**EXPLORING AUTOMATA THEORY'S FUNDAMENTALS AND FRAMEWORK**

**Y. JAHNAVI**

**P. NAGENDRA KUMAR**

LEARNING IS A CONTINUOUS PROCESS

**EXPLORING AUTOMATA THEORY'S FUNDAMENTALS AND FRAMEWORK**

**Y. JAHNAVI**

Department of Computer Science

Dr. V. S. Krishna GDC (A), A.U. TDR\_HUB

Visakhapatnam,

Andhra Pradesh,

India.

**P. NAGENDRA KUMAR**

Department of Computer Science and Engineering

Geethanjali Institute of Science and Technology

Nellore,

Andhra Pradesh,

India

***Dedicated to***

***the divine lotus feet of***

***Sri Venkateswara***

**CONTENTS**

**Page No.**

**CHAPTER-I INTRODUCTION 1**

1.1 BASICS OF SET THEORY 1

1.2 RELATIONS 4

1.3 CENTRAL CONCEPTS OF AUTOMATA THEORY 10

1.3.1 ALPHABET 10

1.3.2 STRING 10

1.3.3 LANGUAGE 12

1.4 GRAMMAR 15

1.5 CHOMSKY HIERARCHY OF LANGUAGES 17

1.6 FINITE AUTOMATA 20

1.7 APPLICATIONS OF FINITE AUTOMATA 20

1.8 TYPES OF FINITE AUTOMATA 20

1.8.1 DETERMINISTIC FINITE AUTOMATA (DFA) 20

1.8.2 NONDETERMINISTIC FINITE AUTOMATA (NFA) 26

1.9 DIFFERENCE BETWEEN DFA AND NFA 28

1.10 CONVERSION OF NFA TO DFA 29

1.11 NFA WITH TRANSITIONS 31

1.12 FINITE AUTOMATA WITH OUTPUT 34

1.12.1 MOORE MACHINE 34

1.12.2 MEALY MACHINE 34

1.13 CONVERSION FROM MOORE TO MEALY MACHINE

AND VICE VERSA 37

1.14 MINIMIZATION OF FINITE AUTOMATA 42

**CHAPTER-II REGULAR EXPRESSION 45** 2.1 INTRODUCTION 45

2.1.1 REGULAR LANGUAGE 45

2.1.2 REGULAR EXPRESSION 45

2.1.3EXAMPLES OF REPRESENTING REGULAR

LANGUAGE INTO REGULAR EXPRESSION 45

2.2FINITE AUTOMATA AND REGULAR EXPRESSION

CONVERSIONS 46

2.3 PROBLEMS ON THE CONSTRUCTION OF FINITE AUTOMATA EQUIVALENT TO THE REGULAR EXPRESSION 48

2.4 IDENTITIES OF REGULAR EXPRESSION 49

2.5 PROVING THE EQUIVALENCE OF

THE REGULAR EXPRESSIONS USING

IDENTITIES OF REGULAR EXPRESSION 50

2.6 CONVERSION OF FINITE AUTOMATA TO REGULAR EXPRESSION USING ARDEN’S THEOREM 50

2.7 EXAMPLES ON THE CONVERSION OF FINITE AUTOMATA

TO REGULAR EXPRESSION 51

2.8 EQUIVALENCE OF TWO FINITE AUTOMATON53

2.9 EXAMPLES FOR PROVING THAT THE GIVEN

TWO FINITE AUTOMATON ARE EQUAL OR NOT 54

2.10 CONVERSION OF REGULAR EXPRESSION

TO FINITE AUTOMATA 55

2.11 EXAMPLES FOR THE CONVERSION OF

REGULAR EXPRESSION TO FINITE AUTOMATA 55

2.12 PUMPING LEMMA FOR REGULAR EXPRESSIONS 58

2.13 EXAMPLES ON PUMPING LEMMA 60

2.14 CONSTRUCTION OF FINITE AUTOMATA FROM

REGULAR GRAMMAR 61

2.15 CONSTRUCTION OF REGULAR GRAMMAR FROM

REGULAR EXPRESSION 63

2.16 CLOSURE PROPERTIES OF REGULAR EXPRESSIONS 65

2.17 APPLICATIONS OF REGULAR EXPRESSIONS AND

FINITE AUTOMATA 68

2.18 DECISION PROBLEMS OF REGULAR EXPRESSION 69

2.19 ALGEBRAIC LAWS FOR REGULAR EXPRESSIONS 70

**CHAPTER-III CONTEXT FREE GRAMMAR 72**

3.1 CONTEXT FREE GRAMMAR 72

3.2 EXAMPLES OF CFG 72

3.3 DERIVATION 74

3.3.1 LEFT MOST DERIVATION 74

3.3.2 RIGHT MOST DERIVATION 75

3.4 DERIVATION TREE 75

3.5 SIMPLIFICATION OF CFG 75

ELIMINATION OF

3.5.1 USELESS SYMBOLS 75

3.5.2 UNIT PRODUCTIONS 75

3.5.3 NULL PRODUCTIONS 77

3.6 NORMAL FORMS 75

3.6.1 CHOMSKY NORMAL FORM (CNF) 75

3.6.2 GREIBACH NORMAL FORM (GNF) 80

3.7 PROBLEMS OF TOP DOWN PARSER 81

3.7.1 AMBIGUITY 81

3.7.2 LEFT RECURSION 83

3.7.3 LEFT FACTORING 84

3.8 PUMPING LEMMA FOR CONTEXT FREE LANGUAGES 85

3.9 CLOSURE PROPERTIES OF

CONTEXT FREE LANGUAGES 88

3.10 DECISION PROBLEM RELATED TO

CONTEXT FREE LANGUAGES 95

**CHAPTER-IV PUSH DOWN AUTOMATA 98**

4.1 BASIC STRUCTURE OF PUSH DOWN AUTOMATA 98

4.2 FORMAL DEFINITION OF PDA 98

4.3 GRAPHICAL REPRESENTATION OF PDA 99

4.4 INSTANTANEOUS DESCRIPTION (ID) OF PDA 99

4.5 ACCEPTANCE OF PDA 99

4.6 TYPES OF PDA 100

4.6.1 DETERMINISTIC PDA 100

4.6.2 NON-DETERMINISTIC PDA 100

4.7 PROBLEMS ON PDA 100

4.8 EQUIVALENCE OF CFG AND PDA 105

4.8.1 CONVERT PDA TO CFG 105

4.8.2 CONVERT CGF TO PDA 106

4.9 TWO STACK PDA 107

4.10 PROBLEM ON THE CONSTRUCTION OF TWO STACK PDA

107

4.11 TYPES OF PDA 108

**CHAPTER-V TURING MACHINE 109**

5.1 INTRODUCTION TO TURING MACHINE 109

5.2 FORMAL DEFINITION OF TM 109

5.3 MECHANICAL DIAGRAM OF TM 109

5.4 INSTANTANEOUS DESCRIPTION (ID) OF TM 110

5.5 TRANSITION DIAGRAM FOR TM 110

5.6 CONVERSION OF REGULAR EXPRESSION TO

TURING MACHINE 112

5.7 TWO STACK PDA AND TURING MACHINE 114

5.8 VARIATIONS OF TURING MACHINES 116

5.9 LINEAR BOUNDED AUTOMATA (LBA) 118

5.10 TURING MACHINE AS AN INTEGER FUNCTION 120

5.11 UNIVERSAL TURING MACHINE 120

5.12 COUNTER MACHINE 120

5.13 DECIDABLE VS UNDECIDABLE 121

5.14 TYPES OF TM LANGUAGES 121

5.15 PROPERTIES OF RECURSIVE AND RECURSIVELY ENUMERABLE LANGUAGES 122

5.16 HALTING PROBLEM 125

5.17 POST CORRESPONDENCE PROBLEM (PCP) 126

5.18 MODIFIED PCP 128

**ABOUT THE AUTHORS**

****

**Dr. Y. Jahnavi** received her BTech (CSE) degree from JNTUH, MTech (CSE) from Sri Venkateswara University, and Ph.D. (CSE) from GITAM University. She qualified GATE, APSET and NET examinations. She is currently working at Department of Computer Science, Dr. V. S. Krishna GDC (A), Andhra University TDR\_HUB, Visakhapatnam. Her interest areas include Data Science, Machine Learning, Data Mining, Natural Language Processing techniques etc. She has been publishing papers in national and international journals. She has been invited as a keynote speaker and chaired various national and international conferences.

.



**P. Nagendra Kumar** received his MTech (CSE) from ANU and PhD (CS) degree from VSU. He qualified APSET examination. He is currently working at Department of Department of Computer Science and Engineering, Geethanjali Institute of Science and Technology, Nellore, Andhra Pradesh, India. His interest areas include Data Science, Machine Learning, Data Mining etc. He has been publishing papers in national and international journals.

**CHAPTER-I**

**INTRODUCTION**

* 1. **BASICS OF SET THEORY**

**Set**

* + **Definition:** Set is defined as collection of objects. These objects are called elements of set. Elements are enclosed within curly brackets i.e., ‘{‘ and ‘ } ’ and every element is separated by commas.
  + **Example:** If ‘a’ is an element of set A, then it is represented as a∈A.

If ‘a’ is not an element of set A, then it is represented as a∉A.

**Subset**

* **Definition:** Consider two sets A and B. If every element of set A is present in set B, then A is said to be subset of B. It is denoted as A⊆B.
* **Example:** A= {1, 2, 3}, B = {1, 2, 3, 4, 5}. Here, A is subset of B.

**Empty set**

* **Definition:** A set doesn’t contain any element is called an empty set or null set. It is represented by { } or ϕ.

**Finite set**

* **Definition:** A set is said to be finite if it has finite number of elements. Otherwise, it is an infinite set.

**A={a,b,c,d……..z}. The number of alphabets are 26.**

**N={1,2,3,4…………………..}: Infinite set**

**Equal set**

* **Definition:** If every element of set A is an element of set B and every element of set B is an element of set A then it is equal set i.e., A=B.

**A={1,2} B={1,2} then A=B**

**Power set**

* **Definition:** The power set of any set S is the set of all subsets of S, including the empty set and S itself. It is denoted as 2S.
* **If the set contains n elements, the powerset contains 2n elements.**
* **Example:** If *S* is the set {*x*, *y*, *z*}, then subsets of *S* are {ϕ, {*x*}, {*y*}, {*z*}, {*x*, *y*}, {*x*, *z*}, {*y*, *z*}, {*x*, *y*, *z*}}

A={a, b} powerset of A={{},{a},{b},{a,b}}

|A|=2 then power set of A contains 22 elements.

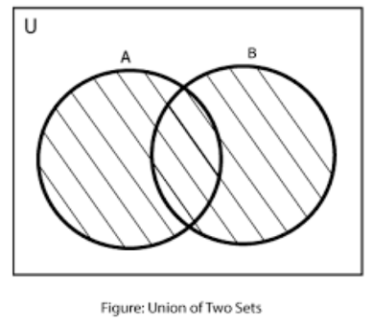
**ALGEBRAIC OPERATIONS ON SETS:**

The following are the operations that can be carried out on sets [1][2].

* Union
* Intersection
* Difference
* Complement

**Union**

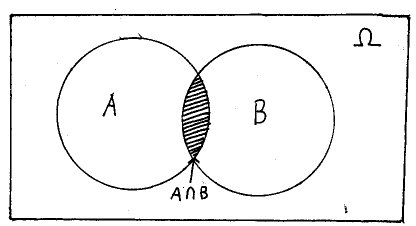
* **Definition:** In [set theory](https://en.wikipedia.org/wiki/Set_theory), the union (denoted by ∪) of a collection of sets is the set of all [elements](https://en.wikipedia.org/wiki/Element_(set_theory)) in the collection. It is one of the fundamental operations through which sets can be combined and related to each other.



* **Example:** if *A* = {1, 3, 5, 7} and *B* = {1, 2, 4, 6} then *A* ∪ *B* = {1, 2, 3, 4, 5, 6, 7}.

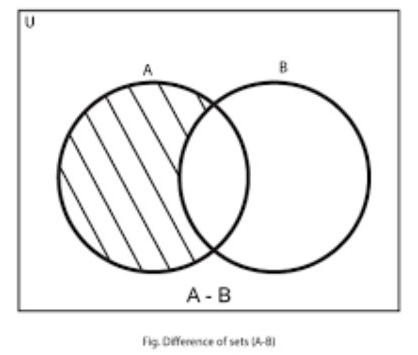
**Intersection**

* **Definition:{\displaystyle A\cup B=\{x:x\in A{\text{ or }}x\in B\}}Intersec** The intersection A ∩ *B* of two [sets](https://en.wikipedia.org/wiki/Set_(mathematics)) *A* and *B* is the set that contains all elements of *A* that also belong to *B*, but no other elements.
* **Example:** The intersection of the sets {1, 2, 3} and {2, 3, 4} is {2, 3}.



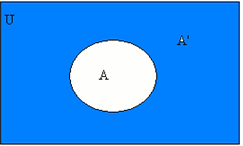
**Difference**

* **Definition:** The difference of two subsets A and B is a subset of U, denoted by A – B and is defined by A – B = {x : x ∈ A and x ∉ B}.
* **Example:**  If A = {2, 3, 4, 5, 6, 7} and B = {3, 5, 7, 9, 11}, then A – B={2, 4, 6}.

****

**Complement**

* **Definition:** In [set theory](https://en.wikipedia.org/wiki/Set_theory), the complement of a [set](https://en.wikipedia.org/wiki/Set_(mathematics)) *A* refers to [elements](https://en.wikipedia.org/wiki/Element_(mathematics)) not in *A.*
* **Example:** Let U={1,2,3,4,5,6} and A={1,3,5}. Then A' = {2,4,6}

****

**Cartesian product**

* **Definition:** In [set theory](https://en.wikipedia.org/wiki/Set_theory), a Cartesian product is a [mathematical operation](https://en.wikipedia.org/wiki/Mathematical_operation) that returns a [set](https://en.wikipedia.org/wiki/Set_(mathematics)) from multiple sets. That is, for sets *A* and *B*, the Cartesian product *A* × *B* is the set of all [ordered pairs](https://en.wikipedia.org/wiki/Ordered_pair) (*a*, *b*) where *a* ∈ *A* and *b* ∈ *B*.
* **Example:** *A* = {1,2}; *B* = {3,4}

*A* × *B* = {1,2} × {3,4} = {(1,3), (1,4), (2,3), (2,4)}

*B* × *A* = {3,4} × {1,2} = {(3,1), (3,2), (4,1), (4,2)}

*A* × *B* is not equal to *B* × *A*

**|** *A* *|=m*

**|** *B* *|=n*

**|***A* × *B|=mn*

* 1. **RELATIONS**
* **Definition:** Consider two sets A and B. A relation R from set A to set B is a subset of A × B. If (a, b) ∈R, then a is related to b i.e., a R b. If (a, b) ∉R, then a is not related to b.
* *A* = {1,2}; *B* = {3,4}
* *A* × *B* = {1,2} × {3,4} = {(1,3), (1,4), (2,3), (2,4)}
* **R1={}**
* **R2={**(1,3)**}**
* **R3={**(1,4)**}**
* **R4={**(2,3)**}**
* **R5={**(2,4)**}**
* **R6={**(1,3), (1,4)**}**
* **R7={**(1,3), (2,3) **}**
* **R8={**(1,3), (2,4)**}**
* **R9={**(1,4), (2,3)**}**
* **R10={**(1,4), (2,4)**}**
* **R11={**(2,3), (2,4)**}**
* **R12={**(1,3), (1,4), (2,3)**}**
* **R13={**(1,3),(1,4),(2,4)**}**
* **R14={**(1,3),(2,3),(2,4)**}**
* **R15={**(1,4),(2,3),(2,4)**}**
* **R16={**(1,3),(1,4),(2,3),(2,4)**}**

**|A|=m=2**

**|B|=n=2**

**|AxB|=mn=4**

**The number of relations=2mn relations=24 =16**

**TYPES OF RELATIONS**

**Reflexive**

**Definition:** A relation R on set A is reflexive if every element of ’A’ is related to itself. Let A be a set and R be the relation defined in it. R is said to be reflexive, if (a, a) ∈ R for all a ∈ A that is, every element of A is related to itself, in other words aRa for every a ∈ A. A relation R in a set A is not reflexive if there be at least one element a ∈ A such that (a, a) ∉ R.

* **Example:** Let A = {B, C, D}. A is a Reflexive relation if {B,B}, {C,C}, {D,D} are all present in A.

**Symmetric**

**Definition:** Let A be a set in which the relation R defined. Then R is said to be a symmetric relation, if (a, b) ∈ R ⇒ (b, a) ∈ R, that is, aRb ⇒ bRa for all (a, b) ∈ R.

**Example 1:** Consider for example, the set A of natural numbers. If relation A be defined by “x + y = 5”, then this relation is symmetric in A, for a + b = 5 ⇒ b + a = 5.

**Example 2:** Consider two parallel lines, a, b.

If a is parallel to b then b is also parallel to a.

If A is a cousin of B, then B is also cousin of A.

**Asymmetric**

**Definition:** Let A be a set in which the relation R defined. Then R is said to be an Asymmetric relation, if(a, b) ∈ R, then (b, a)  R.

* **Example:** If A is a father of B, then B is not father of A.

**Antisymmetric**

**Definition:** Let A be a set in which the relation R defined. Then R is said to be an Antisymmetric

relation, if(a, b) ∈ R, then (b, a)  R unless a=b

* **Example:** if a divides b, then b divides a, unless a=b

2 divides 4, then 4 does not divide 2, unless 2=4.

**Transitive**

**Definition:** Let A be a set in which the relation R defined. R is said to be transitive, if (a, b) ∈ R and (b, c) ∈ R ⇒ (a, c) ∈ R, That is aRb and bRc ⇒ aRc where a, b, c ∈ A.

* **Example 1:** if A is parallel to B and B is parallel to C, then A is parallel to C.
* **Example 2:** For example, in the set A of natural numbers, if the relation R be defined by ‘x less than y’ then a < b and b < c imply a < c, that is, aRb and bRc ⇒ aRc. Hence this relation is transitive.

**Equivalence Relation**

**Definition:** Equivalence relation on set is a relation which is reflexive, symmetric and transitive. A relation R, defined in a set A, is said to be an equivalence relation if and only if (i) R is **reflexive**, that is, aRa for all a ∈ A. (ii) R is **symmetric**, that is, aRb ⇒ bRa for all a, b ∈ A. (iii) R is **transitive**, that is aRb and bRc ⇒ aRc for all a, b, c ∈ A.

**Partial Order Relation**

Partial Order Relationon set is a relation which is reflexive, antisymmetric and transitive. A relation R, defined on a set A, is said to be a Partial Order Relation, if and only if (i) R is **reflexive**, that is, aRa for all a ∈ A. (ii) R is **antisymmetric**, that is, aRb ⇒ bRa only if a=b for all a, b ∈ A. (iii) R is **transitive**, that is aRb and bRc ⇒ aRc for all a, b, c ∈ A.

1. Reflexive, Irreflexive

*  a  A, (a, a) belongs to R
*  a  A, (a, a) should not belong to R

2. Symmetric, Asymmetric, Antisymmetric

* If (a, b) belongs to R then (b, a) should also belongs to R
* If (a, b) belongs to R then (b, a) should not belongs to R
* If (a, b) belongs to R then (b, a) should not belong to R, unless a=b

3. Transitive

* If (a, b) (b, c) belongs to R then (a, c) should belong to R.

4. Equivalence: Reflexive, Symmetric, Transitive

5. Partial order relation: Reflexive, Antisymmetric, Transitive

**Example:**

A = {1,2,3}

R = A x A

**Reflexive Relations:**

(a, a) belongs to R

R1= {(1,1), (2,2), (3,3)}

R2 = {(1,1), (2,2), (3,3), (1,2)}

**Not Reflexive Relation:**

R3= {(1,1), (1,2), (3,3)} because (2,2) does not exist

**Irreflexive Relation:**

(a, a) does not belong to R.

Eg:- A={1,2,3}

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

**Closure**

**Definition:** A set is closed if and only if the operation on two elements of the set produces another element of the set.

* **Example:** If we multiply two integer numbers, we will get another integer number. Here, integer numbers are closed under the operation of multiplication.

1x2=2

2x3=6

**Cardinality of sets**

**Definition:** Cardinality of a [set](https://en.wikipedia.org/wiki/Set_(mathematics)) is a measure of the number of [elements](https://en.wikipedia.org/wiki/Element_(mathematics)) of the set.

* **Example:** Set A = {2, 4, 6} contains 3 elements, and A has a cardinality of 3.
* **B={a,b,c,d} |B|=4**

**Introduction to formal proofs**

There are 2 types of formal proofs. They are: 1) Deductive proof and 2) Inductive proof.

**Deductive proof**

**Definition:** Deductive proof consists of sequence of statements whose truth leads us from some initial statement called the hypothesis or the given statement to a conclusion statement. Each step in the proof must follow Logical principle.

* **Example:** If A then B.

Here, A is called hypothesis

B is called conclusion statement

B is deduced from A

**Inductive proof**

**Definition:** It is used to prove recursively defined objects. This type of proof is called as proof by mathematical induction [5][6].

* The proof by mathematical induction can be carried out by following steps:
  + Basis: In this step, we assume lowest possible value. We prove that the result is true for n=1.
  + Induction Hypothesis: In this step, we will check whether the result is true for n=k or not.
  + Inductive step: In this step, if n=k is true, then check whether the result is true for n=k+1.

**Problems**

1. Prove 1+2+3+4……………+n = n(n+1)/2 by mathematical induction proof.

Sol:

**STEP 1:** We first show that p (1) is true.   
Left Side = 1   
Right Side = 1 (1 + 1) / 2 = 1   
Both sides of the statement are equal hence p (1) is true.

