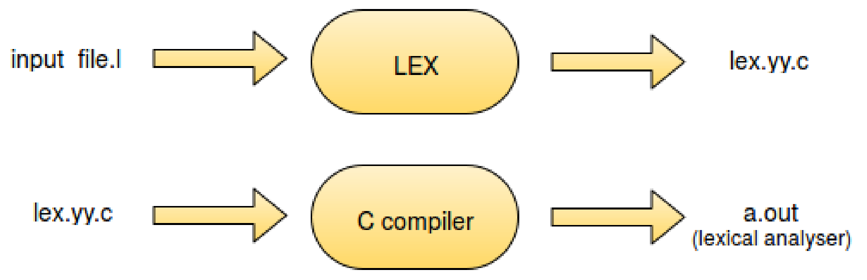
Experiment No.	**05**
Experiment Title	**Lexical Analyzer Tool : Flex** **Task:** **Write lexical analyzer to identify tokens for c Language like identifier, literals, open curly brace, end curly brace, semicolon, ignore white space.**
Student Name	Ronak Surve
Roll No.	64
Objectives:	**Students will be able to understand flex tool and create Lexical Analyzer**
Theory /Algorithm:	**1) LEX** is a tool for generating **lexical analyzers** (also known as scanners or tokenizers) in C or C++. A lexical analyzer reads input text and breaks it up into a sequence of tokens, which are meaningful units of the programming language being analyzed. These tokens can then be passed on to a parser or other program for further processing. The LEX tool works by taking a set of regular expressions and actions as input and generating a C or C++ program that performs the lexical analysis. The input to LEX is a file called a specification file that contains regular expressions and corresponding actions. The LEX tool then generates a C or C++ program that recognizes tokens in the input text based on the regular expressions specified in the specification file.  input file.l LEX lex.yy.c lex.yy.c C compiler a.out (lexical analyser) Technically, LEX translates a set of regular expression specifications (given as input in input_file.l) into a C implementation of a corresponding finite state machine (lex.yy.c). This C program, when compiled, yields an executable lexical analyzer.

2) The **LEX tool** provides several functions to generate lexical analyzers for programming languages and other applications that require parsing and analysis of input text. Here are some of the key functions of LEX:

Generating lexical analyzers: The primary function of LEX is to generate a lexical analyzer (also known as a scanner or tokenizer) for a given input specification. The input specification contains a set of regular expressions and corresponding actions, which are used to recognize tokens in the input text. LEX generates a C or C++ program that performs the lexical analysis based on the input specification.

Defining regular expressions: LEX provides a way to define regular expressions that are used to recognize tokens in the input text. Regular expressions are patterns that describe sequences of characters, such as numbers, strings, or identifiers. LEX provides a set of operators for building regular expressions, such as | for alternation, * for repetition, and () for grouping.

Specifying actions: LEX allows you to specify actions that are taken when a particular regular expression is recognized in the input text. Actions can include any C or C++ code that you want to execute when a token is recognized, such as updating a symbol table or emitting a warning message.

Handling input and output: LEX provides functions for handling input and output in the generated scanner. By default, the scanner reads input from the standard input and writes output to the standard output. However, you can customize the input and output functions to read input from a file or a network socket, or write output to a log file or a database.

Defining start conditions: LEX allows you to define start conditions, which are used to switch between different sets of regular expressions and actions based on the current context of the input text. For example, you might define a start condition for recognizing comments within a program, and switch back to the default start condition when the comment is finished.

3) Regular expressions are used in LEX to describe patterns in the input text that correspond to specific tokens. Regular expressions are composed of a combination of literals, character classes, quantifiers, and other operators that describe the pattern to be matched.

Here are some examples of how to write regular expressions in a

LEX specification:

Matching numbers: To match a number in the input text, you can use the regular expression [0-9]+, which matches one or more digits.

```
[0-9]+   { /* action for number token */ }
```

Matching identifiers: To match an identifier in the input text (i.e., a variable or function name), you can use the regular expression [a-zA-Z_][a-zA-Z0-9_]*, which matches a letter or underscore followed by zero or more letters, digits, or underscores.

```
[a-zA-Z_][a-zA-Z0-9_]*   { /* action for identifier token */ }
```

Matching strings: To match a string literal in the input text (i.e., a sequence of characters enclosed in double quotes), you can use the regular expression \"[^\"\n]*\", which matches a double quote, followed by any number of non-double-quote characters, followed by another double quote.

```
\"[^\"\n]*\"   { /* action for string token */ }
```

