------------------ FIRST & FOLLOW ---------------

import sys

sys.setrecursionlimit(60)

def first(string):

#print("first({})".format(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:

#print("inside while")

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

#print('string[i:]=', string[i:])

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

#print("returning for first({})".format(string),first\_)

return first\_

def follow(nT):

#print("inside follow({})".format(nT))

follow\_ = set()

#print("FOLLOW", FOLLOW)

prods = productions\_dict.items()

if nT==starting\_symbol:

follow\_ = follow\_ | {'$'}

for nt,rhs in prods:

#print("nt to rhs", nt,rhs)

for alt in rhs:

for char in alt:

if char==nT:

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

if following\_str=='':

if nt==nT:

continue

else:

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

else:

follow\_2 = first(following\_str)

if '@' in follow\_2:

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

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

else:

follow\_ = follow\_ | follow\_2

#print("returning for follow({})".format(nT),follow\_)

return follow\_

no\_of\_terminals=int(input("Enter no. of terminals: "))

terminals = []

print("Enter the terminals :")

for \_ in range(no\_of\_terminals):

terminals.append(input())

no\_of\_non\_terminals=int(input("Enter no. of non terminals: "))

non\_terminals = []

print("Enter the non terminals :")

for \_ in range(no\_of\_non\_terminals):

non\_terminals.append(input())

starting\_symbol = input("Enter the starting symbol: ")

no\_of\_productions = int(input("Enter no of productions: "))

productions = []

print("Enter the productions:")

for \_ in range(no\_of\_productions):

productions.append(input())

#print("terminals", terminals)

#print("non terminals", non\_terminals)

#print("productions",productions)

productions\_dict = {}

for nT in non\_terminals:

productions\_dict[nT] = []

#print("productions\_dict",productions\_dict)

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)

#print("productions\_dict",productions\_dict)

#print("nonterm\_to\_prod",nonterm\_to\_prod)

#print("alternatives",alternatives)

FIRST = {}

FOLLOW = {}

for non\_terminal in non\_terminals:

FIRST[non\_terminal] = set()

for non\_terminal in non\_terminals:

FOLLOW[non\_terminal] = set()

#print("FIRST",FIRST)

for non\_terminal in non\_terminals:

FIRST[non\_terminal] = FIRST[non\_terminal] | first(non\_terminal)

#print("FIRST",FIRST)

FOLLOW[starting\_symbol] = FOLLOW[starting\_symbol] | {'$'}

for non\_terminal in non\_terminals:

FOLLOW[non\_terminal] = FOLLOW[non\_terminal] | follow(non\_terminal)

#print("FOLLOW", FOLLOW)

print("{: ^20}{: ^20}{: ^20}".format('Non Terminals','First','Follow'))

for non\_terminal in non\_terminals:

print("{: ^20}{: ^20}{: ^20}".format(non\_terminal,str(FIRST[non\_terminal]),str(FOLLOW[non\_terminal])))

E->TB

B->+TB/@

T->FY

Y->\*FY/@

F->a/(E)

---------------- CODE OPTIMIZATION -----------------

n=int(input('Enter number of operations '))

S=[]

for i in range(n) :

S.append(input())

optimizer = []

for i in range(len(S)) :

S[i] = S[i].split('=')

print(S[i])

for i in range(len(S)) :

for j in range(i+1, len(S)) :

if S[i][1] in S[j][1] :

for k in range(j-1, i-1, -1) :

if S[j][1][1] in S[k][0] or S[j][1][5] in S[k][0] :

break

else :

S[j][1] = S[i][0]

optimizer.append(S[i][0] + ' = ' + S[i][1])

for i in optimizer :

print(i)

----------- LEX TOOL --------------

import java.util.regex.Matcher;

import java.util.regex.Pattern;

import java.util.Scanner;

import java.util.\*;

public class Main

{

static ArrayList<Integer> count = new ArrayList<Integer>();

static ArrayList<String> elm = new ArrayList<String>();

static String[] operators = { "=", "+", "-", "\*", "/"};

static String[] keywords = { "abstract", "assert", "boolean",

"break", "byte", "case", "catch", "char", "class", "const",

"continue", "default", "do", "double", "else", "extends", "false",

"final", "finally", "float", "for", "goto", "if", "implements",

"import", "instanceof", "int", "interface", "long", "native",

"new", "null", "package", "private", "protected", "public",

"return", "short", "static", "strictfp", "super", "switch",

"synchronized", "String", "this", "throw", "throws", "transient", "true",

"try", "void", "volatile", "while" };

static String alphabet = "[a-zA-Z]+\\w\*";

static String[] deli = { "\t","\n",",",";","(",")","{","}","[","]","#","<",">"};

static Pattern numberpattern = Pattern.compile("-?\\d+(\\.\\d+)?");

static Pattern pattern = Pattern.compile(alphabet);

public static void main(String[] args)

