Spcc prac

1.Write a program to implement two- pass assembler. [Generate the symbol table, literal table from ALP code]

code: # Two-Pass Assembler with Symbol Table and Literal Table

```
MOT = { "L": "01", "A": "02", "ST": "03", "MOVER": "04", "MOVEM": "05", "ADD": "06", "SUB": "07" }

POT = ["START", "END", "LTORG", "ORIGIN", "EQU", "DS", "DC"]


alp_code = [

  "START 100",

  "LOOP MOVER AREG, ='5'",

  "    ADD AREG, VALUE",

  "    MOVEM AREG, RESULT",

  "    LTORG",

  "VALUE DC 10",

  "RESULT DS 1",

  "    END"

]


symbol_table = {}

literal_table = []

intermediate_code = []


location_counter = 0

literal_pool = []
```

```
def add_literal(lit):
    if lit not in [x['literal'] for x in literal_table]:
        literal_table.append({'literal': lit, 'address': None})


# ----------- PASS 1 ------------
print("\n=== PASS 1 ===")
for line in alp_code:
    tokens = line.strip().split()

    if not tokens:
        continue

    # Handle START
    if tokens[0] == "START":
        location_counter = int(tokens[1])
        intermediate_code.append((location_counter, "START", ""))
        continue

    # Handle END or LTORG
    if tokens[0] in ["END", "LTORG"]:
        # Assign addresses to literals
        for lit in literal_table:
            if lit['address'] is None:
                lit['address'] = location_counter
                location_counter += 1
        intermediate_code.append((location_counter, tokens[0], ""))
```

```
    continue

label = ""
opcode = ""
operand = ""

# Parse line
if len(tokens) == 3:
    label, opcode, operand = tokens
elif len(tokens) == 2:
    opcode, operand = tokens
else:
    opcode = tokens[0]

if label:
    symbol_table[label] = location_counter

if operand.startswith("='"):
    add_literal(operand)

intermediate_code.append((location_counter, opcode, operand))

# Increment LC if not a declarative
if opcode not in ["DS", "DC"]:
    location_counter += 1
elif opcode == "DS":
```

```
        location_counter += int(operand)
    elif opcode == "DC":
        location_counter += 1



# ---------- PASS 2 (Simulated Code Gen) --------------
print("\n=== PASS 2 ===")
machine_code = []
for loc, opcode, operand in intermediate_code:
    if opcode in MOT:
        code = MOT[opcode]
        machine_code.append((loc, code, operand))
    elif opcode in POT or opcode == "START":
        machine_code.append((loc, opcode, operand))



# ---------- SYMBOL TABLE ----------
print("\n--- SYMBOL TABLE ---")
for symbol, addr in symbol_table.items():
    print(f"{symbol} : {addr}")

# ---------- LITERAL TABLE ----------
print("\n--- LITERAL TABLE ---")
for lit in literal_table:
    print(f"{lit['literal']} : {lit['address']}")
```

---------- INTERMEDIATE CODE ----------

print("\n--- INTERMEDIATE CODE ---")

for line in intermediate_code:

    print(line)

---------- FINAL MACHINE CODE (Simulated) ----------

print("\n--- FINAL MACHINE CODE ---")

for line in machine_code:

    print(line)

```
--- PASS 2 ---

--- SYMBOL TABLE ---
ADD : 101
MOVEM : 102
VALUE : 103
RESULT : 104

--- LITERAL TABLE ---

--- INTERMEDIATE CODE ---
(100, 'START', '')
(100, 'LOOP', '')
(101, 'AREG,', 'VALUE')
(102, 'AREG,', 'RESULT')
(103, 'LTORG', '')
(103, 'DC', '10')
(104, 'DS', '1')
(105, 'END', '')

--- FINAL MACHINE CODE ---
(100, 'START', '')
(103, 'LTORG', '')
(103, 'DC', '10')
(104, 'DS', '1')
(105, 'END', '')
```

2.Write a program to implement two –pass assembler. [Generate the base table, location counter (LC)]

Code: # Sample opcode table

MOT = {

    "L": "01", "A": "02", "ST": "03", "MOVER": "04", "MOVEM": "05", "ADD": "06", "SUB": "07"

