Experiment : 1	Designing Finite State Machines
Date : 11/12/24	- Deterministic Finite Automata

Aim:

To design and construct Deterministic Finite Automata (DFA) for the following language conditions over alphabet $\Sigma = \{a, b\}$. This experiment includes constructing the transition diagrams using JFLAP and verify correctness by testing strings for acceptance or rejection.

Procedure:

Input: It refers to the symbols from the alphabet (Σ) for DFA

States: It refers to the positions of DFA during execution.

Initial state: The state where the DFA begins processing

Transition state: The states through which the DFA moves while processing the string.

Final / Acceptance State: State where DFA accepts string as valid.

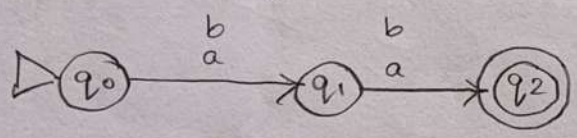
Dead State: A state where the DFA goes if it encounters an invalid input (or) violates the language constrains.

i) $|w| = 2$ (String length exactly equals to 2)

Step 1: Identify the language

$L = \{ab, ba, aa, bb\}$

Step 2: State Diagram



Step 3: Transition States

$\delta = \delta(q_0, a) = q_1$ $\quad$ $\delta(q_1, a) = q_2$

$\delta(q_0, b) = q_1$ $\quad$ $\delta(q_1, b) = q_2$

Step 4: Transition Table

$Q \mid \Sigma$	a	b
q_0	q_1	q_1
q_1	q_2	q_2
q_2	ϕ	ϕ

Initial State = $\{q_0\}$

Acceptance state = $\{q_2\}$

Step 5: Accepting Tuples of Machine M

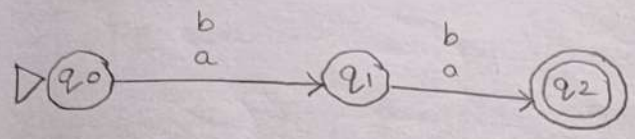
$M = [\{q_0, q_1, q_2\}, \{a, b\}, \delta, \{q_0\}, \{q_2\}]$

ii) $|w| <= 2$ (String length with almost 2).

Step 1: Identify the language

$L = \{a, b, ab, ba, aa, bb\}$

Step 2: State Diagram



Step 3: Transition States

$\delta = \delta(q_0, a) = q_1 \quad \delta(q_1, a) = q_2$

$\delta(q_0, b) = q_1 \quad \delta(q_2, b) = q_2$

Step 4: Transition Table

Q|Σ	a	b
q_0	q_1	q_1
q_1	q_2	q_2
q_2	ϕ	ϕ

Initial State: $\{q_0\}$

Final State: $\{q_0, q_1, q_2\}$

Step 5: Accepting tuples of machine M

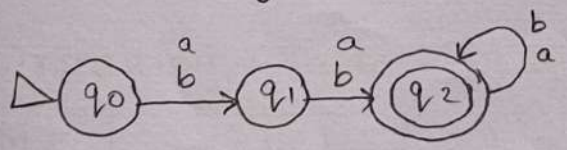
$M = [\{q_0, q_1, q_2\}, \{a, b\}, \delta, \{q_0\}, \{q_0, q_1, q_2\}]$

iii) $\omega > = 2$ (String length with atleast 2)

Step 1: Identify the language

$L = \{aa, ab, bb, ba, aab, aba, abba, baba, \ldots\}$

Step 2: State Diagram



Step 3: Transition States:

$\delta = \delta(q_0, a) = q_1 \quad \delta(q_1, a) = q_2 \quad \delta(q_2, a) = q_2$

$\delta(q_0, b) = q_1 \quad \delta(q_1, b) = q_2 \quad \delta(q_2, b) = q_2$

Step 4 : Transition Table

Q|Σ	a	b
q_0	q_1	q_1
q_1	q_2	q_2
q_2	q_2	q_2

Initial State = $\{q_0\}$

Final State = $\{q_2\}$

Step 5 : Accepting Tuples of Machine M

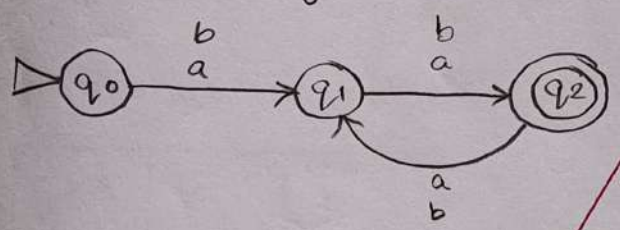
$M = [\{q_0, q_1, q_2\}, \{a, b\}, \delta, \{q_0\}, \{q_2\}]$

iv) $|w| \bmod 2 = 0$ (string length divisible by 2 (or) even string length)

Step 1 : Identify the language

$L = \{ab, aa, ba, abba, baba, baab, abab, \ldots\}$

Step 2 : State Diagram



Step 3 : Transition States

$\delta = \delta(q_0, a) = q_1 \quad \delta(q_1, a) = q_2 \quad \delta(q_2, a) = q_1$

$\delta(q_0, b) = q_1 \quad \delta(q_1, b) = q_2 \quad \delta(q_2, b) = q_1$

Step 4 : Transition Table

Q|Σ	a	b
q_0	q_1	q_1
q_1	q_2	q_2
q_2	q_1	q_1

Initial State: $\{q_0\}$

Final State: $\{q_2\}$

Step 5: Accepting Tuples of Machine M

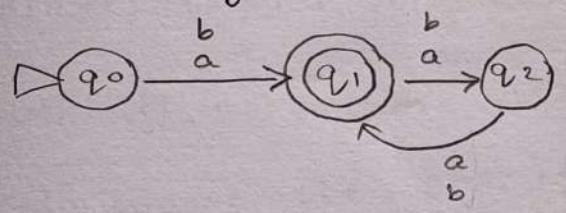
$M = [\{q_0, q_1, q_2\}, \{a, b\}, \delta, \{q_0\}, \{q_2\}]$

v) $|w| \bmod 2 = 1$ (String length not divisible by 2 (or) odd string length)

Step 1: Identify the language

$L = \{a, b, aba, baa, abbab, babab, \ldots\}$

Step 2: State Diagram:



Step 3: Transition States:

$\delta = \delta(q_0, a) = q_1$ $\quad \delta(q_1, a) = q_2$ $\quad \delta(q_2, a) = q_1$

$\delta(q_0, b) = q_1$ $\quad \delta(q_2, a) = q_2$ $\quad \delta(q_2, b) = q_1$

Step 4: Transition Table

$Q \mid \Sigma$	a	b
q_0	q_1	q_1
q_1	q_2	q_2
q_2	q_1	q_1

Initial State : $\{q_0\}$

Final State : $\{q_1\}$

Step 5: Accepting Tuples of Machine M

$M = [\{q_0, q_1, q_2\}, \{a, b\}, \delta, \{q_0\}, \{q_1\}]$

Automata Theory is a study of abstract machines and automata as well as the computational problems that can be solved using them. An automation with a finite numbers of seats is called Finite Automata (FA) (or) Finite State Machine (FSM). It plays a major role in the theory of computation, compiler, construction, parsing

JFLAP is a software for experimenting with formal languages like DFA, NFA, non deterministic pushdown automata, multi-tape turing machines

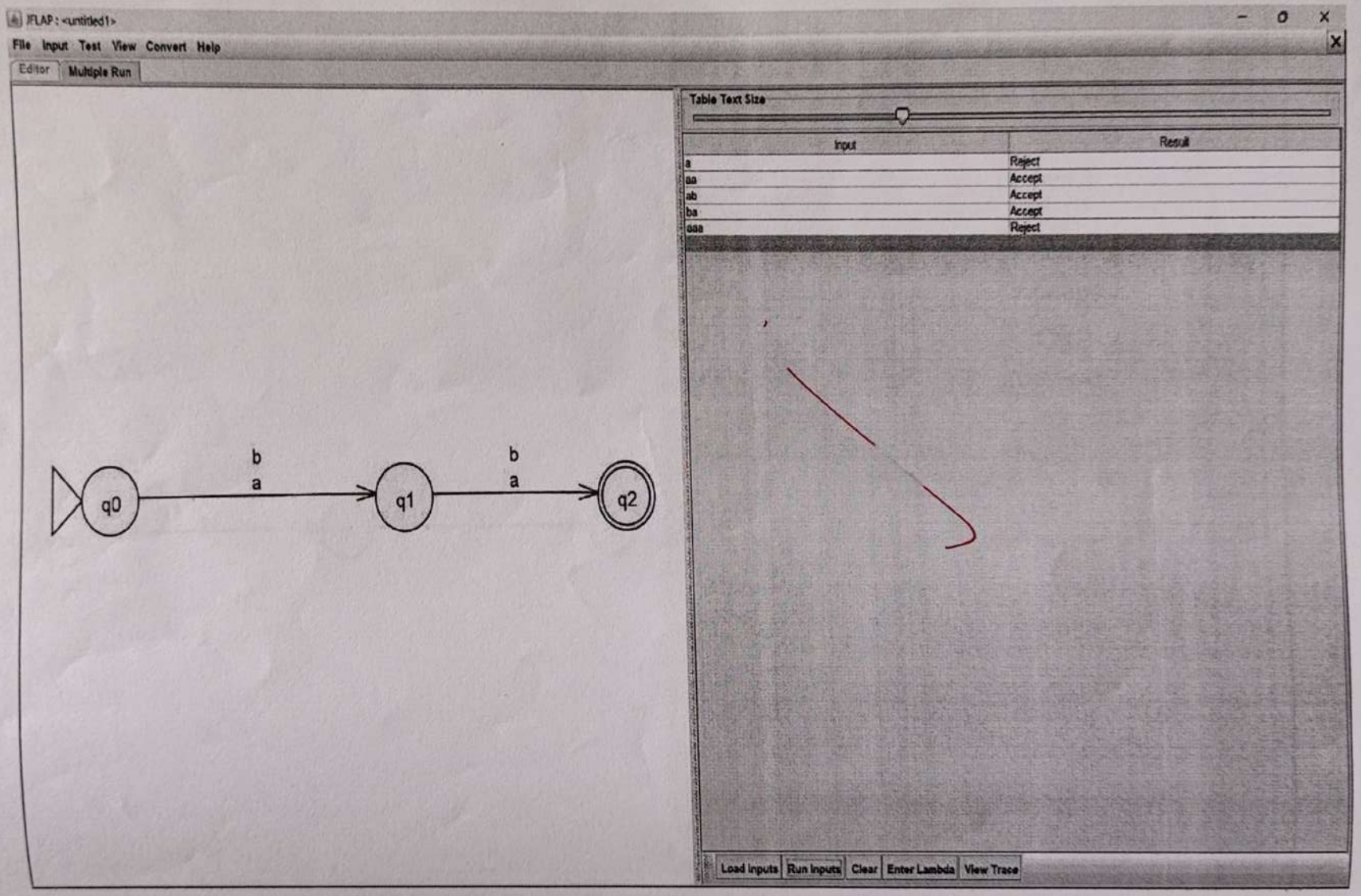
Finite Automata:

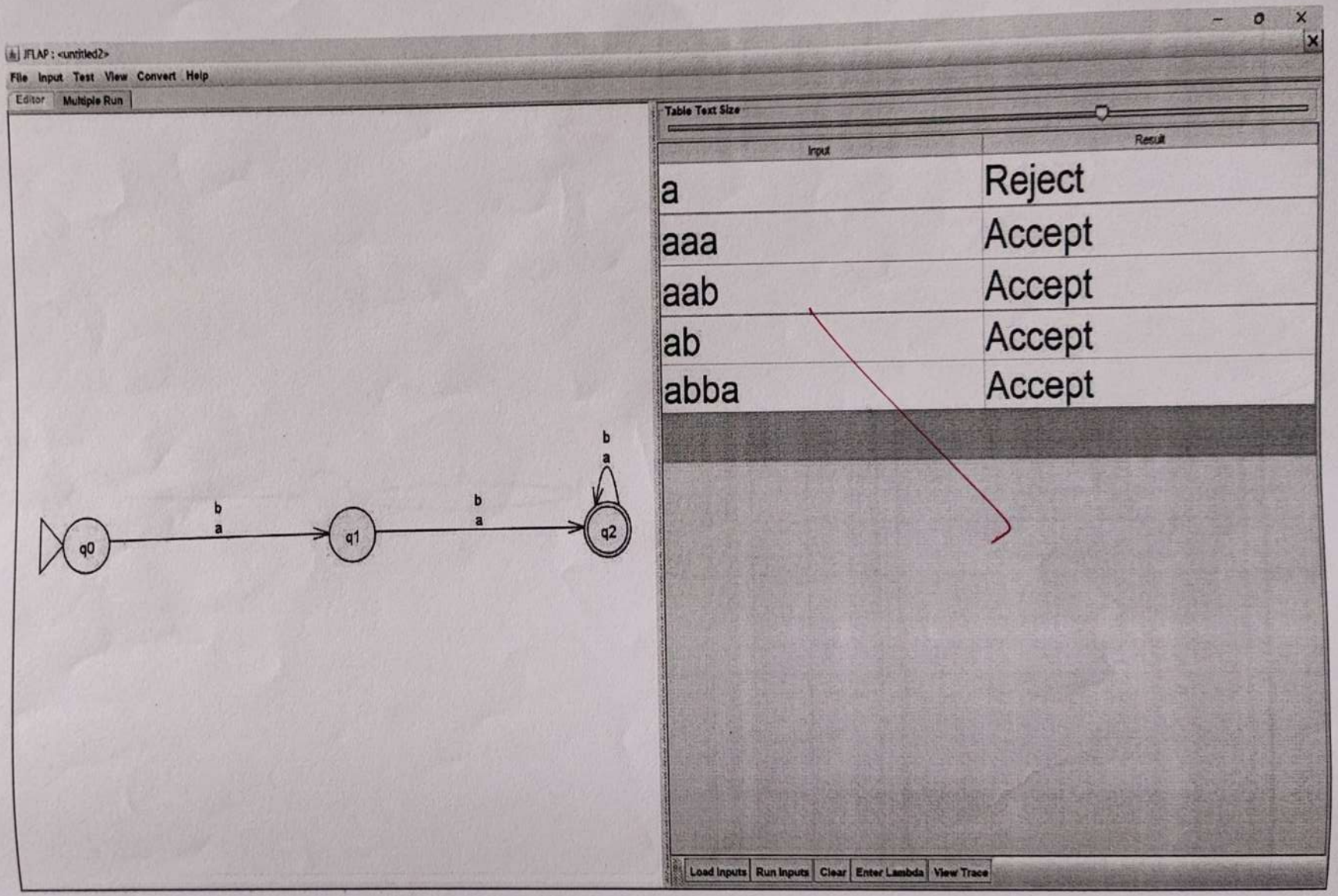
1) A finite Automata is a formal system which can be visualized as a graph / Table.
2) It remember finite amount of data
3) It has only finite number of states.
4) Rules that tell how the state changes are called transitions.

Usage:

1) Design and verification of circuits and communications protocols
2) Text processing applications (Regular expressions)
3) Very important for creating compilers.

Deterministic Finite Automata	Non-Deterministic Finite Automata
Each transition leads to exactly one state	A transition leads to a subset of states i.e 0,1 or more than 1 state
Accepts input if the last state is in Final	Accepts input if one of the last states is in Final
Backtracking is allowed	Backtracking is not always allowed
Requires more space	Requires less space
Empty string transitions are not seen in DFA	Permits empty string transitions
For a given state, on a given input we reach a deterministic and unique state	For a given state, on a given input we reach more than one state
DFA is a subset of NFA	Need to convert NFA to DFA in the design of a compiler.
$\delta : Q \times \Sigma \rightarrow Q$ For example - $\delta(q_0, a) = \{q_1\}$	$\delta : Q \times \Sigma = 2^Q$ For example - $\delta(q_0, a) = \{q_1, q_2\}$
DFA is more difficult to construct	NFA is easier to construct
DFA is understood as one machine	NFA is understood as multiple small machines computing at same time.


