COURSE FILE

ON

AUTOMATA THEORY AND COMPILER DESIGN

CourseCode-22CS427PC

II B.Tech II-SEMESTER

` **A.Y.:2024-2025**

ISO 9001:2015 Certified Institution Estd.:2001

Balaji Institute of Technology & Science

Laknepally (V), Narsampet (M), Warangal District - 506 331, Telangana State, India

(AUTONOMOUS)

Accredited by NBA (UG - CE, ME, ECE & CSE) & NAAC A+ Grade

(Affiliated to JNT University, Hyderabad and Approved by AICTE, New Delhi)

BITS AUTONOMOUS www.bitswgl.ac.in, email: principal@bitswgl.ac.in, Ph:98660 50044, Fax: 08718-230521

DEPARTMENT OF COMPUTER ENGINEERING (SE)

Course File Contents:

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING (AI&ML)

DEPARTMENT VISION AND MISSION

VISION

To be a global leader in Artificial Intelligence and Machine Learning research, innovation, and education, driving transformative advancements that empower industries, enhance human capabilities, and contribute to a smarter, more sustainable world.

MISSION

M1:Innovative Research& Quality Education – To Conduct research on cutting-edge Technologies to address complex real-world problems across diverse domains and provide world-class education and training to equip students with technical expertise, ethical responsibility, and problem-solving skills.

M2: Industry Collaboration & Ethical AI Development –To Foster strong partnerships with industries, academia, and government organizations to develop impactful AI solutions and promote responsible and ethical AI practices that align with societal values and global sustainability.

M3: Entrepreneurship & Innovation – Encourage entrepreneurship and the development of AI-driven start-ups and products that contribute to economic growth.

M4: Community Engagement – Engage with communities to spread AI awareness, inclusivity, and accessibility for societal benefit.

ISO 9001:2015 Certified Institution Estd.:2001

Balaji Institute of Technology & Science

Laknepally (V), Narsampet (M), Warangal District - 506 331, Telangana State, India
(AUTONOMOUS)
Accredited by NBA (UG - CE, ME, ECE & CSE) & NAAC A+ Grade
(Affiliated to JNT University, Hyderabad and Approved by AICTE, New Delhi)
www.bitswgl.ac.in, email: principal@bitswgl.ac.in, Ph:98660 50044, Fax: 08718-230521

BITS
AUTONOMOUS

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING (AI&ML)

Programs Educational Objectives (PEOs)

PEO1: To equip graduates with a robust foundation in AI, ML, and related computational techniques, enabling them to develop and implement intelligent systems across multiple domains.

PEO2: To empower graduates to conduct advanced research, drive innovations in AI and ML, and create transformative solutions for complex real-world challenges.

PEO3: To prepare the graduates to equip with the skills and adaptability to thrive in dynamic industrial environments and pursue continuous learning to stay ahead in emerging AI technologies.

Programs Specific Outcomes (PSOs)

PSO1: Graduates will be able to design, develop, and implement AI and ML-based solutions using modern tools, frameworks, and methodologies.

PSO2: Graduates will be able to analyze, pre-process, and interpret large-scale data, applying statistical and machine learning techniques to derive meaningful insights and solve real-world problems.

PSO3: Graduates will develop expertise in deep learning, computer vision, natural language processing, and reinforcement learning to create innovative AI applications across multiple domains.

ISO 9001:2015 Certified Institution Estd.:2001

Balaji Institute of Technology & Science

Laknepally (V), Narsampet (M), Warangal District - 506 331, Telangana State, India

(AUTONOMOUS)

Accredited by NBA (UG - CE, ME, ECE & CSE) & NAAC A+ Grade

(Affiliated to JNT University, Hyderabad and Approved by AICTE, New Delhi)

www.bitswgl.ac.in, email: principal@bitswgl.ac.in, Ph:98660 50044, Fax: 08718-230521

BITS AUTONOMOUS

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING (AI&ML)

ROGRAMME OUTCOMES (POs)

A graduate of the Software Engineering Program will demonstrate.

- **Engineering knowledge**: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.

- **Problem analysis**: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, Natural sciences and engineering sciences.

- **Design/development of solutions**: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.

- **Conduct investigations of complex problems**: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

- **Modern tool usage**: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.

- **The engineer and society**: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

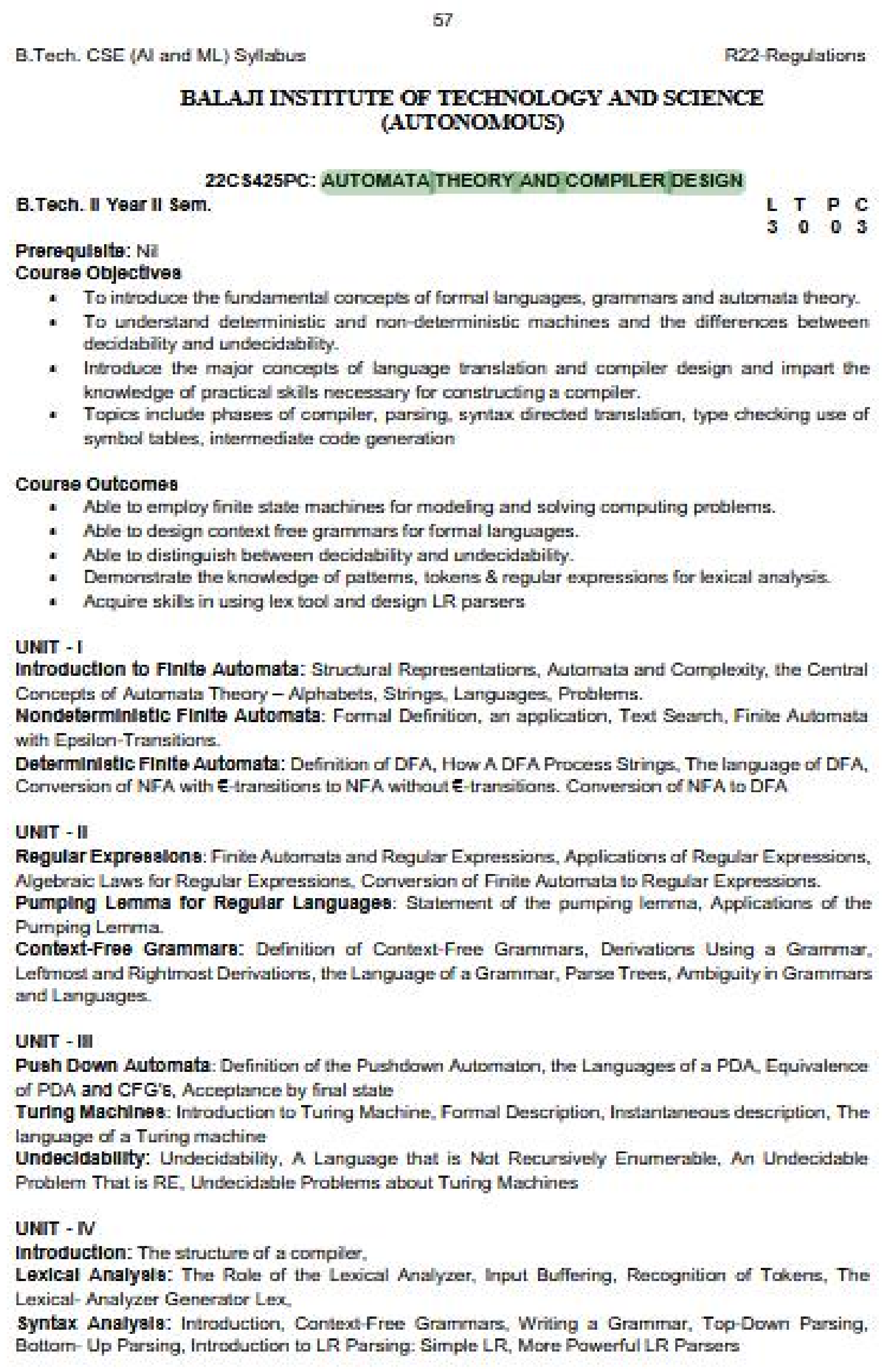
  - **Environment and sustainability**: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
  - **Ethics**: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
  - **Individual and team work**: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
  - **Communication**: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

- **Project management and finance**: Demonstrate knowledge and understanding of the

  engineering and management principles and apply these to one’s own work, as a member and leader in a team, to manage projects and in multidisciplinary environments to manage projects.

  Life-long learning: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

SYLLABUS COPY



57

B.Tech. CSE (AI and ML) Syllabus R22-Regulations

BALAJI INSTITUTE OF TECHNOLOGY AND SCIENCE (AUTONOMOUS)

22CS425PC: AUTOMATA THEORY AND COMPILER DESIGN

B.Tech. II Year II Sem.

L	T	P	C
3	0	0	3

Prerequisite: Nil

Course Objectives

- To introduce the fundamental concepts of formal languages, grammars and automata theory.
- To understand deterministic and non-deterministic machines and the differences between decidability and undecidability.
- Introduce the major concepts of language translation and compiler design and impart the knowledge of practical skills necessary for constructing a compiler.
- Topics include phases of compiler, parsing, syntax directed translation, type checking use of symbol tables, intermediate code generation

Course Outcomes

- Able to employ finite state machines for modeling and solving computing problems.
- Able to design context free grammars for formal languages.
- Able to distinguish between decidability and undecidability.
- Demonstrate the knowledge of patterns, tokens & regular expressions for lexical analysis.
- Acquire skills in using lex tool and design LR parsers

UNIT - I

Introduction to Finite Automata: Structural Representations, Automata and Complexity, the Central Concepts of Automata Theory – Alphabets, Strings, Languages, Problems.

Nondeterministic Finite Automata: Formal Definition, an application, Text Search, Finite Automata with Epsilon-Transitions.

Deterministic Finite Automata: Definition of DFA, How A DFA Process Strings, The language of DFA, Conversion of NFA with €-transitions to NFA without €-transitions. Conversion of NFA to DFA

UNIT - II

Regular Expressions: Finite Automata and Regular Expressions, Applications of Regular Expressions, Algebraic Laws for Regular Expressions, Conversion of Finite Automata to Regular Expressions.

Pumping Lemma for Regular Languages: Statement of the pumping lemma, Applications of the Pumping Lemma.

Context-Free Grammars: Definition of Context-Free Grammars, Derivations Using a Grammar, Leftmost and Rightmost Derivations, the Language of a Grammar, Parse Trees, Ambiguity in Grammars and Languages.

UNIT - III

Push Down Automata: Definition of the Pushdown Automaton, the Languages of a PDA, Equivalence of PDA and CFG's, Acceptance by final state

Turing Machines: Introduction to Turing Machine, Formal Description, Instantaneous description, The language of a Turing machine

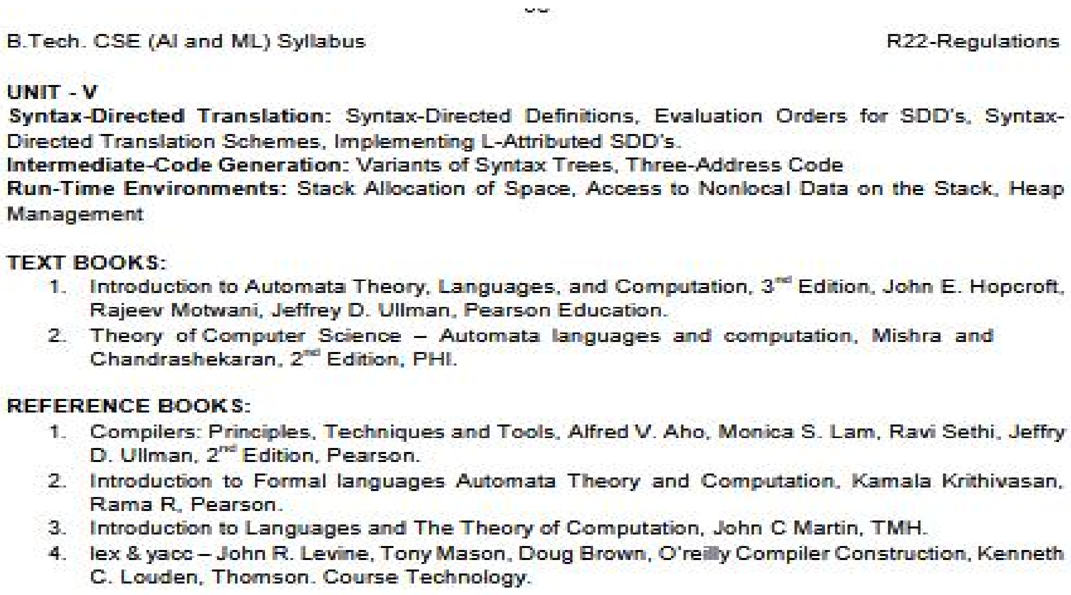
Undecidability: Undecidability, A Language that is Not Recursively Enumerable, An Undecidable Problem That is RE, Undecidable Problems about Turing Machines

UNIT - IV

Introduction: The structure of a compiler,

Lexical Analysis: The Role of the Lexical Analyzer, Input Buffering, Recognition of Tokens, The Lexical- Analyzer Generator Lex,

Syntax Analysis: Introduction, Context-Free Grammars, Writing a Grammar, Top-Down Parsing, Bottom- Up Parsing, Introduction to LR Parsing: Simple LR, More Powerful LR Parsers



UNIT - V

Syntax-Directed Translation: Syntax-Directed Definitions, Evaluation Orders for SDD's, Syntax-Directed Translation Schemes, Implementing L-Attributed SDD's.
Intermediate-Code Generation: Variants of Syntax Trees, Three-Address Code
Run-Time Environments: Stack Allocation of Space, Access to Nonlocal Data on the Stack, Heap Management

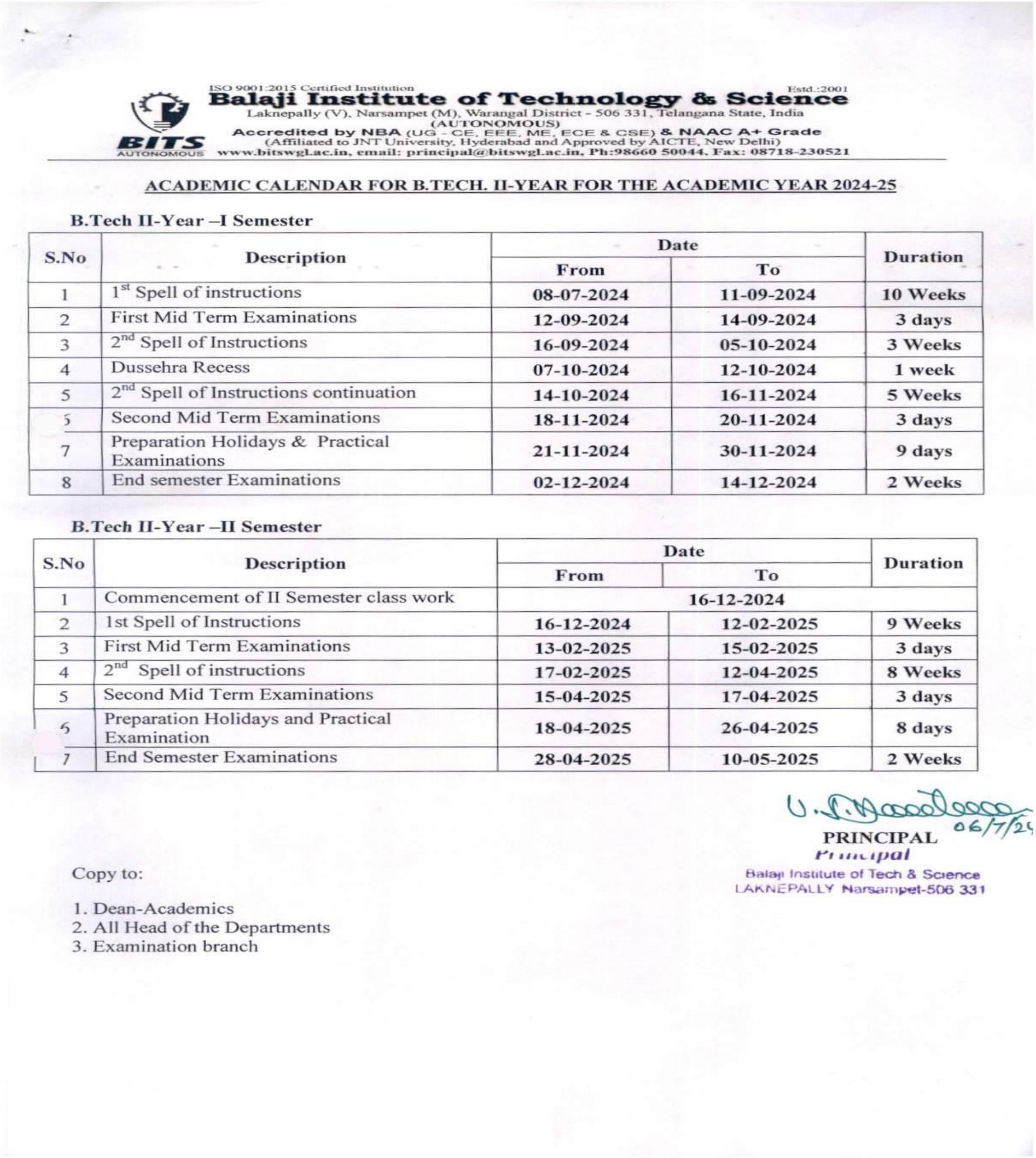
TEXT BOOKS:

1. Introduction to Automata Theory, Languages, and Computation, 3rd Edition, John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman, Pearson Education.
2. Theory of Computer Science – Automata languages and computation, Mishra and Chandrashekaran, 2nd Edition, PHI.

REFERENCE BOOKS:

1. Compilers: Principles, Techniques and Tools, Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffry D. Ullman, 2nd Edition, Pearson.
2. Introduction to Formal languages Automata Theory and Computation, Kamala Krithivasan, Rama R, Pearson.
3. Introduction to Languages and The Theory of Computation, John C Martin, TMH.
4. lex & yacc – John R. Levine, Tony Mason, Doug Brown, O'reilly Compiler Construction, Kenneth C. Louden, Thomson. Course Technology.

ACADEMIC CALENDER



ISO 9001:2015 Certified Institution

Estd.:2001

Balaji Institute of Technology & Science

Laknepally (V), Narsampet (M), Warangal District - 506 331, Telangana State, India

(AUTONOMOUS)

Accredited by NBA (UG - CE, EEE, ME, ECE & CSE) **& NAAC A+ Grade**

(Affiliated to JNT University, Hyderabad and Approved by AICTE, New Delhi)

BITS
AUTONOMOUS

www.bitswgl.ac.in, email: principal@bitswgl.ac.in, Ph:98660 50044, Fax: 08718-230521

ACADEMIC CALENDAR FOR B.TECH. II-YEAR FOR THE ACADEMIC YEAR 2024-25

B.Tech II-Year –I Semester

S.No	Description	Date		Duration
		From	To	
1	1st Spell of instructions	08-07-2024	11-09-2024	10 Weeks
2	First Mid Term Examinations	12-09-2024	14-09-2024	3 days
3	2nd Spell of Instructions	16-09-2024	05-10-2024	3 Weeks
4	Dussehra Recess	07-10-2024	12-10-2024	1 week
5	2nd Spell of Instructions continuation	14-10-2024	16-11-2024	5 Weeks
6	Second Mid Term Examinations	18-11-2024	20-11-2024	3 days
7	Preparation Holidays & Practical Examinations	21-11-2024	30-11-2024	9 days
8	End semester Examinations	02-12-2024	14-12-2024	2 Weeks

B.Tech II-Year –II Semester

S.No	Description	Date		Duration
		From	To	
1	Commencement of II Semester class work	16-12-2024		
2	1st Spell of Instructions	16-12-2024	12-02-2025	9 Weeks
3	First Mid Term Examinations	13-02-2025	15-02-2025	3 days
4	2nd Spell of instructions	17-02-2025	12-04-2025	8 Weeks
5	Second Mid Term Examinations	15-04-2025	17-04-2025	3 days
6	Preparation Holidays and Practical Examination	18-04-2025	26-04-2025	8 days
7	End Semester Examinations	28-04-2025	10-05-2025	2 Weeks

06/7/2

PRINCIPAL

Principal
Balaji Institute of Tech & Science
LAKNEPALLY Narsampet-506 331

Copy to:

1. Dean-Academics
2. All Head of the Departments
3. Examination branch

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING (AI&ML)

Importance of the course

- An **automaton** is a construct that possesses all the indispensable features of a digital computer.
- It accepts input, produces output, may have some temporary storage and can make decisions in transforming the input into the output.
- A **formal language** is an abstraction of the general characteristics o programming languages.
- Aformallanguageconsistsofasetofsymbolsandsomerulesofformationbywhichthese symbols can be combined into entities called sentences.

PRE-REQUISITES:

Mathematical Logic

Set Theory

Discrete Mathematics

Basic Concepts in Computation

Theory of Languages

Course Objectives

- To present the theory of finite automata as the first step towards learning advanced topics such as compiler design.
- To discuss the applications of finite automata towards text processing.
- To develop an understanding of Regular expressions and context free grammars and how these concepts are used in lexical analyzer
- To develop an understanding of finite automata through Turing machines.

Course Outcomes

After completing this course the student will be able to:

C213.1 Design finite automata without output like DFA, NFA, €-NFA and finite automata with output like Moore and mealy machines and also conversions among them like (NFA to DFA). (Synthesis)

C213.2 Recognize about regular expressions, pumping lemma for regular languages and closure properties of regular languages. (Knowledge)

C213.3 Define CFG, derivations (Leftmost &Rightmost)and draw parse trees and gain Knowledge on Ambiguity in Grammars. (Knowledge)

C213.4 Define and design a PDA for a given CFL. Prove the equivalence of CFG and PDA and their inter-conversions. (Knowledge)

C213.5 Illustrate CFG normal forms, Use pumping lemma to prove that a language is not a CFL and Define and design TM for a given computation. (Comprehension)

C213.6 Differentiate between decidability and undecidability ,Generalize Turing Machines into universal TMs (Analysis)

Mapping of course outcomes with program outcomes:

High-3 Medium-2 Low-1

PO/PSO /CO	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12	PSO1	PSO2
C213.1	2	1	2	-	-	-	-	-	-	-	-	-	-	-
C213.2	2	-	1	-	1	-	-	-	-	-	-	-	2	-
C213.3	2	1	2	-	1	-	-	-	-	-	-	-	2	-
C214.4	2	-	-	-	-	-	-	-	-	-	-	-	-	-
C213.5	2	1	2	-	-	-	-	-	-	-	-	-	-	-
C213.6	2	1	-	-	-	-	-	-	-	-	-	-	-	-
C213	2	1	1.75	-	1	-	-	-	-	-	-	-	2	-

CO–PO /PSO Mapping Justification

Course:Formal Languages and Automata Theory

PROGRAMME OUTCOMES(POs):

PO1 **Engineering knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.

PO2 **Problemanalysis:**Identify, formulate,reviewresearchliterature,andanalysecomplex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.

PO3 **Design/developmentofsolutions:**Designsolutionsforcomplexengineeringproblems and design system components or processes that meet the specified needs with appropriateconsiderationforthepublichealthandsafety,andthecultural,societal,and environmentalconsiderations.

PO5 **Moderntoolusage:**Create,select,andapplyappropriatetechniques, resources,and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.

PROGRAM SPECIFICOUTCOMES (PSOs):

PSO1 Professional Skills: The ability to implementcomputer programs of varying complexity In the areas related to web design, cloud computing and networking.

C213.1 DesignfiniteautomatawithoutoutputlikeDFA, NFA,€-NFAandfiniteautomata

WithOutputlikeMooreandmealymachinesandalso conversionsamongthemlike (NFA to DFA). (Synthesis)

	Justification
PO1	Gainknowledgeonfinite automata.(level2)
PO2	Analyseproblemandaccordinglyconstructfiniteautomata.(level1)
PO3	Designsolutionsforengineeringproblemsanddesignsystemcomponentsusingfinite automata.(level2)

C213.2 Recognizeaboutregularexpressions,pumpinglemmaforregularlanguagesand Closure properties of regular languages. (Knowledge)

	Justification
PO1	Gainknowledgeonregularexpressions.(level2)
PO3	Useregularexpressionsconceptinpatternmatching.(level1)
PO5	Tocreatelexprogramsuseregular expressions.(level1)
PSO1	InWebdesigning,fortextsearchinguseregularexpressions.(level2)

C213.3 DefineCFG,derivations(Leftmost &Rightmost)anddrawparsetreesandgain Knowledge on Ambiguity in Grammars. (Knowledge)

	Justification
PO1	GainknowledgeonCFG,derivationsandparsetrees(level2)
PO2	AnalyseproblemandaccordinglyconstructCFG. (level1)
PO3	UseCFGindesignofparsersincompilerdesignandXML.(level2)
PO5	TocreateYACC(parsers) useCFG.(level1)
PSO1	Incompiler design(Parsers),webdesigning(XML,DTD)useCFG.(level2)

C213.4 Define and designa PDA for a givenCFL. Prove the equivalence ofCFG and PDAand their inter-conversions. (Knowledge).

	Justification
PO1	Gainknowledge onpushdownautomata(level2)

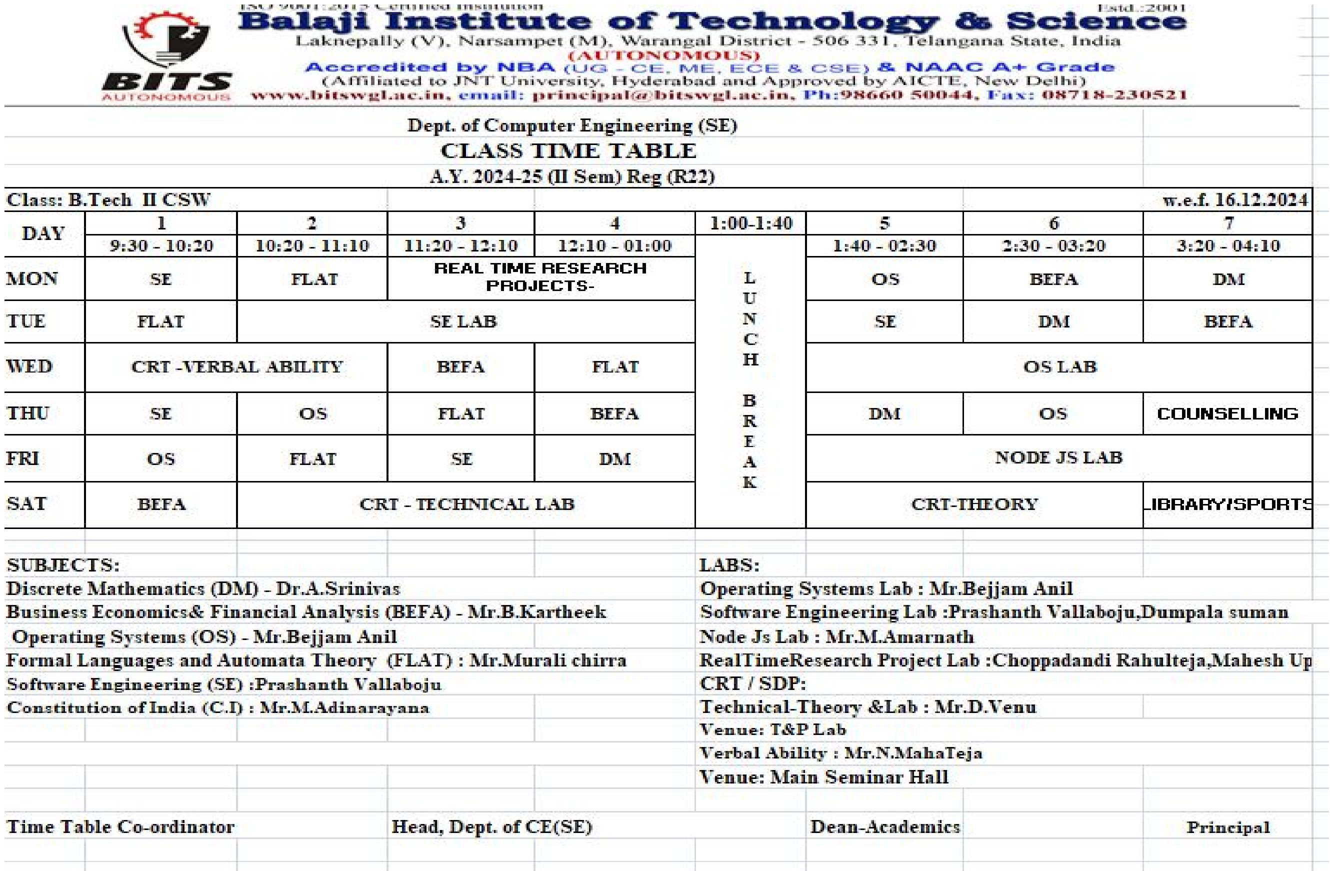
C213.5 IllustrateCFGnormalforms,Usepumping lemmatoprovethatalanguageis notaCFL and Define and design TM fora given computation. (Comprehension)

	Justification
PO1	Gain knowledge on CFG normal formsand Turing machines.(level2)
PO2	Analyse problem and accordingly construct Turing machine(level1)
PO3	Design solutions for engineering problems usingTuring machine(level2)

C213.6 Differentiate between decidability and undesirability,GeneralizeTuring MachinesintouniversalTMs(Analysis)

	Justification
PO1	Gainknowledgeondecidability,undecidability, universalTMandpost correspondence problem(level2)
PO2	Analyseproblemandsolveit.(level1)

CLASS TIME TABLE



ISO 9001:2015 Certified Institution Estd.:2001

Balaji Institute of Technology & Science

Laknepally (V), Narsampet (M), Warangal District - 506 331, Telangana State, India

(AUTONOMOUS)

Accredited by NBA (UG - CE, ME, ECE & CSE) & NAAC A+ Grade

(Affiliated to JNT University, Hyderabad and Approved by AICTE, New Delhi)

www.bitswgl.ac.in, email: principal@bitswgl.ac.in, Ph:98660 50044, Fax: 08718-230521

BITS AUTONOMOUS

Dept. of Computer Engineering (SE)

CLASS TIME TABLE

A.Y. 2024-25 (II Sem) Reg (R22)

Class: B.Tech II CSW w.e.f. 16.12.2024

DAY	1	2	3	4	1:00-1:40	5	6	7
	9:30 - 10:20	10:20 - 11:10	11:20 - 12:10	12:10 - 01:00		1:40 - 02:30	2:30 - 03:20	3:20 - 04:10
MON	SE	FLAT	REAL TIME RESEARCH PROJECTS-		L U N C H B R E A K	OS	BEFA	DM
TUE	FLAT	SE LAB				SE	DM	BEFA
WED	CRT -VERBAL ABILITY		BEFA	FLAT		OS LAB		
THU	SE	OS	FLAT	BEFA		DM	OS	COUNSELLING
FRI	OS	FLAT	SE	DM		NODE JS LAB		
SAT	BEFA	CRT - TECHNICAL LAB				CRT-THEORY		LIBRARY/SPORTS

SUBJECTS:

Discrete Mathematics (DM) - Dr.A.Srinivas

Business Economics& Financial Analysis (BEFA) - Mr.B.Kartheek

Operating Systems (OS) - Mr.Bejjam Anil

Formal Languages and Automata Theory (FLAT) : Mr.Murali chirra

Software Engineering (SE) :Prashanth Vallaboju

Constitution of India (C.I) : Mr.M.Adinarayana

LABS:

Operating Systems Lab : Mr.Bejjam Anil

Software Engineering Lab :Prashanth Vallaboju,Dumpala suman

Node Js Lab : Mr.M.Amarnath

RealTimeResearch Project Lab :Choppadandi Rahulteja,Mahesh Up

CRT / SDP:

Technical-Theory &Lab : Mr.D.Venu

Venue: T&P Lab

Verbal Ability : Mr.N.MahaTeja

Venue: Main Seminar Hall

Time Table Co-ordinator Head, Dept. of CE(SE) Dean-Academics Principal

Personal time table

Mrs.M.Vedavani
TOTAL-14

DAY	1	2	3	4		5	6	7
	9:30 - 10:20	10:20 - 11:10	11:20 - 12:10	12:10 - 01:00	LUNCHBR EA	1:40 - 02:30	2:30 - 03:20	3:20 - 04:10
MON						IICSM-ATCD		
TUE	IIICSM-ML LAB							
WED	IICSM-ATCD							
THU								
FRI		IIICSM-CN LAB					IICSM-ATCDF	
SAT	IICSW-ATCD	IICSM-DS LAB						

DEPARTMENT OF COMPUTER ENGINEERING (SE)

Method of teaching, Chalk and talk/ppts/NPTEL lectures/cd/innovative teaching method, etc.

1. Chalk and Talk (Traditional Method)

- **Pros**: Simple, direct interaction with students, flexible for impromptu explanations, and allows for personalization of teaching pace.
- **How to Improve It for Flat Subjects**:
    - **Use Visuals**: Draw diagrams and flowcharts to illustrate concepts like state diagrams for automata, parsing trees for grammars, etc.
    - **Relate to Real-World Applications**: Try to link abstract concepts to real-life examples or simple computing problems, such as search engines (regular expressions) or programming language compilers (context-free grammars).
    - **Interactive Discussions**: Engage students by asking questions or encouraging them to explain concepts as they are learning.

2. PowerPoint Presentations (PPTs)

- **Pros**: Can include diagrams, bullet points, videos, and other visuals that make abstract concepts clearer.
- **How to Improve It**:
    - **Clear Visuals**: Use animations to show how automata change states, or how a string is parsed by a context-free grammar.
    - **Step-by-Step Breakdown**: Break complex problems into simple steps. For instance, show the process of evaluating a regular expression using a DFA or constructing a CFG.
    - **Interactive Slides**: Include quiz questions or polls during the presentation to check comprehension (e.g., "What happens when this NFA receives input X?").

3. NPTEL (National Programme on Technology Enhanced Learning) Lectures

- **Pros**: High-quality, well-structured content from experts in the field. Self-paced learning is possible.
- **How to Improve It**:
    - **Flipped Classroom Approach**: Assign NPTEL lectures as homework and then spend class time discussing the most challenging concepts from those lectures.
    - **Active Discussion After Viewing**: After watching NPTEL videos, have an in-class discussion or Q&A session to clear doubts.

    - **Supplementary Exercises**: Use practice problems, coding exercises, or simulation tools related to the NPTEL content to enhance learning.

4. Innovative Teaching Methods

- **Gamification**: Create challenges or games that involve solving automata problems or language problems. For example, students could "race" against time to design a DFA or NFA for a given language.
- **Hands-on Software Tools**:
    - **JFLAP**: A great tool for simulating automata, Turing machines, and grammars. Have students experiment with designing automata or proving languages are regular or context-free using JFLAP.
    - **Automata-based Programming**: Use coding assignments where students implement automata algorithms or grammars in their favorite programming language (e.g., writing a program to simulate a DFA or NFA).
- **Role Play**: For complex concepts, have students "become" the automata, grammars, or machines, physically walking through transitions to help visualize concepts like state changes in a DFA or parsing a string using a CFG.
- **Concept Maps**: Encourage students to create mind maps or concept maps for topics like finite automata, regular expressions, context-free grammars, etc., which show the connections between different concepts.

5. Flipped Classroom

- **How It Works**: The idea is that students learn the basic concepts outside of class (via readings, videos, or online resources like NPTEL), and class time is used for active problem-solving and discussions.
- **How to Apply**:
    - Provide short introductory videos or readings on the topic of the day.
    - In class, engage students in solving problems related to the topic, discuss real-world applications, and troubleshoot any confusions.
    - Incorporate peer discussions and group work to help students collaborate on complex problems.

6. Interactive Online Learning Platforms

- **Tools**: Platforms like **Khan Academy**, **Coursera**, **Udacity**, or **edX** can provide online resources, interactive exercises, and quizzes that allow students to learn at their own pace.
- **Benefits**: Students can revisit tough topics, complete interactive quizzes, and engage with a variety of resources such as animations, simulations, and hands-on coding exercises.

7. Project-Based Learning

- **Approach**: Assign a project that requires students to apply what they've learned to real-world problems, such as creating a simple compiler or developing a software tool that recognizes regular languages or simulates a Turing machine.
- **How to Implement**:
    - Divide students into teams and assign them a problem or project that involves multiple concepts (e.g., developing a finite automaton to recognize a specific language).
    - Provide periodic feedback and encourage collaboration.
    - Have students present their projects at the end of the semester, explaining their approach and solutions.

8. Collaborative Learning (Peer Learning)

- **Group Work**: Organize students into groups to discuss difficult topics (e.g., design an automaton for a particular language or prove a language is context-free).
- **Peer Teaching**: Encourage students who grasp concepts faster to explain them to their peers in simple terms.
- **Study Groups**: Organize informal study groups where students can work together to solve problems, learn from each other, and get support from the teacher when necessary.

9. Use of Animation/Visualization Tools

- **Simulations of Automata**: Tools like **JFLAP** or web-based simulators can show state transitions in real-time, which helps students visualize the concepts they are learning.
- **Automata in Action**: Use animations or video clips that demonstrate how finite state machines process input strings, or how context-free grammars generate languages.

10. Incorporating Coding

- **Code along**: Students can write code to implement finite automata, Turing machines, or parsers in various programming languages (e.g., Python, Java, or C++).
- **Project Examples**: Have them write simple regex parsers, or create a language recognizer using finite automata, to give practical exposure to theoretical concepts.

ISO 9001:2015 Certified Institution Estd. : 2001

Balaji Institute of Technology & Science

Laknepally, NARSAMPET, Warangal (Rural) – 506331

Accredited By NBA (UG – CE, ME, ECE & CSE Programmes) & NAAC

(Affiliated to JNTUH, Hyderabad and Approved by the AICTE, New Delhi)

BITS NARSAMPET www.bitswgl.ac.in, email: principal@bitswgl.ac.in :: Ph. 98660 50044, Fax 08718-230521

Department of Computer Science & Engineering

LESSON PLAN & DELIVERY REPORT

Subject: AUTOMATA THEORY AND COMPILER DESIGN *[*CS305PC*]* ***Class***: *B.Tech II CSM*

Regulation: *R22*

Academic Year: *2024-25 (II-Sem)* ***Commencement of Class Work***: *16-12-24*

UNIT I Introduction to Finite Automata, DFA,NFA (No. of Lectures –12)				
Topics (as per syllabus)	**Sub Topics**	**Lect. No.**	**Scheduled Date**	**Topic Delivered Date**
	• About Subject & Guidelines • Vision, Mission, CO's of subject • Text & Reference Books	L1	19.12.24	
Introduction to Finite Automata	• Introduction to Finite Automata:	L2	20.12.24	
	• Automata and Complexity	L3	23.12.24	
	• Central Concepts of Automata Theory – Alphabets	L4	24.12.24	
	• Strings, Languages, Problems	L5	27.12.24	
Nondeterministic Finite Automata	• Nondeterministic Finite Automata	L6	30.12.24	
	• Formal Definition, an application, Text Search	L7	31.12.24	
	• Finite Automata with Epsilon-Transitions	L8	02.01.25	
Deterministic Finite Automata	• Definition of DFA, How A DFA Process Strings,	L9	03.01.25	
	• The language of DFA, Conversion of NFA with €-transitions to NFA without €-transitions	L10	06.01.25	
	• Conversion of NFA to DFA	L11	07.01.25	
Test	• Slip Test	L12	08.01.25	

Topics (as per syllabus)	**Sub Topics**	**Lect. No.**	09.01.25	**Topic Delivered Date**
UNIT II: RE, Pumping Lemma for Regular Languages, Context-Free Grammars **(No. of Lectures – 12)**				
Regular Expressions	• Finite Automata and Regular Expressions	L13	17.01.25	
	• Applications of Regular Expressions	L14	20.01.25	
	• Algebraic Laws for Regular Expressions	L15	21.01.25	
	• Conversion of Finite Automata to Regular Expressions	L16	22.01.25	
Pumping Lemma for Regular Languages	• Statement of the pumping lemma	L17	23.01.25	
	• Applications of the Pumping Lemma	L18	27.01.25	
Context-Free Grammars	• Definition of Context-Free Grammars	L19	28.01.25	
	• Derivations Using a Grammar, Leftmost and Rightmost Derivations	L20	29.01.25	
	• The Language of a Grammar	L21	30.01.25	
	• Parse Trees	L22	31.01.25	
	• Ambiguity in Grammars and Languages	L23	03.02.25	
Test	• Slip Test	L24	04.02.25	
UNIT – III PDA, TM, Undecidability: (No. of Lectures – 12)				
Push Down Automata	• Push Down Automata: Definition of the Pushdown Automaton,	L25	06.02.25	
	• the Languages of a PDA, Equivalence of PDA and CFG"s	L26	07.02.25	
	• Acceptance by final state	L27	08.02.25	
Turing Machines	• Introduction to Turing Machine, Formal Description	L28	10.02.25	
	• Instantaneous description	L29	11.02.25	
Mid I Schedule:		**ATCD**		**Mid I Exam**

Turing Machines:	• The language of a Turing machine	L30	**Mid I Exam (ATCD) :**	
Undecidability	• Undecidability	L31	18.02.25	
Mid I Marks Distribution	• Marks Distribution • Discussion about Paper • Counsel the students (AB/got poor marks)	L32	19.02.25	
Undecidability	• A Language that is Not Recursively Enumerable,	L33	21.02.25	
	• An Undecidable Problem That is RE,	L34	24.02.25	
	• Undecidable Problems about Turing Machines	L35	25.02.25	
	• Slip Test	L36	27.02.25	

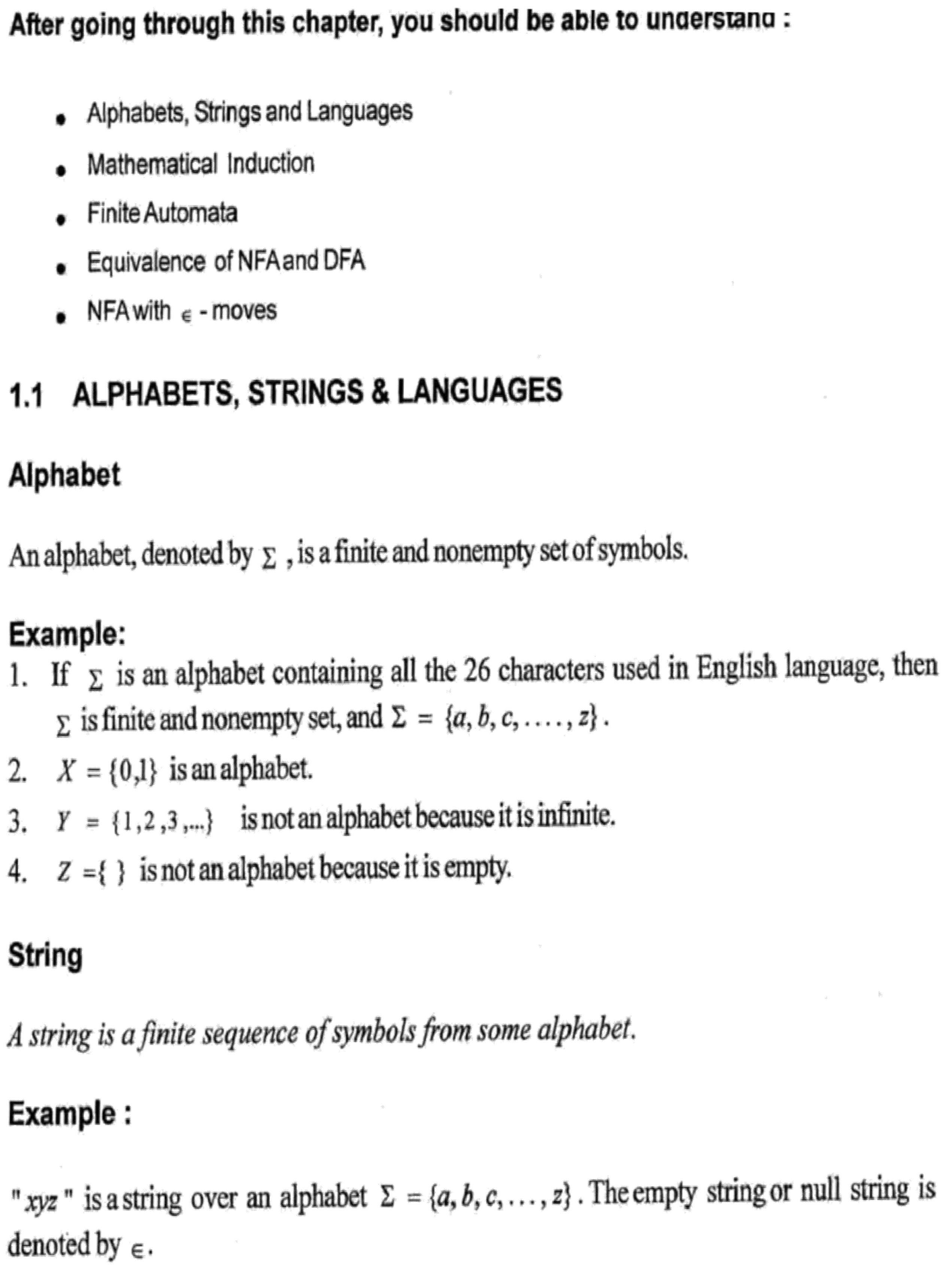
Topics (as per syllabus)	**Sub Topics**	**Lect. No.**	**Scheduled Date**	**Topic Delivered Date**
UNIT – IV	**Compiler , Lexical Analysis , Parsing Techniques: (No. of Lectures – 11)**			
Lexical Analysis	• Introduction: The structure of a compiler,	L37	04.03.25	
	• Lexical Analysis: The Role of the Lexical Analyzer	L38	05.03.25	
	• Input Buffering	L39	06.03.25	
	• Recognition of Tokens	L40	07.03.25	
	• The Lexical- Analyzer Generator Lex,	L41	10.03.25	
Syntax Analysis	• Introduction, Context-Free Grammars,	L42	12.03.25	
Syntax Analysis	• Top-Down Parsing,	L43	14.03.25	
	• Bottom- Up Parsing,	L44	17.03.25	
	• Introduction to LR Parsing: Simple LR	L45	18.03.25	
	• More Powerful LR Parsers	L46	19.03.25	

	• Slip Test	L47	04.03.25	
UNIT – V Syntax-Directed Translation, Intermediate-Code Generation ,Run-Time Environments (No. of Lectures – 11)				
Syntax-Directed Translation	• Syntax-Directed Definitions	L48	20.03.25	
	• Evaluation Orders for SDD's	L49	21.03.25	
	• Syntax- Directed Translation Schemes	L50	24.03.25	
	• Implementing L-Attributed SDD's.	L51	25.03.25	
	• Intermediate-Code Generation: Variants of Syntax Trees,	L52	26.03.25	
	• Three-Address Code • Run-Time Environments:	L53	27.03.25	
Run-Time Environments	• Stack Allocation of Space	L54	28.03.25	
	• Access to Nonlocal Data on the Stack	L55	02.04.25	
	• Heap Management	L56	03.04.25	
	• Slip test	L57	04.04.25	
	• Marks Distribution • Discussion about Paper • Counsel the students (AB/got poor marks)	L58	07.04.25	
Mid II Schedule:		**ATCD**	**Mid II Exam**	

Faculty **HOD**

LECTURE NOTES

UNIT-1



After going through this chapter, you should be able to unaerstana :

- Alphabets, Strings and Languages
- Mathematical Induction
- Finite Automata
- Equivalence of NFA and DFA
- NFA with $\in$ - moves

1.1 ALPHABETS, STRINGS & LANGUAGES

Alphabet

An alphabet, denoted by Σ , is a finite and nonempty set of symbols.

