

FORMAL LANGUAGES AND AUTOMATION THEORY

Putcha Saikiran

Course Educational Objectives:

□ CEO1

To familiarize students to construct regular expressions, regular grammars & to identify non - regular languages

□ CEO2

Teach students to identify context - free languages, to convert the given grammar to various normal forms, & to make use of membership algorithm.

□ CEO3

Teach students to construct Push - Down Automata which represent context - free languages, closure properties, & to identify non - context - free languages.

□ CEO4

To familiarize students to Recursively Enumerable languages, Recursive languages, construction of Turing Machines, PCP, & undecidable problems.

Course Outcomes:

Upon successful completion of this course, students should be able to:

□ **CO1**

Explain the basic concepts of formal computation and its relationship to language.

□ **CO2**

Classify language into their types and derive its equivalent regular expression.

□ **CO3**

Apply formal reasoning to construct different language and grammar set.

□ **CO4**

Analyze complexity of various problem solving techniques using PDA and TM with recursion.

CO-PO & PSO Mapping															
Cos	PROGRAMME OUTCOMES												PSOs		
	1	2	3	4	5	6	7	8	9	10	11	12	1	2	3
C01	3	1													
C02	3	2													
C03	2	3	2												
C04	2	3													
Avg.															

Syllabus:

UNIT:1

(12 Hours)

Mathematical preliminaries: Alphabet, String, Languages, Grammars, Strings and operations on strings.

Finite Automata: Definition, Basic model, Types of Finite Automata, Design of DFA, Design of NFA, NFA vs. DFA, Eliminating ϵ -transitions from NFA, NFA to DFA conversion, NFA or DFA as a language acceptor, Minimization of Finite Automata. Equivalence of two Finite Automata

UNIT:2

(14 Hours)

Regular Expressions: Regular Set and Regular Expressions. Operators in Regular expressions, Building Finite Automata from Regular expression, Arden's theorem & Building Regular expression from Finite Automata, Pumping Lemma for Regular languages, Closure properties of Regular languages.

Grammar: Definition, Regular Grammar, Regular Grammar to Finite Automaton, Finite Automaton to Regular Grammar, Designing Context Free Grammar, String Derivation, Parse Tree Construction, Ambiguous Grammar, Chomsky and Greibach Normal Forms, CYK parsing algorithm, Closure

Properties of CFL, Pumping Lemma for CFL, Introducing Non-Context Free Grammar, Chomsky Hierarchy.

UNIT:3

(12 Hours)

Push Down Automata: Basic Model, Components, Moves of a PDA, ID of a PDA, Design of Deterministic PDA and Non-deterministic PDA, PDA to CFG and CGA to PDA conversion.

Turing Machines: Basic Model, Components, move of a TM, ID of TM, Design of a TM, Variants Of Turing Machine, Recursively Enumerable Languages, Undecidable problems, Post correspondence problem as an Undecidable Problem. Linear Bounded Automata and Context Sensitive Languages

UNIT:4

(10 Hours)

Primitive Recursive functions: μ - Recursive functions, Cantor and Godel numbering, Ackermann's function, Excursiveness of Ackermann and Turing computable functions. Church Turing hypothesis, Recursive and Recursively Enumerable sets, NP Completeness: P and NP, NP complete and NP Hard problems.



UNIT-1

Mathematical preliminaries:

□ Symbols:

Symbols are an entity or individual objects, which can be any letter, alphabet or any picture.

Example: 1, a, b, #

□ **Alphabets:**

Alphabets are a finite set of symbols. It is denoted by Σ .

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

$$\Sigma = \{A, B, C, D\}$$

$$\Sigma = \{0, 1, 2\}$$

$$\Sigma = \{0, 1, \dots, 5\}$$

$$\Sigma = \{\#, \beta, \Delta\}$$

□ String:

It is a finite collection of symbols from the alphabet. The string is denoted by w .

- If $\Sigma = \{a, b\}$, various string that can be generated from Σ are $\{ab, aa, aaa, bb, bbb, ba, aba.....\}$.
- A string with zero occurrences of symbols is known as an empty string. It is represented by ϵ .
- The number of symbols in a string w is called the length of a string. It is denoted by $|w|$.

□ Language:

A language is a collection of appropriate string. A language which is formed over Σ can be **Finite** or **Infinite**.

Example 1: $L1 = \{\text{Set of string of length 2}\} = \{aa, bb, ba, ab\}$ **Finite Language**

Example 2: $L2 = \{\text{Set of all strings starts with 'a'}\} = \{a, aa, aaa, abb, abbb, ababb\}$ **Infinite Language**

□ **Grammar:**

Standard way of representing the language is called grammar in Automata.

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

$V = \text{Variables (Non-Terminal)}$

$T = \text{Terminal}$

$P = \text{Production Rules}$

$S = \text{Start symbol}$

1. **V = Variables (Non-Terminal)**

Finite number of non-empty set Represented by capital symbols. A, B, C

Not a part of string which makes after production rule.

2. **T = Terminal**

Finite set of **Alphabets** (Σ), Represented by small letters, i.e. a,b,c.

All variables are replaced with terminals through production rules.

Terminals are part of string which makes after production rule.

3. P = Production Rules

(Finite set of non-empty rules to makes a string of Language)

i.e. $P = \{ S \rightarrow aSb, S \rightarrow bSa, S \rightarrow \epsilon \}$

4. S = Start symbol

Start symbol is used to start the production rule represented by S.

String:

- String is a part of language. Grammar produce strings through its production rules. String is generally denoted as w and length of a string is denoted as $|w|$. There may be many strings, a language with Empty string represented as epsilon (ϵ) or lamda (λ).
- Suppose a language with four **strings** is given below:

$$L(G) = \{a, bb, abc, b\}$$

Representation: $(w \mid w \in L)$

Number of Strings

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

Number of Strings (of length 2) will be

a a

a b

b a

b b

Length of String $|w| = 2$

Possible Number of Strings = 4

Conclusion: For alphabet $\{a, b\}$ with length n , number of strings can be generated = 2^n .

Operations on strings:

The different operations performed on strings are explained below –

- Concatenation.
- Substring.
- Kleen star operation.
- Reversal.

□ **Concatenation**

Concatenation is nothing but combining the two strings one after another.

□ **Substring**

If 'w' is a string then 'v' is substring of 'w' if there exists string x and y such that $w = xvy$

□ **Kleen star operation**

Let 'w' be a string. w^* is a set of strings obtained by applying any number of concatenations of w with itself, including empty string.

□ Reversal operation

If 'w' is a string, then w^R is the reversal of the string in backwards.

w

Finite Automata:

- Finite automata are used to recognize patterns.
- It takes the string of symbol as input and changes its state accordingly. When the desired symbol is found, then the transition occurs.
- At the time of transition, the automata can either move to the next state or stay in the same state.
- Finite automata have two states, **Accept state** or **Reject state**. When the input string is processed successfully, and the automata reached its final state, then it will accept.

Formal Definition of FA

A finite automaton is a collection of 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where:

- Q : finite set of states
- Σ : finite set of the input symbol
- q_0 : initial state
- F : final state
- δ : Transition function

Finite Automata Model:

Finite automata can be represented by input tape and finite control.

- **Input tape:** It is a linear tape having some number of cells. Each input symbol is placed in each cell.
- **Finite control:** The finite control decides the next state on receiving particular input from input tape.

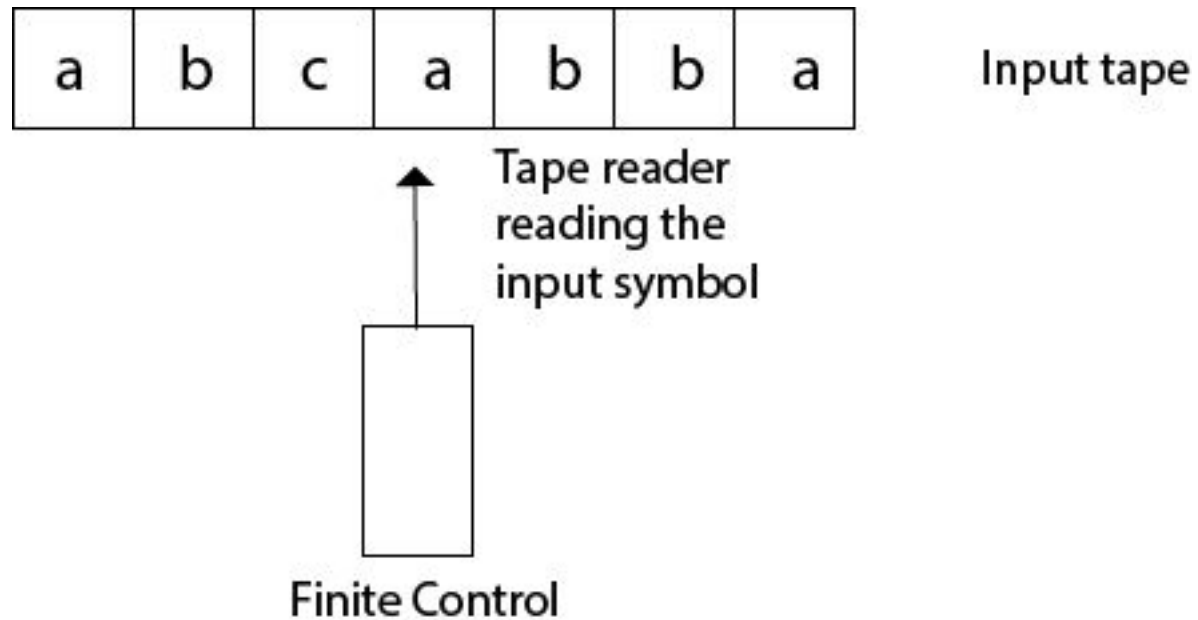
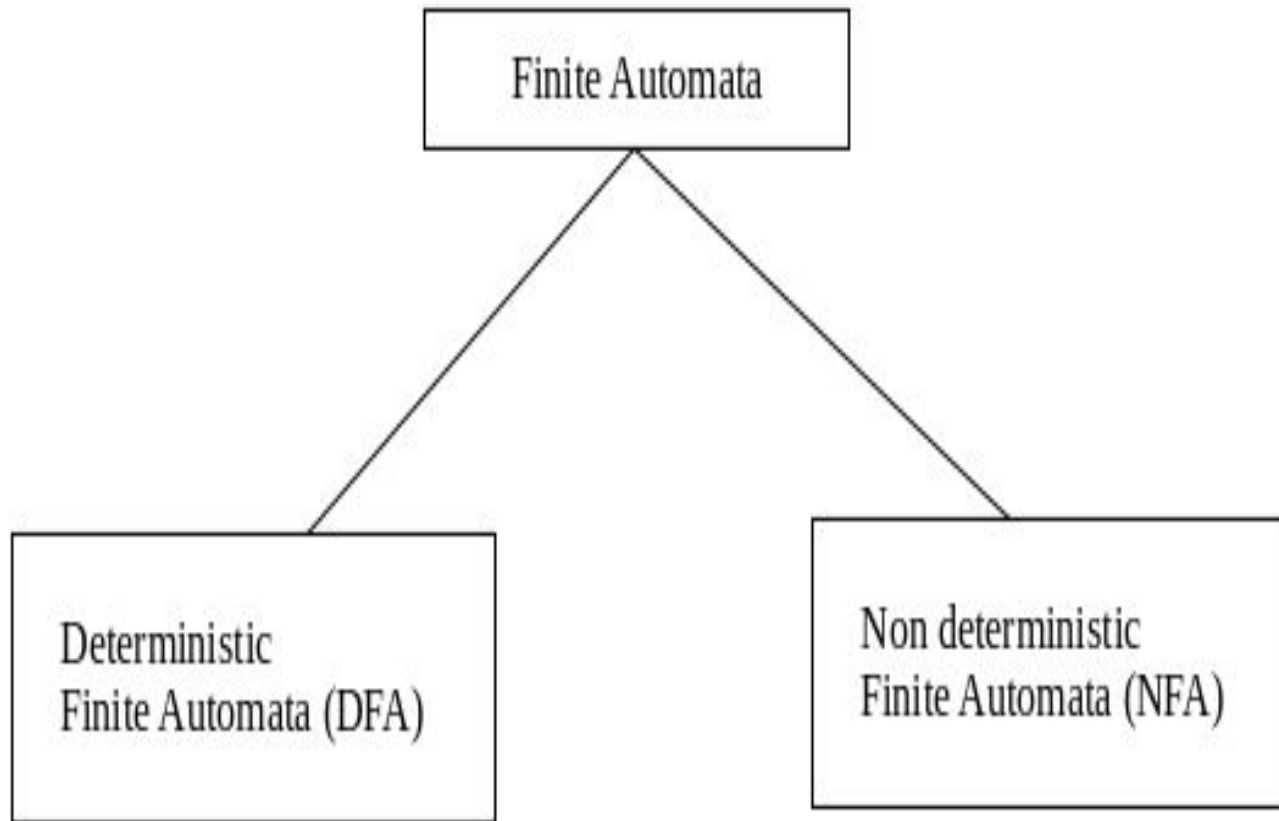


Fig :- Finite automata model

Types of Automata:



1. DFA

- DFA refers to deterministic finite automata. Deterministic refers to the uniqueness of the computation. In the DFA, the machine goes to one state only for a particular input character. DFA does not accept the null move.

2. NFA

- NFA stands for non-deterministic finite automata. It is used to transmit any number of states for a particular input. It can accept the null move.

Transition Diagram:

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

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

Transition Table:

- The transition table is basically a tabular representation of the transition function. It takes two arguments (a state and a symbol) and returns a state (the "next state").

A transition table is represented by the following things:

- Columns correspond to input symbols.
- Rows correspond to states.
- Entries correspond to the next state.
- The start state is denoted by an arrow with no source.
- The accept state is denoted by a star.

DFA (Deterministic finite automata)

A DFA is a collection of 5-tuples same as FA.

- ❖ Q : finite set of states
- ❖ Σ : finite set of the input symbol
- ❖ q_0 : initial state
- ❖ F : final state
- ❖ δ : Transition function

Transition function can be defined as:

- ❖ $\delta: Q \times \Sigma \rightarrow Q$

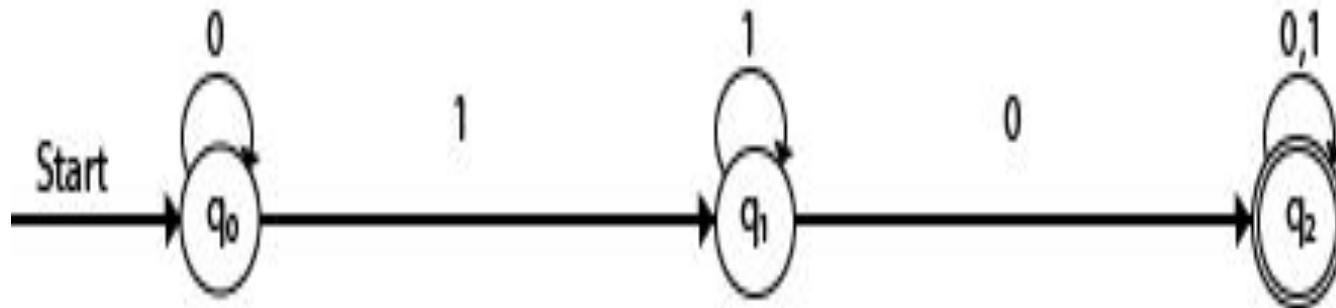
$Q = \{q_0, q_1, q_2\}$

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

$q_0 = \{q_0\}$

$F = \{q_2\}$

Present State	Next state for Input 0	Next State of Input 1
$\rightarrow q_0$	q_0	q_1
q_1	q_2	q_1
$*q_2$	q_2	q_2



Construction Of DFA:

- Following steps are followed to construct a DFA for Type-01 problems-

Step-01:

- Determine the minimum number of states required in the DFA.
- Draw those states.

RULE

- Calculate the length of substring.
- All strings ending with 'n' length substring will always require minimum $(n+1)$ states in the DFA.

Step-02:

- Decide the strings for which DFA will be constructed.

Step-03:

- Construct a DFA for the strings decided in Step-02.

RULE

- While constructing a DFA,
- Always prefer to use the existing path.
- Create a new path only when there exists no path to go with.

Step-04:

- Send all the left possible combinations to the starting state.
- Do not send the left possible combinations over the dead state.

NFA (Non-Deterministic finite automata)

Formal definition of NFA:

NFA also has five states same as DFA, but with different transition function, as shown follows:

- $\delta: Q \times \Sigma \rightarrow 2^Q$

where,

- Q : finite set of states
- Σ : finite set of the input symbol
- q_0 : initial state
- F : **final** state
- δ : Transition function

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

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

$q_0 = \{q_0\}$

$F = \{q_2\}$

Present State	Next state for Input 0	Next State of Input 1
$\rightarrow q_0$	q_0, q_1	q_1
q_1	q_2	q_0
$*q_2$	q_2	q_1, q_2

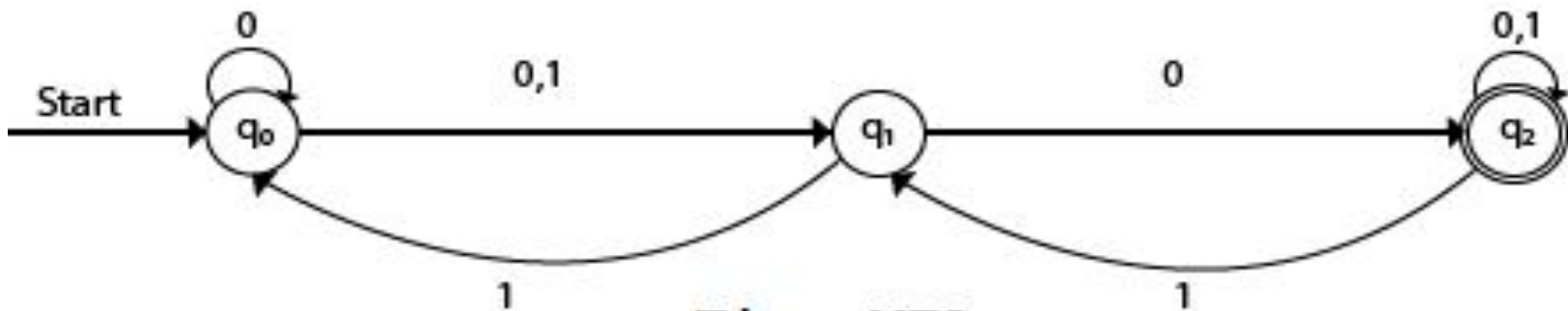


Fig: NFA



DFA (vs) NFA

DFA

- DFA stands for Deterministic Finite Automata.
- For each sigma value (0, 1...n) there must be a provided path from each state.