**STEP 2:** We now assume that p (k) is true   
1 + 2 + 3 + ... + k = k (k + 1) / 2   
**STEP 3:** show that p (k + 1) is true by adding k + 1 to both sides of the above statement   
1 + 2 + 3 + ... + k + (k + 1) = k (k + 1) / 2 + (k + 1)   
 = (k + 1)(k / 2 + 1)   
  
 = (k + 1)(k + 2) / 2

= (k + 1)(k + 1+1) / 2

∴ P(k + 1) is true.

1. = n(n + 1)(2n + 1)/6

Sol:

**Step 1:** We first show that p (1) is true.   
 Left Side =  =1  
 Right Side = 1 (1 + 1)(2+1) / 6 = 1   
 Both sides of the statement are equal hence p (1) is true.

**Step 2:** We now assume that p (k) is true

= K(K + 1)(2K + 1)/6

**Step 3:** we will prove that the statement must be true for *n* = *k* + 1:

12 + 22 + 32 + *· · ·* + *k*2 + (*k* + 1)2

= [k(k + 1)(2k + 1)]/ 6 +

=[ k(k + 1)(2k + 1) + 6 ]/6

= (k + 1)[k(2k + 1) + 6(k + 1)]/ 6

= [(k + 1)(2 *k*2 + 7k + 6)]/ 6

= (k + 1)(k + 2)(2k + 3)/ 6

∴ P(k + 1) is true.

Thus, the left-hand side is equal to the right-hand side. This proves the inductive step. Therefore, by the principle of mathematical induction, the given statement is true for every positive integer *n*.

**Additional forms of proof**

* Proof about sets
* Proof by contradiction
* Proof by counter example

**Proof about sets**

* **Definition:** The set is a collection of elements. By giving proofs about the sets, we try to prove certain properties of the sets.
* **Example:** Consider two expressions A and B. We want to prove that both expressions A and B are equivalent**.** Let PUQ = QUP

A = PUQ

B = QUP

We need to prove A = B

* This proof is of kind “if and only if” that means an element x is in A if and only if it is in B.

|  |  |
| --- | --- |
| **PUQ** | **QUP** |
| x is in PUQ | x is in QUP |
| x is in P or x is in Q | x is in Q or x is in P |
| x is in Q or x is in P | x is in P or x is in Q |
| x is in QUP | x is in PUQ |

Hence, PUQ=QUP. Thus, A=B is true as element x is in B if and only if x is in A.

**Proof by contradiction**

In the statement of the form if A then B, we start with Statement A is not true and thus by assuming false A and try to get the conclusion of statement B. When it becomes impossible to reach statement B we contradict our self and accept that A is true.

**Proof by counter example**

**Definition:** In order to prove certain statements, we need to see all possible conditions in which that statement remains true. There are some situations, statement cannot be true.

**Example:** a mod b = b mod a

If a=2 and b=3 then, a mod b ≠ b mod a

If a=2 and b=2 then, a mod b = b mod a

**1.3. CENTRAL CONCEPTS OF AUTOMATA THEORY**

There are various central concepts of Automata Theory such as Alphabet, String and Language.

### ALPHABET

* **Definition**: An alphabet is any nonempty finite set of symbols.
* **Eg**: Binary Alphabet set ∑ ={0,1}
* **Eg**: English Alphabet set ∑ ={a,b,c,d…………….z}
* Eg: ∑={a,b}
* Eg: ∑={a,e,i,o,u}
* **Example:** ∑ = {a, b, c, d} is an alphabet set where ‘a’, ‘b’, ‘c’, and ‘d’ are symbols.
* **Sets are denoted within { }**

### STRING

* **Definition**: A string is a finite sequence of symbols taken from ∑.
* ∑ ={0,1} The strings are={ε ,0,1,00,01,10,11,000,001,010,011,100……….}
* S0= 0110 |S0|=4
* S1=01 |S1|=2
* **Example**: ‘cabcad’ is a valid string on the alphabet set ∑ = {a, b, c, d}
* ∑ ={a, b} The strings are={ ε, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, baa, bab, bba, bbb…….}

### Length of a String

* **Definition:** It is the number of symbols present in a string. (Denoted by |S|).
* **Example:** If S = ‘cabcad’, |S|= 6

**Kleene Closure / Star**

* **Definition:** The Kleene star, ∑\*, is a unary operator on a set of symbols or strings, ∑, that gives the infinite set of all possible strings of all possible lengths over ∑ including ε.
* Representation − ∑\* = ∑0 ∪ ∑1 ∪ ∑2 ∪……. where ∑p is the set of all possible strings of length p.
* **Example**: If ∑ = {a, b}, ∑\* = { ε, a, b, aa, ab, ba, bb,………..}
* ∑\* = ∑0 ∪ ∑1 ∪ ∑2 ∪…….
* ∑0=  { ε }
* ∑1= { a, b }
* ∑2 = { aa, ab, ba, bb}
* .  
  .
* .
* ∑\* ={ ε, a, b, aa, ab, ba, bb,………………}

### Positive Closure / Plus

* **Definition:** The set ∑+ is the infinite set of all possible strings of all possible lengths over ∑ excluding ε.
* Representation − ∑+ = ∑1 ∪ ∑2 ∪ ∑3 ∪…….

∑+ = ∑\* − { ε }

* **Example:** If ∑ = { a, b } , ∑+ = { a, b, aa, ab, ba, bb,………..}

**Prefix of a string**

* **Definition:** It is the string formed by taking any number of leading symbols of the string.
* **Example 1:** String w= 0111

Prefix of a string for 0111🡪 ε, 0, 01,011 0111

**Example 2:** String w=1010

Prefixes: ε, 1, 10, 101, 1010

**Substring**

* **Definition:** It is the string formed by taking any number of symbols of the string.
* **Example:** String w= 0111

substring of a string for 0111🡪 ε, 0, 01,011 0111,11,111

**Proper Prefix of a string**

* **Definition:** Any Prefix of the string other than the string itself.
* **Example:** String w= 0111

Proper Prefix of a string for 0111🡪 0, 01,011

**Suffix of a string**

* **Definition:** It is the string formed by taking any number of trailing symbols from the end of the string.
* **Example:** String w= 0111

Suffix of a string for 0111🡪 ε, 1, 11, 111, 0111

**Proper Suffix of a string**

* **Definition:** Any Suffix of the string other than the string itself.
* **Example:** String w= 0110

Suffix of a string for 0110🡪 0, 10,110.

### LANGUAGE

* **Definition:** Language is defined as the set of strings. A language is a subset of ∑\* for some alphabet ∑. It can be finite or infinite.
* **Example:** If the language takes all possible strings of length 2 over ∑ = {a, b}, then L = {aa, ab, ba, bb}

Eg 1:- ∑ = {0, 1}

S1= 0, s2=10, s3=110, s4=1010, s5=1011

L={0, 10, 110, 1010, 1011,…………………………….}

∑={a,b}

S1=aa, S2=ab, S3=aba

L={aa, ab, aba,………………………………………}

∑={a, e, i, o, u}

S1=aeo S2=iou S3= ieu, S4=aiu

L={aeo, iou, ieu, aiu…………………………….}

∑={a,b}

L=set of all strings of length 2

S1= aa S2=ab S3= ba S4= bb

L={aa, ab, ba, bb}

Language can be either finite or infinite.

Language is a set of strings taken from a fixed alphabet set.

* 1. Alphabet = set of nonempty and finite symbols

Example: ∑ = {0,1} ∑={a,b} ∑= {a,b,c,d, …………………..z}

* 1. Strings = Sequence of symbols taken from fixed Alphabet

Example: ∑ = {0,1}

S0= ε

S1=0 S2=1

S3=00, S4=01, S5=10, S6=11

S7=000, S8=001, S9=010, S10=011, ………………………

* 1. Language = Set of strings over fixed alphabet ∑

Example 1:

∑ = {0,1}

L1 = set of strings of length 1

L1 = {0,1}

L2 = set of strings of length 3

L2 = {000,001,010,011,100, 101, 110,111}

L3 = set of strings over ∑={0,1}

L3 = {ε , 0, 1, 10, 11, 110,……………………………… }

L\*=L0 U L1 …………………….

L+= L1 UL2U……………..

String1=00011 String2=1010 over ∑ = {0, 1}

L = {00011, 1010, ………………} over ∑ = {0, 1}

Example 2: ∑ ={a, b}

String 1 = aabba String2 = babb over ∑ ={a,b}

L = {aabba, babb,……………..}over ∑ ={a,b}

Example 3: ∑ ={a,b,c…………………….z}

String1= apple String=banana

L = {apple, banana, …………………….}

* **Operations on Languages:**
* Union
* Concatenation
* Intersection
* Difference
* Kleene closure
* Positive closure

Example 1: L1 = {00, 11}

L2 = {01}

L1UL2 = {00,11,01}

L1. L2 = {0001,1101}

L1-L2 = {00, 11}

Example 2 : L1 = {01}

L2 = {11, 10}

L1 U L2 = {01, 11, 10}

L1. L2 = {0111, 0110}

L1-L2 = {01}

Example 3**:** L={0}

L\* = {ε, 0, 00, 000, 0000……………………….}

L+ = {0, 00, 000, 0000,…………………………}

**1.4. GRAMMAR**

**Definition:** Grammar is a set of rules in any language. It consists of 4 tuples G = {V, T, P, S}. V is set of non-terminals, T is set of terminals, P is production rules and S is start state.

* **V:** set of nonterminals or Variables denoted by capital letters.
* **T:** set of terminals denoted by small letters.
* **P:** Production rules
* **S:** Start State
* **Example 1:** A🡪aA / . Here, a,  are terminals and A is non-terminal.

V = {A}

T = {a, }

P = {A->aA, A🡪}

S = {A}

**Example 2:** A->aB

B->b

V = {A, B}

T = {a, b}

P = {A->aB

B->b}

S={A}

**PROBLEMS:**

1. Construct the Language generated from the given grammar S🡪aS/ε.

<V,T,P,S>

V={S}

T={a, ε}

P={ S🡪aS; S-> ε}

S={S}

Sol: S🡪aS

S🡪aaS (Replace S by S🡪 aS)

S->aaaS(Replace S by S🡪 aS)

S->aaaaS(Replace S by S🡪 aS)

S🡪aaaa (Replace S by S🡪 ε)

If we repeat this process by replacing with S with S🡪aS and ε, we get steps.

So, L(G) = , n>=0.

1. Construct the Language generated from the grammar S🡪 aSb/ ε.

Sol: S🡪aSb

S🡪aaSbb

S🡪aaaSbbb

S🡪aaabbb

If we repeat this process, we get equal number of a’s followed by equal number of b’s.

So, L(G) =

1. Construct the language generated by the grammar S🡪aCa, C🡪aCa/b.

Sol: V={S, C}

T={a,b}

P={S->aCa

C->aCa

C->b}

S={S}

S🡪aCa

S🡪aaCaa

S🡪aaaCaaa

S🡪aaabaaa

If we repeat this process, the language generated by grammar is L(G)=

1. Construct the grammar for palindrome of binary numbers

Sol:

∑ = {0,1}

L ={0,1,00,11,010,101,000,111…………}

G ={V,T,P,S}

V={S}

T={0,1}

P:

S🡪0S0/1S1/1/0/ ε

S is start state

If S->0S0->00S00->000S000

S->1S1->10S01->101S101->101101

{00,010,000,000000,0001000,0000000,101101}

1. Construct Grammar for the Language

Sol: V={S}

T={a,b}

P: S🡪aSb/abb

S is start state

{abb, aabbb, aaabbbb, ………………}

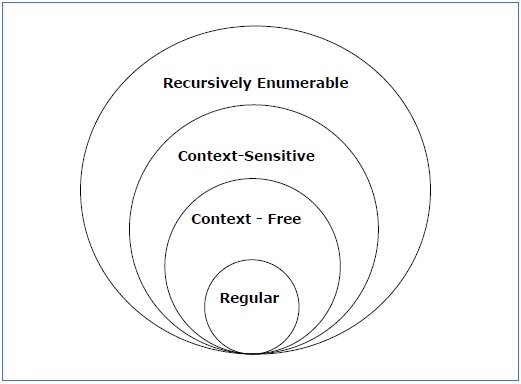
There exists other algorithms for processing natural languages [3][4][8][10-17][19][21][23-25]**.**

**1.5 CHOMSKY HIERARCHY OF LANGUAGES**

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

|  |  |  |  |
| --- | --- | --- | --- |
| **Grammar Type** | **GRAMMAR** | **LANGUAGE** | **AUTOMATON** |
| Type 0 | Unrestricted Grammar | Recursively enumerable Language | Turing Machine |
| Type 1 | Context Sensitive Grammar | Context Sensitive Language | Linear Bounded Automaton |
| Type 2 | Context-free Grammar | Context Free Language | Pushdown Automaton |
| Type 3 | Regular Grammar | Regular Language | Finite State Automaton |

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



**<V, T, P, S>**

**V: Variables or Nonterminals denoted by capital letters**

**T: Terminals denoted by small letters**

**P: Production Rules**

**LHS->RHS**

**S: Start Symbol**

**TYPE 0 GRAMMAR:**

Type-0 grammars generate recursively enumerable languages. The productions have no restrictions. They are any phase structure grammar including all formal grammars. They generate the languages that are recognized by a Turing machine.

The productions can be in the form of LHS→ RHS where LHS is a string of terminals and non-terminals with at least one non-terminal and LHS cannot be null. RHS is a string of terminals and non-terminals.

**Example:** Sab –> ba

* **LHS->RHS**
* LHS cannot be null.
* There are no restrictions on LHS and RHS.
* Unrestricted Grammar, recursively enumerable grammar.

**Type 1 GRAMMAR:** Type-1 grammars generate the context-sensitive languages. The language generated by the grammar is recognized by the [Linear Bound Automata](https://en.wikipedia.org/wiki/Linear_bounded_automaton). Type-1 grammars have the format of LHS->RHS; length of LHS is less than or equal to length of RHS.

**Example:** S🡪AB

AB🡪abc

B🡪b

* LHS->RHS
* |LHS|<=|RHS|

**Type 2 GRAMMAR:** Type-2 grammars generate the context-free languages. The language generated by the grammar is recognized by the [Push Down Automata](https://en.wikipedia.org/wiki/Linear_bounded_automaton).Type-2 grammars have the format of LHS->RHS; length of LHS is less than or equal to length of RHS; LHS should be a single Variable.

**Example:** S🡪 AB

A🡪a

B🡪b

* LHS->RHS
* |LHS|<=|RHS|
* LHS is a single variable.

**Type 3 GRAMMAR:** Type-3 grammars generate regular languages. The language generated by the grammar is recognized by the Finite Automata. Type-3 grammars have the format of LHS->RHS; length of LHS is less than or equal to length of RHS; Type-3 grammars must have a single non-terminal on the left-hand side and a right-hand side consisting of a single terminal or single terminal followed by a single non-terminal or a single non terminal followed by a single terminal.

**Example 1:** A🡪aA

A🡪a

**Example** **2:** A->Aa

A->a

* LHS->RHS
* |LHS|<=|RHS|
* LHS is a single variable.
* RHS is in the form of terminal NT / NT terminal / terminal

**1.6 FINITE AUTOMATA**

A Finite Automaton is a mathematical model of a system with discrete inputs and outputs. Finite means number of possible states and number of symbols in alphabet are both finite and automaton is a machine.

Formal definition of a Finite Automaton

Finite Automata can be represented by a 5-tuple (Q, ∑, δ, q0, F), where −

* Q is a finite set of states.
* ∑ is a finite set of symbols, called the alphabet of the automaton.
* δ is the transition function.
* q0 is the initial state
* F is a set of final state/states of Q

**1.7 APPLICATIONS OF FINITE AUTOMATA**

* Designing and checking the behaviour of digital circuits using software.
* Compiler design: During compilation, Lexical analysers breaks the input text into various lexical units such as identifiers, keywords and punctuations.
* Designing software for scanning large bodies of text such as collection of web pages
* Designing software for verifying systems having finite number of states
* Software for natural language processing

**1.8 TYPES OF FINITE AUTOMATA**

Finite Automata can be classified into two types −

* Deterministic Finite Automata (DFA)
* Non-deterministic Finite Automata (NFA)

**1.8.1 DETERMINISTIC FINITE AUTOMATA**

Deterministic Finite Automaton is a FA in which there is only one path for a specific input from current state to next state.

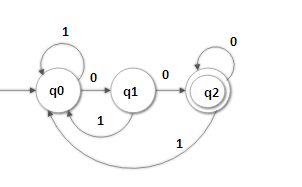
Tuples: (Q, ∑, 𝛿, q0, F)

* Q is a finite set of states.
* ∑ is a finite set of input alphabets
* δ is the transition function. 𝛿: Q \* ∑🡪 Q
* q0 is the initial state
* F is a set of final state/states of Q

**EXAMPLES:**

1) Design a DFA for the language that contains the strings end with 00 over Σ = {0,1}

**Sol:** L = {00, 000, 100, 1100……………..}

****

M = (Q, Σ, , qo, F)

Q = {q0,q1,q2}

Σ = {0,1}

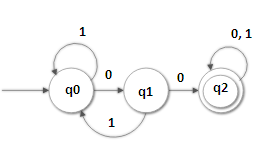
q0🡪start state

F ={q2}

|  |  |  |
| --- | --- | --- |
|  | 0 | 1 |
| q0 | q1 | q0 |
| q1 | q2 | q0 |
| q2 | q2 | q0 |

2) Design DFA for the language containing the strings having the substring 00 over Σ = {0,1}

**Sol:** L = {00, 100, 1001, 1100, 0000, 0001, …………………….}



M = (Q, Σ, , qo, F)

Q = {q0, q1, q2}

Σ = {0,1}

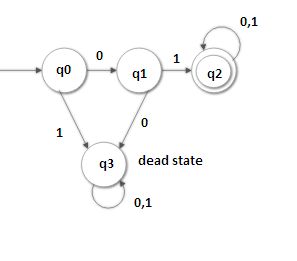
q0🡪start state

F ={q2}

|  |  |  |
| --- | --- | --- |
|  | 0 | 1 |
| q0 | q1 | q0 |
| q1 | q2 | q0 |
| q2 | q2 | q2 |

3) Design DFA for the language containing the strings starting with 01 over Σ = {0,1}. Check whether the strings 01101 and 00011 are acceptable or not.

Sol:



M = (Q, Σ, , qo, F)

Q = {q0,q1,q2, q3}

Σ = {0,1}

q0🡪start state

F ={q2}

|  |  |  |
| --- | --- | --- |
|  | 0 | 1 |
| q0 | q1 | q3 |
| q1 | q3 | q2 |
| q2 | q2 | q2 |
| q3 | q3 | q3 |

(q0, 01101) = ((q0,0),1101) = (q1,1101) = ((q1,1),101) = (q2,101) = ((q2,1),01) = (q2,01) = ((q2,0),1) =(q2,1) = q2

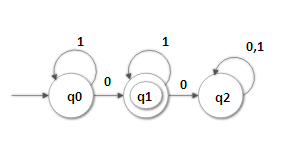
q2 is a Final State. Therefore the input string 01101 is accepted by the given Finite Automata.

(q0,00011) = 0011)=q1,0011)=q3,011)=

q3 is not a final state. Therefore the input string 00011 is not accepted by the given Finite Automata.

4) Design DFA to accept string of 0’s and 1’s having exactly one ‘0’ over Σ = {0,1}.

Sol:



M = (Q, Σ, , qo, F)

Q = {q0,q1,q2}

Σ = {0,1}

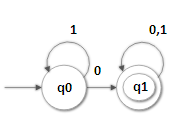
q0🡪start state

F ={q1}

|  |  |  |
| --- | --- | --- |
|  | 0 | 1 |
| q0 | q1 | q0 |
| q1 | q2 | q1 |
| q2 | q2 | q2 |

5) Design DFA to accept strings of 0’s and 1’s having at least one ‘0’.

Sol:



6) Design DFA to accept a) even no. of 0’s and even no. of 1’s

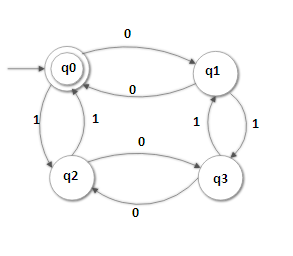
b) odd no. of 0’s and even no. of 1’s

c) even no. of 0’s and odd no. of 1’s

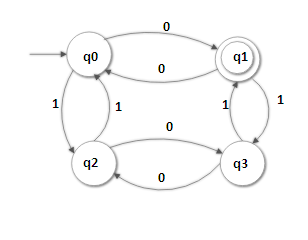
d) odd no. of 0’s and odd no. of 1’s

Sol:

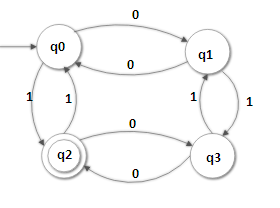
a) Even no. of 0’s and even no. of 1’s



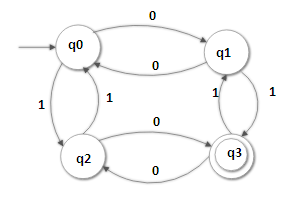
b) odd no. of 0’s and even no. of 1’s



c) even no. of 0’s and odd no. of 1’s



d) odd no. of 0’s and odd no. of 1’s



7) Check whether the string 110101 is accepted by the following FSM.

|  |  |  |
| --- | --- | --- |
|  | 0 | 1 |
| q0 | q2 | q1 |
| q1 | q3 | q0 |
| q2 | q0 | q3 |
| q3 | q1 | q2 |

Sol:

(q0,110101) = (q0,1), 10101) = 10101) = (q1,1),0101) = q0,0101) = q0,0),101) = q2,101) = q2,1),01) = q3, 01) = ((q3,0),1) = q1,1) = q0

Hence q0 is a nonfinal state, the given input is not accepted by the given FSM.

