PROJECT REPORT

Lexical Analyser (C++)

PREPARED BY

Aditya Raj Kumawat (2020BTechCSE005)

Garv Baheti (2020BTechCSE031)

Namish Khandelwal (2020BTechCSE053)

FACULTY GUIDES

Dr. Suman Saha

Theoretical Foundation of Computer Science

Department of Computer Science Engineering

Institute of Engineering and Technology (IET)

JK Lakshmipat University Jaipur

20 December 2021

CERTIFICATE

This is to certify that the project work entitled "**Lexical Analyser (C++)**" submitted by **Aditya Raj Kumawat (2020BTechCSE005), Garv Baheti (2020BTechCSE031), Namish Khandelwal (2020BTechCSE053),** towards the partial fulfilment of the requirements for the degree of **Bachelor of Technology in Computer Science Engineering** of JK Lakshmipat University Jaipur is the record of work carried out by them under my supervision and guidance. In my opinion, the submitted work has reached a level required for being accepted.

Dr. Suman Saha

Associate Professor

Department of Computer Science

Institute of Engineering & Technology (IET)

JK Lakshmipat University Jaipur

Date of Submission: 20 December 2021

ACKNOWLEDGEMENTS

We would like to express our heartfelt gratitude to Dr. Suman Saha, Associate Professor, Department of Computer Science Engineering, Institute of Engineering & Technology for his valuable time and guidance that made the project work a success. We want to express gratitude toward JK Lakshmipat University to give us such great Faculty. We would thank our batchmates for aiding us at whatever second. We would likewise express gratitude toward JK Lakshmipat University to give a decent online stage in this difficult stretch of Coronavirus and unique on account of Dr. Suman Saha Sir for understanding our issues and his method of reaction at whatever point we get in touch with him. At last, I want to thank to everyone individuals who were engaged with this venture.

Sincerely yours,

Aditya Raj Kumawat (2020BTechCSE005)

Garv Baheti (2020BTechCSE031)

Namish Khandelwal (2020BTechCSE053)

OBJECTIVE

To generally build and understand the real-life application of a sort of Finite State Machine with the help of programming in a subtle way.

TABLE OF CONTENTS

INTRODUCTION

Finite Automata(FA) is the simplest machine to recognize patterns. The finite automata or finite state machine is an abstract machine that has five elements or tuples. It has a set of states and rules for moving from one state to another but it depends upon the applied input symbol. Basically, it is an abstract model of a digital computer. The following figure shows some essential features of general automation.

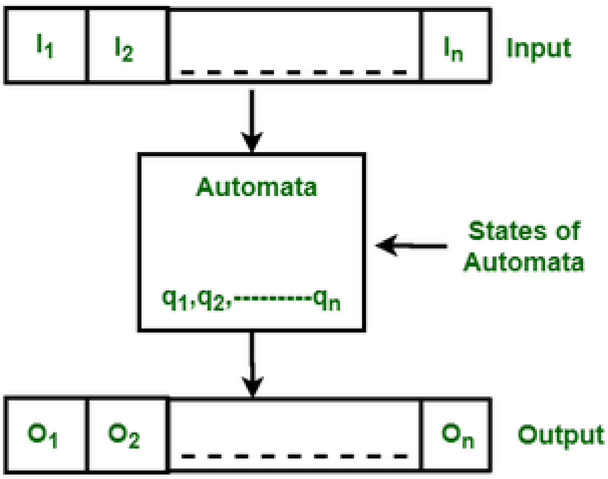


Figure: *Features of Finite Automata*

The above figure shows the following features of automata:

1. Input
2. Output
3. States of automata
4. State relation
5. Output relation

A Finite Automata consists of the following:

Q: Finite set of states.

Σ: set of Input Symbols.

q: Initial state.

F: set of Final States.

δ: Transition Function.

Formal specification of machine is

{ Q, Σ, q, F, δ }

FA is characterized into two types:

1) Deterministic Finite Automata (DFA) –

DFA consists of 5 tuples {Q, Σ, q, F, δ}.

Q: set of all states.

Σ: set of input symbols. (Symbols which machine takes as input)

q: Initial state. (Starting state of a machine)

F: set of final state.

δ: Transition Function, defined as δ : Q X Σ → Q.

