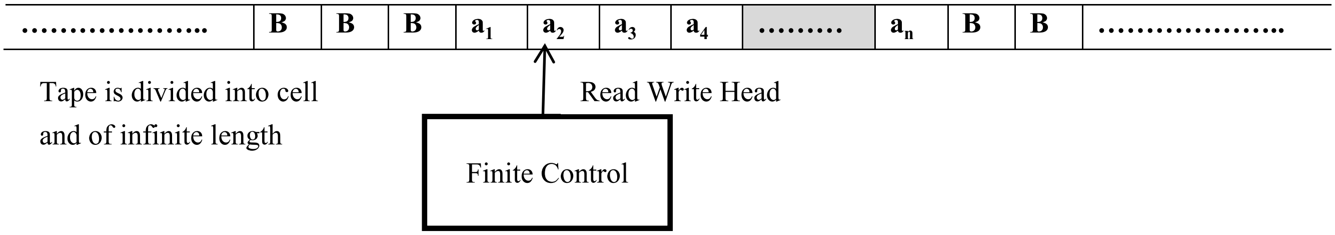
MCA-302

Formal Language and Automata



VARDHMAN MAHAVEER OPEN UNIVERSITY

KOTA

www.vmou.ac.in

MCA-302

Vardhman Mahaveer Open University, Kota

Formal Language and Automata

Course Development Committee

Chair Person

Prof. Ashok Sharma
Vice-Chancellor
Vardhman Mahaveer Open University, Kota

Prof. L.R. Gurjar
Director Academic
Vardhman Mahaveer Open University, Kota

Convener and Members

Convener

Neeraj Arora
Assistant Professor, Computer Science
School of Science and Technology,
Vardhman Mahaveer Open University, Kota.

Members

1. **Prof. (Dr.) Reena Dadich**
   Professor and Head (CS)
   University of Kota, Kota

2. **Prof. (Dr.) N.K. Joshi**
   Professor (CS) and Director, MIMT, Kota

3. **Dr. Harish Sharma**
   Associate Professor, CSE Deptt.
   Rajasthan Technical University, Kota

4. **Mr. Abhishek Nagar,**
   Programmer Officer, VMOU, Kota

5. **Dr. Anuradha Dubey**
   Deputy Director
   School of Science &Technology
   Vardhman Mahaveer Open University, Kota

Editor & Unit Writers **MCA-302: Formal Language and Automata**

Editor

Mr. Neeraj Arora

Assistant Professor and Convener, Computer Science, School of Science and Technology,Vardhman Mahaveer Open University, Kota

Unit Writers	*Units*
Prof. (Dr.) Nemi Chand Barwar Professor, Computer Science and Engineering, M.B.M. Engineering College, Jodhpur.	**10**
Dr. Madnvi Sinha Associate Professor & Head, Dept. of CS, Birla Institute of Technology, Jaipur.	**7, 9**
Dr. Sandeep Poonia Associate Professor & Head, CSE, Jagannath University, Jaipur.	**12,13**
Mr. Pankaj Acharya Associate Professor and Head, Dept. of IT, J.I.E.T., Jodhpur	**6**
Mr. Ravi Gupta Assistant Professor, Mathematics, VMOU, Kota	**1, 2**
Neeraj Arora Assistant Professor of Computer Science, Vardhman Mahaveer Open University, Kota	**5, 11, 14**
Mr. Mudit Chaturvedi Assistant Professor, Computer Science and Engineering, Jaipur National University, Jaipur	**3, 4**
Mrs. Monika Verma Assistant Professor, CSE Dept. , R.N. Modi Engineering College, Kota	**8**

Academic and Administrative Management

Prof. Ashok Sharma
Vice-Chancellor, V.M.O. University, Kota

Prof. L.R. Gurjar
Director (Academic), V.M.O. University, Kota

Dr. Shiv Kumar Mishra
Director (MP&D), V.M.O. University, Kota

Print: 2018 **ISBN: : 978-81-8496-641-1**

Printed and Published on behalf of the Registrar, Vardhman Mahaveer Open University, Kota.
Printed by :

MCA-302

Vardhman Mahaveer Open University, Kota

Formal Language and Automata

Contents

Preface

The present book entitled “Formal Language and Automata” has been designed so as to cover the unit-wise syllabus of MCA-302 course for MCA 3rd Year students of Vardhman Mahaveer Open University, Kota.

The book is begin with some mathematical preliminaries and fundamentals necessary to understand the Formal Language and Automata, end with complexity theory and application areas.

Each unit begins with objectives, introduction and principles together with illustrative and other descriptive material .The illustrative examples serve to illustrate and amplify the theory of computation. The units have been written by various experts in the field. We believe that this book is well suited to self-learning. The text is written in a logical sequence and is beneficial for students. The concise and sequential nature of the book makes it easier to learn. Although we have made all efforts to make the text error free, yet errors may remain in the text. We shall be thankful to the students and teachers alike if they point these out to us. Any further comments and suggestions for future improvement are welcome and will be most gratefully acknowledged.

UNIT-1
Mathematical Preliminaries

Structure of the Unit

1.0 Objective

Objective of this unit are

- To aware learner about concept of set used in computer.
- To aware leaner about utility of relations and function in languages.
- To aware learner basic concept of graph theory.

1.1 Introduction

Set is very useful tool of mathematics it is used in almost all branches of mathematics and computer relation and functions play important role in mathematics and computer science.

Graph theory begins with very simple geometric ideas and has powerful application in computer science.

1.2 Sets

A collection of well defined objects is called set, by well defined collection we means that there exists a rule with the help of it, is possible to determine whether a given object is a member of the given collection or not. We denote sets by using capital letters A, B. C………etc.

1.3 Method of Describing sets

There have two methods to specify a given set.

(a) **Roster Method:** A set may be described by listing all its elements. For example, the set of vowels in the English alphabet is

A={a,e,i,o,u}.

Here the elements are separated by commas and are enclosed in a pair of middle bracket {}. This method of describing a set is called the roster method or the tabular form of the set.

Example I = { ……-3, -2, -1, 0, 1, 2, 3, …….. }.

(b) **Set Builder Form:** The roster method of specifying a set is not always convenient and sometimes it is not possible to use this method to describe a set. A set can also be defined by some property which characterizes all the elements of the set. For example,

A = {x | x is a vowel in the English alphabet}

which reads "A is the set of x such that x is a vowel in the English alphabet. This method of describing a set is called set builders form.

Illustrations:

1. O = {x |x is an odd integer}
2. E = {x: x is an even integer}

Equality of Sets

Two sets A and B are said to be equal if they contain the same elements. This statement is also known as the axiom of extension. We write A = B if the sets A and B are equal and A * B if the sets A and Bare not equal. For example,

1. {a,b,c} = {b,a,c}
2. {2, 3, 5} $\neq$ {2, 3, 7}

Remark: If A = {a, a, b, c} and B = {a, b, c} then it is clear that A = B.

Thus {a, a, b, c} is a redundant representation of the set {a,b,c},. For this reason, some authors defined a set to be a well-defined collection of distinct objects.

Subsets

Let A and B be two sets. If every elements of A are also an element of B then A is called a subset of B. We also say that A is contained in B or that B contains A. In symbols we write

$'A \subseteq B' \, or \, 'B \supseteq A'$

We say A is not a subset of B if at least one elements of A does not belong to B and we write is $A \subseteq B$. It is clear that two sets A and B are equal if any only if $A \subseteq B \, and \, B \supseteq A$.

Illustration:

Consider the set A={1, 3, 4, 5}, B={1, 2, 3, 5} and C={2, 3} then $C \subseteq B \; but \, C \supseteq A$.

Proper subset of a set

From the definition of a subset it is clear that every set is subset of itself. A set B is called proper subset of A if B is a subset of A and B is not equal to A. Briefly, B is a proper subset of A if $B \subseteq A$ and B$\neq$A and write $B \subseteq A$

Empty set

The set which contains no element is called the empty set (or null set or the void set) and is denoted by {}. The Empty set is also denoted by the symbol ϕ. Since ϕ has no element, therefore, empty set is subset of every set. A set which is not empty is called non-empty set.

Disjoint sets

Two sets A and B are said to be disjoint if they have no elements in common. For example the sets A = {1, 2, 3, 4} and B = {0, 5, 6} are disjoint while the set A = {1, 2, 3, 4} and C {1, 2, 6} are no disjoint.

Singleton Set

A set which contains exactly one element is called a singleton set. For example, {2} is a singleton.

Universal Set

In any mathematical discussion, we usually consider all the sets to be sub- sets of a fixed set called the universal set. Universal set is sometimes referred to as the universe or the universe of discourse.

For example, in studying human population the universal set consists of all human in the world and while discussing plane geometry we may consider the universal set to be the set consisting of all the points in the plane.

Power Set ~

If X is any set then the set of all subsets of X is called the power set of X, denoted by P(X). Thus $P(X)=\{A:A\subseteq X\}$

If X contains n elements then P(X) contains 2" elements.

Example. Let X = {1, 2, 3} then

P(X)= { ϕ, {1}, {2}, {3}, {1,2}, {1,3}, {2,3}, {1,2, 3}}.

Since X has three elements, P(X) has $2^3 = 8$ elements.

Finite and Infinite Set

A set is said to be finite if it contains e finite number of distinct elements. A set is said to be infinite if it is not finite.

Example: Let A = {I, 3, 5. 7, 9}. Then A is finite because it contains 5 distinct elements.

Example: Let B = {I, 2, 3,4, ... }. Then B is an infinite set.

1.4 Operations on Sets

We introduce and study some basic operations in the section. Using these operations, we can construct new sets by the elements of given sets.

Union of Sets

Let A and B be two sets. The union of A and B is the set of all elements which are in the set A or in the set B. The union of two sets A and B is denoted by the symbol AUB which is read as 'A union B'. Symbolically,

$A \cup B = \{x \quad x \in \; A \text{ or } x \in B\}$

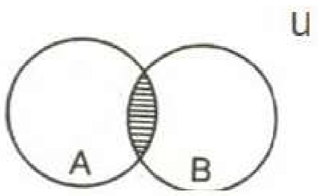


Fig.: A ∪B is shaded

In the adjoining Venn-diagram, the union of A and B is shown by the shaded area.

Intersection of Sets

Let A and B be two sets. The intersection of A and B is the set of all elements which are both in A and B. We denote the intersection of A and B by $A \cap B$, which is read as 'A intersection B'. Symbolically,

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

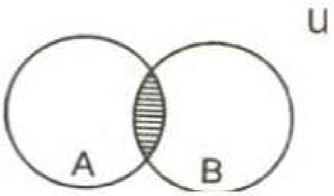


Fig A ∩ B is shaded

In the adjoining venn diagram the intersection of two sets A and B is shown by the shaded region.

Complement of a set.

Let A be a subset of a universal set U. The set of all those elements of U which are not in A is called the complement of A and is denoted by U - . A or simply A'. Symbolically,

$A'=U-A= \{X:X \in U \text{ and } x \in A\}$

Difference of Sets

The difference A-B of two sets A and B is the set of elements which belong to A but which do not belong to B. Thus,

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

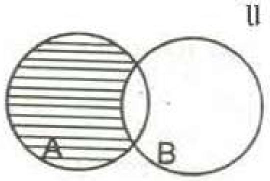


Fig. 3 A-B is shaded

The shaded region represents A-B in the adjoining diagram.

Illustration:

Let U= {a, b, c, d, e,f}

A = {a,b,c,d} and B= {b,d,f}.

Then A-B = {a,c}

B-A = {f}

AUB = {a, b, c, d,!}

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

U-A = {e,f}

1.5 Some results on number of elements of a set

If A, B and C be any sets then

$(i)\ n(A\bigcup B)=n(A)+n(B)-n(A\bigcap B)$

$(ii)\ n(A\bigcap B)=n(A)+n(B)-n(A\bigcup B)$

$(ii)\ if\ A\,and\,B\,aredisjont\,then$

$ifA\bigcap B=\varphi\Rightarrow n(A\bigcap B)=0then$

$n(A\bigcup B)=n(A)+n(B)$

$(iv)\,n(A-B)=n(A)-n(A\bigcap B)$

$(v)\,n(B-A)=n(B)-n(A\bigcap B)$

$(vi)\,n(A\Delta B)=n(A-B)+n(B-A)$

$=n(A)-n(A\bigcap B)+n(B)-n(B-A)$

$=n(A)+n(B)-2n(A\bigcap B)$

$(vii)\,n(A'\bigcup B')=n(\mathrm{A}\bigcap B)'$

$=n(\bigcup)-n(A\bigcap B)$

$=n(\bigcup)-n(A)-n(B)+n(A\bigcup B)$

$(viii)\,n(A'\bigcap B')=n(A\bigcup B)'$

$=n(\bigcup)-n(A\bigcup B)$

$=n(\bigcup)-n(A)-n(B)+n(A\bigcap B)$

$(ix)\quad n(A\bigcup B\bigcup C)=n(A)+n(B)+n(C)-n(A\bigcap B)$

$-n(A\bigcap C)-n(B\bigcap C)+n(A\bigcap B\bigcap C)$

Example 1.1 In a group of 900 people 550 speak Hindi and 450 can speak English. How many can speak both Hindi and English,

Solution : Let H denote the set of people speaking Hindi and E denote the set of people speaking English then we have

$n(H)=550\,and\,n(E)=450$

Now we have to find number of people who can speak both Hindi and English i.e.

$n(H\bigcap E)=100$ and

$(H\bigcup E)=n(H)+n(E)-n(H\bigcap E)$

=550 + 450 – 900

$n(H\cap E)=100$

So 100 persons can speak Hindi and English.

Example 1.2 A survey shows that 63% of Indians like cheese where 76% like apples. If X% of Indian like both cheese and apples find the value of x.

Solution : Let the population of India is 1000 and A denotes the set of Indians who like cheese and B denote the set of Indian who like apples then

$n(A)=63, n(B)=76$

So $n(A\cup B)=n(A)+\cap(B)-n(A\cap B)$

$n(A)=63, n(B)=76$

$\Rightarrow n(A\cap B)+n(A\cap B)=63+76=139$

$\Rightarrow n(A\cap B)+139-n(A\cup B)$ .....(1)

$But\ n(A\cap B)\leq 100\, So$

$-n(A\cup B)\geq 100$

$\Rightarrow 139-n(A\cup B)\geq 39$

$\Rightarrow n(A\cap B)=\geq 39$ (Using (1) ....(2)

Now since and $A\cap B\subset A\ and\ A\cap B\subset B$

$\Rightarrow n(A\cap B)\leq n(A)\, and\, n(A\cap B)\leq n(B)$

$\Rightarrow n(A\cap B)\leq 63\, and\, n(A\cap B)\leq 76$

$\Rightarrow n(A\cap B)\leq 63$

$So\ by\ (1)\ and\ (2)$

$39\leq n(A\cap B)\leq 63$

$\Rightarrow 39\leq x\leq 63$

$So\ by\ (1)\ and\ (2)$

$39\leq n(A\cap B)\leq 63$

$\Rightarrow 39\leq x\leq 63$

Example 1.3 In a village of 1000 families it was found the 40% families have agriculture profession. 20% families have milk product profession and 10% families have other profession. If 5% families have both agriculture and mil product profession 3% have milk product and other profession and 4% have agriculture and other profession and 2% families have all these profession find the number of family which have

(i) Only agriculture profession

(ii) Only milk product profession

(iii) No profession

Solution : Let A is the set of families having Agriculture profession B is the set of families having milk product profession and C is the set of families having other profession.

Then n(A)= 40% of 1000 = 400

$A(B) = 20\% of\, 1000 = 200$

$A(C) = 10\% of\, 1000 = 100$

$n(A \cap B) = 5\% of\, 1000 = 50$

$n(A \cap C) = 3\% of\, 1000 = 30$

$n(C \cap A) = 4\% of\, 1000 = 40$

$n(A \cap B \cap C) = 2\% of\, 1000 = 20$

1. Agriculture profession only

i.e. $n(A \cap B' \cap C') = n(A \cap B' \cap C'))$

$n(A) - n) - n\{(A \cap B)(A \cap C)\}$

$n(A) - n(A \cap B) - n(A \cap C) + n(A \cap B) \cap (A \cap C)$

$n(A) - n(A \cap B) - n(A \cap C) + n(A \cap B \cap C)$

$= 400 - 50 - 40 + 20 = 330$

(ii)Required number $n(A \cap (B' \cap C')$ and proceed as above

$n(A' \cap B' \cap C') = 140$

(iii)Required number

$n(A) - n(A' \cup B' \cup C') = ((A \cup B \cup C'))$

$= n(\cup) - n(A) + n(B) + n(C) - n(A \cap B) - n(B \cap C)$

$= n(C \cap A) + n(A \cap B \cap C)$

1000-{400+200+100-50-30-40+20}

=40

Example 1.4 (De-Morgen's Low) : If A and B be any two set then

$(a)(A\bigcup B)'=A'\bigcap B' \quad (b)\ (A\bigcap B)'=A'\bigcup B'$

Proof : (a) Let $x\in(A\bigcup B)'$

$\Rightarrow x\notin A\bigcup B$

$\Rightarrow x\notin A\, and\, x\notin B$

$\Rightarrow x\notin A'\, and\, x\notin B'$

$\Rightarrow x\in A'\, and\, x\in B'$

$\Rightarrow x\in A'\bigcap B'$

$So(A\bigcup B)'=A'\bigcap B'$

Again let

$x\in A\bigcap B'$

$\Rightarrow x\in A'\, and\, x\notin B'$

$\Rightarrow x\in A\, and\, x\in B$

$\Rightarrow x\notin A\bigcup B$

$\Rightarrow x\notin (A\bigcup B)'$

$\Rightarrow A'\bigcap B'\subset(A\bigcap B)'$

So by (1) and (2)

$(A\bigcap B)'=A'\bigcup B'$

(b) Let $x\in(A\bigcap B)'$

$\Rightarrow x\notin (A\bigcap B)'$

$\Rightarrow x\notin A\ or\ x\notin B$

$\Rightarrow x\in A'or\ x\notin B'$

$\Rightarrow x\in A'\bigcup B'$

$\Rightarrow \quad (A\bigcap B)'\subset A'\bigcup B$

Again let $x\in A'B'$

$\Rightarrow x\notin A\ or\ x\notin B$

$\Rightarrow x\in A'or\ x\notin B'$

$\Rightarrow x\in A'\bigcup B'$

$\Rightarrow(A'\bigcup B')\subset(A\bigcup B)'$

So by (1) and (2)

$(A'\bigcup B') = A'\bigcup B'$

Example 1.5 : Distributive Law : If A, B and C are any sets then

$(a) A\bigcup(B\bigcap C)=(A\bigcup B)\bigcap(A\bigcup C)$

$(b) A\bigcup(B\bigcup C)=(A\bigcap B)\bigcup(A\bigcap C)$

Proof : Let $x\in A\bigcup(B\bigcup C)$

$\Leftrightarrow x\in A\ or\ x\in(B\bigcap C)$

$\Leftrightarrow x\in A\ or\ (x\in B\ \&\ x\in C)$

$\Leftrightarrow (x\in A\ or\ x\in B)\ and\ x\in A\ or\ x\in C)$

$\Leftrightarrow x\in(A\bigcup B)\ and\ x\in(A\bigcup C)$

$\Leftrightarrow x\in(A\bigcup B)\bigcap(A\bigcup C)$

So $A\bigcup(B\bigcap C)=(A\bigcup B)\bigcap(A\bigcup C)$

(b) $A\bigcap(B\bigcup C)=(A\bigcap B)\bigcup(A\bigcap C)$

Let $x\in A\bigcap(B\bigcup C)$

$\Leftrightarrow x\in A\ and\ x\in(B\bigcup C)$

$\Leftrightarrow x\in A\ or\ (x\in B\ \&\ x\in C)$

$\Leftrightarrow (x\in A\ \&\ x\in B)\ or (x\in A\ \&\ x\in C)$

$\Leftrightarrow x\in(A\bigcap B)\ or (x\in A\bigcap C)$

$\Leftrightarrow x\in(A\bigcap B)\bigcup(A\bigcup C)$

So $A\bigcap(B\bigcup C)=(A\bigcap B)\bigcup(A\bigcap C)$

Some Important results :

(i) $A-B=A\bigcap B'$

(ii) $(A-B)\bigcap B=\phi$

(iii) $(A-B)\bigcup A=A$

(iv) $(A-B)\bigcup B=A\bigcup B$

(v) If A and B are disjoint A-B = A and B-A = B

1.6 Self Learning Exercise

1. For any two sets A and B prove the following

$(a) A \bigcap (A \cup B) = A \bigcap B$

$(b) A \bigcap (A \cup B)' = \phi$

$(c) A - (A - B) = A \bigcap B$

$(d) A \Delta (A \bigcap B) = A - B$

2. Prove that

$(a)(A - B) \cup A = A$

$(b)(A - B) \bigcap A = A \bigcap B' = A - B$

$(c)(A - B) \cup B = A \cup B$

$(d)(A - B) \bigcap B = \phi$

3. A and B be two sets containing 2 and 4 element respectively, what can be minimum number of elements in $(A \cup B)$? Also find the maximum numbers of elements in $(A \cup B)$.

4. If A, B and C are three sets and U is their universal set such that $n(U)$ =800, n(A)=250, n(B)=300, $n(A \cap B)$=150 find $n(A' \cap B')$.

5. In a Class of 50 students 20 have taken mathematics, 15 have taken mathematics but not chemistry. Find the number of students who have taken both mathematics and chemistry and number of students who have taken chemistry but both mathematics. If it is given that each student has taken either mathematics or chemistry or both.

6. Members of three athletic team in a certain college 21 are in basketball team 26 in hockey team and 29 in the football them, 14 football and basketball and 8 play hockey and football, 12 play football and basketball and 8 play all the three games. How many members are there in all?

Answers

3. Maximum Number of elements = 6

Maximum Number of elements = 4

4. 400

5. 5 students take both mathematics and chemistry 30 students take chemistry not mathematics

6 . 63

1.7 Relation

If A and B be any two sets then a relation R from A to B in a subset of $A \times B$. So we can define R as a subset of $A \times B$ i.e. $R \subseteq A \times B$.

If R is a Relation from A to B and if (x, y) $\in$ R, and then we write it as x R y and read 'x is related to y be relation R, and if (x, y) $\notin$ R then we read it as x is not related o y and write x R y.

Example 1.6 : if A = {1, 2, 3, 4} and B= {a, b, c, d}

(i) R_1 = {(1,a), (2,b), (3,c)}

(ii) R_2 = {(1,a), (1,b), (4,a), (3,c)}

(iii) R_3 = {(1,a), (a,1), (a,2) (c, 3)}

(iv) R_4 = {(5,a), (2,b), (3,c)}

Then which of the above is a relation from $A \times B$

Solution :

(i) Since $R_1 \subset A \times B\, so\, R_1$ is relation

(ii) Since $R_2 \subset A \times B\, so\, R_2$ is a relation

(iii) Since $R_3 \not\subset A \times B\, so\, R_3$ is not relation

(iv) Since $R_4 \subset A \times B\, so\, R_4$ is a relation

Number of Relations : If $R \subset A \times B$ is any relation and A have m elements and B has n elements the number of relations from A to B is 2^{mn} and number of non void relation is $2^{mn} - 1$

Domain of Range of Relation if R is a relation defined from A to B then domain or R is the set $\{\mathbf{a} \mid \mathbf{a} \in A\}$ i.e. set of all first components of order pair belonging to R. and Range of R is the set $\{b \mid b \in A\}$ i.e. set of all second component or order pair belong to R.

Example 1.7 : If A = {1, 2, 3, 4, 5, 6, 7, 8} and B = {3, 4, 5, 6, 7, 8, 9, 10, 11, 12} and R is defined as R = {(1,3), (2, 5), (3, 7), (4, 9), (5, 11), } then find domain and **Solution** :

Domain or R $= \{a \mid a \in A, (a,b) \in R\}$

= {1, 2, 3, 4, 5}

Range of R $= \{b \mid b \in A, (a,b) \in R\}$

= {3, 5, 7, 9, 11}

Example 1.8: if A = {1, 2, 3, 4, 5} and B ={1, 3, 5} and a Relation R from A to B is defined as $(a,b) \in R \Leftrightarrow a \leq b$ then find the domain and Range of R.

Solution : Clearly R= {(1, 1), (1, 3), (1, 5), (2, 3), (2, 5), (3, 3), (3, 5), (4, 5), (5, 5)}

So Domain of R = {1, 2, 3, 4, 5}

And Range of R= {1, 3, 5}

Relation on a set : if A is any non-void set and a relation R defined from A to A then R is called relation on set A.

Inverse Relation : If R is a relation from set A to a set B then inverse relation of R is denoted by R^{-1} from set B to A and defined as

$R^{-1} = \{(b, a) \mid (a, b) \in R\}$

Thus domain or R^{-1} = Range of R

And Range of R^{-1} = domain of R.

Example 1.9: If R is defined from set A to set B as R = {(2, 4) (4,8), (6,12)} then

R^{-1}={(4, 2) (8, 4), (12, 6)}

domain or R^{-1} = Range of R={4, 8, 12}

Range of R^{-1} = domain of R ={2, 4, 6}

1.8 Type of Relation

(1) Reflexive Relation: If in a Relation R defined on set A, if every element of A is related to itself then it is called Reflexive Relation i.e. If in a relation R defined on set A $(x,x)\in R\,\forall x\in A$ then R is called Reflexive Relation.

Example 1.10: If A = {1, 2, 3, 4} and Identity Relation defined on it as R =. {(1, 1), (2, 2), (3,3), (4, 4)} then it is a Reflexive Relation. Since $\forall x\in A(x,x)\in R$

So R is Reflexive Relation.

Note:- Every Identity Relation is a Reflexive Relation but converse not true for example if R is Relation defined on A by R = {(1, 1), (1, 2), (2, 2), (3, 3), (3, 4), (4, 4)} then R is Reflexive but R is not a Identity Relation.

Example 1.11: In the plane Relation R between two lines $l_1 \& l_2$ defined by $l_1 R l_2 \Leftrightarrow l \sqcup l_2\,(l_1\, is\, parallel\, to\, l_2)$ then R is Reflexive since each line is parallel to itself.

Example1. 12: Relation R defined on set of Natural numbers as $(a,b)\in R \Leftrightarrow a > b$ is not Reflexive since it is not possible that a > a but if we defined R by $(a,b)\in R \Leftrightarrow a\geq b$ then R is Reflexive since $a\geq a\,\forall a\in N$.

(2) Symmetric Relation: A Relation R defined on a set A is called symmetric Relation if $(a,b)\in R \Leftrightarrow (b,a)\in R$

Example 1.13: If A = {1, 2, 3} and R is defined on A as R -= {(1, 1), (1, 2), (2, 1), (2,3), (3, 2)} then R is symmetric since $\forall (a,b)\in R \Leftrightarrow (b,a)\in R$.

Example 1.14: In the plane Relation R between two lines $l_1 \& l_2$ defined by $l_1 R l_2 \Leftrightarrow l_1 \perp l_2$ $(l_1\, is\, Perpendicular\ to\, l_2)$ then R is symmetric since if $l_1 \perp l_2\ then\, l_2 \perp l_1$.

Example 1.15: Relation R defined on set of Natural numbers as $(a,b)\in R \Leftrightarrow a\leq b$ then R is not symmetric since $a\leq b$ then b is not less than or equal to a·

(3) **Transitive** - A Relation R defined on a set A is called Transitive if for every pair $(\mathrm{a, b})\in \mathrm{R}$ and $(\mathrm{b, c})\in \mathrm{R} \Rightarrow (\mathrm{a, c})\in \mathrm{R}$.

Example 1.16: If A = {1,2,3} and R is defined on A as R = {(1, 1), (1. 2), (1, 3), (2, 2), (2, 3), (3, 1), (3, 2), (3, 3)} then R is Transitive since for every pair

$(\mathrm{a, b})\in \mathrm{R}$ and $(\mathrm{b, c})\in \mathrm{R} \Rightarrow (\mathrm{a, c})\in \mathrm{R}$

Example 1.17: In plane Relation R between two triangles $\Delta_1 \& \Delta_2$ defined as $\Delta_1 R \Delta_2 \Leftrightarrow \Delta_1$ is similar to Δ_2 then R is Transitive since for $\Delta_1 R\Delta_2$ & $\Delta_2 R \Delta_3$ i.e. of Triangle Δ_1 similar to Δ_2 and Triangle Δ_2 similar to Δ_3 then Δ_1 also similar to Δ_3,

Example 1.18: Relation R defined on set of lines in a space as $l_1 R l_2 \Leftrightarrow l_1 \perp l_2$ then R is not Transitive since if $l_1 \perp l_2 \; and \; l_2 \perp l_3$ than it is not necessary that $l_1 \perp l_3$ (see figure 1 and 2)

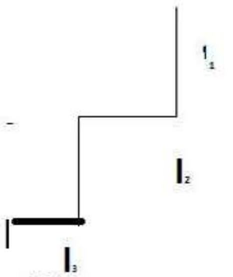


fig.1

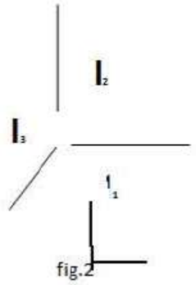


fig.2

- In figure (1) (2 $l_1 \perp l_2 \; and \; l_2 \perp l_3$ but first figure shows that line l_1 is not perpendicular to l_2 and second shows that $l_1 \perp l_3$ it is not necessary that in every case $l_1 \perp l_3$ R is not Transitive.

(4) **Anti-Symmetric Relation :** A Relation R defined on set A is anti symmetric if $(\mathrm{a, b})\in \mathrm{R} \& (\mathrm{b, a})\in \mathrm{R} \Rightarrow \mathrm{a}=\mathrm{b}$

Example 1.19: Relation R is defined on set of Rational numbers as $(x, y) \in R \Leftrightarrow x$ divides y then R is anti symmetric since $(x, y) \in R \Leftrightarrow x$ divides y and $(y, x) \in R \Leftrightarrow$ y divides x if and only if x = y.

Example 1.20: Relation R is defined on N as $(x, y) \in R \Leftrightarrow x \geq y$ then R is anti symmetric since $x \geq y \,\&\, y \geq x$ if and only if x = y.

Example 21: The Identity Relation on a set A is anti Symmetric Relation.

(5) **Equivalence Relation :** A Relation R defined on set A in equivalence Relation if

(i) R is Reflexive

(ii) R is Symmetric

(iii) R is Transitive

Example 1.22: In the plane, Relation R is defined as $l_1 R l_2 \Leftrightarrow l_1 II\, l_2$ is equivalence Relation. Since it is symmetric, Reflexive and Transitive.

Example 1.23: In the set of Triangles R is defined as $\Delta_1 R \Delta_2 \Leftrightarrow \Delta_1$ similar to Δ_2 then it is equivalence Relation since it is Reflexive, symmetric and Transitive.

Example 1.24: The Relation R is defined on set of Natural number as $(a, b) \in R \Leftrightarrow a \geq b$ is not equivalence Relation since it is Reflexive and Transitive but not symmetric.

(6) **Partial Order Relation:-** A Relation R defined on the set A is call partial order Relation if

(i) R is Reflexive

(ii) R is Anti Symmetric

(iii) R is Transitive

Example 1.25: The Relation R defined on any non-void set A as $(a, b) \in R \Leftrightarrow a \geq b$ is partial order Relation since it is Reflexive, Anti symmetric and Transitive.

Example 1.26: The relation R defined on set of Natural numbers as

$(a, b) \in R \Leftrightarrow \frac{a}{b}$ is a positive integer then it is a partial order Relation since it is reflexive. Transitive and Anti symmetric.

Example 1.28: The Relation R defined on set of Integers as

$(x, y) \in R \Leftrightarrow |x| \geq y$ then R is partial order relation since R is reflexive anti symmetric and transitive.

Example 1.29: Let a Relation R_1 on the set R of all real numbers be defined as $(a, b) \in R_1 \Leftrightarrow 1 + ab > 0 \forall a, b \in R$ show that R_1 is Reflexive and symmetric.

(1) Since $1 + a.a = 1 + a^2 \geq 0 \Rightarrow (a, a) \in R_1 \forall a \in R$ So $R_1$is reflexive.

(2) $(a, b) \in R_1 \Rightarrow 1 + a.b \geq 0$
$\Rightarrow 1 + b.a \geq 0 \Rightarrow (b, a) \in R_1$

So R_1 is Symmetric.

Example 30: Show that if R is an equivalence Relation then R^{-1} also equivalence Relation.

Solution : Let R be a relation on A

So, $R \subset A \times A \Rightarrow R^{-1} \subset A \times A$

So R^{-1} is also a relation on A

Now R^{-1} is equivalence if it is

(i) Reflexive : since R is reflexive so (a,a) $\in R \forall a \in A$

$\Rightarrow (a, a) \in R^{-1} \forall a \in A$

$\Rightarrow$ R^{-1} is reflexive.

(ii) Symmetric: Let $(a, b) \in R^{-1}$

$\Rightarrow (b, a) \in R$

$\Rightarrow (a, b) \in R$(Since R is Symmetric)

$\Rightarrow (b, a) \in R^{-1}$

R^{-1} *is symmetric*

(iii) Transitive: Let and $(a,b)\in R^{-1}$ and $(b,c)\in R^{-1}$

$\Rightarrow (b,a)\in R\,\&\,(c,b)\in R$

$\Rightarrow (c,b)\in R\,\&\,(b,a)\in R$

$\Rightarrow (c,a)\in R$

$\Rightarrow (a,c)\in R^{-1}$

$So\ (a,b)\in R^{-1}(b,c)\in R^{-1}$

$\Rightarrow (a,c)\in R^{-1}\ so\ R^{-1}\ Transitive$

Example 1.31: If R is Relation on $N\times N$ defined $(a,b)R(c,d)\Rightarrow ab=bc\forall(a,b)R(c,d)\in N\times N$ then prove that R is equivalence relation.

Solution: (i) Reflexive: Let $(a,b)\in R$ then

$(a,b)R(a,b)\Rightarrow ab=ba$ which is true

So R is reflexive.

(ii) Symmetric: Let $(a,b)R(c,d)\in N\times N$ and (a , b)R(c, d)

$\Leftrightarrow ad=bc$

$\Leftrightarrow bc=ad$

$\Leftrightarrow cd=da$

$\Leftrightarrow (c,d)R(a,b)$

$So\ R\ is\ symmetric$

(iii)Transitive: Let $(a,b)(c,d),(e.f)\in N\times N$

$(a,b)\,\mathrm{R}\,(c,d)\Leftrightarrow ad=bc$

$and \quad (c,d)R(e,f)\Leftrightarrow cf=de$

Multiplying (1) and (2)

$(ad)(ef)=(bc)(de)$

$\Rightarrow af=bd$

$\Rightarrow (a,b)R(e,f)$

$\Rightarrow R\ is\ Transitive$

$So\ R\ is\ a\ equivalence\ relation.$

Example 1.32: If R and S are any two relation on set A then prove that

(i) If R and S are symmetric then $R\bigcup S$ is also symmetric.

(ii) R is any relation and S is Reflexive then $R \bigcup S$ is also Reflexive

Solution : Since R and S are relation on set A so

$R \subset A \times A \& S \subset A \times A$

$\Rightarrow R \bigcup S \subset A \times A$

So $R \bigcup S$ is also a relation on A.

(i) Let $(a,b) \in (R \bigcup S)$

$\Rightarrow (a,b) \in R \; or \; (a,b) \in S$

$\Rightarrow (b,a) \in R \; or \; (b,a) \in S \, (Since \, R \, amd \, S \, are \, Symmetric)$

$\Rightarrow (b,a) \in (R \bigcup S)$

$\Rightarrow R \bigcup S \; is \; Symmetric$

(ii) Since S is Reflexive

So $(a,a) \in S \, \forall a \in A \; and R$ is any Relation so $S \subset R \bigcup S$

$(a,a) \in R \bigcup S \, \forall a \in A$

$\Rightarrow R \bigcup S$ is Reflexive

1.9 Function

Let A and B be two non-empty sets then a function f from set A to B is a relation which relates elements of A to B as

(i) All element of set A are related to elements of B.

(ii) An element of set A us related to a unique element of set B.

We write it as $f : A \rightarrow B$

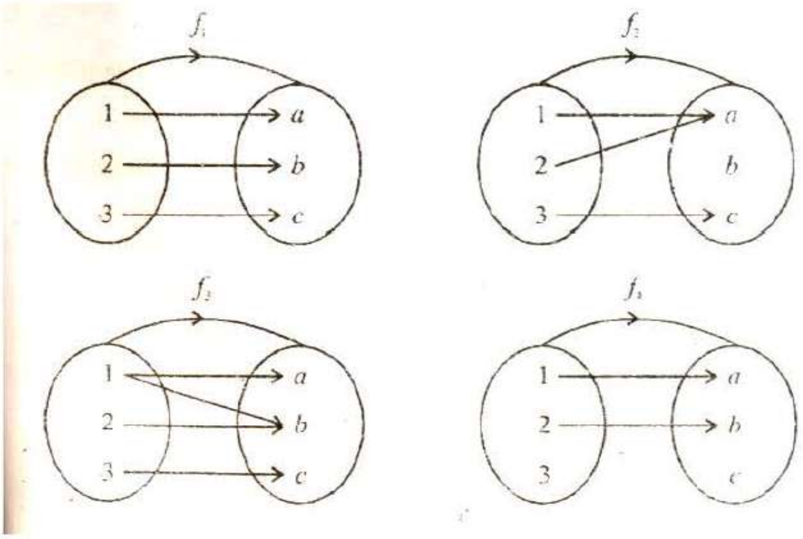
Example 1.33: if A = {1, 2, 3} B {a, b, c} then

$(i) \; f_1 = \{(1,a),(2,b)(3,c)\}$

$(ii) \; f_2 = \{(1,a),(2,a),(3,c)\}$

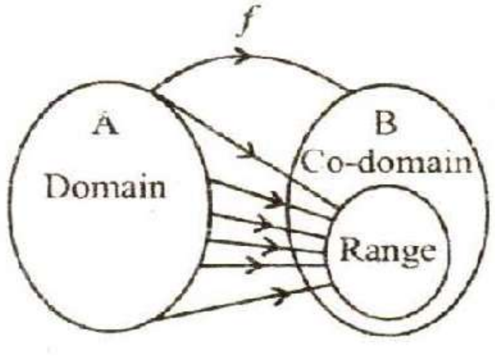
$(iii) \; f_3 = \{(1,a),(1,b),(2,b),(3,c)\}$

$(iv) \; f_4 = \{(1,a),(2,b)\}$



Clearly f_1 & f_2 are function but f_3 & f_4 are not function since f_3 relates $1 \in A$ to a and $b \in B$ so condition (ii) doesn't follow while f_4 doesn't related $3 \in A$ to any elements of B. so condition (i) doesn't follow.

Domain Co-domain and Range of a function: If a function f defined from A to B i.e. $f: B \to B$ then set A is called domain of f, set B is called co-domain of f and f-image set of elements of A is called range of f.



Example 1.34: If A = {-1,0,1} and B= {0, 1, 2, 3} and a function f defined from A o B as $f(x) = x^2$ then domain of f is A= {-1,0,1}, Co-domain is B and range of f is {0, 1}

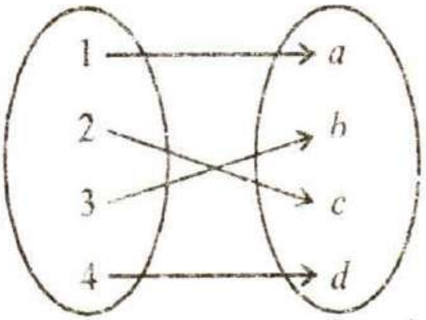
Kinds of Function

(i) **One-one function (injection)**: A function $f:A \rightarrow B$ is said to be one-one function if different elements of A has different images in B. if f is not one-one then it is called many one. Thus $f:A \rightarrow B$ is one one

$\Leftrightarrow a \neq b \Rightarrow f(a) \neq f(b) \forall a, b \notin A$

$of\ f(a) = f(b) \Rightarrow a = b \forall a, b \notin A$

Example 1.35: If A = {1, 2, 3, 4}, B = (a, b, c, d} then he function defined as f = {(1, a), (2, c), (b, b), (4, d)} is one-one function since each element of A is related to a unique element of B.



(ii) **On-to function (surjection) :** A function $f:A \rightarrow B$ is said to be on to if every element of B has a pre f image in A i.e. if range of f is B i.e. f(A) = B

(iii) **In-to Function** : A function which is not on-to is called in-to function

Example : If N = {1,2,3…} and E = {2,4,6….} and $f:N \rightarrow E$ defined as f(x)=2x then f(x) is onto function since each element of E has a pre f – image in N.

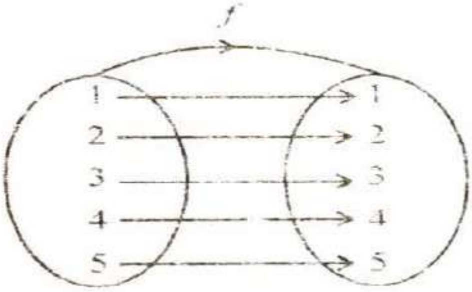
(iv) **One-one onto function (Bijection) :** A function which is one-one as well as onto is called one-one onto function i.e. A function is one – one onto

(i) If different elements of A has different image in B

(ii) Each element of B has a pre f – image in A

Example 1.36:

The identity function on set A= {1,2,3,4,5} is one – one onto since



(3) If N = {1,2,3…..} and E = {2,4,6….} then $f: N \to E$ defined as f(x) = 2x is one-one onto function

One-one: Let f(x) = f(y)

$\Rightarrow$ 2x = 2y

$\Rightarrow$ x = y

Onto – since for each x $\in$ E there have on y $\in$ N such that y = 2x.

So f is onto function.

1.10 Graphs

A graph G is a pair (V, E), where V = {v_1, v_2, ... } is a non- empty set whose elements are called vertices (or nodes) and E = (e_1, e_2, .. } is a set such that each element e_k of E is identified with an unordered pair (v_i v_j) of vertices. The elements of E are called edges of G. The vertices v_i and v_j associated with edge e_k are called the end vertices of e_k and the edge e_k is then denoted as e_k = (v_i v_j)

Graphs are usually represented by diagrams in which a vertex is represented by a dot or a small circle and an edge is represented by a line segment joining its end vertices.

Example1. 37: $V = \{v_1, v_{2,} v_{3,} v_4\}$ and $E = \{e_1, e_{2,} e_{3,} e_4, e_5\}$ be such that $e_1 = (v_1, v_{2,}), e_2 = (v_{3,}\ v_4), e_3 = (v_{1,}\ v_3), e_4 = (v_{2,}\ v_4)$ and $e_5 = (v_1, v_{2,})$ then G = (V, E) is a graph.

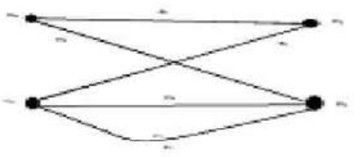


fig: 1.1

Parallel edges in a graph

We observe from the definition of a graph G = (V, E) that while the elements of E are distinct, more than one edge in E may have the same pair of end vertices. All edges having the same pair of end vertices are called parallel edges.

Simple graph

A graph that has neither self-loops nor parallel edges is called a simple graph.

Incidence and Degree

Let e_k be an edge joining two vertices v_i and v_j of a graph G: Then the edge e_k is said to be incident on each of its end vertices v_i and v_j .Two vertices in a graph are said to be adjacent if there exists an edge joining the vertices. Two non-parallel edges are said to be adjacent if they are incident on a common vertex. The degree of a vertex v in a graph G, written as d(v),is equal to the number of edges which are incident on v with self-loop counted twice,

Finite and Infinite Graphs

A graph G = (V, E) is said to be finite if both sets V and E are finite; otherwise it is called infinite graph. The graphs in Figs. 1.1 and 1.2 are finite. **Isolated Vertex, Isolated Vertex, Pendant Vertex and Null Graphs**

A vertex v is said to be isolated vertex if degree d(v) of v is zero in other words, A vertex having no edge incident on it is called an isolated. For example, the vertex v_5 in the graph of Fig. 1.3 is isolated vertex. A vertex is said to be pendant vertex if its degree is one. Vertex v_4 in the graph of Fig. 1.3 is a pendant vertex.

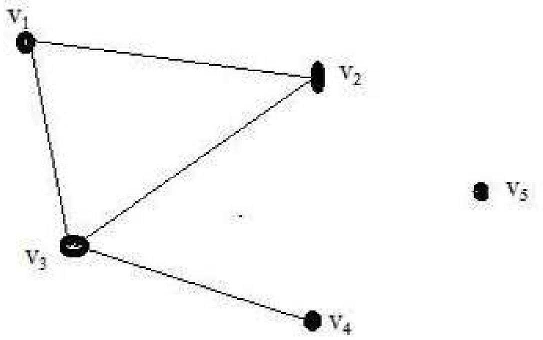


Fig: 1.3

A graph G = (V, E) is said to be null graph if the set of vertices V is non-empty but the set of edges E is empty. A null graph is thus a graph in which every vertex is an isolated vertex. A null graph of five vertices is shown in Fig 1.4. A null graph having one vertex only is called trivial graph.

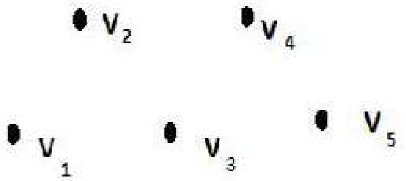


Fig: 1.4 Null graph of five vertices

1.11 Sub-graphs and Complements

Let G= (V, E) be a graph, A graph H= (V', E') is said to be a sub graph of G if E' is a subset of E and V' is a subset of V such that an edge $(v_i\ v_j)$ is in E' only if v_i and v_j are in v_j. For example, the graph in Fig. 1.5(b) is a sub-graph of the graph in Fig. 1.5(a).

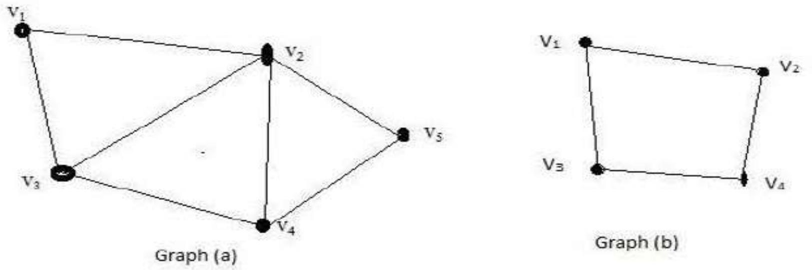


Fig: 1.5 Graph (a) and one of its sub-graph (b)

Spanning Sub-graph

A sub-graph H of a graph G is said to be spanning sub-graph if all the vertices of G are present in the sub graph H.

It is easy to observe that every graph is its own sub-graph. Also, if H is a sub-graph of a graph G and K is a sub-graph of H then K is a sub- graph of G.

Edge-disjoint Sub-graphs

Two sub-graphs H_1 and H_2 of a graph G are said to be edge-disjoint if H_1 and H_2 do not have any edge in common. It may be noted that although edge-disjoint graphs do not have any edge in common, they may have vertices in common.

Vertex-disjoint Sub-graphs

Two sub graphs H_1 and H_2 of a graph G are said to be vertex-disjoint if H_1 and H_2 do not have any vertices in common. Obviously, every vertex-disjoint sub-graphs are edge-disjoint.

Complement of a Sub-graph

Let H= (V', E') be a sub-graph ofa graph G = (V, E). The complement of sub-graph H with respect to the graph G is the sub-graph H(G)= (V, E-E').

Thus, the complement of a sub graph H with respect to graph G is obtained from G by removing the edges of H.

Complement of a Simple Graph

Let G = (V, E) be a simple graph. The complement of Gis the graph G = (V, E ') such that on edge e is in E I if and only if it is not in E. In other words, two vertices v_i and V_j are adjacent in G if and only if they are not adjacent in G.

A simple graph which is isomorphic to its complement is called self complementary graph. For example, the graph given in fig. 1.6 below is self-complementary.

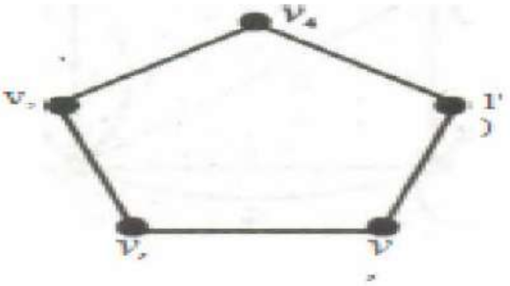


Fig. 1.6

We conclude this section with a puzzle which is also known as Instant Insanity.

Example 1.38: What is the maximum number of vertices in a graph with 35 edges and all vertices are of degree at least 3.

Solution: Let n be the number of vertices. Since the graph has 35 edges and each edge contributes 2 to the sum of degrees of all vertices, therefore sum of degrees of all vertices is 2 x 35 = 70. Also, each vertex is of degree at least 3, therefore

$70 \geq 3n$

The largest integer n satisfying this inequality is 23. Hence maximum number of vertices in a graph with 35 edges such that each vertex is of degree at least 3 is 23.

Example 1.39: Prove that if G is self complementary then G has 4k or 4k+ 1 vertices, where k is an integer.

Solution: Let the number of vertices in a self-complementary graph G be n. We know that maximum number of edges in a simple graph With vertices is $\frac{n(n-1)}{2}$.

Since G is self -complementary an isomorphic graphs have equal number of edges, therefore, G and its complement each must have $\frac{n(n-1)}{4}$ edges. Hence n or n-1 must be a multiple of 4. Thus

n = 4k or n-1 = 4k, where k is an integer.

$\Rightarrow$ n = 4k or 4k+ I, where k is an integer.

Example 1.40: Is it possible to have a group of nine people such that each person is acquainted with exactly five of the other people?

Solution: No such group of nine people can exist. For, if we try to draw a graph with a vertex for each person and an edge between each pair of people who know each other, then we would have a graph with nine vertices all of degree 5. But then sum of degrees of all vertices in this graph would be 45 which is not possible because sum of degrees of all vertices in any graph is always even.

1.12 Trees

A Connected graph without any circuits is called a tree.

Example 1.41: The graph G shown in Fig. 1.6 (b) is a tree while the graph shown, in Fig. 1.6 (a) is not a tree.

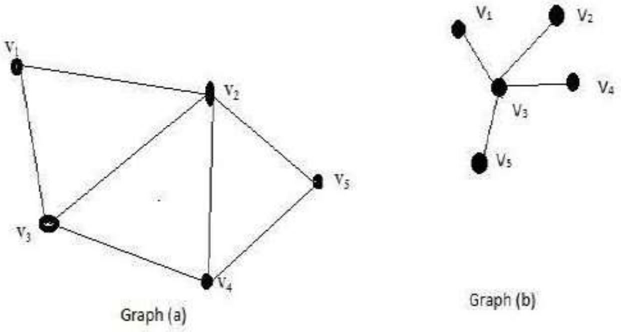


Fig. 1.6

Theorem: A graph G is a tree iff there is one and only one path between any two vertices of G.

Proof: First suppose that the graph G is a tree. Then by definition of a tree, G is a connected graph. Therefore, there must exist at least one path between any two vertices in G. Now suppose that there are two distinct paths between vertices 'a' and 'b' of G. Then the union of these two paths will contain a circuit and G cannot be a tree. Thus there is one and only one path between any two vertices of G.

Conversely, suppose that there is one and only one path between any two vertices of G. We shall show G is a tree. Since there exist a path between any two vertices of G, therefore G is connected. A circuit in a graph with two or more vertices implies that there exists a pair of vertices a, b such that there are two distinct paths between 'a' and 'b'. Since G has one and only one path between any two vertices, G can have no circuit. Thus G is a tree. .

Theorem : A tree with n vertices has n-1. edges.

Proof: We shall prove the theorem by induction on the number of vertices. Clearly, the theorem is true for trees with one or two vertices assume that the theorem is true for all trees with fewer than n vertices.

Let us consider a tree G with n vertices. Let e_k be any edge in G with end vertices V_i and v_j. According to Theorem 1 above, the edge e_k is the only path between V_i and v_j Hence, deletion of e_k from G will disconnect the graph. Thus G-e $_k$ is not connected. Further, G-e_k will contain exactly two components, for otherwise the graph G will not be connected. Let these two components of G - e_k be G_1 and $G_{2.}$ Each of these components is a tree because there were no circuits in G. Let n_1 and n_2 be the number of vertices in G_1 and G_2 respectively. Since $n_1 < n$ and $n_2 < n$, we have by the induction hypothesis

$$\text{number of edges in } G_1 = n_1-1$$

and

$$\text{number of edges in } G_2 = n_2-1$$

Thus, number of edges in G - e_k is equal to $(n_1-1) + (n_2-1) =$

$(n_1 + n_2)-2 = n-2$. Hence G has exactly n-l edges.

Theorem: Every connected graph with n vertices and n-l edges is a tree.

Proof: Let G be a connected graph with n vertices and n - 1 edges. The theorem will be proved if we show that G has no circuit. Suppose that G contains at least one circuit. Since removing an edge from a circuit does not disconnect a graph, we may remove edges, but no vertices from circuits in G until the resulting graph G* is circuit-free. Now G* is a connected graph with n vertices and contains no circuit. Thus G* is a tree with n vertices. Hence G* has n-l edges (by Theorem 2). But now the graph G has more than n-l edges, a contradiction. Hence G has no circuit. This completes the proof.

Theorem: A graph G with n vertices, n-l edges and no circuits is tree.

Proof: Let G be a graph with n vertices, n-l edges and has no circuit. It will be a tree if we show that it is connected. If possible, suppose that G is disconnected. Then G will consist of two or more circuit less components. Without loss of generality let G consist of two components G_1 and G_2. We add an edge e between a vertex VI in G1 and v2 in G2. Since V_I and v_2 are in different components of G, there is no path between v_I and v_2 in G. Thus addition of edge e will not create a circuit. Thus G U e is a circuit less, connected graph (and therefore a tree) of n vertices and n edges, which is not possible because of Theorem 2. This completes the proof.

Minimally Connected Graph

A connected graph G is said to be minimally connected if removal of any edge from G disconnects the graph G. We now have the following theorem:

Theorem: A graph G is a tree iff it is minimally connected.

Proof: Suppose that G is a tree. We show G is minimally connected.

Since G is tree, it is connected. If G is not minimally connected then there must exist an edge e in G such that G-e is connected. Therefore, e is in some circuit, which implies that G is not a tree, a contradiction.

Thus G is minimally connected.

Conversely, suppose that G is a minimally connected graph. Then G is connected and cannot have a circuit; otherwise, we could remove one of the edges in the circuit and still leave the graph connected. Thus a minimally connected graph is a tree.

