Practical 1: To write the Lex code to count number of lines words characters tabs and space.

```
%{
# include <stdio.h>
int newlines = 0;
int spaces = 0;
int charR = 0;
%}
%%
[\n] {newlines++;}
[ ]  {spaces++;}
. {charR++;}
%%
int yywrap(void){}
int main(){
   yylex();
   printf("Number of newlines : %d\n Number of spaces : %d\n Number of characters : %d\n",newlines,spaces,charR);
}
```

OUTPUT

```
root@kali:~/Desktop/compilerlab_work# lex lab1.l
root@kali:~/Desktop/compilerlab_work# gcc lex.yy.c
root@kali:~/Desktop/compilerlab_work# ./a.out
Hello my name is KJ
How are you?
NewLines : 2
 Tabs : 0
 Spaces : 7
  Rest_Characters  : 25
root@kali:~/Desktop/compilerlab_work#
```

Practical 2: Designer Lex code to identify and print valid identifiers of C and C in a given input pattern.

```
%{
#include <stdio.h>
%}
%%
[a-z|A-Z|_][a-z|A-Z|_|0-9]* {printf("Valid Identifier!");}
.* {printf("Invalid Identifier!");}
%%
int yywrap(void){}
int main(){
   printf("Enter the Identifier : \n");
   yylex();

}
```

OUTPUT

```
root@kali:~/Desktop/compilerlab_work# lex lab2.l
root@kali:~/Desktop/compilerlab_work# gcc lex.yy.c
root@kali:~/Desktop/compilerlab_work# ./a.out
Enter The Identifiers:

hello
Valid Identifier
Luffy
Invalid Identifier
_bias_lisa
Valid Identifier
129ia
Invalid Identifier
```

Practical 3: To design a lex code to identify and print integer and float value in given input pattern.

```
%{
#include <stdio.h>
%}
%%
[0-9]+ {printf("Integer");}
[0-9]+[.][0-9]+ {printf("Float");}
.* {printf("Invalid Number");}
%%
int yywrap(void){}
int main(){
printf("Enter a number : \n");
yylex();
}
```

OUTPUT

```
root@kali:~/Desktop/compilerlab_work# lex lab3.l
root@kali:~/Desktop/compilerlab_work# gcc lex.yy.c
root@kali:~/Desktop/compilerlab_work# ./a.out
Enter the Numbers :
123
Integer
123.12
Float
131212
Integer
196
Integer
```

Practical 4: To design a leg sport for tokenizing (identify and print operators separators keywords and identifier) the following C format.

```
%{
#include<stdio.h>
int flag = 0;
%}
%%
auto|int|float|union|double|static|global  {printf("keyword");}
[{};,()] {printf("seperator");}
[+%/*-=] {printf("operator");}
[a-z||A-z][a-z||A-z|0-9]* {printf("identifier");}
%%
int yywrap(){
return 1;
}
```

int main()

{

yylex();

return 0;

}

OUTPUT

```
root@kali:~/Desktop/compilerlab_work# lex lab4.l
root@kali:~/Desktop/compilerlab_work# gcc lex.yy.c
root@kali:~/Desktop/compilerlab_work# ./a.out
Enter the C code :
int p=1,d=0,r=4;
    float m=0.0, n=200.0;
    while (p <= 3)
    {
            if(d==0)
                    { m= m+n*r+4.5; d++; }
            else
                    { r++; m=m+r+1000.0; }
            p++;
    }    Identifier
keyword  Operator       Seperator
keyword  Operator       Seperator
keyword  Operator       Seperator       Identifier
keyword  Operator       Operator        Seperator
keyword  Operator       Operator        Seperator       Identifier      Seperator
keyword  Operator       Seperator       Seperator       Identifier      Seperator
keyword  Operator       Operator        Seperator       Seperator
keyword  Operator
keyword  Operator
keyword  Operator
keyword  Operator       Operator        Seperator
keyword  Operator       Operator        Seperator       Seperator       Identifier      Seperator
keyword  Operator       Operator        Seperator
keyword  Operator
keyword  Operator
keyword  Operator       Operator        Seperator       Seperator
keyword  Operator       Operator        Seperator
```

Practical 5: To design a lex code to count and print the number of total characters words while spaces and give input.text file

%{

#include<stdio.h>

int ch = 0;

int words = 0;

int spaces = 0;

%}

```
%%

[' '] {spaces++;}

. {ch++;}

%%

int yywrap()

{}

int main()

{

extern FILE *yyin;

yyin = fopen("input.txt","r");

yylex();

printf("%d %d %d",spaces,ch,spaces+1);

}
```

OUTPUT

```
root@kali:~/Desktop/compilerlab_work# lex lab5.l
root@kali:~/Desktop/compilerlab_work# gcc lex.yy.c
root@kali:~/Desktop/compilerlab_work# ./a.out
NewLines : 4
 Tabs : 0
 Spaces : 16
  Rest_Characters  : 70
root@kali:~/Desktop/compilerlab_work#
```

Practical 6: To design a lex code to replace while spaces of input.text file by a single blank character into output.text file.