In a DFA, for a particular input character, the machine goes to one state only. A transition function is defined on every state for every input symbol. Also, in DFA null (or ε) move is not allowed, i.e., DFA cannot change state without any input character.

For example, below DFA with Σ = {0, 1} accepts all strings ending with 0.

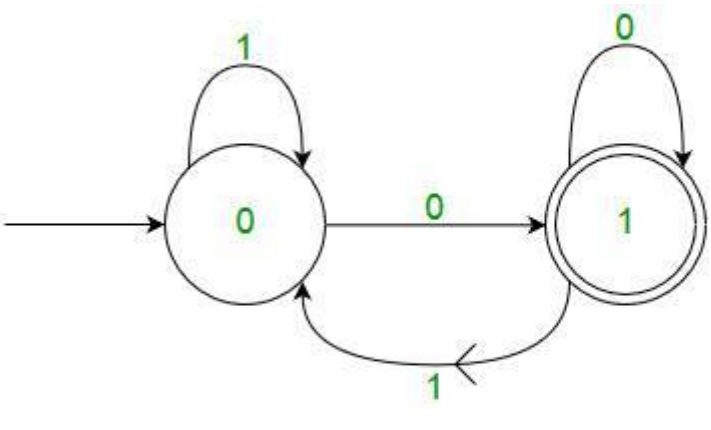


Figure: *DFA with Σ = {0, 1}*

One important thing to note is, ***there can be many possible DFAs for a pattern***. A DFA with a minimum number of states is generally preferred.

2) Nondeterministic Finite Automata (NFA) NFA is like DFA except following additional features:

1. Null (or ε) move is allowed i.e., it can move forward without reading symbols.
2. Ability to transmit to any number of states for a particular input.

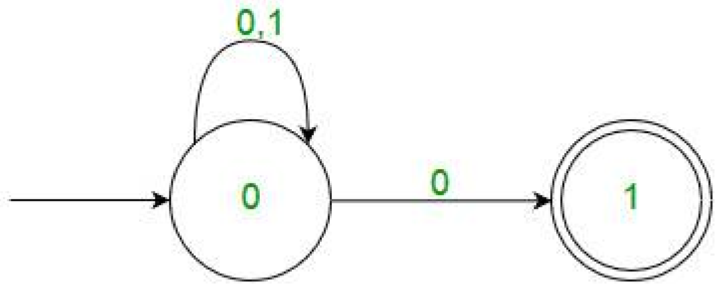
However, these above features don't add any power to NFA. If we compare both in terms of power, both are equivalent.

Due to the above additional features, NFA has a different transition function, the rest is the same as DFA.

δ: Transition Function

δ: $Q \times (\Sigma \cup \varepsilon) \rightarrow 2^Q$

As you can see in the transition function is for any input including null (or ε), NFA can go to any state number of states. For example, below is an NFA for the above problem.



NFA

One important thing to note is, ***in NFA, if any path for an input string leads to a final state, then the input string* is *accepted***. For example, in the above NFA, there are multiple paths for the input string "00". Since one of the paths leads to a final state, "00" is accepted by the above NFA.

Some Important Points:

- **Justification:**

Since all the tuples in DFA and NFA are the same except for one of the tuples, which is Transition Function (δ)

In case of DFA

δ: $Q \times \Sigma \rightarrow Q$

In case of NFA

δ: $Q \times \Sigma \rightarrow 2^Q$

Now if you observe you'll find out $Q \times \Sigma \rightarrow Q$ is part of $Q \times \Sigma \rightarrow 2^Q$.

