FR. CONCEICAO RODRIGUES COLLEGE OF ENGINEERING

System Programming and Compiler Construction

VI Semester (Computer) **Academic Year: 22-23**

Experiment No 5

Aim : Study of Lexical analyzer tool -Flex/ Lex

Leraning Objective: Recognise lexical pattern from given input file

Theory:

Lex is a program generator designed for lexical processing of character input streams. It accepts a high-level, problem oriented specification for character string matching, and produces a program in a general purpose language which recognizes regular expressions. The regular expressions are specified by the user in the source specifications given to Lex. The Lex written code recognizes these expressions in an input stream and partitions the input stream into strings matching the expressions. At the boundaries between strings program sections provided by the user are executed. The Lex source file associates the regular expressions and the program fragments. As each expression appears in the input to the program written by Lex, the corresponding fragment is executed. The user supplies the additional code beyond expression matching needed to complete his tasks, possibly including code written by other generators. The program that recognizes the expressions is generated in the general purpose programming language employed for the user's program fragments. Thus, a high level expression language is provided to write the string expressions to be matched while the user's freedom to write actions is unimpaired. This avoids forcing the user who wishes to use a string manipulation language for input analysis to write processing programs in the same and often inappropriate string handling language.

Lex is not a complete language, but rather a generator representing a new language feature which can be added to different programming languages, called ``host languages.'' Just as general purpose languages can produce code to run on different computer hardware, Lex can write code in different host languages. The host language is used for the output code generated by Lex and also for the program fragments added by the user. Compatible runtime libraries for the different host languages are also provided. This makes Lex adaptable to different environments and different users. Lex itself exists on UNIX, GCOS, and OS/370; but the code generated by Lex may be taken anywhere where appropriate compilers exist.

Lex turns the user's expressions and actions (called source in this pic) into the host general-purpose language; the generated program is named yylex. The yylex program will recognize expressions in a stream (called input in this pic) and perform the specified actions for each expression as it is detected.

```
                 +-------+
     Source ->   |  Lex  |   -> yylex
                 +-------+

                 +-------+
     Input ->    | yylex | -> Output
                 +-------+

           An overview of Lex
```

For a trivial example, consider a program to delete from the input all blanks or tabs at the ends of lines.

```
    %
    [ \t]+$ ;
```

is all that is required. The program contains a %% delimiter to mark the beginning of the rules, and one rule. This rule contains a regular expression which matches one or more instances of the characters blank or tab (written \t for visibility, in accordance with the C language convention) just prior to the end of a line. The brackets indicate the character class made of blank and tab; the + indicates ``one or more ...''; and the $ indicates ``end of line,'' as in QED. No action is specified, so the program generated by Lex (yylex) will ignore these characters. Everything else will be copied. To change any remaining string of blanks or tabs to a single blank, add another rule:

```
%%
[ \t]+$ ;
[ \t]+ printf(" ");
```

The finite automaton generated for this source will scan for both rules at once, observing at the termination of the string of blanks or tabs whether or not there is a newline character, and executing the desired rule action. The first rule matches all strings of blanks or tabs at the end of lines, and the second rule all remaining strings of blanks or tabs.

Lex can be used alone for simple transformations, or for analysis and statistics gathering on a lexical level. Lex can also be used with a parser generator to perform the lexical analysis phase.

The general format of Lex source is:

```
{definitions}
%%
{rules}
%%
{user subroutines}
```

where the definitions and the user subroutines are often omitted. The second %% is optional, but the first is required to mark the beginning of the rules. The absolute minimum Lex program is thus

```
%%
```

(no definitions, no rules) which translates into a program which copies the input to the output unchanged. In the outline of Lex programs shown above, the rules represent the user's control decisions; they are a table, in which the left column contains regular expressions and the right column contains actions, program fragments to be executed when the expressions are recognized. Thus an individual rule might appear integer printf("found keyword INT"); to look for the string integer in the input stream and print the message ``found keyword INT'' whenever it appears. In this example the host procedural language is C and the C library function printf is used to print the string. The end of the expression is indicated by the first blank or tab character. If the action is merely a single C expression, it can just be given on the right side of the line; if it is compound, or takes more than a line, it should be enclosed in braces.

