Theory of Automata and Formal Language — Complete Exam Guide

BT-407 | B.Tech IV Semester | TU-831(A)

Built for Lakshya | Exam Prep | May 2026

Exam Date: May 8, 2026

Contents

Pattern Analysis & Exam Strategy 5
How the Paper is Structured . . . 5
Topics That Appeared in Both 2024 AND 2025 (Highest Probability) . . . 5
Target Score Breakdown for 60+/70 . . . 5
Time Management (3-Hour Exam) . . . 5

Master Cheat Sheet 7
All Tuple Definitions (Memorize Cold) . . . 7
Chomsky Hierarchy at a Glance . . . 7
Key Theorems (One-Liners) . . . 8
Decidability Quick Reference . . . 8

Topic 1 — DFA and NFA 10
The Idea . . . 10
DFA — Formal Definition . . . 10
NFA — Formal Definition . . . 10
DFA vs NFA — Comparison Table . . . 10
Diagrams . . . 11

Topic 2 — Grammar, CFG, Derivation Trees, Ambiguity 12
What is a Grammar? . . . 12
CFG — Context-Free Grammar . . . 12
Derivation Tree (Parse Tree) . . . 12
Leftmost vs Rightmost Derivation . . . 13
Ambiguity . . . 13

Topic 3 — Chomsky Hierarchy 16
The Idea . . . 16
The Four Types — Complete Description . . . 16
Summary Table — Memorize This . . . 17
Visual . . . 17

Topic 4 — Mealy and Moore Machines 18
The Idea . . . 18

Moore Machine — Formal Definition . . . 18
Mealy Machine — Formal Definition . . . 18
Mealy vs Moore — Comparison . . . 18
Mealy → Moore Conversion (4-Step Procedure) . . . 18
Moore → Mealy Conversion (Easier — 1 Step) . . . 19
Diagrams — Worked Example: Counting Substring "ab" . . . 19

Topic 5 — Turing Machine **20**
The Idea . . . 20
Components of a TM . . . 20
Formal Definition (7-tuple — memorize this exactly) . . . 20
Three Outcomes for Any Input . . . 21
Modifications of TM . . . 21
Universal Turing Machine (UTM) . . . 21

Topic 6 — TM Design for L = {$a^n b^n$ | n ≥ 1} **23**
Strategy (Plain English) . . . 23
States . . . 23
Tape Alphabet . . . 23
Transition Table . . . 23
State Diagram . . . 23
Trace for Input "aabb" . . . 23
Closing Statement to Write in Exam . . . 24

Topic 7 — TM Design for L = {$a^n b^n c^n$ | n ≥ 1} **26**
Why This Language Needs a TM . . . 26
Strategy . . . 26
States . . . 26
Transition Table . . . 26
State Diagram . . . 26
Trace Sketch for "aabbcc" . . . 26
Closing Statement . . . 28

Topic 8 — CFG Conversion to CNF **29**
Chomsky Normal Form Definition . . . 29
Conversion Algorithm — 4 Steps . . . 29
PYQ Worked Example — 2025 Q6 . . . 29

Topic 9 — Greibach Normal Form (GNF) **31**
Definition . . . 31
Property . . . 31
CNF vs GNF . . . 31
Example . . . 31

Topic 10 — DFA Design for Binary Numbers Divisible by 5 **32**
The Insight . . . 32
States and Construction . . . 32
Transition Table . . . 32
State Diagram . . . 32
Verification . . . 32
Formal Specification . . . 32

Topic 11 — Regular Expressions and Identities 33
Operators . . . 33
Regular Expression for "Strings Over {0,1} Containing At Least One 1" . . . 33
Identity to Prove — 2024 Section A Q5 . . . 33

Topic 12 — Arden's Theorem 35
Statement . . . 35
Use . . . 35
Procedure . . . 35
Worked Example . . . 35

Topic 13 — Pumping Lemma for Regular Languages 37
What It's For . . . 37
Statement . . . 37
Proof Template (How to Use It) . . . 37
Worked Example — Prove $L = \{a^n b^n \mid n \geq 1\}$ is Not Regular . . . 37
Another Example — $L = \{a^p \mid p \text{ is prime}\}$ . . . 37

Topic 14 — Kleene's Theorem 38
Statement . . . 38
Implications . . . 38
Tools for Each Direction . . . 38
ε-Transitions Bridge the Two . . . 38

Topic 15 — Push Down Automata (PDA) 39
The Idea . . . 39
Formal Definition (7-tuple) . . . 39
Stack Operations (Three Possibilities) . . . 39
Visual . . . 39
DPDA vs NPDA . . . 39
Two-Stack PDA — TM Equivalent . . . 41
PDA for $L = \{a^n b^n \mid n \geq 1\}$ . . . 41

Topic 16 — Linear Bounded Automaton (LBA) 42
The Idea . . . 42
Formal Definition . . . 42
What LBA Accepts . . . 42
Position in the Hierarchy . . . 42
Example . . . 42

Topic 17 — Halting Problem 44
Statement . . . 44
The Result . . . 44
Proof Outline (by Contradiction) . . . 44
Why It's Important . . . 44

Topic 18 — Post Correspondence Problem (PCP) 45
Statement . . . 45
Example . . . 45
The Result . . . 45
Significance . . . 45

Topic 19 — Church-Turing Thesis 46
Statement . . . 46
Origin . . . 46
Why It’s a “Thesis,” Not a “Theorem” . . . 46
Significance . . . 46

Topic 20 — Recursive vs Recursively Enumerable 47
Definitions . . . 47
The Distinction . . . 47
Key Theorem . . . 47
Famous Examples . . . 47
Containment . . . 47

Topic 21 — Decision Problems of CFL 48
Decidable Problems . . . 48
Undecidable Problems . . . 48

Topic 22 — Closure Properties of Regular Languages 49

2024 Paper — Complete Solutions 50
Section A (Very Short Answer — 5 × 2 = 10 marks) . . . 50
Section B (Short Answer — 9 marks each, attempt any 2 of 3) . . . 52
Section C (Long Answer — 14 marks each, attempt any 3 of 5) . . . 56

2025 Paper — Complete Solutions 66
Section A (Very Short Answer — 5 × 2 = 10 marks) . . . 66
Section B (Short Answer — 9 marks each, attempt any 2 of 3) . . . 67
Section C (Long Answer — 14 marks each, attempt any 3 of 5) . . . 71

Final Quick Reference Card 82
All Tuples in One Place . . . 82
Power Hierarchy . . . 82
Theorems One-Liners . . . 82
Famous Undecidable Problems . . . 82
Section A Definition Templates . . . 82

Top 7 Mistakes to Avoid in Exam 84

You’re Ready 84

Pattern Analysis & Exam Strategy

How the Paper is Structured

Section	Type	Marks	What to Attempt
A	Very Short Answer (5 questions × 2 marks)	10	All 5 mandatory
B	Short Answer (3 questions × 9 marks)	18	Any 2
C	Long / Descriptive (5 questions × 14 marks)	42	Any 3
Total		**70**	

Topics That Appeared in Both 2024 AND 2025 (Highest Probability)

Topic	2024	2025	What to Memorize
DFA vs NFA	C-9 (14m)	A-1 (2m)	Comparison table + 5-tuple
Mealy ↔ Moore conversion	C-11 (14m)	C-10 (14m)	4-step conversion procedure
Chomsky hierarchy	B-6 (9m)	B-7 (9m)	4-type table with examples
Turing machine design	C-10 (14m)	C-12 (14m)	TM for $a^n b^n$ AND $a^n b^n c^n$
Linear Bounded Automaton	C-12 (7m)	B-8 (9m)	Definition + tape bounded explanation
Pumping lemma	C-13 (7m)	—	Statement + 1 application
Halting Problem	C-13 (7m)	—	Statement + proof outline
Definitions block	A	A	TM, automata, grammar, CFG, etc.

Target Score Breakdown for 60+/70

- **Section A:** 9-10 / 10 (definitions are easy marks)
- **Section B:** 16-18 / 18 (Chomsky hierarchy is automatic)
- **Section C:** 33-37 / 42 (Mealy/Moore + TM design + one theory question)

Time Management (3-Hour Exam)

Time	Activity
0–5 min	Read entire paper, mark which Section B/C questions to attempt
5–15 min	Section A — answer all 5 (2 min each)
15–55 min	Section B — 2 questions, ~20 min each
55–155 min	Section C — 3 questions, ~33 min each
155–180 min	Review, neat-up diagrams, fill gaps

Master Cheat Sheet

All Tuple Definitions (Memorize Cold)

Machine	Tuple	Key Component
DFA	$(Q, \Sigma, \delta, q_0, F)$	$\delta: Q \times \Sigma \to Q$ (single state)
NFA	$(Q, \Sigma, \delta, q_0, F)$	$\delta: Q \times \Sigma \to 2^Q$ (set of states)
ε-NFA	$(Q, \Sigma, \delta, q_0, F)$	$\delta: Q \times (\Sigma \cup \{\varepsilon\}) \to 2^Q$
Moore	$(Q, \Sigma, \Delta, \delta, \lambda, q_0)$	$\lambda: Q \to \Delta$ (output on state)
Mealy	$(Q, \Sigma, \Delta, \delta, \lambda, q_0)$	$\lambda: Q \times \Sigma \to \Delta$ (output on transition)
PDA	$(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$	$\delta: Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \to 2^{(Q \times \Gamma^*)}$
DPDA	$(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$	δ: at most one move
TM	$(Q, \Sigma, \Gamma, \delta, q_0, b, F)$	$\delta: Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$
LBA	TM + ⊢, ⊣	Tape bounded by input length
Grammar	(V, T, P, S)	Productions $A \to \alpha$

Chomsky Hierarchy at a Glance

Type	Name	Production Rule	Language	Machine	Example
0	Unrestricted	$\alpha \to \beta$ (no restriction)	RE	Turing Machine	Halting problem
1	Context-Sensitive	$\alpha A \beta \to \alpha \delta \beta$, $\|LHS\| \le \|RHS\|$	CSL	LBA	$a^n b^n c^n$
2	Context-Free (CFG)	$A \to \alpha$	CFL	PDA	$a^n b^n$
3	Regular	$A \to a$ or $A \to aB$	RL	DFA / NFA	$a^n b$

Strict containment: RL ⊂ CFL ⊂ CSL ⊂ RE

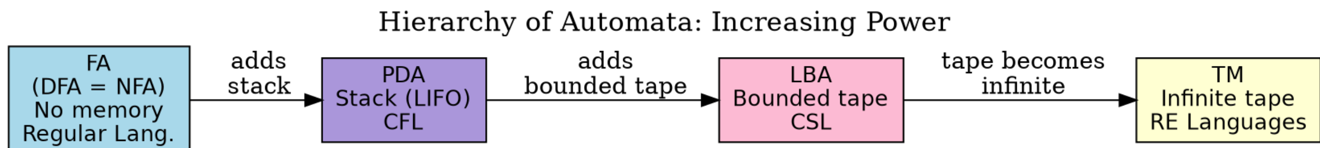


Figure 1: Hierarchy of Automata

Key Theorems (One-Liners)

- **Rabin-Scott:** NFA ≡ DFA (subset construction; n states → at most 2^n states).
- **Kleene's Theorem:** A language is regular ⟺ it is denoted by a regular expression ⟺ it is accepted by a finite automaton.
- **Arden's Theorem:** If P does NOT contain ε, then R = Q + RP has unique solution **R = QP***.
- **Pumping Lemma (Regular):** For any regular L, ∃ n such that ∀ z ∈ L with |z| ≥ n, ∃ split z = uvw with |uv| ≤ n, |v| ≥ 1, ∀ i ≥ 0: uv^iw ∈ L.
- **Church-Turing Thesis:** Anything intuitively algorithmic is computable by a Turing Machine.

Decidability Quick Reference

Decidable problems for CFL: Membership (CYK), Emptiness, Finiteness/Infiniteness. **Undecidable problems:** Equivalence of CFGs, Ambiguity of CFGs, Halting Problem, Post Correspondence Problem.

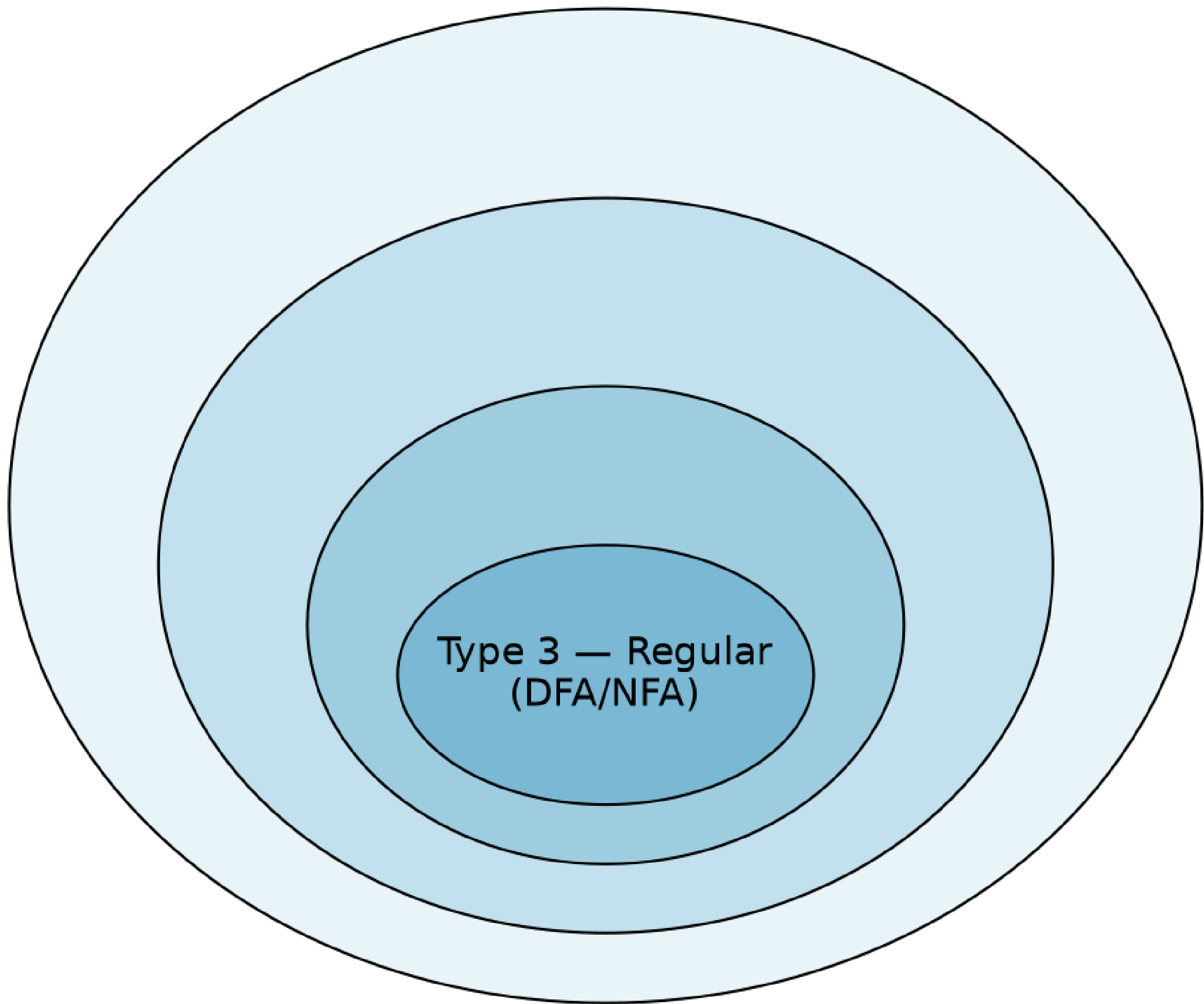


Figure 2: Chomsky Hierarchy — Nested Containment

Topic 1 — DFA and NFA

The Idea

Both DFA and NFA are simple state machines. A **DFA** is strictly deterministic: from any state, on any input symbol, there is *exactly one* next state. An **NFA** is non-deterministic: from a state on an input, there can be zero, one, or many next states, and ε-transitions (moves that consume no input) are allowed.

Despite this difference, **they have equal computational power.** Both accept exactly the class of regular languages.

DFA — Formal Definition

A DFA is a 5-tuple **$(Q, \Sigma, \delta, q_0, F)$**, where:

- **Q** = finite set of states
- **Σ** = input alphabet
- **δ** = transition function, $\delta : Q \times \Sigma \rightarrow Q$
- **q_0** $\in Q$ = initial state
- **F** $\subseteq Q$ = set of final / accepting states

A string w is **accepted** if, after reading all symbols of w starting from q_0, the DFA ends in a state in F.

NFA — Formal Definition

An NFA is a 5-tuple **$(Q, \Sigma, \delta, q_0, F)$**, where the only difference from DFA is:

- **$\delta : Q \times \Sigma \rightarrow 2^Q$** (transition function returns a set of possible next states)

NFAs may also have **ε-transitions** (transitions on the empty string).

A string is **accepted** if at least one of the possible computation paths ends in a final state.

DFA vs NFA — Comparison Table

Aspect	DFA	NFA
Transition function	$\delta: Q \times \Sigma \rightarrow Q$	$\delta: Q \times \Sigma \rightarrow 2^Q$
Next state	Exactly one	Zero, one, or many
ε-transitions	Not allowed	Allowed
Backtracking	Not needed	May be needed
Acceptance	One path ends in F	Some path ends in F
Number of states	Usually more (up to 2^n)	Usually fewer
Construction from RE	Difficult	Easy
Power	Same — both regular languages	Same — both regular languages

Critical exam line: *"By the Rabin-Scott subset construction theorem, every NFA can be converted to an equivalent DFA. Hence DFA and NFA have equal computational power; both accept exactly the class of regular languages."*

Diagrams

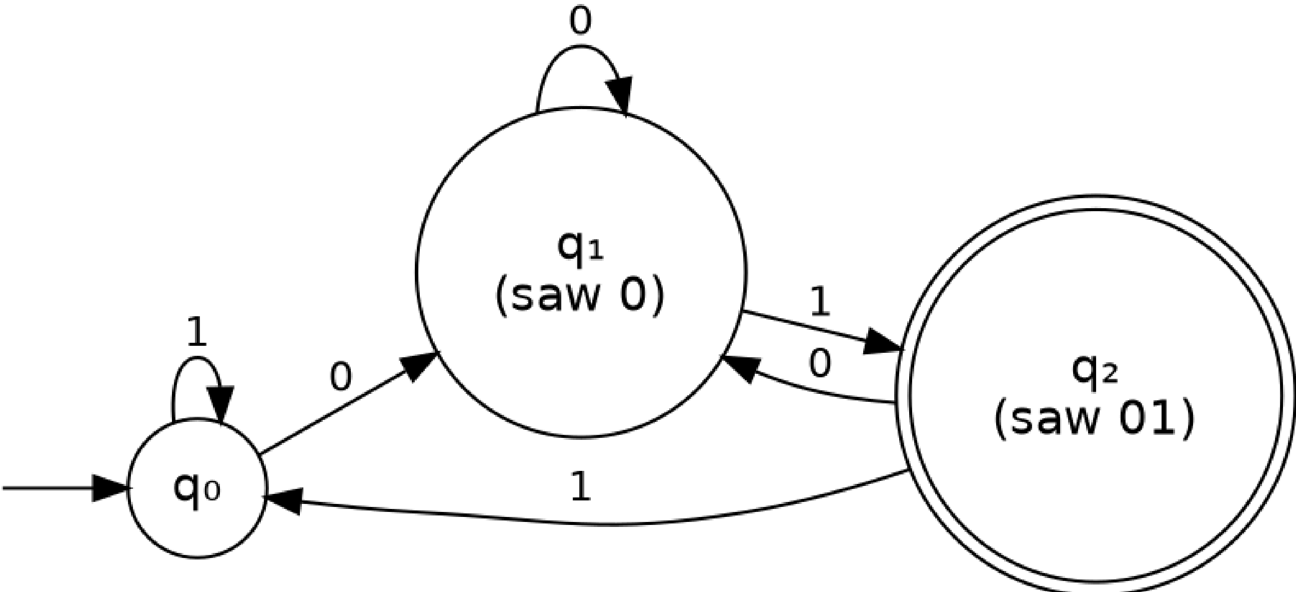


Figure 3: DFA accepting strings ending in “01”

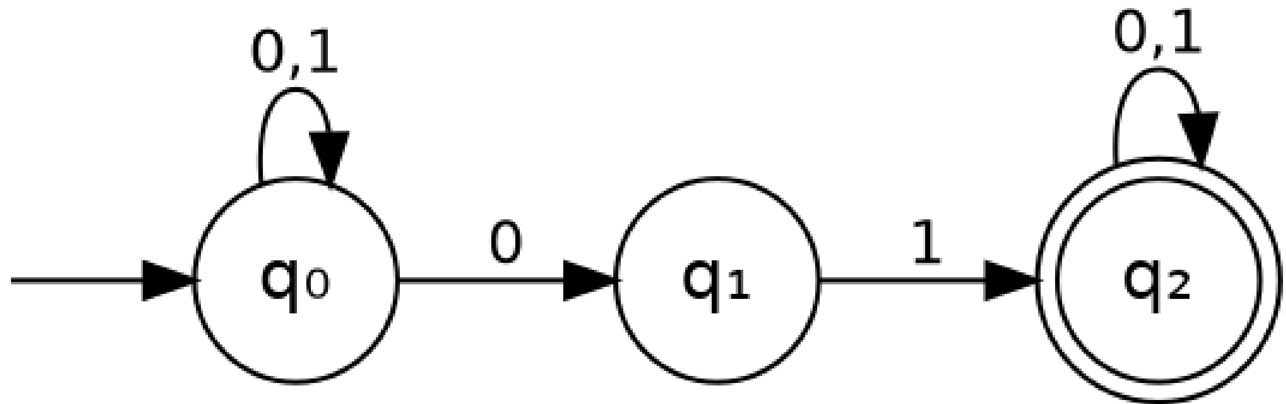


Figure 4: NFA accepting strings containing “01”

Topic 2 — Grammar, CFG, Derivation Trees, Ambiguity

What is a Grammar?

A grammar is a formal system that generates a language using rewriting rules. Formally, a grammar G = (V, T, P, S) where:

- **V** = finite set of non-terminal variables (e.g., S, A, B — placeholders)
- **T** = finite set of terminals (actual alphabet symbols)
- **P** = set of productions $A \to \alpha$, where $A \in V$ and $\alpha \in (V \cup T)^*$
- **S** $\in$ V = start symbol

To **generate** a string, start from S and repeatedly apply productions until only terminals remain.

