M.Sc., Previous Year

Mathematics

(Option - II)

ADVANCED DISCRETE MATHEMATICS

मध्यप्रदेश भोज (मुक्त) विश्वविद्यालय – भोपाल

MADHYA PRADESH BHOJ (OPEN) UNIVERSITY - BHOPAL

COURSE WRITERS

VK Khanna & SK Bhambri, Formerly Associate Professors, Department of Mathematics, Kirori Mal College, University of Delhi.
Units (1.0, 1.1, 1.3, 1.3.1, 1.3.2, 1.3.3, 1.5, 1.6, 1.7, 1.8, 1.9)

N Ch S N Iyengar, Professor, Deptt. of Computer Applications, Vellore Institute of Technology, Vellore.
V M Chandrasekaran, Asstt. Professor, Deptt. of Mathematics, Vellore Institute of Technology, Vellore.
K A Venkatesh, Head – Deptt. of Computer Applications, Alliance Business Academy, Bangalore.
P S Arunachalam. Senior Lecturer, Department of Mathematics, SRM Engineering College, Chennai.
Units (1.2, 1.4, Unit 3, 4.7)

Dr. Shamim Akhtar, Associate Professor, Jaypee Institute of Information Technology (JIIT), Noida.
Unit (Unit 2)

J C Kavitha, Senior Lecturer, Krishna Engineering College, Ghaziabad.
Units (4.0, 4.1, 4.2, 4.3, 4.4, 4.5, 4.6, 4.8, 4.9, 4.10, 4.11, 4.12, Unit 5)

Information contained in this book has been published by VIKAS® Publishing House Pvt. Ltd. and has been obtained by its Authors from sources believed to be reliable and are correct to the best of their knowledge. However, the Madhya Pradesh Bhoj (Open) University, Bhopal, Publisher and its Authors shall in no event be liable for any errors, omissions or damages arising out of use of this information and specifically disclaim any implied warranties or merchantability or fitness for any particular use.

Published by Registrar, MP Bhoj (open) University, Bhopal in 2020

VIKAS®

Vikas® is the registered trademark of Vikas® Publishing House Pvt. Ltd.

VIKAS® PUBLISHING HOUSE PVT. LTD.
E-28, Sector-8, Noida - 201301 (UP)
Phone: 0120-4078900 • Fax: 0120-4078999
Regd. Office: A-27, 2nd Floor, Mohan Co-operative Industrial Estate, New Delhi 1100 44
• Website: www.vikaspublishing.com • Email: helpline@vikaspublishing.com

SYLLABI-BOOK MAPPING TABLE

Advanced Discrete Mathematics

Syllabi	Mapping in Book
UNIT I **Formal Logic:** Statement, Symbolic representation, Tautologies, Quantifiers, Predicates and Validity, Propositional Logic; Semigroups and Monoids: Definitions and Examples of Semigroups and Monoids (including those pertaining to Concentration Operations); Homomorphism of Semigroups and Monoids, Congruence Relation and Quotient Semigroups, Sub Semigroups and Sub Mmonoids, Direct Products, Basic Homomorphism Theorem. **Lattices:** Lattices as Partially Ordered Sets, their Properties, Lattices and Algebraic Systems, Sub Lattices, Direct Products and Homomorphism, Some Special Lattices, for example, Complimented and Distributive Lattices.	**Unit 1:** Formal Logic (**Pages: 3-58**)
UNIT II **Boolean Algebra:** Boolean Algebra as Lattices, Various Boolean Identities, Join-Irreducible Elements, Atoms and Minterms, Boolean Forms and their Equivalence, Minterm Boolean Forms, Sum Of Products, Canonical Forms, Minimization of Boolean Functions, Applications of Boolean Algebra to Switching Theory (using AND, OR and NOT gates), The Karnaugh Map Method.	**Unit 2:** Boolean Algebra (**Pages: 59-102**)
UNIT III **Graph Theory:** Definition of (Undirected) Graphs, Paths, Circuits, Cycles and Subgroups, Induced Subgraphs, Degree of a Vertex, Connnectivity, Planar Graphs and their Properties;Trees, Euler's Formula for Connected Planar Graphs, Complete and Complete Bipartite Graphs; Kuratowski's Theorem (statement only) and its Use; Spanning Trees, Cut-Sets, Fundamental Cut-Sets and Cycles/ Minimal Spanning Trees and Kruskal's Algorithm, Matrix Representations of Graphs, Euler's Theorem on the Existence of Eulerian Paths and Circuits, Directed Graphs. Indegree and Outdegree of a Vertex, Weighted Undirected Graphs, Dijkstra's Algorithm, Strong Connectivity & Warshall's Algorithm, Directed Trees, Search Trees, Tree Traversals.	**Unit 3:** Graph Theory (**Pages: 103-146**)
UNIT IV **Introductory Computability Theory:** Finite State Machines and Their Transition Table Diagrams, Equivalence of Finite State Machines, Reduced Machines, Homomorphism, Finite automata, Acceptors, Non-Deterministic Finite Automata and Equivalence of its Power to that of Deterministic Finite Automata, Moore and Mealy Machines.	**Unit 4:** Introductory Computability Theory (**Pages: 147-172**)
UNIT V **Grammar and Languages:** Phrase Structure Grammars, Rewriting Rules, Derivations Sentential Forms, Language Generated by Grammar, Regular Context Free and Context Sensitive Grammar and Languages, Regular Sets, Regular Expressions and the Pumping Lemma, Kleene's Theorem, Notions of Syntax Analysis, Polish Notations, Conversion of Infix Expressions to Polish Notations, The Reverse Polish Notation.	**Unit 5:** Grammar and Languages (**Pages: 173-221**)

CONTENTS

INTRODUCTION

NOTES

Discrete mathematics is the branch of mathematics that deals with objects that can assume only discrete values. Thus, discrete mathematics is the study of mathematical structures that are fundamentally discrete rather than continuous. In contrast to real numbers that have the property of varying smoothly, the objects studied in discrete mathematics, such as integers, graphs and statements in logic have distinct and separated values. Hence, discrete mathematics excludes topics of continuous mathematics, such as calculus and analysis. Discrete objects can often be enumerated by integers. The set of objects studied in discrete mathematics can be finite or infinite. Logical formulas are discrete structures which form finite trees or more generally directed acyclic graph structures. Graphs are one of the prime objects of study in discrete mathematics. Discrete Algebra includes Boolean algebra which is used in logic gates and programming, while Relational Algebra is used in databases, discrete and finite versions of groups, rings and fields, discrete semi-groups and monoids.

The theory of automata and formal languages is the cornerstone of computer science. It helps you deal with transition systems that are more general than finite automata. A finite automaton is defined as a device which consists of a finite memory and either accepts or rejects the given inputs. Using a finite automaton, you can construct a transition graph and transition table, which help in the evaluation of a regular expression. A formal language is defined as an organized set of symbols that denote the language with their shapes and locations. The theory of formal languages is a branch of mathematics and computer science that exclusively studies language syntax. Problems based on the computational model are efficiently solved by the theory of formal languages and finite automata. This theory constitutes a major part of the theory of computation, which helps you understand the basic principles of computer science.

This book, *Advanced Discrete Mathematics*, is divided into five units and aims to give students the ability to think conceptually in mathematical terms by presenting mathematical techniques, together with simple and clear explanations of the concepts behind them. The topics covered include mathematical logic, propositional logic, semi-groups and monoids, lattices, Boolean algebra, atoms, minterms, Karnaugh map, graphs, complete and bipartite graphs, Kuratowski’s theorem, spanning tree, matrix representation of graphs, Eulerian paths and circuits, trees, tree traversals, finite automata, non-deterministic finite automata, grammars and languages, pumping lemma, and syntax analysis.

All these topics are important for deducing any formula and theorem with accurate logic and mathematical functions. The book follows the self-instruction mode or the SIM format wherein each unit begins with an ‘Introduction’ to the topic followed by an outline of the ‘Objectives’. The content is presented in a simple and structured form interspersed with ‘Check Your Progress’ questions for better understanding. A list of ‘Key Terms’ along with a ‘Summary’ and a set of ‘Self-Assessment Questions and Exercises’ is provided at the end of the each unit for effective recapitulation.

UNIT 1 FORMAL LOGIC

NOTES

Structure

1.0 INTRODUCTION

In this unit, you will learn about Boolean logic and group theory. It resembles the algebra of real numbers, but with the numeric operations of multiplication (xy), addition ($x + y$) and negation ($–x$) are replaced by the respective logical operations of conjunction ($x \wedge y$), disjunction ($x \vee y$) and negation ($\neg x$). Boolean algebra is the algebra of two values. These are usually taken to be 0 and 1. In mathematics, group theory studies the algebraic structures known as groups. A group is an algebraic structure consisting of a set together with an operation that combines any two of its elements to form a third element. To qualify as a group, the set and the operation must satisfy a few conditions called group axioms, namely closure, associativity, identity and invertibility. A subgroup is a subset of group elements of a group that satisfies the four group requirements. A monoid is an algebraic structure with a single associative binary operation and an identity element. Monoids are studied in semi-group theory as they are naturally semi-groups with identity.

1.1 OBJECTIVES

After going through this unit, you will be able to:

- Understand mathematical logic and its applications
- Write propositions in mathematical form

- Use various logical operators
- Construct truth tables of various propositions
- Explain the equivalence formula, tautology and inference theory
- Check the consistency of a statement formula
- Understand predicate calculus
- Explain groups, semi-groups and monoids
- Understand what lattices are

1.2 FORMAL LOGIC

Formal logic is a set of rules used in deductions which are self evident. Logic assumes something that can be True or False.

1.2.1 Mathematical Logic

One of the main aims of mathematical logic is to provide rules. The rules of logic give precise meaning to mathematical statements and distinguish between valid and invalid mathematical arguments. In addition, logic has numerous applications in computer science. These rules are used in the design of computer circuits, construction of computer programs, verification of the correctness of programs, and in many other ways.

Propositions: A proposition is a statement to which only one of the terms, true or false, can be meaningfully applied.

The value of a proposition if true is denoted by 1and if false is denoted by 0. Occasionally they are also denoted by the symbols T and F.

The following are propositions:

(*i*) $4 + 2 = 6$

(*ii*) 4 is an even integer and 5 is not.

(*iii*) 5 is a prime number.

(*iv*) New Delhi is the capital of India.

(*v*) $2 \in \{1, 3, 5, 7\}$

(*vi*) $42 \geq 51$

(*vii*) Paris is in England.

Of the above propositions, (*i*)–(*iv*) are true whereas (*v*)–(*vii*) are false.

The following are *not* propositions:

(*i*) Where are you going? (*ii*) $x + 2 = 5$

(*iii*) $x + y < z$ (*iv*) Beware of dogs

The expressions (*i*) and (*iv*) are not propositions since neither is true or false. The expression (*ii*) and (*iii*) are not propositions, since the variables in these expressions have not been assigned values and hence they are neither true or false.

Note: Letters are used to denote propositions just as letters are used to denote variables. The conventional letters used for this purpose are $p, q, r, s, \ldots$

1.2.2 Logical Operators

There are several ways in which we commonly combine simple statements into compound ones. The words *or*, and, not, if… then and if and only if, can be added to one or more propositions to create a new proposition. New propositions are called compound propositions. Logical operators are used to form new propositions or compound propositions. These logical operators are also called connectives.

NOTES

Conjunction (AND): If p and q are propositions, then the proposition 'p AND q', denoted by $p \wedge q$, is true when both p and q are true and is false otherwise. The proposition $p \wedge q$ is called the conjunction of p and q.

The truth table for $p \wedge q$ is shown in Table 1.1. Note that there are four rows in this truth table, one row for each possible combination of truth values of the propositions p and q.

Table 1.1 *Truth Table for Conjunction*

p	q	$p \wedge q$
0	0	0
0	1	0
1	0	0
1	1	1

Example 1.1: Find the conjunction of the propositions p and q where p is the proposition 'Today is Sunday' and q is the proposition 'It is raining today'.

Solution: The conjunction of these two propositions is $p \wedge q,$ the proposition, 'Today is Sunday and it is raining today'.

Example 1.2: Let p be 'Ravi is rich' and let q be 'Ravi is happy'. Write each of the following in symbolic form:

(*i*) Ravi is poor but happy.

(*ii*) Ravi is neither rich nor happy.

(*iii*) Ravi is rich and unhappy.

Solution:

(*i*) $\sim p \wedge q$ (*ii*) $\sim p \wedge \sim q$ (*iii*) $p \wedge \sim q$

Disjunction (OR): If p and q are propositions, then disjunction p or q, denoted as $p \vee q$, is false when p and q are both false and true otherwise (Refer Table 1.2). The proposition $p \vee q$ is called the disjunction of p and q.

Note that connectives $\sim$ and $\wedge$ defined earlier have the same meaning as the words 'NOT' and 'AND' in general. However, the connective $\vee$ is not always the same as the word 'OR' because of the fact that the word 'OR' in English is commonly used both as an 'Exclusive OR' and as an 'Inclusive OR'. For example, consider the following statements:

(*i*) I shall watch the movie on TV or go to cinema.

(*ii*) There is something wrong with the fan or with the switch.

(*iii*) Ten or twenty people were killed in the fire today.

In statement (*i*), the connective 'OR' is used in the exclusive sense; that is to say, one or the other possibility exists but not both. In (*ii*) the intended meaning is clearly one or the other or both. The connective 'OR' used in (*ii*) is the 'Inclusive

OR'. In (*iii*) the 'OR' is used for indicating an approximate numbr of people, and it is not used as a connective. From the definition of disjunction it is clear that $\vee$ is 'Inclusive OR'.

NOTES

Table 1.2 *Truth Table for Disjunction*

p	q	$p \vee q$
0	0	0
0	1	1
1	0	1
1	1	1

Negation* (*NOT*)*: If p is a proposition, its negation not p is another proposition called the negation p. The negation of p is denoted by $\sim p$. The proposition $\sim p$ is read 'not p'.

Alternate symbols used in the literature are $\neg p$, $\overline{p}$ and 'NOT p'.

Note that a negation is called a connective although it only modifies a statement. In this sense, negation is the only operator that acts on a single proposition.

Truth table of $\sim p$ is shown in Table 1.3.

Table 1.3 *Truth Table for Negation*

p	$\sim p$
0	1
1	0

Example 1.3: Find the negation of the propositions:

(*i*) It is cold.

(*ii*) Today is Sunday.

(*iii*) Ravi is poor.

Solution: The negation of the propositions are:

(*i*) It is not cold.

(*ii*) Today is not Sunday.

(*iii*) Ravi is not poor.

Conditional Operator (if … then): Let p and q be propositions. The implication $p \to q$ is false when p is true and q is false and true otherwise (Refer Table 1.4). In the implication, p is called the *premise* or hypothesis and q is called the *consequence* or conclusion.

Table 1.4 *Truth Table for If.... Then*

p	q	$p \to q$
0	0	1
0	1	1
1	0	0
1	1	1

Because implications arise in many places in mathematical argument, a wide variety of terminology is used to express $p \rightarrow q$. Some of the more common ways of expressing this implication are:

(*i*) p implies q.

(*ii*) if p, then q.

(*iii*) q if p.

(*iv*) p only if q.

(*v*) p is sufficient for q.

(*vi*) q whenever p.

(*vii*) q is necessary for p.

We shall avoid the word 'implies' since it might be used in different contexts.

Example 1.4: Let p denote 'It is below freezing' and let q denote 'It is snowing'. Write the following statements in a symbolic form:

(*i*) If it is below freezing, it is also snowing.

(*ii*) It is not snowing if it is below freezing.

(*iii*) It is below freezing is a necessary condition for it to be snowing.

Solution: Recall that $p \rightarrow q$ can be read 'if p, then q' or 'p only if q' or 'q is necessary for p'. Then (*i*) $p \rightarrow q$ (*ii*) $p \rightarrow \sim q$ (*iii*) $q \rightarrow p$

Example 1.5: Let p and q be the propositions, where

p : You drive over 80 kms per hour.

q : You get a speeding ticket.

Write the following propositions in symbolic form:

(*i*) You will get a speeding ticket if you drive over 80 kms per hour.

(*ii*) If you do not drive over 80 kms per hour, then you will not get a speeding ticket.

(*iii*) Driving over 80 kms per hour is sufficient for getting a speeding ticket.

(*iv*) Whenever you get a speeding ticket, you are driving over 80 kms per hour.

Solution: (*i*) $p \rightarrow q$ (*ii*) $\sim p \rightarrow \sim q$ (*iii*) $p \rightarrow q$ (*iv*) $q \rightarrow p$

Biconditional Operator (if and only if): Let p and q be propositions. The biconditional $p \leftrightarrow q$ is true when p and q have the same truth values and is false otherwise (Refer Table 1.5).

Note that the biconditional $p \leftrightarrow q$ is true when both the implications $p \rightarrow q$ and $q \rightarrow p$ are true. So 'p if and only if q' is used for biconditional. Other common ways of expressing the proposition $p \leftrightarrow q$ or $p = q$ are 'p is necessary and sufficient for q' and 'If p then q, and conversely'.

Table 1.5 *Truth Table for If and Only If*

p	q	$p \leftrightarrow q$
0	0	1
0	1	0
1	0	0
1	1	1

NOTES

Each of the following theorems is well known, and each can be symbolized in the form $p \leftrightarrow q$:

(*i*) Two lines are parallel if and only if they have the same slope.

(*ii*) Two triangles are congruent if and only if all three sets of corresponding sides are congruent.

Example 1.6: Let p denote 'He is poor' and let q denote 'He is happy'. Write each of the following statement in symbolic form using p and q:

(*i*) To be poor is to be unhappy.

(*ii*) He is rich if and only if he is unhappy.

(*iii*) Being rich is a necessary and sufficient condition to being happy.

Solution:

(*i*) $p \leftrightarrow q$ (*ii*) $\sim p \leftrightarrow q$ (*iii*) $\sim p \leftrightarrow q$

1.2.3 Truth Tables

The truth table of a logical operator specifies how the truth value of a proposition using that operator is determined by the truth values of the propositions. A truth table lists all possible combinations of truth values of the propositions in the left most columns and the truth value of the resulting propositions in the right most column.

Our basic concern is to determine the truth table of a proposition for each possible combination of the truth values of the compound propositions. A table showing all such truth values is called truth table of the formula. In general, if there are n distinct components in a proposition or formula, we need to consider 2^n possible combinations of truth values in order to obtain the truth table.

Two methods of constructing truth table are shown in the following examples.

Example 1.7: Construct the truth table for the statement formula $\sim p \wedge q$.

Solution: It is necessary to consider all possible values of p and q (for the variables, as here, four rows are necessary). These values are entered into first two columns of table for both methods.

Method 1: In this method, the truth values of $\sim p$ are entered in the third column, and the truth values of $\sim p \wedge q$ are entered in the fourth column.

Truth Table *(Method 1)*

p	q	$\sim p$	$\sim p \wedge q$
0	0	1	0
0	1	1	1
1	0	0	0
1	1	0	0

Truth Table *(Method 2)*

p	q	p	$\sim$	$\wedge$	q
0	0	0	1	0	0
0	1	0	1	1	1
1	0	1	0	0	0
1	1	1	0	0	1
Step number		1	2	3	1

Method 2: In this method, a column is drawn for each statement as well as for the connectives that appear. The truth values are entered step by step. The step numbers at the bottom of the table show the sequence followed in arriving at the final step.

Example 1.8: Construct the truth tables for:

(*i*) $\sim (p \wedge q)$ (*ii*) $p \wedge (\sim q)$ (*iii*) $(p \wedge q) \wedge r$

(*iv*) $(p \wedge q) \vee (q \wedge r) \vee (r \wedge p)$ (*v*) $(\sim p) \vee (\sim q)$

Solution:

NOTES

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

Truth Table

p	q	$p \wedge q$	$\sim (p \wedge q)$
0	0	0	1
0	1	0	1
1	0	0	1
1	1	1	0

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

Truth Table

p	q	$\sim q$	$p \wedge (\sim q)$
0	0	1	0
0	1	0	0
1	0	1	1
1	1	0	0

(*iii*) $(p \wedge q) \wedge r$

Truth Table

p	q	r	$p \wedge q$	$(p \wedge q) \wedge r$
0	0	0	0	0
0	0	1	0	0
0	1	0	0	0
0	1	1	0	0
1	0	0	0	0
1	0	1	0	0
1	1	0	1	0
1	1	1	1	1

(*iv*) $(p \wedge q) \vee (q \wedge r) \vee (r \wedge p)$

Truth Table

p	q	r	$p \wedge q$	$q \wedge r$	$r \wedge p$	$(p \wedge q) \vee (q \wedge r)$	$(p \wedge q) \vee (q \wedge r) \vee (r \wedge p)$
0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0
0	1	0	0	0	0	0	0
0	1	1	0	1	0	1	1
1	0	0	0	0	0	0	0
1	0	1	0	0	1	0	1
1	1	0	1	0	0	1	1
1	1	1	1	1	1	1	1

(*v*) $(\sim p) \vee (\sim q)$

Truth Table

p	q	$\sim p$	$\sim q$	$(\sim p) \vee (\sim \mathrm{q})$
0	0	1	1	1
0	1	1	0	1
1	0	0	1	1
1	1	0	0	0

1.2.4 Equivalence Formula

Two propositions are logically equivalent or simply equivalent if they have exactly the same truth values under all circumstances. We can also define this notion as follows:

The propositions p and q are called logically equivalent if $p \leftrightarrow q$ is a tautology. The equivalence of p and q is denoted by $p \Leftrightarrow q$.

Notes:

1. One way to determine whether two propositions are equivalent is to use a truth table. In particular, the propositions p and q are logically equivalent if and only if the columns giving their truth values agree.
2. Whenever we find logically equivalent statements, we can substitute one for another as we wish, since this action will not change the truth value of any statement.

Example 1.9: Show that $\sim (p \wedge q)$ and $\sim p \vee \sim q$ are logically equivalent.

Solution: Construct the truth table of these propositions as shown in Truth Table. Since the truth values are same for all combinations, it follows that these propositions are logically equivalent.

Truth Table

p	q	$\sim p$	$\sim q$	$p \wedge \mathrm{q}$	$\sim (p \wedge q)$	$\sim p \vee \sim q$
0	0	1	1	0	1	1
0	1	1	0	0	1	1
1	0	0	1	0	1	1
1	1	0	0	1	0	0

Example 1.10: Show that two propositions $p \to q$ and $\sim p \vee \sim q$ are logically equivalent.

Solution: Construct the required truth table. Since the truth values of $p \to q$ and $\sim p \vee q$ agree,

$$p \to q \Leftrightarrow \sim p \vee q$$

Truth Table

p	q	$\sim p$	$p \to q$	$\sim p \vee q$
0	0	1	1	1
0	1	1	1	1
1	0	0	0	0
1	1	0	1	1

Table 1.6 contains some important equivalences. In these equivalences, 1 denotes any proposition that is a tautology, and 0 denotes any proposition that is a contradiction. The symbol p, q, r represent arbitrary propositions. Most of the equivalences in this table have straightforward intuitive interpretations and all of them can be verified by constructing truth tables.

Table 1.6 *Logical Equivalences*

Equivalence	Name
1. $p \Leftrightarrow (p \vee p)$	Idempotents of $\vee$
2. $p \Leftrightarrow (p \wedge p)$	Idempotents of $\wedge$
3. $(p \wedge q) \Leftrightarrow (q \vee p)$	Commutativity of $\vee$
4. $(p \wedge q) \Leftrightarrow (q \wedge p)$	Communtativity of $\wedge$
5. $(p \vee q) \vee r \Leftrightarrow p \vee (q \vee r)$	Associativity of $\vee$
6. $(p \wedge q) \wedge r \Leftrightarrow p \wedge (q \wedge r)$	Associativity of $\wedge$
7. $\sim (p \vee q) \Leftrightarrow \sim p \wedge \sim q$	De Morgan's law 1
8. $\sim (p \wedge q) \Leftrightarrow \sim p \vee \sim q$	De Morgan's law 2
9. $p \wedge (q \vee r) \Leftrightarrow (\mathrm{p} \wedge \mathrm{q}) \vee (\mathrm{p} \wedge \mathrm{r})$	Distributive of $\wedge$ over $\vee$
10. $p \vee (q \wedge r) \Leftrightarrow (p \vee q) \wedge (p \vee r)$	distributive of $\vee$ over $\wedge$
11. $p \vee 1 \Leftrightarrow 1$	(Null) or Domination law 1
12. $p \wedge 0 \Leftrightarrow 0$	(Null) or Domination law 2
13. $p \wedge 1 \Leftrightarrow p$	Identity law 1
14. $p \vee 0 \Leftrightarrow p$	Identity law 2
15. $p \vee \sim p \Leftrightarrow 1$	Negation law 1
16. $p \wedge \sim p \Leftrightarrow 0$	Negation law 2
17. $\sim (\sim p) \Leftrightarrow p$	Double negation law (involution)
18. $p \rightarrow q \Leftrightarrow \sim p \vee q$	Implication law
19. $p \leftrightarrow q \Leftrightarrow (p \rightarrow q) \wedge (q \rightarrow p)$	Equivalence law
20. $(p \wedge q) \rightarrow r \Leftrightarrow p \rightarrow (q \rightarrow r)$	Exportation law
21. $(p \rightarrow q) \wedge (p \rightarrow \sim q) \Leftrightarrow \sim p$	Absurdity law
22. $p \rightarrow q \Leftrightarrow \sim q \rightarrow \sim p$	Contrapositive law
23. $p \vee (p \wedge q) \Leftrightarrow p$	Absorption law 1
24. $p \wedge (p \vee q) \Leftrightarrow p$	Absorption law 2
25. $p \leftrightarrow q \Leftrightarrow (p \wedge q) \vee (\sim p \wedge \sim q)$	Biconditional law

NOTES

Example 1.11: Write an equivalent formula for $p \wedge (q \leftrightarrow r) \vee (r \leftrightarrow p)$ which does not contain the biconditional.

Solution: Since $p \leftrightarrow q \Leftrightarrow (p \rightarrow q) \wedge (q \rightarrow p), p \wedge (q \leftrightarrow r) \vee (r \leftrightarrow p) \Leftrightarrow p \wedge ((q \rightarrow r) \wedge (r \rightarrow q)) \vee ((r \rightarrow p) \wedge (p \rightarrow r))$

Example 1.12: Write an equivalent formula for $p \wedge (q \leftrightarrow r)$ which contains neither the biconditional nor the conditional.

Solution: Since $p \leftrightarrow q \Leftrightarrow (p \rightarrow q) \wedge (q \rightarrow p)$ and $p \rightarrow q \Leftrightarrow \sim p \vee q$

$$\begin{aligned} p \wedge (q \leftrightarrow r) &\Leftrightarrow p \wedge ((q \rightarrow r) \wedge (r \rightarrow q)) \\ &\Leftrightarrow p \wedge ((\sim q \vee r) \wedge (\sim r \vee q)) \end{aligned}$$

1.2.5 Tautology

The final column of a truth table of a given formula contains both 1 and 0. There are some formulae whose truth values are always 1 or always 0 regardless of the truth value assignments to the variables. Consider for example, the statement formula $p \vee \sim p$ and $p \wedge \sim p$ in Truth Table 1.7.

Truth Table 1.7 *Tautology and Contradiction*

p	$\sim p$	$p \vee \sim p$	$p \wedge \sim p$
0	1	1	0
1	0	1	0

The truth values of $p \vee \sim p$ and $p \wedge \sim p$, which are 1 and 0 respectively, are independent of the statement by which the variable p may be replaced.

Tautology: A statement formula which is true regardless of the truth values of the statements which replace the variables in it is called a tautology or a logical truth or a universally valid formula.

Contradiction: A statement formula which is false regardless of the truth values of the statements which replace the variables in it is called a contradiction.

Contingency: A statement formula that is neither a tautology nor a contradiction is called a *contingency*.

Note: A straight forward method to determine whether a given formula is a tautology is to construct its truth table . We may say that a statement formula which is a tautology is identically true (their truth tables consist of a column of ones) and a formula which is a contradiction is identically false (their truth tables consist of a column of zeros). Obviously, the negation of a contradiction is a tautology.

Example 1.13: Verify if the following propositions are tautologies:

(*i*) $(p \wedge q) \rightarrow p$

(*ii*) $q \rightarrow (p \vee q)$

(*iii*) $(p \vee q) \leftrightarrow (q \vee p)$

(*iv*) $p \vee \sim (p \wedge q)$

(*v*) $\sim (p \wedge q) \leftrightarrow (\sim p) \vee (\sim q)$

Solution: Construct the truth table of the above given propositions.

Truth Table

p	q	$\sim p$	$\sim q$	$p \vee q$	$p \wedge q$	$q \vee p$	$\sim (p \wedge q)$	(a)	(b)	(c)	(d)	(e)
0	0	1	1	0	0	0	1	1	1	1	1	1
0	1	1	0	1	0	1	1	1	1	1	1	1
1	0	0	1	1	0	1	1	1	1	1	1	1
1	1	0	0	1	1	1	0	1	1	1	1	1

Since the truth value of all propositions is 1, for all values of p and q, the given propositions are tautologies.

Example 1.14: Verify if the proposition $(p \wedge q) \wedge \sim (p \vee q)$ is a contradiction.

Solution:

Truth Table

p	q	$p \wedge q$	$p \vee q$	$\sim (p \vee q)$	$(p \wedge q) \wedge \sim (p \vee q)$
0	0	0	0	1	0
0	1	0	1	0	0
1	0	0	1	0	0
1	1	1	1	0	0

Since the truth value of $(p \wedge q) \wedge \sim (p \vee q)$ is 0 for all values of p and q, the proposition is a contradiction.

Example 1.15: Show that the conjunction of two tautologies is also a tautology.

Solution: Let us denote A and B by two statement formulae which are tautologies. If we assign any truth values to the variables of A and B, then the truth values of both A and B will be 1. Thus, the truth value of $A \wedge B$ will be 1, so that $A \wedge B$ will be a tautology.

Example 1.16: From the formulae given below, indicate if they are tautologies or contradictions.

(*i*) $p \rightarrow (p \vee q)$ (*ii*) $(p \rightarrow \sim p) \rightarrow \sim p$

(*iii*) $(\sim q \wedge p) \wedge q$ (*iv*) $(p \vee q) \wedge (\sim p \wedge \sim q)$

Solution:

Truth Table

p	q	$\sim p$	$\sim q$	$p \vee q$	$\sim p \wedge \sim q$	$p \rightarrow \sim p$	$\sim q \wedge p$	(a)	(b)	(c)	(d)
0	0	1	1	0	1	1	0	1	1	0	0
0	1	1	0	1	0	1	0	1	1	0	0
1	0	0	1	1	0	0	1	1	1	0	0
1	1	0	0	1	0	0	0	1	1	0	0

Since the truth values of (*i*) and (*ii*) are 1 for all values of p and q, (*i*) and (*ii*) are tautologies. Since the truth values of (*iii*) and (*iv*) are 0 for all values of p and q, (*iii*) and (*iv*) are contradictions.

Substitution Instance: A formula A is called a substitution instance of another formula B, if A can be obtained from B by substituting formulae for some variables of B, with the condition that the same formula is substituted for the same variables each time it occurs.

Note: Suppose A (p, q, ….) is a tautology. Then it does not depend upon the particular truth values of its variables p,q,… so we can substitute P for p, Q for q… for any propositions P, Q,… in the tautology A (p,q…) and still have a tautology A(P, Q,…).

Example 1.17: Produce the substitution instance of the following formulae for the given substitutions:

(*i*) $(((p \rightarrow q) \rightarrow p) \rightarrow p)$; substitute $(p \rightarrow q)$ for p and $(p \wedge q \rightarrow r)$ for q.

(*ii*) $((p \rightarrow q) \rightarrow (q \rightarrow p)$; substitute q for p and $(p \wedge \sim p)$ for q.

Solution:

(*i*) Substitute $(p \to q)$ for p and $(p \wedge q \to r)$ for q simultaneously, we get

$$(((p \to q) \to (p \wedge q \to r) \ \ (p \to q) \to (p \to q))$$

(*ii*) Substitute q for p and $(p \wedge {\sim} p)$ for q simultaneously in:

$((p \to q) \to \ \ (q \to p))$, we get $(q \to (p \wedge {\sim} p)) \to ((p \wedge {\sim} p) \to q))$

Note: In constructing substitution instances of a formula, substitutions are made for the simple proposition (without connectives) and never for the compound proposition. For example, $p \to q$ is not a substitution instance of $p \ \ {\sim} r$, because it must be replaced by r and not $\sim r$.

Check Your Progress

1. What is proposition?
2. What is the use of logical operators?
3. Define truth table.
4. When do the two propositions logically equivalent?
5. How can we draw inference?

1.2.6 Inference Theory

To draw inference we must have some rule or a set of rules that serve the basis of inference, otherwise inference will not have sound reasoning. This uses the rules of inference given for the statement calculus along with additional rules needed to deal with formulas with quantifiers. We can draw inference on any given statement with symbols and logical connectives either by truth table or by applying rules of inference that are given in subsequent topic.

If case conclusion has the form of a conditional statement, rule of conditional proof called **CP** is used. For using equivalences and implications, some rules are needed to eliminate quantifiers for such derivation.

Rules of specification, known as rules US **(Universal Specification)** and ES **(Existential Specification)** are used for the purpose of elimination. After eliminating quantifiers the inference is drawn. If the desired conclusion is to be quantified, rules of generalization called rules UG **(Universal Generalization)** and EG **(Existential Generalization)** are used to attach a quantifier.

All these rules are given under the section 'Rules of Inference'.

1.2.7 Validity by Truth Table

We can draw inference on any given statement with symbols and logical connectives either by truth table or by applying rules of inference that are given in subsequent topic.

Two statements are equivalent if they have identical truth values. A logical statement is valid when it is a tautology. To check this, truth tables are constructed.

1.2.8 Rules of Inference

Table 1.8 states the various rules of inference.

NOTES

Table 1.8 *Rules of Inference*

Rules of Inference	Implication Form
Addition I_1	
$\therefore \quad \dfrac{p}{p \vee \varphi}$	$P \Rightarrow p \vee \varphi$
Conjunction I_2	
$\therefore \quad \dfrac{\varphi}{p \wedge \varphi}$	$P \wedge \varphi \Rightarrow p \wedge \varphi$
Simplification I_3	
$\therefore \quad \dfrac{p \wedge \varphi}{p}$	$(p \wedge \varphi) \Rightarrow p$
Modus tollens I_4	
$\neg \varphi$	
$\mathrm{p} \Rightarrow \varphi$	$[\neg \varphi \wedge (p \Rightarrow \varphi)] \Rightarrow \neg \mathrm{p}$
$\therefore \quad \neg \mathrm{p}$	
Disjunctive syllogism I_5	
$\neg \mathrm{p}$	
$\therefore \quad \dfrac{p \vee \varphi}{\varphi}$	$[\neg \ p \wedge (p \vee \varphi)] \Rightarrow \varphi$
Modus ponens	
p	
$\therefore \quad \dfrac{p \Rightarrow \varphi}{\varphi}$	$[p \wedge (p \Rightarrow \varphi)] \Rightarrow \varphi$
Hypothetical syllogism I_7	
$p \Rightarrow \varphi$	
$\therefore \quad \dfrac{\varphi \Rightarrow R}{p \Rightarrow R}$	$[(P \Rightarrow \varphi) \wedge (\varphi \Rightarrow R)] \Rightarrow (P \Rightarrow R)$
Conjunctive dilemma	
$[(P \Rightarrow \varphi) \wedge (R \Rightarrow s)]$	
$\therefore \quad \dfrac{p \vee R}{\varphi \vee s}$	$(p \Rightarrow \varphi) \wedge (R \Rightarrow S) \wedge (p \vee R) \Rightarrow (\varphi \vee S)$
Disjunctive dilemma	
$(p \Rightarrow \varphi) \wedge (R \Rightarrow S)$	
$\neg \varphi \vee \neg s$	
$\therefore \quad p \vee R$	$[(p \Rightarrow \varphi) \wedge (R \Rightarrow S)] \wedge [\neg \varphi \vee \neg S]$ $\Rightarrow (\neg p \vee \neg R)$

The two rules of inference are called rules P and T.

Rule P: A premise may be introduced at any point in the derivation.

Rule T: A formula S may be introduced in a derivation if S is tautologically implied by any one or more of the preceeding formulae in the derivation of a truth value.

Example 1.18: For the given set of arguments check the validity of conclusion:

(*i*) If determinism is true then we have no free will.

(*ii*) If Heisenberg's interpretation of quantum physics is correct, then there are events not necessiated by prior events.

(*iii*) If there are events not necessitated by prior events, then we have free will.

Conclusion: If Heisenberg interpretation of physics is correct then we have free will.

Solution: D : Determinism is true.

Q : We have no free will.

R : Heisenberg interpretation of quantum physics is correct.

S : There are events not necessitated by prior events.

Arguments: $(D \to \varphi)\ (R \to S)(S \to \neg\varphi)$

Conclusion: $R \to \neg\varphi$

(*i*) $R \to S$ Rule P (*ii*) $R \to \neg\varphi$ Rule P

From cases (*i*) and (*ii*), we get :

(*iii*) $R \to \neg\varphi$ rule T and hypothetical syllogism.

1.2.9 Predicate Calculus

Consider the two statements 'Rohit is Brilliant' and 'Manaswine is Brilliant'. As propositions, there is no relation between them, but they have something in common. Instead of writing two statements we can write a single statement like 'x is brilliant', because both Rohit and Manaswine share the same nature brilliant. By replacing x by any other name we get many propositions. The common feature expressed by 'is brilliant' is called predicate. Predicate calculus deals with sentences involving predicates.

A part of a declaritive sentence describing the properties of an object or relation among objects can be referred as predicate, e.g. 'Is brilliant'.

Note: A statement of the form $p(x_1, x_2, ..., x_n)$ is the value of the propositional function P at the nth tuple $(x_1,..., x_n)$ and P is also called a predicate.

For example, Statements involving variables such as $x > 5$, $x = y + 6$, and $x + y = z$.

These statement are neither true nor false when the values of the variables are not specified.

Let us consider the statement $x > 5$.

Here the variable x is the subject of the statement. The second part predicate is greater than 5 and > refers to a property that the subject of the statement can have. Therefore $x > 5$ can be written in the form $p(x)$. The statement $p(x)$ is also said to be the value of the propositional function P at x. Once a value has been assigned to the variable x, the statement can be written as $p(x)$.

Statement Calculus

In logical reasoning, a certain number of propositions is assumed to be true, and based on that assumption, some other propositions are derived.

Hypothesis: The propositions that are assumed to be true. It may also be referred to as premises.

Conclusion: The proposition derived by using the rules of inference.

Valid argument: The process of deriving conclusions based on the assumptions of a premise.

Free and Bound Variables

The variable is said to be bound if it is concerned with either universal ($\forall$) or existential ($\exists$) quantifier and the scope of the variable in the formulae immediately following the quantifier. The variable, which is not concerned with any quantifier, is called free variable.

For example, $\forall x\ [p(x, y)]$ in the statement given above, x is said to be bound and the scope of x is upto $p(x, y)$, while y is called free variable.

Quantifiers

When all the variables in a propositional function are assigned values, the resulting statement has a truth value. However, there is another important way to change propositional functions into propositions, called quantification. It has been broadly classified into two types, namely

(*i*) Universal Quantification
(*ii*) Existential Quantification

Universe of Discourse: Many mathematical statements assert that a property is true for all values of a variable in a particular domain, called the universe of discourse. Such a statement is expressed using an universal quantification.

Universal Quantification: The universal quantification of $p(x)$ is a proposition only when $p(x)$ is true for all values of x in the universe of discourse.

Notation: $\forall x\, p(x) \rightarrow$ universal quantification of $p(x)$

It is also expressed as,

'for all $x\, p(x)$' or 'for every $x\, p(x)$'

Example 1.19: 'Every student in this class has studied logic'.

Solution: Let $p(x)$ denote the statement 'x has studied logic',

$\forall x\ [s(x) \rightarrow p(x)]$

Where $s(x)$ is the statement 'x is in this class'.

Example 1.20: What is the truth value of $\forall x\, p(x)$, where $p(x)$: $x^2 < 10$ and the universe of discourse consists of the positive integers not exceeding 4?

Solution: The statement $\forall x\, p(x)$ is the same as the conjunction. $p(1) \wedge p(2) \wedge p(3) \wedge p(4)$.

Since the universe of discourse consists of the integers 1, 2, 3,and 4, $p(4)$ in the statement '$4^2 < 10$' is false, it follows $\forall x\, p(x)$ is also false.

Reason:

NOTES

$p(1) \wedge p(2) \wedge p(3) \wedge p(4)$

$T \wedge T \wedge T \wedge F =$ false

Existential Quantification: The existential quantification of $p(x)$ is the proposition 'There exists an element x in the universe of discourse such that $p(x)$ is true.'

Notation: $\exists x\ p(x)$

It is also expressed as,

'There is an x such that $p(x)$'

'There is atleast one x such that $p(x)$', or for some $x\, p(x)$.

Example 1.21: Let $p(x) : x > 3$, what is the truth value of the quantification $\exists x\ p(x)$ where the universe of discourse is the set of real numbers.

Solution: Since $x > 3$ is true, for example, when $x = 4$ the existential quantification of $p(x)$ is $\exists x\, p(x)$ is true.

Example 1.22: Write the predicate 'x is the father of the mother of y'.

Solution: Let $p(x) : x$ is a person.

$p(x, z) : x$ is the father of z.

$m(z, y) : z$ is the mother of y.

We assume that there exists a person z such that x is the father of z and z is the mother of y.

Universal Specification (US)

For a given predicate $(\forall x)\ [p(x)]$ one can conclude $p(b)$.

Existential Specification (ES): From $(\exists x)$ [p(x)] one can conclude the value of $p(b)$ provided that b is not free in any given premise and also not free in any prior step of the derivation. These requirements can easily be met by choosing a new variable each time *ES* is used.

Existential Generalization (EG): From $p\ (b)$ one can conclude $(\exists x)\ [p(x)]$.

Universal Generalization (UG): From $p\ (b)$ one can conclude $\forall x\ [p(x)]$ provided that b is not free in any of the given premises and provided that if b is free in the prior step which resulted from the use of *ES*, then no variables introduced by that use of *ES* appear free in $p(b)$.

Example 1.23: Some cats are black but all buffaloes are black.

Solution: $C(x) : x$ is a cat.

$B(y) : y$ is a buffalo.

$b(x) : x$ is black.

Thus, $(\exists x)\ (\forall y) [C\,(x) \wedge b(x)] \wedge [B(y) \rightarrow b(y)]$

Example 1.24: Sum of two positive integers is greater than either of the integers.

Solution: $I(x)$: x is a positive integer.

$GT(x, y)$: x is greater than y.

$Su(x, y)$: Sum of x and y.

Thus, $(\forall x)\ (\forall y)\ [(I(x) \wedge I(y))] \rightarrow [GT(Su(x, y), x] \vee GT\ [Su(x, y), y]$

Example 1.25: Every student in this school is either good at studies or good in sports.

Solution: $S(x)$: x is a student of this school.

$ST(x)$: x is good at study.

$SP(x)$: x is good at sports.

Thus, $(\forall x)\ [S(x) \rightarrow (ST(x) \vee SP(x)]$

Quantifiers are distributive over the predicate and negation of universe quantifier is existence quantifier and vice versa. This is being state below.

(*i*) $(\exists x)[A(x) \vee B(x)] \Leftrightarrow (\exists x)\ \{A(x)\} \vee (\exists x)\{B(x)\}$

(*ii*) $(\forall x)[A(x) \wedge B(x)] \Leftrightarrow (\forall x) \wedge [A(x)] \wedge (\forall x)\ [B(x)]$

(*iii*) $\neg\ (\exists x)\ A(x) \Leftrightarrow \forall x \neg A(x)$

(*iv*) $\neg\ (\forall x)\ A(x) \Leftrightarrow (\exists x)\ \neg A(x)$

1.3 SEMI-GROUPS AND MONOIDS

Groups occupy a very important place in the study of abstract algebra.

Definition: A non-empty set group (G), together with a binary composition $*$ (star) is said to form a group, if it satisfies the following postulates.

- Associativity: $a * (b * c) = (a * b) * c$, for all $a, b, c \in G$
- Existence of Identity: $\exists$ an element $e \in G$, such that,

  $a * e = e * a = a$ for all $a \in G$

  (e is then called *identity*)
- Existence of Inverse: For every $a \in G$, $\exists\ a' \in G$ (depending upon a), such that,

  $a * a' = a' * a = e$

  (a' is then called inverse of a)

Notes:

1. Since $*$ is a binary composition on G, it is understood that for all $a, b \in G$, $a * b$ is a unique member of G. This property is called closure property.
2. If, in addition to the above postulates, G also satisfies the commutative law

   $a * b = b * a$ for all $a, b \in G$

   then G is called an abelian group or a commutative group.
3. Generally, the binary composition for a group is denoted by '.' (dot), which is so convenient to write (and makes the axioms look so natural too).

This binary composition '.' is called product or multiplication (although it may have nothing to do with the usual multiplication, that you are so familiar with). In fact, you even drop '.' and simply write ab in place of $a\ .\ b$.

If the set G is finite (i.e., has finite number of elements) it is called a finite group; otherwise, it is called an infinite group.

The symbols e is used for identity of a group and a^{-1} for the inverse of element a of the group.

Definition: By the order of a group, you will mean the number of elements in the group and shall denote it by $o(G)$ or $| G |$.

The following are a few points of systems that form groups or do not form groups:

1. The set **Z** of integers forms an abelian group with respect to the usual addition of integers.
2. In the sets **Q** of rationals, **R** of real numbers would also form abelian groups with respect to addition.
3. Set of integers, with respect to usual multiplication does not form a group, although closure, associativity and identity conditions hold.
   Note 2 has no inverse with respect to multiplication as there does not exist any integer a such that, $2 . a = a . 2 = 1$.
4. The set G of all +ve irrational numbers together with 1 under multiplication does not form a group as closure does not hold. Indeed $\sqrt{3}.\sqrt{3} = 3 \notin$ G, although you would notice that other conditions in the definition of a group are satisfied here.
5. Let G be the set $\{1, -1\}$. Then it forms an abelian group under multiplication. It is again easy to check the properties.
   I would be identity and each element is its own inverse.
6. Set of all 2×2 matrices over integers under matrix addition would be another example of an abelian group.
7. Set of all non zero complex numbers forms a group under multiplication defined by,
   $$(a + ib)\ (c + id) = (ac - bd) + i\ (ad + bc)$$
   $1 = 1 + i.0$ will be identity, $\dfrac{a}{a^2+b^2} - i\dfrac{b}{a^2+b^2}$ will be inverse of $a + ib$.
   Note: $a + ib$ non zero means that not both a and b are zero. Thus, $a^2 + b^2 \neq 0$.
8. The set G of all nth roots of unity, where n is a fixed positive integer, forms an abelian group under usual multiplication of complex numbers.
   The complex number z is an nth root of unity if $z^n = 1$ and also that there exist exactly n distinct roots of unity.
   In fact, the roots are given by,
   $$e\frac{2\pi ir}{n}, \text{ where } r = 1, 2, ..., n \text{ and } e^{ix} = \cos x + i \sin x.$$

If $a, b \in G$ be any two members, then $a^n = 1$, $b^n = 1$ thus $(ab)^n = a^n$ $b^n = 1$.

$\Rightarrow$ ab is an nth root of unity.

$\Rightarrow$ $ab \in G \Rightarrow$ Closure holds.

Associativity of multiplication is true in complex numbers.

Again, since $1 . a = a . 1 = a$, 1 will be identity.

Also for any $a \in G$, $\frac{1}{a}$ will be its inverse as $\left(\frac{1}{a}\right)^n = \frac{1}{a^n} = 1$.

So, inverse of $e\frac{2\pi ir}{n}$ is $e^{2\pi i(n-r)/n}$ and identity is $e\frac{2\pi io}{n} = 1$.

Commutativity is must hence G is an abelian group.

As a particular case, if $n = 4$ then G is $\{1, -1, i, -i\}$.

9. (*i*) Let $G = \{\pm 1, \pm i, \pm j, \pm k\}$. Define product on G by usual multiplication together with,

$$\begin{aligned} i^2 = j^2 = k^2 = -1, \quad & ij = -ji = k \\ & jk = -kj = i \\ & ki = -ik = j \end{aligned}$$

Then G forms a group. G is not abelian as $ij \neq ji$.

This is called the Quaternion Group.

(*ii*) If set G consists of the eight matrices:

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix}, \begin{bmatrix} i & 0 \\ 0 & -i \end{bmatrix}, \begin{bmatrix} -i & 0 \\ 0 & i \end{bmatrix}, \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}, \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix},$$

$$\begin{bmatrix} 0 & i \\ i & 0 \end{bmatrix}, \begin{bmatrix} 0 & -i \\ -i & 0 \end{bmatrix}, \text{ where } i = \sqrt{-1}$$

Then G forms a non-abelian group under matrix multiplication.

10. Let $G = \{(a, b) \mid a, b \text{ rationals}, a \neq 0\}$. Define $*$ on G by

$$(a, b) * (c, d) = (ac, ad + b)$$

Closure follows as $a, c \neq 0 \Rightarrow ac \neq 0$

$$\begin{aligned} [(a, b) * (c, d)] * (e, f) &= (ac, ad + b) * (e, f) \\ &= (ace, acf + ad + b) \\ (a, b) * [(c, d) * (e, f)] &= (a, b) * (ce, cf + d) \\ &= (ace, acf + ad + b) \end{aligned}$$

This proves associativity.

$(1, 0)$ will be identity and $(1/a, -b/a)$ will be inverse of any element (a, b).

G is not abelian as,

$$(1, 2) * (3, 4) = (3, 4 + 2) = (3, 6)$$
$$(3, 4) * (1, 2) = (3, 6 + 4) = (3, 10).$$

11. (*i*) The set G of all 2×2 matrices of the form $\begin{bmatrix} a & b \\ c & d \end{bmatrix}$ over reals, where $ad - bc \neq 0$, forms a non-abelian group under matrix multiplication. It is called the general linear group of 2×2 matrices over reals and is denoted by $GL(2, \mathbf{R})$.

    The matrix $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ will act as identity and

    the matrix $\begin{bmatrix} \dfrac{d}{ad-bc} & \dfrac{-b}{ad-bc} \\ \dfrac{-c}{ad-bc} & \dfrac{a}{ad-bc} \end{bmatrix}$ will be inverse of $\begin{bmatrix} a & b \\ c & d \end{bmatrix}$.

    (*ii*) If G be the set of all $n \times n$ invertible matrices over reals, then G forms a group under matrix multiplication.

12. Let $G = \{2^r \mid r = 0, \pm 1, \pm 2, ...\}$

    It can be shown that G forms a group under usual multiplication.

    For any $2^r, 2^s \in G, 2^r. 2^s = 2^{r+s} \in G$

    Thus closure holds. Associativity is obvious.

    Again as $1 \in G$, and $x \,.\, 1 = 1 \,.\, x = x$ for all $x \in G$, 1 is identity.

    For any $2^r \in G$, as $2^{-r} \in G$ and $2^r. 2^{-r} = 2^0 = 1$. Here each element of G has inverse. Commutativity is evidently true.

13. **Group of Residues:** Let $G = \{0, 1, 2, 3, 4\}$. Define a composition $\oplus_5$ on G by $a \oplus_5 b = c$ where c is the least non –ve integer obtained as remainder when $a + b$ is divided by 5. For example, $3 \oplus_5 4 = 2$, $3 \oplus_5 1 = 4$, etc. Then $\oplus_5$ is a binary composition on G (called addition modulo 5). It is easy to verify that G forms a group under this.

    You can generalize this result to,

    $G = \{0, 1, 2, ..., n - 1\}$

    under addition modulo n where n is any positive integer.

    Here,

    $$a \oplus_n b = \begin{cases} a + b & \text{if } a + b < n \\ a + b - n & \text{if } a + b \geq n \end{cases}$$

    Drop the sub suffix n and simply write $\oplus$. This group is generally denoted by $\mathbf{Z}_n$.

14. Let $G = \{x \in \mathbf{Z} \mid 1 \leq x < n, x, n \text{ being co-prime}\}$ where $\mathbf{Z}$ = Set of integers and x, n being co-prime means H.C.F of x and n is 1.

    You define a binary composition $\otimes$ on G by $a \otimes b = c$ where c is the least +ve remainder obtained when $a \,.\, b$ is divided by n. This composition $\otimes$ is called multiplication modulo n.

    Consequently, G forms a group under $\otimes$.

    Closure: For $a, b \in G$, let $a \otimes b = c$. Then $c \neq 0$, because otherwise $n \mid ab$ which is not possible as a, n and b, n are co-prime.

Thus $c \neq 0$ and also then $1 \leq c < n$.

Now if c, n are not co-prime then $\exists$ some prime number p such that, $p|c$ and $p|n$.

Again, as $ab = nq + c$ for some q.

We obtain $p|ab$ $\quad [p|n \Rightarrow p|nq, p|c \Rightarrow p|nq + c]$

$\Rightarrow p|a$ or $p|b$ (as p is prime)

If $p|a$ then as $p|n$ it means a, n are not co-prime.

But a, n are co-prime. Similarly, $p|b$ leads to a contradiction.

Hence, c, n are co-prime and thus $c \in G$, showing that closure holds.

Associativity: Let $a, b, c \in G$ be any elements.

Let $a \otimes b = r_1$, $(a \otimes b) \otimes c = r_1 \otimes c = r_2$ then r_2 is given by $r_1c = nq_2 + r_2$

Also $a \otimes b = r_1$ means $\quad ab = q_1n + r_1$

Thus, $ab - q_1n = r_1 \quad \Rightarrow (ab - q_1n)c = r_1c = nq_2 + r_2$

$$\Rightarrow (ab)c = r_2 + nq_2 + nq_1c = n(q_1c + q_2) + r_2$$

or that r_2 is the least non-negative remainder got by dividing $(ab)c$ by n.

Similarly, if $a \otimes (b \otimes c) = r_3$ then you can show that r_3 is the least non –ve remainder obtained by dividing $a(bc)$ by n.

But since $a(bc) = (ab)c$, $r_2 = r_3$. Hence $a \otimes (b \otimes c) = (a \otimes b) \otimes c$.

Existence of Identity: It is observed that

$a \otimes 1 = 1 \otimes a = a$ $\quad$ for all $a \in G$ or that 1 acts as identity.

Existence of Inverse: Let $a \in G$ be any element then a and n are co-prime and thus you can find integers x and y such that, $ax + ny = 1$.

By division algorithm,

$$x = qn + r, \quad \text{where } 0 \leq r < n$$
$$\Rightarrow ax = aqn + ar$$
$$\Rightarrow ax + ny = aqn + ar + ny$$
$$\Rightarrow 1 = aqn + ar + ny$$

Or that $\quad ar = 1 + (-aq - y)n$

i.e., $a \otimes r = 1$. Similarly $r \otimes a = 1$. If r, n are co-prime, r will be inverse of a.

If r, n are not co-prime, you can find a prime number p such that, $p \mid r$, $p \mid n$

$$\Rightarrow p \mid qn \text{ and } p \mid r \quad \Rightarrow p \mid qn + r \quad \Rightarrow p \mid x$$
$$\Rightarrow p \mid ax \text{ also } p \mid ny \quad \Rightarrow p \mid ax + ny = 1$$

Which is not possible. Thus, r, n are co-prime and so $r \in G$ and is the required inverse of a.

This proves that G will be abelian. You denote this group by U_n or $U(n)$ and call it the group of integers under multiplication modulo n.

Note: Suppose $n = p$, a prime, then since all the integers 1, 2, 3, ..., $p - 1$ are co-prime to p, these will all be members of G. This can be represented as,

$$G = \{2, 4, 6, ..., 2(p - 1)\}$$

Where $p > 2$ is a prime and forms an abelian group under multiplication modulo $2p$.

15. Let $G = \{0, 1, 2\}$ and define $*$ on G by

    $$a * b = | a - b |$$

    Then closure is established by taking a look at the composition table

$*$	0	1	2
0	0	1	2
1	1	0	1
2	2	1	0

   Since $a * 0 = | a - 0 | = a = 0 * a$, 0 is identity and $a * a = | a - a | = 0$ shows each element will be its own inverse.

   But the system $(G, *)$ fails to be a group as associativity does not hold.

   Indeed $1 * (1 * 2) = 1 * 1 = 0$

   But $(1 * 1) * 2 = 0 * 2 = 2$

16. Let $S = \{1, 2, 3\}$ and let $S_3 = A(S)$ = Set all permutations of S. This set satisfies associativity, existence of identity and existence of inverse conditions in the definition of a group. Also clearly, since f, g permutations on S imply that fog is a permutation on S the closure property is ensured. Hence, S_3 forms a group. That it is not abelian follows by the fact that $fog \neq gof$. This would, in fact, be the smallest non-abelian group and we shall have an occasion to talk about this group again under the section on permutation groups.

    Note: Let X be a non-empty set and let $M(X)$ = Set of *all* maps from X to X, then $A(X) \subseteq M(X)$. $M(X)$ forms a semi-group under composition of maps. Identity map also lies in $M(X)$ and as a map is invertible iff it is 1–1, onto, i.e., a permutation, we find $A(X)$ the subset of all permutations forms a group, denoted by S_X and is called symmetric group of X. If X is finite with say, n elements then $o(M(X)) = n^n$ and $o(S_X) = \lfloor\underline{n}$ and in that case, we use the notation S_n for S_X.

    In the definition of a group, only the existence of identity and inverse of each element is considered. This can be shown that these elements are unique and provide an elementary but exceedingly useful result.

Lemma: In a group G,

1. Identity element is unique.
2. Inverse of each $a \in G$ is unique.
3. $(a^{-1})^{-1} = a$, for all $a \in G$, where a^{-1} stands for inverse of a.
4. $(ab)^{-1} = b^{-1} a^{-1}$ for all $a, b \in G$.
5. $ab = ac \Rightarrow b = c$.

   $ba = ca \Rightarrow b = c$ for all $a, b, c \in G$.

   These are called the cancellation laws.

Proof:

1. Suppose e and e' are two elements of G which act as identity.

   Then, since $e \in G$ and e' is identity, $e'e = ee' = e$

   and as $e' \in G$ and e is identity, $e'e = ee' = e'$

   The two $\Rightarrow e = e'$, which establishes the uniqueness of identity in a group.

2. Let $a \in G$ be any element and let a' and a'' be two inverse elements of a, then

   $$aa' = a'a = e \text{ and } aa'' = a''a = e$$

   Now $a' = a'e = a'(aa'') = (a'a)a'' = ea'' = a''$.

   Showing thereby that the inverse of an element is unique. We can denote inverse of a by a^{-1}.

3. Since a^{-1} is inverse of a

   $$aa^{-1} = a^{-1}a = e$$

   which also implies a is inverse of a^{-1}. Thus $(a^{-1})^{-1} = a$.

4. We can prove that ab is inverse of $b^{-1}a^{-1}$ for which we show

   $$(ab)\,(b^{-1}a^{-1}) = (b^{-1}a^{-1})\,(ab) = e.$$

   Now
   $$\begin{aligned}(ab)\,(b^{-1}a^{-1}) &= [(ab)\,b^{-1}]\,a^{-1}\\ &= [(a(bb^{-1})]\,a^{-1}\\ &= (ae)\,a^{-1} = aa^{-1} = e\end{aligned}$$

   Similarly $(b^{-1}a^{-1})\,(ab) = e$ and thus the result follows.

5. Let $ab = ac$, then

   $$\begin{aligned}b &= eb = (a^{-1}a)b\\ &= a^{-1}(ab) = a^{-1}\,(ac)\\ &= (a^{-1}\,a)c = ec = c\end{aligned}$$

   Thus, $ab = ac \Rightarrow b = c$

   This is called the left cancellation law.

   Similarly, the right cancellation law can be proved.

17. (*i*) Let $X = \{1, 2, 3\}$ and let $S_3 = A(X)$ be the group of all permutations on X. Consider $f, g, h \in A(X)$, defined by,

    $$\begin{array}{lll} f(1) = 2, & f(2) = 3, & f(3) = 1\\ g(1) = 2, & g(2) = 1, & g(3) = 3\\ h(1) = 3, & h(2) = 1, & h(3) = 2\end{array}$$

    It is easy then to verify that $fog = goh$

    But $f \neq h$.

    (*ii*) If $(1, 2) * (3, 4) = (3, 6) = (3, 0) * (1, 2)$ and $(3, 4) \neq (3, 0)$

    Hence, the cross cancellations may not hold in a group.

Definition: A non-empty set G together with a binary composition '.' is called a ***semi-group*** if,

$$a\,.\,(b\,.\,c) = (a\,.\,b)\,.\,c \text{ for all } a, b, c \in G$$

This holds that every group is a semi-group. That the converse is not true follows by considering the set **N** of natural numbers under addition.

The set G in point 15 is not a semi-group.

Theorem 1.1: Cancellation laws may not hold in a semi-group.

Proof: Consider M the set of all 2 × 2 matrices over integers under matrix multiplication, which forms a semi-group.

If we take $A = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}, B = \begin{bmatrix} 0 & 0 \\ 0 & 2 \end{bmatrix}, C = \begin{bmatrix} 0 & 0 \\ 3 & 0 \end{bmatrix}$

Then clearly$AB = AC = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$

But $B \neq C$.

Set of natural numbers under addition is an example of a semi-group in which cancellation laws hold.

Theorem 1.2: A finite semi-group in which cancellation laws hold is a group.

Proof: Let $G = \{a_1, a_2, ..., a_n\}$ be a finite semi-group in which cancellation laws hold.

Let $a \in G$ be any element, then by closure property

$aa_1, aa_2, ..., aa_n$ are all in G.

Suppose any two of these elements are equal

say, $aa_i = aa_j$ for some $i \neq j$

Then $a_i = a_j$ by cancellation

But $a_i \neq a_j$ as $i \neq j$

Hence, no two of $aa_1, aa_2, ..., aa_n$ can be equal.

These being n in number, will be distinct members of G [Note $o(G) = n$].

Thus, if $b \in G$ be any element then $b = aa_i$ for some i

i.e., for $a, b \in G$ the equation $ax = b$ has a solution ($x = a_i$) in G.

Similarly, the equation $ya = b$ will have a solution in G.

G being a semi-group, associativity holds in G.

Hence, G is a group.

Note: The above theorem holds only in finite groups. The semi-group of natural numbers under addition being an example where cancellation laws hold but which is not a group.

Theorem 1.3: A finite semi-group is a group if and only if it satisfies cancellation laws.

Proof: Follows by Theorem 1.2.

Definition: A non-empty set G together with a binary composition '.' is said to form a *monoid* if

(*i*) $a(bc) = (ab)c \;\; \forall\, a, b, c \in G$

(*ii*) $\exists$ an element $e \in G$ such that, $ae = ea = a \;\; \forall\, a \in G$

e is then called identity of G. It is easy to see that e is unique.

So all groups are monoids and all monoids are semi groups.

When we defined a group, we insisted that $\exists$ an element e which acts both as a right as well as a left identity and each element has both sided inverse. We now show that it is not really essential and only one sided identity and the *same* sided inverse for each element could also make the system a group.

Theorem 1.4: A system $< G, . >$ forms a group if and only if

(*i*) $a(bc) = (ab)c$ for all $a, b, c \in G$

(*ii*) $\exists\ e \in G$, *such that*, $ae = a$ *for all* $a \in G$

(*iii*) for all $a \in G, \exists\ a' \in G$, such that, $aa' = e$.

Proof: If G is a group, we have nothing to prove as the result follows by definition. Conversely, let the given conditions hold.

