## Q0

1. Design suitable data structures and implement Pass-1 of a two-pass assembler for hypothetical machine. Generate symbol table and Intermediate code file. Implementation should consider
   1. Sample instructions from each category and few assembler directives.
   2. Forward references
   3. Error handling: symbol used but not defined, invalid instruction/register etc
2. Design suitable data structures and implement Pass- 1 of a two-pass assembler for hypothetical machine. Generate Literal table, Pool table and Intermediate code file. Implementation should consider sample instructions from each category and few assembler directives and error handling.
3. Design suitable data structures and implement Pass-2 of a two-pass assembler for hypothetical machine. Consider the output of Assignment-1 (intermediate code file and symbol table) as input for this assignment.
4. Design suitable data structures and implement simple Macro definition processing for hypothetical machine. Generate different Parameter Tablesand MDT, MNT.
5. Implement simple Macro expansion based on Assignment 4. Assume input as macro call to macros defined in Assignment 5.
6. Design a lexical analyzer for ‘C’ language.

## *Q1*

*Design suitable data structures and implement Pass-1 of a two-pass assembler for hypothetical machine. Generate symbol table and Intermediate code file. Implementation should consider*

1. *Sample instructions from each category and few assembler directives.*
2. *Forward references*
3. *Error handling: symbol used but not defined, invalid instruction/register etc*

Creating a two-pass assembler involves implementing two key components: **Pass-1** and **Pass-2**. Here's how we can design **Pass-1** with appropriate data structures and functionality.

**1. Data Structures**

* **Symbol Table**:
  + Stores symbols and their corresponding addresses.
  + Implement as a dictionary or hash map for efficient lookup.
* symbol\_table = {
* "LABEL1": 1000, # Example: "LABEL1" is at address 1000
* "VAR1": 1050 # Example: "VAR1" is at address 1050
* }
* **Intermediate Code (IC)**:
  + Temporarily stores instructions with placeholders for forward references.
  + Implement as a list of tuples or objects.
* intermediate\_code = [
* {"loc": 1000, "instruction": "MOV R1, VAR1", "symbol": "VAR1"},
* {"loc": 1002, "instruction": "JMP LABEL1", "symbol": "LABEL1"}
* ]
* **Opcode Table**:
  + Stores instruction opcodes and formats.
* opcode\_table = {
* "MOV": {"code": "01", "type": "reg\_mem"}, # Example: MOV is register-memory
* "JMP": {"code": "05", "type": "branch"}
* }
* **Error Table**:
  + Stores errors encountered during Pass-1.
* errors = [
* {"line": 3, "error": "Undefined symbol 'LABEL1'"},
* {"line": 5, "error": "Invalid instruction 'LOADX'"}
* ]

**2. Design Features**

* **Sample Instructions**: Include multiple types of instructions:
  + Arithmetic: ADD R1, R2
  + Branching: JMP LABEL
  + Memory operations: MOV R1, VAR
  + Directives: START, END, WORD, RESB
* **Forward References**:
  + Allow placeholder values for symbols not yet defined.
  + Example: If JMP LABEL is encountered but LABEL is not defined, record it in a **forward reference table**:
  + forward\_references = [
  + {"line": 5, "symbol": "LABEL", "location": 1020}
  + ]
* **Error Handling**: Detect errors such as:
  + **Undefined symbols**: Symbols used but not declared.
  + **Invalid instructions**: Instructions not in the opcode table.
  + **Invalid registers**: Registers not supported by the machine.

**3. Algorithm for Pass-1**

**Input:**

* Assembly program as a list of lines.
* Opcode and symbol table initially empty.

**Output:**

* Symbol table.
* Intermediate code file.
* Error table.

**Steps:**

1. **Initialization**:
   * Set location counter (LC) to the starting address (e.g., 0 or the address given by the START directive).
2. **Line-by-Line Processing**: For each line in the input assembly program:
   * **Check if it’s a Label**:
     + If the line starts with a label (e.g., LABEL:), add it to the symbol table with the current LC.
   * **Instruction Parsing**:
     + Validate the instruction against the opcode table.
     + For operands, check if they are symbols, constants, or registers.
     + Handle forward references:
       - If a symbol is not defined yet, add it to the forward references.
   * **Assembler Directives**:
     + Handle directives like START, END, WORD, RESB:
       - START: Set the LC.
       - END: Mark the end of the program.
       - WORD: Reserve memory (increment LC accordingly).
       - RESB: Reserve bytes (increment LC by the specified number).
   * **Error Handling**:
     + Record errors such as undefined symbols, invalid instructions, etc.
   * **Intermediate Code Generation**:
     + Generate a placeholder for machine code, recording the instruction and operands.
3. **Final Symbol Table Update**:
   * After parsing, ensure all forward references are resolved.
   * If any symbols remain undefined, record errors.
4. **Output**:
   * Write the symbol table to a file.
   * Write the intermediate code to a file.

