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 :Jana Hesham Student ID:200042820

Crn:4852

**Under Supervision**

Name of Doctor:Nehal Abd EL Salam

Name of T. A.:Eng.Fares

1. **Introduction**
   1. **Phases of Compiler**
2. **Lexical Analyzer**
3. **Software Tools**
   1. **Computer Program**
   2. **Programming Language**
4. **Implementation of a Lexical Analyzer**
5. **References**

**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.

**1. Introduction**

A compiler is a program that converts source code into machine code so that a computer can understand and execute it. The process of compilation consists of multiple phases, starting with Lexical Analysis.

**1.1 Phases of a Compiler**

1 Lexical Analysis – Breaks the source code into smaller parts called “tokens”

2 Syntax Analysis – Checks if the tokens follow the correct grammatical structure.

3 Semantic Analysis – Ensures that the code makes sense logically

4 Intermediate Code Generation– Converts the code into a simpler intermediate representation

5 Optimization – Improves the efficiency of the code. 6Code Generation – Produces the final machine code. Linking & Execution – Prepares the code for execute

**2\_**A **lexical analyzer** (or scanner) is a tool that reads a program's text and breaks it into **tokens**—small meaningful pieces like keywords, numbers, and operators. It's like a pattern matcher that finds specific structures in the text.

**How it Works:**

It reads characters and groups them into **lexemes** (words or symbols).

It assigns a **token type** to each lexeme (e.g., if → **KEYWORD**, 123 →**NUMBER**).

It removes unnecessary elements like **spaces and comments**.

It passes tokens to the **syntax analyzer** for further processing.

**Three Ways to Build a Lexical Analyzer:**

use **regular expressions** and tools like Lex to generate one automatically.

Design a **state transition diagram** and write a program that follows it.

Build a **table-driven** version of the state diagram.

**3\_Software Tool**

Computer Program:

Lexical analyzers can be implemented using tools like \*Lex (Lexical Analyzer Generator)\* and \*Flex (Fast Lexical Analyzer Generator)\*. These tools automatically generate lexical analyzers based on a set of predefined rules.

Programming Language:

Lexical analyzers are commonly written in \*C, Python, or Java\*. These languages provide functions to read input, process characters, and generate tokens efficiently.

.

**Implementation of a Lexical Analyze\*4**

python

import sys

# Token Types

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

class Lexer:

def \_\_init\_\_(self, filename):

self.file = open(filename, 'r')

self.lexeme = ""

self.next\_char = ''

self.next\_token = None

self.get\_char()

def get\_char(self):

"""Reads the next character from the file."""

self.next\_char = self.file.read(1)

if self.next\_char.isalpha():

self.next\_token = IDENT

elif self.next\_char.isdigit():

self.next\_token = INT\_LIT

elif self.next\_char in ['+', '-', '\*', '/']:

self.next\_token = { '+': ADD\_OP, '-': SUB\_OP, '\*': MULT\_OP, '/': DIV\_OP }[self.next\_char]

elif self.next\_char == '=':

self.next\_token = ASSIGN\_OP

elif self.next\_char == '(':

self.next\_token = LEFT\_PAREN

elif self.next\_char == ')':

self.next\_token = RIGHT\_PAREN

elif not self.next\_char:

self.next\_token = EOF

def lex(self):

"""Processes the input file and prints tokens."""

while self.next\_token != EOF:

print(f"Token: {self.next\_token}, Lexeme: {self.next\_char}")

self.get\_char()

self.file.close()

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

lexer = Lexer("front.in")

lexer.lex()

A **Lexical Analyzer** reads input code, breaks it into **tokens**, and sends them to the syntax analyzer.

**Token type:**

These constants **define types of tokens** the lexer will recognize.

Using named constants instead of numbers improves **code readability**.

**The lexer class:**

The constructor (\_\_init\_\_) initializes the lexer by opening the input file and reading the first character.

The lexeme stores the current symbol being processed.

next\_char holds the next character to be processed.

next\_token stores the detected token type.

**The get-char function:**

This function reads the next character and assigns its corresponding token type.

Handles identifiers, numbers, operators, and parentheses.

Marks the end of input with EOF.

**The lex function:**

Loops through the input file, reading one character at a time.

Prints each token and its lexeme.

Stops when the end of the file (EOF) is reached.

Closes the file at the end to free up resources.

**Running the Lexer**

Executes the lexical analysis when the script runs.

Reads the source code from front.in, a sample input file.

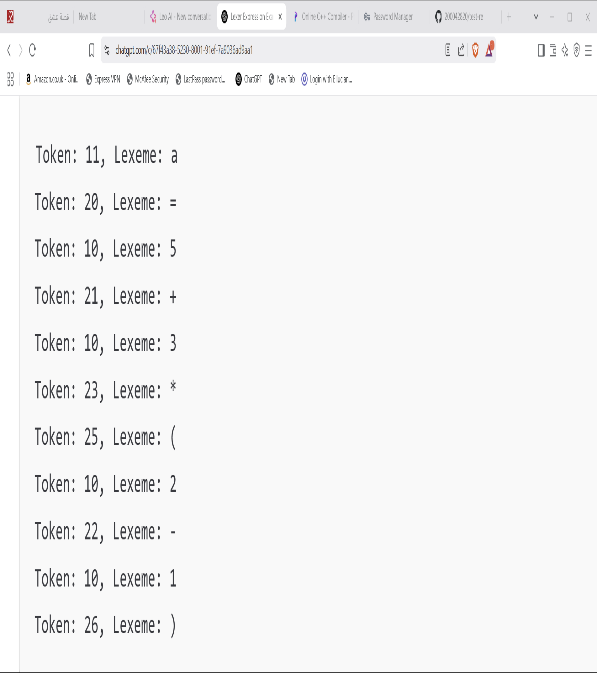
**The expression:**

**a = 5 + 3\*(2-1)**

**This expression contains:**

**Identifiers (a)**

* **Integer literals (5, 3, 2, 1)**
* **Operators (=, +, \*, -)**
* **Parentheses (())**

****

Tokens:11,lexeme:a

Tokens:20, lexeme:=

Tokens:10, lexeme:5

Tokens:21, lexeme:+

Tokens:10, lexeme:3

Tokens:23, lexeme:\*

Tokens:25, lexeme: (

Tokens:10, lexeme:2

Tokens:22, lexeme:-

Tokens:10, lexeme:1

Tokens:26, lexeme: )

**5\_References: textbook: concept of programming language by Robert W.Sebesta ,TWELEFTH EDITION.**

Tokens:

: