TRIBHUVAN UNIVERSITY

INSTITUTE OF ENGINEERING

PULCHOWK CAMPUS

A PROJECT REPORT ON

“RegEx2DFA Converter”

Submitted by:

Aman Dhaubanjar (079BCT010)

Ankit Pokhrel (079BCT016)

Ankit Prasad Kisi (079BCT017)

Submitted to:

DEPARTMENT OF ELECTRONICS AND COMPUTER ENGINEERING

IOE, Pulchowk Campus

Kathmandu Nepal

August 01, 2024

ACKNOWLEDGEMENT

We would like to extend our sincere gratitude to **Mr. Daya Sagar Baral**, Lecturer, Department of Electronics and Computer Engineering, Pulchowk Campus, Institute of Engineering, for his guidance and support throughout this semester teaching us Object Oriented concepts in great detail.

In our project, we pay special thanks for the author and developer of C++, **Bjarne Stroustrup**, whose book laid the foundation for us to write effective code. Additionally, we would like to acknowledge the developers and maintainers of Graphviz, an open-source graph visualization software. Their contributions have been invaluable in providing a robust and flexible tool that has greatly facilitated the visualization of our DFA graphs.

Additionally, we are grateful to **Mr. Anuj Ghimire** for his invaluable insights on Automata, which have provided us with the knowledge to work in this field.

ABSTRACT

This report presents the development process, algorithm analysis, and outcomes of our "**RegEx2DFA Converter**" project. Automata, in simple terms, are abstract machines used to recognize patterns and process strings, such as regular expressions, according to specific rules. Our project aims to develop a program that aids students and educators in the field of theory of computation by automating the construction of Deterministic Finite Automata (DFA). This tool eliminates the need for manual DFA construction, often done through a trial-and-error approach, by leveraging the power of C++ and its object-oriented programming capabilities.

The project involved designing algorithms to address various challenges encountered during the development process, such as converting regular expressions into a computer-understandable format, parsing this format using our coded algorithms to generate a Non-Deterministic Finite Automaton (NFA), and subsequently converting the NFA into a DFA. The final output is a ".dot" file, which can be utilized by Graphviz library to visually represent the graphs.

Throughout the project, we applied principles from our Object-Oriented Programming course, gaining practical experience in software design, algorithms, and teamwork. We collaborated closely, utilizing each team member's strengths to overcome challenges and make informed logical decisions. Our project supervisors, Mr. Daya Sagar Baral and Mr. Anuj Ghimire, provided valuable guidance and feedback, ensuring the project aligned with our learning objectives. As a result, we successfully designed and developed a program that transforms regular expressions into DFAs, effectively applying theoretical knowledge to practical implementation.

CONTENTS

1. OBJECTIVES

1.1. To Automate DFA Construction:

Develop a tool that automatically converts regular expressions into Deterministic Finite Automata (DFA), eliminating the need for manual construction and reducing potential errors.

1.2. To Support Educational Use:

Create a resource that can be used in educational settings to teach the concepts of regular expressions, NFAs, and DFAs.

1.3. To Develop a User-Friendly Interface:

Implement an intuitive interface that allows users to input regular expressions, view the corresponding NFA and DFA, and visualize the results using Graphviz.

1.4. To Apply Object-Oriented Principles:

Utilize C++'s object-oriented programming features to manage data structures and algorithms efficiently, providing a practical application of concepts learned in the Object-Oriented Programming course.

1.5. To Provide Accurate and Efficient Conversion:

Ensure the program accurately and efficiently converts regular expressions to DFAs, handling edge cases and providing reliable results.

1.6. To Generate Visual Representations:

Produce visual representations of the DFAs in DOT format, compatible with graph visualization tools like Graphviz, to aid in the analysis and understanding of the automata.

2. INTRODUCTION

Automata theory studies abstract machines, or automata, and their role in solving computational problems. It has roots in early 20th-century work by Alan Turing and Alonzo Church, who developed foundational concepts like the **Turing machine** and lambda calculus. These ideas led to the Church-Turing thesis, which asserts that any computational problem solvable by algorithms can be performed by a **Turing machine**. Automata theory, including the study of finite automata and regular expressions introduced by Stephen Kleene, is essential for understanding and designing computational systems and languages.

Before diving into automata, it's important to familiarize ourselves with some fundamental terms:

1. **Alphabet**: A finite set of symbols, usually represented as Σ, used to construct strings. For example, *Σ = {a, b}*.
2. **String**: A sequence of symbols from an alphabet. For example, *"abba"* is a string over the alphabet *{a, b}*.
3. **Language**: A set of strings formed from an alphabet. A language can be finite or infinite. For example, the language *{a, ab, abb}* consists of strings made from the alphabet *{a, b}*.
4. **Regular Expression**: A symbolic representation used to describe regular languages. Regular expressions can define patterns within strings, such as *"a*b"* meaning zero or more *'a'*s followed by *a 'b'*.
5. **State**: A condition or situation in the computational process of an automaton, represented by a **circle** in diagrams.
6. **Transition**: The movement from one state to another in an automaton, triggered by input symbols. Transitions are depicted **as arrows** between states.
7. **Initial State**: The state where an automaton begins processing. It is usually represented by an **arrow pointing** to it without a source state.

8. **Accepting (or Final) State**: A state in which an automaton accepts a string as part of the language. It is represented by a **double circle**.

Now that we have a basic understanding of the terms used in Automata, let's focus on some operators we used in regular expressions and how they actually generate languages. In our project, we focused on three fundamental operators: union (+), concatenation (.), and Kleen closure (*). Here's a concise explanation of each:

1. **Concatenation (.)**: This operator connects two expressions in sequence. For instance, if we have the expressions *"a"* and *"b",* the concatenation *"a.b"* results in the string *"ab"*. It signifies that the symbols or expressions should appear one after the other.

2. **Union (+)**: The union operator indicates a choice among multiple options. For example*, "a + b"* matches either *"a"* or *"b"*. It allows the creation of expressions where multiple possibilities are considered valid.

3. **Kleen Closure (*)**: This operator represents zero or more repetitions of the preceding element. For instance, *"a*"* can match *"", "a", "aa", "aaa",* and so on. It is used to specify that a particular pattern can repeat any number of times, including not appearing at all.

Let's quickly look at some examples to be clear in this:

3. ***(a+b)****: { "", "a", "b", "aa", "ab", "ba", "bb", "aaa", "aab", "aba", "abb", ... }*
4. ***(ab)****: { "", "ab", "abab", "ababab", ... }*
5. ***ab(a+b)****: { "ab", "aba", "abb", "abaa", "abab", "abba", "abbb", ... }*
6. ***a*b***: { "b", "ab", "aab", "aaab", ... }*

2.1. How is String Processed in Automata?

In automata theory, an automaton processes a string by starting from an initial state (denoted by an **arrow** pointing to it). As it reads each symbol from the string, the **automaton transitions** to a corresponding state based on predefined rules. This sequence of transitions continues until all symbols in the string have been processed. If the automaton reaches a final state, denoted by a **double circle**, after processing the entire string, then the string is considered **accepted** by the automaton.

There are two main types of automata that can process strings in this way:

1. **Deterministic Finite Automaton (DFA)**: A DFA has **exactly one transition** for each symbol from every state, leading to a unique path through the automaton for any given input string.
2. **Non-deterministic Finite Automaton (NFA)**: An NFA can have **multiple transitions** for the same symbol from a single state, including transitions that don't consume any symbols **epsilon transitions(ε).** An NFA accepts a string if there is at least one path from the initial state to a final state that corresponds to the string.

For a string to be accepted, the automaton must end in one of its final states after processing all the symbols in the string. **This acceptance indicates that the string is part of the language recognized by the automaton**.

Here is a small example to get into automaton with regular expression:

Q) Generate DFA with Regular expression: *ab(a+b)*.*

Solution:

ab(a+b)*: *{ "ab", "aba", "abb", "abaa", "abab", "abba", "abbb", ... }*

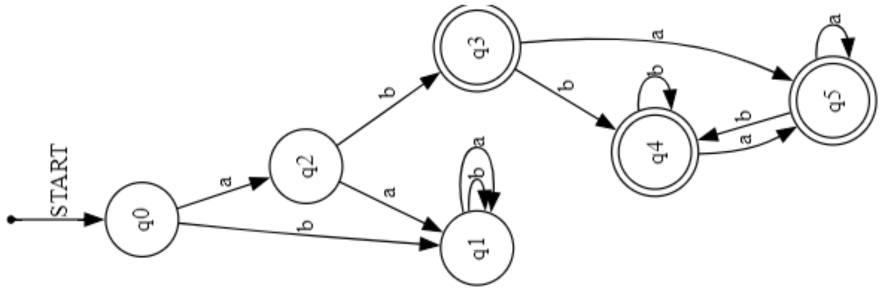


fig: DFA for ab(a+b)* expression

Let's break down the generated DFA, which consists of six states, with q0 as the starting state and q3, q4 and q5 as the ending states. The state transitions are represented by arrows. Let's process a few strings through this DFA:

Processing "abba":

- Starting from q0, the DFA transitions to q2 on reading 'a'.
- From q2, it moves to q3 on reading 'b'.
- With another 'b', it transitions to q4.
- Finally, on reading 'a', it reaches q5. Since q5 is an ending state, the string "abba" is **accepted**.

Processing "baba":

- Starting from q0, the DFA transitions to q1 on reading 'b'.
- From q1, it stays in q1 on reading 'a', then again on 'b', and once more on 'a'. However, since q1 is not an ending state, the string "baba" is **rejected**.

3. APPLICATIONS

Automata, particularly in language processing automation, has broad applications in various fields, including software engineering, data and text processing, cybersecurity and network analysis, control systems and robotics, bioinformatics, and digital systems design. Some of them are listed below:

· **Compiler Design**:

- Tokenization of source code into keywords, identifiers, and symbols.
- Syntax analysis and error detection.

· **Text Processing**:

- Searching and pattern matching in text editors and search engines.
- Data validation (e.g., email validation, password checking).

· **Natural Language Processing (NLP)**:

- Speech recognition and text-to-speech systems.
- Parsing and understanding the grammatical structure of sentences.

· **Finite State Machines in Robotics and Automation**:

- Control systems for managing different operational states.
- Behavior modeling in robotic systems.

· **Digital Circuit Design**:

- Design and analysis of digital circuits using finite state machines.
- State minimization and optimization.

· **Bioinformatics**:

- Sequence alignment and pattern recognition in DNA/RNA analysis.
- Identifying genes and functional motifs.

4. LITERATURE SURVEY

Finite automata are fundamental theoretical models used to recognize patterns and process strings of symbols. The development of finite automata began in the 1950s with significant contributions from Michael Rabin and Dana Scott, who formalized the concepts of deterministic finite automata (DFA) and nondeterministic finite automata (NFA). These models laid the groundwork for modern computation theory.

A Deterministic Finite Automaton (DFA) is characterized by its deterministic nature, where each state has a unique transition for each input symbol. This ensures that for any given input, the DFA has a single predictable path of execution. Conversely, a Nondeterministic Finite Automaton (NFA) allows for multiple transitions from a state for the same input symbol and can include epsilon (ε) transitions that do not consume any input. This nondeterminism enables NFAs to explore multiple computational paths simultaneously.

Despite these differences, DFAs and NFAs are equivalent in terms of the languages they can recognize. Any language accepted by an NFA can also be accepted by a DFA. The conversion from an NFA to a DFA, known as the subset construction method, is crucial for practical applications where deterministic behavior is required.

Finite automata have wide-ranging applications, including pattern matching in regular expressions, lexical analysis in compilers, and modeling network protocols. Their theoretical significance and practical utility make them a cornerstone of computer science, influencing both algorithm design and the development of computational models.

5. EXISTING SYSTEM

Finite Automata (FA) is a fundamental computational model used to recognize patterns and characterize regular languages. It plays a significant role in analyzing and recognizing natural language expressions and has a long history of development. Numerous algorithms and systems have been proposed for working with finite automata, reflecting its importance in theoretical and practical applications. It is also a core topic in our curriculum for the **Theory of Computation** course.