**4. Python Implementation (Pass-1)**

Here's a basic Python implementation:

# Data Structures

symbol\_table = {}

opcode\_table = {"MOV": "01", "JMP": "05", "ADD": "02"}

intermediate\_code = []

forward\_references = []

errors = []

# Input Program

program = [

"START 1000",

"MOV R1, VAR1",

"JMP LABEL1",

"VAR1: WORD 5",

"LABEL1: ADD R1, R2",

"END"

]

# Pass-1 Implementation

def pass1(program):

lc = 0 # Location counter

for line\_no, line in enumerate(program, start=1):

tokens = line.split()

# Handle START Directive

if tokens[0] == "START":

lc = int(tokens[1])

continue

# Handle Labels

if ":" in tokens[0]:

label = tokens[0].replace(":", "")

if label in symbol\_table:

errors.append({"line": line\_no, "error": f"Duplicate label '{label}'"})

else:

symbol\_table[label] = lc

tokens = tokens[1:] # Remove label for further processing

# Handle Instructions

if tokens[0] in opcode\_table:

instruction = tokens[0]

operands = tokens[1:]

for operand in operands:

if operand not in symbol\_table:

forward\_references.append({"line": line\_no, "symbol": operand, "location": lc})

intermediate\_code.append({"loc": lc, "instruction": line, "symbol": operands})

lc += 2 # Assuming 2 bytes per instruction

elif tokens[0] in ["WORD", "RESB"]:

# Handle Directives

if tokens[0] == "WORD":

symbol\_table[tokens[1]] = lc

lc += 1

elif tokens[0] == "RESB":

lc += int(tokens[1])

else:

errors.append({"line": line\_no, "error": f"Invalid instruction '{tokens[0]}'"})

# Run Pass-1

pass1(program)

# Outputs

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

print("Intermediate Code:", intermediate\_code)

print("Forward References:", forward\_references)

print("Errors:", errors)

**5. Output (Example for Given Program)**

**Symbol Table**:

{'VAR1': 1000, 'LABEL1': 1002}

**Intermediate Code**:

[{'loc': 1000, 'instruction': 'MOV R1, VAR1', 'symbol': ['VAR1']},

{'loc': 1002, 'instruction': 'JMP LABEL1', 'symbol': ['LABEL1']}]

**Forward References**:

[{'line': 2, 'symbol': 'VAR1', 'location': 1000},

{'line': 3, 'symbol': 'LABEL1', 'location': 1002}]

**Errors**:

[]

This implementation can be extended to handle Pass-2 by resolving forward references and generating the final machine code.

## Q2

*Design suitable data structures and implement Pass- 1 of a two-pass assembler for hypothetical machine. Generate Literal table, Pool table and Intermediate code file. Implementation should consider sample instructions from each category and few assembler directives and error handling.*

To implement **Pass-1** of a two-pass assembler for a hypothetical machine with a focus on **Literal Table**, **Pool Table**, and **Intermediate Code**, let's first design the appropriate data structures and functionality.

**1. Data Structures**

* **Symbol Table**: Stores symbols (labels) and their corresponding addresses.
* symbol\_table = {
* "LABEL1": 1000,
* "VAR1": 1050
* }
* **Literal Table**: Stores literals and their addresses (resolved during Pass-2).
* literal\_table = [
* {"literal": "=5", "address": None},
* {"literal": "=10", "address": None}
* ]
* **Pool Table**: Tracks the starting index of literal groups in the literal table.
* pool\_table = [0] # First pool starts at index 0
* **Intermediate Code**: Temporarily stores instructions with symbolic references (to be resolved in Pass-2).
* intermediate\_code = [
* {"loc": 1000, "instruction": "MOV R1, =5", "literal\_index": 0},
* {"loc": 1002, "instruction": "JMP LABEL1", "symbol": "LABEL1"}
* ]
* **Error Table**: Logs errors like undefined symbols, invalid instructions, or misplaced literals.
* errors = [
* {"line": 3, "error": "Undefined symbol 'LABEL1'"},
* {"line": 5, "error": "Invalid instruction 'LOADX'"}
* ]

**2. Design Features**

**Assembler Directives:**

* START <address>: Sets the location counter (LC).
* END: Marks the end of the program.
* LTORG: Forces processing of literals in the current pool.
* DS, DC: Define storage.