Implementation Details:
1. Open file in text editor
2. Enter keywords, rules for identifier and constant, operators and relational operators. In the following format

    a) %{
    Definition of constant /header files
    %}
    b) Regular Expressions
    %%
    Transition rules
    %%
    c) Auxiliary Procedure (main( ) function)

3. Save file with **.l** extension e.g. **Mylex.l**
4. Call lex tool on the terminal e.g. [root@localhost]# lex Mylex.l This lex tool will convert ".l" file into ".c" language code file i.e. **lex.yy.c**
5. Compile the file lex.yy.c e.g. **gcc lex.yy.c** .After compiling the file lex.yy.c, this will create the output file **a.out**
6. Run the file a.out e.g. **./a.out**
7. Give input on the terminal to the **a.out** file upon processing output will be displayed

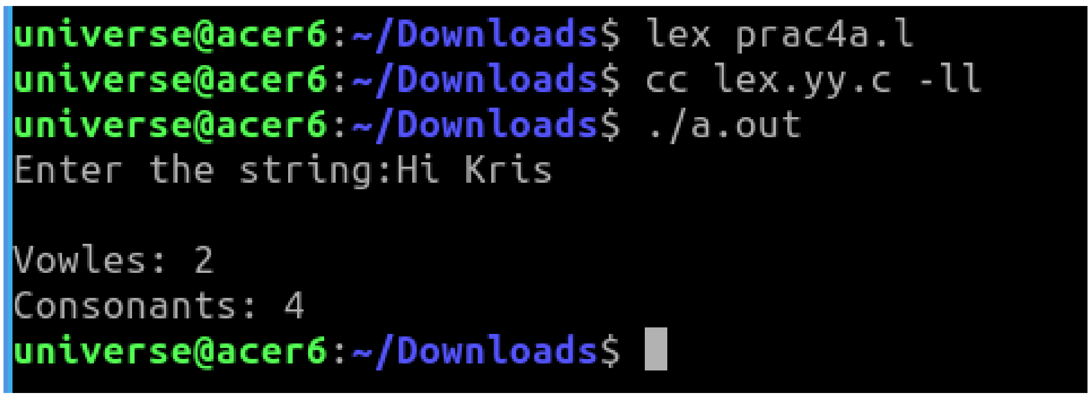
Source Code:

1: Program for counting number of vowels and consonant

```
%{
#include <stdio.h>
int vow=0,con=0;
%}
%%
[aeiouAEIOU] {vow++;}
[a-zA-Z] {con++;}
%%
int yywrap(){
    return 1;
}
int main(){
    printf("Enter the string:");
    yylex();
    printf("Vowles: %d \nConsonants: %d\n",vow,con);
    return 0;
}
```

Output:

#lex alphalex.l
#gcc lex.yy.c
#./a.out
This Lex program counts the number of vowels and consonants in given text.
Enter the text and terminate it with CTRL-d.