Despite its relevance, there is a noticeable lack of coding implementations in this area, particularly in C++. While there are existing programs in languages like Python and JavaScript, we observed a gap in C++ implementations. This presented us with a valuable opportunity to develop and work on a solution in C++, addressing this gap and contributing to the field.

6. METHODOLOGY

To convert regular expressions to DFAs, we follow several steps using different methods. Here's a brief overview of each step:

i) **Conversion of Regular Expression to Postfix Notation**
ii) **Creating an Epsilon-NFA using the Postfix Notation**
iii) **Constructing a Minimized DFA from the Epsilon-NFA**

6.1. Conversion of Regular Expression to Postfix Notation

To begin, we input a regular expression into our code. However, the code needs to determine how to process the expression and understand the precedence of each operator. To address this, we use an operator-precedence parser known as the "Shunting Yard Algorithm."

6.1.1. What is the Shunting Yard Algorithm?

The shunting yard algorithm was invented by Edsger Dijkstra to convert an infix expression to postfix. Many calculators use this algorithm to convert the expression being entered to postfix form. This algorithm is an operator-precedence parser that is specifically designed to parse mathematical expressions into postfix notation for computation. Postfix notation (Reverse Polish notation) is a mathematical notation in which the operators follow the numbers.

For example, `4 + 5` will turn into `4 5 +`

Postfix notation removes the need for parentheses and allows computer programs to read in mathematical expressions one symbol after the other, instead of worrying about operator precedence and parentheses during computation.

Before converting our expression to postfix notation, we must first establish the precedence of the operators. This is crucial during the conversion process, as it dictates the order in which operators are pushed onto the **stack.** The precedence, from highest to lowest, is as follows: Closure (Kleene star) $a*$, Concatenation ab, and Union $a+b$.

Note: The concatenation expression usually does not have a symbol between the two letters *ab*. So, in order to easily handle this we will be using an *?* symbol between the two letters for every concatenation.

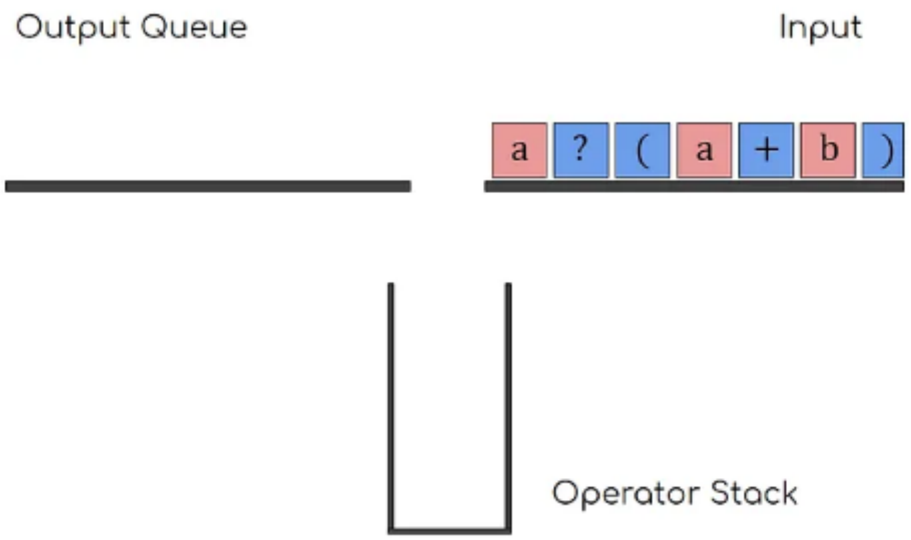
6.1.1.1. Basic Rule of the Algorithm:

- If the input symbol is a letter, append it directly to the output queue.
- If the input symbol is an operator, check the operator stack. If the operator on top has equal or higher precedence, pop it from the stack and append it to the output queue. Repeat until the current input operator has higher precedence or the stack is empty.
- If the input symbol is an operator with a left parenthesis on top of the stack, push the operator onto the stack above the parenthesis.
- If the input symbol is a left parenthesis, push it onto the operator stack.
- If the input symbol is a right parenthesis, pop operators from the stack to the output queue until a left parenthesis is found. Remove both parentheses and continue.

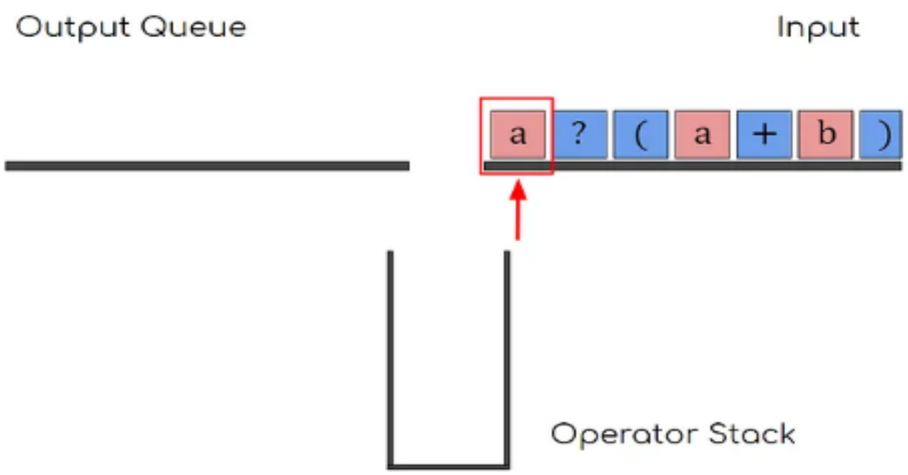
6.1.1.2. Shunting-Yard Algorithm Visualized

This diagram visually explains the Shunting-Yard Algorithm using our regular expression. The input section moves from left to right, reading each character. The operator stack stores operators and responds to new ones based on predefined rules. The output queue ultimately forms the final postfix notation.

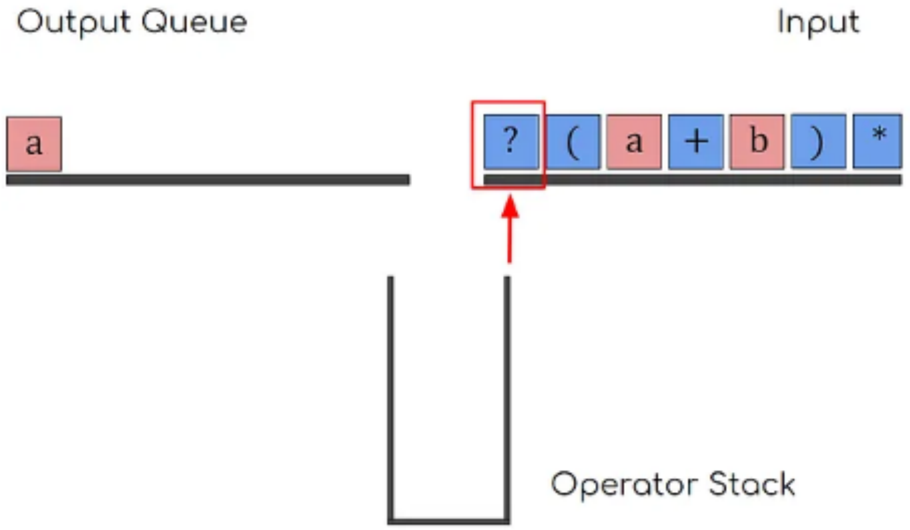
Regular Expression : a(a+b)*b



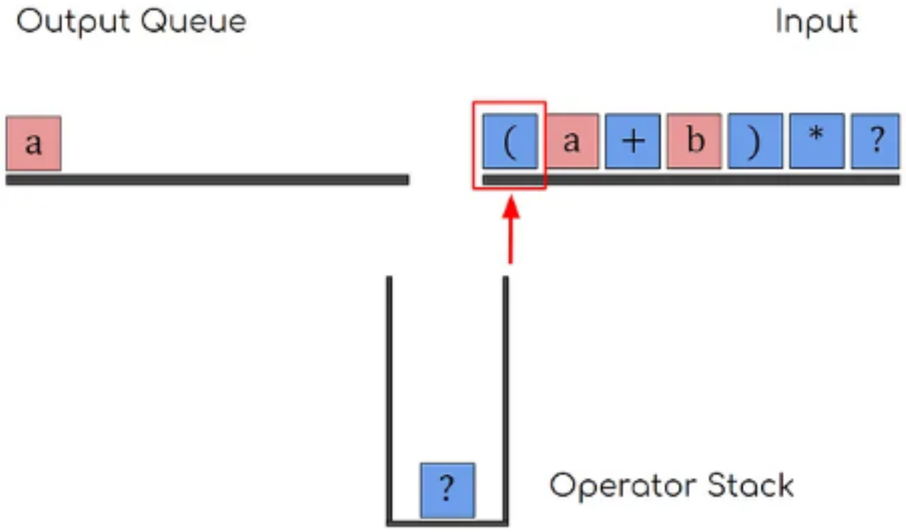
The first symbol we will read is `[a]`



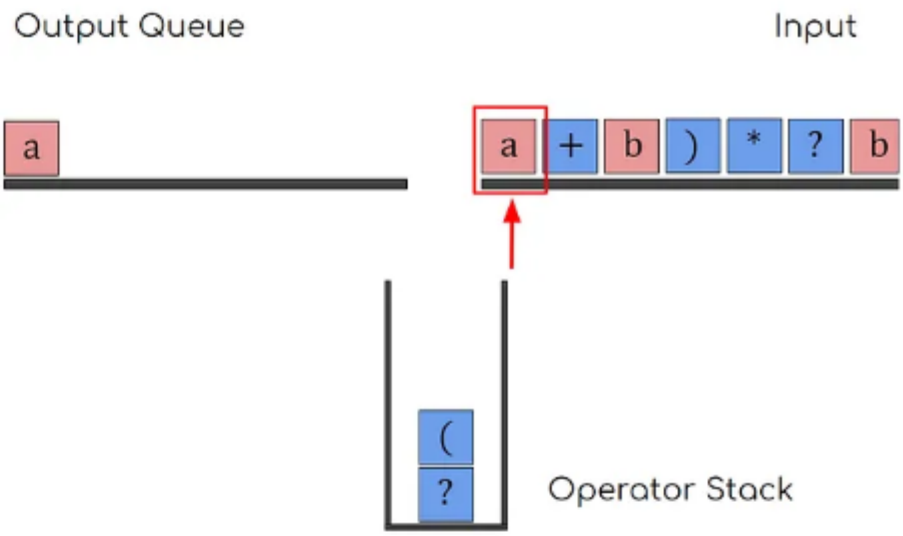
`[a]` is a letter, so we can append it directly to the output queue.



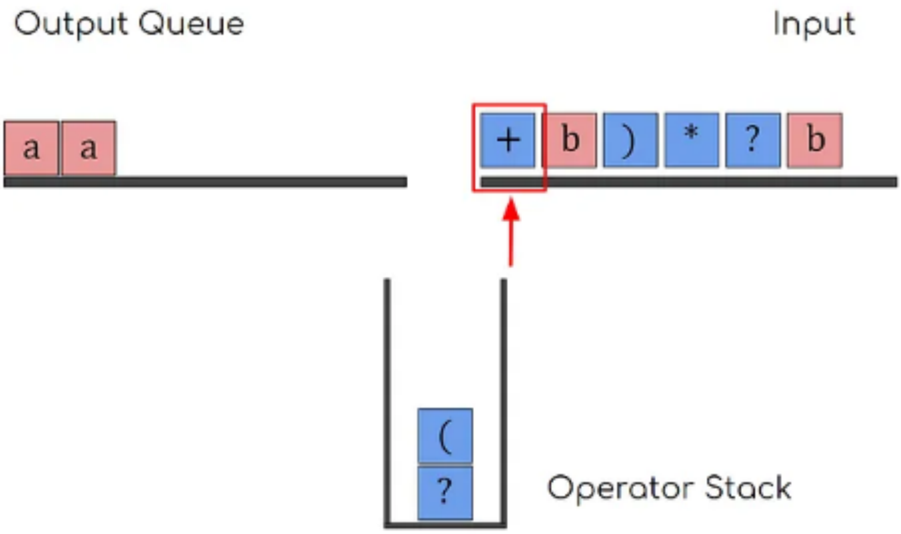
[?] is an operator and there are no operators in the stack yet. So, append [?] to the stack.



