MISR UNIVERSITY
FOR SCIENCE & TECHNOLOGY
College of Information Technology

جـامعـة مـصـر
للعلـوم والتكنـولـوجيـا
كليـة تكنولوجيـا المعلومـات

LEXICAL ANALYZER

Prepared By
Student Name:Mohamed Hanafy
Student ID:200043742

Under Supervision
Name of Doctor :Nahal AbdelSalam
Name of T. A.: Fares Imad Al-din

1. Introduction

- **1.1. Phases of Compiler**
  Explain the different phases of a compiler, including lexical analysis, syntax analysis, semantic analysis, optimization, and code generation. You can highlight the role of the lexical analyzer in breaking down source code into tokens, making the process of parsing and compiling easier.

---

2. Lexical Analyzer

- A lexical analyzer, or lexer, is responsible for reading the input source code and dividing it into a series of tokens. These tokens represent the basic building blocks (keywords, operators, identifiers, etc.) of the source code.
- Discuss the types of tokens that your lexical analyzer identifies, including commands (like int, float, etc.), operators (+, -, etc.), and literals (numeric or decimal literals).
- Mention that the lexical analyzer works by iterating through the characters in the source code, classifying them into different categories based on predefined patterns.

---

3. Software Tools

- **3.1. Computer Program**
  Your C++ program implements the lexical analyzer. The program reads the input string, processes each character, and categorizes it into tokens.

- **3.2. Programming Language**
  The lexical analyzer is implemented using C++, a general-purpose programming language known for its efficiency and control over system resources. C++ offers powerful features like object-oriented programming, which allows you to use structures like enum class and struct to model tokens and their types efficiently.

4. Implementation of a Lexical Analyzer

- This section explains the design and implementation of your lexical analyzer.
    - **Data Structures**
        - You use an enum class to define the possible token types (TokenType), such as commands, operators, literals, etc.
        - You define a Token structure to store the type and value of each token.
    - **Functionality**
        - The tokenize function is the core of your lexical analyzer. It takes an input string, processes each character, and classifies it into tokens based on certain conditions (such as whether a character is a letter, digit, or operator).
    - **Handling Different Token Types**
        - Explain how your program distinguishes between keywords (commands), operators, and literals.
    - **Tokenization Process**
        - Describe the process of reading through the input and identifying different token types, starting from recognizing whitespace and moving through various character checks (e.g., isdigit, isalpha, etc.).

---

5. References

- Include all the sources you referenced while working on the project. This might include textbooks, research papers, online tutorials, or documentation that helped in the implementation of the lexical analyzer.

Additional Notes for the Report:

- **Figures and Tables:** You might want to include a flowchart or diagram to illustrate the tokenization process, as well as a table showing examples of different token types (commands, operators, literals) and their classifications.
- **Example Code Output:** You can show a sample input and output of your program to illustrate how the lexical analyzer works.
- **Challenges Faced:** Briefly mention any challenges you faced while implementing the lexer (e.g., handling multiple token types, handling invalid tokens) and how you overcame them.

- **Table : Token Types Identified by the Lexical Analyzer**

Token Value	Token Type
int,float,string	COMMAND
if,else	COMMAND
return	COMMAND
while	COMMAND
for	COMMAND
cin,cout	COMMAND
A,b,c,…	LETTERS
1,2,3,….	NUM_LITERAL
1.2,2.5,…	DECIMAL_LITERAL
+	PLUS_OP
++	INC_OP
-	MINUS_OP
--	DEC_OP
*	TIMES_OP

Token Value	Token Type
/	DIV_OP
^	POWER_OP
%	MOD_OP
=	ASSIGN_OP
<	LE_COMPARISON
>	RE_COMPARISON
(	L_CIRCLE
)	R_CIRCLE
[	L_SQUARE
]	R_SQUARE
{	L_CURLY
}	R_CURLY
;	SEMICOLON
anything else	INVALID

This table represents all the tokens that the lexical analyzer in your C++ code can identify based on the current implementation. It includes:

- **Commands** (reserved keywords like int, float, if, etc.),
- **Letters** (identifiers such as variables or function names),
- **Number Literals** (integer and decimal values),
- **Operators** (arithmetic, assignment, comparison, etc.),
- **Punctuation** (brackets, braces, parentheses, and semicolons),
- **Invalid tokens** (characters that don't match any expected pattern).

