MAHARISHI DAYANAND UNIVERSITY

Delhi Global Institute of Technology

Compiler Design Lab

Submitted By: Bazgha Razi

Subject Code : LC-CSE-324G

Subject Name: Compiler Design Lab

Registration Number :*191380214*

INDEX

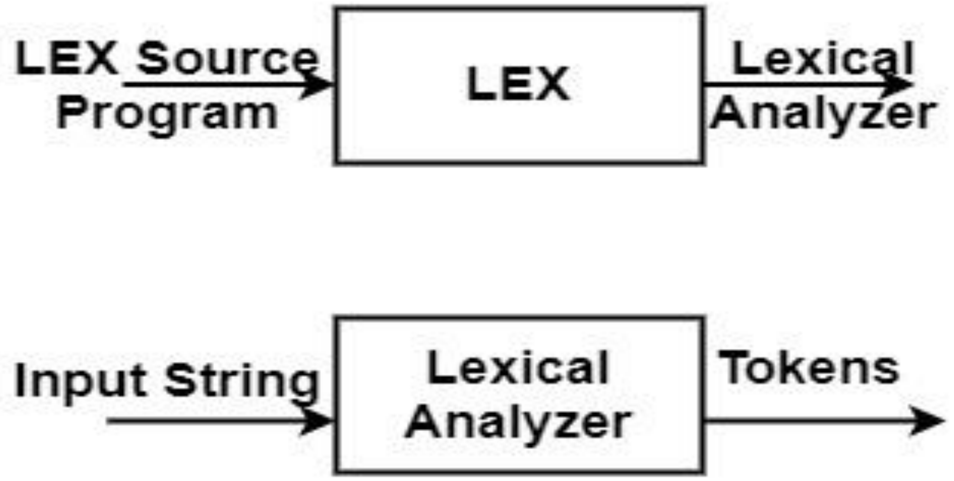
S.No.	Program	Date	Page No.	Remaks
1.	Introduction to LEX.	28/03/22	3-4	
2.	Write a program for token separation with a given expression.	04/04/22	5	
3.	Write a program for token separation with a given file.	04/04/22	6	
4.	Write a program for Lexical analysis using LEX tools.	02/05/22	7-9	
5.	Write a program to identify whether a given line is a comment or not.	02/05/22	10	
6.	Write a program to check whether a given identifier is valid or not.	09/05/22	11	
7.	Write a program to recognize strings under ‘a’, ‘a*b+’, ‘abb’.	30/05/22	12-15	
8.	Write a program to implement operator precedence parser.	06/06/22	16-17	
9.	Introduction to YACC	13/06/22	18-19	
10.	Write a program for implementation of calculator using YACC tool.	13/06/22	20-21	
11.	Write a program for lexical analyser for validating operators.	20/06/22	22-26	
12.	Write a program for implementation of LALR parser.	20/06/22	27	

Program 1: Introduction to Lex

It is a tool or software which automatically generates a lexical analyzer (finite Automata). It takes as its input a LEX source program and produces lexical Analyzer as its output. Lexical Analyzer will convert the input string entered by the user into tokens as its output.

LEX is a program generator designed for lexical processing of character input/output stream. Anything from simple text search program that looks for pattern in its input-output file to a C compiler that transforms a program into optimized code.

In program with structure input-output two tasks occurs over and over. It can divide the input-output into meaningful units and then discovering the relationships among the units for C program (the units are variable names, constants, and strings). This division into units (called tokens) is known as lexical analyzer or LEXING. LEX helps by taking a set of descriptions of possible tokens n producing a routine called a lexical analyzer or LEXER or Scanner.

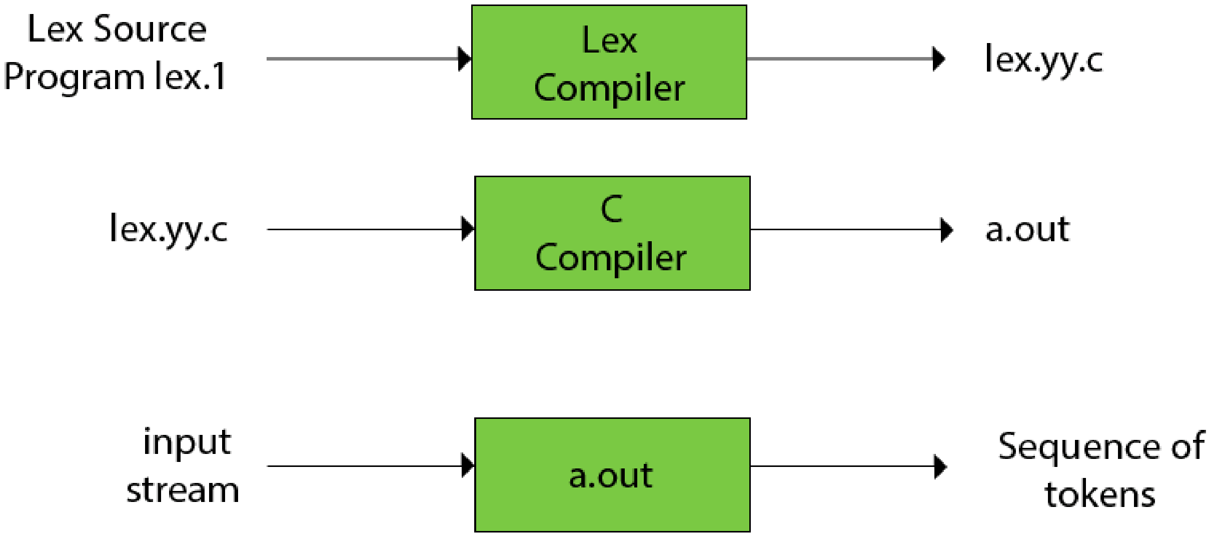


LEX Tool

The function of Lex is as follows:

- Firstly lexical analyzer creates a program lex.1 in the Lex language. Then Lex compiler runs the lex.1 program and produces a C program lex.yy.c.

- Finally C compiler runs the lex.yy.c program and produces an object program a.out.
- a.out is lexical analyzer that transforms an input stream into a sequence of tokens.



