MISR UNIVERSITY
FOR SCIENCE & TECHNOLOGY
College of Information Technology

جـــامعـــة مصـــــر
للعلـــوم والتكنـــولـــوجيـــا
كليـــة تكنولوجيـــا المعلومـــات

LEXICAL ANALYZER

Build Scanner

Prepared By

Youssef Mohamed Youssef

Under Supervision

Nehal Abdelsalam
Menna

1. Introduction

This code implements a lexical analyzer, which is the first step in the process of compiling high-level programming languages. The purpose of a lexical analyzer is to break the source code into tokens—small, meaningful units such as keywords, identifiers, operators, and symbols—that can be further processed by the rest of the compiler. It is the phase where the program reads the raw input and extracts the components needed for understanding and translating the code.

This specific implementation is tailored to handle arithmetic expressions, a subset of programming language constructs, which include:

Identifiers (e.g., variable names),
Integer literals (e.g., 123, 456),
Arithmetic operators (e.g., +, -, *, /),
Parentheses for grouping expressions (e.g., (,)).

The lexer operates by scanning the input string, classifying each character as one of several predefined categories: letters, digits, or unknown (operators or symbols). It then generates tokens based on the recognized patterns. For example, it might encounter a sequence of letters like var1, recognize it as an identifier, and generate the corresponding token for IDENT. If it encounters a number like 42, it will generate an integer literal token.

Additionally, the program demonstrates:

Character Classification: It distinguishes between letters, digits, and unknown characters (such as symbols and operators).

Token Creation: It groups characters into tokens like identifiers, operators, and integer literals.

Whitespace Skipping: The program ignores spaces and tabs, which are not considered valid tokens but are necessary for proper input formatting.

End of File Handling: When the program reaches the end of the input string, it recognizes this by assigning the END_OF_FILE token.

By outputting the token and lexeme, it allows developers to monitor how the input is being parsed. The lexeme is the actual string of characters that matched the token pattern, while the token represents the type of entity found (e.g., addition operator, integer literal). This kind of feedback is essential when debugging or improving the analyzer.

In summary, this program demonstrates the very first step in compiler construction, lexical analysis, where we read and break down source code into tokens for further interpretation and processing. It serves as the groundwork for more complex stages of a compiler, such as syntax analysis and semantic analysis.

1.1. Phases of Compiler

- Lexical Analysis.
- Syntactic Analysis or Parsing.
- Semantic Analysis.
- Intermediate Code Generation.
- Code Optimization.
- Code Generation.

2. Lexical Analyzer

- Suppose we need a lexical analyzer that recognizes only arithmetic expressions, including variable names and integer literals as operands.

- Next, we define some utility subprograms for the common tasks inside the lexical analyzer.

- First, we need a subprogram, which we can name getChar, that has several duties. When called, getChar gets the next character of input from the input program and puts it in the global variable nextChar. getChar also must determine the character class of the input character and put it in the global variable charClass.

- The lexeme being built by the lexical analyzer, which could be implemented as a character string or an array, will be named lexeme.

- We implement the process of putting the character in nextChar into the string array lexeme in a subprogram named addChar.

- This subprogram must be explicitly called because programs include some characters that need not be put in lexeme, for example the white-space characters between lexemes.

- In a more realistic lexical analyzer, comments also would not be placed in lexeme.

- When the lexical analyzer is called, it is convenient if the next character of input is
the first character of the next lexeme.

- Because of this, a function named getNonBlank is used to skip white space every
time the analyzer is called .

- Finally, a subprogram named lookup is needed to compute the token code for the
single-character tokens.

- In our example, these are parentheses and the arithmetic operators. Token codes
are numbers arbitrarily assigned to tokens by the compiler writer.

- Names and reserved words in programs have similar patterns.

- Although it is possible to build a state diagram to recognize every specific reserved
word of aprogramming language, that would result in a prohibitively large state

diagram.

- It is much simpler and faster to have the lexical analyzer recognize names and reserved words with the same pattern and use a lookup in a table of reserved words to determine which names are reserved words.
- Using this approach considers reserved words to be exceptions in the names token category.
- A lexical analyzer often is responsible for the initial construction of the symbol table, which acts as a database of names for the compiler.
- The entries in the symbol table store information about user-defined names, as well as the attributes of the names.
- For example, if the name is that of a variable, the variable’s type is one of its attributes that will be stored in the symbol table.
- Names are usually placed in the symbol table by the lexical analyzer.
- The attributes of a name are usually put in the symbol table by some part of the compiler that is subsequent to the actions of the lexical analyzer.

3. Software Tools

3.1. Computer Program

- Visual studio community 2022

3.2. Programming Language

- **C++**

C++ Language and Its Features

C++ is a high-level programming language developed by Bjarne Stroustrup in the early 1980s as an enhancement to C. It is known for combining both procedural and object-oriented programming paradigms, which makes it a versatile language suitable for a wide range of applications.

