PRACTICAL - 3

Practical Definition : Implementation of a Lexical Analyzer for C Language Compiler

Objective : To design and implement a lexical analyser, the first phase of a compiler, for the C programming language. The lexical analyser should perform the following tasks: (1) tokenizing the input string (2) removing comments (3) removing white spaces (4) entering identifiers into the symbol table (5) generating lexical errors.

Input requirement :

- Accept a C source code file.
- The input can contain keywords, identifiers, constants, strings, punctuation, operators, comments, and white spaces.

Expected output :

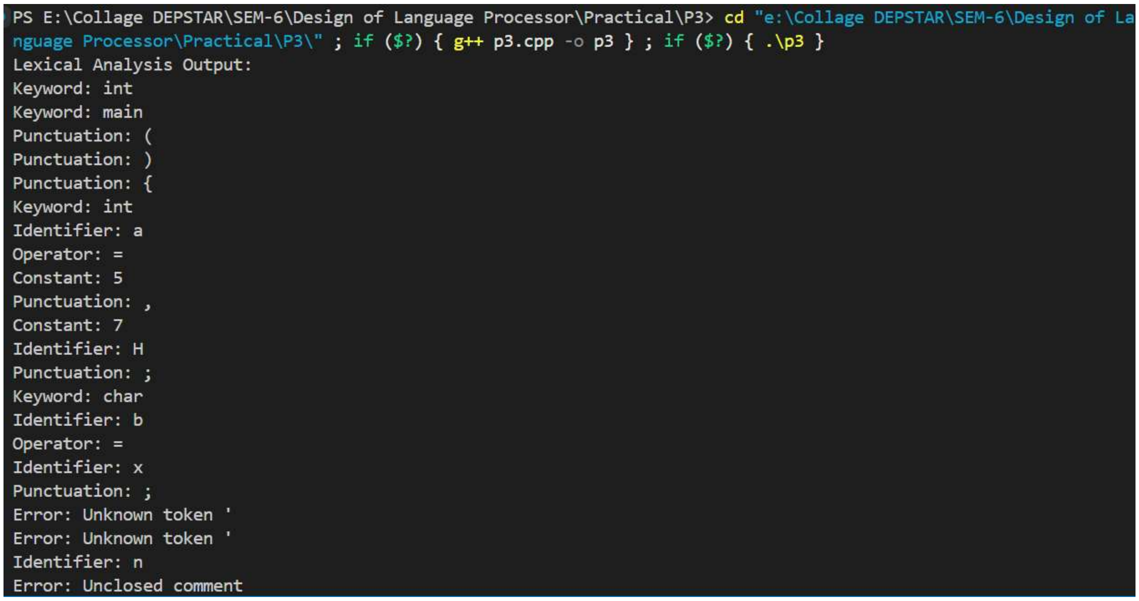
- Tokenized output categorizing tokens into six types: keyword, identifier, constant, string, punctuation, and operator.
- Symbol table with all identified identifiers stored.
- Detection and reporting of lexical errors
- Modified source code

CODE :

