1. Write a program to tokenize the input program(JUPYTER)

def lexical\_analysis(code):

# Create a list of tokens

tokens = []

# Split the code into lines

lines = code.split()

print(lines)

# Iterate through each line of code

for word in lines:

# Check if the word is a keyword

if word in ["int", "float", "double", "char", "void", "for", "while", "if", "else"]:

tokens.append(("keyword", word))

continue

# Check if the word is an operator

if word in ["+", "-", "\*", "/", "=", "==", "!=", ">", "<", ">=", "<="]:

tokens.append(("operator", word))

continue

# Check if the word is a punctuation mark

if word in [";", ",", "{", "}"]:

tokens.append(("punctuation", word))

continue

# If the word is none of the above, it is considered a identifier

tokens.append(("identifier", word))

return tokens

# Test the lexical analysis function

code = """

int main() {

int x = 5;

int y = 6;

int z = x + y;

return 0;

}

"""

lexical\_analysis(code)

#OUTPUT

def lexical\_analysis(code):

# Create a list of tokens

tokens = []

# Split the code into lines

lines = code.split()

print(lines)

# Iterate through each line of code

for word in lines:

# Check if the word is a keyword

if word in ["int", "float", "double", "char", "void", "for", "while", "if", "else"]:

tokens.append(("keyword", word))

continue

# Check if the word is an operator

if word in ["+", "-", "\*", "/", "=", "==", "!=", ">", "<", ">=", "<="]:

tokens.append(("operator", word))

continue

# Check if the word is a punctuation mark

if word in [";", ",", "{", "}"]:

tokens.append(("punctuation", word))

continue

# If the word is none of the above, it is considered a identifier

tokens.append(("identifier", word))

return tokens

# Test the lexical analysis function

code = """

int main() {

int x = 5;

int y = 6;

int z = x + y;

return 0;

}

"""

lexical\_analysis(code)

6. Write a program to compute FIRST and FOLLOW for the given CFG.(PYTHON)

def first\_follow(grammar, non\_terminal, first, follow):

if non\_terminal in first:

return

first[non\_terminal] = set()

for production in grammar[non\_terminal]:

for symbol in production:

if symbol in grammar:

first\_follow(grammar, symbol, first, follow)

first[non\_terminal] |= first[symbol]

else:

first[non\_terminal].add(symbol)

break

if '' in first[non\_terminal]:

follow[non\_terminal] = set()

for key in grammar:

for production in grammar[key]:

if non\_terminal in production:

index = production.index(non\_terminal)

if index == len(production) - 1:

follow[non\_terminal] |= follow[key]

else:

follow[non\_terminal] |= first[production[index+1]]

if '' not in first[production[index+1]]:

break

first[non\_terminal].remove('')

else:

follow[non\_terminal] = set()

grammar = {

'S': ['AB', 'CD'],

'A': ['a', ''],

'B': ['b'],

'C': ['c'],

'D': ['d']

}

first = {}

follow = {}

for non\_terminal in grammar:

first\_follow(grammar, non\_terminal, first, follow)

print(first)

print(follow)

#OUTPUT

{'S': {'d', 'b', 'c', 'a'}, 'A': {'a'}, 'B': {'b'}, 'C': {'c'}, 'D': {'d'}}

{'A': set(), 'B': set(), 'C': set(), 'D': set(), 'S': set()}

7. Write a Progam to Generate three address code for the given Input Expression(python)

operators = ['+', '-', '\*', '/', '(', ')']

precedance = {'+':1, '-':1, '\*':2, '/':2}

def infix\_postfix(exp):

stack = []

postfix = ""

for i in exp:

# print(f"stack = {stack} output {postfix}")

if i not in operators:

postfix+=i

elif i == "(":

stack.append(i)

elif i == ')':

while stack and stack[-1] != "(":

postfix += stack.pop()

stack.pop()

else:

while stack and stack[-1] != '(' and precedance[stack[-1]] >= precedance[i]:

postfix+=stack.pop()

stack.append(i)

while stack:

x=stack.pop()

if x != '(' or x!=')':

postfix+=x

print(postfix)

return postfix

def generate3AC(exp):

stack = []

t = 1

tac = []

for i in exp:

if i not in operators:

stack.append(i)

else:

print(f't{t} = {stack[-2]} {i} {stack[-1]}')

tac.append([f't{t} = {stack[-2]} {i} {stack[-1]}'])

stack = stack[:-2]

stack.append(f"t{t}")

t += 1

return tac

def main():

exp = input("enter expression without spac: ")

pos = infix\_postfix(exp)

print(generate3AC(pos))

main()

#OUTPUT

enter expression without spac: (A+B/C\*(D-A))

ABC/DA-\*+

t1 = B / C

t2 = D - A

t3 = t1 \* t2

t4 = A + t3

[['t1 = B / C'], ['t2 = D - A'], ['t3 = t1 \* t2'], ['t4 = A + t3']]

8. Write a Program to generate the postfix code for the given infix expression(PYTHON)

Operators = set(['+', '-', '\*', '/', '(', ')', '^']) # collection of Operators

Priority = {'+':1, '-':1, '\*':2, '/':2, '^':3} # dictionary having priorities of Operators

def infixToPostfix(expression):

stack = [] # initialization of empty stack

output = ''

