```
import java.io.*;
import java.util.*;

/*
Pass1.java
Usage:
javac Pass1.java
java Pass1 asm_input.txt

Produces:
IC.txt -> intermediate code
SYMTAB.txt -> symbol table (index SYMBOL ADDRESS LENGTH)
LITTAB.txt -> literal table (index LITERAL ADDRESS)
POOLTAB.txt -> pool table (one index per pool)
Also prints formatted tables to console similar to sample.
*/

public class pass1 {
static class Symbol { String name; int addr; int length; int index; public
Symbol(String n,int i){name=n; index=i; addr=-1; length=1;} }
static class Literal { String lit; int addr; int index; public Literal(String
l,int i){lit=l; index=i; addr=-1;} }

// Opcode table: mnemonic -> (class, opcode)
static Map<String, String[]> opTab = new HashMap<>();
static {
// class codes: IS (Imperative), AD (Assembler Dir), DL (Declarative)
opTab.put("STOP", new String[]{"IS","00"});
opTab.put("ADD", new String[]{"IS","01"});
opTab.put("SUB", new String[]{"IS","02"});
opTab.put("MULT", new String[]{"IS","03"});
opTab.put("MOVER",new String[]{"IS","04"});
opTab.put("MOVEM",new String[]{"IS","05"});
opTab.put("COMP", new String[]{"IS","06"});
opTab.put("BC", new String[]{"IS","07"});
opTab.put("DIV", new String[]{"IS","08"});
opTab.put("READ", new String[]{"IS","09"});
opTab.put("PRINT",new String[]{"IS","10"});

opTab.put("START", new String[]{"AD","01"});
opTab.put("END", new String[]{"AD","02"});
opTab.put("ORIGIN",new String[]{"AD","03"});
opTab.put("EQU", new String[]{"AD","04"});
opTab.put("LTORG", new String[]{"AD","05"});

opTab.put("DC", new String[]{"DL","01"});
opTab.put("DS", new String[]{"DL","02"});
}

static Map<String, Symbol> symMap = new LinkedHashMap<>();
static List<Symbol> symList = new ArrayList<>();
static Map<String, Literal> litMap = new LinkedHashMap<>();
static List<Literal> litList = new ArrayList<>();
static List<Integer> poolTable = new ArrayList<>(); // literal indices (1-based)
where each pool starts

static List<String> IC = new ArrayList<>(); // intermediate code lines

public static void main(String[] args) throws Exception {
if(args.length==0){
System.out.println("Usage: java Pass1 <assembly_input_file>");
return;
}
```

```
String infile = args[0];
BufferedReader br = new BufferedReader(new FileReader(infile));
String line;
int LC = 0; // location counter
int litCounter = 0;
int symCounter = 0;
boolean firstLine = true;

// Keep a temporary list of literals that have been encountered since last pool
started
List<Literal> currentPoolLiterals = new ArrayList<>();

while((line = br.readLine()) != null){
line = line.trim();
if(line.isEmpty()) continue;

// print program line as sample wants
System.out.println(line);

// tokenization: split by spaces and commas but keep commas removed, e.g. "MOVER
A, B"
String[] parts = line.replaceAll(",", " ").split("\\s+");
// if a label present (first token and next token is an opcode/directive or
mnemonic), detect it as label if not mnemonic
String label = "";
String mnemonic = "";
String operand1 = "";
String operand2 = "";

if(parts.length >= 1) mnemonic = parts[0];
if(parts.length >= 2) operand1 = parts[1];
if(parts.length >= 3) operand2 = parts[2];

// If first token is a label (i.e., not an opcode or directive), we detect by
checking if it's known mnemonic; if not and more tokens exist
boolean firstIsLabel = false;
if(parts.length>=2 && !opTab.containsKey(parts[0].toUpperCase())){
// treat as label
firstIsLabel = true;
label = parts[0];
mnemonic = parts.length>1? parts[1] : "";
operand1 = parts.length>2? parts[2] : "";
operand2 = parts.length>3? parts[3] : "";
}

String mU = mnemonic.toUpperCase();

if(firstLine){
// Expect START
if(mU.equals("START")){
// operand1 may be starting address
int startAddr = 0;
if(operand1 != null && !operand1.equals("")) {