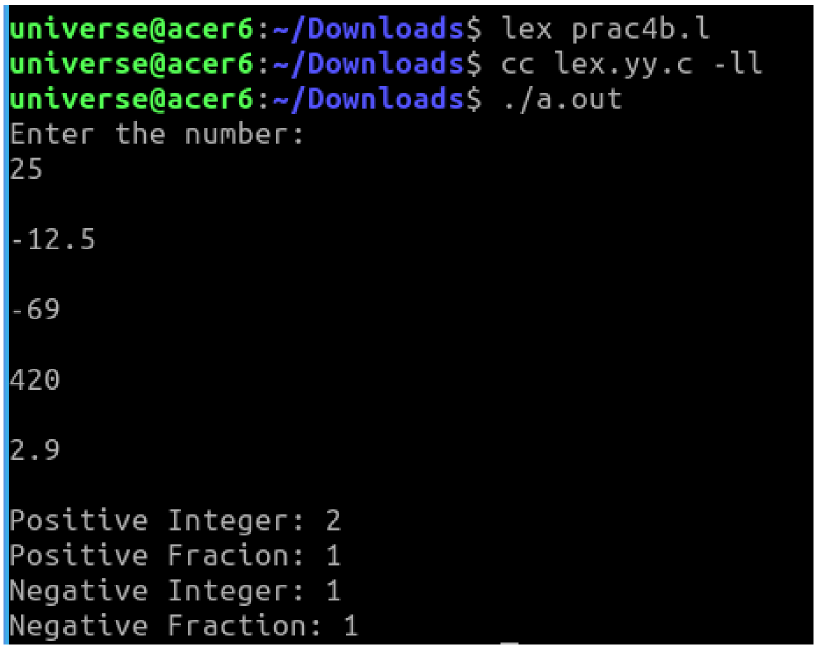


```
universe@acer6:~/Downloads$ lex prac4a.l
universe@acer6:~/Downloads$ cc lex.yy.c -ll
universe@acer6:~/Downloads$ ./a.out
Enter the string:Hi Kris

Vowles: 2
Consonants: 4
universe@acer6:~/Downloads$
```

2: Lexical Analyzer for Number Classification

```
%{
#include <stdio.h>
int pos_int = 0,pos_frac = 0;
int neg_int = 0,neg_frac = 0;
%}
%%
[+]?[0-9]* {pos_int++;}
[+]?[0-9]*\.[0-9]+ {pos_frac++;}
[-][0-9]* {neg_int++;}
[-][0-9]*\.[0-9]+ {neg_frac++;}
%%
int yywrap(){
    return 1;
}
int main(){
    printf("Enter the number:\n");
    yylex();
    printf("Positive Integer: %d\n",pos_int);
    printf("Positive Fracion: %d\n",pos_frac);
    printf("Negative Integer: %d\n",neg_int);
    printf("Negative Fraction: %d\n",neg_frac);
    return 0;
}
```

Output:



```
universe@acer6:~/Downloads$ lex prac4b.l
universe@acer6:~/Downloads$ cc lex.yy.c -ll
universe@acer6:~/Downloads$ ./a.out
Enter the number:
25

-12.5

-69

420

2.9

Positive Integer: 2
Positive Fracion: 1
Negative Integer: 1
Negative Fraction: 1
```

FR. CONCEICAO RODRIGUES COLLEGE OF ENGINEERING

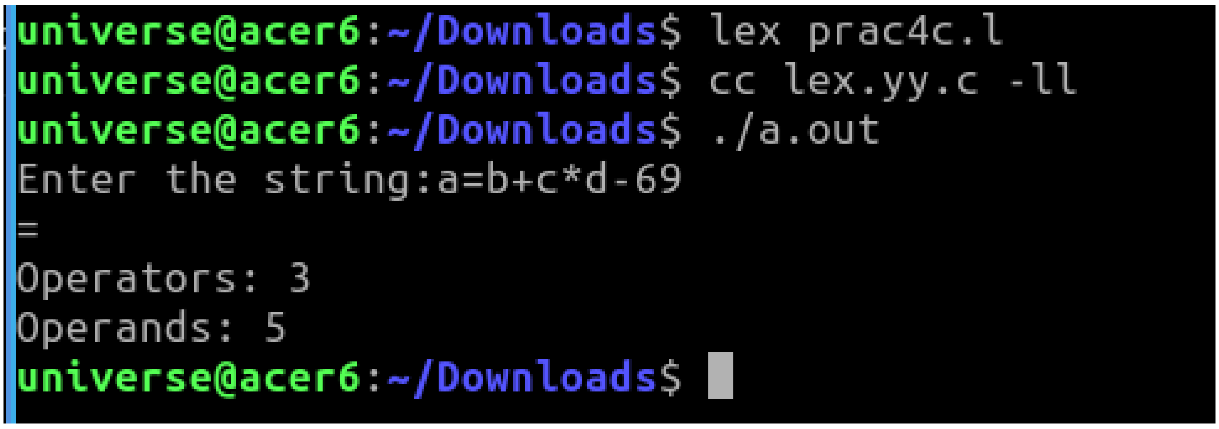
System Programming and Compiler Construction