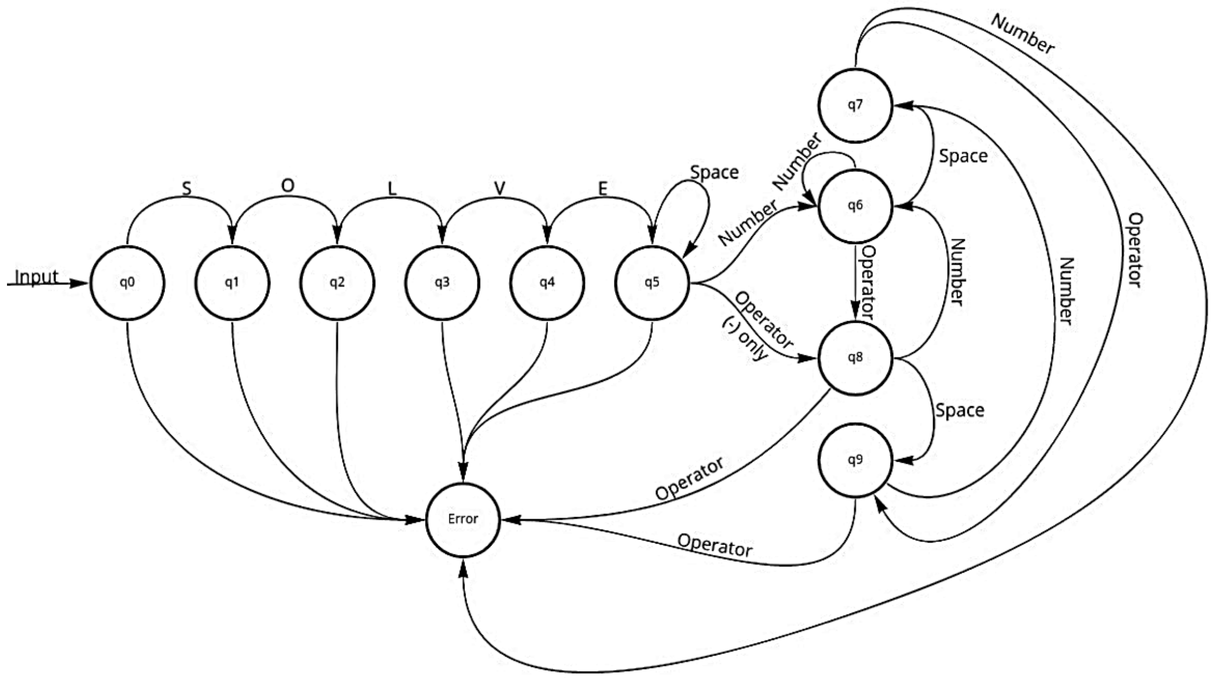
On the RHS side, Q is the subset of 2^Q which indicates Q is contained in 2^Q or Q is a part of 2^Q, however, the reverse isn't true. So mathematically, we can conclude that **every DFA is NFA but not vice-versa**. Yet there is a way to convert an NFA to DFA, so **there exists an equivalent DFA for every NFA**.

1. Both NFA and DFA have the same power and each NFA can be translated into a DFA.
2. There can be multiple final states in both DFA and NFA.
3. NFA is more of a theoretical concept.
4. DFA is used in **Lexical Analysis** in Compiler.

ABOUT PROJECT

Any system that requires some form of computation makes use of states and state machines as state machines provides an elegant approach of problem solving, modern computers use state diagrams for most part of mathematical computations like computing expressions. So, we can basically build a actually finite state machine (FSM) which can actually go through the particularly entire expression and check if it actually is valid or not, further showing how any system that requires some form of computation specifically makes use of states and state machines as state machines provides an elegant approach of problem solving, generally modern computers also use state diagrams to basically make mathematical computations like computing expressions, which really is fairly significant. Then we particularly build an expression tree using the string of expression, and specifically evaluate the value using the expression tree, further showing how then we for the most part build an expression tree using the string of expression, and essentially evaluate the value using the expression tree, which is quite significant.

STATE DIAGRAM



PROGRAM CODE (C++)

Main.cpp

```cpp
#include <bits/stdc++.h>

#include "Evaluator.h"
#include "Node.h"
#include "BC_FSM.h"

int main(int argc, char **argv) {

    for (int i = 0; i < argc; i++) {
            std::string cmd = argv[i];

            if (i == 0) continue;

            if (i == 1 && cmd != "run") {
                std::cout << "Invalid Command!" << std::endl;
                return 0;
            } else if (i == 2 && cmd.substr(cmd.length() - 3, cmd.length()) != ".bc") {
                std::cout << "Only .bc files are compiled" << std::endl;
                return 0;
            }
        }

        std::string file_loc = argv[2];

        std::ifstream file;

        file.open(file_loc);

        std::string file_data;

        if (file.is_open()) {
            char mychar;
            while (file) {
                mychar = file.get();
                file_data += BCUtils::parse_string(mychar);
            }
```

```
    }

    std::string s = file_data;

    auto *state_machine = new BC_FSM();

    state_machine->build_state_machine();
    auto *response = state_machine->validate_expression(&s);

    if (response->status == 1) {
        Evaluator::sanitizeExpression(&s);
        auto *root = Evaluator::buildExpressionTree(&s);
        int val = Evaluator::evaluateTree(root);
        std::cout << val << std::endl;
    } else {
        std::cout << "Incorrect Syntax!" << std::endl;
        std::cout << "Error: " << response->error << std::endl;
    }

    return 0;
}
```

