SYNTAX CHECKER

A PROJECT REPORT

***Submitted by***

BL.EN.U4AIE20041 Nandhitha Ravishankar

BL.EN.U4AIE20047 P Sadhana

***for the course***

**19AIE301- FORMAL LANGUAGE AND AUTOMATA**

***Guided and Evaluated by***

***Dr. Supriya M.***

***Vice Chairperson,***

***Dept. of CSE***



AMRITA SCHOOL OF ENGINEERING, BANGALORE

AMRITA VISHWA VIDYAPEETHAM

**BANGALORE 560 035**

**TABLE OF CONTENTS**

|  |  |  |
| --- | --- | --- |
| **S.NO** | **CHAPTER** | **PAGE NO** |
| 1. | ABSTRACT | 3 |
| 2. | INTRODUCTION | 4 |
| 3. | THEORY | 4 |
| 4. | SYSTEM MODEL | 5 |
| 5. | TOOLS/SOFTWARE USED | 6 |
| 6 | IMPLEMENTATION | 7 |
| 7 | CODE | 10 |
| 8. | CODE OUTPUT | 15 |
| 9.. | CONCLUSION | 16 |
| 10. | FUTURE WORK | 16 |
| 11. | REFERNECES | 16 |

**ABSTRACT**

Formal language and Automata theory presents the theoretical aspects of computer science, and helps define infinite languages in finite ways; construct algorithms for related problems and decide whether a string is in the particular language or not. These are of practical importance in construction of compilers and designing of programming languages.

Syntax checker that has been developed for this project, is a basic compiler designed to check certain syntaxes in a java code. It makes one’s life easier by helping them debug their syntaxes if they aren’t sure of the right syntax.

In our project, we plan to make it easier for the user to check their syntaxes by asking them to enter a syntax and displaying an appropriate message. For loop checking, while loop checking and if statement checking has been done using JFLAP.

The same has been coded in java using data structures such as arrays and linked lists. We have used the java.util.\* package to include all the necessary packages required for this program.

**INTRODUCTION**

These machines are called finite because there are a limited number of possible states which can be reached. Similar to a Deterministic Finite Automata (DFA), a Non-Deterministic Finite Automata (**NFA) is a state machine consisting of states and transitions that can either accept or reject a finite string**. And like a DFA, we must use circles to represent states, and directed arrows to represent transitions. Essentially, NFAs have less restrictions than DFAs, and can therefore make complicated automata easier to understand and depict in a diagram. A DFA can only have one transition for each symbol going outwards from the state. But, an NFA can have multiple transitions for a symbol from the same state**. NFA is not required to have a transition for each symbol.**

An automata is designed in this project to perform the functions of a syntax checker for a while loop, for loop and an if statement. As of now, only logical operators AND (&&) and OR (||) have been implemented. So, the automata when given syntax test cases, checks if it is accepted or rejected. The syntax of only Java is depicted in this checker. The same has been implemented on eclipse. A simple syntax checker has been coded in Java using data structures to obtain the same result and also help to depict the project on a graphical user interface (GUI)which can be done as a part of the project.

**THEORY**

Non-deterministic finite automata (NFA) is defined as a 5 tuple,

M = (Q, Σ, δ, q0, F)

Where,

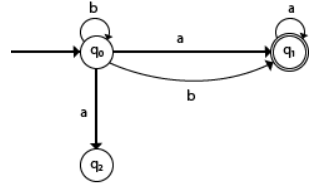
* Q: Finite set of states
* Σ: Finite set of the input symbol
* q0: Initial state
* F: Final state
* δ: Transition function: Q X Σ -> 2Q

