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**

Student Name Student ID

**Ahmed saleh**

**Id/200042639**

**Under Supervision**

Name of Doctor/Nehal

Name of T. A./fares

1. **Introduction**

This C++ program implements a Lexical Analyzer (Lexer), which is a fundamental component of a

compiler or interpreter. The primary function of the lexer is to read an input string (typically a line of source code) and convert it into a sequence of tokens, each representing a meaningful element such as an identifier, operator, number, or punctuation mark.

In this program, the lexer reads arithmetic expressions provided by the user, and identifies various token types such as:

* Identifiers (e.g., variable names like sum)
* Integer literals (e.g., 9, 2, 5)
* Operators (e.g., +, -, \*, /)
* Delimiters (e.g., parentheses ( ) and semicolon ;)
* Assignment operator (=)

Each token is printed along with its corresponding lexeme (the exact substring from the input). The implementation uses a finite-state machine approach to classify input characters and group them into valid tokens based on their character class (letters, digits, or special symbols).

This program is a simplified version of what a real compiler does in its lexical analysis phase, and serves as an educational tool for understanding how programming languages are parsed at the syntactic level.

* 1. **Phases of Compiler**

A compiler typically consists of several phases that work together to translate high-level source code into machine code. The main phases are:

1. Lexical Analysis (Implemented in the code ✅)

* What it does:  
  Reads the raw input (source code) and splits it into tokens — such as identifiers (sum), operators (+, \*), numbers (9, 5), and delimiters (;, (, )).
* Role in your code:  
  This is the exact phase implemented in your C++ code. The class Lexer performs lexical analysis, categorizes each part of the input, and outputs tokens with their lexemes.

2. Syntax Analysis (Parsing) ❌

* What it does:  
  Takes the sequence of tokens from the lexer and builds a parse tree or checks if the sequence follows the grammar of the language.
* In your code:  
  This part is not implemented. The lexer just prints tokens but doesn't check if the expression is valid (e.g., whether sum = 9 + ; is correct or not).

3. Semantic Analysis ❌

* What it does:  
  Ensures that the syntax makes sense semantically. For example, checks if a variable is declared before use, or if types match (e.g., you can't add a string to an integer).
* In your code:  
  Not present.

4. Intermediate Code Generation ❌

* What it does:  
  Converts the syntax tree into an intermediate representation like three-address code.
* In your code:  
  Not included.

5. Code Optimization ❌

* What it does:  
  Improves the intermediate code (e.g., removes unnecessary calculations).
* In your code:  
  Not included.

6. Code Generation ❌

* What it does:  
  Generates the final machine code or assembly from the intermediate representation.
* In your code:  
  Not included.

7. Symbol Table Management & Error Handling (Partial)

* Symbol Table: ❌  
  Your code doesn't maintain a symbol table (a data structure to store variable names, types, etc.)
* Error Handling: ✅ جزئياً  
  It prints an error if a lexeme is too long, but doesn't handle things like invalid symbols properly.

✅ Summary:

In the context of compiler phases, your C++ code only implements Phase 1: Lexical Analysis, and it does so correctly and effectively for basic arithmetic expressions.

1. **Lexical Analyzer**

A Lexical Analyzer, also known as a Lexer or Scanner, is the first phase of a compiler. Its job is to read the source code (as plain text) and convert it into a sequence of tokens, which are meaningful units in the programming language (like keywords, identifiers, numbers, operators, etc.).

✅ Role of the Lexical Analyzer in the Code

In the provided C++ code, the class Lexer performs the lexical analysis. Here’s how it works:

🔸 Key Responsibilities of the Lexer:

1. Reading Input:
   * The lexer reads the entire line of input using getline() from the user.
   * The input is stored in a stream and processed character by character.
2. Character Classification:
   * Each character is classified as:
     + LETTER (e.g., s, u, m)
     + DIGIT (e.g., 0-9)
     + UNKNOWN (symbols like +, \*, =, etc.)
     + END\_OF\_FILE (when input ends)
3. Lexeme Construction:
   * The addChar() function builds a lexeme (the actual string of characters forming a token).
   * For example, sum is a lexeme and is categorized as an IDENT (identifier).
4. Token Recognition:
   * Based on the character class, the lexer decides:
     + If it's an identifier → token is IDENT (code 11)
     + If it's a number → token is INT\_LIT (code 10)
     + If it's a symbol like + or \* → matched using lookup() to get correct token code
     + End of input → token is EOF\_TOKEN
5. Token Output:
   * Each token is printed with its corresponding lexeme:

bash

نسختحرير

Next token is: 11, Next lexeme is sum

1. Whitespace Skipping:
   * Spaces are ignored using getNonBlank() before tokenizing starts.

🔸 Example Output

If the input is:

bash

نسختحرير

( sum = 9 + 2 \* 5; )

The output will be:

mathematica

نسختحرير

Next token is: 25, Next lexeme is (

Next token is: 11, Next lexeme is sum

Next token is: 20, Next lexeme is =

Next token is: 10, Next lexeme is 9

Next token is: 21, Next lexeme is +

Next token is: 10, Next lexeme is 2

Next token is: 23, Next lexeme is \*

Next token is: 10, Next lexeme is 5

Next token is: 27, Next lexeme is ;

Next token is: 26, Next lexeme is )

Next token is: -1, Next lexeme is EOF

🔸 Why It Matters:

This lexical analyzer is a basic yet essential component of a compiler. It helps break down the source code into manageable pieces (tokens) so that the next phases (like parsing and semantic analysis) can understand and process the code correctly.

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

A Computer Program is a set of instructions written in a programming language that tells a computer how to perform a specific task or solve a problem. These instructions are written in a way that the computer can understand, process, and execute.

🔸 Key Characteristics of a Computer Program:

* Written using programming languages such as C++, Python, Java, etc.
* Follows logical steps to process input and produce output.
* Can automate repetitive tasks, perform calculations, manage data, or control hardware.
* Requires compilation or interpretation before it can be executed by the computer.

🔸 Relation to the Given Code:

The code you provided is a C++ computer program that performs lexical analysis. It consists of:

* A class Lexer that contains logic to analyze input.
* A main() function that serves as the entry point of the program.
* It reads an arithmetic expression from the user and breaks it down into tokens.
* Each token is identified and printed, helping simulate how a compiler begins the process of understanding source code.

So, this program is not just a random collection of code — it's a purposeful, well-structured computer program designed to implement part of a compiler's workflow.

✅ Summary:

A Computer Program is a tool created by humans to give precise instructions to computers.  
In this case, the program helps in breaking down expressions into tokens, which is a crucial step in program translation and compilation.

* 1. **Programming Language**

A Programming Language is a formal language used to write instructions that a computer can execute. It provides the syntax and semantics necessary for developers to build software that interacts with hardware, performs calculations, processes data, or automates tasks.

🔸 Types of Programming Languages:

* Low-level languages (e.g., Assembly): Close to machine code, harder to write/read.
* High-level languages (e.g., C++, Python, Java): Easier for humans to read and write, and more abstracted from hardware details.

🔸 Why Programming Languages Matter:

* They are the foundation for software development.
* Allow humans to communicate complex logic to computers in a readable and structured way.
* Each language has its own rules (syntax) and features, making it suitable for different tasks (e.g., system programming, web development, data analysis).

🔸 Relation to the Given Code:

The code you provided is written in C++, which is a high-level, general-purpose programming language known for:

* Performance and efficiency.
* Object-oriented programming features (like classes).
* Being widely used in system software, compilers, and game development.

In your code:

* You defined a class Lexer — an object-oriented feature of C++.
* Used standard input/output (cin, cout) to interact with the user.
* Used control structures (switch, while, if) to implement program logic.

✅ Summary:

A Programming Language is the bridge between human logic and machine execution.  
Your use of C++ in the lexical analyzer program shows how powerful programming languages can be in creating tools like compilers that process and understand other programs.

