**BT\_table.py :**

# Simplified BT (Base Table) Two-Pass Assembler

def bt\_assembler(lines):

    symbol\_table = {}

    base\_table = {}

    location\_counter = 0

    # PASS 1

    for line in lines:

        parts = line.strip().split()

        if not parts: continue

        label, opcode, operand = (parts + ["", ""])[:3]

        if opcode == "START":

            location\_counter = int(operand)

        if label:

            symbol\_table[label] = location\_counter

        if opcode == "USING":

            base\_table[operand.split(',')[0]] = location\_counter

        location\_counter += 4

    # PASS 2

    print("\nMachine Code:")

    loc = 0

    for line in lines:

        parts = line.strip().split()

        if not parts: continue

        label, opcode, operand = (parts + ["", ""])[:3]

        print(f"{loc:04d}: {opcode} {operand}")

        loc += 4

    print("\nSymbol Table:", symbol\_table)

    print("Base Table:", base\_table)

# Sample

bt\_assembler([

    "COPY START 0",

    "FIRST USING \*,15",

    "L 1,FOUR",

    "A 1,FIVE",

    "ST 1,TEMP",

    "FOUR DC F'4'",

    "FIVE DC F'5'",

    "TEMP DS 1F",

    "END"

])

**Codegeneration.py:**

import re

def generate\_assembly(expression):

    """

    Generate assembly code from a simple arithmetic expression.

    Supports +, -, \*, and / operations.

    """

    operators = {'+': 'ADD', '-': 'SUB', '\*': 'MUL', '/': 'DIV'}

    assembly\_code = []

    temp\_counter = 1

    # Tokenize the expression

    tokens = re.findall(r'\d+|[+\-\*/()]', expression)

    def evaluate\_postfix(postfix):

        """

        Generate assembly code for a postfix expression.

        """

        nonlocal temp\_counter

        stack = []

        for token in postfix:

            if token.isdigit():  # Operand

                stack.append(token)

            elif token in operators:  # Operator

                operand2 = stack.pop()

                operand1 = stack.pop()

                temp\_var = f"T{temp\_counter}"

                temp\_counter += 1

                assembly\_code.append(f"MOV R1, {operand1}")

                assembly\_code.append(f"MOV R2, {operand2}")

                assembly\_code.append(f"{operators[token]} R1, R2")

                assembly\_code.append(f"MOV {temp\_var}, R1")

                stack.append(temp\_var)

        return stack[0]  # Final result variable

    def infix\_to\_postfix(tokens):

        """

        Convert infix tokens to postfix notation.

        """

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

        stack = []

        postfix = []

        for token in tokens:

            if token.isdigit():

                postfix.append(token)

            elif token in operators:

                while stack and precedence.get(stack[-1], 0) >= precedence[token]:

                    postfix.append(stack.pop())

                stack.append(token)

            elif token == '(':

                stack.append(token)

            elif token == ')':

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

                    postfix.append(stack.pop())

                stack.pop()  # Remove '('

        while stack:

            postfix.append(stack.pop())

        return postfix

    # Convert infix to postfix and generate assembly code

    postfix = infix\_to\_postfix(tokens)

    final\_result = evaluate\_postfix(postfix)

    assembly\_code.append(f"; Final result stored in {final\_result}")

    assembly\_code.append(f"PRINT {final\_result}  ; Output the final result")

    return '\n'.join(assembly\_code)

# Example usage

python\_expression = input("Enter a Python arithmetic expression: ")

assembly\_output = generate\_assembly(python\_expression)

print("\n### Generated Assembly Code ###")

print(assembly\_output)

**intermmediate.py :**

import re

def get\_temp():

    global temp\_count

    temp = f"T{temp\_count}"

    temp\_count += 1

    return temp

def infix\_to\_postfix(tokens):

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

    output, stack = [], []

    for token in tokens:

        if token.isdigit():

            output.append(token)

        elif token in precedence:

            while stack and stack[-1] != '(' and precedence.get(stack[-1], 0) >= precedence[token]:

                output.append(stack.pop())

            stack.append(token)

        elif token == '(':

            stack.append(token)

        elif token == ')':

