```
#include <cctype>
#include <cstdlib>
#include <iostream>
#include <fstream>
#include <map>
#include <set>
#include <sstream>
#include <string>
#include <utility>
#include <vector>

enum token {
    ID,
    IF,
    ELSE,
    DO,
    WHILE,
    BREAK,
    NUM,
    BOOL,
    LNUM,
    FALSO,
    VERDAD,
    RELOP,
    ARITHOP,
    LOGOP,
    ASSIGN,
    LPAR,
    RPAR,
    LBRACKET,
    RBRACKET,
    LCURLY,
    RCURLY,
    SEMICOLON
};

struct atributes {
    atributes(int i, std::string t)
    : index(i), type(t) {
    }

    int index;
    std::string type;
};

struct symbolsTable {
    int cont = 0;
    std::map <std::string, atributes> symbolsTable;
};

symbolsTable cteTable;
symbolsTable idTable;

void updateType (symbolsTable *st, std::string key, std::string type){
    auto bus = st->symbolsTable.find(key);

    if(bus != st->symbolsTable.end()){
        bus->second.type = type;
    }else{
```

```
        std::cout << "Error Variable no declarada UPDATE\n";
        std::exit(1);
    }
}

std::string getType (symbolsTable *st, std::string key){
    auto bus = st->symbolsTable.find(key);

    if(bus != st->symbolsTable.end()){
        return bus->second.type;
    }else{
        std::cout << "Error Variable no declarada GET\n";
        std::exit(1);
    }
}

std::map <std::pair<std::string, std::string>, std::string> tableOpeSum;
std::map <std::pair<std::string, std::string>, std::string> tableOpeMult;
std::map <std::pair<std::string, std::string>, std::string> tableOpeDiv;
std::map <std::pair<std::string, std::string>, std::string> tableOpeBool;
std::map <std::pair<std::string, std::string>, std::string> tableAssign;

std::string allowedOpeSum (std::string op1, std::string op2){
    auto bus = tableOpeSum.find(std::make_pair(op1, op2));
    if(bus != tableOpeSum.end()){
//std::cout << "Resta1: " << "op1 " << op1 << " op2 " << op2 << " res: " << bus-
>second << std::endl;
        return bus->second;
    }else{
    std::cout << "Error de Tipos Sum\n";
        std::exit(1);
    }
}

std::string allowedOpeMult (std::string op1, std::string op2){
    auto bus = tableOpeMult.find(std::make_pair(op1, op2));
    if(bus != tableOpeMult.end()){
        return bus->second;
    }else{
        std::cout << "Error de Tipos Mult\n";
        std::exit(1);
    }
}

std::string allowedOpeDiv (std::string op1, std::string op2){
    auto bus = tableOpeDiv.find(std::make_pair(op1, op2));
    if(bus != tableOpeDiv.end()){
        return bus->second;
    }else{
//std::cout << "Tipos: " << "op1 " << op1 << "op2" << op2 << std::endl;
        std::cout << "Error de Tipos Div\n";
        std::exit(1);
    }
}

std::string allowedOpeBool (std::string op1, std::string op2){
    auto bus = tableOpeBool.find(std::make_pair(op1, op2));
    if(bus != tableOpeBool.end()){
        return bus->second;
```

```
    }else{
        std::cout << "Error de Tipos Bool\n";
        std::exit(1);
    }
}

std::string allowedAssign (std::string op1, std::string op2){
    auto bus = tableAssign.find(std::make_pair(op1, op2));
    if(bus != tableAssign.end()){
        return bus->second;
    }else{
//std::cout << "Tipos: " << "op1 " << op1 << "op2" << op2 << std::endl;
        std::cout << "Error de Tipos Assign\n";
        std::exit(1);
    }
}

char getSizeType(std::string type){
    if(type == "integer"){
        return '4';
    }else if(type == "floaat"){
        return '4';
    }else if(type == "bool"){
        return '1';
    }
    return '0';
}

std::stringstream mCode;
unsigned int mTempCounter  = {0};
unsigned int mLabelCounter = {0};

void P();
void Bloque();
void Declaraciones();
void D();
void Tipo();
void Sentencias();
void S(std::string sNext);
void I();
void Sp(std::string sNext, std::string bFalse);
void WE();
std::pair<std::string, std::string> L();
std::string B(std::string bTrue, std::string bFalse);
std::string B();
std::string R();
std::string E();
std::string T();
std::string F();
void match(token t);

std::string stringToken(token t){
    switch(t){
        case ID:
            return "ID";
        case IF:
            return "IF";
