Vishakha Agrawal- 2023101040

18-9-24

AT - Theory assignment 2

Ans1. RTP:

$$L = \{ xy \mid x, y \in \{0,1\}^*, |x| = |y|, x \neq y \}$$

is a CFL

Proof:

To show that L is context free we need to show that there exists a grammar that can generate all the strings in the language or construct a PDA that recognizes it.

The language consists of all strings of the form xy where

$x, y \in \{0,1\}^*$.

$|x| = |y|$

$x \neq y$ x & y must always differ in atleast one position.

Let the grammar be defined as follows:

$S \rightarrow AB \mid BA$

$A \rightarrow 0 \mid 0A0 \mid 0A1 \mid 1A0 \mid 1A1$

$B \rightarrow 1 \mid 0B0 \mid 0B1 \mid 1B0 \mid 1B1$

Variables: S, A, B

Terminals: 0, 1

Start state: S

S starts by combining A & B. A & B in return can recursively generate longer & more complex strings. we have to show that L(G) is the same as L.

If we observe the structure of the strings generated by the grammar G, they are of the form:

$$w_1 x w_2 v_1 y v_2$$

$w_1, w_2, v_1, v_2 \in \{0,1\}^*$ st

$|w_1| = |w_2| = k$

$|v_1| = |v_2| = l$

$x, y \in \{0,1\}^*$ st $x \neq y$

The language L can be generated from the grammar G.

- The condition for a string s to belong to L is that if for some i, x & y differ in the i^{th} position. The grammar G ensures that x & y differ at some position(s) as the production rules for A & B to generate strings of equal length, the terminals allow us to generate strings of any length. The interchangeability of 0 & 1 in A & B allows for mismatch in strings. The generation can be thought of as generating the necessary mismatch between x & y & filling in the remaining characters to follow the $w_1 x w_2 v_1 y v_2$ structure.

- The generated language $w_1 x w_2 v_1 y v_2$ can be subjected to nested induction over k & l. For each pair (x, y) $x \neq y$ we can consider all possible cases where mismatch occurs at different positions:

- Since the substrings w_2 and v_1 are independent of the rest of the string, swapping symbols between them demonstrates that the grammar does not pose unnecessary restrictions, while generating all valid strings. The flexibility makes the grammar robust to handle variations in how the substrings are constructed.
- Thus, due to the fact that G does not place any additional constraints on the strings it generates, x & y can differ at any position in their halves ensuring that $L(G) = L$.

∴ L is context free.

Ans 2. Pumping lemma for CFL's

If A is a CFL, ∃ p st if $|S| \geq p$

$S = uvxyz$ st

i) for each $i \geq 0$ $uv^i x y^i z \in A$

ii) $|vy| > 0$ &

iii) $|vxy| \leq p$

1. $L = \{a^n b^m \mid m = n^2\}$

Let p be the pumping length. Let us choose the string $s = a^p b^{p^2} \in L$.

∵ $|vxy| \leq p$, the substring vxy must be entirely contained within the first p characters of s which will be all a's.

Let us pump the string by changing the value of i in $uv^i xy^i z$

Original String : $s = uv^1 xy^1 z = a^p b^{p^2}$

Pumping for $i = 2$ & $s = uv^0 xy^2 z$
This increases # of a's but keeps # of b's unaffected.

Let v contain k_1 a's
$|v| = k_1$
& $|y| = k_2$
After pumping say the string has n' a's

$$n' = p + k_1 + k_2$$
while $m = p^2$ (# of b's)

For the pumped string to belong to L:
$$m = (n')^2$$
$$p^2 = (p + k_1 + k_2)^2$$

$$p^2 = p^2 + 2p(k_1 + k_2) + (k_1 + k_2)^2$$
$$\Rightarrow 2p(k_1 + k_2) + (k_1 + k_2)^2 = 0$$
$$\Rightarrow k_1 + k_2 = 0$$

However, this contradicts the fact that $|vy| > 0$ & that atleast v or y contain one character. Hence, $k_1 + k_2 > 0$

Thus our assumption that L is CF is false.
∴ L is not context-free.