}

```
POT = ["START", "END", "LTORG", "ORIGIN", "EQU"]


# Sample ALP code
alp_code = [
    "START 100",
    "LOOP  MOVER AREG, ONE",
    "      ADD   AREG, TWO",
    "      MOVEM AREG, THREE",
    "      L     BREG, FOUR",
    "      A     BREG, FIVE",
    "      ST    BREG, RESULT",
    "      END"
]


symbol_table = {}
base_table = {}
intermediate_code = []


location_counter = 0
start_address = 0
reg_map = {"AREG": 1, "BREG": 2, "CREG": 3, "DREG": 4}




# ---------- PASS 1 ----------
print("\n=== PASS 1 ===")
```

```
for line in alp_code:
    tokens = line.strip().split()

    if tokens[0] == "START":
        start_address = int(tokens[1])
        location_counter = start_address
        intermediate_code.append((location_counter, "START", ""))
        continue

    if tokens[0] == "END":
        intermediate_code.append((location_counter, "END", ""))
        break

    if len(tokens) == 3:
        label, opcode, operand = tokens
        symbol_table[label] = location_counter
    else:
        label = None
        opcode = tokens[0]
        operand = tokens[1]

    intermediate_code.append((location_counter, opcode, operand))
    location_counter += 1


# ---------- Generate Base Table ----------
```

```
print("\n=== BASE TABLE ===")

for reg, code in reg_map.items():

    base_table[reg] = f"Base Register {code} Initialized"


for reg, info in base_table.items():

    print(f"{reg} -> {info}")



# ---------- PASS 2 ----------

print("\n=== PASS 2 ===")

machine_code = []

for loc, opcode, operand in intermediate_code:

    if opcode in MOT:

        op_code = MOT[opcode]

        reg, sym = operand.split(",")

        address = symbol_table.get(sym.strip(), 0)

        machine_code.append((loc, op_code, reg_map[reg.strip()], address))

    elif opcode in POT:

        machine_code.append((loc, opcode, operand))



# ---------- SYMBOL TABLE ----------

print("\n=== SYMBOL TABLE ===")

for symbol, addr in symbol_table.items():

    print(f"{symbol} : {addr}")
```

```
# ---------- INTERMEDIATE CODE ----------

print("\n=== INTERMEDIATE CODE ===")

for entry in intermediate_code:

    print(entry)



# ---------- FINAL MACHINE CODE ----------

print("\n=== FINAL MACHINE CODE ===")

for code in machine_code:

    print(code)
```

```
=== PASS 1 ===

=== BASE TABLE ===
AREG -> Base Register 1 Initialized
BREG -> Base Register 2 Initialized
CREG -> Base Register 3 Initialized
DREG -> Base Register 4 Initialized

=== PASS 2 ===

=== SYMBOL TABLE ===
ADD : 101
MOVEM : 102
L : 103
A : 104
ST : 105

=== INTERMEDIATE CODE ===
(100, 'START', '')
(100, 'LOOP', 'MOVER')
(101, 'AREG,', 'TWO')
(102, 'AREG,', 'THREE')
(103, 'BREG,', 'FOUR')
(104, 'BREG,', 'FIVE')
(105, 'BREG,', 'RESULT')
(106, 'END', '')

=== FINAL MACHINE CODE ===
(100, 'START', '')
(106, 'END', '')
```

3.

Write a program to implement 2-pass assembler. [Display MOT and POT contents from ALP code]

code: # Predefined Machine and Pseudo Opcode Tables

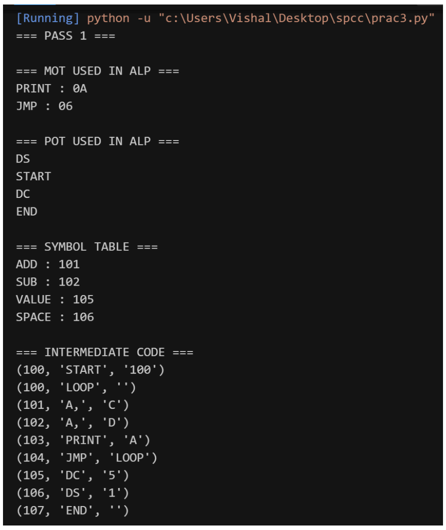
```
MOT = {

    "MOV": "01", "ADD": "02", "SUB": "03", "MUL": "04", "DIV": "05",

    "JMP": "06", "CMP": "07", "LOAD": "08", "STORE": "09", "PRINT": "0A"
```

```
}


POT = ["START", "END", "DC", "DS", "ORG", "EQU"]


# Sample ALP code (as a list of strings)
alp_code = [
    "START 100",
    "LOOP MOV A, B",
    "    ADD A, C",
    "    SUB A, D",
    "    PRINT A",
    "    JMP LOOP",
    "VALUE DC 5",
    "SPACE DS 1",
    "    END"
]


location_counter = 0
symbol_table = {}
intermediate_code = []


used_mot = set()
used_pot = set()


# -------- PASS 1 --------
print("=== PASS 1 ===")
```

```
for line in alp_code:
    tokens = line.strip().split()

    # Handle START
    if tokens[0] == "START":
        location_counter = int(tokens[1])
        used_pot.add("START")
        intermediate_code.append((location_counter, "START", tokens[1]))
        continue

    if tokens[0] == "END":
        used_pot.add("END")
        intermediate_code.append((location_counter, "END", ""))
        break

    label = ""
    opcode = ""
    operand = ""

    if len(tokens) == 3:
        label, opcode, operand = tokens
        symbol_table[label] = location_counter
    elif len(tokens) == 2:
        opcode, operand = tokens
    else:
        opcode = tokens[0]
```

```
    # Track MOT/POT usage
    if opcode in MOT:
        used_mot.add(opcode)
    elif opcode in POT:
        used_pot.add(opcode)

    # Update location counter
    if opcode == "DS":
        location_counter += int(operand)
    elif opcode == "DC":
        location_counter += 1
    else:
        location_counter += 1

    intermediate_code.append((location_counter - 1, opcode, operand))

# -------- DISPLAY OUTPUTS --------

print("\n=== MOT USED IN ALP ===")
for op in used_mot:
    print(f"{op} : {MOT[op]}")

print("\n=== POT USED IN ALP ===")
for op in used_pot:
    print(op)
```

```
print("\n=== SYMBOL TABLE ===")

for sym, addr in symbol_table.items():

    print(f"{sym} : {addr}")



print("\n=== INTERMEDIATE CODE ===")

for line in intermediate_code:

    print(line)
```



```
[Running] python -u "c:\Users\Vishal\Desktop\spcc\prac3.py"
=== PASS 1 ===

=== MOT USED IN ALP ===
PRINT : 0A
JMP : 06

=== POT USED IN ALP ===
DS
START
DC
END

=== SYMBOL TABLE ===
ADD : 101
SUB : 102
VALUE : 105
SPACE : 106

=== INTERMEDIATE CODE ===
(100, 'START', '100')
(100, 'LOOP', '')
(101, 'A,', 'C')
(102, 'A,', 'D')
(103, 'PRINT', 'A')
(104, 'JMP', 'LOOP')
(105, 'DC', '5')
(106, 'DS', '1')
(107, 'END', '')
```

