Finite Automata

Why Study Automata Theory? The Central Concepts of Automata Theory, Automation, Finite Automation, Transition Systems, Acceptance of a String by a Finite Automation, DFA, Design of DFAs, NFA, Design of NFA, Equivalence of DFA and NFA, Conversion of NFA into DFA, Finite Automata with E-Transition, Minimization of Finite Automata, Mealy and Moore Machines, Applications and Limitation of Finite Automata.

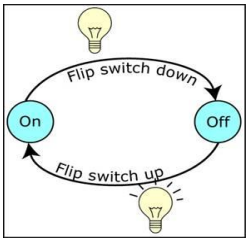
Why Study Automata Theory?

What is Automaton?

- An automaton is a system which obtains, transforms, transmits and uses information to perform its functions without direct human participation. It is self-operational.
- An automaton is a machine that has an input tape, that can be put in any of the several states. Various symbols are written on the tape before execution. The automaton begins reading the symbols on the tape, from left to right. Based on the symbol the state will be changed. After reading the input, machine halls.
- A general model of a machine is called a finite automaton; 'finite' because the number of states and the alphabet of input symbols is finite; automaton because the machine is deterministic, i.e, the change of state is completely governed by the input.

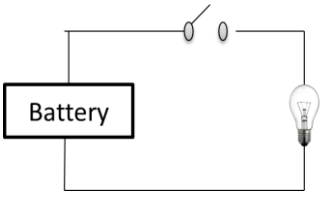
Finite automata model:

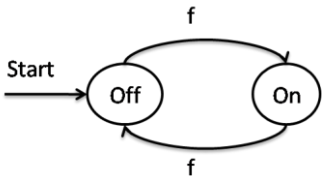
Finite automata are a useful model for any important kinds of hardware and software computer science. The following are some of the most important applications of automata.



1. Software for designing and checking the behaviour of the digital circuits
2. In the lexical phase of a typical compiler that is to breakup given text into logical units.
3. Software for scanning large bodies of information or data such as collection of web pages, to find occurrences of words, phrases and patterns
4. Software for verifying systems of all types that have a finite number of distinct starts, such as communication protocols or protocols for secure exchange of information.
5. Formal languages and grammars used widely to define programming languages.

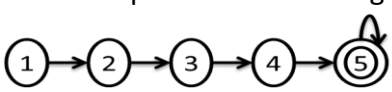
Input: Switch
Output: light bulb
Actions: f for "flip switch"
States: on, off



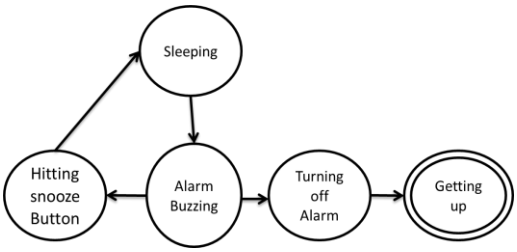


Examples of Finite State Machine

1.*Counting to five*: Here is a FSM, which is a process of counting upto five. The transition to state 5 is a loop back itself.



2. Getting up in the morning *:* In this example more than one arrow emerges from a particular circle i.e, either transition could occur. Notice two edges leaving from "alarm buzzing" state. That is each time when the alarm goes on in the morning, the person might choose to turn alarm off and get up or he might hit snooze button and go back to sleep.



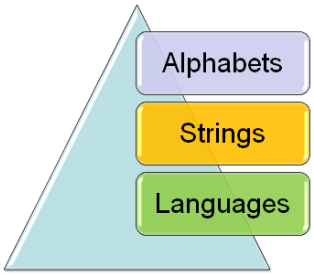
3. ***A simple computer system****:* Consider a computer with an input device, memory, processor and output device. To calculate 2+4, we need the instructions (STORE 2 TO X, STORE 4 TO Y, LOAD X, ADD Y, WRITE TO OUTPUT) Each instruction is executed as it is read. The final output is 6. The machine starts simply at an initial state and it changes from one state to another state based on the instruction and reaches the final state with output 7.

The Central Concepts of Automata Theory:-

Alphabets: An alphabet is a finite non empty set of symbols/ characters. It is denoted by $\sum$.

Examples:

- $\sum$ ={0,1}, is binary alphabet, consisting symbols 0 and 1.
- $\sum$ ={a,b,…,z}, set of lower case letters.
- $\sum$ ={a,b,c}, is an alphabet of three symbols.
- $\sum$ ={ $-,\cup \cap$ }, is alphabet consisting of some symbols of set operations.



Strings: A string is a finite sequence of symbols chosen from some alphabet. In other words, a string is a finite sequence of symbols over an alphabet.

Examples:

- $\sum$ ={0,1}, then 0110,1111,11,00,01,10 are some of the strings chosen from the alphabet.
- $\sum$ ={a,b,c}, then aaabcc,b,ab,bc,abc,cc,etc. are the words chosen from this alphabet.
- cse, is a string over the alphabet $\sum$ ={a,b,…,z}.

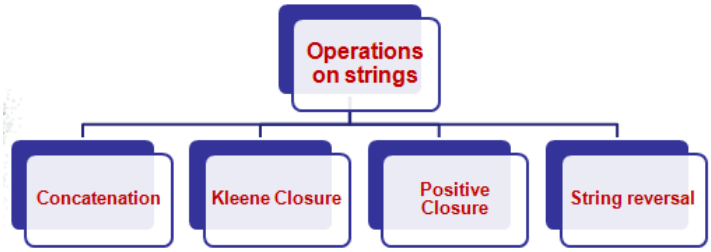
Empty String: It is a string with zero occurrences of symbol. The string denoted $\in$ (Epsilon) is a string that may be chosen from any alphabet.

Length of the String: The length of the string w is denoted |w|. If the number of symbols composing the string.

Eg: 1) 01101 has length 5

2) |011| = 3 and $|\in| = 0$

Operations of Strings: Basically, there are a number of operations on strings.



1. Concatenation of Strings : Concatenation of strings is simply the 'gluing' of strings together and placing them adjacent to each other in order to form a new string. Mathematically, S_1 and S_2 are two strings then the concatenation S' of S_1 and S_2 is given by $S' = S_1 0 S_2$

Example:

- If S_1= para and S_2=graph, then $S_1 \circ S_2$ =paragraph
- If Σ ={a,b}, S_1= ab and S_2=baa, then $S_1 \circ S_2$ =abbaa.
- abba $\circ \in$ = abba, Here $|\in| = 0$.

Recursive Definition for Concatenation of Strings: This kind of inductive definition is very common in computer science, is given by:

Let x be any string over an alphabet Σ. Then $x^0 = \in$; $x^{i+1} = x^i \circ x$, $i \in N$ where N={0,1,2,...}, is the set of non-negative integers.

2. Kleene Closure: It is also called as the kleene star and introduced by Stephen Kleene. It is a unary operation, either on sets of strings or on sets of symbols or characters. The application of the Kleene star to a set S is written as S^*. It is widely used for regular expressions.

If S is a set of symbols or characters, then S^* is the set of all strings over the symbols in S, including the empty string.

$S^* = \cup S^i$ where $i \in N$

$i \geq 0$ $i = \{\in\} \cup S^1 \cup S^2 \cup S^3 \cup \ldots$

where s^i is the i^{th} string of s

Example: Let S={aa, b}, then

S^*= { $\in$ or any word composed of factors of aa and b}

= {$\in$ or all strings of a's and b's in which a occurs in pairs}

= $\cup S^i$, $i \in N$

S^0={ $\in$ }

S^1=S={aa,b}

S^2= SS={ aaaa, aab, baa, bb}

S^3= ...

S^* = $S^0 \cup S^1 \cup S^2 \cup S^3$ ...= {$\in$,aa,b,aaaa,aab,baa,bb,....}

set of all strings that can be obtained by concatenating 0 or more copies of aa and b.

3. Positive Closure: The closure property, extended to set of all words of any length, except the null string i.e, $\in$, is the operation called positive closure. Thus, if S is a set of strings/words, then S^+ is called positive closure of S. Every thing is same as star closure except $\in$.

$$S^+ = S^1 \cup S^2 \cup \cdots$$
$$= S^* - \{\in\}$$

Example: Let S={aa, b}, then $S^+ = \cup S^i$, $i \in 1,..,N$

$S^1 = S = \{aa,b\}$

$S^2 = SS = \{ aaaa, aab, baa, bb\}$

$S^3 = \ldots$

$S^+ = S^1 \cup S^2 \cup S^3 \ldots = \{aa,b,aaaa,aab,baa,bb,....\}$

String reversal: Intuitively, when a string x is reversed i.e, spelt backwards, then x^R is the resulting string which satisfies:

$$x^R(i) = x(n+1-i), \text{ for } 1 \le i \le n$$

$$|x^R| = |x|$$

Example: a) if x = mus , then x^R = sum.

b) Let x = bottle. Then, x^R = elttob and $(x^R)^R$ = bottle = x.

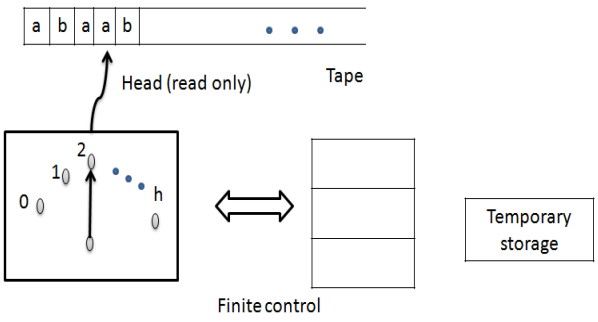
c) Let x=klim and y=rettub. Then, $(x \circ y)^R = (xy)^R = (kilmrettub)^R = buttermilk = butter \circ milk = y^R x^R$

Components of Finite State Automata

A Finite state automaton consists of 3 components, Control unit, a read unit and an input tape. The nature and operations of these components are as follows:

Input Tape:

- It is also called the ***data tape*** of the automaton.
- It consists of sequence of cells.
- Each cell begins at left hand end of the tape, contains one of the words of the string to be processed by the machine.
- The word themselves are usually identified as comprising input/data symbols, with the individual words treated as if they constituted single symbols.



Read Tape: It reads words from the cells of the input tape and provides the individual words, one at a time, to the control unit.

Control unit:

- It governs the operations of the automaton by performing sequence of transitions between internal states available to it.
- From the initial state, the control unit executes a transition as each word is provided to it by the read unit, these transitions are determined by ***transition function***.
- If, immediately after the last word of the input string has been read, the control unit moves to accepting state, then the automaton is said to accept or **recognize** the string of words.
- If the control unit is not in an accepting state after the last word is read, the automaton rejects the string.

Finite automaton can also be thought of as a device, which satisfies the following conditions:

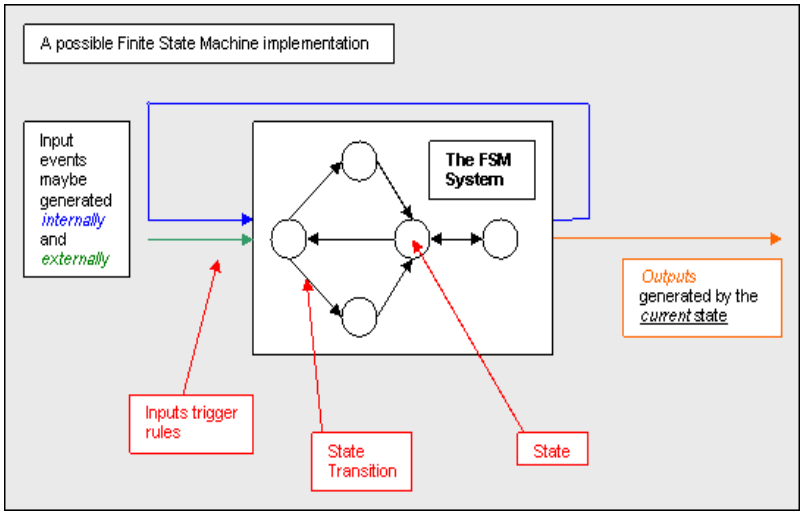
- The tape has left end and extends to right, without an end.
- The tape is divided into squares, where symbols can be written prior to start of the operation of automaton.
- The tape has read only head.
- The head is always at the leftmost square, at the beginning of the operation.
- The head moves to right by one square, every time it reads a symbols. It never moves to the left. When it sees no symbol, then stops and automaton terminate the operation.
- There is a finite control which determines the state of the automaton as well as the controls the movement of the head.

1.4 Elements of finite state system

Finite state machines consists of 4 main elements:

- States which define the behavior and may produce actions
- State transitions, which are movement from one state to another.
- Conditions which must be met to allow a state transition. Input events, which are either externally or internally generated. These may possibly trigger the rules and lead to state transitions.

A finite state machine must have an initial state which provides a starting point and a current state. Received input events act as triggers, which cause an evaluation of some kind of the rules that govern the transitions from the current state to other states. The FSM can be visualized as the flow chart or directed graph of states.



In a finite automaton at any given time, the control unit is in some internal state and the read unit of the automaton reads words from the cells of the input tape. The internal state is determined by the transition function. The transition function gives the next state, in terms of :

- Current state
- current input symbol
- the information currently present in the temporary storage.

During the transition from one state to other, output may be produced, or the information in the temporary storage may be changed. This transition from one state to the next is called **move**.

1. **State:** A state is a complete set of properties, transmitted by an object to an observer via one or more channels. Any change in the nature of such propertied in a state is detected by an observer and thus a transmission of information occurs.

The following are some of the types of states:

a) ***Start state:*** An initial state of a finite state machine.
b) ***Accepting state*****:** If a finite state machine finishes an input string and is in an accepting state, the string is accepted or considered to be valid.
c) ***Next state*****:** The state immediately followed by current state, defined by the transition function of a finite state machine and the input, is the next state.
d) ***Universal state:*** A state in an alternating Turing machine, from which the machine only accepts all the possible moves leading to acceptance, is called a 'Universal state'.
e) ***Existential state:*** A state in a nondeterministic Turing machine, from which the machine accepts any move that leads to acceptance, is the existential state.
f) ***Dead/Trap state****:* A non-final state of a finite state machine, whose transitions on every input symbol terminates on itself.

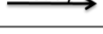
2. Transition

The following are the 2 equivalent definitions for the transitions of a finite state system.

a) Transition is the act of passing from one state to the next.
b) A change from one place or state or subject or stage, to another.

Transition are represented in the following ways:

- State diagram or Transition diagram
- State transition table
- Transition function

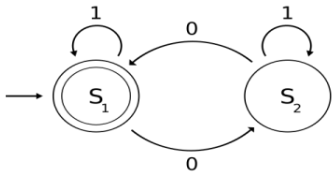
NOTATION	MEANING
	Start of the process
	Final state
	Transition
	Input
	Output

3. State diagram

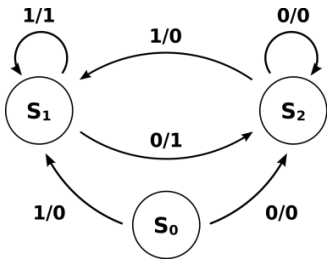
A diagram consists of circles to represent states and directed line segments to represent transitions between the states, is called a state diagram. The symbols used in state diagrams are enlisted.
For a finite state machine, a state diagram is a directed graph where,

a) each edge is a transition between two stages.
    i. For a **DFA,NFA,** and **Moore machine,** input is labeled on each edge.
    ii. For a **mealy machine,** input and outputs are labeled on each edge.
b) each vertex is a state.
    i. for a **moore machine**, output is signified for each state.

Example 1: S_1 and S_2 are states and S_1 accept state. Each edge is labeled with the input.



Example 2: S_0 , S_1 and S_2 are states. Each edge is labeled with "j/k", where j is the input and k is the output.



4. State transition table

A state transition table can be described, in general, as:

- Tabular representation of transitions that take 2 arguments and return a value.
- Rows correspond to states and columns correspond to inputs.
- Entries correspond to next states.
- The start state is marked with an arrow (→)
- The accepting states are marked with a star, (*).

States\ Input	Present Inputs			
	a_1	a_2	...	a_n
S_0				
S_1				
...				
S_n				

Fig: State transition table

For a finite state machine, a state transition table is a table describing the transition function, δ , of a finite automaton. This function governs that state to which the automaton will move, given an input to the machine.

Example1: An example of a state transition table for a machine **M** together with the corresponding state diagram is given below.

State Transition Table

Input State	1	0
S_1	S_1	S_2
S_2	S_2	S_1

All the possible inputs to the machine are enumerated across the columns of the table. All the possible states are enumerated across the rows. From the state transition table given above, it is easy to see that if the machine is in $\mathbf{S_1}$ (the first row), and the next input is character **1**, the machine will stay in $\mathbf{S_1}$. If a character **0** arrives, the machine will transition to $\mathbf{S_2}$ as can be seen from the second column. In the diagram this is denoted by the arrow from $\mathbf{S_1}$ to $\mathbf{S_2}$ labeled with a **0**.For a nondeterministic finite automaton (NFA), a new input may cause the machine to be in more than one state, hence its non-determinism. This is denoted in a state transition table by a pair of curly braces *{ }* with the set of all target states between them.

State diagram from transition table

To draw a state diagram from the table. Simple steps are given below:

Step-1: Draw the circles to represent the states given.

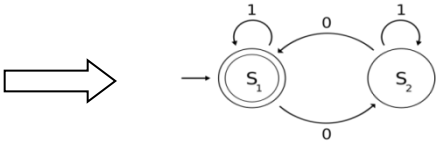
Step-2: For each of the states, scan across the corresponding row and draw an arrow to the destination state(s). There can be multiple arrows for an input character, if the automaton is NFA.

Step-3: Designate a state as the start state.

Step-4: Designate one or more states as the accept state.

Example: Construction of transition diagram from transition table.

Input State	1	0
-> *S_1	S_1	S_2
S_2	S_2	S_1



Mathematical Representation of Finite State Machine

A finite state automaton , A can be specified by the entries in the ordered quintuple $A =< Q, \Sigma, \delta, q_0, F >$,

- Q is a finite non-empty set of labels identifying the states of the machine.
- Σ is a finite vocabulary of input or data symbols.
- δ is the transition function of the machine which maps from $\delta : Q \times \Sigma \rightarrow Q$
- $q_0 \in Q$ identified the starting state of the machine and
- $F \subseteq Q$ is a set of accepting states or the final state.

1. Transition Function

The transition function of a automaton, which is also called its ***transition matrix or transition table,*** specifies the state into which the machine will move, on the basis of its current state and the word that is read. The transition function δ can be represented as a set of ordered triples of the form $< q_i, w, q_j >$ where

- $q_i \in Q$ is a label, identifying the current state
- $w \in \delta$ is the word that is read with the automaton in the q_i state and
- $q_j \in Q$ is the label of the new state into which the automaton shifts, on reading the word w.

Elements of the transition function can also be represented as $q_i w \rightarrow q_j$ or $\delta(q_i, w) = q_j$. In other words there is an arc in the transition function from state q_i to q_j.

2. Language of the Automaton

In general, a finite state machine accepts a string $n = w_1, w_2, \ldots w_i$, if there is a path in the transition diagram such that it

- begins at a start state
- ends at an accepting state
- has a sequence of labels $w_1, w_2, \ldots w_i$.

Formally, a string consisting of the n input symbols $w_1, w_2, \ldots w_i$ is accepted by the automaton A if, for each symbol $w_i \in \Sigma$ where i=1 to n, there is a transition

$$< q_{i-1}, w_i, q_i > \in \delta$$

The first symbol w_1 of the string is read, the transition $< q_0, w_1, q_1 >$ shifts the machine out of its starting state, and as the last symbol w_n is read, the last transition $< q_{n-1}, w_n, q_n >$ in the sequence shifts the machine into an accepting state. The set of all strings of input symbols, accepted by an automaton A, is called the language of the automaton and is denoted L(A).

3. Extending the transition function to strings : Is a function that takes a state q and a string w and returns the state p, that is reached when the automaton starts from q and processes w. Suppose that w=xa, where a denotes the last symbol of w. Then $\delta(q, w) = \delta(\delta(q, x), a)$

4. Design of a finite state machine

Example: Consider the finite state machine with following characteristics:

Set of states: Q={q_0,q_1,q_2,q_3}

Alphabet : where the input string comes from $\Sigma = \{a, b\}$

Initial state: where the start head starts at first element $q_0 = \{q_0\}$

Set of final state: If we stop there we accept the input $F = \{q_0, q_1, q_2\}$

Transition function δ *:* rules how to move from state to state. The following are transition rules:

$\delta(q_0, a) = q_0$ // from state q_0 and with input a, goto state q_0

$\delta(q_0, b) = q_1$ // from state q_0 and with input b, goto state q_1

$\delta(q_1, a) = q_0$ // from state q_1 and with input a, goto state q_0

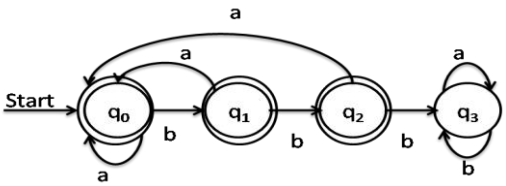
$\delta(q_1, b) = q_2$ // from state q_1 and with input b, goto state q_2

$\delta(q_2, a) = q_0$ // from state q_2 and with input a, goto state q_0

$\delta(q_2, b) = q_3$ // from state q_2 and with input b, goto state q_3

$\delta(q_3, a) = q_3$ or $\delta(q_3, b) = q_3$ // from state q_3 and with any input, goto state q_3

Transition diagram:



Transition table:

States\ Input	Present input	
	a	b
-> *q_0	q_0	q_1
*q_1	q_0	q_2
*q_2	q_0	q_3
*q_3	q_3	q_3

5. FSM Implementation

A finite state machine can be implemented with a state transition matrix.

a. In some cases a sparse matrix is implemented with linked lists or

b. a huge switch- statement for detecting the internal state and then the individual switch statements for decoding the input symbol are used.

In hardware, a FSM may be built from a programmable logic device, relays or even a mechanical cam timer combined with other elements.

Computation

Definition: Computation can be defined as finding a solution to a problem from the given inputs by means of an algorithm.

Models of computation

Model of computation means a *abstract definition of a computer*. Using a model one can analyse more easily, the intrinsic execution time or memory space of an algorithm while ignoring many implementation issues. There are many models of computation which differ in computing power & cost of various operations.

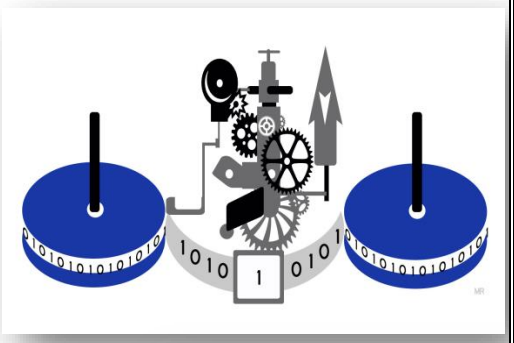
The different computational models are:

a) Serial models

- Turing machine
- Random access machine
- Primitive recursive
- Cellular automaton
- Finite state machine
- Cell probe model
- Pointer machine
- Alternation
- Alternating Turing machine
- Nondeterministic Turing machine
- Oracle Turing machine
- Probabilistic Turing machine
- Universal Turing machine
- Quantum Computation

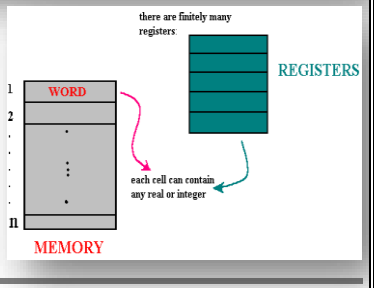
b) Parallel methods

- Multiprocessor model
- Work – depth model
- Parallel random access machine



Serial models

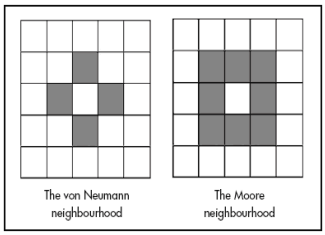
- ***Turing machine:*** It consists of a finite state machine controller, a read-write head and an unbounded sequential tape. Depending on the current state and symbol read on the tape, the machine can change its state and then move the head left or right. Turing machine stores characters on an infinitely long tape, with one square area being scanned by a read/write head at any given time.



- ***Random access machine:*** This is a model of computation whose memory consists of an unbounded sequence of registers, each of which may hold an integer. In this model, arithmetic operations are allowed to compute the address of a memory register.

- ***Primitive recursive**:* A total function which can be written, using only nested conditional (if-then-else) statements and fixed iteration (for) loops. These models use functions and function composition to operate on numbers.

- ***Cellular automaton:*** This is a two-dimensional organization of simple, finite state machine whose next state depends on their own state and the states of their eight closest neighbours. In general the machines may be arranged in meshes of higher or lower dimensions, having larger neighborhoods or arbitrarily complex processors.



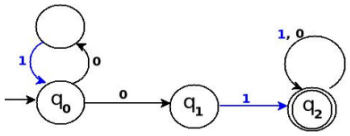
- ***Cell probe model:*** The cost of computation is measured by the total number of memory accesses to a random access memory, with cell size of log.

- ***Pointer machine**:* This is a model of computation whose memory consists of an unbounded collection of registers or records, connected by pointers. Each register may contain arbitrary amount of additional information. In this model, no arithmetic operations are needed to compute the address of a register. The only way to access the registers is by pointers.

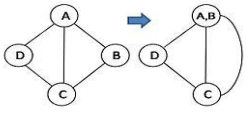
- ***Alteration:*** This a model of computation proposed by A.K.Chandra , L.Stockmeyere, and D.Kozen which has two kinds of states – AND and OR. The definition of accepting computation is adjusted accordingly.

- **Alternating Turing machine:** This is a nondeterministic Turing machine having universal states, from which the machine accepts possible moves of only that state which lead to acceptance.

- **Nondeterministic Turing machine:** This is a Turing machine which has more than one next state for some combinations of contents of the current cell and current state. An input is accepted, if any move sequence leads to acceptance.



- ***Oracle Turing machine:*** This a Turing machine with an extra oracle tape and 3 extra states $q_?$, q_y, q_n. When the machine enters $q_?$, control goes to state q_y, if the oracle tape content is in the oracle set; otherwise control goes to state q_n.



- ***Probabilistic Turing machine*:** This a Turing machine in which some transitions are random choices among finitely many alternatives.

- ***Universal Turing machine*:** This is a Turing machine that is capable of simulating any other Turing machine by encoding the latter.

- ***Quantum Computation****:* Here the computation is based on quantum mechanical effects, such as superposition or entanglement, in addition to classical digital manipulations.

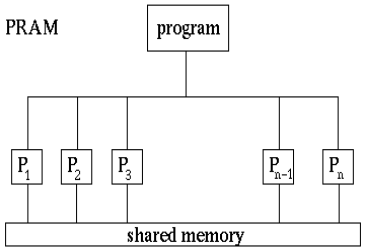
- ***Finite state machine****:* It consists of a set of states, a start state, an input alphabet and a transition function that maps input symbols and current states to a next state. Computation begins in the start state with an input string. It changes to new states, depending on the transition function. There are many variants, for example:
  - *machine associates an output with each transition. (mealy machine)*
  - *machine associates an output with each state. (Moore machine)*
  - *transitions conditioned on no input symbol (∈ - transition, or null)*
  - *one or more states, designated as accepting states (recognizer), etc.*

Parallel models

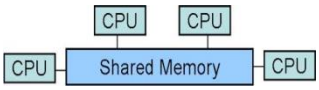
- ***Multiprocessor model:*** This is a model of parallel computation, based on the set of communicating sequential processors.

- ***Work – depth model****:* This is a model of parallel computation in which one keeps track of the total work and depth of computation, without worrying about how it maps onto a machine.

- ***Parallel Random access machine*****:** This is a shared memory model of computation, where the processors typically execute the same instruction synchronously and access to any memory location occurs in unit time.



- ***Shared memory****:* Here all the processors have the same global image of whole memory.



Automata classification

An automaton is a system which obtains, transforms, transmits and uses information to perform its functions without direct human participation. It is self-operational. The following is the classification of automata:

1. **Based on Modeling**

   Probabilistic automata*:* There is a predetermined probability of each of the next states, given the current state and input.

   Non-Probabilistic automata*:* There is a predetermined probability of next states, given the current state and input.

2. **Based on implementation**

   Deterministic automata*:* The output is uniquely determined by the input sequences. The behavior of such automata can be accurately predicted, if the transfer operator is known and given in the form of a table of a logical function. Also, we need to know the initial state and input sequence.

Statistical automata: In this automata, random output sequences are generated, given any fixed input. The amount of randomness can be set by applying a probability of any output, given the systems current state and input sequence.

Memoryless automata: It recognizes only one input at a time and produces the output based on that input. The output is not influenced by any additional inputs which arrived. The reaction time of the automaton is constant for all input signals. The internal state of such an automaton is independent of any external action.

Finite memory automata: It is a type of automata where the group of output signals generated at a given time depends not only on the signals applied at that moment and also arrived earlier. These external actions are recorded in the automaton by a variation of its internal state. The reaction is uniquely determined by the group of input signals that has arrived and by its internal state at a given time. These factors also determine the state into which the automaton goes.

Infinite memory automata: It refers to an abstract circuit of a logical automaton which, in principle, is suitable for realizing any information- processing algorithm. Turing machine belongs to this class of automata.

3. Based on processing

- ***Acceptors and recognizers***

Acceptors are those which either accept the input or do not. Recognizers are those which either recognize the input or do not. The following automata fall in the category:

a. Deterministic finite state machine
b. Nondeterministic finite state machine
c. Pushdown automata (PDA)

- ***Transducers***

Transducers generate output from the given input. The following automata are in this category:

a. Mealy machine
b. Moore machine

Mealy machine: It is a finite state machine; the outputs are determined by current state & the input.

Moore machine: It is a finite state machine; the outputs are determined by the current state alone.

4. Based on the input

Tree automata*:* It is a type of finite state machine. It deals with tree structures, rather than the strings, like more conventional finite state machines.

Linear automata: Finite automata that operate on languages of finite words.

Rabin/Buchi automata: finite automata that operate on languages of infinite words.

5. Based on applications

Cellular Automaton: It consists of a line of cells, each coloured either black or white. At every step, there is a definite rule that determined the colour of a given cell from the colour of its immediate left and right neighbours on the step before. These automata are used for making pretty pictures and animations.

Geographic automata: A system that is used to simulate the behavior and distribution of objects in space such as householders, pedestrians, vehicles, shops, roads, land parcels, sidewalks etc., with respect to their properties and their locations, is a geographic automata system.

Deterministic Finite Automata (DFA) A deterministic finite automaton is a finite state machine where for each pair of states and input symbol there is a unique next state.

A finite automation is a five-tuple structure $M = (Q, \Sigma, q_0, \delta, F)$,where

Q is a finite set of states in which the finite automation can exist,

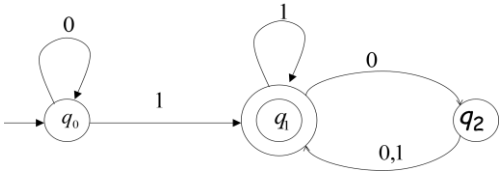
Σ is a set of input symbols that can be fed to the finite automation,

q_0 is the starting state of the finite automation,

δ is the set of transition functions that determine the next state, given the current state of finite automation and the current input symbol, and

$F \subseteq Q$ is the set of final states (also referred to as acceptor states).

Figure 3.1 shows an example of a finite automation M.



Transistion table:

States\Input	1	0
q_0	q_1	q_0
q_1	q_1	q_2
q_2	q_1	q_1

In the finite automation M

1. Q = {q_0,q_1,q_2}

At any instant, the finite automation M can exist in any one of these states. The states are represented by the lowercase alphabets q or s with numeric subscripts to differentiate among various states. The states can also be represented by capital alphabets.