Graphical Representation of an NFA

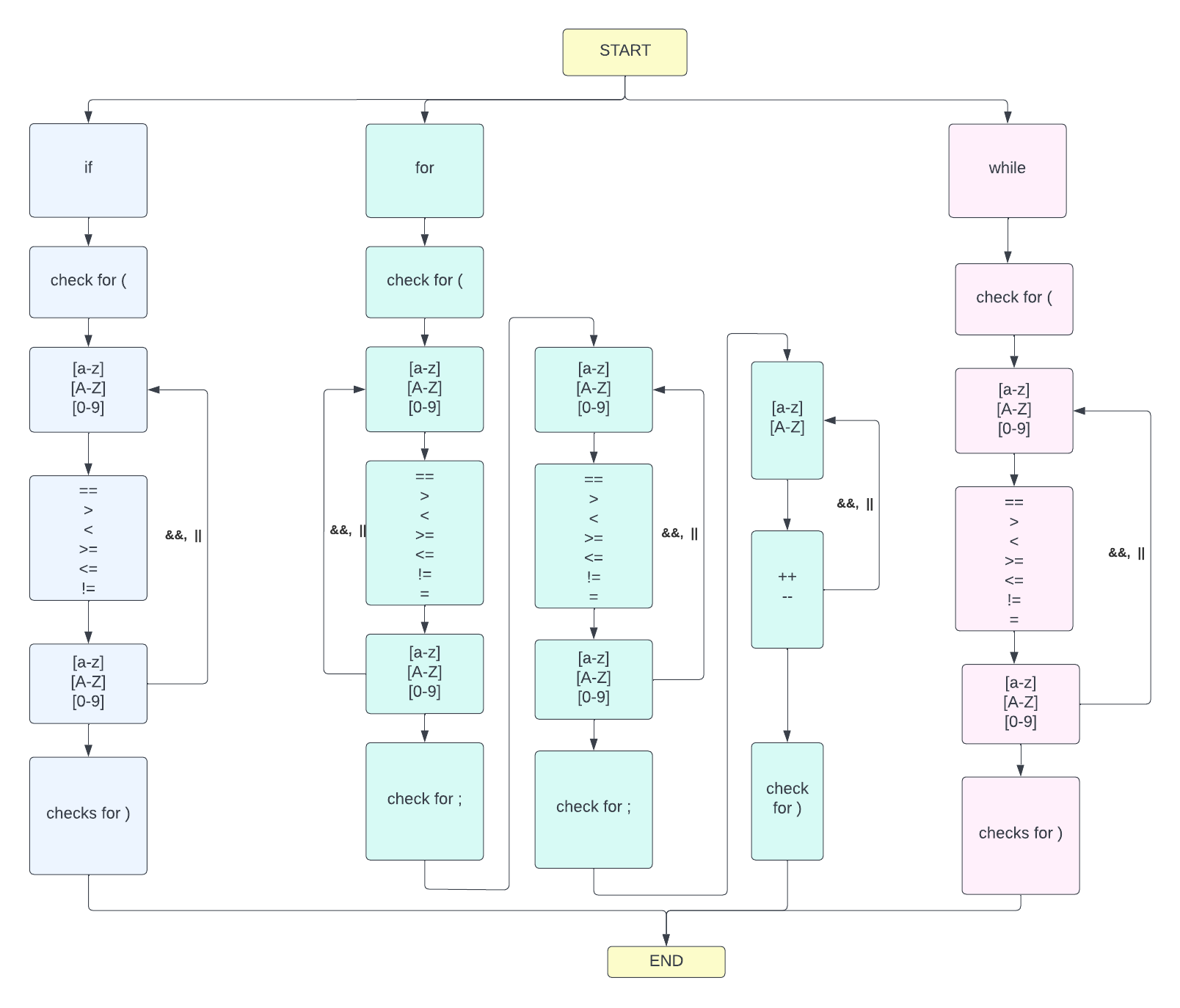
An NFA can be represented by digraphs called state diagrams

The following factors are considered while graphically representing NFA −

* The state is represented by vertices.
* The arc labelled with an input character shows the transitions.
* The initial state is marked with an arrow.
* The final state is denoted by the double circle.

****

**SYSTEM MODEL**

****

**TOOLS USED**

**JFLAP-7.1**

**JFLAP**is a package of graphical tools which can be used as an aid in learning the basic concepts of Formal Languages and Automata Theory. It is software that allows users to experiment with formal language issues such nondeterministic finite automata, nondeterministic pushdown automata, multi-tape Turing machines, various grammars, parsing, and L-systems. JFLAP enables one to experiment with constructing proofs from one form to another, such as converting an NFA to a DFA to a minimal state DFA to a regular expression or regular grammar. This is in addition to creating and testing examples for these.

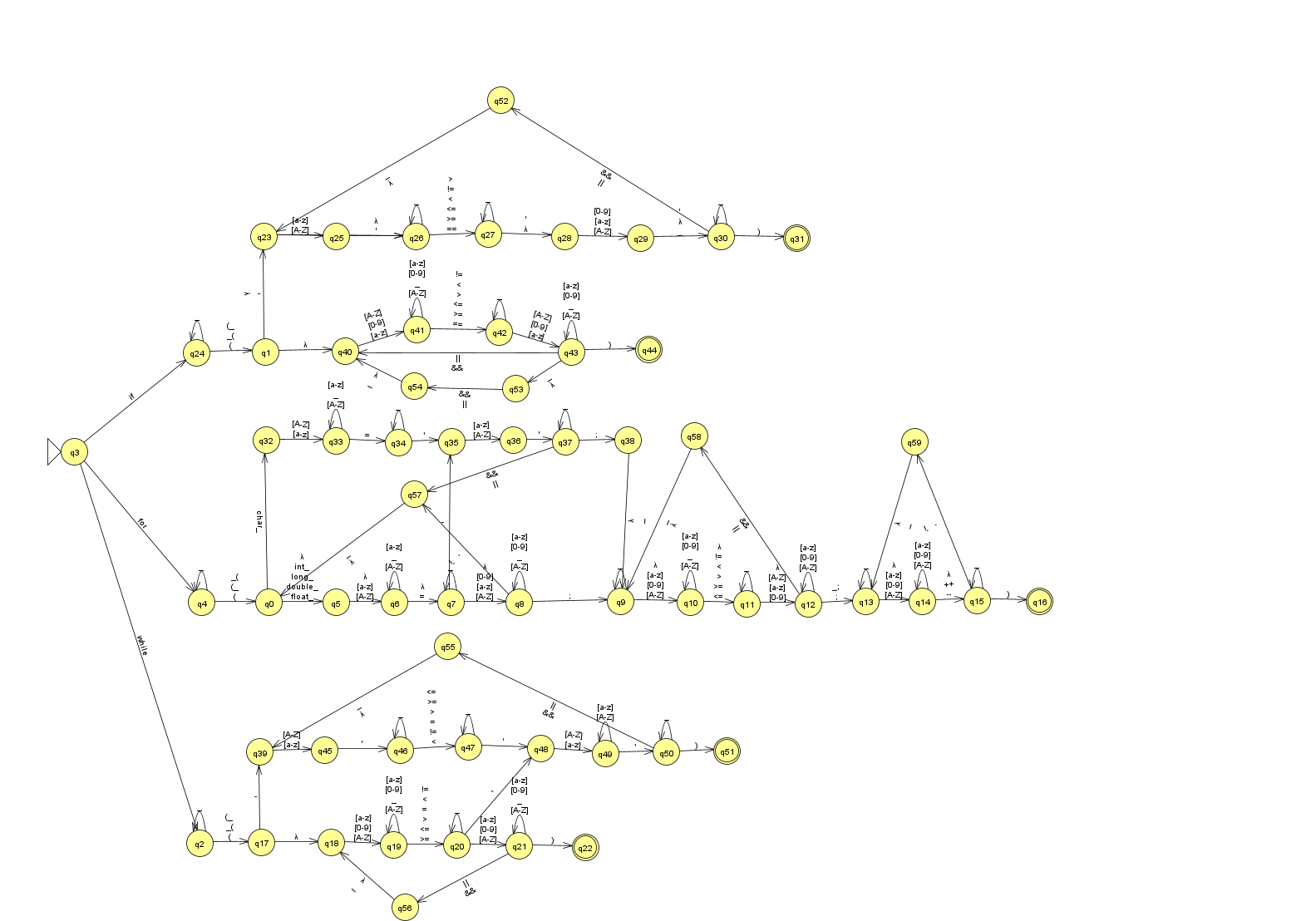
We have depicted a basic syntax checker using JFLAP. An NFA is designed using all the basic requirements such as the start state, final state, loops, transition states , lambda transitions.

