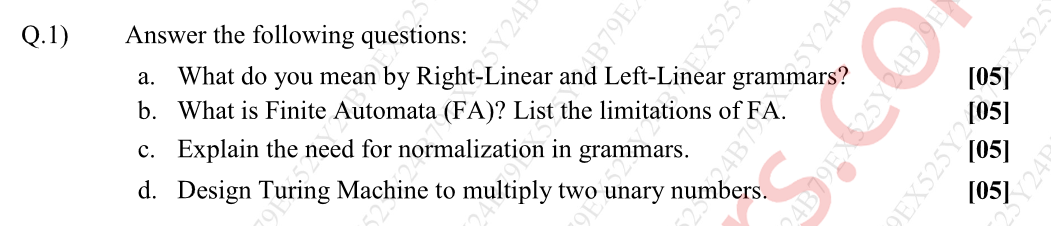
****

**Grammar**

**Definition**:  
A set of production rules to generate strings in a language. Defined as G = (V, Σ, P, S), where:

* V: Set of Non-terminal symbols.
* Σ: Set of Terminal symbols.
* P: Set of rules.
* S: Start symbol.

**Right Linear Regular Grammar**

In this type of regular grammar, all the non-terminals on the right-hand side exist at the rightmost place, or at the right ends.

A ⇢ a, A ⇢ a**B**, A ⇢ ∈  
where,  
A and B are non-terminals,  
a is terminal, and  
∈ is empty string

S ⇢ 00**B** | 11**S**  
B ⇢ 0**B** | 1**B** | 0 | 1  
where,  
S and B are non-terminals, and  
0 and 1 are terminals

**Left Linear Regular Grammar**

In this type of regular grammar, all the non-terminals on the left-hand side exist at the leftmost place, or at the left ends.

A ⇢ a, A ⇢ Ba, A ⇢ ∈  
where,  
A and B are non-terminals,  
a is terminal, and  
∈ is empty string

S ⇢ B00 | S11  
B ⇢ B0 | B1 | 0 | 1  
where  
S and B are non-terminals, and  
0 and 1 are terminals

DFA (Deterministic Finite Automaton)

**Definition**: A finite automaton with **deterministic transitions**; accepts regular languages.

**Tuple** T=(Q,Σ,δ,q0,F):

|  |  |
| --- | --- |
| **Tuple Variable** | **Represents** |
| Q | Finite set of **states**. |
| Σ | Finite **input alphabet**. |
| δ | **Transition function**: Q×Σ→Q. |
| q0​ | **Initial state** (q0∈Q0​). |
| F | Set of **final/accepting states** (F⊆Q. |

**Limitations FA**

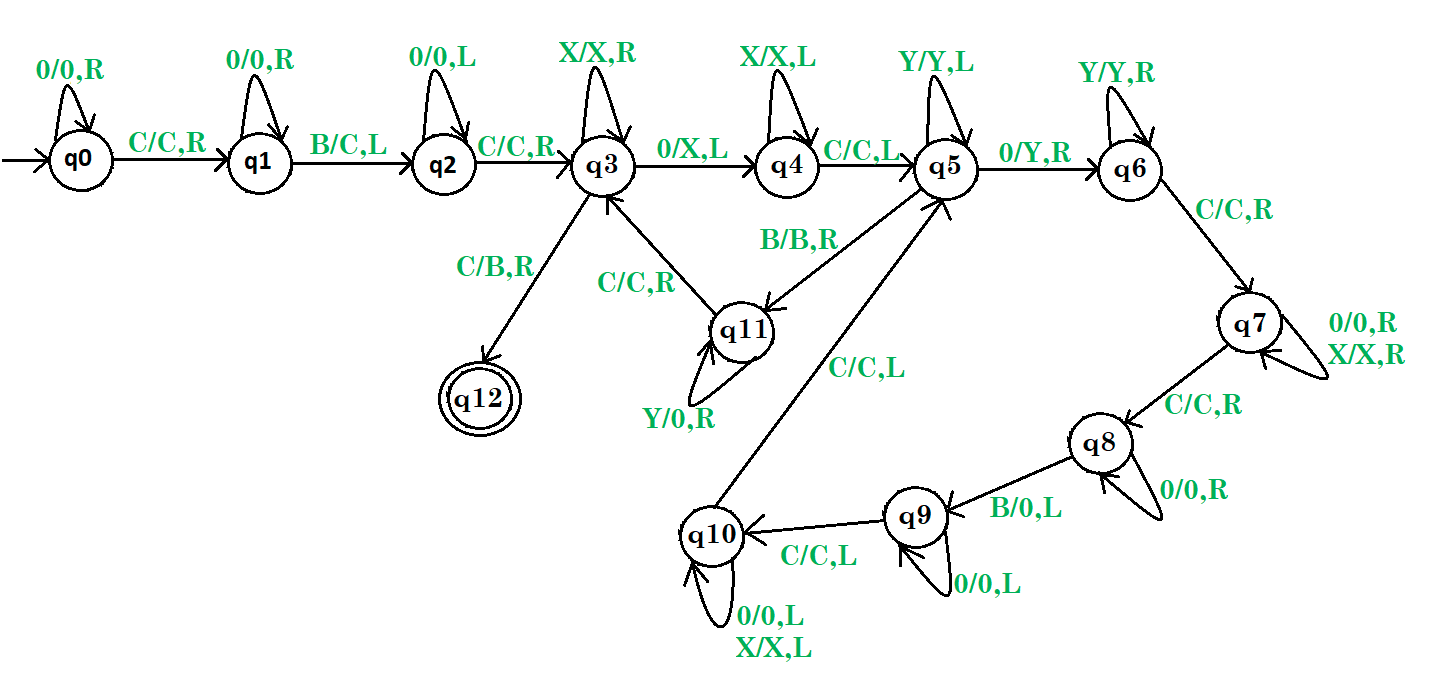
* Limited Memory
* Cannot Handle Context-Free or Higher Languages
* No Stack or External Memory