JFLAP : <untitled1>
File Input Test View Convert Help
Editor
Multiple Run
Table Text Size
Input
Result
a
Reject
aa
Accept
ab
Accept
ba
Accept
aaa
Reject
b
a
b
a
q0
q1
q2
Load Inputs
Run Inputs
Clear
Enter Lambda
View Trace


JFLAP : <untitled2>
File Input Test View Convert Help
Editor
Multiple Run
Table Text Size
Input
Result
a
Reject
aaa
Accept
aab
Accept
ab
Accept
abba
Accept
q0
q1
q2
b
a
b
a
b
a
Load Inputs
Run Inputs
Clear
Enter Lambda
View Trace

JFLAP : <untitled3>

File Input Test View Convert Help

Editor | Multiple Run

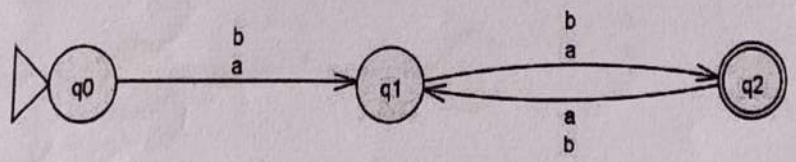
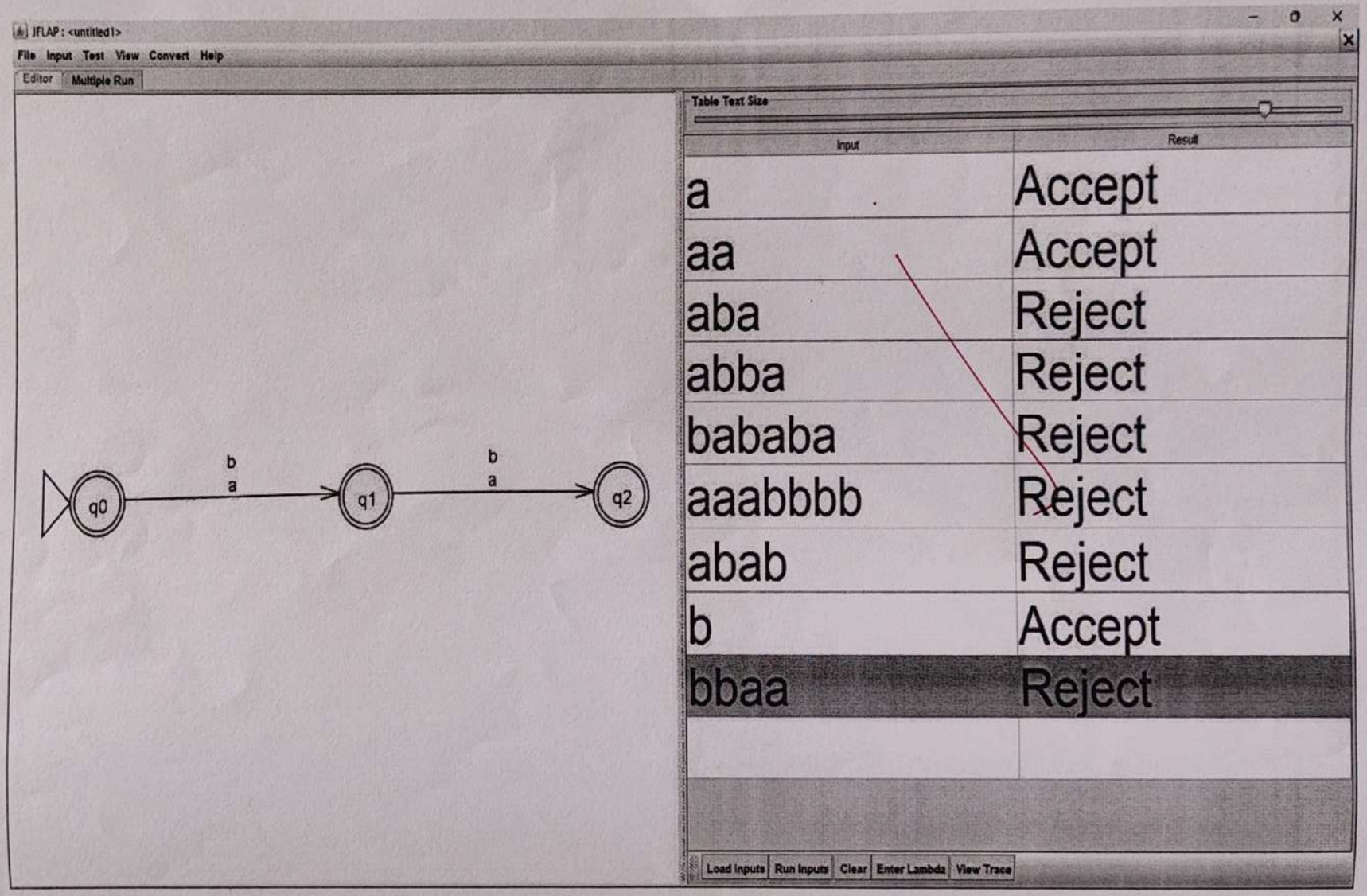


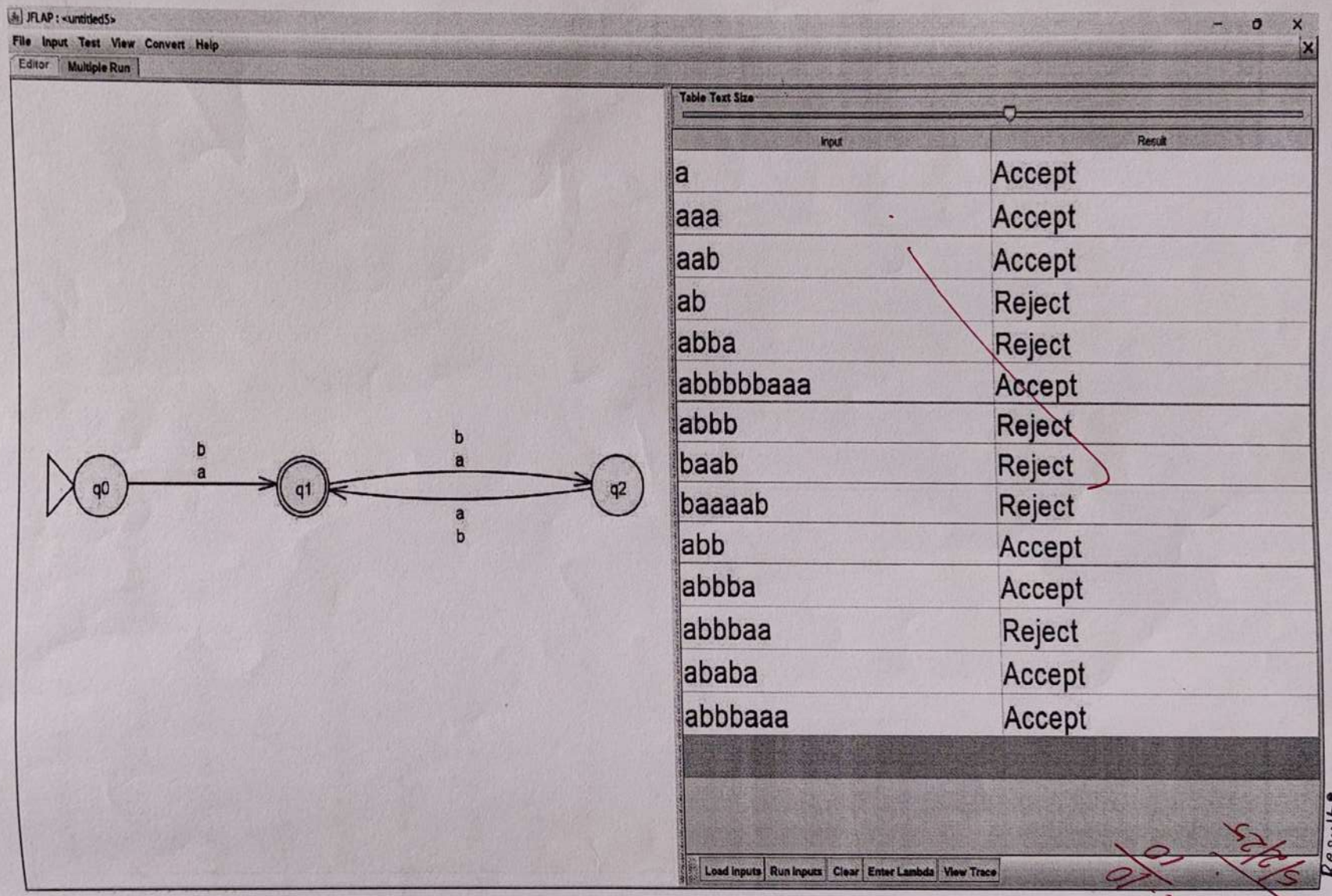
Table Text Size

Input	Result
a	Reject
aaa	Reject
aab	Reject
ab	Accept
abba	Accept
abbbbbaaa	Reject
abbb	Accept
baab	Accept
baaaab	Accept

Load Inputs | Run Inputs | Clear | Enter Lambda | View Trace



Input	Result
a	Accept
aa	Accept
aba	Reject
abba	Reject
bababa	Reject
aaabbbb	Reject
abab	Reject
b	Accept
bbaa	Reject



Input	Result
a	Accept
aaa	Accept
aab	Accept
ab	Reject
abba	Reject
abbbbbaaa	Accept
abbb	Reject
baab	Reject
baaaab	Reject
abb	Accept
abbba	Accept
abbbaa	Reject
ababa	Accept
abbbaaa	Accept

View. 10/10

Result:
The DFA is constructed and language acceptance string is verified successfully in JFLAP

Experiment : 1 (b)	Designing Finite State Machine
Date: 18/12/24	- Non Deterministic Finite Automata

Aim:

To construct a non deterministic finite Automata (NFA) for given language and to verify is the given string is in the language of automata or not using JFLAP

Explanation:

Non- Deterministic Finite Automata:

1) NFA stands for Non-Deterministic Finite Automata. It is early to construct an NFA than DFA for a given regular language.

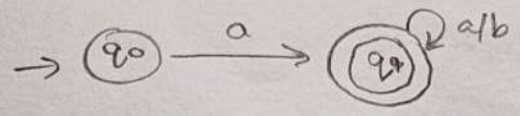
2) The finite automata are called NFA when there exist many paths for specific input from the current state to next state.

3) Every NFA is not DFA, but each NFA can be translated into DFA.

4) NFA is defined in the same way as DFA but with the following two exceptions, it contains multiple next states and transition..

Construct the NFA that accepts the below language over $\Sigma = \{a, b\}$

i) String starts with 'a'.



$L = \{a, ab, aab, ababa, \ldots\}$

Transition Function:

$\delta = \delta(q_0, a) = q_1$

$\delta(q_1, a) = q_1$

$\delta(q_1, b) = q_1$

Transition Table

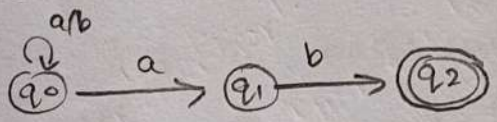
$\delta \mid \Sigma$	a	b
q_0	q_1	ϕ
q_1	q_1	q_1

Tuples:

$Q = \{q_0, q_1\}$ $\Sigma = \{a, b\}$ $\delta = \{\delta(q_0, a) = q_1\}, \delta(q_1, a) = q_1, \delta(q_1, b) = q_1,$ $q_0 = q_0, F = q_1$

$M = [\{q_0, q_1\}, \{a, b\}, \delta, q_0, q_1]$

ii) strings ends with 'ab'



$L = \{ab, aab, abaab, abaabab, \ldots\}$

Transition Function:

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

$\delta(q_0, b) = \{q_0\}$

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

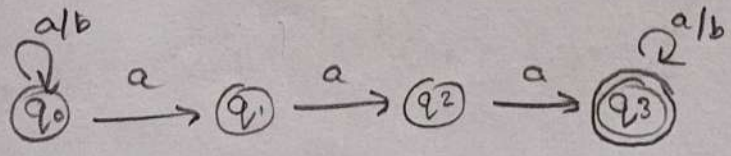
Transition Table:

$\delta \mid \Sigma$	a	b
q_0	q_0, q_1	q_0
q_1	ϕ	q_2
q_2	ϕ	ϕ

Tuples:

$M = [\{q_0, q_1\}, \{a, b\}, \delta, \{q_0\}, \{q_2\}]$

iii) strings that have 3 consecutive 'a's.



$L = \{aaa, baaab, bbaaa, aaaabb, \ldots\}$

Transition Function:

$\delta = \delta(q_0, a) = q_0, q_1$

$\delta(q_0, b) = q_0$

$\delta(q_1, a) = q_2$

$\delta(q_2, a) = q_3$

$\delta(q_3, a) = q_3$

$\delta(q_3, b) = q_3$

Transition Table:

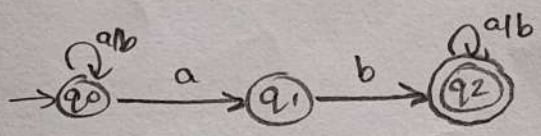
$\delta \mid \Sigma$	a	b
q_0	q_0, q_1	q_0
q_1	q_2	ϕ
q_2	q_3	ϕ
q_3	q_3	q_3

Tuples:

$Q = \{q_0, q_1, q_2, q_3\}$, $\Sigma = \{a, b\}$, δ, $q_0 = q_0$, $F = \{q_3\}$

$M = [\{q_0, q_1, q_2, q_3\}, \{a, b\}, \delta, \{q_0\}, \{q_3\}]$

iv) String that has 'ab' as substring.



$L = \{ab, aab, abaab, abaabab, \dots\}$

Transition Function:

$\delta = \delta(q_0, a) = q_0, q_1$

$\delta(q_0, b) = q_0$

$\delta(q_1, b) = q_2$

$\delta(q_2, a) = q_2$

$\delta(q_2, b) = q_2$

Transition Table:

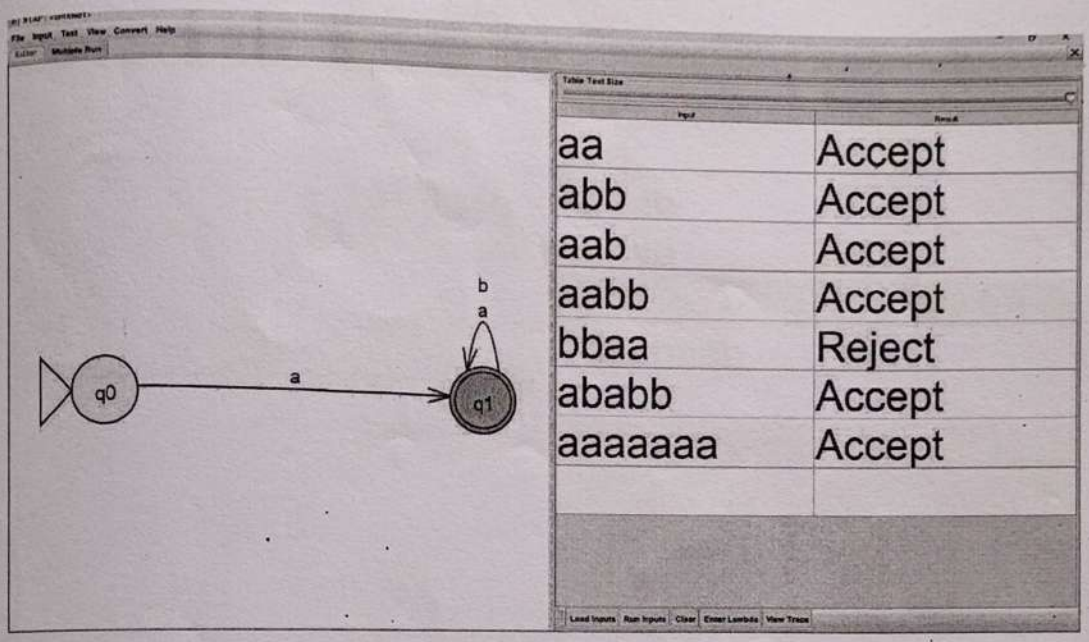
$\delta \mid \Sigma$	a	b
q_0	q_0, q_1	q_0
q_1	ϕ	q_1
q_2	q_2	q_2

Tuples:

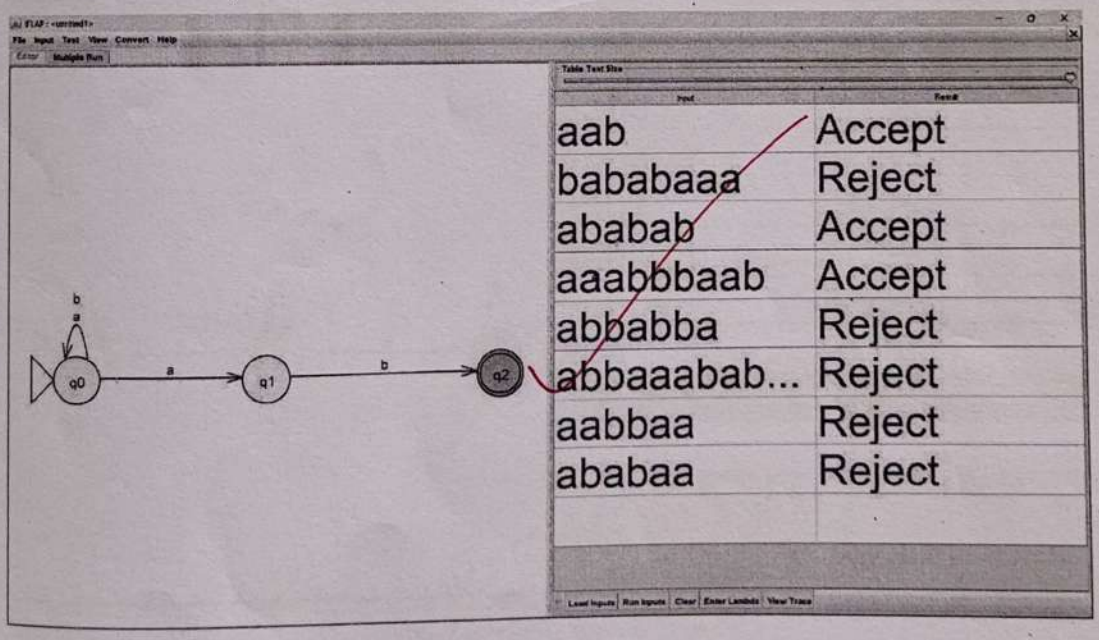
$Q = \{q_0, q_1, q_2\}$ $\Sigma = \{a, b\}$, $q_0 = q_0$, $F = \{q_2\}$

$M = [\{q_0, q_1, q_2\}, \{a, b\}, \delta, \{q_0\}, \{q_2\}]$

1)STING STARTS WITH 'a':



2)STRING ENDS WITH 'ab'



Experiment : 2 Conversion of non Deterministic Finite Automata (NFA) to Deterministic Finite Automata (DFA)

Date: 8/1/25

Aim: To construct the non-deterministic finite Automata (NFA) to deterministic

Steps for converting NFA to DFA:

Step1 : Initially $Q' = \phi$

Step2 : Add q_0 of NFA to Q'. then find the transitions from the start node

Step3 : In Q', find the possible set of states for each output symbol. If this set of states is not in Q', then add it Q'.