{

Scanner sc = new Scanner(System.in);

String str = sc.nextLine();

String[] arr = str.split(" ");

count.add(0); // keyword

count.add(0); // Number

count.add(0); // Operator

count.add(0); // Identifier

count.add(0); // Invalid Indentifier

count.add(0); // Demimiter

elm.add("keyword");

elm.add("constant");

elm.add("operator");

elm.add("identifier");

elm.add("invalid identifier");

elm.add("delimiter");

for (String a : arr)

{

System.out.println(a + " -> " + token(a));

}

for (int i = 0; i < count.size(); i++)

{

System.out.println(elm.get(i) + " = " + count.get(i));

}

}

public static String token(String a)

{

for (String key : keywords)

{

if (key.equals(a))

{

count.set(0, count.get(0) + 1);

return "Keyword";

}

}

Matcher matchernum = numberpattern.matcher(a);

if (matchernum.matches())

{

count.set(1, count.get(1) + 1);

return "Constant";

}

for (String del : deli)

{

if (del.equals(a))

{

count.set(5, count.get(5) + 1);

return "Keyword";

}

}

for (String op : operators)

{

if (op.equals(a))

{

count.set(2, count.get(2) + 1);

return "Operator";

}

}

Matcher matcheralpha = pattern.matcher(a);

if (matcheralpha.matches())

{

count.set(3, count.get(3) + 1);

return "Identifier";

}

count.set(4, count.get(4) + 1);

return "Invalid Identifier";

}

}

---------------- 3AC, ICG -------------------------

import secrets

stack=[]

ans=[]

equation="a=((b+(c\*d))/e)"

g\_left=equation[0]

equation=equation[2:]

t=1

def generate\_random\_special\_character():

special\_characters = "!@#$&\_<>?[]|"

return secrets.choice(special\_characters)

map={}

def replacement(right):

for key,value in map.items():

right=right.replace(value,key)

return right

def solve(eq):

global t

precedence=['\*','/','+','-']

for op in precedence:

for i in range(len(eq)):

ch=eq[i]

if(ch==op):

t=str(t)

left='t'+t

right=eq[i-1]+op+eq[i+1]

right=replacement(right)

ans.append(left+'='+right)

t=int(t)

t=t+1

random=generate\_random\_special\_character()

while random in map.keys():

random=generate\_random\_special\_character()

map[left]=random

eq=eq[:i-1]+map[left]+eq[i+2:]

break

return eq

for ch in equation:

if ch=='(':

stack.append(ch)

elif ch==')':

eq=''

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

eq=stack.pop()+eq

opening=stack.pop()

res=solve(eq)

while(len(res)>1):

res=solve(res)

stack.append(res)

else :

stack.append(ch)

while len(ans)!=0:

pr=ans.pop(0)

print(pr)

final=g\_left+'='+pr[:2]

print(final)

--------- Assembler (MOT, POT, ST) -----------------

from tabulate import tabulate

class MOT:

def \_\_init\_\_(self):

self.instructions = {

"L": {"opcode": "00", "operands": 2},

"ST": {"opcode": "01", "operands": 2},

"A": {"opcode": "02", "operands": 2},

"S": {"opcode": "03", "operands": 2},

"M": {"opcode": "04", "operands": 2},

"D": {"opcode": "05", "operands": 2},

"JMP": {"opcode": "06", "operands": 1},

"HLT": {"opcode": "07", "operands": 0}

}

def get\_instruction(self, mnemonic):

return self.instructions.get(mnemonic, None)

class POT:

def \_\_init\_\_(self):

self.pseudo\_ops = {

"START": {"opcode": "", "operands": 1},

"END": {"opcode": "", "operands": 0},

"DC": {"opcode": "", "operands": 1},

"DS": {"opcode": "", "operands": 1},

"USING": {"opcode": "", "operands": 2} # Added USING as a pseudo-op with two operands

