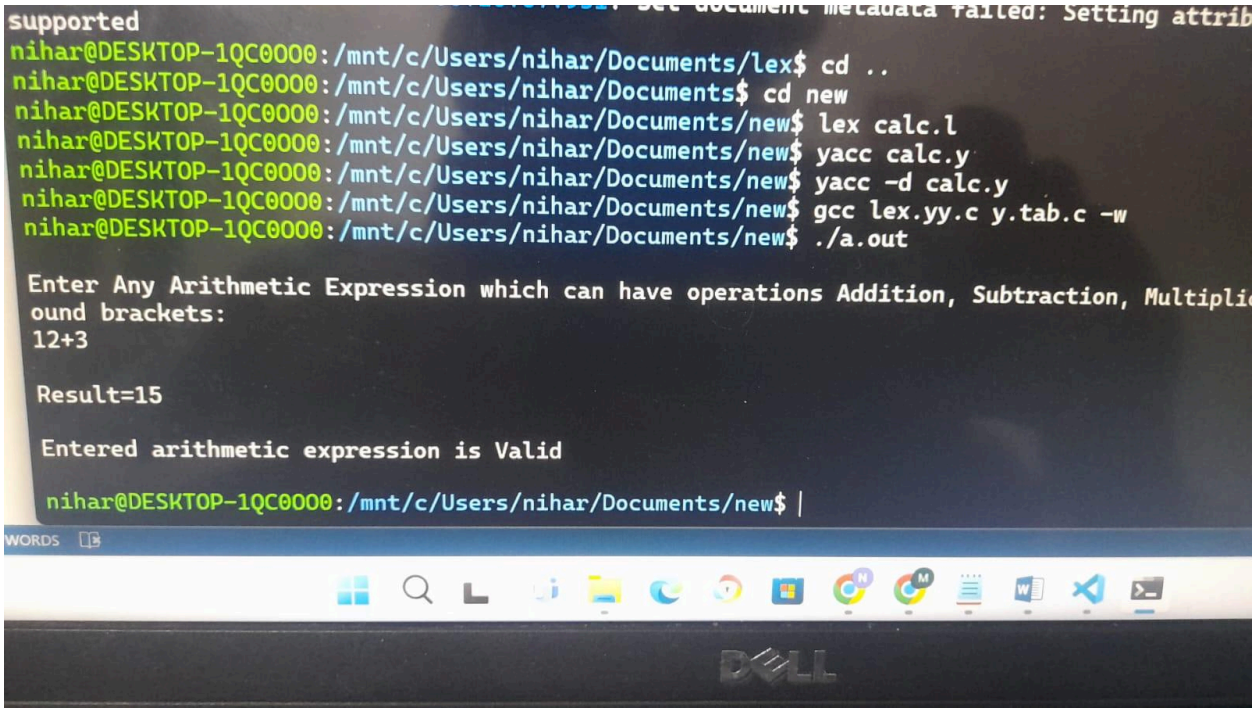
Need ubuntu
Sudo apt update
Sudo apt install flex
Sudo apt install bison
Cd /mnt/c

yacc -d calc.y
lex calc.l
gcc y.tab.c lex.yy.c -o calc
./calc

Q1) To Implement YACC program of calculation.



Calc.l

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

/* Rule Section */
```

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

      }
[\t] ;

[\n] return 0;

. return yytext[0];

%%

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

Calc.y

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

%token NUMBER

%left '+' '-'

%left '*' '/' '%'

%left '(' ')'

/* Rule Section */
%%

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;}
 ;

%%

//driver code
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;
}
```

Q2) To implement lex program of operator
Save as operator_lex.l in vscode

```
nidhi@DESKTOP-E65NOHF:/mnt/c$ cd Users/NItinpatil/Documents/java/spcc/
nidhi@DESKTOP-E65NOHF:/mnt/c/Users/NItinpatil/Documents/java/spcc$ lex operator_lex.l
nidhi@DESKTOP-E65NOHF:/mnt/c/Users/NItinpatil/Documents/java/spcc$ gcc lex.yy.c -o operator_lex -ll
\nidhi@DESKTOP-E65NOHF:/mnt/c/Users/NItinpatil/Documents/java/spcc$ ./operator_lex
*
MUL

+
ADD
```

