**(1)**

**(A)1.** Write a program to generate Symbol table of a two-pass Assembler for the given Assembly language source code.

|  |
| --- |
| **INPUT/CODE**  START 100  READ A  READ B  LOOP MOVER AREG, A  MOVER BREG, B  COMP BREG, =’2’  BC GT, LOOP  BACK SUB AREG, B  COMP AREG, =’5’  BC LT, BACK  STOP  A DS 1  B DS 1  END |

import java.util.\*;

public class Main {

public static void main(String[] args) {

String[] code = {

"START 100",

"READ A",

"READ B",

"LOOP MOVER AREG, A",

" MOVER BREG, B",

" COMP BREG, ='2'",

" BC GT, LOOP",

"BACK SUB AREG, B",

" COMP AREG, ='5'",

" BC LT, BACK",

" STOP",

"A DS 1",

"B DS 1",

" END"

};

Map<String, Integer> symbolTable = new LinkedHashMap<>();

int locationCounter = 0;

int startAddress = 0;

for (String line : code) {

line = line.trim();

if (line.isEmpty()) continue;

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

// Handle START directive

if (tokens[0].equals("START")) {

startAddress = Integer.parseInt(tokens[1]);

locationCounter = startAddress;

continue;

}

// Detect if there's a label at the start

boolean hasLabel = false;

String label = "";

// Labels like LOOP, BACK, A, B

if (!Arrays.asList("READ", "MOVER", "COMP", "BC", "SUB", "STOP", "DS", "END").contains(tokens[0])) {

hasLabel = true;

label = tokens[0];

}

if (hasLabel) {

// Insert label into symbol table

symbolTable.put(label, locationCounter);

if (tokens.length > 1 && tokens[1].equals("DS")) {

locationCounter += Integer.parseInt(tokens[2]); // Reserve space

} else {

locationCounter++; // Normal instruction

}

} else if (tokens[0].equals("DS")) {

locationCounter += Integer.parseInt(tokens[1]);

} else if (tokens[0].equals("END")) {

break;

} else {

locationCounter++; // Instruction without label

}

}

// Print Symbol Table

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

System.out.println("Label\tAddress");

for (Map.Entry<String, Integer> entry : symbolTable.entrySet()) {

System.out.println(entry.getKey() + "\t" + entry.getValue());

}

}

}

**++++++++++++++++++++++++++++++++++++**

**(2)**

**(B)1.**Write a program to generate Symbol table of a two-pass Assembler for the given Assembly language source code.

|  |
| --- |
| **INPUT/CODE**  START 150  READ D  READ E  LOOP MOVER AREG, D  MOVER BREG, E  COMP BREG, =’20’  BC GT, LOOP  BACK SUB AREG, E  COMP AREG, =’50’  BC LT, BACK  STOP  D DS 1  E DS 1  END |

import java.util.\*;

public class Main {

public static void main(String[] args) {

String[] code = {

"START 150",

"READ D",

"READ E",

"LOOP MOVER AREG, D",

" MOVER BREG, E",

" COMP BREG, ='20'",

" BC GT, LOOP",

"BACK SUB AREG, E",

" COMP AREG, ='50'",

" BC LT, BACK",

" STOP",

"D DS 1",

"E DS 1",

" END"

};

Map<String, Integer> symbolTable = new LinkedHashMap<>();

int locationCounter = 0;

for (String line : code) {

line = line.trim();

if (line.isEmpty()) continue;

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

// Handle START directive

if (tokens[0].equals("START")) {

locationCounter = Integer.parseInt(tokens[1]);

continue;

}

// Check if a label exists

boolean hasLabel = false;

String label = "";

if (!Arrays.asList("READ", "MOVER", "COMP", "BC", "SUB", "STOP", "DS", "END").contains(tokens[0])) {

hasLabel = true;

label = tokens[0];

}

if (hasLabel) {

symbolTable.put(label, locationCounter);

if (tokens.length > 1 && tokens[1].equals("DS")) {

locationCounter += Integer.parseInt(tokens[2]); // DS reserves memory

} else {

locationCounter++; // normal instruction

}

} else if (tokens[0].equals("DS")) {

locationCounter += Integer.parseInt(tokens[1]);

} else if (tokens[0].equals("END")) {

break;

} else {

locationCounter++; // for READ, MOVER, COMP, etc.

}

}

// Display the Symbol Table

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

System.out.println("Label\tAddress");

for (Map.Entry<String, Integer> entry : symbolTable.entrySet()) {

System.out.println(entry.getKey() + "\t" + entry.getValue());

}

}

}

**++++++++++++++++++++++++++++++++++++**

**(3)**

**(C) 1.** Write a program to generate Symbol table of a two-pass Assembler for the given Assembly language source code.

|  |
| --- |
| **INPUT/CODE**  START 180  READ M  READ N  LOOP MOVER AREG, M  MOVER BREG, N  COMP BREG, =’200’  BC GT, LOOP  BACK SUB AREG, M  COMP AREG, =’500’  BC LT, BACK  STOP  M DS 1  N DS 1  END |

import java.util.\*;

public class Main {

public static void main(String[] args) {

String[] code = {

"START 180",

"READ M",

"READ N",

"LOOP MOVER AREG, M",

" MOVER BREG, N",

" COMP BREG, ='200'",

" BC GT, LOOP",

"BACK SUB AREG, M",

" COMP AREG, ='500'",

" BC LT, BACK",

" STOP",

"M DS 1",

"N DS 1",

" END"

};

Map<String, Integer> symbolTable = new LinkedHashMap<>();

int locationCounter = 0;

for (String line : code) {

line = line.trim();

if (line.isEmpty()) continue;

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

// Handle START directive

if (tokens[0].equals("START")) {

locationCounter = Integer.parseInt(tokens[1]);

continue;

}

// Check if a label exists

boolean hasLabel = false;

String label = "";

if (!Arrays.asList("READ", "MOVER", "COMP", "BC", "SUB", "STOP", "DS", "END").contains(tokens[0])) {

hasLabel = true;

label = tokens[0];

}

if (hasLabel) {

symbolTable.put(label, locationCounter);

if (tokens.length > 1 && tokens[1].equals("DS")) {

locationCounter += Integer.parseInt(tokens[2]); // Reserve memory

} else {

locationCounter++; // Instruction

}

} else if (tokens[0].equals("DS")) {

locationCounter += Integer.parseInt(tokens[1]);

} else if (tokens[0].equals("END")) {

break;

} else {

locationCounter++; // Instruction without label

}

}

// Display the Symbol Table

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

System.out.println("Label\tAddress");

for (Map.Entry<String, Integer> entry : symbolTable.entrySet()) {

System.out.println(entry.getKey() + "\t" + entry.getValue());

}

}

}

**++++++++++++++++++++++++++++++++++++**

**(4)**

**(A)2.** Write a program to generate Literal table of a two-pass Assembler for the given Assembly language source code.

|  |
| --- |
| **INPUT/CODE**  START 100  READ A  READ B  MOVER AREG, =’50'  MOVER BREG, =’60’  ADD AREG, BREG  LOOP MOVER CREG, A  ADD CREG, ='10'  COMP CREG, B  BC LT, LOOP  NEXT SUB AREG, ='10'  COMP AREG, B  BC GT, NEXT  STOP  A DS 1  B DS 1  END |

import java.util.\*;

public class Main {

public static void main(String[] args) {

String[] code = {

"START 100",

"READ A",

"READ B",

"MOVER AREG, ='50'",

"MOVER BREG, ='60'",

"ADD AREG, BREG",

"LOOP MOVER CREG, A",

"ADD CREG, ='10'",

"COMP CREG, B",

"BC LT, LOOP",

"NEXT SUB AREG, ='10'",

"COMP AREG, B",

"BC GT, NEXT",

"STOP",

"A DS 1",

"B DS 1",

"END"

};

Map<String, Integer> literalTable = new LinkedHashMap<>();

int locationCounter = 0;

boolean foundStart = false;

for (String line : code) {

line = line.trim();

if (line.isEmpty()) continue;

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

// Handle START

if (tokens[0].equals("START")) {

locationCounter = Integer.parseInt(tokens[1]);

foundStart = true;

continue;

}

if (!foundStart) continue;

// Search for literals in line

for (String token : tokens) {

if (token.matches("=.\*")) {

// Remove trailing characters like ',' or ')'

token = token.replaceAll("[^=']\*('.\*')", "$1"); // keep ='value'

if (!literalTable.containsKey(token)) {

literalTable.put(token, -1); // placeholder address

}

}

}

// Handle DS instruction which reserves memory

if (tokens.length >= 2 && tokens[1].equals("DS")) {

locationCounter += Integer.parseInt(tokens[2]);

} else if (!tokens[0].equals("END")) {

locationCounter++;

}

}

// Assign addresses to literals after all instructions

int literalStartAddress = locationCounter;

int address = literalStartAddress;

for (String literal : literalTable.keySet()) {

literalTable.put(literal, address++);

}

// Print Literal Table

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

System.out.println("Literal\tAddress");

for (Map.Entry<String, Integer> entry : literalTable.entrySet()) {

System.out.println(entry.getKey() + "\t" + entry.getValue());

}

}

}

**++++++++++++++++++++++++++++++++++++**

**(5)**

**(B)2.** Write a program to generate Literal table of a two-pass Assembler for the given Assembly language source code.

|  |
| --- |
| START 200  READ X  READ Y  MOVER AREG, =’5'  MOVER BREG, =’6’  ADD AREG, BREG  LOOP MOVER CREG, X  ADD CREG, ='1'  COMP CREG, Y  BC LT, LOOP  NEXT SUB AREG, ='1'  COMP AREG, Y  BC GT, NEXT  STOP  X DS 1  Y DS 1  END |

import java.util.\*;

public class Main {

public static void main(String[] args) {

String[] code = {

"START 200",

"READ X",

"READ Y",

"MOVER AREG, ='5'",

"MOVER BREG, ='6'",

"ADD AREG, BREG",

"LOOP MOVER CREG, X",

"ADD CREG, ='1'",

"COMP CREG, Y",

"BC LT, LOOP",

"NEXT SUB AREG, ='1'",

"COMP AREG, Y",

"BC GT, NEXT",

"STOP",

"X DS 1",

"Y DS 1",

"END"

};

Map<String, Integer> literalTable = new LinkedHashMap<>();

int locationCounter = 0;

boolean startFound = false;

for (String line : code) {

line = line.trim();

if (line.isEmpty()) continue;

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

// Handle START directive

if (tokens[0].equals("START")) {

locationCounter = Integer.parseInt(tokens[1]);

startFound = true;

continue;

}

if (!startFound) continue;

// Check for literals in the line

for (String token : tokens) {

if (token.startsWith("='") && token.endsWith("'")) {

if (!literalTable.containsKey(token)) {

literalTable.put(token, -1); // Placeholder for address

}

}

}

// Handle DS directive (reserves space)

if (tokens.length >= 2 && tokens[1].equals("DS")) {

locationCounter += Integer.parseInt(tokens[2]);

} else if (!tokens[0].equals("END")) {

locationCounter++; // Normal instruction

}

}

// Assign addresses to literals after the last instruction

int address = locationCounter;

for (String literal : literalTable.keySet()) {

literalTable.put(literal, address++);

}

// Print Literal Table

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

System.out.println("Literal\tAddress");

for (Map.Entry<String, Integer> entry : literalTable.entrySet()) {

System.out.println(entry.getKey() + "\t" + entry.getValue());

}

}

}

**++++++++++++++++++++++++++++++++++++**

**(6)**

**(C)2.** Write a program to generate Literal table of a two-pass Assembler for the given Assembly language source code.

|  |
| --- |
| START 300  READ M  READ N  MOVER AREG, =’51'  MOVER BREG, =’61’  ADD AREG, BREG  LOOP MOVER CREG, M  ADD CREG, ='11'  COMP CREG, N  BC LT, LOOP  NEXT SUB AREG, ='11'  COMP AREG, N  BC GT, NEXT  STOP  M DS 1  N DS 1  END |

import java.util.\*;

public class LiteralTablePass1 {

public static void main(String[] args) {

String[] code = {

"START 300",

"READ M",

"READ N",

"MOVER AREG, ='51'",

"MOVER BREG, ='61'",

"ADD AREG, BREG",

"LOOP MOVER CREG, M",

"ADD CREG, ='11'",

"COMP CREG, N",

"BC LT, LOOP",

"NEXT SUB AREG, ='11'",

"COMP AREG, N",

"BC GT, NEXT",

"STOP",

"M DS 1",

"N DS 1",

"END"

};

Map<String, Integer> literalTable = new LinkedHashMap<>();

int locationCounter = 0;

boolean startFound = false;

for (String line : code) {

line = line.trim();

if (line.isEmpty()) continue;

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

// Handle START directive

if (tokens[0].equals("START")) {

locationCounter = Integer.parseInt(tokens[1]);

startFound = true;

continue;

}

if (!startFound) continue;

// Check for literals in the line

for (String token : tokens) {

if (token.matches("='[0-9]+'")) {

literalTable.putIfAbsent(token, -1); // Add literal if not already added

}

}

// Handle DS directive (reserves memory)

if (tokens.length >= 2 && tokens[1].equals("DS")) {

locationCounter += Integer.parseInt(tokens[2]);