NFA

- NFA stands for Nondeterministic Finite Automata.
- For each sigma value (0, 1...n) there is no need to specify path from each state.

DFA

- As DFA provide a path for each input, that's why all digital computers are deterministic.
- Multiple choices are not available corresponding to an output.

NFA

- As NFA does not provide a path for each input, that's why NFA not use in digital computers.
- Multiple choices are available corresponding to an output.

DFA

- DFA cannot use Empty String (Epsilon) transition.
- DFA is more difficult to construct and understand.

NFA

- NFA can use Empty String (Epsilon) transition.
- NFA is easier to construct and understand.

DFA

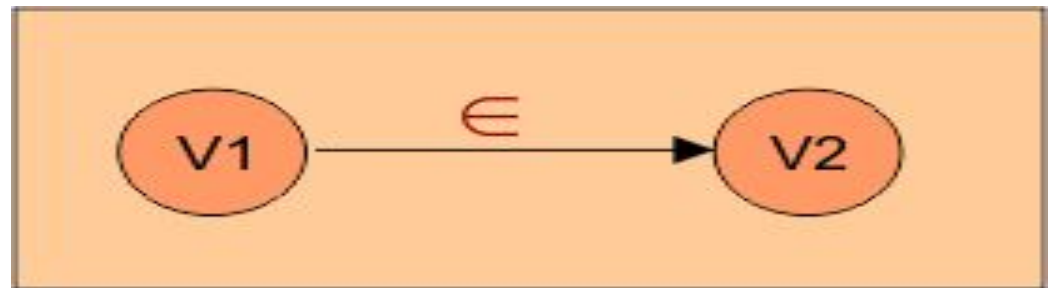
- All DFA are NFA.
- DFA requires more space. Because it has normally more states.

NFA

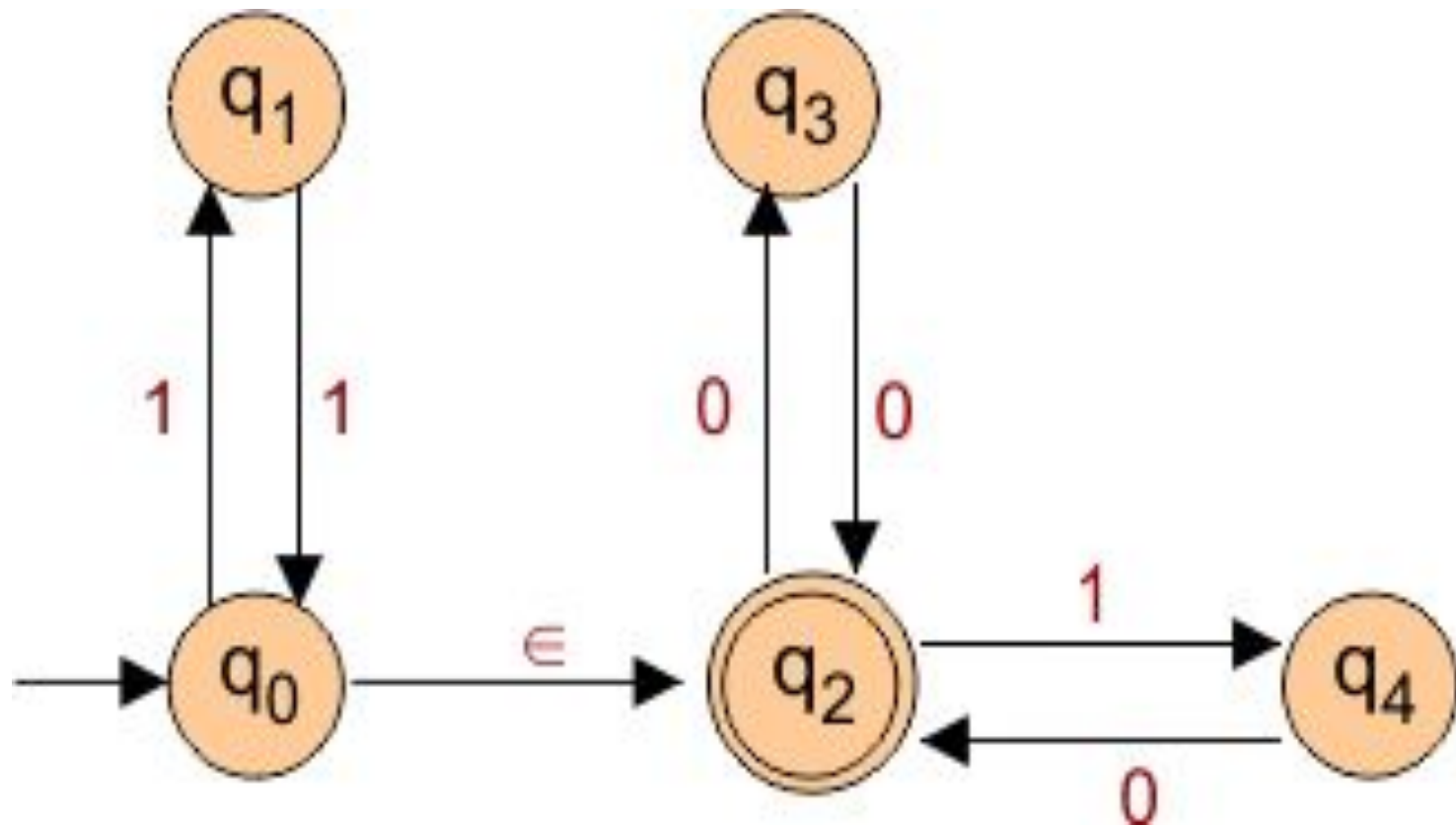
- Not all NFA are DFA.
- NFA requires less space than DFA. Because it has normally less states

Steps For Elimination Of Epsilon From NFA

- ❑ **Step 01:** Find all edges starting from V2
- ❑ **Step 02:** Remove the epsilon. And All Finding edges from V2 are Duplicated to V1 without changing edge labels.
- ❑ **Step 03:** if V1 is initial state, make V2 initial
- ❑ **Step 04:** if V2 is final state, make V1 final
- ❑ **Step 05:** Remove all dead states



Example 1:

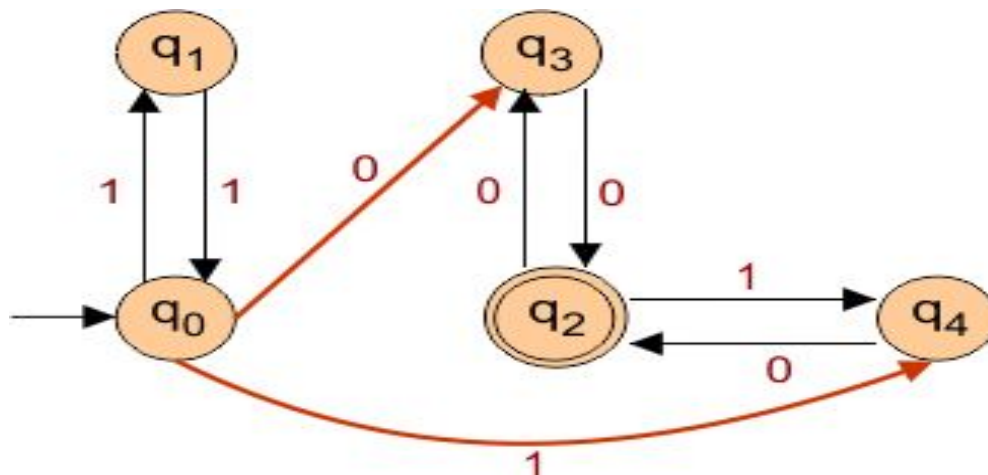


Step 1:

- The first input “0” is starting from V2 (q2) and going to q3.
- The second input “1” is starting from V2 (q2) and going to q3

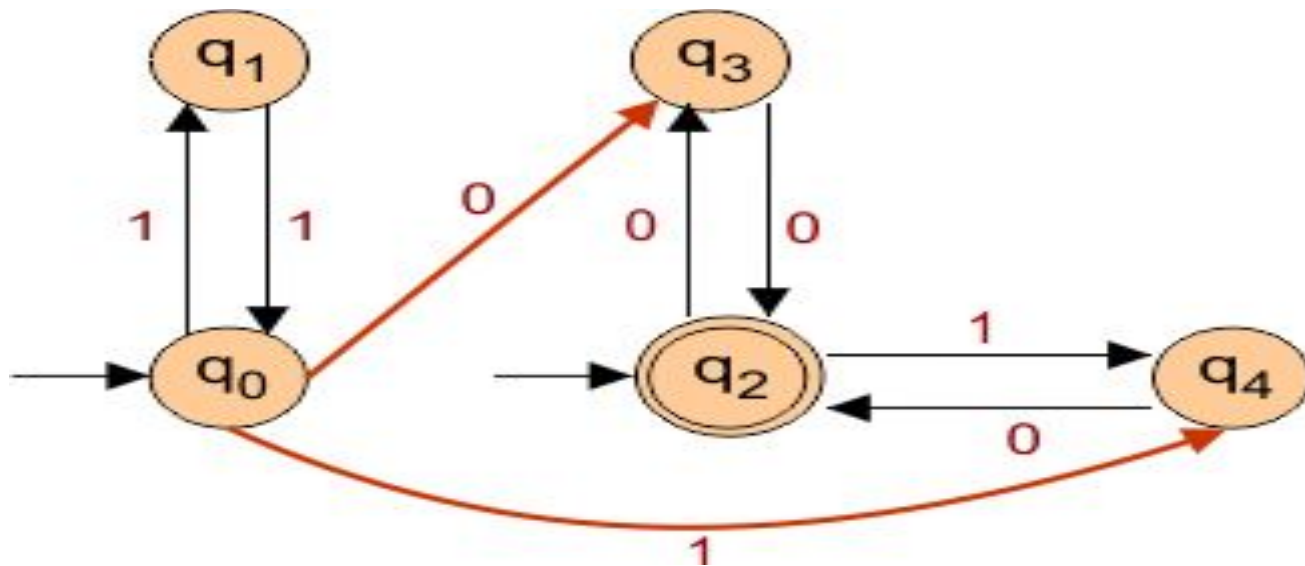
Step 2:

- First Input “0” is going to q_3 from V_2 (q_2). So first duplicate will go from V_1 (q_0) to q_3 .
- Second Input “1” is going to q_4 from V_2 (q_2). So second duplicate will go from V_1 (q_0) to q_4 .



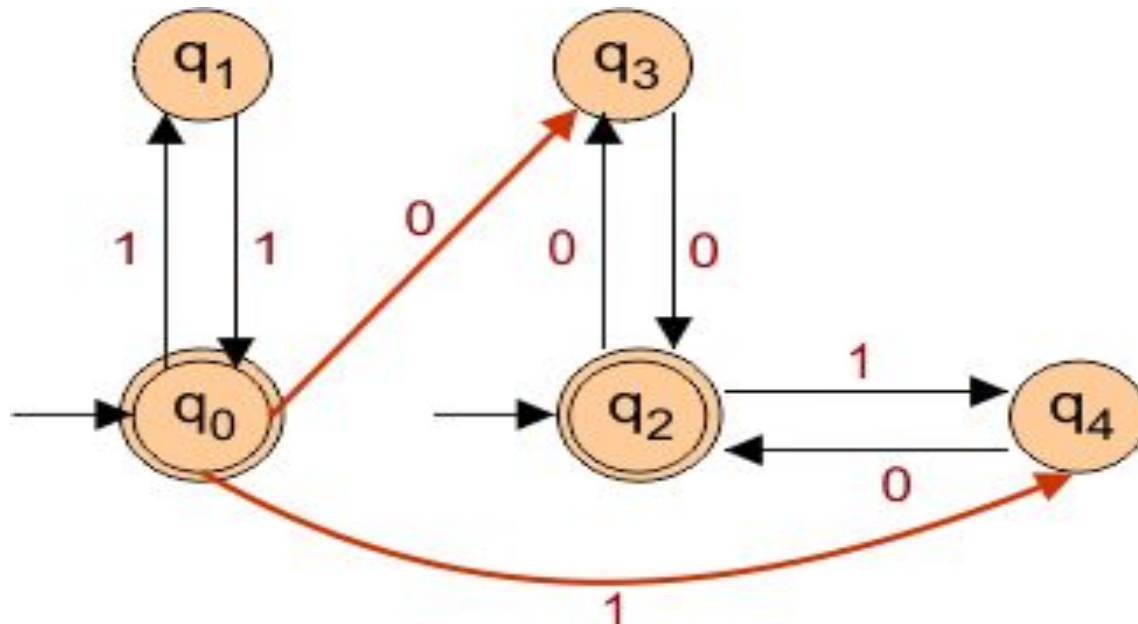
Step 3:

- As $V1(q_0)$ is a initial. So $V2(q_2)$ change to initial.



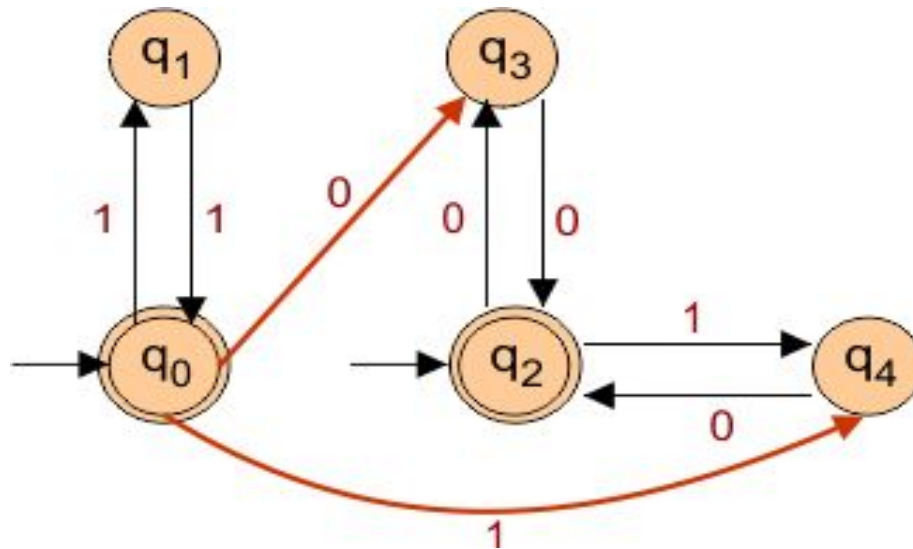
Step 4:

- As V_2 (q_2) is a final state. So $V_1(q_0)$ change to final state.

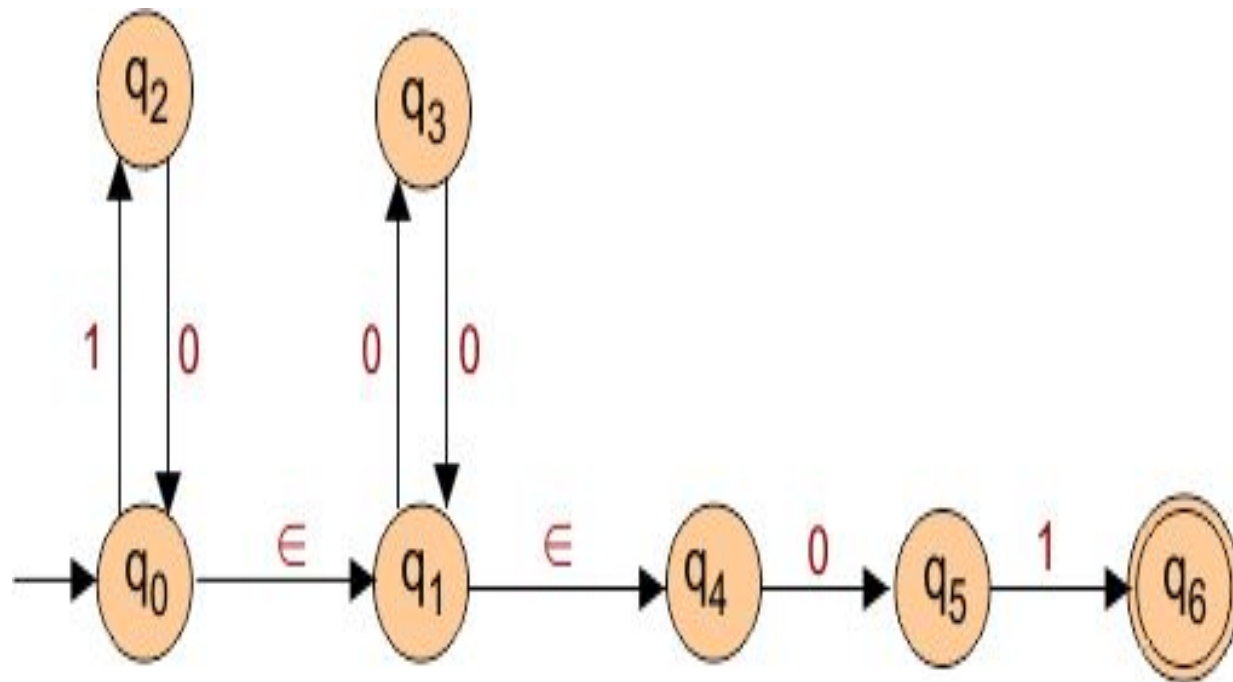


Step 5:

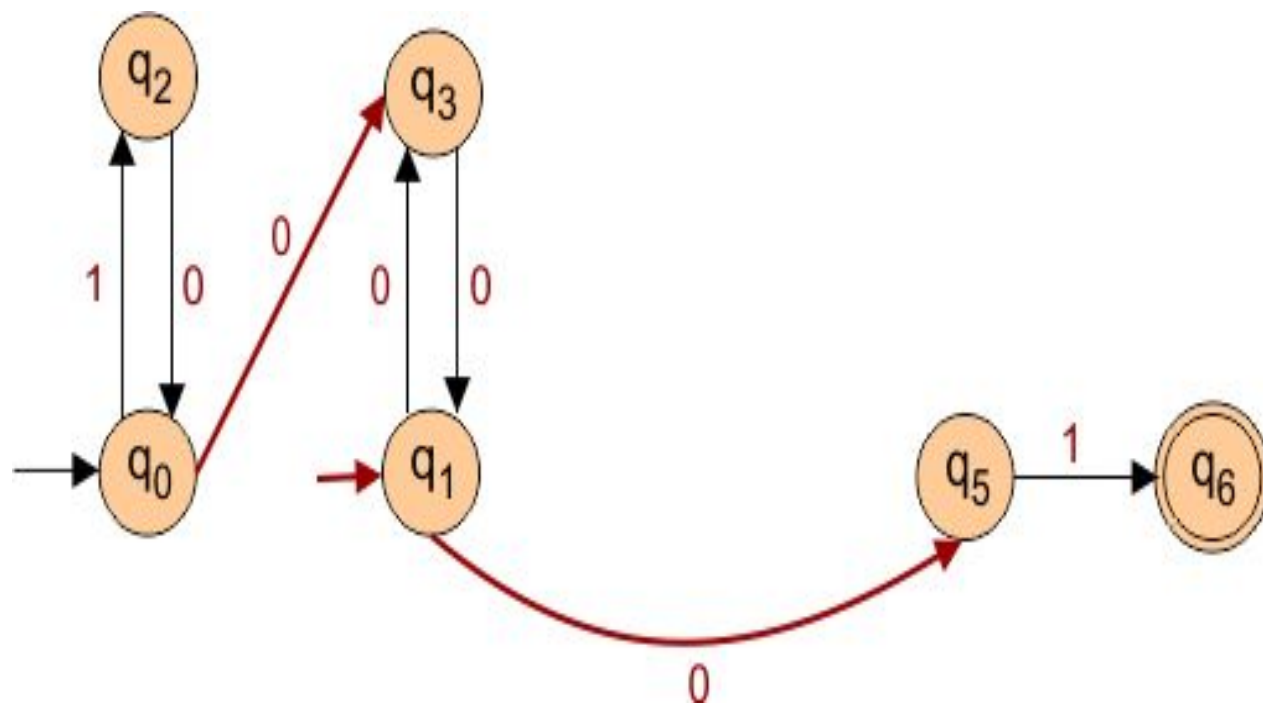
- A state which is unreachable is called dead state. In above NFA, there is no dead state.