Lex file format

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

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

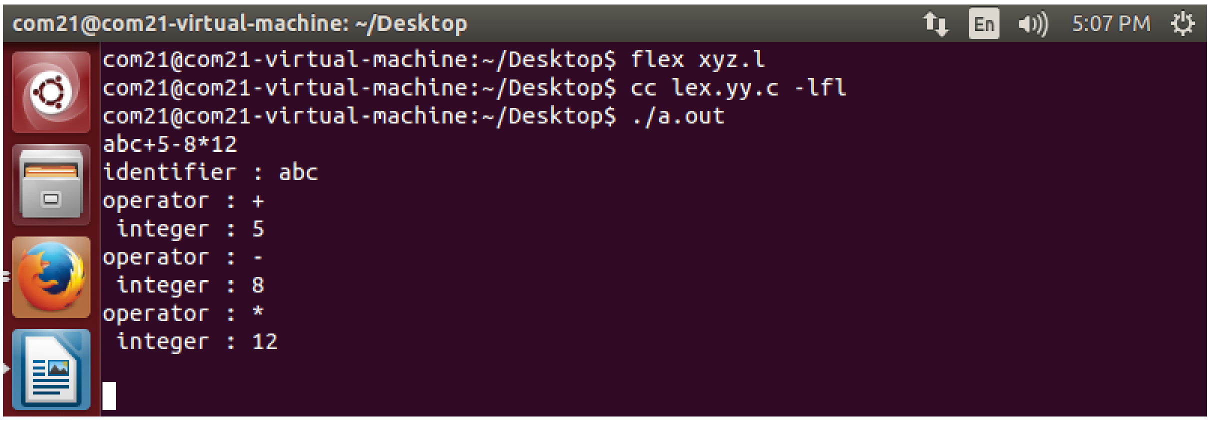
Definitions include declarations of constant, variable and regular definitions.
Rules define the statement of form p1 {action1} p2 {action2}....pn {action}.
Where **pi** describes the regular expression and **action1** describes the actions what action the lexical analyzer should take when pattern pi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

Program 2: Write a program for token separation with a given expression

Code:

```
%%

"while"|"if"|"else" {printf("keywords : %s\n", yytext);}

"int"|"float" {printf("keywords : %s\n", yytext);}

[a-zA-Z_][a-zA-Z0-9_]* {printf("identifier : %s\n", yytext);}

"<="|"=="|"="|"++"|"-"|"*"|"+" {printf("operator : %s\n", yytext);}

[(){}|, ;]    {printf(" separator : %s\n", yytext);}

[0-9]*"."[0-9]+ {printf("float : %s\n", yytext);}

[0-9]+ {printf(" integer : %s\n", yytext);}

%%
main(int argc, char** argv)

{
   yylex();
}
```

Output:



```
com21@com21-virtual-machine:~/Desktop$ flex xyz.l
com21@com21-virtual-machine:~/Desktop$ cc lex.yy.c -lfl
com21@com21-virtual-machine:~/Desktop$ ./a.out
abc+5-8*12
identifier : abc
operator : +
 integer : 5
operator : -
 integer : 8
operator : *
 integer : 12
```

Program 3: Write a program for token separation with a given file.

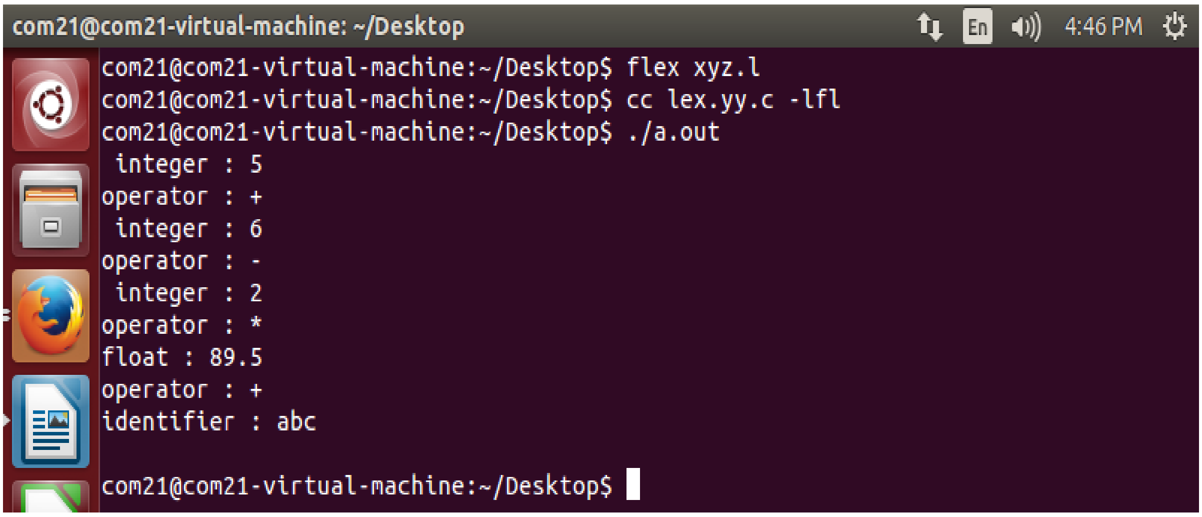
Code:

```
%%

"while"|"if"|"else" {printf("keywords : %s\n", yytext);}
"int"|"float" {printf("keywords : %s\n", yytext);}
[a-zA-Z_][a-zA-Z0-9_]* {printf("identifier : %s\n", yytext);}
"<="|"=="|"="|"++"|"-"|"*"|"+" {printf("operator : %s\n", yytext);}
[(){}|, ;]    {printf(" separator : %s\n", yytext);}
[0-9]*"."[0-9]+ {printf("float : %s\n", yytext);}
[0-9]+ {printf(" integer : %s\n", yytext);}
%%
int yywrap(){}
main(int argc, char** argv)
{
   extern FILE *yyin;
   yyin = fopen("input.txt","r");
   yylex();
}
```

Input:

```
5+6-2*89.5+abc
```

Output:



```
com21@com21-virtual-machine:~/Desktop$ flex xyz.l
com21@com21-virtual-machine:~/Desktop$ cc lex.yy.c -lfl
com21@com21-virtual-machine:~/Desktop$ ./a.out
 integer : 5
operator : +
 integer : 6
operator : -
 integer : 2
operator : *
float : 89.5
operator : +
identifier : abc

com21@com21-virtual-machine:~/Desktop$
```

Program 4: Write a program for lexical analysis using LEX tools.