Example:

1. If Σ is an alphabet containing all the 26 characters used in English language, then Σ is finite and nonempty set, and $\Sigma = \{a, b, c, \ldots, z\}$.
2. $X = \{0,1\}$ is an alphabet.
3. $Y = \{1, 2, 3, \ldots\}$ is not an alphabet because it is infinite.
4. $Z = \{\ \}$ is not an alphabet because it is empty.

String

A string is a finite sequence of symbols from some alphabet.

Example :

" xyz " is a string over an alphabet $\Sigma = \{a, b, c, \ldots, z\}$. The empty string or null string is denoted by $\in$.

Prefix of a string

A string obtained by removing zero or more trailing symbols is called prefix. For example, if a string $w = abc$, then a, ab, abc are prefixes of w.

Suffix of a string

A string obtained by removing zero or more leading symbols is called suffix. For example, if a string $w = abc$, then c, bc, abc are suffixes of w.
A string a is a proper prefix or suffix of a string w if and only if $a \neq w$.

Substrings of a string

A string obtained by removing a prefix and a suffix from string w is called substring of w. For example, if a string $w = abc$, then b is a substring of w. Every prefix and suffix of string w is a substring of w, but not every substring of w is a prefix or suffix of w. For every string w, both w and $\in$ are prefixes, suffixes, and substrings of w.
Substring of $\mathrm{w} = \mathrm{w} - (\text{one prefix}) - (\text{one suffix})$**.**

Language

A Language L over Σ*, is a subset of* Σ^**, i. e., it is a collection of strings over the alphabet* Σ. ϕ, and $\{\in\}$ are languages. The language ϕ is undefined as similar to infinity and $\{\in\}$ is similar to an empty box i.e. a language without any string.

Example:

1. $L_1 = \{01, 0011, 000111\}$ is a language over alphabet $\{0,1\}$
2. $L_2 = \{\in, 0, 00, 000, ...\}$ is a language over alphabet $\{0\}$
3. $L_3 = \{0^n 1^n 2^n : n \geq 1\}$ is a language.

Kleene Closure of a Language

Let L be a language over some alphabet Σ. Then Kleene closure of L is denoted by $L*$ and it is also known as reflexive transitive closure, and defined as follows :

$L^* = \{Set\ of\ all\ words\ over\ \Sigma\}$

$= \{$word of length zero, words of length one, words of length two, ...$\}$

$$= \bigcup_{K=0}^{\infty}(\Sigma^K) = L^0 \cup L^1 \cup L^2 \cup \ldots$$

Example:

1. $\Sigma = \{a,b\}$ and a language L over Σ. Then

   $L^* = L^0 \cup L^1 \cup L^2 \cup \ldots$

   $L^0 = \{\in\}$

   $L^1 = \{a,b\}$,

   $L^2 = \{aa,ab,ba,bb\}$ and so on.

   So, $L^* = \{\in,a,b,aa,ab,ba,bb\ldots\}$
2. $S = \{0\}$, then $S^* = \{\in,0,00,000,0000,00000,\ldots\}$

Positive Closure

If Σ is an alphabet then positive closure of Σ is denoted by Σ^+ and defined as follows :

$\Sigma^+ = \Sigma^* - \{\in\} = \{Set\ of\ all\ words\ over\ \Sigma\ excluding\ empty\ string\ \in\}$

Example :

if $\Sigma = \{0\}$, then $\Sigma^+ = \{0,00,000,0000,00000,\ldots\}$

1. 2 MATHEMATICAL INDUCTION

Based on general observations specific truths can be identified by reasoning. This principle is called mathematical induction. The proof by mathematical induction involves four steps.

Basis : This is the starting point for an induction. Here, prove that the result is true for some n = 0 or 1.

Induction Hypothesis : Here, assume that the result is true for n = k.

Induction step : Prove that the result is true for some n = k + 1.

Proof of induction step : Actual proof.

1.3 FINITE AUTOMATA (FA)

A finite automata consists of a finite memory called input tape, a finite - nonempty set of states, an input alphabet, a read - only head , a transition function which defines the change of configuration, an initial state, and a finite - non empty set of final states.

A model of finite automata is shown in figure 1.1.

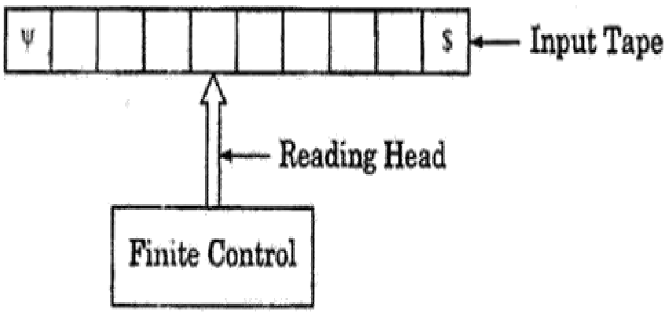


FIGURE 1.1 : Model of Finite Automata

The input tape is divided into cells and each cell contains one symbol from the input alphabet. The symbol 'ψ' is used at the leftmost cell and the symbol '$' is used at the rightmost cell to indicate the beginning and end of the input tape. The head reads one symbol on the input tape and finite control controls the next configuration. The head can read either from left - to - right or right - to -left one cell at a time. The head can't write and can't move backward. So , FA can't remember its previous read symbols. This is the major limitation of FA.

Deterministic Finite Automata (DFA)

A deterministic finite automata M can be described by 5 - tuple $(Q, \Sigma, \delta, q_0, F)$, where

1. Q is finite, nonempty set of states,
2. Σ is an input alphabet,
3. δ is transition function which maps $Q \times \Sigma \rightarrow Q$ i. e. the head reads a symbol in its present state and moves into next state.
4. $q_0 \in Q$, known as initial state
5. $F \subseteq Q$, known as set of final states.

Non - deterministic Finite Automata (NFA)

A non - deterministic finite automata M can be described by 5 - tuple $(Q, \Sigma, \delta, q_0, F)$, where

1 Q is finite, nonempty set of states,

2. Σ is an input alphabet,

3. δ is transition function which maps $Q \times \Sigma \rightarrow 2^Q$ i. e., the head reads a symbol in its present state and moves into the set of next state (s). 2^Q is power set of Q,

4. $q_0 \in Q$, known as initial state, and

5. $F \subseteq Q$, known as set of final states.

The difference between a DFA and a NFA is only in transition function. In DFA, transition function maps on at most one state and in NFA transition function maps on at least one state for a valid input symbol.

States of the FA

FA has following states :

1. **Initial state** : Initial state is an unique state ; from this state the processing starts.
2. **Final states** : These are special states in which if execution of input string is ended then execution is known as successful otherwise unsuccessful.
3. **Non - final states** : All states except final states are known as non - final states.
4. **Hang - states** : These are the states, which are not included into Q, and after reaching these states FA sits in idle situation. These have no outgoing edge. These states are generally denoted by ϕ. For example, consider a FA shown in figure1.2.

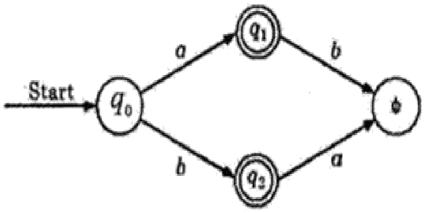


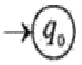
FIGURE 1.2 : Finite Automata

q_0 is the initial state, q_1, q_2 are final states, and ϕ is the hang state.

Notations used for representing FA

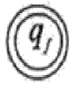
We represent a FA by describing all the five - terms $(Q, \Sigma, \delta, q_0, F)$. By using diagram to represent FA make things much clearer and readable. We use following notations for representing the FA :

1. The initial state is represented by a state within a circle and an arrow entering into circle as shown below :



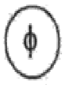
(Initial state q_0)

2. Final state is represented by final state within double circles :

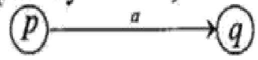


(Final state q_f)

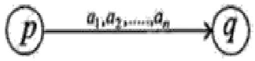
3. The hang state is represented by the symbol 'ϕ' within a circle as follows :



4. Other states are represented by the state name within a circle.
5. A directed edge with label shows the transition (or move). Suppose p is the present state and q is the next state on input - symbol 'a', then this is represented by

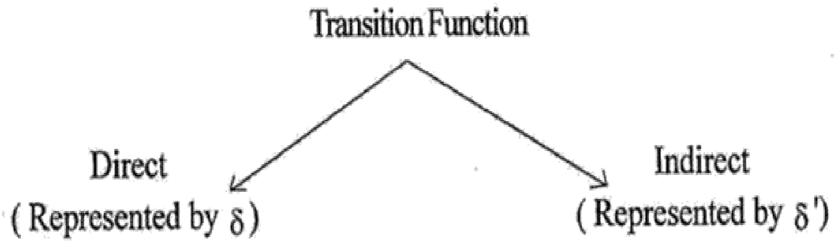


6. A directed edge with more than one label shows the transitions (or moves). Suppose p is the present state and q is the next state on input - symbols 'a_1' or 'a_2' or ... or 'a_n' then this is represented by



Transition Functions

We have two types of transition functions depending on the number of arguments.



Direct transition Function (δ)

When the input is a symbol, transition function is known as direct transition function.

Example : $\delta(p, a) = q$ (Where p is present state and q is the next state).

It is also known as one step transition.

Indirect transition function (δ')

When the input is a string, then transition function is known as indirect transition function.

Example : $\delta'(p, w) = q$, where p is the present state and q is the next state after | w | transitions. It is also known as one step or more than one step transition.

Properties of Transition Functions

1. If $\delta(p,a)=q$, then $\delta(p, ax) = \delta(q, x)$ and if $\delta'(p, x) = q$, then $\delta'(p, xa) = \delta'(q, a)$
2. For two strings x and y ; $\delta(p,xy) = \delta(\delta(p,x),y)$, and $\delta'(p,xy) = \delta'(\delta'(p,x),y)$

Example :1. A DFA $M = (\{q_0, q_1, q_2, q_f\}, \{0,1\}, \delta, q_0, \{q_f\})$ is shown in figure1.3 .

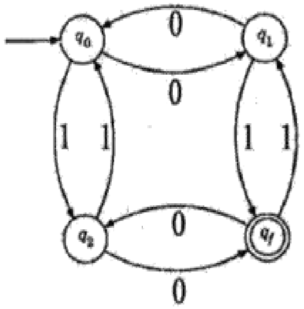


FIGURE 1.3 : Deterministic finite automata

Where δ is defined as follows :

	0	1
→ q_0	q_1	q_2
q_1	q_0	q_f
q_2	q_f	q_0
q_f	q_2	q_1

2. A NFA $M_1 = (\{q_0, q_1, q_2, q_f\}, \{0,1\}, \delta, q_0, \{q_f\})$ is shown in figure1.4.

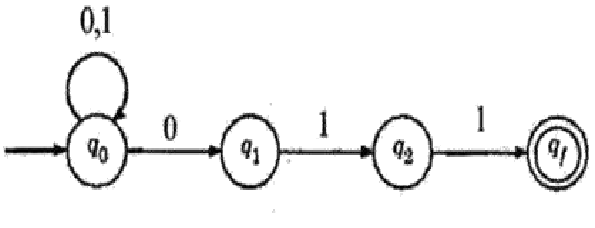
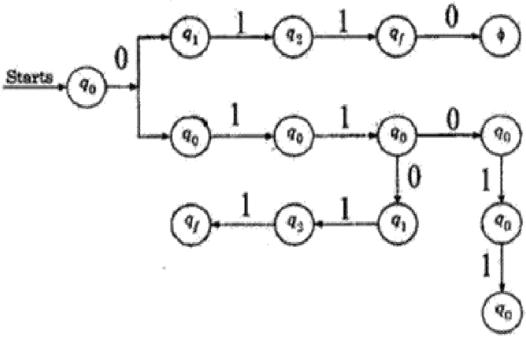


FIGURE 1.4 : Non - deterministic finite automata

3. Transition sequence for the string "011011" is as follows :



One execution ends in hang state ϕ , second ends in non - final state q_0 , and third ends in final state q_f hence string "011011" is accepted by third execution.

Difference between DFA and NFA

Strictly speaking the difference between DFA and NFA lies only in the definition of δ . Using this difference some more points can be derived and can be written as shown :

DFA	NFA
1. The DFA is 5 - tuple or quintuple $M = (Q, \Sigma, \delta, q_0, F)$ where Q is set of finite states Σ is set of input alphabets $\delta : Q \times \Sigma \; to \; Q$ q_0 is the initial state $F \subseteq Q$ is set of final states	The NFA is same as DFA except in the definition of δ . Here, δ is defined as follows : $\delta : Q \times (\Sigma \cup \in)$ to subset of 2^Q
2. There can be zero or one transition from a state on an input symbol	There can be zero, one or more transitions from a state on an input symbol
3. No $\in -$ transitions exist i.e., there should not be any transition or a transition if exist it should be on an input symbol	$\in -$ transitions can exist i. e., without any input there can be transition from one state to another state.
4. Difficult to construct	Easy to construct

The NFA accepts strings a, ab, abbb etc. by using $\in$ path between q_1 and q_2 we can move from q_1 state to q_2 without reading any input symbol. To accept ab first we are moving from q_0 to q_1 reading a and we can jump to q_2 state without reading any symbol there we accept b and we are ending with final state so it is accepted.

Equivalence of NFA with $\in$- Transitions and NFA without $\in$- Transitions

Theorem : If the language L is accepted by an NFA with $\in$- transitions, then the language L_1 is accepted by an NFA without $\in$- transitions.

Proof : Consider an NFA 'N' with $\in$- transitions where $N = (Q, \Sigma, \delta, q_0, F)$

Construct an NFA N_1 without $\in$- transitions $N_1 = (Q_1, \Sigma, \delta_1, q_0, F_1)$

where $Q_1 = Q$ and

$$F_1 = \begin{cases} F \cup \{q_0\} & \textit{if } \in\text{-}\textit{closure}(q_0) \textit{ contains a state of } F \\ F & \textit{otherwise} \end{cases}$$

and $\delta_1(q,a)$ is $\hat{\delta}(q,a)$ for q in Q and a in Σ.

Consider a non - empty string ω. To show by induction $|\omega|$ that $\delta_1(q_0, \omega) = \hat{\delta}(q_0, \omega)$

For $\omega = \in$, the above statement is not true. Because

$$\delta_1(q_0, \in) = \{q_0\},$$

while $$\hat{\delta}(q_0, \in) = \in -closure\ (q_0)$$

Basis :

Start induction with string length one.

i.e., $|\omega| = 1$

Then w is a symbol a, and $\delta_1(q_0, a) = \hat{\delta}(q_0, a)$ by definition of δ_1.

Induction : $|\omega| > 1$

Let $\omega = xy$ for symbol a in Σ.

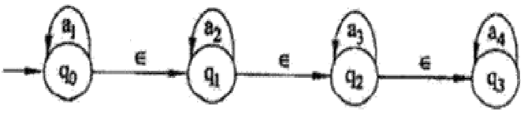
Then $\delta_1(q_0, xy) = \delta_1(\delta_1(q_0, x), y)$

Calculation of ∈ - closure :

∈- closure of state (∈-closure (q)) defined as it is a set of all vertices p such that there is a path from q to p labelled ∈ (including itself).

Example :

Consider the NFA with ∈ - moves



$\in -$ closure $(q_0) = \{ q_0, q_1, q_2, q_3 \}$

$\in -$ closure $(q_1) = \{ q_1, q_2, q_3 \}$

$\in -$ closure $(q_2) = \{ q_2, q_3 \}$

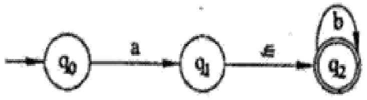
$\in -$ closure $(q_3) = \{ q_3 \}$

Procedure to convert NFA with ∈ moves to NFA without ∈ moves

Let $N = (Q, \Sigma, \delta, q_0, F)$ is a NFA with ∈ moves then there exists $N' = (Q, \in, \hat{\delta}, q_0, F')$ without ∈ moves

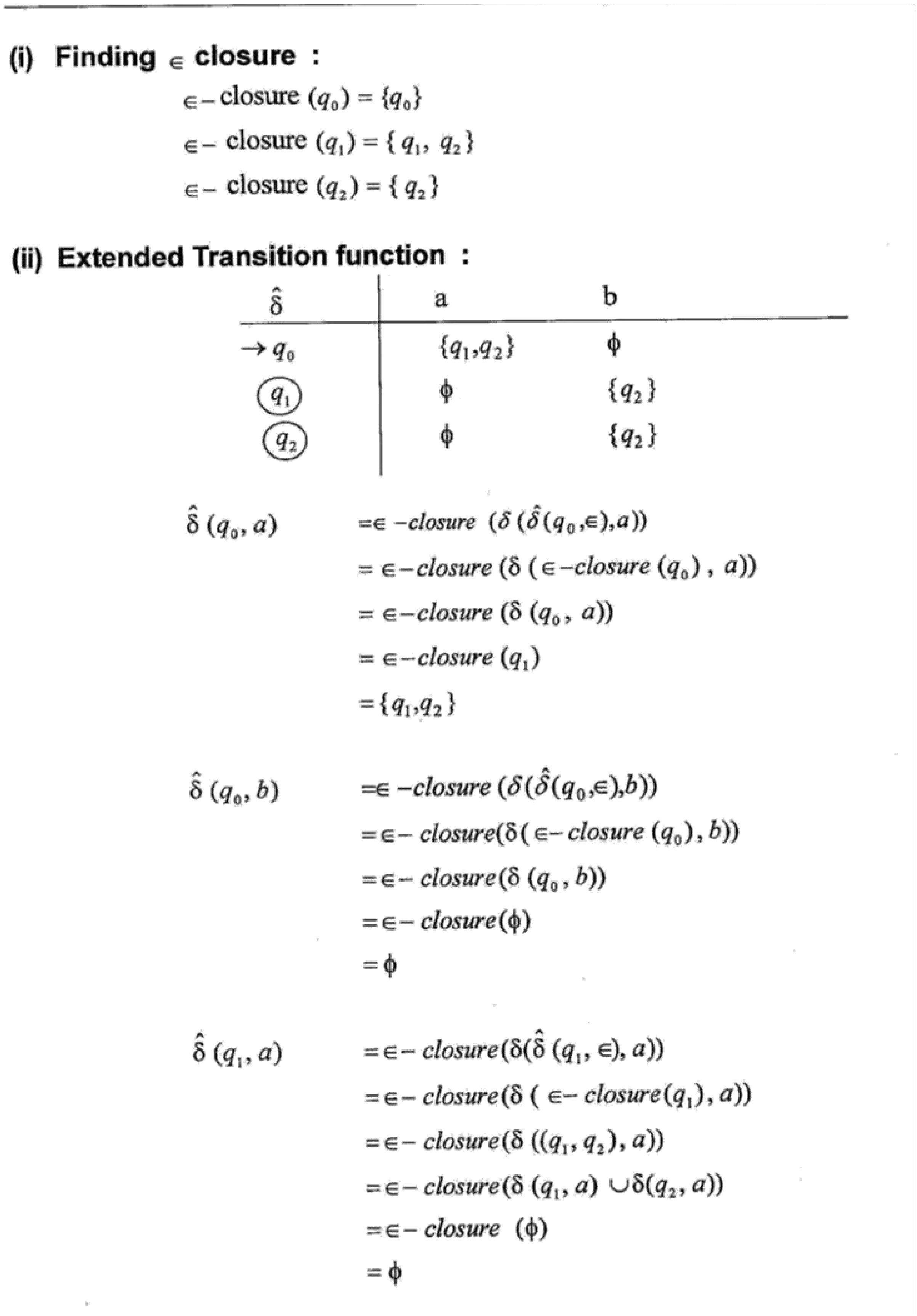
1. First find ∈ – closure of all states in the design.
2. Calculate extended transition function using following conversion formulae.
   (i) $\hat{\delta}(q, x) = \in -$ closure $(\delta(\hat{\delta}(q, \in), x))$
   (ii) $\hat{\delta}(q, \in) = \in -$ closure(q)
3. F' is a set of all states whose ∈ closure contains a final state in F.

Example 1 : Convert following NFA with ∈ moves to NFA without ∈ moves.



Solution : Transition table for given NFA is

δ	a	b	∈
$\rightarrow q_0$	q_1	φ	φ
q_1	φ	φ	q_2
q_2	φ	q_2	φ



(i) Finding $\in$ closure :

$$\in-\text{closure}\ (q_0) = \{q_0\}$$

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

$$\in-\text{closure}\ (q_2) = \{q_2\}$$

(ii) Extended Transition function :

$\hat{\delta}$	a	b
$\rightarrow q_0$	$\{q_1,q_2\}$	ϕ
q_1	ϕ	$\{q_2\}$
q_2	ϕ	$\{q_2\}$

$$\begin{aligned}
\hat{\delta}(q_0, a) &= \in -closure\ (\delta(\hat{\delta}(q_0,\in),a)) \\
&= \in-closure\ (\delta\ (\in-closure\ (q_0)\ ,\ a)) \\
&= \in-closure\ (\delta\ (q_0,\ a)) \\
&= \in-closure\ (q_1) \\
&= \{q_1,q_2\}
\end{aligned}$$

$$\begin{aligned}
\hat{\delta}(q_0, b) &= \in -closure\ (\delta(\hat{\delta}(q_0,\in),b)) \\
&= \in-\ closure(\delta(\in-closure\ (q_0), b)) \\
&= \in-\ closure(\delta\ (q_0, b)) \\
&= \in-\ closure(\phi) \\
&= \phi
\end{aligned}$$

$$\begin{aligned}
\hat{\delta}(q_1, a) &= \in-\ closure(\delta(\hat{\delta}\ (q_1, \in), a)) \\
&= \in-\ closure(\delta\ (\ \in-\ closure(q_1), a)) \\
&= \in-\ closure(\delta\ ((q_1, q_2), a)) \\
&= \in-\ closure(\delta\ (q_1, a)\ \cup \delta(q_2, a)) \\
&= \in-\ closure\ \ (\phi) \\
&= \phi
\end{aligned}$$

$$
\begin{aligned}
\hat{\delta}(q_1, b) &= \in-closure\ (\delta(\hat{\delta}(q_1, \in), b)) \\
&= \in-closure\ (\delta(\in-closure(q_1), b)) \\
&= \in-closure\ (\delta((q_1, q_2), b)) \\
&= \in-closure\ (\delta(q_1, b) \cup \delta(q_2, b)) \\
&= \in-closure\ (q_2) \\
&= \{q_2\}
\end{aligned}
$$

$$
\begin{aligned}
\hat{\delta}(q_2, a) &= \in-closure\ (\delta(\hat{\delta}(q_2, \in), a)) \\
&= \in-closure\ (\delta(\in-closure(q_2), a)) \\
&= \in-closure\ (\delta(q_2, a)) \\
&= \in-closure\ (\phi) \\
&= \phi
\end{aligned}
$$

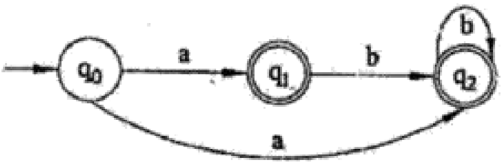
$$
\begin{aligned}
\hat{\delta}(q_2, b) &= \in-closure\ (\delta(\hat{\delta}(q_2, \in), b)) \\
&= \in-closure\ (\delta(\in-closure(q_2), b)) \\
&= \in-closure\ (\delta(q_2, b)) \\
&= \in-closure\ (q_2) \\
&= \{q_2\}
\end{aligned}
$$

(iii) Final states are q_1, q_2, because

$\in-closure\ (q_1)$ contains final state

$\in-closure\ (q_2)$ contains final state

(iv) NFA without $\in$ moves is



2.1 FINITE STATE MACHINES (FSMs)

A finite state machine is similar to finite automata having additional capability of outputs.

A model of finite state machine is shown in below figure .

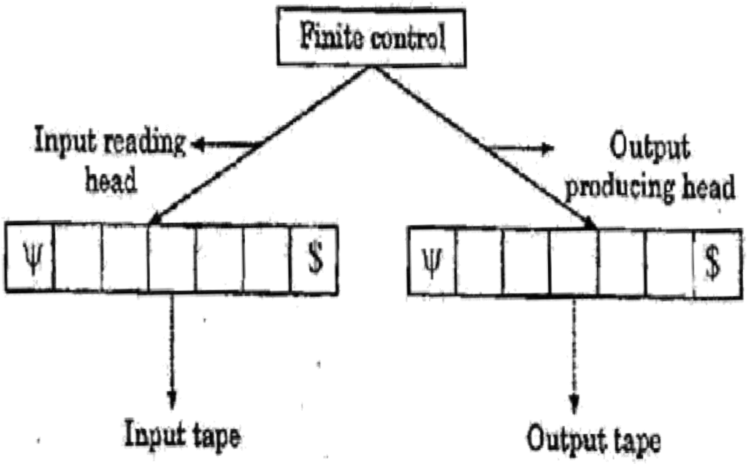


FIGURE : Model of FSM

2.1.1 Description of FSM

A finite state machine is represented by 6 - tuple $(Q,\Sigma,\Delta,\delta,\lambda,q_0)$, where

1. Q is finite and non - empty set of states,
2. Σ is input alphabet,
3. Δ is output alphabet,

4. δ is transition function which maps present state and input symbol on to the next state or $Q \times \Sigma \to Q$,
5. λ is the output function, and
6. $q_0 \in Q$, is the initial state .

2.1.2 Representation of FSM

We represent a finite state machine in two ways ; one is by transition table, and another is by transition diagram . In transition diagram , edges are labeled with Input / output.

Suppose , in transition table the entry is defined by a function F, so for input a_i and state q_i

$F(q_i, a_i) = (\delta(q_i, a_i), \lambda(q_i, a_i))$ (where δ is transition function, λ is output function.)

Example 1 : Consider a finite state machine, which changes 1's into 0's and 0's into 1's (1's complement) as shown in below figure .

Transition diagram :

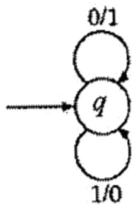


FIGURE : Finite state machine

Transition table :

	Inputs			
	0		1	
Present State(PS)	Next State (NS)	Output	Next State (NS)	Output
q	q	1	q	0

Example 2 : Consider the finite state machine shown in below figure, which outputs the 2's complement of input binary number reading from least significant bit (LSB).

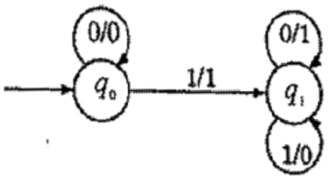
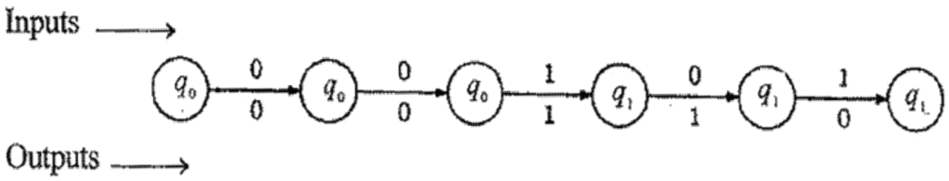


FIGURE : Finite State machine

Suppose, input is 10100. What is the output ?

Solution : The finite state machine reads the input from right side (LSB).

Transition sequence for input 10100 :



So, the output is 01100.

2.2 MOORE MACHINE

If the *output of finite state machine is dependent on present state only,* then this model of finite state machine is known as Moore machine.

A Moore machine is represented by 6-tuple $(Q, \Sigma, \Delta, \delta, \lambda, q_0)$, where

1 Q is finite and non-empty set of states,

2 Σ is input alphabet,

3 Δ is output alphabet,

4 δ is transition function which maps present state and input symbol on to the next state or $Q \times \Sigma \rightarrow Q$,

5 λ is the output function which maps $Q \rightarrow \Delta$, (Present state → Output), and

6 $q_0 \in Q$, is the initial state.

If $Z(t)$, $q(t)$ are output and present state respectively at time t then

$$Z(t) = \lambda(q(t)).$$

$$\boxed{\text{For input } \epsilon \text{ (null string), } Z(t) = \lambda \text{ (initial state)}}$$

Consider three LSBs of

Input	Output
...000 (X)	C
...001 (X)	C
...010 (X)	C
...011 (X)	C
...100 (X)	C
...101	A
...110	B
...111 (X)	C

Transition diagram :

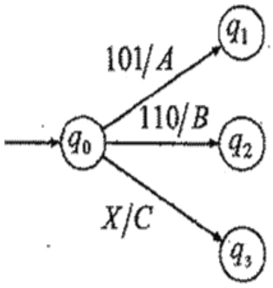


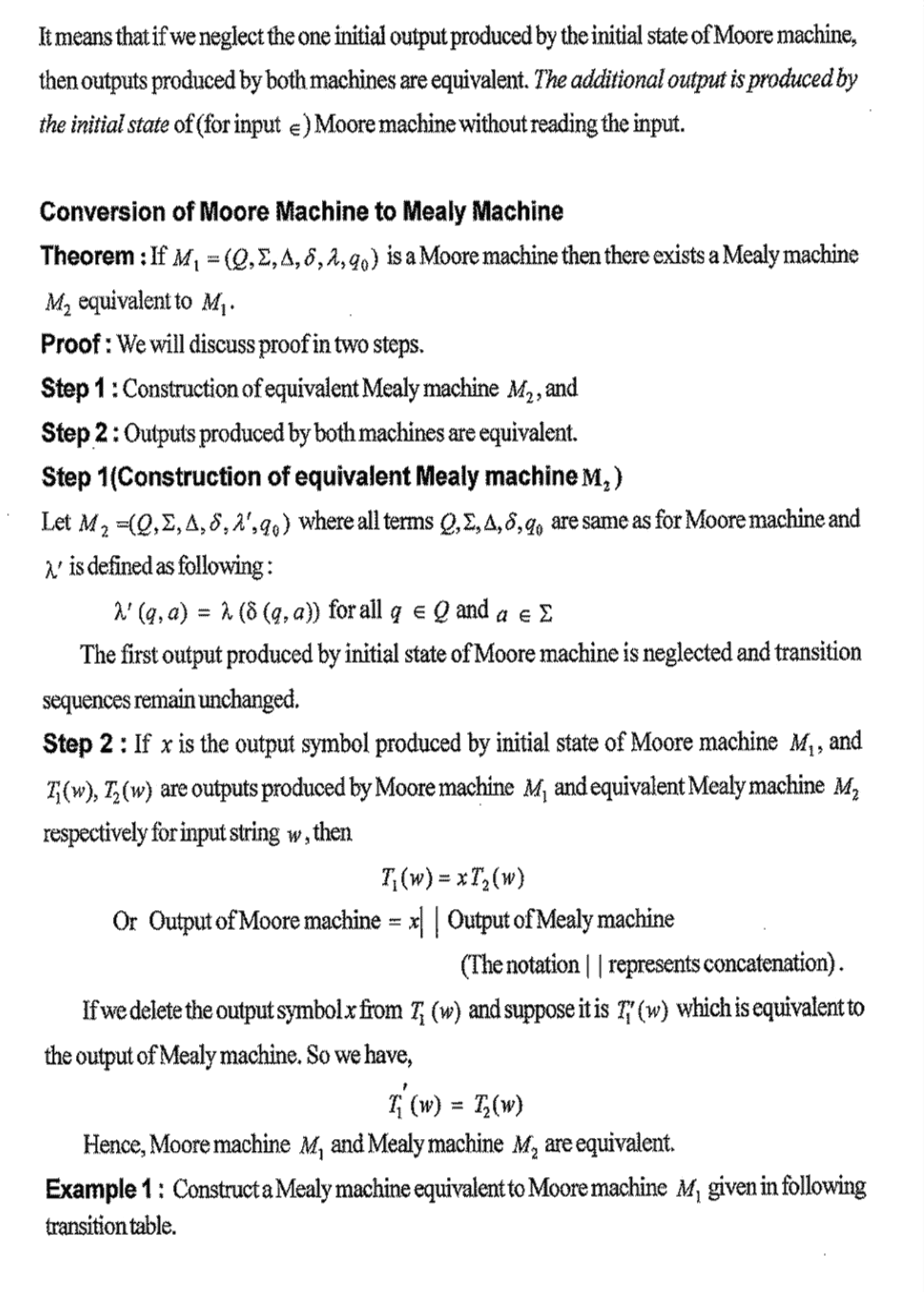
FIGURE : Moore Machine

2.4 EQUIVALENCE OF MOORE AND MEALY MACHINES

We can construct equivalent Mealy machine for a Moore machine and vice-versa. Let M_1 and M_2 be equivalent Moore and Mealy machines respectively. The two outputs $T_1\ (w)$ and $T_2\ (w)$ are produced by the machines M_1 and M_2 respectively for input string w. Then the length of $T_1\ (w)$ is one greater than the length of $T_2(w)$, i.e.

$$|T_1\ (w)| = |T_2\ (w)| + 1$$

The additional length is due to the output produced by initial state of Moore machine. Let output symbol x is the additional output produced by the initial state of Moore machine, then $T_1(w) = x\,T_2(w)$.



It means that if we neglect the one initial output produced by the initial state of Moore machine, then outputs produced by both machines are equivalent. *The additional output is produced by the initial state* of (for input $\in$) Moore machine without reading the input.

Conversion of Moore Machine to Mealy Machine

Theorem : If $M_1 = (Q, \Sigma, \Delta, \delta, \lambda, q_0)$ is a Moore machine then there exists a Mealy machine M_2 equivalent to M_1.

Proof : We will discuss proof in two steps.

Step 1 : Construction of equivalent Mealy machine M_2, and

Step 2 : Outputs produced by both machines are equivalent.

Step 1(Construction of equivalent Mealy machine M_2)

Let $M_2 = (Q, \Sigma, \Delta, \delta, \lambda', q_0)$ where all terms $Q, \Sigma, \Delta, \delta, q_0$ are same as for Moore machine and λ' is defined as following :

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

The first output produced by initial state of Moore machine is neglected and transition sequences remain unchanged.

Step 2 : If x is the output symbol produced by initial state of Moore machine M_1, and $T_1(w), T_2(w)$ are outputs produced by Moore machine M_1 and equivalent Mealy machine M_2 respectively for input string w, then

$$T_1(w) = xT_2(w)$$

Or Output of Moore machine $= x| \,|$ Output of Mealy machine

(The notation | | represents concatenation).

If we delete the output symbol x from $T_1(w)$ and suppose it is $T_1'(w)$ which is equivalent to the output of Mealy machine. So we have,

$$T_1'(w) = T_2(w)$$

Hence, Moore machine M_1 and Mealy machine M_2 are equivalent.

Example 1 : Construct a Mealy machine equivalent to Moore machine M_1 given in following transition table.

3. Δ remains unchanged,

4. λ' is defined as follows :

   $\delta'([q,b],a) = [\delta(q,a), \lambda(q,a)]$, where δ and λ are transition function and output function of Mealy machine.

5. λ' is the output function of equivalent Moore machine which is dependent on present state only and defined as follows :

$$\lambda'([q,b]) = b$$

6. q_0' is the initial state and defined as $[q_0,b_0]$, where q_0 is the initial state of Mealy machine and b_0 is any arbitrary symbol selected from output alphabet Δ.

Step 2 : Outputs of Mealy and Moore Machines

Suppose, Mealy machine M_1 enters states $q_0, q_1, q_2, \ldots q_n$ on input $a_1, a_2, a_3, \ldots . a_n$ and produces outputs $b_1, b_2, b_3, \ldots b_n$, then M_2 enters the states $[q_0,b_0], [q_1,b_1], [q_2,b_2] \ldots, [q_n,b_n]$ and produces outputs $b_0, b_1, b_2, \ldots b_n$ as discussed in Step 1. Hence, outputs produced by both machines are equivalent.

Therefore, Mealy machine M_1 and Moore machine M_2 are equivalent.

Example 1 : Consider the Mealy machine shown in below figure. Construct an equivalent Moore machine.

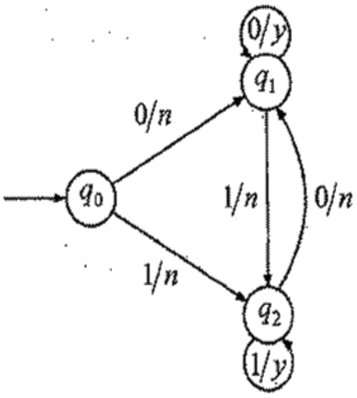


FIGURE : Mealy Machine

Solution : Let $M_1 = (Q,\Sigma,\Delta,\delta,\lambda,q_0)$ is a given Mealy machine and $M_2 = (Q',\Sigma,\Delta,\delta',\lambda',q_0')$ be the equivalent Moore machine,

where

1. $Q' \subseteq \{[q_0,n],[q_0,y],[q_1,n],[q_1,y],[q_2,n],[q_2,y]\}$ (Since, $Q' \subseteq Q \times \Delta$)
2. $\Sigma = \{0, 1\}$

3. $\Delta = \{n, y\}$,

4. $q_0' = [q_0, y]$, where q_0 is the initial state and y is the output symbol of Mealy machine,

5. δ' is defined as following :

For initial state $[q_0, y]$:

$$\delta'([q_0, y], 0) = [\delta(q_0, 0), \lambda(q_0, 0)] = [q_1, n]$$

$$\delta'([q_0, y], 1) = [\delta(q_0, 1), \lambda(q_0, 1)] = [q_2, n]$$

For state $[q_1, n]$:

$$\delta'([q_1, n], 0) = [\delta(q_1, 0), \lambda(q_1, 0)] = [q_1, y]$$

$$\delta'([q_1, n], 1) = [\delta(q_1, 1), \lambda(q_1, 1)] = [q_2, n]$$

For state $[q_2, n]$:

$$\delta'([q_2, n], 0) = [\delta(q_2, 0), \lambda(q_2, 0)] = [q_1, n]$$

$$\delta'([q_2, n], 1) = [\delta(q_2, 1), \lambda(q_2, 1)] = [q_2, y]$$

For state $[q_1, y]$:

$$\delta'([q_1, y], 0) = [\delta(q_1, 0), \lambda(q_1, 0)] = [q_1, y]$$

$$\delta'([q_1, y], 1) = [\delta(q_1, 1), \lambda(q_1, 1)] = [q_2, n]$$

For state $[q_2, y]$:

$$\delta'([q_2, y], 0) = [\delta(q_2, 0), \lambda(q_2, 0)] = [q_1, n]$$

$$\delta'([q_2, y], 1) = [\delta(q_2, 1), \lambda(q_2, 1)] = [q_2, y]$$

(**Note :** We have considered only those states, which are reachable from initial state)

6. λ' is defined as follows :

$$\lambda'[q_0, y] = y$$

$$\lambda'[q_1, n] = n$$

$$\lambda'[q_2, n] = n$$

$$\lambda'[q_1, y] = y$$

$$\lambda'[q_2, y] = y$$

2.5 EQUIVALENCE OF FSMs

Two finite machines are said to be equivalent if and only if every input sequence yields identical output sequence.