CFG — Context-Free Grammar

A CFG is a grammar where every production has the form $\mathbf{A \to \alpha}$, with A being a single non-terminal and α any string of terminals and non-terminals.

The name “context-free” means: A can be replaced by α regardless of what symbols surround A. No context is needed.

Example: Grammar $S \to aSb \mid ab$ generates the language $\{a^n b^n \mid n \geq 1\}$.

Derivation of “aabb”: $S \Rightarrow aSb \Rightarrow aabb$ (replace inner S with ab).

Derivation Tree (Parse Tree)

A derivation tree visually represents how a grammar derives a string.

Rules:

- Root = start symbol S

- Internal nodes = non-terminals
- Leaf nodes = terminals (or ε)
- Reading the leaves left-to-right = the derived string

Derivation Tree for "aabb" using S → aSb | ab

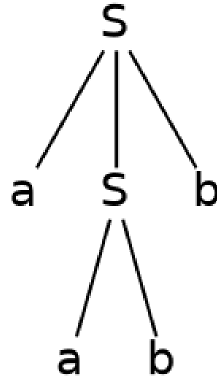


Figure 5: Derivation Tree for “aabb” using S → aSb | ab

Leftmost vs Rightmost Derivation

- **Leftmost Derivation (LMD):** Always replace the leftmost non-terminal at each step.
- **Rightmost Derivation (RMD):** Always replace the rightmost non-terminal at each step.

Both lead to the same final string but may produce different sequences of intermediate strings.

Ambiguity

A grammar is **ambiguous** if some string in the language has **two or more distinct parse trees** (equivalently, two distinct LMDs or two distinct RMDs).

Classic example: S → S + S | S * S | a generates “a + a * a” with two different parse trees, depending on whether + or * is applied first.

Inherently Ambiguous Languages: Some context-free languages have NO unambiguous grammar — every grammar that generates them is ambiguous. Example: $\{a^n b^n c^m \mid n, m \geq 1\} \cup \{a^n b^m c^m \mid n, m \geq 1\}$.

Important: Determining whether a CFG is ambiguous is **undecidable**.

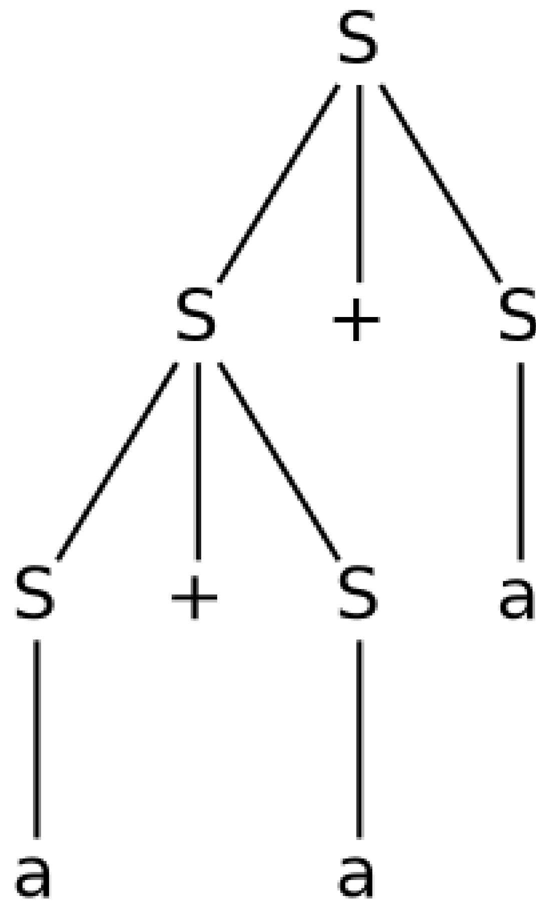


Figure 6: Parse Tree 1: ((a+a)+a)

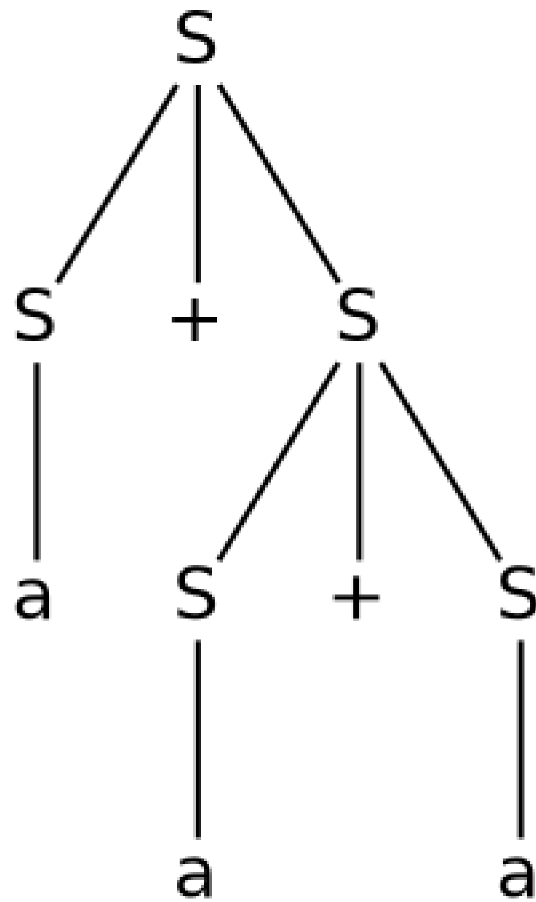


Figure 7: Parse Tree 2: (a+(a+a))

Topic 3 — Chomsky Hierarchy

The Idea

Noam Chomsky organized all formal grammars into 4 levels based on how restrictive their production rules are. The 4 types are nested: each is a proper subset of the level above.

`Type 0 (RE) ⊃ Type 1 (CSL) ⊃ Type 2 (CFL) ⊃ Type 3 (Regular)`

As you go up the hierarchy:

- Productions become less restrictive
- Languages become richer
- The accepting machine becomes more powerful

The Four Types — Complete Description

Type 0 — Unrestricted Grammar

- **Production form:** $\alpha \to \beta$ with no restriction (α must contain at least one non-terminal)
- **Language class:** Recursively Enumerable (RE)
- **Machine:** Turing Machine
- **Power:** Most powerful — can describe anything algorithmically computable

Type 1 — Context-Sensitive Grammar (CSG)

- **Production form:** $\alpha A\beta \to \alpha\delta\beta$, where A is a non-terminal and δ is non-empty
- **Length condition:** $|LHS| \le |RHS|$ — the grammar is non-contracting (productions never shrink the string)
- **Language class:** Context-Sensitive (CSL)
- **Machine:** Linear Bounded Automaton (LBA)
- **Example language:** $\{a^n b^n c^n \mid n \ge 1\}$

Type 2 — Context-Free Grammar (CFG)

- **Production form:** $A \to \alpha$ (single non-terminal on LHS, any string on RHS)
- **Language class:** Context-Free (CFL)
- **Machine:** Pushdown Automaton (PDA)
- **Example language:** $\{a^n b^n \mid n \ge 1\}$ via $S \to aSb \mid ab$

Type 3 — Regular Grammar

- **Production form:**
  - Right-linear: $A \to a$ or $A \to aB$
  - Left-linear: $A \to a$ or $A \to Ba$
- **Restriction:** Cannot mix right-linear and left-linear productions in the same grammar
- **Language class:** Regular Language (RL)
- **Machine:** Finite Automaton (DFA / NFA)
- **Example language:** $\{a^n b \mid n \ge 0\}$ via $S \to aS \mid b$

Summary Table — Memorize This

Type	Name	Production	Language	Machine	Example
0	Unrestricted	$\alpha \to \beta$	RE	TM	Halting problem
1	CSG	$\alpha A\beta \to \alpha\delta\beta$, $\lvert LHS\rvert \leq \lvert RHS\rvert$	CSL	LBA	$a^n b^n c^n$
2	CFG	$A \to \alpha$	CFL	PDA	$a^n b^n$
3	Regular	$A \to a$, $A \to aB$	RL	DFA/NFA	$a^n b$

Visual

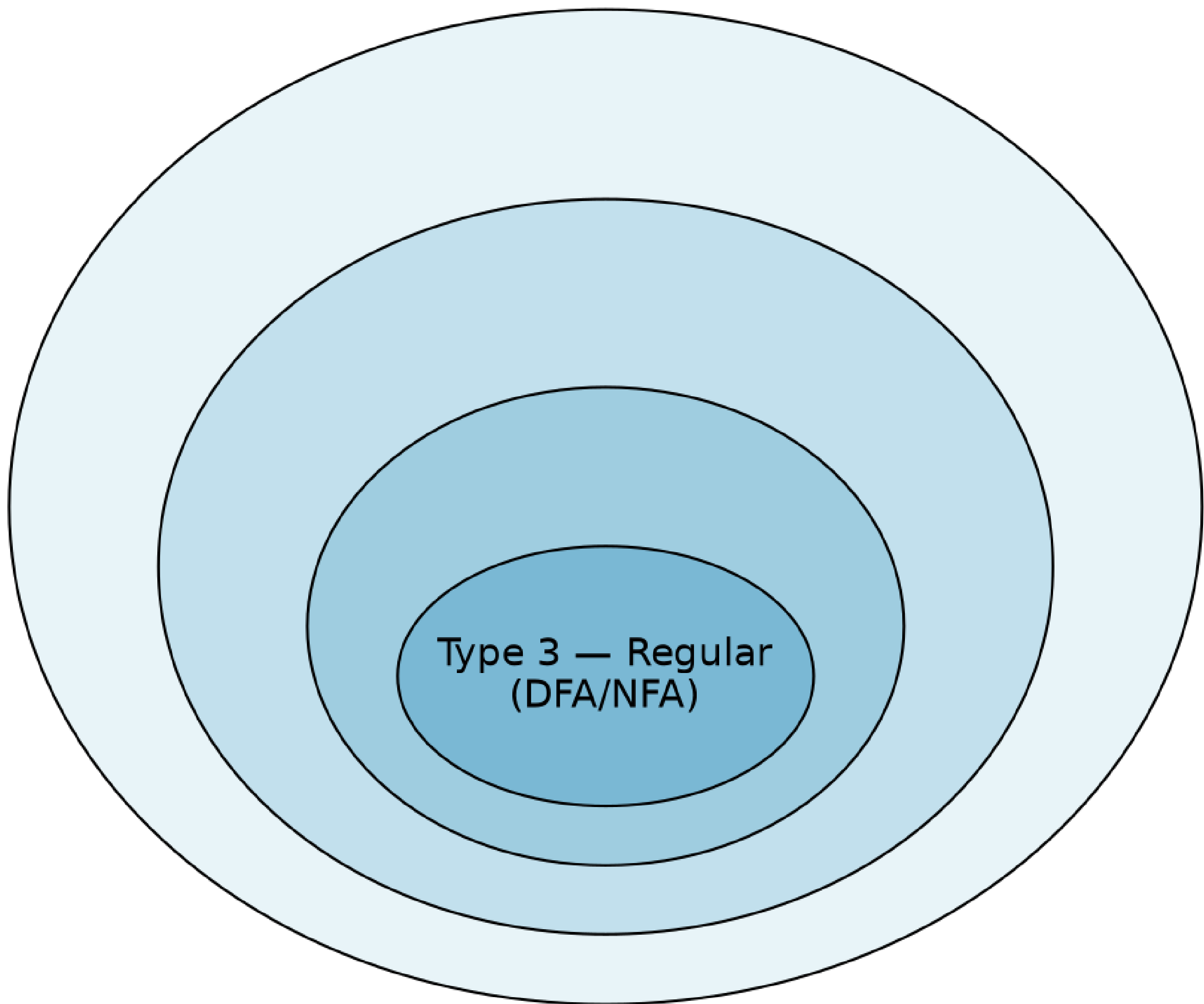


Figure 8: Chomsky Hierarchy showing nested containment

Topic 4 — Mealy and Moore Machines

The Idea

A regular DFA only says yes/no (accept/reject). **Mealy and Moore machines extend a DFA to produce output** as it processes input. They model real systems like vending machines, traffic lights, and digital circuits.

The **only difference** between them is where the output comes from:

- **Moore machine:** Output depends on the **current state alone** — output is associated with states.
- **Mealy machine:** Output depends on the **current state AND current input** — output is associated with transitions.

Moore Machine — Formal Definition

Moore machine = $(Q, \Sigma, \Delta, \delta, \lambda, q_0)$:

- Q = states; Σ = input alphabet; Δ = output alphabet
- $\delta : Q \times \Sigma \to Q$ (transition function — same as DFA)
- $\boldsymbol{\lambda : Q \to \Delta}$ (output function — depends only on state)
- q_0 = initial state

Output is written inside the state circle in diagrams.

Mealy Machine — Formal Definition

Mealy machine = $(Q, \Sigma, \Delta, \delta, \lambda, q_0)$:

- Same as Moore, except:
- $\boldsymbol{\lambda : Q \times \Sigma \to \Delta}$ (output function — depends on state AND input)

Output is written on the transition arrow in diagrams.

Mealy vs Moore — Comparison

Aspect	Mealy	Moore
Output function	$\lambda: Q \times \Sigma \to \Delta$	$\lambda: Q \to \Delta$
Output depends on	State + Input	State only
Output appears on	Transition (edge)	State (node)
# of states	Usually fewer	Usually more
Output for empty string	Not defined	Defined (output of q_0)
Power	Equal — interchangeable	Equal — interchangeable

Mealy → Moore Conversion (4-Step Procedure)

This is the procedure to memorize cold for the 14-mark Section C question.

1. **For each state in the Mealy machine, list all distinct outputs on incoming transitions.**

2. **If a state has exactly one distinct incoming output**, keep it as one Moore state and assign that output to it.
3. **If a state has multiple distinct incoming outputs** (e.g., z_1 and z_2), **split the state into copies** — one copy per output (e.g., q_2 becomes q_2/z_1 and q_2/z_2).
4. **Redirect each transition** to the copy whose output matches the output that was on that Mealy edge. Remove the output labels from the edges (they now live on states).

Note: Outgoing transitions from the split copies are identical to the original state's outgoing transitions.

Moore → Mealy Conversion (Easier — 1 Step)

For every transition q_i → q_j in Moore, take the output of the destination state q_j and place it as the output on that transition arrow. Then remove the outputs from the states. Done.

Diagrams — Worked Example: Counting Substring "ab"

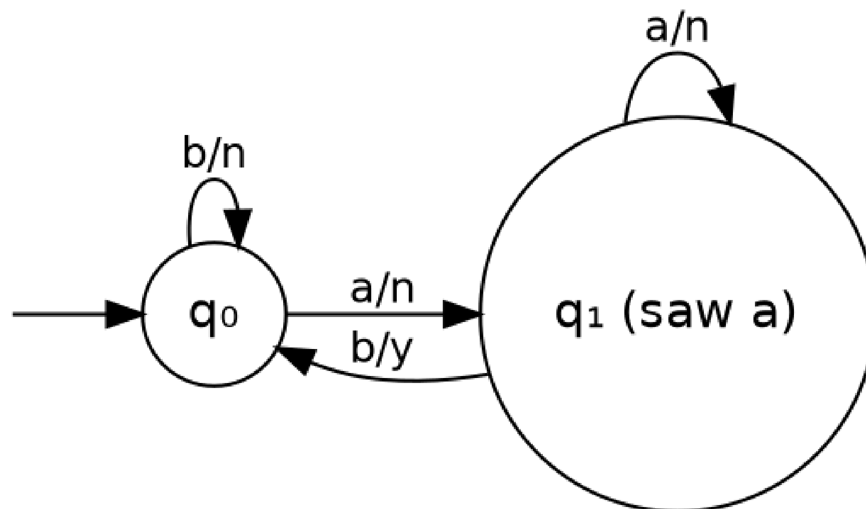


Figure 9: Mealy Machine: outputs y when "ab" detected

Notice the Moore version has **3 states**, while the Mealy version has only **2 states** — Moore typically requires more states.

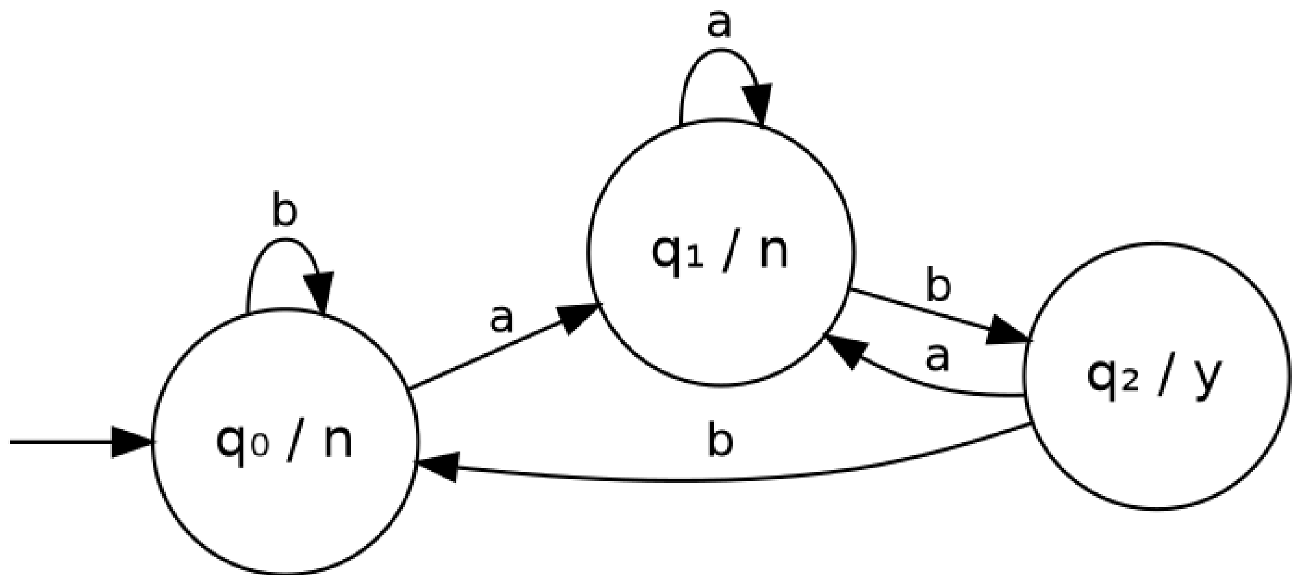


Figure 10: Moore Machine: outputs y in the state representing “just saw ab”

Topic 5 — Turing Machine

The Idea

The Turing Machine (TM) is the most powerful theoretical computer. It is essentially a finite control unit (like a DFA) attached to an **infinite two-way tape** with a read/write head. Anything algorithmically computable can be done by a TM (Church-Turing thesis).

Components of a TM

- **Infinite Tape:** Divided into cells, each holding one tape symbol. Extends infinitely in both directions.
- **Read/Write Head:** Reads the symbol in the current cell, can write a new symbol, and moves either Left or Right by one cell.
- **Finite Control:** A finite set of states that drives the machine’s behavior.

Formal Definition (7-tuple — memorize this exactly)

A Turing Machine $M = (Q, \Sigma, \Gamma, \delta, q_0, b, F)$, where:

- **Q** = finite set of states
- **Σ** = input alphabet
- **Γ** = tape alphabet ($\Sigma \subseteq \Gamma$; includes Σ plus extra symbols TM can write)
- **δ** = transition function: $\boldsymbol{\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}}$
- $\mathbf{q_0} \in Q$ = initial state
- $\mathbf{b} \in \Gamma$, $b \notin \Sigma$ = blank symbol
- $\mathbf{F} \subseteq Q$ = set of final / accepting states

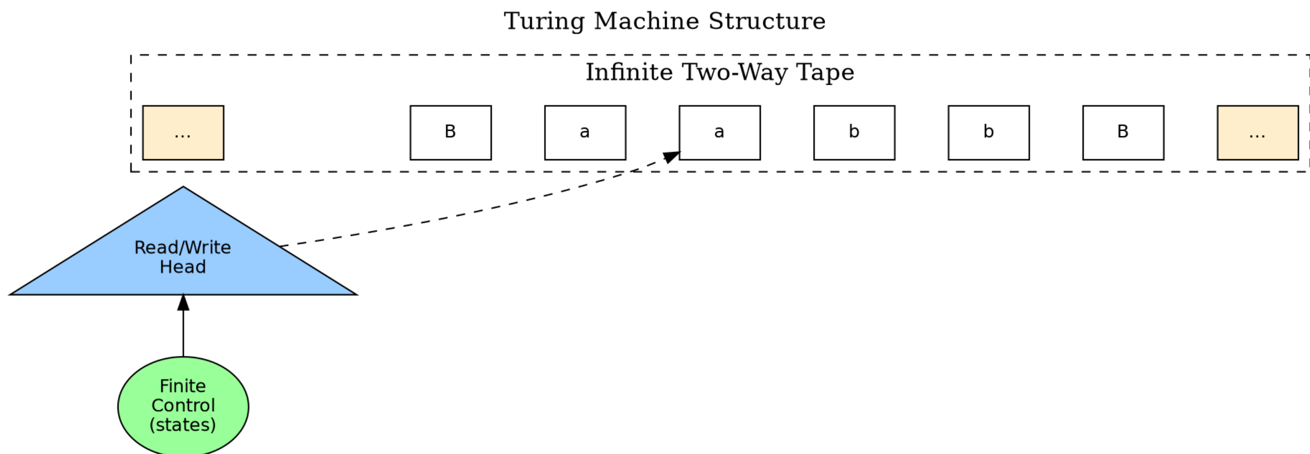


Figure 11: Turing Machine Structure

Decoding the transition: δ(q_i, X) = (q_j, Y, R) means: "in state q_i reading symbol X, move to state q_j, write Y in place of X, and shift the head one cell to the Right."

Three Outcomes for Any Input

When a TM is run on an input, exactly one of three things happens:

1. **Final halt** → reaches a state in F → **ACCEPT**
2. **Non-final halt** → halts in a state not in F (no transition defined) → **REJECT**
3. **Infinite loop** → the TM never halts

The third possibility (infinite loop) is precisely what makes the **Halting Problem** undecidable.

