Practical-3

DEFINATION: 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..

CODE:

```
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include <unordered_set>
#include <cctype>

using namespace std;

struct Token
{
   string type;
   string value;
};

struct SymbolTableEntry
{
   string name;
```

```
};

class LexicalAnalyzer
{
private:
   vector<Token> tokens;
   vector<SymbolTableEntry> symbolTable;
   unordered_set<string> keywords = {"int","printf", "scanf", "char", "return", "void", "struct", "long", "float", "double"};
   unordered_set<char> operators = {'+', '-', '*', '/', '=', '<', '>', '!', '&', '|'};
   unordered_set<char> punctuation = {',', ';', '(', ')', '{', '}', '[', ']'};

   bool isKeyword(const string &word)
   {
      return keywords.find(word) != keywords.end();
   }

   bool isOperator(char ch)
   {
      return operators.find(ch) != operators.end();
   }

   bool isPunctuation(char ch)
   {
      return punctuation.find(ch) != punctuation.end();
   }

   void addToSymbolTable(const string &identifier)
```

```
    {
        if (identifier == "main")
            return; // Do not add 'main' to the symbol table

        for (const auto &entry : symbolTable)
        {
            if (entry.name == identifier)
                return; // Avoid duplicates
        }
        symbolTable.push_back({identifier});
    }


    void reportLexicalError(const string &lexeme)
    {
        cout << "LEXICAL ERROR: Invalid lexeme \"" << lexeme << "\"\n";
    }

public:
    void stripComments(const string &sourceCode, string &cleanedCode)
    {
        size_t i = 0;
        while (i < sourceCode.length())
        {
            if (sourceCode[i] == '/' && sourceCode[i + 1] == '/')
            {
                // Single-line comment
```

```
            while (i < sourceCode.length() && sourceCode[i] != '\n')
                i++;
        }
        else if (sourceCode[i] == '/' && sourceCode[i + 1] == '*')
        {
            // Multi-line comment
            i += 2;
            while (i < sourceCode.length() && !(sourceCode[i] == '*' && sourceCode[i + 1] == '/'))
                i++;
            i += 2;
        }
        else
        {
            cleanedCode += sourceCode[i++];
        }
    }
}

void tokenize(const string &sourceCode)
{
    string lexeme;
    size_t i = 0;

    while (i < sourceCode.length())
    {
        if (isspace(sourceCode[i]))
        {
```

```
            i++;
            continue;
        }

        // Keywords and identifiers
        if (isalpha(sourceCode[i]) || sourceCode[i] == '_')
        {
            lexeme.clear();
            while (i < sourceCode.length() && (isalnum(sourceCode[i]) || sourceCode[i] == '_'))
            {
                lexeme += sourceCode[i++];
            }

            if (isKeyword(lexeme))
            {
                tokens.push_back({"Keyword", lexeme});
            }
            else
            {
                tokens.push_back({"Identifier", lexeme});
                addToSymbolTable(lexeme);
            }
            continue;
        }

        // Numeric constants
        if (isdigit(sourceCode[i]))
```

```
        {
            lexeme.clear();
            while (i < sourceCode.length() && isdigit(sourceCode[i]))
            {
                lexeme += sourceCode[i++];
            }

            if (i < sourceCode.length() && isalpha(sourceCode[i]))
            {
                // Handle invalid numeric constants like 7H
                while (i < sourceCode.length() && (isalnum(sourceCode[i]) || sourceCode[i] == '_'))
                {
                    lexeme += sourceCode[i++];
                }
                reportLexicalError(lexeme);
            }
            else
            {
                tokens.push_back({"Constant", lexeme});
            }
            continue;
        }

        // String and character literals
        if (sourceCode[i] == '\" || sourceCode[i] == '\'')
        {
            char quoteType = sourceCode[i++];
```

```
        lexeme = quoteType;

        while (i < sourceCode.length() && sourceCode[i] != quoteType)
        {
            lexeme += sourceCode[i++];
        }

        if (i < sourceCode.length() && sourceCode[i] == quoteType)
        {
            lexeme += sourceCode[i++];
            tokens.push_back({"String", lexeme});
        }
        else
        {
            reportLexicalError(lexeme); // Unterminated string/character literal
        }
        continue;
    }

    // Operators
    if (isOperator(sourceCode[i]))
    {
        tokens.push_back({"Operator", string(1, sourceCode[i])});
        i++;
        continue;
    }
```

```
        // Punctuation
        if (isPunctuation(sourceCode[i]))
        {
            tokens.push_back({"Punctuation", string(1, sourceCode[i])});
            i++;
            continue;
        }

        // Invalid characters
        lexeme.clear();
        while (i < sourceCode.length() && !isspace(sourceCode[i]) &&
               !isOperator(sourceCode[i]) && !isPunctuation(sourceCode[i]))
        {
            lexeme += sourceCode[i++];
        }
        reportLexicalError(lexeme);
    }
}

void displayTokens()
{
    cout << "TOKENS:\n";
    for (const auto &token : tokens)
    {
        cout << token.type << ": " << token.value << endl;
    }
}
```

```
    void displaySymbolTable()
    {
        cout << "\nSYMBOL TABLE ENTRIES:\n";
        for (size_t i = 0; i < symbolTable.size(); i++)
        {
            cout << i + 1 << ") " << symbolTable[i].name << endl;
        }
    }
};

int main()
{
    string inputFileName, sourceCode, cleanedCode;
    ifstream inputFile;

    cout << "Enter the input file name: ";
    cin >> inputFileName;

    inputFile.open(inputFileName);
    if (!inputFile.is_open())
    {
        cerr << "Error: Could not open file " << inputFileName << endl;
        return 1;
    }

    sourceCode.assign((istreambuf_iterator<char>(inputFile)),
istreambuf_iterator<char>());
```

```
    inputFile.close();

    LexicalAnalyzer lexer;

    // Strip comments
    lexer.stripComments(sourceCode, cleanedCode);

    // Tokenize
    lexer.tokenize(cleanedCode);

    // Display results
    lexer.displayTokens();
    lexer.displaySymbolTable();

    return 0;
}

/*
 */
```

OUTPUT:

```
Enter the input file name: input.txt
LEXICAL ERROR: Invalid lexeme "."
TOKENS:
Identifier: The
Constant: 45
Identifier: is
Identifier: odd
Identifier: number

SYMBOL TABLE ENTRIES:
1) The
2) is
3) odd
4) number

Process returned 0 (0x0)   execution time : 3.628 s
Press any key to continue.
```