CBCS SCHEME

USN | | | | | | | | | | |

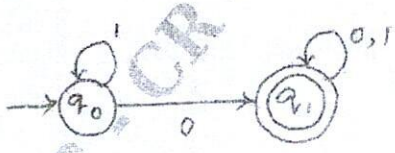
BCS503

Fifth Semester B.E./B.Tech. Degree Examination, Dec.2024/Jan.2025
Theory of Computation

Time: 3 hrs. Max. Marks: 100

Note: 1. Answer any FIVE full questions, choosing ONE full question from each module.
2. M : Marks , L: Bloom's level , C: Course outcomes.

		Module – 1	M	L	C
Q.1	a.	Define the following with example : i) Language ii) String iii) Power of an alphabet.	3	L1	CO1
	b.	Define DFA. Draw a DFA to accepts. i) The set of all strings that contain a substring aba. ii) To accept the stings of a's and b's that contain not more than there b's. iii) L = {w ∈ {a, b}* : No 2 consecutive characters are same in w}.	10	L3	CO1
	c.	Convert the following NFA to DFA. → p {p, q} {p} q {r} {r} r {s} ϕ * s {s} {s}	7	L2	CO1
		OR			
Q.2	a.	Define the following with example : i) Alphabet ii) Reversal of string iii) Concatenation of Languages.	3	L1	CO1
	b.	Design a DFA for the Language : L = {w ∈ {0, 1}* : w is a string divisible by 5}.	7	L3	CO1
	c.	Define NFA. Obtain an ε - NFA which accepts strings consisting of 0 or more a's , followed by 0 or more b's followed by 0 or more C's. Also convert it to DFA.	10	L2	CO1
		Module – 2			
Q.3	a.	Define Regular expression. Write the regular expression for the following languages : i) Strings of a's and b's starting with a and ending with b. ii) Set of strings that consists of alternating 0's and 1's. iii) L = {a^n bm , (n + m) is even}. iv) L = {w : / w / mod 3 = 0 , where w ∈ {a, b}*}.	10	L2	CO2

	b.	Minimize the following finite automata using Table filling algorithm : δ | a | b → A | B | A B | A | C C | D | B * D | D | A E | D | F F | G | E G | F | G H | G | D	10	L2	CO2
		OR			
Q.4	a.	Construct ε - NFA for the following Regular expression : i) $(0+1)\,0\,1(1+0)$ ii) $1(0+1)^*0$ iii) $(0+1)^*\,0\,1\,1^*$	6	L1	CO2
	b.	Obtain the Regular expression that denotes the language accepted by Fig. Q4(b). Fig. Q4(b)  Using Kleene's theorem.	6	L3	CO2
	c.	State the Pumping Lemma for the Regular Languages. And also prove that the following languages are note regular. i) $L = \{0^n\,1^m \mid n \le m\}$ ii) $L = \{0^n\,1^m\,2^n \mid n, m \ge 1\}$.	8	L1	CO2
		Module – 3			
Q.5	a.	Design CFG for the following languages : i) $L = \{a^n\,b^{n+3},\ n \ge 0\}$ ii) $L = \{a^i\,b^j\,c^k,\ j = i+k,\ i \ge 0,\ k \ge 0\}$ iii) $L = \{w \mid /w/ \bmod 3 > 0 \text{ where } w \in \{a\}^*\}$ iv) $L = \{a^m\,b^n \mid m \ne n\}$ v) Palinderomes over 0 and 1.	10	L3	CO3
	b.	Consider the grammar G with productions. $S \to A\,b\,B \mid A \mid B$; $A \to aA \mid \varepsilon$; $B \to a\,B \mid b\,B \mid \varepsilon$. Obtain LMD , RMD and parse tree for the string aaabab. Is the given grammar ambiguous?	10	L2	CO3
		OR			
Q.6	a.	Define the following with example : i) Context free grammar ii) Left most Derivation iii) Parse tree iv) Ambiguous grammar.	4	L1	CO3
	b.	Design PDA for the language : $L = \{a^i\,b^j\,c^k \mid i+k = j,\ i \ge 0,\ k \ge 0\}$ and show the moves made by the PDA for the string aabbbc.	10	L3	CO3

	c.	Convert the following CFG's to PDA : $S \rightarrow aA$; $A \rightarrow a\,ABC / bB / a$; $B \rightarrow b$; $C \rightarrow c$.	6	L2	CO3
		Module – 4			
Q.7	a.	Define CNF. Convert the following CFG to CNF $E \rightarrow E + T / T$ $T \rightarrow T * F / F$ $F \rightarrow (E) / I$ $I \rightarrow Ia / Ib / a / b$.	10	L2	CO4
	b.	Show that $L = \{0^n 1^n 2n / n \geq 1\}$ is no context free.	4	L2	CO4
	c.	Prove that the family of context free languages is closed under union and concatenation.	6	L1	CO4
		OR			
Q.8	a.	Define Greibach Normal Form. Convert the following CFG to GNF. $S \rightarrow AB$; $A \rightarrow aA / bB / b$; $B \rightarrow b$.	6	L2	CO4
	b.	Consider the following CFG : $S \rightarrow ABC / BaB$ $A \rightarrow aA / BaC / aaa$ $B \rightarrow bBb / a / D$ $C \rightarrow CA / AC$ $D \rightarrow \varepsilon$ i) What are useless symbols? ii) Eliminate ε - productions , Unit productions and useless symbols from the grammar.	10	L3	CO4
	c.	Prove that the following languages are not context free. i) $L = \{ai / i \text{ is prime}\}$ ii) $L = \{a^{n^2} / n \geq 1\}$.	4	L2	CO3
		Module – 5			
Q.9	a.	Define a turing machine and explain with neat diagram, the working of a basic turing machine.	6	L1	CO4
	b.	Design a Turing machine to accept the language, $L = \{a^n b^n c^n / n \geq 1\}$. Draw the transition diagram and show the moves for the string aabbcc.	14	L4	CO4
		OR			
Q.10	a.	Design a Turing machine to accept palindrome over {a, b} and draw the transition diagram.	12	L4	CO5
	b.	Write a short notes on : i) Recursively Enumerable Language. ii) Multitape Turing Machine.	8	L1	CO5

* * * * *

Feedback/Remarks **Date:17-01-2025 | Session:02:00:00**

For Subject Code : BCS503

1. Q. 1b. There to be read as **three**
2. Q. 3a. bm to $\mathbf{b^m}$
3. Q. 4c. note to be read as **not**
4. Q. 5c. palinderomes to be read as **palindromes**
5. Q. 7b. 2n to $\mathbf{2^n}$
6. Q. 8c. ai to $\mathbf{a^i}$

1 a) Define the following

Strings

A string is *a finite sequence of symbols selected from some alphabet*. It is generally denoted as w.For example for alphabet ∑ = {0, 1} w = 010101 is a string.

Length of a string is denoted as |w| and is defined as the number of positions for the symbol in the string. For the above example length is 6.

Languages

A language is a *set of string all of which are chosen from some ∑*, where ∑ is a particular alphabet*. This means that language L is subset of ∑*. An example is English language, where the collection of legal English words is a set of strings over the alphabet that consists of all the letters. Another example is the C programming language where the alphabet is a subset of the ASCII characters and programs are subset of strings that can be formed from this alphabet.

Power of an alphabet

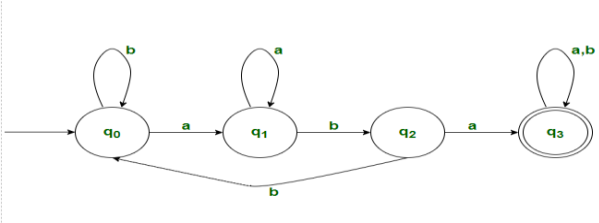
If Σ is an alphabet, the set of all strings can be expressed as a certain length from that alphabet by using exponential notation. The power of an alphabet is denoted by Σk and is the set of strings of length k.

For example,

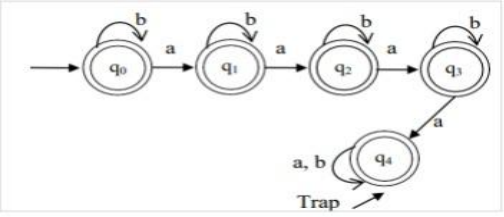
- Σ ={0,1}
- Σ^1= {0,1} (2^1=2)
- Σ^2= {00,01,10,11} (2^2=4)

1b) Define DFA . draw DFA to accept

i)The set of all strings that contain substring aba

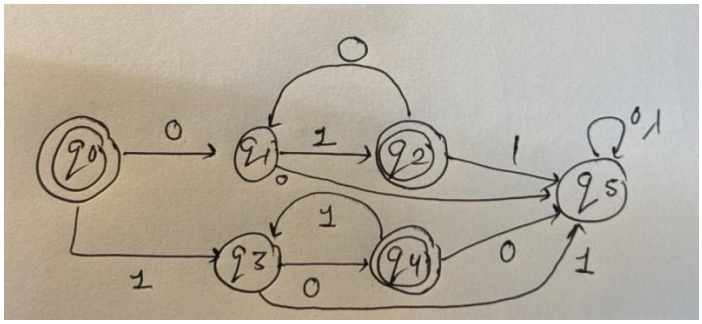


ii) To accept the strings of a's and b's that contain not more than three b's



Not more than 3 b's

iii) L={w belons to {a,b{* : No 2 consecutive characters are same in w}



c) Convert the following NFA to DFA

	0	1
→P	{p.q}	{p}
Q	{r}	{r}
R	{s}	Ø
*S	{s}	{s}

Solution :

	0	1
→P	{p.q}	{p}
{p,q}	{p.q.r}	{p,r}
{p,q,r}	{p.q.r.s}	{p,r}
{p,r}	{p.q.s}	{p}
*{p,q,r,s}	{p,q,r,s}	{p.r.s}
*{p,q,s}	{p.q.r.s}	{p.r.s}
*{p,r,s}	{p,q,s}	{p,s}
*{p,s}	{p,q,s}	{p,s}

2.a Define the following with example

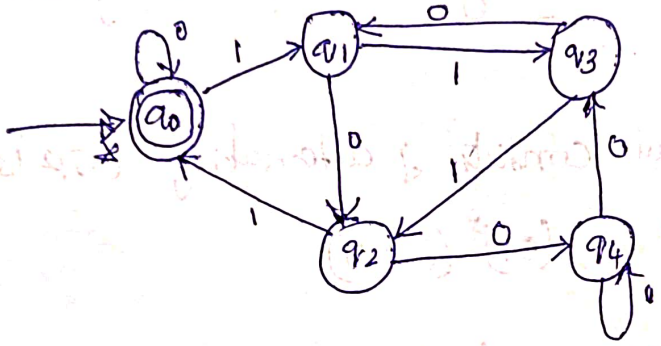
i) Alphabet , ii) Reversal of String, iii) Concatenation of the language

Definition of the above terms

2b. $L = \{w \in \{0,1\}^* : w \text{ is a string divisible by } 5\}$.

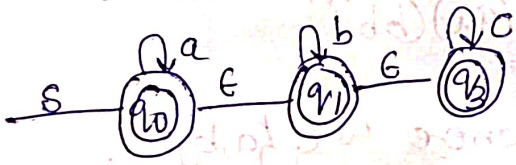
$= \{\epsilon, 5, 10, 15, \ldots\}$.

$= \{\epsilon, 101, 0101, 1010 \ldots\}$.



2c. NFA = $NFM = (Q, \Sigma, \delta, q_0, F)$

$\delta : Q \times \Sigma \to 2^Q$.



$\delta(q_0, \epsilon) = \text{closure}(q_0) = \{q_0, q_1, q_2\}$.

