1. Write a program to break the input (from File) into lexemes.

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

int main() {
    char str[100];
    char result[10][10];
    int i, j, cnt;
    FILE *f;

    // Open the file for reading
    f = fopen("input.txt", "r");
    if (f == NULL) {
        printf("Error opening file.\n");
        return 1;
    }

    // Read the string from the file
    fgets(str, sizeof(str), f);
    fclose(f);

    j = 0;
    cnt = 0;
    for (i = 0; i <= (strlen(str)); i++) {
        if (str[i] == ' ' || str[i] == '\0') {
            result[cnt][j] = '\0';
            cnt++;
            j = 0;
        }
        else {
            result[cnt][j] = str[i];
            j++;
        }
    }
    for (i = 0; i < cnt; i++)
        printf("%s\n", result[i]);

    return 0;
}
```

2. Write a program to count vowels and consonants in the given input.

```
#include <stdio.h>

int main() {
    char s[100];
    printf("Enter the string:");
    scanf("%[^\n]s",&s);
    int i,vowels=0,consonants=0;
    for(i=0;i<strlen(s);i++)
    {
        if(isalpha(s[i]))
        {
            if(s[i]=='a'||s[i]=='e'||s[i]=='i'||s[i]=='o'||s[i]=='u')
            {
                vowels+=1;
            }
            else
            {
                consonants+=1;
            }
        }
    }
    printf("Number of vowels:%d\n",vowels);
    printf("Number of consonants:%d\n",consonants);
}
```

3. Write a program to implement the scanner application without Lex tool (Recognition of Lexemes and Tokens).

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

int main() {
   char s[100],x[20][20];
   printf("Enter the string:");
   scanf("%[^\n]s",&s);
   int i,n=0,k=0;
   for(i=0;i<strlen(s);i++)
   {
      if(s[i]==' ')
      {
         x[n][k]='\0';
         n+=1;k=0;
      }
      else
      {
         x[n][k]=s[i];
         k++;
      }
   }
   for(i=0;i<n+1;i++)
   {
      printf("%s\n",x[i]);
   }
   printf("*************************************************\n");
   int j,dig,spec;
   for(i=0;i<n+1;i++)
   {

if(!strcmp(x[i],"int")||!strcmp(x[i],"char")||!strcmp(x[i],"for")||!strcmp(x[i],"if")||!strcmp(x[i],"while")
||!strcmp(x[i],"else"))
      {
         printf("%s\tKeyword\n",x[i]);
      }
      else
      {
         dig=0;
         spec=0;
         for(j=0;j<strlen(x[i]);j++)
         {
            if(isalnum(x[i][j]))
            {
               if(isdigit(x[i][j]))
               {
```

```
                        dig+=1;
                    }
                }
                else
                {
                    spec+=1;
                }
            }
            if(dig==strlen(x[i]))
            {
                printf("%s\tNumber\n",x[i]);
            }
            else if(dig<strlen(x[i])&&spec==0)
            {
                printf("%s\tIdentifier\n",x[i]);
            }
            else if(spec==strlen(x[i]))
            {
                printf("%s\tSpecial symbols\n",x[i]);
            }
        }
    }


    return 0;
}
```

4. Write detailed description about Compiler, Interpreter, Loader, Linker, Assembler etc.

5. Write detailed description about Lex, Flex, YACC, Bison.

……………..

6. Write a program to identify the Octal or Hexadecimal number using Lex tool.

```
%{

%}
Oct [0][0-9]+
Hex [0][x|X][0-9A-F]+

%%
{Hex} printf("this is a hexadecimal number");
{Oct} printf("this is an octal number");
%%

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

7. Write a program to capitalize input string using Lex tool.