**Literals:**

* Literals (e.g., =5, =10) are stored in the **Literal Table**.
* When LTORG or END is encountered, literals are assigned addresses from the current LC.

**Error Handling:**

* Detect errors such as:
  + Undefined symbols or invalid instructions.
  + Invalid or misplaced literals.

**3. Algorithm for Pass-1**

**Input:**

* Assembly program as a list of lines.

**Output:**

* Symbol Table
* Literal Table
* Pool Table
* Intermediate Code File
* Error Table

**Steps:**

1. **Initialization**:
   * Set the location counter (LC) to the starting address (from START directive or default 0).
2. **Line-by-Line Parsing**:
   * **Label Handling**:
     + If a label is present, add it to the symbol table with the current LC.
   * **Instruction Handling**:
     + Validate the opcode.
     + If a literal is present, add it to the literal table if not already there.
     + Record the instruction in the intermediate code.
   * **Directive Handling**:
     + START: Initialize the LC.
     + LTORG: Process literals in the current pool.
     + END: Finalize literals in the current pool.
     + DS, DC: Reserve memory or define constants.
   * **Error Detection**:
     + Log errors for undefined symbols, invalid instructions, or misplaced directives.
3. **Finalization**:
   * Assign addresses to any remaining literals in the last pool.
   * Write outputs to files.

**4. Python Implementation**

Here's a Python implementation for Pass-1:

# Data Structures

symbol\_table = {}

literal\_table = []

pool\_table = [0]

intermediate\_code = []

errors = []

# Input Program

program = [

"START 1000",

"MOV R1, =5",

"JMP LABEL1",

"LABEL1: DC 10",

"ADD R2, =10",

"LTORG",

"SUB R1, R2",

"END"

]

# Pass-1 Implementation

def pass1(program):

lc = 0 # Location counter

for line\_no, line in enumerate(program, start=1):

tokens = line.split()

# Handle START Directive

if tokens[0] == "START":

lc = int(tokens[1])

continue

# Handle Labels

if ":" in tokens[0]:

label = tokens[0].replace(":", "")

if label in symbol\_table:

errors.append({"line": line\_no, "error": f"Duplicate label '{label}'"})

else:

symbol\_table[label] = lc

tokens = tokens[1:] # Remove label for further processing

# Handle Instructions

if tokens[0] in ["MOV", "ADD", "SUB", "JMP"]:

instruction = tokens[0]

operands = tokens[1:]

literal\_index = None

for operand in operands:

if operand.startswith("="): # Check for literal

if operand not in [lit["literal"] for lit in literal\_table]:

literal\_table.append({"literal": operand, "address": None})

literal\_index = [lit["literal"] for lit in literal\_table].index(operand)

intermediate\_code.append({

"loc": lc,

"instruction": line,

"literal\_index": literal\_index

})

lc += 2 # Assuming 2 bytes per instruction

elif tokens[0] == "DC":

symbol\_table[tokens[1]] = lc

lc += 1

elif tokens[0] == "LTORG" or tokens[0] == "END":

# Assign addresses to literals in the current pool

for lit in literal\_table[pool\_table[-1]:]:

if lit["address"] is None:

lit["address"] = lc

lc += 1

if tokens[0] == "LTORG":

pool\_table.append(len(literal\_table)) # Start a new pool

else:

errors.append({"line": line\_no, "error": f"Invalid instruction '{tokens[0]}'"})

# Run Pass-1

pass1(program)

# Outputs

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

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

print("Pool Table:", pool\_table)

print("Intermediate Code:", intermediate\_code)

print("Errors:", errors)

**5. Output**

**Symbol Table**:

{'LABEL1': 1002}

**Literal Table**:

[{'literal': '=5', 'address': 1006}, {'literal': '=10', 'address': 1007}]

**Pool Table**:

[0, 2]

**Intermediate Code**:

[{'loc': 1000, 'instruction': 'MOV R1, =5', 'literal\_index': 0},

{'loc': 1002, 'instruction': 'JMP LABEL1', 'literal\_index': None},

{'loc': 1004, 'instruction': 'ADD R2, =10', 'literal\_index': 1}]

**Errors**:

[]

**6. Summary**

This implementation handles:

* Symbol and literal processing.
* Pool management for literals.
* Error detection for common mistakes.
* Generates intermediate code, literal table, pool table, and symbol table.