} else if (!tokens[0].equals("END")) {

locationCounter++; // Instruction

}

}

// Assign addresses to literals after the last instruction

int address = locationCounter;

for (String literal : literalTable.keySet()) {

literalTable.put(literal, address++);

}

// Print the Literal Table

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

System.out.println("Literal\tAddress");

for (Map.Entry<String, Integer> entry : literalTable.entrySet()) {

System.out.println(entry.getKey() + "\t" + entry.getValue());

}

}

}

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

**(7)**

**(A)3**.Write a program to generate Pool table of a two-pass Assembler for the given Assembly language source code.

|  |
| --- |
| **INPUT/CODE**  START 100  READ A  MOVER AREG, ='1'  MOVEM AREG, B  MOVER BREG, ='6'  ADD AREG, BREG  COMP AREG, A  BC GT, LAST  LTORG  NEXT SUB AREG, ='1'  MOVER CREG, B  ADD CREG, ='8'  MOVEM CREG, B  PRINT B  LAST STOP  A DS 1  B DS 1  END |

import java.util.\*;

public class Main {

public static void main(String[] args) {

String[] code = {

"START 100",

"READ A",

"MOVER AREG, ='1'",

"MOVEM AREG, B",

"MOVER BREG, ='6'",

"ADD AREG, BREG",

"COMP AREG, A",

"BC GT, LAST",

"LTORG",

"NEXT SUB AREG, ='1'",

"MOVER CREG, B",

"ADD CREG, ='8'",

"MOVEM CREG, B",

"PRINT B",

"LAST STOP",

"A DS 1",

"B DS 1",

"END"

};

List<String> literalTable = new ArrayList<>();

List<Integer> poolTable = new ArrayList<>();

List<String> pendingLiterals = new ArrayList<>();

int locationCounter = 0;

boolean startFound = false;

for (String line : code) {

line = line.trim();

if (line.isEmpty()) continue;

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

// Handle START directive

if (tokens[0].equals("START")) {

locationCounter = Integer.parseInt(tokens[1]);

startFound = true;

continue;

}

if (!startFound) continue;

// Look for literals

for (String token : tokens) {

if (token.matches("='\\d+'")) {

if (!literalTable.contains(token) && !pendingLiterals.contains(token)) {

pendingLiterals.add(token);

}

}

}

// Handle LTORG or END (trigger pool)

if (tokens[0].equals("LTORG") || tokens[0].equals("END")) {

if (!pendingLiterals.isEmpty()) {

poolTable.add(literalTable.size() + 1); // 1-based index

literalTable.addAll(pendingLiterals);

pendingLiterals.clear();

}

}

// Increment location counter for normal instructions

if (!tokens[0].equals("LTORG") && !tokens[0].equals("END")) {

locationCounter++;

}

}

// 🧾 Output Literal Table

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

System.out.println("Index\tLiteral");

for (int i = 0; i < literalTable.size(); i++) {

System.out.println((i + 1) + "\t" + literalTable.get(i));

}

// 🧾 Output Pool Table

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

System.out.println("Index\tLiteral Start Index");

for (int i = 0; i < poolTable.size(); i++) {

System.out.println((i + 1) + "\t" + poolTable.get(i));

}

}

}

**++++++++++++++++++++++++++++++++++++**

**(8)**

**(B)3**.Write a program to generate Pool table of a two-pass Assembler for the given Assembly language source code.

|  |
| --- |
| **INPUT/CODE**  START 200  READ X  MOVER AREG, ='10'  MOVEM AREG, Y  MOVER BREG, ='60'  ADD AREG, BREG  COMP AREG, X  BC GT, LAST  LTORG  NEXT SUB AREG, ='10'  MOVER CREG, Y  ADD CREG, ='80'  MOVEM CREG, Y  PRINT B  LAST STOP  X DS 1  Y DS 1  END |

import java.util.\*;

public class Main {

public static void main(String[] args) {

String[] code = {

"START 200",

"READ X",

"MOVER AREG, ='10'",

"MOVEM AREG, Y",

"MOVER BREG, ='60'",

"ADD AREG, BREG",

"COMP AREG, X",

"BC GT, LAST",

"LTORG",

"NEXT SUB AREG, ='10'",

"MOVER CREG, Y",

"ADD CREG, ='80'",

"MOVEM CREG, Y",

"PRINT B",

"LAST STOP",

"X DS 1",

"Y DS 1",

"END"

};

List<String> literalTable = new ArrayList<>();

List<Integer> poolTable = new ArrayList<>();

List<String> pendingLiterals = new ArrayList<>();

int locationCounter = 0;

boolean started = false;

for (String line : code) {

line = line.trim();

if (line.isEmpty()) continue;

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

// Handle START directive

if (tokens[0].equals("START")) {

locationCounter = Integer.parseInt(tokens[1]);

started = true;

continue;

}

if (!started) continue;

// Collect literals

for (String token : tokens) {

if (token.matches("='\\d+'")) {

if (!literalTable.contains(token) && !pendingLiterals.contains(token)) {

pendingLiterals.add(token);

}

}

}

// Handle LTORG or END - generate pool

if (tokens[0].equals("LTORG") || tokens[0].equals("END")) {

if (!pendingLiterals.isEmpty()) {

poolTable.add(literalTable.size() + 1); // 1-based index

literalTable.addAll(pendingLiterals);

pendingLiterals.clear();

}

}

// Increment LC for each instruction

if (!tokens[0].equals("LTORG") && !tokens[0].equals("END")) {

locationCounter++;

}

}

// 🧾 Output Literal Table

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

System.out.println("Index\tLiteral");

for (int i = 0; i < literalTable.size(); i++) {

System.out.println((i + 1) + "\t" + literalTable.get(i));

}

// 🧾 Output Pool Table

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

System.out.println("Index\tLiteral Start Index");

for (int i = 0; i < poolTable.size(); i++) {

System.out.println((i + 1) + "\t" + poolTable.get(i));

}

}

}

**++++++++++++++++++++++++++++++++++++**

**(9)**

**(C)3**.Write a program to generate Pool table of a two-pass Assembler for the given Assembly language source code.

|  |
| --- |
| **INPUT/CODE**  START 300  READ M  MOVER AREG, ='11'  MOVEM AREG, N  MOVER BREG, ='61'  ADD AREG, BREG  COMP AREG, M  BC GT, LAST  LTORG  NEXT SUB AREG, ='11'  MOVER CREG, N  ADD CREG, ='81'  MOVEM CREG, N  PRINT Y  LAST STOP  M DS 1  N DS 1  END |

import java.util.\*;

public class Main {

public static void main(String[] args) {

String[] code = {

"START 300",

"READ M",

"MOVER AREG, ='11'",

"MOVEM AREG, N",

"MOVER BREG, ='61'",

"ADD AREG, BREG",

"COMP AREG, M",

"BC GT, LAST",

"LTORG",

"NEXT SUB AREG, ='11'",

"MOVER CREG, N",

"ADD CREG, ='81'",

"MOVEM CREG, N",

"PRINT Y",

"LAST STOP",

"M DS 1",

"N DS 1",

"END"

};

List<String> literalTable = new ArrayList<>();

List<Integer> poolTable = new ArrayList<>();

List<String> pendingLiterals = new ArrayList<>();

int locationCounter = 0;

boolean started = false;

for (String line : code) {

line = line.trim();

if (line.isEmpty()) continue;

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

// Handle START directive

if (tokens[0].equals("START")) {

locationCounter = Integer.parseInt(tokens[1]);

started = true;

continue;

}

if (!started) continue;

// Collect literals

for (String token : tokens) {

if (token.matches("='\\d+'")) {

if (!literalTable.contains(token) && !pendingLiterals.contains(token)) {

pendingLiterals.add(token);

}

}

}

// Handle LTORG or END - generate literal pool

if (tokens[0].equals("LTORG") || tokens[0].equals("END")) {

if (!pendingLiterals.isEmpty()) {

poolTable.add(literalTable.size() + 1); // 1-based index

literalTable.addAll(pendingLiterals);

pendingLiterals.clear();

}

}

// Increment LC for each instruction (except directives)

if (!tokens[0].equals("LTORG") && !tokens[0].equals("END")) {

locationCounter++;

}

}

// 🧾 Output Literal Table

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

System.out.println("Index\tLiteral");

for (int i = 0; i < literalTable.size(); i++) {

System.out.println((i + 1) + "\t" + literalTable.get(i));

}

// 🧾 Output Pool Table

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

System.out.println("Index\tLiteral Start Index");

for (int i = 0; i < poolTable.size(); i++) {

System.out.println((i + 1) + "\t" + poolTable.get(i));

}

}

}

**++++++++++++++++++++++++++++++++++++**

**(10)**

**(A)4.**Write a program to generate Intermediate code of a two-pass Assembler for the given Assembly language source code.

|  |
| --- |
| **INPUT/CODE**  START 100  READ A  READ B  MOVER AREG, A  SUB AREG, B  STOP  A DS 1  B DS 1  END |

import java.util.\*;

public class Main {

public static void main(String[] args) {

String[] code = {

"START 100",

"READ A",

"READ B",

"MOVER AREG, A",

"SUB AREG, B",

"STOP",

"A DS 1",

"B DS 1",

"END"

};

Map<String, Integer> symbolTable = new LinkedHashMap<>();

List<String> intermediateCode = new ArrayList<>();

int locCounter = 0;

boolean started = false;

// === FIRST PASS ===

for (String line : code) {

line = line.trim();

if (line.isEmpty()) continue;

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

if (tokens[0].equals("START")) {

locCounter = Integer.parseInt(tokens[1]);

started = true;

continue;

}

if (!started) continue;

if (tokens.length == 3 && tokens[1].equals("DS")) {

symbolTable.put(tokens[0], locCounter);

locCounter += Integer.parseInt(tokens[2]);

continue;

}

if (!Arrays.asList("READ", "MOVER", "SUB", "STOP", "END").contains(tokens[0])) {

symbolTable.put(tokens[0], locCounter);

}

if (!tokens[0].equals("END")) {

locCounter++;

}

}

// === SECOND PASS ===

locCounter = 0;

started = false;

for (String line : code) {

line = line.trim();

if (line.isEmpty()) continue;

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

if (tokens.length == 0) continue;

if (tokens[0].equals("START")) {

locCounter = Integer.parseInt(tokens[1]);

intermediateCode.add(locCounter + "\t(AD,01) (C," + locCounter + ")");

started = true;

continue;

}

if (!started) continue;

String output = locCounter + "\t";

String instr = tokens[0];

switch (instr) {

case "READ":

if (tokens.length >= 2) {

output += "(IS,09) (S," + getSymbolIndex(tokens[1], symbolTable) + ")";

intermediateCode.add(output);

locCounter++;

}

break;

case "MOVER":

case "SUB":

if (tokens.length >= 3) {

String reg = tokens[1].replace(",", "");

String sym = tokens[2];

String opcode = instr.equals("MOVER") ? "04" : "05";

output += "(IS," + opcode + ") (" + reg + ") (S," + getSymbolIndex(sym, symbolTable) + ")";

intermediateCode.add(output);

locCounter++;

}

break;

case "STOP":

output += "(IS,00)";

intermediateCode.add(output);

locCounter++;

break;

case "END":

intermediateCode.add(locCounter + "\t(AD,02)");

break;

default:

// ✅ Add DS to intermediate code

if (tokens.length == 3 && tokens[1].equals("DS")) {

output += "(DL,01) (C," + tokens[2] + ")";

intermediateCode.add(output);

locCounter += Integer.parseInt(tokens[2]);

}

break;

}

}

// === OUTPUT ===

System.out.println("Intermediate Code:");

for (String ic : intermediateCode) {

System.out.println(ic);

}

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

System.out.println("Index\tSymbol\tAddress");

int i = 1;

for (Map.Entry<String, Integer> entry : symbolTable.entrySet()) {

System.out.println(i + "\t" + entry.getKey() + "\t" + entry.getValue());

i++;

}

}

private static int getSymbolIndex(String symbol, Map<String, Integer> symbolTable) {

int index = 1;

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

if (key.equals(symbol)) return index;

index++;

}

return -1;

}

}

**++++++++++++++++++++++++++++++++++++**

**(11)**

**(B)4.**Write a program to generate Intermediate code of a two-pass Assembler for the given Assembly language source code.

|  |
| --- |
| **INPUT/CODE**  START 200  READ M  READ N  MOVER AREG, M  SUB AREG, N  STOP  M DS 1  N DS 1  END |

import java.util.\*;

public class Main {

public static void main(String[] args) {

String[] code = {

"START 200",

"READ M",

"READ N",

"MOVER AREG, M",

"SUB AREG, N",

"STOP",

"M DS 1",

"N DS 1",

"END"

};

Map<String, Integer> symbolTable = new LinkedHashMap<>();

List<String> intermediateCode = new ArrayList<>();

int locCounter = 0;

boolean started = false;

// === FIRST PASS ===

for (String line : code) {

line = line.trim();

if (line.isEmpty()) continue;

