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**

Karim fetyan ab del rahman

200039958

**Under Supervision**

Dr:Nehal Abd El Salam

T.A:Mennatallah sherif

1. **Introduction**

**A lexical analyzer (also known as a lexer or scanner) is the first phase in a compiler or interpreter that processes source code. Its primary function is to convert a sequence of characters from the source program into a sequence of tokens that can be more easily processed by subsequent compiler phases.**

**What is a Lexical Analyzer?**

**A lexical analyzer reads the input character stream from source code and groups characters into meaningful sequences called tokens. Tokens typically include:**

* **Keywords (like "if", "while", "int")**
* **Identifiers (variable and function names)**
* **Operators (+, -, \*, /, etc.)**
* **Literals (numbers, strings)**
* **Punctuation symbols (parentheses, commas, semicolons)**

**For example, the statement int count = 10; would be broken down into tokens:**

* **"int" (keyword)**
* **"count" (identifier)**
* **"=" (assignment operator)**
* **"10" (integer literal)**
* **";" (semicolon)**

**Importance of Lexical Analysis**

**Lexical analysis is crucial for several reasons:**

1. **Simplification: It abstracts away details like whitespace and comments, providing a cleaner input for the parser.**
2. **Error Detection: It identifies lexical errors such as invalid characters or malformed tokens.**
3. **Efficiency: It can handle simple tasks like stripping comments and handling whitespace, allowing the parser to focus on grammar rules.**
4. **Symbol Table Management: Often begins building the symbol table by recording identifiers.**
5. **Portability: Separating lexical analysis makes it easier to adapt compilers to different character sets and encoding schemes.**

**How to Obtain/Implement a Lexical Analyzer**

**There are several approaches to obtaining or implementing a lexical analyzer:**

**1. Using Lexical Analyzer Generators**

**Tools called lexical analyzer generators can automatically create a lexer from a specification:**

* **Lex/Flex: The most common lexical analyzer generator for C/C++**
* **JFlex: For Java**
* **PLY: For Python**
* **ANTLR: Can generate lexers for multiple languages**

**2. Manual Implementation**

**You can write a lexical analyzer from scratch using techniques like:**

* **Table-driven approaches**
* **Direct coding using finite state machines**
* **Regular expression matching**

**3. Library-based Approaches**

**Many programming language libraries provide built-in lexical analysis capabilities:**

* **Python: re module for regular expressions**
* **Java: java.util.regex package**
* **C++: std::regex library**

1. **Software Tools**
   1. **Computer Program**
   2. **Vs code**
   3. **Programming Language**
   4. **C++**
2. **Implementation of a Lexical Analyzer**
3. **References**

**1-** [**https://www.tutorialspoint.com/compiler\_design/compiler\_design\_lexical\_analysis.htm**](https://www.tutorialspoint.com/compiler_design/compiler_design_lexical_analysis.htm)

**2-** [**https://medium.com/@ombelorkar2103/various-lexical-analyzer-tools-4f8fa0d2d675**](https://medium.com/@ombelorkar2103/various-lexical-analyzer-tools-4f8fa0d2d675)

**3-https://stackoverflow.com/questions/17848207/making-a-lexical-analyzer**

**4-https://favtutor.com/blogs/lexical-analyzer-cpp**

**5- Concepts of Programming Language textbook**

**A screen shot of a computer program

AI-generated content may be incorrect.**

**A screen shot of a computer program

AI-generated content may be incorrect.**

**A screen shot of a computer program

AI-generated content may be incorrect.**

**A screen shot of a computer program

AI-generated content may be incorrect.**

**A screenshot of a computer program

AI-generated content may be incorrect.**

**A screen shot of a computer program

AI-generated content may be incorrect.**

**A screen shot of a computer program

AI-generated content may be incorrect.**

**A screen shot of a computer program

AI-generated content may be incorrect.**

**Global Declarations**

* **Character Classes and Token Codes:**
  + **enum { LETTER = 0, DIGIT = 1, UNKNOWN = 99, EOF\_CLASS = -1 };: Defines character classes to categorize input characters.**
  + **enum { 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 };: Defines token codes representing different types of tokens the lexer can identify.**
* **Global Variables:**
  + **int charClass;: Stores the character class of the current character.**
  + **char lexeme[100];: Array to store the current lexeme (a sequence of characters forming a token).**
  + **char nextChar;: Holds the next character from the input.**
  + **int lexLen;: Length of the current lexeme.**
  + **int token;: Represents the current token.**
  + **int nextToken;: Represents the next token to be processed.**
  + **ifstream in\_fp;: Input file stream to read from the source file.**

