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**

**Fares Ahmed**

**200042841**

**Under Supervision**

Dr / Nehal A. Mohamed

Eng / Menntallah Sherif

**1.Introduction**

This document provides an overview of the implementation of a Lexical Analyzer, which is a

fundamental phase in compiler design. It covers the phases of a compiler, the role of a lexical

analyzer, software tools used, and the implementation details.

**1.1 Phases of Compiler**

A compiler consists of several phases, including:

1. **Lexical Analysis**: Tokenizing the input code.

2. **Syntax Analysis**: Checking grammatical structure.

3. **Semantic Analysis**: Ensuring meaningful statements.

4. **Intermediate Code Generation**: Creating an intermediate representation.

5. **Optimization**: Improving performance and efficiency.

6. **Code Generation**: Producing machine code.

**2. Lexical Analyzer**

A **Lexical Analyzer** is responsible for scanning the source code and converting it into tokens.

It identifies keywords, operators, identifiers, and other elements.

**3. Software Tools**

Various software tools are used in compiler construction.

**3.1. Computer Program**

A compiler is a special type of program that translates source code into machine code. It

ensures the correctness of syntax and semantics.

**3.2. Programming Language**

Lexical analyzers are often implemented using programming languages like Python, C,

or Java. The implementation in this document is in Python

**4. Implementation of a Lexical Analyzer**

Below is the Python implementation of a lexical analyzer:

import string

charClass = None

lexeme = []

nextChar = ''

lexLen = 0

token = None

nextToken = None

input\_string = "(sum + 457) / total"

input\_index = 0

LETTER = 0

DIGIT = 1

UNKNOWN = 99

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

EOF = -1

def addChar():

global lexLen

if lexLen <= 98:

lexeme.append(nextChar)

lexLen += 1

else:

print("Error - lexeme is too long")

def getChar():

global nextChar, charClass, input\_index

if input\_index < len(input\_string):

nextChar = input\_string[input\_index]

input\_index += 1

else:

nextChar = ''

if nextChar == '':

charClass = EOF

elif nextChar.isalpha():

charClass = LETTER

elif nextChar.isdigit():

charClass = DIGIT

else:

charClass = UNKNOWN

def getNonBlank():

while nextChar.isspace():

getChar()

def lookup(ch):

global nextToken

if ch == '(':

addChar()

nextToken = LEFT\_PAREN

elif ch == ')':

addChar()

nextToken = RIGHT\_PAREN

elif ch == '+':

addChar()

nextToken = ADD\_OP

elif ch == '-':

addChar()

nextToken = SUB\_OP

elif ch == '\*':

addChar()

nextToken = MULT\_OP

elif ch == '/':

addChar()

nextToken = DIV\_OP

else:

addChar()

nextToken = EOF

return nextToken

def lex():

global lexLen, nextToken

lexLen = 0

getNonBlank()

if charClass == LETTER:

addChar()

getChar()

while charClass == LETTER or charClass == DIGIT:

addChar()

getChar()

nextToken = IDENT

elif charClass == DIGIT:

addChar()

getChar()

while charClass == DIGIT:

addChar()

getChar()

nextToken = INT\_LIT

elif charClass == UNKNOWN:

lookup(nextChar)

getChar()

elif charClass == EOF:

nextToken = EOF

lexeme.append('E')

lexeme.append('O')

lexeme.append('F')

print(f"Next token is: {nextToken}, Next lexeme is {''.join(lexeme)}")

return nextToken

def main():

getChar()

while nextToken != EOF:

lex()

if \_\_name\_\_ == "\_\_main\_\_":

main()

explanation of each line of the code in English:

### 1. Importing the string module:

· This imports Python's built-in string module, which provides various string constants and functions. Although not directly used in this code, it's often useful for string operations.

### 2. Declaring global variables:

· charClass: A variable used to store the class of the current character being processed (e.g., letter, digit, or unknown).

· lexeme: A list to hold the characters of the current token (the "lexeme").

· nextChar: The next character to be analyzed from the input string.

· lexLen: The length of the current lexeme being processed.

· token: Holds the current token type

· nextToken: The next token to be processed.

·input\_string: The input expression that will be lexically analyzed (in this case, the string "(sum + 457) / total").

· input\_index: The index used to track the current position in the input string.