Code:

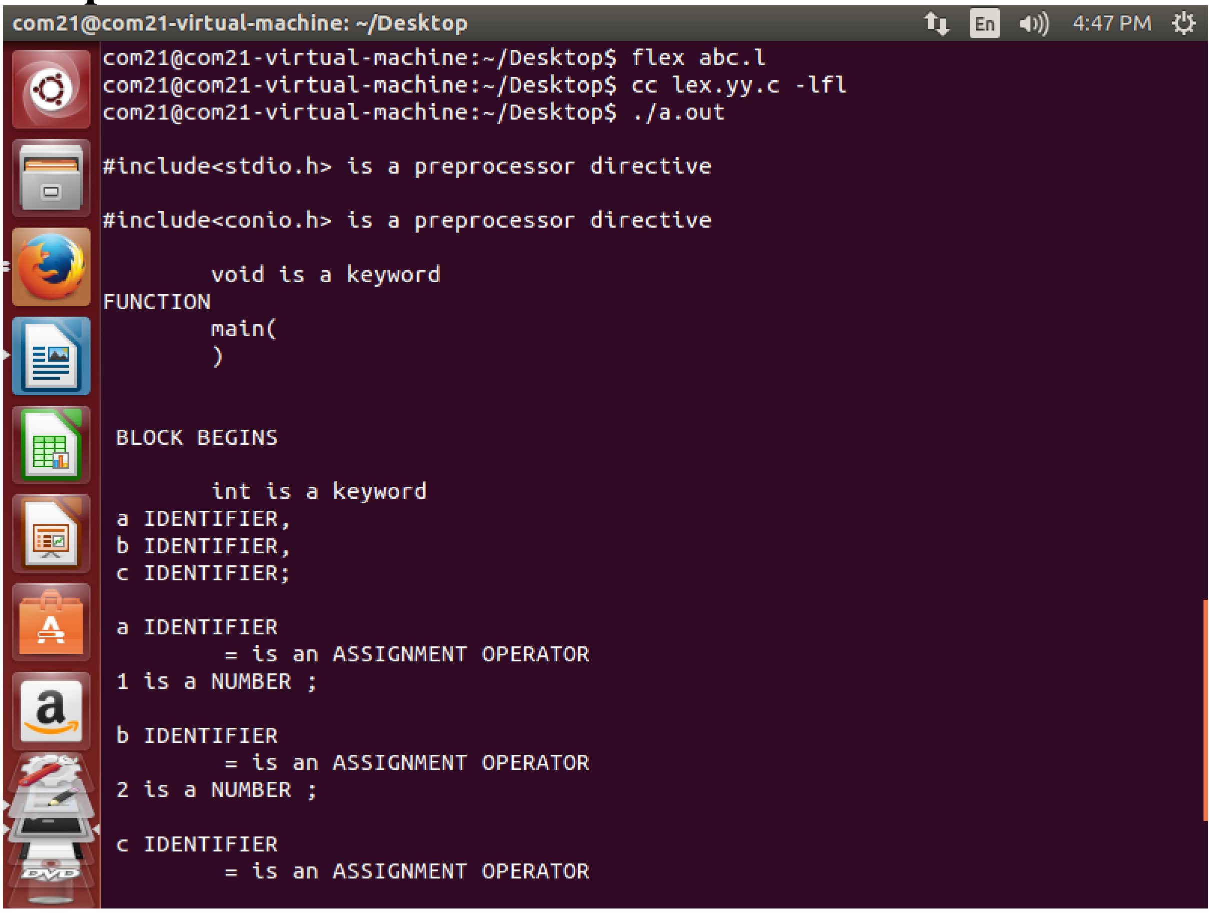
```
%{
int COMMENT=0;
%}
identifier [a-zA-Z][a-zA-Z0-9]*
%%
#.* {printf("\n%s is a preprocessor directive",yytext);}
int |
float |
char |
double |
while |
for |
struct |
typedef |
do |
if |
break |
continue |
void |
switch |
return |
else |
goto {printf("\n\t%s is a keyword",yytext);}
"/*" {COMMENT=1;}{printf("\n\t %s is a COMMENT",yytext);}
{identifier}\( {if(!COMMENT)printf("\nFUNCTION
\n\t%s",yytext);}
\{  {if(!COMMENT)printf("\n BLOCK BEGINS");}
\}  {if(!COMMENT)printf("BLOCK ENDS ");}
{identifier}(\[[0-9]*\])? {if(!COMMENT) printf("\n %s
IDENTIFIER",yytext);}
\".*\" {if(!COMMENT)printf("\n\t %s is a STRING",yytext);}
[0-9]+ {if(!COMMENT) printf("\n %s is a NUMBER ",yytext);}
\)(\:)? {if(!COMMENT)printf("\n\t");ECHO;printf("\n");}
```

```
\( ECHO;
= {if(!COMMENT)printf("\n\t %s is an ASSIGNMENT
OPERATOR",yytext);}
\<= |
\>= |
\< |
== |
\> {if(!COMMENT) printf("\n\t%s is a RELATIONAL
OPERATOR",yytext);}
%%
int main(int argc, char **argv)
{
FILE *file;
file=fopen("var.c","r");
if(!file)
{
printf("could not open the file");
exit(0);
}
yyin=file;
yylex();
printf("\n");
return(0);
}
int yywrap()
{
return(1);
}
```

Input:

```
#include<stdio.h>
#include<conio.h>
void main()
{
int a,b,c;
a=1;
b=2;
c=a+b;
printf("Sum:%d",c);
}
```

Output:



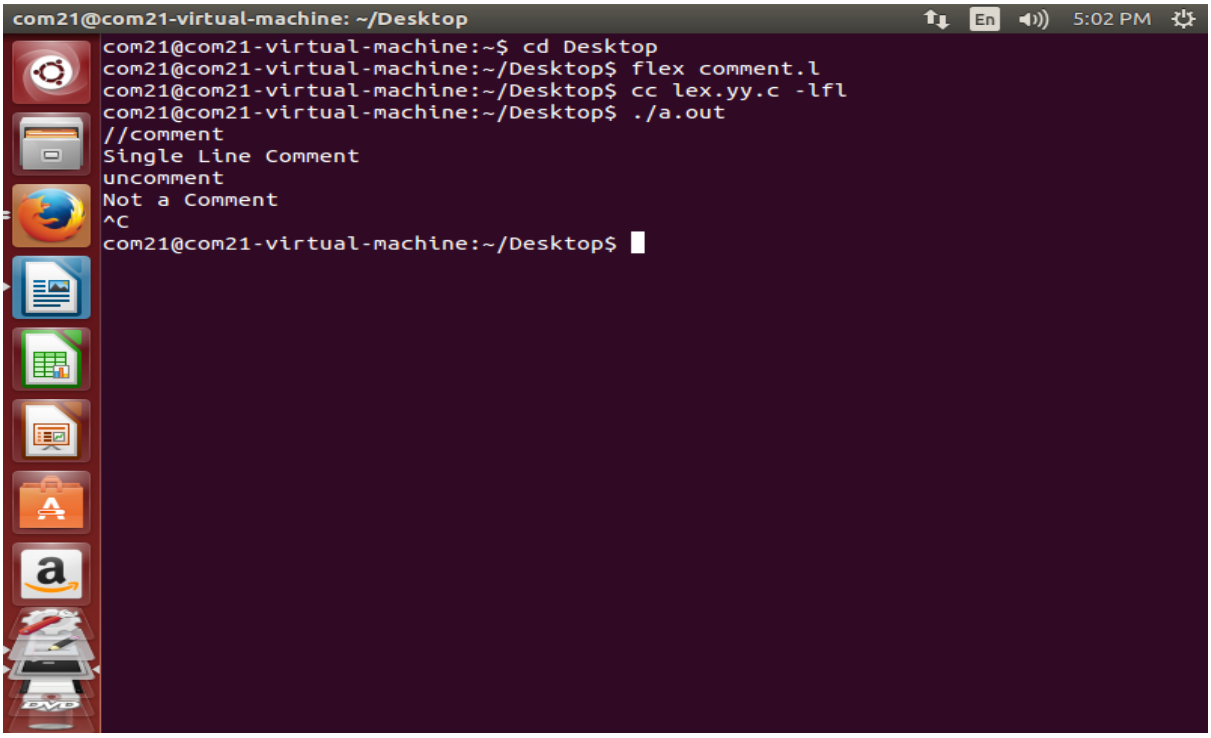
```
com21@com21-virtual-machine:~/Desktop$ flex abc.l
com21@com21-virtual-machine:~/Desktop$ cc lex.yy.c -lfl
com21@com21-virtual-machine:~/Desktop$ ./a.out

#include<stdio.h> is a preprocessor directive

#include<conio.h> is a preprocessor directive

        void is a keyword
FUNCTION
        main(
        )


 BLOCK BEGINS

        int is a keyword
 a IDENTIFIER,
 b IDENTIFIER,
 c IDENTIFIER;

 a IDENTIFIER
         = is an ASSIGNMENT OPERATOR
 1 is a NUMBER ;

 b IDENTIFIER
         = is an ASSIGNMENT OPERATOR
 2 is a NUMBER ;

 c IDENTIFIER
         = is an ASSIGNMENT OPERATOR
```

Program 5: Write a program to identify whether a given line is comment or not.

Code:

```
%{
#include<stdio.h>
%}
%%
[/]{1}[/]{1}[" "a-zA-Z0-9]* {printf("Single Line Comment");}
[/]{1}[*]{1}[" "a-zA-Z0-9]*[*]{1}[/]{1} {printf("MultiLine
Comment");}
.* {printf("Not a Comment");}
%%
main(){yylex();}
```

Output:



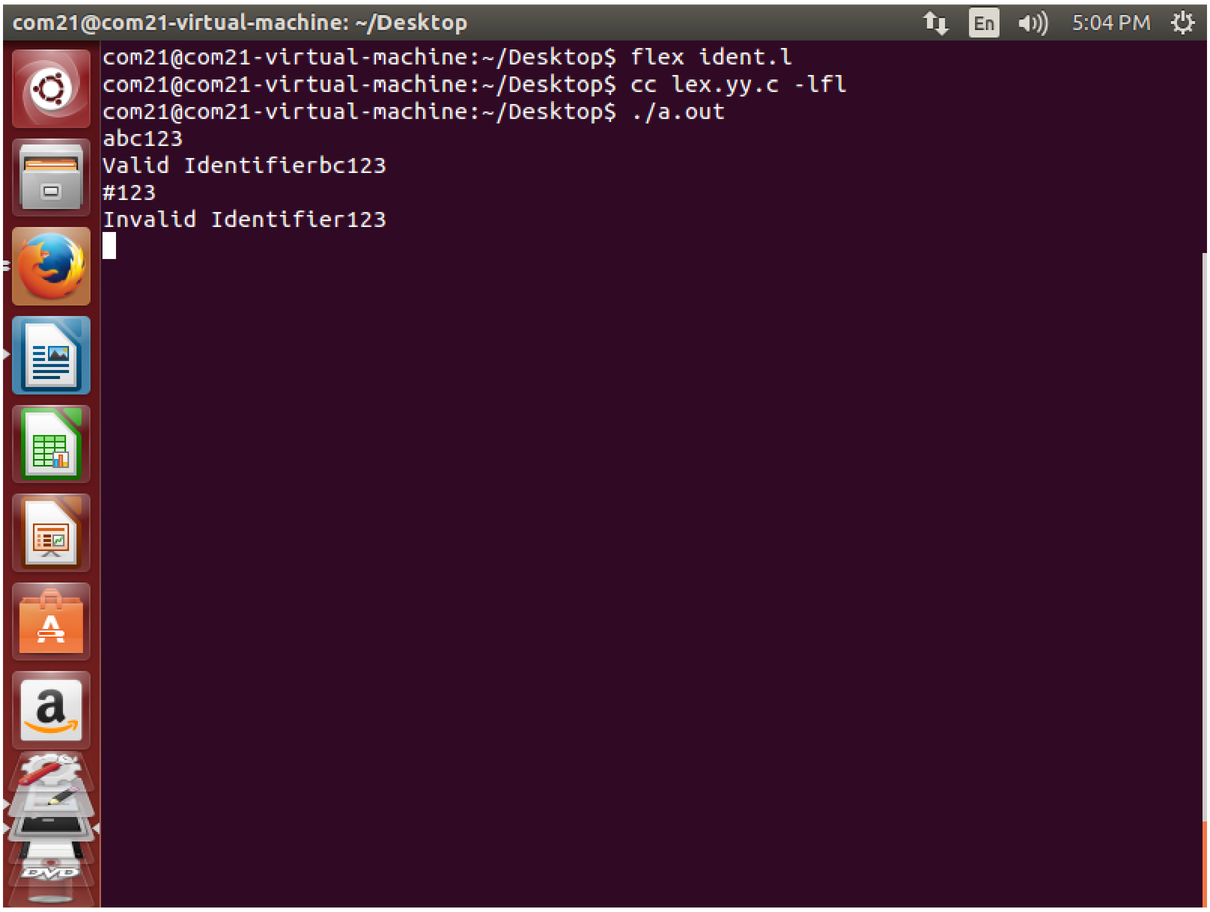
```
com21@com21-virtual-machine:~$ cd Desktop
com21@com21-virtual-machine:~/Desktop$ flex comment.l
com21@com21-virtual-machine:~/Desktop$ cc lex.yy.c -lfl
com21@com21-virtual-machine:~/Desktop$ ./a.out
//comment
Single Line Comment
uncomment
Not a Comment
^C
com21@com21-virtual-machine:~/Desktop$ 
```