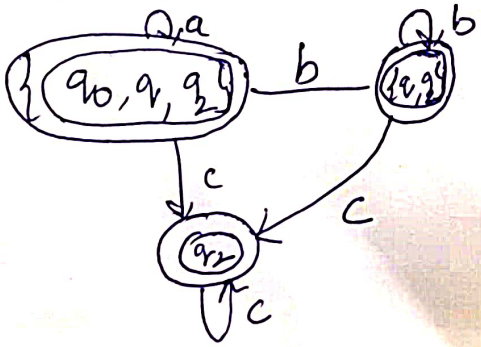
$\delta(q_1, \epsilon) = \text{closure}(q_1) = \{q_1, q_2\}$

$\delta(q_2, \epsilon) = \text{closure}(q_2) = \{q_2\}$.

$\delta(q_0, q_1, q_2), a) = \{q_0\}\{q_1\}\{q_2\} = \{q_0, q_1, q_2\}$

$\delta(q_0, q_2), b) = \{q_1\} = \{q_1, q_2\}$.

$\delta(q_2), c) = \{q_2\} = \{q_2\}$.



Define Regular expression. write regular expression for the foll languages.

Def'n of Regular expression. — 02 M.

(i) Strings of a's & b's starting with a & ending with b.

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

(ii) Set of strings that consists of alternating 0's & 1's.

~~$(01+10)^*$~~ $(\epsilon+1)(01)^*(\epsilon+0)$.

(iii) $L=\{a^n b^m, (n+m) \text{ is even}\}$

$L=\{0, abbb, aabb, aaab, \ldots\}$.

$R= a(aa)^* b(bb)^* + (aa)^*(bb)^*$.

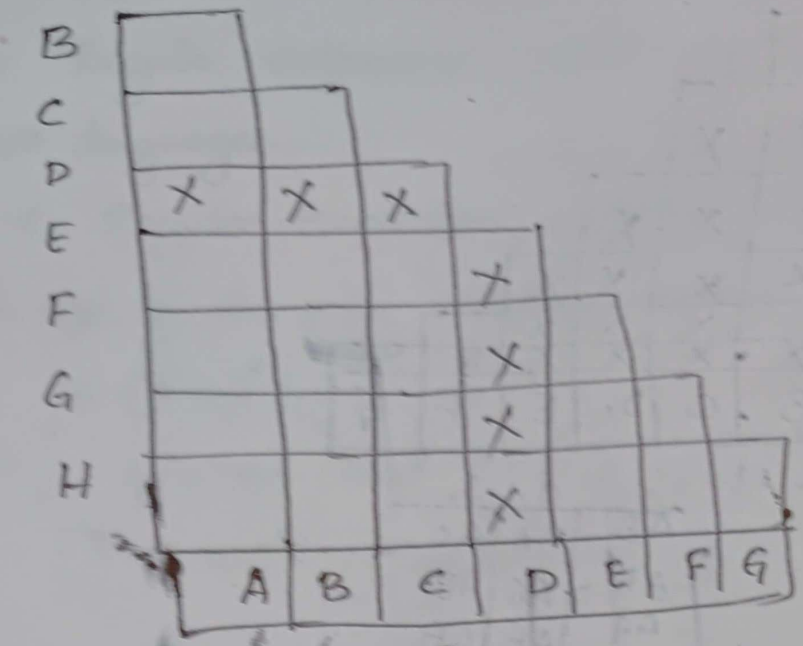
(iv) $L=\{w: |w| \bmod 3 = 0, \text{ where } w \in \{a,b\}^*\}$.

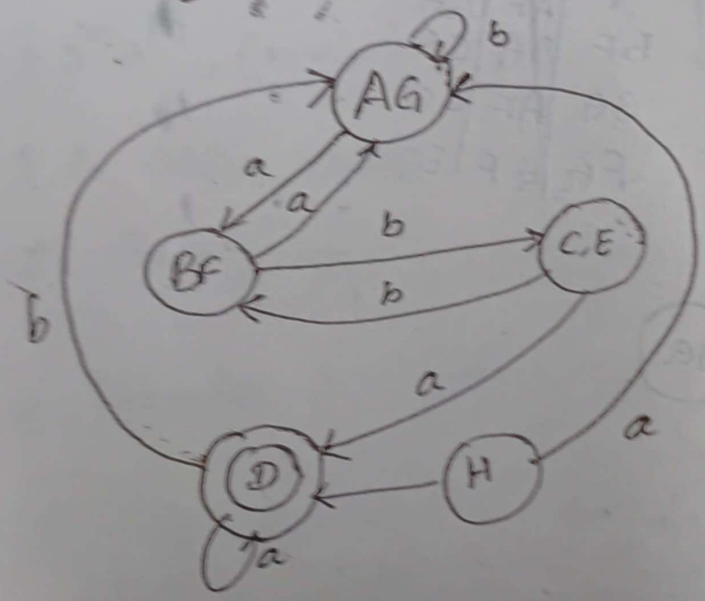
$L=\{0, aaa, aaaaaa \ldots\}$

$R= (aaa)^*((a+b)(a+b)(a+b))^*$

Minimize the following Finite Automata using Tablefilling algorithm.

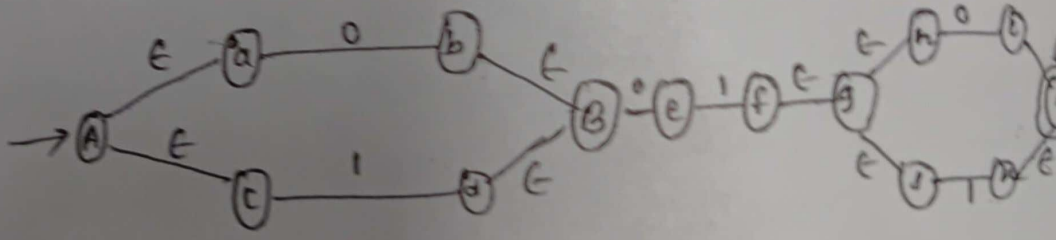
δ	a	b
→ A	B	A
B	A	C
C	D	B
* D	D	A
E	D	F
F	G	E
G	F	G



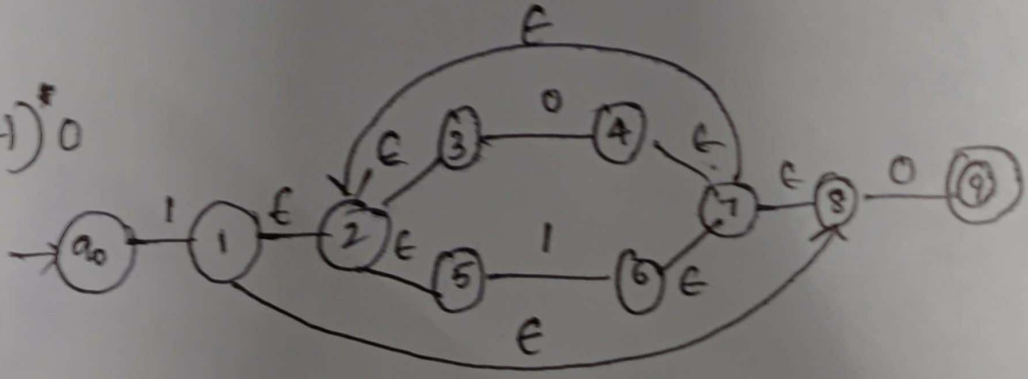


2.

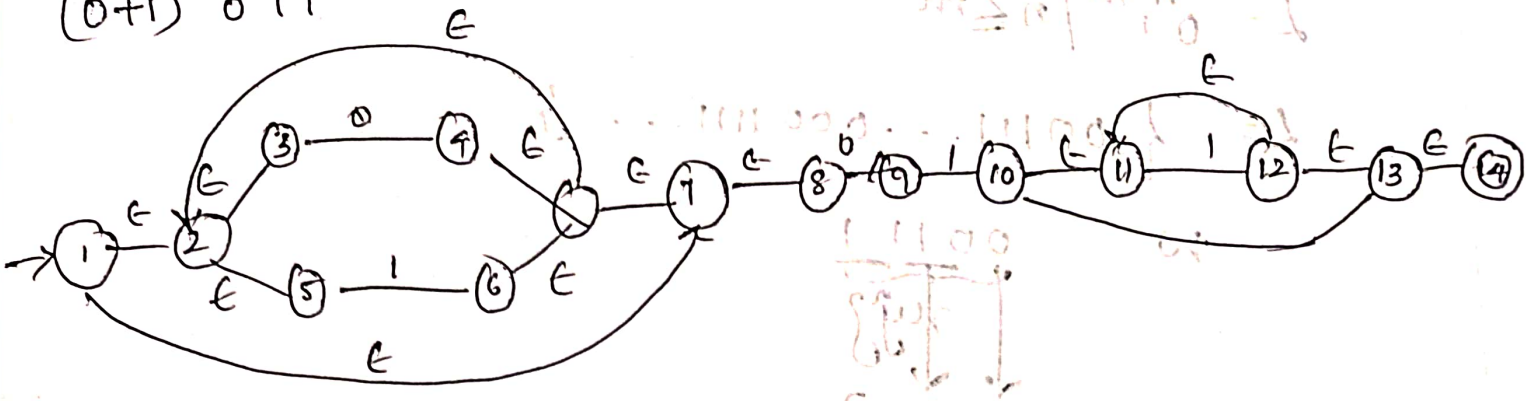
i) $(0+1)0 \mid (1+0)$



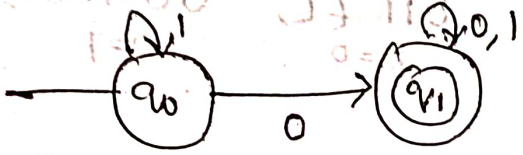
ii) $1(0+1)^*0$



$(0+1)^* 0 1 1^*$



Regular expression that denotes language accepted



R.E	K=0	K=1	K=2
$R_{11}^{(k)}$	$\epsilon+1$	1^*	
$R_{12}^{(k)}$	0	1^*0	$1^*0(0+1)^*$
$R_{21}^{(k)}$	ϕ	ϕ	
$R_{22}^{(k)}$	$\epsilon+0+1$	$\epsilon+0+1$	

Sol: $1^*0(0+1)^*$

State pumping Lemma for regular languages and prove that languages are not regular.

$L = \{0^n 1^m \mid n \le m\}$ (ii) $L = \{0^n 1^m 2^n \mid n, m \ge 1\}$

Sol:- $xy^nz \in L$. $|y| \ge 1$, $\phi \in$

$|xy| \le n$, $|z| \ge n$.

$\{xy^nz \in L, \forall n = 0, 1, 2 \ldots n$.

$L = 0^n 1^m \mid n \leq m$

$L = \{ 00111\ldots 000111\ldots \}$.

$w =$ 00 11 1

$xy^n z$

$x\ y\ z$.

$x = 0,\ y = 01,\ z = 11$

$0.(01)^n 11 = 011 \in L$, $n=0$; $00111 \in L$, $n=1$; $00101111 \notin L$

$L = \{ 0^n 1^m 2^n \mid n, m \geq 1 \}$

$L = \{ 012, 001122, 000111222 \ldots \}$.

$w =$ $\frac{00}{x}\ \frac{11}{y}\ \frac{22}{z}$

$|y| \geq 1 = |01| \geq 1,\ |xz| \leq n$.

$0 0 (11)^n 22$. $\forall n = 0, 1, 2, \ldots$

$0022 \notin L$. Hence they are not regular.

5 a. $L = \{ a^n b^{n+3},\ n \geq 0 \}$.

$P \rightarrow S \rightarrow aSb \mid bbb$, $G = \{ \{S\}, \{a, b\}, P, S \}$

Value of $b = bbb$. $\forall n = 0$

$abbbb$. $\forall n = 1$

$\hookrightarrow$ No. of b is more than the no of a's.

(ii) $L = \{a^i b^j c^k, j = i+k, i \geq 0, k \geq 0\}$

Sol: $P = S_1 S_2$ $S_1 \to aS_1b/\epsilon$, $S_2 \to aS_2b/\epsilon$

$G = (\{S, S_1, S_2\}, \{a, b, c\}, P, S)$

(iii) $L = \{w | w | w \bmod 3 \geq 0$, where $w \in a^*\}$

$P = \{S \to aaaS | a | aa\}$

$G = (\{S\}, \{a\}, P, S)$

(iv) $L = \{a^m b^n | m \geq n\}$

$L = \{a^n a^k b^n | m = n+k\}$

P $S \to aSb | A | B$, $A \to aA | a$; $B \to bB | b$

$G = (\{S, A, B\}, \{a, b\}, P, S)$

(v) palindrome over $\{0, 1\}$

P: $S \to 0S0 | 1S1 | 0 | 1 | \epsilon$

$G = (\{S\}, \{0, 1\}, P, S)$.

