```
package ass1assemblerpass1;

import java.io.BufferedReader;
import java.io.FileInputStream;
import java.io.FileWriter;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.util.ArrayList;
import java.util.Collections;
import java.util.HashMap;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;
import java.util.StringTokenizer;

//Desgining MOT
class Tuple {

    String mnemonic, bin_opcode, type;
    int length;

    Tuple() {
    }

    Tuple(String s1, String s2, String s3, String s4) {
        mnemonic = s1;
        bin_opcode = s2;
        length = Integer.parseInt(s3);
        type = s4;
    }
}
//Desgining ST

class SymTuple {

    String symbol, ra;
    int value, length;

    SymTuple(String s1, int i1, int i2, String s2) {
        symbol = s1;
        value = i1;
        length = i2;
        ra = s2;
    }
}
//Designing Literal

class LitTuple {

    String literal, ra;
    int value, length;

    LitTuple() {
    }
```

```
    LitTuple(String s1, int i1, int i2, String s2) {
        literal = s1;
        value = i1;
        length = i2;
        ra = s2;
    }
}
//Actual Code

public class ass1assemblerpass1 {

    static int lc;
    static List<Tuple> mot; //required to read MOT
    static List<String> pot; //required to read POT
    static List<SymTuple> symtable; //generate symbol table
    static List<LitTuple> littable; //generate literal table
    static List<Integer> lclist;
    static Map<Integer, Integer> basetable; //base table
    static PrintWriter out_pass2; //output of pass 2
    static PrintWriter out_pass1; //output of pass 1
    static int line_no;

    public static void main(String[] args) throws Exception {
        initializeTables(); //initialize everything needed
        System.out.println("====== PASS 1 ======\n");
        pass1(); //Run Pass 1

        //exporting lclist to file, so that it can be used in pass2
        PrintWriter lclistWriter = new PrintWriter(new FileWriter("/home/student/workspace/SPOSL/src/lclist.t
xt"), true); //generate ST
        for (int i = 0; i < lclist.size(); i++) {
            lclistWriter.println(lclist.get(i));
        }
        lclistWriter.close();
    }

    static void initializeTables() throws Exception {
        symtable = new LinkedList<>();
        littable = new LinkedList<>();
        lclist = new ArrayList<>();
        basetable = new HashMap<>();
        mot = new LinkedList<>();
        pot = new LinkedList<>();
        String s;
        BufferedReader br;
        br = new BufferedReader(new InputStreamReader(new FileInputStream("/home/student/workspace/
SPOSL/src/mot.txt")));//reading MOT
        while ((s = br.readLine()) != null) {
            StringTokenizer st = new StringTokenizer(s, " ", false); //convert line into tokens
            mot.add(new Tuple(st.nextToken(), st.nextToken(), st.nextToken(), st.nextToken())); //adding toke
n into list
        }
        br = new BufferedReader(new InputStreamReader(new FileInputStream("/home/student/workspace/
SPOSL/src/pot.txt")));//reading POT
```

```
        while ((s = br.readLine()) != null) {
            pot.add(s); //adding token into POT list
        }
        Collections.sort(pot); //sorting all the POT as per their index
    }
    //Pass 1 Srarts here

    static void pass1() throws Exception {
        //Read Input file
        BufferedReader input = new BufferedReader(new InputStreamReader(new FileInputStream("/home/
student/workspace/SPOSL/src/input.txt")));
        out_pass1 = new PrintWriter(new FileWriter("/home/student/workspace/SPOSL/src/output_pass1.txt"
), true); //writing to Output file pass1
        PrintWriter out_symtable = new PrintWriter(new FileWriter("/home/student/workspace/SPOSL/src/ou
t_symtable.txt"), true); //generate ST
        PrintWriter out_littable = new PrintWriter(new FileWriter("/home/student/workspace/SPOSL/src/out_li
ttable.txt"), true); //generate LT
        String s;
        while ((s = input.readLine()) != null) { //till end of file is reached
            StringTokenizer st = new StringTokenizer(s, " ", false); //convert line into tokens
            String s_arr[] = new String[st.countTokens()]; //initialized s_arr
            for (int i = 0; i < s_arr.length; i++) {
                s_arr[i] = st.nextToken(); //get all tokens into s_arr
            }
            if (searchPot1(s_arr) == false) { //if the token is not available in POT
                searchMot1(s_arr); //search in MOT
                out_pass1.println(s); //write to file pass1
            }
            lclist.add(lc); //add lc into lc list
        }
        int j; //to be used
        String output = new String(); //to be used to print on console
        System.out.println("Symbol Table:");
        System.out.println("Symbol    Value  Length   R/A");
        for (SymTuple i : symtable) { //traverse all symbols from symbol table
            output = i.symbol; //store in output
            for (j = i.symbol.length(); j < 10; j++) { //show symbols
                output += " ";
            }
            output += i.value;
            for (j = new Integer(i.value).toString().length(); j < 7; j++) { //show values
                output += " ";
            }
            output += i.length + "        " + i.ra; //instruction length and relative or absolute
            System.out.println(output);
            out_symtable.println(output);
        }
