Module 1

Q1a. Define the following terms: i) **String**: A string is a finite sequence of symbols taken from a given alphabet. For example, if the alphabet is {a, b}, a possible string is "aab".

ii) **Language**: A language is a set of strings formed from an alphabet. For example, the set of all strings over the alphabet {a, b} is a language.

iii) **Alphabet**: An alphabet is a non-empty finite set of symbols. For example, {a, b} is an alphabet, where ‘a’ and ‘b’ are the symbols.

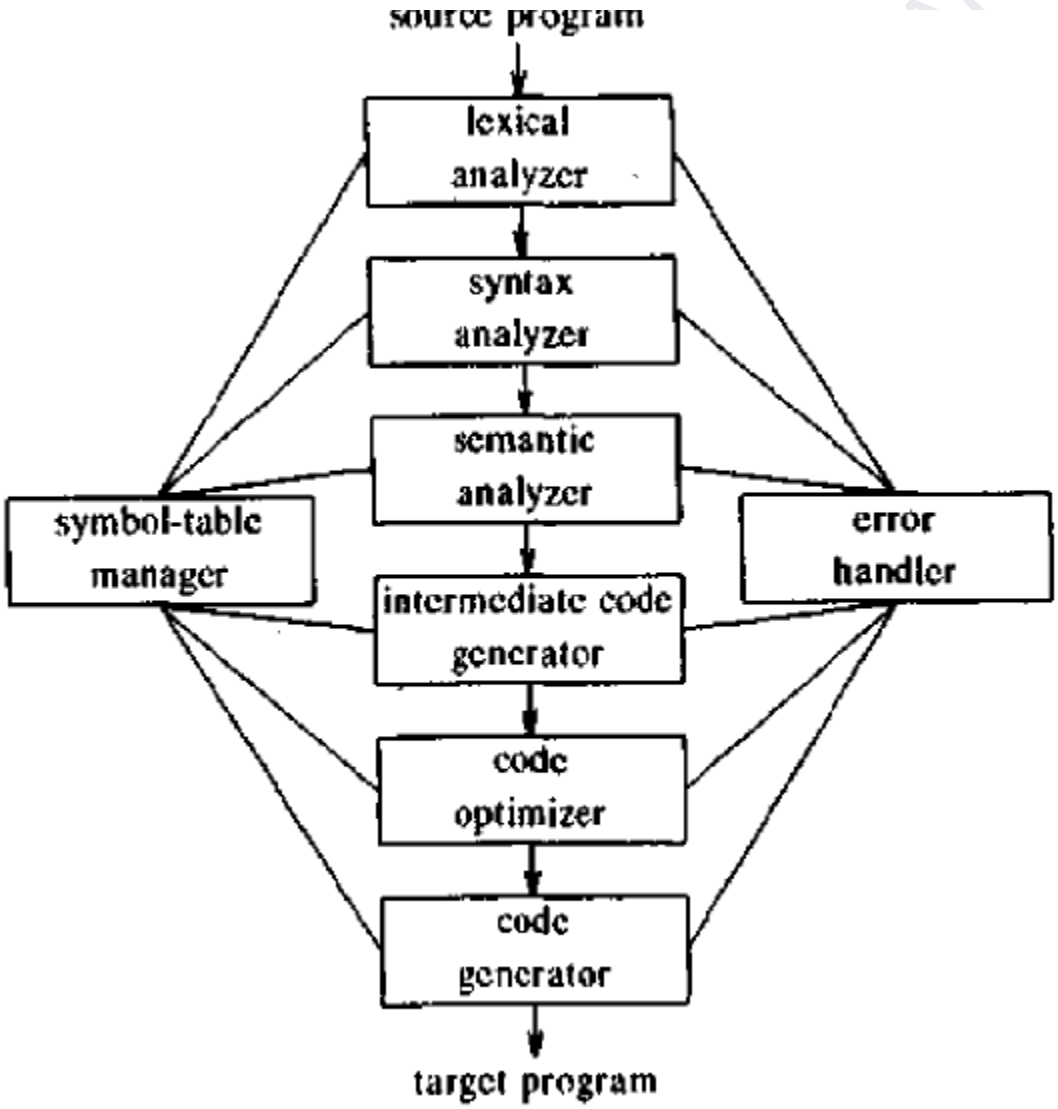
iv) **Length of a string**: The length of a string is the number of symbols in the string. For example, the length of the string "aab" is 3.

Q1b. Explain the various phases of a compiler with a neat diagram.

A **compiler** translates a high-level language program into machine language. It has several phases:

1. **Lexical Analysis**: The input program is converted into tokens by removing whitespace and comments. Each token represents a basic element, such as an identifier or keyword.
    - **Output**: Sequence of tokens.
2. **Syntax Analysis**: The tokens are analyzed according to the grammar rules of the language to generate a parse tree.
    - **Output**: Parse tree or abstract syntax tree (AST).
3. **Semantic Analysis**: This phase checks for semantic consistency, ensuring that operations are valid (e.g., type checking).
    - **Output**: Annotated syntax tree with types and other semantic information.
4. **Intermediate Code Generation**: An intermediate representation (IR) of the source code is generated. This is independent of the target machine.
    - **Output**: Intermediate code (e.g., three-address code).

5. **Code Optimization**: The intermediate code is optimized to improve the performance by reducing redundant calculations, loop optimizations, etc.
    - **Output**: Optimized intermediate code.
6. **Code Generation**: The final machine code or assembly code is generated for the target architecture.
    - **Output**: Machine code or assembly code.
7. **Code Linking and Loading**: The generated code is linked with libraries and other object files to create the final executable file.
    - **Output**: Executable code.



Q1c. Define DFA and design a DFA to accept the following language:

- **DFA (Deterministic Finite Automaton)**: A DFA is a state machine that accepts or rejects strings of a language. It consists of:
    - A finite set of states.
    - A set of input symbols (alphabet).
    - A transition function.
    - An initial state.
    - A set of final states.

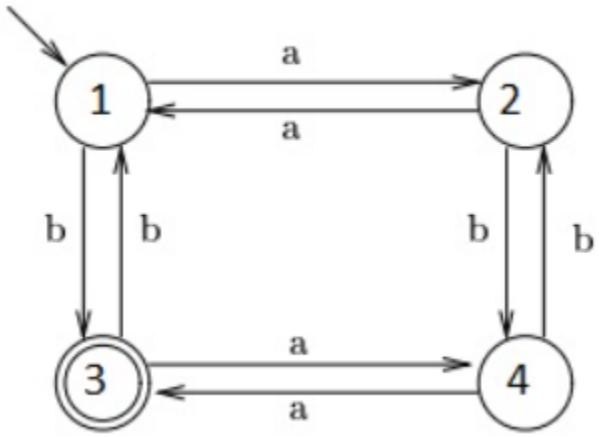
i) To accept strings having an even number of a's and an odd number of b's:

- **States**:
    - q0q_0q0: Even number of a's and even number of b's.
    - q1q_1q1: Even number of a's and odd number of b's.
    - q2q_2q2: Odd number of a's and even number of b's.
    - q3q_3q3: Odd number of a's and odd number of b's.

Transitions:

- From q0q_0q0, an 'a' takes the automaton to q2q_2q2, a 'b' takes it to q1q_1q1.
- From q1q_1q1, an 'a' takes it to q3q_3q3, a 'b' takes it to q0q_0q0.
- From q2q_2q2, an 'a' takes it to q0q_0q0, a 'b' takes it to q3q_3q3.
- From q3q_3q3, an 'a' takes it to q1q_1q1, a 'b' takes it to q2q_2q2.