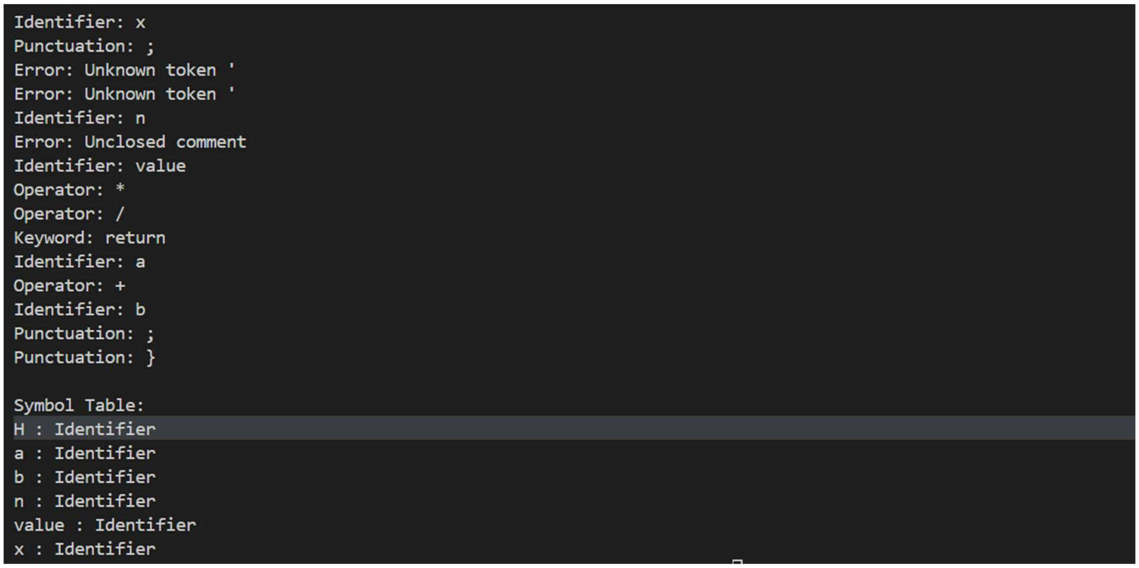
```
#include<bits/stdc++.h>

using namespace std;


const set<string> KEYWORDS = {"int", "float", "char", "if", "else", "while", "for", "return",
"const", "void", "main", "switch", "case", "break", "continue", "printf", "scanf"};

const set<string> OPERATORS = {"+", "-", "*", "/", "=", "==", "!=", "<", ">", "<=", ">=",
"&&", "||", "!", "++", "--", "&", "|", "^", "~"};

const set<char> PUNCTUATION = {';', ',', '(', ')', '{', '}', '[', ']'};


map<string, string> symbolTable;


void analyze(const string &input) {

    size_t index = 0;
    size_t length = input.length();
    vector<string> errors;

    while (index < length) {
        char currentChar = input[index];

        if (isspace(currentChar)) {
            index++;
            continue;
        }

        if (currentChar == '/' && index + 1 < length) {
            if (input[index + 1] == '/') {
                break;
            } else if (input[index + 1] == '*') {
                index += 2;
                while (index + 1 < length && !(input[index] == '*' && input[index + 1] == '/')) {
                    index++;
                }
                if (index + 1 < length) {
                    index += 2;
                } else {
                    errors.push_back("Error: Unclosed comment");
                }
                continue;
            }
        }

        if (currentChar == '"') {
            string stringToken = "\"";
            index++;
            while (index < length && input[index] != '"') {
                if (input[index] == '\\') {
                    stringToken += input[index];
                    index++;
                }
                stringToken += input[index];
                index++;
            }
            if (index < length && input[index] == '"') {
                stringToken += '"';
                cout << "String: " << stringToken << endl;
                index++;
            } else {
                errors.push_back("Error: Unclosed string literal");
            }
            continue;
        }

        if (isdigit(currentChar)) {
            string constant;
            bool hasDot = false;
            while (index < length && (isdigit(input[index]) || input[index] == '.')) {
                if (input[index] == '.') {
                    if (hasDot) {
                        errors.push_back("Error: Malformed constant with multiple dots");
                        break;
                    }
                    hasDot = true;
                }
                constant += input[index];
                index++;
            }
            if (!constant.empty() && constant.back() == '.') {
                errors.push_back("Error: Malformed constant ending with a dot: " + constant);
            } else {
                cout << "Constant: " << constant << endl;
            }
            continue;
        }

        if (isalpha(currentChar) || currentChar == '_') {
            string identifier;
            while (index < length && (isalnum(input[index]) || input[index] == '_')) {
                identifier += input[index];
                index++;
            }
            if (KEYWORDS.count(identifier)) {
                cout << "Keyword: " << identifier << endl;
            } else {
                cout << "Identifier: " << identifier << endl;
                symbolTable[identifier] = "Identifier";
            }
            continue;
        }

        bool matchedOperator = false;
        for (const string &op : OPERATORS) {
            if (input.substr(index, op.size()) == op) {
                cout << "Operator: " << op << endl;
```

```
                index += op.size();
                matchedOperator = true;
                break;
            }
        }
        if (matchedOperator) continue;

        if (PUNCTUATION.count(currentChar)) {
            cout << "Punctuation: " << currentChar << endl;
            index++;
            continue;
        }

        errors.push_back("Error: Unknown token " + string(1, currentChar));
        index++;
    }

    if (!errors.empty()) {
        for (const string &error : errors) {
            cout << error << endl;
        }
    }
}

int main() {
    string filePath = "file1.c";

    ifstream file(filePath);
    if (!file.is_open()) {
        cerr << "Error: Unable to open file: " << filePath << endl;
        return 1;
    }

    string line;
    cout << "Lexical Analysis Output:" << endl;

    bool isEmpty = true;
    while (getline(file, line)) {
        isEmpty = false;
        analyze(line);
    }

    if (isEmpty) {
        cout << "Error: The file " << filePath << " is empty." << endl;
    }

    cout << "\nSymbol Table:" << endl;
    if (symbolTable.empty()) {
        cout << "No identifiers found." << endl;
    } else {
        for (const auto &entry : symbolTable) {
            cout << entry.first << " : " << entry.second << endl;
        }
    }

    file.close();
    return 0;
}
```

OUTPUT :



```
PS E:\Collage DEPSTAR\SEM-6\Design of Language Processor\Practical\P3> cd "e:\Collage DEPSTAR\SEM-6\Design of La
nguage Processor\Practical\P3\" ; if ($?) { g++ p3.cpp -o p3 } ; if ($?) { .\p3 }
Lexical Analysis Output:
Keyword: int
Keyword: main
Punctuation: (
Punctuation: )
Punctuation: {
Keyword: int
Identifier: a
Operator: =
Constant: 5
Punctuation: ,
Constant: 7
Identifier: H
Punctuation: ;
Keyword: char
Identifier: b
Operator: =
Identifier: x
Punctuation: ;
Error: Unknown token '
Error: Unknown token '
Identifier: n
Error: Unclosed comment
```



```
Identifier: x
Punctuation: ;
Error: Unknown token '
Error: Unknown token '
Identifier: n
Error: Unclosed comment
Identifier: value
Operator: *
Operator: /
Keyword: return
Identifier: a
Operator: +
Identifier: b
Punctuation: ;
Punctuation: }

Symbol Table:
H : Identifier
a : Identifier
b : Identifier
n : Identifier
value : Identifier
x : Identifier
```