COMPILER DESIGN

EXPERIMENT-1

TASK-1

AIM: Program to find number of characters, spaces, lines, tabs in a given file

DESCRIPTION:

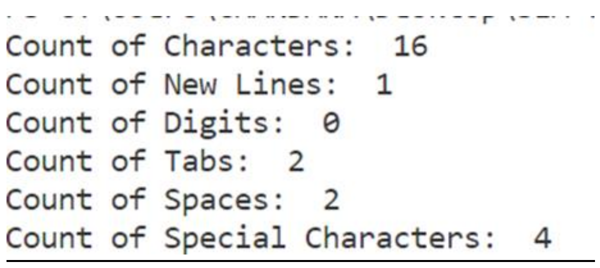
We need to read each character from the input file and analyse them and then categorize them into characters, spaces, tabs, etc. Scan the entire file and check for the relevant conditions and printthe count accordingly.

PROGRAM:

```
with open('token.txt','r') as f:
    text = f.read()
tabs,spaces,newlines,digits,alpha,special = 0,0,0,0,0,0
i = 0
while i<len(text):
    if i+4<len(text) and text[i:i+4]=='    ':
        tabs+=1
        i+=4
    elif text[i]==' ':
        spaces+=1
        i+=1
    elif text[i]=='\n':
        newlines+=1
        i+=1
    elif text[i].isdigit():
        digits+=1
        i+=1
    elif text[i].isalpha():
        alpha+=1
        i+=1
    else:
        special +=1
        i+=1

print("Count of Characters: ",alpha)
print("Count of New Lines: ",newlines)
print("Count of Digits: ",digits)
print("Count of Tabs: ",tabs)
print("Count of Spaces: ",spaces)
print("Count of Special Characters: ",special)
```

OUTPUT:

```
        good morning !

hello!   @@
```



```
Count of Characters:  16
Count of New Lines:  1
Count of Digits:  0
Count of Tabs:  2
Count of Spaces:  2
Count of Special Characters:  4
```

COMPILER DESIGN

EXPERIMENT-2

TASK-1

AIM: Develop a lexical analyser to recognize a few patterns in C. (Example: Identifiers, constants,comments, operators, etc.).

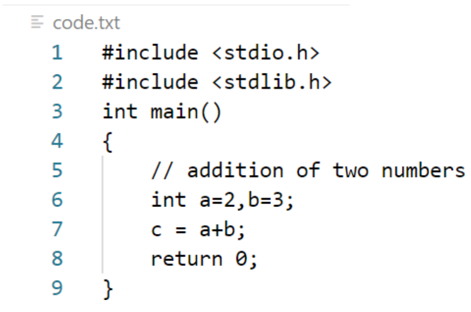
DESCRIPTION:

Get the input in the form of a C code from a file. It consists of set of tokens. A token is either a keyword, constant, operator, identifier, string literal, or a symbol. In lexical analysis the code isscanned from left to right and the tokens are identified.

PROGRAM:

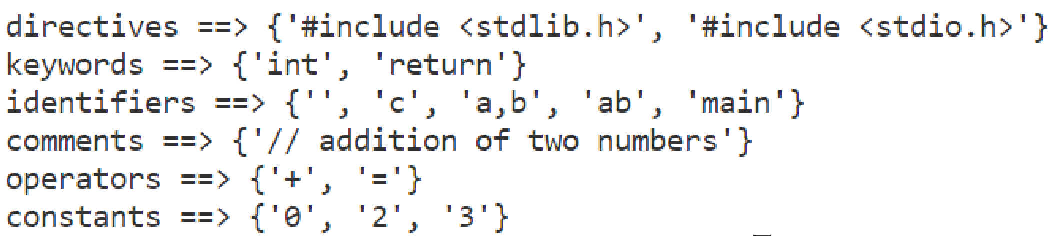
```
from collections import defaultdict
with open('code.txt') as f:
    s = f.readlines()

d = defaultdict(list)

keywords = 
['auto','printf','scanf','break','case','char','const','continue','default','
do','double','else','enum','extern','float','for','goto','if','int','long','r
egister','return','short','signed','sizeof','static','struct','switch','typed
ef','union','unsigned','void','volatile','while']
operators = ['+','-','*','/','%','=','&','|','^','!']

for i in s:
    i = i.strip()
    if i[0]=='#':
        d['directives'].append(i)
    elif i[0]=='/' and (i[1]=='/' or i[1]=='*'):
        d['comments'].append(i)
    else:
        i = i.split()

        for j in i:
            j = j.strip()
            st = []
            for k in j:
                if k.isalpha() or k==',':
                    st.append(k)
                elif k=='(':
                    st.append(k)
                elif k.isdigit():
                    d['constants'].append(k)
                elif k in operators:
                    d['operators'].append(k)
                elif k==')':
                    temp=''

                    while st[-1]!='(':
                        temp+=st.pop()
                    temp  =  temp[::-1]
                    if temp in keywords:
                        d['keywords'].append(temp)
                    else:
                        d['identifiers'].append(temp)
                    temp=''
                    st.pop()
            temp = ''

            while st:
                temp+=st.pop()
            temp  =  temp[::-1]
            if temp in keywords:
                d['keywords'].append(temp)
            else:
                d['identifiers'].append(temp)
            temp=''

for i,j in d.items():
    print(i,'==>',set(j))

f.close()
```

OUTPUT:



code.txt

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    // addition of two numbers
    int a=2,b=3;
    c = a+b;
    return 0;
}
```



```
directives ==> {'#include <stdlib.h>', '#include <stdio.h>'}
keywords ==> {'int', 'return'}
identifiers ==> {'', 'c', 'a,b', 'ab', 'main'}
comments ==> {'// addition of two numbers'}
operators ==> {'+', '='}
constants ==> {'0', '2', '3'}
```

TASK-2

AIM: Develop a lexical analyser to recognize a few patterns in Python. (Example: Identifiers, constants,comments, operators, etc.).

DESCRIPTION:

Get the input in the form of a Python code from a file. It consists of set of tokens. A tokenis either a keyword, constant, operator, identifier, string literal, or a symbol. In lexical analysis the codeis scanned from left to right and the tokens are identified.

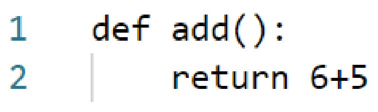
PROGRAM:

```
from collections import defaultdict
with open("code.txt",'r') as f:
    s = f.readlines()

n = len(s)
st = []
i = 0
d = defaultdict(list)
second = False

sepcial_symbols = ['(',')','{','}','[',']']
operators = ['+','-','*','/','%','&','|','^','<','>','=']
keywords =
['if','and','or','not','else','elif','return','class','pass','def','print']

for line in s:
    line = line.strip()
    i = 0
    n = len(line)
    while i<n:
        if line[i]=='(':
            st.append(line[i])
            d['special_characters'].append(line[i])

        elif '0'<=line[i]<='9':
            d['constants'].append(line[i])

        elif line[i].isalpha():
            st.append(line[i])

        elif line[i] in operators:
            d['operators'].append(line[i])
```

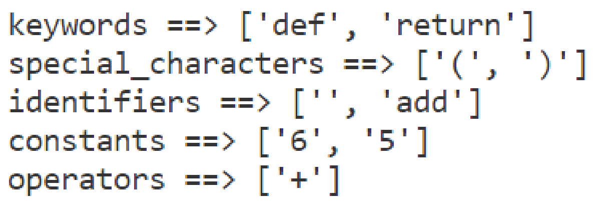
```
elif line[i]==')':
    d['special_characters'].append(line[i])
    temp = ''
    while st[-1]!='(':
        temp += st.pop()
    temp  =  temp[::-1]
    if temp in keywords:
        d['keywords'].append(temp)
    else:
        d['identifiers'].append(temp)
    st.pop()
elif line[i]==' ' or line[i]==':' or line[i]=='\n':
    temp = ''
    if second:
        while st[-1]!='"':
            temp += st.pop()
        temp = temp[::-1]
        d['strings'].append(temp)

    else:
        while st:
            temp += st.pop()

        temp = temp[::-1]

        if temp in keywords:
            d['keywords'].append(temp)

        else:
            d['identifiers'].append(temp)
elif line[i]=='"' and not second:
    st.append(line[i])
    second = True
elif line[i]=='"' and second:
    temp = ''

    while st[-1]!='"':
        temp+=st.pop()

    temp = temp[::-1]



    d['strings'].append(temp)
    second = False
    st.pop()

i+=1
```

```
        if st!=[]:
            temp = ''

            while st:
                temp += st.pop()

            temp = temp[::-1]

            if temp in keywords:
                d['keywords'].append(temp)

            else:
                d['identifiers'].append(temp)

    for i,j in d.items():

        print(i,'==>',j)
```

OUTPUT:



```
def add():
    return 6+5
```



```
keywords ==> ['def', 'return']
special_characters ==> ['(', ')']
identifiers ==> ['', 'add']
constants ==> ['6', '5']
operators ==> ['+']
```

COMPILER DESIGN

EXPERIMENT-3

TASK-1

AIM: Print the lexeme in a formatted order.

Example: Line Number | Token Number | Lexeme | Token

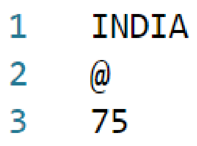
DESCRIPTION:

This program is to scan the program and print the tokens in a formatted order. Read theinput source program from the file and scan it from left to right.

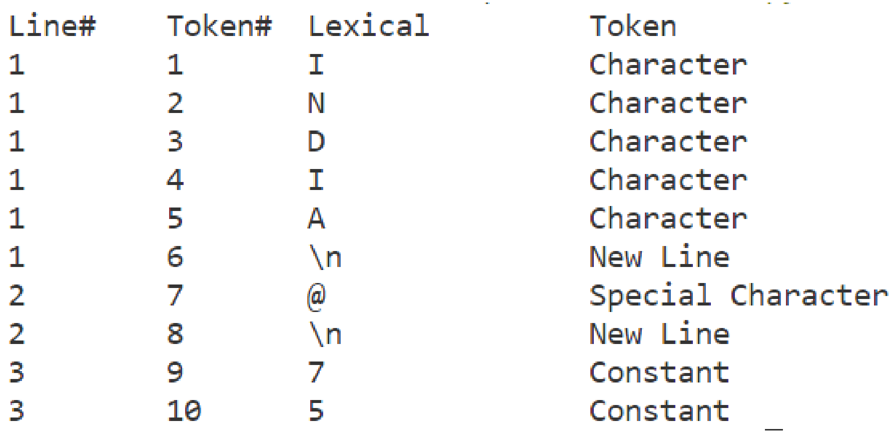
PROGRAM:

```
with open(code.txt','r') as f:
    s = f.readlines()

line_cnt = 0
c = 1

print("Line#\t Token#\t Lexical\t Token")

for line in s:
    i = 0
    line_cnt+=1
    n = len(line)
    for i in range(n):
        if line[i].isalpha():
            print(line_cnt,'\t',c,'\t',line[i],'\t\t','Character')
            c+=1
        elif line[i].isdigit():
            print(line_cnt,'\t',c,'\t',line[i],'\t\t','Constant')
            c+=1
        elif line[i] in ['+','-','*','/','%','^','>','<','=']:
            print(line_cnt,'\t',c,'\t',line[i],'\t\t','Operator')
            c+=1
        elif line[i]=='\n':
            print(line_cnt,'\t',c,'\t','\\n','\t\t','New Line')
            c+=1
        elif not line[i].isalnum() and not line[i]==' ':
            print(line_cnt,'\t',c,'\t',line[i],'\t\t','Special Character')
            c+=1

f.close()
```

OUTPUT:



```
1    INDIA
2    @
3    75
```



```
Line#     Token#   Lexical          Token
1         1        I                Character
1         2        N                Character
1         3        D                Character
1         4        I                Character
1         5        A                Character
1         6        \n               New Line
2         7        @                Special Character
2         8        \n               New Line
3         9        7                Constant
3         10       5                Constant
```

COMPILER DESIGN

EXPERIMENT-4

TASK-1

AIM: Write a program to print count of words in a sentence in Lex.

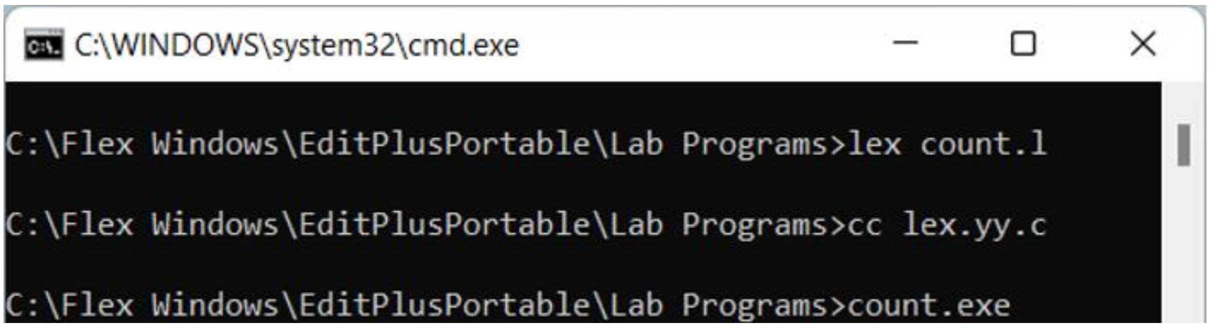
DESCRIPTION:

Lexical analysis is done with the help of lex tool. Here we will count number of wordsfrom a given sentence and return the count of words. Scan the entire string from left to right until space is encountered.

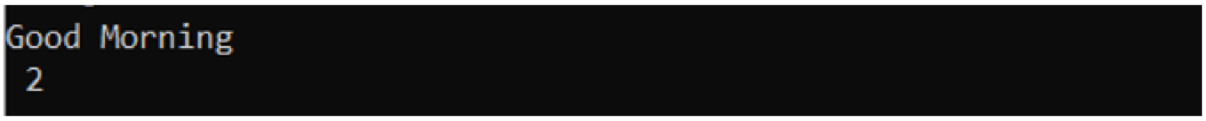
PROGRAM:

```
%{
#include <stdio.h>
#include <string.h>
int i = 0;
%}
%%
([a-zA-Z0-9])* {i++;}
"\n" {printf("%d\n",i);i=0;}
%%

int yywrap(void){}

int main()
{
yylex();
return 0;
}
```

OUTPUT:



```
C:\WINDOWS\system32\cmd.exe

C:\Flex Windows\EditPlusPortable\Lab Programs>lex count.l

C:\Flex Windows\EditPlusPortable\Lab Programs>cc lex.yy.c

C:\Flex Windows\EditPlusPortable\Lab Programs>count.exe
```



```
Good Morning
 2
```

TASK-2

AIM: Write a program to print capital letters in a string in Lex

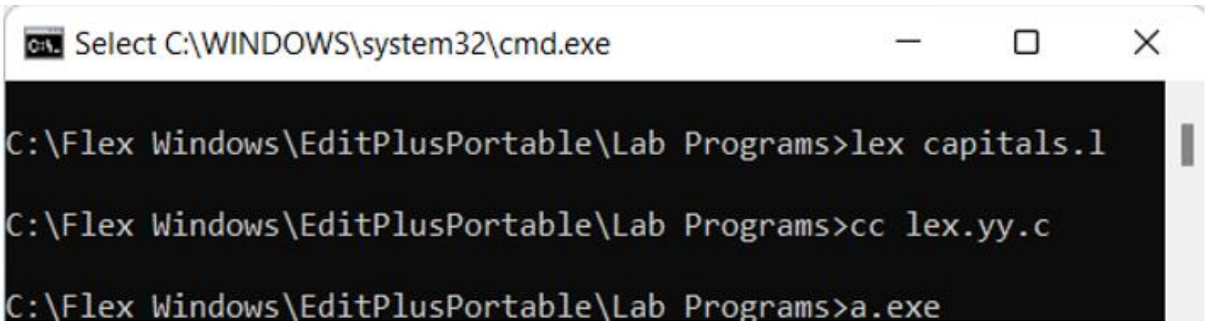
DESCRIPTION:

Lexical analysis to find capital letters in the string and print them.

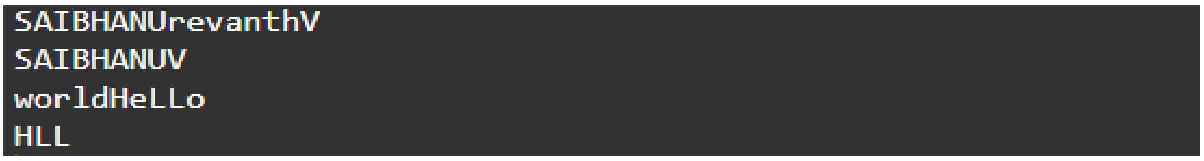
PROGRAM:

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

%%
[A-Z]+ { printf("%s",yytext); }
. ;
%%
int yywrap(void){}
int main()
{
yylex();
return 0;
}
```

OUTPUT:



```
C:\Flex Windows\EditPlusPortable\Lab Programs>lex capitals.l

C:\Flex Windows\EditPlusPortable\Lab Programs>cc lex.yy.c

C:\Flex Windows\EditPlusPortable\Lab Programs>a.exe
```



```
SAIBHANUrevanthV
SAIBHANUV
worldHeLLo
HLL
```

TASK-3

AIM: Write a program to convert a decimal number into octal number in Lex

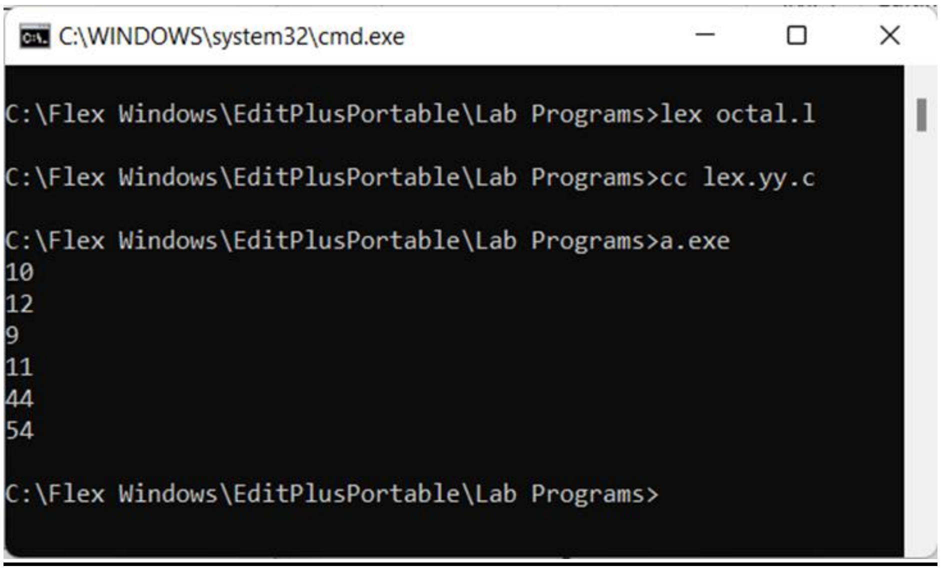
DESCRIPTION:

The octal system is a base-8 system and uses the digits from 0 to 7 to represent anynumber.

PROGRAM:

```
%{
    #include<stdio.h>
    int num, r, digit=0, count, pcount=0, i;
    char a[20];
%}

DIGIT [0-9]

%%

{DIGIT}+ { num=atoi(yytext);

        while(num!=0)
        {

            r=num%8;
            digit='0'+r;
            a[count++]=digit;
            num=num/8;

        }

        for(i=count-1;i>=pcount;--i)
                printf("%c", a[i]);
                pcount=count;
        }

.|\n ECHO;

%%
int yywrap(void){}
int main()
{
    yylex();
    return 0;
}
```

OUTPUT:



```
C:\Flex Windows\EditPlusPortable\Lab Programs>lex octal.l

C:\Flex Windows\EditPlusPortable\Lab Programs>cc lex.yy.c

C:\Flex Windows\EditPlusPortable\Lab Programs>a.exe
10
12
9
11
44
54

C:\Flex Windows\EditPlusPortable\Lab Programs>
```

TASK-4

AIM: Write a program to convert decimal number into hexadecimal number using Lex.

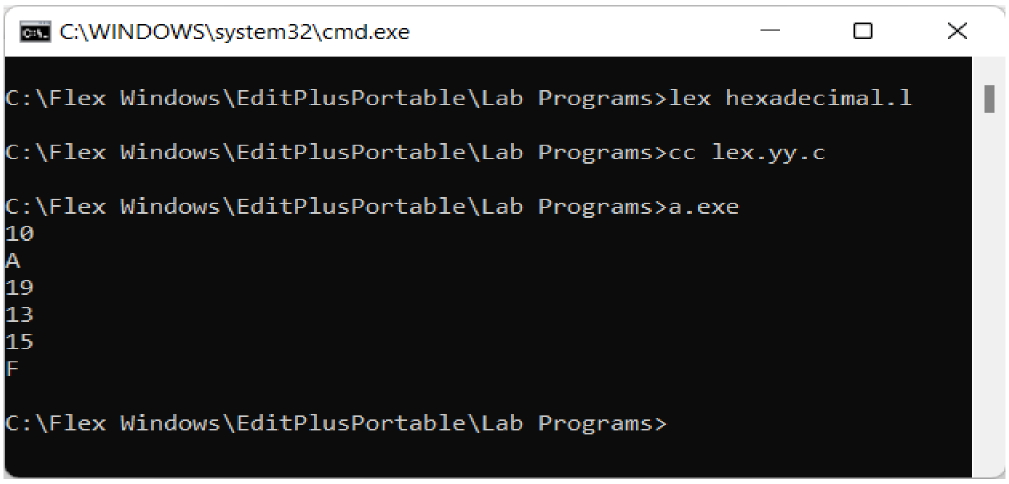
DESCRIPTION:

The hexadecimal number system uses base-16, where all the numbers are in the range of 0-15. Any decimal number is represented in this range

PROGRAM:

```
%{
    #include<stdio.h>
    int num, r, digit=0, count, pcount=0, i;
    char a[20];
%}

DIGIT [0-9]
%%

{DIGIT}+ { num=atoi(yytext);

        while(num!=0)
        {

            r=num%16;
            digit='0'+r;
            if(digit>'9')
            digit+=7;
            a[count++]=digit;
            num=num/16;

        }

        for(i=count-1;i>=pcount;--i)
                printf("%c", a[i]);
                pcount=count;
        }

.|\n ECHO;

%%
int yywrap(void){}
int main()
{
    yylex();
    return 0;
}
```

OUTPUT:


```
C:\Flex Windows\EditPlusPortable\Lab Programs>lex hexadecimal.l

C:\Flex Windows\EditPlusPortable\Lab Programs>cc lex.yy.c

C:\Flex Windows\EditPlusPortable\Lab Programs>a.exe
10
A
19
13
15
F

C:\Flex Windows\EditPlusPortable\Lab Programs>
```

TASK-5

AIM: Write a lex program to print list of tokens.

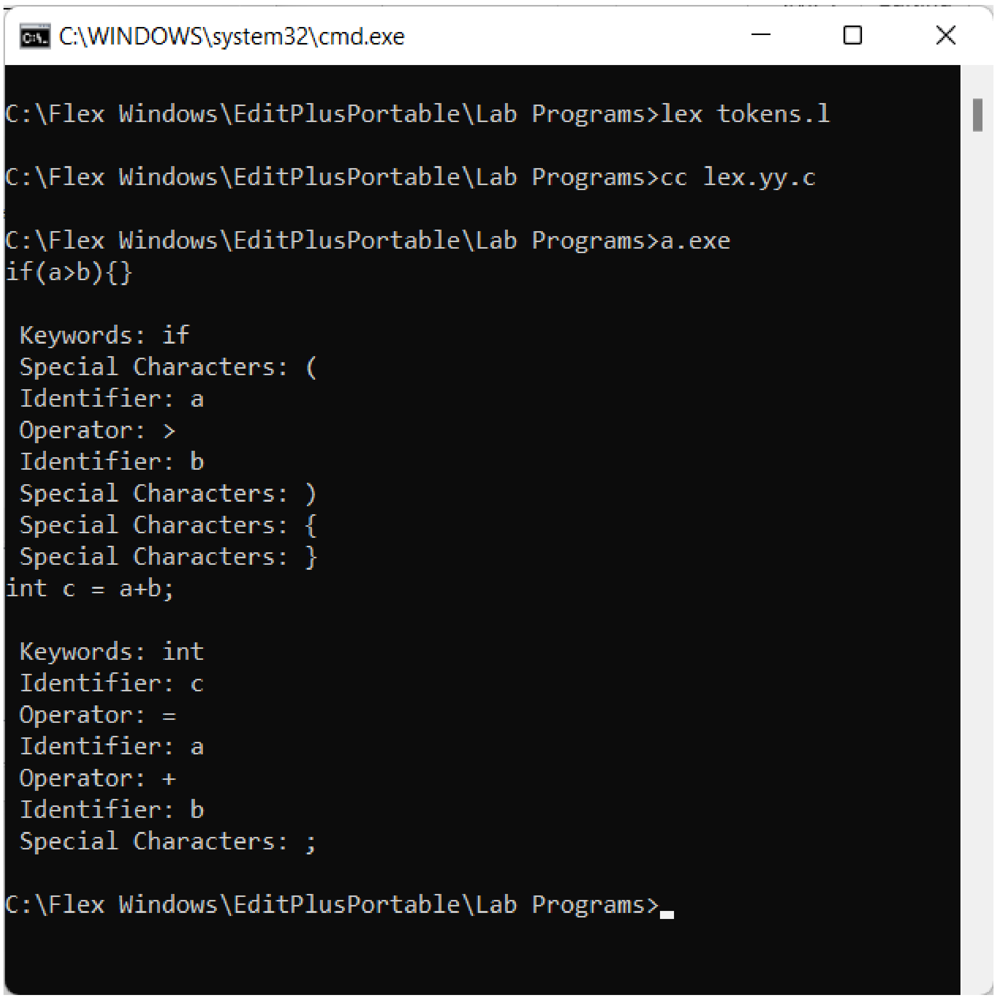
DESCRIPTION:

Token can be an identifier, constant, keyword, string literal, etc. Scan the code from left toright and perform lexical analysis which returns a set of tokens

PROGRAM:

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

%%

"While"|"if"|"else"|"int"|"float"|"return"|"printf"|"scanf" {n++; printf("\n
Keywords: %s",yytext);}

[a-zA-Z_][a-zA-Z_0-9]* {n++; printf("\n Identifier: %s",yytext);}

"<="|">"|"<"|"="|">="|"+"|"-"|"*"|"/" {n++; printf("\n Operator:
%s",yytext);}

[(){}|,;] {n++; printf("\n Special Characters: %s",yytext);}

[0-9]*"."[0-9]+ {n++; printf("\n Float: %s",yytext);}

[0-9]+ {n++; printf("\n Interger: %s", yytext);}

. ;

%%

int yywrap(void){}

int main()

{
yylex();
return 0;
}
```

OUTPUT:



```
C:\Flex Windows\EditPlusPortable\Lab Programs>lex tokens.l

C:\Flex Windows\EditPlusPortable\Lab Programs>cc lex.yy.c

C:\Flex Windows\EditPlusPortable\Lab Programs>a.exe
if(a>b){}

 Keywords: if
 Special Characters: (
 Identifier: a
 Operator: >
 Identifier: b
 Special Characters: )
 Special Characters: {
 Special Characters: }
int c = a+b;

 Keywords: int
 Identifier: c
 Operator: =
 Identifier: a
 Operator: +
 Identifier: b
 Special Characters: ;

