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

```cpp
#include <iostream>
#include <iomanip>
#include <map>
#include <sstream>
#include <vector>

using namespace std;

struct Symbol {
    string label;
    int address;
};

bool isInstruction(const string& word) {
    // Instructions and directives that shouldn't be in symbol table
    vector<string> instructions = {
        "START", "READ", "MOVER", "COMP", "BC", "SUB", "STOP", "END"
    };
    for (const string& inst : instructions) {
        if (word == inst) return true;
    }
    return false;
}

int main() {
    vector<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, int> symbolTable;
    int LC = 0;
```

```cpp
    bool started = false;

    for (const string& line : code) {
        stringstream ss(line);
        vector<string> tokens;
        string token;
        while (ss >> token) {
            tokens.push_back(token);
        }

        if (tokens.empty()) continue;

        // START
        if (tokens[0] == "START") {
            LC = stoi(tokens[1]);
            started = true;
            continue;
        }

        if (!started) continue;

        // END
        if (tokens[0] == "END") break;

        // DS declaration
        if (tokens.size() >= 3 && tokens[1] == "DS") {
            symbolTable[tokens[0]] = LC;
            LC += stoi(tokens[2]);
            continue;
        }

        // Label line (LOOP, BACK, etc.)
        if (tokens.size() >= 2 && !isInstruction(tokens[0])) {
            // First token is a label
            symbolTable[tokens[0]] = LC;
        }

        LC++;  // Increment LC for each instruction
    }

    // Print Symbol Table
    cout << "Symbol Table:\n";
    cout << left << setw(10) << "Label" << "Address\n";
    for (const auto& entry : symbolTable) {
        cout << left << setw(10) << entry.first << entry.second << "\n";
    }

    return 0;
```

```
}
```

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

```
#include <iostream>
#include <iomanip>
#include <map>
#include <vector>
#include <sstream>
using namespace std;

struct Literal {
    string name;
    int address;
};

bool isLiteral(const string& operand) {
    return operand.size() >= 2 && operand[0] == '=' && operand[1] == '\'';
}

int main() {
    vector<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"
};

int LC = 0;
bool started = false;

map<string, int> literalTable;
vector<string> literalPool;

for (const string& line : code) {
    stringstream ss(line);
    vector<string> tokens;
    string token;
    while (ss >> token) {
        tokens.push_back(token);
    }

    if (tokens.empty()) continue;

    // Handle START
    if (tokens[0] == "START") {
        LC = stoi(tokens[1]);
        started = true;
        continue;
    }

    if (!started) continue;

    // Collect literals from operands
    for (const string& tok : tokens) {
        if (isLiteral(tok) && literalTable.find(tok) == literalTable.end()) {
            literalPool.push_back(tok);
            literalTable[tok] = -1;  // placeholder for address
        }
    }

    // After END, assign addresses to literals
    if (tokens[0] == "END") {
        for (const string& lit : literalPool) {
            if (literalTable[lit] == -1) {
                literalTable[lit] = LC++;
            }
        }
    } else {
        LC++;  // Each instruction = 1 memory word
    }
}
```

```
    // Print Literal Table
    cout << "Literal Table:\n";
    cout << left << setw(10) << "Literal" << "Address\n";
    for (const auto& entry : literalTable) {
        cout << left << setw(10) << entry.first << entry.second << "\n";
    }

    return 0;
}
```