Example 2:



□ Solution



NFA to DFA conversion:

Step-01:

- Let Q' be a new set of states of the DFA. Q' is null in the starting.
- Let T' be a new transition table of the DFA.

Step-02:

- Add start state of the NFA to Q' .
- Add transitions of the start state to the transition table T' .
- If start state makes transition to multiple states for some input alphabet, then treat those multiple states as a single state in the DFA.

Step-03:

- If any new state is present in the transition table T' ,
- Add the new state in Q' .
- Add transitions of that state in the transition table T' .

Step-04:

- Keep repeating Step-03 until no new state is present in the transition table T' .
- Finally, the transition table T' so obtained is the complete transition table of the required DFA.

Minimization of DFA

Step-01:

- Eliminate all the dead states and inaccessible states from the given DFA (if any).

Step-02:

- Draw a state transition table for the given DFA.
- Transition table shows the transition of all states on all input symbols in Σ .

Step-03:

Now, start applying equivalence theorem.

- Take a counter variable k and initialize it with value 0.
- Divide Q (set of states) into two sets such that one set contains all the non-final states and other set contains all the final states.
- This partition is called P_0 .

Step-04:

- Increment k by 1.
- Find P_k by partitioning the different sets of P_{k-1} .
- In each set of P_{k-1} , consider all the possible pair of states within each set and if the two states are distinguishable, partition the set into different sets in P_k .

Step-05:

- Repeat step-04 until no change in partition occurs.
- In other words, when you find $P_k = P_{k-1}$, stop.

Step-06:

- All those states which belong to the same set are equivalent.
- The equivalent states are merged to form a single state in the minimal DFA.

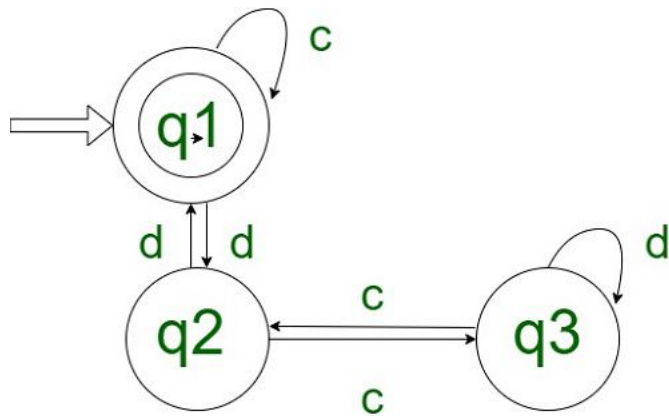
Equivalence Of F.S.A (Finite State Automata)

- An Automaton is a machine that has a finite number of states. Any Two Automaton is said to be equivalent if both accept exactly the same set of input strings.

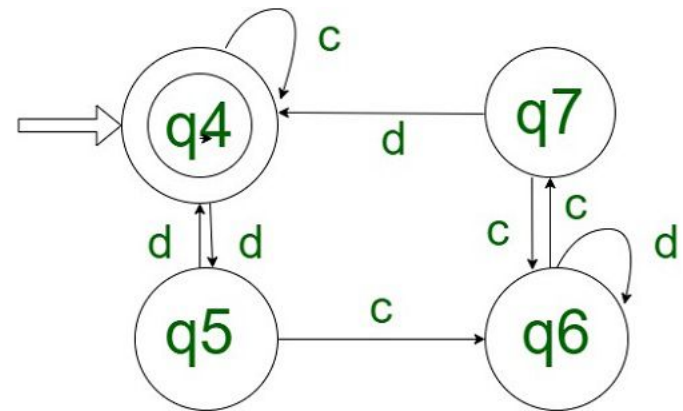
steps

- 1. The initial and final states of both the automata must be same.*
- 2. Every pair of states chosen is from a different automaton only.*
- 3. While combining the states with the input alphabets, the pair results must be either both final states or intermediate states. (i.e. both should lie either in the final state or in the non-final state).*
- 4. If the resultant pair has different types of states, then it will be non-equivalent. (i.e. One lies in the final state and the other lies in the intermediate state).*

Example 1:




AUTOMATON-1



AUTOMATON-2

FIGURE -1


- 
- **Step 1** – Since the initial and final states of both the automaton are the same, so it verifies.
Step 2 – Check for each state by making a table of states with respect to input alphabets.

States	c	d
(q1,q4)	(q1,q4)=>(F.S,F.S)	(q2,q5)=>(I.S,I.S)
(q2,q5)	(q3,q6)=>(I.S,I.S)	(q1,q4)=>(F.S,F.S)
(q3,q6)	(q2,q7)=>(I.S,I.S)	(q3,q6)=>(I.S,I.S)
(q2,q7)	(q3,q6)=>(I.S,I.S)	(q1,q4)=>(F.S,F.S)

Here ,

F.S represents -> Final State.

I.S represents -> Intermediate State
(Non-Final State) .



Step 3 – For every pair of states, the resultant states lie either in F.S or in I.S as a combination.

Conclusion – Both the Automaton are equivalent.

- **Note** : If the resultant pair of states has a different combination of states(i.e. either (F.S, I.S) or (I.S, F.S), then both the automata are said to be non-equivalent.



UNIT-2

Regular expressions:

- ❑ Regular expression is **an important notation for specifying patterns.**
- ❑ The language accepted by finite automata can be easily described by simple expressions called Regular Expressions.
- ❑ The languages accepted by some regular expression are referred to as Regular languages.
- ❑ Each pattern matches a set of strings, so regular expressions serve as names for a set of strings.

Regular set:

- Any set that represents the value of the Regular Expression is called a **Regular Set**.

Operators in RE:

1. **Union:**

- The union of two regular languages, $L1$ and $L2$, which are represented using $L1 \cup L2$, is also regular and which represents the set of strings that are either in $L1$ or $L2$ or both.

Example:

$L1 = \{00, 10, 11, 01\}$

$L2 = \{\epsilon, 100\}$

then $L1 \cup L2 = \{\epsilon, 00, 10, 11, 01, 100\}$.

2. Concatenation:

- The concatenation of two regular languages, $L1$ and $L2$, which are represented using $L1.L2$ is also regular and which represents the set of strings that are formed by taking any string in $L1$ concatenating it with any string in $L2$.

Example:

$L1 = \{ 0, 1 \}$ and $L2 = \{ 00, 11 \}$ then $L1.L2 = \{000, 011, 100, 111\}$.

3. Kleene closure:

- If $L1$ is a regular language, then the Kleene closure i.e. $L1^*$ of $L1$ is also regular and represents the set of those strings which are formed by taking a number of strings from $L1$ and the same string can be repeated any number of times and concatenating those strings.

Example:

$L1 = \{0,1\} = \{\epsilon, 0, 1, 00, 01, 10, 11, \dots\}$, then L^* is all strings possible with symbols 0 and 1 including a null string.

Algebraic properties of regular expressions:

1. Closure:

- If r_1 and r_2 are regular expressions(RE), then
- r_1^* is a RE
- r_2^* is also a RE
- $r_1 + r_2$ is a RE
- $r_1.r_2$ is a RE

2. Closure laws –

- $(r^*)^* = r^*$, closing an expression that is already closed does not change the language.
- $\emptyset^* = \epsilon$, a string formed by concatenating any number of copies of an empty string is empty itself.
- $r^+ = r.r^* = r^*r$, as $r^* = \epsilon + r + rr + rrr + \dots$ and $r.r^* = r + rr + rrr + \dots$
- $r^* = r^* + \epsilon$

3. Associativity –

If r_1, r_2, r_3 are RE, then

i.) $r_1 + (r_2 + r_3) = (r_1 + r_2) + r_3$

ii.) $r_1.(r_2.r_3) = (r_1.r_2).r_3$

4. Identity –

In the case of union operators

if $r + x = r \Rightarrow x = \emptyset$ as $r \cup \emptyset = r$, therefore \emptyset is the identity for $+$.

Therefore, \emptyset is the identity element for a union operator.

In the case of concatenation operator –

if $r.x = r$, for $x = \epsilon$

$r.\epsilon = r \Rightarrow \epsilon$ is the identity element for concatenation operator($.$) .

5. Annihilator –

- If $r + x = r \Rightarrow r \cup x = x$, there is no annihilator for $+$
- In the case of a concatenation operator, $r.x = x$, when $x = \emptyset$, then $r.\emptyset = \emptyset$, therefore \emptyset is the annihilator for the $(.)$ operator. For example $\{a, aa, ab\}.\{\} = \{\}$

6. Commutative property –

If $r1, r2$ are RE, then

- $r1 + r2 = r2 + r1$. For example, for $r1 = a$ and $r2 = b$, then RE $a + b$ and $b + a$ are equal.
- $r1.r2 \neq r2.r1$. For example, for $r1 = a$ and $r2 = b$, then RE $a.b$ is not equal to $b.a$.

7. Distributed property –

If r_1, r_2, r_3 are regular expressions, then

- $(r_1 + r_2).r_3 = r_1.r_3 + r_2.r_3$ i.e. Right distribution
- $r_1.(r_2 + r_3) = r_1.r_2 + r_1.r_3$ i.e. left distribution
- $(r_1.r_2) + r_3 \neq (r_1 + r_3)(r_2 + r_3)$

8. Idempotent law –

- $r1 + r1 = r1 \Rightarrow r1 \cup r1 = r1$, therefore the union operator satisfies idempotent property.
- $r.r \neq r \Rightarrow$ concatenation operator does not satisfy idempotent property.

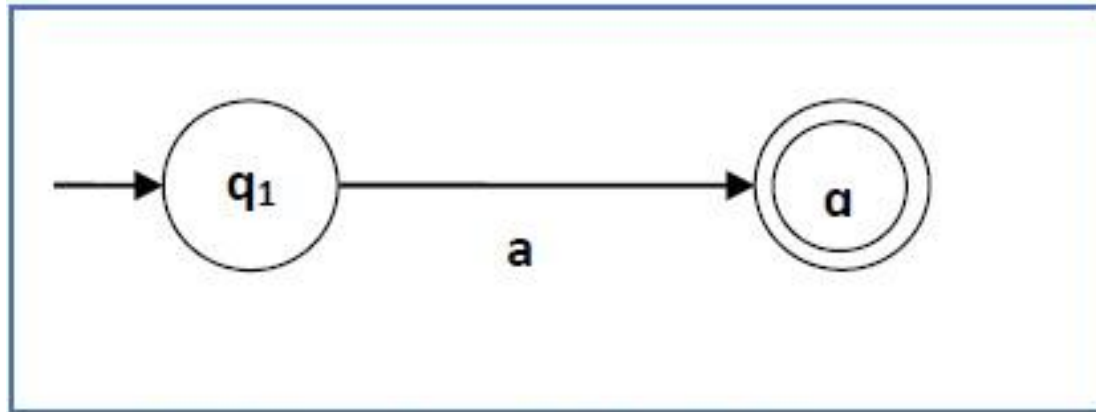
Construction of an FA from an RE

- Two methods:
 - Thompson's construction
 - Subset method

Thompson's Construction

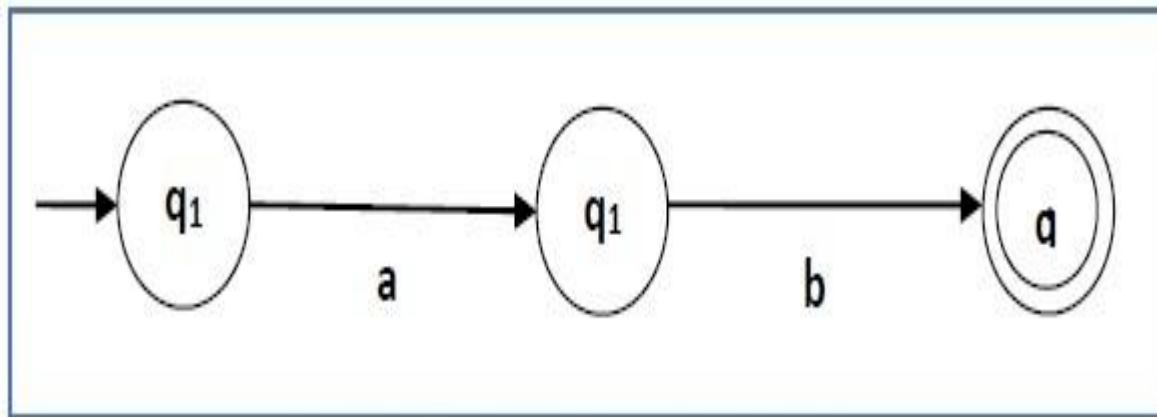
- We will reduce the regular expression into smallest regular expressions and converting these to NFA and finally to DFA.

- ▣ **Case 1** – For a regular expression 'a', we can construct the following FA –



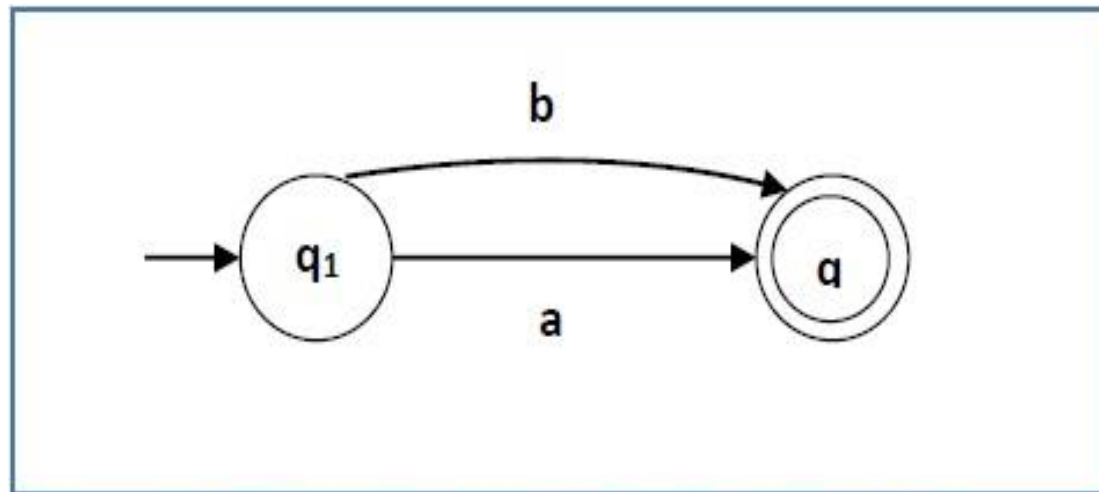
Finite automata for RE = a

- **Case 2** - For a regular expression 'ab', we can construct the following FA –



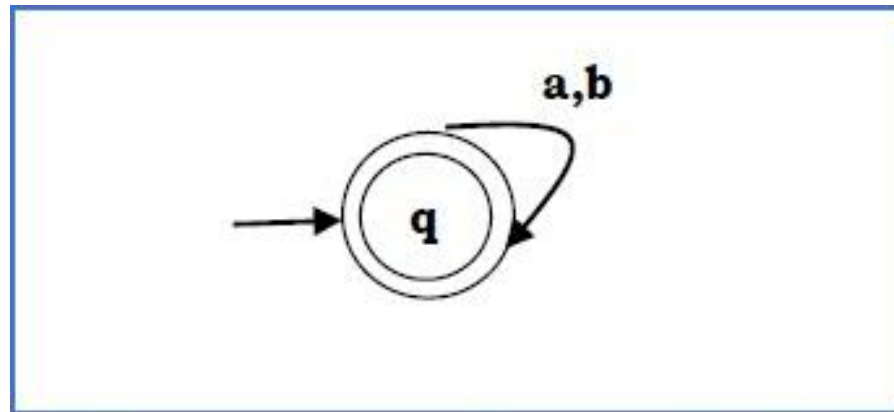
Finite automata for RE = ab

- **Case 3** - For a regular expression ' $a+b$ ', we can construct the following FA –



Finite automata for RE= $(a+b)$

- ▣ **Case 4** – For a regular expression $(a+b)^*$, we can construct the following FA –



Finite automata for RE= $(a+b)^*$

Method:

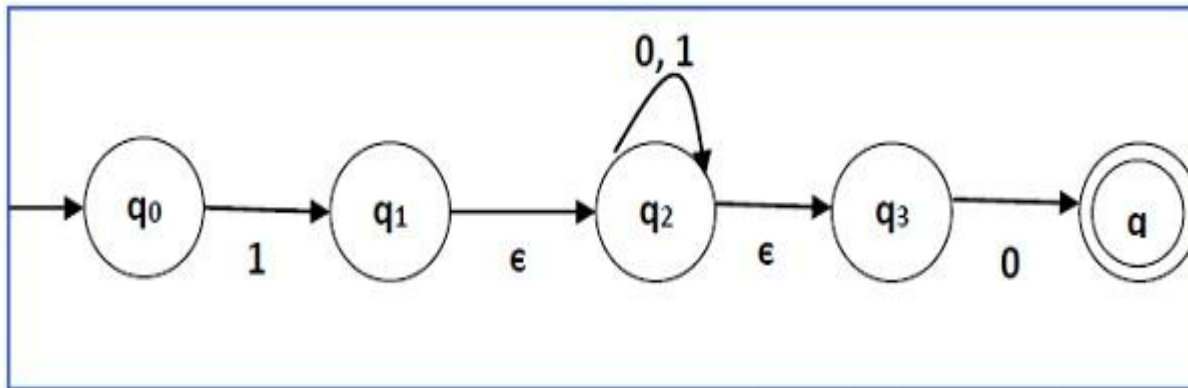
- **Step 1** Construct an NFA with Null moves from the given regular expression.
- **Step 2** Remove Null transition from the NFA and convert it into its equivalent NFA (or) DFA.