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

if (tokens[0].equals("START")) {

locCounter = Integer.parseInt(tokens[1]);

started = true;

continue;

}

if (!started) continue;

if (tokens.length == 3 && tokens[1].equals("DS")) {

symbolTable.put(tokens[0], locCounter);

locCounter += Integer.parseInt(tokens[2]);

continue;

}

if (!Arrays.asList("READ", "MOVER", "SUB", "STOP", "END").contains(tokens[0])) {

symbolTable.put(tokens[0], locCounter);

}

if (!tokens[0].equals("END")) {

locCounter++;

}

}

// === SECOND PASS ===

locCounter = 0;

started = false;

for (String line : code) {

line = line.trim();

if (line.isEmpty()) continue;

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

if (tokens.length == 0) continue;

if (tokens[0].equals("START")) {

locCounter = Integer.parseInt(tokens[1]);

intermediateCode.add(locCounter + "\t(AD,01) (C," + locCounter + ")");

started = true;

continue;

}

if (!started) continue;

String output = locCounter + "\t";

String instr = tokens[0];

switch (instr) {

case "READ":

if (tokens.length >= 2) {

output += "(IS,09) (S," + getSymbolIndex(tokens[1], symbolTable) + ")";

intermediateCode.add(output);

locCounter++;

}

break;

case "MOVER":

case "SUB":

if (tokens.length >= 3) {

String reg = tokens[1].replace(",", "");

String sym = tokens[2];

String opcode = instr.equals("MOVER") ? "04" : "05";

output += "(IS," + opcode + ") (" + reg + ") (S," + getSymbolIndex(sym, symbolTable) + ")";

intermediateCode.add(output);

locCounter++;

}

break;

case "STOP":

output += "(IS,00)";

intermediateCode.add(output);

locCounter++;

break;

case "END":

intermediateCode.add(locCounter + "\t(AD,02)");

break;

default:

// ✅ Add DS to intermediate code

if (tokens.length == 3 && tokens[1].equals("DS")) {

output += "(DL,01) (C," + tokens[2] + ")";

intermediateCode.add(output);

locCounter += Integer.parseInt(tokens[2]);

}

break;

}

}

// === OUTPUT ===

System.out.println("Intermediate Code:");

for (String ic : intermediateCode) {

System.out.println(ic);

}

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

System.out.println("Index\tSymbol\tAddress");

int i = 1;

for (Map.Entry<String, Integer> entry : symbolTable.entrySet()) {

System.out.println(i + "\t" + entry.getKey() + "\t" + entry.getValue());

i++;

}

}

private static int getSymbolIndex(String symbol, Map<String, Integer> symbolTable) {

int index = 1;

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

if (key.equals(symbol)) return index;

index++;

}

return -1;

}

} **+++++++++++++++++++++++++++++++++++**

**(12)**

**(C)4.**Write a program to generate Intermediate code of a two-pass Assembler for the given Assembly language source code.

|  |
| --- |
| **INPUT/CODE**  START 300  READ M  READ N  MOVER AREG, M  SUB AREG, N  STOP  M DS 1  N DS 1  END |

import java.util.\*;

public class Main {

public static void main(String[] args) {

String[] code = {

"START 300",

"READ M",

"READ N",

"MOVER AREG, M",

"SUB AREG, N",

"STOP",

"M DS 1",

"N DS 1",

"END"

};

Map<String, Integer> symbolTable = new LinkedHashMap<>();

List<String> intermediateCode = new ArrayList<>();

int locCounter = 0;

boolean started = false;

// === FIRST PASS ===

for (String line : code) {

line = line.trim();

if (line.isEmpty()) continue;

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

if (tokens[0].equals("START")) {

locCounter = Integer.parseInt(tokens[1]);

started = true;

continue;

}

if (!started) continue;

if (tokens.length == 3 && tokens[1].equals("DS")) {

symbolTable.put(tokens[0], locCounter);

locCounter += Integer.parseInt(tokens[2]);

continue;

}

if (!Arrays.asList("READ", "MOVER", "SUB", "STOP", "END").contains(tokens[0])) {

symbolTable.put(tokens[0], locCounter);

}

if (!tokens[0].equals("END")) {

locCounter++;

}

}

// === SECOND PASS ===

locCounter = 0;

started = false;

for (String line : code) {

line = line.trim();

if (line.isEmpty()) continue;

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

if (tokens.length == 0) continue;

if (tokens[0].equals("START")) {

locCounter = Integer.parseInt(tokens[1]);

intermediateCode.add(locCounter + "\t(AD,01) (C," + locCounter + ")");

started = true;

continue;

}

if (!started) continue;

String output = locCounter + "\t";

String instr = tokens[0];

switch (instr) {

case "READ":

if (tokens.length >= 2) {

output += "(IS,09) (S," + getSymbolIndex(tokens[1], symbolTable) + ")";

intermediateCode.add(output);

locCounter++;

}

break;

case "MOVER":

case "SUB":

if (tokens.length >= 3) {

String reg = tokens[1].replace(",", "");

String sym = tokens[2];

String opcode = instr.equals("MOVER") ? "04" : "05";

output += "(IS," + opcode + ") (" + reg + ") (S," + getSymbolIndex(sym, symbolTable) + ")";

intermediateCode.add(output);

locCounter++;

}

break;

case "STOP":

output += "(IS,00)";

intermediateCode.add(output);

locCounter++;

break;

case "END":

intermediateCode.add(locCounter + "\t(AD,02)");

break;

default:

// ✅ Add DS to intermediate code

if (tokens.length == 3 && tokens[1].equals("DS")) {

output += "(DL,01) (C," + tokens[2] + ")";

intermediateCode.add(output);

locCounter += Integer.parseInt(tokens[2]);

}

break;

}

}

// === OUTPUT ===

System.out.println("Intermediate Code:");

for (String ic : intermediateCode) {

System.out.println(ic);

}

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

System.out.println("Index\tSymbol\tAddress");

int i = 1;

for (Map.Entry<String, Integer> entry : symbolTable.entrySet()) {

System.out.println(i + "\t" + entry.getKey() + "\t" + entry.getValue());

i++;

}

}

private static int getSymbolIndex(String symbol, Map<String, Integer> symbolTable) {

int index = 1;

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

if (key.equals(symbol)) return index;

index++;

}

return -1;

}

}

**++++++++++++++++++++++++++++++++++++**

**(13)**

**(A)5.**Write a program to generate Intermediate code of a two-pass Macro processor.

|  |  |
| --- | --- |
| **INPUT/CODE**  LOAD A  MACRO ABC  LOAD p  SUB q  MEND  STORE B  MULT D  MACRO ADD1 ARG  LOAD X  STORE ARG  MEND  …continued… | …continued…  LOAD B  MACRO ADD5 A1, A2, A3  STORE A2  ADD1 5  ADD1 10  LOAD A1  LOAD A3  MEND  ADD1 t  ABC  ADD5 D1, D2, D3  END |

import java.util.\*;

public class Main {

static class MacroEntry {

String name;

int mdtIndex;

MacroEntry(String name, int mdtIndex) {

this.name = name;

this.mdtIndex = mdtIndex;

}

}

static List<String> MDT = new ArrayList<>();

static List<MacroEntry> MNT = new ArrayList<>();

static Map<String, Integer> macroMap = new HashMap<>(); // For fast lookup

static Map<String, String> ALA = new LinkedHashMap<>();

static List<String> intermediateCode = new ArrayList<>();

public static void main(String[] args) {

String[] code = {

"LOAD A",

"MACRO ABC",

"LOAD p",

"SUB q",

"MEND",

"STORE B",

"MULT D",

"MACRO ADD1 ARG",

"LOAD X",

"STORE ARG",

"MEND",

"LOAD B",

"MACRO ADD5 A1, A2, A3",

"STORE A2",

"ADD1 5",

"ADD1 10",

"LOAD A1",

"LOAD A3",

"MEND",

"ADD1 t",

"ABC",

"ADD5 D1, D2, D3",

"END"

};

// ----- Pass 1 -----

boolean inMacroDef = false;

int mdtIndex = 0;

for (int i = 0; i < code.length; i++) {

String line = code[i].trim();

if (line.startsWith("MACRO")) {

inMacroDef = true;

String[] defParts = code[i + 1].trim().split("\\s+", 2);

String macroName = defParts[0];

macroMap.put(macroName, MNT.size());

MNT.add(new MacroEntry(macroName, MDT.size()));

i++;

continue;

} else if (line.equals("MEND")) {

MDT.add("MEND");

inMacroDef = false;

continue;

}

if (inMacroDef) {

MDT.add(line);

} else {

intermediateCode.add(line); // Non-macro lines

}

}

// ----- Pass 2 -----

List<String> finalCode = new ArrayList<>();

for (String line : intermediateCode) {

String[] tokens = line.split("\\s+|,", 2);

String mnemonic = tokens[0];

if (macroMap.containsKey(mnemonic)) {

// Macro invocation

int mIndex = macroMap.get(mnemonic);

int mdtStart = MNT.get(mIndex).mdtIndex;

ALA.clear();

// Prepare ALA

String[] actualArgs = tokens.length > 1 ? tokens[1].split(",") : new String[0];

String defLine = MDT.get(mdtStart);

String[] defTokens = defLine.split("\\s+");

String[] formalArgs = Arrays.copyOfRange(defTokens, 1, defTokens.length);

for (int i = 0; i < formalArgs.length; i++) {

if (i < actualArgs.length) {

ALA.put(formalArgs[i].trim(), actualArgs[i].trim());

}

}

// Expand macro body

for (int j = mdtStart + 1; !MDT.get(j).equals("MEND"); j++) {

String macroLine = MDT.get(j);

for (Map.Entry<String, String> arg : ALA.entrySet()) {

macroLine = macroLine.replace(arg.getKey(), arg.getValue());

}

finalCode.add(macroLine);

}

} else {

finalCode.add(line); // Normal instruction

}

}

// ----- Output -----

System.out.println("Intermediate Code:");

for (String line : finalCode) {

System.out.println(line);

}

System.out.println("\nMacro Name Table (MNT):");

for (int i = 0; i < MNT.size(); i++) {

System.out.println((i + 1) + ": " + MNT.get(i).name + " -> MDT Index " + MNT.get(i).mdtIndex);

}

System.out.println("\nMacro Definition Table (MDT):");

for (int i = 0; i < MDT.size(); i++) {

System.out.println((i + 1) + ": " + MDT.get(i));

}

}

}

**++++++++++++++++++++++++++++++++++++**

**(14)**

**(B)5.**Write a program to generate Intermediate code of a two-pass Macro processor.

|  |  |
| --- | --- |
| **INPUT/CODE**  STORE P  LOAD Q  MACRO PCG  LOAD m  ADD n  MEND  LOAD H  LOAD K  MACRO ADDi PAR  LOAD A  STORE PAR  MEND  …continued… | …continued…  DIV R  MACRO ADDii V1, V2, V3  STORE V2  ADDi 12  ADDi 7  LOAD V1  LOAD V3  MEND  PCG  ADDii Q1, Q2, Q3  ADDi w  END |

import java.util.\*;

public class Main {

static class MacroEntry {

String name;

int mdtIndex;

MacroEntry(String name, int mdtIndex) {

this.name = name;

this.mdtIndex = mdtIndex;

}

}

static List<String> MDT = new ArrayList<>();

static List<MacroEntry> MNT = new ArrayList<>();

static Map<String, Integer> macroMap = new HashMap<>();

static List<String> intermediateCode = new ArrayList<>();

static Map<String, String> ALA = new LinkedHashMap<>();

public static void main(String[] args) {

String[] code = {

"STORE P",

"LOAD Q",

"MACRO PCG",

"LOAD m",

"ADD n",

"MEND",

"LOAD H",

"LOAD K",

"MACRO ADDi PAR",

"LOAD A",

"STORE PAR",

"MEND",

"DIV R",

"MACRO ADDii V1, V2, V3",

"STORE V2",

"ADDi 12",

"ADDi 7",

"LOAD V1",

"LOAD V3",

"MEND",

"PCG",

"ADDii Q1, Q2, Q3",

"ADDi w",

"END"

};

// ---------- PASS 1 ----------

boolean inMacro = false;

int mdtIndex = 0;

for (int i = 0; i < code.length; i++) {

String line = code[i].trim();

if (line.startsWith("MACRO")) {

inMacro = true;

String[] def = code[i + 1].trim().split("\\s+", 2);

String macroName = def[0];

macroMap.put(macroName, MNT.size());

MNT.add(new MacroEntry(macroName, MDT.size()));

MDT.add(code[i + 1].trim()); // Add macro header

i++;

} else if (line.equals("MEND")) {

MDT.add("MEND");

inMacro = false;

} else if (inMacro) {

MDT.add(line);

} else {

intermediateCode.add(line);

}

}

// ---------- PASS 2 ----------

List<String> finalCode = new ArrayList<>();

for (String line : intermediateCode) {

String[] parts = line.trim().split("\\s+", 2);

String keyword = parts[0];

if (macroMap.containsKey(keyword)) {

// Macro invocation

int mntIdx = macroMap.get(keyword);

int mdtStart = MNT.get(mntIdx).mdtIndex;

ALA.clear();