VI Semester (Computer) **Academic Year: 22-23**

3: Lexical Analyzer for Counting Operators and Operands in a String

```
%{
#include <stdio.h>
int operator=0,operand=0;
%}
%%
[+\-\*\/\%^] {operator++;}
[a-zA-Z0-9]+ {operand++;}
%%
int yywrap(){
    return 1;
}
int main(){
    printf("Enter the string:");
    yylex();
    printf("Operators: %d \nOperands: %d\n",operator,operand);
    return 0;
}
```

Output:

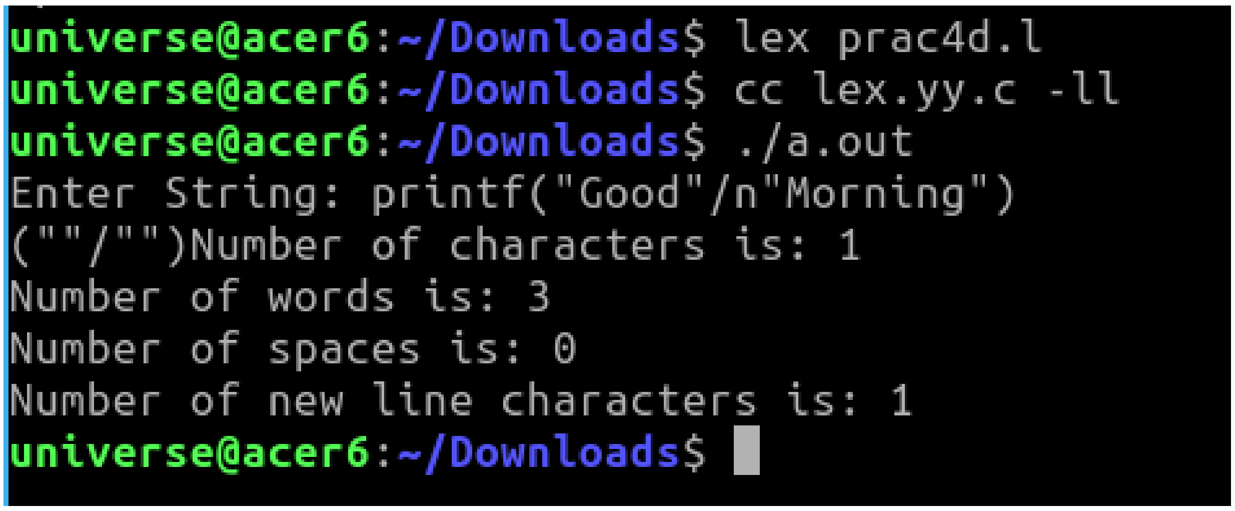


```
universe@acer6:~/Downloads$ lex prac4c.l
universe@acer6:~/Downloads$ cc lex.yy.c -ll
universe@acer6:~/Downloads$ ./a.out
Enter the string:a=b+c*d-69
=
Operators: 3
Operands: 5
universe@acer6:~/Downloads$ 
```

4: **Lex program to count characters, words, spaces, and new lines in a string.**

```
%{
#include <stdio.h>
int chars = 0, word = 0, newLine = 0, space = 0;
%}

%%
[a-zA-Z]{1} {chars++;}
[a-zA-Z]{2,} {word++;}
[ ] {space++;}
[\n] {newLine++;}

%%

int yywrap(){
return 1;
}

int main()
{
printf("Enter String: ");
yylex();
printf("Number of characters is: %d\n", chars);
printf("Number of words is: %d\n", word);
printf("Number of spaces is: %d\n", space);
printf("Number of new line characters is: %d\n", newLine);
}
```

