

INFO400 PROJECT

Submitted in fulfillment of the requirements for the

COMPUTER SCIENCE DEGREE FROM THE LEBANESE UNIVERSITY
FACULTY OF SCIENCE – BRANCH -II-

Major:

Computer science

By:

BOU SLEIMAN Anthony 57238
EL KHOURY Christia 56916

Title

Automaton

Advisor:

Dr. COSTANTIN Joseph

Contents

I.	Introduction	3
II.	Automaton Implementation	3
1.	Add Automaton.....	5
2.	Search Automaton by ID	7
3.	Delete Automaton by ID	8
III.	Lexical Analysis.....	8
IV.	Grammar Analysis	11
V.	Table of Figures.....	23

I. Introduction

The project is about exploring the use of an automaton to tokenize programs, validate them against a predetermined grammar, and conduct language translation using semantic analysis. By combining these ideas, you will know how programming languages are processed and modified, connecting theoretical foundations to practical applications in language design and compiler development.

II. Automaton Implementation

In this part, we will show how to add, search for, and delete several automata. The main function contains a menu allowing users to select the task they want to perform. Using a vector for automata allows for the dynamic insertion of as many automata as needed, resulting in flexibility.

```
#include <iostream>
#include <vector>
#include <array>
using namespace std;

struct Transition {
    char origin;
    char label;
    char destination;
};

struct Automaton {
    int ID;
    char* alphabet;
    char* states;
    char q0;
    Transition* delta;
    char* terminalStates;
    int alphabet_size;
    int states_size;
    int delta_size;
    int terminal_size;
};

int main() {
    vector<Automaton> automaton;
    int action = 1;
    int id;
    int IdCounter = 1;

    while (action != 0) {
        cout << "To exit the program enter 0" << endl;
        cout << "To add automaton enter 1" << endl;
        cout << "To delete automaton by index enter 2" << endl;
        cout << "To search automaton by ID enter 3" << endl;
        cout << "Enter your choice: ";
        cin >> action;
        switch (action) {
            case 1: { // add a new Automaton to an array of structures
```

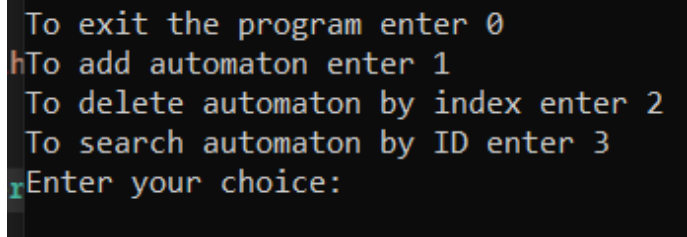
```

Automaton A;
A.ID = IdCounter++;
InsertAutomatonDetails(A);
automaton.push_back(A);
cout << "Automaton added with ID " << A.ID << endl;
cout << endl;
break;
}
case 2: { // delete an Automaton from the array of structures
    cout << "Enter the ID of the Automaton to delete" << endl;
    cin >> id;
    DeleteAutomaton(automaton, id);
    cout << endl;
    break;
}
case 3: { // search for an Automaton by introducing its ID
    cout << "Enter the ID of the Automaton to search for: ";
    cin >> id;
    SearchAutomaton(automaton, id);
    cout << endl;
    break;
}
case 0:
    cout << "Exiting program..." << endl;
    cout << endl;
    break;
default:
    cout << "Invalid choice, please try again!" << endl;
    cout << endl;
}
}

for (int i = 0; i < automaton.size(); ++i) {
    delete[] automaton[i].states;
    delete[] automaton[i].alphabet;
    delete[] automaton[i].delta;
}

return 0;
}

```



```

To exit the program enter 0
To add automaton enter 1
To delete automaton by index enter 2
To search automaton by ID enter 3
Enter your choice:

```

Figure 1 Menu of operations

1. Add Automaton

This function, InsertAutomatonDetails, takes information from the user about a single automaton. It enables for the dynamic input of states, labels, transitions, and terminal states, guaranteeing that the data follows the automaton's rules. The function does validation checks to guarantee that:

- States: The set of defined states includes the initial and final states.
- Labels: Transition labels are part of the defined alphabet.
- Transitions: The origin, label, and destination of each transition are all valid and correspond to the automaton structure.

```
bool check_if_p_not_in_array(char p, char array[], int l) {
    for (int i = 0; i < l; i++)
        if (p == array[i]) return true;
    return false;
}

void InsertAutomatonDetails(Automaton& A) {
    int n1, n2, n3, n4;
    int p;
    do {
        cout << "Enter the number of states: ";
        cin >> n2;
    } while (n2 <= 0);

    A.states_size = n2;
    A.states = new char[n2];
    cout << "Input the states" << endl;
    for (int i = 0; i < n2; i++)
        cin >> A.states[i];

    do {
        cout << "introduce the initial state: ";
        cin >> A.q0;
    } while (!check_if_p_not_in_array(A.q0, A.states, n2));

    do {
        cout << "enter the nbr of labels: ";
        cin >> n1;
    } while (n1 <= 0);

    A.alphabet_size = n1;
    A.alphabet = new char[n1];
    cout << "input the labels " << endl;
    for (int i = 0; i < n1; i++)
        cin >> A.alphabet[i];

    do {
        cout << "enter the nbr of transitions: ";
        cin >> n3;
    } while (n3 <= 0);

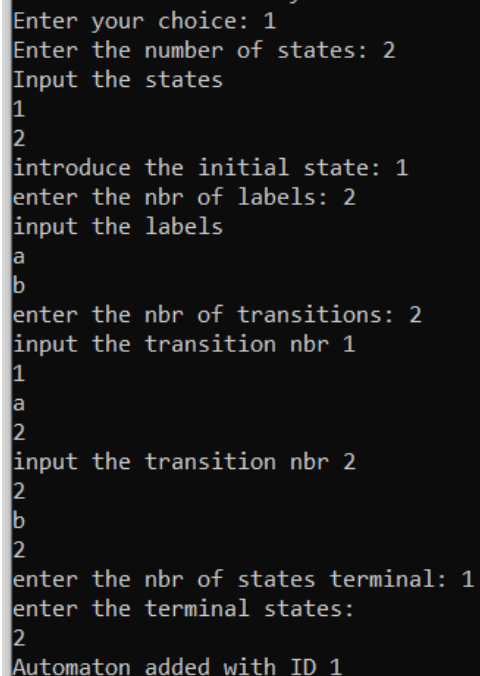
    A.delta_size = n3;
```

```

A.delta = new Transition[n3];
for (int i = 0; i < n3;i++) {
    do {
        cout << "input the transition nbr " << i + 1 << endl;
        cin >> A.delta[i].origin;
        cin >> A.delta[i].label;
        cin >> A.delta[i].destination;
        p = (!check_if_p_not_in_array(A.delta[i].origin, A.states, n2) ||
            !check_if_p_not_in_array(A.delta[i].label, A.alphabet, n1) ||
            !check_if_p_not_in_array(A.delta[i].destination, A.states, n2));
    } while (p);
}

do {
    cout << "enter the nbr of states terminal: ";
    cin >> n4;
} while (n4 <= 0);
A.terminal_size = n4;
A.terminalStates = new char[n4];
cout << "enter the terminal states: " << endl;
for (int i = 0; i < n4;i++) {
    do {
        cin >> A.terminalStates[i];
    } while (!check_if_p_not_in_array(A.terminalStates[i], A.states, n2));
}
}

```



```

Enter your choice: 1
Enter the number of states: 2
Input the states
1
2
introduce the initial state: 1
enter the nbr of labels: 2
input the labels
a
b
enter the nbr of transitions: 2
input the transition nbr 1
1
a
2
input the transition nbr 2
2
b
2
enter the nbr of states terminal: 1
enter the terminal states:
2
Automaton added with ID 1

