CSE6344 Theory of Computation

March Trimester 2024

Assignment (20%)

Lecturer Name: Dr. Nbhan D. Salih

Tutorial Section: TT2L

Group Number: 2

No.	Student Name	Student ID	Participation
1.	Sabrina Amalyn Binti Aminur Rizal	1201102251	25%
2.	Rahmani Erfan	1201102372	25%
3.	Ehsan Bin Shamsul Akmal	1211305254	25%
4.	Omar Elakkad	1211300682	25%
	Total		100%

Submission Date: 16/6/2024

Table of Contents

1. Introduction

1.1 Regular Grammar

A regular grammar is a type of formal grammar used to generate regular languages, which are languages recognized by finite automata. Formally, a regular grammar *G* is defined as a quadruple (V, Σ, R, S):

> V: A finite set of variables (also called non-terminals).
>
> Σ: A finite set of terminal symbols.
>
> R: A finite set of production rules. Each rule has one of these forms:
>
> - A→aB or A→a (for right-linear grammars)
>
> - A→Ba or A→a (for left-linear grammars).
>
> - In these rules, A and B are non-terminal symbols from N, and a is a terminal symbol from Σ.
>
> S: The start symbol, which is a variable from V.

1.2 Finite Automata

A finite automaton, also known as a finite state machine, is a mathematical model used to describe computation. It can be defined as a sextuple ($Q, \Sigma, \delta, q0, F$), where:

> Q: A finite set of states.
>
> Σ: A finite alphabet of input symbols.
>
> δ: A transition function mapping Q×Σ to Q, indicating the next state   given a current state and input symbol.
>
> $q0$; The start state, an element of Q.
>
> F: A set of accept states, a subset of Q.

Finite automata come in two main types:

1) **Deterministic Finite Automaton (DFA):** In a DFA, for each state and input symbol, there is exactly one next state, determined by the transition function. It means the transition from one state to another is uniquely determined by the current state and the input symbol.

2) **Nondeterministic Finite Automaton (NFA):** In an NFA, for each state and input symbol, there can be multiple next states, or there might even be transitions that don't require input (epsilon transitions).

Both finite automata recognize regular languages but vary in input processing and state transitions. DFAs are straightforward with a direct state-to-state mapping, while NFAs are more flexible, potentially needing extra processing for acceptance determination.

1.3 Project Objectives

There are the four main functions to complete for this Java application:

1) Converting regular grammar into a non-deterministic finite automaton (NFA).
2) Removing ε-transitions from non-deterministic finite automata (NFA).
3) Converting NFAs into deterministic finite automata (DFA).
4) Checking strings for acceptance or rejection.

Each member will be responsible for implementing one of these functions in the following order: Omar, Sabrina, Erfan, and Ehsan.

2. Design & Algorithm

2.1 Symbols Used in Flowchart

Before viewing the design flowcharts, it is essential to understand the symbols used in the flowchart for clarity and standardization. Below is a brief explanation of each symbol and its purpose (Programiz, 2023).

Symbol	Purpose	Description
→	Flow line	Indicates the flow of logic by connecting symbols.
	Terminal(Stop/Start)	Represents the start and the end of a flowchart.
	Input/Output	Used for input and output operation.
	Processing	Used for arithmetic operations and data-manipulations.
	Decision	Used for decision making between two or more alternatives.
	On-page Connector	Used to join different flowline
	Off-page Connector	Used to connect the flowchart portion on a different page.
	Predefined Process/Function	Represents a group of statements performing one processing task.

Figure 1 Flowchart Symbols and their Purpose

2.2 RG to NFA

The function parseAndConvertToNFA() initializes data structures for non-terminals, terminals, and productions from input lines, identifying the start symbol if not already set, skips epsilon productions, and then converts the gathered information into an NFA using fromRegularGrammar(), ultimately returning the resulting NFA representation.
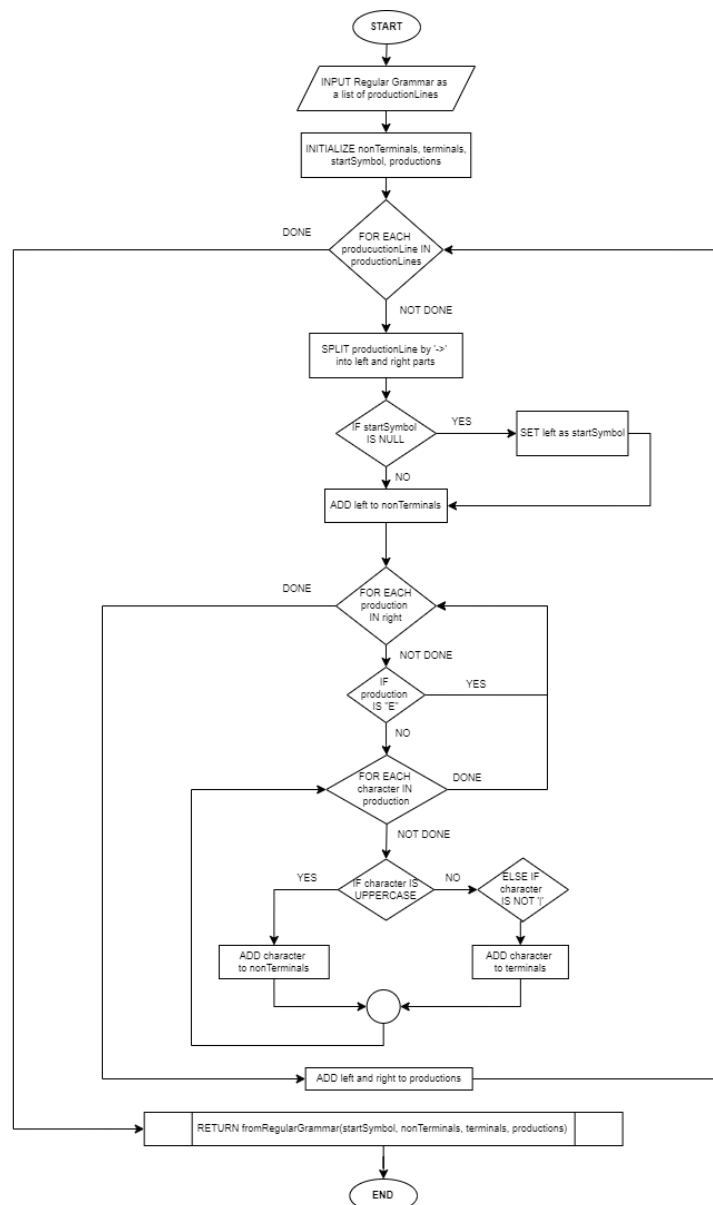