**1.8.2 NON-DETERMINISTIC FINITE AUTOMATA**

NFA or Non-Deterministic Finite Automaton is the one in which there exists many paths for a specific input from current state to next state.

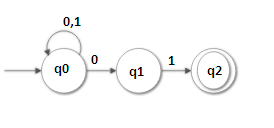
Tuples: (Q, ∑, 𝛿, q0, F)

* Q is a finite set of states.
* ∑ is a finite set of input alphabets
* δ is the transition function. 𝛿: Q \* ∑ 🡪
* q0 is the initial state
* F is a set of final state/states of Q

**EXAMPLES:**

1) Design NFA accepting all strings ending with 01 over ∑ = {0,1}.

Sol:



M = {Q, ∑, δ, q0, F}

Q = {q0, q1, q2}

∑ = {0,1}

δ (q0, 0) = {q0, q1}

δ (q0, 1) = q0

δ (q1, 1) = q2

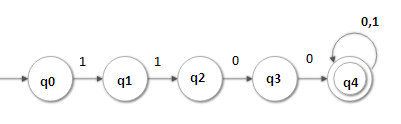
|  |  |  |
| --- | --- | --- |
|  | 0 | 1 |
| q0 | q0, q1 | q0 |
| q1 | - | q2 |
| q2 | - | - |

q0 = {q0}

F = {q2}

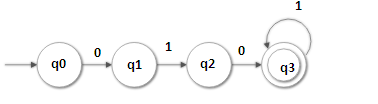
2) Construct NFA in which double ‘1’ is followed by double ‘0’ over ∑ = {0,1}.

Sol:



3) Construct NFA for the language L = {0101n /n>=0} over ∑ = {0,1}.

Sol:

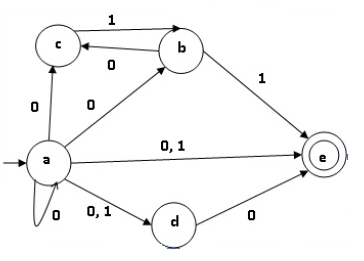


**1.9 DIFFERENCES BETWEEN DFA AND NFA:**

|  |  |
| --- | --- |
| **DFA** | **NFA** |
| * Deterministic Finite Automaton is a FA in which there is only one transition for a specific input from current state to next state.   Tuples: (Q, ∑, 𝛿, q0, F)  𝛿: Q \* ∑🡪 Q | * NFA or Non-Deterministic Finite Automaton is the one in which there exists zero, one or many transitions for a specific input from current state to next state.   Tuples: (Q, ∑, 𝛿, q0, F)  𝛿: Q \* ∑ 🡪 |
| * DFA is difficult to construct. There is a unique transition on each input symbol. | * NFA can be used in theory of computation because they are more flexible and easier to use than DFA. |
| * Backtracking is allowed in DFA | * In NFA, backtracking is not always possible. |
| * Requires more space. | * Requires less space. |

**1.10 CONVERSION OF NFA TO DFA**

**1)** Let us consider the NFA shown in the figure below.



|  |  |  |
| --- | --- | --- |
|  | **δ(q,0)** | **δ(q,1)** |
| a | {a,b,c,d,e} | {d,e} |
| b | {c} | {e} |
| c | ∅ | {b} |
| d | {e} | ∅ |
| e | ∅ | ∅ |

Using the above algorithm, we find its equivalent DFA. The state table of the DFA is shown below.

STEP 1: Start with Start State

STEP 2: Find the transition from Start State

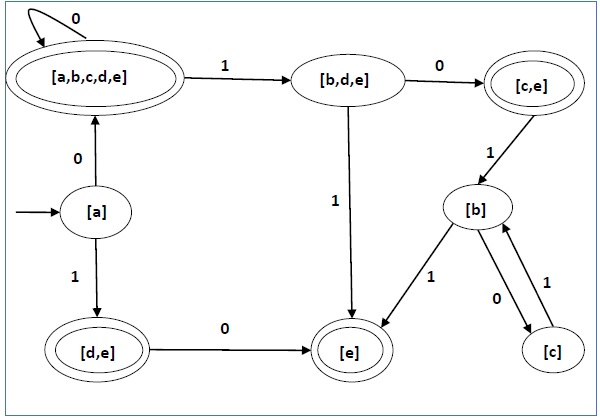
STEP 3: Find the transition for all the new states that have derived from the start state/previous states.

STEP 4: Proceed finding transition until there exists no new states.

STEP 5: Start state of DFA is the start state of NFA. Final state of DFA is the states which contain the final state of NFA.

|  |  |  |
| --- | --- | --- |
| **Δ** | **δ(q,0)** | **δ(q,1)** |
| [a] | [a, b, c, d, e] | [d, e] |
| [a, b, c, d, e] | [a, b, c, d, e] | [b, d, e] |
| [d, e] | [e] | ∅ |
| [b, d, e] | [c, e] | [e] |
| [e] | ∅ | ∅ |
| [c, e] | ∅ | [b] |
| [b] | [c] | [e] |
| [c] | ∅ | [b] |

The state diagram of the DFA is as follows:



**1.11 NFA WITH**  **TRANSITIONS**

Nondeterministic finite automaton with ε-moves (NFA-ε) is a further generalization to NFA. This automaton replaces the transition function with the one that allows the [empty string](https://en.wikipedia.org/wiki/Empty_string) ε as a possible input. The transitions without consuming an input symbol are called ε-transitions [7].

**Formal definition of NFA-ε**

An NFA-ε is represented formally by a [5-tuple](https://en.wikipedia.org/wiki/N-tuple), (Q, [Σ](https://en.wikipedia.org/wiki/Sigma), 𝛿, q0, F), consisting of

* Q 🡪finite [set](https://en.wikipedia.org/wiki/Set_(mathematics)) of [states](https://en.wikipedia.org/wiki/State_(computer_science))
* [Σ](https://en.wikipedia.org/wiki/Sigma)🡪finite set of input alphabets
* 𝛿 🡪transition [function](https://en.wikipedia.org/wiki/Function_(mathematics))  : Q × (Σ ∪ {ε}) →
* q0 🡪[start](https://en.wikipedia.org/wiki/Finite_state_machine#Start_state) state
* F🡪final states

**PROCEDURE:**

**STEP 1:** Find the ε- Closure of each state. These are the set of states that reach from a particular state only with ε- transition.

**STEP 2:** Find the new transitions for all the states.

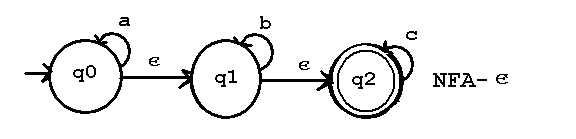
q, a) = ε- Closure (𝛿 (ε- Closure (q), a))

**STEP 3:** Start state of NFA without ε- transitions is the start state of NFA with ε- transitions.

**STEP 4:** Final states of NFA without ε- transitions is all the states whose ε- Closure contains the final states of the given NFA with ε- transitions.

**EXAMPLES:**

1) Convert the following ε-NFA to its equivalent DFA



Sol: ε- Closure (q0) = {q0, q1, q2}

ε- Closure (q1) = {q1, q2}

ε- Closure (q2) = {q2}

**For state q0**

q0, a) = ε- Closure (𝛿 (ε- Closure (q0), a))

= ε- Closure (𝛿 (q0, q1, q2), a)

= ε- Closure (𝛿 (q0, a) ∪ 𝛿 (q1, a) ∪ 𝛿 (q2, a))

= ε- Closure (q0)

= {q0, q1, q2}

q0, b) = ε- Closure (𝛿 (ε- Closure (q0), b))

= ε- Closure (𝛿 (q0, q1, q2), b)

= ε- Closure (𝛿 (q0, b) ∪ 𝛿 (q1, b) ∪ 𝛿 (q2, b))

= ε- Closure (q1)

= {q1, q2}

q0, c) = ε- Closure (𝛿 (ε- Closure (q0), c))

= ε- Closure (𝛿 (q0, q1, q2), c)

= ε- Closure (𝛿 (q0, c) ∪ 𝛿 (q1, c) ∪ 𝛿 (q2, c))

= ε- Closure (q2)

= {q2}

**For state q1**

q1, a) = ε- Closure (𝛿 (ε- Closure (q1), a))

= ε- Closure (𝛿 (q1, q2), a)

= ε- Closure (𝛿 (q1, a) ∪ 𝛿 (q2, a))

= ε- Closure (ᴓ)

= ᴓ

q1, b) = ε- Closure (𝛿 (ε- Closure (q1), b))

= ε- Closure (𝛿 (q1, q2), b)

= ε- Closure (𝛿 (q1, b) ∪ 𝛿 (q2, b))

= ε- Closure (q1)

= {q1, q2}

q1, c) = ε- Closure (𝛿 (ε- Closure (q1), c))

= ε- Closure (𝛿 (q1, q2), c)

= ε- Closure (𝛿 (q1, c) ∪ 𝛿 (q2, c))

= ε- Closure (q2)

= {q2}

**For state q2**

q2, a) = ε- Closure (𝛿 (ε- Closure (q2), a))

= ε- Closure (𝛿 (q2), a)

= ᴓ

q2, b) = ε- Closure (𝛿 (ε- Closure (q2), b))

= ε- Closure (𝛿 (q2), b)

= ε- Closure (ᴓ)

= ᴓ

q2, c) = ε- Closure (𝛿 (ε- Closure (q2), c))

= ε- Closure (𝛿 (q2), c)

= ε- Closure (q2)

= {q2}

Consider A = {q0, q1, q2}

B = {q1, q2}

C= {q2}

**Transition table:**

|  |  |  |  |
| --- | --- | --- | --- |
| **𝛿** | **A** | **B** | **C** |
| A | A | B | C |
| B | ᴓ | B | B |
| C | ᴓ | ᴓ | C |

**1.12 FINITE AUTOMATA WITH OUTPUT**

There are two types of finite automata with output.

They are

* 1. Moore Machine
  2. Mealy Machine.

**1.12.1 MOORE MACHINE**

Moore machine is a FSM whose outputs depend on only the present state. A Moore machine can be described by a 6 tuple (Q, ∑, δ, q0, λ, Δ) where −

* Q is a finite set of states.
* ∑ is a finite set of symbols called the input alphabet.
* Δ is a finite set of symbols called the output alphabet.
* δ is the input transition function where δ: Q × ∑ → Q
* λ is the output transition function where Q → Δ
* q0 is the initial state.

**1.12.2 MEALY MACHINE**

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

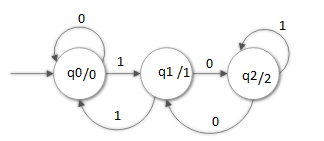
A Mealy machine can be described by a 6 tuple (Q, ∑, δ, q0, λ, Δ) where −

* Q is a finite set of states.
* ∑ is a finite set of symbols called the input alphabet.
* Δ is a finite set of symbols called the output alphabet.
* δ is the input transition function where δ: Q × ∑ → Q
* λ is the output transition function where Q x ∑→ Δ
* q0 is the initial state.

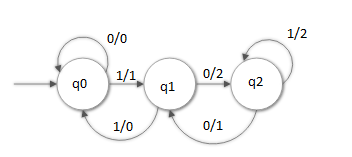
**EXAMPLES:**

1. Design Moore and Mealy machine which determines the residue mod 3 for each binary string

Sol: Moore machine

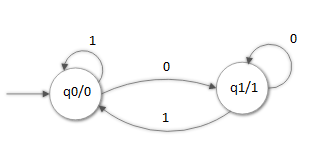


Mealy machine

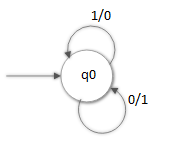


1. Design Moore and Mealy machine for 1's complement of a binary number.

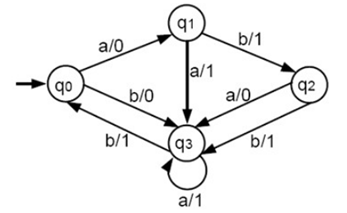
Sol: Moore machine



Mealy machine



1. Consider the following mealy machine and find the output for the input string aaabb



Sol:Given the input string aaabb, the output string is 01110

**1.13 CONVERSION FROM MOORE TO MEALY MACHINE AND MEALY TO MOORE MACHINE**

* + 1. Convert the following Moore Machine to Mealy Machine

|  |  |  |  |
| --- | --- | --- | --- |
| Present state | Input 0 | Input 1 | Output |
| q0 | q1 | q2 | 1 |
| q1 | q3 | q2 | 0 |
| q2 | q2 | q1 | 1 |
| q3 | q0 | q3 | 1 |

Sol:

Q = {q0, q1, q2, q3}

∑ = {0, 1}

Δ = {0, 1}

**For state q0**

(q0,0) = λ (δ(q0,0)) = λ(q1) =0

(q0,1) = λ (δ(q0,1)) = λ(q2) =1

**For state q1**

(q1,0) = λ (δ(q1,0)) = λ(q3) =1

(q1,1) = λ (δ(q1,1)) = λ(q2) =1

**For state q2**

(q2,0) = λ (δ(q2,0)) = λ(q2) =1

(q2,1) = λ (δ(q2,1)) = λ(q1) =0

**For state q3**

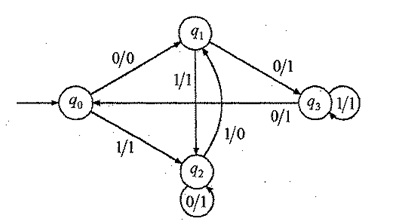
(q3,0) = λ (δ(q3,0)) = λ(q0) =1

(q3,1) = λ (δ(q3,1)) = λ(q3) =1

**Transition table:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Δ | Input 0 | Output | Input 1 | Output |
| q0 | q1 | 0 | q2 | 1 |
| q1 | q3 | 1 | q2 | 1 |
| q2 | q2 | 1 | q1 | 0 |
| q3 | q0 | 1 | q3 | 1 |

**Transition diagram:**

****

* + 1. Convert the following Mealy Machine to Moore Machine

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Δ | Input 0 | Output | Input 1 | Output |
| q0 | q3 | 0 | q1 | 1 |
| q1 | q0 | 1 | q3 | 0 |
| q2 | q2 | 1 | q2 | 0 |
| q3 | q1 | 0 | q0 | 1 |

Sol:

**Step 1:** If the output of the state in both λ1 and λ2 are same, then the state remains same. For example, in the state q0, the output of λ1 and λ2 is 0. So, there is no need to change q0.

**Step 2:** If the output of the state in both λ1 and λ2 are not same, then split the state into two states. For example, in the state q2, the output of λ1 and λ2 is 1 and 0. So, split q2 as q20 and q21.

**Step 3:** The output of q1 and q2 are not same. So, split these states.

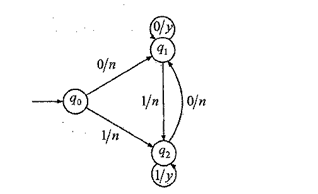
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Δ | Input 0 | Output | Input 1 | Output |
| q0 | q3 | 0 | q11 | 1 |
| q10 | q0 | 1 | q3 | 0 |
| q11 | q0 | 1 | q3 | 0 |
| q20 | q21 | 1 | q20 | 0 |
| q21 | q21 | 1 | q20 | 0 |
| q3 | q10 | 0 | q0 | 1 |

**Step 4:**

Now, convert the above Mealy machine to Moore machine

|  |  |  |  |
| --- | --- | --- | --- |
| Δ | Input 0 | Input 1 | Output |
| q0 | q3 | q11 | 1 |
| q10 | q0 | q3 | 0 |
| q11 | q0 | q3 | 1 |
| q20 | q21 | q20 | 0 |
| q21 | q21 | q20 | 1 |
| q3 | q10 | q0 | 0 |

* + 1. Convert the following mealy machine to moore machine



Sol: Q = {[q0,n], [q0,y], [q1,n], [q1,y], [q2,n], [q2,y]}

Δ = {n, y}

∑ = {0, 1}

**Consider state [q0,n]**

([q0, n], 0) = ([q0, 0], λ[q0,0]

= [q1, N]

([q0, n], 1) = ([q0, 1], λ[q0,1]

= [q2, N]

[(q0, n)] = n

**Consider state [q0,y]**

([q0, y], 0) = ([q0, 0], λ[q0,0]

= [q1, n]

([q0, y], 1) = ([q0, 1], λ[q0,1]

= [q2, n]

[(q0, y)] = y

**Consider state [q1,n]**

([q1, n], 0) = ([q1, 0], λ[q1,0]

= [q1, y]

([q1, n], 1) = ([q1, 1], λ[q1,1]

= [q2, n]

[(q1, n)] = n

**Consider state [q1,y]**

([q1, y], 0) = ([q1, 0], λ[q1,0]

= [q1, y]

([q1, y], 1) = ([q1, 1], λ[q1,1]

= [q2, n]

[(q1, y)] = y

**Consider state [q2,n]**

([q2, n], 0) = ([q2, 0], λ[q2,0]

= [q1, n]

([q2, n], 1) = ([q2, 1], λ[q2,1]

= [q2, y]

[(q2, n)] = n

**Consider state [q2,y]**

([q2, y], 0) = ([q2, 0], λ[q2,0]

= [q1, n]

([q2, y], 1) = ([q2, 1], λ[q2,1]

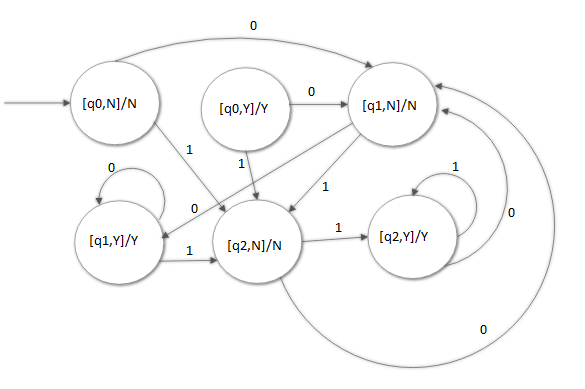
= [q2, y]

[(q0, n)] = n

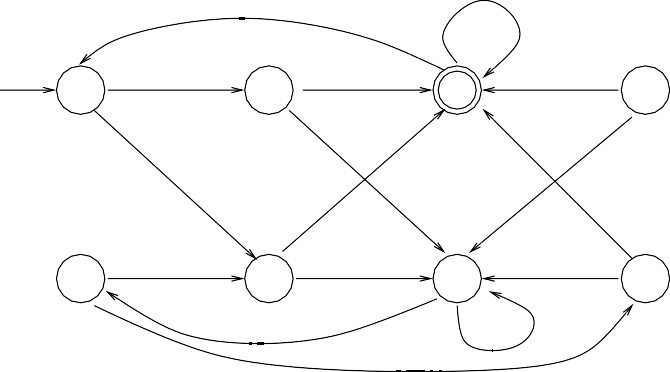
**Transition table:**

|  |  |  |  |
| --- | --- | --- | --- |
| Δ | **Input 0** | **Input 1** | **Output** |
| [q0, n] | [q1,n] | [q2,n] | n |
| [q0, y] | [q1,n] | [q2,n] | y |
| [q1, n] | [q1,y] | [q2,n] | n |
| [q1, y] | [q1,y] | [q2,n] | y |
| [q2, n] | [q1,n] | [q2,y] | n |
| [q2, y] | [q1,n] | [q2,y] | y |

**Transition diagram**

****

**1.14 MINIMIZATION OF THE FOLLOWING FINITE AUTOMATA BY USING MYHILL-NERODE THEOREM**



0

1

A

0

B

1

C

0

D

1

0

1

0

E

1

F

1

G

1

0 H

H

0

1

0

Sol:

Minimization can be applied to only DFA. If the given automata is not DFA, then need to be converted into DFA to apply the minimization procedure.

**METHOD 1:**

**STEP 1:** Group the states into two partitions where one partition belongs to a set of final sates and other partition belongs to set of nonfinal states.

**P1={C} P2 = {A, B, D, E, F, G, H}**

**STEP 2:**

Find the transitions from the two partitions. If the transition for the same input belongs to the same partition, then keep them in the that partition itself, otherwise group into another partition.

**P1={C} P2 = {A, B, D, E, F, G, H}**

**P1={C} P2 = {A, B, E, G, H} P3 = {D, F}**

**P1={C} P2 = {A, B, E, G, H} P3 = {D, F}**

**P1={C} P2 = {A, E} P3 = {D} P4={F} P5 = {B, H} P6 = {G}**

**STEP 3:**

Start state in the minimized automata is the state which contains the start state of the given DFA.

Final state in the minimized automata is the state which contains the final state in the given DFA.

Therefore, start state in the minimized automata is the state {A, E}

Final state is {C}

**METHOD 2:**

**Step 1:** D is unreachable state. So, remove D and proceed to step 2

**Step 2:** First, mark pairs of final and non-final states.

|  |  |
| --- | --- |
| B |  |
| C | X | X |
| E |  |  | X |
| F |  |  | X |  |
| G |  |  | X |  |  |
| H |  |  | X |  |  |  |
|  | A | B | C | E | F | G |

**Step 3:** In the first iteration examine all unmarked pairs. For example, for pair A, B we get δ(A,1)=F and δ(B,1)=C and the pair C,F is marked, so mark A,B too. After doing this for all pairs we get

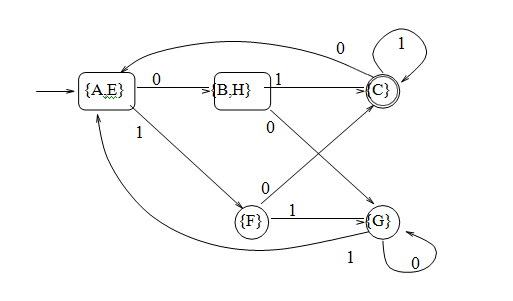
|  |  |
| --- | --- |
| B | X |
| C | X | X |
| E |  | X | X |
| F | X | X | X | X |
| G |  | X | X |  | X |
| H | X |  | X | X | X | X |
|  | A | B | C | E | F | G |

**Step 4:** In the next iteration we examine the remaining pairs

|  |  |
| --- | --- |
| B | X |
| C | X | X |
| E |  | X | X |
| F | X | X | X | X |
| G | X | X | X | X | X |
| H | X |  | X | X | X | X |
|  | A | B | C | E | F | G |

**Step 5:** Here, no new distinguishable pairs will be discovered and now group the states to equivalence classes. Since, {A, E} and {B, H} are equivalent, the classes are {A, E}, {B, H}, {C}, {F}, {G}. T

The minimized automaton is:

****

**CHAPTER -II**

**REGULAR EXPRESSION**

**2.1 INTRODUCTION**

**2.1.1 REGULAR LANGUAGE:** A given language that can be accepted by a Finite Automata is called Regular Language. Every regular language can be described either by DFA or NFA.