2. ∑ = {0,1}

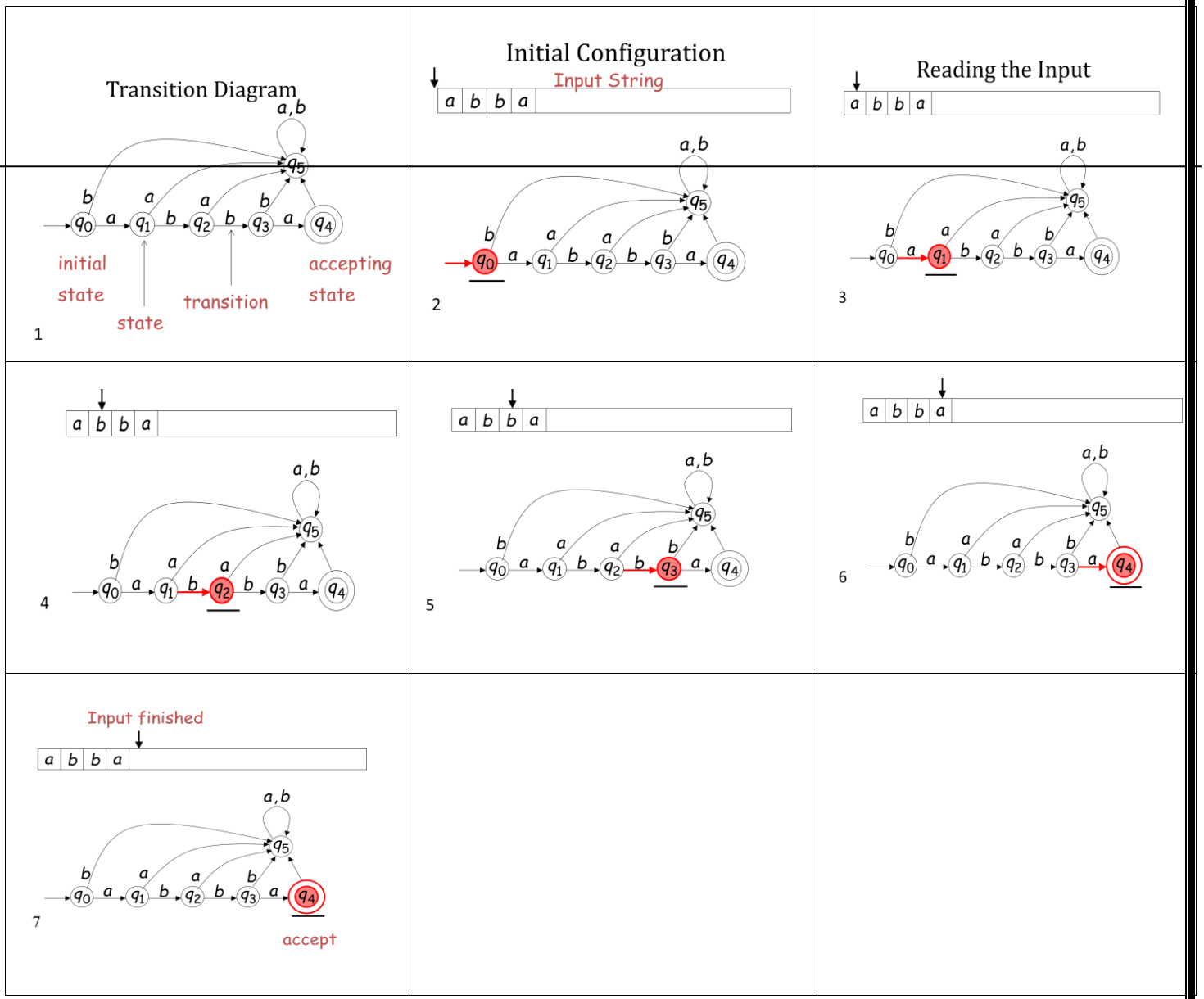
∑ represents the character set of the language L to be recognized by the finite automation M.

3. The state q_0 is the starting state of M in the beginning. The finite automation reaches this state without consuming any character. To reach any other state, the finite automation consumes a character. A state with an arrow from free space (not coming from any other state) is designated as an initial state. Normally, there is only one starting state in a finite automation. However, there is no such restriction and one can construct a finite automation with more than one initial state if necessary.

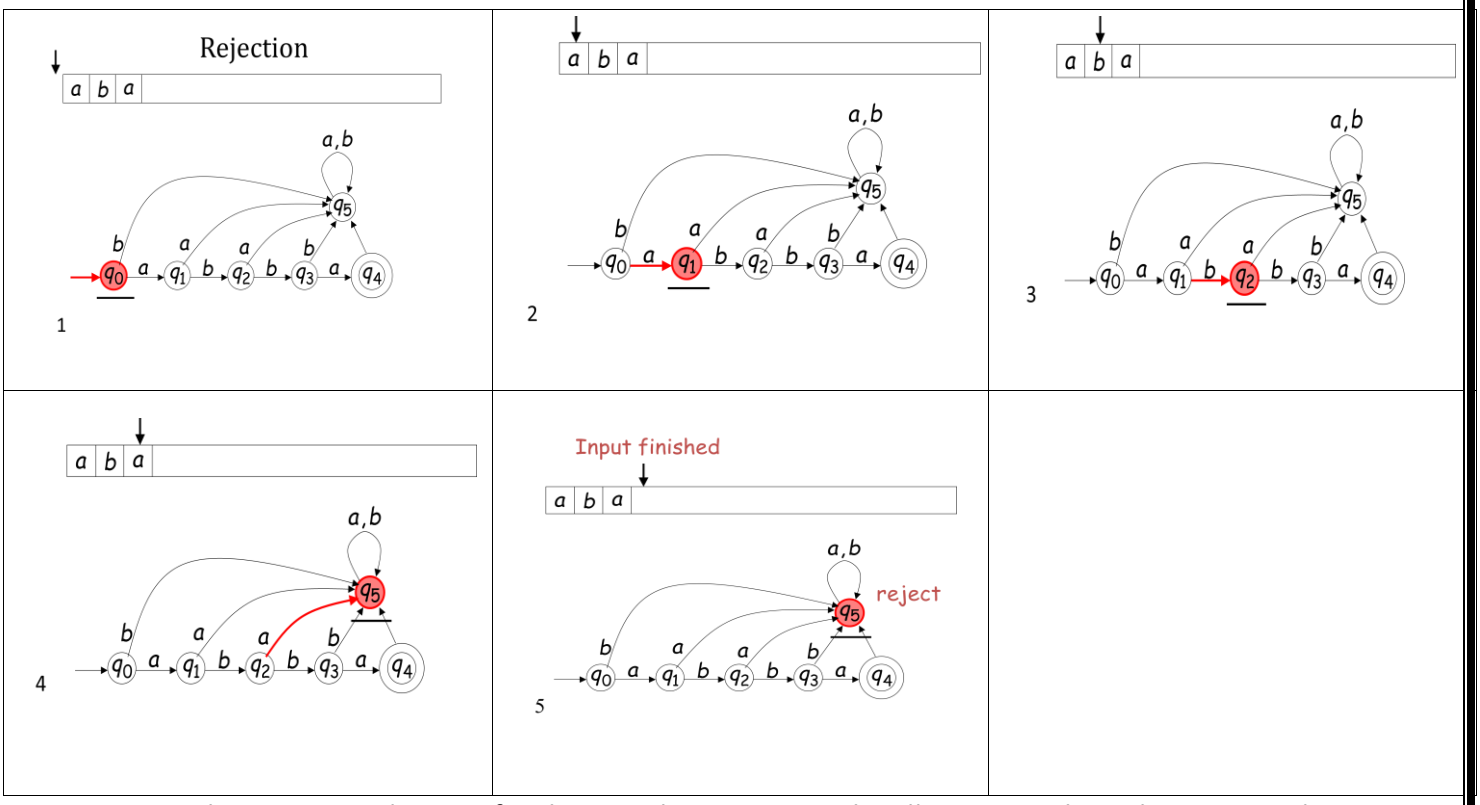
4. The set of transition functions δ is the functional description of a finite automation. It shows how the automation will behave, given its current state and the input symbol. δ Can be expressed by a Transition diagram.

Operation of DFA: Initially the DFA is assumed to be in the initial state q0 with its read head on the left most symbol of the input string. During each move of DFA, the read head moves one position to the right. Thus, each move consumes one input symbol. When the end of string is reached, the string is accepted if DFA is in one of its final states, else rejected.

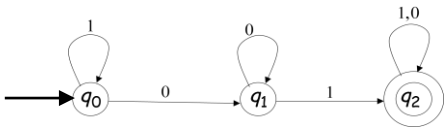
Example: (DFA that accepts the input string abba)



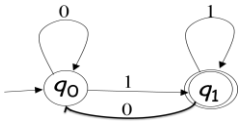
Example: (DFA that rejects the input string aba)



Eg :- 1. Draw the transition diagram for the DFA that accepting the all strings with a substring 0 and 1.



2. Design a DFA that accepts all strings ends with 1, L={w/w ends with 1}.



Language accepted by DFA: The language accepted by a DFA, M = (Q,Σ,q_0,δ,F) is the set of all strings on ∑ accepted by M i.e,

$$L(M)=\{W\in\Sigma^* \mid \delta(q_0,W)\in F$$

The language accepted by a DFA, M = (Q,Σ,q_0,δ,F)

$$L(M)=\{W\in\Sigma^* \mid \delta(q_0,W)\notin F$$

Extended transition function for DFA : It describes what happens when it is starting any state and follow any sequence of inputs. For DFA, M = (Q,Σ,q_0,δ,F) the function δ is extended as $\delta^*:Q\times\Sigma^*\to Q$ and

defined recursively as follows:
a) For any state q of Q . This means that DFA stays in the same state q when it reads an empty string at q.

$$\delta(q,\in)=q$$

b) For any state q of Q, any string $x\in\Sigma^*$ with a as the last symbol of x and $a\in\Sigma$ then $\delta^*(q,xa)=\delta(\delta^*(q,x),a)$

Example: 1) Let us design a DFA to accept a even no. of 0's and 1's.

L = {w/w has both an even no. of 1's and 0's}
$A=(\{q_0,q_1,q_2,q_3\},\{0,1\},\delta,q_0,\{q_0\})$
Consider a string =110101

$$\delta^*(q_0,\in)=q_0$$
$$\delta^*(q_0,1)=\delta(\delta^*(q_0,\in),1)=\delta(q_0,1)=q_1$$
$$\delta^*(q_0,11)=\delta(\delta^*(q_0,1),1)=\delta(q_1,1)=q_0$$
$$\delta^*(q_0,110)=\delta(\delta^*(q_0,11),0)=\delta(q_0,0)=q_2$$
$$\delta^*(q_0,1101)=\delta(\delta^*(q_0,110),1)=\delta(q_2,1)=q_3$$
$$\delta^*(q_0,11010)=\delta(\delta^*(q_0,1101),0)=\delta(q_3,1)=q_1$$
$$\delta^*(q_0,110101)=\delta(\delta^*(q_0,11010),1)=\delta(q_1,1)=q_0$$

Design of DFA:-The basic design strategy for DFA is as follows:

a) Understand the language properties for which the DFA has to be designed.
b) Determine the state set required.
c) Identify the initial, accepting and dead state of DFA.
d) For each state, decide on the transition to be made for each character of the input string.
e) Obtain the transition table and diagram for DFA.
f) Test the DFA obtained on short strings.

Example: To design a DFA that accepts set all strings that contain 0's or 1's and end in 00.
Solution: We are required to design a DFA for the regular expression r = $(0+1)^*00$…. In other words the DFA should be designed to accept the language of r,

L(M) = {00,100,1100,0000,111000,…} where M is a DFA.

Consider, ∑ = {0,1}
Q = $\{q_0,q_1,q_2\}$ q_0 = initial state q_2 = final state and $\delta: Q\times\Sigma\to Q$ is given by

Transition diagram:

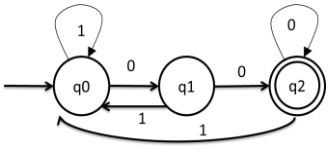


Fig: State Transition to represent
$L(M)=\{w\in\Sigma^* \mid w \text{ ends with } 00\}$

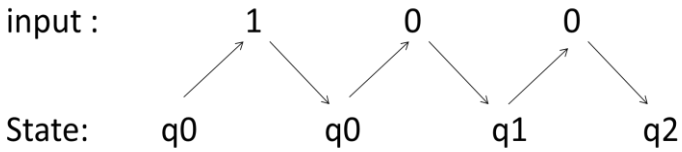
Transition table:

States\Input	0	1
→q_0	q_1	q_0
q_1	q_2	q_0
*q_2	q_2	q_0

DFA action for the input string

1. To show that the string 100 is accepted by DFA:

i) Using sequence state diagram

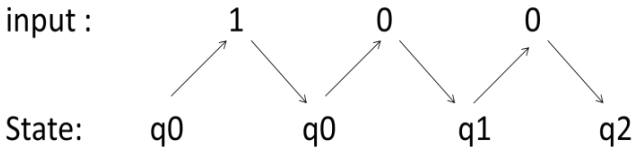


ii) Using extended transition function

$$\delta(q_0,1) = q_0$$
$$\delta(q_0,10) = \delta(\delta(q_0,1),0) = \delta(q_0,0) = q_1$$
$$\delta(q_0,100) = \delta(\delta(q_0,10),0) = \delta(q_1,0) = q_2$$

2. To show that the string 010 is rejected by DFA:

a. Using sequence state diagram



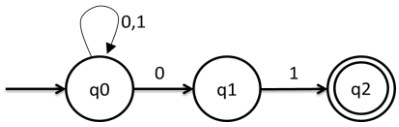
b. Using extended transition function

$$\delta(q_0,0) = q_1$$
$$\delta(q_0,01) = \delta(\delta(q_0,0),1) = \delta(q_1,1) = q_0$$
$$\delta(q_0,010) = \delta(\delta(q_0,01),0) = \delta(q_0,0) = q_1$$

Non-Deterministic Finite Automata (NFA):- It has the power to be in several states at once. Let the DFA, an NFA has a finite set of states, a finite set of input symbols, one start and a set of accepting states. It also has a transition function, which we shall commonly called δ.

For the NFA, δ is a function that takes a state and input symbols as arguments but returns a set of states (rather than returning exactly one state as the DFA must).

DFA that it accepts all strings ending in 01. The transition diagram and transition table is.

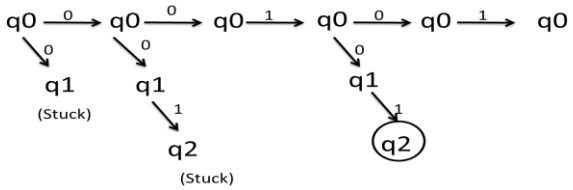


State\ Input	0	1
→q_0	{q_0, q_1}	{q_0}
q_1	ϕ	{q_2}
*q_2	ϕ	ϕ

With respect to Finite Automata,

- On input 0 in state q_0, the next state may be either of the two states.
- There is no next state on input 0 in state q_1.
- There is no next state on input 0 and 1 in state q_2.

The processing of input string 00101 is



For the first input symbols 0 of the string, there exists 2 choices: - whether to stay at the same state q_0, or move to the next state q_1, with the initial state of q_0 i.e, any input symbol does not result in the transition to a unique state, but results in a chain of states. Thus, the above modified FA has multiple paths corresponding to the input 00101.

Path-1: $q_0 \xrightarrow{0} q_0 \xrightarrow{0} q_0 \xrightarrow{1} q_0 \xrightarrow{0} q_0 \xrightarrow{1} q_0$ $\Rightarrow$ stuck at q_0 itself

Path-2: $q_0 \xrightarrow{0} q_1$ $\Rightarrow$ stuck at q_1

Path-3: $q_0 \xrightarrow{0} q_0 \xrightarrow{0} q_1 \xrightarrow{1} q_2$ $\Rightarrow$ stuck at q_2

Path-4: $q_0 \xrightarrow{0} q_0 \xrightarrow{0} q_0 \xrightarrow{1} q_0 \xrightarrow{0} q_1 \xrightarrow{1} (q_2)$

$\Rightarrow$ starts at the initial state and ends at the final state.

Thus, the string is accepted by the machine.

This machine had thus allowed several states as a result of the processing of an input symbol, this is called **non-determinism**. If from any state we can reach several states or a null state, then the finite automaton becomes nondeterministic in nature.

> *If the basic finite automata model is modified in such a way, that from a state on an input symbol; one or more transitions / choices are permitted, then the corresponding finite automata is called as "non-deterministic finite automata" (NFA).*

Elements of NFA: A non-deterministic finite automata exhibits the following 5 characteristics:($Q, \Sigma, q_0, \delta, F$)

Q is a finite set of states in which the finite automation can exist,

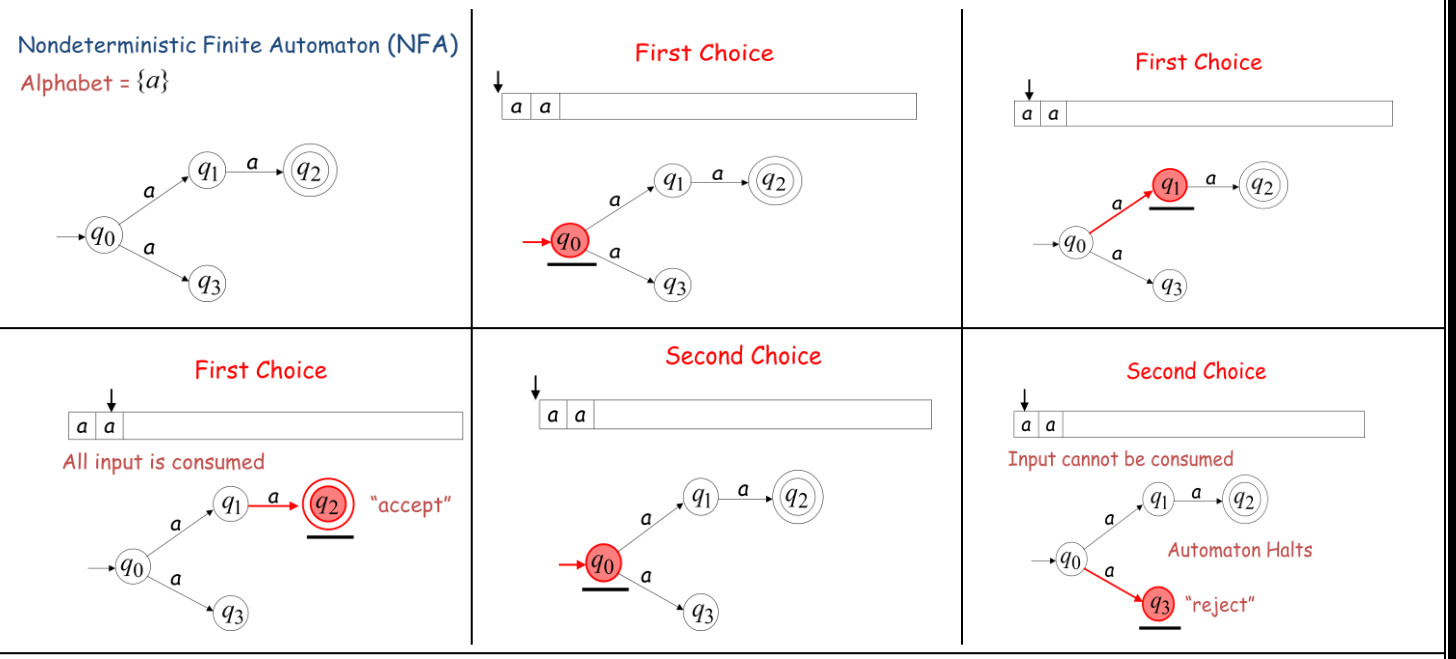
Σ is a set of input symbols that can be fed to the finite automation,

q_0 is the starting state of the finite automation,

δ transition function which specifies the nature of transition at a given state.

$F \subseteq Q$ is the set of final states (also referred to as acceptor states).

How NFA Operates: NFA corresponds to a kind of parallelism in automata. It consists of the same basic model or components as DFA i.e, input tape, read head and finite control. However, when transition function allows more than one next state, for a given state and input, it is required to keep an independent internal state for each of the alternative. (There must be a constantly growing and shrinking set of automata, processing the same input synchronously).



Extended transition function for NFA :- There exists more than one path corresponding to a given input x in $\sum^*$, hence it is required to test the multiple paths corresponding to x in order to decide whether x is accepted by NFA or not. i.e, for NFA to accept x, atleast one path corresponding to x is required in NFA. This path should start in the initial state and end in final states. Where as in case of DFA, since there exists exactly one path corresponding to x in $\sum^*$, it is enough to test whether or not that path starts in the initial state and ends in one of the final states, in order to decide whether x is accepted by a DFA or not.

Definition of δ **:** For a state q and a string x, δ (q,x) is the set of states that NFA can reach when it reads the string x, starting at the state q. In general, NFA nondeterministically goes through a number of states from the state q as it reads the symbols in the string x. Thus for an NFA, M = ($Q, \Sigma, q_0, \delta, F$)

the function is extended as:

$$\delta : Q \times \Sigma^* \rightarrow P(Q)$$

and is defined recursively as follows:

a) For any state q of Q,

$$\boxed{\delta(q, \in) = \{q\}}$$

This means that an NFA stays in the same state q when it reads an empty string at state q.

b) For any state q of Q and any string x $\in \Sigma^*$, with 'a' as the last symbol of x and a $\in \Sigma$, then

$$\delta(q, xa) = \bigcup_{\text{for every p inq}} \delta(p, a) \text{ or } \delta(q, xa) = \bigcup_{p \in \delta(q,x)} \delta(p, a)$$

This means that the set of states that can be reached by NFA after reading string xa starting at state q, is the set of states it can reach by reading the symbol a after reading the string x, starting at state q.

Example: Consider the automaton of table, with input w = 00101

State\ Input	0	1
→q_0	{q_0, q_1}	{q_0}
q_1	ϕ	{q_2}
*q_2	ϕ	ϕ

$$\delta^*(q_0, \in) = q_0$$
$$\delta^*(q_0, 0) = \{q_0, q_1\}$$
$$\delta^*(q_0, 00) = \delta(\delta^*(q_0, 0), 0) = \delta(\{q_0, q_1\}, 0) = \delta(q_0, 0) \cup \delta(q_1, 0) = \{q_0, q_1\} \cup \phi = \{q_0, q_1\}$$
$$\delta^*(q_0, 001) = \delta(\delta^*(q_0, 00), 1) = \delta(\{q_0, q_1\}, 1) = \delta(q_0, 1) \cup \delta(q_1, 1) = \{q_0\} \cup \{q_2\} = \{q_0, q_2\}$$
$$\delta^*(q_0, 0010) = \delta(\delta^*(q_0, 001), 0) = \delta(\{q_0, q_2\}, 0) = \delta(q_0, 0) \cup \delta(q_2, 0) = \{q_0, q_1\} \cup \phi = \{q_0, q_1\}$$
$$\delta^*(q_0, 00101) = \delta(\delta^*(q_0, 00101), 1) = \delta(\{q_0, q_1\}, 1) = \delta(q_0, 1) \cup \delta(q_1, 1) = \{q_0\} \cup \{q_2\} = \{q_0, q_2\}$$

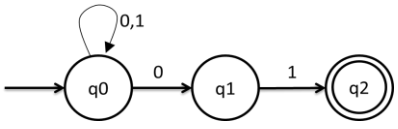
Language accepted by NFA:- The language accepted by NFA, M for a string x is defined as:

$$L(M) = \{x \mid \delta(q_0, x) = p, \text{ where p contains atleast one member of F or}$$

$$L(M) = \{x \mid \delta(q_0, x) \cap F \neq \{\phi\}\}$$

This means that the NFA accepts the string x, iff it can reach an accepting state by reading x starting at the initial state.

Example: For the automata with w = 00101, we have $\delta(q_0, 00101) = \{q_0, q_2\}$, where P = $\{q_0, q_2\}$



Since P contains q_2 which is the final state, here 00101 is accepted.

$$\delta(q_0, 00101) = \{q_0, q_2\} \cap F$$

$= \{q_0, q_2\} \cap \{q_2\}$
$= \{q_2\}$
$\neq \{\phi\}$

Design of NFA's:- The basic design strategy for an NFA is as follows:

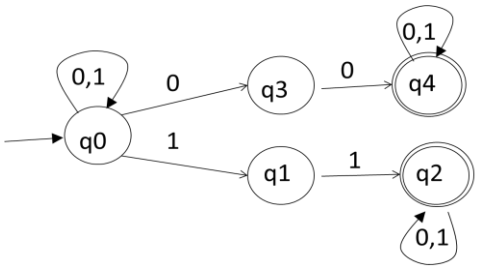
a) Understand the language properties for which the NFA has to be designed.
b) Determine the alphabet and state set required.
c) Identify the initial, accepting and dead states of NFA.
d) Obtain the transitions to be made for each state on each character of the input string.
e) Draw the transition table and diagram for NFA.
f) Test the NFA obtained on short strings.

Example: *Design a NFA that accepts set of all strings over {0,1} that have atleast 2 consecutive 0's or 1's.*

To design a NFA for the regular expression r=(0+1)*(00+11)(0+1)*. In other words, NFA, M should be designed to accept the language of r. i.e, L(M) = {00,11,0110,1001,10110101,…}

Consider, $\Sigma = \{0,1\}, Q = \{q_0, q_1, q_2, q_3, q_4\}$, q_0 = initial state, $\{q_2, q_4\}$ = final state.

Transition diagram



Transition Table

State\Input	0	1
→q_0	{q_0, q_3}	{q_0, q_1}
q_1	ϕ	{q_2}
*q_2	{q_2}	{q_2}
q3	{q_4}	ϕ
*q4	{q_4}	{q_4}

NFA action for the input string := 0100011

1) *Using extended transition function:*

$\delta(q_0, \in) = \{q_0\}, \quad \delta(q_0, 0) = \{q_0, q_3\}, \quad \delta(q_0, 0) = \{q_0, q_1\}$
$\delta(q_0, 01) = \delta(\delta(q_0, 0), 1) = \delta(q_0, 1) \cup \delta(q_3, 1) = \{q_0, q_1\} \cup \phi = \{q_0, q_1\}$
$\delta(q_0, 010) = \delta(\delta(q_0, 01), 0) = \delta(q_0, 0) \cup \delta(q_1, 0) = \{q_0, q_3\} \cup \phi = \{q_0, q_3\}$
$\delta(q_0, 0100) = \delta(\delta(q_0, 010), 0) = \delta(q_0, 0) \cup \delta(q_3, 0) = \{q_0, q_3\} \cup \{q_4\} = \{q_0, q_3, q_4\}$
$\delta(q_0, 01000) = \delta(\delta(q_0, 0100), 0) = \delta(q_0, 0) \cup \delta(q_3, 0) \cup \delta(q_4, 0) = \{q_0, q_3\} \cup \{q_4\} \cup \{q_4\} = \{q_0, q_3, q_4\}$
$\delta(q_0, 010001) = \delta(\delta(q_0, 01000), 1) = \delta(q_0, 1) \cup \delta(q_3, 1) \cup \delta(q_4, 1) = \{q_0, q_1\} \cup \{\phi\} \cup \{q_4\} = \{q_0, q_1, q_4\}$

$$\delta(q_0,0100011)=\delta(\delta(q_0,010001),1)=\delta(q_0,1)\cup\delta(q_1,1)\cup\delta(q_4,1)=\{q_0,q_1\}\cup\{q_2\}\cup\{q_4\}=\{q_0,q_1,q_2,q_4\}$$

Let P= $\{q_0,q_1,q_2,q_4\}$

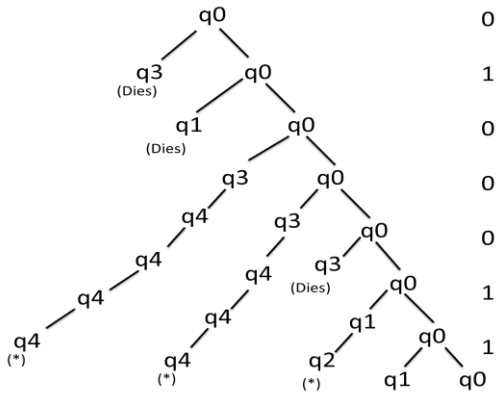
Since P contains q_2,q_4 (which are final states), clearly 0100011 is accepted by L(M).

i.e, $\delta(q_0,0100011)=\{q_0,q_1,q_2,q_4\}\cap F$

$=\{q_0,q_1,q_2,q_4\}\cap\{q_2,q_4\}$

$=\{q_2,q_4\}=\{\phi\}$ $\rightarrow$ 0100011 is accepted.

2) *Using tree state diagram:*



String is accepted.

NFA action for the input string := 010

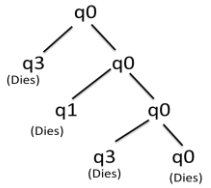
1) *Using extended transition function:*

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

$\delta(q_0,01)=\delta(\delta(q_0,0),1)=\delta(q_0,1)\cup\delta(q_3,1)=\{q_0,q_1\}\cup\phi=\{q_0,q_1\}$

$\delta(q_0,010)=\delta(\delta(q_0,01),0)=\delta(q_0,0)\cup\delta(q_1,0)=\{q_0,q_3\}\cup\phi=\{q_0,q_3\}$ The string is rejected.

2) *Using tree state diagram:*



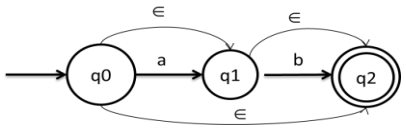
String rejected.

Non-Deterministic Automata with $\in$-Moves : we can extend NFA by introducing a feature that allows us to make a transition on the empty string such transitions are called as epilson transitions.

- The $\in$ transition refers to a transition from one state to another without reading an input symbol.

- $\in$-transition can be inserted between any states there is also a conversion algorithm from an NFA with $\in$ transition to NFA without $\in$ transition.
- This capability does not expand the class of languages that can be accepted by finite automata but gives us some programming convenience.

Example: NFA with transitions between q_0 to q_1 and q_1 to q_2.

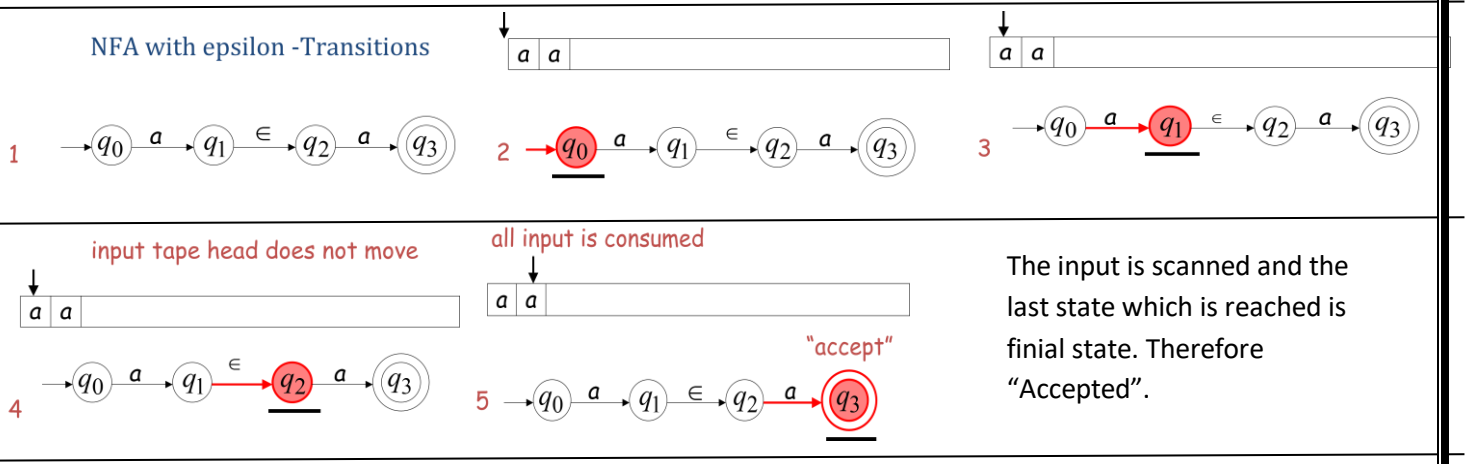


$\in$-transition refers to the fact that "transition takes place without reading any symbols in the input.

Elements of NFA-$\in$: The five characteristics are $(Q, \Sigma, q_0, \delta, F)$

Q is a finite set of states in which the finite automation can exist,
Σ is a set of input symbols that can be fed to the finite automation,
q_0 is the starting state of the finite automation,
δ which specifies the nature of transition at a state due to the input symbols including $\in$.
$F \subseteq Q$ is the set of final states (also referred to as acceptor states).

$$\delta : Q \times (\Sigma \cup \in) \rightarrow P(Q)$$



Transitions of NFA -$\in$:- There exists 2 transitions:

1) $\in$- *transitions:* Transitions that take place without reading any input symbols are called $\in$-transitions.
   If in a state q a transition to a state q′ is possible, then whenever the automaton reaches state q in a computation, it can immediately proceed to state q′ without reading any further input.
2) *non $\in$-transitions:* Transitions that take place with the reading of input symbols are called non $\in$-transitions.

Acceptance of string by NFA - ∈:- A string is said to be accepted by the NFA-∈, if atleast one path exists that starts at the initial state and ends in one of the final states. The path formed contains states with transitions from ∈-transitions or non ∈-transitions. In order to identify all states in the actual path, that are formed with ∈ transitions, the concept of '∈– closure' is used.

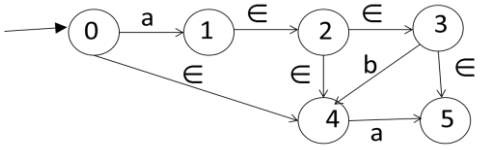
∈-Closure: *∈ - closure for a state is the set of states reachable from the state, without reading any symbol.*

∈- closure of a given state 'q' is defined as the set of all states of automata, that can be reached from q on a path labeled by ∈.

In general, if P is the power set of Q then,

a) P $\subseteq$ ∈- closure(P)
b) For q of Q, if q∈ ∈– closure(P) then $\delta(q,\in) \subseteq \in -closure$

Example:



a) To compute ∈-closure({2})

$\{2\} \subseteq \in -closure(\{2\})$; $[\because \delta(2,\in) = 2]$

$\{2,3,4\} \subseteq \in -closure(\{2\})$; $[\because \delta(2,\in) = \{3,4\}]$

$\{2,3,4,5\} \subseteq \in -closure(\{2\})$; $[\because \delta(3,\in) = \{5\}$ *and* $\delta(4,\in) = \{\phi\}]$

further $\delta(5,\in) = \{\phi\}$.

No new members / states can now be added.

Hence the process of generating ∈-closure terminated and ∈-closure({2})={2,3,4,5}.

This means that from the state {2}, the set of states that can be reached without any input symbol is {2,3,4,5}.

b) To compute ∈-closure({1})

$\{1\} \subseteq \in -closure(\{1\})$; $[\because \delta(1,\in) = \{1\}]$

$\{1,2\} \subseteq \in -closure(\{1\})$ $[\because \delta(1,\in) = \{2\}]$

$\{1,2,3,4\} \subseteq \in -closure(\{1\})$ $[\because \delta(2,\in) = \{3,4\}]$

$\{1,2,3,4,5\} \subseteq \in -closure(\{1\})$ $[\because \delta(3,\in) = \{5\}$ *and* $\delta(4,\in) = \{\phi\}]$

$\{1,2,3,4,5\} \subseteq \in -closure(\{1\})$ $[\because \delta(5,\in) = \{\phi\}]$

Hence the process of generating ∈-closure terminated and ∈-closure({1}) = {1,2,3,4,5}.