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

                output.append(stack.pop())

            stack.pop()

    while stack:

        output.append(stack.pop())

    return output

def postfix\_to\_TAC(postfix):

    tac\_code, stack = [], []

    for token in postfix:

        if token.isdigit():

            stack.append(token)

        else:

            b = stack.pop()

            a = stack.pop()

            temp = get\_temp()

            tac\_code.append(f"{temp} = {a} {token} {b}")

            stack.append(temp)

    return tac\_code

# Main program

temp\_count = 1

expr = input("Enter an arithmetic expression: ")

tokens = re.findall(r'\d+|\+|\-|\\*|\/|\(|\)', expr)

postfix = infix\_to\_postfix(tokens)

tac = postfix\_to\_TAC(postfix)

print("\nGenerated Three-Address Code:")

for line in tac:

    print(line)

**lexical anal.py :**

import re

keywords = {"int", "float", "if", "else", "while", "for","char", "double","include"}

operators = {"+", "-", "\*", "/", "%", "=", "==", "!=", "<", ">", "<=", ">=", "&&", "||", "++", "--"}

punctuations = {",", ";", "(", ")", "{", "}", "[", "]"}

def lexical\_analyzer(code):

    #splitting the code into tokens

    tokens = re.findall(r'\w+|\S', code)

    header = []

    found\_keywords = []

    found\_operators = []

    found\_punctuations = []

    found\_identifiers = []

    found\_constants = []

    for token in tokens:

        if token in keywords:

            found\_keywords.append(token)

        elif re.match(r'\d+', token):

            found\_constants.append(token)

        elif token in operators:

            found\_operators.append(token)

        elif token in punctuations :

            found\_punctuations.append(token)

        #for header files

        elif re.match(r'#\s\*include\s\*<[\w.]+>', token):

                headers.append(token)  # It's a header file like <stdio.h>

        elif re.match(r'[a-zA-Z\_]\w\*', token):

                found\_identifiers.append(token)

    print("header files : ", header)

    print("keywords : ", found\_keywords)

    print("operators : ", found\_operators)

    print("punctuation marks : ", found\_punctuations)

    print("constants : ", found\_constants)

    print("identifiers : ", found\_identifiers)

    #sample c code

code = """

    #include<math.h>

    double power(double base, int exp){

        double result = 1.0;

        while(exp>0){

            result = result\*base;

            exp --;}

        return result;} """

print("Input Program")

print(code)

print("\nLEXICAL ANALYZER :\n")

lexical\_analyzer(code)

**LT\_table.py**

def lt\_assembler(lines):

    symbol\_table = {}

    literal\_table = {}

    intermediate\_code = []

    location\_counter = 0

    # PASS 1

    for line in lines:

        parts = line.strip().split()

        if not parts: continue

        if len(parts) == 3:

            label, opcode, operand = parts

            symbol\_table[label] = location\_counter

        elif len(parts) == 2:

            opcode, operand = parts

            label = ""

        else:

            opcode = parts[0]

            label = ""

            operand = ""

        if operand.startswith("="):

            literal\_table.setdefault(operand, None)

        intermediate\_code.append((location\_counter, label, opcode, operand))

        location\_counter += 1

    # Assign addresses to literals

    for lit in literal\_table:

        literal\_table[lit] = location\_counter

        location\_counter += 1

    # PASS 2

    print("\nMachine Code:")

    for loc, label, opcode, operand in intermediate\_code:

        address = None

        if operand.startswith("="):

            address = literal\_table[operand]

        elif operand in symbol\_table:

            address = symbol\_table[operand]

        elif operand.isdigit():

            address = operand

        else:

            address = operand  # keep as it is

        print(f"{loc:04d}: {opcode} {address}")