```

Figure 2 Add an automaton with its details

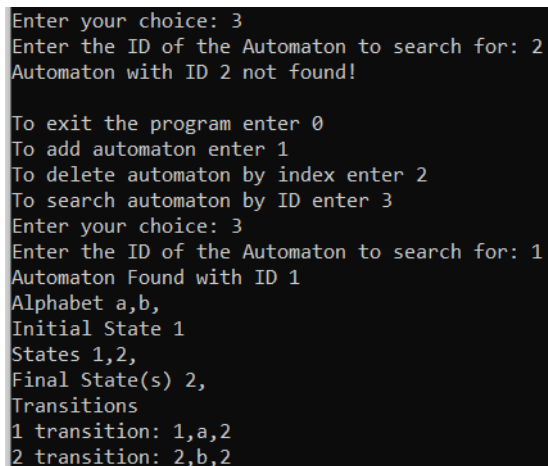
2. Search Automaton by ID

This function, SearchAutomaton, looks for an automaton in a vector of automata using its unique ID. If an automaton with the specific ID is located, its details are displayed, such as the alphabet, beginning state, states, terminal states, and transitions. If no matching automata is identified, a message will be displayed informing that the automaton was not found.

```
void print_transactions(Transition* delta,int size) {
    for (int i = 0; i < size; i++) {
        cout << i + 1 << " transition: " << delta[i].origin << "," << delta[i].label
        << "," << delta[i].destination << endl;}}

void print_array_element(char array[], int size) {
    for (int i = 0; i < size; i++) {
        cout << array[i] << ",";
    }
    cout << endl;
}

void SearchAutomaton(vector<Automaton>& automaton, int id) {
    for (int i = 0; i < automaton.size(); ++i) {
        if (automaton[i].ID == id) {
            cout << "Automaton Found with ID " << id << endl;
            cout << "Alphabet ";
            print_array_element(automaton[i].alphabet, automaton[i].alphabet_size);
            cout << "Initial State "<<automaton[i].q0 << endl;
            cout << "States ";
            print_array_element(automaton[i].states, automaton[i].states_size);
            cout << "Final State(s) ";
            print_array_element(automaton[i].terminalStates,
            automaton[i].terminal_size);
            cout << "Transitions " << endl;
            print_transactions(automaton[i].delta, automaton[i].delta_size);
            return;
        }
    }
    cout << "Automaton with ID " << id << " not found!" << endl;}
```



```
Enter your choice: 3
Enter the ID of the Automaton to search for: 2
Automaton with ID 2 not found!

To exit the program enter 0
To add automaton enter 1
To delete automaton by index enter 2
To search automaton by ID enter 3
Enter your choice: 3
Enter the ID of the Automaton to search for: 1
Automaton Found with ID 1
Alphabet a,b,
Initial State 1
States 1,2,
Final State(s) 2,
Transitions
1 transition: 1,a,2
2 transition: 2,b,2
```

Figure 3 Search for automaton

3. Delete Automaton by ID

This function, DeleteAutomaton, looks for an automaton in a vector of automata using its unique ID. If an automaton with the specific ID is located, it is deleted by displaying a message that it is successfully deleted. If no matching automata is identified, a message will be displayed informing that the automaton was not found.

```
void DeleteAutomaton(vector<Automaton>& automaton, int id) {
    for (int i = 0; i < automaton.size(); ++i) {
        if (automaton[i].ID == id) {
            automaton.erase(automaton.begin()+i);
            cout << "Automaton Deleted with ID " << id << endl;
            return;
        }
    }
    cout << "Automaton with ID " << id << " not found!" << endl;
}
```

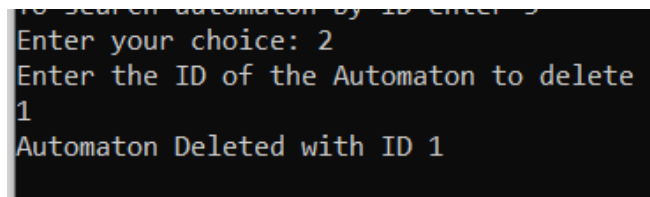


Figure 4 Delete Automaton

III. Lexical Analysis

This program processes an input string character by character, transforming it into tokens based on specified categories.

- State 1 identifies operators(+, -, =, /, *)
- State 2 identifies identifiers (alphanumeric characters).
- State 3 identifies integers (numeric digits).
- State 4 identifies separators (;, (,), :, @).
- State 5 identifies doubles (decimal numbers).

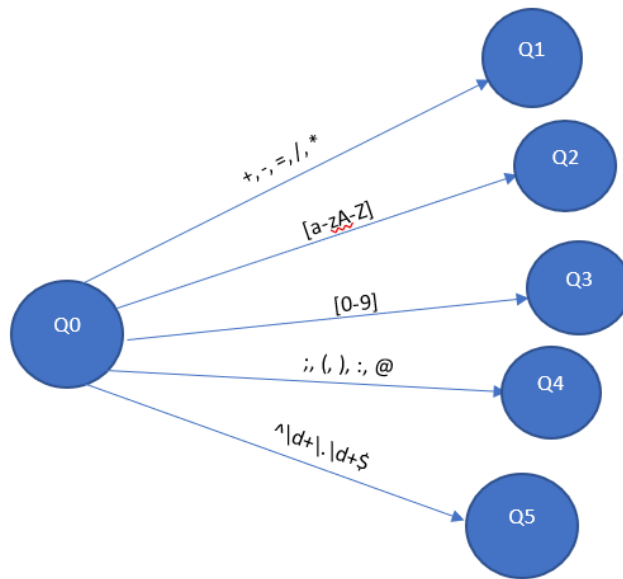


Figure 5 Lexical Analysis

```

#include <iostream>
#include <cctype>
using namespace std;

char UL[80];
char word[80];
char* car;
char* p1;

const char* LA();
void copie(char*, char*);

int main() {
    cout << "Input word: ";
    cin >> word;
    car = word;

    while (*car != '\0') {
        cout << LA() << ':';
        cout << UL << ' ' << endl;
    }

    cout << "\nWord is in the language";
    system("pause");
    return 0;
}

const char* LA() {
    int state = 0;
    while (1) {