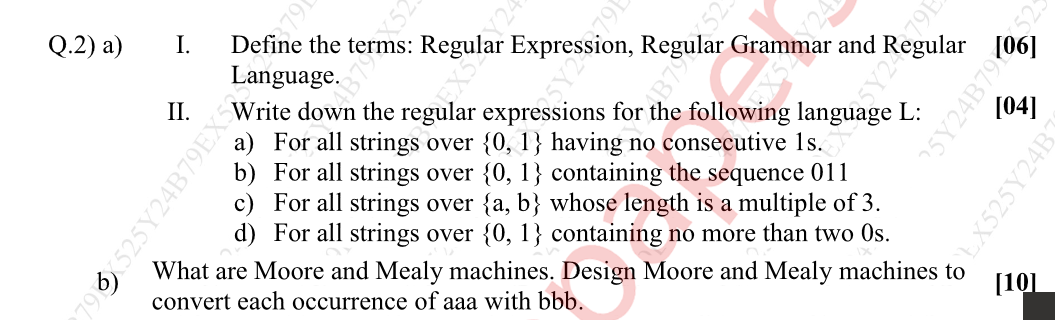
Need for Normalization in Grammars

Normalization in grammars refers to converting a given grammar into a standardized form (like Chomsky Normal Form or Greibach Normal Form) to simplify analysis, parsing, and theoretical proofs

* **Remove Null Productions**: Eliminating ε-productions simplifies grammars by removing empty string derivations, ensuring all rules generate visible symbols.
* **Remove Useless/Unreachable Productions**: Deleting non-terminals that don’t contribute to valid strings optimizes grammars, reducing unnecessary complexity.
* **Eliminate Ambiguity & Redundant Rules**: Normalization removes multiple interpretations and duplicate rules, ensuring consistent language derivation.
* **Simplifies Parsing and Analysis**: Structured grammar forms make parsing algorithms faster and more predictable by enforcing rule constraints.
* **Enables Efficient Algorithms**: Normalized grammars allow optimized parsing techniques like CYK, reducing computational overhead.
* **Helps in Formal Proofs**: Standardized forms support theoretical analysis, such as proving language properties via the Pumping Lemma.
* **Easier Compiler Implementation**: Clean grammar rules integrate smoothly with lexers and parsers, improving compiler design efficiency.

Design Turing Machine to multiply two unary numbers





Regular Expression (Regex)  
A notation used to describe patterns in strings, defined over an alphabet Σ, using operators like concatenation (·), union (|), and Kleene star (\*).

Regular Grammar  
A right-linear or left-linear grammar where all productions are of the form:

A → aB (right-linear)

A → Ba (left-linear)

A → a or A → ε (terminating rule).