Modifications of TM

Modification	Description	Power Change?
Multi-tape TM	Multiple tapes with their own heads	None
Multi-head TM	One tape, multiple heads	None
Multi-dimensional TM	2D / 3D tape grid	None
Non-deterministic TM	Multiple choices per (state, symbol)	None
TM with stay option	δ ends with {L, R, S}	None
Semi-infinite tape	Tape extends in one direction only	None

Critical exam line: *"All modifications of the Turing Machine have the same computational power as the standard Turing Machine. They differ only in efficiency or convenience, not in the class of languages they accept."*

Universal Turing Machine (UTM)

A **Universal Turing Machine** is a Turing Machine that simulates any other Turing Machine on any input, given an encoded description of the target machine and the input on its tape.

Tape input format: ⟨encoded description of TM M⟩ # ⟨input string w⟩

Significance: The UTM is the theoretical foundation of the stored-program computer — one machine that can run any program. This concept, proved by Turing in 1936, is the birth of computer science as we know it.

Topic 6 — TM Design for L = $\{a^n b^n \mid n \geq 1\}$

This is the 2024 Section C Q10 question.

Strategy (Plain English)

Match each a with one b. To do this:

1. Scan right, find the leftmost a, replace it with X.
2. Continue right past more a's, find the leftmost b, replace it with Y.
3. Scan all the way back left until we hit X.
4. Move right and repeat.
5. When all a's become X and all b's become Y (head sees Y from the start state), enter verification mode.
6. Skip past Y's, hit the blank symbol → ACCEPT.

States

- **q_0** — find next unmarked a
- **q_1** — found a, scanning right to find a b
- **q_2** — marked b as Y, scanning back left
- **q_3** — verification mode (only Y's should remain)
- **q_4** — final / accept state

Tape Alphabet

Γ = {a, b, X, Y, B} where X marks a matched a, Y marks a matched b, B is blank.

Transition Table

State	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**	—	—	—	—	—

State Diagram

Trace for Input "aabb"

Step	Tape (head position underlined)	State
0	**a**abb	q_0
1	X**a**bb	q_1
2	Xa**b**b	q_1
3	X**a**Yb	q_2
4	**X**aYb	q_2

Step	Tape (head position underlined)	State
5	X**a**Yb	q_0
6	XX**Y**b	q_1
7	XXY**b**	q_1
8	XX**Y**Y	q_2
9	X**X**YY	q_2
10	XX**Y**Y	q_0
11	XXY**Y**	q_3
12	XXYY**B**	q_3
13	(head past)	q_4 → **ACCEPT**

Closing Statement to Write in Exam

"The Turing Machine $M = (Q, \{a, b\}, \{a, b, X, Y, B\}, \delta, q_0, B, \{q_4\})$ constructed above accepts exactly the language $L = \{a^n b^n \mid n \geq 1\}$ by repeatedly matching the leftmost unmarked a with the leftmost unmarked b, replacing them with X and Y respectively. The machine reaches the final state q_4 if and only if the input has equal numbers of a's followed by equal b's."

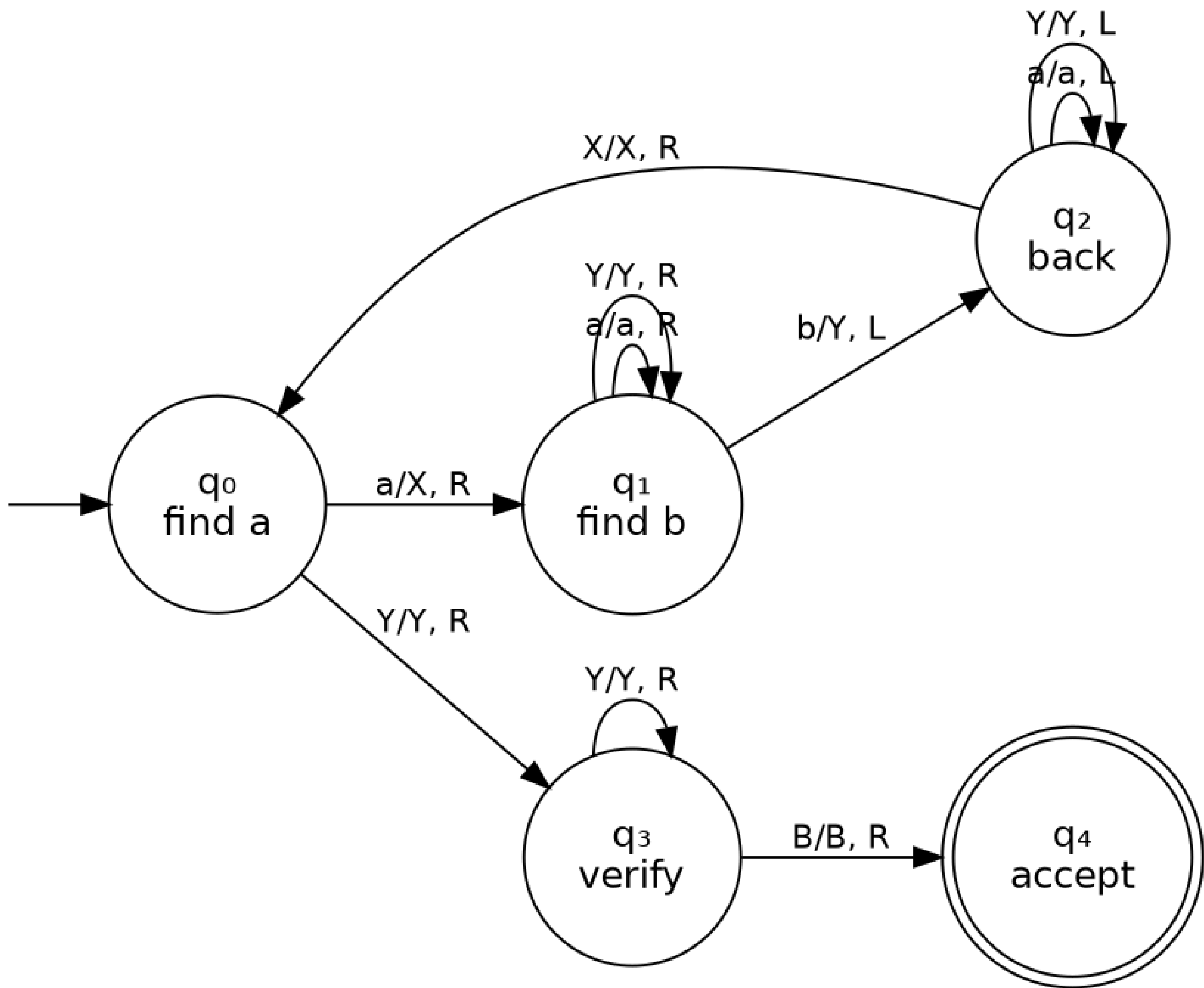


Figure 12: Turing Machine for $L = \{a^n b^n \mid n \geq 1\}$

Topic 7 — TM Design for $L = \{a^n b^n c^n \mid n \geq 1\}$

This is the 2025 Section C Q12 question.

Why This Language Needs a TM

The language $\{a^n b^n c^n \mid n \geq 1\}$ is **NOT context-free**. A PDA (with one stack) cannot accept it — once the stack is used to count a's against b's, there's no memory left to count c's. Only a TM (with its tape memory) can do this.

Strategy

Each pass through the input:

1. Mark one a as X
2. Continue right, mark the next b as Y
3. Continue right further, mark the next c as Z
4. Scan all the way back left
5. Repeat until no a's, b's, or c's remain
6. Verify all symbols are X/Y/Z and the input ends — ACCEPT

States

- **q_0** — find next unmarked a
- **q_1** — found a, scanning right for b
- **q_2** — found b, scanning right for c
- **q_3** — found c, scanning back left
- **q_4** — verification mode (only Y, Z should remain)
- **q_5** — final / accept state

Transition Table

State	a	b	c	X	Y	Z	B
q_0	(q_1, X, R)	—	—	—	(q_4, Y, R)	—	—
q_1	(q_1, a, R)	(q_2, Y, R)	—	—	(q_1, Y, R)	—	—
q_2	—	(q_2, b, R)	(q_3, Z, L)	—	—	(q_2, Z, R)	—
q_3	(q_3, a, L)	(q_3, b, L)	—	(q_0, X, R)	(q_3, Y, L)	(q_3, Z, L)	—
q_4	—	—	—	—	(q_4, Y, R)	(q_4, Z, R)	(q_5, B, R)
***q_5**	—	—	—	—	—	—	—

State Diagram

Trace Sketch for "aabbcc"

- Pass 1: mark first a → X, first b → Y, first c → Z; tape becomes XaYbZc; scan left back to start.
- Pass 2: mark remaining a → X, remaining b → Y, remaining c → Z; tape becomes XXYYZZ; scan left.

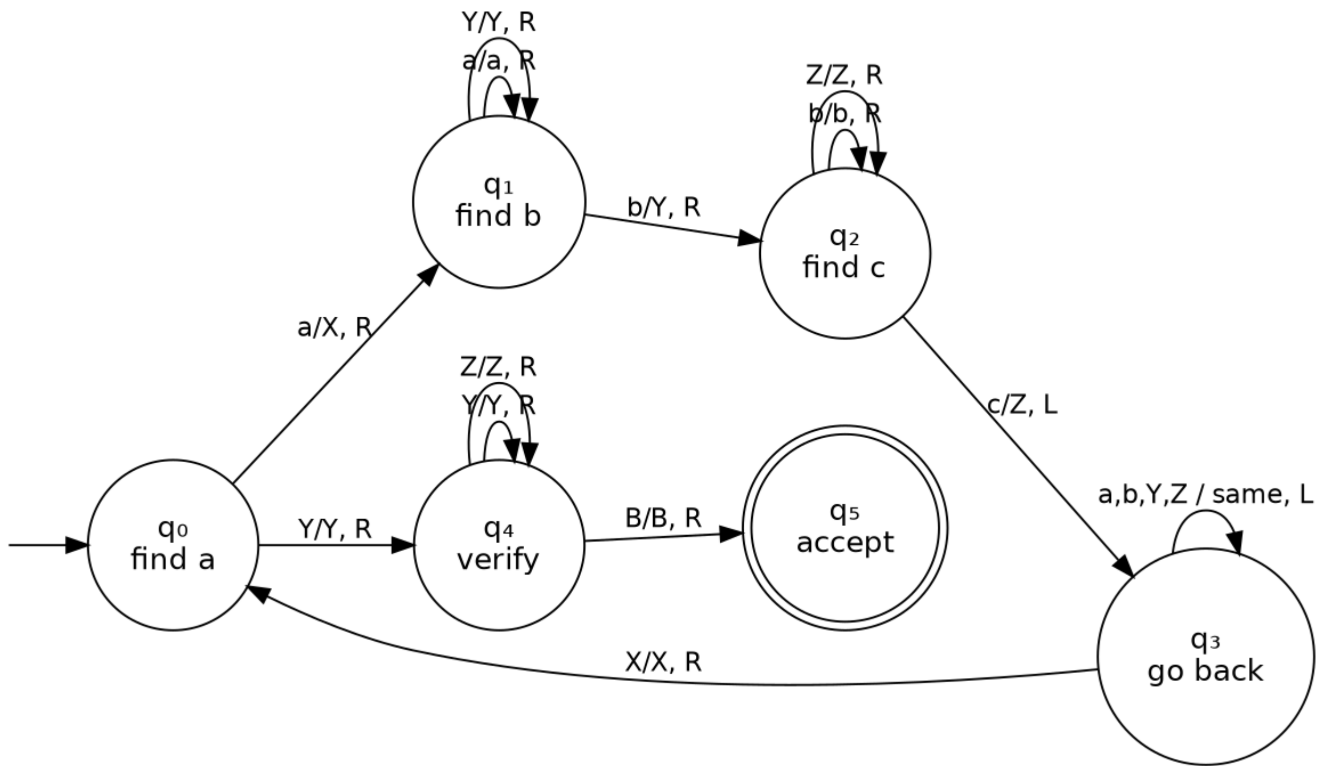


Figure 13: Turing Machine for $L = \{a^n b^n c^n \mid n \geq 1\}$

- Verification: from q_0 read Y → enter q_4; skip Y, Y, Z, Z; read blank B → ACCEPT in q_5.

Closing Statement

"This Turing Machine accepts exactly the language $\{a^n b^n c^n \mid n \geq 1\}$. The language is not context-free (provable by Pumping Lemma for CFLs), establishing that the Turing Machine is strictly more powerful than the PDA."

Topic 8 — CFG Conversion to CNF

Chomsky Normal Form Definition

A CFG is in **Chomsky Normal Form (CNF)** if every production is of one of these forms:

- **A → BC** (exactly two non-terminals on RHS), OR
- **A → a** (exactly one terminal on RHS)

(Optionally **S → ε** if ε is in the language.)

Property: If a CFG is in CNF, deriving a string of length n requires exactly **2n − 1** productions.

Conversion Algorithm — 4 Steps

1. **Eliminate ε-productions.** Find all nullable variables. For each production with a nullable variable, add a copy without it. Remove A → ε (except S → ε if needed).
2. **Eliminate unit productions** (A → B where B is a single non-terminal). If A → B and B → γ, add A → γ.
3. **Replace terminals in long productions.** For any production with 2+ symbols on RHS where a terminal appears, introduce new variable T_a → a and replace each terminal with its variable.
4. **Binarize long productions.** For productions $A \to X_1X_2X_3 \ldots X_n$ with $n \geq 3$, replace with $A \to X_1Y_1$, $Y_1 \to X_2Y_2$, …, introducing new intermediate variables.

PYQ Worked Example — 2025 Q6

Convert S → aS | Sb | ab into CNF.

Step 1 — Eliminate ε-productions: None present. Skip.

Step 2 — Eliminate unit productions: None present (no A → B form). Skip.

Step 3 — Replace terminals in multi-symbol productions:

- aS contains terminal a. Introduce T_a → a.
- Sb contains terminal b. Introduce T_b → b.
- ab contains terminals a and b. Replace both.

After replacement:

- S → T_a S
- S → S T_b
- S → T_a T_b
- T_a → a
- T_b → b

Step 4 — Binarize: All productions now have at most 2 non-terminals on RHS. No binarization needed.

Final CNF Grammar:

$$S \to T_a S \mid S T_b \mid T_a T_b$$
$$T_a \to a$$
$$T_b \to b$$

Verification: All productions are of the form A → BC or A → a. ✓ The grammar is in CNF.

Topic 9 — Greibach Normal Form (GNF)

Definition

A CFG is in **Greibach Normal Form** if every production is of the form:

A → aα

where a is a single terminal and α is a (possibly empty) string of non-terminals.

In words: the **first symbol on the RHS must always be a terminal**, followed by any number of non-terminals (or none).

Property

If a CFG is in GNF, deriving a string of length n requires exactly **n productions**.

CNF vs GNF

Aspect	CNF	GNF
Production form	A → BC or A → a	A → aα
First symbol of RHS	Non-terminal or single terminal	Always a terminal
Derivation length for |w| = n	2n − 1	n
Used in	CYK parsing algorithm	PDA construction

Example

Given CFG: S → AB, A → a, B → b. This is already in CNF but NOT in GNF (S → AB starts with non-terminal A).

Convert to GNF:

S → AB → aB (substitute A by a, since A → a).

GNF version:

- S → aB
- A → a
- B → b

Now every production starts with a terminal. ✓

Topic 10 — DFA Design for Binary Numbers Divisible by 5

This is 2025 Section C Q9.

The Insight

When you read a binary number left to right, each new bit doubles the current value and adds 0 or 1. So the only thing we need to remember is the current value modulo 5 — that's just 5 possible remainders.

Update rule: If current remainder is r, after reading bit b, the new remainder is:

$$\text{new_rem} = (2r + b) \text{ mod } 5$$

States and Construction

States = remainders mod 5:

- **q_0** — remainder 0 (initial AND final, because empty input = value 0, which is divisible by 5)
- **q_1** — remainder 1
- **q_2** — remainder 2
- **q_3** — remainder 3
- **q_4** — remainder 4

Transition Table

Current state (rem)	On 0 → (2r) mod 5	On 1 → (2r+1) mod 5
*→ **q_0** (0)	q_0	q_1
q_1 (1)	q_2	q_3
q_2 (2)	q_4	q_0
q_3 (3)	q_1	q_2
q_4 (4)	q_3	q_4

State Diagram

Verification

- Input "101" = decimal 5 (divisible by 5). Path: $q_0 \to q_1 \to q_2 \to q_0$ ✓ (ends in final state, accepted)
- Input "1010" = decimal 10 (divisible by 5). Path: $q_0 \to q_1 \to q_2 \to q_0 \to q_0$ ✓ (accepted)
- Input "111" = decimal 7 (not divisible). Path: $q_0 \to q_1 \to q_3 \to q_2$ ✗ (not in F, rejected)

Formal Specification

$$M = (\{q_0, q_1, q_2, q_3, q_4\}, \{0, 1\}, \delta, q_0, \{q_0\})$$

with δ given by the transition table above.

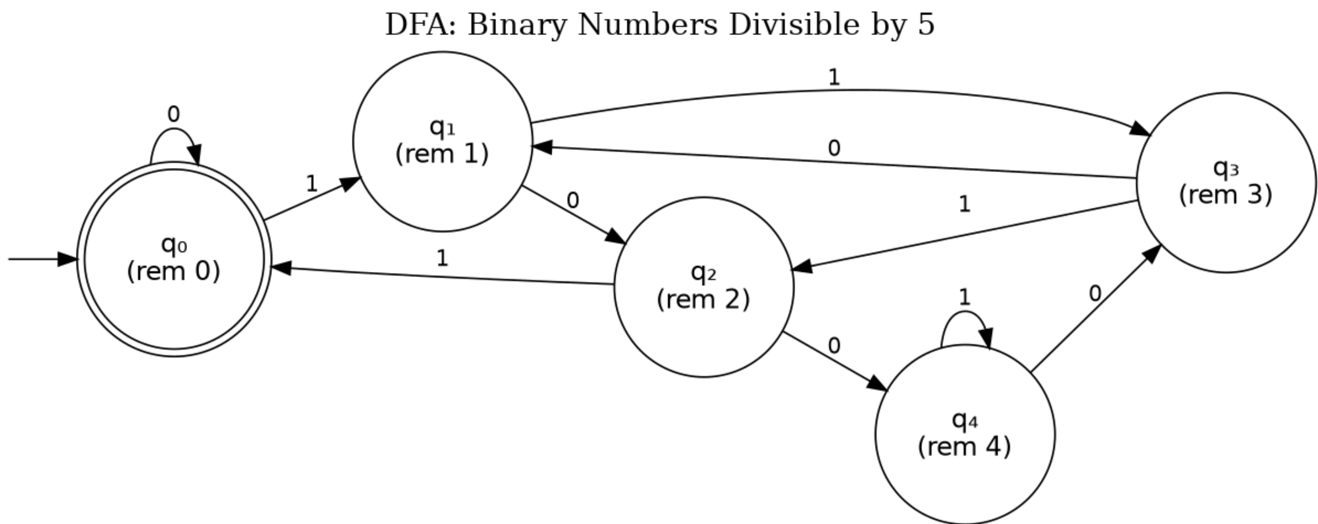


Figure 14: DFA for Binary Numbers Divisible by 5

Topic 11 — Regular Expressions and Identities

Operators

- **Concatenation:** ab — a followed by b
- **Union (+):** a + b — a or b
- **Kleene star (*):** a* — zero or more a's = {ε, a, aa, aaa, ...}
- **Positive closure ($^+$):** a^+ — one or more a's = {a, aa, aaa, ...}

Regular Expression for "Strings Over {0,1} Containing At Least One 1"

This is 2025 Section A Q3.

Regular Expression: (0+1)*1(0+1)* or equivalently **0*1(0+1)***

Explanation: Both forms guarantee at least one explicit 1 in the string. The first matches anything before, then a 1, then anything after. The second is tighter: it matches some leading 0's, then the first 1, then anything.

Identity to Prove — 2024 Section A Q5

Prove (a+b)* = a*(ba*)*

(a + b)* generates all strings over {a, b} including ε.

a*(ba*)* also generates all strings over {a, b}: think of it as zero or more a's, followed by zero or more "ba*" blocks. Every string can be decomposed this way: leading a's, then for each subsequent b, the a's that follow.

Proof:

- (Subset ⊆) Any string in (a + b)* consists of a's and b's in some order. Write it as: leading a-block, then each b followed by its a-block. This matches a*(ba*)*.
- (Superset ⊇) Any string in a*(ba*)* contains only a's and b's, so it is in (a + b)*.

Therefore **(a + b)* = a*(ba*)***. ∎

Topic 12 — Arden's Theorem

Statement

If P does not contain ε, then the equation **R = Q + RP** has the unique solution:

$$R = QP^*$$

Use

Arden's theorem is the algebraic method to **convert a DFA / NFA into a Regular Expression**. We write one equation per state (based on incoming transitions) and solve the system.

Procedure

1. For each state q_i, write an equation: q_i = (sum over transitions arriving at q_i of source_state · symbol). For the initial state, also add ε.
2. The expression for the final state(s) gives the regular expression.

Worked Example

Consider this DFA:

- δ(A, a) = A (self-loop)
- δ(A, b) = B
- δ(B, a) = B (self-loop)
- δ(B, b) = B (self-loop)
- A is initial, B is final

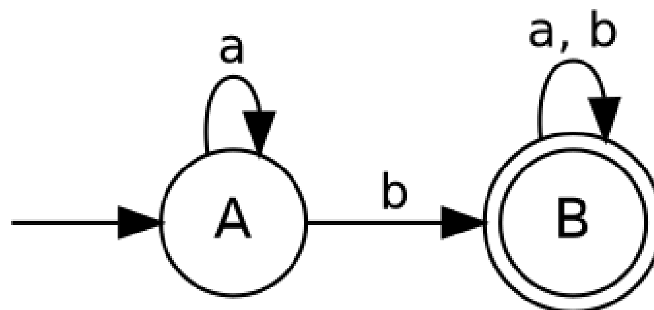


Figure 15: DFA for Arden's worked example

Step 1 — Equations:

- A = ε + A·a (start, plus self-loop on a)
- B = A·b + B·a + B·b = A·b + B(a + b)

Step 2 — Solve A:

$A = \varepsilon + Aa$ is of the form $R = Q + RP$ with $Q = \varepsilon$, $P = a$. By Arden's: $A = \varepsilon \cdot a^* =$ **a^***.

Step 3 — Solve B:

Substitute A: $B = a^*b + B(a + b)$. This is $R = Q + RP$ with $Q = a^*b$, $P = (a+b)$. By Arden's: $B = a^*b(a + b)^*$.