All we need show is that $ea = a$ for all $a \in G$

And $a'a = a$ for any $a \in G$

Let $a \in G$ be any element.

By (*iii*) $\exists\ a' \in G$, such that, $aa' = e$

$\therefore$ For $a' \in G,\ \exists\ a'' \in G$, such that, $a'a'' = e$ [using (*iii*)]

Now, $a'a = a'(ae) = (a'a)e = (a'a)(a'a'')$

$= a'(aa')a'' = a'(e)a'' = (a'e)a'' = a'a'' = e.$

Thus, for any $a \in G,\ \exists\ a' \in G$, such that, $aa' = a'a = e$

Again, $ea = (aa')a = a(a'a) = ae = a$

$\therefore$ $ae = ea = a$ for all $a \in G$

i.e., e is identity of G.

Hence, G is a group.

Theorem 1.5: A system $< G, . >$ forms a group if and only if

(*i*) $a(bc) = (ab)c$ for all $a, b, c \in G$

(*ii*) $\exists\ e \in G$, such that, $ea = a$ *for all* $a \in G$

(*iii*) for all $a \in G, \exists$ some $a' \in G$, such that, $a'a = e$.

A natural question to crop up at this stage would be what happens, when one sided identity and the other sided inverse exists. Would such a system also form a group?

18. Let G be a finite set having at least two elements. Define '.' on G by $ab = b$ for all $a, b \in G$ then clearly associativity holds in G.

Let $e \in G$ by any fixed element.

Then as $ea = a$ for all $a \in G$, e will act as left identity.

Again $a \,.\, e = e$ for all $a \in G$

$\Rightarrow$ e is right inverse for any element $a \in G$.

But G is not a group (cancellation laws do not hold in it).

Hence, for a system $< G, . >$ to form a group it is essential that the same sided identity and inverse exist.

Notation: Let G be a group with binary composition '.'. If $a \in G$ be any element then by closure property $a \,.\, a \in G$. Similarly, $(a \,.\, a) \,.\, a \in G$, and so on.

It can be denoted as $a \,.\, a$ by a^2 and $a \,.\, (a \,.\, a)$ or $(a \,.\, a).\ a$ by a^3, and so on. Again $a^{-1}.\ a^{-1}$ would be denoted by a^{-2}. And since $a \,.\, a^{-1} = e$, it can be denoted as $e = a^0$. The notation states,

$$a^m \,.\, a^n = a^{m+n}$$

$$(a^m)^n = a^{mn}$$

Where m, n are integers.

In case the binary composition of the group is denoted by +, it describes sums and multiples in place of products and powers. Thus, here $2a = a + a$, and $na = a + a + ... + a$ (n times), if n is a +ve integer. In case n is –ve integer then $n = -m$, where m is +ve and we define $na = -ma = (-a) + (-a) + ... + (-a)$ m times.

Example 1.26: If G is a finite group of order n then show that for any $a \in G$, $\exists$ some positive integer r, $1 \le r \le n$, such that, $a^r = e$.

Solution: Since $o(G) = n$, G has n elements.

Let $a \in G$ be any element. By closure property a^2, a^3, ... all belong to G.

Consider, $\quad e, a, a^2, ..., a^n$

These are $n + 1$ elements (all in G). But G contains only n elements.

$\Rightarrow$ at least two of these elements are equal. If any of $a, a^2, ..., a^n$ equals e, our result is proved. If not, then $a^i = a^j$ for some i, j, $1 \le i, j \le n$. Without any loss of generality, we can take $i > j$.

Then, $\quad a^i = a^j$

$$\Rightarrow \; a^i \,.\, a^{-j} = a^j \,.\, a^{-j}$$

$$\Rightarrow \; a^{i-j} = e, \quad \text{where } 1 \le i - j \le n.$$

Putting $i - j = r$ gives us the required result.

Example 1.27: Show that a finite semi-group in which cross cancellation holds is an abelian group.

Solution: Let G be the given finite semi-group. Let $a, b \in G$ be any elements. Since G is a semi-group, by associativity

$$a(ba) = (ab)a$$

By cross cancellation then $ba = ab \;\Rightarrow\; G$ is abelian.

Since G is abelian, cross cancellation laws become the cancellation laws. Hence, G is a finite semi-group in which cancellation laws hold.

Thus, G is a group.

Subgroups

We have seen that **R**, the set of real numbers, forms a group under addition, and **Z**, the set of integers, also forms a group under addition. Also **Z** is a subset of **R**. It is one of the many situations which prompts us to make the following definition:

Definition: A non-empty subset H of a group G is said to be a subgroup of G, if H forms a group under the binary composition of G.

Obviously, if H is a subgroup of G and K is a subgroup of H, then K is subgroup of G.

Abelianization of a Group and its Universal Properties

The abelianization of a group G is defined in the following equivalent ways:

1. It is the quotient of the group by its commutator subgroup, i.e., it is the group $G / [G, G]$.
2. It is the quotient of G by the relation $xy = yx$.
3. It is an abelian group A such that there exists a surjective homomorphism $f: G \to A$ with the property that whenever $\varphi : G \to H$ is a homomorphism and H is an abelian group, there is a unique homomorphism $\psi: A \to H$ such that $\varphi = \psi \text{ o } f$.

All of the aforementioned properties can be explained as follows:

In abstract algebra, the commutator subgroup of a group is the subgroup generated by all the commutators of the group.

For elements g and h of a group G, the commutator of g and h is $[g, h]: = g^{-1}h^{-1}gh$. The commutator $[g, h]$ is equal to the identity element e if and only if $gh = hg$, i.e., if and only if g and h commute. In general, $gh = hg[g, h]$.

Abelianization as a homomorphism is the quotient map $G \to G/[G, G]$, where the kernel, $[G, G]$, is the commutator subgroup of G. It can also be defined as a homomorphism $f : G \to A$ to an abelian group A with the property that whenever $\varphi: G \to H$ is a homomorphism and H is an abelian group, there is a unique homomorphism $\psi: A \to H$ such that $\varphi = \psi \text{o} f$.

Given a group G, a factor group G/N is abelian if and only if $[G, G] \leq N$. The quotient $G / [G, G]$ is an abelian group called the abelianization of G. It is usually denoted by G^{ab} or G_{ab}.

Let $\phi: G \to G^{ab}$ Then φ is universal for homomorphisms from G to an abelian group H and for homomorphism of groups $f: G \to H$ there exists a unique homomorphism $F : G^{ab} \to H$ such that $f = F \text{ o } \phi$. This shows the uniqueness of the abelianization G^{ab} up to canonical isomorphism; whereas, the explicit construction $G \to G/[G, G]$ shows existence.

Congruences

Let $a,\ b,\ c,\ (c > 0)$ be integers. We say a is congruent to b modulo c if c divides $a - b$ and we write this as $a \equiv b \pmod c$.This relation '≡' on the set of integers is an equivalence relation as seen earlier.

Addition, subtraction and multiplication in congruences behave naturally.

Let $\quad a \equiv b \pmod c$

$$\begin{aligned} a_1 \equiv b_1 \pmod c &\Rightarrow c \mid a - b,\ c \mid a_1 - b_1 \\ &\Rightarrow c \mid (a + a_1) - (b + b_1) \\ &\Rightarrow a + a_1 \equiv b + b_1 \pmod c \end{aligned}$$

Similarly, $a - a_1 \equiv b - b_1 \pmod{c}$

Also, $c \mid a - b,\ \ c \mid a_1 - b_1$

NOTES

$\Rightarrow\ c \mid aa_1 - ba_1,\ \ c \mid ba_1 - bb_1$

$\Rightarrow\ c \mid (aa_1 - ba_1) + (ba_1 - bb_1)$

$\Rightarrow\ c \mid aa_1 - bb_1$

$\Rightarrow\ aa_1 \equiv bb_1 \pmod{c}$

We may, however, not be able to achieve the above result in case of division. Indeed $\dfrac{a}{a_1}$ or $\dfrac{b}{b_1}$ may not even be integers.

Again, cancellation in congruences in general may not hold.

i.e., $ad \equiv bd$ (mod c) need not essentially imply

$a \equiv b \pmod{c}$

For example, $2.2 \equiv 2.1 \pmod{2}$

But $2 \not\equiv 1 \pmod{2}$

However, cancellation holds if g.c.d.$(d,\ c) = 1$.

i.e., if $ad \not\equiv bd \pmod{c}$

And g.c.d.$(d, c) = 1$

Then $a \equiv b \pmod{c}$.

Proof: $ad \equiv bd \pmod{c}$

$\Rightarrow\ c \mid ad - bd$

$\Rightarrow\ c \mid d\,(a - b)$

$\Rightarrow\ c \mid a - b$ as g.c.d.$(c, d) = 1$

$\Rightarrow\ a \equiv b \pmod{c}$.

Example 1.28: If $a \equiv b$ (mod n), prove that g.c.d.$(a, n) = (b, n)$.

Solution: Let $d =$ g.c.d.(a, n)

Then $d \mid a,\ \ d \mid n$. But $n \mid a - b$

$\therefore$ $d \mid a - b,\ \ d \mid a$

$\Rightarrow\ d \mid a - (a - b) = b$

$\therefore$ $d \mid b,\ \ d \mid n$

Let $c \mid b,\ \ c \mid n$ $\quad\Rightarrow\ c \mid b,\ \ c \mid a - b$ as $n \mid a - b$

$\Rightarrow\ c \mid a - b + b = a$

$\Rightarrow\ c \mid a,\ \ c \mid n$

$\Rightarrow\ c \mid d$ as $d =$ g.c.d.$(a,\ n)$

$\Rightarrow$ g.c.d.$(b,\ n) = d$

Exammple 1.29: Establish that if a is an odd integer, then

$$a^{2^n} \equiv 1 \ (\text{mod } 2^{n+2}) \text{ for any } n \geq 1.$$

Solution: We prove the result by induction on n.

Let $n = 1$.

Then $\quad a^{2^n} = a^2$

And $\quad 2^{n+2} = 2^3 = 8$

Let $a = 2k + 1$.

Then $\quad a^2 = 4k^2 + 4k + 1$

$\quad = 4k\,(k + 1) + 1$

$\therefore \quad a^2 - 1 = 4k\,(k + 1)$

$\quad$ = Multiple of 8 as either k is even or $k + 1$ is even.

$\therefore \quad a^2 \equiv 1 \pmod 8$

So, result is true for $n = 1$.

Assume that the result is true for $n = k$.

Then $\quad a^{2^k} \equiv 1 \pmod{2^{k+2}}$

Now $\quad a^{2^{k+1}} - 1 = (a^{2^k})^2 - 1$

$\quad = (a^{2^k} - 1)\,(a^{2^k} + 1)$

$\quad$ = (multiple of 2^{k+2}) $(a^{2^k} + 1)$ by induction hypothesis.

But $\quad a = \text{odd} \Rightarrow a^{2^k} = \text{odd} \Rightarrow a^{2^k} + 1 = \text{even}$

$\therefore \quad a^{2^{k+1}} - 1 =$ multiple of 2^{k+3}

$\therefore \quad a^{2^{k+1}} \equiv 1 \pmod{2^{k+3}}$

So, result is true for $n = k + 1$.

By induction, result is true for all $n \geq 1$.

1.3.1 Homomorphisms

In this section we will discuss about an isomorphism which can also be termed as an 'Indirect' equality in algebraic systems. Indeed, if two systems have the same number of elements and *behave* exactly in the same manner, nothing much is lost in calling them equal, although at times the idea of equality may look little uncomfortable, especially in case of infinite sets.

Definition: Let $< G, * >$ and $< G', o >$ be two groups.

A mapping $f : G \rightarrow G'$ is called a homomorphism if,

$$f(a * b) = f(a)\ o\ f(b) \quad a, b \in G$$

We can use the same symbol '.' for both binary compositions.

With that as notation we find a map

$f : G \rightarrow G'$ is a homomorphism if,

$$f(ab) = f(a) f(b)$$

If, in addition, f happens to be one-one, onto, we say f is an *isomorphism* and in that case write $G \cong G'$.

Also clearly then,

$$f(a_1 a_2 \ldots\ldots a_n) = f(a_1) f(a_2) \ldots\ldots f(a_n)$$

holds under an isomorphism (homomorphism)

An onto homomorphism is called *epimorphism.*

A one-one homomorphism is called *monomorphism.*

A homomorphism from a group G to itself is called an *endomorphism* of G.

NOTES

An isomorphism from a group G to itself is called *automorphism* of G.

If $f: G \to G'$ is onto homomorphism, then G' is called *homomorphic image* of G.

Let $< \mathbf{Z}, + >$ and $< \mathbf{E}, + >$ be the groups of integers and even integers.

Define a map $f: \mathbf{Z} \to \mathbf{E}$, s.t.,

$$f(x) = 2x \quad \text{for all } x \in \mathbf{Z}$$

Then f is well defined as $x = y \Rightarrow 2x = 2y \Rightarrow f(x) = f(y)$ such that f is 1-1 is clear by taking the steps backwards.

f is a homomorphism as,

$$f(x + y) = 2(x + y) = 2x + 2y = f(x) + f(y)$$

Also f is onto as any even integer $2x$ would have x as its pre-image.

Hence f is an isomorphism.

In fact this example shows that a subset can be isomorphic to its superset.

Example 1.30: Let f be a mapping from $< \mathbf{Z}, + >$ the group of integers to the group $G = \{1, -1\}$ under multiplication defined as

$$f: \mathbf{Z} \to G, \text{ s.t.,}$$

$$f(x) = 1 \quad \text{if } x \text{ is even}$$

$$= -1 \quad \text{if } x \text{ is odd}$$

Then f is clearly well defined. Verify, if it is a homomorphism.

Solution: Let $x, y \in \mathbf{Z}$ be any elements.

Case (*i*): x, y are both even, then $x + y$ is even and as

$$f(x + y) = 1, f(x) = 1, f(y) = 1$$

Then $f(x + y) = 1 = 1.1 = f(x) \,.\, f(y)$

Case (*ii*): x, y are both odd, then $x + y$ is even and

$$f(x + y) = +1 = (-1)(-1) = f(x)\, f(y)$$

Case (*iii*): x is odd, y is even, then $x + y$ is odd and

$$f(x + y) = -1 = (-1)\,(1) = f(x)\, f(y)$$

thus in all cases $f(x + y) = f(x)\, f(y)$

This proves that f is a homomorphism.

Ontoness is obvious, but f is not 1–1 as $f(x) = f(y)$ does not necessarily mean $x = y$. Indeed $f(2) = f(4)$ but $2 \neq 4$.

Example 1.31: Let $\mathbf{R}^+$ be the group of positive real numbers under multiplication and $\mathbf{R}$ the group of all real numbers under addition. Then show that the map

$$\theta : \mathbf{R}^+ \to \mathbf{R} \text{ s.t.,}$$

$$\theta(x) = \log x$$

is an isomorphism.

Solution: θ is clearly well defined.

$$\begin{aligned} & \theta(x) = \theta(y) \\ \Rightarrow\ & \log x = \log y \\ \Rightarrow\ & e^{\log x} = e^{\log y} \\ \Rightarrow\ & x = y \end{aligned}$$

This shows that θ is one-one.

Since, $\theta(xy) = \log xy = \log x + \log y = \theta(x) + \theta(y)$

We find θ is a homomorphism.

Finally, if $y \in \mathbf{R}$ be any member, then

Since $e^y \in \mathbf{R}^+$ and $\theta(e^y) = y$, we gather that θ is onto and hence on isomorphism. The map $f : \mathbf{R} \to \mathbf{R}^+$, such that, $f(a) = e^a$ can also be considered.)

Theorem 1.6: If $f : G \to G'$ is a homomorphism then

(*i*) $f(e) = e'$

(*ii*) $f(x^{-1}) = (f(x))^{-1}$

(*iii*) $f(x^n) = [f(x)]^n$, n an integer.

where *e, e'* are identity elements of *G* and *G'* respectively.

Proof: (*i*) We have

$$\begin{aligned} & e \,.\, e = e \\ \Rightarrow\ & f(e \,.\, e) = f(e) \\ \Rightarrow\ & f(e) \,.\, f(e) = f(e) \\ \Rightarrow\ & f(e) \,.\, f(e) = f(e) \,.\, e' \\ \Rightarrow\ & f(e) = e' \text{ (cancellation)} \end{aligned}$$

(*ii*) Again, $xx^{-1} = e = x^{-1}x$

$$\begin{aligned} \Rightarrow\ & f(xx^{-1}) = f(e) = f(x^{-1}x) \\ \Rightarrow\ & f(x)\, f(x^{-1}) = e' = f(x^{-1})\, f(x) \\ \Rightarrow\ & (f(x))^{-1} = f(x^{-1}). \end{aligned}$$

(*iii*) Let *n* be a +ve integer.

$$\begin{aligned} f(x^n) &= \underset{(n \text{ times})}{f(x \,.\, x \,........ x)} \\ &= f(x) \,.\, f(x) \,......\, f(x) \quad (n \text{ times}) \\ &= (f(x))^n. \end{aligned}$$

If $n = 0$, we have the result by (*i*). In case *n* is –ve integer, result follows by using (*ii*).

Example 1.32: Show that < **Q**, + > cannot be isomorphic to < **Q***, · >, where **Q*** = **Q** –{0} and **Q** = rationals.

Solution: Suppose *f* is an isomorphism from **Q** to **Q***. Then as $2 \in \mathbf{Q}^*$, *f* is onto, $\exists\ \alpha \in <\mathbf{Q}, +>$, such that, $f(\alpha) = 2$.

$$\Rightarrow \quad f\left(\frac{\alpha}{2} + \frac{\alpha}{2}\right) = 2$$

$$\text{or} \quad f\left(\frac{\alpha}{2}\right) f\left(\frac{\alpha}{2}\right) = 2$$

$$\Rightarrow \quad x^2 = 2 \quad \text{where } x = f\left(\frac{\alpha}{2}\right) \in \mathbf{Q^*}$$

NOTES

But that is a contradiction as there is no rational no. x such that, $x^2 = 2$. Hence the result follows.

Example 1.33: Find all the homomorphisms from $\frac{\mathbf{Z}}{4\mathbf{Z}}$ to $\frac{\mathbf{Z}}{6\mathbf{Z}}$.

Solution: Let $f : \frac{\mathbf{Z}}{4\mathbf{Z}} \to \frac{\mathbf{Z}}{6\mathbf{Z}}$ be a homomorphism.

Then $f(4\mathbf{Z} + n) = n f(4\mathbf{Z} + 1)$

So, f is completely known if $f(4\mathbf{Z} + 1)$ is known.

Now order of $(4\mathbf{Z} + 1)$ is 4 and so $o(f(4\mathbf{Z} + 1))$ divides 4.

Also $o(f(4\mathbf{Z} + 1))$ divides 6 and thus $o(f(4\mathbf{Z} + 1)) = 1$ or 2

If $\quad o(f(4\mathbf{Z} + 1)) = 1$, then $f(4\mathbf{Z} + 1) = 6\mathbf{Z} =$ zero of $\frac{\mathbf{Z}}{6\mathbf{Z}}$

$$\begin{aligned}
\text{Hence,} \qquad f(4\mathbf{Z} + n) &= \text{zero} \\
\text{If} \qquad o(f(4\mathbf{Z} + 1)) &= 2, \text{ then } f(4\mathbf{Z} + 1) = 6\mathbf{Z} + 3 \\
\Rightarrow \qquad f(4\mathbf{Z} + n) &= 6\mathbf{Z} + 3n \\
\text{Also,} \quad f(4\mathbf{Z} + n + 4\mathbf{Z} + m) &= f(4\mathbf{Z} + n + m) \\
&= 6\mathbf{Z} + 3(n + m) \\
&= (6\mathbf{Z} + 3n) + (6\mathbf{Z} + 3m) \\
&= f(4\mathbf{Z} + n) + f(4\mathbf{Z} + m)
\end{aligned}$$

Thus there are two choices for f and it can be defined as,

$$\begin{aligned}
f : \frac{\mathbf{Z}}{4\mathbf{Z}} &\to \frac{\mathbf{Z}}{6\mathbf{Z}} \quad \text{s.t.,} \\
f(4\mathbf{Z} + n) &= 6\mathbf{Z} + 3n \\
\text{Notice} \qquad 4\mathbf{Z} + n &= 4\mathbf{Z} + m \\
\Rightarrow \qquad n - m &\in 4\mathbf{Z} \\
\Rightarrow \qquad 3(n - m) &\in 12\mathbf{Z} \subseteq 6\mathbf{Z} \\
\Rightarrow \qquad 3(n - m) &\in 6\mathbf{Z} \\
\Rightarrow \qquad 6\mathbf{Z} + 3n &\in 6\mathbf{Z} + 3m
\end{aligned}$$

i.e., f is well defined.

So there are two homomorphisms from $\frac{\mathbf{Z}}{4\mathbf{Z}} \to \frac{\mathbf{Z}}{6\mathbf{Z}}$. In fact, in general, there are d homomorphisms from $\frac{\mathbf{Z}}{m\mathbf{Z}} \to \frac{\mathbf{Z}}{n\mathbf{Z}}$ where $d =$ g.c.d.(m, n)

Definition: Let $f : G \to G'$ be a homomorphism. The **Kernel** of f, (denoted by Ker f) is defined by

$$\text{Ker } f = \{x \in G \mid f(x) = e'\}$$

where e' is identity of G'.

Theorem 1.7: If $f : G \to G'$ be a homomorphism, then *Ker f* is a normal subgroup of G.

Proof: Since $f(e) = e'$, $e \in$ Ker f, thus Ker $f \neq \varphi$. Again,

$$x, y \in \text{Ker } f \Rightarrow f(x) = e'$$
$$f(y) = e'$$

Now $\quad f(xy^{-1}) = f(x)\, f(y^{-1}) = f(x)(f(y))^{-1} = e' \,.\, e'^{-1} = e'$

$$\Rightarrow xy^{-1} \text{ Ker } f$$

Hence, it is a subgroup of G.

Again, for any $g \in G$, $x \in$ Ker f

$$f(g^{-1}xg) = f(g^{-1})f(x)f(g)$$
$$= (f(g))^{-1}f(x)\, f(g) \qquad = (f(g))^{-1}\, e' f(g)$$
$$= (f(g))^{-1}f(g) = e'$$

$$\Rightarrow g^{-1}xg \in \text{Ker } f$$

Also, it is a normal subgroup of G.

Theorem 1.8: A homomorphism $f : G \to G'$ is one-one iff *Ker* $f = \{e\}$.

Proof: Let $f : G \to G'$ be one-one.

Let $x \in$ Ker f be any element

Then $\qquad f(x) = e'$ and as $f(e) = e'$

$\qquad f(x) = f(e) \Rightarrow x = e$ as f is 1-1

Hence, $\qquad$ Ker $f = \{e\}$.

Conversely, let Ker f contain only the identity element.

Let $\qquad f(x) = f(y)$

Then $\qquad f(x)\;(f(y))^{-1} = e'$

$$\Rightarrow \qquad f(xy^{-1}) = e'$$
$$\Rightarrow \quad xy^{-1} \in \text{Ker } f = \{e\}$$
$$\Rightarrow \qquad xy^{-1} = e$$
$$\Rightarrow \qquad x = y \text{ or that } f \text{ is one-one.}$$

Example 1.34: Let $f : G \to G'$ be a homomorphism. Let $a \in G$ be such that $o(a) = n$ and $o(f(a)) = m$. Show that $o(f(a)) \mid o(a)$ and f is 1-1 iff $m = n$.

Solution: Since $o(a) = n$

We obtain $\quad a^n = e \;\Rightarrow\; f(a^n) = f(e)$

$$\Rightarrow f(a \,.\, a \,.....\, a) = f(e)$$
$$\Rightarrow (f(a))^n = e$$
$$\Rightarrow o(f(a)) \mid n = o(a)$$

Again, let f be 1-1.

Since $\qquad o(f(a)) = m$

We obtain $\qquad (f(a))^m = e'$

$$\Rightarrow \quad f(a) \,.\, f(a) \,.....\, f(a) = e'$$
$$\Rightarrow \quad f(a \,.\, a \,.....\, a) = e'$$
$$\Rightarrow \quad f(a^m) = e' = f(e)$$

$$\Rightarrow \quad a^m = e \quad (f \text{ is } 1\text{-}1)$$

i.e., $o(a) \mid m$ or $n \mid m$, but already $m \mid n$

Hence, $m = n$.

Conversely, let $o(a) = o(f(a))$.

Then, $f(x) = f(y)$

$$\Rightarrow \quad f(x)\ (f(y))^{-1} = e'$$
$$\Rightarrow \quad f(xy^{-1}) = e'$$
$$\Rightarrow \quad o(f(xy^{-1})) = 1$$
$$\Rightarrow \quad o(xy^{-1}) = 1 \;\Rightarrow xy^{-1} = e \;\Rightarrow x = y$$
$$\Rightarrow \quad f \text{ is } 1\text{-}1.$$

Note: Under an isomorphism, order of any element is preserved.

Example 1.35: Show that the group (**R**, +) of real numbers cannot be isomorphic to the group R' of non zero real numbers under multiplication.

Solution: $-1 \in R'$ and order of -1 is 2 as $(-1)^2 = 1$. But **R** has no element of order 2. As if $x \in$ **R** is of order 2 then $2x = x + x = 0$. But this does not hold in (**R**, +) for any x except $x = 0$.

By above remark, under an isomorphism order of an element is preserved. Thus there cannot be any isomorphism between **R** and R'.

Example 1.36: Let G be a group and $f : G \to G$ such that, $f(x) = x^{-1}$ *be a* homomorphism. Show that G is abelian.

Solution: Let $x, y \in G$ be any elements.

$$\begin{aligned} xy = (y^{-1}x^{-1})^{-1} &= f(y^{-1}x^{-1}) \\ &= f(y^{-1})\, f(x^{-1}) \\ &= yx, \text{ hence } G \text{ is abelian.} \end{aligned}$$

Theorem 1.9: (Fundamental Theorem of Group Homomorphism). If $f : G \to G'$ be an onto homomorphism with $K = Ker f$, then $\frac{G}{K} \cong G'$.

In other words, every homomorphic image of a group G is isomorphic to a quotient group of G.

Proof: Define a map $\varphi : \frac{G}{K} \to G'$, such that,

$$\varphi(Ka) = f(a), \quad a \in G$$

We show φ is an isomorphism.

That φ is well defined follows by,

$$Ka = Kb$$
$$\Rightarrow \quad ab^{-1} \in K = \text{Ker}\, f$$
$$\Rightarrow \quad f(ab^{-1}) = e'$$

$$\Rightarrow \quad f(a)(f(b))^{-1} = e'$$
$$\Rightarrow \quad f(a) = f(b)$$
$$\Rightarrow \quad \varphi(Ka) = \varphi(Kb)$$

By retracing the steps backwards, we will prove that φ is 1-1.

Again as $\varphi(KaKb) = \varphi(Kab) = f(ab) = f(a) f(b)$
$= \varphi(Ka)\ \varphi(Kb)$

We obtain that φ is a homomorphism.

To check that φ is onto, let $g' \in G'$ be any element. Since $f: G \to G'$ is onto, $\exists\ g \in G$, such that,

$$f(g) = g'$$

Now $\varphi(Kg) = f(g) = g'$

Showing thereby that Kg is the required pre-image of g' under φ.

Hence φ is an isomorphism.

Note: The above theorem is also called first theorem of isomorphism.

Direct Products

The reader is well acquainted with the idea of product of two sets as a set of ordered pairs. We explore the possibility of getting a new group through the product of two groups. Let G_1, G_2 be any two groups.

Let $G = G_1 \times G_2 = \{(g_1, g_2) \mid g_1 \in G_1, g_2 \in G_2\}$.

What better way could there be than to define multiplication on G by $(g_1, g_2)\ (g'_1, g'_2) = (g_1 g'_1, g_2 g'_2)$. That G forms a group under this as its composition should not be a difficult task for the reader. Indeed (e_1, e_2) will be identity of G where e_1, e_2 are identities of G_1 and G_2 respectively. Also $(g_1, g_2)^{-1} = (g_1^{-1}, g_2^{-1})$.

We call $G = G_1 \times G_2$ direct product or External Direct Product (EDP) of $G_1,\ G_2$.

Again, if G_1, G_2 are abelian then so would be $G_1 \times G_2$.

In a similar way, we can define external direct product $G_1 \times G_2 \times ... \times G_n$ of arbitrary groups $G_1, G_2 ..., G_n$ as

$$G_1 \times ... \times G_n = \{(g_1,..., g_n) \mid g_i \in G_i\}$$

Where compostion is component wise multiplication.

Let $G = G_1 \times ... \times G_n$ = direct product of $G_1,..., G_n$.

Define $H_1 = \{g_1, e_2,..., e_n\} \mid g_1 \in G_1, e_i = \text{identity of } G_i\}$

$H_2 = \{(e_1, g_2, e_3..., e_n) \mid g_2 \in G_2\}$.

....

$H_n = \{(e_1, e_2, e_3..., g_n) \mid g_n \in G_n\}$

We show that H_1 is normal in G.

$H_1 \neq \varphi$ as $(e_1, e_2,..., e_n) \in H_1$

Let $(g_1, e_2,..., e_n)\ (g'_1, e_2,..., e_n) \in H_1$

Then $(g_1, e_2,..., e_n)\ (g'_1, e_2,..., e_n)^{-1}$

$= (g_1, e_2,..., e_n)\ (g_1^{-1}, e_2,...e_n)$

$= (g_1g_1^{-1}, e_2,...e_n) \in H_1$

Thus, $H_1 \leq G$

NOTES

Let $g = (g_1........g_n) \in G$

$x = (x_1, e_2,..., e_n) \in H_1$

Then, $gxg^{-1} = (g_1,..., g_n)\ (x_1, e_2..., e_n)\ (g_1^{-1},..., g_n^{-1})$

$= (g_1\, x_1\, g_1^{-1}, e_2,...e_n) \in H_1$

$\therefore$ H_1 is normal in G.

Similarly, each H_i is normal in G for all $i = 1,..., n$.

Let $g = (g_1,..., g_n) \in G$

Then, $g = (g_1, e_2,..., e_n)\ (e_1, g_2, e_3...e_n)...(e_1, e_2 ,..., e_{n-1} , g_n) \in H_1\, H_2...H_n$

Suppose $g = h_1\, h_2...h_n = h'_1\, h'_2,...,h'_n,\ h_i,\ h'_i \in H_i$

Then, $(g_1, e_2,..., e_n)\ ...\ (e_1,... e_{n-1}, g_n) = (g'_1,..., e_n)...(e_1,...e_{n-1}, g'_n)$

$\Rightarrow (g_1,..., g_n) = (g'_1...g'_n)$

$\Rightarrow g_i = g'_i$ for all $i = 1,..., n$

$\Rightarrow h_i = h'_i$ for all $i = 1,..., n$

So, $g \in G$ can be written uniquely as product of elements from $H_1,..., H_n$.

We summarize this through the following definition.

Let $H_1 ,..., H_n$ be normal subgroups of G. G is said to be an internal direct product (IDP) of $H_1,..., H_n$ if $G = H_1H_2... H_n$ and each $g \in G$ can be written uniquely as product of elements from $H_1,..., H_n$.

Consider the groups $\mathbf{Z}_2 = \{0, 1\}$, $\mathbf{Z}_3 = \{0, 1, 2\}$ under addition modulo. Here $\mathbf{Z}_2 \times \mathbf{Z}_3 = \{(0, 0), (0, 1), (0, 2), (1, 0)\ (1, 1), (1, 2)\}$ will form a group under element wise multiplication (addition).

Indeed, $2\,(1, 1) = (1, 1) + (1, 1) = (1\oplus_2 1, 1\oplus_3 1) = (0, 2)$,

$3(1, 1) = (1, 1) + (1, 1) + (1, 1) = (1, 0)$ etc.

We further note that since two cyclic groups of same order are isomorphic, we must have $\mathbf{Z}_2 \times \mathbf{Z}_3 \cong \mathbf{Z}_6$.

On the other hand one can show that $\mathbf{Z}_2 \times \mathbf{Z}_2$ is not isomorphic to $\mathbf{Z}_4$. In fact $\mathbf{Z}_2 \times \mathbf{Z}_2$ is not cyclic (whereas $\mathbf{Z}_4$ is). If $\mathbf{Z}_2 \times \mathbf{Z}_2$ is cyclic then it has a generator whose order should be same as $o(\mathbf{Z}_2 \times \mathbf{Z}_2) = 4$. But no element of $\mathbf{Z}_2 \times \mathbf{Z}_2$ has order 4. Notice, $2(1, 1) = (0, 0)$, i.e., order of $(1, 1)$ is less then or equal to 2 etc. Hence no element can be generator of $\mathbf{Z}_2 \times \mathbf{Z}_2$. One can show that $\mathbf{Z}_n \times \mathbf{Z}_m \cong \mathbf{Z}_{nm}$ iff n and m are relatively prime.

Theorem 1.10: Let H_1, H_2 be normal in G. Then G is an IDP of H_1 and H_2 if and only if

(*i*) $G = H_1H_2$

(*ii*) $H_1 \cap H_2 = \{e\}$.

Proof: Suppose G is an IDP of H_1 and H_2. Let $g \in G$.

Then $g = h_1h_2,\ h_1 \in H_1, h_2 \in H_2$.

Then $G \subseteq H_1H_2$. But $H_1H_2 \subseteq G$

$\Rightarrow \qquad G = H_1H_2$

Let $\qquad g \in H_1 \cap H_2 \Rightarrow g \in H_1, g \in H_2$

$\therefore$ $g = ge = eg$ is written in 2 ways as product of elements from H_1 and H_2.

$\therefore \qquad g = e \Rightarrow H_1 \cap H_2 = \{e\}.$

Conversely, let $G = H_1H_2$ and $H_1 \cap H_2 = \{e\}$

Let $g \in G \Rightarrow g \in H_1H_2 \Rightarrow g = h_1h_2,\ h_1 \in H_1, h_2 \in H_2$

Let $g = h_1h_2 = h'_1h'_2,\quad h_1, h'_1 \in H_1, h_2, h'_2 \in H_2$

$\Rightarrow h_1^{-1}\, h'_1 = h_2h_2^{-1} \in H_1 \cap H_2 = \{e\}$

$\Rightarrow h_1 = h'_1, h_2 = h'_2$

$\therefore$ G is an IDP of H_1 and H_2.

For example, let $G = < a >$ be of order 6. Let $H = \{e, a^2, a^4\}$, $K = \{e, a^3\}$ then H and K are normal (G is abelian) subgroups of G. $H \cap K = \{e\}$.

$$HK = \{e, ea^3, a^2e, a^2a^3, a^4e, a^4a^3\}$$

$$= \{e, a^2, a^3, a^4, a^5, a\} = G$$

Hence, G is IDP of H and K.

Theorem 1.11: Let $H_1, H_2,..., H_n$ be normal in G. Then G is an IDP of H_1, $H_2,..., H_n$ if and only if

(*i*) $G = H_1H_2... H_n$

(*ii*) $H_i \cap H_1H_2... H_{i-1}\, H_{i+1}...H_n = \{e\}$

for all $i = 1,... n$

Proof: Suppose G is an IDP of $H_1,...,H_n$. Then (*i*) follows from the definition of IDP

Let $\qquad g \in H_i \cap H_1... H_{i-1}\, H_{i+1} ... H_n$

Then $\qquad g = h_i,\ h_i \in H_i$ and $g = h_1h_2...h_{i-1}\, h_{i+1}... h_n, h_j \in H_j$

$\Rightarrow g = ee ... h_i ... e$

$g = h_1h_2 ... h_{i-1}\, e\, h_{i+1} ... h_n$

Since this representation of g should be unique we get $e = h_1$, $e = h_2$,..., $h_i = e$,...

Or that $g = e$, which proves the result.

Conversely, let $g \in G$ then $g \in H_1... H_n \Rightarrow g = h_1... h_n, h_i \in H_i$

We show this representation is unique.

Let $\qquad g = h'_1 .... h'_n,\ h'_i \in H_i$

$\therefore \qquad h_1 .... h_n = h'_1 .... h'_n$

By (*ii*) $H_i \cap H_j = \{e\}$ for all $i \neq j$ because if $x \in H_i \cap H_j$

Then $x \in H_i,\ x \in H_j$. $(j \neq i)$

$x \in H_j \Rightarrow x \in H_1 ... H_j ... H_{i-1}\, H_{i+1} ... H_n$

As $\qquad x = e ... x ... e. e ... e$

$\Rightarrow x \in H_i \cap H_1 ... H_{i-1}\, H_{i+1} ... H_n = \{e\}$

Also H_i is normal in G, H_j is normal in G for all *i, j,* thus $h_ih_j = h_jh_i$ for all $i \neq j$

$\therefore \qquad h_1 \,..........\, h_n = h'_1 \,..........\, h'_n$

$\Rightarrow \; h_n = (h_1^{-1}\, h'_1)\; (h_2^{-1}\, h'_2)\; ....\; (h_{n-1}^{-1}\, h'_{n-1})\; h'_n$

$\therefore \; h_n\, h_n'^{-1} = (h_1^{-1}\, h'_1)\; ....\; (h_{n-1}^{-1}\, h'_{n-1}) \in H_1 \,........\, H_{n-1} \cap H_n = \{e\}$

NOTES

$\therefore \qquad h_n = h'_n$

Similarly, $\quad h_{n-1} = h'_{n-1}, ..., h_1 = h'_1$

Hence, G is an IDP of $H_1, ..., H_n$.

Note: If G is an IDP of $H_1, H_2, ..., H_n$ then $H_i \cap H_j = \{e\}, i \neq j$.

We now show that IDP of subgroups of G is isomorphic to their External Direct Product (EDP).

Example 1.37: Let G be a finite abelian group. Prove that G is isomorphic to the direct product of its Sylow subgroups.

Solution: Let $o(G) = p_1^{\alpha 1} \ldots p_r^{\alpha r}$

Where $p_1, ..., p_r$ are distinct primes.

Since G is abelian, each Sylow subgroup H_i of G is normal. $o(H_i) = p_i^{\alpha i}$.

Let $\qquad g \in H_i \cap H_1 \ldots H_{i-1}\, H_{i+1} \ldots H_r$

$\Rightarrow \; g \in H_i, g \in H_1 \ldots H_{i-1}\, H_{i+1} \ldots H_r$

Let $\qquad t = p_1^{\alpha 1} \ldots p_{i-1}^{\alpha t-1}\, p_{i+1}^{\alpha i+1} \ldots p_r^{\alpha r}$

$g \in H_1 \ldots H_{i-1}\, H_{i+1} \,..\, H_r$

$\Rightarrow \; g = h_1 \ldots h_{i-1}\, h_{i+1} \ldots h_r, \;\; h_j \in H_j$

$\Rightarrow \; g^t = h_1^t \ldots h_{i-1}^t\; h_{i+1}^t \ldots h_r^t = e \;$ as $h_j^t = e$ for all $j \neq i$

$\Rightarrow \; o(g) \mid t$

But $\quad g \in H_i \qquad \Rightarrow \; o(g) \mid o(H_i) = p_i^{\alpha i}$

$\Rightarrow \; o(g) = p_i^{\beta i}, \;\; \beta_i \geq 0$

$\therefore \qquad p_i^{\beta i} \mid t$

$\Rightarrow \; p_i^{\beta i} \mid p_1^{\alpha 1} \ldots p_{i-1}^{\alpha i-1}\; p_{i+1}^{\alpha i+1} \ldots p_r^{\alpha r}$

$\Rightarrow \; \beta i = 0$

$\Rightarrow \; o(g) = 1 \Rightarrow g = e.$

$\therefore \; H_i \cap H_1 \ldots H_{i-1}\, H_{i+1} \ldots H_r = \{e\}$ for all $i = 1, ..., r$

Now $o(H_1 ...... H_r) = \dfrac{o(H_1)\, o(H_2 \ldots H_r)}{o(H_1 \cap H_2 \ldots H_r)} = o(H_1)\, o(H_2 ...... H_r)$

Again, $\qquad o(H_2 H_3 ... H_r) = \dfrac{o(H_2).\, o(H_3 \ldots H_r)}{o(H_2 \cap H_3 \ldots H_r)}$

Now $\qquad x \in H_2 \cap H_3 \ldots H_r$

$\Rightarrow \; x \in H_2$ and $x \in H_3 \,........\, H_r \subseteq H_1 H_3 ...... H_r$

$\Rightarrow \; x \in H_2 \cap H_1 H_3 \ldots H_r = \{e\}$

So $\; x = e$

$\therefore \qquad o(H_2 .... H_r) = o(H_2)\; o(H_3 ...... H_r)$

In this way, we get

$$o(H_1 ....H_r) = o(H_1)\ o(H_2)........o(H_r) = o(G)$$

$$\Rightarrow\ G = H_1 \ ....\ H_r$$

By Theorem 1.11, G is an IDP of H_1,...... H_r and so isomorphic to EDP of H_1,..., H_r.

Note: If G is a finite group and all its Sylow subgroups are normal then G is direct product of its Sylow subgroups.

Example 1.38: Show that if G is a group of order 45, it is IDP of its Sylow subgroups.

Solution: $o(G) = 45 = 3^2 \times 5$.

Number of Sylow 5-subgroups is $(1 + 5k)$ s.t., $(1 + 5k) \mid 9$ which gives $k = 0$

i.e., $\exists$ a unique normal Sylow 5-subgroup H of G where $o(H) = 5$.

Similarly, $\exists$ *a* unique normal Sylow 3-subgroup K of order 9.

Since $o(H \cap K) \mid 9, 5$, we find $o(H \cap K) = 1 \Rightarrow H \cap K = \{e\}$

Also $\quad o(HK) = \dfrac{5 \times 9}{1} = 45 = o(G) \Rightarrow G = HK$

Hence, G is IDP of its sylow subgroups H & K.

Example 1.39: Let N be normal in G. If $G = H \times K$ where H, K are subgroups of G, then prove that either N is abelian or N intersects H or K non-trivially.

Solution: Suppose $N \cap H = \{e\}$, $N \cap K = \{e\}$.

Since $G = H \times K$, H is normal in G, K is normal in G. So $nh = hn$ for all $n \in N$, $h \in H$ and $nk = kn$ for all $n \in N$, $k \in K$.

Let $n_1, n_2 \in N$.

$$n_2 \in N \Rightarrow n_2 \in G \Rightarrow n_2 = h_2k_2,\ \ h_2 \in H,\ k_2 \in K$$

$$\begin{aligned} \therefore \qquad n_1n_2 &= n_1h_2k_2 \\ &= h_2n_1k_2 \\ &= h_2k_2n_1 \\ &= n_2n_1 \end{aligned}$$

So, N is abelian.

Example 1.40: Let G be the set of matrices of the type $\begin{bmatrix} 1 & a & b \\ 0 & 1 & c \\ 0 & 0 & 1 \end{bmatrix}$ where $a, b, c \in F_3$. Here, $F_3 = \{0, 1, 2\}$ mod 3.

Then one can check that G forms a non-abelian group. In fact, it would be a subgroup of the groups of all 3×3 non-singular matrices over F_3.

Since each of a, b, c have three choices, $o(G) = 3^3$.

Order of each non-identity element of G will be 3 as

$$\begin{bmatrix} 1 & a & b \\ 0 & 1 & c \\ 0 & 0 & 1 \end{bmatrix}^2 = \begin{bmatrix} 1 & 2a & 2b+ac \\ 0 & 1 & 2c \\ 0 & 0 & 1 \end{bmatrix} \neq I \text{ as one of } a, b, c \text{ is non-zero}$$

$$\text{And } \begin{bmatrix} 1 & a & b \\ 0 & 1 & c \\ 0 & 0 & 1 \end{bmatrix}^3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}^3.$$

If we consider the group $\mathbf{Z}_3 \times \mathbf{Z}_3 \times \mathbf{Z}_3$, then it is an abelian group of order 27 in which each non-identity element is of order 3. Thus both the groups have 26 elements of order 3 (plus one identity). But G and $\mathbf{Z}_3 \times \mathbf{Z}_3 \times \mathbf{Z}_3$ cannot be isomorphic as one is abelian and the other a non-abelian group.

1.3.2 Homomorphism of Semi-Groups

In mathematics, a semi-group is an algebraic structure consisting of a set along with an associative binary operation. A semi-group generalizes a monoid such that there may not exist an identity element. Initially, it generalized a group (a monoid with all inverses) to a type in which every element may not have an inverse, hence named as semi-group.

The binary operation of a semi-group is most often denoted multiplicatively, $x \bullet y$ or simply xy, denotes the result that apply the semi-group operation to the ordered pair (x, y). The operation must be associative so that $(x \bullet y) \bullet z = x \bullet (y \bullet z)$ for all x, y and z, but must not be commutative so that $x \bullet y$ does not have to equal $y \bullet x$, this is contrast to the regular multiplication operator where $xy = yx$.

By definition, a semi-group is an associative magma. A semi-group with an identity element is called a monoid. A group is then a monoid in which every element has an inverse element. Semi-groups must not be confused with quasigroups which are sets with a not necessarily associative binary operation such that division is always possible.

Definition

A semi-group is a set 'S' together with a binary operation '$\bullet$' that satisfies the following properties:

Closure: For all a, b in S, the result of the operation $a \bullet b$ is also in S.

Associativity: For all a, b and c in S, the equation $(a \bullet b) \bullet c = a \bullet (b \bullet c)$ holds.

In mathematical notation we have:

$a \bullet b \in S\ \forall a, b \in S$ and $(a \bullet b) \bullet c = a \bullet (b \bullet c)\ \forall a, b, c \in \text{S}$.

This proves that a semi-group is an associative magma.

Examples of Semi-group: The following are the various examples of semi-group:

- Empty semi-group: The empty set forms a semi-group with the empty function as the binary operation.
- Semi-group with one element: There is essentially just one, the singleton $\{a\}$ with operation $a \bullet a = a$.
- The set of positive integers with addition.

- Square nonnegative matrices with matrix multiplication.
- A monoid is a semi-group with an identity element.
- A group is a monoid in which every element has an inverse element.

Identity and Zero

Every semi-group has at most one identity element. A semi-group with identity is called a monoid. A semi-group without identity may be embedded into a monoid simply by adjoining an element $e \notin S$ to S and defining $e \bullet s = s \bullet e = s$ for all $s \in S \cup \{e\}$. The notation S^1 denotes a monoid obtained from S by adjoining an identity if necessary ($S^1 = S$ for a monoid).

Similary, every semi-group has at most one absorbing element which in semi-group theory is called a **zero**. For every semi-group S, you can define S^0, a semi-group with 0 that embeds S.

Subsemi-Groups and Ideals

The semi-group operation induces an operation on the collection of its subsets: given subsets A and B of a semi-group, $A*B$, written commonly as AB, is the set $\{ab \mid a \text{ in } A \text{ and } b \text{ in } B\}$. In terms of this operations, a subset A is called.

- A **subsemi-group** if AA is a subset of A.
- A **right ideal** if AS is a subset of A.
- A **left ideal** if SA is a subset of A.

If A is both a left ideal and a right ideal then it is called an **ideal** or a **two-sided ideal**. If S is a semi-group, then the intersection of any collection of subsemi-groups of S is also a subsemi-group of S. So the subsemi-groups of S form a complete lattice.

An example of semi-group with no minimal ideal is the set of positive integers under addition. The minimal ideal of a commutative semigroup, when it exists, is termed a group.

Homomorphisms and Congruences

A **semi-group homomorphism** is a function that preserves semi-group structure. A function $f: S \to T$ between two semi-groups is a homomorphism if the equation, $f(ab) = f(a)f(b)$ holds for all elements a, b in S, i.e., the result is the same when performing the semi-group operation after or before applying the map f.

A semi-group homomorphism between monoids preserves identity iff it is a monoid homomorphism. But there are semi-group homomorphisms which are not monoid homomorphisms, for example the canonical embedding of a semi-group S without identity into S^1.

Let $f: S_0 \to S_1$ be a semi-group homomorphism. The image of f is also a semi-group. If S_0 is a monoid with an identity element e_0, then $f(e_0)$ is the identity element in the image of f. If S_1 is also a monoid with an identity element e_1 and e_1 belongs to the image of f, then $f(e_0) = e_1$, i.e., f is a monoid homomorphism. Particularly, if f is surjective, then it is a monoid homomorphism.

Two semi-groups S and T are said to be isomorphic if there is a bijection $f : S \rightarrow T$ with the property that, for any elements a, b in S, $f(ab) = f(a)f(b)$. Isomorphic semi-groups have the same structure.

NOTES

A **semi-group congruence** ~ is an equivalence relation that is compatible with the semi-group operation. That is, a subset $\sim\subseteq S \times S$ that is an equivalence relation and $x \sim y$ and $u \sim v$ implies $xu \sim yv$ for every x, y, u, v in S. Similar to any equivalence relation, a semi-group congruence ~ induces congruence classes,

$[a] \sim = \{x \in S\} \mid x \sim a\}$

The semi-group operation induces a binary operation $\circ$ on the congruence classes as follows:

$[u] \sim o[v] \sim = [uv] \sim$

Because ~ is a congruence, the set of all congruence classes of ~ forms a semi-group with o, called the **quotient semi-group** or **factor semi-group** and denoted $S / \sim$. The mapping $x \rightarrow [x]\sim$ is a semi-group homomorphism, called the **quotient map**, **canonical surjection** or **projection**; if S is a monoid then quotient semi-group is a monoid with identity. Conversely, the kernel of any semi-group homomorphism is a semi-group congruence.

Every ideal I of a semi-group induces a subsemi-group via the congruence $x\rho y \Leftrightarrow$ either $x = y$ or both x and y are in I.

Structure of Semi-Groups

For any subset A of S there is a smallest subsemi-group T of S which contains A, and we say that A **generates** T. A single element x of S generates the subsemi-group $\{x^n \mid n$ is a positive integer$\}$. If this is finite, then x is said to be of **finite order**, otherwise it is of **infinite order**. A semi-group is said to be **periodic** if all of its elements are of finite order. A semi-group generated by a single element is said to be monogenic or cyclic. If a monogenic semi-group is infinite then it is isomorphic to the semi-group of positive integers with the operation of addition. If it is finite and nonempty, then it must contain at least one idempotent. It follows that every nonempty periodic semi-group has at least one idempotent.

A subsemi-group which is also a group is called a **subgroup**. There is a close relationship between the subgroups of a semi-group and its idempotent. Each subgroup contains exactly one idempotent, namely the identity element of the subgroup. For each idempotent e of the semi-group there is a unique maximal subgroup containing e. Each maximal subgroup arises in this way, so there is a one-to-one correspondence between idempotent and maximal subgroups. Here the term maximal subgroup differs from its standard use in group theory.

Group of Fractions

The group of fractions of a semi-group S is the group $G = G(S)$ generated by the elements of S as generators and all equations $xy = z$ which hold true in S as relations. There is an obvious map from S to $G(S)$ by sending each element of S to the corresponding generator.

An important question is to characterize those semi-groups for which this map is an embedding. This need not always is the case: for example, take S to be the semi-group of subsets of some set X with set-theoretic intersection as the binary operation (this is an example of a semi-lattice). Since $A.A = A$ holds for all elements of S, this must be true for all generators of G(S) as well: which is therefore the trivial group. It is clearly necessary for embeddability that S have the cancellation property. When S is commutative this condition is also sufficient and the semi-group provides a construction of the group of fractions.

1.3.3 Homomorphism of Monoids

In abstract algebra, a **monoid** is an algebraic structure with a single associative binary operation and an identity element. Monoids are already discussed in semi-group theory as they are natural semi-groups with identity. Monoids occur in several branches of mathematics; for example, they can be regarded as categories with a single object. Thus, they summarize the idea of function composition within a set.

Definition

A monoid is a set, S, together with a binary operation '•' (pronounced 'dot' or 'times') that satisfies the following three axioms:

Closure: For all a, b in S, the result of the operation $a \bullet b$ is also in S.

Associativity: For all a, b and c in S, the equation $(a \bullet b) \bullet c = a \bullet (b \bullet c)$ holds.

Identity Element: There exists an element e in S, such that for all elements a in S, the equation $e \bullet a = a \bullet e = a$ holds.

In mathematical notation we can write these as follows,

- Closure: $\forall a, b \in S : a \bullet b \in S$.
- Associativity: $\forall a, b, c \in S : (a \bullet b) \bullet c = a \bullet (b \bullet c)$.
- Identity element: $\exists e \in S :\ \forall\ a \in S : e \bullet a = \mathrm{a} \bullet e = a$.

More efficiently, a monoid is a semi-group with an identity element. It can also contain associativity and identity. A monoid with invertibility property is termed a group.

The symbol for the binary operation is generally ignored, for example the monoid axioms require $(ab)c = a(bc)$ and $ea = ae = a$.

Monoid Structures

Generators and Submonoids: A **submonoid** of a monoid M is a subset N of M containing the unit element such that if $x,y \in N$ then $x \cdot y \in N$. It is obvious that N is itself a monoid, under the binary operation induced by that of M. Equivalently, a submonoid is a subset N such that N=N*, where the superscript * is the Kleene star, i.e., the set is closed under composition or concatenation of its elements. For any subset N of M, the monoid N* is the smallest monoid that contains N.

A subset N is said to be a **generator** of M if and only if M=N*. If there is a finite generator of M, then M is said to be **finitely generated**.

Commutative Monoid

A monoid whose operation is commutative is called a **commutative monoid** or less commonly an **abelian monoid**. Commutative monoids are often written additively. Any commutative monoid is endowed with its **algebraic** preordering ≤, defined by $x \leq y$ if and only if there exists z such that $x + z = y$. An **order-unit** of a commutative monoid M is an element u of M such that for any element x of M, there exists a positive integer n such that $x \leq nu$. This is often used in case M is the positive cone of a partially ordered abelian group G, in which case we say that u is an order-unit of G.

Examples of Monoids

- Every singleton set $\{x\}$ gives rise to a particular one-element (trivial) monoid. The monoid axioms require that $x*x = x$ in this case.
- Every group is a monoid and every abelian group a commutative monoid.
- Every bounded semilattice is an idempotent commutative monoid.
- Any semi-group S may be turned into a monoid simply by adjoining an element e not in S and defining $e*s = s = s*e$ for all $s \in S$.
- The natural numbers, **N**, form a commutative monoid under addition (identity element zero) or multiplication (identity element one). A submonoid of **N** under addition is called a numerical monoid.
- The positive integers, **N-{0}**, form a commutative monoid under multiplication (identity element one).
- Given any monoid M, the **opposite monoid** M^{op} has the same carrier set and identity element as M and its operation is defined by $x *^{\mathrm{op}} y = y * x$. Any commutative monoid is the opposite monoid of itself.
- Given two sets M and N endowed with monoid structure (or, in general, any finite number of monoids, M_1, ..., M_k), their Cartesian product $M \times N$ is also a monoid (respectively, $M_1 \times ... \times M_k$). The associative operation and the identity element are defined pair wise.
- Fix a monoid M. The set of all functions from a given set to M is also a monoid. The identity element is a constant function mapping any value to the identity of M; the associative operation is defined point wise.
- Fix a monoid M with the operation * and identity element e, and consider its power set $P(M)$ consisting of all subsets of M. A binary operation for such subsets can be defined by $S * T = \{s * t : s \text{ in } S \text{ and } t \text{ in } T\}$. This turns $P(M)$ into a monoid with identity element $\{e\}$. In the same way the power set of a group G is a monoid under the product of group subsets.
- Let S be a set. The set of all functions $S \rightarrow S$ forms a monoid under function composition. The identity is just the identity function.
- The set of homeomorphism classes of compact surfaces with the connected sum. Its unit element is the class of the ordinary 2-sphere.

- Let $\langle f \rangle$ be a cyclic monoid of order *n*, that is, $\langle f \rangle = \{f^0, f^1, ..., f^{n-1}\}$. Then $f^n = f^k$ for some $0 \le k \le n$. In fact, each such *k* gives a distinct monoid of order *n*, and every cyclic monoid is isomorphic to one of these.

Moreover, *f* can be considered as a function on the points {0, 1, 2, ..., *n* – 1} given by,

$$\begin{bmatrix} 0 & 1 & 2 & ... & n-2 & n-1 \\ 1 & 2 & 3 & ... & n-1 & k \end{bmatrix}$$

Or, equivalently

$$f(i) := \begin{cases} i+1, & \text{if } 0 \le i < n-1 \\ k, & \text{if } i = n-1. \end{cases}$$

Multiplication of elements in $\langle f \rangle$ is then given by function composition.

Note also that when $k = 0$ then the function *f* is a permutation of {0, 1, 2, ..., *n* – 1} and gives the unique cyclic group of order *n*.

Properties

In a monoid, you can define positive integer powers of an element $x : x^1=x$, and $x^n=x*...*x$ (*n* times) for $n>1$. The rule of powers $x^{n+p}=x^n * x^p$ is evident.

From the definition, it is evident that the identity element *e* is unique. Then, for any *x* we can set $x^0=e$ and the rule of powers is still true with nonnegative exponents.

It is possible to define invertible elements: an element *x* is called invertible if there exists an element *y* such that $x*y = e$ and $y*x = e$. The element *y* is called the inverse of *x* . If *y* and *z* are inverses of *x*, then by associativity $y = (zx)y = z(xy) = z$. Thus inverses, if they exist, are unique.

If *y* is the inverse of *x*, we can define negative powers of *x* by setting $x^{-1}=y$ and $x^{-n}=y*...*y$ (*n* times) for $n>1$. And the rule of exponents is still verified for all *n*,*p* rational integers. This is why the inverse of *x* is usually written x^{-1}. The set of all invertible elements in a monoid *M*, together with the operation *, forms a group. Hence, every monoid contains a group if only the trivial one consisting of the identity alone.

If a monoid has the cancellation property and is finite, then it is in fact a group, for example fix an element *x* in the monoid. Since the monoid is finite, $x^n = x^m$ for some $m > n > 0$. But then, by cancellation we have that $x^{m-n} = e$ where e is the identity. Therefore $x * x^{m-n-1} = $ e, so x has an inverse.

An **inverse monoid** is a monoid where for every *a* in *M*, there exists a unique a^{-1} in *M* such that $a=a*a^{-1}*a$ and $a^{-1}=a^{-1}*a*a^{-1}$. If an inverse monoid is cancellative, then it is a group.

Monoid Homomorphisms

A homomorphism between two monoids (*M*,*) and (M^2 ,•) is a function $f: M \to M'$ such that

- $f(x*y) = f(x)\bullet f(y)$ for all x, y in M
- $f(e) = e^2$

where e and e' are the identities on M and M' respectively. **Monoid homomorphisms** are sometimes simply called **monoid morphisms**.

Not every semi-group homomorphism is a monoid homomorphism since it may not preserve the identity. Contrast this with the case of group homomorphisms: the axioms of group theory ensure that every semi-group homomorphism between groups preserves the identity. For monoids this isn't always true and it is necessary to state it as a separate requirement.

A bijective monoid homomorphism is called a monoid isomorphism. Two monoids are said to be isomorphic if there is an isomorphism between them.

1.4 LATTICES

Lattice: A poset in which every pair of elements have both a least upper bound and a greatest lower bound is called a lattice.

For example,

(*i*) The poset $(\{1, 2, 4, 8\},1)$ is a lattice.

(*ii*) The poset $(P(S), \subseteq)$ is a lattice.

Here, $A \cup B$ = Least upper bound (LUB) of A and B and $A \cap B$ = Greatest lower bound (GLB) of A and B, for any $A \subseteq S$, $B \subseteq S$.

Definition and Basic Properties

A lattice is a partially ordered set $(L, \leq)$ in which every pair of elements $a, b \in L$ has a greatest lower bound (GLB) and a least upper bound (LUB).

The greatest lower bound (GLB) of a subset $\{a, b\} \subseteq L$ will be denoted by $a \wedge b$ and the least upper bound (LUB) by $a \vee b$. So GLB $\{a, b\} = a \wedge b$, called the meet of a and b and LUB $\{a, b\} = a \vee b$, called the join of a and b.

Note that $\wedge$ and $\vee$ are binary operations and we denote the lattice by $(A, \wedge, \vee)$. From the definition of $\wedge$ and $\vee$, it is clear that,

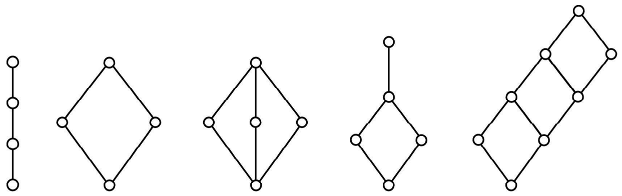
(*i*) $a \leq a \vee b$; $b \leq a \vee b$ (i.e., $a \vee b$ is an UB of a and b)

(*ii*) $a \wedge b \leq a$; $a \wedge b \leq b$ (i.e., $a \wedge b$ is a LB of a and b)

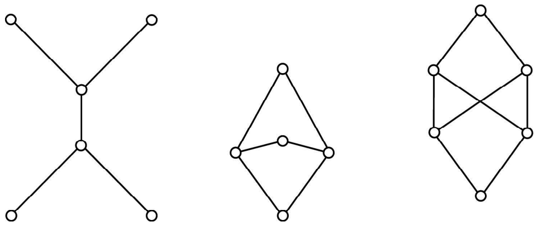
(*iii*) If $a \leq c$ and $b \leq c$ then $a \vee b \leq c$ (i.e., $a \vee b$ is the LUB of a and b)

(*iv*) If $c \leq a$ and $c \leq b$ then $c \leq a \wedge b$ (i.e., $a \wedge b$ is the GLB of a and b)

For example, Lattices



For example, Posets but not lattices



For example, Let A be any set and $L = P(A)$ be its power set. The poset $(L, \subseteq)$ is a lattice in which for any $x, y, \in L$, $x \wedge y = x \cap y$ and $x \vee y = x \cup y$.

For example, Let I be the set of positive integers. For any $x, y \in I$, $x \leq y$ if x/y. Define $x \vee y =$ LCM (x, y) and $x \wedge y =$ GCD $\in (x, y)$. Then $(I, \wedge, \vee)$ is a lattice.

Theorem 1.12: Let $(L, \leq)$ be a lattice in which $\wedge$ and $\vee$ denote the operations of meet and join respectively. For any $a, b, c \in L$, we have

(*i*) $a \wedge a = a$; $a \vee a = a$ (Idempotent)

(*ii*) $a \wedge b = b \wedge a$; $a \vee b = b \vee a$ (Commutative)

(*iii*) $(a \wedge b) \wedge c = a \wedge (b \cap c)$; $(a \vee b) \vee c = a \vee (b \vee c)$ (Associative)

(*iv*) $a \wedge (a \vee b) = a$; $a \vee (a \wedge b) = a$ (Absorption)

Proof:

(*i*) Since, $a \leq a$, we know that a is a lower bound of $\{a, a\} = \{a\}$. If b is also a lower bound and if $a \leq b$ then we have $b \leq a$ and $a \leq b$. By antisymmetry, $a = b$. So, a is the GLB of $\{a\}$. Therefore, $a \wedge a = a$. Dually, $a \vee a = a$ follows.

(*ii*) Let, $x = a \wedge b =$ GLB $\{a, b\}$. Since $\{a, b\} = \{b, a\}$, GLB $\{b, a\} = x$. So $b \wedge a = x$. Hence, $x = a \wedge b = b \wedge a$. Dually, $a \vee b = b \vee a$ follows.

(*iii*) Let $x = a \wedge (b \wedge c)$ and $y = (a \wedge b) \wedge c$.