Figure 2 Flowchart of Initial Parsing before Conversion

The function fromRegularGrammar() transforms a regular grammar into a Non-deterministic Finite Automaton (NFA) by initializing an FA object, adding states based on productions, setting transitions for terminals and non-terminals, handling epsilon transitions, and finally establishing the start state before returning the NFA.
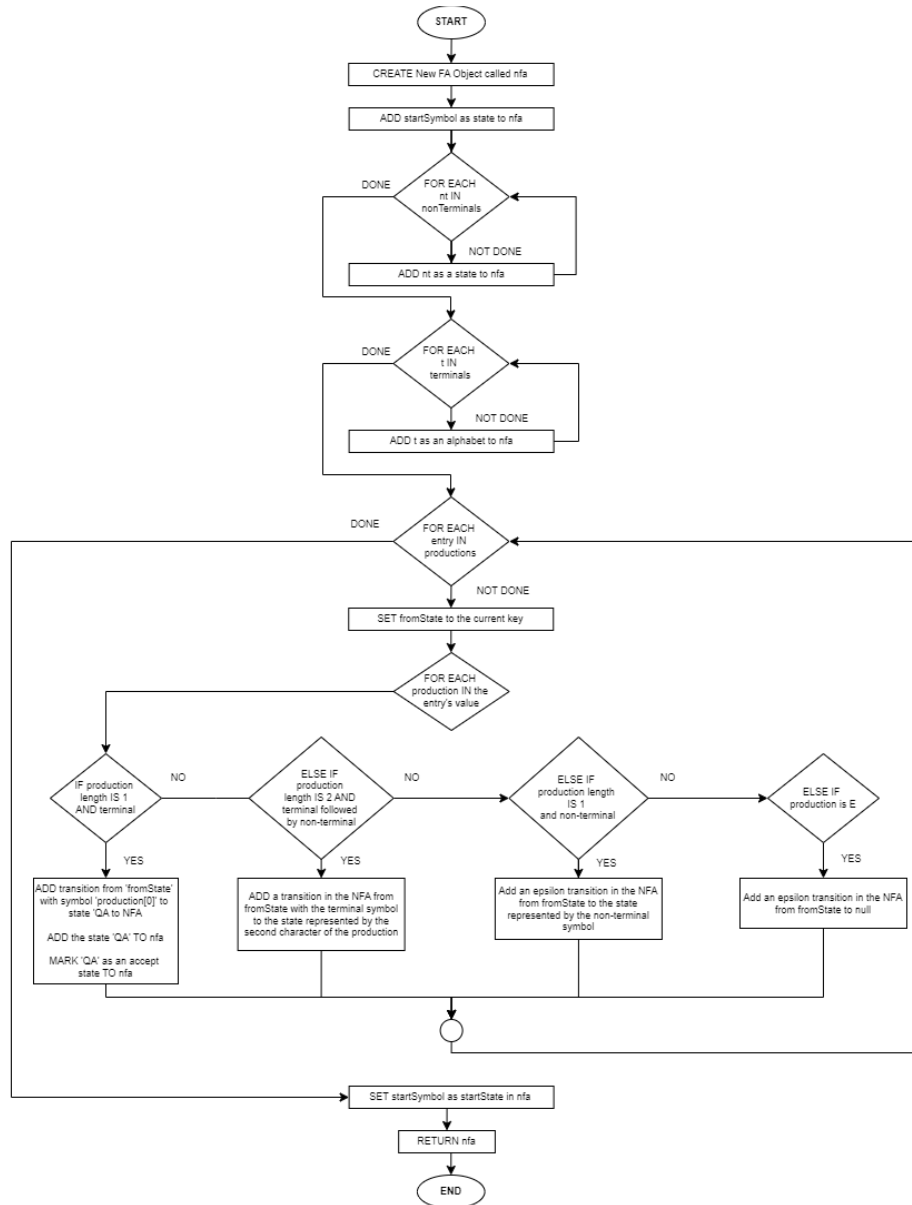
Figure 3 Flowchart of Regular Grammar to ε-NFA Conversion