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

```
#include <iostream>
#include <iomanip>
#include <sstream>
#include <vector>
#include <map>
using namespace std;

bool isLiteral(const string& token) {
    return token.size() >= 3 && token[0] == '=' && token[1] == '\'';
}

int main() {
    vector<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"
    };
```

```
    int LC = 0;
    bool started = false;

    vector<string> literalList;
    map<string, int> literalTable;
    vector<int> poolTable;

    int literalIndex = 0;

    for (size_t i = 0; i < code.size(); ++i) {
        stringstream ss(code[i]);
        vector<string> tokens;
        string token;
        while (ss >> token) {
            tokens.push_back(token);
        }

        if (tokens.empty()) continue;

        // START
        if (tokens[0] == "START") {
            LC = stoi(tokens[1]);
            started = true;
            continue;
        }

        if (!started) continue;

        // If END or LTORG -> assign addresses to pending literals
        if (tokens[0] == "LTORG" || tokens[0] == "END") {
            poolTable.push_back(literalIndex + 1);  // Pool starts from
index+1 (1-based)
            for (const string& lit : literalList) {
                if (literalTable[lit] == -1) {
                    literalTable[lit] = LC++;
                    literalIndex++;
                }
            }
            literalList.clear();
            if (tokens[0] == "END") break;
            continue;
        }

        // Scan for literals
        for (const string& tok : tokens) {
            if (isLiteral(tok) && literalTable.find(tok) ==
literalTable.end()) {
```

```
                literalList.push_back(tok);
                literalTable[tok] = -1;  // placeholder
            }
        }

        LC++;  // each instruction is 1 word
    }

    // --- OUTPUT ---
    // Literal Table
    cout << "Literal Table:\n";
    cout << left << setw(10) << "Literal" << "Address\n";
    for (const auto& entry : literalTable) {
        cout << left << setw(10) << entry.first << entry.second << "\n";
    }

    // Pool Table
    cout << "\nPool Table:\n";
    cout << "Pool#\tIndex (1-based in literal table)\n";
    for (size_t i = 0; i < poolTable.size(); ++i) {
        cout << i + 1 << "\t" << poolTable[i] << "\n";
    }

    return 0;
}
```

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

```
#include <iostream>
#include <iomanip>
#include <map>
#include <vector>
#include <sstream>
using namespace std;

struct Symbol {
    string name;
    int address;
};

map<string, string> registers = {
```

```
    {"AREG", "1"},
    {"BREG", "2"},
    {"CREG", "3"},
    {"DREG", "4"}
};

map<string, pair<string, string>> opcodes = {
    {"START", {"AD", "01"}},
    {"END",   {"AD", "02"}},
    {"READ",  {"IS", "09"}},
    {"MOVER", {"IS", "04"}},
    {"SUB",   {"IS", "05"}},
    {"STOP",  {"IS", "00"}},
    {"DS",    {"DL", "01"}}
};

int main() {
    vector<string> code = {
        "START 100",
        "READ A",
        "READ B",
        "MOVER AREG, A",
        "SUB AREG, B",
        "STOP",
        "A DS 1",
        "B DS 1",
        "END"
    };

    map<string, int> symbolTable;
    int LC = 0;
    int symIndex = 1;

    // Pass 1: Build Symbol Table
    for (const string& line : code) {
        stringstream ss(line);
        vector<string> tokens;
        string tok;
        while (ss >> tok) tokens.push_back(tok);

        if (tokens.empty()) continue;

        if (tokens[0] == "START") {
            LC = stoi(tokens[1]);
        } else if (tokens[0] == "DS") {
            // Do nothing here
        } else if (tokens.size() == 3 && tokens[1] == "DS") {
            symbolTable[tokens[0]] = LC;
```

```
            LC += stoi(tokens[2]);
        } else if (tokens[0] == "END") {
            // Nothing
        } else {
            if (tokens.size() == 2 && tokens[0] == "READ") {
                symbolTable[tokens[1]] = -1;
            } else if (tokens.size() == 3) {
                string operand = tokens[2];
                if (operand.back() == ',') operand.pop_back();
                if (symbolTable.find(operand) == symbolTable.end()) {
                    symbolTable[operand] = -1;
                }
            }
            LC++;
        }
    }

    // Reset LC for Pass 2
    LC = 0;
    cout << "Intermediate Code:\n";

    for (const string& line : code) {
        stringstream ss(line);
        vector<string> tokens;
        string tok;
        while (ss >> tok) tokens.push_back(tok);

        if (tokens.empty()) continue;

        string mnemonic = tokens[0];

        if (mnemonic == "START") {
            LC = stoi(tokens[1]);
            cout << "(" << opcodes[mnemonic].first << "," << opcodes[mnemonic].second << ")
" << "(C," << LC << ")\n";
        } else if (mnemonic == "END") {
            cout << "(" << opcodes[mnemonic].first << "," << opcodes[mnemonic].second <<
")\n";
        } else if (tokens.size() == 3 && tokens[1] == "DS") {
            string label = tokens[0];
            int size = stoi(tokens[2]);
            symbolTable[label] = LC;
            cout << "(" << opcodes["DS"].first << "," << opcodes["DS"].second << ") " <<
"(C," << size << ")\n";
            LC += size;
        } else if (mnemonic == "READ" || mnemonic == "STOP") {
            cout << "(" << opcodes[mnemonic].first << "," << opcodes[mnemonic].second << ")
";
```

```
            if (mnemonic == "READ") {
                cout << "(S," << tokens[1] << ")";
            }
            cout << "\n";
            LC++;
        } else {
            // IS instructions like MOVER, SUB
            string reg = registers[tokens[1]];
            string operand = tokens[2];
            if (operand.back() == ',') operand.pop_back();
            cout << "(" << opcodes[mnemonic].first << "," << opcodes[mnemonic].second << ")
";
            cout << "(" << reg << ") ";
            cout << "(S," << operand << ")\n";
            LC++;
        }
    }

    return 0;
}
```