The accepting state is q1q_1q1 (even a's, odd b's).



ii) To accept strings of a’s and b’s not having the substring "aab":

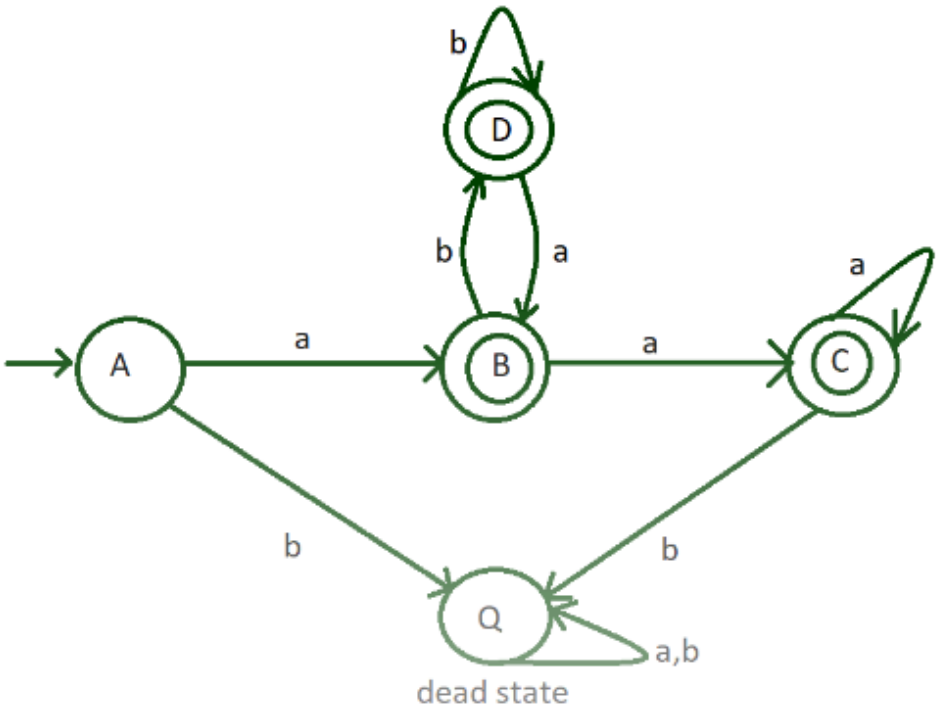
We need to build a DFA that rejects any string containing "aab" and accepts all others.

- **States**:
  - q0q_0q0: No 'aab' seen.
  - q1q_1q1: One 'a' seen.
  - q2q_2q2: Two 'a's seen.
  - q3q_3q3: "aab" seen (trap state).

Transitions:

- From q0q_0q0, an 'a' takes the automaton to q1q_1q1, a 'b' keeps it in q0q_0q0.
- From q1q_1q1, an 'a' takes it to q2q_2q2, a 'b' takes it back to q0q_0q0.
- From q2q_2q2, an 'a' keeps it in q2q_2q2, a 'b' takes it to the trap state q3q_3q3.

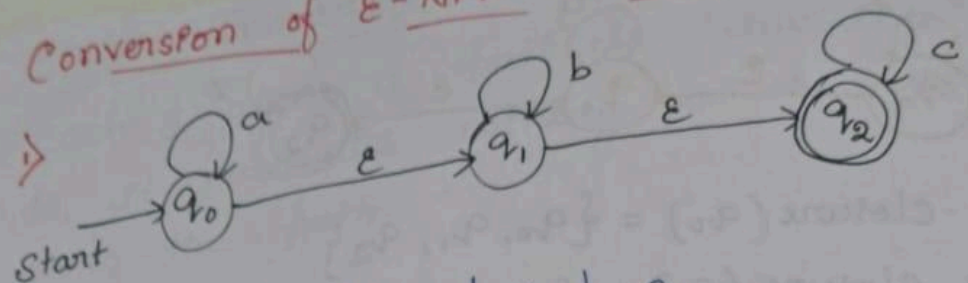
The accepting states are q0q_0q0, q1q_1q1, and q2q_2q2, while q3q_3q3 is the rejecting state.



Q2a. Design the equivalent DFA to the following ε-NFA.

Conversion of ε-NFA to DFA

1)



	ε	a	b	c
q_0	$\{q_0, q_1\}$	$\{q_0\}$	ϕ	ϕ
→ q_0	$\{q_0, q_1\}$	$\{q_0\}$	$\{q_1\}$	ϕ
q_1	$\{q_2\}$	ϕ	$\{q_1\}$	$\{q_2\}$
* q_2	ϕ	ϕ	ϕ	$\{q_2\}$

