|  |
| --- |
| **PRACTICAL FILE**  **BE (CSE) 6th Semester** |
| **COMPILER DESIGN**  **Jan 2025 – May 2025** |
| **Submitted By:**  **Jatin Dhiman**  **Roll Number:**  **UE223047** |
| **Submitted To**  **Dr. Akashdeep**  **Computer Science and Engineering** |

****

**Computer Science and Engineering**

**University Institute of Engineering and Technology**

**Panjab University, Chandigarh – 160014, INDIA**

**2025**

**INDEX**

|  |  |  |  |
| --- | --- | --- | --- |
| **S. No.** | **Name of Practical/Program** | **Page No.s** | **Remarks** |
| 1 | Generate and analyze intermediate files during C/C++ compilation. |  |  |
| 2 | Build a lexical analyzer in C++ to identify keywords, variables, numbers, and operators. |  |  |
| 3 | Perform lexical analysis using the C++ regex library. |  |  |
| 4 | Explore Lex tool by implementing lexical analysis programs. |  |  |
| 5 | Implement Thompson’s construction to convert regex to NFA. |  |  |
| 6 | Develop a recursive descent parser for arithmetic expressions. |  |  |
| 7 | Compute First and Follow sets. |  |  |
| 8 | Build an LL(1) parser with FIRST/FOLLOW sets, parsing table, and input string parsing. |  |  |
| 9 | Implement a Shift-Reduce (LR) parser using a parsing table and grammar. |  |  |
| 10 | To implement an Operator Precedence Parser that builds leading/trailing sets, generates a precedence table, and validates expressions |  |  |
| 11 |  |  |  |
| 12 |  |  |  |
| 13 |  |  |  |
| 14 |  |  |  |
| 15 |  |  |  |
| 16 |  |  |  |
| 17 |  |  |  |
| 18 |  |  |  |
| 19 |  |  |  |
| 20 |  |  |  |
| 21 |  |  |  |
| 22 |  |  |  |
| 23 |  |  |  |
| 24 |  |  |  |

**Practical 1**

**Aim**

To generate and analyze the intermediate files generated during the compilation of a C/C++ program.

**Description**

* The compilation of a C/C++ program occurs in multiple stages, each generating intermediate files.
* These intermediate files help in understanding how the compiler processes the source code.
* The key intermediate files generated are:

1. **Preprocessed file (.i or .ii)**
   * Contains expanded macros and included headers.
   * Generated using the -E flag (e.g., g++ -E program.cpp -o program.i).
2. **Assembly file (.s)**
   * Contains assembly language code generated from the source code.
   * Created using the -S flag (e.g., g++ -S program.cpp -o program.s).
3. **Object file (.o or .obj)**
   * Contains machine code but is not yet executable.
   * Generated using the -c flag (e.g., g++ -c program.cpp -o program.o).
4. **Executable file (.exe or a.out)**
   * The final runnable program after linking all object files.
   * Created using the default compilation command (e.g., g++ program.o -o program).

* Analyzing these files helps in debugging, optimizing, and understanding the compilation process.

**Procedure**

**Sample cpp program**

#include <iostream>

using namespace std;

int main()

{

    int a = 3;

    int b = 4;

    int sum = a + b;

    cout << sum;

    return 0;

}

**Output**

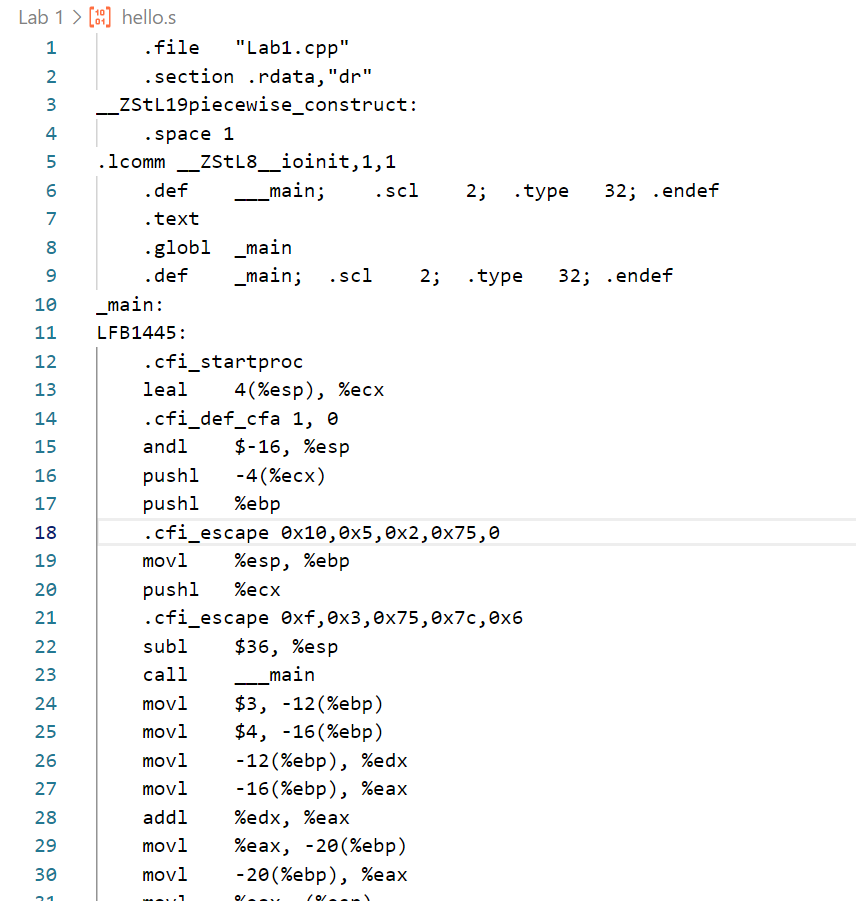
* **Preprocessed file**

Run the command “*gcc -E program.c -o program.i”* to generate the preprocessed file:



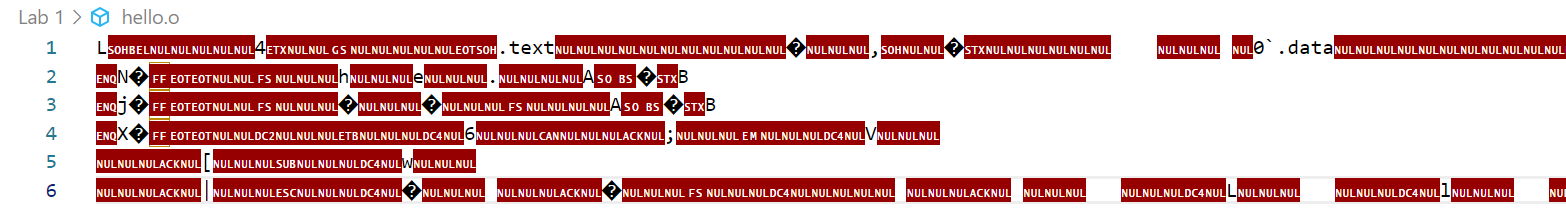
* **Assembly File**

*gcc -S program.c -o program.s*

**

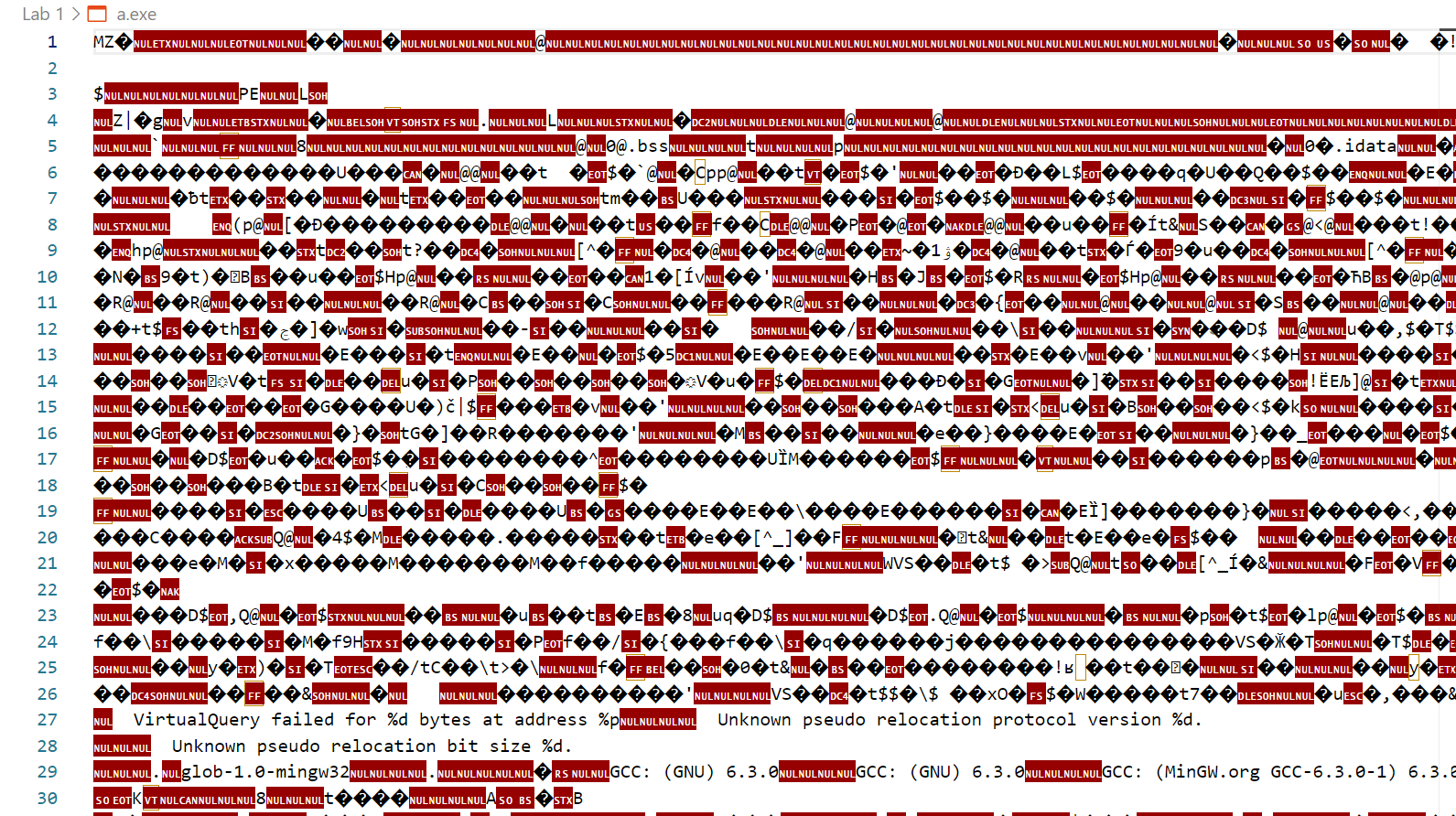
* **Object File**

*gcc -c program.c -o program.o*

**

* **Executable File**

*gcc program.o -o program*

**

**Learnings:**

* Gained an understanding of the different stages of the compilation process.
* Learned how to generate and analyze intermediate files.
* Discovered the role of object files and linking in producing an executable.

**Practical 2**

**Aim**

To develop a lexical analyzer in C++ that reads a C++ source file and identifies reserved keywords, variables, numbers, and operators.

**Description**

* Lexical analysis is the first phase of a compiler where the source code is converted into a sequence of tokens.
* The lexical analyzer (also known as a lexer or scanner) reads the source code character by character, grouping them into meaningful lexemes and classifying them as identifiers, keywords, operators, delimiters, literals, etc.
* This process helps in simplifying the parsing stage by converting complex code into a structured sequence of tokens.