Regular Language  
A language that can be recognized by a DFA/NFA or generated by a regular grammar or expressed by a regular expression.

a) All strings over {0, 1} with no consecutive 1's

**Regex:** (0 + 10)\* (1 + ε)

b) All strings over {0, 1} containing the sequence "011"

**Regex:** (0 + 1)\* 011 (0 + 1)\*

c) All strings over {a, b} with length divisible by 3

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

**Explanation:**

d) All strings over {0, 1} with at most two 0's

**Regex:** 1\* (0 + ε) 1\* (0 + ε) 1\* (0 + ε) 1\*

**Moore Machine**

**Definition**: A finite automaton where **outputs depend on the current state**.

**Tuple** T=(Q,Σ,Δ,δ,λ,q0)T=(Q,Σ,Δ,δ,λ,q0​):

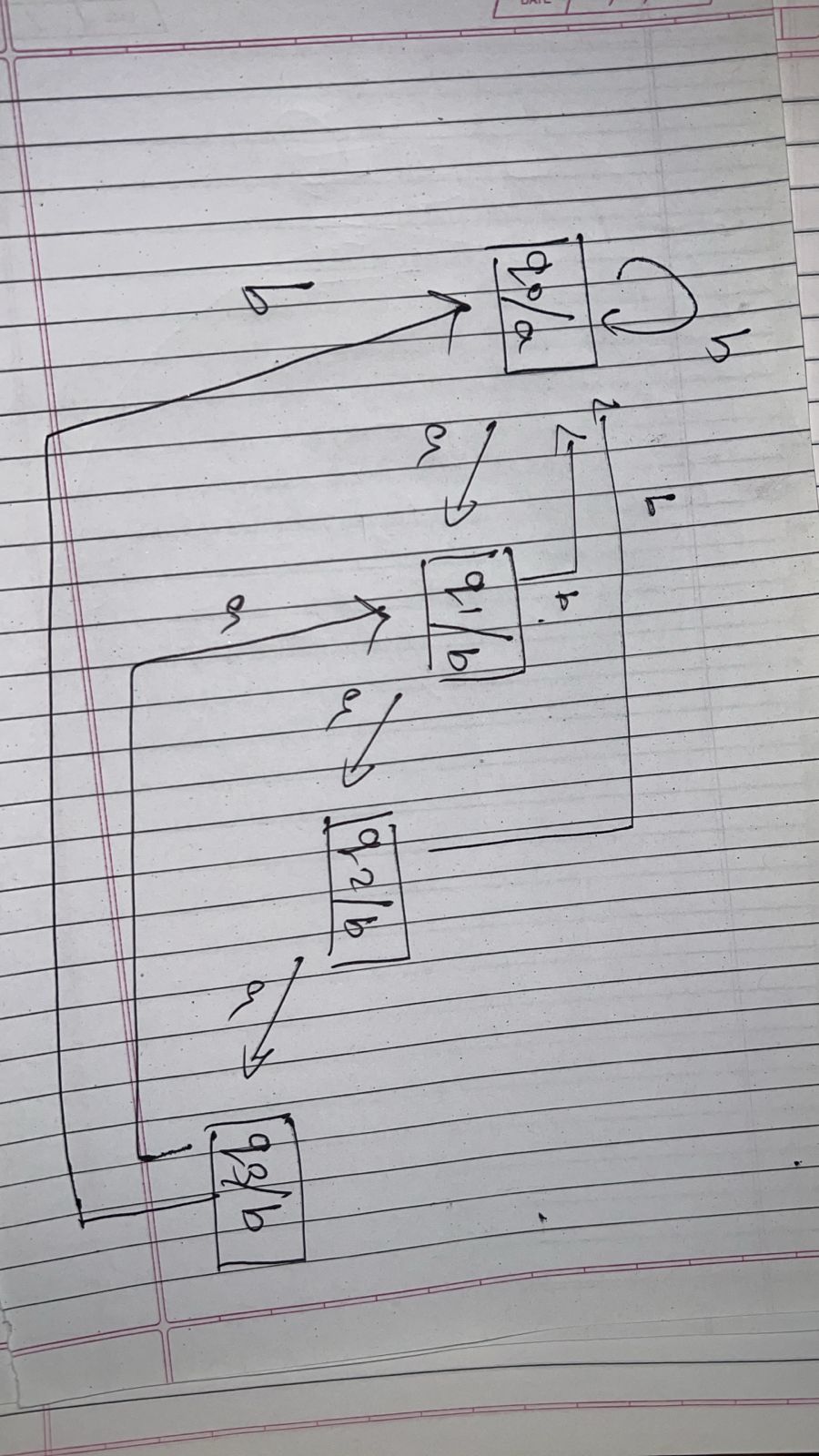
|  |  |
| --- | --- |
| **Tuple Variable** | **Represents** |
| Q | Finite set of **states**. |
| Σ | Finite **input alphabet**. |
| Δ | Finite **output alphabet**. |
| δ | **Transition function**: Q×Σ→Q. |
| λ | **Output function**: Q→Δ(output tied to state). |
| q0​ | **Initial state** (q0∈Q). |