7

2.3 ε-NFA to NFA

The function convertToNFAWithoutEpsilon() transforms an ε-NFA to a standard NFA by computing ε-closures for each state, adjusting transitions based on these closures, including reverse transitions for states with ε-transitions, and designating states reachable by ε-transitions to accept states, resulting in the removal of all ε-transitions from the automaton.
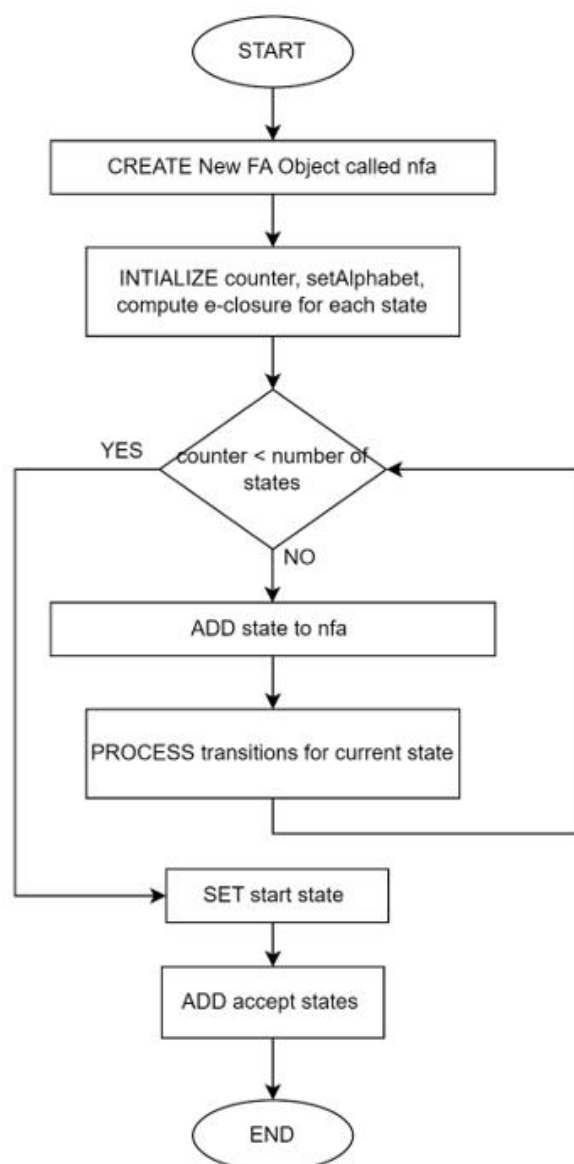

Figure 4 Flowchart of ε-NFA to NFA Conversion

2.4 NFA to DFA

The function convertToDFA() converts an NFA to a DFA by initializing a new DFA
and state mapping, computing ε-closures for states, and iteratively determining
transitions for each state based on input symbols, ensuring each state corresponds to a
set of states from the NFA until all states are processed, ultimately establishing accept
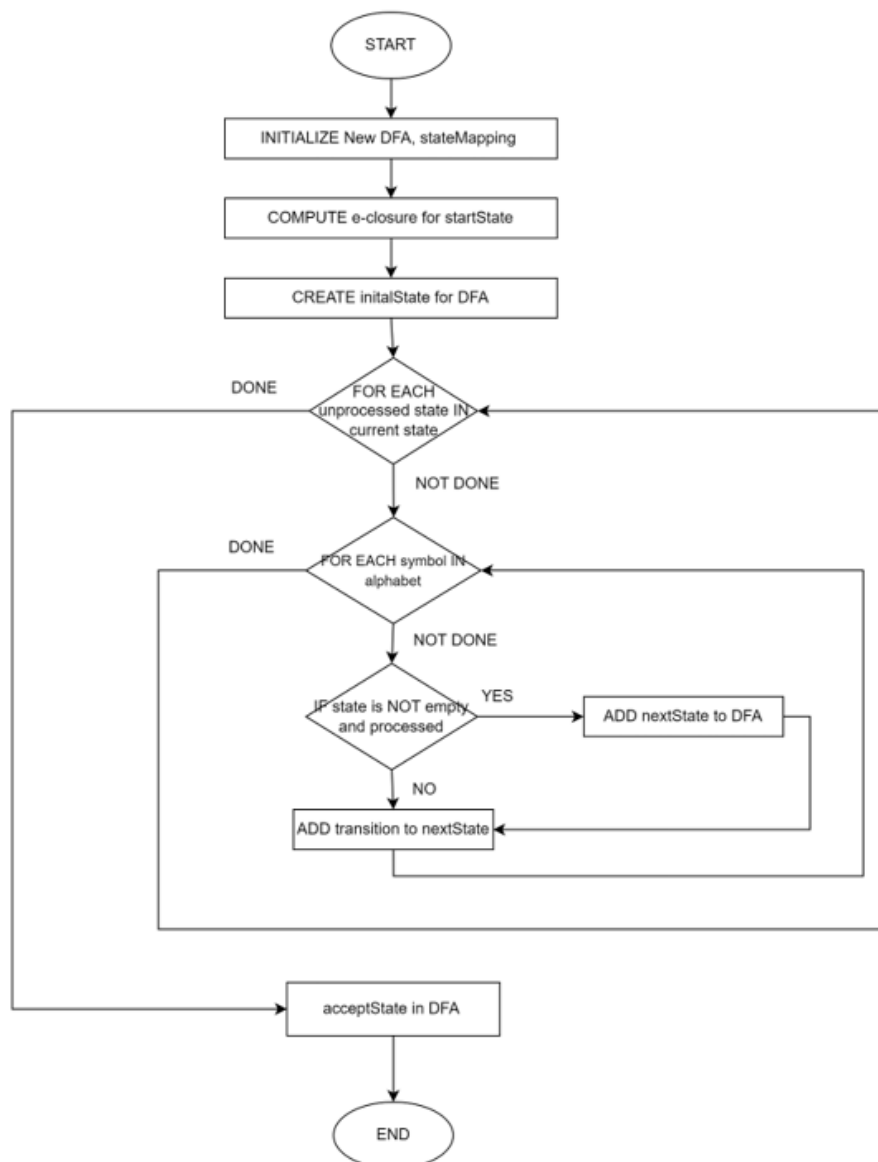states in the DFA.