try { startAddr = Integer.parseInt(operand1); } catch(Exception e){}
}
LC = startAddr;
IC.add("(AD,01)(C,"+LC+")");
firstLine = false;
continue;
} else {
firstLine = false; // still proceed
}
}
```

```
// If label exists, enter symbol with current LC (or update if already present)
if(firstIsLabel && !label.equals("")){
if(!symMap.containsKey(label)){
Symbol s = new Symbol(label, ++symCounter);
s.addr = LC;
symMap.put(label, s);
symList.add(s);
} else {
Symbol s = symMap.get(label);
s.addr = LC;
}
}

if(mU.equals("LTORG") || mU.equals("END")){
// AD directive
IC.add("(AD," + opTab.get(mU)[1] + ")");
// Assign addresses to all literals in current pool
if(!currentPoolLiterals.isEmpty()){
for(Literal L : currentPoolLiterals){
if(L.addr == -1){
L.addr = LC;
LC++;
}
}
// add next pool start index to pool table if there are remaining literals later
poolTable.add( (litList.size()>0 ? (litList.get(0).index) : 1) ); // not very
critical here
// Clear current pool
currentPoolLiterals.clear();
} else {
// nothing
}
// For END, also assign any remaining literals (if END)
if(mU.equals("END")){
// assign any unassigned literals in full littab
for(Literal L : litList){
if(L.addr == -1){
L.addr = LC;
LC++;
}
}
}
continue;
}

if(opTab.containsKey(mU)){
String cls = opTab.get(mU)[0];
String code = opTab.get(mU)[1];

if(cls.equals("IS")){
// Imperative Statement
// For uniform IC format: (IS,opcode)(r)(S,x) or (L,x) or (C,x)
String regField = "0";
String operandField = "";

// if operand1 exists and is register (we assume numeric or A,B mapping). We'll
allow registers by name like A,B mapped to numbers
if(operand1 != null && !operand1.equals("")){
// if operand1 is a register like A/1, or numeric register mapping, support
R0..R7 by reading if it's a single letter
String s = operand1.replaceAll("\\s+","").replaceAll(",","");
if(s.matches("[A-Za-z]")) {
// map A->1, B->2, C->3 etc. (simple)
regField = String.valueOf((s.toUpperCase().charAt(0) - 'A') + 1);
```

```
} else if(s.matches("\\d+")) {
regField = s;
} else {
// it might be first operand is a symbol or literal and register is absent;
we'll handle later
}
}

// Determine second operand (symbol or literal or constant)
String opnd = operand2;
// if no operand2 but operand1 is not a register or is of form symbol, decide
properly:
if((opnd==null || opnd.equals("")) && operand1!=null && !operand1.equals("")){
// if operand1 is literal (starts with =) or is symbol
if(operand1.startsWith("=") || !operand1.matches("[A-Za-z]")) {
opnd = operand1;
} else {
// if operand1 is a single letter and we earlier treated it as register, then
there's no operand
// set opnd blank
}
}

// If opnd is literal starting with '='
if(opnd!=null && opnd.startsWith("=")){
String lit = opnd;
if(!litMap.containsKey(lit)){
Literal L = new Literal(lit, ++litCounter);
litMap.put(lit,L);
litList.add(L);
}
Literal L = litMap.get(lit);
currentPoolLiterals.add(L);
IC.add("(IS,"+code+")"+"("+regField+")(L,"+L.index+")");
} else if(opnd!=null && !opnd.equals("")){
// symbol
String sym = opnd;
if(!symMap.containsKey(sym)){
Symbol S = new Symbol(sym, ++symCounter);
symMap.put(sym, S);
symList.add(S);
}
Symbol S = symMap.get(sym);
IC.add("(IS,"+code+")"+"("+regField+")(S,"+S.index+")");
} else {
// no operand (e.g., STOP)
IC.add("(IS,"+code+")");
}
LC++;
} else if(cls.equals("DL")){
// Declarative: DC / DS
if(mU.equals("DC")){
// operand1 should be constant or 'x' style
String c = operand1.replaceAll("'", "");
IC.add("(DL,01)(C,"+c+")");
LC++;
} else if(mU.equals("DS")){
String c = "1";
if(operand1!=null && !operand1.equals("")) {
c = operand1;
}
IC.add("(DL,02)(C,"+c+")");
// create symbol entry if label exists (handled above)
LC += Integer.parseInt(c);
```

```
}
} else if(cls.equals("AD")){