String[] formalArgs = MDT.get(mdtStart).split("\\s+");

String[] actualArgs = parts.length > 1 ? parts[1].split(",") : new String[0];

for (int j = 1; j < formalArgs.length && j - 1 < actualArgs.length; j++) {

ALA.put(formalArgs[j], actualArgs[j - 1].trim());

}

for (int j = mdtStart + 1; !MDT.get(j).equals("MEND"); j++) {

String macroLine = MDT.get(j);

for (Map.Entry<String, String> entry : ALA.entrySet()) {

macroLine = macroLine.replace(entry.getKey(), entry.getValue());

}

finalCode.add(macroLine);

}

} else {

finalCode.add(line);

}

}

// ---------- OUTPUT ----------

System.out.println("INTERMEDIATE CODE:\n");

for (String line : finalCode) {

System.out.println(line);

}

System.out.println("\nMNT (Macro Name Table):");

for (int i = 0; i < MNT.size(); i++) {

System.out.println((i + 1) + ". " + MNT.get(i).name + " -> MDT Index " + MNT.get(i).mdtIndex);

}

System.out.println("\nMDT (Macro Definition Table):");

for (int i = 0; i < MDT.size(); i++) {

System.out.println((i + 1) + ". " + MDT.get(i));

}

}

}

**++++++++++++++++++++++++++++++++++++**

**(15)**

**(C)5.**Write a program to generate Intermediate code of a two-pass Macro processor.

|  |  |
| --- | --- |
| **INPUT/CODE**  LOAD F  STORE E  MACRO SRS  LOAD s  SUB t  MEND  STORE k  MACRO ADD3 XYZ  LOAD U  STORE XYZ  MEND  …continued… | …continued…  Add m  MACRO ADD1 Si, Sii, Siii  LOAD Sii  ADD3 1  ADD3 11  STORE Si  STORE Siii  MEND  SRS  ADD1 C1, C2, C3  ADD3 q  END |

import java.util.\*;

public class Main {

static class Macro {

String name;

int mdtIndex;

Macro(String name, int mdtIndex) {

this.name = name;

this.mdtIndex = mdtIndex;

}

}

static List<String> MDT = new ArrayList<>();

static List<Macro> MNT = new ArrayList<>();

static Map<String, Integer> macroMap = new HashMap<>();

static List<String> intermediateCode = new ArrayList<>();

static Map<String, String> ALA = new LinkedHashMap<>();

public static void main(String[] args) {

String[] source = {

"LOAD F",

"STORE E",

"MACRO SRS",

"LOAD s",

"SUB t",

"MEND",

"STORE k",

"MACRO ADD3 XYZ",

"LOAD U",

"STORE XYZ",

"MEND",

"Add m",

"MACRO ADD1 Si, Sii, Siii",

"LOAD Sii",

"ADD3 1",

"ADD3 11",

"STORE Si",

"STORE Siii",

"MEND",

"SRS",

"ADD1 C1, C2, C3",

"ADD3 q",

"END"

};

// ---------- PASS 1 ----------

boolean inMacro = false;

for (int i = 0; i < source.length; i++) {

String line = source[i].trim();

if (line.startsWith("MACRO")) {

inMacro = true;

String header = source[++i].trim();

String[] parts = header.split("\\s+", 2);

String macroName = parts[0];

macroMap.put(macroName, MNT.size());

MNT.add(new Macro(macroName, MDT.size()));

MDT.add(header);

} else if (line.equals("MEND")) {

MDT.add("MEND");

inMacro = false;

} else if (inMacro) {

MDT.add(line);

} else {

intermediateCode.add(line);

}

}

// ---------- PASS 2 ----------

List<String> finalCode = new ArrayList<>();

for (String line : intermediateCode) {

String[] parts = line.split("\\s+", 2);

String keyword = parts[0];

if (macroMap.containsKey(keyword)) {

expandMacro(keyword, parts.length > 1 ? parts[1] : "", finalCode);

} else {

finalCode.add(line);

}

}

// ---------- OUTPUT ----------

System.out.println("INTERMEDIATE CODE:");

for (String line : finalCode) {

System.out.println(line);

}

System.out.println("\nMNT (Macro Name Table):");

for (int i = 0; i < MNT.size(); i++) {

System.out.println((i + 1) + ". " + MNT.get(i).name + " -> MDT Index " + MNT.get(i).mdtIndex);

}

System.out.println("\nMDT (Macro Definition Table):");

for (int i = 0; i < MDT.size(); i++) {

System.out.println((i + 1) + ". " + MDT.get(i));

}

}

static void expandMacro(String macroName, String argStr, List<String> output) {

int mdtStart = MNT.get(macroMap.get(macroName)).mdtIndex;

String[] formalArgs = MDT.get(mdtStart).split("\\s+");

String[] actualArgs = argStr.split(",");

ALA.clear();

for (int j = 1; j < formalArgs.length && j <= actualArgs.length; j++) {

ALA.put(formalArgs[j], actualArgs[j - 1].trim());

}

for (int i = mdtStart + 1; !MDT.get(i).equals("MEND"); i++) {

String macroLine = MDT.get(i);

// Check for nested macro call

String[] lineParts = macroLine.trim().split("\\s+", 2);

if (macroMap.containsKey(lineParts[0])) {

expandMacro(lineParts[0], lineParts.length > 1 ? lineParts[1] : "", output);

continue;

}

for (Map.Entry<String, String> entry : ALA.entrySet()) {

macroLine = macroLine.replace(entry.getKey(), entry.getValue());

}

output.add(macroLine);

}

}

}

**++++++++++++++++++++++++++++++++++++**

**(16)**

**(D)5.**Write a program to generateIntermediate code of a two-pass Macro processor.

|  |  |
| --- | --- |
| **INPUT/CODE**  LOAD J  STORE M  MACRO EST  LOAD e  ADD d  MEND  LOAD S  MACRO SUB4 ABC  LOAD U  STORE ABC  MEND | …continued….  LOAD P  ADD V  MACRO ADD7 P4, P5, P6  LOAD P5  SUB4 XYZ  SUB 8  SUB 2  STORE P4  STORE P6  MEND  EST  ADD7 C4, C5, C6  SUB4 z  END |

import java.util.\*;

public class Main {

static class Macro {

String name;

int mdtIndex;

Macro(String name, int mdtIndex) {

this.name = name;

this.mdtIndex = mdtIndex;

}

}

static List<String> MDT = new ArrayList<>();

static List<Macro> MNT = new ArrayList<>();

static Map<String, Integer> macroMap = new HashMap<>();

static Map<String, String> ALA = new LinkedHashMap<>();

static List<String> intermediateCode = new ArrayList<>();

public static void main(String[] args) {

String[] source = {

"LOAD J",

"STORE M",

"MACRO EST",

"LOAD e",

"ADD d",

"MEND",

"LOAD S",

"MACRO SUB4 ABC",

"LOAD U",

"STORE ABC",

"MEND",

"LOAD P",

"ADD V",

"MACRO ADD7 P4, P5, P6",

"LOAD P5",

"SUB4 XYZ",

"SUB 8",

"SUB 2",

"STORE P4",

"STORE P6",

"MEND",

"EST",

"ADD7 C4, C5, C6",

"SUB4 z",

"END"

};

// -------- PASS 1 --------

boolean inMacro = false;

for (int i = 0; i < source.length; i++) {

String line = source[i].trim();

if (line.startsWith("MACRO")) {

inMacro = true;

String header = source[++i].trim();

String[] parts = header.split("\\s+", 2);

String macroName = parts[0];

macroMap.put(macroName, MNT.size());

MNT.add(new Macro(macroName, MDT.size()));

MDT.add(header);

} else if (line.equals("MEND")) {

MDT.add("MEND");

inMacro = false;

} else if (inMacro) {

MDT.add(line);

} else {

intermediateCode.add(line);

}

}

// -------- PASS 2 --------

List<String> finalCode = new ArrayList<>();

for (String line : intermediateCode) {

String[] parts = line.split("\\s+", 2);

String keyword = parts[0];

if (macroMap.containsKey(keyword)) {

String argStr = (parts.length > 1) ? parts[1] : "";

expandMacro(keyword, argStr, finalCode);

} else {

finalCode.add(line);

}

}

// -------- OUTPUT --------

System.out.println("INTERMEDIATE CODE:");

for (String line : finalCode) {

System.out.println(line);

}

System.out.println("\nMNT (Macro Name Table):");

for (int i = 0; i < MNT.size(); i++) {

System.out.println((i + 1) + ". " + MNT.get(i).name + " -> MDT Index " + MNT.get(i).mdtIndex);

}

System.out.println("\nMDT (Macro Definition Table):");

for (int i = 0; i < MDT.size(); i++) {

System.out.println((i + 1) + ". " + MDT.get(i));

}

}

static void expandMacro(String macroName, String argStr, List<String> output) {

int mdtStart = MNT.get(macroMap.get(macroName)).mdtIndex;

String[] formalArgs = MDT.get(mdtStart).split("\\s+");

String[] actualArgs = argStr.split(",");

ALA.clear();

for (int i = 1; i < formalArgs.length && i <= actualArgs.length; i++) {

ALA.put(formalArgs[i].trim(), actualArgs[i - 1].trim());

}

for (int i = mdtStart + 1; !MDT.get(i).equals("MEND"); i++) {

String line = MDT.get(i);

String[] parts = line.split("\\s+", 2);

if (macroMap.containsKey(parts[0])) {

String nestedArgStr = (parts.length > 1) ? parts[1] : "";

expandMacro(parts[0], nestedArgStr, output);

} else {

for (Map.Entry<String, String> entry : ALA.entrySet()) {

line = line.replace(entry.getKey(), entry.getValue());

}

output.add(line);

}

}

}

}

**++++++++++++++++++++++++++++++++++++**

**(17)**

**(A)6.**Write a program to generate MDT(Macro Definition Table) of a two-pass Macro processor.

|  |  |
| --- | --- |
| **INPUT/CODE**  LOAD A  STORE B  MACRO ABC  LOAD p  SUB q  MEND  MACRO ADD1 ARG  LOAD X  STORE ARG  MEND  ….Continued…. | …continued….  MACRO ADD5 A1, A2, A3  STORE A2  ADD1 5  ADD1 10  LOAD A1  LOAD A3  MEND  ABC  ADD5 D1, D2, D3  END |

import java.util.\*;

public class Main {

static class Macro {

String name;

int mdtIndex;

Macro(String name, int mdtIndex) {

this.name = name;

this.mdtIndex = mdtIndex;

}

}

// Macro Name Table

static List<Macro> MNT = new ArrayList<>();

// Macro Definition Table

static List<String> MDT = new ArrayList<>();

// Map to store arguments for macros

static Map<String, String> argumentMap = new HashMap<>();

public static void main(String[] args) {

String[] sourceCode = {

"LOAD A",

"STORE B",

"MACRO ABC",

"LOAD p",

"SUB q",

"MEND",

"MACRO ADD1 ARG",

"LOAD X",

"STORE ARG",

"MEND",

"MACRO ADD5 A1, A2, A3",

"STORE A2",

"ADD1 5",

"ADD1 10",

"LOAD A1",

"LOAD A3",

"MEND",

"ABC",

"ADD5 D1, D2, D3",

"END"

};

// -------- PASS 1 --------

boolean inMacro = false;

int mdtIndex = 0;

for (int i = 0; i < sourceCode.length; i++) {

String line = sourceCode[i].trim();

if (line.startsWith("MACRO")) {

// Beginning of macro definition

inMacro = true;

String macroName = line.split("\\s+")[1];

MNT.add(new Macro(macroName, mdtIndex));

// Storing macro header

MDT.add(line);

} else if (line.equals("MEND")) {

// End of macro definition

MDT.add("MEND");

inMacro = false;

} else if (inMacro) {

// Adding macro body to MDT

MDT.add(line);

mdtIndex++;

} else {

// If not inside a macro, ignore regular code (not needed for MDT)

}

}

// -------- OUTPUT --------

System.out.println("MDT (Macro Definition Table):");

for (int i = 0; i < MDT.size(); i++) {

System.out.println((i + 1) + ". " + MDT.get(i));

}

System.out.println("\nMNT (Macro Name Table):");

for (int i = 0; i < MNT.size(); i++) {

System.out.println((i + 1) + ". " + MNT.get(i).name + " -> MDT Index " + MNT.get(i).mdtIndex);

}

}

} **++++++++++++++++++++++++++++++++++++**

**(18)**

**(B)6.**Write a program to generate MDT(Macro Definition Table) of a two-pass Macro processor.

|  |  |
| --- | --- |
| **INPUT/CODE**  STORE P  LOAD Q  MACRO PCG  LOAD m  ADD n  MEND  MOV S  MACRO ADDi PAR  LOAD A  STORE PAR  MEND  …continued… | …continued…  DIV B  MACRO ADDii V1, V2, V3  STORE V2  ADDi 12  ADDi 7  LOAD V1  LOAD V3  MEND  PCG  ADDii Q1, Q2, Q3  END |

import java.util.\*;

public class Main {

static class Macro {

String name;

int mdtIndex;

Macro(String name, int mdtIndex) {

this.name = name;

this.mdtIndex = mdtIndex;