In **Pass-2**, this data will be used to resolve addresses and generate machine code.

## Q3

*Design suitable data structures and implement Pass-2 of a two-pass assembler for hypothetical machine. Consider the output of Assignment-1 (intermediate code file and symbol table) as input for this assignment.*

The **Pass-2** of a two-pass assembler uses the outputs from Pass-1 (Intermediate Code, Symbol Table, Literal Table, and Pool Table) to generate the final **Machine Code**. Below is the structured approach to design and implement Pass-2.

**1. Input from Pass-1**

1. **Intermediate Code**:
   * A list containing instructions, locations, and references to literals or symbols.
2. intermediate\_code = [
3. {"loc": 1000, "instruction": "MOV R1, =5", "literal\_index": 0},
4. {"loc": 1002, "instruction": "JMP LABEL1", "symbol": "LABEL1"},
5. {"loc": 1004, "instruction": "ADD R2, =10", "literal\_index": 1}
6. ]
7. **Symbol Table**:
   * Maps symbols (e.g., labels or variables) to their memory locations.
8. symbol\_table = {"LABEL1": 1002}
9. **Literal Table**:
   * Maps literals to their resolved memory addresses.
10. literal\_table = [
11. {"literal": "=5", "address": 1006},
12. {"literal": "=10", "address": 1007}
13. ]
14. **Pool Table**:
    * Identifies the starting indices of literal groups in the literal table.
15. pool\_table = [0, 2]

**2. Data Structures for Pass-2**

* **Machine Code**:
  + A list of resolved machine code instructions generated from the intermediate code.
* machine\_code = [
* {"loc": 1000, "code": "01 01 1006"}, # Example: MOV R1, memory address 1006
* {"loc": 1002, "code": "05 1002"}, # Example: JMP to address 1002
* {"loc": 1004, "code": "02 02 1007"} # Example: ADD R2, memory address 1007
* ]
* **Error Table**:
  + Logs any errors encountered during the resolution of symbols or literals.
* errors = [{"line": 2, "error": "Undefined symbol 'LABEL2'"}]

**3. Algorithm for Pass-2**

**Input:**

* Intermediate Code
* Symbol Table
* Literal Table
* Pool Table

**Output:**

* Machine Code
* Error Table

**Steps:**

1. **Initialization**:
   * Prepare an empty list for the machine code.
   * Initialize an error table to log any issues.
2. **Processing Intermediate Code**:
   * For each instruction in the intermediate code:
     + Parse the instruction and identify the opcode and operands.
     + Resolve **symbols** using the symbol table:
       - If the symbol is not defined, log an error.
     + Resolve **literals** using the literal table and literal index.
     + Generate the machine code instruction based on the resolved operands and opcode.
3. **Generate Machine Code**:
   * For each instruction:
     + Translate mnemonics (e.g., MOV, ADD) into opcodes.
     + Replace symbolic references with resolved addresses.
     + Format the instruction as binary or hexadecimal machine code.
4. **Error Handling**:
   * Log errors for undefined symbols, missing literals, or invalid instructions.
5. **Output**:
   * Write the machine code to a file.
   * Write errors to a log file.

**4. Python Implementation**

# Inputs from Pass-1

intermediate\_code = [

{"loc": 1000, "instruction": "MOV R1, =5", "literal\_index": 0},

{"loc": 1002, "instruction": "JMP LABEL1", "symbol": "LABEL1"},

{"loc": 1004, "instruction": "ADD R2, =10", "literal\_index": 1}

]

symbol\_table = {"LABEL1": 1002}

literal\_table = [

{"literal": "=5", "address": 1006},

{"literal": "=10", "address": 1007}

]

pool\_table = [0, 2]

opcode\_table = {

"MOV": "01", # Opcode for MOV

"ADD": "02", # Opcode for ADD

"JMP": "05" # Opcode for JMP

}

# Data Structures for Pass-2

machine\_code = []

errors = []