}

def get\_pseudo\_op(self, mnemonic):

return self.pseudo\_ops.get(mnemonic, None)

class SymbolTable:

def \_\_init\_\_(self):

self.table = {}

def add\_symbol(self, symbol, address):

self.table[symbol] = address

def get\_symbol\_address(self, symbol):

return self.table.get(symbol, None)

class Assembler:

def \_\_init\_\_(self):

self.mot = MOT()

self.pot = POT()

self.st = SymbolTable()

def assemble(self, source\_code):

machine\_code = []

address = 0

for line in source\_code:

parts = line.split()

mnemonic = parts[0]

if mnemonic in self.mot.instructions:

instruction = self.mot.get\_instruction(mnemonic)

if instruction['operands'] == 1:

operands = parts[1:]

if len(operands) != instruction['operands']:

raise ValueError(f"Invalid number of operands for {mnemonic} instruction.")

machine\_code.append(instruction['opcode'] + ''.join(operands))

elif instruction['operands'] == 2:

operands = parts[1].split(',')

if len(operands) != instruction['operands']:

raise ValueError(f"Invalid number of operands for {mnemonic} instruction.")

machine\_code.append(instruction['opcode'] + ''.join(operands))

elif mnemonic in self.pot.pseudo\_ops:

pseudo\_op = self.pot.get\_pseudo\_op(mnemonic)

if mnemonic == "START":

address = int(parts[1])

elif mnemonic == "END":

break

elif mnemonic == "DC":

machine\_code.append(parts[1])

elif mnemonic == "DS":

address += int(parts[1])

elif mnemonic == "USING":

continue # Ignore USING instruction for now

else:

raise ValueError(f"Unknown pseudo-operation: {mnemonic}")

else:

if mnemonic not in self.st.table:

self.st.add\_symbol(mnemonic, address)

else:

raise ValueError(f"Duplicate symbol found: {mnemonic}")

return machine\_code

def print\_pseudo\_op\_table(self):

table = []

print("Pseudo Opcode Table:")

for op, info in self.pot.pseudo\_ops.items():

table.append([op, ""])

print(tabulate(table, headers = ["Pseudo Opcode", "Address"], tablefmt = "github"))

def print\_symbol\_table(self):

table = []

print("Symbol Table:")

lc = 12

for symbol, address in self.st.table.items():

if symbol[:2] in 'PG' :

table.append([symbol, 0, 1, "R"])

else :

table.append([symbol, lc, 4, "R"])

lc += 4

print(tabulate(table, headers = ["Symbol", "Value", "Length", "R|A"], tablefmt="github"))

def print\_machine\_table(self):

table = []

instructions = ['L', 'A', 'ST', 'S', 'M', 'D']

print("Machine Opcode Table:")

for code in source\_code:

if code.split()[0] in instructions :

table.append([code.split()[0],"", "", "RX"])

print(tabulate(table, headers = ["Machine Opcode", "Binary Opcode", "Instruction Length", "Instruction Format"], tablefmt="github"))

# Example usage

source\_code = [

"PG1 START 0",

"USING \*,15",

"L 1, FIVE",

"A 1, FOUR",

"ST 1, TEMP",

"FOUR DC 4",

"FIVE DC 5",

"TEMP DS 1",

"END"

]

assembler = Assembler()

machine\_code = assembler.assemble(source\_code)

assembler.print\_pseudo\_op\_table()

assembler.print\_symbol\_table()

assembler.print\_machine\_table()

------------ QUADRAPULE, TRIPLE, ICG -----------------

class Quadruple:

def \_\_init\_\_(self, op, arg1=None, arg2=None, result=None):

self.op = op

self.arg1 = arg1

self.arg2 = arg2

self.result = result

def \_\_str\_\_(self):

return f"({self.op}, {self.arg1}, {self.arg2}, {self.result})"

class Triple:

def \_\_init\_\_(self, op, arg1=None, arg2=None):

self.op = op

self.arg1 = arg1

self.arg2 = arg2

def \_\_str\_\_(self):

return f"({self.op}, {self.arg1}, {self.arg2})"

class IntermediateCodeGenerator:

def \_\_init\_\_(self):

self.quadruples = []

self.triples = []

self.temp\_count = 1

def generate\_temp(self):

temp = f"t{self.temp\_count}"