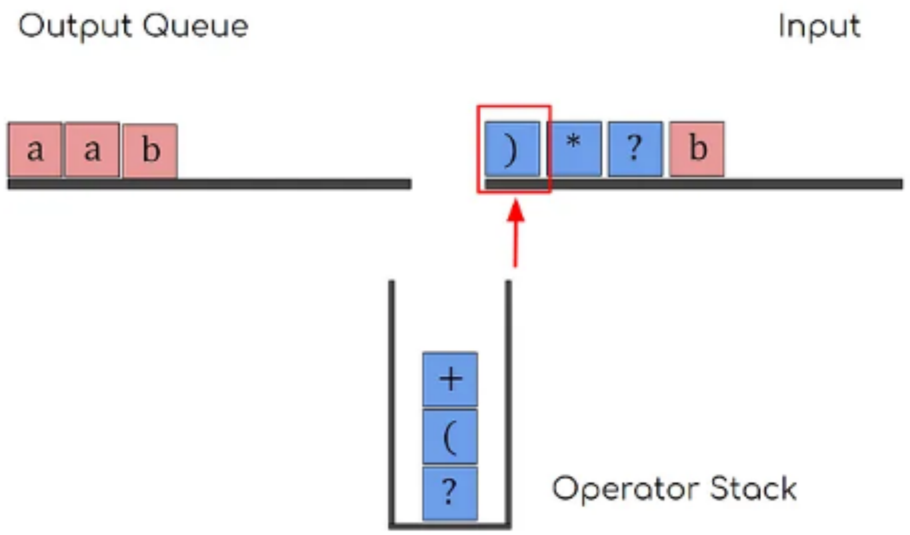
We encountered a **left parenthesis**, push it on to the operator stack



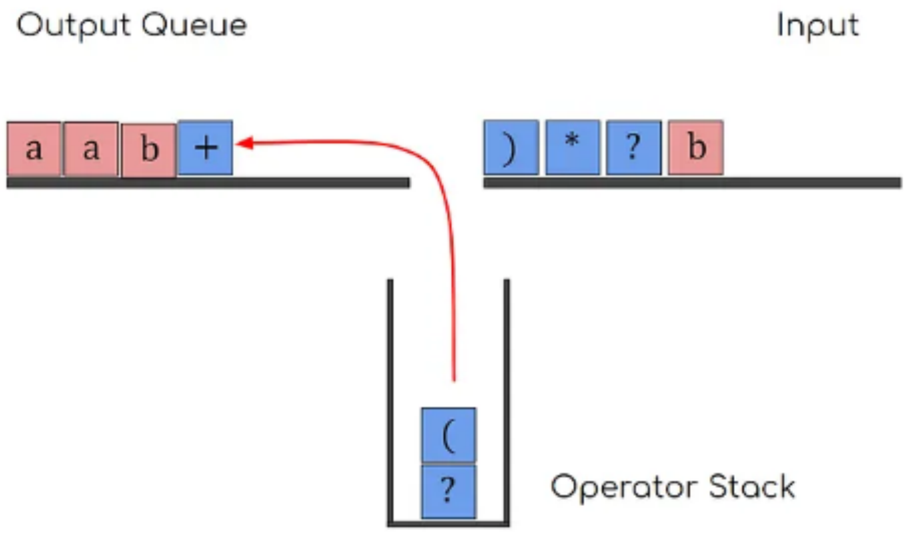
The next symbol is an `[a]`. Append it directly to the output queue.



The next symbol is a `[+]`. Now we have to look at the operator on the top of the stack to decide what to do. The operator on top of the stack is `[(]`. A left parenthesis does not have higher precedence than a union symbol. So we can append `[+]` to the stack.

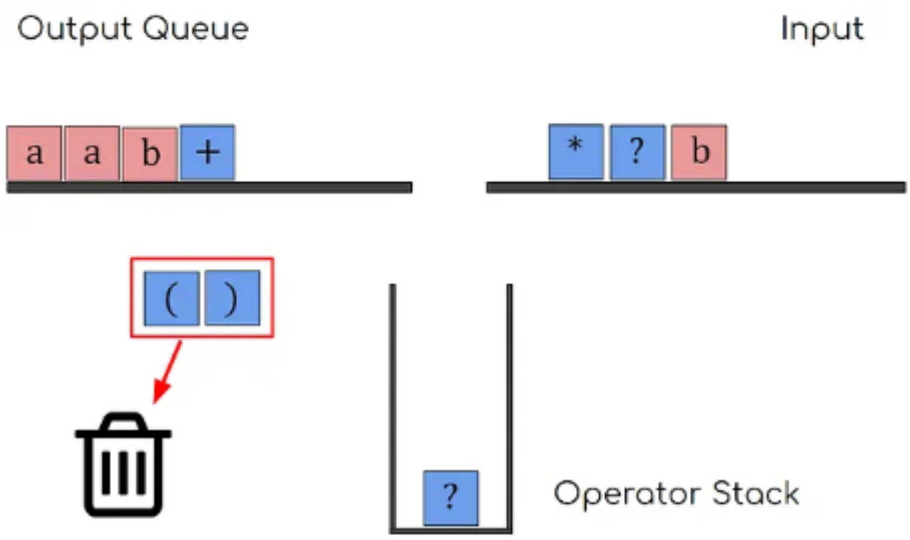


We encountered a **right parenthesis**. We must now look at the operator stack and pop every operator that is on top of a left parenthesis to the output queue.

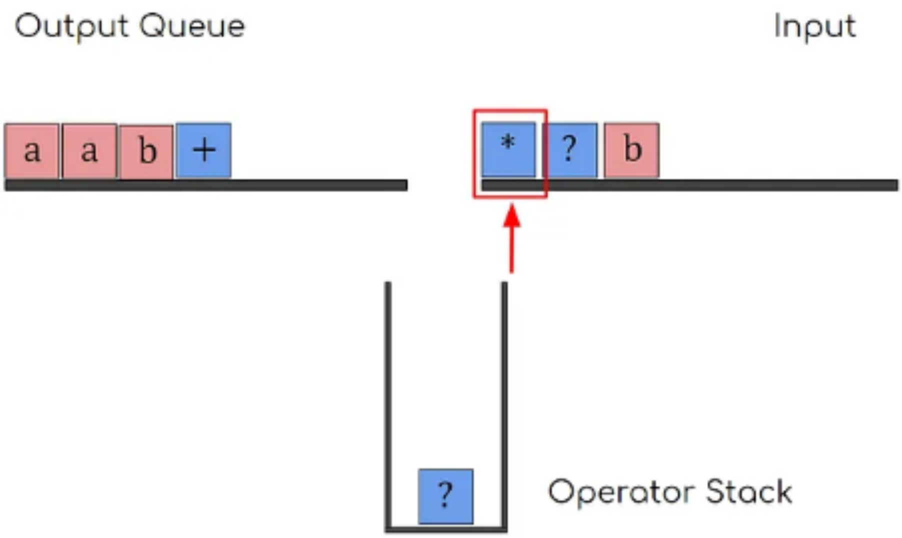


We have encountered the **left parenthesis** we were looking for. Good, that means we have a matching pair of parentheses. If we reach the bottom of the stack without finding a left parenthesis, that means we have mismatching parentheses and will result in an error.

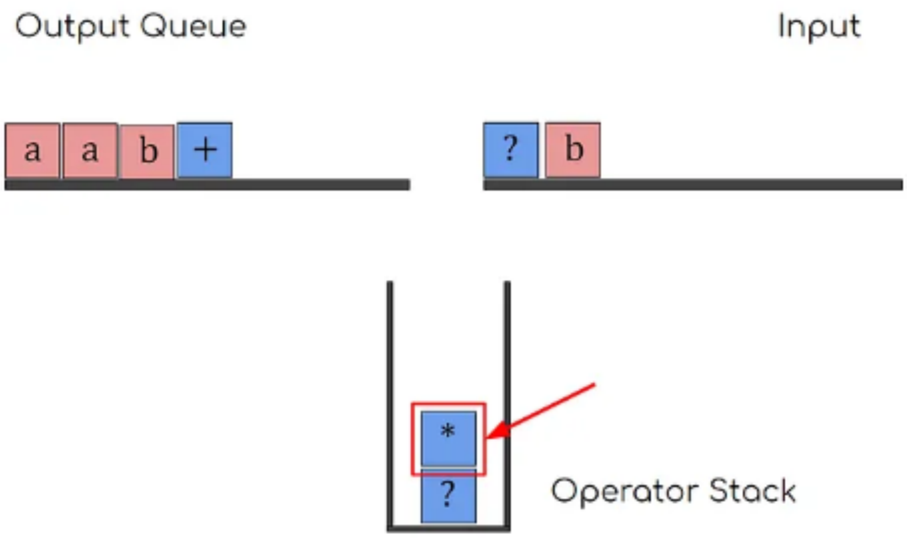
We can now discard both parentheses.



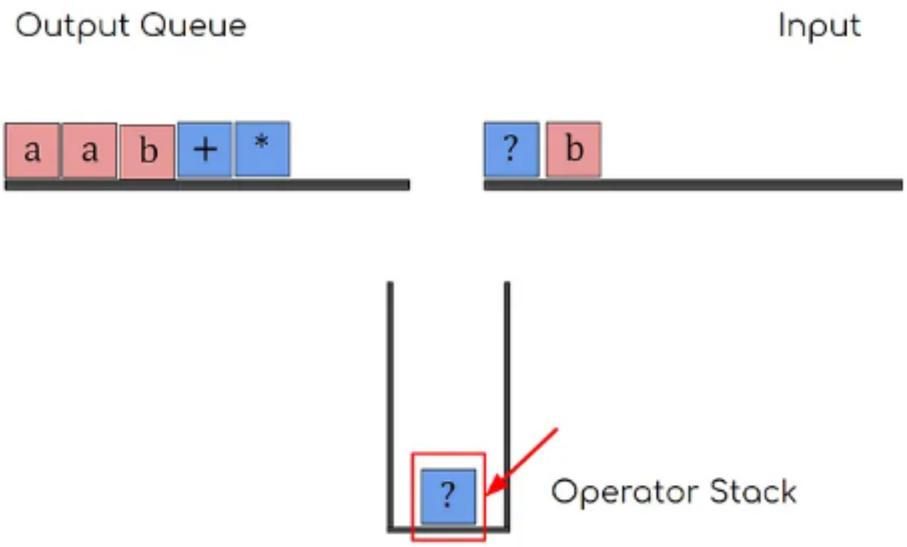
Move to the next symbol.