Problem

Convert the following RE into its equivalent DFA –
 $1 (0 + 1)^* 0$

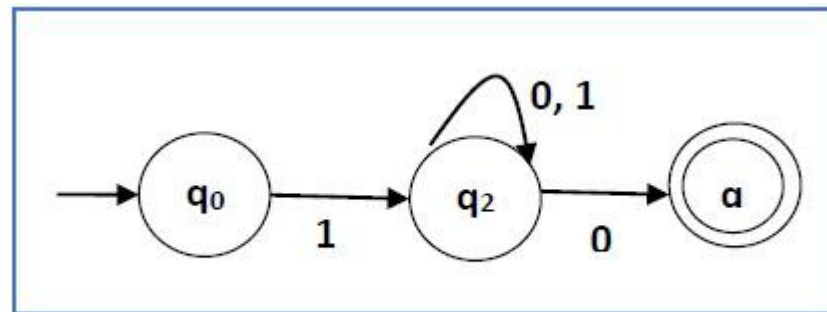
Solution

We will concatenate three expressions "1", " $(0 + 1)^*$ " and "0"



NFA with NULL transition for RA: $1 (0 + 1)^* 0$

- Now we will remove the ϵ transitions. After we remove the ϵ transitions from the NDFA, we get the following –



NDFA without NULL transition for RA: $1 (0 + 1)^* 0$

- It is an NDFA corresponding to the RE – $1 (0 + 1)^* 0$. If you want to convert it into a DFA,

Subset method:

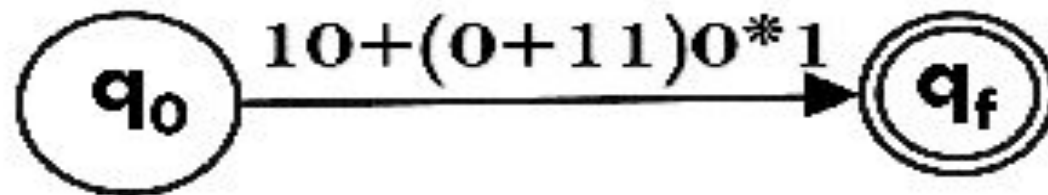
Step 1: Design a transition diagram for given regular expression, using NFA with ϵ moves.

Step 2: Convert this NFA with ϵ to NFA without ϵ .

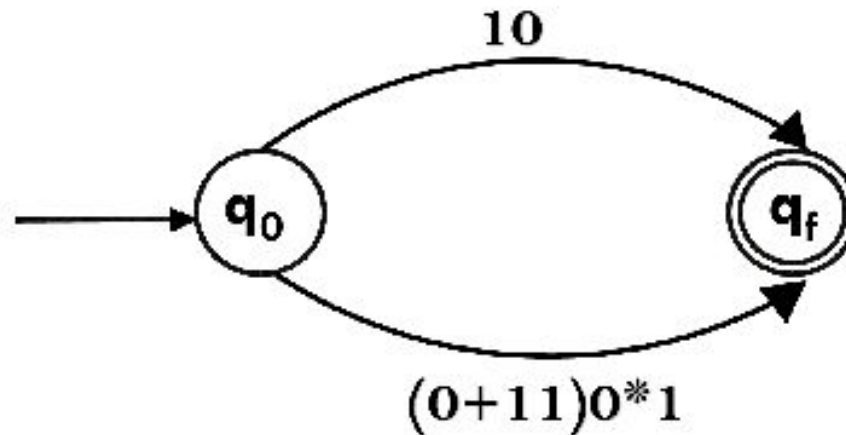
Step 3: Convert the obtained NFA to equivalent DFA.

- **Example 1:** Design a FA from given regular expression $10 + (0 + 11)0^*1$.

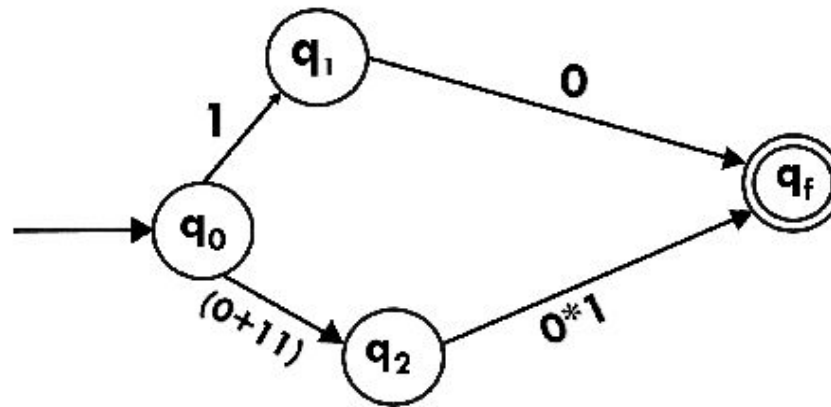
- Step 1:



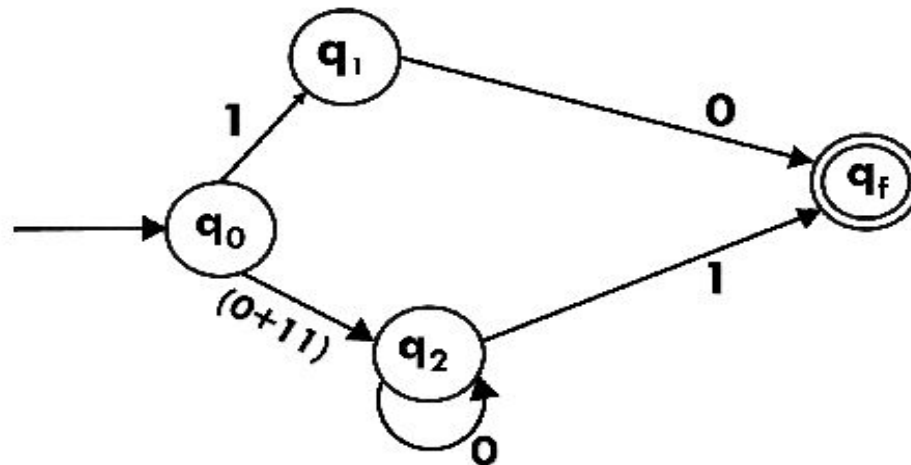
- Step 2



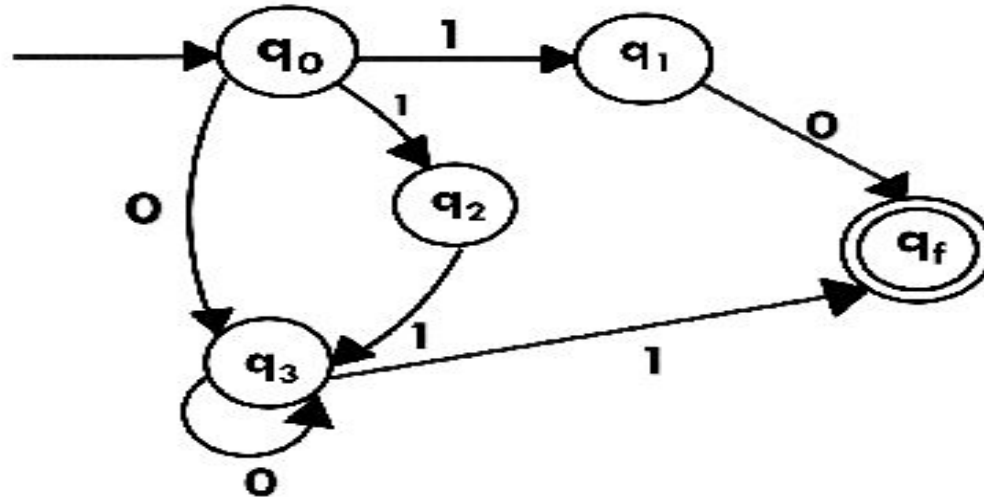
□ Step 3:



□ Step 4:



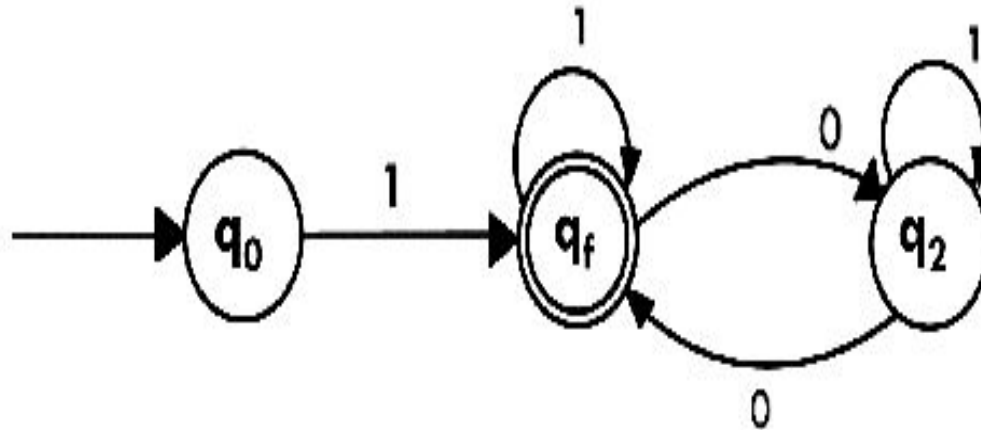
□ Step 5:



□ Step 6: Convert the above NFA to equivalent DFA (if required)

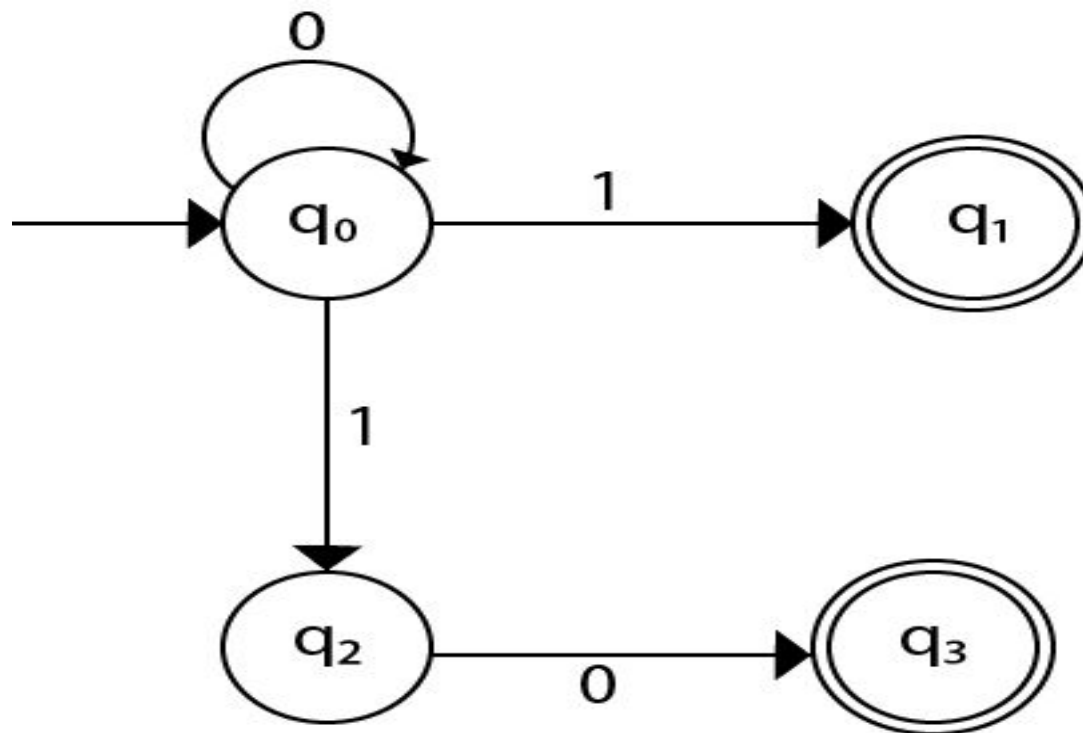
Example 2:

- Design a NFA from given regular expression $1(1^*01^*01^*)^*$.



Example 3:

- Construct the FA for regular expression $0^*1 + 10$.



Arden's theorem:

Statement –

- Let **P** and **Q** be two regular expressions.
- If **P** does not contain null string, then **R = Q + RP** has a unique solution that is **R = QP***

Proof –

$$\begin{aligned} R &= Q + (Q + RP)P \text{ [After putting the value } R = Q + RP\text{]} \\ &= Q + QP + RPP \end{aligned}$$

When we put the value of **R** recursively again and again, we get the following equation –

$$R = Q + QP + QP^2 + QP^3 \dots$$

$$R = Q (\varepsilon + P + P^2 + P^3 + \dots)$$

$$R = QP^* \text{ [As } P^* \text{ represents } (\varepsilon + P + P^2 + P^3 + \dots) \text{]}$$

Hence, proved.

Method

Step 1 – Create equations as the following form for all the states of the DFA having n states with initial state q_1 .

$$q_1 = q_1 R_{11} + q_2 R_{21} + \dots + q_n R_{n1} + \varepsilon$$

$$q_2 = q_1 R_{12} + q_2 R_{22} + \dots + q_n R_{n2}$$

.....

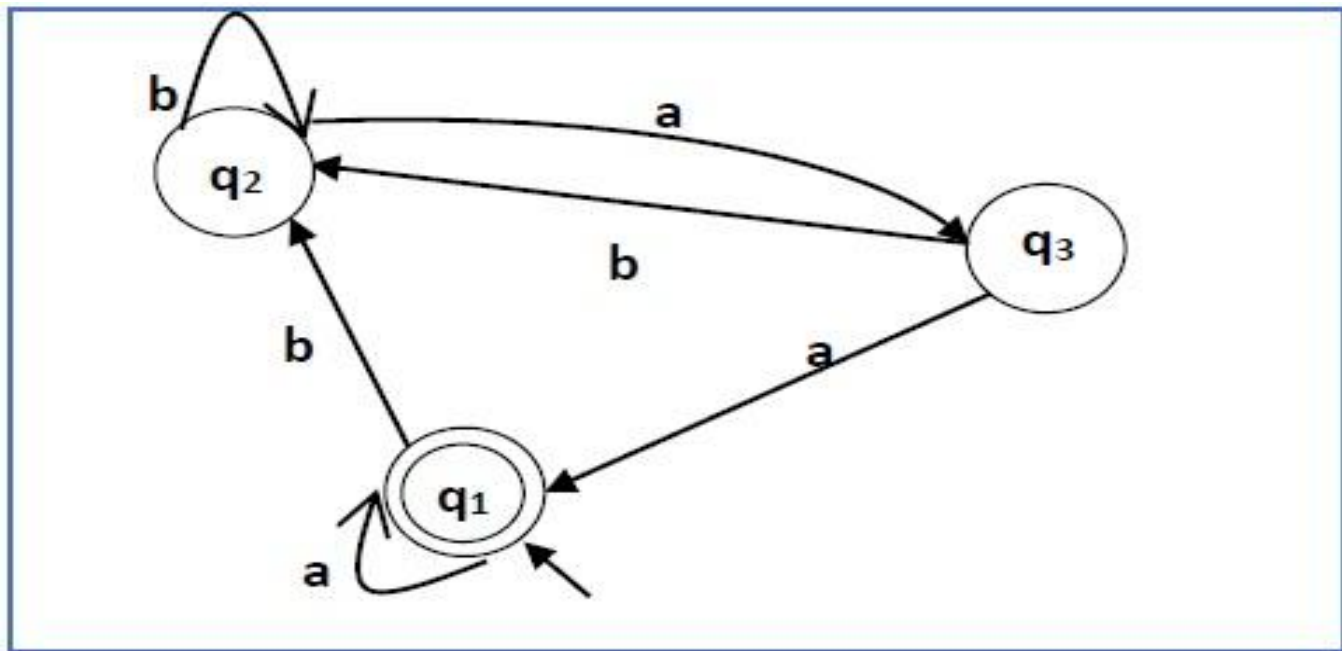
.....

$$q_n = q_1 R_{1n} + q_2 R_{2n} + \dots + q_n R_{nn}$$

R_{ij} represents the set of labels of edges from q_i to q_j , if no such edge exists, then $R_{ij} = \emptyset$

Step 2 – Solve these equations to get the equation for the final state in terms of R_{ij}

Construct a regular expression corresponding to the automata given below –



Solution –

Here the initial state and final state is q_1 .