Minimum Number of Pendant Vertices in a Tree

Recall that a pendant vertex in a graph is that vertex whose degree is one. In general, trees have several pendant vertices. The minimum number of pendant vertices in a tree is given by the following theorem:

Theorem: In any tree (with two or more vertices), there are at least two pendant vertices.

Proof: Let G be any tree having n vertices. Then G has n-l edges. Since each edge contributes two degrees, the sum of the degrees of all vertices in G is 2(n-1). Now 2(n-1) degrees are to be divided among n vertices in G. Let the number of vertices of degree one in G be x. Since no vertex in a tree can be of zero degree, we have

$$\frac{2(n-1)-x}{n-x} \geq 2$$
$$\Rightarrow x \geq 2$$

Thus, we must have at least two vertices of degree one in a tree.

Spanning Trees

In this section, we shall study the tree as a sub-graph of a connected graph.

A sub-graph T of a connected graph G is said to be a spanning tree of G if the sub-graph T is a tree and contains all the vertices of G.

Equivalently, a tree T is said to be a spanning tree of a connected graph G if T is a sub-graph of G and contains all vertices of G. A spanning tree is also called a skeleton or maximal tree sub-graph.

Since a tree is a connected graph, a spanning tree is defined only for a connected graph. If G is an arbitrary graph with k components then each of the k components does have a spanning tree. Thus a disconnected graph with k components has a spanning forest consisting of k spanning trees (A collection of trees is called a forest).

Theorem : Every connected graph has at least one spanning tree.

The graph obtained by removing an edge from a circuit in G will remain connected. If there are more circuits, repeat the process until we get a connected, circuit-free graph that contains all the vertices of G. This graph will then be a spanning tree of G.

1.13 Summary

Sets: Set is defines as well define collection of objects. Basic operations on sets are

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

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

$A' = U - A = \{X : X \in U \text{ and } x \in A\}$

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

Relation: If A and B be any two sets then a relation R from A to B in a subset of $A \times B$. So we can define R as a subset of $A \times B$ i.e. $R \subseteq A \times B$.

Types of Relation :

(a) **Reflexive:** If in a relation R defined on set A $(x, x) \in R \forall x \in A$ then R is called Reflexive Relation.

(b) **Symmetric:** A Relation R defined on a set A is called symmetric Relation if $(a, b) \in R \Leftrightarrow (b, a) \in R$

(c) **Transitive:** A Relation R defined on a set A is called Transitive if for every pair $(a, b) \in R$ and $(b, c) \in R \Rightarrow (a, c) \in R$.

(d) **Anti-Symmetric Relation :** A Relation R defined on set A is anti symmetric if $(a, b) \in R \ \& \ (b, a) \in R \Rightarrow a = b$

(e) **Equivalence Relation :** A Relation R defined on set A in equivalence Relation if

(i) R is Reflexive

(ii) R is Symmetric

(iii) R is Transitive

Function: Let A and B be two non-empty sets then a function f from set A to B is a relation which relates elements of A to B as

(i) All element of set A are related to elements of B.

(ii) An element of set A us related to a unique element of set B.

Types of Function:

(a) **One-one function (injection)**: A function $f:A\to B$ is said to be one-one function if different elements of A has different images in B. if f is not one-one then it is called many one.

(b) **On-to function (surjection) :** A function $f:A\to B$ is said to be on to if every element of B has a pre f image in A i.e. if range of f is B i.e. f(A) = B

(c) **In-to Function** : A function which is not on-to is called in-to function.

(d) **One-one onto function (Bijection):** A function which is one-one as well as onto is called one-one onto function.

Graph: A graph G is a pair (V, E), where V = {v_1, v_2, ... } is a non- empty set whose elements are called vertices (or nodes) and E = (e_1, e_2, .. } is a set such that each element e_k of E is identified with an unordered pair (v_i v_j) of vertices.

Sub-graph: Let G= (V, E) be a graph, A graph H= (V', E') is said to be a sub graph of G if E' is a subset of E and V' is a subset of V such that an edge (v_i , v_j) is in E' only if v_i and v_j are in v_j.

Tree: A Connected graph without any circuits is called a tree.

1.14 Glossary

- Injection Mapping: One-one function.
- Surjection Mapping: On-to function.
- Bijection Mapping: One-one and on-to function.

- Circuit: A closed path.
- Tree**:** A Connected graph without any circuits.
- Singleton Set: A set which contains exactly one element.

Null Set: A set which contains no element.

1.15 Exercise

1. If Set A = {1, 2, 3} and Relation defined on A as

   (i) $R_1 = \{(1, 1) (2, 2) (3, 3), (1, 2) (1, 3) (2, 3)\}$

   (ii) $R_2 = \{(1, 1) (2, 2) (3, 3)\}$

   (iii) $R_3 = \{(1, 1) (2, 3) (3, 1), (3, 2) (1, 3) (3, 3)\}$

   Check whether the following Relation are (a) Reflexive (b) symmetric (c) transitive

2. If a relation R defined on set of rectangles as two rectangles are related if they have same area' then shows that R is a equivalence relation.

3. If a relation R defined on $Z \times Z$ as $(a,b),(c,d) \in Z \times Z$

   $(a,b)(c,d) \Leftrightarrow a+d=b+c$

   then show that R is an equivalence Relation.

4. If a relation R defined on $Z_0 \times Z_0$ define by (a,b) R (c, d)

   $\Leftrightarrow ad(b+c)=bc(a+d)$

   Show that R is an equivalence Relation.

5. Prove that the relation R on any Set A is symmetric if and only if $R = R^{-1}$

Answer

1. (i) Reflexive but not Symmetric and Transitive

   (ii) Reflexive Anti Symmetric and Transitive

   (iv) Transitive but not Reflexive and Symmetric

References and Suggested Readings

1) Discrete Mathematics by Norman L. Biggs Oxford University Press

2) Combinatorics: Ancient and Modern by Robin Wilson and John Watkins Oxford University Press.

3) Numbers and functions: From a classical-experimental Mathematician's point of view by Victor H. Moll University Science Press.

4) Discrete Mathematical Structures with Applications to Combinatorics by V Ramaswamy University Science Press.

5) Discrete Mathematics by M.K. Gupta Krishna Publication.

UNIT- 2

Mathematical Logical

Structure of the Unit

2.0 Objective

After gone through this unit learner will aware of various methods of solving logical problem using mathematical logic. He will be able to use mathematical induction to prove any result for positive integers.

2.1 Introduction

All mathematical proofs are based on logic. In present logic is very important due it's practical applications in developing commutating machines, artificial intelligence etc.

2.2 Propositions

Declarative sentences that are either true or false but not both are called propositions. If a proposition is true then it has truth value "True" and if it is false then truth values is False True value is represent as T or 1 and False as F or 0.

Examples 2.1 : Following statements are propositions

(a) Jaipur is Capital of Rajasthan

(b) 3 is integer

(c) $1+2 = 4$

(d) Sun rises from west.

Propositions (a) and (b) are true but

Propositions (c) and (d) are False

Examples 2.2 : Following sentences are not Propositions

a) $X+2 = 1$

b) $X > 1$

c) What is your name?

d) Don't try to make it easy.

Here sentences (a) and (b) are not propositions unless specific vaues given to x we can't say whether they are true or false and (c) and (d) are not declarative sentences.

logic declarative sentences are denoted in symbolic from by English alphabets as p, q, r...............these are known as propositional variables.

2.3 Compound Propositions

Simple propositions are combined by logical connectives to give compound propositions. If a compound proposition is formed, by combining two propositions then it has 2^2 truth values in all alternative cases. A compound proposition made up of propositions will have 2^2 alternative cases of its truth values.

The various logical connectives also called logical operators, used to combine propositions are the following:

S.No.	Name of Connective	Denoting symbol	Meaning in short
1	Negative of p	~p (not p)	If (not p) p is true ~ p is false and vice-versa.
2	Conjunction of p and q	$p \wedge q$(p and q)	If p is true and q also is true then (p and q) only $p \wedge q$ is true.
3	Disjunction of p and q	$p \vee q$ (p or q)	If p is true or q is true i.e., if either both or atleast one of them is true then $p \vee q$ is true (p true or q true)
4	Conditional p implies q	$p \leftrightarrow q$	It is always true except in one case when p is true but q is false (it is called implication implies or conditional)
5	Biconditional $p \leftrightarrow q$ p is implies q and also q implies p	$p \leftrightarrow q$ (both $p \rightarrow q$ and $q \rightarrow p$	It is true when either p and q are both true or when they are both false.

2.4 Truth Table

If a compound proposition is formed using the above connective on some atomic propositions p, q etc., then the table giving the truth values of the compound proposition in all possible alternative cases is called its truth table.

(a) True table of Negation (~p)

p	~p	$\sim(\sim p) \equiv p$
T	F	T
F	T	F

(b) Truth table of conjunction $(p \wedge q)$

p	q	$(p \wedge q)$
T	T	T
T	F	F
F	T	F
F	F	F

(c) Truth Table of disjunction $p \vee q$

p	q	$p \vee q$
T	T	T
T	F	T
F	T	T
F	F	F

(d) Truth table of implication $(p \leftrightarrow q)$

p	q	$(p \leftrightarrow q)$
T	T	T
T	F	F
F	T	T
F	F	T

(e) Bi-conditional

p	q	$p \rightarrow q$	$q \rightarrow p$	$p \leftrightarrow q \equiv (p \rightarrow q) \wedge (q \rightarrow p)$
T	T	T	T	T
T	F	F	T	F
F	T	T	F	F
F	F	T	T	T

2.5 Tautology and fallacy

If truth values of a compound proposition involving two or more propositions be true in all cases then it is called Tautology and is denoted by T is it is false in all cases then it is called fallacy and is denoted by F.

A compound proposition which is neither a Tautology nor a fallacy is called a contingency.

Examples 2.3 : Show that

i. $p \vee \sim p$ is a Tautology

ii. $p \wedge \sim p$ is a fallacy

p	~p	$p \vee \sim p$	$p \wedge \sim p$
T	F	T	F
F	T	T	F

From truth table all values of $p \wedge \sim p$ are truth so $p \wedge \sim p$ is a Tautology and all values of $p \wedge \sim p$ are false so $p \wedge \sim p$ is a fallacy.

Equivalences:

Compound propositions which have the same truth values in all possible cases are said to be logically equivalent.

Example 2.4: Prove that $p \rightarrow q$ and $\sim q \rightarrow \sim p$ are equivalent

p	**q**	**$p \rightarrow q$**	**~p**	**~q**	**$\sim q \rightarrow \sim p$**
T	T	T	F	F	**T**
T	F	F	F	T	**F**
F	T	T	T	F	**T**
F	**F**	**T**	**T**	**T**	**T**

From the table third and sixth column are same so

$p \rightarrow q \equiv \sim q \rightarrow \sim p$

Example 2.5: Verify the distributive laws:

i. $p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \vee r)$

ii. $p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$

p	**q**	**→**	**$(q \vee r)$**	**$p \wedge (q \vee r)$**		**$(p \wedge q) \vee (p \vee r)$**			
T	**T**	**T**	**T**	**T**		**$T \vee T$**	**T**		**As the column (i) and (ii) representing L.H.S. and R.H.S. of $p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \vee r)$ So these are identical, so the equivalence (i) holds true.**
T	**T**	**F**	**T**	**T**	**(i)**	**$T \vee F$**	**T**	**(ii)**	
T	**F**	**T**	**T**	**T**		**$F \vee T$**	**T**		
F	**T**	**T**	**T**	**F**		**$F \vee F$**	**F**		
T	**F**	**F**	**F**	**F**		**$F \vee F$**	**F**		
F	**T**	**F**	**T**	**F**		**$F \vee F$**	**F**		
F	**F**	**T**	**T**	**F**		**$F \vee F$**	**F**		
F	**F**	**F**	**F**	**F**		**$F \vee F$**	**F**		

(ii)

p	**q**	**r**	**$p \vee (q \wedge r)$**			**$(p \vee q) \wedge (p \vee r)$**			
T	T	T	T	T=T		$T \vee T$	T		As the column (i) and (ii) representing L.H.S. and R.H.S. of $p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$ are identically same, so the equivalence (ii) holds.
T	T	F	T	F=T	(i)	$T \vee T$	T		
T	F	T	T	F=T		$T \vee T$	T	(ii)	
F	T	T	T	F=F		$T \vee T$	T		
T	F	F	T	T=F		$T \vee T$	T		
F	T	F	F	T=F		$T \vee F$	F		

F	F	T	F	T=F		$F \vee F$	F		
F	F	F	F	F=F		$F \vee F$	F		

Example 2.6 : Prove that $p \to q \equiv \sim q \vee p$, using truth table.

p	**q**	**$p \to q$**	**$\sim q \vee p$**
T	**T**	**T**	**$F \vee T = T$**
T	**F**	**F** **(i)**	**$F \vee F = F$** **(ii)**
F	**T**	**T**	**$T \vee T = T$**
F	**F**	**T**	**$T \to F = F$**

As (i) and (ii) column are identical hence the equivalence holds true.

If in $p \to q$.

p be the conjunction of n propositionsH_1, H_2, H_n then the implication $p \to q$ reads.

$H_1 \wedge H_2 \ldots\ldots\ldots \wedge H_n \to q$

It is then called an argument form, where $H_1 \wedge H_2 \ldots\ldots\ldots \wedge H_n$ are called its hypothesis or premises and q the conclusion.

Example 2.7: Prove that following propositions are tautology:

i. $(p \wedge q) \to (p \vee q)$

ii. $\sim(p \to q) \to \sim q$

Sol. (i)

p	**q**	**$p \wedge q$**	**$p \vee q$**	**$(p \wedge q) \to (p \vee q)$**
T	T	T	T	**T**
T	F	F	T	**T**
F	T	F	T	**T**

F	F	F	F	T

(ii)

p	q	$p \to q$	$\sim(p \to q)$	$\sim q$	$\sim(p \to q) \to \sim q$
T	T	T	F	F	T
T	F	F	T	T	T
F	T	T	F	F	T
F	F	T	F	T	T

Example 2.8: Prove that following propositions are fallacies

i. $(p \wedge q) \wedge \sim(p \vee q)$

ii. $(p \vee q) \wedge (\sim p \wedge \sim q)$

(i)

p	q	$p \wedge q$	$p \vee q$	$\sim(p \vee q)$	$(p \vee q) \wedge \sim(p \vee q)$
T	T	T	T	F	F
T	F	F	T	F	F
F	T	F	T	F	F
F	F	F	F	T	F

(ii)

p	q	$\sim p$	$\sim q$	$p \wedge q$	$\sim p \vee \sim q$	$(p \vee q) \wedge (\sim p \vee \sim q)$
T	T	F	F	T	F	F
T	F	F	T	F	T	F

F	T	T	F	F	T	F
F	F	T	T	F	T	F

2.6 Self Learning Exercise

1. Check following statement for propositions

   i. Vardhman Mahaveer Open University is located at Kota.

   ii. 2017 is a leap year.

   iii. x+2 = 5

   iv. x >2 x+1>3

   v. A year has at least 52 Sunday.

   vi. When will you complete your assigned work?

   vii. Hurry up!

   viii. Vardhman Mahaveer Open University take admissions twice in a year.

(Ans: i,ii,iv,v viii are propositions.)

2. Prove the following using truth table

(i) $\sim(p\vee q)\equiv\sim p\wedge\sim q$ (De-Morgan's Law)

(ii) $\sim(p\wedge q)\equiv\sim p\wedge\sim q$ (De-Morgan's Law)

(iii) $p\wedge(q\vee r)\equiv\sim(p\wedge q)\vee(p\wedge r)$ (Distributive Law)

(iv) $p\vee(q\wedge r)\equiv(p\vee q)\wedge(p\vee r)$ (Distributive Law)

(v) $p\vee(q\vee r)\equiv(p\vee q)\vee r$ (Associative Law)

(vi) $\sim(p\vee(\sim p\wedge q))\equiv\sim p\wedge\sim q$

(vii) $(p\rightarrow r)\vee(p\rightarrow q)\equiv p\rightarrow(q\vee r)$

(viii) $(p\rightarrow q)\wedge(p\rightarrow r)\equiv p\rightarrow(q\wedge r)$

(ix) $p\rightarrow q\equiv\sim p\vee q$

(x) $(p\rightarrow q)\wedge(q\rightarrow r)\equiv(p\vee q)\rightarrow r$

3. Prove that following compound propositions are tautologies.

(i) $\sim p\rightarrow(p\rightarrow q)$

(ii) $(p \wedge q) \rightarrow (p \rightarrow q)$

(iii) $(p \rightarrow q) \rightarrow (\sim p \wedge q)$

(vi) $((p \wedge q) \vee r) \rightarrow ((p \vee r) \wedge (q \vee r))$

4. Prove that following compound propositions are fallacies

(i) $(\sim p \wedge \sim q) \wedge (p \wedge q)$

(ii) $\sim (p \wedge q) \wedge \sim (\sim p \wedge \sim q)$

2.7 Predicate

The predicate is the part of sentence which tell us what the subject does or is, i.e. predicate is everything which is not the subject for example in the sentence he is doing well. He is the subject and rest is predicate. Mathematically it is denoted by p(x) over x.

Universe of discourse:

If P(x) be a predicate statement over x then set of values which X may assume is called universe of discourse.

Universal Quantifiers:

If P(X) be true in its domain then it is expressed as $\forall x P(x)$ and called universal quantifiers.

Existential Quantifiers :

If P(x) be true for at least one value of x in its domain then it is expressed as ∃ $x P(x)$ and called existential quantifier.

2.8 Principle of Induction

Induction is a very strong technique to prove any result for integer. There we will study principle of mathematical induction and principle of complete induction.

Principle of mathematics induction is used to prove statement $P(n)$ for positive integers. This method has following steps.

1. **Basic step :** First it is verified whether given statement P(n) it true for n = 1 if p(n) is not true for n = 1 then least vale of n = h0 (say) should be found for which it is true.

2. **Hypothesis step :** In this step after verifying first step for n = 1 or n = h0) it is assumed that p(n) is true for positive integer n = k.

3. **Inductive step :** After hypothesis step for n = k we prove result for n = k+1 in this step.

Since result is true for n = k+1 so it is also true for n = k+1+1 and in similar way result is true for all positive integers.

Example 2.9 : Prove by the principle of mathematical induction that for all positive integers $n \geq 1$

$$p(n) = 1+2+3+........n = \frac{n(n+1)}{2} \textit{ holds true.}$$

Proof : Step 1 : To check if p(1) is true.

For n= 1, L.H.S. P(1) = and R.H.S. P(1) = $\frac{1(1+1)}{2} = 1$

As. L.H.S. = R.H.S. for n = 1.

Step II : Inductive step

Hypothesis : Let p (n) be true for n = k

So we assume

1+2+……….+k = $\frac{k(k+1)}{2}$, to be true.

Step III: Then we have to prove p(k+1) to be true

Or $\{1+2+.........+k\}+(k+1) = \frac{(k+1)(k+1+1)}{2}$

On using (2), L.H.S. of (3).

L.H.S. $\frac{k(k+1)}{2}$ + (k+1)

$(k+1)\left(\frac{k}{2}+1\right)$

$= \frac{(k+1)(k+1)}{2}$ = R.H.S. of (3)

Thus the inductive step ‘p(k+1) is true if p(k) is true, is proved to be true.

Giving to k, values 1,2,3…….

Example 2.10 : Prove that P(n) = $(11)^{n+2}+(12)^{2n+2}$ is divisible by 133 for n $\in$ N.

Sol. (i) P(1) = 11^3+12^3, (11+12) $(11^2-11\times12+122)=133\times23$ so P(1) is true.

(ii) Let for n = k, P(k) be true so

$11^{k+2}12^{2k+1}=133m,$ say where m is an integer be true

$11^{k+2}=133m-12^{2k+1}$, using $(11^{k+3}=11(11^{2k+1})and\,(1)$

Now $11^{k+3}+12^{2(k+1)+1}+11(^{k+2})+12^{2k+3}$

$=11(133-12^{2k+1})+12^{2k+3}$

$=133\times11+12^{2k+1}(-11+12^2)$

$=133(11+12^{2k+1})$

Which is a multiple of 133, so P(k+1) is true

Thus if P(k) is true P(k+1) is also true and P(1) is true. So P(2) is true and

So P(n) is true for all n $\in$N.

Example 2.11: Use mathematical induction to show that sum of n terms of a G.P. with common ratio 2, is (2^n-1). i.e.

P(n) = 1+2+2^2+.2^n-1 = (2^n-1),

For all positive integral values of n.

Sol. : Step : To check whether p(n) is true for n = 1,

L.H.S. p(1)= sum of 1 them = 1,

R.H.S. p(1) = 2^1-1 =1

We see that P(1) is true

Assumption let p(n) be true for a positive integer k. -----(1)

So let p(k) = 1+2+............+ 2^k=2^k-1 be true for n = 1

So p(k) is true

To show that p(k+1) is true or 1+2+.....2^{k+1} + 2^k=2^{k+1}-1

L.H.S. of p(k+1) = sum (k+1) terms

= sum of k terms + (k+1)th therm

= (1+2+.......2^{k-1})+2^k

$=2^k-1+22^k$

$$=2.2^k-1=2^{k+1}-1=R.H.S.$$

Hence p(k+1) is true if p(k) is true. Proceeding in this way. P(n) is true for all positive integral values of n.

Example 2.12: if A be a 2×2 matrix of the form $A=\begin{vmatrix}1 & a\\ 0 & 1\end{vmatrix}$

Show by mathematical induction that

$p(n)\equiv A^n=\begin{vmatrix}1 & na\\ 0 & 1\end{vmatrix}$

For a positive integer n.

Sol. (i) To check whether p(1) is true.

(ii) Let p(k) be true for an integer k.

So $\quad p(1)=A^k=\begin{vmatrix}1 & ka\\ 0 & 1\end{vmatrix}$ be true

Then so show that p(k+1) is true of $A^{k+1}==\begin{vmatrix}1 & (k+1)a\\ 0 & 1\end{vmatrix}$

Now L.H.S. of (4) using (3) = A^k. A

$$=\begin{vmatrix}1 & ka\\ 0 & 1\end{vmatrix}\begin{vmatrix}1 & a\\ 0 & 1\end{vmatrix} \text{Multiplying}$$

$$=\begin{vmatrix}1+0 & ka\\ 0+0 & 1\end{vmatrix} \text{R.H.S of (1)}$$

Hence p(k+1) is true is if p(k) is true. In this way P(n) is true for all $n\in N$.

Example 2.13: Prove that for integers, $3^n>n^3$

Solution: Let $p(n):3^n>n^3\geq 4$

Basic step: $P(4) is true \sin ce$

$3^4=81>4^3=64$

Inductive hypothesis: Let p(k) be true i.e.

$3^{k+1}>k^3,\ k\geq 4$

Inductive Step: We shall prove that P(k+1) is true whenever P(k) is true.

To prove $3^{k+1} > (k+1)^3$

Now $(k+1)^3 = k^3 + 3k^2 + 3k^2 + 3k + 1$

$$= k^3\left(1 + \frac{3}{k} + \frac{3}{k^2} + \frac{3}{k^3}\right)$$

$$< 3^k\left(1 + \frac{3}{k} + \frac{3}{k^2} + \frac{3}{k^3}\right) \ [\because p(k) is true]$$

P(k+1) will be true, if we show that

$$\left(1 + \frac{3}{k} + \frac{3}{k^2} + \frac{3}{k^3} < 3,\ k \geq 4\right)$$

Let $f(k) = 1 + \frac{3}{k} + \frac{3}{k^2} + \frac{3}{k^3}$

$$f(4) 1 + \frac{3}{4} + \frac{3}{4^2} + \frac{3}{4^3} = \frac{125}{64} < 3,\ k \geq 4$$

and $f(k+1) = 1 + \frac{3}{k+1} + \frac{3}{(k+1)^2} + \frac{3}{(k+1)^3} < 1 + \frac{3}{k} + \frac{3}{k^2} + \frac{3}{k^3} = f(k)$

$\Rightarrow f(k)$ is decreasing function of k with supremum $f(4) < 3$.

$$\Rightarrow 1 + \frac{3}{k} + \frac{3}{k^2} + \frac{3}{k^3} < 3$$

$\Rightarrow (k+1)^3 < 3^k . 3^k + 1$ is true.

Hence, by the principle of mathematical induction,

$3^n > n^3\ n \geq 4$

Example 2.14: Show that $2^n < n!\ for\ n \geq 4$

Solution : Let $P(n): 2^n < n!$

Basic step: P(4) is true since

$2^4 = 16 < 4! = 24$

Inductive hypothesis: let P(k) be true $k \geq 4$

$i.e. 2^k < k!$

Inductive Step: We shall prove that P(k+1) : $2^{k+1} < (k+1)!$ is true whenever P(k) is true whenever P(k) is true.

Now $2^{k+1} < 2(k+1)!$

$2^{k+1} < 2(k+1)!$

P(k+1) is true.

Hence, by the principle of mathematical induction, $2^{n} < n!\, for\ n \geq 4$

Example 2.15: Show that for any positive integer $n \geq 2$

$$\frac{1}{\sqrt{1}}+\frac{1}{\sqrt{2}}+\ldots\ldots\frac{1}{\sqrt{n}} > \sqrt{n}$$

Solution : Let P(n): $\frac{1}{\sqrt{1}}+\frac{1}{\sqrt{2}}+\ldots\ldots+\frac{1}{\sqrt{n}} > \sqrt{n}$

Basis step : P(2) is true since

$$\frac{1}{\sqrt{1}}+\frac{1}{\sqrt{2}}=1+\frac{1}{\sqrt{2}} > \sqrt{2}$$

Inductive hypothesis: Let P(k) be true for $k \geq 2$.

$$\frac{1}{\sqrt{1}}+\frac{1}{\sqrt{2}}+\ldots\ldots+\frac{1}{\sqrt{k}} > \sqrt{k}$$

Inductive Step: We shall prove that P(k) $\rightarrow$P(k+1)

Where $P(k+1){:}\frac{1}{\sqrt{1}}+\frac{1}{\sqrt{2}}=1\ldots\ldots\frac{1}{\sqrt{k+1}} > \sqrt{k+1}$

Now= $\frac{1}{\sqrt{1}}+\frac{1}{\sqrt{2}}=1\ldots\ldots\frac{1}{\sqrt{k+1}}$

$$\frac{1}{\sqrt{1}}+\frac{1}{\sqrt{2}}=1\ldots\ldots\frac{1}{\sqrt{k}}+\frac{1}{\sqrt{k+1}} > \sqrt{k+1}+\frac{1}{\sqrt{k+1}} \quad [\therefore p(k) is\, true]$$

$$\sqrt{k+1}\left[\frac{\sqrt{k}\,\sqrt{k}+1+1}{k+1}\right] > \sqrt{k+1}\left[\frac{\sqrt{k}\,\sqrt{k}+1}{k+1}\right] = k\sqrt{k+1}\left[\frac{\sqrt{k}\,\sqrt{k}+1+1}{k+1}\right] = k\sqrt{k+1}$$

$\therefore p(k+1) is\, true$

Hence by the principle of mathematics induction

$$\frac{1}{\sqrt{1}}+\frac{1}{\sqrt{2}}+\ldots\ldots\frac{1}{\sqrt{n}} > \sqrt{n}$$

2.9 Principle of complete induction

In principle of complete induction (also known as principle of strong induction) following steps are followed:

Basic step: First it is verified whether statement P(n) is true for n =1 if P(n) is not true for n = 1 then least values of n = no (say) should be found for which it is true.

Hypothesis step: It this step it is assumed that P(n) is true for all integers which are $\geq$ no and $\leq$ K.

Inductive step: In this step we prove result for n = K+1

Example 2.16: Show that if n is an integer greater then1; then n can be written as product of primes.

Sol. Let p(n) : The proposition that n can be written as the product of primes.

Basic step:

2 is the first prime it is the product of only one prime2,

So p(2) is true. .....(1)

Inductive step: Assume that p(n) is true not only for n = k but also fo all positive integers

$j \leq k$. .....(2)

Now is to be shown that with assumption (2) to be true,

P(k+1) is true .....(3)

Now they may be two cases for n = k+1.

Case (i) k+1 is a prime.

In this case it is a product of only one prime, which is (k+1) itself. So p(k+1) is true.

Then it is composite. Let k+1 = ab

Where a and b are members such that

$2 \leq a < k+1$

And let $a < b < k+1$

As by the assumption that all numbers less than or equal to k are expressible as product of primes, we have

A = a product of primes, as a < k

and   b = a product of primes as b < k

Hence k+1 = ab=a product of primes.

Thus p(k+1) us true if p(j) is true for all j lying between 2 and k. so the proposition is true by the principle of complete induction.

Example 2.17: Prove that every amount of postage of 12 cents or more can be formed using only two types of stamps: (i) 4-cent stamps (ii) 5-cent stamp.

Proof: p(n) = postage of 12 cents or more is to be formed; for $n \leq 12$v and the minimum 12 cents postage can be formed by taking 3 stamps of 4-cents each.

Basic step: so the properly p(n) is true for n= 12

1st we prove it by the principle of mathematical induction.

Assume p(k) to be true

then   k = 4n + 5m.(1)

the   (k+1) = 4n + 5m+(5-4)

= 5(m+1) + 4(n-1)   .... (2)

So (k + 1) is expressible in terms of stamps of 4 cent and 5 cent

So p(k) is true by principle of mathematical induction.

Next we use the method of complete induction to show that if 12, 13, 14, 15 ....k are expressible in terms of stemps of 4 and 5 cents than (k+1) is also expressible like this.

Basis step.

$12 = 4 \times 3$

$13 = 4 \times 2 + 5 + 1$

$14 = 4 \times 1 + 5 \times 2$

$15 = 5 \times 3$

Let postage of k cents and j cents Where $12 \leq j \leq k$ be Expressible in terms of 4 and 5 cents so Let k = 4m+5n be true

To prove k + 1 is also expressible as in (1)

Now   k+1 = 4+(k-3)

As k – 3 is expressible in 4 + 5 cents is

$= 4+4m_1 +n_1$

Hence   p(k + 1) is true

If   P(n) is true for k and less then k.

So as P(12) is true P(13) is also true

When P(13) is true P(14) is true

So P(n) is true for all $n \leq 12$.

Example 2.18: Consider the Fibonacci sequence f_1, f_2, f_3......... where

$f_1, f_2 = 1 \, and \; f_n = f_{n-1} + f_{n-2}, n \geq 3$

Then $f_3 = f_1, + f_2 = 1 + 1 = 2$

$f_4 = f_2 + f_3 = 1 + 2 = 3$

Show by the strong principle of induction that

$$f_n \geq \left(\frac{1+\sqrt{5}}{2} \right)^{n-2}, n \geq 3$$

Solution: Let $P(n)$: $f_n \geq \left(\frac{1+\sqrt{5}}{2} \right)^{3-2}$

We prove that P_n is true for all $n \geq 3$

Basis step : For n = 3

$$f_n \geq \left(\frac{1+\sqrt{5}}{2} \right)^{n-2}$$

Hence P(3) is true.

Inductive hypothesis: Assume that P(3), P(4),…..P(k) are true for

Inductive step: We shall show that P(k+1) is true

$$f_{k+1} = f_k + f_{k-1}$$

$$\text{Now} \left(\frac{1+\sqrt{5}}{2} \right)^{k+1-2} \left(\frac{1+\sqrt{5}}{2} \right)^{2} \left(\frac{1+\sqrt{5}}{2} \right)^{k-3}$$

$$\left(\frac{1+\sqrt{5}}{2} + 1 \right) \left(\frac{1+\sqrt{5}}{2} \right)^{k-3}$$

$$\left(\frac{1+\sqrt{5}}{2} + 1 \right)^{k-2} + \left(\frac{1+\sqrt{5}}{2} \right)^{k-3} \leq f_k + f_{k-1} = f_{k+1}$$

[by the inductive hypothesis]

$\Rightarrow P(k+1)$ is true.

Hence, by the strong principle of mathematical induction, if follows that

$$f_n \geq \left(\frac{1+\sqrt{5}}{2}\right)^{n-2} \quad \forall n \geq 3.$$

Example 2.19: Prove that any postage charges of n cents can be made by using 3 and 5 cent stamps for n ≥ 8 (or n >7).

Solution: Any postage charges of n cents can be made by using 3 and 5 cent stamps.

Basis step : for n = 8, P(8) is true since

8 = 3+5=1.3+1.5 (one 3 and 5 cent stamps)

For n = 9 P(9) is true since

9 = 3 + 3 + 3 = 3.3 (Three 3 cent stamps)

For n = 10 P(10) is true since

10 = 5 + 5 = 2.5 (Two 5 cent stamps)

For n = 11 P(11) is true since

11 = 3 + 3 + 5 =2.3.1.5 (Two 3 cent and one cent stamps)

For n = 12 P(12) is true since

12 = 3 + 3 + 3+3 = 4.3 (Four 3 cent stamps)

Inductive hypothesis : Let P(i) be true for 12 $\leq$<i

i.e. = 3p + 5q for some p, q $\in$ N U {0}

Inductive step : We shall show that P(k + 1) is true since if I < k, it can be expressed as 3p + 5q. So postage of I = (k – 3) cents are expressible as

i = k - 3 = 36 +5q

$\Rightarrow$ k = 3p + 5q + 3 = 3(p + 1) + 5q

$\because$ k is also expressible in terms 3 and 5 cents.

Thus, by the strong principle of mathematical induction, if follows that P(n) is true for all n $\geq$ 8.

2.10 Summary

The various logical connectives also called logical operators, used to combine propositions are the following:

S.No.	Name of Connective	Denoting symbol	Meaning in short
1	Negative of p	~p (not p)	If (not p) p is true ~ p is false and vice-versa.
2	Conjunction of p and q	$p \wedge q$(p and q)	If p is true and q also is true then (p and q) only $p \wedge q$ is true.
3	Disjunction of p and q	$p \vee q$ (p or q)	If p is true or q is true i.e., if either both or atleast one of them is true then $p \vee q$ is true (p true or q true)
4	Conditional p implies q	$p \leftrightarrow q$	It is always true except in one case when p is true but q is false (it is called implication implies or conditional)
5	Bi-conditional $p \leftrightarrow q$ p is implies q and also q implies p	$p \leftrightarrow q$ (both $p \rightarrow q$ and $q \rightarrow p$	It is true when either p and q are both true or when they are both false.

Principle of mathematics induction is used to prove statement $P(n)$ for positive integers. This method has following steps.

1. Basic step: First it is verified whether given statement P(n) it true for n = 1 if p(n) is not true for n = 1 then least vale of n = h_0 (say) should be found for which it is true.

2. Hypothesis step: In this step after verifying first step for n = 1 or n = h_0) it is assumed that p(n) is true for positive integer n = k.

Inductive step: After hypothesis step for n = k we prove result for n = k+1 in this step.

2.11 Glossary

Propositions: Declarative sentences that are either true or false but not both.

Predicate: Part of sentence which tell us what the subject does or is.

Induction: Deduce a result from previous one.

Hypothesis: Default Assumption.

2.12 Self-Learning Exercise

Prove by mathematical induction

1. Show $\frac{n^5}{5}+\frac{n^3}{3}+\frac{7}{15}$ is a natural number for all $n \in N$

2. $1^2+2^2+3^2+..........+n^2=\frac{n(n+1)(2n+1)}{6}$

3. $\frac{1}{1.2}+\frac{1}{1.2}+.....\frac{1}{n(n+1)}=\frac{n}{n+1}$

4. Show that n (2n + 1) (n + 1) is divisible by 6, for m a positive integer.

5. Show that $\mathrm{x^{2n} - y^{2n}}$ is divisible by $(\mathrm{x+y})$ when n is natural number n.

6. 1.2.3 +2.3.4………+n(n + 1) (n + 2) $= \frac{\mathrm{n(n+1(n+2)\,(n+3)}}{4}$

7. $a+ar+ar^2+ \;..........ar^{n-1}=a\left(\frac{r^n-1}{r-1}\right) if\; r\neq 1.$

8. Prove that

(a) $1+\frac{1}{4}+\frac{1}{9}+.........\frac{1}{n^2}<2-\frac{1}{n}, n\geq 1$

[Hint: $\frac{1}{(k+1)^2}+2-\frac{1}{k}<2-\frac{1}{n}, n\geq 1$]

(b) $n! \geq 2^2$ for all integers $n \geq 4$

[Hint: $(k+1)!=(k+1)k!\geq (k+1)2^k > 2.2^k = 2^{k+1}, k\geq 4$]

References and Suggested Readings

1. Discrete Mathematics by Norman L. Biggs Oxford University Press

2. Combinatorics: Ancient and Modern by Robin Wilson and John Watkins Oxford University Press.
3. Numbers and functions: From a classical-experimental Mathematician's point of view by Victor H. Moll University Science Press.
4. Discrete Mathematical Structures with Applications to Combinatorics by V Ramaswamy University Science Press.
5. Discrete Mathematics by M.K. Gupta Krishna Publication.

UNIT-3
Introduction to Theory of Automata

Structure of the Unit

3.0 Objective

The most important objective of automata conception is to advance methods by which scientists can describe and analyze the dynamic conduct of discrete systems, where alerts are sampled periodically. The habits of these discrete programs are set incidentally that the procedure is constructed from storage and combinational factors. Characteristics of such machines incorporate:

Inputs: assumed to be sequences of symbols selected from a finite set I of enter indicators. Specifically, set I is the set x1, x,2, x3... Xk where ok is the quantity of inputs.

Outputs: sequences of symbols selected from a finite set Z. Particularly, set Z is the set y1, y2, y3 ... Ym where m is the quantity of outputs.

States: finite set Q, whose definition is dependent upon the form of automaton.

3.1 Introduction to Automata

The expression "Automata" is gotten from the Greek word "αὐτόματα" which signifies "self-acting". A machine (Automata in plural) is a dynamic self-impelled figuring gadget which takes after a foreordained arrangement of operations consequently.

An example: Controlling a toll gate earlier than we provide a formal definition of a finite automaton, we don't forget an illustration where such an automaton indicates up in a common means. We recollect the difficulty of designing a "pc" that controls a toll gate. When an automobile arrives at the toll gate, the gate is closed. The gate opens as quickly as the driver has payed 25 cents. We expect that we now have handiest three coin denominations: 5, 10, and 25 cents. We also anticipate that no excess exchange is returned.

After having arrived on the toll gate, the driving force inserts a chain of coins into the computer. At any moment, the laptop has to make a decision whether or no longer to open the gate, i.e., whether or now not the driving force has paid 25 cents (or more). With the intention to come to a decision this, the computer is in one of the most following six states, at any second for the period of the process:

- The tabletop is in state q0, if it has no longer amassed any cash yet.
- The computing device is in state q1, if it has accumulated exactly 5 cents.
- The computer is in state q2, if it has gathered exactly 10 cents
- The tabletop is in state q3, if it has gathered exactly 15 cents.
- The machine is in state this autumn, if it has collected exactly 20 cents.

- The computer is in state q5, if it has amassed 25 cents or extra.

Initially (when a car arrives on the toll gate), the computer is in state q0. Anticipate, for illustration, that the driver grants the sequence (10, 5, 5, 10) of cash.

- After receiving the primary 10 cents coin, the laptop switches from state q0 to state q2.
- After receiving the first 5 cents coin, the laptop switches from state q2 to state q3.
- After receiving the 2nd 5 cents coin, the tabletop switches from state q3 to state q4.
- After receiving the 2d 10 cents coin, the computing device switches from state q4 to state q5. At this second, the gate opens. (bear in mind that no alternate is given.)

Figure 3.1: determines under represents the conduct of the computing device for all possible sequences of coins. State q5 is represented by using two circles, considering that it's a particular state: As soon as the laptop reaches this state, the gate opens.

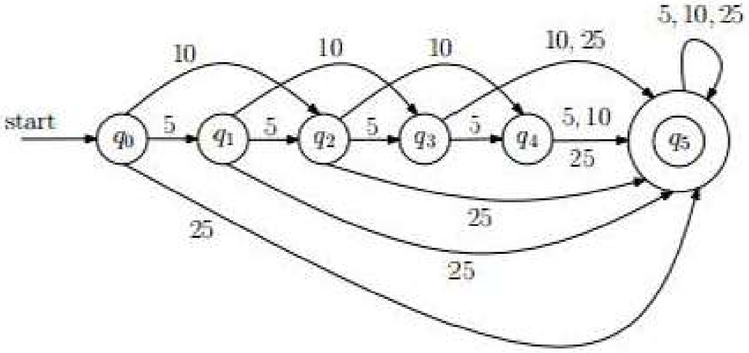


Figure 3.1: Coin Vending machine

Detect that the machine (or pc) most effective has to recall which state it's in at any given time. As a consequence, it desires most effective an extraordinarily small quantity of reminiscence: It must be in a position to differentiate between any one of six possible instances and, therefore, it most effective needs a memory of
[log 6] = 3 bits.

A robot with a limited number of states is known as a Finite Automaton (FA) or Finite State Machine (FSM).

Description of Automaton: An automaton can be defined in an abstract way by the following figure.

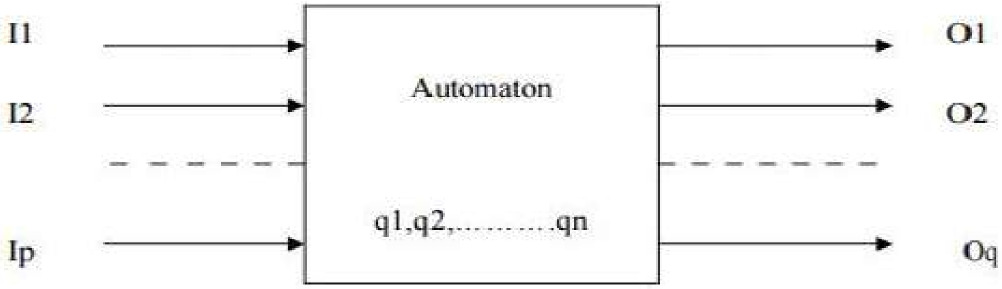


Figure 3.2: Model of a discrete automation

Input: - At each of the discrete instants of time t1,t2,.....input values I1,I2……… each of which can take a finite number of fixed values from the input alphabet ∑, are applied to the input side of the model.

Output:- O1, O2….are the outputs of the model, each of which can take finite numbers of fixed values from an output O.

States :- At any instant of time the automaton can be in one of the states q1, q2…..qn

State relation :- The next state of an automaton at any instant of time is determined by the present state and the present input. i.e., by the transition function.

Output relation :- Output is related to either state only or both the input and the state. It should be noted that at any instant of time the automaton is in some state. On 'reading' an input symbol, the automaton moves to a next state which is given by the state relation.

Definition of a finite automaton: Basic model of finite automata consists of:

- An input tape divided into cells, each cell can hold a symbol
- A read head which can read one symbol at a time from a finite alphabet
- A finite control which works within a finite set of states. At each step, it changes its state depending on the current state and input read. Its change of state is specified by a transition function. It accepts the input if it is in a set of accepting states.

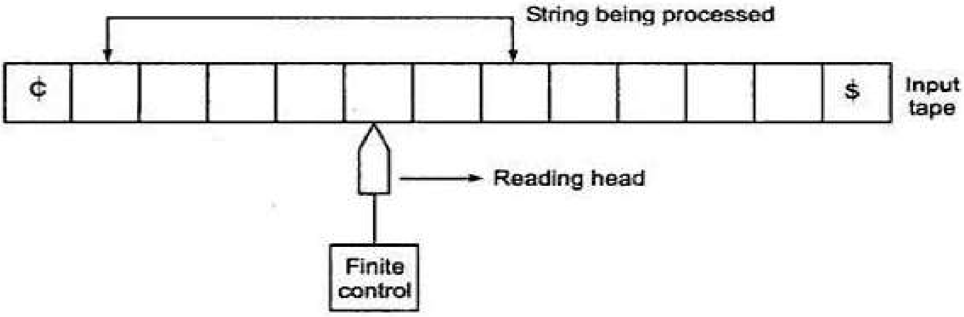


Figure 3.3: Basic Model of finite automata

Formal meaning of a Finite Automaton

A machine can be spoken to by a 5-tuple (Q, $\sum$, δ, q0, F), where −

Q is a limited arrangement of states.

$\sum$ is a limited arrangement of images, called the letters in order of the robot.

δ is the move capacity.

q0 is the underlying state from where any info is handled (q0 $\in$Q).

F is an arrangement of definite state/conditions of Q (F $\subseteq$ Q).

Related Terminologies

(i) Letter set − A letter set is any limited arrangement of images.

Case – $\sum$ = {a, b, c, d} is a letter set where 'a', 'b', 'c', and "d" are letter sets.

(ii) String – A string is a limited grouping of images taken from $\sum$.
Case – "cabcad" is a legitimate string on the letter set $\sum$ = {a, b, c,d}

(iii) Length of a String – It is the quantity of images present in a string. (Indicated by |S|).
Cases – In the event that S='cabcad', |S|= 6

In the event that |S|= 0, it is called an unfilled string (Denoted by λ or ε)

Kleene Star

Definition – The set $\sum$* is the boundless arrangement of every single conceivable string of every conceivable length over $\sum$ including λ. Representation – $\sum$* = $\sum$0 U $\sum$1 U $\sum$2 U …….
Illustration – If $\sum$ = {a, b}, $\sum$*= {λ , a, b, aa, stomach muscle, ba, bb,… … ..}

Kleene Closure/Plus

Definition – The set $\sum$+ is the interminable arrangement of every single conceivable string of every single conceivable length over $\sum$ barring λ.

Representation – $\sum$+ = $\sum$0 U $\sum$1 U $\sum$2 U……. $\sum$+ = $\sum$* – { λ }

Illustration – If $\sum$ = {a, b}, $\sum$+ ={ a, b, aa, abdominal muscle, ba, bb,……..}

Dialect

Definition – A dialect is a subset of $\sum$* for some letters in order $\sum$. It can be limited or vast.

Illustration – If the dialect takes every single conceivable string of length 2 over $\sum$ = {a, b}, then L = {stomach muscle, bb, ba, bb}

Let us look at another example. Consider the following state diagram:

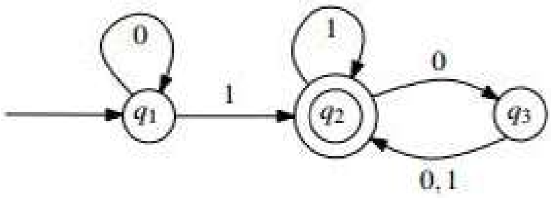


Figure 3.4

We are saying that q1 is the state and q2 is a receive state. Recollect the enter string 1101. This string is processed in the following manner:

- firstly, the laptop is within the start state q1.
- After having learn the primary 1, the computing device switches from state q1 to state q2.
- After having read the 2nd 1, the laptop switches from state q2 to state q2. (So without a doubt, it does not swap.)
- After having learn the first zero, the laptop switches from state q2 to state q3.
- After having read the 1/3 1, the computing device switches from state q3 to state q2.

After the complete string 1101 has been processed, the computing device is in state q2, which is a given state. We say that the string 1101 is authorized via the tabletop.

Consider now the enter string 0101010. After having learn this string from left to correct (starting within the start state q1), the laptop is in state q3.

Considering the fact that q3 will not be a given state, we are saying that the machine rejects the string 0101010.

We hope you're in a position to peer that this laptop accepts every binary string that ends with a 1. Actually, the laptop accepts more strings:

- Each binary string having the property that there are an even number of 0s following the rightmost 1, is authorized through this tabletop.

- Each different binary string is rejected by way of the computer. Detect that each such string is both empty, includes 0s only, or has an extraordinary number of 0s following the rightmost 1. We now come to the formal definition of a finite automaton:

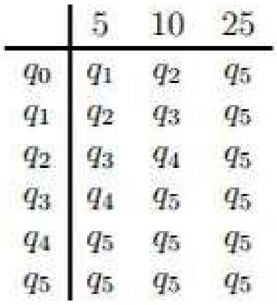
3.2 Deterministic Finite Automata

Definition: A finite automaton is a 5-tuple M = (Q, Σ, δ, q, F), the place

1. Q is a finite set, whose factors are known as states,
2. Σ is a finite set, known as the alphabet; the factors of Σ are called symbols,
3. δ : Q × Σ → Q is a perform, known as the transition operate,
4. Q is an aspect of Q; it's known as the start state,
5. F is a subset of Q; the factors of F are referred to as receive states. That you can think of the transition function δ as being the "software" of the finite automaton M = (Q, Σ, δ, q, F). This operate tells us what M can do in "one step":

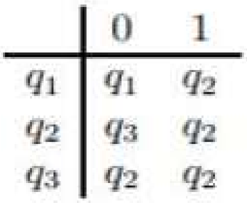
- Let r be a state of Q and let 'a' be a symbol of the alphabet Σ. If the finite automaton M is in state 'r' and reads the symbol 'a', then it switches fromstate 'r' to state δ (r, a). (actually, δ(r, a) is also equal to r.)

The "laptop" that we designed in the toll gate illustration in part is a finite automaton. For this illustration, we have now Q = q0, q1, q2, q3, this fall, q5, Σ = 5, 10, 25, the state is q0, F = q5, and δ is given by way of the following table:



	5	10	25
q_0	q_1	q_2	q_5
q_1	q_2	q_3	q_5
q_2	q_3	q_4	q_5
q_3	q_4	q_5	q_5
q_4	q_5	q_5	q_5
q_5	q_5	q_5	q_5

The illustration given within the establishing of this section can also be a finite automaton. For this illustration, we've got Q = q1, q2, q3, Σ = zero, 1, the begin state is q1, F = q2, and δ is given with the aid of the next table:



	0	1
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_2	q_2

Let us denote this finite automaton by M. The language of M, denoted by L(M), is the set of all binary strings that are accepted by M. As we have seen before, we have L(M) = {w : w contains at least one 1 and ends with an even number of 0s}.

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

Formal Definition of a DFA

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

- Q is a finite set of states.
- ∑ is a finite set of symbols called the alphabet.
- δ is the transition function where δ: Q × ∑ → Q

q0 is the initial state from where any input is processed (q0 ∈ Q).

F is a set of final state/states of Q (F ⊆ Q).

Graphical Representation of a DFA

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

Example 3.1

Let a deterministic finite automaton be →

Q = {a, b, c},

∑ = {0, 1},

q0={a},

F={c}, and

Transition function δ as shown by the following table −

Present State	Next State for Input 0	Next State for Input 1
A	a	b
B	c	a
C	b	c

Its graphical representation would be as follows:

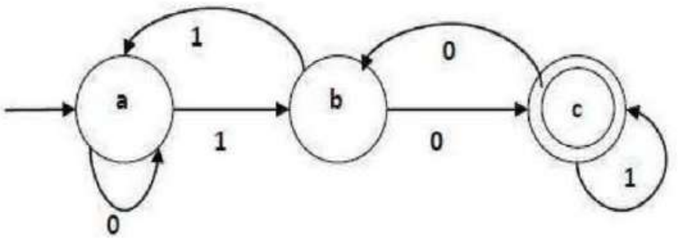


Figure3.5

Example 3.2: Q = { 0, 1, 2 }, Σ= { a, b }, A = { 1 }, the initial state is 0 and £ is as shown in the following table.

State (q)	Input (a)	Next State(δ(q,a))
0	a	1
0	b	2
1	a	2
1	b	2
2	a	2
2	b	2

Note that for each state there are two rows in the table for £ corresponding to the symbols a and b, while in the Example 3.1 there is only one row for each state. A state transition diagram for this DFA is given below.

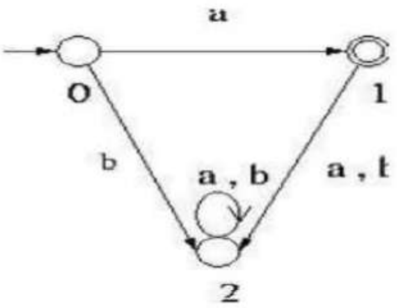


Figure 3.6: State Diagram

3.3 Acceptability of string by Finite Automata

Diagrams (when available) make it very easy to compute δ(q,w) --- just trace the path labeled w starting at q.

- E.g. trace 101 on the diagram below starting at

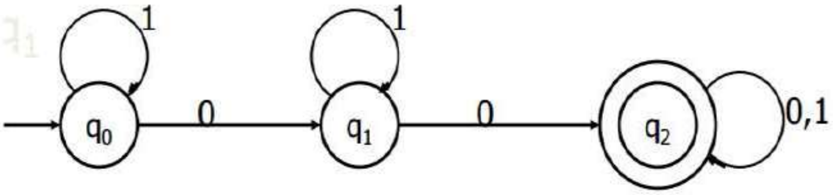


Figure 3.7: State Diagram

Implementation and precise arguments need the formal definition.

$$
\begin{aligned}
\delta(q0,0101) &= \delta(\ \delta(q0,0)\ ,101) \\
&= \delta(\ q1\ ,101) \\
&= \delta(\ \delta(q1,1)\ ,01) \\
&= \delta(q1,\ 01) \\
&= \delta(\ \delta(q1,0)\ ,1) \\
&= \delta(q2,1) \\
&= \delta(\ \delta(q2,1)\ ,\varepsilon) \\
&= \delta(q2,\varepsilon) \\
&= q2
\end{aligned}
$$

3.4 DFA Minimization using Equivalence Theorem

Minimization of a DFA refers to detecting those states whose presence or absence does not affect the language acceptability of FA.

A reduced automata consumes lesser memory, complexity of implementation is reduced, results to faster execution time, easier to analysis.

Unreachable states: if $\delta * (q0,\ w) = q'$ is not true for any w, then q' is unreachable/unaccessible state.

Dead state: $\forall$a, a $\in \Sigma$, q is dead state if δ (q, a) = q and q $\in$ Q −F.

Reachability: FA M is accessible if $\exists$w, w $\in \Sigma*$, and (q0, w) $\vdash *$ (q, ε) for all q $\in$ Q. $\vdash *$ is called reachability relation. Indistinguishable states: Two states are indistinguishable if their behaviours are indistinguishable with respect to each other. For example, p, q are indistinguishable if $\delta * (p, w) = \delta * (q, w) = r \in Q$ for all $w \in \Sigma *$.

Algorithm

If X and Y are two states in a DFA, we can combine these two states into {X, Y} if they are not distinguishable. Two states are distinguishable, if there is at least one string S, such that one of δ (X, S) and δ (Y, S) is accepting and another is not accepting. Hence, a DFA is minimal if and only if all the states are distinguishable.

Step 1: All the states Q are divided in two partitions – final states and non-final states and are denoted by P0. All the states in a partition are 0thequivalent. Take a counter k and initialize it with 0.