Consider the grammar G with prodns

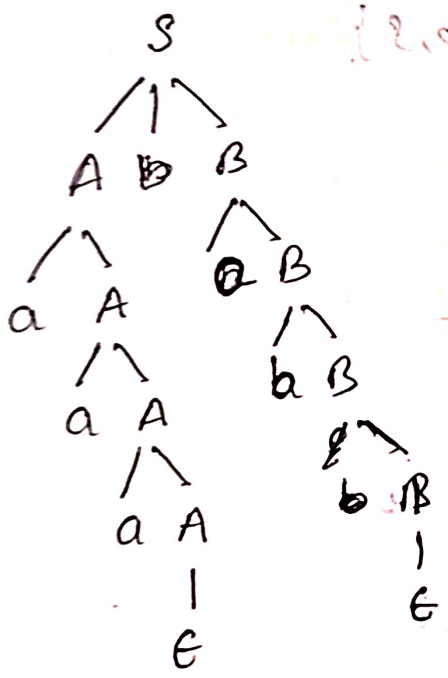
$S \to AbB \mid A \mid B \quad A \to aA \mid \epsilon, \quad B \to aB \mid bB \mid \epsilon$.

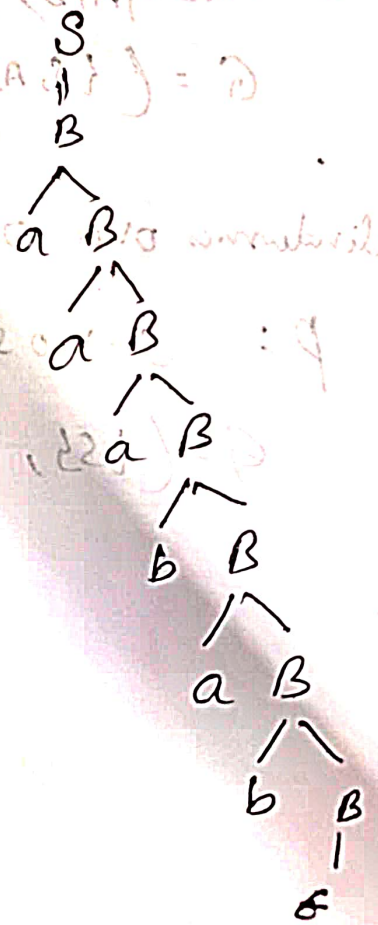
Obtain LMD, RMD & parse tree for string aaabab, is grammar is ambiguous.

$S \Rightarrow AbB$
$S \Rightarrow aAbB$
$S \Rightarrow aaAbB$
$S \Rightarrow aaabB$
$S \Rightarrow aaabaB$
$S \Rightarrow aaababB$
$S \Rightarrow aaababB$
$S \Rightarrow aaababB$

$S \Rightarrow B$
$\Rightarrow aB$
$\Rightarrow aaB$
$\Rightarrow aaaB$
$\Rightarrow aaabB$
$\Rightarrow aaabaB$
$\Rightarrow aaababB$
$\Rightarrow aaababB$

hence grammar is ambiguous.





ANS 6. A)

The definitions of each of these terms in the context of the **Theory of Automata**:

1. Context-Free Grammar (CFG):

A **Context-Free Grammar (CFG)** is a formal grammar where the left-hand side of every production rule consists of a single non-terminal symbol. It is used to define the syntax of programming languages and can generate context-free languages, which are a subset of formal languages.

A CFG is defined by the 4-tuple G=(V,Σ,R,S)G = (V, \Sigma, R, S), where:

- VV is a set of variables (non-terminal symbols),
- Σ\Sigma is a set of terminal symbols (which appear in the strings generated by the grammar),
- RR is a set of production rules (each rule is of the form A→γA \to \gamma, where AA is a non-terminal and γ\gamma is a string of terminals and/or non-terminals),
- SS is the start symbol (a specific non-terminal from VV).

Example:

S → A B

A → a

B → b

2. Left-Most Derivation:

A **left-most derivation** is a process of deriving a string from the start symbol in a way that, at each step, the left-most non-terminal is always replaced first using a production rule.

In a **left-most derivation**:

- The first non-terminal symbol (from the left) in the string is always chosen for expansion.
- This process continues until all the non-terminal symbols are replaced by terminal symbols.

Example of Left-Most Derivation: Given the CFG:

S → A B

A → a

B → b

To derive ab, the left-most derivation would be:

1. Start with S.
2. Replace S with A B (since S → A B).
3. Replace A with a (since A → a).
4. Replace B with b (since B → b).

Resulting string: ab.

3. Parse Tree:

A **parse tree** (or derivation tree) is a tree that represents the syntactic structure of a string according to a context-free grammar. It shows how the start symbol is expanded into a string by applying production rules. Each internal node represents a non-terminal symbol, and the leaves of the tree correspond to the terminal symbols.

In a **parse tree**:

- The root is the start symbol.
- The internal nodes represent the non-terminals, and the edges represent the application of production rules.
- The leaves represent the terminal symbols, which form the string derived.

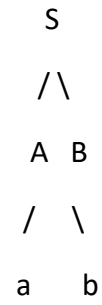
Example of Parse Tree: For the derivation of ab using the CFG:

S → A B

A → a

B → b

The parse tree would look like this:



4. Ambiguous Grammar:

A **grammar is ambiguous** if there is more than one distinct parse tree or left-most derivation for the same string. In other words, a string can be derived in multiple ways using the same grammar.

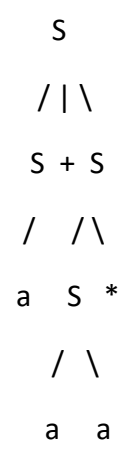
An **ambiguous grammar** may lead to confusion or difficulties in interpreting strings because the meaning or structure of the string could vary depending on the derivation or parse tree chosen.

Example of Ambiguous Grammar: Consider the grammar:

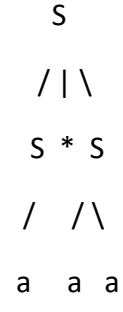
S → S + S | S * S | a

For the string a + a * a, there are two possible parse trees (showing different interpretations of the operations):

1. **First parse tree** (associating * first):



2. **Second parse tree** (associating + first):



Since there are multiple parse trees for the same string, the grammar is **ambiguous**.

Summary:

- **Context-Free Grammar (CFG)**: A formal grammar with rules where each production has a single non-terminal on the left-hand side.

- **Left-Most Derivation**: A derivation where, at each step, the left-most non-terminal is replaced first.
- **Parse Tree**: A tree that represents the syntactic structure of a string derived from a CFG.
- **Ambiguous Grammar**: A grammar that can generate a string in more than one way, leading to multiple parse trees.

ANS 6. B)

Language Description:

The designing a Pushdown Automaton (PDA) for the language L={aibjck|i+k=j}L = \{ a^i b^j c^k \mid i + k = j \}. In other words, the number of bb's should be equal to the sum of the number of aa's and cc's in the string. Specifically, for any string of the form aibjcka^i b^j c^k, the relation i+k=ji + k = j must hold.

PDA Design:

To design the PDA, we need to make use of a stack to keep track of the balance between aa's and cc's, since their combined count needs to match the count of bb's. Here's how we can approach this:

States:

- q0q_0: Initial state.
- q1q_1: Reading aa's and pushing AA onto the stack.
- q2q_2: Reading bb's and matching with the stack.
- q3q_3: Reading cc's and popping from the stack.
- qfq_f: Accepting state (final state).

Stack Alphabet:

- Z0Z_0: Bottom-of-stack marker.
- AA: Marker for each aa (each aa increases the count of i+ki + k).
- BB: Marker for each bb (matches i+ki + k).

Transition Function:

- δ(q0,a,Z0)=(q1,AZ0)\delta(q_0, a, Z_0) = (q_1, A Z_0): On reading aa, push AA onto the stack.

- δ(q1,a,A)=(q1,AA)\delta(q_1, a, A) = (q_1, A A): Continue pushing AA for each aa.
- δ(q1,b,A)=(q2,ϵ)\delta(q_1, b, A) = (q_2, \epsilon): On reading bb, pop AA from the stack.
- δ(q2,b,A)=(q2,ϵ)\delta(q_2, b, A) = (q_2, \epsilon): Continue popping AA for each bb.
- δ(q2,c,Z0)=(q3,Z0)\delta(q_2, c, Z_0) = (q_3, Z_0): On reading cc, move to state q3q_3 without changing the stack (since the count of bb's must balance the sum of aa's and cc's).
- δ(q3,c,A)=(q3,ϵ)\delta(q_3, c, A) = (q_3, \epsilon): On reading cc, pop AA from the stack.
- δ(q3,ϵ,Z0)=(qf,Z0)\delta(q_3, \epsilon, Z_0) = (q_f, Z_0): On reading the empty string and reaching the bottom-of-stack marker, accept.

PDA Moves for the String "aabbbc":

Step-by-Step Execution:

Let's walk through how the PDA would process the string "aabbbc" with this design:

1. **Start at q0q_0, with empty stack Z0Z_0:**
    - Read the first symbol aa:
        - δ(q0,a,Z0)=(q1,AZ0)\delta(q_0, a, Z_0) = (q_1, A Z_0) → Stack: [A,Z0][A, Z_0]
    - Transition to state q1q_1.
2. **In state q1q_1, stack: [A,Z0][A, Z_0]:**
    - Read the second symbol aa:
        - δ(q1,a,A)=(q1,AA)\delta(q_1, a, A) = (q_1, A A) → Stack: [A,A,Z0][A, A, Z_0]
    - Stay in state q1q_1.
3. **In state q1q_1, stack: [A,A,Z0][A, A, Z_0]:**
    - Read the first symbol bb:
        - δ(q1,b,A)=(q2,ϵ)\delta(q_1, b, A) = (q_2, \epsilon) → Stack: [A,Z0][A, Z_0]
    - Transition to state q2q_2.
4. **In state q2q_2, stack: [A,Z0][A, Z_0]:**
    - Read the second symbol bb:
        - δ(q2,b,A)=(q2,ϵ)\delta(q_2, b, A) = (q_2, \epsilon) → Stack: [Z0][Z_0]

- Stay in state q2q_2.

5. **In state q2q_2, stack: [Z0][Z_0]:**
   - Read the third symbol bb:
     - δ(q2,b,Z0)=(q3,Z0)\delta(q_2, b, Z_0) = (q_3, Z_0) → Stack: [Z0][Z_0]
   - Transition to state q3q_3.
6. **In state q3q_3, stack: [Z0][Z_0]:**
   - Read the first symbol cc:
     - δ(q3,c,Z0)=(q3,Z0)\delta(q_3, c, Z_0) = (q_3, Z_0) → Stack: [Z0][Z_0]
   - Stay in state q3q_3.
7. **In state q3q_3, stack: [Z0][Z_0]:**
   - No more symbols to read, and we reach the bottom-of-stack marker Z0Z_0:
     - δ(q3,ϵ,Z0)=(qf,Z0)\delta(q_3, \epsilon, Z_0) = (q_f, Z_0) → Accept.

ANS 6 C)

Let's break down the task of converting the given **Context-Free Grammar (CFG)** into a **Pushdown Automaton (PDA)**.

Given Grammar:

- S→aAS \to aA
- A→aABC|bB|aA \to aABC \mid bB \mid a
- B→bB \to b
- C→cC \to c

Steps for Conversion:

To convert a CFG into a PDA, we'll follow the standard procedure of constructing a PDA that accepts the language generated by the CFG. The PDA will operate by pushing symbols onto the stack as it processes the input and popping them when necessary according to the production rules of the CFG.