```
%{
#include<stdio.h>
#include<ctype.h>
int k;
void display(char *);
%}
letter [a-z]
%%
{letter} {display(yytext);}
%%
main()
{
yylex();
}
void display(char *s)
{
int i;
for(i=0;s[i]!='\0';i++)
printf("%c",toupper(s[i]));
}
int yywrap()
{
return 1;
}
```

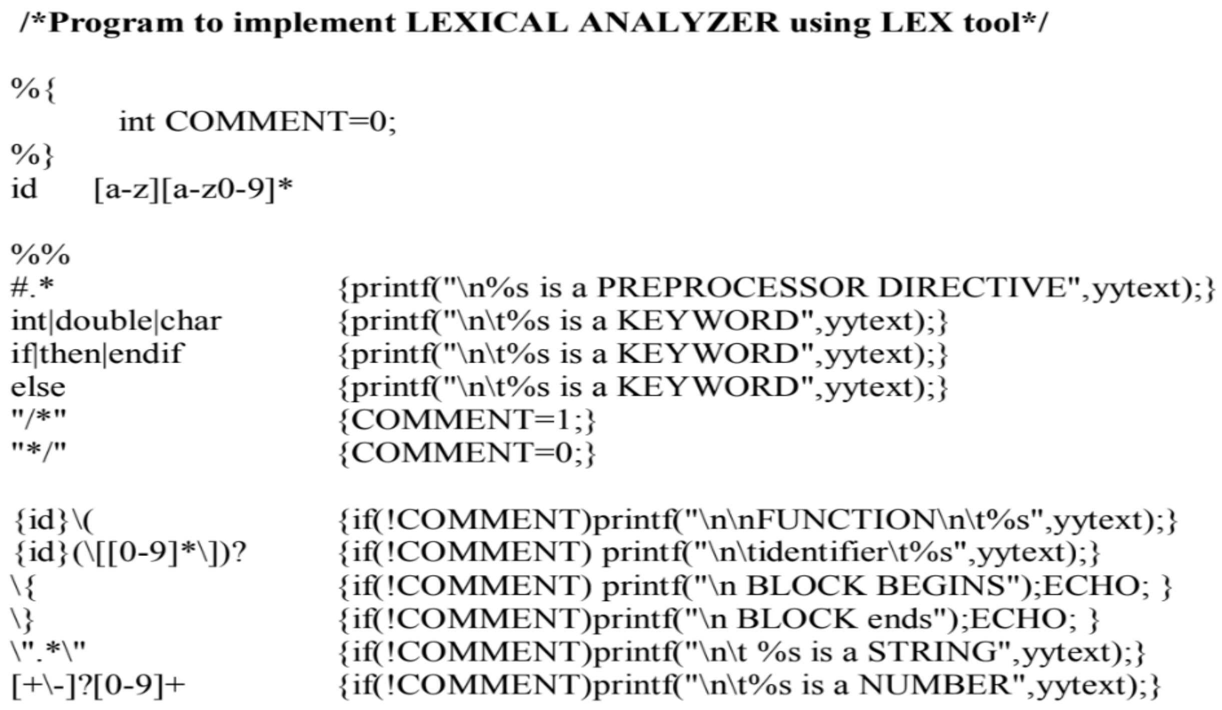
Other solution:

```
%{
%}

lower [a-z]

%%

{lower} printf("%c",yytext[0]-32);

%%
main ()
{
        yylex();
}

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

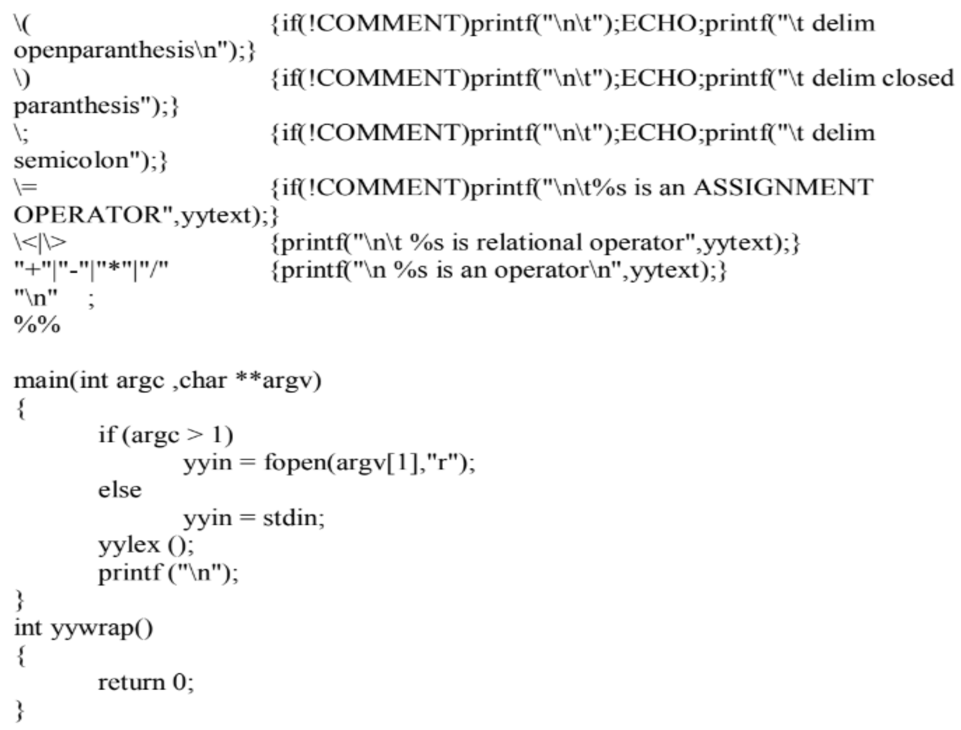
8. Write a program to identify integer or real number using Lex tool.

```
%{

%}
Integ [0-9]+
Real [0-9]+.[0-9]+

%%
{Integ} printf("this is integer");
{Real} printf("this is a real number");
%%

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

9. Write a program to implement scanner application using Lex Tool. (Lexcial Analysis Process, recognition of lexeme, tokens).