4.Write a program to implement 2 pass macro processor. [Display macro name table, macro definition table, Argument List Array

code:

```
assembly_code = [

    "MACRO",

    "INCR &ARG1,&ARG2",

    "LOAD &ARG1",

    "ADD &ARG2",

    "STORE &ARG1",

    "MEND",

    "START",
```

```
    "INCR A,B",
    "END"
]

mnt = []     # Macro Name Table: list of dicts with name and MDT index
mdt = []     # Macro Definition Table: list of strings
ala_table = {}  # ALA: macro name -> list of parameters
expanded_code = []

# ---------- PASS 1: BUILD MNT, MDT, ALA ----------
i = 0
while i < len(assembly_code):
    line = assembly_code[i].strip()
    tokens = line.split()

    if tokens[0] == "MACRO":
        i += 1
        header = assembly_code[i].strip().split()
        macro_name = header[0]
        parameters = header[1].split(",") if len(header) > 1 else []
        ala_table[macro_name] = parameters
        mnt.append({"name": macro_name, "mdt_index": len(mdt)})
        i += 1

        while assembly_code[i].strip() != "MEND":
            def_line = assembly_code[i]
```

```
            for idx, param in enumerate(parameters):
                def_line = def_line.replace(param, f"#{idx+1}")
            mdt.append(def_line)
            i += 1

        mdt.append("MEND")  # Add MEND to MDT
    else:
        expanded_code.append(line)
    i += 1

# ---------- PASS 2: MACRO EXPANSION ----------
final_code = []
for line in expanded_code:
    tokens = line.strip().split()
    if not tokens:
        continue

    macro_call = tokens[0]
    args = tokens[1].split(",") if len(tokens) > 1 else []

    macro_found = next((m for m in mnt if m["name"] == macro_call), None)
    if macro_found:
        mdt_index = macro_found["mdt_index"]
        formal_params = ala_table[macro_call]
        ala_instance = {f"#{idx+1}": args[idx] for idx in range(len(args))}
```

```
        while mdt[mdt_index] != "MEND":
            expanded_line = mdt[mdt_index]
            for k, v in ala_instance.items():
                expanded_line = expanded_line.replace(k, v)
            final_code.append(expanded_line)
            mdt_index += 1
    else:
        final_code.append(line)


# ---------- DISPLAY OUTPUTS ----------
print("\n=== Macro Name Table (MNT) ===")
for idx, entry in enumerate(mnt):
    print(f"{idx}\t{entry['name']}\tMDT Index: {entry['mdt_index']}")


print("\n=== Macro Definition Table (MDT) ===")
for idx, line in enumerate(mdt):
    print(f"{idx}\t{line}")


print("\n=== Argument List Array (ALA) ===")
for macro, params in ala_table.items():
    for idx, param in enumerate(params):
        print(f"{macro} -> {param} -> #{idx+1}")


print("\n=== Final Expanded Code ===")
for line in final_code:
    print(line)
```

```
=== Macro Name Table (MNT) ===
0   INCR    MDT Index: 0

=== Macro Definition Table (MDT) ===
0   LOAD #1
1   ADD #2
2   STORE #1
3   MEND

=== Argument List Array (ALA) ===
INCR -> &ARG1 -> #1
INCR -> &ARG2 -> #2

=== Final Expanded Code ===
START
LOAD A
ADD B
STORE A
END
```