Final Regular Expression:

$$\boxed{a^*b(a+b)^*}$$

This represents all strings over {a, b} containing at least one b.

Topic 13 — Pumping Lemma for Regular Languages

What It's For

The Pumping Lemma is used to **prove that a language is NOT regular**. It cannot prove a language IS regular — only the converse direction is useful.

Statement

For any regular language L, there exists an integer **n** (called the pumping length) such that for every string $z \in L$ with $|z| \geq n$, z can be split as **z = uvw** where:

1. $|uv| \leq n$
2. $|v| \geq 1$
3. **For all** $i \geq 0$, $uv^iw \in L$

In words: any sufficiently long string in L has a non-empty middle section v that can be repeated (or removed) any number of times, and the result still belongs to L.

Proof Template (How to Use It)

To prove L is NOT regular, use **proof by contradiction**:

1. Assume L is regular.
2. Then the pumping lemma applies; let n be the pumping length.
3. **Choose** a specific string $z \in L$ with $|z| \geq n$ (carefully).
4. For ANY split $z = uvw$ satisfying $|uv| \leq n$ and $|v| \geq 1$, show that for some i, $uv^iw \notin L$.
5. This contradicts the pumping lemma → L is not regular.

Worked Example — Prove $L = \{a^nb^n \mid n \geq 1\}$ is Not Regular

1. **Assume** L is regular. Let n be the pumping length.
2. **Choose** $z = a^nb^n$. Clearly $z \in L$ and $|z| = 2n \geq n$.
3. **Split** $z = uvw$ with $|uv| \leq n$ and $|v| \geq 1$.
4. Since $|uv| \leq n$ and z's first n characters are all a's, both u and v consist only of a's. So $v = a^k$ for some $k \geq 1$.
5. **Pump** with $i = 2$: $uv^2w = a^{n+k}b^n$. This has $n + k$ a's but only n b's, and $n + k \neq n$ (since $k \geq 1$).
6. So $uv^2w \notin L$. **Contradiction!**

Therefore L is not regular. ∎

Another Example — $L = \{a^p \mid p \text{ is prime}\}$

Choose $z = a^p$ for some prime $p \geq n + 2$. Split $z = uvw$ with $v = a^k$, $k \geq 1$.

Pump with $i = p$: $|uv^pw| = (p - k) + pk = p + (p - 1)k$. For $k \geq 1$ and $p \geq 3$, $p + (p - 1)k$ is composite. So $uv^pw \notin L$. Contradiction. L is not regular. ∎

Topic 14 — Kleene's Theorem

Statement

Kleene's Theorem (two parts):

1. For every regular expression R, there exists a finite automaton (DFA or NFA) that accepts the language denoted by R.
2. For every finite automaton M, there exists a regular expression R that denotes the language accepted by M.

In short: **Regular expressions and finite automata are equivalent in expressive power**.

Implications

A language L is **regular** if and only if at least one of the following holds:

- L is denoted by a regular expression
- L is accepted by some DFA
- L is accepted by some NFA
- L is generated by a regular grammar

These four statements are all equivalent — they pick out exactly the same class of languages.

Tools for Each Direction

- **RE → FA:** Thompson's construction algorithm builds an ε-NFA recursively from the structure of the RE (base cases for individual symbols, recursive cases for union, concatenation, and Kleene star).
- **FA → RE:** Arden's theorem provides an algebraic method by writing one equation per state and solving the system.

ε-Transitions Bridge the Two

ε-transitions in NFAs are the "glue" that makes Thompson's construction simple. They allow connecting machine fragments together without consuming input.

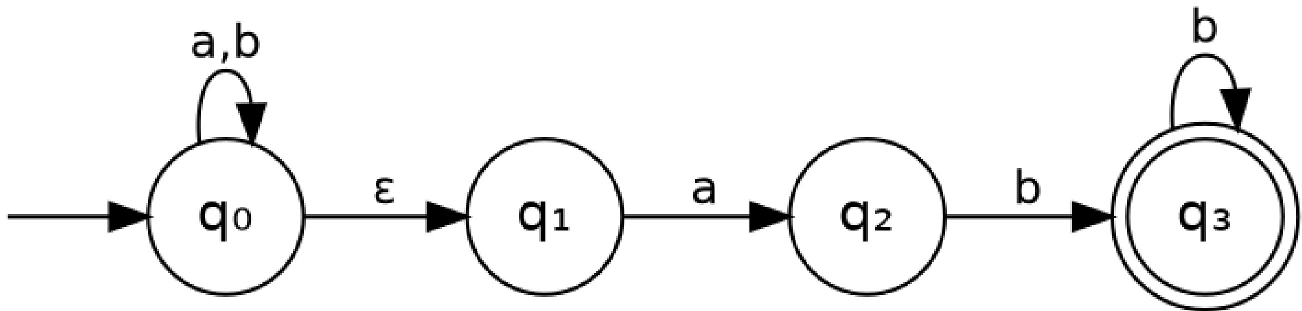


Figure 16: ε-NFA Example

Topic 15 — Push Down Automata (PDA)

The Idea

A PDA = DFA + a stack. The stack provides Last-In-First-Out (LIFO) memory, which lets the PDA recognize languages that finite automata cannot — specifically, the **context-free languages**.

Formal Definition (7-tuple)

A PDA = $(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$:

- **Q** = states
- **Σ** = input alphabet
- **Γ** = stack alphabet
- $\boldsymbol{\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \rightarrow 2^{(Q \times \Gamma^*)}}$ = transition function
- $\boldsymbol{q_0}$ = initial state
- $\boldsymbol{Z_0} \in \Gamma$ = initial stack symbol
- **F** $\subseteq Q$ = set of final states

Stack Operations (Three Possibilities)

For a transition $\delta(q_i, a, Z) = (q_j, \beta)$:

- **PUSH:** β is longer than Z (e.g., AZ pushes A onto the stack)
- **POP:** β = ε (the stack top is removed)
- **SKIP:** β = Z (stack unchanged)

Visual

DPDA vs NPDA

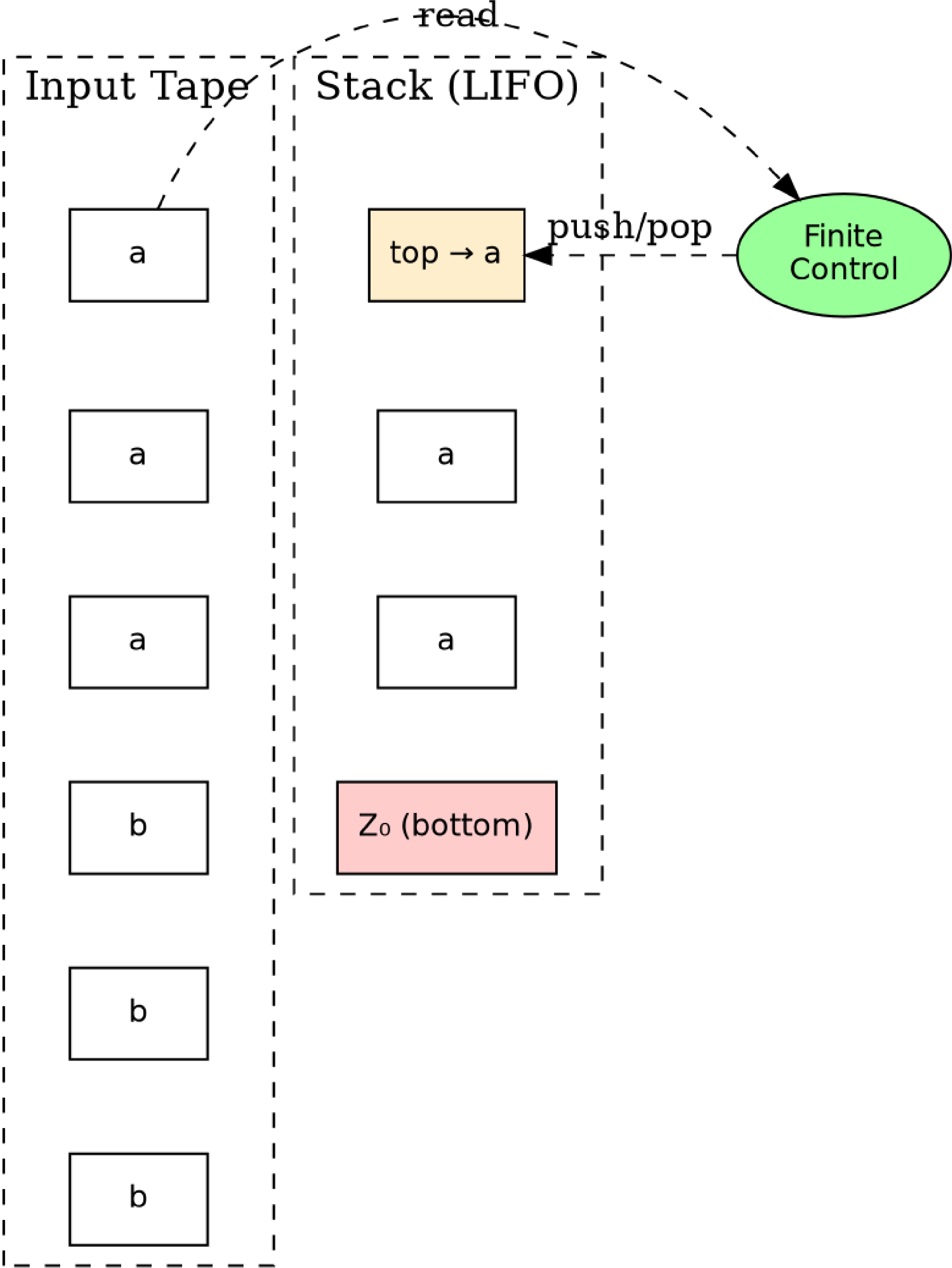


Figure 17: PDA Structure: Input Tape + Stack + Finite Control

Aspect	DPDA (Deterministic)	NPDA (Non-deterministic)
Transition function	At most ONE move per (state, input, stack)	Set of possible moves
Languages accepted	DCFLs (Deterministic CFLs)	All CFLs
Power	Strictly less than NPDA	More powerful than DPDA
Example unique to NPDA	Palindromes $\{ww^R\}$	(cannot be done by DPDA)

Important: NPDA is strictly more powerful than DPDA. DCFLs are a proper subset of CFLs.

Two-Stack PDA — TM Equivalent

A PDA equipped with **two independent stacks** is as powerful as a Turing Machine. Two stacks together can simulate the TM's tape: contents to the left of the head go on stack 1, contents to the right go on stack 2.

Result: A two-stack PDA accepts exactly the recursively enumerable languages — i.e., the same as a TM.

PDA for $L = \{a^n b^n \mid n \geq 1\}$

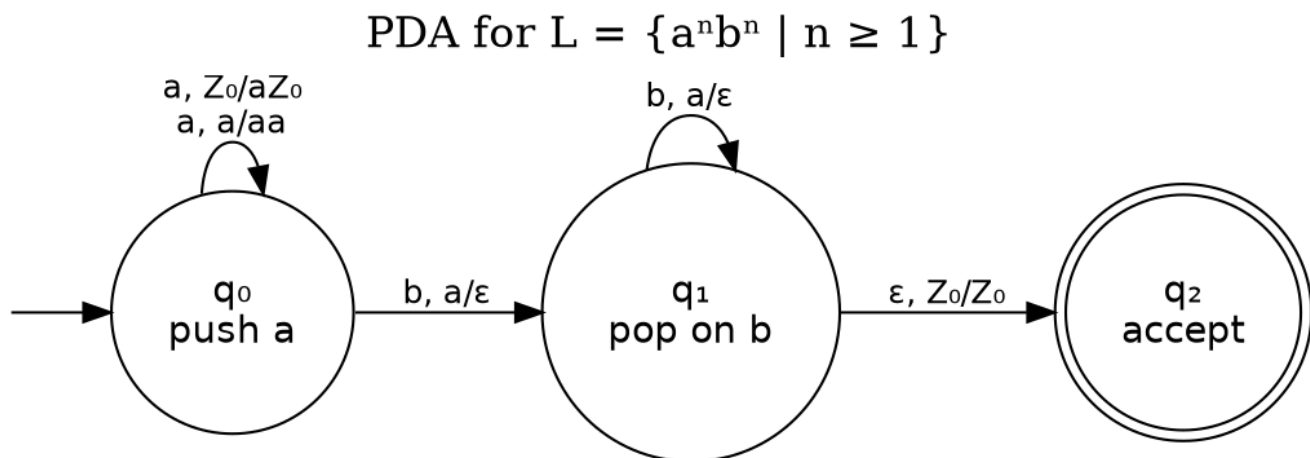


Figure 18: PDA for $L = \{a^n b^n \mid n \geq 1\}$

Strategy: Push every a onto the stack. For each b, pop one a. Accept when input is exhausted and stack is back to Z_0.

Topic 16 — Linear Bounded Automaton (LBA)

The Idea

An LBA is a Turing Machine with one constraint: the tape head cannot move outside the input portion. The tape is bounded by left and right endmarkers, so the available memory is linear in the input length.

Formal Definition

LBA = $(Q, \Sigma, \Gamma, \delta, q_0, b, F, \vdash, \dashv)$:

- Same as TM, with two additional symbols:
- $\vdash$ = left endmarker (head cannot move left of this)
- $\dashv$ = right endmarker (head cannot move right of this)

The tape always has the input bracketed between $\vdash$ and $\dashv$.

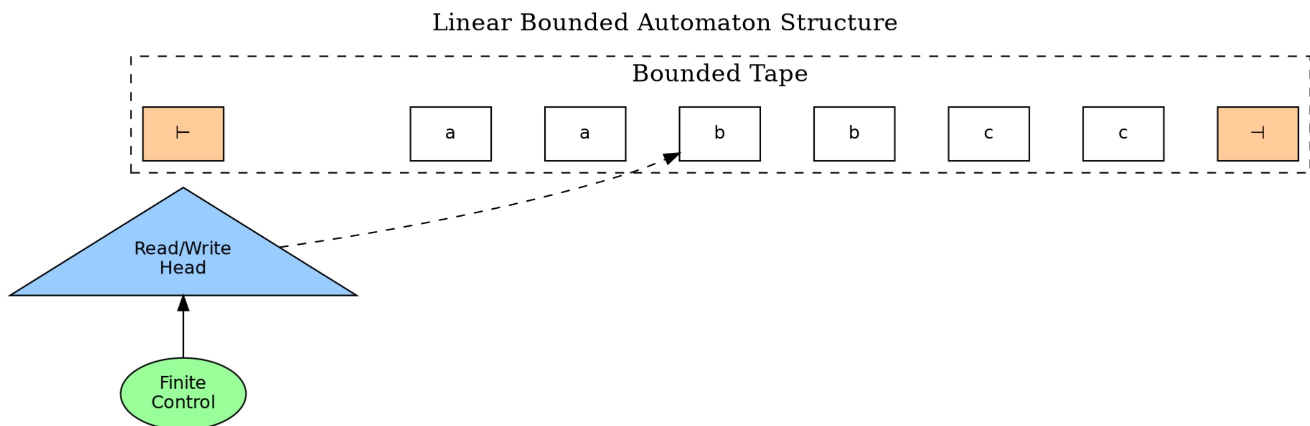


Figure 19: LBA Structure: Bounded Tape

What LBA Accepts

LBAs accept exactly **Context-Sensitive Languages (CSLs)** — Type 1 in the Chomsky Hierarchy.

Position in the Hierarchy

```
DFA  <  PDA  <  LBA  <  TM
RL   <  CFL  <  CSL  <  RE
```

LBA is strictly more powerful than PDA (it can accept $a^n b^n c^n$, which PDA cannot) but strictly less powerful than TM (it cannot accept some recursively enumerable languages that require unbounded tape).

Example

$L = \{a^n b^n c^n \mid n \geq 1\}$ is a CSL accepted by an LBA. The LBA marks each a, then a corresponding b, then a corresponding c in passes — all within the bounded tape, since the input is exactly

3n long.

Topic 17 — Halting Problem

Statement

Given a Turing Machine M and an input string w, decide whether M halts on input w (or runs forever).

The Result

The Halting Problem is undecidable. No Turing Machine (equivalently, no algorithm) can solve it for all possible (M, w) pairs.

This was proved by Alan Turing in 1936 — the first proof of an undecidable problem.

Proof Outline (by Contradiction)

1. **Assume** there exists a TM **H** that solves the Halting Problem:
   - H(M, w) = "yes" if M halts on w
   - H(M, w) = "no" if M runs forever on w
2. **Construct** a TM **D** that takes a TM description M and behaves as follows:
   - Run H(M, M).
   - If H says "yes" (M halts on M), then D enters an infinite loop.
   - If H says "no" (M loops on M), then D halts.
3. **Now run D on its own description: D(D).**
   - If D halts on D, then by D's construction, D loops on D — contradiction.
   - If D loops on D, then by D's construction, D halts on D — contradiction.

Either way, contradiction. Therefore **H cannot exist**. The Halting Problem is undecidable. ∎

Why It's Important

- It's the first proven example of an unsolvable problem.
- Many other problems are proved undecidable by reducing them to the Halting Problem.
- It tells us there are fundamental limits to what computation can decide.

Topic 18 — Post Correspondence Problem (PCP)

Statement

Given two finite lists of strings **A = (a_1, a_2, …, a_n)** and **B = (b_1, b_2, …, b_n)** over a common alphabet, decide whether there exists a sequence of indices i_1, i_2, …, i_k (with $k \geq 1$, repetition allowed) such that:

$$a_{i_1} a_{i_2} \cdots a_{i_k} = b_{i_1} b_{i_2} \cdots b_{i_k}$$

(Concatenating the chosen strings from list A in that order equals concatenating the corresponding strings from list B.)

Example

A = {1, 10, 011}, B = {101, 00, 11}

Try the sequence (1, 3, 1, 2):

- From A: 1 · 011 · 1 · 10 = “1011110”
- From B: 101 · 11 · 101 · 00 = “1011110100”

Different. The challenge is that for arbitrary instances, **no algorithm exists** to decide whether such a sequence exists.

The Result

PCP is undecidable. This was proved by Emil Post in 1946.

Significance

PCP is widely used as a starting point to prove undecidability of other problems via reduction. Famous applications include:

- Undecidability of CFG ambiguity
- Undecidability of CFG equivalence
- Undecidability of the intersection-emptiness problem for CFGs

Topic 19 — Church-Turing Thesis

Statement

Any function or problem that is **intuitively algorithmic** — computable by some finite mechanical procedure carried out by a human or a machine — is computable by a Turing Machine.

Equivalently: **Turing Machine = lambda calculus = recursive functions = real computers** in terms of what is computable (not how efficiently).

Origin

Proposed independently by:

- **Alan Turing** (1936) with his Turing Machines
- **Alonzo Church** (1936) with lambda calculus

Both formalisms were later proved equivalent in computational power.

Why It's a "Thesis," Not a "Theorem"

The notion "intuitively computable" is informal — there is no precise mathematical definition of what it means for a procedure to be algorithmic in general. So the Church-Turing thesis cannot be formally proved. However, it has been universally accepted by computer scientists for nearly a century.

Significance

- The Turing Machine is THE universal model of computation.
- Anything computable by any computer (now or in the future) is computable by a Turing Machine.
- This thesis is the foundation for the rigorous definitions of decidability and undecidability.

Topic 20 — Recursive vs Recursively Enumerable

Definitions

A language L is:

- **Recursive (or decidable):** There exists a TM that always halts and correctly decides whether any given input is in L.
- **Recursively Enumerable (RE) (or semi-decidable):** There exists a TM that halts and accepts every $w \in L$. For $w \notin L$, the TM may either halt and reject OR loop forever.

The Distinction

Property	Recursive (RS)	Recursively Enumerable (RES)
TM behavior	Always halts	Halts on members; may loop on non-members
Membership	Decidable	Semi-decidable only
Closure under complement	Yes (RS is closed under complement)	No

Key Theorem

A language L is recursive $\iff$ both L and its complement $\bar{L}$ are recursively enumerable.

Famous Examples

- **The Halting Problem language** $\{\langle M, w\rangle : M \text{ halts on } w\}$ is RE but NOT recursive.
- **The complement of the Halting Problem** is NOT even RE.

Containment

$$\text{Recursive} \subset \text{Recursively Enumerable} \subset \text{All Languages}$$

Topic 21 — Decision Problems of CFL

Decidable Problems

Problem	Question	Algorithm
Membership	Is $w \in L(G)$?	CYK algorithm in $O(n^3)$ after CNF conversion
Emptiness	Is $L(G) = \emptyset$?	Check whether S is generating
Finiteness	Is L(G) finite?	Look for cycles in dependency graph
Infiniteness	Is L(G) infinite?	Same as above (presence of cycle)

Undecidable Problems

Problem	Question
Equivalence	Is $L(G_1) = L(G_2)$?
Ambiguity	Is grammar G ambiguous?
Inherent ambiguity	Is L inherently ambiguous?
Σ^* test	Is $L(G) = \Sigma^*$?
Intersection emptiness	Is $L(G_1) \cap L(G_2) = \emptyset$?
Containment	Is $L(G_1) \subseteq L(G_2)$?

Most of these undecidability results are proved by **reduction from the Post Correspondence Problem.**

Topic 22 — Closure Properties of Regular Languages

Regular languages are closed under:

- **Union** — $L_1 \cup L_2$
- **Concatenation** — $L_1 \cdot L_2$
- **Kleene Star** — L^*
- **Positive Closure** — L^+
- **Complement** — $\bar{L}$
- **Intersection** — $L_1 \cap L_2$
- **Set Difference** — $L_1 - L_2$
- **Reversal** — L^R
- **Homomorphism** and **Inverse Homomorphism**

Key takeaway: Regular languages are closed under almost every standard operation. CFLs lose closure under intersection and complement.

Operation	Reg	DCFL	CFL	CSL	Rec	RE
Union	✓	✗	✓	✓	✓	✓
Intersection	✓	✗	✗	✓	✓	✓
Complement	✓	✓	✗	✓	✓	✗
Concatenation	✓	✗	✓	✓	✓	✓
Kleene star	✓	✗	✓	✓	✓	✓
Reversal	✓	✓	✓	✓	✓	✓
Intersection w/ regular	✓	✓	✓	✓	✓	✓

2024 Paper — Complete Solutions

Section A (Very Short Answer — 5 × 2 = 10 marks)

Q1. Define automata. (2 marks)

An automaton is a self-operating abstract computing model defined by the 5-tuple **($Q, \Sigma, \delta, q_0, F$)**, where Q is a finite set of states, Σ is the input alphabet, δ is the transition function, q_0 is the initial state, and F is the set of final states. It reads input symbols and transitions between states, accepting a string if it ends in a state in F.

