

## EXPERIMENT NUMBER – 1

**AIM:** To find the number of lines, words, characters in a given text.

**DESCRIPTION:** In this program, we try to find the number of lines, words and characters present in the given text file

### ALGORITHM:

Step-1: Start

Step-2: Define a function count(fname):

    Step-2.1: num\_line->0

    Step-2.2: num\_words->0

    Step-2.3: num\_char->0

    Step-2.4: with open(fname, 'r') as f:

        Step-2.4.1: for line in f:

            Step-2.4.1.1: num\_line+=1

            Step-2.4.1.2: w=line.split()

            Step-2.4.1.3: num\_words -> num\_words+len(w)

            Step-2.4.1.4: for l in line:

                Step-2.4.1.4.1: for i in l: if (i!=" "): num\_char +=1

    Step-2.5: Print num\_line

    Step-2.6: Print num\_words

    Step-2.7: Print num\_char

Step-3: Read the file name from the user

Step-4: try:

    Count(fname)

Step-5: except:

    Print "File not Found"

Step-6: End

### PROGRAM:

**program1.py file:**

```
def count(fname):
    num_line = 0
    num_words = 0
    num_char = 0
    with open(fname, 'r') as f:
        for line in f:
            num_line += 1
            w = line.split()
            num_words = num_words + len(w)
            for l in line:
                for i in l:
                    if (i != ' '):
                        num_char += 1
    print("Number of lines in text file: ", num_line)
    print("Number of words in text file: ", num_words)
```

```
print('Number of characters in text file: ', num_char)

if __name__ == '__main__':
    fname = input("Enter the file name: ")
    try:
        count(fname)
    except:
        print('File not found')
```

**demo.txt file:**

Compiler Design Lab

Write a Program to find number of lines, words, characters in a text

**OUTPUT:**

```
==== RESTART: D:\Engineering\SEM-6\Compiler Design\Lab Work\Week-1\Progl.py ====
Enter the file name: Demo.txt
Number of lines in text file:  2
Number of words in text file:  16
Number of characters in text file:  74
```

**CONCLUSION:**

By executing the above program, we have successfully found number of lines, words and characters in a given text file

## EXPERIMENT NUMBER – 2

**AIM:** To display the number of tokens in a given file

**DESCRIPTION:** In this program, we try to display number of tokens and the type of token present in a given file

**ALGORITHM:**

Step-1: Start

Step-2: Define a function tokens(fname)

Step-2.1: num\_tokens->0

Step-2.2: keywords = ['int', 'float', 'char', 'boolean', 'double', 'def', 'if', 'while', 'with']

Step-2.3: operators = ['=', '+', '-', '==', '\*', '/', '%', '!=', '\*\*']

Step-2.4: Special = [' ', '(', ')', ';', ':', '[', ']', '&']

Step-2.5: with open (fname, 'r') as f:

Step-2.5.1: for line in f:

Step-2.5.1.1: w -> line.split()

Step-2.5.1.2: key->'N'

Step-2.5.1.3: num\_token->num\_token+len(w)

Step-2.5.1.4: for i in w:

Step-2.5.1.4.1: if ( i in Special):

Print 'Special Characters'

key='N'

Step-2.5.1.4.2: else if ( i in keywords):

Print 'Keywords'

key='N'

Step-2.5.1.4.3: else if ( i.isdigit()):

Print 'Constant'

key='N'

Step-2.5.1.4.4: else if ( i in operators):

Print 'Operator'

key='N'

Step-2.5.1.4.5: else:

if (key=='Y'):

Print "Identifier"

key='N'

else:

Print "String"

Step-2.6: Print num\_token

Step-3: Read the file name from the user

Step-4: try:

Count(fname)

Step-5: except:

Print "File not Found"

Step-6: End

**PROGRAM:**

**program2.py file:**

```
def tokens(fname):
    num_token = 0
    keywords = ['int', 'float', 'char', 'boolean', 'double', 'def', 'if', 'while', 'with']
    operators = ['=', '+', '-', '==', '*', '/', '%', '!=', '**']
    Special = [',', '(', ')', ';', ':', '[', ']', '&']
    with open(fname, 'r') as f:
        for line in f:
            w = line.split()
            key = 'N'
            num_token = num_token + len(w)
            for i in w:
                if (i in Special):
                    print(i, ": Special Character")
                    key = 'N'
                elif (i in keywords):
                    print(i, ": Keyword")
                    key = 'Y'
                elif (i.isdigit()):
                    print(i, ": Constant")
                    key = 'N'
                elif (i in operators):
                    print(i, ": Operator")
                    key = 'N'
                else:
                    if (key == 'Y'):
                        print(i, ": Identifier")
                        key = 'N'
                    else:
                        print(i, ": String")
            print("Number of Tokens in text file: ", num_token)

if __name__ == '__main__':
    fname = input("Enter the file name: ")
    try:
        tokens(fname)
    except:
        print("File not found")
```

**file.txt file:**

Compiler Design Lab

int a = 1

Write a Program to find number of lines , words , characters in a text

int abc = nikhitha

### OUTPUT:

```
== RESTART: D:\Engineering\SEM-6\Compiler Design\Lab Work\Week-1\Program-2.py ==
Enter the file name: file.txt
Compiler : String
Design : String
Lab : String
int : Keyword
a : Identifier
= : Operator
1 : Constant
Write : String
a : String
Program : String
to : String
find : String
number : String
of : String
lines : String
, : Special Character
words : String
, : Special Character
characters : String
in : String
a : String
text : String
int : Keyword
abc : Identifier
= : Operator
nikhitha : String
Number of Tokens in text file:  26
```

### CONCLUSION:

By executing the above program, we have successfully displayed the number of tokens and type of tokens present in a file

## EXPERIMENT NUMBER – 3

**AIM:** To identify whether an alphabet is vowel or consonant and also display its count using lex tool

**DESCRIPTION:** In this program, we try to take the input from the user and identify the vowels and consonants and also display its count in the given string.

### ALGORITHM:

Step-1: Start

Step-2: Declare the variables vow->0 and const\_count->0

Step-3: Declare the regular expressions

[aeiouAEIOU] {vow++;}

[a-zA-Z] {const\_count++;}

Step-4: Read the string from the user

Step-5: Print vow

Step-6: Print const\_count

Step-7: End

### PROGRAM:

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

%%
[aeiouAEIOU] {vow++;}
[a-zA-Z] {const_count++;}
%%

int yywrap(){
    return 1;
}

int main(){
    printf("Enter the string: ");
    yylex();
    printf("no. of vowels are: %d\n",vow);
    printf("no. of consonants are %d",const_count);
    return 0;
}
```

**OUTPUT:**

```
student@CSELab3-06:~$ gedit vowels.l
student@CSELab3-06:~$ lex vowels.l
student@CSELab3-06:~$ gcc lex.yy.c
student@CSELab3-06:~$ ./a.out
Enter the string: Nikhitha

no. of vowels are: 3
no. of consonants are 5student@CSELab3-06:~$
```

**CONCLUSION:**

By executing the above program, we have successfully identified whether an alphabet is vowel or consonant and also displayed its count in given string

## EXPERIMENT NUMBER – 4

**AIM:** To identify integer or real number using lex tool

**DESCRIPTION:** In this program, we try to identify whether the number given by the user is integer or real number

**ALGORITHM:**

Step-1: Start

Step-2: Declare the regular expressions

integer [0-9]+

float [0-9]+\.[0-9]+

Step-3: {integer} Print "Integer"

Step-4: {float} Print "Real Number"

Step-5: End

**PROGRAM:**

```
%{  
    #include<stdio.h>  
}%  
  
integer [0-9]+  
float [0-9]+\.[0-9]+  
%%  
  
{integer} printf("This is an integer");  
{float} printf("This is a real number");  
%%  
int main(){  
    yylex();  
}  
int yywrap(){  
    return 1;  
}
```



**OUTPUT:**

```
student@CSELab3-06:~$ gedit intorreal.l
student@CSELab3-06:~$ lex intorreal.l
student@CSELab3-06:~$ gcc lex.yy.c
student@CSELab3-06:~$ ./a.out
4
This is an integer
5.8
This is a real number
^C
```

**CONCLUSION:**

By executing the above program, we have successfully identified whether the number given by the user is integer or real number

## EXPERIMENT NUMBER – 5

**AIM:** To capitalize the character of a string using lex tool

**DESCRIPTION:** In this program, we try to capitalize every character of a string given by the user using lex tool

**ALGORITHM:**

Step-1: Start

Step-2: Declare the regular expression  
lower [a-z]

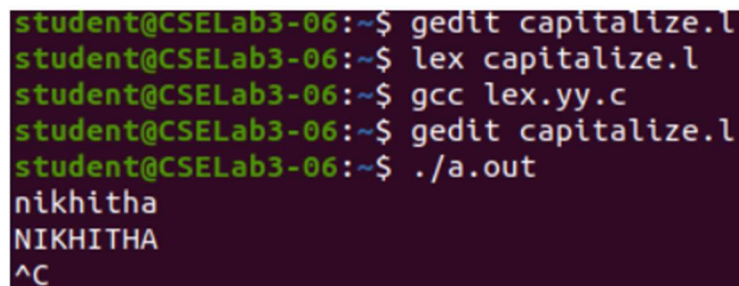
Step-3: {lower} print yytext[0]-32

Step-4: End

**PROGRAM:**

```
%{
%}
lower [a-z]
%%
{lower} printf("%c",yytext[0]-32);
%%
int main(){
yylex();
}
int yywrap(){
return 1;
}
```

**OUTPUT:**



```
student@CSELab3-06:~$ gedit capitalize.l
student@CSELab3-06:~$ lex capitalize.l
student@CSELab3-06:~$ gcc lex.yy.c
student@CSELab3-06:~$ gedit capitalize.l
student@CSELab3-06:~$ ./a.out
nikhitha
NIKHITHA
^C
```

**CONCLUSION:**

By executing the above program, we have successfully capitalized every character of a string given by the user

## EXPERIMENT NUMBER – 6

**AIM:** To implement scanner or lexical analyzer without using lex tool

**DESCRIPTION:** In this program, we try to scanner or lexical analyzer without using the lex tool

### ALGORITHM:

Step-1: Start

Step-2: Define a function tokens(fname)

Step-2.1: num\_tokens->0, number->0

Step-2.2: keywords = ['int', 'float', 'char', 'boolean', 'double', 'def', 'if', 'while', 'with']

Step-2.3: operators = ['=', '+', '-', '==', '\*', '/', '%', '!=', '\*\*']

Step-2.4: Special = [' ', '(', ')', ';', ':', '[', ']', '&']

Step-2.5: with open (fname, 'r') as f:

Step-2.5.1: for line in f:

Step-2.5.1.1: w -> line.split()

Step-2.5.1.2: key->'N'

Step-2.5.1.3: num\_token->num\_token+len(w)

Step-2.5.1.4: for i in w:

Step-2.5.1.4.1: if ( i in Special):

Print number

Print 'Special Characters'

key='N'

Step-2.5.1.4.2: else if ( i in keywords):

Print number

Print 'Keywords'

key='N'

Step-2.5.1.4.3: else if ( i.isdigit()):

Print number

Print 'Constant'

key='N'

Step-2.5.1.4.4: else if ( i in operators):

Print number

Print 'Operator'

key='N'

Step-2.5.1.4.5: else:

if (key=='Y'):

Print number

Print "Identifier"

key='N'

else:

Print number

Print "String"

Step-2.5.1.5: number->number+1

Step-2.6: Print num\_token

Step-3: Read the file name from the user

Step-4: try:

Count(fname)

Step-5: except:

Print "File not Found"

Step-6: End

### PROGRAM:

#### program-1.py file:

```
def tokens(fname):
    num_token = 0
    number=1
    keywords = ['int', 'float', 'char', 'boolean', 'double', 'def', 'if', 'while', 'with']
    operators = ['=', '+', '-', '==', '*', '/', '%', '!=', '**']
    Special = [',', '(', ')', ';', ':', '[', ']', '&']
    print("Line No  lexeme  token")
    with open(fname,'r') as f:
        for line in f:
            w = line.split()
            key = 'N'
            num_token = num_token + len(w)
            for i in w:
                if (i in Special):
                    print(number, ' ', i, "  Special Character")
                    key = 'N'
                elif (i in keywords):
                    print(number, ' ', i, "  Keyword")
                    key = 'Y'
                elif (i.isdigit()):
                    print(number, ' ', i, "  Constant")
                    key = 'N'
                elif (i in operators):
                    print(number, ' ', i, "  Operator")
                    key = 'N'
                else:
                    if (key == 'Y'):
                        print(number, ' ', i, "  Identifier")
                        key = 'N'
                    else:
                        print(number, ' ', i, "  String")
            number=number+1
    print("Number of Tokens in text file: ", num_token)

if __name__ == '__main__':
    fname = input("Enter the file name: ")
    try:
        tokens(fname)
    except:
```

```
print("File not found")
```

**demo.txt:**

```
int l = 1
```

```
nikhitha @
```

**OUTPUT:**

```
== RESTART: D:/Engineering/SEM-6/Compiler Design/Lab Work/Week-2/Program-1.py ==
Enter the file name: demo.txt
Line No    lexeme    token
1          int      Keyword
1          1       Identifier
1          =       Operator
1          1       Constant
2          nikhitha String
2          @       String
Number of Tokens in text file: 6
```

**CONCLUSION:**

By executing the above program, we have successfully implemented scanner or lexical analyzer without using lex tool

## EXPERIMENT NUMBER – 7

**AIM:** To identify octal or hexadecimal number using lex tool

**DESCRIPTION:** In this program, we try to identify whether the given number is octal or hexadecimal number using lex tool

**ALGORITHM:**

Step-1: Start

Step-2: Declare the regular expressions

Oct[o][0-7]+

Hex[0][x][X][0-9 A-F]+

Step-3: {Hex} Print 'Hexadecimal number'

Step-4: {Oct} Print 'Octal number'

Step-5: End

**PROGRAM:**

```
%{
    /*program*/
}%
Oct[o][0-7]+
Hex[0][x][X][0-9 A-F]+
%%
{Hex} printf("This is Hexadecimal number");
{Oct} printf("This is an Octal number");
%%
main()
{
    yylex();
}
int yywrap()
{
    return 1;
}
```

### OUTPUT:

```
student@CSELab3-12:~$ gedit hex.l
student@CSELab3-12:~$ lex hex.l
student@CSELab3-12:~$ gcc lex.yy.c
hex.l:10:1: warning: return type defaults to 'int' [-Wimplicit-int]
    10 | main()
        | ~~~~~
student@CSELab3-12:~$ ./a.out
o32
This is an Octal number
0xX4F
This is Hexadecimal number
```

### CONCLUSION:

By executing the above program, we have successfully identified whether the given number is octal or hexadecimal number.

## EXPERIMENT NUMBER – 8

**AIM:** To accept the words starting with A or a using lex tool

**DESCRIPTION:** In this program, we try to accept the words starting with A or a using lex tool

### ALGORITHM:

Step-1: Start

Step-2: Declare the regular expressions

A [A][a-z A-Z]+

a [a][a-z A-Z]+

b [^A][a-z A-Z]+

c [^a][a-z A-Z]+

Step-3: {A} print 'Accepted'

Step-4: {a} print 'Accepted'

Step-5: {b} print 'Not accepted'

Step-6: {c} print 'Not accepted'

Step-7: End

### PROGRAM:

```
%{
/* Lex Program to accept string starting with vowel */
%}
A [A][a-z A-Z]+
a [a][a-z A-Z]+
b [^A][a-z A-Z]+
c [^a][a-z A-Z]+
%%
{A} printf("Accepted");
{a} printf("Accepted");
{b} printf("Not accepted");
{c} printf("Not accepted");
%%
main()
{
yylex();
}
int yywrap(){
return 1;
}
```



### OUTPUT:

```
student@CSELab3-12:~$ gedit A.l
student@CSELab3-12:~$ lex A.l
student@CSELab3-12:~$ gcc lex.yy.c
A.l:14:1: warning: return type defaults to 'int' [-Wimplicit-int]
   14 | main()
      | ^~~~~
student@CSELab3-12:~$ ./a.out
Asdf
Accepted^C
student@CSELab3-12:~$ ./a.out
ghjk
Not accepted
```

### CONCLUSION:

By executing the above program, we have successfully accepted the words starting with A or a using lex tool

## EXPERIMENT NUMBER – 9

**AIM:** To design token separator for the given file using lex tool

**DESCRIPTION:** In this program, we try to design a token separator for the given file using lex tool

**ALGORITHM:**

Step-1: Start

Step-2: l = 1

Step-3: Declare the regular expressions

delim [ \t\b]

ws {delim}\*

ident [A-Za-z][A-Za-z0-9]\*

op [\+\-\\*/%]=

special [;\{\}\[\]\(\)<>]

Step-4: {ws} print 'Keyword'

Step-5: {ident} print 'Identifier'

Step-6: {op} print 'Operator'

Step-7: {special} print 'Special'

Step-8: l++

Step-9: yyin -> fopen("sample.c", 'r')

Step-10: End

**PROGRAM:**

```
%{
#include<stdio.h>
int l = 1;
%}
delim [ \t\b]
ws {delim}*
ident [A-Za-z][A-Za-z0-9]*
op [\+\-\*/%]=
special [;\{\}\[\]\(\)<>]
%%
{ws}{int|return|include} { printf("%d\t\t%s\t\t\t\tKeyword\n", l, yytext); }
{ident} { printf("%d\t\t%s\t\t\t\tIdentifier\n", l, yytext); }
{op} { printf("%d\t\t%s\t\t\t\tOperator\n", l, yytext); }
{special} { printf("%d\t\t%s\t\t\t\tSpecial\n", l, yytext); }
\n { l++; }
. {}
%%
int main() {
    extern FILE *yyin;
    printf("LineNumber\tLexeme\tToken\n");
    yyin = fopen("sample.c", "r");
    yylex();
```

```
    return 0;
}
int yywrap(){
    return 1;
}
```

**OUTPUT:**

```
student@CSELab3-12:~$ gedit parse.l
student@CSELab3-12:~$ lex parse.l
student@CSELab3-12:~$ gcc lex.yy.c
student@CSELab3-12:~$ ./a.out
LineNumber      Lexme           Token
1      "include"      Keyword
1      "<"            Special
1      "stdio"        Identifier
1      "h"            Identifier
1      ">"            Special
2      "int"          Keyword
2      "main"         Identifier
2      "("            Special
2      ")"            Special
2      "{"            Special
3      "    int"      Keyword
3      "num"          Identifier
3      ";"            Special
4      "printf"       Identifier
4      "("            Special
4      "Enter"        Identifier
4      "an"           Identifier
4      " int"         Keyword
4      "eger"         Identifier
4      ")"            Special
4      ";"            Special
5      "scanf"        Identifier
5      "("            Special
5      "%"            Operator
5      "d"            Identifier
5      "num"          Identifier
5      ")"            Special
5      ";"            Special
7      "if"           Identifier
```

**CONCLUSION:**

By executing the above program, we have successfully designed a token separator for the given file using the lex tool

## EXPERIMENT NUMBER – 10

**AIM:** To implement FIRST Function for a given grammar

**DESCRIPTION:** In this program, we try to find the FIRST Functions of every non-terminal present in the given grammar

**ALGORITHM:**

Step-1: Start

Step-2: Define a function first(string):

Step-2.1: first\_ -> set()

Step-2.2: if string in non\_terminals:

Step-2.2.1: alternatives = productions\_dict[string]

Step-2.2.2: for alternative in alternatives:

Step-2.2.2.1: first\_2 = first(alternative)

Step-2.2.2.2: first\_ = first\_ | first\_2

Step-2.3: elif string in terminals:

Step-2.3.1: first\_ = {string}

Step-2.4: elif string==" or string=="@':

Step-2.4.1: first\_ = {'@'}

Step-2.5: else:

Step-2.5.1: first\_2 = first(string[0])

Step-2.5.2: if '@' in first\_2:

Step-2.5.2.1: i = 1

Step-2.5.2.2: while '@' in first\_2:

first\_ = first\_ | (first\_2 - {'@'})

if string[i:] in terminals:

first\_ = first\_ | {string[i:]}

break

elif string[i:] == "':

first\_ = first\_ | {'@'}

break

first\_2 = first(string[i:])

first\_ = first\_ | first\_2 - {'@'}

i += 1

Step-2.5.3: else:

Step-2.5.3.1: first\_ = first\_ | first\_2

Step-2.6: return first\_

Step-3: Read the number of terminals and the terminals

Step-4: Read the number of non-terminals and the non-terminals

Step-5: Read the Starting symbol

Step-6: Read the number of productions and the productions

Step-7: FIRST = {}

Step-8: for non\_terminal in non\_terminals:

Step-8.1: FIRST[non\_terminal]=set()

Step-9: for non\_terminal in non\_terminals:

Step-9.1: FIRST[non\_terminal] = FIRST[non\_terminal] | first(non\_terminal)

Step-10: for non\_terminal in non\_terminals:

Step-10.1: Print FIRST(non\_terminal)

Step-11: End

**PROGRAM:**