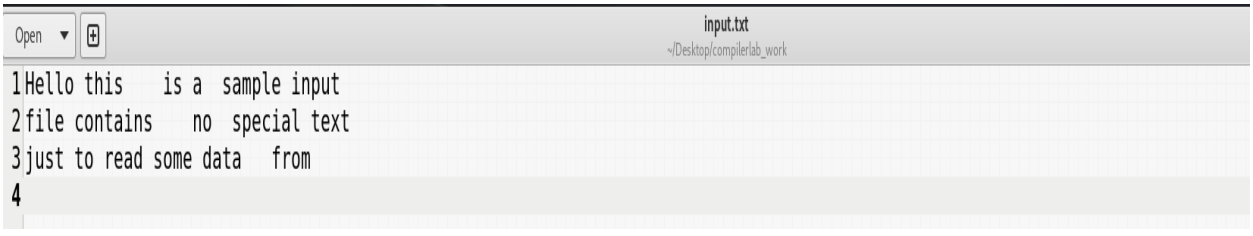
```
%{

#include<stdio.h>

%}

%%
```

```
[\t""]  {fprintf(yyout,"");}
.|\n {fprintf(yyout,"%s",yytext);}
%%
int yywrap()
{}
int main()
{
extern FILE *yyin,*yyout;
yyin = fopen("input.txt","r");
yyout = fopen("output.txt","w");
yylex();
}
```

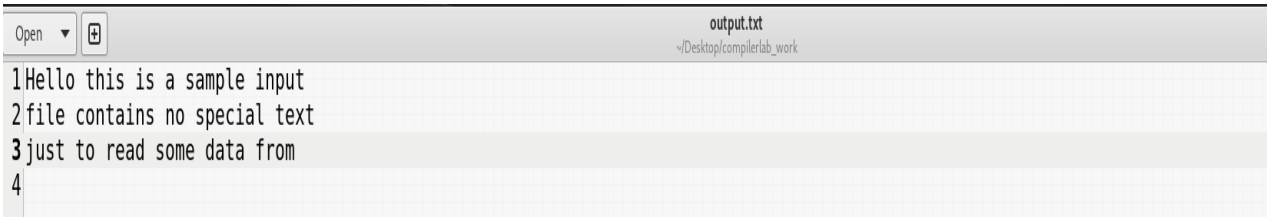
OUTPUT

```
root@kali:~/Desktop/compilerlab_work# lex lab6.l
root@kali:~/Desktop/compilerlab_work# gcc lex.yy.c
root@kali:~/Desktop/compilerlab_work# ./a.out
root@kali:~/Desktop/compilerlab_work#
```

Input



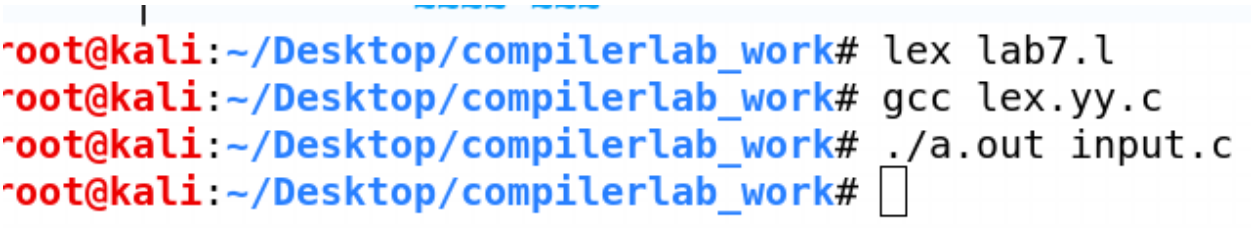
Output



Practical 7: To design a lex code to remove all the comments from any C program given at runtime and store into out.C file.

```
%{
#include<stdio.h>
%}
%%
\/\/(.*) {};
\/\(.\n).\*\/ {};
%%
int yywrap(){}
int main(int k,char **args)
{
extern FILE *yyin,*yyout;
yyin = fopen("input.c","r");
yyout = fopen("output.c","w");
yylex();
}
```

OUTPUT



Input.c

```c
//Start of the code
#include <stdio.h>

//main code
int main(){
 /* multiline comment
 comment
 comment
 */
 return 0;
}
```