Program 6: Write a program to check whether given identifier is valid or not.

Code:

```
%%
^[a - z A - Z _][a - z A - Z 0 - 9 _]* {printf("Valid Identifier");}

^[^a - z A - Z _] {printf("Invalid Identifier");}
%%

   main()
{
   yylex();
}
```

Output:



```
com21@com21-virtual-machine:~/Desktop$ flex ident.l
com21@com21-virtual-machine:~/Desktop$ cc lex.yy.c -lfl
com21@com21-virtual-machine:~/Desktop$ ./a.out
abc123
Valid Identifierbc123
#123
Invalid Identifier123
```

Program 7: Write a program to recognize strings under ‘a’,’a*b+’, ‘abb’.

Code:

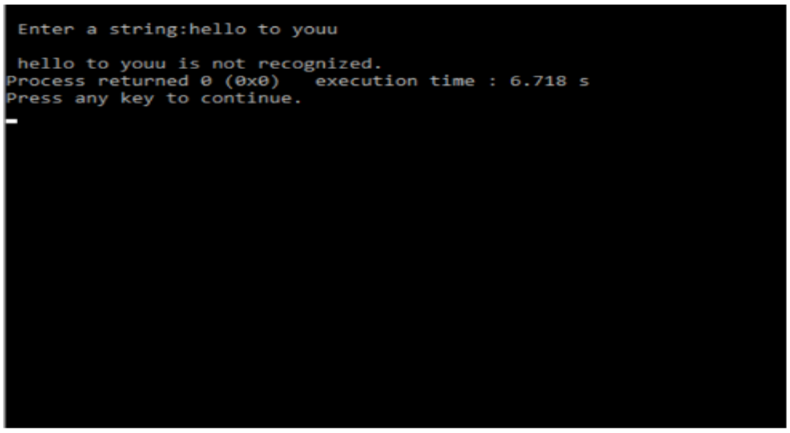
```
#include<stdio.h>

#include<string.h>

#include<stdlib.h>

int main()

{

char s[20],c;

int state=0,i=0;

printf("\n Enter a string:");

gets(s);

while(s[i]!='\0')

{

switch(state)

{

case 0:

c=s[i++];

if(c=='a')

state=1;

else if(c=='b')

state=2;
```

```
else
state=6;
break;
case 1:
c=s[i++];
if(c=='a')
state=3;
else if(c=='b')
state=4;
else
state=6;
break;
case 2:
c=s[i++];
if(c=='a')
state=6;
else if(c=='b')
state=2;
else
state=6;
break;
case 3:
```

```
c=s[i++];
if(c=='a')
state=3;
else if(c=='b')
state=2;
else
state=6;
break;
case 4:
c=s[i++];
if(c=='a')
state=6;
else if(c=='b')
state=5;
else
state=6;
break;
case 5:
c=s[i++];
if(c=='a')
state=6;
else if(c=='b')
```

```
state=2;
else
state=6;
break;
case 6:
printf("\n %s is not recognized.",s);
exit(0); } }
if(state==1)
printf("\n %s is accepted under rule 'a'",s);
else if((state==2)||(state==4))
printf("\n %s is accepted under rule 'a*b+'",s);
else if(state==5)
printf("\n %s is accepted under rule 'abb'",s);
return 0;
}
```

Output:



```
 Enter a string:hello to youu

 hello to youu is not recognized.
Process returned 0 (0x0)   execution time : 6.718 s
Press any key to continue.
```

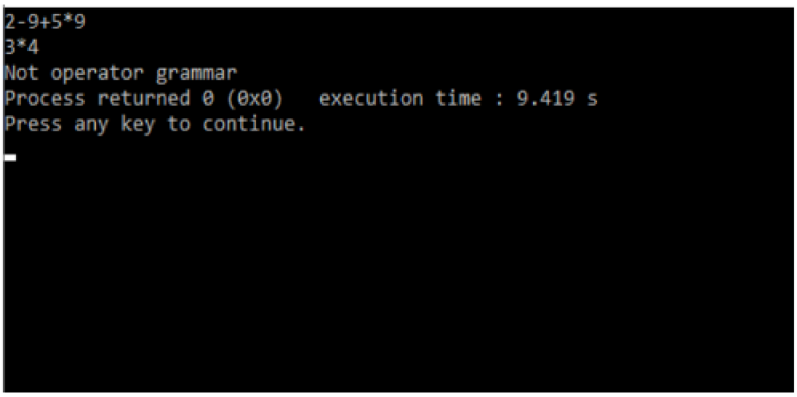
Program 8: Write a program for implementation of Operator Precedence Parser.