```
/*Program to implement LEXICAL ANALYZER using LEX tool*/

%{
        int COMMENT=0;
%}
id      [a-z][a-z0-9]*

%%
#.*                     {printf("\n%s is a PREPROCESSOR DIRECTIVE",yytext);}
int|double|char         {printf("\n\t%s is a KEYWORD",yytext);}
if|then|endif           {printf("\n\t%s is a KEYWORD",yytext);}
else                    {printf("\n\t%s is a KEYWORD",yytext);}
"/*"                    {COMMENT=1;}
"*/"                    {COMMENT=0;}

{id}\(                  {if(!COMMENT)printf("\n\nFUNCTION\n\t%s",yytext);}
{id}(\[[0-9]*\])?       {if(!COMMENT) printf("\n\tidentifier\t%s",yytext);}
\{                      {if(!COMMENT) printf("\n BLOCK BEGINS");ECHO; }
\}                      {if(!COMMENT)printf("\n BLOCK ends");ECHO; }
\".*\"                  {if(!COMMENT)printf("\n\t %s is a STRING",yytext);}
[+\-]?[0-9]+            {if(!COMMENT)printf("\n\t%s is a NUMBER",yytext);}
```



```
\(                      {if(!COMMENT)printf("\n\t");ECHO;printf("\t delim
openparanthesis\n");}
\)                      {if(!COMMENT)printf("\n\t");ECHO;printf("\t delim closed
paranthesis");}
\;                      {if(!COMMENT)printf("\n\t");ECHO;printf("\t delim
semicolon");}
\=                      {if(!COMMENT)printf("\n\t%s is an ASSIGNMENT
OPERATOR",yytext);}
\<|\>                   {printf("\n\t %s is relational operator",yytext);}
"+"|"-"|"*"|"/"         {printf("\n %s is an operator\n",yytext);}
"\n"   ;
%%