# Pass-2 Implementation

def pass2(intermediate\_code, symbol\_table, literal\_table):

for entry in intermediate\_code:

loc = entry["loc"]

instruction = entry["instruction"].split()

opcode = instruction[0] # Instruction mnemonic

operands = instruction[1:] # Operands

# Check if opcode exists

if opcode not in opcode\_table:

errors.append({"loc": loc, "error": f"Invalid opcode '{opcode}'"})

continue

# Initialize machine code for this line

machine\_line = f"{opcode\_table[opcode]} "

# Process operands

for operand in operands:

if operand.startswith("="): # Literal

literal\_index = entry.get("literal\_index")

if literal\_index is not None:

address = literal\_table[literal\_index]["address"]

machine\_line += f"{address} "

else:

errors.append({"loc": loc, "error": f"Unresolved literal '{operand}'"})

elif operand in symbol\_table: # Symbol

address = symbol\_table[operand]

machine\_line += f"{address} "

elif operand.startswith("R"): # Register

machine\_line += f"{operand[-1]} " # Assume single-digit register numbers

else:

errors.append({"loc": loc, "error": f"Invalid operand '{operand}'"})

# Add machine line to output

machine\_code.append({"loc": loc, "code": machine\_line.strip()})

# Run Pass-2

pass2(intermediate\_code, symbol\_table, literal\_table)

# Outputs

print("Machine Code:", machine\_code)

print("Errors:", errors)

**5. Output Example**

**Machine Code:**

[

{"loc": 1000, "code": "01 1 1006"},

{"loc": 1002, "code": "05 1002"},

{"loc": 1004, "code": "02 2 1007"}

]

**Errors:**

[]

**6. Summary**

* **Pass-2 Workflow**:
  + Reads the intermediate code, symbol table, and literal table.
  + Resolves addresses for symbols and literals.
  + Converts mnemonics to opcodes and generates machine code.
* **Key Features**:
  + Error handling for undefined symbols and invalid literals.
  + Outputs final machine code in binary or hexadecimal format.
* **Next Steps**:
  + Write machine code and errors to respective files.
  + Optimize for compact binary representation if required.

## Q4

*Design suitable data structures and implement simple Macro definition processing for hypothetical machine. Generate different Parameter Tables and MDT, MNT.*

Macro definition processing involves identifying macros, defining them with their parameters, and generating the necessary tables (Macro Name Table (MNT), Macro Definition Table (MDT), and Parameter Tables). This enables macros to be expanded during assembly.

**1. Definitions and Tables**

**Macro Definition Table (MDT):**

Stores the macro definition, including all its instructions, starting from the first statement after MACRO and ending before MEND.

MDT Index | Instruction

0 | LOAD &ARG1

1 | ADD &ARG2

2 | STORE RESULT

3 | MEND

**Macro Name Table (MNT):**

Maps the macro name to its index in the MDT and the parameter names.

MNT Index | Macro Name | MDT Index | Parameter Table Index

0 | MAC1 | 0 | 0

**Parameter Name Table (PNT):**

Stores positional names and their indices for each macro.

PNT Index | Parameter Name

0 | &ARG1

1 | &ARG2

**Actual Parameter Table (APT):**

Used during macro expansion to store the values of actual arguments.

APT Index | Parameter Value

0 | VALUE1

1 | VALUE2

**2. Data Structures in Python**

1. **MDT**:
2. mdt = []
3. **MNT**:
4. mnt = []
5. **PNT**:
6. pnt = []
7. **APT** (for macro expansion):
8. apt = []

**3. Algorithm for Macro Definition Processing**

**Input:**

* Assembly program with macro definitions and invocations.

**Output:**

* MDT: Macro Definition Table.
* MNT: Macro Name Table.
* PNT: Parameter Name Table.

**Steps:**

1. **Initialization**:
   * Prepare empty tables for MDT, MNT, and PNT.
   * Maintain indices for MDT and MNT entries.
2. **Macro Definition**:
   * When encountering MACRO, start a new macro definition.
   * Extract the macro name and parameters, and add them to MNT and PNT.
   * Add subsequent instructions to MDT until MEND is encountered.
3. **Macro Invocation**:
   * When encountering a macro invocation, locate its entry in the MNT.
   * Use the PNT to map actual parameters to formal parameters.
   * Replace macro calls with their expanded definitions.

