. . .	. . .	. . .	. . .	. . .
h	21	22	. . .	. . .	. . .	. . .	. . .	. . .
. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .

Fig: Two Dimensional Tape

3. **Turing Machines with Multiple Tapes:**

- A multi-tape TM is like an ordinary TM with several tapes.
- Each tape has its own head for reading and writing.
- Initially the input appears on tape 1, and others are blank.
- The transition function is changed to allow for reading, writing, and moving the heads on all tapes simultaneously. Formally,
  $\delta : Q \times \Gamma^k \rightarrow Q \times \Gamma^k \times \{L, R\}^k$,
  where k is the number of tapes.
- The expression $\delta(q, a_1, ..., a_k) = (r, b_1, ..., b_k, L, R, ..., L)$ means that, if the machine is in state q and heads 1 through k are reading symbols a1 trough ak , the machine goes to state r, writes symbols b1 through and movbeks each head to the left or right as specified.



- Multi-tape TMs appear to be more powerful than ordinary TMs, but we will show that they are equivalent in power.

4. **Turing Machine with Multiple Heads:**
   - This is a kind of Turing Machines that have one finite control and one tape but more than one read-write heads.
   - In each state only one of the head is allowed to read and write.
   - The transition function is partial function
     δ:QX{H1,H2,.....Hn}X{ΓU{B}}--> (QU{h},{ΓU{B})X{R,L,S}
5. **Turing Machine Infinite Tape:**
   - This is a kind of Turing machines that have one finite control and one tape which extends infinitely in both directions.
   - It turns out that this type of Turing machines are only as powerful as one tape Turing machine whose tape has a left end.

Recursive and Recursively Enumerable Languages:

- There are three possible outcomes executing a Turing machine over a given input. The Turing Machine may
  - Halt and accept the input
  - Halt and reject the input

- Never Halt

- A language is recursive if there exists a Turing Machine that accepts every string of the language and reject every string (over the same alphabet) that is not in the language.
  Note: If a language L is recursive then its compliment L' must also be recursive
- A language is recursively enumerable if there exists a Turing machine that accepts every strings of the language, and does not accepts strings that are not in the languages.
  - Strings that are not in the language may be rejected or may cause the Turing machine to go into an infinite loop.
- Every recursive languages is also recursively enumerable. It is not Obvious whether every recursively enumerable languages is also recursive.

Church's Hypothesis:

Church's Hypothesis states that any algorithmic procedure that can be carried out by a human or computer, can also be carried out by Turing Machine.

A Problem can be solved by an algorithm if and only if it can be solved by Turing Machine.

Important of Church's Hypothesis:

1. First we will provide certain problems which can't be solved using TM.
2. If Church thesis is true this implies that problem cannot be solved by any computer or any programming language we might every develop.
3. Thus in studying the capabilities and limitations of Turing machine we are indeed studying the fundamental capabilities and limitations of any computational device we might even construct.

Computability Theory:

Decidability of Problems:

In general life, we have several problems and some of these have solutions also, but some have not.

Simply we say a problem is decidable if there is a solution otherwise undecidable.

Example: 1.Does the sun rise in the east?(YES)

2.Will tomorrow be a rainy day?(No Answer)

Decidable Problems:

A Problem is said to be decidable if

- Its language is recursive or
- It has a solution

Halting Problem:

The halting problem is a decision problem which is informally stated as follow: Given a description of an algorithm and description of its initial arguments, determine whether the algorithm when executed with these arguments, ever halts. The alternative is that given algorithm runs forever without halting.

Alan Turing Proved in 1936 there is no general method or algorithm which can solve the Halting Problem for all possible inputs.

Universal Turing Machine:

The church Turing thesis conjectured that anything that can be done on any existing digital computer can also be done by a TM. To prove this conjuncture A.M.Turing was able to construct a single TM which is the theoretical analogue of a general purpose digital

computer. This machine is called a Universal Turing Machine(UTM). He showed that the UTM is capable of initiating the operations of any other TM, that it, it is reprogrammable Tm. We can define this machine in more formal ways as follows:

A Universal Turing Machine is a TM that can take as input an arbitrary TM T_A with an arbitrary input for T_A and then perform the execution of T_A on its input.

To construct a UTM, we thus require three essentials

a. a uniform method to describe or encode any TM into a string over a finite symbol set, I
b. a similar method of encoding any input string for a TM into a string over I, and
c. a set of TM programs that describe the TMs basic cycle of operations.

Posts Correspondence Problem:

The POST correspondence problem(PCP) was first introduced by Emil Post in 1946.

The problem over an alphabet ∑ belongs to a class of Yes/No problems and is stated as follows:

Consider two lists $x=(x_1,x_2,x_3,.....x_n)$, $y=(y_1,y_2,y_3,.....y_n)$ of non-empty strings over an alphabet ∑={0,1}. The PCP is to determine whether or not there exist $i_1,i_2,....i_m$ where $1<=ij<=n$, such that $x_{i1}.....x_{im}=y_{i1}.....y_{im}$

Example:

Does the PCP with two lists x=(b,bab³,ba) and y=(b³,ba,a) have solution?

The Solution is (2113)

bab^3 b b ba= bab^3 b b ba