}

}

public static void main(String[] args) {

// Input Assembly + Macro code

String[] code = {

"STORE P",

"LOAD Q",

"MACRO PCG",

"LOAD m",

"ADD n",

"MEND",

"MOV S",

"MACRO ADDi PAR",

"LOAD A",

"STORE PAR",

"MEND",

"DIV B",

"MACRO ADDii V1, V2, V3",

"STORE V2",

"ADDi 12",

"ADDi 7",

"LOAD V1",

"LOAD V3",

"MEND",

"PCG",

"ADDii Q1, Q2, Q3",

"END"

};

List<String> MDT = new ArrayList<>();

List<Macro> MNT = new ArrayList<>();

boolean insideMacro = false;

int mdtIndex = 0;

for (int i = 0; i < code.length; i++) {

String line = code[i].trim();

if (line.startsWith("MACRO")) {

insideMacro = true;

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

if (parts.length >= 2) {

String macroName = parts[1];

MNT.add(new Macro(macroName, MDT.size()));

}

MDT.add(line);

} else if (line.equals("MEND")) {

MDT.add("MEND");

insideMacro = false;

} else if (insideMacro) {

MDT.add(line);

}

}

// Displaying MDT

System.out.println("========== Macro Definition Table (MDT) ==========");

for (int i = 0; i < MDT.size(); i++) {

System.out.println((i + 1) + ": " + MDT.get(i));

}

// Displaying MNT

System.out.println("\n========== Macro Name Table (MNT) ==========");

for (int i = 0; i < MNT.size(); i++) {

Macro m = MNT.get(i);

System.out.println((i + 1) + ": " + m.name + " -> MDT Index: " + m.mdtIndex);

}

}

}

**++++++++++++++++++++++++++++++++++++**

**(19)**

**(A)7.**Write a program to generate MNT(Macro Name Table) of a two-pass Macro processor.

|  |  |
| --- | --- |
| **INPUT/CODE**  LOAD F  STORE E  MACRO SRS  LOAD s  SUB t  MEND  MACRO SRS XYZ  LOAD U  STORE XYZ  MEND | MACRO ADD1 Si, Sii, Siii  LOAD Sii  ADD3 1  SRS 11  STORE Si  STORE Siii  MEND  SRS  ADD1 C1, C2, C3  END |

import java.util.\*;

public class MacroProcessorMNT {

static class MacroEntry {

String name;

int paramCount;

MacroEntry(String name, int paramCount) {

this.name = name;

this.paramCount = paramCount;

}

}

public static void main(String[] args) {

String[] code = {

"LOAD F",

"STORE E",

"MACRO SRS",

"LOAD s",

"SUB t",

"MEND",

"MACRO SRS XYZ",

"LOAD U",

"STORE XYZ",

"MEND",

"MACRO ADD1 Si, Sii, Siii",

"LOAD Sii",

"ADD3 1",

"SRS 11",

"STORE Si",

"STORE Siii",

"MEND",

"SRS",

"ADD1 C1, C2, C3",

"END"

};

List<MacroEntry> mnt = new ArrayList<>();

for (String line : code) {

line = line.trim();

if (line.startsWith("MACRO")) {

String[] parts = line.split("\\s+", 3);

if (parts.length >= 2) {

String macroName = parts[1];

int paramCount = 0;

if (parts.length == 3) {

String[] params = parts[2].split(",");

paramCount = params.length;

}

mnt.add(new MacroEntry(macroName, paramCount));

}

}

}

// Print MNT

System.out.println("========== Macro Name Table (MNT) ==========");

int index = 1;

for (MacroEntry entry : mnt) {

System.out.printf("%-3d Name: %-6s Parameters: %d\n", index++, entry.name, entry.paramCount);

}

}

}

**++++++++++++++++++++++++++++++++++++**

**(20)**

**(B)7.**Write a program to generate MNT(Macro Name Table) of a two-pass Macro processor.

|  |  |
| --- | --- |
| **INPUT/CODE**  LOAD J  STORE M  MACRO EST1  LOAD e  ADD d  MEND  MACRO EST ABC  EST1  STORE ABC  MEND | MACRO ADD7 P4, P5, P6  LOAD P5  EST 8  SUB4 2  STORE P4  STORE P6  MEND  EST  ADD7 C4, C5, C6  END |

import java.util.\*;

public class MacroProcessorMNT {

// Class to store Macro details (name and parameter count)

static class MacroEntry {

String name;

int paramCount;

MacroEntry(String name, int paramCount) {

this.name = name;

this.paramCount = paramCount;

}

}

public static void main(String[] args) {

String[] code = {

"LOAD J",

"STORE M",

"MACRO EST1",

"LOAD e",

"ADD d",

"MEND",

"MACRO EST ABC",

"EST1",

"STORE ABC",

"MEND",

"MACRO ADD7 P4, P5, P6",

"LOAD P5",

"EST 8",

"SUB4 2",

"STORE P4",

"STORE P6",

"MEND",

"EST",

"ADD7 C4, C5, C6",

"END"

};

// List to hold the Macro Name Table (MNT)

List<MacroEntry> mnt = new ArrayList<>();

// Iterate over the input code and process macro definitions

for (String line : code) {

line = line.trim();

// Look for MACRO definitions

if (line.startsWith("MACRO")) {

String[] parts = line.split("\\s+", 3); // Split by space

if (parts.length >= 2) {

String macroName = parts[1]; // Macro name

int paramCount = 0;

// Check if the macro has parameters

if (parts.length == 3) {

String[] params = parts[2].split(","); // Split parameters by comma

paramCount = params.length; // Count the parameters

}

// Add macro to MNT

mnt.add(new MacroEntry(macroName, paramCount));

}

}

}

// Print the Macro Name Table (MNT)

System.out.println("========== Macro Name Table (MNT) ==========");

int index = 1;

for (MacroEntry entry : mnt) {

System.out.printf("%-3d Name: %-6s Parameters: %d\n", index++, entry.name, entry.paramCount);

}

}

}

**++++++++++++++++++++++++++++++++++++**

**(21)**

**(A)8.**Write a program using LEX Tool, to implement a lexical analyzer for parts of speech for given English language without Symbol table.

**INPUT**

Dread it. Run from it.

Destiny arrives all the same.

Output-

%{

#include <stdio.h>

#include <string.h>

#include <ctype.h>

// Function declarations

void classifyWord(char \*word);

%}

%%

[a-zA-Z]+ {

classifyWord(yytext); // Classify each word based on regular expression match

}

[.,!?;] {

printf("Punctuation: %s\n", yytext); // Print punctuation

}

[ \t\n]+ ; // Ignore whitespace

%%

void classifyWord(char \*word) {

// Simple POS tagging based on ending patterns

if (strcmp(word, "the") == 0 || strcmp(word, "a") == 0 || strcmp(word, "an") == 0) {

printf("Article: %s\n", word);

} else if (strcmp(word, "run") == 0 || strcmp(word, "arrives") == 0) {

printf("Verb: %s\n", word);

} else if (strcmp(word, "destiny") == 0) {

printf("Noun: %s\n", word);

} else if (strcmp(word, "it") == 0) {

printf("Pronoun: %s\n", word);

} else if (strcmp(word, "dread") == 0) {

printf("Verb: %s\n", word);

} else {

printf("Unknown word: %s\n", word);

}

}

int main() {

yylex(); // Start the lexical analysis

return 0;

}

lex pos.l

gcc lex.yy.c -o pos\_analyzer -ll

./pos\_analyzer

Dread it. Run from it.

Destiny arrives all the same.

**++++++++++++++++++++++++++++++++++++**

**(22)**

**(B)8.**Write a program using LEX Tool, to implement a lexical analyzer for parts of speech for given English language without Symbol table.

**INPUT**

Hello! How are you?

I’m fine, Thank You.

%{

#include <stdio.h>

#include <string.h>

#include <ctype.h>

// Function declaration

void classifyWord(char \*word);

%}

%%

[a-zA-Z]+ {

classifyWord(yytext); // Classify the matched word

}

[.,!?;] {

printf("Punctuation: %s\n", yytext); // Print matched punctuation

}

[ \t\n]+ ; // Ignore whitespace

%%

// Function to classify part of speech based on hardcoded rules

void classifyWord(char \*word) {

// Convert word to lowercase for consistent matching

char lower[100];

int i;

for (i = 0; word[i]; i++)

lower[i] = tolower(word[i]);

lower[i] = '\0';

if (strcmp(lower, "the") == 0 || strcmp(lower, "a") == 0 || strcmp(lower, "an") == 0) {

printf("Article: %s\n", word);

} else if (strcmp(lower, "run") == 0 || strcmp(lower, "arrives") == 0 || strcmp(lower, "dread") == 0) {

printf("Verb: %s\n", word);

} else if (strcmp(lower, "destiny") == 0) {

printf("Noun: %s\n", word);

} else if (strcmp(lower, "it") == 0) {

printf("Pronoun: %s\n", word);

} else if (strcmp(lower, "hello") == 0 || strcmp(lower, "thank") == 0) {

printf("Interjection: %s\n", word);

} else if (strcmp(lower, "i") == 0 || strcmp(lower, "you") == 0) {

printf("Pronoun: %s\n", word);

} else {

printf("Unknown word: %s\n", word);

}

}

int main() {

printf("Enter text input:\n");

yylex(); // Start lexical analysis

return 0;

}

lex pos.l

gcc lex.yy.c -o pos\_analyzer -ll

./pos\_analyzer

Hello! How are you?I’m fine, Thank You.**++++++++++++++++++++++++++++++++++++**

**(23)**

**(C)8.**Write a program using LEX Tool, to implement a lexical analyzer for parts of speech for given English language without Symbol table.

**INPUT**

The important thing is to not stop questioning,

Curiosity has its own reason for existing.

%{

#include <stdio.h>

#include <string.h>

#include <ctype.h>

// Function declaration

void classifyWord(char \*word);

%}

%%

[a-zA-Z]+ {

classifyWord(yytext); // Classify matched words

}

[.,!?;] {

printf("Punctuation: %s\n", yytext); // Recognize punctuation

}

[ \t\n]+ ; // Ignore whitespace

%%

// Part-of-speech classification based on simple rules

void classifyWord(char \*word) {

char lower[100];

int i;

for (i = 0; word[i]; i++)

lower[i] = tolower(word[i]);

lower[i] = '\0';

// POS tags based on simple hardcoded logic

if (strcmp(lower, "the") == 0 || strcmp(lower, "a") == 0 || strcmp(lower, "an") == 0) {

printf("Article: %s\n", word);

} else if (strcmp(lower, "is") == 0 || strcmp(lower, "has") == 0 || strcmp(lower, "stop") == 0) {

printf("Verb: %s\n", word);

} else if (strcmp(lower, "thing") == 0 || strcmp(lower, "curiosity") == 0 || strcmp(lower, "reason") == 0) {

printf("Noun: %s\n", word);

} else if (strcmp(lower, "important") == 0 || strcmp(lower, "own") == 0) {

printf("Adjective: %s\n", word);

} else if (strcmp(lower, "to") == 0 || strcmp(lower, "for") == 0) {

printf("Preposition: %s\n", word);

} else if (strcmp(lower, "not") == 0) {

printf("Adverb: %s\n", word);

} else if (strcmp(lower, "questioning") == 0 || strcmp(lower, "existing") == 0) {

printf("Gerund/Verb: %s\n", word);

} else {

printf("Unknown word: %s\n", word);

}

}

int main() {

printf("Enter input text:\n");

yylex(); // Start lexical analysis

return 0;

}

lex pos.l

gcc lex.yy.c -o pos\_analyzer -ll

./pos\_analyzer

The important thing is to not stop questioning, Curiosity has its own reason for existing.

**++++++++++++++++++++++++++++++++++++**

**(24)**

**(D)8.**Write a program using LEX Tool, to implement a lexical analyzer for parts of speech for given English language without Symbol table.

**INPUT**

If you never light the cigarette,

you never give the thing that can kill you the power it needs to kill you.

%{

#include <stdio.h>

#include <string.h>

#include <ctype.h>

// Function declaration

void classifyWord(char \*word);

%}

%%

[a-zA-Z]+ {

classifyWord(yytext); // Match words and classify

}

[.,!?;] {

printf("Punctuation: %s\n", yytext); // Recognize punctuation

}

[ \t\n]+ ; // Ignore whitespace

%%

void classifyWord(char \*word) {

char lower[100];

int i;

for (i = 0; word[i]; i++)

lower[i] = tolower(word[i]);

lower[i] = '\0';

// Simple POS classification using hardcoded rules

if (strcmp(lower, "if") == 0 || strcmp(lower, "and") == 0 || strcmp(lower, "but") == 0) {

printf("Conjunction: %s\n", word);

} else if (strcmp(lower, "you") == 0 || strcmp(lower, "it") == 0) {

printf("Pronoun: %s\n", word);

} else if (strcmp(lower, "never") == 0) {

printf("Adverb: %s\n", word);

} else if (strcmp(lower, "light") == 0 || strcmp(lower, "give") == 0 || strcmp(lower, "kill") == 0) {

printf("Verb: %s\n", word);

} else if (strcmp(lower, "cigarette") == 0 || strcmp(lower, "thing") == 0 || strcmp(lower, "power") == 0) {

printf("Noun: %s\n", word);

} else if (strcmp(lower, "that") == 0 || strcmp(lower, "can") == 0) {

printf("Auxiliary/Determiner: %s\n", word);

} else if (strcmp(lower, "the") == 0) {

printf("Article: %s\n", word);

} else if (strcmp(lower, "to") == 0) {

printf("Preposition/Infinitive Marker: %s\n", word);

} else {

printf("Unknown word: %s\n", word);

}

}

int main() {

printf("Enter input:\n");

yylex(); // Start lexical analysis

return 0;

}

lex pos.l

gcc lex.yy.c -o pos\_analyzer -ll

./pos\_analyzer

If you never light the cigarette, you never give the thing that can kill you the power it needs to kill you..

**++++++++++++++++++++++++++++++++++++**

**(25)**

**(E)8.**Write a program using LEX Tool, to implement a lexical analyzer for parts of speech for given English language without Symbol table.

**INPUT**

Every generation imagines itself to be more intelligent than the one that went before it, and wiser than the one that comes after it.

%{

#include <stdio.h>

#include <string.h>

#include <ctype.h>

void classifyWord(char \*word);

%}

%%

[a-zA-Z]+ { classifyWord(yytext); }

[.,!?;] { printf("Punctuation: %s\n", yytext); }

[ \t\n]+ ; // Ignore whitespace

%%

void classifyWord(char \*word) {

char lower[100];

int i;

for (i = 0; word[i]; i++)

lower[i] = tolower(word[i]);

lower[i] = '\0';

// POS classification

if (strcmp(lower, "every") == 0 || strcmp(lower, "more") == 0) {

printf("Determiner: %s\n", word);

} else if (strcmp(lower, "generation") == 0 || strcmp(lower, "one") == 0) {

printf("Noun: %s\n", word);

} else if (strcmp(lower, "imagines") == 0 || strcmp(lower, "went") == 0 || strcmp(lower, "comes") == 0) {

printf("Verb: %s\n", word);

} else if (strcmp(lower, "itself") == 0 || strcmp(lower, "it") == 0) {

printf("Pronoun: %s\n", word);

} else if (strcmp(lower, "to") == 0 || strcmp(lower, "than") == 0 || strcmp(lower, "before") == 0 || strcmp(lower, "after") == 0) {

printf("Preposition: %s\n", word);

} else if (strcmp(lower, "be") == 0) {

printf("Auxiliary Verb: %s\n", word);

} else if (strcmp(lower, "intelligent") == 0 || strcmp(lower, "wiser") == 0) {

printf("Adjective: %s\n", word);

} else if (strcmp(lower, "that") == 0 || strcmp(lower, "and") == 0) {

printf("Conjunction: %s\n", word);

} else {

printf("Unknown: %s\n", word);

}

}

int main() {

printf("Enter input text:\n");

yylex();

return 0;

}

lex pos.l

gcc lex.yy.c -o pos\_analyzer -ll

./pos\_analyzer

Every generation imagines itself to be more intelligent than the one that went before it, and wiser than the one that comes after it.

**++++++++++++++++++++++++++++++++++++**

**(26)**

**(F)8.**Write a program using LEX Tool, to implement a lexical analyzer for parts of speech for given English language without Symbol table.

**INPUT**

Human Beings have a strong dramatic instinct toward binary thinking, a basic urge to divide things into two distinct groups, with nothing but an empty gap in between.

%{

#include <stdio.h>

#include <string.h>

#include <ctype.h>

void classifyWord(char \*word);

%}

%%

[a-zA-Z]+ { classifyWord(yytext); }

[.,!?;] { printf("Punctuation: %s\n", yytext); }

[ \t\n]+ ; // Ignore whitespace

%%

void classifyWord(char \*word) {

char lower[100];

int i;

for (i = 0; word[i]; i++)

lower[i] = tolower(word[i]);

lower[i] = '\0';

// POS classification (simplified, hardcoded)

if (strcmp(lower, "human") == 0 || strcmp(lower, "beings") == 0 ||

strcmp(lower, "instinct") == 0 || strcmp(lower, "thinking") == 0 ||

strcmp(lower, "urge") == 0 || strcmp(lower, "gap") == 0 ||

strcmp(lower, "groups") == 0 || strcmp(lower, "things") == 0) {

printf("Noun: %s\n", word);

} else if (strcmp(lower, "have") == 0 || strcmp(lower, "divide") == 0) {

printf("Verb: %s\n", word);

} else if (strcmp(lower, "strong") == 0 || strcmp(lower, "dramatic") == 0 ||

strcmp(lower, "basic") == 0 || strcmp(lower, "binary") == 0 ||

strcmp(lower, "distinct") == 0 || strcmp(lower, "empty") == 0) {

printf("Adjective: %s\n", word);

} else if (strcmp(lower, "a") == 0 || strcmp(lower, "an") == 0) {

printf("Article: %s\n", word);

} else if (strcmp(lower, "to") == 0 || strcmp(lower, "into") == 0 ||

strcmp(lower, "in") == 0 || strcmp(lower, "between") == 0 ||

strcmp(lower, "with") == 0) {

printf("Preposition: %s\n", word);

} else if (strcmp(lower, "but") == 0) {

printf("Conjunction: %s\n", word);

} else if (strcmp(lower, "nothing") == 0) {

printf("Pronoun: %s\n", word);

} else {

printf("Unknown: %s\n", word);

}

}

int main() {

printf("Enter input text:\n");

yylex();

return 0;

}

lex pos.l

gcc lex.yy.c -o pos\_analyzer -ll

./pos\_analyzer

Human Beings have a strong dramatic instinct toward binary thinking, a basic urge to divide things into two distinct groups, with nothing but an empty gap in between..

**++++++++++++++++++++++++++++++++++++**

**(27)**

**(A)9.**Write a program using LEX Tool, to implement a lexical analyzer for given C programming language without Symbol table.

**INPUT**

{

int m=10,n=2,o;

o = m – n;

}

%{

#include <stdio.h>

#include <string.h>

%}

%%

"int" { printf("Keyword: %s\n", yytext); }

"float"|"char"|"return" { printf("Keyword: %s\n", yytext); }

"=" { printf("Assignment Operator: %s\n", yytext); }

"-" { printf("Arithmetic Operator: %s\n", yytext); }

"+"|"\*"|"/" { printf("Arithmetic Operator: %s\n", yytext); }

[0-9]+ { printf("Constant: %s\n", yytext); }

[a-zA-Z\_][a-zA-Z0-9\_]\* { printf("Identifier: %s\n", yytext); }

"{"|"}"|"("|")" { printf("Brace/Parenthesis: %s\n", yytext); }

";" { printf("Semicolon: %s\n", yytext); }

"," { printf("Comma: %s\n", yytext); }

[ \t\n]+ ; // Ignore whitespace

. { printf("Unknown: %s\n", yytext); }

%%

int main() {

printf("Enter C code:\n");

yylex();

return 0;

}

lex c\_lexer.l

gcc lex.yy.c -o c\_lexer -ll

./c\_lexer < input.c

input.c

{

int m=10,n=2,o;

o = m - n;

}**++++++++++++++++++++++++++++++++++++**

**(28)**

**(B)9.**Write a program using LEX Tool, to implement a lexical analyzer for given C programming language without Symbol table.

**INPUT**

%{

#include <stdio.h>

%}

%%

"int"|"float"|"char"|"double"|"void" { printf("Keyword: %s\n", yytext); }

"printf" { printf("Function: %s\n", yytext); }

\"(\\.|[^\\"])\*\" { printf("String Literal: %s\n", yytext); }

[a-zA-Z\_][a-zA-Z0-9\_]\* { printf("Identifier: %s\n", yytext); }

[0-9]+ { printf("Constant: %s\n", yytext); }

"=" { printf("Assignment Operator: %s\n", yytext); }

";" { printf("Semicolon: %s\n", yytext); }

"{"|"}" { printf("Brace: %s\n", yytext); }

[ \t\n]+ ; // Ignore whitespace

. { printf("Unknown Token: %s\n", yytext); }

%%

int main() {

printf("Enter C code:\n");

yylex();

return 0;

}

lex c\_lexer.l

gcc lex.yy.c -o c\_lexer -ll

./c\_lexer < input.c

{

char = a;

printf("Hello World");

}**++++++++++++++++++++++++++++++++++++**

**(29)**

**(C)9.**Write a program using LEX Tool, to implement a lexical analyzer for given C programming language without Symbol table.

**INPUT**

%{

#include <stdio.h>

%}

%%

"int"|"float"|"char"|"double"|"void" { printf("Keyword: %s\n", yytext); }

[a-zA-Z\_][a-zA-Z0-9\_]\* { printf("Identifier: %s\n", yytext); }

[0-9]+ { printf("Constant: %s\n", yytext); }

"=" { printf("Assignment Operator: %s\n", yytext); }

";" { printf("Semicolon: %s\n", yytext); }

"," { printf("Comma: %s\n", yytext); }

"(" { printf("Left Parenthesis: %s\n", yytext); }

")" { printf("Right Parenthesis: %s\n", yytext); }

"{"|"}" { printf("Brace: %s\n", yytext); }

"\\+"|"-"|"\\\*"|"/" { printf("Arithmetic Operator: %s\n", yytext); }

[ \t\n]+ ; // Skip whitespace

. { printf("Unknown Token: %s\n", yytext); }

%%

int main() {

printf("Enter your C code:\n");

yylex();

return 0;

}

lex c\_lexer.l

gcc lex.yy.c -o c\_lexer -ll

./c\_lexer < input.c

{

int d=10,t=2;

int s;

s=d/t; }

**++++++++++++++++++++++++++++++++++++**

**(30)**

**(D)9.**Write a program using LEX Tool, to implement a lexical analyzer for given C programming language without Symbol table.

**INPUT**

%{

#include <stdio.h>

%}

%%

"int"|"float"|"char"|"double"|"void" { printf("Keyword: %s\n", yytext); }

[0-9]+ { printf("Constant: %s\n", yytext); }

[a-zA-Z\_][a-zA-Z0-9\_]\* { printf("Identifier: %s\n", yytext); }

"=" { printf("Assignment Operator: %s\n", yytext); }

";" { printf("Semicolon: %s\n", yytext); }

"," { printf("Comma: %s\n", yytext); }

"(" { printf("Left Parenthesis: %s\n", yytext); }

")" { printf("Right Parenthesis: %s\n", yytext); }

"{"|"}" { printf("Brace: %s\n", yytext); }

"\\+"|"-"|"\\\*"|"/" { printf("Arithmetic Operator: %s\n", yytext); }

[ \t\n]+ ; // Ignore whitespace

. { printf("Unknown character: %s\n", yytext); }

%%

int main() {

printf("Lexical Analysis Output:\n");

yylex();

return 0;

}

lex c\_lexer.l

gcc lex.yy.c -o c\_lexer -ll

./c\_lexer < input.c

{

int a=3;

int b=4;

float c;

c = (a\*a + b\*b) \*2

}**++++++++++++++++++++++++++++++++++++**

**(31)**

**(E)9.** Write a program using LEX Tool, to implement a lexical analyzer for given C programming language without Symbol table.