Step4: In DFA, the final state will be all the states which contain F (final state of NFA).

Steps for converting NFA with ε to DFA:

Step1: we will have the ε - closure for the starting state of NFA as a starting state of DFA.

Step2: Find the states for each input symbol that can be traversed from the present. That means the union of transition value and their closure for each State of NFA present in the current state of DFA

Step3: If we found a newstate, take it as current state and repeat step3.

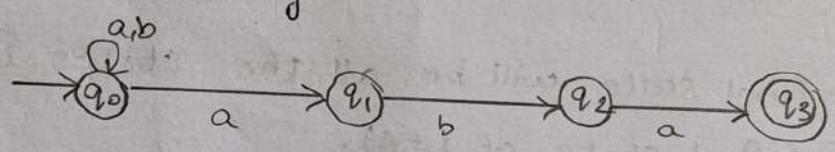
Step4: Repeat step2 and 3 until there is a new state present in the transition table of DFA

Step5 : Mark the state of DFA as a final state which contains the final states of NFA

Algorithm:

1. Create the NFA
2. Open JFLAP, create states and transitions using the tools.
3. Set the initial and final state.
4. Click on convert to convert the NFA to DFA
5. Process states
6. Click done when everything is complete to see the DFA
7. Save and test

Convert the following NFA to DFA



Step1: Transition Table

Q\Σ	a	b
→ q_0	$\{q_0, q_1\}$	q_0
q_1	ϕ	q_2
q_2	q_3	ϕ
* q_3	ϕ	ϕ

Step 2: Identify new states

Let 'q_0' will be state 'A'

$q_0 \rightarrow A$

$\delta(A, a) = \{q_0, q_1\} \rightarrow$ (B) new state

$\delta(A, b) = \{q_0\} \rightarrow$ (A)

$\delta(B,a) = \delta(q_0,a) \cup \delta(q_1,a)$

$= \{q_0, q_1\} \cup \phi$

$= \{q_0, q_1\} \rightarrow B$

$\delta(B,b) = \delta(q_0,b) \cup \delta(q_1,b)$

$= q_0 \cup q_2$

$= \{q_0, q_2\} \rightarrow C$

$\delta(C,a) = \delta(q_0,a) \cup \delta(q_2,a)$

$= \{q_0, q_1\} \cup \{q_3\}$

$= \{q_0, q_1, q_3\} \rightarrow D$

$\delta(C,b) = \delta(q_0,b) \cup \delta(q_2,b)$

$= q_0 \cup \phi$

$= \{q_0\} \rightarrow A$

$\delta(D,a) = \delta(q_0,a) \cup \delta(q_1,a) \cup \delta(q_3,a)$

$= \{q_0, q_1\} \cup \phi \cup \phi$

$= \{q_0, q_1\} \rightarrow B$

$\delta(D,b) = \delta(q_0,b) \cup \delta(q_1,b) \cup \delta(q_3,b)$

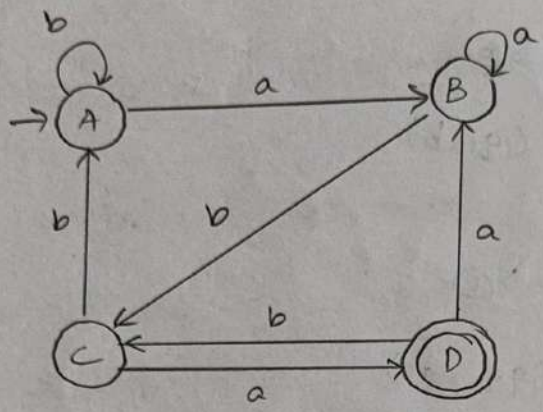
$= q_0 \cup q_2 \cup \phi$

$= \{q_0, q_2\} \rightarrow C$

Step 3: construct new transition table

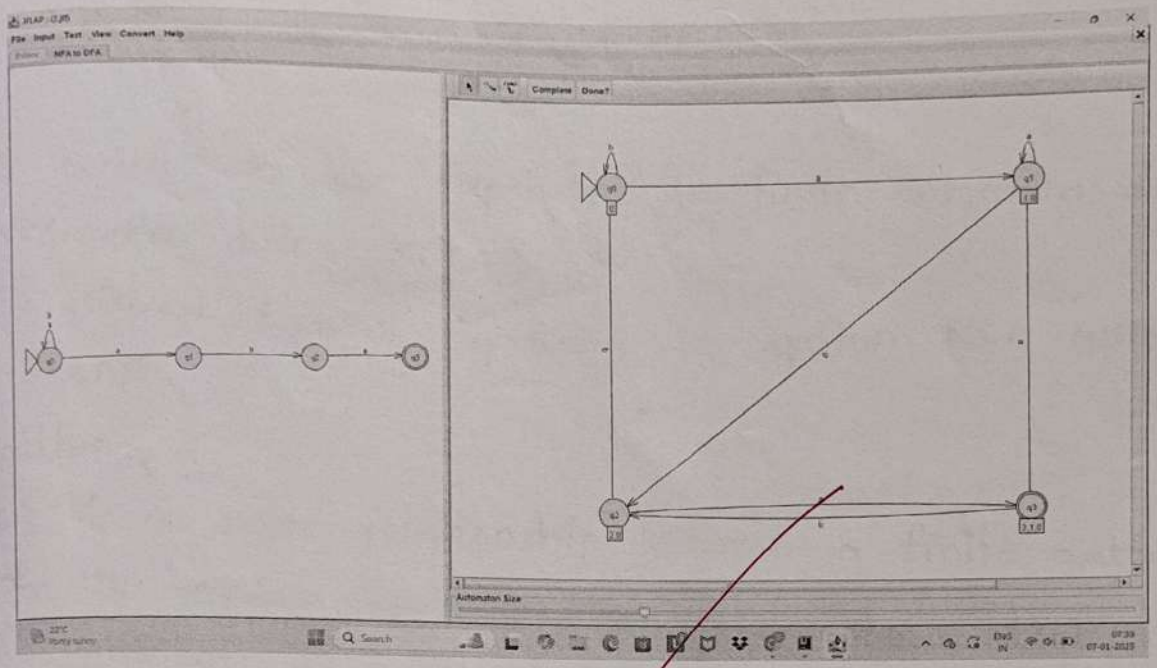
	Q\Σ	a	b
→	A	B	A
	B	B	C
	C	D	A
*	D	B	C

Step 4: construct DFA using new transition table



Step 5: Tuples

M : [{A, B, C, D}, {0, 1}, δ, {A}, {D}]



Result: 5/2/25 Viva: 10/10.

Thus the conversion of non deterministic finite Automata (NFA) to Deterministic finite Automata (DFA). is executed successfully and output is verified.

Experiment: 3

Date: 29/1/25

Automata and Regular Expression Translation using JFLAP

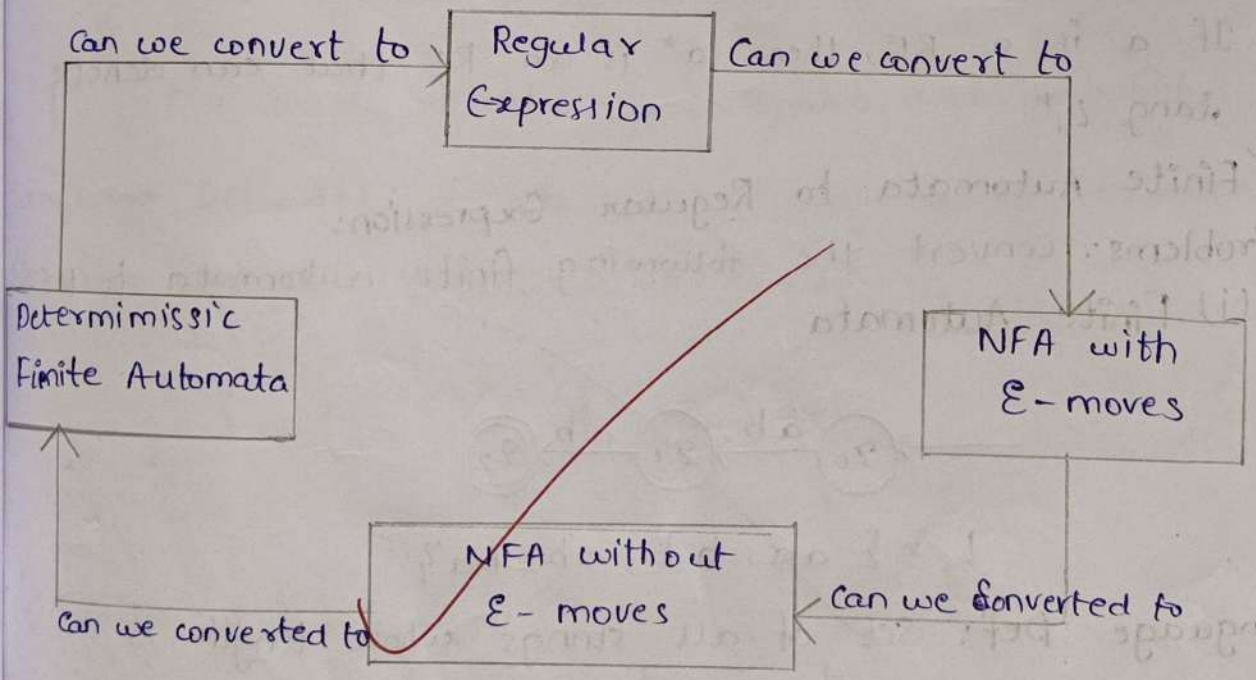
Aim:

(i) To convert Regular Expression to finite Automata and vice-versa using JFLAP.

(ii) To convert Regular Expression to epsilon NFA using JFLAP.

Algorithm:

There is a close relationship between a finite automate and the regular expression. The figure shows that it is the convinent to convert the regular expression to NFA with ε-moves.



A regular expression is a pattern that describes a string (or) language that finite automata accepts.

Regular expression are used to match characters combinations in strings.

Def: An RE is a string of characters that defines a match pattern in test.

RE also known as rational expressions.

RE also used in search engines, word processor and text editors.

RE are used in lexial analysis and input validation.

Examples

→ ϕ is RE, that denotes empty set.

→ ε is a RE that denotes null string.

→ If a and b are RE's then a+b is a RE that denotes the language $L1 \cup L2$

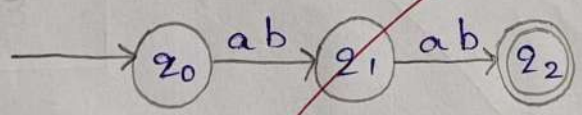
→ If a and b are RE then a b is a RE that denotes $L_1 L_2$.

→ If a is a RE, then a^* is an RE that can denote lang L_1^*

(i) Finite Automata to Regular Expression:

Problems: Convert the following finite automata to RE.

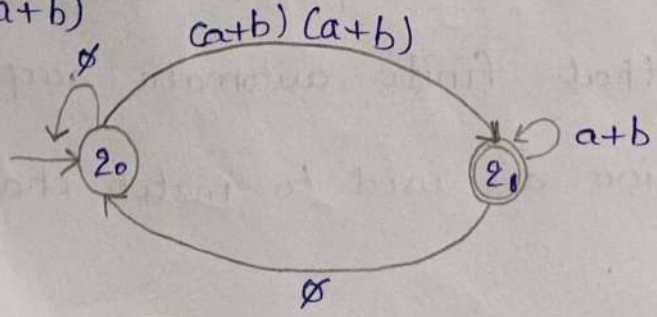
(i) Finite Automata



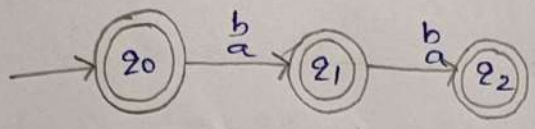
$L = \{aa, bb, ab, ba\}$

Language Def: Set of all strings whose length is exactly 2.

R.E: (a+b)(a+b)



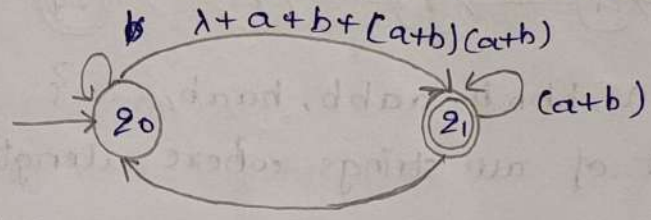
(ii) Finite Automata:



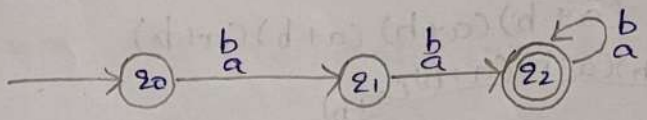
$L = \{ \varepsilon, a, b, ab, ba, bb, aa, \ldots\ldots \}$

Language Definition: Set of all strings whose length is atmost 2.

Regular Expression: $\lambda + a+b + (a+b)(a+b)$



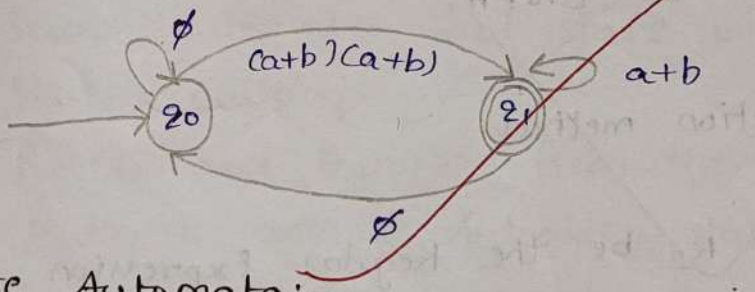
(iii) Finite Automata:



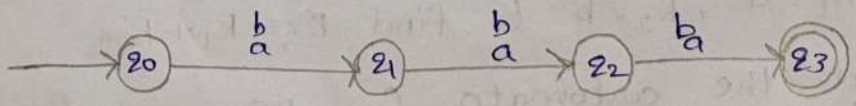
$L : \{ aa, bb, ab, aa, aab, aba, aab, bab, bbb, \ldots \}$

Language Def: Set of all strings where length is atleast 2.

Regular Expression: $(a+b)(a+b)(a+b)^*$



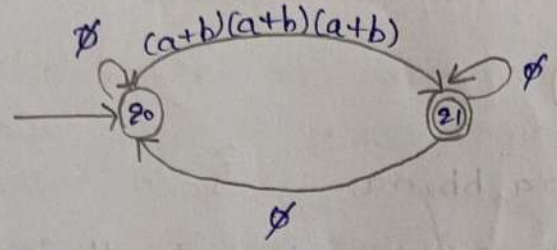
(iv) Finite Automata:



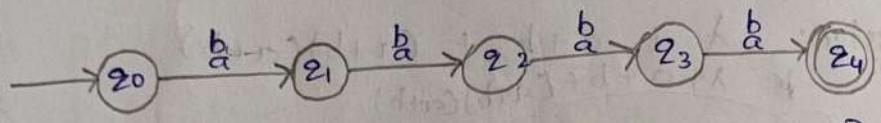
$L = \{ aaa, bbb, aba, bab, abb, baa, \ldots\ldots \}$

Language Def: Set of all strings whose length is exactly 3

Regular Expression: $(a+b)(a+b)(a+b)$



(v) Finite Automata:

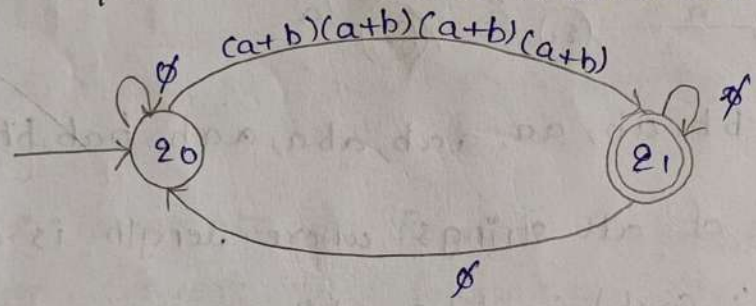


$L = \{aaaa, bbbb, aabb, baab, \ldots\}$

Language Def: Set of all strings where length is exac

4.

Regular Expression: $(a+b)(a+b)(a+b)(a+b)$



i) Regular Expression to ε NFA:

Procedure:

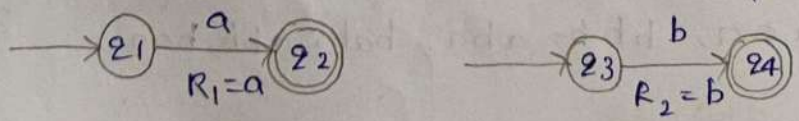
→ Thomson Construction method.

(a) Union case:

Let R_1 and R_2 be the Regular Expression

$R_1 = a$, $R_2 = b$ to find $R_3 = R_1 + R_2$

(i) construct the automata for RE R_1 and R_2

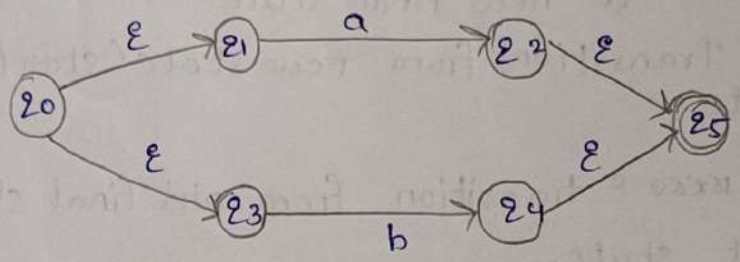


(ii) Draw the 2 Automata parallely. Then create a new start state and new final state.

(iii) Create the null transitions as follows:

a) Create the ε Transition from new state to existing automata start state.

