**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;

**10: Macro processor: MDT, MNT, ALA**

# Macro processor class remains unchanged

class MacroProcessor:

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

self.mnt = []

self.mdt = []

self.ala = {}

def define\_macro(self, name, parameters, body):

mnt\_entry = {"name": name, "mdt\_index": len(self.mdt), "param\_count": len(parameters)}

self.mnt.append(mnt\_entry)

mdt\_entry = {

"name": name,

"parameters": parameters,

"body": body,

}

self.mdt.append(mdt\_entry)

def expand\_macro(self, name, arguments):

macro\_entry = None

for entry in self.mnt:

if entry["name"] == name:

macro\_entry = entry

break

if not macro\_entry:

raise ValueError(f"Macro '{name}' is not defined.")

mdt\_index = macro\_entry["mdt\_index"]

macro\_definition = self.mdt[mdt\_index]

parameters = macro\_definition["parameters"]

body = macro\_definition["body"]

if len(arguments) != len(parameters):

raise ValueError(f"Expected {len(parameters)} arguments, but got {len(arguments)}.")

self.ala[name] = {parameters[i]: arguments[i] for i in range(len(parameters))}

expanded\_body = []

for line in body:

expanded\_line = line

for param, arg in self.ala[name].items():

expanded\_line = expanded\_line.replace(f"&{param}", arg)

expanded\_body.append(expanded\_line)

return expanded\_body

def show\_mnt(self):

print("MNT:")

for entry in self.mnt:

print(f" {entry}")

def show\_mdt(self):

print("MDT:")

for entry in self.mdt:

print(f" {entry}")

def show\_ala(self):

print("ALA:")

for macro, args in self.ala.items():

print(f" Macro: {macro}, Arguments: {args}")

# New Macro definition

processor = MacroProcessor()

macro\_name\_add = "ADD"

macro\_parameters\_add = ["X", "Y"]

macro\_body\_add = ["LOAD &X", "ADD &Y", "STORE RESULT"]

macro\_name\_seq = "SEQUENCE"

macro\_parameters\_seq = ["A", "B", "C"]

macro\_body\_seq = [

"MOVE &A, &B",

"ADD &B, &C",

"SUB &A, &B",

"MUL &C, &A",

"DIV &C, &B",

]

processor.define\_macro(macro\_name\_add, macro\_parameters\_add, macro\_body\_add)

processor.define\_macro(macro\_name\_seq, macro\_parameters\_seq, macro\_body\_seq)

expanded\_add = processor.expand\_macro("ADD", ["A", "B"])

expanded\_seq = processor.expand\_macro("SEQUENCE", ["X", "Y", "Z"])

print("Expanded ADD Macro:")

for line in expanded\_add:

print(f" {line}")

print("Expanded SEQUENCE Macro:")

for line in expanded\_seq:

print(f" {line}")

processor.show\_mnt()

processor.show\_mdt()

processor.show\_ala()

**Macro processor: Expanded code**

class MacroProcessor:

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

self.mnt = []

self.mdt = []

self.ala = {}

def define\_macro(self, name, parameters, body):

mnt\_entry = {"name": name, "mdt\_index": len(self.mdt), "param\_count": len(parameters)}

self.mnt.append(mnt\_entry)

mdt\_entry = {

"name": name,

"parameters": parameters,

"body": body,

}

self.mdt.append(mdt\_entry)

def expand\_macro(self, name, arguments):

macro\_entry = None

for entry in self.mnt:

if entry["name"] == name:

macro\_entry = entry

break

if not macro\_entry:

raise ValueError(f"Macro '{name}' is not defined.")

mdt\_index = macro\_entry["mdt\_index"]

macro\_definition = self.mdt[mdt\_index]

parameters = macro\_definition["parameters"]

body = macro\_definition["body"]

if len(arguments) != len(parameters):

raise ValueError(f"Expected {len(parameters)} arguments, but got {len(arguments)}.")

self.ala[name] = {parameters[i]: arguments[i] for i in range(len(parameters))}

expanded\_body = []

for line in body:

expanded\_line = line

for param, arg in self.ala[name].items():

expanded\_line = expanded\_line.replace(f"&{param}", arg)

expanded\_body.append(expanded\_line)

return expanded\_body

# Create a new macro with a 5-line body

processor = MacroProcessor()

macro\_name\_seq = "SEQUENCE"

macro\_parameters\_seq = ["A", "B", "C"]

macro\_body\_seq = [

"MOVE &A, &B",

"ADD &B, &C",

"SUB &A, &B",

"MUL &C, &A",

"DIV &C, &B",

]

processor.define\_macro(macro\_name\_seq, macro\_parameters\_seq, macro\_body\_seq)

# Expand the macro with specific arguments

expanded\_seq = processor.expand\_macro("SEQUENCE", ["X", "Y", "Z"])

# Output the expanded macro body

print("Expanded SEQUENCE Macro:")

for line in expanded\_seq:

print(f" {line}")