PDA Design:

1. **States:**

- q0q_0: The initial state.
- q1q_1: A working state for expanding non-terminals using the production rules.
- qfq_f: The accepting (final) state.

2. **Alphabet:**
   - The **input alphabet** is {a,b,c}\{a, b, c\}.
   - The **stack alphabet** consists of the non-terminals and terminal symbols from the grammar: {S,A,B,C,a,b,c}\{ S, A, B, C, a, b, c \}.
3. **Stack Operation:**
   - The PDA will push symbols onto the stack for non-terminal symbols and pop them when processing the corresponding terminal symbols.
4. **Start State:**
   - The PDA will start in q0q_0 and begin with the start symbol SS on the stack.
5. **Transitions:** The transitions will be defined to mimic the production rules of the grammar. Let's define these transitions based on the productions.

Conversion Process (Transition Function):

1. Start with the start symbol SS on the stack:

- Transition from the start state q0q_0 to process the start symbol SS.
- $\delta(q0,\epsilon,Z0)=(q1,SZ0)$\delta(q_0, \epsilon, Z_0) = (q_1, S Z_0)
  - Here, Z0Z_0 is the bottom-of-stack marker, and we push SS onto the stack.

2. Handle S→aAS \to aA:

- From state q1q_1, when reading aa, we replace SS with aAaA, pushing AA onto the stack after reading the aa.
- $\delta(q1,a,S)=(q1,A)$\delta(q_1, a, S) = (q_1, A)
  - This simulates the production S→aAS \to aA.

3. Handle A→aABCA \to aABC:

- When the top of the stack is AA, we can expand it into aABCaABC. This requires pushing ABCABC onto the stack and consuming the terminal aa.

- $\delta(q1,a,A)=(q1,ABC)\delta(q_1, a, A) = (q_1, A\ B\ C)$

4. Handle A→bBA \to bB:

- If we encounter a bb while the top of the stack is AA, we replace AA with bBbB, consuming the terminal bb.
- $\delta(q1,b,A)=(q1,B)\delta(q_1, b, A) = (q_1, B)$

5. Handle A→aA \to a:

- If AA is on the top of the stack and we encounter an aa, we pop AA (since A→aA \to a) and consume the aa.
- $\delta(q1,a,A)=(q1,\epsilon)\delta(q_1, a, A) = (q_1, \epsilon)$

6. Handle B→bB \to b:

- If BB is on top of the stack, we replace it with the terminal bb, consuming bb.
- $\delta(q1,b,B)=(q1,\epsilon)\delta(q_1, b, B) = (q_1, \epsilon)$

7. Handle C→cC \to c:

- If CC is on the top of the stack, we replace it with the terminal cc, consuming cc.
- $\delta(q1,c,C)=(q1,\epsilon)\delta(q_1, c, C) = (q_1, \epsilon)$

8. Accepting Condition:

- The PDA reaches an accepting state qfq_f when the input string has been processed, and the stack is empty (all symbols have been matched and popped).
- $\delta(q1,\epsilon,Z0)=(qf,Z0)\delta(q_1, \epsilon, Z_0) = (q_f, Z_0)$

PDA Diagram (Conceptually):

1. **Start state q0q_0:**
    - $\delta(q0,\epsilon,Z0)=(q1,SZ0)\delta(q_0, \epsilon, Z_0) = (q_1, S\ Z_0)$
2. **State q1q_1:** (Process rules for SS and AA)
    - $\delta(q1,a,S)=(q1,A)\delta(q_1, a, S) = (q_1, A)$ (for S→aAS \to aA)
    - $\delta(q1,a,A)=(q1,ABC)\delta(q_1, a, A) = (q_1, A\ B\ C)$ (for A→aABCA \to aABC)
    - $\delta(q1,b,A)=(q1,B)\delta(q_1, b, A) = (q_1, B)$ (for A→bBA \to bB)
    - $\delta(q1,a,A)=(q1,\epsilon)\delta(q_1, a, A) = (q_1, \epsilon)$ (for A→aA \to a)

- δ(q1,b,B)=(q1,ϵ)\delta(q_1, b, B) = (q_1, \epsilon) (for B→bB \to b)
- δ(q1,c,C)=(q1,ϵ)\delta(q_1, c, C) = (q_1, \epsilon) (for C→cC \to c)

3. **Accepting state qfq_f:**
    - δ(q1,ϵ,Z0)=(qf,Z0)\delta(q_1, \epsilon, Z_0) = (q_f, Z_0)

Example of Execution for the String "aaabbbc":

Let's break down how the PDA works on the string **"aaabbbc"**:

1. Start in q0q_0 with the stack Z0Z_0.
    - δ(q0,ϵ,Z0)=(q1,SZ0)\delta(q_0, \epsilon, Z_0) = (q_1, S Z_0)
2. In state q1q_1, with the stack [S,Z0][S, Z_0]:
    - Read aa: δ(q1,a,S)=(q1,A)\delta(q_1, a, S) = (q_1, A)
    - Stack: [A,Z0][A, Z_0]
3. In state q1q_1, with the stack [A,Z0][A, Z_0]:
    - Read aa: δ(q1,a,A)=(q1,ABC)\delta(q_1, a, A) = (q_1, A B C)
    - Stack: [A,B,C,Z0][A, B, C, Z_0]
4. In state q1q_1, with the stack [A,B,C,Z0][A, B, C, Z_0]:
    - Read aa: δ(q1,a,A)=(q1,ϵ)\delta(q_1, a, A) = (q_1, \epsilon)
    - Stack: [B,C,Z0][B, C, Z_0]
5. In state q1q_1, with the stack [B,C,Z0][B, C, Z_0]:
    - Read bb: δ(q1,b,B)=(q1,ϵ)\delta(q_1, b, B) = (q_1, \epsilon)
    - Stack: [C,Z0][C, Z_0]
6. In state q1q_1, with the stack [C,Z0][C, Z_0]:
    - Read bb: δ(q1,b,C)=(q1,ϵ)\delta(q_1, b, C) = (q_1, \epsilon)
    - Stack: [Z0][Z_0]
7. In state q1q_1, with the stack [Z0][Z_0]:
    - Read cc: δ(q1,c,Z0)=(qf,Z0)\delta(q_1, c, Z_0) = (q_f, Z_0)
    - Stack: [Z0][Z_0] (reached the accepting state)

MODULE 4

ANS 7. A)

Definition of CNF (Chomsky Normal Form)

A Context-Free Grammar (CFG) is said to be in Chomsky Normal Form (CNF) if all of its production rules satisfy the following conditions:

1. Each production is of the form:
    - A→BCA \to BC, where AA is a non-terminal and BB and CC are non-terminals (no terminal symbols appear on the right-hand side of the production).
    - OR
    - A→aA \to a, where aa is a terminal symbol (a production that directly leads to a terminal).
2. The start symbol can have the form S→ϵS \to \epsilon, if the language includes the empty string (which is rare and optional in CNF).

To convert a CFG into CNF, we generally follow these steps:

- Eliminate null (ε) productions.
- Eliminate unit productions.
- Eliminate useless symbols.
- Ensure that all productions are of the form A→BCA \to BC or A→aA \to a, by introducing new variables (non-terminals) if needed.

Given Grammar:

- E→E+T|TE \to E + T \mid T
- T→T∗F|FT \to T * F \mid F
- F→(E)|IF \to (E) \mid I
- I→Ia|Ib|a|bI \to I a \mid I b \mid a \mid b

Step-by-Step Conversion to CNF:

We will convert the given grammar into CNF step-by-step. First, let's simplify the grammar and follow the rules of CNF conversion.

1. Eliminate Null Productions (ε-productions):

There are no epsilon productions (productions of the form A→ϵA \to \epsilon) in the given grammar, so we don't need to eliminate any.

2. Eliminate Unit Productions:

A unit production is a production of the form A→BA \to B, where both AA and BB are non-terminals. We need to replace unit productions by expanding them with their respective definitions.

Let's look at the productions involving unit productions:

- E→TE \to T (from E→E+TE \to E + T and E→TE \to T) is a unit production.
- T→FT \to F (from T→T∗FT \to T * F and T→FT \to F) is another unit production.

We need to replace these unit productions with the corresponding right-hand side.

- Replace E→TE \to T with E→T∗F|FE \to T * F \mid F (expand the production rules for TT).
- Replace T→FT \to F with T→(E)|IT \to (E) \mid I (expand the production rules for FF).

After expanding the unit productions, the grammar becomes:

- E→E+T|T∗F|FE \to E + T \mid T * F \mid F
- T→T∗F|FT \to T * F \mid F
- F→(E)|IF \to (E) \mid I
- I→Ia|Ib|a|bI \to I a \mid I b \mid a \mid b

3. Eliminate Left Recursion:

The given grammar contains left recursion, which needs to be removed in order to convert the grammar to CNF.

For example, the rule E→E+TE \to E + T is left-recursive. We eliminate left recursion by rewriting the production rules.

- For E→E+T|TE \to E + T \mid T, we introduce a new non-terminal E’E' and rewrite the productions as:
    - E→TE’E \to T E'

    - E′→+TE′|ϵE' \to + T E' \mid \epsilon

Now, the rules for EE become:

- E→TE′E \to T E'
- E′→+TE′|ϵE' \to + T E' \mid \epsilon

4. Ensure All Productions Are in CNF:

Now we need to ensure that every production is in the form of A→BCA \to BC or A→aA \to a. We may need to introduce new non-terminals to achieve this.

For the production E′→+TE′E' \to + T E':

- We can't have the terminal ++ in the production like this, so we introduce a new non-terminal X+X_+ to replace the terminal ++.
- So, X+→+X_+ \to +.

Thus, we modify the grammar as follows:

- E′→X+TE′|ϵE' \to X_+ T E' \mid \epsilon
- X+→+X_+ \to +

For the production T→T∗FT \to T * F:

- We introduce a new non-terminal X∗X_* to replace the terminal ∗*.
- So, X∗→∗X_* \to *.
- Thus, we modify TT to be:
    - T→X∗FT′|FT \to X_* F T' \mid F
    - T′→X∗FT′|ϵT' \to X_* F T' \mid \epsilon

For the production F→(E)F \to (E):

- We introduce a new non-terminal X()X_() to replace the terminal ((and X()′X_()' to replace)).
- So, X()→(X_() \to (and X()′→)X_()' \to).
- Modify FF to:
    - F→X()EX()′|IF \to X_() E X_()' \mid I

For the production I→IaI \to I a:

- We introduce a new non-terminal XaX_a to replace the terminal aa.
- Xa→aX_a \to a.
- Thus, modify II to:
    - I→IXa|Xab|XaI \to I X_a \mid X_a b \mid X_a

Now, the grammar becomes:

- E→TE’E \to T E'
- E’→X+TE’|ϵE' \to X_+ T E' \mid \epsilon
- X+→+X_+ \to +
- T→X∗FT’|FT \to X_* F T' \mid F
- T’→X∗FT’|ϵT' \to X_* F T' \mid \epsilon
- X∗→∗X_* \to *
- F→X()EX()’|IF \to X_() E X_()' \mid I
- X()→(X_() \to (
- X()’→)X_()' \to)
- I→IXa|Xab|XaI \to I X_a \mid X_a b \mid X_a
- Xa→aX_a \to a
- Xb→bX_b \to b

Final Grammar in CNF:

The resulting grammar in Chomsky Normal Form (CNF) is as follows:

1. E→TE’E \to T E'
2. E’→X+TE’|ϵE' \to X_+ T E' \mid \epsilon
3. X+→+X_+ \to +
4. T→X∗FT’|FT \to X_* F T' \mid F
5. T’→X∗FT’|ϵT' \to X_* F T' \mid \epsilon
6. X∗→∗X_* \to *
7. F→X()EX()’|IF \to X_() E X_()' \mid I

8. X()→(X_() \to (
9. X()'→)X_()' \to)
10. I→IXa|Xab|XaI \to I X_a \mid X_a b \mid X_a
11. Xa→aX_a \to a
12. Xb→bX_b \to b

ANS 7. B)

The language L={0n1n2n|n≥1}L = \{ 0^n 1^n 2^n \mid n \geq 1 \} is not a context-free language (CFL). To show this, we can use the pumping lemma for context-free languages. The pumping lemma is a proof technique that can be used to show that certain languages are not context-free.