```
import sys
sys.setrecursionlimit(60)
def first(string):
    first_ = set()
    if string in non_terminals:
        alternatives = productions_dict[string]
        for alternative in alternatives:
            first_2 = first(alternative)
            first_ = first_ | first_2
    elif string in terminals:
        first_ = {string}
    elif string==" or string=="@':
        first_ = {'@'}
    else:
        first_2 = first(string[0])
        if '@' in first_2:
            i = 1
            while '@' in first_2:
                first_ = first_ | (first_2 - {'@'})
                if string[i:] in terminals:
                    first_ = first_ | {string[i:]}
                    break
                elif string[i:] == "":
                    first_ = first_ | {'@'}
                    break
                first_2 = first(string[i:])
                first_ = first_ | first_2 - {'@'}
                i += 1
            else:
                first_ = first_ | first_2
    return first_
no_of_terminals=int(input("Enter no. of terminals: "))
terminals = []
print("Enter the terminals :")
for _ in range(no_of_terminals):
    terminals.append(input())
no_of_non_terminals=int(input("Enter no. of non terminals: "))
non_terminals = []
print("Enter the non terminals :")
for _ in range(no_of_non_terminals):
    non_terminals.append(input())
starting_symbol = input("Enter the starting symbol: ")
```

```
no_of_productions = int(input("Enter no of productions: "))
productions = []
print("Enter the productions:")
for _ in range(no_of_productions):
    productions.append(input())
productions_dict = {}
for nT in non_terminals:
    productions_dict[nT] = []
for production in productions:
    nonterm_to_prod = production.split("->")
    alternatives = nonterm_to_prod[1].split("/")
    for alternative in alternatives:
        productions_dict[nonterm_to_prod[0]].append(alternative)
FIRST = {}
for non_terminal in non_terminals:
    FIRST[non_terminal] = set()
for non_terminal in non_terminals:
    FIRST[non_terminal] = FIRST[non_terminal] | first(non_terminal)
print("{: ^20}{: ^20}".format('Non Terminals','First'))
for non_terminal in non_terminals:
    print("{: ^20}{: ^20}".format(non_terminal,str(FIRST[non_terminal])))
```

## OUTPUT:

```
===== RESTART: D:\Engineering\SEM-6\Compiler Design\Lab Work\Week-3\1.py =====
Enter no. of terminals: 5
Enter the terminals :
id
(
)
+
*
Enter no. of non terminals: 5
Enter the non terminals :
A
B
C
D
E
Enter the starting symbol: A
Enter no of productions: 5
Enter the productions:
A->CB
B->+CB/@
C->ED
D->*ED/@
E->(A)/id
Non Terminals      First
    A              {'(', 'id'}
    B              {'@', '+'}
    C              {'(', 'id'}
    D              {'@', '*'}
    E              {'(', 'id'}
```

## CONCLUSION:

By executing the above program, we have successfully implemented the FIRST Function for the given grammar

## EXPERIMENT NUMBER – 11

**AIM:** To implement FOLLOW Function for a given grammar

**DESCRIPTION:** In this program, we try to find the FOLLOW Functions of every non-terminal present in the given grammar

**ALGORITHM:**

Step-1: Start

Step-2: Define a function first(string) as defined in the previous program

Step-3: Define a function follow(string):

    Step-3.1: follow\_ = set()

    Step-3.2: prods = productions\_dict.items

    Step-3.3: for nt,rhs in prods:

        Step-3.3.1: for alt in rhs:

            Step-3.3.1.1: for char in alt:

                if char==nT:

                    following\_str = alt[alt.index(char) + 1:]

                    if following\_str=="":

                        if nt==nT:

                            continue

                        else:

                            follow\_ = follow\_ | follow(nt)

                else:

                    follow\_2 = first(following\_str)

                    if '@' in follow\_2:

                        follow\_ = follow\_ | follow\_2-{'@'}

                        follow\_ = follow\_ | follow(nt)

                else:

                    follow\_ = follow\_ | follow\_2

    Step-3.4: return follow\_

Step-4: Read the number of terminals and the terminals

Step-5: Read the number of non-terminals and the non-terminals

Step-6: Read the Starting symbol

Step-7: Read the number of productions and the productions

Step-8: FOLLOW={}

Step-9: FOLLOW[starting\_symbol] = FOLLOW[starting\_symbol] | {'\$'}

Step-10: for non\_terminal in non\_terminals:

    Step-10.1: FOLLOW[non\_terminal] = FOLLOW[non\_terminal] | follow(non\_terminal)

Step-11: for non\_terminal in non\_terminals:

    Step-11.1: Print FOLLOW[non\_terminals]

Step-12: End



**PROGRAM:**

```
import sys
sys.setrecursionlimit(60)
def first(string):
    first_ = set()
    if string in non_terminals:
        alternatives = productions_dict[string]
        for alternative in alternatives:
            first_2 = first(alternative)
            first_ = first_ | first_2
    elif string in terminals:
        first_ = {string}
    elif string==" or string=="@':
        first_ = {'@'}
    else:
        first_2 = first(string[0])
        if '@' in first_2:
            i = 1
            while '@' in first_2:
                first_ = first_ | (first_2 - {'@'})
                if string[i:] in terminals:
                    first_ = first_ | {string[i:]}
                    break
                elif string[i:] == "':
                    first_ = first_ | {'@'}
                    break
                first_2 = first(string[i:])
                first_ = first_ | first_2 - {'@'}
                i += 1
            else:
                first_ = first_ | first_2
    return first_
def follow(nT):
    follow_ = set()
    prods = productions_dict.items()
    if nT==starting_symbol:
        follow_ = follow_ | {'$'}
    for nt,rhs in prods:
        for alt in rhs:
            for char in alt:
                if char==nT:
                    following_str = alt[alt.index(char) + 1:]
                    if following_str=="':
                        if nt==nT:
                            continue
                        else:
                            follow_ = follow_ | follow(nt)
```

```
        else:
            follow_2 = first(following_str)
            if '@' in follow_2:
                follow_ = follow_ | follow_2-{'@'}
                follow_ = follow_ | follow(nt)
            else:
                follow_ = follow_ | follow_2
    return follow_
no_of_terminals=int(input("Enter no. of terminals: "))
terminals = []
print("Enter the terminals :")
for _ in range(no_of_terminals):
    terminals.append(input())
no_of_non_terminals=int(input("Enter no. of non terminals: "))
non_terminals = []
print("Enter the non terminals :")
for _ in range(no_of_non_terminals):
    non_terminals.append(input())
starting_symbol = input("Enter the starting symbol: ")
no_of_productions = int(input("Enter no of productions: "))
productions = []
print("Enter the productions:")
for _ in range(no_of_productions):
    productions.append(input())
productions_dict = {}
for nT in non_terminals:
    productions_dict[nT] = []
for production in productions:
    nonterm_to_prod = production.split("->")
    alternatives = nonterm_to_prod[1].split("/")
    for alternative in alternatives:
        productions_dict[nonterm_to_prod[0]].append(alternative)
FOLLOW = {}
for non_terminal in non_terminals:
    FOLLOW[non_terminal] = set()
FOLLOW[starting_symbol] = FOLLOW[starting_symbol] | {'$'}
for non_terminal in non_terminals:
    FOLLOW[non_terminal] = FOLLOW[non_terminal] | follow(non_terminal)
print("{: ^20}{: ^20}".format('Non Terminals','Follow'))
for non_terminal in non_terminals:
    print("{: ^20}{: ^20}".format(non_terminal,str(FOLLOW[non_terminal])))
```

## OUTPUT:

```
===== RESTART: D:\Engineering\SEM-6\Compiler Design\Lab Work\Week-3\2.py =====
Enter no. of terminals: 5
Enter the terminals :
id
(
)
+
*
Enter no. of non terminals: 5
Enter the non terminals :
A
B
C
D
E
Enter the starting symbol: A
Enter no of productions: 5
Enter the productions:
A->CB
B->+CB/@
C->ED
D->*ED/@
E->(A)/id
Non Terminals      Follow
    A              {'$', ')'}
    B              {'$', ')'}
    C              {')', '$', '+'}
    D              {')', '$', '+'}
    E              {')', '$', '*', '+'}
```

## CONCLUSION:

By executing the above program, we have successfully implemented the FOLLOW Function for the given grammar

## EXPERIMENT NUMBER – 12

**AIM:** To construct LL(1) parsing table and also check whether the string is valid or not ]

**DESCRIPTION:** In this program, we have to construct LL(1) parsing table for the given grammar and also have to check whether the given string is valid or not

**ALGORITHM:**

- Step-1: Start
- Step-2: Read the grammar from the user
- Step-3: Eliminate ambiguity from the grammar
- Step-4: Eliminate left Recursion from the grammar
- Step-5: Left Factor the grammar
- Step-6: Compute the FIRST function
- Step-7: Compute the FOLLOW function
- Step-8: Compute the LL(1) Parsing table
- Step-9: Display the LL(1) Parsing table
- Step-10: Read the input string from the user
- Step-11: Check whether the given string is valid or not.
- Step-12: End

**PROGRAM:**

```
def removeLeftRecursion(rulesDiction):
    store = {}
    for lhs in rulesDiction:
        alphaRules = []
        betaRules = []
        allrhs = rulesDiction[lhs]
        for subrhs in allrhs:
            if subrhs[0] == lhs:
                alphaRules.append(subrhs[1:])
            else:
                betaRules.append(subrhs)
        if len(alphaRules) != 0:
            lhs_ = lhs + ""
            while (lhs_ in rulesDiction.keys()) or (lhs_ in store.keys()):
                lhs_ += ""
            for b in range(0, len(betaRules)):
                betaRules[b].append(lhs_)
            rulesDiction[lhs] = betaRules
            for a in range(0, len(alphaRules)):
                alphaRules[a].append(lhs_)
            alphaRules.append(['#'])
            store[lhs_] = alphaRules
    for left in store:
        rulesDiction[left] = store[left]
    return rulesDiction
```

```
def LeftFactoring(rulesDiction):
    newDict = {}
    for lhs in rulesDiction:
        allrhs = rulesDiction[lhs]
        temp = dict()
        for subrhs in allrhs:
            if subrhs[0] not in list(temp.keys()):
                temp[subrhs[0]] = [subrhs]
            else:
                temp[subrhs[0]].append(subrhs)
        new_rule = []
        tempo_dict = {}
        for term_key in temp:
            allStartingWithTermKey = temp[term_key]
            if len(allStartingWithTermKey) > 1:
                lhs_ = lhs + ""
                while (lhs_ in rulesDiction.keys()) or (lhs_ in tempo_dict.keys()):
                    lhs_ += ""
                new_rule.append([term_key, lhs_])
                ex_rules = []
                for g in temp[term_key]:
                    ex_rules.append(g[1:])
                tempo_dict[lhs_] = ex_rules
            else:
                new_rule.append(allStartingWithTermKey[0])
        newDict[lhs] = new_rule
        for key in tempo_dict:
            newDict[key] = tempo_dict[key]
    return newDict