5.Write a program to implement 2 pass macro processor. [Display macro expansion, predefine MDT and MNT tables]

code:

```
# Predefined Macro Name Table (MNT)

mnt = {

   "INCR": 0

}


# Predefined Macro Definition Table (MDT)

mdt = [

   "LOAD #1",

   "ADD #2",

   "STORE #1",

   "MEND"

]


# Assembly program using the macro
```

```
alp_code = [
    "START",
    "INCR A,B",
    "MOV C,D",
    "INCR X,Y",
    "END"
]

# Store expanded assembly code
expanded_code = []

# ----- PASS 2: Macro Expansion -----
for line in alp_code:
    tokens = line.strip().split()
    if not tokens:
        continue

    macro_name = tokens[0]
    args = tokens[1].split(",") if len(tokens) > 1 else []

    if macro_name in mnt:
        mdt_index = mnt[macro_name]
        # Create local ALA for this invocation
        ala = {f"#{i+1}": args[i] for i in range(len(args))}

        while mdt[mdt_index] != "MEND":
```

```
        temp_line = mdt[mdt_index]
        for key, value in ala.items():
            temp_line = temp_line.replace(key, value)
        expanded_code.append(temp_line)
        mdt_index += 1
    else:
        expanded_code.append(line)


# ----- Output Section -----
print("=== Macro Name Table (MNT) ===")
for name, index in mnt.items():
    print(f"{name} -> MDT Index: {index}")


print("\n=== Macro Definition Table (MDT) ===")
for i, line in enumerate(mdt):
    print(f"{i} : {line}")


print("\n=== Macro Expansion Output ===")
for line in expanded_code:
    print(line)
```

```
INCR -> MDT Index: 0

=== Macro Definition Table (MDT) ===
0 : LOAD #1
1 : ADD #2
2 : STORE #1
3 : MEND

=== Macro Expansion Output ===
START
LOAD A
ADD B
STORE A
MOV C,D
LOAD X
ADD Y
STORE X
END
```

6.Write a program to implement 2 pass macro processor. [Identify macros and perform macro expansion]

code:

```
# Assembly Language Program with Macro Definition

alp_code = [

    "MACRO",

    "INCR &A,&B",

    "LOAD &A",

    "ADD &B",

    "STORE &A",

    "MEND",

    "START",

    "INCR X,Y",

    "MOV A,B",

    "INCR P,Q",

    "END"

]
```

```
# Tables
mnt = {}        # Macro Name Table: name -> MDT index
mdt = []        # Macro Definition Table
ala_map = {}     # Argument List Array: macro -> [&A, &B]

expanded_code = []  # Final code with macros expanded

# ---------- PASS 1: Identify and Store Macros ----------
i = 0
while i < len(alp_code):
    line = alp_code[i].strip()
    tokens = line.split()

    if tokens[0] == "MACRO":
        i += 1
        header = alp_code[i].strip().split()
        macro_name = header[0]
        args = header[1].split(",")
        mnt[macro_name] = len(mdt)
        ala_map[macro_name] = args
        i += 1

        # Process macro body
        while alp_code[i].strip() != "MEND":
            macro_line = alp_code[i]
```

```
        for idx, arg in enumerate(args):
            macro_line = macro_line.replace(arg, f"#{idx+1}")
        mdt.append(macro_line)
        i += 1

    mdt.append("MEND")  # Add MEND
  else:
    expanded_code.append(line)  # Temporarily store non-macro lines
  i += 1

# ---------- PASS 2: Expand Macros ----------
final_code = []
for line in expanded_code:
  tokens = line.strip().split()
  if not tokens:
    continue

  macro_call = tokens[0]
  if macro_call in mnt:
    # Get actual arguments and setup ALA
    actual_args = tokens[1].split(",") if len(tokens) > 1 else []
    ala_instance = {f"#{i+1}": actual_args[i] for i in range(len(actual_args))}

    # Get MDT index for macro body
    mdt_index = mnt[macro_call]
    while mdt[mdt_index] != "MEND":
```

```
            expanded_line = mdt[mdt_index]
            for key, val in ala_instance.items():
                expanded_line = expanded_line.replace(key, val)
            final_code.append(expanded_line)
            mdt_index += 1
    else:
        final_code.append(line)

# ---------- OUTPUT ----------
print("=== Macro Name Table (MNT) ===")
for name, idx in mnt.items():
    print(f"{name} -> MDT Index {idx}")

print("\n=== Macro Definition Table (MDT) ===")
for idx, line in enumerate(mdt):
    print(f"{idx}: {line}")

print("\n=== Argument List Array (ALA) ===")
for name, args in ala_map.items():
    for idx, arg in enumerate(args):
        print(f"{name} : {arg} -> #{idx+1}")

print("\n=== Final Code after Macro Expansion ===")
for line in final_code:
    print(line)
```

```
=== Macro Name Table (MNT) ===
INCR -> MDT Index 0

=== Macro Definition Table (MDT) ===
0: LOAD #1
1: ADD #2
2: STORE #1
3: MEND

=== Argument List Array (ALA) ===
INCR : &A -> #1
INCR : &B -> #2

=== Final Code after Macro Expansion ===
START
LOAD X
ADD Y
STORE X
MOV A,B
LOAD P
ADD Q
STORE P
END

[Done] exited with code=0 in 0.182 seconds
```

7.Write a program to find the first set of given grammar.