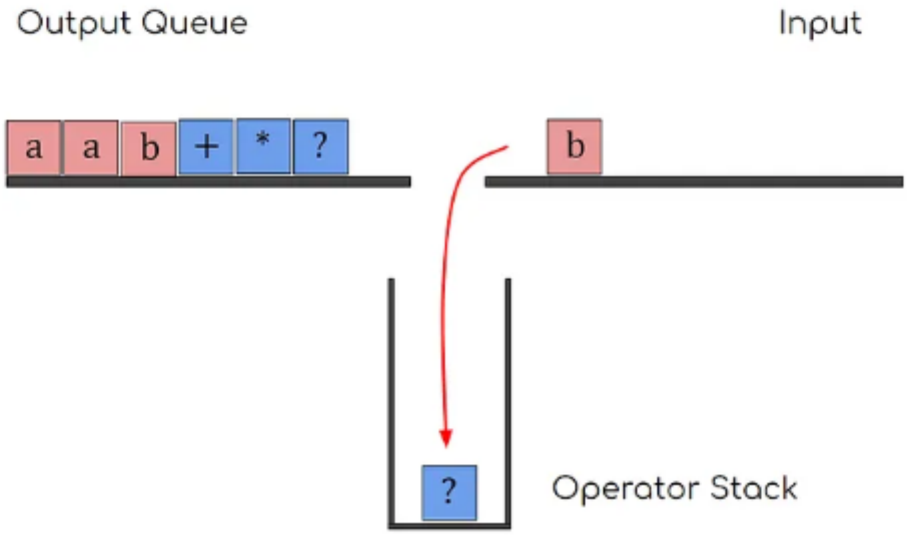
The next symbol is a `[*]`. Now let's check the operator stack to see what to do next... A Kleene star has higher precedence than a concatenation. So, append the Kleene star to the operator stack.



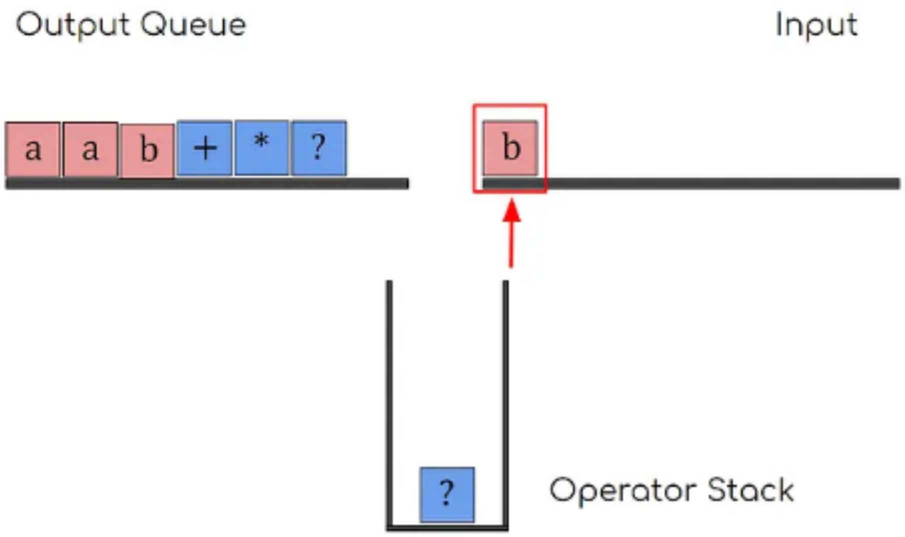
A Kleene star `[*]` is on top of the stack, which has higher precedence than `[?]`. Now we have to move `[*]` to the output queue.



A concat `[?]` is on top of the stack. This is also the symbol of our current character. Because `[?] == [?]`, we also have to pop the `[?]` from the stack and append it to the output queue.

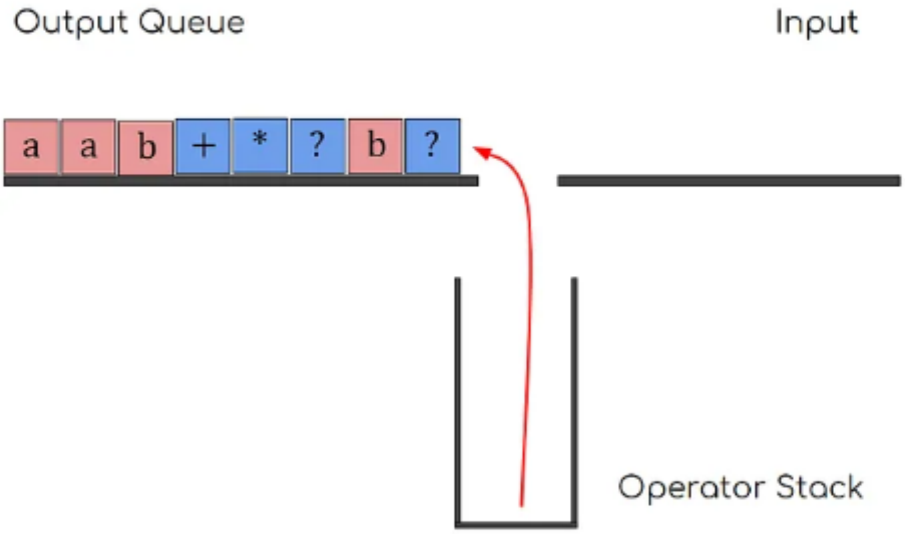


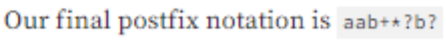
Move to the next symbol.



The next symbol is a `[b]`. Append it directly to the output queue.

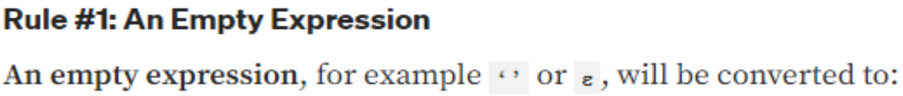
Now this will be our final postfix notation generated by S



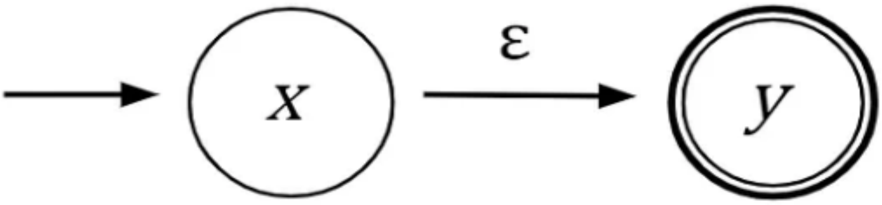

Our final postfix notation is `aab+*?b?`

6.2. Creating an Epsilon-NFA using the Postfix Notation

Now that we've created the postfix notation, we need to construct an ε-NFA using it. For this, we use **"Thompson's Construction Algorithm"** which allows us to build the ε-NFA. Thompson's Construction Algorithm is **a method for converting regular expressions to their respective NFA diagrams**. It uses just **five simple steps** to convert any postfix notation to ε-NFA.

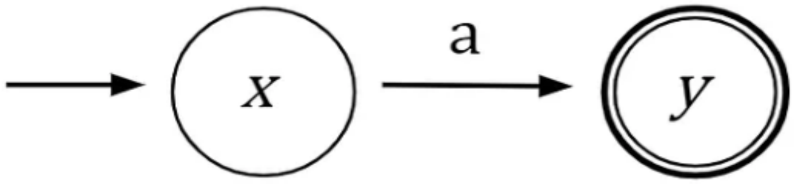

Rule #1: An Empty Expression

An empty expression, for example `‘’` or `ε`, will be converted to:



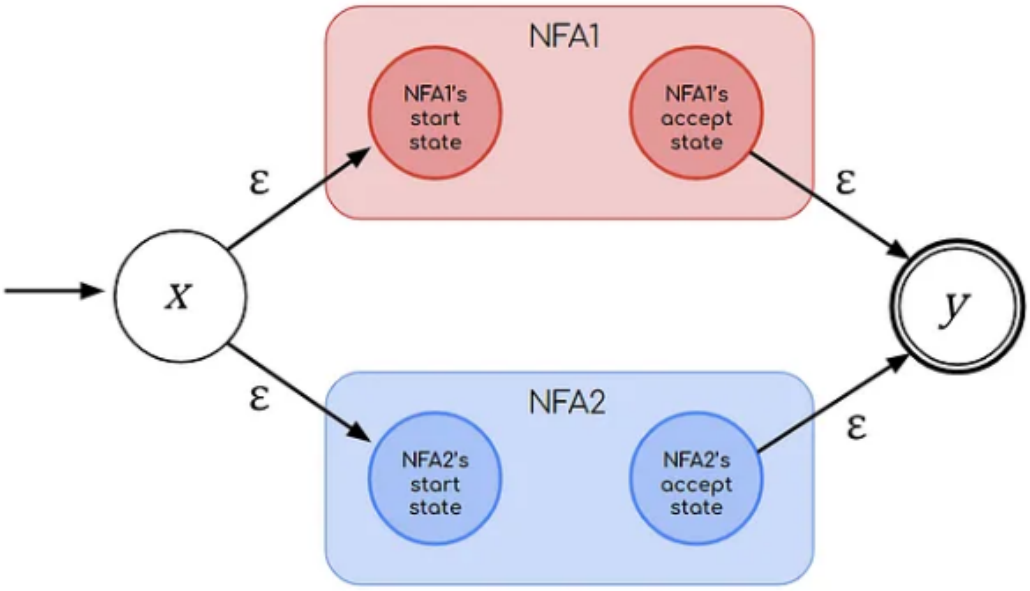
Rule #2: A symbol

A **symbol**, such as `a` or `b`, we will converted to:



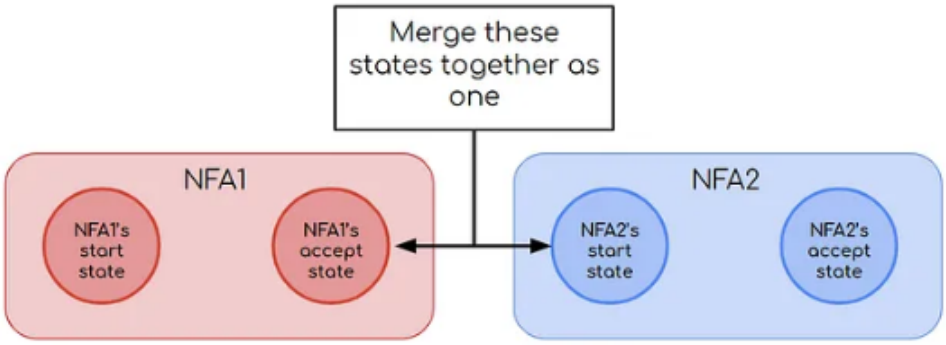
Rule #3: Union expression

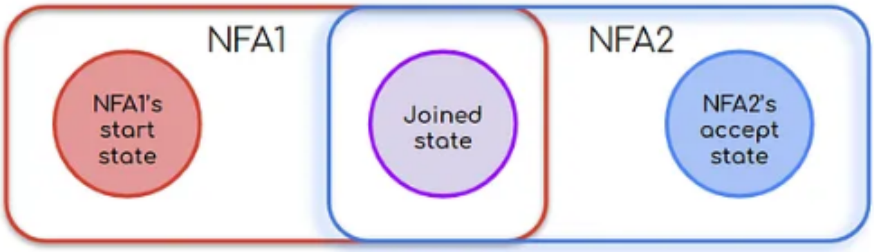
A **union expression** `a+b` will be converted to:



Rule #4: Concatenation expression

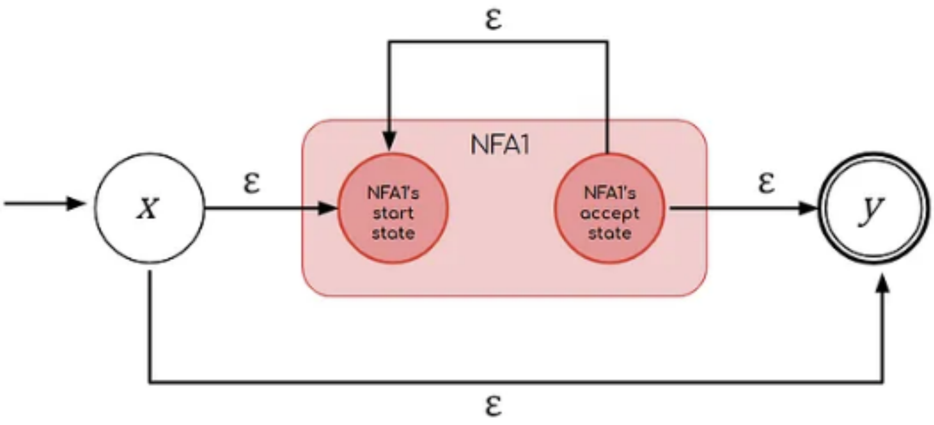
A concatenation expression `ab` or `a?b` will be converted to:





Rule #5: A closure/kleene star expression

A closure `a*` will be converted to:



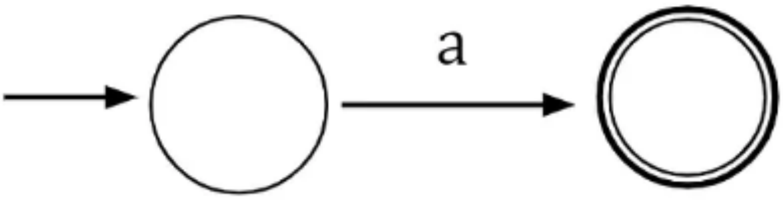
6.2.1. Using Thompson’s Rules for our Regular Expression

Now that we know the rules of Thompson’s Construction Algorithm, let’s convert our regular expression ***“aab+*?b?”*** obtained from “Shunting Yard Algorithm” into its respective NFA diagram.