**4. Python Implementation**

# Data Structures

mdt = [] # Macro Definition Table

mnt = [] # Macro Name Table

pnt\_table = [] # List of Parameter Name Tables

apt = [] # Actual Parameter Table

# Example Assembly Program with Macros

program = [

"MACRO",

"MAC1 &ARG1 &ARG2",

"LOAD &ARG1",

"ADD &ARG2",

"STORE RESULT",

"MEND",

"START",

"MAC1 VALUE1, VALUE2",

"END"

]

# Macro Definition Processor

def macro\_processor(program):

in\_macro = False

pnt = {} # Current Parameter Name Table

mnt\_index = 0

mdt\_index = 0

for line in program:

tokens = line.split()

# Start of Macro Definition

if tokens[0] == "MACRO":

in\_macro = True

continue

# End of Macro Definition

if tokens[0] == "MEND":

mdt.append(line)

in\_macro = False

continue

# Within Macro Definition

if in\_macro:

if tokens[0] not in ["MACRO", "MEND"]:

if len(mnt) == mnt\_index:

# Add to MNT

macro\_name = tokens[0]

params = tokens[1:]

pnt = {param: idx for idx, param in enumerate(params)}

pnt\_table.append(pnt)

mnt.append({

"Macro Name": macro\_name,

"MDT Index": mdt\_index,

"PNT Index": mnt\_index

})

mnt\_index += 1

else:

# Replace parameters with PNT indices

instruction = " ".join(

str(pnt.get(token, token)) if token.startswith("&") else token

for token in tokens

)

mdt.append(instruction)

mdt\_index += 1

continue

# Macro Invocation

if tokens[0] in [entry["Macro Name"] for entry in mnt]:

macro\_entry = next(entry for entry in mnt if entry["Macro Name"] == tokens[0])

pnt = pnt\_table[macro\_entry["PNT Index"]]

apt = {idx: arg for idx, arg in enumerate(tokens[1:])}

mdt\_start = macro\_entry["MDT Index"]

mdt\_end = mdt\_start

while mdt[mdt\_end] != "MEND":

mdt\_end += 1

# Expand macro

for macro\_line in mdt[mdt\_start:mdt\_end]:

expanded\_line = " ".join(

apt[int(token)] if token.isdigit() else token

for token in macro\_line.split()

)

print(expanded\_line) # Simulate expanded code output

else:

print(line) # Non-macro lines

# Run Macro Processor

macro\_processor(program)

# Outputs

print("MDT:", mdt)

print("MNT:", mnt)

print("PNT:", pnt\_table)

**5. Example Output**

Given the program:

MACRO

MAC1 &ARG1 &ARG2

LOAD &ARG1

ADD &ARG2

STORE RESULT

MEND

START

MAC1 VALUE1, VALUE2

END

**Macro Definition Table (MDT)**:

[

"LOAD &ARG1",

"ADD &ARG2",

"STORE RESULT",

"MEND"

]

**Macro Name Table (MNT)**:

[

{"Macro Name": "MAC1", "MDT Index": 0, "PNT Index": 0}

]

**Parameter Name Table (PNT)**:

[

{"&ARG1": 0, "&ARG2": 1}

]

**Expanded Code**:

START

LOAD VALUE1

ADD VALUE2

STORE RESULT

END

**6. Summary**

* **Macro Definition**:
  + Captured in MDT with parameters stored in PNT.
* **Macro Invocation**:
  + Resolved using MNT, replacing parameters with actual arguments.
* **Expansion**:
  + Generates expanded assembly code by replacing parameter placeholders with actual values.
* **Use Cases**:
  + Simplifies repetitive code, allowing reusable macro definitions.

## Q5

*Implement simple Macro expansion based on earlier Assignment 4. Assume input as macro call to macros defined in Assignment 5.*

The task is to implement **macro expansion**, where macro invocations in the input program are replaced by their expanded definitions using the macro tables (MDT, MNT, PNT) generated during macro definition processing.

Here's a structured approach:

**1. Data Structures**

1. **Macro Definition Table (MDT)**: Stores the instructions of macro definitions.
2. mdt = [
3. "LOAD &ARG1",
4. "ADD &ARG2",
5. "STORE RESULT",
6. "MEND"
7. ]
8. **Macro Name Table (MNT)**: Maps macro names to MDT entries and PNT indices.
9. mnt = [
10. {"Macro Name": "MAC1", "MDT Index": 0, "PNT Index": 0}
11. ]
12. **Parameter Name Table (PNT)**: Stores positional names for macro parameters.
13. pnt\_table = [
14. {"&ARG1": 0, "&ARG2": 1}
15. ]
16. **Actual Parameter Table (APT)**: Stores the actual values passed during macro invocation.
17. apt = []

**2. Algorithm for Macro Expansion**

**Input:**

* Assembly program with macro calls.
* MNT, MDT, and PNT generated from macro definition processing.

