```
---------- 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])))
```