**Name** – Bhargav Shamuvel Gurav **PRN** – 20410009

**Class** – L.Y. B-Tech (Computer) **Batch** – B1

**Course Code** – CO406U **Course Name** - CDL

**Practical no. A (4)**

**Aim:** Write a program to simulate lexical analyzer for validating operators.

**Theory :**

A lexical analyzer is responsible for recognizing and validating operators in the source code. Here, I'll provide a simplified Python-based example that simulates a lexical analyzer to validate and recognize operators in a given input. We'll consider a few common operators such as +, -, \*, /, and =.

import re

# Define a regular expression pattern for operators

operator\_pattern = r'[+\-\*/=]'

# Define a list of operators

operators = ['+', '-', '\*', '/', '=']

def lexer(input\_string):

# Find all occurrences of operators in the input using regular expression

operator\_matches = re.findall(operator\_pattern, input\_string)

# Initialize a list to store valid operators

valid\_operators = []

for match in operator\_matches:

if match in operators:

valid\_operators.append(match)

else:

print(f"Invalid operator: {match}")

return valid\_operators

# Example usage:

input\_string = "x = 5 + 3 \* 2 - 7 / 4"

result = lexer(input\_string)

print("Valid operators:", result)

In this example:

1. We define a regular expression pattern `operator\_pattern` that matches common operators (+, -, \*, /, =).

2. We define a list of valid operators in the `operators` list.

3. The `lexer` function takes an input string and uses the `re.findall` function to find all occurrences of operators in the input string based on the regular expression pattern.

4. It then iterates through the matches, checking if each match is a valid operator. If it's valid, it adds it to the `valid\_operators` list; otherwise, it prints an error message for invalid operators.

5. The program demonstrates how to use the `lexer` function to extract and validate operators from the input string.

Please note that this is a simplified example. In a real lexical analyzer, you would need to handle a wide range of operators, keywords, and other language-specific constructs. The regular expression and the list of operators should be adapted to match the specific needs of the programming language you're working with.

**Program Code:**

#include <stdio.h>

#include <string.h>

int main ()

{

char arithmetic[5]={'+','-','\*','/','%'};

char relational[4]={'<','>','!','='};

char bitwise[5]={'&','^','~','|'};

char str[2]={' ',' '};

printf ("Enter value to be identified: ");

scanf ("%s",&str);

int i;

if(((str[0]=='&' || str[0]=='|') && str[0]==str[1]) || (str[0]=='!' && str[1]=='\0'))

{

printf("\nIt is Logical operator");

}

for(i=0;i<4;i++)

{

if(str[0]==relational[i]&&(str[1]=='='||str[1]=='\0'))

{

printf("\n It is releational Operator"); break;

}

}

for(i=0;i<4;i++)

{

if((str[0]==bitwise[i] && str[1]=='\0') || ((str[0]=='<' || str[0]=='>') && str[1]==str[0]))

{

printf("\n It is Bitwise Operator"); break;

}

}

if(str[0]=='?' && str[1]==':')

printf("\nIt is ternary operator");

for(i=0;i<5;i++)

{

if((str[0]=='+' || str[0]=='-') && str[0]==str[1])

{

printf("\nIt is unary operator"); break;

}

else if((str[0]==arithmetic[i] && str[1]=='=') || (str[0]=='=' && str[1]==' '))

{

printf("\nIt is Assignment operator"); break;

}

else if(str[0]==arithmetic[i] && str[1]=='\0')

{

printf("\nIt is arithmetic operator"); break;

}

}

return 0;

}

**Output:**

****

**Conclusion :** In this practical we learnt how lexical analyzer validates the operators from a program.