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

```
#include <iostream>
#include <vector>
#include <map>
#include <sstream>
#include <algorithm>

using namespace std;

// Structure to represent a macro definition
struct MacroDefinition {
    string name;
    vector<string> parameters;
    vector<string> body;
};

// Function to trim whitespace from a string
string trim(const string& str) {
    size_t first = str.find_first_not_of(' ');
    if (string::npos == first) return "";
    size_t last = str.find_last_not_of(' ');
```

```
    return str.substr(first, (last - first + 1));
}

// Function to split a string into tokens
vector<string> split(const string& s) {
    vector<string> tokens;
    string token;
    istringstream tokenStream(s);
    while (tokenStream >> token) {
        tokens.push_back(token);
    }
    return tokens;
}

// Function to process the input and generate intermediate code
void generateIntermediateCode(const vector<string>& input) {
    map<string, MacroDefinition> macroTable;
    vector<string> intermediateCode;
    vector<string> expandedCode;

    // Pass 1: Process macro definitions
    for (size_t i = 0; i < input.size(); i++) {
        string line = input[i];
        vector<string> tokens = split(trim(line));

        if (tokens.empty()) continue;

        if (tokens[0] == "MACRO") {
            MacroDefinition macro;
            macro.name = tokens[1];

            // Extract parameters if any
            for (size_t j = 2; j < tokens.size(); j++) {
                macro.parameters.push_back(tokens[j]);
            }

            // Collect macro body until MEND
            i++; // Move to next line
            while (i < input.size() && trim(input[i]) != "MEND") {
                macro.body.push_back(input[i]);
                i++;
            }

            // Add macro to macro table
            macroTable[macro.name] = macro;
        } else {
            intermediateCode.push_back(line);
        }
```

```
    }

    // Pass 2: Expand macro calls
    for (const string& line : intermediateCode) {
        vector<string> tokens = split(trim(line));

        if (tokens.empty()) {
            expandedCode.push_back("");
            continue;
        }

        // Check if this is a macro call
        if (macroTable.find(tokens[0]) != macroTable.end()) {
            MacroDefinition macro = macroTable[tokens[0]];
            vector<string> args;

            // Extract arguments if any
            for (size_t i = 1; i < tokens.size(); i++) {
                args.push_back(tokens[i]);
            }

            // Expand macro body with argument substitution
            for (const string& macroLine : macro.body) {
                string expandedLine = macroLine;

                // Replace parameters with arguments
                for (size_t i = 0; i < macro.parameters.size() && i < args.size(); i++) {
                    string param = macro.parameters[i];
                    string arg = args[i];

                    size_t pos = expandedLine.find(param);
                    while (pos != string::npos) {
                        expandedLine.replace(pos, param.length(), arg);
                        pos = expandedLine.find(param, pos + arg.length());
                    }
                }

                expandedCode.push_back(expandedLine);
            }
        } else {
            expandedCode.push_back(line);
        }
    }

    // Output the intermediate code
    cout << "Intermediate Code (Expanded):" << endl;
    cout << "----------------------------" << endl;
    for (const string& line : expandedCode) {
```

```
        if (trim(line) == "MEND" || trim(line) == "MACRO") continue;
        cout << line << endl;
    }
}

int main() {
    // Input program
    vector<string> input = {
        "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"
    };

    generateIntermediateCode(input);

    return 0;
}
```