```

```

switch (state) {
case 0:
    if (*car == '*' || *car == '+' || *car == '-' || *car == '/' || *car ==
'=') {
        p1 = car;
        state = 1;
        break;
    }
    else if (isalpha(*car)) {
        p1 = car;
        state = 2;
        break;
    }
    else if (isdigit(*car)) {
        p1 = car;
        state = 3;
        break;
    }
    else if (*car == ';' || *car == '(' || *car == ')' || *car == ':' || *car
== '@') {
        p1 = car;
        state = 4;
        break;
    }
    else {
        cout << "Lexical error";
        exit(1);
    }
case 1:
    copie(p1, car);
    return "OPERATOR";
case 2:
    if (isalpha(*car)) {
        break;
    }
    else {
        copie(p1, car);
        return "ID";
    }
case 3:
    if (isdigit(*car)) {
        break;
    }
    else if (*car == '.') {
        state = 5;
        break;
    }
    else {
        copie(p1, car);
        return "INTEGER";
    }
case 4:
    copie(p1, car);
    return "SEPERATOR";
case 5:

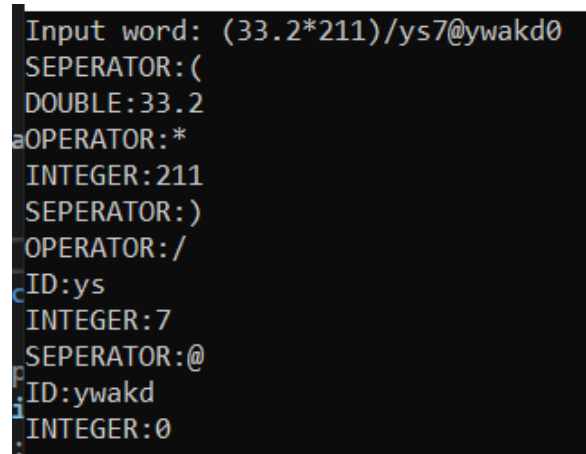
```

```

        if (isdigit(*car)) {
            break;
        }
        else {
            copie(p1, car);
            return "DOUBLE";
        }
    }
    ++car;
}

void copie(char* p, char* car) {
    int i = 0;
    while (p < car) {
        UL[i] = *p;
        p++;
        i++;
    }
    UL[i] = '\0';
}

```



```

Input word: (33.2*211)/ys7@ywakd0
SEPERATOR:(
DOUBLE:33.2
OPERATOR:*
INTEGER:211
SEPERATOR:)
OPERATOR:/
ID:ys
INTEGER:7
SEPERATOR:@
ID:ywakd
INTEGER:0

```

Figure 6 Exemple of lexical analysis

IV. Grammar Analysis

Grammar is:

```

program --> statementList
statementList --> A
A --> eps || ID = exp; A
exp --> term B
B --> + term B || - term B || eps

```

term --> factor C
C --> * factor C || / factor C || eps
factor --> integer || (exp)

There are main functions that are required in the grammar analysis:

```
void accept(const char* symbol, const char* type) {
    const char* token;
    token = la();
    if (!strcmp(type, "sep")) {
        if (!strcmp(lu, symbol))
            return;
        else {
            cout << "Syntax error3.\n"; exit(1);
        }
    }
    else if (!strcmp(type, "integer")) {
        if (!strcmp(token, "integer"))
            return;
        else {
            cout << "Syntax error.\n"; exit(1);
        }
    }
    else if (!strcmp(type, "ID")) {
        if (!strcmp(token, "ID")) {
            if (!strcmp(lu, symbol))
                return;
        }
        else
            cout << "Syntax error1.\n"; exit(1);
    }
    else {
        cout << "Syntax error2.\n"; exit(1);
    }
}
```

The accept() function compares the expected token (based on the grammar rules) with the present token in the input stream, which is what the lexical analyzer or parser is pointing to. The function indicates a syntax error and ends the program if they don't match, but if they do, it permits the processing to proceed. It compares the token's value with the expected symbol and its type with the expected type.

```

void returns() {
    int i;
    for (i = 0; i < strlen(lu); i++)
        car--;
}

```

The parser uses the returns function, a utility function, to move the input pointer by a predetermined amount of characters. This is usually done to "undo" or "rewind" the reading of a token.