**Eclipse IDE**

Eclipse is an [integrated development environment](https://en.wikipedia.org/wiki/Integrated_development_environment) (IDE) used in [computer programming](https://en.wikipedia.org/wiki/Computer_programming). It contains a base [workspace](https://en.wikipedia.org/wiki/Workspace) and an extensible [plug-in](https://en.wikipedia.org/wiki/Plug-in_(computing)) system for customizing the environment. It is the second-most-popular IDE for [Java](https://en.wikipedia.org/wiki/Java_(programming_language)) development, and, until 2016, was the most popular. Eclipse is written mostly in Java and its primary use is for developing Java applications.

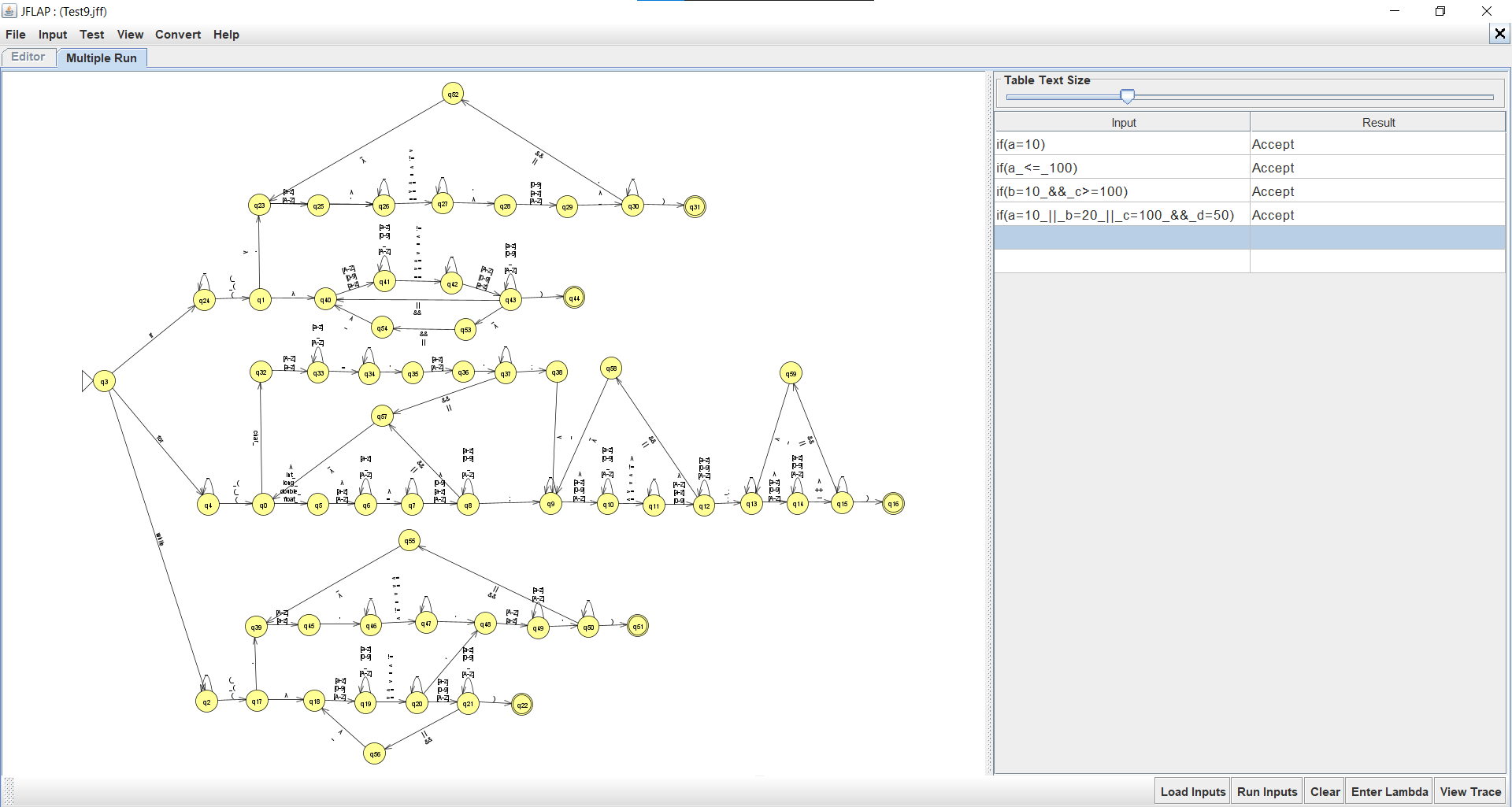
Data Structures such as arrays and stacks are used to design a syntax checker on eclipse ide for java. Stacks have been used in the WhileLoopChecker to input the size of the stack which is basically the size of the inputted expression. The same is used in ForLoopChecker, as well as IfStatementChecker.

