PRACTICAL -1

Problem Statement: Design a LEX Code to count the number of lines, space, tab-meta character and rest of characters in a given input pattern.

Code:

```
%{
    int a=0,b=0,c=0,d=0;
%}

%%
\n {a++;}
\t {b++;}
" " {c++;}
. {d++;}
%%

int yywrap()
{
return 1;
}
int main()
{
    yylex();
    printf("Next Line: %d\nTabs: %d\nSpaces: %d\nCharacters: %d",a,b,c,d);
    return 0;
}
```

OUTPUT

```
The sky is pink.
Hello           There!!
Next Line: 2
Tabs: 2
Spaces: 3
```

PRACTICAL -2

Problem Statement: Design a LEX Code to identify and print valid identifier of C/C++ in a given input pattern.

Code:

```
%{
# include<stdio.h>
int valid=0;
%}
%%
float {valid=0;}
int {valid=0;}
double {valid=0;}
^[_A-Za-z][_A-Za-z0-9]* {valid=1;}
. {valid=0;}
%%
int yywrap()
{
return 1;
}
int main()
{
yylex();
if(valid)
{
printf("VALID");
}
else
{
printf("NOT VALID");
}
```

```
return 0;
}
```

OUTPUT

```
cars

VALID


12cars

NOT VALID
int

NOT VALID
```

PRACTICAL -3

Problem Statement: Design a LEX Code to identify and print integer and float value in a given input pattern.

Code:

```
%{
#include <stdio.h>
%}
%%
[0-9]+\.[0-9]* { printf("Float value: %s\n", yytext); }
\.[0-9]+       { printf("Float value: %s\n", yytext); }
[0-9]+         { printf("Integer value: %s\n", yytext); }
[ \t\n]        ;  // Ignore whitespace
.              ;  // Ignore any other characters
%%
int yywrap() {
    return 1;
}
int main() {
    printf("Enter integers and floats:\n");
    yylex();
    return 0;
}
```

OUTPUT

```
Enter integers and floats:
10
Integer value: 10
3.0
Float value: 3.0
20 0.56
Integer value: 20
Float value: 0.56
```

PRACTICAL -4

Problem Statement: Design a LEX Code for tokenizing (Identify and print OPERATORS, SEPERATORS, KEYWORDS, IDENTIFERS) the following C-fragment:

```
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++; }
```

Code:

```
%{
#include <stdio.h>
#include <string.h>
char keywords[100][20];
char identifiers[100][20];
char operators[100][5];
char separators[100][5];
char int_consts[100][20];
char float_consts[100][20];
int k=0, id=0, op=0, sep=0, ic=0, fc=0;
// Utility function to avoid duplicates
int exists(char arr[][20], int count, const char *val) {
    for (int i = 0; i < count; i++) {
        if (strcmp(arr[i], val) == 0) return 1;
    }
    return 0;
}
%}
%%
"int"|"float"|"while"|"if"|"else" {
    if (!exists(keywords, k, yytext)) {
```

```
        strcpy(keywords[k++], yytext);
    }
}
[a-zA-Z_][a-zA-Z0-9_]* {
    if (!exists(identifiers, id, yytext)) {
        strcpy(identifiers[id++], yytext);
    }
}
[0-9]+\.[0-9]+ {
    if (!exists(float_consts, fc, yytext)) {
        strcpy(float_consts[fc++], yytext);
    }
}
[0-9]+ {
    if (!exists(int_consts, ic, yytext)) {
        strcpy(int_consts[ic++], yytext);
    }
}
"=="|"++"|"+"|"-"|"*"|"/"|"<="|">="|"="|"<"|">" {
    if (!exists(operators, op, yytext)) {
        strcpy(operators[op++], yytext);
    }
}
";"|","|"("|")"|"{"|"}" {
    if (!exists(separators, sep, yytext)) {
        strcpy(separators[sep++], yytext);
    }
}
[ \t\n]   ;
. ;
%%
int yywrap() { return 1; }
```

```
int main() {
    printf("Tokenizing the input C fragment...\n\n");
    yylex();
    printf("\n--- TOKENS ---\n");
    printf("\nKeywords:\n");
    for (int i = 0; i < k; i++) printf("%s\n", keywords[i]);

    printf("\nIdentifiers:\n");
    for (int i = 0; i < id; i++) printf("%s\n", identifiers[i]);
    printf("\nOperators:\n");
    for (int i = 0; i < op; i++) printf("%s\n", operators[i]);
    printf("\nSeparators:\n");
    for (int i = 0; i < sep; i++) printf("%s\n", separators[i]);
    printf("\nInteger Constants:\n");
    for (int i = 0; i < ic; i++) printf("%s\n", int_consts[i]);
    printf("\nFloat Constants:\n");
    for (int i = 0; i < fc; i++) printf("%s\n", float_consts[i]);
    return 0;
}
```

OUTPUT

```
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++; }


--- TOKENS ---

Keywords:
int
float
while
if
else

Identifiers:
p
d
r
m
n

Operators:
=
<=
==
+
*
+
++
++
+
++
```

```
Separators:
,
;
;
(
)
{
(
{
;
}
{
}
}

Integer Constants:
1
0
4
3

Float Constants:
0.0
200.0
4.5
1000.0
```

PRACTICAL -5

Problem Statement: Design a LEX Code to count and print the total number of characters, words, white spaces and lines in a given file named as ‘Input.txt’.

Code:

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

int char_count = 0;
int word_count = 0;
int space_count = 0;
int line_count = 0;
%}

%%

\n          { line_count++; char_count++; space_count++; }

[^\t\n\r \f\v]+  { word_count++; char_count += yyleng; }

[ \t\r\f\v] { space_count++; char_count++; }

.           { char_count++; }

%%

int yywrap() {
    return 1;
}

int main() {
    FILE *fp = fopen("Input.txt", "r");
    if (!fp) {
```

```
        printf("Cannot open file Input.txt\n");
        return 1;
    }

    yyin = fp;
    yylex();
    fclose(fp);

    printf("\n--- File Statistics ---\n");
    printf("Total Characters: %d\n", char_count);
    printf("Total Words: %d\n", word_count);
    printf("Total Whitespaces: %d\n", space_count);
    printf("Total Lines: %d\n", line_count);

    return 0;
}
```

OUTPUT

input.txt

```
1  A random paragraph can also be an excellent way for a writer to tackle writers' block.
2  Writing block can often happen due to being stuck with a current project that the writer is trying to complete.
3
```

```
--- File Statistics ---
Total Characters: 202
Total Words: 36
Total Whitespaces: 39
Total Lines: 2
```

PRACTICAL -6

Problem Statement: Design a LEX Code to replace all the white spaces of ‘Input.txt’ file by a single blank character and store the output in ‘Output.txt’ file.

Code:

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

FILE *out;
int last_was_space = 0;
%}

%%

[ \t\n\r\f\v]+ {
    if (!last_was_space) {
        fputc(' ', out);
        last_was_space = 1;
    }
}

. {
    fputc(yytext[0], out);
    last_was_space = 0;
}

%%

int yywrap() {
    return 1;
}

int main() {
```

```
    FILE *in = fopen("Input.txt", "r");
    out = fopen("Output.txt", "w");

    if (!in) {
        printf("Cannot open Input.txt\n");
        return 1;
    }
    if (!out) {
        printf("Cannot create Output.txt\n");
        return 1;
    }

    yyin = in;
    yylex();

    fclose(in);
    fclose(out);

    printf("Whitespace replaced and result stored in Output.txt\n");
    return 0;
}
```

OUTPUT

input.txt

```
A random paragraph can also be an
excellent way for a writer to              tackle writers'
block.
```

output.txt

```
A random paragraph can also be an excellent way for a writer to tackle writers' block.
```

PRACTICAL -7

Problem Statement: Design a LEX Code to remove the comments from any C-Program (in.c) given at run time and store into ‘out.c’ file.

Code:

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

FILE *out;
%}

%%

"/*"([^*]|\*+[^*/])*"*"+"/"    { /* Remove multi-line comment - don't write */ }

"//".*                         { /* Remove single-line comment - don't write */ }

.|\n                           { fputc(yytext[0], out); }

%%

int yywrap() {
   return 1;
}

int main() {
   FILE *in = fopen("in.c", "r");
   out = fopen("out.c", "w");

   if (!in) {
      printf("Cannot open in.c\n");
      return 1;
   }
```

```
    if (!out) {
        printf("Cannot open or create out.c\n");
        return 1;
    }

    yyin = in;
    yylex();

    fclose(in);
    fclose(out);

    printf("Comments removed. Cleaned code written to out.c\n");
    return 0;
}
```

OUTPUT

in.c > main()

```c
#include <stdio.h>

// This is a single-line comment

/*
 This is a
 multi-line comment
*/

int main() {
    printf("Hello, World!\n");  // Print greeting
    /* Another comment */
    return 0;
}
```

out.c > ...

