Name	Roll No.	Section	DoP
Vidhan Dahatkar	166	B	

Practical - 1(a)

Aim- Implement a Lexical Analyzer using FLEX and develop:

A. Program For converting all small case letters to UPPER case letters and Vice-Versa.

Theory:

Lexical analysis, also known as scanning or lexing, is the initial phase of the compilation process in which the source code is analyzed and broken down into a sequence of **tokens**. Tokens are the smallest units of meaning in a programming language, such as keywords, identifiers, literals, and operators. The purpose of lexical analysis is to simplify the code by reducing it to a more manageable form for further processing by the compiler or interpreter.

Key Concepts in Lexical Analysis

1. **Tokens**
   A token is a group of characters that represent a single unit of meaning in a programming language. For example, in the statement int x = 10;, the tokens are:

   - int (keyword)
   - x (identifier)
   - = (operator)
   - 10 (literal)
   - ; (punctuation)

   Tokens can be classified into different categories such as:

   - **Keywords**: Reserved words with predefined meaning (e.g., int, if, while).
   - **Identifiers**: Names used to identify variables, functions, etc. (e.g., x).
   - **Operators**: Symbols that perform operations on data (e.g., +, =, *).
   - **Literals**: Constants or fixed values (e.g., 10, "hello").
   - **Punctuation**: Symbols that help structure the code (e.g., ;, {}, (), ,).

2. **Regular Expressions**
   Regular expressions are patterns that describe sets of strings. In lexical analysis, regular expressions are used to define the patterns for different types of tokens. These patterns are then used by the lexical analyzer to recognize and extract tokens from the source code.

3. **Lexemes**
   A lexeme is the sequence of characters in the source code that matches the pattern defined by a regular expression. Lexemes are the actual instances of tokens found in the source code. For instance, in the statement int x = 10;, the lexeme for the identifier x is the string "x."

4. **Finite Automata**
   Lexical analyzers often use finite automata to recognize and process tokens efficiently. Finite automata are abstract machines with a finite set of states and transitions between these states based on input characters. Lexical rules are often implemented using finite automata to determine the valid sequences of characters that form tokens.

5. **Lexer Generators (e.g., FLEX)**
   Lexer generators like FLEX take high-level specifications of lexical rules, typically written in the form of regular expressions, and generate efficient C or C++ code for the lexical analyzer. This makes the process of building a lexer for a specific programming language more convenient and less error-prone.

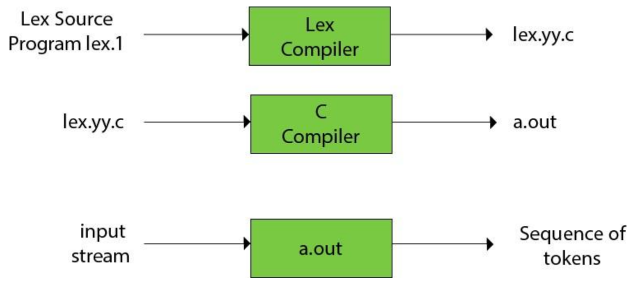


Figure1LexicalAnalyzer

A Lex program is separated into three sections by %% delimiters. The formal of Lex source is as follows:

1. {definitions }
2. %%
3. {rules}
4. %%
5. {usersubroutines }

Definitions include declarations of constant, variable and regular definitions.

Rules define the statement of form p1{action1}p2 {action2} ..... pn{action}.
Where **pi** describes the regular expression and **action1** describes the actions what action the lexical analyzer should take when pattern pi matches a lexeme.

User subroutines are auxiliary procedures needed by the actions. The subroutine can beloaded with the lexical analyzer and compiled separately.

To run the program ,it should be first saved with the extension **.l or.lex**. Run the below commands on terminal in order to run the program file.

Step1: flex filename.l or flex filename.lex depending on the extension file is saved with.