ε-closure$(q_0) = \{q_0, q_1, q_2\}$

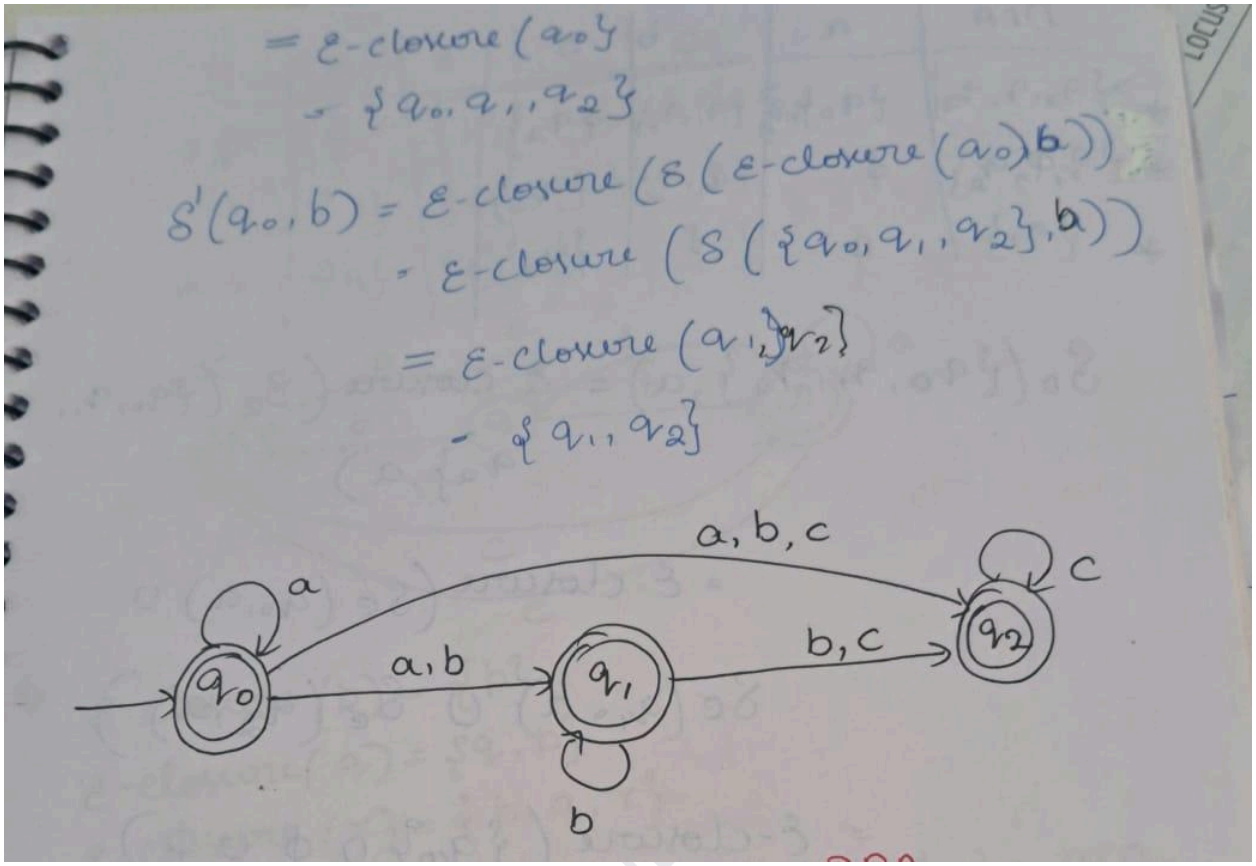
ε-closure$(q_1\}) = \{q_1, q_2\}$

ε-closure$(q_2) = \{q_2\}$

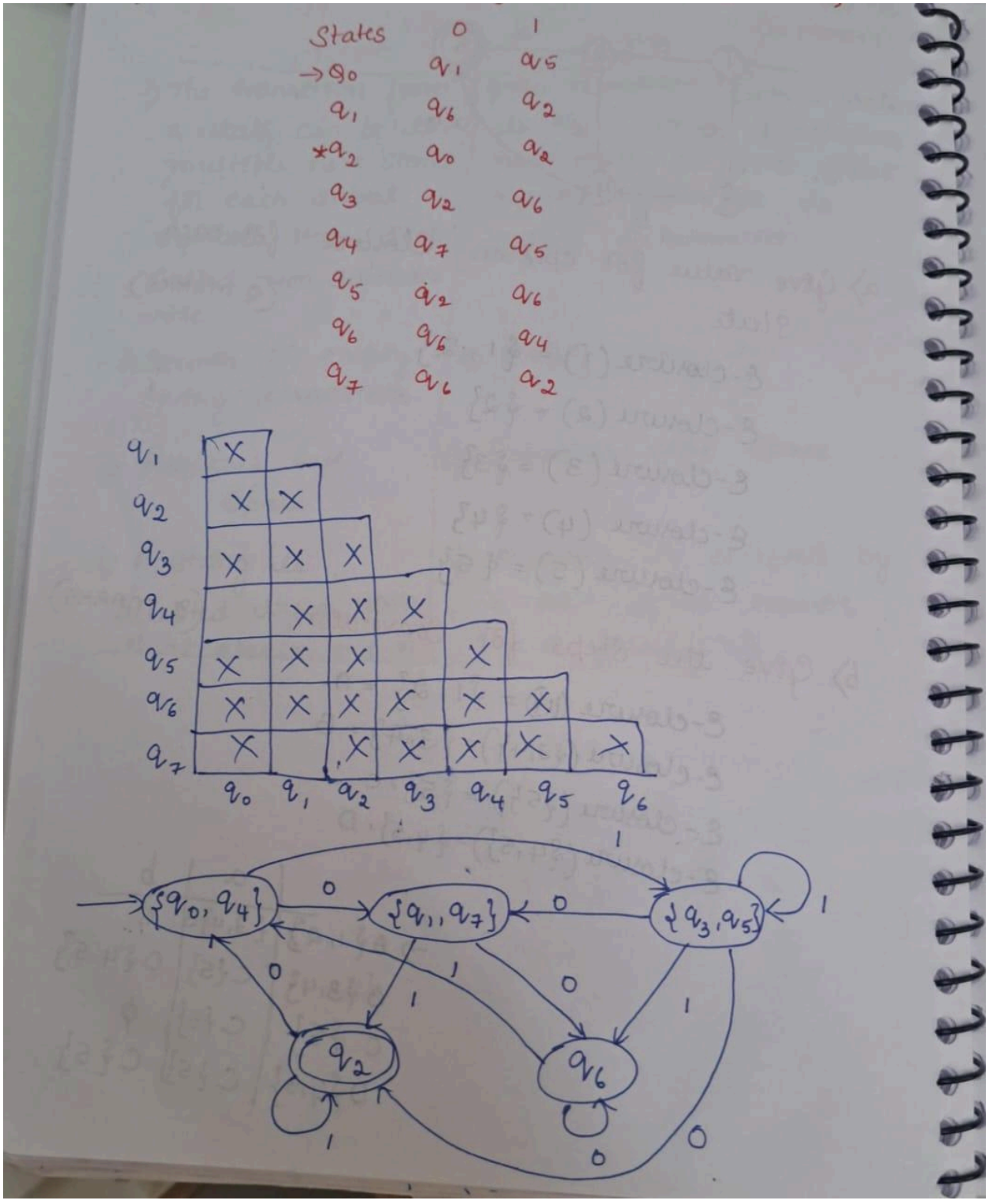
NFA(δ)	a	b	c
*→ q_0	$\{q_0, q_1, q_2\}$	$\{q_1, q_2\}$	$\{q_2\}$
* q_1	ϕ	$\{q_1, q_2\}$	q_2
* q_2	ϕ	ϕ	q_2

$\delta'(q_0, a) = \varepsilon$-closure $(\delta(\varepsilon$-closure $(q_0), a))$

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



Q2b. Minimize the following DFA by identifying distinguishable and non-distinguishable states.



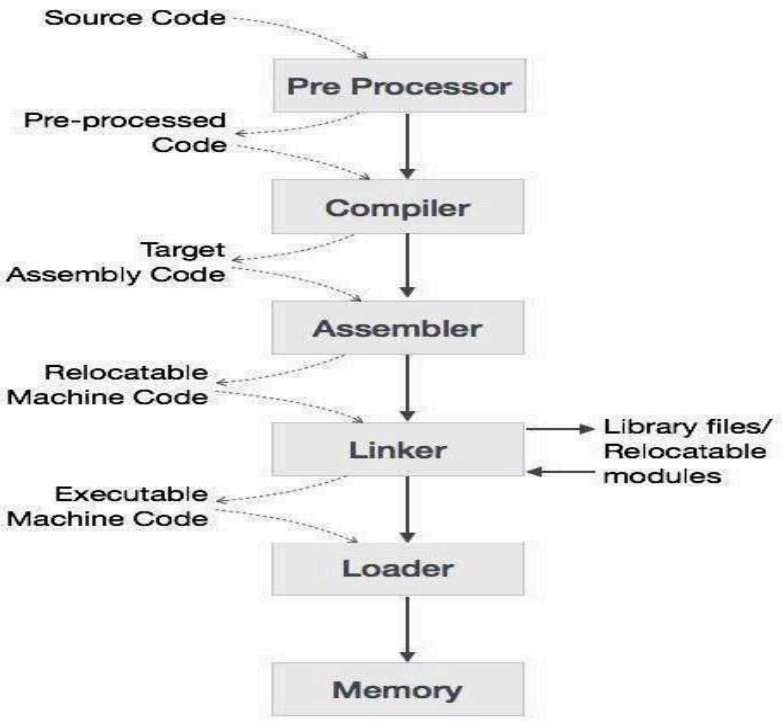
Q2c. With neat diagram explain the components of the language processing system in detail.

(05 Marks)

Components of a Language Processing System:

1. **Lexical Analyzer (Scanner)**: Converts the input program into tokens (identifiers, keywords, symbols) by scanning the source code.
2. **Syntax Analyzer (Parser)**: Analyzes the token sequence to ensure that it adheres to the grammar rules of the language. The output is usually a parse tree.
3. **Semantic Analyzer**: Checks the program for semantic errors, such as type checking and scope resolution, ensuring that the program has meaningful operations.
4. **Intermediate Code Generator**: Generates an intermediate representation of the source code, which is typically easier to optimize than the raw source code.
5. **Code Optimizer**: Improves the intermediate code by eliminating inefficiencies (e.g., removing redundant calculations).
6. **Code Generator**: Produces the target code (assembly or machine code) from the optimized intermediate code.
7. **Symbol Table**: Stores information about variables, functions, objects, etc., used throughout the compilation process.
8. **Error Handler**: Detects and handles errors at various stages (lexical, syntax, semantic, etc.).

9.



Module 2

Q3a. Define Regular Expressions. Write regular expressions for the following:

i) $L = \{a^n b^m | n + m \text{ is even}\}$

- **Regular Expression**:

$$(aa|bb|ab|ba)^*$$

This regular expression ensures that every combination of 'a' and 'b' has an even sum of occurrences.

---

ii) The set of all strings whose 3rd symbol from the right end is 0.

