LEXICAL ANALYZER

Build Scanner

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

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

.

4*Implementation of a Lexical Analyze

```
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"""
(1)self.next_char = self.file.read
:()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:

: