Practical No1

Title: Implement a lexical analyzer for identification of numbers.

(Numbers can be binary, octal, hexadecimal, decimal, float or exponential)

```
/********************* identification.l ***********************/

/*PROGRAM FOR IDENTIFICATION OF NUMBERS.*/

%{
#include<stdio.h>
%}

%%

                        {printf("Please Enter Any Character\t");}

[0|1]+                  {printf("%s is Binary Number\t",yytext);}
[0-7]+                  {printf("%s is Octal Number\t",yytext);}
[0-9 A-F a-f]+                  {printf("%s is Hexa Decimal Number\t",yytext);}
[0-9]*                  {printf("%s is Decimal Number\t",yytext);}
[eE][-+]?[0-9]*                 {printf("%s is Exponential Number\t",yytext);}
[0-9]+\.                {printf("%s is a Not Valid Number\t",yytext);}
[0-9]*\.[0-9]+                  {printf("%s is Floating Point Number\t",yytext);}
```

```
[g-zG-Z]+                    {printf("%s is an Invalid Character\t",yytext);}

%%

int main()
{
yylex();
}

int yywrap()
{
}

/****************** OUTPUT *****************/

shweta123@ubuntu:~$ lex identification.l
shweta123@ubuntu:~$ cc lex.yy.c -ll
shweta123@ubuntu:~$ ./a.out

Please Enter Any Character    0
0 is Binary Number
1
1 is Binary Number
2
```

2 is Octal Number

3

3 is Octal Number

4

4 is Octal Number

5

5 is Octal Number

6

6 is Octal Number

7

7 is Octal Number

8

8 is Hexa Decimal Number

9

9 is Hexa Decimal Number

A

A is Hexa Decimal Number

B

B is Hexa Decimal Number

C

C is Hexa Decimal Number

D

D is Hexa Decimal Number

E

E is Hexa Decimal Number

F

F is Hexa Decimal Number

a

a is Hexa Decimal Number

b

b is Hexa Decimal Number

c

c is Hexa Decimal Number

d

d is Hexa Decimal Number

e

e is Hexa Decimal Number

f

f is Hexa Decimal Number

G

G is an Invalid Character

Z

Z is an Invalid Character

g

g is an Invalid Character

z

z is an Invalid Character

9.

9. is a Not Valid Number

9.9

9.9 is Floating Point Number

99.99

99.99 is Floating Point Number

10.10

10.10 is Floating Point Number

E+1

E+1 is Exponential Number

E-1

E-1 is Exponential Number

e+1

e+1 is Exponential Number

e-1

e-1 is Exponential Number

E+10

E+10 is Exponential Number

E+99

E+99 is Exponential Number

e+11

e+11 is Exponential Number

e-11

e-11 is Exponential Number

Practical No.2

Title: Implement a lexical analyzer for identification of numbers.

(Numbers can be binary, octal, hexadecimal, decimal, float or exponential)

Theory: Decimal Number

A decimal number is constructed by summing multiples of the digits: i.e. thousands, hundreds, tens, units - for a four digit decimal number.

e.g. 576=5*100+7*10+6*1

The number to multiply each digit by is 10^d where d is the position of the digit (0 for units, 1 for tens etc..). The base for this power is called the **Radix.** When dealing with computers it is common to use radices other than 10. The most important radices are 2(binary) 8(octal) and 16(hexadecimal).

Binary Number

In binary, each digit can only be 0 or 1. A binary digit is called a bit. The columns in a binary number represent powers of 2.

Decimal	Binary
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

```
/********************* identification.l ***********************/

/*PROGRAM FOR IDENTIFICATION OF NUMBERS.*/

%{
#include<stdio.h>
%}

%%

                        {printf("Please Enter Any Character\t");}

[0|1]+                  {printf("%s is Binary Number\t",yytext);}
[0-7]+                  {printf("%s is Octal Number\t",yytext);}
[0-9 A-F a-f]+                  {printf("%s is Hexa Decimal Number\t",yytext);}
[0-9]*                  {printf("%s is Decimal Number\t",yytext);}
[eE][-+]?[0-9]*                 {printf("%s is Exponential Number\t",yytext);}
[0-9]+\.                {printf("%s is a Not Valid Number\t",yytext);}
[0-9]*\.[0-9]+                  {printf("%s is Floating Point Number\t",yytext);}
[g-zG-Z]+                       {printf("%s is an Invalid Character\t",yytext);}

%%
```