main(int argc ,char **argv)
{
        if (argc > 1)
                yyin = fopen(argv[1],"r");
        else
                yyin = stdin;
        yylex ();
        printf ("\n");
}
int yywrap()
{
        return 0;
}
```

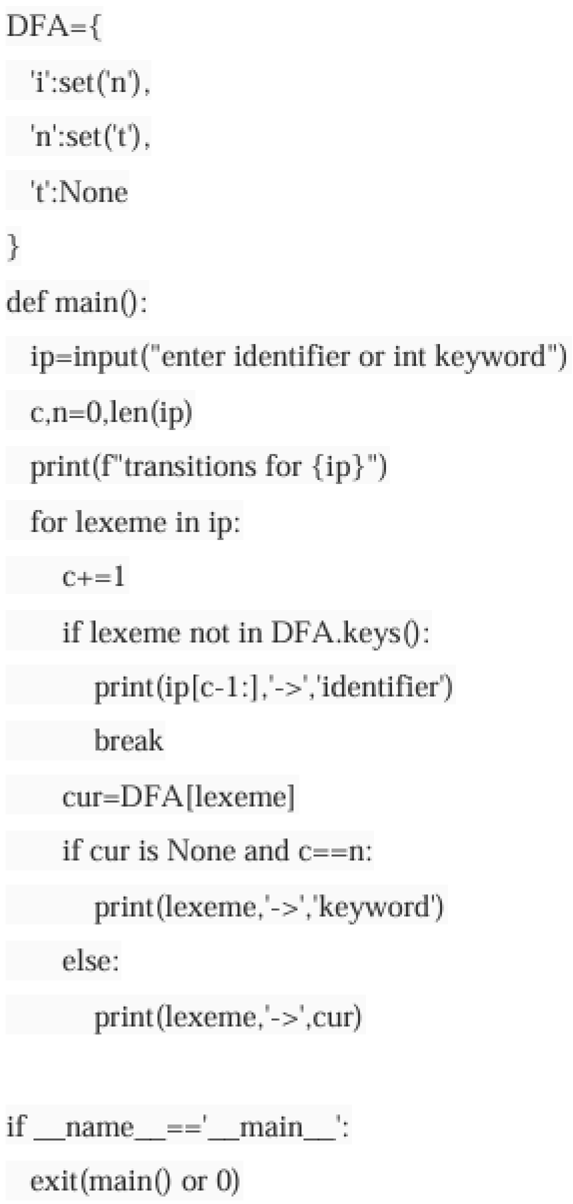
10. Write program to find number of characters, spaces, lines, tabs in a given file using Lex tool.

```
%{
int lines=0,space=0,tabs=0,chars=0;
%}

%%
\n lines++;
" " space++;
\t tabs++;
[^\t" "\n] chars++;
. ;
%%
int main(void)
{
yyin=fopen("textfile.txt","r");
yylex();
printf("Number of lines :  %d\n",lines);
printf("Number of spaces  :  %d\n",space);
printf("Number of tab :  %d\n",tabs);
printf("Number of character :  %d\n",chars);
return 0;
}

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

11. Write program to find number of characters, spaces, lines, tabs in a given file without using Lex tool.

```
#include<stdio.h>
int main()
{
int spaces,chars,lines,tabs;
char i;
spaces=0;
chars=0;
lines=0;
tabs=0;
FILE *f;
f=fopen("textfile.txt","r");
while(!feof(f))
{
i=fgetc(f);
if(i=='\n')
{
lines+=1;
}
else if(i=='\t')
{
tabs+=1;
}
else if(i==' ')
{
spaces+=1;
}
else
{
chars+=1;
}
}
fclose(f);
printf("Lines:%d\n",lines);
printf("Tabs:%d\n",tabs);
printf("Spaces:%d\n",spaces);
printf("Characters:%d\n",chars);
}
```

12. Write a program to demonstrate tokenization – By constructing DFA of Lexical Analyzer (Lex tool).



```
DFA={
  'i':set('n'),
  'n':set('t'),
  't':None
}
def main():
  ip=input("enter identifier or int keyword")
  c,n=0,len(ip)
  print(f"transitions for {ip}")
  for lexeme in ip:
      c+=1
      if lexeme not in DFA.keys():
          print(ip[c-1:],'->','identifier')
          break
      cur=DFA[lexeme]
      if cur is None and c==n:
          print(lexeme,'->','keyword')
      else:
          print(lexeme,'->',cur)

if __name__=='__main__':
  exit(main() or 0)
```

13. Write a lex program to count no of words in a given input.

```
%{
int words=0;
%}
%%
[a-z0-9A-Z]* {words++;};

%%

int main ()
{
        yylex();
        printf("%d",words);
}

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

……………..

14. Write a lex program to design a simple calculator.

```
%{
int op = 0,i;
float a, b;
%}

dig [0-9]+|([0-9]*)"."([0-9]+)

%%

{dig} {digi();}
"+" {op=1;}
"-" {op=2;}
"*" {op=3;}
"/" {op=4;}
\n {printf("\n The Answer :%f\n\n",a);}

%%
digi()
{
if(op==0)
a=atof(yytext);

else
{
b=atof(yytext);

switch(op)
{
case 1:a=a+b;
   break;

case 2:a=a-b;
break;

case 3:a=a*b;
break;

case 4:a=a/b;
break;

}
op=0;
}
}

main()
```

```
{
yylex();
}

yywrap()
{
return 1;
}
```

15. Write a lex program to count no of ‘scanf’ and ‘printf’ statements in a given C program as input.