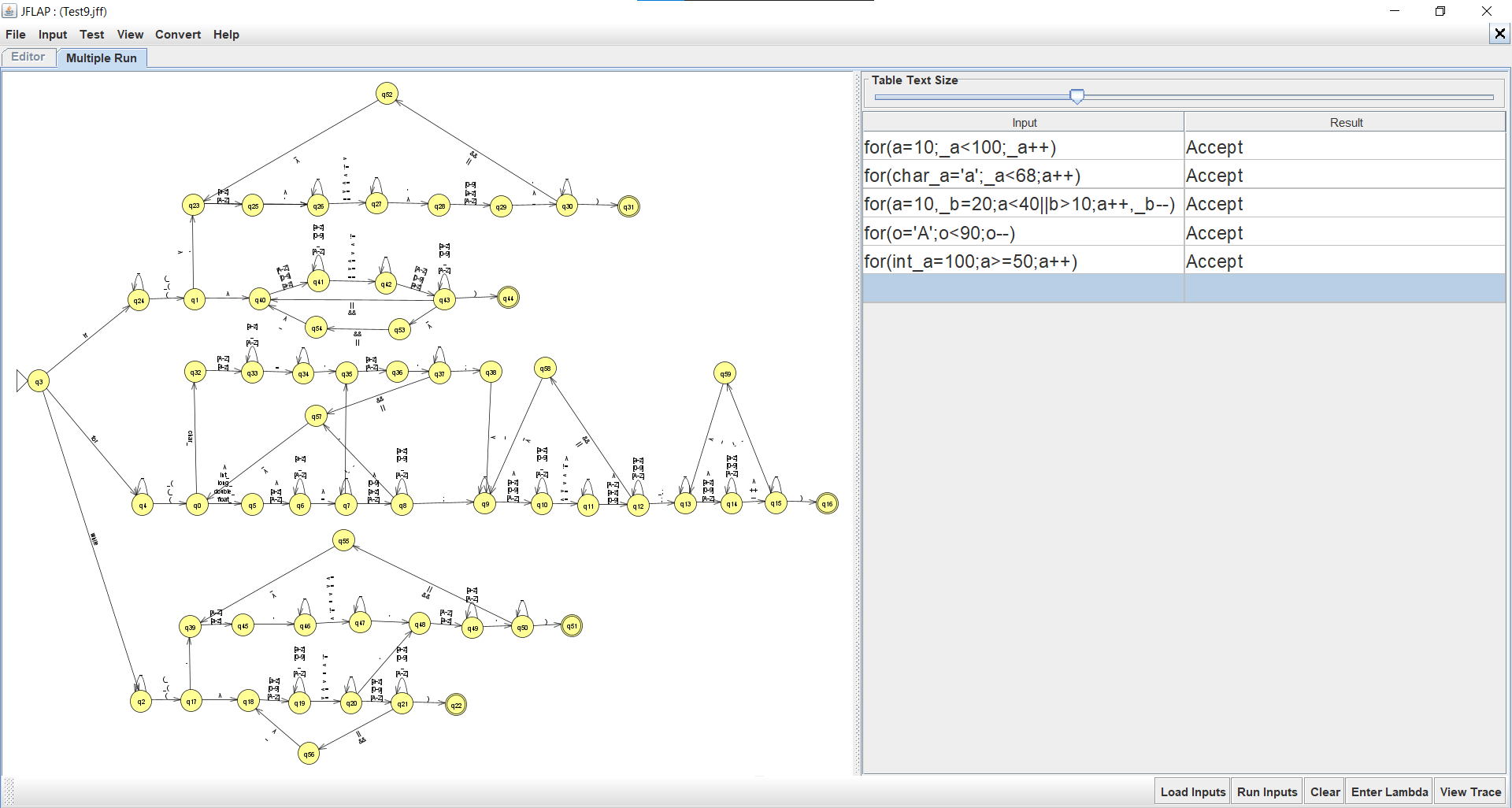
**IMPLEMENTATION**

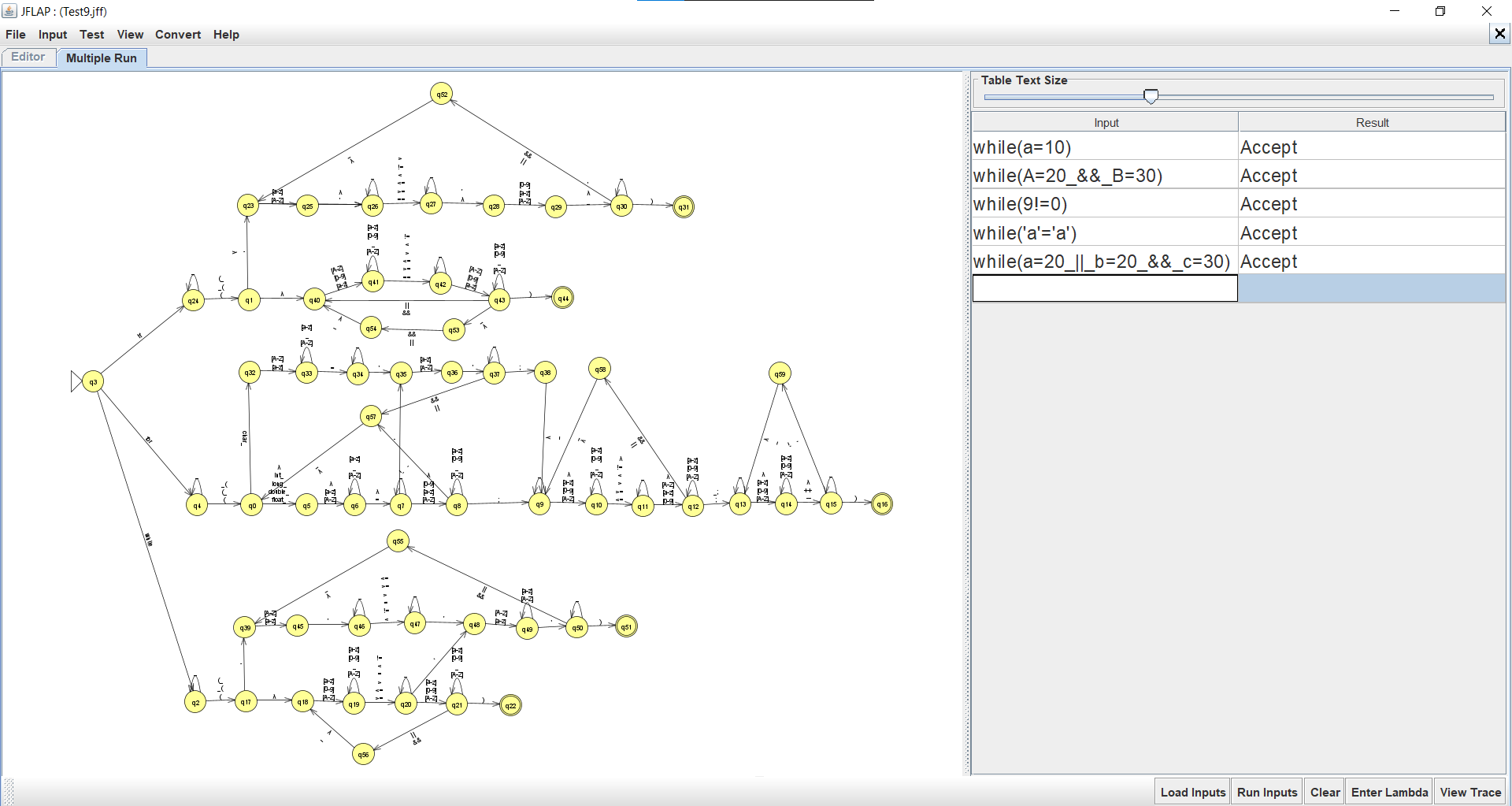
****

61 states have been used to implement the syntax checker. Checking syntaxes of for loop, while loop and if statements have been done. Every time the user enters a syntax, the NFA runs to check the syntaxes. Once done, it displays whether the given input is of the right syntax or not.