        case ELSE:
            return "ELSE";
```

```
        case DO:
            return "DO";
        case WHILE:
            return "WHILE";
        case BREAK:
            return "BREAK";
        case NUM:
            return "NUM";
        case BOOL:
            return "BOOL";
        case LNUM:
            return "LNUM";
        case FALSO:
            return "FALSE";
        case VERDAD:
            return "TRUE";
        case RELOP:
            return "RELOP";
        case ARITHOP:
            return "ARITHOP";
        case LOGOP:
            return "LOGOP";
        case ASSIGN:
            return "ASSIGN";
        case LPAR:
            return "LPAR";
        case RPAR:
            return "RPAR";
        case LBRACKET:
            return "LBRACKET";
        case RBRACKET:
            return "RBRACKET";
        case LCURLY:
            return "LCURLY";
        case RCURLY:
            return "RCURLY";
        case SEMICOLON:
            return "SEMICOLON";
    }
    return "";
}

struct writeToken {
   writeToken(token w)
   : t(w) {
   }
   writeToken(token t, std::string s)
   : t(t), s(s) {
   }

   token t;
   std::string s;
};

struct DFA
{
    unsigned int startingState = { 0 };
    std::set<unsigned int> finalStates;
    std::map<std::pair<unsigned int, char>, unsigned int> transitionTable;
```

```
};

void dfaAddTransition(DFA* dfa, unsigned int state, char symbol,
                      unsigned int newState)
{
    dfa->transitionTable.emplace(std::make_pair(
        std::pair<unsigned int, char>(state, symbol), newState));
}

void dfaAddFinalState(DFA* dfa, unsigned int state)
{
    dfa->finalStates.emplace(state);
}

std::pair<int, std::string> dfaStart(DFA* dfa, std::stringstream& inputStream)
{
    std::string cad;
    unsigned int currentState = dfa->startingState;

    while(inputStream.good())
    {
        char symbol = inputStream.get();
        cad.push_back(symbol);
        symbol = (isdigit(symbol)) ? '1' : symbol;

        std::pair<unsigned int, char> transition = { currentState, symbol };

        if(dfa->transitionTable.count(transition) == 0)
        {
            inputStream.putback(symbol);
            cad.pop_back();

            if(dfa->finalStates.count(currentState) > 0)
            {
                return std::make_pair(currentState, cad);
            }
            else
            {
                return std::make_pair(-1, cad);
            }
        }

        currentState = dfa->transitionTable[transition];

        if(dfa->finalStates.count(currentState) > 0 && (symbol == '+' || symbol ==
'-'))
        {
            return std::make_pair(currentState, cad);
        }
    }

    if(dfa->finalStates.count(currentState) > 0)
    {
        return std::make_pair(currentState, cad);
    }

    return std::make_pair(-1, cad);
}
```

```
std::vector <writeToken> tokens;
auto actual = tokens.begin();

int main () {

    std::freopen("in.txt", "r", stdin);

    char str[100002];
    std::stringstream inputStream;
    std::cin.get(str, 100002, '\0');
    inputStream << str;

    std::map<std::string, token> reservedWords;
    reservedWords.emplace("if", IF);
    reservedWords.emplace("else", ELSE);
    reservedWords.emplace("do", DO);
    reservedWords.emplace("while", WHILE);
    reservedWords.emplace("break", BREAK);

    std::map<std::string, token> typesData;
    typesData.emplace("num", NUM);
    typesData.emplace("bool", BOOL);

    std::map<std::string, token> constBool;
    constBool.emplace("true", VERDAD);
    constBool.emplace("false", FALSO);

    std::map<char, token> symbolsGroup;
    symbolsGroup.emplace('(', LPAR);
    symbolsGroup.emplace(')', RPAR);
    symbolsGroup.emplace('[', LBRACKET);
    symbolsGroup.emplace(']', RBRACKET);
    symbolsGroup.emplace('{', LCURLY);
    symbolsGroup.emplace('}', RCURLY);

    DFA dfa_num;
    dfaAddTransition(&dfa_num, 0, '+', 1);
    dfaAddTransition(&dfa_num, 0, '-', 2);
    dfaAddTransition(&dfa_num, 0, '1', 5);
    dfaAddTransition(&dfa_num, 1, '1', 5);
    dfaAddTransition(&dfa_num, 2, '1', 5);
    dfaAddTransition(&dfa_num, 5, '1', 5);