Example :

Consider the FSM M_1 shown in figure (a) and FSM M_2 shown in figure (b).

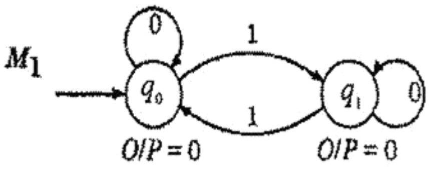


Figure (a)

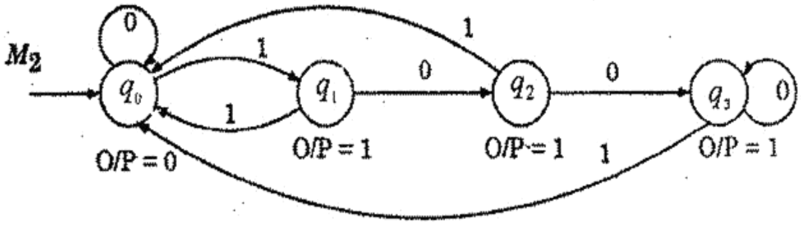


Figure (b)

Are these two FSMs equivalent ?

Solution :

We check this. Consider the input strings and corresponding outputs as given following :

Input string	Output by M_1	Output by M_2
(1) 01	00	00
(2) 010	001	001
(3) 0101	0011	0011
(4) 1000	0111	0111
(5) 10001	01111	01111

Now, we come to this conclusion that for each input sequence, outputs produced by both machines are identical. So, these machines are equivalent. In other words, both machines do the same task. But, M_1 has two states and M_2 has four states. So, some states of M_2 are doing the same task i. e., producing identical outputs on certain input. Such states are known as equivalent states and require extra resources when implemented.

Thus, our goal is to find the simplest and equivalent FSM with minimum number of states.

2.5.1 FSM Minimization

We minimize a FSM using the following method, which finds the equivalent states, and merges these into one state and finally construct the equivalent FSM by minimizing the number of states.

Method : Initially we assume that all pairs (q_0, q_1) over states are non - equivalent states

Step 1 : Construct the transition table.

Step 2 : Repeat for each pair of non - equivalent states (q_0, q_1) :

(a) Do q_0 and q_1 produce same output ?

(b) Do q_0 and q_1 reach the same states for each input $a \in \Sigma$?

(c) If answers of (a) and (b) are YES, then q_0 and q_1 are equivalent states and merge these two states into one state $[q_0, q_1]$ and replace the all occurrences of q_0 and q_1 by $[q_0, q_1]$ and mark these equivalent states.

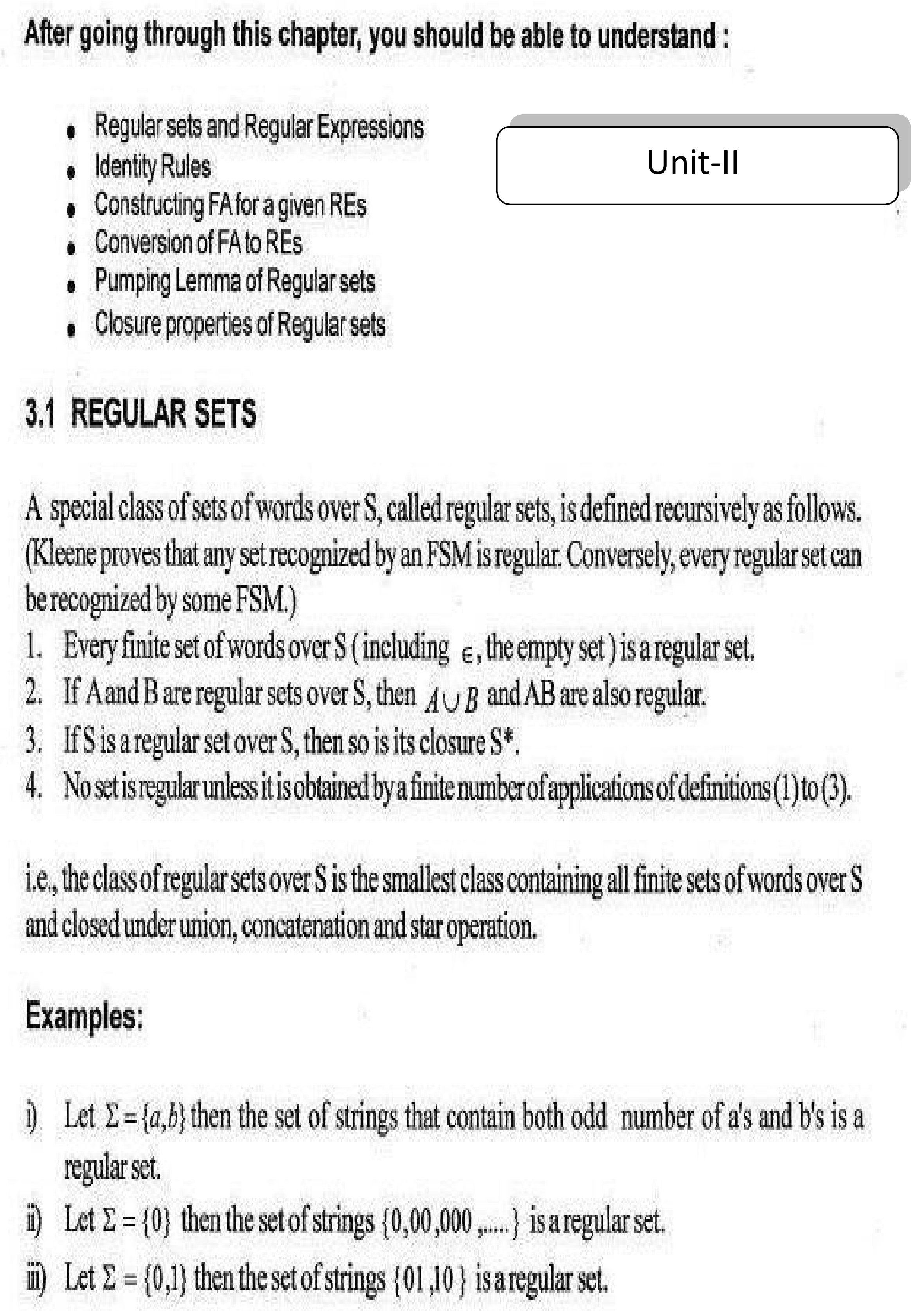
Step 3 : Check the all - present states, if any redundancy is found, remove that.

Step 4 : Exit.

Example 1 : Consider the following transition table for FSM. Construct minimum state FSM.

	Inputs		
	0	1	
Present State(PS)	**Next State (NS)**	**Next State (NS)**	**Output**
q_0	q_0	q_1	0
q_1	q_2	q_0	1
q_2	q_3	q_0	1
q_3	q_3	q_0	1

Unit-II



After going through this chapter, you should be able to understand :

- Regular sets and Regular Expressions
- Identity Rules
- Constructing FA for a given REs
- Conversion of FA to REs
- Pumping Lemma of Regular sets
- Closure properties of Regular sets

3.1 REGULAR SETS

A special class of sets of words over S, called regular sets, is defined recursively as follows. (Kleene proves that any set recognized by an FSM is regular. Conversely, every regular set can be recognized by some FSM.)

1. Every finite set of words over S (including $\in$, the empty set) is a regular set.
2. If A and B are regular sets over S, then $A \cup B$ and AB are also regular.
3. If S is a regular set over S, then so is its closure S*.
4. No set is regular unless it is obtained by a finite number of applications of definitions (1) to (3).

i.e., the class of regular sets over S is the smallest class containing all finite sets of words over S and closed under union, concatenation and star operation.

Examples:

i) Let $\Sigma = \{a,b\}$ then the set of strings that contain both odd number of a's and b's is a regular set.

ii) Let $\Sigma = \{0\}$ then the set of strings $\{0,00,000,.....\}$ is a regular set.

iii) Let $\Sigma = \{0,1\}$ then the set of strings $\{01,10\}$ is a regular set.

3.2 REGULAR EXPRESSIONS

The languages accepted by FA are regular languages and these languages are easily described by simple expressions called regular expressions. We have some algebraic notations to represent the regular expressions.

Regular expressions are means to represent certain sets of strings in some algebraic manner and regular expressions describe the language accepted by FA.

If Σ is an alphabet then regular expression(s) over this can be described by following rules.

1. Any symbol from $\Sigma, \in$ *and* ϕ are regular expressions.
2. If r_1 and r_2 are two regular expressions then *union* of these represented as $r_1 \cup r_2$ or $r_1 + r_2$ is also a regular expression
3. If r_1 and r_2 are two regular expressions then *concatenation* of these represented as $r_1 r_2$ is also a regular expression.
4. The Kleene closure of a regular expression *r* is denoted by $r*$ is also a regular expression.
5. If *r* is a regular expression then (r) is also a regular expression.
6. The regular expressions obtained by applying rules 1 to 5 once or more than once are also regular expressions.

Examples :

(1) If $\Sigma = \{a, b\}$, then

(a) *a* is a regular expression (Using rule 1)

(b) *b* is a regular expression (Using rule 1)

(c) $a + b$ is a regular expression (Using rule 2)

(d) $b*$ is a regular expression (Using rule 4)

(e) ab is a regular expression (Using rule 3)

(f) $ab + b*$ is a regular expression (Using rule 6)

(2) Find regular expression for the following

(a) A language consists of all the words over $\{a, b\}$ ending in b.

(b) A language consists of all the words over $\{a, b\}$ ending in bb.

(c) A language consists of all the words over $\{a, b\}$ starting with *a* and ending in *b*.

(d) A language consists of all the words over $\{a, b\}$ having bb as a substring.

(e) A language consists of all the words over $\{a, b\}$ ending in *aab*.

Solution : Let $\Sigma = \{a,b\}$, and

All the words over $\Sigma = \{\in, a, b, aa, bb, ab, ba, aaa, \ldots\ldots\} = \Sigma * \text{ or } (a + b) * \text{ or } (a \cup b) *$

$= (\{\in, a, b, aa, bb, \ldots\})^*$

$= \{\in, a, b, aa, bb, ab, ba, aaa, bbb, abb, baa, aabb, \ldots\}$

$= \{\text{All the words over } \{a, b\}\}$

$\equiv (a + b)^*$

So, $(a^* + b^*)^* \equiv (a + b)^*$

3.3 IDENTITIES FOR REs

The two regular expressions P and Q are equivalent (denoted as P = Q) if and only if P represents the same set of strings as Q does. For showing this equivalence of regular expressions we need to show some identities of regular expressions.

Let P, Q and R are regular expressions then the identity rules are as given below

1. $\in R = R \in = R$
2. $\in^* = \in$ ∈ is null string
3. $(\phi)^* = \in$ ϕ is empty string.
4. $\phi R = R\phi = \phi$
5. $\phi + = R = R$
6. $R + R = R$
7. $RR^* = R^* R = R^+$
8. $(R^*)^* = R^*$
9. $\in + RR^* = R^*$
10. $(P + Q)R = PR + QR$
11. $(P + Q)^* = (P^* Q^*) = (P^* + Q^*)^*$
12. $R^*(\in + R) = (\in + R)R^* = R^*$
13. $(R + \in)^* = R^*$
14. $\in + R^* = R^*$
15. $(PQ)^* P = P(QP)^*$
16. $R^* R + R = R^* R$

3.3.1 Equivalence of two REs

Let us see one important theorem named Arden's Theorem which helps in checking the equivalence of two regular expressions.

Arden's Theorem : Let P and Q be the two regular expressions over the input set Σ. The regular expression R is given as

$$R = Q + RP$$

Which has a unique solution as $R = QP^*$

Proof : Let, P and Q are two regular expressions over the input string Σ.
If P does not contain $\in$ then there exists R such that

$$R = Q + RP \qquad \text{.... (1)}$$

We will replace R by QP* in equation 1.
Consider R. H. S. of equation 1.

$$= Q + QP^* P$$

$$= Q(\in + P^* P)$$

$$= QP^* \qquad \because \in + R^* R = R^*$$

Thus $R = QP^*$

is proved. To prove that $R = QP^*$ is a unique solution, we will now replace L.H.S. of equation 1 by Q + RP. Then it becomes

$$Q + RP$$

But again R can be replaced by Q + RP.

$$\therefore \quad Q + RP = Q + (Q + RP) P$$

$$= Q + QP + RP^2$$

Again replace R by Q + RP.

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

$$= Q + QP + QP^2 + RP^3$$

Thus if we go on replacing R by Q + RP then we get,

$$Q + RP = Q + QP + QP^2 + \ldots\ldots + QP^i + RP^{i+1}$$

$$= Q(\in + P + P^2 + \ldots\ldots P^i) + RP^{i+1}$$

From equation 1,

$$R = Q(\in + P + P^2 + \ldots\ldots + P^i) + RP^{i+1} \qquad \text{... (2)}$$

Where $i \geq 0$

Consider equation 2,

$$R = Q\underbrace{(\in + P + P^2 + \ldots.. + P^i)}_{P^*} + RP^{i+1}$$

$$\therefore \quad R = QP^* + RP^{i+1}$$

Let w be a string of length i.

$=\{\in,0,00,1,11,111,01,10,.........\}$

$=\{\ \in$, any combination of 0's, any combination of 1's, any combination of 0 and 1 $\}$

Hence, L. H. S. = R. H. S. is proved.

3.4 RELATIONSHIP BETWEEN FA AND RE

There is a close relationship between a finite automata and the regular expression we can show this relation in below figure.

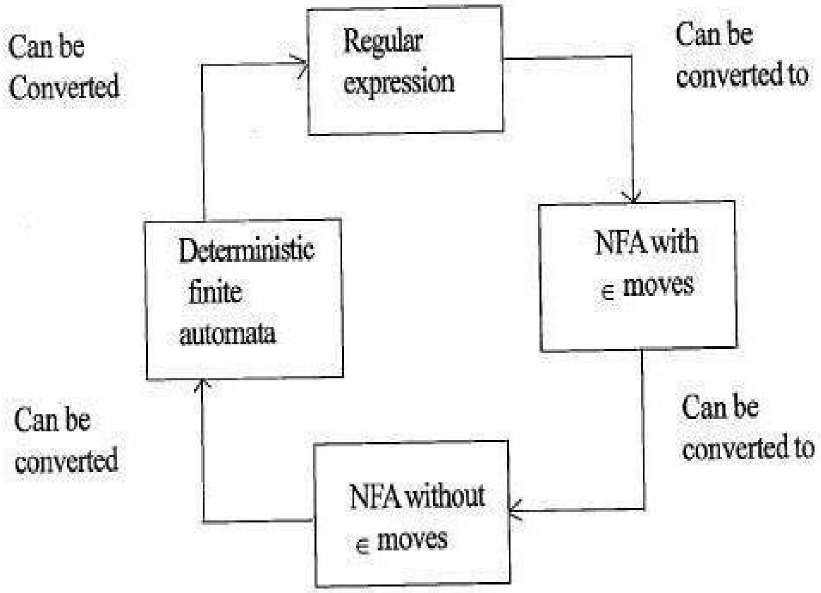


FIGURE : Relationship between FA and regular expression

The above figure shows that it is convenient to convert the regular expression to NFA with $\in$ moves. Let us see the theorem based on this conversion.

3.5 CONSTRUCTING FA FOR A GIVEN REs

Theorem : If r be a regular expression then there exists a NFA with $\in$ - moves, which accepts $L(r)$.

Proof : First we will discuss the construction of NFA M with $\in$ - moves for regular expression r and then we prove that $L(M) = L(r)$.

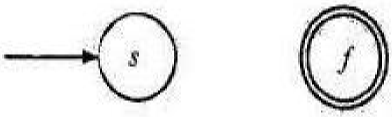
Let r be the regular expression over the alphabet Σ.

Construction of NFA with $\in$ - moves

Case 1 :

(i) $r = \phi$

NFA $M = (\{s, f\}, \{\ \}\delta, s, \{f\})$ as shown in Figure1 (a)

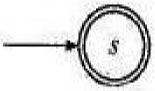


(No path from initial state *s* to reach the final state *f*.)

Figure 1 (a)

(ii) $r = \in$

NFA $M = (\{s\}, \{\ \}, \delta, s, \{s\})$ as shown in Figure 1 (b)

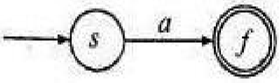


(The initial state *s* is the final state)

Figure 1 (b)

(iii) $r = a$, for all $a \in \Sigma$,

NFA $M = (\{s, f\}, \Sigma, \delta, s, \{f\})$



(One path is there from initial state *s* to reach the final state *f* with label *a*.)

Figure 1 (c)

Case 2 : $|r| \geq 1$

Let r_1 and r_2 be the two regular expressions over Σ_1, Σ_2 and N_1 and N_2 are two NFA for r_1 and r_2 respectively as shown in Figure 2 (a).

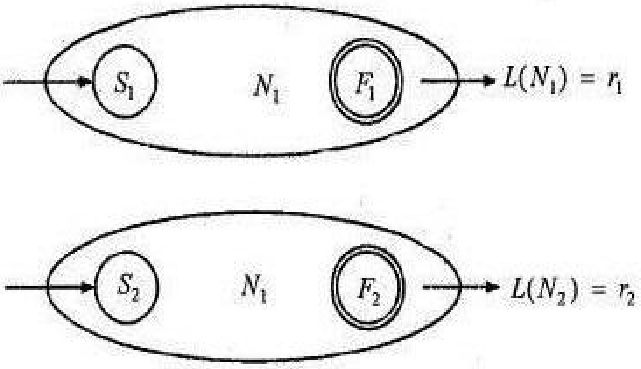


Figure 2 (a) NFA for regular expression r_1 and r_2

Now let us compute for final state, which denotes the regular expression.

r_{12}^2 will be computed, because there are total 2 states and final state is q_1 whose start state is q_0.

$$r_{12}^2 = \left(r_{12}^1\right)\left(r_{22}^1\right)^*\left(r_{22}^1\right)+\left(r_{12}^1\right)$$
$$= 0(\in)^*(\in) + 0$$
$$= 0 + 0$$

$r_{12}^2 = 0$ which is a final regular expression.

3.6.1 Arden's Method for Converting DFA to RE

As we have seen the Arden's theorem is useful for checking the equivalence of two regular expressions, we will also see its use in conversion of DFA to RE.

Following algorithm is used to build the r. e. from given DFA.

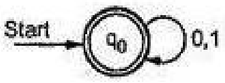
1. Let q_0 be the initial state.
2. There are $q_1, q_2, q_3, q_4, \ldots q_n$ number of states.The final state may be some q_j where $j \leq n$.
3. Let α_{ji} represents the transition from q_j *to* q_i.
4. Calculate q_i such that

   $$q_i = \alpha_{ji} . q_j$$

   If q_i is a start state

   $$q_i = \alpha_{ji} . q_j + \in$$
5. Similarly compute the final state which ultimately gives the regular expression r.

Example 1 : Construct RE for the given DFA.

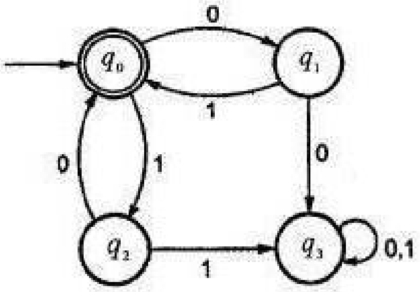


Solution :

Since there is only one state in the finite automata let us solve for q_0 only.

$$q_0 = q_0 0 + q_0 1 + \in$$
$$q_0 = q_0(0+1) + \in$$

Example 3 : Construct RE for the DFA given in below figure.



Solution : Let us see the equations

$$q_0 = q_1 1 + q_2 0 + \in$$
$$q_1 = q_0 0$$
$$q_2 = q_0 1$$
$$q_3 = q_1 0 + q_2 1 + q_3 (0+1)$$

Let us solve q_0 first,

$$q_0 = q_1 1 + q_2 0 + \in$$
$$q_0 = q_0 01 + q_0 10 + \in$$
$$q_0 = q_0 (01+10) + \in$$
$$q_0 = \in .(01+10)^*$$
$$q_0 = (01+10)^*$$

$\because R = Q + RP$

$\Rightarrow QP^*$ where

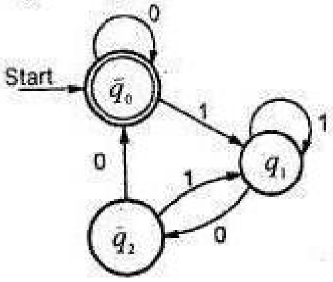
$R = q_0, Q = \in, P = (01+10)$

Thus the regular expression will be

$$r = (01+10)^*$$

Since q_0 is a final state, we are interested in q_0 only.

Example 4 : Find out the regular expression from given DFA.



Example 8 : Show that the language $L = \{a^i\ b^{2i} | i > 0\}$ is not regular.

Solution : The set of strings accepted by language L is,

$L = \{abb, aabbbb, aaabbbbbb, aaaabbbbbbbb...\}$

Applying Pumping lemma for any of the strings above.
Take the string abb.
It is of the form uvw.
Where, $|uv| \leq i, |v| \geq 1$

To find i such that $uv^i w \notin L$
Take i = 2 here, then

$uv^2 w = a(bb)b$

$= abbb$

Hence $uv^2 w = abbb \notin L$
Since abbb is not present in the strings of L.

$\therefore$ L is not regular.

Example 9 : Show that $L = \{0^n | n$ is a perfect square $\}$ is not regular.

Solution :

Step 1 : Let L is regular by Pumping lemma. Let n be number of states of FA accepting L.

Step 2 : Let $z = 0^n$ then $|z| = n \geq 2$.

Therefore, we can write z = uvw ; Where $|uv| \leq n, |v| \geq 1$.
Take any string of the language L = { 00, 0000, 000000 .... }
Take 0000 as string, here u = 0, v = 0, w = 00 to find i such that $uv^i w \notin L$.
Take i = 2 here, then

$uv^i w = 0(0)^2 00$

$= 00000$

This string 00000 is not present in strings of language L. So $uv^i w \notin L$.

$\therefore$ It is a contradiction.

3.9 PROPERTIES OF REGULAR SETS

Regular sets are closed under following properties.

1. Union
2. Concatenation

3. Kleene Closure
4. Complementation
5. Transpose
6. Intersection

1. **Union :** If R_1 and R_2 are two regular sets, then union of these denoted by $R_1 + R_2$ or $R_1 \cup R_2$ is also a regular set.

Proof : Let R_1 and R_2 be recognized by NFA N_1 and N_2 respectively as shown in Figure1(a) and Figure1(b).

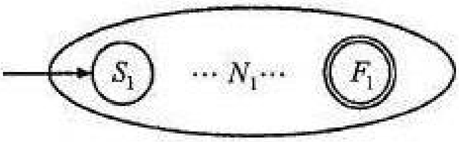


FIGURE 1(a) NFA for regular set R_1

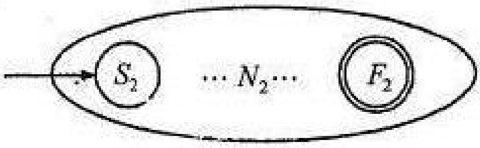


FIGURE 1(b) NFA for regular set R_2

We construct a new NFA N based on union of N_1 and N_2 as shown in Figure 1 (c)

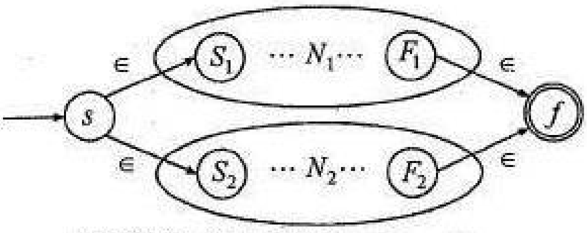


FIGURE 1(c) NFA for $N_1 + N_2$

Now,

$$L(N) = \in L(N_1) \in + \in L(N_2) \in$$

$$= \in R_1 \in + \in R_2 \in$$

$$= R_1 + R_2$$

Since, N is FA, hence $L(N)$ is a regular set (language). Therefore, $R_1 + R_2$ is a regular set.

2. **Concatenation :** If R_1 and R_2 are two regular sets, then concatenation of these denoted by R_1R_2 is also a regular set.

Proof : Let R_1 and R_2 be recognized by NFA N_1 and N_2 respectively as shown in Figure 2(a) and Figure 2(b).

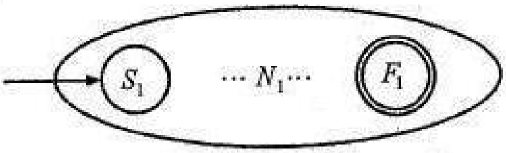


FIGURE 2(a) NFA for regular set R_1

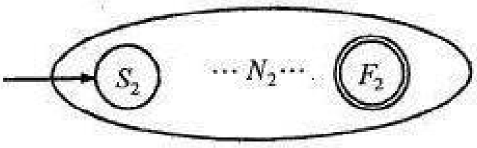


FIGURE 2(b) NFA for regular set R_2

We construct a new NFA N based on concatenation of N_1 and N_2 as shown in Figure2(c).

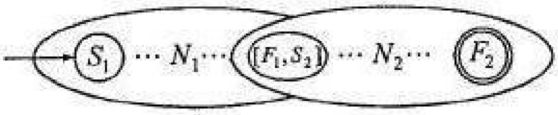


FIGURE 2(c) NFA for regular set R_1R_2

Now,

$L(N)$ = Regular set accepted by N_1 followed by regular set accepted by $N_2 = R_1R_2$

Since, $L(N)$ is a regular set, hence R_1R_2 is also a regular set.

3. **Kleene Closure :** If R is a regular set, then Kleene closure of this denoted by R^* is also a regular set.

Proof : Let R is accepted by NFA N shown in Figure 3(a).

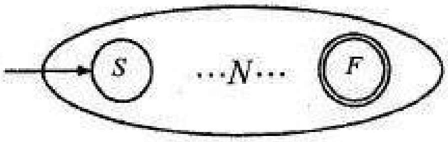


FIGURE 3(a) NFA for regular set R

We construct a new NFA based on NFA N as shown in Figure 3(b).

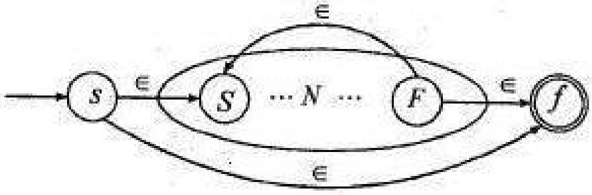


FIGURE 3(b) NFA for regular expression for R^*

Now,

$$L(N) = \{\in, R, RR, RRR, ...\}$$
$$= L^*$$

Since, $L(N)$ is a regular set, therefore R^* is a regular set.

4. Complement : If R is a regular set on some alphabet Σ, then complement of R is denoted by $\Sigma^* - R$ or $\overline{R}$ is also a regular set.

Proof : Let R be accepted by NFA $N = (Q, \Sigma, \delta, s, F)$. It means, $L(N) = R$. N is shown in Figure 4(a).

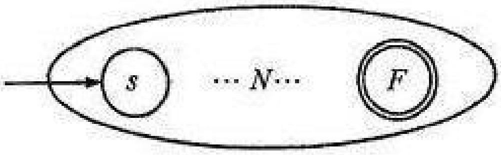


FIGURE 4(a) NFA for regular set R

We construct a new NFA N' based on N as follows :

(a) Change all final states to non-final states.

(b) Change all non-final states to final states.

N' is shown in Figure 4(b)

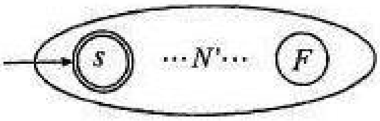


FIGURE 4 (b) NFA

Now,

$L(N') =$ {All the words which are not accepted by NFA N}

= { All the rejected words by NFA N}

$= \Sigma^* - R$

Since, $L(N')$ is a regular set, therefore $(\Sigma^* - R)$ is a regular set.

5. **Transpose :** If R is a regular set, then the transpose denoted by R^T, is also a regular set.

Proof : Let R be accepted by NFA $N = (Q, \Sigma, \delta, s, F)$ as shown in Figure 5(a).

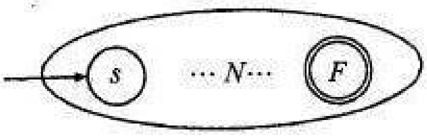


FIGURE 5 (a) NFA *N* for regular set *R*

If w is a word in R, then transpose (reverse) is denoted by w^T.

Let $w = a_1 a_2 \ldots a_n$

Then $w^T = a_n a_{n-1} \ldots . a_1$

We construct a new N' based on N using following rules :

(a) Change the all final states into non-final states and merge all these into one state and make it initial state.

(b) Change initial state to final state.

(c) Reverse the direction of all edges.

N' is shown in Figure5 (b)

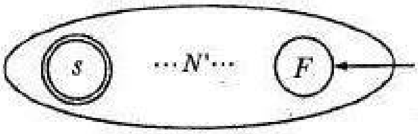


FIGURE 5(b) NFA *N'* for regular set R^T

Let $w = a_1a_2 \dots a_n$ be a word in R, then it is recognized by N and

$w^T = a_n a_{n-1} \dots a_1$ is recognized by N' as shown in Figure5 (b)

In general, we say that if a word w in R is accepted by N, and then N' accepts w^T.

Since, $L(N')$ is a regular set containing all w^T; it means, $L(N') = R^T$.

Thus, R^T is a regular set.

6. **Intersection :** if R_1 and R_2 are two regular sets over Σ, then intersection of these denoted by $R_1 \cap R_2$ is also a regular set.

Proof : By De Morgan's law for two sets A and B over R,

$A \cap B = R^* - ((R^* - A) \cup (R^* - B))$

So, $R_1 \cap R_2 = \Sigma^* - ((\Sigma^* - R_1) \cup (\Sigma^* - R_2))$

Let $R_3 = (\Sigma^* - R_1)$ and $R_4 = (\Sigma^* - R_2)$

So, R_3 and R_4 are regular sets as these are complement of R_1 and R_2.

Let $R_5 = R_3 \cup R_4$

So, R_5 is a regular set because it is the union of two regular sets R_3 and R_4.

Let $R_6 = \Sigma^* - R_5$

So, R_6 is a regular set because it is the complement of regular set R_5.

Therefore, intersection of two regular sets is also regular set.

REGULAR GRAMMARS

After going through this chapter, you should be able to understand :

- Regular Grammar
- Equivalence between Regular Grammar and FA
- Interconversion

4.1 REGULAR GRAMMAR

Definition : The grammar G = (V, T, P, S) is said to be regular grammar iff the grammar is right linear or left linear.

A grammar G is said to be right linear if all the productions are of the form

$$A \to wB \quad \text{and / or} \quad A \to w \ \text{ where } A, B \in V \text{ and } w \in T^*.$$

A grammar G is said to be left linear if all the productions are of the form

$$A \to Bw \quad \text{and / or} \quad A \to w \ \text{ where } A, B \in V \text{ and } w \in T^*.$$

Example 1 : The grammar

$S \to aaB \mid bbA \mid \in$
$A \to aA \mid b$
$B \to bB \mid a \mid \in$

is a right linear grammar. Note that $\in$ and string of terminals can appear on RHS of any production and if non - terminal is present on R. H. S of any production, only one non - terminal should be present and it has to be the right most symbol on R. H. S.

Example 2 :

The grammar

$S \to Baa \mid Abb \mid \in$
$A \to Aa \mid b$
$B \to Bb \mid a \mid \in$

is a left linear grammar. Note that $\in$ and string of terminals can appear on RHS of any production and if non - terminal is present on L. H. S of any production, only one non - terminal should be present and it has to be the left most symbol on L. H. S.

Example 3 :

Consider the grammar

$S \rightarrow aA$
$A \rightarrow aB \mid b$
$B \rightarrow Ab \mid a$

In this grammar, each production is either left linear or right linear. But, the grammar is not either left linear or right linear. Such type of grammar is called linear grammar. So, a grammar which has at most one non terminal on the right side of any production without restriction on the position of this non - terminal (note the non - terminal can be leftmost or right most) is called linear grammar.

Note that the language generated from the regular grammar is called regular language. So, there should be some relation between the regular grammar and the FA, since, the language accepted by FA is also regular language. So, we can construct a finite automaton given a regular grammar.

4.2 FA FROM REGULAR GRAMMAR

Theorem : Let G = (V, T, P, S) be a right linear grammar. Then there exists a language L(G) which is accepted by a FA. i. e., the language generated from the regular grammar is regular language.

Proof : Let $V = (q_0, q_1, ...)$ be the variables and the start state $S = q_0$ Let the productions in the grammar be

$$
\begin{aligned}
q_0 &\rightarrow x_1 q_1 \\
q_1 &\rightarrow x_2 q_2 \\
q_3 &\rightarrow x_3 q_3 \\
&. \\
&. \\
&. \\
q_n &\rightarrow x_n q_n
\end{aligned}
$$

Assume that the language L(G) generated from these productions is w. Corresponding to each production in the grammar we can have a equivalent transitions in the FA to accept the string w. After accepting the string w, the FA will be in the final state. The procedure to obtain FA from these productions is given below :

Step 1 : q_0 which is the start symbol in the grammar is the start state of FA.

Step 2 : For each production of the form

$$q_i \rightarrow wq_j$$

the corresponding transition defined will be

$$\delta^*(q_i, w) = q_j;$$

Step 3 : For each production of the form $q_i \rightarrow w$

the corresponding transition defined will be $\delta^*(q_i, w) = q_f$,where q_f is the final state,

As the string $w \in L(G)$ is also accepted by FA, by applying the transitions obtained from step1 through step3, the language is regular. So, the theorem is proved.

Example 1 : Construct a DFA to accept the language generated by the following grammar

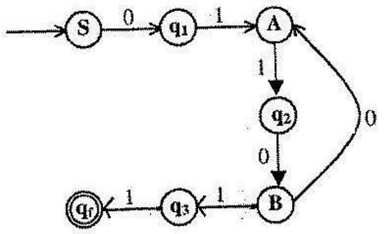
$$S \rightarrow 01A$$
$$A \rightarrow 10B$$
$$B \rightarrow 0A \mid 11$$

Solution :

Note that for each production of the form A → wB, the corresponding transition will be $\delta(A, w) = B$. Also , for each production $A \rightarrow w$, we can introduce the transition $\delta(A, w) = q_f$ where q_f is the final state. The transitions obtained from grammar G is shown using the following table :

Productions	Transitions
S → 01A	$\delta(S,\ 01) = A$
A → 10B	$\delta(A,\ 10) = B$
B → 0A	$\delta(B,\ 0) = A$
B → 11	$\delta(B,\ 11) = q_f$

The FA corresponding to the transitions obtained is shown below :



So, the DFA $M = (Q, \Sigma, \delta, q_0, A)$ where

$Q = \{ S, A, B, q_f, q_1, q_2, q_3\}$, $\Sigma = \{0,1\}$

$q_0 = S$, $A = \{q_f\}$

δ is as obtained from the above table.

The additional vertices introduced are q_1, q_2, q_3.

Example 2 : Construct a DFA to accept the language generated by the following grammar .

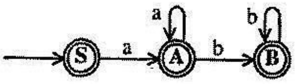
$$\begin{aligned} S &\rightarrow aA \mid \in \\ A &\rightarrow aA \mid bB \mid \in \\ B &\rightarrow bB \mid \in \end{aligned}$$

Solution :

Note that for each production of the form $A \rightarrow wB$, the corresponding transition will be $\delta(A, w) = B$. Also , for each production $A \rightarrow w$, we can introduce the transition $\delta(A, w) = q_f$ where q_f is the final state. The transitions obtained from grammar G is shown using the following table :

Productions	Transitions
S → aA	$\delta(S,a) = A$
S → ∈	S is the final state
A → aA	$\delta(A,a) = A$
A → bB	$\delta(A,b) = B$
A → ∈	A is the final state
B → bB	$\delta(B, b) = B$
B → ∈	B is the final state.

Note : For each transition of the form $A \to \in$, make A as the final state.
The FA corresponding to the transitions obtained is shown below :



So, the DFA $M = (Q, \Sigma, \delta, q_0, A)$ where

$Q = \{ S, A, B \}$, $\Sigma = \{ a, b \}$

$q_0 = S$, $A = \{ S, A, B \}$

δ is as obtained from the above table .

4.3 REGULAR GRAMMAR FROM FA

Theorem : Let $M = (Q, \Sigma, \delta, q_0, A)$ be a finite automaton. If L is the regular language accepted by FA, then there exists a right linear grammar G = (V, T, P, S) so that L = L(G).

Proof : Let $M = (Q, \Sigma, \delta, q_0, A)$ be a finite automata accepting L where

$Q = \{q_0, q_1, ... q_n\}$

$\Sigma = \{a_1, a_2, ... a_m\}$

A regular grammar G = (V, T, P, S) can be constructed where

$V = \{ q_0, q_1, ... q_n \}$

$T = \Sigma$

$S = q_0$

The productions P from the transitions can be obtained as shown below :

Step 1 : For each transition of the form $\delta(q_i, a) = q_j$

the corresponding production defined will be $q_i \to aq_j$

Step 2 : If $q \in A$ i. e., if q is the final state in FA, then introduce the production

$q \to \in$

As these productions are obtained from the transitions defined for FA, the language accepted by FA is also accepted by the grammar.

Unit-3

REGULAR GRAMMARS

After going through this chapter, you should be able to understand :

- Regular Grammar
- Equivalence between Regular Grammar and FA
- Interconversion

4.1 REGULAR GRAMMAR

Definition : The grammar G = (V, T, P, S) is said to be regular grammar iff the grammar is right linear or left linear.

A grammar G is said to be right linear if all the productions are of the form

$A \rightarrow wB$ and / or $A \rightarrow w$ where $A, B \in V$ and $w \in T^*$.

A grammar G is said to be left linear if all the productions are of the form

$A \rightarrow Bw$ and / or $A \rightarrow w$ where $A, B \in V$ and $w \in T^*$.

Example 1 : The grammar

$S \rightarrow aaB \mid bbA \mid \in$

$A \rightarrow aA \mid b$

$B \rightarrow bB \mid a \mid \in$

is a right linear grammar. Note that $\in$ and string of terminals can appear on RHS of any production and if non - terminal is present on R. H. S of any production, only one non - terminal should be present and it has to be the right most symbol on R. H. S.

Example 2 :

The grammar

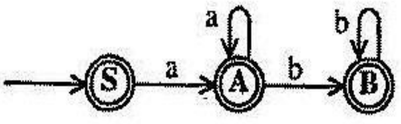
$S \rightarrow Baa \mid Abb \mid \in$

$A \rightarrow Aa \mid b$

$B \rightarrow Bb \mid a \mid \in$

is a left linear grammar. Note that $\in$ and string of terminals can appear on RHS of any production and if non - terminal is present on L. H. S of any production, only one non - terminal should be present and it has to be the left most symbol on L. H. S.

Note : For each transition of the form $A \to \in$, make A as the final state.
The FA corresponding to the transitions obtained is shown below :



So, the DFA $M = (Q, \Sigma, \delta, q_0, A)$ where

$Q = \{S, A, B\}$, $\Sigma = \{a, b\}$

$q_0 = S$, $A = \{S, A, B\}$

δ is as obtained from the above table .

4.3 REGULAR GRAMMAR FROM FA

Theorem : Let $M = (Q, \Sigma, \delta, q_0, A)$ be a finite automaton. If L is the regular language accepted by FA, then there exists a right linear grammar G = (V, T, P, S) so that L = L(G).

Proof : Let $M = (Q, \Sigma, \delta, q_0, A)$ be a finite automata accepting L where

$Q = \{q_0, q_1, ... q_n\}$

$\Sigma = \{a_1, a_2, ..., a_m\}$

A regular grammar G = (V, T, P, S) can be constructed where

$V = \{q_0, q_1, ...q_n\}$

$T = \Sigma$

$S = q_0$

The productions P from the transitions can be obtained as shown below :

Step 1 : For each transition of the form $\delta(q_i, a) = q_j$

the corresponding production defined will be $q_i \to aq_j$

Step 2 : If $q \in A$ i. e., if q is the final state in FA, then introduce the production

$$q \to \in$$

As these productions are obtained from the transitions defined for FA, the language accepted by FA is also accepted by the grammar.

CONTEXT FREE GRAMMARS

After going through this chapter, you should be able to understand :

- Context free grammars
- Left most and Rightmost derivation of strings
- Derivation Trees
- Ambiguity in CFGs
- Minimization of CFGs
- Normal Forms (CNF & GNF)
- Pumping Lemma for CFLs
- Enumeration properties of CFLs

5.1 CONTEXT FREE GRAMMARS

A grammar $G = (V, T, P, S)$ is said to be a CFG if the productions of G are of the form :

$$A \to \alpha, \text{ where } \alpha \in (V \cup T)^*$$

The right hand side of a CFG is not restricted and it may be null or a combination of variables and terminals. The possible length of right hand sentential form ranges from 0 to ∞ i.e., $0 \leq |\alpha| \leq \infty$.

As we know that a CFG has no context neither left nor right. This is why, it is known as CONTEXT - FREE. *Many programming languages have recursive structure that can be defined by CFG's.*

Example 1 : Consider the grammar $G = (V, T, P, S)$ having productions :
$S \to aSa \mid bSb \mid \in$. Check the productions and find the language generated.