C:\Flex Windows\EditPlusPortable\Lab Programs>
```

COMPILER DESIGN

EXPERIMENT-5

TASK-1

AIM: Write a program to compute FIRST, FOLLOW functions, and Predictive Parsing Table for a givengrammar.

DESCRIPTION:

FIRST(X) for a grammar symbol X is the set of terminals that begin the strings derivable from X. Rules to compute FIRST set: If x is a terminal, then FIRST(x) = { 'x' } If x-> Є, is a productionrule, then add Є to FIRST(x).

Follow(X) to be the set of terminals that can appear immediately to the right of Non-Terminal X in somesentential form. In RHS of A -> aBb, b follows Non-Terminal B, i.e., FOLLOW(B) = {b}, and the current input character read is also b. Hence the parser applies this rule. And it is able to get the string “ab” from the given grammar.

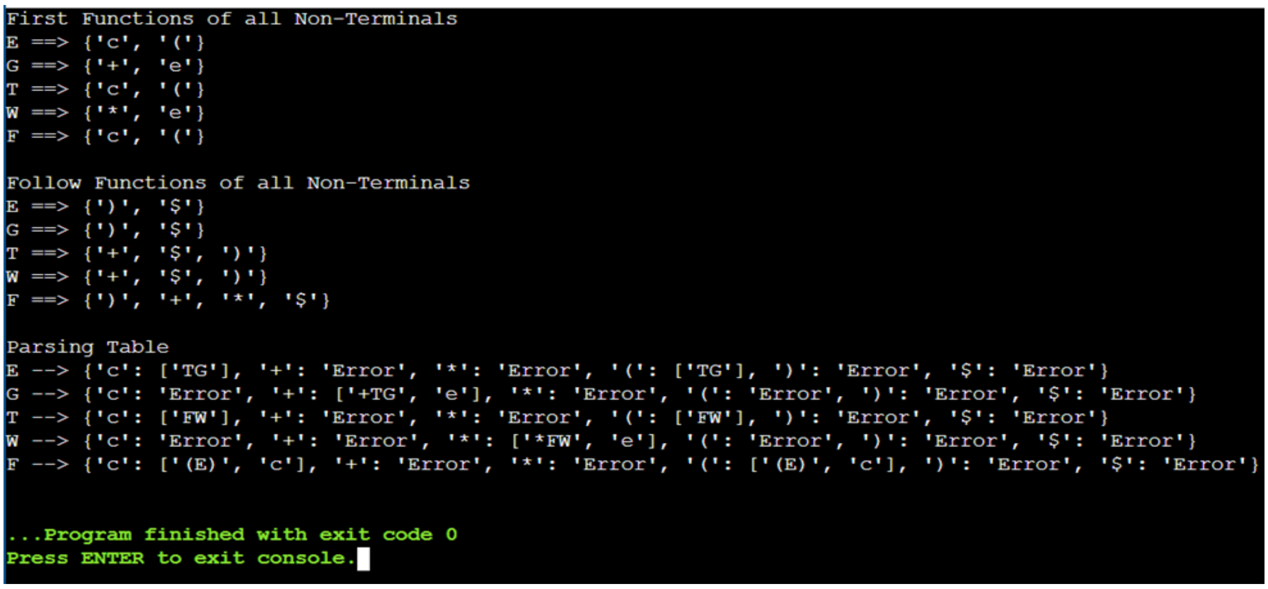
Here the 1st L represents that the scanning of the Input will be done from Left to Right manner and the second L shows that in this parsing technique we are going to use Left most Derivation Tree. And finally, the 1 represents the number of look-ahead, which means how many symbols are you going to seewhen you want to make a decision.

PROGRAM:

```
    from collections import defaultdict

    d = defaultdict(list)

    f = defaultdict(list)

    fo = defaultdict(list)

    p = defaultdict(dict)

    d = {'E':['TG'],'G':['+TG','e'],'T':['FW'],'W':['*FW','e'],'F':['(E)','c']}
#  First  Functions


    for i,j in d.items():
        z = i
        k = 0
        while k<len(d[z]):
            if 'a'<=d[z][k][0]<='z' or d[z][k][0] in '(+)*' or d[z][k][0]=='e':
                f[i].append(d[z][k][0])
                k+=1
            else:
                z=d[z][k][0]
                k = 0




    # Follow Functions

    for z in d.keys():
        for i,j in d.items():
            for k in d[i]:

                if z in k:
                    if z=='E':
                        fo['E'].extend('$')
                    n = k.index(z)+1
                    if n<len(k):
                        if k[n] in '()+*' or ('a'<=k[n]<='z' and k[n]!='e'):
                            fo[z].extend(k[n])
                        if k[n]=='e':
                            fo[z].extend(fo[i])
                        elif 'A'<=k[n]<='Z':
                            fo[z].extend([i for i in f[k[n]] if i!='e'])
                            fo[z].extend(fo[k[n]])
                    else:
                        if z!=i:
                            fo[z].extend(fo[i])


    print("First Functions of all Non-Terminals")
    for i,j in f.items():
        print(i,'==>',set(j))

    print("\nFollow Functions of all Non-Terminals")
    for i,j in fo.items():
        print(i,'==>',set(j))

    # Parsing Table
```

```
        # Parsing Table

    print("\nParsing Table")
    for i in d.keys():
        k = f[i]
        if 'c' in k:
            p[i].update({'c':d[i]})
        else:
            p[i].update({'c':"Error"})
        if '+' in k:
            p[i].update({'+':d[i]})
        else:
            p[i].update({'+':"Error"})
        if '*' in k:
            p[i].update({'*':d[i]})
        else:
            p[i].update({'*':"Error"})
        if '(' in k:
            p[i].update({'(':d[i]})
        else:
            p[i].update({'(':"Error"})
        if ')' in k:
            p[i].update({')':d[i]})
        else:
            p[i].update({')':"Error"})
        if '$' in k:

            for m in fo[k]:
                p[i].update({'m':d[i]})
        else:
            p[i].update({'$':"Error"})

    for i,j in p.items():
        print(i,'-->',j)
```

OUTPUT:



```
First Functions of all Non-Terminals
E ==> {'c', '('}
G ==> {'+', 'e'}
T ==> {'c', '('}
W ==> {'*', 'e'}
F ==> {'c', '('}

Follow Functions of all Non-Terminals
E ==> {')', '$'}
G ==> {')', '$'}
T ==> {'+', '$', ')'}
W ==> {'+', '$', ')'}
F ==> {')', '+', '*', '$'}

Parsing Table
E --> {'c': ['TG'], '+': 'Error', '*': 'Error', '(': ['TG'], ')': 'Error', '$': 'Error'}
G --> {'c': 'Error', '+': ['+TG', 'e'], '*': 'Error', '(': 'Error', ')': 'Error', '$': 'Error'}
T --> {'c': ['FW'], '+': 'Error', '*': 'Error', '(': ['FW'], ')': 'Error', '$': 'Error'}
W --> {'c': 'Error', '+': 'Error', '*': ['*FW', 'e'], '(': 'Error', ')': 'Error', '$': 'Error'}
F --> {'c': ['(E)', 'c'], '+': 'Error', '*': 'Error', '(': ['(E)', 'c'], ')': 'Error', '$': 'Error'}


...Program finished with exit code 0
Press ENTER to exit console.
```

TASK-2

AIM: Write a program to implement RD parser.

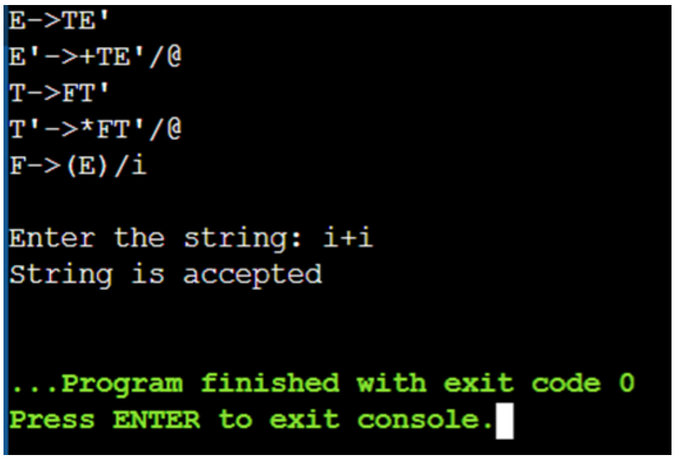
DESCRIPTION:

A parser that uses collection of recursive procedures for parsing the given input string iscalled Recursive descent (RD) parser

PROGRAM:

```
print("E->TE'\nE'->+TE'/@\nT->FT'\nT'->*FT'/@\nF->(E)/i\n")
global s
s=list(input(“Enter the string: “))
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:



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

Enter the string: i+i
String is accepted


...Program finished with exit code 0
Press ENTER to exit console.
```

COMPILER DESIGN

EXPERIMENT-6

TASK-1

AIM: Write a program to implement Shift Reduce parser.

DESCRIPTION:

Shift Reduce parser attempts for the construction of parse in a similar manner as done in bottom-up parsing i.e the parse tree is constructed from leaves(bottom) to the root(up). A more general form of the shift-reduce parser is the LR parser. This parser requires some data structures i.e.

- An input buffer for storing the input string.
- A stack for storing and accessing the production rules.

PROGRAM:

```
def printPattern(s,i,action):
    print("\t{}\t\t{}\t\t{}\t".format(s,i,action))
def shiftReduce():
    global ip;
    global stack;
    if stack == "$"+starting_symbol and ip=="$":
        printPattern(stack,ip,"Reduce")
        print("String is Accepted");
        return;
    n = len(stack)
    flag = False
    for i in range(n-1,-1,-1):
        for (nt,rhs) in productions_dict.items():
            if stack[i:] in rhs:
                printPattern(stack,ip,"Reduce")
                stack = stack[0:i]+nt;
                flag = True;
                break;
        if flag:
            break;

    if flag==False:
        if ip=="$":
            printPattern(stack,ip,"No Operation")
            print("String is Not Accepted")
            return
        else:
            printPattern(stack,ip,"Shift")
            stack = stack + ip[0];
            ip = ip[1:];
    shiftReduce();

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())
```

```
print("Enter the I/P String:");
ip = 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)

printPattern("Stack","I/P String","Action")
stack = "$"
ip = ip + "$"
printPattern(stack,ip,"Shift");
stack = "$"+ip[0]
ip = ip[1:]
shiftReduce();
```

OUTPUT:

```
Enter no. of terminals: 3
Enter the terminals :
*
+
i
Enter no. of non terminals: 1
Enter the non terminals :
S
Enter the starting symbol: S
Enter no of productions: 3
Enter the productions:
S->S+S
S->S*S
S->i
Enter the I/P String:
i+i+i
```

```
            Stack           I/P String          Action
            $               i+i+i$              Shift
            $i              +i+i$               Reduce
            $S              +i+i$               Shift
            $S+             i+i$                Shift
            $S+i            +i$                 Reduce
            $S+S            +i$                 Reduce
            $S              +i$                 Shift
            $S+             i$                  Shift
            $S+i            $                   Reduce
            $S+S            $                   Reduce
            $S              $                   Reduce
String is Accepted
```

COMPILER DESIGN

EXPERIMENT-7

TASK-1

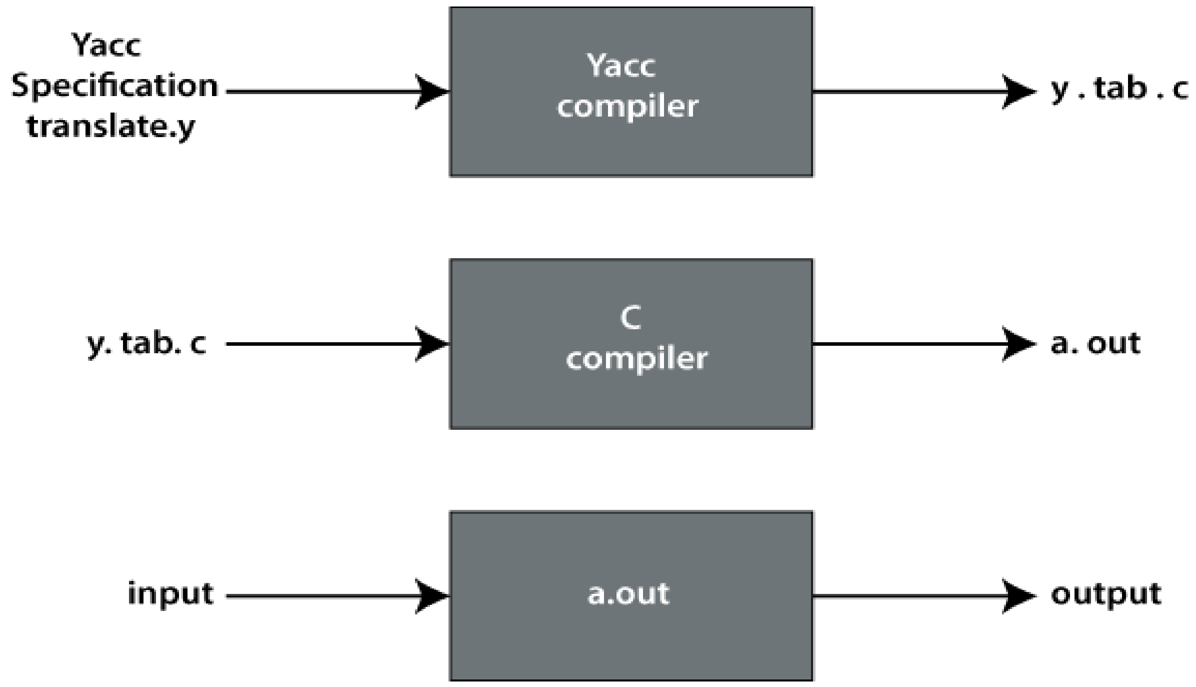
AIM: Write a program to recognize a valid arithmetic expression that uses operator +, –, * and /. (LEX and YACC)

DESCRIPTION:

YACC is known as Yet Another Compiler-Compiler. It is used to produce the source code of the syntactic analyser of the language produced by LALR (1) grammar. The input of YACC is the rule or grammar, and the output is a C program. If we have a file translate.y that consists of YACC specification, then the UNIX system command is:

YACC translate.y

This command converts the file translate.y into a C file y.tab.c. It represents an LALR parser prepared in C with some other user's prepared C routines. By compiling y.tab.c along with the ly library, we will get the desired object program a.out that performs the operation defined by the original YACC program.



Lex is a lexical analysis tool that can be used to identify specific text strings in a structured way from source text. Yacc is a grammar parser; it reads text and can be used to turn a sequence of words into a structured format for processing.

PROGRAM:

valid.lex

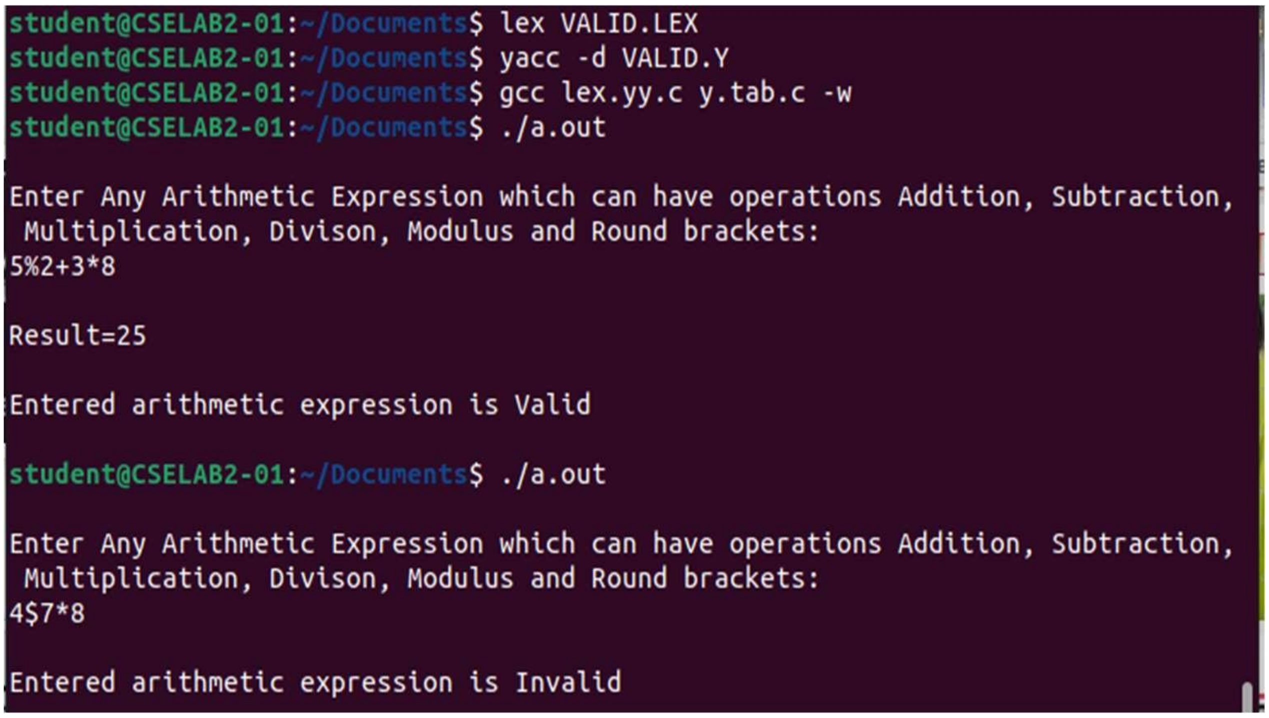
```
%{
#include<stdio.h>
#include "y.tab.h"
extern int yylval;
%}
%%
[0-9]+ {
yylval=atoi(yytext);
return NUMBER;
}
[\t];
[\n] return 0;
. return yytext[0];
%%
int yywrap()
{
return 1;
}
```

valid.y

