A blue and green logo

Description automatically generatedA black background with grey leaves

AI-generated content may be incorrect.

lEXICAL ANALYZER

A grey logo on a black background

AI-generated content may be incorrect.

A green and blue line drawing of a building

Description automatically generated

**Prepared By**

Mark John Rasem  
200036305

**Under Supervision**

Dr / Nehal A. Mohamed

Eng /Eslam Saeed

**1.1 Phases of Compiler**

1. **Lexical Analyzer**: Breaks source code into tokens.
2. **Syntax Analyzer**: Checks grammar using tokens and builds a parse tree.
3. **Semantic Analyzer**: Validates meaning (e.g., types, scope).
4. **Intermediate Code Generator**: Produces platform-independent code.
5. **Machine-Independent Code Optimizer**: Improves efficiency without hardware specifics.
6. **Code Generator**: Converts intermediate code to machine-dependent code.
7. **Machine-Dependent Code Optimizer**: Further optimizes for the target hardware.
8. **Target Code**: Final executable output.

**Supporting Components**:

* **Symbol Table**: Tracks identifiers and their attributes.
* **Error Handler**: Detects and reports errors at each phase.

**Key Divisions**:

* **Machine-Independent**: Phases 1–5 (work universally).
* **Machine-Dependent**: Phases 6–7 (hardware-specific).

The process transforms source code into optimized machine code systematically.

**2. Lexical Analyzer**

1. A diagram of a computer program

   AI-generated content may be incorrect.**Start: Reads first character (Letter → id, Digit → Int\_Lit).**
2. **Build Token:**
   * **For id: Keeps adding letters/digits, checks symbol table (lookup).**
   * **For Int\_Lit: Keeps adding digits.**
3. **Handle Unknowns: Skips invalid chars, resumes after.**
4. **Return Token: Outputs id, Int\_Lit, or error.**

**Key Actions: addChar (append), getChar (next char), lookup (validate).**

**3. Software Tools**

Various software tools are used in compiler construction.

**3.1. Computer Program**

Visual studio code

**3.2. Programming Language**

Python

Lexical Analyzer

# Python Code

import string  
  
char\_class = None  
lexeme = []  
next\_char = ''  
lex\_len = 0  
token = None  
next\_token = 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 add\_char():  
 global lex\_len  
 if lex\_len <= 98:  
 lexeme.append(next\_char)  
 lex\_len += 1  
 else:  
 print("Error - lexeme is too long")  
  
def get\_char():  
 global next\_char, char\_class, input\_index  
 if input\_index < len(input\_string):  
 next\_char = input\_string[input\_index]  
 input\_index += 1  
 else:  
 next\_char = ''  
 if next\_char == '':  
 char\_class = EOF  
 elif next\_char.isalpha():  
 char\_class = LETTER  
 elif next\_char.isdigit():  
 char\_class = DIGIT  
 else:  
 char\_class = UNKNOWN  
  
def get\_non\_blank():  
 while next\_char.isspace():  
 get\_char()  
  
def lookup(ch):  
 global next\_token  
 if ch == '(':  
 add\_char()  
 next\_token = LEFT\_PAREN  
 elif ch == ')':  
 add\_char()  
 next\_token = RIGHT\_PAREN  
 elif ch == '+':  
 add\_char()  
 next\_token = ADD\_OP  
 elif ch == '-':  
 add\_char()  
 next\_token = SUB\_OP  
 elif ch == '\*':  
 add\_char()  
 next\_token = MULT\_OP  
 elif ch == '/':  
 add\_char()  
 next\_token = DIV\_OP  
 else:  
 add\_char()  
 next\_token = EOF  
 return next\_token  
  