**Procedure**

* Store operators, reserved keywords, and identified tokens.
* Read Input File: Open and read MyFile.txt line by line.
* Token Extraction:
  + Use two pointers (bp and fp) to extract words and symbols.
  + Check each token against predefined sets for classification.
* Categorization:
  + If a token matches a keyword, store it in reservedWords.
  + If it is a valid variable name or number, store it in vars\_nums.
  + If it is an operator, store it in operators\_.
* Print categorized tokens as Reserved Keywords, Variables/Numbers, and Operators.

#include <fstream>

#include <iostream>

#include <string>

#include <unordered\_map>

#include <unordered\_set>

using namespace std;

unordered\_set<char> operators = {'+', '-', '\*', '/', '<', '>', '=', '&', '|', ',', '(', ')', '[', ']', '{', '}', '#'};

unordered\_set<string> keywords = {"int", "return", "using", "namespace", "std", "include", "iostream", "main"};

unordered\_set<char> operators\_;

unordered\_set<string> reservedWords;

unordered\_set<string> vars\_nums;

bool isNumber(string word)

{

    for (int i = 0; i < word.size(); i++)

    {

        if (!(word[i] >= '0' && word[i] <= '9'))

        {

            return false;

        }

    }

    return true;

}

bool isVariable(string word)

{

    if ((word[0] >= '0' && word[0] <= '9'))

    {

        return false;

    }

    for (char &ch : word)

    {

        if (!((ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z') || (ch >= '0' && ch <= '9') || ch == '\_'))

        {

            return false;

        }

    }

    return true;

}

void print()

{

    cout << "Reserved Keywords:  ";

    for (auto &kw : reservedWords)

    {

        cout << kw << "  ";

    }

    cout << endl;

    cout << "\nVariables and Numbers:  ";

    for (auto &var : vars\_nums)

    {

        cout << var << "  ";

    }

    cout << endl;

    cout << "\nOperators:  ";

    for (auto op : operators\_)

    {

        cout << op << "  ";

    }

}

int main()

{

    fstream inFile("MyFile.txt", ios::in);

    string currLine;

    while (getline(inFile, currLine))

    {

        int bp = 0, fp = 0;

        while (fp <= currLine.size())

        {

            if (fp < currLine.size() && operators.find(currLine[fp]) == operators.end() && !(currLine[fp] == ' ' || currLine[fp] == '\t' || currLine[fp] == '\n' || currLine[fp] == ';'))

            {

                fp++;

            }

            else

            {

                string token = currLine.substr(bp, fp - bp);

                if (keywords.find(token) != keywords.end())

                {

                    reservedWords.insert(token);

                }

                else if (isNumber(token) || isVariable(token))

                {

                    vars\_nums.insert(token);

                }

                if (operators.find(currLine[fp]) != operators.end())

                {

                    operators\_.insert(currLine[fp]);

                }

                fp++;

                bp = fp;

            }

        }

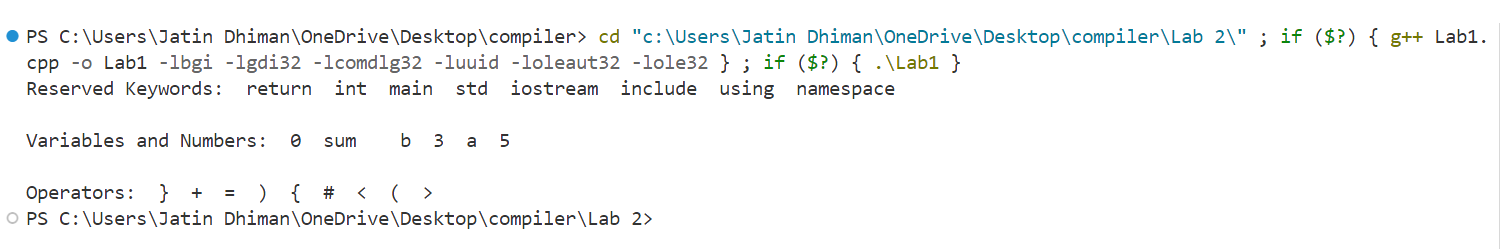
    }

    print();

    return 0;

}

**Output**

****

**Learnings:**

* Understanding of lexical analysis and tokenization in C++.
* String processing techniques for extracting and categorizing tokens.
* File handling in C++ to read and analyze source code files.
* Identifying and distinguishing between different types of tokens in a program.

**Practical 3**

**Aim**

Implementation of lexical analysis through regex library and creation of symbol table

**Description**

* The **regex library** is a powerful tool used for pattern matching and text manipulation in programming. It allows you to define search patterns to identify and process strings based on specific rules, such as matching keywords, operators, or identifiers in source code.
* In lexical analysis, regex is used to break down the input source code into tokens by recognizing different lexemes. A **symbol table** is a data structure that stores information about variables, functions, classes, and other identifiers in a program.
* It maps names to their attributes, such as type, scope, and memory location, which are crucial for semantic analysis, error checking, and code generation in the compilation process.
* Both the regex library and symbol table play vital roles in compiling and interpreting programming languages by organizing and processing source code efficiently

**Procedure**

* File Input: Open the source code file (hello.cpp) in read mode. Read the file content into a string for processing.
* Regular Expressions: Define regular expressions for token types: keywords, identifiers, constants, operators, relational operators, delimiters, and string literals.
* Tokenization:Use std::regex and iterators to match tokens in the source code.Classify each token based on its type using the defined regular expressions.
* Symbol Table Construction:Maintain a symbol table using unordered\_map to store:Token type**,**Length of the token**,**Occurrence count of the token
* Output:Print the source code to verify input ,Display the symbol table with detailed information about each token.
* Compilation and Execution: Compile the C++ program using a C++ compiler.Execute the program with the source code file to generate the symbol table.

**Code:**

#include <iostream>

#include <fstream>

#include <string>

#include <regex>

#include <unordered\_map>

#include <iomanip>

using namespace std;

regex keyword\_regex(R"(\b(auto|break|case|char|const|continue|default|do|double|else|enum|extern|"

                    "float|for|goto|if|inline|int|long|namespace|operator|private|protected|public|"

                    "register|return|short|signed|sizeof|static|struct|switch|template|this|throw|"

                    "try|typedef|union|unsigned|using|virtual|void|volatile|while)\b)");

regex identifier\_regex(R"([\_a-zA-Z][\_a-zA-Z0-9]\*)");

regex constant\_regex(R"(\d+)");

regex operator\_regex(R"([+\\*/%=<>!&|^\-]+|==|!=|<=|>=|<<|>>)");

regex relational\_operator\_regex(R"(\b(==|!=|<=|>=|<|>)\b)");

regex delimiter\_regex(R"([()\{\}\[\];,.:\?=])");

regex string\_literal\_regex(R"("([^"\\]|\\.)\*")");

unordered\_map<string, string> symbol\_table\_type;

unordered\_map<string, int> symbol\_table\_occurrence;

unordered\_map<string, int> symbol\_table\_length;

void lexical\_tokenize(const string &filename)

{

    ifstream file(filename);

    if (!file.is\_open())

    {

        cout << "Error: Could not open file " << filename << endl;

        return;

    }

    string code((istreambuf\_iterator<char>(file)), istreambuf\_iterator<char>());

    file.close();

    cout << "Source Code: " << endl;

    cout << code << endl;

    regex token\_regex(R"(\S+)");

    sregex\_iterator string\_begin(code.begin(), code.end(), string\_literal\_regex);

    sregex\_iterator string\_end;

    for (sregex\_iterator it = string\_begin; it != string\_end; ++it)

    {

        string token = it->str();

        symbol\_table\_type[token] = "string\_literal";

        symbol\_table\_occurrence[token]++;

        symbol\_table\_length[token] = token.length();

    }

    sregex\_iterator words\_begin(code.begin(), code.end(), token\_regex);

    sregex\_iterator words\_end;

    for (sregex\_iterator it = words\_begin; it != words\_end; ++it)

    {

        string token = it->str();

        if (regex\_match(token, string\_literal\_regex))

            continue;

        if (regex\_match(token, keyword\_regex))

        {

            symbol\_table\_type[token] = "keyword";

        }

        else if (regex\_match(token, identifier\_regex))

        {

            symbol\_table\_type[token] = "identifier";

        }

        else if (regex\_match(token, constant\_regex))

        {

            symbol\_table\_type[token] = "constant";

        }

        else if (regex\_match(token, relational\_operator\_regex))

        {

            symbol\_table\_type[token] = "operator";

        }

        else if (regex\_match(token, operator\_regex))

        {

            symbol\_table\_type[token] = "operator";

        }

        else if (regex\_match(token, delimiter\_regex))

        {

            symbol\_table\_type[token] = "delimiter";

        }

        symbol\_table\_occurrence[token]++;

        symbol\_table\_length[token] = token.length();

    }

}

void print\_symbol\_table()

{

    cout << left << setw(20) << "Token"

         << setw(20) << "Type"

         << setw(20) << "Length"

         << setw(15) << "Occurrence" << endl;

    for (const auto &pair : symbol\_table\_type)

    {

        cout << left << setw(20) << pair.first

             << setw(20) << pair.second

             << setw(20) << symbol\_table\_length[pair.first]

             << setw(20) << symbol\_table\_occurrence[pair.first] << endl;

    }

}

int main()

{

    string code\_file = "hello.cpp";

    lexical\_tokenize(code\_file);

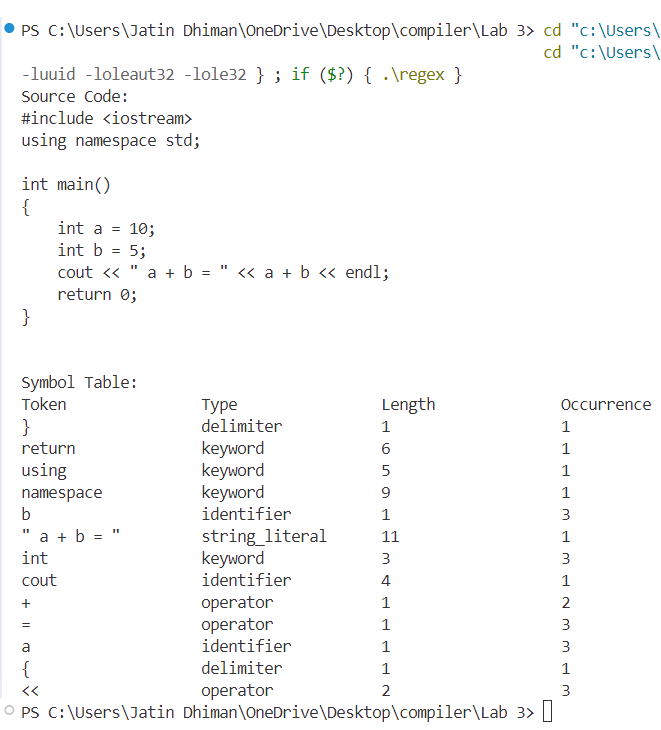
    cout << "\nSymbol Table:\n";

    print\_symbol\_table();

    return 0;

}

**Output:**

****

**Learnings:**

* Understanding lexical analysis and its role in the compilation process.
* Implementing a lexical analyzer using C++ and regular expressions.
* Identifying different types of tokens in a given source code.
* Constructing and managing a symbol table for token classification.
* Working with file handling and regex-based token matching in C++.

**Practical 4**

**Aim**

To understand the working of the **Lex tool** for lexical analysis by implementing various programs.

**Description**

Lex is a lexical analyzer that processes an input stream using regular expressions and pattern-matching rules. It helps in automating lexical analysis in compiler design and text processing applications.