code:

```
from collections import defaultdict


# Sample grammar productions as a list of strings

productions = [

    "E -> T E'",

    "E' -> + T E' | e",

    "T -> F T'",

    "T' -> * F T' | e",

    "F -> ( E ) | id"

]


# Step 1: Parse grammar into a dictionary

grammar = defaultdict(list)


for prod in productions:
```

```
    head, body = prod.split("->")
    head = head.strip()
    alternatives = body.strip().split("|")
    for alt in alternatives:
        grammar[head].append(alt.strip())

# Step 2: Define FIRST set storage
first = defaultdict(set)

# Step 3: Utility to check if a symbol is terminal
def is_terminal(symbol):
    return not symbol.isupper() and symbol != 'e'

# Step 4: Compute FIRST set for a symbol
def compute_first(symbol):
    if is_terminal(symbol) or symbol == 'e':
        return {symbol}

    if first[symbol]:  # Already computed
        return first[symbol]

    for production in grammar[symbol]:
        symbols = production.split()

        for sym in symbols:
            sym_first = compute_first(sym)
            first[symbol].update(sym_first - {'e'})

            if 'e' not in sym_first:
                break
```

```
        else:
            # All symbols had e, so include e in the FIRST set
            first[symbol].add('e')

    return first[symbol]

# Step 5: Compute FIRST sets for all non-terminals
for non_terminal in grammar:
    compute_first(non_terminal)

# Step 6: Display the FIRST sets
print("=== FIRST Sets ===")
for non_terminal, first_set in first.items():
    formatted = ', '.join(sorted(first_set))
    print(f"FIRST({non_terminal}) = {{ {formatted} }}")
```

8.Write a program to find the follow set of given grammar.

code:

```
from collections import defaultdict

# Sample grammar productions
productions = [

    "E -> T E'",
    "E' -> + T E' | e",
    "T -> F T'",
    "T' -> * F T' | e",
    "F -> ( E ) | id"
]
```

```
# Step 1: Parse the grammar
grammar = defaultdict(list)
non_terminals = set()
terminals = set()

for prod in productions:
    head, body = prod.split("->")
    head = head.strip()
    non_terminals.add(head)
    alternatives = body.strip().split("|")
    for alt in alternatives:
        grammar[head].append(alt.strip())

# Step 2: Helper functions
def is_terminal(symbol):
    return not symbol.isupper() and symbol != 'e'

first = defaultdict(set)
follow = defaultdict(set)

# Step 3: Compute FIRST sets
def compute_first(symbol):
    if is_terminal(symbol):
        return {symbol}
    if symbol == 'e':
        return {'e'}
    if first[symbol]:  # already computed
        return first[symbol]
```

```
    for production in grammar[symbol]:
        symbols = production.split()
        for sym in symbols:
            sym_first = compute_first(sym)
            first[symbol].update(sym_first - {'e'})
            if 'e' not in sym_first:
                break
        else:
            first[symbol].add('e')
    return first[symbol]

# Step 4: Compute FIRST of a string of symbols
def compute_first_of_string(symbols):
    result = set()
    for sym in symbols:
        sym_first = compute_first(sym)
        result.update(sym_first - {'e'})
        if 'e' not in sym_first:
            break
    else:
        result.add('e')
    return result

# Step 5: Compute FOLLOW sets
def compute_follow():
    start_symbol = list(grammar.keys())[0]
    follow[start_symbol].add('$')  # $ denotes end of input
```

```
    changed = True
    while changed:
        changed = False
        for head in grammar:
            for production in grammar[head]:
                symbols = production.split()
                for i, symbol in enumerate(symbols):
                    if symbol in non_terminals:
                        next_symbols = symbols[i + 1:]
                        next_first = compute_first_of_string(next_symbols)
                        before = len(follow[symbol])
                        follow[symbol].update(next_first - {'e'})
                        if 'e' in next_first or not next_symbols:
                            follow[symbol].update(follow[head])
                        if len(follow[symbol]) > before:
                            changed = True

# Step 6: Compute all FIRST and FOLLOW sets
for non_terminal in grammar:
    compute_first(non_terminal)
compute_follow()

# Step 7: Display the FOLLOW sets
print("=== FOLLOW Sets ===")
for non_terminal, follow_set in follow.items():
    formatted = ', '.join(sorted(follow_set))
    print(f"FOLLOW({non_terminal}) = {{ {formatted} }}")
```

9.Write a program to design handwritten lexical analyzer using programming language. (Display keyword, identifier, symbols]

```
import re


# List of keywords in a programming language

keywords = {'if', 'else', 'while', 'for', 'return', 'int', 'float', 'char'}


# Regular expressions for different token types

regex_keywords = '|'.join(keywords)  # Keywords regex

regex_identifier = r'[a-zA-Z_][a-zA-Z0-9_]*'  # Identifiers (starts with letter or underscore)

regex_symbol = r'[\+\-\*/\=\(\)\{\}\[\];,]'  # Symbols like operators, braces, semicolons


# Combine the regex patterns

combined_regex = f'({regex_keywords})|({regex_identifier})|({regex_symbol})'


# Function to perform lexical analysis

def lexical_analyzer(code):

    tokens = []

    # Match all tokens using the regex

    for match in re.finditer(combined_regex, code):

        if match.group(1):  # Keyword

            tokens.append(('Keyword', match.group(1)))

        elif match.group(2):  # Identifier

            tokens.append(('Identifier', match.group(2)))

        elif match.group(3):  # Symbol

            tokens.append(('Symbol', match.group(3)))

    return tokens
```

```
# Sample input code (as a string)

code = '''

int main() {

   int a = 10;

   float b = 20.5;

   if (a > b) {

       return a;

   }

   while (a < b) {

       a = a + 1;

   }

}

'''


# Perform lexical analysis

tokens = lexical_analyzer(code)


# Display tokens

print("Tokens in the code:")

for token in tokens:

   print(f"Type: {token[0]}, Value: {token[1]}")
```