One way of describing regular language using regular expressions is in the form of notations. There are three basic operations that are used for obtaining regular expressions. They are: a) Union b) Concatenation c) Kleene-closure.

**2.1.2 REGULAR EXPRESSION FORMAL DEFINITION**

A regular expression over alphabet ∑ is recursively defined as

* is a regular expression that denotes {} i.e., the set containing empty language.
* is a regular expression that denotes {} i.e., the set containing empty string.
* If ‘a’ is a symbol in ∑, then a is a regular expression that denote {a} i.e., the set containing the string a.

Suppose R and S are regular expressions denotes the languages L(R) and L(S) then

a) (R) + (S) is a RE denoting language L(R) L(S).

b) (R). (S) is a RE denoting language L(R).L(S).

c) (R)\* is a regular expression denotes language L(R\*).

**2.1.3 EXAMPLES OF REPRESENTING REGULAR LANGUAGE INTO REGULAR EXPRESSION**

1. Write RE for accepting the string of a’s and b’s starting with a and ending with b.

Sol: a(a+b)\*b

1. Write RE that a) start with abb b) substring abb c) end with abb.

Sol: a) abb(a+b)\* b) (a+b)\*abb(a+b)\* c) (a+b)\*abb

1. Write RE that contains strings consisting of any number of a’s followed by any number of b’s.

Sol: a\*b\*

L={ *,* a, b, ab, aab, abb, aaab, abbb…………}

1. Write RE containing strings with even number of 1’s

Sol: (11)\*

1. Write RE that consists of a zero followed by any number of 1’s.

Sol: 01\*

**2.2 FINITE AUTOMATA AND REGULAR EXPRESSION CONVERSIONS**

* Every regular language accepted by one of these automata is also accepted by regular expression.
* Every language defined by a regular expression is accepted by one of these automata [9].

**Converting Regular Expressions to Finite Automata**

**Theorem:** Every language defined by a regular expression is also defined by finite automata.

**Proof:** Construction of an NFA for each symbol of the regular expression.

Break up the regular expression into sub expressions such that each sub expression has exactly

1. Exactly one final state
2. No edges enter the start state
3. No edges leave the final state

**Basis:** Automata that accept the languages for the simple regular expressions, and a.

The corresponding machine to recognize the language for particular expressions is:

1. There is a path from start state to accepting state is labelled i.e., regular expression =
2. There is no path from start state to accepting state i.e., regular expression =
3. The language of this automaton evidently consists of the one length string a i.e., regular expression = a







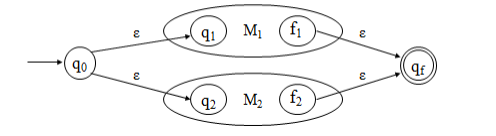
**Induction:** According to the definition of regular expression, if R and S are regular expressions then R + S, R.S and R\* are regular expressions. Consider three cases to construct equivalent machine.

Let M1 = (Q1, ∑1, 1, q1,f1). M1 be the machine which accepts the language L(R) and its regular expression R.

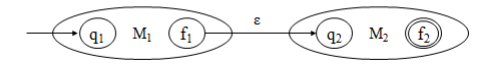
Let M2 = (Q2, ∑2, 2, q2,f2). M2 be the machine which accepts the language L(S) and its regular expression S

The different machines correspond to the regular expressions R + S, R.S and R\* are below

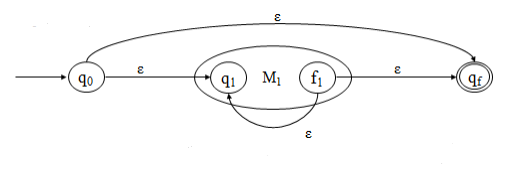
Case 1: R + S (Finite automata which accepts language either L(R) or L(S) which can be represented as L(R+S)).



Case 2: R.S (Finite automata which accepts languages L(R) followed by L(S) which can be represented as L(R.S)).



Case 3: (R)\* (Finite automata which accepts language L(R)\*).



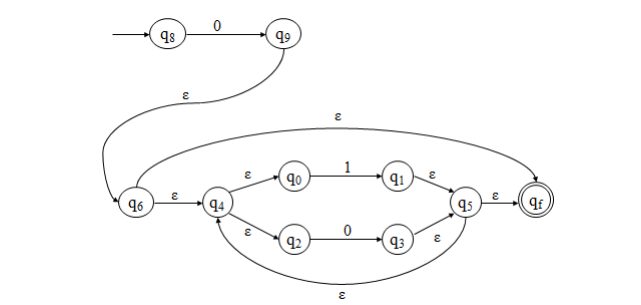
**2.3 PROBLEMS ON THE CONSTRUCTION OF FINITE AUTOMATA EQUIVALENT TO THE REGULAR EXPRESSION**

1. Construct FA equivalent to Regular expression r = 0(0+1)\*.

Sol: r1 = 0

r2= (0+1)\*

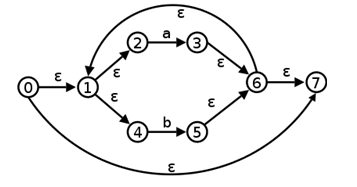
r =r1r2



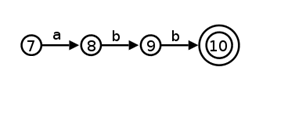
1. Construct FA which accepts strings of a’s and b’s ending with string abb

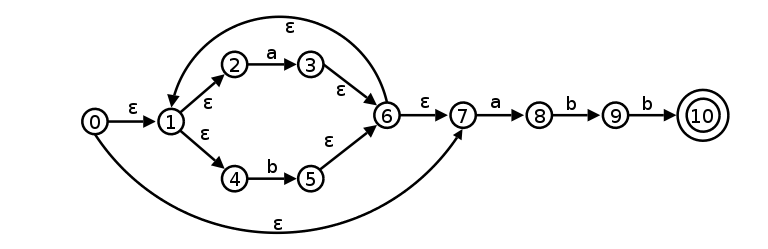
Sol: RE is (a+b)\* abb

Step 1: (a+b)\*



Step 2: abb



Step 3: (a+b)\* abb

**2.4 IDENTITIES OF REGULAR EXPRESSION**

1. + R = R + = R
2. =R =
3. R = R =
4. \* =
5. \* =
6. R\*R\*=R\*
7. R\*R=RR\*
8. (R\*)\* = R\*
9. + RR\* = R\*
10. + R\*R = R\*
11. \* P = P (QP)\*
12. \* = (P\*Q\*)\* = (P\* + Q\*)\*
13. R = PR + QR
14. \* = R\*
15. + R\* = R\*
16. R\* = R\*
17. \* + R\* = R\*
18. (R\*S\*)\* = (R + S)\*

**2.5 PROVING THE EQUIVALENCE OF THE REGULAR EXPRESSIONS USING IDENTITIES OF REGULAR EXPRESSION**

1. From the identities of regular expression, prove that

(1 + 100\*) + (1 + 100\*) (0 + 10\*) (0 + 10\*)\* = 10\*(0 + 10\*)\*.

Sol: **LHS**

(1 + 100\*) + (1 + 100\*) (0 + 10\*) (0 + 10\*)\*

= (1 + 100\*) ( + (0 + 10\*) (0 + 10\*)\*)

= (1 + 100\*) ((0 + 10\*)\*)

= 1 ( + 00\*) (0 + 10\*)\*

= 10\* (0 +10\*)\* **RHS**

Therefore, **LHS = RHS**

1. From the identities of regular expression, prove that

10 + (1010)\* ( + (1010)\*) = 10 + (1010)\*

Sol: **LHS**

10 + (1010)\* ( + (1010)\*)

= 10 + (1010)\* + (1010)\* (1010)\*

= 10 + (1010)\* + (1010)\*

= 10 + (1010)\*  **RHS**

Therefore, **LHS = RHS**

**2.6 CONVERSION OF FINITE AUTOMATA TO REGULAR EXPRESSION USING ARDEN’S THEOREM**

In order to find out a regular expression of a finite automaton, Arden’s theorem is used.

**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**:

R = Q + (Q + RP)P [After putting the value R = Q + RP]

= Q + QP + RPP

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

R = Q + QP + QP2 + QP3…..

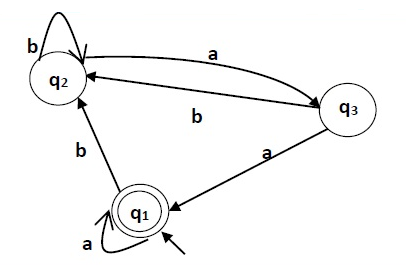
R = Q (ε + P + P2 + P3 + …. )

R = QP\* [As P\* represents (ε + P + P2 + P3 + ….) ]

Hence, proved.

**2.7 EXAMPLES ON THE CONVERSION OF FINITE AUTOMATA TO REGULAR EXPRESSION**

1. Construct a regular expression corresponding to the automata given below −



Sol:

Here the initial state is q1 and the final state is q1.

The equations for the three states q1, q2, and q3 are as follows −

q1 = q1a + q3a + ε (ε move is because q1 is the initial state)

q2 = q1b + q2b + q3b

q3 = q2a

Now, we will solve these three equations −

q2 = q1b + q2b + q3b

= q1b + q2b + (q2a)b (Substituting value of q3)

= q1b + q2(b + ab)

= q1b (b + ab)\* (Applying Arden’s Theorem)

q1 = q1a + q3a + ε

= q1a + q2aa + ε (Substituting value of q3)

= q1a + q1b(b + ab\*)aa + ε (Substituting value of q2)

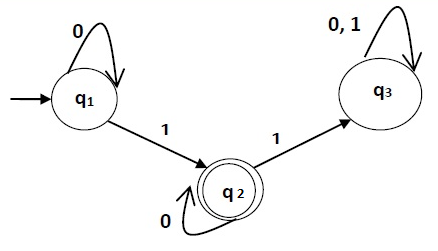
= q1(a + b(b + ab)\*aa) + ε

= ε (a+ b(b + ab)\*aa)\* (Applying Arden’s Theorem)

= (a + b(b + ab)\*aa)\*

Hence, the regular expression is (a + b(b + ab)\*aa)\*.

1. Construct a regular expression corresponding to the automata given below



Sol:

Here the initial state is q1 and the final state is q2

Now we write down the equations −

q1 = q10 + ε

q2 = q11 + q20

q3 = q21 + q30 + q31

Now, we will solve these three equations −

q1 = ε0\* [As, εR = R]

So, q1 = 0\*

q2 = 0\*1 + q20

So, q2 = 0\*1(0)\* [By Arden’s theorem]

Hence, the regular expression is 0\*10\*.

**2.8 EQUIVALENCE OF TWO FINITE AUTOMATON**

Two finite automata are said to be equivalent if they generate the same regular expression.

The following is the process of proving the equivalence between two finite automata.

**STEP 1:** Let there be two finite automata, M and M’, where the number of input symbols are the same.

**STEP 2:** Make comparison table with n+ 1 columns.

**STEP 3:** In the first iteration, there with be pair of vertices, i.e., the initial states of two machines M and M’.

**STEP 4:** The Second column consists of vertices that are reachable from the initial state of the machine M for the first input and the vertices that are reachable from the initial state of machine M’.

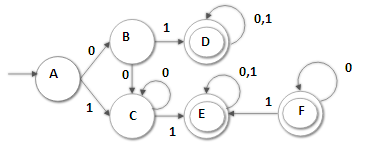
**STEP 5:** If any new pair of states appear, then take take that pair in the present state column.

**STEP 6:** If no new pair of states appear, stop the construction and declare that M and M’ are equivalent.

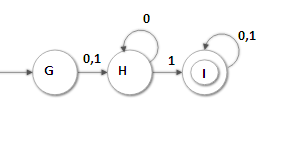
**2.9 EXAMPLES FOR PROVING THAT THE GIVEN TWO FINITE AUTOMATON ARE EQUAL OR NOT**

1. Find whether the two DFA’s are equivalent or not.

M



M’



Sol:

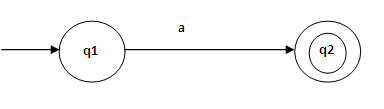
Consider the start states of both M and

|  |  |  |
| --- | --- | --- |
|  | **0** | **1** |
| (A, G) | (B, H) | (C, H) |
| (B, H) | (C, H) | (D, I) |
| (C, H) | (C, H) | (E, I) |
| (D, I) | (D, I) | (D, I) |
| (E, I) | (E, I) | (E, I) |

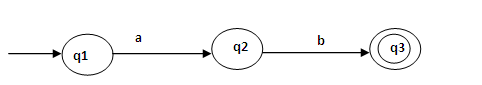
In the above transition table, the pairs of vertices of M and should contain either two final states or two non-final states. So, the above two DFA’s are equivalent.

**2.10 CONVERSION OF REGULAR EXPRESSION TO FINITE AUTOMATA**

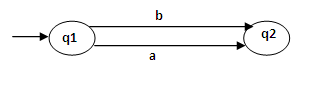
***Case 1*** − For a regular expression ‘a’, we can construct the following FA –



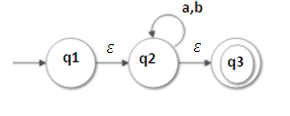
***Case 2*** − For a regular expression ‘ab’, we can construct the following FA –



***Case 3*** − For a regular expression (a+b), we can construct the following FA –



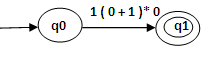
**Case 4** − For a regular expression (a+b)\*, we can construct the following FA –



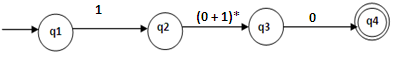
**2.11 EXAMPLES FOR THE CONVERSION OF REGULAR EXPRESSION TO FINITE AUTOMATA**

1) Convert the following RE into its equivalent FA − 1 (0 + 1)\* 0

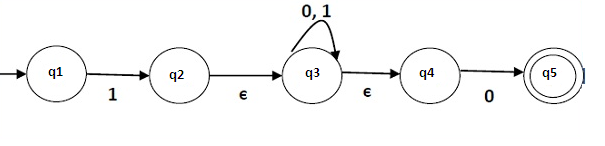
Sol: Step 1:



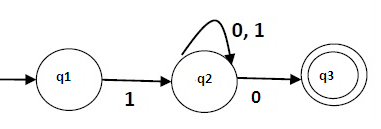
Step 2:



Step 3:



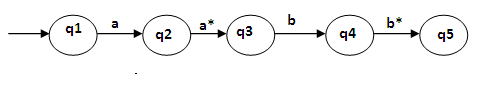
Step 4:



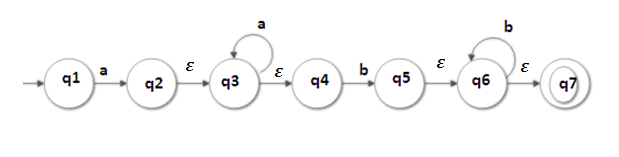
**2)** Construct finite automata for regular expression aa\* bb\*

Sol:

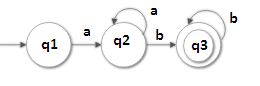
Step 1:



Step 2:



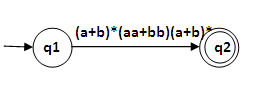
Step 3:



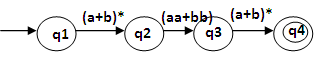
**3)** Construct FA equivalent to RE ( a + b )\* ( aa + bb ) ( a + b )\*

Sol:

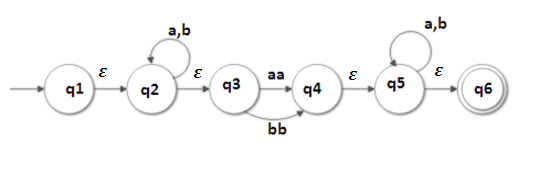
Step 1:



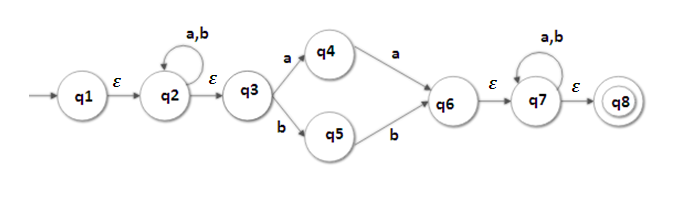
Step 2:



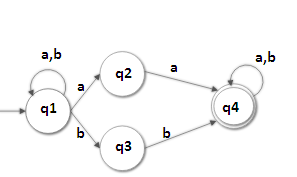
Step 3:



Step 4:



Step 5:

****

**2.12** **PUMPING LEMMA FOR REGULAR EXPRESSIONS**

The pumping lemma is often used to prove that a particular language is non-regular.

**Statement:** Let M = (Q, Σ, , q0, f) be FA with ‘n’ number of states. Let L be the language accepted by M and also assume language is regular. Let w be a string that belongs to set L and |w| ≥ m.

If m≥n, i.e., the length of the string is greater than or equal to the number of states, break w into 3 sub strings.

w=xyz

where, |xy| ≤ n

|y| > 0

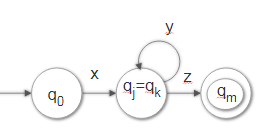
xyiz, i ≥ 0, where xyiz L

**Proof:** Let w be a string of length m.

w = a1a2a3……am

Starting from the beginning state q0, by getting the string w as input, the machine reaches final state.

(q0, a1, a2….ai) = qi for i = 1, 2, 3……….m



Take two integers j and k, 0 ≤ j < k ≤ n

Among the various pairs of repeated states, take a pair qj, qk

The string w is divided into 3 substrings:

a1a2…………aj

aj+1………..ak

ak+1………….am

Let us denote them as x, y, z

This can be expressed using transitions are shown below

(q0, a1, a2, a3………..aj,aj+1……….ak, ak+1,…………….am)

= (qj, aj+1, aj+2,……….ak, ak+1,…………….am)

= (qk, ak+1, ak+2,…………….am)

= qm

The automaton starts from initial state q0

* On applying string x, it reaches state qj
* On applying string y, it comes back to qj as qj=qk
* On applying string z, automaton reaches the final state

Given any long string can be accepted by FA, we should be able to find a substring near the beginning of the string that may be pumped i.e., repeated as many times as we like and resulting string may be accepted by FA.

**2.13 EXAMPLES ON PUMPING LEMMA**

1. Show that L = {an bn where n ≥ 1} is not regular.

Sol: **Step 1:** Assume that set L is regular. Let ‘n’ be the number of states of the finite automata accepting L.

w= anbn L

**Step 2:** Choose a string w such that |w| ≥n. By using pumping lemma, we write

w=xyz

|xy|≤n

|y|>0

w= anbn

w=an-1.a1.bn

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

Consider n=2,

W = aabb (where x=a, y=a, z=bb)

**Step 3:** Find a suitable integer i such that xyiz does not belong to L.

This will contradict our assumption. Therefore, L will be declared as not regular.

Consider i =2

xyiz = a(a)2bb = aaabb doesn’t belongs to L.

Therefore, L is not regular.

1. Prove that L = {wwr | w (0+1)\*} is not regular.

Sol:

**Step 1:** L ={anbnbnan}

Assume L is regular

wwr L i.e., anbnbnan  L

**Step 2:** w =xyz

|xy| ≤ n

|y| >0

aaa.........aabb.........bb = w

bb..........bbaa..........aa = wr

L = {abba, aabbbbaa,.............}

Consider n = 2,

w = aabbbbaa ( where x = a, y = a, z = bbbbaa )

|xy| = 2 ≤ 2

|y| = 1 > 0

**Step 3:** xyiz = a (a)2 bbbbaa = aaabbbbaa doesn’t belongs to L.

Therefore, L is not regular.

**Exercises:**

1. Show that {0n | n is prime} is not regular
2. Show that L = palindrome over {a,b} is not regular.

**2.14 CONSTRUCTION OF FINITE AUTOMATA FROM REGULAR GRAMMAR**

EXAMPLES:

1. Convert the following regular grammar into FA

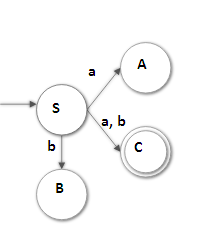
S🡪 aA| bB | a | b

A🡪 aS | bB | b

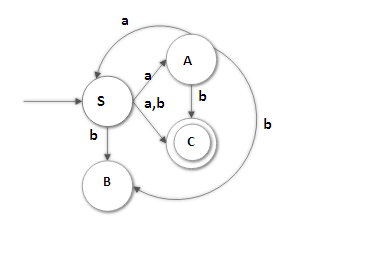
B🡪 aA | bS

Sol: In the grammar, there are three non-terminals, namely S, A and B. Therefore, the number of states of FA is four. Let us name the final state as C.