- **Regular Expression**:

$$(0|1)^*0(0|1)(0|1)$$

This regular expression places a '0' as the third character from the right, with any combination of 0s and 1s before and after it.

iii) $L = \{a^n b^{2m} | n \geq 0, m \geq 0\}$

- **Regular Expression**:

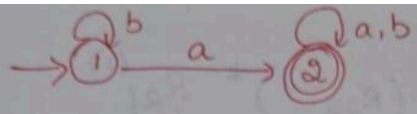
$$a^* b^{2*}$$

This regular expression indicates any number of 'a's followed by an even number of 'b's.

Q3b. Convert the following automata to a regular expression.

For the given automaton with states q1q_1q1, q2q_2q2, and q3q_3q3, and transitions between them labeled with 'a' and 'b', the regular expression for the entire automaton can be derived using state elimination or direct translation.

Q)



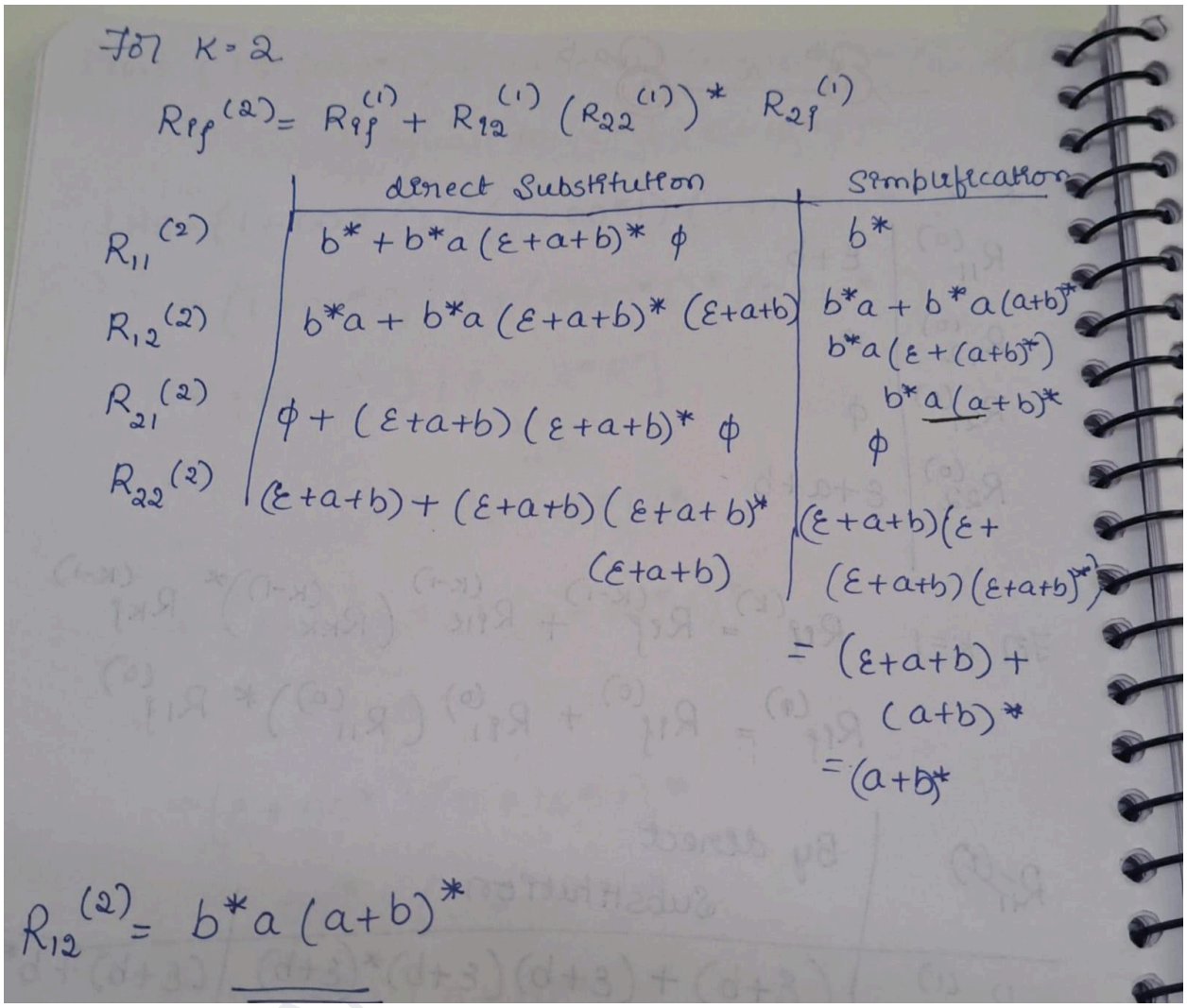
$k=0$

$R_{11}^{(0)}$	$\varepsilon+b$
$R_{12}^{(0)}$	a
$R_{21}^{(0)}$	ϕ
$R_{22}^{(0)}$	$\varepsilon+a+b$

For $k=1$ $\quad R_{ij}^{(k)} = R_{ij}^{(k-1)} + R_{ik}^{(k-1)} \left(R_{kk}^{(k-1)}\right)^* R_{kj}^{(k-1)}$

$$R_{ij}^{(\phi)} = R_{ij}^{(0)} + R_{i1}^{(0)} \left(R_{11}^{(0)}\right)^* R_{1j}^{(0)}$$

~~$R_{11}^{(1)}$~~	By direct substitution	
$R_{11}^{(1)}$	$(\varepsilon+b)+(\varepsilon+b)(\varepsilon+b)^*(\varepsilon+b)$	$(\varepsilon+b)+b^*=b^*$
$R_{12}^{(1)}$	$a+(\varepsilon+b)(\varepsilon+b)^*a$	$a+b^*a=b^*a$
$R_{21}^{(1)}$	$\phi+\phi(\varepsilon+b)^*(\varepsilon+b)$	ϕ
$R_{22}^{(1)}$	$(\varepsilon+a+b)+\phi(\varepsilon+b)^*(a)$	$(\varepsilon+a+b)$



Q3c. Explain the concept of input buffering in the Lexical Analysis along with sentinels.

Input Buffering in Lexical Analysis:

- **Lexical analyzers** often need to handle large input files efficiently. Instead of reading the input one character at a time, input buffering is used to speed up the process.

- **Double Buffering**: The input is divided into two buffers. When one buffer is exhausted, the second buffer is loaded, and the first is refilled in the background.

Sentinels:

- Sentinels are special characters placed at the end of the buffer to signal the end of the input. The sentinel allows the lexical analyzer to avoid continuously checking if it has reached the end of the buffer, improving performance.

Q4a. State and prove Pumping Lemma for regular languages and also prove the language L={anbn∣n≥0}L = \{ a^n b^n | n \geq 0 \}L={anbn∣n≥0} is not regular.

(10 Marks)

Pumping Lemma for Regular Languages: The pumping lemma is a property of all regular languages. It states that for any regular language LLL, there exists a number ppp (pumping length) such that any string sss in the language LLL with length ∣s∣≥p|s| \geq p∣s∣≥p can be divided into three parts, s=xyzs = xyzs=xyz, where:

1. ∣xy∣≤p|xy| \leq p∣xy∣≤p,
2. ∣y∣>0|y| > 0∣y∣>0,
3. xykz∈Lxy^kz \in Lxykz∈L for all k≥0k \geq 0k≥0 (i.e., repeating or removing yyy should still result in a valid string in LLL).

To prove L={anbn∣n≥0}L = \{ a^n b^n | n \geq 0 \}L={anbn∣n≥0} is not regular, we assume the language is regular and apply the pumping lemma. For a string s=apbps = a^p b^ps=apbp, after dividing s=xyzs = xyzs=xyz, and pumping yyy, we will either add or remove a's or b's, breaking the form anbna^n b^nanbn, thus contradicting the lemma. Hence, LLL is not regular.