Figure 5 Flowchart of NFA to DFA Conversion

2.5 String Testing

The function testInput() verifies whether a string is accepted by a DFA by initializing the current state to the DFA's initial state, iterating through each symbol in the input to update the current state based on valid transitions, and determining acceptance based on the final state's status as an accept state, ensuring systematic evaluation of string acceptance or rejection according to DFA rules.
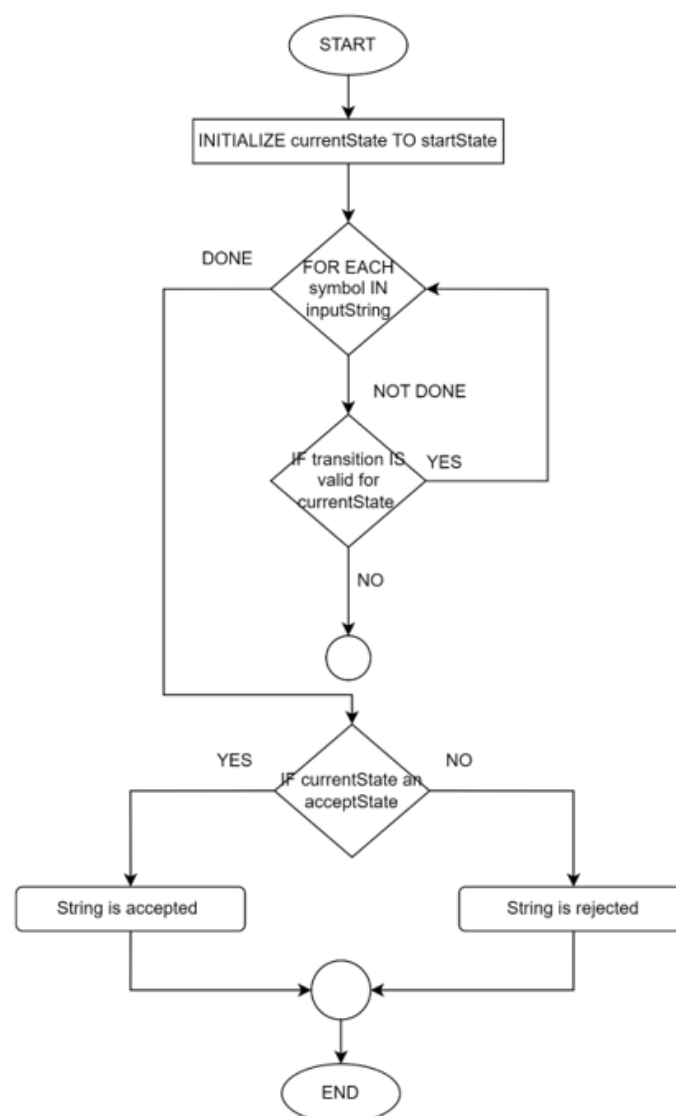

Figure 6 Flowchart of String Testing

3. Application Screenshots

At the top of the application, you will find three tabs for navigating through the different sections of this project: Home, RG to FA, and Help.

3.1 Home Page

The home page introduces the project members, displaying their names, student IDs, contribution percentages, and photos.
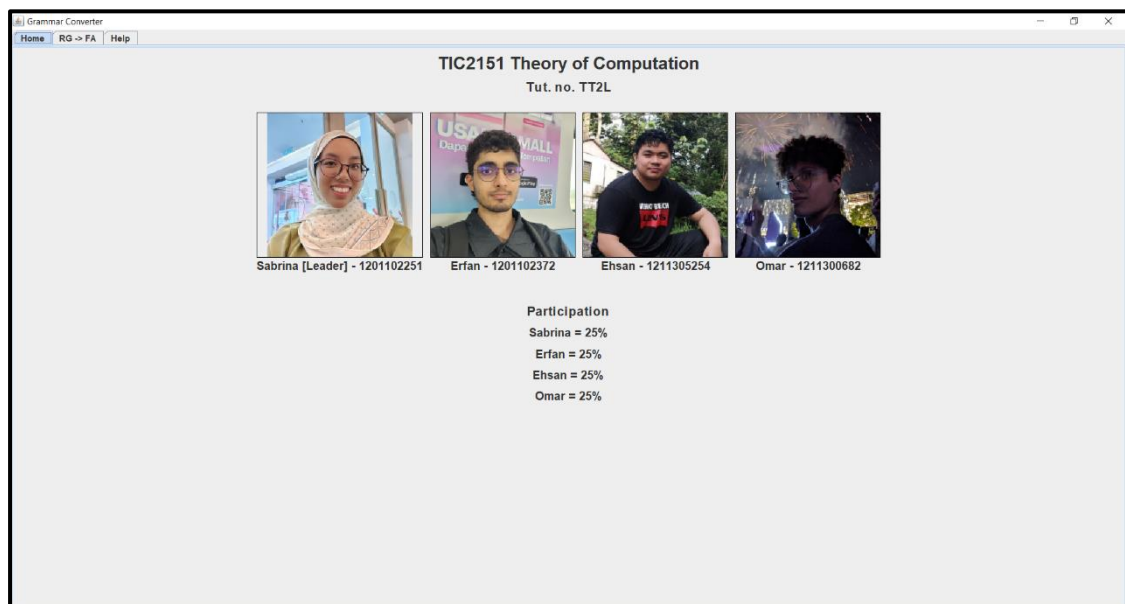