The equations for the three states q_1 , q_2 , and q_3 are as follows

$$q_1 = q_1a + q_3a + \varepsilon \text{ (\varepsilon move is because } q_1 \text{ is the initial state)}$$

$$q_2 = q_1b + q_2b + q_3b$$

$$q_3 = q_2a$$

Now, we will solve these three equations –

$$q_2 = q_1b + q_2b + q_3b$$

$$= q_1b + q_2b + (q_2a)b \text{ (Substituting value of } q_3)$$

$$= q_1b + q_2(b + ab)$$

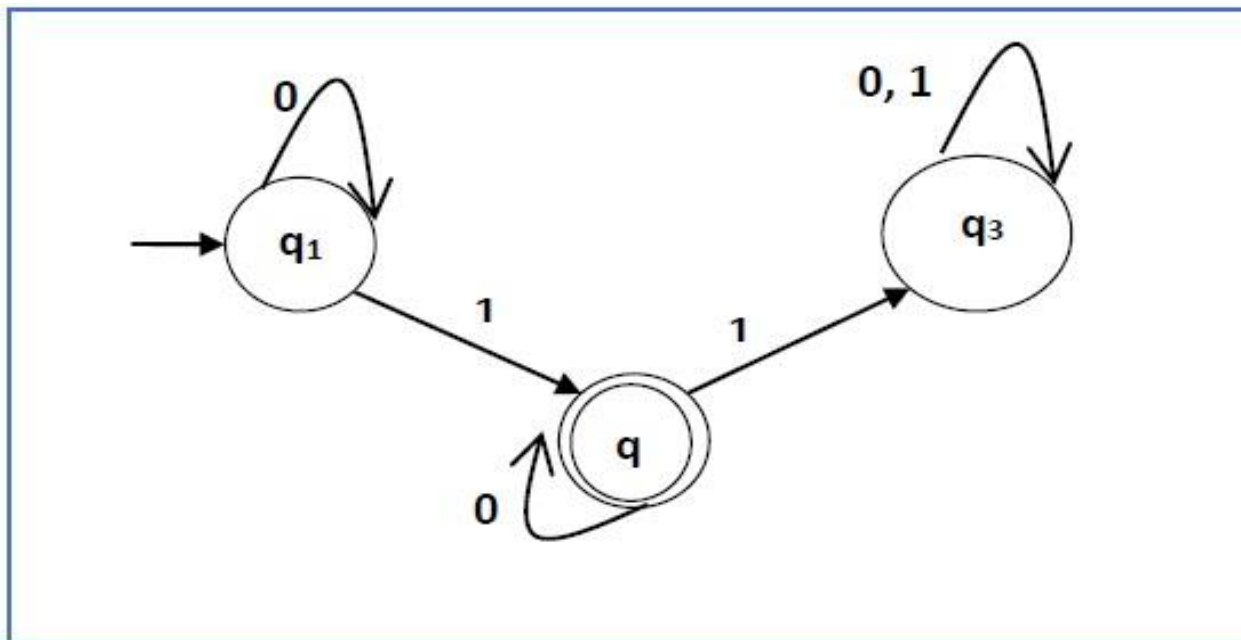
$$= q_1b (b + ab)^* \text{ (Applying Arden's Theorem)}$$

$$\begin{aligned}
q_1 &= q_1 a + q_3 a + \varepsilon \\
&= q_1 a + q_2 a a + \varepsilon \text{ (Substituting value of } q_3) \\
&= q_1 a + q_1 b(b + ab^*)aa + \varepsilon \text{ (Substituting value of } q_2) \\
&= q_1(a + b(b + ab)^*aa) + \varepsilon \\
&= \varepsilon(a + b(b + ab)^*aa)^* \\
&= (a + b(b + ab)^*aa)^*
\end{aligned}$$

Hence, the regular expression is

$$(a + b(b + ab)^*aa)^*.$$

Construct a regular expression corresponding to the automata given below –



Solution –

Here the initial state is q_1 and the final state is q_2

Now we write down the equations –

$$q_1 = q_1 0 + \varepsilon$$

$$q_2 = q_1 1 + q_2 0$$

$$q_3 = q_2 1 + q_3 0 + q_3 1$$

Now, we will solve these three equations –

$$q_1 = \varepsilon 0^* \text{ [As, } \varepsilon R = R]$$

$$\text{So, } q_1 = 0^*$$

$$q_2 = 0^* 1 + q_2 0$$

$$\text{So, } q_2 = 0^* 1 (0)^* \text{ [By Arden's theorem]}$$

Hence, the regular expression is $0^* 1 0^*$.

Pumping Lemma For Regular Languages

- ❑ **Pumping:** The word pumping refers to generating many input strings by pushing a symbol in an input string repeatedly.
- ❑ **Lemma:** The word Lemma refers to the intermediate theorem in a proof.

Theorem: If A is a Regular Language, then A has a Pumping Length ' P ' such that any string ' S ' where $|S| \geq P$ may be divided into three parts $S = xyz$ such that the following conditions must be true:

1.) $xy^kz \in A$ for every $k \geq 0$

2.) $|y| > 0$

3.) $|xy| \leq P$

- Pumping Lemma is used as proof of the **irregularity** of a language. It means, that if a language is regular, it always satisfies the pumping lemma. If at least one string is made from pumping, not in language A , then A is not regular.
- We use the **CONTRADICTION** method to prove that a language is not Regular.

For **example**, let us prove $L_{01} = \{0^n 1^n \mid n \geq 0\}$ is irregular.

- Let us assume that L is regular, then by Pumping Lemma the above given rules follow.
- So, by Pumping Lemma, there exists x, y, z such that (1) – (3) hold.
- We show that for all x, y, z , (1) – (3) does not hold.

- If (1) and (2) hold then $w = 0^n 1^n = uvw$ with $|xy| \leq n$ and $|y| \geq 1$.
- So, $x = 0^a$, $y = 0^b$, $z = 0^c 1^n$ where : $a + b \leq n$, $b \geq 1$, $c \geq 0$, $a + b + c = n$
- But, then (3) fails for $k = 0$, $xy^{0z} = uw = 0^a 0^c 1^n = 0^{a+c} 1^n \notin L$, since $a + c \neq n$.

Example 2:

Example: Using Pumping Lemma, prove that the language **A** = $\{a^n b^n \mid n \geq 0\}$ is **Not Regular**.

Solution: We will follow the steps we have learned above to prove this.

Assume that A is Regular and has a Pumping length = P.

Let a string **S** = $a^P b^P$.

Now divide the S into the parts, x y z.

To divide the S, let's take the value of P = 7.

Therefore, S = aaaaaaabbabbbbbb (by putting P=7 in S = $a^P b^P$).

For all the above cases, we need to show $xy^kz \notin A$ for some k .

Let the value of $k = 2$. $xy^kz \Rightarrow xy^2z$

$xy^2z = aa\ aaaa\ aaaa\ abbbbbbb$

No of 'a' = 11, No. of 'b' = 7.

Since the **No of 'a' \neq No. of 'b'**, but the original language has an equal number of 'a' and 'b'; therefore, this string will not lie in our language.

Closure properties of Regular languages:

In an automata theory, there are different closure properties for regular languages. They are as follows –

- Union
- Intersection
- Concatenation
- Kleene closure
- Complement

Union

- If L_1 and L_2 are two regular languages, their union $L_1 \cup L_2$ will also be regular.

Example

- $L_1 = \{a^n \mid n \geq 0\}$
- $L_2 = \{b^n \mid n \geq 0\}$
- $L_3 = L_1 \cup L_2 = \{a^n \cup b^n \mid n \geq 0\}$ is also regular.

Intersection

- If L_1 and L_2 are two regular languages, their intersection $L_1 \cap L_2$ will also be regular.

Example

- $L_1 = \{a^m b^n \mid n > 0 \text{ and } m > 0\}$ and
- $L_2 = \{a^m b^n \cup b^n a^m \mid n > 0 \text{ and } m > 0\}$
- $L_3 = L_1 \cap L_2 = \{a^m b^n \mid n > 0 \text{ and } m > 0\}$ are also regular.

Concatenation

- If L_1 and L_2 are two regular languages, their concatenation $L_1.L_2$ will also be regular.

Example

- $L_1 = \{a^m \mid m > 0\}$
- $L_2 = \{b^n \mid n > 0\}$
- $L_3 = L_1.L_2 = \{a^m . b^n \mid m > 0 \text{ and } n > 0\}$ is also regular.

Kleene Closure

- If $L1$ is a regular language, its Kleene closure $L1^*$ will also be regular.

Example

- $L1 = (a \cup b)$
- $L1^* = (a \cup b)^*$

Complement

- If $L(G)$ is a regular language, its complement $L'(G)$ will also be regular. Complement of a language can be found by subtracting strings which are in $L(G)$ from all possible strings.

Example

- $L(G) = \{an \mid n > 3\}$
- $L'(G) = \{an \mid n \leq 3\}$

Grammar:

▣ Formal Definition of Grammar :

Any Grammar can be represented by 4 tuples –
 $\langle N, T, P, S \rangle$

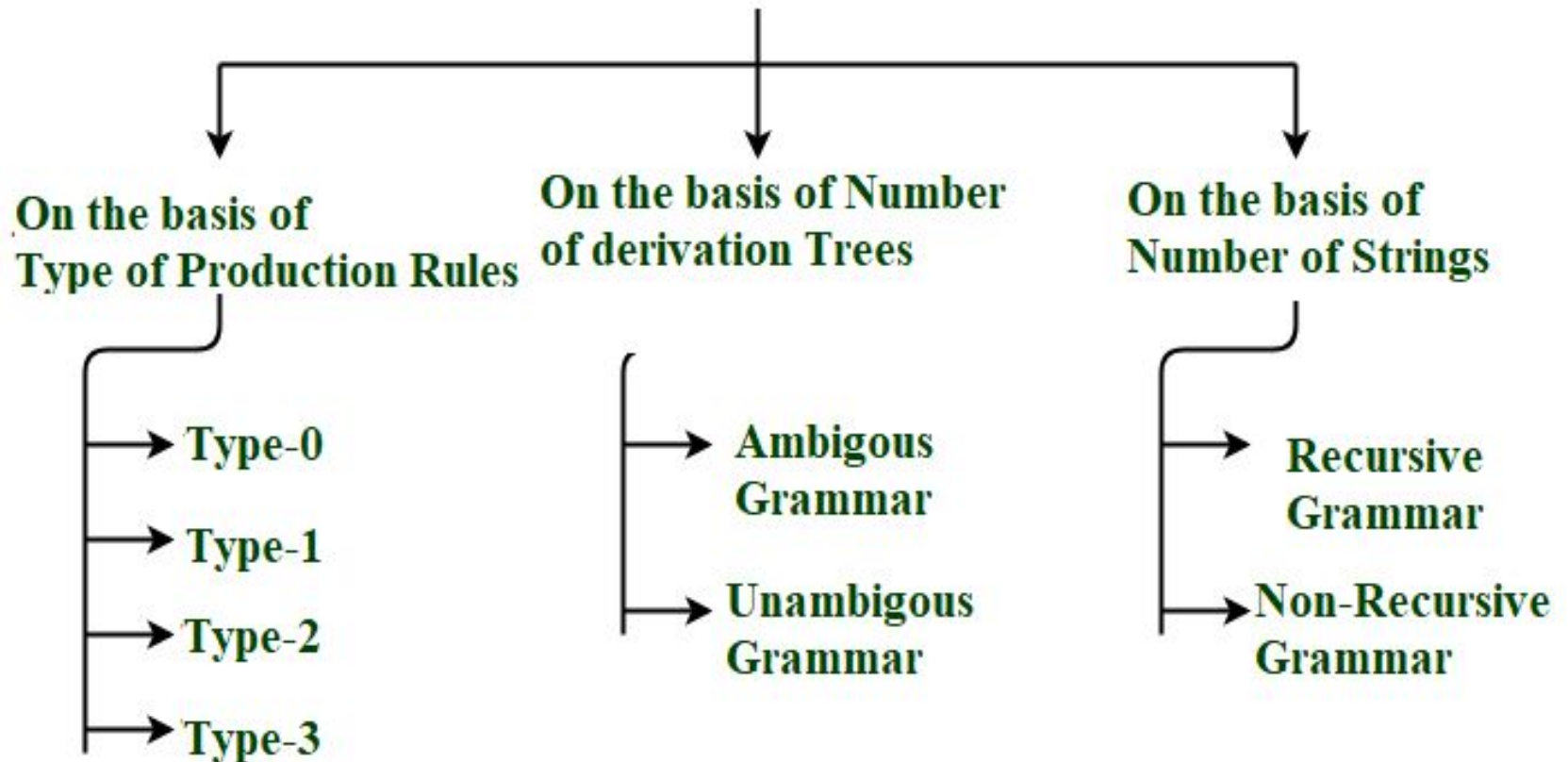
N – Finite Non-Empty Set of Non-Terminal Symbols.

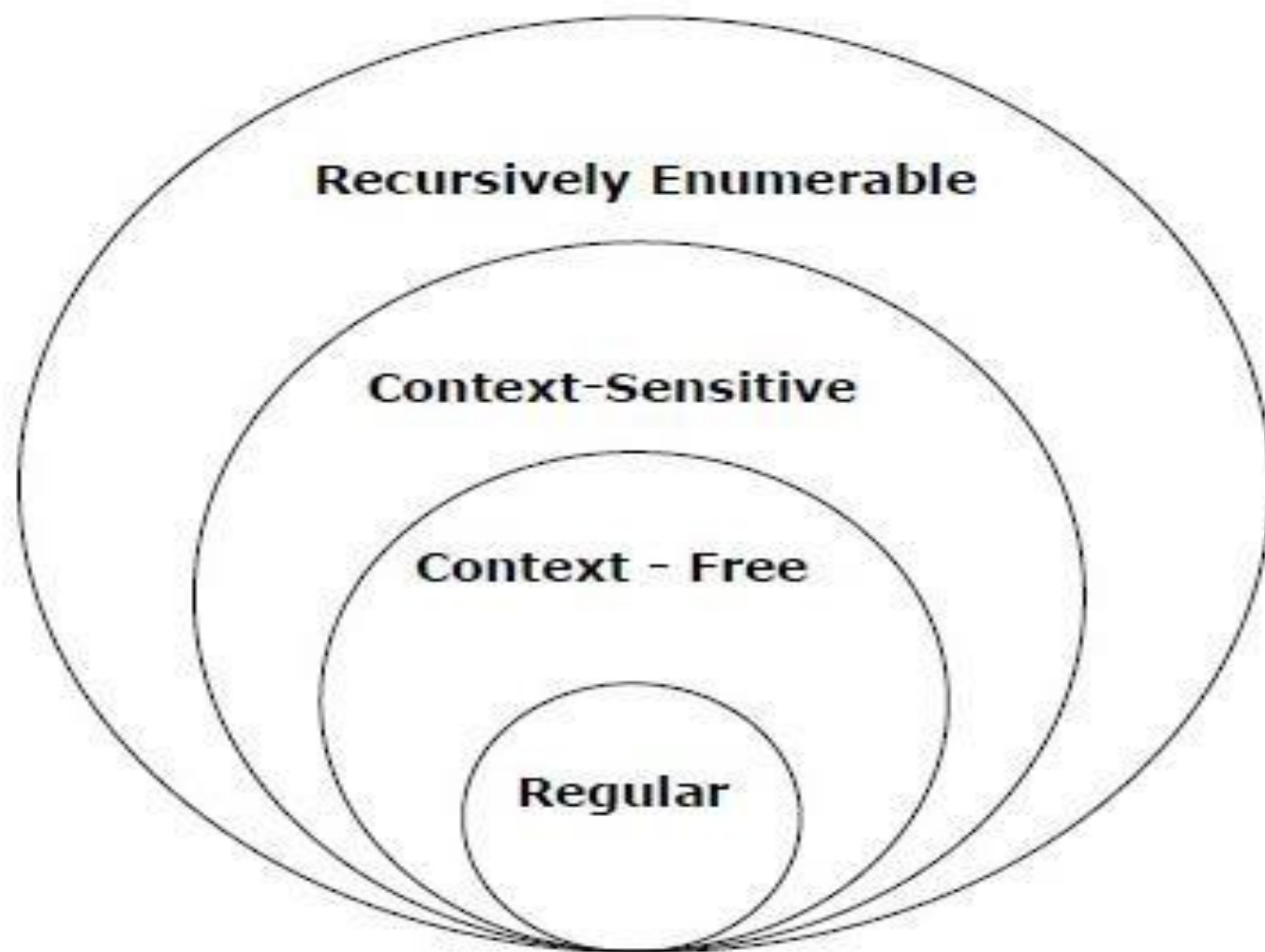
T – Finite Set of Terminal Symbols.

P – Finite Non-Empty Set of Production Rules.

S – Start Symbol (Symbol from where we start producing our sentences or strings).

Types of Grammar





Regular grammar:

- They have a single non-terminal symbol on the left-hand side, a single terminal on the right-hand side, or a single terminal followed by a non-terminal.
- Regular Grammar accepts and generates regular languages.

- All the productions should be in the form of:

$$A \rightarrow xB$$

$$A \rightarrow x$$

$$A \rightarrow Bx$$

where $A, B \in V$ (Set of Variables), and $x \in T^*$

i.e., the string of terminals.

Types of Regular Grammar

- Regular Grammar can be divided into two types:-
 - 1.) Left Linear Grammar (LLG)
 - 2.) Right Linear Grammar (RLG)

1.) Left Linear Grammar(LLG): A grammar is said to be Left Linear if all productions are of the form:

$$A \rightarrow Bx$$

$$A \rightarrow x$$

where $A, B \in V$ (Non-Terminal Symbols), and $x \in T^*$, i.e., the string of terminals.

In LLG, the non-terminal symbol lies on the Left of the terminal symbol.

2.) Right Linear Grammar(RLG): A grammar is said to be Right Linear if all productions are of the form:

$$A \rightarrow xB$$

$$A \rightarrow x$$

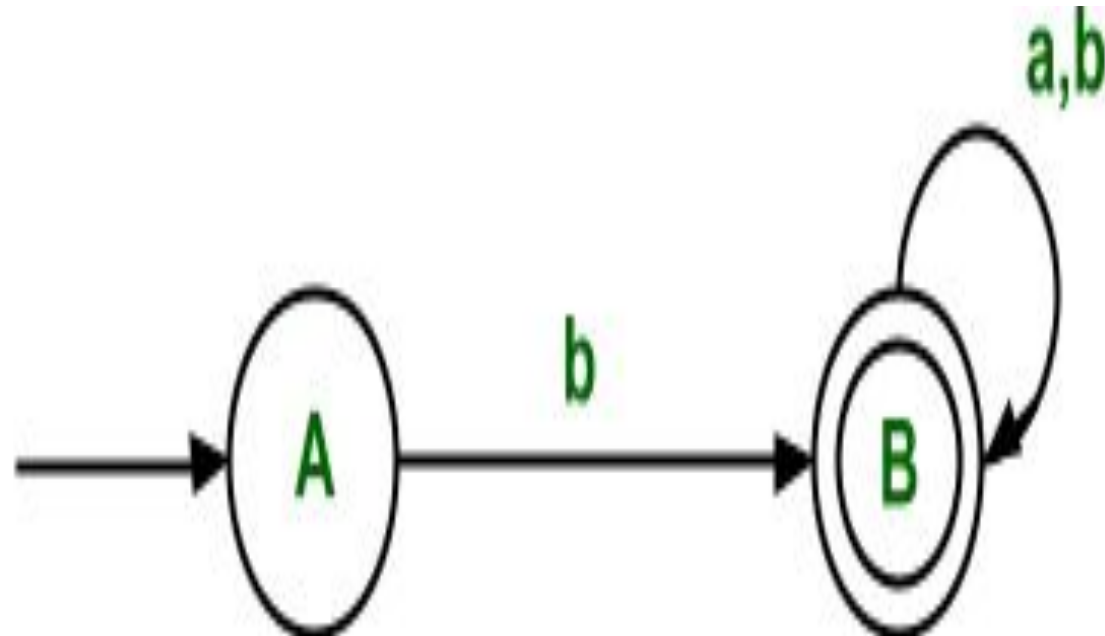
where $A, B \in V$ (Non-Terminal Symbols), and $x \in T^*$, i.e., the string of terminals.

In RLG, the non-terminal symbol lies on the Right of the terminal symbol.