Code:

```
%{
/* Definitions Section */
%}

/* Regular Definitions */
%%

"+"     { printf("ADD\n"); }
"-"     { printf("SUB\n"); }
"*"     { printf("MUL\n"); }
"/"     { printf("DIV\n"); }
"%"     { printf("MOD\n"); }
"="     { printf("ASSIGN\n"); }
"=="    { printf("EQUAL\n"); }
"!="    { printf("NOT_EQUAL\n"); }
"<"     { printf("LESS_THAN\n"); }
">"     { printf("GREATER_THAN\n"); }
"<="    { printf("LESS_THAN_EQUAL\n"); }
">="    { printf("GREATER_THAN_EQUAL\n"); }
"&&"    { printf("LOGICAL_AND\n"); }
"||"    { printf("LOGICAL_OR\n"); }
"!"     { printf("LOGICAL_NOT\n"); }

.       { printf("UNKNOWN\n"); } /* Catch-all for unrecognized characters */

%%

/* User Code Section */

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

Q3) To Implement LEX program of identifier & keyword. Save as lexprogram.l

```
nidhi@DESKTOP-E65NOHF:/mnt/c/Users/NItinpatil/Documents/java/spcc$ lex lexprogram.l
nidhi@DESKTOP-E65NOHF:/mnt/c/Users/NItinpatil/Documents/java/spcc$ gcc lex.yy.c -o lexprogram -ll
nidhi@DESKTOP-E65NOHF:/mnt/c/Users/NItinpatil/Documents/java/spcc$ ./lexprogram
int
Keyword: int
float
Keyword: float
a
Identifier: a
```

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

%%
int|float|char     { printf("Keyword: %s\n", yytext); }
[a-zA-Z][a-zA-Z0-9]*    { printf("Identifier: %s\n", yytext); }
.|\n            { /* ignore other characters */ }
%%

int yywrap() {
    return 1;
}

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

Q4) To Implement LEX program of vowels

Save as lexvowels.l

```
nidhi@DESKTOP-E65NOHF:/mnt/c/Users/NItinpatil/Documents/java/spcc$ lex lexvowels.l
nidhi@DESKTOP-E65NOHF:/mnt/c/Users/NItinpatil/Documents/java/spcc$ gcc lex.yy.c -o lexvowels -ll
nidhi@DESKTOP-E65NOHF:/mnt/c/Users/NItinpatil/Documents/java/spcc$ ./lexvowels
Enter the string of vowels: Nidhi

Number of vowels: 2
```

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

%%
[aeiouAEIOU] {vow_count++;}
```

```
.           ; // Ignore any other character
%%

int main()
{
        char input[100];
        printf("Enter the string of vowels: ");
        fgets(input, sizeof(input), stdin);
        yy_scan_string(input);
        yylex();
        printf("Number of vowels: %d\n", vow_count);
        return 0;
}
```

Q5)To Implement LEX program of prime no.

Save as prime.l

```
nidhi@DESKTOP-E65NOHF:/mnt/c/Users/NItinpatil/Documents/java/spcc$ lex prime.l
nidhi@DESKTOP-E65NOHF:/mnt/c/Users/NItinpatil/Documents/java/spcc$ gcc lex.yy.x -o prime -ll
/usr/bin/ld: cannot find lex.yy.x: No such file or directory
collect2: error: ld returned 1 exit status
nidhi@DESKTOP-E65NOHF:/mnt/c/Users/NItinpatil/Documents/java/spcc$ gcc lex.yy.c -o prime -ll
nidhi@DESKTOP-E65NOHF:/mnt/c/Users/NItinpatil/Documents/java/spcc$ ./prime
5
5 is a prime number.

1
1 is not a prime number.
```

```
%{
#include <stdio.h>
int is_prime(int n) {
    if (n <= 1)
        return 0; // Not prime
    for (int i = 2; i * i <= n; i++) {
        if (n % i == 0)
            return 0; // Not prime
    }
    return 1; // Prime
```

```
}
%}

%%
[0-9]+ {
    int num = atoi(yytext);
    if (is_prime(num))
        printf("%d is a prime number.\n", num);
    else
        printf("%d is not a prime number.\n", num);
}
%%

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

Q6) To Implement LEX program of largest word.
Save as largest.l