    # Now generate literal pool

    for lit, addr in literal\_table.items():

        value = lit.strip("=")

        print(f"{addr:04d}: DC {value}")

    print("\nSymbol Table:", symbol\_table)

    print("Literal Table:", literal\_table)

# SAMPLE

lt\_assembler([

    "START 0",

    "LOAD =5",

    "ADD =10",

    "STORE TEMP",

    "TEMP DS 1",

    "END"

])

**Macro\_assembler.py :**

# Initialize the MNT and MDT tables

def initialize\_tables():

    mnt = []  # Macro Name Table (MNT)

    mdt = []  # Macro Definition Table (MDT)

    return mnt, mdt

# Display MNT and MDT

def display\_tables(mnt, mdt, pass\_num, stage=""):

    print(f"\n--- Pass {pass\_num} {stage} ---")

    print("\nMNT (Macro Name Table):")

    for entry in mnt:

        print(entry)

    print("\nMDT (Macro Definition Table):")

    for entry in mdt:

        print(entry)

# Process the macro from input assembly code

def process\_macro(input\_file):

    mnt, mdt = initialize\_tables()  # Initialize MNT and MDT

    pass1\_output = []

    pass2\_output = []

    with open(input\_file, 'r') as f:

        lines = f.readlines()

    i = 0

    while i < len(lines):

        line = lines[i].strip()

        # If it's a macro definition

        if line.upper() == "MACRO":

            i += 1

            macro\_def = lines[i].strip().split()

            macro\_name = macro\_def[0]

            parameters = macro\_def[1:] if len(macro\_def) > 1 else []

            mnt.append((macro\_name, len(mdt)))  # Insert macro name with its MDT index

            i += 1

            macro\_body = []

            while i < len(lines) and lines[i].strip().upper() != "MEND":

                current\_line = lines[i].strip()

                # Replace parameters with #index in the macro body

                for param in parameters:

                    current\_line = current\_line.replace(param, "#" + str(parameters.index(param)))

                macro\_body.append(current\_line)

                i += 1

            mdt.append(macro\_body)  # Insert macro body into MDT

            mdt.append(["MEND"])  # MEND marks the end of the macro

            display\_tables(mnt, mdt, 1, f"After defining {macro\_name}")

        else:

            pass1\_output.append(line)  # Regular assembly code for Pass 1

            i += 1

    print("\n##### Pass 1 Output #####")

    for idx, line in enumerate(pass1\_output, 1):

        print(f"{idx}\t{line}")

    # Pass 2: Expand macros in the assembly code

    print("\n##### Pass 2 Output #####")

    for line in pass1\_output:

        parts = line.split()

        if parts[0] in [macro[0] for macro in mnt]:  # If macro is called

            macro\_name = parts[0]

            args = parts[1:] if len(parts) > 1 else []

            # Find the macro definition in MDT using MNT

            for macro in mnt:

                if macro[0] == macro\_name:

                    mdt\_index = macro[1]

                    break

            macro\_body = mdt[mdt\_index]

            for definition in macro\_body:

                expanded\_line = definition

                for i in range(len(expanded\_line.split())):

                    if expanded\_line.split()[i].startswith("#"):

                        arg\_index = int(expanded\_line.split()[i][1:])

                        if arg\_index < len(args):

                            expanded\_line = expanded\_line.replace(expanded\_line.split()[i], args[arg\_index])

                pass2\_output.append(expanded\_line)

        else:

            pass2\_output.append(line)

    # Output the final expanded code after Pass 2

    print("\n##### Final Pass 2 Output #####")

    for idx, line in enumerate(pass2\_output, 1):

        print(f"{idx}\t{line}")

# Main function to drive the process

def main():

    input\_file = "input.txt"

    process\_macro(input\_file)

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

    main()