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

int yywrap()
{
}
```

/****************** OUTPUT ******************/

```
shweta123@ubuntu:~$ lex identification.l
shweta123@ubuntu:~$ cc lex.yy.c -ll
shweta123@ubuntu:~$ ./a.out

Please Enter Any Character    0
0 is Binary Number
1
1 is Binary Number
2
2 is Octal Number
3
3 is Octal Number
```

4

4 is Octal Number

5

5 is Octal Number

6

6 is Octal Number

7

7 is Octal Number

8

8 is Hexa Decimal Number

9

9 is Hexa Decimal Number

A

A is Hexa Decimal Number

B

B is Hexa Decimal Number

C

C is Hexa Decimal Number

D

D is Hexa Decimal Number

E

E is Hexa Decimal Number

F

F is Hexa Decimal Number

a

a is Hexa Decimal Number

b

b is Hexa Decimal Number

c

c is Hexa Decimal Number

d

d is Hexa Decimal Number

e

e is Hexa Decimal Number

f

f is Hexa Decimal Number

G

G is an Invalid Character

Z

Z is an Invalid Character

g

g is an Invalid Character

z

z is an Invalid Character

9.

9. is a Not Valid Number

9.9

9.9 is Floating Point Number

99.99

99.99 is Floating Point Number

10.10

10.10 is Floating Point Number

E+1

E+1 is Exponential Number

E-1

E-1 is Exponential Number

e+1

e+1 is Exponential Number

e-1

e-1 is Exponential Number

E+10

E+10 is Exponential Number

E+99

E+99 is Exponential Number

e+11

e+11 is Exponential Number

e-11

e-11 is Exponential Number

Practical No. 3

Title: Implement a calculator using LEX and YACC.

Aim: Implement a Calculator using LEX and YACC.

Theory: Calculator to perform arithmetic operations using LEX and YACC

Grammars for yacc are described using a variant of Backus Naur Form (BNF). This technique, pioneered by John Backus and Peter Naur, was used to describe ALGOL60. A BNF grammar can be used to express *context-free* languages. Most constructs in modern programming languages can be represented in BNF. For example, the grammar for an expression that multiplies and adds numbers is

1 E -> E + E

2 E -> E * E

3 E -> id

Three productions have been specified. Terms that appear on the left-hand side (lhs) of a production, such as E, are nonterminals. Terms such as id (identifier) are terminals (tokens returned by lex) and only appear on the right-hand side (rhs) of a production. This grammar specifies that an expression may be the sum of two expressions, the product of two expressions, or an identifier. We can use this grammar to generate expressions:

E -> E * E (r2)

-> E * z (r3)

-> E + E * z (r1)

-> E + y * z (r3)

-> x + y * z (r3)

Logic:

lex and **yacc** commands.

calci.y

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

%union { double p;}                    //to define possible symbol types
%token <p> num

/*Defining the Precedence and Associativity*/

%left '+','-'                          //lowest precedence
%left '*','/'                          //highest precedence
%type <p> exp                          //Sets the type for non - terminal

%%
```

```
/* for storing the answer */
exp1    :       exp {printf("result is %g\n",$1);}

/* for binary arithmatic operators */
exp     :       exp'+'exp       { $$=$1+$3; }

                |exp'-'exp      { $$=$1-$3; }

                |exp'*'exp      { $$=$1*$3; }

                |exp'/'exp      {
                                        if($3==0)
                                        {
                                        printf("Divide By Zero");
                                        //exit(0);
                                        }
                                        else $$=$1/$3;
                                }

                 |'-'exp        {$$=-$2;}

                 |'('exp')'     {$$=$2;}
```

```
                |num;
%%

void main()
{
   do
        {
                yyparse();
        }while(1);
}

yyerror(char *s)
{
     printf("Standard Errors ");
}
```

calci.l

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

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

[0-9]+|[0-9]*\.[0-9]+  {
                yylval.p = atof(yytext);
                return num;
                }

[\t];
\n      return 0;
.       return yytext[0];