Figure 7 Home Page

3.2 RG to FA Page

In the RG to FA tab, users are presented with four main function buttons: NFA, NFA to NFA without epsilon, NFA to DFA, and String Test. They can import a regular grammar text file using the import button and clear any existing data using the clear button.
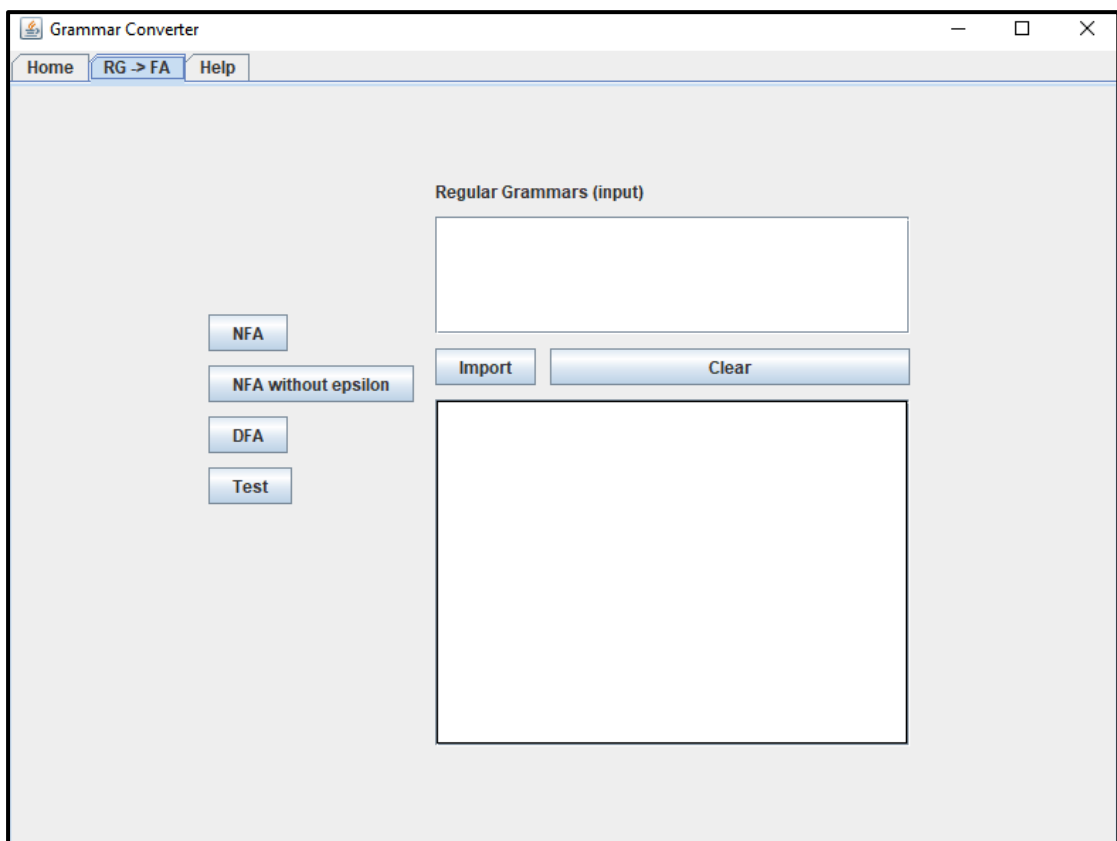

Figure 8 RG to FA Page

3.3 Help Page

The Help page features a user manual that guides users on how to operate the program. It provides step-by-step instructions for inputting data and obtaining results. Additionally, this tab includes examples of the output for each function within the system.
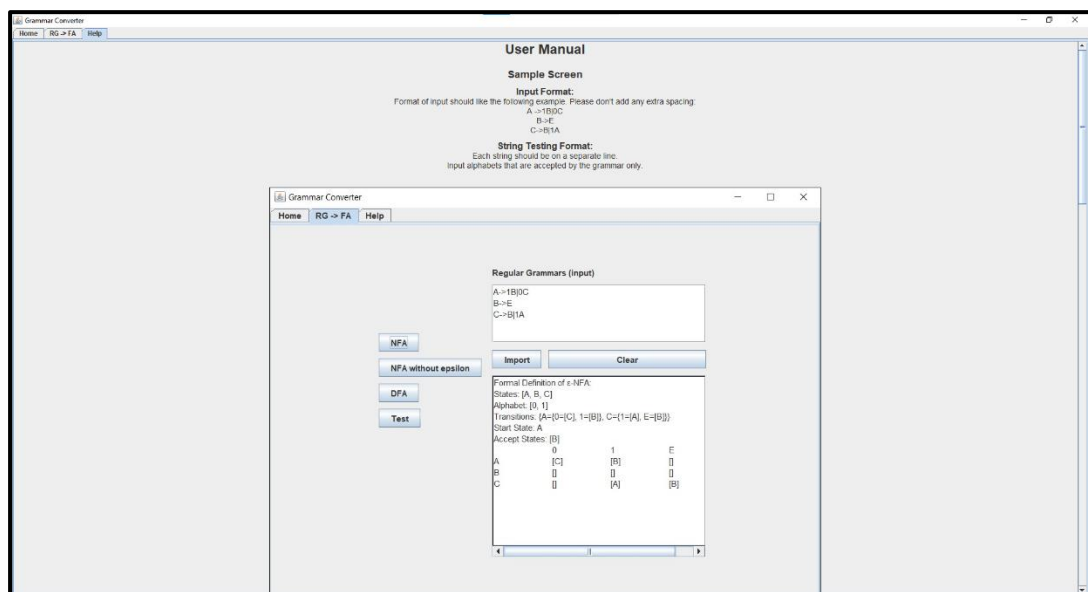