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"\nAfter elimination of left recursion:\n")
    diction = removeLeftRecursion(diction)
    for y in diction:
        print(f"{y}->{diction[y]}")
    print("\nAfter left factoring:\n")
    diction = LeftFactoring(diction)
    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
def createParseTable():
    import copy
    global diction, firsts, follows, term_userdef
    print("\nFirsts and Follow Result table\n")
    mx_len_first = 0
    mx_len_fol = 0
    for u in diction:
        k1 = len(str(firsts[u]))
        k2 = len(str(follows[u]))
        if k1 > mx_len_first:
            mx_len_first = k1
        if k2 > mx_len_fol:
            mx_len_fol = k2
    print(f"{{: <{10}}}"
          f"{{: <{mx_len_first + 5}}}"
          f"{{: <{mx_len_fol + 5}}}"
          .format("Non-T", "FIRST", "FOLLOW"))
    for u in diction:
        print(f"{{: <{10}}}"
              f"{{: <{mx_len_first + 5}}}"
              f"{{: <{mx_len_fol + 5}}}"
              .format(u, str(firsts[u]), str(follows[u])))
    ntlist = list(diction.keys())
    terminals = copy.deepcopy(term_userdef)
    terminals.append('$')
    mat = []
    for x in diction:
        row = []
        for y in terminals:
            row.append("")
        mat.append(row)
    grammar_is_LL = True

```



```

for lhs in diction:
    rhs = diction[lhs]
    for y in rhs:
        res = first(y)
        if '#' in res:
            if type(res) == str:
                firstFollow = []
                fol_op = follows[lhs]
                if fol_op is str:
                    firstFollow.append(fol_op)
                else:
                    for u in fol_op:
                        firstFollow.append(u)
                res = firstFollow
            else:
                res.remove('#')
                res = list(res) + \
                    list(follows[lhs])
        ttemp = []
        if type(res) is str:
            ttemp.append(res)
            res = copy.deepcopy(ttemp)
        for c in res:
            xnt = ntlist.index(lhs)
            yt = terminals.index(c)
            if mat[xnt][yt] == "":
                mat[xnt][yt] = mat[xnt][yt] \
                    + f"{lhs}->{' '.join(y)}"
            else:
                if f"{lhs}->{y}" in mat[xnt][yt]:
                    continue
                else:
                    grammar_is_LL = False
                    mat[xnt][yt] = mat[xnt][yt] \
                        + f",{lhs}->{' '.join(y)}"
        print("\nGenerated parsing table:\n")
        frmt = "{:>12}" * len(terminals)
        print(frmt.format(*terminals))
        j = 0
        for y in mat:
            frmt1 = "{:>12}" * len(y)
            print(f"{ntlist[j]} {frmt1.format(*y)}")
            j += 1
    return (mat, grammar_is_LL, terminals)
def validateStringUsingStackBuffer(parsing_table, grammarll1, table_term_list, input_string,
term_userdef,start_symbol):
    print(f"\nValidate String => {input_string}\n")

```

```

if grammarll1 == False:
    return f"\nInput String = " \
           f"\n{input_string}\n" \
           f"Grammar is not LL(1)"
stack = [start_symbol, '$']
buffer = []
input_string = input_string.split()
input_string.reverse()
buffer = ['$'] + input_string
print("{:>20} {:>20} {:>20}".format("Buffer", "Stack", "Action"))
while True:
    if stack == ['$'] and buffer == ['$']:
        print("{:>20} {:>20} {:>20}".format(' '.join(buffer),
                                           ' '.join(stack), "Valid"))
        return "\nValid String!"
    elif stack[0] not in term_userdef:
        x = list(diction.keys()).index(stack[0])
        y = table_term_list.index(buffer[-1])
        if parsing_table[x][y] != "":
            entry = parsing_table[x][y]
            print("{:>20} {:>20} {:>25}".
                  format(' '.join(buffer),
                         ' '.join(stack),
                         f"T[{stack[0]}][{buffer[-1]}] = {entry}"))
            lhs_rhs = entry.split(">")
            lhs_rhs[1] = lhs_rhs[1].replace('#', "").strip()
            entryrhs = lhs_rhs[1].split()
            stack = entryrhs + stack[1:]
        else:
            return f"\nInvalid String! No rule at " \
                   f"Table[{stack[0]}][{buffer[-1]}]."
    else:
        if stack[0] == buffer[-1]:
            print("{:>20} {:>20} {:>20}"
                  .format(' '.join(buffer),
                         ' '.join(stack),
                         f"Matched:{stack[0]}"))
            buffer = buffer[:-1]
            stack = stack[1:]
        else:
            return "\nInvalid String! " \
                   "Unmatched terminal symbols"
sample_input_string = None
no_of_terminals=int(input("Enter no. of terminals: "))
term_userdef=[]
print("Enter the terminals: ")
for _ in range(no_of_terminals):

```

```
term_userdef.append(input())
no_of_non_terminals=int(input("Enter no. of non terminals: "))
nonterm_userdef=[]
print("Enter the non terminals: ")
for _ in range(no_of_non_terminals):
    nonterm_userdef.append(input())
no_of_productions = int(input("Enter no of productions: "))
rules = []
print("Enter the productions: ")
for _ in range(no_of_productions):
    rules.append(input())
sample_input_string=input("Enter the input string: ")
diction = {}
firsts = {}
follows = {}
computeAllFirsts()
start_symbol = list(diction.keys())[0]
computeAllFollows()
(parsing_table, result, tabTerm) = createParseTable()
if sample_input_string != None:
    validity = validateStringUsingStackBuffer(parsing_table, result, tabTerm,
sample_input_string, term_userdef,start_symbol)
    print(validity)
else:
    print("\nNo input String detected")
```

## OUTPUT:

---

```
Enter no. of terminals: 7
Enter the terminals:
k
o
d
a
c
b
r
Enter no. of non terminals: 3
Enter the non terminals:
A
B
C
Enter no of productions: 4
Enter the productions:
S -> A k o
A -> A d | a B | a C
C -> c
B -> b B C | r
Enter the input string: a r k o

After elimination of left recursion:

S->[['A', 'k', 'o']]
A->[['a', 'B', "A'"], ['a', 'C', "A'"]]
C->[['c']]
B->[['b', 'B', 'C'], ['r']]
A'->[['d', "A'"], ['#']]

After left factoring:

S->[['A', 'k', 'o']]
A->[['a', "A'"]]
A''->[['B', "A'"], ['C', "A'"]]
C->[['c']]
B->[['b', 'B', 'C'], ['r']]
A'->[['d', "A'"], ['#']]

Calculated firsts:
first(S) => {'a'}
first(A) => {'a'}
first(A'') => {'c', 'r', 'b'}
first(C) => {'c'}
first(B) => {'r', 'b'}
first(A') => {'d', '#'}

```

---

Calculated follows:

```
follow(S) => {'$'}
follow(A) => {'k'}
follow(A') => {'k'}
follow(C) => {'c', 'd', 'k'}
follow(B) => {'c', 'd', 'k'}
follow(A') => {'k'}
```

Firsts and Follow Result table

Non-T	FIRST	FOLLOW
S	{ 'a' }	{ '\$' }
A	{ 'a' }	{ 'k' }
A'	{ 'c', 'r', 'b' }	{ 'k' }
C	{ 'c' }	{ 'c', 'd', 'k' }
B	{ 'r', 'b' }	{ 'c', 'd', 'k' }
A'	{ 'd', '#' }	{ 'k' }

Generated parsing table:

	\$	k	o	d	a	c	b	r
S					S->A k o			
A					A->a A'			
A'						A'->C A'	A'->B A'	A'->B A'
C						C->c		
B							B->b B C	B->r
A'		A'->#		A'->d A'				

Validate String => a r k o

Buffer	Stack	Action
\$ o k r a	S \$	T[S][a] = S->A k o
\$ o k r a	A k o \$	T[A][a] = A->a A'
\$ o k r a	a A' k o \$	Matched:a
\$ o k r	A' k o \$	T[A'][r] = A'->B A'
\$ o k r	B A' k o \$	T[B][r] = B->r
\$ o k r	r A' k o \$	Matched:r
\$ o k	A' k o \$	T[A'][k] = A'->#
\$ o k	k o \$	Matched:k
\$ o	o \$	Matched:o
\$	\$	Valid

Valid String!

## CONCLUSION:

By executing the above program, we have successfully constructed LL(1) parsing table for the given grammar and also checked whether the given string is valid or not.

## EXPERIMENT NUMBER – 13

**AIM:** To find whether the number is even or odd number using lex tool

**DESCRIPTION:** In this program, we try to read the input from the user and find whether the number is even or odd using lex tool

**ALGORITHM:**

Step-1: Start

Step-2: Declare i

Step-3: Declare the regular expression and conditions:

```
[0-9]+ {i= atoi(yytext);
```

```
if (i%2==0)
```

```
    printf("Even");
```

```
else
```

```
    printf("Odd");}
```

Step-4: Read input from the user

Step-5: End

**PROGRAM:**

```
%{
```

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

```
int i;
```

```
%}
```

```
%%
```

```
[0-9]+ {i= atoi(yytext);
```

```
if (i%2==0)
```

```
    printf("Even");
```

```
else
```

```
    printf("Odd");}
```

```
%%
```

```
int yywrap() {}
```

```
int main()
```

```
{
```

```
    yylex();
```

```
    return 0;
```

```
}
```

### OUTPUT:

```
nikki@nikki-VirtualBox:~$ gedit evenorodd.l
nikki@nikki-VirtualBox:~$ lex evenorodd.l
nikki@nikki-VirtualBox:~$ gcc lex.yy.c
nikki@nikki-VirtualBox:~$ ./a.out
10
Even
5
Odd
4
Even
3
Odd
7
Odd
^C
nikki@nikki-VirtualBox:~$
```

### CONCLUSION:

By executing the above program, we have successfully found whether the given number is even or odd number

## EXPERIMENT NUMBER – 14

**AIM:** To identify the characters other than alphabets using lex tool

**DESCRIPTION:** In this program, we try to read the input from the user and identify the characters other than alphabets present in the input

**ALGORITHM:**

Step-1: Start

Step-2: Declare len -> 0

Step-3: Declare the regular expressions