Q2. Write the definition of Turing machine. (2 marks)

A Turing Machine is a 7-tuple **($Q, \Sigma, \Gamma, \delta, q_0, b, F$)** where Q is the set of states, Σ is the input alphabet, Γ is the tape alphabet, $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the transition function, q_0 is the initial state, $b \in \Gamma$ is the blank symbol, and F is the set of final states. It has an infinite two-way tape with a read/write head and is the most powerful computational model — it accepts recursively enumerable languages.

Q3. Explain derivation tree in CFG. (2 marks)

A derivation tree (parse tree) is a tree representation of how a CFG derives a string. The **root** is the start symbol S, **internal nodes** are non-terminal variables, and **leaf nodes** are terminal symbols. Reading the leaves left-to-right gives the derived string. For example, the grammar S → aSb | ab generates "aabb" with the parse tree shown below.

Derivation Tree for "aabb" using S → aSb | ab

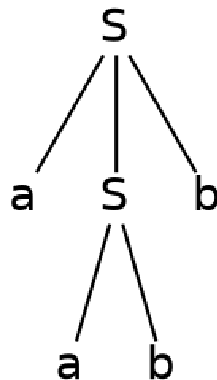


Figure 20: Derivation tree for "aabb"

Q4. Definition of grammar. (2 marks)

A grammar is a formal system **G = (V, T, P, S)** where V is the set of non-terminal variables, T is the set of terminal symbols, P is the set of production rules of the form $A \to \alpha$ (where $A \in V$ and $\alpha \in (V \cup T)^*$), and S is the start symbol. A grammar generates a language by repeatedly applying productions starting from S until only terminals remain.

Q5. Prove (a + b)* = a*(ba*)*. (2 marks)

(a + b)* generates all strings over {a, b} including ε.

a*(ba*)* generates: zero or more a's, followed by zero or more "ba*" blocks (i.e., each subsequent b followed by some a's).

Any string of a's and b's can be uniquely decomposed as: leading a's, then for each b in order, the a's that follow it. This is exactly the structure of a*(ba*)*. Conversely, any string in a*(ba*)* contains only a's and b's, so it is in (a + b)*.

Therefore **(a + b)* = a*(ba*)***. ∎

Section B (Short Answer — 9 marks each, attempt any 2 of 3)

Q6. Explain the Chomsky Hierarchy. (9 marks)

Introduction: Noam Chomsky classified all formal grammars into a four-level hierarchy based on the form of their production rules. The four classes are nested: each is a proper subset of the level above it (RL ⊂ CFL ⊂ CSL ⊂ RE).

Type 0 — Unrestricted Grammar: Productions of the form **$\alpha \rightarrow \beta$** with no restriction (α must contain at least one non-terminal). Generates **Recursively Enumerable languages**, accepted by **Turing Machines**.

Type 1 — Context-Sensitive Grammar: Productions of the form **$\alpha A\beta \rightarrow \alpha\delta\beta$**, with the length condition $|LHS| \leq |RHS|$ (non-contracting). Generates **Context-Sensitive Languages**, accepted by **Linear Bounded Automata**. Example language: $\{a^n b^n c^n \mid n \geq 1\}$.

Type 2 — Context-Free Grammar: Productions of the form **$A \rightarrow \alpha$**, where A is a single non-terminal and $\alpha \in (V \cup T)^*$. Generates **Context-Free Languages**, accepted by **Pushdown Automata**. Example: $S \rightarrow aSb \mid ab$ generates $\{a^n b^n \mid n \geq 1\}$.

Type 3 — Regular Grammar: Productions of the form **$A \rightarrow a$** or **$A \rightarrow aB$** (right-linear), or **$A \rightarrow a$** or **$A \rightarrow Ba$** (left-linear). The two forms cannot be mixed. Generates **Regular Languages**, accepted by **Finite Automata** (DFA / NFA). Example: $S \rightarrow aS \mid b$ generates $\{a^n b \mid n \geq 0\}$.

Summary:

Type	Grammar	Language	Machine	Example
0	Unrestricted	RE	TM	Halting problem
1	Context-Sensitive	CSL	LBA	$a^n b^n c^n$
2	Context-Free	CFL	PDA	$a^n b^n$
3	Regular	RL	DFA/NFA	$a^n b$

The Chomsky Hierarchy provides the fundamental framework for classifying formal languages and the machines that recognize them.

Q7. Construct a DFA with reduced states equivalent to RE: $10 + (0+11)\cdot 0^*\cdot 1$. (9 marks)

Step 1 — Build NFA from RE using Thompson's construction:

The RE has two parts joined by union:

- Part 1: **10** — the literal string "10"
- Part 2: **$(0 + 11)\cdot 0^*\cdot 1$** — starts with 0 or 11, then 0's, then 1

We construct an NFA with multiple paths covering both alternatives. After Thompson's construction we get an NFA with about 6 states.

Step 2 — Convert NFA to DFA using subset construction:

After applying the subset construction algorithm and removing unreachable states, we get a reduced DFA. The resulting DFA must accept exactly:

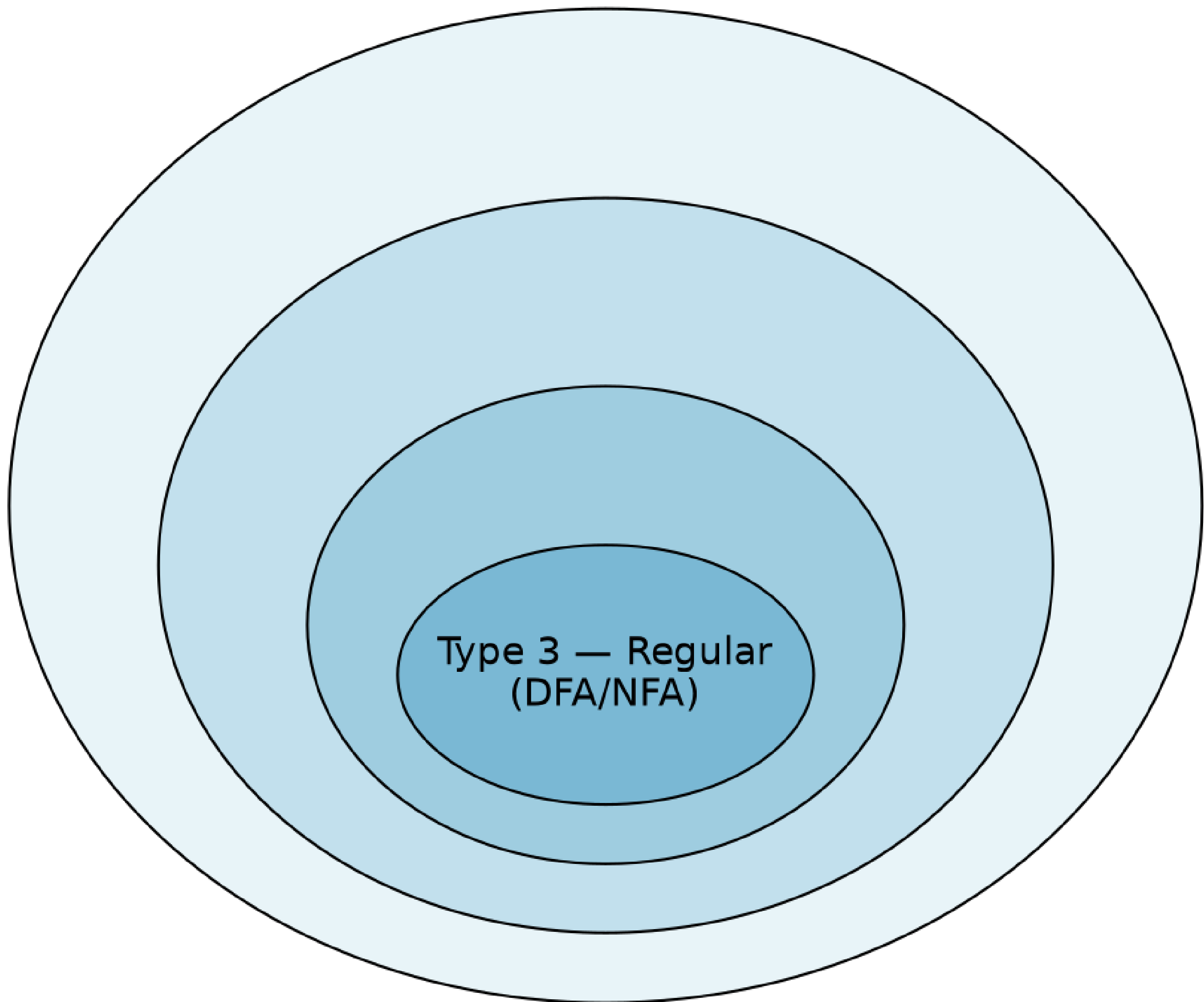


Figure 21: Chomsky Hierarchy nested containment

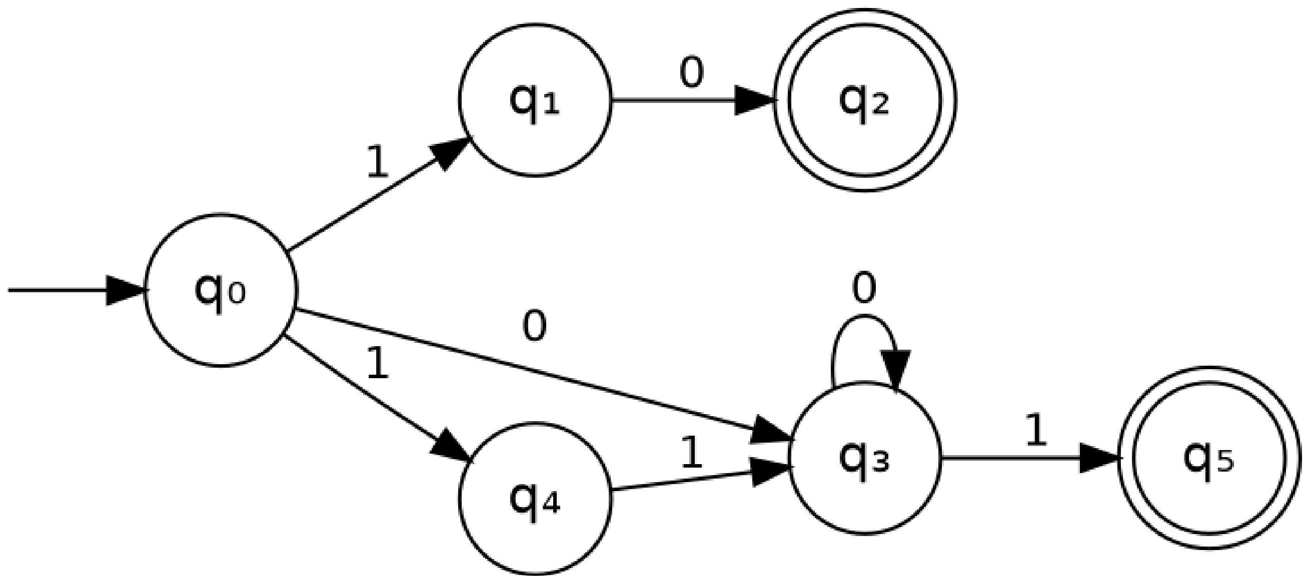


Figure 22: NFA from RE 10 + (0+11)·0*·1

- The string “10”
- Any string starting with “0” or “11”, followed by any number of 0’s, ending with “1”

Step 3 — Reduced DFA:

State	On 0	On 1
→ A	C	B
B	*D	C
C	C	*D
*D	E	E
E (dead)	E	E

States: A (start), B (saw “1” — needs “0” to reach final), C (saw “0” or “11” — needs “1”), D (final), E (dead state).

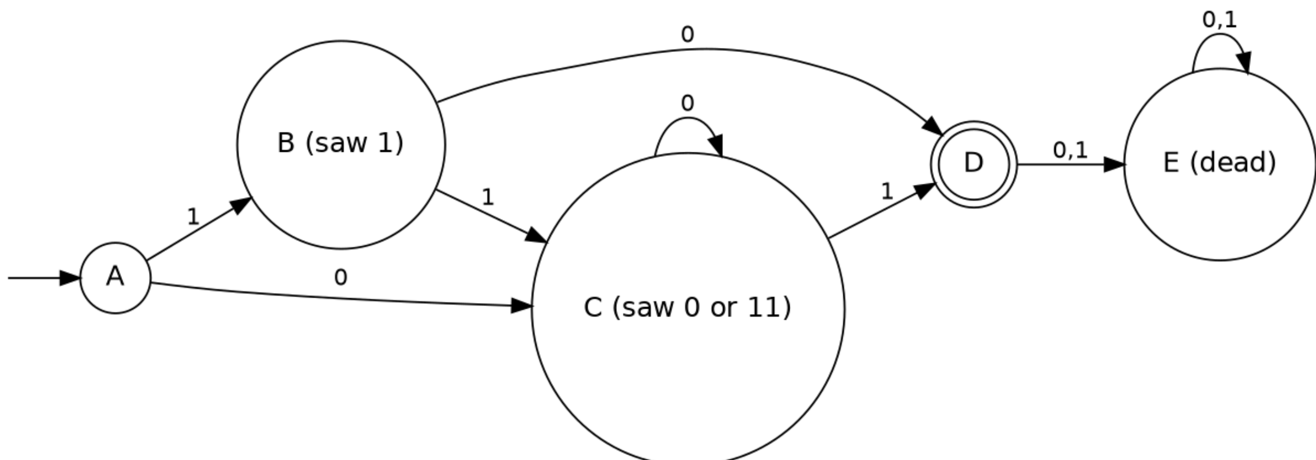


Figure 23: Reduced DFA for RE 10 + (0+11)·0*·1

A grammar is **right-linear** if all productions are of the first two forms, and **left-linear** if all productions are of the first or third form. Mixing left-linear and right-linear productions in the same grammar may produce a non-regular language.

Position in Chomsky Hierarchy: Type 3 — the most restricted grammar type. Generates exactly the regular languages, which are accepted by finite automata.

Example 1 — Right-linear grammar for L = {$a^n b$ | n ≥ 0}:

S → aS | b

Derivations: S ⇒ b, S ⇒ ab, S ⇒ aab, S ⇒ aaab, …

The corresponding DFA has 2 states:

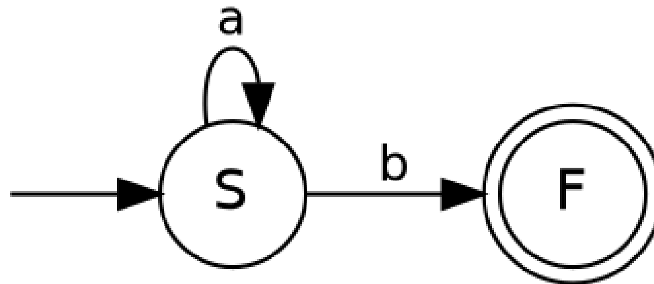


Figure 24: DFA from regular grammar S → aS | b

Example 2 — Right-linear grammar for binary strings ending in 1:

S → 0S | 1S | 1

Generates: 1, 01, 11, 001, 011, 101, 111, …

Example 3 — Left-linear grammar for the same language:

S → S0 | S1 | 1

Same language; different style of grammar.

Equivalence with FA: Every regular grammar can be converted to a finite automaton and vice versa. Non-terminals correspond to states, and productions correspond to transitions. This bidirectional equivalence is the heart of Kleene's theorem.

Section C (Long Answer — 14 marks each, attempt any 3 of 5)

Q9. Differentiate between DFA and NFA with example. (14 marks)

Definitions:

A **DFA (Deterministic Finite Automaton)** is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where $\delta : Q \times \Sigma \to Q$ gives exactly one next state for each (state, input) pair.

An **NFA (Non-deterministic Finite Automaton)** is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where $\delta : Q \times \Sigma \to 2^Q$ gives a set of possible next states. ε-transitions are also allowed.

Comparison Table:

Aspect	DFA	NFA
Transition function	δ: $Q \times \Sigma \to Q$ (single state)	δ: $Q \times \Sigma \to 2^Q$ (set of states)
Determinism	Exactly one next state per (state, symbol)	Zero, one, or many next states
ε-transitions	Not allowed	Allowed
Backtracking	Not needed	May be needed
Acceptance	Unique computation path; ends in F	Some computation path ends in F
Number of states	Usually more (up to 2^n from NFA)	Usually fewer
Construction from RE	Difficult	Easy (Thompson’s construction)
Implementation	Direct in hardware/software	Requires simulation

Example — DFA accepting strings ending in “01”:

States: q_0 (start), q_1 (saw 0), q_2 (saw 01 — final). On 0: $q_0 \to q_1$, $q_1 \to q_1$, $q_2 \to q_1$. On 1: $q_0 \to q_0$, $q_1 \to q_2$, $q_2 \to q_0$.

Example — NFA accepting strings containing “01”:

States: q_0 (start), q_1 (saw 0), q_2 (saw 01 — final). From q_0 on 0: $\{q_0, q_1\}$ (non-deterministic guess: stay or commit). From q_0 on 1: $\{q_0\}$. From q_1 on 1: $\{q_2\}$. From q_2: stay in q_2 on either input.

The NFA has fewer transitions to manage but multiple computation paths. The same language could be accepted by a DFA with more states.

Equivalence:

By the **Rabin-Scott Subset Construction Theorem**, every NFA can be converted to an equivalent DFA. An NFA with n states converts to a DFA with at most 2^n states.

Conclusion: Despite the apparent flexibility of non-determinism, NFAs do NOT have more computational power than DFAs. **Both DFA and NFA accept exactly the class of regular languages.**

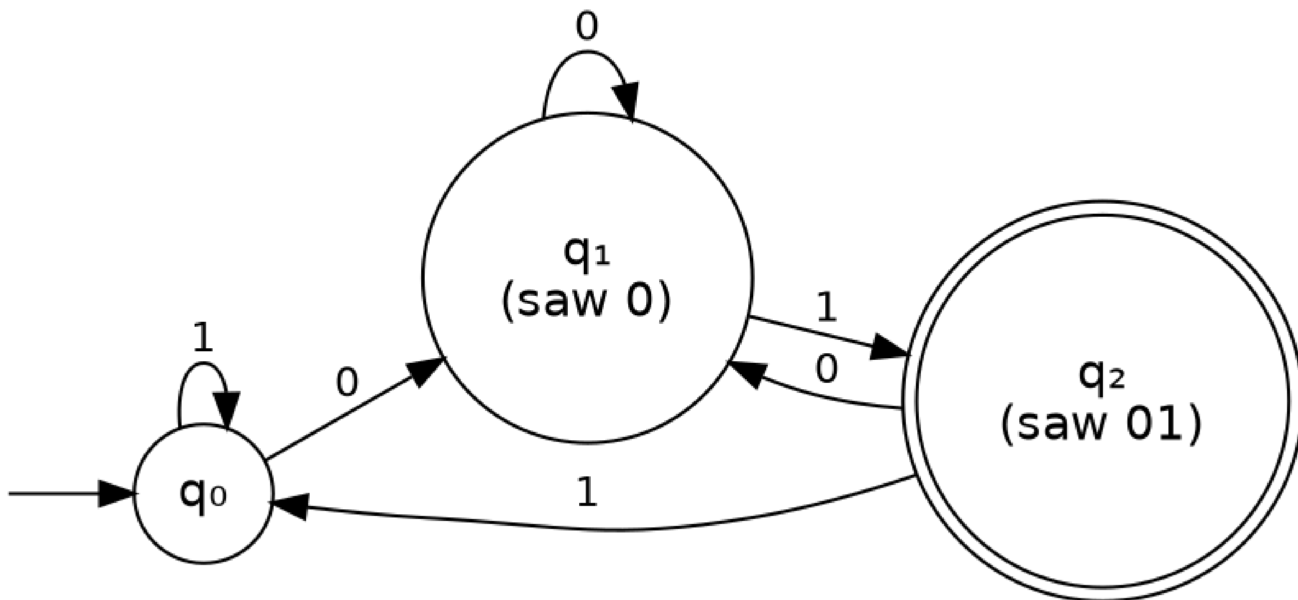


Figure 25: DFA for strings ending in “01”

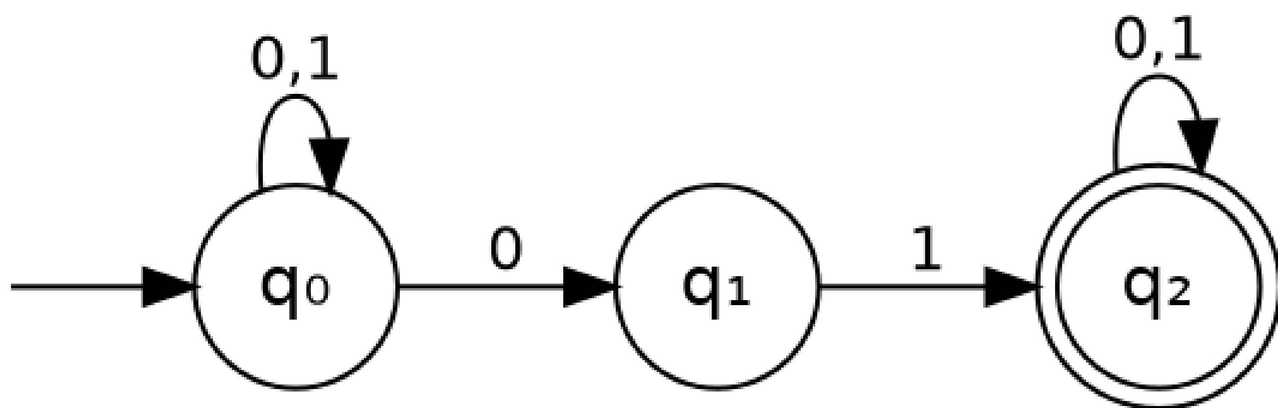


Figure 26: NFA for strings containing “01”

Q10. Design a Turing machine for L = {$a^n b^n$ | n ≤ 1}. (14 marks)

Note: The original paper writes "n ≤ 1" but this is almost certainly a typo for "n ≥ 1" — the standard textbook problem. We solve the standard version.

Strategy: Match each a with one b. Replace matched a's with X and matched b's with Y. When all matched, accept.