Extended transition function for NFA-∈:- For a state q and string x, $\delta(q,x)$ is the set of states of NFA-∈ which it can reach when it reads the string x, starting at state q. In other words, non deterministically, it goes through a number of states from the state q as it reads the symbol in the string x. Thus for an NFA- ∈ the transition function is extended as: $\delta : Q \times \Sigma^* \rightarrow P(Q)$

a) For any state q of Q, $\delta(q, \in) = \in -closure(\{q\})$

The determination of the set of all states that are reachable from state q with $\in$ as input symbols

b) For any state q of Q, any string x $\in \Sigma^*$ with 'a' as the last symbol of x and a $\in \Sigma$,

$$\delta(q, xa) = \in -closure(\cup_{p \in \delta(q,x)}, \delta(p, a))$$

Language Accepted by NFA-$\in$: The language accepted by NFA- $\in$, M for a string x is defined as:

$L(M) = \{x \mid \delta(q_0, x) = p,$ where p contains atleast one member of F or

$$L(M) = \{x \mid \delta(q_0, x) \cap F \neq \{\phi\}\}$$

This means that the NFA accepts the string x, iff it can reach an accepting state by reading x starting at the initial state.

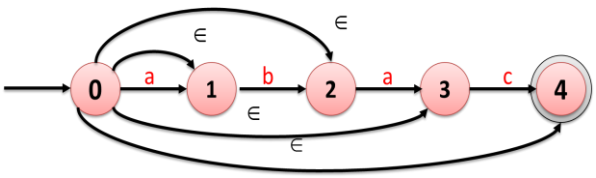
Design of NFA-$\in$:- The basic design strategy for an NFA is as follows:

a) Understand the language properties for which the NFA has to be designed.
b) Determine the alphabet and state set required.
c) Identify the initial, accepting and dead states of NFA.
d) Identify all the $\in$ - transitions.
e) Obtain the non-$\in$ - transitions for each state on each character of the input string.
f) Draw the transition table and diagram for NFA.
g) Compute $\in$ - closure for each state.
h) Test the NFA - $\in$ obtained on short strings.

Example: *Design a NFA- $\in$ that accepts the strings abac and all its suffixes.*

To design a NFA for the regular expression r = c+ac+bac+abac. In other words, NFA- $\in$, M should be designed to accept the language of r. i.e, L(M) = {$\in$,c,ac,bac,abac}

Consider, $\Sigma = \{a, b, c\}$, $Q = \{0,1,2,3,4\}$, q_0 = initial state, {4} = final state.



NFA- $\in$ action for the input string "ac":

a) To compute $\in$ - Closure
   - $\in -closure(\{0\})$: $\{0\} \subseteq \in -closure(\{0\})$, $[\because \delta(0, \in) = \{0\}]$
     $\{0,1\} \subseteq \in -closure(\{0\})$ $[\because \delta(0, \in) = \{1\}]$

$\{0,1,2\} \subseteq \in -closure(\{0\})$ $[\because \delta(0,\in) = \{2\}]$
$\{0,1,2,3\} \subseteq \in -closure(\{0\})$ $[\because \delta(0,\in) = \{3\}]$
$\{0,1,2,3,4\} \subseteq \in -closure(\{0\})$ $[\because \delta(0,\in) = \{4\}]$

Further, $\delta(1,\in) = \delta(2,\in) = \delta(3,\in) = \delta(4,\in) = \phi$. Hence process terminates and $\in -closure(\{0\}) : \{0,1,2,3,4\}$

- $\in -closure(\{1\}) :$ $\{1\} \subseteq \in -closure(\{1\}),$ $[\because \delta(1,\in) = \{1\}]$
  $\{1\} \subseteq \in -closure(\{1\}),$ $[\because \delta(1,\in) = \{\phi\}]$
- $\in -closure(\{2\}) :$ $\{2\} \subseteq \in -closure(\{2\}),$ $[\because \delta(2,\in) = \{2\}]$
  $\{2\} \subseteq \in -closure(\{2\}),$ $[\because \delta(2,\in) = \{\phi\}]$
- $\in -closure(\{3\}) :$ $\{3\} \subseteq \in -closure(\{3\}),$ $[\because \delta(3,\in) = \{3\}]$
  $\{3\} \subseteq \in -closure(\{3\}),$ $[\because \delta(3,\in) = \{\phi\}]$
- $\in -closure(\{4\}) :$ $\{4\} \subseteq \in -closure(\{4\}),$ $[\because \delta(4,\in) = \{4\}]$
  $\{4\} \subseteq \in -closure(\{4\}),$ $[\because \delta(4,\in) = \{\phi\}]$

b) To compute δ

$$\delta(\in -closure(\{0\}), ac) = \delta(\in -closure(\delta(\in -closure(\{0\}), a), c))$$

$$\begin{aligned}\delta(\in -closure(\{0\}), a) &= \delta(\{0,1,2,3,4\}, a)\\ &= \delta(1,a) \cup \delta(2,a) \cup \delta(3,a) \cup \delta(4,a)\\ &= \{1\} \cup \phi \cup \{3\} \cup \phi\\ &= \{1,3\}\end{aligned}$$

$$\begin{aligned}\delta(\in -closure(\{0\}), ac) &= \delta(\in -closure(\delta(\in -closure(\{0\}), a), c))\\ &= \delta(\in -closure(\{1,3\}, c))\\ &= \delta(1,c) \cup \delta(3,c)\\ &= \phi \cup \{4\}\end{aligned}$$

$$\begin{aligned}\in -closure(\delta(\in -closure(\{0\}), ac)) &= \in -closure(4)\\ &= \{4\} \cap F\\ &= \{4\} \cap \{4\}\\ &= \{4\} \neq \phi\end{aligned}$$

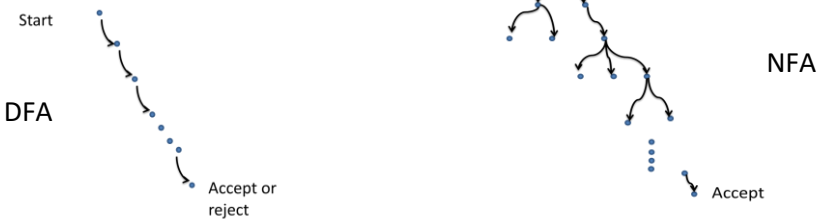
"ACCEPTED"

3.8 Advantages of NFA:

- NFA can be smaller.
- Easy to construct than DFA that accepts the language.
- It is useful for providing some theorems
- It gives good introduction to non-determinism in more powerful computation models.
- Non-determinism is a kind of parallel computation where several processes can be running concurrently.

3.9 NFA Versus DFA:

- A DFA is a special case of NFA
  - In a DFA, at every state q for every symbol 'a' , has a unique a-transition.
  - At any state, an NFA may have multiple a-transitions or none.
  - In a DFA, transition arrows are labelled by symbols from ∑
  - In an NFA, transition are labelled by symbols from $\sum \cup \{\in\}$
    i.e, an NFA may have $\in$ - transitions.
- NFAs are very convenient for demonstrating the languages to be regular. Often it is much easier to provide an NFA for a language than a DFA.
- Space and time taken to recognize regular expressions:
  - The determinism of a DFA is important when implementing a program for recognizing the regular language.
  - NFA are not well suited to language recognition.
  - NFA are most compact (in space), but take time to backtrack all choices.
  - DFA takes more space, but saves time.
- Transition function
  - For DFA, $\delta : Q \times \Sigma \to Q$
  - For NFA, $\delta : Q \times (\Sigma \cup \in) \to P(Q)$
- DFA consists of unidirectional tree structure for acceptance & NFA consists of Multidirectional tree structure for acceptance.



EQUIVALENT AUTOMATA

3.10 EQUIVALENT FINITE-STATE AUTOMATA:

The set of words accepted by a finite state machine M, is the language accepted by M is denoted by L(M).

➨ *Definition:* Two finite automata M1 and M2 (possibly of different types) are equivalent, if and only if
L(M1) = L(M2) i.e, M1 and M2 accept or recognize the same language.

The equivalences of finite automata :

- Equivalence of NFA and DFA
- Equivalence of NFA- $\in$ and DFA
- Equivalence of NFA - $\in$ and NFA

EQUIVALENCE OF NFA/NFA- AND DFA

Definition: Two finite automata N and D are said to be equivalent, if L(N) = L(D)

Where D represents Deterministic Finite Automata.

N represents Non-deterministic Finite Automata.

- That is N and D accept the same language,
  - any language accepted by D can also be accepted by some N,
  - any language accepted by N can also be accepted by some D.

- For any language described by some N, there is a D,
- i.e, for every N, it is easy to construct an equivalent D that accepts the same language.
- In worst case,
  - DFA has 2^n states
  - NFA has 'n' states

Theorem I: *A language L is accepted by some NFA, if and only if, it is accepted by some DFA.*

Proof:

In order to prove this, let us compare the definitions of N and D.

Definition 'D': A DFA, D is defined by 5 - tuple $(Q', \Sigma, \delta', q_0', F')$

Q' is a finite set of states in which the finite automation can exist,

Σ is a set of input symbols that can be fed to the finite automation,

q_0' is the starting state of the finite automation,

δ' Transition function $\delta' : Q' \times \Sigma \to Q'$

$F' \subseteq Q'$ is the set of final states (also referred to as acceptor states).

Definition 'N': A DFA, D is defined by 5 - tuple $(Q, \Sigma, \delta, q_0, F')$

Q is a finite set of states in which the finite automation can exist,

Σ is a set of input symbols that can be fed to the finite automation,

q_0 is the starting state of the finite automation,

δ Transition function $\delta : Q \times \Sigma \to 2^Q$

$F \subseteq Q$ is the set of final states (also referred to as acceptor states).

From the above definitions, it follows that every DFA is also an NFA, which implies that if $W \in L(D)$, then $W \in L(N)$.

Conversion algorithm for NFA/ NFA-$\in$ to DFA:

a) Construct the start state {q0}, consisting of q0 and all the states of NFA that can be reached from q0 by one or several $\in$ - transitions.

Mark {q0} as "unfinished".

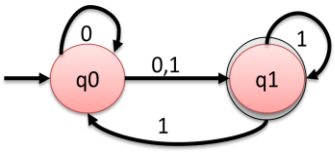
b) While there are "unfinished states"

- Take an "unfinished" state s.
- For each a $\in \sum$, Let $\delta(s,a) = \cup_{q \in s} \{t \mid q \xrightarrow{a} t\}$
  - If $\delta(s,a)$ is neither "finished" nor "unfinished", then mark as "unfinished"

- Mark s as "finished"

c) Mark all states that contain a final state from N as the final state of D.

Example: Construct a DFA, equivalent to the NFA given in fig



Solution: The transition table of the above NFA

States\ Input	0	1
q0	{q0,q1}	{q1}
q1	ϕ	{q0,q1}

No. of states in NFA = 2

No. of states in DFA = $2^2 = 4$

Consider,

q0 = {q0} ⟶

δ	0	1
{q0}		

⟸ Unfinished

a) Process {q0}

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

δ	0	1
{q0}	**{q0,q1}**	**{q1}**
{q0,q1}		
{q1}		

⟸ Finished

Unfinished

b) Process $\{q_0, q_1\}$

$\delta(\{q_0,q_1\},0) = \delta(q_0,0) \cup \delta(q_1,0)$
$\delta(\{q_0,q_1\},1) = \delta(q_0,1) \cup \delta(q_1,1)$ ⟶

δ	0	1
{q0}	**{q0,q1}**	**{q1}**
{q0,q1}	**{q0,q1}**	**{q0,q1}**
{q1}		

Finished

⟸ Unfinished

c) Process $\{q_1\}$

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

δ	0	1
{q0}	{q0,q1}	{q1}
{q0,q1}	{q0,q1}	{q0,q1}
{q1}	ϕ	{q0,q1}
ϕ		

Finished

Unfinished

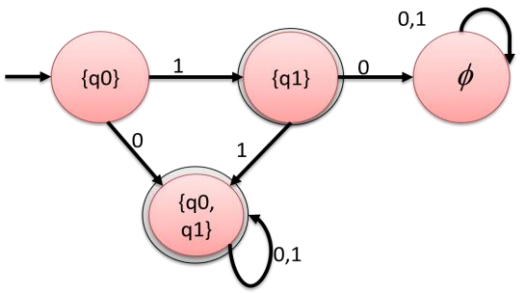
d) Process { ϕ }

$\delta(\phi,0) = \{\phi\}$
$\delta(\phi,1) = \{\phi\}$

	0	1
{q0}	{q0,q1}	{q1}
{q0,q1}	{q0,q1}	{q0,q1}
{q1}	ϕ	{q0,q1}
ϕ	ϕ	ϕ

Finished

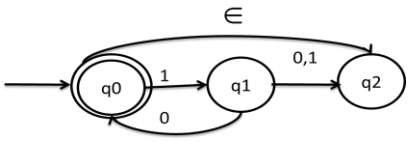
The transition Diagram for the DFA



Construction of a DFA equivalent to NFA-∈: To obtain the DFA from the given NFA-∈, the subset construction algorithm is used. The following points have to be noted for ∈- transitions.

- Start state q_0 consists of start state q_0 and all states, reachable from q_0 by ∈- transition.
- For any state q′, add all transitions from q′ reachable with ∈.

Example: Construct a DFA, equivalent to the NFA given



The transition table of the above NFA- $\in$

States\ Input	0	1	$\in$
q0	ϕ	{q1}	{q2}
q1	{q0,q2}	{q2}	ϕ
q2	ϕ	ϕ	ϕ

No. of states in NFA = 2

No. of states in DFA = $2^3 = 8$

Consider, q0 consisting of start state q0 and all states reachable from q0 by $\in$-transitions

q0 = {q0,q2} ⟶

δ	0	1
{q0,q2}		

⟸ Unfinished

a) Process {q0,q2}

$$\delta(\{q_0,q_2\},0) = \delta(q_0,0) \cup \delta(q_2,0) = \phi$$
$$\delta(\{q_0,q_2\},1) = \delta(q_0,1) \cup \delta(q_2,1) = \{q_1\}$$

δ	0	1	
{q0,q2}	ϕ	**{q1}**	⟸ finished
{ϕ}			Unfinished
{q1}			

b) Process {ϕ}

$$\delta(\phi,0) = \{\phi\}$$
$$\delta(\phi,1) = \{\phi\}$$

δ	0	1	
{q0,q2}	ϕ	**{q1}**	finished
{ϕ}	ϕ	ϕ	
{q1}			⟸ Unfinished

c) Process {q1}

$$\delta(q_1,0) = \{q_0,q_2\}$$
$$\delta(q_1,1) = \{q_2\}$$

δ	0	1	
{q0,q2}	ϕ	**{q1}**	
{ϕ}	ϕ	ϕ	
{q1}	**{q0,q2}**	**q2**	
q2			⟸ Unfinished

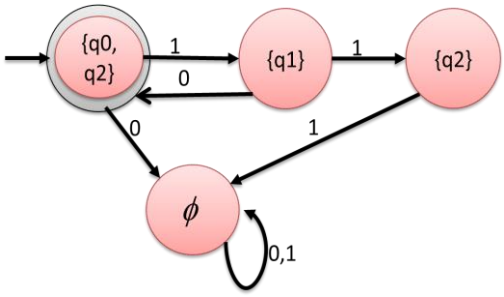
d) Process {q2}

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

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

δ	0	1
{q0,q2}	ϕ	{q1}
$\{\phi\}$	ϕ	ϕ
{q1}	{q0,q2}	q2
{q2}	ϕ	ϕ

The transition diagram for the DFA is



Procedure to construct an equivalent of NFA with $\in$- moves to an NFA without $\in$ - moves: To obtain NFA, N from NFA-$\in$,N$\in$, elimination of $\in$- transitions from a given automata is required. But, simply eliminating $\in$-transitions from N$\in$ will change the language accepted by the automata. Thus, if $\in$ transitions are eliminated then some non- $\in$- transitions should be added as substitutes, in order to maintain the language accepted by automata.

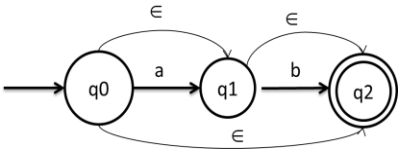
For $N_\in = (Q,\Sigma,\delta,q_0,F)$ and $N = (Q,\Sigma',\delta',q_0,F')$, find

a. $\delta'(q,a) = \in -closure(\delta(\in -closure(q),a))$

b. $F' = F \cup \{q_0\}$ if $\in -closure(q_0)$ contains member of F

= F otherwise

Example: Construct an equivalent NFA without $\in$ moves for a given automata



The transition table of the above NFA- $\in$ $N_\in = (Q,\Sigma,\delta,q_0,F)$

States\ Input	a	b	$\in$
q0	{q1}	ϕ	{q1,q2}
q1	ϕ	{q2}	{q2}
q2	ϕ	ϕ	ϕ

Let $N = (Q, \Sigma', \delta', q_0, F')$

a) Compute F'

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

$F' = F \cup \{q_0\}$ if $\in -closure(q_0)$ contains member of F

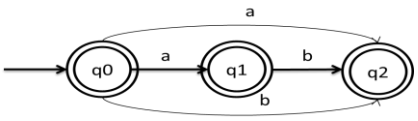
$= \{q_0, q_1, q_2\}$

b) Compute δ'

- $\delta'(q_0, a) = \in -closure(\delta(\in -closure(q_0), a))$
  $= \in -closure(\delta(\{q_0, q_1, q_2\}, a))$
  $= \in -closure(\delta(q_0, a) \cup \delta(q_1, a) \cup \delta(q_2, a))$
  $= \in -closure(q_1 \cup \phi \cup \phi)$
  $= \{q_1, q_2\}$
- $\delta'(q_0, b) = \in -closure(\delta(\in -closure(q_0), b))$
  $= \in -closure(\delta(\{q_0, q_1, q_2\}, b))$
  $= \in -closure(\delta(q_0, b) \cup \delta(q_1, b) \cup \delta(q_2, b))$
  $= \in -closure(\phi \cup q_2 \cup \phi)$
  $= q_2$
- $\delta'(q_1, a) = \in -closure(\delta(\in -closure(q_1), a))$
  $= \in -closure(\delta(\{q_1, q_2\}, a))$
  $= \in -closure(\delta(q_1, a) \cup \delta(q_2, a))$
  $= \in -closure(\phi)$
  $= \phi$
- $\delta'(q_1, b) = \in -closure(\delta(\in -closure(q_1), b))$
  $= \in -closure(\delta(\{q_1, q_2\}, b))$
  $= \in -closure(\delta(q_1, b) \cup \delta(q_2, b))$
  $= \in -closure(q_2)$
  $= q_2$
- $\delta'(q_2, a) = \in -closure(\delta(\in -closure(q_2), a))$
  $= \in -closure(\delta(\{q_2\}, a))$
  $= \in -closure(\phi)$
  $= \phi$
- $\delta'(q_2, b) = \in -closure(\delta(\in -closure(q_2), b))$
  $= \in -closure(\delta(\{q_2\}, b))$
  $= \in -closure(\phi)$
  $= \phi$

States\ Input	a	b
q0	{q1,q2}	{q2}
q1	ϕ	{q2}
q2	ϕ	ϕ

The transition diagram for NFA is



MINIMIZATION/OPTIMIZATION OF DFA

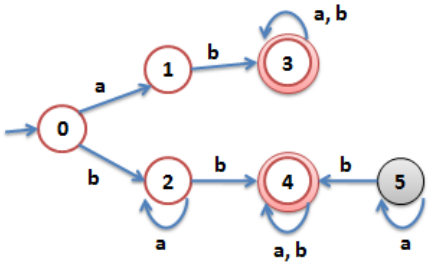
Optimum DFA: It is possible to have more than one DFAs that accepts the same language. Among these equivalent DFAs, it is often useful to find the smallest, i.e, the DFA with the *minimum possible number of states.*

Minimization of a deterministic finite automaton refers to the detection of those states of a DFA, whose presence or absence in a DFA does not affect the language accepted by the automata. The states that can be eliminated from automata, without affecting the language accepted by automata, are:

- Unreachable states
- Dead states
- Indistinguishable states

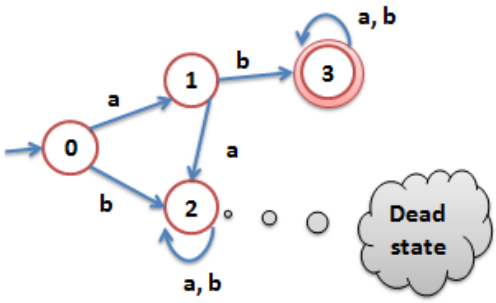
Unreachable states: Are those states that are not reachable from the initial state of the DFA, for any input string.

Eg: Here , state 5 is unreachable, from the initial state 0, with any input string (either b or a).



Dead state: A rejecting state that is essentially a dead end. Once the machine enters a dead state, there is no way for it to reach an accepting state, so that the string is going to be rejected. Graphically, the dead state is often omitted and assumed for any input that the machine does not have explicit instructions on what to do with. A machine may have multiple dead states, but at most only one dead state is needed per machine.

A dead state is a non-final state of a DFA, whose transitions on every input symbol terminated on itself. Formally '**q**' is a dead state, if **q is in Q** and $\delta(q,a) = q$ for every 'a' in Σ.



Here, if the given input is 'b' at state '0' then the machine enters state '3' and remains there for any input string(a or b). Hence '3' is a dead or trap state.

Indistinguishable and Distinguishable states: States are said to be ***indistinguishable,*** if their merger does not change the language accepted by a DFA; otherwise the states are distinguishable states.

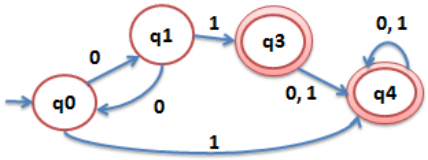
Formally, defined as

- **Indistinguishable:** Two states 'p' and 'q' of a DFA are called indistinguishable,
  if $\delta(p,w) \in F \Rightarrow \delta(q,w) \in F$
  and $\delta(p,w) \notin F \Rightarrow \delta(q,w) \notin F, \quad \forall w \in \Sigma^*$
- **Distinguishable:** Two states 'p' and 'q' of a DFA are called Distinguishable,
  if $\delta(p,w) \in F \Rightarrow \delta(q,w) \notin F$
  and $\delta(p,w) \notin F \Rightarrow \delta(q,w) \in F, \quad \forall w \in \Sigma^*$

Two states 'p' and 'q' of a DFA are equivalent or indistinguishable, if every string *w* leads from p to a final state, if and only leads from 'q' to a final state.

To show that two states are *non-equivalent* or *distinguishable*, we need to find only one string that leads from one of them to a final state and leads from other to a non-final state.

Eg:



Consider two states of the DFA, q_0 and q_1, with Q = {q_0,q_2,q_3,q_4} and F = {q3,q_4}

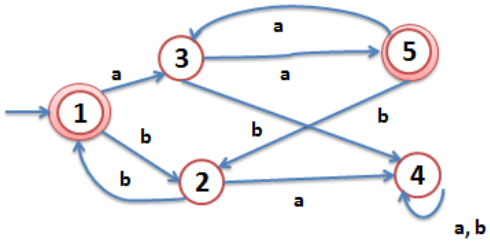
- Input = 1, $\delta(q_0,1) = q_3 \in F$, $\delta(q_1,1) = q_3 \in F$
- Input = 0, $\delta(q_0,0) = q_1 \notin F$, $\delta(q_1,0) = q_0 \notin F$

Since for every string 1, 1 leads from q_0 to the final state q_3 and also 1 leads from q1 to the final state q_3, the two states (q_0,q_1) are called indistinguishable or equivalent states.

Procedure to Detect Indistinguishable State:

1. *Partition $\pi = (F, Q-F)$, where F is the set of final states and Q is the set of states.*
2. *Perform partition π_{new} into subsets of π such that,*
   a. *For any input symbols a and b, if a transition is made to states of the same subset of π or final states, then no partitions are made further. Go to step-5*
   b. *For any input symbol a and b, if a transition is made to states of the different subset of π, then the subset is further partitioned.*
3. *$\pi = \pi_{new}$*
4. *Repeat step-1 through step-3, until $\pi_{new} \neq \pi$*
5. *If members of the subset π on input string 'a' or 'b' go for the same transition states (either final or non-final), then the states are indistinguishable.*

Eg: Identify the indistinguishable states from the DFA, shown in fig:



a. ***π = ({2,3,4}, {1,5})***

Partitions: 2 3 4 | 1 5

From π_{new}, with subset {2,3,4}:

- Input = a,

$$\delta(2,a) = 4$$
$$\delta(3,a) = 5$$
$$\delta(4,a) = 4$$

- Input = b,

$$\delta(2,b) = 1$$
$$\delta(3,b) = 4$$
$$\delta(4,b) = 4$$

Since for the input a, state 3 gives a transition to a state of other member in the subset {1,5}, 3 is partitioned.

$$\pi_{new} = (\{2,4\}, \{3\}, \{1,5\})$$
$$\pi = \pi_{new}$$

b. ***π = ({2,4}, {3}, {1,5})***

Partitions: 2 4 | 3 | 1 5

From π_{new}, with subset {2,4}:

- Input = a,

$$\delta(2,a) = 4$$
$$\delta(4,a) = 4$$

- Input = b,

$$\delta(2,b) = 1$$
$$\delta(4,b) = 4$$

Since for the input b, state 2 gives a transition to a state of other member in the subset {1,5}, 2 is partitioned.

$$\pi_{new} = (\{2\}, \{4\}, \{3\}, \{1,5\})$$
$$\pi = \pi_{new}$$

c. $\pi = (\{2\},\{4\},\{3\}, \{1,5\})$

Partitions: 2 | 4 | 3 | 1 5

From π_{new}, with subset {1,5}:

- Input = a,

$$\delta(1,a) = 3$$
$$\delta(5,a) = 3$$

- Input = b,

$$\delta(1,b) = 2$$
$$\delta(5,b) = 2$$

Since for the input a,b states 1 and 5 gives a transition to the same state, they are not the partitioned.

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

$\pi = \pi_{new}$ *is true i.e,* $\pi \neq \pi_{new}$ is false

Stop iteration.

Thus, states 1 and 5, on the inputs a and b, give transitions to same non-final states. Hence, 1 and 5 are called *indistinguishable states.*

Minimal DFA:-

A DFA is minimal, if and only if,

- All its states are reachable from the start state.
- All its states are distinguishable.

Minimization Algorithm:-

Let $M = <Q, \Sigma, q_0, \delta, F>$ be a DFA that accepts a language L. Then the following algorithm produces the DFA, that has the smallest number of states among all the DFAs, that accepts L.

Step -1: Identify all unreachable or inaccessible states and eliminate them from the DFA,M.

Step -2: Identify all indistinguishable states from the DFA and merge them all to form the DFA with smallest number of states.

Step -3: Construct a DFA from π_{final}

Step -4: END

The following is the procedure to detect the indistinguishable state:

- Construct a partition $\pi = \{F, Q-F\}$
- do

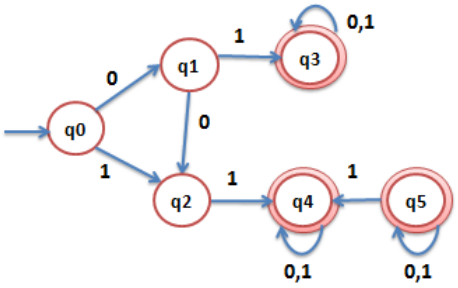
  $\pi_{new} = Make-partition(\pi)$

  $\pi = \pi_{new}$

  $while(\pi_{new} \neq \pi)$
- $\pi_{final} = \pi$
- If the members of subset π on an input string go for same transition states, either final or non-final, then the states are indistinguishable.

> ***Function Make-partition(π)***
> For each set S of π *do*
> *Begin*
> *Partition s into subsets such that 2 states 'p' and 'q' of s are in the same subset, iff, for each input symbol 'a' or 'b', p and q make a transition to the same states of s or final states. Otherwise, no partitions are made.*
> *End.*

Eg: Minimise the DFA



Solution:

Unreachable state: If we enumerate all simple paths starting from the initial state, then we find that the state q_5 is said to be unreachable. So,it is removed.

Indistingusihable state:

a. π = ***({q0,q1,q2}, {q3,q4})***

Partitions: ***q0 q1 q2*** | ***q3 q4***

From π_{new}, with subset { *q0,q1,q2*}:

- Input = 0,

  $\delta(q_0,0) = q_1$

  $\delta(q_1,0) = q_2$

  $\delta(q_2,0) = \phi$

- Input = 1,

  $\delta(q_0,1) = q_2$

  $\delta(q_1,1) = q_3$

  $\delta(q_2,1) = q_4$

Since for the input '1', state q_1 and q_2 gives a transition to a state of other member $\{q_3,q_4\}$, $\{q_1,q_2\}$ are partitioned.

$$\pi_{new} = (\{q_0\}, \{q_1,q_2\}, \{q_3,q_4\})$$
$$\pi = \pi_{new}$$

b. $\pi = (\{q_0\}, \{q_1,q_2\}, \{q_3,q_4\})$

Partitions: q_0 | q_1 q_2 | q_3 q_4

From π_{new}, with subset { q_1 , q_2}:

- Input = 0,
  $\delta(q_1,0) = q_2$
  $\delta(q_2,0) = \phi$
- Input = 1,
  $\delta(q_1,1) = q_3 \in F$
  $\delta(q_2,1) = q_4 \in F$

Since for the input 1, q_1 and q_2 gives a transition to a state of other member in the subset $\{q_3,q_4\}$. Also both q_3 and q_4 belong to final states, thus q_1 and q_2 are partitioned. Hencem they are distinguishable.

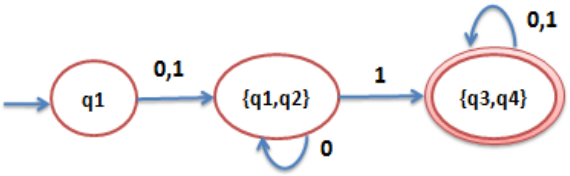
From π_{new}, with subset { q_3 , q_4}:

- Input = 0,
  $\delta(q_3,0) = q_3$
  $\delta(q_4,0) = \phi$
- Input = 1,
  $\delta(q_3,1) = q_3 \in F$
  $\delta(q_4,1) = q_4 \in F$

Since for the input 1, q_3 and q_4 belong to the same subset members, they are not partitioned and are distinguishable.

$$\Rightarrow \pi \neq \pi_{final} \text{ is false.}$$

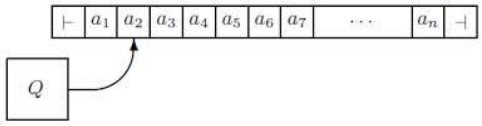
Merging of all indistinguishable states leads to a minimum DFA, given in figure



TWO-WAY DFA: A two-way deterministic finite automaton (**2DFA**) is an abstract machine, a generalized version of the deterministic finite automaton (DFA) which can revisit characters already processed.

or

A mathematical model of a machine, with the ability of a read-head to move left as well as right is a 2DFA. It consists of the symbols of the input string as occupying cells of a finite tape, one symbol per cell. The input string is enclosed in left and right endmarkers ⊣ and ⊢, which are not elements of the input alphabet Σ. The read head may not move outside of the endmarkers.



How 2DFA Operates: Initially, the 2DFA is assumed to be in the initial state q_0 with its read-head on the leftmost symbol of the input string.During each move of 2DFA, the read head moves one position to the right or one position to the left, depending on the transitions defined for a given DFA. The 2DFA is then said to have accepted an input string, if it moves the read-head off the right end of the tape, entering an accepting state simultaneously.

*Formally, a 2DFA **M** is a five tuple **M** =* $(Q, \Sigma, \delta, q_0, F)$

where,

Q *is a finite set of states,*

Σ *is a finite set of input symbols,*

$\delta : Q \times \Sigma \to Q \times \{L, R\}$,

$q_0 \in Q$ *is the start state*

$F \subseteq Q$ *is the set of accept states.*

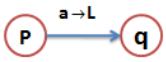
The **transition function** δ is a map from

$$Q \times \Sigma \to Q \times \{L, R\}$$

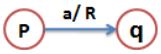
- if $\delta(q_1, a) = (q_2, L)$, then on reading the input symbol 'a' the 2DFA enters from state q_1 to state q_2 and moves its read-head **left** by one square.
- if $\delta(q_1, a) = (q_2, R)$, then on reading the input symbol 'a' the 2DFA enters from state q_1 to state q_2 and moves its read-head **right** by one square.