Output.c

```c

#include <stdio.h>


int main(){

 return 0;
}
```

Practical 8: To design a lex code to extract all html tags in the given html file at run time and stored into the text file given at run time.

```
%{

#include<stdio.h>

%}
```

```
%%

"<"[^>]*> fprintf(yyout,"%s\n",yytext);

.|\n;

%%

int yywrap(){}

int main(int k, char **args)

{

extern FILE *yyin,*yyout;

yyin = fopen(args[1],"r");

yyout = fopen("output.html","w");

yylex();

}
```

OUTPUT

```
root@kali:~/Desktop/compilerlab_work# lex lab8.l
root@kali:~/Desktop/compilerlab_work# gcc lex.yy.c
root@kali:~/Desktop/compilerlab_work# ./a.out input.html
root@kali:~/Desktop/compilerlab_work#
```

Input.html

```
<!DOCTYPE html>
<html>
<head>
<title>Documnet</title>
</head>
<body>
<h1>My First Heading</h1>
<p>My first paragraph.</p>
</body>
</html>
```

Output.txt

```
1 <!DOCTYPE html>  <html>  <head>  <title>  </title>  </head>  <body>
  <h1>  </h1>  <p>  </p>  </body>  </html>
```

Practical 9: Designer lex code to represent dfa which accepts string containing even numbers of a and even numbers of b over input alphabet a and b.

```
%{
#include<stdio.h>
%}
%s A B C DEAD
%%
<INITIAL>a BEGIN A;
<INITIAL>b BEGIN B;
<INITIAL>[^ab\n] BEGIN DEAD;
<INITIAL>\n BEGIN INITIAL;{printf("accepted\n");}
<A>a BEGIN INITIAL;
<A>b BEGIN C;
<A>[^ab\n] BEGIN DEAD;
<A>\n BEGIN INITIAL;{printf("not accepted\n");}
<B>b BEGIN INITIAL;
<B>a BEGIN C;
<B>[^ab\n] BEGIN DEAD;
<B>\n BEGIN INITIAL;{printf("not accepted\n");}
<C>a BEGIN B;
<C>b BEGIN A;
```

```
<C>[^ab\n] BEGIN DEAD;
<C>\n BEGIN INITIAL;{printf("not accepted\n");}
<DEAD>[^ab\n] BEGIN DEAD;
<DEAD>\n BEGIN INITIAL;{printf("INVALID\n");}
%%
int yywrap()
{
return -1;
}
int main()
{
yylex();
}
```

Practical 10: To design a dfa in lex code which accept string containing 3rd list element a over input alphabet a,b.

```
%{
#include<stdio.h>
%}
%s A B C D E F G DEAD
%%
<INITIAL>b BEGIN INITIAL;
<INITIAL>a BEGIN A;
<INITIAL>[^ab\n] BEGIN DEAD;
<INITIAL>\n BEGIN INITIAL; {printf("NOT accepted\n");}
<A>a BEGIN B;
<A>b BEGIN F;
<A>[^ab\n] BEGIN DEAD;
```

```
<A>\n BEGIN INITIAL; {printf("NOT accepted\n");}
<B>a BEGIN C;
<B>b BEGIN D;
<B>[^ab\n] BEGIN DEAD;
<B>\n BEGIN INITIAL; {printf("NOT accepted\n");}
<C>a BEGIN C;
<C>b BEGIN D;
<C>[^ab\n] BEGIN DEAD;
<C>\n BEGIN INITIAL; {printf("accepted\n");}
<D>a BEGIN E;
<D>b BEGIN G;
<D>[^ab\n] BEGIN DEAD;
<D>\n BEGIN INITIAL; {printf("accepted\n");}
<E>a BEGIN B;
<E>b BEGIN F;
<E>[^ab\n] BEGIN DEAD;
<E>\n BEGIN INITIAL; {printf("accepted\n");}
<F>a BEGIN E;
<F>b BEGIN G;
<F>[^ab\n] BEGIN DEAD;
<F>\n BEGIN INITIAL; {printf("NOT accepted\n");}
<G>a BEGIN A;
<G>b BEGIN INITIAL;
<G>[^ab\n] BEGIN DEAD;
<G>\n BEGIN INITIAL; {printf("accepted\n");}
<DEAD>[^ab\n] BEGIN DEAD;
<DEAD>\n BEGIN INITIAL; {printf("invalid\n");}
```

```
%%
int yywrap()
{
}
int main()
{
yylex();
}
```