The first character we encounter is an ***a***.

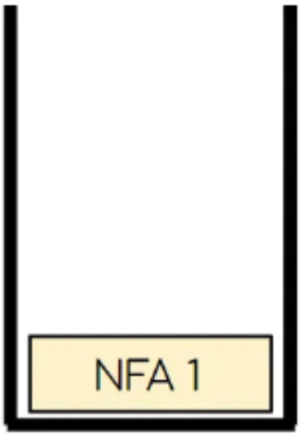
aab+*?b?

`a` is a symbol. So, we follow **Rule #2** to produce the following NFA:



We will call this diagram **NFA 1.**

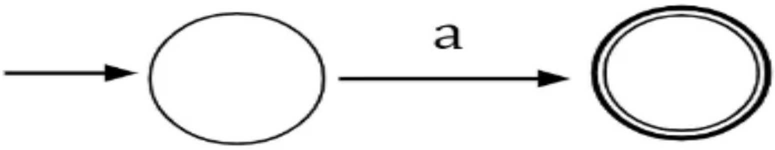
In order to store our NFAs to reference later on in the algorithm, we will be using a **stack.** Let’s push **NFA 1** onto this stack now.



The next character is also an `a`.

aab+*?b?

Follow **Rule #2…**

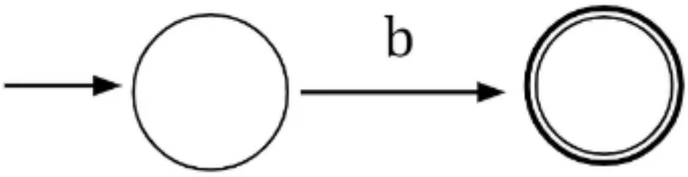


We will call this diagram **NFA 2** and push it onto the stack.

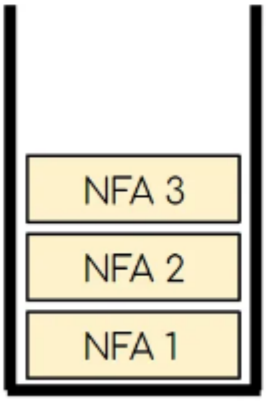
The next character is a `b`.

aab+*?b?

Follow **Rule #2…**

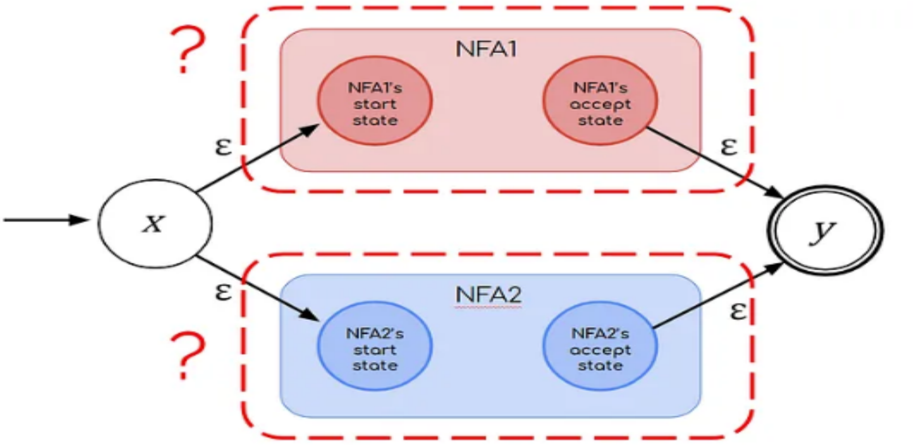


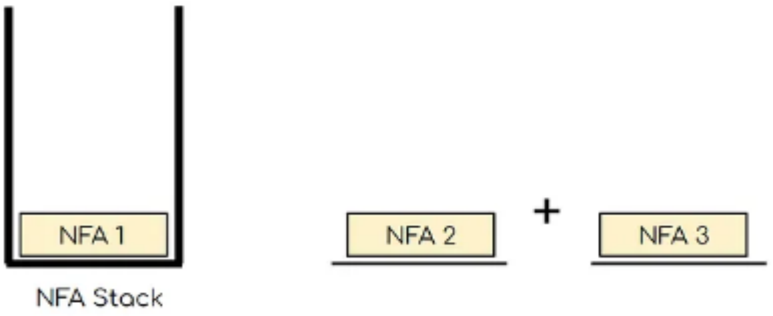
We will call this diagram **NFA 3** and push it onto the stack.



aab+*?b?

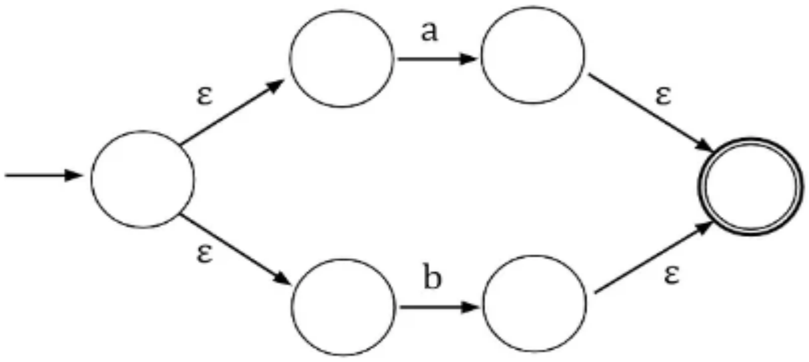
This is the symbol for a **union expression**. That means we have to follow **Rule #3.** You may be asking yourself, *what two NFA graphs would we use for the red (NFA1) and blue (NFA2) diagrams?*





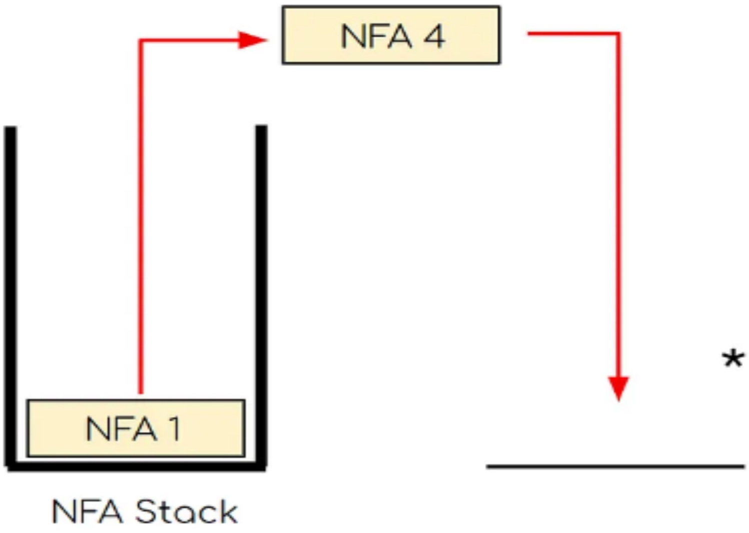
Note that because NFA 2 was pushed on to the stack **BEFORE** NFA 3, we must follow the format **NFA 2 + NFA 3** and not NFA 3 + NFA 2.

Now we can union the two NFAs together. Remember that we have to declassify NFA 1 and NFA 2's start state and accept state's because we now have a new start state and accept state for this union-ed NFA.

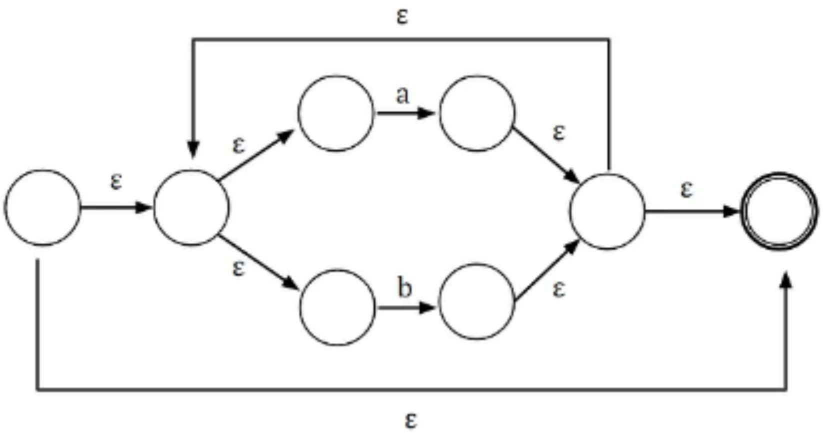


aab+*?b?

Rule #5 shows us how to handle closures. A closure only refers to a single NFA diagram. So we will only pop the top-most NFA off of the stack, which is **NFA 4.**



Following Rule #5, we can create a closure for NFA 4. Remember that we have to declassify NFA 4's start state and accept state because we have a new start state and accept state from Rule #5. Our new NFA will be...

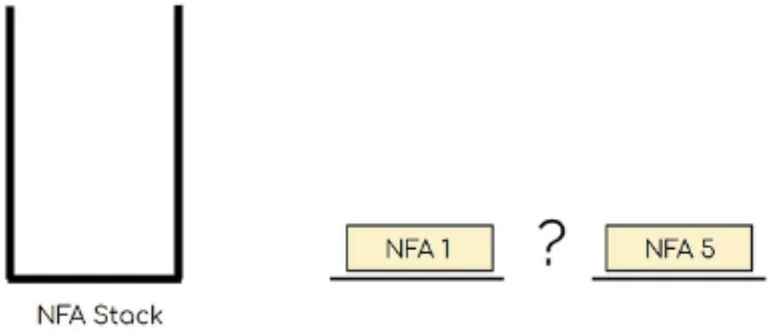


We will call this diagram **NFA 5** and push it onto the stack.

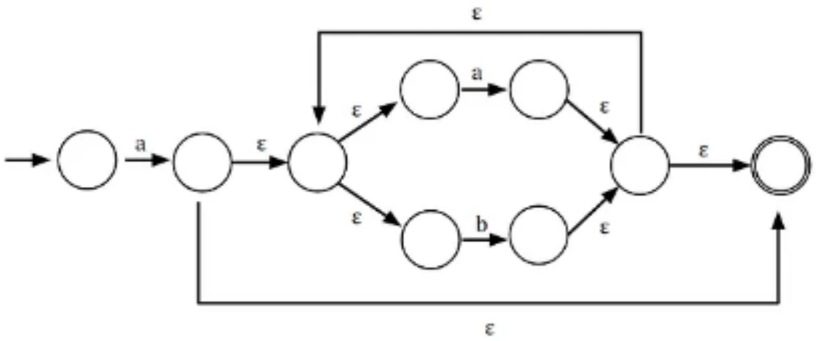
The next symbol in the expression is a **concatenation** `?`.

aab+*?b?

This will require **Rule #4** to compute. Similar to the union expression, a concatenation operation will need *two* NFAs to combine and will follow the same pop order (top-most NFA on the stack goes on the right, and the following NFA goes on the left).