Transition Diagram: For a directed graph, an arc going from one vertex (which corresponds to the state P) to another vertex (that corresponds to the state q) and also the edge label, can be represented in different forms as follows:

Form-1: *input symbol (a)* $\longrightarrow$ *Left(L) or Right(R)*



Form-2: *input symbol (a) / Left(L) or Right(R)*



Transition table: The description of how a 2DFA operates for a given set of symbols on the tape can be represented in a tabular format called transition table. Different table forms are all follows:

Form 1:

Current State	Input Symbol	New state	Move

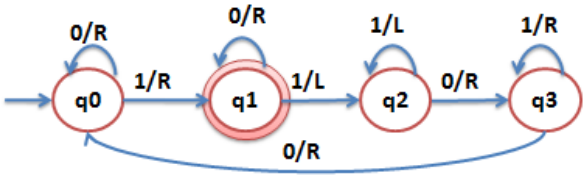
Form 2:

States\Input	a1	a2	.....	an
S1		(S1,move)		
..				
Sn				

Example: Consider a **2DFA** M = (Q,Σ,δ,q_0,F) where

Q = (q_0,q_1,q_2,q_3)
Σ = {0,1},
q0 = {q_0},
F = {q_1}

Transition Diagram:



Transition table:

Current State	Input Symbol	New state	Move
q_0	0	q_0	R
q_0	1	q_1	R
q_1	0	q_1	R
q_1	1	q_2	L
q_2	0	q_3	R
q_2	1	q_2	L
q_3	0	q_0	R
q_3	1	q_3	R

Instantaneous Description of a 2DFA:

To describe the behavior of a 2DFA for a given input, an instantaneous description (ID) has been introduced that describes:

- ✓ the input string
- ✓ current state and
- ✓ current position of read head.

An ID of a 2DFA is a string xqy, where q is the current state, xy are the input symbols from the tape and the read-head points to the first character of the substring *y*. The initial ID is denoted *qxy*, where q is the start state and the head points to the first symbol *x* (from left). The final ID is denoted by *xyqe,* where 'e' indicates that the head has moved off the right end of the input.

Formally, the realtion $\vdash_D$ on ID is defined as $I_1 \vdash_D I_2$ iff D can go from I_1 and I_2 in one move and D is a string in $\Sigma^*Q\Sigma^*$.

Language accepted by a 2DFA:- If the read-head moves to the right end of the tape, entering an accepting state, a 2DFA is said to have accepted the input string.The language accepted by 2DFA, M can be defined as

L(M) = {w/q_0w ⊢ wq for some q ∈ F}

Example: Consider, a 2DFA, M = (Q,Σ,δ,q_0,F) where Q = (q_0,q_1,q_2,q_3,q_4), Σ = {0,1}, q0 = {q_0}, F = {q_2} and δ is given by

States \ Input	0	1
q_0	(q_0,R)	(q_1,R)
q_1	(q_1,R)	(q_2,R)
q_2	(q_2,R)	(q_3,L)
q_3	(q_4,L)	(q_3,L)
q_4	(q_0,R)	(q_4,L)

Formally, ID: $q_0$1001 ⊢ 1$q_1$001
⊢ 10$q_1$01
⊢ 100$q_1$1
⊢ 1001q_2e

For the given input string w =1001, the read-head of the 2DFA moves to the right end of the tape entering an accepting state q_2.

DFA	2DFA
It can read input only once from left to right.	It read the input back and forth with no limit on number of times an input symbol can be read.
Transition function is defined as: $\delta: Q\times\Sigma \to Q$	Transition function is defined as: $\delta: Q\times\Sigma \to Q\times\{L,R\}$
Less meaningful	More meaningful and arguably more realistic model than DFA.
Restricted version of a 2DFA	It solves any problem solved by DFA.
Complex design compare to 2DFA	Simpler in design

TRANSDUCERS

In the case of FSA movements from state q_i to q_j depend on the input at q_i, and no output emerges. However, in the case of FSM, a move from a state q_i to state q_j results in an output. Consequently, a FSM possesses two special features: a finite set of output symbols Γ and an output function $\lambda: Q\times\Sigma \to \Gamma$ where Σ is the input alphabet. Thus, one major symbols and an output is limited to a binary signal: 'accept' or 'don't accept'. In order to make finite automata to have output capabilities, two classical machines are designed that transform input strings into output strings. They are

a. Moore machine and
b. Mealy machine.

A **Moore machine** is a FSM - M_o, named after Edward Moore, who introduced it in 1956.

A **Mealy machine** is a FSM – M_e, named after George H.Mealy, who introduced it in 1955.

These machines are basically DFAs, except that they associate an output symbol with each state or with each state transition. However, there are no final states, because there is no acceptance or rejection involved.

The purpose of moore and mealy machine is not to answer yes or no, to accept or reject a string. They are not a language recogniser but output producer.

Moore Machine:

Definition:- A moore machine associate an output symbol with each state, and each time a state is entered, an output is obtained simultaneously. So the first output always occurs as soon as the machine starts.

Elements of Moore Machine

A Moore machine has the following six characteristics:

a. A finite set Q of states $q_0, q_1, \ldots q_n$.
b. A finite input alphabet of letters, for forming the input String, $\Sigma = \{a, b, ..\}$
c. A finite output alphabet of possible output characters, $\Gamma = \{x, y, ..\}$
d. A transition function 'δ' that shows, for each state and each input letter, what state is reached next.
e. An output function 'λ' that shows what characters from is printed by each state that is entered.
f. An initial state q_o.

Ordered Six-tuple Specification of Moore Machine

Formally, the moore machine M_o is represented by 6-tuples

$$M_o = (Q, \Sigma, \delta, q_0, \Gamma, \lambda)$$

where

a. Q is a finite set of states,
b. Σ is a finite set of input symbols,
c. $\delta : Q \times \Sigma \to Q$ is the transition function
d. $q_0 \in Q$ is the initial state,
e. Γ is a finite set of output symbols,
f. $\lambda = Q \to \Gamma$ is the output function.

Here, the output of M_o, in response to inputs $q_1, q_2, \ldots q_n$, n >= 0, is $\lambda(q_0), \lambda(q_1), \ldots \lambda(q_n)$ and $\lambda(q_0) = \in$. Therefore, the output sequence of M_o consists of (n+1) symbols.

In other words for a Moore machine, each state produce a one-character output immediately, upon the machine's entry into that state. At the beginning, the start state produces an output before any input has been read. Thus the output of a Moore machine is one character larger than its input (n+1), wher n is number of characters in an input string).

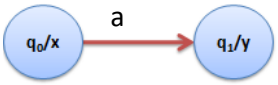
Description of a Moore Machine

The transitions of a Moore machine can be represented using transition diagram and transition table.

Transition diagram:

The transition diagram for Moore machine will include the output for each state. Each circle of a transition diagram is labelled with a compound symbol $q_i|x$, to indicate that

Example:If the output associated with a state q_0/x inside the circle. A typical transition diagram for a Moore machine is shown in fig.



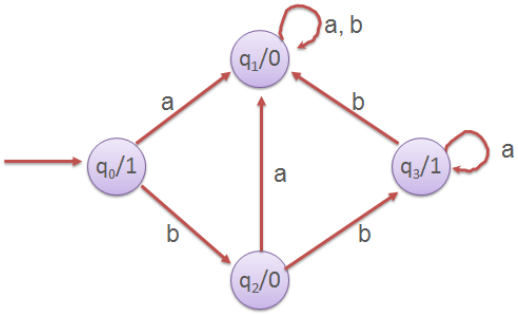
Transition Table:

States \ Input	Inputs			Output
		...		

Since in Moore machine, every state is associated with output, the transition table is called transition and output table. The rows of the table correspond to states and columns correspond to inputs and output. Entries correspond to next state and its output.

EXAMPLE :

Consider a Moore machine, whose transition diagram and table is shown in



	a	b	OUT
q_0	q_1	q_2	1
q_1	q_1	q_1	0
q_2	q_1	q_3	0
q_3	q_3	q_1	1

Design of a Moore Machine:

Basic design strategy for Moore machine is as follows:

a. Understand the problem definition for which the Moore machine has to be designed.
b. Determine the required alphabet and state set.
c. determine the required output set.
d. For each state, decide on the transition to be made for each character of the input string.
e. For each state, decide the required output.
f. Obtain the transition table and diagram.
g. Test the Moore machine obtained on short strings.

There are no accept states in a Moore machine because it is not a language recogniser but an output producer.

Example: Construct a moore machine, for the following:

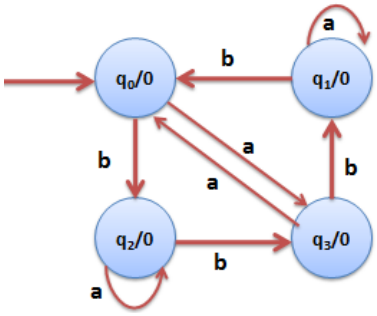
Input alphabet $\Sigma = \{a, b\}$

Output alphabet $\Gamma = \{0,1\}$

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

States \ Input	a	b	Output
q_0	q_3	q_2	0
q_1	q_1	q_0	0
q_2	q_2	q_3	1
q_3	q_0	q_1	0

Solution: Given $\Sigma = \{a, b\}$, $\Gamma = \{0,1\}$ and Q = $\{q_0, q_1, q_2, q_3\}$. The moore machine is shown as



M_0 action for input string:

For the input string S = bababbb, where n = 7, what is the output string?

- $\lambda(\delta(q_0, \in)) = \lambda(q_0) = 0$
- $\lambda(\delta(q_0, b)) = \lambda(q_2) = 1$
- $\lambda(\delta(q_0, ba)) = \lambda(\delta(\delta(q_0, b)a))$
  $= \lambda(\delta(q_2, a))$
  $= \lambda(q_2)$
  $= 1$
- $\lambda(\delta(q_0, bab)) = \lambda(\delta(\delta(q_0, ba)b))$
  $= \lambda(\delta(q_2, b))$
  $= \lambda(q_3)$
  $= 0$
- $\lambda(\delta(q_0, baba)) = \lambda(\delta(\delta(q_0, bab)a))$
  $= \lambda(\delta(q_3, b))$
  $= \lambda(q_0)$
  $= 0$
- $\lambda(\delta(q_0, babab)) = \lambda(\delta(\delta(q_0, baba)b))$
  $= \lambda(\delta(q_0, b))$
  $= \lambda(q_2)$
  $= 1$
- $\lambda(\delta(q_0, bababb)) = \lambda(\delta(\delta(q_0, babab)b))$

$= \lambda(\delta(q_2, b))$

$= \lambda(q_3)$

$= 0$

- $\lambda(\delta(q_0, bababbb)) = \lambda(\delta(\delta(q_0, bababb)b))$

$= \lambda(\delta(q_3, b))$

$= \lambda(q_1)$

$= 0$

Thus, for the input string S, the output is 01100100 (n = n+1)

Mealy Machine:

Definition: A Mealy machine is a finite state machine, where the outputs are determined by the current state and the input.

A Mealy machine associates an output symbol with each transition and the output depends on the current input.

Elements of a Mealy machine

A Mealy machine features the following six characteristics:

a. A finite set Q of states $q_0, q_1, \ldots q_n$.

b. A finite input alphabet of letters, for forming the input String, $\Sigma = \{a, b, ..\}$

c. A finite output alphabet of possible output characters, $\Gamma = \{x, y, ..\}$

d. A transition function 'δ' that shows, for each state and each input letter, what state is reached next.

e. An output function 'λ' that shows what characters from Γ is printed by each state that is entered.

f. An initial state q_o.

Ordered Six-tuple Specification of Mealy Machine

Formally, the mealy machine M_e is represented by 6-tuples

$M_e = (Q, \Sigma, \delta, q_0, \Gamma, \lambda)$

where

a. Q is a finite set of states,

b. Σ is a finite set of input symbols,

c. $\delta : Q \times \Sigma \rightarrow Q$ is the transition function

d. $q_0 \in Q$ is the initial state,

e. Γ is a finite set of output symbols,

f. $\lambda = Q \times \Sigma \rightarrow \Gamma$ is the output function.

Here, the output of M_e, in response to inputs $a_1, a_2, \ldots a_n$, n >= 0, is $\lambda(q_0, a_1), \lambda(q_1, a_2), \ldots\ldots \lambda(q_{n-1}, a_n)$, where $\{q_0, q_1, q_2, \ldots, q_n\}$ is the sequence of states such that $\delta(q_{i-1}, a_i) = q_i$ for $1 \le i \le n$. The output sequence of M_e consists of 'n' symbols.

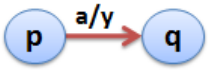
Description of a Mealy Machine

The transitions of a Mealy machine can be represented using transition diagram and transition table.

Transition diagram:

The transition diagram for Mealy machine will include the output for each transition edge. Each edge is labelled with the compound symbol a/b from state p to state q to indicate the $\delta(p,a)=q$ and $\lambda(p,a)=b$. Every state must have exactly one out-going edge for each possible input symbol

***Example*:**



Transition Table:

Q \ Σ	Present Input		
	a1	.....	an

Transition table

Q \ Γ	Present Input		
	a1	...	an
q1	b1	...	bn
.			
.			
.			
qn			

Output table

The transition table for Mealy machine consists of two sub-tables

- *Transition table*: Rows correspond to states and columns correspond to input. Entries correspond to next states.
- *Output table*: Rows correspond to states and columns correspond to input. Entries correspond to ouput.

Design of a Mealy Machine:

Basic design strategy for Mealy machine is as follows:

a. Understand the problem definition for which the Mealy machine has to be designed.
b. Determine the required alphabet, state and output set.
c. For each state, decide on the transition to be made for each character of the input string.
d. Decide the output to be associated, with each edge label.
e. Obtain the transition table, output table and transition diagram.
f. Test the Mealy machine obtained on short strings.

> *There are no accept states in a Mealy machine because it is not a language recogniser but an output producer.*

Example: Construct a mealy machine, for the following:

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

Output alphabet $\Gamma=\{y,n\}$

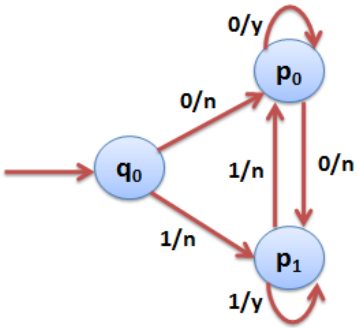
States Q = $\{q_0, p_0, p_1\}$

Q\Σ	0	1
q_0	p_0	p_1
p_0	p_0	p_1
p_1	p_0	p_1

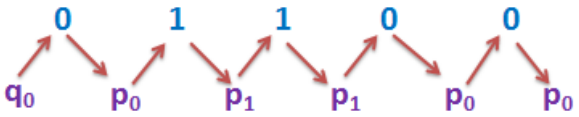
Q\Γ	0	1
q_0	n	n
p_0	y	n
p_1	n	y

Transition and Output table

Solution: Let $M_e = (Q, \Sigma, \delta, q_0, \Gamma, \lambda)$ be the mealy machine. Given $\Sigma = \{0,1\}, Q = \{q_0, p_1, p_2\}, \Gamma = \{y, n\}$



- $\delta(q_0,0) = p_0$
- $\delta(q_0,01) = \delta(\delta(q_0,0),1) = \delta(p_0,1) = p_1$
- $\delta(q_0,011) = \delta(\delta(q_0,01),1) = \delta(p_1,1) = p_1$
- $\delta(q_0,0110) = \delta(\delta(q_0,011),0) = \delta(p_1,0) = p_0$
- $\delta(q_0,01100) = \delta(\delta(q_0,0110),0) = \delta(p_0,0) = p_0$



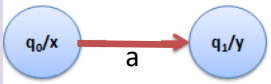
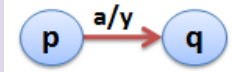
Sequence State Diagram

Sequence of states entered for input 01100 is $q_0p_0p_1p_1p_0p_0$ (number of states entered is m=6, output sequence x = m-1 i.e, x = 5).

From the sequence state diagram:

- $\lambda(q_0,0) = n$
- $\lambda(p_0,1) = n$
- $\lambda(p_1,1) = y$
- $\lambda(p_1,0) = n$
- $\lambda(p_0,0) = y$ Output Sequence = nnyny

Difference between Moore machine and Mealy machine:

Moore Machine	Mealy Machine
It gives output $\lambda(q_0)$ in response to the input $\in$, so the output sequence is n+1.	It gives output $\lambda(q_0)$ in response to the input $\in$, so the output sequence is n.
It has actions associated with states	It has actions associated with transitions
The output of moore machine depends only on the current state and does not depend on current input.	The output of mealy machine depends only on the current input.
Output associated with state q is x, then q/x is written inside the state circle of transition diagram.	Output associated with edge labelled with the letter a is x, it is written as a/x on that edge.
	
It has more numbers of states compare to mealy machine.	It has less number of states.

Construction of Mealy machine for a given Moore machine:

***Theorem*:** *If M_o is a Moore machine, then there is a Mealy machine M_e, that is equivalent to it.*

Proof: Let M_o is a Moore machine given by

$$M_o = (Q, \Sigma, \delta, q_0, \Gamma, \lambda)$$

M_e is a Moore machine given by

$$M_e = (Q, \Sigma, \delta, q_0, \Gamma, \lambda')$$

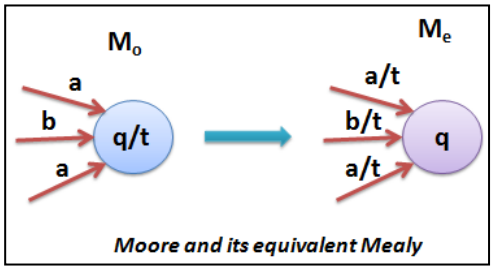
This theorem can be proved in a constructive manner with λ' defined as

$$\lambda'(q,a) = \lambda(\delta(q,a)) \text{ for all q and a.}$$

Thus M_o and M_e enter the same sequence of states on the same input, and with each transition Me produces the output that M_o associates with the state entered.

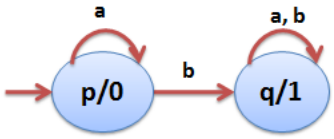
Alternatively, M_e is contructed from M_o as follows:

a) Consider any state q_i for M_o
b) Assume, M_o prints the character t upto entering q_i
c) Hence, the label the state q_i is q_i/t
d) Assume that there are n arcs entering q_i, with labels $a_1, a_2, \ldots, a_n$.
e) Now, create the machine M_e by changing the labels on the incoming arcs from q_i to a_m/t, m = 1,2,...,n.
f) Change the label of state q_i to be just q_i



Moore and its equivalent Mealy

Example: Convert the following Moore machine to an equivalent mealy machine



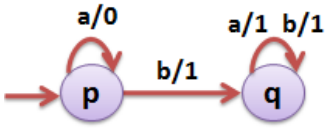
Construct M_e as

$$M_e = (Q, \Sigma, \delta, \Gamma, \lambda', q_0) \text{ where } \lambda'(q,a) = \lambda(\delta(q,a)), \ \forall \text{ q and a}$$

Thus,

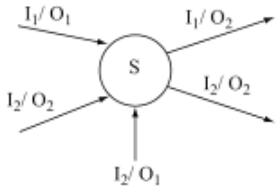
- $\lambda'(q_0,a) = \lambda(\delta(q_0,a))$
  $= \lambda(q_0)$
  $= 0 \quad \Rightarrow \quad a/0$
- $\lambda'(q_0,b) = \lambda(\delta(q_0,b))$
  $= \lambda(q_1)$
  $= 1 \quad \Rightarrow \quad b/1$
- $\lambda'(q_1,a) = \lambda(\delta(q_1,a))$
  $= \lambda(q_1)$
  $= 1 \quad \Rightarrow \quad a/1$
- $\lambda'(q_1,b) = \lambda(\delta(q_1,b))$
  $= \lambda(q_1)$
  $= 1 \quad \Rightarrow \quad b/1$

The corresponding Mealy machine is given as



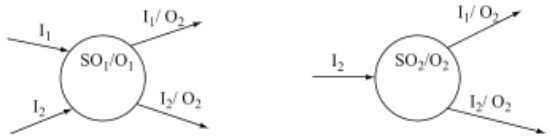
Conversion of Moore machine from a given Mealy machine: For a given Mealy machine M_c there is an equivalent Moore Machine M_o. These can be constructed in several steps.

Step I: In mealy machine output depends on present state and present input. Hence, in the transac-tional diagram of a mealy machine transactional edges are labeled with input as well as output. For a state two types of edges are there, incoming edges and outgoing edges. For incoming edges it may happen that the output differs for two incoming edges like the following:

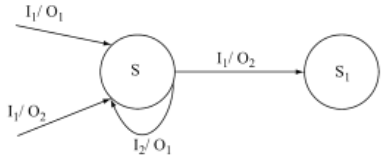


In the transitional diagram for state S, incoming edges are labeled as I_1/O_1, I_2/O_2 and $I_2/O_1$and the outgoing edges are labeled as I_1/O_2 and I_2/O_2. For state S for incoming edges we are getting two types of output O_1 and O_2. These types of cases the state S will be divided into n number of parts, where n = number of different

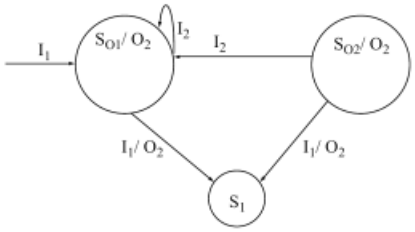
outputs for the incoming edges to the state. The output edges will be repeated for all the divided states. The transitional diagrams for the above case will be



Step II: If a state has a loop and that state also need to be divided like follows:

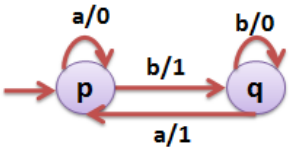


(Here for input I_1, output is O_1 as well as O_2,. Hence the state S needs to be divided. The S has a loop with input I_2 and output O_1) The state S will be divided into two states S_{01} and S_{02}. As the loop for input I_2, is labeled with output O_1 Hence there will be a loop on state S_{01}. However, this loop is not possible on S_{02}, because it produces output O_2. Hence, there will be a transition from S_{02} to S_{01} with input label I_2.



Step III: Repeat Step I and II to get moore machine

Example: Convert the mealy machine into an equivalent moore machine.



Solution: Given the mealy machine $M_e = (\{p,q\},\{a,b\},\{0,1\},\delta,\lambda',q_0)$, we construct an equivalent moore machine $M_o = (Qx,\Sigma,\Gamma,\delta',\lambda',[p,b])$ where, $Qx = \{[p,0],[p,1],[q,0],[q,1]\}$, $\Sigma = \{a,b\}$

We need to find out:

$$\delta'([q,b],a) = [\delta(q,a),\lambda(q,a)] \text{ and } \lambda'(q,b) = b$$

Thus,

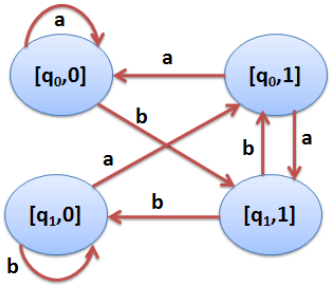
- $\delta'([q_0,0],a) = [\delta(q_0,a),\lambda(q_0,a)]$
  $= [q_0,0]$
- $\delta'([q_0,0],b) = [\delta(q_0,b),\lambda(q_0,b)]$
  $= [q_1,0]$

- $\delta'([q_0,1],a) = [\delta(q_0,a), \lambda(q_0,a)]$
  $= [q_0,0]$
- $\delta'([q_0,1],b) = [\delta(q_0,b), \lambda(q_0,b)]$
  $= [q_1,1]$
- $\delta'([q_1,0],a) = [\delta(q_1,a), \lambda(q_1,a)]$
  $= [q_0,1]$
- $\delta'([q_1,0],b) = [\delta(q_1,b), \lambda(q_1,b)]$
  $= [q_1,0]$
- $\delta'([q_1,1],a) = [\delta(q_1,a), \lambda(q_1,a)]$
  $= [q_0,1]$
- $\delta'([q_1,1],b) = [\delta(q_1,b), \lambda(q_1,b)]$
  $= [q_1,0]$

We also find: $\lambda'(q,b) = b$ for all states i.e,

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

Qx \ Input	a	b
[q0,0]	[q0,0]	[q1,1]
[q0,1]	[q0,0]	[q1,1]
[q1,0]	[q0,1]	[q1,0]
[q1,1]	[q0,1]	[q1,0]



Automata in Real World

1. Advantages of FSM

- It is easy for inexperienced developers to implement them with no extra knowledge.
- FSMs are quick to design, implement and execute.
- FSM is well known, even as an artificial intelligence technique.
- There are number of ways to implement FSM-based system in terms of topology, it is easy to incorporate many other techniques.
- It enables easy transfer from a meaningful abstract representation to a coded implementation.
- Predictable (deterministic FSM) - given a set of inputs and a known current state, the state transition can be predicted, allowing for easy testing.

2. Disadvantages of FSM

- The predictable nature of deterministic FSMs can be unwanted in some domains such as computer games (non-DFSM tries to solve this).

- The conditions for state transitions are ridged, meaning they are fixed.
- Not suited to all problem domains, should only be used when a systems behavior can be decomposed into separate states with well defined conditions for state transitions. This means that all states, transitions and conditions need to be known up front and be well defined !!

3. Applications of FSM

- Used in video game industry.
- The cellular automata are used for making pretty pictures and animations.
- The applications of FSMs are also found in language processing: Parsing, morphological representations, spelling, information retrieval, tagging, stemming, image compressions, cryptography and designing of compilers.
- A geographic automaton is used to simulate the behavior and distribution of objects in space such as householders, pedestrians, vehicles, shops, roads, land parcels, side walks etc.
- FSMs applied to biological and biomedical problem solving.

UNIT-2

UNIT – II: Regular Expressions -Regular Expressions, Regular Sets, Identity Rules, Equivalence of two Regular Expressions, Manipulations of Regular Expressions, Finite Automata and Regular Expressions, Inter Conversion, Equivalence between Finite Automata and Regular Expressions, Pumping Lemma, Closure Properties, Applications of Regular Expressions, Finite Automata and Regular Grammars, Regular Expressions and Regular Grammars.

REGULAR EXPRESSIONS

Regular expressions are useful for representing certain sets of strings in an algebraic fashion. RE's describe the languages accepted by the finite automata.

Definition: A formal recursive definition of regular expressions over ∑ is given as follows:

1. Any terminal symbol (i.e an element of ∑), ∈ & ϕ are RE's.
2. Union of two RE's R1 & R2 written as R1+R2 is also an RE.
3. Concatenation of two RE's R1 & R2 written as R1.R2 or R1R2 is also an RE.
4. Iteration (Closure) of R i.e R* is also an RE.
5. If R is an RE then (R) is also an RE.
6. The RE's over ∑ are obtained by recursive application of 1-5 rules once or several times.

Hierarchy of evaluation of regular expressions (Precedence)

The following order must be followed for evaluating a regular expression

1) Parentheses
2) Closure(Kleene Closure)
3) Concatenation
4) Union

Regular Set: Any set represented by a RE is called a regular set.

Examples:

Regular Expression	Regular Set
1) a+b	**{a,b}**
2) ab	**{ab}**
3) a*	**{∈,a,aa,aaa,….}**
4) (a+b)*	**{a,b}***
5) a	**{a}**

Regular Expressions	Meaning
0^*	Set of strings of 0's of any length including $\in$
0^+	Set of strings of 0's of any length excluding $\in$
$(a+b)^*$	Set of strings of a's and b's of any length including $\in$
$(a+b)^+$	Set of strings of a's and b's of any length excluding $\in$
$(a+b)^*abb$	Set of strings of a's and b's ending with abb
$0^*1^*2^*$	Set of strings of any number of 0's followed by any number of 1's, followed by any number of 2's
$0^+1^+2^+$	Set of strings of any number of 0's followed by any number of 1's, followed by any number of 2's excluding $\in$
$00^*11^*22^*$	Set of strings of 0's,1's and 2's with atleast one zero,followed by atleast one 1,followed by atleast one 2.
$(a+b)^*aa(a+b)^*$	Set of strings of a's and b's of any length having a substring aa.

1) Describe the following sets by regular expressions

a) {101} b) {01,10} c) {$\in$,ab)

d) {abb,a,b,baa} e) {$\in$,0,00,000,.....} f) {1,11,111,.....}

Sol: a) R=101 b) 01+10 c) R=$\in$+ab

d) R=abb+a+b+baa e) R= 0* f) R=1^+ (or) 1.1*

2) Describe the following sets by Regular expressions.

a) L1= the set of all strings of 0's and 1's ending in 00

Sol: (0+1)* 00

b) L2= the set of all strings of 0's and 1's starting with 0 and ending with 1.

Sol: 0(0+1)* 1

c) L3= {$\in$, 11, 1111, 111111....} (or) $(11)^n$, n≥0

Sol: (11)*

Identities for RE's

I_1: ϕ + R =R I_2: ϕR =R ϕ= ϕ I_3: $\in$ R=R$\in$ =R

I_4: $\in$*= $\in$ & ϕ*= $\in$ I_5:R+R=R I_6: R* R*=R*

I_7: RR*=R*R I_8: (R*)*=R* I_9: $\in$+RR*=R*=$\in$+R*R

I_{10}: (PQ)*P=P(QP)* I_{11}:(P+Q)*=(P*Q*)* =(P*+Q*)*

I_{12}: (P+Q)R=PR+QR,R(P+Q)=RP+RQ

Arden's Theorem: If P and Q are RE's over ∑ and if P does not contain $\in$, then R=Q+RP has a unique solution given by R=QP*.

This theorem is used for simplifying regular expressions.

Proof:

R=Q+RP ---------> 1

Substitute R=QP* in equation 1

R=Q+QP*P

=Q(+P*P)

=QP*(since +R*R=R*)

Therefore proved

Uniqueness

R=Q+RP

Substitute R=Q+RP in equation 1

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

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

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

$=Q+QP+QP^2+QP^3+RP^4 = Q+QP+\ldots QP^n+RP^{n+1}$

$=Q+QP+ QP^n+(QP^*)P^{n+1}$ (since R+QP*)

$= Q(+P+P^2+\ldots P^n+RP^{n+1}) = (+P+P^2+..P^n+P^*P^{n+1})$

= QP*

Example: Prove that (1+00*1)+(1+00*1)(0+10*1)*(0+10*1)= 0*1(0+10*1)*

Sol:

L.H.S (1+00*1)+(1+00*1)(0+10*1)*(0+10*1)

= (1+00*1) (∈+(0+10*1)*(0+10*1))

= (1+00*1) (0+10*1)* (∵∈ +R*R=R*)

=(∈.1+00*1)(0+10*1)* (∵ ∈.R=R)

=(∈+00*)1(0+10*1)*

=0*1(0+10*1)* (∵ ∈+R*R=R*)

=***R.H.S***

Hence proved.

Designing RE's

1) Construct a regular expression for the language accepting strings of length exactly 2 over ∑={a,b}

Sol: L= {aa,ab,ba,bb}

R=aa+ab+ba+bb

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

=(a+b)(a+b)

2) Construct a regular expression for the language accepting strings of length atleast 2 over ∑={a,b}

Sol: L= {aa,ab,ba,bb,aaa,aba,….}

Therefore R=(a+b)(a+b)(a+b)*

3) Construct a regular expression for the language accepting strings of length atmost 2 over ∑={a,b}

Sol: L={∈,a,b,aa,ab,ba,bb}

R=∈+a+b+aa+ab+ba+bb

=∈+a(∈+b+a)+b(∈+a+b)

=(∈+a+b)(∈+a+b)

4) Write the regular expression for the set of string's of 0's and 1's whose 10^{th} symbol from the right end is 1.

Sol: $(0+1)^*\ 1\ (0+1)^9$

5) Write the regular expression for the set of string's of 0's and 1's whose 5^{th} symbol from the right end is 1.

Sol: $(0+1)^* 1 (0+1)^4$

6) Write the regular expression for the set of string's of 0's and 1's that contains 101 as sub string.

Sol: (0+1)* 101 (0+1)*

7) Write the regular expression for the set of string's of a's and b's that contains bba as sub string.

Sol: (a+b)* bba (a+b)*

- RE containing even number of 0's.

  *RE = (00)**
- RE containing odd number of 0's.

  RE = (00).0*
- RE to generate a string with any number of 0's followed by any 1's ends with 011.

  RE = (0+1) 011*
- RE to generate a string with even number of a's followed by odd number of b's.

  RE = (aa)(bb)*b*
- RE to generate a string with any number of 0's and 1's and having atleast one pair of consecutive 0's

  RE = (0+1) 00 (0+1)**

- RE to generate a string of symbols, having even number of characters.

  *RE = (ab)**
- RE to generate a string, containing odd no of a's and odd number of b's.

  RE = a(aa) . (bb)*b*
- Set of words of the form: one a followed by some number of b's.

  *RE = ab**
- Set of three-lettered words, starting with b over $\Sigma = \{a, b\}$

  RE = baa + bab + bba + bbb

  Write the regular expression for the set of string's of 0's and 1's whose 5th symbol from the right end is 1.

Languages associated with regular expressions:

Regular expressions can be used to describe some simple languages. If R is a regular expression, we will let L(R) denote the language associated with R.