Now, $x = a \wedge (b \wedge c) \Rightarrow x \leq a,\ x \leq b \wedge c$

$\Rightarrow x \leq a, x \leq b, x \leq c$

$\Rightarrow x \leq a \wedge b, x \leq c$

$\Rightarrow x \leq (a \wedge b) \wedge c = y.$

Similarly, $y \leq x$ follows. By antisymmetry, $x = y$ and hence,

$a \wedge (b \wedge c) = (a \wedge b) \wedge c.$

Dually, $a \vee (b \vee c) = (a \vee b) \vee c$ follows.

(*iv*) By definition, for any $a \in L$,

$a \leq a$ and $a \leq a \vee b$

Therefore, $a \leq a \wedge (a \vee b)$. But $a \wedge (a \vee b) \leq a$. Hence $a \wedge (a \vee b) = a$. Dually, $a \vee (a \wedge b) = a$ follows.

Theorem 1.13: Let $(L, \leq)$ be a lattice in which $\wedge$ and $\vee$ denote the operations of meet and join respectively. For any $a, b \in L$,

$$a \leq b \Leftrightarrow a \wedge b = a$$
$$\Leftrightarrow a \vee b = b$$

Proof: Assume that $a \leq a$. Since $a \leq b$, it follows that $a \leq a \wedge b$. But by definition of $\wedge$, $a \wedge b \leq a$. Therefore $a \wedge b = a$. Conversely, suppose $a \wedge b = a$. Then $a \leq b$. Hence, $a \leq b \Leftrightarrow a \wedge b = a$. Similarly $a \leq b \Leftrightarrow a \vee b = b$ follows.

Isotonicity Law: Let $(L, \leq)$ be a lattice in which $\wedge$ and $\vee$ denote the operations of meet and join respectively. For any $a, b, c, \in L$,

$$b \leq c \Rightarrow \begin{cases} a \wedge b \leq a \wedge c \\ a \vee b \leq a \vee c \end{cases}$$

Proof: Assume that $b \leq c$. Since $a \wedge b \leq b$, by Transitivity, $a \wedge b \leq c$. Since $a \wedge b \leq a$, it follows that,

$$a \wedge b \leq a \wedge c$$

Now, $b \leq c$ and $c \leq a \vee c$ implies $b \leq a \vee c$. But $a \leq a \vee c$. Hence $a \vee b \leq a \vee c$.

Note: For any $a, b, c \in L$, by Isotonicity law,

$a \leq b \wedge a \leq c \Rightarrow a \leq b \vee c$

$a \leq b \wedge a \leq c \Rightarrow a \leq b \vee c$

$c \leq b \wedge a \leq a \Rightarrow b \wedge c \leq a$

$c \leq b \wedge a \leq a \Rightarrow b \vee c \leq a$

Distributive Inequality: Let $(L, \leq)$ be a lattice. For any $a, b, c, \in L$, the following inequalities are hold.

(*i*) $a \vee (b \wedge c) \leq (a \vee b) \wedge (a \vee c)$

(*ii*) $(a \wedge b) \vee (a \wedge c) \leq a \wedge (b \vee c)$

(*i*) Since $a \leq a \vee b$ and $a \leq a \vee c$, we have

$$a \leq (a \vee b) \wedge (a \vee c) \qquad (1.1)$$

Since $b \wedge c \leq b \leq a \vee b$ and $b \wedge c \leq c \leq a \vee c$,

$$b \wedge c \leq (a \vee b) \wedge (a \vee c) \qquad (1.2)$$

From Equations (1.1) and (1.2)

$a \vee (b \wedge c) \leq (a \vee b) \wedge (a \vee c)$.

Similarly, case (*ii*) follows.

Modular Inequality: Let $(L, \leq)$ be a lattice. For any $a, b, c \in L$,

$$a \leq c \Leftrightarrow a \vee (b \wedge c) \leq (a \vee b) \wedge c$$

Proof: Suppose $a \leq c$. Then $a \vee c = c$.

By Distributive inequality, $a \vee (b \wedge c) \leq (a \vee b) \wedge (a \vee c)$

Since $a \vee c = c$,

$$a \vee (b \wedge c) \leq (a \vee b) \wedge c$$

Conversely, let us assume that $a \vee (b \wedge c) \leq (a \vee b) \wedge c$.

Since,

$$\begin{aligned} a &\leq a \vee (b \wedge c) \\ &\leq (a \vee b) \wedge c \\ &\leq c \end{aligned}$$

We get $a \leq c$. Hence proved.

Example 1.41: Prove that in a lattice $(L, \leq)$, for any $a, b, c \in L$, if $a \leq b \leq c \Rightarrow a \vee b = bc$, and $(a \wedge b) \vee (b \wedge c) = b = (a \vee b) \wedge (a \vee c)$.

Solution: Since $a \le b$ and $a \le c$, $a \le b \wedge c$. Again $b \le b$ and $b \le c$ implies $b \le b \wedge c$.

Now $a \le b \wedge c$ and $b \le b \wedge c \Rightarrow a \vee b \le b \wedge c$ (1)

Again, $b \wedge c \le b \le a \vee b$ (2)

From Equations (1) and (2), $a \vee b = b \wedge c$.

Hence $a \wedge b \le b$ and $b \wedge c \le b$, we get $(a \wedge b) \vee (b \wedge c) \le b$. Since $b \le c$ and $b \le b$ implies $b \le b \wedge c$. Again this implies $b \le (b \wedge c) \vee (a \wedge b)$. Hence $(a \wedge b) \vee (b \wedge c) = b$. Similarly, $(a \vee b) \wedge (a \vee c) = b$ follows.

Example 1.42: Prove that in a lattice $(L, \le)$, for any $a, b, c, d, \in L$, if $a \le b$ and $c \le d$ then $a \wedge c \le b \wedge d$.

Solution: Since $a \wedge c \le a \le b$ and $a \wedge c \le c \le d$, $a \wedge c \le b \wedge d$.

Distributive Lattices and Complemented Lattices

Distributive Lattice: A Lattice $(L, \le)$ is said to be distributive lattice if for any $a, b, c, \in L$,

$$a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$$

$$a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$$

Theorem 1.14: Let $a, b, c \in L$, where $(L, \le)$ is a distributive lattice. Then $a \vee b = a \vee c$ and $a \wedge b = a \wedge c \Rightarrow b = c$.

Proof: We know that

$$\begin{aligned} b &= b \vee (b \wedge a) \text{ (Absorption)} \\ &= b \vee (a \wedge b) \text{ (Commutative)} \\ &= b \vee (a \wedge c) \ (\because a \wedge b = a \wedge c) \\ &= (b \vee a) \wedge (b \vee c) \text{ (Distributive)} \\ &= (a \vee b) \wedge (c \vee b) \text{ (Commutative)} \\ &= (a \vee c) \wedge (c \vee b) \ (\because a \vee b = a \vee c) \\ &= (c \vee a) \wedge (c \vee b) \text{ (Commutative)} \\ &= c \vee (a \wedge b) \text{ (Distributive)} \\ &= c \vee (a \wedge c) \ (\because a \wedge b = a \wedge c) \\ &= c \vee (c \wedge a) \text{ (Commutative)} \\ &= c \text{ (Absorption)} \end{aligned}$$

Hence the proof.

Modular Lattice: A lattice $(L, \le)$ is said to be modular lattice if $a \le c \Rightarrow a \vee (b \wedge c) = (a \vee b) \wedge c$.

Bounded Lattice: A lattice $(L, \le)$ which has both, a least element denoted by 0, and the greatest element denoted by 1 is called a bounded lattice.

Note: If $L = \{a_1, a_{2,}...,a_n\}$ with $\bigwedge_{i=1}^{n} a_i = 0$ and $\bigvee_{i=1}^{n} a_i = 1$. It satisfies $a \vee 0 = a$, $a \vee 1 = 1$, $a \wedge 1 = a$ and $a \wedge 0 = 0$.

Complement of an Element: In a bounded lattices $(L, \le)$, an element $b \in L$ is called a complement of an element $a \in L$ if $a \wedge b = 0$ and $a \vee b = 1$, we denote b by a.

NOTES

Complement Lattice: A lattice $(L, \leq)$ is said to be complemented lattice if every element of L has at least one complement.

Example 1.43: Show that De Morgan's laws holds in a complemented distributive lattice.

Solution: To process that $(a \wedge b)' = a' \vee b'$ and

$(a \vee b)' = (a' \wedge b')$, consider

$$\begin{aligned}(a \wedge b)' \wedge (a' \vee b') &= \big((a \wedge b) \wedge a'\big) \;\vee\; \big((a \wedge b) \wedge b'\big) && \text{(Distributive)}\\ &= \big((b \wedge a) \wedge a'\big) \;\vee\; \big((a \wedge b \wedge \wedge b'\big) && \text{(Commutative)}\\ &= \big(b \wedge (a \wedge a')\big) \;\vee\; \big(a \wedge (b \wedge b')\big) && \text{(Associative)}\\ &= (b \wedge 0) \;\vee\; (a \wedge 0)\\ &= 0 \vee 0 \;=\; 0\end{aligned}$$

Again,

$$\begin{aligned}(a \wedge b)' \vee (a' \vee b') &= \big(a \vee (a' \vee b'\big) \wedge \big(b \vee (a' \vee b')\big) && \text{(Distributive)}\\ &= \big(a \vee (a' \vee b')\big) \wedge \big(b \vee (b' \vee a')\big) && \text{(Commutative)}\\ &= \big((a \vee a') \vee b'\big) \wedge \big((b \vee b') \vee a'\big) && \text{(Associative)}\\ &= (1 \vee b') \wedge (1 \vee a')\\ &= 1 \wedge 1 \;=\; 1\end{aligned}$$

Hence, $a' \vee b'$ is the complement of $(a \wedge b)$. So $a' \vee b' = (a \wedge b)'$. Similarly, $(a \vee b)' = a' \wedge b'$ follows.

Example 1.44: Show that in a complemented lattice $(L, \leq)$,
$a \leq b \Leftrightarrow a' \vee b = 1 \Leftrightarrow a \wedge b' = 0 \Leftrightarrow b' \leq a'$

Solution: Consider, $a \leq b \Leftrightarrow a \wedge b = a$

$$\begin{aligned}&\Leftrightarrow a' \vee a = a' \vee (a \wedge b) = 1\\ &\Leftrightarrow (a' \vee a) \wedge (a' \vee b) = 1\\ &\Leftrightarrow 1 \wedge (a' \vee b) = 1\\ &\Leftrightarrow a' \vee b = 1\end{aligned}$$

Again, $a \leq b \Leftrightarrow a \vee b = b$

$$\begin{aligned}&\Leftrightarrow b \wedge b' = (a \vee b) \wedge b' = 0\\ &\Leftrightarrow (a \wedge b') \vee (b \wedge b') = 1\\ &\Leftrightarrow (a \wedge b') \vee 0 = 0\\ &\Leftrightarrow a \wedge b' = 0.\end{aligned}$$

To prove the last one,

$$\begin{aligned}a \leq b &\Leftrightarrow a \vee b = b\\ &\Leftrightarrow (a \vee b') = b'\\ &\Leftrightarrow a' \wedge b' = b'\\ &\Leftrightarrow a' \wedge a' = b' && \text{(Commutative)}\\ &\Leftrightarrow b' \leq a'.\end{aligned}$$

Example 1.45: Consider the lattice $L = \{1, 2, 3, 4, 6, 12\}$, the divisions of 12 ordered by divisibility. Find,

(*i*) The lower bound and upper bound of L.

(*ii*) The complement of 4.

(*iii*) Is L a complemented lattice?

Solution:

(*i*) The lower bound of L is 1and the upper bound is 12.

(*ii*) Since $4 \wedge 3 = \text{gcd}\ (4,3) = 1$

$4 \vee 3 = \text{lcm}\ (4,3) = 12,$

The complement of 4 is 3

(*iii*) Since $6 \wedge x = \text{gcd}\ (6, x)$

$\neq 1$ for $x \neq 1$

$6 \vee 1 = \text{lcm}\ (6,1)$

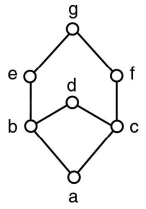
$\neq 12$

6 has no complement and hence L is not a complemented lattice.

Sublattice: Let M be a non-empty subset of a Lattice $(L, \leq)$. We say that M is a sublattice of L if M itself is a lattice with respect to the operations of L.

Note: So M is a sublattice of L if and only if M is closed under the operations $\wedge$ and $\vee$ of L.

Example 1.46: Consider the following lattice L.



Determine whether each of the following is a sublattice of L.

$$M = \{a, b, c, g\}$$
$$N = \{a, b, f, g\}$$
$$O = \{b, d, e, g\}$$
$$P = \{a, d, e, g\}$$

Solution: Since $b \vee c = d$, and $d \notin M$, M is not a sublattice. Since $d \wedge e = b$ and $b \notin p$, p is not a sublattice. But N and O are sublattices.

Example 1.47: Suppose M is a sublattice of a distributive lattice L. Show that M is a distributive lattice.

Solution: For a distributive lattice L, $a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$ and $a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$

for all $a, b, c \in L$. Since M is closed, each element of M is also in L, the distributive laws hold for all elements in M. Hence, M is a distributive lattice.

Example 1.48: Prove that in a distributive lattice $(L, \leq)$, if an element has a complement then this complement is unique.

Solution: Suppose for any $a \in L$ has two complements say b and c in L. Then $a \vee b = 1$; $a \wedge b = 0$ and $a \vee c = 1$; $a \wedge c = 0$.

$$\begin{aligned}\text{Consider } b = b \wedge 1 &= b \wedge (a \vee c)\\ &= (b \wedge a) \vee (b \wedge c) && \text{(Distributive)}\\ &= 0 \vee (b \wedge c) = (a \wedge c) \vee (b \wedge c)\\ &= (a \vee b) \wedge c && \text{(Distributive)}\\ &= 1 \wedge c = c\end{aligned}$$

Check Your Progress

6. Write the condition for a valid logical statement.
7. What are the two rules of inference?
8. Elaborate on the two types of quantifiers.
9. What is a semi-group?
10. Define monoid.
11. What do you mean by a lattice?

1.5 ANSWERS TO 'CHECK YOUR PROGRESS'

1. A proposition is a statement to which only one of the terms, true or false, can be meaningfully applied.
2. Logical operators are used to form new propositions or compound propositions.
3. The truth table of a logical operator specifies how the truth value of a proposition using that operator is determined by the truth values of the propositions.
4. Two propositions are logically equivalent or simply equivalent if they have exactly the same truth values under all circumstances.
5. We can draw inference on any given statement with symbols and logical connectives either by truth table or by applying rules of inference that are given in subsequent topic.
6. A logical statement is valid when it is a tautology.
7. The two rules of inference are called rules *P* and *T*.
8. Two types of quantifiers are Universal Quantification and Existential Quantification.
9. A semi-group is an algebraic structure consisting of a set along with an associative binary operation.
10. A monoid is an algebraic structure with a single associative binary operation and an identity element.

11. A poset in which every pair of elements has both a least upper bound and a greatest lower bound is called a lattice.

1.6 SUMMARY

NOTES

- A proposition is a statement to which only one of the terms, true or false, can be meaningfully applied.
- Logical operators are used to form new propositions or compound propositions.
- The truth table of a logical operator specifies how the truth value of a proposition using that operator is determined by the truth values of the propositions.
- A truth table lists all possible combinations of truth values of the propositions in the left most columns and the truth value of the resulting propositions in the right most column.
- The final column of a truth table of a given formula contains both 1 and 0.
- A statement formula which is true regardless of the truth values of the statements which replace the variables in it is called a tautology or a logical truth or a universally valid formula.
- A statement formula which is false regardless of the truth values of the statements which replace the variables in it is called a contradiction.
- A statement formula that is neither a tautology nor a contradiction is called a contingency.
- We can draw inference on any given statement with symbols and logical connectives either by truth table or by applying rules of inference that are given in subsequent topic.
- Two statements are equivalent if they have identical truth values.
- A logical statement is valid when it is a tautology.
- The two rules of inference are called rules *P* and *T*.
- Two types of quantifiers are universal quantification and existential quantification.
- In a group *G*, identity element is unique.
- A finite semi-group is a group if and only if it satisfies cancellation laws.
- A non-empty set *G* together with a binary composition '.' is said to form a monoid.
- All groups are monoids and all monoids are semi-groups.
- A non-empty subset *H* of a group *G* is said to be a subgroup of *G*, if *H* forms a group under the binary composition of *G*.
- A non-empty subset *H* of a group *G* is said to be a subgroup of *G*, if *H* forms a group under the binary composition of *G*.

NOTES

- An onto homomorphism is called epimorphism.
- A one-one homomorphism is called monomorphism.
- A homomorphism from a group G to itself is called an endomorphism of G.
- A semi-group is an algebraic structure consisting of a set along with an associative binary operation. A semi-group generalizes a monoid such that there may not exist an identity element.
- Every semi-group has at most one identity element. A semi-group with identity is called a monoid.
- A semi-group homomorphism is a function that preserves semi-group structure.
- A monoid is an algebraic structure with a single associative binary operation and an identity element.
- A poset in which every pair of elements has both a least upper bound and a greatest lower bound is called a lattice.

1.7 KEY TERMS

- **Propositions:** A proposition is a statement to which only one of the terms, true or false, can be meaningfully applied.
- **Truth tables:** The truth table of a logical operator specifies how the truth value of a proposition using that operator is determined by the truth values of the propositions.
- **Tautology:** A statement formula which is true regardless of the truth values of the statements which replace the variables in it is called a tautology or a logical truth or a universally valid formula.
- **Substitution instance:** A formula A is called a substitution instance of another formula B, if A can be obtained from B by substituting formulae for some variables of B, with the condition that the same formula is substituted for the same variables each time it occurs.
- **Semi-group:** It is an algebraic structure consisting of a set along with an associative binary operation and generalizes a monoid such that there may not exist an identity element.
- **Monoid:** It is an algebraic structure with a single associative binary operation and an identity element.

1.8 SELF-ASSESSMENT QUESTIONS AND EXERCISES

Short-Answer Questions

1. Write some applications of logic in computer science.
2. What are logical operators?

3. Mention two methods of constructing truth table.
4. What do you mean by tautology?
5. What is the use of universal specification and existential specification?
6. State the rule P and rule T?
7. Define quantifiers.
8. Explain the terms semi-group and monoid.
9. What is modular lattice?

Long-Answer Questions

1. Discuss the concept of mathematical logic.
2. Explain briefly how to use logical operators.
3. Illustrate the concept of truth tables.
4. Describe equivalence formula.
5. Discuss inference theory.
6. Write a note on validity by truth tables.
7. Explain rules of inference theory.
8. Discuss about quantifiers.
9. Discuss the significant points of systems that form groups or do not form groups.
10. Explain the concept of homomorphism, semi-group and monoid.
11. Explain briefly the concept of lattices with the help of examples.

1.9 FURTHER READING

Iyengar, N Ch S N. V M Chandrasekaran, KA Venkatesh and PS Arunachalam. *Discrete Mathematics*. New Delhi: Vikas Publishing House Pvt. Ltd., 2007.

Tremblay, Jean Paul and R. Manohar. *Discrete Mathematical Structures with Applications to Computer Science*. New York: McGraw-Hill Inc., 1975.

Deo, Narsingh. *Graph Theory with Applications to Engineering and Computer Science*. New Delhi: Prentice-Hall of India, 1999.

Singh, Y.N. *Mathematical Foundation of Computer Science*. New Delhi: New Age International Pvt. Ltd., 2005.

Malik, D.S. *Discrete Mathematical Structures: Theory and Applications*. London: Thomson Learning, 2004.

Haggard, Gary, John Schlipf and Sue Whiteside. *Discrete Mathematics for Computer Science*. California: Thomson Learning, 2006.

Cohen, Daniel I.A. *Introduction to Computer Theory*, 2nd edition. New Jersey: John Wiley and Sons, 1996.

NOTES

Hopcroft, J.E., Rajeev Motwani and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*, 3rd edition. Boston: Addison-Wesley, 2006.

Linz, Peter. *An Introduction to Formal Languages and Automata*, 5th edition. Boston: Jones and Bartlett Publishers, 2011.

Mano, M. Morris. *Digital Logic and Computer Design*. New Jersey: Prentice-Hall, 1979.

UNIT 2 BOOLEAN ALGEBRA

NOTES

Structure

2.0 INTRODUCTION

In this unit, you will learn that 'gates' are the basic building blocks of any digital logic circuitry. Logic High at the output is treated as gate 'ON' and Logic Low as gate 'OFF'. In digital electronics, the 'ON' state is often represented by 1 and the 'OFF' state by 0. The basic logic gates are NOT, AND, OR, XOR and XNOR. The details of all the logic gates will be discussed in this unit, along with their truth tables and circuit symbols.

2.1 OBJECTIVES

After going through this unit, you will be able to:

- Analyse basic logic gates
- Explain universal logic gates
- Draw logic circuits
- Analyse logic circuits
- Understand the basic laws of Boolean algebra
- Explain Boolean functions
- Understand the importance of Karnaugh map
- Explain minterm and maxterm
- Represent simplified expressions

2.2 BASIC LOGIC GATES

2.2.1 NOT Gate

The basic NOT gate has only one input and one output. The output is always the opposite or negation of the input. The following is the truth table for NOT gate:

Table 2.1 *Truth Table for NOT Gate*

A	F
0	1
1	0

Symbol: F = A′

The following is the figure of NOT gate representation:

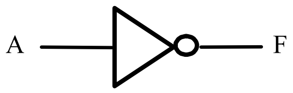


Fig. 2.1 *NOT Gate*

2.2.2 AND Gate

A basic AND gate consists of two inputs and an output. In the AND gate, the output is **‘High’** or gate is **‘On’** only if both the inputs are **‘High’.** The relationship between the input signals and the output signals is often represented in the form of a **truth table**. It is nothing but a tabulation of all possible input combinations and the resulting outputs. For the AND gate, there are four possible combinations of input states: $\{A = 0, B = 0\}$; $\{A = 0, B = 1\}$; $\{A = 1, B = 0\}$ and $\{A = 1, B = 1\}$. In the truth table, these are listed as follows:

Table 2.2 Truth Table for AND Gate

A	B	F
0	0	0
0	1	0
1	0	0
1	1	1

In Table 2.2, F represents the output of two inputs in the AND gate with input signals A and B.

Symbol: F = A.B (where **'.'** implies AND operation)

The Figure 2.2 represents the AND gate:

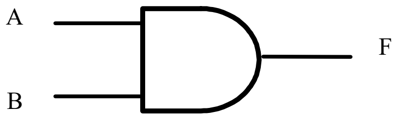


Fig. 2.2 *AND Gate*

2.2.3 OR Gate

A basic OR gate is a two input, single output gate. Unlike the AND gate, the output is 1 when any one of the input signals is 1. The OR gate output is 0 only when both the inputs are 0. The truth table for the OR gate is as follows:

Table 2.3 Truth Table for OR Gate

A	B	F
0	0	0
0	1	1
1	0	1
1	1	1

Symbol: F = A + B (where '+' implies OR operation)

The following figure represents OR gate:

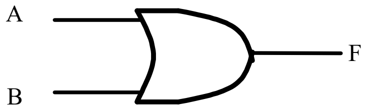


Fig. 2.3 *OR Gate*

2.2.4 XOR Gate

A gate related to the OR gate is the XOR gate or exclusive OR gate in which the output is 1 when one, and only one, of the inputs is 1. In other words, the XOR output is 1 if the inputs are different. The truth table for the XOR gate is as follows:

Table 2.4 Truth Table for XOR Gate

A	B	F
0	0	0
0	1	1
1	0	1
1	1	0

Symbol: F = A ⊕ B (where '⊕' implies XOR operation)

The following figure represents XOR gate:

NOTES

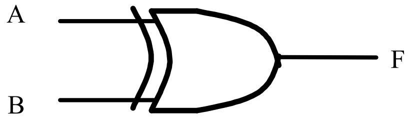


Fig. 2.4 XOR Gate

2.3 UNIVERSAL LOGIC GATES

In addition to the NOT, AND, OR and XOR gates, three more common gates are available. These are identical to AND, OR and XOR, except that the gate output has been negated. These gates are called NAND ('Not AND'), NOR ('Not OR') and XNOR ('Exclusive Not OR'). Their symbols are just the symbols of the un-negated gates with a small circle drawn at the output.

- **NAND Gate:** AND followed by NOT

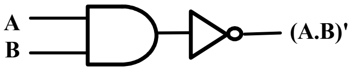


Fig. 2.5 AND Gate followed by NOT Gate

Symbol: NAND

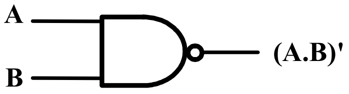


Fig. 2.6 NAND Gate

- **NOR Gate:** OR followed by NOT

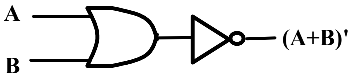


Fig. 2.7 OR Gate followed by NOT Gate

Symbol: NOR

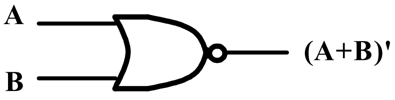


Fig. 2.8 NOR Gate

- **XNOR Gate:** XOR followed by NOT

Symbol: NOR

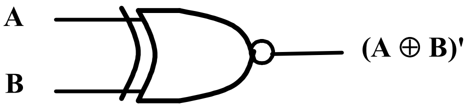
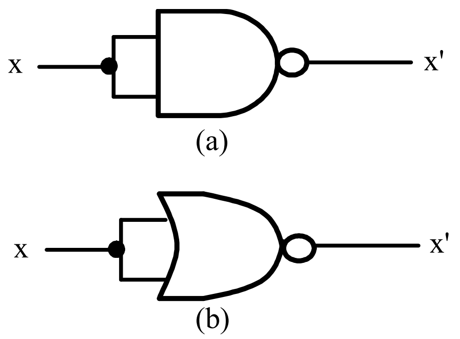


Fig. 2.9 XNOR Gate

NAND and NOR gates are also called **Universal Logic Gates.** The reason being that all the basic logic gates (NOT, AND and OR) can be realized using NAND/NOR gates only.

NOTES

Realization of NOT using NAND and NOR

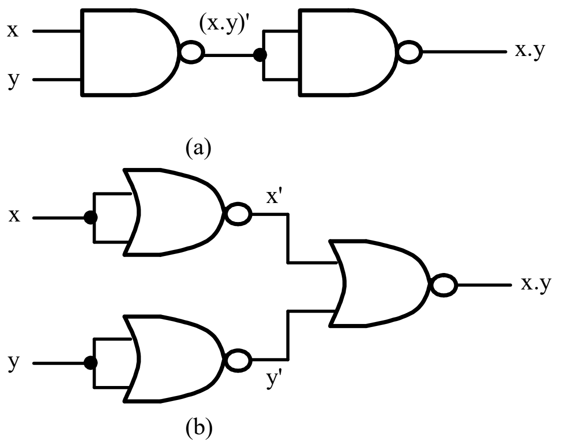


(a)

(b)

Fig. 2.10 (a), (b) *NOT Gate Realization using NAND and NOR Gates*

Realization of AND using NAND and NOR

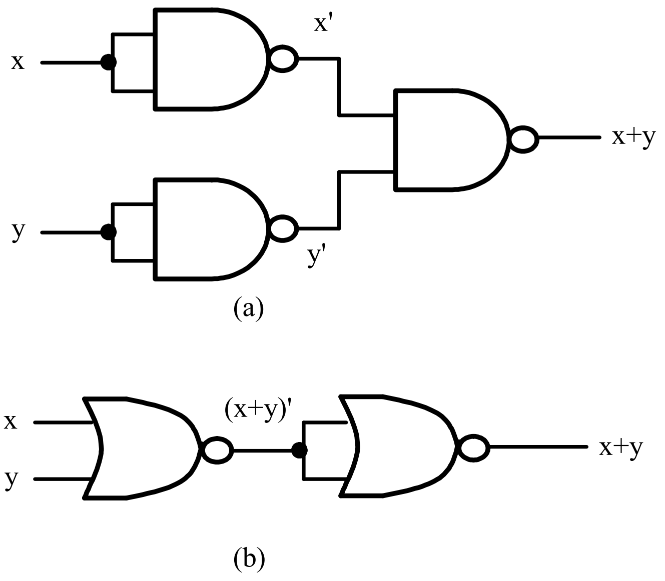


(a)

(b)

Fig. 2.11 (a), (b) *AND Gate Realization using NAND and NOR Gates*

Realization of OR using NAND and NOR



(a)

(b)

Fig. 2.12 (a), (b) *OR Gate Realization using NAND and NOR Gates*

2.3.1 Features of Logic Gates

Similar to the two input gates, you can have more than two inputs to all the logic gates, except the NOT gate. Some points must be remembered while using logic gates. These are as follows:

NOTES

- For multi-input AND and NAND gates, the unused input pin should not be left unconnected. It should be connected either to **Logic High** or to any of the used inputs. This will not affect the overall functionality of logic gate.
- For multi-input OR and NOR gates, the unused input pin should not be left unconnected. It should be connected to **Logic Low**. This will not affect the overall functionality of the logic gate.
- For multi-input XOR gate, the output is Logic High when the total number of Logic High in the inputs signal is **Odd;** otherwise, the output is Logic Low.
- For multi-input XNOR gate, the output is Logic High when the total number of Logic High in the inputs signal is **Even;** otherwise, the output is Logic Low.
- When one of the inputs of two-input XOR gates is Logic High, the output will be NOT of the other input.
- When one of the inputs of two-input XOR gates is Logic Low, the output will be the same as the other input.

2.4 DRAWING LOGIC CIRCUITS

When a Boolean expression is provided, you can easily draw the logic circuit. For example, draw a circuit for $F_1 = ABC'$.

Since the number of inputs is three and one input is in the complemented form, it can be realized by using a three-input AND gate and one NOT gate, which is as follows:

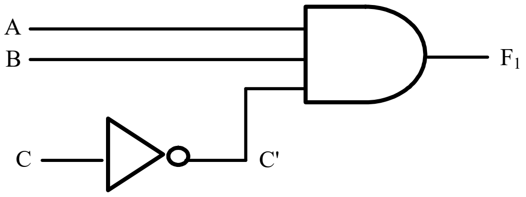


Fig. 2.13 Circuit Diagram for $F_1 = ABC'$

For examples, (i) $F_2 = x + y'z$

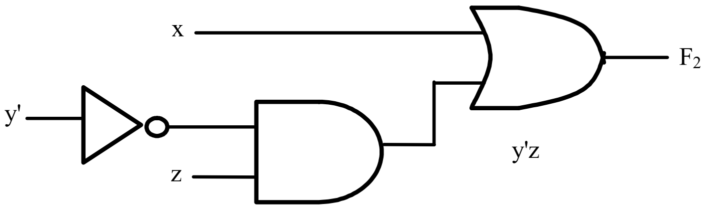


Fig. 2.14 Circuit Diagram for $F_2 = x + y'z$

(ii) $F_3 = xy' + x'z$. Assuming that complemented inputs are available:

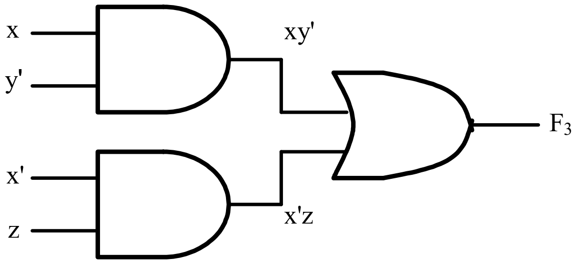


Fig. 2.15 *Circuit Diagram for $F_3 = xy' + x'z$*

2.5 ANALYSING LOGIC CIRCUITS

When a logic circuit diagram is given, you can analyse the circuit to obtain the logic expression. For example, find the expression of F_1 for the following circuit:

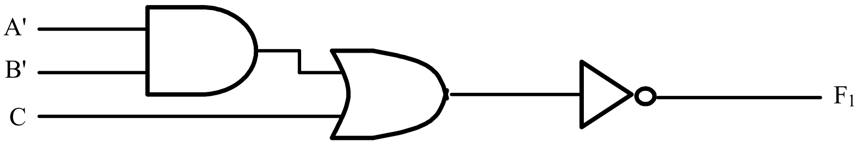


Fig. 2.16 *Analysing Logic Circuit*

You have to start analysing from the input side towards the output side and depending on the logic gate in the path, keep on writing the expression. In the circuit shown in Figure 2.16, , A′ and B′ are applied to the AND gate and the output is given to the OR gate whose second input is C. So you can write the output of the OR gate as **A′B′+C.** The output of the OR gate is given to the NOT gate for getting F_1, so **F_1 = (A′B′+C)′**.

2.6 BOOLEAN ALGEBRA

Boolean algebra is named after George Boole, who used it to study human logical reasoning. For example, any event can be true or false. Similarly, connectives can be of any of the following three basic forms:

1. a OR b
2. a AND b
3. NOT a

Boolean algebra consists of a set of elements B, with two binary operations {+} and {.} and a unary operation {′}, such that the following axioms hold:

- The set B contains at least two distinct elements x and y.
- **Closure**: For every x, y in B,
  - x + y
  - x . y

NOTES

- **Commutative laws**: For every x, y in B,
  - x + y = y + x
  - x . y = y . x
- **Associative laws**: For every x, y, z in B,
  - (x + y) + z = x + (y + z) = x + y + z
  - (x . y) . z = x .(y . z) = x . y . z
- **Identities** (0 and 1):
  - 0 + x = x + 0 = x for every x in *B*
  - 1 . x = x . 1 = x for every x in B
- **Distributive laws**: For every x, y, z in B,
  - x . (y + z) = (x . y) + (x . z)
  - x + (y . z) = (x + y) . (x + z)
- **Complement**: For every x in B, there exists an element x′ in B such that,
  - x + x′ = 1
  - x . x′ = 0

Duality Principle: Every valid Boolean expression (equality) remains valid if the operators and identity elements are interchanged.

+ ↔ .
1 ↔ 0

For example, given the expression,

$$a + (b .c) = (a + b). (a + c)$$

Its dual expression is:

$$a. (b + c) = (a. b) + (a. c)$$

The advantage of this theorem is that if you prove one theorem, the other follows automatically.

For example, if $(x + y + z)' = x'. y.' z'$ is valid, then its dual is also valid:

$$(x. y. z)' = x' + y' + z'$$

Apart from the axioms/postulates, there are other useful theorems. These entire theorems are useful for reducing the expression.

1. **Idempotency**

(a) x + x = x (b) x . x = x

Proof of (a):

$$\begin{aligned} x + x &= (x + x).1 \text{ (Identity)} \\ &= (x + x). (x + x') \text{ (Complement)} \\ &= x + x. x' \text{ (Distributive)} \\ &= x + 0 \text{ (Complement)} \\ &= x \text{ (Identity)} \end{aligned}$$

2. Null elements for '+' and '.' operators

(a) $x + 1 = 1$ (b) $x \,.\, 0 = 0$

3. Involution

$$(x')' = x$$

NOTES

4. Absorption

(a) $x + x \,.\, y = x$ (b) $x \,.\, (x + y) = x$

5. Absorption (variant)

(a) $x + x' \,.\, y = x + y$ (b) $x \,.\, (x' + y) = x.\, y$

6. De Morgan

(a) $(x + y)' = x' \,.\, y'$ (b) $(x \,.\, y)' = x' + y'$

7. Consensus

(a) $x.\, y + x' \,.\, z + y.\, z = x \,.\, y + x' \,.\, z$

(b) $(x + y) \,.\, (x' + z) \,.\, (y + z) = (x + y) \,.\, (x' + z)$

The set B = {0, 1} and the logical operations OR, AND and NOT satisfy all the axioms of Boolean algebra.

A **Boolean expression** is an algebraic statement containing Boolean variables and operators. Theorems can be proved using the truth table method. They can also be proved by an algebraic manipulation using axioms/postulates or other basic theorems.

Check Your Progress

1. What is a NOT gate? Give the truth table for a NOT gate.
2. How is NOT represented?
3. What is an AND gate? Give the truth table for an AND gate.
4. Describe the OR gate and its truth table.
5. What do the '.' and '+' symbols imply?
6. Give the diagrammatic representation of NOT, AND and OR gates.
7. What is a NAND gate?
8. What is an XNOR gate?
9. Explain the duality principle.
10. What is a Boolean expression?

2.7 BOOLEAN FUNCTIONS

A **Boolean function** is an expression formed with binary variables, the two binary operators OR and AND, the unary operator NOT, and the equal and parenthesis signs. Its result is also a binary value. The general usage is '**.**' for AND, '**+**' for OR and '**′**' for NOT.

2.7.1 Precedence of Operators

To lessen the brackets used in writing Boolean expressions, operator precedence can be used. Precedence (highest to lowest): $' \rightarrow . \rightarrow +$

NOTES

For example,

$$a.b + c = (a.b) + c$$

$$b' + c = (b') + c$$

$$a + b'.c = a + ((b').c)$$

In order to avoid confusion, use brackets to overwrite precedence.

2.7.2 Truth Table

A truth table is a table, which consists of every possible combination of inputs and its corresponding outputs.

INPUTS	OUTPUTS
…	…
…	…

For basic logic gates, the truth table is already being discussed. Now, for the complex digital systems, it is very important to derive the truth table.

A truth table describes the behaviour of a system that is to be designed. This is the starting point for any digital system design. A designer must formulate the truth table first. It is the responsibility of the designer to decide the number of output bits to represent the behaviour of the system.

For example, if you have to design a 2-bit multiplier, which multiplies two inputs A and B, each of the two bits, then it should be noted that the output must be at least of 4 bits since the maximum result that you can have from this multiplication is 1001(9) corresponding to the maximum value of both the inputs, i.e., 11(3). The block diagram and the truth table are shown as follows:

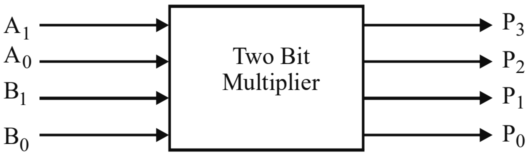


***Fig. 2.17** 2-Bit Multiplier Block Diagram*

In the truth table formation, inputs are taken as A_1A_0 for A input and B_1B_0 for B input. Output resulting from multiplication is to be represented as $P_3P_2P_1P_0$, where P_3 is the MSB and P_0 is the LSB bit. If A = 10, i.e., 2 and B = 11, i.e., 3, then the result of multiplication will be 0110, i.e., 6. So, the bits at the output will be $P_3 = 0$, $P_2 = 1$, $P_1 = 1$, $P_0 = 0$. The complete truth table for the multiplier will be as shown in Table 2.5.

Table 2.5 *Truth Table for 2-Bit Multiplier*

A_1	B_0	B_1	B_0	P_3	P_2	P_1	P_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0
0	0	1	0	0	0	0	0
0	0	1	1	0	0	0	0
0	1	0	0	0	0	0	0
0	1	0	1	0	0	0	1
0	1	1	0	0	0	1	0
0	1	1	1	0	0	1	1
1	0	0	0	0	0	0	0
1	0	0	1	0	0	1	0
1	0	1	0	0	1	0	0
1	0	1	1	0	1	1	0
1	1	0	0	0	0	0	0
1	1	0	1	0	0	1	1
1	1	1	0	0	1	1	0
1	1	1	1	1	0	0	1

After the truth table, you have to write the Boolean expression for the output bit and then realize the reduced expression using logic gates.

Whenever a Boolean expression for any output signal is to be written from the truth table, only those input combinations for which the output is high is to be written. As an example, let us write the Boolean expression for Table 2.6.

Table 2.6 *Truth Table*

x	y	z	F_1	F_2	F_3	F_4
0	0	0	0	0	0	0
0	0	1	0	1	1	1
0	1	0	0	0	0	0
0	1	1	0	0	1	1
1	0	0	0	1	1	1
1	0	1	0	1	1	1
1	1	0	1	1	0	0
1	1	1	0	1	0	0

The Boolean expression for the output F_1 will be $F_1 = x.\ y.\ z'$. This is in the Sum-of-Products form, which will be discussed later.

As can be seen from Table 2.6, output F_1 is 1 only when input xyz is 110. This is represented as $x.\ y.\ z'$. Similarly, you can write the output expression for the rest of the output signals.

$$F_2 = x'.\ y'.\ z + x\ .y'.\ z' + x.\ y'.\ z + x.\ y.\ z' + x.\ y.\ z$$

F_2 can be reduced using Boolean algebra and can be written as follows:

$$\begin{aligned} F_2 &= x'.\ y'.\ z + x\ .y'.\ (z' + z) + x.\ y.\ (z' + z) \\ &= x'.\ y'.\ z + x\ .y' + x.\ y \\ &= x'.\ y'.\ z + x\ .(y' + y) \\ &= x'.\ y'.\ z + x \end{aligned}$$

$= (x' + x).\,(y'.\,z + x)$ (Using **Absorption rule)**

$= \mathbf{1.\,(}y'.\,z + x)$

$= (y'.\,z + x)$

NOTES

Similarly, it can be shown that $F_3 = F_4 = x.\,y' + x'.\,z$

2.7.3 Complement of Functions

For a function F, the **complement** of this function F′ is obtained by interchanging 1 with 0 and vice versa in the function's output values. As an example, take the following function F_1 and its complement, F′:

Table 2.7 *Truth Table of Function and its Complement*

x	**y**	**z**	$\mathbf{F_1}$	$\mathbf{F_1'}$
0	0	0	0	1
0	0	1	0	1
0	1	0	0	1
0	1	1	0	1
1	0	0	0	1
1	0	1	0	1
1	1	0	1	0
1	1	1	0	1

The same can also be verified using the Boolean algebra technique. In Table 2.7, if $F_1 = xyz'$, then its complement will be:

$F_1' = (x.\,y.\,z')'.$

$= x' + y' + (z')'$ (Using De Morgan's theorem)

$= x' + y' + z$

This is the same as that obtained from the truth table by algebraic manipulation, which is given as follows:

$$
\begin{aligned}
F_1' &= x'.y'.z' + x'.y'.z + x'.y.z' + x'.y.z + x.y'.z' + x.y'.z + x.y.z \\
&= x'.y'.(z' + z) + x'.y.(z' + z) + x.y'.(z' + z) + x.y.z \\
&= x'.y' + x'.y + x.y' + x.y.z \\
&= x'.(y' + y) + x.(y' + y.z) \\
&= x' + x.(y' + y.z) \\
&= x' + x(y' + y).\,(y' + z) \\
&= x' + x.\,(y' + z) \\
&= (x' + x).(x' + y' + z\,) \\
&= (x' + y' + z)
\end{aligned}
$$

The following are some more general forms of **De Morgan's Theorems** used for obtaining complement functions:

$$(A + B + C + \ldots + Z)' = A'\mathbf{.}\,B'\mathbf{.}C' \ldots \mathbf{.}\,Z'$$

$$(A\mathbf{.}\,B\mathbf{.}\,C\ldots\mathbf{.}Z)' = A' + B' + C' + \ldots + Z'$$

2.7.4 Standard Forms

NOTES

Certain types of Boolean expressions lead to gating networks, which are desirable from the implementation point of view. The following are two standard forms for writing a Boolean expression:

- Sum-Of-Product (SOP)
- Product-Of-Sum (POS)

Before using SOP and POS forms, you must know the following terms:

- **Literal:** A variable on its own or in its complemented form is known as a literal.

  Examples: x, x', y, y'
- **Product Term**: It is a single literal or a logical product (AND) of several literals.

  Examples: $x, x.y.z', A'.B, A.B$
- **Sum Term**: It is a single literal or a logical sum (OR) of several literals.

  Examples: $x, x + y + z', A'+B, A+B$
- **Sum-Of-Products (SOP) Expression**: It is a product term or a logical sum (OR) of several product terms.

  Examples: $x, x + y. z', x .y' + x'. y. z , A.B+A'.B'$
- **Product-Of-Sum (POS) Expression**: It is a sum term or a logical product (AND) of several sum terms.

  Examples: $x, x.(y + z'), (x +y').(x' + y+ z), (A+B).(A'+B')$

Every Boolean expression can either be expressed as a Sum-Of-Product or Product-Of-Sum expression. For example,

SOP:	$x'.y + x.y' + x.y.z$
POS:	$(x + y').(x' + y).(x' + z')$
Both:	$x' + y + z$ or $x.y.z'$
Neither:	$x.(w' + y.z)$ or $z' + w.x'.y + v.(x.z + w')$

2.7.5 Minterm and Maxterm

Consider two binary variables x, y. Each variable may appear as itself or in the complemented form as literals (i.e., x, x′ and y, y′). For two variables, there are four possible combinations with the AND operator, namely:

$$x'.y', x'.y, x.y' \text{ and } x.y$$

These product terms are called **Minterms**. In other words, A **Minterm** of n variables is the product of n literals from the different variables. In general, n variables can give 2^n Minterms.

Similarly, a **Maxterm** of n variables is the sum of n literals from the different variables.

Examples: $x'+y', x'+y, x+y', x+y$

In general, n variables can give 2^n Maxterms.

The Minterms and Maxterms of 2 variables are denoted by m_0 to m_3 and M_0 to $M_{3,}$ respectively. In Table 2.8, all the Minterms and Maxterms are written.

NOTES

Table 2.8 *Minterms and Maxterms*

Minterms				**Maxterms**	
x	**y**	**Term**	**Notation**	**Term**	**Notation**
0	0	x' .y'	m_0	x + y	M_0
0	1	x' .y	m_1	x + y'	M_1
1	0	x .y'	m_2	x' + y	M_2
1	1	x .y	m_3	x' + y'	M_3

If you examine carefully, each Minterm is the complement of the corresponding Maxterm. For example, $m_2 = x.y'$ and $m_2' = (x.y')' = x' + (y')' = x'+y = M_2$. In other words, **Maxterm is the sum of terms of the corresponding Minterm with its literal complemented.**

2.7.6 Canonical Form: Sum of Minterms

Canonical form is a unique way of representing Boolean expressions. Any Boolean expression can be written in the form of the sum of Minterm. A Σ symbol is used for showing the sum of Minterms. For example,

Table 2.9 *Sum of Minterms*

x	**y**	**x**	**F_1**	**F_2**	**F_3**
0	0	0	0	0	0
0	0	1	0	1	1
0	1	0	0	0	0
0	1	1	0	0	1
1	0	0	0	1	1
1	0	1	0	1	1
1	1	0	1	1	0
1	1	1	0	1	0

Sum-of-Minterms by gathering/summing the Minterms of the function (where result is a 1) can be obtained as follows:

$$F_1 = x.y.z' = \Sigma m\ (6)$$

$$F_2 = x'.y'.z + x.y'.z' + x.y'.z + x.y.z' + x.y.z = \Sigma m(1,4,5,6,7)$$

$$F_3 = x'.y'.z + x'.y.z + x.y'.z' + x.y'.z = \Sigma m(1,3,4,5)$$

2.7.7 Canonical Form: Product of Maxterms

Maxterms are sum terms. For Boolean functions, the Maxterms of a function are the terms for which the result is 0. Boolean functions can be expressed as Products-of-Maxterms. For Table 2.9, each output F_1, F_2 and F_3 can be represented in Product-of-Maxterm. A Π symbol is used to represent Product-of-Maxterms.

$$F_1 = (x + y + z).(x + y + z').(x + y' + z).(x + y'+ z').(x' + y + z)$$
$$.(x' + y + z').(x' + y' + z')$$
$$= \Pi\ M(0,1,2,3,4,5,7)$$

$$F_2 = (x + y + z).(x + y' + z).(x + y' + z')$$
$$= \Pi\ M(0,2,3)$$
$$F_3 = (x + y + z).(x + y' + z).(x' + y' + z).(x' + y' + z')$$
$$= \Pi\ M(0,2,6,7)$$

2.7.8 Conversion of Canonical Forms

Sum-of-Minterms ⇒ Product-of-Maxterms

- Rewrite Minterm shorthand using Maxterm shorthand.
- Replace Minterm indices with indices not already used.

For example, $F_1(x,y,z)= \sum m(6) = \prod M(0,1,2,3,4,5,7)$.

Product-of-Maxterms ⇒ Sum-of-Minterms

- Rewrite Maxterm shorthand using Minterm shorthand.
- Replace Maxterm indices with indices not already used.

For example, $F_2(x,y,z)= \prod M(0,2,3) = \sum m(1,4,5,6,7)$.

Sometimes, you are given the reduced expression for any Boolean expression. In this case, you need to find Minterms or Maxterms present in the expression. To convert from a general expression to a Minterm or Maxterm expression, you can use either the truth table or the algebraic manipulation.

For example, suppose you wish to find all the Minterm expansions of $F = AB' + A'C$.

The truth table for the expression is represented as shown in Table 2.10:

Table 2.10

A	B	C	F
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

From the Table 2.10, $F = A'.B'.C + A'.B.C + A.B'.C' + A.B'.C$

$$= \Sigma m\ (1, 3, 4, 5)$$

Using Algebraic Manipulation

Use $X + X' = 1$ to introduce the missing variables in each term; this introduction will not change the overall expression value. Therefore, for the Boolean expression $F = AB' + A'C$, the missing variable in the first term is C and in the second term is B. So, the missing variable can be introduced as follows:

$$= A.B'.(C + C') + A'.C.(B + B')$$
$$= A.B'.C + A.B'.C' + A'.B.C + A'.B'.C$$
$$= m_5 + m_4 + m_3 + m_1$$
$$= \sum m\ (1, 3, 4, 5)$$

Similarly, you can find all the Maxterms for reduced expressions**.** Find the Maxterms expansion of $F = (A + B')\,(A' + C)$

Using Algebraic Expression: In this case, $XX' = 0$ is used to introduce missing variables in each term.

Therefore, $\mathbf{F = (A + B' + CC').\ (A' + C + BB')}$

Assuming that $(A + B') = X$ and $C.C' = YZ$, you can use the expression rule

$= \mathbf{X+YZ=(X+Y)(X+Z)}$

$$F = (A + B'+C)(A+B'+C')(A'+B+C)(A'+B'+C)$$

$$= \prod(2, 3, 4, 6)$$

Using the Truth Table:

A	B	C	F
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

$$F\,(A,B,C) = \prod(2, 3, 4, 6)$$

2.7.9 Boolen Algebra as Lattices

Let B be a non-empty set with two binary operations + (or $\vee$) and, (or $\wedge$), a unary operation, and two distinct elements 0 and 1. Then B is called a Boolean algebra if the following axioms hold wher a,b, c are any elements in B.

(i) $a + b = b + a$; $a . b = b. a$ (commutative laws)

(ii) $a + (b . c) = (a + b) . (a + c)$; $a.(b + c) = (a . b)\,(a . c)$ (Distributie laws)

(iii) $a + 0 = a$; $a . 1 = a$ (Identity laws)

(iv) $a + a' =1$; $a . a' = 0$ (Complement laws)

Boolean algebra is a lattice which contains a least element and a greatest element and which is both complemented and distributive.

We denote the Boolean algebra B by (B, +, ., 1, 0, 1). Here we call 0 as the zero element, 1 as the unit element, and a′ is complement of a, + and . are called sum and product.

Let B = {0,1}, the set of binary digits with the binary operations of + and . and the unary operation defined by

+	1	0
1	1	1
0	1	0

·	1	0
1	1	0
0	1	0

′	1	0
	0	1

Then B is a Boolean algebra.

2.7.10 Atom

A non zero element ‘a’ in a Boolean algebra (B, +, .,′) is called an atom if for every $x \in B$, $x \wedge a = a$ or $x \wedge a = 0$.

Note: Here the condition $x \wedge a = a$ means that x is a successor of a and $x \wedge a = 0$ is true only when x and a are ‘not connected’. So in any Boolean algebra, the immediate successors of the 0-element are called atoms.

Let A be any non-empty set and P(A) the power set of A. In Boolean algebra (p(A), ∪, ∩, ′) over ⊆, the singleton sets are the atoms since each element p(A) can be described completely and uniquely as the union of singleton sets.

Let B = {1,2, 3, 5, 6, 10, 15, 30} and let the relation ≤ be divides. The operation $\wedge$ is GCD and $\vee$ is LCM. The 0-element is 1. Then the set of atoms of the Boolean algebra is {2,3,5}.

Notes:

1. Let (B, +, . ,′) be any finite Boolean algebra and let A be the set of all atoms. Then (B, +,., ′) is isomorphic to (p(A), ∪, ∩, ′).
2. Every finite Boolean algebra (B,+,.,′) has 2^n elements for some position integer n.
3. All Boolean algebra of order 2^n are isomorphic to each other. Finite Boolean algebras are n-tuples of 0′s and 1′s.

The simplest nontrivial Boolean algebra is the Boolean algebra B = {0, 1}, the set of binary digits with the binary operations of + and . and the unary operation ′ given by,

+	1	0
1	1	1
0	1	0

·	1	0
1	1	0
0	1	0

′	1
1	0
0	1

If we form $B^2 = B \times B$, we obtain the set B^2={(0,0), (0,1), (1, 0), (1, 1)}. Define +, . and ′ by

$$(0, 1) + (1, 1) = (0 + 1, 1+1) = (1, 1),$$

$$(0, 1).(1,1) = (0.1, 1.1) = (0, 1) \text{ and}$$

$$(0,1)' = (0', 1') = (1, 0).$$

The B^2 is a Boolean algebra.

Note: Here B^2 is a Boolean algebra of order 4 under component wise operations. Since all Boolean algebra of order 4 are isomorphic to each other, this is a simple way of describing all Boolean algebras of order 4. In general, any Boolean algebra of order 2^n are isomorphic to B^n.

Example 2.1: Find the atoms of the Boolean algebra (i) B^2 (ii) B^4 (iii) B^n for $n \geq 1$.

Solution:

(i) (0, 1) and (1, 0)

(ii) (1,0,0,0), (0,1,0,0), (0,0,1,0) and (0,0,0,1)

(iii) The n-tuples with exactly one 1.

2.8 FUNCTION SIMPLIFICATION

Sometimes, you need to simplify the Boolean expression. The main advantage in doing so is that, it then uses less logic gates and less power to realize and thus, it is considered sometimes cheaper and faster.

There are basically two types of simplification techniques:

- Algebraic Simplification
- Karnaugh Maps (K-map)

2.8.1 Algebraic Simplification

This involves simplifications using Boolean theorems. Algebraic simplification aims to minimize the number of literals and terms.

For example, to reduce the Boolean expression $F = (x+y).(x+y').(x'+z)$

(6 literals)

$$\begin{aligned}
F &= (x.x + x.y' + x.y + y.y').(x'+z) && \text{(Associative)}\\
&= (x+x.(y'+y)+0).(x'+z) && \text{(Idempotency, Associative, Complement)}\\
&= (x+x.(1)+0).(x'+z) && \text{(Complement)}\\
&= (x+x+0).(x'+z) && \text{(identity 1)}\\
&= (x).(x'+z) && \text{(Idempotency, Identity 0)}\\
&= (x.x'+x.z) && \text{(Associative)}\\
&= (0+x.z) && \text{(Complement)}\\
&= x.\ z && \text{(Identity 0)}
\end{aligned}$$

Number of literals reduced from 6 to 2.

For example,

1. Finding the minimal SOP and POS expressions of :

$$\begin{aligned}
F(x,y,z) &= x'.y.(z + y'.x) + y'.z\\
&= x'.y.z + x'.y.y'.x + y'.z && \text{(Distributive)}\\
&= x'.y.z + 0 + y'.z && \text{(Complement, Null element 0)}\\
&= x'.y.z + y'.z && \text{(Identity 0)}\\
&= x'.z + y'.z && \text{(Absorption)}\\
&= (x' + y').z && \text{(Distributive)}
\end{aligned}$$

Minimal SOP of $F = x'.z + y'.z$ (Two 2-input AND gates and one 2-input OR gate)

Minimal POS of $F = (x' + y').z$ (One 2-input OR gate and one 2-input AND gate)

2. Finding the minimal SOP expression of:

$$\begin{aligned}
F(a,b,c,d) &= a.b.c + a.b.d + a'.b.c' + c.d + b.d'\\
&= a.b.c + \underline{a.b.d} + a'.b.c' + c.d + \underline{b.d'} && \text{(Absorption on underlined terms)}\\
&= a.b.c + \underline{a.b} + \underline{a'.b.c'} + c.d + b.d' && \text{(Absorption on underlined terms)}
\end{aligned}$$

$= \underline{a.b.c} + \underline{a.b} + b.c' + c.d + b.d'$ (Absorption on underlined terms)

$= a.b + \underline{b.c'} + c.d + \underline{b.d'}$ (Distributivity on underlined terms)

$= a.b + c.d + b.(\underline{c'} + \underline{d'})$ (DeMorgan on underlined terms)

$= a.b + \underline{c.d} + \underline{b.(c.d)'}$ (Absorption on underlined terms)

$= \underline{a.b} + c.d + \underline{b}$ (Absorption on underlined terms)

$= b + c.d$

Number of literals is reduced from 13 to 3.

However, the difficulty with this method is that it needs good algebraic manipulation skills.

2.8.2 Karnaugh Map

It is a diagram-based simplification technique. It is easy for the circuit designer and involves pattern-matching skills. It gives simplified Boolean expressions in standard forms. However, this can be effectively utilized for reducing Boolean expressions with input variables less than 6.

It is a systematic method to obtain simplified Sum-Of-Products (SOP) Boolean expressions with the objective of fewest possible terms/literals. It is a diagrammatic technique based on a special form of Venn diagram. Here, Venn diagrams represent the space of Minterms**.** An example of a 2 variable (4 Minterms) Venn diagram is shown in Figure 2.18.

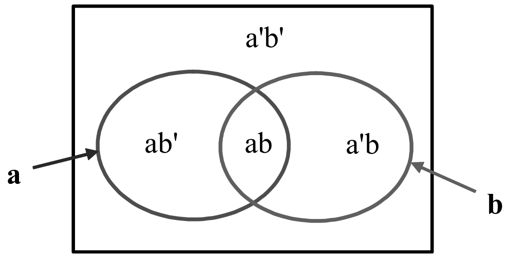


Fig. 2.18 *Venn Diagram (4 Minterms)*

Each set of Minterms represents a Boolean function. For example,

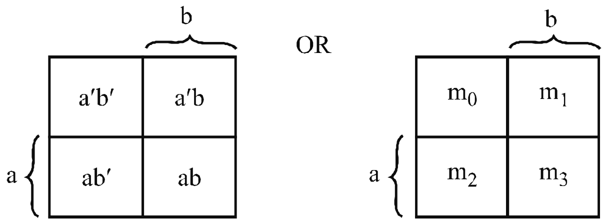
$\{a.b, a.b'\}$	→	$a.b + a.b' = a.(b+b') = a$
$\{a'.b, a.b\}$	→	$a'.b + a.b = (a'+a).b = b$
$\{a.b\}$	→	$a.b$
$\{a.b, a.b', a'.b\}$	→	$a.b + a.b' + a'.b = a + b$
$\{\}$	→	0
$\{a'.b', a.b, a.b', a'.b\}$	→	1

2-Variable K-Map

A Karnaugh map (K-map) is an abstract form of Venn diagram, organized as a matrix of squares, where each square represents a Minterm. Also, adjacent squares always differ by just one literal (so that the unifying theorem may apply: $a + a' = 1$). For 2-variable case (e.g., variables a, b), assuming that **a** is the MSB and **b** is the LSB, the map can be drawn as follows:

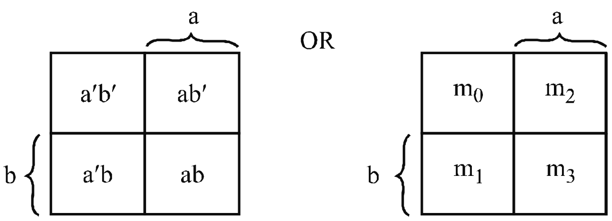
NOTES

Alternative 1:



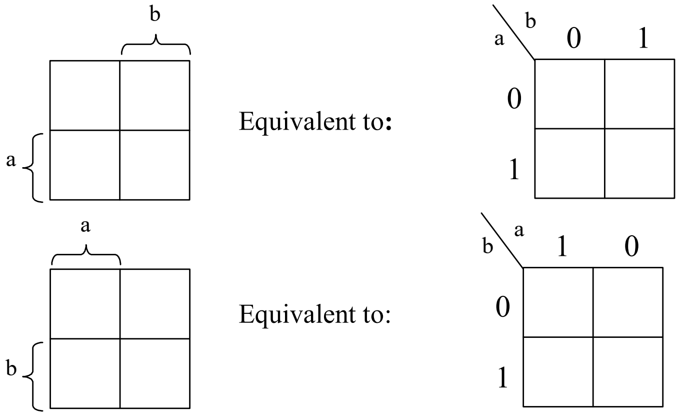
{–symbol implies that corresponding literal is in normal form.

Alternative 2:



{–symbol implies that corresponding literal is in normal form.

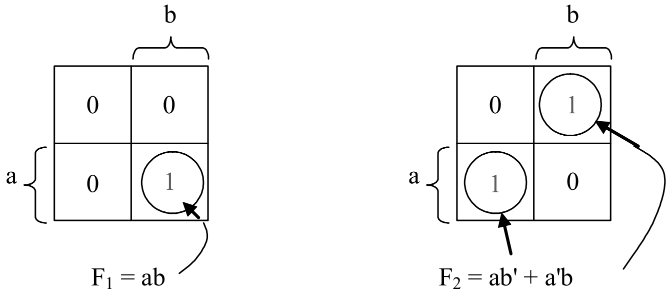
Equivalent Labelling:



The K-map for a function, which is the sum of Minterms, is specified by putting:

- '1' in the square corresponding to a Minterm
- '0' otherwise

For example, if $F_1 = a.b$ and $F_2 = a'.b + a.b'$, then the K-map entry for F_1 and F_2 will be as follows:

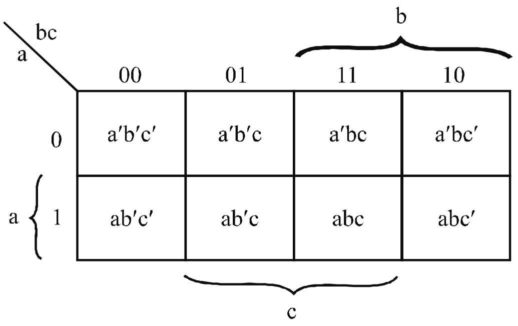


Here 1 is entered to the locations of Minterms of Boolean expression.

3-Variable K-Map

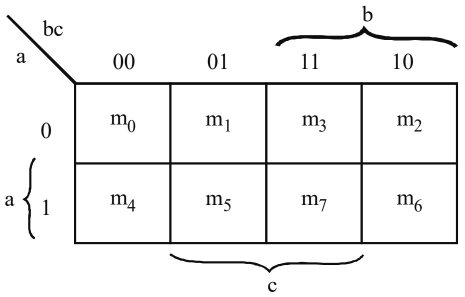
There are 8 Minterms for 3 variables (a, b, c). Therefore, there are 8 cells in a 3-variable K-map.

NOTES



a \ bc	00	01	11	10
0	a′b′c′	a′b′c	a′bc	a′bc′
1	ab′c′	ab′c	abc	abc′

It is to be noted that the above arrangement ensures that Minterms of adjacent cells differ by only *one literal*.



a \ bc	00	01	11	10
0	m_0	m_1	m_3	m_2
1	m_4	m_5	m_7	m_6

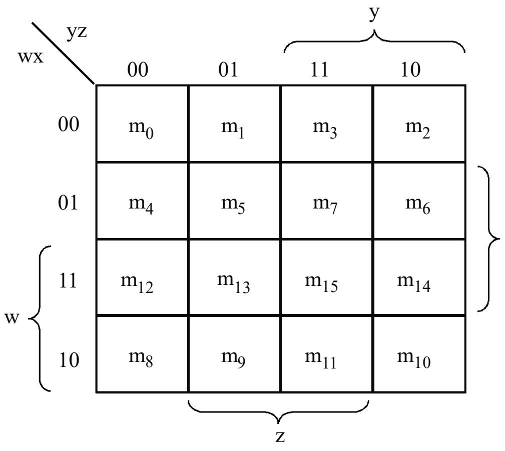
It is to be noted that there is wrap-around in the K-map:

- a′.b′.c′ (m_0) is adjacent to a′.b.c′(m_2) since only one literal **b** is different.
- a.b′.c′ (m_4) is adjacent to a.b.c′ (m_6) since only one literal **b** is different.