AP function will let me start the grammar analysis.

```

void AP() {
    // start of program
    root = new Node;
    nd = root;
    strcpy(nd->label, "program");
    program();
    // end of word $
    accept("$", "sep");
}

```

We start with the first production which is program and from there we go to statementList.

```

void program() {
    // program --> statementList
    strcpy(nd->p, "p1");
    nd->child1 = new Node;
    strcpy(nd->child1->label, "statementList");
    nd->child2 = NULL;
    nd->child3 = NULL;
    nd->child4 = NULL;
    nd->child5 = NULL;
    nd->child6 = NULL;
    nd->child7 = NULL;
    nd = nd->child1;
    statementList();
}

```

From statementlist we can go to production A. It only has one child.

```

void statementList() {
    // statementList --> A
    Node* cur;
    strcpy(nd->p, "p2");
    nd->child1 = new Node;
    strcpy(nd->child1->label, "A");
    nd->child2 = NULL;
    nd->child3 = NULL;
    nd->child4 = NULL;
    nd->child5 = NULL;
}

```

```

    nd->child6 = NULL;
    nd->child7 = NULL;
    nd = nd->child1;
    A();
}

```

For the production A, it has 2 children:

```

void A() {
    // A --> eps
    // -->ID = exp ; A
    Node* cur;
    if (*car == '$') {
        nd->child1 = new Node;
        // peps means production for epsilon
        strcpy(nd->p, "peps");
        strcpy(nd->child1->label, "eps");
        strcpy(nd->child1->p, "peps");
        nd->child1->child1 = NULL;
        nd->child1->child2 = NULL;
        nd->child1->child3 = NULL;
        nd->child1->child4 = NULL;
        nd->child1->child5 = NULL;
        nd->child1->child6 = NULL;
        nd->child1->child7 = NULL;
        nd->child1->child8 = NULL;
        nd->child1->child9 = NULL;
        nd->child1->child10 = NULL;
        nd->child1->child11 = NULL;
        nd->child2 = NULL;
        nd->child3 = NULL;
        nd->child4 = NULL;
        nd->child5 = NULL;
        nd->child6 = NULL;
        nd->child7 = NULL;
        nd->child8 = NULL;
        nd->child9 = NULL;
        nd->child10 = NULL;
        nd->child11 = NULL;
        return;
    }
    const char* token = la(), * token1;
    if (!strcmp(token, "ID")) { //assignment
        strcpy(nd->p, "p4");
        nd->child1 = new Node;
        nd->child2 = new Node;
        nd->child3 = new Node;
        nd->child4 = new Node;
        strcpy(nd->child1->label, lu);
        nd->child1->child1 = NULL;
        nd->child1->child2 = NULL;
        nd->child1->child3 = NULL;
        nd->child1->child4 = NULL;
        nd->child1->child5 = NULL;
        nd->child1->child6 = NULL;
    }
}

```

```

        nd->child1->child7 = NULL;
        strcpy(nd->child2->label, "=");
        nd->child2->child1 = NULL;
        nd->child2->child2 = NULL;
        nd->child2->child3 = NULL;
        nd->child2->child4 = NULL;
        nd->child2->child5 = NULL;
        nd->child2->child6 = NULL;
        nd->child2->child7 = NULL;
        // to see if where i am in the grammar is equal to where i am in the
word
        accept("=", "sep");
        strcpy(nd->child3->label, "expression");
        cur = nd;
        // we changed the pointer to nd cause its a production/ node so we ccan
traverse it
        nd = nd->child3;
        expression();
        nd = cur;
        strcpy(nd->child4->label, ";");
        nd->child4->child1 = NULL;
        nd->child4->child2 = NULL;
        nd->child4->child3 = NULL;
        nd->child4->child4 = NULL;
        nd->child4->child5 = NULL;
        nd->child4->child6 = NULL;
        nd->child4->child7 = NULL;
        accept(";", "sep");
        nd->child5 = new Node;
        strcpy(nd->child5->label, "A");
        cur = nd;
        nd = nd->child5;
        A();
        nd = cur;
        nd->child6 = NULL;
        nd->child7 = NULL;
        return;
    }

    // If it's not "ID", it means we need to backtrack (invalid case)
    returns();
    nd->child1 = new Node;
    strcpy(nd->child1->label, "eps");
    strcpy(nd->p, "peps");
    nd->child1->child1 = NULL;
    nd->child1->child2 = NULL;
    nd->child1->child3 = NULL;
    nd->child1->child4 = NULL;
    nd->child1->child6 = NULL;
    nd->child1->child5 = NULL;
    nd->child1->child7 = NULL;
    nd->child2 = NULL;
    nd->child3 = NULL;
    nd->child4 = NULL;
    nd->child5 = NULL;