Code:

```
#include<stdlib.h>

#include<stdio.h>

#include<string.h>

not true void f()

{

printf("Not operator

grammar"); exit(0);

}

int main()

{

char grm[20][20], c;

grammar int i, n, j = 2, flag = 0;

from user

scanf("%d", &n);

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

scanf("%s", grm[i]);

for (i = 0; i < n; i++) {

c = grm[i][2];

while (c != '\0') {
```

```
if (grm[i][3] == '+' || grm[i][3] == '-'|| grm[i][3] == '*' || grm[i][3] == '/')

flag = 1;

else {

flag = 0;

f();

}

if (c == '$') {

flag = 0;

f();

}

c = grm[i][++j];

}

if (flag == 1)

printf("Operator grammar");

return 0;

}
```

Output:



```
2-9+5*9
3*4
Not operator grammar
Process returned 0 (0x0)   execution time : 9.419 s
Press any key to continue.
```

Program 9: Introduction to YACC

YACC is known as Yet Another Compiler Compiler. It is used to produce the source code of the syntactic analyzer of the language produced by LALR (1) grammar. The input of YACC is the rule or grammar, and the output is a C program. Stephen C. Johnson creates the first kind of YACC. If we have a file translate.y that consists of YACC specification, then the UNIX system command is:

YACC translate.y

This command converts the file translate.y into a C file y.tab.c. It represents an LALR parser prepared in C with some other user's prepared C routines. By compiling y.tab.c along with the ly library, we will get the desired object program a.out that performs the operation defined by the original YACC program.

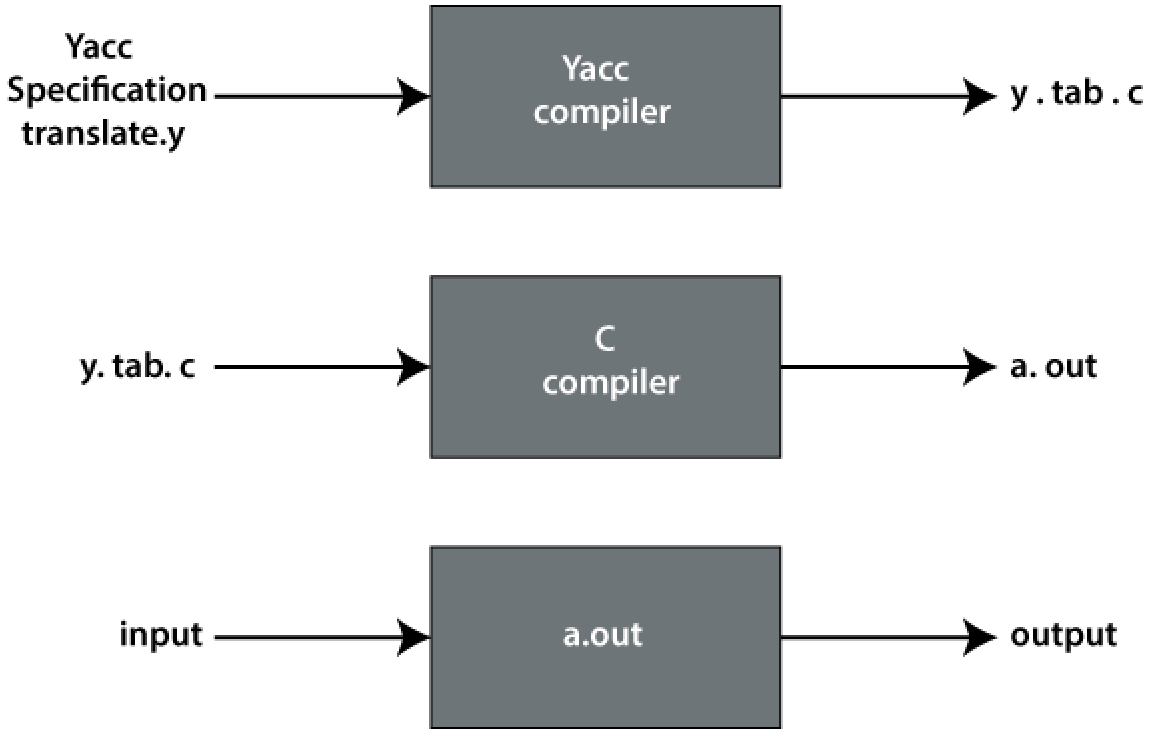
Input: A CFG- file.y

Output: A parser y.tab.c (yacc)

- The output file "file.output" contains the parsing tables.
- The file "file.tab.h" contains declarations.
- The parser called the yyparse ().
- Parser expects to use a function called yylex () to get tokens.

Yacc provides a general tool for imposing structure on the input to a computer program. The Yacc user prepares a specification of the input process; this includes rules describing the input structure, code to be invoked when these rules are recognized, and a low-level routine to do the basic input. Yacc then generates a function to control the input process. This function, called a parser, calls the user-supplied low-level input routine (the lexical analyzer) to pick up the basic items (called tokens) from the input stream. These tokens are organized according to the input structure rules, called grammar rules; when one of these rules has been recognized, then user code supplied for this rule, an action, is invoked; actions have the ability to return values and make use of the values of other actions.

The construction of translation using YACC is illustrated in the figure below:



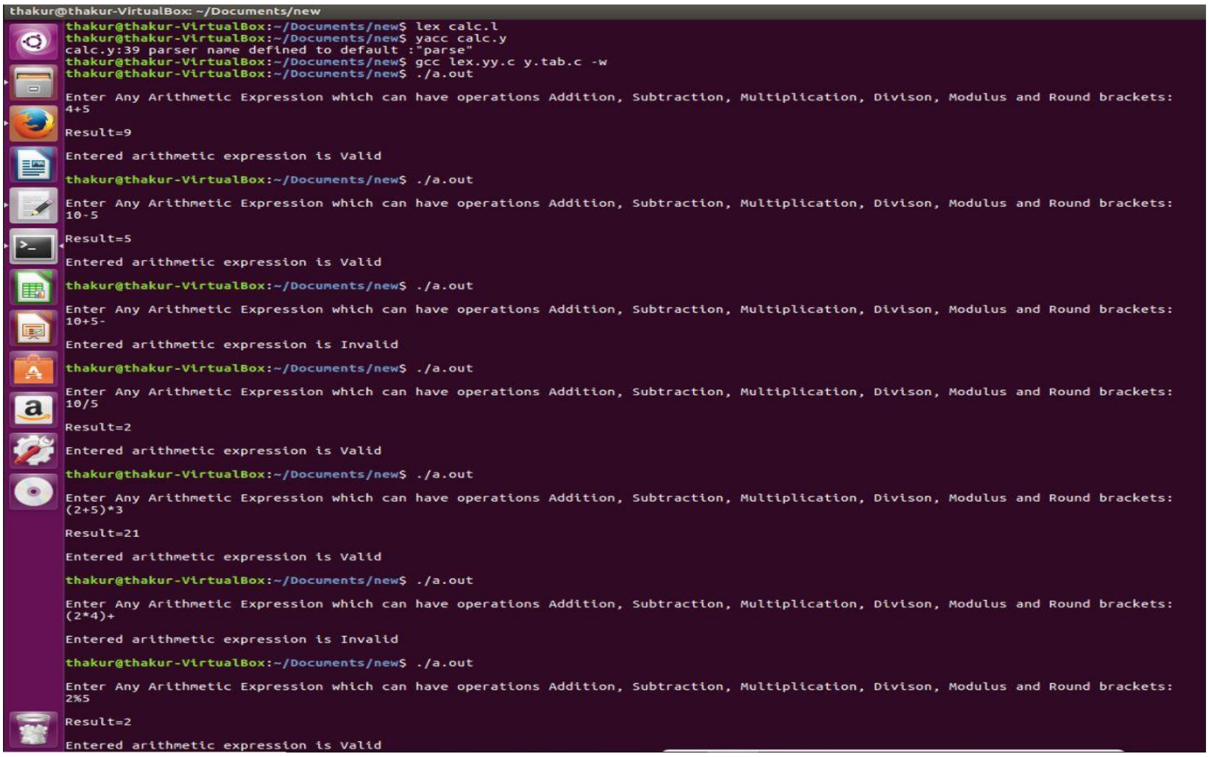
Program 10: Write a program for implementation of calculator using YACC tool.