Now we can follow Rule #4 to concat NFA 1 and NFA 5.

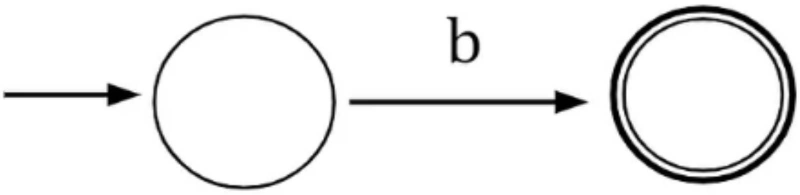


We will call this diagram **NFA 6** and push it onto the stack.

The next symbol is another `b`.

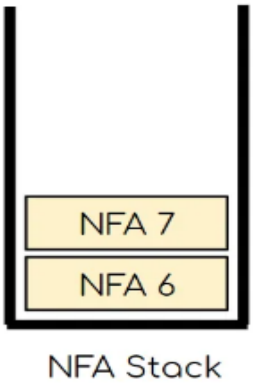
aab+*?b?

Just follow **Rule #2...**



We will call this diagram **NFA 7** and push it onto the stack.

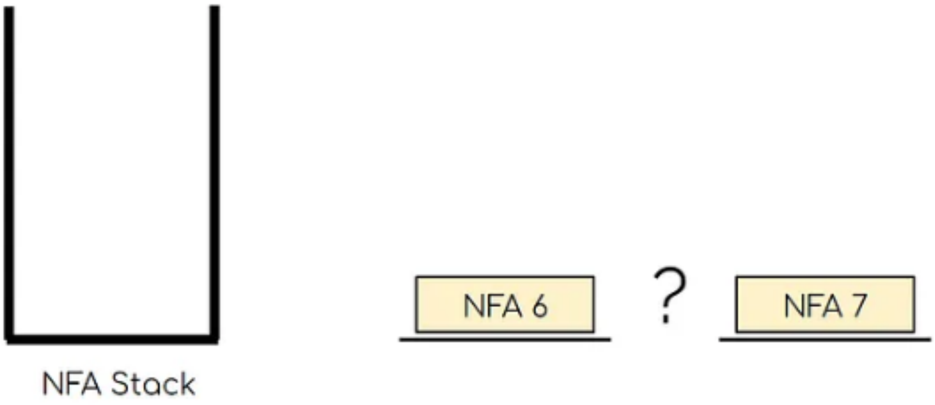
We will call this diagram **NFA 7** and push it onto the stack.



The last symbol is another **concatenation** ?.

aab+*?b?

Once again, we have to follow **Rule #4**. First, we must pop the NFAs off of the stack...



Now we can concat these NFAs together.

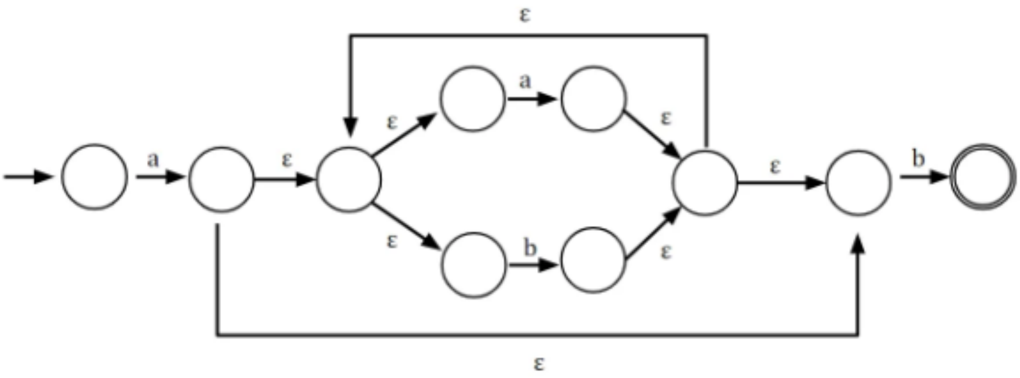


Fig: Final NFA diagram for "aab+*?b?"

6.3. Constructing a Minimized DFA from the ε-NFA

With the ε-NFA now constructed, the next step is to convert it into a DFA using the ***"Subset Construction Method"***. The steps below outline how to efficiently construct a DFA from an NFA.

1. **Remove ε-transitions using ε-closure**: The ε-closure of a state is the set of all states reachable from that state by following ε-transitions.

2. **Find the ε-closure of the start state**: Begin by finding the ε-closure for the start state of the NFA.

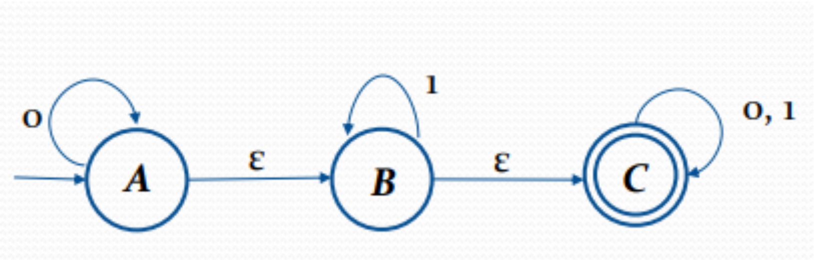
3. **Determine transitions for each alphabet symbol**: For each symbol in the alphabet, determine the set of states reachable from the current set of states.

4. **Find the ε-closure for new states**: For each new set of states found in step 3, compute the ε-closure and determine transitions for each alphabet symbol. Repeat this process until no new states are discovered.

5. **Define start and accepting states**: The start state of the DFA corresponds to the ε-closure of the NFA's start state. Accepting states in the DFA include any state sets that contain an accepting state from the NFA.

Let's walk through a basic example to see how the "Subset Construction Method works" in converting an ε-NFA to a DFA.

Example:



Here we are converting ε-NFA to DFA

Starting state = ε-closure(A), since A is starting state of ε-NFA

= {A, B, C}

Input symbol are *{0, 1}*

Let, *δD* be the transition function of DFA.

Now we process for *={A, B, C}* and input as follows:-

δD({A,B,C}, 0) = ε-closure(δ({A,B,C}, 0)

= ε-closure({A,C})

= ε-closure(A) È ε-closure(C)

= {A,B,C} È {C}

= {A,B,C}

δD({A,B,C}, 1)

= ε-closure(δ({A,B,C}, 1)

= ε-closure({B,C})

= ε-closure(B) È ε-closure(C)

= {B,C} È {C}

= {B,C}

δD({B,C}, 0)

= ε-closure(δ({B,C}, 0)

= ε-closure(C)

= {C} δD({B,C} 1)

= ε-closure(δ({B,C}, 1)

= ε-closure({B,C})

= ε-closure(B) È ε-closure(C)

= {B, C} È {C}

={B,C}

δD(C, 0)

= ε-closure(δ(C),0)

= ε-closure(C)

= {C} δD(C, 1)

= ε-closure(δ(C),1)

= ε-closure(C)

= {C}

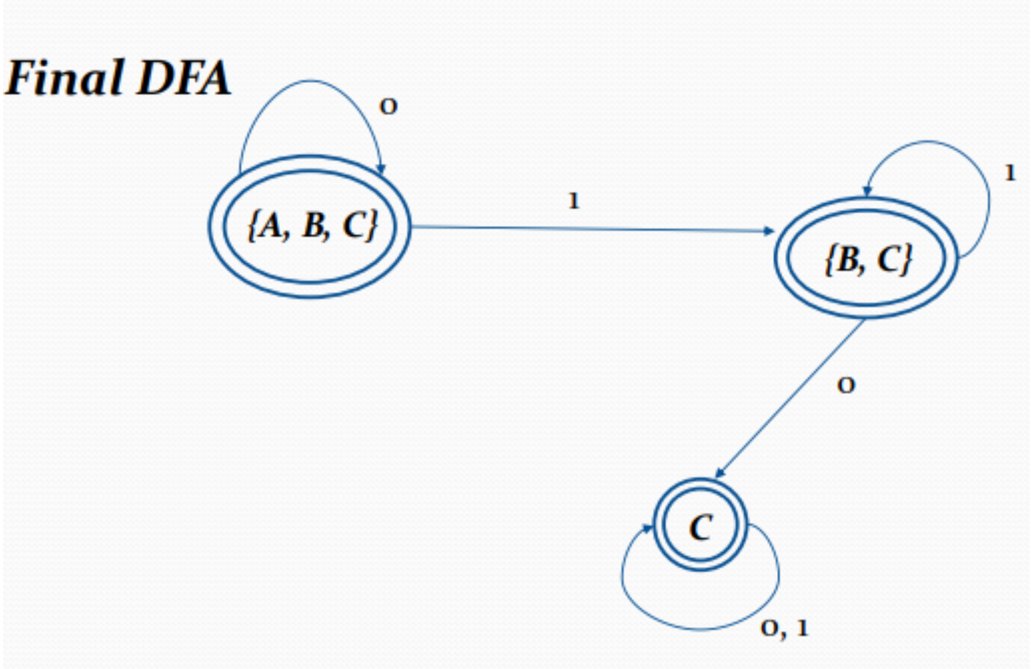


Fig: DFA form ε-NFA generated using "Subset Construction Method"

To summarize, we began with a regular expression and converted it into postfix notation using the Shunting Yard Algorithm. We then applied Thompson's Algorithm to the postfix notation to construct an ε-NFA. Finally, we employed the Subset Construction Method to transform the ε-NFA into a DFA.

7. IMPLEMENTATION

With the proposed objective and methodology, the pathway was implemented almost similar to what was proposed.

- Within the first week of undertaking the project, project was chosen and thus, subsequently a rough path was discussed and project proposal was made.
- After the project proposal, a rough sketch of the project schedule was created.
- Then we studied the Shunting-Yard Algorithm and Thompson’s Construction Algorithm.
- Then we created an infix to postfix converter which gave us the expression that can be used in Thompson’s Construction Algorithm.
- Then we studied about various data structures like vectors, stacks, queues, unordered sets, unordered maps and others.
- And using concept of Thompson’s Construction Algorithm, we developed regular expression to eplison-NFA converter.
- Then we learnt about Graphviz library as an extension and used it to visualize our eplison-NFA and fix bugs and errors.
- Then we integrated Graphviz into our code.
- Then we organized the codes by separating them into multiple files to simplify code for debugging purpose.
- Then we learnt the concept of Subset Construction Algorithm and developed eplison-NFA to NFA converter.
- Then we merged the regular expression to eplison-NFA converter and eplison-NFA to DFA converter.
- Then we added exception handling to handle unexpected errors.
- Final Code was tested and debugged.
- Final Program was documented and a presentation was prepared.