This lab covers multiple exercises that include:

* Text transformation (e.g., converting uppercase to lowercase)
* Text processing (e.g., counting words, detecting numbers, handling comments)
* Code processing (e.g., adding line numbers, extracting arrays, removing comments)
* Token generation for programming languages

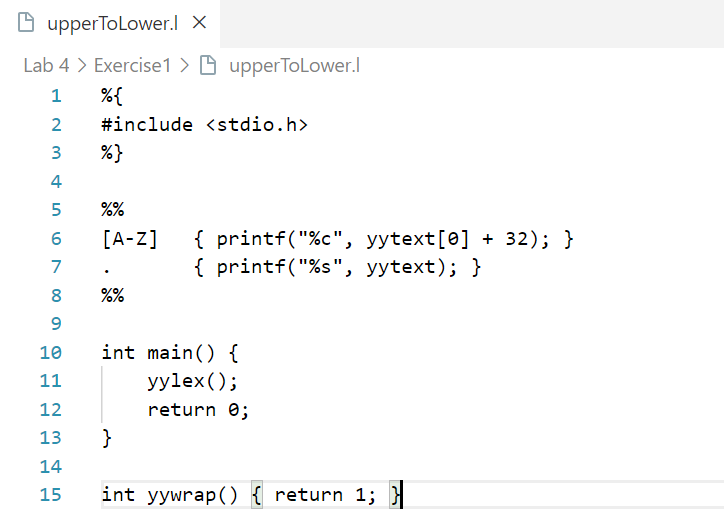
Each program demonstrates the power of regular expressions and Lex rules in processing and modifying input text effectively.

**Procedure**

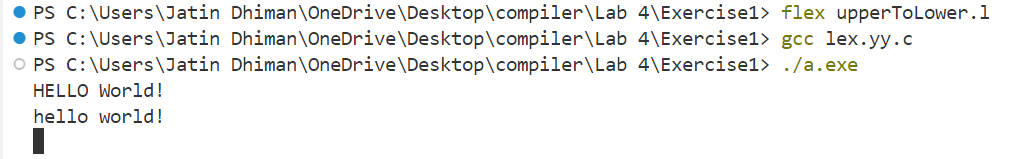
The general steps to execute a Lex program are as follows:

* Write the Lex program: Define the required patterns and actions.
* Save the file with a .l extension (e.g., program.l).
* Compile the Lex file using: **flex program.l**
* Generate the executable using: **gcc lex.yy.c**
* Run the program using: **./a.exe**