1. Ø is a regular expression denoting the empty set
2. ε is a regular expression denoting {ε}
3. For every a ∈ ∑, a is a regular expression denoting {a}.

if r1 and r2 are regular expressions, then:

1. L(r1 + r2) = L(r1) ∪ L(r2)
2. L(r1.r2) = L(r1).L(r2)
3. L((r1)) = L(r1)
4. L(r1*) = (L(r1))*

Examples: *Form the string set for the regular expression given below.*

- **1^*0**

$1^*0 = \{\in, 1, 11, 111, ...\} . 0$

$= \{0, 10, 110, 1110, ...\}$

- **$(100^+)^*$**

$(100^+)^* = (10. \{0, 00, 000, ... \})^*$

$= \{100, 1000, 10000, ...\}^*$

$= \{\in, 100, 100100, 100100100, 1000, 10001000, ...\}$

- **$(0+1)^*011$**

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

$= 0^*011 \cup 1^*011$

$= \{\in, 0, 00, ...\} 011 \cup \{\in, 1, 11, ...\} 011$

$= \{011, 0011, 00011, ...\} \cup \{011, 1011, 11011, ...\}$

$= \{011, 0011, 00011, 1011, 11011, ...\}$

Examples: Find the language, for the regular expressions given below

- **$R = a^*(a+b)$**

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

$= L(a^*). L(a+b)$

$= \{\in, a, aa, aaa, ...\} . \{a,b\}$

$= \{a, aa, aaa, b, ab, aab, aaab, ...\}$

- **$R = (a+b)^* (a+bb)$**

$L(R) = L((a+b)^*.(a+bb))$

$= L(a+b)^*. L(a+bb)$

$= (L(a+b))^*. L(a+bb)$

$= (L(a)^* \cup L(b)^*) .(L(a) \cup L(bb))$

$= ((L(a))^* \cup (L(b))^*) . (L(a) \cup L(bb))$

$= (\{\in, a, aa, aaa, ...\} \cup \{\in, b, bb, bbb, ...\}) . (\{a,bb\})$

$= \{\in, a, aa, b, bb, ...\}.\{a,bb\}$

$= \{a, aa, aaa, bb, abb, bbb, bbbb, ...\}$

Examples: Find the regular expressions for given languages below

- **L(R) = {a,b,ab,ba,abb,baa,…}**

R = ab* +ba* because $L(R) = L(ab^*+ba^*)$

$= L(ab^*) \cup L(ba^*)$

$= \{a\} . \{\in, b, bb\} \cup \{b\}. \{\in, a, aa\}$

$= \{a, ab, abb\} \cup \{b, ba, baa\}$

$= \{a, b, ab, ba, abb, baa\}$

- **L(R) = {*0,1, 10, 11, 100, 101, 110, 111,…}***

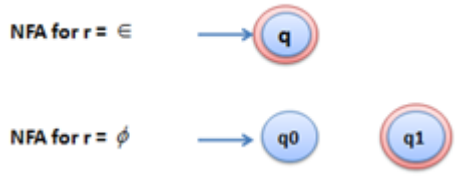
R = *0+1(0+1)* because* *L(R) = L(0+1(0+1)*)*
= L(0) U (L(1) . L(0+1))*
= {0} U ({1}. L(0) U L(1*))*
= {0} U ({1}. {∈,0, 00} U {∈,1,11})
= {0} U ({1}. {∈,0 ,00, 1, 11})
= {0} U {1, 10, 100, 11, 111}
= {0, 1, 10, 100, 11, 111}

Finite automata & Regular Expressions

Conversion of RE's to NFA with epsilon moves

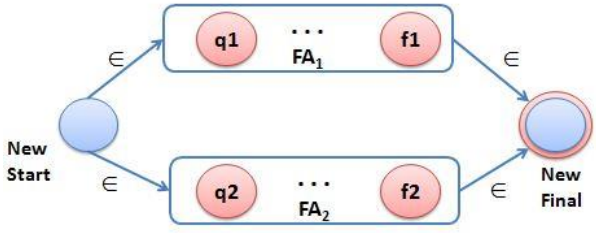
For every regular expression r, there exists an NFA with ∈-transitions that accepts L(r).
Any regular expression can be expressed in the form of NFA with ∈ moves. To do so we have to follow the steps given below.
Basic steps:

r=∈ r=ϕ



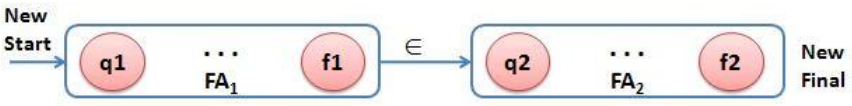
r=a

r=r1+r2 i.e L1+L2



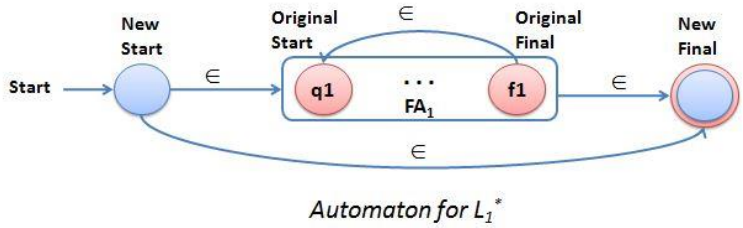
Automaton for L_1+L_2

r=r1r2 i.e L1L2



Automaton for L_1 L_2

r=r* i.e L1*



Automaton for L_1^*

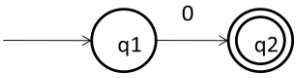
Example-1: R=00*+1

Sol: We can construct an equivalent e-NFA for the given RE by following the basic steps.

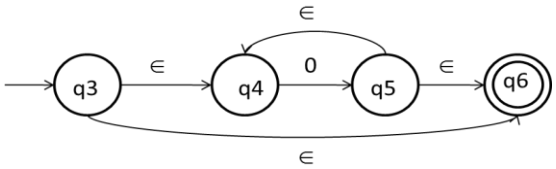
R= 00*+1 $\Rightarrow$ R=R1+R2 where R1=00* and R2=1.

R1=R3R4 where R3=0 & R4=0*

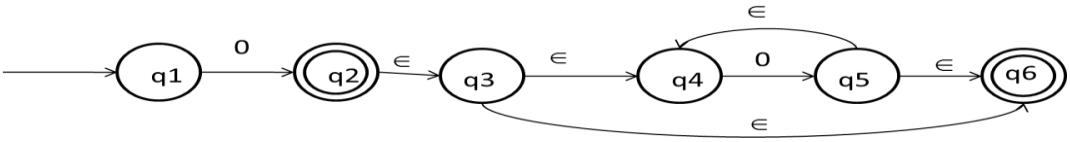
R3=0



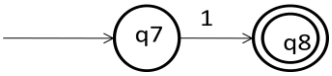
R4=0*



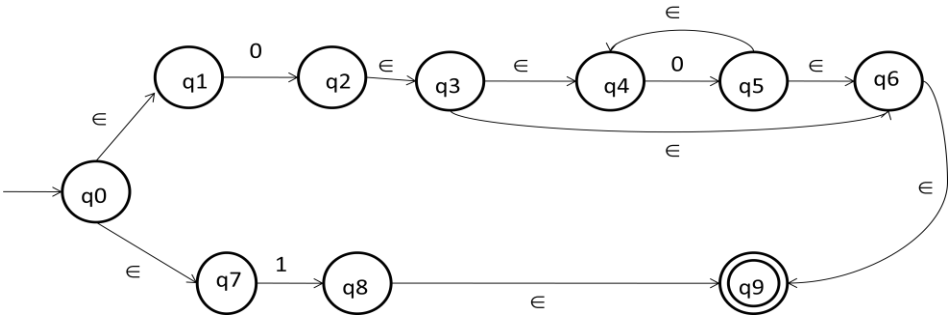
R1=R3.R4



R2=1



R=R1+R2

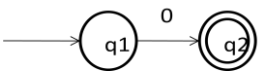


Example-2: R=(0+01)*

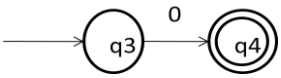
Sol: R=(0+01)*

R=(R1+R2)* where R1=0, R2=01 ,& R2=R3R4 where R3=0,R4=1

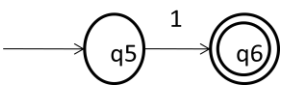
R1=0



R3=0

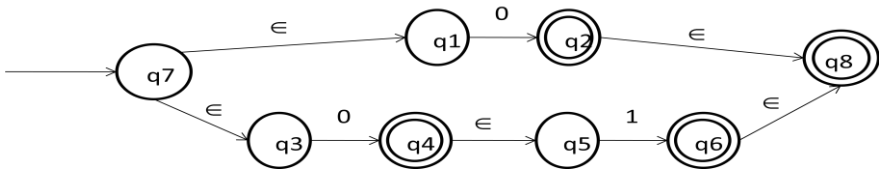


R4=1

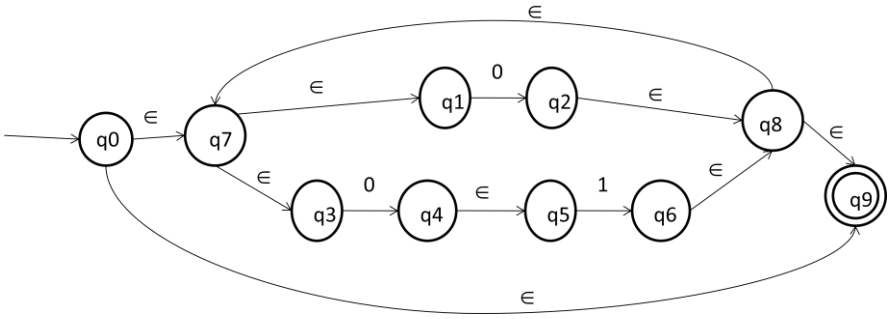


R2=R3R4

R1+R2



(R1+R2)*



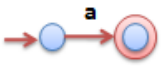
Example-3: Construct an NFA for r = (a+bb)*.ba*.

Sol: Let r = (a+bb)*. ba*

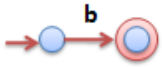
$\Rightarrow r = r_1.r_2$ where, $r_1 = (a+bb)^*$ and $r_2 = ba^*$

a. Construct a machine for r_1:

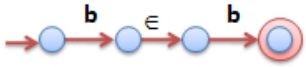
To accept a:



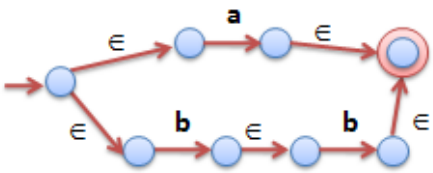
To accept b:



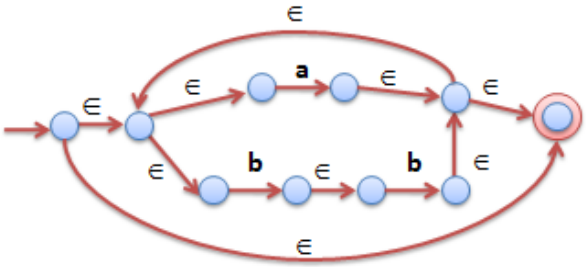
To accept bb:



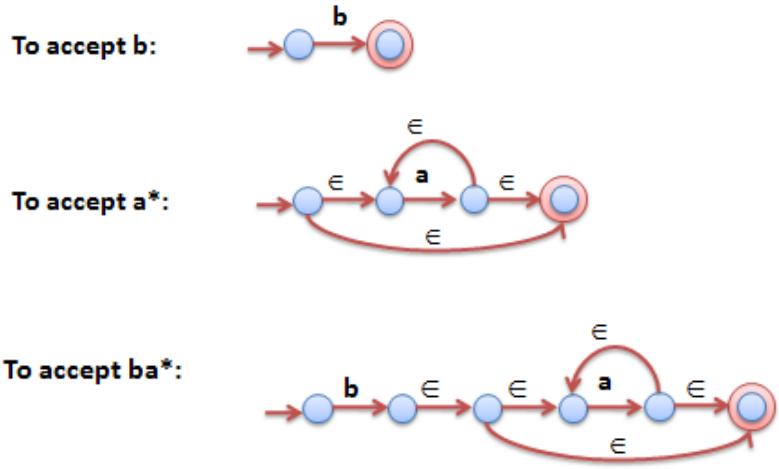
To accept a+bb:



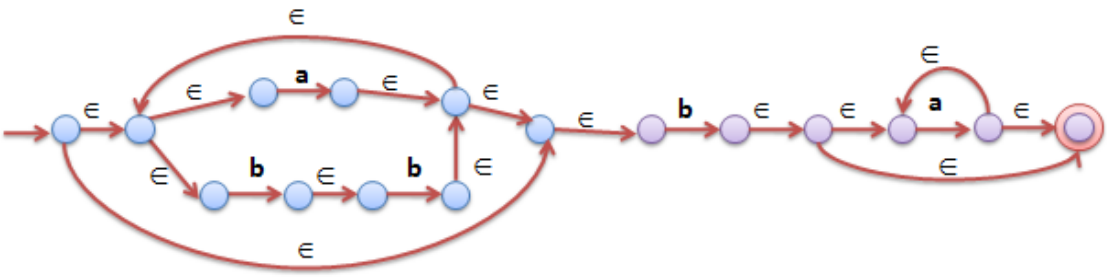
To accept (a+bb)*:



a. Construct a machine for r_2 :



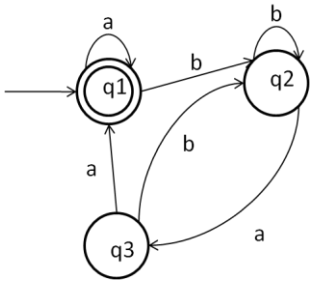
b. Construct a machine for r = $r_1.r_2$:



Conversion of Finite automata to Regular expressions

A finite automaton can be converted into a regular expression by using Arden's Theorem.

Example-1:



Sol: Now, the equations for each state can be written, by considering all edges that enter into that state.

Thus,

q1= ∈+q1a+q3a (∈ only for initial state)

q2=q1b+q2b+q3b

q3=q2a

Substitute the value of q3 in q2

q2=q1b+q2b+(q2a)b

q2=q1b+q2(b+ab)

q2=q1b(b+ab)* $(\because \text{if } R = Q + RP \text{ then } R = QP^*)$

q1=∈+q1a+q3a

substitute the value of q3 in q1

$\Rightarrow$q1=∈+q1a+q2aa

$\Rightarrow$q1=∈+q1a+[q1b(b+ab)*]aa (substitute the value of q2)

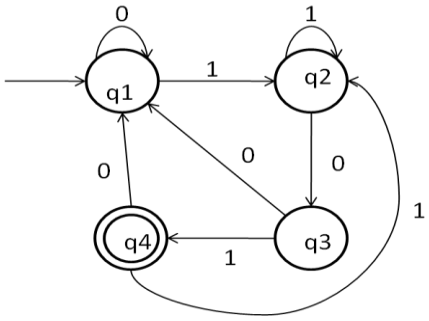
$\Rightarrow$q1= ∈+q1[a+b(b+ab)*aa]

$\Rightarrow$q1=∈ (a+b(b+ab)*aa)* $(\because \text{if } R = Q + RP \text{ then } R = QP^*)$

$\Rightarrow$ q1= (a+b(b+ab)*aa)*

Since, q_1 is the final state, the required RE is (a+b(b+ab)*aa)*

Example-2:



Sol: Now, the equations for each state can be written, by considering all edges that enter into that state.

q_1=∈+$q_1$0+$q_3$0+$q_4$0 (∈ only for initial state)

q_2=$q_1$1+$q_2$1+$q_4$1

q_3=$q_2$0

q_4=$q_3$1

substitute q_3 in q_4

q_4=$q_2$01

substitute q_4 in q_2

$q_2=q_11+q_21+q_2011$

$\Rightarrow q_2=q_11+q_2(1+011)$

$\Rightarrow q_2=q_11(1+011)^*$ (By Arden's theorem) ($\because$ if $R=Q+RP$ then $R=QP^*$)

Substitute q_3 & q_4 in q_1

$q_1=\in+q_10+q_200+q_2010$

$\Rightarrow q_1=\in+q_10+q_2(00+010)$

$\Rightarrow q_1=\in+q_10+q_11(1+011)^*(00+010)$ (substituting the value of q_2)

$\Rightarrow q_1=\in+q_1[0+1(1+011)^*(00+010)]$

$\Rightarrow q_1=\in(0+1(1+011)^*(00+010))^*$ ($\because$ if $R=Q+RP$ then $R=QP^*$)

$\Rightarrow q_1=(0+1(1+011)^*(00+010))^*$

But final state is q_4

So, $q_4=q_31$

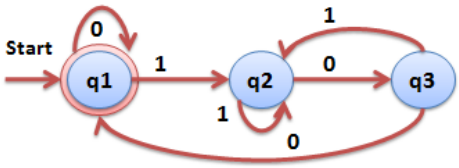
$\Rightarrow q_4=q_201$ (substituting the value of q_3)

$\Rightarrow q_4=q_11(1+011)^*01$ (substituting the value of q_2)

$\Rightarrow q_4=(0+1(1+011)^*(00+010))^*(1(1+011)^*01)$ (substituting the value of q_1)

Since, q_4 is the final state, the required RE is $(0+1(1+011)^*(00+010))^*(1(1+011)^*01)$

Example-3:



Sol: Now, the equations for each state can be written, by considering all edges that enter into that state.
Thus,

$$q_1 = q_10 + q_30 + \in \quad (\in, \text{only for initial state})$$
$$q_2 = q_11 + q_21 + q_31$$
$$q_3 = q_20$$

Substituting q_3 in q_2

$$\Rightarrow q_2 = q_11 + q_21 + q_201$$
$$= q_11 + q_2(1+01)$$
$$= q_11(1+01)^* \quad (\because \text{if } R=Q+RP \text{ then } R=QP^*)$$

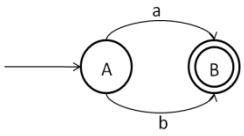
Now,

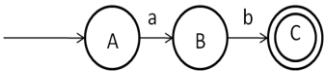
$$\Rightarrow q_1 = q_1 0 + q_3 0 + \in$$
$$= q_1 0 + q_2 0.0 + \in$$
$$= q_1 0 + q_1 1(1+01)*.00 + \in$$
$$= q_1 (0 + 1(1+01)*.00) + \in$$
$$= \in (0 + 1(1+01)*.00)* \qquad (\because R = Q + RP \text{ is } R = QP*)$$
$$q_1 = (0 + 1(1+01)*.00)*$$

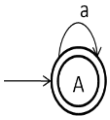
Since, q_1 is the final state, RE for DFA will be $(0+1(1+01)*.00)$

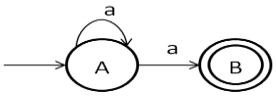
Conversion of Regular expressions to Finite automata

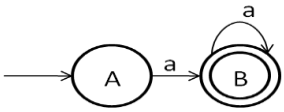
Regular expressions can be converted into finite automata. This conversion can be done by using the basic steps of conversion.

i) a+b 

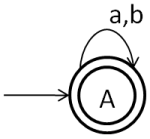
ii) ab 

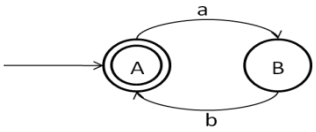
ii) a* 

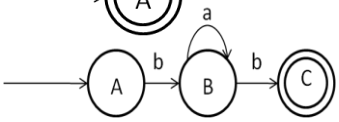
a+ = aa* 

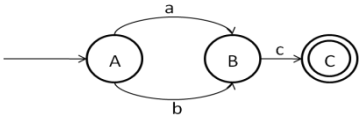
(or) a*a 

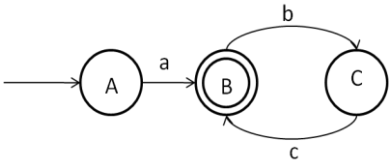
Convert the following RE's into their equivalent finite automata

1) (a+b)* 

2) (ab)* 

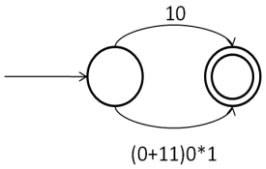
3) ba*b 

4) (a+b)c 

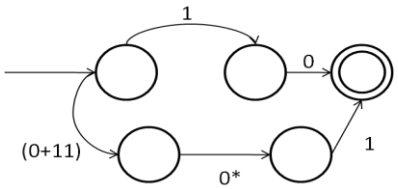
5) a(bc)* 

6) 10+(0+11)0*1

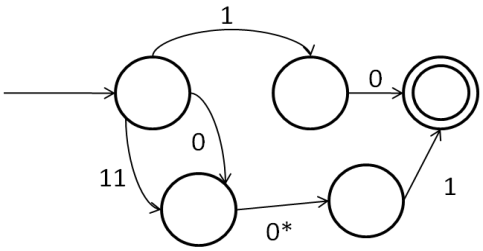
i)



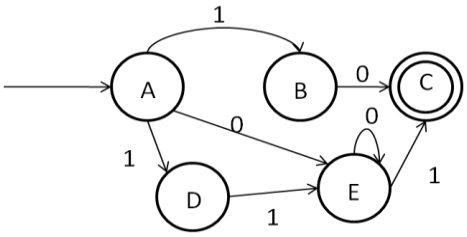
ii)



iii)



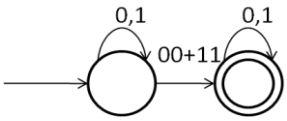
iv)



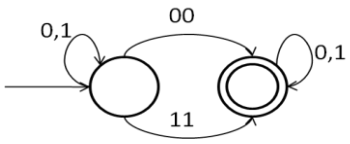
7) (0+1)*(00+11)(0+1)*

Sol:

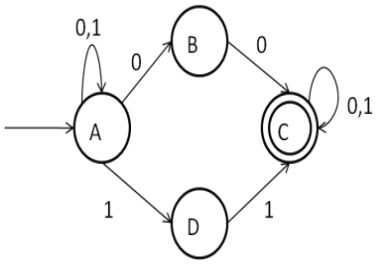
i)



ii)



iii)



Equivalence of regular expressions

Equivalence of regular expressions can be established by using any of the three methods stated below:

1) ***Set Equivalence:*** By constructing regular sets to the given RE's and showing that the two sets are equivalent.

2) ***FA Equivalence:*** By constructing Finite automata sets to the given RE's and showing that the two FA's are equivalent.

3) ***Identities:*** By using the identities.

Example: Show that (a+b)*=a*(ba*)*

Sol:

LHS		***RHS***	
(a+b)*		a*(ba*)*	
=(a*b*)*	[(P+Q)*=(P*Q*)*]	=a*b*(a*)*	
=(a*)*(b*)*		=a*b*a*	[(R*)*=R*]
=a*b*	[(R*)*=R*]	=a*b*	[(R*R*)=R*]

LHS=RHS. Hence proved.

Closure properties of Regular sets/ Regular Languages

Regular sets/Regular languages are closed under the operations union, intersection, complement, difference, concatenation, Kleene closure, Transpose (Reversal), Homomorphism and inverse Homomorphism.

Union:

The union of two regular sets is regular

Eg: RE1=a(aa)* RE2=(aa)*
Sol: Given RE1=a(aa)* RE2=(aa)*
L1={a,aaa,aaaaa,….} Strings of odd length excluding $\in$.
L2={$\in$,aa,aaaa,….} strings of even length including $\in$.
L1UL2={$\in$,a,aa,aaa,aaaa,….} =a* which is a RE.

Intersection:

The intersection of two regular sets is regular.

Eg: RE1=a(a)* RE2=(aa)*
Sol: Given RE1=a(a)* RE2=(aa)*
L1={a,aa,aaa,….} = a+
L2={$\in$,aa,aaaa,….} strings of even length including $\in$.
L1^L2={aa,aaaa,aaaaaa,….} =strings of even length excluding $\in$. =aa(aa)* which is a RE.

Complement:

The complement of a regular set is regular.

Eg: RE=(aa)*
Sol: Given RE=(aa)*
L={$\in$,aa,aaaa,…..} strings of even length including $\in$.
L^C={a,aaa,aaaaa…..} [strings not in L] = strings of odd length excluding $\in$ = a(aa)* which is a RE.

Difference:

The difference of two regular sets is regular.

Eg: RE1=a(a)* RE2=(aa)*
Sol: Given RE1=a(a)* RE2=(aa)*
L1={a,aa,aaa,….} = a+
L2={∈,aa,aaaa,….} strings of even length including ∈ .
L1-L2={a,aaa,aaaaa,….} =strings of odd length excluding ∈. =a (aa)* which is a RE.

Concatenation:
The concatenation of two regular sets is regular.

Eg: RE1=(0+1)*0 RE2=01(0+1)*
Sol: Given RE1=(0+1)*0 RE2=01(0+1)*
L1={0,00,10,000,010….}
L2={01,010,011……}
L1.L2={001,0010,0011,00010,00011…..} =(0+1)*001(0+1)* which is a RE.

Reversal/Transpose:
The transpose of a regular set is regular.

Eg: If L={0,01,100} then L^T/L^R={0,10,001}
Properties: i) $(L1+L2)^R=L1^R+L2^R$ ii) $(L1L2)^R=L2^R.L1^R$ iii) $(L^*)^R=(L^R)^*$
Eg: If RE=01*+10*.Find its transpose
Sol: Given RE=01*+10*.
$(RE)^T=(01^*+10^*)^T$ $=(01^*)^T+(10^*)^T$ $=(1^*)^T.0^T+(0^*)^T.1^T$ $=(1^T)^*.0+(0^T)^*.1$ $=1^*0+0^*1$

Kleene closure:
The iteration/closure of a regular set is regular.

Eg: RE=a
Sol: Given RE=a
L= {a}
L* ={∈,a,aa,aaa……..} = {a}* is a RE.

Homomorphism
It is a function on strings that works by substituting a particular string for each symbol. It is denoted as ***h:∑→Δ*** where ∑ is the set of input symbols and is Δ the set of output symbols.
Regular Expressions/regular sets are closed under Homomorphism.

Eg-1: h(0) = ab; h(1) = ε. What is the value of h(01010)
Sol: h(01010) = abababab.

Eg-1: h(0) = ab; h(1) = b. What is the value of h(010)
Sol: h(010) = abbab.

Inverse homomorphism

Let h be a homomorphism and L a language whose alphabet is the output language of h. Then inverse homomorphism denoted by $h^{-1}(L)$ is given by $h^{-1}(L) = \{w \mid h(w) \text{ is in } L\}$.

Eg-1: h(0) = ab; h(1) = b and L = {abab }. What is $h^{-1}(L)$.
Sol: $h^{-1}(L)$=00.

Eg-1: h(0) = a; h(1) = b and L = {abbab }. What is $h^{-1}(L)$.
Sol: $h^{-1}(L)$=01101.

Applications of Regular Expressions:

- ✓ Regular expression are used to define languages. They define the languages accepted by finite automata, exactly or used in *design of finite state systems*.
- ✓ RE offers a declarative way (algebraic definition) to express set of strings.
  Example: Set of strings that consists of single 0 followed by any number of 1s or single 1, followed by any number of 0s i.e, 01* + 10*.
- ✓ In general, the application of regular expression can be classified as follows:
  - ***Validation:*** Checking the correctness of inputs.
  - ***Search and Selection:*** Identification of a subset of items from a larger set, on the basis of pattern match, i.e, RE to locate files and lines within files.
  - ***Tokenization:*** Conversion of a string of characters into a sequence of words for later interpretation i.e, the generation of a program called ***lexical analyzer*** that performs lexical processing of its character input. The first step of compilation is lexical analysis is to convert the input from a simple sequence of characters in to a list of *tokens* of different kinds- like numerical and string constants, variable identifiers and programming language keywords.

Regular expressions are also used in:

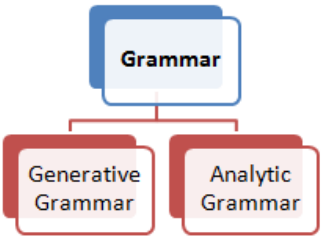
- Network protocols
- Text editors
- Sequential circuit design
- Financial analysis
- Biology-Cellular automata
- String matching algorithms
- Spell checkers

Context Free Grammars

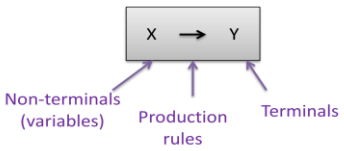
Formal Languages, Grammars, Classification of Grammars, Chomsky Hierarchy Theorem, Context Free Grammar, Leftmost and Rightmost Derivations, Parse Trees, Ambiguous Grammars, Simplification of Context Free Grammars-Elimination of Useless Symbols, E-Productions and Unit Productions, Normal Forms for Context Free Grammars-Chomsky Normal Form and Greibach Normal Form, Pumping Lemma, Closure Properties, Applications of Context Free Grammars.

FORMAL LANGUAGES:- A formal language is a set of words over a fixed alphabet. The language is finite if it contains only a finite number of words.

Grammars: A grammar is a notation for designing a language through a finite number of rules.



Structure of Grammar: Grammars not only produce natural language, but also formal ones. If L is a language over an alphabet A, then the grammar for L consists of a set of rules of the form:



Where x and y denote string of symbols taken from A and from set of grammar symbols, disjoint from A.

Production rule : The grammar rule x → y is called a production rule.

Eg: <id> → <letter><rest> is a production rule.

Start symbol: Every grammar has a special symbol called the start symbol.

- <sentence> <noun-phrase><predicate>
- <id> <letters><rest>

Here <sentence>, <id> are called start symbol.

There must be atleast one production with left side, consisting of only the start symbol.

Non-terminals and Terminals:-

- The symbols that may be replaced by other symbols are called ***non-terminals or variables.***
- The symbols that cannot be replaced by other symbols are called ***terminals.***
- Typically, terminals are smaller case letters and non-terminals are upper case letters.
- ∈ is neither a terminal nor a non-terminal.

Eg: 1) <sentence> → <noun-phrase><predicate>

<noun-phrase> → <article><noun>

<predicate> → <verb>

Non-terminals are: <noun-phrase> and <predicate>

Terminals are: <verb>, <noun> and <article>.

Eg: 2) If A = {a,b,c}, then the grammar for the language A is

$S \rightarrow \in$

$S \rightarrow aS$

$S \rightarrow bS$

Non-terminals are: S

Terminals are: a,b.

Generative Grammars:- A generative grammars consists of a set of rules for transforming strings. To generate a string in the language, one begins with a string consisting of only a single start symbol and then applies the rules successively to rewrite the string.

Formally, a grammar G is a quad-tuple

$G = (V,T,P,S)$

where,

V is a finite set of non-terminals

T is a finite set of terminals

P is the finite set of production rules, each of the form $(T \cup V)^* V (T \cup V)^* = (T \cup V)^*$

i.e, each production rule maps from one string of symbols to another, where the first string contains atleast one non-terminals symbol.

S is the start symbol, $S \in V$

The language of a generative grammar G, denoted by L(G), is defined as all those strings over T, that can be generated by starting with the start symbol S and then applying the production rules in P until no more non-terminal symbols are present.

Classification of Grammars:

In 1956, Noam Chomsky classified the generative grammars into types known as the **Chomsky hierarchy**. Each type of generative grammar distinguishes the other types in the form of production rules. The following are the types:

- Unrestricted Structure Grammar.
- Context- Sensitive Grammar.
- Context – free Grammar.
- Regular Grammar.

a) **Unrestricted or Phrase – Structure Grammars**:- An unrestricted grammar is a formal grammar, on which no restrictions are made on the left and right sides of the grammar's productions.

Formally, a generative grammar G is a quad- tuple

$$G = (V,T,P,S)$$

where,

V is a finite set of non-terminals

T is a finite set of terminals

P is the finite set of production rules, each of the form $(V \cup T)^+ \rightarrow (V \cup T)^*$

i.e, no $\in$ *on the left- hand side of any productions. However,* $\in$ *can appear on the right-hand side of the productions.*

S is the start symbol, $S \in V$

Example: Let G = (V,T,P,S) be unrestricted grammar whose production rules are

$$P = \{S \rightarrow AS \mid \in$$
$$A \rightarrow aA \mid a$$
$$aaaA \rightarrow ba\}$$

The derivation for w = bbaa is as follows:

$$S \Rightarrow AS$$
$$\Rightarrow AAS$$
$$\Rightarrow AAAS$$
$$\Rightarrow AAA$$
$$\Rightarrow aAAA$$
$$\Rightarrow aaAAA$$
$$\Rightarrow \underline{\underline{aaa}}AAA$$
$$\Rightarrow baAA$$
$$\Rightarrow baaAA$$
$$\Rightarrow b\underline{aaa}AA$$
$$\Rightarrow bbaA$$
$$\Rightarrow bbaa$$

In chomsky's hierarchy, an unrestricted grammar is called **type 0 grammar.** The language generated by this grammar is *recursively enumerable language* and the recognisers are **Turing machines.** In other words, for every unrestricted grammar G, there exists some TM capable of recognizing L(G) and vice versa.

b) Context – Sensitive Grammar: A context - sensitive grammar is a formal grammar, in which the left- hand sides and right –hand sides of any production rule may be surrounded by a context of terminal and non-terminal symbol.

Formally, a generative grammar G is a quad- tuple G = (V,T,P,S)

where,

V is a finite set of non-terminals

T is a finite set of terminals

S is the start symbol, $S \in V$

P is the finite set of production rules, each of the form $(V \cup T)^+ \rightarrow (V \cup T)^+$

i.e, no $\in$ *on the left and right hand sides of any productions. Hence, this grammar is* $\in$ *-free.*

Example: Let G = (V,T,P,S) be a context- sensitive grammar whose production rules are

$$P = \{S \rightarrow aBb$$
$$aB \rightarrow bBB$$
$$bB \rightarrow aa$$
$$B \rightarrow b\}$$

The derivation for w = aaabb is as follows:

$$S \Rightarrow \underline{aB}b$$
$$\Rightarrow \underline{bB}Bb$$
$$\Rightarrow a\underline{aB}b$$
$$\Rightarrow a\underline{bB}Bb$$
$$\Rightarrow aaaBb$$
$$\Rightarrow aaabb$$

c) Context-Free Grammar:- A context – free grammar is a grammar in which the left-hand side of each production rule consists of only a single non-terminal symbol. This restriction does not make all languages generate context-free grammar.

Formally, a generative grammar G is a quad- tuple

G = (V,T,P,S)

where,

V is a finite set of non-terminals

T is a finite set of terminals

P is the finite set of production rules, each of the form $A \rightarrow (V \cup T)^*$

i.e, a single non-terminal symbol on left-hand side and any number of terminals and non-terminals on the right hand side of production.