node.cpp

```
#include "Node.h"
#include "string"

Node::Node() {
    this->data = "";
    this->left = nullptr;
    this->right = nullptr;
}

   Node::Node(std::string data) {
       this->data = data;
       this->left = nullptr;
       this->right = nullptr;
   }

   void Node::setData(std::string nodeValue) { this->data = nodeValue;
}

   std::string Node::getData() const { return data; }

   void Node::setLeft(Node *leftSubTree) { this->left = leftSubTree; }

   Node *Node::getLeft() { return left; }

   void Node::setRight(Node *rightSubTree) { this->right =
rightSubTree; }

   Node *Node::getRight() { return right; }

```

node.h

```cpp
#ifndef BC_COMPILER_NODE_H
#define BC_COMPILER_NODE_H

#include "string"

class Node {
    std::string data;
    Node *left;
    Node *right;

  public:
    Node();

    explicit Node(std::string data);

    void setData(std::string nodeValue);

    std::string getData() const;

    void setLeft(Node *leftSubTree);

    Node *getLeft();

    void setRight(Node *rightSubTree);

    Node *getRight();
  };

  #endif // BC_COMPILER_NODE_H

```

Evaluator.cpp

```
#include "Evaluator.h"
#include "Node.h"

#include <bits/stdc++.h>

Node *Evaluator::buildExpressionTree(std::string *s) {
    /*
     *  A stack to save the nodes, in order to
     *  compute and save the values of nodes
        *
        *  node: for the expression tree
        */
       std::stack<Node *> stackOfNodes;

       /*
        * A stack to save characters, parse the
        * expression and check its correctness.
        */
       std::stack<std::string> stackOfCharacter;

       /*
        * A map to store operators and their
        * corresponding priorities.
        */
       std::unordered_map<std::string, int> map;

       /*
        * define priorities of operators,
        * or operator precedence.
        */
       map[")"] = 0;

       map["+"] = map["-"] = 1;

       map["/"] = map["*"] = 2;

       map["^"] = 3;

       std::string num;
```

```
        /*
         * root, left and right nodes,
         * for forming the tree.
         */
        Node *root = nullptr, *left = nullptr, *right = nullptr;

        std::string str = *s;
        for (int i = 0; i < s->length(); i++) {
            std::string currentValue = str.substr(i, 1);

            num = currentValue;

            // to skip spaces
            if (currentValue == " ")
                continue;

            // starting of a expression/ subexpression
            if (currentValue == "(")
                stackOfCharacter.push(currentValue);

            // add the root node of tree to stack of nodes
            else if (!num.empty() && BCUtils::is_number(&num)) {

                // gather the entire number before moving on
                while (i + 1 < s->length() &&
BCUtils::is_digit(&currentValue)) {
                    std::string next_digit = str.substr(i + 1, 1);

                    if (next_digit == " ") {
                        i++;
                        break;
                    }

                    if (BCUtils::is_digit(&next_digit)) {
                        num += next_digit;
                        i++;
                        currentValue = next_digit;
                    } else {
                        break;
```

```
                    }
                }

                root = new Node(num);
                stackOfNodes.push(root);
                num = "";
            }

            //
            else if (map[currentValue] > 0) {
                bool stackHasChars = !stackOfCharacter.empty();
                bool charStackTopIsNotOpen = stackOfCharacter.top() != "(";

                while (stackHasChars && charStackTopIsNotOpen &&
                      ((currentValue != "^" &&
                        map[stackOfCharacter.top()] >= map[currentValue]) ||
                       (currentValue == "^" &&
                        map[stackOfCharacter.top()] > map[currentValue]))) {

                    // the char at the top of chars stack will the root of new
                    // subtree.
                    root = new Node(stackOfCharacter.top());
                    stackOfCharacter.pop();

                    // the one at top of nodes stack will be the left child of that
                    // root, defined above.
                    right = stackOfNodes.top();
                    stackOfNodes.pop();

                    // similarly, the second top of nodes stack will be the right
                    // child root.
                    left = stackOfNodes.top();
                    stackOfNodes.pop();