Step 2: Increment k by 1. For each partition in Pk, divide the states in Pk into two partitions if they are k-distinguishable. Two states within this partition X and Y are k-distinguishable if there is an input S such that δ(X, S) and δ(Y, S) are (k-1)-distinguishable.

Step 3: If Pk ≠ Pk-1, repeat Step 2, otherwise go to Step 4.

Step 4: Combine kth equivalent sets and make them the new states of the reduced DFA.

Example 3.3:

q	δ(q,0)	δ(q,1)
a	**b**	**c**
b	**a**	**d**
c	**e**	**f**
d	**e**	**f**
e	**e**	**f**
f	**f**	**f**

Let us consider the following DFA –

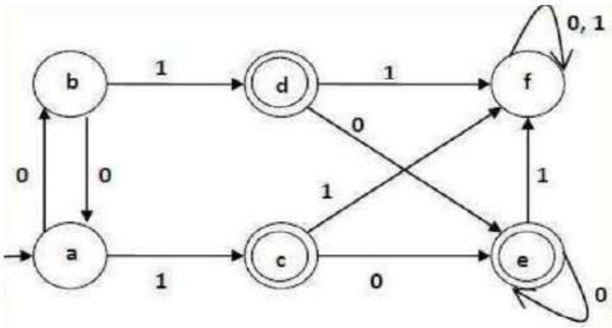


Figure 3.8: State Diagram

Let us apply above algorithm to the above DFA –

- P0 = {(c,d,e), (a,b,f)}
- P1 = {(c,d,e), (a,b),(f)}
- P2 = {(c,d,e), (a,b),(f)}

Hence, P1 = P2.

There are three states in the reduced DFA. The reduced DFA is as follows –

The State table of DFA is as follows –

Q	δ(q,0)	δ(q,1)
(a, b)	(a, b)	(c, d, e)
(c, d, e)	(c, d, e)	(f)
(f)	(f)	(f)

Its graphical representation would be as follows –

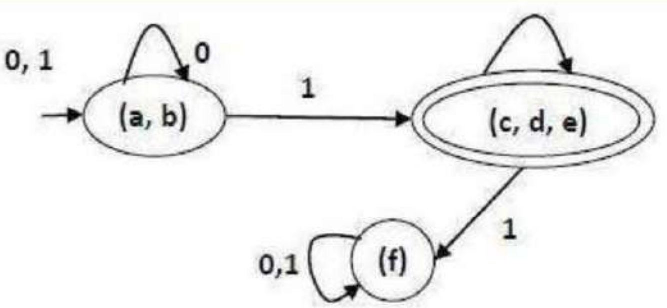


Figure 3.9: State Diagram

Example 3.4

Let us try to minimize the number of states of the followi ng DFA.

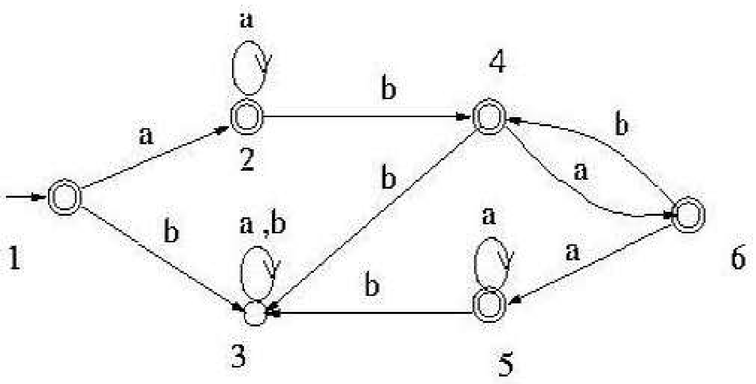


Figure 3.10

Initially = { { 3 } , { 1 , 2 , 4 , 5 , 6 } }.

By applying new_partition to this , new = { { 3 } , { 1 , 4 , 5 } , { 2 , 6 } } is obtained.

Applying new- partition to this, new = { { 3 } , { 1 , 4 } , { 5 } , { 2 } , { 6 } } is obtained.

Applying new -partition again, new = { { 1 } , { 2 } , { 3 } , { 4 } , { 5 } , { 6 } } is obtained.

Thus the number of states of the given DFA is already minimum and it can not be reduced any further.

3.5 Non-deterministic Finite Automaton

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

Formal Definition of an NDFA

An NDFA can be represented by a 5-tuple (Q, $\sum$, δ, q0, F) where–

- Q is a finite set of states.
- $\sum$ is a finite set of symbols called the alphabets.

- δ is the transition function where δ: Q × {∑ ∪ ε} → 2Q (Here the power set of Q (2Q) has been taken because in case of NDFA, from a state, transition can occur to any combination of Q states)
- q0 is the initial state from where any input is processed (q0 ∈ Q).
- F is a set of final state/states of Q (F ⊆ Q).
- Graphical Representation of an NDFA – (same as DFA)
- An NDFA is represented by digraphs called state diagram.
- The vertices represent the states.
- The arcs labeled with an input alphabet show the transitions.
- The initial state is denoted by an empty single incoming arc.
- The final state is indicated by double circles.

Example 3.5:

Let a non-deterministic finite automaton be →

Q = {a, b, c}

∑ = {0, 1}

q0 = {a}

F={c}

The transition function δ as shown below –

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

Its graphical representation would be as follows –

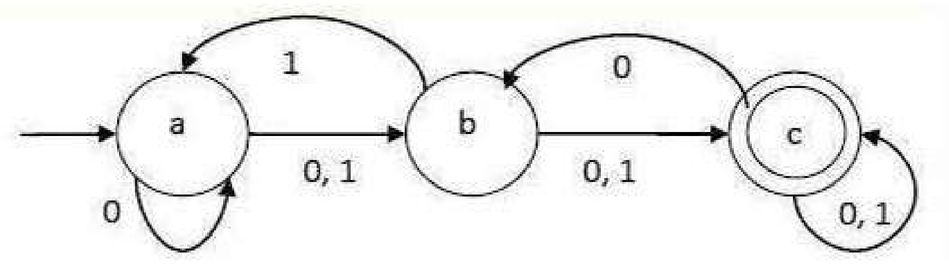


Figure 3.11

3.6 DFA vs. NDFA

The following table lists the differences between DFA and NDFA.

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

3.7 Self Learning Questions

Q.1 Which of the following statements is correct?

a) A = { If an bn | n = 0,1, 2, 3 ..} is regular language

b) Set B of all strings of equal number of a's and b's denies a regular language

c) L (A* B*)∩ B gives the set A

d) None of these

Q.2 Pumping lemma is generally used for proving that

a) given grammar is regular

b) given grammar is not regular

c) whether two given regular expressions are equivalent or not

d) None of these

Q.3 Let L be a language recognizable by a finite automaton. The language REVERSE (L) = {w such that w is the reverse of v where v ∈ L } is a

a) regular language

b) context-free language

c) context-sensitive language

d) recursively enumerable language

Q.4 The logic of pumping lemma is a good example of

a) pigeon-hole principle

b) divide-and-conquer technique

c) recursion

d) iteration

3.8 Summary

1. A robot with a limited number of states is known as a Finite Automaton (FA) or Finite State Machine (FSM).

2. States: - At any instant of time the automaton can be in one of the states q1, q2…..qn. The next state of an automaton at any instant of time is determined by the present state and the present input.
3. Definition of a finite automaton Basic model of finite automata consists of:
   (i) An input tape divided into cells, each cell can hold a symbol
   (ii) A read head which can read one symbol at a time from a finite alphabet
   (iii) A finite control which works within a finite set of states.
4. If the finite automaton M is in state r and reads the symbol a, then it switches from state r to state δ(r, a).
5. Non-deterministic Finite Automaton in NDFA, for a particular input symbol, the machine can move to any combination of the states in the machine.
6. The transition from a state can be to multiple next states for each input symbol.
7. δ is the transition function where δ: Q × {Σ ∪ ε } →2Q (Here the power set of Q (2Q) has been taken because in case of NDFA, from a state, transition can occur to any combination of Q states) q0 is the initial state from where any input is processed (q0 ∈ Q).

3.9 Glossary

Automata: - An automaton (Automata in plural) is an abstract self-propelled computing device which follows a predetermined sequence of operations automatically.

A model of a computational system, consisting of a set of states, a set of possible i nputs, and a rule to map eachstate to another state, or to itself, for any of the possib le inputs. The computational core of a Turing machine is afinite state machine. Als o called finite state automaton.

Kleene Star

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

Deterministic finite state machine

- This kind allows only one possible transition for any allowed input. This is like the "if" statement in that if x = true then doThis elsedoThat is not possible. The computer must perform *one* of the two options.

Non-deterministic finite state machine

- Given some state, an input can lead to more than one different state.

3.10 Answers to Self-Learning Exercise

Q.1 (c)

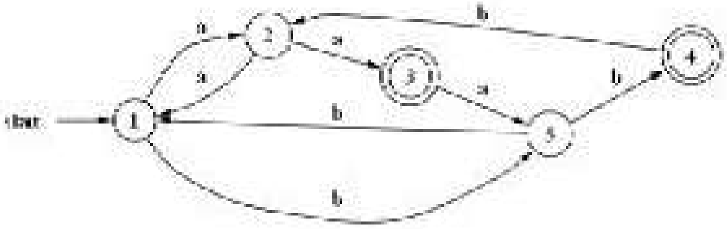
Q.2 (b)

Q.3 (a)

Q.4 (a)

3.11 Exercise

Q.1 Let M be the following DFA.



(i) Write down four strings accepted by M and the sequence of Configurations that shows this.

(ii) Write down four strings not accepted by M

Q.2 Construct a DFA which accepts the following language:

$L = \{w | w \in \Sigma * \wedge$ w contains the substring 0101} That is,w = x 0101 y for two arbitrary strings x and y

References and Suggested Readings

1. Peter Linz, An Introduction to Formal Languages and Automata, Jones and Bartlett Publication, Third Edition.

2. K.L.P. Mishra, N. Chandrasekaran, Theory of Computer Science, BPB Publication, Prentice-Hall of India, Second Edition.
3. H.E. Hopcraft and J.D. Ullamn, Introduction to Automata Theory, Languages and Computation, Narosa Publications.
4. J.C. Martin, Introduction to Languages and the Theory of Automata, Tata McGraw-Hill.

UNIT -4

Properties of Finite Automata

Structure of the Unit

4.0 Objective

- Determine the detailed action of given automata on given inputs (e.g., determine whether a given DFA accepts a given string).
- Devise simple automata to satisfy given properties (e.g., devise a pushdown automaton to recognize a given language).
- Perform tasks analogous to the above for grammars and other linguistic formalisms (e.g., devising a formal grammar for a language described in English).
- Use standard algorithms to transform automata and languages in various ways (e.g., mapping).
- Demonstrate understanding of the above by drawing suitable diagram.

4.1 Introduction

Finite automata are good models for computers with an extremely limited amount of memory. What can a computer do with such a small memory? Many useful things! In fact, we interact with such computers all the time, as they lie at the heart of various electromechanical devices.

The controller for an automatic door is one example of such a device. Often found at supermarket entrances and exits, automatic doors swing open when sensing that a person is approaching. An automatic door has a pad in front to detect the presence of a person about to walk through the doorway. Another pad is located to the rear of the doorway so that the controller can hold the door open long enough for the person to pass all the way through and also so that the door does not strike someone standing behind it as it opens. This configuration is shown in the following figure.

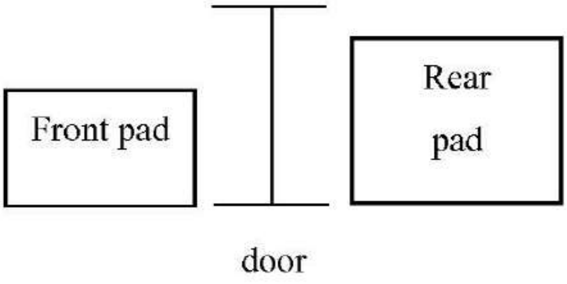


Figure 4.1

Top view o

f an automatic door

The controller is in either of two states: "OPEN" or "CLOSED," representing the corresponding condition of the door. As shown in the following figures, there are four possible input conditions: "FRONT" (meaning that a person is standing on the pad in front of the doorway), "REAR" (meaning that a person is standing on the pad to the rear of the door way), "BOTH" (meaning that people are standing on both pads), and "NEITHER" (meaning that no one is standing on either pad).

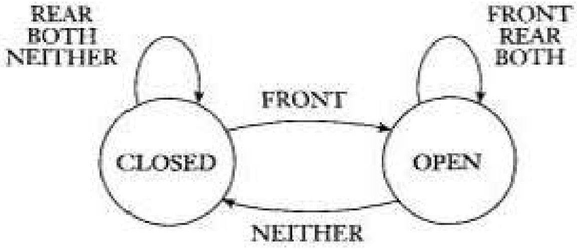


Figure 4.2

State diagram for automatic door controller

Input Signal

State	Neither	Front	Rear	Both
Closed	Closed	Open	Closed	Closed
Open	Closed	Open	Open	Open

The controller moves from state to state, depending on the input it receives. When in the CLOSED state and receiving input NEITHER or REAR, it remains in the CLOSED state. In addition, if the input BOTH is received, it stays CLOSED because opening the door risks knocking someone over on the rear pad. But if the input FRONT arrives, it moves to the OPEN state. In the OPEN state, if input FRONT, REAR, or BOTH is received, it remains in OPEN. If input NEITHER arrives, it returns to CLOSED. For example, a controller might start in state CLOSED and receive the series of input signals FRONT, REAR, NEITHER, FRONT, BOTH, NEITHER, REAR, and NEITHER. It then would go through the series of states CLOSED (starting), OPEN, OPEN, CLOSED, OPEN, OPEN, CLOSED, CLOSED, and CLOSED. Thinking of an automatic door controller as a finite automaton is useful because that suggests standard ways of representation as in **Figures 1.2 and 1.3.** This controller is a computer that has just a single bit of memory, capable of recording which of the two states the controller is in. Other common devices have controllers with somewhat larger memories. In an elevator controller a state may represent the floor the elevator is on and the inputs might be the signals received from the buttons. This computer might need several bits to keep track of this information. Controllers for various household appliances such as dishwashers and electronic thermostats, as well as parts of digital watches and calculators are additional examples of computers with limited memories. The design of such devices requires keeping the methodology and terminology of finite automata in mind. Finite automata and their probabilistic counterpart Markov chains are useful tools when we are attempting to recognize patterns in data. These devices are used in speech processing and in optical character recognition. Markov chains have even been used to model and predict price changes in financial markets. We will now take a closer look at finite automata from a mathematical perspective. We will develop a precise definition of a finite automaton, terminology for describing and manipulating finite automata, and theoretical results that describe their power and limitations. Besides giving you a clearer understanding of what finite automata are and what they can and cannot do, this theoretical development will allow you to practice and become more comfortable with mathematical definitions, theorems, and proofs in a relatively simple setting.

4.2 Non Determinism

Non-determinism is a useful concept that has had great impact on the theory of computation. So far in our discussion, every step of a computation follows in a unique way from the preceding step. When the machine is in a given state and reads the next input symbol, we know what the next state will be-it is determined. We call this deterministic computation. In a nondeterministic machine, several choices may exist for the next state at any point. Nondeterminism is a generalization of determinism, so every deterministic finite automaton is automatically a nondeterministic finite automaton. As Figure 3 shows, nondeterministic finite automata may have additional features.

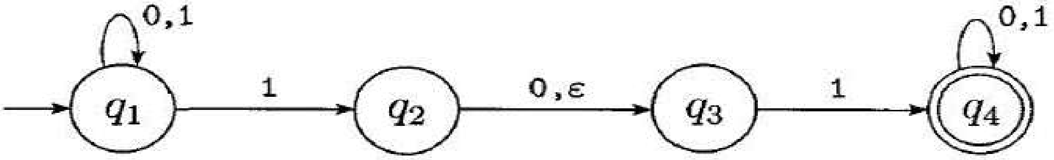


Figure 4.3

The nondeterministic finite automaton N1

The difference between a deterministic finite automaton, abbreviated DFA, and a nondeterministic finite automaton, abbreviated NFA, is immediately apparent. First, every state of a DFA always has exactly one exiting transition arrow for each symbol in the alphabet. The nondeterministic automaton shown in Figure violates that rule. State qi has one exiting arrow for 0, but it has two for 1; q2 has one arrow for 0, but it has none for 1. In an NFA a state may have zero, one, or many exiting arrows for each alphabet symbol.

Second, in a DFA, labels on the transition arrows are symbols from the alphabet. This NFA has an arrow with the label e. In general, an NFA may have arrows labeled with members of the alphabet or E. Zero, one, or many arrows may exit from each state with the label E.

How does an NFA compute? Suppose that we are running an NFA on an input string and come to a state with multiple ways to proceed. For example, say that we are in state qi in NFA N1 and that the next input symbol is a 1. After reading that symbol, the machine splits into multiple copies of itself and follows all the possibilities in parallel. Each copy of the machine takes one of the possible ways to proceed and continues as before. If there are subsequent choices, the machine splits again. If the next input symbol doesn't appear on any of the arrows exiting the state occupied by a copy of the machine, that copy of the machine dies, along with the branch of the computation associated with it. Finally, if any one of these copies of the machine is in an accept state at the end of the input, the NFA accepts the input string.

If a state with an E symbol on an exiting arrow is encountered, something similar happens. Without reading any input, the machine splits into multiple copies, one following each of the exiting E-labeled arrows and one staying at the current state. Then the machine proceeds non-deterministically as before.

Non determinism may be viewed as a kind of parallel computation wherein multiple independent "processes" or "threads" can be running concurrently. When the NFA splits to follow several choices, that corresponds to a process "forking" into several children, each proceeding separately. If at least one of these processes accepts, then the entire computation accepts.

Another way to think of a nondeterministic computation is as a tree of possibilities. The root of the tree corresponds to the start of the computation. Every branching point in the tree corresponds to a point in the computation at which the machine has multiple choices. The machine accepts if at least one of the computation branches ends in an accept state, as shown in Figure .

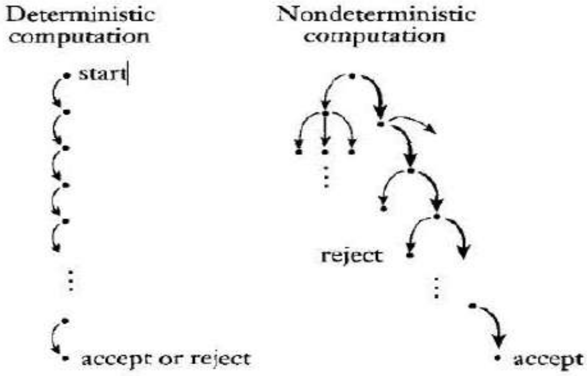


Figure 4.4

Deterministic and nondeterministic computations with an accepting branch

Let's consider some sample runs of the NFA N1 shown in Figure 1.27. The computation of N1 on input 010110 is depicted in the following figure.

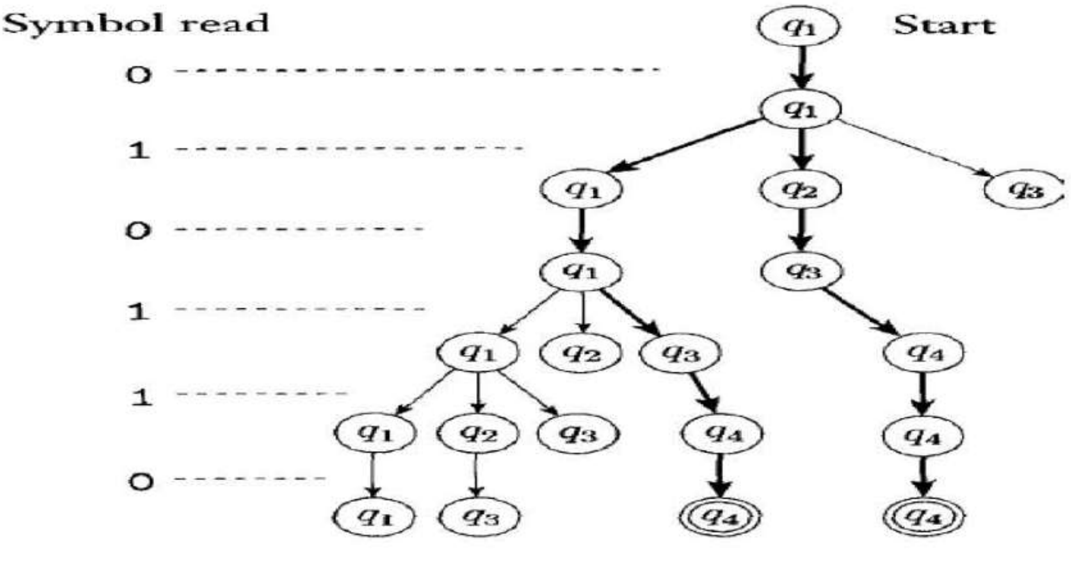


Figure 4.5

The computation of N1 on input 010110

On input 010110 start in the start state qi and read the first symbol 0. From qi there is only one place to go on a 0-namely, back to qi, so remain there. Next read the second symbol 1. In q, on a 1 there are two choices: either stay in q, or move to q2. Non-deterministically, the machine splits in two to follow each choice. Keep track of the possibilities by placing a finger on each state where a machine could be. So you now have fingers on states qi and q2. An C arrow exits state q2 so the machine splits again; keep one finger on q2, and move the other to q3. You now have fingers on qi, q2, and q3.

When the third symbol 0 is read, take each finger in turn. Keep the finger on qi in place, move the finger on q2 to q3, and remove the finger that has been on q3. That last finger had no 0 arrow to follow and corresponds to a process that simply "dies." At this point you have fingers on states qi and q3.

When the fourth symbol 1 is read, split the finger on qi into fingers on states qi and q2, then further split the finger on q2 to follow the E arrow to q3, and move the finger that was on q3 to q You now have a finger on each of the four states.

When the fifth symbol 1 is read, the fingers on qi and q3 result in fingers on states q1, q2, q3, and q4, as you saw with the fourth symbol. The finger on state q2 is removed. The finger that was on q4 stays on q Now you have two fingers on q4, so remove one, because you only need to remember that q4 is a possible state at this point, not that it is possible for multiple reasons.

When the sixth and final symbol 0 is read, keep the finger on qi in place, move the one on q2 to q3, remove the one that was on q3, and leave the one on q4 in place. You are now at the end of the string, and you accept if some finger is on an accept state. You have fingers on states qi, q3, and q4, and as q4 is an accept state, N1 accepts this string.

What does N1 do on input 01 o? Start with a finger on q1. After reading the 0 you still have a finger only on q1, but after the 1 there are fingers on q1, q2, and q3 (don't forget the E arrow). After the third symbol 0, remove the finger on q3, move the finger on q2 to q3, and leave the finger on qi where it is. At this point you are at the end of the input, and as no finger is on an accept state, N, rejects this input.

By continuing to experiment in this way, you will see that N1 accepts all strings that contain either 101 or 11 as a substring.

Nondeterministic finite automata are useful in several respects. As we will show, every NFA can be converted into an equivalent DFA, and constructing NFAs is sometimes easier than directly constructing DFAs. An NFA may be much smaller than its deterministic counterpart, or its functioning may be easier to understand. Nondeterminism in finite automata is also a good introduction to nondeterminism in more powerful computational models because finite automata are especially easy to understand. Now we turn to several examples of NFAs.

Example

Let A be the language consisting of all strings over {0, 1} containing a 1 in the third position from the end (e.g., 000100 is in A but 0011 is not). The following four-state NFA N2 recognizes A.

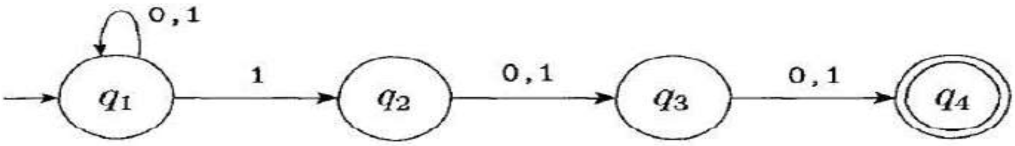


Figure 4.6

The NFA N2 recognizing A

One good way to view the computation of this NFA is to say that it stays in the start state q, until it "guesses" that it is three places from the end. At that point, if the input symbol is a 1, it branches to state q2 and uses q3 and q4 to "check" on whether its guess was correct.

As mentioned, every NFA can be converted into an equivalent DFA, but sometimes that DFA may have many more states. The smallest DFA for A contains eight states. Further more, understanding the functioning of the NFA is much easier, as you may see by examining the following figure for the DFA.

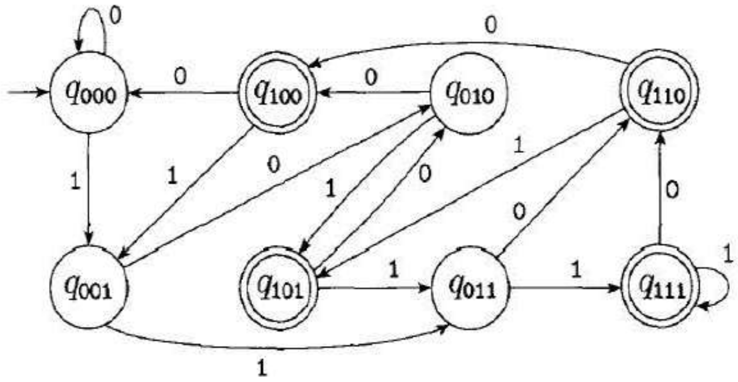


Figure 4.7

Example

Consider the following NFA N3 that has an input alphabet {0} consisting of a single symbol. An alphabet containing only one symbol is called a unary alphabet.

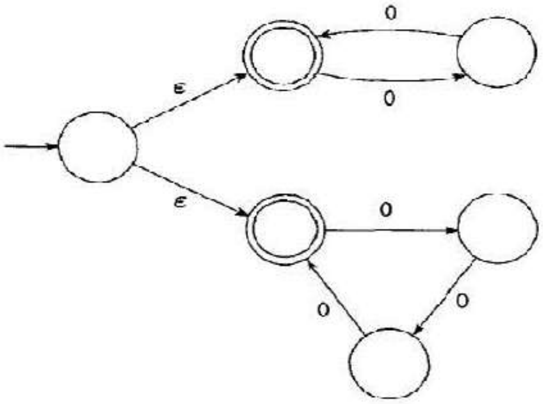


Figure 4.8

4.3 Formal Definition of Non-Deterministic Finite Automata

The formal definition of a nondeterministic finite automaton is similar to that of a deterministic finite automaton. Both have states, an input alphabet, a transition function, a start state, and a collection of accept states. However, they differ in one essential way: in the type of transition function. In a DFA the transition function takes a state and an input symbol and produces the next state. In an NFA the transition function takes a state and an input symbol or the empty string and produces the set of possible next states. In order to write the formal definition, we need to set up some additional notation. For any set Q we write P (Q) to be the collection of all subsets of Q. Here P (Q) is called the power set of Q.

For any alphabet E we write SE to be E U {e}. Now we can write the formal description of the type of the transition function in an NFA as δ: Q x Σ->, P(Q).

A nondeterministic finite automaton is a 5-tuple (Q, Σ, δ, q_0, F),

Where

1. **Q** is a finite set of states,
2. Σ is a finite alphabet,
3. δ : **Q** x Σ -> P(Q)
4. $\mathbf{q_0} \in$ **Q** is the start state, and
5. **F** $\subseteq$ **Q** is the set of accept states.

Example

Recall the NFA N1 :

The formal description of N1 is (Q, A, δ, qj, F), where

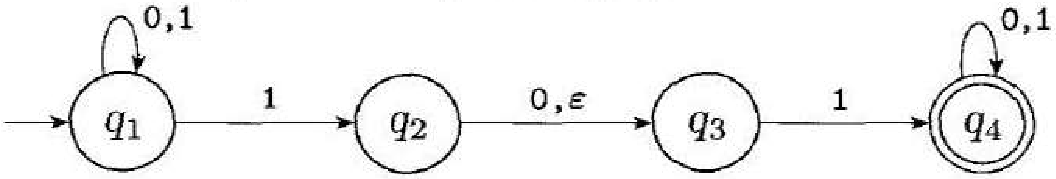


Figure 4.9

The formal description of N1 is (Q, A, Σ, qj, F), where

1. Q = {q1, q2, q3, q 4},
2. Σ= {O,1},
3. δ is given as

	0	1	ε
q1	{q1}	{q1, q2}	Ø
q2	{q3}	Ø	**{q3}**
q3	Ø	{q4}	Ø
q4	**{q4}**	**{q4}**	Ø

q, is the start state, and

4. F= {q4}.

The formal definition of computation for an NFA is similar to that for a DFA.Let N = (Q, Z, δ, q0, F) be an NFA and w a string over the alphabet E. Then we say that N accepts w if we can write w as w = Y_1Y_2 Y_n, where each y_i is a member of E, and a sequence of states r_0, r_1, .r . , r_m exists in Q with three conditions:

1. r_0 = q0,
2. $r_{i+1} \in \delta$ (r_i, y_{i+1}), for i = 0, ……m-1 and
3. $r_m \in$ F

Condition 1 says that the machine starts out in the start state. Condition 2 says that state r±i+ is one of the allowable next states when N is in state ri and reading Yi+t. Observe that δ(ri, yi+) is the set of allowable next states and so we say that rj+j is a member of that set. Finally, condition 3 says that the machine accepts its input if the last state is an accept state.

4.4 Equivalence of NFAS and DFAS

Deterministic and nondeterministic finite automata recognize the same class of languages. Such equivalence is both surprising and useful. It is surprising because NFAs appear to have more power than DFAs, so we might expect that NFAs recognize more languages. It is useful because describing an NFA for a given language sometimes is much easier than describing a DFA for that language. Say that two machines are equivalent if they recognize the same language.

Theorem

Every nondeterministic finite automaton has an equivalent deterministic finite automaton.

Proof Idea

If a language is recognized by an NFA, then we must show the existence of a DFA that also recognizes it. The idea is to convert the NFA into an equivalent DFA that simulates the NFA.

Recall the "reader as automaton" strategy for designing finite automata. How would you simulate the NFA if you were pretending to be a DFA? What do you need to keep track of as the input string is processed? In the examples of NFAs you kept track of the various branches of the computation by placing a finger on each state that could be active at given points in the input. You updated the simulation by moving, adding, and removing fingers according to the way the NFA operates. All you needed to keep track of was the set of states having fingers on them.

If k is the number of states of the NFA, it has 2k subsets of states. Each subset corresponds to one of the possibilities that the DFA must remember, so the DFA simulating the NFA will have 2k states. Now we need to figure out which will be the start state and accept states of the DFA, and what will be its transition function. We can discuss this more easily after setting up some formal notation.

Proof: Let N = (Q, Z, δ, qo, F) be the NFA recognizing some language A. We construct a DFA M = (Q', Z, Y', qo', F') recognizing A. Before doing the full construction, let's first consider the easier case wherein N has no E arrows. Later we take the E arrows into account.

1. Q' = P (Q).

   Every state of A/ is a set of states of N. Recall that P (Q) is the set of subsets of Q.

2. For R e Q' and a E E let δ'(R, a) {q G QI q E d(r, a) for some r G R}. If R is a state of M, it is also a set of states of N. When M reads a symbol a in state R, it shows where a takes each state in R. Because each state may go to a set of states, we take the union of all these sets. Another way to write this expression is

$$\delta'(R, a) = \bigcup_{r \in R} \delta(r, a).^{4}$$

3. qo' {qo}.

   M starts in the state corresponding to the collection containing just the start state of N.

F' = {R ∈ Q'I R contains an accept state of N}.

The machine M accepts if one of the possible states that N could be in at this point is an accept state.

Now we need to consider the E arrows. To do so we set up an extra bit of notation. For any state R of M we define E(R) to be the collection of states that can be reached from R by going only along E arrows, including the members of R themselves. Formally,

for R $\subseteq$ Q let.

E(R) = {qj q can be reached from R by traveling along 0 or more E arrows}.

Then we modify the transition function of M to place additional fingers on all states that can be reached by going along E arrows after every step. Replacing δ(r, a) by E (δ(r, a)) achieves this effect. Thus

$$\delta'(R, a) = \{q \in Q \mid q \in E(\delta(r, a)) \text{ for some } r \in R\}$$

Additionally we need to modify the start state of M to move the fingers initially to all possible states that can be reached from the start state of N along the E arrows. Changing q0' to be E ({qo}) achieves this effect. We have now completed the construction of the DFA M that simulates the NFA N. The construction of M obviously works correctly. At every step in the computation of M on an input, it clearly enters a state that corresponds to the subset of states that N could be in at that point. Thus our proof is complete.

If the construction used in the preceding proof were more complex we would need to prove that it works as claimed. Usually such proofs proceed by induction on the number of steps of the computation. Most of the constructions that we use in this book are straightforward and so do not require such a correctness proof. An example of a more complex construction that we do prove correct appears in the proof of Theorem

Theorem states that every NFA can be converted into an equivalent DFA. Thus nondeterministic finite automata give an alternative way of characterizing the regular languages. We state this fact as a corollary of Theorem.

Corollary: A language is regular if and only if some nondeterministic finite automaton recognizes it. One direction of the "if and only if" condition states that a language is regular if some NFA recognizes it. Theorem shows that any NFA can be converted into an equivalent DFA. Consequently, if an NFA recognizes some language, so does some DFA, and hence the language is regular. The other direction of the "if and only if" condition states that a language is regular only if some NFA recognizes it. That is, if a language is regular, some NFA must be recognizing it. Obviously, this condition is true because a regular language has a DFA recognizing it and any DFA is also an NFA.

Example: Let's illustrate the procedure we gave in the proof of Theorem for converting an NFA to a DFA by using the machine N4 that appears in Figure 4.10. For clarity, we have relabeled the states of N4 to be {1, 2, 3}. Thus in the formal description of N4 = (Q, {a, b}, 6, 1, {1}), the set of states Q is {1, 2, 3} as shown in the following figure.

To construct a DFA D that is equivalent to N4, we first determine D's states. N4 has three states, {1, 2, 3}, so we construct D with eight states, one for each subset of N4's states. We label each of D's states with the corresponding subset.

Thus D's state set is

{0, {1}, {2}, {3}, {1,2}, {1,3}, {2,3}, {1,2,3}}.

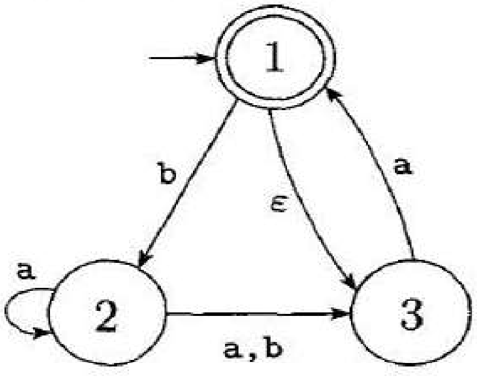


Figure 4.10

The NFA N4

Next, we determine the start and accept states of D. The start state is E ({1}), the set of states that are reachable from 1 by traveling along E arrows, plus 1 itself. An E arrow goes from 1 to 3, so E ({1}) = {1, 3}. The new accept states are those containing N4's accept state; thus {{1}, {1, 2}, {1, 3}, {1, 2, 3}}.

Finally, we determine D's transition function. Each of D's states goes to one place on input a and one place on input b. We illustrate the process of determining the placement of D's transition arrows with a few examples.

In D, state {2} goes to {2, 3} on input a, because in N4, state 2 goes to both 2 and 3 on input a and we can't go farther from 2 or 3 along E arrows. State {2} goes to state {3} on input b, because in N4, state 2 goes only to state 3 on input b and we can't go farther from 3 along r arrows.

State {1} goes to 0 on a, because no arrows exit it. It goes to {2} on b. Note that the procedure in Theorem specifies that we follow the e arrows after each input symbol is read. An alternative procedure based on following the E arrows before reading each input symbol works equally well, but that method is not illustrated in this example.

State {3} goes to {1,3} on a, because in N4, state 3 goes State {1,2} on a goes to {2,3} because 1 points at no states with a arrows and 2 points at both 2 and 3 with a arrows and neither points anywhere with E arrows. State {1, 2} on b goes to {2,3}. Continuing in this way we obtain the following diagram for D.

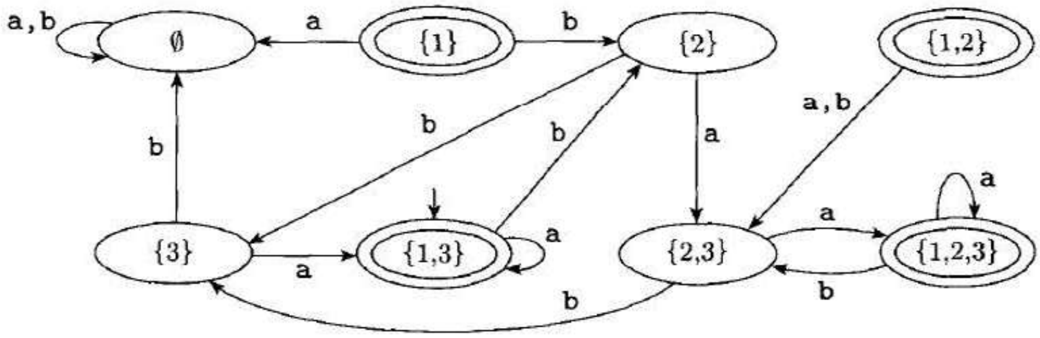


Figure 4.11

A DFA D that is equivalent to the NFA N4

We may simplify this machine by observing that no arrows point at states {1} and {1, 21, so they may be removed without affecting the performance of the machine. Doing so yields the following figure.

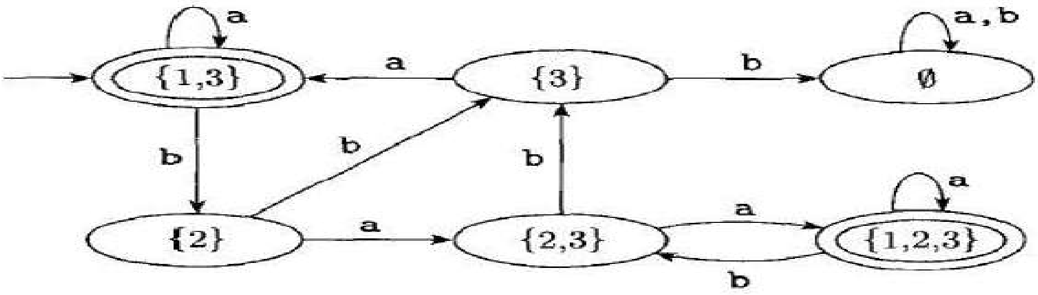


Figure 4.12

DFA D after removing unnecessary states

4.5 Melay and Moore Machine

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

1. Mealy Machine
2. Moore Machine

1. **Mealy Machine**

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

It can be described by a δ tuple (Q, ∑, O, δ, X, q0) where −

- Q is a finite set of states.
- ∑ is a finite set of symbols called the input alphabet.
- is a finite set of symbols called the output alphabet.
- δ is the input transition function where δ: Q × ∑ → Q
- X is the output transition function where X: Q → O
- q0 is the initial state from where any input is processed (q0 ∈ Q).

The state diagram of a Mealy Machine is shown below –

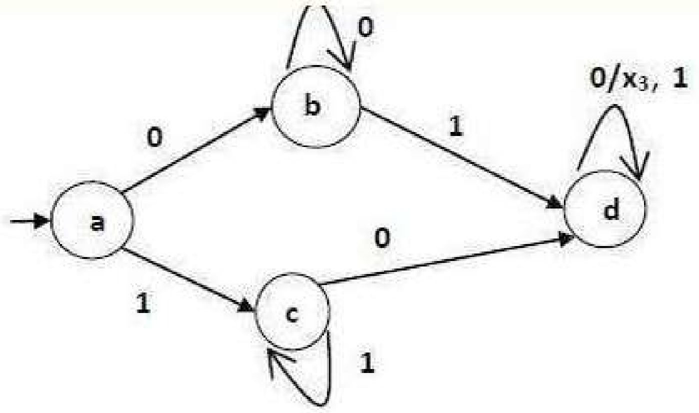


Figure 4.13

2. Moore Machine

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

A Moore machine can be described by a 6 tuple (Q, ∑, O, δ, X, q0) where Q is a finite set of states.

- ∑ is a finite set of symbols called the input alphabet.
- is a finite set of symbols called the output alphabet.
- δ is the input transition function where δ: Q × ∑ → Q
- X is the output transition function where X: Q × ∑ → O
- q0 is the initial state from where any input is processed (q0 ∈ Q).
- The state diagram of a Moore Machine is shown below –

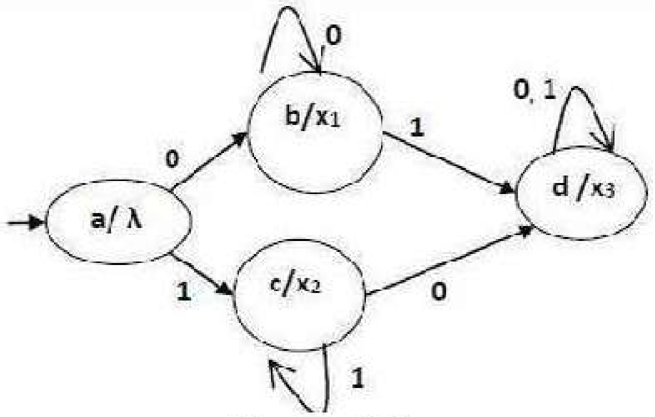


Figure 4.14

Mealy Machine vs. Moore Machine

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

Mealy Machine	Moore Machine
Output depends both upon present state and present input.	Output depends only upon the present state.
Generally, it has fewer states than Moore Machine.	Generally, it has more states than Mealy Machine.
Output changes at the clock edges.	Input change can cause change in output change as soon as logic is done.
Mealy machines react faster to inputs.	In Moore machines, more logic is needed to decode the outputs since it has more circuit delays.

Moore Machine to Mealy Machine

Algorithm 4

Input: Moore Machine

Output: Mealy Machine

Step 1 Take a blank Mealy Machine transition table format.

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

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

Example

Let us consider the following Moore machine −

Present State	Next State		Output
	a=0	a=1	
→a	d	b	1
B	a	d	0
C	c	c	0
D	b	a	1

Now we apply Algorithm 4 to convert it to Mealy Machine.

Step 1 & 2

Present State	Next State			
	a=0		a=1	
	State	Output	State	Output
→a	d		b	
B	a		d	
C	c		c	
D	b		a	

Step 3

Present State	Next State	
	a=0	a=1

	State	Output	State	Output
⇒a	d	1	b	0
B	a	1	d	1
C	c	0	c	0
D	b	0	a	1

Mealy Machine to Moore Machine

Algorithm 5

Input: Mealy Machine

Output: Moore Machine

Step 1 Calculate the number of different outputs for each state (Qi) that are available in the state table of the Mealy machine.

Step 2 If all the outputs of Qi are same, copy state Qi. If it has n distinct outputs, break Qi into n states as Qin where n = 0, 1, 2.......

Step 3 If the output of the initial state is 1, insert a new initial state at the beginning which gives 0 output.

Example

Let us consider the following Mealy Machine −

Present State	Next State			
	a=0		a=1	
	Next State	Output	Next State	Output
→a	d	0	b	1
B	a	1	d	0

C	c	1	c	0
D	b	0	a	1

Here, states ‘a’ and ‘d’ give only 1 and 0 outputs respectively, so we retain states ‘a’ and ‘d’. But states ‘b’ and ‘c’ produce different outputs (1 and 0). So, we divide b into b0, b1 and c into c0, c1.

Present State	Next State		Output
	a=0	a=1	
→a	d	b1	1
b0	a	d	0
b1	a	d	1
c0	c1	C0	0
c1	c1	C0	1
D	b0	a	0

4.6 Summary

In an NFA the transition function takes a state and an input symbol or the empty string and produces the set of possible next states.The nondeterministic finite automaton N1 The difference between a deterministic finite automaton, abbreviated DFA,and a nondeterministic finite automaton, abbreviated NFA, is immediately apparent. First, every state of a DFA always has exactly one exiting transition arrow for each symbol in the alphabet.

There are two types of finite state machines that generate output − • Mealy Machine • Moore Machine Mealy Machine A Mealy Machine is an FSM whose output depends on the present state as well as the present input.

Mealy Machine Moore Machine Output depends both upon present state and present input

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

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

4.7 Glossary

Deterministic Finite Automaton (DFA):-

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

Non-Deterministic Finite Automaton (NDFA):-

An NFA, similar to a DFA, consumes a string of input symbols. For each input symbol, it transitions to a new state until all input symbols have been consumed. Unlike a DFA, it is non-deterministic, i.e., for some state and input symbol, the next state may be nothing or one or two or more possible states.

Melay Machine:-

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

Moore Machine:-

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

Regular Expression:-

A sequence of symbols and characters expressing a string or pattern to be searched for within a longer piece of text.

4.8 Exercise

Q.1 Define NFA with ∈-transition. Prove that if L is accepted by an NFA with ∈-transition then L is also accepted by a NFA without ∈-transition.

Q.2 Let L be a set accepted by a NFA then show that there exists aDFA that accepts L

Q.3 Consider the below sample transition table of the mealy machine. Convert it into corresponding Moore machine.

Sample Transition table:

Present State	**Next State**			
	a = 0		**a = 1**	
	State	Output	State	**Output**
> q0	**q3**	**0**	ql	**1**
ql	**q0**	**1**	**q3**	**0**
q2	**q2**	**1**	**q2**	**0**
q3	**ql**	**0**	**q0**	**1**

References and Suggested Readings

1. Peter Linz, An Introduction to Formal Languages and Automata, Jones and Bartlett Publication, Third Edition.
2. K.L.P. Mishra, N. Chandrasekaran, Theory of ComputerScience, BPB Publication, Prentice-Hall of India, SecondEdition.
3. H.E. Hopcraft and J.D. Ullamn, Introduction to Automata Theory, Languages and Computation, Narosa Publications.
4. J.C. Martin, Introduction to Languages and the Theory ofAutomata, Tata McGraw-Hill.

UNIT-5

Formal Languages

Structure of the Unit

5.0 Objective

After reading this chapter you will be able to understand the following:

- Basic Definition of Grammar
- Significance of Grammar and Languages in Theory of Computation.
- Chomsky Classification of Languages.
- Types of Grammar and Languages
- Relation between Grammars and Languages.

5.1 Introduction

The theory of formal language is an area with a number of application in computer science like linguistics. The notion of the formal language comes from describing English to the computer translation. Firstly, Noam Chomsky gave a mathematical model of grammar in 1956. Later Backus –Naur form used to describe ALGOL followed the definition of grammar (a context free grammar) given by chomskey. The concept of grammar is very important to understand the theory of computation.

5.2 Basic Definition

Definition: A grammar is characterized by four-tuples (V_N, Σ, P, S) where,

- V_N is finite non empty set of elements called variables.
- Σ is finite non empty set of elements called terminals.
- S is a element of V_N is a special variable called start symbol.
- P is a finite set whose elements are $\alpha \longrightarrow \beta$, where α and β are string on $V_N \cup \Sigma$. α has at least one symbol from V_N. The elements of P are called production rules.

Example 5.1: Suppose our vocabulary only contains 'Ram' 'Sam' 'Gita' 'ate' 'walked' 'quickly' and slowly and our statements are of the form

<noun><verb><adverb>

<noun> <verb>

We can define the grammar by a 4 tuple (V_N, Σ, P, S) where

V_{N}, = {<noun >, <verb>, <adverb>}

Σ={'Ram', 'Sam', 'Gita', 'ate', 'ran' 'walked', 'quickly', slowly}

S = starting symbol

P = is the collection of production rules defined as

S ⟶ <noun > <verb> <adverb>

S ⟶ <noun> <verb>

<noun > ⟶ Ram | Sam | Gita

<verb>⟶ ate | ran | walked

<adverb> ⟶ slowly | quickly

Here, each arrow (⟶) represents a rule meaning that the word on the right hand side of the arrow can replace the word on the left hand side of the arrow.

Separator (|) is used for choosing either of the two or more production.

From the above grammar we can generate grammatically correct sentences like Sam ran, Gita walked, Ram ran slowly etc.

Derivation of Grammar

The productions are used to derive one string from over $V_N \cup \Sigma$ from another string

Suppose,

$$G= (\{S\}, \{0, 1,\}, \{S \longrightarrow OS1; S \longrightarrow 01\}, S)$$

The above grammar has production rule $S \longrightarrow 0S1$. So in 0^4S1^4 can be replaced by 0S1. The resulting string is 0^40S11^4 can be denoted as

$$0^4S1^4 \Rightarrow 0^40S11^4$$

The above process is known as ***One-Step Derivation***.

Languages generated by Grammar

Definition: The language generated by a grammar G, L (G) is defined as L (G) = $\{w \in \Sigma^* \mid S \Rightarrow^* w\}$.The elements of L (G) are called as sentences. In other word, L (G) is the set of all terminals strings derived from the start symbol S.

Example 5.2: If G= ({s}, {0, 1}, P, S), where P is $S \longrightarrow 0S1$, $S \longrightarrow 01$then find the language corresponding to grammar.

Solution: As $S \longrightarrow 01$ is a production therefore for $S \Rightarrow 01$, 01 is in L (G).Also $n \geq 1$.

$S \Rightarrow 0S1 \Rightarrow 0^2S1^2 \Rightarrow \ldots\ldots\ldots\ldots\ldots 0^nS1^n \Rightarrow 0^{n+1}S1^{n+1}$

Or $0^n1^n \in L(G)$ for n>0

So, $L(G) \subseteq \{0^n1^n \mid n>0\}$

Example 5.3: If G= ({S}, {a}, S⟶SS, S}) find language corresponding to grammar G.

Solution: Since, the only production in G is S⟶SS and in that there is no terminal.

Therefore, language generated by G is L (G) = ∅

Example 5.4: If G is S⟶aS | bS | a | b then find the language L(G)

Solution: We observe that L (G) contains L (G) = {a,b}* but not S⟶/\

Thus, the production rule

$$L(G) \subseteq \{a,b\}^* - \{\Lambda\} = \{a,b\}^+$$

5.3 Chomsky Classification of Languages

According to Chomsky (Name of Scientist) there are four types of grammar:-

(i) Type-3 Grammar or Regular Grammar

(ii) Type-2 Grammar or Context Free Grammar

(iii) Type-1 Grammar or Context Sensitive Grammar

(iv) Type-0 Grammar or Unrestricted Grammar

(i) Type-3 Grammar or Regular Grammar

These types of grammar follows the following rule of production:-

m⟶n is a production rule for regular grammar.

Where, VN = {A, B}

∑= {a, b,}

And A is starting non terminal (start symbol)

Example 5.5: Which of the following are productions of regular grammar?

Given $V_N = \{A, B\}$

$\sum = \{a, b,\}$

(i) $C \rightarrow \lambda$

(ii) $A \rightarrow a$

(iii) $A \rightarrow b$

(iv) $A \rightarrow aA$

(v) $A \rightarrow bB$

(vi) $B \rightarrow b$

(vii) $B \rightarrow aA$

(viii) $B \rightarrow aB$

Solution: All (i) to (viii) are regular grammar.

Remark: In regular grammar left side of production will always be only one variable (Ex-A, B) and in right side there will be single terminal (Ex-a, b) or one non terminal (variable) followed by terminal (E-aA, aB, bA) or λ only.

Example 5.6: Which of the following are production of regular grammar

Given $V_N = \{A, B, C\}$

$\sum = \{0, 1,\}$, A is a start symbol. Production rules are following:-

(i) $C \rightarrow \lambda$

(ii) $A \rightarrow BC$

(iii) $A \rightarrow 0C$

(iv) $AB \rightarrow 0B$

(v) $AC \rightarrow 0$

(vi) $A \rightarrow 01$

Solution: (i) and (iii) are regular grammar.

(ii) $A \rightarrow BC$ are not in RG (Regular Grammar) because right side must be one non terminal followed by terminal like (aA, aB).

(iv) $AB \rightarrow 0B$ are not in RG because left side must only one non terminal (variable)

(v) $AC \rightarrow 0$ same as (iv) not in RG

(vi) $A \rightarrow 01$ are not in RG (same logic as (ii))

(ii) Type -2 Grammar or Context Free Grammar

These Types of grammar follows the following rule of production:

$m \rightarrow n$ is a production rule for context free grammar (CFG), where

$$m \in V_N \text{ and } n \in (V_N \cup \Sigma)^*$$

Example 5.7: Which of the following production are under the context free grammar (CFG) for

$V_N = \{S,A,B\}$ and $\Sigma = \{0,1\}$:-

(a) $S \rightarrow 0$.

(b) $S \rightarrow 0A$.

(c) $A \rightarrow \lambda$.

(d) $S \rightarrow 0SA$.

(e) $S \rightarrow 0AB$.

(f) $S \rightarrow B$.

Solution: Here from (a) to (f) left side of production are single variable and right side of production are any combination of terminal and variable means $(V_N \cup \Sigma)^*$ or (S, A, B, 0, 1)*

Remarks: In the Example 5.7

(a) $S \rightarrow 0$.

(b) $S \rightarrow 0A$

(c) $A \rightarrow \lambda$.

are also follow the production rules for regular grammar or Type-3 grammar. So we can say that production rules for regular grammar is subset of production ruIes for context free grammar is subset of production rules for context free grammar.

Although it is noted that any grammar that will also context free grammar but the converse is not always true i.e. some production which follow the CFG property ($A \rightarrow aBB$) do not follow the regular grammar property.

Example 5.8: Which of the following production are under the context free grammar (CFG) for

Given $V_N = \{A, B, C\}$

And $\sum = \{0, 1\}$:- A is a start symbol

(a) $A \rightarrow 0AB$

(b) $A \rightarrow AB0$

(c) $A \rightarrow AOC$

(d) $A \rightarrow 0CB$

(e) $A \rightarrow 0$

(f) $AB \rightarrow 0C$

(g) $BC \rightarrow 1A$

(h) $A1 \rightarrow 1B$

Solution: From (a) to (e) are production rule for context free grammar because left hand side are only one variable and right side have element of $(V_N \cup \Sigma)^*$

(f), (g)), (h) are not production rule for context free grammar because left hand side are not only one variable or non-terminal.

(iii) Type-1 Grammar or context sensitive Grammar

These types of grammar follows the following rule of production:-

$$\oint A \psi \rightarrow \oint \alpha\ \psi$$

Where,

- A is variable (non-terminal).
- $\alpha \neq \wedge$

- $\oint$ (left context)and ψ(right context) are string of terminals and non-terminals
- Erasing A is not permitted.

