Assignment-1

**🧪 Experiment: Design of Pass 1 of Two Pass Assembler for Pseudo Machine (Java)**

**🔍 Aim:**

To design and implement **Pass 1** of a **Two Pass Assembler** using suitable **data structures** in **Java** for a pseudo machine.

**📖 Theory (Summary):**

* **Assembler**: Translates assembly language to machine code.
* **Two Pass Assembler**:
  + **Pass 1** builds tables (SYMTAB, LITTAB, POOLTAB), handles directives (START, EQU, etc.), and generates **Intermediate Code**.
  + **Pass 2** uses these tables to generate **final machine code**.
* **OPTAB**: Stores operation codes and instruction classes (IS, AD, DL).
* **SYMTAB**: Stores symbols and their addresses.
* **LITTAB**: Stores literals and assigned addresses.
* **POOLTAB**: Tracks literal pools for LTORG and END.
* **Error Handling in Pass 1**: Duplicate symbols, undefined symbols, invalid opcodes, and incorrect directive usage.

**📥 Input:**

An assembly language program for a pseudo machine with labels, instructions, symbols, and directives.

**📤 Output:**

* **SYMTAB** (Symbol Table)
* **OPTAB** (Opcode Table)
* **LITTAB & POOLTAB**
* **Intermediate Code (IC)**

**✅ Conclusion:**

The experiment gives practical knowledge of how assemblers resolve symbols and literals using internal tables, and builds the foundation for compiler design.

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

**🔧 Assembler Basics**

1. **What is the purpose of an assembler?**  
   → It translates assembly language into machine code.
2. **What is a two-pass assembler?**  
   → An assembler that performs symbol resolution in pass 1 and code generation in pass 2.
3. **What is the function of pass 1?**  
   → It creates SYMTAB, LITTAB, POOLTAB, and generates intermediate code.

**📜 Tables**

1. **What is SYMTAB?**  
   → A table that stores symbols and their corresponding addresses.
2. **What is LITTAB?**  
   → A table storing all literals and their assigned memory addresses.
3. **What is POOLTAB?**  
   → A table used to manage multiple pools of literals in the code.
4. **What is OPTAB?**  
   → A table storing mnemonics, their classes, and opcodes.
5. **What is the use of Intermediate Code?**  
   → It helps bridge the gap between pass 1 and pass 2 for final code generation.

**⚙️ Assembler Directives**

1. **What is the use of START directive?**  
   → It sets the starting address of the program.
2. **What does EQU do?**  
   → It assigns a constant value or address to a symbol.
3. **What is the role of ORIGIN?**  
   → It changes the location counter to a specified address.
4. **What is LTORG used for?**  
   → It forces allocation of memory to literals at that point.
5. **What is the significance of END?**  
   → It marks the end of the program and triggers literal assignment.

**🪛 Error Handling**

1. **What errors are detected in Pass 1?**  
   → Duplicate symbols, undefined symbols, invalid opcodes, syntax errors.
2. **What happens with a duplicate symbol?**  
   → An error is raised and the symbol is not inserted again in SYMTAB.
3. **What if a symbol is used before being defined?**  
   → It is added to SYMTAB with undefined address for resolution later.

**💻 Implementation**

1. **Which data structures are used for SYMTAB and OPTAB?**  
   → HashMaps or Lists are used for efficient storage and lookup.
2. **How are literals handled in Pass 1?**  
   → They are stored in LITTAB and assigned addresses during LTORG or END.
3. **Why is Intermediate Code important?**  
   → It helps Pass 2 generate final machine code using resolved addresses.
4. **How does DS and DC affect location counter?**  
   → DS reserves space, DC defines a constant and both increment LC.
5. **What is the class of instructions in OPTAB?**  
   → IS (Imperative), AD (Assembler Directive), DL (Declarative).

import java.io.\*;

import java.util.HashMap;

public class assembler {

    public static void main(String[] args) {

        System.out.println("Hello, Assembler");