### Defining constants for character classes and token codes:

These are constant values representing different character classes (LETTER, DIGIT, etc.) and token types (INT\_LIT for integer literals, ADD\_OP for the + operator, etc.).

### addChar function:

This function appends the current character (nextChar) to the lexeme list and increments the lexLen (length of the lexeme).

If the lexeme exceeds 98 characters, an error message is printed.

5. getChar() function:

This function reads the next character from input\_string based on input\_index and updates nextChar.

If nextChar is an empty string (indicating the end of the string), charClass is set to EOF (end of file).

If the character is alphabetic (isalpha()), charClass is set to LETTER.

If the character is numeric (isdigit()), charClass is set to DIGIT.

Otherwise, charClass is set to UNKNOWN.

### 6. getNonBlank() function:

This function skips any whitespace characters (spaces, tabs, newlines) by repeatedly calling getChar() until a non-whitespace character is encountered.

7. lookup() function:

This function checks the current character (ch) and determines which operator or symbol it corresponds to.

If the character is a parenthesis or an operator (+, -, \*, /), it adds the character to the lexeme and sets the appropriate nextToken.

If the character is not recognized, it sets nextToken to EOF.

8. lex() function:

This function is the main lexical analyzer. It processes the input string character by character, building the lexeme and determining the nextToken.

If charClass is LETTER, it collects all letters and digits to form an identifier (IDENT).

If charClass is DIGIT, it collects all digits to form an integer literal (INT\_LIT)

If charClass is UNKNOWN, it looks up the character to determine the token (operator, parenthesis, etc.).

If charClass is EOF, it sets the token to EOF and appends 'EOF' to lexeme.

The function prints the current token and lexeme.

9. main() function:

This function serves as the entry point of the program.

It calls getChar() to read the first character from the input string.

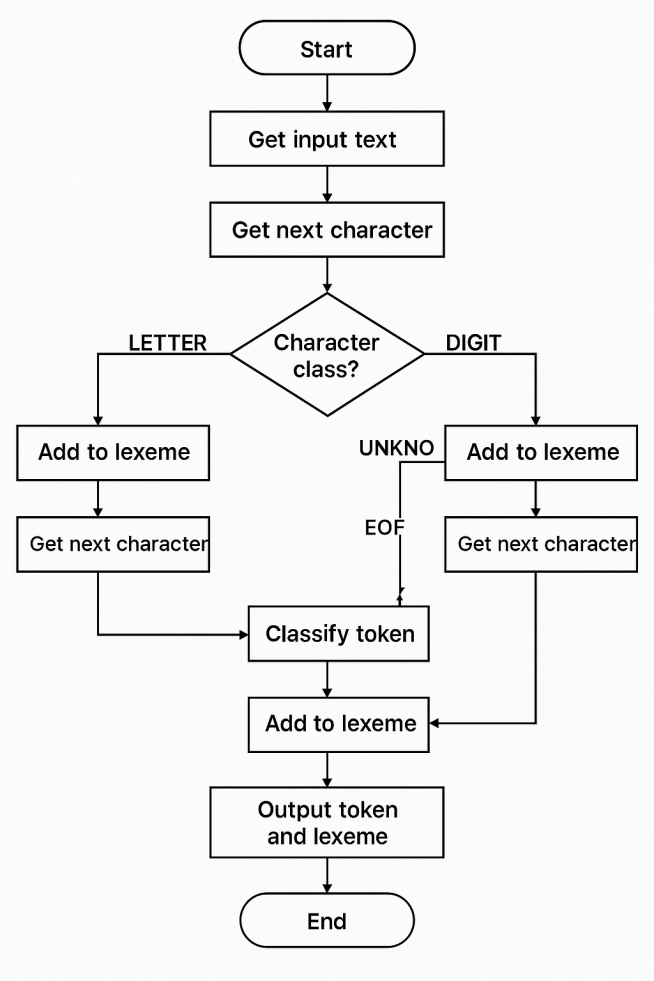
It then enters a loop that repeatedly calls lex() to process the input string until the end of the file (EOF) is reached.

10. Running the program:

This conditional block checks if the script is being run directly (not imported as a module).

If the script is being run directly, it calls the main() function to start the lexical analysis process.

Daigram



**5. References:**

1.Parr, T. (2022). *Language Implementation Patterns: Create Your Own Domain-Specific and*

*General Programming Languages with Python*.

2.Parsons, D. (2021). *Introduction to Compiler Design*.