Example 5.9: Show that the following production are under the context free grammar (CFG) for Given V_N = {A, B, D} and $\sum$ = {a,b,c} where, A is a start symbol

(a) aAbcD ⟶ abcDbcD

(b) AB ⟶ AbBc

(c) A ⟶ abA

(d) ABD ⟶ AaBb

(e) ab ⟶ ba

Solution:

(a) Suppose the production is in the form of $\oint A\psi \rightarrow \oint \alpha\ \psi$, where $\oint$ is ab, ψ is bcd and A is replaced by bcD $\neq$ ^. So, the above production rule is under context sensitive grammar.

(b) Suppose the production is in the form of $\oint A\psi \rightarrow \oint \alpha\ \psi$, where $\oint$ is A, ψ is ^ and α is bBc. So, the above production rule is under context sensitive grammar.

(c) Suppose the production is in the form of $\oint A\psi \rightarrow \oint \alpha\ \psi$, where $\oint$ is ^, ψ is ^ and α is abA. So, the above production rule is under context sensitive grammar.

(d) Suppose the production is in the form of $\oint A\psi \rightarrow \oint \alpha\ \psi$, where $\oint$, ψ are not possible. So, the above production rule is not under context sensitive grammar.

(e) Since the production rules does not contain any variable, it is not under context sensitive grammar.

(iv) Type - 0 Grammar or Unrestricted Grammar

A type 0 grammar is any phrase structure grammar without any restrictions. All the grammar we have considered are as type 0 grammar.

So, these types of grammar follows the following rule of production:-

$$\alpha \rightarrow \beta$$

Where,

$\alpha \in (V_N \cup \sum)^*$ which contain at least one variable.

$\beta \in (VN \cup\sum)^*$

Example 5.10: Show that the following production are under the unrestricted grammar (CFG) for Given $V_N = \{A, B, D\}$ and $\sum = \{a,b,c\}$ where, A is a start symbol

(a) ab ⟶ ba

Solution:

(a) This production rule are not grammar or unrestricted grammar because left side of the production should contains at least one non-terminal.

5.4 Languages and Their Relations

So far we studied different types of grammars i.e. regular grammar, context-free grammar, context-sensitive grammars and unrestricted grammar. The languages generated by these grammars are shown as the following:

Grammar	**Formal Language**
Regular Grammar or Type-3	Regular Language (L_{REG})
Context Free Grammar or Type-2	Context free Language (L_{CF})
Context-sensitive or Type-1	Context Sensitive Language(Lcs)
Unrestricted Grammar or Type-1	Recursively Enumerable Language (L_{RE})

The relation between these languages can be described by Noam Chomsky (Founder of formal language) called Chomsky hierarchy. According to Chomsky hierarchy

$$L_{REG} \subseteq L_{CF} \subseteq Lcs \subseteq L_{RE}$$

This can be nested as shown in Figure 5.1.

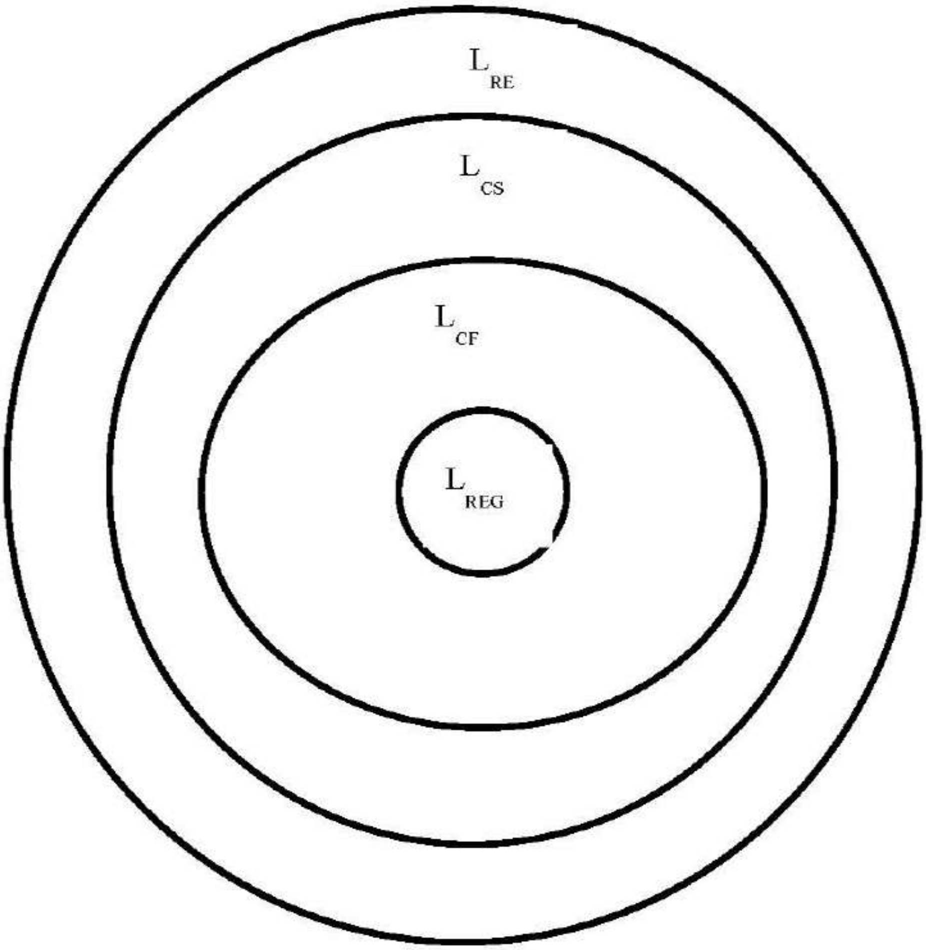


Figure 5.1: Relation between the Formal Languages

Other language families can be fitted in the picture like deterministic context-free language (L_{DCF}) and recursive languages (L_{RE}). The relationship among language including these languages can be structure using the fig.

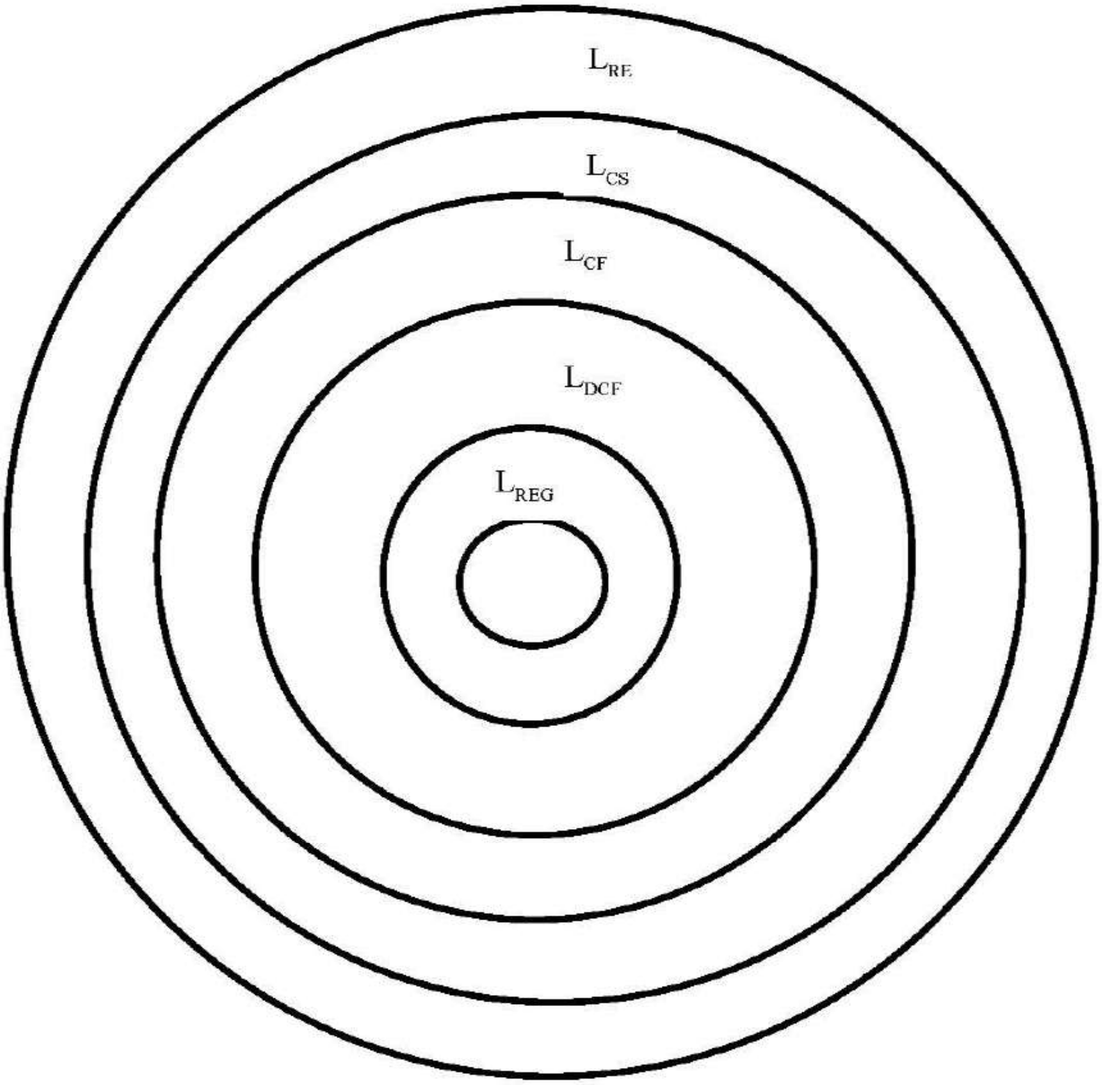


Figure 5.2: Relation among different Languages

Recursive and Recursively Enumerable Language:-

These type of languages are associated with turning machine. These can be defined as:-

Definition of Recursively Enumerable Language:-

A language L is said to be ***Recursively Enumerable*** if there exists a Turning machine that accept it.

The definition of Recursively Enumerable implies that there exists a Tuning machine M such that for every $w \in L$, q_0w --|* $x_1q_fx_2$ with q_f is final state.

The definition says nothing about what happens for w not in L: it may be that the machine halts in non-final state or that it never halts and goes into an infinite loop.

Definition of Recursive Language:-

A language L on $\sum$ is said to be ***recursive*** if there exists a Turing Machine M that accepts L and that halts on every w in $\sum^{+}$.

5.5 Operations on Languages

In this section we will discuss various operations that are applied on the language and their effect when applied to these languages. Some of the common operations are:

(i) **Concatenation Operation**: Let A and B be any sets of strings, The Concatenation of A and B is defined by

$$AB= \{uv \mid u \in A, C \in B\}$$

Here, uv is the concatenation of the string u and v.

Example 5.11: $L_1= \{a, ab\}$ $L_2= \{b, ba\}$ then $L_1L_2=\{ab, baa, abb, abba\}$

Some properties of language w.r.t concatenation operation:-

(i) $L_1L_2 \neq L_2L_1$ in general

(ii) $L\emptyset = \emptyset$

(iii) $L\{\in\} = L\{\in\}L$

(ii) **Transpose Operation**: Let A and B be any sets of strings. Then the transpose operation A^T is defined as

$$A^T=\{u^T \mid u \in A\}$$

(iii) **Union Operation**: If L_1 & L_2 are the two languages then the union of L_1 & L_2 denoted by $L_1 \cup L_2$ such that any word $x \in L_1 \cup L_2$, iff $x \in L1$ or $x \in L2$.

Example 5.12: Suppose $L_1 = \{0,11,01,011\}$ and $L_2= \{0, 01,011\}$ then,

$L_1 \cup L_2 = \{0, 11, 01, 011, 111\}$

(iv) **Intersection Operation**: If L_1 and L_2 are the two languages then the intersection of L_1 and L_2 is denoted by $L_1 \cap L_2$ such that any word $x \in L \cap L$, iff $x \in L_1$ and $x \in L_2$.

Example 5.13: Suppose $L_1 = \{0,11,01,011\}$ and $L_2 = \{0, 01,011\}$ then

$L_1 \cap L_2 = \{01\}$

(v) **Complement Operation**: Usually Σ^* is referred as the universe of all the language s over the alphabets over Σ. So, complement of any language is taken with respect to Σ^*. Thus for a language L, the complement is denoted by L'.

$$L' = \{x \in \Sigma^* \text{ and } x \notin L\}.$$

Example 5.14: Let $L' = \{x|\ |x| \text{ is even}\}$ then its component

$L' = \{x|\ |x| \text{ is odd}\}$

(iv) **Kleene's star operation**: The kleene star operation on the language L, denoted as L* is defined as

$L^* = L0 \cup L1 \cup L2\ldots\ldots$

Or

$L^* = \{x \mid x \text{ is concatenation of zero or more string}\}$

Closure Properties of Languages: Closure property is a helping technique to know the class of the resulting language when we do an operation on two languages of the same class. Suppose L_1 and L_2 belong to CFL and if CFL is closed under operation ∪, then $L_1 \cup L_2$ will be a CFL. But if CFL is not closed under ∩, that doesn't mean $L_1 \cap L_2$ won't be a CFL. For a class to be closed under an operation, it should hold true for all languages in that class. So, if a class is not closed under an operation, we cannot say anything about the class of the resulting language of the operation – it may or may not belong to the class of the operand languages. In short, closure property is applicable, only when a language is closed under an operation.

Closure properties of different languages can be summarized:

Table 5.1

Type	Union	Concatenation	Transpose	Intersection	Complement
Recursive Language	YES	YES	YES	YES	YES
Recursively Enumerable Language	YES	YES	YES	YES	NO
Context Sensitive Language	YES	YES	YES	YES	YES
Context-Free Language	YES	YES	YES	NO	NO
Deterministic Context-Free Language	NO	NO	NO	NO	YES
Regular Language	YES	YES	YES	YES	YES

5.6 Languages and Automata

Previously, we discussed about the relationship between the language and grammar. In this section we will discuss about the relationship between language and automata. The relationship between the same is described in the following table:

Table 5.2

Language	Automata
Unrestricted or Type-0	Turning Machine
Context-sensitive Language or Type-1	Linear Bond Automata
Context-Free Language or Type-2	Push down Automata
Regular Language orType-3	Finite Automata

The above table suggest that the unrestricted language, context sensitive language, context free language and regular languages are accepted by Turning Machine, Linear Bound Automata, and Push down Automata and Finite Automata respectively.

5.7 Self Learning Exercise

Q.1 Which of the following language is recognized by TM?

a) Regular Language

b) Context Free Language

c) Context-sensitive Language

d) All of the Above

Q.2 Context-Free Language is NOT closed under:

a) Union operation.

b) Concatenation operation

c) Intersection operation.

d) None of the above.

Q.3 $\{a^n b^n c^m \mid n, m >= 1\}$ is

a) regular.

b) context-free but not regular.

c) context-sensitive but not context-free.

d) none of these.

Q.4 If a grammar G has three productions S -> aSa | bSb | c, then

a) abcba and bacab ∈ L(G)

b) abcba and abcab ∈ L(G)

c) accca and bcccb ∈ L(G)

d) acccb and bccca ∈ L(G)

5.8 Summary

In this chapter you learned about various languages and its corresponding automata. This can be summarized as:-

A grammar is characterized by four-tuples (V_N, Σ, P, S) where,

- V_N is finite non empty set of elements called variables.
- Σ is finite non empty set of elements called terminals.
- S is a element of V_N is a special variable called start symbol.
- P is a finite set whose elements are $\alpha \longrightarrow \beta$, where α and β are string on $V_N \cup \Sigma$. α has at least one symbol from V_N. The elements of P are called production rules.

According to Chomsky (Name of Scientist) there are four types of grammar:-

- Type-3 Grammar or Regular Grammar
- Type-2 Grammar or Context Free Grammar
- Type-1 Grammar or Context Sensitive Grammar
- Type-0 Grammar or Unrestricted Grammar

The languages generated by these grammars are shown as the following:

Grammar	**Formal Language**
Regular Grammar or Type-3	Regular Language (L_{REG})
Context Free Grammar or Type-2	Context free Language (L_{CF})
Context-sensitive or	Context Sensitive Language(Lcs)

Type-1	
Unrestricted Grammar or Type-1	Recursively Enumerable Language (L_{RE})

Closure properties of different languages can be summarized:

Type	Union	Concatenation	Transpose	Intersection	Complement
Recursive Language	YES	YES	YES	YES	YES
Recursively Enumerable Language	YES	YES	YES	YES	NO
Context Sensitive Language	YES	YES	YES	YES	YES
Context-Free Language	YES	YES	YES	NO	NO
Deterministic Context-Free Language	NO	NO	NO	NO	YES
Regular Language	YES	YES	YES	YES	YES

The relationship between the same is described in the following table:

Language	Automata
Unrestricted or Type-0	Turning Machine
Context-sensitive Language or Type-1	Linear Bond Automata
Context-Free Language or Type-2	Push down Automata

Regular Language orType-3	Finite Automata

5.9 Answers to Self-Learning Exercise

Q.1 (d)

Q.2 (c)

Q.3 (a)

5.10 Exercise

Q.1 Show that the family of context sensitive language in closed under union.

Q.2 Explain the Chomsky Classification of Language in detail.

Q.3 Define Grammar. Also explain types of Grammar and Languages in TOC.

Q.4 Construct a context-free grammar generating

(a) $L_1= \{a^n b^{2n} \mid n>=1\}$

(b) $L_2 = \{a^m b^n \mid m> n, n>=1 \}$

(c) $L_3 = \{ a^m b^n \mid m< n, n>=1 \}$

(d) $L_4. = \{ a^m b^n \mid m, n >=0, m \neq n \}$

5.11 References and Suggested Readings

1. Peter Linz, An Introduction to Formal Languages and Automata, Jones and Bartlett Publication, Third Edition.
2. K.L.P. Mishra, N. Chandrasekaran, Theory of Computer Science, BPB Publication, Prentice-Hall of India, Second Edition.
3. H.E. Hopcraft and J.D. Ullamn, Introduction to Automata Theory, Languages and Computation, Narosa Publications.
4. J.C. Martin, Introduction to Languages and the Theory of Automata, Tata McGraw-Hill.

UNIT 6

Finite Automata and Regular Expressions

Structure of the Unit

6.0 Objective

After reading this chapter you will be able to understand the following:

- Basic Meaning of Regular Expressions
- Significance of Finite Automata and its regular expressions.
- Non-deterministic Finite Automata and its regular expressions.
- Arden's theorem

6.1 Introduction

A finite-state machine (FSM) or finite-state automaton (FSA, plural: automata), finite automaton, or simply a state machine, is a mathematical model of computation. It is an abstract machine that can be in exactly one of a finite number of states at any given time. The FSM can change from one state to another in response to some external inputs; the change from one state to another is called a transition. An FSM is defined by a list of its states, its initial state, and the conditions for each transition.The concept of grammar is very important to understand the theory of computation.

A regular expression is, in theoretical computer science and formal language theory, a sequence of characters that define a search pattern. Usually this pattern is then used by string searching algorithms for "find" or "find and replace" operations on string. Regular expressions are used in search engines, search and replace dialogs of word processors and text editors, in text processing utilities such as sed and AWK and in lexical analysis.

6.2 Regular Expressions

The regular expressions are useful for representing certain sets of strings in an algebraic fashion. Actually these describe the languages accepted by finite state automata.

A formal recursive definition of regular expressions over $\sum$ as follows:

1. Any terminal symbol (i.e. an element of $\sum$), A and Ø are regular expressions.

   When we view a in $\sum$ as a regular expression, we denote it by **a**.
2. The union of two regular expressions R_1 and R_2 written as R_1+R_2, is also a regular expression.
3. The concatenation of two regular expressions R_1 and R_2, written as R_1 R_2, is also a regular expression.
4. The iteration (or closure) of a regular expression R written as R*, is also a regular expression.
5. If R is a regular expression, then (R) is also a regular expression.
6. The regular expressions $\sum$ over are precisely those obtained recursively by the application of the rules 1-5 once or several times.

Definition 5.1 Any set represented by a regular expression is called a *regular set.*

If for example, *a, b € L.* then (i) a denotes the set *{a},* (ii) a + b denotes *{a, b},* (iii) **ab** denotes *{ab},* (iv) a* denotes the set {A. *a, aa. aaa,* ...} and (v) (a + b)* denotes *{a, b}*.*

The set represented by *R* is denoted by ***L(R),***

Now we shall explain the evaluation procedure for the three basic operations. Let R_1 and R_2 denote any two regular expressions. Then (i) a string in $L(R_1+R_2)$is a string from R_1 or a string from R_2 (ii) a string in $L(R_1R_2)$ is a string from R_1 followed by a string from R_2 and (iii) a string in L(R*) is a string obtained by concatenating *n* elements for some *n*>=0.

Consequently, (i) the set represented by R_1+ R_2 is the union of the sets represented by R_1 and R_2 (ii) the set represented by R_1R_2 is the concatenation of the sets represented by R_1 and R_2.

EXAMPLE 6.1--Describe the following sets by regular expressions: (a) {101} (b) *{abba},* (c) {01, 10}, (d) {^, *ab},* (e) *{abb. a, b, bba},* (f) {^, 0, 00, 000....}, and (g) {1, 11, 111 ... }.

Solution

(a) Now, {l}, {0} are represented by 1 and 0. respectively. 101 is obtained by concatenating 1, 0 and 1. So, {101} is represented by 101.

(b) abba represents *{abba}.*

(c) As {01, 10} is the union of {01} and {10}, we have {01,10} represented by 01 + 10.

(d) The set {^, *ab}* is represented by ^ + ab.

(e) The set *{abb, a, b, bba}* is represented by abb + a + b + bba.

(f) As {^, 0, 00, 000, ... } is simply {0}*. It is represented by 0*.

(g) Any element in {1, 11, 111 ... } can be obtained by concatenating 1 and any element of {l}*. Hence 1(1)* represents {1, 11, 111, ...}.

IDENTITIES FOR REGULAR EXPRESSIONS

Two regular expressions P and Q are equivalent (we write P =Q) if P and Q represent the same set of strings. We now give the identities for regular expressions; these are useful for simplifying regular expressions.

I_1 Ø + R =R

I_2 Ø R =R Ø = Ø

I_3 ^R =R^ =R

I_4 ^* = and Ø * = ^

I_5 R+R=R

I_6 R*R* = R*

I_7 RR* = R*R

I_8 (R*)* =R*

I_9 ^ + RR* = R* = ^ + R*R

I_{10} (PQ)* P = P (QP)*

6.3 Finite Automata and Regular Expressions

The transition systems can be generalized by permitting A-transitions or ^-moves which are associated with a null symbol ^. These transitions can occur when no input is applied. But it is possible to convert a transition system with ^-moves into an equivalent transition system without ^-moves. We shall give a simple method of doing it with the help of an example.

Suppose we want to replace a ^-move from vertex v_1 to vertex v_2. Then we proceed as follows:

Step 1 Find all the edges starting from v_2.

Step 2 Duplicate all these edges starting from v_1 without changing the edge labels.

Step 3 If v_1 is an initial state; make v_2 also as initial state.

Step 4 If v_2 is a final state. Make v_1 also as the final state.

EXAMPLE 6.1 Consider a finite automaton, with ^-moves, given in Fig. 6.1. Obtain an equivalent automaton without ^-moves.

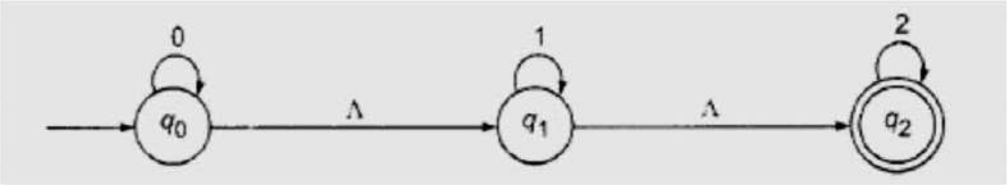


Fig. 6.1 Finite automaton of Example 6.1

Solution

We first eliminate the ^-move from q_0 ta q_1 to get Fig. 6.2(a). q_1 is made an initial state. Then we eliminate the ^-move from q_0 to q_2 in Fig. 6.2(a) to get Fig. 6.2(b). As q_2 is a final state, q_0 is also made a final state. Finally, the ^-move from q_1 to q_2 is eliminated in Fig. 6.2(c).

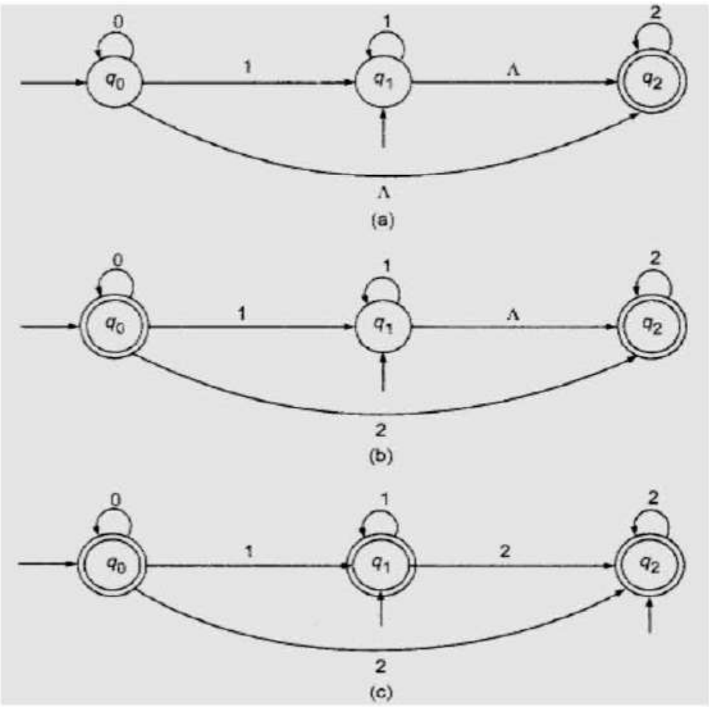


Figure 6.2 Transition System for example 6.1, without ^-moves

EXAMPLE 6.2

Consider a graph (i.e. transition system), containing a ^-move, given in Fig. 6.3. Obtain an equivalent graph (i.e. transition system) without ^-moves.

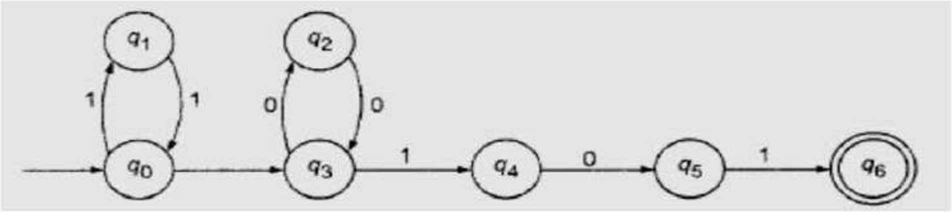


Fig. 6.3 Finite automaton of Example 6.3

Solution

There is a ^-move from *q0* to *q3*. There are two edges, one from *q3* to *q2* with label 0 and another from *q3* to *q4*. Duplicate these edges can from *q0.* As *q0* is an initial state, *q3* is made an initial state. The resulting transition graph is given in Fig. 6.4.

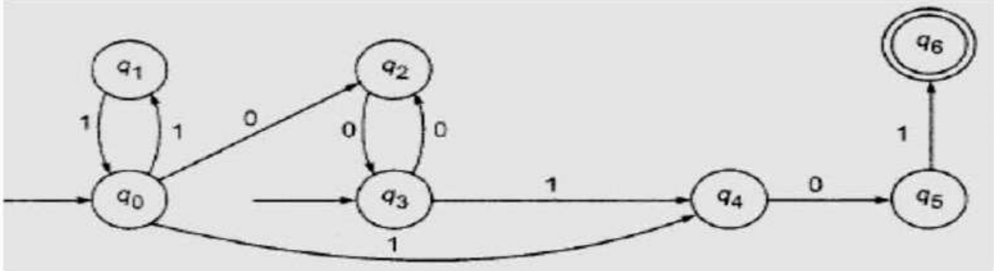


Fig. 6.4 Transition system for Example 6.2, without ^-moves.

6.4 NDA with Null moves and Regular Expressions

In this section, we prove that every regular expression is recognized by a nondeterministic finite automaton (NDFA) with ^-moves.

Theorem (Kleene's theorem):- If **R** is a regular expression over $\sum$ representing $L \subseteq \sum^*$, then there exists an NDFA *M* with ^-moves such that *L* = *T(M).*

Proof The proof is by the principle of induction on the total number of characters in R. By 'character' we mean the elements of $\sum$, ^, Ø, * and +. For example, if R =^ + **10*11*0,** the characters are ^, +, 1, 0, *. 1, 1, *, 0, and the number of characters is 9.

Let *L(R)* denote the set represented by R.

Basis. Let the number of characters in R be 1. Then R =^, or R = Ø, or R =a_i, $a_i \in \sum$. The transition systems given in Fig. 6.5 will recognize these regular expressions.

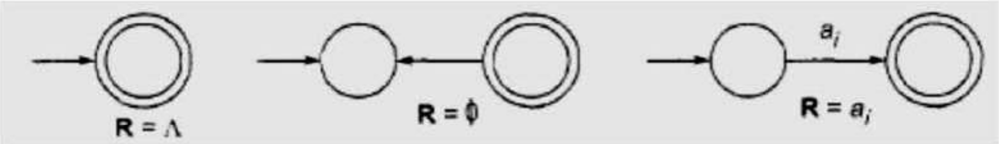


Fig. 6.5 Transition systems for recognizing elementary regular sets.

Induction step: Assume that the theorem is true for regular expressions having *n* characters. Let R be a regular expression having *n* + 1 characters. Then,

R=P+Q or R = **PQ** or R = p*

according as the last operator in R is +, product or closure. Also P and Q are regular expressions having *n* characters or less. By induction hypothesis, *L(P)* and *L(Q)* are recognized by M_1 and M_2: where M_1 and M_2: are NDFAs with ^-moves, such that *L(P)* = $T(M_1)$ and *L(Q)* = $T(M_2)$. M_1 and M_2 are represented in Fig. 6.6.

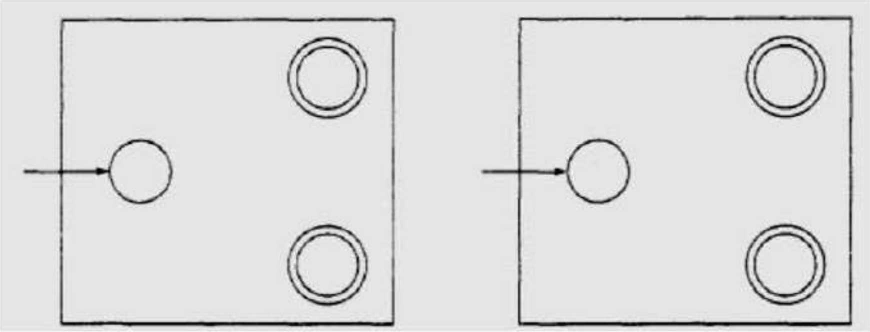


Fig. 6.6 Nondeterministic finite automata M1 and *M2*.

The initial state and the final states of M_1 and M_2 are represented in the usual way.

***Case* 1 R =P + Q**. In this case we construct an NDFA *M* with ^-moves that accepts *L(P* + Q) as follows: q_0 is the initial state of *M,* q_0 not in M_1 or M_2, q_f is the final state of *M:* once again q_f not in M_1 or M_2. *M* contains all the states of M_1 and M_2. and also their transitions. We add additional ^-transitions from q_0 to the initial states of M_1 and M_2 and from the final states of M_1 and M_2 to q_f. The NDFA *M* is as in Fig. 6.7. It is easy to see that $T(M) = T(M_1) \cup T(M_2) = L(P + Q)$.

***Case* 2 R =PQ**. In this case we introduce q_0 as the initial state of *M* and q_f as the final state of *M.* both q_0, q_f not in M_1 or M_2. New A-transitions are added between q_0 and the initial state of M_2, between final states of M_1 and the initial state of M_2 and between final states of M_2 and the final state q_f of *M.* See Fig. 6.8.

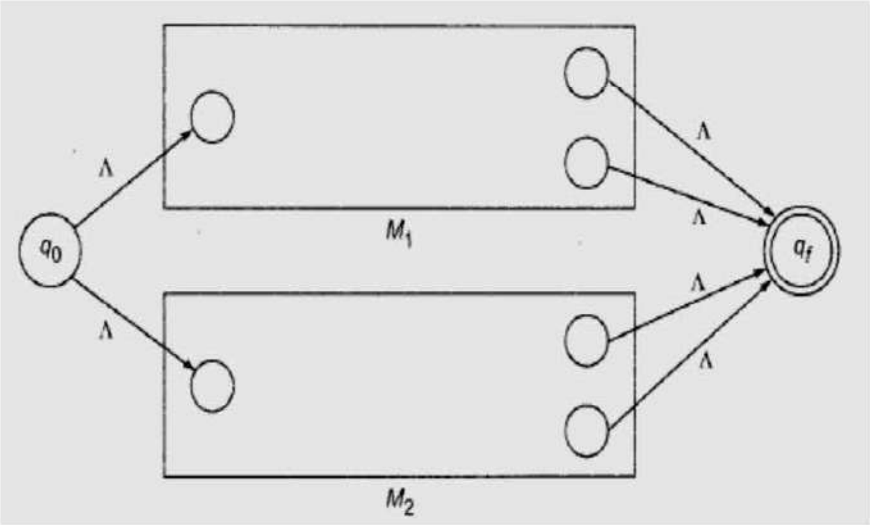


Fig. 6.7 NDFA accepting *L(P* + Q).

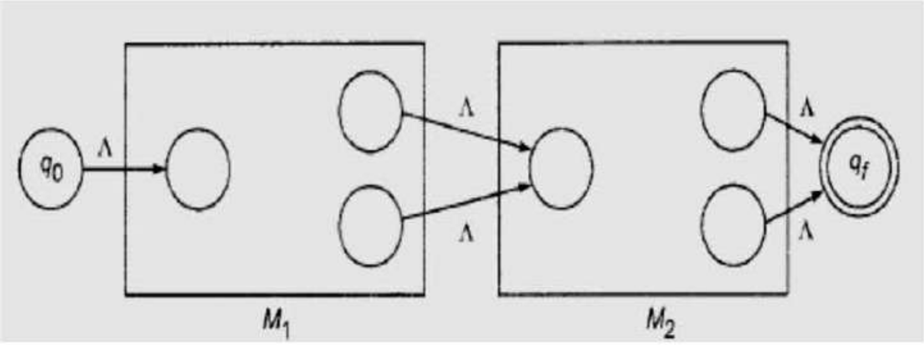


Fig. 6.8 NDFA accepting *L(PQ).*

***Case* 3 R = (P)*.** In this case, q_0, q and q_f are introduced. New ^-transitions are introduced from q_0 to q, q to q_f, q to the initial state of M_1 and from the final states of M_1 to q. See Fig. 6.9.

Thus in all the cases, there exists an NDFA *M* with ^-moves, accepting the regular expression **R** with $n + 1$ characters. By the principle of induction, this theorem is true for all regular expressions.

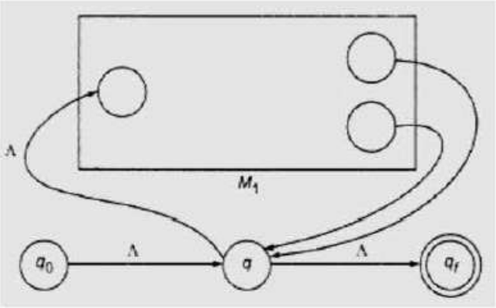


Fig. 6.9 NDFA accepting *L(P*)*.

Above theorem gives a method of constructing NDFAs accepting P + Q, **PQ** and P* using the NDFAs corresponding to P and Q. Thus, if a regular expression P is given, we can construct a DFA accepting *L (P)*.

EXAMPLE 6.3

Obtain the deterministic graph (system) equivalent to the transition system given in Fig. 6.10

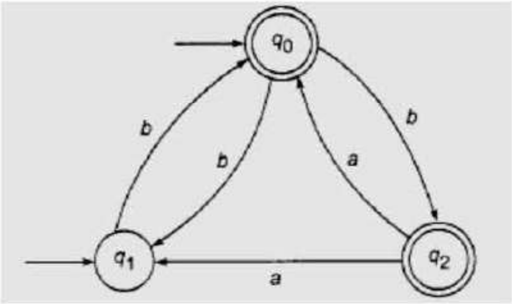
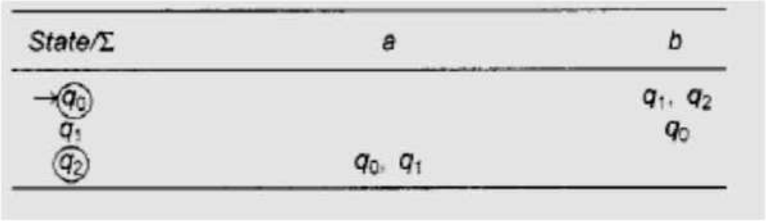


Fig. 6.10 Nondeterministic transition system of Example 6.3

Solution

We construct the transition table corresponding to the given nondeterministic system. It is given in Table 6.1.

TABLE 6.1 Transition Table for Example 6.3

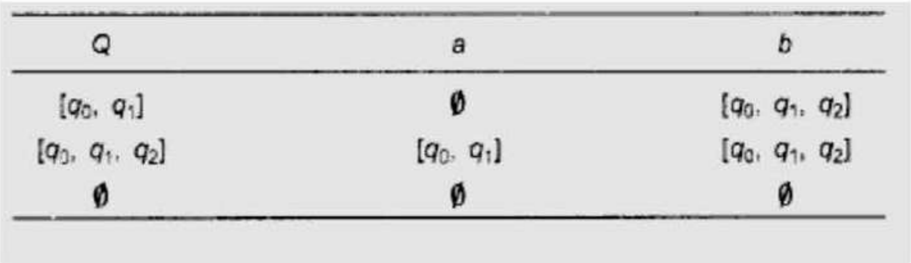


State/Σ	*a*	*b*
$\rightarrow q_0$		q_1, q_2
q_1		q_0
q_2	q_0, q_1	

We construct the successor table by starting with $[q_0, q_1]$. From Table 6.1 we see that $[q_0, q_1, q_2]$ is reachable from $[q0, q_1]$ by a b-path. There are no a-paths from $[q0, q_1]$. Similarly, $[q0, q_1]$ is reachable from $[q_0, q_1, q2]$ by an a-path and $[q0, q_1, q2]$ is reachable from itself. We proceed with the construction for all the elements in Q'.

We terminate the construction when all the elements of Q' appear in the successor table. Table 6.2 gives the successor table. From the successor table it is easy to construct the deterministic transition system described by Fig. 6.11.

TABLE 6.2 Deterministic Transition Table for Example 6.3



Q	a	b
$[q_0, q_1]$	$\emptyset$	$[q_0, q_1, q_2]$
$[q_0, q_1, q_2]$	$[q_0, q_1]$	$[q_0, q_1, q_2]$
$\emptyset$	$\emptyset$	$\emptyset$

as $q0$ and $q2$ are the final states of the nondeterministic system $[qo, q_1]$ and $[q_0, q_1 ,q2]$ are the final states of the deterministic system.

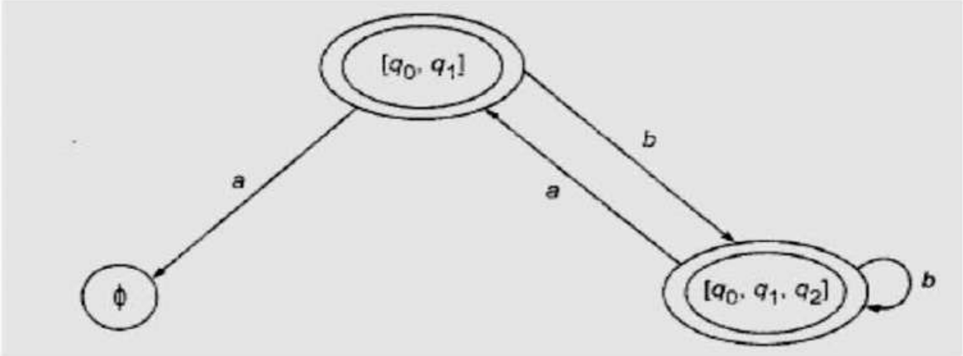


Fig. 6.11 Deterministic transition system for Example 6.3

6.5 Algebraic method using Arden's Theorem

The following method is an extension of the Arden's theorem. This is used to find the r.e. recognized by a transition system.

The following assumptions are made regarding the transition system:

(i) The transition graph does not have ^-moves.

(ii) It has only one initial state, say v_1.

(iii) Its vertices are $v_1 \ldots\ldots v_n$.

(iv) V_i the r.e. represents the set of strings accepted by the system even though v_i is a final state.

(v) $\propto_{ij}$ denotes the r.e. representing the set of labels of edges from v_i to v_j. When there is no such edge, $\propto ij == Ø$. Consequently, we can get the following set of equations in V1…..Vn.

$$V_1 = V_1 \propto_{11} + V_2 \propto_{21} + \ldots\ldots.+ V_n \propto_{n1} + \wedge$$

$$V_2 = V_1 \propto_{12} + V_2 \propto_{22} + \ldots\ldots.+ V_n \propto_{n2} + \wedge$$

.

.

$$V_n = V_1 \propto_{1n} + V_2 \propto_{2n} + \ldots\ldots.+ V_n \propto_{nn} + \wedge$$

By repeatedly applying substitutions and Theorem 5.1 (Arden's theorem), we can express V_i in terms of a_{ij}*'s*.

For getting the set of strings recognized by the transition system, we have to take the 'union" of all Vi's corresponding to final states.

EXAMPLE 6.4

Consider the transition system given in Fig. 6.12. Prove that the strings recognized are (a + a(b + aa)*b)* arb + aa)* a.

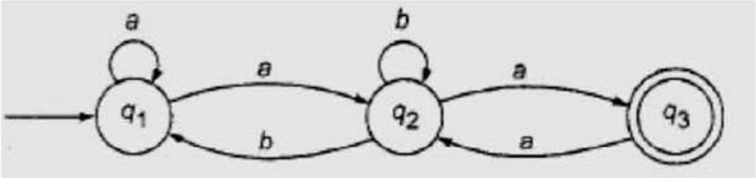


Fig. 6.12 Transition system of Example

Solution

We can directly apply the above method since the graph does not contain any ^-move and there is only one initial state.

The three equations for q_1, q_2 and q_3 can be written as

$q_1 = q_1a + q_2b + \wedge$, $q2 = q_1a + q_2b + q_3a$, $q_3 = q_2a$

It is necessary to reduce the number of unknowns by repeated substitution. By substituting q_3 in the q_2-equation. We get by applying Arden's theorem.

$q2 = q_1a + q_2b + q_2aa$

$= q_1a + q_2(b + aa)$

$= q_1a(b + aa)^*$

Substituting q_2 in q_1 we get

$q_1 = q_1a + q_1a(b + aa)^*b + \wedge$

$= q_1(a + a(b + aa)^*b) + \wedge$

Hence,

$ql = \wedge(a + a(b + aa)^*b)^*$

$q2 = (a + a(b + aa)^*b)^* a(b + aa)^*$

$q3 = (a + a(b + aa)^*b)^* a(b + aa)^*a$

Since q_3 is a final state, the set of strings recognized by the graph is given by

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

EXAMPLE 6.5

Prove that the finite automaton whose transition diagram is as shown in Fig. 6.13 accepts the set of all strings over the alphabet *{a, b}* with an equal number of *a's* and *b's,* such that each prefix has at most one more *a* than the b's and at most one more *b* than the *a's.*

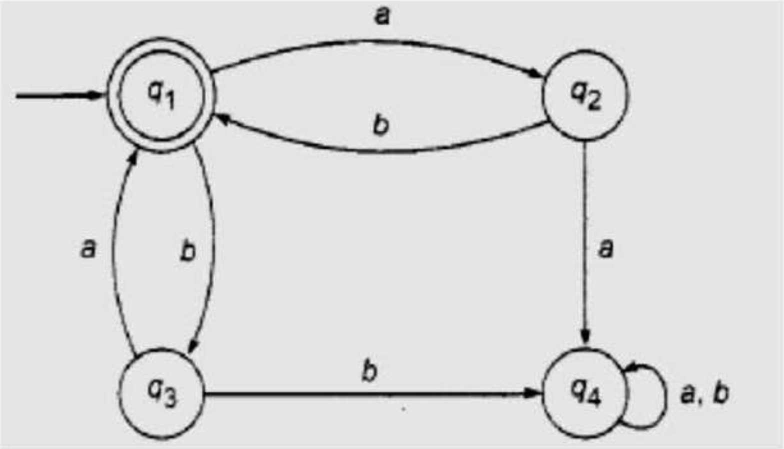


Fig. 6.13 Finite automaton of Example 6.5

Solution

We can apply the above method directly since the graph does not contain the A-move and there is only one initial state. We get the following equations for q_1, q_2. q_3 , q_4:

$q_1 = q_2b + q_3a + \wedge$

$q_2 = q_1a$

$q_3 = q_1b$

$q_4 = q_2a + q_3b + q_4a + q_4b$

As q_1 is the only final state and the q_1-equation involves only q_2 and q_3. We use only q_2- and q_3 - equations (the q_4-equation is redundant for our purposes). Substituting for q2 and q3' we get

$q_1 = q_1ab + q_1ba + \wedge = q_1(ab + ba) + \wedge$

By applying Arden's theorem, we get

$q_1 = \wedge(ab + ba)^* = (ab + ba)^*$

As q_1 is the only final state, the strings accepted by the given finite automaton are the strings given by (ab + ba)*. As any such string is a string of *ab's,* and *ba's,* we get an equal number of *a's* and *b's.* If a prefix *x* of a sentence accepted by the finite automaton has an even number of symbols, then it should have an equal number of *a's* and *b's* since *x* is a substring formed by *ab's* and *ba's.* If the prefix *x* has an odd number of symbols, then we can write *x* as *ya* or *yb.* As *y* has an even number of symbols, *y* has an equal number of *a's* and b's. Thus, *x* has one more *a* than *b* or vice versa.

EXAMPLE 6.6

Describe in English the set accepted by the finite automaton whose transition diagram is as shown in Fig. 6.14.

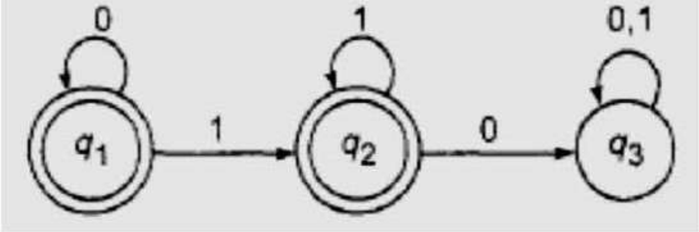


Fig. 6.14 Finite automaton of Example 6.6

Solution

We can apply the above method directly as the transition diagram does not contain more than one initial state and there are no A-moves. We get the following equations for q_1, q2, q3,

$q_1 = q_1 0 + \wedge$

$q2 = q_1 1 + q_2 1$

$q3 = q_2 0 + q_3 (0 + 1)$

By applying Arden's theorem to the q_1-equation, we get

$q_1 = \wedge 0^* = 0^*$

So, $q2 = q_1 1 + q_2 1 = 0^* 1 + q_2 1$

Therefore,

$q_2 = (0^* 1) 1^*$

As the final states are q_1 and q_2, we need not solve for q_3:

$q_1 + q_2 = 0^* + 0^*(11^*) = 0^*(\wedge + 11^*) = 0^*(1^*)$ by I_9

The strings represented by the transition graph are 0*1*. We can interpret the strings in the English language in the following way: The strings accepted by the finite automaton are precisely the strings of any number of 0's (possibly ^) followed by a string of any number of l's (possibly ^).

EXAMPLE 6.7

Construct a regular expression corresponding to the state diagram described by Fig. 6.15

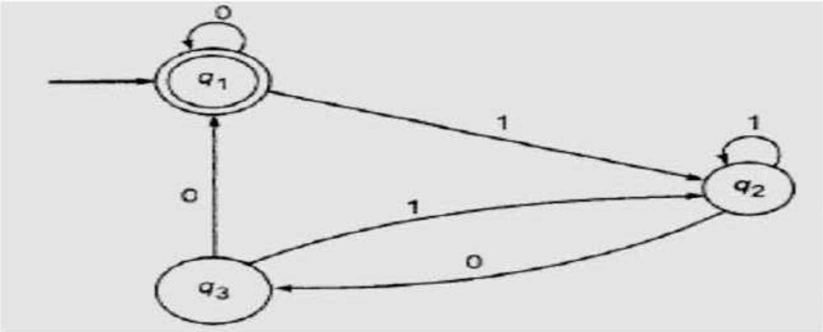


Fig. 6.15 Finite automaton of Example 6.7

Solution

There is only one initial state. Also, there are no ^-moves. The equations are

$q_1 = q_1 0 + q_3 0 + \wedge$

$q_2 = q_1 1 + q_2 1 + q_3 1$

$q_3 = q_2 0$

So,

$q_2 = q_1 1 + q_2 1 + (q_2 0) 1 = q_1 1 + q_2 (1 + 0 1)$

By applying Arden's theorem we get

$q_2 = q_1 1 \ (1 + 0 1)^*$

Also,

$q_1 = q_1 0 + q_3 0 + \wedge = q_1 0 + q_2 00 + \wedge$

$\quad = q_1 0 + (q_1 1 (1 + 0 1)^*) 00 + \wedge$

$\quad = q_1 (0 + 1 (1 + 0 1)^* \ 00) + \wedge$

Once again applying Arden's theorem, we get

$q1 = \wedge (0 + 1 (1 + 0 1)^* \ 00)^* = (0 + 1 (1 + 0 1)^* \ 00)^*$

As q_1 is the only final state, the regular expression corresponding to the given diagram is $(0 + 1 (1 + 0 1)^* \ 00)^*$.

EXAMPLE 6.8

Find the regular expression corresponding to Fig. 6.8.

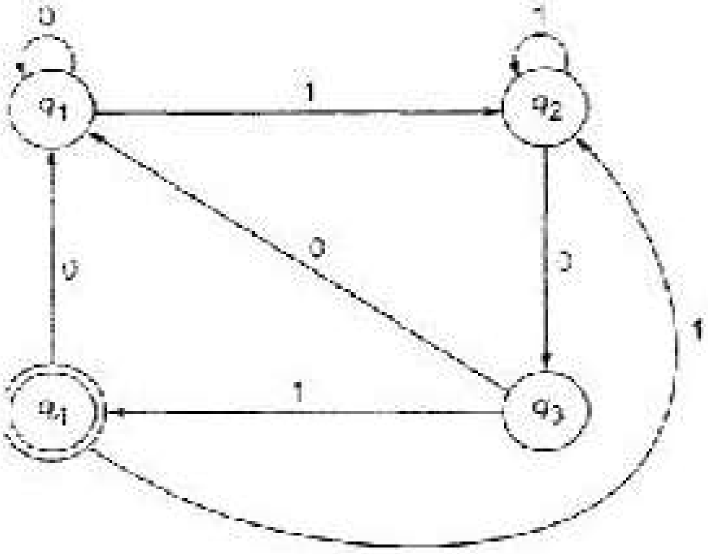


Fig. 6.18 Finite automaton of Example 6.8

Solution

There is only one initial state. and there are no A-moves. So, we form the equations corresponding to q_1, q_2, q_3, q_4:

$$q_1 = q_1 0 + q_3 0 + q_4 0 + \wedge$$

$$q2 = q_1 1 + q_2 1 + q_4 1$$

$$q_3 = q_2 0$$

$$q_4 = q_3 1$$

Now

$$q_4 = q_3 1 = (q_2 0)1 = q_2 01$$

Thus, we are able to write q3, q4 in terms of q2. Using the q2-equation, we get

$$q2 = q_1 1 + q_2 1 + q_2 011 = q_1 1 + q_2(1+011)$$

By applying Arden's Theorem, we obtain

$$q2 = q_1 1\ (1+011)^* = q_1\ (1(1+011)^*)$$

From the q_1-equation, we have

$$q_1 = q_1 0 + q_2 00 + q_2 010 + \wedge$$

$$= q_1 0 + q_2(00 + 010) + \wedge$$

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

Again, by applying Arden's Theorem, we obtain

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

$q_4 = q_2 01 = q_1 1(1 + 011)^*\ 01$

$= (0 + 1(1 + 011)^*(00 + 010))^*(1(1 + 011)^*\ 01)$

6.6 Self Learning Exercise

Q.1 Which of the following pair of regular expression are not equivalent?

a) 1(01)* and (10)*1

b) x (xx)* and (xx)*x

c) (ab)* and a*b*

d) x+ and x*x+

Q.2 A language is regular if and only if

a) accepted by DFA

b) accepted by PDA

c) accepted by LBA

d) accepted by Turing machine

Q.3 Which of the following is true?

a) (01)*0 = 0(10)*

b) (0+1)*0(0+1)*1(0+1) = (0+1)*01(0+1)*

c) (0+1)*01(0+1)*+1*0* = (0+1)*

d) All of the mentioned

6.7 Summary

A deterministic finite automaton (DFA) also known as a deterministic finite acceptor (DFA) and a deterministic finite state machine (DFSM) is a finite-state machine that accepts and rejects strings of symbols and only produces a unique computation (or run) of the automaton for each input string. A DFA is defined as an abstract mathematical concept, but is often implemented in hardware and software for solving various specific problems. DFAs recognize exactly the set of regular languages, which are, among other things, useful for doing lexical analysis and pattern matching.

A regular expression is a sequence of characters that define a search pattern. Usually this pattern is then used by string searching algorithms for "find" or "find and replace" operations on strings.

6.8 Answers to Self-Learning Exercise

Q.1 (c)

Q.2 (a)

Q.3 (d)

6.9 Exercise

Q.1 Construct a finite automaton M which can recognize DFA in a given string over the alphabet {A, B, ..., Z}. For example, M has to recognize DFA in the string ATXDFAMNQ.

Q.2 Construct a finite automaton for the regular expression (a + b)*abb.