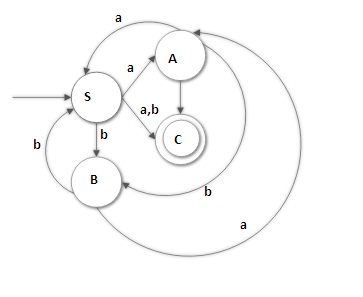
Step 1:



Step 2:



Step 3:



1. Convert the following regular grammar into FA

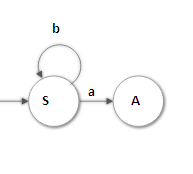
S🡪 aA | bS

A🡪 bB | a

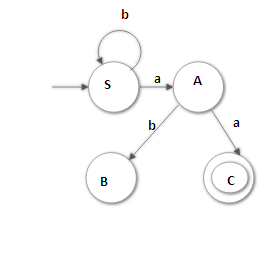
B🡪 aS | b

Sol:

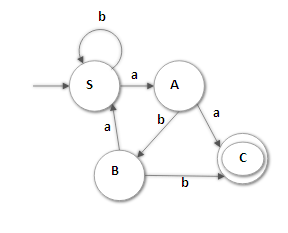
Step 1:



Step 2:



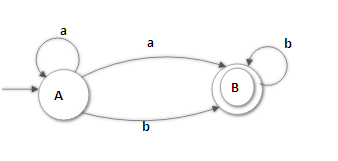
Step 3:



**2.15 CONSTRUCTION OF REGULAR GRAMMAR FROM REGULAR EXPRESSION**

1. Construct regular grammar for following RE’s
2. a\* (a + b) b\*
3. ab (a + b)\*

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



Grammar G = {V, T, P, S}

V = {A, B}

T = {a, b}

P:

{

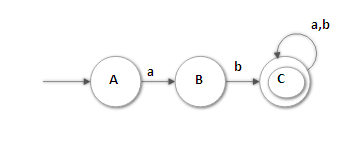
A🡪aA| aB | bB | a | b

B🡪bB | b

}

S = {A}

**b). ab (a + b)\***

****

V = {A, B, C}

T = {a, b}

P:

{

A🡪aB

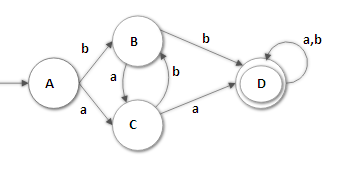
B🡪 bC | b

C🡪aC | bC | a | b

}

S = {A}

1. Construct regular Grammar for given finite automata



Sol: G = {V, T, P, S}

V = {A, B, C, D}

T = {a, b}

P:

{

A🡪bB | aC

B🡪 bD | b | aC

C🡪 bB | aD | a

D🡪 aD | bD | a | b

}

S = {A}

**2.16 CLOSURE PROPERTIES OF REGULAR EXPRESSIONS**

**PROPERTY 1:** The union of two regular set is regular.

**Proof:**

Let us take two regular expressions

RE1 = a(aa)\* and RE2 = (aa)\*

So, L1 = {a, aaa, aaaaa,.....} (Strings of odd length excluding Null)

and L2 = {ε, aa, aaaa, aaaaaa,.......} (Strings of even length including Null)

L1 ∪ L2 = {ε, a, aa, aaa, aaaa, aaaaa, aaaaaa,.......}

(Strings of all possible lengths including Null)

RE (L1 ∪ L2) = a\* (which is a regular expression itself)

Hence, proved.

**PROPERTY 2:** The intersection of two regular set is regular.

**Proof:**

Let us take two regular expressions

RE1 = a(a\*) and RE2 = (aa)\*

So, L1 = {a, aa, aaa, aaaa, ....} (Strings of all possible lengths excluding Null)

L2 = {ε, aa, aaaa, aaaaaa,.......} (Strings of even length including Null)

L1 ∩ L2 = {aa, aaaa, aaaaaa,.......} (Strings of even length excluding Null)

RE (L1 ∩ L2) = aa(aa)\* which is a regular expression itself.

Hence, proved.+

**PROPERTY 3:** The complement of a regular set is regular.

**Proof:**

Let us take a regular expression −

RE = (aa)\*

So, L = {ε, aa, aaaa, aaaaaa, .......} (Strings of even length including Null)

Complement of L is all the strings that is not in L.

So, L’ = {a, aaa, aaaaa, .....} (Strings of odd length excluding Null)

RE (L’) = a(aa)\* which is a regular expression itself.

Hence, proved.

**PROPERTY 4:** The difference of two regular set is regular.

**Proof:**

Let us take two regular expressions −

RE1 = a (a\*) and RE2 = (aa)\*

So, L1 = {a, aa, aaa, aaaa, ....} (Strings of all possible lengths excluding Null)

L2 = { ε, aa, aaaa, aaaaaa,.......} (Strings of even length including Null)

L1 – L2 = {a, aaa, aaaaa, aaaaaaa, ....}

(Strings of all odd lengths excluding Null)

RE (L1 – L2) = a (aa)\* which is a regular expression.

Hence, proved.

**PROPERTY 5:** The reversal of a regular set is regular.

**Proof:**

We have to prove LR is also regular if L is a regular set.

Let, L = {01, 10, 11, 10}

RE (L) = 01 + 10 + 11 + 10

LR = {10, 01, 11, 01}

RE (LR) = 01 + 10 + 11 + 10 which is regular

Hence, proved.

**PROPERTY 6:** The closure of a regular set is regular.

**Proof:**

If L = {a, aaa, aaaaa, .......} (Strings of odd length excluding Null)

i.e., RE (L) = a (aa)\*

L\* = {a, aa, aaa, aaaa , aaaaa,……………} (Strings of all lengths excluding Null)

RE (L\*) = a (a)\*

Hence, proved.

**PROPERTY 7:** The concatenation of two regular sets is regular.

**Proof:**

Let RE1 = (0+1)\*0 and RE2 = 01(0+1)\*

Here, L1 = {0, 00, 10, 000, 010, ......} (Set of strings ending in 0)

and L2 = {01, 010,011,.....} (Set of strings beginning with 01)

Then, L1 L2 = {001,0010,0011,0001,00010,00011,1001,10010,.............}

Set of strings containing 001 as a substring which can be represented by an RE − (0 + 1)\*001(0 + 1)\*

Hence, proved.

**CLOSURE PROPERTIES OF REGULAR LANGUAGES**

1. The set of regular languages is closed under the union operation

2. Finite union is closed for regular languages but infinite union is not closed

3. The set of regular languages is closed under the intersection operation

4. Regular languages are closed under finite intersection but not under infinite intersection

5. The set of regular languages is closed under the difference operation

6. The set of regular languages is closed under the complement operation

7. The set of regular languages is closed under the concatenation operation

8. The set of regular languages is closed under the Kleene closure operation

9. The set of regular languages is closed under the Positive closure operation

10. The set of regular languages is closed under the prefix operation as well as suffix operation

11. The set of regular languages is closed under the reversal operation

12. Half of a Regular Language is also regular.

13. One third of a Regular Language is also regular.

14. Quotient of Regular languages is also regular.

…………………..

15. Regular languages are not closed under subset, superset, infinite union as well as infinite intersection.

**2.17 APPLICATIONS OF REGULAR EXPRESSIONS AND FINITE AUTOMATA**

Regular Expressions can be modelled by finite automata. The smaller unit of regular expressions can express the given language over certain input set. The following are applications of RE:

1. Regular expressions in UNIX
2. Lexical analyzers (compiler design)
3. Finding patterns in text
4. **Regular expressions in UNIX**

There are various UNIX notations used for regular expressions. UNIX regular expressions allow writing character classes. Rules for these character classes given below:

. (dot) symbol stands for any single character.

Eg: “a.cd” matches “abcd”

[ ] symbol denotes range of characters specified using square brackets.

Eg: [a-z] denotes set of lower case letters.

[ ^ ] symbol denotes a character that does not contain within brackets.

Eg: [^a-z] matches a character that is not a lower case letter.

^ symbol matches start of the line.

$ symbol matches end of the line.

/ symbol used to match with some special characters

[:digit:] is the set of ten digits, i.e., [0-9]

[:alpha:] stands for any alphabetic characters [A-Z a-z]

[:alnum:] stands for digits and letters i.e., [A-Z a-z 0-9]

| symbol used to denote union in place of +.

+ used to denote one or more occurrences.

\* used to denote zero or more occurrences.

? used to denote preceding zero or single character.

Eg: ?[0-9] means number can be positive or negative

1. **Lexical analyzers**

Lexical analyzer is the component of a compiler that scans the source program and recognizes all tokens . Example: Keywords and identifiers.

Identifier is a category of a token in the source language and is represented by RE as (letter) (letter + digit)\*.

The definition of identifier in a programming language is

Letter🡪A| B|c|........|Z|a|b|.......z.

Digit🡪0|1|2|......|9

Id🡪(letter) (letter + digit)\*.

1. **Finding patterns in text**

RE are useful notations used for finding the patterns from given text.

Example: Street address🡪 Street|st\. | Avenue|Ave\. |Road|Rd\.

Name of streets🡪 [A-Z] [a-z]\* ([A-Z][a-z]\*)\*

House number🡪 [0-9] + [A-Z]?

**2.18** **DECISION PROBLEMS OF REGULAR EXPRESSION**

Decision problems are the problems which can be answered in ‘yes’ or ‘no’. Finite automata are one type of finite state machines which contain a finite number of memory elements. FA can answer to those decision problems which require only a finite amount of memory.

The following are decision problems of RE:

Whether string X belongs to RE R?

Whether language set of FA M is empty?

Whether language set of FA is finite?

Whether there exists any string accepted by two FA M1 and M2?

Whether language set of FA M1 is subset of M2?

Whether two finite automata’s M1 and M2 accept same language?

Whether two RE’s R1 and R2 are from same language?

Whether an FA is the minimum state FA for language ‘L’?

**2.19 ALGEBRAIC LAWS FOR REGULAR EXPRESSIONS**

|  |  |  |  |
| --- | --- | --- | --- |
| Commutativity | Commutative law for union | This law says that we may take union of two languages in either order. | L + M = M + L |
| Associative law for union | This law says that we may take the union of three languages either by taking the union of the first two initially, or taking the union of last two initially. | (L + M) + N = L + (M + N) |
| Associative law for concatenation | This law says that we can concatenate 3 languages by concatenating either first two or last two initially. | (LM)N = L(MN) |
| Identity | Identity for union | This law asserts that is the identity for union |  |
| Identity for concatenation | This law asserts that is the identity for concatenation |  |
| Annihilator for concatenation | This law asserts that is the annihilator for concatenation |  |
| Distributive law | Left distributive law | This law asserts that left distributive law of concatenation over union | L (M+N) = LM + LN |
| Right distributive law | This law asserts that right distributive law of concatenation over union | (M+N)L = ML + NL |
| Idempotent law | Idempotent law for union | This law asserts that if we take union of two identical expressions, we can replace them by one copy of the expression. | L + L = L |
| Laws involving closures | **-** | Concatenation of strings from L and in the language of L\*. | (L\*)\* = L\* |
| Consider L and M. If we close their union, we get same language | Closure of contains only string. | \* = |

**CHAPTER -III**

**CONTEXT FREE GRAMMAR**

**3.1 Context Free Grammar:** A context-free grammar (CFG) consisting of a finite set of grammar rules is a quadruple (V, T, P, S)

where,

* V is a set of non-terminal symbols.
* T is a set of terminals.
* P is a set of production rules.
* S is the start symbol.

Production Rules are in the form of LHS->RHS

|LHS|<=|RHS|

LHS should be a Variable

**3.2** **EXAMPLES OF CFG**

**1)** Find the language generated by the CFG: S-> 0S1 | 0A | 0 |1B | 1,

A -> 0A | 0,

B -> 1B | 1

**Sol:** The minimum string is S-> 0 | 1

S->0S1=>001

S->0S1=>011

S->0S1=>00S11=>000S111=>0000A111=>00000111

Thus L = {0 **n** 1 **m** | m not equal to n, and n, m >=1}

**2)** Construct a grammar for the language L which has all the strings which are all palindrome over Σ = {a, b}.

**Sol:** G = ({S}, {a, b}, P, S)

P:

{

S -> aSa

S -> bSb

S -> a

S -> b

S -> Є

}

**3)** Construct grammar for the language L which has all the strings which are all even palindrome over Σ = {a, b}.

**Sol:** G = ({S}, {a, b}, P, S)

P:

{

S -> aSa

S -> bSb

S -> Є

}

**4)** Construct grammar for the language L which has all the strings which are all odd palindrome over Σ = {a, b}.

**Sol:** G = ({S}, {a, b}, P, S)

P:

{

S -> aSa

S -> bSb

S -> a

S -> b

}

**5)** Let G = ({S, C}, {a, b}, P, S) where P consists of S -> aCa, C-> aCa | b. Find L(G).

**Sol:** S -> aCa => aba

S -> aCa => a aCa a=>aabaa

S -> aCa => a aCa a=> a a aCa a a =>aaabaaa

L = {aba, aabaa, aaabaaa, aaaabaaaa, aaaaabaaaaa,…………………….}

Thus L(G)= {a n b a n, where n >= 1}

6**)** Find L(G) where G = ({S}, {0,1}, {S->0S1, S->Є}, S)

**Sol:**

S -> 0S1 => 0Є1 => 01

S -> 0S1 => 00S11 => 0011

Thus L(G) = {0 **n** 1 **n** | n>=0}

**7)** Find L(G) where G = ({S}, {0,1}, {S->0S1, S->01}, S)

**Sol:**

S => 01

S=> 0S1 => 0011

S=> 0S1 => 00S11 => 000111

L = {01, 0011, 000111,……………..}

Thus L(G) = {0 **n** 1 **n** | n>=1}

**3.3 DERIVATION:** A derivation uses the productions of a grammar to replace nonterminals until string is formed. There are two types of derivations. They are: 1) left most derivation and 2) right most derivation. Derivation is represented using the symbol => (derives).

**3.3.1 LEFT MOST DERIVATION:** A leftmost derivation is a derivation that always selects the leftmost non-terminal to rewrite. For example, leftmost derivation of id + id \* id.

CFG: E → E + E / E\*E / id

Derive the string id + id \* id

E => E + E

=> id + E

=> id+ E \* E

=> id + id \* E

=> id + id \* id

**3.3.2 RIGHT MOST DERIVATION**: A rightmost derivation is a derivation that always selects the rightmost non-terminal to rewrite. For example, rightmost derivation of id + id \* id.

CFG: E→ E + E / E\*E / id

Derive the string id + id \* id

E => E + E

=> E + E \* E

=> E + E \* id

=> E + id \* id

=> id + id \* id

**3.4** **DERIVATION TREE:** Derivation is pictorially represented using a tree called as derivation or parse tree. A derivation tree or parse tree is an ordered rooted tree that graphically represents the semantic information a string derived from a context-free grammar.

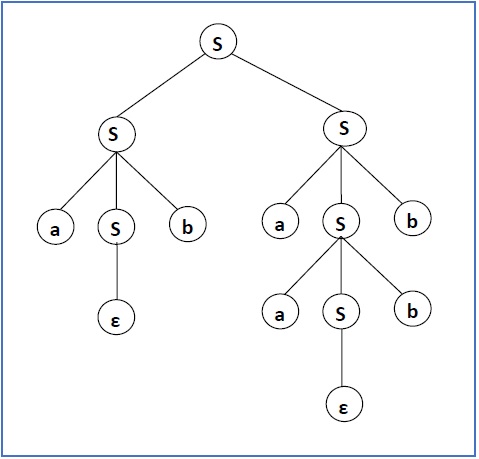
**EXAMPLE**

Let a CFG {V, T, P, S} be

V = {S}, T = {a, b}, S = {S}, P = S → SS | aSb | ε

One derivation from the above CFG is “abaabb”

S => SS => aSbS => abS → abaSb → abaaSbb → abaabb



**3.5** **SIMPLIFICATION OF CFG**

In a CFG, it may happen that all the production rules and symbols are not needed for the derivation of strings. Besides, there may be some null productions and unit productions. Elimination of these productions and symbols is called simplification of CFGs.

Simplification essentially comprises of the following steps −

* Removal of USELESS Symbols
* Removal of UNIT Productions
* Removal of NULL Productions

**3.5.1 USELESS SYMBOLS:** The productions that can never take part in derivation of any string, are called useless productions. The process to eliminate useless symbols: a) non-generating symbols and b) non-reachable symbols.

**EXAMPLE:**

Consider the following productions:

S -> AB/AC

A -> aAb/bAa/a

B -> bbA/aaB/AB

C -> abCA/aDb

D -> bD/aC

After elimination of useless symbols, we get

S -> AB

A -> aAb/bAa/a

B -> bbA/aaB/AB

**3.5.2** **UNIT PRODUCTIONS:** The productions of type NT-> NT are called unit productions.

**EXAMPLE:**

**1)Remove the unit productions from the following grammar**

S -> AB  
A -> a  
B -> C / b  
C -> D  
D -> E  
E -> a

Sol: Here, there are 3-unit productions in the grammar  
B -> C  
C -> D  
D -> E

After eliminating unit productions, we get

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

**2)Identify and remove the unit productions from the following CFG**  
S -> S + T / T  
T -> T \* F / F  
F -> (S) / a

Sol:

UNIT productions in the given CFG are S -> T; T -> F

After removal of unit productions, we get

S -> S + T / T\*F/(S)/a  
T -> T \* F/ (S)/a  
F -> (S)/a

**3.5.3 NULL PRODUCTIONS:** The productions of type ‘A -> ’ are called null productions.

**EXAMPLES:**

1)Identify and remove the null productions from the following CFG

S -> ABAC  
A -> aA / ϵ  
B -> bB / ϵ  
C -> c

Sol: S -> ABAC / ABC / BAC / BC / AAC / AC / C  
A -> aA / a  
B -> bB / b  
C -> c

2) Identify and remove the null productions from the following CFG

S → ASA | aB | b, A → B, B → b | ε

**Sol:** After removing A → ε and B → ε, the production set becomes −

S→ASA | aB | b | a | SA | AS | S, A → B, B → b

**3.6 NORMAL FORMS**

The RHS part of the production rules in Context Free Grammar contains any number of terminals and nonterminal. We need to normalize such grammar. The normalization is nothing but representing the grammar in some specific format.

There are two types of normal forms.

They are:

* Chomsky Normal Form (CNF)
* Greibach Normal Form (GNF)

**3.6.1 CHOMSKY NORMAL FORM (CNF)**

A CFG is in Chomsky Normal Form if the Productions are in the following forms –

NT -> NT NT

NT -> terminal

Example CFG which is in CNF:

* A → BC
* A → a

**Procedure for converting into CNF:**

**Step 1:** Eliminate Useless Symbols, Unit Productions and NULL Productions if any

**Step 2:** Replace the terminals on RHS which are associated with the Non Terminals or with other terminals.

**Step 3:** Restrict the number of Non Terminals on RHS.

**EXAMPLES:**

**1) Convert the following CFG to CNF**

S → aXbX, X → aY | bY | ε, Y → X | c

Sol: Step 1: Eliminate null and unit productions.

After elimination of ε, we obtain: S → aXbX | abX | aXb | ab

X → aY | bY | a | b

Y → X | c

After elimination of the unit production Y → X, we obtain:

S → aXbX | abX | aXb | ab

X → aY | bY | a | b

Y → aY | bY | a | b | c

Step 2: Replace a by A, b by B.

S -> AXBX | ABX | AXB | AB

X -> AY | BY | a | b

Y -> AY | BY | a | b | c

Step 3:

S -> AC

C -> XD

D -> BX

S -> AD

S -> AE

E -> XB

S -> AB

X -> AY | BY | a | b

Y -> AY | BY | a | b |c

**2) Convert the following CFG to CNF**

Convert the following CFG into Chomsky Normal Form: S → AbA, A → Aa | ε

Sol: Step 1: Eliminate useless symbols, unit and null productions.

There are no Useless Symbols, Unit Productions.

Null productions exist. Therefore, remove NULL productions.

S → AbA | bA | Ab | b

A → Aa | a

Step 2: Replace a with C1 and b with C2

S → AC2A | C2A | AC2 | b

A → AC1 | a

C1 → a

C2 → b

Step 3: Replace C2A with D1

S → AD1 | C2A | AC2 | b

A → AC1 | a

D1→ C2A

C1 → a

C2 → b

(or)

S → AS | C2A | AC2 | b

A → AC1 | a

Grammar G={V,T,P,S}

V = {S, A, C1, C2, D1}

T = {a, b}

P:

{

S → AD1 | C2A | AC2 | b

A → AC1 | a

D1→ C2A

C1→ a

C2→ b

}

S is start state.

**3.6.2 GREIBACH NORMAL FORM (GNF)**

A Context free grammar is in Greibach Normal Form (GNF) if the right-hand sides of all production rules start with a terminal symbol or a single terminal.

A🡪aA1A2…………An

A🡪a

Where, a is a terminal and A, A1, A2,………An are non-terminals.

Problems:

1) Convert the following CFG to GNF

S 🡪 AB

A 🡪 aA | bB |b

B 🡪 b

Sol:

G = {V, T, P, S}

V = {S, A, B}

T = {a, b}

P:

{

S 🡪 aAB | bBB | bB

A🡪aA |bB |b

B🡪b

}

S is start state.

2) Convert the following grammar G into Greibach Normal Form (GNF).

S→ AS|AB

A→ BS| a

B→ AA| b

Step 1: Non-terminal symbol (variable) change

Replace A1=S, A2=A, A3=B

A1→ A2A1|A2A3

A2→ A3A1| a

A3→ A2A2| b

Step 2: Consider, A3 → A2A2| b

Substitute A2 production in A3

A3 → A3A1A2 | aA2|b

We see A3 → A3A1A2| aA2|b which has left-recursive from left side.