```

```
                // set the left and right child's of root.
                root->setLeft(left);
                root->setRight(right);

                stackOfNodes.push(root);
            }

            stackOfCharacter.push(currentValue);
        } else if (currentValue == ")") {
            while (!stackOfCharacter.empty() && stackOfCharacter.top() != "(") {
                root = new Node(stackOfCharacter.top());
                stackOfCharacter.pop();

                // the one at top of nodes stack will be the left child of that
                // root, defined above.
                right = stackOfNodes.top();
                stackOfNodes.pop();

                // similarly, the second top of nodes stack will be the right
                // child root.
                left = stackOfNodes.top();
                stackOfNodes.pop();

                // set the left and right child's of root.
                root->setLeft(left);
                root->setRight(right);

                stackOfNodes.push(root);
            }

            stackOfCharacter.pop();
        }
    }

    root = stackOfNodes.top();
    return root;
}
```

```cpp

int Evaluator::evaluateTree(Node *root) {
    if (!root)
        return 0;

    if (!root->getLeft() && !root->getRight()) {
        std::string val = root->getData();
        return BCUtils::parseInt(&val);
    }

    int leftValue = evaluateTree(root->getLeft());
    int rightValue = evaluateTree(root->getRight());

    if (root->getData() == "+")
        return leftValue + rightValue;
    else if (root->getData() == "-")
        return leftValue - rightValue;
    else if (root->getData() == "*")
        return leftValue * rightValue;
    else
        return leftValue / rightValue;
}

void Evaluator::sanitizeExpression(std::string *expression) {
    // more sanitizations checks to be added
    *expression = "(" + *expression;
    *expression += ")";
}

```

Evaluator.h

```
#ifndef BC_COMPILER_EVALUATOR_H
#define BC_COMPILER_EVALUATOR_H

#include "BCUtils.h"
#include "Node.h"

#include "string"

class Evaluator {
    public:
      static Node *buildExpressionTree(std::string *s);
      static int evaluateTree(Node *root);
      static void sanitizeExpression(std::string *expression);
  };

  #endif // BC_COMPILER_EVALUATOR_H

```

BCUtils.h

```
#ifndef BC_COMPILER_BCUTILS_H
#define BC_COMPILER_BCUTILS_H

#include <string>

class BCUtils {
  public:
    static bool is_digit(std::string *x);
       static bool is_number(std::string *numberI);
       static int parseInt(std::string *s);
       static std::string get_char_at(std::string *s, int i);
       static bool is_alpha(std::string *s);
       static bool is_operator(std::string *s);
       static std::string parse_string(char x);
       static bool is_space(std::string *s);
   };

   #endif // BC_COMPILER_BCUTILS_H
```

BCUtils.cpp

```cpp
#include "BCUtils.h"

#include <string>

bool BCUtils::is_digit(std::string *x) {
    std::string digit = *x;

    bool result = digit == "0" || digit == "1" || digit == "2" ||
                  digit == "3" || digit == "4" || digit == "5" ||
                  digit == "6" || digit == "7" || digit == "8" || digit == "9";

    return result;
}

bool BCUtils::is_number(std::string *numberI) {
    std::string number = *numberI;
    for (int i = 0; i < number.length(); i++) {
        std::string digit = number.substr(i, 1);
        if (!is_digit(&digit))
            return false;
    }
    return true;
}

int BCUtils::parseInt(std::string *s) {
    int num = 0;
    std::string str = *s;

    if (str[0] != '-') {
        for (char i: str) {
            num = num * 10 + (int(i) - 48);
        }
    } else {
        for (char i: str) {
            num = num * 10 + (int(i) - 48);
            num = num * -1;
        }
    }
```

```cpp

        return num;
    }

    std::string BCUtils::get_char_at(std::string *s, int i) {
        return (*s).substr(i, 1);
    }

    bool BCUtils::is_alpha(std::string *s) {
        return std::isalpha((*s)[0]);
    }

    bool BCUtils::is_operator(std::string *s) {
        char op = (*s)[0];
        return (
                op == '+' || op == '-' || op == '*' || op == '/' || op == '^'
        );
    }

    std::string BCUtils::parse_string(char x) {
        std::string s(1, x);
        return s;
    }

    bool BCUtils::is_space(std::string *s) {
        return (*s) == " ";
    }