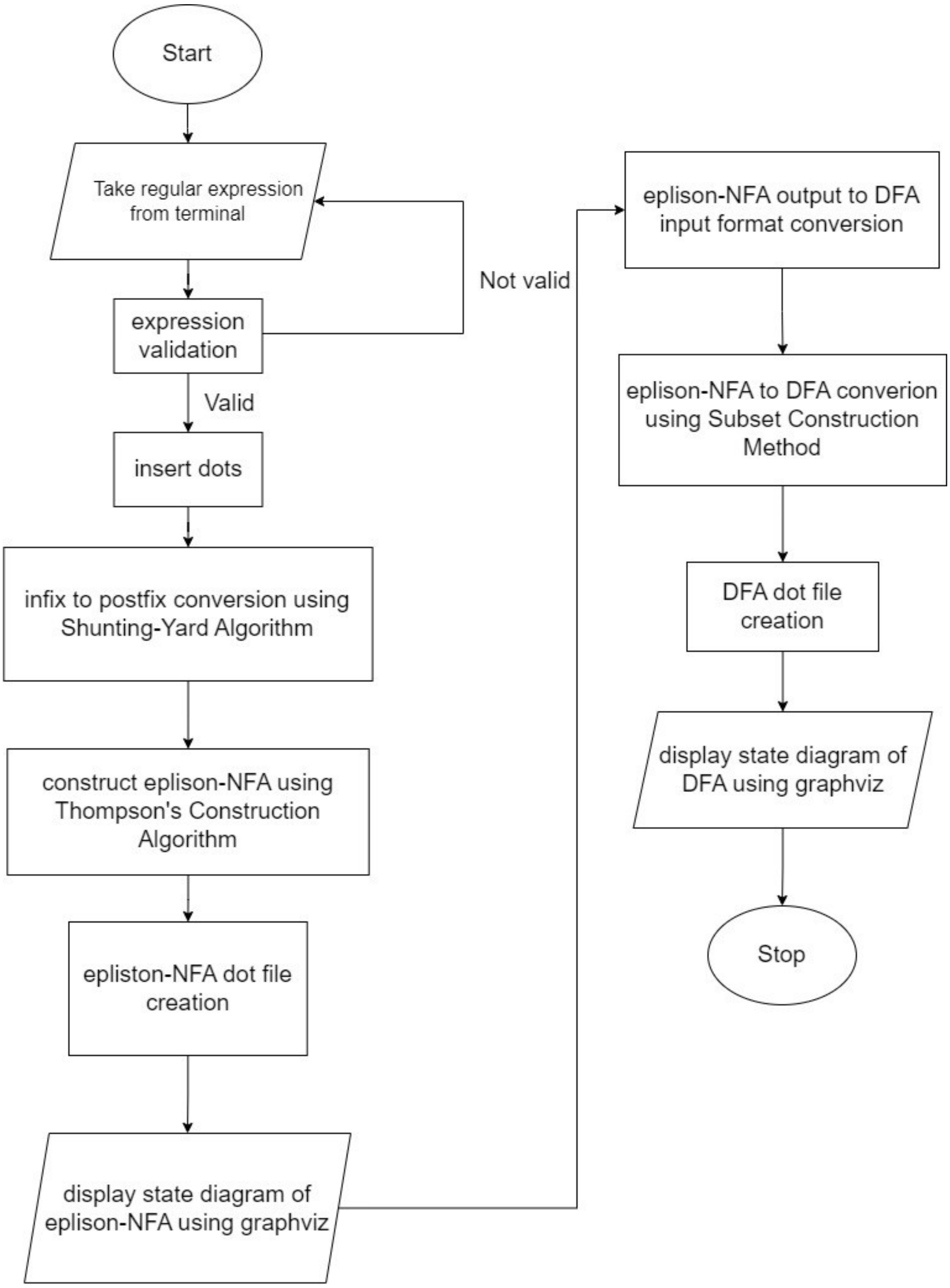

Start
Take regular expression from terminal
expression validation
Not valid
Valid
insert dots
infix to postfix conversion using Shunting-Yard Algorithm
construct eplison-NFA using Thompson's Construction Algorithm
epliston-NFA dot file creation
display state diagram of eplison-NFA using graphviz
eplison-NFA output to DFA input format conversion
eplison-NFA to DFA converion using Subset Construction Method
DFA dot file creation
display state diagram of DFA using graphviz
Stop

Fig: Flowchart

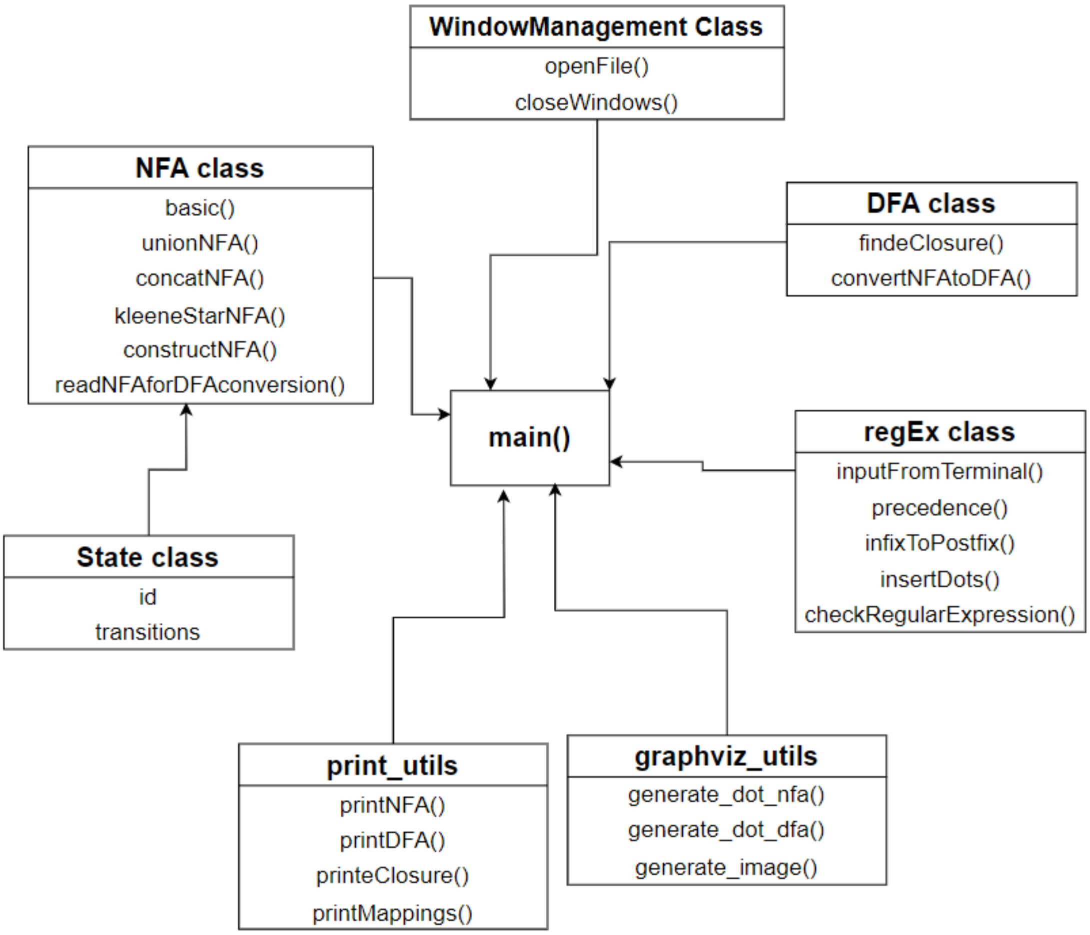


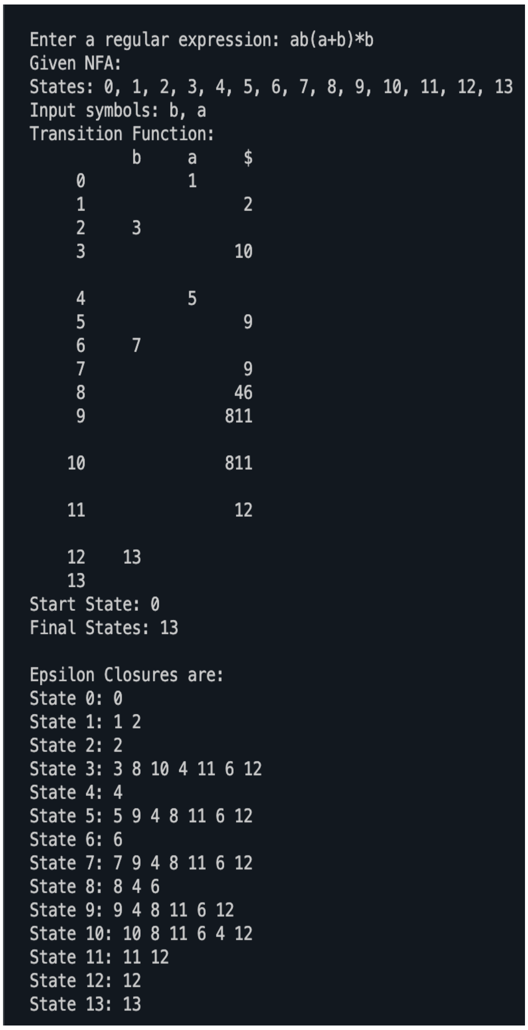
Fig: Generalized Block Diagram

8. RESULTS

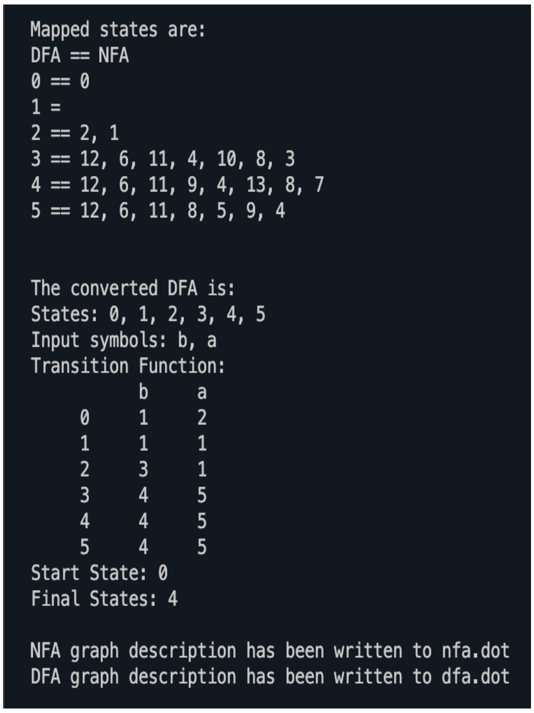
After implementing the final version of the **RegEx2DFA** program, the results were closely aligned with our expectations. The application performs as intended with the following functionality:

8.1. Regular Expression to DFA Conversion:

- The program successfully converts regular expressions into deterministic finite automata (DFA).
- It processes regular expressions using various algorithms, including the Shunting-Yard Algorithm for parsing and ***Subset Construction Method*** for conversion.



```
Enter a regular expression: ab(a+b)*b
Given NFA:
States: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13
Input symbols: b, a
Transition Function:
           b     a     $
     0           1
     1                 2
     2     3
     3                10

     4           5
     5                 9
     6     7
     7                 9
     8                46
     9               811

    10               811

    11                12

    12    13
    13
Start State: 0
Final States: 13

Epsilon Closures are:
State 0: 0
State 1: 1 2
State 2: 2
State 3: 3 8 10 4 11 6 12
State 4: 4
State 5: 5 9 4 8 11 6 12
State 6: 6
State 7: 7 9 4 8 11 6 12
State 8: 8 4 6
State 9: 9 4 8 11 6 12
State 10: 10 8 11 6 4 12
State 11: 11 12
State 12: 12
State 13: 13
```



```
Mapped states are:
DFA == NFA
0 == 0
1 =
2 == 2, 1
3 == 12, 6, 11, 4, 10, 8, 3
4 == 12, 6, 11, 9, 4, 13, 8, 7
5 == 12, 6, 11, 8, 5, 9, 4


The converted DFA is:
States: 0, 1, 2, 3, 4, 5
Input symbols: b, a
Transition Function:
           b     a
     0     1     2
     1     1     1
     2     3     1
     3     4     5
     4     4     5
     5     4     5
Start State: 0
Final States: 4

NFA graph description has been written to nfa.dot
DFA graph description has been written to dfa.dot
```

.

Fig: Conversion Process

8.2. Graph Visualization:

- The generated DFA is visualized using Graphviz, providing a clear graphical representation of the automaton.
- The program utilizes the Graphviz library to create visualizations that accurately reflect the structure of the DFA.