1. **Implementation of a Lexical Analyzer**

**#include <iostream>**

**#include <sstream>**

**#include <cctype>**

**#include <string>**

**using namespace std;**

**enum CharClass {**

**LETTER, DIGIT, UNKNOWN, END\_OF\_FILE**

**};**

**enum TokenCode {**

**INT\_LIT = 10, IDENT = 11, ASSIGN\_OP = 20, ADD\_OP = 21,**

**SUB\_OP = 22, MULT\_OP = 23, DIV\_OP = 24, LEFT\_PAREN = 25,**

**RIGHT\_PAREN = 26, SEMICOLON = 27, EOF\_TOKEN = -1**

**};**

**class Lexer {**

**private:**

**CharClass charClass;**

**string lexeme;**

**char nextChar;**

**int nextToken;**

**istringstream input;**

**public:**

**Lexer(const string& data) : input(data) {**

**getChar();**

**}**

**lexeme**

**void addChar() {**

**if (lexeme.length() <= 98)**

**lexeme += nextChar;**

**else**

**cout << "Error - lexeme is too long" << endl;**

**}**

**void getChar() {**

**if (input.get(nextChar)) {**

**if (isalpha(nextChar))**

**charClass = LETTER;**

**else if (isdigit(nextChar))**

**charClass = DIGIT;**

**else**

**charClass = UNKNOWN;**

**} else {**

**charClass = END\_OF\_FILE;**

**}**

**}**

**void getNonBlank() {**

**while (isspace(nextChar))**

**getChar();**

**}**

**int lookup(char ch) {**

**switch (ch) {**

**case '(': addChar(); return LEFT\_PAREN;**

**case ')': addChar(); return RIGHT\_PAREN;**

**case '+': addChar(); return ADD\_OP;**

**case '-': addChar(); return SUB\_OP;**

**case '\*': addChar(); return MULT\_OP;**

**case '/': addChar(); return DIV\_OP;**

**case ';': addChar(); return SEMICOLON;**

**case '=': addChar(); return ASSIGN\_OP;**

**default: addChar(); return EOF\_TOKEN;**

**}**

**}**

**int lex() {**

**lexeme.clear();**

**getNonBlank();**

**switch (charClass) {**

**case LETTER:**

**addChar(); getChar();**

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

**addChar(); getChar();**

**}**

**nextToken = IDENT;**

**break;**

**case DIGIT:**

**addChar(); getChar();**

**while (charClass == DIGIT) {**

**addChar(); getChar();**

**}**

**nextToken = INT\_LIT;**

**break;**

**case UNKNOWN:**

**nextToken = lookup(nextChar);**

**getChar();**

**break;**

**case END\_OF\_FILE:**

**nextToken = EOF\_TOKEN;**

**lexeme = "EOF";**

**break;**

**}**

**lexeme**

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

**return nextToken;**

**}**

**void analyze() {**

**while (nextToken != EOF\_TOKEN) {**

**lex();**

**}**

**}**

**};**

**int main() {**

**string inputLine;**

**cout << "Enter an arithmetic expression like: ( sum = 9 + 2 \* 5; ): ";**

**getline(cin, inputLine);**

**Lexer lexer(inputLine);**

**lexer.analyze();**

**return 0;**

**}**

# Explain the code

**#include <iostream>**

**#include <sstream>**

**#include <cctype>**

**#include <string>**

**🔹 Includes standard libraries:**

**iostream: for input/output operations (e.g., cout, cin)**

**sstream: to use string streams like istringstream**

**cctype: for character classification functions (e.g., isalpha, isdigit)**

**string: to use the string type**

**using namespace std;**

**🔹 Allows you to use standard names like string, cin, cout without prefixing them with std::.**

**تحرير**

**enum CharClass {**

**LETTER, DIGIT, UNKNOWN, END\_OF\_FILE**

**};**

**🔹 Defines an enumeration CharClass to classify characters:**

**LETTER: alphabetic characters**

**DIGIT: numeric digits**

**UNKNOWN: symbols (e.g., +, \*)**

**END\_OF\_FILE: when input ends**

**enum TokenCode {**

**INT\_LIT = 10, IDENT = 11, ASSIGN\_OP = 20, ADD\_OP = 21,**

**SUB\_OP = 22, MULT\_OP = 23, DIV\_OP = 24, LEFT\_PAREN = 25,**

**RIGHT\_PAREN = 26, SEMICOLON = 27, EOF\_TOKEN = -1**

**};**

**🔹 Defines token types with numerical codes:**

**INT\_LIT: Integer literal (e.g., 123)**

**IDENT: Identifier (e.g., variable name)**

**Operators and symbols like +, -, \*, /, =, ;**

**EOF\_TOKEN: Marks the end of input**

**class Lexer {**

**🔹 Starts the definition of a class called Lexer, which will handle lexical analysis.