```
nidhi@DESKTOP-E65NOHF:/mnt/c/Users/NItinpatil/Documents/java/spcc$ lex largest.l
nidhi@DESKTOP-E65NOHF:/mnt/c/Users/NItinpatil/Documents/java/spcc$ gcc lex.yy.x -o largest -ll
/usr/bin/ld: cannot find lex.yy.x: No such file or directory
collect2: error: ld returned 1 exit status
nidhi@DESKTOP-E65NOHF:/mnt/c/Users/NItinpatil/Documents/java/spcc$ gcc lex.yy.c -o largest -ll
nidhi@DESKTOP-E65NOHF:/mnt/c/Users/NItinpatil/Documents/java/spcc$ ./largest
this is
Largest word: this
```

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

#define MAX_LEN 100 // Maximum length of a word

char largest_word[MAX_LEN + 1]; // To store the largest word found

void update_largest_word(const char *word) {
    if (strlen(word) > strlen(largest_word))
        strcpy(largest_word, word);
}
```

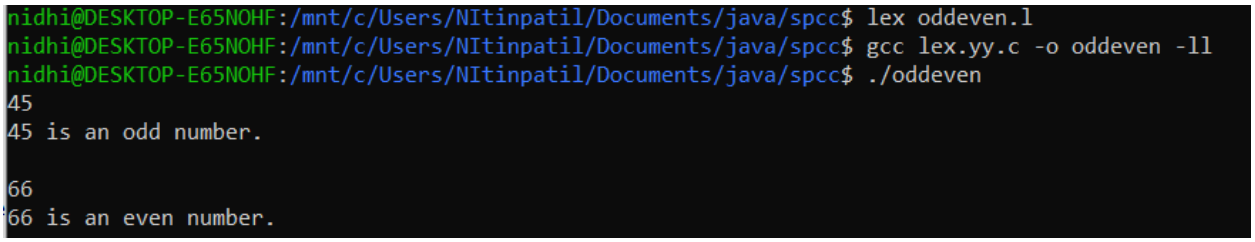
```
%}

%%
[a-zA-Z]+ {
    update_largest_word(yytext);
}
\n {
    if (strlen(largest_word) > 0)
        printf("Largest word: %s\n", largest_word);
    strcpy(largest_word, ""); // Reset largest word for next line
}
. ; // Ignore all other characters
%%

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

Q7) To Implement LEX program of odd and even no.

Save as oddeven.l in vscode




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

%%
[0-9]+ {
    int num = atoi(yytext);
    if (num % 2 == 0)
        printf("%d is an even number.\n", num);
    else
        printf("%d is an odd number.\n", num);
}
```

```
%%

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

Q8) To implement Lexical Analyzer programs (identifier, keywords)(JAVA/C/C++/Python/R-lang /Lex).

```
import re

def lex(code):

    tokens = []

    non_tokens = []

    lines = code.split('\n')

    for line in lines:

        lexeme_count = 0

        while lexeme_count < len(line):

            lexeme = line[lexeme_count]

            if lexeme == '#':

                directive = ""

                while lexeme_count < len(line) and line[lexeme_count] != "\n":

                    directive += line[lexeme_count]

                    lexeme_count += 1

                non_tokens.append(('preprocessor directive', directive.strip()))


            elif lexeme == '/' and lexeme_count + 1 < len(line) and line[lexeme_count + 1]
== '/':
```

```
            comment = ""
            lexeme_count += 2 # Skip the //
            while lexeme_count < len(line) and line[lexeme_count] != "\n":
                comment += line[lexeme_count]
                lexeme_count += 1
            non_tokens.append(('comment', comment.strip()))

        elif lexeme.isalpha() or lexeme == '_':
            typ, tok, consumed = lex_id(line[lexeme_count:])
            lexeme_count += consumed
            tokens.append((typ, tok))

        else:
            lexeme_count += 1

    return tokens, non_tokens

def lex_id(line):
    keywords = ['int', 'float', 'if', 'else', 'return', 'double', 'typedef', 'union']
    id = ""
    for c in line:
        if not c.isdigit() and not c.isalpha() and c != "_":
            break
```

```
        id += c
    if id in keywords:
        return "keyword", id, len(id)
    else:
        return "identifier", id, len(id)

# Example usage
code = """#include <stdio.h>
#define NUMS 8,9
// This will compare 2 numbers
int maximum (int x, int y) {
    if (x < y){
        return x
    } else {
        return y
    }
}"""

tokens, non_tokens = lex(code)

print("Tokens:")
for token in tokens:
    print(token)
```

```
print("\nNon-Tokens:")

for non_token in non_tokens:

    print(non_token)
```

Q9) To implement Lexical Analyzer programs (operators) (JAVA/C/C++/Python/R-lang/Lex).