```

```

        nd->child6 = NULL;
        nd->child7 = NULL;
        returns();
    }

```

From production expression we can go to B:

```

void expression() {
    // exp --> term B
    Node* cur;
    nd->child1 = new Node;
    nd->child2 = new Node;
    strcpy(nd->p, "p9");
    strcpy(nd->child2->label, "B");
    strcpy(nd->child1->label, "term");
    nd->child3 = NULL;
    nd->child4 = NULL;
    nd->child5 = NULL;
    nd->child6 = NULL;
    nd->child7 = NULL;
    cur = nd;
    nd = nd->child1;
    term();
    nd = cur;
    nd = nd->child2;
    B();
}

```

Production B has 3 children.

```

void B() {
    //B --> + term B
    // --> - term B
    // --> eps
    Node* cur;
    la();
    nd->child1 = new Node;
    nd->child4 = NULL;
    nd->child5 = NULL;
    nd->child6 = NULL;
    nd->child7 = NULL;
    if (!strcmp("-", lu)) {
        nd->child2 = new Node;
        nd->child3 = new Node;
        strcpy(nd->p, "p10");
        strcpy(nd->child1->label, "-");
        nd->child1->child1 = NULL;
        nd->child1->child2 = NULL;
        nd->child1->child3 = NULL;
        nd->child1->child4 = NULL;
        nd->child1->child5 = NULL;
        nd->child1->child6 = NULL;
        nd->child1->child7 = NULL;
    }
}

```



```

        strcpy(nd->child2->label, "term");
        cur = nd;
        nd = nd->child2;
        term();
        nd = cur;
        nd = nd->child3;
        strcpy(nd->child3->label, "B");
        B();
    }
    else if (!strcmp("+", lu)) {
        nd->child2 = new Node;
        nd->child3 = new Node;
        strcpy(nd->p, "p11");
        strcpy(nd->child1->label, "+");
        nd->child1->child1 = NULL;
        nd->child1->child2 = NULL;
        nd->child1->child3 = NULL;
        nd->child1->child4 = NULL;
        nd->child1->child5 = NULL;
        nd->child1->child6 = NULL;
        nd->child1->child7 = NULL;
        strcpy(nd->child2->label, "term");
        cur = nd;
        nd = nd->child2;
        term();
        nd = cur;
        strcpy(nd->child3->label, "B");
        nd = nd->child3;
        B();
    }
    else {
        returns();
        strcpy(nd->child1->label, "eps");
        nd->child1->child1 = NULL;
        nd->child1->child2 = NULL;
        nd->child1->child3 = NULL;
        nd->child1->child4 = NULL;
        nd->child1->child5 = NULL;
        nd->child1->child6 = NULL;
        nd->child1->child7 = NULL;
        strcpy(nd->p, "peps");
        nd->child2 = NULL;
        nd->child3 = NULL;
    }
}

```

Term has one child and we can go from it to factor and then to C

```

void term() {
    // term --> factor C
    Node* cur;
    strcpy(nd->p, "p13");
    nd->child1 = new Node;
    nd->child2 = new Node;
    nd->child3 = NULL;
}

```

```

nd->child4 = NULL;
nd->child5 = NULL;
nd->child6 = NULL;
nd->child7 = NULL;
strcpy(nd->child1->label, "factor");
strcpy(nd->child2->label, "C");
cur = nd;
nd = nd->child1;
factor();
nd = cur;
nd = nd->child2;
C();
}

```

This is production C:

```

void C() {
    // C --> * factor C
    // --> / factor C
    // --> eps
    Node* cur;
    la();
    nd->child1 = new Node;
    nd->child4 = NULL;
    nd->child5 = NULL;
    nd->child6 = NULL;
    nd->child7 = NULL;
    if (!strcmp("*", lu)) {
        nd->child2 = new Node;
        nd->child3 = new Node;
        strcpy(nd->p, "p14");
        strcpy(nd->child1->label, "*");
        nd->child1->child1 = NULL;
        nd->child1->child2 = NULL;
        nd->child1->child3 = NULL;
        nd->child1->child4 = NULL;
        nd->child1->child5 = NULL;
        nd->child1->child6 = NULL;
        nd->child1->child7 = NULL;
        strcpy(nd->child2->label, "factor");
        cur = nd;
        nd = nd->child2;
        factor();
        nd = cur;
        strcpy(nd->child3->label, "C");
        nd = nd->child3;
        C();
    }
    else if (!strcmp("/", lu)) {
        nd->child2 = new Node;
        nd->child3 = new Node;
        strcpy(nd->p, "p15");
        strcpy(nd->child1->label, "/");
        nd->child1->child1 = NULL;
        nd->child1->child2 = NULL;
    }
}