    dfaAddTransition(&dfa_num, 5, '.', 6);
    dfaAddTransition(&dfa_num, 5, 'e', 8);
    dfaAddTransition(&dfa_num, 6, '1', 7);
    dfaAddTransition(&dfa_num, 7, '1', 7);
    dfaAddTransition(&dfa_num, 7, 'e', 8);
    dfaAddTransition(&dfa_num, 8, '+', 9);
    dfaAddTransition(&dfa_num, 8, '-', 9);
    dfaAddTransition(&dfa_num, 8, '1', 10);
    dfaAddTransition(&dfa_num, 9, '1', 10);
    dfaAddTransition(&dfa_num, 10, '1', 10);
    dfaAddTransition(&dfa_num, 10, '.', 11);
    dfaAddTransition(&dfa_num, 11, '1', 12);
    dfaAddTransition(&dfa_num, 12, '1', 12);
    dfaAddFinalState(&dfa_num, 1);
    dfaAddFinalState(&dfa_num, 2);
    dfaAddFinalState(&dfa_num, 5);
    dfaAddFinalState(&dfa_num, 7);
```

```
    dfaAddFinalState(&dfa_num, 8);
    dfaAddFinalState(&dfa_num, 10);
    dfaAddFinalState(&dfa_num, 12);

    DFA dfa_arithop;
    dfaAddTransition(&dfa_arithop, 0, '*', 1);
    dfaAddTransition(&dfa_arithop, 0, '/', 2);
    dfaAddTransition(&dfa_arithop, 0, '%', 3);
    dfaAddFinalState(&dfa_arithop, 1);
    dfaAddFinalState(&dfa_arithop, 2);
    dfaAddFinalState(&dfa_arithop, 3);

    DFA dfa_lograop;
    dfaAddTransition(&dfa_lograop, 0, '<', 1);
    dfaAddTransition(&dfa_lograop, 0, '>', 4);
    dfaAddTransition(&dfa_lograop, 0, '=', 7);
    dfaAddTransition(&dfa_lograop, 0, '!', 10);
    dfaAddTransition(&dfa_lograop, 0, '&', 12);
    dfaAddTransition(&dfa_lograop, 0, '|', 13);
    dfaAddTransition(&dfa_lograop, 1, '=', 2);
    dfaAddTransition(&dfa_lograop, 4, '=', 5);
    dfaAddTransition(&dfa_lograop, 7, '=', 8);
    dfaAddTransition(&dfa_lograop, 10, '=', 11);
    dfaAddTransition(&dfa_lograop, 12, '&', 14);
    dfaAddTransition(&dfa_lograop, 13, '|', 15);
    dfaAddFinalState(&dfa_lograop, 1);
    dfaAddFinalState(&dfa_lograop, 2);
    dfaAddFinalState(&dfa_lograop, 4);
    dfaAddFinalState(&dfa_lograop, 5);
    dfaAddFinalState(&dfa_lograop, 7);
    dfaAddFinalState(&dfa_lograop, 8);
    dfaAddFinalState(&dfa_lograop, 11);
    dfaAddFinalState(&dfa_lograop, 14);
    dfaAddFinalState(&dfa_lograop, 15);

    int lines = 1;
    std::pair <int, std::string> state;
    std::map<std::string, token>::iterator bus;
    std::map<char, token>::iterator busc;
    bool error = false;

    std::cout <<
"--------------------------------------------------------------------------\n";
    std::cout << "Analisis Lexico\n";

    std::vector <writeToken> printTokens;

    cteTable.symbolsTable.emplace("true", *(new atributes(cteTable.cont++,
"bool")));
    cteTable.symbolsTable.emplace("false", *(new atributes(cteTable.cont++,
"bool")));

    while(inputStream.good()){
        char c = inputStream.get();
        if(isalpha(c) || c=='_'){
            std::string word;
            do{
                word.push_back(c);
            }while((c = inputStream.get()) == '_' || isalnum(c));
```

```
            inputStream.putback(c);
            if((bus = reservedWords.find(word)) != reservedWords.end()){
                tokens.push_back({bus->second});
                printTokens.push_back({bus->second});
            }else if((bus = typesData.find(word)) != typesData.end()){
                tokens.push_back({bus->second});
                printTokens.push_back({bus->second});
            }else if((bus = constBool.find(word)) != constBool.end()){
                tokens.push_back({bus->second, word});
                printTokens.push_back({bus->second});
//std::cout << "\nNombre: " << state.second << "\n";
            }else{
                tokens.push_back({ID, word});
                printTokens.push_back({ID});
//std::cout << "\nNombre: " << word << "\n";
                idTable.symbolsTable.emplace(word, *(new atributes(idTable.cont++,
""")));
            }
            word.clear();
        }else if(isdigit(c) || c=='+' || c=='-'){
            inputStream.putback(c);
            if((state = dfaStart(&dfa_num, inputStream)).first > 0){
                if(state.first == 5){
                    tokens.push_back({LNUM, state.second});
                    printTokens.push_back({LNUM});
//std::cout << "\nNombre: " << state.second << "\n";