Output:

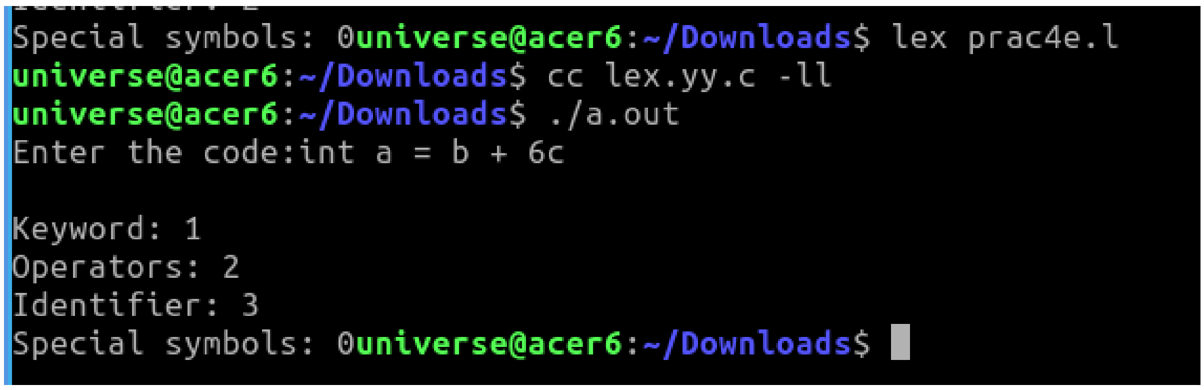


```
universe@acer6:~/Downloads$ lex prac4d.l
universe@acer6:~/Downloads$ cc lex.yy.c -ll
universe@acer6:~/Downloads$ ./a.out
Enter String: printf("Good"/n"Morning")
(""/"")Number of characters is: 1
Number of words is: 3
Number of spaces is: 0
Number of new line characters is: 1
universe@acer6:~/Downloads$ 
```

5: Lex program to count characters, words, spaces, and new lines in a string.

```
%{
#include <stdio.h>
int keyword=0;
int operator=0;
int special=0;
int identifier=0;
%}
%%
[int||char||float||return||if||else||do||while] {keyword++;}
[=+\-\*\/^] {operator++;}
[a-zA-Z0-9]+ {identifier++;}
[%/&/$/;] {special++;}

%%
int yywrap(){
    return 1;
}
int main(){
    printf("Enter the code:");
    yylex();
    printf("Keyword: %d \nOperators: %d \nIdentifier: %d \nSpecial
symbols: %d",keyword,operator,identifier,special);
    return 0;
}
```

Output:



```
Special symbols: 0universe@acer6:~/Downloads$ lex prac4e.l
universe@acer6:~/Downloads$ cc lex.yy.c -ll
universe@acer6:~/Downloads$ ./a.out
Enter the code:int a = b + 6c

Keyword: 1
Operators: 2
Identifier: 3
Special symbols: 0universe@acer6:~/Downloads$
```

FR. CONCEICAO RODRIGUES COLLEGE OF ENGINEERING

System Programming and Compiler Construction

VI Semester (Computer) **Academic Year: 22-23**

Aim : Study of Parser generator tool – Yacc

Leraning Objective:

Theory:

Parser for a grammar is a program which takes in the language string as its input and produces either a corresponding parse tree or a error. Syntax of a Language The rules which tells whether a string is a valid program or not are called the syntax Semantic s of Language The rules which give meaning to programs are called the semantic of a language Tokens When a string representing a program is broken into sequence of substrings, such that each substring represents a constant, identifier, operator, keyword etc of the language, these substrings are called the tokens of the language.

Lexical Analysis

The function of lexical Analyzer is to read the input stream representing the source program , one character at a time and translate into valid tokens.