        // Opcode Table

        HashMap<String, String[]> opcodeTab = new HashMap<>();

        opcodeTab.put("STOP", new String[]{"IS", "00"});

        opcodeTab.put("ADD", new String[]{"IS", "01"});

        opcodeTab.put("SUB", new String[]{"IS", "02"});

        opcodeTab.put("MULT", new String[]{"IS", "03"});

        opcodeTab.put("MOVER", new String[]{"IS", "04"});

        opcodeTab.put("MOVEM", new String[]{"IS", "05"});

        opcodeTab.put("COMP", new String[]{"IS", "06"});

        opcodeTab.put("BC", new String[]{"IS", "07"});

        opcodeTab.put("DIV", new String[]{"IS", "08"});

        opcodeTab.put("READ", new String[]{"IS", "09"});

        opcodeTab.put("PRINT", new String[]{"IS", "10"});

        opcodeTab.put("DC", new String[]{"DL", "01"});

        opcodeTab.put("DS", new String[]{"DL", "02"});

        opcodeTab.put("START", new String[]{"AD", "01"});

        opcodeTab.put("END", new String[]{"AD", "02"});

        opcodeTab.put("ORIGIN", new String[]{"AD", "03"});  // Added support for ORIGIN

        // Register Table

        HashMap<String, String> registerTab = new HashMap<>();

        registerTab.put("AREG", "1");

        registerTab.put("BREG", "2");

        registerTab.put("CREG", "3");

        registerTab.put("DREG", "4");

        String filepath = "input.txt";

        int locationCounter = 0;

        // Symbol Table

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

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

            String line;

            System.out.println("Processing instructions from file:");

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

                // Trim and ignore empty lines

                line = line.trim();

                if (line.isEmpty()) continue;

                // Split the line into tokens

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

                int tokenCount = tokens.length;

                if (tokenCount == 0) continue;

                String instruction = "";

                String operand1 = "";

                String operand2 = "";

                String label = null;

                // If the first token is an instruction

                if (opcodeTab.containsKey(tokens[0])) {

                    instruction = tokens[0];

                    if (tokenCount > 1) operand1 = tokens[1];

                    if (tokenCount > 2) operand2 = tokens[2];

                }

                // If the first token is NOT an instruction, assume it's a label

                else {

                    label = tokens[0];

                    if (tokenCount > 1) instruction = tokens[1];

                    if (tokenCount > 2) operand1 = tokens[2];

                    if (tokenCount > 3) operand2 = tokens[3];

                }

                // Handle START directive

                if ("START".equals(instruction)) {

                    locationCounter = operand1.isEmpty() ? 0 : Integer.parseInt(operand1);

                    System.out.println("LC initialized to: " + locationCounter);

                    continue;

                }

                // Handle ORIGIN directive (change location counter)

                if ("ORIGIN".equals(instruction)) {

                    locationCounter = Integer.parseInt(operand1);

                    System.out.println("LC updated to: " + locationCounter + " due to ORIGIN");

                    continue;

                }

                // If the instruction is END, stop processing

                if ("END".equals(instruction)) {

                    System.out.println("END reached. Final LC: " + locationCounter);

                    break;

                }

                // If a label is encountered, add it to the symbol table

                if (label != null) {

                    if (!symbolTable.containsKey(label)) {

                        symbolTable.put(label, locationCounter);

                    }

                    // After processing the label, treat it like an instruction

                    instruction = tokens[1];  // Next token is the instruction

                    if (tokenCount > 2) operand1 = tokens[2];

                    if (tokenCount > 3) operand2 = tokens[3];

                }

                // If instruction is valid

                if (opcodeTab.containsKey(instruction)) {

                    String[] details = opcodeTab.get(instruction);

                    // If operands contain registers, replace them with corresponding values

                    if (registerTab.containsKey(operand1)) {

                        operand1 = registerTab.get(operand1);

                    }

                    // Handle DL type instructions (DC, DS)

                    if (instruction.equals("DC")) {

                        System.out.println("LC=" + locationCounter + " (" + details[0] + ", " + details[1] + ") (C, " + operand1 + ")");

                        locationCounter++;

                    } else if (instruction.equals("DS")) {

                        int size = Integer.parseInt(operand1);

                        System.out.println("LC=" + locationCounter + " (" + details[0] + ", " + details[1] + ") (C, " + operand1 + ") -> Allocating " + size + " memory units");

                        locationCounter += size;

                    } else {

                        System.out.println("LC=" + locationCounter + " (" + details[0] + ", " + details[1] + ") " +

                        (!operand1.isEmpty() ? "(C, " + operand1 + ")" : "") + " " +

                        (!operand2.isEmpty() ? "(C, " + operand2 + ")" : ""));

                        locationCounter++;

                    }

                } else {

                    System.out.println("Error: Unknown instruction '" + instruction + "' in line: " + line);

                }

            }

            // Print the symbol table

            System.out.println("\nSymbol Table:");

            for (String symbol : symbolTable.keySet()) {

                System.out.println(symbol + " => " + symbolTable.get(symbol));

            }

        } catch (IOException e) {

            System.err.println("Error reading the file: " + e.getMessage());

        }

    }

}