        System.out.println("\nLiteral Table:"); //printing literal table
        System.out.println("Literal   Value  Length   R/A");
        for (LitTuple i : littable) { //traverse the literal tuple to print
            output = i.literal;
            for (j = i.literal.length(); j < 10; j++) {
                output += " ";
            }
            output += i.value;
```

```
        for (j = new Integer(i.value).toString().length(); j < 7; j++) {
            output += " ";
        }
        output += i.length + "        " + i.ra;
        System.out.println(output);
        out_littable.println(output);
    }
}

static boolean searchPot1(String[] s) {
    int i = 0; //to be used
    int l = 0; //to be used
    int potval = 0; //to be used

    if (s.length == 3) {
        i = 1;
    }
    s = tokenizeOperands(s); //tokenize all the operands given by s_arr

    if (s[i].equalsIgnoreCase("DS") || s[i].equalsIgnoreCase("DC")) {
        potval = 1; //if DC or DS
    }
    if (s[i].equalsIgnoreCase("EQU")) {
        potval = 2; //if EQU
    }
    if (s[i].equalsIgnoreCase("START")) {
        potval = 3; //if START
    }
    if (s[i].equalsIgnoreCase("LTORG")) {
        potval = 4; //if LTORG
    }
    if (s[i].equalsIgnoreCase("END")) {
        potval = 5; //if END
    }

    switch (potval) { //doing actions as per input from POT
        case 1:
            // DS or DC statement
            String x = s[i + 1]; //point to next token after DC or DS
            int index = x.indexOf("F"); //get the index position of F
            if (i == 1) {
                symtable.add(new SymTuple(s[0], lc, 4, "R"));
            }
            if (index != 0) {
                // Ends with F
                l = Integer.parseInt(x.substring(0, x.length() - 1));
                l *= 4;
            } else {
                // Starts with F
                for (int j = i + 1; j < s.length; j++) {
                    l += 4;
                }
            }
            lc += l; //update LC
            return true;
```

```
            case 2:
                // EQU statement
                if (!s[2].equals("*")) { //check if there is no pointer
                    symtable.add(new SymTuple(s[0], Integer.parseInt(s[2]), 1, "A")); //add absolute address in S
T
                } else {
                    symtable.add(new SymTuple(s[0], lc, 1, "R")); //else add Relative address in ST
                }
                return true;

            case 3:
                // START statement
                symtable.add(new SymTuple(s[0], Integer.parseInt(s[2]), 1, "R"));  //add program name in ST
                return true;

            case 4:
                // LTORG statement
                ltorg(false); //call to LTORG method
                return true;

            case 5:
                // END statement
                ltorg(true); //call to LTORG method
                return true;
        }
        return false;
    }

    static void searchMot1(String[] s) {
        Tuple t = new Tuple(); //MOT object
        int i = 0;
        if (s.length == 3) { //check if 3 tokens
            i = 1; //keep i=1
        }
        s = tokenizeOperands(s); //again tokenize the operands
        for (int j = i + 1; j < s.length; j++) {
            if (s[j].startsWith("=")) { //check if literal
                littable.add(new LitTuple(s[j].substring(1, s[j].length()), -1, 4, "R")); //add into LT
            }
        }
        if ((i == 1) && (!s[0].equalsIgnoreCase("END"))) { //if 3 tokens in a line and not an END statement
            symtable.add(new SymTuple(s[0], lc, 4, "R")); //add entry to symbol table
        }
        for (Tuple x : mot) { //traverse all MOTs
            if (s[i].equals(x.mnemonic)) { //if mnemonic is found
                t = x; //store all mnemonics in t
                break;
            }
        }
        lc += t.length; //update location counter
    }

    static String[] tokenizeOperands(String[] s) {
        List<String> temp = new LinkedList<>(); //to be used
```

```
        for (int j = 0; j < s.length - 1; j++) { //adding all tokens into temp
            temp.add(s[j]);
        }
        StringTokenizer st = new StringTokenizer(s[s.length - 1], " ,", false); //convert line into tokens
        while (st.hasMoreTokens()) {
            temp.add(st.nextToken()); //adding all tokens
        }
        s = temp.toArray(new String[0]); //convert linked list to array list
        return s;
    }

    static void ltorg(boolean isEnd) { //adding literals in LT
        Iterator<LitTuple> itr = littable.iterator();  //Iterator used to store literal objects
        LitTuple lt = new LitTuple(); //created object
        boolean isBroken = false; //to be used
        while (itr.hasNext()) { //check the iterators
            lt = itr.next(); //check the literals
            if (lt.value == -1) {
                isBroken = true;
                break;
            }
        }
        if (!isBroken) { //if LTORG occurs
            return;
        }
        if (!isEnd) { //if not END
            while (lc % 8 != 0) {
                lc++; //reach up to END statement
            }
        }
        lt.value = lc;
        lc += 4;
        while (itr.hasNext()) {
            lt = itr.next(); //adding literals to lt
            lt.value = lc; //update LT Value
            lc += 4; //update location counter
        }
    }
}
```