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**

Student Name:Yousef Maher

Student ID:200043815

**Under Supervision**

Dr:Nehal A.Mohamed

Name of T. A.

**-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**

**-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.**

**-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**

**-Software tools**

**Various software tools are used in compiler construction.**

**-Computer program**

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

**Ensures the correctness of syntax and semanti**

**-Programming Language**

**Lexical analyzers are often implemented using programming languages like Python, C, or Java. The implementation in this document is in Python.**

**Implementation of lexical Analyzer**

**# front.py – a lexical analyzer system for simple arithmetic expressions**

**Import string**

**# Global declarations**

**# Variables**

**charClass = None**

**lexeme = [‘’] \* 100**

**nextChar = ‘’**

**lexLen = 0**

**token = None**

**nextToken = None**

**in\_fp = None**

**# Function declarations**

**Def addChar():**

**Global lexLen, lexeme, nextChar**

**If lexLen <= 98:**

**Lexeme[lexLen] = nextChar**

**lexLen += 1**

**lexeme[lexLen] = ‘\0’**

**else:**

**print(“Error – lexeme is too long”)**

**def getChar():**

**global nextChar, charClass, in\_fp**

**nextChar = in\_fp.read(1)**

**if nextChar == ‘’:**

**charClass = ‘EOF’**

**else:**

**if nextChar.isalpha():**

**charClass = ‘LETTER’**

**elif nextChar.isdigit():**

**charClass = ‘DIGIT’**

**else:**

**charClass = ‘UNKNOWN’**

**def getNonBlank():**

**global nextChar**

**while nextChar.isspace():**

**getChar()**

**def lookup(ch):**

**global nextToken**

**if ch == ‘(‘:**

**addChar()**

**nextToken = 25 # LEFT\_PAREN**

**elif ch == ‘)’:**

**addChar()**

**nextToken = 26 # RIGHT\_PAREN**

**elif ch == ‘+’:**

**addChar()**

**nextToken = 21 # ADD\_OP**

**elif ch == ‘-‘:**

**addChar()**

**nextToken = 22 # SUB\_OP**

**elif ch == ‘\*’:**

**addChar()**

**nextToken = 23 # MULT\_OP**

**elif ch == ‘/’:**

**addChar()**

**nextToken = 24 # DIV\_OP**

**else:**

**addChar()**

**nextToken = ‘EOF’**

**return nextToken**

**def lex():**

**global lexLen, nextToken, charClass**

**lexLen = 0**

**getNonBlank()**

**if charClass == ‘LETTER’:**

**addChar()**

**getChar()**

**while charClass == ‘LETTER’ or charClass == ‘DIGIT’:**

**addChar()**

**getChar()**

**nextToken = 11 # IDENT**

**elif charClass == ‘DIGIT’:**

**addChar()**

**getChar()**

**while charClass == ‘DIGIT’:**

**addChar()**

**getChar()**

**nextToken = 10 # INT\_LIT**

**elif charClass == ‘UNKNOWN’:**

**lookup(nextChar)**

**getChar()**

**elif charClass == ‘EOF’:**

**nextToken = ‘EOF’**

**lexeme[0] = ‘E’**

**lexeme[1] = ‘O’**

**lexeme[2] = ‘F’**

**lexeme[3] = ‘\0’**

**print(f”Next token is: {nextToken}, Next lexeme is {‘’.join(lexeme)}”)**

**return nextToken**

**# Main driver**

**Def main():**

**Global in\_fp**

**In\_fp = open(“front.in”, “r”)**

**If in\_fp is None:**

**Print(“ERROR – cannot open front.in”)**

**Else:**

**getChar()**

**while nextToken != ‘EOF’:**

**lex()**

**if \_\_name\_\_ == ‘\_\_main\_\_’:**

**main()**

**-Code documentation**

**Code Structure**

**The main components of the lexer are as follows:**

**Global Variables**

**charClass: Classifies the type of character (LETTER, DIGIT, UNKNOWN, EOF).**

**Lexeme: Array to store the current lexeme being processed.**

**nextChar: The next character read from the input file.**

**lexLen: Length of the current lexeme.**

**Token: Current token being processed.**

**nextToken: The next token to be returned.**

**In\_fp: File pointer for reading input.**

**Functions**

**addChar(): Adds the current character to the lexeme and updates its length.**

**getChar(): Reads the next character from the input file and updates the character class.**

**getNonBlank(): Skips whitespace characters in the input.**

**Lookup(ch): Identifies the token type for a given character and updates the lexeme.**

**Lex(): Main function to generate the next token from the input.**

**Main(): Entry point of the program that initializes file reading and starts the token generation process.**

**-Reference**

**Parr, T. (2022). Language Implementation Patterns:.1**

**Create Your Own Domain-Specific and General Programming Languages with Python Parsons, D. (2021). Introduction to Compiler Design.2**

**Important Note: -**

Technical reports include a mixture of text, tables, and figures. Consider how you can present the information best for your reader. Would a table or figure help to convey your ideas more effectively than a paragraph describing the same data?

Figures and tables should: -

* Be numbered
* Be referred to in-text, e.g. *In Table 1*…, and
* Include a simple descriptive label - above a table and below a figure.