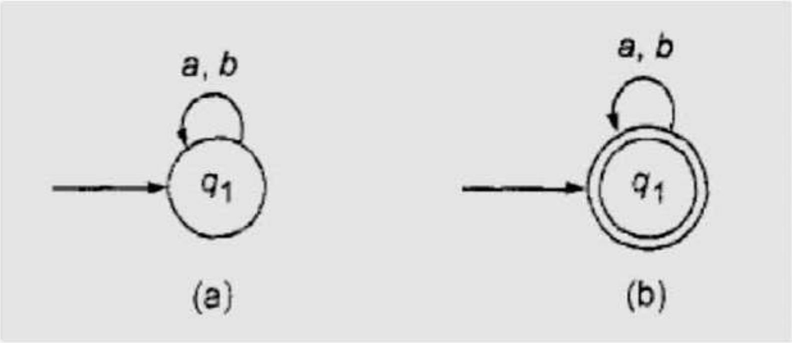
Q.3 Find all strings of length 5 or less in the regular set represented by the following regular expressions:

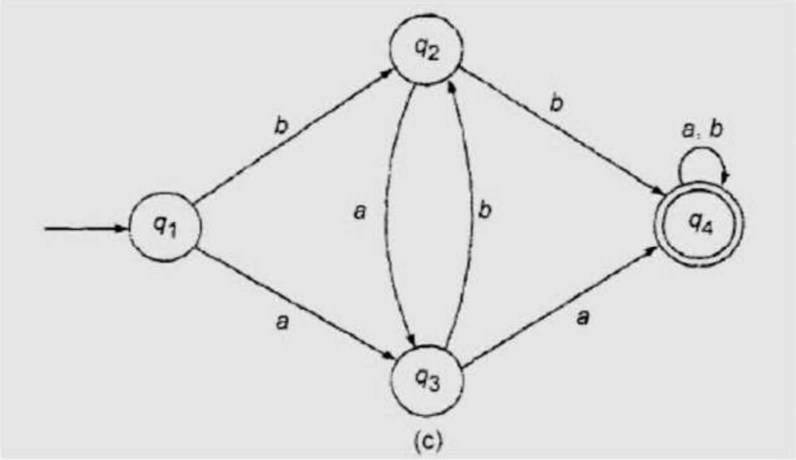
(a) (ab + a)*(aa + b)

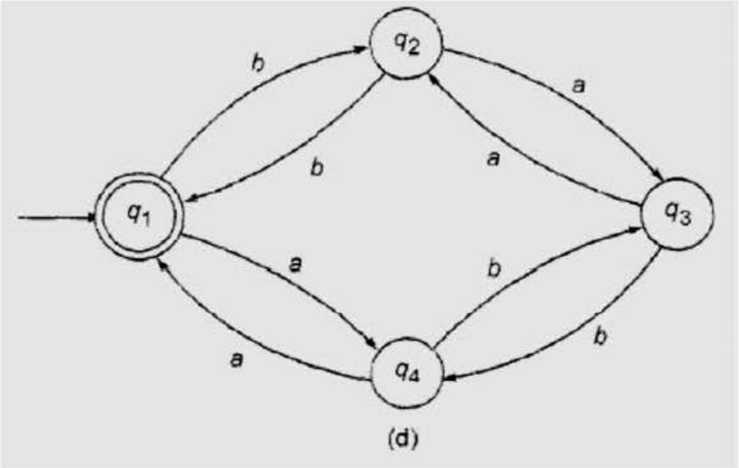
(b) (a*b + b*a)*a

(c) a* + Cab + a)*

Q. 4 Find the set of strings over L = {a, b} recognized by the transition systems shown in Fig.(a-d).







(d)

References and Suggested Readings

1. Peter Linz, An Introduction to Formal Languages and Automata, Jones and Bartlett Publication, Third Edition.

2. K.L.P. Mishra, N. Chandrasekaran, Theory of Computer Science, BPB Publication, Prentice-Hall of India, Second Edition.

3. H.E. Hopcraft and J.D. Ullamn, Introduction to Automata Theory, Languages and Computation, Narosa Publications.

4. J.C. Martin, Introduction to Languages and the Theory of Automata, Tata McGraw-Hill.

UNIT-7
Regular Languages and Properties of Regular Languages

Structure of the Unit

7.0 Objective

In this chapter we shall focus upon the following topics

- Closure properties of Regular Sets
- Pumping Lemma for Regular Sets
- Application of Pumping Lemma
- Constructions of a Regular Grammar for Given DFA
- Constructions of a Transition System for a Given Regular Grammar

7.1 Introduction

A regular language (also called a rational language) is a formal language that can be expressed using a regular expression. Alternatively, a regular language can be defined as a language recognized by a finite automaton. The equivalence of regular expressions and finite automata is known as Kleene's theorem. In the Chomsky hierarchy, regular languages are defined to be the languages that are generated by Type-3 grammars (regular grammars).

Regular languages are very useful in input parsing and programming language design. The collection of regular languages over an alphabet Σ is defined recursively as follows:

The empty language Ø, and the empty string language $\{\varepsilon\}$ are regular languages.

- For each a $\in \Sigma$ (a belongs to Σ), the singleton language {a} is a regular language.
- If A and B are regular languages, then A $\cup$ B (union), A • B (concatenation), and A* (Kleene star) are regular languages.
- No other languages over Σ are regular.

When is a language is regular?

if we are able to construct one of the following:

DFA or NFA or ε -NFA or regular expression

When is it not?

If we can show that no FA can be built for a language

7.2 Closure properties of Regular Sets

Closure property:

If a set of regular languages are combined using an operator, then the resulting language is also regular.Let L and M be regular languages. Then the following languages are all regular:

- Union :L∪M
- Intersection :L∩M
- Complement : $\overline{N}$
- Difference : L \ M
- Reversal : $L^R = \{w^R : w \in L\}$
- Closure : L*
- Concatenation : L.M
- Homomorphism : $\boxed{h(a_1, a_2, \ldots a_n) = h(a_1)h(a_2)\ldots h(a_n)}$

  h(L) =$\{h(w) : w \in L, h \text{ is homomorphism}\}$
- Inverse homomorphism:

  $$h^{-1}(L) = \{w \in \sum : h(w) \in L, h: \sum \to \Delta^* \text{is a homom.}\}$$

Theorem 7.1: For any regular L and M, L∪M is regular.

Proof. Let L = L(E) and M = L(M). Then

L(E +F) = L∪M by definition.

Theorem 7.2 : If L is a regular language over Σ then so is $\overline{L} = \Sigma^* \backslash L$.

Proof. Let L be recognized by a DFA

A = (Q, Σ , δ, q_0,F).

Let B = (Q, Σ , δ, q_0,Q\F). Now L(B) = $\overline{L}$.

Example:

Let L be recognized by the DFA given below

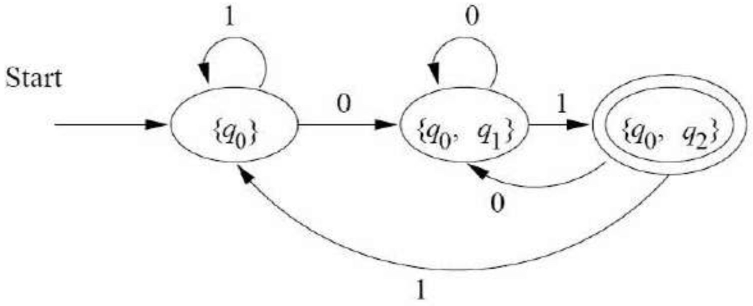


Figure 7.1: State Diagram of DFA for L

Then $\overline{\text{L}}$ is recognized by

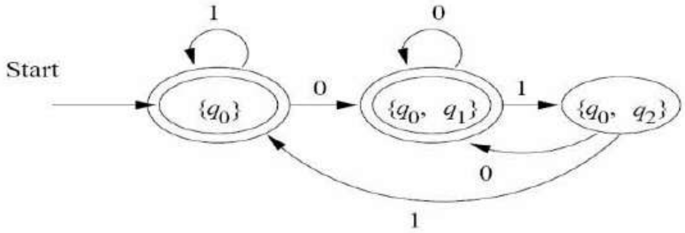


Figure 7.2: State Diagram of DFA for $\overline{\text{L}}$

Theorem 7.3. If L and M are regular, then so is L$\cap$M.

Proof 1. By DeMorgan's law L$\cap$M = $\overline{\overline{L} \cup \overline{M}}$.

We already know that regular languages are closed under complement and union. We shall also give a nice direct proof.

Proof 2. Let L be the language of

$$A_L = (Q_L, \Sigma, \delta_L, q_L, F_L)$$

and M be the language of

$$A_M = (Q_M, \Sigma, \delta_M, q_M, F_M)$$

We assume that both automata are deterministic.

We shall construct an automaton that simulates A_L and A_M in parallel, and accepts if and only if both A_L and A_M accept.

If A_L goes from state p to state s on reading a,and A_M goes from state q to state t on reading a, then $A_{L\cap M}$ will go from state (p, q) to state (s, t) on reading a.

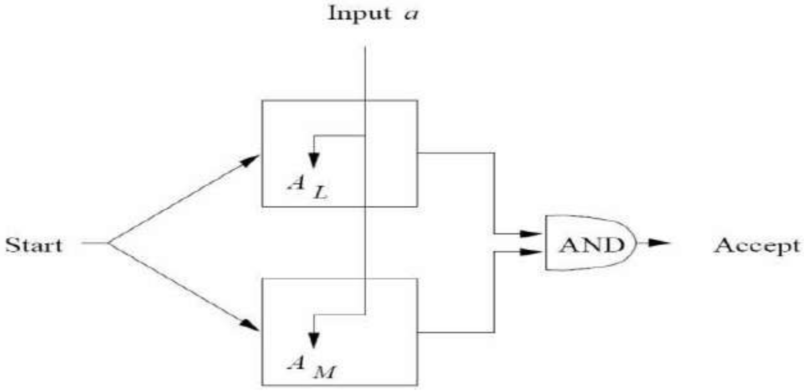


Figure 7.3

Formally

$A_{L\cap M}$=($Q_L X Q_L, \Sigma$, $\delta_{L\cap M}$,(q_L, q_{M}), $F_L X F_M$),

where

$\delta_{L\cap M}\big((p,q),a\big) = (\delta_L(p,q), \delta_M\big((q,q),\big)$

It can be shown by induction on $|W|$ that

$\hat{\delta}_{L\cap M}((q_L, q_{M}), w) = \hat{\delta}_L((q_L, w), \hat{\delta}_M(q_M, w))$

The claim then follows

Example : (c) = (a) × (b)

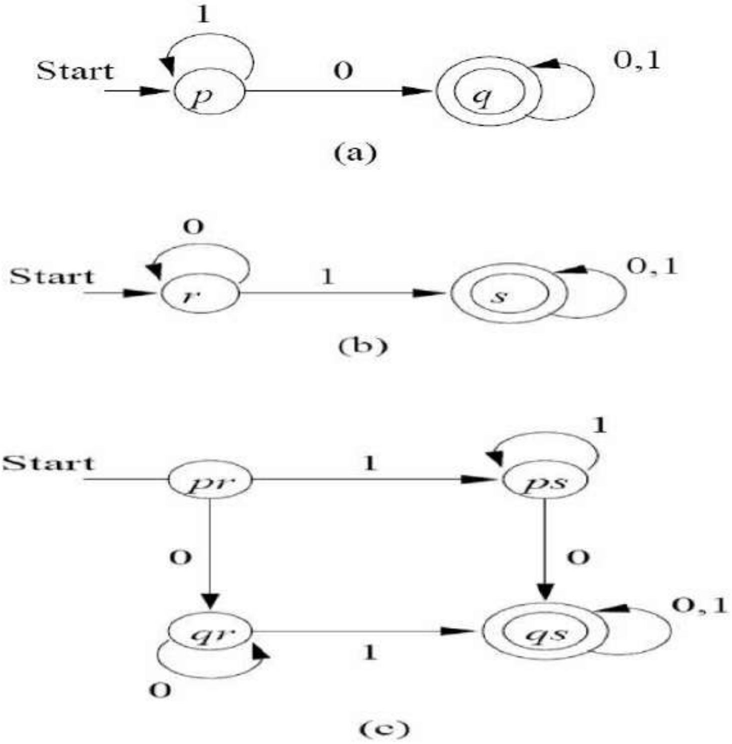


Figure 7.4

Theorem 7.4. If L and M are regular languages,then so in L\ M.

Proof. Observe that L \M =L∩ $\bar{M}$. We already know that regular languages are closed under complement and intersection

Theorem 7.5. If L is a regular language, then so is L^R.

Proof 1: Let L be recognized by an FA A.Turn A into an FA for L^R, by

1. Reversing all arcs.
2. Make the old start state the new sole accepting state.

3. Create a new start state p_0 , with δ (p_0,ϵ) = F (the old accepting states).

Proof 2: Let L be described by a regex E. We shall construct a regex ER, such that $L(E^R) = (L(E))^R$.

We proceed by a structural induction on E.

Basis: If E is ϵ,$\emptyset$ or a then $E^R = E$.

Induction:

1. $E = F + G$. Then $E^R = F^R + G^R$
2. $E = F.G$. Then $E^R = G^R.F^R$
3. $E = F^*$. Then $E^R = (F^R)^*$

We can show by structural induction on E

$L(E^R) = (L(E))^R$

Homomorphism

A homomorphism on Σ is a function h : $\Sigma \rightarrow \Theta^*$,where Σ and Θ are alphabets.

Let $w = a_1a_2 \ldots a_n \in \Sigma^*$ Then $h(w) = h(a_1)h(a_2) \ldots h(a_n)$ and

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

Example: Let h : $\{0,1\}^* \rightarrow \{a, b\}^*$ be defined by h(0) = ab, and h(1) = ϵ Now h(0011) = abab

Theorem 7.6: h(L) is regular, whenever L is.

e.g. h(0*1+(0+1)*0)= h(0)*h(1)+(h(0)+h(1))*h(0)

Proof:

Let L = L(E) for a regex E. We claim that L(h(E)) = h(L).

Basis: If E is ϵ or $\emptyset$;. Then h(E) = E, and L(h(E)) = L(E) = h(L(E)).

If E is a, then L(E) =$\{a\}$, L(h(E)) = L(h(a)) =$\{h(a)\}$ = h(L(E)).

Induction:

Case 1: G = E + F. Now L(h(E + F)) = L(h(E)+h(F)) = L(h(E)) $\cup$L(h(F)) = h(L(E)) $\cup$ h(L(F)) = h(L(E) $\cup$ L(F)) = h(L(E +F)).

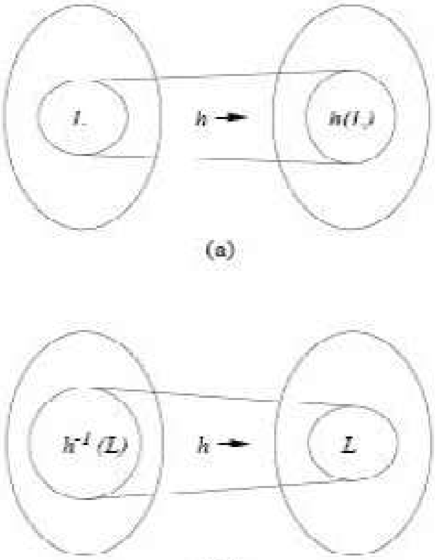
Case 2: G = E.F . Now L(h(E.F)) = L(h(E)).L(h(F))= h(L(E)).h(L(F)) = h(L(E).L(F)) = h(L(E.F))

Case 3: G = E*. Now L(h(E*)) = L(h(E)*) =L(h(E))* = h(L(E))* = h(L(E*))

Inverse Homomorphism

Let h : $\Sigma \rightarrow \Theta^*$ be a homom. Let L $\subseteq \Theta^*$,and define

$h^{-1}(L) = \{w \in \Sigma^*: h(w) \in L\}$



(a)

(b)

Figure 7.5

Example: Let h : $\{a,b\} \rightarrow \{0,1\}^*$ be defined by h(a) = 01, and h(b) = 10. If L = L((00+1)*), then h^{-1}(L) = L((ba)*).

Claim: h(w) $\in$ L if and only if w = $(ba)^n$

Proof: Let w = $(ba)^n$. Then h(w) = $(1001)^n \in$ L.

Let h(w) $\in$ L, and suppose w $\notin$ L((ba)*). There are four cases to consider.

1 w begins with a. Then h(w) begins with 01 and $\notin$ L($(00+1)^*$).

2. w ends in b. Then h(w) ends in 10 and $\notin$ L($(00+1)^*$).

3. w = xaay. Then h(w) = z0101v and $\notin$ L($(00+1)^*$).

4. w = xbby. Then h(w) = z1010v and $\notin$ L((00+1)*).

Theorem 7.7: Let h : $\Sigma \rightarrow \Theta^*$ be a homom.,and L $\subseteq \Theta^*$regular. Then h^{-1} (L) is regular.

Proof: Let L be the language of A = (Q, Θ , δ, q_0,F).

We define B = (Q, Σ, γ , q_0,F), where

$$\gamma(q, a) = \bar{\delta}\,(q, h(a))$$

It can be shown by induction on $|w|$ that

$$\bar{\gamma}(q_0, w) = \bar{\delta}\,(q_0, h(w))$$

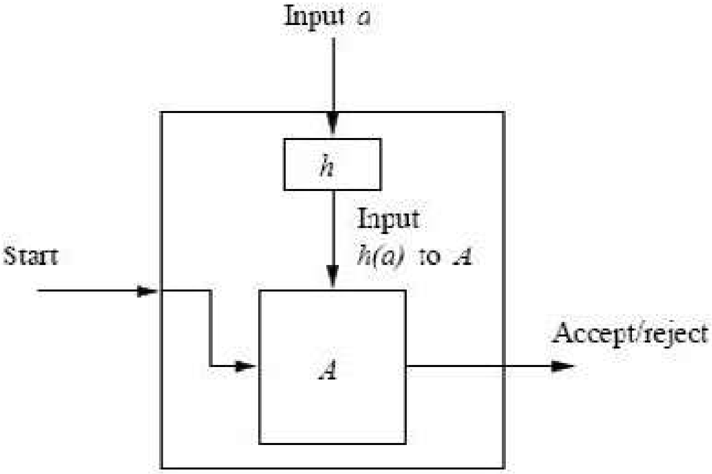


Figure 7.6

7.3 Pumping Lemma for Regular Sets

In the previous sections, we have seen that the class of regular languages is closed under various operations, and that these languages can be described by (deterministic or nondeterministic) finite automata and regular expressions.

These properties helped in developing techniques for showing that a language is regular. In this section, we will present a tool that can be used to prove that certain languages are not regular. Observe that for a regular language,

1. the amount of memory that is needed to determine whether or not a given string is in the language is finite and independent of the length of the string, and
2. if the language consists of an infinite number of strings, then this language should contain infinite subsets having a fairly repetitive structure.

   Intuitively, languages that do not follow 1. or 2. should be non-regular. For example, consider the language

$\{0^n1^n : n \geq 0\}$.

This language should be non-regular, because it seems unlikely that a DFA can remember how many 0s it has seen when it has reached the border between the 0s and the 1s. Similarly the language $\{0^n$: n is a prime number$\}$ should be non-regular, because the prime numbers do not seem to have any repetitive structure that can be used by a DFA. To be more rigorous about this, we will establish a property that all regular languages must possess. This property is called the pumping lemma. If a language does not have this property, then it must be non-regular.

The pumping lemma states that any sufficiently long string in a regular language can be pumped, i.e., there is a section in that string that can be repeated any number of times, so that the resulting strings are all in the language.

Theorem (Pumping Lemma for Regular Languages)

Let A be a regular language. Then there exists an integer $p \geq 1$, called the pumping length, such that the following holds: Every string s in A, with $|s| \geq p$, can be written as $s = xyz$, such that

1. $y \neq \varepsilon$ (i.e., $|y| \geq 1$),
2. $|xy| \leq p$, and
3. for all $i \geq 0$, $xy^iz \in A$

In words, the pumping lemma states that by replacing the portion y in s by zero or more copies of it, the resulting string is still in the language A.

Proof. Let $\sum$ be the alphabet of A. Since A is a regular language, there exists a DFA $M = (Q, \sum, \delta, q, F)$, that accepts A. We define p to be the number of states in Let $s = s_1s_2 \ldots s_n$ be an arbitrary string in A such that $n \geq$ p. Define $r_1 = q$, $r_2 = \delta (r_1, s_1)$, $r_3 = \delta (r_2, s_2)$, . . ., $r_n+1 = \delta (r_n, s_n)$. Thus, when the DFA M reads the string s from left to right, it visits the states $r_1, r_2, \ldots, r_{n+1}$. Since s is a string in A, we know that r_{n+1}.belongs to F.

Consider the first p + 1 states $r_1, r_2, \ldots, r_{p+1}$ in this sequence. Since the number of states of M is equal to p, the pigeonhole principle implies that there must be a state that occurs twice in this sequence. That is, there are indices j and ℓ such that $1 \leq j < \ell \leq p + 1$ and $r_j = r_\ell$

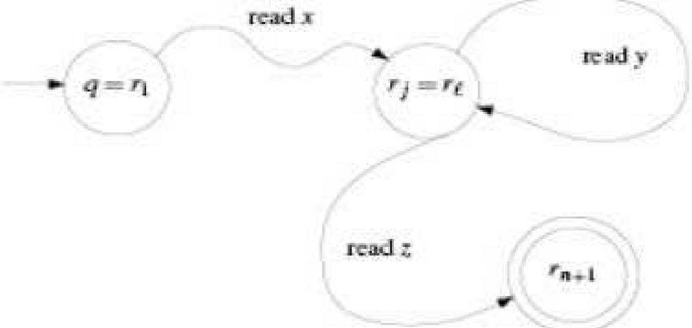


Figure 7.7

We define $x = s_1 s_2 \ldots s_{j-1}$, $y = s_j \ldots s_{\ell-1}$, and $z = s_\ell \ldots s_n$. Since $j < \ell$, we have $y \neq \varepsilon$, proving the first claim in the theorem. Since $\ell \leq p+1$, we have $|xy| = \ell - 1 \leq p$, proving the second claim in the theorem. To see that the third claim also holds, recall that the string s = xyz is accepted by M.

While reading x, M moves from the start state q to state r_j . While reading y, it moves from state r_j to state $r_\ell = r_j$, i.e., after having read y, M is again in state r_j . While reading z, M moves from state r_j to the accept state r_{n+1}.Therefore, the substring y can be repeated any number $i \geq 0$ of times, and the corresponding string xy^iz will still be accepted by M. It follows that $xy^iz \in A$ for all $i \geq 0$.

7.4 Application of Pumping Lemma

Pumping Lemma is to be applied to show that certain languages are not regular. It should never be used to show a language is regular.

- If **L** is regular, it satisfies Pumping Lemma.
- If **L** does not satisfy Pumping Lemma, it is non-regular.

Method to prove that a language L is not regular

- At first, we have to assume that **L** is regular.

- So, the pumping lemma should hold for **L**.
- Use the pumping lemma to obtain a contradiction −
  - ✓ Select **w** such that $|\mathbf{w}| \geq \mathbf{c}$
  - ✓ Select **y** such that $|\mathbf{y}| \geq \mathbf{1}$
  - ✓ Select **x** such that $|\mathbf{xy}| \leq \mathbf{c}$
  - ✓ Assign the remaining string to **z.**
  - ✓ Select **k** such that the resulting string is not in **L.**

Hence L is not regular.

Example 7.1:

Prove that $\mathbf{L = \{a^ib^i \mid i \geq 0\}}$ is not regular.

Solution

At first, we assume that L is regular and n is the number of states.

Let $w = a^nb^n$. Thus $|w| = 2n \geq n$. By pumping lemma, let w = xyz, where $|xy| \leq n$.

Let $x = a^p$, $y = a^q$, and $z = a^rb^n$, where $p + q + r = n$, $p \neq 0$, $q \neq 0$, $r \neq 0$. Thus $|y| \neq 0$. Let k = 2. Then $xy^2z = a^pa^{2q}a^rb^n$. Number of as = $(p + 2q + r) = (p + q + r) + q = n + q$

Hence, $xy^2z = a^{n+q}\ b^n$. Since $q \neq 0$, xy^2z is not of the form a^nb^n.

Thus, xy^2z is not in L. Hence L is not regular.

Example 7.2:

Consider the language

$A = \{w \in \{0, 1\}^*$: the number of 0s in w equals the number of 1s in w$\}$.

Solution

Again, we prove by contradiction that A is not a regular language.

Assume that A is a regular language. Let $p \geq 1$ be the pumping length, as given by the pumping lemma. Consider the string $s = 0^p1^p$. Then $s \in A$ and $|s| = 2p \geq p$. By the pumping lemma, s can be written as s = xyz, where $y \neq \varepsilon$, $|xy| \leq p$, and $xy^iz \in A$ for all $i \geq 0$.

Since $|xy| \leq p$, the string y contains only 0s. Since $y \neq \varepsilon$, y contains at least one 0. Therefore, the string $xy^2z = xyyz$ contains more 0s than 1s, which implies that this string is not contained in A. But, by the pumping lemma, this string is contained in A. This is a contradiction and, therefore, A is not a regular language.

Example 7.3:

Consider the language

$A = \{ww : w \in \{0, 1\}^*\}$.

Solution

We prove by contradiction that A is not a regular language.

Assume that A is a regular language. Let $p \geq 1$ be the pumping length, as given by the pumping lemma. Consider the string $s = 0^p10^p1$. Then $s \in A$ and $|s| = 2p + 2 \geq p$. By the pumping lemma, s can be written as s = xyz, where $y \neq \varepsilon$, $|xy| \leq p$, and $xy^iz \in A$ for all $i \geq 0$.

Since $|xy| \leq p$, the string y contains only 0s. Since $y \neq \varepsilon$, y contains at least one 0. Therefore, the string $xy^2z = xyyz$ is not contained in A. But, by the pumping lemma, this string is contained in A. This is a contradiction and, therefore, A is not a regular language.

You should convince yourself that by choosing $s = 0^2p$ (which is a string in A whose length is at least p), we do not obtain a contradiction. The reason is that the string y may have an even length. Thus, 0^2p is the "wrong" string for showing that A is not regular. By choosing s = = 0^p10^p1, we do obtain a contradiction; thus, this is the "correct" string for showing that A is not regular.

Example 7.4:

Consider the language

$A = \{1^n : n \text{ is a prime number}\}$.

Solution:

We prove by contradiction that A is not a regular language.

Assume that A is a regular language. Let $p \geq 1$ be the pumping length, as given by the pumping lemma. Let $n \geq p$ be a prime number, and consider the string $s = 1^n$.

Then s $\in$ A and $|s| = n \geq p$. By the pumping lemma, s can be written as s = xyz, where $y \neq \varepsilon$, $|xy| \leq p$, and $xy^iz \in A$ for all $i \geq 0$. Let k be the integer such that $y = 1^k$. Since $y \neq \varepsilon$, we have $k \geq 1$. For each $i \geq 0$, $n + (i - 1)k$ is a prime number, because $xy^iz = 1^{n+(i-1)k} \in A$. For $i = n + 1$, however, we have $n + (i - 1)k = n + nk = n(1 + k)$, which is not a prime number, because $n \geq 2$ and $1 + k \geq 2$. This is a contradiction and, therefore, A is not a regular language

Kleene's seminal article defines regular expressions and their relationship to finite automata. Kleene proves the equivalence of finite automata and regular expressions. There are three methods to construct regular grammar from given DFA

- Transitive Closure Method
- State Removal Method
- Brzozowski Algebraic Method

Transitive Closure Method

Suppose the given DFA M is to be represented as a regular expression

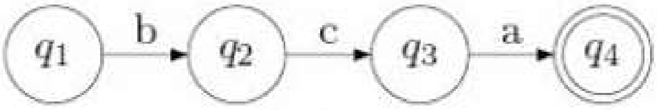


Figure 7.8

Consider the automaton in Figure 7.8. The input for edge in the automaton is a regular expression. Quite simply, the regular expression for the transition from q_1 to q_2 is b, the transition from q_2 to q_3 is c and so on. Furthermore, the regular expression representing the transition from

q_1 to q_3 is the concatenation of the regular expressions thus forming bc.

Thus, we can find the regular expression for the automaton to be bca since that expression is the concatenation of all of the transitions from the starting state q_1 to the final state q_4.

More generally, for a path from q_λ to q_f , the concatenation of the regular expression for each transition in the path forms a regular expression that represents the same string as the path from q_λ to q_f in the automaton.

Supposing there exists only one unique path in automaton M from q_λ to q_f, there exists only one regular expression R such that R represents the same string as the DFA M. However, this is a trivial automaton, let us examine how to expand this to a more general case.

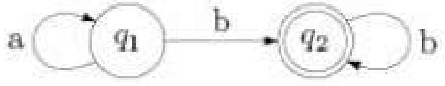


Figure 7.9

Now consider the DFA in Figure 7.9. It is clear that multiple paths exist from q_1 to q_2. We cannot derive a simple regular expression to represent the DFA, however using the other operators (union and iteration) we can build on our previous approach to create a construction that works for all types of DFA.

Suppose regular expression R_{ij} represents the set of all strings that transition the automaton M from q_i to q_j. Furthermore, suppose R^k_{ij} represents the set of all strings that transition the automaton M from q_i to q_j without passing through any state higher than q_k. We can construct R_{ij} by successively constructing R^1_{ij},R^2_{ij} ………R^m_{ij} ,..R^k_{ij} is recursively defined as:

$$R_{ij}^{k} = R_{ik}^{k-1}\left(R_{kk}^{k-1}\right) * R_{kj}^{k-1} + R_{ij}^{k-1}$$

Assuming we have initialized R_{ij}^{0} to be

$$R_{ij}^{0} = \begin{cases} \text{r if i } \neq \text{ j and r transitions M from } q_i \text{ to } q_j \\ \text{r } + \lambda \text{ if i } = \text{ j and r transitions M from } q_i \text{ to } q_j \\ \phi \text{ otherwise} \end{cases}$$

As we can see, this successive construction builds up regular expressions until we have R_{ij}. We can then construct a regular expression representing M as the union of all $R_{\lambda f}$ where q_λ is the starting state and f ∈ M_f(the final states for M).

This technique is similar in nature to the all-pairs shortest path problem. The only difference being that we are taking the union and concatenation of regular expressions instead of summing up distances. This solution is of the same form as transitive closure and belongs to the constellation of problems associated with closed semirings.

The chief problem of the transitive closure approach is that it creates very large regular expressions. Examining the formula for an R_{ij}^{0}, it is clear the significant length is due to the repeated union of concatenated terms. Even by using the previous identities, we still have long expressions.

State Removal Method

The state removal approach identifies patterns within the graph and removes states, building up regular expressions along each transition. The advantage of this technique over the transitive closure method is that it is easier to visualize. This technique is described by Du and Ko, but we examine a much simpler approach is given by Linz.

First, any multi-edges are unified into a single edge that contains the union of inputs. Suppose from q_2 to q_5 there is an edge a and an edge b, those would be unified into one edge from q_2 to q_5 that has the value a + b.

Now, consider a subgraph of the automaton M which is of the form given in figure. State q may be removed and the automaton may be reduced to the form in figure. The pattern may still be applied if edges are missing. For an edge that is missing, leave out the corresponding edge in figure.

This process repeats until the automaton is of the form in figure. Then by direct calculation, the regular expression is:

$r = r_1 \cdot r_2(r_4 + r_3 r_1 \cdot r_2) \cdot$

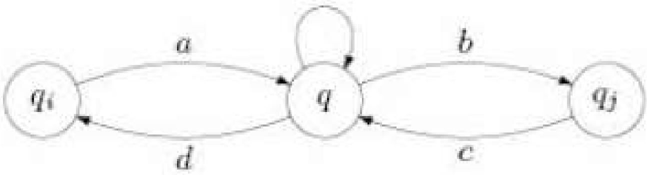


Figure 7.10: Desired pattern for state removal

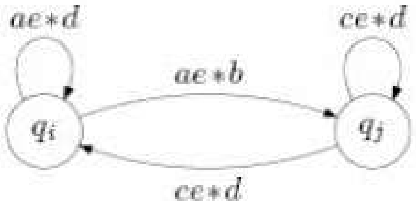


Figure 7.11: Results after state removal

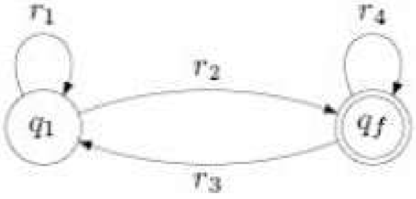


Figure 7.12: Final Form

Brzozowski Algebraic Method

Brzozowski method takes a unique approach to generating regular expressions. We create a system of regular expressions with one regular expression unknown for each state in M, and then we solve the system for R_λ where R_λ is the regular expression associated with starting state q_λ. These equations are the characteristic equations of M.

Constructing the characteristic equations is straightforward. For each state q_i in M, the equation for R_i is a union of terms. Each term can be constructed like so: for a transition a from q_i to q_j , the term is aR_j . If R_i is a final state, λ is also one of the terms. This leads to a system of equations in the form:

$$\begin{gathered}R_1 = a_1R_1 + a_2R_2 + \ldots\ldots \\ R_2 = a_1R_1 + a_2R_2 + \cdots \\ R_3 = a_1R_1 + a_2R_2 + \cdots \lambda \\ \vdots \\ R_m = a_1R_1 + a_2R_2 + \cdots \lambda\end{gathered}$$

where $a_x = \phi$ if there is no transition from R_i to R_j .

The system can be solved via straightforward substitution, except when an unknown appears on both the right and left hand side of the equation. This situation occurs when there is a self-loop for state q_i. Arden's theorem is the key to solving these situations. The theorem is as follows:

Given an equation of the form X = AX + B where λ A, the equation has the solution X = A*B.

We use this equation to isolate R_i on the left hand size and successively substitute R_i into the another equation. We repeat the process until we have found R_λ with no unknowns on the right hand side.

For example, consider again the automaton in figure. The characteristic equations are as follows (where $R_\lambda = R_1$)

$$R_1 = a_1R_1 + a_2R_2$$
$$R_2 = bR_2 + \lambda$$

We solve for R_2 using Arden's theorem and the previously mentioned identities:

$$R_2 = bR_2 + \lambda$$

= b*λ

= b*

We substitute into R_1 and solve:

$$R_1 = aR_1 + b(b^*)$$

$= aR_1 + bb^*$

= a* (bb*)

= a*bb*

Thus, the regular expression for the automaton in Figure is a*bb*.

The state removal approach seems useful for determining regular expressions via manual inspection, but is not as straightforward to implement as the transitive closure approach and the algebraic approach. The transitive closure approach gives a clear and simple implementation, but tends to create very long regular expressions. The algebraic approach is elegant, leans toward a recursive approach, and generates reasonably compact regular expressions. Brzozowski's method is particularly suited for recursion oriented languages, such as functional languages, where the transitive closure approach would be cumbersome to implement.

7.6 Constructions of a Transition System for a Given Regular Grammar

If all production of a CFG are of the form A → wB or A→ w, where A and B are variables and w some string of zero or more terminals. then we say that grammar is right linear. If all production of a CFG are of the form A → Bw or A→ w, we call it left linear. A right or left linear grammar is called a regular grammar. Every regular expression can be represented by a regular grammar. As there is a finite automaton for every regular expression we can generate a finite automaton for the regular grammar.

Given a regular grammar G, a finite automata accepting L(G) can be obtained as follows:

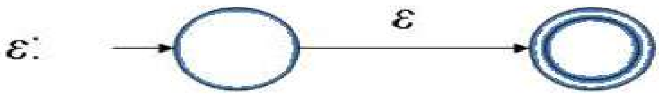
1. The number of states in the automata will be equal to the number of non-terminals plus one. Each state in automata represents each non-terminal in the regular grammar. The additional state will be the final state of the automata. The state corresponding to the start symbol of the grammar will be the initial state of automata. If L(G) contains ∈ that is start symbol is grammar devices to ∈, then make start state also as final state.
2. The transitions for automata are obtained as follows

1. For every production A → aB make δ(A, a) = B that is make an arc labeled 'a' from A to B.
2. For every production A → a make δ(A, a) = final state.
3. For every production A → ∈, make δ(A, ∈) = A and A will be final state

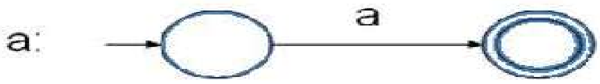
Building Transition Diagrams

- The basic cases.

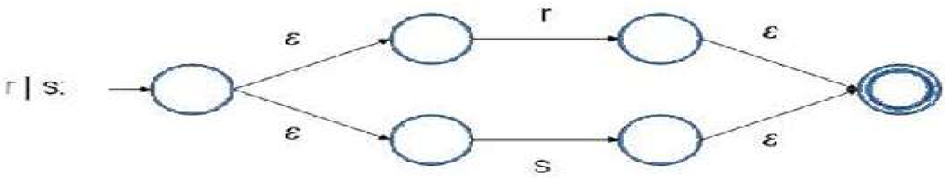
- For ε, build



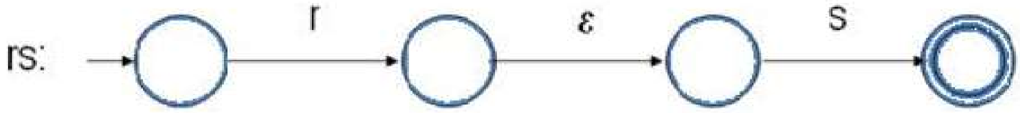
- For each symbol a ∈ Σ, build



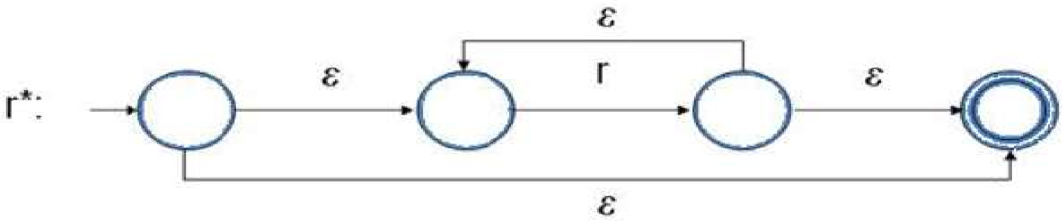
- The recursive cases.
- For the expression r | s, build



- For the expression rs, build



- For the expression r*, build

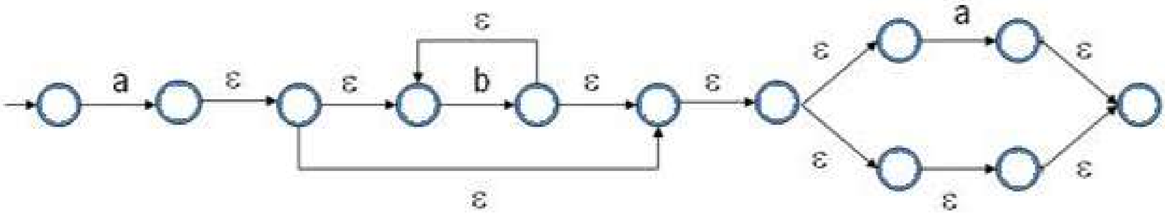


Building Transition Diagrams

- Applying these rules builds an NFA representing the regular expression.
- Note that each diagram has unique start and accepting states.
- Note also that generous use was made of ε-moves.
- This facilitates joining them together without any complications.

Example 7.5: Building a Transition Diagram

- Build a transition diagram from the regular expression ab*(a | ε).
- Applying the rules rigorously produces the following.



7.7 Self Learning Exercise

Q.1 language is regular if and only if

a) accepted by DFA

b) accepted by PDA

c) accepted by LBA

d) accepted by Turing machine

Q.2 Regular grammar is

a) context free grammar

b) non context free grammar

c) english grammar

d) none of the mentioned

Q.3 Context free grammar is closed under

a) Union

b) Intersection

c) Both

d) None of these

Q.4 If L1 and L2 are regular languages is/are also regular language(s).

a) L1 + L2

b) L1.L2

c) L1

d) All of the above

7.8 Summary

A regular language (also called a rational language) is a formal language that can be expressed using a regular expression. If we are able to construct DFA or NFA or Ɛ -NFA then It is a language is regular or regular expression and If we can show that no FA can be built for a language then it is not regular. Regular languages are closed under these operations: Union, Intersection, Difference,
Concatenation, Kleene Closure, Reversal, Homomorphism and Inverse Homomorphism The pumping lemma and its applications are discussed.
The Purpose of the Pumping Lemma for RL is to prove that some languages cannot be regular.

7.9 Glossary

Languages: A language is a *set of string all of which are chosen from some ∑*, where ∑ is a particular alphabet*

Concatenation of Languages:

I f L1 and L2 are two languages then their concatenation can be defined as :

L = L1 . L2 where L = {w: w = xy where x Є L1, y Є L2}
It means that all the strings in the language L are concatenation of stings of language L1 and L2

Kleen Closure:

If S is a set of words then by S* we mean the set of all finite strings formed by concatenating words from S, where any word may be used as often we like, and where the null string is also included.

7.10 Answers to Self-Learning Exercise

Q.1 (a)

Q.2 (a)

Q.3 (c)

Q.4 (d)

7.11 Exercise

Q.1 Which of the following is true?

(A) (01)*0 = 0(10)*

(B) (0+1)*0(0+1)*1(0+1) = (0+1)*01(0+1)*

(C) (0+1)*01(0+1)*+1*0* = (0+1)*

(D) All of the mentioned

Q.2 Which of the following is not a regular expression?

(A) [(a+b)*-(aa+bb)]*

(B) [(0+1)-(0b+a1)*(a+b)]*

(C) (01+11+10)*

(D) (1+2+0)*(1+2)*

Q.3 Which of the following is true?

(A) Every subset of a regular set is regular

(B) Every finite subset of non-regular set is regular

(C) The union of two non regular set is not regular

(D) Infinite union of finite set is regular

Q.4 Regular expressions are closed under

(A) Union

(B) Intersection

(C) Kleen star

(D) All of the mentioned

Q.5 Pumping lemma is generally used for proving that

(A) given language is regular

(B) given language is not regular

(C) whether two given regular expressions are equivalent or not

(D) All of the mentioned

Q.6 L= language of words containing even number of a's. Regular Expression is

(A) (a+b)aa(a+b)

(B) a+bbaaba

(C) (b+aba)

(D) (a+b)ab(a+b)

Q.7 The languages -------------- are the examples of non-regular languages.

(A) PALINDROME and PRIME

(B) PALINDROME and EVEN-EVEN

(C) EVEN-EVEN and PRIME

(D) FACTORIAL and SQURE

Q.8 Which of the following statement is false?

(A) If L is context free language then L* is also a context free language

(B) If L1 and L2 are context free language then there intersection is not a context free language

(C) If L1 and L2 are context free language then there union is also a context free language

(D) None of these

Q.9 (a+b)* is equivalent to

(A) b*a*

(B) (a*b*)*

(C) a*b*

(D) none of above

Q.10 Precedence of regular expression in decreasing order is

(A) * , . , +

(B) . , * , +

(C) . , + , *

(D) + , a , *

7.12 Answers of Exercise

Q.1 (D)

Q.2 (B)

Q.3 (B)

Q.4 (D)

Q.5	(B)
Q.6	(C)
Q.7	(A)
Q.8	(D)
Q.9	(B)
Q.10	(A)

References and Suggested Readings

1. K.Krithivasan and R.Rama; Introduction to Formal Languages, Automata Theory and Computation; Pearson Education, 2009.
2. J.E.Hopcroft, R.Motwani and J.D. Ullman , "Introduction to Automata Theory Languages and computation", Pearson Education Asia , 2001.
3. Peter Linz, "An Introduction to Formal Language and Automata", 4th Edition, Narosa Publishing house , 2006.
4. M.Sipser; Introduction to the Theory of Computation; Singapore: Brooks/Cole, Thomson Learning, 1997.
5. John.C.martin, "Introduction to the Languages and the Theory of Computation",Third edition, Tata McGrawHill, 2003.
6. https://www.tutorialspoint.com/automata_theory/pumping_lemma_for_regular_grammar.htm
7. http://www.sanfoundry.com/automata-theory-pumping-lemma-regular-languages/

UNIT-8

Context-Free Grammars and Languages

Structure of the Unit

8.0 Objective

In this chapter we shall focus upon the following topics:

- Context free grammar
- Context free Languages
- Derivation Trees
- Ambiguity in Context free grammars
- Simplification of Context Free Grammars

8.1 Introduction

Grammar is nothing but a set of rules to define valid sentences in any languages. This chapter introduces context-free grammars, which generates context-free languages. Context-free languages have great practical significance in defining programming languages and in simplifying the translation for programming languages.

8.2 Context Free Grammar

A context-free grammar basically consists of a finite set of grammar rules. In order to define grammar rules, we assume that we have two kinds of symbols: the terminals, which are the symbols of the alphabet underlying the languages under consideration, and the non-terminals, which behave like variables ranging over strings of terminals. A rule is of the form $A \longrightarrow \alpha$, where A is a single non-terminal, and the right-hand side α is a string of terminal and/or non-terminal symbols.

Definition: A context-free grammar (for short, CFG) is a quadruple G = (V_n,V_t, P, S), Where

V_n : A finite set of non-terminals, generally represented by capital letters, A,B,C,D….

V_t: A finite set of terminals, generally represented by small letters, like, a, b ,c ,d…

S : Starting non-terminal, called start symbol of the grammar. S belongs to V_n .

P: Set of rules or productions in CFG.

G is context-free and all production in P has the form

$$\alpha \rightarrow \beta$$

Where $\alpha \in V_n$ and $V_n \in (V_t \cup V_n)^*$

Every regular grammar is context-free, so regular language is also a context-free one.

Example 8.1: G1 = ({E, a, b}, {a, b}, P,E), where P is the set of rules

$E \longrightarrow aEb,$

$E \longrightarrow ab$

This grammar generates the language L1 = $\{a^n b^n \mid n \geq 1\}$, which is not regular.

8.2.1 Derivations

We know define the notations to represent a derivation. First we define two notations $=\Rightarrow_G$ and $=^{*}\Rightarrow_G$. If $\alpha \rightarrow \beta$ is production of P in CFG and a and b are string in $(V_t \cup V_n)^*$, then

$a\,\alpha\, b =\Rightarrow_G a\,\beta\, b$

We say that the production $\alpha \rightarrow \beta$ is applied to the string $a\,\alpha\, b$ to obtain $a\,\beta\, b$ or we say that $a\,\alpha\, b$ directly $a\,\beta\, b$.

Now suppose $\alpha_{1,}\ \alpha_{2,}\ \alpha_{3\ldots}\ \alpha_m$ are string in $(V_t \cup V_n)^*$,

$m \geq 1$ and $\alpha_1 =\Rightarrow_G \alpha_{2,}\ \alpha_2 =\Rightarrow_G \alpha_{3,}\ \alpha_3 =\Rightarrow_G \alpha_{4,}\ \alpha_4 =\Rightarrow_G \alpha_{5,}\ldots\ldots\ \alpha_{m-1} =\Rightarrow_G \alpha_m$

Then we say that $\alpha_1 =^{*}\Rightarrow_G \alpha_m$, i.e. , we say α_1 drives α_m in grammar G. If α drives β by exactly i steps we say $\alpha \overset{i}{=\Rightarrow_G} \beta$.

8.2.2 Sentential Forms

Derivation from the start symbol produce strings that have a special rule. We call these "Sentential forms". That is, if $G=(V_n, V_t, P, S)$ is CFG , then any string α *in*

$(V_t \cup V_n)^*$ such that $S =^{*}\Rightarrow \alpha$ is sentential form.

8.3 Context-Free Grammar Language

If $G=(V_n, V_t, P, S)$ is a CFG, the language of G, denoted by L(G), is the set of terminal strings that have derivations from the start symbol. That is

$L(G) = \{w \text{ in } V_t / S =^{*}\Rightarrow G\ \ w\}$

If a language L is language of some context-free grammar, then L is said to be context-free language, or CFL.

Example 8.2: Consider a grammar G=(V_n,V_t, P, S) where V_n = {S}, V_t = {a,b} and set of production P is given by P = {S $\longrightarrow$ aSb, $S \longrightarrow ab$}

Here S is the only non-terminal which is the starting symbol for the grammar; 'a' and 'b' are terminals. There are two production S $\longrightarrow$ aSb and S $\longrightarrow$ ab. Now we will show how the string a^2b^2 can be derived.

$S \Longrightarrow aSb$

$\Longrightarrow aabb$

$\Longrightarrow a^2b^2$

Here we need to apply the first production then second production. By applying first production n-1 times, followed by an application of second production, we get

$S \Longrightarrow aSb$

$\Longrightarrow aaSbb$

$\Longrightarrow$:

:

$\Longrightarrow a^{n-1}\ S\ b^{n-1}$

$\Longrightarrow a^n b^n$

Hence we say that language for the above grammar is

L(G) = { $a^n b^n$ / n>=1}

Example 8.3: Following is a CFG for the language

L(G) = {wcw^R / w $(a,b)^*$ }

Solution. Let G be CFG for language L= {wcw^R / w $(a,b)^*$ }

G=(V_n,V_t, P, S)

here V_n = {S}, V_t = {a, b, c} and P is given by

S $\longrightarrow$ aSb

$S \longrightarrow bSb$

$S \longrightarrow c$

Let us check that abbcbba can be drived from the given CFG.

$S \Longrightarrow aSa \quad (use\ the\ S \longrightarrow aSb)$

$\Longrightarrow abSba\ (use\ the\ S \longrightarrow bSb\ )$

$\Longrightarrow abbSbba\ (use\ the\ S \longrightarrow bSb\ )$

$\Longrightarrow abbcbba\ (use\ the\ S \longrightarrow c)$

Example 8.4: Write a CFG for the regular expression

$R=0^{*}1(0+1)^{*}$

Solution. Let us analyzes regular expression

$R=0^{*}1(0+1)^{*}$

Clearly regular expression is the set of string which starts with any number of 0's followed by a one and end with any combination of 0's and 1's.

Let CFG be G=(V_n, V_v, P, S)

here V_n = {S, A, B}

V_t = {0, 1}

and productions P are defined as

$S \longrightarrow A1B$

$A \longrightarrow 0A \mid \in$

$B \longrightarrow 0B \mid 1B \mid \in$

Let us see the derivation of the string 00101

$S \Longrightarrow A1B$

$\Longrightarrow 0A10B$

$\Longrightarrow 00A101B$

$\Longrightarrow 00101$

So clearly G is CFG for regular expression.

Example 8.5: Write a CFG which generates strings having equal number of a's and b's.

Solution . Let CFG be

G=(V_n,V_t, P, S)

V_n = {S}

V_t = {a, b}

Where P is defined as S $\longrightarrow$ $aSbS \mid bSaS \mid \in$

W=bbabaabbaa

$$S \Longrightarrow bSaS$$

$\Longrightarrow bbSaSaS$

$\Longrightarrow bbaSbSaSaS$

$\Longrightarrow bbaSbaSbSaSaS$

$\Longrightarrow bbaSbaaSbSbSaSaS$

$\Longrightarrow bbabaaSbSbSaSaS$

$\Longrightarrow bbabaabaSbSaSaS$

$\Longrightarrow bbabaabbSaSaS$

$\Longrightarrow bbabaabbaSaS$

$\Longrightarrow bbabaabbaaS$

$\Longrightarrow bbabaabbaa$

Example 8.6: Design a CFG for the regular expression $r = (a + b)^*$

Solution : Let G be CFG

$G=(V_n, V_t, P, S)$

$V_n = \{S\}$

$V_t = \{a, b\}$

P are defined as

$S \longrightarrow aS$

$S \longrightarrow bS$

$S \longrightarrow a$

$S \longrightarrow b$

$S \longrightarrow \in$

The word ab can be generated by the derivation

$S \Longrightarrow aS$

$\Longrightarrow abS$

$\Longrightarrow ab\in$

$\Longrightarrow ab$

Or by the derivation $S \Longrightarrow aS$

$\Longrightarrow ab$

Example 8.7: Design CFG for $V_t=\{a,b\}$ that generate the set of

(a) All strings with exactly one a.

(b) All strings with at least one a.

(c) All strings with at least 3 a's.

Solution . (a) Let CFG be G_1

$G_1=(V_n, V_t, P, S)$

$V_n = \{S\}$

$V_t = \{a, b\}$

P is defined as

$S \longrightarrow AaA$

$A \longrightarrow bA \mid \in$

Let us derive a string bbaab

$S \Longrightarrow AaA$

$\Longrightarrow bAaA$

$\Longrightarrow bbAaA$

$\Longrightarrow bbAabA$

$\Longrightarrow bbAab\in$

$\Longrightarrow bbAab$

We don't have any choice for deriving second G so string bbaab cannot be derive from $G_{1.}$

(b) Let CFG be G_2

$G_2=(V_n, V_t, P, S)$

$V_n = \{S, A\}$

$V_t = \{a, b\}$

P is defined as

$S \longrightarrow AaA$

$A \longrightarrow aA \mid bA \mid \in$

Let us derive string baab

$S \Longrightarrow AaA$

$\Longrightarrow bAaA$

$\Longrightarrow baAaA$

$\Longrightarrow ba\in AaA$

$\Longrightarrow baaA$

$\Longrightarrow baabA$

$\Longrightarrow baab\in$

$\Longrightarrow baab$

(c) Let CFG be G_3

and $G_3=(V_n, V_t, P, S)$

$V_n = \{S, A\}$

$V_t = \{a, b\}$

P is defined as

$S \rightarrow AaAaAaA$

$A \rightarrow aA \mid bA \mid \in$

8.4 Derivations

For any string, derivable from start symbol of the grammar, using the productions of the grammar, there are two different derivations possible namely,

(1) Leftmost derivation

(2) Rightmost derivation

8.4.1 Leftmost Derivation

If at each step in a derivation, a production is applied to the leftmost variable (non-terminal), then the derivation is called as leftmost derivation.

Example 8.8: Let the grammar G be consist of

({E}. {+, *, id}, P, E)

Where P is,

(i) E $\longrightarrow$ E + E

(ii) E $\longrightarrow$ E * E

(iii) E $\longrightarrow$ id

and we want to generate the string "id + id"

Let us start with the start symbol i.e. 'E'.

$E \Longrightarrow$ E + E , production (i)

$\Longrightarrow$ id + E, production (iii) and applied to leftmost non-terminal

$\Longrightarrow$ id + id , production (3), applied to 'E' which is the only non-terminal.

8.4.2 Rightmost Derivation (or, Canonical Derivation)

If at each step in a derivation, a production is applied to the rightmost variable (non-terminal), then the derivation is called as rightmost derivation.

Example 8.9: Let the grammar G be consist of

({E}. {+, *, id}, P, E)

Where P is,

(i) E $\longrightarrow E + E$

(ii) E $\longrightarrow E * E$

(iii) E $\longrightarrow id$

and we want to generate the string *"id + id * id"*

Let us start with the start symbol i.e. *'E'.*

$E \Longrightarrow$ E + E , production (i)

$\Longrightarrow$ E + E * E , production (ii), applied to rightmost 'E'

$\Longrightarrow$ E + E * id, production (iii)

$\Longrightarrow$ E + id * id, production (iii)

$\Longrightarrow$ id + id * id, production (iii)

Above derivation is rightmost derivation as we are, replacing the rightmost variable by the right-hand side of the production at every step in derivation.