%%
```

/********** OUTPUT **********/

```
[root@localhost ~]# lex calci.l
[root@localhost ~]# yacc calci.y
[root@localhost ~]# cc lex.yy.c y.tab.c -ll
calci.l: In function 'yylex':
calci.l:8: error: 'yylval' undeclared (first use in this function)
```

calci.l:8: error: (Each undeclared identifier is reported only once

calci.l:8: error: for each function it appears in.)

calci.l:9: error: 'num'undeclared (first use in this function)

[root@localhost ~]# lex calci.l

[root@localhost ~]# yacc -d calci.y

[root@localhost ~]# cc lex.yy.c y.tab.c -ll

[root@localhost ~]# ./a.out

2+3

=5

2-3

=-1

3*4

=12

4-2

=2

10+10

=20

12/2

=6

12/0

Divide By Zero

PRACTICAL NO. 4

Aim: Implementation of Context Free Grammar

Theory: **Context-Free Grammars (CFG's)**

CFG's are very useful for representing the syntactic structure of programming languages. A CFG is sometimes called Backus-Naur Form (BNF).

A context-free grammar consists of

- A finite set of terminal symbols,
- A finite nonempty set of nonterminal symbols,
- One distinguished nonterminal called the start symbol, and
- A finite set of rewrite rules, called productions, each of the form A → α where A is a nonterminal and α is a string (possibly empty) of terminals and nonterminals.

Consider the context-free grammar G with the productions

E → E + T | T

T → T * F | F

F → (E) | id

- The terminal symbols are the alphabet from which strings are formed. In this grammarthe set of terminal symbols is { id, +, *, (,) }. The terminal symbols are the token names.
- The nonterminal symbols are syntactic variables that denote sets of strings of terminal symbols. In this grammar the set of nonterminal symbols is {E, T, F}.
- The start symbol is E.

Derivations and Parse Trees

L(G), the language generated by a grammar G, consists of all strings of terminal symbols that can be derived from the start symbol of G.

A leftmost derivation expands the leftmost nonterminal in each sentential form:

E ⇒ E + T
⇒ T + T
⇒ F + T
⇒ id + T
⇒ id + T * F
⇒ id + F * F
⇒ id + id * F
⇒ id + id * id

A rightmost derivation expands the rightmost non terminal in each sentential form:

E ⇒ E + T
⇒ E + T * F
⇒ E + T * id
⇒ E + F * id
⇒ E + id * id
⇒ T + id * id
⇒ F + id * id
⇒ id + id * id

Note that these two derivations have the same parse tree.

Title: Implementation of context Free Grammar.

CFG.y

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

```
%token One Zero Nl

%%

S:A Nl {printf("The String is Valid ");};
A:One Zero B |One Zero;
B:Zero B |One B|Zero |One;

%%
int main()
{
printf("Please enter any string in a format 0 or 1 \n");
yyparse();
}

void yyerror()
{
printf("Error\n");
}
```

CFG.l

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

%%
"0" { return Zero;}
"1" { return One;}
[\n] { return Nl;}
.;
%%
```

/********** OUTPUT **********/

```
shweta123@ubuntu:~$ lex CFG.l

shweta123@ubuntu:~$ yacc -d CFG.y

shweta123@ubuntu:~$ cc y.tab.c lex.yy.c -ll

shweta123@ubuntu:~$ ./a.out
```

Please enter any string in a format 0 or 1

10

The String is Valid

shweta123@ubuntu:~$ ./a.out

Please enter any string in a format 0 or 1

100

The String is Valid

shweta123@ubuntu:~$ ./a.out

Please enter any string in a format 0 or 1

101

The String is Valid

shweta123@ubuntu:~$ ./a.out

Please enter any string in a format 0 or 1

1000

The String is Valid

shweta123@ubuntu:~$ ./a.out

Please enter any string in a format 0 or 1

1001

The String is Valid

shweta123@ubuntu:~$ ./a.out

Please enter any string in a format 0 or 1

1010

The String is Valid shweta123@ubuntu:~$ ./a.out

Please enter any string in a format 0 or 1

1011

The String is Valid

PRACTICAL NO. 5

Aim: Implementation of code generator

Theory: A code generator is expected to have an understanding of the target machine's runtime environment and its instruction set. The code generator should take the following things into consideration to generate the code:

- **Target language**: The code generator has to be aware of the nature of the target language for which the code is to be transformed. That language may facilitate some machine-specific instructions to help the compiler generate the code in a more convenient way. The target machine can have either CISC or RISC processor architecture.
- **IR Type**: Intermediate representation has various forms. It can be in Abstract Syntax Tree (AST) structure, Reverse Polish Notation, or 3-address code.
- **Selection of instruction**: The code generator takes Intermediate Representation as input and converts (maps) it into target machine's instruction set. One representation can have many ways (instructions) to convert it, so it becomes the responsibility of the code generator to choose the appropriate instructions wisely.
- **Register allocation**: A program has a number of values to be maintained during the execution. The target machine's architecture may not allow all of the values to be kept in the CPU memory or registers. Code generator decides what values to keep in the registers. Also, it decides the registers to be used to keep these values.
- **Ordering of instructions**: At last, the code generator decides the order in which the instruction will be executed. It creates schedules for instructions to execute them.