***Test cases:***







**CODE**

package Project;

import java.util.\*;

public class SyntaxChecker\_FLA

{

public static void main(String args[])

{

String expression;

String expre;

String[] a = new String[100];

Scanner sc = new Scanner(System.***in***);

int choice;

do

{

System.***out***.println("Enter your choice: ");

System.***out***.println("1. while loop checking");

System.***out***.println("2. for loop checking");

System.***out***.println("3. if statement checking");

System.***out***.println("4. exit");

choice = sc.nextInt();

switch(choice)

{

case 1: System.***out***.println("Enter your while expression: ");

expre = sc.next();

SyntaxChecker.*whileLoopChecker*(expre);

System.***out***.println();

break;

case 2: System.***out***.println("Enter your for expression: ");

expre = sc.next();

SyntaxChecker.*forLoopChecker*(expre);

System.***out***.println();

break;

case 3: System.***out***.println("Enter your if expression: ");

expre = sc.next();

SyntaxChecker.*IfStatementChecker*(expre);

System.***out***.println();

break;

default: System.***out***.println("Invalid Input!");

}

}

while(choice!=4);

sc.close();

}

public static boolean whileLoopChecker(String input)

{

int flag=0;

int stackSize = input.length();

if( (input.length()<6) ||(input.charAt(0) != 'w') || (input.charAt(1) != 'h') ||

(input.charAt(2) != 'i') || (input.charAt(3) != 'l') || (input.charAt(4) != 'e'))

{

System.***out***.println("\nError in 'while' keyword usage");

return false;

}

else

{

if((input.charAt(5)!='(') && (input.charAt(stackSize- 1 ) != ')'))

{

System.***out***.println("\nSyntax error, Brackets are missing");

flag++;

}

else if(input.charAt(5)!='(')

{

System.***out***.println("\nOpening parenthesis absent after 'while' keyword");

flag++;

}

else if(input.charAt(stackSize-1) != ')')

{

System.***out***.println("\nClosing parenthesis absent at the end of while");

flag++;

}

else if((input.charAt(7)) != '>' && (input.charAt(7))!='<' &&

((input.charAt(7))!='=' || (input.charAt(8))!='=') &&

((input.charAt(7))!='!' || (input.charAt(8))!='='))

{

System.***out***.println("\nError in the usage of symbol");

flag++;

}

if(flag == 0)

{

System.***out***.println("\nNo error!");

}

}

return true;

}

public static boolean forLoopChecker(String input)

{

int bracket1 = 0, bracket2 = 0, flag = 0;

int stackSize = input.length();

if( (input.length()<3) || (input.charAt(0) != 'f') ||

(input.charAt(1) != 'o') || (input.charAt(2) != 'r'))

{

System.***out***.println("\nError in 'for' keyword usage");

return false;

}

else

{

for(int i=0;i<stackSize;i++)

{

char ch = input.charAt(i);

if(ch == '(')

{

bracket1 ++;

}

else if(ch == ')')

{

bracket2 ++;

}

else if(ch == ';')

{

}

else if(ch == ' ')

{

continue;

}

else

{

continue;

}

}

if((input.charAt(3)!='(') && (input.charAt(stackSize- 1 ) != ')'))

{

System.***out***.println("\nSyntax error, Brackets are missing");

flag++;

}

else if(input.charAt(3)!='(')

{

System.***out***.println("\nOpening parenthesis absent after 'for' keyword");

flag++;

}

else if(input.charAt(stackSize-1 ) != ')')

{

System.***out***.println("\nClosing parenthesis absent at the end of for");

flag++;

}

else if((input.charAt(stackSize-5))!=';' || (input.charAt(stackSize-9))!=';' && (input.charAt(stackSize-10))!=';')

{

System.***out***.println("\nSemicolon Error");

flag++;

}

else if(bracket1 != 1 || bracket2 != 1 || bracket1 != bracket2)

{

System.***out***.println("\nParentheses Count Error");

flag++;

}

if(flag == 0)

{

System.***out***.println("\nNo error!");

}

return true;

}

}

public static boolean IfStatementChecker(String input)