Step2:gcc lex.yy.c

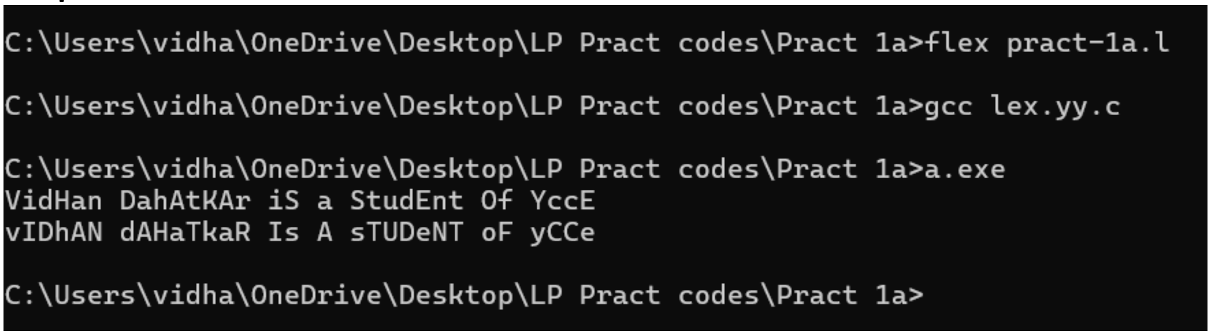
Step3: /a.out

Step 4: Provide the input to program in case it is required

Program:

```
%{
#include <stdio.h>
%}

lower [a-z]
upper [A-Z]

%%

{lower}   { printf("%c", yytext[0] - 32); }  // Convert lowercase to uppercase
{upper}   { printf("%c", yytext[0] + 32); }  // Convert uppercase to lowercase
[\t\n]+   { printf("\n"); }  // Handle tabs and newlines
.       { printf("%c", yytext[0]); }  // Print other characters as is

%%

int yywrap() {
   return 1;  // Indicate end of file (EOF)
}

int main() {
   yylex();  // Start lexical analysis
   return 0;
}
```

Output:



```
C:\Users\vidha\OneDrive\Desktop\LP Pract codes\Pract 1a>flex pract-1a.l

C:\Users\vidha\OneDrive\Desktop\LP Pract codes\Pract 1a>gcc lex.yy.c

C:\Users\vidha\OneDrive\Desktop\LP Pract codes\Pract 1a>a.exe
VidHan DahAtKAr iS a StudEnt Of YccE
vIDhAN dAHaTkaR Is A sTUDeNT oF yCCe

C:\Users\vidha\OneDrive\Desktop\LP Pract codes\Pract 1a>
```

Conclusion:
With FLEX, the program can smartly figure out and count things like words, spaces, and lines in a file. This shows that FLEX is good at understanding and working with different parts of language. These programs prove that FLEX is really flexible and can be used for many language tasks. In the big picture, they help make programs that understand language better, showing that FLEX is great for creating tools that focus on specific goals.

Name	Roll No.	Section	DoP
Vidhan Dahatkar	166	B	

Practical - 1(b)

Aim- Implement a Lexical Analyzer using FLEX and develop:
B. Program to count the words, spaces, and lines in a given input file.

Theory:

A **Lexical Analyzer** (also known as a lexer or scanner) is a critical component in the process of **compilation** or **interpretation** of a programming language. It performs the task of reading a source code (input text) and breaking it down into a sequence of tokens—meaningful units that the compiler or interpreter can work with. Lexical analysis helps in categorizing and recognizing different components of the source code, such as keywords, operators, identifiers, literals, and even whitespace characters.
In this practical exercise, the aim is to **implement a Lexical Analyzer using FLEX** that can **count the number of words, spaces, and lines** in a given input file. This program will analyse the input text and keep track of these components, providing valuable insights into the structure of the input text.

Components of the Task:

Words:

- **Words** can be defined as any sequence of characters that are separated by spaces, punctuation marks, or line breaks. In this case, we are particularly interested in **Alpha numeric sequences** that form meaningful units, such as identifiers, keywords, or any other string of characters that can be considered a word.
- **Pattern for Words**: In the context of this task, a word can be defined as a sequence of one or more alphabetic characters and digits ([a-zA-Z0-9]+), which FLEX will recognize as a token.

Spaces:

- **Spaces** refer to the whitespace characters in the input text, which include spaces (''), tabs ('\t'), and newlines ('\n').
- **Pattern for Spaces**: The regular expression to match spaces can be something like [\t\n
-]+ which cover stabs, newlines, and spaces. FLEX will count these spaces as it processes the input.

Lines:

- A **line** is defined by the presence of new line characters ('\n'), marking the end of one line of text and the beginning of another.
- **Pattern for Lines**: Every time a new line character is encountered, the program will increment the line counter.

FLEX and its Role:
FLEX is a **Lexical Analyzer Generator** that takes regular expressions as input and generates a C program that can recognize patterns in text and execute actions when those patterns are matched. It provides an easy way to create a lexer for analysing text, making it a powerful tool in language processing .