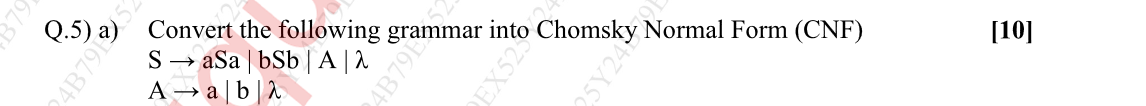
**Mealy Machine**

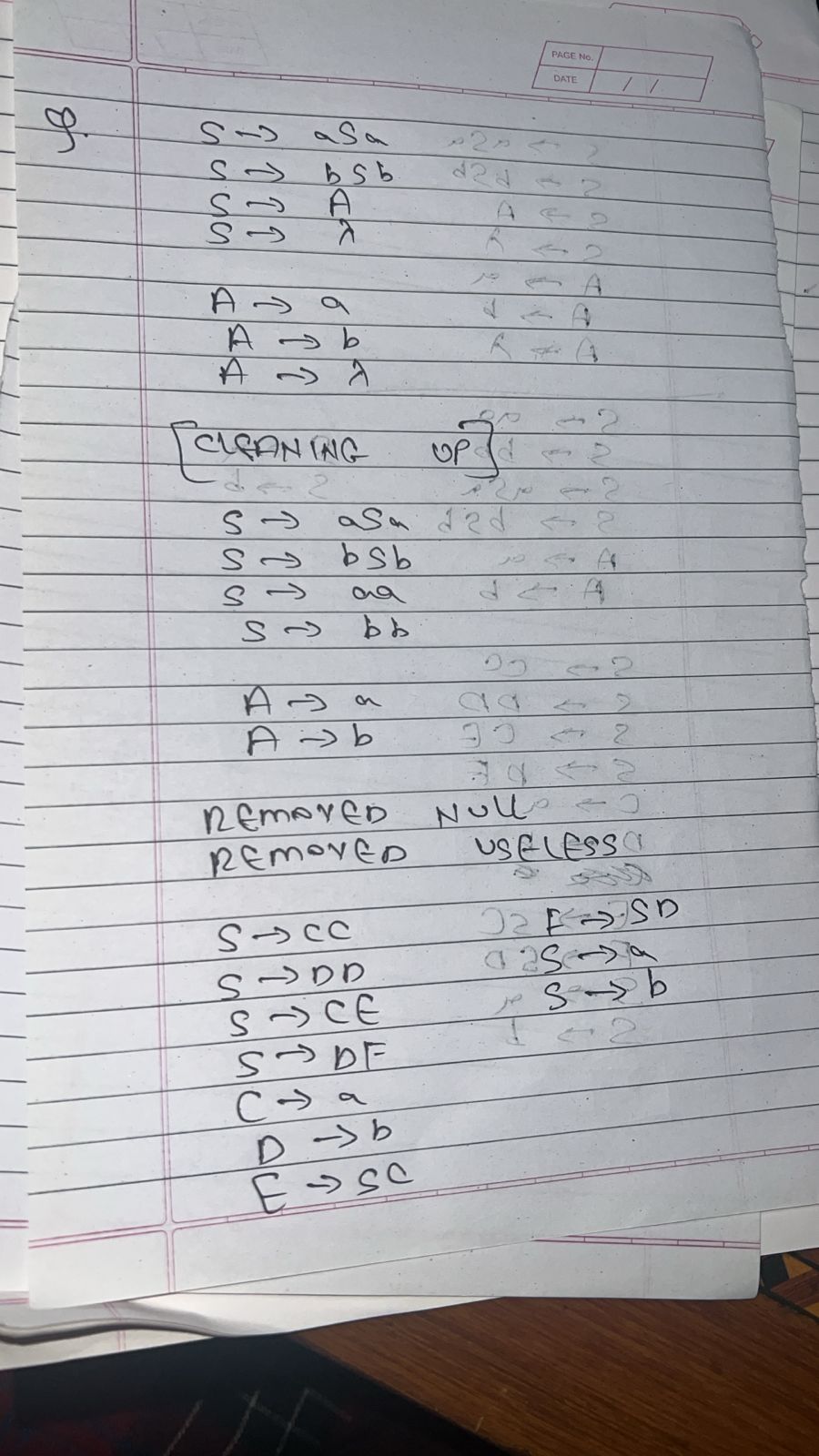
**Definition**: A finite automaton where **outputs depend on transitions** (current state + input).