Each cell in a 3-variable K-map has 3 adjacent neighbours. In general, each cell in an n-variable K-map has n adjacent neighbours. For example, m_0 has 3 adjacent neighbours m_1, m_2 and m_4.

4-Variable K-Map

There are 16 cells in a 4-variable (w, x, y, z) K-map. The K-map for the same is given as follows:



wx \ yz	00	01	11	10
00	m_0	m_1	m_3	m_2
01	m_4	m_5	m_7	m_6
11	m_{12}	m_{13}	m_{15}	m_{14}
10	m_8	m_9	m_{11}	m_{10}

Every cell thus has 4 neighbours. For example, the cell corresponding to Minterm m_0 has neighbours m_1, m_2, m_4 and m_8.

2.8.3 Steps for Forming Karnaugh Map

The K-map of a function is easily drawn when the function is given in canonical Sum-of-Products form or Sum-of-Minterms form. When the function is not in the

Sum-of-Minterms form, then first convert it to Sum-Of-Products (SOP) form. Expand the SOP expression into Sum-of-Minterms expression or fill in the K-map directly based on the SOP expression.

NOTES

To Summarize:

- Find all the Minterms of the function using the method already discussed.
- Fill '1' for the Minterms in the appropriate location.
- Fill '0'Otherwise.

Example 2.2: Draw the K-map for the function F:

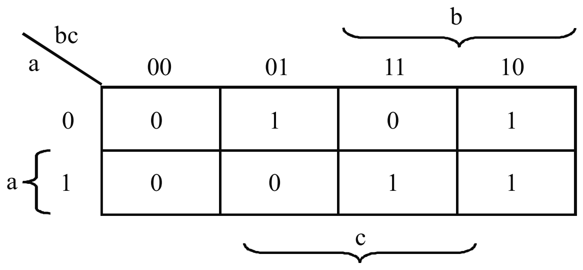
$$F(a, b, c) = a.b + b.c' + a'.b'.c$$

Solution: Find all the Minterms.

$$\begin{aligned} F(a, b, c) &= a.b(c + c') + b.c'(a + a') + a'.b'.c \\ &= a.b.c + a.b.c' + b.c'.a + b.c'.a' + a'.b'.c \end{aligned}$$

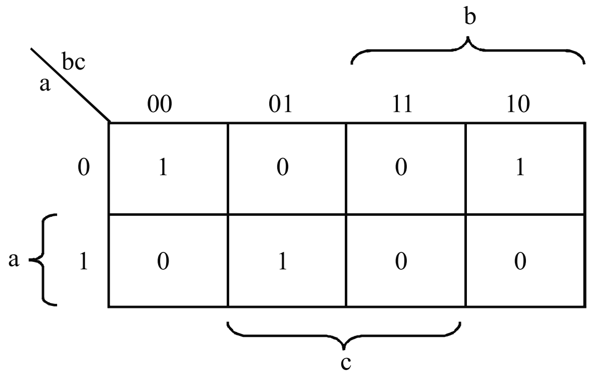
Rearranging the terms with the MSB first and then the next bit up to the LSB, and removing repeated Minterms, you get,

$$F(a, b, c) = a.b.c + a.b.c' + a'b.c' + a'.b'.c = \Sigma\ m(1,2,6,7)$$



a \ bc	00	01	11	10
0	0	1	0	1
1	0	0	1	1

Example 2.3: The K-map of a 3-variable function F is as follows.



a \ bc	00	01	11	10
0	1	0	0	1
1	0	1	0	0

What is the sum-of-Minterms expression of F?

Solution: Assuming that **a** is the MSB and **c** is the LSB and function is of the form F (a, b, c), then by seeing the entry of 1, you can say that Minterms are m_0, m_2 and m_5. So,

$$F = \Sigma\ m\ (0, 2, 5) = a'.b'.c' + a'.b.c' + a.b'.c$$

2.8.4 Simplification of Expressions using Karnaugh Map

Once the K-Map for any Boolean expression is known, it can be used to find the minimized expression, which consists of less number of literals. The main advantage

NOTES

of reduction is that it needs less hardware in terms of logic gate. Less number of literals gives realization based on logic gate with less input pin.

The K-map based Boolean reduction is based on the following Unifying Theorem:

$$A + A' = 1$$

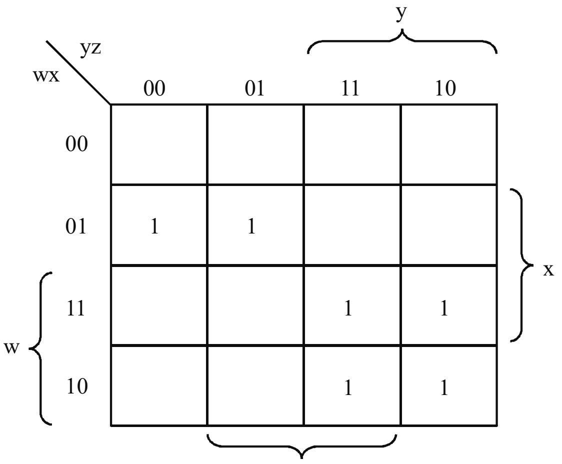
In a K-map, each cell containing a '1' corresponds to a Minterm of a given function F. Each group of adjacent cells containing '1' (a group must have size *in powers of twos*: 1, 2, 4, 8, …) then corresponds to a simpler product term of F.

- Grouping 2 adjacent squares eliminates 1 variable, grouping 4 squares eliminates 2 variables, grouping 8 squares eliminates 3 variables and so on. In general, grouping 2^n squares eliminates n variables.
- Group as many squares as possible. The larger the group, the fewer the number of literals in the resulting product term.
- Select as few groups as possible to cover all the squares (Minterms) of the function. The fewer the groups, the fewer the number of product terms in the minimized function.

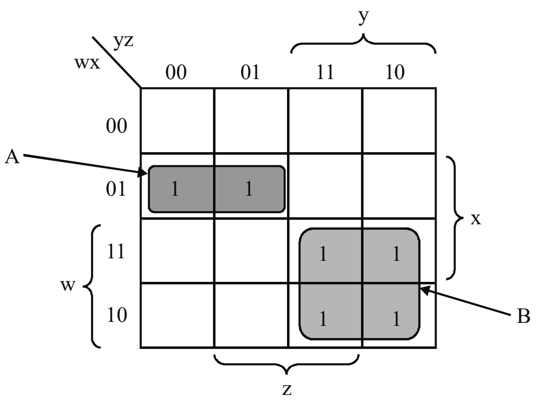
Example 2.4: Find the reduced expression for the function given by:

$$\begin{aligned} F(w,x,y,z) &= w'.x.y'.z' + w'.x.y'.z + w.x'.y.z' + w.x'.y.z + w.x.y.z' + w.x.y.z \\ &= \Sigma\, m(4, 5, 10, 11, 14, 15) \end{aligned}$$

Solution: First draw the K-map. Cells with '0' are not shown for clarity.



Each group of adjacent Minterms (group size in powers of twos) corresponds to a possible product term of the given function.



NOTES

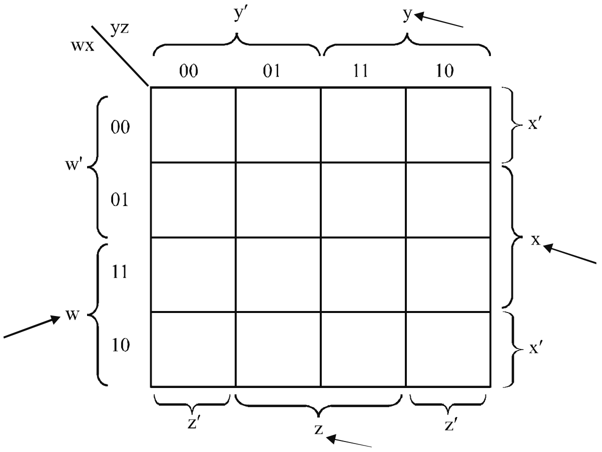
There are 2 groups of Minterms: A and B, where:

$$\begin{aligned} A &= w'.x.y'.z' + w'.x.y'.z \\ &= w'.x.y'.(z' + z) \\ &= w'.x.y' \\ B &= w.x'.y.z' + w.x'.y.z + w.x.y.z' + w.x.y.z \\ &= w.x'.y.(z' + z) + w.x.y.(z' + z) \\ &= w.x'.y + w.x.y \\ &= w.(x' + x).y \\ &= w.y \end{aligned}$$

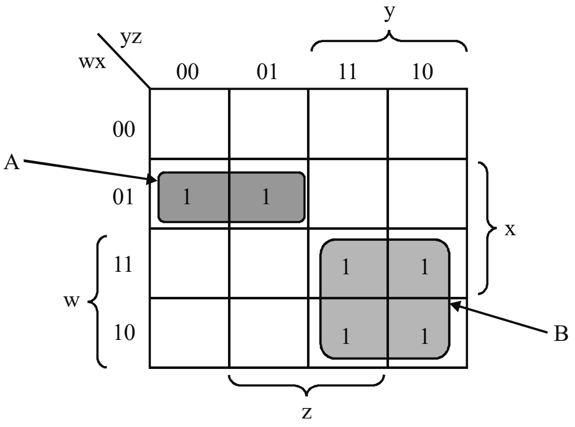
Each product term of a group, $w'.x.y'$ and $w.y$, represents the *Sum-of-Minterms* in that group. The Boolean function is, therefore, the Sum-Of-Product terms (SOP), which represents all groups of the Minterms of the function.

$$F(w,x,y,z) = A + B = w'.x.y' + w.y$$

Another way of getting the expression for the groups A and B is based on the intersection area concept. For example, take a look at four variables of K-map given as follows:



The notation w pointed to by an arrow shows that the complete region has Minterms in which w is 1. The region above w shows w′ region. Similarly, the notation x pointed by the arrow shows that the complete region is having Minterms in which x is 1 and the region above and below x is termed as x′. The same is true for y and z. Using this technique, the K-map shown in the previous example can be solved directly.

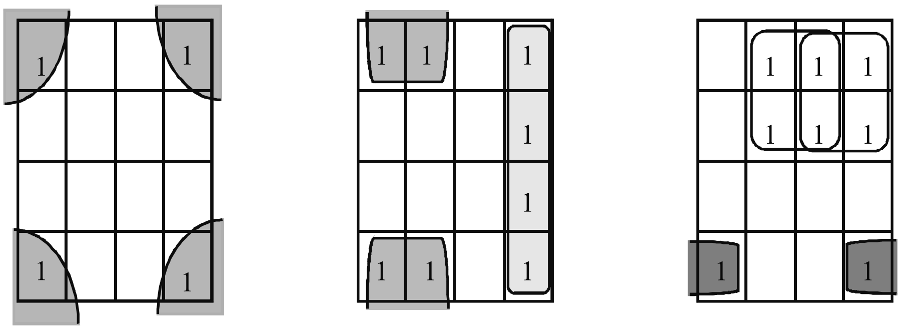


NOTES

The intersection area A shows intersection of w′, x and y′. So, the Boolean expression for the region A can be written as w′.x .y′. Similarly, region B is the intersection of y, w and the Boolean expression for B = w.y; so the overall expression can be written as follows:

$$F(w,x,y,z) = A + B = w'.x.y' + w.y.$$

Larger groups correspond to product terms of fewer literals. In the case of a 4-variable K-map, if you have 1 cell, then you have 4 literals; if you have 2 cells, then 3 literals; if 4 cells, then 2 literals; if 8 cells, then 1 literal and at last, if 16 cells, then no literal. Also, some other possible valid groupings of a 4-variable K-map are shown as follows:



2.8.5 Join-Irreducible Element

In a lattice, an element *x* is join-irreducible if *x* is not the join of a finite set of other elements. Equivalently, *x* is join-irreducible if it is neither the bottom element of the lattice (the join of zero elements) nor the join of any two smaller elements. For example, in the lattice of divisors of 120, there is no pair of elements whose join is 4, so 4 is join-irreducible. An element *x* is join-prime if it differs from the bottom element, and whenever $x \le y \vee z$, either $x \le y$ or $x \le z$. In the same lattice, 4 is join-prime: whenever LCM(*y*, *z*) is divisible by 4, at least one of *y* and *z* must itself be divisible by 4.

In any lattice, a join-prime element must be join-irreducible. Equivalently, an element that is not join-irreducible is not join-prime. For, if an element *x* is not join-irreducible, there exist smaller *y* and *z* such that $x = y \vee z$. But then $x \le y \vee z$, and *x* is not less than or equal to either *y* or *z*, showing that it is not join-prime.

There exist lattices in which the join-prime elements form a proper subset of the join-irreducible elements, but in a distributive lattice the two types of elements coincide. For example, suppose that *x* is join-irreducible, and that $x \le y \vee z$. This inequality is equivalent to the statement that $x = x \wedge (y \vee z)$, and by the distributive law $x = (x \wedge y) \vee (x \wedge z)$. But since *x* is join-irreducible, at least one of the two terms in this join must be *x* itself, showing that either $x = x \wedge y$ (equivalently $x \le y$) or $x = x \wedge z$ (equivalently $x \le z$).

The lattice ordering on the subset of join-irreducible elements forms a partial order, Birkhoff's theorem states that the lattice itself can be recovered from the lower sets of this partial order. In any partial order, the lower sets form a lattice in which the lattice's partial ordering is given by set inclusion, the join operation corresponds to set union, and the meet operation corresponds to set intersection, because unions and intersections preserve the property of being a lower set. Because set unions and intersections obey the distributive law, this is a distributive lattice. Birkhoff's theorem states that any finite distributive lattice can be constructed in this way.

Check Your Progress

11. What is a Boolean function? Why are operators used in an expression?
12. What is a truth table?
13. How is the complement of a function represented?
14. Explain the standard form of writing Boolean expressions.
15. Explain Minterm with the help of an example.
16. Define the types of simplification techniques.
17. What is a K-map? Why is it used?
18. What is the advantage of reduction in the simplification of expression using a K-map?
19. Define the join-irreducible element.

2.9 DON'T CARE CONDITIONS

As you know, a function with n variable in an input can have 2^n possible combinations. If you have four variables in an input, then you have 16 possible combinations for the input. In designing some digital systems, it may happen that for some input combination(s), a designer may not be interested in output(s) corresponding to those input combinations. In these cases, the outputs can be either '1' or '0'. They are called **don't care conditions**, denoted by X (or sometimes, d).

How to represent Boolean Expression with the Don't Care Term

If any Boolean expression has the following Minterms (m_0, m_3, m_7, m_9) and the don't care term is (m_{11}, m_{12}), then it will be represented as follows:

$$F = \Sigma m\,(0, 3, 7, 9) + d\,(11, 12)$$

For example, a truth table for a system, which is taking BCD as an input and generating output P = 1 whenever the number of 1's in the BCD input is even would be as follows:

Table 2.11 *BCD Range*

No.	A	B	C	D	P
0	0	0	0	0	1
1	0	0	0	1	0
2	0	0	1	0	0
3	0	0	1	1	1
4	0	1	0	0	0
5	0	1	0	1	1
6	0	1	1	0	1
7	0	1	1	1	0
8	1	0	0	0	0
9	1	0	0	1	1
10	1	0	1	0	x
11	1	0	1	1	x
12	1	1	0	0	x
13	1	1	0	1	x
14	1	1	1	0	x
15	1	1	1	1	x

As there are four input bits, you can have 16 possible combinations ranging from 0000 to 1111. As you know, valid BCD inputs range from 0000 to 1001. The output corresponding to the valid BCD range is listed in Table 2.11. The input range from 1010 to 1111 is an invalid BCD, so the output is not of much concern for the designer. So these output are shown as ‘X’, which means that the output is a don’t care output.

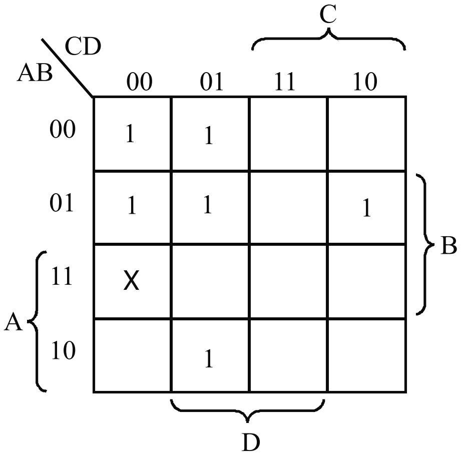
$$F = \Sigma m(0,1,2,3,4,5,6,7,8,9) + d(10,11,12,13,14,15)$$

Don’t care conditions can be used for simplifying Boolean expressions in K-maps. The don’t care value could be chosen to be either ‘1’ or ‘0’, depending on which gives the simpler expression. Therefore, don’t care terms help in giving the reduced expression without affecting the overall behaviour of the desired system. So, it is the responsibility of the designer to judge whether don’t care should be ‘1’ or be ‘0’.

It should be kept in mind that don’t care terms should be used along with the terms that are present in Minterms. It may happen that some of the don’t care terms are assigned as ‘1’ while others are kept as ‘0’.

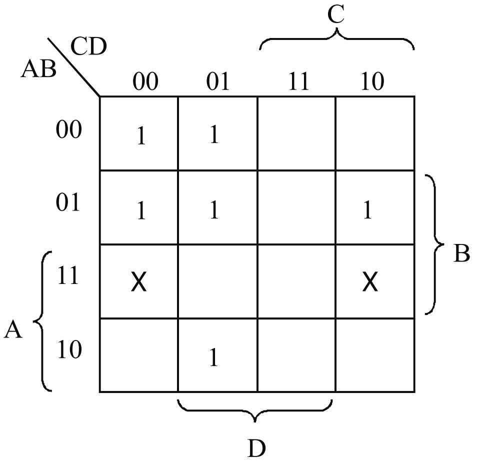
In the following K-map, putting the don’t care term as ‘0’ is beneficial because putting 1 is not helping in reduced expression.

NOTES



AB \ CD	00	01	11	10
00	1	1		
01	1	1		1
11	X			
10		1		

If you are putting the don't care term as '1', then you are getting four product terms formed by Minterms (m_0, m_1, m_4, m_5), (m_4, m_{12}), (m_4, m_6) and (m_1, m_9). If you are putting don't care terms as '0', then the following K-map will result:



AB \ CD	00	01	11	10
00	1	1		
01	1	1		1
11	X			X
10		1		

The final expression will consist of three product terms formed by Minterms (m_0, m_1, m_4, m_5), (m_4, m_6) and (m_1, m_9). So this will give the reduced expression. Take another example of K-map as shown in Table 2.12. Putting the don't care term at Minterm location m_{12} and m_{14} as '1' is beneficial so that it can be combined with Minterm m_4 and m_6 to form a group of 4 adjacent cells.

Table 2.12 *Truth Table with Don't Care Term*

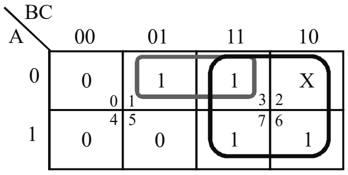
A	B	C	Z
0	0	0	0
0	0	1	1
0	1	0	X
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

NOTES

Example 2.6: For the truth table shown in Table 2.12 with the don't care term, find the K-map and then find the reduced Boolean expression.

Solution:

1. The K-map is as follows:



A \ BC	00	01	11	10
0	0	1	1	X
1	0	0	1	1

2. Minterms presentation:

$$F\ (A, B, C) = \Sigma m\ (1, 3, 6, 7) + d\ (2)$$

3. Maxterms presentation:

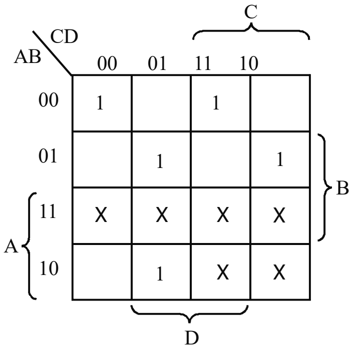
$$F\ (A, B, C) = \Pi M\ (0, 4, 5) + d\ (2)$$

It is to be noted that the don't care term is appearing with both the presentations.

4. The reduced expression will be obtained when the don't care term is taken as '1' since this will result in grouping with other adjacent '1's to form a grouping of 4 cells, causing two variables to get eliminated.

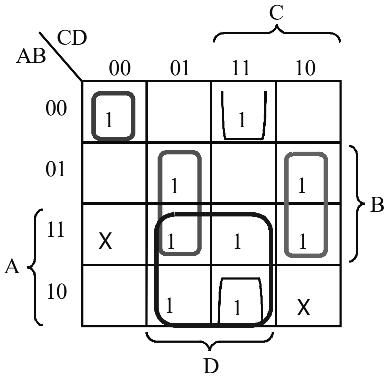
$$F(A, B, C) = B + A'C$$

Example 2.7: Find the reduced expression for the K-map shown as follows:



AB \ CD	00	01	11	10
00	1		1	
01		1		1
11	X	X	X	X
10		1	X	X

Solution: It should be noted that don't care terms at Minterm locations m(11,13,14,15) are being taken as '1' while the rest of the don't care terms are kept at '0'. Therefore, you have the following K-map for grouping.



AB \ CD	00	01	11	10
00	1		1	
01		1		1
11	X	1	1	1
10		1	1	X

2.10 REPRESENTATION OF SIMPLIFIED EXPRESSIONS USING NAND/NOR GATES

NOTES

It is possible to implement any Boolean expression using NAND gates. The following procedure is to be followed:

- Obtain Sum-Of-Products of Boolean expression: e.g., $F_3 = xy'+x'z$
- Use De Morgan theorem to obtain expression using 2-level NAND gates.

e.g., $F_3 = xy'+x'z$

$= \{(xy'+x'z)'\}'$ Involution

$= ((xy')'.(x'z)')'$ De Morgan theorem

Implement this modified expression using NAND gate.

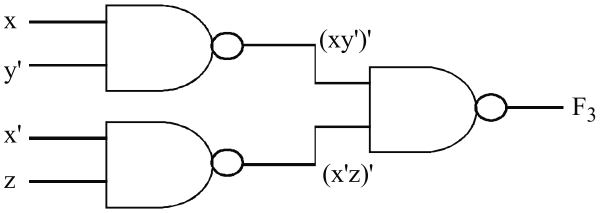


Fig. 2.19 Implementation Using NAND Gate

It is also possible to implement any Boolean expression using NOR gates. The following procedure is to be followed:

(i) Obtain Product-Of-Sums Boolean expression: e.g., $F_6 = (x+y').(x'+z)$

(ii) Use DeMorgan theorem to obtain expression using 2-level NOR gates

e.g., $F_6 = (x+y').(x'+z)$

$= ((x+y').(x'+z))''$ Involution

$= ((x+y')'+(x'+z)')'$ De Morgan theorem

Implement this modified expression using NOR gate.

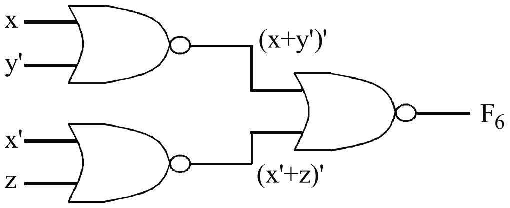
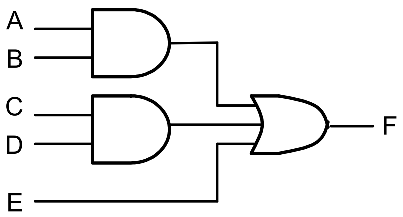


Fig. 2.20 Implementation Using NOR Gate

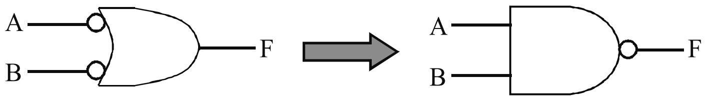
2.10.1 Implementation of SOP Expressions

Sum-of-Products (SOP) expressions can be implemented using either (1) 2-level AND-OR logic circuits or (2) 2-level NAND logic circuits.

(1) 2-level AND-OR logic circuit: F = AB + CD + E



It can be proved that the OR gate with all its input complemented is equivalent to the AND gate with bubble at the output, i.e., NAND gate.

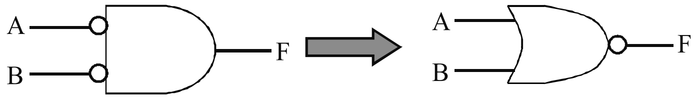


Proof: With OR gate

$$F = A' + B'$$

$$= (A.B)' \text{ De Morgan theorem}$$

Similarly, it can be proved that the AND gate with all its input complemented is equivalent to the OR gate with bubble at the output, i.e., NOR gate.



Proof: With AND gate

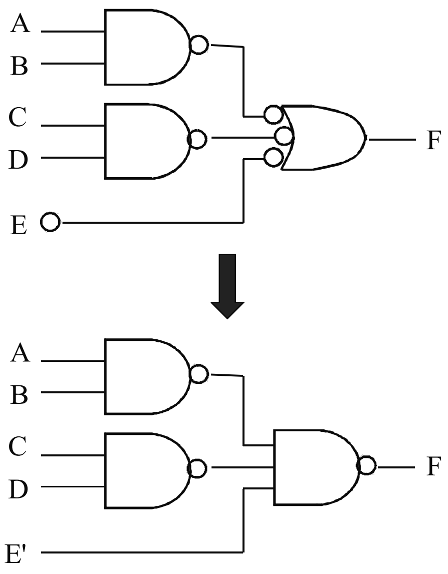
$$F = A'.\, B'$$

$$= (A+B)' \text{ De Morgan theorem}$$

So using the transformation method discussed, you can realize any SOP realization of AND-OR with only NAND. This is known as NAND-NAND circuit transformation. The following steps are involved:

(i) Add double bubbles in the path between the AND gate outputs and the OR gate inputs.

(ii) Change OR with inverted inputs to NAND and bubbles at inputs to their complements.



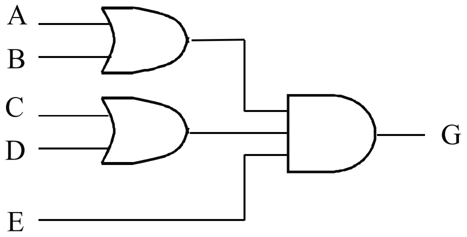
2.10.2 Implementation of POS Expressions

NOTES

Product-of-Sums expressions can be implemented using:

- 2-level OR-AND logic circuits
- 2-level NOR logic circuits

(1) OR-AND logic circuit: G = (A+B). (C+D).E

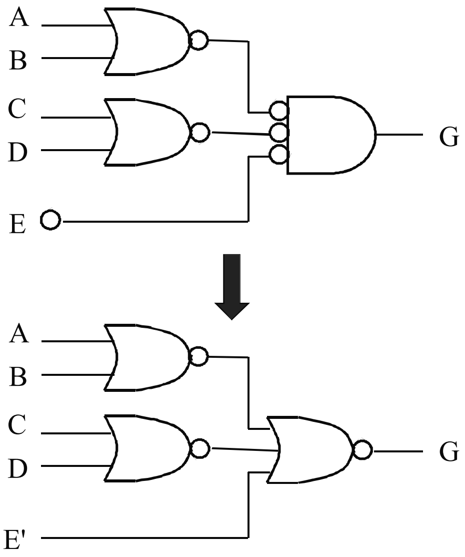


(2) NOR-NOR-based realization:

Using the transformation method discussed, you can realize any POS realization of OR-AND with only NOR. This is known as NOR-NOR circuit transformation. The following steps are involved:

(i) Add double bubbles in the path between OR gate outputs and AND gate inputs.

(ii) Changed AND-with-inverted-inputs to NOR and bubbles at inputs to their complements.



2.11 XOR AND ITS APPLICATION

There are many situations in logic design in which simplification of logic expression is possible in terms of XOR and XNOR operations. These logic gates are widely used in digital design and therefore are available in IC form. This section discusses the recognition of K-map pattern indicating XOR and XNOR functions. This is proposed by Donald K Fronek and is known as Ring map. In AND-OR or OR-AND simplification, the adjacent ones or zeros are grouped. The adjacency used in earlier K-maps is horizontal or vertical. In case of XOR/XNOR simplification, you have to look for the following:

- Diagonal Adjacencies
- Offset Adjacencies

Example of diagonal and offset adjacencies for single ones is as follows:

Two-Variable K-Map

B \ A	0	1
0	1	
1		1

K-map for F_1

$F_1 = A'.B' + A.B = (A\oplus B)' = A\cup B$

B \ A	0	1
0		1
1	1	

K-map for F_2

$F_2 = A'.B + A.B' = A\oplus B$

Three-Variable K-Map

C \ AB	00	01	11	10
0		1		1
1				

K-map for F_3

C \ AB	00	01	11	10
0				
1	1		1	

K-map for F_4

All the possible diagonal and offset adjacencies are marked in the figures and the terms corresponding to each group of such adjacency involving XOR or XNOR are also given.

These entries are known as offset mapping and the resultant expression is as follows:

$F_3 = (A'.B + A.B').C' = (A\oplus B).C'$ $F_4 = (A'.B' + A.B).C = (A\oplus B)'.C = (A\cup B).\ C$

C \ AB	00	01	11	10
0		1		
1	1			

K-map for F_5

C \ AB	00	01	11	10
0				1
1			1	

K-map for F_6

Entries known as diagonal mapping are as follows:

The resultant expression is given as:

$F_5 = A'.B.C' + A'.B'.\ C = A'.\ (B\oplus C)$ $F_6 = A.B'.C' + A.B.C = A.\ (B\oplus C)'$

From this, you can observe that if a standard K-map grouping of two ones occurs in a diagonal or offset adjacent pattern, then this can be recognized as

XOR or XNOR function and the function can be simplified in terms of XOR or XNOR functions.

The direct method for obtaining expressions when there are offset or diagonal adjacencies in K-map are very interesting. For diagonal element grouping, identify the following terms, i.e., input variables, which are not changing and then identify the pattern from the rest of the variables, for {01 to 10 or 10 to 01} or {00 to 11 or 11 to 00} types.

Case 1. When variables are changing in the pattern, 01 to 10 or 10 to 01, then the XOR gate is to be taken for the realization.

Case 2. When variables are changing in the pattern, 11 to 00 or 00 to 11, then XNOR gate is to be taken for the realization.

For example, take the following K-map:

C \ AB	00	01	11	10
0		1		1
1	1		1	

In this case, there are two diagonal adjacencies. They are at {010 and 001} and {100 and 111} locations.

As discussed above, 010 and 001 can be combined and can be written in terms of XOR expression. It can be seen that for the above diagonal element, A is not changing and it is in the complemented form, and BC values are changing from 10 to 01. So you can directly write the combined expression as $A'.\,(B\oplus C)$.

Similarly, for the other diagonal grouping, i.e., for 100 and 111, you can write $A(B\cup C)$.

So, the overall expression can be written as follows:

$$F = A'.(B\oplus C) + A(B\cup C)$$

Example 2.8: Using the technique discussed above, find the realization of circuit resulting from the following K-map grouping:

C \ AB	00	01	11	10
0	1		1	
1		1		1

Solution: Diagonal elements for grouping are $\{m_0, m_3\}$ and $\{m_6, m_5\}$. For the grouping of m_0 and m_3, i.e., for 000 and 011, A is constant and pattern is of 00 to 11 type, so the expression can be written for this combination as $A'.(B\cup C)$.

Similarly, for the other grouping, which is for 110 and 101 grouping, the term, which is not changing is A and the pattern of 10 to 01 type, so you can write $A.\,(B\oplus C)$.

So you can write, $$F = A'.\,(B\cup C) + A.\,(B\oplus C)$$
$$= A\cup(B\oplus C)$$

2.12 APPLICATIONS OF BOOLEAN ALGEBRA TO SWITCHING THEORY

Boolean algebra is used to simplify the practical use of logic circuits. The function of logic circuit is translated into symbolic form. Some rules of algebra are used for the resulting equation which is able to lessen the number of arithmetic operations and the simplified equation is translated again into the form of logic circuit. This equivalent function is achieved with the help of components. Various rules are presented to reduce the Boolean expressions into the simplest way. The identities and properties are used to review the many identities. For example, 'A' can be proved symbolically in two terms, such as A+1 = 1 and 1A=A to achieve the final result and the logical circuit is designed in the following way (refer Figure 2.21).

$$\mathbf{A + AB = A}$$

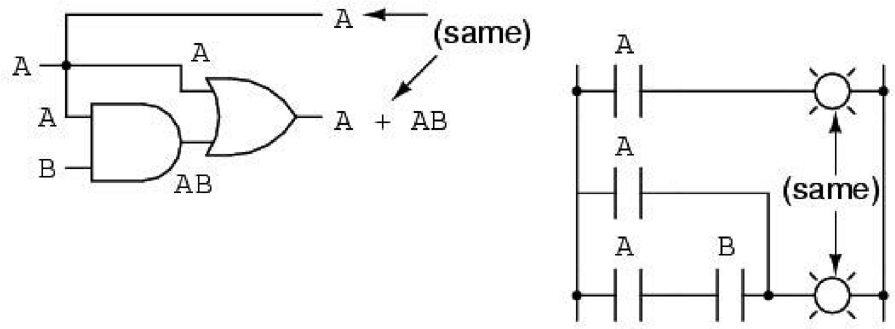


Fig. 2.21 Logical Circuit for Expression A + AB =A

Let take an expression as A+AB and factoring A out of both terms. Applying identity A+1 = 1 and in the next step, applying identity 1A=A that returns value A. It can be proved in the following way:

$$
\begin{array}{ll}
A + AB & \\
\downarrow & \text{Factoring } \mathbf{A} \text{ out of both terms} \\
A(1 + B) & \\
\downarrow & \text{Applying identity } \mathbf{A + 1 = 1} \\
A(1) & \\
\downarrow & \text{Applying identity } \mathbf{1A = A} \\
A &
\end{array}
$$

The rule A+1 = 1 is used to reduce (B+1) term to 1. If rule A+1=1 is expressed by using alphabet A then it is not necessary that it only applies to expression containing A. The Boolean expression ABC+1 reduces to 1 with the help of A+1=1 identity. The term A in identity's standard form is used to represent ABC in the expression. The following Figure 2.22 shows the arrangement of logical circuit for the expression $\mathbf{A + \bar{A}B = A + B}$.

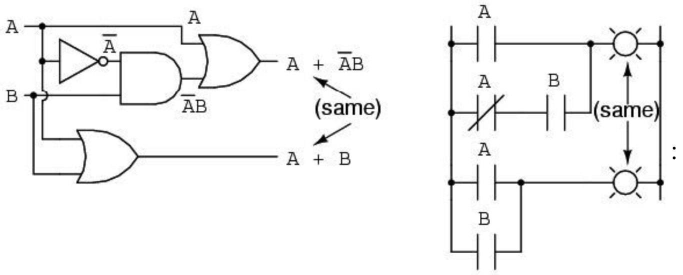


Fig. 2.22 *Logical Circuit for Expression*

The expression can be explained in the following way:

$A + \overline{A}B$

↓ Applying the previous rule to expand **A** term
A + AB = A

$A + AB + \overline{A}B$

↓ Factoring **B** out of 2nd and 3rd terms

$A + B(A + \overline{A})$

↓ Applying identity $\mathbf{A + \overline{A} = 1}$

$A + B(1)$

↓ Applying identity **1A = A**

$A + B$

The expression (A+AB=A) is used with the rule to simplify 'A' term and then change 'A' into the expression 'A+AB'. Other rule is involved to simplify the product-of-sum expression and the logical circuit is designed for the expression (A+B) (A+C) = A + BC (Refer Figure 2.23).

(A + B) (A + C) = A + BC

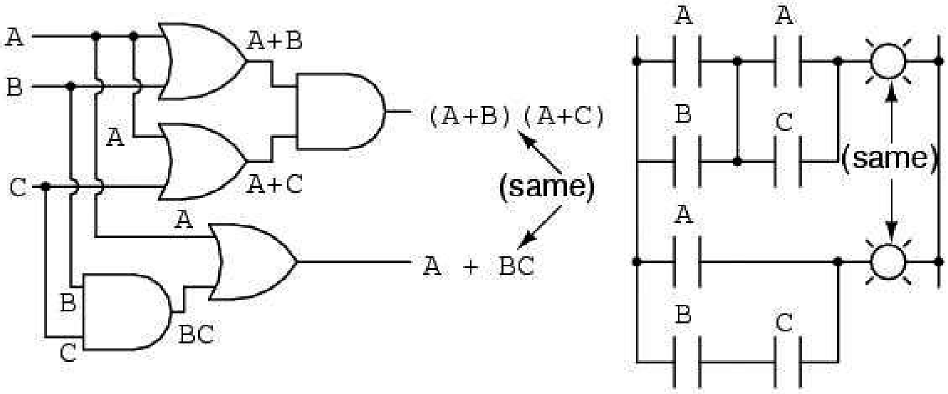


Fig. 2.23 *Logical Circuit for Expression*

This expression can be simplified in the following way:

$$\begin{tikzcd}
(A + B)(A + C) \arrow[d, "\text{Distributing terms}"'] \\
AA + AC + AB + BC \arrow[d, "\text{Applying identity } \mathbf{AA = A}"'] \\
A + AC + AB + BC \arrow[d, "\text{Applying rule } \mathbf{A + AB = A} \text{ to the } A + AC \text{ term}"'] \\
A + AB + BC \arrow[d, "\text{Applying rule } \mathbf{A + AB = A} \text{ to the } A + AB \text{ term}"'] \\
A + BC
\end{tikzcd}$$

Basically, the three useful Boolean rules are implied to simplify the Boolean expression in the following way:

$$A + AB = A$$

$$A + \overline{A}B = A + B$$

$$(A + B)(A + C) = A + BC$$

Check Your Progress

20. What is a Don't Care condition? Explain with the help of an example.
21. What is a valid range for BCD inputs?
22. How are Sum-Of-Product expressions implemented?
23. How are Product-Of-Sum expressions implemented?
24. How is AND-OR and OR-AND simplification done?
25. State the three useful Boolean rules implied to simplify the Boolean expression.

2.13 ANSWERS TO 'CHECK YOUR PROGRESS'

1. The basic NOT gate has only one input and one output. The output is always the opposite or negation of the input. The following is the truth table for NOT gate:

A	F
0	1
1	0

2. For example, NOT A is represented as A′.
3. A basic AND gate consists of two inputs and an output. In the AND gate, the output is 'High' or gate is 'On' only if both the inputs are 'High'. The relationship between the input signals and the output signals is often

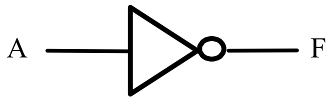
represented in the form of a truth table. It is nothing but a tabulation of all possible input combinations and the resulting outputs. For the AND gate, there are four possible combinations of input states: {A = 0, B = 0}; {A = 0, B = 1}; {A = 1, B = 0}; and {A = 1, B = 1}. In the truth table, these are listed as follows:

A	B	F
0	0	0
0	1	0
1	0	0
1	1	1

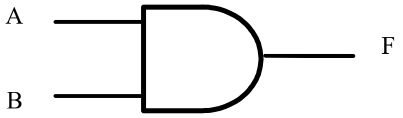
4. A basic OR gate is a two input, single output gate. Unlike the AND gate, the output is 1 when any one of the input signals is 1. The OR gate output is 0 only when both the inputs are 0. The truth table for the OR gate is as follows:

A	B	F
0	0	0
0	1	1
1	0	1
1	1	1

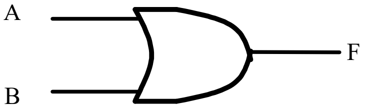
5. **('.'** implies AND operation) ('+' implies OR operation)
6. The following figure shows NOT gate representation:



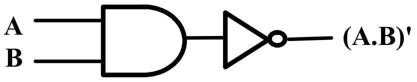
The following figure represents AND gate:



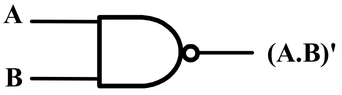
The following figure represents OR gate:



7. NAND Gate: AND followed by NOT

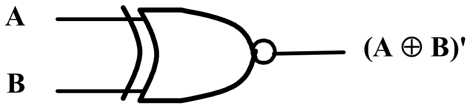


NAND



8. XNOR Gate: XOR followed by NOT

   NOR



9. Duality Principle: Every valid Boolean expression (equality) remains valid if the operators and identity elements are interchanged.

   $+ \leftrightarrow .$

   $1 \leftrightarrow 0$

10. A Boolean expression is an algebraic statement containing Boolean variables and operators.
11. A Boolean function is an expression formed with binary variables, the two binary operators OR and AND, the unary operator NOT, and the equal and parenthesis signs. Its result is also a binary value. The general usage is '**.**' for AND, **'+'** for OR and **'′'** for NOT.

    To lessen the brackets used in writing Boolean expressions, operator precedence can be used. Precedence (highest to lowest): $' \rightarrow \mathbf{.} \rightarrow +$
12. A truth table is a table, which consists of every possible combination of inputs and its corresponding outputs.
13. For a function F the complement of this function F′ is obtained by interchanging 1 with 0 and vice versa in the function's output values.
14. The following are two standard forms for writing a Boolean expression:
    - Sum-Of-Product (SOP)
    - Product-Of-Sum (POS)
15. A Minterm of *n* variables is the product of *n* literals from the different variables. In general, *n* variables can give 2^n Minterms.
16. There are basically two types of simplification techniques:
    - Algebraic Simplification
    - Karnaugh Maps (K-Map)
17. It is a diagram-based simplification technique. It gives simplified Boolean expressions in standard forms. It is a systematic method to obtain simplified Sum-Of-Products (SOP) Boolean expressions with the objective of fewest possible terms/literals. It is a diagrammatic technique based on a special form of Venn diagram. A Karnaugh map (K-map) is an abstract form of Venn diagram, organized as a matrix of squares, where each square represents a Minterm.
18. The main advantage of reduction is that it needs less hardware in terms of logic gate. Less number of literals gives realization based on logic gate with less input pin.
19. In a lattice, an element *x* is join-irreducible if *x* is not the join of a finite set of other elements. Equivalently, *x* is join-irreducible if it is neither the bottom element of the lattice (the join of zero elements) nor the join of any two

NOTES

smaller elements. For example, in the lattice of divisors of 120, there is no pair of elements whose join is 4, so 4 is join-irreducible. An element x is join-prime if it differs from the bottom element, and whenever $x \leq y \vee z$, either $x \leq y$ or $x \leq z$. In the same lattice, 4 is join-prime: whenever LCM(y, z) is divisible by 4, at least one of y and z must itself be divisible by 4.

20. A function with n variable in an input can have 2^n possible combinations. If you have four variables in an input, then you have 16 possible combinations for the input. In designing some digital systems, it may happen that for some input combination(s), a designer may not be interested in output(s) corresponding to those input combinations. In such cases, the outputs can be either '1' or '0'. These are called don't care conditions, denoted by X (or sometimes, d).
21. Valid BCD inputs range from 0000 to 1001.
22. Sum-of-Products expressions can be implemented using either (i) 2-level AND-OR logic circuits or (ii) 2-level NAND logic circuits.
23. Product-of-Sums expressions can be implemented using:
    - 2-level OR-AND logic circuits
    - 2-level NOR logic circuits
24. In AND-OR or OR-AND simplification, the adjacent ones or zeros are grouped.
25. The three useful Boolean rules implied to simplify the Boolean expression are:
    1. $A + AB = A$
    2. $A + \overline{A}B = A + B$
    3. $(A + B)(A + C) = A + BC$

2.14 SUMMARY

- The output of the NOT gate is always the opposite or negation of the input.
- In the AND gate, the output is 'High' or gate is 'On' only if both the inputs are 'High'.
- In the OR gate, the output is 1 when any one of the input signal is 1 and 0 otherwise.
- The output of the XOR gate is 1 when one and only one of the inputs is 1.
- In addition to the NOT, AND, OR and XOR gates, three more common gates are available. These gates are called NAND ('Not AND'), NOR ('Not OR') and XNOR ('Exclusive Not OR').
- When a Boolean expression is provided, you can easily draw the logic circuit.
- When a logic circuit diagram is given, you can analyse the circuit to obtain the logic expression.

- Boolean algebra consists of a set of elements B, with two binary operations {+} and {.} and a unary operation {′}, such that the following axioms hold:
  - o The set B contains at least two distinct elements x and y.
  - o Closure
  - o Commutative laws
  - o Associative laws
  - o Identities
  - o Distributive laws
  - o Complement
- Every valid Boolean expression remains valid if the operators and identity elements are interchanged.
- A Boolean expression is an algebraic statement containing Boolean variables and operators.
- A Boolean function is an expression formed with binary variables, the two binary operators OR and AND, the unary operator NOT, and the equal and parenthesis signs.
- To lessen the brackets used in writing Boolean expressions, operator precedence can be used.
- A truth table is a table, which consists of every possible combination of inputs and its corresponding outputs.
- For a function F, the complement of this function F′ is obtained by interchanging 1 with 0 and vice versa in the function's output values.
- The two standard forms for writing a Boolean expression are Sum-Of-Product and Product-Of Sum.
- A Minterm of n variables is the product of n literals from the different variables.
- A Maxterm of n variables is the sum of *n* literals from the different variables.
- Canonical way is a unique way of representing Boolean expressions.
- Boolean algebra is a lattice which contains a least element and a greatest element and which is both complemented and distributive.
- A non-zero element 'a' in a Boolean algebra (B, +, ., ′) is called an atom if for every $x \in B$, $x \wedge a = a$ or $x \vee a = 0$.
- The two basic techniques for simplification of Boolean expression are algebraic simplification and Karnaugh maps.
- Algebraic simplification involves simplifications using Boolean theorems. Karnaugh map is a diagram-based simplification technique.
- In designing some digital systems, it may happen that for some input combination(s), a designer may not be interested in output(s) corresponding to those input combinations. In these cases, the outputs can be either '1' or '0'.
- It is possible to implement any Boolean expression using NAND and NOR gates.

- There are many situations in logic design in which simplification of logic expression is possible in terms of XOR and XNOR operations.
- Boolean algebra is used to simplify the practical use of logic circuits.

2.15 KEY TERMS

- **Basic OR gate:** A basic OR gate is a two-input, single-output gate. Unlike the AND gate, the output is 1 when any one of the input signals is 1. The OR gate output is 0 only when both inputs are 0.
- **Boolean expression:** It is an algebraic statement containing Boolean variables and operators.
- **Minterm:** Minterm of *n* variables is the product of *n* literals from the different variables.
- **Maxterm:** It is the sum of terms of the corresponding Minterm with its literal complemented.
- **Canonical form:** It is a unique way of representing Boolean expressions. Any Boolean expression can be written in the sum-of-Minterm form.
- **Karnaugh map (K-map):** It is a diagram-based simplification technique which is easy for a circuit designer and involves pattern-matching skills.

2.16 SELF-ASSESSMENT QUESTIONS AND EXERCISES

Short-Answer Questions

1. What are the basic logic gates?
2. What will be the OR gate output if both the inputs are 0?
3. What will be the XOR output if the inputs are different?
4. Explain XOR gate with the help of a truth table and draw the symbolic representation.
5. What are universal logic gates?
6. What are the basic laws of Boolean algebra?
7. Explain Boolean expression.
8. Explain Minterm and Maxterm.
9. What is a Karnaugh map?

Long-Answer Questions

1. Define a basic OR gate. Compare it with the AND gate.
2. Explain NOT and XOR gates. What are their applications?
3. ‘Similar to the two input gates, we can have more than two inputs to all the logic gates excluding the NOT gate’. Is the statement true? Explain.

4. What are universal logic gates? Why are they so called? Describe each type with the help of symbols.
5. Draw a logic circuit for $F = xy' + x'z$.
6. Explain the laws of Boolean algebra.
7. What is a Boolean function? Explain with the help of a truth table.
8. Discuss canonical form.
9. Explain the basic types of function simplification techniques.
10. 'There are many situations in logic design in which simplification of logic expression is possible in terms of XOR and XNOR operations'. Elaborate further on the statement.
11. Discuss the Karnaugh Map (K-map) technique.
12. What is a Don't Care condition?
13. Explain XOR and its applications.
14. Convert the following into other canonical forms:

    (*a*) $F(x, y, z) = \Sigma(1, 3, 7)$

    (*b*) $F(A, B, C, D) = \Sigma(0, 2, 6, 11, 13, 14)$

    (*c*) $F(x, y, z) = \Pi(0, 3, 6, 7)$

    (*d*) $F(A, B, C, D) = \Pi(0, 1, 2, 3, 4, 6,12)$
15. Express the following function in a Sum of Minterms and a Product of Maxterms:

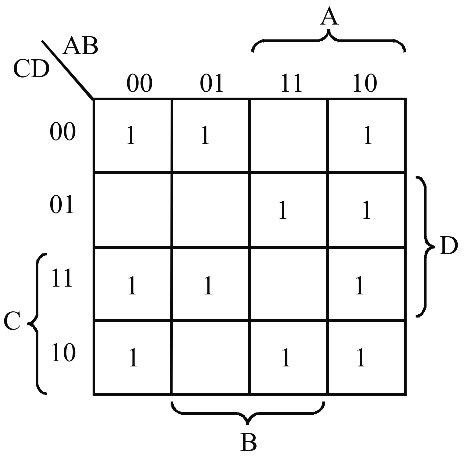
    (*a*) $F(A, B, C, D) = D(A' + B) + B'D'$

    (*b*) $F(w, x, y, z) = y'z + wxy' + wxz' + w'x'z$

    (*c*) $F(A, B, C, D) = (A + B' + C)(A + B')(A + C' + D')$

    (*d*) $F(A. B, C) = (A' + B)(B' + C)$

    (*e*) $F(x, y, z) = (xy + yz)(y + xz)$
16. Identify the prime implicants and the essential prime implicants of the following K-map:



CD \ AB	00	01	11	10
00	1	1		1
01			1	1
11	1	1		1
10	1		1	1

17. $F(A,B,C,D) = \sum m(2,8,10,15) + \sum d(0,1,3,7)$, find the reduced expression using K-map.

NOTES

18. Simplify each of the following functions and implement them with NAND gates. Give two alternatives.

    (a) $F_1 = ac' + ace + ace' + a'cd' + a'd'e'$

    (b) $F_2 = (b' + d')(a' + c' + d)(a + b' + c' + d)(a' + b + c' + d')$

19. Simplify the following functions and implement them with NOR gates. Give two alternatives.

    $F_1 = ac' + ace + ace' + a'cd' + a'd'e'$

2.17 FURTHER READING

Iyengar, N Ch S N. V M Chandrasekaran, K A Venkatesh and P S Arunachalam. *Discrete Mathematics*. New Delhi: Vikas Publishing House Pvt. Ltd., 2007.

Tremblay, Jean Paul and R. Manohar. *Discrete Mathematical Structures with Applications to Computer Science*. New York: McGraw-Hill Inc., 1975.

Deo, Narsingh. *Graph Theory with Applications to Engineering and Computer Science*. New Delhi: Prentice-Hall of India, 1999.

Singh, Y.N. *Mathematical Foundation of Computer Science*. New Delhi: New Age International Pvt. Ltd., 2005.

Malik, D.S. *Discrete Mathematical Structures: Theory and Applications*. London: Thomson Learning, 2004.

Haggard, Gary, John Schlipf and Sue Whiteside. *Discrete Mathematics for Computer Science*. California: Thomson Learning, 2006.

Cohen, Daniel I.A. *Introduction to Computer Theory*, 2nd edition. New Jersey: John Wiley and Sons, 1996.

Hopcroft, J.E., Rajeev Motwani and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*, 3rd edition. Boston: Addison-Wesley, 2006.

Linz, Peter. *An Introduction to Formal Languages and Automata*, 5th edition. Boston: Jones and Bartlett Publishers, 2011.

Mano, M. Morris. *Digital Logic and Computer Design*. New Jersey: Prentice-Hall, 1979.

UNIT 3 GRAPH THEORY

Structure

3.0 INTRODUCTION

After reading this unit you will learn about the fundamentals of graph theory. A graph contains a set of vertices, a set of edges and a function that connects two vertices to form an edge. Every graph has its associated graph which is useful for understanding problems. Nodes are represented by small circles and edges by lines. We denote a graph as a set of vertices V and edges E. Mathematically, it is written as $G = (V, E)$. There are simple graphs with no self loops and parallel edges. There are other types of graphs too, like directed and undirected graphs, pseudographs, bipartite graph, regular graph, etc.

You will also learn about matrix representation of graphs. Such matrix is known as incidence matrix and adjacency matrix. This type of matrix representation is used to verify isomorphism between two graphs.

After reading this unit you will learn the basic properties of trees and graphs. These are important data structures in computer science. You will be able to make a distinction between a tree and a graph. A graph contains nodes and paths joining these nodes. A graph is cyclic when one starts from a node and after traversing a few nodes, it returns to that node. A graph in which the starting node is not reached after traversal is known as an acyclic graph. A connected acyclic graph is known as tree.

You will also learn about various types of trees like, directed tree, rooted tree, binary tree, k-ary tree, etc., and enhance your concept. A directed tree is one in which every edge has a direction. A vertex that is a starting node is the vertex of zero degree and is called a root. A tree with its root is called a rooted tree. A vertex may be connected to another and those above, are called parent node and those below, that are called children. A root node has no parent and a node that has no child is known as leaf. In a tree, if a node has maximum of two nodes, then it is called binary tree.

You will learn about the application of trees in a binary search, decision tree, spanning tree and searching strategies like, depth first search, breadth first search, etc. A binary search tree is a binary tree and each child node is either left child or right child. No node has more than two children, one left and another right. A decision tree is a rooted tree in which each internal node is assigned with a decision for a sub-tree at the vertices and such outcome forms a decision tree.

You will familiarize yourself with traversal of a tree, pre-order, in-order and post-order traversals.

You will also understand searching strategies and two most popular algorithms, Prism's algorithm and Kruskal algorithm. Finally, you will learn about planar graphs Dijkstra's algorithm and Warshall's algorithm.

3.1 OBJECTIVES

After going through this unit, you will be able to:

- Understand the basic terminology used in Graph theory
- Define different types of graphs
- Understand the importance of graphs in visualizing problems
- Understand the path and circuits of graphs
- Differentiate between connected and disconnected graphs
- Explain Euler's graph
- Represent a graph as a matrix
- Find the minimum spanning tree of a weighted graph
- Define trees and graphs
- Understand the applications of trees and graphs
- Write algorithms for pre-order, in-order and post-order traversals
- Develop an understanding of Prim's and Kruskal's algorithms for finding spanning tree of a graph
- Know about planar graphs, Dijkstra's algorithm and Warshall's algorithm.

3.2 BASIC TERMINOLOGY

Graph

A graph *G*, a triplet $(V(G), E(G), \theta_G)$ consisting of a non-empty set *V*(*G*) of vertices, a set $E(G)$ of edges, and a function θ_G assigns to each edge, a subset {*u*, *v*} of *V*(*G*) (*u*, *v* need not be distinct). If *e* is an edge and *u*, *v* are vertices such that $\theta_G(e) = uv$, then *e* is a line (edge) between *u* and *v*; the vertices *u* and *v* are the end points of the edge *e*.

NOTES

For example, (*i*) $G = (V(G),\ E(G),\ \theta_G)$

Where, $V(G) = \{v_1,\ v_2,\ v_3,\ v_4\}$

$$E(G) = \{e_1,\ e_2,\ e_3,\ e_4,\ e_5,\ e_6\}$$

$$\theta_G(e_1) = \{v_1v_2\},\ \ \theta_G(e_2) = \{v_2v_2\},\ \ \theta_G(e_3) = \{v_2v_3\}$$

$$\theta_G(e_4) = \{v_1v_3\};\ \ \theta_G(e_5) = \{v_4v_5\};\ \ \theta_G(e_6) = \{v_1v_4\}$$

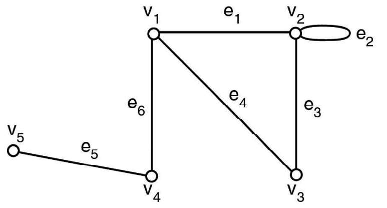
(*ii*) $G = (\ V(G),\ E(G),\ \theta_G)$

Where, $V(G) = \{v_1,\ v_2,\ v_3\};\ \ E(G) = \{e_1,\ e_2,\ e_3\}$

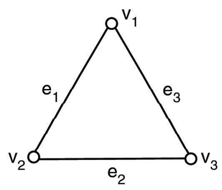
$$\theta_G\ \ (e_1) = \{v_1v_2\};\ \ \theta_G(e_2) = \{v_2v_3\};\ \ \theta_G(e_3) = \{v_3v_1\}$$

Every graph has a diagram associated with it. These diagrams are useful for understanding problems involving such a graph. In the pictorial representation, we represent the vertices by small circles and the edges by lines whenever the corresponding pair of vertices forms an edge.

The following are the pictorial representation of examples (*i*) and (*ii*):



(i)



(ii)

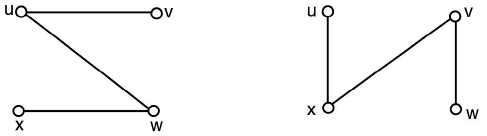
Notes:

1. In example (*i*), e_2 joins the vertex v_2 to itself. Such an edge is called self loop (loop).
2. Suppose there is more than one edge between a pair of vertices in a graph, these edges are called parallel edges.
3. Hereafter, we denote the graph $G = (V, E)$ for simplicity.
4. A graph, which consists of parallel edges, is called a multigraph.

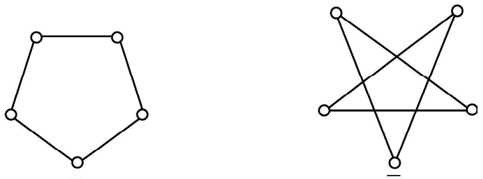
Simple Graph: A graph with no self loops and parallel edges is called a simple graph.

Complement of a Graph: The complement $\overline{G}$ of a graph *G* is a graph with $V(G) = V(\overline{G})$ and such that *uv* is an edge of $\overline{G}$ if and only if *uv* is not an edge of *G*.

For example,



G (i) $\overline{G}$



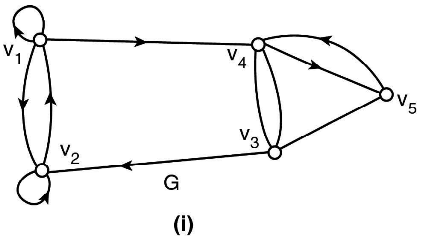
G $\overline{G}$

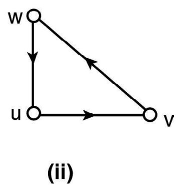
(ii)

There are also some useful terminology for graphs with directed edges.

Graphs with directed edges: When (u, v) is an edge of the graph G with directed edges, u is said to be adjacent to v and v is said to be adjacent from u. The vertex u is called the initial vertex of (u, v) and v is called the terminal or end vertex of the edge (u, v).

For example,



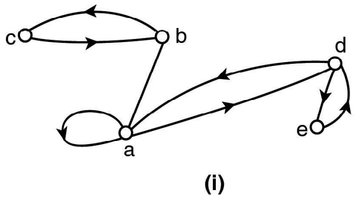


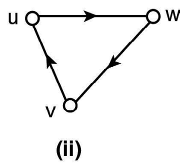
In-Degree and Out-Degree: In a graph with directed edges, the in-degree of a vertex v denoted by $d^{-}(v)$ is the number of edges with v as their terminal vertex. The out-degree of v denoted by $d^{+}(v)$ is the number of edges with v as their initial vertex.

Note: Self loop at a vertex contributes 1 to both in-degree and out-degree of this vertex.

Example 3.1: Find the in-degree and out-degree of the following graphs.

Solution:





(i) $d^{-}(a)=3;\ d^{-}(b)=1;\ d^{-}(c)=1;\ d^{-}(d)=2;$ and $d^{-}(e)=1$

$d^{+}(a)=2;\ d^{+}(b)=2;\ d^{+}(c)=1;\ d^{+}(d)=2;\ d^{+}(e)=1$

(ii) $d^{-}(u)=1;\ d^{-}(v)=1;\ d^{-}(w)=1$

$d^{+}(u)=1;\ d^{+}(v)=1;\ d^{+}(w)=1$

Notes:

1 Let $G=(V,E)$ be a graph with directed edges. Then, $\sum\limits_{v \in V} d^{-}(v) = \sum\limits_{v \in V} d^{+}(v) = e.$

2 By ignoring directions of edges in a graph with directed edges, we will get an undirected graph. Such graphs are called underlying undirected graphs.

3.3 DIFFERENT TYPES OF GRAPH

Null Graph

A null graph is a totally disconnected graph. A null graph does not have any edge. Every vertex in a null graph is an isolated vertex. Figure 3.1 shows a null graph with six vertices.

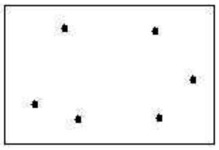


Fig. 3.1 *Null Graph*

Directed and Undirected Graph

In a directed graph every edge has a direction (refer Figure 3.2). If there is movement from a vertex to its adjacent vertex the direction is notified. If movement is from vertex v_1 to vertex v_2, then v_1v_2 and v_2v_1 are different. Here movement is in one direction only. But in undirected graph, if there is movement in between v_1 and v_2, then movement in both the direction is possible. Such graphs are known as undirected graphs (Refer Figure 3.3).

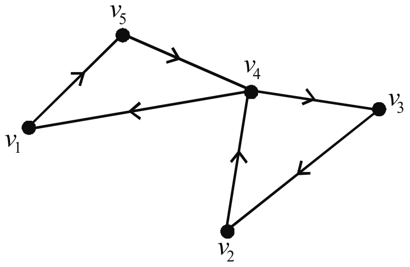


Fig. 3.2 *Directed Graph*

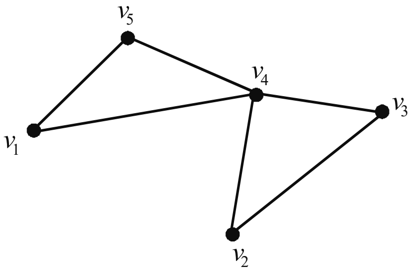


Fig. 3.3 *Undirected Graph*

Simple Graph

A graph with no self loop and parallel edges is known as simple graph. Figure 3.4 shows a simple graph.

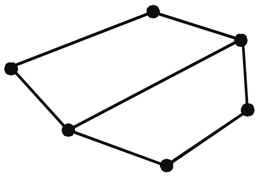


Fig. 3.4 *Simple Graph*

Multigraph

A graph which has loops and parallel edges is a multigraph. Figure 3.5 shows a multigraph.

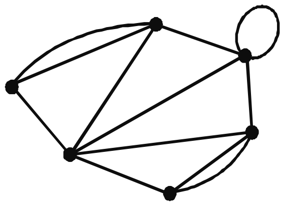


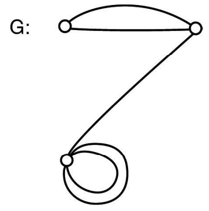
Fig. 3.5 *Multigraph*

Pseudograph

A graph with self loops and parallel edges is called a pseudograph.

Note: Every simple graph and every multigraph is a pseudograph, but the converse is not true.