Pumping Lemma for Context-Free Languages:

The pumping lemma for CFLs states that for any context-free language LL, there exists a constant pp (called the pumping length) such that any string ww in LL with |w|≥p|w| \geq p can be decomposed into five parts, w=uvxyzw = uvxyz, such that:

1. |vxy|≤p|vxy| \leq p
2. |vy|≥1|vy| \geq 1
3. For all i≥0i \geq 0, the string uvixyizuv^i x y^i z is in LL.

In other words, we can "pump" the middle part of the string (represented by vv and yy) any number of times, and the resulting string will still belong to the language.

Proof:

To prove that the language L={0n1n2n|n≥1}L = \{ 0^n 1^n 2^n \mid n \geq 1 \} is not context-free, we will use the pumping lemma.

Step 1: Assume that the language is context-free.

Assume, for the sake of contradiction, that LL is a context-free language. Then, by the pumping lemma for CFLs, there exists a pumping length pp such that any string w∈Lw \in L with |w|≥p|w| \geq p can be decomposed into five parts w=uvxyzw = uvxyz, where the conditions of the pumping lemma hold.

Step 2: Choose a string from LL.

Let's choose the string w=0p1p2p∈Lw = 0^p 1^p 2^p \in L, where pp is the pumping length given by the pumping lemma. The string has length 3p3p, which is greater than or equal to the pumping length, so it must be pumpable according to the pumping lemma.

Step 3: Decompose the string ww.

According to the pumping lemma, we can decompose ww as w=uvxyzw = uvxyz, where:

- |vxy|≤p|vxy| \leq p
- |vy|≥1|vy| \geq 1
- For all i≥0i \geq 0, the string uvixyiz∈Luv^i x y^i z \in L.

Since w=0p1p2pw = 0^p 1^p 2^p, we know that the string consists of three distinct sections: 0p0^p, 1p1^p, and 2p2^p. The substring vxyvxy must be entirely contained within one or two of these sections, because |vxy|≤p|vxy| \leq p and each section has length pp.

Step 4: Analyze possible locations for vxyvxy.

Let's consider the possible cases for the location of the substring vxyvxy:

1. Case 1: vxyvxy is entirely within the first block (the 0p0^p block).
    - In this case, vv and yy will both consist of 0's, and pumping vv and yy will add more 0's.
    - After pumping, the string will have more 0's than 1's and 2's, which cannot be in the form 0n1n2n0^n 1^n 2^n. Thus, the string will not belong to LL.
2. Case 2: vxyvxy is entirely within the second block (the 1p1^p block).
    - In this case, vv and yy will both consist of 1's, and pumping vv and yy will add more 1's.
    - After pumping, the string will have more 1's than 0's and 2's, which again cannot be in the form 0n1n2n0^n 1^n 2^n. Thus, the string will not belong to LL.
3. Case 3: vxyvxy is entirely within the third block (the 2p2^p block).
    - In this case, vv and yy will both consist of 2's, and pumping vv and yy will add more 2's.
    - After pumping, the string will have more 2's than 0's and 1's, which also cannot be in the form 0n1n2n0^n 1^n 2^n. Thus, the string will not belong to LL.

4. Case 4: vxyvxy spans across two blocks (e.g., between the 0p0^p and 1p1^p blocks, or between the 1p1^p and 2p2^p blocks).
    - In this case, pumping vv and yy will alter the balance between the different symbols (0's, 1's, and 2's).
    - For example, if vxyvxy spans the 0's and 1's, pumping will result in an unequal number of 0's and 1's, violating the structure of the string, which requires equal numbers of 0's, 1's, and 2's.

Step 5: Conclusion.

In all possible cases, pumping the string results in a string that no longer belongs to LL. This contradicts the pumping lemma, which states that for a context-free language, pumping should always produce strings that still belong to the language.

Therefore, our assumption that LL is a context-free language must be false. Hence, the language L={0n1n2n|n≥1}L = \{ 0^n 1^n 2^n \mid n \geq 1 \} is not a context-free language.

Final Answer:

The language L={0n1n2n|n≥1}L = \{ 0^n 1^n 2^n \mid n \geq 1 \} is not context-free, as shown by applying the pumping lemma for context-free languages.

ANS 7. C)

Proof that Context-Free Languages (CFLs) are Closed under Union and Concatenation

In this proof, we will demonstrate that the class of Context-Free Languages (CFLs) is closed under union and concatenation. This means that if L1L_1 and L2L_2 are both CFLs, then:

- The union L1∪L2L_1 \cup L_2 is also a CFL.
- The concatenation L1⋅L2L_1 \cdot L_2 is also a CFL.

We will prove both closure properties separately by constructing context-free grammars (CFGs) for the union and concatenation of two CFLs.

---

1. Closure under Union

Goal:

If L1L_1 and L2L_2 are context-free languages, then L1∪L2L_1 \cup L_2 is also a context-free language.

Proof:

Let G1=(V1,Σ,R1,S1)G_1 = (V_1, \Sigma, R_1, S_1) be a context-free grammar for L1L_1, and let G2=(V2,Σ,R2,S2)G_2 = (V_2, \Sigma, R_2, S_2) be a context-free grammar for L2L_2, where:

- V1V_1 and V2V_2 are sets of non-terminals,
- Σ\Sigma is the alphabet,
- R1R_1 and R2R_2 are sets of production rules,
- S1S_1 and S2S_2 are the start symbols for G1G_1 and G2G_2, respectively.

To construct a grammar for L1∪L2L_1 \cup L_2, we create a new start symbol SS, and define a new grammar G=(V,Σ,R,S)G = (V, \Sigma, R, S), where:

1. Non-terminal set: V=V1∪V2∪{S}V = V_1 \cup V_2 \cup \{ S \}, the union of the non-terminals from both grammars and the new start symbol SS.
2. Production rules:
    - S→S1∣S2S \to S_1 \mid S_2
    - All the rules from G1G_1: R1R_1
    - All the rules from G2G_2: R2R_2
3. Start symbol: SS, which can derive either S1S_1 or S2S_2.

Explanation:

- SS can derive either the start symbol S1S_1 from G1G_1 or the start symbol S2S_2 from G2G_2, allowing the new grammar to generate strings from either L1L_1 or L2L_2.
- The grammar GG generates exactly the union of L1L_1 and L2L_2, i.e., L(G)=L1∪L2L(G) = L_1 \cup L_2.

Since this is a valid context-free grammar, it follows that CFLs are closed under union.

---

2. Closure under Concatenation

Goal:

If L1L_1 and L2L_2 are context-free languages, then L1⋅L2L_1 \cdot L_2 (the concatenation of L1L_1 and L2L_2) is also a context-free language.

Proof:

Let G1=(V1,Σ,R1,S1)G_1 = (V_1, \Sigma, R_1, S_1) be a context-free grammar for L1L_1, and let G2=(V2,Σ,R2,S2)G_2 = (V_2, \Sigma, R_2, S_2) be a context-free grammar for L2L_2.

To construct a grammar for L1·L2L_1 \cdot L_2, we create a new start symbol SS and define a new grammar G=(V,Σ,R,S)G = (V, \Sigma, R, S), where:

1. Non-terminal set: V=V1∪V2∪{S}V = V_1 \cup V_2 \cup \{ S \}, the union of the non-terminals from both grammars and the new start symbol SS.
2. Production rules:
    - S→S1S2S \to S_1 S_2, where S1S_1 is the start symbol for G1G_1 and S2S_2 is the start symbol for G2G_2.
    - All the rules from G1G_1: R1R_1
    - All the rules from G2G_2: R2R_2
3. Start symbol: SS, which produces the concatenation of the strings generated by L1L_1 and L2L_2 (i.e., S→S1S2S \to S_1 S_2).

Explanation:

- The start symbol SS in the new grammar produces a string from L1L_1 followed by a string from L2L_2, which is the definition of concatenation.
- The rest of the grammar generates strings from L1L_1 and L2L_2 according to the original grammars G1G_1 and G2G_2.

Since this is a valid context-free grammar, it follows that CFLs are closed under concatenation.

Conclusion:

We have shown that context-free languages are closed under both union and concatenation by constructing context-free grammars for these operations. Therefore:

- CFLs are closed under union.
- CFLs are closed under concatenation.

Thus, the class of context-free languages is indeed closed under both the union and concatenation operations.

ANS 8. A)

Greibach Normal Form (GNF)

A Context-Free Grammar (CFG) is said to be in Greibach Normal Form (GNF) if all of its production rules are of the form:

- A→aαA \to a \alpha

Where:

- AA is a non-terminal,
- aa is a terminal symbol,
- α\alpha is a string of non-terminals (which can be empty, i.e., α\alpha can be the empty string ϵ\epsilon).

In GNF, the right-hand side of each production must start with a terminal symbol, followed by zero or more non-terminals.

Steps to Convert a CFG to GNF

The general steps to convert a CFG to Greibach Normal Form are:

1. Eliminate epsilon (ε) productions: Remove any production of the form A→ϵA \to \epsilon, unless ϵ\epsilon is part of the language.
2. Eliminate unit productions: Remove any production of the form A→BA \to B, where both AA and BB are non-terminals.
3. Eliminate left recursion: A CFG in GNF should not have left recursion (i.e., A→AαA \to A\alpha type rules). Left recursion must be removed by applying a transformation.
4. Rearrange rules into GNF: Convert all production rules so that they are in the form A→aαA \to a\alpha.

Given Grammar:

- S→ABS \to AB
- A→aA|bB|bA \to aA \mid bB \mid b
- B→bB \to b

Step-by-Step Conversion to GNF

We need to convert the given CFG into Greibach Normal Form.

Step 1: Eliminate epsilon (ε) productions

- The grammar does not contain any epsilon productions (i.e., no rule of the form A→ϵA \to \epsilon).

Step 2: Eliminate unit productions

A unit production is a rule where a non-terminal produces another non-terminal directly, like A→BA \to B. We need to eliminate these.

- We have A→bBA \to bB. Here, AA produces bBbB, so we can expand BB using its production rule B→bB \to b.

Thus, A→bBA \to bB can be rewritten as A→bbA \to bb.

After expanding the unit production, the grammar becomes:

- S→ABS \to AB
- A→aA|bb|bA \to aA \mid bb \mid b
- B→bB \to b

Step 3: Eliminate left recursion

The rule A→aAA \to aA is left recursive because AA appears on the right-hand side immediately after the terminal symbol aa. To eliminate the left recursion, we rewrite the rules for AA using the following standard method for removing left recursion:

1. Split the production rules for AA into two parts:
    - A→aA’A \to aA'
    - A’→aA’|bb|bA' \to aA' \mid bb \mid b

Here, we replace the left-recursive rule with a new non-terminal A’A', and the new production A’A' generates the recursive and non-recursive parts.

Step 4: Rearranging to GNF

Now, all rules need to start with a terminal symbol. The current grammar rules are:

- S→ABS \to AB
- A→aA’|bb|bA \to aA' \mid bb \mid b
- A’→aA’|bb|bA' \to aA' \mid bb \mid b
- B→bB \to b

We will now express each rule in GNF, i.e., where each production starts with a terminal.

- S→ABS \to AB can remain as it is, as AA and BB will be expanded recursively into terminal symbols.
- A→aA’A \to aA' is in GNF.
- A→bbA \to bb is in GNF (since it starts with terminal bb).
- A→bA \to b is also in GNF.
- A’→aA’A' \to aA' is already in GNF.
- A’→bbA' \to bb is in GNF.
- A’→bA' \to b is in GNF.
- B→bB \to b is already in GNF.

Thus, after converting the CFG, we have the following Greibach Normal Form grammar:

Final Grammar in GNF:

- S→ABS \to AB
- A→aA’|bb|bA \to aA' \mid bb \mid b
- A’→aA’|bb|bA' \to aA' \mid bb \mid b
- B→bB \to b

AND 8. B)

Let's go through the steps systematically to eliminate null productions, unit productions, and useless symbols from the given Context-Free Grammar (CFG).

