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 /Mahmoud ahmed

Student ID/200040427

**Under Supervision**

Name of Doctor/Nehal

Name of T. A/Fares

**Introduction**

**A lexical analyzer, also known as a scanner, is a fundamental component of a compiler. Its primary role is to read the source code as a stream of characters and convert it into a stream of tokens. Tokens are sequences of characters that represent the smallest unit of meaning, such as keywords, identifiers, constants, operators, and punctuation.**

**Phases of Compiler**

**The compiler is typically divided into several phases:  
1. Lexical Analysis  
2. Syntax Analysis  
3. Semantic Analysis  
4. Intermediate Code Generation  
5. Code Optimization  
6. Code Generation  
7. Error Handling**

**Lexical Analyzer**

**The lexical analyzer reads the input program and divides it into tokens. It removes whitespace and comments and detects lexical errors. It plays a crucial role in ensuring the syntactic correctness of the source code.**

**Software Tools**

**The lexical analyzer can be implemented using tools like Lex (flex in modern systems), which generate scanners based on regular expressions and rules defined by the user.**

**Computer Program**

**The following is a simplified version of a lexical analyzer implemented in C++ that recognizes identifiers, numbers, and basic operators.**

**Programming Language**

**The implementation is done in C++ due to its efficiency and widespread usage in system-level programming.**

**Implementation of a Lexical Analyzer**

**```cpp  
// C++ Program to implement a lexical analyzer**

**#include <iostream>**

**#include <string>**

**#include <unordered\_map>**

**#include <vector>**

**using namespace std;**

**// Enum class to define different types of tokens**

**enum class TokenType {**

**KEYWORD,**

**IDENTIFIER,**

**INTEGER\_LITERAL,**

**FLOAT\_LITERAL,**

**OPERATOR,**

**PUNCTUATOR,**

**UNKNOWN**

**};**

**// Struct to represent a token with its type and value**

**struct Token {**

**TokenType type;**

**string value;**

**Token(TokenType t, const string& v)**

**: type(t)**

**, value(v)**

**{**

**}**

**};**

**// Class that implements the lexical analyzer**

**class LexicalAnalyzer {**

**private:**

**string input;**

**size\_t position;**

**unordered\_map<string, TokenType> keywords;**

**// Function to initialize the keywords map**

**void initKeywords()**

**{**

**keywords["int"] = TokenType::KEYWORD;**

**keywords["float"] = TokenType::KEYWORD;**

**keywords["if"] = TokenType::KEYWORD;**

**keywords["else"] = TokenType::KEYWORD;**

**keywords["while"] = TokenType::KEYWORD;**

**keywords["return"] = TokenType::KEYWORD;**

**}**

**// Function to check if a character is whitespace**

**bool isWhitespace(char c)**

**{**

**return c == ' ' || c == '\t' || c == '\n'**

**|| c == '\r';**

**}**

**// Function to check if a character is alphabetic**

**bool isAlpha(char c)**

**{**

**return (c >= 'a' && c <= 'z')**

**|| (c >= 'A' && c <= 'Z');**

**}**

**// Function to check if a character is a digit**

**bool isDigit(char c) { return c >= '0' && c <= '9'; }**

**// Function to check if a character is alphanumeric**

**bool isAlphaNumeric(char c)**

**{**

**return isAlpha(c) || isDigit(c);**

**}**

**// Function to get the next word (identifier or keyword)**

**// from the input**

**string getNextWord()**

**{**

**size\_t start = position;**

**while (position < input.length()**

**&& isAlphaNumeric(input[position])) {**

**position++;**

**}**

**return input.substr(start, position - start);**

**}**

**// Function to get the next number (integer or float)**

**// from the input**

**string getNextNumber()**

**{**

**size\_t start = position;**

**bool hasDecimal = false;**

**while (position < input.length()**

**&& (isDigit(input[position])**

**|| input[position] == '.')) {**

**if (input[position] == '.') {**

**if (hasDecimal)**

**break;**

**hasDecimal = true;**

**}**

**position++;**

**}**

**return input.substr(start, position - start);**

**}**

**public:**

**// Constructor for LexicalAnalyzer**

**LexicalAnalyzer(const string& source)**

**: input(source)**

**, position(0)**

**{**

**initKeywords();**

**}**

**// Function to tokenize the input string**

**vector<Token> tokenize()**

**{**

**vector<Token> tokens;**

**while (position < input.length()) {**

**char currentChar = input[position];**

**// Skip whitespace**

**if (isWhitespace(currentChar)) {**

**position++;**

**continue;**

**}**

**// Identify keywords or identifiers**

**if (isAlpha(currentChar)) {**

**string word = getNextWord();**

**if (keywords.find(word) != keywords.end()) {**

**tokens.emplace\_back(TokenType::KEYWORD,**

**word);**

**}**

**else {**

**tokens.emplace\_back(**

**TokenType::IDENTIFIER, word);**

**}**

**}**

**// Identify integer or float literals**

**else if (isDigit(currentChar)) {**

**string number = getNextNumber();**

**if (number.find('.') != string::npos) {**

**tokens.emplace\_back(**

**TokenType::FLOAT\_LITERAL, number);**

**}**

**else {**

**tokens.emplace\_back(**

**TokenType::INTEGER\_LITERAL, number);**

**}**

**}**

**// Identify operators**

**else if (currentChar == '+'**

**|| currentChar == '-'**

**|| currentChar == '\*'**

**|| currentChar == '/') {**

**tokens.emplace\_back(TokenType::OPERATOR,**

**string(1, currentChar));**

**position++;**

**}**

**// Identify punctuators**

**else if (currentChar == '('**

**|| currentChar == ')'**

**|| currentChar == '{'**

**|| currentChar == '}'**

**|| currentChar == ';') {**

**tokens.emplace\_back(TokenType::PUNCTUATOR,**

**string(1, currentChar));**

**position++;**

**}**

**// Handle unknown characters**

**else {**

**tokens.emplace\_back(TokenType::UNKNOWN,**

**string(1, currentChar));**

**position++;**

**}**

**}**

**return tokens;**

**}**

**};**

**// Function to convert TokenType to string for printing**

**string getTokenTypeName(TokenType type)**

**{**

**switch (type) {**

**case TokenType::KEYWORD:**

**return "KEYWORD";**

**case TokenType::IDENTIFIER:**

**return "IDENTIFIER";**

**case TokenType::INTEGER\_LITERAL:**

**return "INTEGER\_LITERAL";**

**case TokenType::FLOAT\_LITERAL:**

**return "FLOAT\_LITERAL";**

**case TokenType::OPERATOR:**

**return "OPERATOR";**

**case TokenType::PUNCTUATOR:**

**return "PUNCTUATOR";**

**case TokenType::UNKNOWN:**

**return "UNKNOWN";**

**default:**

**return "UNDEFINED";**

**}**

**}**

**// Function to print all tokens**

**void printTokens(const vector<Token>& tokens)**

**{**

**for (const auto& token : tokens) {**

**cout << "Type: " << getTokenTypeName(token.type)**

**<< ", Value: " << token.value << endl;**

**}**

**}**

**// Driver Code**

**int main()**

**{**

**// Sample source code to be analyzed**

**string sourceCode**

**= "int main() { float x = 3.14; float y=3.15; "**

**"float z=x+y; return 0; }";**

**// Create a LexicalAnalyzer object**

**LexicalAnalyzer lexer(sourceCode);**

**// Tokenize the source code**

**vector<Token> tokens = lexer.tokenize();**

**// Print the original source code**

**cout << "Source code: " << sourceCode << endl << endl;**

**// Print all identified tokens**

**cout << "Tokens Generate by Lexical Analyzer:" << endl;**

**printTokens(tokens);**

**return 0;**

**}**

**References**

**1. Aho, A.V., Lam, M.S., Sethi, R., & Ullman, J.D. (2006). Compilers: Principles, Techniques, and Tools.  
2. https://en.wikipedia.org/wiki/Lexical\_analysis  
3. https://www.geeksforgeeks.org/introduction-of-compiler-design/**