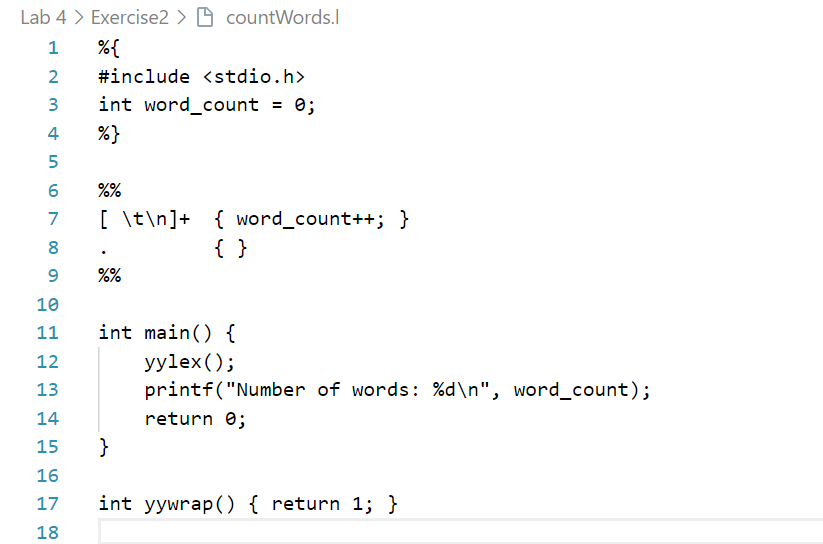
**Exercise 1:**

****

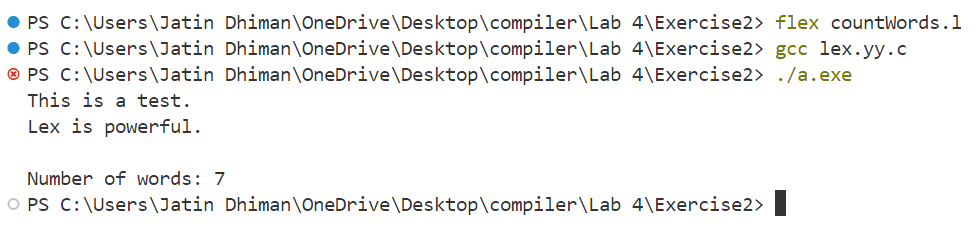
**Output:**



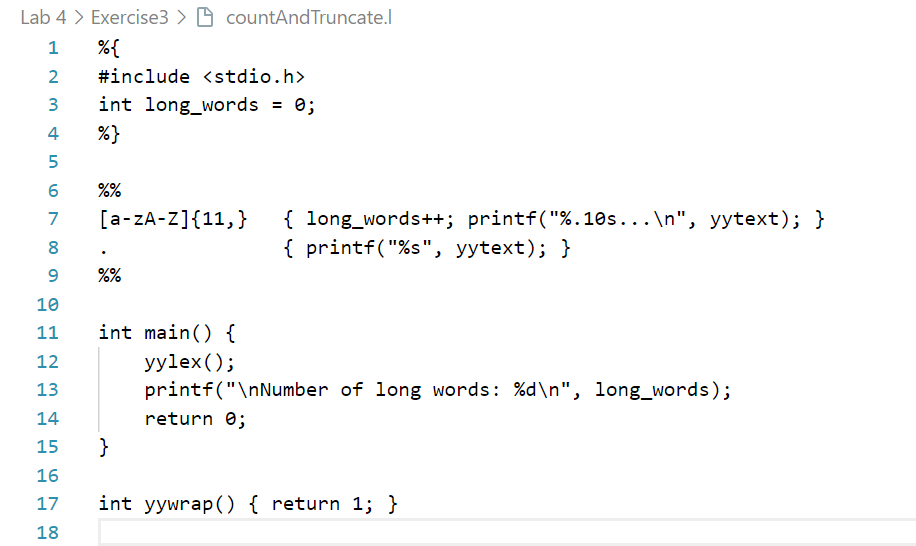
**Exercise 2:**

****

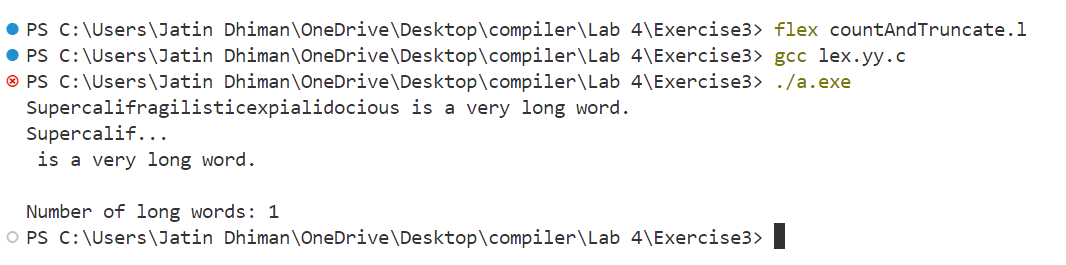
**Output:**

****

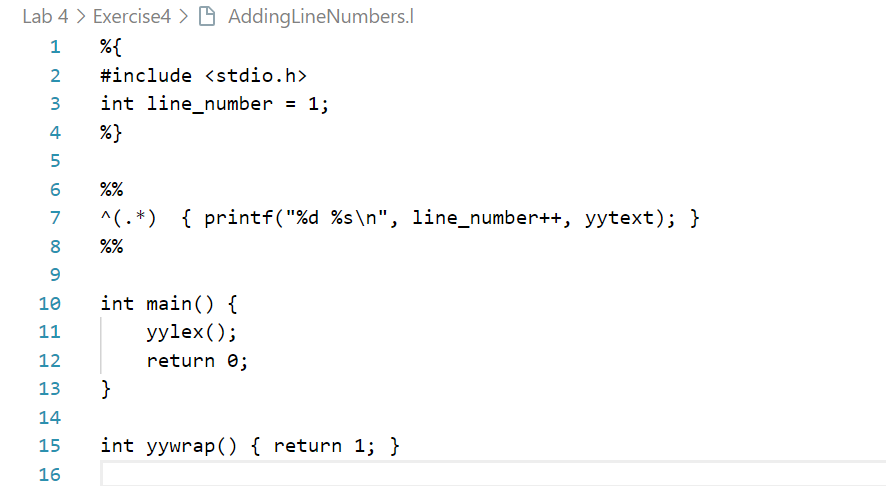
**Exercise 3:**

****

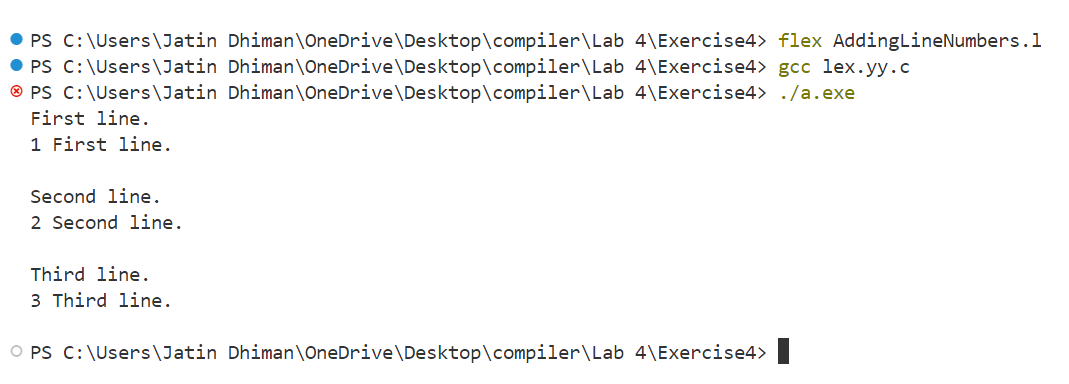
**Output:**



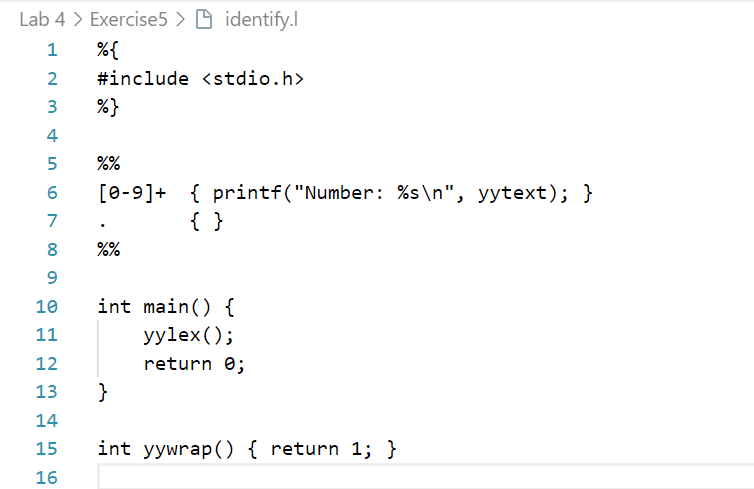
**Exercise 4:**

****

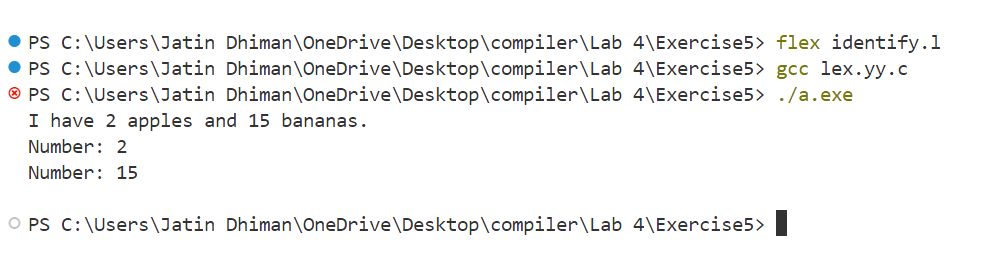
**Output:**



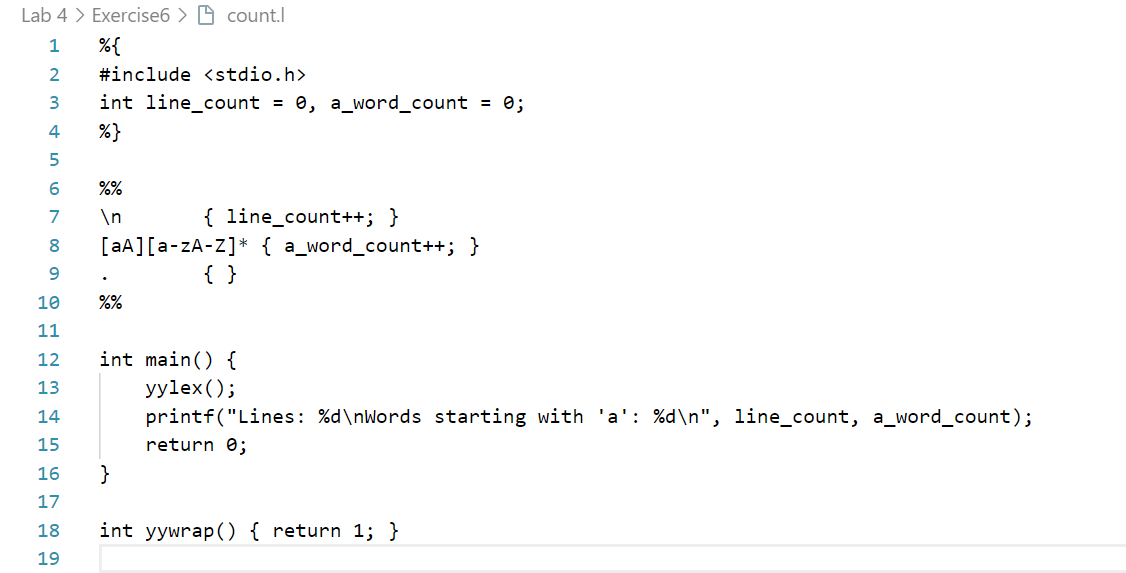
**Exercise 5:**

****

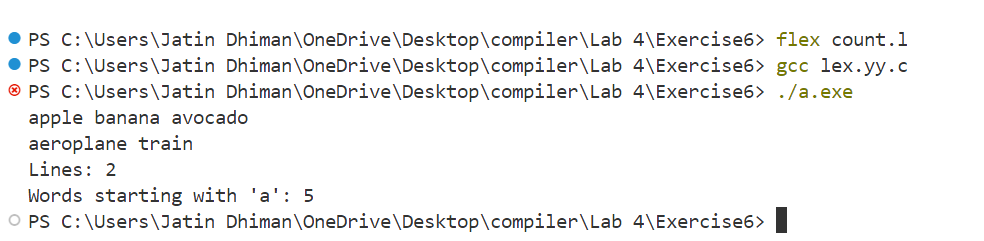
**Output:**



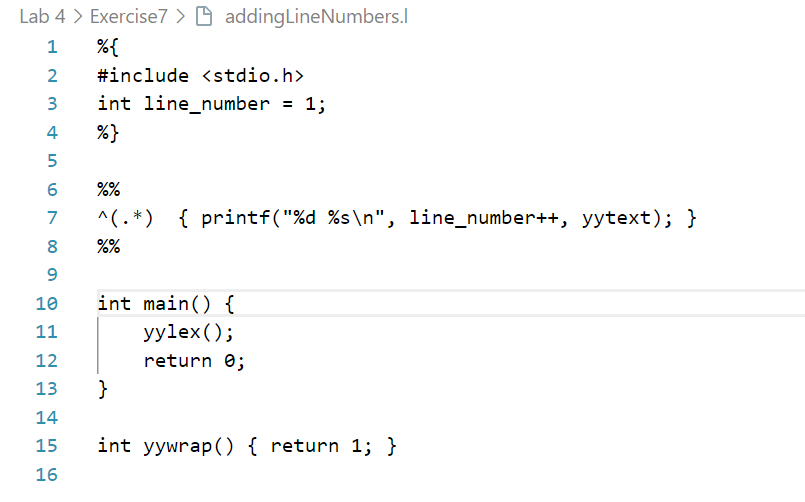
**Exercise 6:**

****

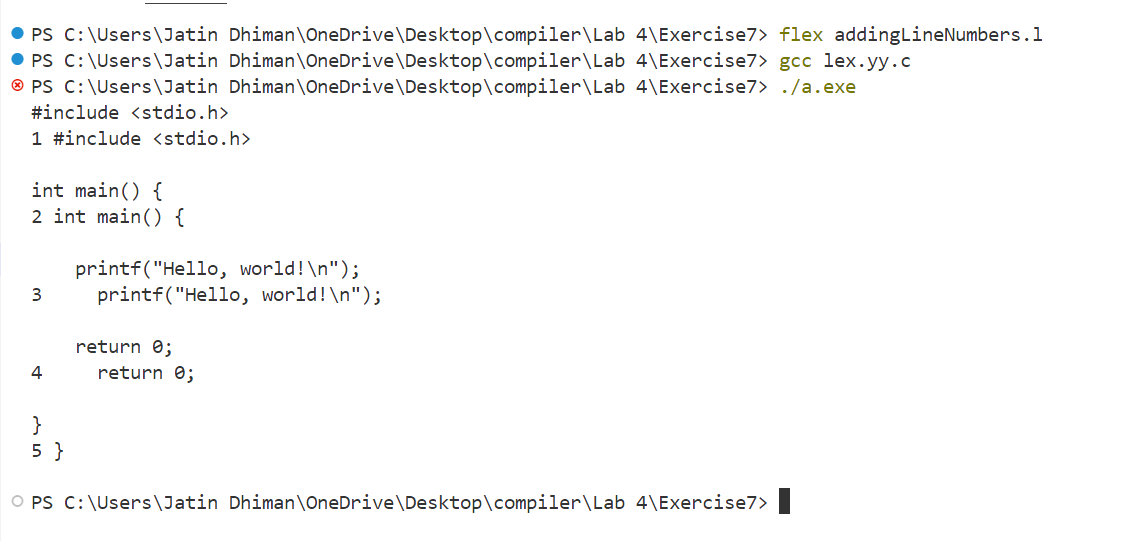
**Output:**



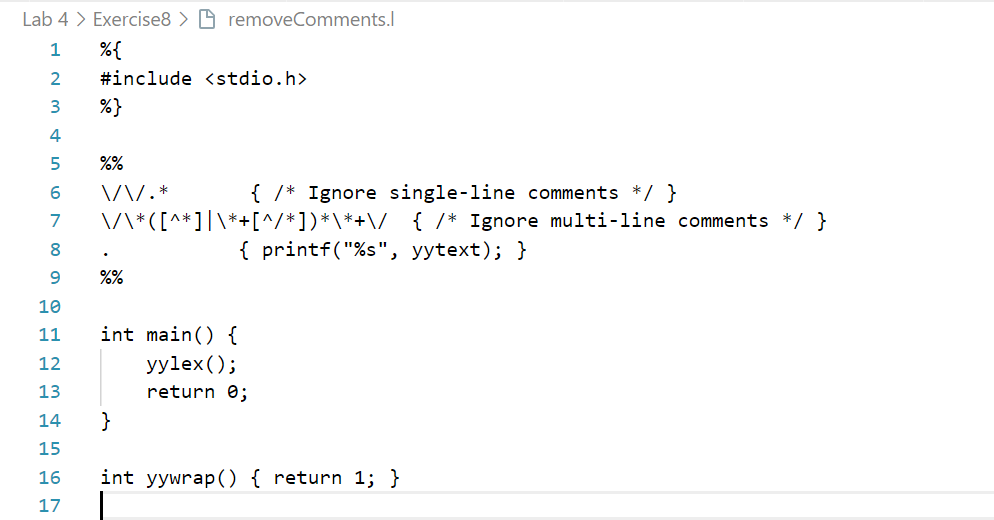
**Exercise 7:**

****

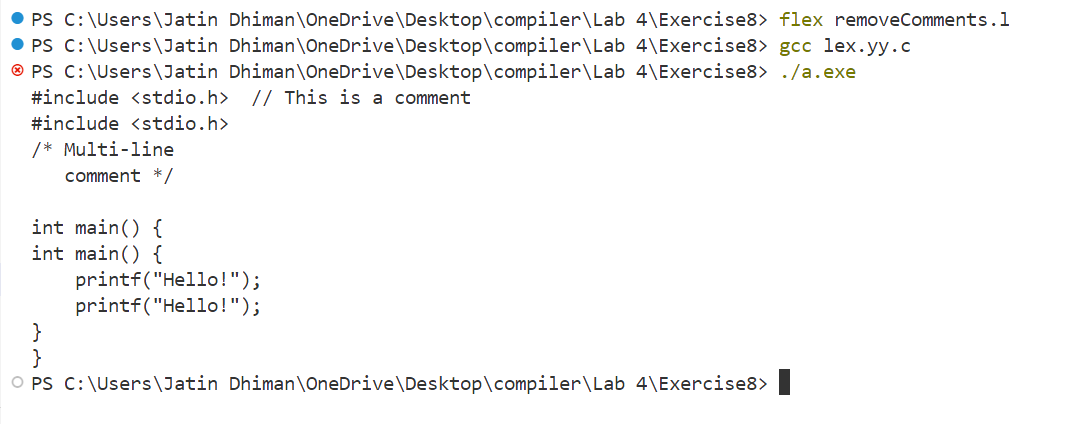
**Output:**



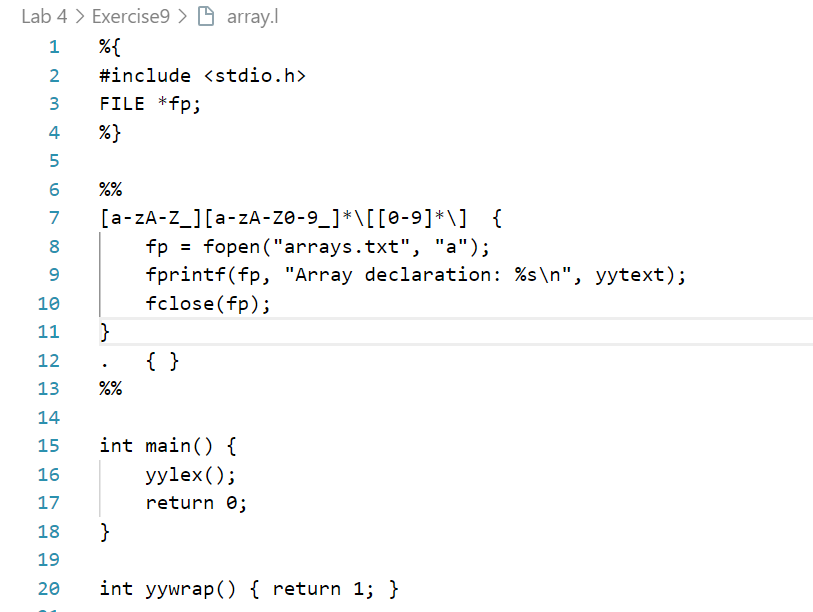
**Exercise 8:**

****

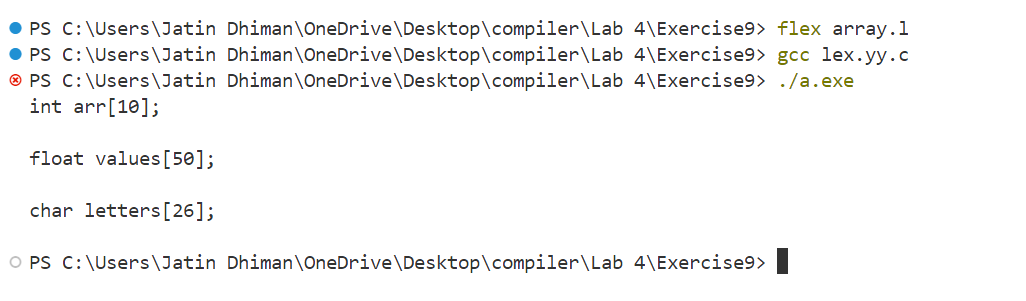
**Output:**

****

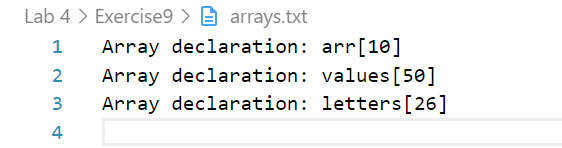
**Exercise 9:**

****

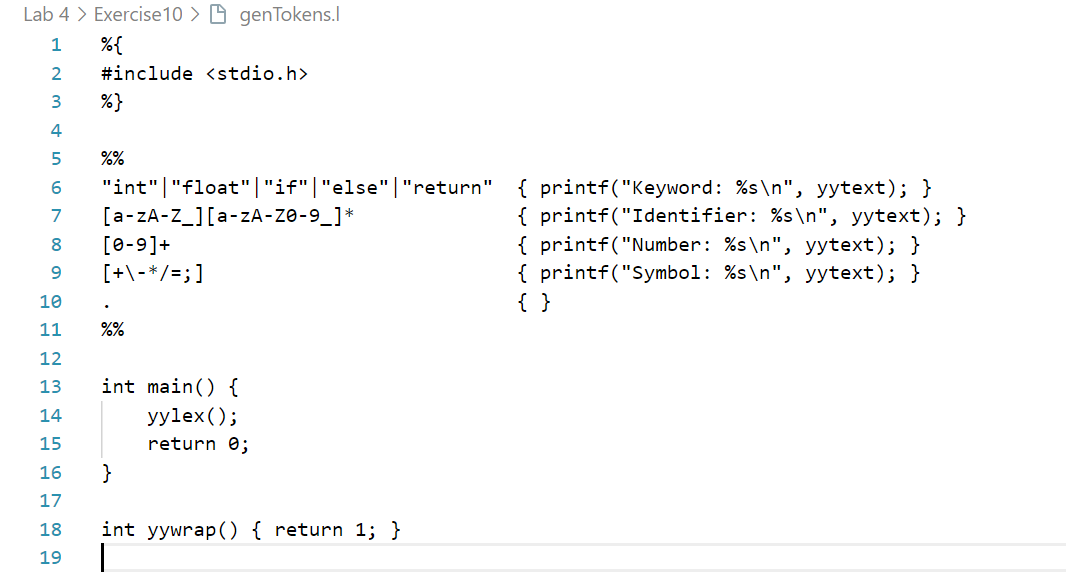
**Output:**

****

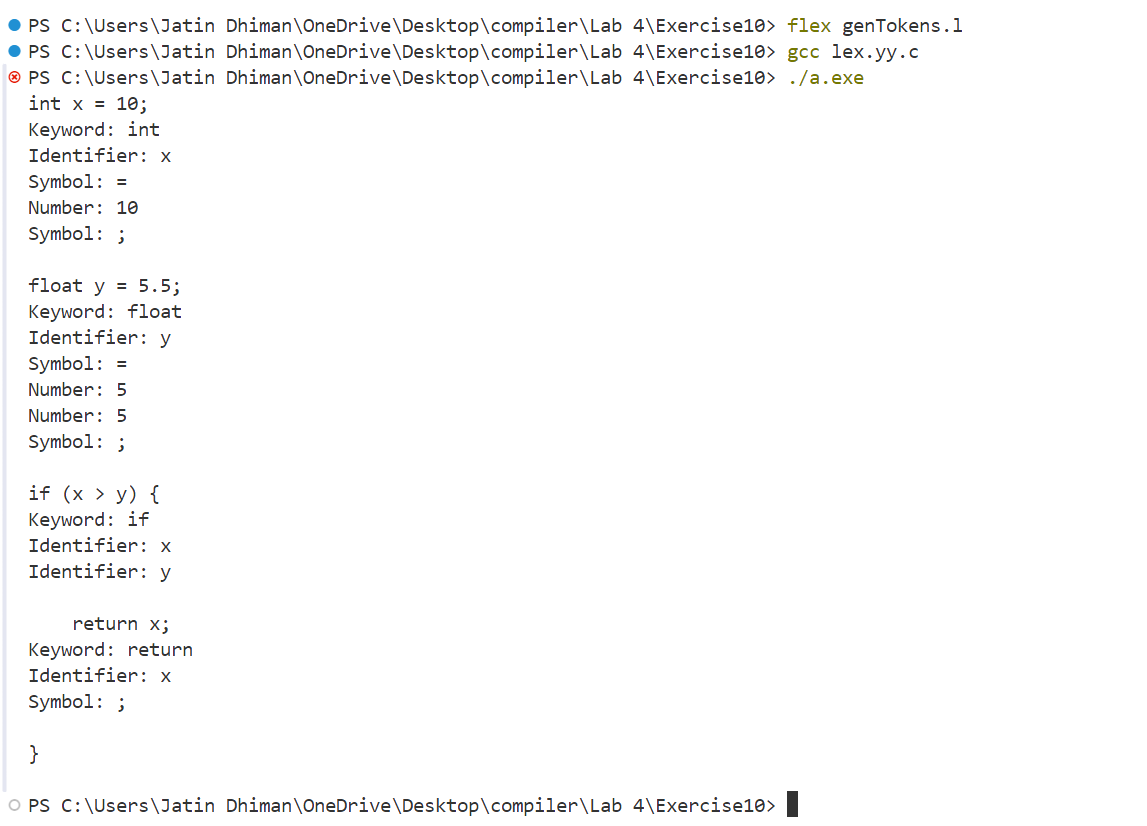
*Arrays.txt*

****

**Exercise 10:**

****

**Output:**

****

**Learnings:**

* Through these exercises, we gained a solid understanding of:
* Lex as a lexical analyzer for pattern recognition.
* Regular expressions for processing and extracting information.
* Handling text transformations, including case conversion, word counting, and line numbering.
* Processing source code, such as adding line numbers and removing comments.
* Tokenizing programming languages, which is a key step in compiler construction.

**Practical 5**

**Aim**

To implement Thompson's construction algorithm for converting a regular expression into a Non-deterministic Finite Automaton (NFA).

**Description**

Thompson’s Construction is a method for converting a regular expression into an NFA. The algorithm constructs an NFA using basic building blocks for concatenation (.), union (|), and Kleene star (\*). The implementation uses a stack-based approach to process the regular expression and build the corresponding NFA.

In this implementation:

* Each state has two possible transitions (out1 and out2).
* A stack is used to store partial NFAs during construction.
* Operators such as |, . (concatenation), and \* (Kleene star) are handled appropriately to build the final NFA.
* A function printNFA() is used to traverse and print the NFA structure.

**Code:**

#include <iostream>

#include <stack>

#include <vector>

#include <set>

using namespace std;

struct State

{

    int id;

    State \*out1 = nullptr; // first transition

    State \*out2 = nullptr; // second transition (for | and \*)

    char symbol;

    State(int id, char symbol = 0){

        this->id = id;

        this->symbol = symbol;

    }

};

struct NFA

{

    State \*start;

    State \*accept;

};

class ThompsonNFA

{

private:

    int stateCount = 0;

    stack<NFA> nfaStack;

    State \*createState(char symbol = 0)

    {