{

int flag = 0;

int stackSize = input.length();

if( (input.length()<3) ||(input.charAt(0) != 'i') || (input.charAt(1) != 'f') )

{

System.***out***.println("\nError in 'if' keyword usage");

return false;

}

else

{

if((input.charAt(2)!='(') && (input.charAt(stackSize- 1 ) != ')'))

{

System.***out***.println("\nSyntax error, Brackets are missing");

flag++;

}

else if(input.charAt(stackSize- 1 ) != ')')

{

System.***out***.println("\nClosing parenthesis absent at the end of statement");

flag++;

}

else if(input.charAt(2)!='(')

{

System.***out***.println("\nOpening parenthesis absent after 'if' keyword");

flag++;

}

else if(((input.charAt(stackSize-4) != '>') || (input.charAt(stackSize-3) != '>'))

&&

((input.charAt(stackSize-4) != '<') || (input.charAt(stackSize-3) != '<'))

&&

((input.charAt(stackSize-5) != '!') || (input.charAt(stackSize-4) != '!'))

&&

((input.charAt(stackSize-4) != '=') || (input.charAt(stackSize-3) != '='))

&&

((input.charAt(stackSize-5) != '=') || (input.charAt(stackSize-4) != '='))

&&

((input.charAt(stackSize-4) != '=') || (input.charAt(stackSize-3) != '=')))

{

System.***out***.println("\nError in the usage of symbol");

flag++;

}

if(flag == 0)

{

System.***out***.println("\nNo error!");

}

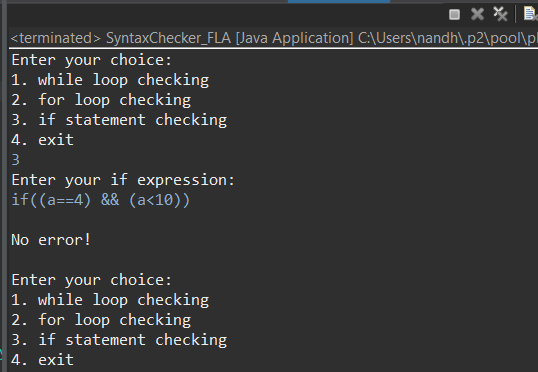
}

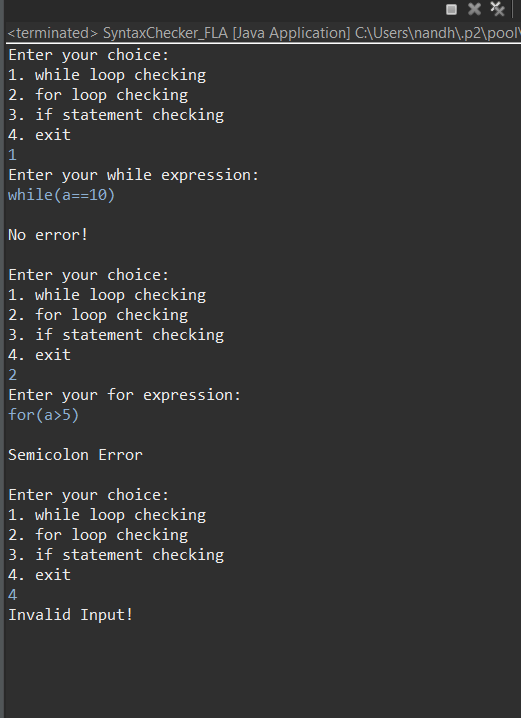
return true;

}

}

**CODE OUTPUT**

****

****

**CONCLUSION**

In this project, a syntax checker has been implemented successfully using Non – Deterministic Finite Automata (NFA). The work as also been implemented on java to depict the project on a graphical user interface (GUI) as the future work. More additions can be made to the project such checking syntax’s of do-while loop, keyword checker and so on.

**FUTURE WORK**

* Add more syntaxes such as do-while, if-else, if-else-if ladder
* Add more operations such as relational, bitwise etc

**REFERENCE**

1. <https://www.cs.ucdavis.edu/~rogaway/classes/120/spring13/eric-dfa.pdf>
2. <https://www.jflap.org/>
3. <https://en.wikipedia.org/wiki/Eclipse_(software)>