10.Write a program to design handwritten lexical analyzer using programming language. (Display numbers, identifier, preprocessor directives]

```
import re


# Regular expressions for different token types

regex_number = r'\b\d+(\.\d+)?\b'  # Matches integers and floating-point numbers
```

```
regex_identifier = r'\b[a-zA-Z_][a-zA-Z0-9_]*\b'  # Identifiers (starts with letter or underscore)
regex_preprocessor = r'^\s*#\s*(\w+)'  # Matches preprocessor directives starting with #

# Function to perform lexical analysis
def lexical_analyzer(code):
    tokens = []

    # Preprocess the code to handle different token types
    lines = code.splitlines()  # Split code into lines for easy preprocessor directive handling

    for line in lines:
        # Match Preprocessor Directives
        if re.match(regex_preprocessor, line):
            directive = re.match(regex_preprocessor, line).group(1)
            tokens.append(('Preprocessor Directive', directive))

        # Match Numbers (integers and floats)
        for match in re.finditer(regex_number, line):
            tokens.append(('Number', match.group()))

        # Match Identifiers
        for match in re.finditer(regex_identifier, line):
            tokens.append(('Identifier', match.group()))

    return tokens

# Sample input code (as a string)
```

```
code = '''
#include <stdio.h>
#define MAX 100
int main() {
    int a = 10;
    float b = 20.5;
    a = a + 2;
    if (a > b) {
        return a;
    }
}
'''


# Perform lexical analysis
tokens = lexical_analyzer(code)


# Display tokens
print("Tokens in the code:")
for token in tokens:
    print(f"Type: {token[0]}, Value: {token[1]}")
```

11.Write a program to implement following code optimization techniques. 1) Algebraic simplification 2) Common sub expression elimination.

```
import re


# Function for Algebraic Simplification
def algebraic_simplification(expression):
    # Simplify expressions: a * 1 = a, a + 0 = a, etc.
    expression = re.sub(r'(\w+) *\* 1', r'\1', expression)  # a * 1 = a
```

```
    expression = re.sub(r'(\w+) *\+ 0', r'\1', expression)  # a + 0 = a
    expression = re.sub(r'(\w+) *- \1', '0', expression)  # a + (-a) = 0
    expression = re.sub(r'(\w+) *\* 0', '0', expression)  # a * 0 = 0
    return expression


# Function for Common Subexpression Elimination
def common_subexpression_elimination(statements):
    # Track already encountered expressions
    seen_expressions = {}
    optimized_statements = []
    temp_var_count = 1  # Temporary variable counter for CSE


    for statement in statements:
        # Identify expressions in the statement
        expr = statement.split('=')[1].strip()


        # Check if the expression has already been seen
        if expr in seen_expressions:
            # Replace with previously computed expression
            optimized_statements.append(f"{statement.split('=')[0].strip()} = {seen_expressions[expr]}")
        else:
            # If not seen, store the expression
            temp_var = f"temp{temp_var_count}"
            seen_expressions[expr] = temp_var
            optimized_statements.append(f"{temp_var} = {expr}")
            temp_var_count += 1
```

```
    return optimized_statements


# Sample input code (in the form of arithmetic expressions)
code = [
    "x = a * b + a * b",
    "y = a * c + d",
    "z = a * b + a * b",
    "w = x + 0",
    "v = y * 1"
]


# Apply Algebraic Simplification
simplified_code = [algebraic_simplification(statement) for statement in code]


# Apply Common Subexpression Elimination
optimized_code = common_subexpression_elimination(simplified_code)


# Display Optimized Code
print("Original Code:")
for line in code:
    print(line)


print("\nSimplified Code:")
for line in simplified_code:
    print(line)


print("\nOptimized Code (After CSE):")
```

```
for line in optimized_code:
    print(line)
```