```
%{
int p=0,s=0,o=0;
%}
%%
scanf {s++;}
printf {p++;}

%%
int main(void)
{
yyin=fopen("textfile.txt","r");
yylex();
printf("Number of printfs :  %d\n",p);
printf("Number of scanfs  :  %d\n",s);
return 0;
}
int yywrap()
{return 1;}
```

16. Write a lex program to recognize and count the number of identifiers in a given input file.

```
%{
int ids=0,digits=0,keywords=0,sp=0,ops=0;
%}
%%
[0-9]+ {digits++;}
int|float|while|for|if|else {keywords++;}
[(,),;,},{,",',#,<,>,.,!,:] {sp++;}
[+,-,*,=,%] {ops++;}
[a-zA-z][0-9]*[a-zA-Z]* {ids++;}
%%
int main(void)
{
yyin=fopen("textfile.txt","r");
yylex();
printf("Number of identifiers:  %d\n",ids);
return 0;
}
int yywrap()
{return 1;}
```

....

17. Implementing Recursive Decent Parser for a grammar

S->AA
A->aB/ϵ
B->b

```
#include<stdio.h>
#include <stdlib.h>
char l;

void match(char c)
  {
        if(l==c)
            l=getchar();
        else
          {
            printf("Invalid Input\n");
            exit(0);
          }
  }

void B()
 {
    if(l=='b')
      {
        match('b');
      }
    else
      {
        printf("Invalid Input\n");
        exit(0);
      }
 }

void A()
 {
   if(l=='a')
     {
       match('a');
       B();
     }
   else
       return;
 }

void S()
 {
   A();
   A();
 }

void main()
{
   char input[10];
```

```
    printf("Enter String with $ at the end\n");
    l=getchar();
    S();
    if(l=='$')
        {
            printf("\nParsing Successful\n");
        }
    else
      {
        printf("Invalid Input\n");
      }

}
```

18. Program to generate predictive LL1 parsing table for the grammar

19. Program to find FIRST function of a grammar

```
def first(rule):
    global rules, nonterm_userdef, \
        term_userdef, diction, firsts
    if len(rule) != 0 and (rule is not None):
        if rule[0] in term_userdef:
            return rule[0]
        elif rule[0] == '#':
            return '#'

    if len(rule) != 0:
        if rule[0] in list(diction.keys()):
            fres = []
            rhs_rules = diction[rule[0]]
            for itr in rhs_rules:
                indivRes = first(itr)
                if type(indivRes) is list:
                    for i in indivRes:
                        fres.append(i)
                else:
                    fres.append(indivRes)
            if '#' not in fres:
                return fres
            else:
                newList = []
                fres.remove('#')
                if len(rule) > 1:
                    ansNew = first(rule[1:])
                    if ansNew != None:
                        if type(ansNew) is list:
                            newList = fres + ansNew
                        else:
                            newList = fres + [ansNew]
                    else:
                        newList = fres
                    return newList
                fres.append('#')
                return fres
```

```
def computeAllFirsts():
    global rules, nonterm_userdef,
        term_userdef, diction, firsts
    for rule in rules:
        k = rule.split("->")
        k[0] = k[0].strip()
        k[1] = k[1].strip()
        rhs = k[1]
        multirhs = rhs.split('|')
        for i in range(len(multirhs)):
            multirhs[i] = multirhs[i].strip()
            multirhs[i] = multirhs[i].split()
        diction[k[0]] = multirhs
    print(f"\nRules: \n")
    for y in diction:
        print(f"{y}->{diction[y]}")

    for y in list(diction.keys()):
        t = set()
        for sub in diction.get(y):
            res = first(sub)
            if res != None:
                if type(res) is list:
                    for u in res:
                        t.add(u)
                else:
                    t.add(res)
        firsts[y] = t
    print("\nCalculated firsts: ")
    key_list = list(firsts.keys())
    index = 0
    for gg in firsts:
        print(f"first({key_list[index]}) "
            f"=> {firsts.get(gg)}")
        index += 1

rules=["S -> A | B C",
    "A -> a | b",
    "B -> p | #",
    "C -> c"]
```

```
nonterm_userdef=['A','S','B','C']
term_userdef=['a','c','b','p']

diction = {}
firsts = {}
computeAllFirsts()
```

-20. Program to find FOLLOW function of a grammar