Now, eliminate left recursion

A3→ aA2 A3’| b A3’

A’3→A1A2 A3’|

Step 3: After elimination of , we get

A3 → aA2 | b | aA2 A3’ | b A3’

A3’ → A1A2 | A1A2 A3’

Step 4: Now, consider A2 production and substitute A3 in A2.

A2 → A3A1| a

A2 → aA2A1|bA1|aA2A3’A1|bA3’A1|a

Step 5: Now, consider A1 production and substitute A2 in A1

A1 → A2A1|A2A3

A1 → aA2A1A1|bA1A1|aA2A3’A1A1|bA3’A1A1|aA1

A1 → aA2A1A3|bA1A3|aA2A3’A1A3|bA3’A1A3|aA3

Step 6: Now, consider A3’ and substitute A1 in A3.

A3’ → A1A2|A1A2 A3’

A3’ → aA2A1A1A2|bA1A1A2|aA2 A3’A1A1A2|b A3’ A1A1A2|

aA1A2|aA2A1A3A2|bA1A3A2|aA2A3’A1A3A2|b A3’A1A3A2|

aA3A2|aA2A1A1A2A3’ |bA1A1A2 A3’|aA2 A3’A1A1A2 A3’|

1. A3’A1A1A2 A3’|aA1A2 A3’|aA1A2 A3’|aA2A1A3A2 A3’| bA1A3A2 A3’|aA2 A3’A1A3A2 A3’|b A3’A1A3A2 A3’|aA3A2 A3’

**3.7 PROBLEMS OF TOP DOWN PARSER**

There are various problems of Top Down Parsers such as Ambiguity, Left Recursion and Left Factoring.

**3.7.1 AMBIGUITY**

Ambiguity is the major problem in Context Free Grammars. This ambiguity has to be removed.

**AMBIGUOUS GRAMMAR:** If a context free grammar G has more than one derivation tree for some string w ∈ L(G), it is called an ambiguous grammar.

**EXAMPLE:** Check whether the grammar G with production rules −

X → X+X | X\*X |X| a

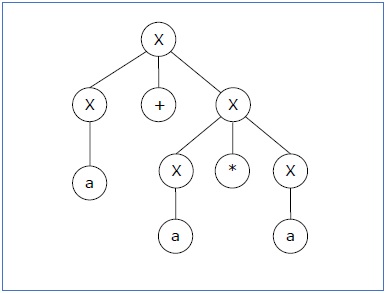
is ambiguous or not.

## Sol: Let’s find out the derivation tree for the string "a+a\*a". It has two leftmost derivations.

Derivation 1 − X → X+X → a +X → a+ X\*X → a+a\*X → a+a\*a

2+3\*4 = 2+12 = 14

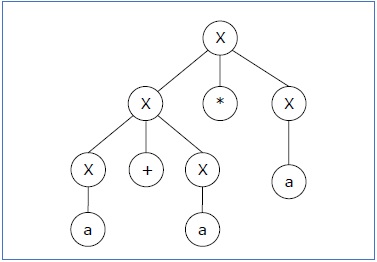
Parse tree 1 −



Derivation 2 − X → X\*X → X+X\*X → a+ X\*X → a+a\*X → a+a\*a

2+3\*4 = 5 \* 4 = 20

Parse tree 2 −



Since there are two parse trees for a single string "a+a\*a", the grammar G is ambiguous.

Ambiguous Grammar:

E -> E+E

E -> E\*E

E -> id

After removing the ambiguity from the grammar, the resultant grammar is going to be

E -> E+T

E -> T

T -> T\*F

T -> F

F -> id

**3.7.2 LEFT RECURSION**

A Context free grammar G = {V, T, P, S} is left recursive if it has a non-terminal ‘A’ such that there is a derivation.

A 🡪 A, where is some string.

From above derivation, the first symbol on the right hand side is same as the symbol on left hand side from where the derivation starts. If the derivation is obtained from the same non-terminal then we say that the grammar is having left recursion.

Procedure for left recursion elimination:

Step 1: Consider A production A🡪A1 | A2 | …….| Am | 1 | 2 | ………..| n.

Step 2: The above productions can be replaced by

A🡪 1A’ | 2 A’| ………..| nA’

Where, A’ 🡪 1A’| 2 A’| …….| mA’ |

**Problems:**

1) Eliminate left recursion from the following grammar

E 🡪 E + T | T

T 🡪 T \* F | F

F🡪(E) | id

**Sol: Step 1:** eliminate left recursion by using productions

A🡪 1A’ | 2 A’| ………..| nA’

A’ 🡪 1A’| 2 A’| …….| mA’ |

**Step 2:** The following two productions are in left recursion.

E 🡪 E + T | T

T 🡪 T \* F | F

**Step 3:** After elimination of left recursion, we get

E 🡪 TE’

E’ 🡪 + TE’ |

T 🡪 FT’

T’ 🡪 \*FT’ |

**Step 4:** Final grammar is:

V = {E, E’, T, T’, F}

T = {+ ,\* ,( ,) , id}

P:

{

E🡪TE’

E’ 🡪 + TE’ |

T 🡪 FT’

T’ 🡪 \*FT’ |

F 🡪 (E) | id

}

E is start symbol.

2) Eliminate left recursion from the following grammar

S 🡪 aA | A

A 🡪 Aa | Sb | b

Sol: The above grammar can be written as

S 🡪 aA | A

A 🡪 Aa | aAb | Ab | b

The final grammar after eliminating left recursion is given by

V = {S, A, A’}

T = {a, b}

P:

{

S 🡪 aA | A

A 🡪 aAbA’ | bA’

A’ 🡪 aA’ | bA’ |.

}

S is start state

**3.7.3 LEFT FACTORING**

In a grammar there is a production rule in the form A🡪1 | 2 | n. The parser generated from this kind of grammar is not sufficient. To avoid this problem, left factoring is needed.

A 🡪 A’

A’ 🡪1 | 2……………. | n

**Problems:**

1) Left factor the following grammar

A🡪abB | aB | cdg | cdeB | cdfB

Sol: In the given grammar, the RHS productions abB and aB, both start with ‘a’. So, they can be left factored. In the same way, cdg, cdeB and cdfB, all start with ‘cd’. So, they can also be left factored.

A🡪abB | aB

A🡪cdg | cdeB | cdfB

The left factored grammar is:

A🡪 a

🡪 bB | B

A🡪 cd

🡪 g | eB | fB

**3.8 PUMPING LEMMA FOR CONTEXT FREE LANGUAGES**

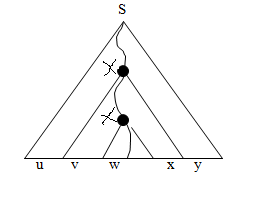
Pumping lemma is used to check whether a grammar is context free or not.

**Theorem:**

**Statement:** For every CFL ‘L’, there is a constant K0 such that for every word Z in ‘L’, there are strings u, v, w, x, y such that z = uvwxy, vx≠ |VWX| k, and for i≥0, the string uviwxiy belongs to L.

**Proof:**

* Let G be a CNF grammar for ‘L’.
* A long string must have a deep parse tree, which in turn means a path with a repeated non-terminal.
* A string of length 2n, must have a parse tree of at least n+2 levels.
* Take K = 2n where ‘n’ is number of non-terminals in G



* Choose the longest path from root to leaf
* Choose the first repeated non-terminal X
* Path from upper X down to leaf is atmost n+2 levels. Also it must be the longest path in the subtree rooted at x.
* Length of VWX is atmost 2n
* Also, VX≠, as G is a CNF grammar.
* Each uviwxiy also belongs to L(G).

**Applications of Pumping Lemma**

Pumping lemma of CFL is used to show that some languages are not context-free. To show that given language is not CFL, the following steps to be followed.

Step 1: Assume the given language L is infinite and it is context free.

Step 2: Consider string z and break into substrings u, v, w, x and y such that

Z = uvwxy

|VWX| ≤ n

|VX| ≥ 1

Step 3: Find suitable number of i, such that uviwxiy ∉ L. This is a contradiction. So, L is not context-free.

**Problems on pumping lemma:**

1) Show that L = {anbncn | n ≥ 1} is not context free.

Sol: Step 1: Assume that the language set L is a CFl. Let ‘n’ be a natural number obtained by using the pumping lemma. anbncn L.

Step 2: z = anbncn

We can write z = uvwxy for strings u, v, w, x, y

| vwx | ≤ n

| vx | ≥ 1

Let n = 4

z = a4b4c4

z = aaaabbbbcccc (where u = aaa, v = ab, w=b, x=b, y=bcccc)

| vwx | = 4 ≤ 4

| vx | = 3 ≥ 1

Step 3: Find suitable integer i such that uviwxiy ∉ L.

Let i = 4

z = a3 (ab)4 (b) (b)4 b (c)4

z = a3a4b4b6c4

z = a7b10c4 ∉ L

This is a contradiction, and L is not context free.

2) Prove L ={ ww | w (a + b)\* } is not context free

Sol: Step 1: Assume language L is CFL.

ww L

Step 2: L = ww

aa……….aabb………bb = w

aa……….aabb………bb = w

w = anbn

ww = anbn anbn

let n = 4

z = a4b4a4b4

z = aaaabbbbaaaabbbb ( u = aaaab, v = bb, w = b, x = a, y = aaabbbb)

| vwx | = 4 ≤ 4

| vx | = 3 ≥ 1

Step 3: Find suitable integer value for i

Let i = 4

z = uviwxiy

z = a4b(b2)4(b)(a)4a3b4

z = a4b10a7b4 ∉ L.

This is a contradiction. So, L is not context free.

Exercises:

3) Prove that the language L = {0n1n0n | n ≥ 0} is not context free language.

4) Prove that L = {0n | where n is prime} is not CFL?

**3.9 CLOSURE PROPERTIES OF CONTEXT FREE LANGUAGES**

Context-free languages are **closed** under −

* Union
* Concatenation
* Kleene Star operation

## Union

Let L1 and L2 be two context free languages. Then L1 ∪ L2 is also context free.

### Example

Let L1 = { anbn , n > 0}. Corresponding grammar G1 will have P: S1 → aAb|ab

Let L2 = { cmdm , m ≥ 0}. Corresponding grammar G2 will have P: S2 → cBb| ε

Union of L1 and L2, L = L1 ∪ L2 = { anbn } ∪ { cmdm }

The corresponding grammar G will have the additional production S → S1 | S2

## Concatenation

If L1 and L2 are context free languages, then L1L2 is also context free.

### Example

Union of the languages L1 and L2, L = L1L2 = {anbncmdm }

The corresponding grammar G will have the additional production S → S1 S2

## Kleene Star

If L is a context free language, then L\* is also context free.

### Example

Let L = {anbn , n ≥ 0}. Corresponding grammar G will have P: S → aAb| ε

Kleene Star L1 = {anbn}\*

The corresponding grammar G1 will have additional productions S1 → SS1 | ε

Context-free languages are **not closed** under −

* **Intersection** − If L1 and L2 are context free languages, then L1 ∩ L2 is not necessarily context free.
* **Intersection with Regular Language** − If L1 is a regular language and L2 is a context free language, then L1 ∩ L2 is a context free language.
* **Complement** − If L1 is a context free language, then L1’ may not be context free.

**Linear Grammar**

A grammar is called linear grammar if it is context free and the RHS of all productions have at most one non- terminal.

Ex: S🡪 aA

A🡪aBb

There are two types of grammar. They are 1) left linear grammar and 2) right linear grammar.

**Left linear grammar:**

In a grammar, if all productions are in the form A🡪B and A🡪, then grammar is called left-linear. Here, A, B are non-terminals and is terminal.

Example: A🡪Aa

**Right-linear grammar:**

In a grammar, if all the productions are in the form A🡪B and A🡪, then grammar is called right linear.

Example: A🡪aA

**Problems:**

1) Convert the following right-linear to left-linear grammar

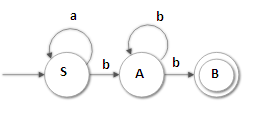
S🡪aS

S🡪bA

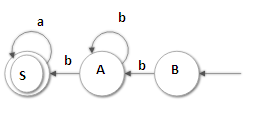
A🡪bA

A🡪b

Sol: Step 1: convert the above production to FA.



Step 2: now change final state as start state and start state as final state



Step 3: B🡪bA

A🡪bA | bS | b

S🡪aS | a

Step 4: Now convert above productions to left linear

B🡪Ab

A🡪Ab | Sb |b

S🡪Sa | a

Final grammar is:

G= {V, T, P, S}

V = {B, A, S}

T = {a,b}, B is start state.

P:

{

B🡪Ab

A🡪Ab | Sb | b

S🡪Sa | a

}

**2)** Convert the following right-linear grammar to left-linear grammar?

S🡪10A | 1

A🡪0A | 00

Sol:

Step 1: Consider A production

A🡪0A

A🡪00A (replace A by A🡪0A)

A🡪000A (replace A by A🡪0A)

A🡪00000 (Replace A by A🡪00)

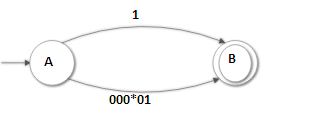
Step 2: Consider S production

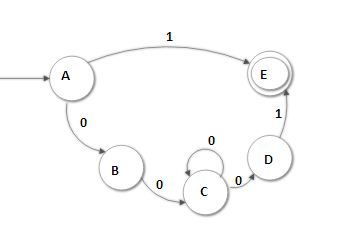
S🡪10A + 1(replace A with RE 0\*00)

S🡪100\*00 + 1

Therefore, the RE generated by grammar 100\*00 + 1

Step 3: Reverse the RE and we get 1 + 000\*01





Step 4: A🡪1E | 1 | 0B

B🡪0C

C🡪0C | 0D

D🡪1E | 1

Step 5: In the productions, E is useless. So remove E.

A🡪0B | 1

B🡪0C

C🡪0C | 0D

D🡪1

Step 6: Reverse the productions to left-linear.

A🡪B0 | 1

B🡪C0

C🡪C0 | D0

D🡪1

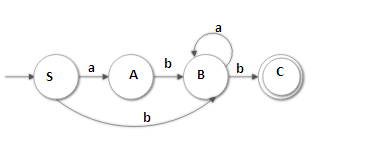
3) Convert the following left linear to right linear grammar?

S🡪Aa | Bb

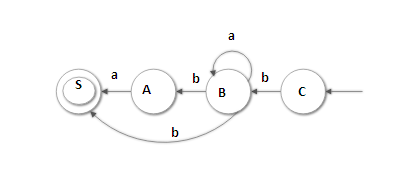
A🡪Bb

B🡪Ba | b

Sol: Step 1: Design FA for the given productions



Step 2: Now change initial state as final state and final state as initial state and interchange directions.



Step 3: C🡪bB

B🡪aB | bS | bA | b

A🡪aS | a

4) Convert the following left-linear to right-linear grammar?

S🡪S10 | A1

A🡪A00 | 00

Sol: Step 1: Reverse the productions

S🡪01S | 1A

A🡪00A | 00

Step 2: Consider A production

A🡪00A

A🡪0000A (replace A by A🡪00A)

A🡪000000A (replace A by A🡪00A)

A🡪00000000 (replace A by A🡪00)

RE for A is (00)\*00

Step 3: Consider S production

S🡪01S | 1A

S🡪01S

S🡪0101S (replace S by S🡪01S)

S🡪010101S (replace S by S🡪01S)

S🡪0101011A (replace S by S🡪1A)

RE for S is:

S🡪01S | 1A

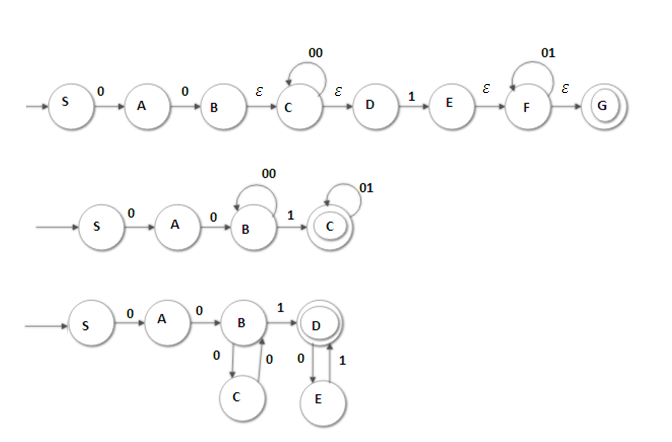
S🡪(01)\*S | 1A

S🡪(01)\*S | 1(00)\*00

S🡪(01)\*1(00)\*00

So, the regular expression is (01)\*1(00)\*00

Step 4: Reverse regular expression 00(00)\*1(01)\*



Step 5: S🡪0A

A🡪0B

B🡪1D | 0C | 1

C🡪0B

D🡪0E

E🡪1D | 1

**3.10 DECISION PROBLEM RELATED TO CONTEXT FREE LANGUAGES**

Decision problems are the problems which can be answered in ‘yes’ or ‘no’. Some of them are:

a) Whether given CFG is empty or not?

b) Whether given CFG is finite?

c) Whether a particular string w is generated by CFG?

d) Whether given CFG is ambiguous

e) Whether two given CFG have a common word

f) Whether two different CFG generate same language

g) Whether the compliment of a given CFL is also a CFL.

**Problems:**

1) Check whether the following CFG is empty or not

S🡪AB | D

A🡪aBC

B🡪bC

C🡪d

D🡪CD

Sol:

Step 1: Replace C🡪d

S🡪AB | D

A🡪aBd

B🡪bd

D🡪dD

Step 2: Replace B🡪bd

S🡪Abd | D

A🡪abdd

D🡪dD

Step 3: Replace A🡪abdd

S🡪abddbd | D

D🡪dD

S generates string of terminals. So, CFG is non-empty.

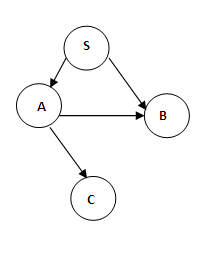
2) Verify whether languages generated by the following grammar are finite or not.

a) S🡪AB

A🡪BC

B🡪a

Sol: The language generated by CFG is finite.



**Applications of CFG:**

CFG is a powerful tool used in compiler for syntax analysis.

CFG used to develop extensive markup language (XML).

**CHAPTER -IV**

**PUSH DOWN AUTOMATON**

**4.1 Basic Structure of Push Down Automata**

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

Basically a pushdown automaton is −

**"Finite state machine" + "a stack"**

A pushdown automaton has three components −

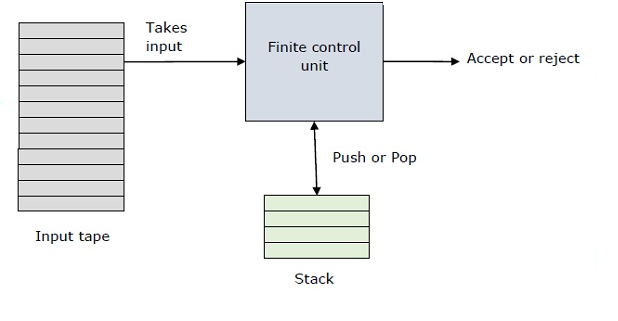
* an input tape,
* a control unit, and
* a stack with infinite size.

The stack head scans the top symbol of the 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 has to read the top of the stack in every transition.



**4.2 Formal definition of PDA**

A PDA can be formally described as a 7-tuple (Q, Σ, Γ, δ, q0, Z0, F).

* Q is the finite number of states
* ∑ is finite set of input alphabets
* Γ is finite set of stack symbols
* δ is the transition function

Q X (Σ U {Є} ) X Ґ to finite subsets of Q X Ґ \*.

* q0 is the initial state
* Z0 is the initial stack top symbol
* F is a set of accepting states

**4.3 Graphical representation of PDA**

Graphical representation of PDA is transition diagram of PDA. The nodes correspond to the states of the PDA. An arrow labeled *Start* indicates the unique start state. Doubly circled states are accepting states. The arc represents the previous and current top of the stack.

**4.4 Instantaneous description (ID) of PDA**

ID is an informal notation of how a PDA computes a input string and make a decision that string is accepted or rejected.

Notation is (q,w, γ). q is current state, w is input string and γ is current contents of the PDA stack.

**4.5 Acceptance of PDA**

There are two different ways to define PDA acceptability.

**Final state acceptability**

In final state acceptability, a PDA accepts a string when, after reading the entire string, the PDA is in a final state. From the starting state, we can make moves that end up in a final state with any stack values. The stack values are irrelevant as long as we end up in a final state.

For a PDA (Q, ∑, S, δ, q0, I, F), the language accepted by the set of final states F is −

L(PDA) = {w | (q0, w, I) ⊢\* (q, ε, x), q ∈ F} for any input stack string **x**.

**Empty stack Acceptability**

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

For a PDA (Q, ∑, S, δ, q0, I, F), the language accepted by the empty stack is −

L(PDA) = {w | (q0, w, I) ⊢\* (q, ε, ε), q ∈ Q}

**4.6 Types of PDA:**

There are two types of PDA:

Deterministic PDA

Non-deterministic PDA [18]

**4.6.1 Deterministic PDA**

A deterministic pushdown automaton has at most one legal transition for the same combination of input symbol, state, and top stack symbol.  The class of deterministic pushdown automata accepts the deterministic context free languages.

Example: L = {0n 1n | n ≥ 0}

**4.6.2 Non-deterministic PDA**

Non-deterministic PDA is basically an NFA with a stack added to it. It has more than one legal transition for the same combination of input symbol, state, and top stack symbol.