**operatorprecedence.py :**

class OperatorPrecedenceParser:

    precedence = {

        '+': {'+': '>', '\*': '<', '-': '>', 'a': '<', '$': '>'},

        '\*': {'+': '>', '\*': '>', '-': '>', 'a': '<', '$': '>'},

        '-': {'+': '<', '\*': '<', '-': '>', 'a': '<', '$': '>'},

        'a': {'+': '>', '\*': '>', '-': '>', 'a': '>', '$': '>'},

        '$': {'+': '<', '\*': '<', '-': '<', 'a': '<', '$': 'A'}

    }

    def \_\_init\_\_(self, expr):

        self.expr = ['$'] + list(expr.replace(" ", "")) + ['$']

        self.stack = ['$']

    def top\_terminal(self):

        for sym in reversed(self.stack):

            if sym in self.precedence:

                return sym

        return '$'

    def precedence\_of(self, top, next\_sym):

        return self.precedence.get(top, {}).get(next\_sym, ' ')

    def reduce(self):

        for i in range(len(self.stack) - 1, -1, -1):

            if self.stack[i] == 'a':

                self.stack[i] = 'E'

                return True, "Reduce E → a"

            if i >= 2 and self.stack[i-2] == 'E' and self.stack[i-1] in '+-\*' and self.stack[i] == 'E':

                op = self.stack[i-1]

                self.stack[i-2] = 'E'

                del self.stack[i-1:i+1]

                return True, f"Reduce E → E {op} E"

        return False, None

    def is\_valid(self):

        prev = ''

        for i, ch in enumerate(self.expr):

            if ch in '+-\*' and prev in '+-\*':

                return False, i

            prev = ch

        return True, -1

    def parse(self):

        print(f"{'STACK':<20}{'INPUT':<20}{'OUTPUT'}")

        print("=" \* 60)

        valid, err\_idx = self.is\_valid()

        if not valid:

            print(f"{' '.join(self.stack):<20}{' '.join(self.expr[1:-1]):<20}Error: Consecutive Operators")

            print("Not Accepted")

            return

        idx = 1

        while idx < len(self.expr):

            top = self.top\_terminal()

            next\_sym = self.expr[idx]

            prec = self.precedence\_of(top, next\_sym)

            if prec in '<=':

                self.stack.append(next\_sym)

                print(f"{' '.join(self.stack):<20}{' '.join(self.expr[idx+1:]):<20}Shift {next\_sym}")

                idx += 1

            elif prec == '>':

                reduced, action = self.reduce()

                if reduced:

                    print(f"{' '.join(self.stack):<20}{' '.join(self.expr[idx:]):<20}{action}")

                else:

                    print("Error: No valid reduction")

                    return

            elif prec == 'A':

                print(f"{' '.join(self.stack):<20}{' '.join(self.expr[idx:]):<20}Parsing done.")

                print("Accepted")

                return

            else:

                print("Error: Invalid precedence lookup")

                return

        while True:

            reduced, action = self.reduce()

            if reduced:

                print(f"{' '.join(self.stack):<20}{'':<20}{action}")

            else:

                break

        if self.stack == ['$', 'E']:

            print(f"{' '.join(self.stack):<20}{'':<20}Parsing done.")

            print("Accepted")

        else:

            print("Error: Expression not fully reduced.")

            print("Not Accepted")

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

    expr = input("Enter an arithmetic expression: ")

    parser = OperatorPrecedenceParser(expr)

    parser.parse()

optimised using constant prop.py :

# Input

n = int(input("Enter the number of expressions: "))

print("Enter the expressions in the format 'left = right':")

expressions = [{"left": input("Left: ").strip(), "right": input("Right: ").strip()} for \_ in range(n)]

# Intermediate Code

print("\nIntermediate Code:")

for expr in expressions:

    print(f"{expr['left']} = {expr['right']}")

# Step 1: Identify Constants

constants = {}

optimized\_exprs = []

for expr in expressions:

    right = expr["right"]

    # If right side is a constant number

    if right.isdigit():

        constants[expr["left"]] = right

    else:

        # Replace variables in right side with constants if available

        parts = right.split()

        new\_right = []

        for part in parts:

            new\_right.append(constants.get(part, part))

        expr["right"] = ' '.join(new\_right)

    optimized\_exprs.append(expr)