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

Let p be the pumping length. Let us consider

$s = 0^p 1^p 0^p 1^p \in L$

$|s| = 4p$

The substring we choose to pump can either contain only one type of symbol or contain both types of symbols.

Case 1:

Let us assume that v & y consist only of one type of symbol meaning either 0's or 1's.

After pumping, the string $= uv^{\text{for } i=2}xy^2z$ will look like

$0^{p+k} 1^p 0^p 1^p$

or $0^p 1^{p+k} 0^p 1^p$

or $0^p 1^p 0^{p+k} 1^p$

or $0^p 1^p 0^p 1^{p+k}$

In all these cases we obtain strings where one of the four blocks has more ~~str~~ symbols than the others. The pumped string will no longer belong to L.

case 2:

Let us consider the case where v & y contain more than one type of symbol & that v and y span parts of the 0 & 1 blocks.

∴ v any row contains both 0 & 1, pumping these substrings will break the correct ordering of symbols in the string. The resulting string will no longer maintain the $0^n1^n0^n1^n$ ordering.

The pumped string does not belong to the language.

∴ L is not CF.

Ans3.

$E \rightarrow I$

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow (E)$

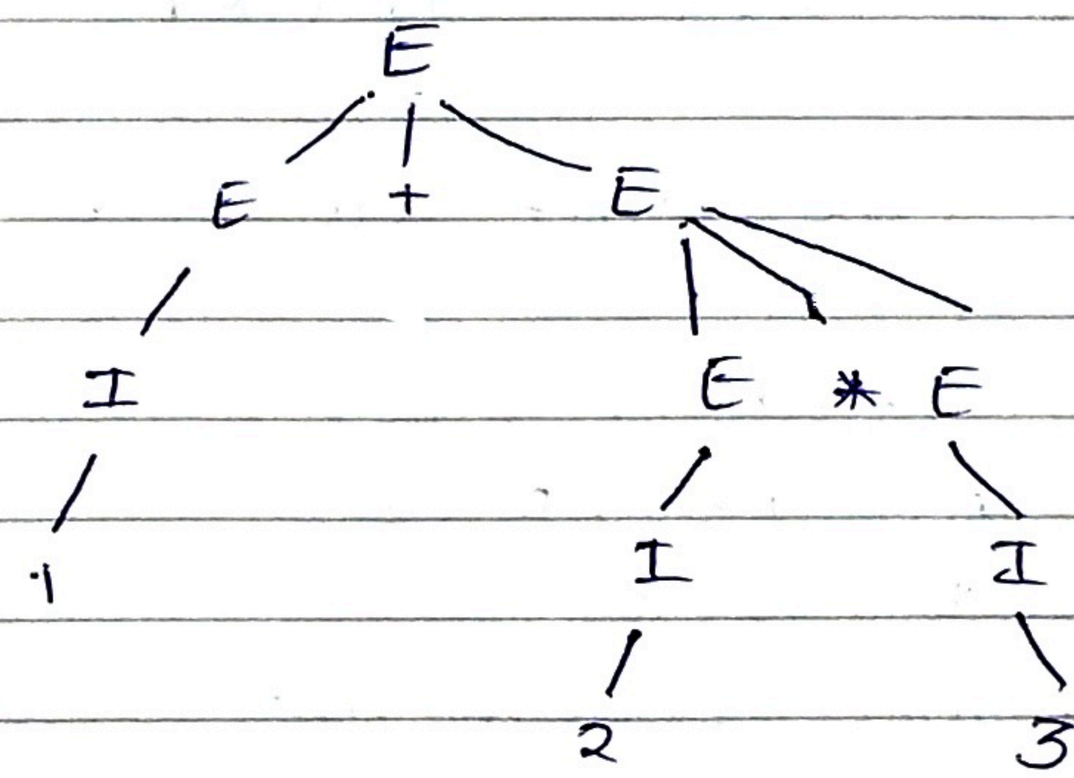
$I \rightarrow \varepsilon \mid 0 \mid 1 \mid 2 \mid \ldots \mid 9$.