self.temp\_count += 1

return temp

def generate\_quadruple(self, op, arg1=None, arg2=None, result=None):

quad = Quadruple(op, arg1, arg2, result)

self.quadruples.append(quad)

def generate\_triple(self, op, arg1=None, arg2=None):

triple = Triple(op, arg1, arg2)

self.triples.append(triple)

def generate\_code(self, expression):

tokens = expression.split('=')

result = tokens[0].strip()

expr = tokens[1].strip()

self.temp\_count = 1 # Reset temporary variable count for each expression

self.\_generate\_code(expr, result)

def \_generate\_code(self, expr, result):

stack = []

op\_stack = []

for token in expr:

if token.isalpha() or token.isdigit():

stack.append(token)

elif token in '+-\*/':

op\_stack.append(token)

elif token == ')':

op = op\_stack.pop()

arg2 = stack.pop()

arg1 = stack.pop()

temp = self.generate\_temp()

self.generate\_quadruple(op, arg1, arg2, temp)

self.generate\_triple(op, arg1, arg2)

stack.append(temp)

# Perform multiplication if there's a previous addition or subtraction operation

if len(op\_stack) > 0 and op\_stack[-1] in '\*/':

op = op\_stack.pop()

arg2 = stack.pop()

arg1 = stack.pop()

temp = self.generate\_temp()

self.generate\_quadruple(op, arg1, arg2, temp)

self.generate\_triple(op, arg1, arg2)

stack.append(temp)

self.generate\_quadruple('=', stack.pop(), None, result)

def display\_quadruples(self):

print("Quadruples:")

for quad in self.quadruples:

print(quad)

def display\_triples(self):

print("\nTriples:")

for triple in self.triples:

print(triple)

if \_\_name\_\_ == "\_\_main\_\_":

generator = IntermediateCodeGenerator()

# Example expression

expression = "a = (e - b) \* (c + d)"

generator.generate\_code(expression)

generator.display\_quadruples()

generator.display\_triples()

--------- BT, ST, LT, Assembler ------------

from tabulate import tabulate

class SymbolTable:

def \_\_init\_\_(self):

self.table = {}

def add\_symbol(self, symbol, address):

self.table[symbol] = address

def get\_symbol\_address(self, symbol):

return self.table.get(symbol, None)

def print\_table(self):

table = []

print("Symbol Table:")

lc = 12

for symbol, address in self.table.items():

if symbol[:2] in 'PG' :

table.append([symbol, 0, 1, "R"])

else :

table.append([symbol, lc, 4, "R"])

lc += 4

print(tabulate(table, headers = ["Symbol", "Value", "Length", "R|A"], tablefmt="github"))

class LiteralTable:

def \_\_init\_\_(self):

self.table = {}

def add\_literal(self, literal, address):

self.table[literal] = address

def get\_literal\_address(self, literal):

return self.table.get(literal, None)

def print\_table(self):

print("Literal Table:")

table = []

length = (len(source\_code)) \* 4

for literal, address in self.table.items():

table.append([literal, length, 4, "R"])

print(tabulate(table, headers = ["Literal", "Value", "Length", "R|A"], tablefmt="github"))

class BaseTable:

def \_\_init\_\_(self):

self.table = {}

def add\_base(self, base\_register, base\_address):

self.table[base\_register] = base\_address

def get\_base\_address(self, base\_register):

return self.table.get(base\_register, None)

def print\_table(self):

print("Base Table:")

table = []

for base\_register, base\_address in self.table.items():

table.append([base\_address, 1])

print(tabulate(table, headers = ['availability of indicator', 'Content of BR'], tablefmt="github"))

def process\_directives(source\_code):

symbol\_table = SymbolTable()

literal\_table = LiteralTable()

base\_table = BaseTable()

for line in source\_code:

parts = line.split()

directive = parts[0]

if directive == "ST":

symbol\_table.add\_symbol(parts[1], int(parts[2]))

elif directive == "LT":

literal\_table.add\_literal(parts[1], int(parts[2]))

elif directive == "USING":

base\_table.add\_base(parts[1], int(parts[2]))

elif directive == "=":

literal = parts[0]

value = int(parts[1][1:]) # Remove the '=' and parse the value

literal\_table.add\_literal(literal, value)

symbol\_table.print\_table()

literal\_table.print\_table()

base\_table.print\_table()