Implementation Details

1: Create a lex file

The general format for lex file consists of three sections:

1. Definitions
2. Rules
3. User subroutine Section

Definitions consists of any external 'C' definitions used in the lex actions or subroutines. The other types of definitions are definitions are lex definitions which are essentially the lex substitution strings, lex start states and lex table size declarations. The rules is the basic part which specifies the regular expressions and their corresponding actions. The user Subroutines are the functions that are used in the Lex actions.

2 : Yacc is the Utility which generates the function 'yyparse' which is indeed the Parser. Yacc describes a context free, LALR(1) grammer and supports both bottom up and top-down parsing. The general format for the yacc file is very similar to that of the lex file.

1. Declarations
2. Grammar Rules
3. Subroutines

In declarations apart from the legal 'C' declarations here are few Yacc specific declarations which begins with a % sign.

1. % union It defines the Stack type for the Parser.
   It is union of various datas/structures/objects.
2. % token These are the terminals returned by the yylex function to the yacc. A token cal also have type associated with it for good type checking and syntax directed translation. A type of a token can be specified as % token <stack member> tokenName.
3. %type The type of non-terminal symbol in the grammar rule can be specified with this. The format is %type <stack member> non termainal.
4. % noassoc Specifies that there is no associativity of a terminal symbol.
5. % left Specifies the left associativity of a terminal symbol.
6. % right Specifies the right associativity of a terminal symbol.

7. % start specifies the L.H.S. non-terminal symbol of a production rule which specifies starting point of grammar rules.
8. % prac changes the precedence level associated with a particular rule to that of the following token name or literal.

The Grammar rules are specified as follows:

Context free grammar production-

p-> AbC

Yacc Rule-

P: A b C { /* ‘C’ actions*/}

The general style of coding the rules is to have all Terminals in lower –case and all non-terminals in upper –case.

To facilitate a proper syntax directed translation the Yacc has something calls pseudo-variables which forms a bridge between the values of terminals/non-terminals and the actions. These pseudo variables are $$, $1, $2, $3,…………The $$ is the L.H.S value of the rule whereas $1 is the first R. H. S value of the rule, so is the $2 etc. The default type for pseudo variables is integer unless they are specified by % type.

%token <type> etc.

Perform the following steps, in order, to create the desk calculator example program:

1. Process the **yacc** grammar file using the **-d** optional flag (which tells the **yacc** command to create a file that defines the tokens used in addition to the C language source code):
   ```
   yacc -d calc.yacc
   ```
2. Use the **li** command to verify that the following files were created:

   y.tab.c The C language source file that the **yacc** command created for the parser.

   y.tab.h A header file containing define statements for the tokens used by the parser.
3. Process the **lex** specification file:
   ```
   lex calc.lex
   ```
4. Use the **li** command to verify that the following file was created:

   lex.yy.c The C language source file that the **lex** command created for the lexical analyzer.
5. Compile and link the two C language source files:
   ```
   cc y.tab.c lex.yy.c
   ```
6. Use the **li** command to verify that the following files were created:

   y.tab.o The object file for the **y.tab.c** source file

   lex.yy.o The object file for the **lex.yy.c** source file

   a.out The executable program file
7. To then run the program directly from the **a.out** file, enter:
8.
```
$ a.out
```

first.l:

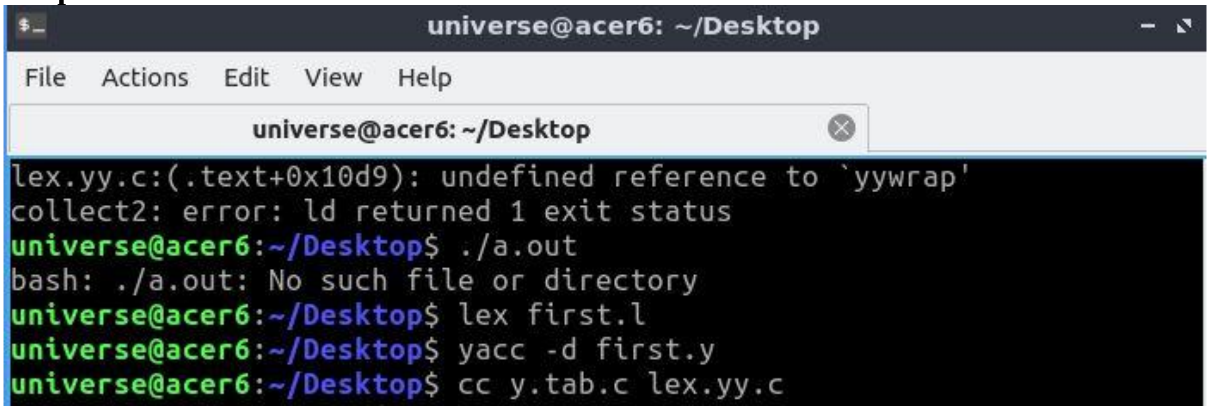
```
%{
    /* Definition section*/
#include "y.tab.h"
extern yylval;
%}

%%
[0-9]+  { yylval = atoi(yytext); return NUMBER;  }
[a-zA-Z]+ { return ID; }
[\t]+ ;
\n { return 0; }
. { return yytext[0]; }
%%
int yywrap()
{return 1;}
```

first.y:

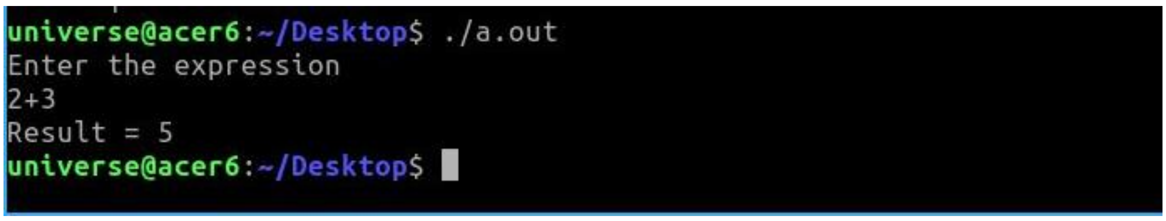
```
%{
  /* Definition section */
#include<stdio.h>
%}

%token NUMBER ID
// setting the precedence
// and associativity of operators
%left '+' '-'
%left '*' '/'

/* Rule Section */
%%
 E : T { printf("Result = %d\n", $$);   return 0;  }

T : T '+' T { $$ = $1 + $3; }
| T '-' T { $$ = $1 - $3; }
| T '*' T { $$ = $1 * $3; }
| T '/' T { $$ = $1 / $3; }
| NUMBER { $$ = $1; }

%%
```

```
   int main()
 {
printf("Enter the expression\n");
yyparse();
 }

/* For printing error messages */
int yyerror(char* s)
 {
printf("\nExpression is invalid\n");
 }
```

Output:



```
lex.yy.c:(.text+0x10d9): undefined reference to `yywrap'
collect2: error: ld returned 1 exit status
universe@acer6:~/Desktop$ ./a.out
bash: ./a.out: No such file or directory
universe@acer6:~/Desktop$ lex first.l
universe@acer6:~/Desktop$ yacc -d first.y
universe@acer6:~/Desktop$ cc y.tab.c lex.yy.c
```



```
universe@acer6:~/Desktop$ ./a.out
Enter the expression
2+3
Result = 5
universe@acer6:~/Desktop$ 
```

Conclusion:

To conclude, the experiment focused on studying the Lexical Analyzer tool Flex/Lex to recognize lexical patterns from a given input file. Lex program generator provides a high-level language to write string expressions for pattern matching, and the user has the freedom to write actions in a general-purpose programming language. Overall, the experiment helped understand the working of Lexical Analyzer and its usefulness in analyzing input streams.

Postlab:

1. **Write the structure of Lex**

2. **Write the structure of Yacc**