Example: L = { wwR | w ϵ (a+b)\* }

**4.7 Problems on PDA**

1. Design the pushdown automata for language {an bn | n > 0}

Sol: Rule 1: δ(, a, ) = { (, X ) }  
 δ(, a, X) = { (, XX ) }

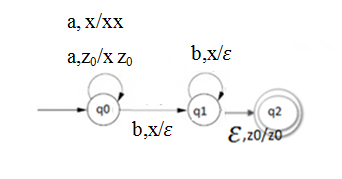
Rule 2: δ(, b, X) = { (, ɛ) }  
 δ(, b, X) = { (, ɛ) }  
Rule 3: δ(, ɛ, ) = { (, )) }

Q = { q0, q1 }, ∑ = { a, b }, 🡪 start state, 🡪final state

δ: δ(, a, ) = { (, X ) }  
 δ(, a, X) = { (, XX ) }

δ(, b, X) = { (, ɛ) }

δ(, b, X) = { (, ɛ) }  
δ(, ɛ, ) = { (, )) }



1. Design PDA for the language L = { | w є {a,b}\* }

Sol: P=({q0 ,q1 ,q2 }, {a,b}, {a,b,Z0 }, δ, q0 , Z0 , {q2 })

δ is as follows:

**rule 1:** One of these rules applies initially when we are in q0 and we see the start symbol at the top of the stack.

δ( ,a, ) = {( ,a )}

δ( ,b,) = {( ,b )}

**rule 2:** These four similar rules allow us to stay in q0 and read inputs, pushing each onto the top of the stack, and leaving the previous stack symbol alone.

δ( ,a,a) = {( ,aa)}

δ( ,b,b) = {( ,bb)}

δ( ,b,a) = {( ,ba)}

δ( ,a,b) = {( ,ab)}

**rule 3:** These three rules allow us to go from q0 to state q1 spontaneously (on є input)

δ( ,є, ) = {(q1 , )}

δ( ,є,a) = {(q1 ,a)}

δ( ,є,b) = {(q1 ,b)}

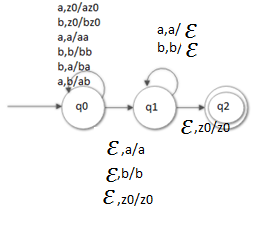
**rule 4:** In state q1 we can match input symbols against the top of the stack and pop when they match.

δ( ,a,a) = {( ,є)},

δ( ,b,b) = {( ,є)}

**rule 5:** Finally, if we see the bottom of the stack marker Z0 and we are in state q1 we have found a w. Thus go to q2 and accept.

δ( ,є, ) = {( , )}



1. Design the pushdown automata for language {an  | n > 0}

Sol: Rule 1: δ(, a, ) = { (, XX ) }  
 δ(, a, X) = { (, XXX ) }

Rule 2: δ(, b, X) = { (, ɛ) }  
 δ(, b, X) = { (, ɛ) }  
Rule 3: δ(, ɛ, ) = { (, )) }

Q = { , }, ∑ = { a, b }, 🡪 start state, 🡪final state

δ:

δ(, a, ) = { (, XX ) }  
 δ(, a, X) = { (, XXX ) }

δ(, b, X) = { (, ɛ) }

δ(, b, X) = { (, ɛ) }  
δ(, ɛ, ) = { (, )) }

1. Design PDA for the language L = { w| w є {a,b}\* }

Sol: P=({q0 ,q1 ,q2 }, {a,b}, {a,b,Z0 }, δ, q0 , Z0 , {q2 })

δ is as follows:

**rule 1:** One of these rules applies initially when we are in q0 and we see the start symbol at the top of the stack.

δ( ,a, ) = {( ,a )}

δ( ,b,) = {( ,b )}

**rule 2:** These four similar rules allow us to stay in q0 and read inputs, pushing each onto the top of the stack, and leaving the previous stack symbol alone.

δ( ,a,a) = {( ,aa)}

δ( ,b,b) = {( ,bb)}

δ( ,b,a) = {( ,ba)}

δ( ,a,b) = {( ,ab)}

**rule 3:** These three rules allow us to go from q0 to state q1 spontaneously (on є input)

δ( ,c, ) = {(q1 , )}

δ( ,c, a) = {(q1 ,a)}

δ( ,c, b) = {(q1 ,b)}

**rule 4:** In state q1 we can match input symbols against the top of the stack and pop when they match.

δ( ,a,a) = {( ,є)},

δ( ,b,b) = {( ,є)}

**rule 5:** Finally, if we see the bottom of the stack marker Z0 and we are in state q1 we have found a w. Thus, go to q2 and accept.

δ( ,є, ) = {( , )}

1. Design PDA for Palindrome of binary numbers.

Sol: P=({ , , }, {a,b}, {a,b, }, δ, , , {q2 })

δ is as follows:

**rule 1:** One of these rules applies initially when we are in q0 and we see the start symbol at the top of the stack.

δ( ,a, ) = {( ,a )}

δ( ,b,) = {( ,b )}

**rule 2:** These four similar rules allow us to stay in q0 and read inputs, pushing each onto the top of the stack, and leaving the previous stack symbol alone.

δ( ,a,a) = {( ,aa)}

δ( ,b,b) = {( ,bb)}

δ( ,b,a) = {( ,ba)}

δ( ,a,b) = {( ,ab)}

**rule 3:** These three rules allow us to go from q0 to state q1 spontaneously (on є input)

δ( , ɛ, ) = {(q1 , )}

δ( ,a, a) = {(q1 , ɛ)}

δ( ,b, b) = {(q1 , ɛ)}

**rule 4:** In state q1 we can match input symbols against the top of the stack and pop when they match.

δ( ,a,a) = {( ,є)},

δ( ,b,b) = {( ,є)}

**rule 5:** Finally, if we see the bottom of the stack marker Z0 and we are in state q1 we have found a palindrome of binary numbers. Thus, go to q2 and accept.

δ( ,є, ) = {( , )}

1. Design DPDA for number of a(w) = number of b(w)

Sol: P = ({ ,}, {a, b}, {a, b, }, δ, , , { })

δ( ,a, ) = {( ,a )}

δ( ,b,) = {( ,b )}

δ( ,a,a) = {( ,aa)}

δ( ,b,b) = {( ,bb)}

δ( ,a,a) = {( ,є)},

δ( ,b,b) = {( ,є)}

δ( ,є, ) = {( , )}

1. Design a PDA to accept string of balanced parenthesis.

Sol: Consider [ ] and { }.

Rule 1: δ( ,[, ) = ( ,[ )

δ( ,{,) = ( ,{ )

Rule 2: δ( ,[,[) = ( ,[[ )

δ( ,[,{) = ( ,[{ )

δ( ,{,[) = ( ,{[ )

δ( ,{,{) = ( ,{{ )

Rule 3: δ( ,],[) = ( є)

δ( ,},{) = ( є)

Rule 4: δ( ,є, ) = {( , )}

**4.8 Equivalence of CFG and PDA**

If L is a context free language, then there exists a PDA M such that L=N(M). If L is N(M) for some PDA m, then L is a context free language [20].

**4.8.1 Convert PDA to CFG**

**Procedure:**

**Step 1:**

For every transition of the form

δ( ,a, ) = {( , )}

Introduce the production of the form

δ( ) a ( ) ( B )

**Step 2:**

For every transition of the form

δ( ,a, ) = {( ,)}

Introduce the production of the form

δ( ) a

1. Convert the following PDA to CFG.

δ(q, 0, Z) = { (q, XZ ) }  
 δ(q, 0, X) = { (q, XX ) }

δ(q, 1, X) = { (r, ɛ) }  
δ(r, 1, X) = { (r, ɛ) }  
δ(r, ɛ, Z) = { (r, ɛ)) }

Sol: step 1: For the transition functions 3,4 and 5, the productions are:

[qXr] → 1

[rZr] → ɛ

[rXr] → 1

Step 2: For the transition function 1, the productions are:

[qZq] → 0[qXq][qZq] | 0[qXr][rZq]

[qZr] → 0[qXq][qZr] | 0[qXr][rZr]

` Step 3: For the transition function 2, the productions are:

[qXq] → 0[qXq][qXq] | 0[qXr][rXq]

[qXr] → 0[qXq][qXr] | 0[qXr][rXr]

**4.8.2 Convert CGF to PDA**

Procedure to convert CFG to PDA is as follows:

* + - δ(q, a, a) = (q, ε)
    - δ(q, ε, A) = (q, α)

**Problems:**

1. Construct PDA M equivalent to the CFG S 🡪 0BB, B🡪 1S|0S|0 and check whether 010000 is in N (M) or not?

Sol: δ( , ε , ) = {( ,0BB )}

δ( , ε, B ) = {( ,1S )}

δ( , ε, ) = {( ,0S)}

δ( , ε, ) = {( ,0 )}

δ( ,0,0 ) = {( , ε )}

δ( ,1,1 ) = {( , ε )}

For the string w=010000, M has following moves:

(, 010000, S) |-- (, 010000, 0BB)

|-- (, 10000, BB)

|-- (, 10000, 1SB)

|-- (, 0000, SB)

|-- (, 0000, 0BBB)

|-- (, 000, BBB)

|-- (, 000, 0BB)

|-- (, 00, BB)

|-- (, 00, 0B)

|-- (, 0, B)

|-- (, 0, 0)

|-- (, ε )

**4.9 Two stack PDA**

The two-stack PDA is similar to PDA, but it has two stacks instead of one. In each transition we must denote the push and pop action on both stacks.

Input Tape

Finite state control unit

Tape Head

Head moves from left to write

Stack 1

Stack 2

**4.10 Problem on the construction of two stack PDA**

Construct Two stack PDA for language L={ an bn cn /n>=0 }

Sol:

δ( , ε ,) = {( , )}

δ( , a, ) = {( , )}

δ( , a, ) = {( ,aa,)}

δ( , b, ) = {( , a,b)}

δ( ,b, a, b ) = {( , a, bb )}

δ( ,c, a, b ) = {( , ε, ε )}

δ( ,c, a, b ) = {( , ε, ε )}

δ( , ε, ) = {( , )}

**4.11 Types of PDA**

There are two types of PDA. They are 1) Deterministic PDA and 2) non- Deterministic PDA.

|  |  |
| --- | --- |
| DPDA | NPDA |
| 1. DPDA means deterministic push down automata | 1. NPDA means Non-deterministic PDA |
| 1. A PDA is said to be DPDA if all the derivations in the design give only a single move. | 2) A PDA is said to be NPDA if one of the derivations generates more than one move. |
| 1. If a PDA being in a state with a single input and a single stack symbol gives a single move, then PDA is called DPDA. | 3) If a PDA being in a state with a single input and a single stack symbol gives more than one move for any of its transition functions, then pda is called NPDA. |
| 1. Example: | 4) Example: |

**CHAPTER -V**

**TURING MACHINE**

**5.1 Introduction to Turing Machine**

Turing Machine (TM) is a simple mathematical model of modern digital computers having strong computational capacity. The TM is similar to finite automata with an unlimited and unrestricted memory. TM is more powerful than PDA. Alen Turing proposed Turing machine.

**5.2 Formal Definition of TM**

A TM can be formally described as a 7-tuple (Q, ∑, δ, q0, B, Γ, F) where −

* Q is a finite set of states
* Γ is the finite set of tape alphabet symbols
* ∑ is the finite set of input symbols
* δ is a transition function; δ : Q × Γ → Q × Γ × {L, R, H}.
* q0 is the initial state
* B is the blank symbol
* F is final state

**5.3 Mechanical diagram of TM**

A Turing machine consists of an input tape, a read-write head and finite control [22].

**Input tape:**

A **tape** divided into cells, one next to the other. Each cell contains a symbol from some finite alphabet. The alphabet contains a special blank symbol and one or more other symbols.

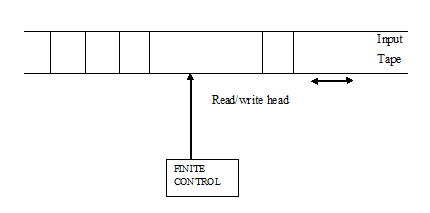
**Read/write head:**

A **head** can read and write symbols on the tape and move the tape left and right one cell at a time.

**Finite control:**

In this transitional functions are written. The following 3 operations are performed for every transition function.

* Machine goes into some state
* Machine writes a symbol in the cell of input tape from where the input symbol was scanned.
* The machine moves reading head to the left or right or halts.



**5.4 Instantaneous Description (ID) of TM**

The contents of all the cells of the tape, contains a non-blank symbol and contains all cells up to the cell being scanned. The cell is currently being scanned by the read-write head and the state of the machine.

Ex: consider string aabb from the language L = {an bn}

q0 aabb move to xq1abb move to xaq1bb move to xq2ayb move to q2xayb move to xq0ayb move toxxq1yb move to xxyq1b move toxxq2yy move to xq2xyy move to xxq0yy move to xxyq3y move toxxyyq3B move to xxyyBq4B

**5.5 Transition diagram for TM**

Transition diagram of TM is represented diagrammatically which consists of set of nodes

corresponds to the states of Turing machine. Circle with an arrow indicates start state. State with double circle indicates final state. State transitions are denoted by arrows. The labels of the state transitions consist of input symbol, the symbol written on the tape after traversal, and the direction of movement of the read-write head.

**Problems:**

1. Design a Turing machine to accept the language L = {0n1n/ n≥1}

Sol: δ(,0) = (, X, R )

δ (,0) = (, 0, R )

δ (,1) = (, Y, L )

δ (,0) = (, 0, L )

δ (,x) = (, X, R )

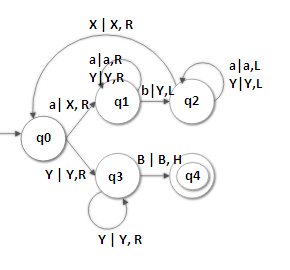
δ (,y) = (, Y, R )

δ (,y) = (, Y, L )

δ (,y) = (, Y, R )

δ (,y) = (, Y, R )

δ (,B) = (, B, H )



M = (Q, ∑, δ, q0, B, Γ, F)

Q = {q0, q1, q2, q3, q4}

∑ = {a, b}

Γ = {a, b, X, Y, B}

q0 🡪 start state

B🡪 Blank symbol

q4 🡪 final state

2) Design a Turing machine to accept the language L = {anbn cn/ n≥1}

Sol: δ(,a) = (, X, R )

δ (,a) = (, a, R )

δ (,b) = (, Y, R )

δ (,b) = (, b, R )

δ (,c) = (, Z, L )

δ (,b) = (, b, L )

δ (,y) = (, Y, L )

δ (,a) = (, a, L )

δ (,X) = (, x, R )

δ (,y) = (, Y, R )

δ (,z) = (, Z, R )

δ (,z) = (, Z, L )

δ (,y) = (, Y, R )

δ (,y) = (, Y, R )

δ (,z) = (, Z, R )

δ (,z) = (, Z, R )

δ (,B) = (, B, H )

M = (Q, ∑, δ, q0, B, Γ, F)

Q = {q0, q1, q2, q3, q4, q5}

∑ = {a, b}

Γ = {a, b, X, Y, B}

q0 🡪 start state

B🡪 Blank symbol

q5 🡪 final state

**5.6 Conversion of Regular Expression to Turing Machine**

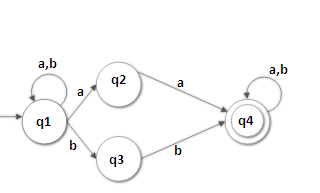
Steps to convert RE to TM:

* Convert RE to FA with out -move
* Insert a new initial state with transition (B, B, R)
* Convert transitions with label a to (a, a, R)
* Insert a new final state with transition (B, B, R)

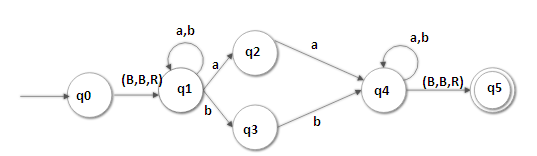
**Problems:**

1) Construct TM for the RE (a+b)\* (aa + bb) (a + b)\*

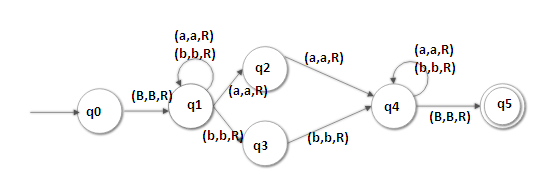
Sol: Step 1: Convert RE to FA



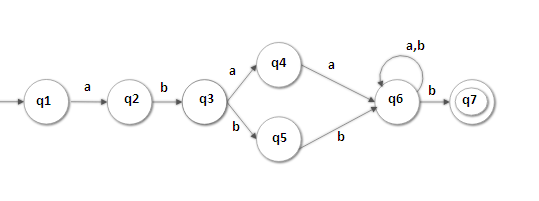
Step 2: Add a new state before the start state and after the final state and label it as (B,B,R).

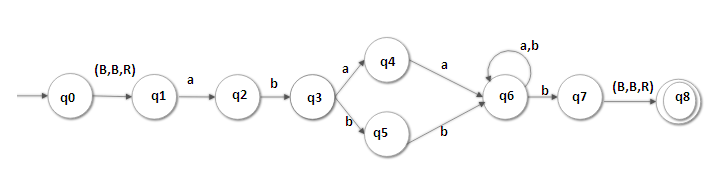
****

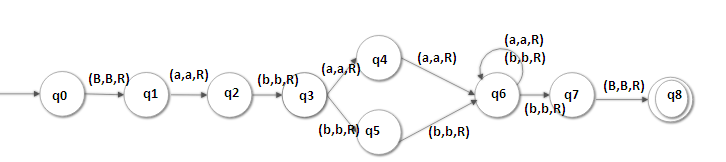
Step 3: Replace a with (a,a,R) and b with (b,b,R).

****

**2)** Construct TM for RE ab(aa+bb)(a+b)\*b.

****

****

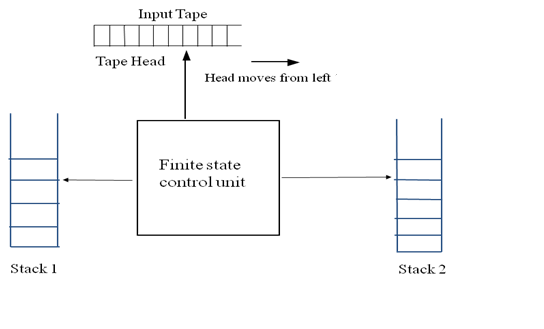
****

**5.7 Two stack PDA and Turing Machine**

Any Language accepted by a two- stack PDA can also be accepted by some TM and vice versa.

**Minsky theorem**

Two stack PDA contains input tape, reading head and finite control as it was in PDA. The two stacks S1 and s2 are included in 2-stack PDA. In real, the language accepting power of a PDA increases by adding extra stacks.

****

**Problem:**

1. Design two stack PDA for (anbncndn, n≥1)

Sol:

Step 1: Read ‘a’ and push ‘a’ in stack s1

Step 2: Read ‘b’ and pop ‘a’ from s1 by ‘b’ and push ‘b’ into stack s2

Step 3: Read ‘c’ and pop ‘b’ from s2 by ‘c’ and push ‘c’ into stack s1

Step 4: Read‘d’ and pop ‘c’ from s1

δ( , ε ,) = {( , )}

δ( , a, ) = {( , )}

δ( , a, ) = {( ,aa,)}

δ( , b, ) = {( ,ε, b)}

δ( ,b, a, b ) = {( , ε, bb )}

δ( ,c, z1, b ) = {( , cz1, ε )}

δ( ,c, c, b ) = {( , cc, ε )}

δ( ,c, c, b ) = {( , cc, ε )}

δ( ,d, c, z2 ) = {( , ε, z2 )}

δ( ,d, c, z2 ) = {( , ε, z2 )}

δ( , ε, ) = {( , )}

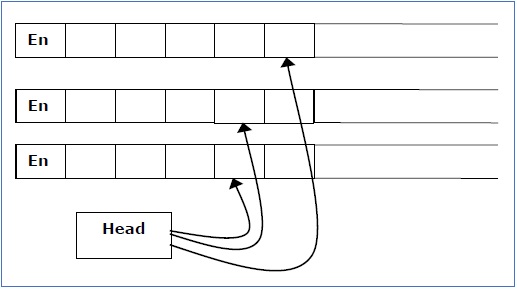
**5.8 Variations of Turing Machines**

The different types of Turing Machines are:

* + Multi- tape TM
  + Multi – Head TM
  + Two- way infinite tape
  + K-dimensional TM
  + Non- deterministic TM
  + Enumerator

**Multi-tape TM**

Multi-tape Turing Machines have multiple tapes where each tape is accessed with a separate head. Each head can move independently of the other heads. Initially the input is on tape 1 and others are blank. At first, the first tape is occupied by the input and the other tapes are kept blank. Next, the machine reads consecutive symbols under its heads and the TM prints a symbol on each tape and moves its heads.

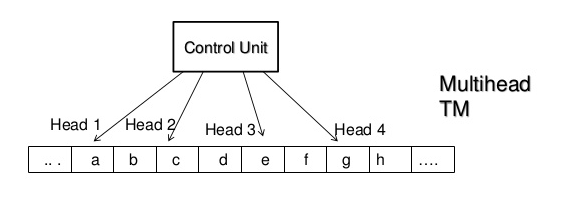


**Non-Deterministic Turing Machine**

In a Non-Deterministic Turing Machine, for every state and symbol, there are a group of actions the TM can have. So, here the transitions are not deterministic. The computation of a non-deterministic Turing Machine is a tree of configurations that can be reached from the start configuration. An input is accepted if there is at least one node of the tree which is an accept configuration, otherwise it is not accepted. If all branches of the computational tree halt on all inputs, the non-deterministic Turing Machine is called a **Decider** and if for some input, all branches are rejected, the input is also rejected.

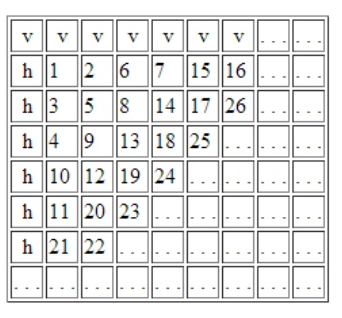
**Multi-Head Turing machine**

There are n heads, but in any state, only one head can move. This type of Turing machines is as powerful as one-tape TM. When more than one head scans a particular cell at the same time, the symbol written by one head is different from the symbol written by other head. In this situation, the change done by the head with highest priority will remain.