Solution :

Let $P_1 : S \to aSa$ (RHS is terminal variable terminal)
$P_2 : S \to bSb$ (RHS is terminal variable terminal)
$P_3 : S \to \in$ (RHS is null string)

Since, all productions are of the form $A \to \alpha$, where $\alpha \in (V \cup T)^*$, hence G is a CFG.

So, the final grammar to generate the language $L = \{ w \mid n_a(w) = n_b(w) \}$ is G = (V, T, P, S) where

$$V = \{ S \} \quad , \quad T = \{ a, b \}$$

$$\begin{aligned} P = \{ \; & S \to \in \\ & S \to aSb \\ & S \to bSa \\ & S \to SS \\ \} \; & \end{aligned}$$

S is the start symbol

5 . 2 LEFTMOST AND RIGHTMOST DERIVATIONS

Leftmost derivation :

If $G = (V, T, P, S)$ is a CFG and $w \in L(G)$ then a derivation $S \underset{L}{\overset{*}{\Rightarrow}} w$ is called leftmost derivation if and only if all steps involved in derivation have leftmost variable replacement only.

Rightmost derivation :

If $G = (V, T, P, S)$ is a CFG and $w \in L(G)$, then a derivation $S \underset{R}{\overset{*}{\Rightarrow}} w$ is called rightmost derivation if and only if all steps involved in derivation have rightmost variable replacement only.

Example 1 : Consider the grammar $S \to S + S | \; S * S | a | b$. Find leftmost and rightmost derivations for string $w = a * a + b$.

Solution :

Leftmost derivation for $w = a * a + b$

$S \underset{L}{\Rightarrow} S * S$ (Using $S \to S * S$)

$\underset{L}{\Rightarrow} a * S$ (The first left hand symbol is a, so using $S \to a$)

$\underset{L}{\Rightarrow} a * S + S$ (Using $S \to S + S$, in order to get $a + b$)

$\underset{L}{\Rightarrow} a * a + S$ (Second symbol from the left is a, so using $S \to a$)

$\underset{L}{\Rightarrow} a * a + b$ (The last symbol from the left is b, so using $S \to b$)

Rightmost derivation for $w = a * a + b$

$S \underset{R}{\Rightarrow} S * S$ (Using $S \to S * S$)

$\underset{R}{\Rightarrow} S * S + S$ (Since, in the above sentential form second symbol from the right is * so, we can not use $S \to a|b$. Therefore, we use $S \to S + S$)

$\underset{R}{\Rightarrow} S * S + b$ (Using $S \to b$)

$\underset{R}{\Rightarrow} S * a + b$ (Using $S \to a$)

$\underset{R}{\Rightarrow} a * a + b$ (Using $S \to a$)

Example 2 : Consider a CFG $S \to bA| aB,\ A \to aS| aAA| a,\ B \to bS| aBB| b$. Find leftmost and rightmost derivations for $w = aaabbabbba$.

Solution :

Leftmost derivation for $w = aaabbabbba$:

$S \Rightarrow aB$ (Using $S \to aB$ to generate first symbol of w)

$\Rightarrow aaBB$ (Since, second symbol is a, so we use $B \to aBB$)

$\Rightarrow aaaBBB$ (Since, third symbol is a, so we use $B \to aBB$)

$\Rightarrow aaabBB$ (Since fourth symbol is b, so we use $B \to b$)

$\Rightarrow aaabbB$ (Since, fifth symbol is b, so we use $B \to b$)

$\Rightarrow aaabbaBB$ (Since, sixth symbol is a, so we use $B \to aBB$)

$\Rightarrow aaabbabB$ (Since, seventh symbol is b, so we use $B \to b$)

$\Rightarrow aaabbabbS$ (Since, eighth symbol is b, so we use $B \to bS$)

$\Rightarrow aaabbabbbA$ (Since, ninth symbol is b, so we use $S \to bA$)

$\Rightarrow aaabbabbba$ (Since, the tenth symbol is a, so using $A \to a$)

Rightmost derivation for $w = aaabbabbba$

$S \Rightarrow aB$ (Using $S \to aB$ to generate first symbol of w)

$\Rightarrow aaBB$ (We need a as the rightmost symbol and second symbol from the left side, so we use $B \to aBB$)

$\Rightarrow aaBbS$ (We need a as rightmost symbol and this is obtained from A only, we use $B \to bS$)

$\Rightarrow aaBbbA$ (Using $S \to bA$)

$\Rightarrow aaBbba$ (Using $A \to a$)

$\Rightarrow aaaBBbba$ (We need b as the fourth symbol from the right)

$\Rightarrow aaaBbbba$ (Using $B \to b$)

$\Rightarrow aaabSbbba$ (Using $B \to bS$)

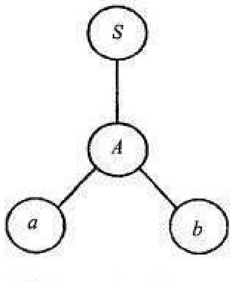


Figure (c) Parse tree for $w = ab$

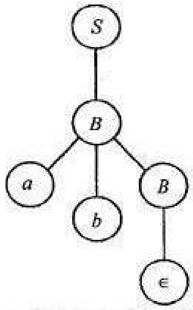


Figure (d) Parse tree for $w = ab$

So, the given grammar is ambiguous.

5.4.1 Removal of Ambiguity

5.4.1.1 Left Recursion

A grammar can be changed from one form to another accepting the same language. If a grammar has left recursive property, it is undesirable and left recursion should be eliminated. The left recursion is defined as follows.

Definition : A grammar G is said to be left recursive if there is some non terminal A such that $A \Rightarrow^+ A\alpha$. In other words, in the derivation process starting from any non - terminal A, if a sentential form starts with the same non - terminal A, then we say that the grammar is having left recursion.

Elimination of Left Recursion

The left recursion in a grammar G can be eliminated as shown below. Consider the A - production of the form $A \to A\alpha_1|A\alpha_2|A\alpha_3.........A\alpha_n|\beta_1|\beta_2|\beta_3 ...... \beta_m$

where β_i's do not start with A. Then the A productions can be replaced by

$$A \to \beta_1 A^1 | \beta_2 A^1 |\beta_3 A^1| ....\beta_m A^1$$

$$A^1 \to \alpha_1 A^1 | \alpha_2 A^1 |\alpha_3 A^1| ....| \alpha_n A^1 | \epsilon$$

Note that α_i's do not start with A^1.

Example 1 : Eliminate left recursion from the following grammar

$$E \to E + T \mid T$$

$$T \to T * F \mid F$$

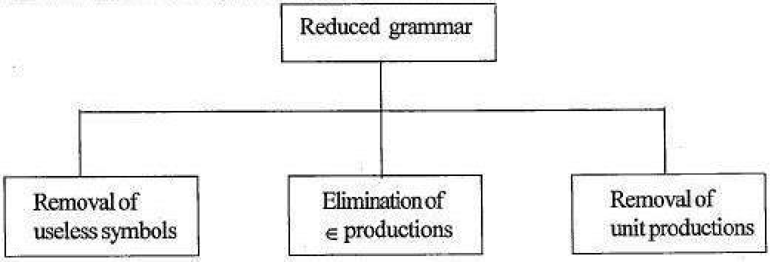
$$F \to (E) \mid id$$

5.5 MINIMIZATION OF CFGs

As we have seen various languages can effectively be represented by context free grammar. All the grammars are not always optimized. That means grammar may consists of some extra symbols (non - terminals). Having extra symbols unnecessary increases the length of grammar. Simplification of grammar means reduction of grammar by removing useless symbols. The properties of reduced grammar are given below :

1. Each variable (i. e. non - terminal) and each terminal of G appears in the derivation of some word in L.
2. There should not be any production as $X \to Y$ where X and Y are non - terminals.
3. If $\in$ is not in the language L then there need not be the production $X \to \in$.

We see the reduction of grammar as shown below :


Reduced grammar
Removal of useless symbols
Elimination of ∈ productions
Removal of unit productions

5.5.1 Removal of useless symbols

Definition : A symbol X is useful if there is a derivation of the form

$$S \Rightarrow^* \alpha X \beta \Rightarrow^* w$$

Otherwise, the symbol X is useless. Note that in a derivation, finally we should get string of terminals and all these symbols must be reachable from the start symbol S. Those symbols and productions which are not at all used in the derivation are useless.

Theorem 5.5.1 : Let G = (V, T, P, S) be a CFG. We can find an equivalent grammar $G_1 = (V_1, T_1, P_1, S)$ such that for each A in $(V_1 \cup T_1)$ there exists α and β in $(V_1 \cup T_1)^*$ and x in T^+ for which $S \Rightarrow^* \alpha A \beta \Rightarrow^* x$.

P_1	T_1	V_1
-	-	S
S → a | Bb | Aa	a, b	S, A, B
A → a B	a, b	S, A, B
B → a | A a	a, b	S, A, B

The resulting grammar $G_1 = (V_1, T_1, P_1, S)$ where

V_1 = { S, A, B }

T_1 = { a, b }

P_1 = {

S → a | Bb | aA
A → aB
B → a | Aa

} S is the start symbol

such that each symbol X in $(V_1 \cup T_1)$ has a derivation of the form $S \Rightarrow^* \alpha X \beta \Rightarrow^* w$.

5.5.2 Eliminating ∈ - productions

A production of the form $A \rightarrow \in$ is undesirable in a CFG, unless an empty string is derived from the start symbol. Suppose, the language generated from a grammar G does not derive any empty string and the grammar consists of ∈- productions. Such ∈ - productions can be removed. An ∈ - production is defined as follows :

Definition 1 : Let G = (V, T, P, S) be a CFG. A production in P of the form

$$A \rightarrow \in$$

is called an ∈ - production or NULL production. After applying the production the variable A is erased. For each A in V, if there is a derivation of the form

$$A \Rightarrow^* \in$$

then A is a nullable variable.

Example : Consider the grammar

S → ABCa | bD
A → BC | b
B → b | ∈

Step 2 : Construction of productions P_1. Add a non $\in$- production in P to P_1. Take all the combinations of nullable variables in a production, delete subset of nullable variables one by one and add the resulting productions to P_1.

Productions			Resulting productions (P_1)
S	→	BAAB	S → BAAB | AAB | BAB | BAA | AB | BB | BA | AA | A | B
A	→	0A 2	A → 0A2 | 02
A	→	2A0	A → 2A0 | 20
B	→	AB	B → AB | B | A
B	→	1 B	B → 1 B | 1

We can delete the productions of the form A → A. In P_1, the production $B \to B$ can be deleted and the final grammar obtained after eliminating $\in$ -productions is shown below.

The grammar $G_1 = (V_1, T_1, P_1, S)$ where

V_1 = { S, A, B, C, D }

T_1 = { a, b, c, d }

P_1 = { S → BAAB | AAB | BAB | BAA| AB | BB | BA | AA | A | B
A → 0A2 | 02 | 2A0 | 20
B → AB | A | 1B | 1
} S is the start symbol

5.5.3 Eliminating unit productions

Consider the production $A \to B$. The left hand side of the production and right hand side of the production contains only one variable. Such productions are called unit productions. Formally, a unit production is defined as follows.

Definition : Let G = (V, T, P, S) be a CFG. Any production in G of the form

$$A \to B$$

where A, $B \in V$ is a unit production.
In any grammar, the unit productions are undesirable. This is because one variable is simply replaced by another variable.

In a CFG, there is no restriction on the right hand side of a production. The restrictions are imposed on the right hand side of productions in a CFG resulting in normal forms. The different normal forms are :

1. Chomsky Normal Form (CNF)
2. Greiback Normal Form (GNF)

5.6.1 Chomsky Normal Form (CNF)

Chomsky normal form can be defined as follows.

Non - terminal → Non - terminal.Non - terminal
Non - terminal → terminal

The given CFG should be converted in the above format then we can say that the grammar is in CNF. Before converting the grammar into CNF it should be in reduced form. That means remove all the useless symbols, $\in$ productions and unit productions from it. Thus this reduced grammar can be then converted to CNF.

Definition :

Let G = (V, T, P, S) be a CFG. The grammar G is said to be in CNF if all productions are of the form

$$A \rightarrow BC$$

or

$$A \rightarrow a$$

where A, B and $C \in V$ and $a \in T$.

Note that if a grammar is in CNF, the right hand side of the production should contain two symbols or one symbol. If there are two symbols on the right hand side those two symbols must be non - terminals and if there is only one symbol, that symbol must be a terminal.

Theorem 5.6.1 : Let G = (V, T, P, S) be a CFG which generates context free language without $\in$. We can find an equivalent context free grammar $G_1 = (V_1, T, P_1, S)$ in CNF such that $L(G) = L(G_1)$ i. e., all productions in G_1 are of the form

$$A \rightarrow BC$$

or

$$A \rightarrow a$$

Thus, from (7), (8) and (9), the resultant grammar becomes :

$$
\begin{aligned}
&S \rightarrow V_1\ S \mid V_2V_5V_6 \mid a \mid b \\
&V_1 \rightarrow - \\
&V_2 \rightarrow [\\
&V_5 \rightarrow SV_3 && \text{.....(C)} \\
&V_6 \rightarrow SV_4 \\
&V_3 \rightarrow \uparrow \\
&V_4 \rightarrow]
\end{aligned}
$$

Now, in the resultant grammar (C), following is the production which is not in the form of CNF:

$$S \rightarrow V_2 V_5 V_6$$

We can write this production as :

$$
\begin{aligned}
&S \rightarrow V_2 V_7 && \text{.....(10)} \\
&V_7 \rightarrow V_5V_6 && \text{.....(11)}
\end{aligned}
$$

Thus, from (10) and (11), the resultant grammar becomes :

$$
\begin{aligned}
&S \rightarrow V_1S \mid V_2V_7 \mid a \mid b \\
&V_1 \rightarrow - \\
&V_2 \rightarrow [\\
&V_7 \rightarrow V_5\ V_6 && \text{.....(D)} \\
&V_5 \rightarrow\ SV_3 \\
&V_6 \rightarrow\ SV_4 \\
&V_3 \rightarrow \uparrow \\
&V_4 \rightarrow\ ]
\end{aligned}
$$

Thus, the resultant grammar (D) is in the form of CNF, which is the required solution.

5.6.2 Greibach Normal form (GNF)

Greibach normal form can be defined as follows :

$$\boxed{\text{Non - terminal} \rightarrow \text{one terminal. Any number of non - terminals}}$$

Example :

$S \rightarrow aA$	is in GNF
$S \rightarrow a$	is in GNF

From the subtree shown in figure (b), we get $S \overset{*}{\Rightarrow} aaS \in$ or $S \overset{*}{\Rightarrow} z_3 \; S \; z_4$ and considering the subtree shown in figure(c), we get $S \overset{*}{\Rightarrow} a$ or $S \overset{*}{\Rightarrow} z_2$.

The subtree shown in figure (b) can be added as many times as we like in the parse tree shown in figure (a). So, $S \overset{*}{\Rightarrow} z_3^i \; S \; z_4^i \overset{*}{\Rightarrow} z_3^i z_2 z_4^i$

Therefore, string z can be written as $uz_3z_2z_4y$ for some u and y substrings of z. The substrings z_3 and z_4 can be pumped as many times as we like. Replacing z_3, z_2 and z_4 by v, w and x respectively, we get z = uvwxy and $S \overset{*}{\Rightarrow} uv^iwx^iy$ for some i = 0, 1, 2, ...
Hence, the statement of theorem is proved.

Application of Pumping Lemma for CFLs

We use the pumping lemma to prove certain languages are not CFL. We proceed as we have seen in application of pumping lemma for regular sets and get contradiction. The result of this lemma is always negative.

Procedure for Proving Language is not Context - free

The following steps are considered to show a given language is not context - free.

Step 1 :

Suppose that L is context - free. Let 1 be the natural number obtained by using pumping lemma.

Step 2 :

Choose a string $x \in L$ such that $|x| \geq 1$ using pumping lemma principle write z = uvwxy.

Step 3 :

Find suitable i so that $uv^iwx^iy \notin L$. This is a contradiction. So L is not context - free.

Case 2 :

$v \in a^+$ and $x \in c^*$. Let $v = a^p$ and pq=n!. Pumping v and x, $(q+1)$ times, we get : $z' = uv^{q+1}wx^{q+1}y$.

In z', no. of a's will be $n - p + n! + p = n! + n$.
No.of b's in z' will remain n! + n. Hence, no. of a's = no. of b's in z'.
Similarly, in other cases, we can arrive at strings not as per specification of L.
Hence, L is not context free.

5.8 CLOSURE PROPERTIES OF CFLs

The closure properties that hold for regular languages do not always hold for context free languages. Consider those operations which preserve CFL.

The purpose of these operations are to prove certain languages are CFL and certain languages are not CFL.

Context-free languages are closed under following properties.

1. Union
2. Concatenation and
3. Kleene Closure (Context-free languages **may** or **may not** close under following properties)
4. Intersection
5. Complementation

Theorem 5.8.1 : If L_1 and L_2 are two CFLs, then union of L_1 and L_2 denoted by $L_1 + L_2$ or $L_1 \cup L_2$ is also a CFL.

Proof :

Let CFG $G_1 = (V_1, T_1, P, S)$ generates L_1 and CFG $G_2 = (V_2, T_2, P, S)$ generates L_2 and $G = (V, T, P, S)$ generates $L = L_1 + L_2$.

We construct G as follows :

Step 1 : Rename the variables of CFG G_1

If $V_1 = \{S, A, B, ..., X\}$, then the renamed variables are $\{S_1, A_1, B_1, ... X_1\}$. This modification should be reflected in productions also.

.

Step 2 : Rename the variables of CFG G_2

If $V_2 = \{S, A, B, ... X\}$, then the renamed variables are $\{S_2, A_2, B_2 .... X_2\}$. This modification should be reflected in production also.

Step 3 : We get of the productions of G_1 and G_2 to get productions of G as follows :

$S \rightarrow S_1 \mid S_2$, where S_1 and S_2 are starting symbols of grammars G_1 and G_2 respectively and S_1 - productions and S_2 - productions remain unchanged.

$T = T_1 \cup T_2$,

$V = \{S_1, A_1, B_1, ... X_1\} \cup \{S_2, A_2, B_2, ... X_2\}$

Since, all productions of G_1 and G_2 including $S \rightarrow S_1 \mid S_2$ are in context-free form, so G is a CFG.

Language generated by *G* :

$L(G)$ = Language generated from $(S_1 \ or \ S_2)$

= Language generated from S_1 or language generated from S_2

= $L(G_1)$ or $L(G_2)$ (Since, S_1 and S_2 are starting symbols of G_1 and G_2 respectively.)

= $L_1 \ or \ L_2$ (Since, G_1 produces L_1 and G_2 produces L_2 .)

= $L_1 + L_2$

Hence, statement of the theorem is proved.

Example : Consider the CFGs $S \rightarrow aSb \mid ab$ and $S \rightarrow cSdd \mid cdd$, which generate languages L_1 and L_2 respectively. Construct grammar for $L = L_1 + L_2$.

Solution :

Let G_1 generates L_1 and G_2 generates L_2 and $G = (V, T, P, S)$ generates $L = L_1 + L_2$.

Renaming the variables of G_1 and G_2, we get

$V_1 = \{S_1\}$ and $V_2 = \{S_2\}$, where S_1 - productions are $S_1 \rightarrow aS_1b \mid ab$, and S_2 - productions are $S_2 \rightarrow cS_2dd \mid cdd$

PUSH DOWN AUTOMATA

After going through this chapter, you should be able to understand :

- Push down automata
- Acceptance by final state and by empty stack
- Equivalence of CFL and PDA
- Interconversion
- Introduction to DCFL and DPDA

6.1 INTRODUCTION

A PDA is an enhancement of finite automata (FA). Finite automata with a stack memory can be viewed as pushdown automata. Addition of stack memory enhances the capability of Pushdown automata as compared to finite automata. The stack memory is potentially infinite and it is a data structure. Its operation is based on **last - in - first - out** (LIFO). It means, the last object pushed on the stack is popped first for operation. We assume a stack is long enough and linearly arranged. We add or remove objects at the left end.

6.1.1 Model of Pushdown Automata (PDA)

A model of pushdown automata is shown in below figure. It consists of a finite tape, a reading head, which reads from the tape, a stack memory operating in LIFO fashion.

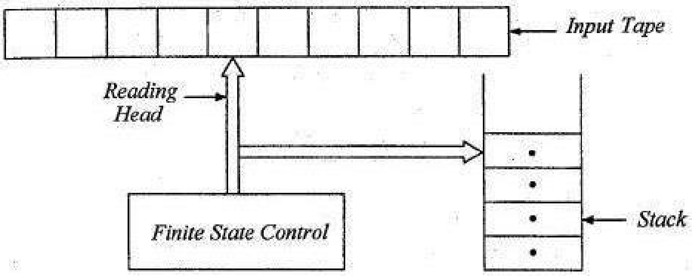


FIGURE : Model of Pushdown Automata

There are two alphabets ; one for input tape and another for stack. The stack alphabet is denoted by Γ and input alphabet is denoted by Σ. PDA reads from both the alphabets ; one symbol from the input and one symbol from the stack.

6.1.2 Mathematical Description of PDA

A pushdown automata is described by 7 - tuple $(Q,\Sigma,\Gamma,\delta,q_0,Z_0,F)$, where

1. Q is finite and nonempty set of states,
2. Σ is input alphabet,
3. Γ is finite and nonempty set of pushdown symbols,
4. δ is the transition function which maps
   From $Q \times (\Sigma \cup \{\in\}) \times \Gamma$ to **(finite subset** of) $Q \times \Gamma^*$,
5. $q_0 \in Q$, is the starting state,
6. $Z_0 \in \Gamma$, is the starting (top most or initial) stack symbol, and
7. $F \subseteq Q$, is the set of final states.

6.1.3 Moves of PDA

The move of PDA means that what are the options to proceed further after reading inputs in some state and writing some string on the stack. As we have discussed earlier that PDA is nondeterministic device having some finite number of choices of moves in each situation.

The **move** will be of two types :

1. In the first type of move, an input symbol is read from the tape, it means, the head is advanced and depending upon the topmost symbol on the stack and present state, PDA has number of choices to proceed further.
2. In the second type of move, the input symbol is not read from the tape, it means, head is not advanced and the topmost symbol of stack is used. The topmost of stack is modified without reading the input symbol. It is also known as an $\in$ - move.

Mathematically first type of move is defined as follows.

$\delta(q,a,Z)=\{(p_1,\alpha_1),(p_2,\alpha_2),...(p_n,\alpha_n)\}$, where for $1 \le i \le n, q, p_i$ are states in Q, $a \in \Sigma$, $Z \in \Gamma, and$ $\alpha_i \in \Gamma^*$.

PDA reads an input symbol a and one stack symbol Z in present state q and for any value(s) of i, enters state p_i, replaces stack symbol Z by string $\alpha_i \in \Gamma^*$, and head is advanced one cell on the tape. Now, the leftmost symbol of string α_i is assumed as the topmost symbol on the stack.

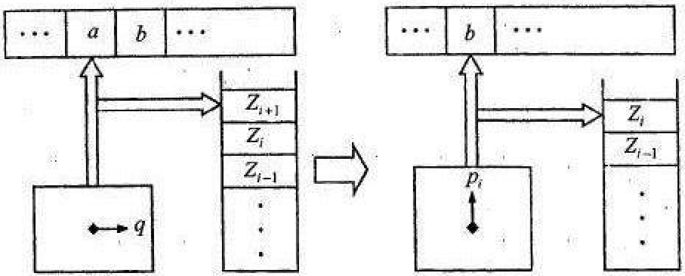
Mathematically second type of move is defined as follows.

$\delta(q,\in,Z)=\{(p_1,\alpha_1),(p_2,\alpha_2),....(p_n,\alpha_n)\}$, where for $1 \le i \le n, q, p_i$ are states in $Q, a \in \Sigma$, $Z \in \Gamma, and$ $\alpha_i \in \Gamma^*$.

PDA does not read input symbol but it reads stack symbol Z in present state q and for any value(s) of i, enters state p_i, replaces stack symbol Z by string $\alpha_i \in \Gamma^*$, and head is not advanced on the tape. Now, the leftmost symbol of string α_i is assumed as the topmost symbol on the stack.

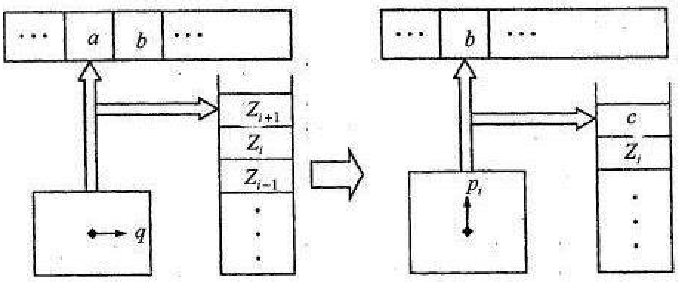
The **string** α_i be any one of the following :

1. $\alpha_i = \in$ in this case the topmost stack symbol Z_{i+1} is erased and second topmost symbol becomes the topmost symbol in the next move. It is shown in figure (a).



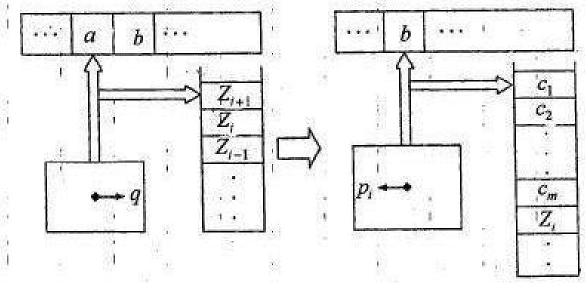
FIGURE(a): Move of PDA

2. $\alpha_i = c, c \in \Gamma$, in this case the topmost stack symbol Z_{i+1} is replaced by symbol c. It is shown in figure(b)



FIGURE(b): Move of PDA

3. $\alpha_i = c_1c_2...c_m$, in this case the topmost stack symbol Z_{i+1} is replaced by string $c_1c_2\ldots c_m$. It is shown in figure(c).



FIGURE(c): Move of PDA

6.1.4 Instantaneous Description (ID) of PDA

Let PDA $M = (Q,\Sigma,\Gamma,\delta,q_0,Z_0,F)$, then its configuration at a given instant can be defined by instantaneous description (ID). An ID includes state, remaining input string, and remaining stack string (symbols). So, an ID is (q,x,α), where $q \in Q, x \in \Sigma^*, \alpha \in \Gamma^*$.

The relation between two consecutive IDs is represented by the sign $\vdash$.

We say $(q,ax,Z\beta) \vdash_M (p,x,\alpha\beta)$ if $\delta(q,a,Z)$ contains (p,α), where $Z,\beta,\alpha \in \Gamma^*$, a may be null or $a \in \Sigma$, $p,q \in Q$ for M

The reflexive and transitive closure of the relation $\vdash_M$ is denoted by $\vdash_M^*$

Properties :

1. If $(q,x,\alpha) \vdash_M^* (p,\in,\alpha)$, where $\alpha \in \Gamma^*, x \in \Sigma^*$, and $p,q \in Q$, then for all $y \in \Sigma^*$.

   $(q,xy,\alpha) \vdash_M^* (p,y,\alpha)$,

2. If $(q,xy,\alpha) \vdash_M^* (p,y,\alpha)$, where $\alpha \in \Gamma^*, x,y \in \Sigma^*$, and $p,q \in Q$, then

   $(q,x,\alpha) \vdash_M^* (p,\in,\alpha)$, and

3. If $(q,x,\alpha) \vdash_M^* (p,\in,\beta)$, where $\alpha,\beta \in \Gamma^*, x \in \Sigma^*$, and $p,q \in Q$, then

   $(q,x,\alpha\gamma) \vdash_M^* (p,\in,\beta\gamma)$, where $\gamma \in \Gamma^*$

6.1.5 Acceptance by PDA

Let *M* be a PDA, the accepted language is represented by N(M). We defined the acceptance by PDA in two ways.

1. Let $M = (Q,\Sigma,\Gamma,\delta, q_0, Z_0,F)$, then N(M) is accepted by final state such that

   $N(M)=\{w:(q_0,w,Z_0)\overset{*}{\underset{M}{\vdash}}(q_f,\in,\beta)$, where $q \in Q$, $w \in\Sigma^*, Z_0, \beta \in\Gamma^*$, and $q_f \in F\}$

   It is similar to the acceptance by FA discussed earlier. We define some final states and the accepted language *N(M)* is the set of all input strings for which some choice of moves leads to some final state.

2. Let $M = (Q,\Sigma,\Gamma,\delta,q_0,Z_0,\phi)$, then *N(M)* is accepted by empty stack or null stack such that $N(M) = \{w:(q_0,w,Z_0)\overset{*}{\underset{M}{\vdash}}(p,\in,\in)$, where $p \in Q, w \in \Sigma^*\}$

   The language *N(M)* is the set of all input strings for which some sequence of moves causes the PDA to empty its stack.

Note : If acceptance is defined by empty stack then there is no meaning of final state and it is represented by ϕ .

Example : consider a PDA $M = (\{q_0,q_1,q_2\},\{a,c\},\{a,Z_0\},\delta,q_0,Z_0,\{q_2\})$ shown in below figure. Check the acceptability of string aacaa.

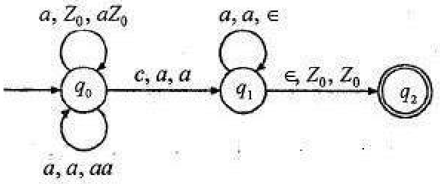


FIGURE : PDA accepting $\{a^nca^n : n \geq 1\}$

Note : *Edges are labeled with Input symbol, stack symbol, written symbol on the stack.*

Solution :

The transition function δ is defined as follows :

$\delta(q_0,a,Z_0) = \{(q_0,aZ_0)\}$,

$\delta(q_0,a,a) = \{(q_0,aa)\}$,

$\delta(q_0,c,a) = \{(q_1,a)\}$,

$\delta(q_1,a,a) = \{(q_1,\in)\}$, and

$\delta(q_1,\in,Z_0) = \{(q_2,Z_0)\}$

Following moves are carried out in order to check acceptability of string $aacaa$:

$$
\begin{aligned}
(q_0, aacaa, Z_0) &\vdash (q_0, acaa, aZ_0) \\
&\vdash (q_0, caa, aaZ_0) \\
&\vdash (q_1, aa, aaZ_0) \\
&\vdash (q_1, a, aZ_0) \\
&\vdash (q_1, \in, Z_0) \\
&\vdash (q_2, \in, Z_0)
\end{aligned}
$$

Hence, $(q_0, aacaa, Z_0) \overset{*}{\underset{M}{\vdash}} (q_2, \in, Z_0)$.

Therefore, the string **aacaa** is accepted by M.

6. 2 CONSTRUCTION OF PDA

In this section, we shall see how PDA's can be constructed.

Example 1 : Obtain a PDA to accept the language $L(M) = \{ wCw^R | \ w \in (a+b)^* \}$ where W^R is reverse of W .

Solution:

It is clear from the language $L(M) = \{ wCw^R \}$ that if $w = abb$

then reverse of w denoted by W^R will be $W^R = bba$ and the language L will be wCw^R i. e., abbCbba which is a string of palindrome.

To accept the string :

The sequence of moves made by the PDA for the string **aabCbaa** is shown below.

Initial ID

$$\begin{aligned}
(q_0,\ aabCbaa,\ Z_0) \quad & \vdash \quad (q_0,\ abCbaa,\ aZ_0)\\
& \vdash \quad (q_0,\ bCbaa,\ aaZ_0)\\
& \vdash \quad (q_0,\ Cbaa,\ baaZ_0)\\
& \vdash \quad (q_1,baa,baaZ_0)\\
& \vdash \quad (q_1,aa,aaZ_0)\\
& \vdash \quad (q_1,a,aZ_0)\\
& \vdash \quad (q_1,\in,Z_0)\\
& \vdash \quad (q_2,\ \in,\ Z_0)
\end{aligned}$$

(Final Configuration)

Since q_2 is the final state and input string is $\in$ in the final configuration, the string **aabCbaa** is accepted by the PDA.

To reject the string :

The sequence of moves made by the PDA for the string **aabCbab** is shown below.

Initial ID

$$\begin{aligned}
(q_0,\ aabCbab,\ Z_0) \quad & \vdash \quad (q_0,\ abCbab,\ aZ_0)\\
& \vdash \quad (q_0,\ bCbab,\ aaZ_0)\\
& \vdash \quad (q_0,\ Cbab,\ baaZ_0)\\
& \vdash \quad (q_1,\ bab,\ baaZ_0)\\
& \vdash \quad (q_1,\ ab,\ aaZ_0)\\
& \vdash \quad (q_1,\ b,\ aZ_0)
\end{aligned}$$

(Final Configuration)

Since the transition $\delta(q_1,\ b,\ a)$ is not defined, the string **aabCbab** is not a palindrome and the machine halts and the string is rejected by the PDA.

Example 2 : Obtain a PDA to accept the language $L = \{\ a^n\ b^n |\ n \geq 1\ \}$ by a final state.

Solution :

The machine should accept n number of a's followed by n number of b's.

6.3 DETERMINISTIC AND NONDETERMINISTIC PUSHDOWN AUTOMATA

In this section, we will discuss about the deterministic and nondeterministic behavior of pushdown automata.

6.3.1 Nondeterministic PDA (NPDA)

Like NFA, nondeterministic PDA (NPDA) has finite number of choices for its inputs. As we have discussed in the mathematical description that transition function δ which maps from $Q \times (\Sigma \cup \{\in\}) \times \Gamma$ to (**finite subset** of) $Q \times \Gamma^*$. A nondeterministic PDA accepts an input if a sequence of choices leads to some final state or causes PDA to empty its stack. Since, sometimes it has more than one choice to move further on a particular input ; it means, PDA guesses the right choice always, otherwise it will fail and will be in hang state.

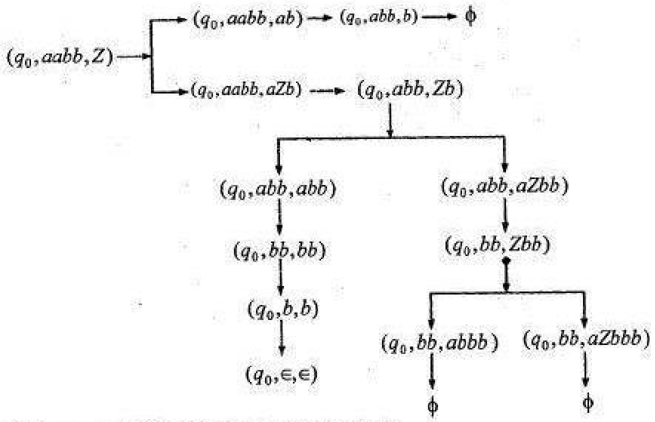
Example : consider a nondeterministic PDA $M = (\{q_0\}, \{a,b\}, \{a,b,Z\}, \delta, q_0, Z, \phi)$, for the language $L = \{a^n b^n : n \geq 1\}$, where δ is defined as follows :

$\delta(q_0, \in, Z) = \{(q_0, ab), (q_0, aZb)\}$ (Two possible moves for input $\in$ on the tape and Z on the stack),

$\delta(q_0, a, a) = \{(q_0, \in)\}$, and $\delta(q_0, b, b) = \{(q_0, \in)\}$

Check whether string $w = aabb$ is accepted or not ?

Solution : Initial configuration is $(q_0, aabb, Z)$. Following moves are possible :



Hence, $w = aabb$ is accepted by empty stack.

One thing is noticeable here that only one move sequence leads to empty store and other don't. In other words, we say that some move sequence(s) leads to accepting configuration and other lead to hang state.

6.3.2 Deterministic PDA (DPDA)

Deterministic PDA (DPDA) is just like DFA, which has *at most one choice* to move for certain input. A PDA $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ is deterministic if it satisfies both the conditions given as follows :

1. For any $q \in Q$, $a \in (\Sigma \cup \{\in\})$, and $Z \in \Gamma$, $\delta(q, a, Z)$ has at most one choice of move.
2. For any $q \in Q$, and $Z \in \Gamma$, if $\delta(q, \in, Z)$ is defined i.e. $\delta(q, \in, Z) \neq \phi$, then $\delta(q, a, Z) = \phi$ for all $a \in \Sigma$

Example : Consider a DPDA $M = (\{q_0, q_1\}, \{a, c\}, \{a, Z_0\}, \delta, q_0, Z_0, \phi)$ accepting the language $\{a^n c a^n : n \geq 1\}$, where δ is defined as follows :

$\delta(q_0, a, Z_0) = \{(q_0, aZ_0)\}$

$\delta(q_0, a, a) = \{(q_0, aa)\}$,

$\delta(q_0, c, a) = \{(q_1, a)\}$,

$\delta(q_1, a, a) = \{(q_1, \in)\}$, and $\delta(q_1, \in, Z_0) = \{(q_1, \in)\}$

Check whether the string $w = aacaa$ is accepted by empty stack or not ?

Solution :

We see that in each transition DPDA has at most one move. Initial configuration is $(q_0, aacaa, Z_0)$. Following are the possible moves.

$$(q_0, aacaa, Z_0) \rightarrow (q_0, acaa, aZ_0) \rightarrow (q_0, caa, aaZ_0) \rightarrow (q_1, aa, aaZ_0)$$

$$\downarrow$$

$$(q_1, \in, \in) \leftarrow (q_1, \in, Z_0) \leftarrow (q_1, a, aZ_0)$$

Hence, the string $w = aacaa$ is accepted by empty stack.

As we have discussed in earlier chapters that DFA and NFA are equivalent with respect to the language acceptance, but the same is not true for the PDA.

For example, language $L = \{ww^R : w \in (a \cup b)^*\}$ is accepted by nondeterministic PDA, can not by any deterministic PDA. A nondeterministic PDA can not be converted into equivalent deterministic PDA, but all DCFLs which are accepted by DPDA, are also accepted by NPDA. So, we say that deterministic PDA is a proper subset of nondeterministic PDA. Hence, the power of nondeterministic PDA is more as compared to deterministic PDA.

6.4 ACCEPTANCE OF LANGUAGE BY PDA

The language can be accepted by a Push Down Automata using two approaches.

1. **Acceptance by Final State :** The PDA accepts its input by consuming it and then it enters in the final state.

2. **Acceptance by empty stack :** On reading the input string from initial configuration for some PDA, the stack of PDA gets empty.

6.4.1 Equivalence of Empty Store and Final state acceptance

Theorem:

If $M_1 = (Q_1, \Sigma, \Gamma_1, \delta_1, p_1, Z_1, \phi)$ is a PDA accepting CFL L by empty store then there exists PDA $M_2 = (Q_2, \Sigma, \Gamma_2, \delta_2, p_2, Z_2, \{q_f\})$ which accepts L by final state.

Proof :

First we construct PDA M_2 based on PDA M_1 and then we prove that both accept L.

Step 1 : Construction of PDA M_2 based on given PDA M_1

Σ is same for both PDAs. We add a new initial state and a new final state with given PDA M_1.

So, $Q_2 = Q_1 \cup \{p_2 \cup q_f\}$

The stack alphabet Γ_2 of PDA M_2 contains one additional symbol Z_2 with Γ_1.

So, $\Gamma_2 = \Gamma_1 \cup \{Z_2\}$

The transition function δ_2 contains all the transitions of given PDA M_1 and two additional transitions (R_1 *and* R_3) as defined as follows :

$R_1 : \delta_2(p_2, \in, Z_2) = \{(p_1, Z_1 Z_2)\}$,

$R_2 : \delta_2(q, a, Z) = \delta_1(q, a, Z)$ for all (q, a, Z) in $Q_1 \times (\Sigma \cup \{\in\}) \times \Gamma_1$

(the original transitions of M_1), and

$R_3 : \delta_2(q, \in, Z_2) = \{(q_f, \in)\}$ for all $q \in Q_1$

By the R_1, M_2 moves from its initial ID $(p_2, \in, Z_2)$ to the initial ID of M_1.By R_2, M_2 uses all the transitions of M_1 after reaching the initial ID of M_1 and by using R_3 M_2 reaches the final state q_f.

The block diagram is shown in below figure.

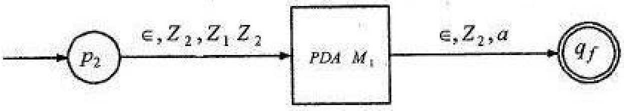


FIGURE : Block diagram of PDA M_2

Step 2 : The language accepted by PDA M_1 and PDA M_2

The behaviors of M_1 and M_2 are same except the two by $\in$ - moves defined by R_1 *and* R_3.
Let string $w \in L$ and accepted by M_1, then

$(p_1, w, Z_1) \vdash^{*}_{M_1} (q, \in, \in)$ where $q \in Q_1$ **(Result 1)**

For M_2, the initial ID is (p_2, w, Z_2) and it can be written as $(p_2, \in w \in, Z_2)$. So,

$(p_2, \in w \in, Z_2) \vdash_{M_2} (p_1, w, Z_1 Z_2)$ (This initial ID of M_1)

$\vdash^{*}_{M_2} (q, \in, Z_2)$ (by R_2 and Result 1)

$\vdash^{*}_{M_2} (q_f, \in, \alpha)$ $\alpha \in \Gamma_2^*$ (By R_3)

Thus, if M_1 accepts w, then M_2 also accepts it.
It means $L(M_2) \subseteq L(M_1)$ **(Result 2)**
Let string $w \in L$ and accepted by PDA M_2, then

$(p_2, \in w \in, Z_2)$ $\vdash_{M_2} (p_1, w, Z_1 Z_2)$ (By R_1) **(Result 3)**

$\vdash^{*}_{M_2} (q, \in, Z_2)$ (By R_2) **(Result 4)**

$\vdash_{M_2} (q_f, \in, \alpha)$ $\alpha \in \Gamma_2^*$ (By R_3)

Note : The Result 3 is the initial ID of M_1. The Result 4 shows the empty store for M_1 if symbol Z_2 is not there.

For M_1, the initial ID is (p_1, w, Z_1)

So, $(p_1, w, Z_1) \overset{*}{\underset{M_2}{\vdash}} (q, \in, \in)$, where $q \in Q_1$ (By Result 3 and Result 4) Thus, if M_2 accepts w, then M_1 also accepts it.

It means, $L(M_1) \subseteq L(M_2)$ **(Result 5)**

Therefore, $L = L(M_2) = L(M_1)$ (From Result 2 and Result 5)

Hence, the statement of theorem is proved.

