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**

Ahmed Ehab Amin

200038678

**Under Supervision**

Doctor

Nehal Abdelsalam

Name of T. A.

Mennatallah shrief

1. **Introduction**

A compiler is a fundamental tool in computer science that translates high-level programming languages into machine code. One of the critical stages in this process is **Lexical Analysis**, where the input source code is converted into a sequence of meaningful tokens. This stage acts as the foundation for further phases such as syntax and semantic analysis. The **Lexical Analyzer**, also known as a scanner, plays a vital role in detecting the basic syntactic units of a program such as keywords, identifiers, operators, and literals.

In this report, we will explore the core concepts of lexical analysis, its implementation, and the tools used in building a lexical analyzer. A practical implementation will also be discussed to illustrate how a lexical analyzer operates within a compiler pipeline.

1. **Lexical Analyzer**

**A diagram of a computer program

AI-generated content may be incorrect.**

**A compiler is responsible for translating source code written in a high-level programming language into a target language, usually machine code. This process is divided into several phases, as shown in the diagram. These phases can be categorized into two main parts: Front End and Back End.**

**🔹 Front End:**

1. **Source Language  
   This is the original code written by the programmer, usually in a high-level language like C, Java, or Python.**
2. **Lexical Analyzer  
   Also known as the scanner, it reads the source code and breaks it into tokens — the smallest units like keywords, identifiers, symbols, etc.**
3. **Syntax Analyzer  
   This phase checks the syntax or grammar of the token sequence. It builds a parse tree to represent the syntactic structure of the program.**
4. **Semantic Analyzer  
   Ensures that the code makes sense semantically — for example, it checks if variables are declared before use or if operations are applied to the correct data types.**
5. **Intermediate Code Generator  
   Converts the parse tree into an intermediate representation that is easier to analyze and optimize. This is not machine-specific code.**

**🔹 Intermediate Code:**

* **Intermediate Code  
  A simplified version of the code that lies between the high-level source code and the final machine code. It helps in optimization and portability.**

**🔹 Back End:**

1. **Code Optimizer  
   Improves the intermediate code by making it run faster or take up less space, without changing its behavior.**
2. **Target Code Generator  
   Converts the optimized intermediate code into the target language, which is typically machine code or assembly code.**
3. **Target Language  
   This is the final output of the compiler — the code that the machine can execute directly.**
4. **Software Tools**

I used python and visual studio

1. **Implementation of a Lexical Analyzer**

**import sys # Importing the sys module, typically used for system-specific parameters and functions (not used explicitly in this code).**