        return new State(stateCount++, symbol);

    }

public:

    NFA regexToNFA(string regex)

    {

        for (char c : regex)

        {

            if (c == '\*')

            {

                NFA nfa = nfaStack.top();

                nfaStack.pop();

                State \*start = createState();

                State \*accept = createState();

                start->out1 = nfa.start;

                start->out2 = accept;

                nfa.accept->out1 = nfa.start;

                nfa.accept->out2 = accept;

                nfaStack.push({start, accept});

            }

            else if (c == '.')

            {

                NFA nfa2 = nfaStack.top();

                nfaStack.pop();

                NFA nfa1 = nfaStack.top();

                nfaStack.pop();

                nfa1.accept->out1 = nfa2.start;

                nfaStack.push({nfa1.start, nfa2.accept});

            }

            else if (c == '|')

            {

                NFA nfa2 = nfaStack.top();

                nfaStack.pop();

                NFA nfa1 = nfaStack.top();

                nfaStack.pop();

                State \*start = createState();

                State \*accept = createState();

                start->out1 = nfa1.start;

                start->out2 = nfa2.start;

                nfa1.accept->out1 = accept;

                nfa2.accept->out1 = accept;

                nfaStack.push({start, accept});

            }

            else

            {

                State \*start = createState();

                State \*accept = createState();

                start->out1 = accept;

                start->symbol = c;

                nfaStack.push({start, accept});

            }

        }

        return nfaStack.top();

    }

    void printNFA(State \*state, set<int> &visited)

    {

        if (!state || visited.count(state->id))

            return;

        visited.insert(state->id);

        cout << "State " << state->id;

        if (state->symbol)

        {

            cout << " ('" << state->symbol << "')";

        }

        else

        {

            cout << " (eps)";

        }

        cout << endl;

        if (state->out1)

        {

            cout << "  -> " << state->out1->id;

            if (!state->out1->symbol)

                cout << " (eps)";

            cout << endl;

        }

        if (state->out2)

        {

            cout << "  -> " << state->out2->id << " (eps)" << endl;

        }

        printNFA(state->out1, visited);

        printNFA(state->out2, visited);

    }

};

int main()

{

    string regex = "ab|\*";

    ThompsonNFA thompson;

    NFA nfa = thompson.regexToNFA(regex);

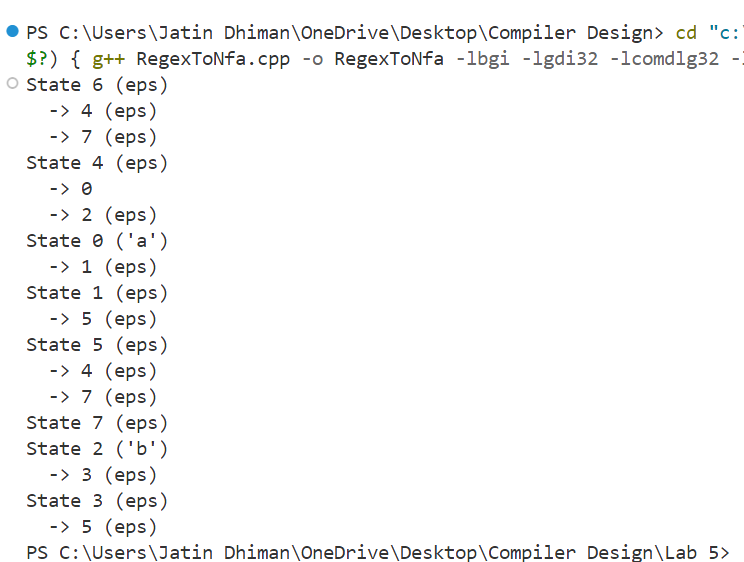
    set<int> visited;

    thompson.printNFA(nfa.start, visited);

    return 0;

}

**Output:**

****

**Learnings:**

* Learned how Thompson's algorithm systematically builds NFAs for regular expressions using basic operators (concatenation, union, Kleene star) and epsilon transitions.