```
[a-zA-Z]+ {printf("No character other than alphabets");}  
.* {printf("character other than alphabets present"); }
```

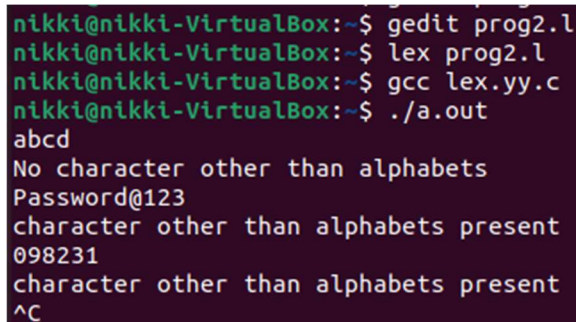
Step-4: Read the input

Step-5: End

**PROGRAM:**

```
%{  
    int len=0;  
}%  
%%  
[a-zA-Z]+ {printf("No character other than alphabets");}  
.* {printf("character other than alphabets present"); }  
%%  
int yywrap() { }  
int main()  
{  
    yylex();  
    return 0;  
}
```

**OUTPUT:**



```
nikki@nikki-VirtualBox:~$ gedit prog2.l  
nikki@nikki-VirtualBox:~$ lex prog2.l  
nikki@nikki-VirtualBox:~$ gcc lex.yy.c  
nikki@nikki-VirtualBox:~$ ./a.out  
abcd  
No character other than alphabets  
Password@123  
character other than alphabets present  
098231  
character other than alphabets present  
^C
```

**CONCLUSION:**

By executing the above program, we have successfully identified the characters other than alphabets



## EXPERIMENT NUMBER – 15

**AIM:** To add line number to statements in the given file

**DESCRIPTION:** In this program, we try to add line numbers to the statements present in a particular file

**ALGORITHM:**

Step-1: Start

Step-2: Declare line\_number->1

Step-3: Declare the conditions

```
{line} { printf("%10d %s", line_number++, yytext); }
```

Step-4: extern FILE \*yyin

Step-5: yyin -> fopen("sample.c", "r")

Step-6: End

**PROGRAM:**

**lineno.l:**

```
%{  
int line_number = 1;  
%}  
line .*\\n  
%%  
{line} { printf("%10d %s", line_number++, yytext); }  
%%  
int yywrap(){}  
int main(int argc, char*argv[])  
{  
extern FILE *yyin;  
yyin = fopen("sample.c", "r");  
yylex();  
return 0;  
}
```

**sample.c:**

```
#include <stdio.h>  
int main()  
{  
printf("Welcome to CBIT");  
printf("CD LAB");  
printf("WEEK-6");  
return 0;  
}
```

### OUTPUT:

```
nikki@nikki-VirtualBox:~$ gedit lineno.l
nikki@nikki-VirtualBox:~$ lex lineno.l
nikki@nikki-VirtualBox:~$ gcc lex.yy.c
nikki@nikki-VirtualBox:~$ ./a.out
1 #include <stdio.h>
2 int main()
3 {
4     printf("Welcome to CBIT");
5     printf("CD LAB");
6     printf("WEEK-6");
7     return 0;
8 }
```

### CONCLUSION:

By executing the above program, we have successfully added the line numbers to the statements present in a particular file

## EXPERIMENT NUMBER – 16

**AIM:** To implement Recursive Decent Parser

**DESCRIPTION:** In this program, we try to implement recursive decent parser

**ALGORITHM:**

Step-1: Start

Step-2: Declare a global variable s

Step-3: Read the string from user and store it as s

Step-4: Declare a global variable i=0

Step-5: Define a match function used to match the sting elements with the productions

Step-6: Define E, F, T, Tx, Ex for different productions

Step-7: if (E())

    Step-7.1: if i==len(s)

        Step-7.1.1: Print "String is accepted"

    Step-7.2: else

        Step-7.2.1: Print "String is not accepted"

Step-8: else

    Step-8.1: Print "String is not accepted"

Step-9: Stop

**PROGRAM:**

```
print("Recursive Desent Parsing For following grammar\n")
print("E->TE'\nE'->+TE'/@\nT->FT'\nT'->*FT'/@\nF->(E)/i\n")
print("Enter the string want to be checked\n")
global s
s=list(input())
global i
i=0
def match(a):
    global s
    global i
    if(i>=len(s)):
        return False
    elif(s[i]==a):
        i+=1
        return True
    else:
        return False
def F():
    if(match("(")):
        if(E()):
            if(match(")")):
                return True
            else:
                return False
```

```
        else:
            return False
    elif(match("i")):
        return True
    else:
        return False
def Tx():
    if(match("*")):
        if(F()):
            if(Tx()):
                return True
            else:
                return False
        else:
            return False
    else:
        return True
def T():
    if(F()):
        if(Tx()):
            return True
        else:
            return False
    else:
        return False
def Ex():
    if(match("+")):
        if(T()):
            if(Ex()):
                return True
            else:
                return False
        else:
            return False
    else:
        return True
def E():
    if(T()):
        if(Ex()):
            return True
        else:
            return False
    else:
        return False
if(E()):
    if(i==len(s)):
        print("String is accepted")
```

```
else:  
    print("String is not accepted")
```

```
else:  
    print("string is not accepted")
```

### OUTPUT:

```
Recursive Descent Parsing For following grammar
```

```
E->TE'  
E'->+TE'/@  
T->FT'  
T'->*FT'/@  
F->(E)/i
```

```
Enter the string want to be checked
```

```
(i)*i  
String is accepted
```

### CONCLUSION:

By executing the above program, we have successfully implemented the Recursive Decent Parser.

## EXPERIMENT NUMBER – 17

**AIM:** To implement Canonical LR(0) items

**DESCRIPTION:** In this program, we try to implement canonical LR(0) items

**ALGORITHM:**

Step-1: Start

Step-2: Define a function findlr0:

    Step-2.1: for i in range(len(rhs)+1):

        Step-2.1.1: x=lhs+'->'+rhs[:i]+'.'+rhs[i:]

        Step-2.1.2: lr0.append(x)

Step-3: Read the number of productions

Step-4: Read the productions

Step-5: for i in range(n):

    Step-5.1: lr0=[]

    Step-5.2: for i in range(len(arr)):

        Step-5.2.1: ip=arr[i]

        Step-5.2.2: lhs,rhs=ip.split("->")

        Step-5.2.3: productions=list(rhs.split('|'))

        Step-5.2.4: for prod in productions:

            Step-5.2.4.1: findlr0(lhs,prod,lr0)

Step-6: Print lr0

Step-7: End

**PROGRAM:**

```
def findlr0(lhs,rhs,lr0):
    for i in range(len(rhs)+1):
        x=lhs+'->'+rhs[:i]+'.'+rhs[i:]
        lr0.append(x)
n=int(input("Enter the no. of productions:"))
arr=[]
for i in range(n):
    arr.append(str(input()))
lr0=[]
for i in range(len(arr)):
    ip=arr[i]
    lhs,rhs=ip.split("->")
    productions=list(rhs.split('|'))
    for prod in productions:
        findlr0(lhs,prod,lr0)
print("LR(0) items for given production:")
for i in range(len(lr0)):
    print(i,"->",lr0[i])
```

**OUTPUT:**

```
Enter the no. of productions:3
E->E+T|T
T->T*F|F
F->(E)|i
LR(0) items for given production:
0 -> E->.E+T
1 -> E->E.+T
2 -> E->E+.T
3 -> E->E+T.
4 -> E->.T
5 -> E->T.
6 -> T->.T*F
7 -> T->T.*F
8 -> T->T*.F
9 -> T->T*F.
10 -> T->.F
11 -> T->F.
12 -> F->.(E)
13 -> F->(.E)
14 -> F->(E.)
15 -> F->(E) .
16 -> F->.i
17 -> F->i.
```

**CONCLUSION:**

By executing the above program, we have successfully implemented Canonical LR(0) items

## EXPERIMENT NUMBER – 18

**AIM:** To recognize a valid arithmetic expression that uses operator +, -, \*, % using lex and yacc tool

**DESCRIPTION:** In this program, we try to recognize a valid arithmetic expression that uses operator +, -, \*, % using lex and yacc tool

### ALGORITHM:

Step-1: Start

Step-2: Create a lex file

Step-3: Declare the regular expressions

```
[a-zA-Z_][a-zA-Z_0-9]* return id;
[0-9]+(\.[0-9]*)?    return num;
[+/*]                return op;
.                    return yytext[0];
\n                   return 0;
```

Step-4: Create a Yacc file

Step-5: valid = 1

Step-6: Declare required variables

```
start : id '=' s ';'
s :    id x
      | num x
      | '-' num x
      | '(' s ')' x
      ;
x :    op s
      | '-' s
      |
      ;
```

Step-7: Read the expression from user

Step-8: if(valid) Print "Valid Expression"

Step-9: else Print "Invalid Expression"

Step-10: End

### PROGRAM:

#### Lex part:

```
%{
#include "y.tab.h"
%}
%%
[a-zA-Z_][a-zA-Z_0-9]* return id;
[0-9]+(\.[0-9]*)?    return num;
[+/*]                return op;
.                    return yytext[0];
\n                   return 0;
%%
```



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

**Yacc Part:**

```
%{
#include<stdio.h>
int valid=1;
%}
%token num id op
%%
start : id '=' s ';'
s : id x
  | num x
  | '-' num x
  | '(' s ')' x
  ;
x : op s
  | '-' s
  |
  ;
%%
int yyerror()
{
valid=0;
printf("\nInvalid expression!\n");
return 0;
}
int main()
{
printf("\nEnter the expression:\n");
yyparse();
if(valid)
{
printf("\nValid expression!\n");
}
}
```

### OUTPUT:

```
nikki@nikki-VirtualBox:~$ gedit arth.l
nikki@nikki-VirtualBox:~$ gedit arth.y
nikki@nikki-VirtualBox:~$ lex arth.l
nikki@nikki-VirtualBox:~$ yacc -d arth.y
arth.y:17 parser name defined to default : "parse"
nikki@nikki-VirtualBox:~$ gcc lex.yy.c y.tab.c -w
nikki@nikki-VirtualBox:~$ ./a.out

Enter the expression:
a=b+c;

Valid expression!
nikki@nikki-VirtualBox:~$ ./a.out

Enter the expression:
a+b-;

Invalid expression!
```

### CONCLUSION:

By executing the above program, we have successfully recognized a valid arithmetic expression that uses operator +, -, \*, % using lex and yacc tool

## EXPERIMENT NUMBER – 19

**AIM:** To recognize a valid variable using lex and yacc tool

**DESCRIPTION:** In this program, we try to recognize a valid variable using lex and yacc tool

**ALGORITHM:**

Step-1: Start

Step-2: Create a lex file

Step-3: Declare the regular expressions

```
[a-zA-Z_][a-zA-Z_0-9]* return letter;  
[0-9]          return digit;  
.             return yytext[0];  
\n            return 0;
```

Step-4: Create a yacc file

Step-5: valid=1

Step-6: Declare required variables