For example,

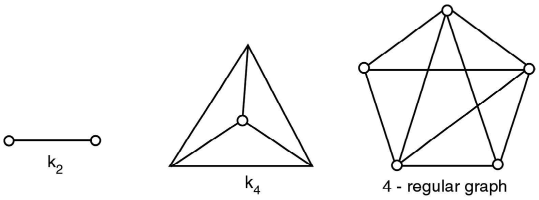


The above graph G is neither a simple graph nor a multigraph.

Complete Graph

A simple graph in which each pair of distinct vertices is joined by an edge is called a complete graph. A complete graph on n vertices is denoted by k_n.

For example,



Complete graphs on 2 and 4 vertices respectively.

Notes:

1. Every complete graph k_n is a $(n–1)$ regular graph.
2. There is no 1-regular or 3-regular graphs with 5 vertices. (since no graph has an odd number of vertices).

Subgraphs

Let there be a graph given by $G(V, E)$. If another graph (denoted as $H(V', E')$) is obtained by deleting few vertices and edges then it is the subgraph of G, if V' in graph H contains all the terminal points of edges in E'. If we remove an edge, its terminal points remain in place, but if a vertex is removed, then edges that are meeting on this vertex are also removed.

Examples of graph and its subgraph are shown below:

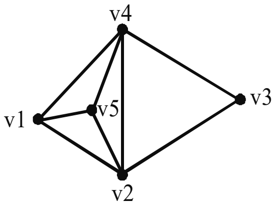


Fig. 3.6 *Graph G*

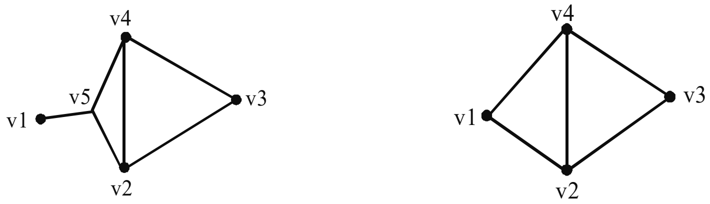


Fig. 3.7 (i) *Subgraph H of G with Edges v_1v_4 and v_1v_2 Removed*

(ii) *Subgraph H of G with Vertex v_5 Removed*

Regular graph

In a regular graph every vertex is of the same degree. If every vertex in a graph is of degree 2, then, it is called a regular graph of degree 2. Thus, a null graph may be called a regular graph of degree zero.

Bipartite Graph

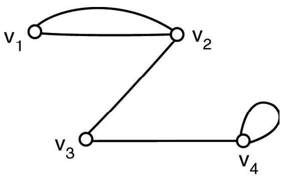
A simple graph G is called bipartite if its vertex set V can be partitioned into two disjoint non-empty sets V_1 and V_2, such that, every edge in the graph connects a vertex in V_1 and a vertex in V_2. Note that no edge in G connects either two vertices in V_1 or two vertices in V_2.

For example, G is bipartite, because its vertex set $V = \{v_1, v_2, v_3, v_4, v_5, v_6\}$ is partitioned into two non-empty sets $V_1 = \{v_1, v_3, v_5\}$ and $V_2 = \{v_2, v_4, v_6\}$. Also every edge in G connects a vertex in V_1 and a vertex in V_2.

3.4 INCIDENCE AND DEGREE

Degree of a Vertex: The degree of a vertex v is the number of edges incident with that vertex. In other words, the degree of a vertex is the number of edges, having that vertex as an end point, and is denoted by $d(v)$.

For example,



Here, $d(v_1) = 2$
$d(v_2) = 3$
$d(v_3) = 2$
$d(v_4) = 3$

A loop contributes 2 to the degree of vertex.

Isolated Vertex: A vertex with degree zero is called an isolated vertex.

Pendant Vertex: A vertex with degree one is called a pendant vertex.

Adjacent Vertices: A pair of vertices that determine an edge are called adjacent vertices.

Note: A vertex is even or odd if as its degree is even or odd.

Example 3.2: Let G be a simple graph with n vertices. Prove that the number of edges $E(G)$ is atmost ${}^nC_2$.

Solution: Let $G = (V(G), E(G), \theta_G)$ be a simple graph with $|V(G)| = n.$

Since θ_G assigns to each edge, a 2 element subset $\{u, v\}$ of $V(G)$, there are atmost ${}^nC_2$ number of 2 element subsets.

Hence, $E(G) \leq \dfrac{n(n-1)}{2}$

Theorem 3.1: Let G be a graph with n vertices and e edges. Then $\sum_{i=1}^{n} d(v_i) = 2e$

Proof: Let G be a graph with n vertices and e edges.

Since every edge contributes degree 2 to this sum, $\sum_{i=1}^{n} d(v_i) = 2e$

NOTES

Theorem 3.2: In a graph G, the number of odd vertices is an even number.

Proof: Let G be a graph with n vertices and e edges.

By Theorem 3.1, we have

$$\sum_{i=1}^{n} d(v_i) = 2e = \text{Even number} \tag{3.1}$$

Among n vertices, some are even vertices and some are odd vertices. Let V_e and V_o be the even and odd vertices respectively.

Now Equation (3.1) can be written as,

$$\sum_{v\in V_e}^{n} d(v) + \sum_{v\in V_0} d(v) = \text{Even number}$$

$$\therefore \quad \sum_{v\in V_0} d(v) = \text{Even number } - \sum_{v\in V_e}^{n} d(v) \quad (3.2)$$

Since every term in the right side of Equation (3.2) is even, the sum on the left side must contain an even number of terms, i.e., the number of odd vertices in G is even.

Minimum and Maximum Degrees: Let G be a graph. The minimum and maximum degrees of G are respectively $\delta(G)$ and $\Delta(G)$ and given as

$$\delta(G) = \min\ \{d(v); v \in V(G)\}$$
$$\Delta(G) = \max\ \{d(v); v \in V(G)\}$$

k-Regular: A graph G is k-regular or regular of degree k, if every vertex of G has degree k.

3.5 PATH AND CIRCUITS OF A GRAPH

Walk

A walk is a sequence of vertices and edges starting from any vertex and travelling through edges to a destination vertex, such that no edge appears more than once. But in a walk a vertex may be visited more than once. Examples of walk are being given below:

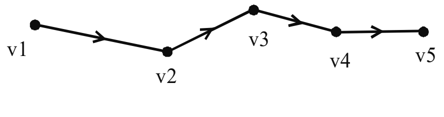


Fig. 3.8 *This Shows a Walk with Single Visit on Every Vertex.*

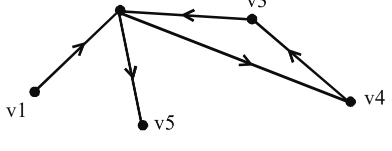


Fig. 3.9 *This Shows a Walk with Two Visits on Vertex v_2*

Spanning Tree

A connected graph might contain more than one spanning tree. Consider the following graphs.

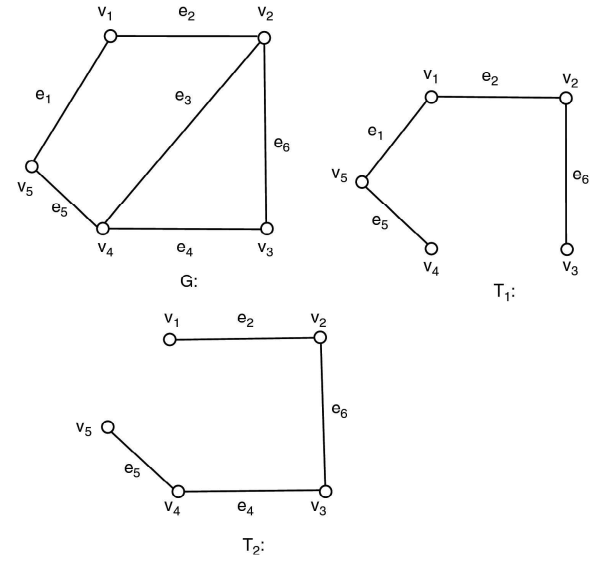


Fig. 3.10 *Spanning Trees*

In T_1, the edges e_1, e_2, e_5, e_6 are present, whereas in T_2, edges e_2, e_4, e_5, e_6 are present.

Edges of *G*, which are present in a spanning tree *T*, are called as the *branches* of *G* with respect to *T*. The edges of *G*, which are not present in its spanning tree *T*, are called the *chords* of *G* with respect to *T*.

In the above example, the branches of *G* are e_1, e_2, e_5, e_6, with respect to T_1 and the branches of *G* are e_2, e_4, e_5, e_6, with respect to T_2.

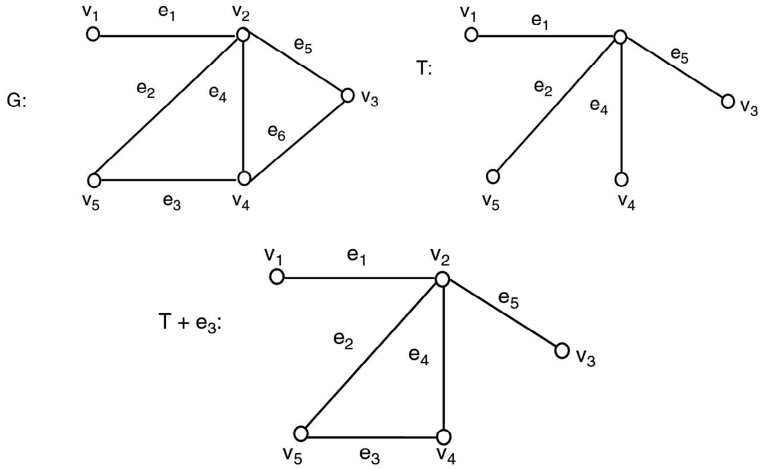
Note: Let *G* be a connected graph on *n* vertices; *e*-edges and *T* be one of its spanning tree. Since *T* is a tree on *n* vertices, it has $(n - 1)$ edges, i.e., the number of branches of *G* with respect to *T* is $(n - 1)$; the number of chords of *G* with respect to *T* is $e - (n - 1)$. Often the number of branches of *G* is called as rank of *G* and is denoted by $r(G)$; the number of chords of *G* is called as the nullity of *G*, denoted by $\mu(G)$. In general, for a connected graph of *n*-vertices and *e*-edges, $r(G)$, the rank of *G* is $(n - 1)$; $\mu(G)$, the nullity of *G* is $e - n + 1$.

Fundamental Circuit

Let *T* be the spanning tree of a connected graph *G*, and *e* be a chord of *G* with respect to *T*. Since the spanning tree *T* is minimally acyclic, the graph *T*+*e* contains a unique cycle. This cycle is called a fundamental cycle in *G* with respect to *T*.

Every chord of *G* gives rise to a fundamental cycle. Therefore, the number of fundamental cycles possible for a connected graph is atmost $\mu(G)$.

For example,



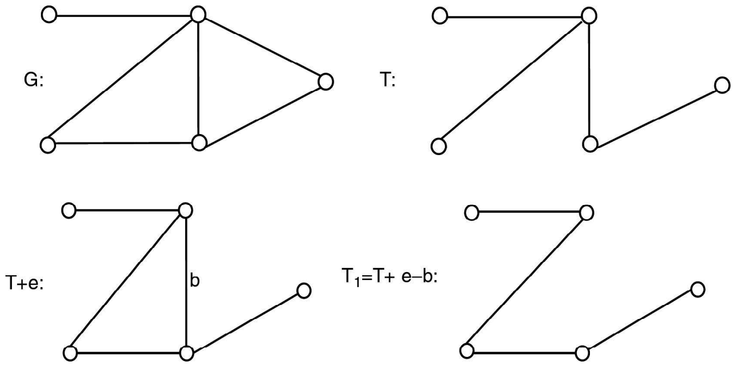
G – graph, T – Spanning tree of G, T+e_3 – Fundamental cycle

Fig. 3.11 *Fundamental Circuit*

Cyclic Interchange

Let *T* be a spanning tree of *G* and *e* be a chord of *G* with respect to *T*. The graph *T*+*e* is a fundamental circuit. In this circuit other than edge *e*, all the other edges are branches of *G* with respect to *T*. On removal of any of the branches from the fundamental circuit, we get a spanning tree T_1, i.e., *b* is a branch in the fundamental circuit with respect to a chord *e*, then spanning tree T_1 is obtained by removing *b* from $T + e$, i.e, $T_1 = T + e - b$. This process is called cyclic interchange.

For example,



G – Connected graph, T – Spanning tree

T + e – Fundamental circuit, T_1 *– Spanning tree obtained by cyclic interchange*

Check Your Progress

1. What are graphs with directed edges?
2. What is null graph?
3. State degree of vertex.
4. What is walk?

3.6 CONNECTED AND DISCONNECTED GRAPHS AND COMPONENTS

NOTES

In this section, we study the structure of graphs. A walk in a graph G is an alternating sequence.

$W : v_0, e_1, v_1, e_2, ..., v_{n-1}, e_n, v_n \ (n \geq 0)$ of vertices and edges, beginning and ending with vertices, such that $e_i = v_{i-1} \ v_i, \ i = 1, 2, ..., n.$ It is denoted by $(v_0 - v_n)$ walk. The number of edges (not necessarily distinct) is called the length of walk. In graph G, $u, e_1, v, e_2, w, e_6, x, e_4, u$ is a walk of length 4.

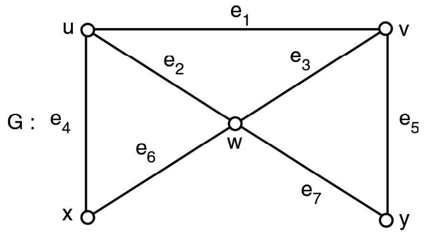


Fig. 3.12 *Structure of Graph*

A trail is a walk in which no edge is repeated and a path is a trail in which no vertex is repeated. Thus, a *path* is a trail, but not every trail is a path. In the above graph G,

$x, e_6, w, e_3, v, e_1, u, e_2, w, e_7, y$ is a trail that is not a path, and $u, e_4, x, e_6, w, e_3, v,$ is a path.

Result: Every $(u - v)$ walk in a graph contains a $(u - v)$ path.

Proof: Let W be a $(u - v)$ walk in a graph G. If $u = v$, then w is the trail path, i.e., walk of length zero.

Suppose $u \neq v$ and $W : u = u_0, u_1, u_2, ..., u_n = v.$ If no vertex of G appears in W more than once, then w itself is a $(u - v)$ path. Otherwise, there are vertices of G that occur in w twice or more. Let i and j be distinct positive integers such that $i < j$ with $u_i = u_j$. Then say $u_i, u_{i+1}, ..., u_{j-2}, u_{j-1}$ are removed from w, and the resulting sequence is $(u - v)$ walk w_1 whose length is less than that of w. (By induction hypothesis, this w_1 contains a $(u - v)$ path and hence w has a $(u - v)$ path). If no vertex of G appears more than once in w_1, then w_1 is a $(u - v)$ path. If not, apply the above procedure, until we get a $(u - v)$ path.

Cycle: A cycle is a walk. $v_0, v_1, ..., v_n$ is a walk in which $n \geq 3, \ v_0 = v_n$ and the 'n'-vertices $v_1, v_2, \ldots, v_n$ are distinct. We say that a $(u - v)$ walk is closed if $u = v$ and open if $u \neq v.$

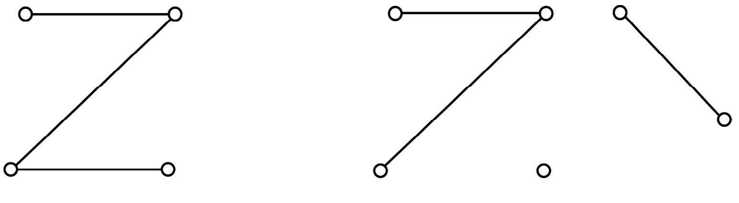
Connection: Let u and v be vertices in a graph G. We say that u is connected to v if G contains a $(u - v)$ path. The graph G is connected, if u is connected to v for every pair u, v of vertices of G.

Disconnection: A graph G is disconnected, if there exists two vertices u and v for which there is no $(u-v)$ path.

Component: A sub-graph H of a graph G is called a component of G, if H is a maximal connected sub-graph of G and component is denoted by $\omega(G)$.

Note: If $\omega(G) > 1$, then G is disconnected.

For example,



(*i*) (*ii*)

Fig. 3.13 *Components of Graph*

Graph (*i*) is connected and (*ii*) is disconnected.

Note that graph (*ii*) has 3 components.

Connectedness in Directed Graph

Strongly Connected: A directed graph is strongly connected if there is a path from u to v and v to u, whenever u and v are vertices in the graph.

Weakly Connected: A directed graph is weakly connected, if there is a path between any two vertices in the underlying undirected graph.

Unilaterally Connected: A directed graph is said to be unilaterally connected, if in the two vertices u and v, there exists a directed path either from u to v or from v to u.

For example,

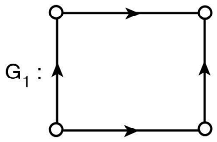


Fig. 3.14 *G_1 is Weakly Connected*

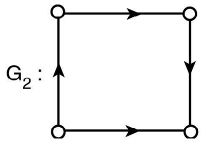


Fig. 3.15 *G_2 Unilaterally Connected*

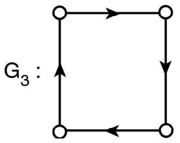
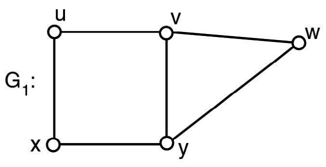


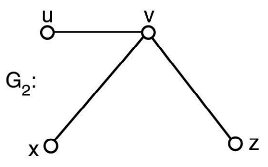
Fig. 3.16 *G_3 is Strongly Connected*

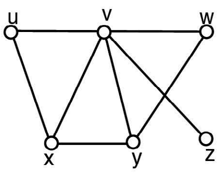
Operations on Graphs

i. The union of two simple graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ is the simple graph with vertex set $V_1 \cup V_2$ and edge set $E_1 \cup E_2$ and is denoted by $G_1 \cup G_2$.

   For example,



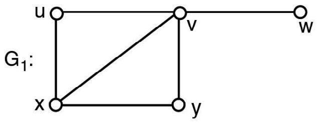


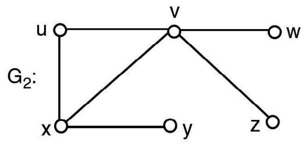


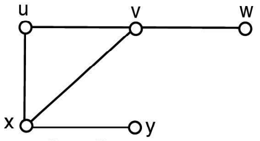
$G_1 \cup G_2$

ii. The intersection of two simple graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ is the simple graph with vertex set $V_1 \cap V_2$ and edge set $E_1 \cap E_2$ and is denoted by $G_1 \cap G_2$. (Note that for $G_1 \cap G_2$, $V_1 \cap V_2$ is non-empty always.)

For example,







$G_1 \cap G_2$

iii. The ring sum of two graphs G_1 and G_2 is a graph consisting of the vertex set $V_1 \cup V_2$ and edges that are either in G_1 or in G_2, but not in both and is denoted by $G_1 \oplus G_2$.

$$\text{i.e., if} \quad G_1 = (V_1, \ E_1); \quad G_2 = (V_2, E_2)$$
$$\text{then} \quad G_1 \oplus G_2 = (V_1 \cup V_2, \quad E_1 \mathit{\Delta} E_2),$$

Where Δ is the symmetric difference.

3.7 EULER GRAPHS

A trail that traverses every edge of *G* is called an Euler trail of *G*. A circuit (tour) of *G* is a closed walk that traverses each edge of *G* atleast once. An Euler tour is a tour which traverses each edge exactly once. A graph is Eulerian if it contains an Euler tour.

Theorem 3.3: A connected graph is Eulerian iff it has no vertices of odd degree.

Proof: Let *G* be Eulerian and let *C* be an Euler tour of *G*, which begins and ends at some vertex *u*.

Claim: *G* has no vertices of odd degree, i.e., to prove that every vertex of *G* is even. Consider a vertex $w \neq u.$ Since *w* is neither the first nor the last vertex of *C*, each time *w* is encountered, it is reached by some edge and left by another edge. Hence each occurrence of *w* in *C* contributes 2 to its degree. Thus *w* is of even degree. This is true for all internal vertices of *C*. The initial occurrence and final occurrence of the vertex *u* in *C* contributes 1 to the degree of *u*. Therefore, every vertex of *G* is of even degree.

Conversely, let us assume that every vertex of a connected graph *G* is even.

Claim: *G* is Eulerian.

Suppose *G* be a connected non-Eulerian graph with no vertices of odd degree.

Among such graphs, choose one, say *G* having least number of edges.

Since each vertex of *G* has atleast two edges, *G* contains a trail. Let *C* be a closed trail of maximum possible length in *G*. By assumption, *C* is not a Euler circuit of *G* and hence $G - E(C)$ has edges.

Therefore $G-E(C)$ has some component G' with edges. Since C itself is Eulerian, degree of every vertex in C is even. Hence degree of every vertex in $G-E(C)$ is also even. Therefore degree of every vertex in G' is even. Since $E(G')<E(G)$.

G' is Eulerian and hence G' has an Euler circuit (tour) say C'. Since G is connected, there is a vertex v in $V(C)\cap V(C')$ and we may assume without loss of generality that v is the initial and the terminal vertex of both circuits C and C'. Now $(C\cup C')$ is a closed trail of G with $E(C\cup C')>E(C)$. This contradicts the choice of C. Hence, every non-empty connected graph with no vertices of odd degree is Eulerian.

For example,

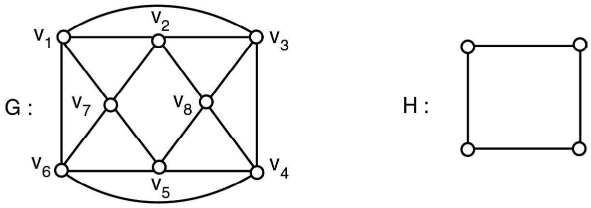


Fig. 3.17 Eulerian Graphs

G and H are Eulerian graphs.

Theorem 3.4: A connected graph G has an Eulerian trail iff G has exactly two odd vertices.

Proof: Let G be a connected graph with an Eulerian $(u-v)$ trail. By the similar argument in the previous theorem, we conclude that all the vertices on the trail except u and v, have even degree. Conversely, let G be connected graph with two odd vertices u and v. Let G' be the graph obtain from G by adding a new edge $e = uv$ between u and v. By applying the previous theorem to G', we can obtained an Eulerian tour in which the edge e is the first edge. Hence, this Eulerian trail of G can be obtained that starts at v and ends at u. Therefore, G has an Eulerian trail.

Eulerian Digraphs

An Eulerian trail of a connected directed graph D, is a trail of D that contains all the edges of D; while an Eulerian circuit of D, is a circuit which contains every edges of D. A directed graph that contains an Eulerian circuit is called Eulerian digraph.

For example,

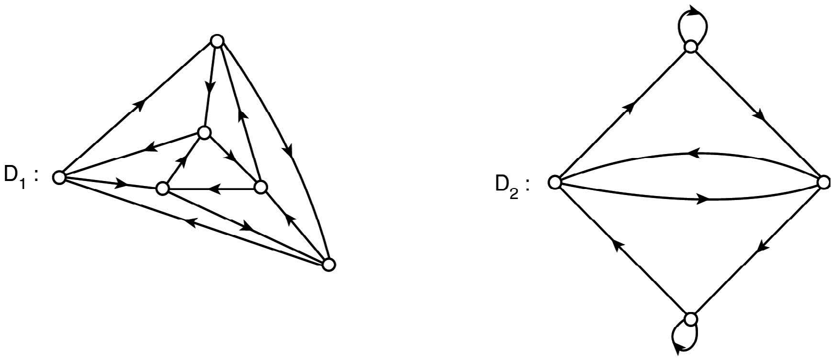


Fig. 3.18 Eulerian Digraphs

Theorem 3.5: Let D be a connected directed graph. D is Eulerian iff $d^+(v) = d^-(v)$, $\forall v \in G$, G is called balanced digraph.

Proof: Let D be an Euler directed graph. Then D contains an Euler circuit C with common initial and terminal vertex v. Let b_u be the number of occurrence of an internal vertex u in C.

Whenever C enters u through some edge incident into u, there is another edge incident out of u through which C leaves u. Thus, each occurrence of u contributes one in-degree and one out-degree. Moreover, C contains all the edges of D. Thus,

$d^+(u) = d^-(u) = bu.$ Similarly $d^+(v) = d^-(v)$

Hence, $d^+(v) = d^-(v), \ \forall v \in V(D)$

Conversely, suppose the connected digraph D is balanced. Then, for each vertex u, $d^+(u) = d^-(u) \neq 0$. Start with an arbitrary vertex $u_1, d^+(u_1) \neq 0.$ There exists an edge, incident out of u_1. Let u_2 be the terminal vertex of this edge, $d^+(u_2) \neq 0.$ Hence, there exists an edge, incident out of u_2. Continuing like this, we reach a vertex which has been traversed directly. Thus, we obtain a directed circuit C_1 in D. If $E(C_1) = E(D),$ then, C_1 is the required Euler circuit. If not, i.e., $E(C_1) \neq E(D),$ then remove all the edge of C_1 from D to obtain a spanning subgraph D_1. Since D is balanced, D_1 is also balanced. Applying the above process to D_1, we will obtain a circuit C_2 in D_1. If $E(D) = E(C_1) \cup E(C_2)$ and C_1 then C_2 can be combined to obtain an Euler circuit in D_1. Otherwise, we remove the edges of C_2 from D_1 to obtain a spanning subgraph D_2 of D. We repeat the above process in D_2 and after a finite number of steps, we obtain edge disjoint circuits C_1, C_2,..., C_k such that $E(D) = E(C_1) \cup E(C_2) \cup ... \cup E(C_k)$. Since D is connected, any two of these cycles have a common vertex. Then the circuits C_1, C_2,...,C_k can be combined to obtain an Euler circuit in D. Hence, D is an Euler graph.

Check Your Progress

5. What is component?
6. Define Euler trail.

3.8 MATRIX REPRESENTATION OF GRAPHS

Incidence and Adjacency Matrices

To any graph G, there corresponds a $V \times E$ matrix called the incidence matrix of G and is denoted by $I(G) = [a_{ij}]_{V \times E}$, where

$$a_{ij} = \begin{cases} 1, \text{ if } j\text{th edge is incident with } i\text{th vertex} \\ 0, \text{otherwise} \end{cases}$$

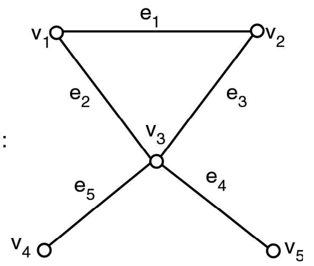
One more matrix associated with graph G is the adjacency matrix, e is denoted by $A(G) = [b_{ij}]_{V \times V}$,

$$a_{ij} = \begin{cases} 1, \text{ if } j\text{th edge is incident with } i\text{th vertex} \\ 0, \text{ otherwise} \end{cases}$$

NOTES

Some authors used to define a_{ij} as the number of times (0, 1, and 2) v_i and e_j are incident ; b_{ij} is the number of edges v_i and v_j.

For example,



	e_1	e_2	e_3	e_4	e_5
v_1	1	1	0	0	0
v_2	1	0	1	0	0
v_3	0	1	1	1	1
v_4	0	0	0	0	1
v_5	0	0	0	1	0

I(*G*), incidence matrix of *G*

	v_1	v_2	v_3	v_4	v_5
v_1	0	1	1	0	0
v_2	1	0	1	0	0
v_3	1	1	0	1	1
v_4	0	0	0	1	0
v_5	0	0	0	1	0

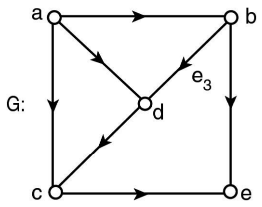
A(*G*), adjacency matrix of *G*

The adjacency matrix $A(G) = [b_{ij}]$ of a directed graph is also a $V \times V$ matrix,

$$\text{Where} \quad b_{ij} = \begin{cases} 1, \text{ if there is a directed edge from } v_i \text{ to } v_j \\ 0, \text{ otherwise} \end{cases}$$

(Similarly one can define the incidence matrix of a directed graph)

For example,

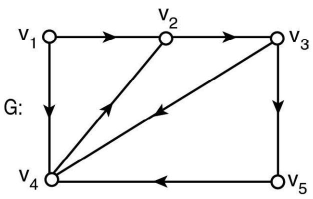


$$A(G) = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Example 3.3: Write the adjacency matrix of the following graphs:

Solution:

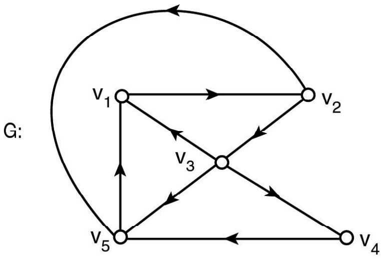
(*i*)



$$A(G) = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

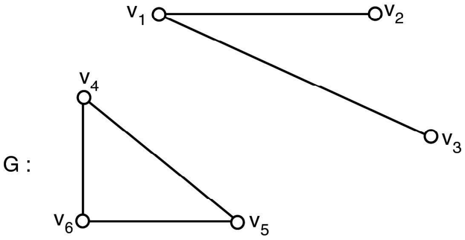
NOTES

(*ii*)



$$A(G) = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

(*iii*)



$$A(G) = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$

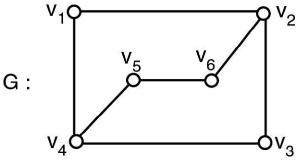
Notes: From Example 3.3 one can conclude that:

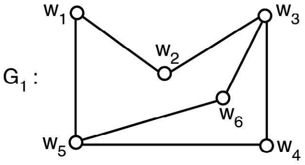
1. The diagonal entries of an adjacency matrix are all zero, iff the graph is a graph with no self-loops.
2. If G is disconnected and it has two components, then its adjacency matrix $A(G)$ can be written as,

$$A(G) = \begin{bmatrix} A(G_1) & 0 \\ 0 & A(G_2) \end{bmatrix}, G_1 \text{ and } G_2 \text{ are components.}$$

With the help of these matrices, one can verify whether the given graphs are isomorphic or not.

Example 3.4: Verify if G and G_1 are isomorphic.





Solution: First we shall write the adjacency matrices of G and G_1.

$$A(G) = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 \end{bmatrix} \qquad A(G_1) = \begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 \end{bmatrix}$$

By keeping one matrix fixed, and by applying permutation of rows and corresponding columns permutations on the unfixed matrix, yields the fixed one. Then the given graphs are isomorphic.

Here keep $A(G)$ fixed.

Also G and G_1 have 4 vertices of degree 2 and two vertices of degree 3. Since $d(v_1) = 2$ and v_1 is not adjacent to any other vertex of degree 2, corresponding vertex in G_1 is either w_4 or w_6, the only vertices of degree 2 in G_1 not adjacent to a vertex of degree 2.

Without loss of generality, let us take $v_1 \to w_6$. Suppose this $v_1 \to w_6$ is not ending with isomorphism, we have to take $v_1 \to w_4$.

Similarly, for other vertices of G, we can set $v_2 \to w_3; v_3 \to w_4;$ $v_4 \to w_5; v_5 \to v_1; v_6 \to v_2$.

Thus, we can modify $A(G_1)$ as

$$A(G_1) = \begin{array}{c} \\ w_6 \\ w_3 \\ w_4 \\ w_5 \\ w_1 \\ w_2 \end{array} \begin{array}{c} \begin{array}{cccccc} w_6 & w_3 & w_4 & w_5 & w_2 & w_1 \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

$\therefore\ A(G) = A(G_1)$ and hence $G \cong G_1$.

3.9 TREES

In this section we shall study the characteristics of a tree.

Acyclic Graph: A graph G which has no cycles is called an acyclic graph.

Tree: A connected acyclic graph G is called a tree.

For example,

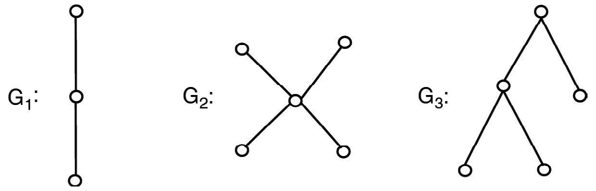


Fig. 3.19 *Trees*

Notes:

1. Trees are often known as open graphs.
2. Any organizational hierarchy is also an example of tree.

Theorem 3.6: Every two vertices in a tree, are joined by a unique path.

Proof: By contradiction: Let G be a tree and assume that there are two distinct (v, w) paths P_1 and P_2 in G. Since $P_1 \neq P_2$, there is an edge $e = V_1V_2$ of P_1 that is not in P_2. Clearly $(P_1 \cup P_2) - e$ is connected. Therefore it contains a $(V_1–V_2)$ path P. Now $P + e$ is a cycle in the acyclic graph G, which is a contradiction to the fact that G is a tree.

Theorem 3.7: If G is a tree of n vertices, then G has $(n-1)$ edges.

Proof: By induction on the number of vertices.

When $n = 1$, $E(G) = 0 = n - 1\ (G \cong K_1)$

When $n = 2$, $E(G) = 1 = n - 1\ (G \cong K_2)$

Let us assume that this theorem is true for all trees of G with fewer than n vertices.

Now, let G be a tree on n vertices. Let $e = uv$ be an edge in G. Then $G - e$ is disconnected and G has two components say G_1 and G_2 of $G - e$. Since G is acyclic, G_1 and G_2 are also acyclic and hence G_1 and G_2 are also trees. Moreover G_1 and G_2 has fewer than n vertices say n_1 and n_2 respectively. Therefore, by induction hypothesis,

G_1 has $(n_1 - 1)$ edges and G_2 has $(n_2 - 1)$ edges.

$\therefore E(G) = E(G_1) + (G_2) + 1$ (Here 1 in the sum corresponds to the edge e)

$$= (n_1 - 1) + (n_2 - 1) + 1$$
$$= n_1 + n_2 - 1$$
$$= n - 1$$

Therefore, an n vertex tree has $(n - 1)$ edges.

Theorem 3.8: Every tree has atleast two vertices of degree one in a tree, i.e., there are atleast two pendant vertices.

Proof: Let G be a tree on n vertices. Then,

$$d(v) \geq 1, \ \forall\, v \in v(G) \tag{3.3}$$

Already we have, "$\sum_{v \in v} d(v) = 2.E(G) = 2.e$" (3.4)

Since G is an n-vertex tree, it has $(n - 1)$ edges.

$$\therefore \sum_{v \in v(G)} d(v) = (2n - 2) \tag{3.5}$$

From Equations (3.3) and (3.5), it follows that $d(v) = 1$ for atleast two vertices.

Note: In a tree, every edge is a cut-edge.

Rooted and Binary Trees

Rooted Tree: In a directed tree (every edge assigned with a direction), a particular vertex is called a root if that vertex is of degree zero. A tree together with its root produces a graph called a rooted tree.

(Note that in the rooted tree, every edge is directed away from the root)

For example,

Suppose T is a rooted tree. If a vertex u is a vertex in T other than the root, the parent of u is the unique vertex u_1 such that there is a directed edge from u_1 to u. Here u is called as a child of u_1. Vertices of the same parent are called as siblings. A vertex of a rooted tree is called as a leaf if it has no children and those vertices which have children, are called as internal vertices.

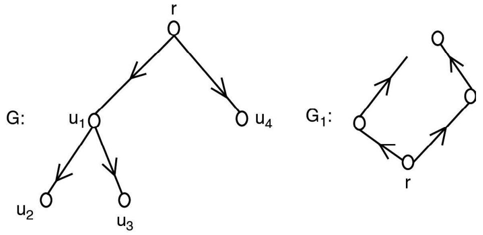


Fig. 3.20 *Rooted Trees*

NOTES

If v is a vertex in a tree, the subtree with v as its root is the subgraph of the tree consisting of v and its children and all edges incident to these children.

For example,

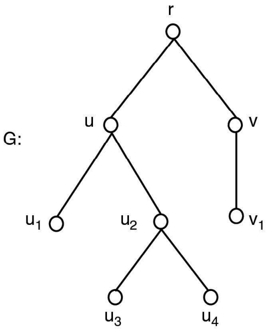


Fig. 3.21 *Rooted Tree T*

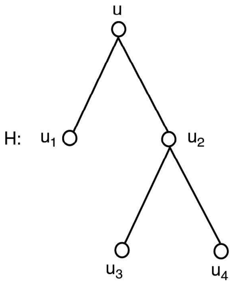


Fig. 3.22 *Subtree of T with its Root u*

k-Ary Tree: A rooted tree is called a k-ary tree if every internal vertex has, not more than k-children. The tree is called a full k-ary tree if every internal vertex has, exactly k-children. A k-ary tree with $k = 2$ is called a binary tree.

For example,

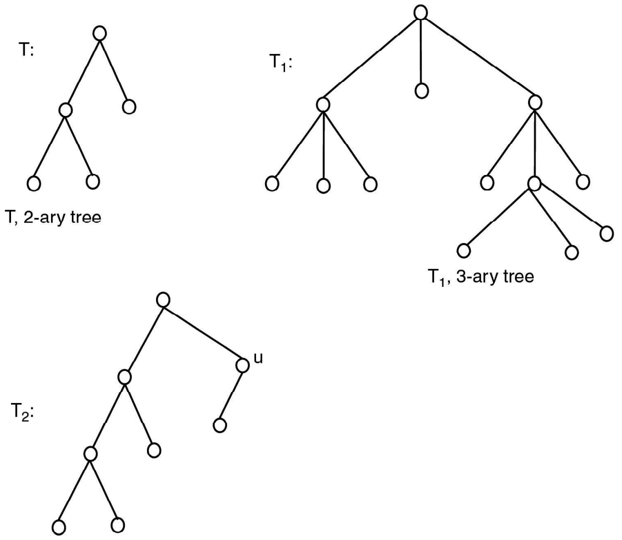


Fig. 3.23 *K-ary Trees*

T_2, not a 2-ary tree. (vertex u has only one child, whereas all the other vertices have two children).

A tree T is called as a binary tree if there is only one vertex with degree 2 and the remaining vertices are of degree 1 or 2.

Example 3.5: Prove that a full k-ary tree with i-internal vertices contains k_{i+1} vertices.

Solution: In a full k-ary tree, every internal vertex has k-children and hence a full k-ary tree with i-internal vertices can have ki vertices. If we include the root, the tree has $k_i + 1$ vertices. By looking at the full k-ary tree, we can observe the following:

(*i*) n vertices has $i = (n - 1)/k$ internal vertices and $p = [(k - 1)n + 1]/k$ leaves.

(*ii*) i internal vertices has $n = ki + 1$ vertices and $p = (m - 1)i + 1$ leaves.

(*iii*) p leaves has $n = (kp - 1)/(k - 1)$ vertices and $i = (p - 1)/(k - 1)$ internal vertices.

NOTES

Level and height in a rooted tree: The level of a vertex v in a rooted tree is the length of the path from the root to this vertex. The height of a rooted tree is the length of the longest path from the root to any vertex.

For example,

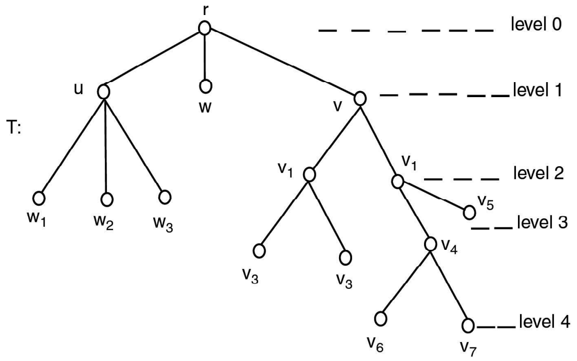


Fig. 3.24 *Levels of Trees*

A rooted tree T with its levels. Height of T is 4.

Balanced Tree: A rooted k-ary tree of height h is balanced if all the leaves are at level h or $(h - 1)$.

Application of Trees

In this section, we shall discuss problems using trees.

Binary Search Trees

Binary search tree is a binary tree in which each child is either a left or right child; no vertex has more than one left child and one right child, and the data are associated with vertices.

Example 3.6: Build a binary search tree for the words banana, peach, apple, pear, coconut, mango and papaya using the alphabetical order.

Solution:

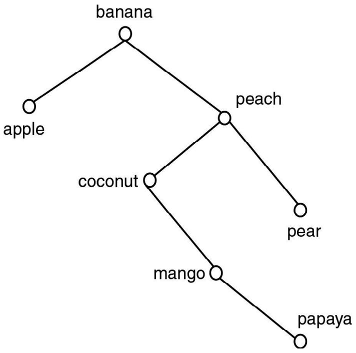


Fig. 3.25 *Binary Search Tree*

For if apple < peach, coconut < pear.

Further mango is the right child of coconut and papaya is the right child of mango.

Decision Trees

A rooted tree in which each internal vertex is assigned to a decision with a subtree at the vertices, then each possible outcome of the decision is called a decision tree.

NOTES

Traversal of a Tree

A systematic method for visiting every vertex of an ordered rooted tree is called as a 'Traversal algorithm'.

Pre-Order: Let T be an ordered rooted tree with root r. Suppose T has one and only vertex say r, then r is the pre-order traversal of T. Suppose that $T_1, T_2, ..., T_k$ are the subtrees at r from left to right in T, then pre-order traversal begins by visiting r. It continues by traversing T_1 in pre-order, then T_2 in pre-order and so on, until T_k is reached.

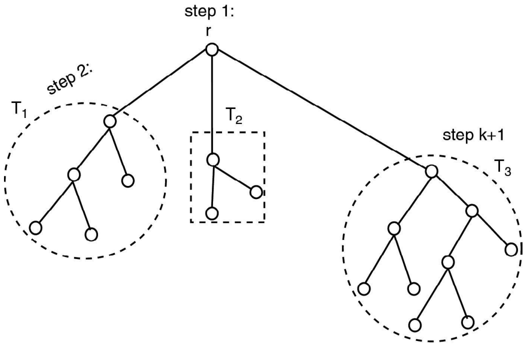


Fig. 3.26 *Pre-Order Traversal*

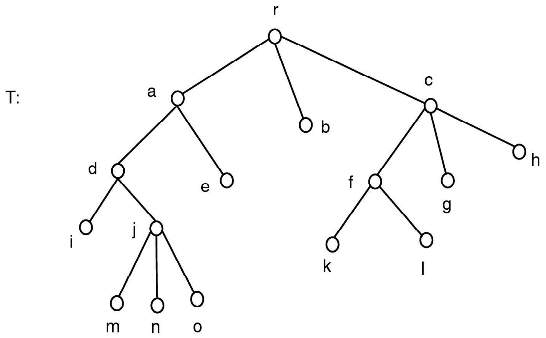
Step 1: Visit the root r.

Step 2: Visit T_1 in pre-order.

Step 3: Visit T_2 in pre-order.

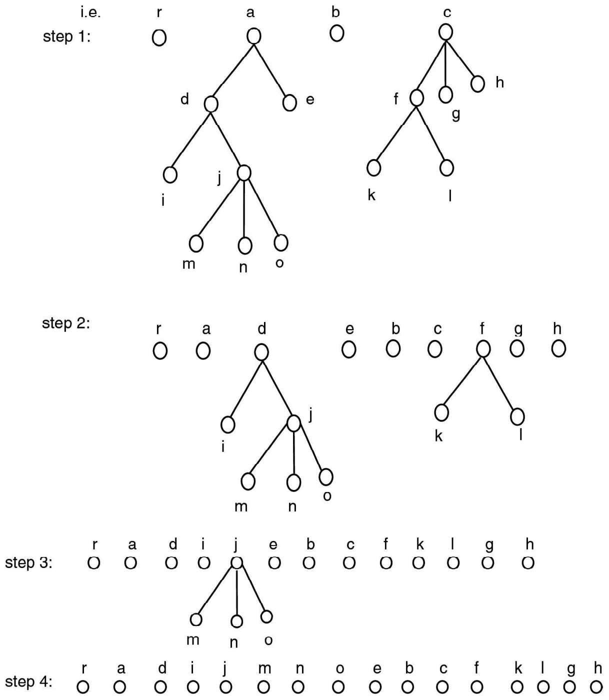
Step k+1: Visit T_k in pre-order.

Let us try to understand the above with an example.



Let T be an ordered root tree. The steps of the pre-order traversal of T are as follows:

We traverse T in pre-order by listing the root r, followed by the pre-order list of subtree with root a, the pre-order list of subtree with root b, and the pre-order list of subtree with root c.



Algorithm: Pre-Order Traversal

Step 1: Visit root *r* and then list *r*.

Step 2: For each child of *r* from left to right, list the root of first subtree then next subtree and so on until we complete listing the roots of subtrees at level 1.

Step 3: Repeat Step 2, until we arrive at the leaves of the given tree.

Step 4: Stop.

In-Order Traversal: Let T be an ordered rooted tree with its root at vertex r. Suppose T consists only root r, then r is the in-order traversal of T. If not, i.e., suppose T has subtrees $T_1,T_2,...,T_k$ at r from left to right. The in-order traversal begins by traversing T_1 in-order, then visiting r. It continues by traversing T_2 in-order, then T_s inorder and so on and finally T_k in-order.

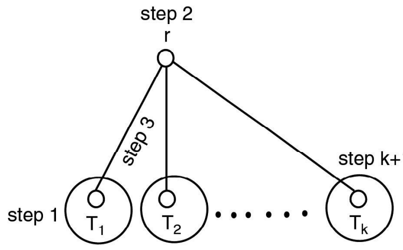


Fig. 3.27 *In-Order Traversal*

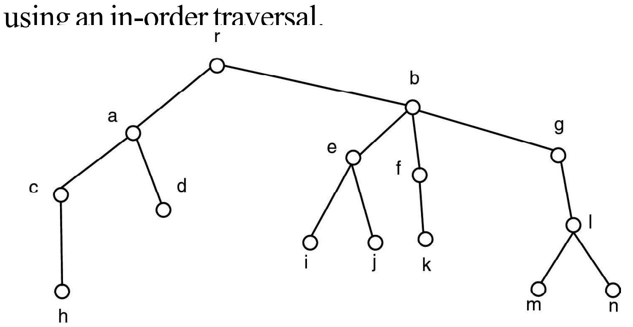
Step 1: Visit T_1 in-order.

Step 2: Visit root.

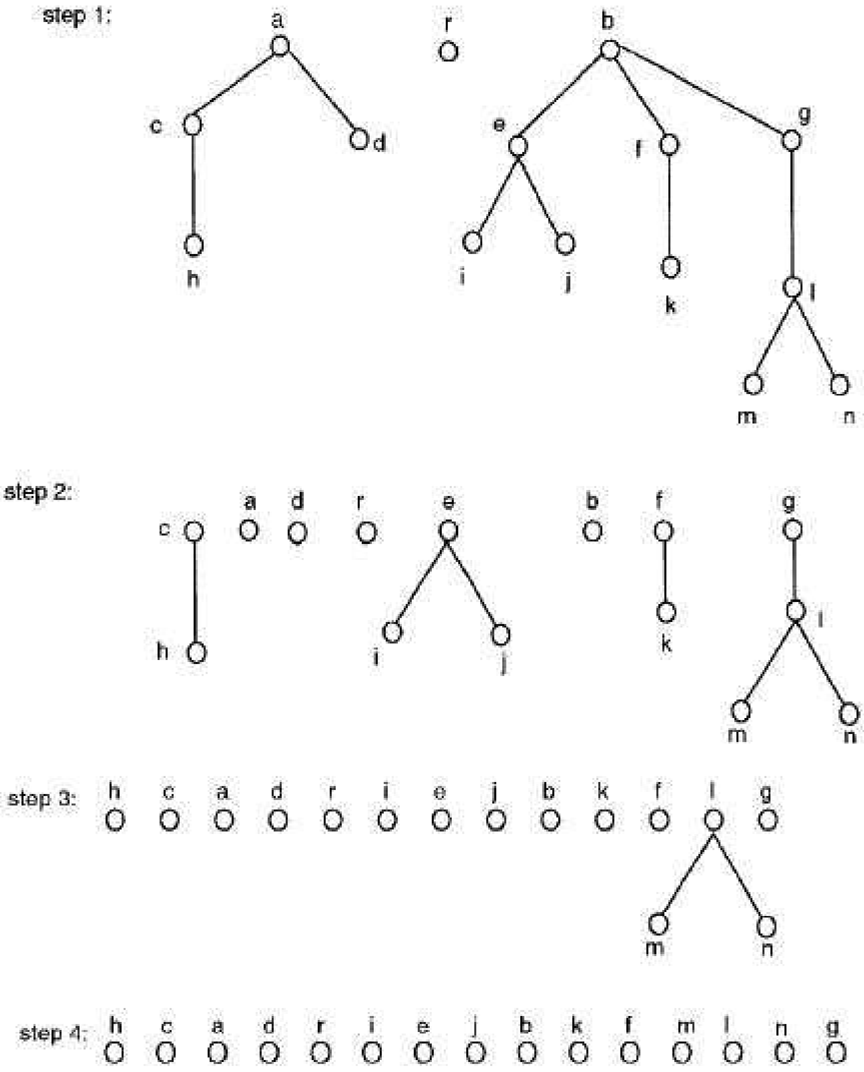
Step 3: Visit T_2 in-order.

Step k+1: Visit T_k in-order.

Example 3.7: Determine the order in which the vertices of the following rooted tree is visited using an in-order traversal.



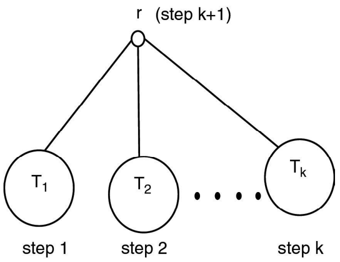
Solution: The in-order traversal begins with an in-order traversal of the subtree with root at *a*, followed by the root *r*, and the in-order listing of the subtree with root *b*.



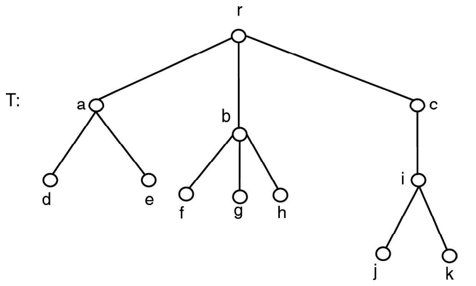
Definition Post-Order Traversal

Let *T* be an ordered rooted tree with root *r*. If *T* has only one vertex *r*, then *r* is the post-order traversal of *T*. But if *T* has subtrees $T_1, T_2, ..., T_k$ at *r* from left to right, the post-order traversal begins by traversing T_1 in post-order, then T_2 in post-order and so on until T_k and ends by visiting *r*.

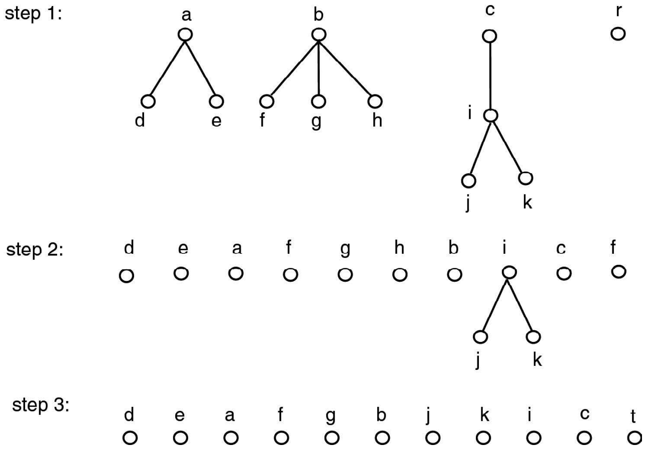
For example,



For example,



The post-order traversal begins with the post-order traversal of the subtree with root *a*, the post-order traversal of the subtree with root *b* and the post-order traversal of the subtree with root *c*, followed by the root *r*.



Path Length

In a rooted tree, every vertex has a path length which is given by the number of edges it has to traverse from the root to that vertex. Every vertex has a unique path length. A tree has been shown here. To find the path length of any vertex, one has to start from the root and travel up to that vertex. For example: Path length of vertex D, E, F and G is 2 whereas for H, I and J it is 3 and for K and L it is 4.

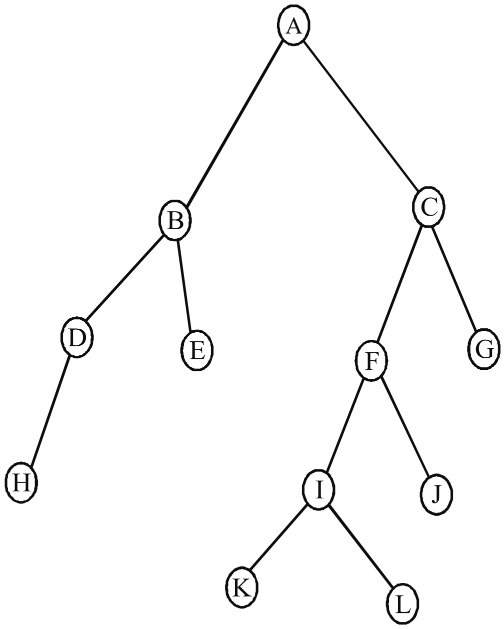


Fig. 3.28 Path Length

Spanning Trees

In this section, we shall study the spanning acyclic subgraph of a connected subgraph and its optimality.

Let *G* be a simple connected graph. A spanning tree of *G* is a subgraph of *G*, i.e., a tree containing every vertex of *G*.

For example,

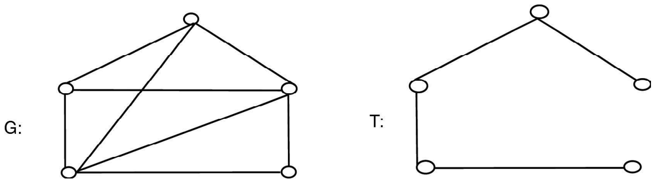


Fig. 3.29 Simple Graph G and its Spanning Tree T

Theorem 3.9: A simple graph is connected if there exists atleast one spanning tree.

Proof: Let *G* be a simple connected graph.Suppose *G* has no circuits then *G* itself is a spanning tree. Suppose *G* has a simple circuit. By deleting an edge from one of these simple circuits, the resulting subgraph is still connected if it is a spanning subgraph. If this subgraph has simple circuits, then delete an edge from one of these simple circuits. Repeat this process until no simple circuits are there. Thus in this manner a tree *T* is obtained in which $V(T) = V(G)$. Therefore *T* is a spanning tree of *G*.

Note: The converse of the above theorem is obvious.

Depth-First Search and Breadth-First Search

We can build the spanning tree of a connected graph using DFs and BFs. First we shall see how DFs are useful in construction of a spanning tree from a given connected graph.

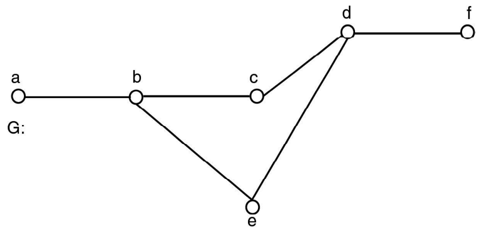
Depth-First Search

Let G be the given connected graph. Arbitrarily choose a vertex as the root. Find a path starting from this choosen vertex by successively adding edges, where each edge is incident with the last vertex in the path and a vertex not already in the path. Continue adding edges to this path as long as possible. If this path consists of all the vertices of G, this path is the required spanning tree. If not, more edges should be added. Move back to the next to last vertex in this path, and if possible, form a new path starting at this vertex passing through vertices that were not already visited. If this is not possible, move back to another vertex in this path (i.e., 2 vertices back from the last) and try again. Repeat this procedure, beginning at the last vertex visited, moving back up the path one vertex at a time, forming new long paths until no more edges can be added. This process ends with a spanning tree.

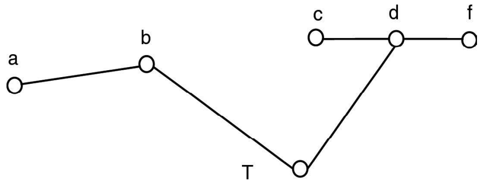
NOTES

When this procedure returns to vertices previously visited, it is also called as backtracking.

Example 3.8: Construct a spanning tree for the following graph G.



Solution: First we choose arbitrarily a vertex say e as the root. Form a path at e, i.e., $c\ d\ f$ is the path. Backtrack to d. Form a path beginning at d in such a way that it has to visit the vertices which where not visited in the previous path, $d\ e\ b\ a$. Since all the vertices of G are visited, this procedure gives the spanning tree T.



Breadth-First Search

First choose a vertex arbitrarily as the root. Add the edges of G which are incident with this vertex. The new vertices added at this stage becomes level 1 in the spanning tree. Order these vertices arbitrarily. Next, for each vertex at level 1 visited in order, add each edge incident to this vertex to the tree as long as it does not create a simple circuit. Order the children of each vertex at level 1 arbitrarily. This produces the vertices at level 2 in the tree. Continue in this manner until all the vertices of G have been added. Ultimately we end with a spanning tree.

Prim's Algorithm

Let G be a connected graph.

Step 1: Choose arbitrarily a vertex say v_1 and an edge e_1 with minimum weight among the edges incident with v_1.

Step 2: Having selected the vertices $v_1,v_2,..,v_k$ and the edges $e_1,e_2,..,e_k$ choose the edge e_{k+1} as follows. e_{k+1} is incident with any one of the vertices $\{v_1,v_2,..,v_k\}$ and incident with $v(G) - \{v_1,v_2,..,v_k\}$. Moreover the subgraph formed with $v_1,v_2,...,v_k$, v_{k+1} and the edges $e_1,e_2,..,e_k,e_{k+1}$ is acyclic and of the remaining edges e_{k+1} has minimum weight.

Step 3: Repeat Steps 1 and 2 till $(n-1)$ edges are arrived at.

For example,

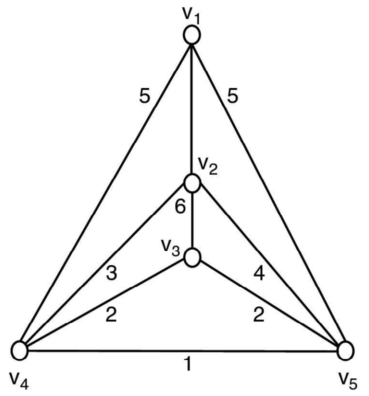
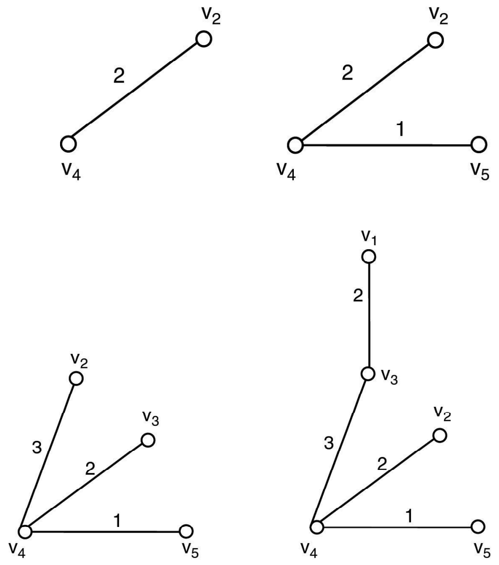


Fig. 3.30 *Prim's Algorithm*

Step 1: Choose arbitrarily vertex v_3 and apply Step 2 and Step 3. Now we get the spanning tree.



Weight of the spanning tree is 8.

Kruskal's Algorithm

Let G be a connected graph on n vertices.

Step 1: Arrange the edges in ascending order according to their weights. Choose the minimum weight edge say e_1.

Step 2: Having selected $e_1,e_2,...,e_k$ in such a way that the subgraph formed by these edges $<e_1,e_2,...,e_k>$ is acyclic, choose e_{k+1} such that of the remaining edges, weight of e_{k+1} is minimum.

Step 3: Repeat Steps 1 and 2 until $(n-1)$ edges are selected.

For example,

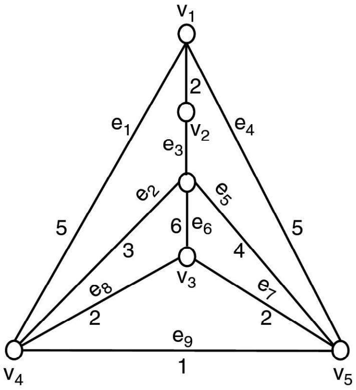
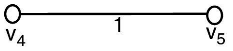


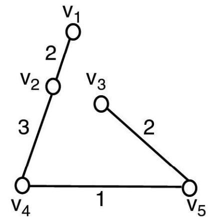
Fig. 3.31 Kruskal's Algorithm

Step 1: $e_9, e_7, e_8, e_3, e_2, e_5, e_{4,} e_1, e_6$

Among these equations e_9 has the minimum weight 1.



After applying Step 2 and Step 3, we get the spanning tree as,



Weight of the optimal spanning tree is 2 + 3 + 1 + 2 = 8.

3.10 PLANAR GRAPHS

A graph G is said to be *planar* if there exists some geometric representation of G which can be drawn on a plane such that no two of its edges intersect ('meeting' of edges at a vertex is not considered an intersection). A graph that cannot be drawn on a plane without a crossover between its edges is called nonplanar. A drawing of a geometric representation of a graph on any surface such that no edges intersect is called embedding.

Kuratowski's Two Graphs and Euler's Formula: Statement and Corollary

To show a graph G is nonplanar we have to prove that of all possible geometric representations of G, none can be embedded in a plane.

Theorem 3.10: The complete graph of fine vertices is nonplanar.

Proof: Let the fine vertices in the complete graph be v_1, v_2, v_3, v_4 and v_5. Using the definition of complete graph, we must have a circuit going from v_1 - v_2 - v_3 - v_4 - v_5 to v_1, i.e., a pentagon. This pentagon must divide the plane of the paper into two regions, one inside and the other outside.

Since v_1 is to be connected to v_3 by means of an edge, this edge may be drawn inside or outside the pentagon. Suppose that we choose to draw a line

NOTES

from v_1 to v_3 inside the pentagon, we have to draw an edge from v_2 to v_H and another one from v_2 to v_5. Since neither of these edges can be drawn inside the pentagon without crossing over the edge already drawn, we draw both these edges outside the pentagon. The edge connecting v_3 and v_5 cannot be drawn outside the pentagon without crossing the edge between v_2 and v_4. Therefore v_3 and v_5 have to be connected with an edge inside the pentagon.

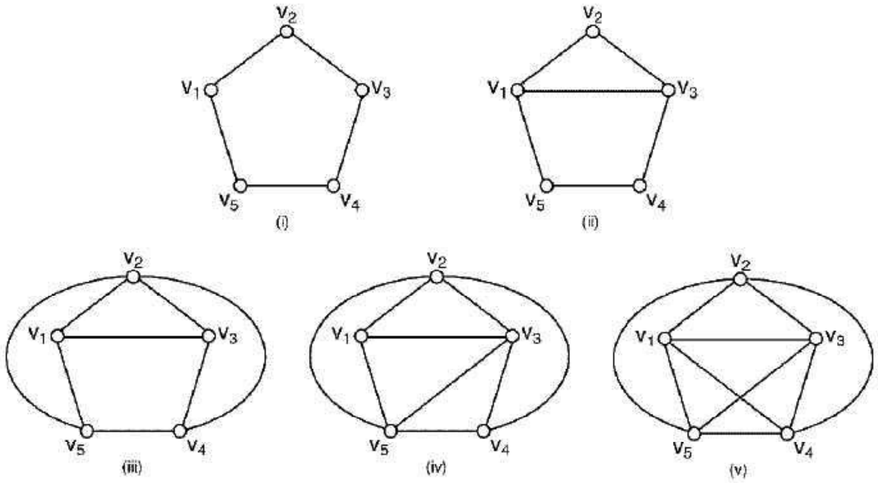


Fig. 3.32

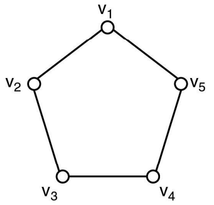
Note: Complete graph is nothing but a simple graph in which every vertex is joined to every other vertex by means of an edge.