Given CFG:

- S→ABC|BaBS \to ABC \mid BaB
- A→aA|BaC|aaaA \to aA \mid BaC \mid aaa
- B→bBb|a|DB \to bBb \mid a \mid D
- C→CA|ACC \to CA \mid AC
- D→ϵD \to \epsilon (Null production)

Step 1: Eliminate Null Productions

A null production is a production of the form A→ϵA \to \epsilon, where ϵ\epsilon represents the empty string. In this grammar, D→ϵD \to \epsilon is a null production.

To eliminate null productions, we replace occurrences of non-terminals that could derive ϵ\epsilon with alternatives that account for the possibility of producing nothing.

- D→ϵD \to \epsilon implies that DD can be removed or replaced by ϵ\epsilon in any production where it appears.

1.1. Modify productions containing DD:

- B→bBb|a|DB \to bBb \mid a \mid D
    - Since D→ϵD \to \epsilon, we need to replace DD with ϵ\epsilon in the production B→DB \to D. This results in an additional production: B→ϵB \to \epsilon.
    - So, B→bBb|a|ϵB \to bBb \mid a \mid \epsilon.

1.2. Modify productions containing BB:

Since B→ϵB \to \epsilon, we need to consider the following rules:

- In the rule S→BaBS \to BaB, replace BB with ϵ\epsilon to get:
    - S→BaB|aS \to BaB \mid a
    - This means we add a new production: S→aS \to a.
- In the rule A→BaCA \to BaC, replace BB with ϵ\epsilon to get:
    - A→aC|aaaA \to aC \mid aaa.

After eliminating null productions, the grammar becomes:

- S→ABC|BaB|aS \to ABC \mid BaB \mid a
- A→aA|BaC|aaaA \to aA \mid BaC \mid aaa
- B→bBb|a|ϵB \to bBb \mid a \mid \epsilon
- C→CA|ACC \to CA \mid AC
- D→ϵD \to \epsilon

Step 2: Eliminate Unit Productions

A unit production is a production of the form A→BA \to B, where AA and BB are non-terminals. Let's eliminate the unit productions:

- B→DB \to D is a unit production. Since D→ϵD \to \epsilon, we replace B→DB \to D with B→ϵB \to \epsilon.

- So, we already have the production B→ϵB \to \epsilon, so we can ignore the unit production B→DB \to D.

After eliminating unit productions, the grammar becomes:

- S→ABC|BaB|aS \to ABC \mid BaB \mid a
- A→aA|BaC|aaaA \to aA \mid BaC \mid aaa
- B→bBb|a|ϵB \to bBb \mid a \mid \epsilon
- C→CA|ACC \to CA \mid AC
- D→ϵD \to \epsilon

Step 3: Eliminate Useless Symbols

A useless symbol is a symbol that doesn't contribute to generating any terminal string. To identify and eliminate useless symbols, we follow two steps:

1. Identify non-generating symbols: A symbol is generating if it can eventually produce a string of terminal symbols.
2. Identify reachable symbols: A symbol is reachable if it can be reached from the start symbol SS.

3.1. Identify non-generating symbols

We first check which non-terminals can generate terminal strings:

- D→ϵD \to \epsilon, so DD is generating.
- B→bBb|a|ϵB \to bBb \mid a \mid \epsilon, so BB is generating because it can produce aa, or bBbbBb, or ϵ\epsilon.
- A→aA|BaC|aaaA \to aA \mid BaC \mid aaa, so AA is generating because aaaaaa is a terminal string.
- C→CA|ACC \to CA \mid AC, but CC does not generate terminal strings directly because it only generates other non-terminals AA and CC, which can only generate terminal strings indirectly.

So, C is not generating and should be removed.

3.2. Identify reachable symbols

Now, let's check which symbols are reachable from the start symbol SS:

- S→ABCS \to ABC, so SS reaches AA, BB, and CC.
- A→aA|BaC|aaaA \to aA \mid BaC \mid aaa, so AA is reachable.
- B→bBb|a|ϵB \to bBb \mid a \mid \epsilon, so BB is reachable.
- C→CA|ACC \to CA \mid AC, so CC is reachable from AA.
- D→ϵD \to \epsilon, so DD is reachable.

Since CC is not generating, it should be eliminated from the grammar.

After eliminating useless symbols, the grammar becomes:

- S→ABC|BaB|aS \to ABC \mid BaB \mid a
- A→aA|aaa|BaCA \to aA \mid aaa \mid BaC (remove CC from productions)
- B→bBb|a|ϵB \to bBb \mid a \mid \epsilon

Final Grammar After All Reductions:

- S→ABC|BaB|aS \to ABC \mid BaB \mid a
- A→aA|aaa|BaA \to aA \mid aaa \mid Ba
- B→bBb|a|ϵB \to bBb \mid a \mid \epsilon

ANS 8. C)

Let's prove that the following two languages are not context-free using the pumping lemma for context-free languages:

1. L1={ai|i is prime}L_1 = \{ a^i \mid i \text{ is prime} \}
2. L2={an2|n≥1}L_2 = \{ a^{n^2} \mid n \geq 1 \}

1. Proving L1={ai|i is prime}L_1 = \{ a^i \mid i \text{ is prime} \} is not context-free

We will use the pumping lemma for context-free languages to prove that L1L_1 is not context-free.

Pumping Lemma for CFLs:

The pumping lemma for context-free languages states that if a language LL is context-free, then there exists a pumping length pp such that any string ww in LL with length |w|≥p|w| \geq p can be decomposed into five parts: w=uvxyzw = uvxyz, satisfying the following conditions:

1. |vxy|≤p|vxy| \leq p
2. |vy|≥1|vy| \geq 1
3. For all i≥0i \geq 0, the string uvixyizuv^i x y^i z is in LL.

Proof:

Let's assume, for the sake of contradiction, that L1={ai|i is prime}L_1 = \{ a^i \mid i \text{ is prime} \} is context-free. By the pumping lemma, there exists a pumping length pp such that any string w∈L1w \in L_1 with |w|≥p|w| \geq p can be decomposed into w=uvxyzw = uvxyz, satisfying the above conditions.

Now, let's choose the string w=aqw = a^q where qq is a prime number greater than or equal to pp. Thus, w=aq∈L1w = a^q \in L_1.

By the pumping lemma, we can write w=uvxyzw = uvxyz, where:

- |vxy|≤p|vxy| \leq p
- |vy|≥1|vy| \geq 1
- uvixyiz∈L1uv^i x y^i z \in L_1 for all i≥0i \geq 0

Key Observation:

- The string w=aqw = a^q is a string of length qq, where qq is prime.
- Pumping vv and yy (which consist only of aa's) results in strings of the form aq+ka^{q + k} for some integer kk, where kk is the number of additional aa's added by pumping.

If we pump vv and yy, we get strings of the form aq+ka^{q + k}. For these strings to be in L1L_1, the length of the string must be prime. However, if we pump the string to any value aq+ka^{q+k}, we get a length that is not prime for most values of kk. The prime property is lost after pumping, which contradicts the fact that L1L_1 only contains strings of prime length.

Thus, we cannot ensure that the resulting string remains in L1L_1, which implies that L1L_1 is not context-free.

2. Proving L2={an2|n≥1}L_2 = \{ a^{n^2} \mid n \geq 1 \} is not context-free

Now, we will prove that L2={an2|n≥1}L_2 = \{ a^{n^2} \mid n \geq 1 \} is not a context-free language.

We will also use the pumping lemma for context-free languages to prove this.

Proof:

Let's assume, for the sake of contradiction, that L2={an2|n≥1}L_2 = \{ a^{n^2} \mid n \geq 1 \} is context-free. By the pumping lemma, there exists a pumping length pp such that any string w∈L2w \in L_2 with |w|≥p|w| \geq p can be decomposed into w=uvxyzw = uvxyz, where:

- |vxy|≤p|vxy| \leq p
- |vy|≥1|vy| \geq 1
- uvixyiz∈L2uv^i x y^i z \in L_2 for all i≥0i \geq 0

Now, let's consider the string w=ap2w = a^{p^2}, which is in L2L_2 since p2p^2 is a perfect square.

According to the pumping lemma, we can decompose w=ap2=uvxyzw = a^{p^2} = uvxyz such that the conditions of the pumping lemma hold. Let's analyze the behavior when we pump the middle part vv and yy.

Key Observation:

- The string w=ap2w = a^{p^2} is of length p2p^2, a perfect square.
- When we pump the middle part vv and yy, we get strings of the form uvixyizuv^i x y^i z, where the new length of the string is p2+kp^2 + k for some integer kk, corresponding to the number of aa's added by pumping.

Now, the length of the string after pumping must still be a perfect square, but after pumping, we are unlikely to get a perfect square. This is because adding or removing a number of aa's will typically break the perfect square property, and the resulting length will not be a perfect square.

For example, if i=2i = 2, the new length will be p2+2kp^2 + 2k, which is generally not a perfect square. Hence, after pumping, the resulting string will not belong to L2L_2, because it will not have a length of the form n2n^2 for any integer nn.

Thus, L2L_2 is not context-free because we cannot guarantee that pumping will result in strings whose length is a perfect square.

Conclusion:

Both languages L1={ai|i is prime}L_1 = \{ a^i \mid i \text{ is prime} \} and L2={an2|n≥1}L_2 = \{ a^{n^2} \mid n \geq 1 \} are not context-free. We proved this using the pumping lemma for context-free languages, showing that pumping leads to strings that no longer satisfy the conditions for being in the respective languages.

MODULE 5

ANS 9. A)

Definition of a Turing Machine

A Turing Machine (TM) is a mathematical model of computation that defines a machine capable of simulating any algorithm. It consists of an infinite tape, a head that can read and write symbols, and a set of states that determine its operations. A Turing Machine is a formal model of computation that is widely used to study computability and complexity.

A Turing Machine consists of the following components:

1. Tape:
    - An infinite sequence of cells, each containing a symbol from a finite alphabet Σ\Sigma (including a special blank symbol #\#).
    - The tape is used for both input and output, and it moves left and right under the control of the machine.
2. Tape Head:
    - A device that reads and writes symbols on the tape.
    - It can move left, right, or stay in the current position depending on the machine's transition function.
3. State Register:
    - The machine has a finite set of states QQ (including a special start state and halting states).
    - At any given time, the Turing Machine is in one of these states.
4. Transition Function:
    - The transition function δ\delta defines the behavior of the Turing Machine.
    - It is a function: δ:Q×Σ→Q×Σ×{L,R,S}\delta: Q \times \Sigma \to Q \times \Sigma \times \{L, R, S\} This means that for a given state and a symbol on the tape, the machine will:
        - Change to a new state,
        - Write a new symbol on the tape,
        - Move the tape head either left (L), right (R), or stay in the same position (S).
5. Start State:

- The state from which the machine begins its operation, denoted as q0q_0.

6. Accepting and Rejecting States:
    - Some states are designated as accepting (halting) states, where the machine halts and indicates success.
    - Some states are rejecting (halting) states, where the machine halts and indicates failure.