12.Write a program to implement following code optimization techniques. 1)Dead Code Elimination 2) Constant Propagation

```
import re


# Function for Dead Code Elimination
def dead_code_elimination(statements):
    used_vars = set()
    assigned_vars = set()


    # First pass: Identify all used variables
    for statement in statements:
        parts = statement.split("=")
        if len(parts) > 1:
            # Track assigned variables
            assigned_vars.add(parts[0].strip())
            # Track used variables (ignore assignments)
            used_vars.update(re.findall(r'\b[a-zA-Z_][a-zA-Z0-9_]*\b', parts[1]))


    # Second pass: Remove statements where assigned variable is not used
    optimized_statements = []
    for statement in statements:
        parts = statement.split("=")
        if len(parts) > 1 and parts[0].strip() in used_vars:
            optimized_statements.append(statement)
        elif len(parts) == 1:  # In case of expression with no assignment
            optimized_statements.append(statement)
```

```
    return optimized_statements


# Function for Constant Propagation
def constant_propagation(statements):
    const_values = {}

    # First pass: Collect constant assignments
    for statement in statements:
        parts = statement.split("=")
        if len(parts) > 1:
            var, expr = parts[0].strip(), parts[1].strip()
            # If expression is constant, store it
            if expr.isdigit() or (expr[0] == '-' and expr[1:].isdigit()):
                const_values[var] = int(expr)

    # Second pass: Replace variables with constant values
    optimized_statements = []
    for statement in statements:
        parts = statement.split("=")
        if len(parts) > 1:
            var, expr = parts[0].strip(), parts[1].strip()
            # Replace variables in expression with their constant values
            for key, value in const_values.items():
                expr = expr.replace(key, str(value))
            optimized_statements.append(f"{var} = {expr}")
        else:
```

```
            optimized_statements.append(statement)


    return optimized_statements


# Sample input code (in the form of assignments and expressions)

code = [

    "x = 5",

    "y = 10",

    "z = x + y",

    "a = 0",

    "b = a + 1",

    "c = b + 2",

    "d = z + 3",

    "x = x + 5",

    "z = 100",

    "e = z + 50"

]


# Apply Constant Propagation

code_with_constant_propagation = constant_propagation(code)


# Apply Dead Code Elimination

optimized_code = dead_code_elimination(code_with_constant_propagation)


# Display the original code

print("Original Code:")

for line in code:
```

```
    print(line)


# Display the code after constant propagation
print("\nCode After Constant Propagation:")
for line in code_with_constant_propagation:
    print(line)


# Display the optimized code after Dead Code Elimination
print("\nOptimized Code After Dead Code Elimination:")
for line in optimized_code:
    print(line)
```

13.Write a program to implement Intermediate Code Generator using 3-Address code using triples.

```
import re


class IntermediateCodeGenerator:
    def __init__(self):
        self.temp_count = 1
        self.triples = []


    def generate_temp(self):
        temp = f't{self.temp_count}'
        self.temp_count += 1
        return temp


    def infix_to_postfix(self, tokens):
        precedence = {'=': 1, '+': 2, '-': 2, '*': 3, '/': 3}
        output = []
```

```
        stack = []

        for token in tokens:
            if token.isalnum():
                output.append(token)
            elif token in precedence:
                while stack 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()  # remove '('

        while stack:
            output.append(stack.pop())

        return output

    def generate_triples(self, expression):
        tokens = re.findall(r'[a-zA-Z_][a-zA-Z0-9_]*|\d+|[+\-*/=()]', expression)
        postfix = self.infix_to_postfix(tokens)

        operands_stack = []
```

```
        for token in postfix:
            if token.isalnum():
                operands_stack.append(token)
            elif token in '+-*/':
                operand2 = operands_stack.pop()
                operand1 = operands_stack.pop()
                temp = self.generate_temp()
                self.triples.append((token, operand1, operand2))
                operands_stack.append(temp)
            elif token == '=':
                operand2 = operands_stack.pop()
                operand1 = operands_stack.pop()
                self.triples.append(('=', operand1, operand2))
                operands_stack.append(operand1)

        return self.triples

    def display_triples(self):
        print("Intermediate Code (using Triples):")
        for idx, triple in enumerate(self.triples):
            print(f"({idx + 1}) {triple[0]} {triple[1]} {triple[2]}")

# Example usage
if __name__ == "__main__":
    expression = "a = b + c * d"
    code_generator = IntermediateCodeGenerator()
    code_generator.generate_triples(expression)
```

```
    code_generator.display_triples()
```

14.Write a program to implement Intermediate Code Generator using 3-Address code using quadruples.

```
import re


class QuadrupleGenerator:
    def __init__(self):
        self.temp_count = 1
        self.quadruples = []


    def new_temp(self):
        temp = f"t{self.temp_count}"
        self.temp_count += 1
        return temp


    def precedence(self, op):
        if op == '+' or op == '-':
            return 1
        if op == '*' or op == '/':
            return 2
        return 0


    def generate(self, expression):
        # Split assignment: a = b + c * d
        var, expr = expression.split('=')
        var = var.strip()
        expr = expr.strip()
```

```
# Convert to tokens
tokens = re.findall(r'[a-zA-Z_]\w*|\d+|[+\-*/()]', expr)

# Convert to postfix using Shunting Yard Algorithm
output = []
stack = []

for token in tokens:
    if token.isalnum():
        output.append(token)
    elif token in '+-*/':
        while stack and self.precedence(stack[-1]) >= self.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())

# Generate Quadruples from postfix
eval_stack = []
```

```
        for token in output:
            if token not in '+-*/':
                eval_stack.append(token)
            else:
                arg2 = eval_stack.pop()
                arg1 = eval_stack.pop()
                result = self.new_temp()
                self.quadruples.append((token, arg1, arg2, result))
                eval_stack.append(result)

        # Final assignment to the variable
        final_result = eval_stack.pop()
        self.quadruples.append(('=', final_result, '-', var))

    def display(self):
        print("Generated 3-Address Code (Quadruples):")
        print(f"{'Op':^8} {'Arg1':^8} {'Arg2':^8} {'Result':^8}")
        print("-" * 36)
        for quad in self.quadruples:
            print(f"{quad[0]:^8} {quad[1]:^8} {quad[2]:^8} {quad[3]:^8}")


# Example Usage
if __name__ == "__main__":
    expression = "a = b + c * d"
    generator = QuadrupleGenerator()
    generator.generate(expression)
```

```
    generator.display()
```