```
%{
#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:



```
student@CSELAB2-01:~/Documents$ lex VALID.LEX
student@CSELAB2-01:~/Documents$ yacc -d VALID.Y
student@CSELAB2-01:~/Documents$ gcc lex.yy.c y.tab.c -w
student@CSELAB2-01:~/Documents$ ./a.out

Enter Any Arithmetic Expression which can have operations Addition, Subtraction,
 Multiplication, Divison, Modulus and Round brackets:
5%2+3*8

Result=25

Entered arithmetic expression is Valid

student@CSELAB2-01:~/Documents$ ./a.out

Enter Any Arithmetic Expression which can have operations Addition, Subtraction,
 Multiplication, Divison, Modulus and Round brackets:
4$7*8

Entered arithmetic expression is Invalid
```

TASK-2

AIM: Write a program to recognize a valid variable which starts with a letter followed by any Number of letters or digits. (LEX and YACC).

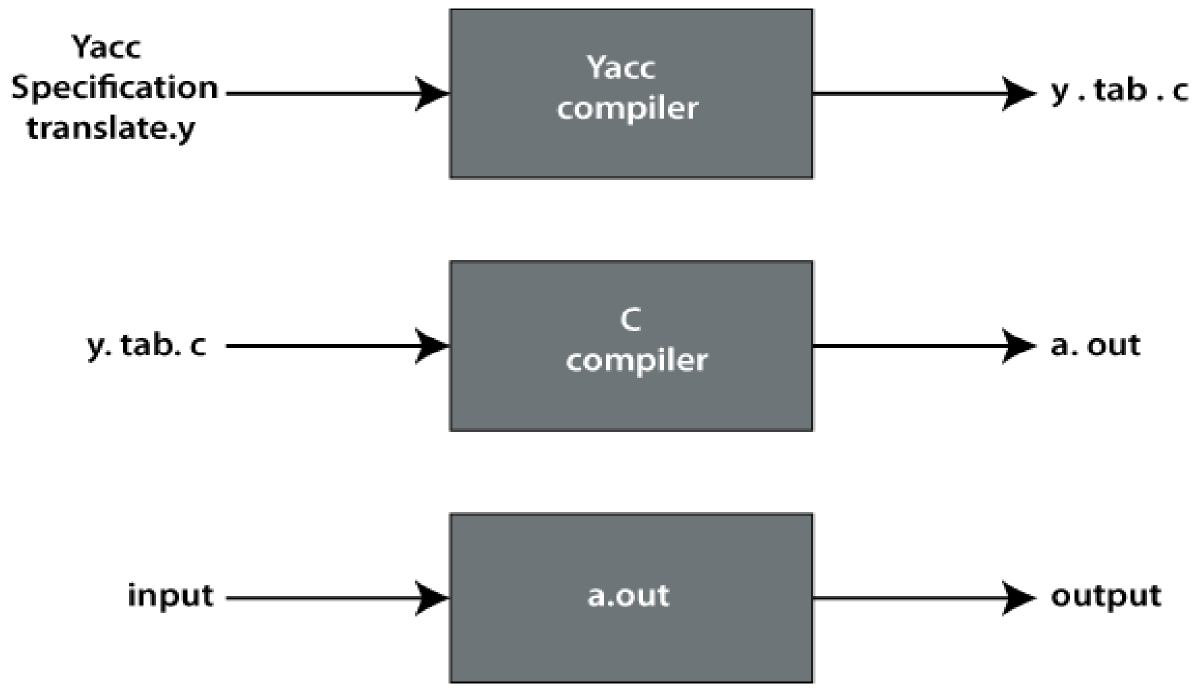
DESCRIPTION:

YACC is known as Yet Another Compiler-Compiler. It is used to produce the source code of the syntactic analyser of the language produced by LALR (1) grammar. The input of YACC is the rule or grammar, and the output is a C program. If we have a file translate.y that consists of YACC specification, then the UNIX system command is:

YACC translate.y

This command converts the file translate.y into a C file y.tab.c. It represents an LALR parser prepared in C with some other user's prepared C routines. By compiling y.tab.c along with the ly library, we will get the desired object program a.out that performs the operation defined by the original YACC program.

Lex is a lexical analysis tool that can be used to identify specific text strings in a structured way from source text. Yacc is a grammar parser; it reads text and can be used to turn a sequence of words into a structured format for processing.



PROGRAM:

Letter.y

```
%{

#include<stdio.h>

int valid=1;

%}

%token digit letter

%%

start : letter s

s : letter s

| digit s

|

;

%%

int yyerror()

{
```

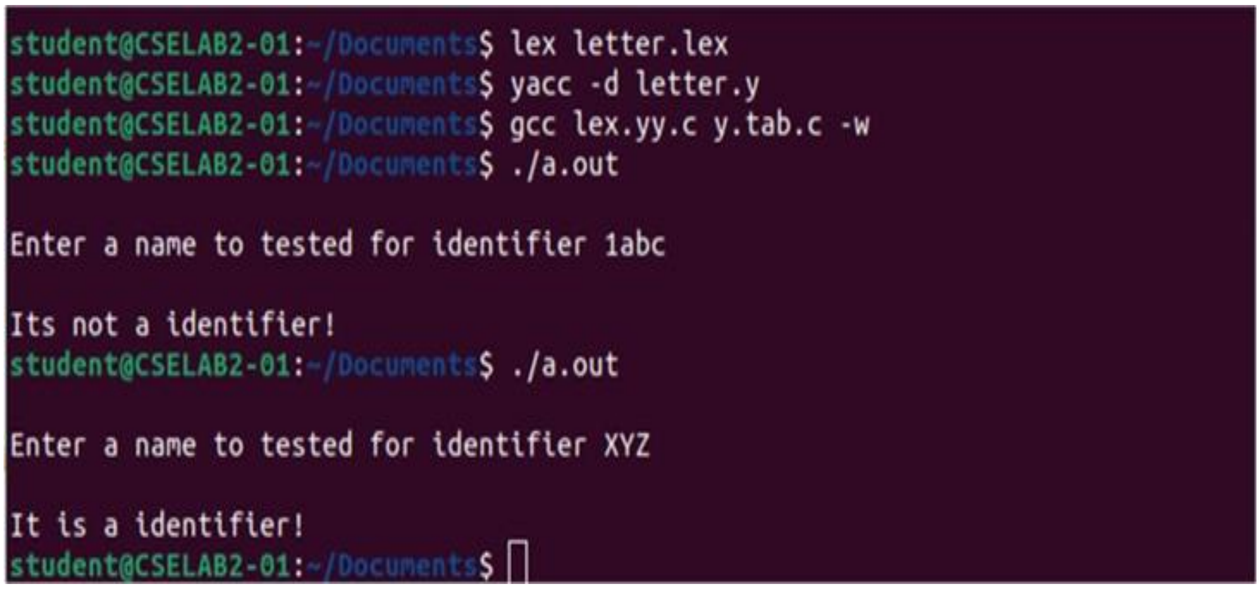
```
printf("\nIts not a identifier!\n");

valid=0;

return 0;

}

int main()

{

printf("\nEnter a name to tested for identifier ");

yyparse();

if(valid)

{

printf("\nIt is a identifier!\n");

}

}
```

<u>Letter.lex</u>

```
%{

#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;

}
```

OUTPUT:



```
student@CSELAB2-01:~/Documents$ lex letter.lex
student@CSELAB2-01:~/Documents$ yacc -d letter.y
student@CSELAB2-01:~/Documents$ gcc lex.yy.c y.tab.c -w
student@CSELAB2-01:~/Documents$ ./a.out

Enter a name to tested for identifier 1abc

Its not a identifier!
student@CSELAB2-01:~/Documents$ ./a.out

Enter a name to tested for identifier XYZ

It is a identifier!
student@CSELAB2-01:~/Documents$ 
```

TASK-3

AIM: Write a program to demonstrate Calculator using LEX and YACC tool.

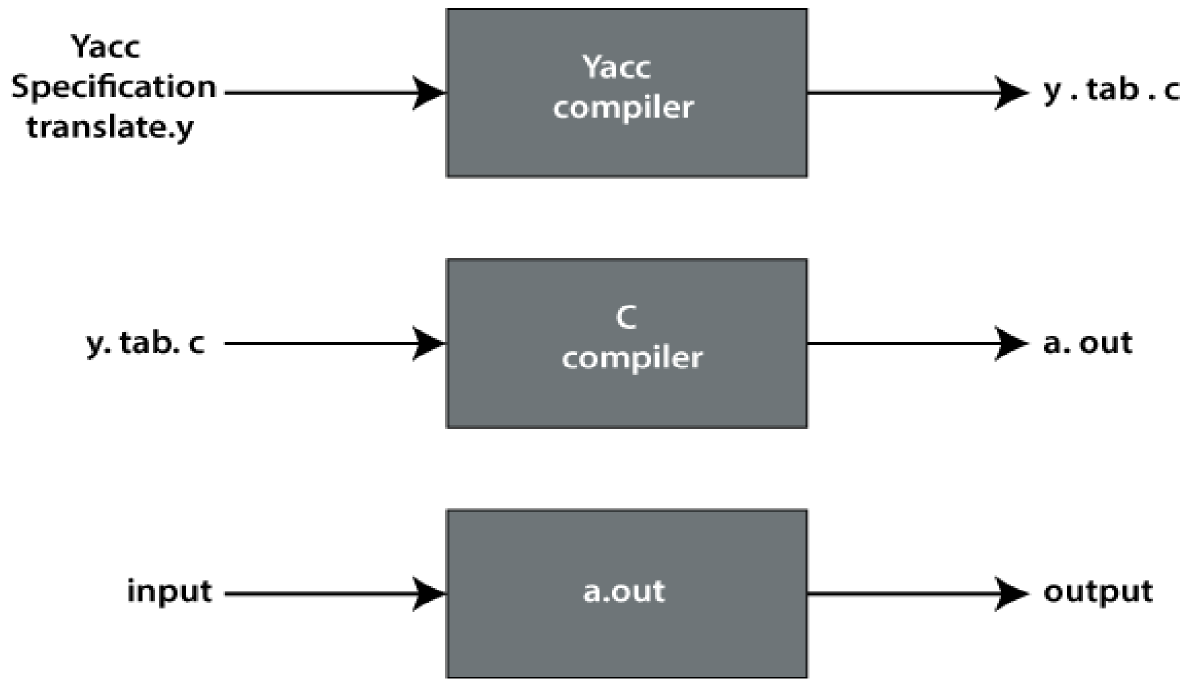
DESCRIPTION:

YACC is known as Yet Another Compiler-Compiler. It is used to produce the source code of the syntactic analyser of the language produced by LALR (1) grammar. The input of YACC is the rule or grammar, and the output is a C program. If we have a file translate.y that consists of YACC specification, then the UNIX system command is:

YACC translate.y

This command converts the file translate.y into a C file y.tab.c. It represents an LALR parser prepared in C with some other user's prepared C routines. By compiling y.tab.c along with the ly library, we will get the desired object program a.out that performs the operation defined by the original YACC program.

Lex is a lexical analysis tool that can be used to identify specific text strings in a structured way from source text. Yacc is a grammar parser; it reads text and can be used to turn a sequence of words into a structured format for processing.