Example: Consider a nondeterministic PDA $M_1 = (\{q_0\}, \{a,b\}, \{a,b,S\}, \delta, q_0, S, \phi)$ which accepts the language $L = \{a^n b^n : n \geq 1\}$ by empty store, where δ is defined as follows :

$\delta(q_0, \in, S) = \{(q_0, ab), (q_0, aSb)\}$ (Two possible moves),

$\delta(q_0, a, a) = \{(q_0, \in)\}$, and $\delta(q_0, b, b) = \{(q_0, \in)\}$

Construct an equivalent PDA M_2 which accepts L in final state and check whether string $w = aabb$ is accepted or not ?

Solution : Following moves are carried out by PDA M_1 in order to accept $w = aabb$:

$$\begin{aligned}(q_0, aabb, S) &\vdash (q_0, aabb, aSb)\\ &\vdash (q_0, abb, Sb)\\ &\vdash (q_0, abb, abb)\\ &\vdash (q_0, bb, bb)\\ &\vdash (q_0, b, b)\\ &\vdash (q_0, \in, \in)\end{aligned}$$

Hence, $(q_0, aabb, S) \overset{*}{\underset{M_1}{\vdash}} (q_0, \in, \in)$

Therefore, $w = aabb$ is accepted by M_1.

TURING MACHINES

After going through this chapter, you should be able to understand :

- Turing Machine
- Design of TM
- Computable functions
- Recursively Enumerable languages
- Church's Hypothesis & Counter machine
- Types of Turing Machines

7.1 INTRODUCTION

The Turing machine is a generalized machine which can recognize all types of languages viz, regular languages (generated from regular grammar), context free languages (generated from context free grammar) and context sensitive languages (generated from context sensitive grammar). Apart from these languages, the Turing machine also accepts the language generated from unrestricted grammar. Thus, Turing machine can accept any generalized language. This chapter mainly concentrates on building the Turing machines for any language.

7.2 TURING MACHINE MODEL

The Turing machine model is shown in below figure . It is a finite automaton connected to read - write head with the following components :

- Tape
- Read - write head
- Control unit

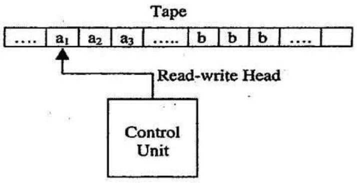


FIGURE : Turing machine model

Tape : It is a temporary storage and is divided into cells. Each cell can store the information of only one symbol. The string to be scanned will be stored from the left most position on the tape. The string to be scanned should end with infinite number of blanks.

Read - write head : The read - write head can read a symbol from where it is pointing to and it can write into the tape to where the read - write head points to.

Control Unit : The reading / writing from / to the tape is determined by the control unit. The different moves performed by the machine depends on the current scanned symbol and the current state. The read - write head can move either towards left or right i.e., movement can be on both the directions. The various moves performed by the machine are :

1. Change of state from one state to another state
2. The symbol pointing to by the read - write head can be replaced by another symbol.
3. The read - write head may move either towards left or towards right.

The Turing machine can be represented using various notations such as

- Transition table
- Instantaneous description
- Transition diagram

7.2.1 Transition Table

The table below shows the transition table for some Turing machine. Later sections describe how to obtain the transition table.

δ	Tape Symbols (Γ)				
States	a	b	X	Y	B
q_0	(q_1, X, R)	-	-	(q_3, Y, R)	-
q_1	(q_1, a, R)	(q_2, Y, L)	-	(q_1, Y, R)	-
q_2	(q_2, a, L)	-	(q_0, X, R)	(q_2, Y, L)	-
q_3	-	-	-	(q_3, Y, R)	(q_4, B, R)
q_4	-	-	-	-	-

Note that for each state q, there can be a corresponding entry for the symbol in Γ. In this table the symbols a and b are input symbols and can be denoted by the symbol Σ. Thus $\Sigma \subseteq \Gamma$ excluding the symbol B. The symbol B indicates a blank character and usually the string ends with infinite number of B's i. e., blank characters. The undefined entries indicate that there are no - transitions defined or there can be a transition to dead state. When there is a transition to the dead state, the machine halts and the input string is rejected by the machine. It is clear from the table that

$$\delta : Q \times \Gamma \text{ to } (Q \times \Gamma \times \{L, R\})$$

where $Q = \{q_0, q_1, q_2, q_3, q_4\}$; $\Sigma = \{a, b\}$

$\Gamma = \{a, b, X, Y, B\}$

q_0 is the initial state ; B is a special symbol indicating blank character

$F = \{q_4\}$ which is the final state.

Thus , a Turing Machine M can be defined as follows.

Definition : The Turing Machine $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ where

Q is set of finite states

Σ is set of input alphabets

Γ is set of tape symbols

δ is transition function $Q \times \Gamma \text{ to } (Q \times \Gamma \times \{L, R\})$

q_0 is the initial state

B is a special symbol indicating blank character

$F \subseteq Q$ is set of final states.

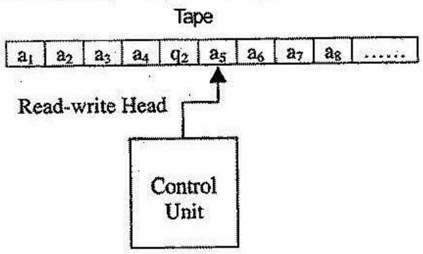
7.2.2 Instantaneous description (ID)

Unlike the ID described in PDA, in Turing machine (TM), the ID is defined on the whole string (not on the string to be scanned) and the current state of the machine.

Definition :

An ID of TM is a string in $\alpha q \beta$, where q is the current state, $\alpha \beta$ is the string made from tape symbols denoted by Γ i. e., α and $\beta \in \Gamma^*$. The read - write head points to the first character of the substring β. The initial ID is denoted by $q \alpha \beta$ where q is the start state and the read - write head points to the first symbol of α from left. The final ID is denoted by $\alpha \beta q B$ where $q \in F$ is the final state and the read - write head points to the blank character denoted by B.

Example : Consider the snapshot of a Turing machine



In this machine, each $a_i \in \Gamma$ (i. e., each a_i belongs to the tape symbol). In this snapshot, the symbol a_5 is under read - write head and the symbol towards left of a_5 i. e., q_2 is the current state. Note that, in the Turing machine, the symbol immediately towards left of the read - write head will be the current state of the machine and the symbol immediately towards right of the state will be the next symbol to be scanned. So, in this case an ID is denoted by

$$a_1a_2a_3a_4q_2a_5a_6a_7a_8.......$$

where the substring $a_1a_2a_3a_4$ towards left of the state q_2 is the left sequence, the substring $a_5a_6a_7a_8.....$ towards right of the state q_2 is the right sequence and q_2 is the current state of the machine. The symbol a_5 is the next symbol to be scanned.

Assume that the current ID of the Turing machine is $a_1a_2a_3a_4q_2a_5a_6a_7a_8......$ as shown in snapshot of example.

Suppose, there is a transition $\delta(q_2, a_5) = (q_3, b_1, R)$

It means that if the machine is in state q_2 and the next symbol to be scanned is a_5, then the machine enters into state q_3 replacing the symbol a_5 by b_1 and R indicates that the read - write head is moved one symbol towards right. The new configuration obtained is

$$a_1a_2a_3a_4b_1q_3a_6a_7a_8.....$$

This can be represented by a move as $a_1a_2a_3a_4q_2a_5a_6a_7a_8.... \vdash a_1a_2a_3a_4b_1q_3a_6a_7a_8.....$

Similarly if the current ID of the Turing machine is $a_1a_2a_3a_4q_2a_5a_6a_7a_8.....$

and there is a transition

$$\delta(q_2, a_5)=(q_1, c_1, L)$$

means that if the machine is in state q_2 and the next symbol to be scanned is a_5, then the machine enters into state q_1 replacing the symbol a_5 by c_1 and L indicates that the read - write head is moved one symbol towards left. The new configuration obtained is

$$a_1a_2a_3q_1a_4c_1a_6a_7a_8......$$

This can be represented by a move as $a_1a_2a_3a_4q_2a_5a_6a_7a_8....\ \vdash\ a_1a_2a_3q_1a_4c_1a_6a_7a_8....$

This configuration indicates that the new state is q_1, the next input symbol to be scanned is a_4. The actions performed by TM depends on

1. The current state.
2. The whole string to be scanned
3. The current position of the read - write head

The action performed by the machine consists of

1. Changing the states from one state to another
2. Replacing the symbol pointed to by the read - write head
3. Movement of the read - write head towards left or right.

7.2.3 The move of Turing Machine M can be defined as follows

Definition : Let $M = (Q,\Sigma,\Gamma,\delta,q_0,B,F)$ be a TM. Let the ID of M be $a_1a_2a_3......a_{k-1}qa_ka_{k+1}......a_n$ where $a_j \in \Gamma$ for $1 \le j \le n-1$, $q \in Q$ is the current state and a_k as the next symbol to scanned. If there is a transition $\delta(q, a_k) = (p, b, R)$ then the move of machine M will be $a_1a_2a_3......a_{k-1}qa_ka_{k+1}.....a_n \ \vdash a_1a_2a_3......a_{k-1}bpa_{k+1}.....a_n$

If there is a transition $\delta(q, a_k) = (p, b, L)$ then the move of machine M will be

$$a_1a_2a_3......a_{k-1}qa_ka_{k+1}....a_n \ \vdash a_1a_2a_3......a_{k-2}pa_{k-1}ba_{k+1}......a_n$$

7.2.4 Acceptance of a language by TM

The language accepted by TM is defined as follows.

Definition :

Let $M = (Q,\Sigma,\Gamma,\delta,q_0,B,F)$ be a TM. The language L(M) accepted by M is defined as

$$L(M) = \{w \mid q_0w \vdash^* \alpha_1\ p\ \alpha_2 \text{ where } w \in \Sigma^*, p \in F \text{ and } \alpha_1, \alpha_2 \in \Gamma^*\}$$

i.e., set of all those words w in Σ^* which causes M to move from start state q_0 to the final state p. The language accepted by TM is called recursively enumerable language.

The string w which is the string to be scanned, should end with infinite number of blanks. Initially, the machine will be in the start state q_0 with read - write head pointing to the first symbol of w from left. After some sequence of moves, if the Turing machine enters into the final state and halts, then we say that the string w is accepted by Turing machine.

7.2.5 Differences between TM and PDA

Push Down Automa :

1. A PDA is a nondeterministic finite automaton coupled with a stack that can be used to store a string of arbitrary length.
2. The stack can be read and modified only at its top.
3. A PDA chooses its next move based on its current state, the next input symbol and the symbol at the top of the stack.
4. There are two ways in which the PDA may be allowed to signal acceptance. One is by entering an accepting state, the other by emptying its stack.
5. ID consisting of the state, remaining input and stack contents to describe the "current condition" of a PDA.
6. The languages accepted by PDA's either by final state or by empty stack, are exactly the context - free languages.
7. A PDA languages lie strictly between regular languages and CSL's.

Turing Machines :

1. The TM is an abstract computing machine with the power of both real computers and of other mathematical definitions of what can be computed.
2. TM consists of a finite - state control and an infinite tape divided into cells.
3. TM makes moves based on its current state and the tape symbol at the cell scanned by the tape head.
4. The blank is one of tape symbols but not input symbol.
5. TM accepts its input if it ever enters an accepting state.
6. The languages accepted by TM's are called Recursively Enumerable (RE) languages.
7. Instantaneous description of TM describes current configuration of a TM by finite - length string.
8. Storage in the finite control helps to design a TM for a particular language.
9. A TM can simulate the storage and control of a real computer by using one tape to store all the locations and their contents.

7.3 CONSTRUCTION OF TURING MACHINE (TM)

In this section, we shall see how TMs can be constructed.

Example 1 : Obtain a Turing machine to accept the language $L = \{ 0^n 1^n \mid n \geq 1 \}$.

Solution : Note that n number of 0's should be followed by n number of 1's. For this let us take an example of the string $w = 00001111$. The string w should be accepted as it has four zeroes followed by equal number of 1's.

General Procedure :

Let q_0 be the start state and let the read - write head points to the first symbol of the string to be scanned. The general procedure to design TM for this case is shown below :

1. Replace the left most 0 by X and change the state to q_1 and then move the read - write head towards right. This is because, after a zero is replaced, we have to replace the corresponding 1 so that number of zeroes matches with number of 1's.
2. Search for the leftmost 1 and replace it by the symbol Y and move towards left (so as to obtain the leftmost 0 again). Steps 1 and 2 can be repeated.

Consider the situation

$$\begin{array}{c} XX00YY11 \\ \uparrow \\ q_0 \end{array}$$

where first two 0's are replaced by Xs and first two 1's are replaced by Ys. In this situation, the read - write head points to the left most zero and the machine is in state q_0 . With this as the configuration , now let us design the TM.

Step 1 : In state q_0 , replace 0 by X, change the state to q_1 and move the pointer towards right. The transition for this can be of the form

$$\delta(q_0,\ 0) = (q_1,\ X,\ R)$$

The resulting configuration is shown below .

$$\begin{array}{c} XXX0YY11 \\ \uparrow \\ q_1 \end{array}$$

Step 2 : In state q_1 , we have to obtain the left - most 1 and replace it by Y. For this, let us move the pointer to point to leftmost one. When the pointer is moved towards 1, the symbols encountered may be 0 and Y. Irrespective what symbol is encountered, replace 0 by 0, Y by Y, remain in state q_1 and move the pointer towards right. The transitions for this can be of the form

$$\delta(q_1,0)=(q_1,0,R)$$
$$\delta(q_1,Y)=(q_1,Y,R)$$

When these transitions are repeatedly applied, the following configuration is obtained.

$$\begin{array}{c} XXX0YY11 \\ \uparrow \\ q_1 \end{array}$$

Step 3 : In state q_1, if the input symbol to be scanned is a 1, then replace 1 by Y, change the state to q_2 and move the pointer towards left. The transition for this can be of the form

$$\delta(q_1,1)=(q_2,Y,L)$$

and the following configuration is obtained.

$$\begin{array}{c}\text{XXX0YYY1}\\ \uparrow\\ q_2\end{array}$$

Note that the pointer is moved towards left. This is because, a zero is replaced by X and the corresponding 1 is replaced by Y. Now, we have to scan for the left most 0 again and so, the pointer was move towards left.

Step 4 : Note that to obtain leftmost zero, we need to obtain right most X first. So, we scan for the right most X. During this process we may encounter Y's and 0's . Replace Y by Y, 0 by 0, remain in state q_2 only and move the pointer towards left. The transitions for this can be of the form

$$\delta(q_2,Y)=(q_2,Y,L)$$
$$\delta(q_2,0)=(q_2,0,L)$$

The following configuration is obtained

$$\begin{array}{c}\text{XXX0YYY1}\\ \uparrow\\ q_2\end{array}$$

Step 5 : Now, we have obtained the right most X. To get leftmost 0, replace X by X, change the state to q_0 and move the pointer towards right. The transition for this can be of the form

$$\delta(q_2,X)=(q_0,X,R)$$

and the following configuration is obtained

$$\begin{array}{c}\text{XXX0YYY1}\\ \uparrow\\ q_0\end{array}$$

Now, repeating the steps 1 through 5, we get the configuration shown below :

$$\begin{array}{c}\text{XXXXYYYY}\\ \uparrow\\ q_0\end{array}$$

Step 6 : In state q_0, if the scanned symbol is Y, it means that there are no more 0's. If there are no zeroes we should see that there are no 1's. For this we change the state to q_3, replace Y by Y and move the pointer towards right. The transition for this can be of the form

$$\delta(q_0,Y)=(q_3,Y,R)$$

and the following configuration is obtained

$$\begin{array}{c}\text{XXXXYYYY}\\ \uparrow \\ q_3\end{array}$$

In state q_3, we should see that there are only Ys and no more 1's. So, as we can replace Y by Y and remain in q_3 only. The transition for this can be of the form

$$\delta(q_3,Y)=(q_3,Y,R)$$

Repeatedly applying this transition, the following configuration is obtained .

$$\begin{array}{c}\text{XXXXYYYYB}\\ \uparrow \\ q_3\end{array}$$

Note that the string ends with infinite number of blanks and so, in state q_3 if we encounter the symbol B, means that end of string is encountered and there exists n number of 0's ending with n number of 1's. So, in state q_3, on input symbol B, change the state to q_4, replace B by B and move the pointer towards right and the string is accepted. The transition for this can be of the form

$$\delta(q_3,B)=(q_4,B,R)$$

The following configuration is obtained

$$\begin{array}{c}\text{XXXXYYYYBB}\\ \uparrow \\ q_4\end{array}$$

So, the Turing machine to accept the language $L=\{a^n\ b^n \mid n\geq 1\}$

is given by $M=(Q,\Sigma,\Gamma,\delta,q_0,B,F)$

where

$Q=\{q_0, q_1, q_2, q_3\}$; $\Sigma=\{0,1\}$; $\Gamma=\{0,1,X,Y,B\}$

$q_0\in Q$ is the start state of machine ; $B\in\Gamma$ is the blank symbol.

$F=\{q_4\}$ is the final state.

δ is shown below.

$$\delta(q_0,0) = (q_1,X,R)$$

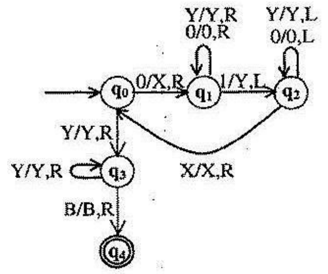
$$\delta(q_1,0)=(q_1,0,R)$$

$$\delta(q_1, Y) = (q_1, Y, R)$$
$$\delta(q_1, 1) = (q_2, Y, L)$$
$$\delta(q_2, Y) = (q_2, Y, L)$$
$$\delta(q_2, 0) = (q_2, 0, L)$$
$$\delta(q_2, X) = (q_0, X, R)$$
$$\delta(q_0, Y) = (q_3, Y, R)$$
$$\delta(q_3, Y) = (q_3, Y, R)$$
$$\delta(q_3, B) = (q_4, B, R)$$

The transitions can also be represented using tabular form as shown below.

δ	Tape Symbols (Γ)				
States	0	1	X	Y	B
q_0	(q_1, X, R)	-	-	(q_3, Y, R)	-
q_1	$(q_1, 0, R)$	(q_2, Y, L)	-	(q_1, Y, R)	-
q_2	$(q_2, 0, L)$	-	(q_0, X, R)	(q_2, Y, L)	-
q_3	-	-	-	(q_3, Y, R)	(q_4, B, R)
q_4	-	-	-	-	-

The transition table shown above can be represented as transition diagram as shown below :



To accept the string :

The sequence of moves or computations (IDs) for the string 0011 made by the Turing machine are shown below :

Initial ID

$q_0 0011$ $\vdash Xq_1 011$ $\vdash X0q_1 11$

$\vdash Xq_2 0Y1$ $\vdash q_2 X0Y1$

$\vdash Xq_0 0Y1$ $\vdash XXq_1 Y1$

$\vdash XXYq_1 1$ $\vdash XXq_2 YY$

$\vdash Xq_2 XYY$ $\vdash XXq_0 YY$

$\vdash XXYq_3 Y$ $\vdash XXYYq_3$

$\vdash XXYYBq_4$

(Final ID)

Example 2 : Obtain a Turing machine to accept the language $L(M) = \{ 0^n 1^n 2^n \mid n \geq 1\}$

Solution : Note that n number of 0's are followed by n number of 1's which in turn are followed by n number of 2's. In simple terms, the solution to this problem can be stated as follows :

Replace first n number of 0's by X's, next n number of 1's by Y's and next n number of 2's by Z's. Consider the situation where in first two 0's are replaced by X's , next immediate two 1's are replaced by Y's and next two 2's are replaced by Z's as shown in figure 1(a).

XX00YY11ZZ22	XXX0YY11ZZ22	XXX0YY11ZZ22
↑	↑	↑
q_0	q_1	q_1
(a)	(b)	(c)

FIGURE 1 : Various Configurations

Now, with figure 1(a). a as the current configuration, let us design the Turing machine. In state q_0, if the next scanned symbol is 0 replace it by X, change the state to q_1 and move the pointer towards right and the situation shown in figure 1(b) is obtained . The transition for this can be of the form

$$\delta(q_0, 0) = (q_1, X, R)$$

In state q_1, we have to search for the leftmost 1. It is clear from figure 1(b) that, when we are searching for the symbol 1, we may encounter the symbols 0 or Y. So, replace 0 by 0 , Y by Y and move the pointer towards right and remain in state q_1 only. The transitions for this can be of the form

$$\delta(q_1, 0) = (q_1, 0, R)$$

$$\delta(q_1, Y) = (q_1, Y, R)$$

The configuration shown in figure 1(c) is obtained. In state q_1, on encountering 1 change the state to q_2, replace 1 by Y and move the pointer towards right. The transition for this can be of the form

$$\delta(q_1,1)=(q_2,Y,R)$$

and the configuration shown in figure 2(a) is obtained

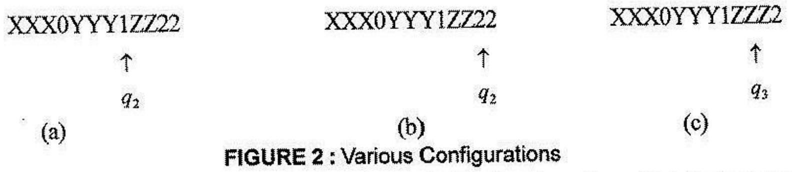


FIGURE 2 : Various Configurations

In state q_2, we have to search for the leftmost 2. It is clear from figure 2(a) that, when we are searching for the symbol 2, we may encounter the symbols 1 or Z. So, replace 1 by 1, Z by Z and move the pointer towards right and remain in state q_2 only and the configuration shown in figure 2(b) is obtained. The transitions for this can be of the form

$$\delta(q_2,1)=(q_2,1,R)$$

$$\delta(q_2,Z)=(q_2,Z,R)$$

In state q_2, on encountering 2, change the state to q_3, replace 2 by Z and move the pointer towards left. The transition for this can be of the form

$$\delta(q_2,2)=(q_3,Z,L)$$

and the configuration shown in figure 2(c) is obtained. Once the TM is in state q_3, it means that equal number of 0's, 1's and 2's are replaced by equal number of X's, Y's and Z's respectively. At this point, next we have to search for the rightmost X to get leftmost 0. During this process, it is clear from figure 2(c) that the symbols such as Z's, 1,s, Y's, 0's and X are scanned respectively one after the other. So, replace Z by Z, 1 by 1, Y by Y, 0 by 0, move the pointer towards left and stay in state q_3 only. The transitions for this can be of the form

$$\delta(q_3,Z)=(q_3,Z,L)$$

$$\delta(q_3,1)=(q_3,1,L)$$

$$\delta(q_3,Y)=(q_3,Y,L)$$

$$\delta(q_3,0)=(q_3,0,L)$$

Only on encountering X, replace X by X, change the state to q_0 and move the pointer towards right to get leftmost 0. The transition for this can be of the form

$$\delta(q_3,X)=(q_0,X,R)$$

All the steps shown above are repeated till the following configuration is obtained.

$$\begin{array}{c}\text{XXXXYYYYZZZZ}\\ \uparrow \\ q_0\end{array}$$

In state q_0, if the input symbol is Y, it means that there are no 0's . If there are no 0's we should see that there are no 1's also. For this to happen change the state to q_4, replace Y by Y and move the pointer towards right. The transition for this can be of the form

$$\delta(q_0, Y) = (q_4, Y, R)$$

In state q_4 search for only Y's, replace Y by Y, remain in state q_4 only and move the pointer towards right. The transition for this can be of the form

$$\delta(q_4, Y) = (q_4, Y, R)$$

In state q_4, if we encounter Z, it means that there are no 1's and so we should see that there are no 2's and only Z's should be present. So, on scanning the first Z, change the state to q_5, replace Z by Z and move the pointer towards right. The transition for this can be of the form

$$\delta(q_4, Z) = (q_5, Z, R)$$

But, in state q_5 only Z's should be there and no more 2's. So, as long as the scanned symbol is Z, remain in state q_5, replace Z by Z and move the pointer towards right. But, once blank symbol B is encountered change the state to q_6, replace B by B and move the pointer towards right and say that the input string is accepted by the machine. The transitions for this can be of the form

$$\delta(q_5, Z) = (q_5, Z, R)$$
$$\delta(q_5, B) = (q_6, B, R)$$

where q_6 is the final state.

So, the TM to recognize the language $L = \{ 0^n 1^n 2^n \mid n \geq 1 \}$ is given by

$$M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$$

where

$Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}$; $\Sigma = \{ 0, 1, 2 \}$

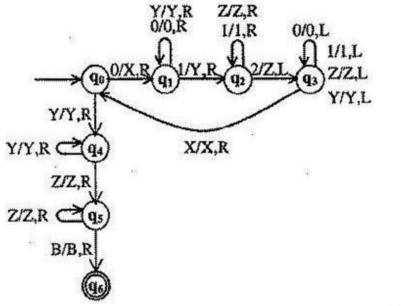
$\Gamma = \{ 0, 1, 2, X, Y, Z, B \}$; q_0 is the initial state

B is blank character ; F = { q_6 } is the final state

δ is shown below using the transition table.

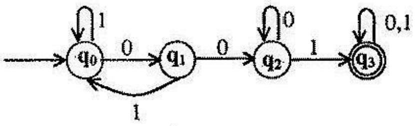
	Γ						
States	0	1	2	Z	Y	X	B
q_0	q_1, X, R				q_4, Y, R		
q_1	q_1, 0, R	q_2, Y,R			q_1, Y,R		
q_2		q_2, 1,R	q_3, Z,L	q_2, Z,R			
q_3	q_3, 0,L	q_3, 1,L		q_3, Z,L	q_3, Y,L	q_0, X,R	
q_4				q_5, Z,R	q_4, Y,R		
q_5				q_5, Z,R			(q_6, B, R)
q_6							

The transition diagram for this can be of the form



Example 3 : Obtain a TM to accept the language $L = \{w \mid w \in (0+1)^*\}$ containing the substring 001.

Solution : The DFA which accepts the language consisting of strings of 0's and 1's having a sub string 001 is shown below :



The transition table for the DFA is shown below :

	0	1
q_0	q_1	q_0
q_1	q_2	q_0
q_2	q_2	q_3
q_3	q_3	q_3

We have seen that any language which is accepted by a DFA is regular. As the DFA processes the input string from left to right in only one direction, TM also processes the input string in only one direction (unlike the previous examples, where the read - write header was moving in both the directions). For each scanned input symbol (either 0 or 1), in whichever state the DFA was in, TM also enters into the same states on same input symbols, replacing 0 by 0 and 1 by 1 and the read - write head moves towards right. So, the transition table for DFA and TM remains same (the format may be different. It is evident in both the transition tables). So, the transition table for TM to recognize the language consisting of 0's and 1's with a substring 001 is shown below :

	0	1	B
q_0	q_1 ,0, R	q_0 , 1, R	-
q_1	q_2 , 0, R	q_0 , 1, R	-
q_2	q_2 , 0, R	q_3 , 1, R	-
q_3	q_3 , 0, R	q_3 , 1, R	q_4 , B, R
q_4			

The TM is given by

$$M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$$

where

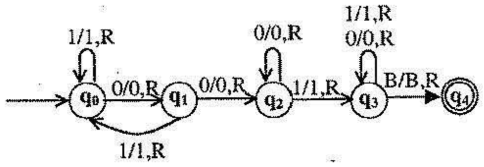
$Q = \{ q_0, q_1, q_2, q_3, q_4 \}$; $\Sigma = \{0, 1\}$

$\Gamma = \{0, 1\}$; δ – is defined already

q_0 is the initial state ; B blank character

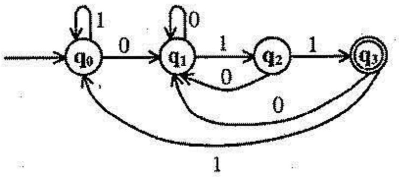
F = { q_4 } is the final state

The transition diagram for this is shown below.



Example 4 : Obtain a Turing machine to accept the language containing strings of 0's and 1's ending with 011.

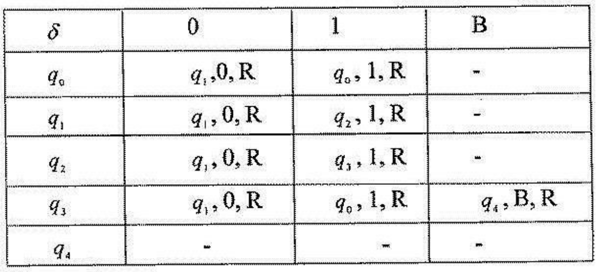
Solution : The DFA which accepts the language consisting of strings of 0's and 1's ending with the string 001 is shown below :



The transition table for the DFA is shown below :

δ	0	1
q_0	q_1	q_0
q_1	q_1	q_2
q_2	q_1	q_3
q_3	q_1	q_0

We have seen that any language which is accepted by a DFA is regular. As the DFA processes the input string from left to right in only one direction, TM also processes the input string in only one direction. For each scanned input symbol (either 0 or 1), in whichever state the DFA was in, TM also enters into the same states on same input symbols, replacing 0 by 0 and 1 by 1 and the read - write head moves towards right. So, the transition table for DFA and TM remains same (the format may be different. It is evident in both the transition tables). So, the transition table for TM to recognize the language consisting of 0's and 1's ending with a substring 001 is shown below :



δ	0	1	B
q_0	q_1,0, R	q_0, 1, R	-
q_1	q_1, 0, R	q_2, 1, R	-
q_2	q_1, 0, R	q_3, 1, R	-
q_3	q_1, 0, R	q_0, 1, R	q_4, B, R
q_4	-	-	-

The TM is given by $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$
where

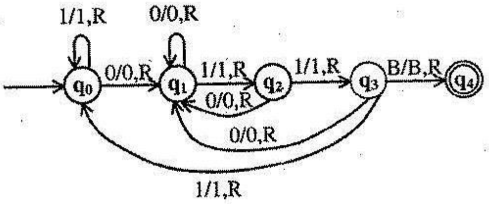
$Q = \{ q_0, q_1, q_2, q_3 \}$; $\Sigma = \{0, 1\}$; $\Gamma = \{0, 1\}$

δ – is defined already

q_0 is the initial state ; B does not appear

F = { q_4 } is the final state

The transition diagram for this is shown below :

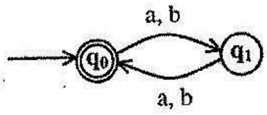


Example 5 : Obtain a Turing machine to accept the language

$L = \{ w | w \text{ is even and } \Sigma = \{ a, b \} \}$

Solution :

The DFA to accept the language consisting of even number of characters is shown below.



The transition table for the DFA is shown below :

	a	b
q_0	q_1	q_1
q_1	q_0	q_0

We have seen that any language which is accepted by a DFA is regular. As the DFA processes the input string from left to right in only one direction, TM also processes the input string in only one direction. For each scanned input symbol (either a or b), in whichever state the DFA was in, TM also enters into the same states on same input symbols, replacing a by a and b by b and the read - write head moves towards right. So, the transition table for DFA and TM remains same (the format may be different). So, the transition table for TM to recognize the language consisting of a's and b's having even number of symbols is shown below :

δ	a	b	B
q_0	q_1,a, R	q_1, b, R	q_2, B, R
q_1	q_0, a, R	q_0, b, R	-
q_2	-	-	-

The TM is given by

$$M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$$

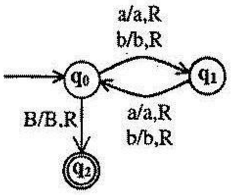
where

Q = { q_0, q_1 }; $\Sigma = \{a, b\}$; $\Gamma = \{a, b\}$

δ – is defined already ; q_0 is the initial state

B does not appear ; F = { q_2 } is the final state

The transition diagram of TM is given by



Example 6 : Obtain a Turing machine to accept a palindrome consisting of a's and b's of any length.

Solution : Let us assume that the first symbol on the tape is blank character B and is followed by the string which in turn ends with blank character B. Now, we have to design a Turing machine which accepts the string, provided the string is a palindrome. For the string to be a palindrome, the first and the last character should be same. The second character and last but one character in the string should be same and so on. The procedure to accept only string of palindromes is shown below. Let q0 be the start state of Turing machine.

Step 1 : Move the read - write head to point to the first character of the string. The transition for this can be of the form $\delta(q_0, B) = (q_1, B, R)$

Step 2 : In state q_1, if the first character is the symbol a, replace it by B and change the state to q_2 and move the pointer towards right. The transition for this can be of the form

$$\delta(q_1, a) = (q_2, B, R)$$

Now , we move the read - write head to point to the last symbol of the string and the last symbol should be a . The symbols scanned during this process are a's , b's and B. Replace a by a, b by b and move the pointer towards right. The transitions defined for this can be of the form

$$\delta(q_2, a) = (q_2, a, R)$$

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

But, once the symbol B is encountered, change the state to q_3, replace B by B and move the pointer towards left. The transition defined for this can be of the form

$$\delta(q_2, B) = (q_3, B, L)$$

In state q_3, the read - write head points to the last character of the string. If the last character is a, then change the state to q_4, replace a by B and move the pointer towards left. The transitions defined for this can be of the form

$$\delta(q_3, a) = (q_4, B, L)$$

At this point, we know that the first character is a and last character is also a. Now, reset the read - write head to point to the first non blank character as shown in step5.

In state q_3, if the last character is B (blank character), it means that the given string is an odd palindrome. So, replace B by B change the state to q_7 and move the pointer towards right. The transition for this can be of the form

$$\delta(q_3, B) = (q_7, B, R)$$

Step 3 : If the first character is the symbol b, replace it by B and change the state from q_1 to q_5 and move the pointer towards right. The transition for this can be of the form

$$\delta(q_1, b) = (q_5, B, R)$$

Now, we move the read - write head to point to the last symbol of the string and the last symbol should be b. The symbols scanned during this process are a's, b's and B. Replace a by a, b by b and move the pointer towards right. The transitions defined for this can of the form

$$\delta(q_5,a)=(q_5,a,R)$$
$$\delta(q_5,b)=(q_5,b,R)$$

But, once the symbol B is encountered, change the state to q_6, replace B by B and move the pointer towards left. The transition defined for this can be of the form

$$\delta(q_5,B)=(q_6,B,L)$$

In state q_6, the read - write head points to the last character of the string. If the last character is b, then change the state to q_6, replace b by B and move the pointer towards left. The transitions defined for this can be of the form

$$\delta(q_6,b)=(q_4,B,L)$$

At this point, we know that the first character is b and last character is also b. Now, reset the read - write head to point to the first non blank character as shown in step 5.

In state q_6, If the last character is B (blank character), it means that the given string is an odd palindrome. So, replace B by B, change the state to q_7 and move the pointer towards right. The transition for this can be of the form

$$\delta(q_6,B)=(q_7,B,R)$$

Step 4 : In state q_1, if the first symbol is blank character (B), the given string is even palindrome and so change the state to q_7, replace B by B and move the read - write head towards right. The transition for this can be of the form

$$\delta(q_1,B)=(q_7,B,R)$$

Step 5 : Reset the read - write head to point to the first non blank character. This can be done as shown below.

If the first symbol of the string is a, step 2 is performed and if the first symbol of the string is b, step 3 is performed. After completion of step 2 or step 3, it is clear that the first symbol and the last symbol match and the machine is currently in state q_4. Now, we have to reset the read - write head to point to the first nonblank character in the string by repeatedly moving the head towards left and remain in state q_4. During this process, the symbols encountered may be a or b or B (blank character). Replace a by a, b by b and move the pointer towards left. The transitions defined for this can be of the form $\delta(q_4,a)=(q_4,a,L)$

$$\delta(q_4,b)=(q_4,b,L)$$

But, if the symbol B is encountered, change the state to q_1, replace B by B and move the pointer towards right. the transition defined for this can be of the form

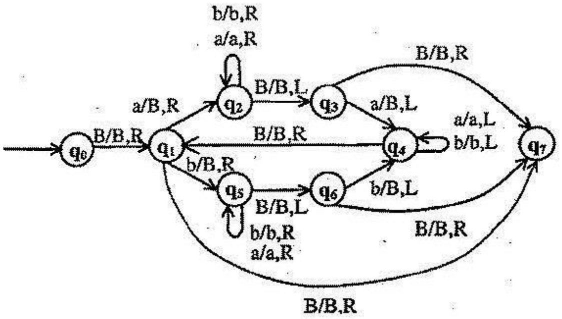
$$\delta(q_4, B) = (q_1, B, R)$$

After resetting the read - write head to the first non - blank character, repeat through step 1.

So, the TM to accept strings of palindromes over { a, b } is given by $M = (Q, \Sigma, \delta, q_0, B, F)$ where $Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7\}$; $\Sigma=\{a, b\}$; $\Gamma=\{a,b,B\}$; q_0 is the initial state B is the blank character ; F = { q_7 } ; δ is shown below using the transition table

	Γ		
δ	a	b	B
q_0	-	-	q_1, B, R
q_1	q_2, B, R	q_5, B, R	q_7, B, R
q_2	q_2, a, R	q_2, b, R	q_3, B, L
q_3	q_4, B, L	-	q_7, B, R
q_4	q_4, a, L	q_4, b, L	q_1, B, R
q_5	q_5, a, R	q_5, b, R	q_6, B, L
q_6	-	q_4, B, L	q_7, B, R
q_7	-	-	-

The transition diagram to accept palindromes over { a, b } is given by



The reader can trace the moves made by the machine for the strings abba, aba and aaba and is left as an exercise.

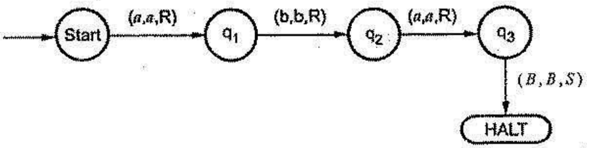
Example 7 : Construct a Turing machine which accepts the language of aba over $\Sigma = \{a, b\}$.

Solution : This TM is only for L = { aba }

We will assume that on the input tape the string 'aba' is placed like this

a	b	a	B	B	....
↑					

The tape head will read out the sequence upto the B character if 'aba' is readout the TM will halt after reading B.



The triplet along the edge written is (input read, output to be printed, direction)

Let us take the transition between start state and q_1 is (a, a, R) that is the current symbol read from the tape is a then as a output a only has to be printed on the tape and then move the tape head to the right. The tape will look like this

a	b	a	B	B	....
	↑				

Again the transition between q_1 and q_2 is (b, b, R). That means read b, print b and move right. Note that as tape head is moving ahead the states are getting changed.

a	b	a	B	B	....
		↑			

The TM will accept the language when it reaches to halt state. Halt state is always a accept state for any TM. Hence the transition between q_3 and halt is (B, B, S). This means read B, print B and stay there or there is no move left or right. Eventhough we write (B, B, L) or (B, B, R) it is equally correct. Because after all the complete input is already recognized and now we simply want to enter into a accept state or final state. Note that for invalid inputs such as abb or ab or bab ..... there is either no path reaching to final state and for such inputs the TM gets stucked in between. This indicates that these all invalid inputs can not be recognized by our TM.

The same TM can be represented by another method of transition table

	a	b	B
Start	(q_1, a, R)	-	-
q_1	-	(q_2, b, R)	-
q_2	(q_3, a, R)	-	-
q_3	-	-	(HALT, B, S)
HALT	-	-	-

In the given transition table, we write the triplet in each row as :

(Next state, output to be printed, direction)

Thus TM can be represented by any of these methods.

Example 8 : Design a TM that recognizes the set $L = \{0^{2n}1^n \mid n \geq 0\}$.

Solution : Here the TM checks for each one whether two 0's are present in the left side. If it match then only it halts and accept the string.

The transition graph of the TM is ,

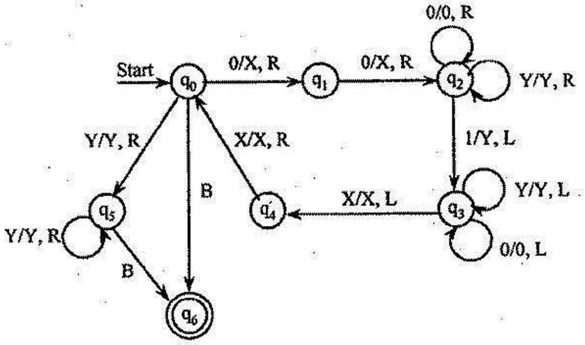


FIGURE : Turing Machine for the given language $L = \{0^{2n}1^n \mid n \geq 0\}$

Example 11 : What does the Turing Machine described by the 5 - tuples, $(q_0,0,q_0,1,R),(q_0,1,q_1,0,r),(q_0,B,q_2,B,R)$, $(q_1,0,q_1,0,\ R),\ (q_1,1,q_0,1,\ R)$ *and* (q_1,B,q_2,B,R) .Do when given a bit string as input ?

Solution : The transition diagram of the TM is ,

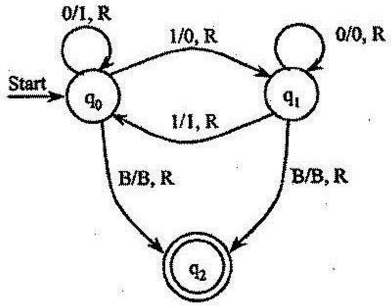


FIGURE : Transition Diagram for the given TM

The TM here reads an input and starts inverting 0's to 1's and 1's to 0's till the first 1. After it has inverted the first 1, it read the input symbol and keeps it as it is till the next 1. After encountering the 1 it starts repeating the cycle by inverting the symbol till next 1. It halts when it encounters a blank symbol.

7.4 COMPUTABLE FUNCTIONS

A Turing machine is a language acceptor which checks whether a string x is accepted by a language L. In addition to that it may be viewed as computer which performs computations of functions from integers to integers. In traditional approach an integer is represented in unary, an integer $i \geq 0$ is represented by the string 0^i.

Example 1 : 2 is represented as 0^2. If a function has k arguments, $i_1, i_2, ... i_k$, then these integers are initially placed on the tape separated by 1's, as $0^{i}10^{i_2}1 .... 10^{i_k}$.

If the TM halts (whether in or not in an accepting state) with a tape consisting of 0's for some m, then we say that $f(i_1, i_2, ... i_k) = m$, where f is the function of k arguments computed by this Turing machine.

$\delta(q_4,1) = (q_4, B, L)$

$\delta(q_4,0) = (q_4, 0, L)$

$\delta(q_4,0) = (q_6, 0, R)$

If in state q_2 a B is encountered before a 0, we have situation (i) described above. Enter state q_4 and move left, changing all 1's to B 's until encountering a 'B'. This B is changed back to a 0, state q_6 is entered, and M halts.

6. $\delta(q_0,1) = (q_5, B, R)$

$\delta(q_5,0) = (q_5, B, R)$

$\delta(q_5,1) = (q_5, B, R)$

$\delta(q_5,B) = (q_6, B, R)$

If in state q_0 a 1 is encountered instead of a 0, the first block of 0's has been exhausted, as in situation (ii) above. M enters state q_5 to erase the rest of the tape, then enters q_6 and halts.