15.Write a program to implement automated lexical analyzer using LEX tool

```
%{
#include <stdio.h>
#include <string.h>

int isKeyword(char *word) {
    char *keywords[] = { "int", "float", "char", "if", "else", "while", "for", "return", "void" };
    for (int i = 0; i < 9; i++) {
        if (strcmp(word, keywords[i]) == 0) {
            return 1;
        }
    }
    return 0;
}
%}

%%

[ \t\n]+              ; // Ignore whitespace

"=="|"!="|"<="|">="|"="|"+"|"-"|"*"|"/"    { printf("Operator: %s\n", yytext); }

"(" | ")" | ";" | "," | "{" | "}"        { printf("Delimiter: %s\n", yytext); }

[0-9]+(\.[0-9]+)?      { printf("Number: %s\n", yytext); }
```

```
[a-zA-Z_][a-zA-Z0-9_]* {
                    if (isKeyword(yytext))
                        printf("Keyword: %s\n", yytext);
                    else
                        printf("Identifier: %s\n", yytext);
                  }

.                   { printf("Unknown symbol: %s\n", yytext); }

%%

int main(int argc, char **argv) {
    printf("LEXICAL ANALYSIS STARTED\n");
    yylex();
    printf("LEXICAL ANALYSIS COMPLETED\n");
    return 0;
}

int yywrap() {
    return 1;
}
```

16.Write a program to design handwritten lexical analyzer using programming language. (Display identifier, symbols and remove comment from program]

```
import re

# List of symbols/operators
symbols = ['+', '-', '*', '/', '=', '(', ')', '{', '}', ';', ',', '<', '>', '==', '!=', '<=', '>=']
```

```
def remove_comments(code):
    # Remove single-line comments
    code = re.sub(r'//.*', '', code)
    # Remove multi-line comments
    code = re.sub(r'/\*[\s\S]*?\*/', '', code)
    return code


def is_identifier(token):
    return re.match(r'^[a-zA-Z_][a-zA-Z0-9_]*$', token)


def analyze_code(code):
    code = remove_comments(code)
    tokens = re.findall(r'[a-zA-Z_][a-zA-Z0-9_]*|==|!=|<=|>=|[=+\-*/(){};,<>]', code)

    identifiers = []
    symbol_list = []

    print("Lexical Analysis Result:\n")

    for token in tokens:
        if token in symbols:
            symbol_list.append(token)
            print(f"Symbol/Operator: {token}")
        elif is_identifier(token):
            identifiers.append(token)
            print(f"Identifier: {token}")
```

```
        else:
            print(f"Unknown token: {token}")


    print("\nSummary:")
    print("Identifiers:", identifiers)
    print("Symbols/Operators:", symbol_list)


# Example program to analyze
program = """
// This is a single-line comment
int a = b + c;  /* multi-line
comment */
if (a > 10) {
    sum = sum + a;
}
"""


# Run the analyzer
analyze_code(program)
```

17.Write a program to implement two- pass assembler. [Generate the symbol table, literal table from ALP code]

```
import re


symbol_table = {}
literal_table = []
intermediate_code = []


location_counter = 0
```

```
# Define instructions for reference (simplified)
instructions = ['START', 'END', 'DS', 'DC', 'MOVER', 'MOVEM', 'ADD', 'SUB', 'JMP']


# Pass 1: Build symbol and literal tables
def pass_one(alp_lines):
    global location_counter
    literal_pool = []


    for line in alp_lines:
        parts = line.strip().split()
        if not parts:
            continue


        if parts[0] == 'START':
            location_counter = int(parts[1])
            continue


        label = None
        if parts[0] not in instructions:
            label = parts[0]
            parts = parts[1:]


        opcode = parts[0]


        # Handle literal
        if len(parts) > 1 and "='" in parts[1]:
```

```
            literal = parts[1].split(',')[-1]
            literal_value = literal.strip()
            if literal_value not in literal_table:
                literal_table.append(literal_value)

        if label:
            if label not in symbol_table:
                symbol_table[label] = location_counter

        if opcode == 'DS':
            location_counter += int(parts[1])
        else:
            location_counter += 1

# Display Symbol and Literal Tables
def display_tables():
    print("SYMBOL TABLE:")
    print(f"{'Symbol':<10} {'Address':<10}")
    for symbol, addr in symbol_table.items():
        print(f"{symbol:<10} {addr:<10}")

    print("\nLITERAL TABLE:")
    print(f"{'Literal':<10} {'Address':<10}")
    literal_address = location_counter
    for literal in literal_table:
        print(f"{literal:<10} {literal_address:<10}")
        literal_address += 1
```

```
# Sample ALP Code
alp_code = [
    "START 100",
    "MOVER AREG, ='5'",
    "ADD BREG, ONE",
    "ONE DS 1",
    "LOOP SUB AREG, ='1'",
    "    JMP LOOP",
    "    END"
]


# Run Two-Pass Assembler Simulation
pass_one(alp_code)
display_tables()
```

All the best