```
def lex(code):

        operators = { '=': 'Assignment Operator', '+': 'Addition Operator', '-' : 'Subtraction Operator',

            '/' : 'Division Operator', '*': 'Multiplication Operator', '++' : 'Increment Operator',

            '--' : 'Decrement Operator', '%': 'Modulus Operator', '^': 'Exponential Operator', '!': 'Not Operator'}


        operator = []

        non_operator = []


        tokens = input_string.split()

        for token in tokens:

        if token in operators:

         operator.append((operators[token], token))

        else:

         non_operator.append(token)
```

```
        return operator, non_operator


input_string = "a + b - c * d / e % f ^ g ! h"


operator, non_operator = lex(input_string)


print("Operator:")
for token in operator:
        print(token)


print("\nNon-Operator:")
for non_token in non_operator:
    print(non_token)
```

Q10) Write a program to remove left recursion by direct method for given set of production rules(JAVA/C/C++/Python/R-lang /Lex).

```
class NonTerminal :
                def __init__(self, name) :
                        self.name = name
                        self.rules = []
                def addRule(self, rule) :
                        self.rules.append(rule)
```

```
        def setRules(self, rules) :
                self.rules = rules
        def getName(self) :
                return self.name
        def getRules(self) :
                return self.rules
        def printRule(self) :
                print(self.name + " -> ", end = "")
                for i in range(len(self.rules)) :
                        print(self.rules[i], end = "")
                        if i != len(self.rules) - 1 :
                                print(" | ", end = "")
                print()


class Grammar :
        def __init__(self) :
                self.nonTerminals = []


        def addRule(self, rule) :
                nt = False
                parse = ""


                for i in range(len(rule)) :
```

```
                    c = rule[i]
                    if c == ' ' :
                            if not nt :
                                    newNonTerminal = NonTerminal(parse)
self.nonTerminals.append(newNonTerminal)
                                    nt = True
                                    parse = ""
                            elif parse != "" :
                                    self.nonTerminals[len(self.nonTerminals)
- 1].addRule(parse)
                                    parse = ""
                    elif c != '|' and c != '-' and c != '>' :
                            parse += c
            if parse != "" :
                    self.nonTerminals[len(self.nonTerminals) - 1].addRule(parse)

    def inputData(self) :
            self.addRule("S -> Sa | Sb | c | d")

    def solveNonImmediateLR(self, A, B) :
            nameA = A.getName()
            nameB = B.getName()
            rulesA = []
```

```
                rulesB = []
                newRulesA = []
                rulesA = A.getRules()
                rulesB = B.getRules()

                for rule in rulesA :
                        if rule[0 : len(nameB)] == nameB :
                                for rule1 in rulesB :
                                        newRulesA.append(rule1 +
rule[len(nameB) : ])
                        else :
                                newRulesA.append(rule)
                A.setRules(newRulesA)

        def solveImmediateLR(self, A) :
                name = A.getName()
                newName = name + """
                alphas = []
                betas = []
                rules = A.getRules()
                newRulesA = []
                newRulesA1 = []
                rules = A.getRules()
                for rule in rules :
```

```
        if rule[0 : len(name)] == name :
                alphas.append(rule[len(name) : ])
        else :
                betas.append(rule)

# If no left recursion, exit
if len(alphas) == 0 :
        return

if len(betas) == 0 :
        newRulesA.append(newName)

for beta in betas :
        newRulesA.append(beta + newName)

for alpha in alphas :
        newRulesA1.append(alpha + newName)

A.setRules(newRulesA)
newRulesA1.append("\u03B5")

# Adds new production rule
newNonTerminal = NonTerminal(newName)
```

```
                newNonTerminal.setRules(newRulesA1)
                self.nonTerminals.append(newNonTerminal)

        def applyAlgorithm(self) :
                size = len(self.nonTerminals)
                for i in range(size) :
                        for j in range(i) :
self.solveNonImmediateLR(self.nonTerminals[i], self.nonTerminals[j])
                        self.solveImmediateLR(self.nonTerminals[i])

        def printRules(self) :
                for nonTerminal in self.nonTerminals :
                        nonTerminal.printRule()



grammar = Grammar()
grammar.inputData()
grammar.applyAlgorithm()
grammar.printRules()
```

Q9) To implement any one parser first and follow:

```
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_index, char in enumerate(alt):
```

```
            if char == nT:
                following_str = alt[char_index + 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)
    return follow_

terminals = list(map(str, input("Enter the terminals: ").replace(',', ' ').split()))
non_terminals = list(map(str, input("Enter the non-terminals (First non-terminal should
be starting symbol): ").replace(',', ' ').split()))
starting_symbol = non_terminals[0]
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].replace('/', '|').split("|")
    for alternative in alternatives:
        productions_dict[nonterm_to_prod[0]].append(alternative)

FIRST = {}
FOLLOW = {}
for non_terminal in non_terminals:
    FIRST[non_terminal] = set()
    FIRST[non_terminal] = FIRST[non_terminal] | first(non_terminal)
```

```
for non_terminal in non_terminals:
    FOLLOW[non_terminal] = set()
    FOLLOW[non_terminal] = FOLLOW[non_terminal] | follow(non_terminal)

print("{: <15}\t{: ^30}\t{: ^20}".format('Non Terminals', 'First', 'Follow'))
for non_terminal in non_terminals:
    print("{: ^15}\t{: <30}\t{: <20}".format(non_terminal, str(sorted(FIRST[non_terminal])),
str(sorted(FOLLOW[non_terminal]))))
```

ONLY FIRST:

```
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_

terminals = list(map(str, input("Enter the terminals: ").replace(',', ' ').split()))

non_terminals = list(map(str, input("Enter the non-terminals (First non-terminal should
be starting symbol): ").replace(',', ' ').split()))

starting_symbol = non_terminals[0]

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].replace('/', '|').split("|")
    for alternative in alternatives:
        productions_dict[nonterm_to_prod[0]].append(alternative)
FIRST = {}
for non_terminal in non_terminals:
    FIRST[non_terminal] = set()
    FIRST[non_terminal] = FIRST[non_terminal] | first(non_terminal)
print("\nFirst Sets:")
print("{: <15}\t{: ^30}".format('Non Terminals', 'First'))
for non_terminal in non_terminals:
    print("{: ^15}\t{: <30}".format(non_terminal, str(sorted(FIRST[non_terminal]))))
```

ONLY FOLLOW

```
import sys
```

```
sys.setrecursionlimit(60)


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_index, char in enumerate(alt):
                if char == nT:
                    following_str = alt[char_index + 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)

    return follow_


terminals = list(map(str, input("Enter the terminals: ").replace(',', ' ').split()))

non_terminals = list(map(str, input("Enter the non-terminals (First non-terminal should
be starting symbol): ").replace(',', ' ').split()))

starting_symbol = non_terminals[0]

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].replace('/', '|').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[non_terminal] = FOLLOW[non_terminal] | follow(non_terminal)



print("\nFollow Sets:")

print("{: <15}\t{: ^20}".format('Non Terminals', 'Follow'))

for non_terminal in non_terminals:

    print("{: ^15}\t{: <20}".format(non_terminal, str(sorted(FOLLOW[non_terminal]))))
```

Output:

```
Enter the terminals: a,b,d,g,h
Enter the non-terminals (First non-terminal should be starting symbol): S,A,B,C
Enter no of productions: 4
Enter the productions:
S->ACB|CbB|Ba
A->da|BC
B->g|#
C->h|#
Non Terminals                First                       Follow
       S         ['#', 'a', 'b', 'd', 'g', 'h'] ['$']
       A         ['#', 'd', 'g', 'h']           ['$', 'g', 'h']
       B         ['#', 'g']                     ['$', 'g', 'h']
       C         ['#', 'h']                     ['$', 'g', 'h']
```

Q11) To implement Intermediate code generation (ex: Three Address Code)(JAVA/C/C++/Python/R-lang /Lex).