The code generator has to track both the registers (for availability) and addresses (location of values) while generating the code. For both of them, the following two descriptors are used:

- **Register descriptor**: Register descriptor is used to inform the code generator about the availability of registers. Register descriptor keeps track of values stored in each register. Whenever a new register is required during code generation, this descriptor is consulted for register availability.
- **Address descriptor**: Values of the names (identifiers) used in the program might be stored at different locations while in execution. Address descriptors are used to keep track of memory locations where the values of identifiers are stored. These locations may include CPU registers, heaps, stacks, memory or a combination of the mentioned locations.

```
**********     codegenerator.c  **********

#include<stdio.h>
#include<ctype.h>
#include<string.h>
void main()
{
char a[20];
int i=4,n;
clrscr();
printf("\nENTER ANY EXPRESSION:");
scanf("%s", a);
n=strlen(a);
printf("\n***************GENERATED CODE************************\n");
if(isalpha(a[4])||isdigit(a[4]))
        printf("\n\t\t\t\tMOV A,%c\n",a[i]);
i+=2;
while(i<n)
{
switch(a[i])
        {
                case '*':printf("\t\t\t\tMUL A,B\n");
                i+=2;
                break;
case '/':printf("\t\t\t\tDIV A,B\n");
                i+=2;
                break;
case '+':printf("\t\t\t\tADD A,B\n");
```

```
                i+=2;
                break;
case '-':printf("\t\t\t\tSUB A,B\n");
                i+=2;
                break;
default:if(isalpha(a[i]))
                printf("\t\t\t\tMOV B,%c\n",a[i]);
                else
                printf("\t\t\t\tMOV A,%c\n",a[i]);
                i--;
}
}
i=2;
if(isalpha(a[i])||isdigit(a[i]))
        printf("\n\t\t\t\tMOV C,%c\n",a[i]);
i=3;
while(i<n)
{
switch(a[i])
        {
                case '*':printf("\t\t\t\tMUL C,A\n");
                i+=4;
                break;
case '/':printf("\t\t\t\tDIV C,A\n");
                i+=4;
                break;
case '+':printf("\t\t\t\tADD C,A\n");
                i+=4;
                break;
case '-':printf("\t\t\t\tSUB C,A\n");
                i+=4;
                break;
default:if(isalpha(a[i]))
                printf("\t\t\t\tMOV B,%c\n",a[i]);
                else
                printf("\t\t\t\tMOV A,%c\n",a[i]);
                i--;
}
}
i=0;
printf("\t\t\t\tMOV %c,A\n",a[i]);
```

- getch();
- }
- ********** OUTPUT FOR 4 VARIABLES **********
- ENTER ANY EXPRESSION: c= a + b * d
-
- ********** GENERATED CODE **********
- MOV A, b
- MOV B, d
- MUL A, B
-
- MOV C, a
- ADD C, A
- MOV c, A
-
- ********** OUTPUT FOR 3 VARIABLES **********
- ENTER ANY EXPRESSION: c = a + b
-
- ********** GENERATED CODE **********
- MOV A, b
-
- MOV C, a
- ADD C, A
- MOV c, A

PRACTICAL NO. 6

Aim: Implement Deterministic Finite Automata

Theory: Deterministic Finite Automaton (DFA)

In DFA, for each input symbol, one can determine the state to which the machine will move. Hence, it is called **Deterministic Automaton.** As it has a finite number of states, the machine is called **Deterministic Finite Machine** or **Deterministic Finite Automaton.**

Formal Definition of a DFA

A DFA can be represented by a 5-tuple $(Q, \sum, \delta, q_0, F)$ where −

- **Q** is a finite set of states.
- $\sum$ is a finite set of symbols called the alphabet.
- **δ** is the transition function where $\delta: Q \times \sum \rightarrow Q$
- $\mathbf{q_0}$ is the initial state from where any input is processed $(q0 \in Q)$.
- **F** is a set of final state/states of Q $(F \subseteq Q)$.