def lex():  
 global lex\_len, next\_token  
 lex\_len = 0  
 get\_non\_blank()  
 if char\_class == LETTER:  
 add\_char()  
 get\_char()  
 while char\_class == LETTER or char\_class == DIGIT:  
 add\_char()  
 get\_char()  
 next\_token = IDENT  
 elif char\_class == DIGIT:  
 add\_char()  
 get\_char()  
 while char\_class == DIGIT:  
 add\_char()  
 get\_char()  
 next\_token = INT\_LIT  
 elif char\_class == UNKNOWN:  
 lookup(next\_char)  
 get\_char()  
 elif char\_class == EOF:  
 next\_token = EOF  
 lexeme.extend(['E', 'O', 'F'])  
 print(f"Next token is: {next\_token}, Next lexeme is {''.join(lexeme)}")  
 return next\_token  
  
def main():  
 get\_char()  
 while next\_token != EOF:  
 lex()  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 main()

# Explanation of Code

Lexical Analyzer Code Explanation

# Token Classes

LETTER = 0 # Represents alphabetic characters (A-Z, a-z)

DIGIT = 1 # Represents numeric characters (0–9)

UNKNOWN = 99 # Represents unknown characters (e.g., +, -, /, etc.)

EOF = -1 # End of input

# Token Types

INT\_LITERAL = 10 # Integer literal (like 457)

IDENTIFIER = 11 # Identifiers (like variable names: result, total)

ASSIGN\_OP = 20 # Assignment operator (=), not used here but can be added

ADD\_OP = 21 # Addition operator (+)

SUB\_OP = 22 # Subtraction operator (-)

MULT\_OP = 23 # Multiplication operator (\*)

DIV\_OP = 24 # Division operator (/)

LEFT\_PAREN = 25 # Left parenthesis '('

RIGHT\_PAREN = 26 # Right parenthesis ')'

# Global State

char\_class = None # The type of the current character (letter, digit, etc.)

current\_lexeme = [] # Stores the current sequence of characters (like a word or number)

current\_char = '' # The current character being analyzed

current\_token = None # The token type identified from the input

input\_string = "(result + 457) / total" # The string to be tokenized

input\_index = 0 # Index to keep track of position in the input string

def add\_char():

# Adds the current character to the lexeme list

current\_lexeme.append(current\_char)

def get\_char():

# Gets the next character from input and determines its class

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

current\_char = input\_string[input\_index]

input\_index += 1

if current\_char.isalpha():

char\_class = LETTER

elif current\_char.isdigit():

char\_class = DIGIT

else:

char\_class = UNKNOWN

else:

current\_char = ''

char\_class = EOF

def skip\_whitespace():

# Skips any space characters between tokens

while current\_char.isspace():

get\_char()

def lookup\_operator(char):

# Determines the token type for operators or parentheses

if char == '(':

add\_char()

current\_token = LEFT\_PAREN

elif char == ')':

add\_char()

current\_token = RIGHT\_PAREN

elif char == '+':

add\_char()

current\_token = ADD\_OP

elif char == '-':

add\_char()

current\_token = SUB\_OP

elif char == '\*':

add\_char()

current\_token = MULT\_OP

elif char == '/':

add\_char()

current\_token = DIV\_OP

else:

add\_char()

current\_token = EOF

return current\_token

def lexer():

# Main lexical analyzer function

current\_lexeme = []

skip\_whitespace()

if char\_class == LETTER:

add\_char()

get\_char()

while char\_class in (LETTER, DIGIT):

add\_char()

get\_char()

current\_token = IDENTIFIER

elif char\_class == DIGIT:

add\_char()

get\_char()

while char\_class == DIGIT:

add\_char()

get\_char()

current\_token = INT\_LITERAL

elif char\_class == UNKNOWN:

lookup\_operator(current\_char)

get\_char()

elif char\_class == EOF:

current\_token = EOF

current\_lexeme = ['E', 'O', 'F']

print(f"Next token is: {current\_token}, Next lexeme is {''.join(current\_lexeme)}")

return current\_token

def main():

get\_char()

while True:

current\_token = lexer()

if current\_token == EOF:

break

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

main()

# 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.