Figure 9 Help Page

4. User Manual

4.1 Importing Text File

To import a regular grammar from a text file, the grammar should be saved in a .txt file on your device.
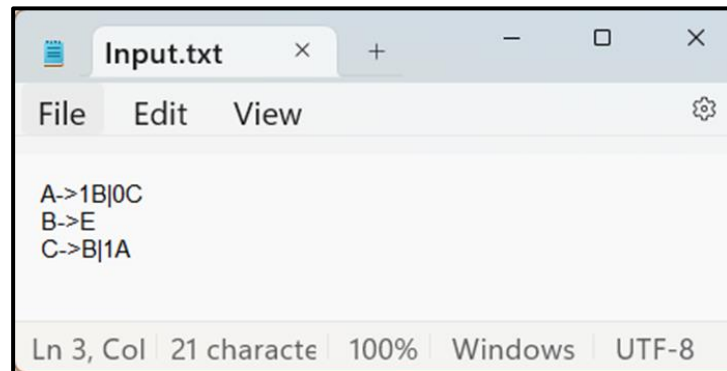

Figure 10 Regular Grammar Text File

Upon clicking the "Import" button, a window would pop up to allow the user to select the text file that is to be imported.
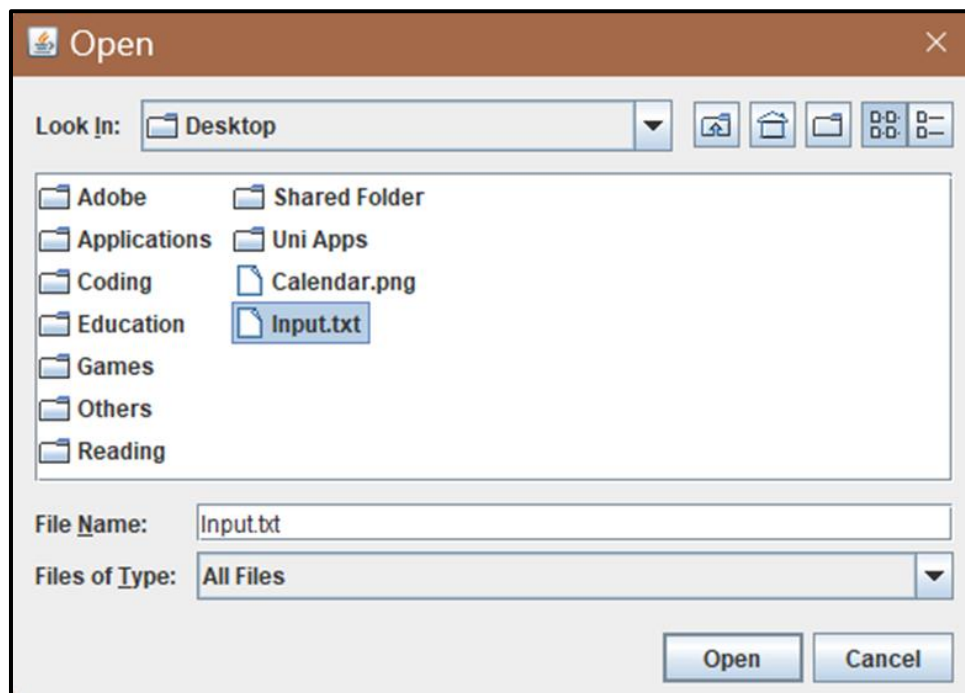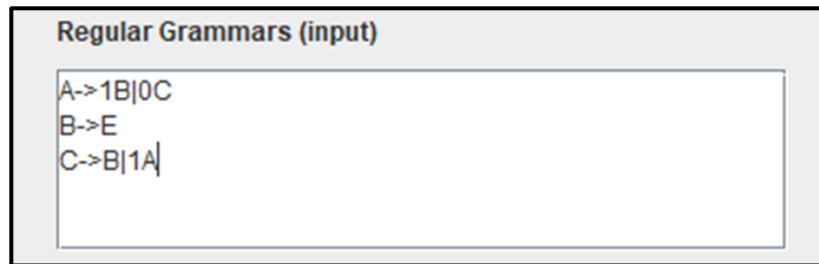
Figure 11 Selecting Text File