**Practical 6**

**Aim**

To implement a recursive descent parser for validating arithmetic expressions using context-free grammar.

**Description**

A recursive descent parser is a top-down parser that uses a set of mutually recursive procedures to process the input. Each procedure implements one of the non-terminal symbols of the grammar. The given program parses arithmetic expressions involving identifiers and operators (`+`, `-`, `\*`, `/`), with support for parentheses for grouping. The parser validates the input based on the following grammar:  
  
- <E> -> <T> <E'>  
- <E'> -> + <T> <E'> | - <T> <E'> | e  
- <T> -> <F> <T'>`  
- <T'> -> \* <F> <T'> | / <F> <T'> | e  
- <F> -> (<E>) | identifier  
  
The program uses a Token structure to represent the different types of tokens and implements functions to parse the grammar rules. It uses backtracking (save pointers) to try alternate parsing paths when a match fails.

**Code:**

#include <iostream>

#include <vector>

#include <cctype>

#include <string>

using namespace std;

enum TokenType

{

    IDENTIFIER,

    PLUS,

    MINUS,

    MULTIPLY,

    DIVIDE,

    OPEN\_PAREN,

    CLOSE\_PAREN

};

struct Token

{

    TokenType type;

    string value;

};

vector<Token> tokens;

Token \*next\_tok;

bool match(TokenType expected)

{

    if (next\_tok < &tokens[tokens.size()] && next\_tok->type == expected)

    {

        next\_tok++;

        return true;

    }

    return false;

}

bool E();

bool Eprime();

bool T();

bool Tprime();

bool F();

// <E> → <T> <E'>

bool E()

{

    return T() && Eprime();

}

// <E'> → + <T> <E'> | - <T> <E'> | ε

bool Eprime()

{

    Token \*save = next\_tok;

    if (match(PLUS) && T() && Eprime())

    {

        return true;

    }

    next\_tok = save;

    if (match(MINUS) && T() && Eprime())

    {

        return true;

    }

    next\_tok = save;

    return true; // ε (empty string)

}

// <T> → <F> <T'>

bool T()

{

    return F() && Tprime();

}

// <T'> → \* <F> <T'> | / <F> <T'> | ε

bool Tprime()

{

    Token \*save = next\_tok;

    if (match(MULTIPLY) && F() && Tprime())

    {

        return true;

    }

    next\_tok = save;

    if (match(DIVIDE) && F() && Tprime())

    {

        return true;

    }

    next\_tok = save;

    return true; // ε (empty string)

}

// <F> → (<E>) | identifier

bool F()

{

    Token \*save = next\_tok;

    if (match(OPEN\_PAREN) && E() && match(CLOSE\_PAREN))

    {

        return true;

    }

    next\_tok = save;

    if (match(IDENTIFIER))

    {

        return true;

    }

    return false;

}

vector<Token> tokenize(const string &input)

{

    vector<Token> tokens;

    size\_t i = 0;

    while (i < input.size())

    {

        if (isspace(input[i]))

        {

            i++;

            continue;

        }

        if (isalpha(input[i]))

        {

            tokens.push\_back({IDENTIFIER, string(1, input[i])});

        }

        else if (input[i] == '+')

        {

            tokens.push\_back({PLUS, "+"});

        }

        else if (input[i] == '-')

        {

            tokens.push\_back({MINUS, "-"});

        }

        else if (input[i] == '\*')

        {

            tokens.push\_back({MULTIPLY, "\*"});

        }

        else if (input[i] == '/')

        {

            tokens.push\_back({DIVIDE, "/"});

        }

        else if (input[i] == '(')

        {

            tokens.push\_back({OPEN\_PAREN, "("});

        }

        else if (input[i] == ')')

        {

            tokens.push\_back({CLOSE\_PAREN, ")"});

        }

        else

        {

            cout << "Error: Unknown character '" << input[i] << "' at position " << i << endl;

            exit(1);

        }

        i++;

    }

    return tokens;

}

bool parse(const string &input)

{

    tokens = tokenize(input);

    next\_tok = &tokens[0];

    return (E() && next\_tok == &tokens[tokens.size()]);

}

int main()

{

    string input = "x + (y \* z) / w - u";

    cout << "Parsing: " << input << endl;

    if (parse(input))

    {

        cout << "Valid Expression!" << endl;

    }

    else

    {

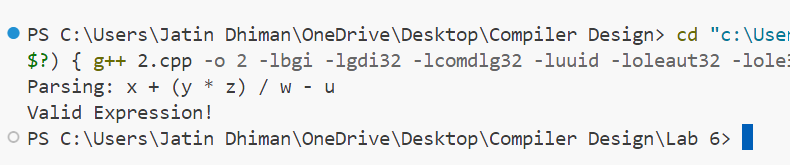
        cout << "Invalid Expression!" << endl;

    }

    return 0;

}

**Output:**

****

**Learnings:**

* Recursive descent parsing is a straightforward way to implement parsers for context-free grammars.
* Backtracking is achieved using `save` pointers to ensure that alternate parsing paths can be explored.
* Tokenization is crucial for converting input strings into manageable structures for parsing.
* Recursive functions handle hierarchical grammar rules elegantly.
* Proper error handling ensures that invalid input is detected gracefully.

**Practical 7**

**Aim**

To compute the First and Follow sets of a given grammar using a C++ program.

**Description**

First and Follow sets are fundamental concepts in compiler design, particularly in the context of parsing techniques. They are used in LL(1) parsers and other parsing strategies to construct parse tables and validate grammar productions.

1. **First Set**:
   * The First set of a non-terminal is the set of terminals that can appear at the beginning of any string derived from that non-terminal.
   * It also includes the epsilon symbol (‘#’ in this program) if the non-terminal can derive an empty string.
2. **Follow Set**:
   * The Follow set of a non-terminal is the set of terminals that can appear immediately to the right of that non-terminal in some "sentential" form.
   * For the start symbol, the Follow set includes the end-of-input symbol ('$').

**Code:**

#include <iostream>

#include <vector>

#include <string>

#include <unordered\_map>

#include <unordered\_set>

#include <cctype>

using namespace std;

unordered\_map<char, vector<string>> grammar;

unordered\_map<char, unordered\_set<char>> firstSet;

unordered\_map<char, unordered\_set<char>> followSet;

char startSymbol;

// Calculate First of a symbol

void calculateFirst(char symbol)

{

    if (firstSet.count(symbol))

        return;

    unordered\_set<char> result;

    for (const string &production : grammar[symbol])

    {

        for (char ch : production)

        {

            if (isupper(ch))

            { // Non-terminal

                calculateFirst(ch);

                result.insert(firstSet[ch].begin(), firstSet[ch].end());

                if (!firstSet[ch].count('#'))

                    break;

            }

            else

            { // Terminal

                result.insert(ch);

                break;

            }

        }

    }

    firstSet[symbol] = result;

}

// Calculate Follow of a symbol

void calculateFollow(char symbol)

{

    if (followSet.count(symbol))

        return;

    unordered\_set<char> result;

    if (symbol == startSymbol)

        result.insert('$');

    for (auto it = grammar.begin(); it != grammar.end(); ++it)

    {

        char lhs = it->first;

        const vector<string> &productions = it->second;

        for (const string &production : productions)

        {

            for (int i = 0; i < production.size(); ++i)

            {

                if (production[i] == symbol)

                {

                    int k = i + 1;

                    while (k < production.size())

                    {

                        char next = production[k];

                        if (isupper(next))

                        {

                            result.insert(firstSet[next].begin(), firstSet[next].end());

                            result.erase('#');

                            if (!firstSet[next].count('#'))

                                break;

                        }

                        else

                        {

                            result.insert(next);

                            break;

                        }

                        k++;

                    }

                    if (k == production.size() && lhs != symbol)

                    {

                        calculateFollow(lhs);

                        result.insert(followSet[lhs].begin(), followSet[lhs].end());

                    }

                }

            }

        }

    }

    followSet[symbol] = result;

}

int main()

{

    grammar['E'] = {"TR"};

    grammar['R'] = {"+TR", "#"};

    grammar['T'] = {"FY"};

    grammar['Y'] = {"\*FY", "#"};

    grammar['F'] = {"(E)", "i"};

    startSymbol = 'E';

    // Calculate First sets

    for (auto it = grammar.begin(); it != grammar.end(); ++it)

        calculateFirst(it->first);

    cout << "First Sets:\n";

    for (auto it = firstSet.begin(); it != firstSet.end(); ++it)

    {

        cout << "First(" << it->first << ") = { ";

        for (char ch : it->second)

            cout << ch << " ";

        cout << "}\n";

    }

    // Calculate Follow sets

    for (auto it = grammar.begin(); it != grammar.end(); ++it)

        calculateFollow(it->first);

    cout << "\nFollow Sets:\n";

    for (auto it = followSet.begin(); it != followSet.end(); ++it)

    {

        cout << "Follow(" << it->first << ") = { ";

        for (char ch : it->second)

            cout << ch << " ";

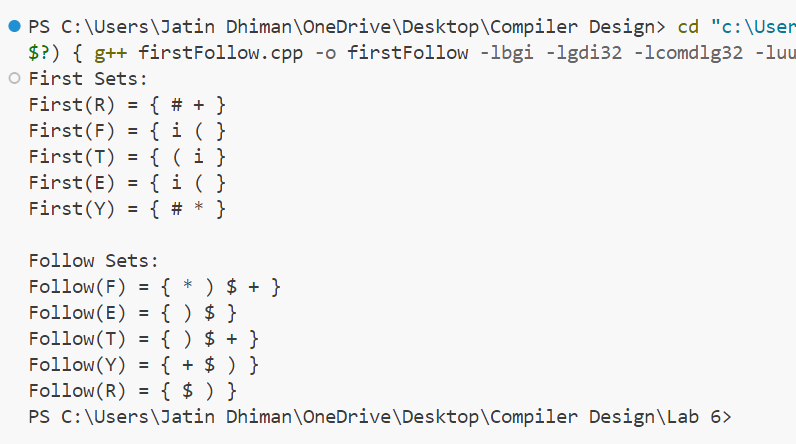
        cout << "}\n";

    }

    return 0;

}

**Output:**

****

**Learnings:**

* Gained understanding of First and Follow sets in grammar analysis

**Practical 8**

**Aim**

To implement an LL(1) parser in C++ that constructs the FIRST and FOLLOW sets, builds the LL(1) parsing table, and simulates the parsing of an input string based on a given grammar.

**Description**

This program demonstrates the working of an LL(1) parser. The grammar is hard-coded and follows a typical expression grammar:  
E → T R  
R → + T R | ε  
T → F Y  
Y → \* F Y | ε  
F → ( E ) | i

Key Components:

* Grammar Storage: Stored using a unordered\_map<char, vector<string>> where each non-terminal maps to its list of productions.
* FIRST Set Calculation: For each non-terminal, this set contains the terminals that can appear as the first symbol in any string derived from it.
* FOLLOW Set Calculation: For each non-terminal, this set contains the terminals that can appear immediately to the right of it in some 'sentential' form.
* Parsing Table Construction: A 2D table (unordered\_map<char, unordered\_map<char, string>>) is built using FIRST and FOLLOW sets to decide parsing rules.
* Parsing Simulation: The parser simulates stack-based parsing of an input string by comparing symbols on the stack with the current input character and applying grammar rules accordingly.