8.5 Ambiguity in Context Free Grammar

The CFG for a language is said to be ambiguous if there exist at least one string which can be generated (or, derived) in more than one way. That means, there can be more than one rightmost derivation, more than one rightmost derivations and also more than one derivation trees associated with such a string.

Example 8.10: Consider the following grammar,

E $\longrightarrow E + E \mid E * E \mid id$

Let us derive the string *"id + id * id"*

(i) Using Leftmost Derivation:

E =⇒ E +E

=⇒ id +E

=⇒ id +E * E

=⇒ id + id * E

=⇒ id + id * id

OR

E =⇒ E + E

=⇒ E + E * E

=⇒ id + E * E

=⇒ id + id * E

=⇒ id + id * id

(ii) Using Rightmost Derivation:

E =⇒E + E

=⇒E + E * E

=⇒E + E * id

=⇒ E + id * id

$\Longrightarrow$ id + id * id

OR

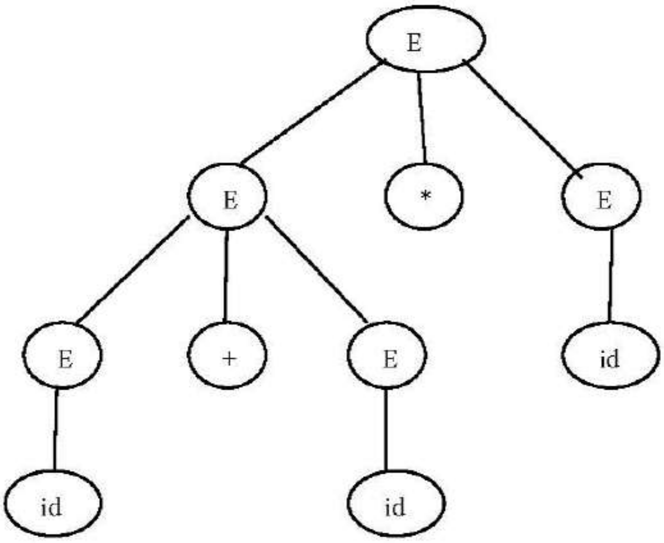
E $\Longrightarrow$ E * E

$\Longrightarrow$ E * id

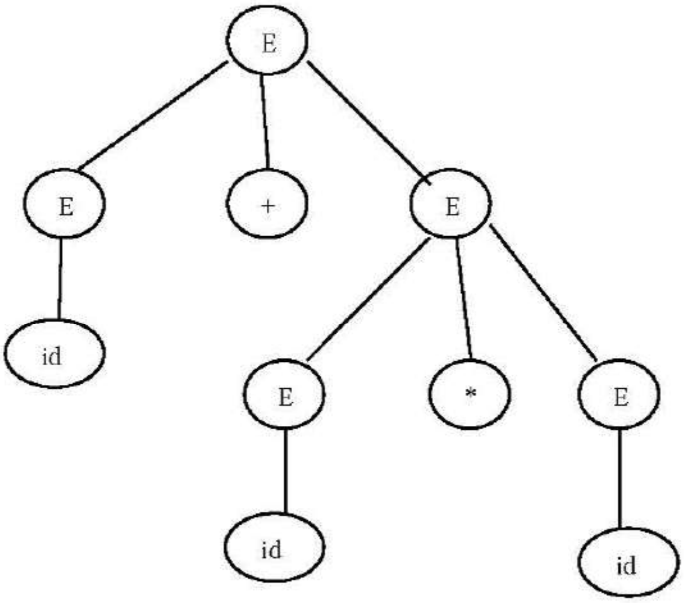
$\Longrightarrow$ E + E * id

$\Longrightarrow$ E + id * id

$\Longrightarrow$ id + id * id



(a) First derivation tree for "id + id * id"



(b) Second derivation tree for "id + id * id"

Figure 8.1: Ambiguous Grammar

There can be two derivation trees as shown in figure 8.1, as there are 2 different ways of deriving, one using start production as '$E \longrightarrow E + E$' and other using

$E \longrightarrow E * E$'.

Thus, the grammar given is ambiguous. As there are two ways to derive, then the question is, which one is the right derivation, or both are correct. The answer is, first way to derive i.e. to start with 'E + E' is correct as id + id * id = id + (id * id) because '*' has higher priority than '+'. Therefore, basic operation is '+' of 'id' and 'id * id'. Derivation tree 8.1 (a) is correct.

8.5.1 Removal of Ambiguity

We can remove the ambiguity to get the CFG which is equivalent to the ambiguous grammar i.e. generating the same set of words (same language). One technique while removing the ambiguity from an expression grammar, one similar to what we have seen, is postpone the higher priority operation or assign a separate non-terminal to take care of that. This can be explained as follows.

Example 8.11: Consider the grammar,

$E \longrightarrow E + E \mid E * E \mid id$

Here, the problem of ambiguity is due to fact that 'E' is the only non-terminal deriving two separate operations with different priorities. Here, '*' has higher priority than '+'.

The grammar after removing the ambiguity i.e. the unambiguous grammar can be written as follows,

$\mathrm{E} \longrightarrow \mathrm{E} + \mathrm{T} \mid \mathrm{T}$

$\mathrm{T} \longrightarrow \mathrm{T} * \mathrm{F} \mid \mathrm{F}$

$\mathrm{F} \longrightarrow \mathrm{id}$

This is obviously equivalent to the previous one. Let us derive the string "id + id * id", the same string for which we had detected ambiguity in the grammar, using this unambiguous grammar.

(i) Leftmost derivation

$\mathrm{E} \Longrightarrow \mathrm{E} + \mathrm{T}$

$=\Rightarrow T + T$

$=\Rightarrow F + T$

$=\Rightarrow id + T$

$=\Rightarrow id + T * F$

$=\Rightarrow id + F * F$

$=\Rightarrow id + id * F$

$=\Rightarrow id + id * id$

(ii) Rightmost derivation

$E =\Rightarrow E + T$

$=\Rightarrow E + T * F$

$=\Rightarrow E + T * id$

$=\Rightarrow E + F * id$

$=\Rightarrow E + id * id$

$=\Rightarrow T + id * id$

$=\Rightarrow F + id * id$

$=\Rightarrow id + id * id$

(iii) Derivation tree

Now, as we can see, there is only one way to deriving the required string. Therefore, there will be only one derivation tree as shown in the figure. 8.2.

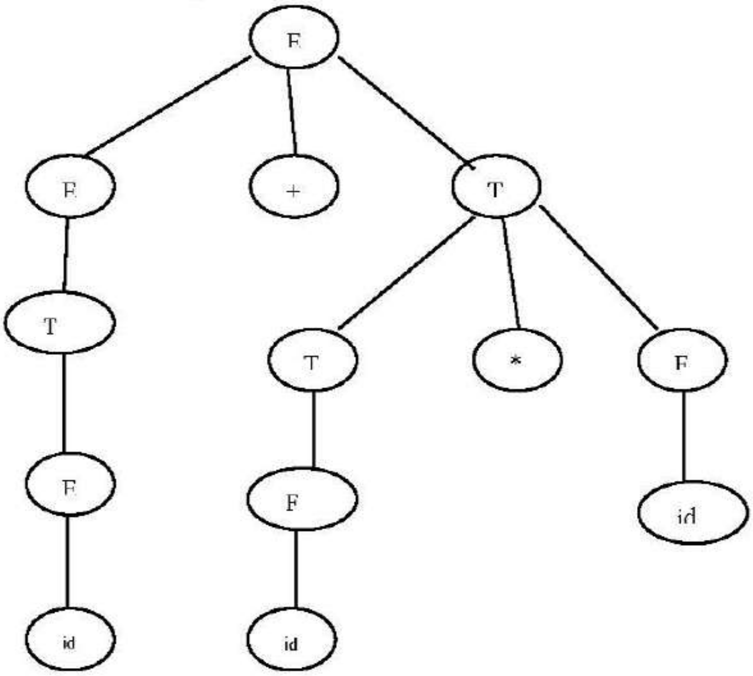


Figure: 8.2 Unambiguous Grammar (Derivation tree for "id + id * id")

8.6 Simplification of CFG

Following are the rules for having the given context-free grammar in the reduced form:

(1) Each variable and each terminal of CFG should appear in derivation of at least one word in L(G).

(2) There should be production of the form A ⟶ B, where 'A' and 'B' are both non-terminals.

8.6.1 Removal of Useless Symbols

A symbol 'X' is useful, if there exists a derivation, $S \overset{*}{\Rightarrow} \alpha X \beta \overset{*}{\Rightarrow} w$, Where, '$\alpha$', '$\beta$' are sentential forms and 'w' is any string in T^*; $(w \subset T)$.

Otherwise, if no such derivation exists, then symbol 'X' won't appear in any of the derivations that means, 'X' is a useless symbol.

Three aspects of useless of a non-terminal 'X' are as follows:

(i) Some string must be derived from 'X'.

(ii) 'X' must appear in the derivation of at least one string derivable from 'S' (start symbol).

(iii) It should not occur in any sentential form that contains a variable form which no terminal string can be derived.

Example 8.12: G =(V, T, P, S) where V = {S, T, X}, T= {0,1}

$S \longrightarrow 0T \mid 1T \mid X \mid 0 \mid 1$ rule 1

$T \longrightarrow 00$ rule 2

Now, in the above CFG, the non-terminals are S, T and X. To derive some string we have to start from start symbol S.

S

0T $S \longrightarrow 0T$

00 $T \longrightarrow 00$

Thus we can reach to certain string after following these rules.

But if S ⟶ X then there is no further rule as a definition to X. That means there is no point in the rule S ⟶ X. Hence we can declare that X is a useless symbol. And we can remove this so after removal of this useless symbol CFG becomes

G =(V, T, P, S) where V = {S, T}, T= {0,1} and

P = {S ⟶ 0T | 1T | 0 | 1

T ⟶ 00}

S is start symbol.

Example 8.13: G = (V, T, P, S) where V = {S, A, B}, T= {0, 1} and

P = {S ⟶ A 11 B | 11A

S ⟶ B | 11

A ⟶ 0

B ⟶ BB} For removing useless symbols.

Solution. Now in the given CFG if we try to derive any string A given some terminal symbol as 0 but B does not give any terminal string. By following the rules with B we simply get sample no. of B's and no significant string. Hence we can declare B as useless symbol and can remove the rules associated with it. Hence after removal of useless symbol we get,

S ⟶ A 11| 11

A ⟶ 0

8.6.2 Removal of Unit Production

A production of the form 'A ⟶ *B*' where, 'A' and 'B' both are non-terminals, are called as unit productions. All other productions(including ϵ- production) are non-unit productions.

Elimination Rules:

For every pair of non-terminals 'A' and 'B',

i) If CFG has a unit production of the form 'A ⟶ B' or

ii) If there is chain of unit productions leading from 'A' to 'B' such as,

$$A \Longrightarrow X_1 \Longrightarrow X_2 \ldots\Longrightarrow B$$

Where, all X_i s ($i > 0$) are non-terminals, then introduce new production(s) according to the rule stated as follows:

If the non-unit production for 'B' are

$B \longrightarrow \alpha_1 \mid \alpha_2 \mid \ldots.$

Where, $\alpha_1, \alpha_2, \ldots..$ are all sentential forms(not containing only one non-terminal)

Then, create the production for 'A' as,

$A \longrightarrow \alpha_1 \mid \alpha_2 \mid \ldots.$

Example 8.14: If the CFG is as below

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

$A \longrightarrow 0S \mid 00$

$B \longrightarrow 1 \mid A$

$C \longrightarrow 01$

Then remove the unit productions.

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

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

Similarly $B \longrightarrow A$ is also a unit production so we can modify it as

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

Thus finally we can write CFG without unit production as

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

$A \longrightarrow 0S \mid 00$

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

$C \longrightarrow 01$

Example 8.15: Optimize the CFG given below by reducing the grammar, S is start symbol.

$S \longrightarrow A \mid 0C1$

$A \longrightarrow B|01 \mid 10$

$C \longrightarrow \epsilon \mid CD$

Solution : S $\longrightarrow$ A $\longrightarrow$ B is a unit production

C $\longrightarrow$ ϵ is null production

C $\longrightarrow$ CD, B and D are useless symbol.

Reduction a grammar we have to avoid all the above conditions.

Let S $\rightarrow$ A

i.e. A $\rightarrow$ B is a useless symbol because B is not defined further more.

S $\rightarrow$ 01 | 10

i.e. S $\longrightarrow$ 01 | 10 | 0C1

but C $\rightarrow$ ϵ

Hence ultimately S $\rightarrow$ 01 | 10

A $\longrightarrow$ B but we can remove this production since B is a useless symbol.

Hence A $\longrightarrow$ 01 | 10

But the start symbol S $\longrightarrow$ 01 | 10

There is no A in the derivation of A so by considering A also as a useless symbol we get final CFG as

S $\longrightarrow$ 01 |10

8.6.3 Elimination of ϵ- production

Production of the form 'A $\longrightarrow\epsilon$' where, 'A' is any non-terminal is called as ϵ-production.

Surely, if 'ϵ' is in L(G) for grammar 'G', we cannot eliminate all ϵ- productions from ' G ', but is ϵ is not in L(G), then we can.

Theorem : If 'L' is a CFG generated by a CFG that include ϵ- production, then there exist another CFG without ϵ- productions which generates either the whole language 'L' (if 'L' does not contain the word 'ϵ') or else generates a language containing all the word of 'L' but not 'ϵ' i.e. L – {ϵ}.

The eliminate method is based on the concept of nullable non-terminals.

Nullable non-terminal:

In a given CFG, if there is a non-terminal 'N' and a production ' N $\longrightarrow \in$ ' or, 'N' is deriving ' $\in$ ' in more than one steps *'N* $\overset{*}{\Longrightarrow}$ $\in$' ,then 'N' is called as nullable non-termianl.

Example 8.16:

$A \longrightarrow B \mid \in$

$B \longrightarrow a$

In given grammar, 'A' is nullable non-terminal, but 'B' is not nullable.

Elimination Procedure:

The procedure to get rid of $\in$ - productions can be stated as follows:
The steps involved are

i) Delete all $\in$ - production from the grammar

ii) Identify nullable non-terminals

iii) If there is a production of the form 'A $\longrightarrow \alpha$', where 'α' is any sentential form containing at least one nullable non-terminal, then add new productions having right hand Side formed by deleting all possible subsets of nullable non-terminals from 'α'.

iv) If using step (iii) above, we get production of the form 'A $\longrightarrow \in$ ' then, do not add that to the final grammar.

Example 8.17: For the CFG given below remove the $\in$ - production

$S \longrightarrow aSa$

$S \longrightarrow bSb$

$S \longrightarrow \in$

Solution . Now the $\in$ - production is $B \longrightarrow \in$ we will delete this production. And then add the production having B replaced by $\in$

$A \longrightarrow 0\ B\ 1$

If $B = \in$

$A \longrightarrow 0\ 1$

Similarly $A \longrightarrow 1\ B\ 1 \longrightarrow 11$

$A \longrightarrow 0\,B\,1 \mid 1\,B\,1 \mid 01 \mid 11$

Similarly $B \longrightarrow 0\,B$

If $B = \in$

$B \longrightarrow 0$

As well as $B \longrightarrow 1$

$B \longrightarrow 0\,B \mid 1\,B \mid 0 \mid 1$

Collectively, we can write

$A \longrightarrow 0\,B\,1 \mid 1\,B\,1 \mid 01 \mid 11$

$B \longrightarrow 0\,B \mid 1\,B \mid 0 \mid 1$

8.7 Self Learning Exercise

Q.1 A context free language is called ambiguous if

a) It has two or more leftmost derivations for some terminal string W ∈ L (G)

b) It has two or more rightmost derivations for some terminal string W ∈ L (G)

c) Both (a) and (b)

d) None of these

Q.2 The context free grammar S ⟶ SS | 0S1 | 1S0 | ϵ generates

a) Equal number of 0's and 1's

b) Unequal number of 0's and 1's

c) Any number of 0's followed by any number of 1's

d) None of these

Q.3 Consider the grammar G as given below S ⟶ *aSa* | *bSb* | ϵ Which of the following strings cannot be generated using the grammar G?

(a) aabbaabb

(b) bbaabaab

(c) aaabbbaaa

(d) none of these

8.8 Summary

Context free grammar is a way of describing language by recursive rules called productions. A CFG consists of a set of variables, a set of terminal symbols and a start variable, as well as the productions. Each production consists of a head variable and a body consisting of a string of zero or more variables and/or terminals.

Beginning with the start symbol, derive strings by repeatedly replacing a variable by the body of some production with that variable in the head.

The language of the CFG is the set of terminal strings can so derive; it is called a context-free language.

For some CFG's, it is possible to find a terminal string with more than one parse tree, or equivalently, more than one rightmost derivation. Such a grammar is called ambiguous.

8.9 Glossary

Derivation tree - graphical representation or description of how the sentence has been derived or generated using the grammar.

8.10 Answers to Self-Learning Exercise

Q. 1 (c)

Q. 2 (a)

Q. 3 (d)

8.11 Exercise

Q.1 FORTRAN is a

(a) Regular language

(b) Context free grammar

(c) Context sensitive language

(d) None of these

Q.2 Which of the following does not belong to the context free grammar?

(a) Terminal symbol

(b) Non-terminal symbol

(c) Start symbol

(d) End symbol

Q.3 Show that the grammar $S \longrightarrow a \mid abSb \mid aAb$ $A \longrightarrow bS \mid aAAb$ is ambiguous.

Q.4 Design a CFG for the language $L=\{a^n b^{2n} \mid n >= 0\}$

Q.5 Consider a CFG $S \longrightarrow AB \mid a$ $A \longrightarrow b$ Identified and eliminate useless symbols.

8.12 Answers of Exercise

Q.1 (b)

Q.2 (d)

Q.3 Solution. Given grammar

$S \longrightarrow a$

$S \longrightarrow abSb$

$S \longrightarrow aAb$

$A \longrightarrow bS$

$A \longrightarrow aAAb$

Let us consider a string w = abab, it has two different derivations, as follows:

(i) $S \Longrightarrow abSb \Longrightarrow abab$

(ii) $S \Longrightarrow aAb \Longrightarrow abSb \Longrightarrow abab$

Q.4 Solution : Let CFG be G for the language L

$G = (V_n, V_t, P, S)$

$V_n = \{S\}$

$V_t = \{a, b\}$

and P is defined as

$S \longrightarrow aSbb / \in$

Let derive a string aaabbbbbb

$S \Longrightarrow aSbb$

$\Longrightarrow aaSbbbb$

$=\Rightarrow aaaSbbbbbb$

$=\Rightarrow aaa\in bbbbbb$

$=\Rightarrow aaabbbbbb$

Which is required string.

Q.5 Solution. Given CFG is

$S \rightarrow AB \mid a$

$A \rightarrow b$

Here by observing the given CFG, it is clear that B is non-generating symbol. Since A derive b, S derive a, but B is not deriving any string w.

So we can eliminate S S → AB from the context free grammar, now CFG becomes

$S \rightarrow a$

$A \rightarrow b$

Here A is non-reachable symbol, since it cannot be reached by starting non-terminal S.

$S \rightarrow a$

References and Suggested Readings

1. Theory of Computer Science (Automata, Language and Computation) by K.L.P. Mishra and N.Chandrasekaran, PHI Learning, 2009.
2. Theory of Computation by A.A. Puntambekar, Technical Publication Pune, 2009.
3. Theory of Automata and computation by Adesh K. Pandey, Katson books, 2014.
4. Introduction to Automata Theory, Language, and Computation by Jhon E. Hopcroft, Rajeev Motwani and Jeffrey D. Ullman, Pearson Education, 2001.

UNIT-9
Simplification of CFG

Structure of the Unit

9.0 Objective

In this chapter we shall focus upon the following topics

- Chomsky Normal Form
- Greibach Normal Form
- Decision Algorithms for Context Free Languages
- Pumping Lemma for Context free Languages

9.1 Introduction

The definition of context free grammars (CFGs) allows us to develop a wide variety of grammars. Most of the time, some of the productions of CFGs are not useful and are redundant. This happens because the definition of CFGs does not restrict us from making these redundant productions.

By simplifying CFGs we remove all these redundant productions from a grammar , while keeping the transformed grammar equivalent to the original grammar.

In this module two normal forms are presented for CFG:

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

The pumping lemma for context-free languages describes a property that all context-free languages are guaranteed to have. It is used to check given grammar is regular or not.

9.2 Chomsky Normal Form

A grammar where every production is either of the form A → BC or A → c (where A, B, C are arbitrary variables and c an arbitrary symbol).
i.e.
A context-free grammar G is in Chomsky normal form if every rule is of the form:

A → BC or A → a
Where a is a terminal, A,B and C are nonterminals, and B,C may not be the start variable.

Example 9.1

S → AS | a

A → SA | b

(If language contains ε, then we allow S → ε where S is start symbol, and forbid S on RHS.)
The conversion to Chomsky Normal Form has four main steps:

[A new start symbol S' may be added with the production S' →S , S is start symbol of given CFG , this change guarantees that the start symbol of CNF Grammar does not occur on the right hand side of any rule]

1. Eliminate the ε productions.

2. Remove the unit rules.

3. Replace every production that is too long by shorter productions.

4. Move all terminals to productions where RHS is one terminal.

1. Eliminate ε Productions

Determine the nullable variables (those that generate ε). Go through all productions, and for each, omit every possible subset of nullable variables.

For example, if P→ AxB with both A and B nullable, add productions

P → xB | Ax | x. After this, delete all productions with empty RHS.

2. Eliminate Variable Unit Productions

A unit production is where RHS has only one symbol. Consider production

A → B. Then for every production B → α, add the production A → α. Repeat until done .

3. Replace Long Productions by Shorter Ones

For example, if have production A → BCD, then replace it with A → BE and E → CD. (In theory this introduces many new variables but one can re-use variables if careful.)

4. Move Terminals to Unit Productions

For every terminal on the right of a non-unit production, add a substitute variable. For example, replace production A → bC with productions A → BC and B → b

Example 9.2

Consider the CFG:

S → aXbX

X → aY | bY | ϵ

Y → X | c

The variable X is nullable; and so therefore is Y . 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

Now, break up the RHSs of S; and replace a by A, b by B and c by C wherever not units:

S → EF | AF | EB | AB

X →AY | BY | a | b

Y → AY | BY | a | b | c

E → AX

F →BX

A → a

B → b

C → c

Example 9.3

Convert the following CFG into Chomsky Nomal Form:

S → AbA

A → Aa | ϵ

After the first step, one has:

S → AbA | bA | Ab | b

A → Aa | a

The second step does not apply. After the third step, one has:

S → TA | bA | Ab | b

A → Aa | a

T → Ab

And finally, one has:

S → TA | BA | AB | b

A →AC | a

T → AB

B → b

C → a

Example 9.4

Consider the grammar whose rules are:

$S \to ASA\ |aB$
$A \to B\ |S$
$B \to b\ |\epsilon$

After first step of transformation we get:

$S_0 \to S$
$S \to ASA\ |aB$
$A \to B\ |S$
$B \to b\ |\epsilon$

Removing $B \to \epsilon$

$S_0 \to S$
$S \to ASA\ |aB|a$
$A \to B\ |S|\epsilon$
$B \to b$

Removing $A \to \epsilon$

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

Removing $S \to S$

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

Removing $S_0 \to S$

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

Removing $A \rightarrow B$

$S_0 \rightarrow ASA\ |aB|a|SA|AS$
$S \rightarrow ASA\ |aB|a|SA|AS$
$A \rightarrow B\ |S$
$B \rightarrow b$

Removing $A \rightarrow S$

$S_0 \rightarrow ASA\ |aB|a|SA|AS$
$S \rightarrow ASA\ |aB|a|SA|AS$
$A \rightarrow b|ASA\ |aB|a|SA|AS$
$B \rightarrow b$

Converting remaining rules

$S_0 \rightarrow AA_1\ |UB|a|SA|AS$
$S \rightarrow AA_1\ |UB|a|SA|AS$
$A \rightarrow b|AA_1\ |UB|a|SA|AS$
$A_1 \rightarrow SA$
$U \rightarrow b$
$B \rightarrow b$

NOTE: The conversion procedure produces several variables $\mathrm{U_i}$ along with several rules $\mathrm{U_i} \rightarrow \mathrm{a}$. Since all these represent the same rule, we may simplify the result using a single variable U and a single rule $\mathrm{U} \rightarrow \mathrm{a}$

9.3 Greibach Normal Form

A context-free grammar is in Greibach normal form (GNF) if the right-hand sides of all production rules start with a terminal symbol, optionally followed by some variables. A non-strict form allows one exception to this format restriction for allowing the empty word (epsilon,ϵ) to be a member of the described language. The normal form was established by Sheila Greibach and it bears her name.

More precisely, a context-free grammar is in Greibach normal form, if all production rules are of the form:

$A \rightarrow aA_1A_2\,A_3\ \cdots\cdots A_n$

Or

$S \rightarrow \epsilon$

where A is a nonterminal symbol, a is a terminal symbol, $A_1A_2\,A_3\,\cdots\cdots A_n$ is a (possibly empty) sequence of nonterminal symbols not including the start symbol, S is the start symbol, and ϵ is the empty string.

Observe that the grammar does not have left recursions.

A CFG G = (V, T,R, S) is said to be in GNF if every production is of the form

A → $a\alpha$, where $a\epsilon\ T$ and $\alpha\ \epsilon V^*$ i.e., α is a string of zero or more variables.

Definition: A production $u\epsilon\ R$ is said to be in the form left recursion, if

U : A → $A\alpha$ for some $A\epsilon\ V$.

Left recursion in R can be eliminated by the following scheme:

$A \rightarrow A\alpha_1|\ A\alpha_2|\ A\alpha_3|\ \cdots A\alpha_r$ are all A left recursive rules and

$A \rightarrow \beta_1\ |\beta_2|\beta_3\ \cdots \beta_s$ are all remaining A-rules then chose a new nonterminal, say B

- Add the new B rules $B \rightarrow \alpha_i|\ \alpha_i\,B\ \ 1 \le i \le r$
- Replace the A-rules by $A \rightarrow \beta_i|\ \beta_i\,B\ \ 1 \le i \le s$

This construction preserves the language

If $G = (V, T, R, S)$ is a CFG, then we can construct another CFG $G_1 = (V_1, T, R_1, S)$ in Greibach Normal Form (GNF) such that

$L(G_1) = L(G) - \{\epsilon\}$

The stepwise algorithm is as follows:

1. Eliminate null productions, unit productions and useless symbols from the grammar G and then construct a $G' = (V', T, R', S)$ in Chomsky Normal Form (CNF) generating the language $L(G') = L(G) - \{\epsilon\}$
2. Rename the variables like $A_1A_2 \cdots A_n$starting with $S = A_1$
3. Modify the rules in R' so that $A_i \rightarrow A_j\gamma\ \in\ R'$ then $j\ > i.$
4. Starting with A_1 and proceeding to A_n this is done as follows:

(a) Assume that productions have been modified so that for $1 \le i \le k,$ $A_i \rightarrow A_j\gamma\ \in\ R'$ only if $j\ > i$

(b) If $A_k \rightarrow A_j\gamma$ is a production with $j < k$, generate a new set of productions substituting for the A_j the body of each A_j production.

(c) Repeating (b) at most k − 1 times we obtain rules of the form $A_k \to A_p\gamma$ $p \geq k$

(d) Replace rules $A_k \to A_k\gamma$ by removing left-recursion as stated above.

5. Modify the $A_i \to A_j\gamma$ for i = n−1, n−2, ., 1 in desired form at the same time change the B production rules.

Example 9.5: Convert the following grammar G into Greibach Normal Form (GNF).

S → XA|BB

B →b|SB

X → b

A → a

To write the above grammar G into GNF, we shall follow the following steps:

1. Rewrite G in Chomsky Normal Form (CNF) :It is already in CNF.
2. Re-label the variables

   S with A_1

   X with A_2

   A with A_3

   B with A_4

   After re-labeling the grammar looks like:

   $A_1 \to A_2A_3|A_4A_4$
   $A_4 \to b|A_1A_4$
   $A_2 \to b$
   $A_3 \to a$
3. Identify all productions which do not conform to any of the types listed below:

   $A_i \to A_jx_k$ such that $j > i$

   $B_i \to A_jx_k$ such that $j \leq n$

   $A_i \to ax_k$ such that $x_k \in V^*$ and $a \in T$
4. $A_4 \to A_1A_4$ …………… identified
5. $A_4 \to b|A_1A_4$

To eliminate A_1we will use the substitution rule $A_1 \rightarrow A_2A_3|A_4A_4$.

Therefore we have $A_4 \rightarrow A_2A_3A_4|A_4A_4A_4|b$. The above two productions still do not conform to any of the types in step 3.

Substituting for $A_2 \rightarrow b$

Now we have to remove left recursive production $A_4 \rightarrow A_4A_4A_4$

$A_4 \rightarrow bA_3A_4|b|bA_3A_4Z|bZ$
$Z \rightarrow A_4A_4|A_4A_4Z$

6. At this stage our grammar now looks like

$A_1 \rightarrow A_2A_3|A_4A_4$
$A_4 \rightarrow bA_3A_4|b|bA_3A_4Z|bZ$
$Z \rightarrow A_4A_4|A_4A_4Z$
$A_2 \rightarrow b$
$A_3 \rightarrow a$

All rules now conform to one of the types in step 3. But the grammar is still not in Greibach Normal Form!

7. All productions for A_2, A_3 and A_4 are in GNF For $A_1 \rightarrow A_2A_3|A_4A_4$

Substitute for $A_2$and A_4 to convert it to GNF

$A_1 \rightarrow bA_3|bA_3A_4A_4|bA_4|bA_3A_4ZA_4|bZA_4$
For $Z \rightarrow A_4A_4|A_4A_4Z$

Substitute for A_4 to convert it to GNF

$Z \rightarrow bA_3A_4A_4|bA_4|bA_3A_4ZA_4|bZA_4|bA_3A_4A_4Z|bA_4Z|bA_3A_4ZA_4Z|bZA_4Z$

8. Finally the grammar in GNF is

$A_1 \rightarrow bA_3|bA_3A_4A_4|bA_4|bA_3A_4ZA_4|bZA_4$
$A_4 \rightarrow bA_3A_4|b|bA_3A_4Z|bZ$
$Z \rightarrow bA_3A_4A_4|bA_4|bA_3A_4ZA_4|bZA_4|bA_3A_4A_4Z|bA_4Z|bA_3A_4ZA_4Z|bZA_4Z$
$A_2 \rightarrow b$
$A_3 \rightarrow a$

9.4 Decision Algorithms for Context Free Languages

As usual, when we talk about “a CFL” we really mean “a representation for the CFL, e.g., a CFG or a PDA accepting by final state or empty stack.

There are algorithms to decide if:

1. String w is in CFL L.
2. CFL L is empty.

3. CFL L is infinite.

Non-Decision Properties

Many questions that can be decided for regular sets cannot be decided for CFL's.

- Example: Are two CFL's the same?
- Example: Are two CFL's disjoint?

How would you do that for regular languages? Need more theory (Turing machines, decidability) to prove no algorithm exists.

Testing Emptiness

Theorem. 9.1 : Given CFG G, there is an algorithm for deciding if L(G) = $\emptyset$

Proof:

1. Use the algorithm for eliminating useless variables.
2. If the start variable is useless, return true (L(G) = $\emptyset$) else return false.

Testing infiniteness

Theorem. 9.2 : Given CFG G , $|L(G)| < \infty$ is decidable.

Proof:

1. Let G = (V, T, S, P).

   Assume G has no λ-productions, no unit productions, and no useless variables or symbols.
2. Suppose there is an A $\in$ V such that A $\Rightarrow$ *xAy.
3. x and y cannot both be empty, since G has no unit productions.
4. Since G has no useless variables $\Rightarrow$ * uAv$\Rightarrow$ * uzy, where u, v, and z $\in T^+$
5. Since A is repeating, S $\Rightarrow$ *uAv $\Rightarrow$ $* \, ux^nAy^n \Rightarrow* \, ux^nzy^nv$ for n = 0,1,…
6. Thus, under the assumption that there is a repeating variable, L(G) is infinite.
7. If there is no repeating symbol, then $|L(G)| < \infty$
8. The question thus reduces to whether there is a repeating variable.
9. There is an algorithm to decide if G has a repeating variable:

(a) Construct a directed graph where there is a node for each variable, and an directed edge from node A to node B if there is a production of the form A → xBy.

(b) If the graph has a cycle, there is a repeating variable.

(c) There is an algorithm for detecting whether or not a digraph has a cycle (e.g., depth first search).

Decision algorithm for testing finiteness of a CFL :

1. Test all input strings beginning with those of length n (in nondecreasing order of length) for membership. (we will discuss it later)

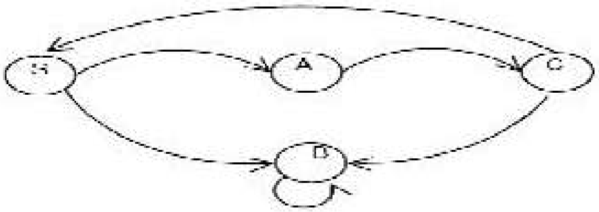
- If there is a string x with length $n \leq |x| < 2n$ such that $x \in L$, then L is infinite otherwise L is finite.

Proof : If $|x| \geq n$ and $x \in L$,, then x can be pumped according to the pumping lemma and the language is infinite. We need to test strings of length less than 2n only . Because if there were a string $z = uvwxy$ of length 2n or longer, we can always find a shorter string $uwy \in L$,, , (by pumping lemma), but it is atmost n shorter. Thus if there are any strings of length 2n or more we can repeatedly cut out the substring vx to get, eventually, a string whose length is in the range n to 2n-1.

Example 9.6

Let the gramamr be:

S→AB, A→aCb|a, B→bB|bb C→cBS



Since there is a loop in the dependency graph, the language is infinite. The derivation is S ⇒* $(acbb)^i S(bbb)^i$

Testing Membership

Given a CFL L and a string x, the membership, problem is to determine whether x ∈ L?

Given a PDA P for L, simulating the PDA on input string x does not quite work, because the PDA can grow its stack indefinitely on ε input, and the process may never terminate, even if the PDA is deterministic.

So, we assume that a CFG $G = (N, \Sigma, P, S)$is given such that L = L(G).

Let us first present a simple but in-efficient algorithm.

Convert G to in $G' = (N', \Sigma', P', S')$ CNF generating $L(G) - \{\epsilon\}$. If the input string $x = \varepsilon$, then we need to determine whether $S \underset{G}{\Rightarrow} \epsilon$ and it can easily be done using the technique given in the context of elimination of ϵ -production. If , $x \neq \epsilon$then $x \in L(G')$iff $x \in L(G)$. Consider a derivation under a grammar in CNF. At every step, a production in CNF in used, and hence it adds exactly one terminal symbol to the sentential form. Hence, if the length of the input string x is n, then it takes exactly n steps to derive x (provided x is in L(G')).

Let the maximum number of productions for any nonterminal in G' is K. So at every step in derivation, there are atmost k choices. We may try out all these choices, systematically., to derive the string x in G'. Since there are atmost $K^{|x|}$i.e. K^n choices. This algorithms is of exponential time complexity. We now present an efficient (polynomial time $O(n^3)$) membership algorithm.

CYK Algorithm to decide membership in CFL

We now present a cubic-time algorithm due to cocke, Younger and Kasami. It uses the dynamic programming technique-solves smaller sub-problems first and then builds up solution by combining smaller sub-solutions. It determines for each substring y of the given string x the set of all nonterminals that generate y. This is done inductively on the length of y.

CYK Algorithm

- Let w= $a_1 a_2 \cdots a_n$
- We construct an n-by-n triangular array X containing sets of variables.
- $X_{ij} = \{Variable\ A \mid A\ \geq *\ a_i \cdots a_j\}$
- Induction on j–i+1. The length of the derived string.
- Finally, ask if S is in X_{1n}

Basis : $X_{ii} = \{A \mid A \rightarrow a_i$ is a production$\}$

Induction :

$X_{ij} = \{A \mid there\ is\ a\ production\ A \rightarrow BC$ and an integer k with i ≤ k < $j, such\ that\ B\ is\ in\ X_{ik}$ and C is in $X_{k+1\ j}\}$

9.5 Pumping Lemma for Context free Languages

The pumping lemma gives us a technique to show that certain languages are not context free

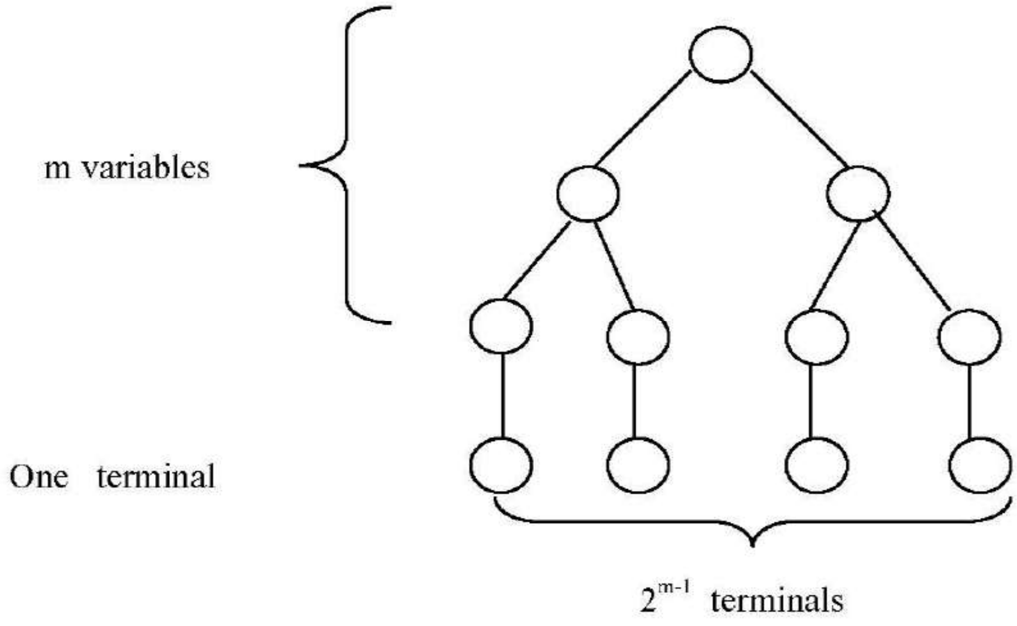
- Just like we used the pumping lemma to show certain languages are not regular
- But the pumping lemma for CFL's is a bit more complicated than the pumping lemma for regular languages

Informally The pumping lemma for CFL's states that for sufficiently long strings in a CFL, we can find two, short, nearby substrings that we can "pump" in tandem and the resulting string must also be in the language.

To prove the Pumping Lemma first another given lemma will be proved,

Lemma 9.1: A parse tree with yield z must have a path of length m+2 or more.

Proof : If all paths in the parse tree of a CNF grammar are of length <m+1, then the longest yield has length 2^{m-1}, as in:



Statement of the CFL Pumping Lemma

Let L be a CFL. Then there exists a constant p such that if z is any string in L where $|z| \geq p$, then we can write z = uvwxy subject to the following conditions:

1. $|vwx| \leq p$. This says the middle portion is not larger than p.
2. $vx \neq \epsilon$. We'll pump v and x. One may be empty, but both may not be empty.
3. For all $i \geq 0$, uv^iwx^iy is also in L. That is, we pump both v and x.

Given any context free grammar G, we can convert it to CNF. The parse tree creates a binary tree.

Let G have m variables. Choose this as the value for the longest path in the tree.

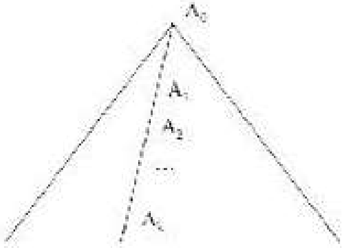
– The constant p can then be selected where $p = 2^m$.

– Suppose a string z = uvwxy where $|z| \geq p$ is in L(G)

- We showed previously that a string in L of length m or less must have a yield of 2m-1 or less.
- Since $p = 2^m$, then 2^{m-1} is equal to p/2.
- This means that z is too long to be yielded from a parse tree of length m.
    - What about a parse tree of length m+1?
- Choose longest path to be m+1, yield must then be 2^m or less
- Given $p=2^m$ and $|z| \leq p$ this works out
- Any parse tree that yields z must have a path of length at least m+1. This is illustrated in the following figure

Parse Tree

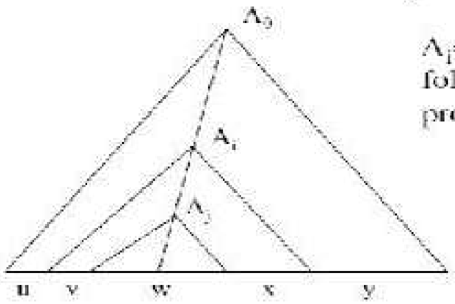
- z=uvwxy where $|z| \geq p$



- Variables A_0,A_1, … A_k

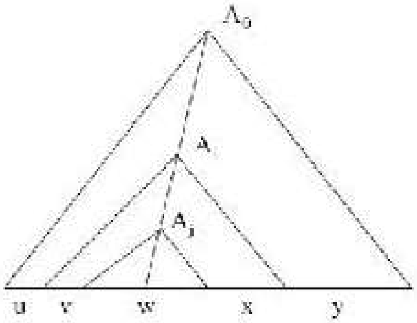
- If $k \geq m$ then at least two of these variables must be the same, since only m unique variables.

Suppose the variables are the same at $A_i = A_j$ where $k-m \leq i < j \leq k$



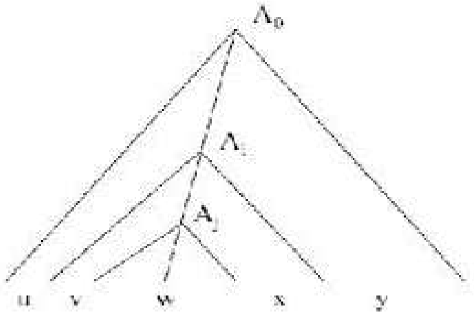
$A_i = A_j$ although we may follow different production rules for each

- Condition 2: $vx \neq \epsilon$
- Follows since we must use a production from A_i to A_j and can't be a terminal or there would be no A_j.
- Therefore we must have two variables; one of these must lead to A_j and the other must lead to v or x or both.
- This means v and x cannot both be empty but one might be empty.

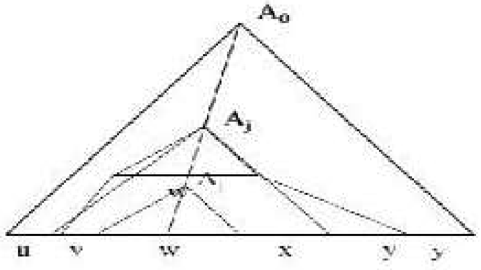


- Condition 1 stated that $|vwx| \leq p$
- This says the yield of the subtree rooted at A_i is $\leq p$
- We picked the tree so the longest path was m+1, so it easily follows that $|vwx| \leq p \leq 2^{m+1}-1$

(A_i could be A_0 so vwx is the entire tree)

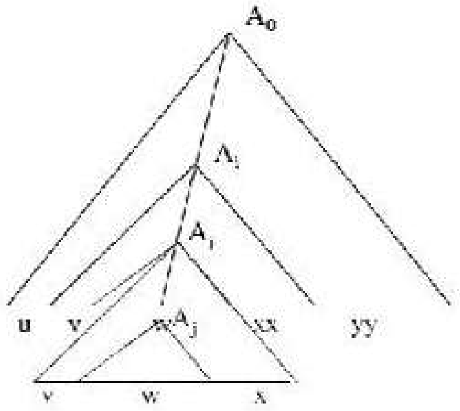


- Condition 3 stated that for all i ≥ 0, uv^iwx^iy is also in L
- We can show this by noting that the symbol A_i=A_j
- This means we can substitute different production rules for each other
- Substituting A_j for A_i the resulting string must be in L



- Substituting A_i for A_j
- Result:

uv^1wx^1y,
uv^2wx^2y, etc



- We have now shown all conditions of the pumping lemma for context free languages
- To show a language is not context free we

– Pick a language L to show that it is not a CFL

– Then some p must exist, indicating the maximum yield and length

– of the parse tree

– We pick the string z, and may use p as a parameter

– Break z into uvwxy subject to the pumping lemma constraints

– |vwx| $\leq$ p, |vx| $\neq \epsilon$

– We win by picking i and showing that uv^iwx^iy is not in L, therefore L is not context free

Applications of Pumping Lemma

Pumping lemma is used to check whether a grammar is context free or not. Let us take an example and show how it is checked.

Example :Find out whether the language L = $\{ x^n y^n z^n | \, n \geq 1\}$ is context free or not.

Solution : Let L is context free. Then, L must satisfy pumping lemma.

At first, choose a number n of the pumping lemma. Then, take z as $0^n 1^n 2^n$. Break z into uvwxy, where $|vwx| \leq n \; and \; vx \; \neq \epsilon$. Hence vwx cannot involve both 0s and 2s, since the last 0 and the first 2 are at least (n+1) positions apart. There are two cases −

Case 1 − vwx has no 2s. Then vx has only 0s and 1s. Then uwy, which would have to be in L, has n 2s, but fewer than n 0s or 1s.

Case 2 − vwx has no 0s.

Here contradiction occurs. L is not a context-free language.

Hence, **L** is not a context-free language.

9.6 Self Learning Exercise

1. Consider the following language $L = \{a^n b^n c^n d^n | n \geq 1\}$, L is

   A. CFL but not regular

   B. CSL but not CFL

C. regular

D. type 0 language but not type 1

2. While converting the context free grammar into Greibach normal form, which of the following is not necessary

A. Elimination of null production

B. Elimination of unit production

C. Converting given grammar in Chomsky normal form

D. None of these

3. Pumping lemma for context free grammar is used for

A. Proving certain languages are not context free

B. Proving language is infinite

C. Both (a) and (b)

D. None of these

9.7 Summary

There are special forms for CFGs such as Chomsky Normal Form, where every production has the form A → BC or A → C. The algorithm to convert to this form involves (1) determining all nullable variables and getting rid of all ε productions, (2) getting rid of all variable unit productions, (3) breaking up long productions, and (4) moving terminals to unit productions. The key advantage is that in Chomsky Normal Form, every derivation of a string of n letters has exactly $2n - 1$ steps.

The pumping lemma gives us a technique to show that certain languages are not context free. Informally The pumping lemma for CFL's states that for sufficiently long strings in a CFL, we can find two, short, nearby substrings that we can "pump" in tandem and the resulting string must also be in the language.

9.8 Glossary

CNF :A context-free grammar *G* is said to be in **Chomsky normal form** if all of its production rules are of the form:

$A \rightarrow BC$, or

$A \rightarrow a$, or

$S \rightarrow \varepsilon$,

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, optionally followed by some variables.

9.9 Answers to Self-Learning Exercise

1. B
2. D
3. A

9.10 Exercise

Q.1 A context free grammar G is in Chomsky normal form if every production is of the form

a) A → BC or A→ A

b) A → BC or A → a

c) A → BCa or B→ b

d) None of these

Q.2 Which of the following statement is false?

a) The context free language can be converted into Chomsky normal form

b) The context free language can be converted into Greibach normal form

c) The context free language is accepted by pushdown automata

d) None of these

Q.3 A context free language is called ambiguous if

a) It has two or more leftmost derivations for some terminal string $\omega \in L(G)$

b) It has two or more leftmost derivations for some terminal string $\omega \in L(G)$

c) Both (a) and (b)

d) None of these

Q.4 The context free grammar S → A111|S1, A → A0 | 00 is equivalent to

a) $\{0^n1^m \mid n=2, m=3\}$

b) $\{0^n1^m \mid n=1, m=5\}$

c) $\{0^n1^m$ | n should be greater than two and m should be greater than four}

d) None of these

Q.5 The context free grammar S → SS | 0S1 | 1S0 | ϵ generates

a) Equal number of 0's and 1's

b) Unequal number of 0's and 1's

c) Any number of 0's followed by any number of 1's

d) None of these

Q.6 CYK algorithm is named CYK because it was invented by

a) John Cocke, Tadao Kasami and Daniel H. Younger

b) Jass Carry, Tom Kosami and Daniel Richy

c) Cammy lenna, Yousaf, Kosami

d) None of these

Q.7 The CYK algorithm start with

a) Chomsky normal form grammar

b) Greibach normal form grammar

c) Both (a) and (b)

d) None of these

Q.8 Pumping lemma for context free language breaks the strings into

a) Two parts

b) Three parts

c) Five parts

d) Six parts

Q.9 In pumping lemma for context free language

a) We start by assuming the given language is context free and then we get contradict

b) We first convert the given language into regular language and then apply steps on

c) Both (a) and (b)

d) None of these

Q.10 The CYK algorithm constructs table from where we can conclude whether w is in L

a) O(n^2) time

b) O(n^3) time

c) O(n) time

d) None of these

9.11 Answers of Exercise

Q.1 a

Q.2 d

Q.3 c

Q.4 a

Q.5 a

Q.6 a

Q.7 a

Q.8 c

Q.9 a

Q.10 b

References and Suggested Readings

1. K.Krithivasan and R.Rama; Introduction to Formal Languages, Automata Theory and Computation; Pearson Education, 2009.

2. J.E.Hopcroft, R.Motwani and J.D.Ullman , "Introduction to Automata Theory Languages and computation", Pearson Education Asia , 2001.

3. Peter Linz, "An Introduction to Formal Language and Automata", 4th Edition, Narosa Publishing house , 2006.

4. M.Sipser; Introduction to the Theory of Computation; Singapore: Brooks/Cole, Thomson Learning, 1997.

5. John.C.martin, "Introduction to the Languages and the Theory of Computation",Third edition, Tata McGrawHill, 2003.
6. https://people.cs.clemson.edu/~goddard/texts/theoryOfComputation/9a.pdf
7. http://www.iitg.ernet.in/gkd/ma513/oct/oct18/note.pdf
8. http://www.cs.colostate.edu/~massey/Teaching/cs301/RestrictedAccess/Slides/301lecture17.pdf

UNIT-10

Pushdown Automata

Structure of the Unit

10.0 Objective

After reading this chapter you will be able to:

- Understand Pushdown Automata
- Build PDA (Pushdown Automata) using context Free Grammar
- Understand the relation between CFG and PDA
- Understand the Parsing mechanism using PDA

10.1 Introduction

In previous chapters, we have discussed the concept of FA and CFG with their acceptability. As FA accept regular languages like ab*, However Finite Automata has some limitations that it fails to accept context free language like L={$a^n cb^n$ | n>=0}.

It is noted that L has some number of a's and b's string separated by c. Since finite automata have strictly finite memories whereas L requires storing an unbounded string of a's and b's. Since n is unbounded, the counting cannot be done with finite memory. So due to these limitations we need machine that has ability to store and matches a sequence of symbol in reverse order.

In this unit we discuss a machine called 'Pushdown Automata' where we use a stack data structure for machining equal number of a's and b's without counting them directly.

The PDA is used in theories about how computing is done by machines. It is more powerful than a Finite State Machine but less capable then Turning machine. It can be used in designing a parser.

10.2 Basic Definition of PDA

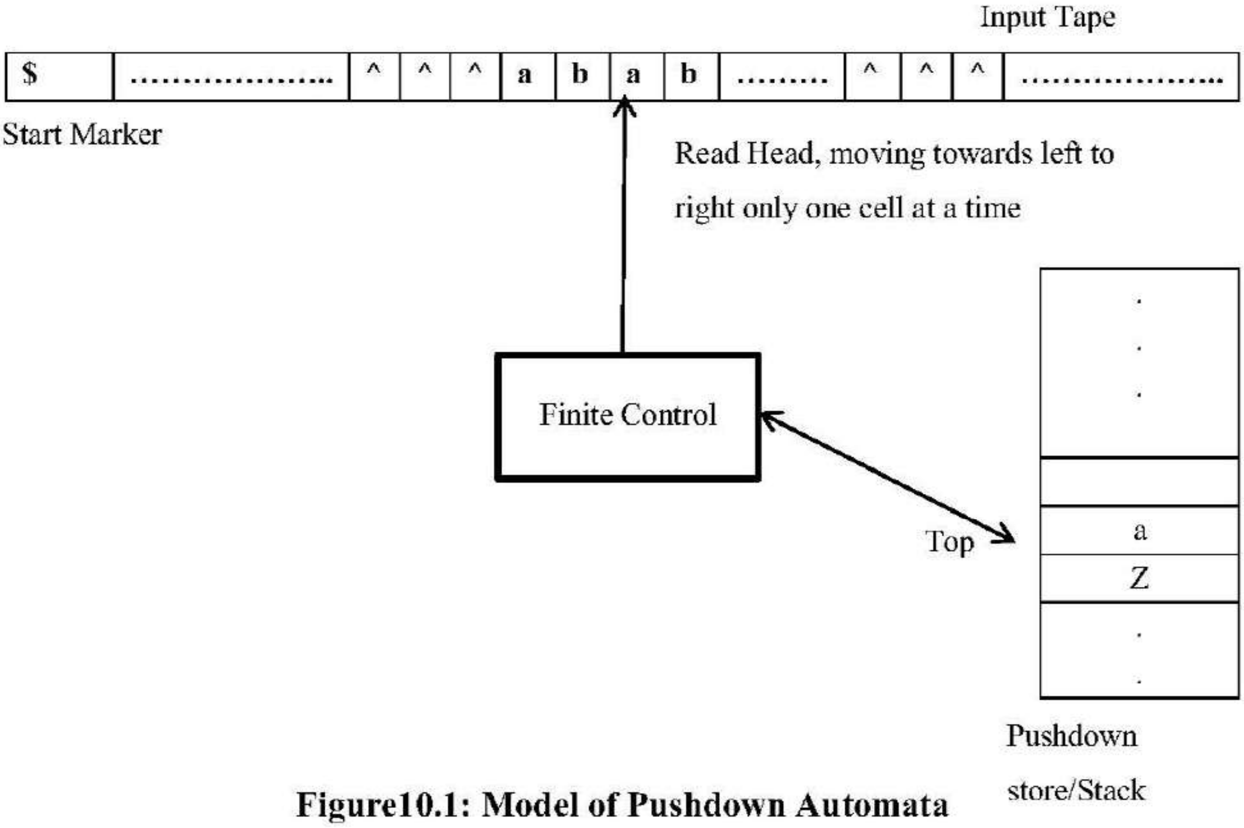


Figure10.1: Model of Pushdown Automata

Pushdown Automata (PDA) is a type of automation which is supported by stack. Stack is a data structure which allows operation like push (for inserting element) and pop (for deleting elements) in LIFO (Last in First out) manner. It also has a pointer called 'top' which pointing to the top element in the stack and inserting and deleting is done only from top to bottom of the stack. Basically a push down automation is *Finite state Machine + a Stack.*

There are mainly two differences between PDA and FA operation:-

- PDA uses the top of stack to decide which transition to take.
- PDA manipulates the stack as part of performing a transition.