```
start : letter s  
      s : letter s  
        | digit s  
        |  
      ;
```

Step-7: Read the string for the user

Step-8: if (valid) Print "It's an Identifier"

Step-9: else Print "It's not an Identifier"

Step-10: End

**PROGRAM:**

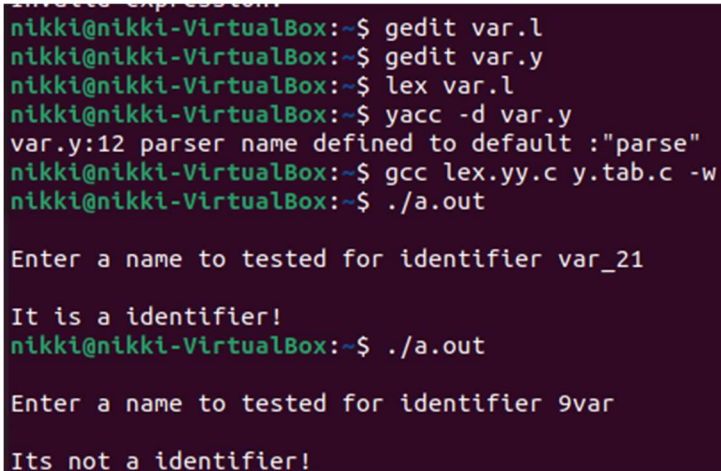
**Lex Part:**

```
%{  
    #include "y.tab.h"  
%}  
%%  
[a-zA-Z_][a-zA-Z_0-9]* return letter;  
[0-9]          return digit;  
.             return yytext[0];  
\n            return 0;  
%%  
int yywrap()  
{  
    return 1;  
}
```

### Yacc Part:

```
%{
    #include<stdio.h>
    int valid=1;
}%
%token digit letter
%%
start : letter s
s : letter s
    | digit s
    |
    ;
%%
int yyerror()
{
    printf("\nIts not a identifier!\n");
    valid=0;
    return 0;
}
int main()
{
    printf("\nEnter a name to tested for identifier ");
    yyparse();
    if(valid)
    {
        printf("\nIt is a identifier!\n");
    }
}
```

### OUTPUT:



```
nikki@nikki-VirtualBox:~$ gedit var.l
nikki@nikki-VirtualBox:~$ gedit var.y
nikki@nikki-VirtualBox:~$ lex var.l
nikki@nikki-VirtualBox:~$ yacc -d var.y
var.y:12 parser name defined to default : "parse"
nikki@nikki-VirtualBox:~$ gcc lex.yy.c y.tab.c -w
nikki@nikki-VirtualBox:~$ ./a.out

Enter a name to tested for identifier var_21

It is a identifier!
nikki@nikki-VirtualBox:~$ ./a.out

Enter a name to tested for identifier 9var

Its not a identifier!
```

### CONCLUSION:

By executing the above program, we have successfully recognized a valid variable or Identifier

## EXPERIMENT NUMBER – 20

**AIM:** To demonstrate calculator using lex and yacc tool

**DESCRIPTION:** In this program, we try to demonstrate the calculator operations using lex and yacc tool

**ALGORITHM:**

Step-1: Start

Step-2: Create a lex file

Step-3: Declare the regular expressions

```
yylval=atoi(yytext);  
return NUMBER;  
}
```

```
[\t];
```

```
[\n] return 0;
```

```
. return yytext[0];
```

Step-4: Create a yacc file

Step-5: Declare required variables

```
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; }
```

```
;
```

Step-6: Read the expression from the user

Step-7: if (valid) Print result of the expression and "Valid Expression"

Step-8: else Print "Invalid Expression"

Step-9: End

**PROGRAM:**

**Lex Part:**

```
%{  
#include<stdio.h>  
#include "y.tab.h"  
extern int yyval;  
%}  
%%  
[0-9]+ {  
    yyval=atoi(yytext);  
    return NUMBER;
```

```
    }
[\t];
[\n] return 0;
. return yytext[0];
%%
int yywrap()
{
return 1;
}
```

### **Yacc Part:**

```
%{
    #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, Divison, 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:

```
nikki@nikki-VirtualBox:~$ gedit calc.l
nikki@nikki-VirtualBox:~$ gedit calc.y
nikki@nikki-VirtualBox:~$ lex calc.l
nikki@nikki-VirtualBox:~$ yacc -d calc.y
calc.y:22 parser name defined to default : "parse"
nikki@nikki-VirtualBox:~$ gcc lex.yy.c y.tab.c -w
nikki@nikki-VirtualBox:~$ ./a.out

Enter Any Arithmetic Expression which can have operations Addition, Subtraction
, Multiplication, Divison, Modulus and Round brackets:
54+980

Result=1034

Entered arithmetic expression is Valid

nikki@nikki-VirtualBox:~$ ./a.out

Enter Any Arithmetic Expression which can have operations Addition, Subtraction
, Multiplication, Divison, Modulus and Round brackets:
69+90-

Entered arithmetic expression is Invalid
```

## CONCLUSION:

By executing the above program, we have successfully demonstrated calculator operations using lex and yacc tool

## EXPERIMENT NUMBER – 21

**AIM:** To check whether a given string is accepted by the given grammar

**DESCRIPTION:** In this program, we try to check whether the string aaabbb is accepted by the given grammar  $S \rightarrow aSb | \epsilon$

**ALGORITHM:**

Step-1: Start

Step-2: Create a lex file

Step-3: Declare the regular expressions

[a] {return A;}

[b] {return B;}

[\n] {return '\n';}

Step-4: Create a yacc file

Step-5: Declare required variable

start : S '\n' {return 0;}

S: A S B

;

Step-6: Read the string from the user

Step-7: if (valid) Print "Valid"

Step-8: else Print "Invalid"

Step-9: End

**PROGRAM:**

**Lex Part:**

```
%{
#include "y.tab.h"
%}
%%
[a] {return A;}
[b] {return B;}
[\n] {return '\n';}
%%
```

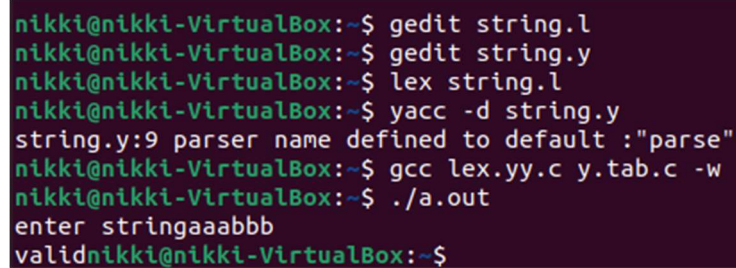
**Yacc Part:**

```
%{
#include<stdio.h>
%}
%token A B
%%
start : S '\n' {return 0;}
S: A S B
;
%%
main()
{
```



```
printf("enter string");
if(yyparse()==0)
printf("valid");
}
yyerror()
{printf("not accepted");
exit(0);
}
yywrap()
{
return 1;
}
```

**OUTPUT:**



```
nikki@nikki-VirtualBox:~$ gedit string.l
nikki@nikki-VirtualBox:~$ gedit string.y
nikki@nikki-VirtualBox:~$ lex string.l
nikki@nikki-VirtualBox:~$ yacc -d string.y
string.y:9 parser name defined to default : "parse"
nikki@nikki-VirtualBox:~$ gcc lex.yy.c y.tab.c -w
nikki@nikki-VirtualBox:~$ ./a.out
enter stringaaabbb
validnikki@nikki-VirtualBox:~$
```

**CONCLUSION:**

By executing the above program, we have successfully checked whether the string aaabbb is accepted by the grammar  $S \rightarrow aSb | \epsilon$

## EXPERIMENT NUMBER – 22

**AIM:** To stimulate symbol table management

**DESCRIPTION:** In this program, we try to demonstrate symbol table management

**ALGORITHM:**

Step-1: Start

Step-2: Read the expression from the user

Step-3: if isalpha(toascii(c)) Print "Identifier"

Step-4: else if isdigit(toascii(c)) Print "Constant"

Step-5: else Print "Operator"

Step-6: Stop

**PROGRAM:**

```
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
void main()
{
    int i=0,j=0,x=0,n;
    void *p,*add[5];
    char ch,srch,b[15],d[15],c;
    printf("Expression terminated by $:");
    while((c=getchar())!='$')
    {
        b[i]=c;
        i++;
    }
    n=i-1;
    printf("Given Expression:");
    i=0;
    while(i<=n)
    {
        printf("%c",b[i]);
        i++;
    }
    printf("\n Symbol Table\n");
    printf("Symbol \t addr \t type");
    while(j<=n)
    {
        c=b[j];
        if(isalpha(toascii(c)))
        {
            p=malloc(c);
```

```
add[x]=p;
d[x]=c;
printf("\n%c \t %d \t identifier\n",c,p);
x++;
j++;
}
else if (isdigit(c))
{
    p=malloc(c);
    add[x]=p;
    d[x]=c;
    printf("\n%c \t %d \t Constant\n",c,p);
    x++;
    j++;
}
else
{
    ch=c;
    if(ch=='+' || ch=='-' || ch=='*' || ch=='=')
    {
        p=malloc(ch);
        add[x]=p;
        d[x]=ch;
        printf("\n %c \t %d \t operator\n",ch,p);
        x++;
        j++;
    }
}
```

#### OUTPUT:

```
Expression terminated by $:x=9$
Given Expression:x=9
Symbol Table
Symbol    addr      type
x          11801680   identifier
=          11801808   operator
9          11801888   Constant
-----
Process exited after 6.827 seconds with return value 3
Press any key to continue . . . |
```

#### CONCLUSION:

By executing the above program, we have successfully stimulated the symbol table management

## EXPERIMENT NUMBER – 23

**AIM:** To implement language to an intermediate form

**DESCRIPTION:** In this program, we try to implement language to an intermediate form

**ALGORITHM:**

Step-1: Start  
Step-2: Define a structure three  
Step-3: f1=fopen("sum.txt","r")  
Step-4: f2=fopen("out.txt","w")  
Step-5: while(fscanf(f1,"%s",s[len].data)!=EOF)  
    Step-5.1: len++  
Step-6: itoa(j,d1,7)  
Step-7: strcat(d2,d1)  
Step-8: strcpy(s[j].temp,d2)  
Step-9: strcpy(d1,"")  
Step-10: strcpy(d2,"t")  
Step-11: if(!strcmp(s[3].data,"+"))  
    Step-11.1: Print(s[j].temp,s[i+2].data,s[i+4].data)  
    Step-11.2: j++  
Step-12: else if(!strcmp(s[3].data,"-"))  
    Step-12.1: Print(s[j].temp,s[i+2].data,s[i+4].data)  
    Step-12.2: j++  
Step-13: for(i=4;i<len-2;i+=2)  
    Step-13.1: itoa(j,d1,7)  
    Step-13.2: strcat(d2,d1)  
    Step-13.3: strcpy(s[j].temp,d2)  
    Step-13.4: if(!strcmp(s[i+1].data,"+"))  
        Step-13.4.1: Print(s[j].temp,s[j-1].temp,s[i+2].data)  
    Step-13.5: else if(!strcmp(s[i+1].data,"-"))  
        Step-13.5.1: Print(s[j].temp,s[j-1].temp,s[i+2].data)  
    Step-13.6: strcpy(d1,"")  
    Step-13.7: strcpy(d2,"t")  
    Step-13.8: j++  
Step-14: Print(s[0].data,s[j-1].temp)  
Step-15: fclose(f1)  
Step-16: fclose(f2)  
Step-17: End