S is the start symbol, $S \in V$

Example: Let G = (V,T,P,S) be a context-free grammar whose production rules are

$$P = \{S \rightarrow aSa$$
$$S \rightarrow bSb$$
$$S \rightarrow a \mid b \mid \in\}$$

The derivation for w = ababa is as follows:

$$S \Rightarrow aSa$$
$$\Rightarrow abSba$$
$$\Rightarrow ababa$$

In chomsky's hierarchy, context – free grammar is called **type 2 grammar.** The language generated by this grammar is context-free language and the corresponding recognizer is **PDA.**

d) Regular Grammar:- A grammar is said to be regular, iff it is right-linear or left-linear. Informally, a regular grammar is a formal grammar with the restrictions on both left and right-hand sides of the productions.

- Left – hand side of each production rule consists of only a single non-terminal symbol.
- Right- hand side of each production rule may be nothing, or a single terminal symbol, or a single terminal symbol followed by a non-terminal symbol, but nothing else.

Formally, a generative grammar G is a quad- tuple

$G = (V,T,P,S)$

where,

V is a finite set of non-terminals, T is a finite set of terminals S is the start symbol, $S \in V$

P is the finite set of production rules, each of the form $A \to t$ *or* $A \to tB$ *or* $A \to \in$

where A.B $\in$ *V and t* $\in$ *T.*

Example: Let G = (V,T,P,S) be a regular grammar whose production rules are

$$P = \{S \to aS$$
$$S \to a\}$$

The derivation for w = ababa is as follows:

$$S \Rightarrow aS$$
$$\Rightarrow aaS$$
$$\Rightarrow aaa$$

In chomsky's hierarchy, regular grammar is called a **type 3 grammar.** The language generated by this grammar is a regular language and the corresponding recognizer is FA.

The regular grammar specified with 2 different notation, right-linear grammar and left-linear grammar.

Right Linear grammar: A grammar G = (V,T,P,S) is said to be right linear if all production are of the form.

$A \to T^*V$
$A \to T^*$

$S \to abS$
$S \to a$

$A \to xB$
$A \to x$

Left Linear grammar: A grammar G = (V,T,P,S) is said to be left linear if all production are of the form.

$A \to VT^*$
$A \to T^*$

$S \to Aab$
$A \to Aab \mid b$
$B \to a$

$A \to xB$
$A \to x$

Chomsky's Hierarchy	Grammar	Restriction on productions	Accepted
0	Unrestricted	$(V \cup T)^+ \to (V \cup T)^*$	Turing machine
1	Context- Sensitive	$(V \cup T)^+ \to (V \cup T)^+$	Linear Bounded Automata
2	Context – Free	$A \to (V \cup T)^*$	Pushdown automata
3	Regular	$A \to t$ or $A \to tB$ or $A \to \in$	Finite automata

Context-free Grammars:- A context – free grammar is a grammar in which the left-hand side of each production rule consists of only a single non-terminal symbol. This restriction does not make all languages generate context-free grammar.

Formally, a generative grammar G is a quad- tuple

G = (V,T,P,S)

where,

V is a finite set of non-terminals

T is a finite set of terminals

P is the finite set of production rules, each of the form $A \rightarrow (V \cup T)^*$

i.e, a single non-terminal symbol on left-hand side and any number of terminals and non-terminals on the right hand side of production.

S is the start symbol, $S \in V$

Example: Let G = (V,T,P,S) be a context-free grammar whose production rules are

{S $\rightarrow$aSa S $\rightarrow$bSb S $\rightarrow$a|b|$\in$}

Types of Grammar:

a) ***Linear and Non-linear Grammars:*** A grammar, with atmost one variable (non-terminal) at the right side of a production is a linear grammar, otherwise it is non-linear.

Linear grammar

Eg: 1) S $\rightarrow$aSb
S $\rightarrow \in$

Eg:2) S $\rightarrow$Ab
A $\rightarrow$aAb
A $\rightarrow \in$

Non-Linear grammar

Eg:1) S $\rightarrow$SS
S $\rightarrow \in$
S $\rightarrow$aSb

Eg:2) S $\rightarrow$aA
A $\rightarrow \in$

b) ***Right and left-Linear Grammars:***

A grammar G = {V,T,P,S} is a right-linear, if all productions are of the form

A $\rightarrow$xB or A $\rightarrow$x where A,B $\in$ V and x $\in T^*$.

Eg: S $\rightarrow$aSb
S $\rightarrow$a

A grammar G = {V,T,P,S} is a left-linear, if all productions are of the form

$A \rightarrow Bx$ *or* $A \rightarrow x$ *where* $A,B \in V$ *and* $x \in T^*$.

Eg: $S \rightarrow Ab$

$S \rightarrow Sb$

$A \rightarrow \in$

c) ***Regular Grammar:*** A grammar G = {V,T,P,S} is said to be regular, if it is either right-linear or left-linear.

Eg: $S \rightarrow abS$

$S \rightarrow a$

d) ***Non-regular grammar:*** A grammar G = {V,T,P,S} is said to be non-regular, if it is neither right-linear nor left-linear. Eg: $S \rightarrow aSb$

$S \rightarrow \in$

e) ***Simple or S-Grammar:*** A grammar G = {V,T,P,S} is said to be s-grammar, if all productions are of the form: $A \rightarrow ax$, where $A \in V$, $a \in T$ and $x \in V^*$.

Eg: $S \rightarrow aS|bSS|c$ is a simple grammar, since pairs (S,a), (S,b) occur only once in production.

f) ***Recursive Grammar:*** A grammar G = {V,T,P,S} is said to be recursive, if it contains either a recursive production or an indirectly recursive productions.

Languages of a Context-Free Grammar: The language generated by a CFG, is the set of all strings of terminals, that can be produced from the start symbol S using the productions as substitutions. A language generated by CFG, G is called a *context-free language (CFL).*

Thus, if G is a CFG, with S as a start symbol and set of terminals T, then the language of G is the set defined as

$$L(G) = \{w \mid w \in T^* \; S \overset{*}{\Rightarrow} w\}$$

Example:

1. $\{\in, ab, aabb, ..., a^n b^n, ...\}$
   - L(CFG) = $\{a^n b^n \mid n \geq 0\}$
   - CFG to derive the above strings:

     Here, any string in this language is either $\in$ or of the form **axb** for some string **x** in the laguage. The following grammar will derive any of the strings.

     $P = \{ S \rightarrow \in$

     $S \rightarrow aSb\}$

     where V = {S}, T= {a,b} and S = {S}.
   - RE = $(a*b*)$

     Derivation for the string aaabbb:

     $S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \Rightarrow aaabbb$

i.e, $S \overset{*}{\Rightarrow} \text{aaabbb}$

2. $\{\in, ab, abab, ..., (ab)^n, ...\}$

- L(CFG) = $\{(ab)^n \mid n \geq 0\}$
- CFG to derive the above strings:

  Here, any string in this language is either $\in$ or of the form **abx** for some string **x** in the language. The following grammar will derive any of the strings.

$$P = \{ S \rightarrow \in$$
$$S \rightarrow abS\}$$

  where V = {S}, T= {a,b} and S = {S}.

- RE = $(ab + ab)^*$

  Derivation for the string ababab:

$$S \Rightarrow abS \Rightarrow ababS \Rightarrow abababS \Rightarrow ababab$$

  i.e, $S \overset{*}{\Rightarrow} ababab$

Operations on Production Rules:

Suppose M and N are languages, whose grammars have disjoint sets of non-terminals. Also, assume that start symbols, for the grammars of M and N, are A and B respectively. Then, following are the rules to find new grammars generated from M and N:

- Union Rule: The language M U N starts with two productions,

$$\boxed{S \rightarrow A \mid B}$$

- Product Rule: The language MN starts with the productions

$$\boxed{S \rightarrow AB}$$

- Closure Rule: The language M* starts with the productions

$$\boxed{S \rightarrow AS \mid \in}$$

Example:

a. Union Rule

If L = $\{\in, a, b, aa, bb, ..., a^n, b^n, ...\}$ with RE = (a+b)*,

L can be written as the union of M and N i.e, L = M U N

Where $M = \{a^n \mid n \geq 0\}$ and $N = \{b^n \mid n \geq 0\}$

The grammar is

$$\boxed{\begin{array}{l} S \rightarrow A \mid B \quad (\text{union rule}) \\ S \rightarrow \in \mid aA \quad (\text{grammar for M}) \\ S \rightarrow \in \mid bB \quad (\text{grammar for M}) \end{array}}$$

b. Product Rule

If L = $\{\in, ab, aabb, aaaabbbb., aabbb, ...\}$ with RE = (ab)*, $L = \{a^m b^n \mid m, n \geq 0\}$

L can be written as the union of M and N i.e, L = MN

Where $M = \{a^m \mid m \geq 0\}$ and $N = \{b^n \mid n \geq 0\}$

The grammar is

$S \rightarrow AB$	(product rule)
$S \rightarrow \in \mid aA$	(grammar for M)
$S \rightarrow \in \mid bB$	(grammar for M)

c. Closure Rule

If L = $\{\in, aa, bb, aabb, aaaabb, bbbb, aaaabbbb.....\}$ with RE = (aa)*. (bb)*, $L = \{aa, bb\}^*$

L can be written as the union of M and N i.e, L = M*

The grammar is

$S \rightarrow AS \mid \in$	(closure rule)
$A \rightarrow aa \mid bb$	(grammar for M)

The grammar can also be written as

$S \rightarrow aaS \mid bbS \mid \in$

Design of a CFG:

a. Understand the language specification by listing the example strings in the language.
b. Determine the 'base case' productions of CFG by listing the shortest strings in the language.
c. Identify the rules for combining smaller sentences into larger ones.
d. Test the CFG obtained on a number of carefully chosen examples. All of the base cases should be tested, along with all the alternative productions. Also, whether the grammar is consistent with the strings listed in step a or not is checked.

Example: *Obtain a CFG for the lanuage of odd palindrome over the alphabet* $\Sigma = \{a, b\}$

Let the CFG, G = (V,T,P,S), where:

V = {S}

T = {a,b}

P = {

$S \rightarrow aSa$

$S \rightarrow bSb$ S is the start symbol

$S \rightarrow a \mid b$

}

Derivation for the string aaa which is an odd palindrome is

$S \Rightarrow aSa \Rightarrow aaa$

Example: *Obtain a CFG to generate the language L = {w: |w| mod 3 = 0} over* $\Sigma = \{a\}$

Let the CFG, G = (V,T,P,S), where:

V = {S}

T = {a}

P = {

$S \rightarrow \in$

$S \rightarrow aaaS$ S is the start symbol

}

Derivation for the string aaa

$S \Rightarrow aaaS \Rightarrow aaa$

Forms of derivations:

- **Leftmost derivation** – A derivation is said to be leftmost if in each step, the leftmost variable in the sentential form is replaced.
- **Rightmost derivation** – A derivation is said to be rightmost, if in each step, the rightmost variable in the sentential form is replaced.

Eg: Consider the production rules:

$S \rightarrow aAB$

$A \rightarrow bBb$

$B \rightarrow A \mid \in$

Left most derivation of abbbb

$S \Rightarrow aAB$

$\Rightarrow abBbB$

$\Rightarrow abAbB$

$\Rightarrow abbBbbB$

$\Rightarrow abbbbB$

$\Rightarrow abbbb$

Right most derivation of abbb

$S \Rightarrow aAB$

$\Rightarrow aA$

$\Rightarrow abBb$

$\Rightarrow abAb$

$\Rightarrow abbBbb$

$\Rightarrow abbbb$

*The $\overset{*}{\Rightarrow}$ notation in derivation:*Any derivation involves the application of production rules. If the production rule is applied once, then we write:

$$W_1 \Rightarrow W_2$$

If the production rule is applied more than once, then we write:

$$W_1 \overset{*}{\Rightarrow} W_2$$ Which means $W_1, W_2, W_3, W_4 \ldots$

Eg: a) Consider the productions

$S \rightarrow aSb$

$S \rightarrow \in$

Derivation for aaabbb: $S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \Rightarrow aaabbb$

Instead , we write this as $S \overset{*}{\Rightarrow} aaabbb$.

b) Consider the productions

$S \rightarrow aSb$

$S \rightarrow \in$

$S \overset{*}{\Rightarrow} \in$ $(S \Rightarrow \in)$

$S \overset{*}{\Rightarrow} ab$ $(S \Rightarrow aSb \Rightarrow a \in b \Rightarrow ab)$

$S \overset{*}{\Rightarrow} aabb$ $(S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb)$

$S \overset{*}{\Rightarrow} aaSbb$ $(S \Rightarrow aSb \Rightarrow aaSbb)$

Derivation Tree:

When deriving a string *w* from S, if every derivation is considered to be a step in the tree contruction, then the graphical display of the derivation of string *w* results in a tree structure. This is called a *derivation tree or **parse tree** or generation tree or production tree*.

A tree is said to be derivation tree if it satisfies the following requirements:

- All leaf nodes of the tree are labelled by terminals of the grammar.
- The root of the tree is labelled with start symbol of the grammar.
- The interior nodes are labelled using non-terminals.
- If an interior node has a label A, and it has n descendants with label $x_1,x_2,...x_n$ from left to right, then the production rule $A \rightarrow x_1, x_2, x_3,...x_n$ must exist in the grammar.

Formal Definition:

Let G = (V,T,P,S) be a CFG. A tree is a derivation tree for G, if and only if,

a) every vertex has a label, which is a symbol of terminal (T), non-terminal (V) or the null string $\in$.

b) the root of the tree has start symbol of the grammar as its label (S).

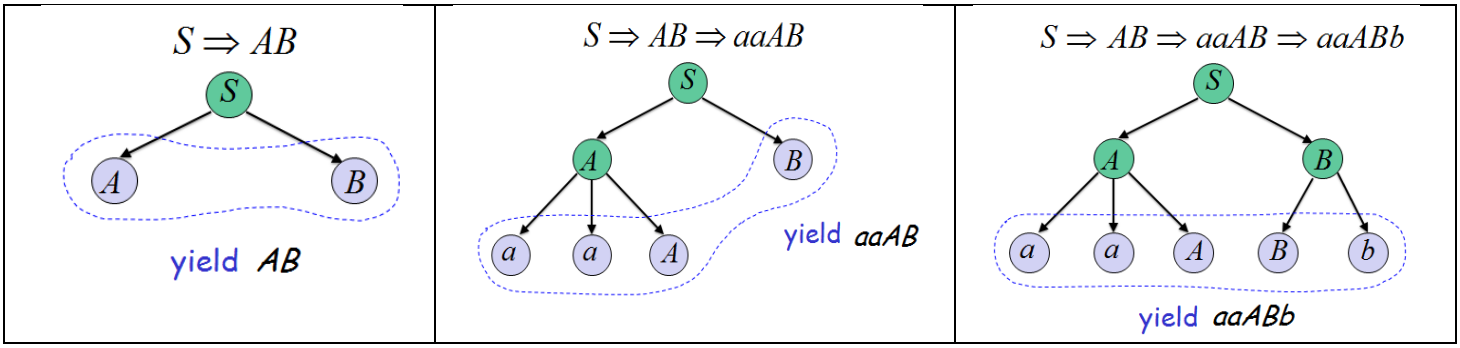
c) if a vertex is interior and has a label A, then $A \in V$.

d) if A is a label and $A \rightarrow x_1, x_2, x_3,...x_n$ is the production, then the production rule $A \rightarrow x_1, x_2, x_3,...x_n$ must exist in the grammar.

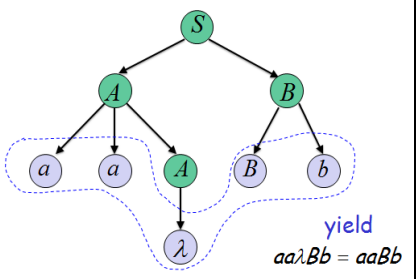
Consider Grammar

$S \rightarrow AB \quad A \rightarrow aaA \mid \lambda \quad B \rightarrow Bb \mid \lambda$

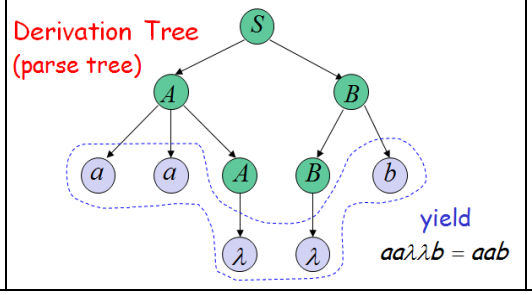
A derivation tree of G, to obtain the string w = aabbaa, is given by:



$S \Rightarrow AB \Rightarrow aaAB \Rightarrow aaABb \Rightarrow aaBb$



$S \Rightarrow AB \Rightarrow aaAB \Rightarrow aaABb \Rightarrow aaBb \Rightarrow aab$



The yield of a tree is the string of symbols obtained by only reading the leaves of the tree from left to right, without considering $\in$ -symbols. The yield is always derived from the root and is always a terminal string.

Ambiguous and unambiguous Grammars:

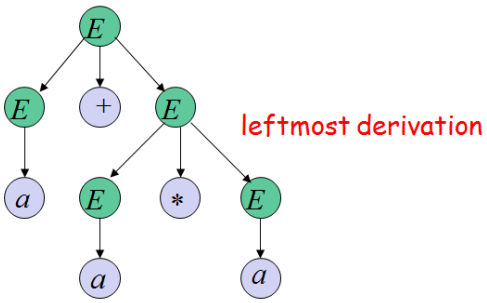
- A context-free grammar is said to be an ambiguous grammar if there exists a string which can be generated by the grammar in more than one way (i.e., the string admits more than one parse tree or, equivalently, more than one leftmost derivation)
- A grammar is said to be unambiguous, if its language has exactly one parse tree.

Example: Show that the grammar is ambiguous $E \to E+E \mid E*E \mid (E) \mid a$

For the word $a+a*a$ has two different leftmost derivations, that correspond to different parse trees.

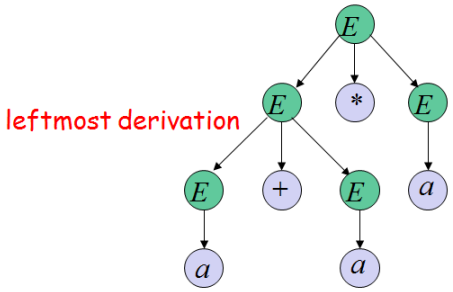
a. $E \Rightarrow E+E \Rightarrow a+E \Rightarrow a+E*E$

$\Rightarrow a+a*E \Rightarrow a+a*a$



b. $E \Rightarrow E*E \Rightarrow E+E*E \Rightarrow a+E*E$

$\Rightarrow a+a*E \Rightarrow a+a*a$

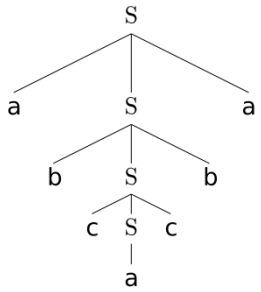


Example: Given below is the grammar for palindrome over alphabets {a,b}. Verify whether it is ambiguous or unambiguous.

$$S \rightarrow aSa \mid bSb \mid cSc \mid a$$

Solution: Consider w = abcacba

$$S \Rightarrow aSa \Rightarrow abSba \Rightarrow abcScba \Rightarrow abcacba$$



There is exactly one parse tree and hence the grammar is unambiguous.

SIMPLIFICATION OF CONTEXT FREE GRAMMAR

In a Context Free Grammar (CFG), it may not be necessary to use all the symbols in V ∪ T, or all the production rules in P while deriving sentences. Let us try to eliminate symbols and productions in G which are not useful in deriving sentences. Let G = (V,T,P,S) be a context-free grammar. Suppose that P contains a production of the form

$$A \rightarrow x_1 B x_2$$

Assume that A and B are different variables and that

$$B \rightarrow y_1 \mid y_2 \mid \cdots\cdots \mid y_n.$$

is the set of all productions in P which have B as the left side.

Let $\hat{G} = (V, T, S, \hat{P})$ be the grammar in which $\hat{P}$ is constructed by deleting

$$A \rightarrow x_1 B x_2$$

from P, and adding to it

$$A \rightarrow x_1 y_1 x_2 \mid x_1 y_2 x_2 \mid \cdots \mid x_1 y_n x_2.$$

Then $L(\hat{G}) = L(G)$.

Substitution Rule

A production $A \rightarrow x_1 B x_2$ can be eliminated from a grammar if we put in its place the set of productions in which B is replaced by all strings it derives in one step. In this result, it is necessary that A and B are different variables.

1.Eliminating Useless Productions

A symbol X is useful if:

1. If X is *generating*, i.e., X =>* w, where w ϵ L(G) and w in V_t*, this means that the string leads to a string of terminal symbols.
2. If X is *reachable* If there is a derivation S =>* αXβ =>* w, w ϵ L(G), for same α and β, then X is said to be reachable.

A number that is useful is both generating and reachable. For reduction of a given grammar G:

1. Identify non-generating symbols in the given CFG and eliminate those productions which contains non-generating symbols.
2. Identify non-reachable symbols and eliminate those productions which contain the non-reachable symbols

Example: Remove the useless symbol from the given context free grammar:

S -> aB / bX

A -> Bad / bSX / a

B -> aSB / bBX

X -> SBD / aBx / ad

Solution:

A and X directly derive string of terminals a and ad, hence they are useful. Since X is a useful symbol so S is also a useful symbol as S -> bX. But B does not derive any string w in V^*_t so clearly B is a non-generating symbol.So eliminating those productions with B in them we get

S -> bX

A -> bSX / a

X -> ad

In the reduced grammar A is a non-reachable symbol so we remove it and the final grammar after elimination of the useless symbols is

S -> bX

X -> ad

2. Null Production Removal

The productions of context-free grammars can be coerced into a variety of forms without affecting the expressive power of the grammars. If the empty string does not belong to a language, then there is a way to eliminate the productions of the form $A \rightarrow \lambda$ from the grammar.

If the empty string belongs to a language, then we can eliminate λ from all productions save for the single production $S \rightarrow \lambda$. In this case we can also eliminate any occurrences of S from the right-hand side of productions.

Procedure to find CFG without empty Productions

Step (i): For all productions $A \rightarrow \lambda$, put A into V_N.

Step (ii): Repeat the following steps until no further variables are added to V_N.

For all productions

$$B \rightarrow A_1 A_2 \ \ldots\ldots\ A_n.$$

where $A_1, A_2, A_3, \ldots\ldots, A_n$ are in V_N, put B into V_N.

To find P, let us consider all productions in P of the form

$$A \rightarrow x_1 x_2 \ \ldots\ldots\ x_m, m \geq 1$$

for each $x_i \in V \cup T$.

Example: Remove the null productions from the following grammar

S -> ABAC
A -> aA / ϵ
B -> bB / ϵ
C -> c

Solution: We have two null productions in the grammar A -> ϵ and B -> ϵ. To eliminate A -> ϵ we have to change the productions containing A in the right side. Those productions are S -> ABAC and A -> aA.
So replacing each occurrence of A by ϵ, we get four new productions

S -> ABC / BAC / BC
A -> a

Add these productions to the grammar and eliminate A -> ϵ.

S -> ABAC / ABC / BAC / BC
A -> aA / a
B -> bB / ϵ
C -> c

To eliminate B -> ϵ we have to change the productions containing B on the right side. Doing that we generate these new productions:

S -> AAC / AC / C
B -> b

Add these productions to the grammar and remove the production B -> ϵ from the grammar. The new grammar after removal of ϵ – productions is:

S -> ABAC / ABC / BAC / BC / AAC / AC / C
A -> aA / a
B -> bB / b
C -> c

3.Eliminating unit productions :

A unit production is a production A -> B where both A and B are non-terminals. Unit productions are redundant and hence should be removed. Follow the following steps to remove the unit production

Repeat the following steps while there is a unit production

1. Select a unit production A -> B, such that there exist a production B -> α, where α is a terminal
2. For every non-unit production, B -> α repeat the following step
    1. Add production A -> α to the grammar
3. Eliminate A -> B from the grammar

Example: Remove the unit productions from the following grammar

S -> AB
A -> a
B -> C / b
C -> D

D -> E

E -> a

Solution: There are 3 unit production in the grammar

B -> C

C -> D

D -> E

For production D -> E there is E -> a so we add D -> a to the grammar and add D -> E from the grammar. Now we have C -> D so we add a production C -> a to the grammar and delete C -> D from the grammar. Similarly we have B -> C by adding B -> a and removing B -> C we get the final grammar free of unit production as:

S -> AB

A -> a

B -> a / b

C -> a

D -> a

E -> a

We can see that C, D and E are unreachable symbols so to get a completely reduced grammar we remove them from the CFG. The final CFG is:

S -> AB

A -> a

B -> a / b

Left Recursion Removal

A variable A is left-recursive if it occurs in a production of the form

$$A \rightarrow Ax$$

for any $x \in (V \cup T)^*$.

A grammar is left-recursive if it contains at least one left-recursive variable.

Every content-free language can be represented by a grammar that is not left-recursive.

NORMAL FORMS

Two kinds of normal forms viz., Chomsky Normal Form and Greibach Normal Form (GNF) are considered here.

Chomsky Normal Form (CNF)

Any context-free language L without any λ-production is generated by a grammar is which productions are of the form $A \rightarrow BC$ or $A \rightarrow a$, where A, B $\in V_N$, and a $\in V_T$.

Procedure to find Equivalent Grammar in CNF
(i) Eliminate the unit productions, and λ-productions if any,
(ii) Eliminate the terminals on the right hand side of length two or more.
(iii) Restrict the number of variables on the right hand side of productions to two.

Proof:

For Step (i): Apply the following theorem: "Every context free language can be generated by a grammar with no useless symbols and no unit productions".

At the end of this step the RHS of any production has a single terminal or two or more symbols. Let us assume the equivalent resulting grammar as G = (V_N ,V_T ,P ,S).

For Step (ii): Consider any production of the form

$$A \to y_1 y_2 \ \ldots\ldots\ y_m, \quad m \geq 2.$$

If y_1 is a terminal, say 'a', then introduce a new variable B_a and a production

$$B_a \to a$$

Repeat this for every terminal on RHS.

$B_a \to a$. Let V_N' be the set of variables in V_N together with $B_a's$ introduced for every terminal on RHS.

The resulting grammar $G_1 = (V_N', V_T, P', S)$ is equivalent to G and every production in P' has either a single terminal or two or more variables.

For step (iii): Consider $A \to B_1 B_2 \ \ldots\ldots\ B_m$

where B_i's are variables and $m \geq 3$.

If $m = 2$, then $A \to B_1, B_2$ is in proper form.

The production $A \to B_1 B_2 \ \ldots\ldots\ B_m$ is replaced by new productions

$$A \to B_1 D_1,$$
$$D_1 \to B_2 D_2,$$
$$\ldots\ldots\ \ldots\ldots$$
$$\ldots\ldots\ \ldots\ldots$$
$$D_{m-2} \to B_{m-1} B_m$$

where $D_i's$ are new variables.

The grammar thus obtained is G_2, which is in CNF.

Example : Consider the *CFG* : $S \to aSb \mid \in$ generating the language $L = \{a_n b_n \mid n \geq 0\}$. we will construct a*CNF* to generate the language $L - \{\in\}$ i.e. $\{a_n b_n \mid n \geq 1\}$.

Solutions : We first eliminate $\in$-productions (generating the language $\{a_n b_n \mid n \geq 1\}$) using the procedure already described to get $S \to aSb \mid ab$.

***Step* 1 :** Introduce non-terminals *A*, *B* and replace these productions with $S \to ASB \mid AB,\ A \to a,\ B \to b$

***Step* 2 :** Introduce non-terminal *C* and replace the only production $S \to ASB$ (which is not allowable form in *CNF*) with $S \to AC$ and $C \to SB$

The final grammar in CNF is now

$S \to AC \mid AB$
$C \to SB$
$A \to a$
$B \to b$

Example :Obtain a grammar in Chomsky Normal Form (CNF) equivalent to the grammar G with productions P given

$$S \to aAbB$$
$$A \to aA \mid a$$
$$B \to bB \mid b.$$

Solution

(i) There are no unit productions in the given set of P.
(ii) Amongst the given productions, we have

$$A \to a,$$
$$B \to b$$

which are in proper form.
For $S \to aAbB$, we have

$$S \to B_a AB_b B,$$
$$B_a \to a$$
$$B_b \to b.$$

For $A \to aA$, we have

$$A \to B_a A$$

For $B \to bB$, we have

$$B \to B_b B.$$

Therefore, we have G_1 given by

$$G_1 = (\{S, A, B, B_a, B_b\}, \{a, b\}, P', S)$$

where P' has the productions

$$S \to B_a AB_b B$$
$$A \to B_a A$$
$$B \to B_b B$$
$$B_a \to a$$
$$B_b \to b$$
$$A \to a$$
$$B \to b$$

(iii) In P' above, we have only

$$S \to B_a AB_b B$$

not in proper form.

Hence we assume new variables D_1 and D_2 and the productions

$$S \to B_a D_1$$
$$D_1 \to AD_2$$
$$D_2 \to B_b B$$

Therefore the grammar in Chomsky Normal Form (CNF) is G_2 with the productions given by

$$S \to B_a D_1,$$
$$D_1 \to AD_2,$$
$$D_2 \to B_b B,$$
$$A \to B_a A,$$
$$B \to B_b B,$$
$$B_a \to a,$$
$$B_b \to b,$$
$$A \to a,$$

and $B \to b.$

Algorithm to Convert into Chomsky Normal Form −

Step 1 − If the start symbol S occurs on some right side, create a new start symbol S' and a new production S'→ S.

Step 2 − Remove Null productions. (Using the Null production removal algorithm discussed earlier)

Step 3 − Remove unit productions. (Using the Unit production removal algorithm discussed earlier)

Step 4 − Replace each production $A \to B_1 \ldots B_n$ where $n > 2$ with $A \to B_1 C$ where $C \to B_2 \ldots B_n$. Repeat this step for all productions having two or more symbols in the right side.

Step 5 − If the right side of any production is in the form $A \to aB$ where a is a terminal and A, B are non-terminal, then the production is replaced by $A \to XB$ and $X \to a$. Repeat this step for every production which is in the form $A \to aB$.

Greibach Normal Form (GNF)

In Chomsky's Normal Form (CNF), restrictions are put on the length of right sides of a production, whereas in Greibach Normal Form (GNF), restrictions are put on the positions in which terminals and variables can appear.

GNF is useful in simplifying some proofs and making constructions such as Push Down Automaton (PDA) accepting a CFG.

Definition: A context-free grammar is said to be in Greibach Normal Form (GNF) if all productions have the form

$$A \to ax \quad \text{where a} \in \text{T and x} \in \text{V}^* .$$

For a grammar in GNF, the RHS of every production has a single terminal followed by a string of variables.

Every context-free language L without $\in$, can be generated by a grammar in which every production is of the form A a α, where A is a variable and a is a terminal, and α is a (possibly empty) string of variables.

Converting to GNF is complex, even if we start with a grammar in CNF. Roughly, we expand the first variable of each production, until we get a non-terminal. But there can be cycles, where we never reach a terminal.

We "short circuit" the process, creating a product ion that introduces a terminal as the first symbol of the body and has variables following it to generate all the sequences of variables that might have been generated on the

way to generation of that terminal.

Since each use of a rule introduces exactly one terminal, a string of length n, has a derivation of exactly n steps

Lemma Define a A-production to be a production with variable A on the left. Let G = (V,T,P,S) be a CFG.

Let A → α1Bα2 be a production in P and B → β1 | β2 | … | βr be the set of B-productions. Let G1 = (V,T,P,S) be obtained from G by deleting the production A→ α1Bα2 from P and adding the productions A→α1β1α2 | α1β2α2 | α1βrα2. Then L(G) = L(G1)

Lemma Let G = (V,T,P,S) be a CFG.

Let A→ Aα1 | Aα2 | … |A αr be the set of A-productions for which A is the leftmost symbol for the right hand side. Let A → β1| β2| … | βs be the remaining A-productions.

Let G1 = (V∪ {B},T,P1,S) be the CFG formed by adding the variable B to V and replacing all the A-productions by the productions:

1) A→ βI A βiB 1 ≤ i ≤ s 2) B αI B αiB 1 ≤ i ≤ r

Then L(G1) = L(G).

Algorithm to Convert a CFG into Greibach Normal Form

Step 1 − If the start symbol S occurs on some right side, create a new start symbol S' and a new production S' → S.

Step 2 − Remove Null productions. (Using the Null production removal algorithm discussed earlier)

Step 3 − Remove unit productions. (Using the Unit production removal algorithm discussed earlier)

Step 4 − Remove all direct and indirect left-recursion.

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

Example : *A→ BB B→ AC | a C→ AB | BA | a* . We will construct an equivalent CFG in GNF.

Step 1: Renaming the nonterminal, we get

$$A_1 \rightarrow A_2A_2$$
$$A_2 \rightarrow A_1A_3 \mid a$$
$$A_3 \rightarrow A_1A_2 \mid A_2A_1 \mid a$$

Step 2 : A_1 -productions already satisfy INP.

Process A_2 and A_3-productions to enforce the INP.

First consider A_2 -productions:

Apply lemma 2 to $A_2 \rightarrow A_1A_3$ obtaining $A_2 \rightarrow A_2A_2A_3 \mid a$ .Now apply lemma 1 to eliminate left-recursion We get

$$A_2 \rightarrow a \mid A_{-2}$$
$$A_{-2} \rightarrow A_2A_3 \mid A_2A_3A_{-2}$$

which satisfy the INP property.