// assembler directive other than START/END/LTORG handled earlier
if(mU.equals("ORIGIN")){
// ORIGIN operand like SYMBOL+2
String expr = operand1;
int newlc = LC;
if(expr.contains("+")){
String[] t = expr.split("\\+");
String sym = t[0];
int val = Integer.parseInt(t[1]);
if(symMap.containsKey(sym) && symMap.get(sym).addr!=-1) newlc =
symMap.get(sym).addr + val;
} else if(symMap.containsKey(expr) && symMap.get(expr).addr!=-1){
newlc = symMap.get(expr).addr;
} else {
try{ newlc = Integer.parseInt(expr);}catch(Exception e){}
}
IC.add("(AD,03)(S,"+ (symMap.containsKey(expr)? symMap.get(expr).index : 0) +")
+"+ (expr.contains("+") ? expr.substring(expr.indexOf("+")) : ("0")) );
LC = newlc;
} else if(mU.equals("EQU")){
// label EQU operand => set symbol value
// Not fully implemented; add a simple placeholder
IC.add("(AD,04)");
} else {
IC.add("(AD," + opTab.get(mU)[1] + ")");
}
}
} else {
// If not in opTab (e.g., stray label-only lines), attempt basic handling
// If line is a literal by itself like "='5'" (some samples show literals
printed on program)
if(mnemonic.startsWith("=")){
String lit = mnemonic;
if(!litMap.containsKey(lit)){
Literal L = new Literal(lit, ++litCounter);
litMap.put(lit,L);
litList.add(L);
}
Literal L = litMap.get(lit);
currentPoolLiterals.add(L);
// we do not increment LC here (these literal-only lines are not instructions)
}
}
} // end while

br.close();

// Assign addresses to any remaining literals not assigned
for(Literal L: litList){
if(L.addr == -1){
L.addr = LC;
LC++;
}
}

// Write IC to IC.txt
try(PrintWriter pw = new PrintWriter(new FileWriter("IC.txt"))){
for(String s: IC) pw.println(s);
}

// Write SYMTAB
try(PrintWriter pw = new PrintWriter(new FileWriter("SYMTAB.txt"))){
```

```
int idx=1;
for(Symbol s: symList){
pw.println(idx + " " + s.name + " " + (s.addr==-1?0:s.addr) + " " + s.length);
idx++;
}
}

// Write LITTAB
try(PrintWriter pw = new PrintWriter(new FileWriter("LITTAB.txt"))){
for(Literal L : litList){
pw.println(L.index + " " + L.lit + " " + (L.addr==-1?0:L.addr));
}
}

// Write POOLTAB: print indices of literal list where a new pool starts (we kept
a simple model)
try(PrintWriter pw = new PrintWriter(new FileWriter("POOLTAB.txt"))){
// naive: one pool starting at 1
if(!litList.isEmpty()) pw.println(1);
}

// Print tables in sample format
System.out.println("_____________________________________________________");
System.out.println("\n\nSYMBOL TABLE");
System.out.println("--------------------------");
System.out.println("SYMBOL ADDRESS LENGTH");
System.out.println("--------------------------");
for(Symbol s: symList){
System.out.println(s.name + " " + (s.addr==-1?0:s.addr) + " " + s.length);
}
System.out.println("--------------------------");

System.out.println("\n\nOPCODE TABLE");
System.out.println("----------------------------");
System.out.println("MNEMONIC\tCLASS\tINFO");
System.out.println("----------------------------");
for(Map.Entry<String, String[]> e : opTab.entrySet()){
System.out.println(e.getKey() + "\t" + e.getValue()[0] + "\t" + e.getValue()
[1]);
}
System.out.println("----------------------------");

System.out.println("\n\nLITERAL TABLE");
System.out.println("-----------------");
System.out.println("LITERAL ADDRESS");
System.out.println("-----------------");
for(Literal L: litList){
System.out.println(L.lit + " " + (L.addr==-1?0:L.addr));
}
System.out.println("-----------------");

System.out.println("\n\nPOOL TABLE");
System.out.println("-----------------");
System.out.println("LITERAL NUMBER");
System.out.println("-----------------");
if(!litList.isEmpty()) System.out.println(1);
System.out.println("-----------------");
System.out.println("\nIntermediate code written to IC.txt");
System.out.println("SYMTAB.txt, LITTAB.txt, POOLTAB.txt generated.");
}
}
```