Before giving a formal definition of PDA, we will discuss the model of pushdown automata (PDA) and the way of operations PDA can do. PDA has three major components:

i. **Input tape:** It is read only input tape divided into cells; each cell can hold symbol.

ii. **Finite store control:** This component decide the head movement and pop/push operation on the basic of input symbol and stack element. After doing these operations move to the new elements from PDS.

iii. **A Stack with infinite size:** A Stack also known as Pushdown store. It is read-write pushdown store i.e. we can add or remove elements from PDA.

A pushdown automaton (PDA) is formally defined as 7-tuples:

P = (Q, Σ, Γ, δ, q0, Z0, F), where

Q is a finite non empty set of state.

Σ is the finite set of input alphabet/symbol;

Γ is a finite set of stack alphabet;

δ is a transition function from Q × (Σ ∪ {^}) × F to the set of finite subset Q× Γ*

q0 ∈ Q is the initial state;

$Z0 \in \Gamma$ is the initial stack symbol; and

$F \subseteq Q$ is the set of accepting states.

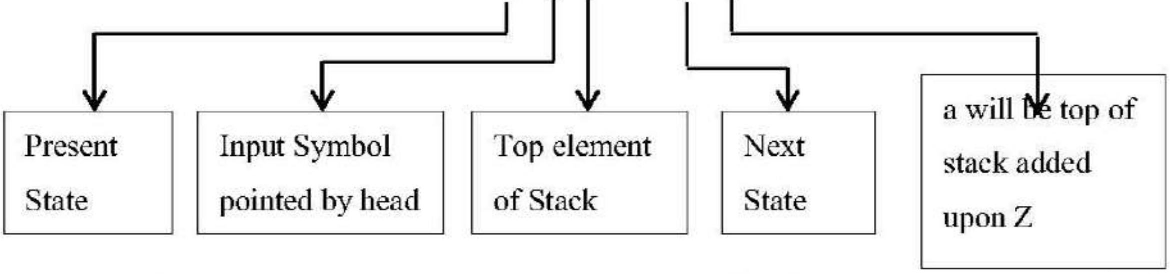
For better understanding the definition, we take an example.

Example: Suppose $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ where

$Q=\{q_0,q_1,q_f\}$ $\Sigma=\{a, b\}$, $\Gamma=\{a, Z\}$,$F=\{q_f\}$

δ is given by

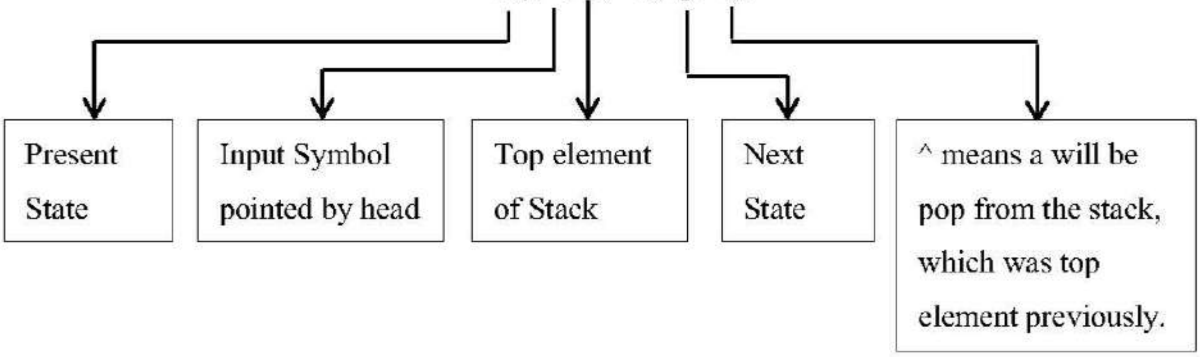
$$\delta\,(q_0, a, Z)=\{(q_0, aZ)\}$$



Present State
Input Symbol pointed by head
Top element of Stack
Next State
a will be top of stack added upon Z

At some time PDA is in some state q_0, and PDS (pushdown store) has symbol 'a' from Γ. The PDA reads an input symbol 'a' and top most symbol 'Z' in PDS using the above transition function δ. After transition, the PDA moves to transition state q_0 and insert 'a' after 'Z' in stack.

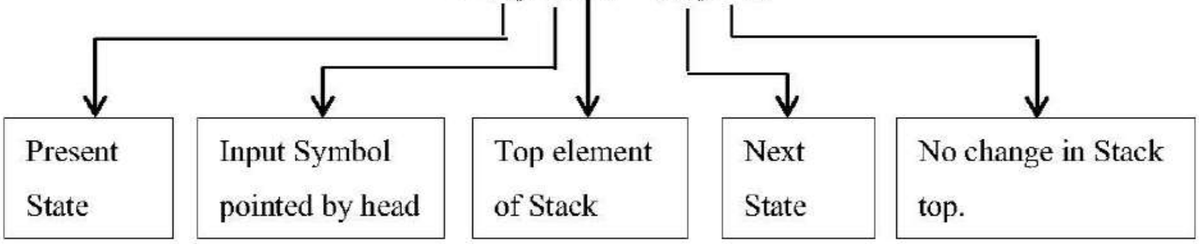
$$\delta\,(q_1, b, a)=\{(q_2, \wedge)\}$$



Present State
Input Symbol pointed by head
Top element of Stack
Next State
^ means a will be pop from the stack, which was top element previously.

Some point PDA is in some state $q_1$and PDS has symbol 'a' from Γ. The PDA reads on input symbol 'b' and read top most symbol 'a' in PDS. Using the above transition function, the PDA moves to transition state q_2 and 'a' is popped from stack.

$$\delta\,(q_0, a, Z) = \{(q_0, Z)\}$$



The above transition function does nothing. It is just read the input from the tape and does not make any change to the state and the symbol at the stack.

Instantaneous Description for PDA

Instantaneous description (ID) for PDA is a snapshot of Pushdown automaton in action and can be used to describe a Pushdown automaton.

Let P = (Q, Σ, Γ, δ, q_0, Z_0, F) be a PDA. The Instantaneous description for PDA can be written as

$$(q, x, \alpha)$$

Where,

$q \in Q$ and represent the current state of PDA.

$x \in \sum^*$ and represent input string to be processed.

$\alpha \in \Gamma^*$ and represent stack symbols.

Example:

(q, abcd…, ABCD…..)

In the above example

- q is the current state of PDA.
- abcd… is input string to be processed in the order.
- ABCD… is stack symbols, where A is at the top of stack then B and follow in that order.

In ID representation the moves is represented by '|--'as ID1 |-- ID2, means that the PDA moves from ID1 to ID2.

"ID1 |--* ID2" means ID1 is the initial Instantaneous description and after many moves ID2 is the final Instantaneous description of the PDA.

10.3 Acceptance by PDA

Since PDA has an additional structure called stack/pushdown store, the PDA accept the input string on the basics of final state and in terms of PDS.

Acceptance by Final State

Let P = (Q, Σ, Γ, δ, q_0, Z_0, F), the language accepted by P is the set

$L(P) = \{w \in \sum^* \mid (q_0, w, Z_0) \mid\text{--}^* (q_f, \wedge, u)\}$

The above statement say that the language accepted by P is the set of all the string that can put P into a final state at the end of the string. The content of stack u is irrelevant to this definition.

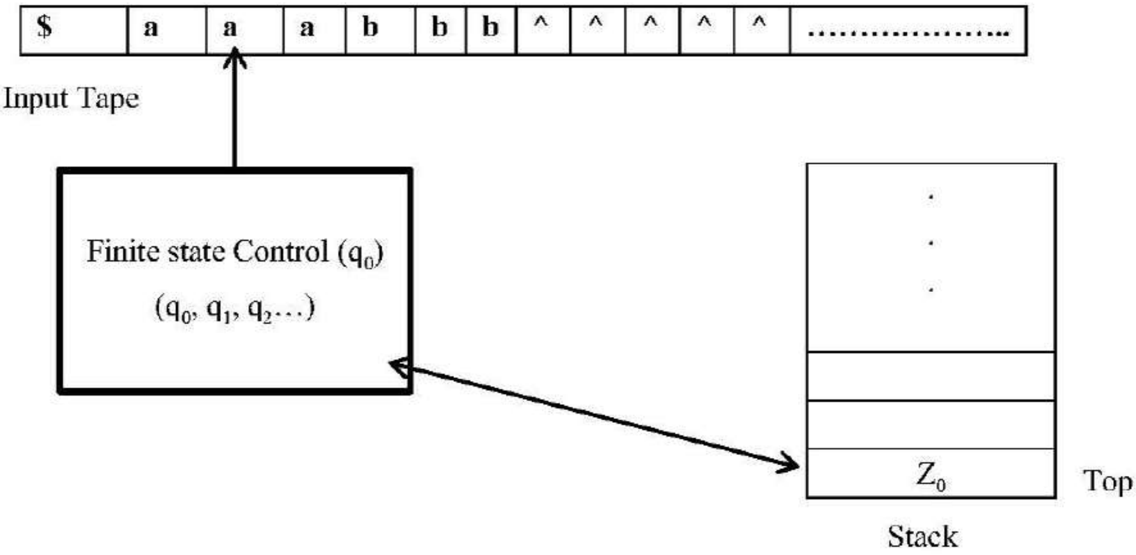
Acceptance by Empty Stack

Let P = (Q, Σ, Γ, δ, q_0, Z_0, F), the language accepted by P is the set

$L(P) = \{w \in \sum^* \mid (q_0, w, Z_0) \mid\text{--}^* (q_f, \wedge, \wedge)\}$

The above statement says that after many moves the final state is q_f and input symbol read is ^ and stack symbol is also empty i.e. ^.

Example:



$ a a a b b b ^ ^ ^ ^ ^ ………………..
Input Tape
Finite state Control (q_0)
(q_0, q_1, q_2…)
.
.
.
Z_0
Top
Stack

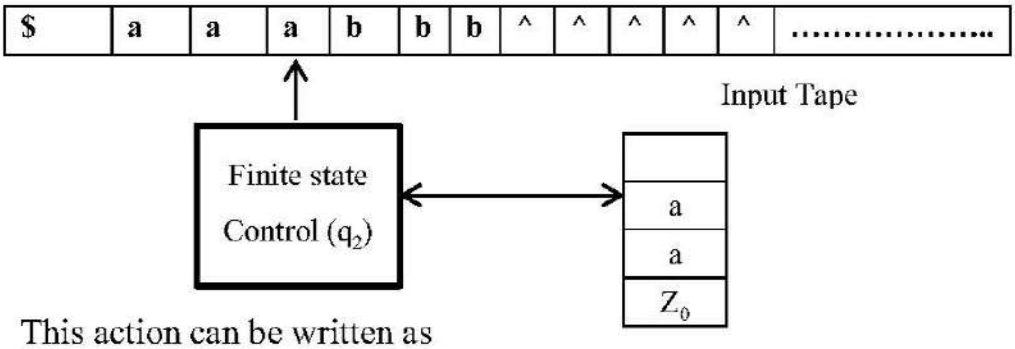
The above example shows the input tap filled with strings aaabbb or a^3b^3 the initial state is q_0 and initially Z_0 is stored at the top of push down store.

Now we shows the following input string is acceptable by PDA or not. The basics is first finite state control read the first symbol 'a' of the string aaabbb and PDA will store 'a' to the top of the stack. Suppose by doing this head moves to the right cell and the next state will be q_1.

The above action can be written as

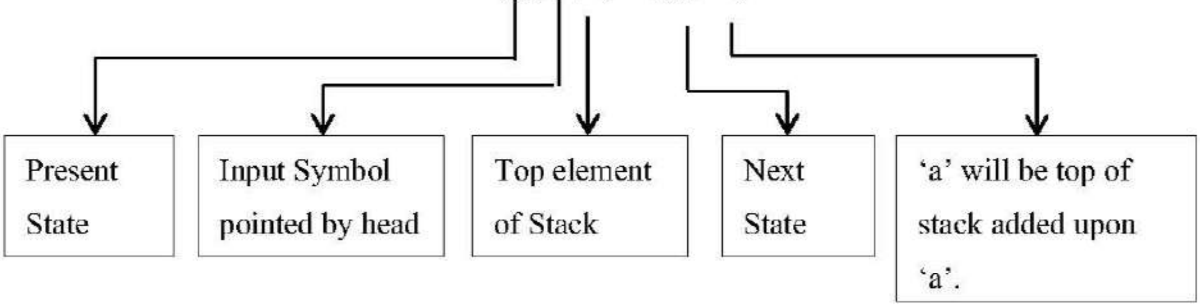
$$\delta\ (q_0,a,Z_0)= (q1,\ aZ_0)$$

Now again PDA reads the symbol 'a' pointed by head with state q_1 and stack top symbol 'a' action may be result into next state q_2 and push 'a' into stack and head moves toward next input symbol of the tape as:


$ | a | a | a | b | b | b | ^ | ^ | ^ | ^ | ^ | ....
Input Tape
Finite state Control (q_2)
a
a
Z_0

This action can be written as

$$\delta\ (q_1,\ a,\ a)= (q_1,\ aa)$$


Present State
Input Symbol pointed by head
Top element of Stack
Next State
'a' will be top of stack added upon 'a'.

Similarly for every b, a will be pop from the stack by the PDA.

The action can be written as

$$\delta\ (q_1,\ b,\ a)= (q_2,\ \wedge)$$

So the acceptability of the string can be summarize as when 'a' is found we will push into pushdown store without changing its state and when we found first 'b' then state will be change and pop the element of stack whenever another 'b' is found then pop the top element of the stack. So if there is equal number of a' s and b' s then after scanning the complete string the stack is empty and then we will change the current state to final state.

Therefore language acceptance by PDA will be determined by

- When the state is final state
- Stack is empty or special symbol (Z_0) on to stack
- String has been scanned completely.

Example: Construct a PDA which recognized L = $\{wcw^R \mid w \in \{a, b\}^*\}$.

Solution:

Let the P= (Q, Σ, Γ, δ, q0, Z0, F)

Where,

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

$F = \{q_f\}$

$\Gamma = (a, b, Z_0)$

The transition functions can be described as

Case 1: When $w = w^R = \in$

$$\delta(q_0, c, Z_0) =(q_f \ , Z0) \text{ accept}$$

Case 2: If 'a' or 'b' comes with Z_0 on top then push 'a' or 'b' on to push down store.

$$\delta\ (q_0, a, Z_0) =(q_0, aZ_0)$$

$$\delta\ (q_0, b, Z_0\ ) =(q_0, bZ_0)$$

Case 3: When 'c' is input symbol then PDA do nothing onto stack but change the state from q_0 to $q_{1.}$

$$\delta\ (q_0, c, a) = (q_1, a)$$

$$\delta\ (q_0, c, b) = (q_1, b)$$

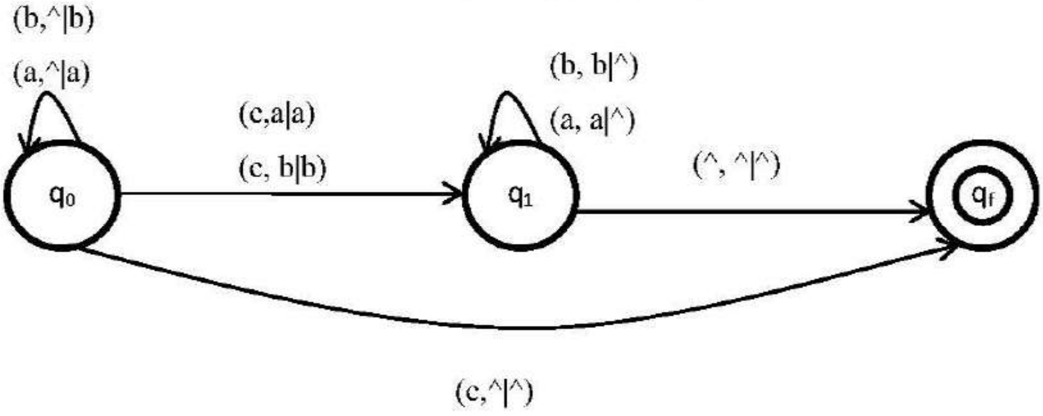
Case 4: If input is 'a' and top of stack is 'a' then pop 'a' or if input is 'b' and top is also 'b' then pop 'b'.

$$\delta\ (q_1, a, a) = (q_1, \wedge)$$

$$\delta\ (q_1, b, b) = (q_1, \wedge)$$

Case 5: When all input are read and stack is also empty then moves to the final state.

$$\delta\ (q_1, \wedge, Z_0) = (q_f, Z_0)$$



Example: Construct a PDA to recognized L ={$a^n b^n$ | n >= 0}

Solution: Let the PDA P= (Q, Σ, Γ, δ, q0, Z0, F) where Q= {q_0, q_1, q_f} and F={q_f}

Transition function can be described as δ

$\delta\ (q_0, \wedge, \wedge) = (q_f, \wedge)$ For empty string

$\delta(q_0, a, \wedge) = (q_0, A)$

$\delta(q_0, b, A) = (q_1, \wedge)$

$\delta(q_1, b, A) = (q_1, \wedge)$

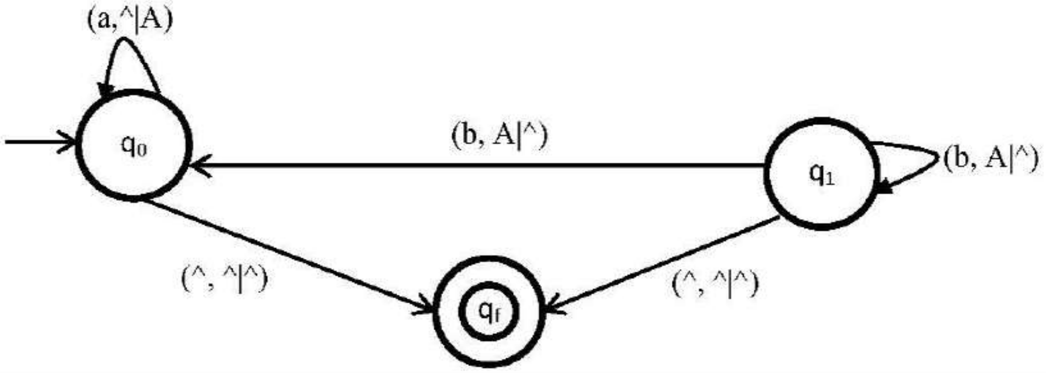
$\delta(q_1, A, \wedge) = (q_2, \wedge)$; accept |--

The moves can be described as

(q_0, aabb, Z_0) |-- (q_0, abb, AZ_0) |-- (q_0, bb, AAZ_0) |-- (q_1, b, AZ_0) |-- (q_f, ^, Z_0)

The PDA halts and accepted.

The Transition Diagram of above example is shown as:



10.4 Pushdown Automata and CFG

As we see in the previous units, a context free grammar provides a simple and mathematically precise mechanism for describing the methods by which phrases in some natural language are built from smaller blocks capturing the block structure of sentences in a natural way. This section will be focused on the conversion from PDA to CFG and vice versa with their relationship with CFG.

Conversion from PDA to CFG and vice versa:

First we show that for every context free programmer there is a non-deterministic PDA which accepts the language that are provided by context free grammar

Algorithm to find PDA corresponding to given CFG:

Input: A CFG G=(V,T,P,S)

Output: Equivalent PDA, P= P= ((Q, Σ, Γ, δ, q0, Z0, F)

Step-1: Convert the production of CFG into GNF

Step-2: The PDA will have only one state {q} [optional]

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

Step-4: All non-terminals of the CFG will be the stack symbol of the PDA and all the Terminals of the CFG will be the input symbol of PDA

For each production in the form A→aX where a is terminal and A, X are combination of terminal or non-terminals make a transition δ (q,a, A).

Example: Construct an NPDA that accepts the language grammar by programmer with production: S->aSbb | a

Solution: First we transform grammar in GNF (Grebian Normal Form) as

$$S \rightarrow aSA \mid a;$$
$$A \rightarrow bB$$
$$B \rightarrow b$$

Corresponding PDA will have three state {q0, q and q2} will q0 is starting and final.

First we start symbol S is put on the stack by the following transition function:

$$\delta\,(q_0, \hat{}, Z) \rightarrow (q_1, SZ)$$

(i) For production S → aSA transition function is $\delta\,(q_1, a,s) \rightarrow (q_1,SA)$.

(ii) For production S → a transition function is $\delta\,(q_1, a,s) \rightarrow (q_1, \hat{})$.

(iii) For production A → bB transition function is $\delta\,(q_1, b, A) \rightarrow (q_1, B)$.

(iv) For production B → b transition function is $\delta\,(q_1, b, B) \rightarrow (q_1, \hat{})$.

The appearance of the start symbol on top of the stack signals the completion of the derivation and the PDA is put into its final state by

$$\delta\,(q_1, \hat{}, Z) \rightarrow (q_2, \hat{})$$

Algorithm to find CFG corresponding to a given PDA:

Input − A CFG, G= (V, T, P, S)

Output − Equivalent PDA, P = (Q, ∑, S, δ, q0, I, F) such that the non-terminals of the grammar G will be {Xwx | w,x ∈ Q} and the start state will be Aq0,F.

Step 1 For every w, x, y, z ∈ Q, m ∈ S and a, b ∈ ∑, if δ (w, a, ε) contains (y, m) and (z, b, m) contains (x, ε), add the production rule Xwx → a Xyzb in grammar G.

Step 2 For every w, x, y, z ∈ Q, add the production rule Xwx ⟶ XwyXyx in grammar G.

Step 3 For w ∈ Q, add the production rule Xww ⟶ ε in grammar G.

10.5 Pushdown Automata & Parsing

Parsing is used to derive a string using the production rules of a grammar. It is used to check the acceptability of a string. Compiler is used to check whether or not a string is syntactically correct. A parser takes the inputs and builds a parse tree.

A parser can be of two types −

- **Top-Down Parser** − Top-down parsing starts from the top with the start-symbol and derives a string using a parse tree.
- **Bottom-Up Parser** − Bottom-up parsing starts from the bottom with the string and comes to the start symbol using a parse tree.

Design of Top-Down Parser

For top-down parsing, a PDA has the following four types of transitions:

- Pop the non-terminal on the left hand side of the production at the top of the stack and push its right-hand side string.
- If the top symbol of the stack matches with the input symbol being read, pop it.
- Push the start symbol 'S' into the stack.
- If the input string is fully read and the stack is empty, go to the final state 'F'.

Example: Design a top-down parser for the expression "x+y*z" for the grammar G with the following production rules −

P: S ⟶ S+X | X, X ⟶ X*Y | Y, Y ⟶ (S) | id

Solution: If the PDA is (Q, ∑, S, δ, q0, I, F), then the top-down parsing is −

(x+y*z, I) ⊢ (x +y*z, SI) ⊢ (x+y*z, S+XI) ⊢ (x+y*z, X+XI)

⊢(x+y*z, Y+X I) ⊢ (x+y*z, x+XI) ⊢ (+y*z, +XI) ⊢ (y*z, XI)

⊢ (y*z, X*YI) ⊢ (y*z, y*YI) ⊢ (*z,*YI) ⊢ (z, YI) ⊢ (z, zI) ⊢ (ε, I)

Design of a Bottom-Up Parser

- For bottom-up parsing, a PDA has the following four types of transitions −
- Push the current input symbol into the stack.
- Replace the right-hand side of a production at the top of the stack with its left-hand side.
- If the top of the stack element matches with the current input symbol, pop it.
- If the input string is fully read and only if the start symbol 'S' remains in the stack, pop it and go to the final state 'F'.

Example: Design a top-down parser for the expression "x+y*z" for the grammar G with the following production rules −

P: S ⟶ S+X | X, X ⟶ X*Y | Y, Y ⟶ (S) | id

Solution: If the PDA is (Q, ∑, S, δ, q0, I, F), then the bottom-up parsing is −

(x+y*z, I) ⊢ (+y*z, xI) ⊢ (+y*z, YI) ⊢ (+y*z, XI) ⊢ (+y*z, SI)

⊢ (y*z, +SI) ⊢ (*z, y+SI) ⊢ (*z, Y+SI) ⊢ (*z, X+SI) ⊢ (z, *X+SI)

⊢(ε, z*X+SI) ⊢ (ε, Y*X+SI) ⊢ (ε, X+SI) ⊢ (ε, SI)

10.6 Self Learning Exercise

Q.1 PDA is the machine format of

a) Type-0 Language

b) Type-1 Language

c) Type-2 Language

d) Type-3 Language

Q.2 Which is not true for model of PDA?

a) PDA contains a stack.

b) The head reads as well as writes.

c) The head moves from left to right.

d) Input string is surrounded by infinite number of blank in both side.

Q.3 The difference between finite automata and PDA is in .

a) Reading Head.

b) Input tape.

c) Finite Control.

d) Stack.

Q.4 Which of the following is not true?

a) Power of deterministic automata is equivalent to power of nondeterministic automata.

b) Power of deterministic pushdown automata is equivalent to power of non-deterministic pushdown automata.

c) Power of deterministic Turing machine is equivalent to power of non-deterministic Turing machine.

d) All the above.

10.7 Summary

This unit can be summarized as:

- Pushdown automata uses stack as extra memory for keeping information of past scanned symbols.
- Accepting a string in Pushdown Automata will possible when string is completely scanned, stack is empty or machine is in final state.
- Designing of pushdown automata will be possible by empty store, final state and sometimes by both.
- There is equivalent context free grammar for push down automata and vice versa.
- When we change context free grammar to pushdown automata, the grammar should be in GNF.
- PDA can also be used in parsing a string.

- The Parser can be of two types: Top-down parser and Bottom-Up parser.
- Both the parsing can be performed using PDA.

10.8 Answers to Self-Learning Exercise

Q.1 (c)

Q.2 (b)

Q.3 (d)

Q.4 (b)

10.9 Exercise

Q.1 Construct a PDA (Pushdown Automata) accepting by empty store each of the languages-

(i) $\{a^{2n}b^{n} \mid n>=1\}$

(ii) $\{a^{n}b^{2n} \mid n>=1\}$

(iii) $\{a^{m}b^{m}c^{n} \mid m>n>=1\}$

(iv) $\{a^{m}b^{m} \mid m>n>1\}$

Q.2 Construct a PDA accepting by final state each of the language given in Q.1.

Q.3 Construct a PDA accepting the set of all strings over (a,b). Consisting of equal number of a's and b's.

Q.4 Construct a PDA accepting by final state for the language containing even number of a's over $\sum = \{a, b\}$. Convert it into context free framer (CFG) also.

Q.5 Construct a PDA accepting the set of all even length palindrome over {a, b} by empty store.

Q.6 Design a PDA which converts infix to prefix

Q.7 Construct the equivalent PDA for the following CFGs.

(i) S →Saa | aSa | aaS

(ii) S → (S) (S) | a

(iii) $S \rightarrow XaY \mid YbX$

$X \rightarrow YY \mid aY \mid b$

$Y \rightarrow b \mid bb$

References and Suggested Readings

1. Peter Linz, An Introduction to Formal Languages and Automata, Jones and Bartlett Publication, Third Edition.
2. K.L.P. Mishra, N. Chandrasekaran, Theory of Computer Science, BPB Publication, Prentice-Hall of India, Second Edition.
3. H.E. Hopcraft and J.D. Ullamn, Introduction to Automata Theory, Languages and Computation, Narosa Publications.
4. J.C. Martin, Introduction to Languages and the Theory of Automata, Tata McGraw-Hill.

UNIT-11
Turing Machine

Structure of the Unit

11.0 Objective

After reading this chapter you will be able to understand the most powerful abstract computing device, the Turing machine and the following:

- Understand the standard Turing machine and its definition
- Understand the computing problem solved by Turing machine.
- Turing Thesis

11.1 Introduction

In the later chapters, we have seen several abstract models of computing devices such as Finite Automata and Pushdown Automata, the concept of regular language and context-free language and their association with the Finite automata and Pushdown automata. In the previous chapters it is noted that the pushdown automata is more powerful than the Finite automata. However, none of the above "seem to be" as powerful as the real computer. This diverts our attention to the more powerful abstract model of computing device called Turing Machine.

Turing Machine was invented in 1936 by Alan Turing. Turing Machine is considered to be extremely simple calculating device but very powerful device. It is stated by Church – Turing thesis that any algorithmic procedure that can be carried out by human/computer can be carried out by a Turing Machine. So, Turing machine provides an ideal theoretical model of a computer.

11.2 Turing Machine Model

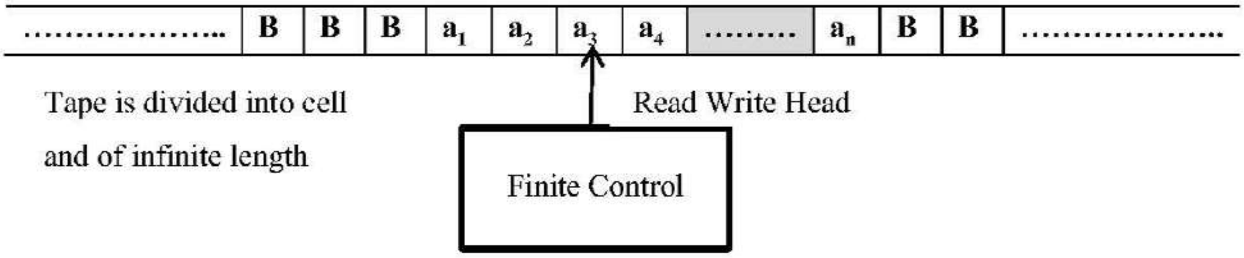


Figure11.1: Turing Machine Model

A Turing machine is a mathematical model which consists of:

Tape: It is divided into cell; each cell can hold one of a finite number of symbols. Initially the input (a finite-length string) is placed on the tape. All other tape cell holds a special symbol called Blank, denoted by B.

It is noted that the Blank (B) is a tape symbol, not the input symbol.

Tape head: It is associated with the tape, can read-write and move in both left or right on the tape.

Finite Control: A move of the TM is a function of the state of the finite control and the tape symbol scanned. In one move the TM can:

1. Change state.
2. Read/Write the tape symbol in the cell scanned

3. Move the tape left or right.

Definition: A Turing Machine M is a 7 tuple:

$M = (Q, \Sigma, \Gamma, \delta, qo, B, F)$ where

Q is the set of internal states

Σ is the input alphabet/symbol

Γ is a finite set of symbols called tape alphabet/symbol

δ is the transition function and given by

qo $\in$ Q is the initial state

B $\in \Gamma$ is a special symbol, called Blank.

F is the set of final states.

The transition function δ is defines as

δ: Q x Γ -> Q x Γ x {L,R}

In general δ is a partial function in Q x δ; its interpretation gives the principle by which a Turing Machine operates. The arguments of δ function are the current input symbol and current state and the result of the δ function is the next state, new symbol on the tape and R/W head movement (L or R).

This can be better understood by an example:

Suppose the transition function is given by δ (q5, b) = (q8, c, R). This means the current symbol is q5 and the current symbol under R/W head is b. Now after performing the action of the above transition function the state changes to q8, the input symbol is replaced by c and the R/W head is moves towards right.

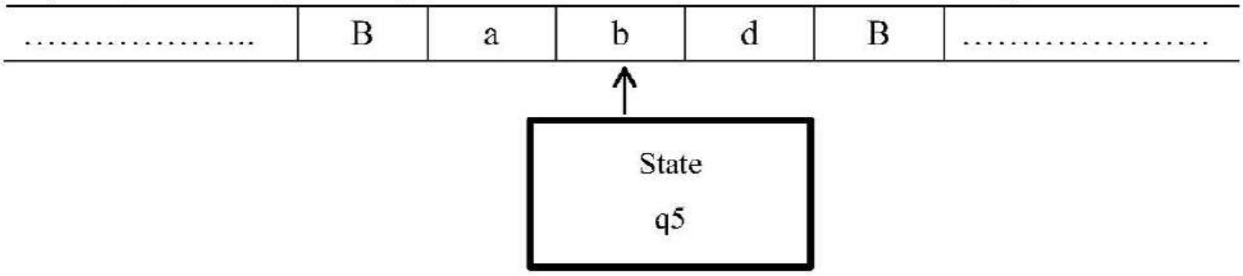


Figure11.2: Current Status of TM.

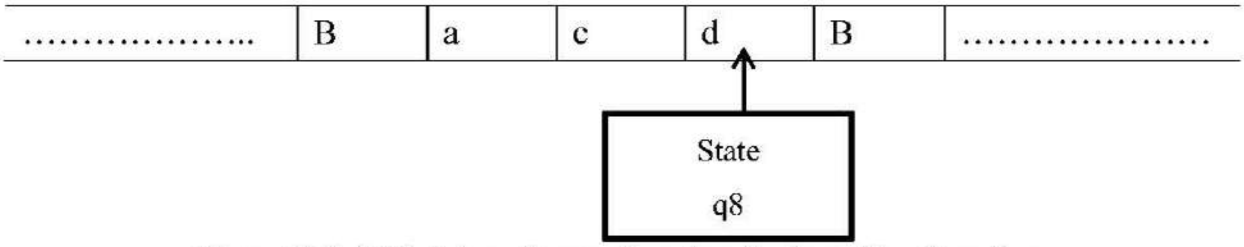


Figure11.3: TM status after performing the transition function.

11.3 Representation of Turing Machine

The following method is used to describe the Turing Machine:

i. Instantaneous Description using move relation

ii. Transition Table and

iii. Transition diagram

Instantaneous Description for TM

Instantaneous description (ID) of TM is a snapshot of a Turing Machine in action and can be used to describe a Turing Machine.

An ID of a Turing Machine M is a string $\alpha\beta\gamma$ where:

β is the present state of the Turing Machine M.

γ = suppose the first symbol of γ substring is 'a' under R/W head. And γ has all the subsequent symbols of the input string

α is the substring of the input string formed by all the symbols to the left of a.

Example: A snapshot of Turing machine is shown in fig 11.4. Suppose we want to obtain Instantaneous Diagram.

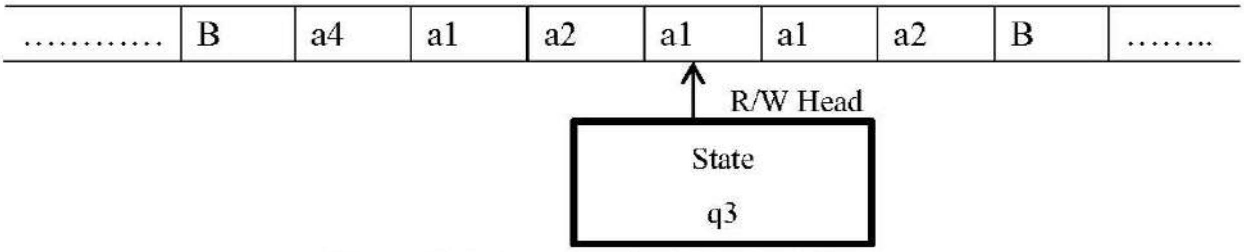


Figure11.4: An example of Turing Machine

The above snapshot of TM shows that the current symbol under R/W head is a1 and the present state of TM is q3. The non-blank substring on the left of a1 (symbol under R/W head) is a4, a1, a2. The non-blank sequence to the right of a1 (symbol under R/W head) is a1, a2. Thus the ID of the given example can be represented as

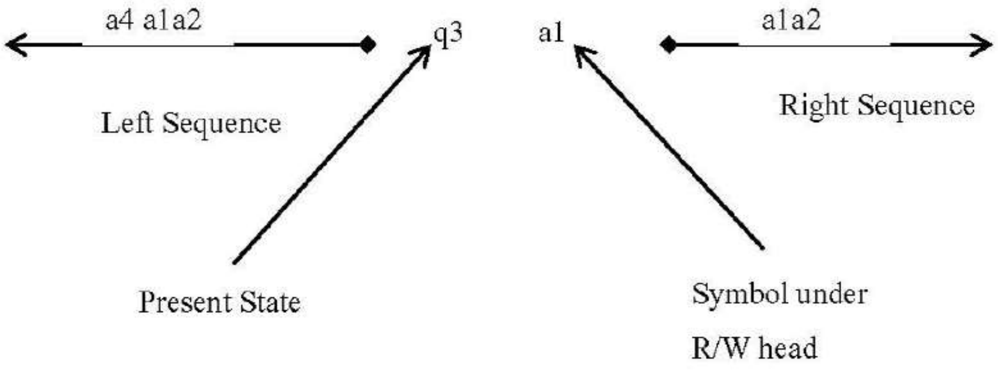


Figure11.5: Description of ID.

So, the ID for the above example is a4a1a2q3a1a1a2.

Moves in a TM

Suppose Turing Machine M = (Q, ∑, Γ, δ, qo, B, F)

We use the notation |-- to represent moves of a TM M from one configuration to another. |-- is used as usual.

For example:

If δ (q_5, b) = (q_8, c, R) then a possible move might be

abbabq5baab |-- abbabcq8aab

For another transition function δ (q_8, c) = (q_9, a, L) the move might be

abbabc q_8 aab |-- abbab q_9 aaab

ID of the above transition functions can be written as

abbab q_5 baab |-- abbabc q_8 aab |-- abbab q_9 aaab

Transition Table for TM

We can also define transition function in the form of table, called Transition Table. If δ (q, a) = (γ, α, β) then in transition table αβγ is in α-column and in the q-row. So if αβγ is in the column then it means α is written in the current cell, β give the movement of head (L or R) and γ denote the new state into which TM enters.

Example:

Suppose a TM has five states q_1, q_2, q_3, q_4, and q_5 where q_1 is the initial state and q_5 is the (only) final state. The tape symbols are 0, 1, and B. The transition table equivalent to transition function can be described as the table:

Table 11.1: Transition Table

Present State	Tape Symbol		
	B	**0**	1
$\rightarrow$ **q_1**	$1Lq_2$	$0Rq_1$	
q_2	BRq_3	$0Lq_2$	**$1Lq_2$**
q_3		BRq_4	**BRq_5**
q_4	$0Rq_5$	$0Rq_4$	**$1Rq_4$**
$\textcircled{q_5}$	**$0Lq_2$**		

The initial state in the transition table is marked with → and the final state with O.

Transition diagram for Turing Machine

Another way to represent Turing Machine is with the help of Transition Diagram. In the Transition Diagram, the states are represented by vertices and directed edges are used to represent transition of states. The labels are triples of the form (α, β, γ) where α, β ϵ Γ and γ ϵ {L, R}.

When the directed edge from qi to qj with label (α, β, γ), it mean that

$$\delta(q_i, \alpha)=(q_j,\beta,\gamma)$$

Therefore,

α = present symbol under R/W head.

β = symbol write in the cell under R/W head.

γ = R/W head movement (either Left or Right).

The initial state in the transition diagram is marked with → and the final state with ◎ .

Example: A Turing Machine M is defined as M= ({q1, q2}, {1, B}, {1, B}, δ, q1, B, {q1}) where the δ (transition function) is define as

δ (q1, 1)=(q2,B,R)

δ (q2, 1)=(q1, B, R)

For the above example the transition diagram can be represented as

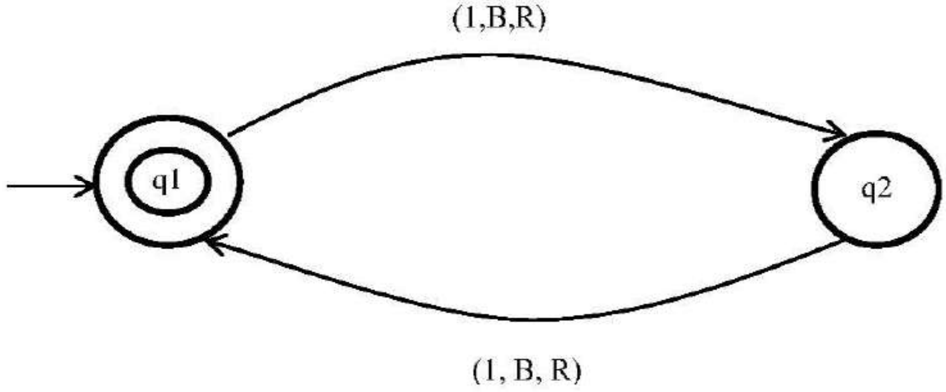


Figure11.6: Turing Machine of Example .

The above Turing Machine is recognizing all the strings consisting of an even number of 1's.

11.4 Language acceptability of Turing Machine

A Turing Machine can be viewed as Language accepters. A string w is written on the tape, with blanks filling out the unused portion. The Machine is started at initial state q0 with the read-write head positioned on the leftmost symbol of w. A string w is treated to be accepted by Turing Machine if after the sequence of moves, TM enters the final state and halts.

Definition:

Let Turing Machine M = (Q, ∑, Γ, δ, qo, B, F), then the language accepted by M is

L (M) = {w ϵ ∑+: q0w |--* x1q_fx2 for some q_f ϵ F, x1, x2 ϵ Γ *}

As we saw earlier the tape in the Turing Machine is infinite in both directions (Left and Right). The Blank symbol (B) written on the either side of the w assure that the string w is restricted to well defined region on the tape. That is why Blank Symbol (B) is excluded from the set of input alphabets (∑). Without this assumption machine could not limit the region in which it must look for the input string.

The set of languages that a Turing Machine can accept is often called as **recursively enumerable language** or **RE language**. A language is Turing-recognizable if some Turing Machine recognizes it.

Example: Design a TM that accept L = {0^n1^n : n>= 1}

Solution:

We require the following moves:

(a) If the leftmost symbol in the given input string w is 0, replace it by x and move right till we encounter a leftmost 1 in w. Change it to y and move backwards.

(b) Repeat (a) with leftmost 0. If we move back and forth and no 0 or 1 remains, move to the final state.

(c) For the strings not in the form 0^n1^n, the resulting state has to be nonfinal.

Now, we construct a TM M as follows:

M = (Q, ∑, Γ, δ, qo, B, F)

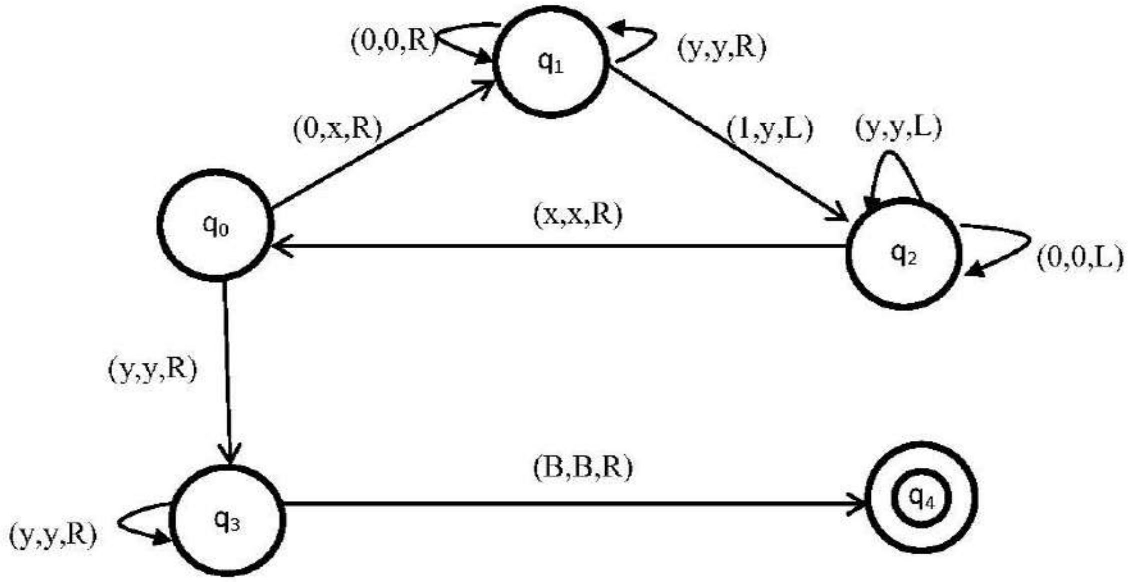
Where,

Q = (q_0,q_1,q_2,q_3,q_f)

∑ = {0,1}

Γ = {0,1,x,y,B}

The transition diagram of M that accept L = {0^n1^n: n>= 1} is shown in the figure.



The move for input string 0011 and 010 are given as

$q_0$001 |-- x$q_1$011 |-- x0$q_1$11 |-- x$q_2$0y1 |-- q_2x0y1 |-- x$q_0$0y1 |--xxq_1y1 |--xxy$q_1$1 |-- xx$q_2$yy |--x$q_2$xyy |--xx$q_0$yy |--xxyq_3y |--xxyyq_3 |-- xxyyq_3B |-- xxyyBq_4B

Hence 0011 is accepted by M.

$q_0$010 |-- x$q_1$10 |--$q_2$xy0 |--xq_0y0 |-- xy$q_3$0

As (q_3, 0) is not defined, M halts. So the input string 010 is not accepted by M.

11.5 Design of Turing Machine

There are some basic guidelines for designing a Turing Machine:

a) The fundamental objective in scanning a symbol by R/W head is to know what to do in the future. The machine must remember the past symbols scanned. The Turing machine can remember this by going to the next unique state.

b) The number of states must be minimized. This can be achieved by changing the state only when there is a change in the written symbol or when there is a change in the movement of the R/W head.

The above guideline can be better understood by taking some of the examples.

Example: Design a Turing Machine over {1,B} which can compute a concatenation function over $\sum$ = {1}. If a pair of words {w1, w2} is the input then the output has to be w1w2.

Solution:

Initially w1 and w2 are written on the tape separated by B. Suppose w1 = 1 and w2 = 1 then the tape looks like

…………	B	B	B	1	B	1	B	B	B	B	………………..

After processing, the output will be look like

…………	B	B	B	1	1	B	B	B	B	B	………………..

We can follow the following steps to design the Turing machine:

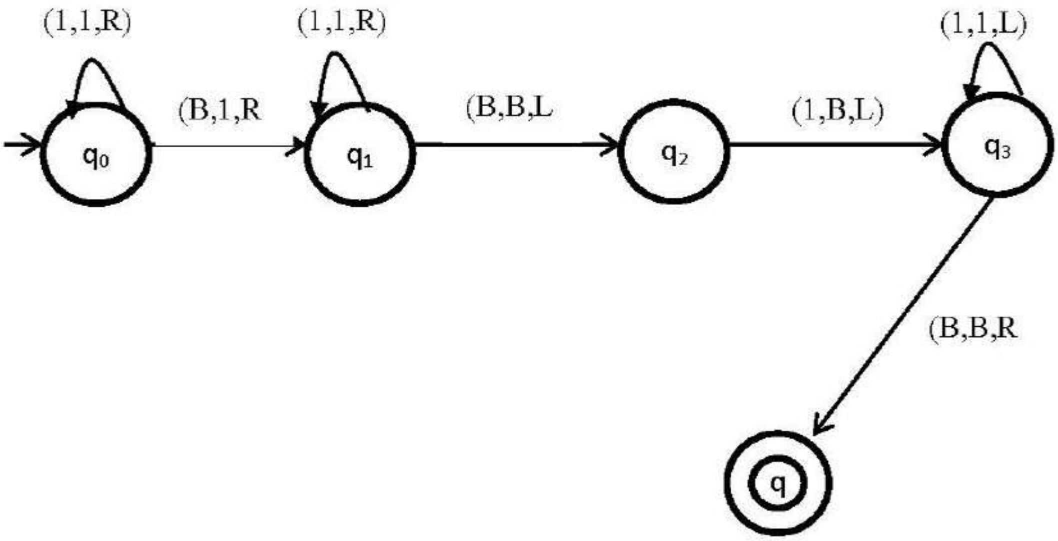
1. Frist we find the separate symbol B and replace B by 1.
2. Find the rightmost 1 and replace 1 by B.
3. Finally, move the R/W head to the initial position.

The moves can be describe as

$q_0$1B1 |-- 1q_0B1 |-- 11$q_1$1 |--111q_1B |--11q_2B |-- 1$q_3$1BB |--$q_3$11BB |-- q_3B11BB |-- Bq_f11BB

Now we draw the generalize transition diagram of Turing Machine that accept all types of w1 and w2 defined as per given language.

Transition Diagram



Transition Table

Present State	Tape Symbol	
	B	1
→ **q_0**	1Rq_1	**1Rq_0**
q_1	BLq_2	**BLq_2**
q_2	--	--
q_3	BRq_f	**1Lq_3**
q_f	--	--

Example: Design a TM that copies strings of 1's

Solution:

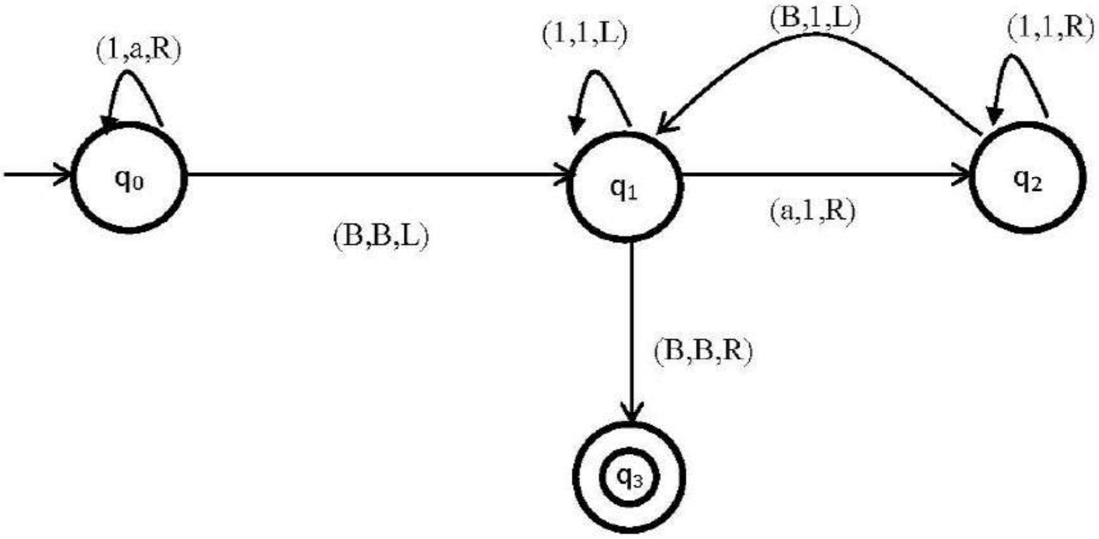
We can follow the following steps to design the Turing Machine:

1. M replaces every 1 by the symbol a.
2. M replaces the rightmost a by 1.
3. M goes to right end of the string and write character a and 1.
4. Thus for the rightmost 1, M added character a and 1.
5. This process can be repeated.

Now, we design the TM M such that tape has ww after coping w ∈ {1}*

M = ($\{q_0, q_1, q_2, q_3\}$, $\{1\}$, $\{1, a, B\}$, δ, q_o, B, $\{q_3\}$)

Transition Diagram:



Transition Table:

Present State	Tape Symbol		
	B	**1**	a
→ q_0	BLq_1	aRq_0	--
q_1	BRq_3	$1Lq_1$	$\mathbf{1Rq_2}$
q_2	$1Lq_1$	$1Rq_2$	--
q_3	--	--	--

A sample computation is given below:

$q_0$11 |-- a$q_0$1 |-- aaq_0B |--aq_1a |-- a1q_2B |--a$q_1$11 |--q_1a11 |--1$q_2$qq |--11$q_2$1 |-- 111q_2B |--11$q_2$11 |-- 1$q_1$111 |--$q_1$1111 |--q_1B1111 |--$q_3$1111

11.6 Techniques of Turing Machine

Writing the Turing Machine for complicated languages can be a difficult and boring. But one can use some techniques. The goal of this section is to convince any language that a computer program can recognize will also be recognized by a Turing Machine.

- **Storing a tape symbol in the finite control**

As we know states are being used for remember the symbol. We can build a TM whose states are pairs [q, a] where q is the state and 'a' is a tape symbol stored in [q, a]. So the new set of states becomes $Q \times \Gamma$.

This can be better understood by taking an example. Consider a TM that recognizes the language

$$L = ab^* + ba^*$$

In this example TM first remembers the first symbol and check that the same symbol does not appear anywhere i.e. if the symbol is 'a' then TM stores symbol 'a' and check that whether 'a' is not repeat anywhere in the tape. The same case if the first symbol is 'b'. We can represent as:

$\delta(q_0, a) = ([q, a], a, R)$

$\delta(q_0, a) = ([q, b], b, R)$

$\delta([q, a], b) = ([q, a], b, R)$

$\delta([q, b], a) = ([q, b], a, R)$

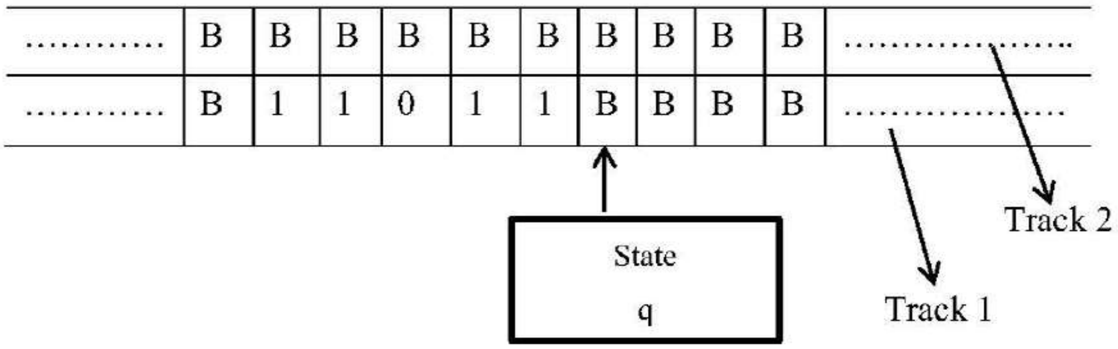
$\delta([q, a], B) = (q_f, B, R)$

$\delta([q, b], B) = (q_f, B, R)$

- **Multiple tracks**

Earlier we saw TM with a single tape. In Multiple tracks TM the tape is assumed to be divided into various different tracks. Now the tape alphabet is required to consist of k-tuple of tape symbol where k is the number of track. The only difference between the standard Turing Machine and Multiple tracks TM is the set of tape symbols. In the case of the standard Turing Machine tape symbols are element of Γ and in case of multiple tracks Turing Machine it is Γ^K. The moves are defined in a similar way.

Example: A TM with 2-tracks is shown in figure



………… B B B B B B B B B B ………………..
………… B 1 1 0 1 1 B B B B ……………….…
Track 2
State
q
Track 1

Tape position in two-track is represented by [x, y], where x is symbol in track 1 and y is in tack-2. The states, Σ, Γ, q0, F of a two-track machine are same as for standard machine.