Key Features of C++:

Object-Oriented Programming (OOP): C++ supports OOP, which helps in organizing and managing code efficiently using concepts like classes, inheritance, and polymorphism.

High Performance: C++ allows developers to directly manipulate memory and interact with hardware, making it ideal for applications that require high performance, such as gaming and system software.

Memory Management: C++ gives programmers control over memory allocation and deallocation using pointers, allowing efficient resource management.

C Compatibility: C++ is compatible with C, so existing C code can be reused in C++ programs.

Libraries: C++ provides powerful libraries like the Standard Template Library (STL), which offers reusable data structures and algorithms.

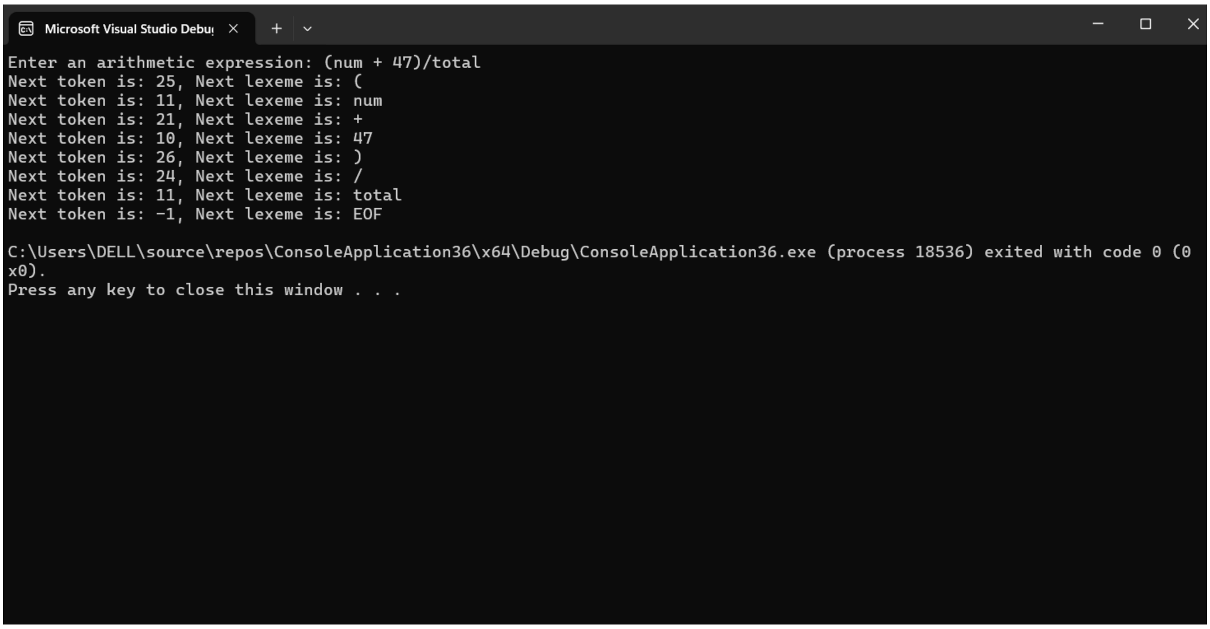
Cross-Platform: C++ can be used to develop applications that run across multiple platforms, such as Windows, Linux, and Mac.

4. Implementation of a Lexical Analyzer

```
#include <iostream>  // For input/output operations
#include <string>    // To handle strings
#include <cctype>    // To check character types (e.g., isalpha, isdigit)

using namespace std;

// Character classes
#define LETTER 0        // Character class for letters (a-z, A-Z)
#define DIGIT 1         // Character class for digits (0-9)
#define UNKNOWN 99      // Character class for unknown characters (operators, etc.)
#define END_OF_FILE -1  // Character class for the end of file/input

// Token codes
#define INT_LIT 10      // Token code for integer literals
#define IDENT 11        // Token code for identifiers (e.g., variable names)
#define ASSIGN_OP 20    // Token code for assignment operator '='
#define ADD_OP 21       // Token code for addition operator '+'
```

```
#define SUB_OP 22       // Token code for subtraction operator '-'
#define MULT_OP 23      // Token code for multiplication operator '*'
#define DIV_OP 24       // Token code for division operator '/'
#define LEFT_PAREN 25   // Token code for left parenthesis '('
#define RIGHT_PAREN 26  // Token code for right parenthesis ')'

// Global variables
string input;   // String to hold the input text
size_t pos = 0;  // Position index to keep track of the current character in the input
char nextChar;   // Holds the current character being processed
int charClass;   // Holds the character class (LETTER, DIGIT, UNKNOWN, END_OF_FILE)
string lexeme;   // Holds the current lexeme (substring of the input)
int nextToken;   // Holds the token code for the current lexeme

// Function declarations
void getChar();      // Function to get the next character from input
void addChar();      // Function to add the current character to the lexeme
void getNonBlank();  // Function to skip over any whitespace characters
int lookup(char ch); // Function to lookup operators and return corresponding tokens
int lex();           // Main function to perform lexical analysis and identify tokens

// Main driver function
int main() {
    // Prompt the user for input and read a line of input into the 'input' string
    cout << "Enter an arithmetic expression: ";
    getline(cin, input);  // Read input from the user

    getChar();  // Initialize by getting the first character from the input
    do {
        lex();  // Process the input and identify tokens
    } while (nextToken != END_OF_FILE);  // Continue until we reach the end of the input

    return 0;  // Return 0 to indicate successful execution
}

// Function to get the next character from the input and classify it
void getChar() {
    // Check if there are more characters in the input
    if (pos < input.length()) {
        // Get the next character from the input
        nextChar = input[pos++];  // Increment the position after getting the character

        // Classify the character as a letter, digit, or unknown
        if (isalpha(nextChar)) {
            charClass = LETTER;  // Letter characters (a-z, A-Z)
        }
        else if (isdigit(nextChar)) {
            charClass = DIGIT;   // Digit characters (0-9)
        }
        else {
            charClass = UNKNOWN; // Non-alphanumeric characters (operators, etc.)
        }
    }
    else {
        charClass = END_OF_FILE; // Set to END_OF_FILE when we reach the end of input
    }
}

// Function to add the current character to the lexeme
void addChar() {
    lexeme += nextChar;  // Append the current character to the lexeme string
}

// Function to skip over whitespace characters (spaces, tabs, etc.)
void getNonBlank() {
    // Continue calling getChar until we find a non-whitespace character
    while (isspace(nextChar)) {
        getChar();  // Skip over spaces or tabs
    }
}

// Function to lookup operators and parentheses, returning the appropriate token
int lookup(char ch) {
    // Match each operator and return the corresponding token code
    switch (ch) {
    case '(': addChar(); return LEFT_PAREN;  // Left parenthesis
    case ')': addChar(); return RIGHT_PAREN; // Right parenthesis
    case '+': addChar(); return ADD_OP;     // Addition operator
    case '-': addChar(); return SUB_OP;     // Subtraction operator
    case '*': addChar(); return MULT_OP;    // Multiplication operator
    case '/': addChar(); return DIV_OP;     // Division operator
    case '=': addChar(); return ASSIGN_OP;  // Assignment operator '='
    default:  addChar(); return END_OF_FILE; // Unknown character or end of input
    }
}

// Function for lexical analysis to identify tokens
int lex() {
    lexeme = "";  // Reset lexeme before processing each token
    getNonBlank();  // Skip any whitespace characters

    switch (charClass) {
    case LETTER:
        // If the character is a letter, start forming an identifier
        addChar();   // Add the letter to the lexeme
        getChar();   // Get the next character
        // Continue adding characters to the lexeme as long as they are letters or digits
        while (charClass == LETTER || charClass == DIGIT) {
            addChar();  // Add character to lexeme
            getChar();  // Get next character
        }
        nextToken = IDENT;  // Set the token to IDENT (identifier)
        break;

    case DIGIT:
        // If the character is a digit, start forming an integer literal
        addChar();   // Add the digit to the lexeme
        getChar();   // Get the next character
        // Continue adding digits to the lexeme
        while (charClass == DIGIT) {
            addChar();  // Add character to lexeme
            getChar();  // Get next character
        }
        nextToken = INT_LIT;  // Set the token to INT_LIT (integer literal)
        break;

    case UNKNOWN:
        // If the character is an unknown operator or symbol, lookup its token
        nextToken = lookup(nextChar);  // Look up operator
        getChar();  // Get the next character
        break;

    case END_OF_FILE:
        // If we've reached the end of input, set the token to END_OF_FILE
        lexeme = "EOF";  // Set lexeme to "EOF"
        nextToken = END_OF_FILE;  // Set the token to END_OF_FILE
        break;
    }

    // Print the token and lexeme
    cout << "Next token is: " << nextToken << ", Next lexeme is: " << lexeme << endl;

    return nextToken;  // Return the identified token
}
```

5. OutPut:



```
Enter an arithmetic expression: (num + 47)/total
Next token is: 25, Next lexeme is: (
Next token is: 11, Next lexeme is: num
Next token is: 21, Next lexeme is: +
Next token is: 10, Next lexeme is: 47
Next token is: 26, Next lexeme is: )
Next token is: 24, Next lexeme is: /
Next token is: 11, Next lexeme is: total
Next token is: -1, Next lexeme is: EOF

C:\Users\DELL\source\repos\ConsoleApplication36\x64\Debug\ConsoleApplication36.exe (process 18536) exited with code 0 (0
x0).
Press any key to close this window . . .
```

6. References :

- Google

- Book of subject

- Chatgpt