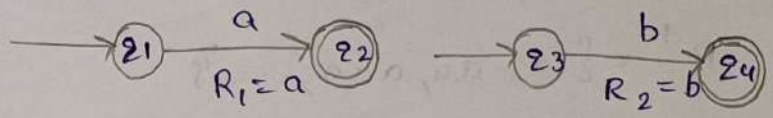
b) Create ε Transition from existing Automata to new final state



b) con catenation case:

Let R_1 and R_2 be the two regular expressions $R_1 = a$ & $R_2 = b$ to find $R_3 = R_1 R_2$.

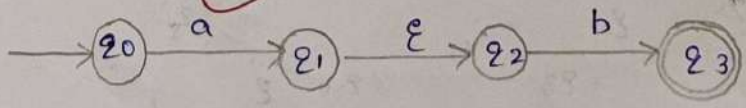
(i) Construct the automata for RE R_1 and R_2



(ii) Draw the two automata serially such that the first automata start will be the start state ~~of~~ resulting Automata.

(iii) Second automata final state will be the final state resulting.

(iv) Provide and transition from the Automata's final state to second Automata state.



c) Closure case

Kleene closure:

Let R_1 be the RE, $R_1 = a$, $R_1^* = a^*$

(i) Construct the Automata from RE R_1

(ii) Create a new transition state q_0 and a new final

state.

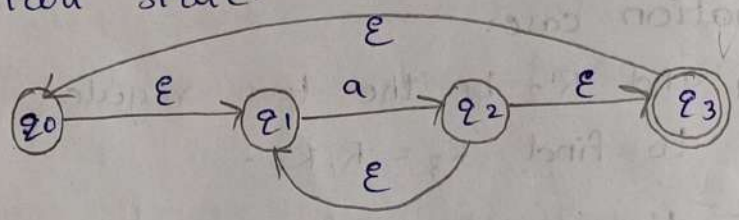
(iii) Provide ε Transition as follows.

a) Create a ε Transition from new state to existing automata start state.

b) Create a ε Transition from existing Automata's final state to new final state.

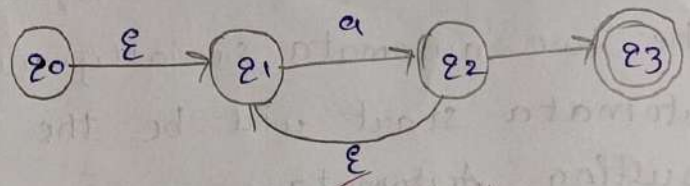
c) Create ε Transition from new state (start) to new final state.

d) Creat a ε transition from old final state to old start state.



(iv) Positive Closure:

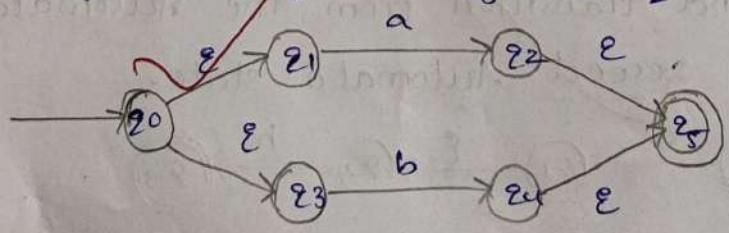
$R^+ = \{a, aa, aaa, \ldots\}$



Problem: Convert the RE $(a+b)^*$ ~~(ab)b~~ to epsilon NFA

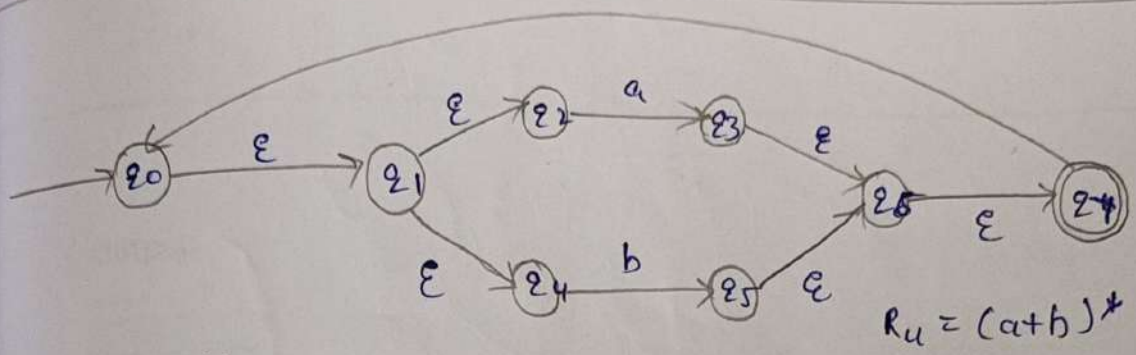
Sol: union case: $(a+b)$

$R_1 = a, \quad R_2 = b \quad R_3 = R_1 + R_2 = (a+b)$



Closure Case: $(a+b)^*$

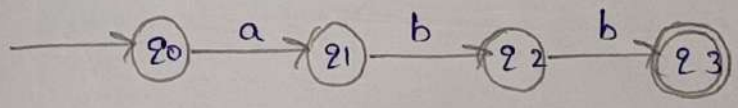
$R_4 : (R_3)^* = (a+b)^*$



$R_4 = (a+b)^*$

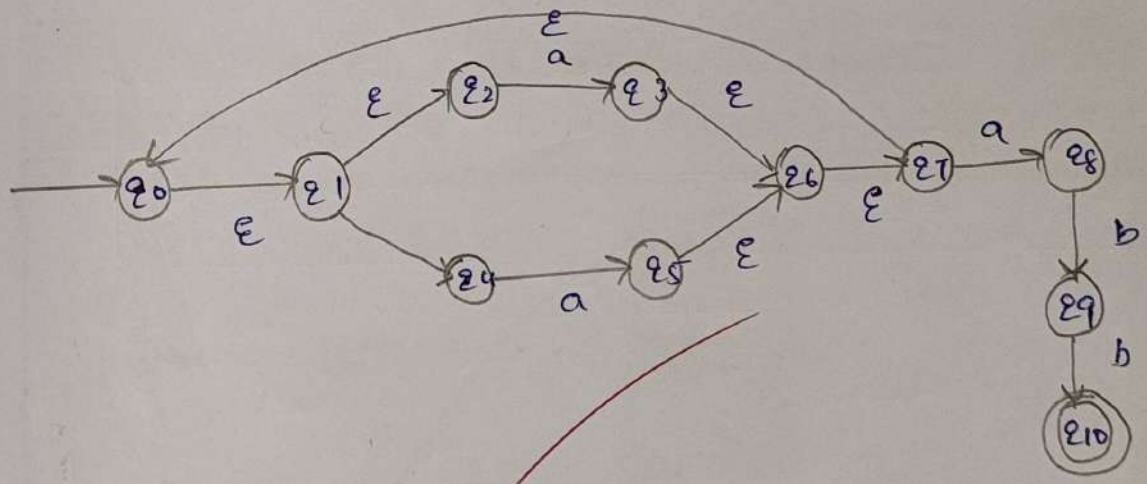
Concatenation case: a bb

$R_5 = abb$

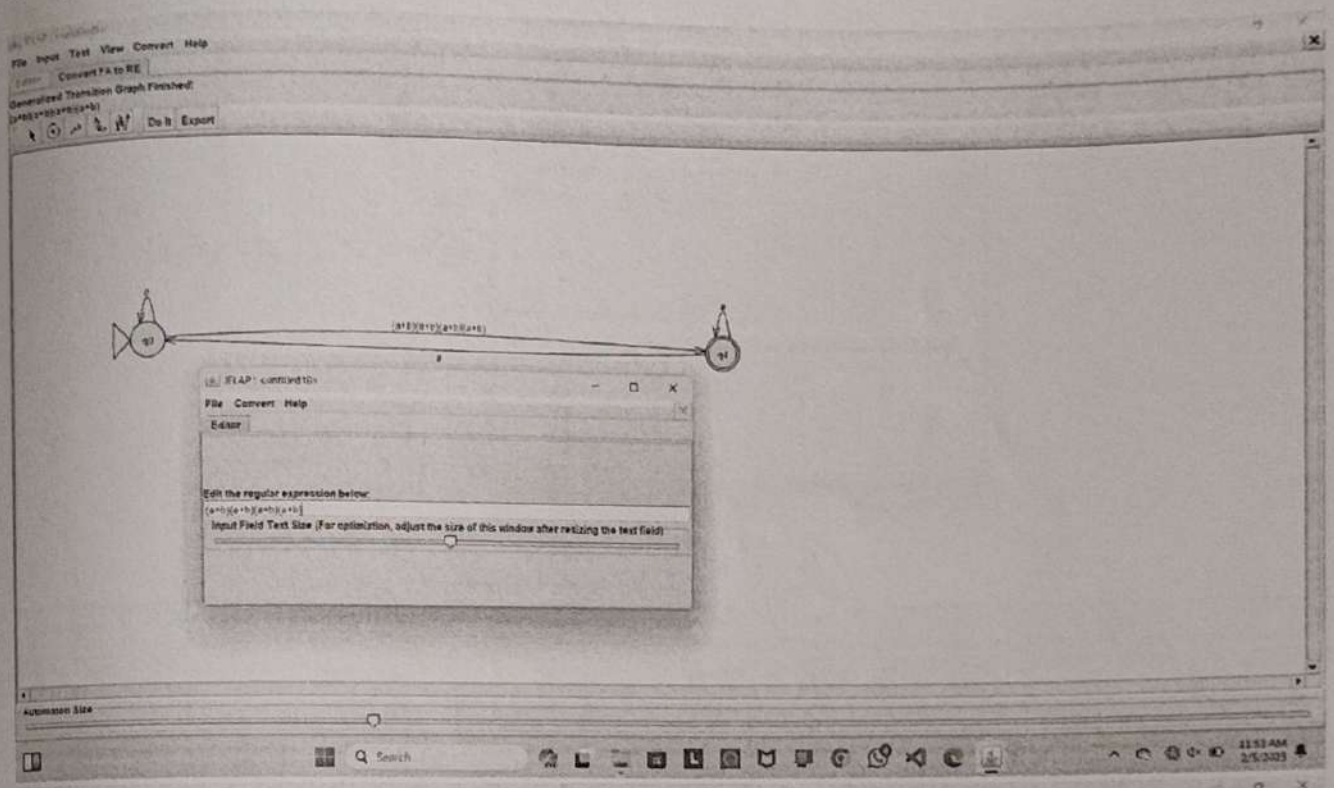


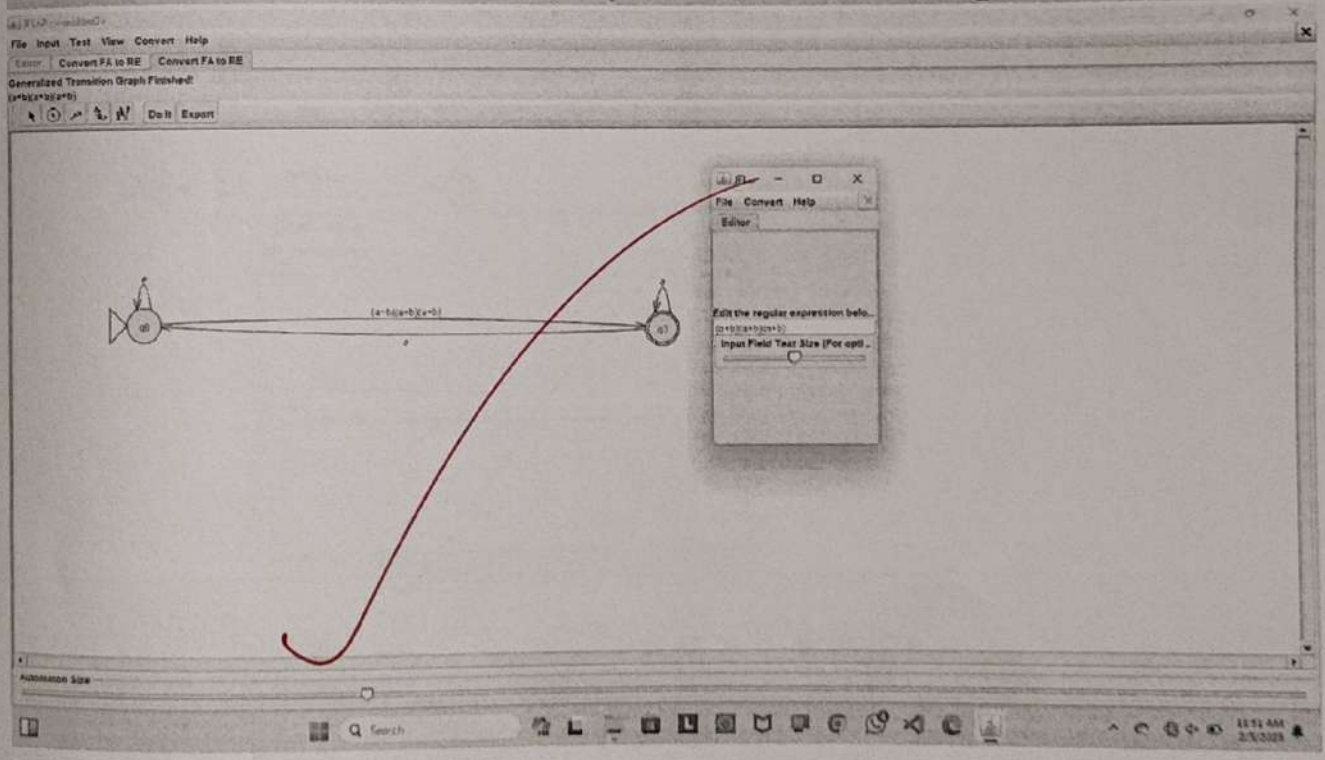
Final epsilon NFA:

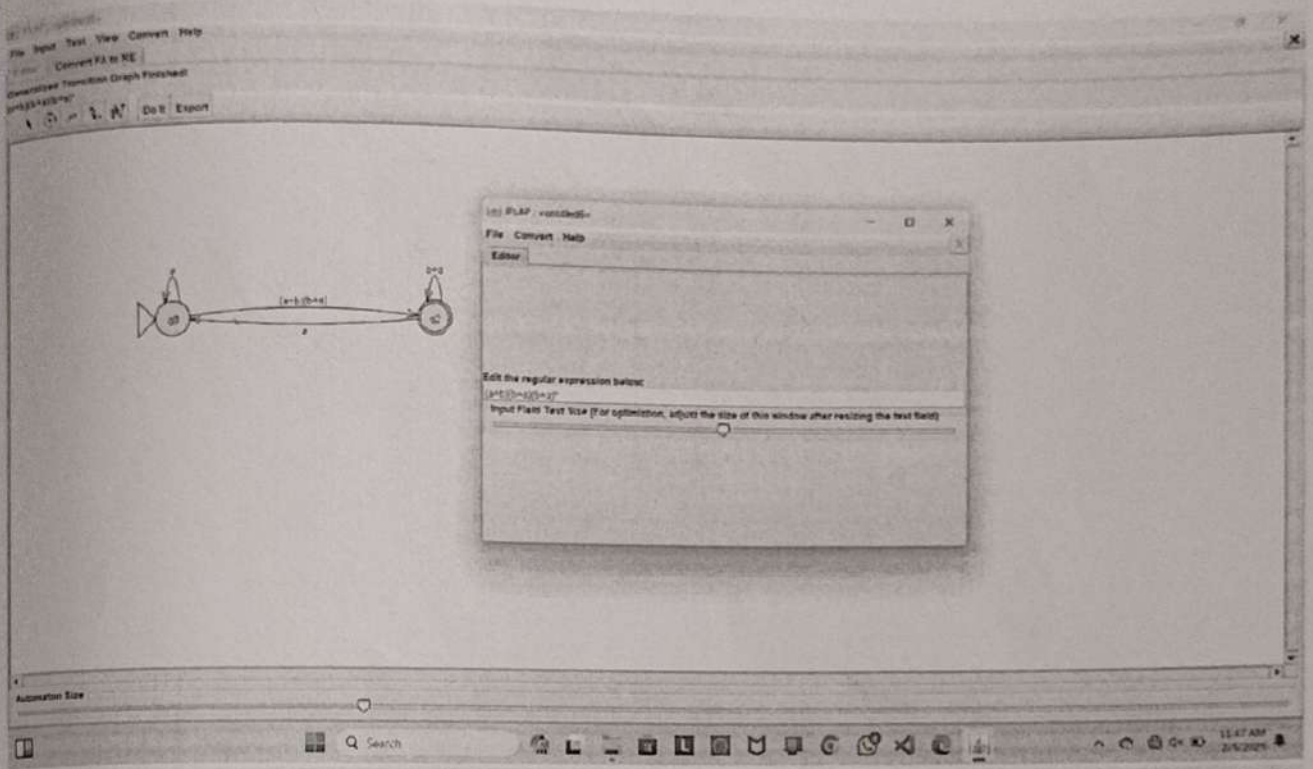
$R_6 = R_4 . R_5 = (a+b)^* abb$

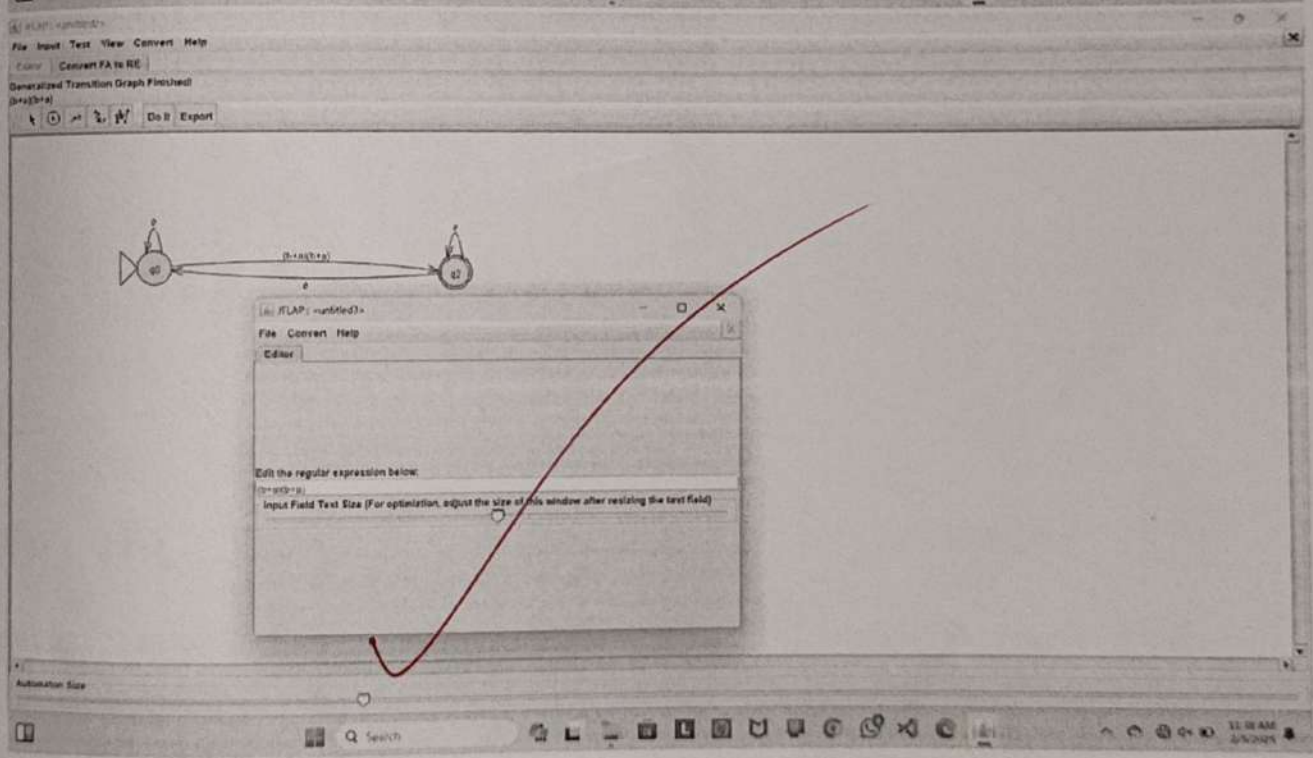


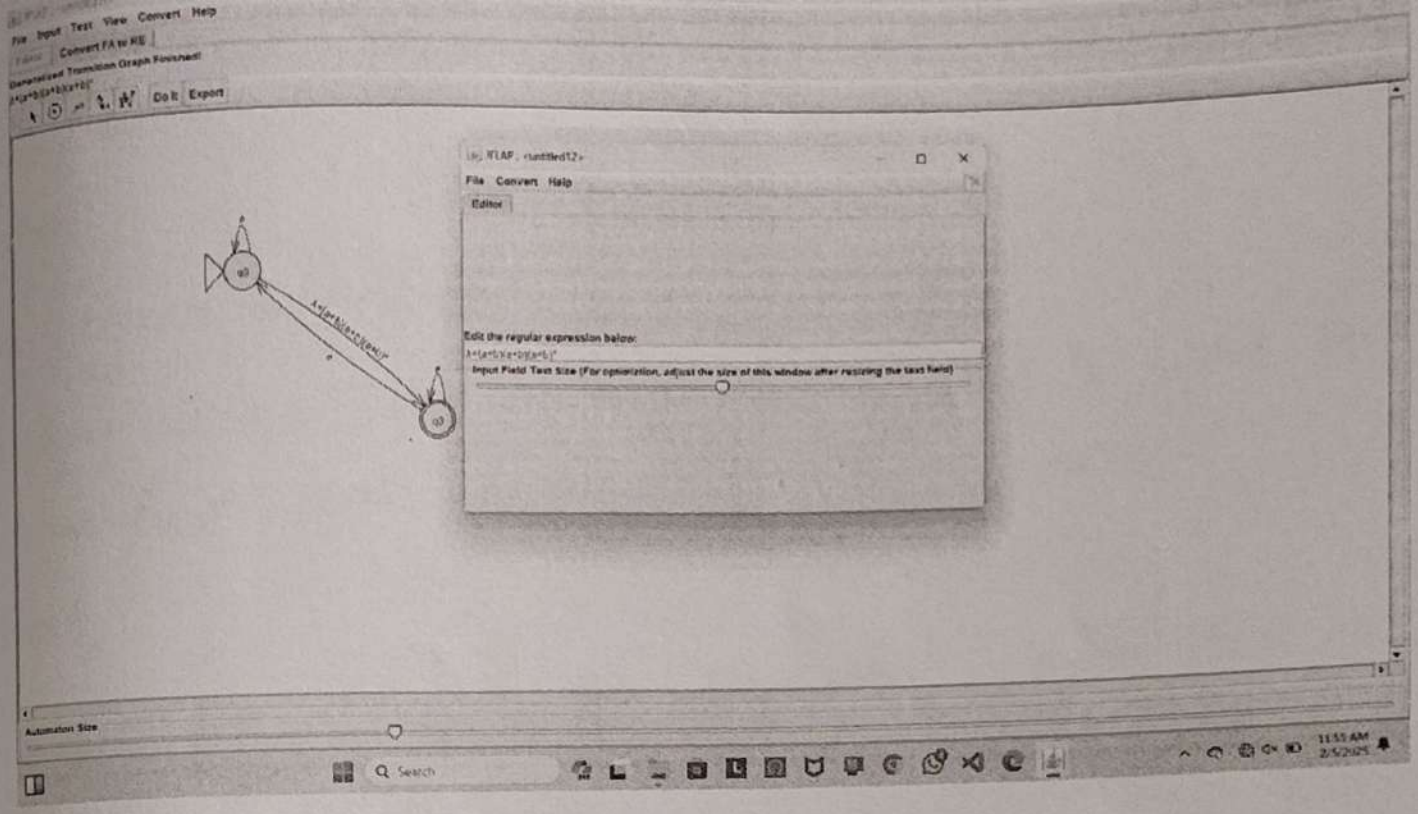
Output:


File Input Test View Convert Help
Convert FA to RE
Generalized Transition Graph Finished!
Do It
Export
File Convert Help
Editor
Edit the regular expression below:
Input Field Text Size (For optimization, adjust the size of this window after resizing the text field)
Automaton Size


File Input Test View Convert Help
Convert FA to RE
Convert FA to RE
Generalized Transition Graph Finished!
Do It
Export
File Convert Help
Editor
Automaton Size


File Input Test View Convert Help
Convert FA to RE
Generalized Transition Graph Finished!
Do It
Export
File Convert Help
Editor
Edit the regular expression below:
Input Field Text Size (For optimization, adjust the size of this window after resizing the text field)
Automation Size
Search


File Input Test View Convert Help
Convert FA to RE
Generalized Transition Graph Finished!
Do It
Export
File Convert Help
Editor
Edit the regular expression below:
Input Field Text Size (For optimization, adjust the size of this window after resizing the text field)
Automation Size
Search



Result:

Thus the regular expression is converted into finite Automata is executed successfully and output is verified.

5/4/25

Viva: 10/10

Experiment: 4

Date: 5/2/25

Proving Language Not to be Regular.

Aim:

To prove the given language is regular or not using pumping lemma with JFLAP

Algorithm:

Pumping Lemma Definition:

Pumping lemma for regular expression (RE) is used to prove that the given language is not regular.

Statement:

Let z be the string in the language 'L' such that z is divided into 3 parts.

$$z = uvw$$

If the string 'z' statistics the below 3 conditions then the string $z \in L$ and language 'L' is called regular language else not regular language.

i) $|uv| \leq c$

ii) $|v| > 0$

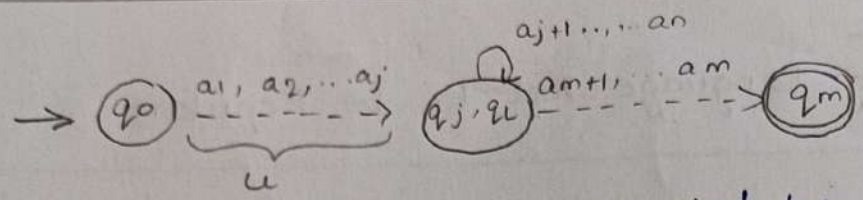
iii) $|uv| \leq c$

iv) for $i \geq 0$ every string $uv^i w \in L$

Proof: All the regular language is accepted by the model finite Automata.

⇒ So there exists a finite automata

$FA = (Q, \Sigma, \delta, q_0, F)$ for all regular language.



* Q is the set of all states denoted by

$Q = \{q_0, q_1, \dots q_j, q_{j+1}, \dots q_k, q_{k+1} \dots q_m\}$

* The input alphabet is given by ϵ.

$\Sigma = \{a_1, a_2, \dots, a_j, a_j, a_{j+1}, \dots a_k, a_{k+1}, \dots a_m\}$

* q_0 is the starting state q_m, is the final state where $m > n$

* The transition function f is given by $\delta(\delta q_0, q_1, \dots q_j, q_{j+1}, \dots q_k, q_{k+1} \dots q_m\}\{a_1, a_2, \dots a_{j-1}, a_j, a_{j+1}, \dots a_k, a_{k+1}, \dots a_m) = q_j$ where $j \geq 1$

$\delta(q_0, q_1, \dots q_{j-1}\}, \{a_1, a_2, \dots a_j\}) = a_j$

$\delta(q_j, \{a_{j+1}, a_{j+2}, \dots a_k\}) = q_k$

$\delta(q_k, \{a_{k+1}, \dots a_m\}) = q_m$

The string $z = uvw$ reaches final states

$\therefore$ The string $z \in L$ and the regular said to be Regular

where,

$u = \{a_1, a_2, \dots a_j\}$

$v = \{a_{j+1}, \dots a_k\}$

$w = \{a_{k+1}, \dots a_m\}$

roblems:

roblem to prove the given language is not Regular using Pumping Lemma

$L = \{a^n b^n / n > 0\}$

Language = $L = \{ab, aabb, aaabbb, aaaabbbb, \dots\}$

case 1: (v contain only 'a')

$z = uvw$

$z = \underbrace{aaa}_{u}\ \underbrace{a}_{v}\ \underbrace{bbbb}_{w}$

$u = aaa \quad v = a \quad w = bbbb$

i) $|v| \geq 0$

$a \geq 0$

$1 \geq 0$

ii) $|uv| \leq c$, $c = 8$

$|uv| \leq 8$

$u \leq 8$

iii) $z = uv^2w$

$= aaa(a)^2(bbbb)$

$= aaaaabbbb$

case 2: v contains on 'b'

$z = uvw$

$z = aaaa\ bbbb$

$u = aaaa \quad v = b \quad w = bbb$

i) $|v| \geq 0$

$b \geq 0$

$1 \geq 0$

ii) $|uv| \leq c$, $c = 8$

$|aaaa\ b| \leq 8$

$5 \leq 8$

iii) $z = uv^2w$

$z = aaaa(b)^2bbb$

$= aaaa\ bbbbb$

case 3: (v contains both 'a' and 'b')

$z = aaaa\ bbbb$

$u = aaa \quad v = ab \quad w = bbb$

i) $|v| \geq 0$

$|ab| \geq 0$

$2 \geq 0$

ii) $|uv| \leq c$; $c = 8$

$|aaaab| \leq c$

$5 \leq 8$

iii) $z = uv^2$

$aaa(ab)^2(bbb)$

$aaa(ab)(ab)(bbb)$

$aaaababbbbb$

$a^5b^5 \notin L$

2) Prove that the given language is not regular using pumping lemma $L = \{a^n b^k c^{n+k} \ (n, k > 0)\}$

Sol: Language, $L = \{abc^2, ab^2c^3, a^2bc^3, a^2b^2, \ldots\}$

case-1 : (v contains only 'a')

$z = a^2b^2c^4$

$= \{aa, bb, ccc\}$

$u = a \quad v = ab \quad w = bbccc$

$z = uvw$

i) $|v| \geq 0$

$|a| \geq 0$

$1 \geq 0$

ii) $|uv| \leq c, \ c = 6$

$|aa| \leq 8$

$2 \leq 8$

iii) $z = uv^2w$

$= (a)(a^2)(bbcccc)$

$z = a^3b^2c^4 \in L$

case 2: (v contains only 'b')

$z = a^2b^2c^4$

$z = aabbcccc$

$u = aa, \ v = b, \ w = bcccc$

i) $|v| \geq 0$

$|b| \geq 0$

$1 \geq 0$

ii) $|uv| \leq c \ ; \ c = 6$

$|aab| \leq 6$

$3 \leq 6$

iii) $z = uv^2w$

$= (aa)(b)^2(bcccc)$

$= aabbbcccc$

case-3: v contains only 'c'

$z = a^2b^2c^4 \Rightarrow z = aabbcccc$

i) $|uv| \leq c \ ; \ c = 6 \Rightarrow |aabbc| \leq 6 \Rightarrow 5 \leq 6$

ii) $|v| \geq 0$

$|c| \geq 0$

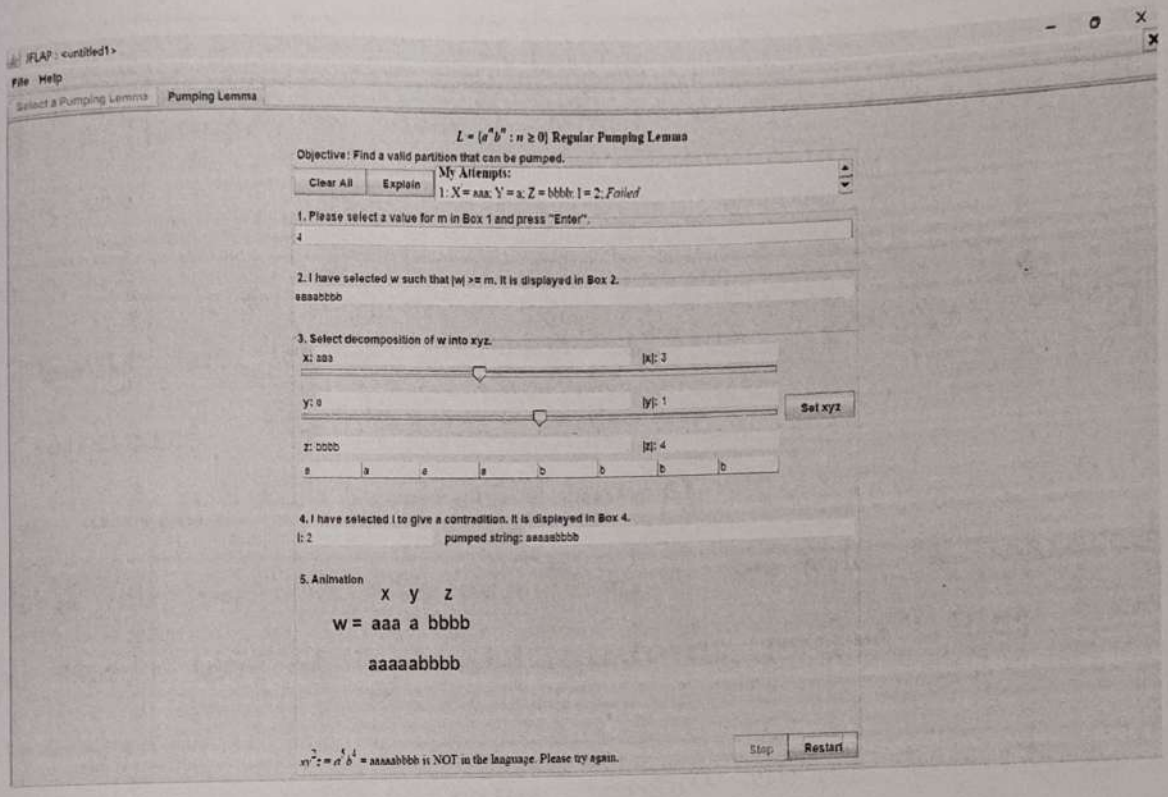
$1 \geq 0$

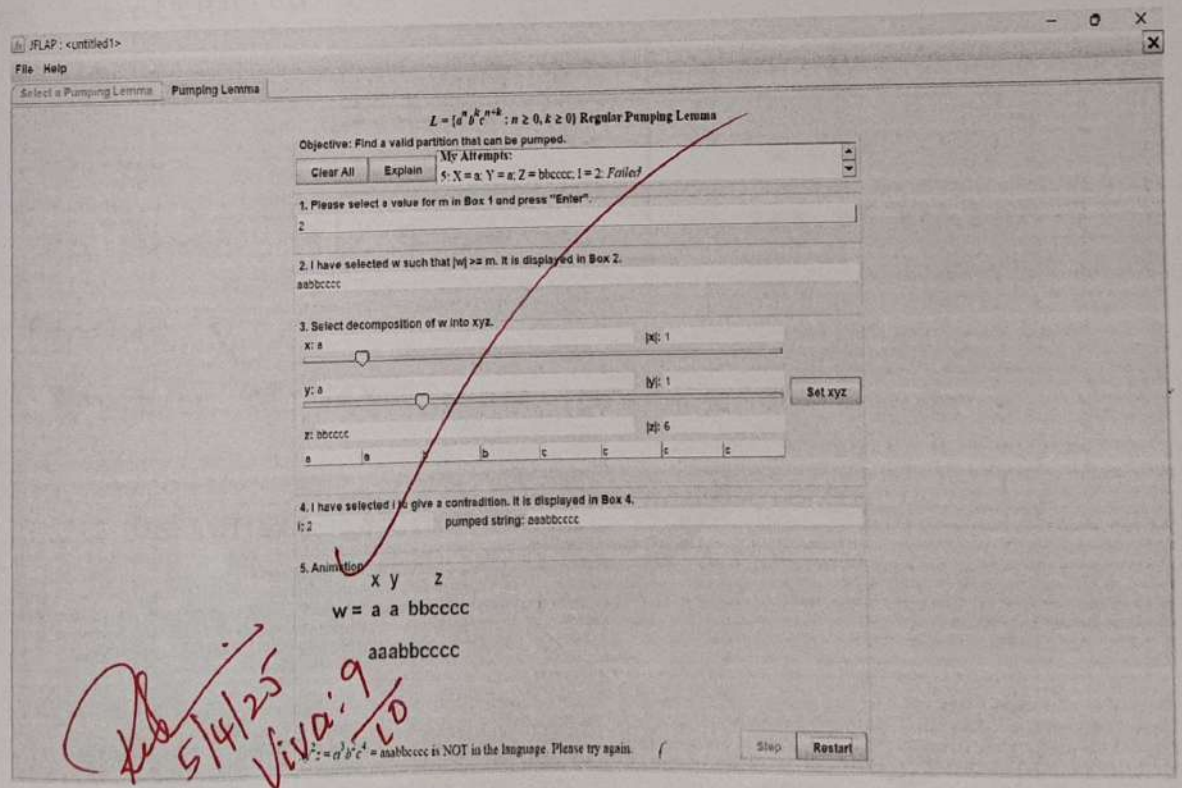
iii) $z = uv^2w = aabb(c)^2ccc$

$= aabbccccc$

$z = a^2b^2c^5$

Hence proved language is not regular.





