**🧪 Assignment: Pass 2 of Two Pass Assembler (Pseudo Machine)**

**📌 Title:**

**Design of Pass 2 of Two Pass Assembler**

**🎯 Aim:**

To design and implement **Pass 2** of a **Two Pass Assembler** using suitable data structures in **Java** for a pseudo machine.

**📚 Objective:**

To synthesize machine-level code using the **Intermediate Code** and **Symbol Table** generated during **Pass 1**.

**📖 Theory (Summary):**

**🛠️ 1. Assembler Synthesis Phase (Pass 2):**

* Reads **Intermediate Code**, **SYMTAB**, **LITTAB**, and **POOLTAB**.
* Replaces symbolic references with actual **memory addresses**.
* Converts each line of IC into actual **machine code (object code)**.

**📜 2. Algorithm for Pass 2:**

1. Read IC line by line.
2. For each IC instruction:
   * Translate opcode to binary using OPTAB.
   * Resolve symbol/literal from SYMTAB/LITTAB.
   * Generate machine instruction.
3. Handle pseudo-ops (DC, DS) for memory allocation.
4. Output object code line by line.

**⚠️ 3. Error Handling in Pass 2:**

* Undefined symbols or literals.
* Invalid instruction format.
* Memory out-of-bounds.
* Illegal operand type.

**📥 Input:**

* **Intermediate Code (IC)** from Pass 1
* **Symbol Table (SYMTAB)**
* **Literal Table (LITTAB)**
* **Opcode Table (OPTAB)**

**📤 Output:**

* Final **Object Code** (Machine Language Translation)

**✅ Conclusion:**

Pass 2 completes the assembler process by converting symbolic intermediate code into actual machine instructions using SYMTAB and LITTAB.

**🎤 Viva Questions & One-Line Answers**

**🧠 Assembler Basics**

1. **How is a single pass assembler different from a two pass assembler?**  
   → A single pass resolves symbols and generates code simultaneously, while two pass separates these phases.
2. **What is backpatching and why is it required?**  
   → Backpatching fills in addresses for forward-referenced symbols that are undefined during the first scan.

**⚙️ Pass 2 Specific**

1. **What is the function of Pass 2?**  
   → It generates the final machine code by resolving symbols and opcodes from intermediate code.
2. **What are the inputs to Pass 2?**  
   → Intermediate Code, SYMTAB, LITTAB, and OPTAB.
3. **What is the output of Pass 2?**  
   → The object program (machine language translation) of the source code.
4. **How does Pass 2 handle DS and DC?**  
   → DS reserves memory; DC defines and stores constant data in object code.
5. **How does Pass 2 resolve symbols?**  
   → By looking up addresses from SYMTAB and LITTAB.
6. **What if a symbol is undefined in SYMTAB during Pass 2?**  
   → An error is thrown or flagged for undefined symbol.

**📘 Variants and Examples**

1. **Explain Variant 1 and Variant 2 with example.**  
   → Variant 1 uses register + memory operand; Variant 2 uses two memory addresses or constant + memory (depends on instruction format).

*Example:*  
ADD AREG, SYMBOL → Variant 1  
MOV SYMBOL1, SYMBOL2 → Variant 2

**🔍 Error Handling**

1. **What errors are detected in Pass 2?**  
   → Undefined symbols/literals, invalid instruction formats, and illegal operand types.
2. **How are errors displayed to the user?**  
   → Errors are listed with line numbers and messages during code generation.

**💻 Implementation Details**

1. **Which data structures are used in Pass 2?**  
   → HashMaps for SYMTAB, LITTAB, OPTAB; Lists for Intermediate Code and Output.
2. **Why is OPTAB required in Pass 2?**  
   → To convert mnemonics into actual machine opcodes.
3. **What does the final output of Pass 2 look like?**  
   → A list of machine instructions with resolved memory addresses and opcodes.
4. **What is the role of the location counter in Pass 2?**  
   → It helps keep track of the address at which each machine instruction is placed.

import java.io.\*;

import java.util.HashMap;

public class MachineCodeGenerator {

    public static void main(String[] args) {

        String intermediateFile = "intermediate.txt";

        String symbolTableFile = "symbol\_table.txt";

        String outputFile = "machine\_code.txt";

        HashMap<Integer, Integer> symbolTable = new HashMap<>();

        // Step 1: Read Symbol Table and Store in HashMap

        try (BufferedReader br = new BufferedReader(new FileReader(symbolTableFile))) {

            String line;

            while ((line = br.readLine()) != null) {

                String[] tokens = line.split("\\s+");

                if (tokens.length == 2) {

                    symbolTable.put(Integer.parseInt(tokens[0]), Integer.parseInt(tokens[1]));

                }

            }

        } catch (IOException e) {

            System.err.println("Error reading symbol table: " + e.getMessage());

            return;

        }

        // Step 2: Process Intermediate Code and Generate Machine Code

        try (BufferedReader br = new BufferedReader(new FileReader(intermediateFile));

             BufferedWriter bw = new BufferedWriter(new FileWriter(outputFile))) {

            String line;

            while ((line = br.readLine()) != null) {

                String[] tokens = line.split("\\s+");

                if (tokens.length < 3) {

                    System.out.println("Invalid format: " + line);

                    continue;

                }

                int LC = Integer.parseInt(tokens[0]); // Location Counter

                String opcode = tokens[1].replaceAll("[^0-9]", ""); // Extract opcode

                String operandType = tokens[2].substring(1, 2); // Extract type (S = Symbol, C = Constant)

                String operandValue = tokens[2].replaceAll("[^0-9]", ""); // Extract operand value

                int operand;

                if (operandType.equals("S")) {

                    operand = symbolTable.getOrDefault(Integer.parseInt(operandValue), -1);

                } else {

                    operand = Integer.parseInt(operandValue);

                }

                // Write to output file

                String machineCode = LC + " " + opcode + " " + operand;

                bw.write(machineCode);

                bw.newLine();

                System.out.println(machineCode);

            }

            System.out.println("Machine code generated in " + outputFile);

        } catch (IOException e) {

            System.err.println("Error processing intermediate code: " + e.getMessage());

        }

    }

}