This grammar provides a way of expressing arithmetic expressions with no clear enforcement of precedence between + & * except for the fact that parantheses can be used to clarify an order.

We can show that this grammar is ambiguous if a string can be generated using it has more than 1 leftmost derivation or parse tree.

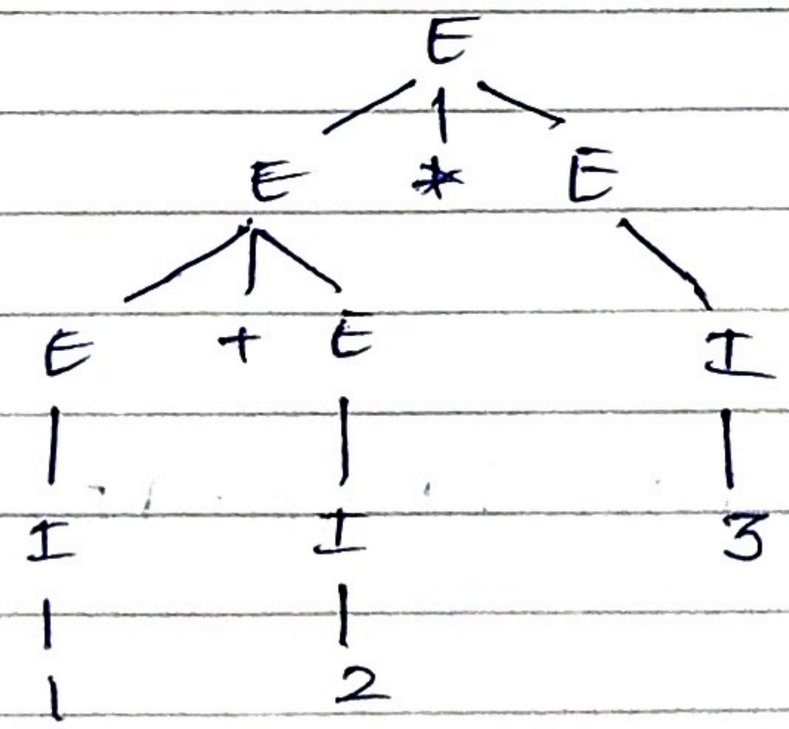
Let us consider 1 + 2 * 3

This can yield the following parse tree



$E \rightarrow E+E \rightarrow I+E*E \rightarrow 1+I*I \rightarrow 1+2*3$

We can obtain a parse tree where * is performed first



$E \rightarrow E*E \rightarrow E+E*E \rightarrow E+E*I$
$\rightarrow E+E*3 \rightarrow I+I*3 \rightarrow 1+2*3$

Because more than one parse trees exists for $1+2*3$, the grammar is ambiguous.

Ans4. Converting $A \rightarrow BAB \mid B \mid \varepsilon$

$B \rightarrow 00 \mid \varepsilon$ to CNF

Let us add a new start variable S_0 such that

$S_0 \rightarrow A$

$A \rightarrow BAB \mid B \mid \varepsilon$

$B \rightarrow 00 \mid \varepsilon$

The start variable should not occuron the RHS of anule.

Remove ε rules of the form $A \rightarrow \varepsilon$

Removing $A \rightarrow \epsilon$

$S_0 \rightarrow A \mid \epsilon$

$A \rightarrow BAB \mid B \mid BB$

$B \rightarrow 00 \mid \epsilon$

Removing $B \rightarrow \epsilon$

$S_0 \rightarrow A \mid \epsilon$

$A \rightarrow BAB \mid BA \mid AB \mid A \mid BB \mid B$

$B \rightarrow 00$.