A transition of a two-track machine reads and writes the entire position on all tracks.

δ is: δ(qi, [x, y]) = [qj, [z, w], d], where d$\in$ {L,R}. The input for two-track is put at track-1, and all positions on track-2 is initially blank. The acceptance in multi-track is by final state.

- **Subroutine**

If we want a repetitive work in any computer language then subroutine is used. We can implement this facility for TMs as well.

First, a TM program for the subroutine is written. This will have an initial state and a return state. After reaching the return state, there is a temporary halt. For using a subroutine, new state are introduced. When there is a need for calling the subroutine, moves are effected to enter the initial state for the subroutine (when the return state of the subroutine is reached) and to return to the main program of TM.

11.7 Self Learning Exercise

Q.1 Which of the following can be used to simulate any turing machine?

a) Finite State Automaton

b) Universal Turing Machine

c) Counter machines

d) None of the above

Q.2 Which of the following are correct statements?

a) TMs that always halt are known as Decidable problems.

b) TMs that are guaranteed to halt only on acceptance are recursive ennumerable.

c) Both (a) and (b).

d) None of the above.

Q.3 L = $\{0^n1^n : n>= 1\}$ is accepted by?

a) Finite automata.

b) Moore Machine.

c) Turing Machine.

d) None of the above.

11.8 Summary

In this chapter you learned about Turning machine and its variants. As the turning machine is the most powerful machine in the universe. We also look the different way to represent the TM Authentically by transition diagrams, transition table and by ID s.

The set of language that can be accepted by the TM is called as recurring enumerable language or RE language by accepting the language grammar; the turning machine has a powerful enough to make the subroutine and further make some bigger program. This makes the turning-church thesis corrects which suggest that nothing is powerful enough in comparison to turning machine. The legacy of

TM is known from the fact that any algorithm is exists only when its corresponding TM would be made.

11.9 Answers to Self-Learning Exercise

Q.1 (b)

Q.2 (c)

Q.3 (d)

11.10Exercise

Q.1 Construct a TM with one Tape, that accept the language L={$0^{2n}1^{n}$ | n>=0}. Assume that at the start of computation the tape head is on the leftmost symbol of the input tape.

Q.2 Construct a TM that copies string of 1's on the input tape.

Q.3 Prove that the following function are computable functions:

a) f(x) = 3x

b) f(a,b)=2a+3b

c) f(a)=a mod 5

d) f(a,b) = a-b if a<b

f(a,b)=0 if a<=b

Q.4 Design a TM to recognize the following Language:-

a) L={$0^{n}1^{n}0^{n}$ | n>=1}

b) L={WW^{R}|Wis in (0+1)*}

c) The set of string with an equal number of 0's and 1's

References and Suggested Readings

1. Peter Linz, An Introduction to Formal Languages and Automata, Jones and Bartlett Publication, Third Edition.
2. K.L.P. Mishra, N. Chandrasekaran, Theory of Computer Science, BPB Publication, Prentice-Hall of India, Second Edition.

3. H.E. Hopcraft and J.D. Ullamn, Introduction to Automata Theory, Languages and Computation, Narosa Publications.
4. J.C. Martin, Introduction to Languages and the Theory of Automata, Tata McGraw-Hill.

UNIT-12

Decidability and Recursively Enumerable

Structure of the Unit

12.0 Objective

In this chapter we shall focus upon the following topics

- Algorithm, Decidability and Recursively Enumerable Languages.

12.1 Definition of an Algorithm

An algorithm is a well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output.

An algorithm is thus a sequence of computational steps that transform the input into the output.

A solution for a given problem may be in the form of an algorithm or a program. An algorithm is a step by step procedure for solving the given problem. An algorithm is independent of any programming language and machine. An algorithm is also called a *Pseudo Code*. Pseudo means false. An algorithm is called a false code as it is not tagged to any specific language syntax. A program is a language specific implementation of the algorithm. A program is synonymous with *Code*.

Properties of an Algorithm

An algorithm must have following five properties.

- Finiteness
- Definiteness
- Input
- Output
- Effectiveness

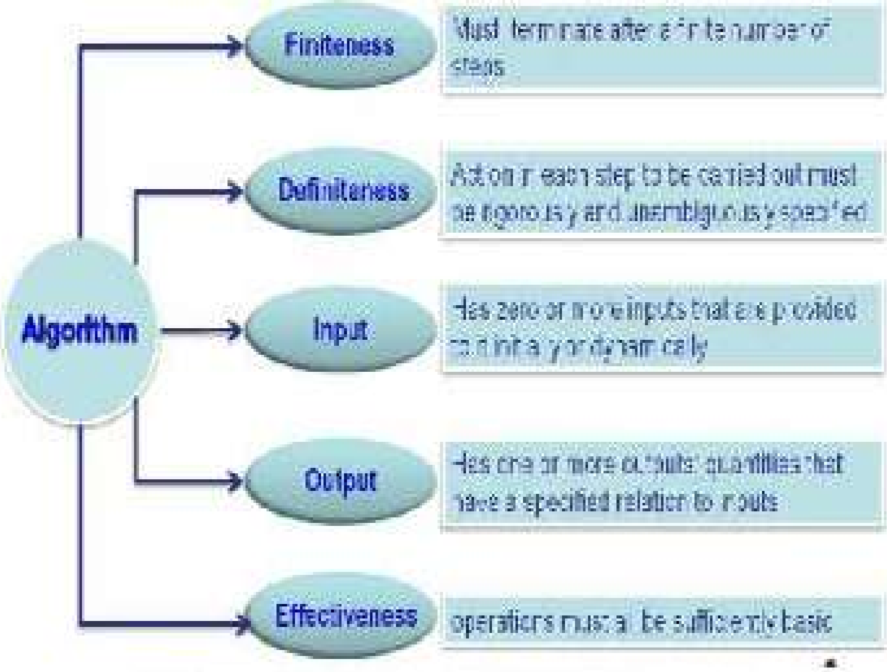

Algorithm
Finiteness
Must terminate after a finite number of steps
Definiteness
Action at each step to be carried out must be rigorously and unambiguously specified
Input
Has zero or more inputs that are provided to it initially or dynamically
Output
Has one or more outputs: quantities that have a specified relation to inputs
Effectiveness
operations must all be sufficiently basic

Figure: 12.1. Properties of an Algorithm

Steps to develop an Algorithm

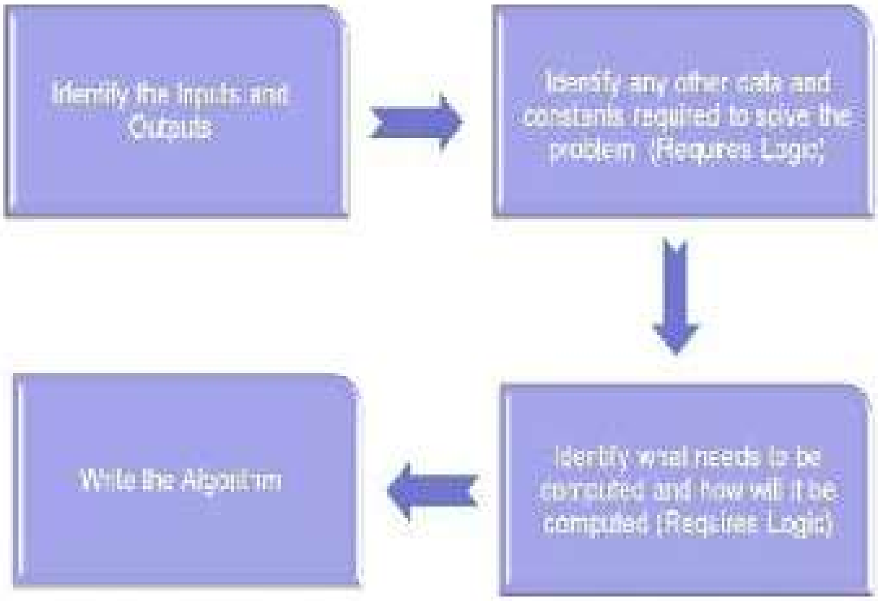

Identify the Inputs and Outputs
Identify any other data and constants required to solve the problem (Requires Logic)
Identify what needs to be computed and how will it be computed (Requires Logic)
Write the Algorithm

Figure: 12.2. Steps to develop an Algorithm

12.2 Decidability

The term decidable connected with the decision problem, the question of the existence of an efficient method for determining membership in a set of formulas, or, more specifically, an algorithm that return a Boolean value either true or false.

Assume that a language be any set of strings (or words) over a given finite set of alphabet. The alphabet could consist of the symbols we normally use for communication, such as the ASCII characters on a keyboard, including spaces and punctuation marks. In this way any story can be regarded as a "word".

A language is called decidable if there exist any method to determine whether a given word belongs to that language or not.

Working hypothesis I: Decidability is well defined above.

The above mentioned hypothesis is certainly reasonable. On the other hand, taking into consideration that the concept of definability is not well defined, one might wonder whether the situation for decidability is different. Church' thesis says that it is, and in a very profound way. Here it is concluded without invoking Church' thesis, so we stick with the working hypothesis.

An algorithm or recipe for recognizing languages is anything that can be fed a word over the given alphabet and that depending on its input either "accepts" the input after some time, or "rejects" it, or runs forever without accepting or rejecting its input. An algorithm is halting, or guaranteed to halt, if the third possibility does not occur, i.e. if it accepts or rejects within a finite amount of time. Any method that qualifies for the definition of decidability above counts as a halting algorithm for recognizing languages.

12.3 Decidable Languages

Proposition: The decidable languages are closed under union and intersection.

Proof: Let L and M be languages that are decided by algorithms A and B respectively. In order to decide their union (or intersection) simply run A and B in parallel on the same given input string until they either accept or reject. The input string is accepted if and only if either one (or both, respectively) accepts it, and rejected otherwise.

Proposition: The decidable languages are closed under complementation.

Proof: Upon halting, simply exchange the verdicts accept and reject.

A language is called semi-decidable (or recognizable) if there exists an algorithm that accepts a given string if and only if the string belongs to that language. In case the string does not belong to the language, the algorithm either rejects it or runs forever.

Clearly, any decidable language is recognizable. We still have to see whether or not there are recognizable languages that are not decidable, and whether or not there are languages that are not recognizable.

Proposition: The recognizable languages are closed under union and intersection.

Proof: Let L and M be languages that are recognized by algorithms A and B respectively. In order to decide their union (or intersection) simply run A and B in parallel on the same given input string. The input string is accepted when either one (or both, respectively) accepts it.

Theorem: A language is decidable if and only if both it and its complement are recognizable.

Proof: Surely, a decidable language is recognizable. Moreover, if a language is decidable, then so is its complement, and hence that complement is recognizable.

Now suppose a language L and its complement is recognizable. Let A be a recognizer for L, and B for its complement. A decision method for L is obtained by running A and B in parallel on a given input string. In case A accepts the string, it is accepted as a member of L, and in case B accepts it, it is rejected as member of L. One of these outcomes will occur within a finite amount of time.

Definition: An enumerator is an algorithm that doesn't take an input, and outputs a possibly infinite sequence of strings, allowing repetitions.

A language is called enumerable if there exists an enumerator that outputs exactly those words that belong to that language.

Theorem: A language is recognizable if and only if it is enumerable.

Proof: Suppose a language L is recognizable. Let A be the algorithm that recognizes the strings that belong to L. It is not hard to come up with an enumerator that enumerates all pairs (w, n) with *w* a string and *n* a natural number. Construct an enumerator E for L as follows: enumerate all pairs (w, n), and for each such pair, run algorithm A on string *w* for *n* minutes. If in that time A accept *w*, output *w* as part of the enumeration. If it doesn't, just go on with the next pair. As every string in L will be accepted by A in a finite amount of time, it will be enumerated by E (infinitely often). Strings not accepted by A will never be enumerated.

Now suppose L is enumerable. Let E be an enumerator for L. Define algorithm A as follows: given an input string *w*, run E until it outputs w; when that happens accept. In case E will never output *w*, the algorithm A will run forever without accepting *w*. Therefore A accepts exactly those strings that are enumerated by E.

Based on this result, the words "semi-decidable", "recognizable" and "enumerable" may be used interchangeably.

Working hypothesis II: An algorithm can be presented as text, i.e. as a string of letters from the finite alphabet we use for communication.

The hypothesis above is extremely plausible, and can be taken for granted. Any method to recognize the strings in a language can be put into words.

Now take as alphabet all symbols used for communication, and consider the algorithms for recognizing languages over that alphabet. Such an algorithm can be represented (or given) by a string over that very alphabet. As any string can be fed to an algorithm, one distinguishes those algorithms (represented as strings) that accept themselves, from those that don't.

12.4 Undecidable Languages

Theorem: There exists an unrecognizable language.

Proof: Consider the set L of algorithms (for recognizing languages) - represented as strings - that do not accept themselves. I claim that L is an unrecognizable language. For assume that there is an algorithm, represented as a word *w*, that recognizes L. Suppose that *w* accepts itself. Then, by the definition of L, *w* is not in L. But by the definition of *w* recognizing L, *w* must be in L. This is a contradiction. Now suppose that *w* does not accept itself. Then, by the definition of L, *w* is in L. But by the definition of *w* recognizing L, *w* cannot be in L. This is a contradiction as well. As both possibilities concerning *w* accepting itself lead to a contradiction, the assumption that their exists an algorithm recognizing L must be false. Therefore L is not recognizable.

Hypothesis: For each piece of text we can decide whether it counts as an algorithm for recognizing languages or not.

The hypothesis above is extremely plausible, for when one cannot decide from a piece of text whether it is an algorithm or not, it appears impossible to apply that algorithm to recognize a particular language, and hence it is no algorithm at all. On the other hand, one might argue that there are pieces of text for which it is a judgment call to decide whether they are sufficiently precise to count as an algorithm. However, it appears that this kind of judgment calls can be resolved in variety of ways, and that each of these ways leads to a workable version of the hypothesis above.

Theorem: There exists a language that is recognizable but not decidable.

Proof: Consider the set M of algorithms (for recognizing languages) - represented as strings - that do accept themselves. I claim that the language M is recognizable but not decidable. For the first claim, construct an algorithm recognizing M as follows: when presented with a string w, check if w is an algorithm for recognizing languages or not. If not, reject. If so, run the algorithm represented by w on the string w. Accept exactly when w accepts. This algorithm clearly recognizes M.

For the second claim, suppose that M would be decidable. Then also its complement would be decidable, as well as the intersection of its complement with the language of all algorithms for recognizing languages. But this intersection is exactly L, the language shown to be unrecognizable, and thus surely undecidable, in the previous proof. This contradiction shows that M is undecidable.

Corollary (the acceptance problem): The language AP of all strings (v, w) where v is an algorithm accepting the word *w* is recognizable but undecidable.

Proof: That AP is recognizable is shown in the same way as for the language M above. Construct an algorithm recognizing AP as follows: when presented with a string (v, w), check if *v* is an algorithm for recognizing languages or not. If not, reject. If so, run the algorithm represented by *v* on the string *w*. Accept exactly when *w* accepts. This algorithm clearly recognizes AP.

That AP is undecidable follows by reduction to the undecidability of the language M above. Suppose we would have an algorithm deciding AP. Then an algorithm for deciding whether a string *w* is in M would simply consist of feeding the string (w, w) to the algorithm deciding AP. As M is undecidable, such an algorithm cannot exists, and hence AP must be undecidable as well.

The acceptance problem as formulated above suffers from ambiguity: it may be that a given string can be parsed in several different ways as a pair (v, w). This happens if there are commas in v or w. A solution to this problem is to write any comma "," appearing in v or w as "\," and any backslash "\" as "\\". Under this convention the comma separating v from w is uniquely recognizable as such.

We say that an algorithm halts on input w if it either accepts or rejects the word w (in a finite amount of time). An algorithm is halting if it halts on every input.

Corollary (the halting problem): The language HP of all strings (v, w) where v is an algorithm halting on the input w is undecidable.

Proof: by reduction to the undecidability of AP. Suppose we would have an algorithm deciding HP. Then an algorithm deciding AP is obtained as follows. Feed the input string to HP. If it is rejected, reject. Otherwise, the input string must have the form (v, w) in which v is an algorithm halting on input w. So run v on w and check whether w is accepted. One finds out about that in a finite amount of time, because it is known already that v will halt.

12.5 Halting Problem of Turing Machine

Turing machines are capable of doing any computation that computers can do, that is computationally they are equally powerful, and that any of their variations do not exceed the computational power of deterministic Turing machines. It is also conjectured that any "computation" human beings perform can be done by Turing machines (Church's thesis).

Halting Problem

One of well-known unsolvable problems is the halting problem. It asks the following question: Given an arbitrary Turing machine M over alphabet = {a , b}, and an arbitrary string w over, does M halt when it is given w as an input ?

It can be shown that the halting problem is not decidable, hence unsolvable.

12.6 The Post Correspondence Problem

The Post correspondence problem is an undecidable decision problem that was introduced by Emil Post in 1946. Because it is simpler than the halting problem and the Entscheidungs problem it is often used in proofs of undecidability.

The Post correspondence problem (due to Emil Post) is another undecidable problem that turns out to be a very helpful tool for proving problems in logic or in formal language theory to be undecidable.

12.7 Summary

Algorithms play very crucial role in performance improvement. They are not only method but as important as hardware technology. Recursively enumerable languages are known as type-0 languages in the Chomsky hierarchy of formal languages. All regular, context-free, context-sensitive and recursive languages are recursively enumerable.

12.8 Glossary

Algorithm: An algorithm is a well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output.

Decidability: A problem is decidable if some Turing machine decides (solves) the problem.

Regular Language: a regular language is a formal language that can be expressed using a regular expression, in the strict sense of the latter notion used in theoretical computer science.

Context-free Language: a context-free language (CFL) is a language generated by some context-free grammar (CFG).

Context-sensitive Language: a context-sensitive language is a formal language that can be defined by a context-sensitive grammar (and equivalently by a noncontracting grammar).

Recursive language: a formal language (a set of finite sequences of symbols taken from a fixed alphabet) is called recursive if it is a recursive subset of the set of all possible finite sequences over the alphabet of the language.

Recursively enumerable Language: A recursively enumerable language is a recursively enumerable subset in the set of all possible words over the alphabet of the language.

12.9 Exercise

Q.1 Let L1 be a recursive language. Let L2 and L3 be languages that are recursively enumerable but not recursive. Which of the following statements is not necessarily true?

(a) L2 – L1 is recursively enumerable.

(b) L1 – L3 is recursively enumerable

(c) L2 ∩ L1 is recursively enumerable

(d) L2 ∪ L1 is recursively enumerable

Q.2 If L and L' are recursively enumerable, then L is

(a) Regular

(b) context-free

(c) context-sensitive

(d) recursive

Q.3 Let L1 be a recursive language, and let L2 be a recursively enumerable but not a recursive language. Which one of the following is TRUE?

L1' --> Complement of L1

L2' --> Complement of L2

(a) L1' is recursive and L2' is recursively enumer-able

(b) L1' is recursive and L2' is not recursively enumerable

(c) L1' and L2' are recursively enumerable

(d) L1' is recursively enumerable and L2' is recursive

Q.4 Which of the following is true?

(a) The complement of a recursive language is recursive.

(b) The complement of a recursively enumerable language is recursively enumerable.

(c) The complement of a recursive language is either recursive or recursively enumerable.

(d) The complement of a context-free language is context-free.

Q.5 For any two languages L1 and L2 such that L1 is context free and L2 is recursively enumerable but not recursive, which of the following is/are necessarily true?

1. L1' (complement of L1) is recursive
2. L2' (complement of L2) is recursive
3. L1' is context-free
4. L1' ∪ L2 is recursively enumerable

(a) 1 only

(b) 3 only

(c) 3 and 4 only

(d) 1 and 4 only

Q.6 Let X be a recursive language and Y be a recursively enumerable but not recursive language. Let W and Z be two languages such that Y' reduces to W, and Z reduces to X' (reduction means the standard many-one reduction). Which one of the following statements is TRUE

(a) W can be recursively enumerable and Z is recursive.

(b) W an be recursive and Z is recursively enumerable.

(c) W is not recursively enumerable and Z is recursive.

(d) W is not recursively enumerable and Z is not recursive

Q.7 Consider the following types of languages:

L1 Regular,

L2: Context-free,

L3: Recursive,

L4: Recursively enumerable.

Which of the following is/are TRUE?

I. L3' U L4 is recursively enumerable

II. L2 U L3 is recursive

III. L1* U L2 is context-free

IV. L1 U L2' is context-free

(a) I only (b) I and III only (c) I and IV only (d) I, II and III only

12.10 Answers to Exercise

Ans.1: b

Explanation:

a) Always True as (Recursively enumerable - Recursive) is Recursively enumerable

b) Not always true as L1 - L3 = L1 intersection (Complement L3). L1 is recursive , L3 is recursively enumerable but not recursive Recursively enumerable languages are NOT closed under complement.

c) and d) Always true Recursively enumerable languages are closed under intersection and union.

Ans.2: d

Explanation: If L is recursively enumerable, then L' is recursively enumerable if and only if L is also recursive.

Ans.3: b

Explanation: Recursively enumerable languages are known as type-0 languages in the Chomsky hierarchy of formal languages. All regular, context-free, context-sensitive and recursive languages are recursively enumerable. Recursive Languages are closed under complementation, but recursively enumerable are not closed under complementation. If a languages L is recursively enumerable, then the complement of it is recursively enumerable if and only if L is also recursive. Since L2 is recursively enumerable, but not recursive, L2' is not recursively enumerable.

Ans.4: a

Ans.5: d

Explanation:

1. L1' (complement of L1) is recursive is true L1 is context free. Every context free language is also recursive and recursive languages are closed under complement.
2. L2' (complement of L2) is recursive is false: Recursively enumerable languages are not closed under set difference or complementation
3. L1' is context-free: Context-free languages are not closed under complement, intersection, or difference.
4. L1' ∪ L2 is recursively enumerable is true Since L1' is recursive, it is also recursively enumerable and recursively enumerable languages are closed under union. Recursively enumerable languages are known as type-0 languages in the Chomsky hierarchy of formal languages. All regular, context-free, context-sensitive and recursive languages are recursively enumerable.

Ans.6: c

Explanation: Since X is recursive and recursive language is closed under complement. So X' is also recursive. Since Z ≤ X' is recursive. (Rule : if Z is reducible to X' , and X' is recursive, then Z is recursive.) Option (B) and (D) is eliminated. And Y is recursive enumerable but not recursive, so Y' cannot be recursively enumerable. Since Y' reduces to W. And we know complement of recursive enumerable is not recursive enumerable and therefore, W is not recursively enumerable. So Correct option is (C). Here Y' is complement of Y and X' is complement of X.

Ans.7: d

Explanation:

Statement 1: As L3 is Recursive and recursive languages are closed under complementation, L3' will also be recursive. L3' U L4 is also recursive as recursive languages are closed under union.

Statement 2: As L2 is Context- Free, it will be recursive as well. L2 U L3 is recursive because as recursive languages are closed under union.

Statement 3: L1* is regular because regular languages are closed under kleene – closure. L1* U L2 is context free as union of regular and context free is context free.

Statement 4: L2' may or may not be context free because CFL are not closed under complementation. So it is not true. So I, II and III are correct.

References and Suggested Readings

1. Mishra, K. L. P., and N. Chandrasekaran. Theory of Computer Science: Automata, Languages and Computation. PHI Learning Pvt. Ltd., 2006.
2. "Decidability - Stanford University." N.p., n.d. Web. 20 Oct. 2016.
3. Recursively enumerable language. (2016, July 9). In Wikipedia, The Free Encyclopedia. Retrieved 17:04, July 9, 2016, from https://en.wikipedia.org/w/index.php?title=Recursively_enumerable_language&oldid=729067684.

UNIT-13

Introduction to Complexity Theory

Structure of the Unit

13.0 Objective

In this chapter we shall focus upon the following topics

- Growth Rate of Functions
- The Classes P and NP
- Polynomial Time Reduction and NP completeness
- Importance of NP complete problem
- Use of NP complete
- Other NP complete Problems

13.1 Growth Rate of Functions

One of the most important problems in computer science is to get the best measure of the growth rates of algorithms, best being those algorithms whose run times grow the slowest as a function of the size of their input. Efficiency can mean survival of a company.

When we look at input sizes large enough to make only the order of growth of the running time relevant, we are studying the asymptotic efficiency of algorithms. That is, we are concerned with how the running time of an algorithm increases with the size of the input in the limit, as the size of the input increases without bound. Usually, an algorithm that is asymptotically more efficient will be the best choice for all but very small inputs.

13.2 The Classes P and NP

The first is a class which contains all of the problems we solve using computers. If we think about the problems we actually present to the computer we note that not too many computations require more than $O(n^3)$ or $O(n^4)$ time. In fact, most of the important algorithms we compute are somewhere in the $O(\log n)$ to $O(n^3)$ range. Thus we shall state that practical computation resides within polynomial time bounds. There is a name for this class of problems.

The class of polynomially solvable problems, P contains all sets in which membership may be decided by an algorithm whose running time is bounded by a polynomial.

Besides containing all of what we have decided to consider practical computational tasks, the class P has another attractive attribute. Its use allows us to not worry about our machine model since all reasonable models of computation (including programs and Turing machines) have time complexities, which are polynomially related.

That was the class of problems we actually compute. But there is another important class. This one is the class of problems that we would love to solve but are unable to do so exactly.

Consider another problem that of finding a minimal length tour of n cities where we begin and end at the same place. (This is called the closed tour problem.) Again, there are many solutions; in fact n factorial different tours are possible.

And, once more, if we have a tour, we can easily check to see how long it is. Thus if we want a tour of less than some fixed length, we can quickly check candidates to see if they qualify.

This is interesting and provides some hope of solving problems of this kind. If we can determine the worth of an answer, then maybe we can investigate promising solutions and keep the best one.

Let us consider a class of problems, which all seem very complex, but have solutions, which are easily checked. Here is a class, which contains the problems for which solutions can be verified in polynomial time.

Definition: The class of nondeterministic polynomially acceptable problems, NP, contains all sets in which membership can be verified in polynomial time.

This may seem to be quite a bizarre collection of problems. But think for a moment. The examination scheduling problem does fit here. If we were to find a solution, it could be checked out very quickly. Lots of other problems fall into this category. Another instance is closed tours of groups of cities. Many graph problems used in CAD algorithms for computer chip design fit in here also. That's the difference: A problem is in P if we can decide them in polynomial time. It is in NP if we can decide them in polynomial time, if we are given the right certificate.

13.3 Polynomial Time Reduction and NP Completeness

In previous section we discussed P and NP classes. The general class of problems for which some algorithm can provide a solution in polynomial time is called "class P" or just "P". For some problems, there is no known way to find a solution quickly, but if one is provided with information showing what the solution is, it is possible to verify the solution swiftly. The class of problems for which a solution can be verified in polynomial time is called NP, which stands for "nondeterministic polynomial time."

A reduction is a way of converting one problem to another problem, so that the solution to the second problem can be used to solve the first problem.

- Finding the area of a rectangle, reduces to measuring its width and height
- Solving a set of linear equations reduces to inverting a matrix.

Reducibility involves two problems A and B. If A reduces to B, you can use a solution to B to solve A. When A is reducible to B, solving A cannot be "harder" than solving B. If A is reducible to B and B is decidable, then A is also decidable. If A is undecidable and reducible to B, then B is undecidable.

We want prove some problems are computationally difficult. As a first step, we settle for relative judgments:

Problem X is at least as hard as problem Y

To prove such a statement, we reduce problem Y to problem X:

If you had a black box that can solve instances of problem X, can you solve any instance of Y using polynomial number of steps, plus a polynomial number of calls to the black box that solves X?

If problem Y can be reduced to problem X, we denote this by

$$Y \leq_P X$$

This can be interpreted as "Y is polynomial-time reducible to X." It also means that X is at least as hard as Y because if you can solve X, you can solve Y.

Suppose $Y \leq_P X$, and there is a polynomial time algorithm for X. Then, there is a polynomial time algorithm for Y.

If $Y \leq_P X$ and Y cannot be solved in polynomial time, then X cannot be solved in polynomial time. It is contradicting the assumption "if we could solve X in polynomial time, then we'd be able to solve Y in polynomial time". Thus if we could find one hard problem Y, we could prove that another problem X is hard by reducing Y to X.

Example: Reduction of Independent Set to Vertex Cover

Independent Set: Given graph G and a number k, does G contain a set of at least k independent vertices?

Vertex Cover: Given a graph G and a number k, does G contain a vertex cover of size at most k.

Can we reduce independent set to vertex cover?

Theorem: If G = (V, E) is a graph, then S is an independent set implies that V - S is a vertex cover.

Proof: Suppose S is an independent set, and let e = (u, v) be some edge. Only one of u, v can be in S. Hence, at least one of u, v is in V - S. So, V - S is a vertex cover.

Suppose V - S is a vertex cover, and let $u, v \in S$. There can't be an edge between u and v (otherwise, that edge wouldn't be covered in V - S). So, S is an independent set.

In order to prove that Independent Set $\leq_p$ Vertex Cover, we change any instance of Independent Set into an instance of Vertex Cover.

Proof

- Given an instance of Independent Set < G, k >, with |G| = n
- We ask our Vertex Cover black box if there is a vertex cover of with n - k vertices.

By our previous theorem, S is an independent set if and only if V - S is a vertex cover. So, G has an independent set of size k if and only if G has a vertex cover of size n - k.

In computational complexity theory, a decision problem is NP-complete when it is both in NP and NP-hard. The set of NP-complete problems is often denoted by NP-C or NPC. A problem Y is NP-hard if, for every problem X in NP, $X \leq_p Y$. A Problem is in NP-complete if problem Y is in NP and is NP-hard.

We say X is NP-complete if:

- $X \in NP$
- for all $Y \in NP$, $Y \leq_p X$.

If these hold, then X can be used to solve every problem in NP. Therefore, X is definitely at least as hard as every problem in NP.

Theorem: If X is NP-complete, then X is solvable in polynomial time if and only if P = NP.

Proof. If P = NP, then X can be solved in polytime. Suppose X is solvable in polytime, and let Y be any problem in NP. We can solve Y in polynomial time: reduce it to X. Therefore, every problem in NP has a polytime algorithm and P = NP.

13.4 Importance of NP complete problem

The class of NP-complete (Non-deterministic polynomial time complete) problems is a very important and interesting class of problems in Computer Science. The interest surrounding this class of problems can be attributed to the following reasons.

- No polynomial-time algorithm has yet been discovered for any NP-complete problem; at the same time no NP-complete problem has been shown to have a super polynomial-time (for example exponential time) lower bound.
- If a polynomial-time algorithm is discovered for even one NP-complete problem, then all NP-complete problems will be solvable in polynomial-time. It is believed (but so far no proof is available) that NP-complete problems do not have polynomial-time algorithms and therefore are intractable. The basis for this belief is the second fact above, namely that if any single NP-complete problem can be solved in polynomial time, then every NP-complete problem has a polynomial-time algorithm. Given the wide range of NP-complete problems that have been discovered to date, it will be sensational if all of them could be solved in polynomial time.

It is important to know the rudiments of NP-completeness for anyone to design "sound" algorithms for problems. If one can establish a problem as NP-complete, there is strong reason to believe that it is intractable. We would then do better by trying to design a good approximation algorithm rather than searching endlessly seeking an exact solution. An example of this is the TSP (Traveling Salesman Problem), which has been shown to be intractable. A practical strategy to solve TSP therefore would be to design a good approximation algorithm. A variation of Kruskal's minimal spanning tree algorithm is used to approximately solve the TSP. Another important reason to have good familiarity with NP-completeness is many natural interesting and innocuous-looking problems that on the surface seem no harder than sorting or searching are in fact NP-complete.

13.5 Use of NP complete

One practical use in discovering that problem is NP-complete is that it prevents us from wasting our time and energy over finding polynomial or easy algorithms for that problem. Also we may not need the full generality of an NP-complete problem. Particular cases may be useful and they may admit polynomial algorithms.

Also there may exist polynomial algorithms for getting an approximate optimal solution to a given NP-complete problem. For example, the travelling salesman problem satisfying the triangular inequality for distances between cities (i.e. $d_{ij} \leq d_{ik} + d_{kj}$ for all i, j, k) has approximate polynomial algorithm such that the ratio of the error to the optimal values of total distance travelled is less than or equal to 1/2.

13.6 Other NP complete Problems

There are hundreds of known problems in the list of NP-Complete. Here is a list of some of the more commonly known problems that are NP-complete when expressed as decision problems. Almost each and every area has problems of this class. Some of them are as follow:

- **Graphs and hyper graphs**

  Graphs occur frequently in everyday applications. Examples include biological or social networks, which contain hundreds, thousands and even billions of nodes in some cases. This area includes, 1-planarity, 3-dimensional matching, Bipartite dimension, Capacitated minimum spanning tree, Route inspection problem, Clique problem, Complete coloring, Domatic number, Dominating set, Hamiltonian completion, Longest path problem, Minimum k-cut, Minimum spanning tree, or Steiner tree, for a subset of the vertices of a graph. (The minimum spanning tree for an entire graph is solvable in polynomial time.)

- **Mathematical programming**

  This class includes 3-partition problem, Bin packing problem, Knapsack problem, quadratic knapsack problem, and several variants, Variations on the Traveling salesman problem. The problem for graphs is NP-complete if the edge lengths are assumed integers. The problem for points on the plane is NP-complete with the discretized Euclidean metric and rectilinear metric. The problem is known to be NP-hard with the (non-discretized) Euclidean metric.

- **Formal languages and string processing**

  It includes closest string, longest common subsequence problem, the bounded variant of the Post correspondence problem, shortest common super-sequence.

- **Games and puzzles**

  Battleship, Bulls and Cows, marketed as Master Mind, Candy Crush Saga and Verbal arithmetic etc.

- **Other NP-Complete Problems**

  It includes Art gallery problem and its variations, Berth allocation problem, Assembling an optimal Bit coin block, Boolean satisfiability problem (SAT). There are many variations that are also NP-complete. An important variant is where each clause has exactly three literals (3SAT), since it is used in the proof of many other NP-completeness results and Vehicle routing problem.

13.7 Summary

Complexity theory, also called complexity strategy is the use of the study of complexity systems in the field of strategic management and organizational studies. Complexity theory is an interdisciplinary theory that grew out of systems theory in the 1960s. It is useful to analyze nature of problems.

13.8 Glossary

Class P: It consists of all those decision problems that can be solved on a deterministic sequential machine in an amount of time that is polynomial in the size of the input.

Class NP: The class NP consists of all those decision problems whose positive solutions can be verified in polynomial time.

Polynomial-time reduction: Polynomial-time reduction is a method of solving one problem by means of a hypothetical subroutine for solving a different problem (that is, a reduction), that uses polynomial time excluding the time within the subroutine.

NP completeness: A decision problem is NP-complete when it is both in NP and NP-hard

13.9 Exercise

Q.1 Assuming P != NP, which of the following is true ?

(A) NP-complete = NP

(B) $NP-Complete \cap P = \phi$

(C) NP-hard = NP

(D) P = NP-complete

Q.2 Let S be an NP-complete problem and Q and R be two other problems not known to be in NP. Q is polynomial time reducible to S and S is polynomial-time reducible to R. Which one of the following statements is true?

(A) R is NP-complete

(B) R is NP-hard

(C) Q is NP-complete

(D) Q is NP-hard

Q.3 Let X be a problem that belongs to the class NP. Then which one of the following is TRUE?

(A) There is no polynomial time algorithm for X.

(B) If X can be solved deterministically in polynomial time, then P = NP.

(C) If X is NP-hard, then it is NP-complete.

(D) X may be undecidable

Q.4 Which of the following is true about NP-Complete and NP-Hard problems.

(A) If we want to prove that a problem X is NP-Hard, we take a known NP-Hard problem Y and reduce Y to X

(B) The first problem that was proved as NP-complete was the circuit satisfiability problem.

(C) NP-complete is a subset of NP Hard

(D) All of the above

(E) None of the above

Q.5 A problem in NP is NP-complete if

(A) It can be reduced to the 3-SAT problem in polynomial time

(B) The 3-SAT problem can be reduced to it in polynomial time

(C) It can be reduced to any other problem in NP in polynomial time

(D) some problem in NP can be reduced to it in polynomial time

Q.6 For problems X and Y, Y is NP-complete and X reduces to Y in polynomial time. Which of the following is TRUE?

(A) If X can be solved in polynomial time, then so can Y

(B) X is NP-complete

(C) X is NP-hard

(D) X is in NP, but not necessarily NP-complete

13.10 Answers to Exercise

Ans.1: B

Explanation: The answer is B (no NP-Complete problem can be solved in polynomial time). Because, if one NP-Complete problem can be solved in polynomial time, then all NP problems can solved in polynomial time. If that is the case, then NP and P set become same which contradicts the given condition.

Ans.2: B

Explanation:

(A) Incorrect because R is not in NP. A NP Complete problem has to be in both NP and NP-hard.

(B) Correct because a NP Complete problem S is polynomial time educable to R. (C) Incorrect because Q is not in NP.

(D) Incorrect because there is no NP-complete problem that is polynomial time Turing-reducible to Q.

Ans.3: C

Explanation:

(A) is incorrect because set NP includes both P(Polynomial time solvable) and NP-Complete .

(B) is incorrect because X may belong to P (same reason as (A))

(C) is correct because NP-Complete set is intersection of NP and NP-Hard sets.

(D) is incorrect because all NP problems are decidable in finite set of operations.

Ans.4: D

Ans.5: B

Explanation: A problem in NP becomes NPC if all NP problems can be reduced to it in polynomial time. This is same as reducing any of the NPC problem to it. 3-SAT being an NPC problem, reducing it to a NP problem would mean that NP problem is NPC.

Ans.6: D

References and Suggested Readings

1. Mishra, K. L. P., and N. Chandrasekaran. Theory of Computer Science: Automata, Languages and Computation. PHI Learning Pvt. Ltd., 2006.
2. Computational complexity theory. (2016, December 7). In Wikipedia, The Free Encyclopedia. Retrieved 09:19, December 7, 2016, from https://en.wikipedia.org/w/index.php?title=Computational_complexity_theory&oldid=753465257
3. NP-completeness. (2016, December 18). In Wikipedia, The Free Encyclopedia. Retrieved 19:48, December 18, 2016, from https://en.wikipedia.org/w/index.php?title=NPcompleteness&oldid=755550998
4. NP-hardness. (2016, December 21). In Wikipedia, The Free Encyclopedia. Retrieved 16:01, December 21, 2016, from https://en.wikipedia.org/w/index.php?title=NP-hardness&oldid=756030165

UNIT-14
Application Area

Structure of the Unit

14.0 Objective

In this unit we discuss about the application areas of where formal languages and automata are used. The objective is to understand the application areas of formal language and automata in A.I. (Artificial Intelligence), in N.L.P. (Natural Language Processing), and various other applications like String Matching Algorithms, Compiler Construction (Lexical Analyzers), Complexity Theory, Network Protocols.

14.1 Introduction

Theory of computation is a part of theoretical Computer Science. Theory of computation is mainly concerned with the study of how problems can be solved using algorithms. The Theory of Computation aims at understanding the nature of computation, and specifically the inherent possibilities and limitations of efficient computations.

A model of computation is the definition of the operations used in computation. It is used to measure the complexity of an algorithm in execution time and memory space. Examples of models of computation other than Turing machines are: finite state machines, recursive functions, lambda calculus, combinatory logic, and abstract rewriting systems. The Theory of Programming is concerned with the actual task of implementing computations (i.e., writing computer programs).

There are broad application areas of theory of computation like Artificial Intelligence, Natural Language Processing, Cryptography, and Distributed Computing etc. Finite automata are used in text processing, compilers, and hardware design. Context-free grammar (CFGs) are used in programming languages and artificial intelligence. Originally, CFGs were used in the study of the human languages.

14.2 Application of Automata in A.I.

The formality of automata theory can be applied to the analysis and manipulation of actual human language as well as the development of human-computer interaction (HCI) and artificial intelligence (AI).

Automata theory is the basis for the theory of formal languages. A proper treatment of formal language theory begins with some basic definitions:

- A symbol is simply a character, an abstraction that is meaningless by itself.
- An alphabet is a finite set of symbols.
- A word is a finite string of symbols from a given alphabet.
- Finally, a language is a set of words formed from a given alphabet.

The set of words that form a language is usually infinite, although it may be finite or empty as well. Formal languages are treated like mathematical sets, so they can undergo standard set theory operations such as union and intersection. Additionally, operating on languages always produces a language. As sets, they are defined and classified using techniques of automata theory.

Formal languages are normally defined in one of three ways, all of which can be described by automata theory:

- regular expressions
- standard automata
- a formal grammar system

Regular Expressions Example

alphabet A1 = {a, b}

alphabet A2 = {1, 2}

language L1 = the set of all words over A1 = {a, aab, ...}

language L2 = the set of all words over A2 = {2, 11221, ...}

language L3 = L1 $\cup$ L2

language L4 = {a^n | n is even} = {aa, aaaa, ...}

language L5 = {$a^n b^n$ | n is natural} = {ab, aabb, ...}

Languages can also be defined by any kind of automaton, like a Turing Machine. In general, any automata or machine M operating on an alphabet A can produce a perfectly valid language L. The system could be represented by a bounded Turing Machine tape, for example, with each cell representing a word. After the instructions halt, any word with value 1 (or ON) is accepted and becomes part of the generated language. From this idea, one can define the complexity of a language, which can be classified as P or NP, exponential, or probabilistic, for example.

Noam Chomsky extended the automata theory idea of complexity hierarchy to a formal language hierarchy, which led to the concept of formal grammar. A formal grammar system is a kind of automata specifically defined for linguistic purposes. The parameters of formal grammar are generally defined as:

- a set of non-terminal symbols N
- a set of terminal symbols Σ

- a set of production rules P
- a start symbol S

Grammar Example

start symbol = S

non-terminals = {S}

terminals = {a, b}

production rules: S ⟶ aSb, S ⟶ ba

S ⟶ aSb ⟶ abab

S ⟶ aSb ⟶ aaSbb ⟶ aababb

L = {abab, aababb,}

As in purely mathematical automata, grammar automata can produce a wide variety of complex languages from only a few symbols and a few production rules. Chomsky's hierarchy defines four nested classes of languages, where the more precise a classes have stricter limitations on their grammatical production rules.

These rules and

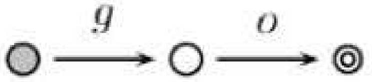
14.3 Applications of Automata in N.L.P. (Natural Language Processing).

The goal of natural language processing (NLP) is to build computational models of natural language for its analysis and generation. Formal-state automata can be viewed as a recognizing devices i.e. for some automaton A and a word w apply some rule to produce the result for example Is w a member of L(A), the language accepted by automaton A. This property of automata can be viewed as a necessary application of natural language processing.

We can better understand by taking an example of Dictionary lookup, which is frequently used in NLP:

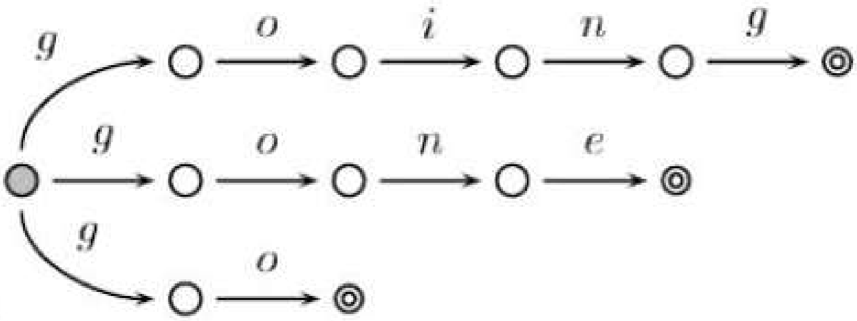
As we know the dictionary has thousands of words of entries in it, Finite state automata provides an efficient means for storing dictionaries, accessing them and modifying their contents.

Suppose a finite state machine has alphabet 'a' to 'z' and consider how a single word 'go' can be represented:



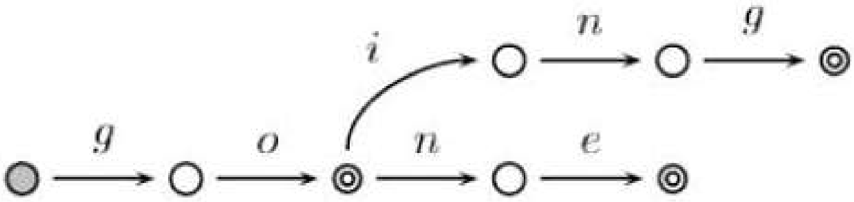
go:

To represent more than one word, we simply add paths to our "lexicon", one path for each additional word as:-



go, gone, going:

The above automaton can be minimized as:



go, gone, going:

As you see that after minimization the lookup operation amount to checking the word is in dictionary or not is also less. This is efficient operation that the time required to search a word is linear in the length of w.

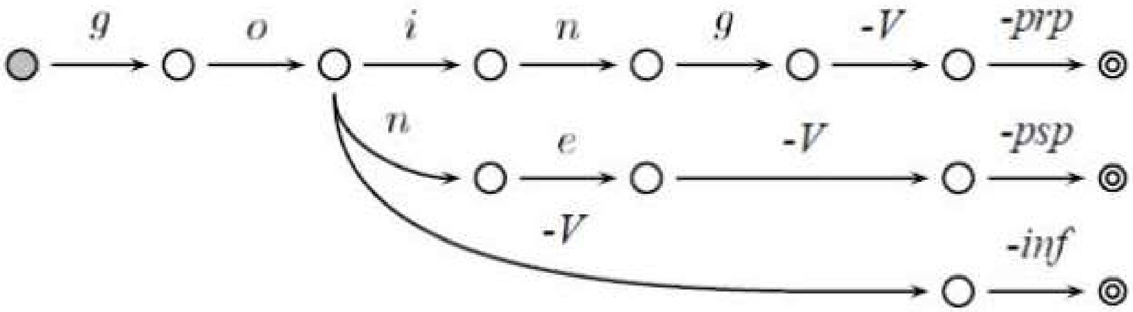
For real application one is usually interested in associating certain information with every word in the lexicon. For simplicity, assume that we do not have to list a full dictionary entry with each word; rather, we only need to store some morpho-phonological information, such as the part-of-speech of the word, or it's tense (in the case of verbs) or number (in the case of nouns). One way to achieve this goal is by extending the alphabet Σ: in addition to "ordinary" letters, Σ can include also special symbols, such as part-of-speech tags, morpho-phonological information, etc. An "analysis" of a (natural language) word will in this case amount to recognition by the automaton of an extended word, w, followed by some special tags.

Example:

Suppose we want to add to the lexicon information about part-of-speech, and we use two tags: -N for nouns, -V for verbs. Additionally, we encode the number of nouns as -sg or -pl, and the tense of verbs as -inf, -prp or -psp (for infinitive, present participle and past participle, respectively). It is very important to note that the additional symbols are multi-character symbols: there is nothing in common to the alphabet symbol -sg and the sequence of two alphabet letters <s,g>! In other words, the extended alphabet is:

Σ = {a, b, c, …… , z, -N, -V, -inf, -prp, -psp, -sg, -pl}

With the extended alphabet, we might construct the following automaton:



The language generated by the above automaton is no longer a set of words in English. Rather, it is a set of (simplisticly) "analyzed" strings, namely {go-V-inf, gone-V-psp, going-V-prp}.

Regular languages are particularly appealing for natural language processing for two main reasons.

First, it turns out that most phonological and morphological processes can be straight-forwardly described using the operations that regular languages are closed under, and in particular concatenation. With very few exceptions (such as the interdigitation word-formation processes of Semitic languages or the duplication phenomena of some Asian languages), the morphology of most natural languages is limited to simple concatenation of affixes, with some morpho-phonological alternations, usually on a morpheme boundary. Such phenomena are easy to model with regular languages, and hence are easy to implement with finite state automata. The other advantage of using regular languages is the inherent efficiency of processing with finite-state automata. Most of the algorithms one would want to apply to finite-state automata take time proportional to the length of the word being processed, independently of the size of the automaton. In computational terminology, this is called linear time complexity, and is as good as things can get.

14.4 Various other applications of formal languages and their automata

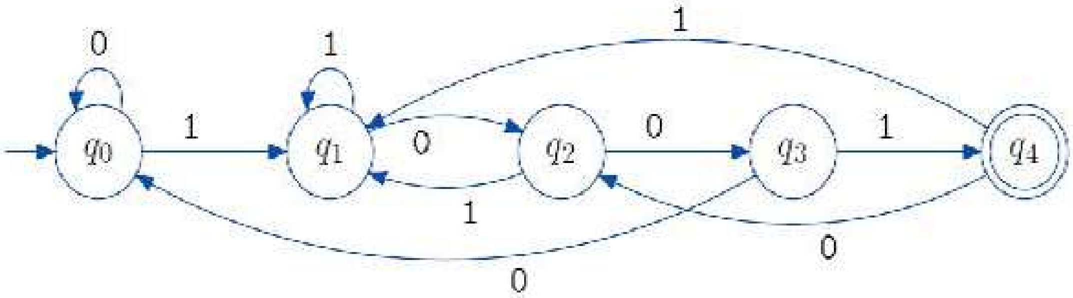
The various other applications of formal languages and automata are:-

- String Matching Algorithms.
- Compiler Construction (Lexical Analyzers).
- Complexity Theory.
- Network Protocols

Let us discuss one by one in detail.

(a) **String Matching Algorithms**: Our first application involves finding all occurrences of a short string (pattern string) within a long string (text string). This can be done by processing the text through a DFA: the DFA for all strings that end with the pattern string. Each time the accept state is reached, the current position in the text is output.
Example: Finding 1001

To find all occurrences of pattern 1001, construct the DFA for all strings ending in 1001.



(b) **Compiler Construction (Lexical Analyzers)**: A lexical analyzer is the part of a compiler that groups the characters of a program into lexical items or tokens. The modern approach to specifying a lexical analyzer for a programming language uses regular expressions. E.g., this is the approach taken by the lexical analyser generator Lex.

In compiling a program, the first step is lexical analysis. This isolates keywords, identifiers etc., while eliminating irrelevant symbols. A token is a category, for example "identifier", "relation operator" or specific keyword. The lexical analyzer gets the input character by character. Then by using pattern-matching technique, it identifies the symbol. Symbol table manager and error handler also associated with the six phases of a compiler. The methods used to implement lexical analyzers can also be applied to other areas such as query languages and information retrieval systems.

SUM	=	NUMBER1
IDENTIFIER1	OPERATOR	IDENTIFIER2
+	NUMBER2	;
OPERATOR	IDENTIFIER3	SEPERATOR

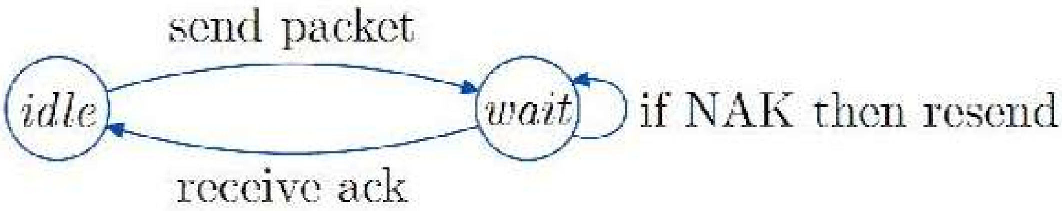
LEXEME	**Token category**
SUM	"Identifier"
=	"Assignment operator"
NUMBER1	"Identifier"

+	“Addition opeartor”
NUMBER2	"Identifier"
;	"End of statement."

A lexeme is the unit derived from the source program with each group is related to anyone symbolic category. The given source code "SUM=NUMBER1+NUMBER2; " is having six lexeme, as given in the table 1. The lexical analyzer read the input character by character until finding the lexeme. First character ‘S’ is obtained. Then character ‘U’ followed by ‘M’ and ‘=’. The last character ‘=’ omitted The first three characters combined to form the lexeme “SUM”. Then the character ‘=’ and ‘N’ scanned by the lexical analyzer. The last character ‘N’ omitted to form the lexeme “=”. Then the remaining lexemes “NUMBER1”, “+”,"NUMBER2”, ";" identified by the lexical analyzer. The pattern matching technique is used to match the lexeme with the token type.

(c) **Complexity Theory**: As discussed in Unit 13.

(d) **Network Protocols**: A finite-state machine is an FA together with actions on the arcs. A trivial example for a communication link:



14.5 Self Learning Exercise

Q.1 Which of the following is the part of a compiler

a) Regular files

b) Device files

c) Compiler

d) Directory files

Q. 2 Which of the following is an example of natural language processing (NLP):

a) Dictionary lookup.

b) Noam Chomsky.

c) TOC.

d) None of the above.

14.6 Summary

- The Theory of Computation aims at understanding the nature of computation, and specifically the inherent possibilities and limitations of efficient computations.
- The application areas of formal language and automata includes A.I. (Artificial Intelligence), in N.L.P. (Natural Language Processing), and various other applications like String Matching Algorithms, Compiler Construction (Lexical Analyzers), Complexity Theory, Network Protocols.
- The formality of automata theory can be applied to the analysis and manipulation of actual human language as well as the development of human-computer interaction (HCI) and artificial intelligence (AI).
- The goal of natural language processing (NLP) is to build computational models of natural language for its analysis and generation.
- A lexical analyzer is the part of a compiler that groups the characters of a program into lexical items or tokens.
- The modern approach to specifying a lexical analyzer for a programming language uses regular expressions.
- The pattern matching technique is used to match the lexeme with the token type.
- A finite-state machine is an FA is used to simulate Network Protocols.

14.7 Answers to Self-Learning Exercise

Q.1 (c)

Q.2 (a)

Q.3 (a)

14.8 Exercise

Q. 1 How Turing Machine is used in computing complexity of an algorithm? Explain with example.

Q. 2 What is the significance of Automaton? Discuss various applications of formal language and automata.

Q. 3 What do you mean by Natural Language processing? How NLP is related to Formal language and automaton? Discuss in detail.

Q. 4 Construct the DFA to find all occurrences of pattern 1001. Also explain it.

References and Suggested Readings

1. Peter Linz, An Introduction to Formal Languages and Automata, Jones and Bartlett Publication, Third Edition.
2. K.L.P. Mishra, N. Chandrasekaran, Theory of Computer Science, BPB Publication, Prentice-Hall of India, Second Edition.
3. H.E. Hopcraft and J.D. Ullamn, Introduction to Automata Theory, Languages and Computation, Narosa Publications.
4. J.C. Martin, Introduction to Languages and the Theory of Automata, Tata McGraw-Hill.