Example 4 : Design a TM which computes the addition of two positive integers.

Solution : Let TM $M = (Q, \{0, 1, \#\}, \delta, s)$ computes the addition of two positive integers m and n. It means, the computed function f (m, n) defined as follows :

$$f(m,n) = \begin{cases} m+n(\mathit{If}\ \ m, n \geq 1) \\ 0 \qquad (m = n = 0) \end{cases}$$

1 on the tape separates both the numbers m and n. Following values are possible for m and n.

1.	$m = n = 0$	(# 1 # ...... is the input),
2.	m = 0 and $n \neq 0$	($\#10^n\#$ .... is the input),
3.	$m \neq 0$ and n = 0	($\#0^m1\#$ ... is the input), and
4.	$m \neq 0$ and $n \neq 0$	($\#0^m10^n\#$ ..... is the input)

Several techniques are possible for designing of M, some are as follows :

(a) M appends (writes) m after n and erases the m from the left end.

(b) M writes 0 in place of 1 and erases one zero from the right or left end . This is possible in case of $n \neq 0$ or $m \neq 0$ only. If m =0 or n = 0 then 1 is replaced by #.

We use techniques (b) given above. M is shown in below figure.

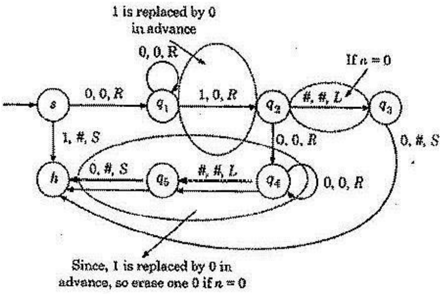


FIGURE : TM for addition of two positive integers

7.5 RECURSIVELY ENUMERABLE LANGUAGES

A language L over the alphabet Σ is called recursively enumerable if there is a TM M that accept every word in L and either rejects (crashes) or loops for every word in language L' the complement of L.

Accept (M) = L

Reject (M) + Loop (M) = L'

When TM M is still running on some input (of recursively enumerable languages) we can never tell whether M will eventually accept if we let it run for long time or M will run forever (in loop).

Example : Consider a language (a + b) * bb (a + b) *.

TM for this language is ,

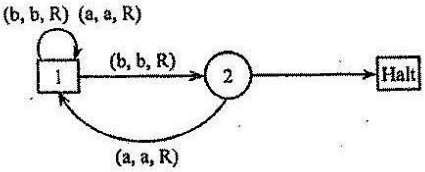


FIGURE : Turing Machine for (a + b) * bb (a + b) *

Here the inputs are of three types.

1. All words with bb = accepts (M) as soon as TM sees two consecutive b's it halts.
2. All strings without bb but ending in b = rejects (M). When TM sees a single b, it enters state2. If the string is ending with b, TM will halt at state 2 which is not accepting state. Hence it is rejected.
3. All strings without bb ending in 'a' or blank 'B' = loop (M) here when the TM sees last a it enters state 1. In this state on blank symbol it loops forever.

Recursive Language

A language L over the alphabet Σ is called recursive if there is a TM M that accepts every word in L and rejects every word in L' i. e.,

accept (M) = L
reject (M) = L'
loop (M) = ϕ.

Example : Consider a language b (a + b) * . It is represented by TM as :

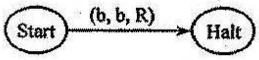


FIGURE : Turing Machine for b (a + b) *

This TM accepts all words beginning with 'b' because it enters halt state and it rejects all words beginning with a because it remains in start state which is not accepting state.

A language accepted by a TM is said to be recursively enumerable languages. The subclass of recursively enumberable sets (r. e) are those languages of this class are said to be recursive sets or recursive language.

7.6 CHURCH'S HYPOTHESIS

According to church's hypothesis, all the functions which can be defined by human beings can be computed by Turing machine. The Turing machine is believed to be ultimate computing machine.

The church's original statement was slightly different because he gave his thesis before machines were actually developed. He said that any machine that can do certain list of operations will be able to perform all algorithms. TM can perform what church asked, so they are possibly the machines which church described.

Church tied both recursive functions and computable functions together. Every partial recursive function is computable on TM. Computer models such as RAM also give rise to partial recursive functions. So they can be simulated on TM which confirms the validity of churches hypothesis.

Important of church's hypothesis is as follows .

1. First we will prove certain problems which cannot be solved using TM.

2. If churches thesis is true this implies that problems cannot be solved by any computer or any programming languages we might every develop.

3. Thus in studying the capabilities and limitations of Turing machines we are indeed studying the fundamental capabilities and limitations of any computational device we might even construct.

It provides a general principle for algorithmic computation and, while not provable, gives strong evidence that no more powerful models can be found.

7.7 COUNTER MACHINE

Counter machine has the same structure as the multistack machine, but in place of each stack is a counter. Counters hold any non negative integer, but we can only distinguish between zero and non zero counters.

Counter machines are off - line Turing machines whose storage tapes are semi - infinite, and whose tape alphabets contain only two symbols, Z and B (blank). Furthermore the symbol Z, which serves as a bottom of stack marker, appears initially on the cell scanned by the tape head and may never appear on any other cell. An integer i can be stored by moving the tape head i cells to the right of Z. A stored number can be incremented or decremented by moving the tape head right or left. We can test whether a number is zero by checking whether Z is scanned by the head, but we cannot directly test whether two numbers are equal.

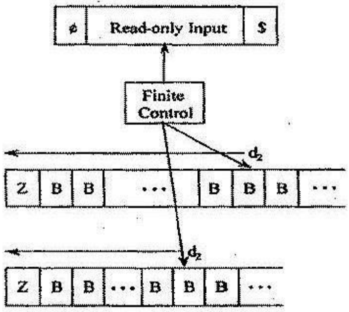


FIGURE : Counter Machine

¢ and $ are customarily used for end markers on the input. Here Z is the non blank symbol on each tape. An instantaneous description of a counter machine can be described by the state, the input tape contents, the position of the input head, and the distance of the storage heads from the symbol Z (shown here as d_1 and d_2). We call these distances the counts on the tapes. The counter machine can only store a count an each tape and tell if that count is zero.

Power of Counter Machines

- Every language accepted by a counter Machine is recursively enumerable.
- Every language accepted by a one - counter machine is a CFL so a one - counter machine is a special case of one - stack machine i. e., a PDA

7.8 TYPES OF TURING MACHINES

Various types of Turing Machines are :

i. With multiple tapes.
ii. With one tape but multiple heads.
iii. With two dimensional tapes.
iv. Non deterministic Turing machines.

It is observed that computationally all these Turing Machines are equally powerful. That means one type can compute the same that other can. However, the efficiency of computation may vary.

1. Turing machine with Two - Way Infinite Tape :

This is a TM that have one finite control and one tape which extends infinitely in both directions.

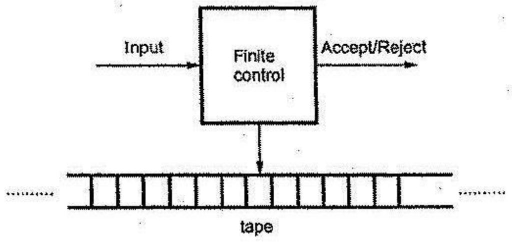


FIGURE : TM with infinite Tape

It turns out that this type of Turing machines are as powerful as one tape Turing machines whose tape has a left end.

2. Multiple Turing Machines :

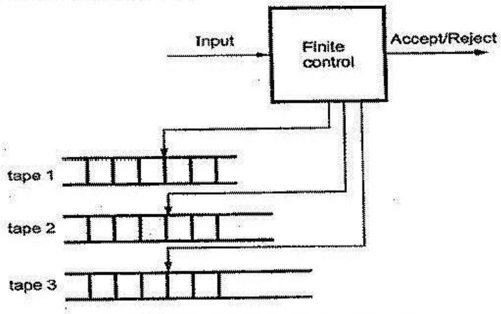


FIGURE : Multiple Turing Machines

A multiple Turing machine consists of a finite control with k tape heads and k tapes, each tape is infinite in both directions. On a single move depending on the state of the finite control and the symbol scanned by each of the tape heads, the machine can

1. Change state.
2. Print a new symbol on each of the cells scanned by its tape heads.
3. Move each of its tape heads, independently, one cell to the left or right or keep it stationary.

Initially, the input appears on the first tape and the other tapes are blank.

3. Nondeterministic Turing Machines :

A nondeterministic Turing machine is a device with a finite control and a single, one way infinite tape. For a given state and tape symbol scanned by the tape head, the machine has a finite number of choices for the next move. Each choice consists of a new state, a tape symbol to print, and a direction of head motion. Note that the non deterministic TM is not permitted to make a move in which the next state is selected from one choice, and the symbol printed and / or direction of head motion are selected from other choices. The non deterministic TM accepts its input if any sequence of choices of moves leads to an accepting state.

As with the finite automaton, the addition of nondeterminism to the Turing machine does not allow the device to accept new languages.

4. Multidimensional Turing Machines :

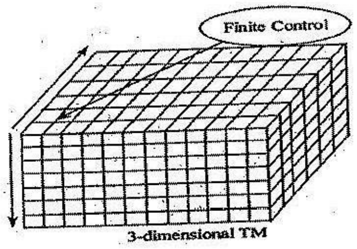


FIGURE : Multidimensional Turing Machine

The multidimensional Turing machine has the usual finite control, but the tape consists of a k - dimensional array of cells infinite in all 2k directions, for some fixed k. Depending on the state and symbol scanned, the device changes state, prints a new symbol, and moves its tape head in one of 2 k directions, either positively or negatively, along one of the k axes. Initially, the input is along one axis, and the head is at the left end of the input.At any time, only a finite number of rows in any dimension contains nonblank symbols, and these rows each have only a finite number of nonblank symbols

5. Multihead Turing Machines :

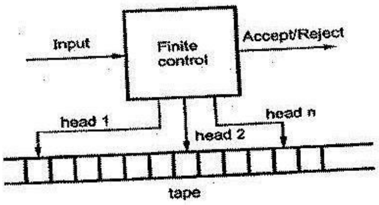


FIGURE : Multihead Turing Machine

A k - head Turing machine has some fixed number, k, of heads. The heads are numbered 1 through k, and a move of the TM depends on the state and on the symbol scanned by each head. In one move, the heads may each move independently left, right or remain stationary.

6. Off - Line Turing Machines :

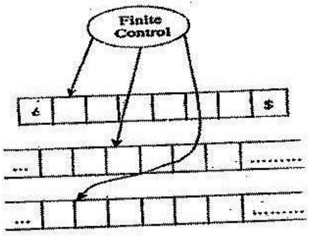


FIGURE : Off - line Turing Machine

COMPUTABILITY THEORY

After going through this chapter, you should be able to understand :

- Chomsky hierarchy of Languages
- Linear Bounded Automata and CSLs
- LR (0) Grammar
- Decidability of problems
- UTM and PCP
- P and NP problems

8.1 CHOMSKY HIERARCHY OF LANGUAGES

Chomsky has classified all grammars in four categories (type 0 to type 3) based on the right hand side forms of the productions.

(a) Type 0

These types of grammars are also known as phrase structured grammars, and RHS of these are free from any restriction. All grammars are type 0 grammars.

Example : productions of types $AS \to aS$, $SB \to Sb, S \to \in$ are type 0 production.

(b) Type 1

We apply some restrictions on type 0 grammars and these restricted grammars are known as type 1 or **context - sensitive grammars** (CSGs). Suppose a type 0 production $\gamma\alpha\delta \to \gamma\beta\delta$ and the production $\alpha \to \beta$ is restricted such that $|\alpha| \leq |\beta|$ and $\beta \neq \in$. Then these type of productions is known as type 1 production. If all productions of a grammar are of type 1 production, then grammar is known as type 1 grammar. The language generated by a context - sensitive grammar is called context - sensitive language (CSL).

In CSG, there is left context or right context or both. For example, consider the production $\alpha A\beta \rightarrow \alpha a\beta$. In this, α is left context and β is right context of A and A is the variable which is replaced.

The production of type $S \rightarrow \in$ is allowed in type 1 if $\in$ is in L(G), but S should not appear on right hand side of any production.

Example : productions $S \rightarrow AB, S \rightarrow \in, A \rightarrow c$ are type 1 productions, but the production of type $A \rightarrow Sc$ is not allowed . Almost every language can be thought as CSL.

Note : If left or right context is missing then we assume that $\in$ is the context.

(c) Type 2

We apply some more restrictions on RHS of type 1 productions and these productions are known as type 2 or context - free productions. A production of the form $\alpha \rightarrow \beta$, where $\alpha, \beta \in (V \cup \Sigma)^*$ is known as type 2 production. A grammar whose productions are type 2 production is known as type 2 or context - free grammar (CFG) and the languages generated by this type of grammars is called context - free languages (CFL).

Example : $S \rightarrow S + S, S \rightarrow S * S, S \rightarrow id$ are type 2 productions.

(d) Type 3

This is the most restricted type. Productions of types $A \rightarrow a$ or $A \rightarrow aB|Ba$, where $A, B \in V$, and $a \in \Sigma$ are known as type 3 or regular grammar productions. A production of type $S \rightarrow \in$ is also allowed, if $\in$ is in generated language.

Example : productions $S \rightarrow aS, S \rightarrow a$ are type 3 productions.

Left - linear production : A production of type $A \rightarrow Ba$ is called left - linear production.

Right - linear production : A production of type $A \rightarrow aB$ is called right - linear production. A left - linear or right - linear grammar is called regular grammar. The language generated by a regular grammar is known as regular language.

8.2 LINEAR BOUNDED AUTOMATA

The Linear Bounded Automata (LBA) is a model which was originally developed as a model for actual computers rather than model for computational process. A linear bounded automaton is a restricted form of a non deterministic Turing machine.

A linear bounded automaton is a multitrack Turing machine which has only one tape and this tape is exactly of same length as that of input.

The linear bounded automaton (LBA) accepts the string in the similar manner as that of Turing machine does. For LBA halting means accepting. In LBA computation is restricted to an area bounded by length of the input. This is very much similar to programming environment where size of variable is bounded by its data type.

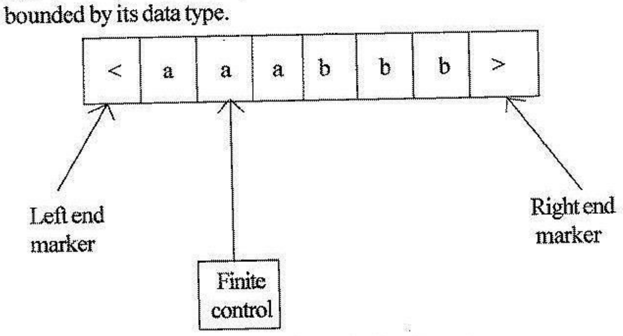


FIGURE : Linear bounded automaton

The LBA is powerful than NPDA but less powerful than Turing machine. The input is placed on the input tape with beginning and end markers. In the above figure the input is bounded by < and >.

A linear bounded automata can be formally defined as :

LBA is 7 - tuple on deterministic Turing machine with

$$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject}) \text{ having}$$

1. Two extra symbols of left end marker and right end marker which are not elements of Γ.
2. The input lies between these end markers.
3. The TM cannot replace < or > with anything else nor move the tape head left of < or right of >.

8.3 CONTEXT SENSITIVE LANGUAGES (CSLs)

The context sensitive languages are the languages which are accepted by linear bounded automata. These type of languages are defined by context sensitive grammar. In this grammar more than one terminal or non terminal symbol may appear on the left hand side of the production rule. Along with it, the context sensitive grammar follows following rules :

i. The number of symbols on the left hand side must not exceed number of symbols on the right hand side.
ii. The rule of the form $A \rightarrow \in$ is not allowed unless A is a start symbol. It does not occur on the right hand side of any rule.

The classic example of context sensitive language is $L = \{a^n\ b^n\ c^n \mid n \geq 1\}$. The context sensitive grammar can be written as :

S	→	aBC
S	→	SABC
CA	→	AC
BA	→	AB
CB	→	BC
aA	→	aa
aB	→	ab
bB	→	bb
bC	→	bc
cC	→	cc

Now to derive the string aabbcc we will start from start symbol :

S	rule S →	SABC
$\underline{S}$ABC	rule S →	aBC
aB$\underline{CA}$BC	rule CA →	AC
aBA$\underline{CB}$C	rule CB →	BC
a$\underline{BA}$BCC	rule BA →	AB
$\underline{aA}$BBCC	rule aA →	aa
a$\underline{aB}$BCC	rule aB →	ab
aa$\underline{bB}$CC	rule bB →	bb
aab$\underline{bC}$C	rule bC →	bc
aabb$\underline{cC}$	rule cC →	cc
aabbcc		

Note : The language $a^n\ b^n\ c^n$ where $n \geq 1$ is represented by context sensitive grammar but it can not be represented by context free grammar.

Every context sensitive language can be represented by LBA.

8.4 LR (k) GRAMMARS

Before going to the topic of LR (k) grammar, let us discuss about some concepts which will be helpful understanding it.

In the unit of context free grammars you have seen that to check whether a particular string is accepted by a particular grammar or not we try to derive that sentence using rightmost derivation or leftmost derivation. If that string is derived we say that it is a valid string.

Example :

$$E \to E + T \mid T$$
$$T \to T * F \mid F$$
$$F \to id \mid (E)$$

Suppose we want to check validity of a string id + id * id . Its rightmost derivation is

$$\begin{aligned}
\text{E} &\Rightarrow E + T \\
&\Rightarrow E + T * F \\
&\Rightarrow E + T * id \\
&\Rightarrow E + F * id \\
&\Rightarrow E + id * id \\
&\Rightarrow T + id * id \\
&\Rightarrow F + id * id \\
&\Rightarrow id + id * id
\end{aligned}$$

FIGURE(a) : Rightmost Derivation of id + id * id

Since this sentence is derivable using the given grammar. It is a valid string. Here we have checked the validity of string using process known as derivation.

In reduction process we have seen that we repeat the process of substitution until we get starting state. But some times several choices may be available for replacement. In this case we have to backtrack and try some other substring . For certain grammars it is possible to carry out the process in deterministic. (i. e., having only one choice at each time). LR grammars form one such subclass of context free grammars. Depending on the number of look ahead symbolized to determine whether a substring must be replaced by a non terminal or not, they are classified as LR(0) , LR(1).... and in general LR(k) grammars.

LR(k) stands for left to right scanning of input string using rightmost derivation in reverse order (we say reverse order because we use reduction which is reverse of derivation) using look ahead of k symbols.

8.4.1 LR(0) Grammar

LR(0) stands for left to right scanning of input string using rightmost derivation in reverse order using 0 look ahead symbols.

Before defining LR(0) grammars, let us know about few terms.

Prefix Property : A language L is said to have prefix property if whenever w in L, no proper prefix of w is in L. By introducing marker symbol we can convert any DCFL to DCFL with prefix property. Hence $L\$ = \{ w\$ \mid w \in L \}$ is a DCFL with prefix property whenever w is in L.

Example : Consider a language L = { cat, cart, bat, art, car } . Here, we can see that sentence cart is in L and its one of the prefixes car is also is in L. Hence, it is not satisfying property. But L$ = { cat $, cart $, bat $, art $, car $ }

Here, cart $ is in L$ but its prefix cart or car are not present in L$. Similarly no proper prefix is present in L$. Hence, it is satisfying prefix property.

Note : LR(0) grammar generates DCFL and every DCFL with prefix property has a LR(0) grammar.

LR Items

An item for a CFG is a production with dot any where in right side including beginning or end. In case of $\in$ production, suppose $A \rightarrow \in$, $A \rightarrow .$ is an item.

Computing Valid Item Sets

The main idea here is to construct from a given grammar a deterministic finite automata to recognize viable prefixes. We group items together into sets which give to states of DFA. The items may be viewed as states of NFA and grouped items may be viewed as states of DFA obtained using subset construction algorithm.

To compute valid set of items we use two operations goto and closure.

Closure Operation

It I is a set of items for a grammar G, then closure (I) is the set of items constructed from I by two rules.

1. Initially, every item I is added to closure (I).
2. If $A \to \alpha.B\beta$ is in closure (I) and $B \to \delta$ is production then add item $B \to \delta$ to I, if it is not already there. We apply this rule until no more new items can be added to closure (I).

Example : For the grammar,

$$S' \to S$$
$$S \to cAd$$
$$A \to a$$

If $S' \to S$ is set of one item in state I then closure of I is,

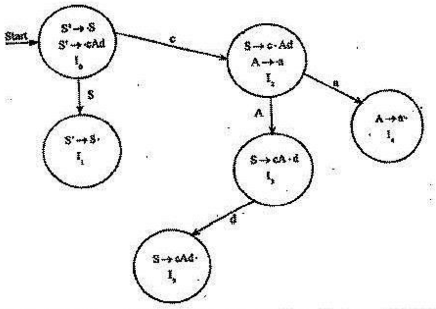
$$I_1 : S' \to .s$$
$$S \to .cAD$$

The first item is added using rule 1 and $S \to .cAd$ is added using rule 2. Because ' . ' is followed by nonterminal S we add items having S in LHS. In $S \to .cAd$ ' . ' is followed by terminal so no new item is added.

Goto Function : It is written as goto (I, X) where I is set of items and X is grammar symbol.

If $A \to \alpha.X\beta$ is in some item set I then goto (I, X) will be closure of set of all item $A \to \alpha.X.\beta$.

DFA :



FIGURE(a) : DFA whose States are the Sets of Valid Items

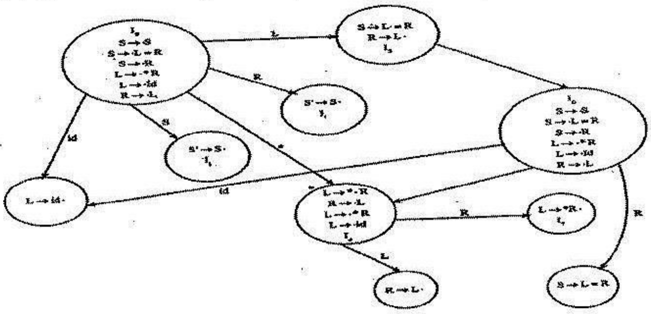
Definition of LR(0) Grammar : We say G is an LR (0) grammar if,

1. Its start symbol does not appear on the right hand side of any production and
2. For every viable prefix γ of G, whenever $A \to \alpha$ is a complete item valid for γ, then no other complete item nor any item with terminal to the right of the dot is valid for γ.

Condition 1 : For a grammar to be LR(0) it should satisfy both the conditions. The first condition can be made to satisfy by all grammars by introduction of a new production $S' \to S$ is known augmented grammar.

Condition 2 : For the DFA shown in Figure(a), the second condition is also satisfied because in the item sets I_1, I_4 and I_5 each containing a complete item, there are no other complete items nor any other conflict.

Example : Consider the DFA given in figure(b).



FIGURE(b) : DFA for the given Grammar

Each problem P is a pair consisting of a set and a question, where the question can be applied to each element in the set. The set is called the domain of the problem, and its elements are called the instances of the problem.

Example :

Domain = { All regular languages over some alphabet Σ } ,
Instance : L = { w : w is a word over Σ ending in abb} ,
Question : Is union of two regular languages regular ?

8.5.1 Decidable and Undecidable Problems

A problem is said to be decidable if

1. Its language is recursive, or
2. It has solution

Other problems which do not satisfy the above are undecidable. We restrict the answer of decidable problems to " YES" or "NO" . If there is some algorithm exists for the problem, then outcome of the algorithm is either "YES" or "NO" but not both. Restricting the answers to only "YES" or "NO" we may not be able to cover the whole problems, still we can cover a lot of problems. One question here. Why we are restricting our answers to only "YES" or "NO"? The answer is very simple ; we want the answers as simple as possible.

Now, we say " If for a problem, there exists an algorithm which tells that the answer is either "YES" or "NO" then problem is decidable."

If for a problem both the answers are possible ; some times "YES" and sometimes "NO", then problem is undecidable.

8.5.2 Decidable Problems for FA, Regular Grammars and Regular Languages

Some decidable problems are mentioned below :

1. Does FA accept regular language ?
2. Is the power of NFA and DFA same ?
3. L_1 and L_2 are two regular languages. Are these closed under following :
   (a) Union
   (b) Concatenation
   (c) Intersection
   (d) Complement

6. We have following co - theorem based on above discussion for recursive enumerable and recursive languages.

Let L and $\overline{L}$ are two languages, where $\overline{L}$ the complement of L, then one of the following is true :

(a) Both L and $\overline{L}$ are recursive languages,

(b) Neither L nor $\overline{L}$ is recursive languages,

(c) If L is recursive enumerable but not recursive, then $\overline{L}$ is not recursive enumerable and vice versa.

Undecidable Problems about Turing Machines

In this section, we will first discuss about halting problem in general and then about TM.

Halting Problem (HP)

The **halting problem** is a decision problem which is informally stated as follows :

"Given a description of an algorithm and a description of its initial arguments, determine whether the algorithm, when executed with these arguments, ever halts. The alternative is that a given algorithm runs forever without halting."

Alan Turing proved in 1936 that there is no general method or algorithm which can solve the halting problem for all possible inputs. An algorithm may contain loops which may be infinite or finite in length depending on the input and behaviour of the algorithm . The amount of work done in an algorithm usually depends on the input size. Algorithms may consist of various number of loops, nested or in sequence. The HP asks the question :

Given a program and an input to the program, determine if the program will eventually stop when it is given that input ?

One thing we can do here to find the solution of HP. Let the program run with the given input and if the program stops and we conclude that problem is solved. But, if the program doesn't stop in a reasonable amount of time, we can not conclude that it won't stop. The question is : " how long we can wait .... ?" . The waiting time may be long enough to exhaust whole life. So, we can not take it as easier as it seems to be. We want specific answer, either "YES" or "NO", and hence some algorithm to decide the answer.

Now, we analyse the following :

1. If H outputs "YES" and says that Q halts then Q itself would loop (that's how we constructed it).
2. If H outputs "NO" and says that Q loops then Q outputs "YES" and will halts.

Since , in either case H gives the wrong answer for Q. Therefore, H cannot work in all cases and hence can't answer right for all the inputs. This contradicts our assumption made earlier for HP. Hence, HP is undecidable.

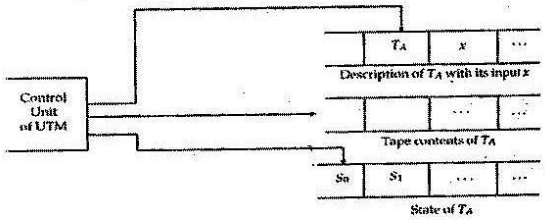
Theorem : HP of TM is undecidable.
Proof : HP of TM means to decide whether or not a TM halts for some input w. We can prove this following the similar steps discussed in above theorem.

8.6 UNIVERSAL TURING MACHINE

The Church - Turing thesis conjectured that anything that can be done on any existing digital computer can also be done by a TM. To prove this conjecture. A. M. Turing was able to construct a single TM which is the theoretical analogue of a general purpose digital computer. This machine is called a Universal Turing Machine (UTM). He showed that the UTM is capable of initiating the operation of any other TM, that is, it is a reprogrammable TM. We can define this machine in more formal way as follows :

Definition : A Universal Turing Machine (denoted as UTM) is a TM that can take as input an arbitrary TM T_A with an arbitrary input for T_A and then perform the execution of T_A on its input.

What Turing thus showed that a single TM can acts like a general purpose computer that stores a program and its data in memory and then executes the program. We can describe UTM as a 3 - tape TM where the description of TM, T_A and its input string $x \in A^*$ are stored initially on the first tape, t_1. The second tape, t_2 used to hold the simulated tape of T_A, using the same format as used for describing the TM, T_A. The third tape, t_3 holds the state of T_A



Now, suppose that a Turing machine, T_A, is consisting of a finite number of configurations, denoted by, $c_0, c_1, c_2, ..., c_p$ and let $\overline{c}_0, \overline{c}_1, \overline{c}_2, ..., \overline{c}_p$ represent the encoding of them. Then, we can define the encoding of T_A as follows :

$$* \ \overline{c}_0 \ \# \ \overline{c}_1 \ \# \ \overline{c}_2 \ \# ...... \# \ \overline{c}_p *$$

Here, * and # are used only as separators, and cannot appear elsewhere. We use a pair of *'s to enclose the encoding of each configuration of TM, T_A.

The case where $\delta(s,a)$ is undefined can be encoded as follows :

$$\# \ \overline{s} \ 0\overline{a} \ 0\overline{B} \ \#$$

where the symbols $\overline{s}$, $\overline{a}$ and $\overline{B}$ stand for the encoding of symbols, s , a and B (Blank character), respectively.

Working of UTM

Given a description of a TM, T_A and its inputs representation on the UTM tape, t_1 and the starting symbol on tape , t_3, the UTM starts executing the quintuples of the encoded TM as follows :

1. The UTM gets the current state from tape, t_3 and the current input symbol from tape t_2.
2. then, it matches the current state - symbol pair to the state symbol pairs in the program listed on tape, t_1.
3. if no match occurs, the UTM halts, otherwise it copies the next state into the current state cell of tape, t_3, and perform the corresponding write and move operations on tape, t_2.
4. if the current state on tape, t_3 is the halt state, then the UTM halts, otherwise the UTM goes back to step 2.

8.7 POST'S CORRESPONDENCE PROBLEM (PCP)

Post's correspondence problem is a combinatorial problem formulated by Emil Post in 1946. This problem has many applications in the field theory of formal languages.

Definition :

A correspondence system P is a finite set of ordered pairs of nonempty strings over some alphabet.

Here, $u_1 = b,\ u_2 = a,\ u_3 = abc,\ \upsilon_1 = ca,\ \upsilon_2 = ab,\ \upsilon_3 = c$.

We have a solution $w = u_3\, u_2 = \ \upsilon_2\, \upsilon_1 = abca$.

8.8 TURING REDUCIBILITY

Reduction is a technique in which if a problem A is reduced to problem B then any solution of B solves A. In general, if we have an algorithm to convert some instance of problem A to some instance of problem B that have the same answer then it is called A reduces to B.

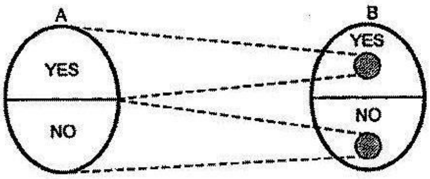


FIGURE: Reduction

Definition : Let A and B be the two sets such that $A, B \subseteq N$ of natural numbers. Then A is Turing reducible to B and denoted as $A \leq_T B$.

If there is an oracle machine that computes the characteristic function of A when it is executed with oracle machine for B.

This is also called as A is B - recursive and B - computable. The oracle machine is an abstract machine used to study decision problem. It is also called as **Turing machine** with **black box.** We say that A is Turing equivalent to B and write $A \equiv_T B$ if $A \leq_T B$ and $B \leq_T A$.

Properties :

1. Every set is Turing equivalent to its complement.
2. Every computable set is Turing equivalent to every other computable set.
3. If $A \leq_T B$ and $B \leq_T C$ then $A \leq_T B$.

8.9 DEFINITION OF P AND NP PROBLEMS

A problem is said to be solvable if it has an algorithm to solve it. Problems can be categorized into two groups depending on time taken for their execution.

1. The problems whose solution times are bounded by polynomials of small degree. **Example:** bubble sort algorithm obtains n numbers in sorted order in polynomial time $P(n) = n^2 - 2n + 1$ where n is the length of input. Hence, it comes under this group.

2. Second group is made up of problems whose best known algorithm are non polynomial example, travelling salesman problem has complexity of $O(n^2\ 2^n)$ which is exponential. Hence, it comes under this group.

A problem can be solved if there is an algorithm to solve the given problem and time required is expressed as a polynomial p(n) , n being length of input string. The problems of first group are of this kind.

The problems of second group require large amount of time to execute and even require moderate size so these problems are difficult to solve. Hence, problems of first kind are tractable or easy and problems of second kind are intractable or hard.

8.9.1 P - Problem

P stands for deterministic polynomial time. A deterministic machine at each time executes an instruction. Depending on instruction, it then goes to next state which is unique.

Hence, time complexity of deterministic TM is the maximum number of moves made by M is processing any input string of length n, taken over all inputs of length n.

Definition : A language L is said to be in class P if there exists a (deterministic) TM M such that M is of time complexity P(n) for some polynomial P and M accepts L.
Class P consists of those problem that are solvable in polynomial time by DTM.

8.9.2 NP - Problem

NP stands for nondeterministic polynomial time.

The class NP consists of those problems that are verifiable in polynomial time. What we mean here is that if we are given certificate of a solution then we can verify that the certificate is correct in polynomial time in size of input problem.

8.10 NP - COMPLETE AND NP - HARD PROBLEMS

A problem S is said to be NP- Complete problem if it satisfies the following two conditions.

1. $S \in NP$, and
2. For every other problems $S_i \in NP$ for some $i = 1, 2, n,$ there is polynomial - time transformation from S_i *to* S i. e. every problem in NP class polynomial - time reducible to S.

We conclude one thing here that if S_i is NP - complete then S is also NP - Complete.

As a consequence, if we could find a polynomial time algorithm for S, then we can solve all NP problems in polynomial time, because all problems in NP class are polynomial - time reducible to each other.

"A problem P is said to be NP - Hard if it satisfies the second condition as NP - Complete, but not necessarily the first condition .".

The notion of NP - hardness plays an important role in the discussion about the relationship between the complexity classes P and NP. It is also often used to define the complexity class NP - Complete which is the intersection of NP and NP - Hard. Consequently, the class NP - Hard can be understood as the class of problems that are NP - complete or harder.

Example : An NP - Hard problem is the decision problem SUBSET - SUM which is as follows.

" Given a set of integers, do any non empty subset of them add up to zero? This is a yes / no question, and happens to be NP - complete ".

There are also decision problems that are NP - Hard but not NP - Complete , for example, the halting problem of Turing machine. It is easy to prove that the halting problem is NP - Hard but not NP - Complete. It is also easy to see that halting problem is not in NP since all problems in NP are decidable but the halting problem is not (voilating the condition first given for NP - complete languages).

In Complexity theory, the **NP - complete** problems are the hardest problems in NP class, in the sense that they are the ones most likely not to be in P class. The reason is that if we could find a way to solve any NP - complete problem quickly, then you could use that algorithm to solve all NP problems quickly.

At present time, all known algorithms for NP - complete problems require time which is exponential in the input size. It is unknown whether there are any faster algorithms for these are not.

UNIT-IV

INTRODUCTION TO LANGUAGE ROCESSING:

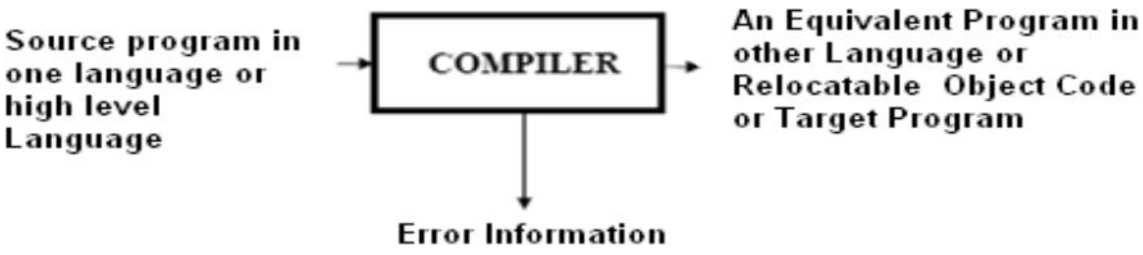
AsComputersbecameinevitableandindigenouspartofhumanlife,andseverallanguages withdifferentandmoreadvancedfeaturesareevolvedintothisstreamtosatisfyorcomforttheuser in communicating with the machine , the development of the translators or mediator Software‘s have become essential to fill the huge gap between the human and machine understanding. This process is called Language Processing to reflect the goal and intent of the process. On the wayto this process to understand it in a better way, we have to be familiar with some key terms and concepts explained in following lines.

LANGUAGETRANSLATORS:

Is a computer program which translates a program written in one (Source) language to its equivalentprograminother[Target]language.TheSourceprogramisahighlevellanguagewhereas the Target language can be any thing from the machine language of a target machine (between Microprocessor to Supercomputer) to another high level languageprogram.

∑TwocommonlyUsed Translators areCompilerandInterpreter

1. **Compiler :** Compilerisaprogram,readsprograminonelanguagecalledSourceLanguage andtranslatesintoitsequivalentprograminanotherLanguagecalledTargetLanguage,in addition to this its presents the error information to the User.


Source program in one language or high level Language
COMPILER
An Equivalent Program in other Language or Relocatable Object Code or Target Program
Error Information

∑ Ifthetargetprogramisanexecutablemachine-languageprogram,itcanthenbecalledby the users to process inputs and produce outputs.

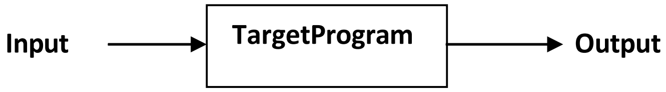


Figure1.1:RunningthetargetProgram

2. **Interpreter:**Aninterpreterisanothercommonlyusedlanguageprocessor.Insteadofproducing a target program as a single translation unit, an interpreter appears to directly execute the operations specified in the source program on inputs supplied by theuser.

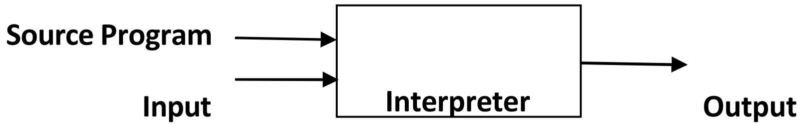


Figure1.2:RunningthetargetProgram

LANGUAGE PROCESSING SYSTEM:

Basedontheinputthetranslatortakesandtheoutputitproduces,alanguagetranslatorcanbe called as any one of the following.

Preprocessor: Apreprocessortakestheskeletalsourceprogramasinputandproducesanextended version of it, which is the resultant of expanding the Macros, manifest constants if any, and includingheader files etc in thesourcefile. For example, the C preprocessor is amacro processor thatisusedautomaticallybytheCcompilertotransformoursourcebeforeactualcompilation.Over and above a preprocessor performs the following activities:

∑ Collectsallthemodules,filesincaseifthesourceprogramisdividedintodifferentmodules stored at different files.

∑ Expandsshorthands/ macrosintosourcelanguagestatements.

Compiler: Is a translator that takes as input a source program written in high level language and convertsitintoitsequivalenttarget programinmachinelanguage.Inadditiontoabovethecompiler also

∑Reportstoitsuserthepresenceoferrorsinthesourceprogram.

∑Facilitatestheuserinrectifyingtheerrors,andexecutethecode.

Assembler: Isaprogramthattakesasinputanassemblylanguageprogramandconvertsitintoits equivalent machine language code.

Loader/Linker: Thisisaprogramthattakesasinputarelocatablecodeandcollectsthelibrary functions, relocatable object files, and produces its equivalent absolute machine code.

Specifically,

∑**Loading**consistsoftakingtherelocatablemachinecode,alteringtherelocatableaddresses, and placing the altered instructions and data in memoryat the proper locations.

∑**Linking** allows us to make a single program from several files of relocatable machine code. These files may have been result of several different compilations, one or moremaybe libraryroutines provided bythe system available to any program that needs them.

In addition to these translators, programs like interpreters, text formatters etc., may be used in language processing system. To translate a program in a high level language program to an executable one, the Compiler performs by default the compile and linking functions.

Normally the steps in a language processing system includes Preprocessing the skeletal Source program which produces an extended or expanded source program or a ready to compile unit of the source program, followed by compiling the resultant, then linking / loading , and finally its equivalent executable code is produced. As I said earlier not all these steps are mandatory. In some cases, the Compiler only performs this linking and loading functions implicitly.

The steps involved in a typical language processing system can be understood with following diagram.

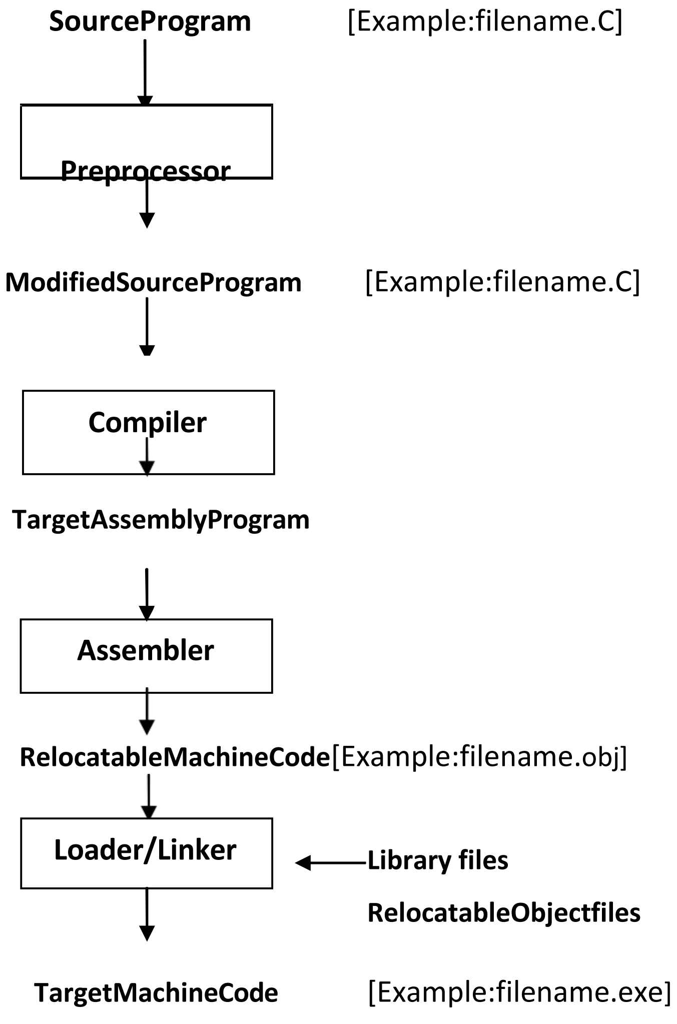


Figure1.3:ContextofaCompilerinLanguageProcessingSystem

PHASES OF A COMPILER:

Due to the complexity of compilation task, a Compiler typically proceeds in a Sequence of compilation phases. The phases communicate with each other via clearly defined interfaces. Generally an interface contains a Data structure (e.g., tree), Set of exported functions.Each phase works on an abstract **intermediate representation** of the source program, not the source program text itself (except the first phase)

Compiler Phases are the individual modules which are chronologicallyexecuted to perform their respective Sub-activities, and finally integrate the solutions to give target code.