FLEX Program Structure:
A typical FLEX program consists of three parts:

1. **Definitions Section**: In this section, regular expressions and C declarations are defined. It may also include variables like counters for words, spaces, and lines.
2. **Rules Section**: This is where the patterns (regular expressions) are defined along with their corresponding actions (i.e., what happens when a pattern is matched).
3. **User Subroutines Section**: This section can include C code, such as helper functions or additional code for printing results.

Steps for the Program:
The following steps are involved in developing a FLEX program to count the number of words, spaces, and lines:
1. Define Regular Expressions:

- Define are regular expression for **words**, which could include sequences of alphabetic characters and numbers ([a-zA-Z0-9]+).
- Define regular expressions for **spaces** and **newlines** ([\t\n]+for spaces and tabs, and
- \n for line breaks).

2. Action for Counting:

- When a word is matched, increment the word counter.
- When spaces are matched, increment the space counter.
- When a newline character is encountered, increment the line counter.

3. Print the Results:

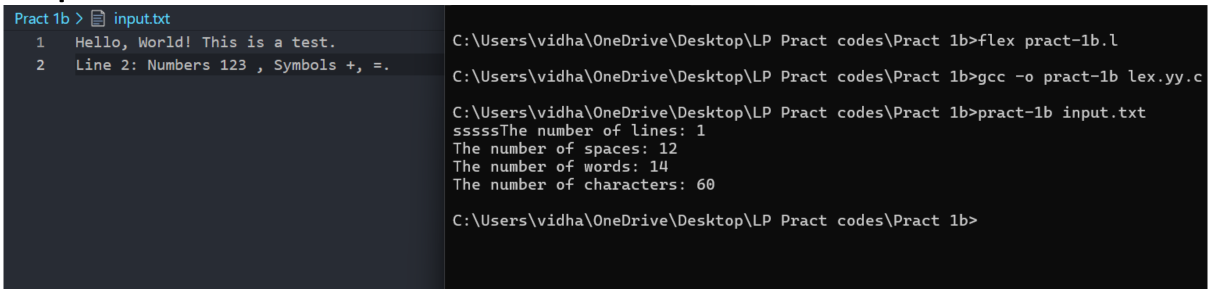
- After the input file is processed, the program will print the total counts for words, spaces, and lines.

Program:

```
%{
#include <stdio.h>
int lines = 0, spaces = 0, words = 0, characters = 0;
int in_word = 0;
%}

%%
\n          { lines++; characters++; in_word = 0; }  // Increment lines and characters on newline
" "         { spaces++; characters++; in_word = 0; }  // Increment spaces and characters
```

```
[^\n\s]      { characters++; if (!in_word) { words++; in_word = 1; } } // Count words and
characters

%%
int main(int argc, char *argv[]) {
   if (argc < 2) {
      fprintf(stderr, "Usage: %s <input_file>\n", argv[0]);
      return 1;
   }
   FILE *file = fopen(argv[1], "r");  // Open the file passed as argument
   if (file == NULL) {
      perror("Error opening file");
      return 1;
   }
   yyin = file;  // Set the input file for FLEX
   yylex();      // Start lexical analysis
   printf("The number of lines: %d\n", lines);
   printf("The number of spaces: %d\n", spaces);
   printf("The number of words: %d\n", words);
   printf("The number of characters: %d\n", characters);
   fclose(file);  // Close the file after processing
   return 0;
}

int yywrap() {
   return 1;  // Indicate end of input
}
```

Output:



Pract 1b > input.txt

```
Hello, World! This is a test.
Line 2: Numbers 123 , Symbols +, =.
```

```
C:\Users\vidha\OneDrive\Desktop\LP Pract codes\Pract 1b>flex pract-1b.l

C:\Users\vidha\OneDrive\Desktop\LP Pract codes\Pract 1b>gcc -o pract-1b lex.yy.c

C:\Users\vidha\OneDrive\Desktop\LP Pract codes\Pract 1b>pract-1b input.txt
sssssThe number of lines: 1
The number of spaces: 12
The number of words: 14
The number of characters: 60

C:\Users\vidha\OneDrive\Desktop\LP Pract codes\Pract 1b>
```

Conclusion:

In this exercise, we have implemented a **lexical analyzer using FLEX** to process a given input file and count the number of **words**, **spaces**, and **lines**. The ability to perform lexical analysis using regular expressions and associating them with actions is a powerful feature of FLEX, making it a valuable tool for various language processing tasks. The resulting program demonstrates how FLEX can be used for simple text analysis, such as counting different types of tokens in an input file.