**Code:**

#include <iostream>

#include <vector>

#include <string>

#include <unordered\_map>

#include <unordered\_set>

#include <iomanip>

using namespace std;

unordered\_map<char, vector<string>> grammar;

unordered\_map<char, unordered\_set<char>> firstSet;

unordered\_map<char, unordered\_set<char>> followSet;

unordered\_map<char, unordered\_map<char, string>> parsingTable;

char startSymbol;

// Calculate First of a symbol

void calculateFirst(char symbol)

{

    if (firstSet.count(symbol))

        return;

    unordered\_set<char> result;

    for (const string &production : grammar[symbol])

    {

        for (char ch : production)

        {

            if (isupper(ch))

            { // Non-terminal

                calculateFirst(ch);

                result.insert(firstSet[ch].begin(), firstSet[ch].end());

                if (!firstSet[ch].count('#'))

                    break;

            }

            else

            { // Terminal

                result.insert(ch);

                break;

            }

        }

    }

    firstSet[symbol] = result;

}

// Calculate Follow of a symbol

void calculateFollow(char symbol)

{

    if (followSet.count(symbol))

        return;

    unordered\_set<char> result;

    if (symbol == startSymbol)

        result.insert('$'); // End marker

    for (auto &rule : grammar)

    {

        char lhs = rule.first;

        const vector<string> &productions = rule.second;

        for (const string &production : productions)

        {

            for (size\_t i = 0; i < production.size(); ++i)

            {

                if (production[i] == symbol)

                {

                    size\_t k = i + 1;

                    while (k < production.size())

                    {

                        char next = production[k];

                        if (isupper(next))

                        {

                            result.insert(firstSet[next].begin(), firstSet[next].end());

                            result.erase('#');

                            if (!firstSet[next].count('#'))

                                break;

                        }

                        else

                        {

                            result.insert(next);

                            break;

                        }

                        k++;

                    }

                    if (k == production.size() && lhs != symbol)

                    {

                        calculateFollow(lhs);

                        result.insert(followSet[lhs].begin(), followSet[lhs].end());

                    }

                }

            }

        }

    }

    followSet[symbol] = result;

}

// Construct Parsing Table

void constructParsingTable()

{

    for (auto &rule : grammar)

    {

        char nonTerminal = rule.first;

        for (const string &production : rule.second)

        {

            unordered\_set<char> firstAlpha;

            for (char ch : production)

            {

                if (isupper(ch))

                {

                    firstAlpha = firstSet[ch];

                    if (!firstAlpha.count('#'))

                        break;

                }

                else

                {

                    firstAlpha.insert(ch);

                    break;

                }

            }

            for (char terminal : firstAlpha)

            {

                if (terminal != '#')

                {

                    parsingTable[nonTerminal][terminal] = production;

                }

            }

            if (firstAlpha.count('#'))

            {

                for (char terminal : followSet[nonTerminal])

                {

                    parsingTable[nonTerminal][terminal] = production;

                }

            }

        }

    }

}

// Print the Parsing Table

void printParsingTable()

{

    cout << "\nLL(1) Parsing Table:\n";

    cout << setw(10) << " ";

    unordered\_set<char> terminals;

    for (auto &rule : grammar)

    {

        for (auto &production : rule.second)

        {

            for (char ch : production)

            {

                if (!isupper(ch) && ch != '#')

                    terminals.insert(ch);

            }

        }

    }

    terminals.insert('$'); // Add the end-of-input marker

    for (char terminal : terminals)

        cout << setw(10) << terminal;

    cout << endl;

    for (auto &row : parsingTable)

    {

        cout << setw(10) << row.first;

        for (char terminal : terminals)

        {

            if (parsingTable[row.first].count(terminal))

                cout << setw(10) << parsingTable[row.first][terminal];

            else

                cout << setw(10) << " ";

        }

        cout << endl;

    }

}

// Simulate Parsing

void parseString(const string &input)

{

    cout << "\nParsing Input: " << input << endl;

    string augmentedInput = input + "$";

    vector<char> stack;

    stack.push\_back('$');

    stack.push\_back(startSymbol);

    int i = 0;

    while (!stack.empty())

    {

        char top = stack.back();

        char current = augmentedInput[i];

        cout << "Stack: ";

        for (char c : stack)

            cout << c;

        cout << " | Input: " << augmentedInput.substr(i) << " | Action: ";

        if (top == current)

        {

            cout << "Match '" << top << "'" << endl;

            stack.pop\_back();

            i++;

        }

        else if (!isupper(top))

        {

            cout << "Error (Expected '" << top << "', got '" << current << "')" << endl;

            return;

        }

        else if (parsingTable[top].count(current))

        {

            string production = parsingTable[top][current];

            cout << top << " -> " << production << endl;

            stack.pop\_back();

            if (production != "#")

            {

                for (int j = production.size() - 1; j >= 0; j--)

                    stack.push\_back(production[j]);

            }

        }

        else

        {

            cout << "Error (No rule for [" << top << ", " << current << "])" << endl;

            return;

        }

    }

    cout << "\nString accepted by the grammar.\n";

}

// Main Function

int main()

{

    grammar['E'] = {"TR"};

    grammar['R'] = {"+TR", "#"};

    grammar['T'] = {"FY"};

    grammar['Y'] = {"\*FY", "#"};

    grammar['F'] = {"(E)", "i"};

    startSymbol = 'E';

    // Compute First sets

    for (auto &rule : grammar)

        calculateFirst(rule.first);

    cout << "First Sets:\n";

    for (auto &entry : firstSet)

    {

        cout << "First(" << entry.first << ") = { ";

        for (char ch : entry.second)

            cout << ch << " ";

        cout << "}\n";

    }

    // Compute Follow sets

    for (auto &rule : grammar)

        calculateFollow(rule.first);

    cout << "\nFollow Sets:\n";

    for (auto &entry : followSet)

    {

        cout << "Follow(" << entry.first << ") = { ";

        for (char ch : entry.second)

            cout << ch << " ";

        cout << "}\n";

    }

    // Construct Parsing Table

    constructParsingTable();

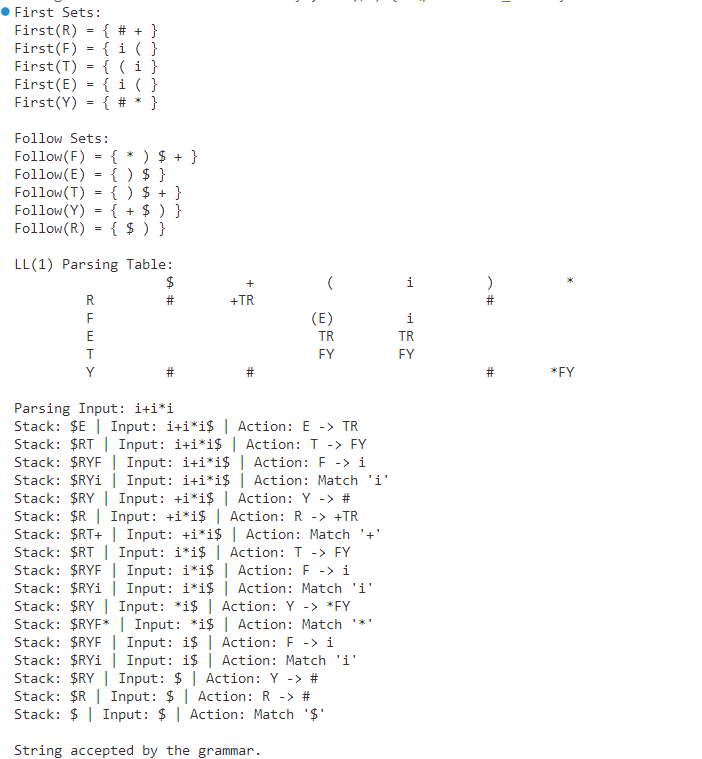
    printParsingTable();

    parseString("i+i\*i");

    return 0;

}

**Output:**

****

**Learnings:**

* Understanding of Grammar Structures: How non-terminals and terminals form the foundation of a context-free grammar.
* FIRST and FOLLOW Sets: How they are used to resolve ambiguity and construct parsing tables.
* Predictive Parsing: How an LL(1) parser uses lookahead to choose production rules without backtracking.
* Stack-based Parsing Mechanism: Simulation of parsing using a stack that helps track derivations and matches between input and grammar rules.
* Error Handling in Parsing: How to detect and report unexpected tokens or missing rules.
* C++ Data Structures Usage: Effective use of STL containers like unordered\_map, unordered\_set, and vector to manage grammar and parsing efficiently.

**Practical 9**

**Aim**

To implement a Shift-Reduce Parser (specifically an LR parser) in C++ that simulates parsing of an input string using a predefined parsing table and a set of grammar productions.

**Description**

This C++ program implements a basic LR parser using a manually defined parsing table and production rules. It processes an input string token by token and simulates the behavior of a bottom-up parser (Shift-Reduce Parser). The parser uses:

* A **stack** to maintain parser states and symbols.
* A **parsing table** (action/goto) to decide on shift, reduce, or accept actions.
* A **set of grammar productions** for performing reductions.

**Key components:**

* **Parsing Table**: Implemented as a nested unordered\_map, where actions like shift (e.g., s5), reduce (e.g., r2), goto transitions, and accept (Acc) are defined.
* **Productions**: Stored in an unordered\_map with rule numbers mapping to their LHS (left-hand side) and RHS (right-hand side).
* **Input String**: Tokenized input provided as a vector simulating lexer output. For example: id \* id + id $
* **Parser Logic**:
  + Reads current state from top of the stack.
  + Uses current token to look up action from the parsing table.
  + Executes shift, reduce, or accept based on the action retrieved.

If a parsing error occurs (i.e., no entry in the table for a particular state and token), the program prints Error.