5/4/25
Viva: 9
LD

Result: Thus the experiment to prove the given language is regular or not using pumping lemma is executed successfully and output is verified.

Experiment : 5	Installing Lex, simple lex program of Lexical Analyzer
Date: 12/2/25	

Aim:

To write a program to identify tokens in the source program using LEX tool.

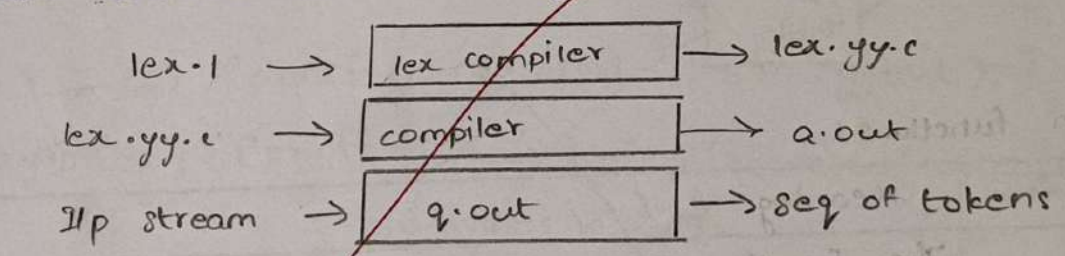
Algorithm:

Lex stands for lexical analyzer. LEX is a unique utility which generates the lexical analyzer.

Lex is a tool for generating scanner

Lex and cc are tightly compiled, a lex file is assesed through the lex utility and produces output files in clex.yy.c.

The form lex.yy.c basically consists of a transition diagram constructed from Regular expressions of first l.l



Specifications:

the structure of lex pgrms consists of 3 parts

%{

Definitions section

%}

%%

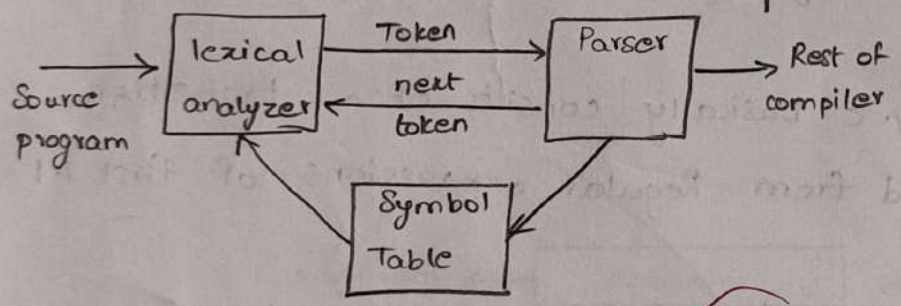
Rules section

%%

Auxilary function.

Regular expression:

RE	meaning
a	Matches a
abc	matches abc
[abc]	Matches a or b or c
[a-f]	Matches a,b,c,d,e,f
[0-9]	Matches any digit
x^+	Matches zero or more x
x^*	Matches one or more x
$[0-9]^+$	Matches any integer



Built in functions:

Function	Meaning
yylex()	The function that starts the analyze.
yywrap()	The function is called when end of the file is encountered.
yyless()	This function is used to push back all but first n characters of token
yymore()	This function tells the lexer to append token
yyerror()	This fn is used for displaying only error message

Program to find identifier:

```
%{
  #include<stdio.h>
%}
%%
[a-z] [A-Z_] [a-z A-Z 0-9 _]* {printf ("Valid identifier \n"); }
                              {printf ("Not an identifier \n"); }
%%
int main () {
      yylex ();
      return 0;
}
int yywrap () {return 1; }
```

Lex program to find operator and sequence of characters.

```
%{
  #include <stdio.h>
%}
%%
 [+ \- * / =] {printf ("operators : %s \n", yytext); }
 (xy) {3, } {printf ("Sequence "xy" repeated atleast
                 3 times \n"); }
 [a-z A-Z] {7, } {printf ("String longer than 6 characters \n")}
                 {; /* ignore other characters*/ }
%%
int main() {
   yylex();
   return 0;
}
int yywrap() {return 1; }
```

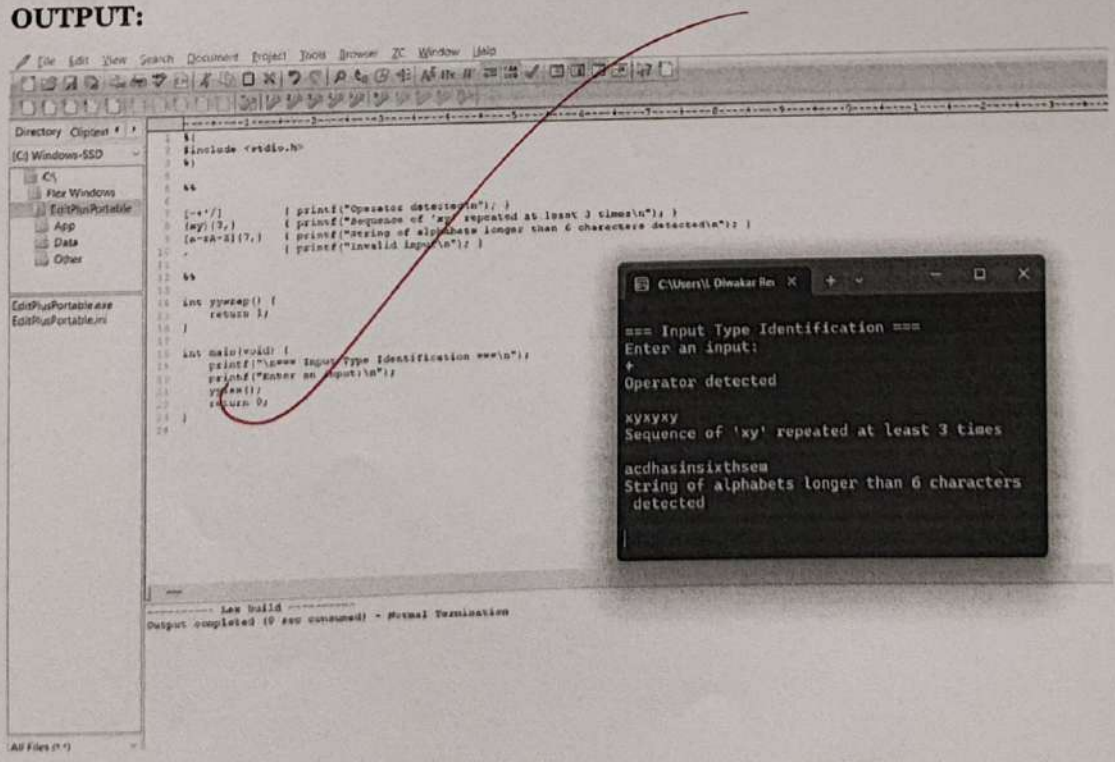
3) lex program to count key words, symbols, Delimiters, Digits

```
%{
    #include <stdio.h>
    #include <string.h>
int keywords =0, symbols =0, delimiters =0, identifiers =0, digits =0;
char *keyword_list[] = {"if", "else", "while", "for", "return", "int", "float"};
int is_keyword (char *word){
        for (int i=0; i<=8; i++){
            if (strcmp (word, keyword_list [i] ==0)
                return 1;
            return 0;
        }
%}
%%
[a-zA-Z_][a-zA-Z0-9_]* {
    if (is_keyword (yytext))
        keywords ++;
    else
        identifiers ++; }
[0-9]+ { digits ++}
[+\-*/=<>] { operators ++; }
[(){}[\],;] { symbols ++}
(is init ) { delimiters ++}
%%
int main (){
    yylex();
    printf ("keyword : %d\n Symbols : %d\n Delimiters %d\n
        identifiers : %d\n Digits : %d\n operators : %d\n", keywords,
        symbols, delimiters, identifiers );
    return 0; }
int yywrap () { return 1; }
```

PROGRAM:

```
%{
#include <stdio.h>
%}
%%
[-+*/]        { printf("Operator detected\n"); }
(xy){3,}      { printf("Sequence of 'xy' repeated at least 3 times\n"); }
[a-zA-Z]{7,}  { printf("String of alphabets longer than 6 characters detected\n"); }
.             { printf("Invalid input\n"); }
%%
int yywrap() {
   return 1;
}
int main(void) {
   printf("\n=== Input Type Identification ===\n");
   printf("Enter an input:\n");
   yylex();
   return 0;
}
```

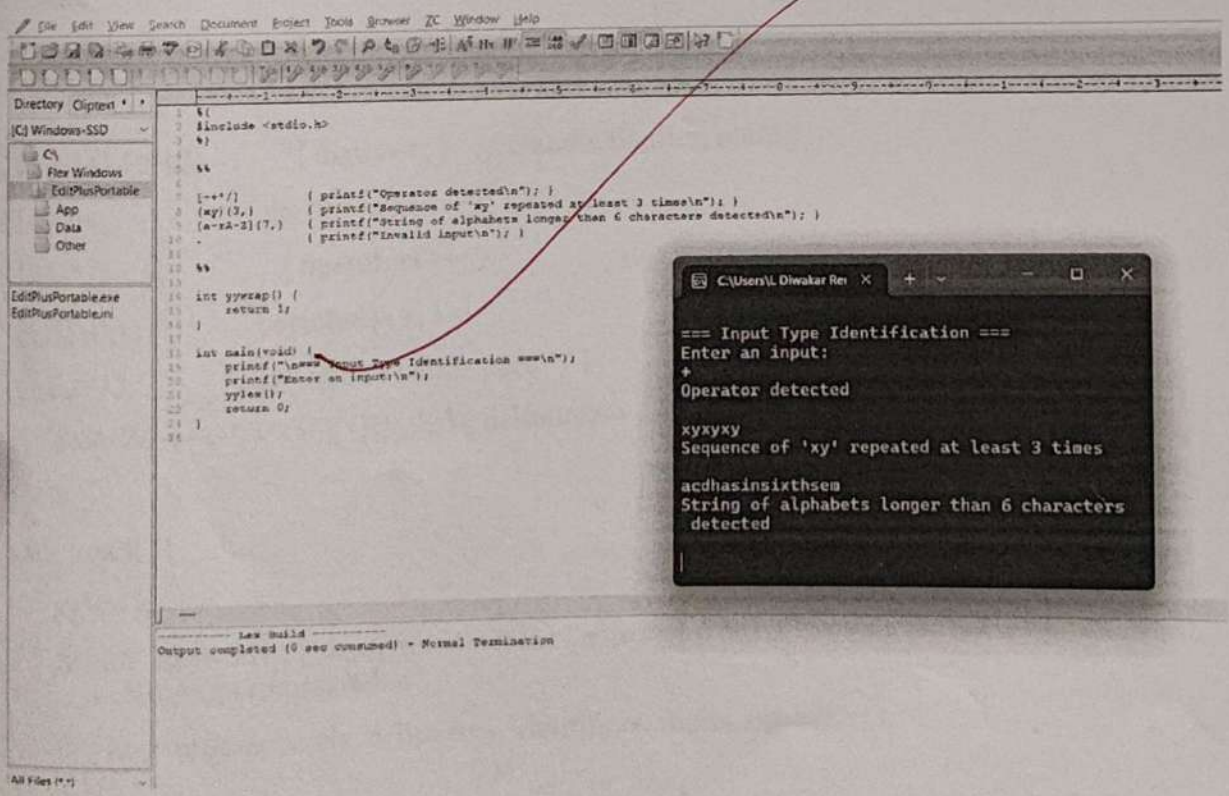
OUTPUT:



PROGRAM:

```
%{
#include <stdio.h>
%}
%%
[-+*/]        { printf("Operator detected\n"); }
(xy){3,}      { printf("Sequence of 'xy' repeated at least 3 times\n"); }
[a-zA-Z]{7,}  { printf("String of alphabets longer than 6 characters detected\n"); }
.             { printf("Invalid input\n"); }
%%
int yywrap() {
  return 1;
}
int main(void) {
  printf("\n=== Input Type Identification ===\n");
  printf("Enter an input:\n");
  yylex();
  return 0;
}
```

OUTPUT:

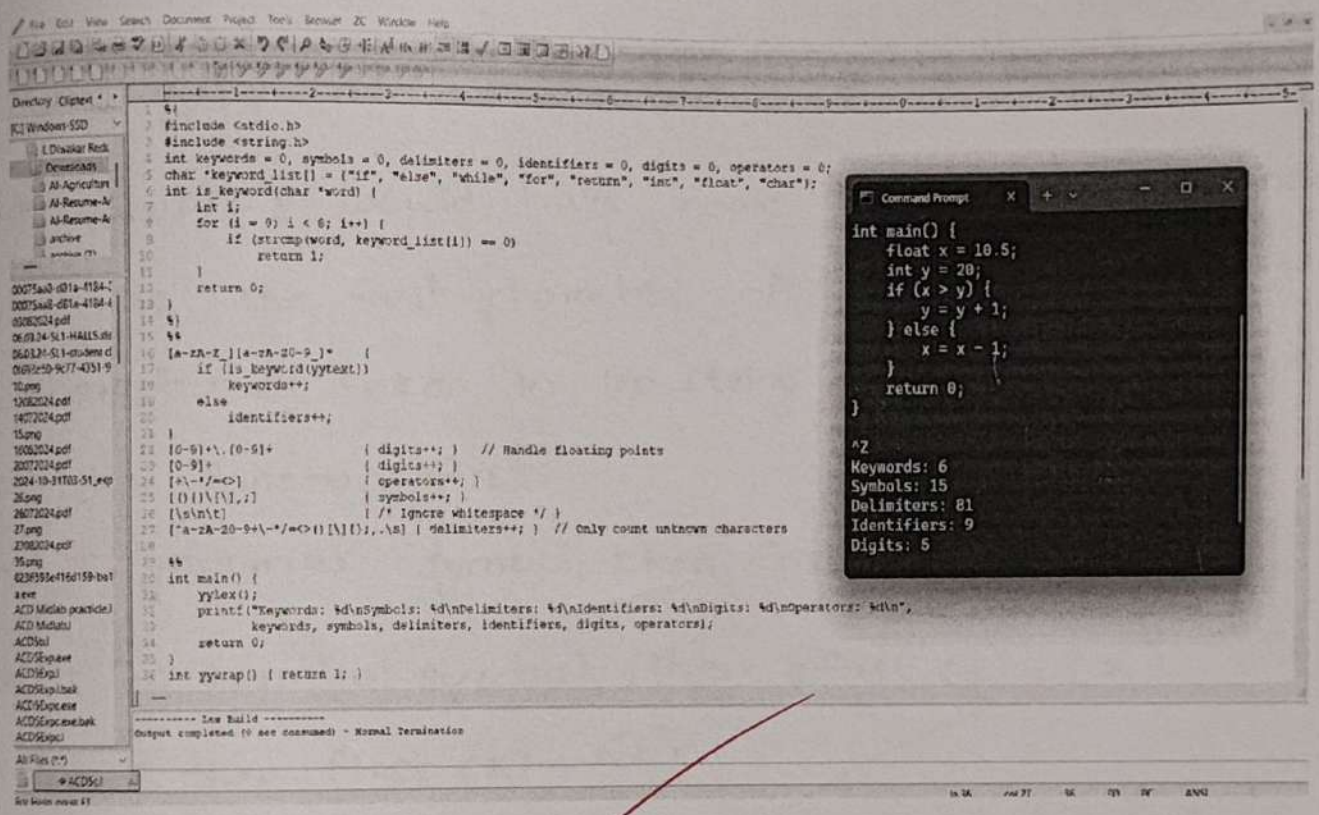


PROGRAM:

```
%{
#include <stdio.h>
#include <string.h>
int keywords = 0, symbols = 0, delimiters = 0, identifiers = 0, digits = 0, operators = 0;
char *keyword_list[] = {"if", "else", "while", "for", "return", "int", "float", "char"};
int is_keyword(char *word) {
  int i;
  for (i = 0; i < 8; i++) {
    if (strcmp(word, keyword_list[i]) == 0)
      return 1;
  }
  return 0;
}
%}
%%
[a-zA-Z_][a-zA-Z0-9_]*   {
  if (is_keyword(yytext))
    keywords++;
  else
    identifiers++;
}
[0-9]+\.[0-9]+        { digits++; }   // Handle floating points
[0-9]+              { digits++; }
[+\-*/=<>]            { operators++; }
[(){}\[\],;]          { symbols++; }
[\s\n\t]              { /* Ignore whitespace */ }
[^a-zA-Z0-9+\-*/=<>()[\]{};,.\s] { delimiters++; }  // Only count unknown characters
%%
int main() {
  yylex();
  printf("Keywords: %d\nSymbols: %d\nDelimiters: %d\nIdentifiers: %d\nDigits:
      %d\nOperators: %d\n",
      keywords, symbols, delimiters, identifiers, digits, operators);
```

```
    return 0;
}
int yywrap() { return 1; }
```

OUTPUT:



Result:

Thus the program to identify tokens in source program using LEX tool is executed successfully and output is verified.