```

BC_State.cpp

```cpp
#include "BC_State.h"

BC_State::BC_State(std::string *state_name) {
    this->state_name = *state_name;
}

void BC_State::add_transition(BC_State *state, std::string *transition_type) {
    std::pair<BC_State *, std::string> state_and_transition (state, *transition_type);
    transitions.push_back(state_and_transition);
}

std::vector<std::pair<BC_State *, std::string>> BC_State::get_transitions() {
    return transitions;
}

```

BC_State.h

```

#ifndef BC_COMPILER_BC_STATE_H
#define BC_COMPILER_BC_STATE_H

#include <string>
#include <vector>
#include <utility>

class BC_State {

        /*
         * List of transitions from the current
         * state, stored in a vector of pairs having
         * memory address of state and the transition
         * type to that state
         */
        // [{state, }]
        std::vector<std::pair<BC_State *, std::string>> transitions;

    public:
        /*
         * Give name to a state to uniquely
         * identify a particular state.
         */
        std::string state_name;
        explicit BC_State(std::string *state_name);
        void add_transition(BC_State *state, std::string *transition_type);
        std::vector<std::pair<BC_State *, std::string>> get_transitions();
    };

    #endif // BC_COMPILER_BC_STATE_H

```

BC_FSM_ParsingResponse.cpp

```cpp

#include "BC_FSM_ParsingResponse.h"

BC_FSM_ParsingResponse::BC_FSM_ParsingResponse(int status, std::string *error) {
    this->status = status;
    this->error = *error;
}

```

BC_FSM_ParsingResponse.h

```cpp

#ifndef BC_COMPILER_BC_FSM_PARSINGRESPONSE_H
#define BC_COMPILER_BC_FSM_PARSINGRESPONSE_H

#include <string>

class BC_FSM_ParsingResponse {

public:
        // status -> 1: OK
        // status -> 2: ERROR

        int status;
        std::string error;

        BC_FSM_ParsingResponse(int status, std::string *error);
    };

    #endif //BC_COMPILER_BC_FSM_PARSINGRESPONSE_H

```

BC_FSM.h

```

#ifndef BC_COMPILER_BC_FSM_H
#define BC_COMPILER_BC_FSM_H

#include "BC_State.h"
#include "Node.h"
#include "BC_FSM_ParsingResponse.h"
#include <string>
#include <vector>

    class BC_FSM {
        std::vector<BC_State *> states;
        std::vector<std::string> state_names;
        std::vector<std::string> transition_types;
        int builder_state_count;
        BC_State **global_current_state{};

    private:
        std::string *get_state_name();

        std::string *get_transition_type(int i);

    public:
        BC_FSM();

        void build_state_machine();

        BC_FSM_ParsingResponse *validate_expression(std::string *syntax);
    };

    #endif // BC_COMPILER_BC_FSM_H

```

BC_FSM.cpp

```cpp

#include "BC_FSM.h"
#include "BCUtils.h"

BC_FSM::BC_FSM() {
    // initialize starting of state machine
    builder_state_count = 0;

    // possible states in our syntax fsm
        state_names = {
                "q0",
                "q1",
                "q2",
                "q3",
                "q4",
                "q5",
                "q6",
                "q7",
                "q8",
                "q9",
                "q10",
                "ISE", // Invalid Syntax Error
                "IOE"  // Invalid Operation Error
        };

        transition_types = {
                "s",            // 0
                "o",            // 1
                "l",            // 2
                "v",            // 3
                "e",            // 4
                "<space>",      // 5
                "<number>",     // 6
                "<operator>",   // 7
                "<any>"         // 8
        };
   }

   std::string *BC_FSM::get_state_name() {
```

```cpp
    return &(state_names[builder_state_count++]);
}

std::string *BC_FSM::get_transition_type(int i) {
    return &(transition_types[i]);
}

void BC_FSM::build_state_machine() {
    auto *q0 = new BC_State(get_state_name());
    auto *q1 = new BC_State(get_state_name());
    auto *q2 = new BC_State(get_state_name());
    auto *q3 = new BC_State(get_state_name());
    auto *q4 = new BC_State(get_state_name());
    auto *q5 = new BC_State(get_state_name());
    auto *q6 = new BC_State(get_state_name());
    auto *q7 = new BC_State(get_state_name());
    auto *q8 = new BC_State(get_state_name());
    auto *q9 = new BC_State(get_state_name());
    auto *q10 = new BC_State(get_state_name());
    auto *ise = new BC_State(get_state_name());
    auto *ioe = new BC_State(get_state_name());

    // connect these states according to the state diagram

    // for 'solve' part
    q0->add_transition(q1, get_transition_type(0));
    q0->add_transition(ise, get_transition_type(8));

    q1->add_transition(q2, get_transition_type(1));
    q1->add_transition(ise, get_transition_type(8));

    q2->add_transition(q3, get_transition_type(2));
    q2->add_transition(ise, get_transition_type(8));

    q3->add_transition(q4, get_transition_type(3));
    q3->add_transition(ise, get_transition_type(8));

    q4->add_transition(q5, get_transition_type(4));
    q4->add_transition(ise, get_transition_type(8));