Matching whitespace: To match whitespace (i.e., spaces, tabs, and newlines) in the input text, you can use the regular expression [\t\n]+, which matches one or more spaces, tabs, or newlines.

```
[ \t\n]+   /* ignore whitespace */
```

Matching operators: To match operators (i.e., symbols that represent mathematical or logical operations), you can use a combination of literal characters and character classes. For example, to match the plus sign (+) and minus sign (-), you can use the regular expression [\+\-], which matches either a plus or a minus sign.

```
[\+\-]   { /* action for plus or minus token */ }
```

These are just a few examples of how to write regular expressions in a LEX specification. The specific regular expressions used will depend on the language being analyzed and the tokens that need to be recognized.

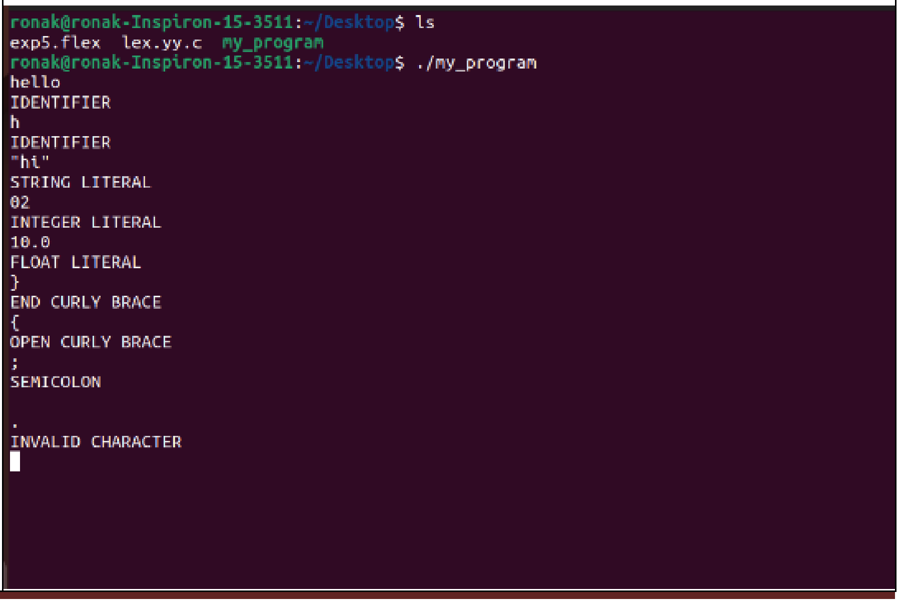
Program Code:	%{
	 #include <stdio.h>
	 %}
	
	 %%

```
[a-zA-Z_][a-zA-Z0-9_]*  { printf("IDENTIFIER\n"); }
[0-9]+                  { printf("INTEGER LITERAL\n"); }
[0-9]*\.[0-9]+          { printf("FLOAT LITERAL\n"); }
\"([^\\\n]|(\\.))*\"    { printf("STRING LITERAL\n"); }
\'([^\\\n]|(\\.))*\'    { printf("CHARACTER LITERAL\n"); }
\{                      { printf("OPEN CURLY BRACE\n"); }
\}                      { printf("END CURLY BRACE\n"); }
\;                      { printf("SEMICOLON\n"); }
[ \t\n]+                /* Ignore whitespace */
.                       { printf("INVALID CHARACTER\n"); }
%%

int main(int argc, char** argv)
{
    yylex();
    return 0;
}
int yywrap() {
    return 1;
}
```

Input to the Program:

```
Hello
h
“hi”
02
10.0
}
{
;


.
```

Output of the program:



```
ronak@ronak-Inspiron-15-3511:~/Desktop$ ls
exp5.flex  lex.yy.c  my_program
ronak@ronak-Inspiron-15-3511:~/Desktop$ ./my_program
hello
IDENTIFIER
h
IDENTIFIER
"hi"
STRING LITERAL
02
INTEGER LITERAL
10.0
FLOAT LITERAL
}
END CURLY BRACE
{
OPEN CURLY BRACE
;
SEMICOLON

.
INVALID CHARACTER
```

Outcome of the Experiment:	1) Successfully created and executed C code for lexical analyzer in **LEX** tool. 2) Understanding the function of lexical analysis in compiler design. 3) Understanding the working of LEX tool.
References:	https://silcnitc.github.io/lex.html