```
def first(rule):
    global rules, nonterm_userdef, \
        term_userdef, diction, firsts
    if len(rule) != 0 and (rule is not None):
        if rule[0] in term_userdef:
            return rule[0]
        elif rule[0] == '#':
            return '#'

    if len(rule) != 0:
        if rule[0] in list(diction.keys()):
            fres = []
            rhs_rules = diction[rule[0]]
            for itr in rhs_rules:
                indivRes = first(itr)
                if type(indivRes) is list:
                    for i in indivRes:
                        fres.append(i)
                else:
                    fres.append(indivRes)
            if '#' not in fres:
                return fres
            else:
                newList = []
                fres.remove('#')
                if len(rule) > 1:
                    ansNew = first(rule[1:])
                    if ansNew != None:
                        if type(ansNew) is list:
                            newList = fres + ansNew
                        else:
                            newList = fres + [ansNew]
                    else:
                        newList = fres
                    return newList
                fres.append('#')
                return fres

def follow(nt):
    global start_symbol, rules, nonterm_userdef, \
        term_userdef, diction, firsts, follows

    solset = set()
    if nt == start_symbol:
        solset.add('$')

    for curNT in diction:
        rhs = diction[curNT]
        for subrule in rhs:
            if nt in subrule:
                while nt in subrule:
                    index_nt = subrule.index(nt)
                    subrule = subrule[index_nt + 1:]
                    if len(subrule) != 0:
                        res = first(subrule)
                        if '#' in res:
                            newList = []
                            res.remove('#')
                            ansNew = follow(curNT)
                            if ansNew != None:
                                if type(ansNew) is list:
                                    newList = res + ansNew
                                else:
                                    newList = res + [ansNew]
                            else:
                                newList = res
                            res = newList
                    else:
                        if nt != curNT:
                            res = follow(curNT)

                    if res is not None:
                        if type(res) is list:
                            for g in res:
                                solset.add(g)
                        else:
                            solset.add(res)
    return list(solset)


def computeAllFirsts():
    global rules, nonterm_userdef, \
        term_userdef, diction, firsts
    for rule in rules:
        k = rule.split("->")
        k[0] = k[0].strip()
        k[1] = k[1].strip()
        rhs = k[1]
        multirhs = rhs.split('|')
        for i in range(len(multirhs)):
            multirhs[i] = multirhs[i].strip()
            multirhs[i] = multirhs[i].split()
        diction[k[0]] = multirhs

    print(f"\nRules: \n")
    for y in diction:
        print(f"{y}->{diction[y]}")

    for y in list(diction.keys()):
        t = set()
        for sub in diction.get(y):
            res = first(sub)
            if res != None:
                if type(res) is list:
                    for u in res:
                        t.add(u)
                else:
                    t.add(res)

        firsts[y] = t

    print("\nCalculated firsts: ")
    key_list = list(firsts.keys())
    index = 0
    for gg in firsts:
        print(f"first({key_list[index]}) "
            f"=> {firsts.get(gg)}")
        index += 1


def computeAllFollows():
    global start_symbol, rules, nonterm_userdef,\
                term_userdef, diction, firsts, follows
        for NT in diction:
                solset = set()
                sol = follow(NT)
                if sol is not None:
                        for g in sol:
                                solset.add(g)
                follows[NT] = solset

        print("\nCalculated follows: ")
        key_list = list(follows.keys())
        index = 0
        for gg in follows:
                print(f"follow({key_list[index]})"
                        f" => {follows[gg]}")
                index += 1

rules=["S -> A | B C",
         "A -> a | b",
         "B -> p | #",
         "C -> c"]
nonterm_userdef=['A','S','B','C']
term_userdef=['a','c','b','p']

diction = {}
firsts = {}
follows = {}

computeAllFirsts()
start_symbol = list(diction.keys())[0]
computeAllFollows()
```