**# Global Declarations**

**char\_class = None # Variable to store the current character class (e.g., letter, digit, unknown)**

**lexeme = "" # Variable to store the lexeme being built**

**next\_char = "" # Variable to store the next character to be processed**

**lex\_len = 0 # Variable to store the length of the current lexeme**

**token = None # Placeholder for token (not used in this code)**

**next\_token = None # Variable to store the next token to be processed**

**in\_fp = None # File pointer to read the input file**

**# Character Classes**

**LETTER = 0 # Constant representing letter characters**

**DIGIT = 1 # Constant representing digit characters**

**UNKNOWN = 99 # Constant representing unknown characters**

**# Token Codes**

**INT\_LIT = 10 # Token for integer literal**

**IDENT = 11 # Token for identifiers**

**ASSIGN\_OP = 20 # Token for assignment operator (=)**

**ADD\_OP = 21 # Token for addition operator (+)**

**SUB\_OP = 22 # Token for subtraction operator (-)**

**MULT\_OP = 23 # Token for multiplication operator (\*)**

**DIV\_OP = 24 # Token for division operator (/)**

**LEFT\_PAREN = 25 # Token for left parenthesis '('**

**RIGHT\_PAREN = 26 # Token for right parenthesis ')'**

**EOF\_TOKEN = -1 # Token for end of file**

**# Function declarations**

**def add\_char():**

**global lexeme, lex\_len # Declare global variables that will be modified**

**if lex\_len <= 98: # If the length of the lexeme is less than or equal to 98**

**lexeme += next\_char # Add the current character to the lexeme**

**lex\_len += 1 # Increment the length of the lexeme by 1**

**else:**

**print("Error - lexeme is too long") # If the lexeme is too long, print an error**

**def get\_char():**

**global next\_char, char\_class # Declare global variables that will be modified**

**next\_char = in\_fp.read(1) # Read one character from the file**

**if next\_char: # If a character was read**

**if next\_char.isalpha(): # If the character is a letter**

**char\_class = LETTER # Set the character class to letter**

**elif next\_char.isdigit(): # If the character is a digit**

**char\_class = DIGIT # Set the character class to digit**

**else: # If the character is unknown**

**char\_class = UNKNOWN # Set the character class to unknown**

**else: # If no character is found (end of file)**

**char\_class = EOF\_TOKEN # Set the character class to end of file**

**def get\_non\_blank():**

**while next\_char.isspace(): # While the current character is a whitespace**

**get\_char() # Read the next character**

**def lex():**

**global lex\_len, next\_token # Declare global variables that will be modified**

**lex\_len = 0 # Reset the lexeme length to 0**

**get\_non\_blank() # Skip over any whitespace**

**if char\_class == LETTER: # If the character is a letter**

**add\_char() # Add the character to the lexeme**

**get\_char() # Read the next character**

**while char\_class == LETTER or char\_class == DIGIT: # While the character is a letter or digit**

**add\_char() # Add the character to the lexeme**

**get\_char() # Read the next character**

**next\_token = IDENT # Set the next token to identifier**

**elif char\_class == DIGIT: # If the character is a digit**

**add\_char() # Add the character to the lexeme**

**get\_char() # Read the next character**

**while char\_class == DIGIT: # While the character is a digit**

**add\_char() # Add the character to the lexeme**

**get\_char() # Read the next character**

**next\_token = INT\_LIT # Set the next token to integer literal**

**elif char\_class == UNKNOWN: # If the character is unknown**

**lookup(next\_char) # Call the lookup function to identify the character**

**get\_char() # Read the next character**

**elif char\_class == EOF\_TOKEN: # If the character is the end of file**

**next\_token = EOF\_TOKEN # Set the next token to end of file**

**def lookup(ch):**

**global next\_token # Declare global variable that will be modified**

**if ch == '(': # If the character is a left parenthesis**

**add\_char() # Add the character to the lexeme**

**next\_token = LEFT\_PAREN # Set the next token to left parenthesis**

**elif ch == ')': # If the character is a right parenthesis**

**add\_char() # Add the character to the lexeme**

**next\_token = RIGHT\_PAREN # Set the next token to right parenthesis**

**elif ch == '+': # If the character is a plus sign**

**add\_char() # Add the character to the lexeme**

**next\_token = ADD\_OP # Set the next token to addition operator**

**elif ch == '-': # If the character is a minus sign**

**add\_char() # Add the character to the lexeme**

**next\_token = SUB\_OP # Set the next token to subtraction operator**

**elif ch == '\*': # If the character is an asterisk (multiplication)**

**add\_char() # Add the character to the lexeme**

**next\_token = MULT\_OP # Set the next token to multiplication operator**

**elif ch == '/': # If the character is a forward slash (division)**

**add\_char() # Add the character to the lexeme**

**next\_token = DIV\_OP # Set the next token to division operator**

**elif ch == '=': # If the character is an equal sign (assignment)**

**add\_char() # Add the character to the lexeme**

**next\_token = ASSIGN\_OP # Set the next token to assignment operator**

**else: # If the character is not recognized**

**if ch != '': # If the character is not empty**

**add\_char() # Add the character to the lexeme**

**next\_token = EOF\_TOKEN # Set the next token to end of file**

**return next\_token # Return the identified token**

**# Main driver**

**if \_\_name\_\_ == "\_\_main\_\_": # If the script is being run directly**

**try:**

**with open("front.in", "r") as in\_fp: # Open the input file "front.in" for reading safely using 'with' statement**

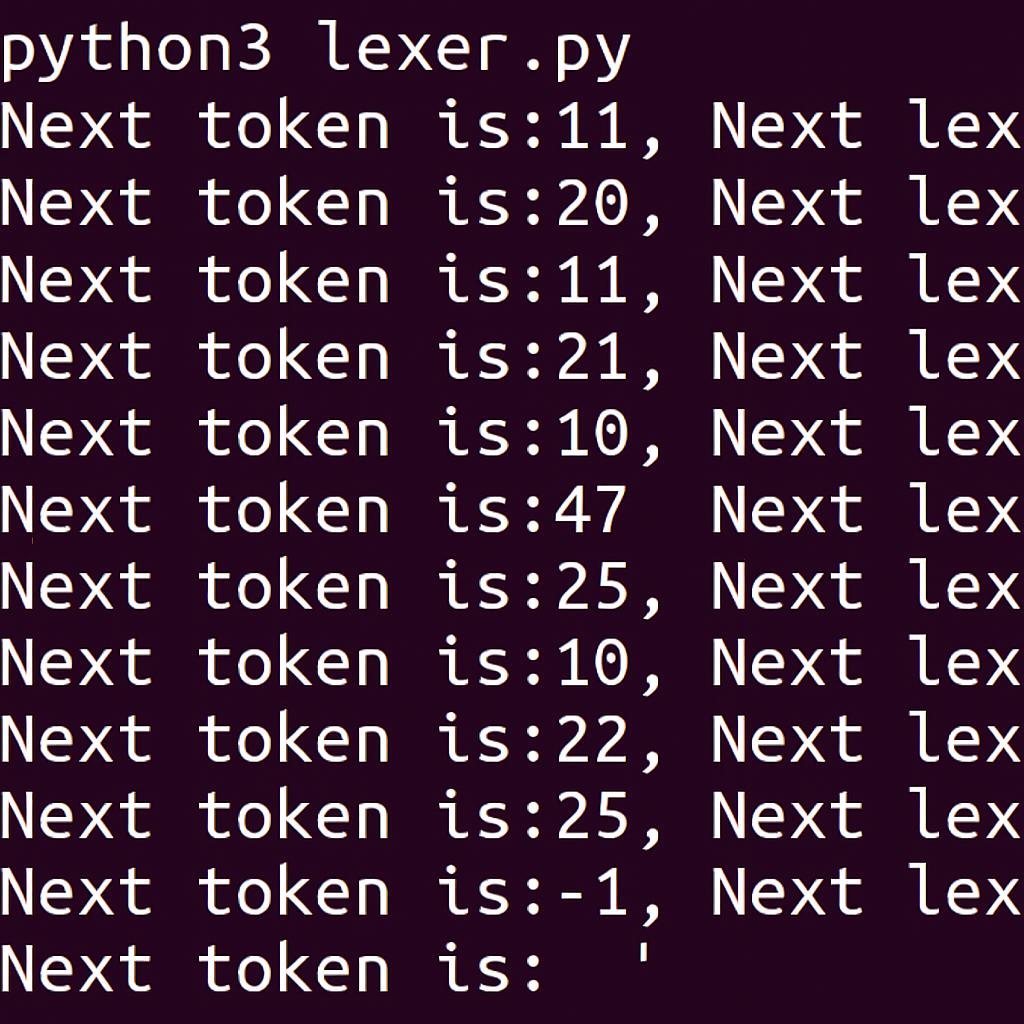
**get\_char() # Read the first character from the file**

**while next\_token != EOF\_TOKEN: # While the next token is not end of file**

**lex() # Call the lex function to process the text**

**except FileNotFoundError: # If the file is not found**

**print("ERROR - cannot open front.in") # Print an error message**

****

**A diagram with text and words

AI-generated content may be incorrect.**

**The diagram helps visualize how the analyzer reads characters and identifies different types of tokens such as identifiers (names), integers, parentheses, and arithmetic operators.**

**Main Points:**

1. **Purpose of the Diagram:**
   * **The diagram is used to show the flow of reading characters and recognizing patterns for tokens.**
   * **It also shows what actions are taken during each transition between states (e.g., adding a character, getting the next character).**
2. **States in the Diagram:**
   * **Start: The initial state.**
   * **From the start, depending on the character type:**
     + **If it’s a letter, the analyzer moves to the id state (for identifiers).**
     + **If it’s a digit, it goes to the int state (for integer literals).**
     + **If it’s something else (like an operator or parenthesis), it goes to the unknown state.**
3. **Transitions and Actions:**
   * **Actions like addChar and getChar are performed during transitions.**
   * **When a full token is formed:**
     + **id state calls lookup(lexeme) to check if it’s a keyword or identifier.**
     + **int state returns the token type Int\_Lit.**
     + **unknown calls lookup(nextChar) to classify the symbol (like +, -, (, )), then returns the token t.**
4. **Final Step:**
   * **After processing the unknown character, it reaches the Done state and returns the final token.**

**This kind of state diagram is essential in building a lexical analyzer for a compiler, helping to break the input code into meaningful tokens.**

1. **References**
   1.  Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools* (2nd ed.). Pearson Education.
   2.  Fischer, C. N., & LeBlanc, R. J. (2009). *Crafting a Compiler* (2nd ed.). Pearson.
   3.  Louden, K. C. (2006). *Compiler Construction: Principles and Practice*. Cengage Learning.
   4.  Torben Ægidius Mogensen. (2011). *Introduction to Compiler Design*. Springer.
   5.  TutorialsPoint. (n.d.). *Compiler Design - Lexical Analysis*. Retrieved from