A black background with grey leaves

AI-generated content may be incorrect.

Lexical Analyzer

Build Scanner

A grey logo on a black background

AI-generated content may be incorrect.

**Prepared By**

Mahmoud Karem Mahmoud

200045706

**Under Supervision**

Nehal Abdelsalam

Menna

1. **Introduction :**

**This code simulates the first phase of a compiler, which is called Lexical Analysis. It’s a fundamental step in the compilation process, where the compiler reads the source code and breaks it down into smaller pieces called tokens. These tokens are things like variable names, numbers, operators, and symbols, which the compiler can then use in later stages like syntax and semantic analysis.**

**In this program, I take a simple arithmetic expression entered by the user (such as a = b + 10) and process it character by character. The goal is to recognize each component and classify it as:**

* **An identifier (like a, b)**
* **An integer literal (like 10)**
* **An operator or symbol (like =, +, \*, /)**

**I structured the code in a clean and organized way, with each function having a clear responsibility:**

* **getChar() reads the current character and determines its class (letter, digit, or symbol).**
* **addChar() adds the current character to the lexeme being formed.**
* **getNonBlank() skips white spaces so we only process meaningful characters.**
* **lookup() identifies symbols or operators and returns the corresponding token.**
* **lex() is the main function that brings everything together it determines the type of each lexeme and prints out the token and the lexeme.**

**The program handles different situations like:**

* **Reaching the end of input (EOF)**
* **Dealing with unknown characters**
* **Distinguishing between identifiers, numbers, and symbols**

**The purpose of this code is to practically understand how a compiler takes its first step in analyzing source code, and how it breaks down what we write into small pieces (tokens) that can be used later in the syntax and semantic analysis phases**

* 1. **Phases of Compiler :**
* **Lexical Analysis.**
* **Syntactic Analysis or Parsing.**
* **Semantic Analysis.**
* **Intermediate Code Generation.**
* **Code Optimization.**
* **Code Generation.**

1. **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.**

1. **Software Tools**
   1. **Computer Program :**

* **Visual Studio Community 2022**
  1. **Programming Language :**
* **C++**

**I chose C++ because it offers fine control over character processing and memory management. Since lexical analysis involves handling characters one by one, C++'s ability to manipulate strings and check character types efficiently made it a great choice. It also allows me to write structured, modular code using functions, which made the program easy to organize and understand. Additionally, C++ is a compiled language, similar to the languages that compilers work with, making it a natural fit for simulating the compilation proces**

1. **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

#define DIGIT 1 // Character class for digits

#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);

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++];

// Classify the character as a letter, digit, or unknown

if (isalpha(nextChar)) {

charClass = LETTER; // Letter characters

}

else if (isdigit(nextChar)) {

charClass = DIGIT; // Digit characters

}

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();

}

}

// 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();

getChar();

// Continue adding characters to the lexeme as long as they are letters or digits

while (charClass == LETTER || charClass == DIGIT) {

addChar();

getChar();

}

nextToken = IDENT; // Set the token to IDENT (identifier)

break;

case DIGIT:

// If the character is a digit, start forming an integer literal

addChar();

getChar();

// Continue adding digits to the lexeme

while (charClass == DIGIT) {

addChar();

getChar();

}

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);

getChar();

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;

}

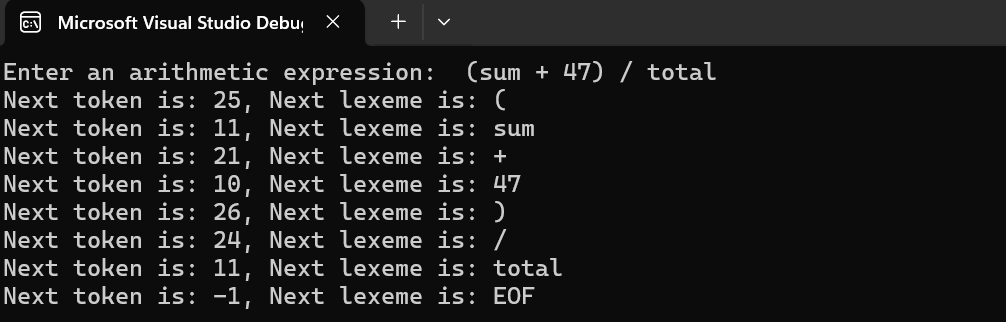
// Print the token and lexeme

cout << "Next token is: " << nextToken << ", Next lexeme is: " << lexeme << endl;

return nextToken; // Return the identified token

}

1. **Output :**

****

1. **References :**

* **Book of the subject of concept of programming language**
* **Google**