**INPUT**

%{

#include <stdio.h>

%}

%%

"int"|"float"|"char"|"double"|"void" { printf("Keyword: %s\n", yytext); }

[0-9]+ { printf("Constant: %s\n", yytext); }

[a-zA-Z\_][a-zA-Z0-9\_]\* { printf("Identifier: %s\n", yytext); }

"=" { printf("Assignment Operator: %s\n", yytext); }

";" { printf("Semicolon: %s\n", yytext); }

"," { printf("Comma: %s\n", yytext); }

"(" { printf("Left Parenthesis: %s\n", yytext); }

")" { printf("Right Parenthesis: %s\n", yytext); }

"{"|"}" { printf("Brace: %s\n", yytext); }

"\\+"|"-"|"\\\*"|"/" { printf("Arithmetic Operator: %s\n", yytext); }

"%" { printf("Modulus Operator: %s\n", yytext); }

"\"[^\"]\*\"" { printf("String Literal: %s\n", yytext); }

[ \t\n]+ ; // Ignore whitespace

. { printf("Unknown character: %s\n", yytext); }

%%

int main() {

printf("Lexical Analysis Output:\n");

yylex();

return 0;

}

lex c\_lexer.l

gcc lex.yy.c -o c\_lexer -ll

./c\_lexer < input.c

{

int total = 100;

int i = 10;

printf("The value of total and i is : %d, %d", total, i);

}

**++++++++++++++++++++++++++++++++++++**

**(32)**

**(A)10.**Write a program to evaluate a given arithmetic expression using YACC specification.

**INPUT**

0.33\*12-4-4+(3\*2)

%{

#include "y.tab.h"

#include <stdlib.h>

%}

%%

[0-9]+\.[0-9]+ { yylval.f = atof(yytext); return FLOAT; } // Float

[0-9]+ { yylval.f = atof(yytext); return FLOAT; } // Treat INTs as float too

"+" { return PLUS; }

"-" { return MINUS; }

"\*" { return MUL; }

"/" { return DIV; }

"(" { return LPAREN; }

")" { return RPAREN; }

[ \t\n]+ { /\* Skip whitespace \*/ }

. { return yytext[0]; }

%%

int yywrap() { return 1; }

%{

#include <stdio.h>

#include <stdlib.h>

int yylex(void);

int yyerror(char \*s);

float result = 0;

%}

%union {

float f;

}

%token <f> FLOAT

%token PLUS MINUS MUL DIV

%token LPAREN RPAREN

%type <f> expr term factor

%%

input:

expr { result = $1; }

;

expr:

term { $$ = $1; }

| expr PLUS term { $$ = $1 + $3; }

| expr MINUS term { $$ = $1 - $3; }

;

term:

factor { $$ = $1; }

| term MUL factor { $$ = $1 \* $3; }

| term DIV factor { $$ = $1 / $3; }

;

factor:

FLOAT { $$ = $1; }

| LPAREN expr RPAREN { $$ = $2; }

;

%%

int main() {

printf("Enter an arithmetic expression:\n");

if (yyparse() == 0) {

printf("Result = %.2f\n", result);

}

return 0;

}

int yyerror(char \*s) {

fprintf(stderr, "Syntax error: %s\n", s);

return 1;

}

lex lexer.l

yacc -d parser.y

gcc lex.yy.c y.tab.c -o calc -lm

./calc

0.33\*12-4-4+(3\*2)

Ctrl + D

**++++++++++++++++++++++++++++++++++++**

**(33)**

**(B)10.**Write a program to evaluate a given arithmetic expression using YACC specification.

**INPUT**

(78-44\*4)/3\*2

%{

#include "y.tab.h"

#include <stdlib.h>

%}

%%

[0-9]+\.[0-9]+ { yylval.f = atof(yytext); return FLOAT; } // Float

[0-9]+ { yylval.f = atof(yytext); return FLOAT; } // Treat INTs as float too

"+" { return PLUS; }

"-" { return MINUS; }

"\*" { return MUL; }

"/" { return DIV; }

"(" { return LPAREN; }

")" { return RPAREN; }

[ \t\n]+ { /\* Skip whitespace \*/ }

. { return yytext[0]; }

%%

int yywrap() { return 1; }

%{

#include <stdio.h>

#include <stdlib.h>

int yylex(void);

int yyerror(char \*s);

float result = 0;

%}

%union {

float f;

}

%token <f> FLOAT

%token PLUS MINUS MUL DIV

%token LPAREN RPAREN

%type <f> expr term factor

%%

input:

expr { result = $1; }

;

expr:

term { $$ = $1; }

| expr PLUS term { $$ = $1 + $3; }

| expr MINUS term { $$ = $1 - $3; }

;

term:

factor { $$ = $1; }

| term MUL factor { $$ = $1 \* $3; }

| term DIV factor { $$ = $1 / $3; }

;

factor:

FLOAT { $$ = $1; }

| LPAREN expr RPAREN { $$ = $2; }

;

%%

int main() {

printf("Enter an arithmetic expression:\n");

if (yyparse() == 0) {

printf("Result = %.2f\n", result);

}

return 0;

}

int yyerror(char \*s) {

fprintf(stderr, "Syntax error: %s\n", s);

return 1;

}

lex lexer.l

yacc -d parser.y

gcc lex.yy.c y.tab.c -o calc -lm

./calc

(78-44\*4)/3\*2

Enter

Ctrl + D

**++++++++++++++++++++++++++++++++++++**

**(34)**

**(C)10.**Write a program to evaluate a given arithmetic expression using YACC specification.

**INPUT**

98/44-(3\*8)\*4\*2

%{

#include "y.tab.h"

#include <stdlib.h>

%}

%%

[0-9]+\.[0-9]+ { yylval.f = atof(yytext); return FLOAT; } // Float

[0-9]+ { yylval.f = atof(yytext); return FLOAT; } // Treat INTs as float too

"+" { return PLUS; }

"-" { return MINUS; }

"\*" { return MUL; }

"/" { return DIV; }

"(" { return LPAREN; }

")" { return RPAREN; }

[ \t\n]+ { /\* Skip whitespace \*/ }

. { return yytext[0]; }

%%

int yywrap() { return 1; }

%{

#include <stdio.h>

#include <stdlib.h>

int yylex(void);

int yyerror(char \*s);

float result = 0;

%}

%union {

float f;

}

%token <f> FLOAT

%token PLUS MINUS MUL DIV

%token LPAREN RPAREN

%type <f> expr term factor

%%

input:

expr { result = $1; }

;

expr:

term { $$ = $1; }

| expr PLUS term { $$ = $1 + $3; }

| expr MINUS term { $$ = $1 - $3; }

;

term:

factor { $$ = $1; }

| term MUL factor { $$ = $1 \* $3; }

| term DIV factor { $$ = $1 / $3; }

;

factor:

FLOAT { $$ = $1; }

| LPAREN expr RPAREN { $$ = $2; }

;

%%

int main() {

printf("Enter an arithmetic expression:\n");

if (yyparse() == 0) {

printf("Result = %.2f\n", result);

}

return 0;

}

int yyerror(char \*s) {

fprintf(stderr, "Syntax error: %s\n", s);

return 1;

}

lex lexer.l

yacc -d parser.y

gcc lex.yy.c y.tab.c -o calc -lm

./calc

98/44-(3\*8)\*4\*2

Enter

Ctrl + D

**++++++++++++++++++++++++++++++++++++**

**(35)**

**(D)10.**Write a program to evaluate a given arithmetic expression using YACC specification.

**INPUT**

1.44+22.4-12.8+11\*3

%{

#include "y.tab.h"

#include <stdlib.h>

%}

%%

[0-9]+\.[0-9]+ { yylval.f = atof(yytext); return FLOAT; } // Float

[0-9]+ { yylval.f = atof(yytext); return FLOAT; } // Treat INTs as float too

"+" { return PLUS; }

"-" { return MINUS; }

"\*" { return MUL; }

"/" { return DIV; }

"(" { return LPAREN; }

")" { return RPAREN; }

[ \t\n]+ { /\* Skip whitespace \*/ }

. { return yytext[0]; }

%%

int yywrap() { return 1; }

%{

#include <stdio.h>

#include <stdlib.h>

int yylex(void);

int yyerror(char \*s);

float result = 0;

%}

%union {

float f;

}

%token <f> FLOAT

%token PLUS MINUS MUL DIV

%token LPAREN RPAREN

%type <f> expr term factor

%%

input:

expr { result = $1; }

;

expr:

term { $$ = $1; }

| expr PLUS term { $$ = $1 + $3; }

| expr MINUS term { $$ = $1 - $3; }

;

term:

factor { $$ = $1; }

| term MUL factor { $$ = $1 \* $3; }

| term DIV factor { $$ = $1 / $3; }

;

factor:

FLOAT { $$ = $1; }

| LPAREN expr RPAREN { $$ = $2; }

;

%%

int main() {

printf("Enter an arithmetic expression:\n");

if (yyparse() == 0) {

printf("Result = %.2f\n", result);

}

return 0;

}

int yyerror(char \*s) {

fprintf(stderr, "Syntax error: %s\n", s);

return 1;

}

lex lexer.l

yacc -d parser.y

gcc lex.yy.c y.tab.c -o calc -lm

./calc

1.44 + 22.4 - 12.8 + 11 \* 3

**++++++++++++++++++++++++++++++++++++**

**(36)**

**(A)11.**Write a program to evaluate a given variable name using YACC specification.

**SAMPLE INPUT**

1. pune
2. PUNE
3. Pune1
4. pUNE\_2

%{

#include "y.tab.h" // Include YACC header to access token definitions

%}

%%

[a-z][a-z0-9\_]\* { return VAR\_NAME; } // Variable name starts with lowercase letter, followed by letters, digits, or underscore

[A-Z][A-Z0-9\_]\* { return VAR\_NAME; } // Variable name starts with uppercase letter, followed by letters, digits, or underscore

[a-zA-Z][a-zA-Z0-9\_]\* { return VAR\_NAME; } // Mixed-case variable name (both lowercase and uppercase)

. { return yytext[0]; } // Return the first character for invalid symbols

[ \t\n] { /\* Ignore whitespace \*/ }

%%

int yywrap() {

return 1;

}

%{

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

extern char \*yytext; // From the lexer

int yylex(); // Forward declaration of lexer function

int yyerror(char \*s); // Forward declaration of error handler

%}

%token VAR\_NAME

%%

input:

| input line

;

line:

VAR\_NAME { printf("Valid variable name: %s\n", yytext); }

| error { printf("Invalid variable name: %s\n", yytext); }

;

%%

int main() {

printf("Enter variable names:\n");

yyparse(); // Start parsing

return 0;

}

int yyerror(char \*s) {

printf("Error: %s\n", s);

return 0;

}

lex lexer.l

yacc -d parser.y

gcc lex.yy.c y.tab.c -o calc -lm

./calc

pune

PUNE

Pune1

pUNE\_2

**++++++++++++++++++++++++++++++++++++**

**(37)**

**(B)11**.Write a program to convert small case letters to UPPER case or vise versa using YACC specification.

**SAMPLE INPUT**

1. **Pune – pUNE**
2. **PUNE –pune**

%{

#include "y.tab.h"

%}

%%

[a-z] { return LOWERCASE; }

[A-Z] { return UPPERCASE; }

[ \t\n] { return WHITESPACE; }

. { return OTHER; }

%%

int yywrap() {

return 1;

}

%{

#include <stdio.h>

#include <ctype.h>

extern char \*yytext;

int yylex();

int yyerror(char \*s);

%}

%token LOWERCASE UPPERCASE WHITESPACE OTHER

%%

input:

/\* empty \*/

| input token

;

token:

LOWERCASE { printf("%c", toupper(yytext[0])); }

| UPPERCASE { printf("%c", tolower(yytext[0])); }

| WHITESPACE { printf("%c", yytext[0]); }

| OTHER { printf("%c", yytext[0]); }

;

%%

int main() {

printf("Enter text (Ctrl+D to end input):\n");

yyparse();

printf("\n");

return 0;

}

int yyerror(char \*s) {

return 0; // Silently ignore errors

}

lex lexer.l

yacc -d parser.y

gcc lex.yy.c y.tab.c -o case\_converter -ll

./case\_converter

Pune

PUNE

**+++++++++++++++++++++++++++++++++++**

**(38)**

**(C)11.**Write a program to evaluate a given built-in functions using YACC specification.

**INPUT**

1.u= sqrt(36)

2. v = strlen(“pune”)

%{

#include "y.tab.h"

#include <string.h>

#include <stdlib.h>

%}

%%

[0-9]+ { yylval.num = atoi(yytext); return NUM; }

\"([^\"]\*)\" {

int len = yyleng - 2;

yylval.str = (char\*)malloc(len + 1);

strncpy(yylval.str, yytext + 1, len);

yylval.str[len] = '\0';

return STRING;

}

[a-zA-Z\_][a-zA-Z0-9\_]\* {

if (strcmp(yytext, "sqrt") == 0) return SQRT;

if (strcmp(yytext, "strlen") == 0) return STRLEN;

yylval.str = strdup(yytext);

return VAR;

}

"=" { return ASSIGN; }

"(" { return LPAREN; }

")" { return RPAREN; }

"," { return COMMA; }

"+" { return '+'; }

"-" { return '-'; }

"\*" { return '\*'; }

"/" { return '/'; }

[ \t\n]+ { /\* skip whitespace \*/ }

. { printf("Unknown character: %s\n", yytext); }

%%

int yywrap() {

return 1;

}

%{

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#include <string.h>

extern int yylex();

extern char \*yytext;

int yyerror(char \*s); // Forward declaration (correct return type)

%}

%union {

int num;

char\* str;

}

%token <num> NUM

%token <str> VAR STRING

%token SQRT STRLEN ASSIGN LPAREN RPAREN COMMA

%type <num> expression

%left '+' '-'

%left '\*' '/'

%right UMINUS

%%

program:

statements

;

statements:

statement

| statements statement

;

statement:

VAR ASSIGN expression { printf("%s = %d\n", $1, $3); free($1); }

;

expression:

NUM { $$ = $1; }

| VAR { $$ = 0; printf("Variable: %s\n", $1); free($1); }

| SQRT LPAREN expression RPAREN { $$ = sqrt($3); }

| STRLEN LPAREN STRING RPAREN { $$ = strlen($3); free($3); }

| expression '+' expression { $$ = $1 + $3; }

| expression '-' expression { $$ = $1 - $3; }

| expression '\*' expression { $$ = $1 \* $3; }

| expression '/' expression { $$ = $1 / $3; }

| '-' expression %prec UMINUS { $$ = -$2; }

| LPAREN expression RPAREN { $$ = $2; }

;

%%

int main() {

printf("Enter expressions:\n");

yyparse();

return 0;

}

int yyerror(char \*s) {

printf("Error: %s\n", s);

return 0;

}

lex lexer.l

yacc -d parser.y

gcc lex.yy.c y.tab.c -o evaluator -lm -ll

./evaluator

u = sqrt(36)

v = strlen("pune")