The resulting grammar is

$$A_1 \rightarrow A2A2$$
$$A_2 \rightarrow a \mid aA_{-2}$$
$$A_{-2} \rightarrow A_2A_3 \mid A_2A_3A_{-2}$$
$$A_3 \rightarrow A_1A_2 \mid A_2A_1 \mid a$$

Next consider A_3-productions. Applying lemma 2 to $A_3 \rightarrow A_1A_2$ we get

$$A_3 \rightarrow A_2A_2A_2 \mid A_2A_1 \mid a$$

Applying lemma 2 again on the first two A_3-productions above we get

$$A_3 \rightarrow aA_2A_2 \mid aA_{-2}A_2A_2 \mid aA_1 \mid aA_{-2}A_1 \mid a$$

Now, all productions satisfy the INP.

The resulting grammar is:

$$A_1 \rightarrow A_2A_2 \mid$$
$$A_2 \rightarrow a \mid aA_{-2}$$
$$A_{-2} \rightarrow A_2A_3 \mid A_2A_3A_{-2}$$
$$A_3 \rightarrow aA_2A_2 \mid aA_{-2}A_2A_2 \mid aA_1 \mid aA_{-2}A_1 \mid a$$

Step 3 : All A_3-productions and A_2-productions are already in GNF. Apply lemma 2 to A_1-productions,to get

$$A_1 \rightarrow aA_2 \mid aA_{-2}A_2.$$

Similarly, applying lemma 2 to A_2 -production we get

$$A_{-2} \rightarrow aA_3 \mid aA_{-2}A_3 \mid aA_3A_{-2} \mid aA_{-2}A_3A_{-2}$$

All the productions are in GNF now. So, the resulting equivalent grammar in GNF is

$$A_1 \rightarrow aA_2 \mid aA_{-2}A_2$$
$$A_{-2} \rightarrow aA_3 \mid aA_{-2}A_3 \mid aA_3A_{-2} \mid aA_{-2}A_3A_{-2}$$
$$A_2 \rightarrow a \mid aA_{-2}$$
$$A_3 \rightarrow aA_2A_2 \mid aA_{-2}A_2A_2 \mid aA_1 \mid aA_{-2}A_1 \mid a$$

Example: -

Bring the grammar G with V={S, A, B}, T={a, b} and productions P

S → A

A → a B a | a

B → b A b | b into GNF.

Solution:

1. Simplify G: No useless variables or productions, no λ-productions.

Remove unit-production S → A:

Replace A with PRODUCTION of A (after calculating transitive closure of unit-productions - but there is only one unit-dependency here A ⇒ B):

S → a B a | a new rule

2. Transform G into an equivalent grammar G' in Chomsky NF:

Substitute terminals on PRODUCTION with variables:

$S \rightarrow C_1\ B\ C_1 \mid a$ $\quad C_1 \rightarrow a$
$A \rightarrow C_1\ B\ C_1 \mid a$ $\quad C_2 \rightarrow b$
$B \rightarrow C_2\ A\ C_2 \mid b$

Break down the rules:

$S \rightarrow C_1\ D_1 \mid a$ $\quad D_1 \rightarrow B\ C_1$ $\quad C_1 \rightarrow a$
$A \rightarrow C_1\ D_1 \mid a$
$B \rightarrow C_2\ D_2 \mid b$ $\quad D_2 \rightarrow A\ C_2$ $\quad C_2 \rightarrow b$

3. Transform G' into equivalent grammar G' in Greibach NF:

a. Rename the variables to V_1 , V_2 , ... in the productions P':

$S=V_1$; $A=V_2$; $B=V_3$; $C_1=V_4$; $C_2=V_5$; $D_1=V_6$; $D_2=V_7$

$V_1 \rightarrow V_4\ V_6 \mid a$
$V_6 \rightarrow V_3\ V_4$
$V_4 \rightarrow a$
$V_2 \rightarrow V_4\ V_6 \mid a$
$V_3 \rightarrow V_5\ V_7 \mid b$
$V_7 \rightarrow V_2\ V_5$
$V_5 \rightarrow b$

b. Order the productions:

$V_1 \rightarrow V_4\ V_6 \mid a$
$V_2 \rightarrow V_4\ V_6 \mid a$
$V_3 \rightarrow V_5\ V_7 \mid b$
$V_4 \rightarrow a$
$V_5 \rightarrow b$
$V_6 \rightarrow V_3\ V_4$
$V_7 \rightarrow V_2\ V_5$

Rules for V_1 , V_2 , V_3 , V_4 , V_5 are okay.

Modify V6 -rule $V6 \rightarrow V3\ V4$:

Substitute PRODUCTIONs of V3 in V6 -rule:

$V6 \rightarrow V5\ V7\ V4 \mid b\ V4$

Substitute PRODUCTIONs of V5 in modified V6 -rule:

$V6 \rightarrow b\ V7\ V4 \mid b\ V4$ ***final V6 -rule***

Modify V7 -rule $V7 \rightarrow V2\ V5$:

Substitute PRODUCTIONs of V2 in V7 -rule:

$V7 \rightarrow V4\ V6\ V5 \mid a\ V5$

Substitute PRODUCTIONs of V4 in modified V7 -rule:

$V7 \rightarrow a\ V6\ V5 \mid a\ V5$ ***final V7 -rule***

c. *Substitute to achieve Greibach Normal Form:*

Everything already done in this case.

Grammar is in GNF:

$V1 \rightarrow V4\ V6 \mid a$
$V2 \rightarrow V4\ V6 \mid a$
$V3 \rightarrow V5\ V7 \mid b$
$V4 \rightarrow a$

V5 → b
V6 → b
V7 V4 | b V4
V7 → a V6 V5 | a V

Chomsky Normal Form	Greibach Normal Form
✦ Named after Noam chomsky	✦ Named after Sheila Greibach
✦ Put restrictions on number of symbols, on the right side of the production	✦ Put restrictions on the positions of terminals and variables.
✦ Productions of the form $A \rightarrow BC$ or $A \rightarrow a$	✦ Productions of the form $A \rightarrow ax$
✦ It yields efficient algorithms	✦ Used in parsing ✦ To construction PDA for the CFG.

Pumping Lemma In any sufficiently long string in a CFL, it is possible to find at most two short, nearby substrings that we can "pump" i times in tandem, for any integer i, and the resulting string will still be in that language.

Pumping lemma for CFLs: Let L be a CFL. Then there exists a constant n such that if $z \in L$ with $|z| \geq n$, then we can write z = uvwxy, subject to the following conditions:

1. $|vwx| \leq n$.
2. $vx \neq \in$
3. For all $i \geq 0$, we have $uv^iwx^iy \in L$.

Proof: If the string z is sufficiently long, then the parse tree produced by z has a variable symbol that is repeated on a path from the root to a leaf. Suppose $A_i = A_j$, such that the overall parse tree has yield z = uvwxy, the subtree for root A_j has yield w, and the subtree for root Ai has yield vwx.

We can replace the subtree for root A_i with the subtree for root A_j, giving a tree with yield uwy (corresponding to the case i = 0), which also belongs to L.

We can replace the subtree for root A_j with the subtree for root Ai, giving a tree with yield uv^2wx^2y (corresponding to the case i = 2), which also belongs to L.

While CFLs can match two sub-strings for (in) equality of length, they cannot match three such sub-strings.

Example 1: Consider $L = \{0^m1^m2^m \mid m \geq 1\}$.

Pick n of the pumping lemma.

Pick $z = 0^n1^n2^n$. Break z into uvwxy, with $|vwx| \leq n$ and $vx \neq \in$. Hence vwx cannot involve both 0's and 2's, since the last 0 and the first 2 are at least n + 1 positions apart.

There are two cases:

- vwx has no 2's. Then vx has only 0's and 1's. Then uwy, which would have to be in L, has n 2's, but fewer than n 0s or 1s.
- vwx has no 0's. Analogous.

Hence L is not a CFL.

Example 2: Consider $L = \{0^i1^j2^i3^j \mid i, j \geq 1\}$.

Pick n of the pumping lemma.

Pick z = 0n1 n2 n3 n. Break z into uvwxy, with $|vwx| \leq n$ and $vx \neq \in$.

Then vwx contains one or two different symbols. In both cases, the string uwy cannot be in L.
CFLs cannot match two sub-strings of arbitrary length over an alphabet of at least two symbols.

Closure Properties

Union : If L1 and If L2 are two context free languages, their union L1 ∪ L2 will also be context free.

For example,

$L1 = \{ a^n b^n c^m \mid m >= 0 \text{ and } n >= 0 \}$ and $L2 = \{ a^n b^m c^m \mid n >= 0 \text{ and } m >= 0 \}$

$L3 = L1 \cup L2 = \{ a^n b^n c^m \cup a^n b^m c^m \mid n >= 0, m >= 0 \}$ is also context free.

L1 says number of a's should be equal to number of b's and
L2 says number of b's should be equal to number of c's.
Their union says either of two conditions to be true. So it is also context free language.

Concatenation : If L1 and If L2 are two context free languages, their concatenation L1.L2 will also be context free.

For example,

$L1 = \{ a^n b^n \mid n >= 0 \}$ and $L2 = \{ c^m d^m \mid m >= 0 \}$

$L3 = L1.L2 = \{ a^n b^n c^m d^m \mid m >= 0 \text{ and } n >= 0\}$ is also context free.

L1 says number of a's should be equal to number of b's and
L2 says number of c's should be equal to number of d's.
Their concatenation says first number of a's should be equal to number of b's, then number of c's should be equal to number of d's.
So, we can create a PDA which will first push for a's, pop for b's, push for c's then pop for d's. So it can be accepted by pushdown automata, hence context free.

Kleene Closure : If L1 is context free, its Kleene closure L1* will also be context free.

For example,

$L1 = \{ a^n b^n \mid n >= 0 \}$

$L1^* = \{ a^n b^n \mid n >= 0 \}^*$ is also context free.

Intersection and complementation : If L1 and If L2 are two context free languages, their intersection L1 ∩ L2 need not be context free.

For example,

$L1 = \{ a^n b^n c^m \mid n >= 0 \text{ and } m >= 0 \}$ and $L2 = (a^m b^n c^n \mid n >= 0 \text{ and } m >= 0 \}$

$L3 = L1 \cap L2 = \{ a^n b^n c^n \mid n >= 0 \}$ need not be context free.

L1 says number of a's should be equal to number of b's and
L2 says number of b's should be equal to number of c's.
Their intersection says both conditions need to be true, but push down automata can compare only two. So it cannot be accepted by pushdown automata, hence not context free.
Similarly, complementation of context free language L1 which is $\sum^* - L1$, need not be context free.

Applications of Context Free Grammars:-

- ✓ Used in compilers and in particular for parsing (describing programming languages).
- ✓ CFG parsing for high speed network applications
- ✓ Data processing
- ✓ Natural Language processing

- ✓ Human activities recognition
- ✓ Used in speech recognition and also in processing spoken words.
- ✓ Multi functional radar constructions
- ✓ Extensible Markup language(XML)

Relation between Regular Grammar and Finite Automata:-

Since the language accepted by finite automata is a regular language, hence the following relation exists between regular grammar and finite automaton.

a) *Finite automaton from regular grammar*

For a given right-linear grammar G, there exists a language L(G), which is accepted by a finite automaton.

b) *Regular grammar from finite automaton*

For a given finite automaton M, there exists a right-linear grammar G such that L(M) = L(G).

c) *Regular expression from regular grammar*

For a given regular grammar G, there exists a regular expression that specifies L(G).

Finite automaton from regular grammar:-

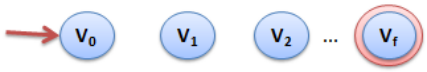
Let G = (V,T,P,S) be a right - linear grammar. Let $V = \{V_0, V_1, \ldots\}$ be the variables and productions P:

$$P = \{V_i \to a_1 a_2 .. a_m V_j\} \quad \text{or} \quad P = \{V_i \to a_1 a_2 .. a_m\}$$

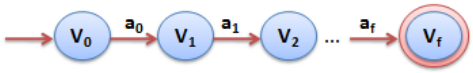
We construct a finite automaton $M = (Q, \Sigma, \delta, q_0, F)$, from these productions, by using the following steps:

Step-1: *Start symbol of grammar is the start symbol of M ($q_0 = V_0$).*

Step-2: *Each variable V_i, of G, corresponds to a state in M.*



Step-3: *For each production of the form $P = \{V_i \to a_1 a_2 .. a_m V_j\}$, if the transition of M are of the form $S(V_i, a_1 a_2 \ldots a_m) = V_j$, add the transitions and intermediate states.*



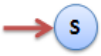
Step-3: *For each production of the form $P = \{V_i \to a_1 a_2 .. a_m\}$, if the transition of M are of the form $(V_i, a_1 a_2 \ldots a_m) = V_f$, add the transitions and intermediate states.*

Example*: Construct finite automaton to accept the language generated by the following grammar:*

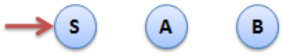
$$S \to aA \mid \in$$
$$A \to aA \mid bB \mid \in$$
$$B \to bB \mid \in$$

Sol: Let G be a right linear grammar and M the finite automata.

Step - 1: The start symbol of G is the start symbol for M i.e, q_0 = {S}

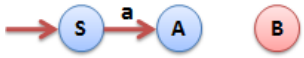


Step - 2: Every variable V = {S, A, B} of G, corresponds to a state in M.

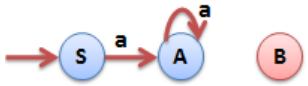


Step - 3: Compute for all productions of the form $P = \{V_i \rightarrow a_1 a_2 .. a_m V_j\}$

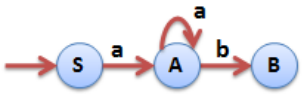
a) $S \rightarrow aA$



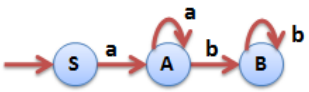
b) $A \rightarrow aA$



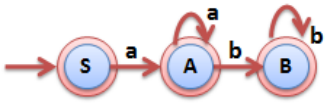
c) $A \rightarrow bB$



d) $B \rightarrow bB$



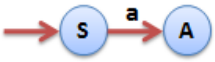
There are productions of the form $S \rightarrow \in, A \rightarrow \in$ and $B \rightarrow \in$ Thus, final M is



Regular grammar from finite automaton:

Let $M = (Q, \Sigma, \delta, q_0, F)$ be a finite automata, Where Q is the finite number of states and Σ is the set of input symbols.

Step-1: *For any transition of the form,*

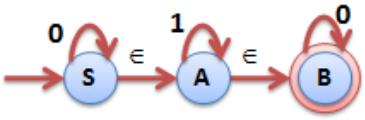


i.e, $\delta(S, a) = A$ *, the corresponding production is* $S \rightarrow aA$

Step-2: *For any final state of M,*

the production added is $A \rightarrow \in$

Example: Obtain the regular grammar for the FA given in figure



Sol: For each transition of FA, the corresponding productions are shown below:

$q_i \overset{a}{\rightarrow} q_j$	**Productions**

$S \xrightarrow{0} S$	$S \to 0S$
$S \xrightarrow{\in} A$	$S \to A$
$A \xrightarrow{1} A$	$A \to 1A$
$A \xrightarrow{\in} B$	$A \to B$
$B \xrightarrow{0} B$	$B \to 0B$
$q_f = B$	$B \to \in$

Thus, G = (V, T, P, S) is the required grammar, where V = {S,A,B} , T = {0,1} , S = {S} , F = {B} and

P = { $S \to 0S$ $S \to A$ $A \to 1A$ $A \to B$ $B \to 0B$ $B \to \in$ }

Regular expression from regular grammar: Let G = (V,T,P,S) be a right linear grammar. A regular expression R, that specifies L(G), can be directly obtained as follows:

a. *Replace the $\to$ symbols in the grammar's productions with "=" symbol to get a set of equations.*
b. *Convert the equation of the form A = aA|b as A = a^*b.*
c. *Solve the set of equations obtained, to obtain the value of the variable S (where S is the start symbol of the grammar). The result is the regular expression, specifying L(G).*

Example: Obtain the regular expression, for the grammar given below:

$$S \to 01B \mid 0$$
$$B \to 1B \mid 11$$

Sol:

Step – 1: Replace $\to$ by = in the above productions, so that,

$S \to 01B \mid 0$ ..... (1)

$B \to 1B \mid 11$ ..... (2)

Step – 2: $B = 1B \mid 11 \Rightarrow$ B = 1^*11

Step – 3: Substituting for B in eq.(1), we get $S = 01B \mid 0 \Rightarrow S = 011^*11 \mid 0$

Pushdown Automata

Pushdown Automata, Definition, Model, Graphical Notation, Instantaneous Description Language Acceptance of pushdown Automata, Design of Pushdown Automata, Deterministic and Non – Deterministic Pushdown Automata, Equivalence of Pushdown Automata and Context Free Grammars Conversion, Two Stack Pushdown Automata, Application of Pushdown Automata.

Pushdown Automata (PDA):- A PDA is a computational model that is used to solve any problem that has an algorithmic solution and requires a single stack memory (infinite memory) as storage. The PDA can accept or recognize context-free languages (CFLs). A set of all regular languages is a proper subset of CFL; hence PDA can also accept regular languages. They can only be used to implement LIFO.

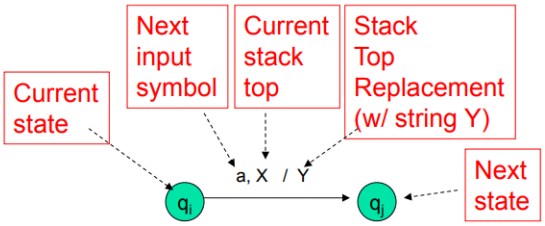
Definition:- A Pushdown Automaton is a nondeterministic finite state automaton (NFA) that permits ε-transitions and a stack (or) Pushdown Automata is a finite automata with extra memory called stack which helps Pushdown automata to recognize Context Free Languages.

Basically a pushdown automaton is –"**Finite state machine**" + "**a stack**"

A Pushdown Automata (PDA) can be defined as :

$M = (Q, \sum, \Gamma, \delta, q_0, Z_0, F)$

- Q is the set of states
- ∑ is the set of input symbols
- Γ is the set of pushdown symbols (which can be pushed and popped from stack)
- q_0 is the initial state
- Z_0 is the initial pushdown symbol (which is initially present in stack)
- F is the set of final states
- δ is a transition function which maps $Q \times \{ \sum \cup \varepsilon \} \times \Gamma$ into $Q \times \Gamma^*$.
  In a given state, PDA will read input symbol and stack symbol (top of the stack) and move to a new state and change the symbol of stack.



Pushdown Automata has 2 alphabets:

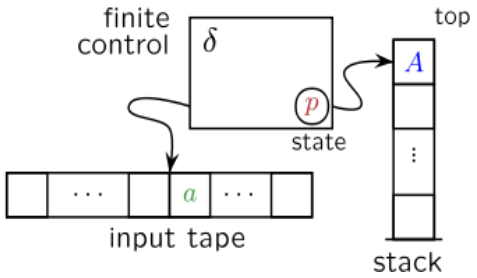
a) An input alphabet ∑ (for input strings)
b) A stack alphabet Γ (stored on the stack)

A move on PDA may indicate:

- An element may be added to the stack
- An element may be deleted from the stack: $(q,a,Z_0) = (q, \varepsilon)$
- There may or may not be change of state.

Model:- Pushdown Automata consists of three components:

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



Input tape:- It consists of linear configuration of cells each of which contains a character from the input alphabet.This tape can be moved one cell at a time to the left.

Finite control:- The control unit has some pointer (head) which points the current symbol that is to be read. The head positioned over the current stack element can read and write special stack characters from the position. The control unit contains both tape head and stack head and finds itself at any moment in a particular state.

Stack structure:- Stack is a sequential structure that has a first element and grows in either direction from the other end. The current stack element is always the top element of the stack; hence, the name 'stack'

A stack does two operations –

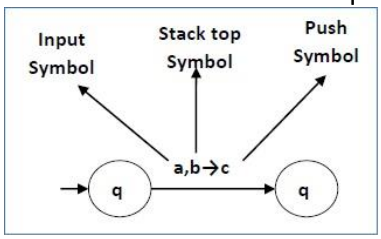
- ***Push –*** a new symbol is added at the top.
- ***Pop*** – the top symbol is read and removed.

A PDA may or may not read an input symbol, but it has to read the top of the stack in every transition.

Graphical Notation:-

- The nodes correspond to the states of the PDA.
- An arrow labeled Start indicates the unique start state.
- Doubly circled states are accepting states.
- Edges correspond to transitions in the PDA.

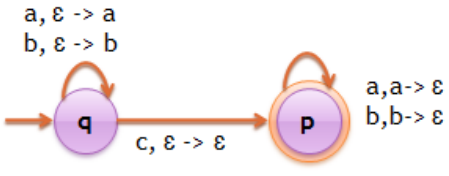
The following diagram shows a transition in a PDA from a state q1 to state q2, labeled as a,b → c –



This means at state q1, if we encounter an input string 'a' and top symbol of the stack is 'b', then we pop 'b', push 'c' on top of the stack and move to state q2.

Let M = (Q, ∑, Γ, δ ,q_0, Z_0, F) be a PDA where Q = {p,q}, ∑ = {a,b,c}, Γ = {a,b} ,q_0=q_0, F = {p} and δ is given by the following equations:

δ (q,a,ε) = {(q,a)}
δ (q,b,ε) = {(q,b)}
δ (q,c,ε) = {(p,ε)}
δ (p,a,a)= {(p,ε)}
δ (p,b,b) = {(p,ε)}



It is very hard to visualize what M does. The transistion diagram helps us to understand. We use the notation a,b→ c on a transistion between states q_i and q_j to mean δ (q_i,a,b) = {(q_j,c)}

In words, M works as follows: It stays in state q pushing input symbols 'a' and 'b' onto the stack until it encounters a 'c'. Then it moves to state p, in which it repeatedly pops a and b symbols off the stack provided the input symbol is identical to that on top of the stack. If at the end of the string, the stack is empty, then the machine accepts. If the input string has no 'c' in it, then M will never leave state q. Once the input is finished, we are not in final state and so cannot accept the string.

Instantaneous Description:- Instantaneous Description (ID) is an informal notation of how a PDA "computes" a input string and make a decision that string is accepted or rejected. Intuitively, PDA goes from configuration to configuration, in response to input symbols (or sometimes ε), the PDA's configuration involves both the state and the contents of the stack. Being arbitrarily large, the stack is often the more important part of the total configuration of the PDA at any time. It is also useful to represent as part of the configuration the portion of the input that remains.
Thus we can simplify makes a convenient notation for describing the successive configurations of a pushdown automata during the processing of a string. The relevant factors of pushdown configuration notation by a triple (q, w, γ) where;

- q is the current state of the control unit
- w is the unread part of the input string or the remaining input alphabets
- γ is the current contents of the PDA stack

Conventionally, we show leftmost symbol indicating the top of the stack γ and the bottom at the right end. Such a triple notation is called an instantaneous description or ID of the pushdown automata. In short an instantaneous Description (ID) or configuration of a pushdown automaton (PDA) describes its execution status at any time.

A move from one instantaneous description to another will be denoted by the symbol '$\vdash$' thus

$$(q_0, aw, z_0) \vdash (q_1, w, yz_0)$$

is possible if and only if

$$\delta(q_0, a, z_0) \in (q_1, yz_0).$$

Moves involving an arbitrary number of steps will be denoted by '$\vdash^*$'. On occasions where several automata are under consideration we will use $\vdash_M$ to emphasize that the move is made by the particular automaton M.

Example 1: Write down the IDs or moves for input string w = "aabb" of PDA as M = ($\{q_0, q_1, q_2\}$, {a, b}, {a, b, Z_0}, δ, q0, Z_0, $\{q_2\}$), where δ is defined by following rules:

$\delta(q_0, a, Z_0)$	$= \{(q_0, aZ_0)\}$	Rule (1)
$\delta(q_0, a, a)$	$= \{(q_0, aa)\}$	Rule (2)
$\delta(q_0, b, a)$	$= \{(q_1, \lambda)\}$	Rule (3)
$\delta(q_1, b, a)$	$= \{(q_1, \lambda)\}$	Rule (4)
$\delta(q_1, \lambda, Z_0)$	$= \{(q_2, \lambda)\}$	Rule (5)
$\delta(q_0, \lambda, Z_0)$	$= \{(q_2, \lambda)\}$	Rule (6)

Also check string w is accepted by PDA or not?
Solution: Instantaneous Description for string w = "aabb"

$(q_0, aabb, Z_0)$	$\vdash (q_0, abb, aZ_0)$	as per Rule (1)
	$\vdash (q_0, bb, aaZ_0)$	as per Rule (2)

$|-(q_1, b, aZ_0)$ as per Rule (3)
$|-(q_1, \lambda, Z_0)$ as per Rule (3)
$|-(q2, \lambda, \lambda)$ as per Rule (5)

Finally PDA reached a configuration of (q_2, λ, λ) i.e. the input tape is empty or input string w is completed, PDA stack is empty and PDA has reached a final state. So the string 'w' is **accepted**.

Example 2: Write down the IDs or moves for input string w = "aaabb" of PDA. Also check it is accepted by PDA or not?

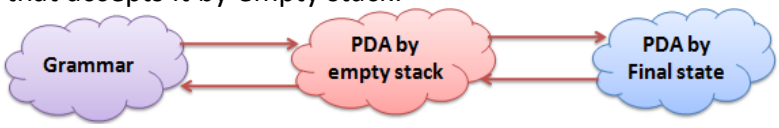
Solution: Instantaneous Description for string w = "aaabb"

$(q_0, aaabb, Z_0) |-(q_0, aabb, aZ_0)$ as per transition Rule (1)
$|-(q_0, abb, aaZ_0)$ as per transition Rule (2)
$|-(q_0, bb, aaaZ_0)$ as per transition Rule (2)
$|-(q_1, b, aaZ_0)$ as per transition Rule (3)
$|-(q_1, \lambda, aZ_0)$ as per transition Rule (3)
|- There is no defined move.

So the pushdown automaton stops at this move and the string is not accepted because the input tape is empty or input string w is completed but the PDA stack is not empty. So the string 'w' is **not accepted**.

Language Acceptance of pushdown Automata:-

The two methods axe equivalent, in the sense that a language L has a PDA A that accepts it by final state if and only if L has a PDA B that accepts it by empty stack.



Acceptance by Final State

- ✓ A PDA accepts its input by consuming it and entering an accepting state. This approach is called as "***acceptance by final state***."
- ✓ Let P = $(Q, \sum, \Gamma, \delta, q_0, Z_0, F)$ be a PDA, The language accepted by P by final state is

  $L(P) = \{w: (q_0, w, Z_0) \vdash^* (q, \varepsilon, \alpha), q \epsilon F\}$

  That is, starting in the initial ID with w waiting on the input, PDA consumes w from the input and enters an accepting state.
- ✓ The contents of the stack at that time is irrelevant.

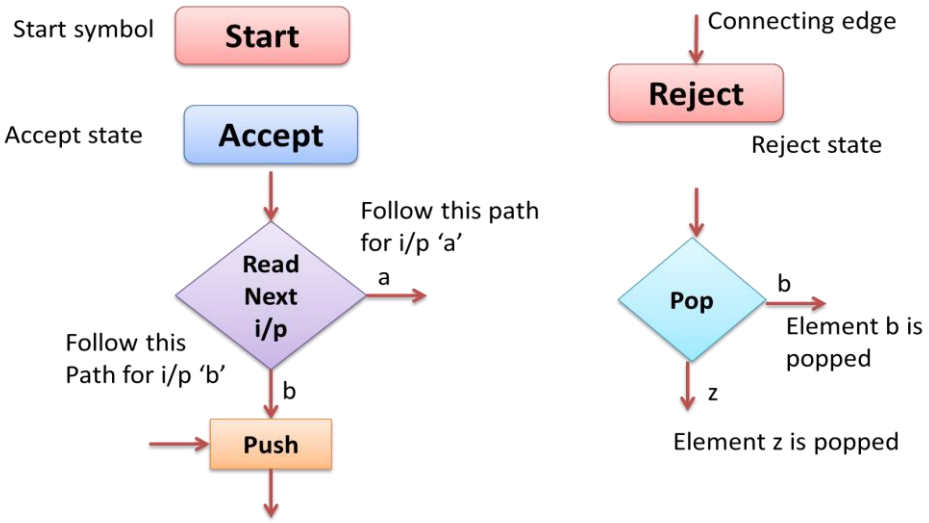
Acceptance by Empty Stack

- ✓ The other approach known as "***accepted by empty stack***". The set of strings that cause the PDA to empty its stack, starting from the initial ID.
- ✓ Let P = $(Q, \sum, \Gamma, \delta, q_0, Z_0, F)$ be a PDA, The language accepted by P by empty stack is

  $L(P) = \{w: (q_0, w, Z_0) \vdash^* (q, \varepsilon, \varepsilon)\}$

  That is, N(P) is the set of inputs w that P can consume and at the same time empty its stack.

PDA can also be shown using special symbols



Design of Pushdown Automata:-

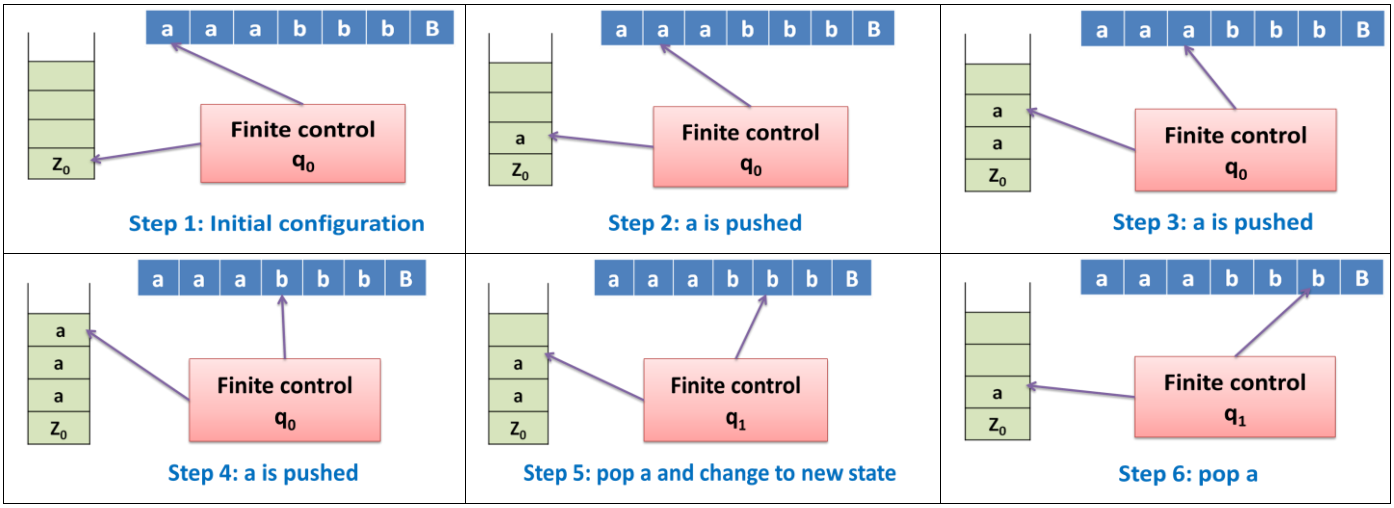
Example: **Design a PDA which accepts the language L = {$a^n b^n$ |n >=1}**

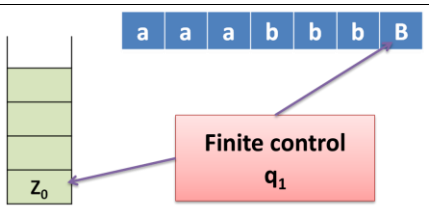
Let q_0 be the initial state, q_f final state and Z_0 bottom of the stack. Read each 'a' and push it onto the stack. Then read each b and pop out from stack for matching with a's. When all b's are read, if the stack is empty, then string is valid. If any a's are left over the stack or b's on input tape then the string is rejected.

Suppose the input is aaabbb. The steps involved are shown in fig

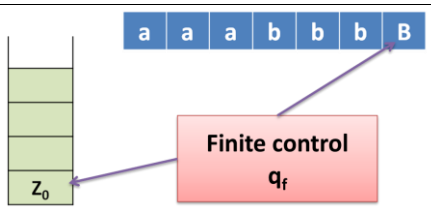
The PDA moves are the following:

$\delta(q_0,a,Z_0) = \{(q_0,aZ_0)\}$	push a
$\delta(q_0,a,a) = \{(q_0,aa)\}$	push a
$\delta(q_0,b,a) = \{(q_1,\varepsilon)\}$	**pop a and change the state**
$\delta(q_1,b,a) = \{(q_1,\varepsilon)\}$	pop a
$\delta(q_1,\varepsilon,Z_0) = \{(q_f,Z_0)\}$	change to final state q_f and halt

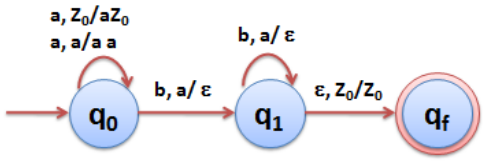


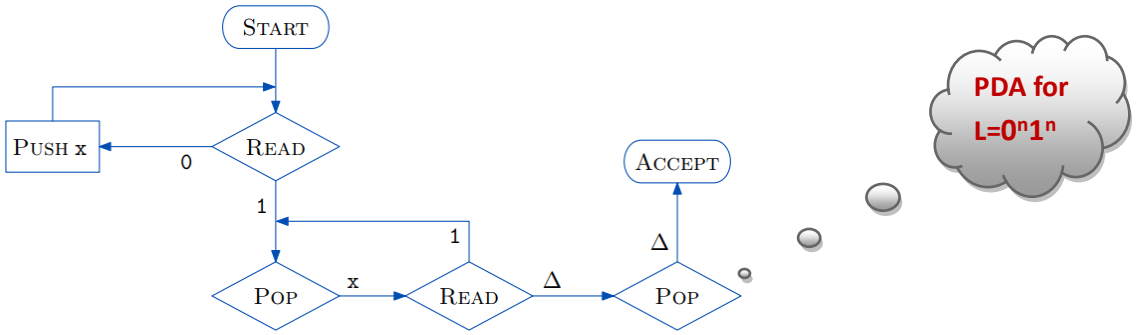


Step 7: pop a



Step 8: Change to final state and halt





Example: Design a PDA which accepts equal number of a's and b's over ∑ = {a,b}

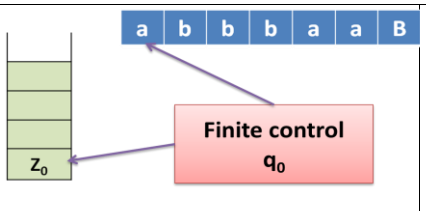
The input can start with either a or b,and they can occur in any order, as bbaaba or bababa or babbaa and so on are all valid in the language.