- Code

```
#include <iostream>
using namespace std;

enum class TokenType {
   COMMAND, LETTERS, NUM_LITERAL, DECIMAL_LITERAL,
   PLUS_OP, MINUS_OP, TIMES_OP, DIV_OP, POWER_OP, MOD_OP, ASSIGN_OP,
   INC_OP, DEC_OP, LE_COMPARISON, RE_COMPARISON,
   L_SQUARE, R_SQUARE, L_CIRCLE, R_CIRCLE, L_CURLY, R_CURLY, SEMICOLON,
   INVALID
};

struct Token {
   TokenType type;
   string value;
};

bool isCommand(const string& word) {
   return (word == "int" || word == "float" || word == "if" || word == "else" ||
         word == "return" || word == "while" || word == "for" || word == "cin" ||
         word == "cout" || word == "string");
}
const int MAX_TOKENS = 100;
int tokenize(const string& input, Token tokens[], int maxTokens) {
   int count = 0;
   size_t i = 0;
   while (i < input.length() && count < maxTokens) {
      char c = input[i];
      if (isspace(c)) {
         i++;
         continue;
      } else if (isalpha(c)) {
         string word;
         while (i < input.length() && (isalnum(input[i]) || input[i] == '_')) {
            word += input[i];
            i++;
         }
         tokens[count++] = {isCommand(word) ? TokenType::COMMAND :
TokenType::LETTERS, word};
      } else if (isdigit(c)) {
```

```
            string number;
            bool hasDecimal = false;
            while (i < input.length() && (isdigit(input[i]) || (input[i] == '.' && !hasDecimal))) {
                if (input[i] == '.') hasDecimal = true;
                number += input[i];
                i++;
            }
            tokens[count++] = {hasDecimal ? TokenType::DECIMAL_LITERAL :
TokenType::NUM_LITERAL, number};
        } else if (c == '+') {
            if (i + 1 < input.length() && input[i + 1] == '+') {
                tokens[count++] = {TokenType::INC_OP, "++"};
                i += 2;
            } else {
                tokens[count++] = {TokenType::PLUS_OP, "+"};
                i++;
            }
        } else if (c == '-') {
            if (i + 1 < input.length() && input[i + 1] == '-') {
                tokens[count++] = {TokenType::DEC_OP, "--"};
                i += 2;
            } else {
                tokens[count++] = {TokenType::MINUS_OP, "-"};
                i++;
            }
        } else if (c == '*') {
            tokens[count++] = {TokenType::TIMES_OP, "*"};
            i++;
        } else if (c == '/') {
            tokens[count++] = {TokenType::DIV_OP, "/"};
            i++;
        } else if (c == '^') {
            tokens[count++] = {TokenType::POWER_OP, "^"};
            i++;
        } else if (c == '%') {
            tokens[count++] = {TokenType::MOD_OP, "%"};
            i++;
        } else if (c == '=') {
            tokens[count++] = {TokenType::ASSIGN_OP, "="};
            i++;
```

```
        } else if (c == '<') {
            tokens[count++] = {TokenType::LE_COMPARISON, "<"};
            i++;
        } else if (c == '>') {
            tokens[count++] = {TokenType::RE_COMPARISON, ">"};
            i++;
        } else if (c == '(') {
            tokens[count++] = {TokenType::L_CIRCLE, "("};
            i++;
        } else if (c == ')') {
            tokens[count++] = {TokenType::R_CIRCLE, ")"};
            i++;
        } else if (c == '[') {
            tokens[count++] = {TokenType::L_SQUARE, "["};