**

**private:**

**CharClass charClass;**

**string lexeme;**

**char nextChar;**

**int nextToken;**

**istringstream input;**

**🔹 Private members of the class:**

**charClass: current character's class**

**lexeme: the current token string**

**nextChar: current character being processed**

**nextToken: code of the identified token**

**input: stream to hold the input string**

**public:**

**Lexer(const string& data) : input(data) {**

**getChar();**

**}**

**🔹 Constructor that takes the input string, initializes the stream, and reads the first character using getChar().**

**void addChar() {**

**if (lexeme.length() <= 98)**

**lexeme += nextChar;**

**else**

**cout << "Error - lexeme is too long" << endl;**

**}**

**🔹 Adds nextChar to the current lexeme. If the lexeme gets too long (over 98 characters), it shows an error.**

**void getChar() {**

**if (input.get(nextChar)) {**

**if (isalpha(nextChar)) charClass = LETTER;**

**else if (isdigit(nextChar)) charClass = DIGIT;**

**else charClass = UNKNOWN;**

**} else {**

**charClass = END\_OF\_FILE;**

**}**

**}**

**🔹 Reads the next character from the stream and classifies it as a letter, digit, or unknown. If no characters are left, it's END\_OF\_FILE.**

**void getNonBlank() {**

**while (isspace(nextChar)) getChar();**

**}**

**🔹 Skips over any spaces until it finds a non-space character.**

**int lookup(char ch) {**

**switch (ch) {**

**case '(': addChar(); return LEFT\_PAREN;**

**case ')': addChar(); return RIGHT\_PAREN;**

**case '+': addChar(); return ADD\_OP;**

**case '-': addChar(); return SUB\_OP;**

**case '\*': addChar(); return MULT\_OP;**

**case '/': addChar(); return DIV\_OP;**

**case ';': addChar(); return SEMICOLON;**

**case '=': addChar(); return ASSIGN\_OP;**

**default: addChar(); return EOF\_TOKEN;**

**}**

**}**

**🔹 Matches special characters and returns the correct token code. Also adds the character to the lexeme.**

**int lex() {**

**lexeme.clear();**

**getNonBlank();**

**🔹 Starts the process of identifying the next token. Clears the old lexeme and skips spaces.**

**switch (charClass) {**

**case LETTER:**

**addChar(); getChar();**

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

**addChar(); getChar();**

**}**

**nextToken = IDENT;**

**break;**

**🔹 If the first character is a letter, continue adding letters/digits. This forms an identifier.**

**case DIGIT:**

**addChar(); getChar();**

**while (charClass == DIGIT) {**

**addChar(); getChar();**

**}**

**nextToken = INT\_LIT;**

**break;**

**🔹 If it's a digit, keep reading digits to form an integer literal.**

**case UNKNOWN:**

**nextToken = lookup(nextChar);**

**getChar();**

**break;**

**🔹 If the character is a symbol (e.g., +, -), use lookup() to get the corresponding token code.**

**case END\_OF\_FILE:**

**nextToken = EOF\_TOKEN;**

**lexeme = "EOF";**

**break;**

**}**

**🔹 If no characters are left, return EOF\_TOKEN.**

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

**return nextToken;**

**}**

**🔹 Prints the current token and lexeme, then returns the token code.**

**void analyze() {**

**while (nextToken != EOF\_TOKEN) {**

**lex();**

**}**

**}**

**🔹 Repeatedly calls lex() until the end of the input is reached.**

**int main() {**

**string inputLine;**

**cout << "Enter an arithmetic expression like : ( sum = 9 + 2 \* 5;): ";**

**getline(cin, inputLine);**

**🔹 Main function: prompts the user to enter an arithmetic expression and reads the entire line.**

**Lexer lexer(inputLine);**

**lexer.analyze();**

**return 0;**

**}**

**🔹 Creates a Lexer object with the input and calls analyze() to start lexical analysis. Ends the program.**

1. References
2. The code is from the book it was written in c language and I converted the same c code to a c ++ code .