# Example usage

source\_code = [

"ST A 100",

"ST B 200",

"LT =1 300",

"LT =2 400",

"USING \* 15"

]

process\_directives(source\_code)

----------- LEX KEYWORD, IDENTIFIERS ----------

%{

int n = 0 ;

%}

%%

"while"|"if"|"else" {n++;printf("\t keywords : %s", yytext);}

"int"|"float" {n++;printf("\t keywords : %s", yytext);}

[a-zA-Z\_][a-zA-Z0-9\_]\* {n++;printf("\t identifier : %s", yytext);}

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

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

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

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

"end" {printf("\n total no. of token = %d\n", n);}

%%

int main()

{

yylex();

}

int yywrap () {

return 1;

}

Follow this below flow:->

gedit demo.l

flex demo.l

gcc lex.yy.c

./a.out

int i = 1000;

------------- MACROPROCESSOR (ALL) -------------

from tabulate import tabulate # To format tables

class MacroProcessor:

def \_\_init\_\_(self):

self.mnt = []

self.mdt = []

self.ala = []

def define\_macro(self, macro\_name, macro\_body, macro\_args):

mnt\_entry = {

'Index': len(self.mnt) + 1,

'MacroName': macro\_name,

'MDT\_Index': len(self.mdt) + 1

}

self.mnt.append(mnt\_entry)

for line in macro\_body:

mdt\_entry = {

'Index': len(self.mdt) + 1,

'Macro\_Definition': line

}

self.mdt.append(mdt\_entry)

for arg in macro\_args:

ala\_entry = {

'Index': len(self.ala) + 1,

'Argument': arg

}

self.ala.append(ala\_entry)

def list\_tables(self):

print("Macro Name Table (MNT):")

print(tabulate(self.mnt, headers="keys", tablefmt="grid"))

print("\nMacro Definition Table (MDT):")

print(tabulate(self.mdt, headers="keys", tablefmt="grid"))

print("\nArgument List Array (ALA):")

print(tabulate(self.ala, headers="keys", tablefmt="grid"))

def process\_macro(self, macro\_name, arg\_lists):

mnt\_entry = next((entry for entry in self.mnt if entry['MacroName'] == macro\_name), None)

if not mnt\_entry:

print(f"Macro '{macro\_name}' not found!")

return

mdt\_index = mnt\_entry['MDT\_Index']

mdt\_entries = self.mdt[mdt\_index - 1:]

macro\_body = []

for entry in mdt\_entries:

if "MEND" in entry['Macro\_Definition']:

break

macro\_body.append(entry['Macro\_Definition'])

expanded\_macros = []

for arg\_values in arg\_lists:

expanded\_macro = []

for line in macro\_body:

if line.startswith("MACRO"):

continue

expanded\_line = line

for arg\_name, arg\_value in zip(macro\_args, arg\_values):

expanded\_line = expanded\_line.replace(arg\_name, arg\_value)

expanded\_macro.append(expanded\_line)

expanded\_macros.append(expanded\_macro)

for expanded\_macro in expanded\_macros:

for line in expanded\_macro:

print(line)

processor = MacroProcessor()

macro\_body = ["MACRO INCR &ARG1, &ARG2", "A 1, &ARG1", "A 2, &ARG2", "MEND"]

macro\_args = ["&ARG1", "&ARG2"]

processor.define\_macro("INCR", macro\_body, macro\_args)

processor.list\_tables()

arg\_lists = [

["DATA1", "DATA2"],

["DATA3", "DATA4"]

]

processor.process\_macro("INCR", arg\_lists)

------------- LEX TOOL -----------------

import re

import keyword

keywords = [

"auto", "break", "case", "char", "const", "continue", "default", "do", "double",

"else", "enum", "extern", "float", "for", "goto", "if", "int", "long", "register",

"return", "short", "signed", "sizeof", "static", "struct", "switch", "typedef",

"union", "unsigned", "void", "volatile", "while"

]

operators = ["=", "+", "-", "\*", "/"]

alphabet = r"[a-zA-Z]+\w\*"

numberpattern = re.compile(r"-?\d+(\.\d+)?")