# After Constant Propagation

print("\nAfter Constant Propagation:")

for expr in optimized\_exprs:

    print(f"{expr['left']} = {expr['right']}")

# Step 2: Dead Code Elimination (optional to match your structure)

used\_vars = {expr["left"] for expr in optimized\_exprs if any(expr["left"] in other["right"] for other in optimized\_exprs)}

final\_exprs = [expr for i, expr in enumerate(optimized\_exprs) if expr["left"] in used\_vars or i == len(optimized\_exprs) - 1]

print("\nAfter Dead Code Elimination:")

for expr in final\_exprs:

    print(f"{expr['left']} = {expr['right']}")

# Step 3: Final Duplicate Removal

seen = set()

optimized\_final = []

for expr in final\_exprs:

    key = (expr["left"], expr["right"])

    if key not in seen:

        seen.add(key)

        optimized\_final.append(expr)

# Optimized Code

print("\nOptimized Code:")

for expr in optimized\_final:

    print(f"{expr['left']} = {expr['right']}")

optimised using copy prop.py :

# Input

n = int(input("Enter the number of expressions: "))

print("Enter the expressions in the format 'left = right':")

expressions = [{"left": input("Left: ").strip(), "right": input("Right: ").strip()} for \_ in range(n)]

# Intermediate Code

print("\nIntermediate Code:")

for expr in expressions:

    print(f"{expr['left']} = {expr['right']}")

# Step 1: Dead Code Elimination

used\_vars = {expr["left"] for expr in expressions if any(expr["left"] in other["right"] for other in expressions)}

optimized\_exprs = [expr for i, expr in enumerate(expressions) if expr["left"] in used\_vars or i == len(expressions) - 1]

print("\nAfter Dead Code Elimination:")

for expr in optimized\_exprs:

    print(f"{expr['left']} = {expr['right']}")

# Step 2: Common Subexpression Elimination

subexp\_map = {}

final\_exprs = []

for expr in optimized\_exprs:

    if expr["right"] in subexp\_map:

        replacement = subexp\_map[expr["right"]]

        for e in optimized\_exprs:

            e["right"] = e["right"].replace(expr["left"], replacement)

    else:

        subexp\_map[expr["right"]] = expr["left"]

        final\_exprs.append(expr)

print("\nAfter Eliminating Common Expressions:")

for expr in final\_exprs:

    print(f"{expr['left']} = {expr['right']}")

# Step 3: Final Optimization (Duplicate Removal)

seen = set()

optimized\_final = []

for expr in final\_exprs:

    key = (expr["left"], expr["right"])

    if key not in seen:

        seen.add(key)

        optimized\_final.append(expr)

# Optimized Code

print("\nOptimized Code:")

for expr in optimized\_final:

    print(f"{expr['left']} = {expr['right']}")

**ST\_table.py :**

# Simplified ST (Simple Table) Two-Pass Assembler

def st\_assembler(lines):

    symbol\_table = {}

    location\_counter = 0

    # PASS 1

    for line in lines:

        parts = line.strip().split()

        if not parts: continue

        label, opcode, operand = (parts + ["", ""])[:3]

        if opcode == "START":

            location\_counter = int(operand)

        if label:

            symbol\_table[label] = location\_counter

        location\_counter += 1

    # PASS 2

    print("\nMachine Code:")

    loc = 0

    for line in lines:

        parts = line.strip().split()

        if not parts: continue

        label, opcode, operand = (parts + ["", ""])[:3]

        address = symbol\_table.get(operand, operand)

        print(f"{loc:04d}: {opcode} {address}")

        loc += 1

    print("\nSymbol Table:", symbol\_table)

# Sample

st\_assembler([

    "START 0",

    "LOAD A",

    "ADD B",

    "STORE C",

    "A DC 5",

    "B DC 3",

    "C DS 1",

    "END"

])

**Input.txt :**

MACRO

test &p &q

move &p &q

MEND

begin

test x y

stop