**Code:**

#include <iostream>

#include <unordered\_map>

#include <string>

#include <stack>

#include <vector>

using namespace std;

void storeActions(unordered\_map<int, unordered\_map<string, string>> &parsingTable);

void storeProductions(unordered\_map<int, pair<string, vector<string>>> &productions);

int main()

{

    unordered\_map<int, unordered\_map<string, string>> parsingTable;

    unordered\_map<int, pair<string, vector<string>>> productions;

    stack<string> st;

    st.push("0");

    storeActions(parsingTable);

    storeProductions(productions);

    vector<string> stringTokens = {"id", "\*", "id", "+", "id", "$"};

    int idx = 0;

    while (idx < stringTokens.size())

    {

        int state = stoi(st.top());

        string token = stringTokens[idx];

        string what = parsingTable[state][token];

        if (what == "")

        {

            cout << "Error"<< endl;

            break;

        }

        if (what[0] == 's')

        {

            //shift

            string nextState = what.substr(1);

            st.push(token);

            st.push(nextState);

            idx++;

        }

        else if (what[0] == 'r')

        {

            // reduce

            int num = stoi(what.substr(1));

            pair<string, vector<string>> prod = productions[num];

            string lhs = prod.first;

            vector<string> rhs = prod.second;

            for (int i = 0; i < rhs.size() \* 2; i++)

                st.pop();

            int newState = stoi(st.top());

            st.push(lhs);

            string goto\_ = parsingTable[newState][lhs];

            st.push(goto\_);

        }

        else if (what == "Acc")

        {

            cout << "Accepted String\n";

            break;

        }

    }

    return 0;

}

void storeProductions(unordered\_map<int, pair<string, vector<string>>> &productions)

{

    productions[1] = {"E", {"E", "+", "T"}};

    productions[2] = {"E", {"T"}};

    productions[3] = {"T", {"T", "\*", "F"}};

    productions[4] = {"T", {"F"}};

    productions[5] = {"F", {"(", "E", ")"}};

    productions[6] = {"F", {"id"}};

}

void storeActions(unordered\_map<int, unordered\_map<string, string>> &parsingTable)

{

    parsingTable[0]["id"] = "s5";

    parsingTable[0]["("] = "s4";

    parsingTable[0]["E"] = "1";

    parsingTable[0]["T"] = "2";

    parsingTable[0]["F"] = "3";

    parsingTable[1]["+"] = "s6";

    parsingTable[1]["$"] = "Acc";

    parsingTable[2]["+"] = "r2";

    parsingTable[2]["\*"] = "s7";

    parsingTable[2][")"] = "r2";

    parsingTable[2]["$"] = "r2";

    parsingTable[3]["+"] = "r4";

    parsingTable[3]["\*"] = "r4";

    parsingTable[3][")"] = "r4";

    parsingTable[3]["$"] = "r4";

    parsingTable[4]["id"] = "s5";

    parsingTable[4]["("] = "s4";

    parsingTable[4]["E"] = "8";

    parsingTable[4]["T"] = "2";

    parsingTable[4]["F"] = "3";

    parsingTable[5]["+"] = "r6";

    parsingTable[5]["\*"] = "r6";

    parsingTable[5][")"] = "r6";

    parsingTable[5]["$"] = "r6";

    parsingTable[6]["id"] = "s5";

    parsingTable[6]["("] = "s4";

    parsingTable[6]["T"] = "9";

    parsingTable[6]["F"] = "3";

    parsingTable[7]["id"] = "s5";

    parsingTable[7]["("] = "s4";

    parsingTable[7]["F"] = "10";

    parsingTable[8]["+"] = "s6";

    parsingTable[8][")"] = "s11";

    parsingTable[9]["+"] = "r1";

    parsingTable[9]["\*"] = "s7";

    parsingTable[9][")"] = "r1";

    parsingTable[9]["$"] = "r1";

    parsingTable[10]["+"] = "r3";

    parsingTable[10]["\*"] = "r3";

    parsingTable[10][")"] = "r3";

    parsingTable[10]["$"] = "r3";

    parsingTable[11]["+"] = "r5";

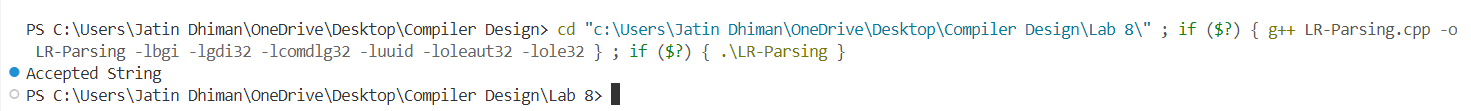
    parsingTable[11]["\*"] = "r5";

    parsingTable[11][")"] = "r5";

    parsingTable[11]["$"] = "r5";

}

**Output:**

****

**Learnings:**

* Understanding of **bottom-up parsing**, particularly **LR parsing** techniques.
* Concept of **shift-reduce parsing** and how actions are determined using parsing tables.
* Use of C++ data structures like unordered\_map, vector, and stack for implementing compiler design concepts.
* Insight into how parsing tables are constructed and how they guide the parsing process.

**AIM**

To implement an Operator Precedence Parser that constructs leading and trailing sets, generates the precedence relations table, and parses an input expression to validate it using operator precedence rules.

**DESCRIPTION**

This program implements an Operator Precedence Parser for the given grammar productions:

E -> E + T

E -> T

T -> T \* F

T -> F

F -> (E)

F -> 9

Steps:

1. Compute Leading and Trailing Sets for each non-terminal.

2. Generate the Operator Precedence Table using these sets.

3. Parse input strings using shift-reduce moves guided by the precedence table.

**CODE**

#include<iostream>

#include<set>

#include<map>

#include<vector>

#include<stack>

using namespace std;

map<pair<char,char>,char>table;

vector<string> productions;

map<char,set<char>>trailing;

map<char, set<char>> leading;

bool isTerminal(char c) {

return (c >= 'a' && c <= 'z') || c == '+' || c == '-' || c == '\*' || c == '/' || c == '(' || c == ')' || c == '9';

}

void leading\_fun(char nonter, set<char>& lead1, set<char>& visited) {

if (visited.count(nonter)) return;

visited.insert(nonter);

for (const string& prod : productions) {

if (prod[0] == nonter) {

bool terminal\_found = false;

for (int i = 3; i < prod.length(); i++) {

char symbol = prod[i];

if (isTerminal(symbol)) {

lead1.insert(symbol);

terminal\_found = true;

break;

}

}

if (!terminal\_found && prod.length() > 3) {

char firstSymbol = prod[3];

if (firstSymbol >= 'A' && firstSymbol <= 'Z') {

if (leading.find(firstSymbol) == leading.end()) {

set<char> temp;

leading\_fun(firstSymbol, temp, visited);

leading[firstSymbol] = temp;

}

lead1.insert(leading[firstSymbol].begin(), leading[firstSymbol].end());

}

}

}

}

}

void trailing\_fun(char nonter, set<char>& trail1, set<char>& visited) {

if (visited.count(nonter)) return;

visited.insert(nonter);

for (const string& prod : productions) {

if (prod[0] == nonter) {

bool terminal\_found = false;

for (int i = prod.length(); i >= 3; i--) {

char symbol = prod[i];

if (isTerminal(symbol)) {

trail1.insert(symbol);

terminal\_found = true;

break;

}

}

if (!terminal\_found && prod.length() > 3) {

char firstSymbol = prod[3];

if (firstSymbol >= 'A' && firstSymbol <= 'Z') {

if (trailing.find(firstSymbol) == trailing.end()) {

set<char> temp;

trailing\_fun(firstSymbol, temp, visited);

trailing[firstSymbol] = temp;

}

trail1.insert(trailing[firstSymbol].begin(), trailing[firstSymbol].end());

}

}

}

}

}

bool isOperator(char symbol) {

return (symbol == '+' || symbol == '\*' || symbol == '(' || symbol == ')' || symbol == '9');

}

void create\_operator\_precedence\_table() {

for (const string& prod : productions) {

string rhs = prod.substr(3);

for (size\_t i = 0; i < rhs.size(); ++i) {

char curr = rhs[i];

if (isOperator(curr) && i + 1 < rhs.size() && isupper(rhs[i + 1])) {

char nonterm = rhs[i + 1];

for (char lead : leading[nonterm]) {

if (isOperator(lead) || lead == '9' || lead == '(') {

table[{curr, lead}] = '<';

}

}

}

if (isupper(curr) && i + 1 < rhs.size() && isOperator(rhs[i + 1])) {

char next = rhs[i + 1];

for (char trail : trailing[curr]) {

if (isOperator(trail) || trail == '9' || trail == ')') {

table[{trail, next}] = '>';

}

}

}

if (i + 2 < rhs.size() && isOperator(rhs[i]) &&

isupper(rhs[i + 1]) && isOperator(rhs[i + 2])) {

table[{rhs[i], rhs[i + 2]}] = '=';

}

if (i + 1 < rhs.size() && isOperator(curr) && isOperator(rhs[i + 1])) {

table[{curr, rhs[i + 1]}] = '>';

}

}

}

char startSymbol = productions[0][0];

for (char lead : leading[startSymbol]) {

if (isOperator(lead) || lead == '9' || lead == '(') {

table[{'$', lead}] = '<';

}

}

for (char trail : trailing[startSymbol]) {

if (isOperator(trail) || trail == '9' || trail == ')') {

table[{trail, '$'}] = '>';

}

}

}

void print\_precedence\_table() {

for (const auto& entry : table) {

cout << "table[{'" << entry.first.first << "', '" << entry.first.second << "'}] = '" << entry.second << "';" << endl;

}

}

void operator\_parser(string input){

stack<char> s;

s.push('$');

int i = 0;

while(i<input.length()){

char move = table[{s.top(),input[i]}];

if(move=='<' || move=='='){

cout << "shift move!" << endl;

s.push(input[i]);

i++;

}

if(move=='>'){

cout << "reduce move!" << endl;

while(table[{s.top(),input[i]}] == '>'){

s.pop();

}

}

if(s.top() == '$' && input[i] == '$'){

cout << "string is valid" << endl;

return;

}

}

}

int main(){

productions = {"E->E+T","E->T","T->T\*F","T->F","F->(E)","F->9"};

int n = productions.size();

for (int i = 0; i < n; i++) {

char non\_term = productions[i][0];

if (leading.find(non\_term) == leading.end()) {

set<char> lead\_1;

set<char> visited;

leading\_fun(non\_term, lead\_1, visited);

leading[non\_term] = lead\_1;

}

}

for (const auto& i : leading) {

cout << "Leading(" << i.first << ") - { ";

for (char ch : i.second) {

cout << ch << " ";

}

cout << "}" << endl;

}

for (int i = 0; i < n; i++) {

char non\_term = productions[i][0];

if (trailing.find(non\_term) == trailing.end()) {

set<char> trail\_1;

set<char> visited1;

trailing\_fun(non\_term, trail\_1, visited1);

trailing[non\_term] = trail\_1;

}

}

for (const auto& i : trailing) {

cout << "trailing(" << i.first << ") - { ";

for (char ch : i.second) {

cout << ch << " ";

}

cout << "}" << endl;

}

create\_operator\_precedence\_table();

print\_precedence\_table();

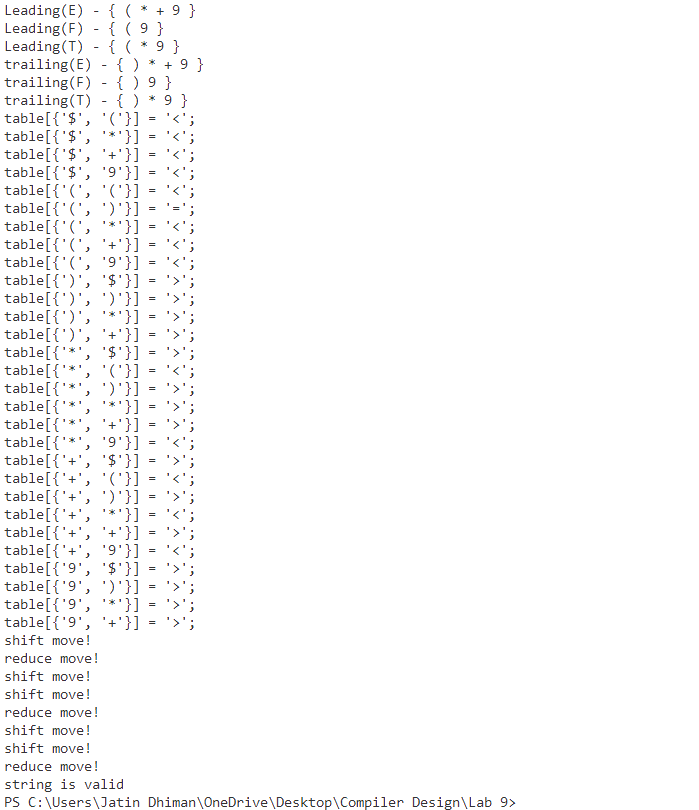
string input = "9+9\*9$";

operator\_parser(input);

return 0;

}

**OUTPUT**

****

**LEARNINGS**

- Construction of Leading and Trailing sets

- Operator precedence parsing table generation

- Use of maps, sets, and stacks in C++

- Shift-reduce parsing based on precedence

- Parsing decisions driven by '<', '=', '>' relations