Regular grammar to FA:

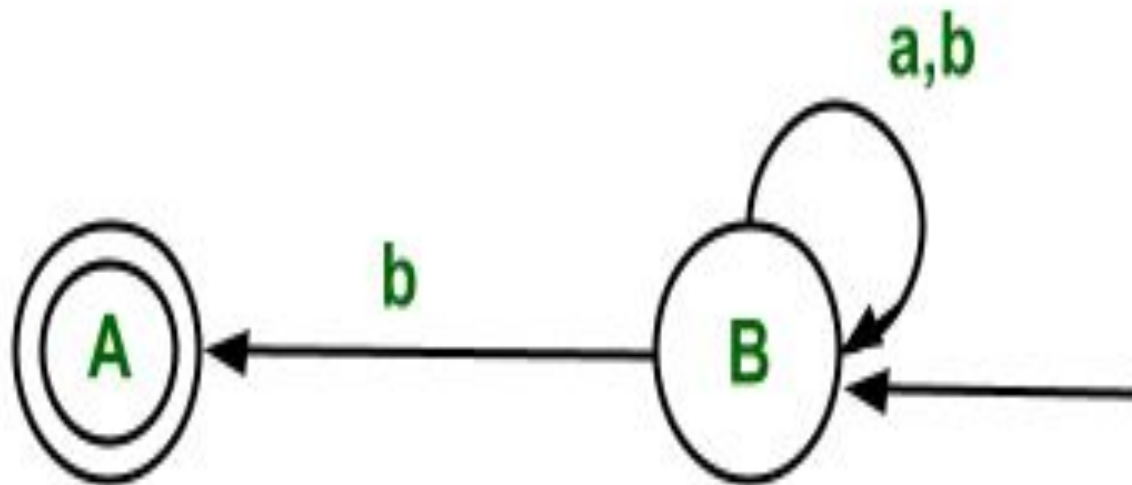
$A \rightarrow bB$

$B \rightarrow \epsilon / aB / bB$



$B \rightarrow aB/bB/bA$

$A \rightarrow \epsilon$

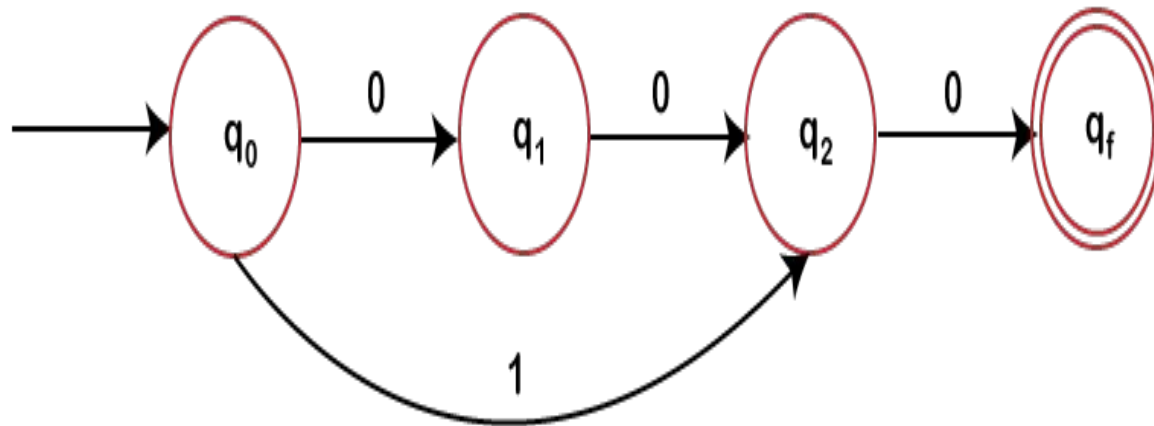


$A_0 \rightarrow 0A_1$

$A_0 \rightarrow 1A_2$

$A_1 \rightarrow 0A_2$

$A_2 \rightarrow 0$



FA to Regular grammar:

Pick the start state A and output is on symbol 'a' going to state B

$$A \rightarrow aB$$

Now we will pick state B and then we will go on each output

$$\text{i.e } B \rightarrow aB$$

$$B \rightarrow bB$$

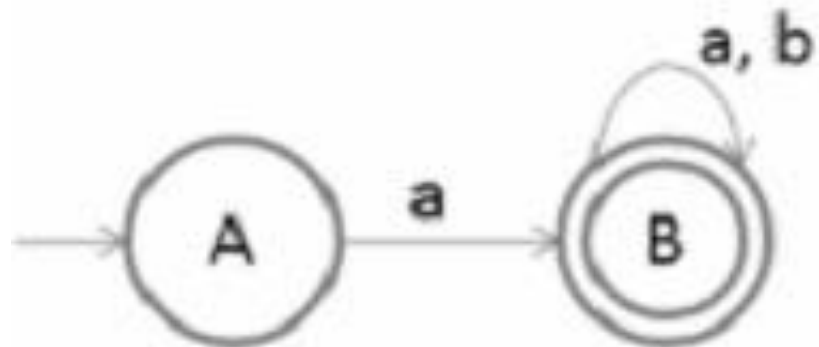
$$B \rightarrow \epsilon$$

Therefore,

Final right linear grammar is as follows –

$$A \rightarrow aB$$

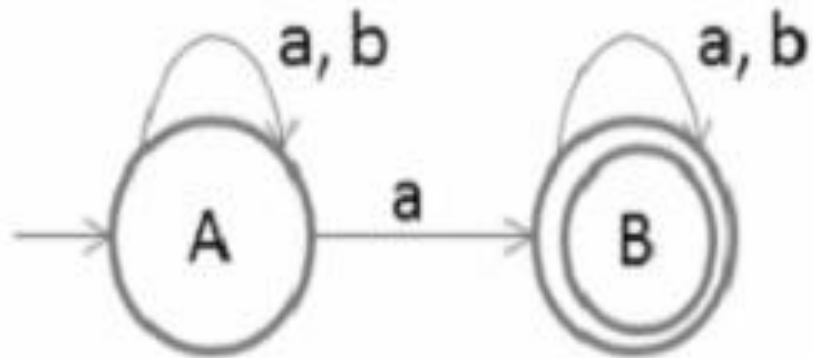
$$B \rightarrow aB / bB / \epsilon$$



□ Example

$A \rightarrow aA / bA / aB$

$B \rightarrow aB / bB / \varepsilon$



Conversion of LLG to RLG:



Context-free grammar:

- A grammar is said to be the context free grammr if every production rule is in the form
 $A \rightarrow \alpha$
where A is V and α is $(V \cup T)^*$
- Context free grammar is a formal grammar which is used to generate all possible strings in a given formal language.

- $L = \{wcwR \mid w \in (a, b)^*\}$

- Production rules: $S \rightarrow aSa$

$$S \rightarrow bSb$$

$$S \rightarrow c$$

- Now check that `abbcbbba` string can be derived from the given CFG.

$$S \Rightarrow aSa \Rightarrow abSba \Rightarrow abbSbba \Rightarrow abbcbbba$$

Designing of CFG:

Construct the CFG for the language having any number of a 's over the set $\Sigma = \{a\}$

□ **RE = a^***

Production rule for the Regular expression is as follows:

$S \rightarrow aS$ rule 1

$S \rightarrow \epsilon$ rule 2

Construct a CFG for the regular expression $(0+1)^*$

The CFG can be given by,

Production rule (P):

$$S \rightarrow 0S \mid 1S$$

$$S \rightarrow \epsilon$$

Construct a CFG for a language $L = \{wcwR \mid w \in (a, b)^*\}$.

The string that can be generated for a given language is $\{aaca, bcb, abcba, bacab, abbcbb, \dots\}$

The grammar could be:

$S \rightarrow aSa$ rule 1

$S \rightarrow bSb$ rule 2

$S \rightarrow c$ rule 3

Construct a CFG for the language $L = a^n b^{2n}$ where $n \geq 1$.

The string that can be generated for a given language is $\{abb, aabbbb, aaabbbbbbb....\}$.

The grammar could be:

$S \rightarrow aSbb \mid abb$

String derivation:

- Derivation is a sequence of production rules. It is used to get the input string through these production rules. During parsing, we have to take two decisions. These are as follows:
 - We have to decide the non-terminal which is to be replaced.
 - We have to decide the production rule by which the non-terminal will be replaced.

1. Leftmost Derivation:

- In the leftmost derivation, the input is scanned and replaced with the production rule from left to right. So in leftmost derivation, we read the input string from left to right.

Production rules:

$$E = E + E$$

$$E = E - E$$

$$E = a \mid b$$

Input

$a - b + a$

The leftmost derivation is:

$$E = E + E$$

$$E = E - E + E$$

$$E = a - E + E$$

$$E = a - b + E$$

$$E = a - b + a$$

2. Rightmost Derivation:

- In rightmost derivation, the input is scanned and replaced with the production rule from right to left. So in rightmost derivation, we read the input string from right to left.

Production rules:

$$E = E + E$$

$$E = E - E$$

$$E = a \mid b$$

Input

$a - b + a$

The rightmost derivation is:

$$E = E - E$$

$$E = E - E + E$$

$$E = E - E + a$$

$$E = E - b + a$$

$$E = a - b + a$$

Derivation tree or Parse tree

- Derivation tree is a graphical representation for the derivation of the given production rules of the context free grammar (CFG).
- It is a way to show how the derivation can be done to obtain some string from a given set of production rules. It is also called as the Parse tree.
- The Parse tree follows the precedence of operators.

Properties

The properties of the derivation tree are given below

—

- The root node is always a node indicating the start symbols.
- The derivation is read from left to right.
- The leaf node is always the terminal node.
- The interior nodes are always the non-terminal nodes.

Example

- The production rules for the derivation tree are as follows –

$$E = E + E$$

$$E = E * E$$

$$E = a \mid b \mid c$$

Here, let the input be **$a * b + c$**

Consider the Grammar given below –

$$E \Rightarrow E + E \mid E * E \mid id$$

- Find Leftmost and Rightmost Derivation for the string.

□ Leftmost Derivation

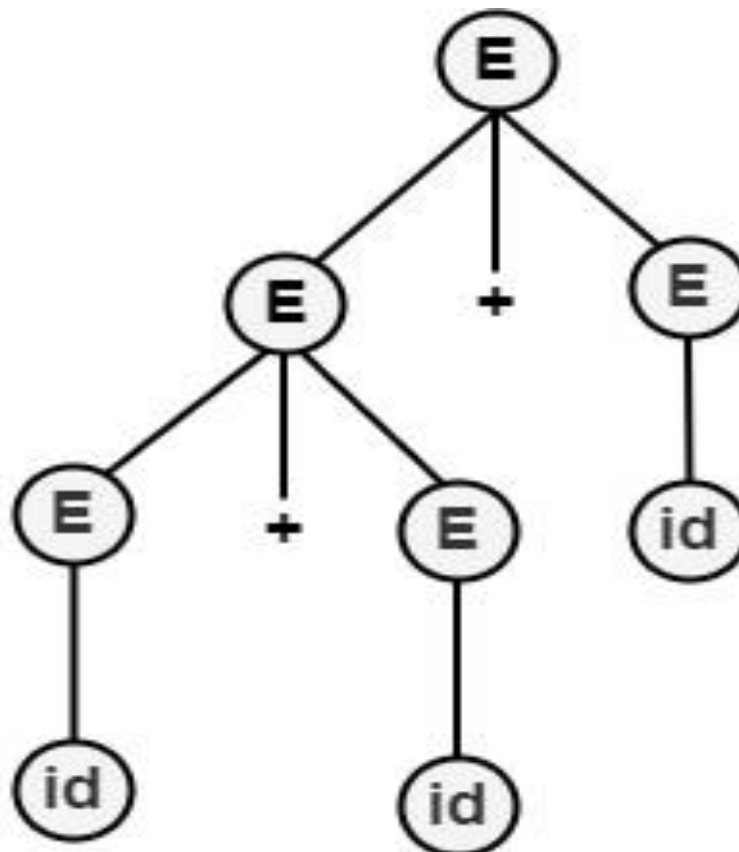
$E \Rightarrow E + E$

$\Rightarrow E + E + E$

$\Rightarrow id + E + E$

$\Rightarrow id + id + E$

$\Rightarrow id + id + id$



□ Rightmost Derivation

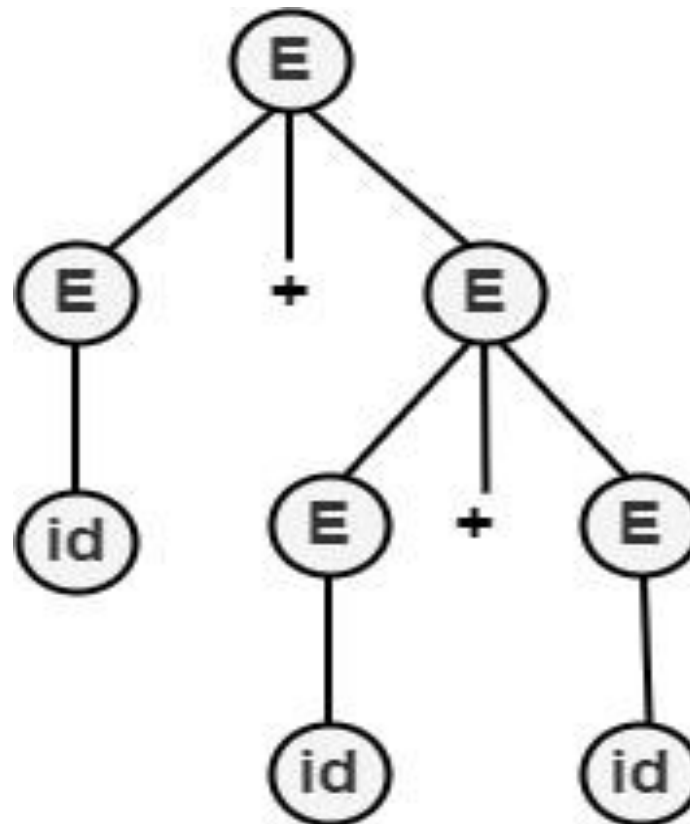
$E \Rightarrow E + E$

$\Rightarrow E + E + E$

$\Rightarrow E + E + id$

$\Rightarrow E + id + id$

$\Rightarrow id + id + id$



Ambiguity in grammar

- A grammar is said to be ambiguous if there exists more than one left most derivation or more than one right most derivation or more than one parse tree for a given input string.
 - If the grammar is not ambiguous then we call it unambiguous grammar.
 - If the grammar has ambiguity then it is good for compiler construction.

Let us consider a grammar with production rules, as shown below –

$$E = I$$

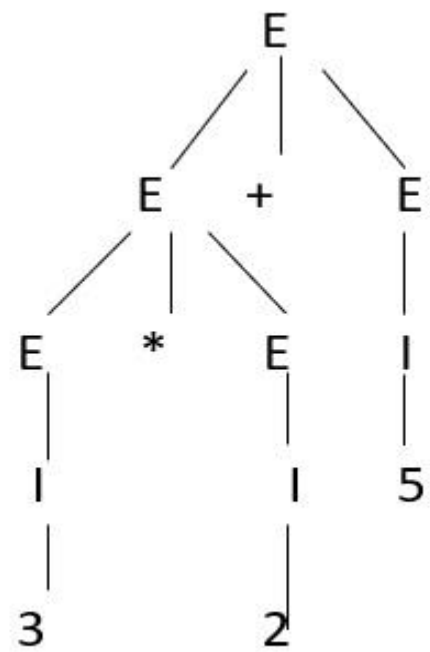
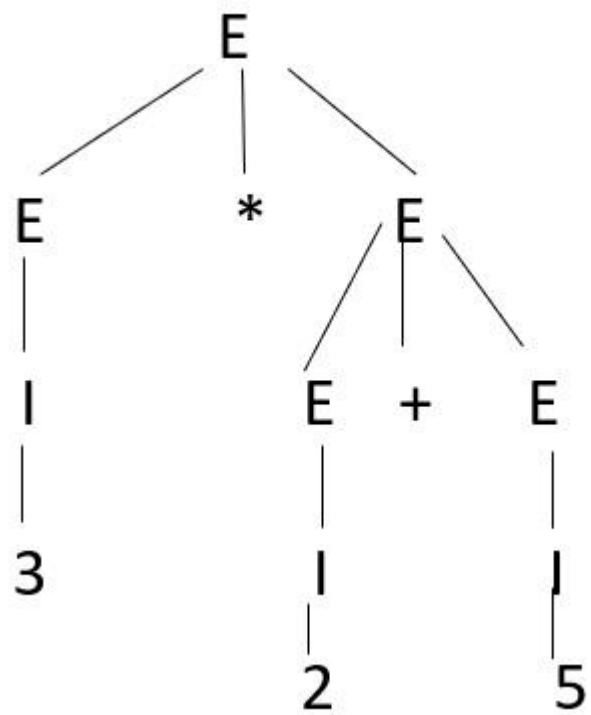
$$E = E + E$$

$$E = E * E$$

$$E = (E)$$

$$I = \varepsilon \mid 0 \mid 1 \mid 2 \mid 3 \dots 9$$

- Let's consider a string **"3*2+5"**



Reducing CFG:

Simplification essentially comprises of the following steps –

- Removal of useless symbols
- Removal of Unit Productions
- Removal of Null Productions

Chomsky Normal Form:

- CNF stands for Chomsky normal form. A CFG(context free grammar) is in CNF(Chomsky normal form) if all production rules satisfy one of the following conditions:
 - Start symbol generating ϵ . For example, $A \rightarrow \epsilon$.
 - A non-terminal generating two non-terminals. For example, $S \rightarrow AB$.
 - A non-terminal generating a terminal. For example, $S \rightarrow a$.