Theorem 3.11: Kuratowski's (Polish mathematician) second graph is also nonplanar. $k_{3,3}$ is nonplanar.

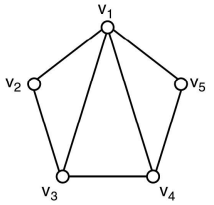
Note: In the plane, a continuous non-self intersecting curve whose origin and terminus coincide is said to be a jordan curve. If *j* is a jordan curve in the plane π, then $\pi - j$ is a union of two disjoint connected open sets called the interior and the exterior of *j*.

For example, Prove that k_5 is nonplanar.

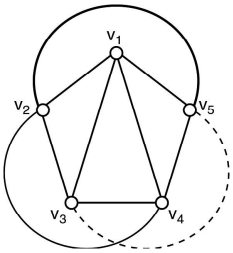
Step 1: Draw a circuit *c* on 5 vertices. This circuit *c* divides the plane into two regions called interior and exterior of *c*.



Step 2: Draw the edges v_1v_3, v_1v_4 in the interior. We cannot draw any more edge in the interior of *c*, without intersecting any edge.



Now draw the edges v_2v_5, v_2v_4 in the exterior of *c*. But the edge v_3v_5 cannot be drawn in the interior or exterior of *c*, without intersecting the edge of *c*.



Thus k_5 is nonplanar.

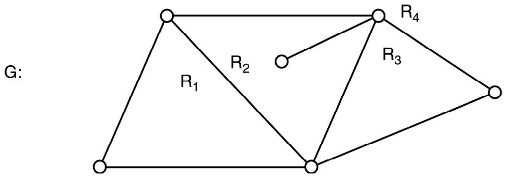
We can prove that $k_{3,\ 3}$ is nonplanar in the following manner.

Proof: Assume that $k_{3,3}$ is planar. Let the vertex of $k_{3,3}$ is $\{v_1,...,v_6\}$. Let $P = \{v_1, v_3, v_5\}$ and $Q = \{v_2, v_4, v_6\}$.

Let C be the cycle of $v_1\ v_2\ v_3\ v_4\ v_5\ v_6\ v_1$. It is a Jordan curve. The other three edges v_1v_4, v_2v_5, v_3v_6 are chords of the cycle C. So either interior of C or exterior of C contains two of the three chords. Say there are two chords in Int c. These two chords must cross each other, which is a contradiction hence $k_{3,3}$ is nonplannar.

Contour: Let G be a connected planar graph. A region of G is the domain of the plane surrounded by edges of the graph such that any two points in it can be joined by a line not crossing any edge. The edges 'touching' a region contain a simple cycle called the contour of the region. Two regions are said to be adjacent if the contours of the two regions have atleast one edge in common.

For example,



G, a planar graph: R_i, $i = 1, 2, 3, 4$ are the regions of G. Here R_4 is the infinite region.

Euler's Formula

If G, a connected planar graph has n vertices, e edges and r regions, then, $n - e + r = 2$

Proof: By induction on e, the number of edges.

If $e = 0$, then $G = K_1$($\because$ G is connected)

$\therefore$ $n = 1$; $r = 1$ (infinite face) $\therefore$ $n - e + r = 1 - 0 + 2 = 2$

If $e = 1$ then $n = 2$ ($\because$ G is connected) and $r = 1$ (infinite face)

$\therefore n - e + R = 2 - 1 + 1 = 2$ $\therefore$ This result is true for $e = 0$ and $e = 1$

Let us assume that this result is true for all the connected planar graphs on $(e - 1)$ edges.

Let G be a connected planar graphs with e edges.

Case 1. If G is a tree with e edges then $n = e + 1$ [$\because$ Tree on n vertices has $(n–1)$ edges]. $r = 1$

$\therefore$ $n - e + r = e + 1 - \mathrm{e} + 1 = 2$

Case 2. If G is not a tree.

Since G is connected, it contains cycles.

Let e_1 be an edge in some simple circuit of G.

Let G_1 be the graph obtained from G by deleting the e_1, i.e., $G_1 = G - e_1$

Now, number of vertices in $G_1 = n$

Number of edges in $G_1 = e–1$

Number of regions in $G_1 = r–1$

Since G_1 has less then e edges, the result is true for G_1 also.

$\therefore$ By induction hypothesis, $n_1 - e_1 + r_1 = 2$, where n_1 is the number of vertices, e_1 is the number of edges and r_1 is the number of regions of G_1 respectively.

$$\therefore\ n - (e - 1) + r - 1 = 2 \Rightarrow n - e + r = 2$$

$\therefore$ In all the cases, the result in true.

Corollary: If G is a connected simple planar graph without loops and has n vertices, $e \geq 2$ edges and r regions, then $3/2\ r \leq e \leq 3n - 6$.

Proof: If $r = 1$ then $3/2 \leq e \leq 3n - 6$ is true, since $e \geq 2$.

If $r > 1$. Let k be the number of edges in the contours of the finite regions.

Since G is simple, each region (finite) is bounded by atleast 3 edges.

Therefore, $k \geq 3\ (r - 1)$ (3.6)

But, in a planar graph, an edge belongs to the contours of atmost two regions and atleast 3 edges touch the infinite region.

$\therefore\ k \leq 2e - 3$ (3.7)

From Equations (3.6) and (3.7), $3r - 3 \leq k \leq 2e–3$

$\Rightarrow\ 3r - 3 \leq 2e - 3$

$\Rightarrow\ 3r \leq 2e\ \Rightarrow\ 3/2\ r \leq e$ (3.8)

Since G is planar, $n - e + r = 2$, by Euler's Formula.

$\therefore\ n - e + 2/3\ e \geq 2$ $[\because$ From Equation (3.8) $r \leq 2/3\ e]$

$\Rightarrow\ 3n - 3e + 2e \geq 6$

$\Rightarrow\ - e \geq - 3n + 6$

$\Rightarrow\ e \leq 3n - 6$ (3.9)

From Equations (3.8) and (3.9), $3/2r \leq e \leq 3n - 6$

Example 3.9: Prove that K_5 is nonplanar.

Solution: Suppose K_5 is planar, then by the above corollory, $e \leq 3n - 6$. In K_5 $n = 5$, $e = 10$;

$\therefore\ 10 \leq 3 \times 5 - 6 = 9$, which is absurd. $\therefore\ K_5$ is nonplanar

Note: K_5, $K_{3,3}$ are called Kuratowski's first graph, second graph respectively.

Corollary: If G is a simple connected planar graph on n vertices, e edges and r regions and does not contain any triangle, then $2r \leq e \leq (2n - 4)$.

Subdivision: A subdivision of a graph G is obtained by inserting vertices (of degree 2) into the edges of G.

For example,

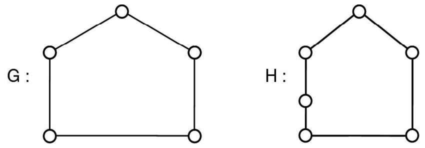


Fig. 3.33 *Subdivision of a Graph*

H is the subdivision of *G.*

Kuratowski Theorem: A graph is planar if it contains no subgraph that is isomorphic to or is a subdivision of K_5 or $K_{3,3}$.

Detection of Planarity

Using planar drawings, it is easy to understand the structure of a given graph by removing crossing edges. These crossing edges are often confused as additional vertices. Graphs in many applications, like road networks or PCBs (Printed Circuit Boards), are planar as they are defined by surface structures.

Every planar graph is *sparse*, which is a remarkable property of planar graphs. For planar graph $G = (V, E)$, Euler's formula gives a relationship in between its edges (E) and vertices (V) which is $|E| \leq 3|V| - 6$. Every linear graph has linear number of edges. Further, every planar graph has a vertex of degree not more than 5.

Algorithms for planarity detection begin by embedding an arbitrary cycle from the graph in the plane. After this, additional paths in *G* between vertices on this cycle are considered. In the event of crossing of two such paths, one must be drawn outside the cycle and another inside. If three such paths cross mutually, the graph is not planar.

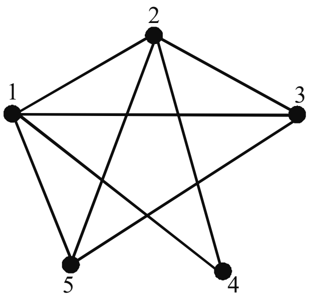


Fig. 3.34 *Given Graph*

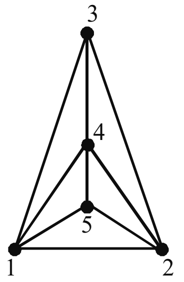


Fig. 3.35 *Redrawn Graph*

In the above figure, the graph for detection of planarity is shown on the left side and redrawn graph is shown on the right side.

3.11 DIJKSTRA'S ALGORITHM

Shortest path problem deals with ways of finding the minimum path distance form a selected node, say s which is source to a destination node *d*. Such problems arise in our real life and also in the field of computer science. For example, vertices of a weighted graph may represent cities and weights on edges represent costs of driving distances between pairs of cities connected by a direct road or rail link.

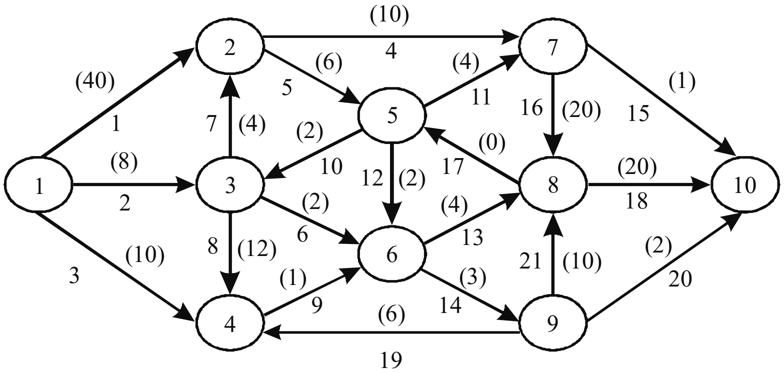
Edsger Dijkstra, a Dutch computer scientist, devised an algorithm to solve this problem and is known as **Dijkstra's algorithm**, on his name. It is an algorithm for graph search solving the single-source shortest path problem for a weighted graph having non-negative edge path costs, producing a tree that gives the shortest path. This algorithm is of great use in routing. The concept of 'shortest path first' finds extensive use in network routing protocols, like IS-IS and OSPF (Open Shortest Path First).

The Algorithm

We take a node as initial node or starting node. We select a destination X and find it's the distance from the **initial node.** This algorithm will assign some initial distance values and then will go step-by-step. These are as below:

1. Assigning every node, a distance value. For initial node, it is zero and for all other nodes, it is infinity.
2. Initially all unvisited nodes are marked. Initial node is set as current.
3. For current node, distance from the initial node to every unvisited neighbour is calculated. For example, if current node (A) has distance of 7, and an edge connecting it with another node (B) is 3, the distance to B through A will be 7 + 3 = 10. If it is less than the previously recorded distance which is infinity in the beginning as in Step 1. The distance is overwritten.
4. When all neighbours of the current node are done with, it is marked as visited so that it is not visited node again and the distance recorded is final and minimal.

In the figure below problem of shortest path is presented. There are 10 nodes and starting node is node 1.



It is required to find the set of paths which is minimum from the source node to another node in the network. This problem of shortest path is a tree which is solved as below:

This has m number of nodes, starting with the source node alone, this procedure makes the number of iterations one less than the number of vertices, i.e., $m - 1$ for finding shortest path and construct a shortest path tree. In the above example, there are 10 number of nodes and will require 9 iterations.

Let S be the the set of nodes already visited. Nodes that are not solved are not in S. In each iteration, a number is assigned to each node. A node d_i denotes the length of minimum distance or shortest path from source node to ith node. After finishing the traversal d_i shows the shortest path to that node. algorithm assigns numbers *di* to each node in the network, where *di* is the length of the shortest path to node *i* from the source node. At the end of the algorithm p*i* is the length of the shortest path to node *i*. Let ***M*** be the set of all edges which is also called arcs.

In the beginning, at source node $S = \{s\}$ and $d_s = 0$.

Repeat until all nodes are in set *S*.

Find the edge, $p(i, j)$, where *i* is the solved node and *j* is the unsolved node and arc moves from a node already solved to those not yet solved.

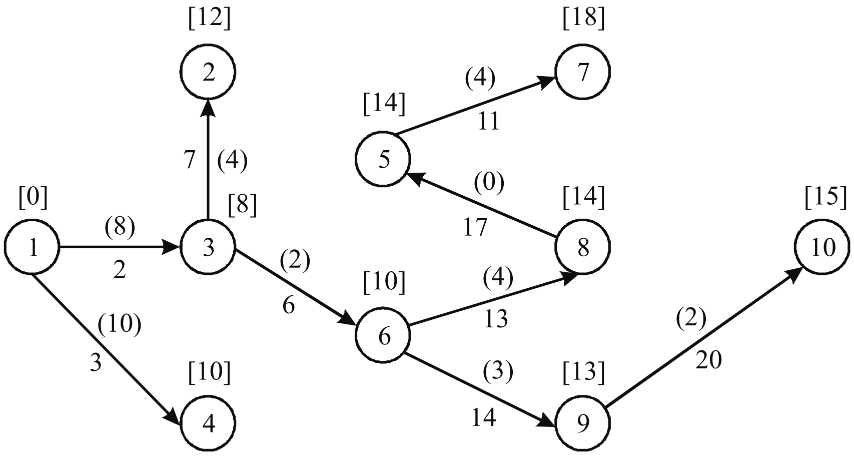
$p(i,j) = \text{arcmin}\{di' + cp' : p'(i', j') \in \boldsymbol{M}, i' \in \boldsymbol{S}, j' \in \boldsymbol{S}^c\}$

Add node *j* and arc *p* to the tree. Add node *j* to the solved set ***S***.

Let dj = di + cp.

In each iteration this algorithm computes the length of path, (which is path-length) from solved node to unsolved nodes. Node having the shortest length is included in the set of solved nodes. After the spanning tree is created, the process terminates.

The spanning tree obtained is shown below:



The numbers in bold, put under brackets indicate the value of length which is associated with nodes . For example shortest path of node 6 is 10. The numbers in the bracket for the nodes in S_c indicates the shortest path length to unsolved nodes passing through the solved nodes in the set *S*. After this the arc is selected with the smallest *dj* value for $i \in \boldsymbol{S}^c$. Hence, choice is made from minimum of 18, 22, 14, 13 and is expressed as min{18, 22, 14, 13}. So, node 9 and arc 14 are included in the spanning tree.

Tabular Presentation of This Algorithm

The algorithm creates a table of seven columns as shown here. Column 1 shows the value of *h* which shows the number of nodes in the set ***S***. Second column lists the members of the set ***S*** which contains solved nodes having minimum one arc connected to a node which is not yet traversed and called unsolved node. Third column shows the closest unsolved node for each node listed in column 2. Column

4 contain every *i,* which is the index of nodes listed in second column, third column lists *j* as the index of the node which is listed in third column. We set *d* as the index of the path or arc oining nodes *i* and *j* and carrying out computation for each case *dj' = di + pk*.

Fifth Column selects the least number from fourth column. Second column has nodes denoted by *i* and *j* is that for node in third column from where, this number was calculated. Fifth column contains node *j* and sixth column lists the length of the shortest path to the node which is added. This is the minimum and is obtained from fourth column. Seventh column 7 has the arc $p(i, j)$ Shortest path tree is formed by adding the node *j* and arc *p* and *j* is added to the set *S*.

h	***Solved Nodes***	***Unsolved Node, Closest to Solved Node***	***Path Length to Unsolved Node***	***Node Added to the Set of Solved Node***	***Shortest Path***	***Arc Added to Tree***
1	1	3	8	3	8	2
2	1	4	10	4	10	3
	3	6	10			
3	1	2	40	6	10	6
	3	6	10			
	4	6	11			
4	1	2	40	2	12	7
	3	2	12			
	6	9	13			
5	2	5	18	9	13	14
	6	9	13			
6	2	5	18	8	14	13
	6	8	14			
	9	10	15			
7	2	5	18	5	14	17
	8	5	14			
	9	10	15			
8	2	7	22	10	15	20
	5	7	18			
	8	10	34			
	9	10	15			
9	2	7	22	7	18	11
	5	7	18			

3.12 WARSHALL'S ALGORITHM

This algorithm is more efficient in determining the access of all pairs of node in a graph whether directed or undirected. A graph *G* with *n* nodes, this method constructs a sequence of *n* adjacency matrices, $P_1,...,P_n$, using the same set of nodes. We start by setting $P_0 = G$.

If P_k is already defined then P_{k+1} has all the edges of P_k, and additional edges, if any, needed to ensure that every pair of nodes joined by an edge of P_k to node $k + 1$ are joined by an arc of P_{k+1} (in the undirected case) and also for every path $a \rightarrow k + 1 \rightarrow b$. The pair (a, b) is an edge of P_{k+1} (in the directed case).

The algorithm terminates after n iterations and P_n contains all adjacency relationships which are shown as edges.

Floyd-Warshall algorithm is an algorithm for graph analysis that finds shortest paths in a graph that is weighted and directed. The algorithm computes the shortest paths between all pairs of vertices. This algorithm is an example of dynamic programming.

We define a path in the matrix k such that path $k[i][j]$ is true if and only if there is a path from node i to node j and there is no node higher than k, except i and j themselves. For any i and j if path $k[i][j]$ = true, this implies that path $(k+1)[i][j]$ is also true. If there is a situation in which path $(k+1)[i][j]$ is true while path $k[i][j]$ is false, is possible if there is a path from i to j via node $k + i$, but no path from i to j via nodes i through k. This means that there is a path from i to $k + 1$ through nodes i through k and a similar path from $k + i$ to j. This follows that the path $(k+ i)[i][j]$ is true if and only if one of the following two conditions holds:

(*i*) path $k[i][j]$ is true.

(*ii*) (path $k[i][k+1]) \wedge$ (path $k[k+1][j]$) is true.

Also, path $0[i][j]$ = adjacent. This is since, direct path is there from node i to node j with no intermediate node. It can also be noted that, path (*MAXNODES*–1)$[i][j]$ = *path*$[i][j]$, because if a path exists through any node, then any path from node i to node j may be selected.

This is Warshall's algorithm which is named, after its discoverer.

This algorithm compares every possible path between every pair of vertices in the graph. It makes only V^3 comparisons. Maximum number of edges may be given as V^2 in the graph with every combination of edges tested. It estimates the shortest path between two vertices, by improving it incrementally to find an optimal solution.

Let there be a graph with a set V, of vertices and each vertex numbered 1 through N. Let there be a function defined as shortest path (i, j, k) which returns the shortest possible path from i to j using only vertices 1, 2, 3, ….., k as intermediate nodes. The objective is to find the shortest possible path from each i to each j using only nodes 1, 2, 3, …., $k + 1$.

There are two alternative paths:

(*i*) Shortest path that only uses nodes in the set (1...k);

or

(*ii*) Another path that goes from i to j, via $k + 1$.

Best path from i to j is one that uses only nodes 1…. k which is defined by the function, shortest path (i,j,k). If there was a better path from i to j via $k + 1$, then the length of this path would be the sum total of the shortest path from i to $k + 1$, traversing vertices 1.....k and the shortest path from $k + 1$ to j using same set of vertices, i.e., 1…....k.

We may define *shortest path* (i,j,k), which is recursive in nature.

This formula is the heart of Floyd Warshall algorithm which works by first computing *shortest path* $(i, j, 1)$ for all (i, j) pairs, then using that to find *shortest path* $(i, j, 2)$ for all (i, j) pairs, and so on and terminates when $k = n$. This finds the shortest path for all (i, j) pairs using any intermediate vertices.

3.13 CUT SET

In graph theory, a cut is a partition of the vertices of a graph into two disjoint subsets. Any cut determines a cut-set, the set of edges that have one endpoint in each subset of the partition. These edges are said to cross the cut. In a connected graph, each cut-set determines a unique cut, and in some cases cuts are identified with their cut-sets rather than with their vertex partitions.

In a flow network, an s–t cut is a cut that requires the source and the sink to be in different subsets, and its cut-set only consists of edges going from the source's side to the sink's side. The capacity of an s–t cut is defined as the sum of the capacity of each edge in the cut-set.

A cut $C = (S, T)$ is a partition of V of a graph $G = (V, E)$ into two subsets S and T. The cut-set of a cut $C = (S, T)$ is the set $\{(u, \upsilon) \in \mathrm{E}, \upsilon \in \mathrm{T}\}$ of edges that have one endpoint in S and the other endpoint in T. If s and t are specified vertices of the graph G, then an s–t cut is a cut in which s belongs to the set S and t belongs to the set T.

In an unweighted undirected graph, the size or weight of a cut is the number of edges crossing the cut. In a weighted graph, the value or weight is defined by the sum of the weights of the edges crossing the cut. A bond is a cut-set that does not have any other cut-set as a proper subset.

Keep repeating the delection process so that all the circuits are 'Broken' and the resultant subgraph is connected and circuit-free which contains all the vertices of *G*.

Subsequently, at the end of the total producure we will obtain a spanning tree.

Hence, it proves that every connected graph has atleast one spanning tree.

Example 3.10: Describe a method of find all spanning tree of a graph.

Solution: Let *G* be a connected graph.

If *G* is a tree then *G* itself will be one and only one spanning tree of *G*.

Now, as shown in the Figure 3.36 consider a connected graph *G*. It is not a tree because it has one circuit. Let T_1 be a spanning tree of *G* that contains the branches *a, b, c, d*.

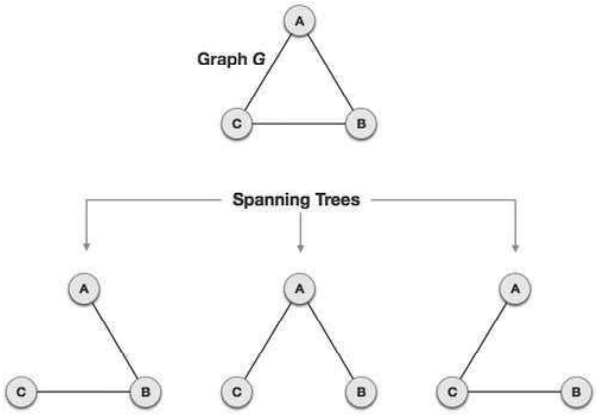


Fig. 3.36 Finding a Spanning Tree

Add a chord, say *h,* to the tree which will form a fundamental circuit through *b, c, h, d*. Removal of the branch *c* of T_1 from the fundamental circuit *b, c, h, d* will break the circuit and will create another spanning tree, say T_2.

Instead of deleting *C*, if we delete *d* or *b* then we will obtain two more different spanning trees, namely *a, b, c, h* and *a, d, h, c*. This process generates all possible trees corresponding to the chord *h and associated fundamental circuit.*

Restarting with the initial tree T_1 and repeating the process of deletion or removal with the chord *h*, using another chord *e* or *f* or *g* you can obtain all possible different spanning trees corresponding to each chord addition to T_1.

Therefore, we can obtain all possible spanning trees of a connected graph.

3.13.1 Fundamental Cut Sets

For defining the concept of Cut Set, let us consider a spanning tree *T* of a connected graph *G*. In graph *G* take any branch *b* in *T.* Subsequently (b) is cut set in *T,* therefore (b) partitions all vertices of *T* into two disjoint sets–one at each end of *b.*

Consdider the same partition of vertices in *G,*and the cut set *S* in *G* that corresponds to this partition. Cut set *s* will contain only one branch *b* of *T*, and the rest (if any) of the edges in *S* will be referred as chords with respect to *T*. This cut-set *S* containing exactly one branch of a tree *T* is termed as **Fundamental Cut Set** with respect to *T.* In addition a fundamental cut set is also termed as **Basic Cut Set.**

In Figure 3.37, a spanning tree *t* (shown with dark lines) and all five of the fundamental cut sets with respect to *T* are shown with broken lines cutting through each cut set.

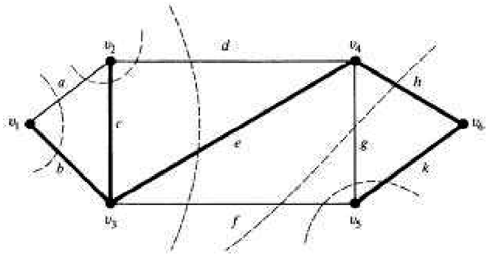


Fig. 3.37 Fundamental Cut Set of a Graph

Every chord of a spanning tree defines a **Unique Fundamental Circuit** while every branch of a spanning tree defines a **Unique Fundamental Cut Set**. Remember that the term fundamental cut set can be defined only with respect to a given spanning tree.

The cut sets of a graph can be obtained from a given set of cut sets.

Theorem 3.12: The ring sum of any two cut sets in a graph is either a third cut set or an edge-disjoint union of cut sets.

In the Figure 3.37 consider that ring sums of the following three pairs of cut sets are given:

$\{d, e, f\} \oplus \{f, g, h\} = \{d, e, g, h\}$, Another cut set.

$\{a, b\} \oplus \{b, c, e, f\} = \{a, c, e, f\}$, Another cut set.

$\{d, e, g, h\} \oplus \{f, g, k\} = \{d, e, f, h, k\}$,

$= \{d, e, f\} \cup \{h, k\}$, An edge-disjoint Union of cut sets

Check Your Progress

7. What is incidence matrix?
8. What is tree?
9. Define planar graph.
10. Where is the concept of shortest path first used?
11. What does Floyd-Warshall algorithm find?
12. Elaborate on the cut-set of a graph.

3.14 ANSWERS TO 'CHECK YOUR PROGRESS'

1. When (u, v) is an edge of the graph G with directed edges, u is said to be adjacent to v and v is said to be adjacent from u. The vertex u is called the initial vertex of (u, v) and v is called the terminal or end vertex of the edge (u, v).
2. A null graph is a totally disconnected graph. A null graph does not have any edge.
3. The degree of a vertex v is the number of edges incident with that vertex. In other words, the degree of a vertex is the number of edges, having that vertex as an end point and is denoted by $d(v)$.
4. A walk is a sequence of vertices and edges starting from any vertex and travelling through edges to a destination vertex, such that no edge appears more than once.
5. A sub-graph H of a graph G is called a component of G, if H is a maximal connected sub-graph of G and component is denoted by $\omega(G)$.
6. A trail that traverses every edge of G is called an Euler trail of G.

7. To any graph G, there corresponds a $V \times E$ matrix called the incidence matrix of G.
8. A connected acyclic graph G is called a tree.
9. A graph G is said to be planar if there exists some geometric representation of G which can be drawn on a plane such that no two of its edges intersect.
10. The concept of 'shortest path first' finds extensive use in network routing protocols, like IS-IS and OSPF.
11. Floyd-Warshall algorithm is an algorithm for graph analysis that finds shortest paths in a graph that is weighted and directed.
12. In graph theory, a cut is a partition of the vertices of a graph into two disjoint subsets. Any cut determines a cut-set, the set of edges that have one endpoint in each subset of the partition. These edges are said to cross the cut. In a connected graph, each cut-set determines a unique cut, and in some cases cuts are identified with their cut-sets rather than with their vertex partitions.

NOTES

3.15 SUMMARY

- A graph G, a triplet $(V(G), E(G), \theta_G)$ consisting of a non-empty set $V(G)$ of vertices, a set of $E(G)$ edges, and a function θ_G assigns to each edge, a subset $\{u, v\}$ of $V(G)$ (u, v needn't be distinct).
- A graph with no self loops and parallel edges is called a simple graph.
- The complement $\overline{G}$ of a graph G is a graph with $V(G) = V(\overline{G})$ and such that uv is an edge of $\overline{G}$ if and only if uv is not an edge of G.
- In a graph with directed edges, the in-degree of a vertex v denoted by $d^-(v)$ is the number of edges with v as their terminal vertex. The out-degree of v denoted by $d^+(v)$ is the number of edges with v as their initial vertex.
- In a directed graph every edge has a direction.
- A simple graph in which each pair of distinct vertices is joined by an edge is called a complete graph.
- Let there be a graph given by $G(V, E)$. If another graph (denoted as $H(V', E')$) is obtained by deleting few vertices and edges then it is the sub-graph of G, if V'in graph H contains all the terminal points of edges in E'.
- The degree of a vertex is the number of edges incident with that vertex.
- A walk is a sequence of vertices and edges starting from any vertex and travelling through edges to a destination vertex such that no edge appears more than once.
- A connected graph might contain more than one spanning tree.
- Let T be the spanning tree of a connected graph G, and e be a chord of G with respect to T. Since the spanning tree T is minimally acyclic, the graph T+e contains a unique cycle. This cycle is called a fundamental cycle in G with respect to T.

- A cycle is a walk. $v_0, v_1,...,v_n$ is a walk in which $n \geq 3,\ v_0 = v_n$ and the 'n'-vertices $v_1, v_2,\ldots,v_n$ are distinct.
- Let u and v be vertices in a graph G. We say that u is connected to v if G contains a $(u - v)$ path.
- A directed graph is strongly connected if there is a path from u to v and v to u, whenever u and v are vertices in the graph.
- A trail that traverses every edge of G is called an Euler trail of G. A circuit (tour) of G is a closed walk that traverses each edge of G at least once. An Euler tour is a tour which traverses each edge exactly once. A graph is Eulerian if it contains an Euler tour.
- Graphs can be represented using adjacency and incidence matrices.
- A graph G which has no cycles is called an acyclic graph. A connected acyclic graph G is called a tree.
- Binary search tree is a binary tree in which each child is either a left or right child; no vertex has more than one left child and one right child, and the data are associated with vertices.
- A systematic method for visiting every vertex of an ordered rooted tree is called as a 'Traversal algorithm'.
- Let G be a simple connected graph. A spanning tree of G is a sub-graph of G, i.e., a tree containing every vertex of G.
- A graph G is said to be planar if there exists some geometric representation of G which can be drawn on a plane such that no two of its edges intersect ('meeting' of edges at a vertex is not considered an intersection).
- To show a graph G is nonplanar we have to prove that of all possible geometric representations of G, none can be embedded in a plane.
- If G, a connected planar graph has n vertices, e edges and r regions, then, $n - e + r = 2$.
- Dijkstra's algorithm is an algorithm for graph search solving the single-source shortest path problem for a weighted graph having non-negative edge path costs, producing a tree that gives the shortest path.
- Warshall's algorithm is an efficient algorithm in determining the access of all pairs of node in a graph whether directed or undirected.

3.16 KEY TERMS

- **Simple graph:** A graph with no self loops and parallel edges is called a simple graph.
- **Multigraph:** MultigraphA graph which has loops and parallel edges is a multigraph.
- **Pseudograph:** A graph with self loops and parallel edges is called a pseudograph.

- **Complete graph:** A simple graph in which each pair of distinct vertices is joined by an edge is called a complete graph.
- **Isolated vertex:** A vertex with degree zero is called an isolated vertex.
- **Pendant vertex:** A vertex with degree one is called a pendant vertex.
- **Adjacent vertices:** A pair of vertices that determine an edge are called adjacent vertices.

3.17 SELF-ASSESSMENT QUESTIONS AND EXERCISES

Short-Answer Questions

1. What is a graph?
2. Name the different types of graphs.
3. Define degree of a vertex.
4. Define a path.
5. What is the difference between connected and disconnected graphs?
6. Define Eulerian graph.
7. What are adjacency matrices?
8. What are rooted trees?
9. State Kuratowski's theorem.
10. State the basic idea behind Dijkstra's algorithm.
11. What is the complexity of Floyd-Warshall algorithm?

Long-Answer Questions

1. Discuss briefly about graphs.
2. Explain different types of graphs.
3. Illustrate incidence and degree of graphs.
4. Explain fundamental circuit and cyclic interchange with examples.
5. Discuss the properties on graphs with the help of diagrams.
6. Explain the concept of Eulerian digraphs.
7. Illustrate matrix representation of graphs.
8. Describe briefly the concept of traversal of a Tree.
9. State and prove Euler's formula.
10. Write the pseudocode of Dijkstra's algorithm.
11. Illustrate Floyd-Warshall algorithm.

3.18 FURTHER READING

NOTES

Iyengar, N Ch S N. V M Chandrasekaran, K A Venkatesh and P S Arunachalam. *Discrete Mathematics*. New Delhi: Vikas Publishing House Pvt. Ltd., 2007.

Tremblay, Jean Paul and R. Manohar. *Discrete Mathematical Structures with Applications to Computer Science*. New York: McGraw-Hill Inc., 1975.

Deo, Narsingh. *Graph Theory with Applications to Engineering and Computer Science*. New Delhi: Prentice-Hall of India, 1999.

Singh, Y.N. *Mathematical Foundation of Computer Science*. New Delhi: New Age International Pvt. Ltd., 2005.

Malik, D.S. *Discrete Mathematical Structures: Theory and Applications*. London: Thomson Learning, 2004.

Haggard, Gary, John Schlipf and Sue Whiteside. *Discrete Mathematics for Computer Science*. California: Thomson Learning, 2006.

Cohen, Daniel I.A. *Introduction to Computer Theory*, 2nd edition. New Jersey: John Wiley and Sons, 1996.

Hopcroft, J.E., Rajeev Motwani and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*, 3rd edition. Boston: Addison-Wesley, 2006.

Linz, Peter. *An Introduction to Formal Languages and Automata*, 5th edition. Boston: Jones and Bartlett Publishers, 2011.

Mano, M. Morris. *Digital Logic and Computer Design*. New Jersey: Prentice-Hall, 1979.

UNIT 4 INTRODUCTORY COMPUTABILITY THEORY

NOTES

Structure

4.0 INTRODUCTION

In this unit we will learn about the introductory computability theory. Computability theory, also called recursion theory, is a branch of mathematical logic that originated in the 1930s with the study of computable functions and Turing degrees. A regular language is a formal language that can be expressed using a regular expression. Finite automata are computing devices that accept/recognize regular languages and are used to model operations of many systems. Their operations can be simulated by a very simple computer program. A Moore machine is a finite-state machine, whose output values are determined solely by its current state. A Mealy machine is a finite-state machine whose output values are determined both by its current state and the current inputs.

4.1 OBJECTIVES

After going through this unit, you will be able to:

- Understand finite state machines and their transition table diagrams
- Explain regular languages
- Discuss equivalence of DFA and NFA
- Describe the concept of reduced machines
- Illustrate the concept of Moore and Mealy machines

4.2 FINITE STATE MACHINES AND THEIR TRANSITION TABLE DIAGRAMS

Finite automata are mathematical models of machines which are used for accepting languages over the input alphabets. In finite automata, the number of input symbols and the number of states are finite.

The components of finite automata are input tape, finite control and reading head (Refer Figure 4.1).

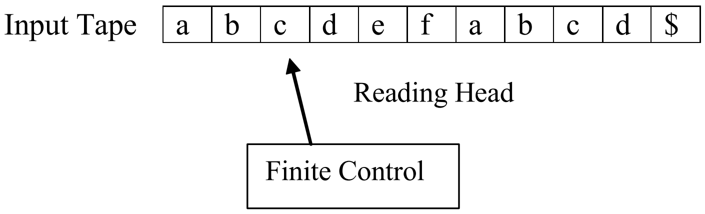


Fig. 4.1 *Block Diagram of Finite Automata*

1. **Input Tape:** It consists of input symbols with $ at the end to indicate the end of the input tape. If $ is not present it means that it is an infinite input string. The input symbols are processed from left to right in sequence.
2. **Reading Head:** It reads data from the input tape.
3. **Finite control:** The control moves from one state to another state depending on the current input symbol.

DFA

Deterministic Finite Automata (DFA) refers to the system where on reading each input symbol the system may have a transition from the current state to only one state or it may remain in the same state itself.

A deterministic finite automata or DFA is defined by 5-tuples (Q, Σ, δ, q_0, F). Here,

1. Q is a finite non-empty set of states.
2. Σ is a finite non-empty set of input symbols.
3. δ is a transition function which maps Q × Σ into Q. Transition functions represent the change of state during the transition for a given input symbol.
4. q_0 represents the initial state such that $q_0 \varepsilon Q$.
5. F represents the final state and it is otherwise known as accepting state.

DFA can be described using the following two notations:

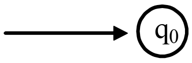
1. Transition diagram or transition graph
2. Transition table

Transition Diagram or Transition Graph

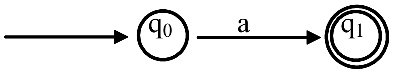
A transition diagram is a finite directed labelled graph in which each vertex (node) represents a state and the directed edges indicate the transition of a state; the edges are labelled with input/output. The initial state is represented by an arrow pointing towards the state. The final or accepting state is represented by double circles.

By means of an input symbol the system moves from one state to another state.

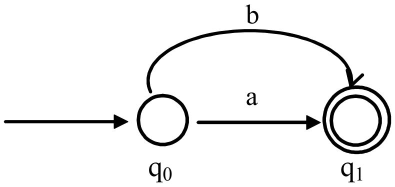
1. If q_0 is the initial state, then it is represented as



2. If q_1 is the final state, then it is represented as



3. If there is a transition from one state to another state by means of two input symbols, then the transition diagram is given by

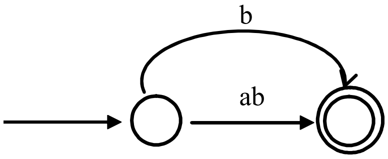


The transitions $\delta(q_0, a) = q_1$ and $\delta(q_0, b) = q_1$.

Generalized Transition Graph

Generalized transition graphs are transition graphs where the labels of the edges are regular expressions but not strings. The labels of the edges may be ε.

A generalized transition graph is a finite directed labelled graph in which each vertex (node) represents a state and the directed edges indicate the transition of a state; the edges are labelled with input. The labels may be input symbols, regular expressions or ε. The initial state is represented by an arrow pointing towards the state. The final or accepting state is represented by double circles.



Transition Table

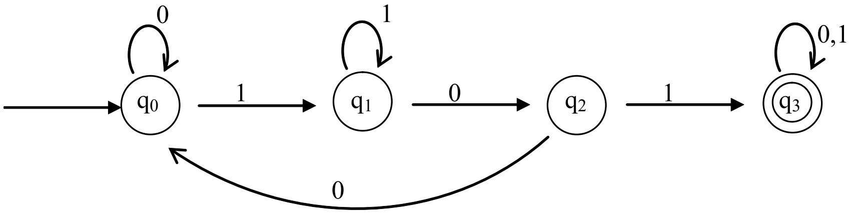
A transition table is a tabular representation of a transition. It has two arguments, namely states and inputs. The rows of the table correspond to states and the columns correspond to input symbols.

In the transition table, the start state is marked with an arrow and the accepting state is marked with a circle or star.

Example 4.1: Design a DFA for the language that consists of 101 as a substring over the alphabet $\Sigma = \{0, 1\}$ and draw the transition table.

NOTES

Solution:



States	0	1
→q_0	q_0	q_1
q_1	q_2	q_1
q_2	q_0	q_3
q_3	q_3	q_3

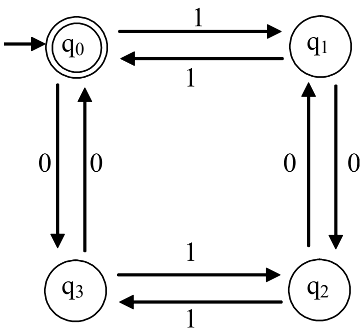
Example 4.2: Design a DFA that has even number of 0s and even number of 1s.

Solution: L = {w | w has even number of 0s and even number of 1s}.

Transition Table

State	0	1
→q_0	Even 0	Even 1
q_1	Even 0	Odd1
q_2	Odd 0	Even 1
q_3	Odd 0	Odd1

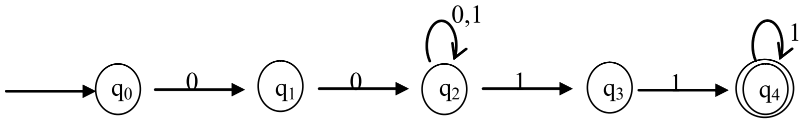
For example, if the input string is 10101010



Suppose if the input string is 10101010, then initially we consider (q_0, 1). By default we have assigned q_0 as even 0, even 1. If 1 is the input, then q_0 becomes even 0, odd 1 which is q_1. Again, q_1 = even 0, odd 1 and by receiving the input 0 it becomes odd 0, odd 1. Similarly, the process continues until the input string is completed.

Example 4.3: Design a finite automata that accepts all possible strings that start with 00 and end with 11 over the alphabet S = {0, 1}.

Solution:



Acceptance of Strings by Finite Automata

A string x is accepted by a finite automata M = (Q, Σ, δ, q_0, F) if δ (q_0, x) = q for some q ε F.

Example 4.4: Consider the finite state automaton whose transition table is given below, where Q ={q_0, q_1, q_2, q_3}, Σ = {0,1} and F = q_0. Check whether or not the input string 110101 is accepted by the finite automata.

States	0	1
$\rightarrow q_0$	q_2	q_1
q_1	q_3	q_0
q_2	q_0	q_3
q_3	q_1	q_2

Solution:

$$\begin{aligned}
\delta (q_0, 110101) &= \delta [\delta (q_0, 1), 10101] \\
&= \delta (q_1, 10101) \\
&= \delta [\delta (q_1, 1), 0101] \\
&= \delta (q_0, 0101) \\
&= \delta [\delta (q_0, 0), 101] \\
&= \delta (q_2, 101) \\
&= \delta (\delta (q_2, 1), 01) \\
&= \delta (q_3, 01) \\
&= \delta (\delta (q_3, 0), 1) \\
&= \delta (q_1, 1) \\
&= q_0
\end{aligned}$$

It has reached the final state. Hence, the given input string is accepted.

Properties of Transition Function

1. In a finite automaton δ (q, ε) = q, i.e., the state of the transition system is changed only by means of an input symbol.
2. For all strings w and input symbols a

$$\delta (q, aw) = \delta (\delta (q, a), w)$$

or

$$\delta (q, wa) = \delta (\delta (q, w), a)$$

Example 4.5: Prove that if δ (q, x) = δ (q, y) then show that δ(q, xz) = δ (q, yz) for all strings.

Solution:

$$\begin{aligned}
\text{LHS:}\quad \delta(q, xz) &= \delta(\delta(q, x), z) && \text{By Property 2}\\
&= \delta(\delta(q, y), z) && \text{Given}\\
&= \delta(q, yz)
\end{aligned}$$

RHS:

$$\begin{aligned}
\delta(q, yz) &= \delta(\delta(q, y), z) && \text{By Property 2}\\
&= \delta(\delta(q, x), z) && \text{Given}\\
&= \delta(q, xz)
\end{aligned}$$

Language of DFA

The language of DFA is denoted by L(A) and it is defined by

$L(A) = \{w \mid \delta(\hat{q}_0, w) \text{ is in } \Sigma\}$

In other words, the language of A is defined as the set of strings w that assumes the start state as q_0 and reaches one of the final states. If L is L(A) for some DFA, A, then we say that L is a regular language.

NFA

A Non-deterministic Finite Automata (NFA) is defined by 5-tuples $(Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite non-empty set of states.
2. Σ is a finite non-empty set of inputs.
3. δ is the transition function mapping from $Q \times \Sigma$ into P(Q), the power set of Q.
4. q_0 is the initial state.
5. F is the set of final states.

Basic Difference between NFA and DFA

1. In NFA by means of the same input symbol, the system can change its state from one state to more than one state, e.g., $\delta(q_0, a) = \{q_1, q_2\}$. It can move to either q_1 or q_2.
2. A state can move to the next state without having any input symbol (i.e., by means of an empty string).

   $\delta(\hat{q}_0,) = q_1$ or $\delta(q_0, \varepsilon) = q_1$
3. In NFA, the transition $\delta(q_0, a)$ may be empty, i.e., no transition for this particular state.

Extended Transition Function

In non-deterministic finite automata the transition function $\hat{\delta}$ is extended to the transition function δ, such that it maps $Q \times \Sigma$ into 2^Q resulting in

$\hat{\delta}(q, \varepsilon) = \{q\}$

Language of Non-Deterministic Finite Automata

A language L accepted by an NFA is defined as

$L(A) = \{w \mid \hat{\delta}\ (q_0, w)) \cap F \neq \varphi\}$

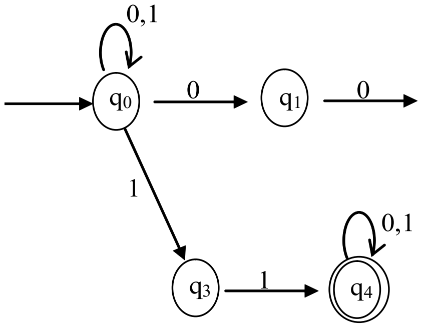
In other words, the language accepted by the NFA is the set of strings such that

$L(A) = \{w\ \varepsilon\ \Sigma^* : \hat{\delta}\ (q_0, w) \cap F = \text{at least one final state}\}$

A string $w\ \varepsilon\ \Sigma^*$ is accepted by the NFA if it contains at least one final state.

Acceptance of Strings by Non-Deterministic Finite Automata

Example 4.6: Consider the following NFA:



Check the acceptance of 01001.

Solution:

Transition Table

States	0	1
→ q_0	$\{ q_0, q_1 \}$	$\{ q_0 , q_3 \}$
q_1	$\{ q_2 \}$	-
$\textcircled{q_2}$	$\{ q_2 \}$	$\{ q_2 \}$
q_3	-	$\{ q_4 \}$
$\textcircled{q_4}$	$\{ q_4 \}$	$\{ q_4 \}$

Let the input string be 01001.

$$\begin{aligned}
\delta\ (q_0, 0) &= \delta(q_0, 0) = (q_0, q_1)\\
\hat{\delta}\ (q_0, 01) &= \delta[\delta\ (q_0, 0), 1]\\
&= \delta[(q_0, q_1), 1]\\
&= \delta(q_0, 1), \cup\ \hat{\delta}\ (q_1, 1)\\
&= (q_0, q_3) \cup \phi\\
&= (q_0, q_3)\\
\hat{\delta}\ (q_0, 010) &= \delta(\delta\ (q_0, 01), 0)\\
&= \delta[(q_0, q_3), 0]\\
&= \delta[(q_0, 0) \cup \delta\ (q_3, 0)]
\end{aligned}$$

$$= \quad (q_0, q_1) \cup \phi = (q_0, q_1)$$

$$\hat{\delta}\ (q_0, 0100) \quad = \quad \delta\ (\delta\ (q_0, 010), 0)$$

$$= \quad \delta\ [(q_0, q_1), 0]$$

$$= \quad \delta\ [(q_0, 0) \cup\ \delta\ (q_1, 0)]$$

$$= \quad (q_0, q_1) \cup (q_2)$$

$$= \quad (q_0, q_1, q_2)$$

$$\hat{\delta}\ (q_0, 01001) \quad = \quad [\delta\ (q_0, 0100), 1]$$

$$= \quad \delta\ [(q_0, q_1, q_2), 1]$$

$$= \quad \delta\ [(q_0, 1) \cup \delta\ (q_1, 1) \cup \delta\ (q_2, 1)]$$

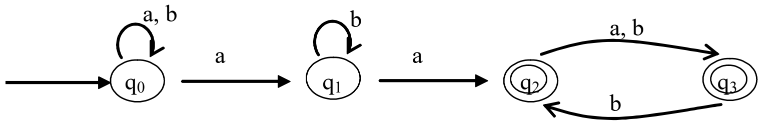
$$= \quad (q_0, q_2, q_3)$$

Since it contains one of the final states q_2, the given input string is accepted.

Conversion of NFA to DFA

For every NFA, there exists an equivalent DFA.

Example 4.7: Convert the following NFA into DFA.



Solution:

Step 1:

Start from the initial state q_0 for every input symbol. If we get a set of states as the output then consider them as a new state.

Let $q_0 = A$

$$\delta(q_0, a) \quad = \quad \{q_0, q_1\} = B$$

$$\delta(q_0, b) \quad = \quad \{q_0\} = A$$

Step 2:

Now we consider $\{q_0, q_1\}$ as a single state

$$\delta\ (\{q_0, q_1\}, a) \quad = \quad \delta\ (q_0, a) \cup \delta\ (q_1, a)$$

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

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

$$\delta\ (\{q_0, q_1\}, b) = \quad \delta\ (q_0, b) \cup \delta\ (q_1, b)$$

$$= \quad \{q_0\} \cup \{q_1\}$$

$$= \quad \{q_0, q_1\} = B$$

Step 3:

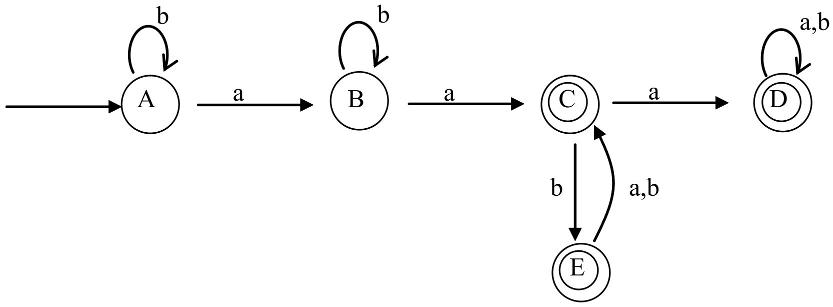
$$\delta\ (\{q_0, q_1, q_2\}, a) \quad = \delta\ (q_0, a) \cup \delta\ (q_1, a) \cup \delta\ (q_2, a)$$

$$= \{q_0, q_1, q_2, q_3\} = D$$

$$\begin{aligned}
\delta(\{q_0, q_1, q_2\}, b) &= \delta(q_0, b) \cup \delta(q_1, b) \cup \delta(q_2, b)\\
&= \{q_0, q_1, q_3\} = E\\
\delta(\{q_0, q_1, q_2, q_3\}, a) &= \delta(q_0, a) \cup \delta(q_1, a) \cup \delta(q_2, a) \cup \delta(q_3, a)\\
&= \{q_0, q_1, q_2, q_3\} = D\\
\delta(\{q_0, q_1, q_2, q_3\}, b) &= \delta(q_0, b) \cup \delta(q_1, b) \cup \delta(q_2, b) \cup \delta(q_3, b)\\
&= \{q_0, q_1, q_2, q_3\} = D\\
\delta(\{ q_0, q_1, q_3\}, a) &= \{ q_0, q_1, q_2\} = C\\
\delta(\{ q_0, q_1, q_3\}, b) &= \{ q_0, q_1, q_2\} = C
\end{aligned}$$

States	**a**	**b**
→ A	B	A
B	C	B
C	D	E
D	D	D
E	C	C

Transition Diagram



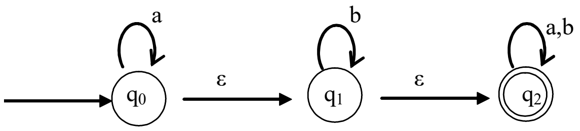
NFA with ε-Moves

In non-deterministic finite automata there is a transition from one state to another state by means of zero, one or more transitions.

NFA also makes a transition from one state to another state without receiving any input symbol.

The transitions without any input symbols are called NFA with ε-transitions.

Consider the following NFA:



This is an example of NFA with ε-transitions. It starts from the initial state q_0 and reaches the final state q_2 by means of ε-transitions.

Non-Deterministic Finite Automata with ε-Transitions

The non-deterministic finite automata with ε-transitions is denoted by 5-tuples, A = { Q, Σ, δ, q_0, F }

Here, all the elements are same as non-deterministic finite automata, except the input alphabet Σ.

Here, Σ = Σ ∪ { ∈ }, i.e., the input alphabet includes ∈, i.e., empty string also.

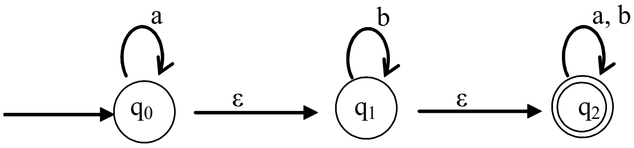
Acceptance of a String by NFA with ε-Moves

A string x in Σ* is accepted by the NFA, if there exists at least one path that corresponds to x, that starts from the initial state. But the path is formed by means of ε-transitions as well as non ε-transitions. To find whether the given string x is accepted or not by the NFA, we define a function, ε-closure (q), where q is the state of automata.

The function ε-closure (q) is defined as follows:

ε-closure (q) is a set of all those states that can be reached from q on a path labelled by ε.

Example 4.8: Consider the following NFA and find its ∈-closure.



Solution:

ε-Closure (q_0) = {q_0, q_1, q_2}

ε-Closure (q_1) = {q_1, q_2}

ε-Closure (q_2) = {q_2}

Conversion of NFA with ε-Transitions into DFA without ε-Transitions

The main idea behind NFA to DFA conversion is that each DFA state corresponds to a set of NFA states. The DFA uses its state to keep track of all possible states, the NFA can be in, after reading each input symbol.

ε-Closure(s): Set of NFA states reachable from NFA states on ε-transitions alone.

4.3 REGULAR LANGUAGES

In theoretical computer science, a regular grammar is a formal grammar that describes a regular language. A regular grammar is a left or right regular grammar. Right regular grammar is also known as right linear grammar and left regular grammar are also known as left linear grammar.

Right Linear and Left Linear Grammar

Right Linear Grammar

A grammar G is said to be right linear if each production has one variable at the left side and the right side consists of zero or more number of terminals followed by an optional single variable or $\in$.

$A \rightarrow aB, A \rightarrow a$

These are the examples of right linear grammar.

Left Linear Grammar

A grammar G is said to be left linear if each production consists of a single variable at the left side and the right side consists of an optional single variable followed by any number of terminals.

$A \rightarrow Ba, A \rightarrow a$

These are the examples of left linear grammar.

A regular grammar is one that is either right linear or left linear.

A linear grammar is a grammar in which at most one variable can occur on the right side of any production, but the position of the variable in not restricted.

Example 4.9: Consider the grammar G = {{S, A, B}, {a, b}, S, P}, where P is given as

$S \rightarrow A$

$A \rightarrow bA / \in$

$B \rightarrow Ba$

Check whether the grammar is regular or not.

Solution: The productions for A, B are right linear and left linear but the production for S is neither left linear nor right linear.

Therefore, it is not regular. But it is a linear grammar.

Notes:

1. Regular and linear grammars are context free.
2. Context-free grammars are not always linear.

Sentential Forms

If G = (V, T, P, S) is a CFG, then any string in $(V \cup T)^*$ such that $S \Rightarrow \alpha$ is a sentential form.

If $\alpha \in L(G)$ then

$S \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots\ldots\ldots.. \Rightarrow \alpha_n \Rightarrow \alpha$ is the derivation of α. The strings $\alpha_1, \alpha_2, ..., \alpha_n$ may contain variables as well as terminals. These are called sentential forms of the derivation.

If $S \overset{*}{\Rightarrow} \alpha$ and if the variable is replaced from the left side, then it is called left sentential form.

If $S \overset{*}{\Rightarrow} \alpha$ and if the variable is replaced from the right side, then it is called right sentential form.

Two grammars G_1 and G_2 are said to be equivalent if they generate the same language, i.e.,

$L(G_1) = L(G_2)$

4.4 EQUIVALENCE OF DFA AND NFA

An NFA is easily constructed than a DFA. Every language, described by some NFA is also described by some DFA. The smallest DFA can have 2^n states while the smallest NFA for the same language has only *n* states.

'Subset construction' involves constructing all subsets of the set of states of the NFA, which are eventually DFA.

The subset construction starts from the NFA and is given as,

$$N = (Q_N, \Sigma, \delta_N, q_0, F_N)$$

The goal is to make DFA, $D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$

so that $L(D) = L(N)$

The input alphabets of the two automata are the same and the start state of D is the set having only the start state of D.

Other components of D are constructed as follows:

1. Q_D is the power set of Q_N. Q_N has n states and hence, Q_D will have 2^n states. All are accessible states and can be 'thrown away' so effectively that the number of states of D may be much smaller than 2^n.
2. F_D is the power set of Q_N such that,

   $S \cap F_N \neq \Phi$

   F_D is set of N states that have at least one accepting state of N.
3. For each set $S \subset Q_N$ and for each input symbol in Σ:

   $\delta_D(S, a) = p$ in $S \subset \delta_N(p, a)$

To compute $\delta_D(S, a)$, you can look at all states of p in S (N goes to form p on input a) and take the union of all those states.

	0	1
Φ	Φ	Φ
$\{ q_0 \}$	$\{ q_0, q_1 \}$	$\{ q_0 \}$
$\{ q_1 \}$	Φ	Φ
$\{ q_2 \}$	Φ	Φ
. . .	. . .	. . .
$\{ q_0, q_1 \}$	$\{ q_0, q_1 \}$	$\{ q_0, q_2 \}$
$*\{ q_0, q_2 \}$	$\{ q_0, q_1 \}$	$\{ q_0 \}$
$*\{ q_1, q_2 \}$	Φ	$\{ q_2 \}$
$*\{ q_0, q_1, q_2 \}$	$\{ q_0, q_1 \}$	$\{ q_0, q_2 \}$

4.5 REDUCED MACHINES

Minimization refers to constructing an automata with minimum number of states to a given automata M. In this we delete the states of the automata that do not affect the language accepted by the automata.

Equivalence of relations:

1. Two states q_1 and q_2 are equivalent if both $\delta(q_1, x)$ and $\delta(q_2, x)$ are final states or both of them are non-final states for all $x \in \Sigma^*$.
2. Two states q_1 and q_2 are k-equivalent $(k \geq 0)$ if both $\delta(q_1, x)$ and $\delta(q_2, x)$ are final states or both are non-final states for all strings x of length $\leq$ k.
3. Any two final or any two non-final states are also 0-equivalent.

Construction of Minimum Automata

Step 1: Construction of Π_k. By definition of 0-equivalence, $\Pi_0 = \{Q_1^0, Q_2^0\}$, where Q_1^0 is the set of all final states and $Q_2^0 = Q - Q_1^0$.

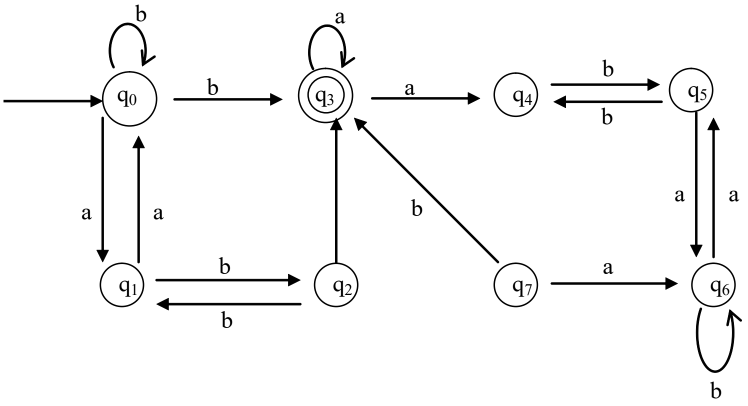
Step 2: Construction of Π_{K+1} from Π_K. Let Q_i^k be any subset in Π_k. If q_1 and q_2 are in q_1^k, they are (k +1) equivalent, provided $\delta(q_1, a)$ and $\delta(q_2, a)$ are k-equivalent for every $a \in \Sigma$. Repeat this procedure for every Q_1^k in Π_k to get Π_{k+1}.

Step 3: Construct $\Pi_n = \Pi_{n+1}$ for n = 1, 2, ……

Step 4: Now for the required minimum state automata, the states are the equivalence classes obtained in Step 3.

The transition table is obtained by replacing a state q by the corresponding equivalence class q.

Example 4.10: Construct the minimum state automata equivalent to the given transition diagram.



Solution:

Transition Table

States Σ	0	1
→ q_0	q_1	q_0
q_1	q_0	q_2
q_2	q_3	q_1
q_3	q_3	q_0
q_4	q_3	q_5
q_5	q_6	q_4
q_6	q_5	q_6
q_7	q_6	q_3

NOTES

Since there is only one final state

$$Q_1^0 = \{q_3\}$$

$$Q_2^0 = Q - Q_1^0 = \{q_0, q_1, q_2, q_4, q_5, q_6, q_7\}$$

$\therefore$ $\Pi_0 = \{\{q_3\}, \{q_0, q_1, q_2, q_4, q_5, q_6, q_7\}\}$

Now q_0 is 1 equivalent to q_1, q_5, q_6 but not to q_2, q_4, q_7.

$\therefore$ $Q_3^0 = \{q_0, q_1, q_5, q_6\}$

q_2 is 1,which is equivalent to q_4.

$\therefore$ $Q_3^1 = \{q_2, q_4\}$

$\therefore$ $Q_4^1 = \{q_7\}$

Therefore,

$$\Pi_1 = \{\{q_3\}, \{q_0, q_1, q_5, q_6\}, \{q_2, q_4\}, \{q_7\}\}$$

Now

q_0 is 2, which is equivalent to q_6 but not to q_1, q_5.

$\therefore$ $Q_2^2 = \{q_0, q_6\}$

$\therefore$ $Q_3^2 = \{q_1, q_5\}$

Again q_2 is 2, which is equivalent to q_4.

$$Q_3^2 = \{q_2, q_4\}$$

$$Q_5^2 = \{q_7\}$$

$$\Pi_2 = \{\{q_3\}, \{q_0, q_6\}, \{q_1, q_5\}, \{q_2, q_4\}, \{q_7\}\}$$

As q_0 is 3, which is equivalent to q_6

$$Q_2^3 = \{q_0, q_6\}$$

q_1 is 3, which is equivalent to q_5

$$Q_3^3 = \{q_1, q_5\}$$

q_2 is 3, which is equivalent to q_4

$$Q_4^3 = \{q_2, q_4\}$$

$$Q_5^3 = \{q_7\}$$

$$\Pi_3 = \{ \{q_3\}, \{q_0, q_6\}, \{q_1, q_5\}, \{ q_2, q_4\}, \{q_7\}\}$$

The minimum state automata is given as

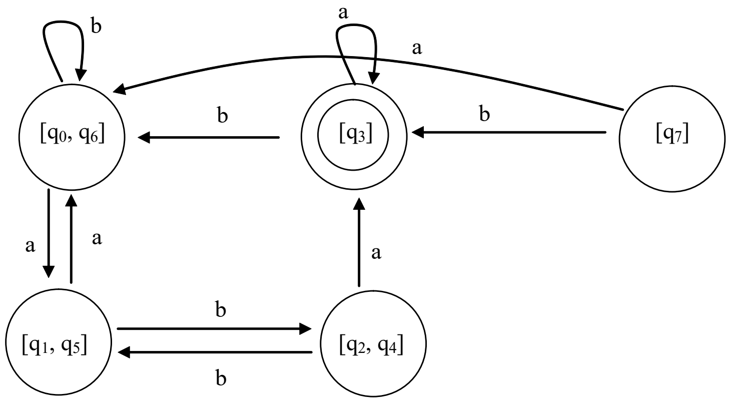
$$M = (Q_1, \{a, b\}, \delta, q_0^1, F^1)$$

Here, $Q^1 = \{\{q_3\}, \{q_0, q_6\}, \{q_1, q_5\}, \{q_2, q_4\}, \{q_7\}\}$

$$Q_0^1 = \{q_0, q_6\} \; F^1 = \{q_3\}$$

Transition Table

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



4.6 MOORE AND MEALY MACHINES

Two equivalent theoretical machines with output are discussed here.