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

```cpp
#include <iostream>
#include <vector>
#include <map>
#include <sstream>
#include <iomanip>

using namespace std;

struct MacroDefinition {
    string name;
    vector<string> parameters;
    vector<string> body;
};

string trim(const string& str) {
    size_t first = str.find_first_not_of(' ');
    if (string::npos == first) return "";
    size_t last = str.find_last_not_of(' ');
    return str.substr(first, (last - first + 1));
}

vector<string> split(const string& s) {
    vector<string> tokens;
    string token;
    istringstream tokenStream(s);
    while (tokenStream >> token) {
        tokens.push_back(token);
    }
    return tokens;
}

void generateMDT(const vector<string>& input) {
    map<string, MacroDefinition> macroTable;
    vector<pair<string, vector<string>>> mdtEntries;

    for (size_t i = 0; i < input.size(); i++) {
        string line = input[i];
        vector<string> tokens = split(trim(line));

        if (tokens.empty()) continue;

        if (tokens[0] == "MACRO") {
            MacroDefinition macro;
            macro.name = tokens[1];
```

```
            // Store parameters
            for (size_t j = 2; j < tokens.size(); j++) {
                macro.parameters.push_back(tokens[j]);
            }

            // Process macro body
            i++;
            while (i < input.size() && trim(input[i]) != "MEND") {
                string bodyLine = input[i];
                // Replace parameters with placeholders (#1, #2, etc.)
                for (size_t paramIdx = 0; paramIdx < macro.parameters.size();
paramIdx++) {
                    string param = macro.parameters[paramIdx];
                    size_t pos = bodyLine.find(param);
                    while (pos != string::npos) {
                        bodyLine.replace(pos, param.length(), "#" +
to_string(paramIdx+1));
                        pos = bodyLine.find(param, pos + 2);
                    }
                }
                macro.body.push_back(bodyLine);
                i++;
            }
            macro.body.push_back("MEND");
            macroTable[macro.name] = macro;

            // Add to MDT entries
            for (const auto& line : macro.body) {
                mdtEntries.emplace_back(macro.name, vector<string>{line});
            }
        }
    }

    // Print MDT in the requested format
    cout << "Macro Definition Table (MDT):" << endl;
    cout << "Index\tLine" << endl;
    int index = 0;
    for (const auto& entry : mdtEntries) {
        cout << index << "\t" << entry.second[0] << endl;
        index++;
    }
}

int main() {
    vector<string> input = {
        "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"
    };

    generateMDT(input);

    return 0;
}
```

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

```
#include <iostream>
#include <vector>
#include <map>
#include <sstream>
#include <iomanip>

using namespace std;

struct MacroDefinition {
    string name;
    int mdtIndex;
};
```

```
void generateMNT(const vector<string>& input) {
    vector<MacroDefinition> mnt;
    int mdtIndex = 0;
    bool inMacro = false;

    cout << "Macro Name Table (MNT):" << endl;
    cout << "Macro\tMDT Index" << endl;
    cout << "----------------" << endl;

    for (const auto& line : input) {
        string trimmed = line;
        // Simple trim (remove leading/trailing whitespace)
        trimmed.erase(0, trimmed.find_first_not_of(" \t"));
        trimmed.erase(trimmed.find_last_not_of(" \t") + 1);

        if (trimmed.empty()) continue;

        vector<string> tokens;
        istringstream iss(trimmed);
        string token;
        while (iss >> token) {
            tokens.push_back(token);
        }

        if (tokens.empty()) continue;

        if (tokens[0] == "MACRO") {
            if (tokens.size() > 1) {
                string macroName = tokens[1];
                // Handle macros with parameters (take just the name)
                size_t commaPos = macroName.find(',');
                if (commaPos != string::npos) {
                    macroName = macroName.substr(0, commaPos);
                }
                mnt.push_back({macroName, mdtIndex});
                inMacro = true;
            }
        }
        else if (tokens[0] == "MEND") {
            inMacro = false;
            mdtIndex++; // MEND counts as one entry
        }
        else if (inMacro) {
            mdtIndex++;
        }
    }
```

```
    // Print MNT in the requested format
    for (const auto& entry : mnt) {
        cout << entry.name << "\t" << entry.mdtIndex << endl;
    }
}

int main() {
    vector<string> input = {
        "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"
    };

    generateMNT(input);

    return 0;
}
```