```

```

        nd->child1->child3 = NULL;
        nd->child1->child4 = NULL;
        nd->child1->child5 = NULL;
        nd->child1->child6 = NULL;
        nd->child1->child7 = NULL;
        strcpy(nd->child2->label, "factor");
        cur = nd;
        nd = nd->child2;
        factor();
        nd = cur;
        strcpy(nd->child3->label, "C");
        nd = nd->child3;
        C();
    }
    else {
        returns();
        strcpy(nd->child1->label, "eps");
        nd->child1->child1 = NULL;
        nd->child1->child2 = NULL;
        nd->child1->child3 = NULL;
        nd->child1->child4 = NULL;
        nd->child1->child5 = NULL;
        nd->child1->child6 = NULL;
        nd->child1->child7 = NULL;
        strcpy(nd->p, "peps");
        nd->child2 = NULL;
        nd->child3 = NULL;
    }
}

```

And this is production Factor from which we can accept integers or (exp)

```

void factor() {
    //factor --> integer
    // --> (exp)
    Node* cur;
    nd->child1 = new Node;
    nd->child1->child1 = NULL;
    nd->child1->child2 = NULL;
    nd->child1->child3 = NULL;
    nd->child1->child4 = NULL;
    nd->child1->child5 = NULL;
    nd->child1->child6 = NULL;
    nd->child1->child7 = NULL;
    const char* token = la();
    if (!strcmp(lu, "(")) {
        strcpy(nd->p, "p17");
        strcpy(nd->child1->label, "(");
        nd->child1->child1 = NULL;
        nd->child1->child2 = NULL;
        nd->child1->child3 = NULL;
        nd->child1->child4 = NULL;
        nd->child1->child5 = NULL;
        nd->child1->child6 = NULL;
        nd->child1->child7 = NULL;
    }
}

```

```

        nd->child2 = new Node;
        strcpy(nd->child1->label, "expression");
        cur = nd;
        nd = nd->child2;
        expression();
        nd = cur;
        accept(")", "sep");
        strcpy(nd->child3->label, ")");
        nd->child3 = new Node;
        nd->child3->child1 = NULL;
        nd->child3->child2 = NULL;
        nd->child3->child3 = NULL;
        nd->child3->child4 = NULL;
        nd->child3->child5 = NULL;
        nd->child3->child6 = NULL;
        nd->child3->child7 = NULL;
    }
    else{
        if (!strcmp(token, "integer"))
        {
            strcpy(nd->p, "p18");
            strcpy(nd->child1->label, lu);
            strcpy(nd->child1->p, "p20");
            nd->child1->child1 = NULL;
            nd->child1->child2 = NULL;
            nd->child1->child3 = NULL;
            nd->child1->child4 = NULL;
            nd->child1->child5 = NULL;
            nd->child1->child6 = NULL;
            nd->child1->child7 = NULL;
            nd->child2 = NULL;
            nd->child3 = NULL;
        }
    }
    nd->child4 = NULL;
    nd->child5 = NULL;
    nd->child6 = NULL;
    nd->child7 = NULL;
    return;
}

```

Here are some results of inputs with our grammar:

- X=5;

```

Microsoft Visual Studio Debug Console

\\\\\\\\\\\\\\\\PART 4\\\\\\\\\\\\\\\\n Input word to be transformed from cpp to c: x=5;
Word is in the language cpp

Printing the tree:
program->statementList
statementList->A
A->x=expression;A
expression->termB
term->factorC
factor->5
C->eps
B->eps
A->eps

C:\Users\PC\source\repos\I400_Project_Christia_Elkhoury_Anthony_BouSleiman\x64\Debug\I400_Project_Christia_Elkhoury_Anthony_BouSleiman.exe (process 7392) exited with code 0 (0x0).
Press any key to close this window . . .