**++++++++++++++++++++++++++++++++++++**

**(39)**

**(D)11.**Write a program to evaluate a given built-in functions using YACC specification.

**INPUT**

u= sin(12)+cos(12)

%{

#include "y.tab.h"

#include <math.h>

#include <stdlib.h>

#include <string.h>

%}

%%

[ \t\n]+ { /\* Ignore whitespace \*/ }

sin { return SIN; }

cos { return COS; }

[0-9]+(\.[0-9]+)? {

yylval.num = atof(yytext);

return NUM;

}

[a-zA-Z\_][a-zA-Z0-9\_]\* {

yylval.str = strdup(yytext);

return VAR;

}

"=" { return '='; }

"+" { return '+'; }

"-" { return '-'; }

"\*" { return '\*'; }

"/" { return '/'; }

"(" { return '('; }

")" { return ')'; }

. { printf("Unknown character: %s\n", yytext); }

%%

int yywrap() {

return 1;

}

%{

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#include <string.h>

extern int yylex();

void yyerror(const char \*s);

%}

%union {

double num;

char\* str;

}

%token <num> NUM

%token <str> VAR

%token SIN COS

%left '+' '-'

%left '\*' '/'

%right UMINUS

%type <num> expression

%%

program:

statement

;

statement:

VAR '=' expression { printf("%s = %f\n", $1, $3); free($1); }

;

expression:

NUM { $$ = $1; }

| VAR { printf("Variable: %s\n", $1); $$ = 0; free($1); }

| SIN '(' expression ')' { $$ = sin($3); }

| COS '(' expression ')' { $$ = cos($3); }

| expression '+' expression { $$ = $1 + $3; }

| expression '-' expression { $$ = $1 - $3; }

| expression '\*' expression { $$ = $1 \* $3; }

| expression '/' expression { $$ = $1 / $3; }

| '-' expression %prec UMINUS { $$ = -$2; }

| '(' expression ')' { $$ = $2; }

;

%%

int main() {

printf("Enter an expression:\n");

return yyparse();

}

void yyerror(const char \*s) {

fprintf(stderr, "Error: %s\n", s);

}

lex lexer.l

yacc -d parser.y

gcc lex.yy.c y.tab.c -o evaluator -lm -ll

./evaluator

u = sin(12) + cos(12)

**++++++++++++++++++++++++++++++++++++**

**(40)**

**(E)11.**Write a program to evaluate a given built-in functions using YACC specification.

**INPUT**

p= pow(3,2) / log (24)

%{

#include "y.tab.h"

#include <math.h>

#include <stdlib.h>

#include <string.h>

%}

%%

[ \t\n]+ { /\* Ignore whitespace \*/ }

pow { return POW; }

log { return LOG; }

[0-9]+(\.[0-9]+)? {

yylval.num = atof(yytext);

return NUM;

}

[a-zA-Z\_][a-zA-Z0-9\_]\* {

yylval.str = strdup(yytext);

return VAR;

}

"=" { return '='; }

"+" { return '+'; }

"-" { return '-'; }

"\*" { return '\*'; }

"/" { return '/'; }

"(" { return '('; }

")" { return ')'; }

"," { return ','; }

. { printf("Unknown character: %s\n", yytext); }

%%

int yywrap() {

return 1;

}

%{

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#include <string.h>

extern int yylex();

void yyerror(const char \*s);

%}

%union {

double num;

char\* str;

}

%token <num> NUM

%token <str> VAR

%token POW LOG

%left '+' '-'

%left '\*' '/'

%right UMINUS

%type <num> expression

%%

program:

statement

;

statement:

VAR '=' expression {

printf("%s = %f\n", $1, $3);

free($1);

}

;

expression:

NUM { $$ = $1; }

| VAR { printf("Variable: %s\n", $1); $$ = 0; free($1); }

| POW '(' expression ',' expression ')' { $$ = pow($3, $5); }

| LOG '(' expression ')' { $$ = log($3); }

| expression '+' expression { $$ = $1 + $3; }

| expression '-' expression { $$ = $1 - $3; }

| expression '\*' expression { $$ = $1 \* $3; }

| expression '/' expression { $$ = $1 / $3; }

| '-' expression %prec UMINUS { $$ = -$2; }

| '(' expression ')' { $$ = $2; }

;

%%

int main() {

printf("Enter an expression:\n");

return yyparse();

}

void yyerror(const char \*s) {

fprintf(stderr, "Error: %s\n", s);

}

lex lexer.l

yacc -d parser.y

gcc lex.yy.c y.tab.c -o evaluator -lm -ll

./evaluator

p = pow(3, 2) / log(24)

**++++++++++++++++++++++++++++++++++++**

**(41)**

**(A)12.**Write a program to generate three address code for the given simple expression.

**INPUT**

w = u\*u - u\*v+ v\*v

public class ThreeAddressCodeGenerator {

public static void main(String[] args) {

// Variables for the expression

String u = "u";

String v = "v";

String w = "w";

// Temporary variables to store intermediate results

String t1 = "t1";

String t2 = "t2";

String t3 = "t3";

String t4 = "t4";

// Print the Three-Address Code step by step

System.out.println("Three-Address Code for the expression:");

// Step 1: t1 = u \* u

System.out.println(t1 + " = " + u + " \* " + u);

// Step 2: t2 = u \* v

System.out.println(t2 + " = " + u + " \* " + v);

// Step 3: t3 = v \* v

System.out.println(t3 + " = " + v + " \* " + v);

// Step 4: t4 = t1 - t2

System.out.println(t4 + " = " + t1 + " - " + t2);

// Step 5: w = t4 + t3

System.out.println(w + " = " + t4 + " + " + t3);

}

}

**++++++++++++++++++++++++++++++++++++**

**(42)**

**(B)12.**Write a program to generate three address code for the given simple expression.

**INPUT**

y=x\*x + w-v / r+r

public class Main {

public static void main(String[] args) {

// Variables for the expression

String x = "x";

String w = "w";

String v = "v";

String r = "r";

String y = "y";

// Temporary variables to store intermediate results

String t1 = "t1";

String t2 = "t2";

String t3 = "t3";

String t4 = "t4";

// Print the Three-Address Code step by step

System.out.println("Three-Address Code for the expression:");

// Step 1: t1 = x \* x

System.out.println(t1 + " = " + x + " \* " + x);

// Step 2: t2 = v / r

System.out.println(t2 + " = " + v + " / " + r);

// Step 3: t3 = w - t2

System.out.println(t3 + " = " + w + " - " + t2);

// Step 4: t4 = t1 + t3

System.out.println(t4 + " = " + t1 + " + " + t3);

// Step 5: y = t4 + r

System.out.println(y + " = " + t4 + " + " + r);

}

}

**++++++++++++++++++++++++++++++++++++**

**(43)**

**(C)12**.Write a program to generate three address code for the given simple expression.

**INPUT**

w = u\*u - u\*v+ v\*v

public class Main {

public static void main(String[] args) {

// Variables for the expression

String u = "u";

String v = "v";

String w = "w";

// Temporary variables to store intermediate results

String t1 = "t1";

String t2 = "t2";

String t3 = "t3";

// Print the Three-Address Code step by step

System.out.println("Three-Address Code for the expression:");

// Step 1: t1 = u \* u

System.out.println(t1 + " = " + u + " \* " + u);

// Step 2: t2 = u \* v

System.out.println(t2 + " = " + u + " \* " + v);

// Step 3: t3 = v \* v

System.out.println(t3 + " = " + v + " \* " + v);

// Step 4: t4 = t1 - t2

String t4 = "t4";

System.out.println(t4 + " = " + t1 + " - " + t2);

// Step 5: w = t4 + t3

System.out.println(w + " = " + t4 + " + " + t3);

}

}

**++++++++++++++++++++++++++++++++++++**

**(44)**

**(D)12**.Write a program to generate three address code for the given simple expression.

**INPUT**

t = o\*a - o\*b+ o\*c

public class Main {

public static void main(String[] args) {

// Variables for the expression

String o = "o";

String a = "a";

String b = "b";

String c = "c";

String t = "t";

// Temporary variables to store intermediate results

String t1 = "t1";

String t2 = "t2";

String t3 = "t3";

// Print the Three-Address Code step by step

System.out.println("Three-Address Code for the expression:");

// Step 1: t1 = o \* a

System.out.println(t1 + " = " + o + " \* " + a);

// Step 2: t2 = o \* b

System.out.println(t2 + " = " + o + " \* " + b);

// Step 3: t3 = o \* c

System.out.println(t3 + " = " + o + " \* " + c);

// Step 4: t4 = t1 - t2

String t4 = "t4";

System.out.println(t4 + " = " + t1 + " - " + t2);

// Step 5: t5 = t4 + t3

String t5 = "t5";

System.out.println(t5 + " = " + t4 + " + " + t3);

// Step 6: t = t5

System.out.println(t + " = " + t5);

}

}

**++++++++++++++++++++++++++++++++++++**

**(45)**

**(E)12.** Write a program to generate three address code for the given simple expression.

**INPUT**

t = j / k – y / u – i

public class Main {

public static void main(String[] args) {

// Variables for the expression

String j = "j";

String k = "k";

String y = "y";

String u = "u";

String i = "i";

String t = "t";

// Temporary variables to store intermediate results

String t1 = "t1";

String t2 = "t2";

String t3 = "t3";

// Print the Three-Address Code step by step

System.out.println("Three-Address Code for the expression:");

// Step 1: t1 = j / k

System.out.println(t1 + " = " + j + " / " + k);

// Step 2: t2 = y / u

System.out.println(t2 + " = " + y + " / " + u);

// Step 3: t3 = t1 - t2

System.out.println(t3 + " = " + t1 + " - " + t2);

// Step 4: t = t3 - i

System.out.println(t + " = " + t3 + " - " + i);

}

}

**++++++++++++++++++++++++++++++++++++**

**(46)**

**(F)12.**Write a program to generate three address code for the given simple expression.

**INPUT**

a = m \* n - o – p / q

public class Main {

public static void main(String[] args) {

// Variables for the expression

String m = "m";

String n = "n";

String o = "o";

String p = "p";

String q = "q";

String a = "a";

// Temporary variables to store intermediate results

String t1 = "t1";

String t2 = "t2";

String t3 = "t3";

// Print the Three-Address Code step by step

System.out.println("Three-Address Code for the expression:");

// Step 1: t1 = m \* n

System.out.println(t1 + " = " + m + " \* " + n);

// Step 2: t2 = p / q

System.out.println(t2 + " = " + p + " / " + q);

// Step 3: t3 = t1 - o

System.out.println(t3 + " = " + t1 + " - " + o);

// Step 4: a = t3 - t2

System.out.println(a + " = " + t3 + " - " + t2);

}

}

**++++++++++++++++++++++++++++++++++++**

**(47)**

**(G)12.**Write a program to generate three address code for the given simple expression.

**INPUT**

a = f ^ r – u \* f \* t – p

public class Main {

public static void main(String[] args) {

// Variables for the expression

String f = "f";

String r = "r";

String u = "u";

String t = "t";

String p = "p";

String a = "a";

// Temporary variables to store intermediate results

String t1 = "t1"; // f ^ r

String t2 = "t2"; // u \* f

String t3 = "t3"; // u \* f \* t

String t4 = "t4"; // (f ^ r) - (u \* f \* t)

// Print the Three-Address Code step by step

System.out.println("Three-Address Code for the expression:");

// Step 1: t1 = f ^ r

System.out.println(t1 + " = " + f + " ^ " + r);

// Step 2: t2 = u \* f

System.out.println(t2 + " = " + u + " \* " + f);

// Step 3: t3 = t2 \* t

System.out.println(t3 + " = " + t2 + " \* " + t);

// Step 4: t4 = t1 - t3

System.out.println(t4 + " = " + t1 + " - " + t3);

// Step 5: a = t4 - p

System.out.println(a + " = " + t4 + " - " + p);

}

}

**++++++++++++++++++++++++++++++++++++**

**(48)**

**(H)12.**Write a program to generate three address code for the given simple expression.

**INPUT**

a = ( b\*b + c\*c ) \* (p – q – r)

public class Main {

public static void main(String[] args) {

// Variables for the expression

String b = "b";

String c = "c";

String p = "p";

String q = "q";

String r = "r";

String a = "a";

// Temporary variables to store intermediate results

String t1 = "t1";

String t2 = "t2";

String t3 = "t3";

String t4 = "t4";

// Print the Three-Address Code step by step

System.out.println("Three-Address Code for the expression:");

// Step 1: t1 = b \* b

System.out.println(t1 + " = " + b + " \* " + b);

// Step 2: t2 = c \* c

System.out.println(t2 + " = " + c + " \* " + c);

// Step 3: t3 = t1 + t2

System.out.println(t3 + " = " + t1 + " + " + t2);

// Step 4: t4 = p - q

System.out.println(t4 + " = " + p + " - " + q);

// Step 5: t5 = t4 - r

String t5 = "t5";

System.out.println(t5 + " = " + t4 + " - " + r);

// Step 6: a = t3 \* t5

System.out.println(a + " = " + t3 + " \* " + t5);

}

}

**++++++++++++++++++++++++++++++++++++**