(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.

```
%{
#include <stdio.h>
#include <ctype.h>
#include <string.h>

int isArticle(char *word);
int isPreposition(char *word);
int isPronoun(char *word);
int isVerb(char *word);
int isNoun(char *word);
int isAdjective(char *word);
int isAdverb(char *word);
int isConjunction(char *word);
%}

%%
[a-zA-Z]+ {
   char *word = yytext;
   printf("'%s' - ", word);

   if (isArticle(word)) {
      printf("Article\n");
   } else if (isPreposition(word)) {
      printf("Preposition\n");
   } else if (isPronoun(word)) {
      printf("Pronoun\n");
   } else if (isVerb(word)) {
      printf("Verb\n");
```

```
    } else if (isNoun(word)) {
        printf("Noun\n");
    } else if (isAdjective(word)) {
        printf("Adjective\n");
    } else if (isAdverb(word)) {
        printf("Adverb\n");
    } else if (isConjunction(word)) {
        printf("Conjunction\n");
    } else {
        printf("Unknown\n");
    }
}

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

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

. {
    printf("'%s' - Unknown symbol\n", yytext);
}
%%

int isArticle(char *word) {
    char *articles[] = {"a", "an", "the", NULL};
    for (int i = 0; articles[i] != NULL; i++) {
        if (strcasecmp(word, articles[i]) == 0) {
            return 1;
        }
    }
    return 0;
}

int isPreposition(char *word) {
```

```
    char *prepositions[] = {"in", "on", "at", "from", "to", "with", "by", "for", "of", "about", NULL};
    for (int i = 0; prepositions[i] != NULL; i++) {
        if (strcasecmp(word, prepositions[i]) == 0) {
            return 1;
        }
    }
    return 0;
}

int isPronoun(char *word) {
    char *pronouns[] = {"i", "you", "he", "she", "it", "we", "they", "me", "him", "her", "us", "them", NULL};
    for (int i = 0; pronouns[i] != NULL; i++) {
        if (strcasecmp(word, pronouns[i]) == 0) {
            return 1;
        }
    }
    return 0;
}

int isVerb(char *word) {
    char *verbs[] = {"is", "am", "are", "was", "were", "be", "have", "has", "had", "do", "does", "did", "run", "arrives",
"dread", NULL};
    for (int i = 0; verbs[i] != NULL; i++) {
        if (strcasecmp(word, verbs[i]) == 0) {
            return 1;
        }
    }
    return 0;
}

int isNoun(char *word) {
    char *nouns[] = {"destiny", "it", "same", NULL};
    for (int i = 0; nouns[i] != NULL; i++) {
        if (strcasecmp(word, nouns[i]) == 0) {
            return 1;
        }
```

```
    }
    return 0;
}

int isAdjective(char *word) {
    char *adjectives[] = {"all", "same", NULL};
    for (int i = 0; adjectives[i] != NULL; i++) {
        if (strcasecmp(word, adjectives[i]) == 0) {
            return 1;
        }
    }
    return 0;
}

int isAdverb(char *word) {
    char *adverbs[] = {"all", NULL};
    for (int i = 0; adverbs[i] != NULL; i++) {
        if (strcasecmp(word, adverbs[i]) == 0) {
            return 1;
        }
    }
    return 0;
}

int isConjunction(char *word) {
    char *conjunctions[] = {"and", "but", "or", "yet", "so", NULL};
    for (int i = 0; conjunctions[i] != NULL; i++) {
        if (strcasecmp(word, conjunctions[i]) == 0) {
            return 1;
        }
    }
    return 0;
}

int yywrap() {
```

```
   return 1;
}

int main() {
   yylex();
   return 0;
}
```

(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>
%}

DIGIT    [0-9]
LETTER   [a-zA-Z]
ID       {LETTER}({LETTER}|{DIGIT})*
NUMBER   {DIGIT}+
OPERATOR [+\-*/%=]
RELOP    [<>!=]=|==|<=|>=
PUNCT    [(){},;]

%%
```

```
"int"     { printf("KEYWORD: %s\n", yytext); }
"float"   { printf("KEYWORD: %s\n", yytext); }
"char"    { printf("KEYWORD: %s\n", yytext); }
"double"  { printf("KEYWORD: %s\n", yytext); }
"if"      { printf("KEYWORD: %s\n", yytext); }
"else"    { printf("KEYWORD: %s\n", yytext); }
"while"   { printf("KEYWORD: %s\n", yytext); }
"for"     { printf("KEYWORD: %s\n", yytext); }
"return"  { printf("KEYWORD: %s\n", yytext); }

{NUMBER}  { printf("NUMBER: %s\n", yytext); }
{ID}      { printf("IDENTIFIER: %s\n", yytext); }
{OPERATOR} { printf("OPERATOR: %s\n", yytext); }
{RELOP}   { printf("RELATIONAL OPERATOR: %s\n", yytext); }
{PUNCT}   { printf("PUNCTUATION: %s\n", yytext); }

[ \t\n]   ; /* Skip whitespace */
.         { printf("UNKNOWN: %s\n", yytext); }

%%

int yywrap() {
   return 1;
}

int main() {
   yylex();
   return 0;
}
```

(E)9. Write a program using LEX Tool, to implement a lexical analyzer for given C programming language without Symbol table. INPUT { int total =100; inti=10; printf("The value of total and i is : %d, %d", total, i); }