States:

- q_0 — find next unmarked a
- q_1 — found a, scanning right for b
- q_2 — marked b, scanning back left
- q_3 — verification mode
- q_4 — accept (final)

Transition Table:

State	a	b	X	Y	B (blank)
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**	—	—	—	—	—

State Diagram:

Trace for input "aabb":

Step	Tape	State
0	aabb	q_0
1	Xabb	q_1
2	Xabb (head moved)	q_1
3	XaYb (b → Y, head left)	q_2
4	XaYb (head left)	q_2
5	XaYb (X found, head right)	q_0
6	XXYb	q_1
7	XXYY	q_2
8	XXYY → final scan	$q_0 \rightarrow q_3$
9	Read blank	q_4 — **ACCEPT**

Conclusion: The TM M = (Q, {a, b}, {a, b, X, Y, B}, δ, q_0, B, {q_4}) accepts exactly the language {$a^n b^n$ | n ≥ 1}.

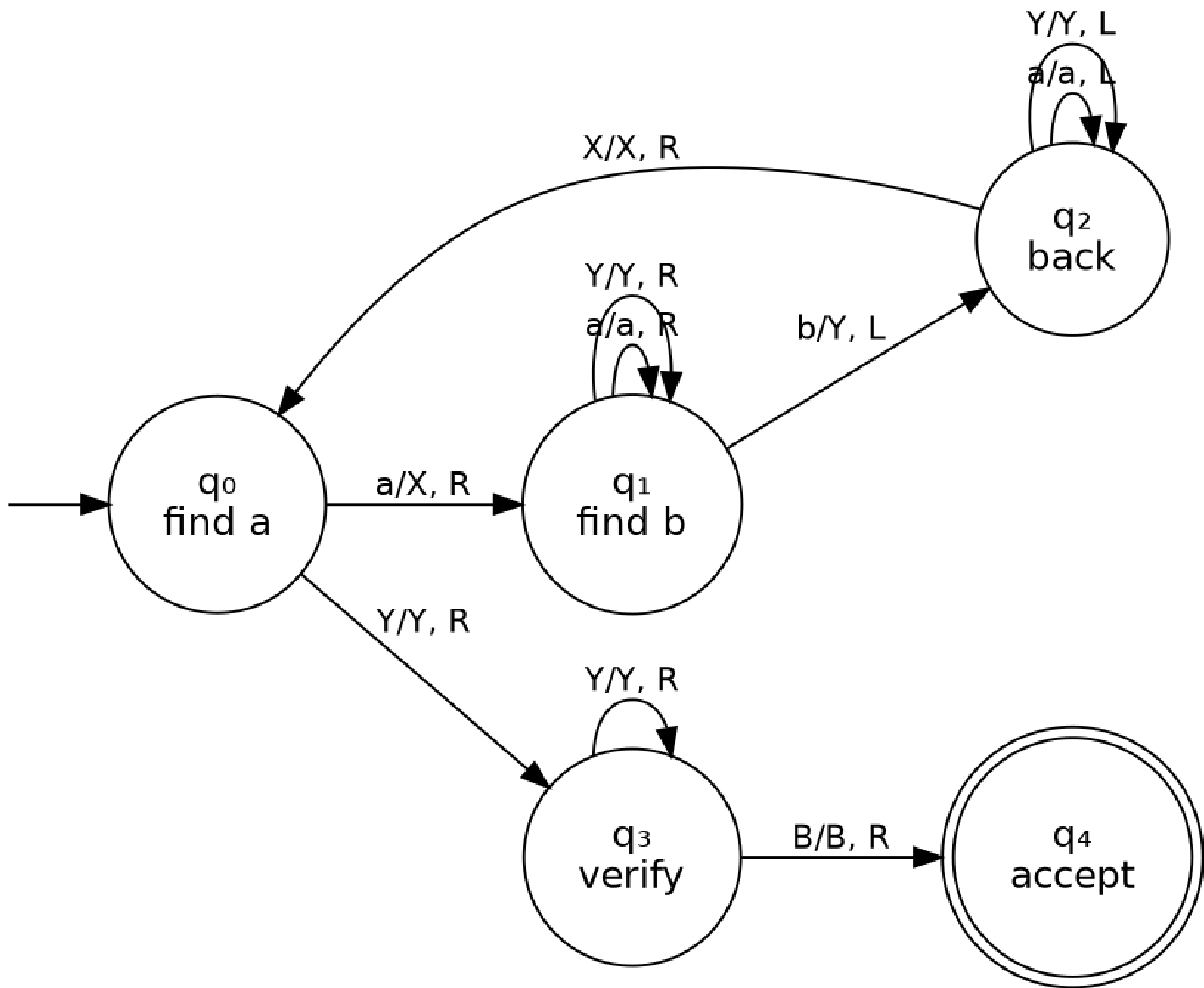


Figure 27: TM for $L = \{a^n b^n \mid n \geq 1\}$

Q11. Explain the procedure to convert a Mealy machine into its corresponding Moore machine, with the help of an example. (14 marks)

Mealy and Moore Definitions:

A **Mealy machine** is a 6-tuple $(Q, \Sigma, \Delta, \delta, \lambda, q_0)$ where the output function $\lambda : Q \times \Sigma \to \Delta$ depends on the **current state and the current input**. Outputs are placed on transitions.

A **Moore machine** is a 6-tuple $(Q, \Sigma, \Delta, \delta, \lambda, q_0)$ where the output function $\lambda : Q \to \Delta$ depends only on the **current state**. Outputs are placed on states.

Both machines are equal in computational power and can be inter-converted.

Conversion Procedure (Mealy → Moore) — 4 Steps:

1. **For each state in the Mealy machine, list all distinct outputs on incoming transitions.**
2. **If a state has only one distinct incoming output**, keep it as a single Moore state and assign that output.
3. **If a state has multiple distinct incoming outputs** (e.g., z_1 and z_2), **split the state into copies**, one per output value (e.g., q_2 becomes q_2/z_1 and q_2/z_2).
4. **Redirect each transition** to the copy whose output matches the output that was on that Mealy edge. Then remove the output labels from the edges.

For the **starting state**, since it has no incoming transition, assign any default output (or it may also need splitting if the rest of the procedure requires).

Worked Example:

Consider this Mealy machine where the output is y whenever the input completes the sub-string “ab”, and n otherwise:

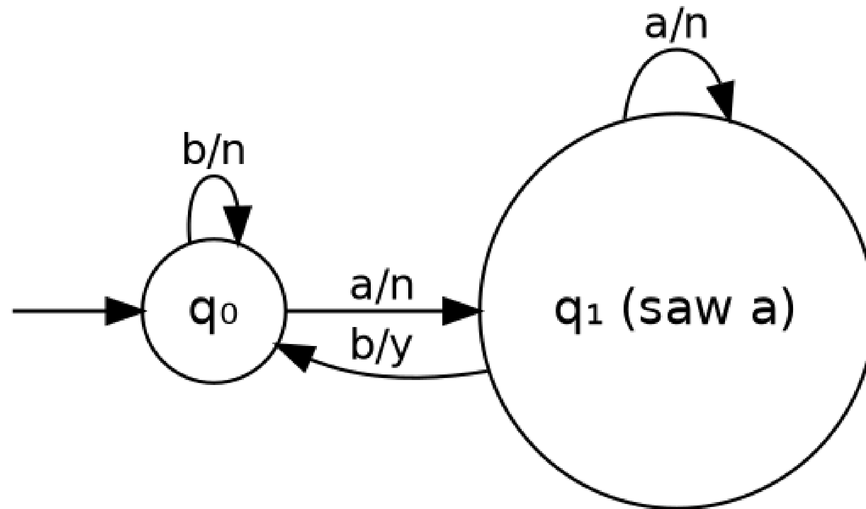


Figure 28: Mealy machine for substring “ab”

States: q_0 (start), q_1 (just saw a). Outputs: y, n.

Transitions: q_0 on a/n → q_1; q_0 on b/n → q_0; q_1 on a/n → q_1; q_1 on b/y → q_0.

Apply the conversion:

- Incoming outputs to q_0: from q_0 on b (output n), from q_1 on b (output y). → Multiple outputs → **split q_0 into q_0^a (output n) and q_0^b (output y)**.
- Incoming outputs to q_1: from q_0 on a (output n), from q_1 on a (output n). → Single output → keep q_1 with output n.

After conversion:

- q_0^a (initial, output n) — on a → q_1; on b → q_0^a
- q_0^b (output y) — on a → q_1; on b → q_0^a
- q_1 (output n) — on a → q_1; on b → q_0^b

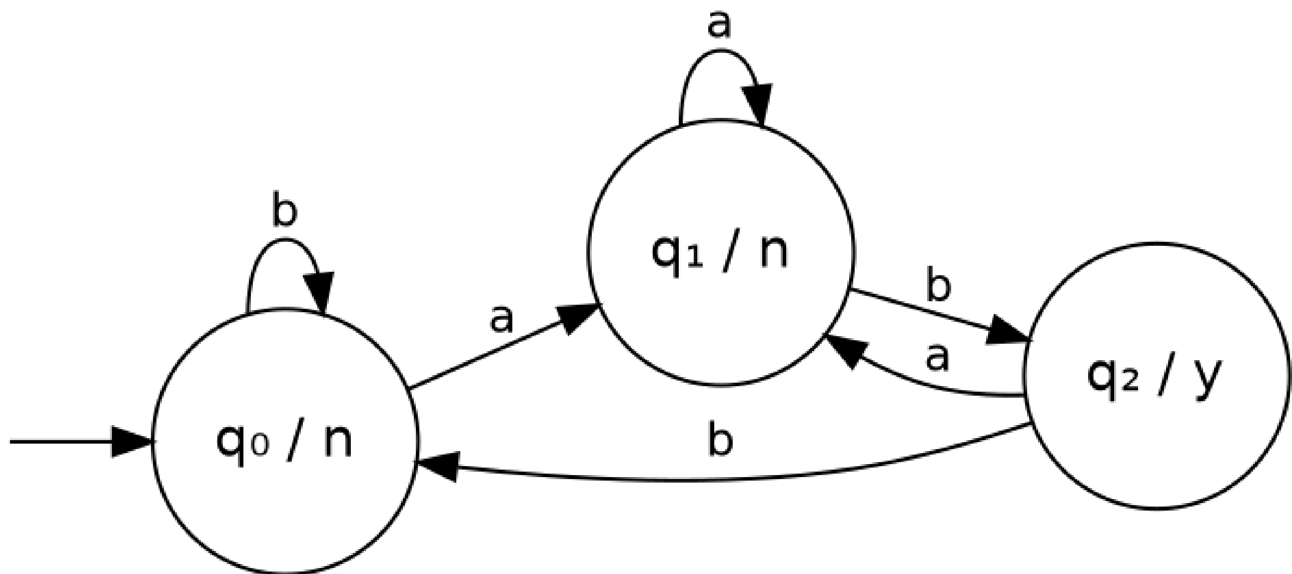


Figure 29: Moore machine after conversion

Verification — process "aab":

- Mealy: q_0 → (a/n) q_1 → (a/n) q_1 → (b/y) q_0. Output sequence: n, n, y.
- Moore: q_0^a → q_1(n) → q_1(n) → q_0^b(y). Output sequence (from each state visited): n, n, y. ✓ Same.

Conclusion: The conversion always produces an equivalent Moore machine, possibly with more states. Mealy and Moore machines accept the same class of computations and are interchangeable.

Q12. Explain the following: (a) Linear Bounded Automata (b) Arden's Theorem. (14 marks)

Part (a) — Linear Bounded Automata (7 marks) A **Linear Bounded Automaton (LBA)** is a restricted form of Turing Machine in which the tape head cannot move outside the input portion of the tape. The tape is bounded by a left endmarker ⊢ and a right endmarker ⊣, and the head is required to stay between them.

Formal Definition: LBA = (Q, Σ, Γ, δ, q_0, b, F, ⊢, ⊣). All components are the same as a Turing Machine, with the addition of the two endmarker symbols.

Working: The LBA reads input bracketed by ⊢ and ⊣. The head can read, write, and move L/R within these bounds, but cannot move past either endmarker. The available memory is therefore **linear in the input length**, which is where the name comes from.

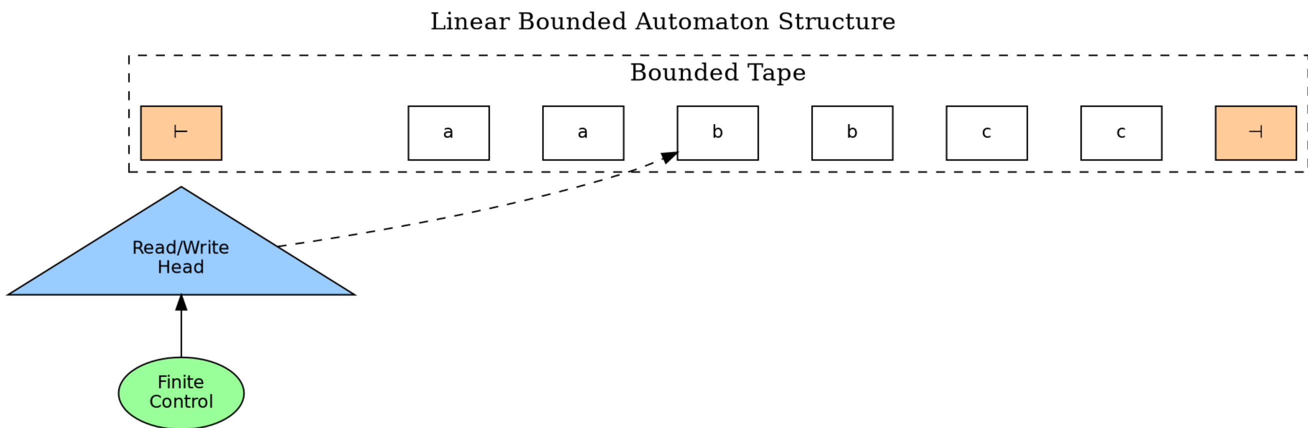


Figure 30: LBA Structure

Languages accepted: Context-Sensitive Languages (CSLs) — Type 1 in the Chomsky Hierarchy.

Example: L = $\{a^n b^n c^n \mid n \geq 1\}$ is a CSL accepted by an LBA. The LBA marks one a as X, one b as Y, one c as Z in each pass; everything fits within the bounded tape since the input has fixed length 3n.

Position in the hierarchy:

$$\text{DFA} < \text{PDA} < \text{LBA} < \text{TM}$$

LBA is strictly more powerful than PDA (it accepts $a^n b^n c^n$ which PDA cannot), and strictly less powerful than TM (it cannot accept all RE languages because the tape is bounded).

Part (b) — Arden's Theorem (7 marks) **Statement:** If P is a regular expression that does NOT contain ε (empty string), then the equation **R = Q + RP** has a unique solution **R = QP***.

Proof: Substituting R = QP* into the RHS:

$$Q + (QP^*)P = Q + QP^*P = Q(\varepsilon + P^*P) = Q \cdot P^* = R\checkmark$$

So $R = QP^*$ satisfies the equation. Uniqueness follows because the equation $R = Q + RP$, with P not containing ε, has a least fixed-point that is exactly QP^*.

Use: Arden's theorem provides an algebraic method to **convert a finite automaton into a regular expression**.

Procedure:

1. For each state q_i, write an equation of the form q_i = (sum over edges arriving at q_i of source_state · symbol). For the initial state, also include ε.
2. Solve the system of equations using Arden's theorem (substituting and applying the formula).
3. The expression for the final state(s) gives the regular expression for L(M).

Worked Example:

Consider the DFA: A (initial), B (final). δ(A, a) = A; δ(A, b) = B; δ(B, a) = B; δ(B, b) = B.

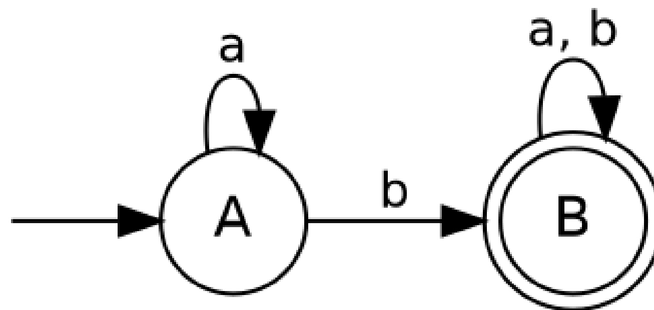


Figure 31: DFA for Arden example

Equations:

- $A = \varepsilon + Aa$
- $B = Ab + Ba + Bb = Ab + B(a + b)$

Solve A: $A = \varepsilon + Aa$ is in the form $R = Q + RP$ with $Q = \varepsilon$, $P = a$. Apply Arden's: $A = \varepsilon \cdot a^*$ = **a^***.

Solve B: Substitute A: $B = a^*b + B(a + b)$. Apply Arden's with $Q = a^*b$, $P = (a + b)$: B = **$a^*b(a + b)^*$**.

Regular Expression: L(M) = **$a^*b(a + b)^*$** — strings over {a, b} containing at least one b.

Q13. Explain the following: (a) Halting Problem in Turing Machine (b) Pumping Lemma for regular expression. (14 marks)

Part (a) — Halting Problem (7 marks) **Statement:** The Halting Problem is the decision problem of determining, given a Turing Machine M and an input string w, whether M halts on input w (or runs forever).

Result: Alan Turing proved in 1936 that the Halting Problem is **undecidable** — there is no Turing Machine (equivalently, no algorithm) that can decide it for all possible (M, w) pairs.

Proof by Contradiction:

1. Assume there exists a TM **H** that solves the Halting Problem:
   - H(M, w) outputs "yes" if M halts on w; "no" otherwise.
2. Construct a new TM **D** that takes a TM description M as input and:
   - Runs H(M, M).
   - If H outputs "yes" (M halts on M), then D enters an infinite loop.
   - If H outputs "no" (M loops on M), then D halts immediately.
3. Now run **D(D)** — feed D its own description as input:
   - If D halts on D, then by construction D loops on D — **contradiction**.
   - If D loops on D, then by construction D halts on D — **contradiction**.

Either way, we have a contradiction. Therefore **H cannot exist**, and the Halting Problem is undecidable. ∎

Significance: This was the first proven undecidable problem and is the foundation for proving many other problems undecidable via reduction. It establishes a fundamental limit on what computation can decide.

Part (b) — Pumping Lemma for Regular Languages (7 marks) **Statement:** For any regular language L, there exists an integer **n** (called the pumping length) such that for every string $z \in L$ with $|z| \geq n$, z can be split as $z = uvw$ satisfying:

1. $|uv| \leq n$
2. $|v| \geq 1$
3. **For all** $i \geq 0$, $uv^iw \in L$

Use: The Pumping Lemma is used to **prove that a language is NOT regular** by contradiction. (It cannot prove regularity — only non-regularity.)

Proof Template:

To show L is not regular:

1. Assume L is regular; let n be the pumping length.
2. Choose a specific $z \in L$ with $|z| \geq n$.
3. Show that for any split $z = uvw$ with the lemma's constraints, some choice of i makes $uv^iw \notin L$.
4. This contradiction proves L is not regular.

Worked Example: Prove $L = \{a^nb^n \mid n \geq 1\}$ **is not regular.**

Assume L is regular; let n be the pumping length.

Choose $z = a^nb^n \in L$ with $|z| = 2n \geq n$.

Split $z = uvw$ with $|uv| \leq n$ and $|v| \geq 1$. Since the first n characters of z are all a's, both u and v consist only of a's. So $v = a^k$ for some $k \geq 1$.

Pump with $i = 2$: $uv^2w = a^{n+k} b^n$. This has $n + k$ a's but only n b's; since $k \geq 1$, $n + k \neq n$.

So $uv^2w \notin L$. **Contradiction!** ∎

Therefore $L = \{a^n b^n \mid n \geq 1\}$ is not regular.

Second Example: Prove $L = \{a^p \mid p$ is prime$\}$ is not regular.

Choose $z = a^p$ for a prime $p \geq n + 2$. Split $z = uvw$ with $v = a^k$, $k \geq 1$.

Pump with $i = p$: $|uv^pw| = p + (p - 1)k$. For $k \geq 1$, this expression is composite. So $uv^pw \notin L$. Contradiction. ∎

2025 Paper — Complete Solutions

Section A (Very Short Answer — 5 × 2 = 10 marks)

Q1. What is the difference between DFA and NFA? (2 marks)

In a **DFA**, the transition function $\delta : Q \times \Sigma \rightarrow Q$ gives **exactly one** next state for each (state, input) pair, and ε-transitions are not allowed. In an **NFA**, $\delta : Q \times \Sigma \rightarrow 2^Q$ gives a **set** of possible next states, and ε-transitions are allowed. However, both have **equal computational power**: every NFA can be converted to an equivalent DFA via the subset construction, and both accept exactly the class of regular languages.

Q2. Give an example of a non-regular language. (2 marks)

Example: $L = \{a^n b^n \mid n \geq 1\} = \{ab, aabb, aaabbb, aaaabbbb, \ldots\}$

Why it's non-regular: This language requires counting the number of a's and matching it with an equal number of b's. Finite automata have only finite memory and cannot count to arbitrary n. This is formally proved using the Pumping Lemma. The language is, however, context-free (accepted by a PDA via $S \rightarrow aSb \mid ab$).

Q3. Write a Regular Expression for strings over {0, 1} that contain at least one 1. (2 marks)

Regular Expression: (0 + 1)* · 1 · (0 + 1)* or equivalently **0* · 1 · (0 + 1)***.

This generates any binary string with at least one explicit 1: zero or more arbitrary symbols, then a 1, then zero or more arbitrary symbols.

Q4. State two differences between PDA and Turing Machine. (2 marks)

Difference 1 (Memory): A PDA uses a single LIFO **stack** as its only memory; a Turing Machine uses an **infinite two-way tape** that supports both reading and writing at any position.

Difference 2 (Power): A PDA accepts only **Context-Free Languages** and cannot recognize $\{a^n b^n c^n\}$; a Turing Machine accepts all **Recursively Enumerable Languages**, including non-CFLs like $\{a^n b^n c^n\}$, making it strictly more powerful.

Q5. What is the role of stack in a PDA? (2 marks)

The stack in a PDA provides **Last-In-First-Out (LIFO) auxiliary memory**, enabling the PDA to store and recall symbols. This allows the PDA to accept languages beyond the capability of finite automata. For instance, on input $a^n b^n$, the PDA pushes each a onto the stack and pops one for each b, accepting if the stack is empty at the end. **Without the stack, a PDA reduces to a DFA** and cannot recognize context-free languages.