Practical 11: To design a DFA in lex code to identify which string containing ‘1’ as the last in element over input 0 and 1.

```
%{
#include<stdio.h>
%}
%s A
%%
<INITIAL>0 BEGIN INITIAL;
<INITIAL>1 BEGIN A;
<A>0 BEGIN INITIAL;
<A>1 BEGIN A;
<INITIAL>\n {printf("not valid\n");}
<A>\n {printf("valid\n");}
%%
int yywrap(){}
int main()
{
yylex();
}
```

Practical 12: Designer YACC /lex code to recognize arithmetic expression involving operators +, -, * and / without operator precedence grammar.

```
%{
#include"y.tab.h"
%}
%%
[a-zA-Z\_][a-zA-Z\_0-9]* return NAME;
[0-9]+ return NUMBER;
"+" return PLUS;
"-" return MINUS;
"=" return EQUL;
%%
YACC
%{
#include<stdio.h>
int valid=1;
%}
%token NAME NUMBER EQUL PLUS MINUS
%%
Stmt : NAME EQUL exp
        | exp
        ;
exp : NUMBER PLUS NUMBER
        | NUMBER MINUS NUMBER
        | NUMBER MINUS exp
        | NUMBER PLUS exp
```

```
        ;
%%
void yyerror(char * s)
{
        valid=0;
        printf( "%s\n", s);
}
int yywrap(){
        return 1;
}
int main(void)
{
        printf("Enter a expression: \n");
        yyparse();
        if(valid!=0){
                printf("Valid expression \n\n");
        }
        else{
                printf("Invalid expression \n\n");
        }
        return 0;
}
```

Practical 13: Design YACC /Lex code to evaluate arithmetic expression involving operators and without operators +, -, * and / precedence grammar and with operator presidents grammar.

yacc

```
%{
#include<stdio.h>
#include<stdlib.h>
%}
%token PLUS MINUS MUL DIV NEWLINE RPAR LPAR
%token NUMBER
%%
        lines : lines line
        |       ;
        line : expr NEWLINE { printf("%d\n> ", $1); }
        | NEWLINE { printf ("> "); }    ;
        expr : expr PLUS term { $$ = $1 + $3; }
        | expr MINUS term { $$ = $1 - $3; }
        | term { $$ = $1; }     ;
        term : term MUL factor { $$ = $1 * $3; }
        | term DIV factor { if ($3 == 0)
        yyerror("divide by zero");
        else
        $$ = $1 / $3;}
        | factor { $$ = $1; }    ;
        factor : LPAR expr RPAR { $$ = $2; }
        | NUMBER
        { $$ = $1; }     ;
%%
yylex() {
        int c;
        do {
```

```
        c=getchar();
        switch (c) {
        case '0': case '1': case '2': case '3': case '4': case '5': case '6':
        case '7': case '8': case '9':
        yylval= c - '0';
        return NUMBER;
        case '+': return PLUS; break;
        case '-': return MINUS;break;
        case '*': return MUL;break;
        case '/': return DIV;break;
        case '(': return LPAR;break;
        case ')': return RPAR;break;
        case '\n': return NEWLINE; break;    } // Switch case ends
        } while (c!= EOF);
        return(EOF);
}
void yyerror(char * s){          printf ( "%s\n", s);  }
int yywrap(void){return 1;}
void main() {
        printf("Enter a expression: \n");
        yyparse();
}
```

Practical 14: Design desk calculator using YACC and lex code.

lex

```
%{
#include <stdio.h>
# include "y.tab.h"
```

```
extern int yylval;
%}
%%
[0-9]+ {
yylval = atoi(yytext);
return NUMBER;
}
[a-zA-Z]+ {return ALPHA;}
[\t]+ ;
[\n] {return '\n';}
. {return yytext[0];}
%%
int yywrap(void){
    return 0;
}
```

YACC

```
%{
#include <stdio.h>
#include <stdlib.h>
int yylex(void);
int yyerror(char*s);
%}
%token NUMBER ALPHA
%left '+' '-'
%left '*' '/'
%left '(' ')'
%%
```

```
grammer: expr '\n' {
printf("\n Arithematic Expression is Valid.");
printf("\n Expression Result : %d\n",$$);
exit(0);
}
expr : expr'+'expr {$$ = $1 + $3;}
    | expr'*'expr {$$ = $1 * $3;}
    | expr'/'expr {$$ = $1 / $3;}
    | expr'-'expr {$$ = $1 - $3;}
    | '('expr')'  {$$ = $2;}
    | NUMBER      {$$ = $1;}
    | ALPHA
    ;
%%
int main(void){
        printf("Enter the Arithematic Expression : ");
        yyparse();
        return 0;
}
int yyerror(char*s){
        printf("Arthematic Expression is Invalid \n\n");
        exit(0);
}
```