4.2 Input Format
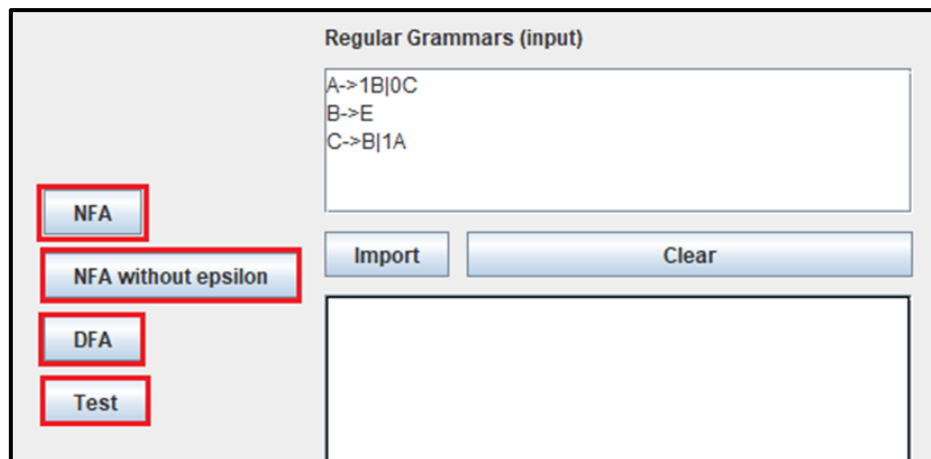
Figure 12 Format of Regular Grammar Input

When typing in or importing the regular grammar, please adhere to the following

guidelines for proper functionality:

1) Use right-linear grammar format.

2) Ensure there are no extra spaces in the grammar input.

3) Each production rule must be entered on a separate line.

4) Use the capitalized letter "E" to denote epsilon (ε).

5) If a state has multiple transitions, input them together on a single line in the

format "A->0B|1C".

4.3 Output Generation

Following the input, the buttons on the left-hand side should be clicked to generate

the output.

Figure 13 Output Generation Buttons

4.3.1 RG to NFA Output

Regular Grammars (input)

```
A->1B|0C
B->E
C->B|1A
```

Import   Clear

```
Formal Definition of ε-NFA:
States: [A, B, C]
Alphabet: [0, 1]
Transitions: {A={0=[C], 1=[B]}, C={1=[A], E=[B]}}
Start State: A
Accept States: [B]
```

	0	1	E
A	[C]	[B]	[]
B	[]	[]	[]
C	[]	[A]	[B]

Figure 14 NFA Output

4.3.2 NFA to NFA Without Epsilon Output

Regular Grammars (input)

```
A->1B|0C
B->E
C->B|1A
```

Import   Clear

```
Formal Definition of NFA (without ε-transitions):
States: [A, B, C]
Alphabet: [0, 1]
Transitions: {A={0=[B, C], 1=[B]}, C={1=[A]}}
Start State: A
Accept States: [B, C]
              0              1
A             [B, C]         [B]
B             []             []
C             []             [A]
```

Figure 15 NFA without Epsilon Output

4.3.3 NFA to DFA Output

Regular Grammars (input)

```
A->1B|0C
B->E
C->B|1A
```

Import   Clear

```
Formal Definition of DFA:
States: [A, BC, B]
Alphabet: [0, 1]
Transitions: {BC={1=[A]}, A={0=[BC], 1=[B]}}
Start State: A
Accept States: [BC, B]
                    0              1
A               [BC]           [B]
BC              []             [A]
B               []             []
```

Figure 16 DFA Output

4.3.4 Testing String

Upon clicking on the "Test" button, a window would pop up to insert the strings that are to be tested. Each line should consist of one string only.
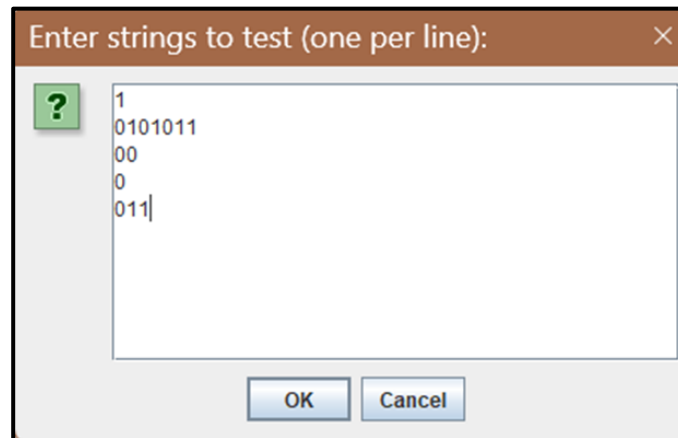

Figure 17 Testing String Input

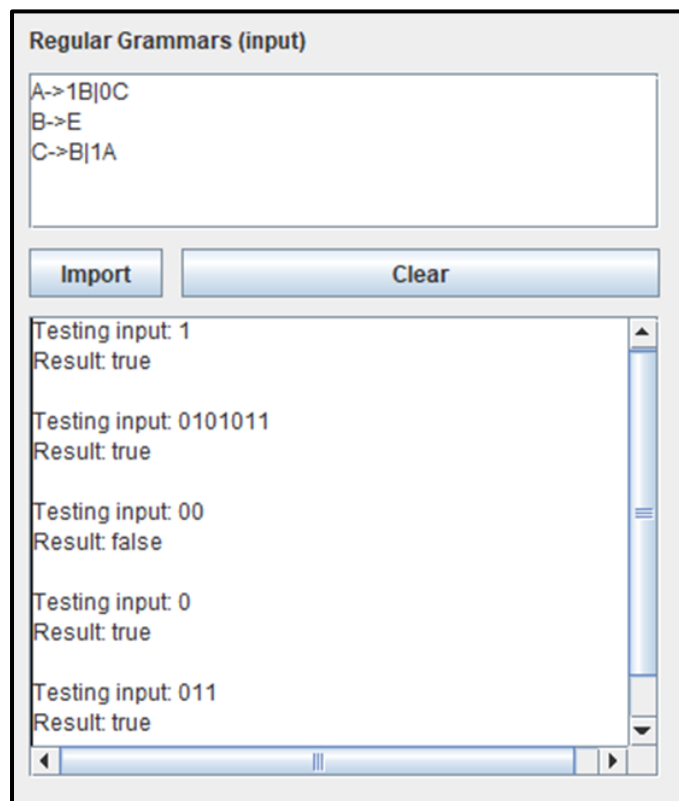After selecting "OK" in the pop-up window, the results will be displayed in the output text area box.