            i++;
        } else if (c == ']') {
            tokens[count++] = {TokenType::R_SQUARE, "]"};
            i++;
        } else if (c == '{') {
            tokens[count++] = {TokenType::L_CURLY, "{"};
            i++;
        } else if (c == '}') {
            tokens[count++] = {TokenType::R_CURLY, "}"};
            i++;
        } else if (c == ';') {
            tokens[count++] = {TokenType::SEMICOLON, ";"};
            i++;
        } else {
            tokens[count++] = {TokenType::INVALID, string(1, c)};
            i++;
        }
    }
    return count;
}
void printTokens(const Token tokens[], int count) {
    for (int i = 0; i < count; i++) {
        const Token& token = tokens[i];
        cout << "[ " << token.value << " : ";
        switch (token.type) {
            case TokenType::COMMAND: cout << "COMMAND"; break;
```

```
            case TokenType::LETTERS: cout << "LETTERS"; break;
            case TokenType::NUM_LITERAL: cout << "NUM_LITERAL"; break;
            case TokenType::DECIMAL_LITERAL: cout << "DECIMAL_LITERAL"; break;
            case TokenType::PLUS_OP: cout << "PLUS_OP"; break;
            case TokenType::MINUS_OP: cout << "MINUS_OP"; break;
            case TokenType::TIMES_OP: cout << "TIMES_OP"; break;
            case TokenType::DIV_OP: cout << "DIV_OP"; break;
            case TokenType::POWER_OP: cout << "POWER_OP"; break;
            case TokenType::MOD_OP: cout << "MOD_OP"; break;
            case TokenType::ASSIGN_OP: cout << "ASSIGN_OP"; break;
            case TokenType::INC_OP: cout << "INC_OP"; break;
            case TokenType::DEC_OP: cout << "DEC_OP"; break;
            case TokenType::LE_COMPARISON: cout << "LE_COMPARISON"; break;
            case TokenType::RE_COMPARISON: cout << "RE_COMPARISON"; break;
            case TokenType::L_SQUARE: cout << "L_SQUARE"; break;
            case TokenType::R_SQUARE: cout << "R_SQUARE"; break;
            case TokenType::L_CIRCLE: cout << "L_CIRCLE"; break;
            case TokenType::R_CIRCLE: cout << "R_CIRCLE"; break;
            case TokenType::L_CURLY: cout << "L_CURLY"; break;
            case TokenType::R_CURLY: cout << "R_CURLY"; break;
            case TokenType::SEMICOLON: cout << "SEMICOLON"; break;
            case TokenType::INVALID: cout << "INVALID"; break;
        }
        cout << " ]\n";
    }
}
int main() {
    string input;
    while (true) {
        cout << "Enter source code (type 'exit' to quit): ";
        getline(cin, input);
        if (input == "exit") break;
        Token tokens[MAX_TOKENS];
        int count = tokenize(input, tokens, MAX_TOKENS);
        cout << "\nTokens:\n";
        printTokens(tokens, count);
    }
    return 0;
}
```

- Image for output code

```
Enter source code (type 'exit' to quit): +

Tokens:
[ + : PLUS_OP ]
Enter source code (type 'exit' to quit): -

Tokens:
[ - : MINUS_OP ]
Enter source code (type 'exit' to quit): (

Tokens:
[ ( : L_CIRCLE ]
Enter source code (type 'exit' to quit): {

Tokens:
[ { : L_CURLY ]
Enter source code (type 'exit' to quit): <

Tokens:
[ < : LE_COMPARISON ]
Enter source code (type 'exit' to quit): 1

Tokens:
[ 1 : NUM_LITERAL ]
Enter source code (type 'exit' to quit): 3.5

Tokens:
[ 3.5 : DECIMAL_LITERAL ]
Enter source code (type 'exit' to quit): exit
```