                    cteTable.symbolsTable.emplace(state.second, *(new
atributes(cteTable.cont++, "integer")));
                }
                else if(state.first == 7 || state.first==10 || state.first==12){
                    tokens.push_back({LNUM, state.second});
                    printTokens.push_back({LNUM});
//std::cout << "\nNombre: " << state.second << "\n";
                    cteTable.symbolsTable.emplace(state.second, *(new
atributes(cteTable.cont++, "float")));
                }
                else if (state.first == 8){
                    tokens.push_back({LNUM, state.second});
                    printTokens.push_back({LNUM});
                    inputStream.putback('e');
                    cteTable.symbolsTable.emplace(state.second, *(new
atributes(cteTable.cont++, "integer")));
                }
                else if (state.first == 1){
                    tokens.push_back({ARITHOP, "+"});
                    printTokens.push_back({ARITHOP, "+"});
                }
                else if (state.first == 2){
                    tokens.push_back({ARITHOP, "-"});
                    printTokens.push_back({ARITHOP, "-"});
                }
            }else{
                error = true;
                goto termina;
            }
        }else if(c=='*' || c=='/' || c=='%'){
            inputStream.putback(c);
            if((state = dfaStart(&dfa_arithop, inputStream)).first > 0){
                if(state.first == 1){
```

```
            tokens.push_back({ARITHOP, "*"});
            printTokens.push_back({ARITHOP, "*"});
        }
        else if(state.first == 2){
            tokens.push_back({ARITHOP, "/"});
            printTokens.push_back({ARITHOP, "/"});
        }
        else if(state.first == 3){
            tokens.push_back({ARITHOP, "%"});
            printTokens.push_back({ARITHOP, "%"});
        }
    }else{
        error = true;
        goto termina;
    }
}else if(c=='<' || c=='>' || c=='=' || c=='!' || c=='&' || c=='|'){
    inputStream.putback(c);
    if((state = dfaStart(&dfa_lograop, inputStream)).first > 0){
        if(state.first == 1){
            tokens.push_back({RELOP, "<"});
            printTokens.push_back({RELOP, "<"});
        }
        else if(state.first == 2){
            tokens.push_back({RELOP, "<="});
            printTokens.push_back({RELOP, "<="});
        }
        else if(state.first == 4){
            tokens.push_back({RELOP, ">"});
            printTokens.push_back({RELOP, ">"});
        }
        else if(state.first == 5){
            tokens.push_back({RELOP, ">="});
            printTokens.push_back({RELOP, ">="});
        }
        else if(state.first == 7){
            tokens.push_back({ASSIGN});
            printTokens.push_back({ASSIGN});
        }
        else if(state.first == 8){
            tokens.push_back({RELOP, "=="});
            printTokens.push_back({RELOP, "=="});
        }
        else if(state.first == 11){
            tokens.push_back({RELOP, "!="});
            printTokens.push_back({RELOP, "!="});
        }
        else if(state.first == 14){
            tokens.push_back({LOGOP, "&&"});
            printTokens.push_back({LOGOP, "&&"});
        }
        else if(state.first == 15){
            tokens.push_back({LOGOP, "||"});
            printTokens.push_back({LOGOP, "||"});
        }
    }else{
        error = true;
        goto termina;
    }
}else if((busc = symbolsGroup.find(c)) != symbolsGroup.end()){
```

```
            tokens.push_back({busc->second});
            printTokens.push_back({busc->second});
        }else if(c == ';'){
            tokens.push_back({SEMICOLON});
            printTokens.push_back({SEMICOLON});
        }else if(c == ' ' || c == '\n' || c == '\t'){
            do{
                if(c == '\n'){
                    if(!printTokens.empty()){
                        std::cout << lines << ' ';
                        for(auto w : printTokens){
                            std::cout << stringToken(w.t) << ':' << w.s << ' ';
                        }
                        std::cout << std::endl;