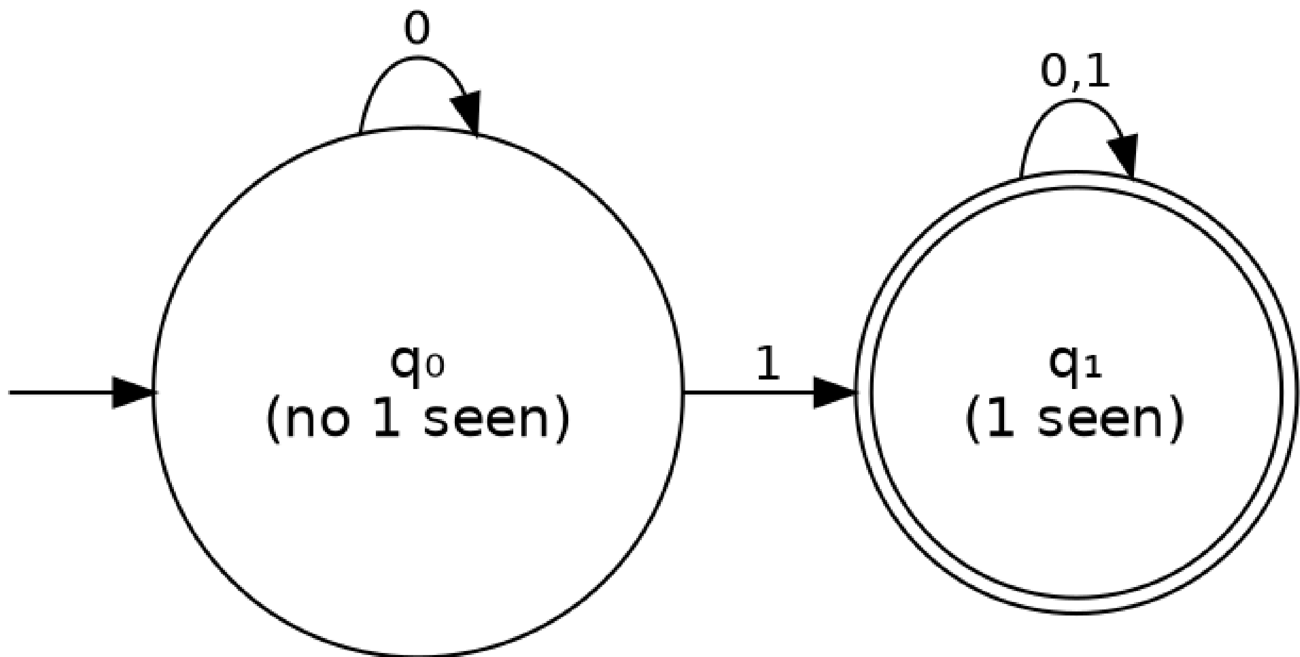


Figure 32: DFA accepting "at least one 1"

Section B (Short Answer — 9 marks each, attempt any 2 of 3)

Q6. Convert the CFG S → aS | Sb | ab into CNF. (9 marks)

A grammar is in **Chomsky Normal Form (CNF)** if every production is of the form **A → BC** (two non-terminals) or **A → a** (single terminal).

Conversion:

Step 1 — Eliminate ε-productions: None present. Skip.

Step 2 — Eliminate unit productions (A → B form): None present. Skip.

Step 3 — Replace terminals in productions of length ≥ 2:

- In S → aS, the terminal a appears with another symbol. Introduce **T_a → a**.
- In S → Sb, the terminal b appears with another symbol. Introduce **T_b → b**.
- In S → ab, replace both terminals.

After replacement:

$$\begin{aligned} S &\to T_a S \mid S T_b \mid T_a T_b \\ T_a &\to a \\ T_b &\to b \end{aligned}$$

Step 4 — Binarize: All productions now have at most two non-terminals on the RHS, so no further binarization is needed.

Final CNF Grammar:

$$\boxed{S \to T_aS \mid ST_b \mid T_aT_b, \quad T_a \to a, \quad T_b \to b}$$

Verification: Every production is of the form A → BC or A → a. The grammar is in CNF. ✓

Q7. Explain the Chomsky Hierarchy with suitable example for each language type. (9 marks)

See 2024 Q6 — same answer reproduced here for completeness.

Introduction: Noam Chomsky classified all formal grammars into a four-level hierarchy based on the form of their production rules. The four classes are nested: each is a proper subset of the level above (Regular ⊂ CFL ⊂ CSL ⊂ RE).

Type 0 — Unrestricted Grammar: Productions α → β with no restriction. Generates **Recursively Enumerable** languages, accepted by **Turing Machines**.

Example: Grammars that simulate arbitrary algorithms — generates languages such as {⟨M, w⟩ : M halts on w}.

Type 1 — Context-Sensitive Grammar: Productions of the form αAβ → αδβ with the length condition |LHS| ≤ |RHS|. Generates **Context-Sensitive Languages**, accepted by **Linear Bounded Automata**.

Example: L = $\{a^nb^nc^n \mid n \geq 1\}$.

Type 2 — Context-Free Grammar: Productions A → α, where A is a single non-terminal. Generates **Context-Free Languages**, accepted by **Pushdown Automata**.

Example: Grammar S → aSb | ab generates L = $\{a^nb^n \mid n \geq 1\}$.

Type 3 — Regular Grammar: Productions A → a or A → aB (right-linear), or A → a or A → Ba (left-linear). Generates **Regular Languages**, accepted by **Finite Automata**.

Example: S → aS | b generates L = $\{a^nb \mid n \geq 0\}$.

Summary Table:

Type	Grammar	Language	Machine	Example
0	Unrestricted	RE	TM	Halting problem
1	Context-Sensitive	CSL	LBA	$a^nb^nc^n$
2	Context-Free	CFL	PDA	a^nb^n
3	Regular	RL	DFA/NFA	a^nb

Conclusion: As we move up the hierarchy, grammar restrictions relax and the recognizing machine becomes more powerful. The hierarchy provides the fundamental classification of formal languages.

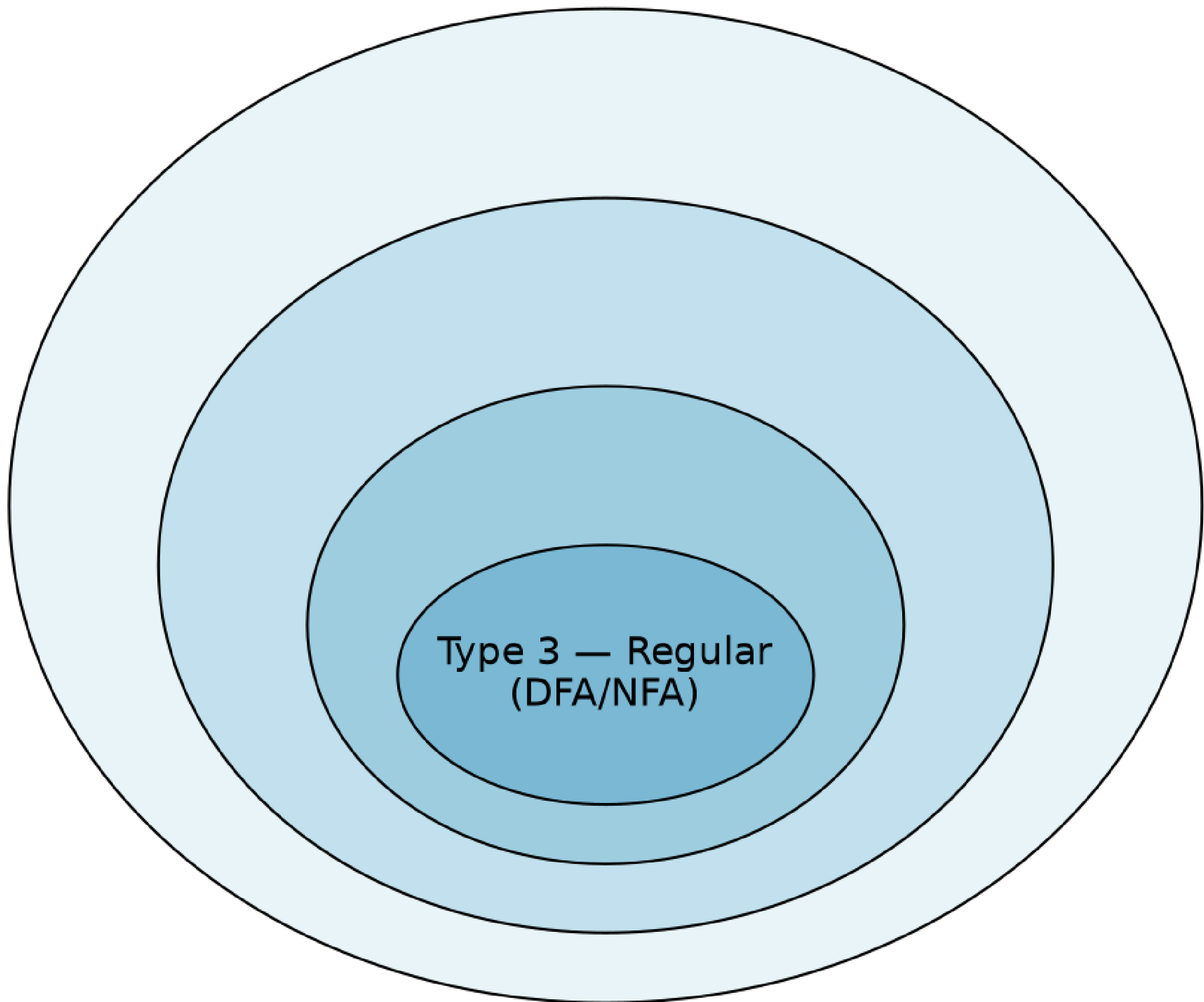


Figure 33: Chomsky Hierarchy nested visualization

Q8. Explain the working of Linear Bounded Automata with an example. (9 marks)

Definition: A Linear Bounded Automaton (LBA) is a restricted Turing Machine where the tape head cannot move outside the input portion of the tape. The tape is bounded by left and right endmarkers (⊢ and ⊣), so available memory is linear in input length.

Formal Definition: LBA = $(Q, \Sigma, \Gamma, \delta, q_0, b, F, \vdash, \dashv)$. Same as TM with the addition of endmarker symbols.

Structure and Working:

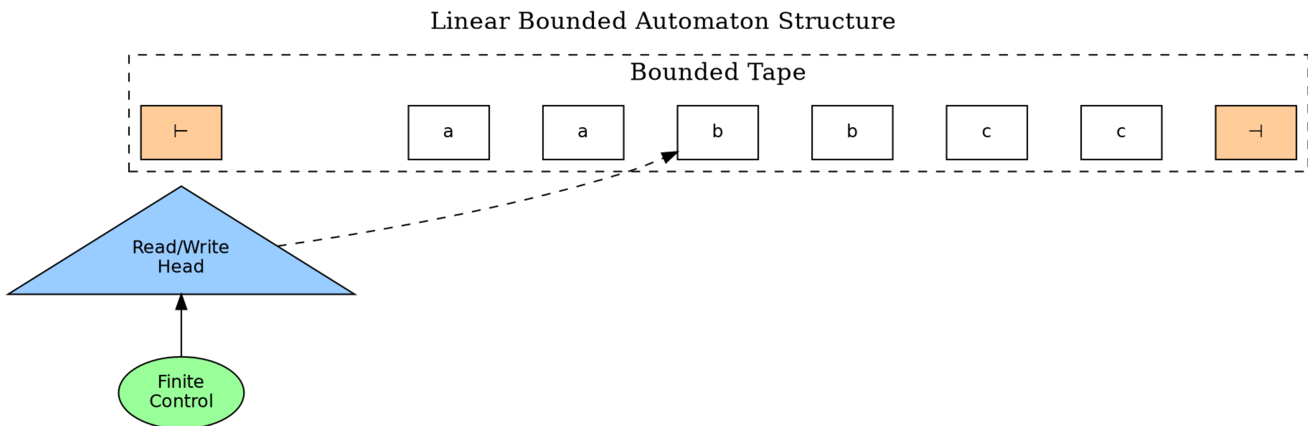


Figure 34: LBA Structure

The LBA reads input bracketed by ⊢ and ⊣. Like a TM, its head can read, write, and move left or right — but it cannot cross either endmarker.

Languages Accepted: Context-Sensitive Languages (Type 1 in Chomsky Hierarchy).

Example: L = $\{a^n b^n c^n \mid n \geq 1\}$

The LBA processes the input as follows:

1. Start at the leftmost cell.
2. Find the first unmarked a and mark it (e.g., as X).
3. Scan right past more a's and any Y's, find the first unmarked b, mark it (as Y).
4. Continue right past b's and Z's, find the first unmarked c, mark it (as Z).
5. Scan back left to the leftmost unmarked symbol.
6. Repeat. If at any point we expect a symbol that's missing, reject.
7. When all symbols are marked X, Y, or Z, verify there are no leftover unmarked input — accept.

This works within the bounded tape because the input has fixed length 3n.

Position in Hierarchy:

$$\text{Regular} \subset \text{CFL} \subset \text{CSL (LBA)} \subset \text{RE (TM)}$$

LBA is strictly more powerful than PDA and strictly less powerful than TM.

Conclusion: LBA is the machine model corresponding to context-sensitive languages and provides a key intermediate level in the Chomsky Hierarchy.

Section C (Long Answer — 14 marks each, attempt any 3 of 5)

Q9. Define DFA and design a DFA that accepts the binary number whose equivalent is divisible by 5. (14 marks)

Definition of DFA:

A Deterministic Finite Automaton is a 5-tuple $\mathbf{M = (Q, \Sigma, \delta, q_0, F)}$ where Q is a finite set of states, Σ is a finite input alphabet, $\delta : Q \times \Sigma \to Q$ is the transition function, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is the set of final states. A string w is **accepted** if, after starting in q_0 and reading w, the DFA ends in a state in F. DFAs accept exactly the class of regular languages.

Designing the DFA for binary divisibility by 5:

Insight: Read the binary number left to right. Each new bit doubles the current value and adds 0 or 1. We track only the current value modulo 5 — that gives 5 states.

Update rule: If current remainder is r and we read bit b, the new remainder is **(2r + b) mod 5**.

States: q_i = "current value ≡ i (mod 5)" for i = 0, 1, 2, 3, 4.

- q_0 is both initial AND final (since 0 mod 5 = 0).

Transition Table:

State (rem)	On 0 → (2r) mod 5	On 1 → (2r + 1) mod 5
*→ $\mathbf{q_0}$ (0)	q_0	q_1
q_1 (1)	q_2	q_3
q_2 (2)	q_4	q_0
q_3 (3)	q_1	q_2
q_4 (4)	q_3	q_4

State Diagram:

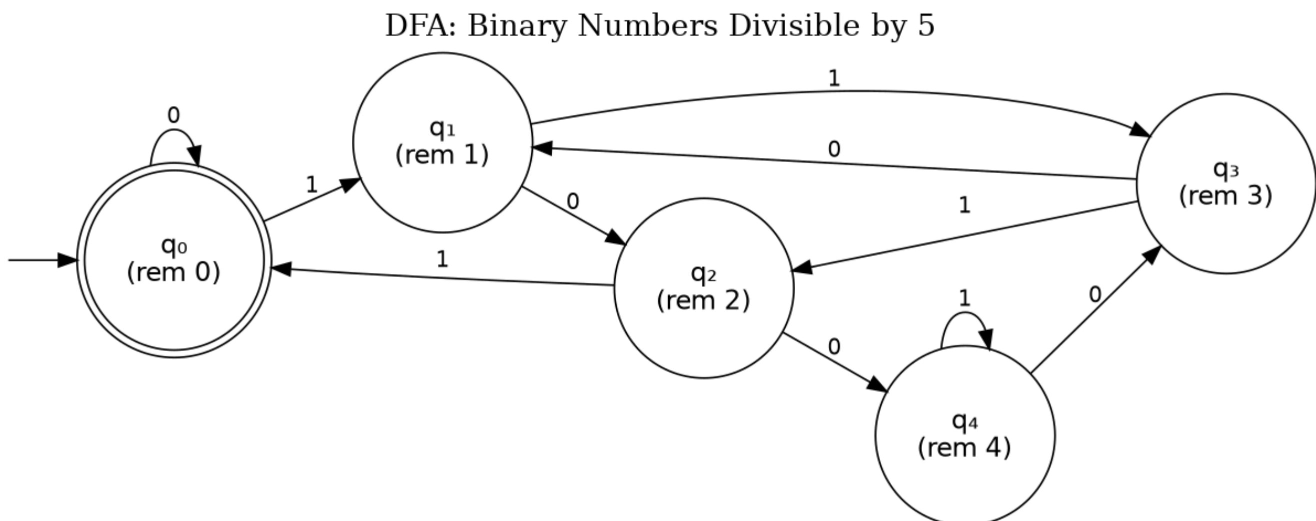


Figure 35: DFA accepting binary numbers divisible by 5

Formal Specification:

$$M = (\{q_0, q_1, q_2, q_3, q_4\}, \{0, 1\}, \delta, q_0, \{q_0\})$$

Verification:

- Input “101” = decimal 5 → q_0 → q_1 → q_2 → q_0 ✓ accepted (5 is divisible by 5).
- Input “1010” = decimal 10 → q_0 → q_1 → q_2 → q_0 → q_0 ✓ accepted.
- Input “1111” = decimal 15 → q_0 → q_1 → q_3 → q_2 → q_0 ✓ accepted.
- Input “111” = decimal 7 → q_0 → q_1 → q_3 → q_2 ✗ rejected (7 not divisible).
- Input “0” = decimal 0 → q_0 → q_0 ✓ accepted (0 is divisible by 5).

Conclusion: The DFA M correctly accepts exactly the binary representations of natural numbers divisible by 5.

Q10. Describe Mealy and Moore machines with example. Convert the given Mealy machine into Moore machine. (14 marks)

Mealy and Moore Machine Descriptions:

A **Moore machine** is a finite state machine with output, where the output depends only on the current state. Formally a 6-tuple $(Q, \Sigma, \Delta, \delta, \lambda, q_0)$ with $\boldsymbol{\lambda : Q \rightarrow \Delta}$. Outputs are placed on states.

A **Mealy machine** is a finite state machine where output depends on both the current state and the current input. Formally a 6-tuple $(Q, \Sigma, \Delta, \delta, \lambda, q_0)$ with $\boldsymbol{\lambda : Q \times \Sigma \rightarrow \Delta}$. Outputs are placed on transitions.

Comparison:

Aspect	Mealy	Moore
Output function	$\lambda: Q \times \Sigma \rightarrow \Delta$	$\lambda: Q \rightarrow \Delta$
Output depends on	State + Input	State only
Output appears on	Edges	Nodes
# states	Usually fewer	Usually more
Output for empty input	Not defined	Defined
Power	Equivalent	Equivalent

Conversion Procedure (Mealy → Moore) — 4 Steps:

1. For each state in the Mealy machine, list distinct outputs on incoming transitions.
2. If a state has only one distinct incoming output, keep it as-is and assign that output.
3. If a state has multiple distinct incoming outputs, **split it** into copies (one per output value).
4. Redirect transitions to the appropriate copy based on the output that was on the edge. Remove output labels from edges.