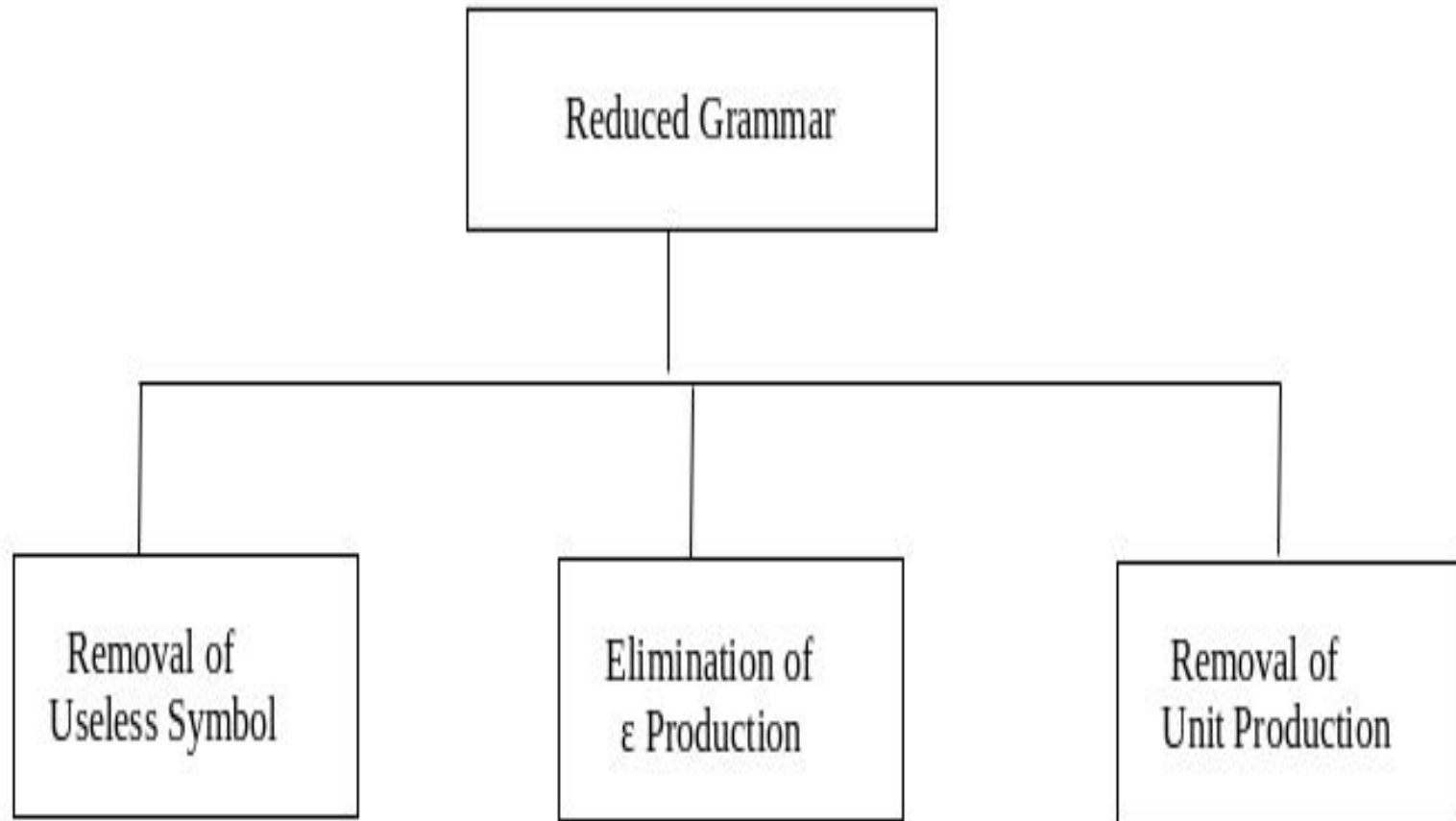
For example:

- $G1 = \{S \rightarrow AB, S \rightarrow c, A \rightarrow a, B \rightarrow b\}$
- $G2 = \{S \rightarrow aA, A \rightarrow a, B \rightarrow c\}$

$G1$ is in CNF but $G2$ is not in CNF because $s \rightarrow$

$S \rightarrow aA$ is violating the rules of CNF

Simplification/reduction of CFG



Removal of Useless Symbols

- A symbol can be useless if it does not appear on the right-hand side of the production rule and does not take part in the derivation of any string. That symbol is known as a useless symbol. Similarly, a variable can be useless if it does not take part in the derivation of any string. That variable is known as a useless variable.

Q) $T \rightarrow aaB \mid abA \mid aaT$

$A \rightarrow aA$

$B \rightarrow ab \mid b$

$C \rightarrow ad$

Here $C \rightarrow ad$ is removed because C is not present in the RHS of any production rule.

Also, $T \rightarrow abA$ and $A \rightarrow aA$ are removed because they cannot form a string.



Production rules after removal of useless symbols:

$$T \rightarrow aaB \mid aaT$$

$$B \rightarrow ab \mid b$$

Elimination of ϵ Production

- The productions of type $S \rightarrow \epsilon$ are called ϵ productions. These type of productions can only be removed from those grammars that do not generate ϵ .

Remove the production from the following CFG by preserving the meaning of it.

- $S \rightarrow XYX$
- $X \rightarrow 0X \mid \varepsilon$
- $Y \rightarrow 1Y \mid \varepsilon$

□ **Solution :**

Now, while removing ϵ production, we are deleting the rule $X \rightarrow \epsilon$ and $Y \rightarrow \epsilon$. To preserve the meaning of CFG we are actually placing ϵ at the right-hand side whenever X and Y have appeared.

- Let us take

$$S \rightarrow XYX$$

- If the first X at right-hand side is ε . Then

$$S \rightarrow YX$$

- Similarly if the last X in R.H.S. = ε . Then

$$S \rightarrow XY$$

- If $Y = \varepsilon$ then

$$S \rightarrow XX$$

- If Y and X are ε then,

$$S \rightarrow X$$

- If both X are replaced by ε

$$S \rightarrow Y$$

- Now,

$$S \rightarrow XYX \mid XY \mid YX \mid XX \mid X \mid Y$$

- Now let us consider $X \rightarrow 0X \mid \varepsilon$

If we place ε at right-hand side for X then, $X \rightarrow 0$

$$\mathbf{X \rightarrow 0X \mid 0}$$

- Similarly

$$\mathbf{Y \rightarrow 1Y \mid 1}$$

- Collectively we can rewrite the CFG with removed ε production as

$S \rightarrow XYX \mid XY \mid YX \mid XX \mid X \mid Y$

$X \rightarrow 0X \mid 0$

$Y \rightarrow 1Y \mid 1$

Removing Unit Productions

- The unit productions are the productions in which one non-terminal gives another non-terminal. Use the following steps to remove unit production:

For example:

$$S \rightarrow 0A \mid 1B \mid C$$

$$A \rightarrow 0S \mid 00$$

$$B \rightarrow 1 \mid A$$

$$C \rightarrow 01$$

Solution:

$S \rightarrow C$ is a unit production. But while removing $S \rightarrow C$ we have to consider what C gives. So, we can add a rule to S .

$$S \rightarrow 0A \mid 1B \mid 01$$

$$B \rightarrow 1 \mid 0S \mid 00$$

Thus finally we can write CFG without unit production as

$$\mathbf{S \rightarrow 0A \mid 1B \mid 01}$$

$$\mathbf{A \rightarrow 0S \mid 00}$$

$$\mathbf{B \rightarrow 1 \mid 0S \mid 00}$$

$$\mathbf{C \rightarrow 01}$$

Conversion of CFG to CNF

Step 1: Eliminate start symbol from the RHS. If the start symbol T is at the right-hand side of any production, create a new production as:

$$S1 \rightarrow S$$

Where $S1$ is the new start symbol.

Step 2: In the grammar, remove the null, unit and useless productions.

Step 3: Eliminate terminals from the RHS of the production if they exist with other non-terminals or terminals. For example, production $S \rightarrow aA$ can be decomposed as:

$$S \rightarrow RA$$

$$R \rightarrow a$$

Step 4: Eliminate RHS with more than two non-terminals. For example, $S \rightarrow ASB$ can be decomposed as:

$$S \rightarrow RS$$

$$R \rightarrow AS$$

Example:

Convert the given CFG to CNF. Consider the given grammar G_1 :

$$S \rightarrow a \mid aA \mid B$$

$$A \rightarrow aBB \mid \varepsilon$$

$$B \rightarrow Aa \mid b$$

□ Solution:

Step 1: We will create a new production $S1 \rightarrow S$, as the start symbol S appears on the RHS. The grammar will be:

$$S1 \rightarrow S$$

$$S \rightarrow a \mid aA \mid B$$

$$A \rightarrow aBB \mid \varepsilon$$

$$B \rightarrow Aa \mid b$$

Step 2: As grammar $G1$ contains $A \rightarrow \varepsilon$ null production, its removal from the grammar yields:

$$S1 \rightarrow S$$

$$S \rightarrow a \mid aA \mid B$$

$$A \rightarrow aBB$$

$$B \rightarrow Aa \mid b \mid a$$

Now, as grammar G_1 contains Unit production $S \rightarrow B$,
its removal yield:

$$S_1 \rightarrow S$$

$$S \rightarrow a \mid aA \mid Aa \mid b$$

$$A \rightarrow aBB$$

$$B \rightarrow Aa \mid b \mid a$$

- Also remove the unit production $S1 \rightarrow S$, its removal from the grammar yields:

$$S0 \rightarrow a \mid aA \mid Aa \mid b$$

$$S \rightarrow a \mid aA \mid Aa \mid b$$

$$A \rightarrow aBB$$

$$B \rightarrow Aa \mid b \mid a$$

Step 3: In the production rule $S0 \rightarrow aA \mid Aa$, $S \rightarrow aA \mid Aa$, $A \rightarrow aBB$ and $B \rightarrow Aa$, terminal a exists on RHS with non-terminals. So we will replace terminal a with X :

$$S0 \rightarrow a \mid XA \mid AX \mid b$$

$$S \rightarrow a \mid XA \mid AX \mid b$$

$$A \rightarrow XBB$$

$$B \rightarrow AX \mid b \mid a$$

$$X \rightarrow a$$

Step 4: In the production rule $A \rightarrow XBB$, RHS has more than two symbols, removing it from grammar yield:

$$S0 \rightarrow a \mid XA \mid AX \mid b$$

$$S \rightarrow a \mid XA \mid AX \mid b$$

$$A \rightarrow RB$$

$$B \rightarrow AX \mid b \mid a$$

$$X \rightarrow a$$

$$R \rightarrow XB$$

- Hence, for the given grammar, this is the required CNF.

Types of grammar (on the basis of no.of strings)

1. Recursive Grammar-

- A grammar is said to be recursive if it contains at least one production that has the same variable at both its LHS and RHS.

OR

- A grammar is said to be recursive if and only if it generates infinite number of strings.

A) Left Recursive Grammar-

- A recursive grammar is said to be left recursive if the leftmost variable of RHS is same as variable of LHS.

$$S \rightarrow Sa / b$$

(Left Recursive Grammar)

B) Right Recursive Grammar-

- A recursive grammar is said to be right recursive if the rightmost variable of RHS is same as variable of LHS.

$$S \rightarrow aS / b$$

2. Non-Recursive Grammar-

- A grammar is said to be non-recursive if it contains no production that has the same variable at both its LHS and RHS.

$$S \rightarrow aA / bB$$

$$A \rightarrow a / b$$

$$B \rightarrow c / d$$

Eliminate left recursion

A Grammar $G (V, T, P, S)$ is left recursive if it has a production in the form.

$$A \rightarrow A \alpha \mid \beta$$

It can eliminate left recursion by replacing a pair of production with

$$A \rightarrow \beta A'$$

$$A' \rightarrow \alpha A' \mid \epsilon$$

Example – Consider the Left Recursion from the Grammar.

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

$$F \rightarrow (E) \mid \text{id}$$

Eliminate immediate left recursion from the Grammar.

Solution:

Comparing $E \rightarrow E + T \mid T$ with $A \rightarrow A \alpha \mid \beta$

- $E \rightarrow E + T \mid T$ $A \rightarrow A \alpha \mid \beta$. $\therefore A = E, \alpha = +T, \beta = T$
- $\therefore A \rightarrow A \alpha \mid \beta$ is changed to $A \rightarrow \beta A'$ and $A' \rightarrow \alpha A' \mid \epsilon$
- $\therefore A \rightarrow \beta A'$ means $E \rightarrow TE'$
- $A' \rightarrow \alpha A' \mid \epsilon$ means $E' \rightarrow +TE' \mid \epsilon$

Comparing $T \rightarrow T * F \mid F$ with $A \rightarrow A\alpha \mid \beta$

- $T \rightarrow T * F \mid F$ $A \rightarrow A\alpha \mid \beta$. $\therefore A = T, \alpha = * F, \beta = F$
- $\therefore A \rightarrow \beta A'$ means $T \rightarrow FT'$
- $A \rightarrow \alpha A' \mid \varepsilon$ means $T' \rightarrow^* FT' \mid \varepsilon$

Production $F \rightarrow (E) \mid id$ does not have any left recursion

- ∴ Combining productions 1, 2, 3, 4, 5, we get

$$E \rightarrow TE'$$

$$E' \rightarrow +TE' \mid \varepsilon$$

$$T \rightarrow FT'$$

$$T \rightarrow^* FT' \mid \varepsilon$$

$$F \rightarrow (E) \mid \text{id}$$

Example – Eliminate the left recursion for the following Grammar.

$$S \rightarrow a \mid ^ \mid (T)$$

$$T \rightarrow T, S \mid S$$

Solution:

$$S \rightarrow a \mid ^ \mid (T)$$

$$T \rightarrow ST'$$

$$T' \rightarrow ,ST' \mid \varepsilon$$

Example – Eliminate the left recursion for the following Grammar.

$$S \rightarrow a \mid ^ \mid (T)$$

$$T \rightarrow T, S \mid S$$

Solution:

$$S \rightarrow a \mid ^ (T)$$

$$T \rightarrow ST'$$

$$T' \rightarrow ,ST' \mid \varepsilon$$

Example – Remove the left recursion from the grammar

$$E \rightarrow E(T) \mid T$$

$$T \rightarrow T(F) \mid F$$

$$F \rightarrow \text{id}$$

Solution : Eliminating immediate left-recursion among all $A\alpha$ productions, we obtain

$$E \rightarrow TE'$$

$$E' \rightarrow (T)E' \mid \varepsilon$$

$$T \rightarrow FT'$$

$$T' \rightarrow (F)T' \mid \varepsilon$$

$$F \rightarrow \text{id}$$

Greibach Normal Form (GNF)

- GNF stands for Greibach normal form. A CFG(context free grammar) is in GNF(Greibach normal form) if all the production rules satisfy one of the following conditions:
 - A start symbol generating ϵ . For example, $S \rightarrow \epsilon$.
 - A non-terminal generating a terminal. For example, $A \rightarrow a$.
 - A non-terminal generating a terminal which is followed by any number of non-terminals. For example, $S \rightarrow aASB$.

For example:

$$G1 = \{S \rightarrow aAB \mid aB, A \rightarrow aA \mid a, B \rightarrow bB \mid b\}$$

$$G2 = \{S \rightarrow aAB \mid aB, A \rightarrow aA \mid \varepsilon, B \rightarrow bB \mid \varepsilon\}$$

- The production rules of Grammar G1 satisfy the rules specified for GNF, so the grammar G1 is in GNF. However, the production rule of Grammar G2 does not satisfy the rules specified for GNF as $A \rightarrow \varepsilon$ and $B \rightarrow \varepsilon$ contains ε (only start symbol can generate ε). So the grammar G2 is not in GNF.

Converting CFG into GNF

- Step 1** – Convert the grammar into CNF. If the given grammar is not in CNF, convert it into CNF.
- Step 2** – If the grammar consists of left recursion, eliminate it. If the context free grammar contains any left recursion, eliminate it.
- Step 3** – In the grammar, convert the given production rule into GNF form. If any production rule in the grammar is not in GNF form, convert it.

Example

Consider the context free grammar

$$S \rightarrow SS \mid (S) \mid a$$

Convert this grammar to Greibach Normal Form.

Solution

Given below is an explanation for conversion of CFG to GNF –

Step 1: Converting to CNF:

$$S \rightarrow SS \mid XSY \mid a$$
$$X \rightarrow ($$
$$Y \rightarrow)$$

Step 2: Remove left recursion from

$S \rightarrow SS$

$S \rightarrow XSYP/aP$

$P \rightarrow SP/\epsilon$

$X \rightarrow ($

$Y \rightarrow)$

Step 3: Remove null production $P \rightarrow \epsilon$

$S \rightarrow XSYP/aP$

$P \rightarrow SP/S$

$X \rightarrow ($

$Y \rightarrow)$

Step 4: Convert to GNF as $S \rightarrow XSYP$ is not in GNF,

Replace X with (

$S \rightarrow (SYP/aP$

$P \rightarrow SP/S$

$X \rightarrow ($

$Y \rightarrow)$

Step 5: Convert to GNF as $P \rightarrow SP$ is not in GNF,
Replace S with $(SYP/aP$

$S \rightarrow (SYP/aP$

$P \rightarrow (SYPP/aPP/(SYP/aP$

$X \rightarrow ($

$Y \rightarrow)$

Closure properties of Context free languages:

Context free languages can be generated by context free grammars.

- **Union :** If L_1 and L_2 are two context free languages, their union $L_1 \cup L_2$ will also be context free. For example,

$$L_1 = \{ a^n b^n c^m \mid m \geq 0 \text{ and } n \geq 0 \} \text{ and } L_2 = \{ a^n b^m c^m \mid n \geq 0 \text{ and } m \geq 0 \}$$

$$L_3 = L_1 \cup L_2 = \{ a^n b^n c^m \cup a^n b^m c^m \mid n \geq 0, m \geq 0 \} \text{ is also context free.}$$

- **Concatenation** : If $L1$ and $L2$ are two context free languages, their concatenation $L1.L2$ will also be context free. For example,
$$L1 = \{ a^n b^n \mid n \geq 0 \}$$
$$L2 = \{ c^m d^m \mid m \geq 0 \}$$
$$L3 = L1.L2$$
$$= \{ a^n b^n c^m d^m \mid m \geq 0 \text{ and } n \geq 0 \}$$
 is also context free.

- **Kleene Closure** : If $L1$ is context free, its Kleene closure $L1^*$ will also be context free. For example,
 $L1 = \{ a^n b^n \mid n \geq 0 \}$
 $L1^* = \{ a^n b^n \mid n \geq 0 \}^*$ is also context free.

Pumping lemma for Context free languages:

- Pumping Lemma for CFL states that for any Context-Free Language L , it is possible to find two substrings that can be 'pumped' any number of times and still be in the same language. We break its strings into five parts for any language L and pump the second and fourth substrings. If any string does not satisfy its conditions, then the language is not CFL.

Theorem

- If L is a context-free language, there is a constant ' n ' that depends exclusively on L , such that if $w \in L$ and $|w| \geq n$, w can be divided into five pieces, $w = uvxyz$, meeting the following requirements.
- $|vxy| \geq n$
- $|vy| \leq \epsilon$
- For all $k \geq 0$, the string $u v^k x y^k z \in L$

□ Example

Find out whether $L = \{X^n Y^n Z^n \mid n \geq 1\}$ is context-free or not

Let L be context-free.

' L ' must be satisfying the pumping length, say n .

Now we can take a string such that $s = x^n y^n z^n$

We divide s into 5 strings $uvxyz$.

Let $n=4$ so, $s = x^4 y^4 z^4$

Case 1

v and y each contain only one type of symbol.

