Experiment No.5
Study and Implementation of Lexical Analyzer.
Date of Performance: 24-02-2025
Date of Submission: 03-03-2025

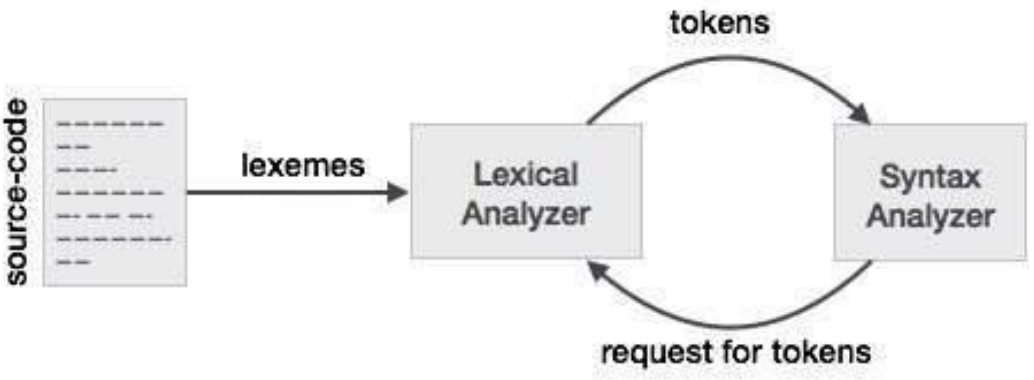
Aim: Study and Implementation of Lexical Analyzer.

Objective: To study and implement lexical analyzer program for identifying tokens from given input source program.

Theory:

Lexical analysis is the first phase of a compiler. It takes the modified source code from language preprocessors that are written in the form of sentences. The lexical analyzer breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code.

If the lexical analyzer finds a token invalid, it generates an error. The lexical analyzer works closely with the syntax analyzer. It reads character streams from the source code, checks for legal tokens, and passes the data to the syntax analyzer when it demands.



Lexical analysis : process of taking an input string of characters (such as the source code of a computer program) and producing a sequence of symbols called lexical tokens, or just tokens, which may be handled more easily by a parser. The pattern matches each string in the set.

Tokens:

Lexemes are said to be a sequence of characters (alphanumeric) in a token. There are some predefined rules for every lexeme to be identified as a valid token. These rules are defined by grammar rules, by means of a pattern. A pattern explains what a token can be, and these patterns are defined by means of regular expressions.

In programming language, keywords, constants, identifiers, strings, numbers, operators and punctuations symbols can be considered as tokens.

For example, in C language, the variable declaration line
int value = 100;
contains the tokens:
int (keyword), value (identifier), = (operator), 100 (constant) and; (symbol).

Specifications of Tokens:

Let us understand how the language theory undertakes the following terms:

Alphabets

Any finite set of symbols {0,1} is a set of binary alphabets,
{0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F} is a set of Hexadecimal alphabets, {a-z, A-Z} is a set of English language alphabets.

Strings

Any finite sequence of alphabets is called a string. Length of the string is the total number of occurrence of alphabets, e.g., the length of the string tutorialspoint is 14 and is denoted by |tutorialspoint| = 14. A string having no alphabets, i.e. a string of zero length is known as an empty string and is denoted by ε (epsilon).

Special Symbols

A typical high-level language contains the following symbols:-

Arithmetic Symbols	Addition(+), Subtraction(-), Modulo(%), Multiplication(*), Division(/)
Punctuation	Comma(,), Semicolon(;), Dot(.), Arrow(->)
Assignment	=
Special Assignment	+=, /=, *=, -=
Comparison	==, !=, <, <=, >, >=
Preprocessor	#
Location Specifier	&
Logical	&, &&, |, ||, !

Shift Operator	>>, >>>, <<, <<<

Language

A language is considered as a finite set of strings over some finite set of alphabets. Computer languages are considered as finite sets, and mathematically set operations can be performed on them. Finite languages can be described by means of regular expressions.