```
OPERATORS = set(['+', '-', '*', '/', '(', ')'])
PRI = {'+':1, '-':1, '*':2, '/':2}
```

```
def infix_to_postfix(formula):
    stack = [] # only pop when the coming op has priority
    output = ''
    for ch in formula:
        if ch not in OPERATORS:
            output += ch
        elif ch == '(':
            stack.append('(')
        elif ch == ')':
            while stack and stack[-1] != '(':
                output += stack.pop()
            stack.pop() # pop '('
        else:
            while stack and stack[-1] != '(' and PRI[ch] <= PRI[stack[-1]]:
                output += stack.pop()
            stack.append(ch)
    while stack:
            output += stack.pop()
    print("POSTFIX: {}".format(output))
    return output
def infix_to_prefix(formula):
    op_stack = []
    exp_stack = []
    for ch in formula:
        if not ch in OPERATORS:
            exp_stack.append(ch)
        elif ch == '(':
            op_stack.append(ch)
        elif ch == ')':
            while op_stack[-1] != '(':
                op = op_stack.pop()
                a = exp_stack.pop()
                b = exp_stack.pop()
                exp_stack.append( op+b+a )
            op_stack.pop() # pop '('
        else:
```

```
            while op_stack and op_stack[-1] != '(' and
               PRI[ch] <= PRI[op_stack[-1]]: op =
               op_stack.pop()
            a = exp_stack.pop()
             b = exp_stack.pop()
            exp_stack.append( op+b+a )
         op_stack.append(ch)
   # leftover
   while op_stack:
      op = op_stack.pop()
      a = exp_stack.pop()
      b = exp_stack.pop()
      exp_stack.append( op+b+a )
   print(f'PREFIX: {exp_stack[-1]}')
   return exp_stack[-1]
def generate3AC(pos):
        print("--- THREE ADDRESS CODE GENERATION ---")
        exp_stack = []
        t = 1
        for i in pos:
                if i not in OPERATORS:
                        exp_stack.append(i)
                else:
    print(f't{t} := {exp_stack[-2]} {i} {exp_stack[-1]}')
                        exp_stack=exp_stack[:-2]
                        exp_stack.append(f't{t}')
                        t+=1
expres = input("INPUT THE EXPRESSION: ")
pre = infix_to_prefix(expres)
pos = infix_to_postfix(expres)
generate3AC(pos)
```

```
INPUT THE EXPRESSION: a+b*c-d
PREFIX: -+a*bcd
POSTFIX: abc*+d-
--- THREE ADDRESS CODE GENERATION ---
t1 := b * c
t2 := a + t1
t3 := t2 - d
```

Q12)To study & implement Code Generation Algorithm. (JAVA/C/C++/Python/R-lang/Lex).

```
def infix_to_postfix(exp):
    precedence = {'+': 1, '-': 1, '*': 2, '/': 2}
    stack = []
    postfix = []
    for char in exp:
        if char.isalnum():
            postfix.append(char)
        elif char == '(':
            stack.append(char)
        elif char == ')':
            while stack and stack[-1] != '(':
                postfix.append(stack.pop())
            stack.pop()
        else:
            while stack and precedence.get(stack[-1], 0) >= precedence.get(char, 0):
postfix.append(stack.pop())
            stack.append(char)
```

```
    while stack:
        postfix.append(stack.pop())
    return postfix

def generate3AC(pos):
    three_address_code = []
    for token in pos:
        if token.isalnum():
            three_address_code.append(token)
        else:
            operand2 = three_address_code.pop()
            operand1 = three_address_code.pop()
            temp = f't{len(three_address_code) + 1}'
            three_address_code.append((temp, operand1, token, operand2))
    return three_address_code

op_code = {'+': 'ADD', '-': 'SUB', '*': 'MUL', '/': 'DIV'}

def code_gen(res):
    reg_idx = 1
    moved = {}
    curr_reg = {}
    for exps in res:
```

```
        print(f'\n#{exps[0]} = {exps[1]} {exps[2]} {exps[3]}')
        operands = [1, 3]
        new = []
        for x in operands:
            if exps[x] not in moved and '#' not in exps[x]:
                moved.update({exps[x]: reg_idx})
                reg_idx += 1
                new.append(exps[x])
            if exps[x] not in moved and '#' in exps[x]:
                moved.update({'#' + str(exps[x]).strip('#'): curr_reg[int(str(exps[x]).strip('#'))]})
        for x in new:
            print(f'MOV R{moved[x]}, {x}')
        print(f'{op_code[exps[2]]} R{moved[exps[1]]}, R{moved[exps[3]]}')
        curr_reg.update({exps[0]: moved[exps[1]]})



exp = input("Enter your expression: ")
pos = infix_to_postfix(exp)
res = generate3AC(pos)
code_gen(res)
```