```c
#include <stdio.h>

int main() {
    printf("Hello, World!\n");

    return 0;
}
```

PRACTICAL -8

Problem Statement: Design a LEX Code to extract all html tags in the HTML file given at run time and store into text file given at run time.

Code:

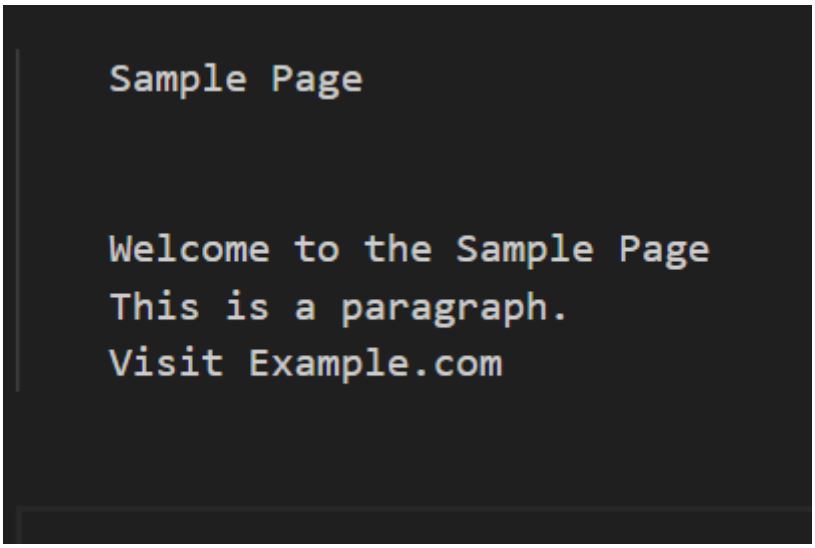
```
%{
#include <stdio.h>
#include <string.h>

FILE *out;
%}

%%

\<[^>]+\>      ;
.|\n           { fprintf(out, "%s\n", yytext); }

%%

int yywrap() {
    return 1;
}

int main(int argc, char *argv[]) {
    if (argc != 3) {
        printf("Usage: %s input.html output.txt\n", argv[0]);
        return 1;
    }

    FILE *in = fopen(argv[1], "r");
    out = fopen(argv[2], "w");
```

```
    if (!in) {
        printf("Cannot open input file: %s\n", argv[1]);
        return 1;
    }
if (!out) {
        printf("Cannot open output file: %s\n", argv[2]);
        fclose(in);
        return 1;
    }

    yyin = in;
    yylex();

    fclose(in);
    fclose(out);

    printf("Tags extracted to %s\n", argv[2]);
    return 0;
}
```

OUTPUT

input.txt

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>Sample Page</title>
</head>
<body>
    <h1>Welcome to the Sample Page</h1>
    <p>This is a paragraph.</p>
    <a href="https://example.com">Visit Example.com</a>
</body>
</html>

```



PRACTICAL -9

Problem Statement: Design a DFA in LEX Code which accepts string containing even number of ‘a’ and even number of ‘b’ over the input alphabet {a, b}.

Code:

```
%{
#include <stdio.h>
%}
%s EVEN_EVEN ODD_EVEN EVEN_ODD ODD_ODD
%%

<INITIAL>a        BEGIN ODD_EVEN;
<INITIAL>b        BEGIN EVEN_ODD;
<INITIAL>\n       BEGIN INITIAL; printf("VALID: Even 'a' and Even 'b'\n");

<EVEN_EVEN>a      BEGIN ODD_EVEN;
<EVEN_EVEN>b      BEGIN EVEN_ODD;
<EVEN_EVEN>\n     BEGIN INITIAL; printf("VALID: Even 'a' and Even 'b'\n");

<ODD_EVEN>a       BEGIN EVEN_EVEN;
<ODD_EVEN>b       BEGIN ODD_ODD;
<ODD_EVEN>\n      BEGIN INITIAL; printf("INVALID: Odd 'a' and Even 'b'\n");

<EVEN_ODD>a       BEGIN ODD_ODD;
<EVEN_ODD>b       BEGIN EVEN_EVEN;
<EVEN_ODD>\n      BEGIN INITIAL; printf("INVALID: Even 'a' and Odd 'b'\n");

<ODD_ODD>a        BEGIN EVEN_ODD;
<ODD_ODD>b        BEGIN ODD_EVEN;
<ODD_ODD>\n       BEGIN INITIAL; printf("INVALID: Odd 'a' and Odd 'b'\n");

%%
```

```
int yywrap() {
    return 1;
}

int main() {
    printf("Enter strings containing 'a' and 'b' (press Enter to evaluate each string):\n");
    yylex();
    return 0;
}
```