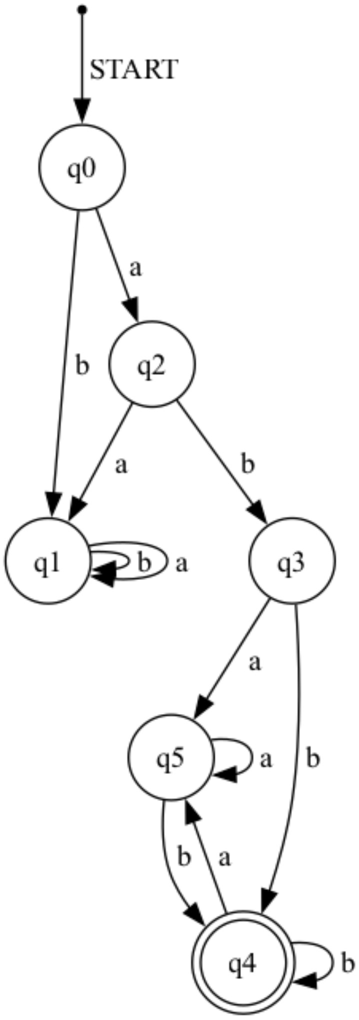
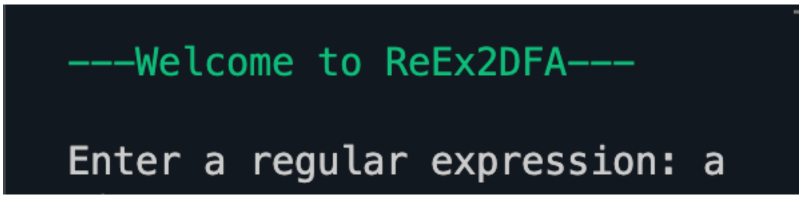


Fig: DFA state-diagram using Graphviz

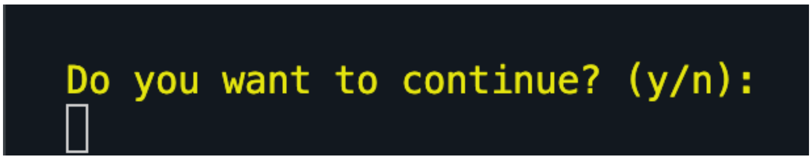
8.3. User Interaction:

- The application is command-line based. Users input their regular expressions directly, and the program generates the corresponding DFA.



```
---Welcome to ReEx2DFA---

Enter a regular expression: a
```

Fig: User inputs regular expression



```
Do you want to continue? (y/n):
```

Fig: User is asked if they want to continue

- The Graphviz-generated images of the DFA can be viewed using the default image viewer on the user's system. The image files are opened automatically.

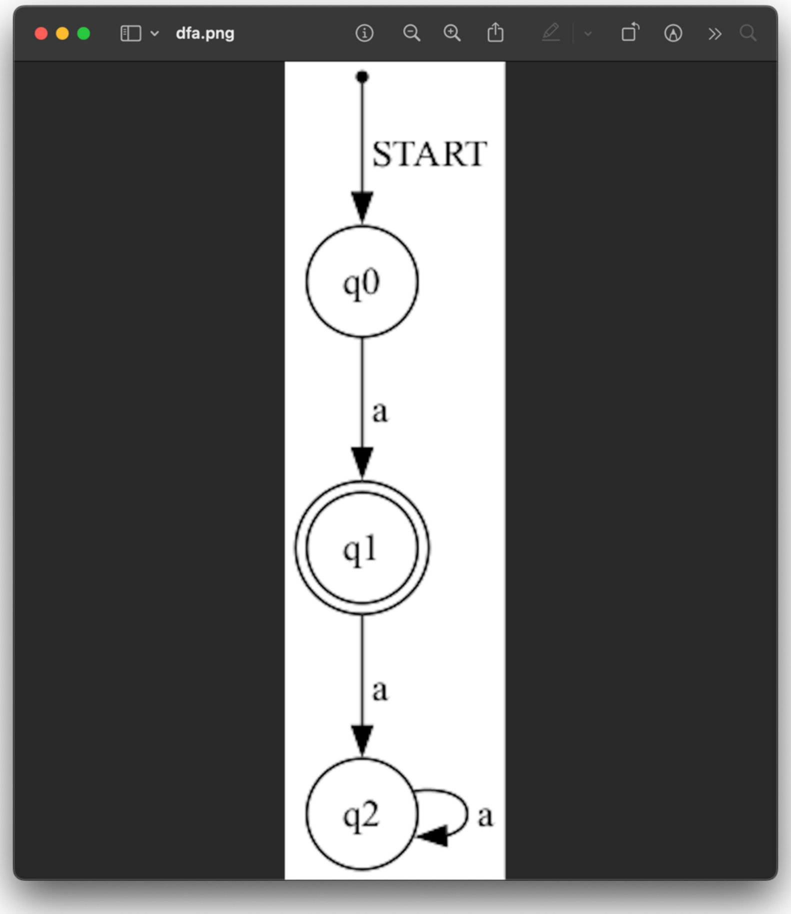


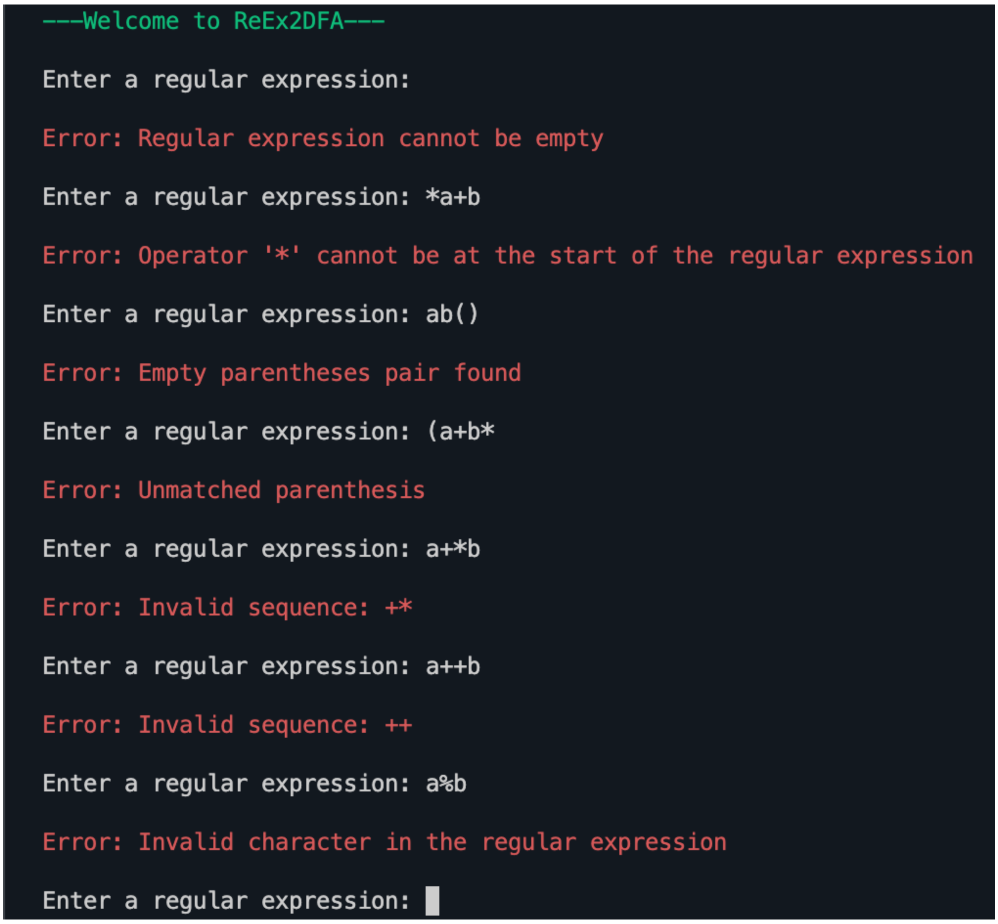
Fig: DFA image in default image viewer

8.4. File Handling:

- The program handles file input/output effectively, storing and retrieving generated DFA images as needed.

8.5. Error Handling:

- The program includes robust error handling for invalid regular expressions and file operations, ensuring a smooth user experience.



```
---Welcome to ReEx2DFA---

Enter a regular expression:

Error: Regular expression cannot be empty

Enter a regular expression: *a+b

Error: Operator '*' cannot be at the start of the regular expression

Enter a regular expression: ab()

Error: Empty parentheses pair found

Enter a regular expression: (a+b*

Error: Unmatched parenthesis

Enter a regular expression: a+*b

Error: Invalid sequence: +*

Enter a regular expression: a++b

Error: Invalid sequence: ++

Enter a regular expression: a%b

Error: Invalid character in the regular expression

Enter a regular expression: █
```

Fig: Error Handling for regular expression

Overall, **RegEx2DFA** meets the project requirements and provides a functional tool for converting and visualizing regular expressions as deterministic finite automata.

9. PROBLEMS FACED AND SOLUTIONS

- **Conceptualization and Understanding of Algorithms**: The idea for our project emerged from our "Theory of Computation" class. Although we had a solid understanding of several algorithms, we initially encountered challenges in visualizing and coding them due to the complexity involved.

- **Modular Code Integration**: We needed to divide the project into separate modules for each algorithm. Combining these modules into a cohesive system posed a challenge, but we successfully managed to integrate them.

- **Graphical Representation of State Diagrams**: Presenting state diagrams graphically was challenging until we discovered and utilized the Graphviz library, which facilitated this process.

- **Learning Data Structures**: We incorporated concepts such as stacks, queues, and vectors, which required additional learning. While this wasn't a problem per se, it did present a learning curve.

- **Cross-Platform Development**: Working in a group with different operating systems (macOS, Windows, Linux) posed compatibility challenges, but we eventually found effective solutions to collaborate smoothly across platforms.

10. LIMITATIONS AND FUTURE ENHANCEMENTS

While we completed the project with accurate results, there were some additional features we couldn't implement due to time constraints and limited resources. Our program currently supports three main regular expression operators: Concatenation (.), Kleene Closure (*), and Union (+). However, there are many other operations we could support, and expanding the set of supported regular expression operators is a potential area for future enhancement. Further, we believe that we could organize our code and make it much cleaner and easier to read. We could have used the object-oriented approach more effectively.

Additionally, our focus was primarily on the algorithmic aspects, so we did not include graphical elements in the user interface. In the future, we plan to develop a more interactive and user-friendly interface. Furthermore, although we can generate DFA, we noticed that other DFA implementations for the same regular expression often use fewer states. Therefore, we aim to refine our algorithms to produce more optimized and efficient DFA representations.

11. CONCLUSION AND RECOMMENDATION

Our goal was to develop a program that assists students and teachers in solving problems related to DFA construction, and we successfully achieved this objective. We applied Object-Oriented Programming (OOP) concepts and C++ to implement various algorithms, utilized different data structures, and employed Graphviz for generating state diagrams. We effectively constructed both NFA and DFA from regular expressions.

For future work, we recommend expanding the program’s functionality to support a broader range of regular expression operators and optimizing the algorithms for more efficient DFA construction. Enhancing the user interface to make it more interactive and user-friendly would also improve the overall experience. Additionally, refining the algorithms to produce more compact and optimized DFA representations could further enhance the program’s performance and utility.

12. REFERENCES

Research Paper and Notes:

1. Shravani Bahirat and team, “Regular Expression to Deterministic Finite Automata”, Research Paper, Jan 2023, available at *“https://www.irjet.net/archives/V10/i1/IRJET-V10I178.pdf”*
2. Anuj Ghimire, *“Finite Automata”*, Lecture notes on Theory of Computation, 2024
3. Indu, Jyoti, *“Technique for Conversion of Regular Expression to and from Finite Automata”*, Research Paper, June 2016, available at “*https://ijrra.net/Vol3issue2/IJRRA-03-02-14.pdf*”

Websites:

1. *https://medium.com/*
2. *https://graphviz.org/*
3. *https://geeksforgeeks.com*
4. *https://stackoverflow.com*
5. *https://youtube.com*

Books:

1. Daya Sagar Baral and Diwakar Baral’s *“The Secrets of Object Oriented Programming in C++”*
2. Robert Lafore, *“Object-Oriented Programming in C++”*, Fourth Edition