PROGRAM:

Calc.lex

```
%{

#include<stdio.h>

#include "y.tab.h"

extern int yylval;

%}

%%

[0-9]+ {

yylval=atoi(yytext);

return NUMBER;

}

[\t];

[\n] return 0;

. return yytext[0];

%%
```

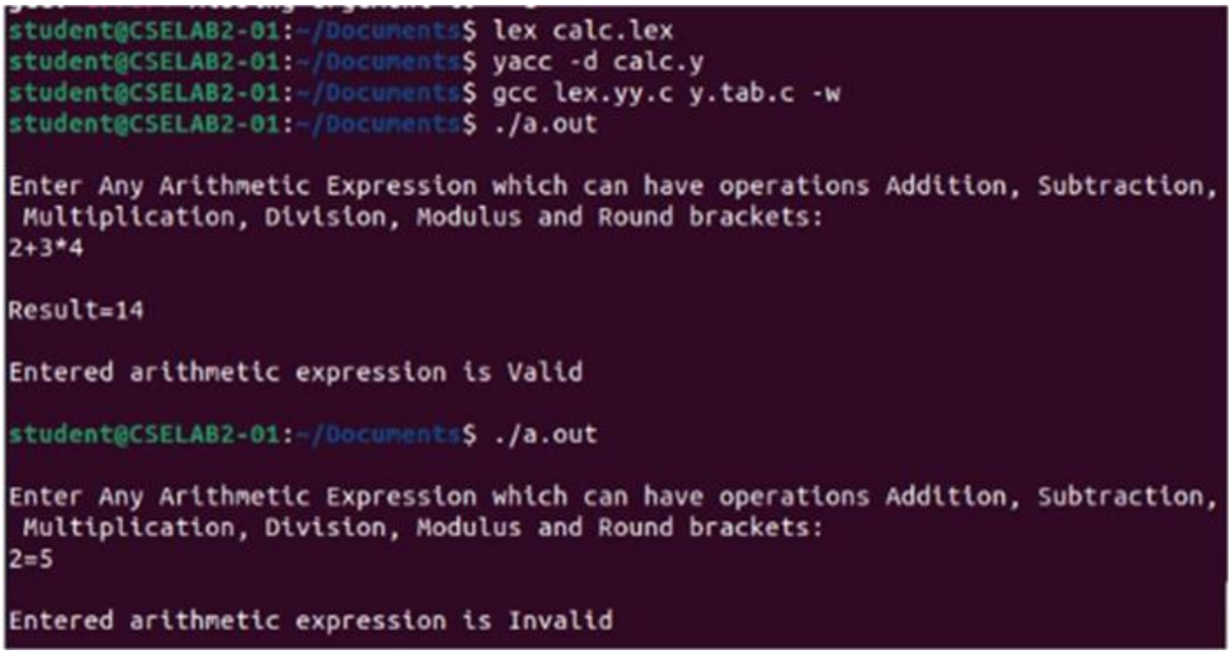
```
int yywrap()

{

return 1;

}
```

Calc.y

```
%{

#include<stdio.h>

int flag=0;

%}

%token NUMBER

%left '+' '-'

%left '*' '/' '%'

%left '(' ')'

%%

ArithmeticExpression: E{

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, Division, 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:



```
student@CSELAB2-01:~/Documents$ lex calc.lex
student@CSELAB2-01:~/Documents$ yacc -d calc.y
student@CSELAB2-01:~/Documents$ gcc lex.yy.c y.tab.c -w
student@CSELAB2-01:~/Documents$ ./a.out

Enter Any Arithmetic Expression which can have operations Addition, Subtraction,
 Multiplication, Division, Modulus and Round brackets:
2+3*4

Result=14

Entered arithmetic expression is Valid

student@CSELAB2-01:~/Documents$ ./a.out

Enter Any Arithmetic Expression which can have operations Addition, Subtraction,
 Multiplication, Division, Modulus and Round brackets:
2=5

Entered arithmetic expression is Invalid
```

COMPILER DESIGN

EXPERIMENT-8

TASK-1

AIM: Write a program to generate LR(0) items for the given grammar.

DESCRIPTION:

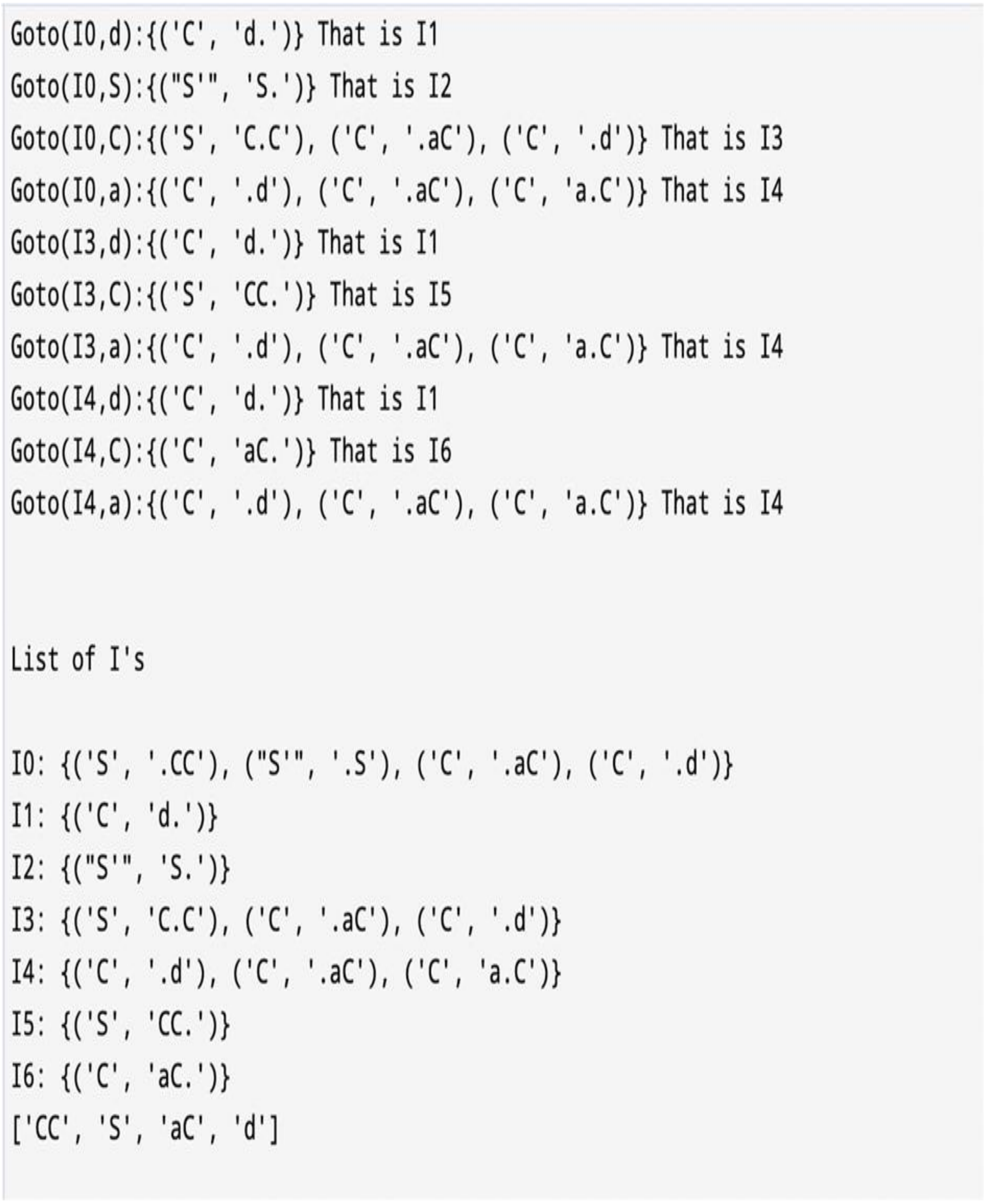
The LR parser is an efficient bottom-up syntax analysis technique that can be used for a large class of context-free grammar. This technique is also called LR(0) parsing.
L stands for the left to right scanning
R stands for rightmost derivation in reverse
0 stands for no. of input symbols of lookahead.
An LR (0) item is a production G with dot at some position on the right side of the production. LR(0) items is useful to indicate that how much of the input has been scanned up to a given point in the process of parsing. In the LR (0), we place the reduce node in the entire row.

PROGRAM:

```
gram = {
        "S":["CC"],
        "C":["aC","d"]
}
start = "S"
terms = ["a","d","$"]

non_terms = []
for i in gram:
        non_terms.append(i)
gram["S'"]= [start]


new_row = {}
for i in terms+non_terms:
        new_row[i]=""


non_terms += ["S'"]
# each row in state table will be dictionary {nonterms ,term,$}
stateTable = []
# I = [(terminal, closure)]
# I = [("S","A.A")]

def Closure(term, I):
        if term in non_terms:
                for i in gram[term]:
                        I+=[(term,"."+i)]
        I = list(set(I))
        for i in I:
                # print("." != i[1][-1],i[1][i[1].index(".")+1])
                if "." != i[1][-1] and i[1][i[1].index(".")+1] in non_terms and
i[1][i[1].index(".")+1] != term:
                        I += Closure(i[1][i[1].index(".")+1], [])
        return I

Is = []
Is+=set(Closure("S'", []))


countI = 0
omegaList = [set(Is)]
while countI<len(omegaList):
        newrow = dict(new_row)
        vars_in_I = []
        Is = omegaList[countI]
        countI+=1
        for i in Is:
                if i[1][-1]!=".":
                        indx = i[1].index(".")
                        vars_in_I+=[i[1][indx+1]]
        vars_in_I = list(set(vars_in_I))
        # print(vars_in_I)
        for i in vars_in_I:
                In = []
                for j in Is:
                        if "."+i in j[1]:
                                rep = j[1].replace("."+i,i+".")
                                In+=[(j[0],rep)]
                if (In[0][1][-1]!="."):
                        temp = set(Closure(i,In))
                        if temp not in omegaList:
                                omegaList.append(temp)
                        if i in non_terms:
                                newrow[i] = str(omegaList.index(temp))
                        else:
                                newrow[i] = "s"+str(omegaList.index(temp))
                        print(f'Goto(I{countI-1},{i}):{temp}                That                is
I{omegaList.index(temp)}')
                else:
                        temp = set(In)
                        if temp not in omegaList:
                                omegaList.append(temp)
                        if i in non_terms:
                                newrow[i] = str(omegaList.index(temp))
                        else:
                                newrow[i] = "s"+str(omegaList.index(temp))
                        print(f'Goto(I{countI-1},{i}):{temp}                That                is
I{omegaList.index(temp)}')

        stateTable.append(newrow)
print("\n\nList of I's\n")
for i in omegaList:
        print(f'I{omegaList.index(i)}: {i}')


#populate replace elements in state Table
I0 = []
for i in list(omegaList[0]):
        I0 += [i[1].replace(".","")]
print(I0)

for i in omegaList:
        for j in i:
                if "." in j[1][-1]:
                        if j[1][-2]=="S":
                                stateTable[omegaList.index(i)]["$"] = "Accept"
                                break
                        for k in terms:
                                stateTable[omegaList.index(i)][k]                              =
"r"+str(I0.index(j[1].replace(".","")))
print("\nStateTable")

print(f'{" ": <9}',end="")
for i in new_row:
        print(f'|{i: <11}',end="")

print(f'\n{"-":-<66}')
```

```
for i in stateTable:
        print(f'{"I("+str(stateTable.index(i))+")": <9}',end="")
        for j in i:
                print(f'|{i[j]: <10}',end=" ")
        print()
```

OUTPUT:



```
Goto(I0,d):{('C', 'd.')} That is I1
Goto(I0,S):{("S'", 'S.')} That is I2
Goto(I0,C):{('S', 'C.C'), ('C', '.aC'), ('C', '.d')} That is I3
Goto(I0,a):{('C', '.d'), ('C', '.aC'), ('C', 'a.C')} That is I4
Goto(I3,d):{('C', 'd.')} That is I1
Goto(I3,C):{('S', 'CC.')} That is I5
Goto(I3,a):{('C', '.d'), ('C', '.aC'), ('C', 'a.C')} That is I4
Goto(I4,d):{('C', 'd.')} That is I1
Goto(I4,C):{('C', 'aC.')} That is I6
Goto(I4,a):{('C', '.d'), ('C', '.aC'), ('C', 'a.C')} That is I4



List of I's


I0: {('S', '.CC'), ("S'", '.S'), ('C', '.aC'), ('C', '.d')}
I1: {('C', 'd.')}
I2: {("S'", 'S.')}
I3: {('S', 'C.C'), ('C', '.aC'), ('C', '.d')}
I4: {('C', '.d'), ('C', '.aC'), ('C', 'a.C')}
I5: {('S', 'CC.')}
I6: {('C', 'aC.')}
['CC', 'S', 'aC', 'd']
```

TASK-2

AIM: Write a program to generate SLR parsing table.

DESCRIPTION:

SLR stands for Simple LR grammar. It is an example of a bottom-up parser. The "L" in SLR represents the scanning that advances from left to right and the "R" stands for constructions of derivation in reverse order, and the "(1)" represents the number of input symbols of lookahead.

Steps for constructing the SLR parsing table:

- Writing augmented grammar
- LR(0) collection of items to be found
- Find FOLLOW of LHS of production
- Defining 2 functions: goto[list of terminals] and action[list of non-terminals] in the parsing table

PROGRAM:

```
import copy

def grammarAugmentation(rules, nonterm_userdef,

                                                            start_symbol):

        newRules = []

        newChar = start_symbol + "'"

        while (newChar in nonterm_userdef):

                newChar += "'"

        newRules.append([newChar,

                                                    ['.', start_symbol]])



        for rule in rules:

                k = rule.split("->")

                lhs = k[0].strip()

                rhs = k[1].strip()

                multirhs = rhs.split('|')
        for rhs1 in multirhs:
            rhs1 = rhs1.strip().split()
            rhs1.insert(0, '.')
            newRules.append([lhs, rhs1])
    return newRules
def findClosure(input_state, dotSymbol):
    global start_symbol, \
        separatedRulesList, \
        statesDict
    closureSet = []
    if dotSymbol == start_symbol:
        for rule in separatedRulesList:
            if rule[0] == dotSymbol:
                closureSet.append(rule)
    else:
        closureSet = input_state
    prevLen = -1
    while prevLen != len(closureSet):
        prevLen = len(closureSet)
        tempClosureSet = []
        for rule in closureSet:
            indexOfDot = rule[1].index('.')
            if rule[1][-1] != '.':
                dotPointsHere = rule[1][indexOfDot + 1]
```

```
                    for in_rule in separatedRulesList:
                        if dotPointsHere == in_rule[0] and \
                                in_rule not in tempClosureSet:
                            tempClosureSet.append(in_rule)

            for rule in tempClosureSet:
                if rule not in closureSet:
                    closureSet.append(rule)
        return closureSet
    def compute_GOTO(state):
        global statesDict, stateCount
        generateStatesFor = []
        for rule in statesDict[state]:
            if rule[1][-1] != '.':
                indexOfDot = rule[1].index('.')
                dotPointsHere = rule[1][indexOfDot + 1]
                if dotPointsHere not in generateStatesFor:
                    generateStatesFor.append(dotPointsHere)
        if len(generateStatesFor) != 0:
            for symbol in generateStatesFor:
                GOTO(state, symbol)
        return
    def GOTO(state, charNextToDot):
        global statesDict, stateCount, stateMap
```

```
newState = []
for rule in statesDict[state]:
    indexOfDot = rule[1].index('.')
    if rule[1][-1] != '.':
        if rule[1][indexOfDot + 1] == \
                charNextToDot:
            shiftedRule = copy.deepcopy(rule)
            shiftedRule[1][indexOfDot] = \
                shiftedRule[1][indexOfDot + 1]
            shiftedRule[1][indexOfDot + 1] = '.'
            newState.append(shiftedRule)
addClosureRules = []
for rule in newState:
    indexDot = rule[1].index('.')
    if rule[1][-1] != '.':
        closureRes = \
            findClosure(newState, rule[1][indexDot + 1])
        for rule in closureRes:
            if rule not in addClosureRules \
                    and rule not in newState:
                addClosureRules.append(rule)
for rule in addClosureRules:
    newState.append(rule)
    stateExists = -1
    for state_num in statesDict:
        if statesDict[state_num] == newState:
            stateExists = state_num
            break
    if stateExists == -1:
        stateCount += 1
        statesDict[stateCount] = newState
        stateMap[(state, charNextToDot)] = stateCount
    else:
        stateMap[(state, charNextToDot)] = stateExists
    return
def generateStates(statesDict):
    prev_len = -1
    called_GOTO_on = []
    while (len(statesDict) != prev_len):
        prev_len = len(statesDict)
        keys = list(statesDict.keys())
        for key in keys:
            if key not in called_GOTO_on:
                called_GOTO_on.append(key)
                compute_GOTO(key)
    return
```