**Output:**

* Expanded program with macro calls replaced by their definitions.

**Steps:**

1. **Initialization**:
   * Load the input program and the macro tables (MNT, MDT, PNT).
2. **Macro Invocation Detection**:
   * For each line in the input program:
     + Check if it matches a macro name in MNT.
3. **Process Macro Invocation**:
   * If a macro call is detected:
     + Look up the corresponding MNT entry.
     + Map actual parameters to formal parameters using the PNT and APT.
     + Fetch the macro definition from MDT and replace parameter placeholders with actual values.
     + Append expanded instructions to the output program.
4. **Non-Macro Lines**:
   * Copy non-macro lines directly to the output program.
5. **Output**:
   * Generate the final expanded assembly program.

**3. Python Implementation**

# Macro Tables

mdt = [

"LOAD &ARG1",

"ADD &ARG2",

"STORE RESULT",

"MEND"

]

mnt = [

{"Macro Name": "MAC1", "MDT Index": 0, "PNT Index": 0}

]

pnt\_table = [

{"&ARG1": 0, "&ARG2": 1}

]

# Input Program with Macro Invocation

program = [

"START",

"MAC1 VALUE1, VALUE2",

"END"

]

def expand\_macros(program, mnt, mdt, pnt\_table):

expanded\_program = []

for line in program:

tokens = line.split()

macro\_name = tokens[0]

# Check if line is a macro call

if macro\_name in [entry["Macro Name"] for entry in mnt]:

# Fetch MNT entry

mnt\_entry = next(entry for entry in mnt if entry["Macro Name"] == macro\_name)

mdt\_index = mnt\_entry["MDT Index"]

pnt = pnt\_table[mnt\_entry["PNT Index"]]

# Map actual parameters to formal parameters

actual\_params = tokens[1].split(",") # Assuming comma-separated arguments

apt = {pnt[param]: actual\_params[idx] for param, idx in pnt.items()}

# Expand macro from MDT

while mdt[mdt\_index] != "MEND":

instruction = mdt[mdt\_index]

expanded\_line = " ".join(

apt[int(token[1:])] if token.startswith("&") else token

for token in instruction.split()

)

expanded\_program.append(expanded\_line)

mdt\_index += 1

else:

# Non-macro line

expanded\_program.append(line)

return expanded\_program

# Expand the program

expanded\_program = expand\_macros(program, mnt, mdt, pnt\_table)

# Output the expanded program

print("Expanded Program:")

for line in expanded\_program:

print(line)

**4. Input Example**

Input Program:

START

MAC1 VALUE1, VALUE2

END

MDT:

[

"LOAD &ARG1",

"ADD &ARG2",

"STORE RESULT",

"MEND"

]

MNT:

[

{"Macro Name": "MAC1", "MDT Index": 0, "PNT Index": 0}

]

PNT:

[

{"&ARG1": 0, "&ARG2": 1}

]

**5. Output Example**

Expanded Program:

START

LOAD VALUE1

ADD VALUE2

STORE RESULT

END

**6. Summary**

* **Macro Expansion**:
  + Replaces macro invocations with expanded instructions.
  + Resolves parameter placeholders using PNT and actual values.
* **Data Structures**:
  + MDT: Stores macro definitions.
  + MNT: Maps macro names to MDT entries.
  + PNT: Manages formal parameters.
  + APT: Maps actual parameters during invocation.
* **Python Code**:
  + Processes macro calls dynamically, expanding them inline in the output.

## Q6

*Design a lexical analyzer for ‘C’ language.*

Designing a lexical analyzer for the **C programming language** involves processing input source code and breaking it down into a stream of **tokens**. Tokens are the smallest elements in the program, such as keywords, identifiers, literals, operators, and punctuation.

**1. Components of a Lexical Analyzer**

**1.1. Input**

The source code written in C (as a text file or string).

**1.2. Output**

A list of tokens, each categorized as a specific type (e.g., keyword, identifier, etc.), and its associated value.

**1.3. Token Types**

1. **Keywords**: if, else, while, return, etc.
2. **Identifiers**: Variable names, function names, etc.
3. **Literals**:
   * **Numeric literals**: 123, 45.67
   * **String literals**: "Hello World"
   * **Character literals**: 'a'
4. **Operators**: +, -, \*, /, ==, !=, etc.
5. **Punctuation**: ;, {, }, (, ), etc.
6. **Comments**: /\* \*/, //.