Q4b. Construct ε-NFA for the following regular expression: (0+11)0*1(04 Marks)

An ε-NFA (epsilon-NFA) can be constructed for the regular expression (0+1)0*1(0 + 1) 0^* 1(0+1)0*1 by following these steps:

1. Start by creating transitions for the expression 0+10 + 10+1, which will have two branches.
2. From the end of each branch, connect an ε-transition to a state that handles 0*0^*0* (zero or more occurrences of 0).
3. Finally, add a transition for the trailing 1.

Q4c. Define Token, Lexeme, and Pattern with an example.

(06 Marks)

- **Token**: A token is a category of lexemes. It represents a syntactic unit in the source code such as keywords, operators, or identifiers. For example, in int x = 10;, int is a token.
- **Lexeme**: A lexeme is the actual string of characters in the source program that matches a pattern and forms a token. For example, x in int x = 10; is a lexeme for the identifier token.
- **Pattern**: A pattern is a rule or a set of rules that defines how the lexemes for a particular token are structured. For example, a pattern for an identifier could be [a-zA-Z_][a-zA-Z_0-9]*, which defines how variable names are structured.

Q5a. Define CFG. Write a CFG for the following languages:

i) **All strings over {a, b} that are even and odd palindromes.**
A **palindrome** is a string that reads the same forward and backward. For even and odd palindromes over the alphabet {a, b}, the CFG can be written as:

i) **All strings over {a, b} that are even and odd palindromes.**

A **palindrome** is a string that reads the same forward and backward. For even and odd palindromes over the alphabet {a, b}, the CFG can be written as:

- **Even palindromes**:
  $S \rightarrow aSa \mid bSb \mid \epsilon$
  This generates even-length palindromes such as "abba", "aabaa".
- **Odd palindromes**:
  $S \rightarrow aSa \mid bSb \mid a \mid b$
  This generates odd-length palindromes such as "aba", "bab".

---

ii) $L = \{a^n | n \geq 0\}$

This is the language that accepts any number of 'a's. The CFG for this language is:

$$S \rightarrow aS \mid \epsilon$$

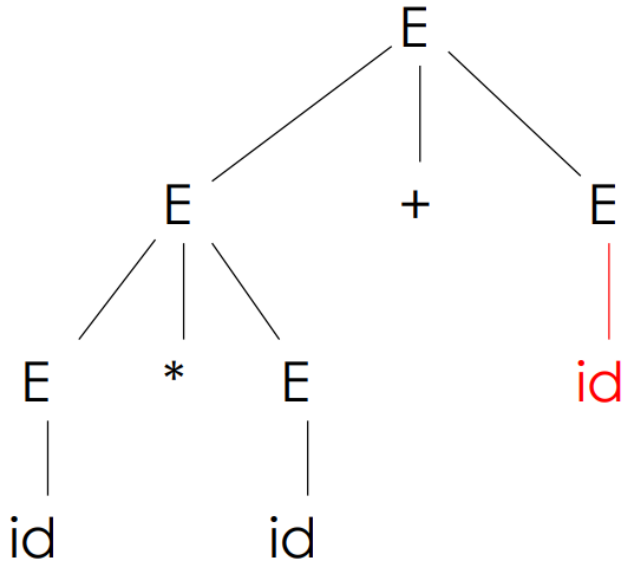
This generates strings like "", "a", "aa", etc.

Q5b. Define ambiguity. Consider the grammar E→E+E|E*E|(E)|idE \rightarrow E + E | E * E | (E) | idE→E+E|E*E|(E)|id. Construct the leftmost and rightmost derivation, parse tree for the string "id + id * id". Also, show that the grammar is ambiguous.

- **Ambiguity**: A grammar is **ambiguous** if there exists a string that can be generated by the grammar in more than one way (i.e., it has more than one parse tree or derivation).

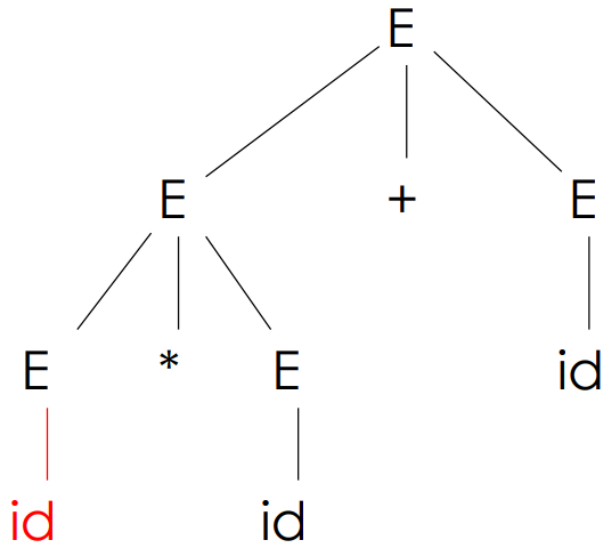
- **Leftmost Derivation for "id + id * id":**
  1. $E \rightarrow E + E$
  2. $E \rightarrow id$
  3. $E \rightarrow E * E$
  4. $E \rightarrow id$
  5. $E \rightarrow id$
- **Rightmost Derivation for "id + id * id":**
  1. $E \rightarrow E * E$
  2. $E \rightarrow id$
  3. $E \rightarrow id$
  4. $E \rightarrow E + E$
  5. $E \rightarrow id$
- The grammar is ambiguous because the s ↓ g "id + id * id" can be derived in more than o way, leading to different parse trees

$$
\begin{aligned}
& E \\
\rightarrow\ & E+E \\
\rightarrow\ & E*E+E \\
\rightarrow\ & id*E+E \\
\rightarrow\ & id*id+E \\
\rightarrow\ & id*id+id
\end{aligned}
$$



Right-most Derivation in Detail (6)

$$
\begin{aligned}
& E \\
\rightarrow\ & E+E \\
\rightarrow\ & E+id \\
\rightarrow\ & E*E+id \\
\rightarrow\ & E*id+id \\
\rightarrow\ & id*id+id
\end{aligned}
$$



Q6a. Consider the CFG given below with the production set, compute the following for the same:

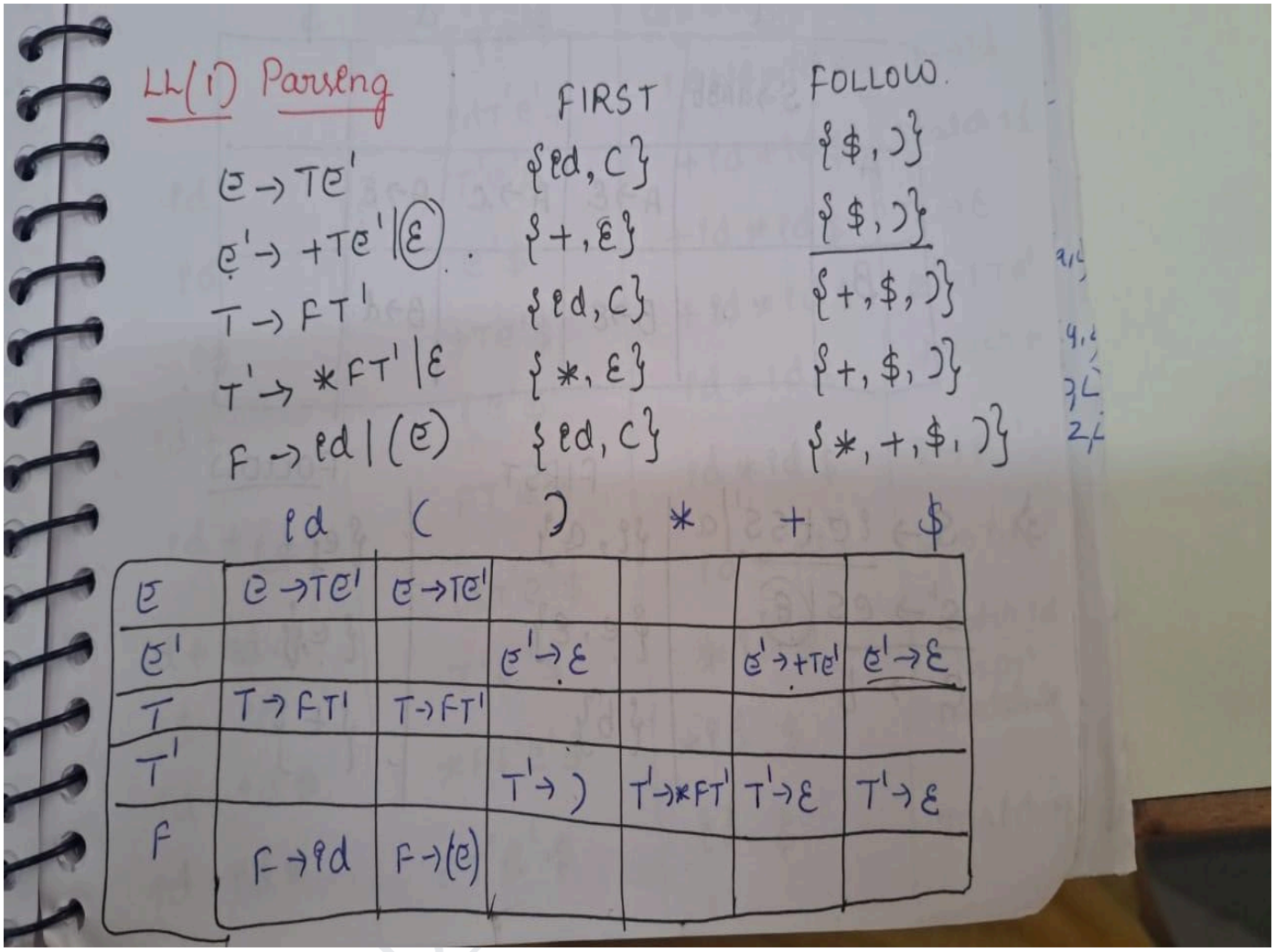
- $E \rightarrow TE'$
- $E' \rightarrow +TE'|\epsilon$
- $T \rightarrow FT'$
- $T' \rightarrow *FT'|\epsilon$
- $F \rightarrow (E)|id$

i) **First() and Follow() set**

- **First()**: For each non-terminal, compute the set of terminals that appear as the first symbol in some derivation.
- **Follow()**: For each non-terminal, compute the set of terminals that can appear immediately to its right in some derivation.

ii) **Predictive Parsing Table**

Construct the predictive parsing table by using the First and Follow sets and the production rules.



Q6b. Write an algorithm to eliminate left recursion from a grammar. Also, eliminate left recursion from the grammar:

- $S \rightarrow Aa|b$
- $A \rightarrow Ac|Sd|\epsilon$

Algorithm to Eliminate Left Recursion:

1. Separate the productions for the non-terminal A into recursive and non-recursive ones.
2. Introduce a new non-terminal A'.
3. Rewrite the recursive and non-recursive productions to eliminate the left recursion.

For the given grammar, after applying the algorithm, the left-recursion-free grammar would be:

$$S \rightarrow bA'$$

$$A \rightarrow dA' \mid cA'$$

$$A' \rightarrow \epsilon$$

Q7a. Define PDA. Design PDA for the language L={WCWR∣W∈{a,b}}L = \{ WCW^R | W \in \{a, b\} \}L={WCWR∣W∈{a,b}} and also show the Instantaneous Description (ID) for the input aabCbaa.

(10 Marks)

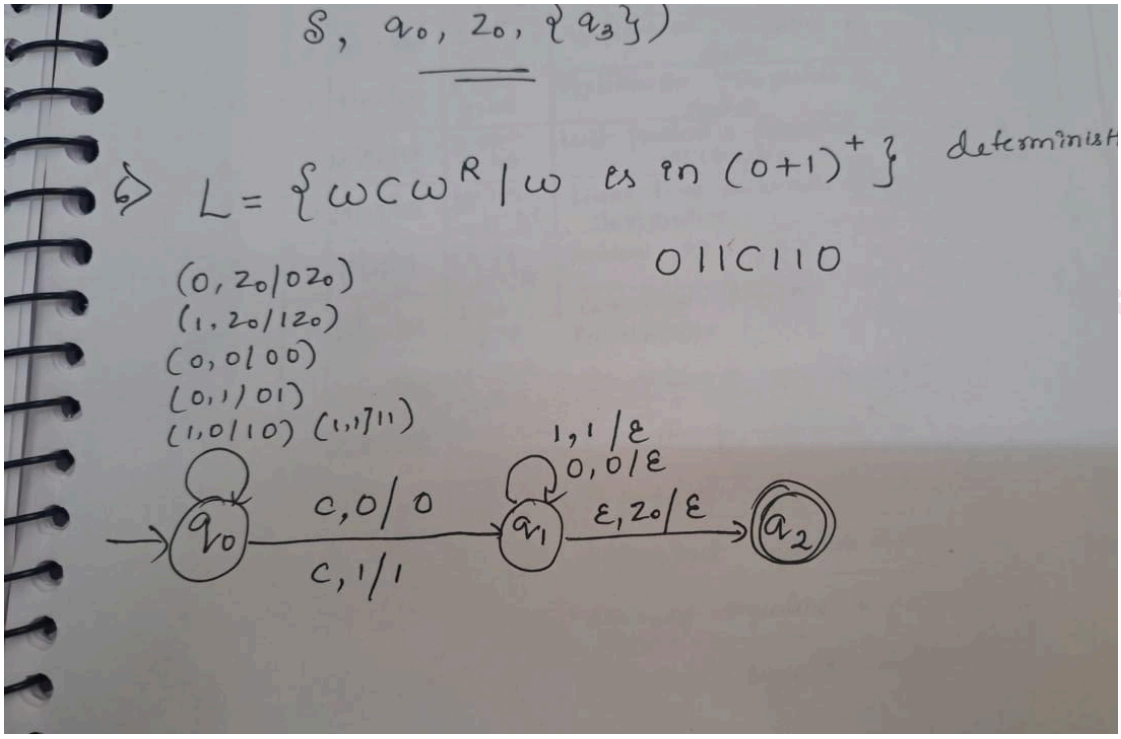
- **PDA (Pushdown Automaton)**: A **PDA** is an automaton that uses a stack to handle context-free languages. It has states, transitions, an input tape, and a stack.

Designing a PDA for L={WCWR}L = \{ WCW^R \}L={WCWR}:

- **W** is any string of a's and b's, and **W^R** is its reverse.
- The PDA will push all symbols before **C** onto the stack and then, after **C**, it will pop symbols from the stack and match them with the reverse of the input string.

Instantaneous Description (ID) for input "aabCbaa":

- Step through the input, pushing "aab" onto the stack before encountering **C**.
- After **C**, start popping symbols from the stack and matching them with the remaining input "baa". The stack will empty, confirming the s