Code:

```
#include <stdio.h>

#include <ctype.h>

#include <string.h>


const char *keywords[] = {"int", "float", "char", "double", "return", "if", "else", "while", "for", "do", "break", "continue", "switch", "case", "default", "void", "str"};

const int num_keywords = sizeof(keywords) / sizeof(keywords[0]);


int isKeyword(const char *word) {
    for (int i = 0; i < num_keywords; i++) {
        if (strcmp(word, keywords[i]) == 0) return 1;
    }
    return 0;
}


int isOperator(char ch) {
    char operators[] = "+-*/%=<>&|!";
    for (int i = 0; i < strlen(operators); i++) {
        if (ch == operators[i]) return 1;
```

```
    }
    return 0;
}

int isPunctuation(char ch) {
    char punctuations[] = "();{},:[]";
    for (int i = 0; i < strlen(punctuations); i++) {
        if (ch == punctuations[i]) return 1;
    }
    return 0;
}

typedef struct {
    char token[50];
    int count;
} Token;

Token keywords_list[50], identifiers_list[50], constants_list[50], operators_list[50], punctuations_list[50], strings_list[50];

int keyword_count = 0, identifier_count = 0, constant_count = 0, operator_count = 0, punctuation_count = 0, string_count = 0;

void addToken(Token list[], int *count, const char *token) {
    for (int i = 0; i < *count; i++) {
        if (strcmp(list[i].token, token) == 0) {
            list[i].count++;
            return;
```

```
        }
    }
    strcpy(list[*count].token, token);
    list[*count].count = 1;
    (*count)++;
}

void lexicalAnalyzer(const char *code) {
    int i = 0;
    while (code[i] != '\0') {
        if (isspace(code[i])) {
            i++;
            continue;
        }

        // Identifier or Keyword
        if (isalpha(code[i]) || code[i] == '_') {
            char buffer[50];
            int j = 0;
            while (isalnum(code[i]) || code[i] == '_') {
                buffer[j++] = code[i++];
            }
            buffer[j] = '\0';
            if (isKeyword(buffer)) addToken(keywords_list, &keyword_count, buffer);
            else addToken(identifiers_list, &identifier_count, buffer);
        }
```

```
// Numeric constant
else if (isdigit(code[i])) {
    char buffer[50];
    int j = 0;
    while (isdigit(code[i]) || code[i] == '.') {
        buffer[j++] = code[i++];
    }
    buffer[j] = '\0';
    addToken(constants_list, &constant_count, buffer);
}

// String literal
else if (code[i] == '"') {
    char buffer[50];
    int j = 0;
    buffer[j++] = code[i++]; // opening quote
    while (code[i] != '"' && code[i] != '\0') {
        buffer[j++] = code[i++];
    }
    if (code[i] == '"') buffer[j++] = code[i++];
    buffer[j] = '\0';
    addToken(strings_list, &string_count, buffer);
}

// Operators (basic 1-char only for simplicity, extendable)
```

```
        else if (isOperator(code[i])) {
            char buffer[3] = {code[i], '\0'};
              if ((code[i] == '=' || code[i] == '<' || code[i] == '>' || code[i] == '!') && code[i + 1] ==
'=') {
                buffer[1] = code[i + 1];
                buffer[2] = '\0';
                i += 2;
            } else {
                i++;
            }
            addToken(operators_list, &operator_count, buffer);
        }

        // Punctuation
        else if (isPunctuation(code[i])) {
            char buffer[2] = {code[i], '\0'};
            addToken(punctuations_list, &punctuation_count, buffer);
            i++;
        }

        // Unknown character
        else {
            i++;
        }
    }

    // Output
```

```
printf("\n--- Lexical Analysis Result ---\n");
printf("Keywords: ");
for (int i = 0; i < keyword_count; i++) {
    printf("%s ", keywords_list[i].token);
}
printf("\nIdentifiers: ");
for (int i = 0; i < identifier_count; i++) {
    printf("%s ", identifiers_list[i].token);
}
printf("\nConstants: ");
for (int i = 0; i < constant_count; i++) {
    printf("%s ", constants_list[i].token);
}
printf("\nOperators: ");
for (int i = 0; i < operator_count; i++) {
    printf("%s ", operators_list[i].token);
}
printf("\nPunctuations: ");
for (int i = 0; i < punctuation_count; i++) {
    printf("%s ", punctuations_list[i].token);
}
printf("\nStrings: ");
for (int i = 0; i < string_count; i++) {
    printf("%s ", strings_list[i].token);
}
printf("\n");
```

```
}

int main() {
    const char code[] = "int main() { int a, b; a = 10; char str[] = \"hello\"; return 0; }";
    printf("Input Code: %s\n", code);
    lexicalAnalyzer(code);
    return 0;
}
```

Output:

Output Clear

```
Input Code: int main() { int a, b; a = 10; char str[] = "hello"; return 0; }

--- Lexical Analysis Result ---
Keywords: int char str return
Identifiers: main a b
Constants: 10 0
Operators: =
Punctuations: ( ) { , ; [ ] }
Strings: "hello"


=== Code Execution Successful ===
```

Conclusion:

The lexical analyzer program successfully scans the source code and breaks it into meaningful tokens such as keywords, identifiers, constants, operators, punctuations, and string literals. Each token is accurately classified under its respective lexical category. The program utilizes character-by-character analysis and pattern matching to identify these tokens. It effectively distinguishes between reserved keywords and user-defined identifiers. This implementation demonstrates how the lexical analysis phase simplifies the input code for further syntactic and semantic analysis, forming the foundation for compiler design and enhancing understanding of language structure.