delimiters = {"(", ")", "{", "}", "[", "]", ",", ";", ".", ":", "'", '"'}

num\_keyw = 0

num\_cons = 0

num\_deli = 0

num\_oper = 0

num\_iden = 0

def analyze\_token(token):

global num\_keyw, num\_cons, num\_deli, num\_oper, num\_iden

# if keyword.iskeyword(token):

if token in keywords:

num\_keyw += 1

return "Keyword"

if numberpattern.fullmatch(token):

num\_cons += 1

return "Constant"

if token in delimiters:

num\_deli += 1

return "Delimiter"

if token in operators:

num\_oper += 1

return "Operator"

if re.fullmatch(alphabet, token):

num\_iden += 1

return "Identifier"

return "Invalid Token"

def lexical\_analyzer(code):

tokens = re.findall(r'[\w]+|[^\_\w\s]', code)

analyzed\_tokens = []

for token in tokens:

analyzed\_tokens.append((token, analyze\_token(token)))

return analyzed\_tokens

code = input("Enter your code: ")

analyzed\_tokens = lexical\_analyzer(code)

print("{: ^15}{: ^10}{: ^10}".format("Token", "|", "Token Type"))

print("-------------------------------------------")

for token, token\_type in analyzed\_tokens:

print("{: ^15}{: ^10}{: ^10}".format(token, '|', token\_type))

print(f"\n\nNo. of Keywords: \t{num\_keyw}\nNo. of Operators: \t{num\_oper}\nNo. of Identifiers: \t{num\_iden}\nNo. of Delimiters: \t{num\_deli}\nNo. of Constant: \t{num\_cons}")

—---- FIRST & FOLLOW (gayatri version) —------------------

print('input productions')

print('~ is being used for epsilon')

p = {}

while True:

add = input()

if add == 'end':

break

else:

[lhs, rhs] = list(add.split(' -> '))

prods = list(rhs.split('|'))

p[lhs] = prods

#print(p)

vars = list(p.keys())[::-1]

terms = []

t = []

for i in p.values():

for j in i:

t.extend([\*j])

t = list(set(t))

for i in t:

if i not in vars and i!='~':

terms.append(i)

print(vars)

#print(terms)

Firsts = {}

#FIRST

def First(X):

#print('='\*60)

#print('X =', X)

if X in terms:

#print('term')

Firsts[X] = [X]

return [X]

if X=='~':

#print('ep')

Firsts[X] = ['~']

return ['~']

#print('var')

first = []

R = p[X]

#print('rhs-->', R)

for i in R:

#print('checking', i, 'in', R)

add = []

if i[0] not in Firsts.keys():

Firsts[i[0]] = First(i[0])

if '~' not in Firsts[i[0]]:

first.extend(Firsts[i[0]])

continue

s=0

while '~' in Firsts[i[s]]:

add.extend(Firsts[i[s]])

add.remove('~')

if s+1!=len(i):

s+=1

else:

add.append('~')

break

if s!=len(i)-1:

add.extend(Firsts[i[s]])

#print('being added-> ', add)

first.extend(add)

return first

for i in terms:

Firsts[i] = First(i)

print('First of', i, 'is', list(set(Firsts[i])))

for i in vars:

Firsts[i] = First(i)

print('First of', i, 'is', list(set(Firsts[i])))

#yayy :D

print('='\*60)

#FOLLOW

#only vars

def Follow(X):

follow = []

for i in p.keys():

R = p[i]

for j in R:

if X in j:

add = []

s = 0

while j[s:].count(X)!=0:

ind = j[s:].index(X) + s

if ind==len(j)-1:

if i!=X:

if i not in Follows.keys():

Follows[i] = Follow(i)

add.extend(Follows[i])

break

else:

if j[ind+1] not in Firsts.keys():

Firsts[j[ind+1]] = First(j[ind+1])

if '~' not in Firsts[j[ind+1]]:

add.extend(Firsts[j[ind+1]])

while '~' in Firsts[j[ind+1]]:

add.extend(Firsts[j[ind+1]])

add.remove('~')

if ind+2!=len(j):

ind+=1

else:

#follow of the start lhs

if i!=X:

if i not in Follows.keys():

Follows[i] = Follow(i)

add.extend(Follows[i])

break

s = ind+1

follow.extend(add)

return follow

Follows = {}

Follows[vars[-1]] = ['$'] + Follow(vars[-1])

print('Follow of', vars[-1], 'is', Follows[vars[-1]])

for i in vars[:-1][::-1]:

Follows[i] = Follow(i)

print('Follow of', i, 'is', list(set(Follows[i])))