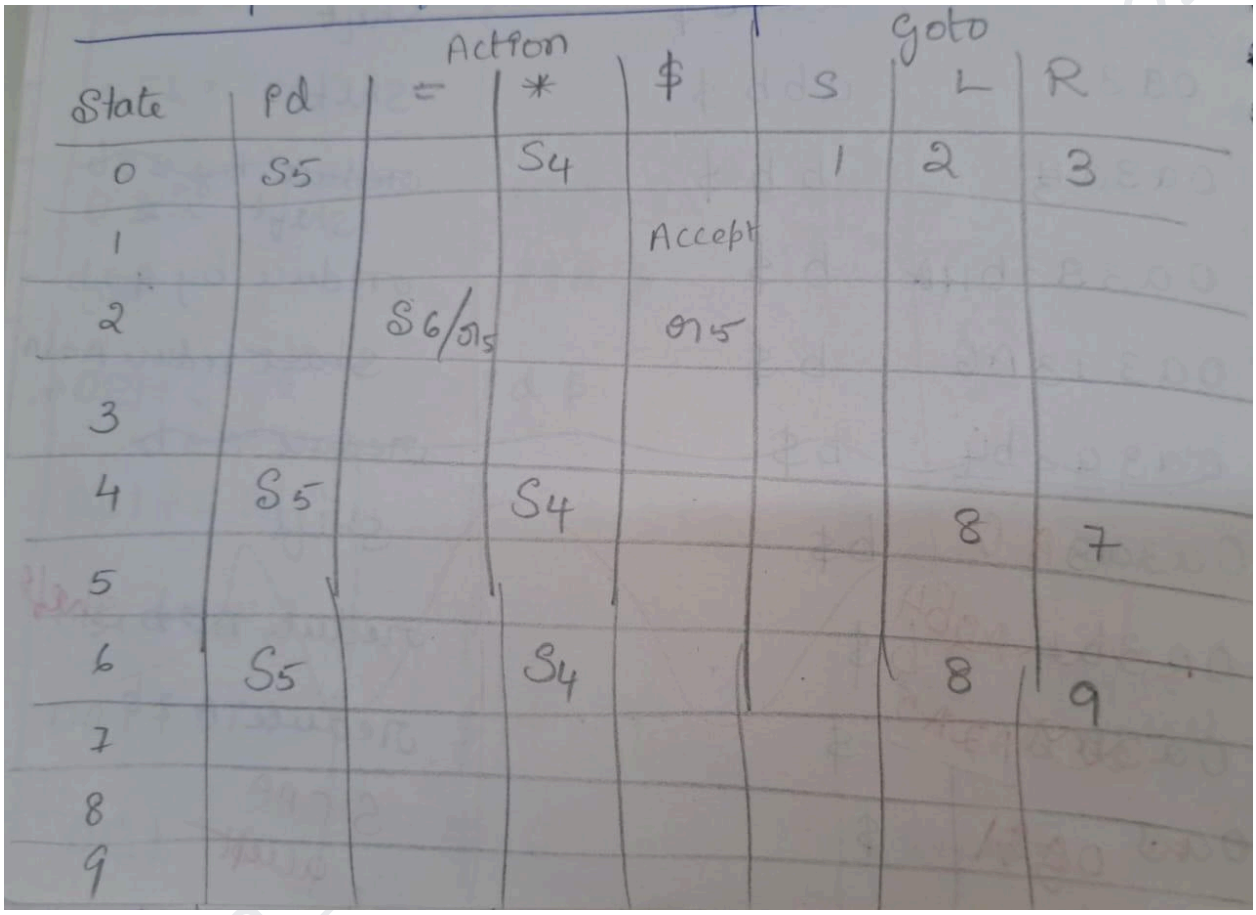
tring belongs to the language.

Q7b. Construct LR(0) automata for the grammar given below:

$$S \rightarrow L = R \,|R$$

$$L \rightarrow *R \,|id$$

$$R \rightarrow L$$



(10 Marks)

LR(0) Automaton:

- **LR(0)** parsing uses a finite automaton to recognize valid prefixes of a rightmost derivation. You need to construct the states and transitions based on the production rules and item sets.
- This involves computing the **closure** of each item and the **goto** function for shifts and reductions.

Q8a. Define shift-reduce parser and handle. Also list and explain the different actions available in Bottom-up parsers.

(10 Marks)

- **Shift-Reduce Parser**: A **shift-reduce parser** is a type of bottom-up parser that works by shifting symbols onto a stack and then reducing them into non-terminals according to the grammar rules.
- **Handle**: A **handle** is the substring that matches the right-hand side of a production and can be replaced by the corresponding left-hand side during the reduce step.
- **Actions in Bottom-up Parsing**:
    - **Shift**: Move the next input symbol onto the stack.
    - **Reduce**: Replace a string of symbols on the stack with a non-terminal based on a grammar rule.
    - **Accept**: Successfully parse the input string.
    - **Error**: An error occurs when no valid shift or reduce action can be applied.

Q8b. Construct the LR(1) automata for the given grammar:

$$S \rightarrow AA$$

$$A \rightarrow aA \mid b$$

I_0 : goto()
$S' \to .S$
$S \to .AA$
$A \to .aA | b$

Goto (I_0, S)
I_1 $S' \to S.$

Goto (I_0, A)
I_2 : $S \to A.A$
$A \to .aA | .b$

Goto (I_0, a)
I_3 : $A \to a.A$
$A \to .aA | .b$

Goto (I_0, b)
I_4 : $A \to b.$

FOLLOW(S) = {$}
FOLLOW(A) = {a, b, $}

Goto (I_1, A)
I_5 : $S \to AA.$

Goto (I_3, A)
I_6 : $A \to aA.$

States	Action			GOTO	
	a	b	$	A	S
0	S_3	S_4		2	1
1			Accept		
2	S_3	S_4		5	
3	S_3	S_4		6	
4	r_3	r_3	r_3		
5	r_1	r_1	r_1		
6	r_2	r_2	r_2		

(10 Marks)

LR(1) Automaton:

- **LR(1)** uses lookahead symbols to make parsing decisions. Construct the automaton by computing item sets with lookaheads and defining the transitions based on the grammar rules and lookahead symbols.

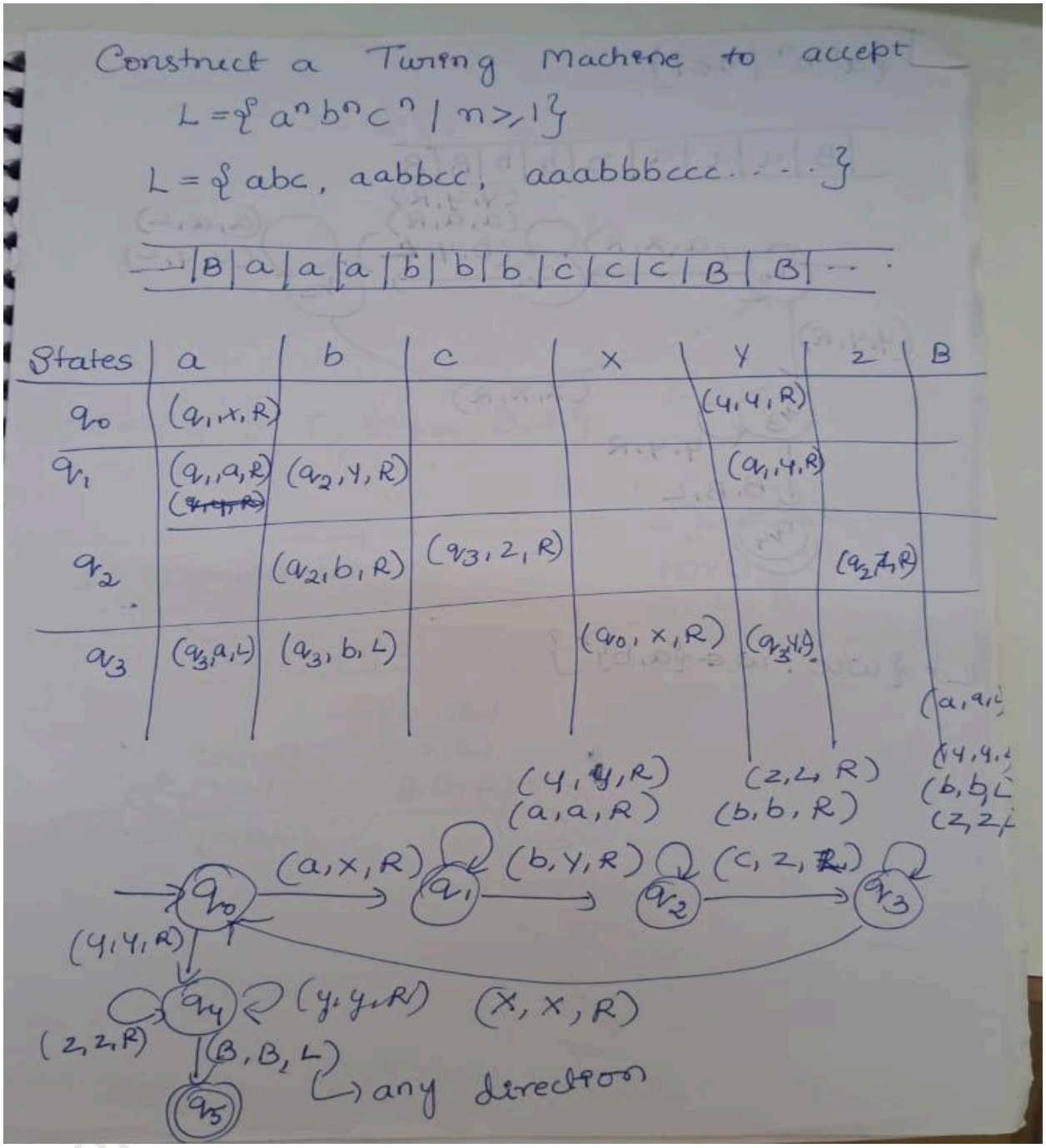
Module 5

Q9a. Design a Turing machine to accept the language $L = \{0^n 1^n 2^n | n \geq 1\}$