Worked Example with the given Mealy Machine:

The diagram in the question shows a Mealy machine with 3 states q_1, q_2, q_3 over input {0, 1} with outputs $\{z_1, z_2\}$. Visible transitions (reconstructed from typical structure):

- $q_1 \rightarrow q_2$ on $0/z_1$
- $q_1 \rightarrow q_3$ on $1/z_1$
- $q_2 \rightarrow q_2$ on $0/z_2$ (self-loop)
- $q_2 \rightarrow q_3$ on $1/z_1$
- $q_3 \rightarrow q_1$ on $1/z_1$
- $q_3 \rightarrow q_3$ on $0/z_2$ (self-loop)

Step 1 — Inventory of incoming outputs per state:

- q_1: incoming from q_3 on $1/z_1$ → output set = $\{z_1\}$ (single)
- q_2: incoming from q_1 on $0/z_1$ AND from q_2 on $0/z_2$ → output set = $\{z_1, z_2\}$ **must split**
- q_3: incoming from q_1 on $1/z_1$, from q_2 on $1/z_1$, from q_3 on $0/z_2$ → output set = $\{z_1, z_2\}$ **must split**

Step 2 — Build Moore states:

- q_1 → keep as $\mathbf{q_1}$ with output z_1

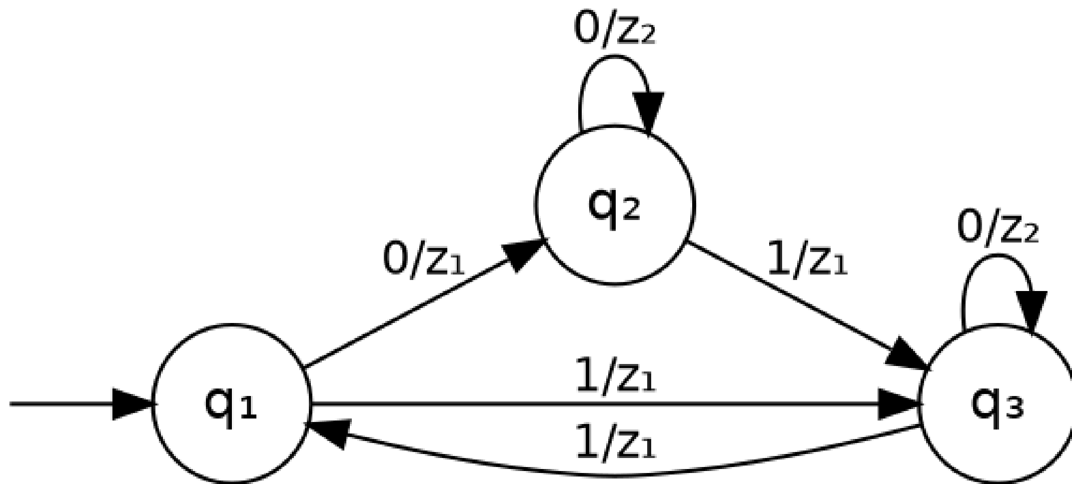


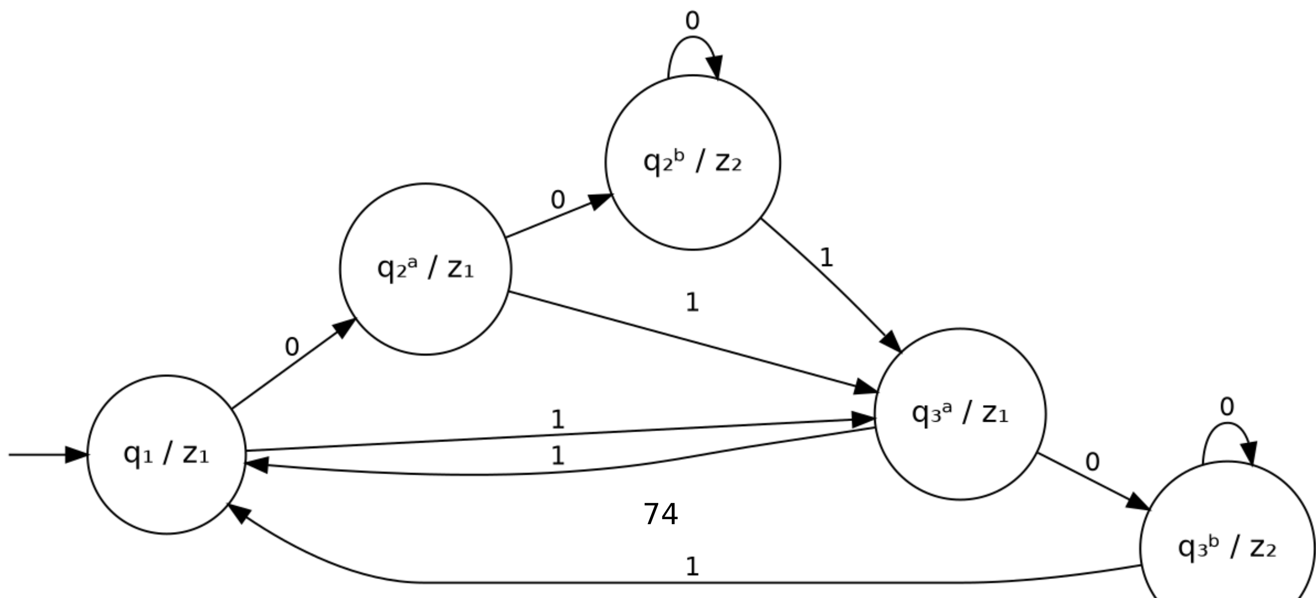
Figure 36: Mealy Machine from 2025 paper

- q_2 → split into $\mathbf{q_2^a}$ (output z_1) and $\mathbf{q_2^b}$ (output z_2)
- q_3 → split into $\mathbf{q_3^a}$ (output z_1) and $\mathbf{q_3^b}$ (output z_2)

Step 3 — Redirect transitions:

Moore state	Output	On 0	On 1
→ $\mathbf{q_1}$	z_1	q_2^a (since $q_1 \to q_2$ had output z_1)	q_3^a (since $q_1 \to q_3$ had output z_1)
q_2^a	z_1	q_2^b ($q_2 \to q_2$ has output z_2)	q_3^a ($q_2 \to q_3$ has output z_1)
q_2^b	z_2	q_2^b	q_3^a
q_3^a	z_1	q_3^b ($q_3 \to q_3$ output z_2)	q_1 ($q_3 \to q_1$ output z_1)
q_3^b	z_2	q_3^b	q_1

Resulting Moore Machine:



Q11. Explain the following: (a) ε-transition (b) Kleene's Theorem (c) Post Correspondence Problem. (14 marks)

Part (a) — ε-transition (5 marks) An **ε-transition** is a state transition in a finite automaton that occurs **without consuming any input symbol**. It is denoted δ(q, ε) and lets the automaton move spontaneously from one state to another.

Where used: ε-transitions appear in **ε-NFA** (also called NFA with ε-moves), which is commonly used as an intermediate step when converting a regular expression to a finite automaton via Thompson's construction.

ε-closure of a state q: The set of all states reachable from q via zero or more ε-transitions, including q itself. Used during conversion of ε-NFA to NFA / DFA.

Example:

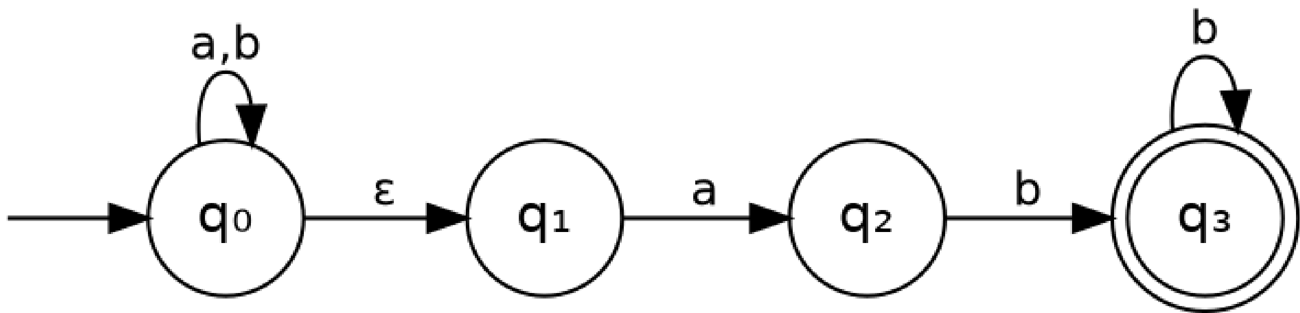


Figure 38: ε-NFA Example

Power: ε-transitions do **not** add computational power. Every ε-NFA can be converted to an equivalent NFA without ε-transitions, which can further be converted to a DFA. All three machines (DFA, NFA, ε-NFA) accept exactly the regular languages.

Part (b) — Kleene's Theorem (5 marks) **Statement:** Kleene's Theorem establishes the equivalence between regular expressions and finite automata. It has two parts:

1. **(RE → FA)** Every language denoted by a regular expression is accepted by some finite automaton.
2. **(FA → RE)** Every language accepted by a finite automaton can be expressed as a regular expression.

Implication: A language L is regular if and only if it can be described by a regular expression — equivalently, accepted by a DFA, NFA, or ε-NFA.

Tools for the conversions:

- **RE → FA: Thompson's construction algorithm** builds an ε-NFA recursively from the RE structure. Base cases for individual symbols and ε; recursive cases for union, concatenation, and Kleene star.
- **FA → RE: Arden's theorem** (algebraic method): write one equation per state and solve.

Significance: Kleene's theorem unifies two completely different ways of describing regular languages — algebraic (regular expressions) and operational (finite automata) — proving they describe exactly the same class.

Part (c) — Post Correspondence Problem (4 marks) The **Post Correspondence Problem (PCP)**, introduced by Emil Post in 1946, is defined as follows:

Given two finite lists $A = (a_1, a_2, \ldots, a_n)$ and $B = (b_1, b_2, \ldots, b_n)$ of strings over a common alphabet, determine whether there exists a sequence of indices $i_1, i_2, \ldots, i_k$ ($k \geq 1$, repetition allowed) such that:

$$a_{i_1} a_{i_2} \cdots a_{i_k} = b_{i_1} b_{i_2} \cdots b_{i_k}$$

Example: A = {1, 10, 011}, B = {101, 00, 11}. Various sequences could be tried; finding a solution (or proving none exists) is the challenge.

Key Property: PCP is **undecidable** — no algorithm exists that can determine for every PCP instance whether a solution exists.

Significance: PCP serves as a "starting point" undecidable problem widely used to prove undecidability of other problems via reduction (e.g., CFG ambiguity, CFG equivalence).

Q12. What is Turing machine? Design a Turing machine for the language $L = \{a^n b^n c^n \mid n \geq 1\}$. (14 marks)

What is a Turing Machine?

A Turing Machine is the most powerful theoretical computational model. It consists of an infinite two-way tape divided into cells, a read/write head, and a finite control unit. The TM reads a symbol from the current cell, can replace it with another symbol, and moves the head one cell left or right.

Formal Definition: A 7-tuple $TM = (Q, \Sigma, \Gamma, \delta, q_0, b, F)$ where:

- Q = finite set of states
- Σ = input alphabet
- Γ = tape alphabet ($\Sigma \subseteq \Gamma$)
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ = transition function
- q_0 = initial state
- $b \in \Gamma \ \Sigma$ = blank symbol
- $F \subseteq Q$ = set of final states

A string is accepted if, starting in q_0, the TM eventually halts in a state in F.

TM Design for $L = \{a^n b^n c^n \mid n \geq 1\}$:

Why this language needs a TM: L is not context-free. A PDA with one stack cannot count three quantities simultaneously. Only a TM (with random-access tape) can do this.

Strategy: Each pass through the tape, mark one a as X, one b as Y, and one c as Z. Repeat until all symbols are marked. Then verify and accept.

States:

- q_0 — find next unmarked a
- q_1 — found a, scanning right for b
- q_2 — found b, scanning right for c
- q_3 — found c, scanning back left
- q_4 — verification pass
- q_5 — accept (final)

Transition Table:

State	a	b	c	X	Y	Z	B
q_0	(q_1, X, R)	—	—	—	(q_4, Y, R)	—	—
q_1	(q_1, a, R)	(q_2, Y, R)	—	—	(q_1, Y, R)	—	—
q_2	—	(q_2, b, R)	(q_3, Z, L)	—	—	(q_2, Z, R)	—
q_3	(q_3, a, L)	(q_3, b, L)	—	(q_0, X, R)	(q_3, Y, L)	(q_3, Z, L)	—
q_4	—	—	—	—	(q_4, Y, R)	(q_4, Z, R)	(q_5, B, R)
***q_5**	—	—	—	—	—	—	—

State Diagram:

Trace for "aabbcc":

- Pass 1: replace first a → X, first b → Y, first c → Z. Tape: XaYbZc. Scan back left.
- Pass 2: replace remaining a → X, b → Y, c → Z. Tape: XXYYZZ. Scan back left.

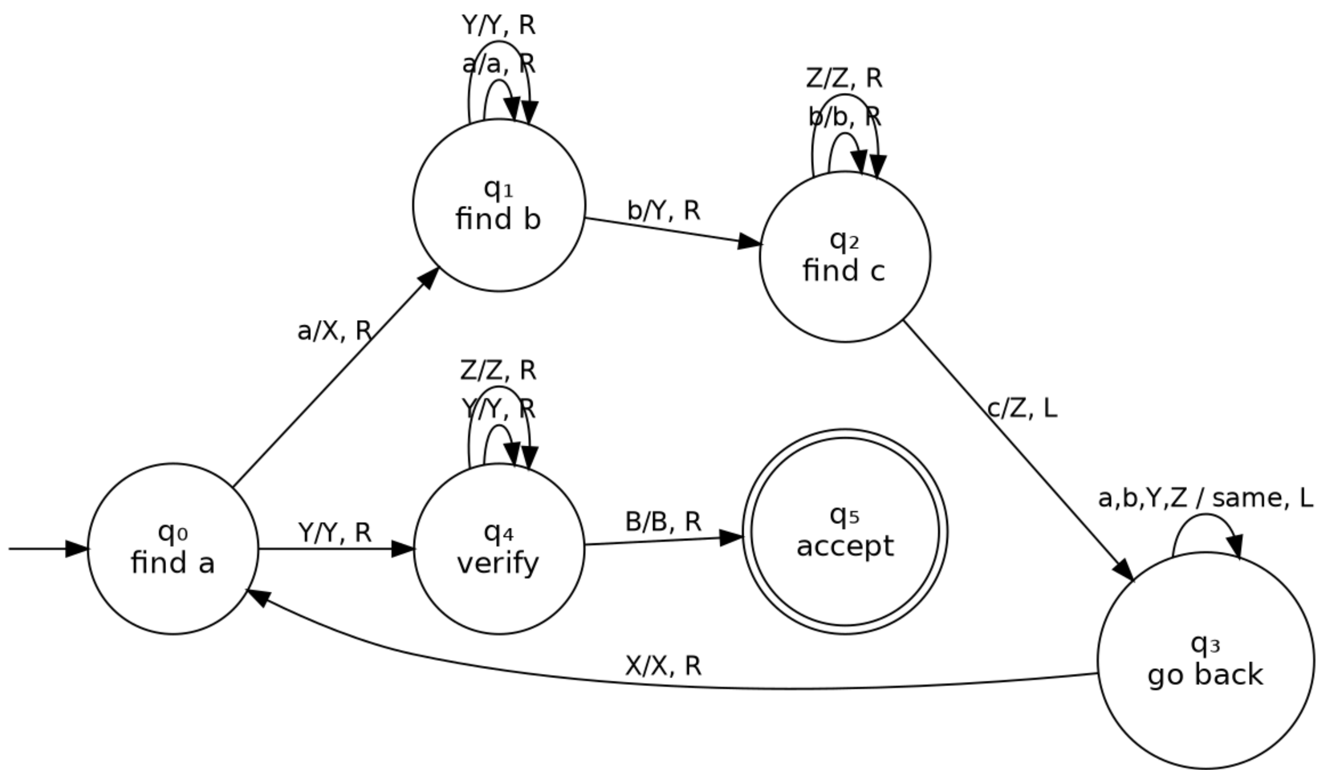


Figure 39: TM for $L = \{a^n b^n c^n \mid n \geq 1\}$

- From q_0, read Y → enter q_4 (verification mode).
- From q_4, scan past Y, Y, Z, Z. Read blank B → ACCEPT in q_5.

Conclusion: The TM $M = (Q, \{a,b,c\}, \{a,b,c,X,Y,Z,B\}, \delta, q_0, B, \{q_5\})$ accepts exactly $L = \{a^n b^n c^n \mid n \geq 1\}$. This language is not context-free, demonstrating that the Turing Machine is strictly more powerful than the Pushdown Automaton.

Q13. Explain the following: (a) Deterministic Pushdown Automata (b) Derivation Trees (c) Decision Problems of CFL. (14 marks)

Part (a) — Deterministic Pushdown Automata (5 marks) A **Deterministic Pushdown Automaton (DPDA)** is a PDA where, at each configuration (state, input symbol, stack-top), there is at most ONE possible move.

Formal Definition: DPDA = $(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ where the transition function δ is restricted such that:

- For any $q \in Q$ and $Z \in \Gamma$, if $\delta(q, \varepsilon, Z)$ is defined, then $\delta(q, a, Z)$ must be undefined for all $a \in \Sigma$.
- For any q, a, Z, the result $\delta(q, a, Z)$ contains at most one (state, stack-string) pair.

Languages Accepted: Deterministic Context-Free Languages (DCFLs).

Power Comparison: DCFLs are a **proper subset** of CFLs:

$$\text{DCFL} \subsetneq \text{CFL}$$

Example languages:

- $\{a^n b^n \mid n \geq 1\}$ is a DCFL — accepted by DPDA by pushing a's and popping for b's.
- $\{ww^R \mid w \in \{a,b\}^*\}$ (palindromes) is a CFL but NOT a DCFL — requires non-deterministic guessing of the midpoint.

This shows that **NPDA is strictly more powerful than DPDA**.

Real-world significance: DCFLs are particularly important because deterministic parsing is efficient. Most programming language syntax is designed to be recognized by DPDAs.

Part (b) — Derivation Trees (5 marks) A **Derivation Tree** (also called a parse tree) is a tree representation of how a CFG derives a string from its start symbol.

Construction Rules:

- The **root** is labeled with the start symbol S.
- **Internal nodes** are labeled with non-terminal variables.
- **Leaf nodes** are labeled with terminal symbols (or ε).
- For each internal node A with children $X_1, X_2, \ldots, X_n$, there is a production $A \rightarrow X_1X_2\ldots X_n$ in the grammar.
- Reading the leaves left-to-right gives the **yield** of the tree, which is the derived string.

Example: Grammar $S \rightarrow aSb \mid ab$. Parse tree for "aabb":

Reading leaves: a, a, b, b → "aabb" ✓

Leftmost vs Rightmost Derivation: A derivation tree can be read in two ways:

- **Leftmost derivation (LMD):** at each step, replace the leftmost non-terminal.
- **Rightmost derivation (RMD):** at each step, replace the rightmost non-terminal.

Both yield the same parse tree for an unambiguous grammar.

Ambiguity: A grammar is **ambiguous** if some string has two distinct parse trees (equivalently, two distinct LMDs). Determining ambiguity of an arbitrary CFG is undecidable.

Derivation Tree for "aabb" using S → aSb | ab

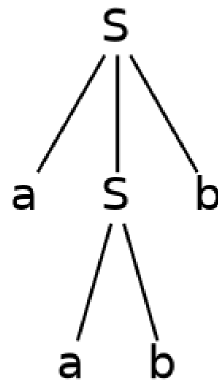


Figure 40: Parse tree for aabb

Part (c) — Decision Problems of CFL (4 marks) **Decidable Problems for CFL:**

- **Membership:** Given CFG G and string w, is w ∈ L(G)? Decidable using the **CYK algorithm** in $O(n^3)$ time after converting G to CNF.
- **Emptiness:** Is L(G) = ∅? Decidable by checking if S is generating.
- **Finiteness / Infiniteness:** Is L(G) finite/infinite? Decidable by checking for cycles in the variable dependency graph.

Undecidable Problems for CFL:

- **Equivalence:** Given two CFGs G_1 and G_2, is $L(G_1) = L(G_2)$?
- **Ambiguity:** Is a given CFG G ambiguous?
- **Inherent Ambiguity:** Is L(G) inherently ambiguous (i.e., has no unambiguous grammar)?
- **Σ* test:** Is L(G) = Σ*?
- **Intersection emptiness:** Is $L(G_1) \cap L(G_2) = \emptyset$?

Reason for Undecidability: Most undecidability results for CFL are proved by **reduction from the Post Correspondence Problem (PCP)**, which is itself undecidable. Since PCP can be encoded as a CFG question, decidability of these CFG questions would imply decidability of PCP, which is impossible.

Problem	Decidable for CFL?
Membership	✓
Emptiness	✓
Finiteness	✓
Equivalence	✗
Ambiguity	✗
Σ* test	✗

Final Quick Reference Card

All Tuples in One Place

Machine	Tuple	Distinguishing δ
DFA	$(Q, \Sigma, \delta, q_0, F)$	$\delta: Q \times \Sigma \rightarrow Q$
NFA	$(Q, \Sigma, \delta, q_0, F)$	$\delta: Q \times \Sigma \rightarrow 2^{\wedge}Q$
ε-NFA	$(Q, \Sigma, \delta, q_0, F)$	$\delta: Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow 2^{\wedge}Q$
Moore	$(Q, \Sigma, \Delta, \delta, \lambda, q_0)$	$\lambda: Q \rightarrow \Delta$
Mealy	$(Q, \Sigma, \Delta, \delta, \lambda, q_0)$	$\lambda: Q \times \Sigma \rightarrow \Delta$
PDA	$(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$	$\delta: Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \rightarrow 2^{\wedge}(Q \times \Gamma^*)$
TM	$(Q, \Sigma, \Gamma, \delta, q_0, b, F)$	$\delta: Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$
LBA	TM with $\vdash$, $\dashv$	bounded tape

Power Hierarchy

$$\text{DFA} = \text{NFA} = \varepsilon\text{-NFA} \subsetneq \text{DPDA} \subsetneq \text{NPDA} \subsetneq \text{LBA} \subsetneq \text{TM}$$

$$\text{Regular} \subsetneq \text{DCFL} \subsetneq \text{CFL} \subsetneq \text{CSL} \subsetneq \text{RE}$$

Theorems One-Liners

- **Rabin-Scott:** NFA ≡ DFA (n states → up to 2^n states).
- **Kleene's:** RE ↔ FA (regular expressions equivalent to finite automata).
- **Arden's:** $R = Q + RP$ has unique solution $R = QP^*$ (when $\varepsilon \notin P$).
- **Pumping Lemma:** $\exists n, \forall z \in L$ $(|z| \geq n)$, $z = uvw$, $|uv| \leq n$, $|v| \geq 1$, $\forall i$: $uv^iw \in L$.
- **Church-Turing:** Algorithmically computable = Turing-computable.

Famous Undecidable Problems

- Halting Problem
- Post Correspondence Problem
- CFG Ambiguity
- CFG Equivalence
- TM Equivalence

Section A Definition Templates

Term	One-line answer
Automata	5-tuple $(Q, \Sigma, \delta, q_0, F)$; abstract self-operating model with states, alphabet, transition, initial and final states
DFA	FA where $\delta: Q \times \Sigma \rightarrow Q$ gives one next state; accepts regular languages

Term	One-line answer
NFA	FA where $\delta: Q \times \Sigma \to 2^Q$ gives a set of states; allows ε-moves; equivalent to DFA
Grammar	G = (V, T, P, S); formal system generating a language via productions
CFG	Grammar with all productions of form $A \to \alpha$ (single non-terminal LHS)
Derivation tree	Tree showing how grammar derives a string; root = S, leaves = terminals
CNF	Productions of form $A \to BC$ or $A \to a$
GNF	Productions of form $A \to a\alpha$
Regular expression	Algebraic expression using +, ·, * to denote regular languages
Pumping Lemma	Tool to prove non-regularity by pumping a middle section
Turing Machine	7-tuple model with infinite read/write tape; most powerful
LBA	TM with tape bounded by input length; accepts CSL
Halting Problem	Undecidable: decide if TM M halts on input w
PCP	Undecidable string-matching problem
Recursive	Language with TM that always halts (decidable)
RE language	Language with TM accepting members; may loop on non-members
Church-Turing thesis	Computable = Turing-computable

Top 7 Mistakes to Avoid in Exam

1. **Read the question's verb carefully.** "Are NFAs more powerful?" — they aren't (equal power).
2. **Tuple sizes:** DFA = 5-tuple; Mealy/Moore = 6-tuple; PDA / TM = 7-tuple. **Don't confuse them.**
3. **TM transition** has THREE components on RHS: (state, symbol, direction L or R). Not two.
4. **CNF** allows only A → BC or A → a — **not** A → aB or A → ε (except for S → ε if needed).
5. **Pumping lemma is for proving NON-regular**, not regular.
6. **In Mealy → Moore conversion**, split states based on **incoming** outputs, not outgoing.
7. Always include the **closing line:** "Both machines have equal power and accept the same class of languages X." — earns marks.

You're Ready

Get a good night's sleep — sleep cements memory more than another hour of revision. In the morning, do a 30-minute glance through this document over chai, then walk in confident.

Good luck. □