Moore Machine

The value of the output function z (t) depends only on the present state and it is independent of the current input.

The output function may be written as

$z(t) = \lambda q(t)$

Moore machine is defined by a 6-tuple M = (Q, Σ, Δ, δ, λ, q_0), where

Q – Non-empty finite set of states

Σ – Non-empty finite set of input symbols

Δ – Non-empty finite set of outputs

δ – Transition function that maps Σ × Q into Q

λ – Output function that maps Q into Δ

q_0 – Initial state

Representation of Moore Machine

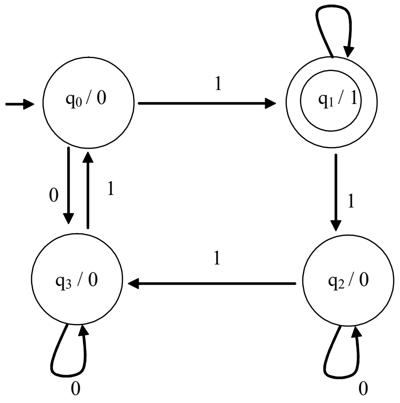
Moore machine is represented by a transition table and transition diagram.

Let M be a Moore machine.

Transition Table

Present State	Next State		Output
	a = 0	a = 1	
$\rightarrow q_0$	q_3	q_1	0
q_1	q_1	q_2	1
q_2	q_2	q_3	0
q_3	q_3	q_0	0

Transition Diagram



In the transition diagram, state is represented by a circle. Inside the circle the state is separated by a slash and the output of the present state is marked after the state. The left side symbol which is present represents the state and the right side is the output from that state.

Acceptance of Strings by Moore Machine

Let the input string be 0111.

When the input string is processed, the transition of states is given by:

$$\overset{0}{q_0} \xrightarrow{1} \overset{0}{q_3} \xrightarrow{1} \overset{0}{q_0} \xrightarrow{0} \overset{1}{q_1} \xrightarrow{1} \overset{0}{q_2}$$

The corresponding output string is 00010.

In Moore machine we always start with the input string. For the input string, the output is given as:

$$\lambda (q_0) = 0$$

In the Moore machine the first symbol printed is the character (output) specified in the start state.

If the input string is of length n then the output string is of length n + 1.

For example, if the input string has seven letters then the output string will have eight letters, because it has eight states in its path. In Moore machine, the process terminates whenever the last input symbol is read and the last output character is printed.

NOTES

Mealy Machine

The value of the output function z (t) is a function of the present state q (t) and present input x (t).

$$z(t) \;=\; \lambda(q(t), x(t))$$

A Mealy machine is a 6-tuple,

$M = (Q, \Sigma, \Delta, \delta, \lambda, q_0)$, where all the symbols except λ have the same meaning as that in Moore machine.

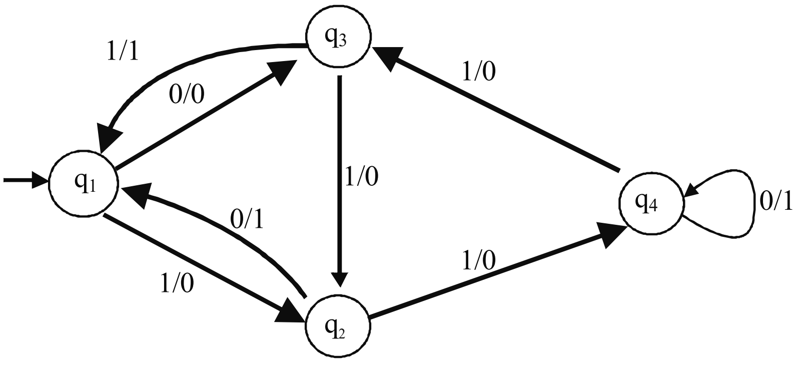
In Mealy machine, λ is the output function that maps $\Sigma \times Q$ into Δ.

Representation of Mealy Machine

The Mealy machine can be represented by a transition table and transition diagram.

Present State	**Next State**			
	a = 0		a = 1	
	State	Output	State	Output
$\rightarrow q_1$	q_3	0	q_2	0
q_2	q_1	1	q_4	0
q_3	q_2	1	q_1	1
q_4	q_4	1	q_3	0

Transition Diagram



The edges are labelled by a/b, where a represents the input symbol and b is the output character.

Acceptance of Strings by Mealy Machine

Let the input string be 0111.

When the string is processed by Mealy machine, the transition is as follows:

$$q_1 \xrightarrow[0]{0} q_3 \xrightarrow[1]{1} q_1 \xrightarrow[0]{1} q_2 \xrightarrow[0]{1} q_4$$

Output string is 0100.

In the case of Mealy machine we get an output string only on the application of an input symbol.

Here, if the input string is of length n then the output string is also of the same length n.

Equivalence of Moore and Mealy Machines

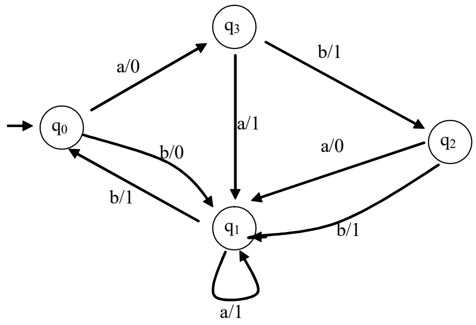
In general, two machines are said to be equivalent if they accept the same language. Similarly, two automata are said to be equivalent if they produce the same output with the same input.

In this sense, we cannot compare a Moore machine with a Mealy machine. Two Moore machines may be equivalent, two Mealy machines may be equivalent, but a Moore machine can never be directly equivalent to a Mealy machine. This is because the length of the output string is different in both the cases for the same input string. The length of the output string in a Moore machine is one greater than that in the Mealy machine, because a Moore machine always starts with one automatic start symbol.

Definition

Given the Mealy machine M_e and Moore machine M_0 which prints the automatic start state character *x*, we say that these two machines are equivalent if for every input string the output string from M_0 is exactly *x* concatenated with the output from M_e.

Example 4.11: Draw a transition table for the Mealy machine for the given transition diagram.



Solution:

Transition Table

Present State	**Next State**			
	Input a = a		Input a = b	
	State	Output	State	Output
→ q_0	q_3	0	q_1	0
q_1	q_1	1	q_0	1
q_2	q_1	0	q_1	1
q_3	q_1	1	q_2	1

Conversion of Moore Machine into Mealy Machine

Procedure:

1. First we define the output function λ^1 for Mealy machine as a function of the present state and input symbol.

We define λ^1 by

$\lambda^1(q, a) = \lambda(\delta(q, a))$ for all states q and input symbol a.

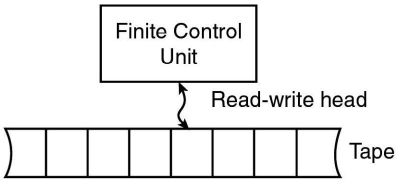
2. The transition function is the same as that of given Moore machine.

Conversion of Mealy Machine into Moore Machine

1. We first look into the next state column for any state, say q_i and determine the number of different outputs associated with q_i in that column.
2. Split the q_i into several different states, the number of such states being equal to the number of different outputs associated with q_i.

4.7 TURING MACHINE

It can be visualized as a single, one dimensional array of cells, each of which can hold a single symbol. This array extends indefinitely in both directions and is therefore capable of holding an unlimited amount of information. This information can be read and changed in any order. It has a storage device called Tape which is quite analogous to the magnetic tapes used in actual computers. Read-Write head is associated with the tape that can travel right or left on the tape and that can read and write a single symbol on each move. The automaton that we use as a Turing machine will have neither an input file nor any output file. Whatever input and output is necessary will be done on the machine's tape.



A turing machine M is defined by $M = \{Q, \Sigma, \rho, \delta, q_0, B, F\}$

Where Q is the set of internal states,

Σ is the input alphabet,

ρ is a finite set of symbols called the tape alphabet,

δ is the transition function,

$B \in \rho$ is a special symbol called blank, $q_0 \in Q$ is the initial state, and

$F \leq Q$ is the set of final states.

Assumption We assume $\Sigma \subseteq \rho - \{B\}$; that is, the input alphabet is a subset of the tape alphabet, not including the blank.

Note: The transition function δ is defined as $\delta : Q \times \rho \to Q \times \rho \times \{L, R\}$

Where δ is a partial function on $Q \times \rho$. Its interpretation gives the principle by which a Turing machine operates. The arguments of δ are the current state of the control unit and the current tape symbol being read. Then the result is a new state of the control unit, a new tape symbol, which replaces the old one, and a move symbol L or R. The move symbol indicates whether the read-write head moves left or right, one cell after the new symbol has been written on the tape.

For Example,

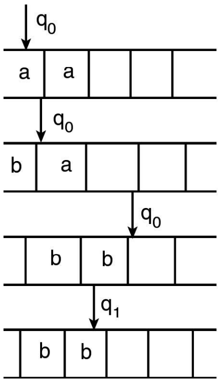
NOTES

$$M = \{Q, \Sigma, \rho, q_0, F, B\}$$

where $Q = \{q_0, q_1\}, \Sigma = \{a, b\}, \rho = \{a, b, B\}$ and $F = \{q_1\}$

and

$$\delta(q_0, a) = (q_0, b, R)$$
$$\delta(q_0, b) = (q_0, b, R)$$
$$\delta(q_0, B) = (q_0, B, L)$$



We can think of a Turing machine as a simple computer. It has a processing unit, which has a finite memory, and the tape, is a secondary storage of unlimited capacity. Here δ defines how this computer acts and is often called the "Program" of the machine.

We know that the automaton always start in the given initial state with some information on the tape. It then goes through a sequence of steps controlled by the transition function δ. During this process, the contents of any cell on the tape may be examined and changed many times. Eventually, the whole process may terminate. We say this situation as the turing machine is brought into halting state. A Turing machine is said to be at halt whenever it reaches a configuration for which δ is not defined; this is possible because δ is a partial function.

In fact, we will assume that no transitions are defined for any final state. So the Turing machine will halt whenever it enters a final state.

Note: If the read-write head moves alternatively between right and left, but makes no modifications to the tape, this is an instance of the Turing machine not at halt. As an analogy with programming terminology, we say that the Turing machine is in an infinite loop.

Instantaneous Description of a Turing Machine

An instantaneous description (ID) is defined as $\alpha_1 q \alpha_2$ where α_1 is the string processed already, α_2 is the string to be processed when the turing machine is in the state q.

Language accepted by a Turing Machine Let M be a Turing machine. Then the language accepted by M denoted by $T(M) = \{w \in \Sigma^+ : q_0 w \vdash^* \alpha_1 q_f \alpha_2$ for some $q_f \in F$ and $\alpha_1, \alpha_2 \in \rho^*\}$

Example 4.12: Let $\Sigma = \{0,1\}$. Design a Turing machine that accepts the language denoted by the regular expression 00^*

Solution

$$\delta(q_0, 0) = (q_{0,} 0, R)$$
$$\delta(q_0, B) = (q_0, B, R)$$

Starting at the left end of the input, we read each symbol and check that it is a 0. If it is, we continue by moving right. If we reach a blank without encountering it, we continue by moving right. If we reach blank without encountering any thing but 0, we terminate and accept that string. If the input contains 1 anywhere, the string is not in $L(00^*)$ and we halt in a non final state. To keep track of the computation, two internal states $Q = (q_{0,} q_1)$ and one final state $F = (q_1)$ are sufficient. As long as *0* appears under the read-write head, the head will move to the right. If at any time a 1 is read, the machine will halt in the non final state q_0, since $\delta(q_0, 1)$ is undefined.

Example 4.13: Design a turing machine that accepts $L = \{a^n b^n; n \geq 1\}$

Solution

Let $\Sigma\{a, b\}$, $Q\{q_0, q_1, q_2, q_3, q_4\}$, $F = \{q_4\}$ and $P = \{a, b, x, y, B\}$

The transitors can be broken into several parts. The set

$$\partial(q_0, a) = (q, x, R)$$
$$\partial(q_1, a) = (q_1, a, R)$$
$$\partial(q_1, y) = (q_1, y, R)$$
$$\partial(q_1, b) = (q_2, y, L)$$

replaces the leftmost a with an *x*, then canses the read - write head to travel right to the first *b*, replacing it with a *y* when *y* is written, the machine enters a state q_2 indicating that *a* has been successfully poined with *ab*.

The next set of transition reverses the direction until an *x* is encountered, repositions the read-write head over the leftmost *a*, and returns to the initial state.

$$\partial(q_2, y) = (q_2, y, L)$$
$$\partial(q_2, a) = (q_2, a, L)$$
$$\partial(q_2, x) = (q_0, x, R)$$

This gives the initial state q_0, ready to deal with next a and *b*.

Computation after first pass

$$q_0 aa...abb...b \overset{*}{\vdash} xq_0 a...ayb...b$$

After second pass

$$q_0 aa...abb...b \overset{*}{\vdash} xxq_0...ayy...b$$

To detect input where an a follows *ab*

$$\partial(q_0, y) = (q_3, y, R)$$
$$\partial(q_3, y) = (q_3, y, R)$$
$$\partial(q_3, B) = (q_4, B, R)$$

The particular input *aabb* gives the following successive instantaneous descriptions.

$$q_0\, aabb \vdash x\, q_1 aabb \vdash xaq_1 bb$$
$$\vdash x q_2\, ayb$$

NOTES

$\vdash q_2\, x\, ayb$

$\vdash x\, q_0\, ayb$

$\vdash xx\, q_1\, yb$

$\vdash xx\, yq_1 b \vdash xxq_2\, yy$

$\vdash x\, q_2\, xyy$

$\vdash xx\, q_0\, yy$

$\vdash xxy\, q_3\, y$

$\vdash xxyy\, q_3\, B$

$\vdash xxyy\, Bq_4\, B$

At this point the turing machine halts in a final state, so the string *aabb* is accepted.

Example 4.14: Let *x* and *y* be two positive integers represented in Mary notation. Construct a Turing machine that halts in a final state q_y if $x \geq y$ and that will halt in non final state q_n if $x < y$.

i.e. $q_0 w(x) O\, w(y) \overset{*}{\vdash} q_y w(x) O\, w(y) \text{ if } x \geq y$

$q_0 w(x) O\, w(y) \overset{*}{\vdash} q_n w(x) O\, w(y) \text{ if } x < y$

Solution To solve this problem in the previous example instead of matching *a*'s, we match each 1 on the left of the dividing 0 with 1 on the right. At the end of matching, we will have on the tape either

$xx...110xxx...xB$ or $xx...xx0xx...x11B$,

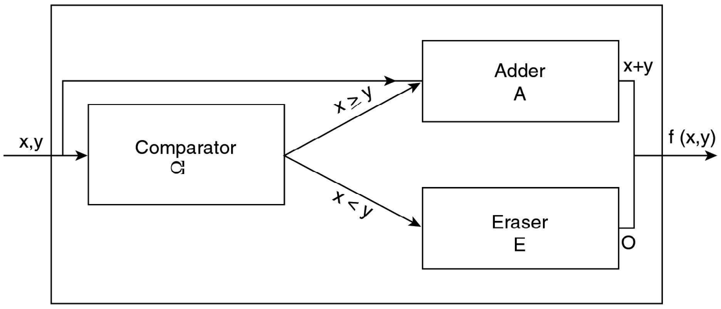
depending an whether $x > y$ or $y > x$. In the first case, when we attempt to match another 1, we encounter the blank at the right of the working space. This can be used as a signal to enter the state q_y.

We still find 1 on the right when all 1's on the left have been replaced. We use this to get into the other state q_n (case two).

Example 4.15: Desingn a turing machine that computes the function

$$f(x, y) = x + y \text{ if } x \geq y$$
$$= 0 \text{ if } x < y$$

Solution



The diagram shows that first we use a comparing machine to determine whether or not $x \geq y$. If so, the comparator sends a start signal to the adder, which then computes $x+y$. If not, an erasing program is started that changes every 1 to blank. For the eraser E, we construct a Turing machine having states indexed with E. The computations to be done by C are

$$q_{c,o}\, w(x)O\, w(y) \overset{*}{\vdash\!\!-\!\!-} q_{A,o}\, w(x)\, O\, w(y) \text{ if } x \geq y$$

And

$$q_{c,o}\, w(x)O\, w(y) \overset{*}{\vdash\!\!-\!\!-} q_{E,o}\, w(x)\, O\, w(y) \text{ if } x < y$$

If we take $q_{A,o}$ and $q_{E,o}$ as the initial states of A and E respectively, we see that C starts either A or E.

Computations performed by the adder A

$$q_{A,o}\, w(x)\, Ow(y) \overset{*}{\vdash\!\!-\!\!-} q_{A,f}\, w(x+y)O,$$

Computations performed by Eraser E

$$q_{E,o}\, w(x)\, Ow(y) \overset{*}{\vdash\!\!-\!\!-} q_{E,f}\, O.$$

Check Your Progress

1. What is the use of finite automata?
2. Define right linear grammar.
3. How many states the smallest DFA can have?
4. What do you mean by minimization?
5. What are the two equivalent theoretical machines?
6. Elaborate on the Turing machine.

4.8 ANSWERS TO ‘CHECK YOUR PROGRESS’

1. Finite automata are used for accepting languages over the input alphabets.
2. A grammar G is said to be right linear if each production has one variable at the left side and the right side consists of zero or more number of terminals followed by an optional single variable or ε.
3. The smallest DFA can have 2^n states.
4. Minimization refers to constructing an automata with minimum number of states to a given automata M.
5. Moore and Mealy are the two equivalent theoretical machines.
6. The automaton that we use as a Turing machine will have neither an input file nor any output file. Whatever input and output is necessary will be done on the machine’s tape.

4.9 SUMMARY

NOTES

- Finite automata are mathematical models of machines which are used for accepting languages over the input alphabets.
- In finite automata, the number of input symbols and the number of states are finite.
- A generalized transition graph is a finite directed labelled graph in which each vertex (node) represents a state and the directed edges indicate the transition of a state; the edges are labelled with input.
- A transition table is a tabular representation of a transition.
- NFA also makes a transition from one state to another state without receiving any input symbol.
- The main idea behind NFA to DFA conversion is that each DFA state corresponds to a set of NFA states.
- A regular grammar is a formal grammar that describes a regular language.
- A regular grammar is a left or right regular grammar.
- Right regular grammar is also known as right linear grammar and left regular grammar are also known as left linear grammar.
- A regular grammar is one that is either right linear or left linear.
- A linear grammar is a grammar in which at most one variable can occur on the right side of any production, but the position of the variable in not restricted.
- The smallest DFA can have 2^n states while the smallest NFA for the same language has only n states.
- Minimization refers to constructing an automata with minimum number of states to a given automata
- In Moore Machine, the value of the output function z (t) depends only on the present state and it is independent of the current input.
- In Moore machine we always start with the input string.
- If the input string is of length n then the output string is of length n + 1.
- In Mealy Machine, the value of the output function z(t) is a function of the present state q(t) and present input x(t).
- In general, two machines are said to be equivalent if they accept the same language.
- Two automata are said to be equivalent if they produce the same output with the same input.

4.10 KEY TERMS

- **Input tape:** It consists of input symbols with $ at the end to indicate the end of the input tape.

- **Reading head:** It reads data from the input tape.
- **Finite control:** The control moves from one state to another state depending on the current input symbol.
- **Generalized transition graph:** Generalized transition graphs are transition graphs where the labels of the edges are regular expressions but not strings.
- **Transition table:** A transition table is a tabular representation of a transition.

NOTES

4.11 SELF-ASSESSMENT QUESTIONS AND EXERCISES

Short-Answer Questions

1. What are the components of finite automata?
2. Define regular grammar.
3. What is equivalence of DFA and NFA?
4. State equivalence of relations in reduced machines.
5. How is Moore machine represented?

Long-Answer Questions

1. Explain the concept of finite automata machines with the help of transition table diagrams.
2. Make a note on regular languages.
3. Discuss the construction components of DFA.
4. Explain briefly the steps to construct minimum automata.
5. Illustrate Moore and Mealy machines with the help of transition tables.
6. Analyse the Turing machine giving examples.

4.12 FURTHER READING

Iyengar, N Ch S N. V M Chandrasekaran, K A Venkatesh and P S Arunachalam. *Discrete Mathematics*. New Delhi: Vikas Publishing House Pvt. Ltd., 2007.

Tremblay, Jean Paul and R. Manohar. *Discrete Mathematical Structures with Applications to Computer Science*. New York: McGraw-Hill Inc., 1975.

Deo, Narsingh. *Graph Theory with Applications to Engineering and Computer Science*. New Delhi: Prentice-Hall of India, 1999.

Singh, Y.N. *Mathematical Foundation of Computer Science*. New Delhi: New Age International Pvt. Ltd., 2005.

Malik, D.S. *Discrete Mathematical Structures: Theory and Applications*. London: Thomson Learning, 2004.

Haggard, Gary, John Schlipf and Sue Whiteside. *Discrete Mathematics for Computer Science*. California: Thomson Learning, 2006.

NOTES

Cohen, Daniel I.A. *Introduction to Computer Theory*, 2nd edition. New Jersey: John Wiley and Sons, 1996.

Hopcroft, J.E., Rajeev Motwani and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*, 3rd edition. Boston: Addison-Wesley, 2006.

Linz, Peter. *An Introduction to Formal Languages and Automata*, 5th edition. Boston: Jones and Bartlett Publishers, 2011.

Mano, M. Morris. *Digital Logic and Computer Design*. New Jersey: Prentice-Hall, 1979.

UNIT 5 GRAMMAR AND LANGUAGES

NOTES

Structure

5.0 INTRODUCTION

In this unit, you will learn about the concept of languages and grammar. Languages accepted by finite automata are called regular expression. A language that can be defined by regular expression is called a regular language. Any set represented by a regular expression is called a regular set. Grammars are used to provide the structure of files. A regular grammar is a formal grammar that is used to describe a regular language. You will also learn about the classification of grammar by Noam Chomsky. He classified grammar into four types, namely Type 0 grammar, Type 1 grammar, Type 2 grammar and Type 3 grammar. Chomsky hierarchy is the relationship among languages. A context-free language is a language generated by some context-free grammar. We can represent the context-free languages or the derivations by means of parse trees. You will also learn normal forms of context-free grammars and pumping lemma. Pumping lemma is used to prove that the given language is not context-free. The decidability properties of context-free languages check the emptiness of the string, decide whether the language is infinite or not and also check whether the given string is present in the language or not. Derivation of trees, sentential forms and notations of syntax analysis are also explained in this unit.

5.1 OBJECTIVES

After going through this unit, you will be able to:

- Know the concept of languages and grammar

- Illustrate phrase structure grammars
- Explain regular expressions, languages and grammars
- Understand context free grammars and derivation trees
- Discuss sentential forms
- Give notations of syntax analysis

5.2 CONCEPT OF LANGUAGES AND GRAMMAR

A language processor is a type of software that bridges a specification or execution gap in software programming. The software design expresses the ideas in terms related to the application domain of the software. These ideas are then implemented in the execution domain.

However, a semantic gap may result in poor quality of software. A Programming Language (PL) facilitates in overcoming quality issues in the software.

The programming language comprises the following:

- Specification, design and coding
- Implementation steps

Software implementation using PL introduces PL domain between the application domain and the execution domain.

- Specification gap is between application domain and PL domain
- Execution gap is between PL domain and execution domain

A language processor is a type of software that bridges a specification or execution gap. It takes input from the source program, checks it and makes it error-free so that it can be a target program. The target program is executed to give the desired performance.

Components of Language Processors

1. A language processor bridges an execution gap of the machine language of a computer system. An assembler is a language translator whose source language is assembly language. A compiler is any language translator that is not an assembler.
2. A processor bridges the same direction gap as the language translator but in the reverse direction.
3. A preprocessor is a language processor that bridges an execution gap but is not a language translator.
4. A language migratory bridges the specification gap between two PLs.

For example,

1. A language processor that converts C++ program into C is also called preprocessor.
2. Language translator for C++ produces a machine language program. The target programs are the C program and the machine language program (Refer Figures 5.1 a and b).

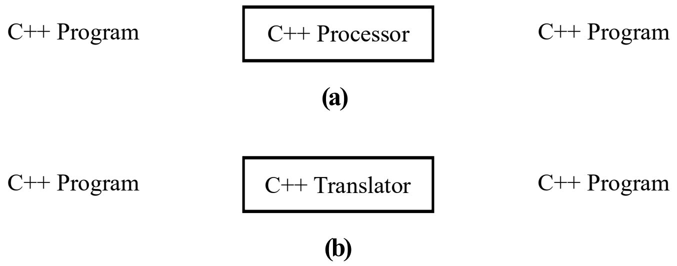


Fig. 5.1 *A Language Processor and Translator*

Interpreter

An interpreter is a program that executes a code to display the desired result to a user without generating a machine language program. The interpreter executes the programming language domain code to the execution domain (Refer Figure 5.2).

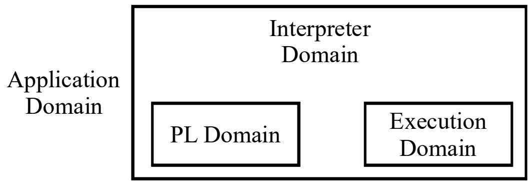


Fig. 5.2 *An Interpreter*

Language Processing Activities

Language processing activities involve two steps:

(i) Program generation

(ii) Program execution

Program Generation

A program generation activity comprises changing a source code into a target program that can be executed to give a desired result. The program generator can either be converted to an executable code (maybe a machine code or bytecode) by a compiler, or an interpreter can directly use it for execution.

For example,

1. In C and C++, the compiler generates a machine code from the source program.
2. In Java, the source program is compiled to target a program that is known as a bytecode. It is portable and can be taken into another machine for execution.
3. Languages like PERL, LISP and PROLOG— used for writing the source code—are immediately generated for execution by the compiler that also works for the interpreter.

You can say that the compiler compiles code that is further executed by the interpreter.

In LISP, PERL and PROLOG the interpreter does both compilation and interpretation at the same time. Thus, it compiles and interprets codes. These are also called interpreter languages (Refer Figure 5.3).

NOTES

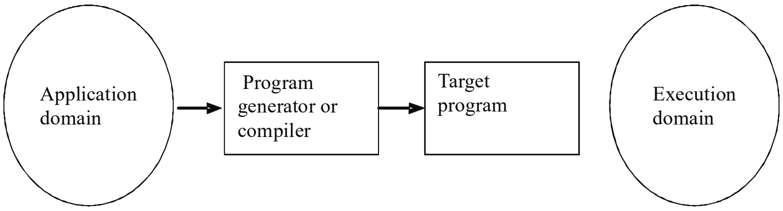


Fig. 5.3 *Interpreter Domain*

Program Execution

Program execution is done in two steps:

1. Translation
2. Interpreter

1. *Translation*

- A program must be translated before it can be executed.
- The translated program may be saved in a file. The saved program may be executed repeatedly.
- A program must be retranslated following any modification (Refer Figure 5.4).

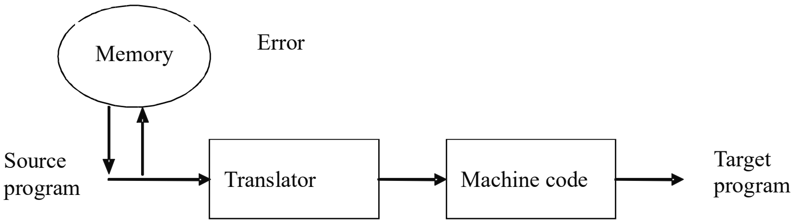


Fig. 5.4 *Translation Model*

2. *Interpreter*

An interpreter reads the source program and stores it in its memory. There are two steps for interpretation:

(a) Instruction cycle
(b) Interpretation cycle

(a) Instruction Cycle

The Central Processing Unit (CPU) here uses a Program Counter (PC) to note the address of the next instruction to be executed. This instruction is subject to the instruction execution cycle that consists of the following steps:

a. Fetching the instruction
b. Decoding the instruction to determine the operations to be performed and its operands
c. Executing the instruction

The instruction address in the PC is updated and the cycle is repeated for the next instruction. Program interpretation can proceed in an analogous manner. The PC can indicate which statement of the source program is to be interpreted next.

(b) Interpretation Cycle

The interpretation cycle consists of the following steps (Refer Figure 5.5):

a. Fetching the statement
b. Analysing the statement and its meaning
c. Executing the statement

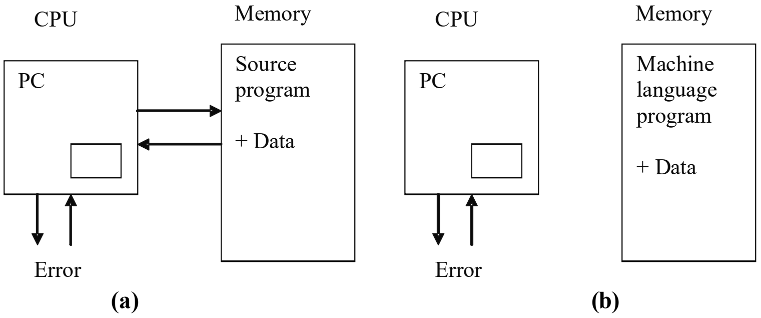


Fig. 5.5 Steps in Interpretation

Phases and Passes of a Language Processor

A program is first analysed and then synthesized by a language processor (Refer Figure 5.6). A language processor consists of two phases.

(i) Analysis phase
(ii) Synthesis phase

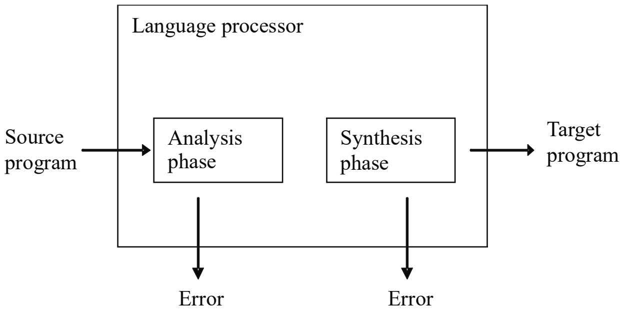


Fig. 5.6 A Language Processor

Forward References

A forward reference of a program entity is a reference to the entity that precedes its definition in the program.

Consider the program.

```
Interest = (principal* rate * time)/100;
Long Interest;
.
.
.
```

`Interest` has double precision value, but it comes later in the statement of the source code. Thus, reference to `Interest` in assignment statement comes later in a program. So, it is not possible to generate correct code statement by statement. Here, you will need multiple pass model of language processing.

Language Processor Pass

A language processor pass is the processing of every statement in a source or its equivalent representation to perform a language processing function.

Pass 1: It performs analysis of the program and notes relevant information.

Pass 2: It performs synthesis of the target program.

Information collecting type about `Interest` is noted in Pass 1. This information is used during Pass 2 to perform code generation.

Intermediate Representation

An Intermediate Representation (IR) is a representation of a source program that reflects the effect of some, but not all, analysis and synthesis tasks performed during language processing.

Take the following example. The first pass performs analysis of the source program and reflects its results in the intermediate representation. The second pass reads and analyses the IR, instead of the source program, to perform synthesis of the target program. This avoids repeated processing of the source program. The first pass is called front-end as it works with the source language and the second pass is called back-end as it produces the target program (Refer Figures 5.8 and 5.9).

The front-end and back-end language processors do not necessarily co-exist in the memory. This reduces the memory requirement (Refer Figure 5.7).

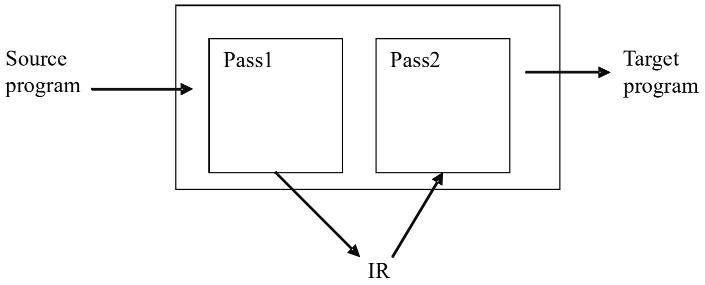


Fig. 5.7 *Language Passes*

Compiler

The front-end language processor performs the following functions:

- Lexical analysis—scanning of lines of code
- Syntax analysis—parsing of codes
- Semantic analysis—meaning of codes

The output of the front-end language processor is (Refer Figure 5.8).

- Table of information—constant table, symbol table, etc.
- Intermediate information—intermediate code and intermediate representation

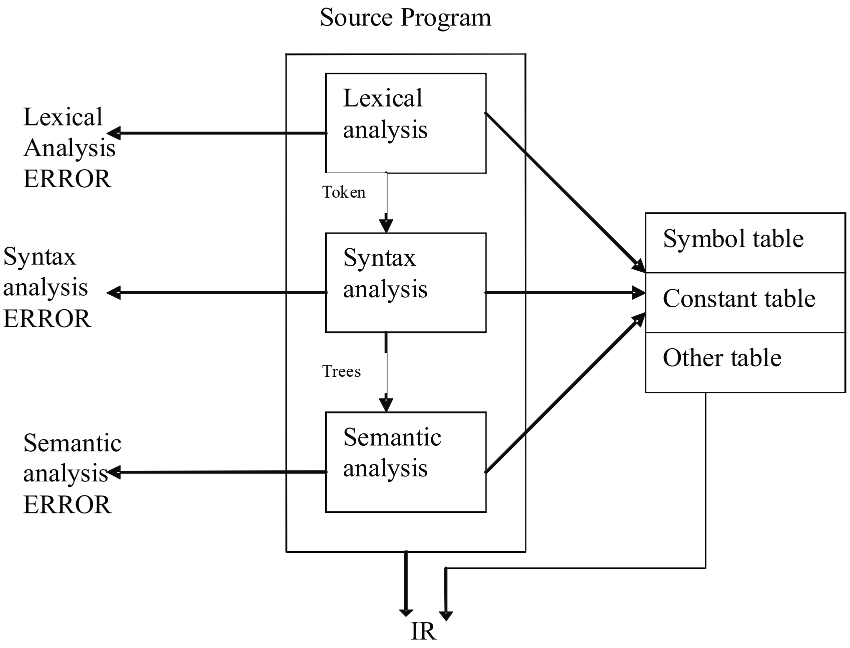


Fig. 5.8 *Front-End of a Compiler*

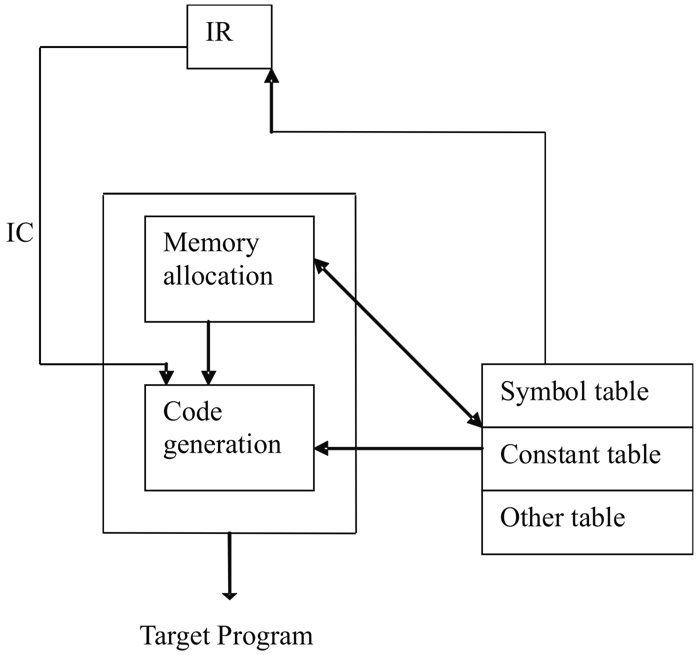


Fig. 5.9 *Back-End of a Compiler*

Fundamentals of Language Processing Specification

Programming Language Grammar

(i) Terminal Symbols, Alphabets and Strings

Alphabet: An alphabet is a finite non-empty set of symbols. Common alphabets are:

(a) Σ ={0,1}

(b) Σ ={a, b, c,…,z}

(c) The set of all ASCII characters

Terminal symbols: The symbol in the alphabet is known as a terminal symbol T.

Metasymbols: Symbols like |, (,), →, : are part of notation. These are called metasymbols.

NOTES

Non-terminal symbols: A Non-Terminal (NT) is the name of a syntax category of a language, e.g., noun, verb, etc. An NT is written as a single capital letter or as a name enclosed between <…>, e.g., <Noun> represents a noun.

Strings: A string is a finite sequence of symbols chosen from some alphabet, for example, 0101 is a string from binary alphabet $\Sigma = \{0,1\}$.

Empty string: The empty string is the string with zero occurrences of symbols. This string denoted by $\in$ is a string that may be chosen from any alphabet.

(ii) Grammar

(a) **Type 0 Grammar:** This grammar, known as phase structure grammar, contains production of the form

$\alpha \to \beta$

where α and β can be strings of Ts and NTs. Such productions permit arbitrary substitution of strings during derivation or reduction. Therefore, these are relevant to specification of programming languages.

(b) **Type 1 Grammar:** These grammars are called context sensitive grammars as their production, derivation or reduction of strings can take place only in a specific context. Type 1 production has the form

$\beta \quad \alpha A\beta \to \alpha\pi\beta$

Thus, a string π in a sentential form can be replaced by 'A' only when it is closed by the string α and β.

(c) **Type 2 Grammar:** These impose no context requirements on derivation or reduction. A typical Type 2 production is of the form

$A \to \pi$

which can be applied independent of its context. These grammars are, therefore, known as Context-Free Grammar (CFGs). CFGs are ideally suited for programming language specification ALGOL60. PASCAL specification uses Type 2 grammar.

(d) **Type 3 Grammar:** These are characterized by production of the form

$A \to tB \mid t$

$A \to Bt \mid t$

This grammar satisfies the requirement of Type 2 grammar. The use of Type 3 production is restricted to specification of lexical units, identifiers, constants, labels, etc.

The production for <constant>, <identifier> are Type 3

<id> → l |<id> l | <id > d

Where l and d stand for letter and digit respectively. This is also called linear grammar or regular grammar.

NOTES

(e) **Operator Grammar:** An Operator Grammar (OG) is a grammar none of whose production contains two or more consecutive NTs in any RHS. Thus, nonterminals occurring in a RHS string are separated by one or more terminal symbols. All the terminal symbols occurring in the RHS strings are called operators of the grammar. The symbols *, +, (,) and ↑ are operator grammar.

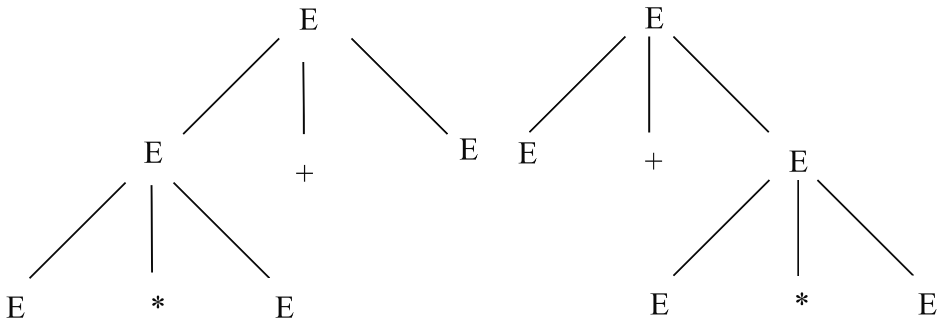
(iii) Ambiguity of Grammar

A grammar that produces more than one parse tree for some sentence is said to be ambiguous. An ambiguous grammar produces more than one leftmost or more than one rightmost derivation for the same sentence.

Consider the example of the expression

$$E + E * E$$

Now, there are two parse trees for this expression.



Now, this derivation shows the ambiguity of the grammar. Thus, G = (V, T, P, S) is ambiguous if there is at least one string w in T* for which you can find two different parse trees each with root labelled s and yield w. If each string has just one parse tree in the grammar then the grammar is unambiguous.

(iv) Binding and Binding Time

(a) Binding

Each program has many entities, such as interest, principal, rate and time, etc. Now, each of the entities have many attributes—variable, procedure or reserve identity (keyword). A variable attribute includes type, demotionality, scope and memory address. Thus, the attribute of one program entity may be another program entity like type is an attribute of a variable. It is also a program entity with its own attribute size (number of memory bytes). The values of the attributes of a type should be determined as follows:

A Int;

B String;

C Long;

Here, C is binding with Long datatype.

Binding is the association of an attribute of a program entity with a value.

(b) Binding Time

Binding time is the time at which a binding is performed. Thus, the type attribute of variable C to Int in declaration is processed.

Binding times are:

- Language definition time of T_L
- Language implementation of T_L
- Compilation time of a program P is T_P
- Execution init time of procedure T_{PROC}
- Execution time of procedure PROC is T_{PROC}
- L is a language.
- P is a program written in L.
- PROC is procedure in program P.

Language implementation time is when a language translator is designed. The preceding list of binding times is not exhaustive. Other binding times can be defined, such as binding at the linking time of P, etc. The binding time of an attribute of a program entity determines the manner in which a language processor can handle the use of the entity. A compiler can generate code specifically tailored to a binding performed during or before compilation time. However, compilers cannot generate such code for binding performed later than compilation time. This affects execution efficiency of the target program.

(c) Static Binding

A static binding is a binding performed before the execution of program begins.

(d) Dynamic Binding

A dynamic binding is a binding performed after the execution has begun.

Automation is based on the mathematical concept of computation which includes symbols, alphabets and strings.

Length of a String

Length of a string is defined as the total number of symbols or number of positions of symbols present in the string.

If w is the string, the length of the string is denoted by | w |.

For example, (i) If w = 00111, | w | = 5.

(ii) If w = aab, | w | = 3.

(iii) If the string is empty ε, then | ε | = 0.

Powers of an Alphabet

If $\sum$ is an alphabet, the set of all strings of a certain length from that alphabet can be expressed using an exponential notation. We define $\sum^k$ to be the set of strings of length k, each of whose symbols is in $\sum$.

For example, $\Sigma^0 = \{\varepsilon\}$, whatever be Σ.

If $\Sigma = \{ 0,1 \}$, then $\Sigma^1 =\{ 0,1 \}$, $\Sigma^2 =\{ 00,11,01,10 \}$,

$\Sigma^3 = \{ 000,001,010,100,101,110,111\}$

Kleen's Closure and Positive Closure

The Kleen's Closure denotes the set of all strings including the empty string $\in$ over the alphabet Σ. Kleen's Closure is otherwise known as Star Closure and it is denoted by Σ^*.

If $\Sigma = \{ 0,1\}$, then $\Sigma^* = \{ \varepsilon , 0, 1, 01, 10, 11, 101, 000,\ldots..\}$.

In other words, $\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \ldots\ldots$

Positive Closure:

The set of non-empty strings from alphabet Σ is denoted by Σ^+.

$\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \ldots\ldots$

In other words,

$\Sigma^+ = \Sigma^* - \{ \varepsilon \}$

Concatenation of Strings

Let x, y be any two strings of alphabet Σ. The concatenation of two strings is a new string obtained by combining the two strings x and y.

Let $x = x_1x_2\ldots..x_m$, $y = y_1y_2\ldots y_n$, then $xy = x_1x_2\ldots..x_m\, y_1y_2\ldots y_{n.}$

Length of the string is given by

$| xy | = | x | + | y |$

Reverse of a String

The reverse of a string is obtained by writing symbols in the reverse order. If s is a string then s^R is the reverse of s. For example, if s = xyz, then s^R = zyx.

Substring

Let s be a string. Any string of consecutive characters from s is known as substring of s. If s = abcdef, then bcd is the substring of s.

Prefix and Suffix of a Substring

A prefix of a string is a substring of leading symbols of that string. If w = vy for some y, then v is called the prefix of w.

In other words, v is a prefix of w if there exists $y \,\varepsilon\, \Sigma^*$ such that w = vy.

A suffix of a string is a substring of trailing symbols of that string. If w = xv for some x, then v is called the suffix of w.

In other words, v is a suffix of w if there exists $x \,\varepsilon\, \Sigma^*$ such that w = xv.

Palindrome

A palindrome is a string that can be read the same way in both forward and backward directions, e.g., madam.

Languages

A set of strings all of which are chosen from some $\sum^*$, where $\sum$ is an alphabet is called a language. If $\sum$ is an alphabet and L $\varepsilon\sum^*$ then L is a language over $\sum$. There are some languages that appear in automata.

1. The languages of all strings consisting of n 0s followed by n 1s for some, $n \geq 0$: { ε, 01, 0011, 000111, …..}
2. Set of strings over 0s and 1s with equal number of 0s and 1s { ε , 01,0011,10,1100, …..}
3. Set of binary numbers whose value is a prime { 10, 11, 101,111,1011, …}
4. $\sum^*$ is a language for any alphabet $\sum$.
5. φ, the empty language is a language over any alphabet.
6. {ε}, the language consisting of only the empty string.

If we start with an alphabet having only one letter, the letter x

$$\sum = \{x\}$$

Then, we can define a language by saying that any non-empty string of alphabet characters is $L_1 = \{ x \;\; xx \;\; xxx \;\; xxxx……… \}$

In other words,

$L_1 = \{ x^n \text{ for } n = 1, 2, 3, ……. \}$

Concatenation of Languages

Concatenation of languages is usually used for combining two words. When two words in a language are concatenated, they produce a new word.

For example, if the language is $L_1 = \{ a \;\; aa \;\; aaa \;\; aaaa……… \}$

$= \{ a^{\text{odd}} \}$

$= \{a^{2n+1} \text{ for } n= 0, 1, 2, 3….. \}$

If x = aaa , y = aaaaa then xy= aaaaaaaa which is not in L_1.

Similarly, if we concatenate a with b it is ab which is the same as concatenating b with a (ba). But the situation is different while using words in English. For example, classroom is different from roomclass, which does not has any meaning in English.

Concatenation of languages can be defined in the following way:

If L_1 and L_2 are two languages, their concatenation is given as

$L = L_1 . L_2$

Here, $L = \{w/w = xy, \text{ where } x \;\varepsilon\; L_1, y \;\varepsilon\; L_2\}$

For example, if $\sum = \{ 0,1 \}$

$L_1 = \{u \text{ in } \sum^* : \text{number of 0s in u is even} \}$

$L_2 = \{ u \text{ in } \sum^* : \text{u starts with 0 and all the remaining characters are 1s} \}$

$L_1 . L_2 = \{ u \text{ is in } \sum^* : \text{number of 0s in u is odd} \}$

Union of Languages

If L_1 and L_2 are two languages then the union of two languages is given by $L_1 \cup L_2$.

For example, if $L_1 = \{ x \varepsilon \{ 0 \}^* | x$ has even number of zeroes.}

$L_2 = \{ x \varepsilon \{ 0 \}^* | x$ has odd number of zeroes.}

Then the union of two languages $L_1 \cup L_2 = \{0\}^*$.

Intersection of Languages

If L_1 and L_2 are two languages then the intersection of two languages is given by $L_1 \cap L_2$.

For example, if $L_1 = \{ x \varepsilon \{ a \}^* | x$ has even number of 'a's.}

$L_1 = \{\varepsilon$, aa, aaaa, aaaaaa , ...}

$L_2 = \{x \varepsilon \{a, b\}^* | x$ has even number of 'a's or even number of 'b's.}

$L_1 = \{\varepsilon$, aa, bb, aaaa , bbbb, aaaaaa, bbbbbb,.......}

Then the intersection of two languages $L_1 \cap L_2 = \{\varepsilon$, aa, aaaa, ...}

Reversal of Languages

If L is a language then reversal of the language is given by L^R.

For example, if $L = \{0\}^*$ then $L^R = L$.

If $L = \{0^n1^n\}$ then $L^R = \{1^n0^n\}$.

Complement of Languages

If $L = \{x \varepsilon \{a, b\}^* | x$ has even number of 'a's}, then the complement is $L = \{x \varepsilon \{a, b\}^* | x$ has odd number of 'a's}.

5.3 PHRASE STRUCTURE GRAMMARS

The classification of grammar that Chomsky put forward is called the phrase structure grammar. It is classified into four types:

(i) Type 0: Unrestricted Grammar

(ii) Type 1: Context-Sensitive Grammar

(iii) Type 2: Context-Free Grammar

(iv) Type 3: Regular Grammar

(i) Type 0: Unrestricted Grammar

A grammar G is said to be restricted if the productions are of the form $\alpha \rightarrow \beta$ where $a \in (V_N \cup V_T)^+$ and β is in $(V \cup T)^*$.

In an unrestricted grammar, there is no restriction on productions. Also, there is no restriction on the number of variables or terminals, on the left or right, but the null symbol should not be present on the left side.

Unrestricted grammars are more powerful than restricted grammars like regular and context-free grammars.

(ii) Type 1: Context-Sensitive Grammar

The language generated by a Type 1 grammar is called context-sensitive grammar. In Type 1 grammar, the production of the form $S \rightarrow \in$ is allowed.

A grammar G is said to be context sensitive if the productions of the left hand side are not longer than the right side.

The productions are of the form, $x \rightarrow y$

where $x , y \in (V_N \cup V_T)^+$ and $| x | \leq | y |$

Example 5.1: Construct grammar for the language $L = \{a^n b^n c^n \mid n$ is a positive integer$\}$.

Solution:

$S \rightarrow abc \mid aAbc$
$Ab \rightarrow bA$
$Ac \rightarrow Bbcc$
$bB \rightarrow Bb$
$aB \rightarrow aa \mid aaA$

For the input string aabbcc

$S \rightarrow aAbc \rightarrow abAc \rightarrow abBbcc \rightarrow aBbbcc \rightarrow aabbcc$

(iii) Type 2: Context-Free Grammar

A grammar is called Type 2 grammar if the production is of the form $A \rightarrow \alpha$, where $A \in V_N$ and $a \in (V_N \cup \Sigma)^*$.

It is otherwise called context-free grammar. The language generated by a context-free grammar is called context-free language.

$S \rightarrow Aa$, $A \rightarrow a$, $B \rightarrow abc$, $A \rightarrow \in$ are Type 2 productions.

(iv) Type 3: Regular Grammar

A production of the form $A \rightarrow a$ or $A \rightarrow aB$, where $A, B \in V_N$ and $a \in \Sigma$ is called Type 3 production.

A production $S \rightarrow \in$ is allowed in Type 3 grammar, but S does not appear on the right hand side of any production. Type 3 grammar is otherwise called a regular grammar.

Example 5.2: Consider the grammar whose productions P are given as:

$S \rightarrow aY \mid b$, $X \rightarrow a \mid b$. Find the type of the grammar.

Solution: Since the left hand side of the production consists of single variable, it is an example of Type 3 grammar or regular grammar.

Chomsky Hierarchy

In 1955, Chomsky developed a theory of transformation grammar that revolutionized the scientific study of language. He classified the grammars into four types. The relationship among languages is named as *Chomsky Hierarchy*. The original Chomsky hierarchy is given in Figure 5.10.

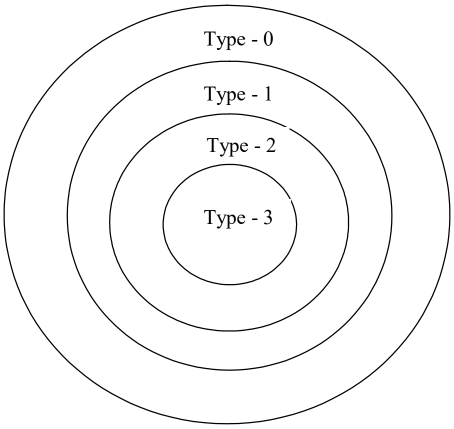


Fig. 5.10 *Chomsky Hierarchy*

The extended Chomsky hierarchy includes the families of deterministic context-free languages and recursive languages. The extended hierarchy is shown in Figure 5.11.

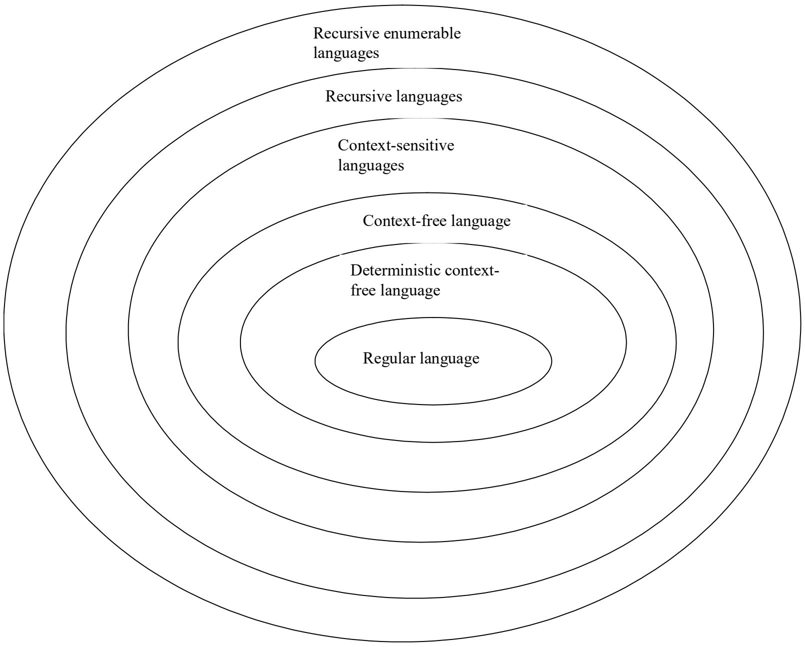

Recursive enumerable languages
Recursive languages
Context-sensitive languages
Context-free language
Deterministic context-free language
Regular language

Fig. 5.11 *Extended Chomsky Hierarchy*

The context-free language L = {w: $n_a(w) = n_b(w)$} where the number of 'a's and 'b's are equal is deterministic but not linear.

On the other hand, the language L = {$a^n b^n$ } ∪ { $a^n b^{2n}$} is linear but not deterministic. This shows the relationship between regular, linear deterministic context free and non-deterministic context-free languages (Refer Figure 5.12).

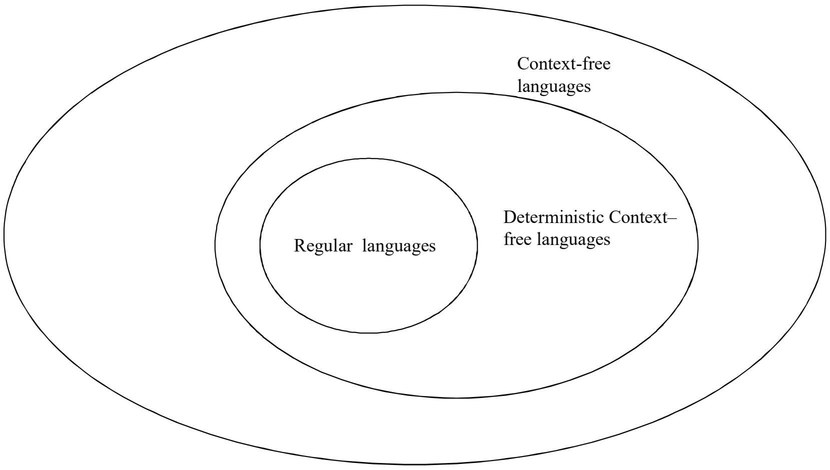


Fig. 5.12 Relationship among Different Languages

Example 5.3: Find the type of grammar for the following:

a) $S \rightarrow Aa, \quad A \rightarrow c \mid Ba \quad B \rightarrow abc$

b) $S \rightarrow ASB \mid d, \quad A \rightarrow aA$

Solution:

a) $S \rightarrow Aa$, $A \rightarrow Ba$, $B \rightarrow abc$ are Type 2 grammars and $A \rightarrow c$ is Type 3 grammar.

b) $S \rightarrow ASB$ is Type 2 grammar and $S \rightarrow d$, $A \rightarrow aA$ are Type 3 grammars.

Example 5.4: Consider the grammar G = {V, T, P, S } where the productions P are given by $S \rightarrow \in$, $S \rightarrow ABA$, $AB \rightarrow aa$, $aA \rightarrow aaaA$, $A \rightarrow a$. Find the type of grammar.

Solution: In this, there are two productions $AB \rightarrow aa$ and $aA \rightarrow aaaA$ such that the left hand side of these productions is not a single variable, hence it is not Type 2 grammar. However, if the length of the left side of the production is not exceeding the length of the right side of the production, the grammar is Type 1 or context-sensitive grammar.

Relation between Recursive and Context-Sensitive Languages

Theorem 5.1: Every context-sensitive language L is recursive.

Proof:

Every context-sensitive language is accepted by the Turing machine and it is therefore, recursively enumerable.

If w is a string then the number of steps in deriving a string is a bounded function of $|w|$. If the set of productions are finite and the string w can be derived from the context-free grammar then $w \in L$, otherwise not. Therefore, every context-sensitive language is recursive.

Theorem 5.2: There exists a recursive language that is not context-sensitive.

Proof:

Consider the set of context-sensitive grammars on T = {a , b}. The grammar has a variable set V denoted by

$V = \{ v_0, v_1, v_2, \ldots\ldots\ldots\ldots \}$.

Every context-sensitive grammar is specified by its productions.

$x_1 \rightarrow y_1 ; x_2 \rightarrow y_2 ; \ldots\ldots\ldots\ldots\ldots\ldots\ldots ; x_m \rightarrow y_m$

We apply homomorphism to the string:

$h (a) = 010$

$h (b) = 0110$

$h (\rightarrow) = 01110$

$h (;) = 01^4 0$

$h (v_i) = 01^{i+5}0$

The context-sensitive grammar can be represented by a string $L ((011^* 0)^*)$.

Let us introduce a proper ordering on $\{ 0, 1 \}^+$, so that we can write strings in the order $w_1, w_2, \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots$.

A given string w_j may not define a context-sensitive grammar. If it defines, it denotes the grammar G_j.

The language L is defined by:

$L = \{ w_i : w_i$ defines a context-sensitive grammar G_i and $w_i \notin L (G_i) \}$.

(i) L is well defined and recursive, you can construct a membership algorithm. Given w_i, check whether it defines a context-sensitive grammar G_i. If it does not defines, $w_i \notin L$. If the string defines a grammar, then $L (G_i)$ is recursive. You can use an algorithm to find out if $w_i \in L (G_i)$, and if not then w_i belongs to L.

(ii) L is not context-sensitive. There exists some w_j such that $L = L (G_j)$. If you assume that $w_j \in L (G_i)$, then by definition w_j is not in L. But $L = L (G_j)$, hence, there is a contradiction. If you assume that $w_j \notin L (G_j)$, then by definition, $w_j \in L$ and hence again there is a contradiction. Therefore, L is not context sensitive.

5.4 REGULAR EXPRESSIONS

The definition of regular expression over the alphabet $\sum$ is defined as follows:

1. ε is a regular expression and it is denoted by $\{ \varepsilon \}$.
2. $\emptyset$ or φ is a regular expression and it denotes the null set or empty set.
3. For any input alphabet $a \, \varepsilon \sum$, a is the regular expression and it is denoted by {a}.

NOTES

Operators of Regular Expression

1. **Union of languages**: If L and M are two languages, the union of two languages is denoted by L ∪ M. The regular expression is denoted by L + M. Union of languages specifies the strings that belong to entire L or M or both.
2. **Concatenation of languages:** If L and M are two languages, the concatenation of two languages is denoted by LM. It is derived by concatenating L and M.
3. **Closure (Kleen's closure or star closure):** The closure of any language L is denoted by L*.
4. If L is a regular expression then (L) is also a regular expression.

Example 5.5: Describe the following sets by regular expression:

1. {0, 1} = Union of 0 and 1 = 0 + 1
2. { Λ , 1,11, 111 , ………………..} = (1)*
3. {1, 11, 111, 1111, ………………} = 1 (1)*
4. {10} = Concatenating 1 with 0 = 10
5. { Λ , ab} = Λ + ab
6. { 01 , 10 } = 01 + 10

Example 5.6: Describe the following language by regular expression.

1. Set of strings of one or more zeroes followed by 1

   Set of one or more zeroes = 0 (0)*

   Set of one or more zeroes followed by 1 = 0 (0)* 1
2. Set of strings of 0s and 1s ending with 01.

   Set of strings of 0s and 1s = (0 + 1)*

   Ends with 01 = (0 + 1)* 01
3. Set of strings of one or more 0s followed by 1

   Set of strings of one or more 0s = 0 (0)*

   Followed by 1 = 0 (0)*1
4. Set of all strings of 0s and 1s beginning with 0 and ending with 1.

   Set of strings of 0s and 1s = (0 + 1)*

   Beginning with 0 and ending with 1 = 0 (0 + 1)* 1
5. Set of strings that has at least one pair of consecutive zeroes

   (0 + 1)* 00 (0 + 1)*
6. All strings having at least two occurrences of substring 00.

   00 (0 + 1)* 00
7. Set of strings with even number of as followed by an odd number of 'b's

   (aa)* (bb)* b

8. Set of all strings over $\Sigma = \{a, b\}$ that begins with a and ends with a

   a (a + b)* a

9. Set of all strings over $\Sigma = \{a, b\}$ that as exactly two a's.

   $b^* ab^* ab^*$

10. Set of all strings over $\Sigma = \{a, b\}$ ending with bb and starting with a.

    $a(a + b)^*$ bb

11. Set of all strings of 0s and 1s in which every 0 is immediately followed by at least two 1s.

    $(1 + 011)^*$

12. Set of all strings of 0s and 1s that end with 11 or 0.

    $(0 + 1)^* (11 + 0)$

13. Set of all strings of 'a's and 'b's that contain at least two 'a's.

    $(a + b)^*$ a $(a + b)^*$ a $(a + b)^*$

Example 5.7: Find the language for the given regular expression.

1. For $\Sigma = \{0, 1\}$, the regular expression is r = (0 + 1)* (0 + 11).

   L (r) ={ 0, 11, 00, 011, 10, 111,……………….}

2. For $\Sigma = \{a, b\}$, the regular expression is r = (aaa)*(bb)* b.

   $L(r) = \{a^{3n}. b^{2m+1}: n \geq 0, m \geq 0\}$

3. For $\Sigma = \{a, b\}$, the regular expression is ab*a.

   L (r) = {aa, aba, abba, abbba, …………….}

4. For $\Sigma = \{a, b\}$, the regular expression is $r = (ab)^*$.

   L (r) = { Λ , ab, abab, ababab,…..}

Example 5.8: Give the regular expression for the following languages:

1. $L = \{ a^n b^m : n \geq 3 , m \geq 2 \}$

   The regular expression is (aaa) a* (bb) b*.

2. $L = \{a^{odd}\}$

   The regular expression is a (aa)* or (aa)*a

3. L = {a, ab, abb, abbb,……} over the alphabet $\Sigma = \{a, b\}$

   The regular expression is a (b^*).

4. $L = \{ab^n c : n \geq 2 , c\ \varepsilon\ (a, b)^+ \}$

   The regular expression is $abbb^* (a + b)^+$.

5. $L = \{w : | w | \bmod 3 = 0$ where $w\ \varepsilon\ (a, b)^* \}$

   Here, | w | represents the length of the string, which should be in multiples of 3.

   The regular expression is $((a + b)^3)^*$.

6. $L = \{w\ \varepsilon\ (a, b)^* : n_a (w) \bmod 3 = 0 \}$

   The regular expression is $(b^*ab^* ab^* ab^*)^*$.