**PROGRAM:**

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#include<string.h>
struct three
{
    char data[10],temp[7];
}s[30];
void main()
{
    char d1[7],d2[7]="t";
    int i=0,j=1,len=0;
    FILE *f1,*f2;
    f1=fopen("sum.txt","r");
    f2=fopen("out.txt","w");
    while(fscanf(f1,"%s",s[len].data)!=EOF)
        len++;
    itoa(j,d1,7);
    strcat(d2,d1);
    strcpy(s[j].temp,d2);
    strcpy(d1,"");
    strcpy(d2,"t");
    if(!strcmp(s[3].data,"+"))
    {
        fprintf(f2,"%s=%s+%s",s[j].temp,s[i+2].data,s[i+4].data);
        j++;
    }
    else if(!strcmp(s[3].data,"-"))
    {
        fprintf(f2,"%s=%s-%s",s[j].temp,s[i+2].data,s[i+4].data);
        j++;
    }
    for(i=4;i<len-2;i+=2)
    {
        itoa(j,d1,7);
        strcat(d2,d1);
        strcpy(s[j].temp,d2);
        if(!strcmp(s[i+1].data,"+"))
            fprintf(f2,"\n%s=%s+%s",s[j].temp,s[j-1].temp,s[i+2].data);
        else if(!strcmp(s[i+1].data,"-"))
            fprintf(f2,"\n%s=%s-%s",s[j].temp,s[j-1].temp,s[i+2].data);
        strcpy(d1,"");
        strcpy(d2,"t");
        j++;
    }
    fprintf(f2,"\n%s=%s",s[0].data,s[j-1].temp);
```

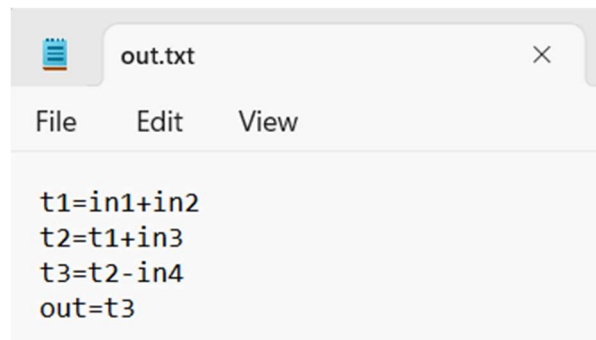
```
    fclose(f1);  
    fclose(f2);  
    getch();  
}
```

**sum.txt:**

out = in1 + in2 + in3 - in4

**OUTPUT:**

**out.txt:**



**CONCLUSION:**

By executing the above program, we have successfully implemented language to an intermediate code

## EXPERIMENT NUMBER – 24

**AIM:** To generate target code

**DESCRIPTION:** In this program, we try to generate target code from intermediate code

### ALGORITHM:

Step-1: Start  
Step-2: Define a structure three  
Step-3: f1=fopen("exe.txt","r")  
Step-4: f2=fopen("exe1.txt","w")  
Step-5: while(fscanf(f1,"%s",s[len].data)!=EOF)  
    Step-5.1: len++  
Step-6: for(i=0;i<=len;i++)  
    Step-6.1: if(!strcmp(s[i].data,"="))  
        Step-6.1.1: Print(s[i+1].data)  
    Step-6.2: if(!strcmp(s[i+2].data,"+"))  
        Step-6.2.1: Print(s[i+3].data)  
    Step-6.3: if(!strcmp(s[i+2].data,"-"))  
        Step-6.3.1: Print(s[i+3].data)  
    Step-6.4: Print(s[i-1].data)  
Step-7: fclose(f1)  
Step-8: fclose(f2)  
Step-9: Stop

### PROGRAM:

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
struct three
{
    char data[10],temp[7];
}s[30];
void main()
{
    char *d1,*d2;
    int i=0,len=0;
    FILE *f1,*f2;
    f1=fopen("exe.txt","r");
    f2=fopen("exe1.txt","w");
    while(fscanf(f1,"%s",s[len].data)!=EOF)
        len++;
    for(i=0;i<=len;i++)
    {
        if(!strcmp(s[i].data,"="))
        {
            fprintf(f2,"\nLDA\t%s",s[i+1].data);
```

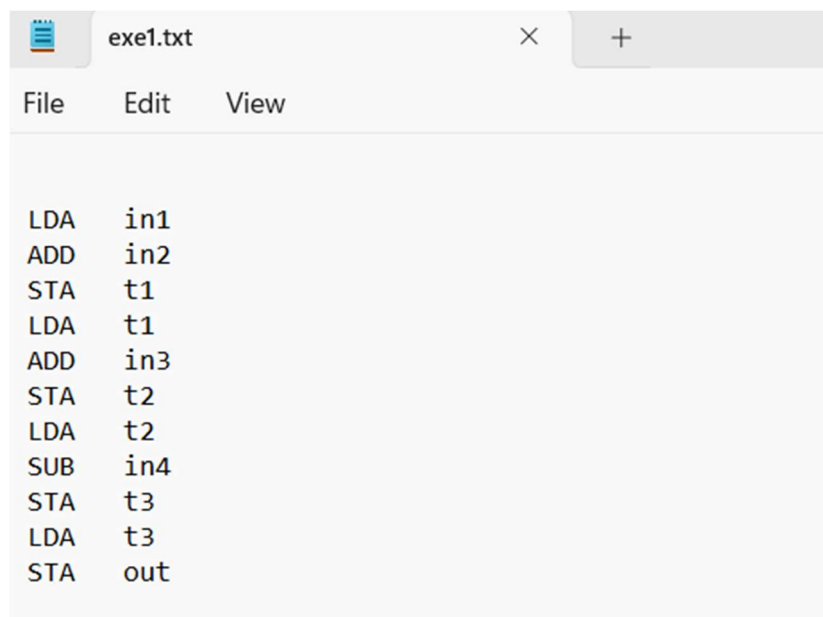
```
        if(!strcmp(s[i+2].data,"+"))
            fprintf(f2,"\nADD\t%s",s[i+3].data);
        if(!strcmp(s[i+2].data,"-"))
            fprintf(f2,"\nSUB\t%s",s[i+3].data);
        fprintf(f2,"\nSTA\t%s",s[i-1].data);
    }
}
fclose(f1);
fclose(f2);
getch();
}
```

**exe.txt:**

```
t1 = in1 + in2
t2 = t1 + in3
t3 = t2 - in4
out = t3
```

**OUTPUT:**

**exe1.txt:**



**CONCLUSION:**

By executing the above program, we have successfully generated target code from the intermediate code.



## EXPERIMENT NUMBER – 25

**AIM:** To implement Yacc program to check for relational operator.

**DESCRIPTION:** In this program, we try to implement Yacc program to check for relational operator.

**ALGORITHM:**

Step-1: Start the program.

Step-2: Reading an expression.

Step-3: Checking the validating of the given expression for relational operator according to the rule using Yacc.

Step-4: Using expression rule print the result of the given values

Step-5: Stop the program.

**PROGRAM:**

**Lex Part:**

```
%{
#include "y.tab.h"
}%
%%
[0-9]+    { yylval = atoi(yytext); return NUM; }
"=="      { return EQ; }
"!="      { return NEQ; }
"<"       { return LT; }
">"       { return GT; }
"<="      { return LTE; }
">="      { return GTE; }
[ \t]     ; /* ignore whitespace */
\n        ; /* ignore newline */
.         { return yytext[0]; } /* catch-all rule for unmatched characters */
%%
int yywrap() {
    return 1;
}
```

**Yacc Part:**

```
%{
#include <stdio.h>
int valid=1;
}%
%token NUM
%token EQ NEQ LT GT LTE GTE
%left EQ NEQ LT GT LTE GTE
%start expr
%%
expr: NUM { printf("Expression: %d\n", $1); }
```

```
| expr EQ expr { printf("Expression: %d == %d\n", $1, $3); }
| expr NEQ expr { printf("Expression: %d != %d\n", $1, $3); }
| expr LT expr { printf("Expression: %d < %d\n", $1, $3); }
| expr GT expr { printf("Expression: %d > %d\n", $1, $3); }
| expr LTE expr { printf("Expression: %d <= %d\n", $1, $3); }
| expr GTE expr { printf("Expression: %d >= %d\n", $1, $3); }
;
%%
int main() {
    yyparse();
    if(valid)
    {
        printf("Sucess");
    }
    return 0;
}
void yyerror(const char *s) {
    valid=0;
    printf("Error: %s\n", s);
}
```

#### OUTPUT:

```
nikki@nikki-VirtualBox:~$ gedit relational.l
nikki@nikki-VirtualBox:~$ gedit relational.y
nikki@nikki-VirtualBox:~$ lex relational.l
nikki@nikki-VirtualBox:~$ yacc -d relational.y
relational.y:18 parser name defined to default : "parse"
nikki@nikki-VirtualBox:~$ gcc lex.yy.c y.tab.c -w
nikki@nikki-VirtualBox:~$ ./a.out
3=5
Expression: 3
Error: parse error
```

```
nikki@nikki-VirtualBox:~$ gedit relational.l
nikki@nikki-VirtualBox:~$ gedit relational.y
nikki@nikki-VirtualBox:~$ lex relational.l
nikki@nikki-VirtualBox:~$ yacc -d relational.y
relational.y:18 parser name defined to default : "parse"
nikki@nikki-VirtualBox:~$ gcc lex.yy.c y.tab.c -w
nikki@nikki-VirtualBox:~$ ./a.out
3==5
Expression: 3
Expression: 5
Expression: 3 == 5
Success
```

#### CONCLUSION:

By executing the above program, we have successfully implemented Yacc program to check for relational operator.

## EXPERIMENT NUMBER – 26

**AIM:** To improve code with the help of optimization techniques

**DESCRIPTION:** In this program, we try to implement a program to improve code with the help of any one of the optimization techniques

**ALGORITHM:**

Step-1: Start

Step-2: Start by defining the structures for op and pr with l and r as members, representing left and right sides of an assignment statement.