Graphical Representation of a DFA

A DFA is represented by digraphs called **state diagram.**

- The vertices represent the states.
- The arcs labeled with an input alphabet show the transitions.
- The initial state is denoted by an empty single incoming arc.
- The final state is indicated by double circles.

Logic:

Conclusion: Thus, Deterministic Finite Automata has been implemented successfully.

Title: Implement Deterministic Finite Automata.

```
//WRITE A C PROGRAM TO IMPLEMENT THE DFA.
#include<stdio.h>
#include<conio.h>
int ninputs;
int check(char,int);                //function declaration
int dfa[10][10];
char c[10], string[10];
void main()
{
    int nstates,nfinals;
    int f[10];
    int i,j,s=0,final=0;
    clrscr();
    printf("Enter the number of states that your DFA consist of \n");
    scanf("%d",&nstates);
    printf("\nEnter the number of input symbol that DFA have \n");
    scanf("%d",&ninputs);
    printf("\nEnter the input symbols\t");
    for(i=0; i<ninputs; i++)
    {
```

```
        printf("\n\n %d input\t", i+1);
        printf("%c",c[i]=getch());
}
printf("\n\nenter number of final states\t");
scanf("%d",&nfinals);
for(i=0;i<nfinals;i++)
{
        printf("\n\nFinal state %d : q",i+1);
        scanf("%d",&f[i]);
}
printf("-----------------------------------------------------------------------");
printf("\n\ndefine transition rule as (Initial State, Input Symbol ) = Final State\n");
for(i=0;i<ninputs; i++)
{
        for(j=0;j<nstates; j++)
        {
            printf("\n(q%d , %c ) = q",j,c[i]);
            scanf("%d",&dfa[i][j]);
        }
}
        i=0;
        printf("\n\nEnter Input String\t");
```

```
        scanf("%s",string);


        while(string[i]!='\0')
        if((s=check(string[i++],s))<0)
        break;
        for(i=0 ;i<nfinals ;i++)
        if(f[i] ==s )
        final=1;
        if(final==1)
        printf("\n Valid String");
        else
        printf("\n Invalid String");
        getch();
    getch();
}
int check(char b,int d)
{
    int j;
    for(j=0; j<ninputs; j++)
    if(b==c[j])
    return(dfa[d][j]);
    return -1;
```

```
}

/****** Output for DFA with 2 states for acceptance of string ******/

Enter the number of states that your DFA consists of

2

Enter the number of input symbol that DFA have

2

Enter the input symbols 2

 1 input       0

 2 input       1

enter number of final states     1

Final state 1 : q0

-----------------------------------------------

define transition rule as (Initial State, Input Symbol ) = Final State
```

(q0 , 0) = q1

(q1 , 0) = q0

(q0 , 1) = q0

(q1 , 1) = q1

Enter Input String 0101

Valid String

/******Output for DFA with 2 states for rejection of string *******/

Enter the number of states that your DFA consists of

2

Enter the number of input symbol that DFA have

2

Enter the input symbols 2

1 input 0

2 input 1

enter number of final states 1

Final state 1 : q0

---

define transition rule as (Initial State, Input Symbol) = Final State

(q0 , 0) = q1

(q1 , 0) = q0

(q0 , 1) = q0

(q1 , 1) = q1

Enter Input String 01010

Invalid String

/****** Output for DFA with 3 states for acceptance of string *******/

Enter the number of states that your DFA consists of

3

Enter the number of input symbol that DFA have

2

Enter the input symbols 2

1 input 0

2 input 1

enter number of final states 1

Final state 1 : q2

---

define transition rule as (Initial State, Input Symbol) = Final State

(q0 , 0) = q2

(q1 , 0) = q2

(q2 , 0) = q2

(q0 , 1) = q1

(q1 , 1) = q1

(q2 , 1) = q1

Enter Input String 01110

Valid String

/****** Output for DFA with 3 states for rejection of string*********/

Enter the number of states that your DFA consists of

3

Enter the number of input symbol that DFA have

2

Enter the input symbols 2

1 input 0

2 input 1

enter number of final states 1

Final state 1 : q2

---

define transition rule as (Initial State, Input Symbol) = Final State

(q0 , 0) = q2

(q1 , 0) = q2

(q2 , 0) = q2

(q0 , 1) = q1

(q1 , 1) = q1

(q2 , 1) = q1

Enter Input String 111101

Invalid String