{we are considering only v and y because v and y has power uv^2xy^2z }

$X \ xx \ x \ yyyyz \ z \ zz$

$= uv^k x y^k z$ when $k=2$

$= uv^2 x y^2 z$

$= xxxxxx yyy y zzzzz$

$= x^6 y^4 z^5$

(Number of x # number of y # number of z)

Therefore, The resultant string is not satisfying the condition

$x^6 y^4 z^5 \notin L$

If one case fails, there is no need to check another condition.

Non- context free grammar:

- Every regular language is context free.
Example – $\{a^m b^l c^k d^n \mid m, l, k, n \geq 1\}$ is context free, as it is regular too.
- Given an expression such that it is possible to obtain a center or mid point in the strings, so we can carry out comparison of left and right sub-parts using stack.
Example 1 – $L = \{a^n b^n \mid n \geq 1\}$ is context free, as we can push a's and then we can pop a's for each occurrence of b.
Example 2 – $L = \{a^n b^n c^{(m+n)}\}$ is context free. We can rewrite it as $\{a^n b^n c^m c^n\}$.

Example 3 – $L = \{a^n b^{(2n)}\}$ is context free, as we can push two a's and pop an a for each occurrence of b. Hence, we get a mid-point here as well.

Example 4 – $L = \{a^n b^n c^n\}$ is not context free.

- Given expression is a combination of multiple expressions with mid-points in them, such that each sub-expression is independent of other sub-expressions, then it is context free.

Example 1 – $L = \{a^m b^m c^n d^n\}$ is context free. It contains multiple expressions with a mid-point in each of them.

Example 2 – $L = \{a^m b^n c^m d^n\}$ is not context free.

- An expression that doesn't form a pattern on which linear comparison could be carried out using stack is not context free language.

Example 1 – $L = \{ a^m b^{n^2} \}$ is not context free.

Example 2 – $L = \{ a^n b^{2^n} \}$ is not context free.

Example 3 – $L = \{ a^{n^2} \}$ is not context free.

Example 4 – $L = \{ a^m \mid m \text{ is prime} \}$ is not context free.

- An expression that involves counting and comparison of three or more variables independently is not context free language, as stack allows comparison of only two variables at a time.

Example 1 – $L = \{a^n b^n c^n\}$ is not context free.

Example 2 – $L = \{a^i b^j c^k \mid i > j > k\}$ is not context free.

- A point to remember is counting and comparison could only be done with the top of stack and not with bottom of stack in Push Down Automata, hence a language exhibiting a characteristic that involves comparison with bottom of stack is not a context free language.

Example 1 – $L = \{a^m b^n c^m d^n\}$ is not context free.

Pushing a's first then b's. Now, we will not be able to compare c's with a's as the top of the stack has b's.

UNIT-3

Push Down Automata: Design of Deterministic PDA and Non-deterministic PDA, PDA to CFG and CGA to PDA conversion.

PDA (Push Down Automata)

- Pushdown automata is a way to implement a CFG in the same way we design DFA for a regular grammar.
- A DFA can remember a finite amount of information, but a PDA can remember an infinite amount of information.
- A PDA is more powerful than FA. Any language which can be acceptable by FA can also be acceptable by PDA.
- PDA also accepts a class of language which even cannot be accepted by FA. Thus PDA is much more superior to FA.

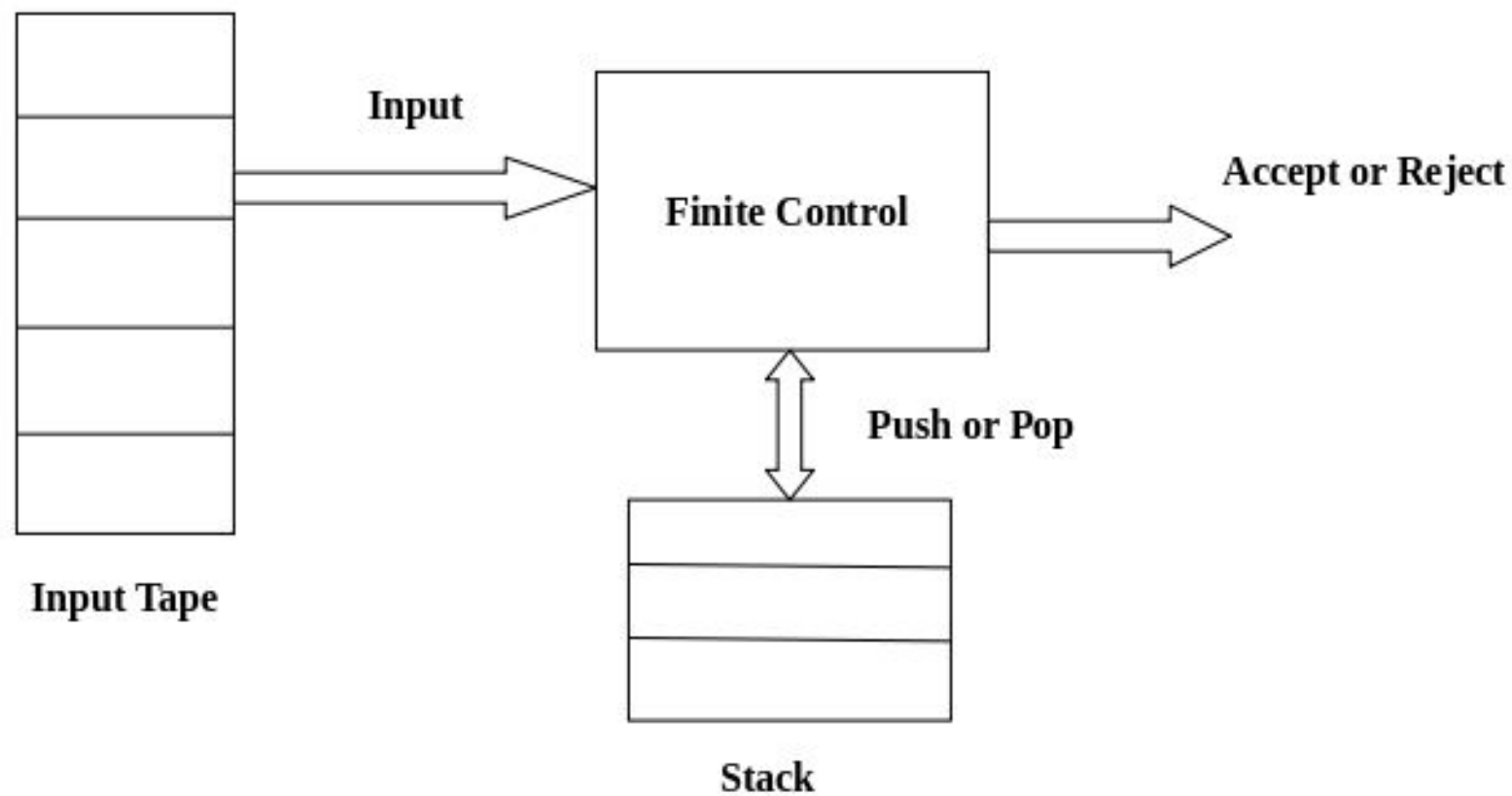


Fig: Pushdown Automata

PDA Components:

- ❑ **Input tape:** The input tape is divided in many cells or symbols. The input head is read-only and may only move from left to right, one symbol at a time.
- ❑ **Finite control:** The finite control has some pointer which points the current symbol which is to be read.
- ❑ **Stack:** The stack is a structure in which we can push and remove the items from one end only. It has an infinite size. In PDA, the stack is used to store the items temporarily.

Formal definition of PDA:

- The PDA can be defined as a collection of 7 components:

Q: the finite set of states

Σ : the input set

Γ : a stack symbol which can be pushed and popped from the stack

q0: the initial state

Z: a start symbol which is in Γ .

F: a set of final states

δ : mapping function which is used for moving from current state to next state.

Instantaneous Description (ID)

- ID is an informal notation of how a PDA computes an input string and make a decision that string is accepted or rejected.
- **An instantaneous description is a triple (q, w, α) where:**
 - q** describes the current state.
 - w** describes the remaining input.
 - α** describes the stack contents, top at the left.

Turnstile Notation:

- \vdash sign describes the turnstile notation and represents one move.
- \vdash^* sign describes a sequence of moves.
- **For example,**

$$(p, b, T) \vdash (q, w, \alpha)$$

In the above example, while taking a transition from state p to q , the input symbol 'b' is consumed, and the top of the stack 'T' is represented by a new string α .

Turing machine

- Turing Machine was invented by Alan Turing in 1936 and it is used to accept Recursive Enumerable Languages (generated by Type-0 Grammar).
- It provides a mathematical model of a simple abstract computer.
- Turing machines are used to study the properties of algorithms and to determine what problems can and cannot be solved by computers.
- They provide a way to model the behavior of algorithms and to analyze their computational complexity, which is the amount of time and memory they require to solve a problem.

- Turing machine is a finite automaton that can read, write, and erase symbols on an infinitely long tape.
- The tape is divided into squares, and each square contains a symbol.
- The Turing machine can only read one symbol at a time, and it uses a set of rules (the transition function) to determine its next action based on the current state and the symbol it is reading.
- The Turing machine begins in the start state and performs the actions specified by the transition function until it reaches an accept or reject state.

- If it reaches an accept state, the computation is considered successful; if it reaches a reject state, the computation is considered unsuccessful.
- The tape consists of infinite cells on which each cell either contains input symbol or a special symbol called blank.
- It also consists of a head pointer which points to cell currently being read and it can move in both directions

Formal definition

A TM is expressed as a 7-tuple $(Q, T, B, \Sigma, \delta, q_0, F)$ where:

- **Q** is a finite set of states
- **T** is the tape alphabet (symbols which can be written on Tape)
- **B** is blank symbol (every cell is filled with B except input alphabet initially)
- Σ is the input alphabet (symbols which are part of input alphabet)
- δ is a transition function which maps $Q \times T \rightarrow Q \times T \times \{L, R\}$.
- **q₀** is the initial state
- **F** is the set of final states. If any state of F is reached, input string is accepted.

Variants of TM:

1. Multiple track Turing Machine:

- A k -track Turing machine (for some $k > 0$) has k -tracks and one R/W head that reads and writes all of them one by one.
- A k -track Turing Machine can be simulated by a single track Turing machine

2. Two-way infinite Tape Turing Machine:

- Infinite tape of two-way infinite tape Turing machine is unbounded in both directions left and right.
- Two-way infinite tape Turing machine can be simulated by one-way infinite Turing machine (standard Turing machine).

3. Multi-tape Turing Machine:

- It has multiple tapes and is controlled by a single head.
- The Multi-tape Turing machine is different from k-track Turing machine but expressive power is the same.
- Multi-tape Turing machine can be simulated by single-tape Turing machine.

4. Multi-tape Multi-head Turing Machine:

- The multi-tape Turing machine has multiple tapes and multiple heads
- Each tape is controlled by a separate head
- Multi-Tape Multi-head Turing machine can be simulated by a standard Turing machine.

5. Multi-dimensional Tape Turing Machine:

- It has multi-dimensional tape where the head can move in any direction that is left, right, up or down.
- Multi dimensional tape Turing machine can be simulated by one-dimensional Turing machine

6. Multi-head Turing Machine:

- A multi-head Turing machine contains two or more heads to read the symbols on the same tape.
- In one step all the heads sense the scanned symbols and move or write independently.
- Multi-head Turing machine can be simulated by a single head Turing machine.

7. Non-deterministic Turing Machine:

- A non-deterministic Turing machine has a single, one-way infinite tape.
- For a given state and input symbol has at least one choice to move (finite number of choices for the next move), each choice has several choices of the path that it might follow for a given input string.
- A non-deterministic Turing machine is equivalent to the deterministic Turing machine.

Recursively enumerable language

- A recursively enumerable language is a formal language for which there exists a Turing machine (or other computable function) that will halt and accept when presented with any string in the language as input but may either halt and reject or loop forever when presented with a string not in the language.

Undecidable problems

- The problems for which we can't construct an algorithm that can answer the problem correctly in finite time are termed as Undecidable Problems.
- These problems may be partially decidable but they will never be decidable. That is there will always be a condition that will lead the Turing Machine into an infinite loop without providing an answer at all.

- We can understand Undecidable Problems intuitively by considering **Fermat's Theorem**, a popular Undecidable Problem which states that no three positive integers a , b and c for any $n > 2$ can ever satisfy the equation: $a^n + b^n = c^n$.

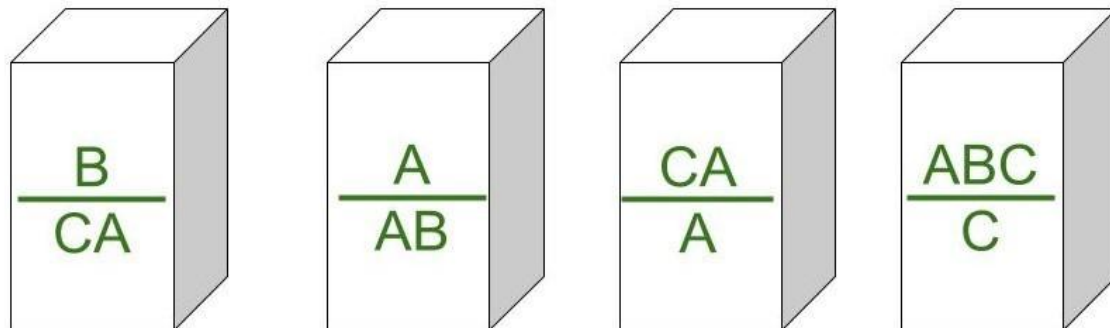
Post correspondence problem as an Undecidable Problem

- The post correspondence problem is an undecidable decision problem introduced by Emil post in 1946. The post correspondence problem consists of two strings, A and B, of equal input. The two lists of strings are $A = y_1, y_2, y_3, \dots, y_n$, $B = x_1, x_2, x_3, \dots, x_n$, then there exists a non-empty list of integers i_1, i_2, i_3, \dots Such that, $x_{i_1}, x_{i_2}, x_{i_3}, \dots, x_{i_n} = y_1, y_2, y_3, \dots, y_n$.

- There is **N** number of dominos in this problem, also known as tiles. The task is to arrange dominos so that strings made by the numerator and strings produced by the denominator are equal.

- Here in this problem, there is N number of dominos, i.e., tiles. And we have to arrange the dominos in such a way that the string produced by the denominators and the numerators is same.

1. Domino's Form :



2. Table form

	A	B
1	1	111
2	10111	10
3	10	0

Linear bounded automata

A Linear Bounded Automaton (LBA) is similar to Turing Machine with some properties stated below:

- Turing Machine with Non-deterministic logic,
- Turing Machine with Multi-track, and
- Turing Machine with a bounded finite length of the tape.

□ **Tuples Used in LBA :**

LBA can be defined with eight tuples (elements that help to design automata) as:

$M = (Q, T, E, q_0, M_L, M_R, S, F),$

Q -> A finite set of transition states

T -> Tape alphabet

E -> Input alphabet

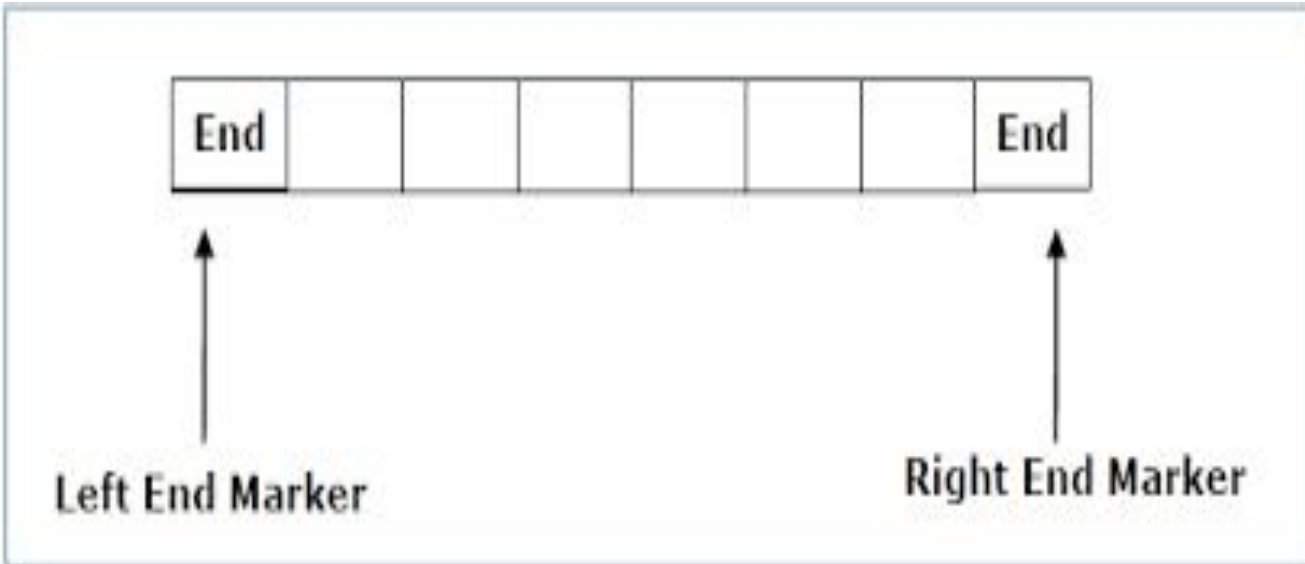
q_0 -> Initial state

M_L -> Left bound of tape

M_R -> Right bound of tape

S -> Transition Function

F -> A finite set of final states



Examples:

Languages that form LBA with tape as shown above,

- $L = \{a^{n!} \mid n \geq 0\}$
- $L = \{wn \mid w \text{ from } \{a, b\}^+, n \geq 1\}$
- $L = \{wwwR \mid w \text{ from } \{a, b\}^+\}$

Context sensitive languages

Context-Sensitive Grammar –

A Context-sensitive grammar is an Unrestricted grammar in which all the productions are of form –

$$\alpha \rightarrow \beta$$

where $\alpha, \beta \in (V \cup T)^+$ and $|\alpha| \leq |\beta|$

Context-sensitive Language: The language that can be defined by context-sensitive grammar is called CSL.

- Consider the following CSG.

$S \rightarrow abc/aAbc$

$Ab \rightarrow bA$

$Ac \rightarrow Bbcc$

$bB \rightarrow Bb$

$aB \rightarrow aa/aaA$

What is the language generated by this grammar?

- ***Solution:*** $S \rightarrow aAbc \rightarrow abAc \rightarrow abBbcc \rightarrow aBbbcc \rightarrow aaAbbcc$
 $\rightarrow aabAbcc \rightarrow aabbAcc \rightarrow aabbBbcc \rightarrow aabBbbcc$
 $\rightarrow aaBbbbcc \rightarrow aaabbbcc$

The language generated by this grammar is $\{a^n b^n c^n \mid n \geq 1\}$.