To design such a PDA, read the first symbol either a or b push it on the stack. Then read the next symbol. If the next symbol and top of stack are same, push it onto the stack.

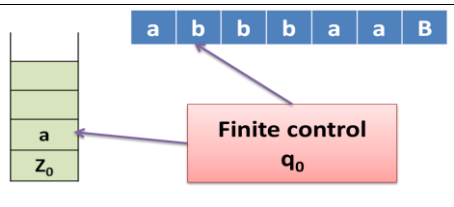
Whenever top of stack and tape symbol are different, pop off the stack. When entire tape is read, if the stack becomes empty, then string is accepted.

The PDA moves are the following:

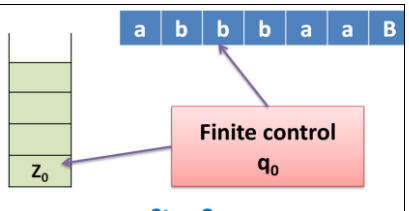
$\delta(q_0,a, Z_0) = \{(q_0,aZ_0)\}$
$\delta(q_0,b, Z_0) = \{(q_0,bZ_0)\}$
$\delta(q_0,a,a) = \{(q_0,aa)\}$
$\delta(q_0,b,b) = \{(q_0,bb)\}$
$\delta(q_0,a,b) = \{(q_0,\varepsilon)\}$
$\delta(q_0,b,a) = \{(q_0,\varepsilon)\}$
$\delta(q_0,\varepsilon, Z_0) = \{(q_f, Z_0)\}$



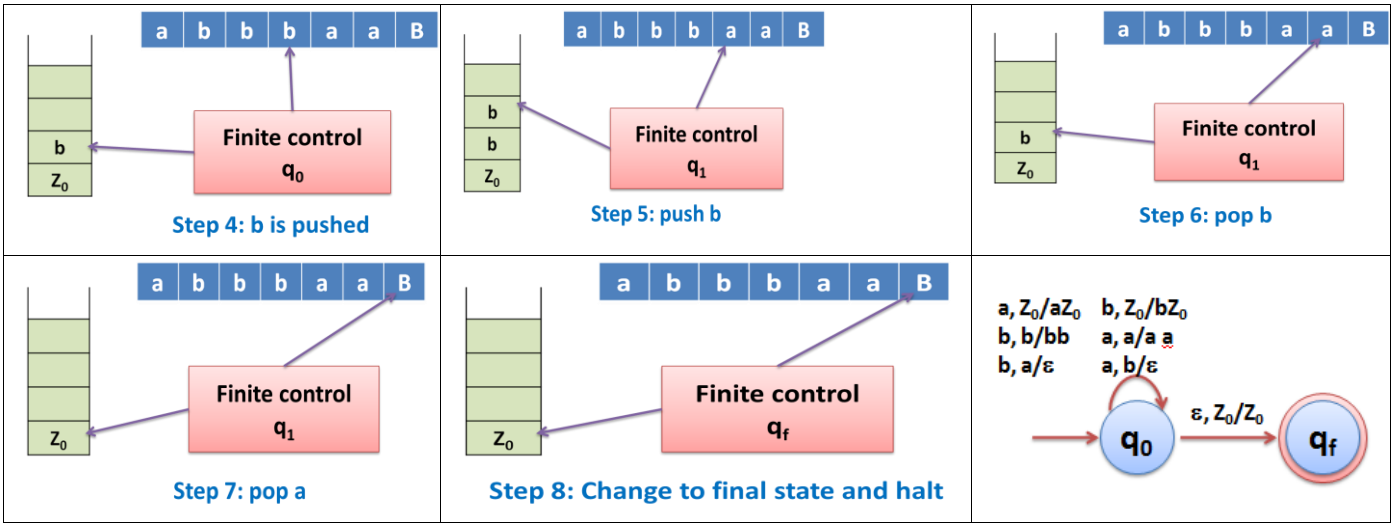
Step 1: Initial configuration



Step 2: a is pushed



Step 3: pop a



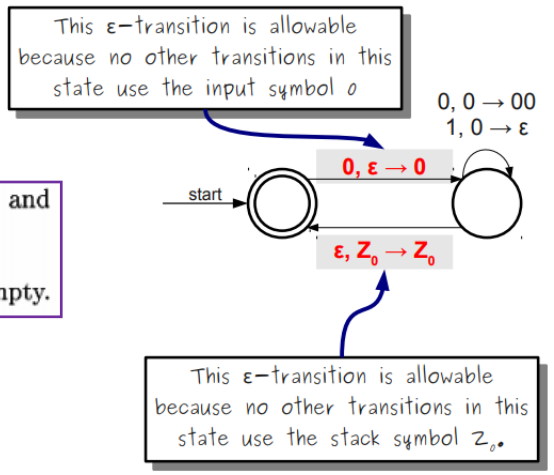
Deterministic and Non–Deterministic Pushdown Automata:-

DPDAs:- A *deterministic pushdown automaton* is a PDA with the extra property that
For each state in the PDA, and for any combination of a current input symbol and a current stack symbol, there is **atmost** one transition defined.

- In other words, there is at most one legal sequence of transitions that can be followed for any input.
- This does not preclude ε-transitions, as long as there is never a conflict between following the ε-transition or some other transition.
- However, there can be at most one ε-transition that could be followed at any one time.
- This does not preclude the automaton "dying" from having no transitions defined; DPDAs can have undefined transitions.

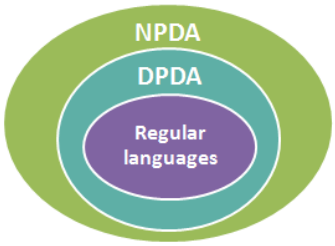
A PDA is said to be deterministic PDA if it satisfies the below conditions

1. $\delta(q, a, X)$ has at most one member for any q in Q, a in Σ or $a = \epsilon$, and X in Γ.
2. If $\delta(q, a, X)$ is nonempty, for some a in Σ, then $\delta(q, \epsilon, X)$ must be empty.



- Because DPDAs are deterministic, they can be simulated efficiently:
  - Keep track of the top of the stack.
  - Store an action/goto table that says what operations to perform on the stack and what state to enter on each input/stack pair.
  - Loop over the input, processing input/stack pairs until the automaton rejects or ends in an accepting state with all input consumed.
- If we can find a DPDA for a CFL, then we can recognize strings in that language efficiently

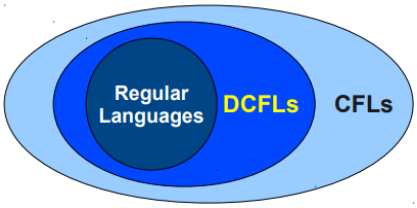
NPDAs:-



- In a PDA, if there are multiple nondeterministic choices, you cannot treat the machine as being in multiple states at once.
- Each state might have its own stack associated with it.
- Instead, there are multiple parallel copies of the machine running at once, each of which has its own stack.
- When dealing with finite automata, there is no difference in the power of NFAs and DFAs.
- NPDAs are more powerful than DPDAs
- However, when dealing with PDAs, there are CFLs that can be recognized by NPDAs that cannot be recognized by DPDAs.
- Simple example: The language of palindromes.
  - How do you know when you've read half the string?

Deterministic CFLs

- ✓ A context-free language L is called a deterministic context-free language (DCFL) if there is some DPDA that recognizes L.
- ✓ Not all CFLs are DCFLs, though many important ones are.
- ✓ Balanced parentheses, most programming languages, etc.



Equivalence of Pushdown Automata and Context Free Grammars Conversion:-

Pushdown automata and context-free grammars are equivalent in expressive power, that is, the language accepted by PDAs are exactly the context-free languages. To show this, we have to prove each of the following:

i) Given any arbitrary CFG G there exists some PDA M that accepts exactly the same language generated by G.

ii) Given any arbitrary PDA M there exists a CFG G that generates exactly the same language accepted by M.

Algorithm to find PDA corresponding to a given CFG

Input – A Context Free Grammar, G = (V, T, P, S)

Output – Equivalent Pushdown Automata, P = (Q, ∑, S, δ, q_0, I, F)

Step 1 – Convert the productions of the CFG into GNF.

Step 2 – The PDA will have only one state {q}.

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

Step 4 – All non-terminals of the CFG will be the stack symbols of the PDA and all the terminals of the CFG will be the input symbols of the PDA.

Step 5 – For each production in the form **A → aX** where a is terminal and **A, X** are combination of terminal and non-terminals, make a transition **δ (q, a, A)**.

Example: Construct a PDA equivalent to the following grammar

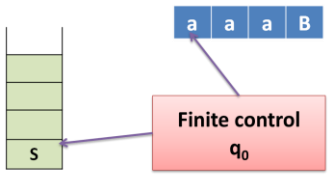
$$S \rightarrow aAA$$

$$A \rightarrow aS \mid bS \mid a$$

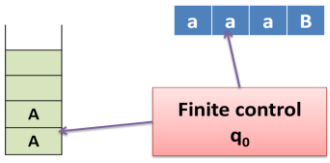
Sol:- The grammar is in GNF, Hence we can apply the rule as follows:

$S \rightarrow aAA$	corresponding	$\delta(q,a,S) = (q,AA)$
$A \rightarrow aS$	corresponding	$\delta(q,a,A) = (q,S)$
$A \rightarrow bS$	corresponding	$\delta(q,b,A) = (q,S)$
$A \rightarrow a$	corresponding	$\delta(q,a,A) = (q,\in)$

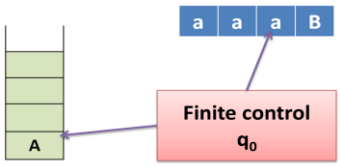
Consider the string 'aaa' accepted by the grammar and the change of state is shown in figure:



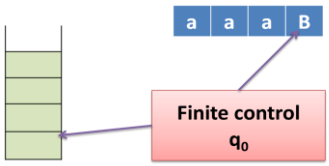
Step 1: Initial configuration



Step 2: Pop S and push AA



Step 3: Pop A



Step 4: Pop A

Example: Construct a PDA from the following CFG.
G = ({S, X}, {a, b}, P, S) where the productions are – **S → XS | ε , A → aXb | Ab | ab**

Solution:-

Let the equivalent PDA,
P = ({q}, {a, b}, {a, b, X, S}, δ, q, S)
where δ –

δ(q, ε , S) = (q, XS)
δ(q, ε , S) = (q, ε)
δ(q, ε , X) = (q, aXb)
δ(q, ε , X) =(q, Xb)
δ(q, ε , X) =(q, ab)
δ(q, a, a) = (q, ε)
δ(q, 1, 1) = {(q, ε)}

Algorithm to find CFG corresponding to a given PDA

Input – A CFG, G = (V, T, P, S)
Output – Equivalent PDA, P = (Q, ∑, S, δ, q_0, I, F) such that the non- terminals of the grammar G will be $\{X_{wx} \mid w,x \in Q\}$ and the start state will be $A_{q0,F}$.

Rule 1 – The productions for start symbol S are given by

$S \rightarrow [q_0, Z_0, q]$ for each q in Q

Rule 2 – Each move that pops a symbol from stack with transition as

$$\delta(q, a, Z_i) = (q_1, \varepsilon)$$

Induces a production as

$[q, Z_i, q_1] \rightarrow$ a for q_1 in Q

Rule 3 – Each move that does not pops a symbol from stack with transition as

$$\delta(q, a, Z_0) = (q_1, Z_1 Z_2 Z_3 Z_4 ....)$$

Induces a production as

$$[q, Z_0, q_m] \rightarrow a\,[q_1, Z_1, q_2][q_2, Z_2, q_3][q_3, Z_3, q_4][q_4, Z_4, q_5]....[q_{m-1}, Z_m, q_m]$$

For each q_i in Q where 1<= i<=m.

After defining all the rules, apply simplification of grammar to get reduced grammar.

Example: ***Construct a CFG from the following PDA that accepts if-else of a C program***

$M = \{\{q\}, \{i, e\}\{Z, Z_0\}, \delta, q, Z_0, \Phi\}$ where δ is defined as

$$\delta(q, i, Z_0) = (q, ZZ_0)$$
$$\delta(q, e, Z) = (q, \varepsilon)$$
$$\delta(q, \varepsilon, Z_0) = (q, \varepsilon)$$

Solution: The state is q, and the stack symbols are Z and Z_0.

The states are $\{S, [q, Z, q], [q, Z_0, q]\}$

S- Production are given by Rule 1

$$S \rightarrow [q, Z_0, q]$$

1. The CFG for $\delta(q, i, Z_0) = (q, ZZ_0)$ is obtained by rule 3

   $$[q, Z_0, q] \rightarrow i\,[q, Z, q][q, Z, q]$$

2. The CFG for $\delta(q, e, Z) = (q, \varepsilon)$ is obtained by rule 2

   $$[q, Z, q] \rightarrow \varepsilon$$

3. The CFG for $\delta(q, \varepsilon, Z_0) = (q, \varepsilon)$ is obtained by rule 3

   $$[q, Z_0, q] \rightarrow \varepsilon$$

Example: ***Construct a CFG from the following PDA***

$M = \{\{q_0, q_1\}, \{a, b\}\{Z, Z_0\}, \delta, q_0, Z_0, \Phi\}$ where δ is defined as

$$\delta(q_0, b, Z_0) = (q_0, ZZ_0)$$
$$\delta(q_0, \varepsilon, Z_0) = (q_0, \varepsilon)$$
$$\delta(q_0, b, Z) = (q_0, ZZ)$$
$$\delta(q_0, a, Z) = (q_1, Z)$$
$$\delta(q_1, b, Z) = (q_1, \varepsilon)$$
$$\delta(q_1, a, Z_0) = (q_0, Z_0)$$

Solution: The states are q_0 and q_1, and the stack symbols are Z and Z_0.

The states are $\{S, [q_0, Z_0, q_0], [q_0, Z_0, q_1], [q_1, Z_0, q_0], [q_1, Z_0, q_1], [q_0, Z, q_0], [q_0, Z, q_1], [q_1, Z, q_0], [q_1, Z, q_1]$

S- Production are given by Rule 1

$$S \rightarrow [q_0, Z_0, q_0], [q_0, Z_0, q_1]$$

4. The CFG for $\delta(q_0, b, Z_0) = (q_0, ZZ_0)$ is obtained by rule 3

$$[q_0, Z_0, q_0] \rightarrow b\,[q_0, Z, q_0][q_0, Z_0, q_0]$$
$$[q_0, Z_0, q_0] \rightarrow b\,[q_0, Z, q_1][q_1, Z_0, q_0]$$
$$[q_0, Z_0, q_1] \rightarrow b\,[q_0, Z, q_0][q_0, Z_0, q_1]$$
$$[q_0, Z_0, q_1] \rightarrow b\,[q_0, Z, q_1][q_1, Z_0, q_1]$$

5. The CFG for $\delta(q_0, \varepsilon, Z_0) = (q_0, \varepsilon)$ is obtained by rule 2

$$[q_0, Z_0, q_0] \rightarrow \varepsilon$$

6. The CFG for $\delta(q_0, b, Z) = (q_0, ZZ)$ is obtained by rule 3

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

7. The CFG for $\delta(q_0, a, Z) = (q_1, Z)$ is obtained by rule 3

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

8. The CFG for $\delta(q_1, b, Z) = (q_1, \varepsilon)$ is obtained by rule 2

$$[q_1, Z, q_1] \rightarrow b$$

9. The CFG for $\delta(q_1, a, Z_0) = (q_0, Z_0)$ is obtained by rule 2

$$[q_1, Z_0, q_0] \rightarrow a\,[q_0, Z_0, q_0]$$
$$[q_1, Z_0, q_1] \rightarrow a\,[q_0, Z_0, q_1]$$

Simplifying the grammar: First identify the non-terminals that are not defined and eliminate the productions that refer to these productions. Similarily, use the procedure of eliminating the useless symbols and useless productions. Hence the complete grammar is as follows:

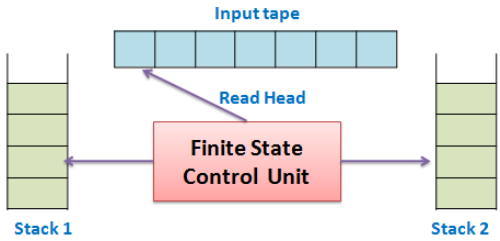
$$S \rightarrow [q_0, Z_0, q_0]$$
$$[q_0, Z_0, q_0] \rightarrow b\,[q_0, Z, q_1][q_1, Z_0, q_0]$$
$$[q_0, Z_0, q_0] \rightarrow \varepsilon$$
$$[q_0, Z, q_1] \rightarrow b\,[q_0, Z, q_1][q_1, Z, q_1]$$
$$[q_0, Z, q_1] \rightarrow a\,[q_1, Z, q_1]$$
$$[q_1, Z, q_1] \rightarrow b$$
$$[q_1, Z_0, q_0] \rightarrow a\,[q_0, Z_0, q_0]$$

Two Stack Pushdown Automata:-

Two-Stack PDA is a computational model based on the generalization of Pushdown Automata (PDA).

- Some languages that are not CFL can be accepted by a two-stack PDA.
- Actually, a PDA with two stacks has the same computation power as a Turing Machine.
- The move of the Two-Stack PDA is based on
  - The state of the finite control.

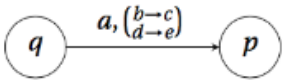
- The input symbol read.
- The top stack symbol on each of its stacks.



✶ In each transition includes pop and push actions on both stacks.

For example, the diagram contains input a

1. b is popped from the first stack and c is pushed on the first stack, and
2. c is popped from the second stack and e is pushed on the second stack.



Formal Definition:- Let us define a two-stack pushdown automaton to be a six-tuple

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

Q - is a finite set of states,

Σ - is an input alphabet

Γ - is a stack of symbols

δ - the transition relation, is a finite subset of

$$(Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma^* \times \Gamma^*) \Rightarrow (Q \times \Gamma^* \times \Gamma^*)$$

$q_0 \in Q$ is the initial state and

$F \subseteq Q$ is the set of final states

Where in the left-hand side, the third parameter indicates the first stack symbol, the fourth parameter indicates the second stack symbol. On the right-hand side, the second parameter indicates the operation on the first stack, and the third parameter indicates the operation on the second stack.

Example: ***Design a two stack-PDA for the language L = {$a^n b^n c^n$ \| n∈N}***

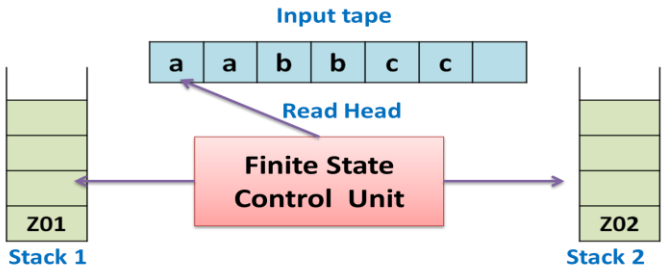
This problem is not solvable with a normal PDA, but can solved with a two-stack PDA.

The PDA M is defined by

$$M = \{\{q_0, q_1, q_2, q_3\}, \{a, b, c\}, \{a, b\}, \delta, q_0, q_3\}$$

Where δ is defined by

$$\delta(q_0, a, Z_{01}, Z_{02}) = (q_0, aZ_{01}, Z_{02})$$
$$\delta(q_1, a, a, Z_{02}) = (q_1, aa, Z_{02})$$
$$\delta(q_1, b, a, Z_{02}) = (q_2, \varepsilon, bZ_{02})$$
$$\delta(q_2, b, a, b) = (q_2, \varepsilon, bb)$$
$$\delta(q_2, c, Z_{01}, b) = (q_3, Z_{01}, \varepsilon)$$
$$\delta(q_3, c, Z_{01}, b) = (q_3, Z_{01}, \varepsilon)$$
$$\delta(q_3, \varepsilon, Z_{01}, Z_{02}) = (q_f, \varepsilon, \varepsilon)$$



Example: ***Design a two stack-PDA for the language L = {$a^n b^n a^n b^n$ | n∈N}***

This problem is not solvable with a normal PDA, but can solved with a two-stack PDA.

The PDA M is defined by

$$M = \{\{q_0, q_1, q_2, q_3, q_4\}, \{a,b,c\}, \{a,b\}, \delta, q_0, q_4\}$$

Where δ is defined by

$$\delta(q_0, a, \varepsilon, \varepsilon\} = (q_0, a, \varepsilon)$$
$$\delta(q_0, a, a, \varepsilon\} = (q_0, aa, \varepsilon)$$
$$\delta(q_0, \varepsilon, \varepsilon, \varepsilon\} = (q_1, \varepsilon, \varepsilon)$$
$$\delta(q_1, b, a, \varepsilon\} = (q_1, a, b)$$
$$\delta(q_1, b, a, b\} = (q_1, a, bb)$$
$$\delta(q_1, a, a, b\} = (q_2, \varepsilon, b)$$
$$\delta(q_2, a, a, b\} = (q_2, \varepsilon, b)$$
$$\delta(q_2, b, \varepsilon, b\} = (q_3, \varepsilon, \varepsilon)$$
$$\delta(q_3, b, \varepsilon, b\} = (q_3, \varepsilon, \varepsilon)$$
$$\delta(q_3, \varepsilon, \varepsilon, \varepsilon\} = (q_4, \varepsilon, \varepsilon)$$

Application of Pushdown:-

- ✓ PDA is used in compiler design (parser design for syntactic analysis).
- ✓ Online Transaction process system.
- ✓ Tower of Hanoi (Recursive Solution).
- ✓ Conversion of infix expression to postfix expression.
- ✓ Evaluation of the input expression.

Evaluation of the input expression:- A compiler converts an arithmetic expression into code that can be evaluated using a stack.

For example, 1 + 5 * (3 + 2) + 4 might become

PUSH(1)	MUL
PUSH(5)	ADD
PUSH(3)	PUSH(4)
PUSH(2)	ADD
ADD	

PDA as a parser:- During compilation it is required to check the input validation according to the syntax of the language. The syntax for writing arithmetic statements is given below:

S → i A E T
E→ I O E| i
O → +|-|*|/
A → =
T → ;

Using these rules, we can frame a statement in the source program as i = i + i;

S → i A E T corresponds $\delta(q, i, S) = (q, AET)$
E→ I O E corresponds $\delta(q, i, E) = (q, OE)$

E→ i corresponds $\delta(q,i,E)=(q,\in)$

O → +|-|*|/ corresponds $\delta(q,+,O)=(q,\in)$

corresponds $\delta(q,+,O)=(q,\in)$

corresponds $\delta(q,-,O)=(q,\in)$

corresponds $\delta(q,*,O)=(q,\in)$

corresponds $\delta(q,/,O)=(q,\in)$

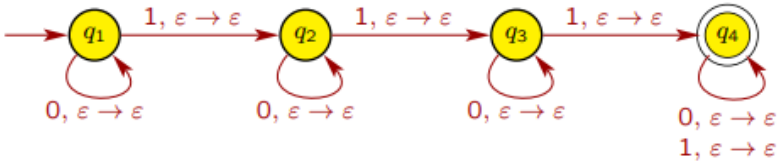
A → = corresponds $\delta(q=,A)=(q,\in)$

T → ; corresponds $\delta(q,;,T)=(q,\in)$

$$(q,i=i+i_j;,S)\Rightarrow(q,=i+i;,AET)\Rightarrow(q,i+i;,ET)\Rightarrow(q,+i;,OE)\Rightarrow(q,i;,E)\Rightarrow(q,\in,\in)$$

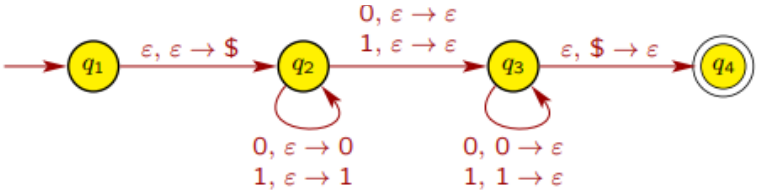
Problems

1. **Give pushdown automata that recognize the A = { w ∈ {0, 1}* | w contains at least three 1s }**



Note that A is a regular language, so the language has a DFA. We can easily convert the DFA into a PDA by using the same states and transitions and never push nor pop anything from the stack.

2. **Give pushdown automata B = { w ∈ {0, 1}∗ | w = w^R and the length of w is odd }**

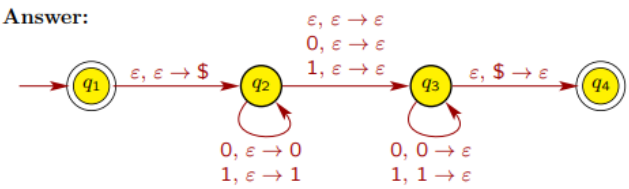


Since the length of any string w ∈ B is odd, w must have a symbol exactly in the middle position; i.e., |w| = 2n + 1 for some n ≥ 0, and the (n + 1)th symbol in w is the middle one. If a string w of length 2n + 1 satisfies w = wR, the first n symbols must match (in reverse order) the last n symbols, and the middle symbol doesn't have to match anything.

Thus, in the above PDA, the transition from q2 to itself reads the first n symbols and pushes these on the stack. The transition from q2 to q3 non-deterministically identifies the middle symbol of w, which doesn't need to match any symbol, so the stack is unaltered. The transition from q3 to itself then reads the last n symbols of w, popping the stack at each step to make sure the symbols after the middle match (in reverse order) the symbols before the middle.

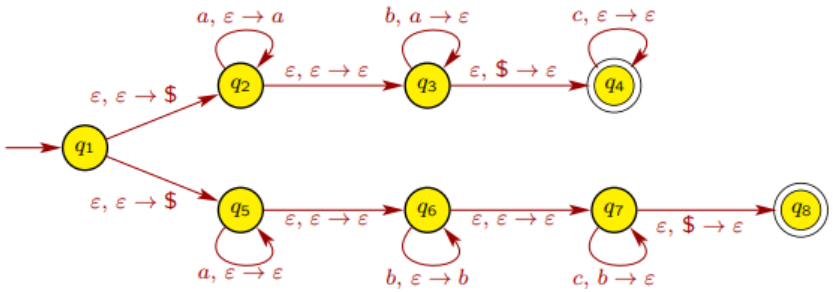
3. **Give pushdown automata C = { w ∈ {0, 1}* | w = w^R }**

Answer:



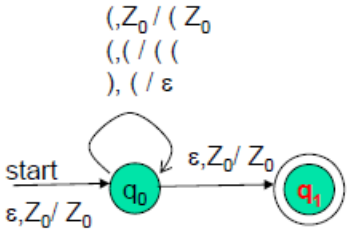
The length of a string w ∈ C can be either even or odd. If it's even, then there is no middle symbol in w, so the first half of w is pushed on the stack, we move from q2 to q3 without reading, pushing, or popping anything, and then match the second half of w to the first half in reverse order by popping the stack. If the length of w is odd, then there is a middle symbol in w, and the description of the PDA in part (b) applies.

4. **Give pushdown automata D = { $a^i b^j c^k$ | i, j, k ≥ 0, and i = j or j = k }**

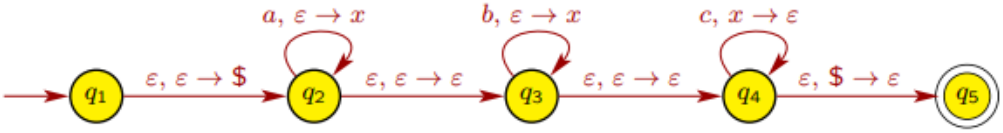


The PDA has a nondeterministic branch at q1. If the string is a i b j c k with i = j, then the PDA takes the branch from q1 to q2. If the string is $a^i b^j c^k$ with j = k, then the PDA takes the branch from q1 to q5.

5. **The language H of strings of properly balanced left and right paranthesis:** Every left paranthesis can be paired with a unique subsequent right paranthesis, and every right paranthesis can be paired with a unique preceding left paranthesis. Moreover, the string between any such pair has the same property. For example, (())() ∈ A.



6. **Give pushdown automata E = { $a^i b^j c^k$ | i, j, k ≥ 0 and i + j = k }**



For every a and b read in the first part of the string, the PDA pushes an x onto the stack. Then it must read a c for each x popped off the stack.

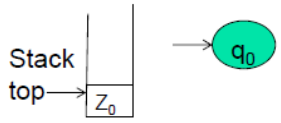
7. **L_{ww^r} = {ww^R | w is in (0+1)*}**

CFG for Lww^r : S==> 0S0 | 1S1 | ϵ

PDA for Lww^r : P := (Q,Σ, Γ , δ,q_0,Z_0,F)

= ({q_0, q_1, q_2},{0,1},{0,1,Z_0},δ,q_0,Z_0,{q_2})

Initial state of the PDA:



1. $\delta(q_0,0, Z_0)=\{(q_0,0Z_0)\}$
2. $\delta(q_0,1, Z_0)=\{(q_0,1Z_0)\}$

} First symbol push on stack

3. $\delta(q_0,0, 0)=\{(q_0,00)\}$
4. $\delta(q_0,0, 1)=\{(q_0,01)\}$
5. $\delta(q_0,1, 0)=\{(q_0,10)\}$
6. $\delta(q_0,1, 1)=\{(q_0,11)\}$

} Grow the stack by pushing new symbols on top of old (w-part)

7. $\delta(q_0, \varepsilon, 0)=\{(q_1, 0)\}$
8. $\delta(q_0, \varepsilon, 1)=\{(q_1, 1)\}$
9. $\delta(q_0, \varepsilon, Z_0)=\{(q_1, Z_0)\}$

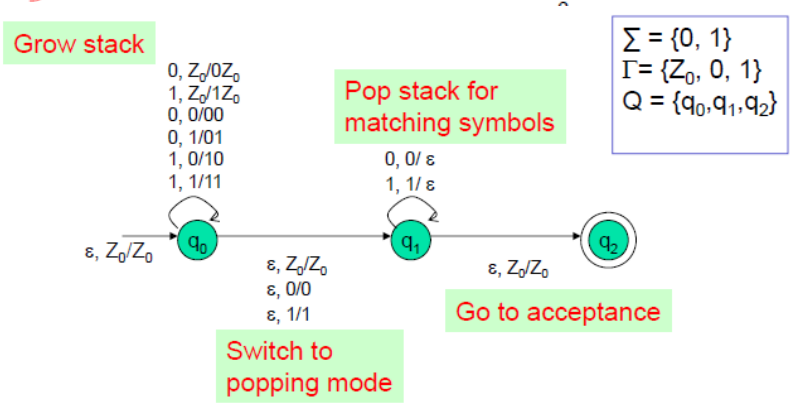
} Switch to popping mode, nondeterministically (boundary between w and w^R)

10. $\delta(q_1,0, 0)=\{(q_1, \varepsilon)\}$
11. $\delta(q_1,1, 1)=\{(q_1, \varepsilon)\}$

} Shrink the stack by popping matching symbols (w^R-part)

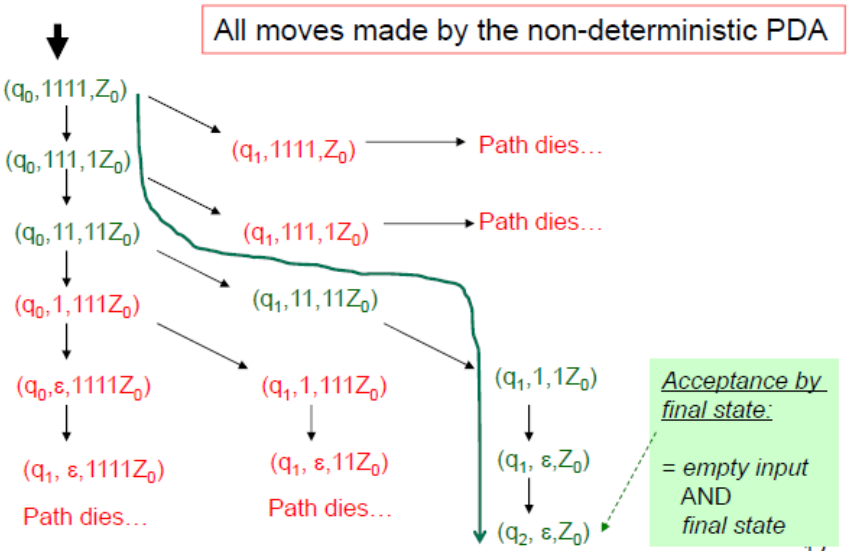
12. $\delta(q_1, \varepsilon, Z_0)=\{(q_2, Z_0)\}$

} Enter acceptance state



This would be a non-deterministic PDA

For string "1111"



8. D-PDA for L = {wcw^R | c is some special symbol not in w}