(10 Marks)

- A **Turing machine** for this language needs to ensure that for every sequence of 0's, there is a corresponding sequence of 1's and 2's in the same number. The machine would:

  1. Mark off the first 0, the first 1, and the first 2, and repeat the process.
  2. If the input string is successfully processed (i.e., all 0's, 1's, and 2's are accounted for in the same number), the machine accepts the string.
  3. If the number of 0's, 1's, and 2's does not match, the machine rejects the string.



Q9b. Write a short note on the following:

i) **Post correspondence problem**

The **Post correspondence problem (PCP)** is an undecidable problem in computer science. It involves determining whether a given set of pairs of strings (over some

alphabet) has a sequence such that, when concatenated, the two resulting strings are identical. The problem is known to be undecidable, meaning no algorithm can be constructed that always provides a correct yes-or-no answer for every instance of the problem.

ii) **Design issues in code generation**

In **code generation**, several issues need to be addressed to produce efficient machine code:

- **Target Machine Dependence**: The generated code must be suitable for the architecture of the target machine.
- **Efficiency**: The generated code should make optimal use of resources, such as registers and memory.
- **Instruction Selection**: The choice of instructions should minimize execution time.
- **Register Allocation**: Efficient use of registers is critical to reducing memory access times.
- **Optimization**: Optimizations, like dead code elimination and loop unrolling, improve the performance of the generated code.

Q10a. Translate the arithmetic expression $a = b * -c + b * -c$ into:

i) Three address code

- **Three address code** is an intermediate representation used in compilers where each statement contains at most three addresses.

$$t1 = -c$$

$$t2 = b * t1$$

$$t3 = -c$$

$$t4 = b * t3$$

$$a = t2 + t4$$

eg:- $a = b * -c + b * -c$

$t_1 = \text{uminus } c$

$t_2 = b * t_1$

~~$t_3 = \text{uminus } c = t_1$~~ $t_3 = t_2 + t_2$

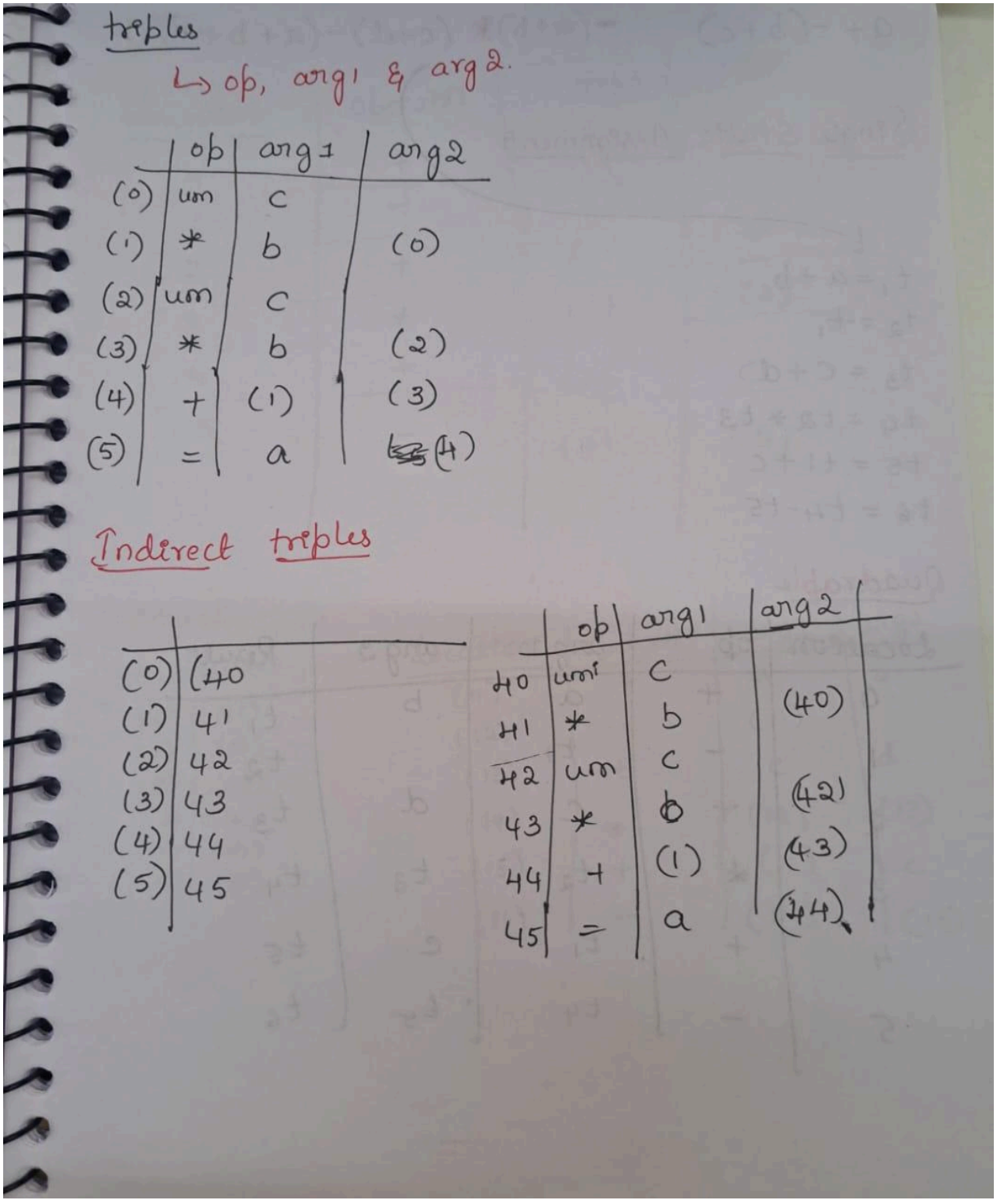
~~$t_4 = b * t_1$~~ $a = t_3$

~~$t_5 = t_2 + t_4$~~

~~$a = t_5$~~

Quadruples

	op	arg 1	arg 2	result
0	uminus	~~uminus~~ c		t_1
1	(*)	b	t_1	t_2
2	uminus	c		t_3
3	*	b	t_3	t_4
4	+	t_2	t_4	t_5
5	=	t_5		a



Q10b. Write a short note on:

i) **Decidable language**

A **decidable language** is a formal language for which there exists a Turing machine that can decide whether any given string belongs to the language or not. In other words, the Turing machine halts with a definitive yes-or-no answer for every input. Examples of decidable languages include regular languages and context-free languages.

ii) **Halting problem in Turing machines**

The **halting problem** is a decision problem that asks whether a given Turing machine will halt on a given input or continue running forever. Alan Turing proved that this problem is undecidable—there is no algorithm that can determine whether any arbitrary Turing machine will halt or not for every possible input.