If S is the start variable, rules of the form $S \rightarrow \epsilon$ are valid

Removing unit rules of the form $A \rightarrow B$

Removing $A \rightarrow A$

$S_0 \rightarrow A \mid \epsilon$

$A \rightarrow BAB \mid BA \mid AB \mid BB \mid B$

$B \rightarrow 00$

Removing $A \to B$

If a rule $B \to u$ occurs. (u is a string of terminals & variables) we add $A \to u$ unless it is already removed

$S_0 \to A1 \in$

$A \to BAB \mid BA \mid AB \mid BB \mid 00$

$B \to 00$

Removing $S_0 \to A$

$S_0 \to BAB \mid BA \mid AB \mid BB \mid 00 \mid \in$

$A \to BAB \mid BA \mid AB \mid BB \mid 00$

$B \to 00$

Remove long rules of the form $A \to u_1 \cdots u_k$ $k \geq 3$ with $A \to u_1 A_1$

Replace any terminal u_i with $u_i \to U_i$

Replace string of terminals UO with $U \to 0$

$S_0 \to BAB \mid AB \mid BA \mid UU \mid BB \mid \in$

$A \to BAB \mid BA \mid AB \mid UU \mid BB$

$B \to UU$

$U \to 0$

To obtain the correct form of rules for S_0 & A (with only two variables on RHS) add $V \to AB$.

$S \rightarrow BV \mid BA \mid AB \mid UU \mid BB \mid \epsilon$

$A \rightarrow BV \mid BA \mid AB \mid UU \mid BB$

$B \rightarrow UU$

$U \rightarrow 0$

$V \rightarrow AB$

This is the grammar converted to CNF.

Ans Definition of PDA with 1 stack

$(Q, \Sigma, \Gamma, \delta, q_0, F)$

where

$\delta : Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma_\epsilon \rightarrow P(Q \times \Gamma_\epsilon)$

~~A two stack PDA can be defined as follows:~~

A Turing machine is a 7 tuple

$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$

$\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L \times R\}$

A two stack PDA can be simulated by a TM ~~with two tapes:~~.

We want to formally show that a PDA with 2 stacks is more powerfull than one with a single stack by showing that a PDA with two stacks is computationally equivalent to a Turing machine.

This can be done if we show that:

A TM can simulate a two-stack PDA
A PDA with two stacks can simulate a TM.

Simulating TM using 2 stack PDA:
A TM has an infinite tape and a tape head that reads, writes, and moves along the tape.

We can simulate the tape using two stacks:

Let one stack (say stack 1) represent the portion of the tape to the left of the tape head.
Let another stack (say stack 2) represent the tape cell currently under the tape head and the portion to the right of it.

Tape: $\cdots a_3, a_2, a_1, a_0, a_1', a_2', a_3' \cdots$

↑ Stack 1 ↑ Stack 2.

a_0 is currently under the tape head.

Simulating TM operations:

Reading & writing:
the PDA reads a new symbol a_0 from top of stack 2, to write a new symbol b, it pops a_0 & pushes b onto stack 2.

Moving tape head:

Right:

Pop the top of stack 2 (under the tape head) & push the symbol onto stack 1.

The new top of stack 2 is the symbol to the right of the current.

If stack 2 is empty after popping, push a blank symbol to represent empty tape cell.

Left:

Pop the top of stack 1 (symbol to left of tape head) & Push this onto stack 2. The symbol just moved becomes the new symbol under the tape head.

If stack 1 is empty, after popping, push blank onto stack 1.

The PDAs # states must change to simulate TM's transitions.

The stack alphabet includes the the TM's tape symbols (Γ_1, Γ_2) and blank.

For each TM transition $(q, a) \rightarrow (q', b, D)$ the PDA transitions to:

Pop a from stack 2

Push b to stack 2

Move the tape in either direction D

Change state to q'.

A new PDA $P = (Q', \Sigma, \Gamma', \delta', q_0', F')$ with two stacks can simulate M.