5/4/25

Viva: 10/10

periment : 6	Compute First and Follow functions using
ate: 19/2/25	Predictive Parsing.

m:

To find first and follow of a given context free grammar.

scription:

hy first ?

To avoid base tralking in parsing we need to calculate

irst $S \rightarrow cAd$ and the I/p string is "cad".

$A \rightarrow bc|a$

If the compiler would have come to know in advance

hat what is the first character of the string product

urrent character or token in I/p string is less it can wisely

ompute the function first:

If x is grammar symbol, then first(x) will be

) If x is a terminal symbol, then first $(x) = (x)$

If $x \rightarrow \varepsilon$ then first $(x) = \{\varepsilon\}$

) If x is non-terminal $X \rightarrow a\alpha$ then first $(x) = \{a\}$

) If $X \rightarrow y_1 y_2 y_3$ then first(X) will be

a) If y_1 is terminal then

first (x) = first $(y_1 y_2 y_3) = \{y_1\}$.

b) If y_1 is non terminal then

If y_1 doesnt derive to an empty string i.e, if first(y_1) doesn't contain ε then

first (X) = first (y_1, y_2, y_3) = first (y_1)

i.e, If first (y_1) contains ε, then

first (X) = first (y_1, y_2, y_3) = first $(y_1) - \{\varepsilon\} \cup$ first (y_2, y_3)

first (y_2, y_3) = first (y_2), if first (y_2) doesn't contain ε

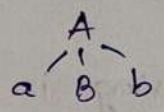
Why follow?

The parsing focus one more problem let us consider below grammar to understand this problem.

$A \rightarrow aBb$

$B \rightarrow c \mid \varepsilon$

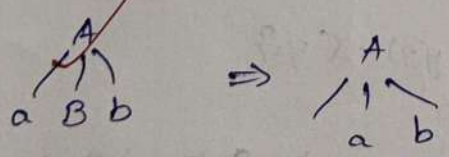
And supper the o/b or I/b string is "ab" to parse as the 1st character in the I/P is a, the parser applies the rule $A \rightarrow aBb$



Now the parser checks for the 2nd character of the I/P string which is b and the non terminal to derive is B, but the parser can't get any string derivable from B that contains b as 1st character

But the grammar does not contain a production rule $B \rightarrow \varepsilon$. If that is applied then B will vanish and the parser gets the I/P 'ab', as shown below

on RHS of $A \rightarrow aBb$, b follows non-terminal B. (ie) follow (B) = {b} and the current i/p character read is also b



So follow can make a non-terminal vanish.

```python
from collections import defaultdict

def compute_first(grammar, symbol, first):
    if symbol in first:
        return first[symbol]

    first[symbol] = set()
    productions = grammar[symbol]

    for production in productions:
        if production == "ε":
            first[symbol].add("ε")
        else:
            for char in production:
                if char.isupper():  # Non-terminal
                    first_char = compute_first(grammar, char, first)
                    first[symbol].update(first_char - {"ε"})
                    if "ε" not in first_char:
                        break
                else:  # Terminal
                    first[symbol].add(char)
                    break
            else:
                first[symbol].add("ε")

    return first[symbol]

def compute_follow(grammar, start_symbol, first, follow):
    follow[start_symbol].add("$")

    for symbol in grammar:
        for production in grammar[symbol]:
            for i, char in enumerate(production):
                if char.isupper():
                    if i + 1 < len(production):
                        next_char = production[i + 1]
                        if next_char.isupper():
                            follow[char].update(first[next_char] - {"ε"})
                            if "ε" in first[next_char]:
                                follow[char].update(follow[symbol])
                        else:
                            follow[char].add(next_char)
                    else:
                        follow[char].update(follow[symbol])

def main():
    grammar = {
        "S": ["cAd"],
        "A": ["bc", "a"]
    }

    first = {}
    follow = defaultdict(set)
    start_symbol = "S"

    for non_terminal in grammar:
        compute_first(grammar, non_terminal, first)

    compute_follow(grammar, start_symbol, first, follow)

    print("FIRST sets:")
    for non_terminal in first:
        print(f"FIRST({non_terminal}): {first[non_terminal]}")

    print("\nFOLLOW sets:")
    for non_terminal in follow:
        print(f"FOLLOW({non_terminal}): {follow[non_terminal]}")

if __name__ == "__main__":
    main()
```

```
FIRST sets:
FIRST(S): {'c'}
FIRST(A): {'b', 'a'}

FOLLOW sets:
FOLLOW(S): {'$'}
FOLLOW(A): {'d'}
```

Result:

Thus the program to find first and follow of a give context free grammar is executed successfully and output is verified.

3/4/25 Viva: 9/10

Experiment: 7 | Construct of Predictive Parsing Table

Date: 26/2/25

Aim:

To implement the predictive parsing table using any programming language.

Algorithm:

Step-1: Take input as grammar as G.

Step-2: Output as parsing table M.

Step-3: for each instruction $A \to \alpha$. for grammar do step 4 & 5

Step-4: for each terminal 'a' in first(α) add $A \to \alpha$ to $M[A, \alpha]$

Step-5: If ε is in first(α) add $A \to \alpha$ to $M[A, b]$ for each terminal 'b', in follow(A). If ε is in first(α) and \$ is follow(A)
Add $A \to \alpha$ to $M[A, \$]$

Step-6: Make each undefined entry of 'M' as error

1) Eliminate left recursion and left factoring
2) compute first()
3) Compute follow()
4) construct the predictive parsing table
5) Parse the string 'w' using the predictive parsing algorithm.

```
from collections import defaultdict

def compute_first(grammar, symbol, first):
    if symbol in first:
        return first[symbol]

    first[symbol] = set()
    productions = grammar[symbol]

    for production in productions:
        if production == "ε":
            first[symbol].add("ε")
        else:
            for char in production:
                if char.isupper():  # Non-terminal
                    first_char = compute_first(grammar, char, first)
                    first[symbol].update(first_char - {"ε"})
                    if "ε" not in first_char:
                        break
                else:  # Terminal
                    first[symbol].add(char)
                    break
            else:
                first[symbol].add("ε")

    return first[symbol]

def compute_follow(grammar, start_symbol, first, follow):
    follow[start_symbol].add("$")

    for _ in range(len(grammar)):  # Iterating multiple times to ensure completeness
        for symbol in grammar:
            for production in grammar[symbol]:
                for i, char in enumerate(production):
                    if char.isupper():
                        if i + 1 < len(production):
                            next_char = production[i + 1]
                            if next_char.isupper():
                                follow[char].update(first[next_char] - {"ε"})
                                if "ε" in first[next_char]:
                                    follow[char].update(follow[symbol])
                            else:
                                follow[char].add(next_char)
                        else:
                            follow[char].update(follow[symbol])

def construct_parsing_table(grammar, first, follow):
    table = defaultdict(dict)

    for non_terminal in grammar:
        for production in grammar[non_terminal]:
            first_set = set()
            if production == "ε":
                first_set = follow[non_terminal]
            else:
                for symbol in production:
                    if symbol.isupper():
                        first_set.update(first[symbol] - {"ε"})
                        if "ε" not in first[symbol]:
                            break
                    else:
                        first_set.add(symbol)
```

```
                [non_terminal][terminal] = production

    return table

def main():
    grammar = {
        "S": ["cAd"],
        "A": ["bc", "a"]
    }

    first = {}
    follow = defaultdict(set)
    start_symbol = "S"

    for non_terminal in grammar:
        compute_first(grammar, non_terminal, first)

    compute_follow(grammar, start_symbol, first, follow)

    parsing_table = construct_parsing_table(grammar, first, follow)

    print("FIRST sets:")
    for non_terminal in first:
        print(f"FIRST({non_terminal}): {first[non_terminal]}")

    print("\nFOLLOW sets:")
    for non_terminal in follow:
        print(f"FOLLOW({non_terminal}): {follow[non_terminal]}")

    print("\nPredictive Parsing Table:")
    for non_terminal in parsing_table:
        for terminal in parsing_table[non_terminal]:
            print(f"M[{non_terminal}, {terminal}] = {parsing_table[non_terminal][terminal]}")

if __name__ == "__main__":
    main()
```

```
FIRST sets:
FIRST(S): {'c'}
FIRST(A): {'a', 'b'}

FOLLOW sets:
FOLLOW(S): {'$'}
FOLLOW(A): {'d'}

Predictive Parsing Table:
M[S, c] = cAd
M[A, b] = bc
M[A, a] = a
```

Start coding or generate with AI.

Result:

Thus the program to implement predictive parsing table is executed successfully and output is verified.

5/4/25
Viva: 10/10

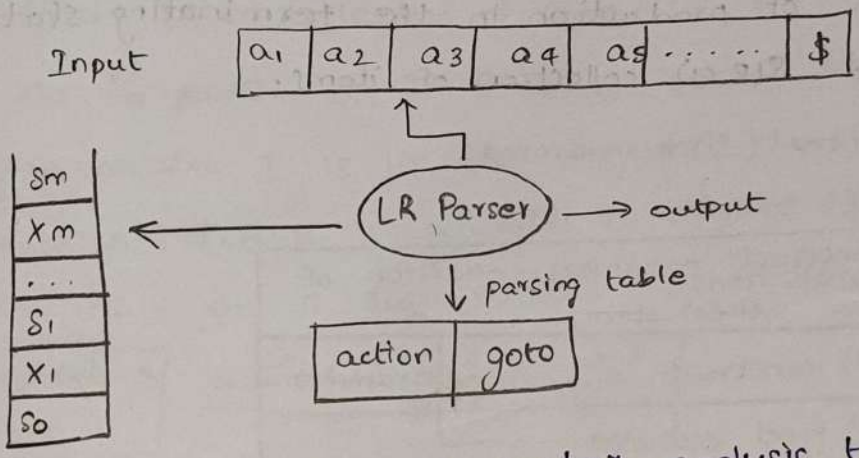
Experiment : 8	Construction of SLR Parsing Table
Date: 5/3/25	

Aim:

To write a program for implementing SLR bottom up parser for given grammar.

Theory:

LR Parser:



It is an efficient bottom up syntax analysis technique that can be used to prove large classes of context free grammar is called LR(0) parsing.

L stands for left to right scanning

R stands for right most derivation in reverse.

0 stands for no. of input symbols of lookahead.

Advantages of LR parsing:

1) It recognizes virtually as programming language constructs CFG can be written.

2) It is able to detect syntatic errors

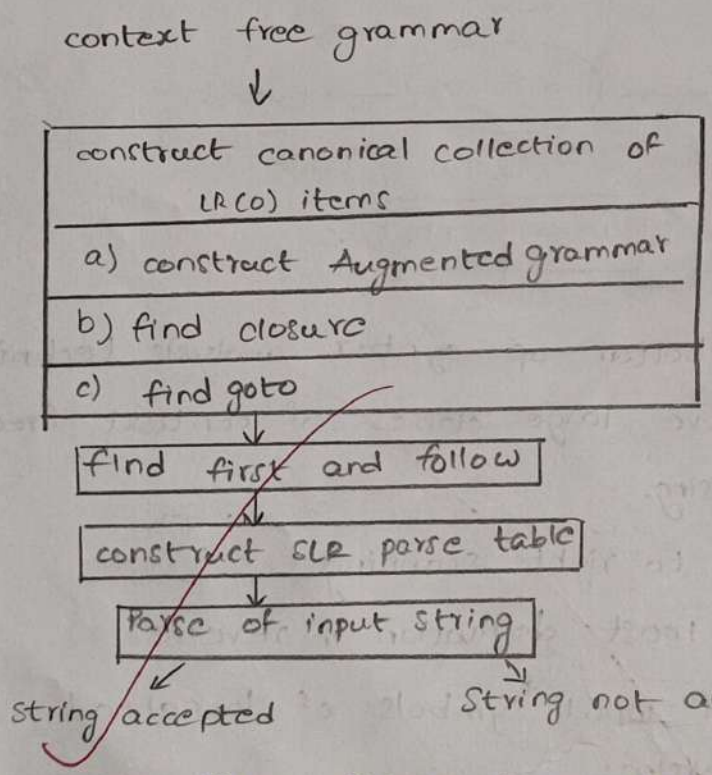
3) It is an efficient non back tracking shift reducing parsing method.

Types of LR parsing methods:

1) SLR 2) CLR 3) LALR

SLR Parser:

SLR is simple LR. It is the smallest class of grammar having few no. of states. SLR is very easy to construct and is similar to LR parsing. The only difference between SLR parser and LR(0) parser is that in LR(0) parsing table, there is a chance of 'shift reduce' conflict because we are entering 'reduce' correspond to all terminal states. we can solve this problem by entering 'reduce' corresponding to follow of LHS of production in the terminating state. This is called SLR(1) collection of items.



Steps for constructing SLR parsing table:

1) writing augmented grammar
2) LR(0) collection of items to be found
3) find follow of LHS of production
4) Defining 2 functions: goto [list of terminals] and action [list of non-terminals] in the parsing table.

mple: Construct LR parsing table for given context free mmar

$S \to AA$

$A \to aA \mid b$

Step 1: find Augmented Grammar:

$S' \to .S$

$S \to .AA$

$A \to .aA$

$A \to .b$

ep 2: find LR(0) collection of items

sure: for a CFG G, If I is the set of items then

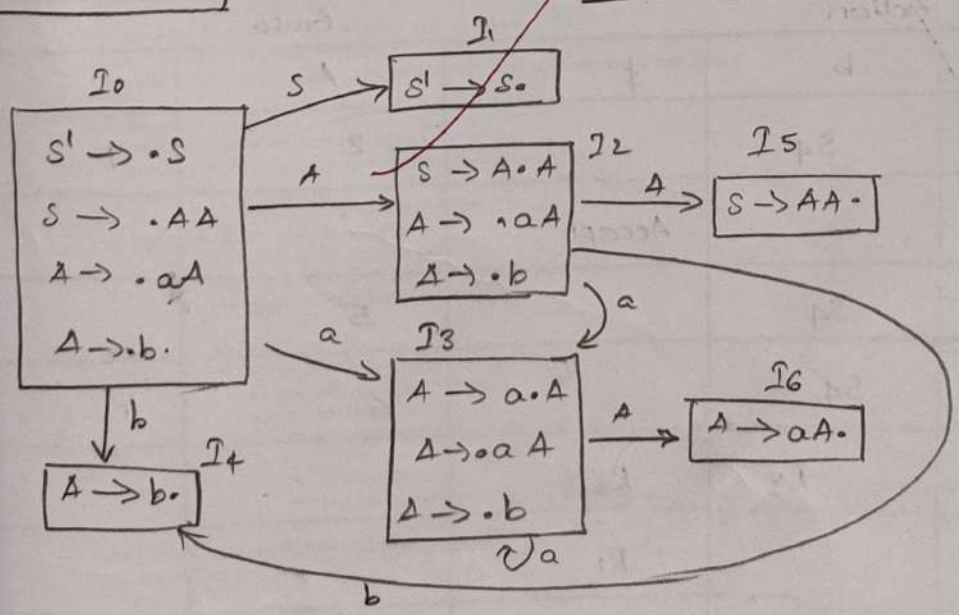
→ Every item in I is in the closure (I)

→ If rule $A \to \alpha . B . \beta$ is a rule in closure (I) and there is another rule for B such as $B \to \gamma$ then closure (I) $A \to \alpha . B . \beta$

$$\boxed{\begin{array}{l} A \to \alpha . B . \beta \\ B \to \gamma \end{array}} \xrightarrow{\text{closure } (I)} \boxed{\begin{array}{l} A \to \alpha . B . \beta \\ B \to . \gamma \end{array}}$$

goto (I, X) - If there is a production $A \to \alpha . X . \beta$ in I then goto (I, X) is defined as closure of set of items $A \to \alpha X . \beta$.

$$\boxed{A \to \alpha . X . \beta} \xrightarrow{\text{goto } (I, X)} \boxed{\text{closure } (A \to \alpha X . \beta)}$$



Step 3: find follow of LHS

follow (S) = $ follow (A) = a, b, $

Step 4:

Defining 2 functions : goto [list of non-terminals] and action [list of terminals]

Algorithm:

Input - An augmented Grammar G

Output - SLR parsing table.

Method:

1) Initially construct set of items

$C = \{I_0, I_1, I_2, \ldots I_n\}$ where C is set of LR(0) items

2) Parsing actions are parsed on each item of I_i

→ If $A \to \alpha \cdot a\beta$ is in I_i goto $(I_i, a) = I_j$ Action (i,a) = shif (j)

→ If $A \to \alpha \cdot$ is in I_i; Action (i,a) "reduce $A \to \alpha$" where $a \in$ follow (A)

→ If $S' \to S \cdot$ in I_i Action [i,$] = accept.