It is desirable to have relatively few phases, since it takes time to read and write immediate files. Followingdiagram (Figure1.4) depicts the phases of a compiler through which it goes duringthe compilation. There fore a typical Compiler is having the following Phases:

1.LexicalAnalyzer(Scanner),2.SyntaxAnalyzer(Parser),3.SemanticAnalyzer, 4.Intermediate Code Generator(ICG), 5.Code Optimizer(CO) , and 6.Code Generator(CG)

In addition to these, it also has **Symbol table management**, and **Error handler** phases. Not all the phases are mandatory in every Compiler. e.g, Code Optimizer phase is optional in some cases.

Thedescription isgiven innext section. The Phases of compiler divided in to two parts, first three phases we are called as Analysis part remaining three called as Synthesis part.

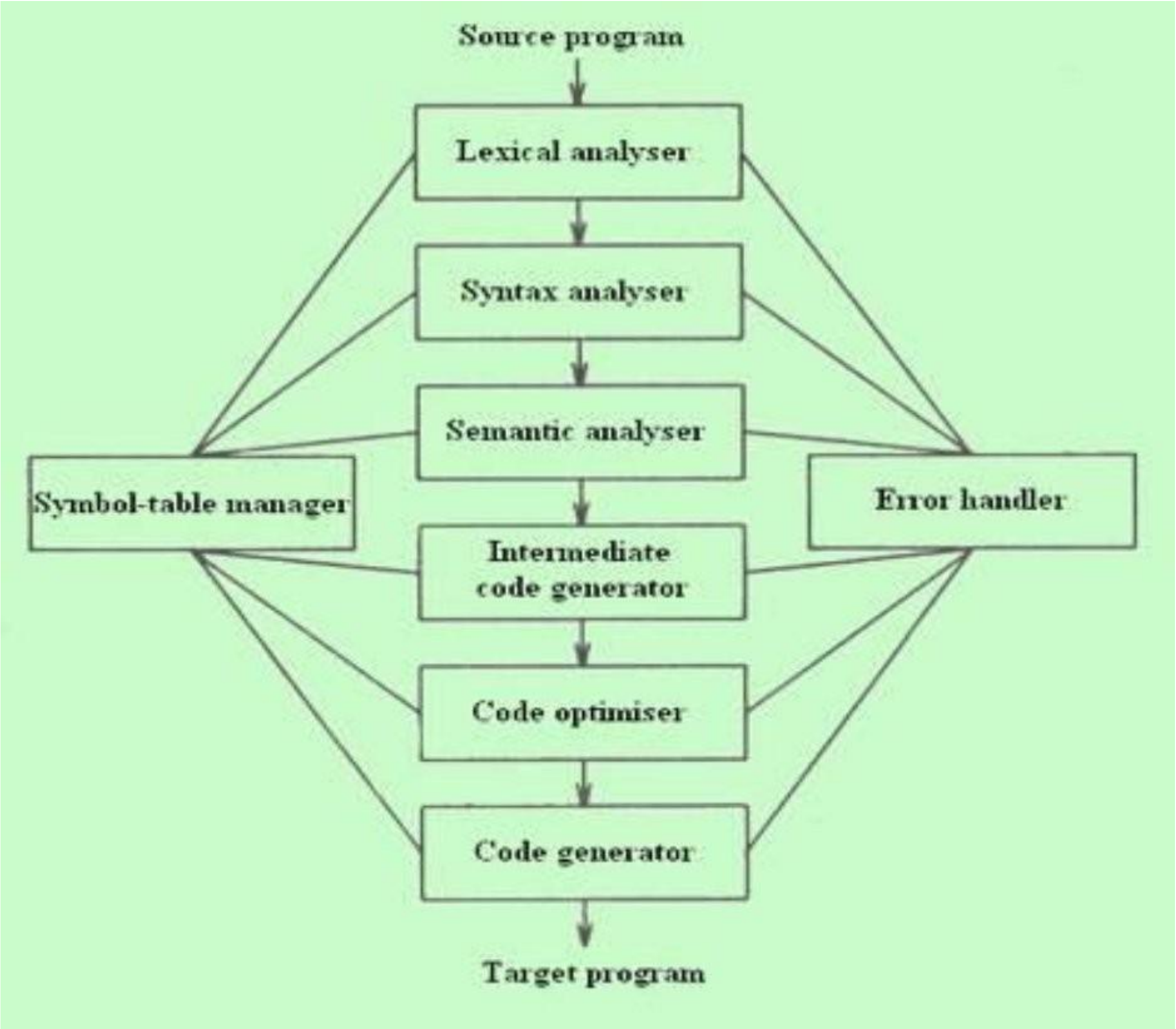


Figure1.4:PhasesofaCompiler

PHASE,PASSESOFACOMPILER:

In some application we can have a compiler that is organized into what is called passes. Where a pass is a collection of phases that convert the input from one representation to a completely deferent representation. Each pass makes a complete scan of the input and produces its output to be processed by the subsequent pass. For example a two pass Assembler.

THEFRONT-END& BACK-ENDOFACOMPILER

All of these phases of a general Compiler are conceptually divided into **The Front-end**, and **The Back-end**. This division is due to their dependence on either the Source Language orthe Target machine. This model is called an Analysis & Synthesis model of a compiler.

The **Front-end** of the compiler consists of phases that depend primarily on the Source language and are largely independent on the target machine. For example, front-end of the compiler includes Scanner, Parser, Creation of Symbol table, Semantic Analyzer, and the Intermediate Code Generator.

The **Back-end** of the compiler consists of phases that depend on the target machine, and thoseportionsdon‘tdependentontheSourcelanguage,justthe Intermediatelanguage.Inthiswe have different aspects of Code Optimization phase, code generation along with the necessary Error handling, and Symbol table operations.

LEXICAL ANALYZER(SCANNER): TheScanneristhefirstphasethatworksasinterface betweenthe compilerandtheSourcelanguageprogramandperformsthefollowingfunctions:

∑Reads the characters in the Source program and groups them into a stream of tokens in which each token specifies a logically cohesive sequence of characters, such as an identifier , a Keyword , a punctuation mark, a multi character operator like := .

∑ Thecharactersequenceformingatoken iscalleda**lexeme**ofthetoken.

∑ TheScannergeneratesatoken-id,andalsoentersthatidentifiersnameintheSymbol table if it doesn‘t exist.

∑ Alsoremoves theComments, and unnecessaryspaces.

Theformat ofthetoken is **< Tokenname, Attributevalue>**

SYNTAX ANALYZER(PARSER): TheParserinteractswiththeScanner,anditssubsequent phase Semantic Analyzer and performs the following functions:

∑ Groupstheabovereceived,andrecordedtokenstreamintosyntacticstructures,usually into a structure called **Parse Tree** whose leaves are tokens.

∑ Theinteriornodeofthistreerepresentsthestreamoftokensthatlogicallybelongs together.

- Itmeansitchecksthesyntax ofprogramelements.

SEMANTICANALYZER: This phase receives the syntax tree as input, and checksthe semanticallycorrectnessoftheprogram.Thoughthetokensarevalidandsyntacticallycorrect,it may happen that they are not correct semantically. Therefore the semantic analyzer checks thesemantics (meaning) of the statements formed.

- TheSyntacticallyandSemanticallycorrectstructuresareproducedhereintheformofa Syntax tree or DAG or some other sequential representation like matrix.

INTERMEDIATE CODE GENERATOR(ICG): This phase takes the syntactically and semantically correct structure as input, and produces its equivalent intermediate notation of the source program. The Intermediate Code should have two important properties specified below:

- Itshouldbeeasytoproduce,andEasytotranslateintothetargetprogram.Example intermediate code forms are:

- Threeaddress codes,

- Polishnotations,etc.

CODE OPTIMIZER: This phase is optional in some Compilers, but so useful and beneficial in terms of saving development time, effort, and cost. This phase performs the following specific functions:

- Attempts to improve the IC so as to have a faster machine code. Typical functionsinclude – Loop Optimization, Removal of redundant computations, Strength reduction, Frequency reductions etc.

- Sometimesthedatastructuresusedinrepresentingtheintermediateformsmayalsobe changed.

CODE GENERATOR: This is the final phase of the compiler and generates the target code, normally consisting of the relocatable machine code or Assembly code or absolute machinecode.

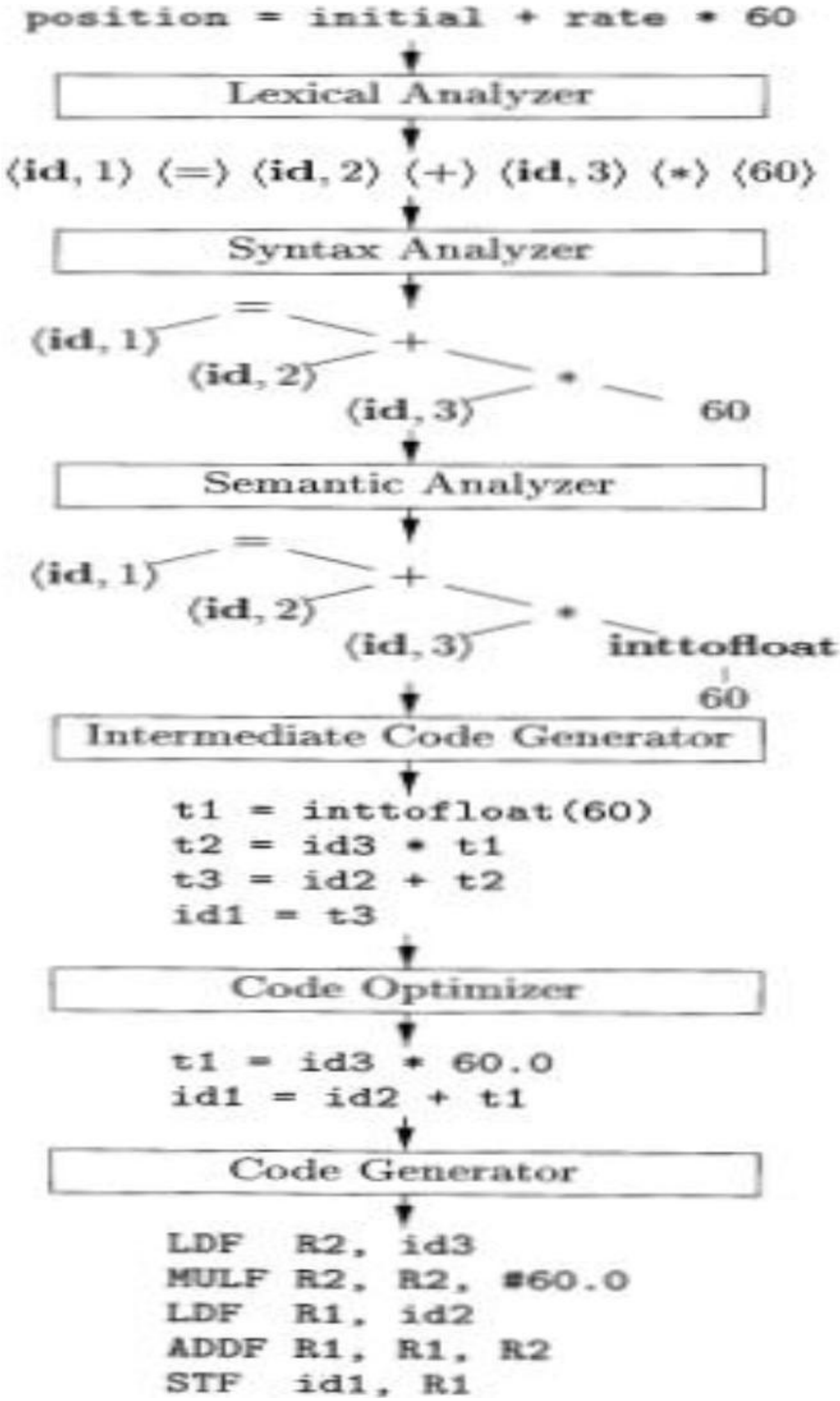
- Memorylocationsareselectedforeachvariableused,andassignmentofvariablesto registers is done.

- Intermediateinstructionsaretranslatedintoasequenceofmachine instructions.

TheCompileralsoperformsthe**Symboltablemanagement**and**Errorhandling**throughoutthe compilation process. Symbol table is nothing but a data structure that stores different source language constructs, and tokens generated during the compilation. These two interact with all phases of the Compiler.

Forexamplethesourceprogramisanassignmentstatement;thefollowingfigureshowshowthe phases of compiler will process the program.

Theinput sourceprogram is**Position=initial+rate*60**



```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

```
t1 = id3 * 60.0
id1 = id2 + t1
```

```
LDF  R2, id3
MULF R2, R2, #60.0
LDF  R1, id2
ADDF R1, R1, R2
STF  id1, R1
```

Figure1.5:TranslationofanassignmentStatement

LEXICALA NALYSIS:

As the first phase of a compiler, the main task ofthelexical analyzeristo read the input characters of the source program, group them into lexemes, and produce as output tokens for each lexeme in the source program. This stream of tokens is sent to the parser for syntax analysis. It is common for the lexical analyzer to interact with the symbol table as well.

When the lexical analyzer discovers a lexeme constituting an identifier,it needstoenter that lexeme into the symbol table. This process is shown in the followingfigure.

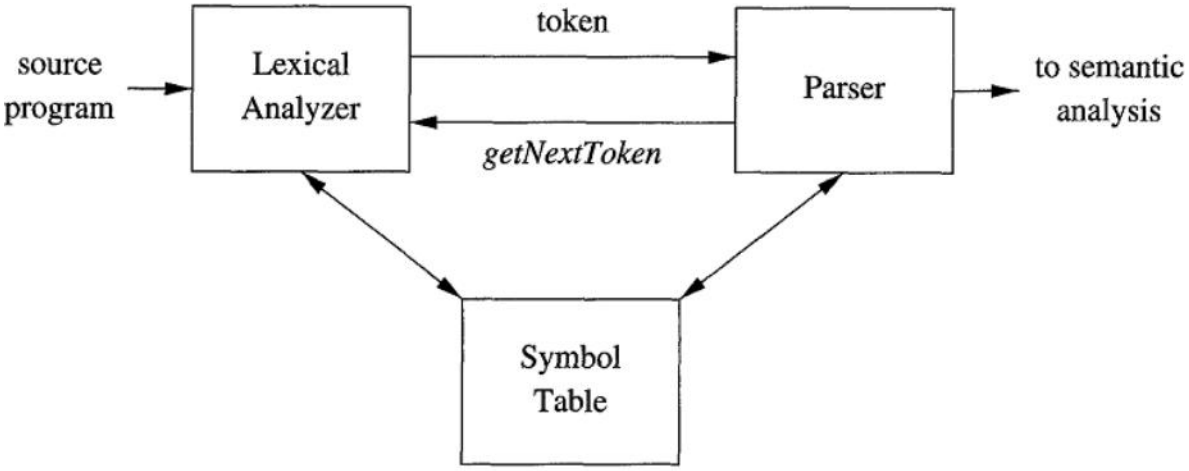


Figure1.6:LexicalAnalyzer

. When lexical analyzer identifies the first token it will send it to the parser, the parser receives the token and calls the lexical analyzer to send next token by issuing the **getNextToken()** command. This Process continues until the lexical analyzer identifies all the tokens. During this process the lexical analyzer will neglect or discard the white spaces and comment lines.

TOKENS,PATTERN SAND LEXEMES:

A token is a pair consisting of a tokennameandan optional attribute value.The token name is an abstract symbol representing a kind of lexical unit, e.g., a particular keyword, or a sequence of input characters denoting an identifier. The token names are the input symbols that the parser processes. In what follows, we shall generally write the name of a token in boldface. We will often refer to a token by its token name.

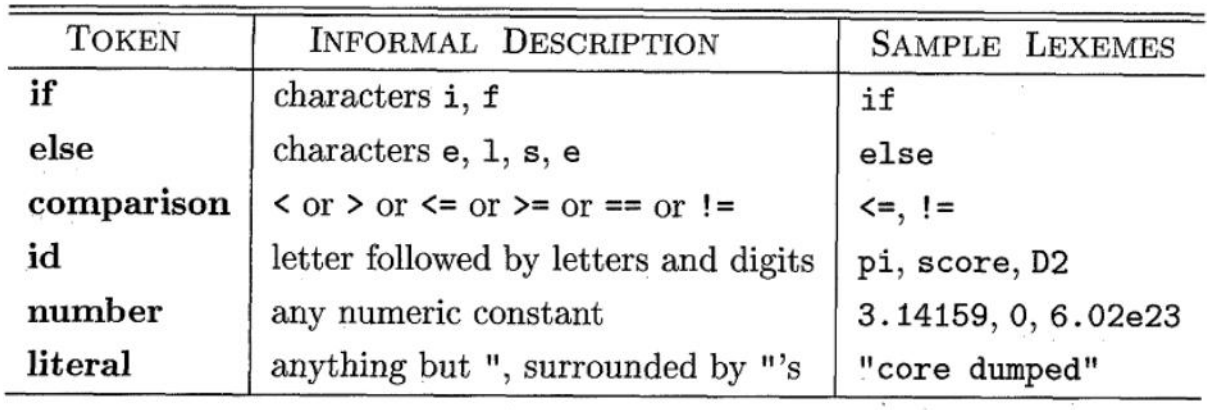
A pattern is a description of the form that the lexemes of a token may take [or match]. In the case of a keyword as a token, the pattern is just the sequence of characters that form the keyword. For identifiers and some other tokens, the pattern is a more complex structure that is matched by many strings.

Alexemeisasequenceofcharactersinthesourceprogramthatmatchesthepatternfora token and is identified by the lexical analyzer as an instance of that token.

Example:InthefollowingClanguagestatement, printf

("Total = %d\n‖, score) ;

both **printf** and **score**are lexemes matchingthe**pattern**fortoken**id**, and **"Total=%d\n‖** is a lexeme matching **literal [or string]**.



TOKEN	INFORMAL DESCRIPTION	SAMPLE LEXEMES
if	characters i, f	if
else	characters e, l, s, e	else
comparison	< or > or <= or >= or == or !=	<=, !=
id	letter followed by letters and digits	pi, score, D2
number	any numeric constant	3.14159, 0, 6.02e23
literal	anything but ", surrounded by "'s	"core dumped"

Figure1.7:ExamplesofTokens

LEXICAL ANALYSIS Vs PARSING:

Thereareanumberofreasonswhytheanalysisportionofacompilerisnormallyseparatedinto lexical analysis and parsing (syntax analysis) phases.

∑1. **Simplicity of design is the most important consideration.** The separation of Lexical and Syntactic analysis often allows us to simplify at least one of these tasks. Forexample, a parser that had to deal with comments and whitespace as syntactic unitswould be considerably more complex than one that can assume comments andwhitespace have already been removed by the lexicalanalyzer.

∑**2. Compiler efficiency is improved**. A separate lexical analyzer allows us to apply specialized techniques that serve only the lexical task, not the job of parsing. In addition, specialized buffering techniques for reading input characters can speed up the compiler significantly.∑

3.Compilerportabilityisenhanced:Input-device-specificpeculiaritiescanbe restricted to the lexical analyzer.

INPUTBUFFERING:

Before discussing the problem of recognizinglexemesin the input,let us examine some ways that the simple but important task of reading the source program canbe speeded. This task is made difficult by the fact that we often have to look one or more characters beyond the next lexeme before we can be sure we have the right lexeme. There are many situationswhere we need to look at least one additional character ahead. For instance, we cannot be sure we've seen the end of an identifier until we see a character that is not a letter or digit, and therefore is not part of the lexeme forid.InC, single-characteroperators like-,=,or< could also be the beginning of a two-character operator like ->, ==, or <=. Thus, we shall introduce a two-buffer scheme that handles large look aheads safely. We then consider an improvement involving "sentinels" that saves time checking for the ends of buffers.

Buffer Pairs

Because of the amount of time taken to process characters and the large number of charactersthat must be processed during the compilation of a large source program, specialized buffering techniques have been developed to reduce the amount of overhead required to process a single input character. An important scheme involves two buffers that are alternately reloaded.

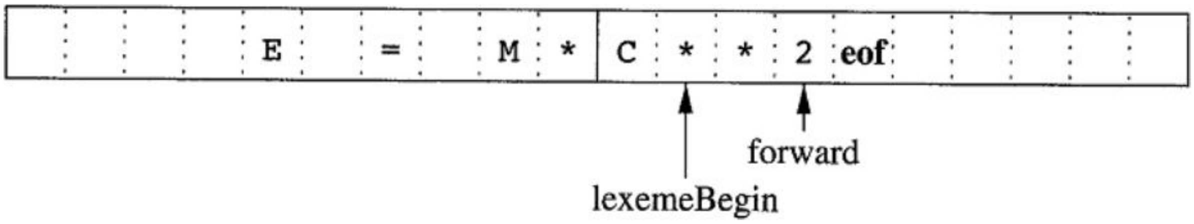


Figure1.8 :UsingaPairof InputBuffers

Each buffer is of the same size N, and N is usually the size of a disk block, e.g., 4096 bytes. Using one system readcommand we canread Ncharacters in to a buffer, rather than using one system call per character. If fewer than N characters remain in the input file, then a special character, represented by eof, marks the end of the source file and is different from any possible character of the source program.

- Twopointers tothe inputaremaintained:

1. ThePointer**lexemeBegin**,marksthebeginningofthecurrentlexeme,whoseextent we are attempting to determine.
2. Pointer **forward** scans ahead until a pattern match is found; the exact strategy wherebythisdeterminationismadewillbecoveredinthebalanceofthischapter.

Once the next lexeme is determined, forward is set to the character at its right end. Then, after the lexeme is recorded as an attribute value of a token returned to the parser, 1exemeBegin is set to the character immediatelyafter the lexeme just found. In Fig, we see forward has passed the end of the next lexeme, ** (the FORTRAN exponentiation operator), and must be retracted one position to its left.

Advancing forward requires that we first test whether we have reachedthe end ofoneof the buffers, and if so, we must reload the other bufferfromthe input, and move forward to the beginning of the newly loaded buffer. As long as we never need to look so far ahead of the actual lexeme that the sumof the lexeme'slengthplusthe distance welookahead isgreater than N, we shall never overwrite the lexeme in its buffer before determining it.

SentinelsToImproveScanners Performance:

If we use the above scheme as described, we must check, each time we advance forward, that we have not moved off one of the buffers; if we do, then we must also reload the other buffer. Thus, for each character read, we make two tests: one for the end of the buffer, and oneto determine what character is read (the latter may be a multi way branch). We can combine the buffer-end test with the test for the current character if we extend each buffer to hold a **sentinel** character at the end. The sentinel is a special character that cannot be part of the source program, and a natural choice is the character **eof**. Figure 1.8 shows the same arrangement as Figure 1.7, but with the sentinels added. Note that eof retains its use as a marker for the end of the entire input.

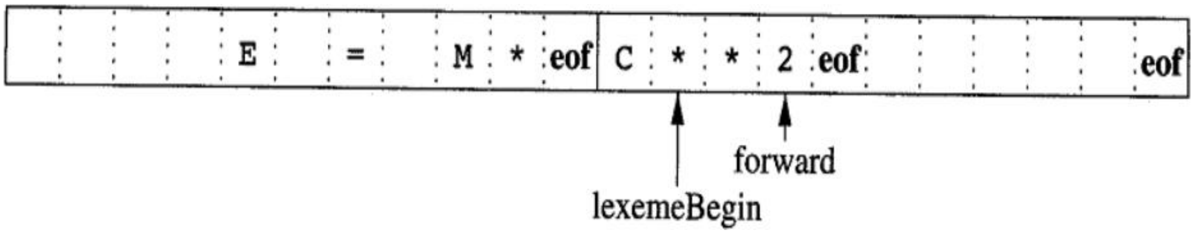


Figure1.8:Sententialattheendof eachbuffer

Anyeofthatappearsotherthanattheendofabuffermeansthattheinput isatanend.Figure1.9 summarizes the algorithm for advancing forward. Notice how the first test, which can be part of amultiwaybranchbased onthecharacterpointedtobyforward,istheonlytestwemake,except in the case where we actually are at the end of a buffer or the end of the input.

```
switch(*forward++)
{
        caseeof:if(forwardisatendof firstbuffer)
                    {
                      reloadsecondbuffer;
                      forward=beginningofsecond buffer;
                     }
                    elseif (forwardisatend of secondbuffer)
                    {
                      reloadfirst buffer;
                      forward=beginningof first buffer;
                    }
                      else    /*eofwithinabuffermarkstheendofinput*/
                        terminate lexical analysis;
           break;
}
```

Figure1.9:useof switch-caseforthesentential

SPECIFICATIONOF TOKENS:

Regular expressions are an important notation for specifying lexeme patterns. While they cannot express allpossiblepatterns,theyareveryeffectiveinspecifyingthosetypesofpatternsthat weactuallyneedfor tokens.

LEXtheLexicalAnalyzer generator

Lex is a tool used to generate lexical analyzer, the input notation for the Lex tool is referred to as the Lex language and the tool itself is the Lex compiler. Behind the scenes, theLex compiler transforms theinputpatterns into a transition diagram and generates code, in a filecalledlex.yy.c,itisacprogramgivenforCCompiler,givestheObjectcode.Hereweneed to know how to write the Lex language. The structure of the Lex program is givenbelow.

Structureof LEXProgram: A Lex programhasthefollowingform:

Declarations

%%

Translation rules

%%

Auxiliaryfunctionsdefinitions

Thedeclarationssection:includesdeclarationsofvariables,manifestconstants(identifiers declaredtostandforaconstant,e.g.,thenameofatoken),andregulardefinitions.Itappears between %{. . .%}

Inthe**Translationrules**section,WeplacePatternActionpairswhereeachpairhavetheform Pattern

{Action}

Theauxiliaryfunctiondefinitionssectionincludesthedefinitionsoffunctionsusedtoinstall identifiers and numbers in the Symbol tale.

LEXProgramExample:

```
%{

/*definitionsofmanifestconstantsLT,LE,EQ,NE,GT,GE,IF,THEN,ELSE,ID,NUMBER,
RELOP */

%}

/*regulardefinitions */

delim           [\t\n]
```

```
        Ws      {       delim}+
        letter          [A-Za-z]
        digit           [o-91
        Id              {letter}({letter} |{digit})*
        number          {digit}+(\. {digit}+)?(E[+-I]?{digit}+)?
        %%
        {ws}             {/*no actionand noreturn */}
        If               {return(1F);}
then    {return(THEN);}
        else            {return(ELSE);}
        (id)            {yylval=(int)installID(); return(1D);}
        (number)        {yylval=(int)installNum();return(NUMBER);}
        ‖<‖             {yylval=LT; return(REL0P);)}
        –<=‖            {yylval=LE; return(REL0P);}
        —=‖             {yylval=EQ;return(REL0P);}
        —<>‖            {yylval= NE;return(REL0P);}
        —<‖             {yylval=GT;return(REL0P);)}
        —<=‖            {yylval=GE;return(REL0P);}
        %%
        intinstallID0(){/*functionto installthelexeme,whosefirstcharacterispointedtobyyytext, and
                                whose length is yyleng, into the symbol table and return apointer thereto */
        intinstallNum(){/*similartoinstallID,butputsnumericalconstantsinto aseparatetable*/}
```

Figure1.10:LexProgramfortokenscommontokens

SYNTAX ANALYSIS(PARSER)

THEROLEOFTHE PARSER:

In our compiler model, the parser obtains astring oftokens fromthelexical analyzer, as shown inthe below Figure,and verifiesthatthestringoftoken names canbe generated by thegrammarforthesource language.We expect the parser to report any syntaxerrors in an intelligible fashion and to recover from commonlyoccurring errors to continue processing the remainder of the program. Conceptually, for well-formed programs, the parser constructs a parse tree and passes it to the rest of the compiler for further processing.

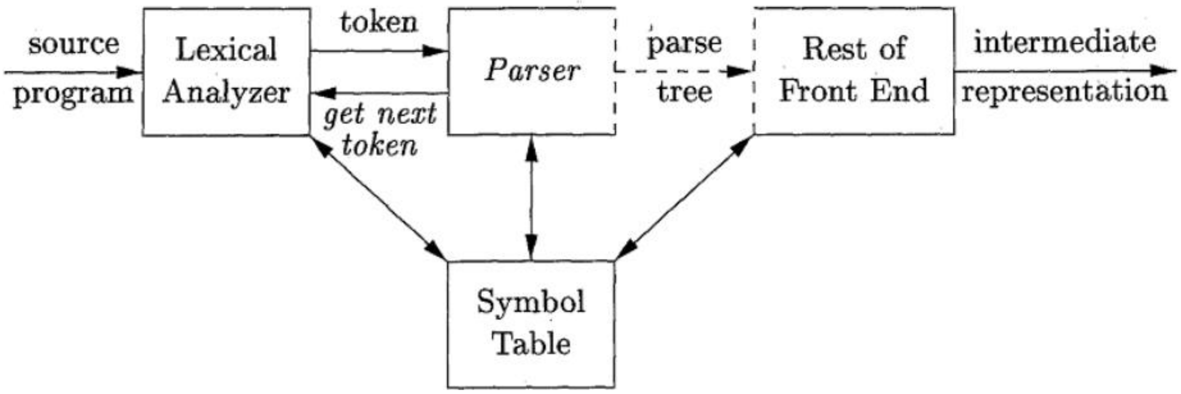


Figure2.1:Parserin theCompiler

Duringtheprocessofparsingitmayencountersomeerrorandpresenttheerrorinformationback to the user

Syntacticerrorsincludemisplacedsemicolonsorextraormissingbraces;thatis, —{" or "}."Asanother example, in CorJava,the appearance ofacasestatementwithout anenclosing switchisasyntactic error(however,thissituationisusually allowedbythe parser and caught later in the processing, as the compiler attempts to generate code).

Basedontheway/ordertheParseTreeisconstructed,**Parsing**isbasically**classified**into following two types:

1. **TopDownParsing:**Parsetreeconstructionstartattherootnodeandmovestothe children nodes (i.e., top down order).
2. **BottomupParsing:**Parsetreeconstructionbeginsfromtheleafnodesandproceeds towards the root node (called the bottom up order).

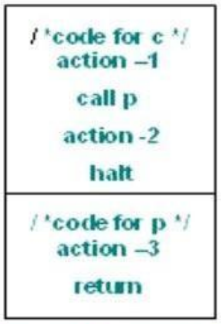
UNIT-V

RUNTIMESTORAGE MANAGEMENT:

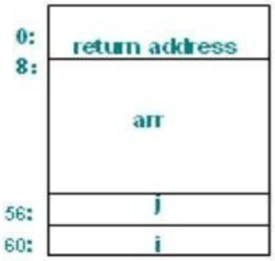
To study the run-time storage management system it is sufficient to focus on the statements: action, call, return and halt, because they by themselves give us sufficient insight into the behavior shown by functions in calling each other and returning.

And the run-time allocation and de-allocation of activations occur on the call of functions and when they return.

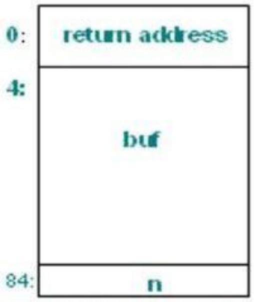
There are mainly two kinds of run-time allocation systems: **Static allocation** and **Stack Allocation**. Whilestaticallocation is used bythe FORTRAN class of languages, stack allocation is used by the Ada class of languages.



Three address code



Activation record for c (64 bytes)



Activation record for p (88 bytes)

STATICALLOCATION: Inthis,Acallstatementisimplementedbyasequenceoftwo instructions.

- ∑ Amoveinstructionsavesthereturn address
- ∑ Agototransferscontroltothetargetcode.

The instruction sequence is

```
MOV#here+20,callee.static-area

GOTO callee.code-area
```

callee.static-areaandcallee.code-areaareconstantsreferringtoaddressoftheactivationrecord and the first address of called procedure respectively.

.#here+20inthemoveinstructionisthereturnaddress;theaddressoftheinstructionfollowing the goto instruction

.Areturnfromprocedurecalleeisimplementedby

```
GOTO *callee.static-area
```

For the call statement, we need to save the return address somewhere and then jump tothe location of the callee function. And to return from a function, we have to access the return address as stored by its caller, and then jump to it. So for call, we first say: MOV #here+20, callee.static-area. Here, #here refers to the location of the current MOV instruction, and callee.static- area is a fixed location in memory. 20 is added to #here here, as the code corresponding to the call instruction takes 20 bytes (at 4 bytes for each parameter: 4*3 for this instruction, and 8 for the next). Then we say GOTO callee. code-area, to take us to the code of the callee, as callee.codearea is merely the address where the code of the callee starts. Then a return from the callee is implemented by: GOTO *callee.static area. Note that this works only because callee.static-area is a constant.

Example:

.Assumeeach action blocktakes20 bytesof space .Start address ofcodefor c andp is 100and 200

```
100:ACTION-l
120: MOV140,364
132:GOTO200
140:ACTION-2
160: HALT
:
200:ACTION-3
220:GOTO*364
```

. The activation	:
Records	300:
arestatically	304:
allocatedstarting	:
ataddresses	364:
300and 364.	368:

This example corresponds to the code shown in slide 57. Statically we say that the code for c starts at 100 and that for p starts at 200. At some point, c calls p. Using the strategy discussed earlier, and assuming that callee.staticarea is at the memory location 364, we get the code as given. Here we assume that a call to 'action' corresponds to a single machine instruction which takes 20 bytes.

STACK ALLOCATION:.Position oftheactivationrecordis notknownuntilrun time

∑ .Positionisstoredinaregisteratruntime,andwordsintherecordareaccessedwithan offset from the register

∑ .Thecodeforthefirstprocedureinitializesthestackbysetting upSPtothestartofthe stack area

MOV#Stackstart,SP

codeforthefirstprocedure

HALT

In stack allocation we do not need to know the position of the activation record until run-time. This gives us an advantage over static allocation, as we can have recursion. So this is used in many modern programming languages like C, Ada, etc. The positions of the activations are stored in the stack area, and the position for the most recent activation is pointed to by the stack pointer. Words in a record are accessed with an offset from the register. The code for the first procedure initializes the stack by setting up SP to the stack area by the following command: MOV #Stackstart, SP. Here, #Stackstart is the location in memory where the stack starts.

AprocedurecallsequenceincrementsSP,savesthereturnaddressandtransferscontroltothe called procedure

ADD#caller.recordsize,SP

MOVE #here+ 16, *SP

GOTO callee.code_area

Consider the situation when a function (caller) calls the another function(callee), then procedure call sequence increments SP by the caller record size, saves the return address and transfers control to the callee by jumping to its code area. In the MOV instruction here, we only need to add 16, as SP is a register, and so no space is needed to store *SP. The activations keep getting pushed on the stack, so #caller.recordsize needs to be added to SP, to update the value of SP to its new value. This works as #caller.recordsize is a constant for a function, regardless ofthe particular activation being referred to.

DATASTRUCTURES: Followingdatastructures areusedtoimplementsymbol tables

LISTDATASTRUCTURE:Couldbeanarraybasedorpointerbasedlist.Butthis implementation is

- Simplest to implement
- Useasingle arraytostorenames andinformation
- Searchfor anameislinear
- Entryandlookup areindependent operations
- Costofentryandsearchoperations areveryhighandlot oftimegoes intobook keeping

Hashtable: HashtableisadatastructurewhichgivesO(1)performanceinaccessingany element of it. It uses the features of both array and pointer based lists.

-Theadvantagesareobvious

REPRESENTINGSCOPE INFORMATION

The entries in the symbol table are for declaration of names. When an occurrence of a name in the source text is looked up in the symbol table, the entry for the appropriate declaration, according to the scoping rules of the language, must be returned. A simple approach is to maintain a separate symbol table for each scope.

Most closely nested scope rules can be implemented by adapting the data structuresdiscussed in theprevious section. Each procedure is assigned auniquenumber. If thelanguageis block-structured, the blocks must also be assigned unique numbers. The name is represented as a pair of a number and a name. This new name is added to the symbol table. Most scope rules can be implemented in terms of following operations:

a) Lookup- find themost recentlycreatedentry.
b) Insert-makeanew entry.
c) Delete-removethemostrecentlycreatedentry.
d) Symboltablestructure
e) .Assignvariablestostorageclassesthatprescribescope,visibility,andlifetime

f) -scoperulesprescribethesymbol tablestructure

g) -scope:unitof staticprogramstructure withone ormorevariabledeclarations

h) - scope maybenested

i) .Pascal:proceduresarescopingunits

j) .C: blocks,functions, filesarescopingunits

k) .Visibility,lifetimes,global variables

l) . Common (in Fortran)

m) .Automaticorstack storage

n) .Staticvariables

o) **storage class :** A storage class is an extra keyword at the beginning of a declaration which modifies the declaration in some way. Generally, the storage class (if any) is the first word in the declaration, preceding the type name. Ex. static, extern etc.

p) Scope: The scope of a variable is simply the part of the program where it may beaccessed or written. It is the part of the program where the variable's name may be used. If a variable is declared within a function, it is local to that function. Variables of the same name may be declared and used within other functions without any conflicts. For instance,

q)

```
int fun1()
{

   inta;
   int b;
   ....

}

int fun2()
{

   inta;
   int c;
   ....