```
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 createParseTable(statesDict, stateMap, T, NT):
        global separatedRulesList, diction
        rows = list(statesDict.keys())
        cols = T+['$']+NT
        Table = []
        tempRow = []
        for y in range(len(cols)):
                tempRow.append('')
        for x in range(len(rows)):
                Table.append(copy.deepcopy(tempRow))
        for entry in stateMap:
                state = entry[0]
                symbol = entry[1]
                a = rows.index(state)
                b = cols.index(symbol)
                if symbol in NT:
                        Table[a][b] = Table[a][b]\
                                + f"{stateMap[entry]} "
                elif symbol in T:
                        Table[a][b] = Table[a][b]\
                                + f"S{stateMap[entry]} "
        numbered = {}
```

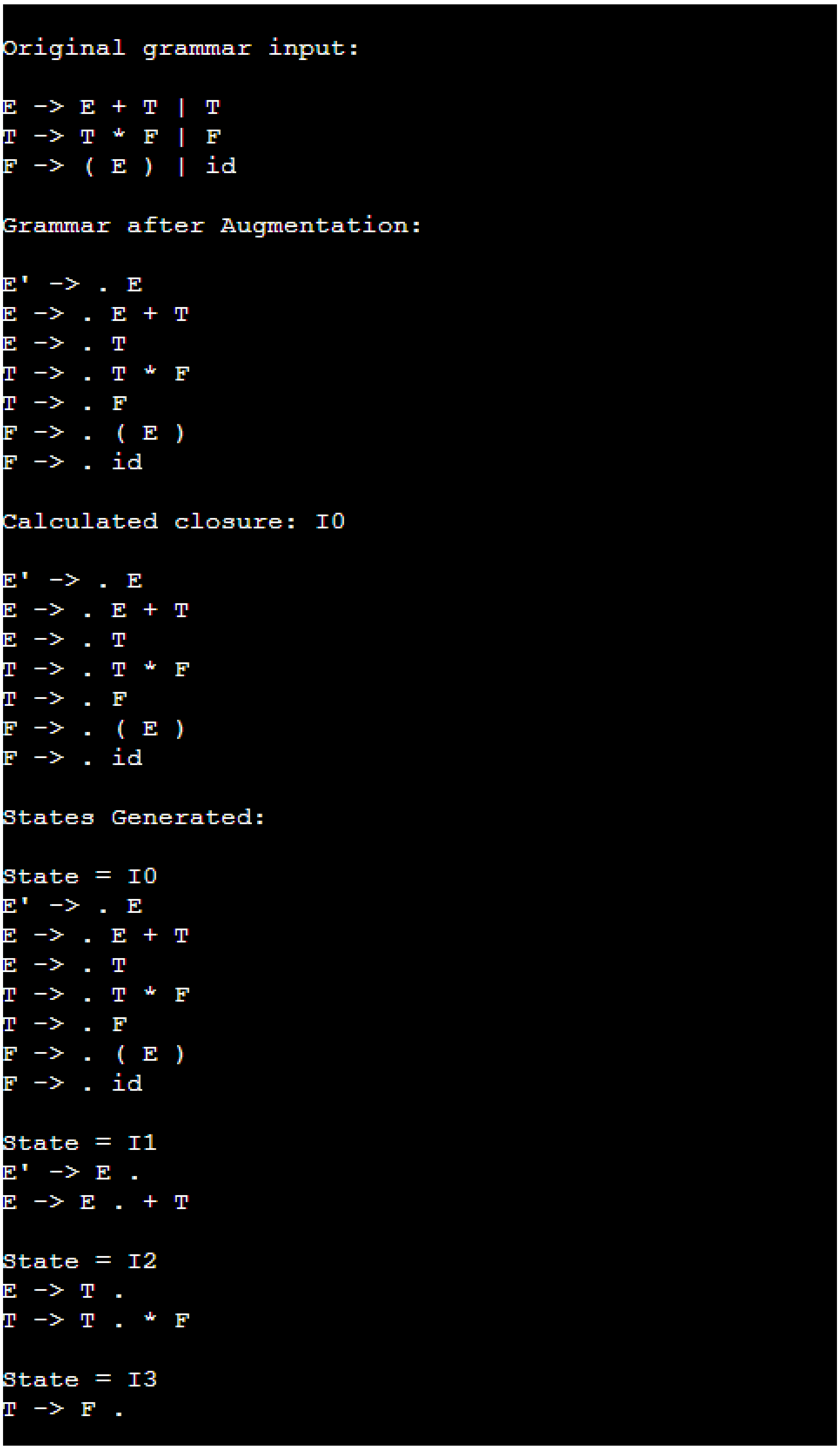
```
key_count = 0
for rule in separatedRulesList:
    tempRule = copy.deepcopy(rule)
    tempRule[1].remove('.')
    numbered[key_count] = tempRule
    key_count += 1
addedR = f"{separatedRulesList[0][0]} -> " \
    f"{separatedRulesList[0][1][1]}"
rules.insert(0, addedR)
for rule in rules:
    k = rule.split("->")
    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

for stateno in statesDict:
    for rule in statesDict[stateno]:
        if rule[1][-1] == '.':
            temp2 = copy.deepcopy(rule)
```

```
                        temp2[1].remove('.')
                        for key in numbered:
                            if numbered[key] == temp2:
                                follow_result = follow(rule[0])
                                for col in follow_result:
                                    index = cols.index(col)
                                    if key == 0:
                                        Table[stateno][index] =
"Accept"
                                    else:
                                        Table[stateno][index] =\

    Table[stateno][index]+f"R{key} "
    print("\nSLR(1) parsing table:\n")
    frmt = "{:>8}" * len(cols)
    print(" ", frmt.format(*cols), "\n")
    ptr = 0
    j = 0
    for y in Table:
        frmt1 = "{:>8}" * len(y)
        print(f"{{:>3}} {frmt1.format(*y)}"
            .format('I'+str(j)))
        j += 1
def printResult(rules):
    for rule in rules:
```

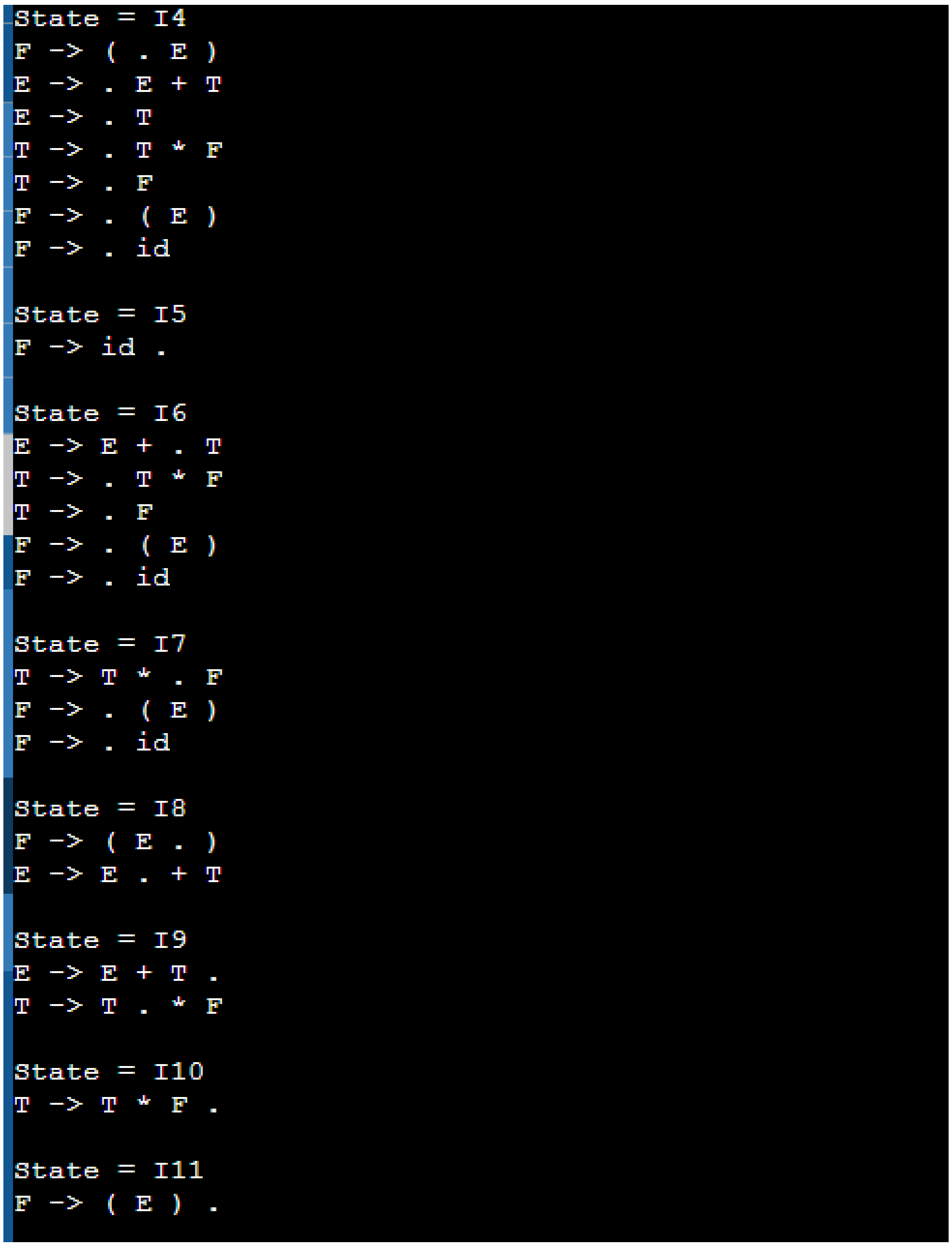
```
                print(f"{rule[0]} ->"
                        f" {' '.join(rule[1])}")
def printAllGOTO(diction):
        for itr in diction:
                print(f"GOTO ( I{itr[0]} ,"
                        f" {itr[1]} ) = I{stateMap[itr]}")
rules = ["E -> E + T | T",
                "T -> T * F | F",
                "F -> ( E ) | id"
                ]
nonterm_userdef = ['E', 'T', 'F']
term_userdef = ['id', '+', '*', '(', ')']
start_symbol = nonterm_userdef[0]
print("\nOriginal grammar input:\n")
for y in rules:
        print(y)
print("\nGrammar after Augmentation: \n")
separatedRulesList = \
        grammarAugmentation(rules,
                                        nonterm_userdef,
                                        start_symbol)
printResult(separatedRulesList)
start_symbol = separatedRulesList[0][0]
print("\nCalculated closure: I0\n")
```

```
I0 = findClosure(0, start_symbol)

printResult(I0)

statesDict = {}

stateMap = {}

statesDict[0] = I0

stateCount = 0

generateStates(statesDict)

print("\nStates Generated: \n")

for st in statesDict:

        print(f"State = I{st}")

        printResult(statesDict[st])

        print()

print("Result of GOTO computation:\n")

printAllGOTO(stateMap)

diction = {}

createParseTable(statesDict, stateMap,

                                term_userdef,

                                nonterm_userdef)
```

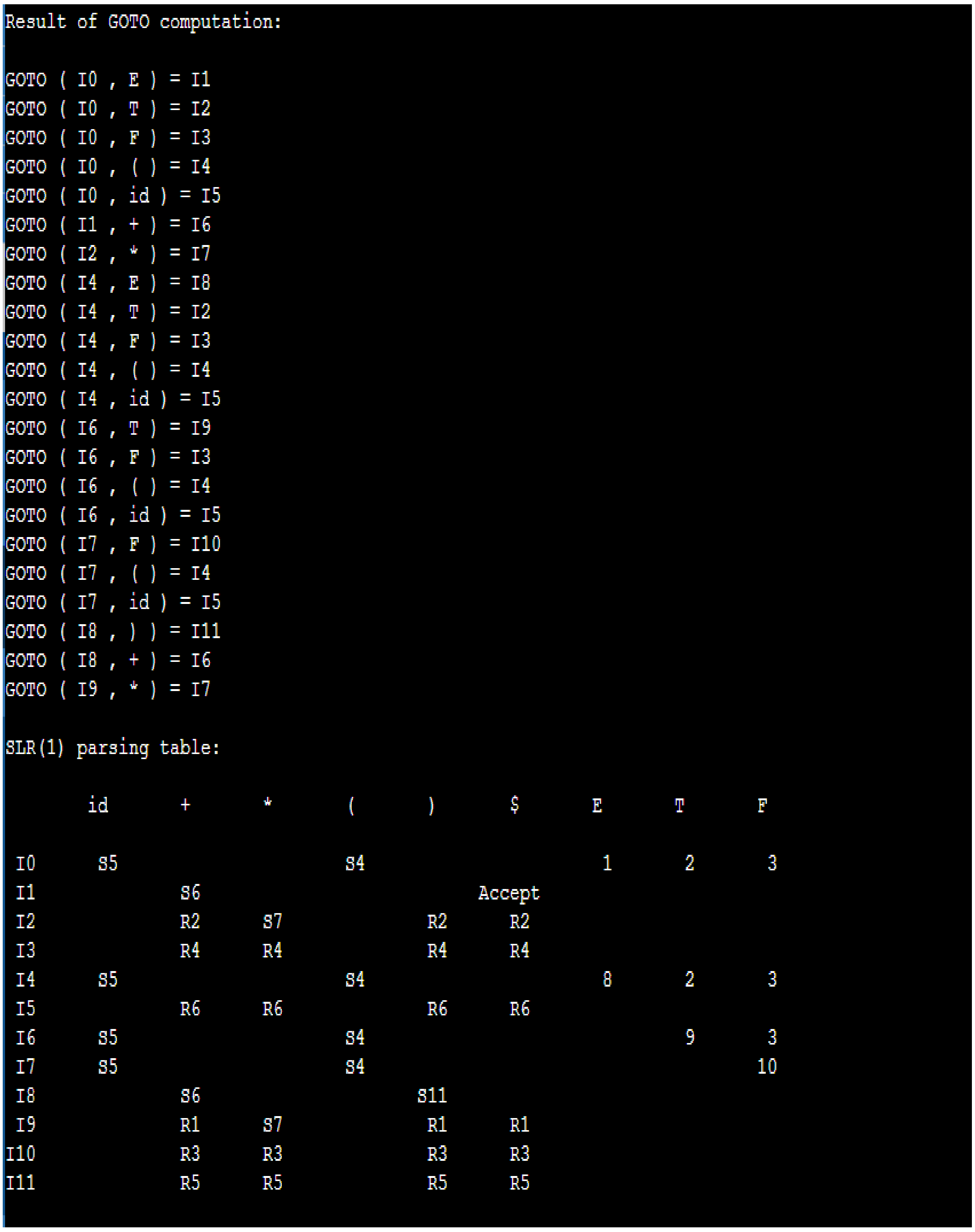
OUTPUT:



```
Original grammar input:

E -> E + T | T
T -> T * F | F
F -> ( E ) | id

Grammar after Augmentation:

E' -> . E
E -> . E + T
E -> . T
T -> . T * F
T -> . F
F -> . ( E )
F -> . id

Calculated closure: I0

E' -> . E
E -> . E + T
E -> . T
T -> . T * F
T -> . F
F -> . ( E )
F -> . id

States Generated:

State = I0
E' -> . E
E -> . E + T
E -> . T
T -> . T * F
T -> . F
F -> . ( E )
F -> . id

State = I1
E' -> E .
E -> E . + T

State = I2
E -> T .
T -> T . * F

State = I3
T -> F .
```



```
State = I4
F -> ( . E )
E -> . E + T
E -> . T
T -> . T * F
T -> . F
F -> . ( E )
F -> . id

State = I5
F -> id .

