First and 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])))
```

Code optimization: common subexpression elimination and algebraic simplification

```
import re
def optimize_expression(expression):
    # Step 1: Common subexpression elimination
    expression, substitutions = eliminate_common_subexpressions(expression)
    # Step 2: Algebraic simplification
    expression = simplify_algebraic(expression)
    return expression
def eliminate_common_subexpressions(expression):
    # Regular expression to match subexpressions within parentheses
    pattern = re.compile(r'\((.*?)\)')
    matches = pattern.findall(expression)
    # Dictionary to store identified subexpressions and their replacement variables
    substitutions = {}
    for match in matches:
        # Check if the subexpression appears more than once
        count = expression.count(match)
        if count > 1:
            # Check if the subexpression is already substituted
            if match not in substitutions:
                # Replace all occurrences of the subexpression with a unique variable
                variable = f'_{len(substitutions)}'
                expression = expression.replace(match, variable, count - 1)
                # Store the subexpression and its replacement variable
                substitutions[match] = variable
    return expression, substitutions
def simplify_algebraic(expression):
    # Step 2: Algebraic simplification
    # In this example, we'll implement simple algebraic simplifications
    # Replace addition of 0 or multiplication by 1
    expression = re.sub(r'\b0\+(\w+)', r'\1', expression)
    expression = re.sub(r'(\w+)\*1\b', r'\1', expression)
    # Replace multiplication by 0 with 0
    expression = re.sub(r'(\w+)\*0\b', r'0', expression)
    # Replace multiplication by -1 with negation
    expression = re.sub(r'(\w+)\*-1\b', r'-\1', expression)
    return expression
# Example usage
expression = "(2 * x + y) * (2 * x + z) - (2 * x + y)"
print("Original expression:", expression)
optimized_expression = optimize_expression(expression)
print("Optimized expression:", optimized_expression)
```

Lexical Analyser

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <math.h>

// Function to check if a string is a keyword
int isKeyword(char *str) {
    char keywords[][10] = {"int", "char", "float", "double", "if", "else", "while", "for", "return"};
    int num_keywords = sizeof(keywords) / sizeof(keywords[0]);
    for (int i = 0; i < num_keywords; i++) {
        if (strcmp(str, keywords[i]) == 0)
            return 1; // Keyword found
    }
    return 0; // Not a keyword
}
// Function to check if a character is an operator
int isOperator(char ch) {
    return (ch == '+' || ch == '-' || ch == '*' || ch == '/' || ch == '%' || ch == '=' || ch == ';' || ch == '(' ||
ch == ')');
}
int main() {
    int i, identifier = 0, number = 0, operator = 0, keyword = 0;
    char s[100]; // Increased size to accommodate longer statements
    printf("Enter Statement: ");
    fflush(stdout);
    fgets(s, sizeof(s), stdin); // Read input as a whole line
    char *token = strtok(s, " "); // Tokenize the input string by space
    while (token != NULL) {
        // Check if the token is a keyword
        if (isKeyword(token)) {
            printf("%s is a keyword.\n", token);
            keyword++;
        }
        // Check if the token is an identifier
        else if (isalpha(token[0])) {
            printf("%s is an identifier.\n", token);
            identifier++;
        }
        // Check if the token is a number
        else if (isdigit(token[0])) {
            printf("%s is a number.\n", token);
            number++;
        }
        // Check if the token is an operator
        else if (isOperator(token[0])) {
            printf("%s is an operator.\n", token);
```

```
            operator++;
        }
        // If none of the above, it's not a valid token
        else {
            printf("%s is not a valid token.\n", token);
        }

        token = strtok(NULL, " "); // Move to the next token
    }
    printf("Total identifiers: %d\n", identifier);
    printf("Total numbers: %d\n", number);
    printf("Total operators: %d\n", operator);
    printf("Total keywords: %d\n", keyword);
    printf("Total tokens: %d\n", (identifier + number + operator + keyword));
    return 0;
}
```

3AC(cant handle nums)

```
import secrets
stack = []
ans = []
equation = "a=(b+c+d)"
# equation = "a=((c*d)+(a+b)+(a+b))"
# 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)
```

7. Assembler design: MOT, POT, ST

```
class MOT:
    def __init__(self):
        self.instructions = {
            "LOAD": {"opcode": "00", "operands": 1},
            "STORE": {"opcode": "01", "operands": 1},
            "ADD": {"opcode": "02", "operands": 1},
            "SUB": {"opcode": "03", "operands": 1},
            "MULT": {"opcode": "04", "operands": 1},
            "DIV": {"opcode": "05", "operands": 1},
            "JUMP": {"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):
      print("Pseudo Operation Table:")
      for op, info in self.pot.pseudo_ops.items():
         print(f"Mnemonic: {op}, Opcode: {info['opcode']}, Operands: {info['operands']}")
   def print_symbol_table(self):
      print("Symbol Table:")
      for symbol, address in self.st.table.items():
         print(f"Symbol: {symbol}, Address: {address}")
# Example usage
source_code = [
   "PG1 START 0",
   "USING *,15",
   "LOAD FOUR",
   "STORE FIVE",
   "ADD FOUR",
   "FOUR DC 5",
   "FIVE DC 5",
   "TEMP DS 1",
   "END"
]
assembler = Assembler()
machine_code = assembler.assemble(source_code)
print("Machine Code:")
for code in machine_code:
   print(code)
```

```
assembler.print_pseudo_op_table()
assembler.print_symbol_table()
```

8. Assembler Design: ST, LT, BT

```
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):
        print("Symbol Table:")
        for symbol, address in self.table.items():
            print(f"Symbol: {symbol}, Address: {address}")
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:")
        for literal, address in self.table.items():
            print(f"Literal: {literal}, Address: {address}")
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:")
        for base_register, base_address in self.table.items():
            print(f"Base Register: {base_register}, Base Address: {base_address}")
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 == "BT":
            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",
    "BT BASE 500"
]
process_directives(source_code)
```

Intermediate code generator: Quadruple and Triple representation

```
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()
```

Lex tool

```
%{
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;
```