Code:

```
%{
 #include<stdio.h>
 int flag=0;
%}
%token NUMBER
%left '+' '-'
%left '*' '/' '%'
%left '(' ')'
%%
ArithmeticExpression: E{ printf("\nResult=%d\n", $$);
         return 0;
        };
 E:E'+'E {$$=$1+$3;}
 |E'-'E {$$=$1-$3;}
 |E'*'E {$$=$1*$3;}
 |E'/'E {$$=$1/$3;}
 |E'%'E {$$=$1%$3;}
 |'('E')' {$$=$2;}
 | NUMBER {$$=$1;};
```

```
%%
void main()
{
   printf("\nEnter Any Arithmetic Expression which can have operations Addition, Subtraction, Multiplication, Division, Modulus and Round brackets:\n");
   yyparse();
   if(flag==0)
   printf("\nEntered arithmetic expression is Valid\n\n");
}
void yyerror()
{ printf("\nEntered arithmetic expression is Invalid\n\n");
   flag=1; }
```

Output:



```
thakur@thakur-VirtualBox:~/Documents/new$ lex calc.l
thakur@thakur-VirtualBox:~/Documents/new$ yacc calc.y
calc.y:39 parser name defined to default :"parse"
thakur@thakur-VirtualBox:~/Documents/new$ gcc lex.yy.c y.tab.c -w
thakur@thakur-VirtualBox:~/Documents/new$ ./a.out

Enter Any Arithmetic Expression which can have operations Addition, Subtraction, Multiplication, Divison, Modulus and Round brackets:
4+5

Result=9

Entered arithmetic expression is Valid

thakur@thakur-VirtualBox:~/Documents/new$ ./a.out

Enter Any Arithmetic Expression which can have operations Addition, Subtraction, Multiplication, Divison, Modulus and Round brackets:
10-5

Result=5

Entered arithmetic expression is Valid

thakur@thakur-VirtualBox:~/Documents/new$ ./a.out

Enter Any Arithmetic Expression which can have operations Addition, Subtraction, Multiplication, Divison, Modulus and Round brackets:
10+5-

Entered arithmetic expression is Invalid

thakur@thakur-VirtualBox:~/Documents/new$ ./a.out

Enter Any Arithmetic Expression which can have operations Addition, Subtraction, Multiplication, Divison, Modulus and Round brackets:
10/5

Result=2

Entered arithmetic expression is Valid

thakur@thakur-VirtualBox:~/Documents/new$ ./a.out

Enter Any Arithmetic Expression which can have operations Addition, Subtraction, Multiplication, Divison, Modulus and Round brackets:
(2+5)*3

Result=21

Entered arithmetic expression is Valid

thakur@thakur-VirtualBox:~/Documents/new$ ./a.out

Enter Any Arithmetic Expression which can have operations Addition, Subtraction, Multiplication, Divison, Modulus and Round brackets:
(2*4)+

Entered arithmetic expression is Invalid

thakur@thakur-VirtualBox:~/Documents/new$ ./a.out

Enter Any Arithmetic Expression which can have operations Addition, Subtraction, Multiplication, Divison, Modulus and Round brackets:
2%5

Result=2

Entered arithmetic expression is Valid
```