Step-3: Take input for the number of values n.

Step-4: Loop through n times and take input for left and right sides of the assignment statements, storing them in the op structure.

Step-5: Print the intermediate code by looping through op and displaying l and r values.

Step-6: Perform dead code elimination by looping through op and checking if the l value is present in the r value of other op structures. If present, store it in pr structure.

Step-7: Print the result of dead code elimination by looping through pr and displaying l and r values.

Step-8: Perform common expression elimination by looping through pr and checking if the r value of one pr structure is a substring of r value of other pr structures. If present, replace the common expression with the l value of the first pr structure.

Step-9: Print the result of common expression elimination by looping through pr and displaying l and r values.

Step-10: Finally, eliminate redundant assignments by looping through pr and checking for duplicate assignments with same l and r values. If found, mark the l value as '\0'.

Print the optimized code by looping through pr and displaying l and r values, excluding the ones with l value as '\0'.

Step-11: Stop.

**PROGRAM:**

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

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

```
struct op
```

```
{
```

```
    char l;
```

```
    char r[20];
```

```
}op[10],pr[10];
```

```
void main()
```

```
{
```

```
    int a,i,k,j,n,z=0,m,q;
```

```
    char *p,*l;
```

```
    char temp,t;
```

```
    char *tem;
```

```
    printf("Enter the Number of Values:");
```

```
    scanf("%d",&n);
```

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

```

{
    printf("left: ");
    scanf(" %c",&op[i].l);
    printf("right: ");
    scanf(" %s",&op[i].r);
}
printf("Intermediate Code\n") ;
for(i=0;i<n;i++)
{
    printf("%c=",op[i].l);
    printf("%s\n",op[i].r);
}
for(i=0;i<n-1;i++)
{
    temp=op[i].l;
    for(j=0;j<n;j++)
    {
        p=strchr(op[j].r,temp);
        if(p)
        {
            pr[z].l=op[i].l;
            strcpy(pr[z].r,op[i].r);
            z++;
        }
    }
}
pr[z].l=op[n-1].l;
strcpy(pr[z].r,op[n-1].r);
z++;
printf("\nAfter Dead Code Elimination\n");
for(k=0;k<z;k++)
{
    printf("%c\t=",pr[k].l);
    printf("%s\n",pr[k].r);
}
for(m=0;m<z;m++)
{
    tem=pr[m].r;
    for(j=m+1;j<z;j++)
    {
        p=strstr(tem,pr[j].r);
        if(p)
        {
            t=pr[j].l;
            pr[j].l=pr[m].l;
            for(i=0;i<z;i++)
            {

```

```

                                l=strchr(pr[i].r,t) ;
                                if(l)
                                {
                                    a=l-pr[i].r;
                                    printf("pos: %d\n",a);
                                    pr[i].r[a]=pr[m].l;
                                }
                            }
                        }
                    }
                }
            }
        printf("Eliminate Common Expression\n");
        for(i=0;i<z;i++)
        {
            printf("%c\t=",pr[i].l);
            printf("%s\n",pr[i].r);
        }
        for(i=0;i<z;i++)
        {
            for(j=i+1;j<z;j++)
            {
                q=strcmp(pr[i].r,pr[j].r);
                if((pr[i].l==pr[j].l)&&!q)
                {
                    pr[i].l='\0';
                }
            }
        }
        printf("Optimized Code\n");
        for(i=0;i<z;i++)
        {
            if(pr[i].l!='\0')
            {
                printf("%c=",pr[i].l);
                printf("%s\n",pr[i].r);
            }
        }
    }
}

```

## OUTPUT:

```
Enter the Number of Values:5
left: a
right: 9
left: b
right: c+d
left: e
right: c+d
left: f
right: b+e
left: r
right: f
Intermediate Code
a=9
b=c+d
e=c+d
f=b+e
r=f

After Dead Code Elimination
b      =c+d
e      =c+d
f      =b+e
r      =f
pos: 2
Eliminate Common Expression
b      =c+d
b      =c+d
f      =b+b
r      =f
Optimized Code
b=c+d
f=b+b
r=f

-----
Process exited after 23.39 seconds with return value 4
Press any key to continue . . .
```

## CONCLUSION:

By executing the above program, we have successfully implemented Optimization techniques

## EXPERIMENT NUMBER – 27

**AIM:** To implement a standalone Scanner without lex tool. (Tokenization-by constructing DFA of lexical analyzer)

**DESCRIPTION:** This is a program in C language that implements a DFA (Deterministic Finite Automaton) for the lexical analysis of an input file. The program identifies the keywords, constants, and relational operators present in the file.

### ALGORITHM:

Step-1: Initialize the state to 0 and the flag to 0.

Step-2: Read the input file character by character until the end of file is reached.

Step-3: Based on the current state and the input character, transition to the next state.

Step-4: If a final state is reached, output the corresponding token (keyword, constant, or relational operator) and transition back to state 0.

Step-5: If the end of file is reached, set the flag to 1 and exit the loop.

Step-6: Close the input file.

### PROGRAM:

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<ctype.h>
void main()
{
    int state=0,flag=0,i,p=0,id=0;
    char
ch,word[20],kw[20][20]={"int","float","char","long","double","if","else","for","while","void",
"do","switch","case","break"};
    FILE *f;
    f=fopen("input.txt","r");
    while(flag!=1)
    {
        switch(state)
        {
            case 0:
                ch=fgetc(f);
                if(isalnum(ch))
                    if(isalpha(ch))
                        state=11;
                else
                    state=13;
                else if(ch=='<')
                    state=1;
                else if(ch=='>')
                    state=4;
                else if(ch=='!')
```

```
        state=7;
    else if(ch=='=')
        state=9;
    break;
case 1:
    ch=fgetc(f);
    if(ch=='=')
        state=2;
    else
        state=3;
    break;
case 2:
    printf("\n'<=' is a relational operator.");
    state=0;
    break;
case 3:
    fseek(f,-1,SEEK_CUR);
    printf("\n'<' is a relational operator.");
    state=0;
    break;
case 4:
    ch=fgetc(f);
    if(ch=='=')
        state=5;
    else
        state=6;
    break;
case 5:
    printf("\n'>=' is a relational operator.");
    state=0;
    break;
case 6:
    fseek(f,-1,SEEK_CUR);
    printf("\n'>' is a relational operator.");
    state=0;
    break;
case 7:
    ch=fgetc(f);
    if(ch=='=')
        state=8;
    else
    {
        fseek(f,-1,SEEK_CUR);
        state=0;
    }
    break;
case 8:
```



```
    printf("\n!=' is a relational operator.");
    state=0;
    break;
case 9:
    ch=fgetc(f);
    if(ch=='=')
        state=8;
    else
    {
        fseek(f,-1,SEEK_CUR);
        state=0;
    }
    break;
case 10:
    printf("\n'==' is a relational operator.");
    state=0;
    break;
case 11:
    word[p++]=ch;
    while(isalnum(ch=fgetc(f)))
        word[p++]=ch;
    fseek(f,-1,SEEK_CUR);
    word[p]='\0';
    state=12;
    p=0;
    break;
case 12:
    for(i=0;i<14;i++)
        if(strcmp(kw[i],word)==0)
        {
            printf("\n%s is a keyword.",word);
            id=1;
            break;
        }
    if(id==0)
        printf("\n%s is an identifier.",word);
    state=0;
    id=0;
    break;
case 13:
    word[p++]=ch;
    while(isdigit(ch=fgetc(f)))
        word[p++]=ch;
    fseek(f,-1,SEEK_CUR);
    word[p]='\0';
    state=14;
    p=0;
```

```
        break;
    case 14:
        printf("\n%s is a constant.",word);
        state=0;
        break;
    default:
        break;
}
if(ch==EOF)
    flag=1;
}
fclose(f);
}
```

**input.txt:**

int i = 1;

**OUTPUT:**

```
int is a keyword.
i is an identifier.
1 is a constant.
-----
Process exited after 0.02435 seconds with return value 0
Press any key to continue . . .
```

**CONCLUSION:**

By executing the above program, we have successfully implemented a standalone Scanner without lex tool. (Tokenization-by constructing DFA of lexical analyzer).

## EXPERIMENT NUMBER – 28

**AIM:** To implement a parser for small language.

**DESCRIPTION:** In this program, we try to implement a parser for LISP language.

**ALGORITHM:**

Step 1: Start by defining the grammar rules of the language that the parser will be parsing.

Step 2: Create a lexical analyzer (also known as a lexer or scanner) that will read in the input code and tokenize it according to the grammar rules.

Step 3: Create a parser that will use the tokens generated by the lexer to construct a parse tree. The parse tree represents the structural relationship between the different parts of the code.

Step 4: Use a stack-based approach to parse the input code. The parser will push the tokens onto the stack and use a set of rules to determine how to reduce the input code into a parse tree.

Step 5: Implement error handling mechanisms to detect and recover from syntax errors in the input code.

Step 6: Once the parse tree has been constructed, the parser may do additional processing to generate intermediate code or perform semantic analysis.

Step 7: Finally, the parser may output the result of its processing in some form, such as machine code or a high-level representation of the input code.

**PROGRAM:**

```
def generate_AST(string):
    number_symbols = ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9', '.', '-']
    ind = 1
    arr_to_return = []
    while ind < len(string):
        char = string[ind]
        if char == "(":
            open_cnt = 1
            closed_cnt = 0
            sub_str = "("
            for c in string[ind + 1:]:
                if c == "(": open_cnt += 1
                if c == ")": closed_cnt += 1
                sub_str += c
            if open_cnt == closed_cnt: break
            arr_to_return.append(generate_AST(sub_str))
            ind += len(sub_str)
        elif char == " " or char == ")":
            ind += 1
        else:
            stop_ind = string.find(" ", ind)
            if stop_ind == -1:
                stop_ind = string.find(")", ind)
```

```
s = string[ind:stop_ind]
if all(x in number_symbols for x in list(s)):
    if s.find('-', 1) == -1:
        num = float(s)
        arr_to_return.append(num)
    else:
        arr_to_return.append(s)
ind = stop_ind + 1
return arr_to_return
```

```
if __name__ == "__main__":
    ip = "(first (list -1.5 (+ 2 3) 9))" # lisp
    print("Input(LISP):", ip)
    print("Output (AST):", generate_AST(ip))
```

#### OUTPUT:

```
~/.../lab/cd-lablast % python lisp_ast.py
Input(LISP): (first (list -1.5 (+ 2 3) 9))
Output (AST): ['first', ['list', -1.5, ['+', 2.0, 3.0], 9.0]]
```

#### CONCLUSION:

By executing the above program, we have successfully implemented a parser for small language.