Also, a TM is atleast as powerful as a two stack PDA as a TM is exactly as powerful as a two tape TM. The usage of both tapes can be restricted to simulate stacks.

To push ~~onto~~ onto the stacks, move the tape head to the end of the region simulating the stack & write the symbol being pushed onto the ~~tape~~ and adjust the head. To pop, ~~no~~ move the tape head to the end and erase the symbol.

Since a TM can simulate a PDA with two stacks and vice-versa, a two stack PDA is computationally equivalent to a TM.

Thus, a PDA with 2 stacks is more powerful than a PDA with one stack.

Ans6. We want to do the following:
We weant to use 3 tapes:
T_1 : Stores binary representation of x & store final computation
T_2 : Store intermediate values such as duplicate of x
T_3 : Keep track of carry bits during binary addition.

Idea:
If 'x' is even, then we replace its LSB (which is 0) with blank to simulate division by 2 & stop.
If x is odd, duplicate 'x' on the second tape, add a terminating 0 in tape 1 (to get 2x)
Add leading 0 to the first tape's end & 00 to the second one to align them for addition to obtain 3x. Then add 1 to the original tape.

Description of machine:

1) The machine starts by ensuring that the first character is #. If then scans the remaining binary string for x. If the LSB is 0, x is even, the machine replaces the 0 with a blank and halts.
If LSB is 1, machine continues with process to compute 3x+1

2) The machine writes #00 on tape 2 to prepare for multiplication (for correct addition alignment)
The contents of tape 1 are copied to Tape 2 after #00

3) The machine shifts contents of tape 1 one position to the right (multiplying 2 by x)
After the shift the machine makes the new first symbol after the #, a 0 to ensure alignment.

4) Replace the first blank on Tape 1 with a 0 to extend the number for addition.

5) The machine writes # on tape 3 for each symbol on tape 1 (except the last one) & rightmost # with 0.

6) Add contents of tape 2 (shifted) to tape 1:
The machine starts from rightmost digit on tape 2 that has not been crossed-off. For each corresponding digit (s) from both tapes write the result on to the corresponding place on tape 1 (correct # of spaces to the left of first blank) & carry to tape 3.

If all 3 bits are 0, write 0 on T_1, ~~T_3 with 0~~ T_3 with 0 on rightmost #.
If only one bit is 1, write 1 to the right place on T_1 & replace the rightmost # on T_3 with 0
If two bits are 1, write 0 to the correct place on T_1 & replace the rightmost # on T_3 with 1.
If 3 bits are 1, write 1 to T_1. and set carry bit to T_3 to 1.

After each addition, rightmost digit on T_2 is crossed off to show its been added.

After computing 3x, the machine adds 1 to the result by starting at the rightmost bit of T_1. If the bit is 0, change to 1 & halt, If bit is 1, change to 0 & carry the one to next bit.

If the first symbol after # on T_1 is 0, machine deletes it & shifts the remaining number to avoid unnecessary leading 0s.
The machine halts.

Ans 7. 1. $R_1 \cup R_2$

Let M_1 & M_2 decide R_1 & R_2 respectively, (We want to build a TM deciding $L_1 \cup L_2$ say M'.

M' runs on input w. (string)
Run M_1 on w. to check if $w \in R_1$,
If M_1 accepts, accept.

If M_1 rejects. ($w \notin R_1$) run M_2 on w to check if $w \in L_2$. If M_2 accepts, accept

Reject if both M_1, M_2 reject $\therefore w \notin R_1$ & $w \notin R_2$

M' accepts w if M_1 or M_2 accepts.
Rejects if both reject.

∴ $R_1 \cup R_2 = L(M')$
Recursive lang. closed under union.

2. $RE_1 \cup RE_2$

Let M_{RE_1}, M_{RE_2} recognize RE_1 & RE_2

L' denote the union of RE_1 & RE_2
We have to design a TM M' that recognizes the union.