```
%{
#include <stdio.h>
%}

%%
"int"                 { printf("Keyword: %s\n", yytext); }
"printf"              { printf("Function: %s\n", yytext); }
[ \t\n]+              { /* Ignore whitespace */ }
[0-9]+                { printf("Number: %s\n", yytext); }
[a-zA-Z_][a-zA-Z0-9_]*  { printf("Identifier: %s\n", yytext); }
"="                   { printf("Operator: %s\n", yytext); }
";"                   { printf("Delimiter: %s\n", yytext); }
","                   { printf("Delimiter: %s\n", yytext); }
"{"                   { printf("Delimiter: %s\n", yytext); }
"}"                   { printf("Delimiter: %s\n", yytext); }
"("                   { printf("Delimiter: %s\n", yytext); }
")"                   { printf("Delimiter: %s\n", yytext); }
"\""([^"]*)"\""       { printf("String Literal: %s\n", yytext); }
.                     { printf("Unknown Token: %s\n", yytext); }
%%

int main() {
   printf("Lexical Analysis Output:\n");
   yylex();
   return 0;
}

int yywrap() {
   return 1;
}
```

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

INPUT

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

LEX FILE:expr.l

```
%{
#include "y.tab.h"
%}

%%
[0-9]+\.[0-9]+     { yylval.fval = atof(yytext); return FLOAT; }
[0-9]+             { yylval.fval = atoi(yytext); return FLOAT; }
[\t ]+             ; // Ignore whitespace
\n                 { return '\n'; }
.                  { return yytext[0]; }
%%
```

expr.y YACC file

```
%{
#include <stdio.h>
#include <stdlib.h>
%}

%token FLOAT
```

```
%left '+' '-'
%left '*' '/'
%left UMINUS

%%
input:
   | input expr '\n'     { printf("Result = %.2f\n", $2); }
   ;

expr:
     expr '+' expr       { $$ = $1 + $3; }
   | expr '-' expr       { $$ = $1 - $3; }
   | expr '*' expr       { $$ = $1 * $3; }
   | expr '/' expr       {
                         if ($3 == 0) {
                            printf("Error: Division by zero\n");
                            exit(1);
                         }
                         $$ = $1 / $3;
                        }
   | '(' expr ')'        { $$ = $2; }
   | '-' expr %prec UMINUS { $$ = -$2; }
   | FLOAT               { $$ = $1; }
   ;
%%

int main() {
   printf("Enter an expression:\n");
   yyparse();
   return 0;
}

int yyerror(const char *s) {
   printf("Parse error: %s\n", s);
```

```
    return 0;
}
```

1. In terminal, compile and run:

```
bash
CopyEdit
yacc -d expr.y
lex expr.l
gcc y.tab.c lex.yy.c -o expr_calc -lm
./expr_calc
```

(F)12. Write a program to generate three address code for the given simple expression.

INPUT

a = m * n - o – p / q

```
#include <iostream>
#include <vector>
#include <string>
using namespace std;

struct Instruction {
    string result;
    string arg1;
    string op;
    string arg2;
};

int tempCount = 1;
string newTemp() {
    return "t" + to_string(tempCount++);
```

```
}

int main() {
    // Hardcoded input
    // Expression: a = m * n - o - p / q
    vector<Instruction> code;

    // Break down the expression step-by-step:
    // Step 1: t1 = m * n
    string t1 = newTemp();
    code.push_back({t1, "m", "*", "n"});

    // Step 2: t2 = t1 - o
    string t2 = newTemp();
    code.push_back({t2, t1, "-", "o"});

    // Step 3: t3 = p / q
    string t3 = newTemp();
    code.push_back({t3, "p", "/", "q"});

    // Step 4: t4 = t2 - t3
    string t4 = newTemp();
    code.push_back({t4, t2, "-", t3});

    // Step 5: a = t4
    code.push_back({"a", t4, "=", ""});

    // Print TAC
    cout << "Three Address Code:\n";
    for (auto &inst : code) {
        if (inst.op == "=" && inst.arg2 == "")
            cout << inst.result << " = " << inst.arg1 << endl;
        else
            cout << inst.result << " = " << inst.arg1 << " " << inst.op << " "
<< inst.arg2 << endl;
    }

    return 0;
}
```