```

Figure 7 result of acceptable grammar -1

- $Y=3+2;$

```

Microsoft Visual Studio Debug Console

\\\\\\\\\\\\\\\\PART 4\\\\\\\\\\\\\\\\n Input word to be transformed from cpp to c: y=3+2;
Word is in the language cpp

Printing the tree:
program->statementList
statementList->A
A->y=expression;A
expression->termB
term->factorC
factor->3
C->eps
B->+termB
term->factorC
factor->2
C->eps
B->eps
A->eps

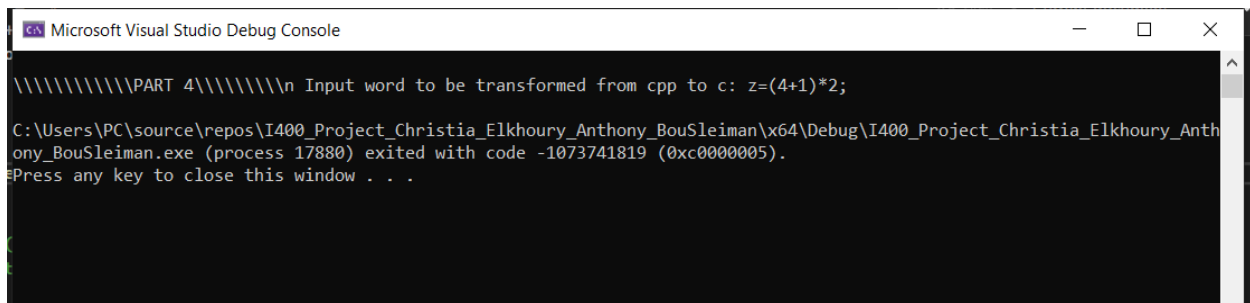
C:\Users\PC\source\repos\I400_Project_Christia_Elkhoury_Anthony_BouSleiman\x64\Debug\I400_Project_Christia_Elkhoury_Anthony_BouSleiman.exe (process 2148) exited with code 0 (0x0).
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .

```

Figure 8 result of acceptable grammar-2

- $Z=(4+1)*2;$

Here it returned epsilon.



```
Microsoft Visual Studio Debug Console

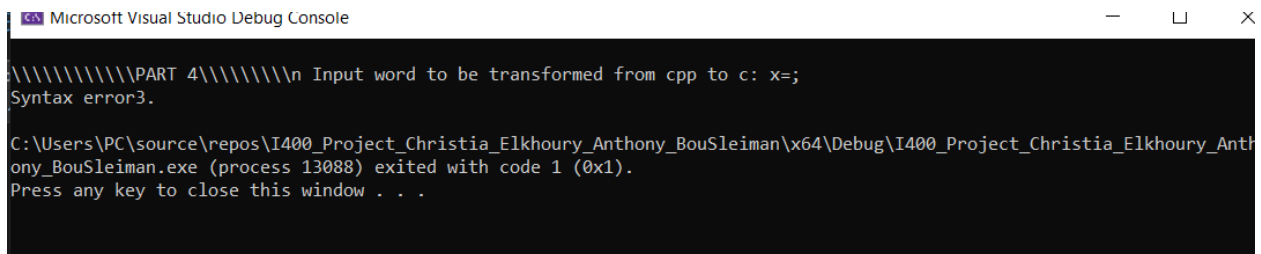
\\\\\\\\\\\\\\\\PART 4\\\\\\\\\\\\\\\\\\n Input word to be transformed from cpp to c: z=(4+1)*2;

C:\Users\PC\source\repos\I400_Project_Christia_Elkhoury_Anthony_BouSleiman\x64\Debug\I400_Project_Christia_Elkhoury_Anthony_BouSleiman.exe (process 17880) exited with code -1073741819 (0xc0000005).
Press any key to close this window . . .
```

Figure 9 result returning epsilon

- $X=;$

Which is not acceptable in our grammar.



```
Microsoft Visual Studio Debug Console

\\\\\\\\\\\\\\\\PART 4\\\\\\\\\\\\\\\\\\n Input word to be transformed from cpp to c: x=;
Syntax error3.

C:\Users\PC\source\repos\I400_Project_Christia_Elkhoury_Anthony_BouSleiman\x64\Debug\I400_Project_Christia_Elkhoury_Anthony_BouSleiman.exe (process 13088) exited with code 1 (0x1).
Press any key to close this window . . .
```

Figure 10 not acceptable input for our grammar

V. Table of Figures

Figure 1 Menu of operations

Figure 2 Add an automaton with its details

Figure 3 Search for automaton

Figure 4 Delete Automaton

Figure 5 Lexical Analysis

Figure 6 Exemple of lexical analysis

Figure 7 result of acceptable grammar -1

Figure 8 result of acceptable grammar-2

Figure 9 result returning epsilon

Figure 10 not acceptable input for our grammar