}
```

Visibility: The visibility of a variable determines how much of the rest of the program canaccessthatvariable.Youcanarrangethatavariableisvisibleonlywithinonepartof one function, or in one function, or in one source file, or anywhere in the program.

r) **Local and Global variables:** A variable declared within the braces {} of a function is visible only within that function; variables declared within functions are called local variables. On the other hand, a variable declared outside of any function is a global variable , and it is potentially visible anywhere within the program.

s) **Automatic Vs Static duration:** How long do variables last? By default, local variables (those declared within a function) have automatic duration: they spring into existence whenthefunctioniscalled,andthey(andtheirvalues)disappearwhenthefunction returns. Global variables, on the other hand, have static duration: theylast, and the values stored in them persist, for as long as the program does. (Of course, the values can in general still be overwritten, so they don't necessarily persist forever.) By default, local variables have automatic duration. To give them static duration (so that, instead ofcoming and going as the function is called, they persist for as long as the function does), you precede their declaration with the static keyword: static int i; By default, adeclaration of a global variable (especially if it specifies an initial value) is the defining instance. To make it an external declaration, of a variable which is defined somewhere else, you precede it with the keyword extern: extern int j; Finally, to arrange that a global variable is visible only within its containing source file, you precede it with the static keyword: staticint k; Noticethat thestatickeywordcan do two different things: it adjusts the duration of a local variable from automatic to static, or it adjusts the visibility of a global variable from truly global to private-to-the-file.

t) Symbol attributesandsymboltable entries

u) Symbolshaveassociated attributes

v) Typicalattributesarename,type, scope,size,addressingmode etc.

w) Asymboltableentrycollectstogetherattributessuchthattheycanbeeasilysetand retrieved

x) Exampleoftypicalnamesinsymboltable

Name	Type
name	characterstring
class	enumeration
size	integer
type	enumeration

LOCALSYMBOLTABLEMANAGEMENT :

Followingareprototypesoftypicalfunctiondeclarationsusedformanaginglocalsymboltable. The right hand side of the arrows is the output of the procedure and the left side has the input.

NewSymTab : SymTab →SymTab
DestSymTab : SymTab →SymTab
InsertSym : SymTab X Symbol →boolean
LocateSym:SymTabXSymbol→boolean
GetSymAttr : SymTab X Symbol X Attr→boolean
SetSymAttr:SymTabXSymbolXAttrXvalue→boolean
NextSym : SymTab X Symbol →Symbol
MoreSyms:SymTabXSymbol→boolean

MID EXAMINATION QUESTION PAPER

Estd.: 2001

Balaji Institute of Technology & Science
Laknepally, NARSAMPET, Warangal – 506 331
Accredited by NBA, NAAC & ISO 9001:2015 Certified Institution
(Affiliated to JNTUH, Hyderabad and Approved by the AICTE, New Delhi)
www.bitswgl.ac.in, email:principal@bitswgl.ac.in::Ph.98660 50044, Fax 08718-230521

MID-I Examination, FEB-2025

Course: B.Tech, Branch-CSM (A&B), **Year & Semester: II-IISem**

Subject: Automata Theory and compiler design **Date: 12-02-2025**

Duration: 2 Hour, Max Marks: 30

PART-A

Answer any four Questions **Marks [20]**

1. **a)** a finite automaton accepting all strings over {0, 1} having even number of 0's and even number of 1's ?
   b) Construct a finite automaton accepting all strings over {0, 1} starts with abb?
2. **Construct** a DFA for the regular expression (0+1)* using indirect method?
3. a) List down the Identity Rules for the Regular Expression?
   b) Explain the Arden's theorem?
4. **Explain** with an example about Minimization of the DFA?
5. What is Grammar ? Explain CFG with an example ?
6. Explain pumping lemma concept with an example ?

PART –B

Multiple choice questions **Marks [5]**

1. There are ________ tuples in finite state machine. [ ]
a) 4
b) 5
c) 6
d) unlimited

2. Transition function maps. [ ]
a) Σ * Q -> Σ
b) Q * Q -> Σ
c) Σ * Σ -> Q
d) Q * Σ -> Q

3. Number of states requires accepting string ends with 10. [ ]
a) 3

b) 2
c) 1
d) can’t be represented.

4. Extended transition function is. [ ]
a) Q * Σ* -> Q
b) Q * Σ -> Q
c) Q* * Σ* -> Σ
d) Q * Σ -> Σ

5. δ*(q,ya) is equivalent to . [ ]

a) δ((q,y),a)
b) δ(δ*(q,y),a)
c) δ(q,ya)
d) independent from δ notation

6. String X is accepted by finite automata if [ ] .
a) δ*(q,x) E A
b) δ(q,x) E A
c) δ*(Q0,x) E A
d) δ(Q0,x) E A

7. Languages of a automata is [ ]
a) If it is accepted by automata
b) If it halts
c) If automata touch final state in its life time
d) All language are language of automata

8. Language of finite automata is. [ ]
a) Type 0
b) Type 1
c) Type 2
d) Type 3

9. Finite automata requires minimum _______ number of stacks.
a) 1
b) 0
c) 2
d) None of the mentioned

10. Number of final state require to accept Φ in minimal finite automata. [ ]
a) 1
b) 2
c) 3
d) None of the mentioned

Fill in the Blanks **Marks [5]**

11. How many DFA's exits with two states over input alphabet {0,1} ________________

12. The basic limitation of finite automata is that_________________

13. Moore Machine is an application of______________

14 . In Moore machine, output is produced over the change of________-

15. The finite automata is called NFA when there exists____________ for a specific input from current state to next state

16. ε-closure of state is combination of self state and ________________

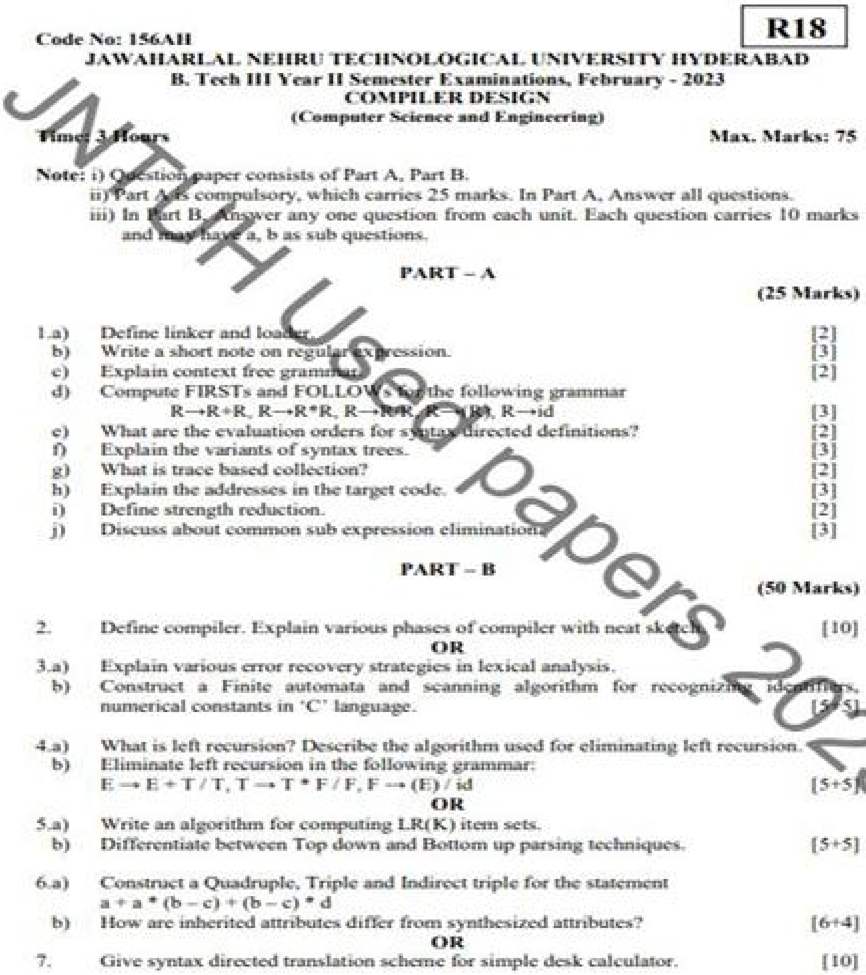
17. In mealy machine, the O/P depends upon_____________

18. The major difference between Mealy and Moore machine is about__________-

19. Mealy and Moore machine can be categorized as:

20.An e-NFA is ___________ tuple representation.

Previous year questions



Code No: 156AH **R18**

JAWAHARLAL NEHRU TECHNOLOGICAL UNIVERSITY HYDERABAD

B. Tech III Year II Semester Examinations, February - 2023

COMPILER DESIGN

(Computer Science and Engineering)

Time: 3 Hours **Max. Marks: 75**

Note: i) Question paper consists of Part A, Part B.
ii) Part A is compulsory, which carries 25 marks. In Part A, Answer all questions.
iii) In Part B, Answer any one question from each unit. Each question carries 10 marks and may have a, b as sub questions.

PART – A

(25 Marks)

1.a) Define linker and loader. [2]
b) Write a short note on regular expression. [3]
c) Explain context free grammar. [2]
d) Compute FIRSTs and FOLLOWs for the following grammar
R→R+R, R→R*R, R→RR, R→(R), R→id [3]
e) What are the evaluation orders for syntax directed definitions? [2]
f) Explain the variants of syntax trees. [3]
g) What is trace based collection? [2]
h) Explain the addresses in the target code. [3]
i) Define strength reduction. [2]
j) Discuss about common sub expression elimination. [3]

PART – B

(50 Marks)

2. Define compiler. Explain various phases of compiler with neat sketch. [10]

OR

3.a) Explain various error recovery strategies in lexical analysis.
b) Construct a Finite automata and scanning algorithm for recognizing identifiers, numerical constants in 'C' language. [5+5]

4.a) What is left recursion? Describe the algorithm used for eliminating left recursion.
b) Eliminate left recursion in the following grammar:
E → E + T / T, T → T * F / F, F → (E) / id [5+5]

OR

5.a) Write an algorithm for computing LR(K) item sets.
b) Differentiate between Top down and Bottom up parsing techniques. [5+5]

6.a) Construct a Quadruple, Triple and Indirect triple for the statement
a + a * (b – c) + (b – c) * d
b) How are inherited attributes differ from synthesized attributes? [6+4]

OR

7. Give syntax directed translation scheme for simple desk calculator. [10]

UNIT WISE IMPORTANT QUESTIONS

AUTOMATA THEORY AND COMPILER DESIGN IMPORTANTQUESTIONS.

Unit-IV

SHORT QUESTIONS:

Definecompiler.
WhatisContextfreegrammar?
Definepre-processor.Whatarethe functionsof pre-processor?
What is input buffer?
Differentiatecompilerandinterpreter
What is input buffering?
Definethe followingterms:a)Lexemeb) Token
Defineinterpreter.
WhatarethedifferencesbetweentheNFA andDFA?

LONG questions:

Explainthe variousphasesof acompilerwith anillustrativeexample
DefineRegularexpression.ExplainthepropertiesofRegularexpressions.
Differentiatebetweentop downandbottomupparsing techniques.
ConstructanFA equivalenttotheregularexpression (0+1)*(00+11)(0+1)*
Explainthe various phases of acompilerin detail. Alsowritedown theoutput forthefollowing expression:position:=initial+rate* 60
ConstructanFA equivalenttotheregularexpression 10+(0+11)0*1
Defineaugmented grammar
ComparetheLRParsers.
CompareandcontrastLRand LLParsers
Differentiatebetweentop downparsers
DefineDeadcodeelimination?
Eliminateimmediateleftrecursionforthefollowinggrammar: E- >E+T | T T->T*F |F F-> (E)| id
MentionthetypesofLR parser.
Explainbottomupparsingmethod

Discussinaboutleft recursionandleft factoringwith examples.
Constructthepredictiveparserforthefollowinggrammar S- >(L)/a L->L,S/S
CheckwhetherthefollowinggrammarisSLR(1)ornot.ExplainyouranswerwithReasons. S→L=R S→R L→*R L→id R→L
ConstructSLRparsetablefor S->L=R/R R->L L->*R/id
Stateandexplaintherulestocomputefirstandfollowfunctions E- >E+T/T T->T*F/F F->F*/a/b
ConstructCLRparsetablefor S->L=R/R R->L L->*R/id
ConstructtheLRParsingtableforthefollowinggrammar: E→E + T | T T→T*F|F F → (E)/id
ConstructanLALRParsingtableforthefollowinggrammar: E-> E+T |T T->T*F|F F->id
FindtheSLRparsingtableforthegivengrammar: E- >E+E | E*E | (E) | id. Andparsethesentence(a+b)*

UNIT-5

SHORTQUESTIONS:

DefineTypeEquivalence
Explaintherole ofintermediatecodegeneratorincompilation process
Defineleftmostderivationandrightmostderivationwith example
Whatarethevarioustypesofintermediate code representation?
Writeanote onthespecification ofasimple typechecker.
Explainintermediatecode representations?
Definetype expression withan example?
Stategeneralactivation record?
Explaintype expressionandtype systems

LONGQUESTIONS:

Explainin briefabout equivalenceof typeexpressions with examples
Explainabout TypecheckingandTypeConversion with examples
Whatisathreeaddresscode?Mentionitstypes.Howwouldyouimplementthethree address statements?Explainwithexamples.
Whatistypechecker?Explainthespecificationof asimpletype checker
Translatethefollowingexpression: (a + b) * (c + d) + (a + b + c) into a)Quadruplesb)Triples
Constructaquadruple,triplesforthefollowingexpression: a +a*(b-c)+(b-c)*d?
Explainvariousstorageallocationstrategieswithexamples.
Explainstaticandstackstorageallocations?

Writethe quadrupleforthefollowingexpression (x +y)∗(y+z) +(x+y+z)
WhatisaDAG?Mentionits applications.
WhatareAbstractSyntaxtrees?
Defineaddressdescriptor andregisterdescriptor
Discussaboutcommon subexpression elimination
Whatis aFlow graph?
Defineconstantfolding?
Definereductioninstrength?

LONGQUESTIONS:

Explaintheissueandthe differencebetweentheheap allocatedactivationrecordsversus stack allocatedactivation records
Writethe principalsources of optimization
Discussabout the following: a) CopyPropagation b) DeadcodeElimination c) Codemotion.
ExplainLazy-codemotionproblemwithanalgorithm
Explainthe followingwith an example: a) Redundantsub expression elimination b) Frequencyreduction c) Copypropagation
Explainvariousmethod tohandlepeepholeoptimization.
Explain thefollowingpeephole optimizationtechniques: a) EliminationofRedundantCode b) EliminationofUnreachableCode
Illustrateloopoptimizationwithsuitableexample.
Explainvarious codeoptimization techniques in detail.

Whataretheinduction variables?
Explain about code motion.
Whatareinductionvariables?Whatisinductionvariableelimination?
Whatismachine independentcode optimization?
Writeashort note on copyPropagation
Whataretheinduction variables?
Writeashort note on Flowgraph.
Explaindata-flowschemason basicblocks withflow graphs
ExplainLazy-codemotionproblemwithanalgorithm
Explaininbriefabout different Principalsourcesof optimizationtechniqueswith suitable examples.

GiveanexampletoshowhowDAGisused forregisterallocation
Explainindetailaboutmachinedependent codeoptimizationtechniqueswiththeir drawbacks
Explainin briefabouttheissues in thedesign ofcodegenerator.
Explainindetailaboutpeephole optimization.
Explainmachinedependent andmachineindependent optimization?
Explaindata-flowanalysisofstructuralprograms.
Explainindetailtheprocedurethateliminates globalcommonsub expression

Tutorial problems with blooms mapping

In the context Automata Theory and compiler design , Bloom's Taxonomy can be a useful framework for structuring problem-solving and learning outcomes. Bloom's Taxonomy categorizes learning objectives into levels of complexity: Remember, Understand, Apply, Analyze, Evaluate, and Create. I'll present a few tutorial problems that correspond to different levels of Bloom's Taxonomy.

1. Remember (Knowledge):

- **Problem 1:**
  Define the following terms:
  a. Alphabet
  b. String
  c. Language
  d. Finite Automaton
  e. Regular Expression

- **Solution:**
  Provide definitions for each term, with examples if needed.
  - **Alphabet**: A finite set of symbols, e.g., $\Sigma = \{0, 1\}$.
  - **String**: A finite sequence of symbols from an alphabet, e.g., "101".
  - **Language**: A set of strings over an alphabet, e.g., L = { "0", "1", "01", "10" }.
  - **Finite Automaton**: A machine with a finite set of states used to recognize regular languages.
  - **Regular Expression**: A formal notation for defining regular languages.

---

2. Understand (Comprehension):

- **Problem 2:**
  Explain why the set of strings consisting of an even number of 0s and an odd number of 1s is not regular. Use the pumping lemma to justify your answer.

  Solution:

- **Solution:**
  This problem requires an understanding of the pumping lemma. You would show that the string "000111" cannot be pumped without breaking the conditions of having an even number of 0s and an odd number of 1s. Pumping a portion of the string could lead to an imbalance in the numbers of 0s and 1s. Thus, the language is not regular.

---

3. Apply (Application):

- **Problem 3:**
  Construct a Deterministic Finite Automaton (DFA) that accepts the language of strings over {0, 1} where the string contains at least one '1'.

Solution:

- The DFA should have two states:
  - **q0 (initial state)**: If a '0' is read, the machine stays in state q0; if a '1' is read, the machine transitions to state q1.
  - **q1 (accepting state)**: Once in state q1, the machine stays in q1, Once in state q1, the machine stays in q1, accepting any further input.

State transitions:

- q0 → on input '0' → q0
- q0 → on input '1' → q1
- q1 → on input '0' → q1
- q1 → on input '1' → q1

Acceptance condition: The string is accepted if the machine ends in state q1.

4. Analyze (Analysis):

- **Problem 4:**
  Given the context-free grammar:
- Analyze the language generated by this grammar. What kind of strings does it accept?

  Solution:
  The grammar generates strings that consist of an equal number of 'a's followed by an equal number of 'b's, with no other characters. The analysis of the grammar reveals that the language is of the form { a^n b^n | n ≥ 0 }. This is a classic example of a context-free language.

---

5. Evaluate (Evaluation):

- **Problem 5:**
  Evaluate whether the following language is context-free or not:

  Solution:
  This language is **not context-free**. The intuition comes from the fact that a context-free grammar cannot ensure that two arbitrary halves of a string are identical. The pumping lemma for context-free languages can be used to formally prove that this language cannot be context-free.

6. Create (Synthesis):

- **Problem 6:**
  Create a pushdown automaton (PDA) for the language L = { w ∈ {a, b}* | the number of 'a's is equal to the number of 'b's }.

  Solution:
  To create a PDA for this language, one would design a machine that pushes symbols onto a stack

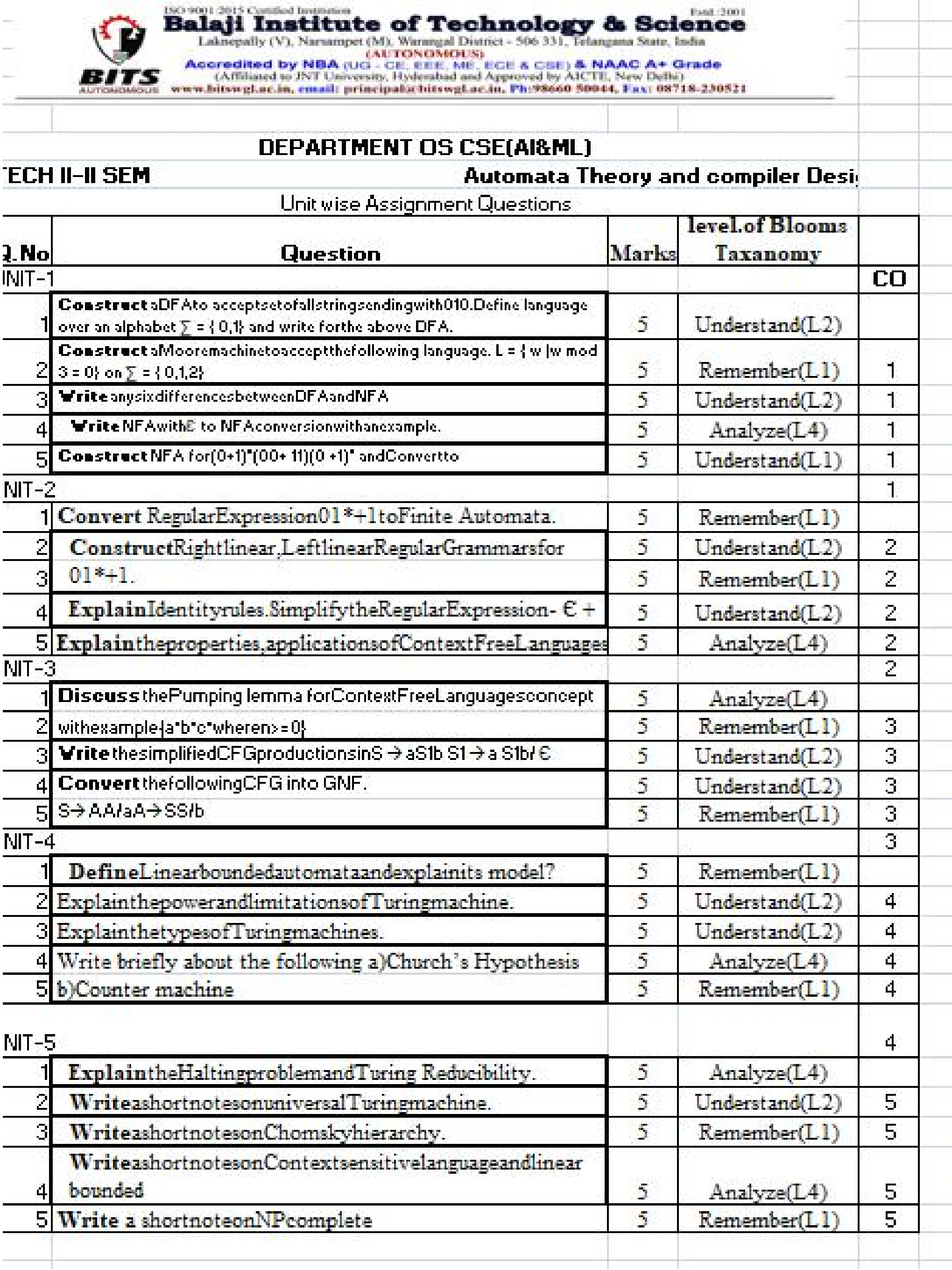
when it reads 'a' and pops symbols when it reads 'b', ensuring that the number of 'a's and 'b's are equal. The PDA would need to handle the following transitions:

- On reading 'a', push 'A' onto the stack.
- On reading 'b', pop an 'A' from the stack.
- Accept if the stack is empty after reading the entire input string.

Transitions:

- (q0, a, ε) → (q0, A)
- (q0, b, A) → (q0, ε)
- (q0, ε, ε) → (q_accept, ε) (if the stack is empty)

Assignment questions with blooms mapping



ISO 9001:2015 Certified Institution Estd.:2001
Balaji Institute of Technology & Science
Laknepally (V), Narsampet (M), Warangal District - 506 331, Telangana State, India
(AUTONOMOUS)
Accredited by NBA (UG - CE, EEE, ME, ECE & CSE) & NAAC A+ Grade
(Affiliated to JNT University, Hyderabad and Approved by AICTE, New Delhi)
BITS AUTONOMOUS www.bitswgl.ac.in, email: principal@bitswgl.ac.in, Ph:98660 50044, Fax: 08718-230521

DEPARTMENT OS CSE(AI&ML)

ECH II-II SEM Automata Theory and compiler Desi

Unit wise Assignment Questions

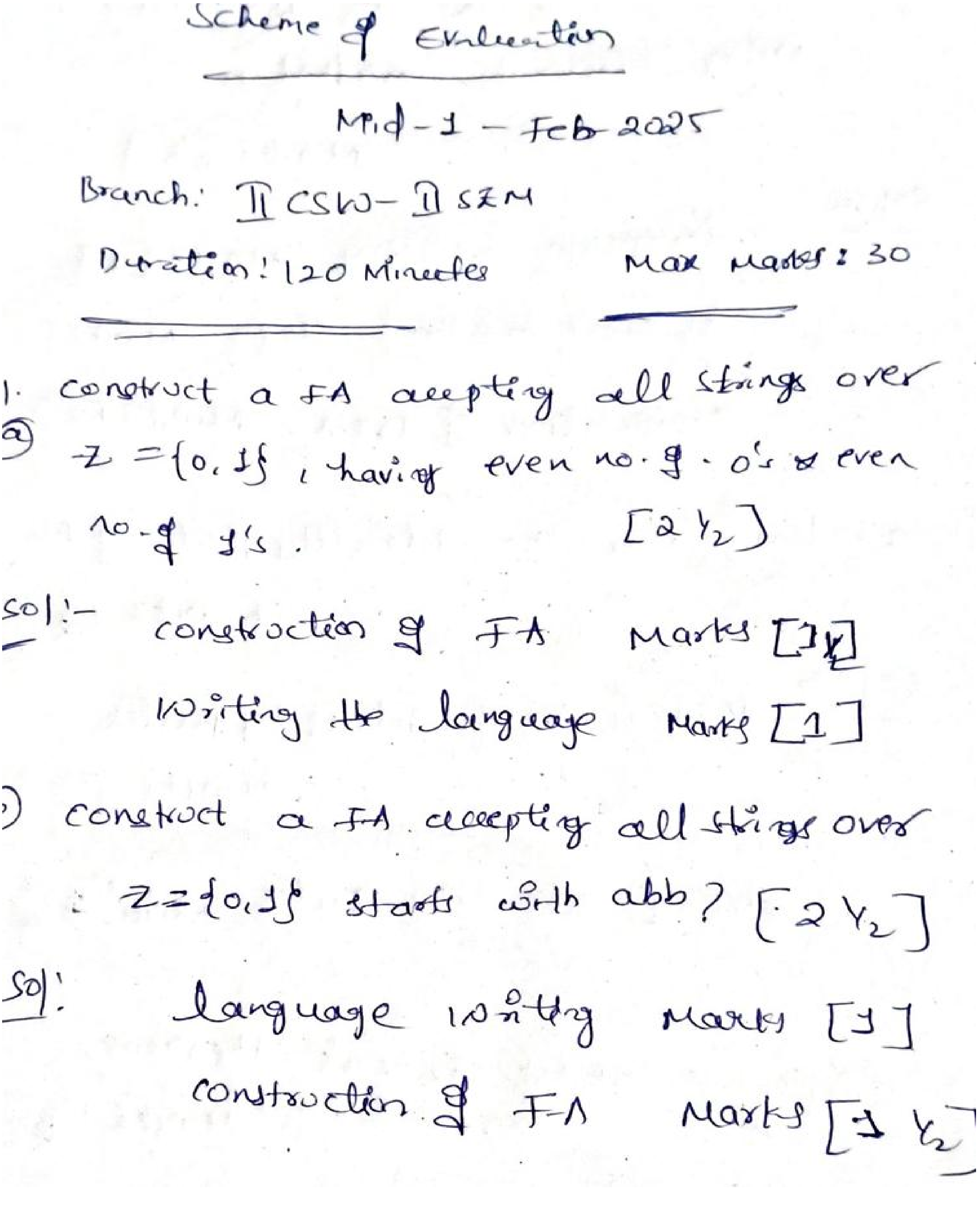
Q.No	Question	Marks	level.of Blooms Taxanomy	
NIT-1				CO
1	**Construct** aDFAto acceptsetofallstringsendingwith010.Define language over an alphabet ∑ = { 0,1} and write forthe above DFA.	5	Understand(L2)	
2	**Construct** aMooremachinetoacceptthefollowing language. L = { w |w mod 3 = 0} on ∑ = { 0,1,2}	5	Remember(L1)	1
3	**Write** anysixdifferencesbetweenDFAandNFA	5	Understand(L2)	1
4	**Write** NFAwith∈ to NFAconversionwithanexample.	5	Analyze(L4)	1
5	**Construct** NFA for(0+1)*(00+ 11)(0 +1)* andConvertto	5	Understand(L1)	1
NIT-2				1
1	**Convert** RegularExpression01*+1toFinite Automata.	5	Remember(L1)	
2	**Construct**Rightlinear,LeftlinearRegularGrammarsfor	5	Understand(L2)	2
3	01*+1.	5	Remember(L1)	2
4	**Explain**Identityrules.SimplifytheRegularExpression- ∈ +	5	Understand(L2)	2
5	**Explain**theproperties,applicationsofContextFreeLanguages	5	Analyze(L4)	2
NIT-3				2
1	**Discuss** thePumping lemma forContextFreeLanguagesconcept	5	Analyze(L4)	
2	withexample{a*b*c*wheren>=0}	5	Remember(L1)	3
3	**Write** thesimplifiedCFGproductionsinS → aS1b S1 → a S1b/ ∈	5	Understand(L2)	3
4	**Convert** thefollowingCFG into GNF.	5	Understand(L2)	3
5	S→ AA/aA→ SS/b	5	Remember(L1)	3
NIT-4				3
1	**Define**Linearboundedautomataandexplainits model?	5	Remember(L1)	
2	Explainthepowerandlimitationsof Turingmachine.	5	Understand(L2)	4
3	ExplainthetypesofTuringmachines.	5	Understand(L2)	4
4	Write briefly about the following a)Church's Hypothesis	5	Analyze(L4)	4
5	b)Counter machine	5	Remember(L1)	4
NIT-5				4
1	**Explain**theHaltingproblemandTuring Reducibility.	5	Analyze(L4)	
2	**Write**ashortnotesonuniversalTuringmachine.	5	Understand(L2)	5
3	**Write**ashortnotesonChomskyhierarchy.	5	Remember(L1)	5
4	**Write**ashortnotesonContextsensitivelanguageandlinear bounded	5	Analyze(L4)	5
5	**Write** a shortnoteonNPcomplete	5	Remember(L1)	5

List of students

1	23C31A6601	ADAPA RAKESH
2	23C31A6602	AITHA PRAVEEN
3	23C31A6603	AKARAPU ARPAN
4	23C31A6604	AMREEN
5	23C31A6605	ARURI PAVAN
6	23C31A6606	ARUTLA AJAY
7	23C31A6607	ATLA SAIKRISHNA
8	23C31A6608	BAIRABOINA PREETHI
9	23C31A6609	BAJJURI SANTHOSH
10	23C31A6610	BALABAKTHULA MANISHA
11	23C31A6611	BATTHULA DEEPIKA
12	23C31A6612	BEERUM LAXMI SRINIVAS
13	23C31A6613	BOINI AJAY
14	23C31A6614	BOLLENA VARSHA
15	23C31A6615	BOMMANAPELLY POOJITHA
16	23C31A6616	BURA SANJAY
17	23C31A6617	CHINNALA ARJUN
18	23C31A6618	CHINNAPALLY ASHWITHA
19	23C31A6619	CHINTHIREDDY PRAVEEN
20	23C31A6620	DARAVATH JASHWANTH
21	23C31A6621	DASARI LAHARI SRI
22	23C31A6622	DASARI SRINIVAS
23	23C31A6623	DASU SAIPRIYA
24	23C31A6624	DOLI ARCHANA
25	23C31A6625	DUDDE NITHISH
26	23C31A6626	DUPPATI PRANEETH
27	23C31A6627	EGA SHIVANI
28	23C31A6628	ELDI KARTHIK
29	23C31A6629	ENUGALA BHAVANI
30	23C31A6630	GAJJALA VARUN
31	23C31A6631	GANDHAM KARTHIK
32	23C31A6632	GANGINENI NAVEEN KUMAR
33	23C31A6633	GANJI KAVYA SHRI

34	23C31A6634	GOLI LAXMI PRASANNA
35	23C31A6635	GUJJULA RAMYA
36	23C31A6636	GUNDAMALA ARUN
37	23C31A6637	GUNISHETTI GANGOTHRI
38	23C31A6638	INDLA SANDHYA
39	23C31A6639	INTSHAR ALAM
40	23C31A6640	IPPA RITHWIK
42	23C31A6642	KANDUKURI JAYALAXMI
43	23C31A6643	KANKALA SUSHMITHA
44	23C31A6644	KANNAM SHIVA SAI
45	23C31A6645	KARRA SAHITHI REDDY
46	23C31A6646	KASANABOINA BHASKAR
47	23C31A6647	KATLA ARUN KUMAR
48	23C31A6648	KEESARI SRIRAM
49	23C31A6649	KOLA SIDDHARTHA
50	23C31A6650	KONTAM DIVYA
51	23C31A6651	KOTHA DIVYA
52	23C31A6652	KOTTURI CHAITHANYA
53	23C31A6653	KUCHANA SRAVANI
54	23C31A6654	LAKKA VARUN RAJ
55	23C31A6655	LEKKALA VARAPRASAD
57	24C35A6602	JADALA SHIVA KUMAR
58	24C35A6603	KUCHANA SANDEEP
59	24C35A6604	LAKKARSU SUNNY
60	24C35A6605	MOHAMMAD ARIF AHMED
61	24C35A6606	NARUGULA SAI CHANDANA

Scheme and evaluation of internal tests



Scheme of Evaluation

Mid-1 - Feb 2025

Branch: II CSW - II SEM

Duration: 120 Minutes Max Marks: 30

1. construct a FA accepting all strings over
a) $\Sigma = \{0, 1\}$, having even no. of 0's & even no. of 1's. [2 1/2]

Sol:- construction of FA Marks [1 1/2]

Writing the language Marks [1]

) construct a FA accepting all strings over $\Sigma = \{0, 1\}$ starts with abb? [2 1/2]

Sol: language writing Marks [1]

construction of FA Marks [1 1/2]

② Construct a DFA for the RE $(0+1)^*$ Using indirecte method?

Marks [5]

Sol:- . language writing Marks [1]

Indireed method steps Marks [2]

Construction of DFA Marks [2]

③ a. List down the Identity Rules of RE?

Marks [2½]

Sol:- Definition of identity Rules

Marks [1]

List of identity Rules Marks [1½]

④ Explain about Arden's theorem?

Marks [2½]

Sol:- Arden's theorem Definition

④ Explain with an example about Minimization of the DFA?

Marks [5]

Sol:- Steps for Minimizing the DFA

Marks [2]

Example problem explanation

Marks [3]

⑤ Q: What is Grammar? Explain with an example?

Marks [5]

Sol:- Grammar Definition Marks [2]

Examples of Grammar [2]

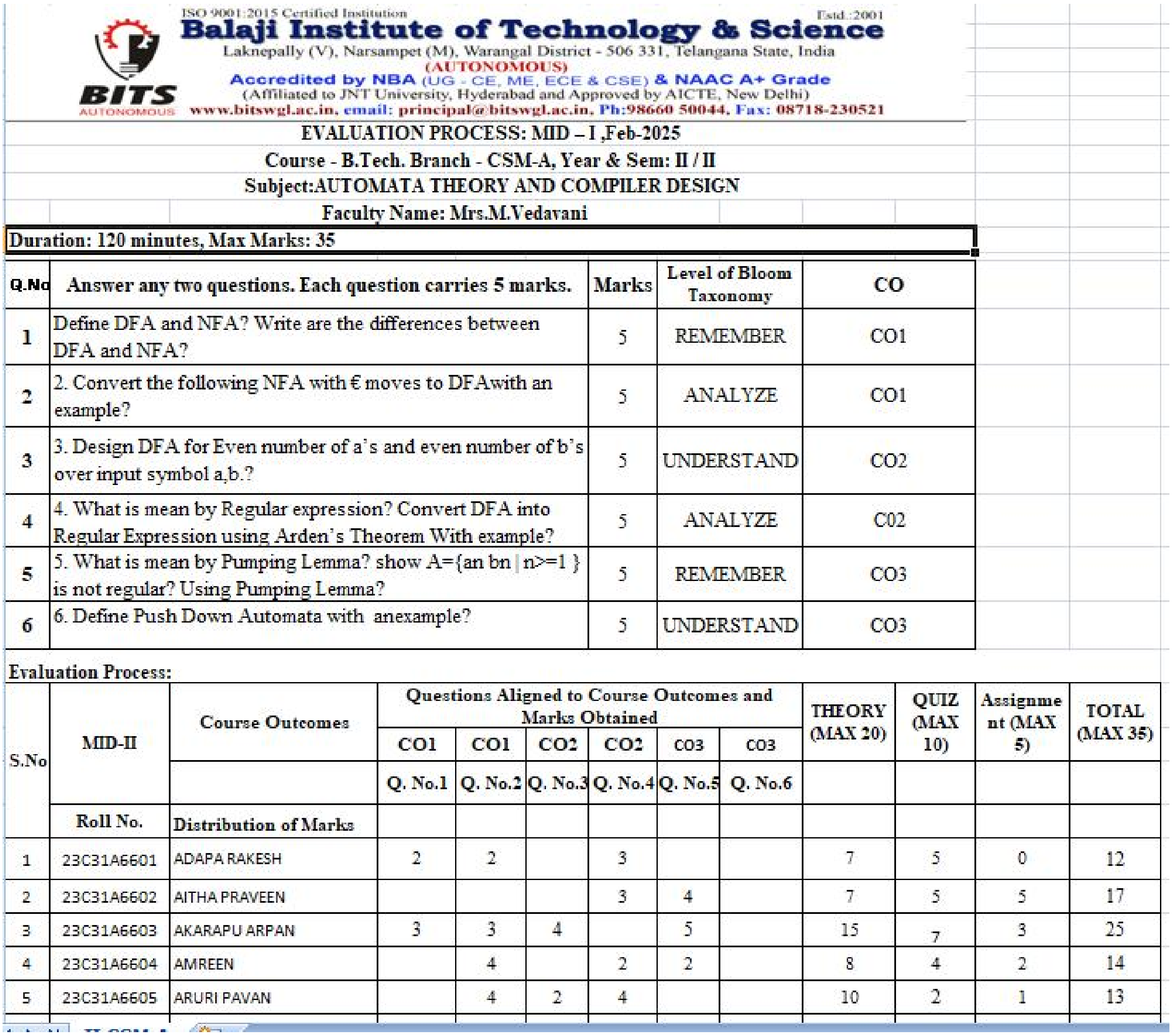
parse tree Marks [1]

⑥ Q:- Explain pumping Lemma concept with an example?

Marks [5]

Sol:- Definition

Marks sheet



ISO 9001:2015 Certified Institution Estd.:2001

Balaji Institute of Technology & Science

Laknepally (V), Narsampet (M), Warangal District - 506 331, Telangana State, India

(AUTONOMOUS)

Accredited by NBA (UG - CE, ME, ECE & CSE) & NAAC A+ Grade

(Affiliated to JNT University, Hyderabad and Approved by AICTE, New Delhi)

BITS AUTONOMOUS www.bitswgl.ac.in, email: principal@bitswgl.ac.in, Ph:98660 50044, Fax: 08718-230521

EVALUATION PROCESS: MID – I ,Feb-2025

Course - B.Tech. Branch - CSM-A, Year & Sem: II / II

Subject:AUTOMATA THEORY AND COMPILER DESIGN

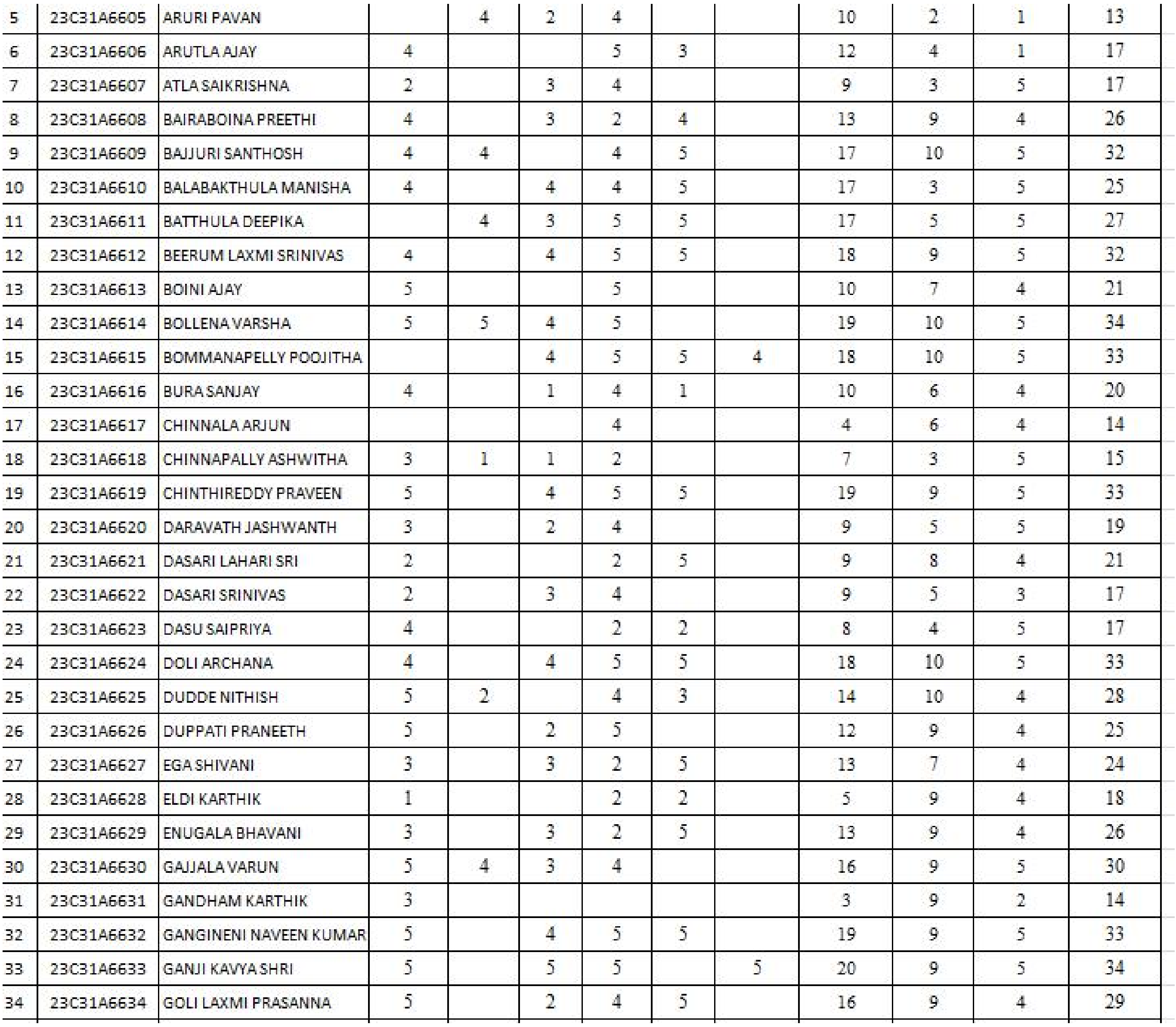
Faculty Name: Mrs.M.Vedavani

Duration: 120 minutes, Max Marks: 35

Q.No	Answer any two questions. Each question carries 5 marks.	Marks	Level of Bloom Taxonomy	CO
1	Define DFA and NFA? Write are the differences between DFA and NFA?	5	REMEMBER	CO1
2	2. Convert the following NFA with € moves to DFAwith an example?	5	ANALYZE	CO1
3	3. Design DFA for Even number of a's and even number of b's over input symbol a,b.?	5	UNDERSTAND	CO2
4	4. What is mean by Regular expression? Convert DFA into Regular Expression using Arden's Theorem With example?	5	ANALYZE	C02
5	5. What is mean by Pumping Lemma? show A={an bn | n>=1 } is not regular? Using Pumping Lemma?	5	REMEMBER	CO3
6	6. Define Push Down Automata with anexample?	5	UNDERSTAND	CO3

Evaluation Process:

S.No	MID-II	Course Outcomes	Questions Aligned to Course Outcomes and Marks Obtained						THEORY (MAX 20)	QUIZ (MAX 10)	Assignment (MAX 5)	TOTAL (MAX 35)
			CO1	CO1	CO2	CO2	CO3	CO3				
			Q. No.1	Q. No.2	Q. No.3	Q. No.4	Q. No.5	Q. No.6				
	Roll No.	Distribution of Marks										
1	23C31A6601	ADAPA RAKESH	2	2		3			7	5	0	12
2	23C31A6602	AITHA PRAVEEN				3	4		7	5	5	17
3	23C31A6603	AKARAPU ARPAN	3	3	4		5		15	7	3	25
4	23C31A6604	AMREEN		4		2	2		8	4	2	14
5	23C31A6605	ARURI PAVAN		4	2	4			10	2	1	13



5	23C31A6605	ARURI PAVAN		4	2	4			10	2	1	13
6	23C31A6606	ARUTLA AJAY	4			5	3		12	4	1	17
7	23C31A6607	ATLA SAIKRISHNA	2		3	4			9	3	5	17
8	23C31A6608	BAIRABOINA PREETHI	4		3	2	4		13	9	4	26
9	23C31A6609	BAJJURI SANTHOSH	4	4		4	5		17	10	5	32
10	23C31A6610	BALABAKTHULA MANISHA	4		4	4	5		17	3	5	25
11	23C31A6611	BATTHULA DEEPIKA		4	3	5	5		17	5	5	27
12	23C31A6612	BEERUM LAXMI SRINIVAS	4		4	5	5		18	9	5	32
13	23C31A6613	BOINI AJAY	5			5			10	7	4	21
14	23C31A6614	BOLLENA VARSHA	5	5	4	5			19	10	5	34
15	23C31A6615	BOMMANAPELLY POOJITHA			4	5	5	4	18	10	5	33
16	23C31A6616	BURA SANJAY	4		1	4	1		10	6	4	20
17	23C31A6617	CHINNALA ARJUN				4			4	6	4	14
18	23C31A6618	CHINNAPALLY ASHWITHA	3	1	1	2			7	3	5	15
19	23C31A6619	CHINTHIREDDY PRAVEEN	5		4	5	5		19	9	5	33
20	23C31A6620	DARAVATH JASHWANTH	3		2	4			9	5	5	19
21	23C31A6621	DASARI LAHARI SRI	2			2	5		9	8	4	21
22	23C31A6622	DASARI SRINIVAS	2		3	4			9	5	3	17
23	23C31A6623	DASU SAIPRIYA	4			2	2		8	4	5	17
24	23C31A6624	DOLI ARCHANA	4		4	5	5		18	10	5	33
25	23C31A6625	DUDDE NITHISH	5	2		4	3		14	10	4	28
26	23C31A6626	DUPPATI PRANEETH	5		2	5			12	9	4	25
27	23C31A6627	EGA SHIVANI	3		3	2	5		13	7	4	24
28	23C31A6628	ELDI KARTHIK	1			2	2		5	9	4	18
29	23C31A6629	ENUGALA BHAVANI	3		3	2	5		13	9	4	26
30	23C31A6630	GAJJALA VARUN	5	4	3	4			16	9	5	30
31	23C31A6631	GANDHAM KARTHIK	3						3	9	2	14
32	23C31A6632	GANGINENI NAVEEN KUMAR	5		4	5	5		19	9	5	33
33	23C31A6633	GANJI KAVYA SHRI	5		5	5		5	20	9	5	34
34	23C31A6634	GOLI LAXMI PRASANNA	5		2	4	5		16	9	4	29

References, Journals, websites and E-links if any

TEXT BOOKS:

1. Introduction to Automata Theory, Languages, andComputation, 3ndEdition, JohnE. Hop croft, Rajeev Motwani, Jeffrey D. Ullman, Pearson Education.
2. Theory of Computer Science– Automata languages and computation, Mishra and Chandrashekaran, 2nd edition, PHI.

REFERENCEBOOKS:

1. Introduction to Languages and The Theory of Computation, John.C Martin,TMH.
2. Introduction to Computer Theory, DanielI.A.Cohen, JohnWiley.
3. A Text book on Automata Theory, P.K.Srimani, Nasir S.F.B, Cambridge University Press.
4. Introduction to the Theory of Computation, Michael Sipser, 3rdedition, Cengage Learning.
5. Introduction to Formal languages Automata Theory and computation Kamala Krithivasan, Rama R, Pearson.