Formal Definition of a Turing Machine:

A Turing Machine is formally defined as a 7-tuple:

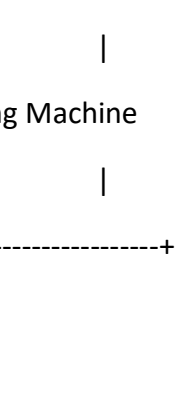
M=(Q,Σ,Γ,δ,q0,qaccept,qreject)M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})

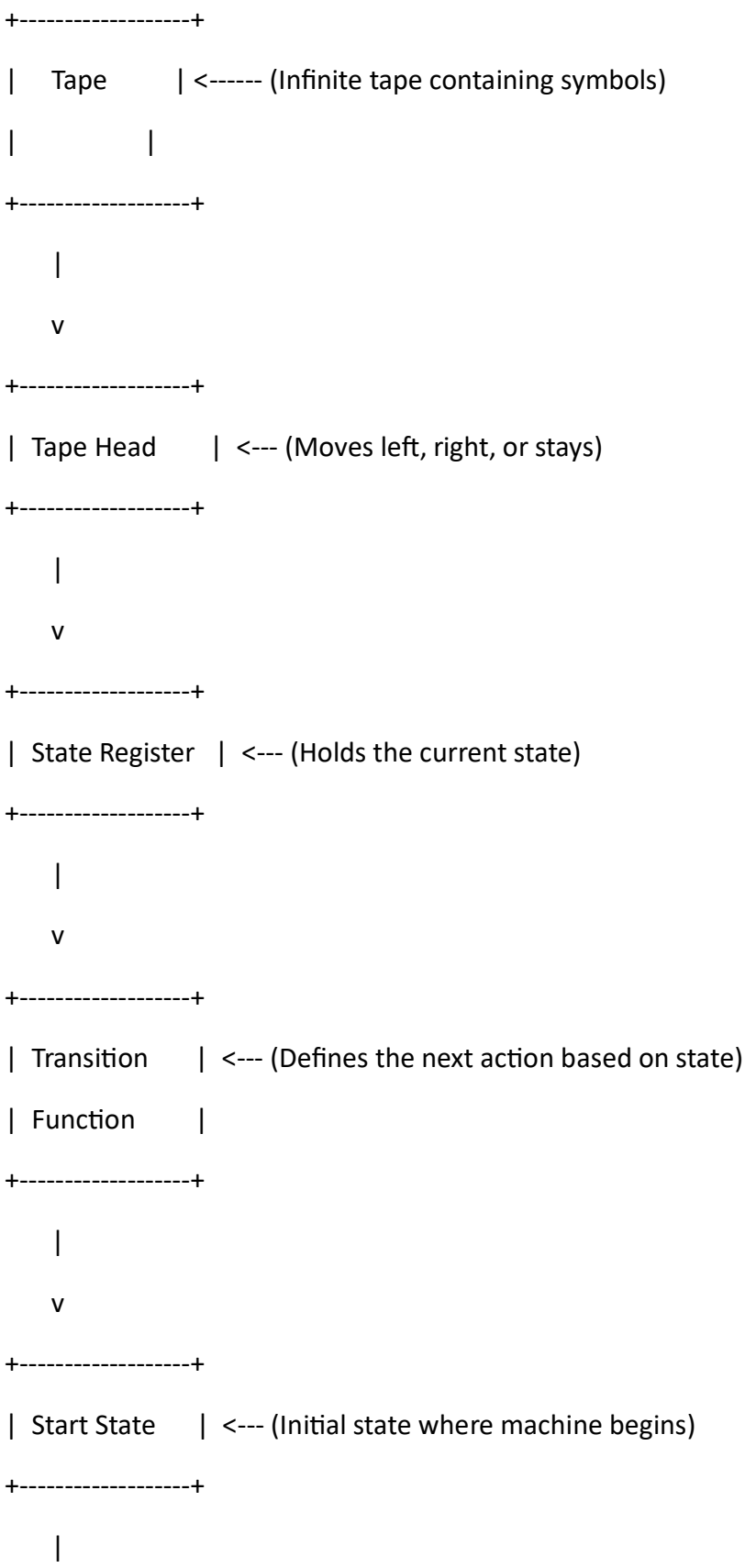
Where:

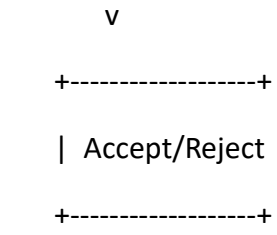
- QQ is the finite set of states,
- Σ\Sigma is the input alphabet (excluding the blank symbol),
- Γ\Gamma is the tape alphabet (including the blank symbol),
- δ\delta is the transition function,
- q0q_0 is the initial state,
- qacceptq_{\text{accept}} is the accepting state,
- qrejectq_{\text{reject}} is the rejecting state.

Diagram of a Turing Machine

Here is a simple diagram to illustrate the components of a Turing Machine:




Tape
(Infinite tape containing symbols)
Tape Head
(Moves left, right, or stays)
State Register
(Holds the current state)
Transition Function
(Defines the next action based on state)
Start State
(Initial state where machine begins)



Working of a Turing Machine:

1. Start State: The Turing Machine starts in the initial state q0q_0 and begins reading the input from the leftmost cell of the tape.
2. Transition Function: Based on the current state and the symbol read by the tape head, the transition function determines:
    - The new state to transition to.
    - The symbol to write on the current tape cell.
    - The direction to move the tape head (left, right, or stay).
3. Accepting and Rejecting:
    - The Turing Machine continues processing until it reaches an accepting state qacceptq_{\text{accept}} (indicating successful computation) or a rejecting state qrejectq_{\text{reject}} (indicating failure or halting).

Example of Turing Machine Operation:

Consider a Turing Machine that accepts strings consisting of an even number of 1's, i.e., L={0∗1∗}L = \{ 0^* 1^* \} where the number of 1's is even. The machine can proceed by:

1. Reading the first '1', changing it to a special marker (say 'X'), and moving to the right.
2. Looking for another '1', changing it to 'X' and moving back to the left.
3. Repeating this process until no more unmarked '1's are found.
4. If an even number of '1's are marked, the machine enters an accepting state; if not, it enters a rejecting state.

ANS 9. B)

Turing Machine for Language L={anbncn|n≥1}L = \{ a^n b^n c^n \mid n \geq 1 \}

We will design a Turing Machine (TM) to accept the language LL, which consists of strings in the form anbncna^n b^n c^n where n≥1n \geq 1. This language has strings where:

- The number of 'a's is equal to the number of 'b's and the number of 'b's is equal to the number of 'c's.

Steps for Designing the Turing Machine:

The Turing Machine will:

1. Check for the first 'a', replace it with an 'X', and move the head right.
2. Move right across the 'b's, and replace the first 'b' with 'Y'.
3. Move right across the 'c's, and replace the first 'c' with 'Z'.
4. Return to the leftmost unmarked 'a' to repeat the process.
5. If the machine encounters any of the following conditions, it rejects the string:
    - If the first 'a' is not found when expected.
    - If the first 'b' is not found when expected.
    - If the first 'c' is not found when expected.
    - If any character appears out of order or if the number of 'a's, 'b's, and 'c's do not match.

The machine will halt and accept when all 'a's, 'b's, and 'c's have been marked correctly and the string is consumed.

Formal Description of the Turing Machine:

The Turing Machine will have the following states and transitions:

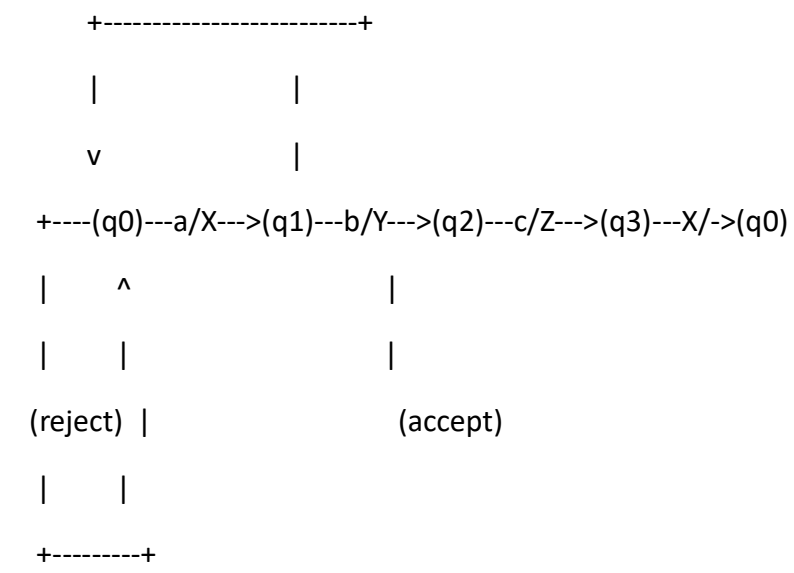
1. State q0q_0: Start state. The machine looks for the first 'a', replaces it with 'X', and moves right.
    - q0q_0, reading 'a' → write 'X', move right, transition to state q1q_1.
    - q0q_0, reading 'Y' or 'Z' → move right, stay in q0q_0 (skips over the marked characters).
2. State q1q_1: The machine looks for the first 'b', replaces it with 'Y', and moves right.
    - q1q_1, reading 'b' → write 'Y', move right, transition to state q2q_2.
    - q1q_1, reading 'X' → move right, stay in q1q_1 (skip over the 'a' portion).
3. State q2q_2: The machine looks for the first 'c', replaces it with 'Z', and moves left.

- q2q_2, reading 'c' → write 'Z', move left, transition to state q3q_3.
- q2q_2, reading 'Y' → move right, stay in q2q_2 (skip over the marked 'b' portion).

4. State q3q_3: The machine moves left across the tape, skipping over 'Y's and 'Z's, to return to the first unmarked 'a'.
    - q3q_3, reading 'X' → move right, transition to state q0q_0 (return to the unmarked 'a' to repeat the process).
    - q3q_3, reading 'Y' or 'Z' → move left, stay in q3q_3.
5. State qacceptq_{\text{accept}}: If the machine reaches the end of the tape and all symbols are marked, the machine enters the accepting state.
6. State qrejectq_{\text{reject}}: If an invalid situation occurs (e.g., the machine expects a 'b' but finds an 'a', or vice versa), it enters the rejecting state.

Transition Diagram

Below is the transition diagram for the Turing Machine that recognizes L={anbncn|n≥1}L = \{ a^n b^n c^n \mid n \geq 1 \}.



Explanation of the Diagram:

- State q0q_0: The machine begins by looking for the first 'a'. If it finds it, it replaces it with 'X' and moves to state q1q_1. If it encounters a 'Y' or 'Z', it keeps moving right until it reaches the next unmarked 'a'.

- State q1q_1: In this state, the machine looks for the first 'b'. If it finds it, it replaces it with 'Y' and moves to state q2q_2. If it encounters an 'X', it moves right and continues scanning for the next 'b'.
- State q2q_2: In this state, the machine looks for the first 'c'. If it finds it, it replaces it with 'Z' and moves left to return to the previous unmarked 'a' by transitioning to state q3q_3.
- State q3q_3: The machine moves left, skipping over any 'Y' or 'Z' encountered, and goes back to the leftmost unmarked 'a'. It then transitions to q0q_0 to repeat the process.
- Accepting State: The machine will enter the accepting state if all the 'a's, 'b's, and 'c's are replaced by 'X', 'Y', and 'Z', respectively, and there are no more characters to process.
- Rejecting State: If the machine encounters an unexpected situation (e.g., trying to find a 'b' when there are no more 'a's or trying to find a 'c' when there are no more 'b's), it will reject the string.

Example: String aabbccaabbcc

Let’s simulate the transitions for the input string "aabbcc".

Initial Configuration:

Tape: a a b b c c Head: Points to the first 'a'

1. Start (q0): Read 'a', replace it with 'X', move right.
    - Tape: X a b b c c
    - Head: Points to 'a' (second position)
2. State (q1): Read 'a', move right (skip to 'b').
    - Tape: X a b b c c
    - Head: Points to 'b' (third position)
3. State (q1): Read 'b', replace it with 'Y', move right.
    - Tape: X Y b b c c
    - Head: Points to 'b' (fourth position)
4. State (q2): Read 'b', move right (skip to 'c').
    - Tape: X Y b b c c

- Head: Points to 'c' (fifth position)

5. State (q2): Read 'c', replace it with 'Z', move left.
   - Tape: X Y b b Z c
   - Head: Points to 'b' (fourth position)
6. State (q3): Read 'b', move left (skip over 'Y').
   - Tape: X Y b b Z c
   - Head: Points to 'Y' (third position)
7. State (q3): Read 'Y', move left.
   - Tape: X Y b b Z c
   - Head: Points to 'X' (second position)
8. State (q0): Read 'X', move right (return to 'a').
   - Tape: X Y b b Z c
   - Head: Points to 'b'
9. State (q1): Read 'b', move right (skip to 'c').
   - Tape: X Y b b Z c
   - Head: Points to 'c'
10. State (q2): Read 'c', move right to accept the input since all symbols are processed.