On input w:
The machine M_{RE_1} is simulated on w for a single step by M'. Then M' simulates the run of M_{RE_2} on w. M_{RE_1} & M_{RE_2} step by step alternating between them.

This is crucial because one or both machines could loop forever. By alternating we ensure acceptance by either machine in finite steps.

If either machine accepts ⇒ $w \in RE_1$ or $w \in RE_2$ accept
Reject if both halt & reject as $w \notin RE_1$ and $w \notin RE_2$
Loop if both machines loop

∴ $RE_1 \cup RE_2$ is recursively enumerable.

3. $R_1 \cup RE_2$

On input w:
Run the decider ~~for~~ M_1 for R_1 on w
If M_1 accepts, accept
If M_1 rejects, run accepter M_2 for RE_2 on w.
If M_2 accepts, accept

Termination:
If $w \in R_1$, we accept
If $w \notin R_1$, $w \in RE_2$ accept
If $w \notin R_1 \cup RE_2$, M_1 rejects and M_2 may loop infinitely.

$R_1 \cup RE_2$ is recursively enumerable.

4. $R_1 \cap RE_2$

On input w:
Run the decider M_1 for R_1 on w
If M_1 rejects, reject.
($\because w \notin R_1$, $w \notin R_1 \cup RE_2$)
If M_1 accepts, run accepter M_2 for RE_2 on w
If M_2 accepts, accept

If M_2 does not accept, the machine may not halt (loop).

Termination:

If $\omega \in R_1 \cap RE_2$ M_1 & M_2 accept, accept
If $\omega \notin R_1$, M_1 rejects, reject
If $\omega \in R_1$ but $\omega \notin RE_2$, M_2 may loop,
So we loop:

$R_1 \cap RE_2$ is recursively enumerable.

s 8. L_1 is recognized by M_1,
L_2 by M_2
$S_1 \in L_1$ $S_2 \in L_2$
$S_3 = S_1 S_2 \; S_1 S_2 \circ \cdots \circ$

Construct M_3 to read S_3 & extract S_1 & S_2

Tape 1: Contains S_3
Tape 2: To simulate M_1, to recognizes,
Tape 3: To simulate M_2 to recognize S_2

Position the head of tape 1 on start of S_3

Steps: S_1
1) Copy Read the characters from tape 1 and write it to tape 2.
After copying each character, run M_1 on contents of tape 2.
If M_1 accepts., then the substring on tape 2 is S_1.
Move to the next segment for S_2.
If the end of tape 1 is reached before

M_1 accepts, halt & reject.

2) Extract S_2 :

After recognizing S_1, the head will ideally be placed at the start of the next segment S_2.