Operators = set(['+', '-', '\*', '/', '(', ')', '^']) # collection of Operators

Priority = {'+':1, '-':1, '\*':2, '/':2, '^':3} # dictionary having priorities of Operators

def infixToPostfix(expression):

stack = [] # initialization of empty stack

output = ''

for character in expression:

if character not in Operators: # if an operand append in postfix expression

output+= character

elif character=='(': # else Operators push onto stack

stack.append('(')

elif character==')':

while stack and stack[-1]!= '(':

output+=stack.pop()

stack.pop()

else:

while stack and stack[-1]!='(' and Priority[character]<=Priority[stack[-1]]:

output+=stack.pop()

stack.append(character)

while stack:

output+=stack.pop()

return output

expression = input('Enter infix expression ')

print('infix notation: ',expression)

print('postfix notation: ',infixToPostfix(expression))

#OUTPUT

Enter infix expression m\*n+(p-q)+r

infix notation: m\*n+(p-q)+r

postfix notation: mn\*pq-+r+

9. Write a program to evaluate the postfix code(PYTHON)

def inf\_pos(exp):

operators = ['+', '-', '\*', '/', '(', ')']

precedence = {'+':1, '-':1, '\*':2, '/':2}

postfix = ""

stack = []

for i in exp:

print(f"stack: {stack} postfix: {postfix}")

if i not in operators:

postfix = postfix+i

if i in operators:

if len(stack)==0 or stack[-1]=="(":

stack.append(i)

elif i == '(':

stack.append(i)

elif i == ')':

while stack and stack[-1] != '(':

postfix += stack.pop()

stack.pop() # pop '('

else:

while(precedence[stack[-1]] >= precedence[i] or stack[-1] == ')'):

postfix = postfix+stack[-1]

stack.pop(-1)

if len(stack) == 0:

break

stack.append(i)

for item in stack:

print(f"stack: {stack} postfix: {postfix}")

if item == '(' or item == ')':

pass

else:

postfix = postfix+item

print(postfix)

return postfix

def generate3AC(pos):

print("### THREE ADDRESS CODE GENERATION ###")

exp\_stack = []

t = 1

op = []

for i in pos:

print(exp\_stack)

if i not in op:

exp\_stack.append(i)

else:

print(f't{t} := {exp\_stack[-2]} {i} {exp\_stack[-1]}')

op.append((f't{t} := {exp\_stack[-2]} {i} {exp\_stack[-1]}'))

exp\_stack=exp\_stack[:-2]

exp\_stack.append(f't{t}')

t+=1

return op

def main():

exp = input("enter expression without space: ")

pos = inf\_pos(exp)

print(generate3AC(pos))

main()

#OUTPUT

enter expression without space: (A+B/C\*(D-A))

stack: [] postfix:

stack: ['('] postfix:

stack: ['('] postfix: A

stack: ['(', '+'] postfix: A

stack: ['(', '+'] postfix: AB

stack: ['(', '+', '/'] postfix: AB

stack: ['(', '+', '/'] postfix: ABC

stack: ['(', '+', '\*'] postfix: ABC/

stack: ['(', '+', '\*', '('] postfix: ABC/

stack: ['(', '+', '\*', '('] postfix: ABC/D

stack: ['(', '+', '\*', '(', '-'] postfix: ABC/D

stack: ['(', '+', '\*', '(', '-'] postfix: ABC/DA

stack: ['('] postfix: ABC/DA-\*

stack: ['(', ')'] postfix: ABC/DA-\*

stack: ['(', ')'] postfix: ABC/DA-\*

ABC/DA-\*

### THREE ADDRESS CODE GENERATION ###

[]

['A']

['A', 'B']

['A', 'B', 'C']

['A', 'B', 'C', '/']

['A', 'B', 'C', '/', 'D']

['A', 'B', 'C', '/', 'D', 'A']

['A', 'B', 'C', '/', 'D', 'A', '-']

[]

>

10. Write a program to generate the target code based on the code production table for the

Intermediate representation in three-address code format.(PYTHON)

#a = 1;

# b = a + 1;

# c = b + 1;

# LOAD 1

# STORE a

# LOAD a

# ADD 1

# STORE b

# LOAD b

# ADD 1

# STORE c

# c = a+b

# LOAD A

# ADD B

# STORE C

# intermidiate\_code = [c=a+b, z=c+r]

# Input 3 address code

# Input 3 add code as a list of strings

# ins\_set = []

# Traverse the list using 'for' loop

# c = a+b

# add\_code\_divided = [c, =, a, +, b]

def machineCodeGen(intermidiate\_code):

ins\_set = []

for i in range(len(intermidiate\_code)):

line = intermidiate\_code[i].split()

if(len(line) == 5):

if (line[3] == "+"):

ins\_set.append(f"LOAD {line[2]}")

ins\_set.append(f"ADD {line[4]}")

ins\_set.append(f"STORE {line[0]}")

elif (line[3] == "-"):

ins\_set.append(f"LOAD {line[2]}")

ins\_set.append(f"SUB {line[4]}")

ins\_set.append(f"STORE {line[0]}")

elif (line[3] == "/"):

ins\_set.append(f"LOAD {line[2]}")

ins\_set.append(f"DIV {line[4]}")

ins\_set.append(f"STORE {line[0]}")

elif (line[3] == "\*"):

ins\_set.append(f"LOAD {line[2]}")

ins\_set.append(f"MUL {line[4]}")

ins\_set.append(f"STORE {line[0]}")

elif (len(line) == 3):

ins\_set.append(f"LOAD {line[2]}")

ins\_set.append(f"STORE {line[0]}")

else:

raise ValueError("Enter a better instruction.")

return ins\_set

code = ["c = a + b", "c = b + 1", "d = -a", "c = d"]

x = machineCodeGen(code)

for i in x:

print(i)

#OUTPUT

LOAD a

ADD b

STORE c

LOAD b

ADD 1

STORE c

LOAD -a

STORE d

LOAD d

STORE c