ANS 10. A)

Turing Machine to Accept Palindromes Over {a,b}\{a, b\}

A palindrome is a string that reads the same forward and backward. We will design a Turing Machine that accepts strings which are palindromes over the alphabet {a,b}\{a, b\}.

Approach:

1. The Turing Machine starts by scanning the leftmost character of the string.
2. It replaces the first unmarked character (either 'a' or 'b') with a special marker (let's use 'X' for 'a' and 'Y' for 'b'), then moves to the right to find the corresponding matching character from the right end of the string.

3. If a matching character is found, it replaces that character with the marker ('X' or 'Y'), then moves back to the leftmost unmarked character and repeats the process.
4. If it ever finds a mismatch (e.g., the first unmarked character does not match the last unmarked character), it rejects the string.
5. The machine accepts the string if all characters are successfully paired and marked, and the tape head reaches the blank space.

Formal Description:

Let the Turing Machine MM for accepting palindromes over {a,b}\{a, b\} be defined as follows:

- States: Q={q0,q1,q2,q3,q4,qaccept,qreject}Q = \{ q_0, q_1, q_2, q_3, q_4, q_{\text{accept}}, q_{\text{reject}} \}
- Alphabet: Σ={a,b}\Sigma = \{a, b\}
- Tape Alphabet: Γ={a,b,X,Y,#}\Gamma = \{a, b, X, Y, \# \} (including 'X' for 'a', 'Y' for 'b', and the blank symbol #\#).
- Start State: q0q_0
- Accepting State: qacceptq_{\text{accept}}
- Rejecting State: qrejectq_{\text{reject}}

Transition Function:

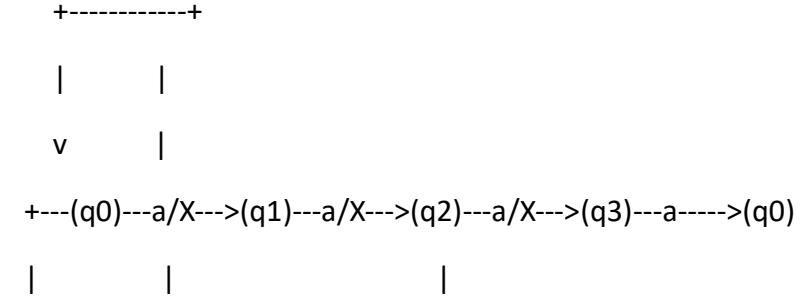
1. State q0q_0 (Looking for the leftmost unmarked symbol):
    - q0,a→X,R,q1q_0, a \to X, R, q_1 (If 'a', mark with 'X', move right, and transition to q1q_1).
    - q0,b→Y,R,q1q_0, b \to Y, R, q_1 (If 'b', mark with 'Y', move right, and transition to q1q_1).
    - q0,#→#,S,qacceptq_0, \# \to \# , S, q_{\text{accept}} (If the tape is empty, accept).
2. State q1q_1 (Moving right to find the matching character):
    - q1,a→a,R,q1q_1, a \to a, R, q_1 (Move right, skip over unmarked 'a').
    - q1,b→b,R,q1q_1, b \to b, R, q_1 (Move right, skip over unmarked 'b').
    - q1,X→X,R,q2q_1, X \to X, R, q_2 (Found an 'X', move right to find corresponding character).

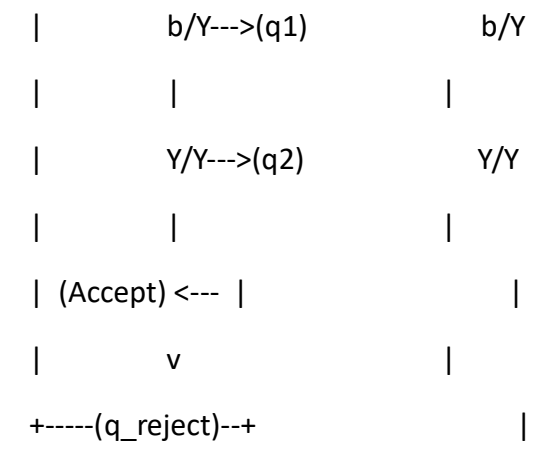
- q1,Y→Y,R,q2q_1, Y \to Y, R, q_2 (Found a 'Y', move right to find corresponding character).

3. State q2q_2 (Looking for the rightmost unmarked symbol):
    - q2,a→X,L,q3q_2, a \to X, L, q_3 (If 'a' is found, mark with 'X' and move left).
    - q2,b→Y,L,q3q_2, b \to Y, L, q_3 (If 'b' is found, mark with 'Y' and move left).
    - q2,#→#,S,qrejectq_2, \# \to \# , S, q_{\text{reject}} (If no matching symbol found, reject).
4. State q3q_3 (Moving left to find the leftmost unmarked symbol):
    - q3,a→a,L,q3q_3, a \to a, L, q_3 (Move left, skip over unmarked 'a').
    - q3,b→b,L,q3q_3, b \to b, L, q_3 (Move left, skip over unmarked 'b').
    - q3,X→X,L,q0q_3, X \to X, L, q_0 (If 'X' is found, go back to q0q_0 to repeat the process).
    - q3,Y→Y,L,q0q_3, Y \to Y, L, q_0 (If 'Y' is found, go back to q0q_0 to repeat the process).
5. Final Acceptance (State qacceptq_{\text{accept}}):
    - If all symbols are marked and the machine reaches the blank symbol #\#, it enters the accepting state.
6. Rejection (State qrejectq_{\text{reject}}):
    - The machine enters the rejecting state if it finds a mismatch or if an unexpected situation occurs (e.g., reaching a blank symbol when not expected).

Transition Diagram

Below is the transition diagram for the Turing Machine that recognizes palindromes over {a,b}\{a, b\}.





How the Turing Machine Works:

Let's simulate the Turing Machine for an example string: abba.

1. Start State (q0):
    - The machine reads the first 'a', replaces it with 'X', and moves right to q1q_1.
2. State (q1):
    - The machine moves right and finds 'b', then moves to q2q_2.
3. State (q2):
    - The machine reads the last 'b', replaces it with 'Y', and moves left to q3q_3.
4. State (q3):
    - The machine moves left and finds the 'X' (previously marked 'a'). It returns to q0q_0.
5. State (q0):
    - The machine reads the 'X' (the first marked 'a') and moves right to the next unmarked character, which is 'Y'.
6. State (q1):
    - The machine checks if it has reached the blank symbol and accepts the string.

This process works similarly for all palindromes, and the Turing Machine will accept the input if it is a palindrome and reject it if it is not.

ANS 10. B)

Recursively Enumerable Language (RE Language)

A recursively enumerable language (RE) is a type of formal language that can be recognized by a Turing machine. It is also known as a Turing-recognizable language. These languages are the ones for which there exists a Turing machine that can accept any valid string in the language and either halt or run indefinitely when the string is not in the language. In other words, a language is recursively enumerable if there exists a Turing machine that will:

- Accept all strings that belong to the language (halt in the accepting state).
- Reject or never halt for strings that do not belong to the language.

Formal Definition:

A language LL is recursively enumerable if there exists a Turing machine MM such that:

- For any string w∈Lw \in L, MM halts and accepts ww.
- For any string w∉Lw \notin L, MM either halts and rejects ww or runs forever (does not halt).

Key Points about Recursively Enumerable Languages:

1. Decidability vs Recognizability:
    - A decidable language is one for which there exists a Turing machine that halts for every input and correctly decides whether the input is in the language or not (i.e., it always halts with either "accept" or "reject").
    - A recursively enumerable language is one for which there exists a Turing machine that accepts strings in the language, but for strings not in the language, the machine may run forever (i.e., it might not halt).
2. Examples:
    - The set of all valid C programs is a recursively enumerable language because a Turing machine can recognize whether a string is a valid C program, but it may never halt if the program contains an infinite loop.
    - The Halting Problem itself is a classic example of a problem that is RE but not decidable. The language of Turing machines that halt on a given input is recursively enumerable because we can simulate the Turing machine and see if it halts. If it halts, we accept; if it doesn’t, we may never halt.

3. Properties of RE Languages:
    - Closure Properties: The class of recursively enumerable languages is closed under union, but not under intersection or complementation. That is, while the union of two RE languages is also RE, the intersection or complement of two RE languages might not be RE.
    - Non-Closure under Complement: The complement of a recursively enumerable language might not be recursively enumerable. This is a key distinction from decidable languages, which are closed under complementation.

Multi-Tape Turing Machine

A multi-tape Turing machine is an extension of the standard (single-tape) Turing machine. In this model, there are multiple tapes (instead of just one), and each tape has its own read/write head. The machine operates in much the same way as a single-tape Turing machine, except now it can access multiple tapes simultaneously, each one operating independently of the others. The machine is still governed by a finite set of states, and it has a transition function that dictates how the head of each tape moves based on the current state and the symbols read from the tapes.

Components of a Multi-Tape Turing Machine:

1. Multiple Tapes: A multi-tape Turing machine has more than one tape. Each tape is infinite in length and serves as the storage for the machine.
2. Multiple Tape Heads: Each tape has its own tape head that can read, write, and move independently.
3. Transition Function: The transition function in a multi-tape Turing machine is slightly more complex than that of a single-tape machine because it takes the symbols read from all tapes and uses them to decide the next state, the symbols to write on each tape, and the direction in which to move each tape head.

Formal Definition:

A multi-tape Turing machine is defined as a 7-tuple (Q,Σ,Γ,δ,q0,qaccept,qreject)(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}}), where:

- QQ is the finite set of states.
- Σ\Sigma is the input alphabet.
- Γ\Gamma is the tape alphabet (which includes symbols that can be written on the tape).
- δ\delta is the transition function that now operates on multiple tapes.

- q0q_0 is the start state.
- qacceptq_{\text{accept}} is the accepting state.
- qrejectq_{\text{reject}} is the rejecting state.

Key Features of Multi-Tape Turing Machines:

1. More Efficient Computation:
    - Multi-tape Turing machines can often perform computations more efficiently than single-tape machines. For example, they can simulate certain operations, like copying data, more quickly by writing to one tape and reading from another.
2. Computational Power:
    - A multi-tape Turing machine is more powerful than a single-tape machine in terms of time complexity, but both have the same computational power in terms of the class of languages they can recognize. Any language that can be recognized by a multi-tape Turing machine can also be recognized by a single-tape Turing machine (though potentially with a different time complexity). This is due to the Church-Turing thesis which states that the single-tape Turing machine is as powerful as any computational model, including multi-tape Turing machines.
3. Simulating Multi-Tape Turing Machine on a Single-Tape Machine:
    - While multi-tape Turing machines may be more efficient, a single-tape Turing machine can still simulate a multi-tape machine. However, the simulation may take more time. For instance, a multi-tape Turing machine that works in O(n)O(n) time can be simulated by a single-tape Turing machine in O(n2)O(n^2) time.

Example:

Consider a multi-tape Turing machine that copies the input string to a second tape:

- Tape 1: Contains the input string.
- Tape 2: Is initially empty and will hold a copy of the input string.

The multi-tape Turing machine works by scanning the input string on Tape 1, reading each symbol, and copying it to Tape 2. The heads on both tapes move simultaneously, and the machine halts once the entire string has been copied.

A single-tape Turing machine would require additional steps to simulate this process, such as moving back and forth between tapes and using additional markers.

Summary:

1. Recursively Enumerable Languages (RE):
    - RE languages are those for which a Turing machine can recognize and accept strings that belong to the language.
    - For strings not in the language, the machine may run forever without halting.
    - These languages are not necessarily decidable because the machine may not halt on all inputs.
2. Multi-Tape Turing Machines:
    - A multi-tape Turing machine extends the concept of the standard Turing machine by having multiple tapes and heads.
    - These machines are more efficient in certain tasks but do not increase the computational power of Turing machines.
    - A multi-tape machine can be simulated by a single-tape machine, though with a potential increase in time complexity.