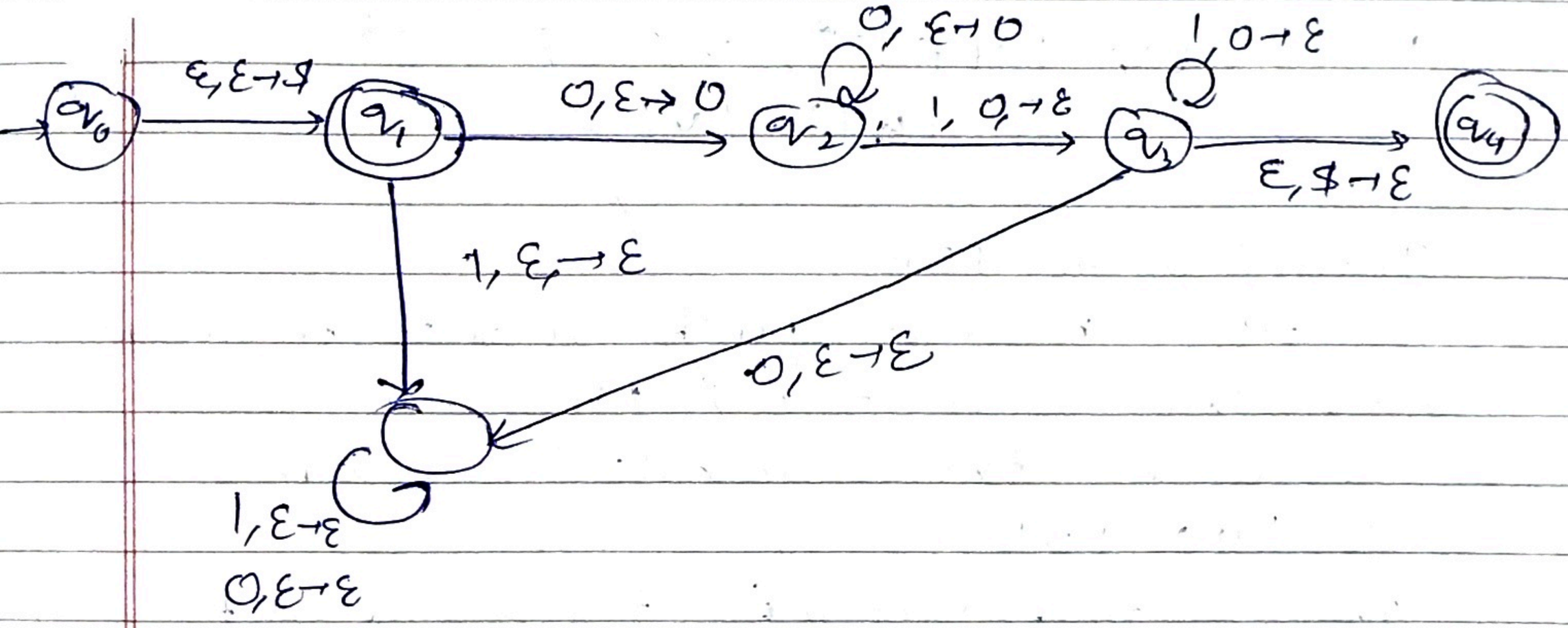
Read the characters from tape 1 & copy to tape 3.
After copying each character, run M_2 on tape 3.
If M_2 accepts, record S_2 as the second recognized substring.
Move to the next segment to confirm repitition.
If M_2 rejects, continue copying until M_2 accepts.
If the tape ends before M_2 accepts, halt & reject.

While the tape does not end:
After recognizing S_1 & S_2 once, confirm the pattern.
Repeat step 1 to check if the next portion of the input matches S_1
Repeat Step 2 to check if next portion of input matches S_2.

If the entire tape 1 is successfully read & matches $S_1 S_2$ repetitions, S_1 & S_2 have been successfully extracted.

Halt & accept.

Ans 9. $L = \{ 0^m 1^n \mid m \geq 0 \}$



Intuition:

Empty string must be accepted that's why q_1 is final.

The rest of the non-empty string are dealt normally.

To enforce determinism, if str s starts with 1, go to invalid state & stay there.

In q_3 if a 0 is encounterd after already counting the 1's, the string does not follow the given format. Therefore go to invalid state. From q_3 to accept state, once input is read, nothing is on the input tape. The stack is empty.

Ans 10. $\Sigma = \{0, 1\}$

$B = \{ xyx \mid x \in \Sigma^*, y \in \Sigma^+, |\Sigma^+|, \varepsilon, |x| \geq |y| \}$

$G = (V, \Sigma, R, S)$

CFG:

$A \rightarrow 0A0 \mid 0B1 \mid 1A0 \mid 1B1$

$B \rightarrow 0B0 \mid 0B1 \mid 1B0 \mid 1B1 \mid 0 \mid 1 \mid \varepsilon$

$V = \{A, B\}$

$\Sigma = \{0, 1\}$

$S = A$

R = set of production rules

Explanation: The productions for A builds the string from both ends. This ensures we are able keep track of the middle of the string. The productions for B indicate that a 1 has been placed somewhere in the second half. Once we get B, it ensures that we add a 1 to the second half of our string the continue building the string symmetrically or terminate. Thus we can terminate Once we reach a B in our derivation.