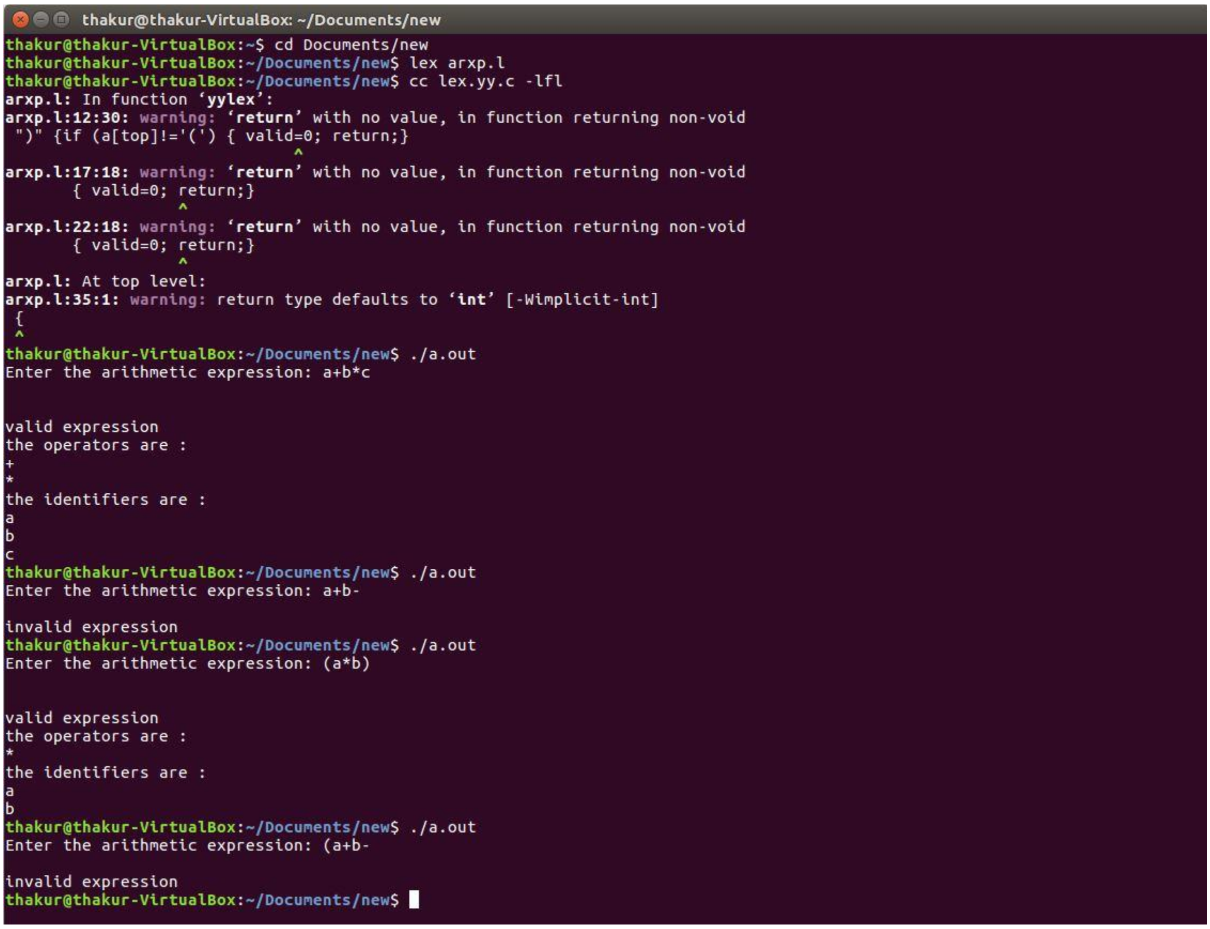
Program 11: Write a program to simulate lexical analyser for validating operators.

Code:

```
%{
#include <stdio.h>
#include <string.h>
    int operators_count = 0, operands_count = 0, valid = 1, top = -1, l = 0, j = 0;
    char operands[10][10], operators[10][10], stack[100];
%}
%%
"(" {
    top++;
    stack[top] = '(';
}
"{" {
    top++;
    stack[top] = '{';
}
"[" {
    top++;
    stack[top] = '[';
}
```

```
")" {
    if (stack[top] != '(') {
        valid = 0;
    }
    else if(operands_count>0 && (operands_count-
operators_count)!=1){
        valid=0;
    }
    else{
        top--;
        operands_count=1;
        operators_count=0;
    }
}
"}" {
    if (stack[top] != '{') {
        valid = 0;
    }
    else if(operands_count>0 && (operands_count-
operators_count)!=1){
        valid=0;
    }
    else{
        top--;
        operands_count=1;
        operators_count=0;
    }
}
"]" {
    if (stack[top] != '[') {
        valid = 0;
    }
    else if(operands_count>0 && (operands_count-operators_count)!=1){
        valid=0;
    }
    else{
        top--;
        operands_count=1;
        operators_count=0;
    }

}
"+"|"-"|"*"|"/" {
    operators_count++;
    strcpy(operators[l], yytext);
    l++;
}
[0-9]+|[a-zA-Z][a-zA-Z0-9_]* {
    operands_count++;
    strcpy(operands[j], yytext);
    j++;
}
%%
int yywrap()
{
    return 1;
}
int main()
{
    int k;
    printf("Enter the arithmetic expression: ");
    yylex();
    if (valid == 1 && top == -1) {
        printf("\nValid Expression\n");
    }
    else
        printf("\nInvalid Expression\n");
    return 0;

}
```

Output:



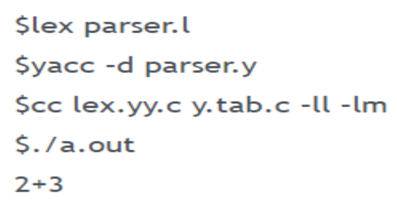
```
thakur@thakur-VirtualBox:~$ cd Documents/new
thakur@thakur-VirtualBox:~/Documents/new$ lex arxp.l
thakur@thakur-VirtualBox:~/Documents/new$ cc lex.yy.c -lfl
arxp.l: In function ‘yylex’:
arxp.l:12:30: warning: ‘return’ with no value, in function returning non-void
 ")" {if (a[top]!='(') { valid=0; return;}
                              ^
arxp.l:17:18: warning: ‘return’ with no value, in function returning non-void
       { valid=0; return;}
                  ^
arxp.l:22:18: warning: ‘return’ with no value, in function returning non-void
       { valid=0; return;}
                  ^
arxp.l: At top level:
arxp.l:35:1: warning: return type defaults to ‘int’ [-Wimplicit-int]
 {
 ^
thakur@thakur-VirtualBox:~/Documents/new$ ./a.out
Enter the arithmetic expression: a+b*c


valid expression
the operators are :
+
*
the identifiers are :
a
b
c
thakur@thakur-VirtualBox:~/Documents/new$ ./a.out
Enter the arithmetic expression: a+b-

invalid expression
thakur@thakur-VirtualBox:~/Documents/new$ ./a.out
Enter the arithmetic expression: (a*b)


valid expression
the operators are :
*
the identifiers are :
a
b
thakur@thakur-VirtualBox:~/Documents/new$ ./a.out
Enter the arithmetic expression: (a+b-

invalid expression
thakur@thakur-VirtualBox:~/Documents/new$ 
```

Program 12: Write a program for implementation of LALR Parser.

Code:

```
<parser.y>
%{
#include<stdio.h>
%}
%union
{ double dval; }
%token <dval> DIGIT
%type <dval> expr
%type <dval> term
%type <dval> factor
%%
line: expr '\n' {
printf("%g\n",$1); };
expr: expr '+' term {$$=$1 + $3 ;}
| term;
term: term '*' factor {$$=$1 * $3 ;}
| factor;
factor: '(' expr ')' {$$=$2 ;}
| DIGIT;
%%
int main()
{ yyparse();
}
yyerror(char *s)
{ printf("%s",s);
}
```

Output:



```
$lex parser.l
$yacc -d parser.y
$cc lex.yy.c y.tab.c -ll -lm
$./a.out
2+3
```