OUTPUT

```
Enter strings containing 'a' and 'b' (press Enter to evaluate each string):
abaabbbaaaab
INVALID: Odd 'a' and Odd 'b'
aabbbbaabbaababa
VALID: Even 'a' and Even 'b'
```

PRACTICAL -10

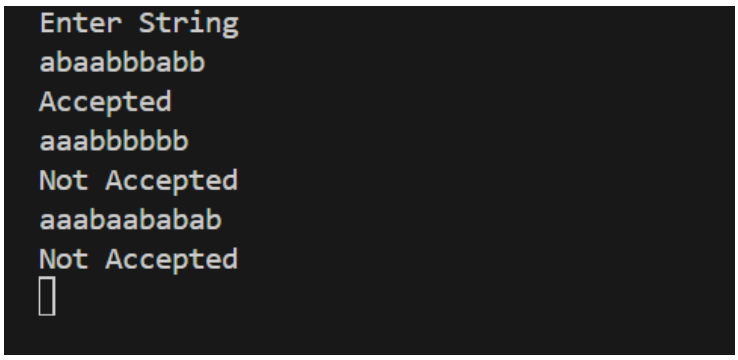
Problem Statement: Design a DFA in LEX Code which accepts string containing third last element ‘a’ over the input alphabet {a, b}.

Code:

```
%{
%}
%s A B C D E F G
%%

<INITIAL>b BEGIN INITIAL;
<INITIAL>a BEGIN A;
<INITIAL>\n BEGIN INITIAL; {printf("Not Accepted\n");}

<A>b BEGIN F;
<A>a BEGIN B;
<A>\n BEGIN INITIAL; {printf("Not Accepted\n");}

<B>b BEGIN D;
<B>a BEGIN C;
<B>\n BEGIN INITIAL; {printf("Not Accepted\n");}

<C>b BEGIN D;
<C>a BEGIN C;
<C>\n BEGIN INITIAL; {printf("Accepted\n");}

<D>b BEGIN G;
<D>a BEGIN E;
<D>\n BEGIN INITIAL; {printf("Accepted\n");}

<E>b BEGIN F;
<E>a BEGIN B;
<E>\n BEGIN INITIAL; {printf("Accepted\n");}
```

```
<F>b BEGIN G;
<F>a BEGIN E;
<F>\n BEGIN INITIAL; {printf("Not Accepted\n");}

<G>b BEGIN INITIAL;
<G>a BEGIN A;
<G>\n BEGIN INITIAL; {printf("Accepted\n");}

. ;

%%

int yywrap()
{
   return 1;
}

int main()
{
   printf("Enter String\n");
   yylex();
   return 0;
}
```

OUTPUT


Enter String
abaabbbabb
Accepted
aaabbbbbb
Not Accepted
aaabaababab
Not Accepted

PRACTICAL -11

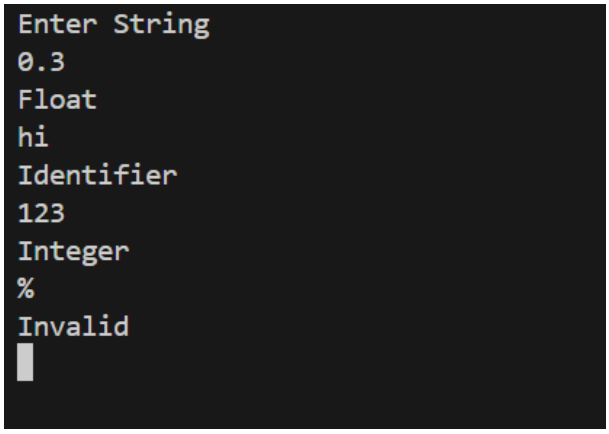
Problem Statement: Design a DFA in LEX Code to identify and print integer & float constants and identifier.

Code:

```
%{
%}

%s A B C DEAD

%%

<INITIAL>[0-9]+ BEGIN A;
<INITIAL>[0-9]+[.][0-9]+ BEGIN B;
<INITIAL>[A-Za-z_][A-Za-z0-9_]* BEGIN C;
<INITIAL>[^\n] BEGIN DEAD;
<INITIAL>\n BEGIN INITIAL; {printf("Not Accepted\n");}

<A>[^\n] BEGIN DEAD;
<A>\n BEGIN INITIAL; {printf("Integer\n");}

<B>[^\n] BEGIN DEAD;
<B>\n BEGIN INITIAL; {printf("Float\n");}

<C>[^\n] BEGIN DEAD;
<C>\n BEGIN INITIAL; {printf("Identifier\n");}

<DEAD>[^\n] BEGIN DEAD;
<DEAD>\n BEGIN INITIAL; {printf("Invalid\n");}

%%

int yywrap()
```

```
{
 return 1;
}

int main()
{
 printf("Enter String\n");
 yylex();
 return 0;
}
```

OUTPUT



PRACTICAL -12

Problem Statement: Design YACC/LEX code to recognize the valid string from the language L= {anbn | n>=1}.

Code:

Lang.l

```
%{
#include "y.tab.h"
%}

%%

a    { return A; }
b    { return B; }
\n   { return '\n'; }
.    { return yytext[0]; }

%%
```

Lang.y

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

%token A B

%%

start:
    string '\n'   { printf("Valid string in a^n b^n format\n"); exit(0); }
  | '\n'          { printf("Empty input is not valid\n"); exit(1); }
  ;
```

```
string:
    A string B    // Recursively match a's and b's
  | A B           // Base case: one 'a' followed by one 'b'
  ;

%%

int main() {
    printf("Enter a string from the language L = { a^n b^n | n >= 1 }:\n");
    yyparse();
    return 0;
}

int yyerror(const char *s) {
    printf("Invalid string: Not in a^n b^n format\n");
    exit(1);
}
```

OUTPUT

```
Enter a string from the language L = { a^n b^n | n >= 1 }:
aaaaaabbbbbb
Valid string in a^n b^n format
```

```
Enter a string from the language L = { a^n b^n | n >= 1 }:
aabbbbb
Invalid string: Not in a^n b^n format
```

PRACTICAL -13

Problem Statement: Design YACC/LEX code to recognize valid arithmetic expression with operators +, -, * and /.

Code:

B1.l

```
%{
#include<stdio.h>
#include "y.tab.h"
extern int yylval;
%}

%%
[0-9]+ {
        yylval=atoi(yytext);
        return NUMBER;

      }
[\t] ;

[\n] return 0;

. return yytext[0];

%%

int yywrap()
{
 return 1;
}
```

Calc.y

```
%{
#include<stdio.h>
int flag = 0;
%}

%token NUMBER

%left '+' '-'
%left '*' '/' '%'
%left '(' ')'

%%

ArithmeticExpression: E {
        if (flag == 0)
            printf("\nValid arithmetic expression\n");
        else
            printf("\nInvalid arithmetic expression\n");
        return 0;
    };

E: E '+' E { }
 | E '-' E { }
 | E '*' E { }
 | E '/' E { }
 | E '%' E { }
 | '(' E ')' { }
 | NUMBER { }
 ;

%%
```

```
int main()
{
    printf("\nEnter Arithmetic Expression:\n");
    yyparse();
    return 0;
}

int yyerror(const char *s)
{
    printf("\nError: %s\n", s);
    flag = 1;
    return 1;
}
```

OUTPUT

```
Enter Arithmetic Expression:
2+3*4-5

Valid arithmetic expression
```

```
Enter Arithmetic Expression:
2+3-(4**5)

Invalid arithmetic expression
```

PRACTICAL -14

Problem Statement: Design YACC/LEX code to evaluate the arithmetic expression involving operators +, -, * and / with operator precedence grammar.

Code:

B1.l

```
%{
#include<stdio.h>
#include "y.tab.h"
extern int yylval;
%}

%%
[0-9]+ {
        yylval=atoi(yytext);
        return NUMBER;

      }
[\t] ;

[\n] return 0;

. return yytext[0];

%%

int yywrap()
{
 return 1;
}
```

Calc.y

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

%token NUMBER

%left '+' '-'
%left '*' '/' '%'
%left '(' ')'

%%

ArithmeticExpression: E {
        printf("\nResult = %d\n", $1);
        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; }
 ;

%%

int main()
{
   printf("\nEnter Arithmetic Expression:\n");
```

```
    yyparse();
    return 0;
}

int yyerror(const char *s)
{
    printf("\nError: %s\n", s);
    return 1;
}
```

OUTPUT

```
Enter Arithmetic Expression:
2+3-8

Result = -3
```

```
Enter Arithmetic Expression:
(5-4)+6*7

Result = 43
```