```

```
        q5->add_transition(q6, get_transition_type(5));
        q5->add_transition(ise, get_transition_type(8));

        // post 'solve' part
        q6->add_transition(q6, get_transition_type(5));
        q6->add_transition(q7, get_transition_type(6));
        q6->add_transition(q8, get_transition_type(7));

        q7->add_transition(q7, get_transition_type(6));
        q7->add_transition(q8, get_transition_type(7));
        q7->add_transition(q10, get_transition_type(5));

        q8->add_transition(q7, get_transition_type(6));
        q8->add_transition(q9, get_transition_type(5));
        q8->add_transition(ioe, get_transition_type(7));

        q9->add_transition(q7, get_transition_type(6));
        q9->add_transition(ioe, get_transition_type(7));

        q10->add_transition(q8, get_transition_type(7));
        q10->add_transition(ioe, get_transition_type(6));

        states.push_back(q0);
        states.push_back(q1);
        states.push_back(q2);
        states.push_back(q3);
        states.push_back(q4);
        states.push_back(q5);
        states.push_back(q6);
        states.push_back(q7);
        states.push_back(q8);
        states.push_back(q9);
        states.push_back(q10);
        states.push_back(ise);
        states.push_back(ioe);
    }

    BC_FSM_ParsingResponse *BC_FSM::validate_expression(std::string
*syntax) {
```

```
        // traversing the FSM state diagram like a directed weighted graph

        std::string error;

        if ((*syntax).substr(0, 5) != "solve") {
            error = "Invalid Syntax Error";
            return new BC_FSM_ParsingResponse(2, &error);
        }

        global_current_state = (&states[0]);

        // test case: "solve 4 + 11 * 2"
        for (int i = 0; i < (*syntax).length(); i++) {
            std::string current_value = BCUtils::get_char_at(syntax, i);

            if ((**global_current_state).state_name == "IOE") {
                error = "Invalid Operation Error";
                return new BC_FSM_ParsingResponse(2, &error);
            }

            std::vector<std::pair<BC_State *, std::string>> current_transitions = (**global_current_state).get_transitions();
            std::string state_name = (**global_current_state).state_name;

            for (int j = 0; j < current_transitions.size(); j++) {
                std::pair<BC_State *, std::string> current_transition = current_transitions[j];

                std::string transition = current_transition.second;

                bool is_alpha_and_equal = (BCUtils::is_alpha(&current_value) && current_value == transition);
                bool is_space_and_transition = (BCUtils::is_space(&current_value) && transition == "<space>");
                bool is_number_and_transition = (BCUtils::is_number(&current_value) && transition == "<number>");
                bool is_operator_and_transition = (BCUtils::is_operator(&current_value) && transition == "<operator>");
```

```
                if (is_alpha_and_equal || is_space_and_transition || is_number_and_transition ||
                    is_operator_and_transition) {
                    global_current_state = &(current_transition.first);
                    j = current_transitions.size();
                }
            }
        }

        return new BC_FSM_ParsingResponse(1, &error);
    }
```

REFRENCES

https://www.geeksforgeeks.org/introduction-of-finite-automata/

https://en.wikipedia.org/wiki/Finite-state_machine

https://www.javatpoint.com/finite-state-machine

https://www.geeksforgeeks.org/expression-tree/

https://visualstudio.microsoft.com/vs/features/cplusplus/

https://cplusplus.com/