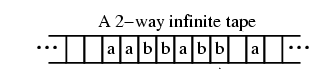
**K-dimensional TM (k=2)**

This is a kind of Turing machines that have one finite control, one read-write head and one two dimensional tape. The tape has the top end and the left end but extends indefinitely to the right and down. It is divided into rows of small squares. For any Turing machine of this type there is a Turing machine with a one dimensional tape that is equally powerful, that is, the former can be simulated by the latter.   
To simulate a two dimensional tape with a one dimensional tape, first we map the squares of the two dimensional tape to those of the one dimensional tape diagonally as shown in the following tables:



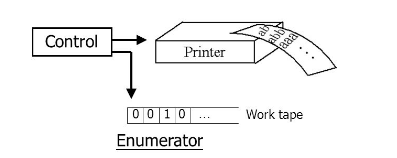
**Two-way Infinite tape**

This is a kind of Turing machines that have one finite control and one tape which extends infinitely in both directions.



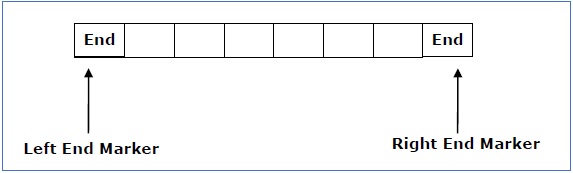
**Enumerator**

An enumerator is Turing Machine (TM) variant with as attached printer. The enumerator uses the printer as output device to print strings. An enumerator E starts with a blank input tape. Whenever the TM wants to print something, it sends the string to the printer. If the enumerator does not halt, it may print an infinite list of strings. The language of E = the set of strings that are (eventually) printed by E.



**5.9 Linear bounded Automata (LBA)**

A linear bounded automaton is a multi-track non-deterministic Turing machine with a tape of some bounded finite length. The computation is restricted to the constant bounded area. The input alphabet contains two special symbols which serve as left end markers and right end markers which mean the transitions neither move to the left of the left end marker nor to the right of the right end marker of the tape. A deterministic linear bounded automaton is always context-sensitive and the linear bounded automaton with empty language is undecidable.



A linear bounded automaton can be defined as an 8-tuple (Q, Γ, ∑, q0, <, >, δ, F) where −

* Q is a finite set of states
* Γ is the finite set of tape alphabets
* ∑ is the finite set of input alphabets
* q0 is the initial state
* < is the left end marker
* > is the right end marker
* δ is a transition function
* F is the final state

Q x Γ 🡪 Q x 2Q x Γ x {L, R}

Problem:

1. Construct linear bounded automata for the following context-sensitive language.

L = {anbncn/n≥0}

Sol: : δ(,[ ) = (, [, R )

δ (,a) = (, X, R )

δ (,a) = (, a, R )

δ (,b) = (, X, R )

δ (,b) = (, b, R )

δ (,c) = (, X, L )

δ (,X) = (, X, L )

δ (,b) = (, b, L )

δ (,a) = (, a, L )

δ (,[ ) = (, [, R )

δ (,X) = (, X, R )

δ (,X) = (, X, R )

δ (,X) = (, X, R )

δ (,] ) = (, ], H )

δ (,y) = (, Y, R )

δ (,z) = (, Z, R )

δ (,z) = (, Z, R )

δ (,B) = (, B, H )

**5.10 Turing Machine as an integer function**

An integer function is a function which takes its arguments as integers and returns the result in integer, If we consider integer addition, subtraction, multiplication, etc., then all these functions take the arguments in integer and produce the result in integer.

**5.11 Universal Turing Machine**

A Turing machine is said to be universal Turing machine if it can accept:

* The input data, and
* An algorithm (description) for computing.
* This is precisely what a general-purpose digital computer does. A digital computer accepts a program written in high level language. Thus, a general-purpose Turing machine will be called a universal Turing machine if it is powerful enough to simulate the behaviour of any digital computer, including any Turing machine itself.
* More precisely, a universal Turing machine can simulate the behaviour of an arbitrary Turing machine over any set of input symbols. Thus, it is possible to create a single machine that can be used to compute any computable sequence.
* Designing a general purpose Turing machine is a more complex task. Once the transition of Turing machine is defined, the machine is restricted to carrying out one particular type of computation.
* By modifying our basic model of a Turing machine we can design a universal Turing machine. The modified Turing machine must have a large number of states for stimulating even a simple behaviour. We modify our basic model by:
* Increase the number of read/write heads
* Increase the number of dimensions of input tape
* Adding a special purpose memory

All the above modification in the basic model of a Turing machine will almost speed up the operations of the machine can do.

**5.12 Counter Machine**

Counter Machine has the same structure as the multi-stack machine but in place of each stack is a counter. Counters hold any non-negative integer, but we can only distinguish between zero and nonzero counters. That's the move of the counter machine depends on its state, input symbol and which if any of the counters are zero.

In one move, the counter machine can a) change state b) Add or subtract 1 from any of its counters, independently. However, a counter is not allowed to become negative, so it can’t subtract from a counter that is currently 0. Every language accepted by a counter machine is recursively enumerable. Every Language accepted by one counter machine is a context free language.

**5.13 Decidable vs Undecidable:**

**Decidable:** A language is called Decidable if there is a Turing machine which accepts and halts on every input string w. Every decidable language is Turing-Acceptable. A decision problem **P** is decidable if the language **L** of all yes instances to **P** is decidable.

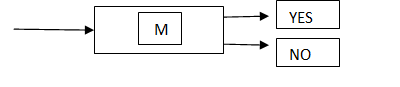
**Undecidable:** For an undecidable language, there is no Turing Machine which accepts the language and makes a decision for every input string w (TM can make decision for some input string though). A decision problem P is called “undecidable” if the language L of all yes instances to P is not decidable. Undecidable languages are not recursive languages, but sometimes, they may be recursively enumerable languages.

**5.14 Types of TM Languages**

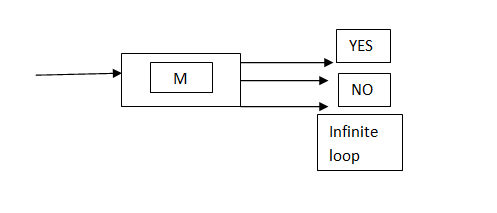
There are two types of Turing machine languages. They are Recursive languages and recursively enumerable languages.

**Recursive language:**

A formal language is recursive if there exists a total Turing Machine (a Turing Machine that halts for every given input) that, when given a finite sequence of symbols as input, accepts it if it belongs to the language and rejects it otherwise. Recursive languages are also called decidable.

****

**Recursively enumerable language:** A recursively enumerable language is a formal language for which there exists a Turing machine 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.

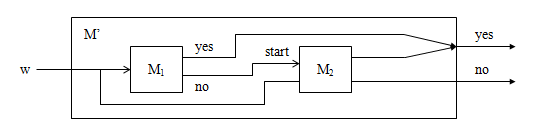


**5.15 Properties of recursive and recursively enumerable languages**

**Property 1:**

Statement: The recursive languages are closed with respect to union, i.e., if *L1* and *L2* are recursive languages, then so is L3 = L1

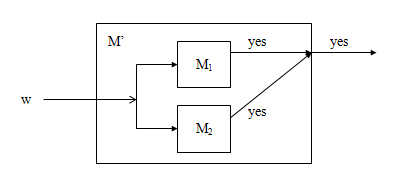
Proof: Let *M1* and *M2* be TMs such that L1 = L(M1) and L2 = L(M2) and *M1* and *M2* always halts. Construct TM *M’* as follows:



**Property 2:**

Statement: The recursive enumerable languages are closed with respect to union, i.e., if L1 and L2  are recursively enumerable languages, then so is L3 = L1

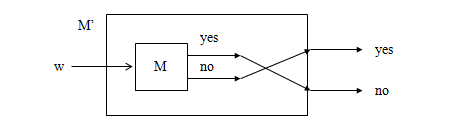
Proof: Let M1 and M2 be TMs such that L1 = L(M1) and L2 = L(M2). Construct M’ as follows:



**Property 3:**

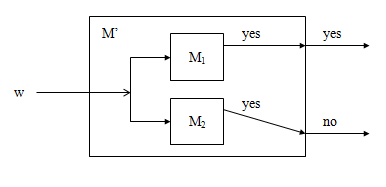
Statement: The recursive languages are closed with respect to complementation, i.e., if *L* is a recursive language, then so is

Proof: Let *M* be a TM such that L = L(M) and *M* always halts. Construct TM *M’* as follows:



**Property 4:**

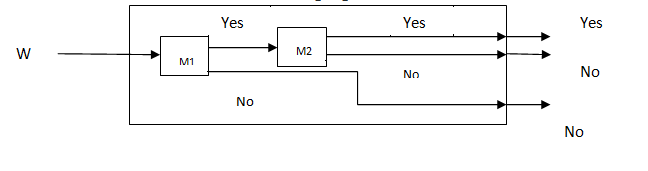
Statement: If *L* and are both recursively enumerable then *L* (and therefore ) is recursive.

Proof: Let *M1* and *M2* be TMs such that L = L(M1) and = L(M2). Construct *M’* as follows: 

**Property 5:**

Statement: Intersection of 2 recursive languages is recursive.

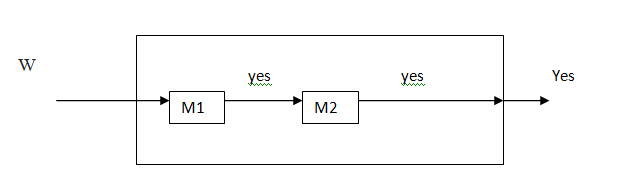
Proof: Let L1 and L2 be two recursive languages for M1 and M2, then L1L2 is recursive.

****

**Property 6:**

Statement: Intersection of two recursively enumerable languages(REL) is recursive.

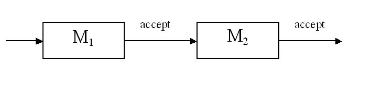
Proof: Let L1 and L2 be 2 REL for M1 and M2, then L1L2 is recursively enumerable.



**Property 7:**

Statement: Concatenation of two recursive languages is recursive.

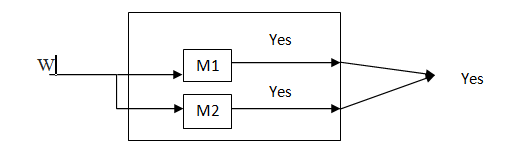
Proof: Let L1 and L2 be two recursive languages for TM’s M1 and M2. Let L1=L(M1) and L2=L(M2).



**Property 8:**

Statement: Concatenation of two RE languages is recursively enumerable

Proof: Let L1 and L2 be two RE languages, for TM’s M1 and M2.



**Property 9:**

Statement: If L is recursive language then L\* is also recursive

Proof: On input x, if x= accept. Else divide string x as w1,w2,w3............wn. If any if the wi is not accepted by M, then reject.

**Property 10:**

Statement: If L is RE language then L\* is RE.

Proof: On input x, if x= accept. Else, divide x as w1,w2,w3...........wn. Run M1 on each wiand accept if M1 accepts all. Else reject.

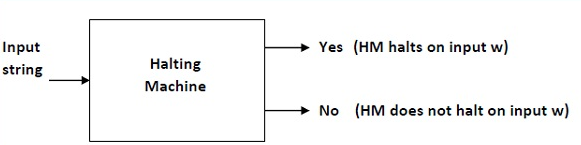
**5.16 Halting problem**

Statement:

Does the Turing machine finish computing of the string **w** in a finite number of steps? The answer must be either yes or no.

Proof:

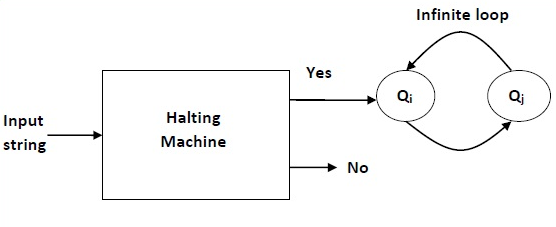
At first, we will assume that such a Turing machine exists to solve this problem and then we will show it is contradicting itself. We will call this Turing machine as a Halting machine that produces a ‘yes’ or ‘no’ in a finite amount of time. If the halting machine finishes in a finite amount of time, the output comes as ‘yes’, otherwise as ‘no’. The following is the block diagram of a Halting machine −



Now we will design an inverted halting machine (HM)’ as −

* If H returns YES, then loop forever.
* If H returns NO, then halt.

The following is the block diagram of an ‘Inverted halting machine’ –



Further, a machine (HM)2 which input itself is constructed as follows −

* If (HM)2 halts on input, loop forever.
* Else, halt.

Here, we have got a contradiction. Hence, the halting problem is undecidable.

**5.17 Post Correspondence Problem (PCP)**

The Post Correspondence Problem (PCP), introduced by Emil Post in 1946, is an undecidable decision problem. The PCP problem over an alphabet ∑ is stated as follows −

Given the following two lists, **M** and **N** of non-empty strings over ∑ −

M = (x1, x2, x3,………, xn)

N = (y1, y2, y3,………, yn)

We can say that there is a Post Correspondence Solution, if for some i1,i2,………… ik, where 1 ≤ ij ≤ n, the condition xi1 …….xik = yi1 …….yik satisfies.

**Problems:**

1. Find whether the lists M = (abb, aa, aaa) and N = (bba, aaa, aa) have a Post Correspondence Solution?

**Sol:**

|  |  |  |  |
| --- | --- | --- | --- |
|  | X1 | X2 | X3 |
| M | Abb | aa | Aaa |
| N | Bba | aaa | Aa |

Here, x2x1x3 = ‘aaabbaaa’ and y2y1y3 = ‘aaabbaaa’

We can see that x2x1x3 = y2y1y3

Hence, the solution is i = 2, j = 1, and k = 3.

1. Find whether the lists M = (ab, bab, bbaaa) and N = (a, ba, bab) have a Post Correspondence Solution?

Sol:

|  |  |  |  |
| --- | --- | --- | --- |
|  | X1 | X2 | X3 |
| M | ab | bab | Bbaaa |
| N | a | ba | Bab |

In this case, there is no solution because −

| x2x1x3 | ≠ | y2y1y3 | (Lengths are not same)

Hence, it can be said that this Post Correspondence Problem is undecidable.

1. Find whether there is a solution for the PCP problem

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| W | 0 | 011 | 0 | 011 | 1 |
| X | 10 | 00 | 11 | 0 | 11 |

Sol:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| W1 | W2 | W3 | W4 | W5 |
| 0 | 011 | 0 | 011 | 1 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| X1 | X2 | X3 | X4 | X5 |
| 10 | 00 | 11 | 0 | 11 |

Now change the positions to form the same string in both W and X.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| W4 | W3 | W2 | W5 | W1 |
| 011 | 0 | 011 | 1 | 0 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| X4 | X3 | X2 | X5 | X1 |
| 0 | 11 | 00 | 11 | 10 |

Both string of w and x are same in position 4, 3, 2, 5, 1.

**5.18 Modified PCP**

First pair in the A and B lists must be the first pair in the solution i.e., the problem is to determine if there is sequence of zero or more integers i1, i2,....im such that w1,wi1,wi2......wim = x1,xi1,xi2........xim.

Problem:

1. Find whether the lists M = (ab, bab, bbaaa) and N = (a, ba, bab) have a Post Correspondence Solution?

Sol:

|  |  |  |  |
| --- | --- | --- | --- |
|  | X1 | X2 | X3 |
| M | 10 | 110 | 11 |
| N | 10 | 11 | 011 |

This MPCP has a solution 1, 2, 3

w1,w2,w3=x1,x2,x3

10 110 11 = 10 11 011

**Reducing MPCP to PCP**

Procedure to convert MPCP to PCP

Step 1: Place a symbol \* after every symbol in w

Step 2: Place a symbol \* before every symbol in x

Step 3: Place \* before start symbol in w

Step 4: Introduce a special symbol $ and \*$ in w and x

**Problems:**

1. **Convert the following MPCP to PCP**

|  |  |  |
| --- | --- | --- |
| **i** | **wi** | **xi** |
| 1 | 1 | 111 |
| 2 | 10111 | 10 |
| 3 | 10 | 0 |

|  |  |  |
| --- | --- | --- |
| **I** | **wi** | **xi** |
| 0 | \*1\* | \*1\*1\*1 |
| 1 | 1\* | \*1\*1\*1 |
| 2 | 1\*0\*1\*1\*1\* | \*1\*0 |
| 3 | 1\*0\* | \*0 |
| 4 | $ | \*$ |

Sol:

* All languages are not Regular. The language becomes Regular if it can be represented by using FA.
* L can be Finite or Infinite Lang
* All Finite Languages can be represented by FA. Therefore all Finite Languages are Regular Languages.
* Some infinite can be represented by FA. If the infinite language can be represented by FA, then that language is Regular Language.

**REFERENCES:**

[1] Aho, A.V., Hopcroft, J.E., and Ullman, J.D. (1972). The Design and Analysis of Computer Algorithms. Addison-Wesley.

[2] Angluin, D. (1987). Learning Regular Sets from Queries and Counterexamples. Information and Computation, 75(2), 87-106.

[3] Arala, Prasad et al., Enabling Cloud Data Service Security with Key Exposure, Indian Journal of Public Health Research & Development, Vol. 10,No. 5, 2019.

[4] Bhargav K et al., An Extensive Study for the Development of Web Pages, Indian Journal of Public Health Research & Development, Vol. 10, No. 5, 2019.

[5] Chomsky, N. (1959). On Certain Formal Properties of Grammars. Information and Control, 2(2), 137-167.

[6] Garey, M.R., and Johnson, D.S. (1979). Computers and Intractability: A Guide to the Theory of NP-Completeness. W.H. Freeman & Co Ltd.

[7] Ginsburg, S., and Greibach, S.A. (1966). Deterministic Context-Free Languages. Information and Control, 9(6), 620-648.

[8] Haripriya, K., et al., "Using Social Media to Promote E-Commerce Business." International Journal of Recent Research Aspects 5.1 (2018).

[9] Hopcroft, J.E. (1969). An n log n Algorithm for Minimizing States in a Finite Automaton. Theory of Computing Systems, 3(2), 181-196.

[10] Jahnavi, Y., and Y. Radhika. "A cogitate study on text mining." International Journal of Engineering and Advanced Technology (IJEAT) 1.6 (2012): 189-196.

[11] Jahnavi, Yeturu. "Analysis of weather data using various regression algorithms." International Journal of Data Science 4.2 (2019): 117-141.

[12] Jahnavi, Y., and Y. Radhika. "FPST: a new term weighting algorithm for long running and short lived events." International Journal of Data Analysis Techniques and Strategies 7.4 (2015): 366-383.

[13] Jahnavi, Y., and Y. Radhika. "Hot topic extraction based on frequency, position, scattering and topical weight for time sliced news documents." 2013 15th International Conference on Advanced Computing Technologies (ICACT). IEEE, 2013.

[14] Jahnavi, Yeturu. "Statistical data mining technique for salient feature extraction." International Journal of Intelligent Systems Technologies and Applications 18.4 (2019): 353-376.

[15] Jahnavi, Yeturu, et al. "A new algorithm for time series prediction using machine learning models." Evolutionary Intelligence (2022): 1-12.

[16] Jahnavi, Yeturu, et al. "A novel ensemble stacking classification of genetic variations using machine learning algorithms." International Journal of Image and Graphics 23.02 (2023): 2350015.

[17] Jahnavi, Y. et al., "A Novel Processing of Scalable Web Log Data Using Map Reduce Framework.", Computer Vision and Robotics. Algorithms for Intelligent Systems. Springer, Singapore. https://doi.org/10.1007/978-981-19-7892-0\_2, (2023).

[18] Kozen, D.C. (1977). On the Equivalence of Regular Expressions with Squaring and Regular Expressions with Intersection. Information Processing Letters, 6(6), 237-240.

[19] Mutyala Lakshmi et al., Security Health monitoring and Attestation of Virtual Machines in Cloud computing,Indian Journal of Public Health Research & Development, Vol. 10, No. 5, 2019.

[20] Rabin, M.O., and Scott, D. (1959). Finite Automata and Their Decision Problems. IBM Journal of Research and Development, 3(2), 114-125.

[21] Sukanya, G., N et al., Country Location Classification on Tweets, Indian Journal of Public Health Research & Development, Vol. 10, No. 5, 2019.

[22] Valiant, L.G. (1975). General Context-Free Recognition in Less Than Cubic Time. Journal of Computer and System Sciences, 10(2), 308-315.

[23] Vijaya, U et al., "Community-Based Health Service for Lexis Gap in Online Health Seekers.", International Journal of Information and Technology (IJIT), Vol. 2, No. Mutyala Lakshmi5, 2016.

[24] Jahnavi, Y., Balasaraswathi, V.R., Nagendra Kumar, P. (2023). Model Building and Heuristic Evaluation of Various Machine Learning Classifiers. In: Pandit, M., Gaur, M.K., Kumar, S. (eds) Artificial Intelligence and Sustainable Computing. ICSISCET 2022. Algorithms for Intelligent Systems. Springer, Singapore. https://doi.org/10.1007/978-981-99-1431-9\_30.

[25] Jahnavi, Y., Kumar, P.N., Anusha, P., Prasad, M.S. (2023). Prediction and Evaluation of Cancer Using Machine Learning Techniques. In: Pandit, M., Gaur, M.K., Kumar, S. (eds) Artificial Intelligence and Sustainable Computing. ICSISCET 2022. Algorithms for Intelligent Systems. Springer, Singapore. https://doi.org/10.1007/978-981-99-1431-9\_31.