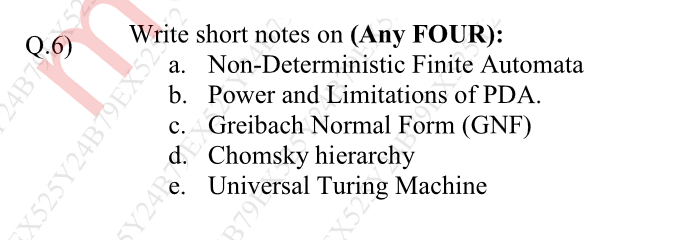
**Tuple** T=(Q,Σ,Δ,δ,λ,q0):

|  |  |
| --- | --- |
| **Tuple Variable** | **Represents** |
| Q | Finite set of **states**. |
| Σ | Finite **input alphabet**. |
| Δ | Finite **output alphabet**. |
| δ | **Transition function**: Q×Σ→Q. |
| λ | **Output function**: Q×Σ→Δ. |
| q0​ | **Initial state** (q0∈Q). |

Design Moore and Mealy machines to convert each occurrence of aaa with bbb.







Non-Deterministic Finite Automata (NFA)

NFAs extend finite automata by allowing multiple transitions from a state on the same input symbol, including ε-transitions (moves without consuming input). While seemingly more powerful than DFAs, NFAs recognize exactly the same class of languages—regular languages—but often provide a more intuitive representation. Their non-determinism is theoretical; in practice, NFAs are converted to equivalent DFAs via the powerset construction, which may exponentially increase states. NFAs are particularly useful in regex engines and lexical analysis, where their compact representation aids pattern matching.

Example:

Regex a(b|c)\* can be represented as an NFA with:

A start state transitioning to q1 on a.

q1 looping on b or c non-deterministically.

Applications:

Lexical Analysis: NFAs are used in compilers (e.g., Flex) to tokenize input.

Pattern Matching: Regex engines convert expressions to NFAs for efficient search.

Power and Limitations of Pushdown Automata (PDA)

PDAs enhance finite automata with a stack, enabling recognition of context-free languages (CFLs) like balanced parentheses (aⁿbⁿ) and nested structures. This makes them indispensable for parsing programming languages. However, PDAs are limited by their single-stack memory: they cannot handle languages requiring cross-serial dependencies (e.g., aⁿbⁿcⁿ) or parallel tracking (e.g., palindromes with a center marker). These limitations place PDAs below Turing machines in the Chomsky hierarchy, as they cannot solve problems requiring arbitrary memory access.

Example:

Language {aⁿbⁿ | n ≥ 1} is recognized by a PDA that:

Pushes a onto the stack for each a in the input.

Pops a for each b.

Accepts if the stack is empty at the end.

Applications:

Syntax Parsing: PDAs underpin parsers for programming languages (e.g., LALR parsers in Yacc).

Balanced Symbols: Checking nested structures like HTML/XML tags.

Greibach Normal Form (GNF)

Meko nahi pata

Universal Turing Machine (UTM)

A UTM is a Turing machine capable of simulating any other Turing machine by reading its description (program) and input. This theoretical construct underpins modern general-purpose computers, demonstrating that a single machine can perform all algorithmic computations. The UTM validates the Church-Turing thesis, which posits that any computable function can be computed by a Turing machine. Despite its simplicity, the UTM’s universality highlights the fundamental limits of computation, including undecidable problems like the Halting Problem.

Example:

A UTM can simulate a Turing machine for aⁿbⁿcⁿ by:

Reading the description of the machine and input (e.g., aaabbbccc).

Executing each step of the simulated machine on its tape.

Applications:

Modern Computers: The von Neumann architecture is a practical realization of UTM principles.

Undecidability Proofs: Demonstrates problems like the Halting Problem cannot be solved algorithmically.