3) All entries not defined by rules 2 & 3 are considered as "error".

	Action			Goto	
	a	b	$	A	S
0	S3	S4		2	1
1			Accept		
2	S3	S4		5	
3	S3	S4		6	
4	R3	R3	R3		
5			R1		
6	R2	R2	R2		

```
class SLRParser:
    def __init__(self, grammar, terminals, non_terminals, start_symbol):
        self.grammar = grammar
        self.terminals = terminals
        self.non_terminals = non_terminals
        self.start_symbol = start_symbol
        self.first_sets = {nt: set() for nt in non_terminals}
        self.follow_sets = {nt: set() for nt in non_terminals}
        self.parsing_table = {}

        self.compute_first_sets()
        self.compute_follow_sets()
        self.construct_parsing_table()

    def compute_first_sets(self):
        """ Compute FIRST sets for all non-terminals. """
        for nt in self.non_terminals:
            self.first(nt)

    def first(self, symbol):
        """ Compute FIRST set for a given symbol. """
        if symbol in self.first_sets and self.first_sets[symbol]:
            return self.first_sets[symbol]

        first_set = set()

        if symbol in self.terminals:
            first_set.add(symbol)
        else:
            for production in self.grammar.get(symbol, []):
                if production == 'ε':  # If production contains epsilon
                    first_set.add('ε')
                else:
                    for char in production:
                        char_first = self.first(char)
                        first_set.update(char_first - {'ε'})
                        if 'ε' not in char_first:
                            break
                    else:
                        first_set.add('ε')

        self.first_sets[symbol] = first_set
        return first_set

    def compute_follow_sets(self):
        """ Compute FOLLOW sets for all non-terminals. """
        self.follow_sets[self.start_symbol].add('$')  # Start symbol FOLLOW contains $

        for nt in self.non_terminals:
            self.follow(nt)

    def follow(self, non_terminal):
        """ Compute FOLLOW set for a given non-terminal. """
        if self.follow_sets[non_terminal]:
            return self.follow_sets[non_terminal]  # Avoid redundant recursion

        follow_set = set()
        if non_terminal == self.start_symbol:
            follow_set.add('$')  # Start symbol always contains '$'

        for lhs, productions in self.grammar.items():
            for production in productions:
                if non_terminal in production:
                    index = production.index(non_terminal)
                    rest = production[index + 1:]

                    if rest:
                        first_rest = self.first(rest[0])
                        follow_set.update(first_rest - {'ε'})
                        if 'ε' in first_rest:
                            follow_set.update(self.follow(lhs))
                    else:
                        if lhs != non_terminal:
                            follow_set.update(self.follow(lhs))

        self.follow_sets[non_terminal] = follow_set
        return follow_set

    def construct_parsing_table(self):
        """ Construct the SLR parsing table. """
        for state, production_list in self.grammar.items():
```

```
            for production in production_list:
                first_rhs = self.first(production[0])

                for terminal in first_rhs - {'ε'}:
                    self.parsing_table[(state, terminal)] = f"shift {production}"

                if 'ε' in first_rhs or not production:
                    for terminal in self.follow_sets[state]:
                        self.parsing_table[(state, terminal)] = f"reduce {state} -> {production}"

        self.parsing_table[(self.start_symbol, '$')] = "accept"

    def display_parsing_table(self):
        """ Display the parsing table. """
        print("\nSLR Parsing Table:")
        for key, value in self.parsing_table.items():
            print(f"{key}: {value}")

# Example grammar
grammar = {
    "S": ["AA"],
    "A": ["aA", "b"]
}

terminals = {"a", "b", "$"}
non_terminals = {"S", "A"}
start_symbol = "S"

parser = SLRParser(grammar, terminals, non_terminals, start_symbol)
parser.display_parsing_table()
```

```
SLR Parsing Table:
('S', 'b'): shift AA
('S', 'a'): shift AA
('A', 'a'): shift aA
('A', 'b'): shift b
('S', '$'): accept
```

Result:

Thus the program for implementing SLR bottom up parser for given grammar is executed successfully and output is verified.

5/4/25
Viva: 10/10

periment : 9 te: 12/3/25	Construct The Three Address code for Given Expression

m: To write a program to construct the three address code for the given expression.

Theory:

Three address code

Three address code is an intermediate code

It is used by code optimizer

In three address code, the given expression is broken ion into several parts, These instructions can easily translate into assemble language.

Each three address code instruction has at most three operands. combination of assignment and binary operator.

General form:

a = b op c

a, b, c are operands

op is operator

Ex:

a = b+c

c = a×b

Common 3 address instructions forms:

1. Assignment Statement:

x = y op z and x = op y

Here x, y, and z operands

op → operator

2. Copy Statement

x = y

x and y operands

= is operator

y value is copied into x.

3. conditional jump

if x relop y goto x

x and y operands

relop operators

If condition ' x relop y '

→ The control is sent directly to location of x

→ The statements are skipped.

If condition " x relop y " false then.

* The control is not sent to location by x.
* The next statement in between are skipped.

4. Unconditional jump

goto x

x is the tag or label of target statement

5. Procedure call:

param x call p return %

Ex:

1. write Three Address code for the following expression.

$(a \times b) + (c + d) - (a + b + c + d)$

1) $T_1 = a \times b$
2) $T_2 =$ ur ino's T_1
3) $T_3 = c + d$
4) $T_4 = T_2 + T_3$
5) $T_5 = a + b$
6) $T_6 = T_3 + T_5$
7) $T_7 = T_4 - T_6$

2.) If $A < B$ and $C < D$ then $t = 1$ else $t = 0$

1. If $(A < B)$ goto(3)
2. goto(4)
3. If $(C < D)$ goto(6)
4. $t = 0$
5. goto (7)
6. $t = 1$
7.

PROGRAM:

```
import re

def generate_tac(expression):

    tokens = re.findall(r'[a-zA-Z]+|\d+|[+\-*/<>=()and]', expression)

    temp_count = 1

    label_count = 1

    tac_code = []

    def get_temp():

        nonlocal temp_count

        temp_var = f'T{temp_count}'

        temp_count += 1

        return temp_var

    def get_label():

        nonlocal label_count

        label = f'({label_count})'

        label_count += 1

        return label

    def process_arithmetic(tokens):

        stack = []

        operators = []

        precedence = {'+': 1, '-': 1, '*': 2, '/': 2}

        for token in tokens:

            if token.isainum():

                stack.append(token)

            elif token in precedence:

                while (operators and precedence.get(operators[-1], -1) >= precedence[token]):
```

```
        op = operators.pop()
        right = stack.pop()
        left = stack.pop()
        temp = get_temp()
        tac_code.append(f'{temp} = {left} {op} {right}')
        stack.append(temp)
    operators.append(token)
while operators:
    op = operators.pop()
    right = stack.pop()
    left = stack.pop()
    temp = get_temp()
    tac_code.append(f'{temp} = {left} {op} {right}')
    stack.append(temp)
return stack[0] if stack else None
def process_condition(tokens):
    if "and" in tokens:
        index = tokens.index("and")
        left_expr = tokens[:index]
        right_expr = tokens[index + 1:]
        label1 = get_label()
        label2 = get_label()
        label3 = get_label()
        label_end = get_label()
        tac_code.append(f'If ({left_expr[0]} {left_expr[1]} {left_expr[2]}) goto {label1}')
        tac_code.append(f'goto {label2}')
```

```
        tac_code.append(f'{label1}: If ({right_expr[0]} {right_expr[1]} {right_expr[2]}) goto
{label3}')
        tac_code.append(f'{label2}: t = 0')
        tac_code.append(f'goto {label_end}')
        tac_code.append(f'{label3}: t = 1')
        tac_code.append(f'{label_end}:')
    else:
        process_arithmetic(tokens)
  if any(op in tokens for op in ['<', '>', '=']):
    process_condition(tokens)
  else:
    process_arithmetic(tokens)
  return '\n'.join(tac_code)
# **User Input**
expression = input("Enter an expression: ")
tac_code = generate_tac(expression)
print("\nThree Address Code:")
print(tac_code)
```

OUTPUT:

```
Enter an expression: (a*b)+(c+d)-(a+b+c+d)

Three Address Code:
T1 = a * b
T2 = T1 + c
T3 = T2 + d
T4 = T3 - a
T5 = T4 + b
T6 = T5 + c
T7 = T6 + d
PS C:\Users\madak\OneDrive\Desktop\sc> & C:/Users/madak/AppData/Local/Programs/Python/Python312/python.exe c:/Users/madak/OneDrive/Desktop/sc
Enter an expression: If A<B and C<D then t=1 else t=0

Three Address Code:
If (If A <) goto (1)
goto (2)
(1): If (C < D) goto (3)
(2): t = 0
goto (4)
(3): t = 1
(4):
```

Result: Thus the program to construct the three address code for given expression is executed successfully and output is verified.

5/4/25 Viva: 10/10

Experiment: 10 Implementation of 3 Address Code

Date: 26/3/25 CIRCPLES, QUADRUPLES AND INDIRST TRIPLES

Aim:

To write a program to implement a code that converts given expression to triples, quadruples, and indirect triples.

Theory:

1. Quadruples
2. Triples
3. Indirect Triples

1. Quadruple:

It is a structure with consists of 4 fields namely op, arg1, arg2 and result. op denotes the operator and arg1 and arg2 denotes the two operands and result is used to store the result of the expression.

Example:

a+bxc / c↑f +bxc

Three Address Code

$T_1 = e \uparrow f$

$T_2 = b \times c$

$T_3 = T_2 / T_1$

$T_4 = b \times a$

$T_5 = a + T_3$

$T_6 = T_5 + T_4$

location	op	Arg 1	Arg 2	Result
(0)	↑	e	f	T_1
(1)	x	b	c	T_2
(2)	/	T_2	T_1	T_3
(3)	x	b	a	T_4
(4)	+	a	T_3	T_5
(5)	+	T_5	T_4	T_6

2. Triples:

This representation doesn't make use of extra, temporary variable to represent a single operation instead when a reference to another triple's value is needed, a pointer to that triple is used.

location	op	arg1	arg2
(0)	↑	e	f
(1)	x	b	c
(2)	/	(1)	(0)
(3)	x	b	a
(4)	+	a	(2)
(5)	+	(4)	(3)

3) Indirect triples:

* This representation is an enhancement and triples.
* It uses an additional instruction array to list of pointer to trips in the desired order
* This representation makes use of pointer to the listing of all routers to computations which is made separately and stored
* Thus, instead of position pointer are used to store details.

Statement	location
35	(0)
36	(1)
37	(2)
38	(3)
39	(4)
40	(5)

Location	op	Arg1	Arg2
(0)	↑	e	f
(1)	x	b	e
(2)	/	(1)	(0)
(3)	x	b	a
(4)	+	a	(2)
(5)	+	(4)	(3)

PROGRAM:

```
import re

def generate_triples(expression):
    tokens = re.findall(r'[a-zA-Z]+|\d+|[+\-*/]', expression)
    triples = []
    temp_count = 0
    while len(tokens) > 1:
        for i in range(len(tokens)):
            if tokens[i] in '+-*/':
                op, arg1, arg2 = tokens[i], tokens[i-1], tokens[i+1]
                triples.append((op, arg1, arg2))  # No temporary variables, using pointers
                tokens[i-1] = f'({len(triples) - 1})'  # Reference to previous operation
                tokens.pop(i)
                tokens.pop(i)
                break
    return triples

def generate_quadruples(expression):
    tokens = re.findall(r'[a-zA-Z]+|\d+|[+\-*/]', expression)
    quadruples = []
    temp_count = 0
    while len(tokens) > 1:
        for i in range(len(tokens)):
            if tokens[i] in '+-*/':
                op, arg1, arg2 = tokens[i], tokens[i-1], tokens[i+1]
                quadruples.append((temp_count, op, arg1, arg2, f'T{temp_count}'))
                tokens[i-1] = f'T{temp_count}'
```

```
            tokens.pop(i)
            tokens.pop(i)
            temp_count += 1
            break
    return quadruples
def generate_indirect_triples(expression):
    triples = generate_triples(expression)
    instruction_array = list(range(len(triples)))  # Indirectly referencing triples in order
    return instruction_array, triples
expression = input("Enter an arithmetic expression: ")
triples = generate_triples(expression)
quadruples = generate_quadruples(expression)
instruction_array, indirect_triples = generate_indirect_triples(expression)
print("Triples:")
print("Operator\tArg1\tArg2")
for op, arg1, arg2 in triples:
    print(f"{op}\t{arg1}\t{arg2}")
print("\nQuadruples:")
print("Location\tOperator\tArg1\tArg2\tResult")
for location, op, arg1, arg2, result in quadruples:
    print(f"{location}\t{op}\t{arg1}\t{arg2}\t{result}")
print("\nIndirect Triples:")
print("Instruction Order:\tTriples")
for i, (op, arg1, arg2) in zip(instruction_array, indirect_triples):
    print(f"{i}\t{op}\t{arg1}\t{arg2}")
```

OUTPUT:

```
Enter an arithmetic expression:  a+b*c/e*f+b*c
Triples:
Operator        Arg1    Arg2
+       a       b
*       (0)     c
/       (1)     e
*       (2)     f
+       (3)     b
*       (4)     c

Quadruples:
Location        Operator        Arg1    Arg2    Result
0       +       a       b       T0
1       *       T0      c       T1
2       /       T1      e       T2
3       *       T2      f       T3
4       +       T3      b       T4
5       *       T4      c       T5

Indirect Triples:
Instruction Order:      Triples
0       +       a       b
1       *       (0)     c
2       /       (1)     e
3       *       (2)     f
4       +       (3)     b
5       *       (4)     c
```

Result:

Thus the program to implement a code that converts given expression into triples, quadruples, and indirect triples is executed successfully and output is verified.

5/4/25 Viva: 10/10

Experiment: 11 Date: 2/4/25	Implementation of Optimization Techniques

Aim:

To write a program to implement a code optimizer to perform possible optimization like dead code estimation, common sub expression elimination etc.

Theory:

Reason to optimize the code

* To enhance execution and efficiency of source code
* It is mandatory to delivery efficient target code by lowering the number of instructions in program

when to optimize

an important step that is usually performed at last stage of development.

Types of optimization:

Two types

1) Machine independent
2) Machine Dependent

Machine Independent optimization

It positively affects the efficiency of intermediate code by transforming a part of code that does not employ hardware parts. It usually optimizes code by eliminating un needed code.

Machine Dependent optimization:

After the target code has been constructed and transformed according to the target machine architecture, machine, architecture, machine dependent. optimization is performed

Machine dependent optimizes work hard to maximizes the period of the memory hierarchy.

loop optimization:

* Invariant code
* Induction analysis
* Strength reduction.

Algorithm:

* Input the code / expression
* Identify Redundant computations
* Apply constant folding & propogation
* Perform strength Reduction Algebentic Simplification
* Reorder instruction for efficiency
* Generate optimize code

PROGRAM:

```
def get_user_input():
  assignments = []
  num_values = int(input("Enter number of assignments: "))
  for _ in range(num_values):
    lhs = input("Enter left-hand side variable: ").strip()
    rhs = input(f"Enter right-hand side expression for {lhs}: ").strip()
    assignments.append((lhs, rhs))
  return assignments
def eliminate_dead_code(assignments):
  used_vars = set()
  # Collect all variables that are used in RHS expressions
  for _, rhs in assignments:
    used_vars.update(rhs.replace('+', ' ').replace('-', ' ').replace('*', ' ').replace('/', ' ').split())
  # Retain assignments where LHS is in used_vars or is 'r' (the final result variable)
  optimized_assignments = [(lhs, rhs) for lhs, rhs in assignments if lhs in used_vars or lhs ==
'r']
  return optimized_assignments
def eliminate_common_subexpressions(assignments):
  expr_map = {}
  optimized_assignments = []
  for lhs, rhs in assignments:
    # If the RHS expression has been seen before, map the LHS to the existing variable
    if rhs in expr_map:
      optimized_assignments.append((lhs, expr_map[rhs]))
    else:
```

```
        expr_map[rhs] = lhs
        optimized_assignments.append((lhs, rhs))
    return optimized_assignments
def main():
    print("INPUT:")
    assignments = get_user_input()
    print("\nIntermediate Code:")
    for lhs, rhs in assignments:
        print(f"{lhs} = {rhs}")
    assignments = eliminate_dead_code(assignments)
    print("\nAfter Dead Code Elimination:")
    for lhs, rhs in assignments:
        print(f"{lhs} = {rhs}")
    assignments = eliminate_common_subexpressions(assignments)
    print("\nAfter Common Subexpression Elimination:")
    for lhs, rhs in assignments:
        print(f"{lhs} = {rhs}")
    print("\nOptimized Code:")
    for lhs, rhs in assignments:
        print(f"{lhs} = {rhs}")
if __name__ == "__main__":
    main()
```

OUTPUT:

```
INPUT:
Enter number of assignments: 5
Enter left-hand side variable: a
Enter right-hand side expression for a: 9
Enter left-hand side variable: b
Enter right-hand side expression for b: c+d
Enter left-hand side variable: e
Enter right-hand side expression for e: c+d
Enter left-hand side variable: f
Enter right-hand side expression for f: b+e
Enter left-hand side variable: r
Enter right-hand side expression for r: f

Intermediate Code:
a = 9
b = c+d
e = c+d
f = b+e
r = f

After Dead Code Elimination:
b = c+d
e = c+d
f = b+e
r = f

After Common Subexpression Elimination:
b = c+d
e = b
f = b+e
r = f

Optimized Code:
b = c+d
e = b
f = b+e
r = f
```

Result:

Hence the program to implement optimization techniques is executed successfully and output is verified.

5/4/25
Viva: 10/10.