State = I6
E -> E + . T
T -> . T * F
T -> . F
F -> . ( E )
F -> . id

State = I7
T -> T * . F
F -> . ( E )
F -> . id

State = I8
F -> ( E . )
E -> E . + T

State = I9
E -> E + T .
T -> T . * F

State = I10
T -> T * F .

State = I11
F -> ( E ) .
```



```
Result of GOTO computation:

GOTO ( I0 , E ) = I1
GOTO ( I0 , T ) = I2
GOTO ( I0 , F ) = I3
GOTO ( I0 , ( ) = I4
GOTO ( I0 , id ) = I5
GOTO ( I1 , + ) = I6
GOTO ( I2 , * ) = I7
GOTO ( I4 , E ) = I8
GOTO ( I4 , T ) = I2
GOTO ( I4 , F ) = I3
GOTO ( I4 , ( ) = I4
GOTO ( I4 , id ) = I5
GOTO ( I6 , T ) = I9
GOTO ( I6 , F ) = I3
GOTO ( I6 , ( ) = I4
GOTO ( I6 , id ) = I5
GOTO ( I7 , F ) = I10
GOTO ( I7 , ( ) = I4
GOTO ( I7 , id ) = I5
GOTO ( I8 , ) ) = I11
GOTO ( I8 , + ) = I6
GOTO ( I9 , * ) = I7

SLR(1) parsing table:

        id      +       *       (       )       $       E       T       F

 I0      S5                     S4                       1       2       3
 I1              S6                            Accept
 I2              R2      S7              R2      R2
 I3              R4      R4              R4      R4
 I4      S5                     S4                       8       2       3
 I5              R6      R6              R6      R6
 I6      S5                     S4                               9       3
 I7      S5                     S4                                      10
 I8              S6                     S11
 I9              R1      S7              R1      R1
I10              R3      R3              R3      R3
I11              R5      R5              R5      R5
```

COMPILER DESIGN

EXPERIMENT-9

TASK-1

AIM: Write a program to simulate Symbol table Management.

DESCRIPTION:

Symbol table is an important data structure created and maintained by compilers in order to store information about the occurrence of various entities such as variable names, function names, objects, classes, interfaces, etc. Symbol table is used by both the analysis and the synthesis parts of a compiler.

A symbol table may serve the following purposes depending upon the language in hand:

- To store the names of all entities in a structured form at one place.
- To verify if a variable has been declared.
- To implement type checking, by verifying assignments and expressions in the source code are semantically correct.
- To determine the scope of a name (scope resolution).

PROGRAM:

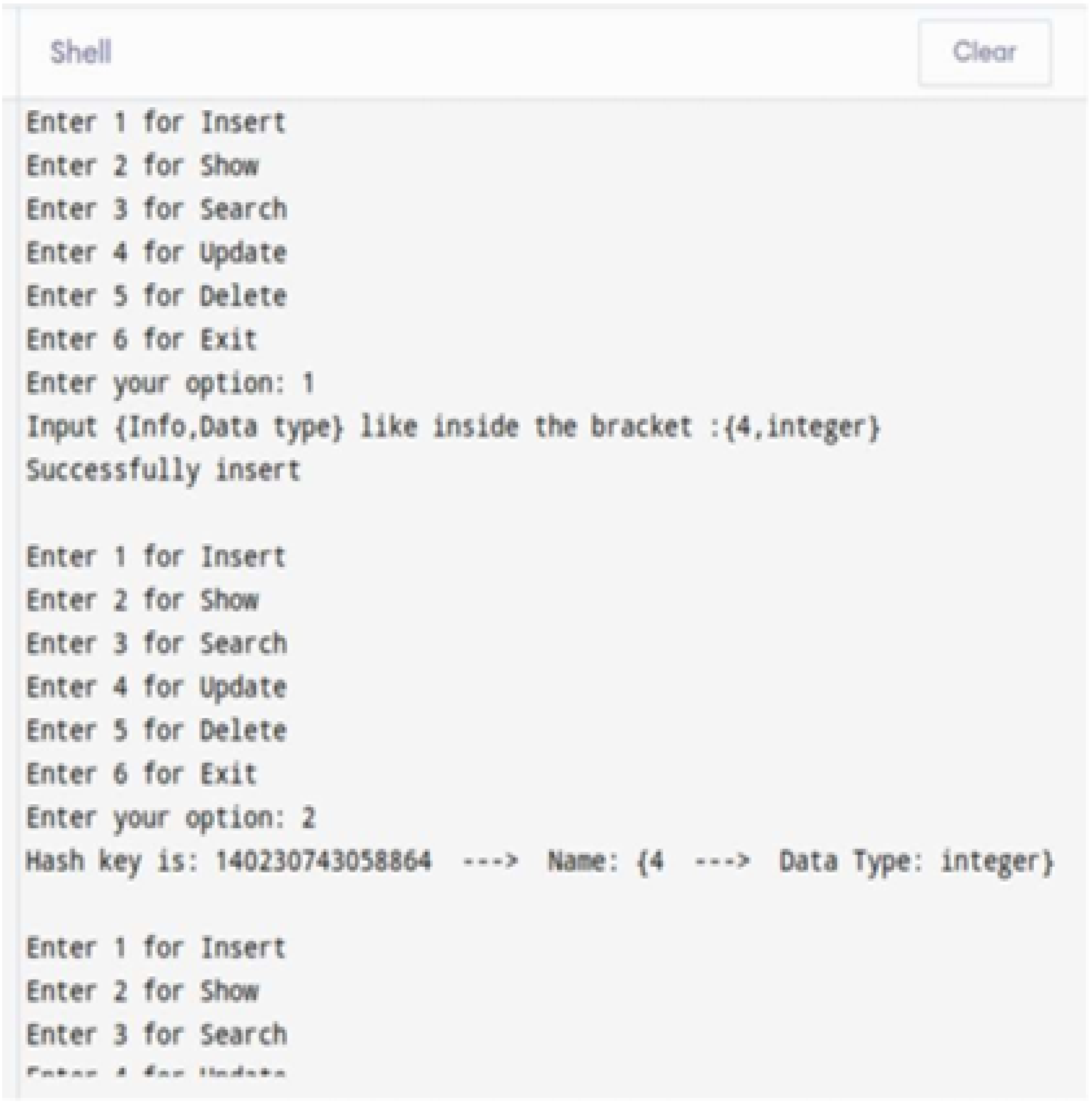
```
 class Entry:
   def _init_(self, hash_key, name, data_type):
     self.hash_key = hash_key
     self.name = name
     self.data_type = data_type


symbol_table = []

def is_comma_separated(my_input):
   for charecter in my_input:
     if charecter == ',':
       return True
   return False

def get_hash_key(name):
   ascii_value = 0
   #for cha in name:
   # ascii_value += ord(cha)
   #return ascii_value % 100
   ascii_value=id(name)
```

```
    return ascii_value

def is_match_found(name):
    index = -1
    if len(symbol_table) > 0:
        for index, element in enumerate(symbol_table):
            if element.name == name:
                return True, index
        return False, index

    return False, index

# Insert new String Data ( Nime,string)

def insert(my_input):
    if is_comma_separated(my_input):
        users_input = my_input.split(',')
        name = users_input[0].strip()
        data_type = users_input[1].strip()
        hash_key = get_hash_key(name)
        new_entry = Entry(hash_key, name, data_type)

        match_found, index = is_match_found(name)
        if not match_found:
            symbol_table.append(new_entry)
            return 'Successfully insert'
        return "Name already exists."

    return "Sorry! You didn't enter comma between words ... "

def show():
    for entry in symbol_table:
        print(f"Hash key is: {entry.hash_key} --->  Name: {entry.name}  --->  Data Type: 
{entry.data_type} ")

def search(name):
    match_found, index = is_match_found(name)
    if match_found:
        return f"Match found. Hash Key is --> {symbol_table[index].hash_key} and Data
Type --> {symbol_table[index].data_type}"

    return "No match found with this Name. "

def update(name):
    match_found, index = is_match_found(name)

    if match_found:
      data_type = input("Enter new data type to update: ")
      if symbol_table[index].data_type == data_type:
        return "There is nothing to update. You entered same data type."

      symbol_table[index].data_type = data_type
      return "Update data type successfully."

    return "No match found with this name. "


def delete(name):
  match_found, index = is_match_found(name)

  if match_found:
    del symbol_table[index]
    return "Delete successfully"

  return "No match found with this name. "



while True:
    print("Enter 1 for Insert")
    print("Enter 2 for Show")
    print("Enter 3 for Search")
    print("Enter 4 for Update")
    print("Enter 5 for Delete")
    print("Enter 6 for Exit")
```

```
user_input = input('Enter your option: ')

if user_input == '1':upport.mozilla.org/
    new_data = input("Input {Info,Data type} like inside the bracket :")
    print('')
    print(insert(new_data))
    print('')

elif user_input == '2':
  if len(symbol_table) > 0:
      print('')
      show()
      print('')
  else:
    print("Symbol table is empty. Please insert data. \n")

elif user_input == '3':
  if len(symbol_table) > 0:
    new_data = input("Enter existing name from symbol table: ")
    print('')
    print(search(new_data))
    print('')
  else:
    print("Symbol table is empty. Please insert data. \n")

elif user_input == '4':
  if len(symbol_table) > 0:
    new_data = input("Enter existing name from symbol table update: ")
    print('')
    print(update(new_data))
    print('')
  else:
    print("Symbol table is empty. Please insert data.\n ")

elif user_input == '5':
  if len(symbol_table) > 0:
    new_data = input("Enter existing name from symbol table to delete: ")
    print('')
    print(delete(new_data))
    print('')
```

```
            else:
              print("Symbol table is empty. Please insert data. \n")

          elif user_input == '6':
            break

          else:
              print("Wrong input")
```

OUTPUT:



```
Enter 1 for Insert
Enter 2 for Show
Enter 3 for Search
Enter 4 for Update
Enter 5 for Delete
Enter 6 for Exit
Enter your option: 1
Input {Info,Data type} like inside the bracket :{4,integer}
Successfully insert

Enter 1 for Insert
Enter 2 for Show
Enter 3 for Search
Enter 4 for Update
Enter 5 for Delete
Enter 6 for Exit
Enter your option: 2
Hash key is: 140230743058864  --->  Name: {4  --->  Data Type: integer}

Enter 1 for Insert
Enter 2 for Show
Enter 3 for Search
Enter 4 for Update
```

TASK-2

AIM: Write a program to generate 3-address code.

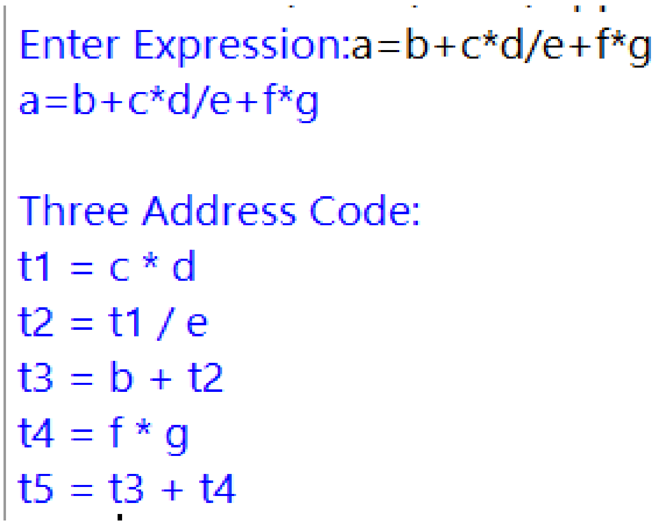
DESCRIPTION:

Three-address code is an intermediate code. It is used by the optimizing compilers. In three-address code, the given expression is broken down into several separate instructions. These instructions can easily translate into assembly language. Each Three address code instruction has at most three operands. It is a combination of assignment and a binary operator.

PROGRAM:

```
Operators = ['+', '-', '*', '/', '(', ')', '^']
Priority = {'+':1, '-':1, '*':2, '/':2, '^':3}
def printCode(st, output, no):
    a = st.pop()
    c1 = output.pop()
    c2 = output.pop()
    print(f't{no[0]} = {c2} {a} {c1}')
    newVar = f't{no[0]}'
    no[0] += 1
    output.append(newVar)


def threeAddressCode(exp):
    st = []
    output = []
    no = [1]
    for char in exp:
        if char not in Operators:
            output.append(char)
        elif char == '(':
            st.append('(')
        elif char == ')':
            while st and st[-1]!= '(':
                printCode(st, output, no)

            st.pop()
        else:
            while len(st) > 0 and st[-1]!='(' and Priority[char] <= Priority[st[-1]]:
                printCode(st, output, no)
            st.append(char)
    while len(st) > 0:
        printCode(st, output, no)

exp = input('Enter Expression:')
print(exp)
print()
print('Three Address Code:')
threeAddressCode(exp)
```

OUTPUT:



```
Enter Expression:a=b+c*d/e+f*g
a=b+c*d/e+f*g

Three Address Code:
t1 = c * d
t2 = t1 / e
t3 = b + t2
t4 = f * g
t5 = t3 + t4
```

COMPILER DESIGN

EXPERIMENT-10

TASK-1

AIM: Write a program to generate code.

DESCRIPTION:

Target code generation deals with assembly language to convert optimized code into machine understandable format. Target code can be machine readable code or assembly code. The main purpose of the Target Code generator is to write a code that the machine can understand and also register allocation, instruction selection, etc. The output is dependent on the type of assembler. This is the final stage of compilation. The optimized code is converted into relocatable machine code which then forms the input to the linker and loader.

PROGRAM:

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
int label[20];
int no=0;
int main()
{
FILE *fp1,*fp2;
char fname[10],op[10],ch;
char operand1[8],operand2[8],result[8];
int i=0,j=0;
printf("\nEnter the filename of intermediate code\n");
scanf("%s",&fname);
fp1=fopen(fname,"r");
fp2=fopen("target.txt","w");
if(fp1==NULL||fp2==NULL)
{
printf("\nError opening the file");
exit(0);
}
while(!feof(fp1))
{
fprintf(fp2,"\n");
fscanf(fp1,"%s",op);
i++;
if(check_label(i))
fprintf(fp2,"\nlabel# %d",i);
if(strcmp(op,"print")==0)
```

```
{
fscanf(fp1,"%s",result);
fprintf(fp2,"\n\tOUT %s",result);
}
if(strcmp(op,"goto")==0)
{
fscanf(fp1,"%s%s",operand1,operand2);
label[no++]=atoi(operand2);
}
if(strcmp(op,"[]=")==0)
{
fscanf(fp1,"%s%s%s",operand1,operand2,result);
fprintf(fp2,"\n\tSTORE %s[%s],%s",operand1,operand2,result);
}
if(strcmp(op,"uminus")==0)
{
fscanf(fp1,"%s,%s",operand1,result);
fprintf(fp2,"\n\t,LOAD %s",operand1);
fprintf(fp2,"\n\tSTORE R1,%s",result);
}
switch(op[0])
{
case '*':
fscanf(fp1,"%s%s%s",operand1,operand2,result);
fprintf(fp2,"\n\t LOAD",operand1);
fprintf(fp2,"\n\t LOAD %s,R1",operand2);
fprintf(fp2,"\n\t MUL R1,R0");
fprintf(fp2,"\n\t STORE R0,%s",result);
break;
case '+':
fscanf(fp1,"%s%s%s",operand1,operand2,result);
fprintf(fp2,"\n\t LOAD %s,R0",operand1);
fprintf(fp2,"\n\t LOAD %s,R1",operand2);
fprintf(fp2,"\n\t ADD R1,R0");
fprintf(fp2,"\n\t STORE R0,%s",result);
break;
case '-':
fscanf(fp1,"%s%s%s",operand1,operand2,result);
fprintf(fp2,"\n\t LOAD %s,R0",operand1);
fprintf(fp2,"\n\t LOAD %s,R1",operand2);
fprintf(fp2,"\n\t SUB R1,R0");
fprintf(fp2,"\n\t STORE R0,%s",result);
break;
case '/':
fscanf(fp1,"%s%s%s",operand1,operand2,result);
fprintf(fp2,"\n\t LOAD %s,R0",operand1);
fprintf(fp2,"\n\t LOAD %s,R1",operand2);
fprintf(fp2,"\n\t DIV R1,R0");
fprintf(fp2,"\n\t STORE R0,%s",result);
break;
case '=':
fscanf(fp1,"%s%s",operand1,result);
fprintf(fp2,"\n\t STORE %s%s",operand1,result);
break;
case '>':
fscanf(fp1,"%s%s%s",operand1,operand2,result);
fprintf(fp2,"\n\t LOAD %s,R0",operand1);
fprintf(fp2,"\n\t JGT %S,label# %s",operand2,result);
label[no++]=atoi(result);
break;
case '<':
fscanf(fp1,"%s%s%s",operand1,operand2,result);
fprintf(fp2,"\n\t LOAD %s,R0",operand1);
fprintf(fp2,"\n\t JLT %S,label# %s",operand2,result);
label[no++]=atoi(result);
break;
}
}
fclose(fp2);
fclose(fp1);
fp2=fopen("target.txt","r");
if(fp2==NULL)
{
printf("Error opening in file\n");
exit(0);
}
do{
ch=fgetc(fp2);
printf("%c",ch);
}
while(ch!=EOF);
fclose(fp1);
return 0;
}
int check_label(int k)
{
int i;
for(i=0;i<no;i++)
{
if(k==label[i])
return 1;
```

```
}
return 0;
}
```

OUTPUT:

Input.txt

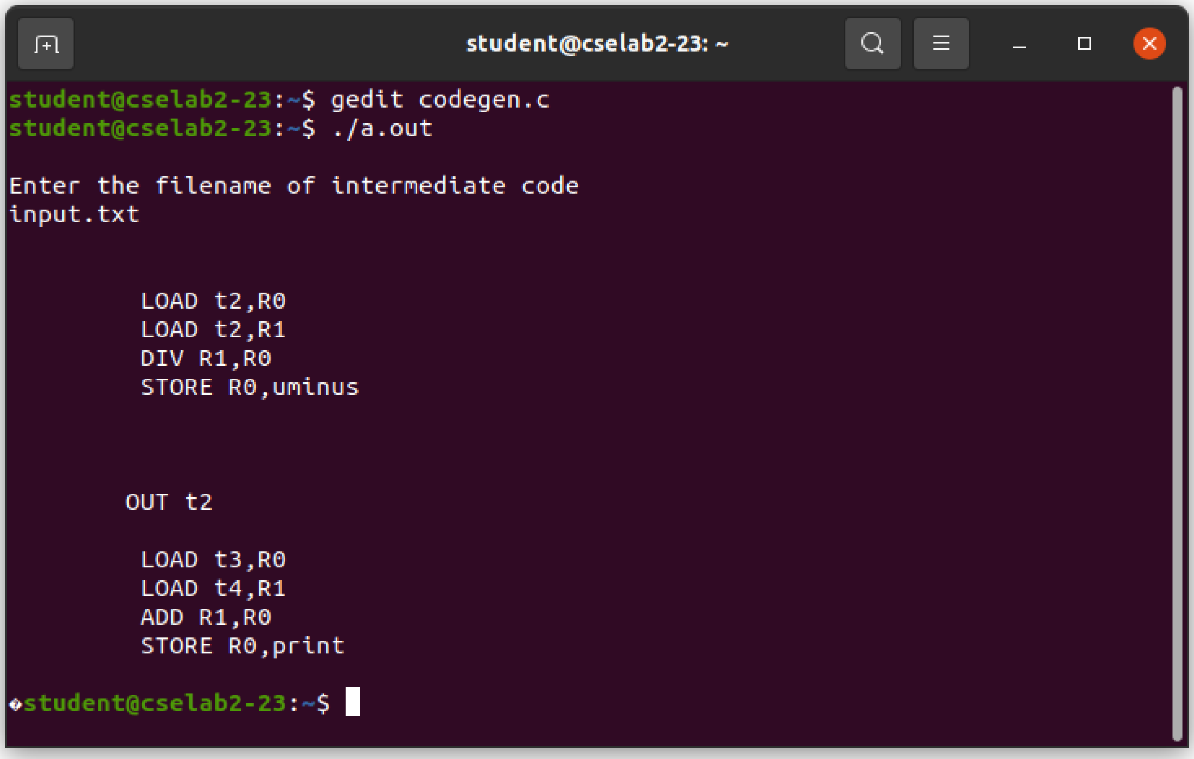
/t3 t2 t2

uminus t2 t2

print t2

+t1 t3 t4

print t4



```
student@cselab2-23:~$ gedit codegen.c
student@cselab2-23:~$ ./a.out

Enter the filename of intermediate code
input.txt


         LOAD t2,R0
         LOAD t2,R1
         DIV R1,R0
         STORE R0,uminus



        OUT t2

         LOAD t3,R0
         LOAD t4,R1
         ADD R1,R0
         STORE R0,print

◆student@cselab2-23:~$ 
```

TASK-2

AIM: Write a program to optimize the code.

DESCRIPTION:

The code optimization in the synthesis phase is a program transformation technique, which tries to improve the intermediate code by making it consume fewer resources (i.e. CPU, Memory) so that faster-running machine code will result. It transforms the code so that it consumes fewer resources and produces more speed. The meaning of the code being transformed isn't altered. Optimization can be categorized into two types: machine-dependent and machine-independent.

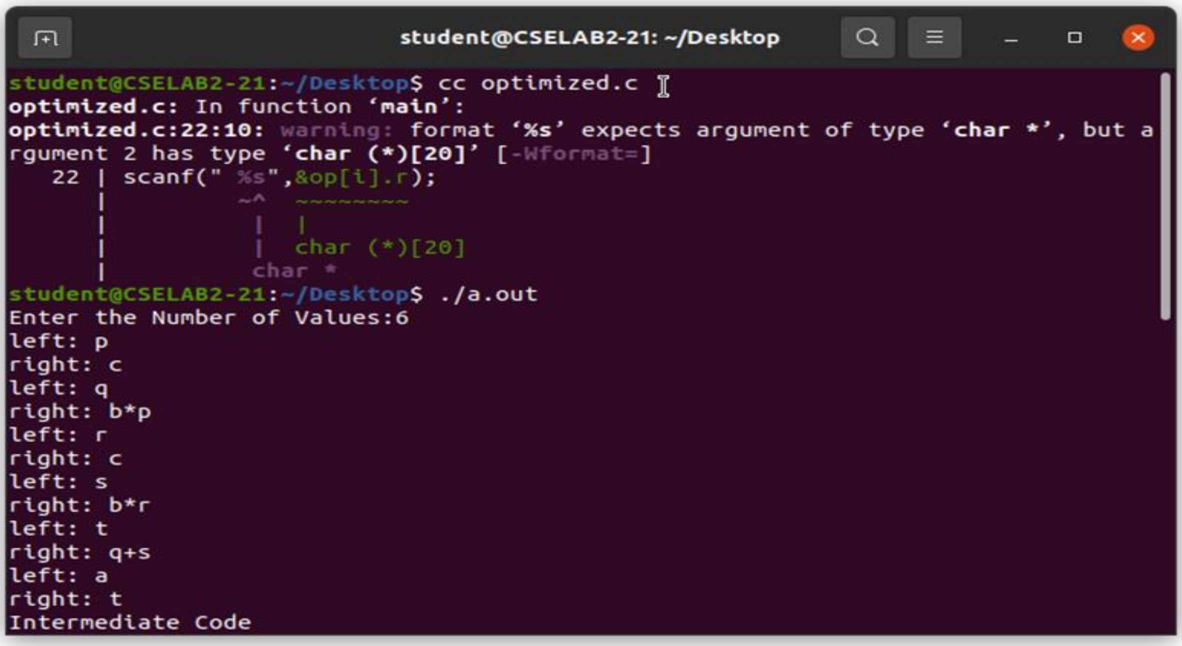
PROGRAM:

```
#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;
```

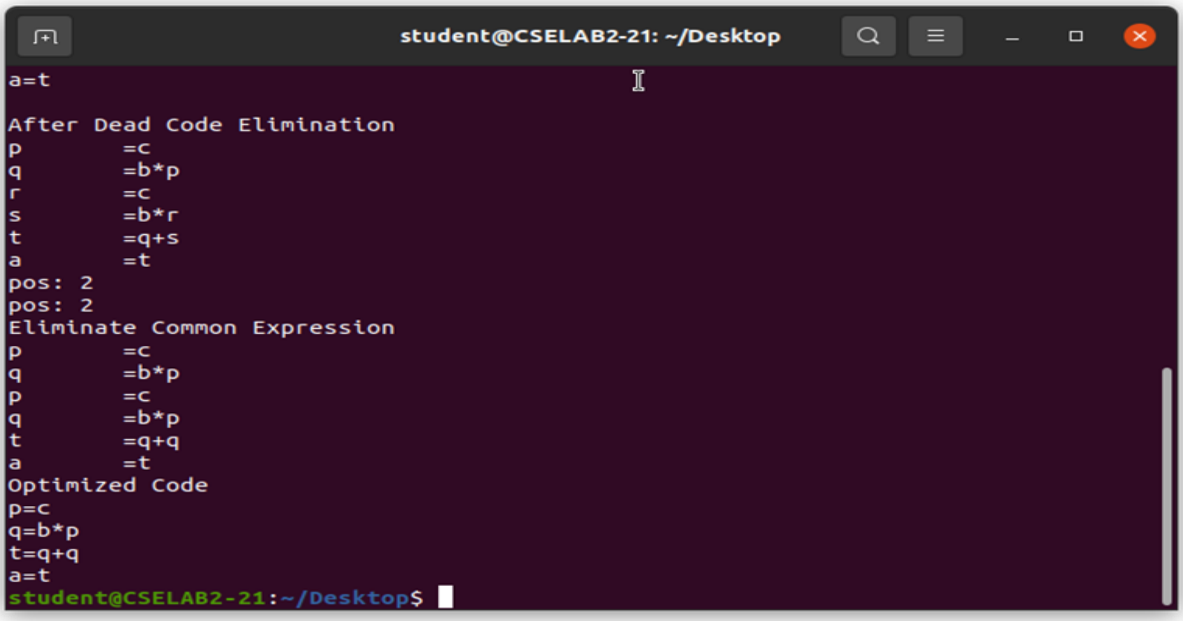
```
strcpy(pr[z].r,op[n-1].r);
z++;
printf("\nAfter Dead Code Elimination\n");
for(k=0;k<z;k++)
{
printf("%c\t=",pr[k].l);
printf("%s\n",pr[k].r);
}
for(m=0;m<z;m++)
{
tem=pr[m].r;
for(j=m+1;j<z;j++)
{
p=strstr(tem,pr[j].r);
if(p)
{
t=pr[j].l;
pr[j].l=pr[m].l;
for(i=0;i<z;i++)
{
l=strchr(pr[i].r,t) ;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



```
student@CSELAB2-21:~/Desktop$ cc optimized.c
optimized.c: In function 'main':
optimized.c:22:10: warning: format '%s' expects argument of type 'char *', but a
rgument 2 has type 'char (*)[20]' [-Wformat=]
   22 | scanf(" %s",&op[i].r);
      |         ~^  ~~~~~~~~
      |          |  |
      |          |  char (*)[20]
      |          char *
student@CSELAB2-21:~/Desktop$ ./a.out
Enter the Number of Values:6
left: p
right: c
left: q
right: b*p
left: r
right: c
left: s
right: b*r
left: t
right: q+s
left: a
right: t
Intermediate Code
```



```
a=t

After Dead Code Elimination
p       =c
q       =b*p
r       =c
s       =b*r
t       =q+s
a       =t
pos: 2
pos: 2
Eliminate Common Expression
p       =c
q       =b*p
p       =c
q       =b*p
t       =q+q
a       =t
Optimized Code
p=c
q=b*p
t=q+q
a=t
student@CSELAB2-21:~/Desktop$
```