**Function Definitions**

1. **void addChar()**
   * **Purpose: Adds nextChar to the lexeme array.**
   * **Details: Checks if lexLen is within bounds to prevent buffer overflow. If within bounds, appends nextChar to lexeme and null-terminates it. If the lexeme is too long, outputs an error message.**
2. **void getChar()**
   * **Purpose: Reads the next character from the input file and determines its character class.**
   * **Details: Uses in\_fp.get(nextChar) to read the next character. Sets charClass based on whether nextChar is a letter, digit, or an unknown character. If the end of the file is reached, sets charClass to EOF\_CLASS.**
3. **void getNonBlank()**
   * **Purpose: Skips over whitespace characters to find the next non-whitespace character.**
   * **Details: Continuously calls getChar() until nextChar is not a whitespace character.**
4. **int lookup(char ch)**
   * **Purpose: Determines the token code for single-character operators and parentheses.**
   * **Details: Uses a switch statement to match ch with known operators and parentheses. Calls addChar() to add the character to lexeme and sets nextToken to the corresponding token code. If ch doesn't match any known characters, sets nextToken to EOF.**
5. **int lex()**
   * **Purpose: The core lexical analyzer function that processes the input and identifies tokens.**
   * **Details:**
     + **Initializes lexLen to 0 and calls getNonBlank() to skip whitespace.**
     + **Uses a switch statement based on charClass:**
       - **LETTER: Recognizes identifiers by adding the initial letter to lexeme, then continues to add subsequent letters or digits. Sets nextToken to IDENT.**
       - **DIGIT: Recognizes integer literals by adding the initial digit to lexeme, then continues to add subsequent digits. Sets nextToken to INT\_LIT.**
       - **UNKNOWN: Calls lookup(nextChar) to identify operators or parentheses.**
       - **EOF\_CLASS: Sets nextToken to EOF and assigns the string "EOF" to lexeme.**
     + **Outputs the identified token and its corresponding lexeme.**
     + **Returns nextToken.**

**main() Function**

* **Purpose: Serves as the entry point of the program, orchestrating the lexical analysis process.**
* **Details:**
  + **Opens the input file front.in using in\_fp.open("front.in"). If the file cannot be opened, outputs an error message and returns with an error code.**
  + **Calls getChar() to initialize the first character read.**
  + **Enters a do-while loop that repeatedly calls lex() to process and identify tokens until nextToken equals EOF.**
  + **Closes the input file stream before terminating the program.**

**Program Flow**

1. **Initialization: The program attempts to open front.in. If unsuccessful, it outputs an error and terminates.**
2. **Character Reading: Reads characters one by one, determining their class (LETTER, DIGIT, UNKNOWN, or EOF\_CLASS).**
3. **Token Identification: Based on character classes and specific characters, the program constructs lexemes and assigns appropriate token codes.**
4. **Output: For each token identified, the program outputs its token code and lexeme.**
5. **Termination: The process continues until the end of the file is reached, at which point the program closes the file and terminates.**

**Key Points**

* **Lexeme Construction: The addChar() function ensures that characters are added to lexeme safely, preventing buffer overflow.**
* **Character Classification: The getChar() function reads characters and classifies them, facilitating the identification of tokens.**
* **Whitespace Handling: The getNonBlank() function ensures that whitespace does not interfere with token recognition.**
* **Operator and Parenthesis Recognition: The lookup() function matches specific characters to their corresponding token codes.**
* **Token Processing Loop: The main() function's loop continues processing tokens until the end of the file, ensuring all tokens are identified and output.**

**Github link:**

**https://github.com/KarimAbdElRahman13/lexical-analyzer**