Figure 18 Testing String Output

5. Important Codes

1) **parseAndConvertToNFA()**

The parseAndConvertToNFA() method begins by accepting an array of production rules as input. It initializes sets to store non-terminals and terminals, and a map to hold production rules. The start symbol is identified from the left-hand side (LHS) of the first production rule. Iterating over each production rule, it splits them into LHS and RHS (Right-Hand Side) components, where RHS terms are split into individual symbols. Terms identified as terminal symbols are added to the terminals set. Finally, the method invokes fromRegularGrammar() to convert the parsed regular grammar into a Non-deterministic Finite Automaton (NFA). This approach ensures systematic conversion from a regular grammar representation to an NFA, leveraging structured parsing and symbol identification.

2) **fromRegularGrammar()**

The fromRegularGrammar() method begins by initializing a new NFA object to represent the regular grammar. It proceeds to add states to the NFA corresponding to each non-terminal symbol found in the grammar. Terminal symbols are also added directly to the NFA as states. Iterating over each production rule, it creates transitions within the NFA based on the RHS of each production: if the RHS consists of a single terminal symbol, a transition is made to an accept state; if followed by a non-terminal, a transition is established to the corresponding non-terminal state. Finally, the method sets the start state of the NFA to match the start symbol of the regular grammar and returns the fully constructed NFA, thereby converting the parsed regular grammar into an NFA representation suitable for further processing or analysis.

3) **convertToNFAWithoutEpsilon ()**

The convertToNFAWithoutEpsilon() method aims to eliminate ε-transitions within a given ε-NFA by computing ε-closures for each state and updating transitions accordingly. It initializes a new NFA instance and computes ε-closures for all states in the ε-NFA. The method iterates through each state and symbol, adjusting transitions based on these ε-closures: if a state has an ε-transition to another state, it incorporates all transitions of the latter state. If a state can be reached via ε-transitions and is marked as an accept state, it retains this status. Finally, the method establishes the initial state of the newly created NFA and returns it, thereby converting the ε-NFA into an equivalent NFA without ε-transitions, ready for further use or analysis.

4) **convertToDFA()**

The convertToDFA() method employs the subset construction algorithm to convert an NFA to a DFA, where each state in the DFA represents a set of NFA states. Beginning with the creation of a new DFA object, it maps sets of NFA states to corresponding DFA states. The initial state of the DFA is determined by computing the ε-closure of the start state of the NFA. The method iterates over each state of the DFA, computing transitions for every input symbol: the target state for each transition is derived by computing the ε-closure of the union of states that can be reached via the input symbol from each state in the set. If any state within the set of NFA states is an accept state, the corresponding DFA state inherits this designation. Finally, the method establishes the initial state of the DFA and returns the fully constructed DFA, thus converting the NFA into an equivalent DFA suitable for further analysis or processing.

5) testInput()

The method testInput() checks if a given input string is accepted by the DFA by traversing its states based on the input symbol. The method process() starts calculating the epsilon-closure of the initial state and using it as the initial set of current states. While processing each symbol in the input string, the method iterates through the current states to identify all potential transitions for the current symbol. With every transition, it computes the epsilon-closure of the subsequent state and modifies the set of next states accordingly. Ensuring that all states reachable by epsilon transitions are taken into account. Once all symbols in the input string have been processed, the method verifies if any of the current states are accept states. If any current state is considered an accept state, the method will return true, indicating that the input string has been accepted by the automaton. If none of the current states are accept states, it returns false to indicate rejection. This function is called by the testInput() function for every input string, enabling the program to determine and present whether each string is accepted or rejected by the automaton.

6. Problems & Limitations

Problems Faced:

1) Each member's contributions to the conversion algorithm may differ due to challenges in understanding the assignment or other members' code. To balance this, we assigned tasks such as creating the GUI or writing the report documentation to ensure an even distribution of contributions among group members.

2) To avoid accidentally altering the conversion results when integrating the GUI and the conversion algorithm, the output is presented solely in text format which may be less intuitive and harder to interpret.

Limitations:

1) Users can only input right-linear grammar. Other forms, like left-linear grammar, are not supported.

2) There should not be any extra spacing in the regular grammar input.

3) Each production rule must be inputted on a separate line for clarity and proper parsing.

4) Users must use the capitalized letter "E" to represent epsilon. If the text file contains the symbol epsilon or a state starting with "E", it must be adjusted beforehand.

5) If a state has more than one transition, it must be input as "A->0B|1C" on a single line. Inputting transitions on separate lines like "A->0B" and "A->1C" will result in incorrect output.

7. References

Programiz. (2023, November 28). *Design flowchart in Programming (With*

    examples). https://www.programiz.com/article/flowchart-programming

GeeksforGeeks. (2023, November 7). *Java JTabbedPane*. GeeksforGeeks.

    https://www.geeksforgeeks.org/java-jtabbedpane/

Data structures in Java - Javatpoint. (n.d.). www.javatpoint.com.

    https://www.javatpoint.com/data-structures-in-java

NESO Academy. (n.d.). https://nesoacademy.org/cs/04-theory-of-computation

W3Schools.com. (n.d.). https://www.w3schools.com/java/java_oop.asp