Example 5.9: Give the regular expression and the language for the following set:

1. Set of strings of even number of 'a's followed by odd number of 'b's.

   $L = \{a^{2n}.b^{2m+1}: n \geq 0, m \geq 0\}$

   Regular expression: $(aa)^* (bb)^* b$

2. Set of strings of 'a's and 'b's of length exactly three.

   L = {aaa, aab, aba, abb, baa, bab, bba, bbb}

   Regular expression: (a + b) (a + b) (a + b)

5.5 PUMPING LEMMA FOR REGULAR LANGUAGES

Let L be a regular language. Then there exists a constant n such that for every string w in L where $|w| \geq n$, we can represent w = xyz such that

1. $|y| > 0$
2. $|xy| \leq n$
3. For all $k \geq 0$, the string xy^kz is also in L.

Proof of Pumping Lemma

If L is regular then L = L (A) for some DFA, A. Let n be the number of states of M.

Consider w ε A, where $w = a_1a_2............a_m$ $(m \geq n)$ and each a_i is an input symbol. Let $q_i = \hat{\delta}\ (q_0, a_1a_2.....a_i)$ for i = 0, 1, ..., n, where δ is the transition function and q_0 is the start state of A.

Since there are only n different states it is not possible for (n +1) different q_i's for i = 0, 1,............., n to be distinct. Thus, we can find two different integers i and j with $0 \leq i \leq j \leq n$, such that $q_i = q_j$.

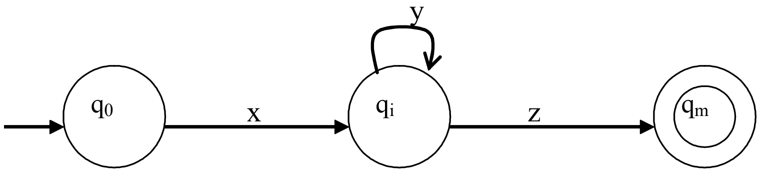
We can break xyz as follows

$x = a_1a_2 \ldots\ldots\ldots\ldots a_i$

$y = a_{i+1}a_{i+2} \ldots\ldots\ldots\ldots a_j$

$z = a_{j+1}a_{j+2} \ldots\ldots\ldots\ldots a_m$

Then by repeating the loop from q_i to q_i with label $a_{i+1}...............a_j$, zero or more times, we can show that xy^iz is accepted by A.



Every string longer than the number of states must cause a state to repeat.

Applications of Pumping Lemma

Pumping Lemma is used to prove that certain sets are not regular. The steps needed for proving that a given set is not regular are as follows:

1. Assume L is regular. Let n be the number of states in the corresponding finite automata.

Check Your Progress

1. What is a language processor?
2. Define context free language.
3. Write any one operator of regular expression.
4. What is the use of pumping lemma?

5.6 REGULAR GRAMMAR

In theoretical computer science, a regular grammar is a formal grammar that describes a regular language. A regular grammar is a left or right regular grammar. Right regular grammar is also known as right linear grammar and left regular grammar are also known as left linear grammar.

Right Linear and Left Linear Grammar

Right Linear Grammar

A grammar G is said to be right linear if each production has one variable at the left side and the right side consists of zero or more number of terminals followed by an optional single variable or ∈.

$A\rightarrow aB, A\rightarrow a$

These are the examples of right linear grammar.

Left Linear Grammar

A grammar G is said to be left linear if each production consists of a single variable at the left side and the right side consists of an optional single variable followed by any number of terminals.

$A\rightarrow Ba, A\rightarrow a$

These are the examples of left linear grammar.

A regular grammar is one that is either right linear or left linear.

A linear grammar is a grammar in which at most one variable can occur on the right side of any production, but the position of the variable in not restricted.

Example 5.10: Consider the grammar G = {{S, A, B}, {a, b}, S, P}, where P is given as

$S\rightarrow A$

$A\rightarrow bA / \in$

$B\rightarrow Ba$

Check whether the grammar is regular or not.

Solution: The productions for A, B are right linear and left linear but the production for S is neither left linear nor right linear.

Therefore, it is not regular. But it is a linear grammar.

Notes:

1. Regular and linear grammars are context free.
2. Context-free grammars are not always linear.

Sentential Forms

If G = (V, T, P, S) is a CFG, then any string in $(V \cup T)^*$ such that $S \Rightarrow \alpha$ is a sentential form.

If $\alpha \in L(G)$ then

$S \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots\ldots\ldots\ldots \Rightarrow \alpha_n \Rightarrow \alpha$ is the derivation of α. The strings α_1, α_2, ..., α_n may contain variables as well as terminals. These are called sentential forms of the derivation.

If $S \overset{*}{\Rightarrow} \alpha$ and if the variable is replaced from the left side, then it is called left sentential form.

If $S \overset{*}{\Rightarrow} \alpha$ and if the variable is replaced from the right side, then it is called right sentential form.

Two grammars G_1 and G_2 are said to be equivalent if they generate the same language, i.e.,

$L(G_1) = L(G_2)$

5.7 CONTEXT-FREE GRAMMARS

Context-free languages play a central role in the natural languages since 1950s and in the compilers since 1960s. CFG is the basis of Backus-Naur Form (BNF) syntax.

Context-free grammars are used in the implementation of parsers in compiler design. We can represent the context-free languages or the derivations by means of parse trees. Grammars follow the process of derivation by which the strings are derived from the language of the grammar. Design of compilers and interpreters for programming languages are obtained from the grammar for the language.

Definition

A context-free grammar or CFG is represented by G= (V, T, P, S) where

V represents a finite set of variables or non-terminals.

T represents a finite set of symbols called terminals.

P is a finite set of productions or rules that represent the recursive definition of a language.

S represents the start symbol.

A grammar is said to be context free if the left side of the production is a single non-terminal and the right side has a special form.

$A \to \alpha$ is a production where $A \varepsilon V$ and $\alpha \in (V \cup T)^*$.

The productions are used to derive one string from another string.

If $\alpha \to \beta$ is a production in a grammar G, then we can replace α by β in that string.

In general, it can be written as

$\alpha \to \beta$

Here, $\to$ indicates unspecified number of steps, i.e., zero or more number of steps.

If $A \to \alpha$, then α is called a sentential form.

If $A \to \alpha$ is a production, where $A \varepsilon V$, then it is called A-production.

If $A \to \alpha_1, A \to \alpha_2, \ldots\ldots\ldots\ldots, A \to \alpha_n$ are A-productions, then it can be written as

$A \to \alpha_1|\alpha_2|\ldots\ldots\ldots\ldots\ldots\ldots|\alpha_n$

Construction of a Finite Automata L(G) for a given Regular Grammar G

Let $G = \{\{A_0, A_1, \ldots\ldots., A_n\}, \Sigma, P, A_0\}$

The transition system is constructed such that

a) It consists of states corresponding to variables.

b) The initial state q_0 corresponds to A_0.

c) Transitions correspond to the productions in P.

If any production is of the form $A_i \to \alpha$, where α is any terminal, then the transition terminates at this state and this is the unique final state.

The transition system A is defined as

$\{q_0, q_1, \ldots\ldots., q_f\}, \Sigma, \delta, q_0, \{q_f\}$, if

(i) Each production $A_i \to aAj$, induces a transition from q_i to q_j with label a.

(ii) Each production $A_i \to \alpha$, induces a transition from q_i to q_f with label α.

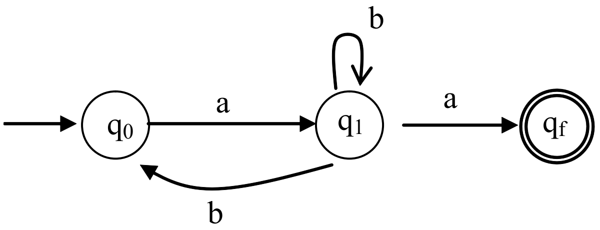
Example 5.11: Let $G = \{\{A_0, A_1,\}, \{a, b\}, P, A_0\}$ where the productions are $A_0 \to aA_1, A_1 \to bA_1, A_1 \to a, A_1 \to bA_0$. Construct a transition system A accepting L (G).

Solution: Since there are two variables A_0 and A_1, there are two corresponding states q_0 and q_1, and a final state q_f.

If the production is $A_0 \to aA_1$, it has a transition from q_0 to q_1 with label a. Similarly, if the production is $A_1 \to bA_1$, it has a transition from q_1 to q_1 with label b. If the production is $A_1 \to bA_0$, it has a transition from q_1 to q_0 with label b.

If the production is $A_1 \to a$, it has a transition from q_1 to q_f with label a.

The transition system A is given by



Example 5.12: Construct a transition system A, if the productions are given as

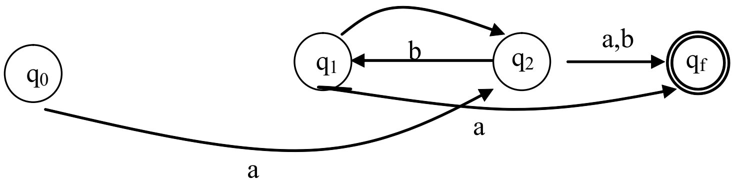
$$S \to aA_2,\ A_2 \to bA_1, A_2 \to b, A_2 \to a, A_1 \to aA_2, A_1 \to a.$$

Solution: Since there are three variables, there are four states $\{q_0, q_1, q_2, q_f\}$.

NOTES

Here, q_0 corresponds to S, q_1 corresponds to A_1, q_2 corresponds to A_2, q_f is the final state.

$S \to aA_2$ induces a transition from $q_0 \to q_2$ with label a. $A_1 \to aA_2$ induces a transition from q_1 to q_2 with label a. $A_2 \to b$ and $A_2 \to a$ have transitions from q_2 to q_f with labels a, b.



Construction of Regular Grammar for a Given Regular Language or Given DFA A

Let $A = \{\{q_0, \ldots\ldots, q_n\}, \Sigma, \delta, q_0, F\}$.

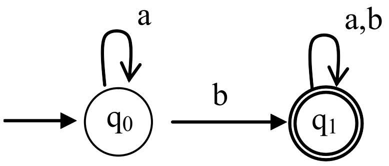
The grammar is constructed in such a way that the productions correspond to the transitions.

If there is a transition from q_i with a label a to q_j, then $\delta(q_i, a) = q_j \notin F$. Then the production is given as $A_i \to aA_{j.}$

If $\delta(q_i, a) = q_j \in F$, then the productions are given as $A_i \to aA_j, A_i \to a$.

Example 5.13: Construct a regular grammar for a given regular language $R = a^* b(a+b)^*$.

Solution: Given the regular language, it is converted into DFA.



From this the grammar is constructed such that

$G = \{\{A_0, A_1\}, \Sigma = \{a,b\}, P, A_0\}$

Where the productions are given by

$A_0 \to aA_0$

$A_0 \to bA_1,\ A_0 \to b$

$A_1 \to aA_1 \mid bA_1$

$A_1 \to a|b$

Language Generated by a Grammar

The set of all terminal strings that can be derived from the start symbol is called the language generated by the grammar L(G).

$L(G) = \{w \text{ in } T \mid S \rightarrow w\}$

The terminal strings are derived from the context-free grammar by substituting the variable on the right side of the production.

The substitution can be done any number of times until no non-terminal variable is present on the right side.

A language L is said to be context free, if and only if, there is a context-free grammar G such that L = L(G).

Example 5.14: Consider the grammar G = {{S}, {a, b}, S , P } where the productions are given as

$S \rightarrow aSbb$

$S \rightarrow abb \mid \in$

Solution:

$S \rightarrow aSbb \qquad S \rightarrow abb$

$\rightarrow aabbbb \qquad = a^2 b^4$

$S \rightarrow abb \qquad S \rightarrow \in$

$= a^1 b^2$

Therefore, the language generated by the grammar is

$L(G) = \{ a^n b^{2n} \mid i \geq 1 \}$

5.8 DERIVATION TREES

Another way of representing the derivation, independent of the order in which the productions are used is the derivation tree. A derivation tree is also known as a parse tree. A derivation tree is an ordered tree in which the nodes are labelled with the left sides of the production and the children of the nodes represent the corresponding right sides. A derivation tree begins with the start symbol (the root) and ends with the terminals (leaves).

Definition

Let G= (V, T, S, P) be a CFG. A derivation tree for G is possible if and only if it has the following properties:

1. The label of the root is S.
2. Every vertex has a label $V \cup T \cup \{\in\}$.
3. Every interior vertex has a label from V.
4. If V has a label $A \in V$ and vertices $V_1, V_2, \ldots\ldots, V_n$ then the productions are given as

   $A \rightarrow V_1, V_2, \ldots\ldots\ldots\ldots\ldots\ldots\ldots, V_n$
5. A leaf labeled $\in$ has no siblings, i.e., a vertex with a child labeled $\in$ can have no children.

Partial Derivation Tree

A tree that has the Properties 3, 4 and 5 but the Property 1 does not necessarily hold and has the Property – every leaf has a label $V \cup T \cup \{\in\}$ – is said to be a partial derivation tree.

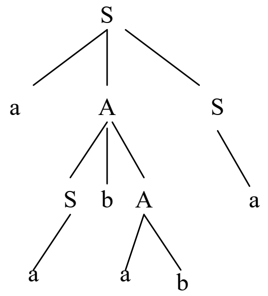
Yield of a Derivation Tree

The string of terminals obtained by traversing the leaves of the tree from the left to the right (depth-first manner) is said to be the yield of the tree. In other words, the yield of a derivation tree is the concatenation of the leaves of the parse tree from left to right.

Example 5.15: Construct a derivation tree for the grammar G = {{ S, A}, { a, b}, P, S } where P is given by

$$S \rightarrow aAS,\ S \rightarrow a,\ A \rightarrow SbA \mid ab$$

Solution: For the string aababa, the derivation tree is given by



Here, the string aSbAa is a partial derivation tree for G, while the string aababa is a derivation tree. The yield of a partial derivation tree is a sentential form of G while the yield of a derivation tree is a sentence of L(G).

Subtree

A subtree of a derivation tree T is a tree whose root is some vertex V of T, together with all the descendants of V along with their labels and the edges that connect the descendants of V.

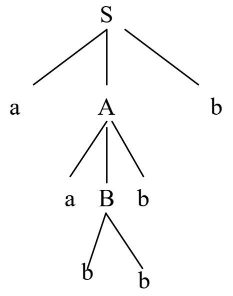
A subtree looks like a derivation tree if the label of the root is not S. It is called an A-tree if the label of the root is A.

Example 5.16: Consider the grammar G with production

$$S \rightarrow aAb,\ A \rightarrow aBb,\ B \rightarrow bb$$

For the string aabbbb, construct the derivation tree.

Solution: For the string aabbbb, following is the derivation tree:



Leftmost and Rightmost Derivation

In the derivation of a string, at each step if we replace the leftmost variable by any one of its productions, it is called the leftmost derivation. The leftmost derivation is denoted by $A \Rightarrow \alpha$, i.e., the terminal string is obtained in zero or more number of steps.

In the derivation of a string, at each step if we replace the rightmost variable by any one of its production, then it is called the rightmost derivation. The rightmost derivation is denoted by $A \Rightarrow \alpha$, i.e., the terminal string is obtained in zero or more number of steps.

Example 5.17: Let G be a CFG where the productions are

$S \to aB \mid bA$

$A \to a \mid aS \mid bAA$

$B \to b \mid bS \mid aBB$

For the string aabbabab find:

(i) Leftmost derivation

(ii) Rightmost derivation

(iii) Parse tree

Solution:

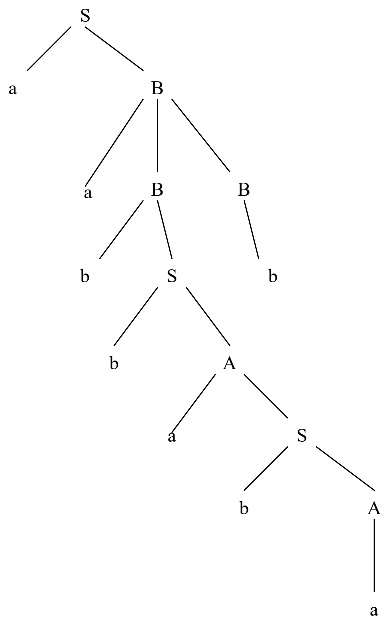
(i) Leftmost derivation for aabbabab

$$S \to aB \to aaBB \to aabSB \to aabbAB \to aabbaSB$$
$$\to aabbabAB \to aabbabaB \to aabbabab$$

(ii) Rightmost derivation for aabbabab

$$S \to a\underline{B} \to aaB\underline{B} \to aaBb\underline{S} \to aaBba\underline{B} \to aaBbab\underline{S}$$
$$\to aaBbaba\underline{B} \to aa\underline{B}babab \to aabbabab$$

(iii) Parse tree



5.9 SENTENTIAL FORMS

A parse tree is a representation of the structure and grammar of a computer language. The process of parsing helps to translate human understandable grammar to a computer understandable code. When used in a compiler, the parse tree is the data structure of choice to represent the source program. In a compiler, the parse tree facilitates the translation of the source program into executable code by allowing natural, recursive functions to perform this translation process.

The elements of a parse tree are called nodes. Like in a parent-child relationship, a node must have one parent. It is drawn above the node. It can have zero or many children drawn below. Lines connect parents to their children.

Root is a node that has no parent node. This node appears at the top of the tree. Those nodes that have no children are called leaves and nodes that are not leaves are called interior nodes.

The child of a child node is a descendant of that node. A parent of a parent node and so on are called ancestors. In short, nodes are ancestors and descendents of themselves.

Children of a node are ordered 'From the Left' and are drawn likewise. If node N is to the left of node M, then all the descendants of N are considered to be the descendants of M.

Abstract Syntax Tree

An Abstract Syntax Tree (AST) represents the structure of a source string in a more economical manner. The word 'abstract' implies that it is a representation designed by a compiler for its own purpose. Thus, the designer has total control over the information represented in an AST. It suggest that an AST for a source string is not unique as a parse tree.

Consider the grammar G = (V, T, P, S). The parse trees for G are trees with the following given conditions:

1. A variable in V labels each interior node.
2. Again a variable, a terminal or $\in$ labels each leaf. At the same time, in case of being the only child of its parents the leaf will be labelled $\in$.
3. And if an interior node is labelled A and its children are labelled X_1, $X_{2,}$ X_3,....., X_k from the left, then A $\rightarrow$ X_1, X_2, X_3,....., X_k is a production in P. X can be $\in$ if it is labelled as the only child and A $\rightarrow$ $\in$ is the production of G.

See the following production:

1. $E \rightarrow I$
2. $E \rightarrow E + E$
3. $E \rightarrow E * E$
4. $E \rightarrow (E)$
5. $I \rightarrow a$

6. $I \rightarrow b$
7. $I \rightarrow Ia$
8. $I \rightarrow Ib$
9. $I \rightarrow I0$
10. $I \rightarrow I1$

The grammar for this expression is stated as

$G = (\ \{E, I\}, T, P, E)$

The parse tree shows I + E from E (Refer Figure 5.13).

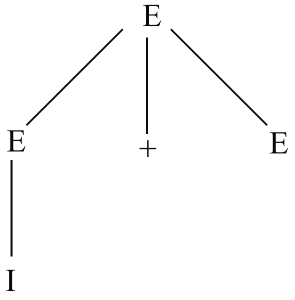


Fig. 5.13 *A Parse Tree*

Example 5.18: (Grammar of palindrome)

$G_{pa} = \{\{P\},\{0,1\}, A, P\}$

Where A represents the set of five productions.

Define context-free grammar rule for a palindrome

Solution: $P \rightarrow \in$

$P \rightarrow 0$

$P \rightarrow 1$

$P \rightarrow 0P0$

$P \rightarrow 1P1$

A parse tree $P \rightarrow 00110$

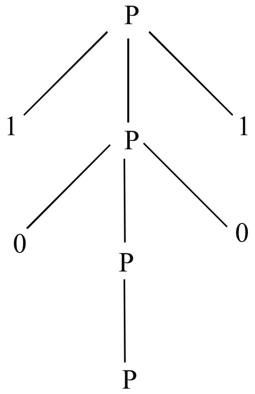


Fig. *A Parse Tree*

The production used at the root is

$$P \rightarrow 1P1$$

At the middle, the child of the root is

$$P \rightarrow 0P0$$

At the bottom, $P \rightarrow \in$.

The label $\in$ is only used once on the parse tree.

Parse Trees and Derivation

A parse tree is a graphical depiction for a source that filters out the options about the substitution order. Some non-terminal 'A' labels every interior node of a parse tree and so the children of the node are labelled from left to right. This is done by the symbols in the right side of the production by which this A was replaced in the source. Non-terminal or terminals label the leaves of the parse tree. These are read from the left to right and are known as the yield of frontier of the tree.

Sentence and Sentential Form

Given a grammar G with start symbol + you can use → relation to define L(G)—the language generated by G. Strings in L(G) may contain only terminal symbols of G. A string of terminal w is in L(G) if and only if

$$S \overset{+}{\rightarrow} w$$

The string w is called a sequence of G. A language that can be generated by a grammar is said to be a context-free language. If two grammars generate the same language, the grammars are said to be equivalent. If

$$S \overset{+}{\rightarrow} \alpha$$

Where α may contain non-terminals, then α is a sentential form of G. A sentence is a sentential form with no non-terminals.

Let us take the example of a parse tree –(id * id).

Consider any derivation $\alpha_1 \rightarrow \alpha_2 \rightarrow \alpha_3 \rightarrow \ldots \rightarrow \alpha_n$.

Where α_1 is a single non-terminal A.

For each sentential α_1, α_2, etc., you can construct a parse tree whose yield is α_i. The process is an induction on i.

Basis: The tree for α_1

A is a single node labelled A.

Induction: Suppose you have constructed a parse tree whose yield is

$$\alpha_{i-1} = X_1\, X_2\, X_3 \ldots X_k$$

Suppose, α_i is derived from α_{i-1} by replacing X, a non-terminal, by

$$\beta_{i-1} = Y_1\, Y_2\, Y_3 \ldots Y_r$$

At the ith step of the derivation, production is

$X_j \rightarrow \beta$ applied to α_{i-1} to derive,

$\alpha_i = X_1 X_2 \ldots X_{j-1} \beta X_j \ldots X_k$

You will find jth leaf from the left in the current parse tree. This leaf is labelled X_j, with *r* children labelled from left (Refer Figure 5.14).

As a special case

If *r* = 0, i.e.,

B=∈ then you can give the jth leaf one child labelled ε.

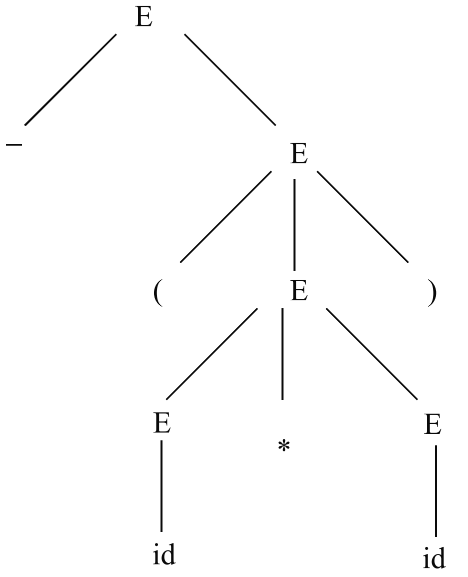


Fig. 5.14 *A Parse Tree*

Top-Down Parsing

In top-down parsing, the derivation is for the leftmost input string. A common type of top-down parsing known as recursive descent parsing involves backtracking, i.e., it repeatedly scans the input. In natural language parsing, tabular methods, such as dynamic programming algorithm, is preferred as the latter backtracking is not very efficient. The recursive descent parsing where no backtracking is required is called predictive parsing.

Example 5.19

Consider the grammar

$S \rightarrow pQr$

$Q \rightarrow lm \mid l$

And input string w=plr

In the beginning, you can create a tree consisting of a single node labelled S to make a parse tree for this string top-down the first symbol of w and an input point to p.

NOTES

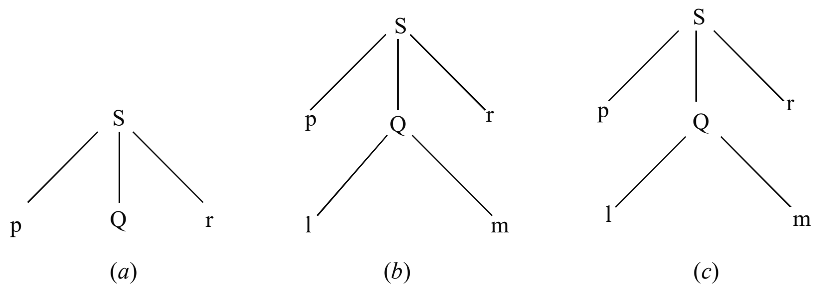


Fig. *Using the First Alternative for Q to Obtain Tree*

The leftmost leaf, labelled p, matches the first symbol of w, so you can now advance the input pointer to the next leaf, labelled Q.

Expand Q using the first alternative for Q to obtain tree.

Now match a second input symbol, so the next input points to r, the third input symbol, and compare r against m. Since m and r do not match, a failure is reported. It goes back to Q to get another alternative for Q. While going back the input pointer is set to position 2, the position it had when it first came to Q. Thus, the input pointer should be stored in a local variable for the procedure to access Q. Now, Figure (c) is obtained. The leaf l matches the second symbol of w and the leaf r matches the third symbol.

Implementing Top-Down Parsing

The following features are needed to implement top-down parsing:

1. Source String Marker (SSM): This points to the first unmatched symbol in the source string.
2. Prediction making mechanism: This mechanism systematically selects the right hand side alternatives of production during prediction-making. It must ensure that any string in L_G can be derived from S.
3. Matching and backtracking mechanism: This mechanism matches every terminal symbol generated during a derivation with the source symbol pointed to SSM. Backtracking is performed if the match fails. This involves resetting the Current Sequential Form (CSF) and SSM.

Predictive Parser

You can get a grammar that can be parsed by recursive descent parser. This needs no backtracking. To make a predictive parser, given the current input symbol α and the non-terminal A to be expanded, one of the alternatives of production is

$$A \rightarrow \alpha_1 | \alpha_2 | \ldots | \alpha_n$$

Bottom-Up Parsing

Bottom-up parsing is also known as shift-reduce parsing. It tries to make a parse tree for an input string starting at the leaves (the bottom) and writing up towards the root.

Consider the grammar

$S \rightarrow pPQt$
$P \rightarrow Pqr \mid q$
$Q \rightarrow s$

Consider the sentence pqqrst.

1.	pqqrst	
2.	pPqrst	[as P → q]
3.	pPst	[as P → Pqr]
4.	pPQt	[as Q → s]
5.	S	[as S → pPQt]

Thus, by a series of four declines, you are able to condense pqqrst to S.

Operator Precedence Grammar and Parsing

A grammar that produces more than one parse tree for some sentence is said to be ambiguous. An ambiguous grammar produces more than one leftmost or more than one rightmost derivation for the same sentence.

Consider the expression

$$E + E * E$$

There are two parse trees for this expression (Refer Figure 5.15).

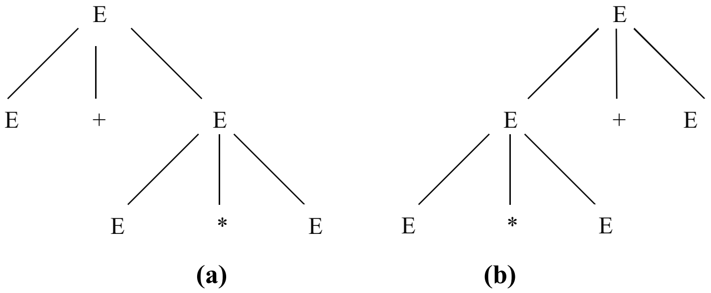


Fig. 5.15 *Two Parse Trees for Expression E + E * E*

Now this derivation shows the ambiguity of the grammar. Thus, a CFG, G = (V, T, P, S) is unclear if there is at least one string w in T* for which you can find two different parse trees each with root labelled s and yield w. The grammar is considered unambiguous if each string has only one parse tree in it.

There are two causes for ambiguity of grammar:

1. The precedence operator is not enforced.

   In the example E + E * E, if you consider the precedence operator *, then there is no ambiguity, i.e., E + (E * E) is right parse tree and (E +E) * E is not considered.

2. A sequence of identical operators can be grouped either from the right or from the left.

   Like in E + E+ E or E* E * E, you can see the two parse trees, each labelled E (Refer Figure 5.16).

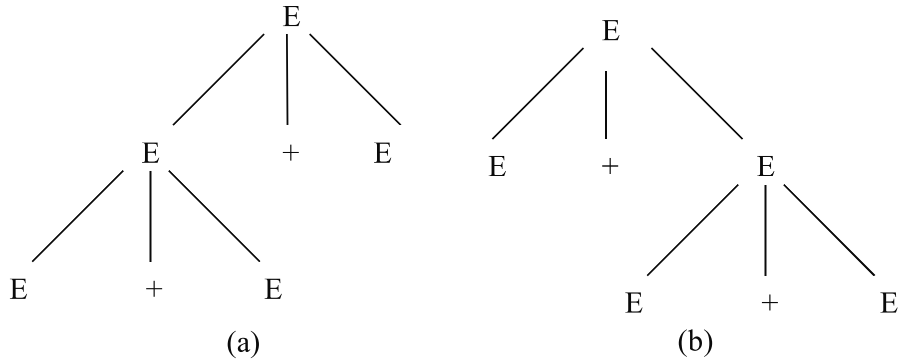


Fig. 5.16 *Two Causes for Ambiguity of Grammar*

This is ambiguous grammar, as there are two parse trees for the same expression. Since addition and multiplication are associative, it does not matter whether you group them from left to right or vice versa. The conventional approach is to insist on grouping from the left, so the structure is the only correct grouping of two + signs.

Operator Precedence Grammar

An Operator Precedence Grammar (OPG) is an operator in which the precedence between operators is unique.

There are three levels of binding strength in a language:

(i) Factor
(ii) Term
(iii) Expression

(i) Factor

An adjacent operator cannot break apart a factor, either through a * or a+. The only factors in our expression language are:

- ***Identifiers*****:** It is not possible to separate the letters of two identifiers by attaching an operator.
- ***Parenthesized expression*****:** Parentheses prevent what is inside from becoming the operand of any operator outside the parentheses.

(ii) Term

A term is an expression that cannot be broken by the + operator. Where + and * are only operators, a term is a product of one or more factors. A proper grouping of a+ a * b is a + (a * b) and of (a* b) + a is (a*b) + a.

(iii) Expression

An expression refers to any possible expression, including those that can be broken by either an adjacent * or an adjacent +.

$$I \rightarrow a|\ b|\ Ia \mid Ib \mid I0 \mid I1$$

$$I \rightarrow I \mid (E)$$

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

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

is an unambiguous expression grammar.

Now, parse tree for a + a * b for this grammar is shown in Figure 5.17.

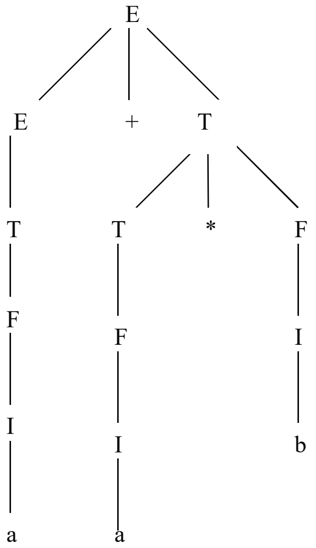


Fig. 5.17 *Parse Tree for a + a * b*

1. The string derived from T is a term which is in a sequence of one more factors, connected by '*'s. A factor by definition is either an identifier or a parentheses expression. F is defined the same as a factor s.
2. As a production factor, the only parse tree for a sequence of factors is the one that breaks

   $$f_1 * f_2 *f_3 * \ldots * f_n \text{for } n > 1$$

   into a term $f_1 * f_2 *f_3 * \ldots * f_{n-1}$ and a factor f_n. Now F cannot derive an expression like $f_{n-1} * f_n$ without introducing parentheses around them. This is not possible when using the production,

   $$T \rightarrow T * F$$

   The F derives anything but the last of the factors.
3. When the production $E \rightarrow E + T$ is used to derive $t_1 + t_2 + t_3 + \ldots + t_n$ the T must derive only t_n and the ε in the body derives $t_1 + t_2 + t_3 + \ldots + t_n$.

Operator Precedence Parsing

Operator precedence parsing explains three disjointed precedence relations <·, =·, ·>, between certain pairs of terminals.

p <· q   means a 'yields precedence to' b

p ·> q   means a 'takes precedence over ' b

p =· q means a 'yields same precedence to' b

For example, if * operator has precedence of + operator then it is written as

* · > + or   + <· *

Operator Precedence Matrix

Operator Precedence Matrix (OPM) represents operator precedence relations between pairs of operator. The entry OPM (a, b) represents the precedence of operator a operator b in a sentential form …aPb… where P may be a null string.

The operator-precedence relation is shown in Table 5.1.

Table 5.1 *Operator-Precedence Relations*

	id	+	*	$
Id		·>	·>	·>
+	<·	·>	<·	·>
*	<·	·>	·>	·>
$	<·	<·	<·	

Deterministic parsing refers to parsing algorithms that do not back up. LR-parsers are an example of deterministic parsing.

LR Parsers

An LR parser is a type of Shift/Reduce (S/R) parser that does not uses backtracking. It is the most general type of non-backtracking S/R parser. Its main drawback is the difficulty of constructing suitable parse tables. LR parsers also tend to produce larger tables than LL parsers. The LR parser was first described by Knuth in 1965. All LR parsers use the same algorithm and they differ in the way the table is generated. An LR parser consists of a stack and a state table. It takes input in the form of a series of symbols. The state table lists actions for each terminal symbol in each state and also lists goto states for non-terminal symbols. The stack contains value and state pairs and initially contains just the state 0. The action table for each state may contain one of the following four values:

- Shift and move to state *n*
- Reduce using rule number *n*
- Error
- Accept

The algorithm proceeds by reading the current state from the top of the stack and the next symbol from the input. It looks up the action for that state and symbol in the table. A shift action causes the parser to push the read symbol and the new state onto the stack. A reduce action causes the parser to pop a suitable number of symbols off the stack, make the state one now on top of the stack and push the non-terminal representing rule *n* onto the stack, followed by the state specified for that non-terminal in the goto part of the state table. Error causes the parser to enter error handling mode and accept causes the parser to accept the grammar. Table 5.2 summarizes the two LR(0) parsing tables for this grammar.

Table 5.2 *Two LR(0) Parsing Tables*

	Action					goto	
State	*****	**+**	**0**	**1**	**$**	**E**	**B**
0			s1	s2		3	4
1	r4	r4	r4	r4	r4		
2	r5	r5	r5	r5	r5		
3	s5	s6			acc		
4	r3	r3	r3	r3	r3		
5			s1	s2			7
6			s1	s2			8
7	r1	r1	r1	r1	r1		
8	r2	r2	r2	r2	r2		

The action table is indexed by a state of the parser and a terminal including a special non-terminal $ that indicates the end of the input stream and contains three types of actions. A *shift* that is written as 's*n*' to indicate that the next state is *n*, a *reduce* that is written as 'r*m*' and indicates that a reduction with grammar rule *m* should be performed, and an *accept* that is written as 'acc' and indicates that the parser accepts the string in the input stream. An LR(0) *item* of a grammar *G* is a production of *G* with a dot at some position of the right side indicating how much of a production we have seen up to a given point. For example, for the production E → E + T the items are appeared as follows:

$[E \rightarrow .E + T]$

$[E \rightarrow E. + T]$

$[E \rightarrow E +. T]$

$[E \rightarrow E + T.]$

These are called LR(0) items because they contain no explicit reference to lookahead. The central idea of the LR method is first to construct a deterministic finite automaton to recognize viable prefixes from the grammar.

Constructing LR(0) Parsing Tables

Items, item sets and closure of item sets are the prime factors which are used to construct LR(0) parsing table. They are discussed below:

Items

The construction of these parsing tables is based on the notion of LR(0) item simply called item which are grammar rules with a special dot added somewhere in the right hand side. For example the rule E → E + B has the following four corresponding items:

$E \rightarrow \cdot\ E + B$

$E \rightarrow E \cdot + B$

$E \rightarrow E + \cdot\ B$

$E \rightarrow\ E + B \cdot$

NOTES

Exceptions are rules of the form A → ε with which only the item A → · corresponds. These rules will be used to denote the state of the parser. The item E → E · + B, for example indicates that the parser has recognized a string corresponding with E on the input stream and now expects to read a '+' followed by another string corresponding with E.

Item Sets

It is usually not possible to characterize the state of the parser with a single item because it may not know in advance which rule it is going to use for reduction. For example, if there is also a rule E → E * B then the items E → E · + B and E → E · * B will both apply after a string corresponding with E has been read. Therefore we will characterize the state of the parser by a set of items—in this case the set {E → E · + B, E → E · * B }.

Closure of Item Sets

An item with a dot in front of a non-terminal, such as E → E + · B, indicates that the parser expects to parse the non-terminal B next. To ensure that the item set contains all possible rules (the parser may be in the midst of parsing), it must include all items describing how B itself will be parsed. This means that if there are rules such as B → 1 and B → 0 then the item set must also include the items B → · 1 and B → · 0. In general this can be formulated as follows:

If there is an item of the form A → v·Bw in an item set and in the grammar there is a rule of the form *B* → *w'* then the item *B* → · *w'* should also be in the item set.

Any set of items can be extended such that it satisfies this rule: simply continue to add the appropriate items until all nonterminals preceeded by dots are accounted for. The minimal extension is called the closure of an item set and written as clos(I) where *I* is an item set. We will take these closed item sets as the states of the parser, although only the ones that are actually reachable from the begin state will be included in the tables.

5.10 NOTIONS OF SYNTAX ANALYSIS

Another important application of stacks is the conversion of expressions from the infix notation to the postfix and prefix notations. The general way of writing arithmetic expressions is known as the **infix notation** where the binary operator is placed between two operands on which it operates (For simplicity, we have ignored expressions containing unary operators). For example, the expressions `a+b` and `(a-c)*d`, `((a+b)*(d/f)-f)` are in the infix notation. The order of evaluation in these expressions depends on the parentheses and the precedence of operators. For example, the order of evaluation of the expression `(a+b)*c` is different from that of `a+(b*c)`. As a result, it is difficult to evaluate an expression in the infix notation. Thus, the arithmetic expressions in the infix notation are converted to another notation, which can be easily evaluated by a computer system to produce the correct result.

A Polish mathematician, Jan Lukasiewicz, suggested two alternative notations to represent an arithmetic expression. In these notations, the operators are written either before or after the operands on which they operate. The notation in which an operator occurs before its operands is known as the **prefix notation** (also known as **Polish notation**). For example, the expressions `+ab` and `*-acd` are in the prefix notation. On the other hand, the notation in which an operator occurs after its operands is known as the **postfix notation** (also known as the **Reverse Polish** or **suffix notation**). For example, the expressions `ab+` and `ac-d*` are in the postfix notation.

A characteristic feature of the prefix and postfix notations is that the order of evaluation of the expression is determined by the position of the operator and operands in the expression. That is, the operations are performed in the order in which the operators are encountered in the expression. Hence, parentheses are not required for the prefix and postfix notations. Moreover, while evaluating the expression, the precedence of the operators is insignificant. As a result, they are compiled faster than the expressions in the infix notation. Note that the expressions in the infix notation can be converted to both the prefix and postfix notation. This section discusses both types of conversions.

Conversion of Infix to Postfix Notation

To convert an arithmetic expression from an infix notation to a postfix notation, the precedence and associativity rules of operators are always kept in mind. The operators of the same precedence are evaluated from left to right. This conversion can be performed either manually (without using stacks) or by using stacks. The steps for converting the expression manually are as follows:

1. The actual order of evaluation of the expression in the infix notation is determined by inserting parentheses in the expression according to the precedence and associativity of operators.
2. The expression in the innermost parentheses is converted into the postfix notation by placing the operator after the operands on which it operates.
3. Step 2 is repeated until the entire expression is converted into a postfix notation.

For example, to convert the expression `a+b*c` into an equivalent postfix notation, these steps are followed:

1. Since the precedence of `*` is higher than `+`, the expression `b*c` has to be evaluated first. Hence, the expression is written as

   `(a+(b*c))`

2. The expression in the innermost parentheses, that is, `b*c` is converted into its postfix notation. Hence, it is written as `bc*`. The expression now becomes

   `(a+bc*)`

3. Now the operator + has to be placed after its operands. The two operands for + operator are ~~a~~ and the expression `bc*`. The expression now becomes

   `(abc*+)`

Hence, the equivalent postfix expression is

```
abc*+
```

When expressions are complex, manual conversion becomes difficult. On the other hand, the conversion of an infix expression into a postfix expression is simple when it is implemented through stacks. In this method, the infix expression is read from left to right, and a stack is used to store the operators and the left parenthesis temporarily. The order in which the operators are pushed onto and popped from the stack depends on the precedence of operators and the occurrence of parenthesis in the infix expression. The operands in the infix expression are not pushed onto the stack; rather they are directly placed in the postfix expression. Note that the operands maintain the same order as the original infix notation.

Algorithm 5.1 Infix to Postfix Conversion

```
infixtopostfix(s, infix, postfix)

1. Set i = 0
2. While (i < number_of_symbols_in_infix)
      If infix[i] is a whitespace or comma
            Set i = i + 1 and go to step 2
      If infix[i] is an operand, add it to postfix
      Else If infix[i] = '(', push it onto the stack
      Else If infix[i] is an operator, follow these steps:
        i. For each operator on the top of stack whose precedence
           greater than or equal to the precedence of the curre
           operator, pop the operator from stack and add it
           postfix
        ii. Push the current operator onto the stack
      Else If infix[i] = ')', follow these steps:
        i. Pop each operator from top of the stack and add it
           postfix until '(' is encountered in the stack
        ii. Remove '(' from the stack and do not add it to postfix
      End If
      Set i = i + 1
   End While
3. End
```

For example, consider the conversion of the following infix expression to the postfix expression:

```
a-(b+c)*d/f
```

Initially, a left parenthesis '(' is pushed onto the stack, and the infix expression is appended with a right parenthesis ')'. The initial states of the stack, infix expression and postfix expression are shown in Figure 5.18.

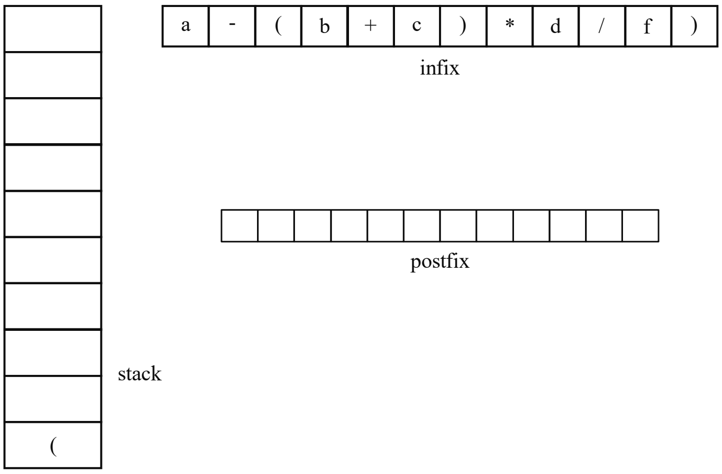
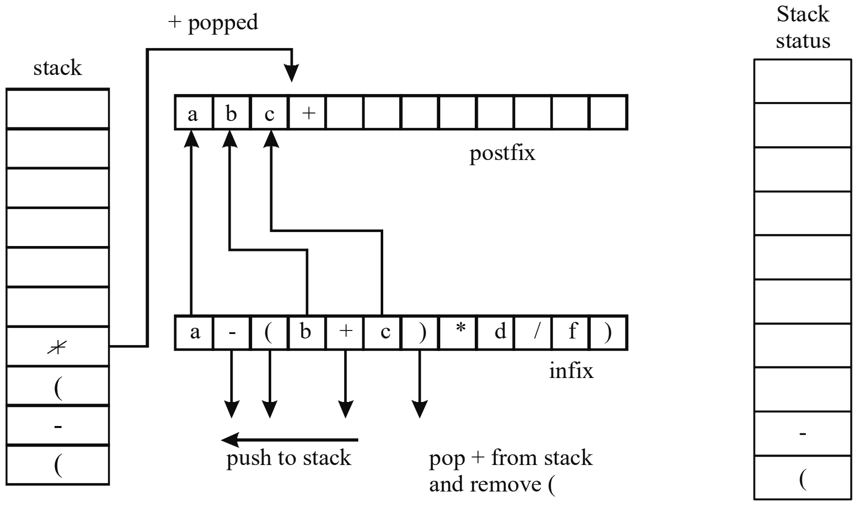


Fig. 5.18 *Initial States of the Stack, Infix Expression and Postfix Expression*

infix is read from left to right and the following steps are performed:

1. The operand a is encountered, which is directly put to postfix.
2. The operator − is pushed onto the stack.
3. The left parenthesis '(' is pushed onto the stack.
4. The next element is b, which being an operand is directly put to postfix.
5. + being an operator is pushed onto the stack.
6. Next, c is put to postfix.
7. The next element is the right parenthesis ')' and hence, the operators on the top of stack are popped until '(' is encountered in stack. Till now, the only operator in the stack above the '(' is +, which is popped and put to postfix. '(' is popped and removed from the stack [Refer Figure 5.19 (a)]. Figure 5.19(b) shows the current position of the stack.



(a) *Postfix Expression when + is popped* **(b)** *State of the Stack*

Fig. 5.19 *Intermediate States of Postfix and Infix Expressions and the Stack*

8. After this, the next element * is an operator and hence, it is pushed onto the stack.
9. Then, d is put to postfix.
10. The next element is /. Since the precedence of / is the same as the precedence of *, the operator * is popped from the stack and / is pushed onto the stack (Refer Figure 5.20).
11. The operand f is directly put to postfix after which ')' is encountered.
12. On reaching ')', the operators in the stack before the next '(' is reached are popped. Hence, / and − are popped and put to postfix as shown in Figure 5.20.

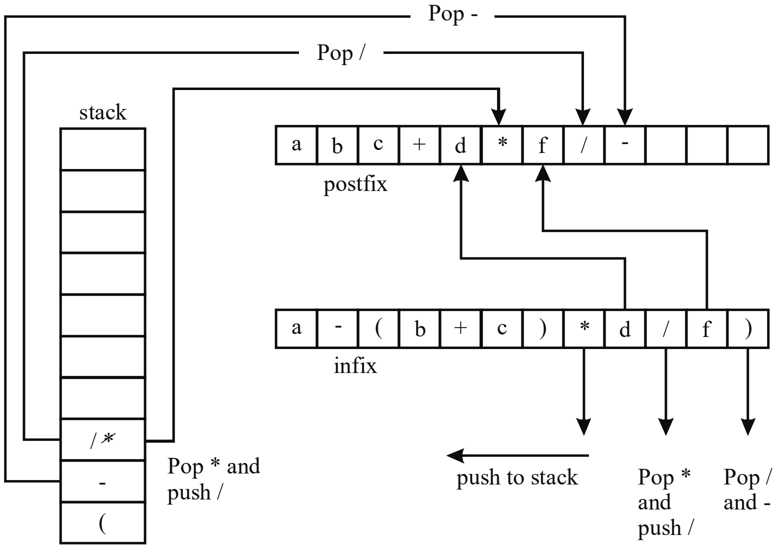


Fig. 5.20 *The State When – and / are Popped*

13. ‘`(`’ is removed from the stack. Since the stack is empty, the algorithm is terminated and `postfix` is printed.

The step-wise conversion of the infix expression `a-(b+c)*d/f` into its equivalent postfix expression is shown in Table 5.3.

Table 5.3 *Conversion of Infix Expression into Postfix Expression*

Element	Action performed	Stack status	Postfix expression
a	Put to postfix	(	A
-	Push	(-	a
(	Push	(-(	a
b	Put to postfix	(-(	ab
+	Push	(-(+	ab
c	Put to postfix	(-(+	abc
)	Pop +, put to postfix, pop (	(-	abc+
*	Push	(-*	abc+
d	Put to postfix	(-*	abc+d
/	Pop *, put to postfix, push /	(-/	abc+d*
f	Put to postfix	(-/	abc+d*f
)	Pop / and -	Empty	abc+d*f/-

Conversion of Infix to Prefix Notation

The conversion of an infix expression to a prefix expression is similar to the conversion of an infix expression to a postfix expression. The only difference is that the expression in the infix notation is scanned in the reverse order, that is, from right to left. Therefore, the stack in this case stores the operators and the closing (right) parenthesis.

Algorithm 5.2 Infix to Prefix Conversion

```
infixtoprefix(s, infix, prefix)

1. Set i = 0
2. While (i < number_of_symbols_in_infix)
       If infix[i] is a whitespace or comma
           Set i = i + 1 go to step 2
       If infix[i] is an operand, add it to prefix
       Else If infix[i] = ')', push it onto the stack
       Else If infix[i] is an operator, follow these steps:
       i. For each operator on the top of stack whose precedence
          greater than or equal to the precedence of the curre
          operator, pop the operator from stack and add it to prefix
       ii. Push the current operator onto the stack
       Else If infix[i] = '(', follow these steps:
       i. Pop each operator from top of the stack and add it to pref
          until ')' is encountered in the stack
       ii.   Remove ')' from the stack and do not add it to prefix
       End If
       Set i = i + 1
   End While
3. Reverse the prefix expression
4. End
```

For example, consider the conversion of the following infix expression to a prefix expression:

    a-(b+c)*d/f

The step-wise conversion of the expression `a-(b+c)*d/f` into its equivalent prefix expression is shown in Table 5.4. Note that initially, '`)`' is pushed onto the stack, and '`(`' is inserted in the beginning of the infix expression. Moreover, since the infix expression is scanned from right to left and the elements are inserted in the resultant expression from left to right, the prefix expression needs to be reversed.

Table 5.4 *Conversion of Infix Expression into Prefix*

Element	Action performed	Stack status	Prefix expression
f	Put to expression	)	f
/	Push	)/	f
d	Put to expression	)/	fd
*	Push	)/*	fd
)	Push	)/*)	fd
c	Put to expression	)/*)	fdc
+	Push	)/*)+	fdc
b	Put to expression	)/*)+	fdcb
(	Pop and + and put to expression, pop	)/*	fdcb+
-	)	)-	fdcb+*/
a	Pop *, / and push -	)/*-	fdcb+a
(	Put to expression	Empty	fdcb+*/a-
	Pop - and put to expression, pop (		-a/*+bcdf
	Reverse the resultant expression		

The equivalent prefix expression is `-a/*+bcdf`.

Evaluation of Postfix Expression

In a computer system, when an arithmetic expression in an infix notation needs to be evaluated, it is first converted into its postfix notation. The equivalent postfix expression is then evaluated. The evaluation of postfix expressions is also implemented through stacks. Since the postfix expression is evaluated in the order of appearance of operators, parentheses are not required in the postfix expression. During the evaluation, a stack is used to store the intermediate results of evaluation.

Since an operator appears after its operands in a postfix expression, the expression is evaluated from left to right. Each element in the expression is checked

NOTES

whether it is an operator or an operand. If the element is an operand, it is pushed onto the stack. On the other hand, if the element is an operator, the first two operands are popped from the stack and the operation is performed on them. The result of the operation is then pushed back to the stack. This process is repeated until the entire expression is evaluated.

Algorithm 5.3 Evaluation of a Postfix Expression

```
evaluationofpostfix(s, postfix)

1. Set i = 0, RES=0.0
2. While (i < number_of_characters_in_postfix)
      If postfix[i] is a whitespace or comma
            Set i = i + 1 and continue
      If postfix[i] is an operand, push it onto the stack
      If postfix[i] is an operator, follow these steps:
         i. Pop the top element from stack and store it in operand2
         ii.      Pop the next top element from stack and store it
            operand1
         iii.     Evaluate operand2 op operand1, and store the resu
            in RES (op is the current operator)
         iv.      Push RES back to stack
      End If
      Set i = i + 1
   End While
3. Pop the top element and store it in RES
4. Return RES
5. End
```

For example, consider the evaluation of the following postfix expression using stacks:

`abc+d*f/-`

Where, `a=6, b=3, c=6, d=5, f=9`

After substituting the values of `a`, `b`, `c`, `d` and `f`, the postfix expression becomes

`636+5*9/-`

The expression is evaluated as follows:

1. The expression is read from left to right and each element is checked to see whether it is an operand or an operator.
2. The first element is 6, which being an operand is pushed onto the stack.
3. Similarly, the operands 3 and 6 are pushed onto the stack.
4. The next element is +, which is an operator. Hence, the element at the top of the stack (6) and the next top element (3) are popped from the stack as shown in Figure 5.21.

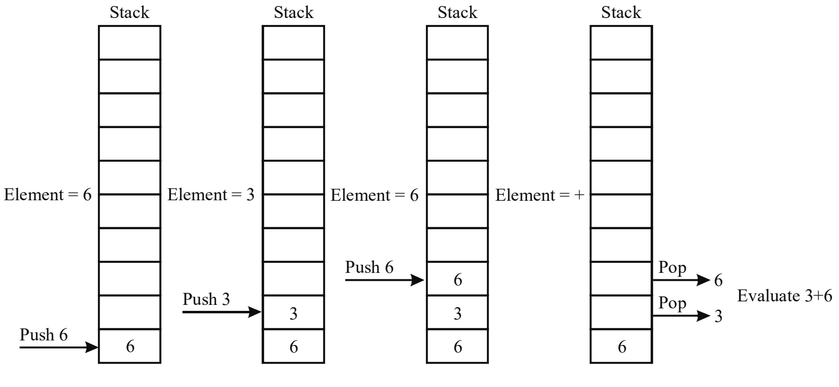


Fig. 5.21 *Evaluation of the Expression using Stacks*

5. The expression `3+6` is evaluated and the result (that is, 9) is pushed back to the stack as shown in Figure 5.22.
6. The next element in the expression, that is `5`, is pushed to the stack.
7. The next element is `*`, which is a binary operator. Hence, the stack is popped twice and the elements `5` and `9` are taken off from the stack as shown in Figure 5.22.

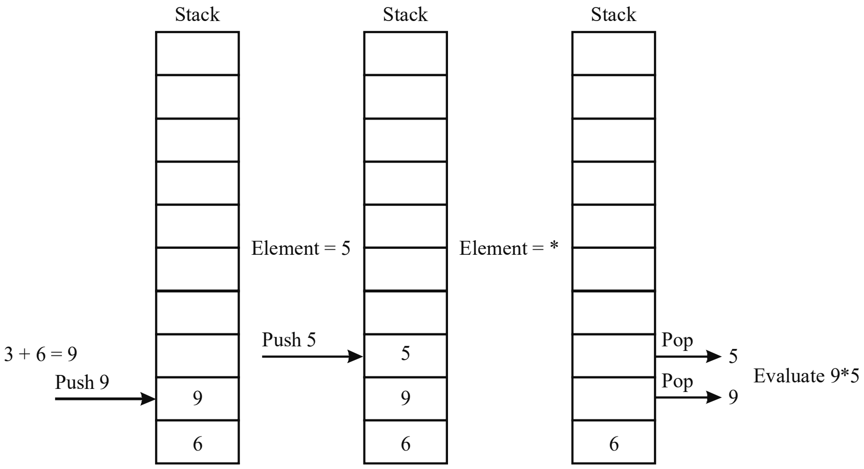


***Fig. 5.22** Popping 9 and 5 from Stack*

8. The expression `9*5` is evaluated and the result, that is, 45, is pushed back to the stack.
9. The next element in the postfix expression is 9, which is pushed onto the stack.
10. The next element is the operator `/`. Therefore, the two operands from the top of the stack, that is, 9 and 45, are popped from the stack, and the operation `45/9` is performed. The result 5 is again pushed to the stack.
11. The next element in the expression is −. Hence, 5 and 6 are popped from the stack and operation `6−5` is performed. The resulting value, that is, 1, is pushed to the stack (Refer Figure 5.23).

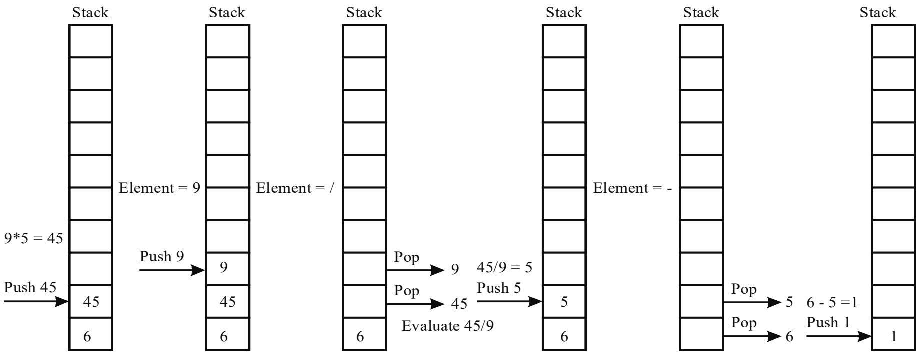


***Fig. 5.23** Final State of Stack with the Result*

12. There are no more elements to be processed in the expression. The element at the top of the stack is popped, which is the result of the evaluation of the postfix expression. Thus, the result of the expression is 1.

The step-wise evaluation of the expression 636+5*9/− is shown in Table 5.5.

Table 5.5 *Evaluation of the Postfix Expression*

Element	Action Performed	Stack Status
6	Push to stack	6
3	Push to stack	6 3
6	Push to stack	6 3 6
+	Pop 6	6 3
	Pop 3	6
	Evaluate 3+6=9	6
	Push 9 to stack	6 9
5	Push to stack	6 9 5
*	Pop 5	6 9
	Pop 9	6
	Evaluate 9*5=45	6
	Push 45 to stack	6 45
9	Push to stack	6 45 9
/	Pop 9	6 45
	Pop 45	6
	Evaluate 45/9=5	6
	Push 5 to stack	6 5
-	Pop 5	6
	Pop 6	EMPTY
	Evaluate 6-5=1	EMPTY
	Push 1 to stack	1
	Pop VALUE=1	EMPTY

Check Your Progress

5. Define regular grammar.
6. Write an application of context free grammars.
7. What is a derivation tree?
8. Define a root.
9. What is infix notation?

5.11 ANSWERS TO 'CHECK YOUR PROGRESS'

1. A language processor is a type of software that bridges a specification or execution gap in software programming.
2. The language generated by a context free grammar is called context free language.
3. Union of languages one of the operators of regular expression.
4. Pumping lemma is used to prove that certain sets are not regular.
5. A regular grammar is a formal grammar that describes a regular language.
6. Context free grammars are used in the implementation of parsers in compiler design.
7. A way of representing the derivation, independent of the order in which the productions are used is the derivation tree.
8. Root is a node that has no parent node.
9. The general way of writing arithmetic expressions is known as the infix notation.

5.12 SUMMARY

- A language processor is a type of software that bridges a specification or execution gap in software programming.
- The programming language comprises of specification, design and coding, Implementation steps.
- An assembler is a language translator whose source language is assembly language.
- A compiler is any language translator that is not an assembler.
- An interpreter is a program that executes a code to display the desired result to a user without generating a machine language program.
- A program generation activity comprises changing a source code into a target program that can be executed to give a desired result.
- The program generator can either be converted to an executable code (maybe a machine code or byte code) by a compiler, or an interpreter can directly use it for execution.
- A language processor pass is the processing of every statement in a source or its equivalent representation to perform a language processing function.
- An alphabet is a finite non-empty set of symbols.
- Binding time is the time at which a binding is performed.
- A static binding is a binding performed before the execution of program begins.
- A dynamic binding is a binding performed after the execution has begun.
- The extended Chomsky hierarchy includes the families of deterministic context-free languages and recursive languages.
- Minimization refers to constructing an automata with minimum number of states to a given automata.
- Context-free grammars are used in the implementation of parsers in compiler design.
- Grammars follow the process of derivation by which the strings are derived from the language of the grammar.
- A derivation tree is an ordered tree in which the nodes are labelled with the left sides of the production and the children of the nodes represent the corresponding right sides.
- A sub tree of a derivation tree T is a tree whose root is some vertex V of T, together with all the descendants of V along with their labels and the edges that connect the descendants of V.
- A parse tree is a representation of the structure and grammar of a computer language.
- The process of parsing helps to translate human understandable grammar to a computer understandable code.

NOTES

- The elements of a parse tree are called nodes.
- Root is a node that has no parent node.
- A parse tree is a graphical depiction for a source that filters out the options about the substitution order.
- In top-down parsing, the derivation is for the leftmost input string.
- A grammar that produces more than one parse tree for some sentence is said to be ambiguous.
- The general way of writing arithmetic expression is known as the infix notation.
- The conversion of an infix expression to a prefix expression is similar to the conversion of an infix expression to a postfix expression.

5.13 KEY TERMS

- **Language processor:** It is a type of software that bridges a specification or execution gap in software programming.
- **Intermediate representation:** It is a representation of a source program that reflects the effect of some, but not all, analysis and synthesis tasks performed during language processing.
- **Ambiguity of grammar:** A grammar that produces more than one parse tree for some sentence is said to be ambiguous.
- **Static binding:** A static binding is a binding performed before the execution of program begins.
- **Empty string or mull string:** An empty string or null string is defined as the string with zero occurrences of symbols.
- **Parsing:** A parse tree is a representation of the structure and grammar of a computer language.
- **Prefix notation:** The notation in which an operator occurs before its operands is known as the prefix notation.

5.14 QUESTIONS AND EXERCISES

Short-Answer Questions

1. What are the two language processing activities?
2. Define phrase structure grammar.
3. Define regular expression.
4. What is pumping lemma?
5. Define left linear grammar.
6. How can we represent context free languages?
7. Define partial derivation tree.
8. What are sentential forms?
9. Define postfix notation.

Long-Answer Questions

1. Explain the concept of languages and grammar.
2. Illustrate different types of phrase structure grammars.
3. Discuss about the operators of regular expression.
4. State and prove pumping lemma for regular languages.
5. Describe the concept of regular grammar.
6. Write a note on language generated by grammar.
7. Discuss briefly about derivation trees.
8. Explain the concept of parsing.
9. Illustrate notations of syntax analysis.

NOTES

5.15 FURTHER READING

Iyengar, N Ch S N. V M Chandrasekaran, K A Venkatesh and P S Arunachalam. *Discrete Mathematics*. New Delhi: Vikas Publishing House Pvt. Ltd., 2007.

Tremblay, Jean Paul and R. Manohar. *Discrete Mathematical Structures with Applications to Computer Science*. New York: McGraw-Hill Inc., 1975.

Deo, Narsingh. *Graph Theory with Applications to Engineering and Computer Science*. New Delhi: Prentice-Hall of India, 1999.

Singh, Y.N. *Mathematical Foundation of Computer Science*. New Delhi: New Age International Pvt. Ltd., 2005.

Malik, D.S. *Discrete Mathematical Structures: Theory and Applications*. London: Thomson Learning, 2004.

Haggard, Gary, John Schlipf and Sue Whiteside. *Discrete Mathematics for Computer Science*. California: Thomson Learning, 2006.

Cohen, Daniel I.A. *Introduction to Computer Theory*, 2nd edition. New Jersey: John Wiley and Sons, 1996.

Hopcroft, J.E., Rajeev Motwani and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*, 3rd edition. Boston: Addison-Wesley, 2006.

Linz, Peter. *An Introduction to Formal Languages and Automata*, 5th edition. Boston: Jones and Bartlett Publishers, 2011.

Mano, M. Morris. *Digital Logic and Computer Design*. New Jersey: Prentice-Hall, 1979.

NOTES

NOTES

NOTES