**1.4. States**

A finite set of states guides the lexical analyzer to process characters and classify them into tokens.

**2. Approach**

**2.1. Steps**

1. **Read Input**: Read the input source code line by line.
2. **Scan Characters**: Traverse the characters of the input.
3. **Token Matching**: Match substrings to known patterns (e.g., keywords, operators).
4. **Skip Whitespace and Comments**: Ignore irrelevant characters.
5. **Error Handling**: Identify invalid tokens and report errors.

**2.2. Token Regular Expressions**

* Keywords: if|else|while|return|int|float|char|void
* Identifiers: [a-zA-Z\_][a-zA-Z0-9\_]\*
* Numeric Literals: [0-9]+(\.[0-9]+)?
* String Literals: "(\\.|[^"\\])\*"
* Character Literals: '(\\.|[^'\\])'
* Operators: \+|\-|\\*|\/|==|!=|<=|>=|<|>|=
* Punctuation: \(|\)|\{|\}|\[|\]|;|,

**3. Implementation in Python**

Here’s a Python implementation of a basic lexical analyzer:

import re

# Token types

TOKEN\_TYPES = {

"KEYWORD": r"\b(if|else|while|return|int|float|char|void)\b",

"IDENTIFIER": r"\b[a-zA-Z\_][a-zA-Z0-9\_]\*\b",

"NUMERIC\_LITERAL": r"\b\d+(\.\d+)?\b",

"STRING\_LITERAL": r'"(\\.|[^"\\])\*"',

"CHAR\_LITERAL": r"'(\\.|[^'\\])'",

"OPERATOR": r"==|!=|<=|>=|<|>|=|\+|\-|\\*|\/",

"PUNCTUATION": r"[{}()\[\];,]",

"COMMENT": r"(//.\*?$|/\\*.\*?\\*/)",

"WHITESPACE": r"\s+"

}

# Regex patterns

TOKEN\_REGEX = re.compile("|".join(f"(?P<{type}>{regex})" for type, regex in TOKEN\_TYPES.items()), re.DOTALL)

# Lexical Analyzer

def lexical\_analyzer(source\_code):

tokens = []

for match in TOKEN\_REGEX.finditer(source\_code):

token\_type = match.lastgroup

value = match.group(token\_type)

if token\_type == "WHITESPACE" or token\_type == "COMMENT":

continue # Skip comments and whitespace

tokens.append((token\_type, value))

return tokens

# Test Input

source\_code = """

int main() {

// This is a comment

int a = 10;

float b = 20.5;

if (a < b) {

return a + b;

}

}

"""

# Analyze the code

tokens = lexical\_analyzer(source\_code)

# Print Tokens

for token in tokens:

print(f"{token[0]}: {token[1]}")

**4. Example Output**

For the input source code:

int main() {

// This is a comment

int a = 10;

float b = 20.5;

if (a < b) {

return a + b;

}

}

The output tokens would be:

KEYWORD: int

IDENTIFIER: main

PUNCTUATION: (

PUNCTUATION: )

PUNCTUATION: {

KEYWORD: int

IDENTIFIER: a

OPERATOR: =

NUMERIC\_LITERAL: 10

PUNCTUATION: ;

KEYWORD: float

IDENTIFIER: b

OPERATOR: =

NUMERIC\_LITERAL: 20.5

PUNCTUATION: ;

KEYWORD: if

PUNCTUATION: (

IDENTIFIER: a

OPERATOR: <

IDENTIFIER: b

PUNCTUATION: )

PUNCTUATION: {

KEYWORD: return

IDENTIFIER: a

OPERATOR: +

IDENTIFIER: b

PUNCTUATION: ;

PUNCTUATION: }

PUNCTUATION: }

**5. Enhancements**

1. **Error Handling**:
   * Detect unclosed strings or invalid characters.
   * Report errors with line and column numbers.
2. **Symbol Table Integration**:
   * Store identifiers in a symbol table for later use in semantic analysis.
3. **Extending Features**:
   * Add support for preprocessor directives (#include, #define).
   * Expand regex patterns for additional operators and data types.

**6. Summary**

* This implementation tokenizes C source code by matching patterns against regular expressions.
* It handles common token types like keywords, identifiers, literals, operators, punctuation, and comments.
* Future extensions can include better error handling, preprocessor support, and integration with later compiler stages.