                        printTokens.clear();
                    }
                    lines++;
                }
            }while((c = inputStream.get()) == ' ' || c == '\n' || c == '\t');
            inputStream.putback(c);
        }else{
            if(inputStream.good()){
                error = true;
                goto termina;
            }
        }
    }

    termina:
    if(error){
        if(!printTokens.empty()){
            std::cout << lines << ' ';
            for(auto w : printTokens){
                std::cout << stringToken(w.t) << ':' << w.s << ' ';

            }
            std::cout << std::endl;
            std::cout << "ERROR LINE " << lines << std::endl;
        }else{
            std::cout << lines << std::endl;
            std::cout << "ERROR LINE " << lines << std::endl;
        }
    }else{
        if(!printTokens.empty()){
            std::cout << lines << ' ';
            for(auto w : printTokens){
                std::cout << stringToken(w.t) << ':' << w.s << ' ';
            }
            std::cout << std::endl;
            printTokens.clear();
            lines++;
        }
        std::cout << lines << std::endl;
    }

    std::cout <<
"\n------------------------------------------------------------------------\n";
    std::cout << "Analisis Sintactico\n";
```

```
    actual = tokens.begin();

    tableOpeSum.emplace(std::make_pair("integer", "integer"), "integer");
    tableOpeSum.emplace(std::make_pair("integer", "float"), "float");
    tableOpeSum.emplace(std::make_pair("float", "integer"), "float");
    tableOpeSum.emplace(std::make_pair("float", "float"), "float");

    tableOpeMult.emplace(std::make_pair("integer", "integer"), "integer");
    tableOpeMult.emplace(std::make_pair("integer", "float"), "float");
    tableOpeMult.emplace(std::make_pair("float", "integer"), "float");

    tableOpeDiv.emplace(std::make_pair("integer", "integer"), "integer");
    tableOpeDiv.emplace(std::make_pair("integer", "float"), "float");
    tableOpeDiv.emplace(std::make_pair("float", "integer"), "float");
    tableOpeDiv.emplace(std::make_pair("float", "float"), "float");

    tableOpeBool.emplace(std::make_pair("bool", "bool"), "bool");
    tableOpeBool.emplace(std::make_pair("integer", "integer"), "bool");
    tableOpeBool.emplace(std::make_pair("float", "float"), "bool");

    tableAssign.emplace(std::make_pair("integer", "integer"), "integer");
    tableAssign.emplace(std::make_pair("integer", "float"), "float");
    tableAssign.emplace(std::make_pair("integer", "bool"), "integer");
    tableAssign.emplace(std::make_pair("float", "integer"), "float");
    tableAssign.emplace(std::make_pair("float", "float"), "float");
    //tableAssign.emplace(std::make_pair("float", "bool"), "integer");
    //tableAssign.emplace(std::make_pair("bool", "integer"), "integer");
    //tableAssign.emplace(std::make_pair("bool", "float"), "integer");
    tableAssign.emplace(std::make_pair("bool", "bool"), "bool");

    P();

    if(actual == tokens.end()){
        std::cout << "Success\n";
    }else{
        std::cout << "Error Sintactico\n";
    }

    std::cout <<
"-------------------------------------------------------------------------\n\n";

    return 0;
}

void P(){
    Bloque();

    std::ofstream outputStream("three_address_code.txt");
    outputStream << mCode.str();
    outputStream.close();
}

void Bloque(){
    if(actual->t == LCURLY){
        match(LCURLY);
        Declaraciones();
        Sentencias();
        match(RCURLY);
    }else{
```

```
        std::cout << "Error 618\n";
        std::exit(1);
    }
}

void Declaraciones(){
    while(1){
        if(actual->t == NUM || actual->t == BOOL){
            D();
        }else if(   actual->t == ID || actual->t == IF || actual->t == WHILE ||
                    actual->t == DO || actual->t == BREAK || actual->t == LCURLY ||
actual->t == RCURLY
                    || actual == tokens.end()){
            return;
        }else{
            std::cout << "Error 632\n";
            std::exit(1);
        }
    }
}

void D(){
    Tipo();

    while(1){
        if(actual->t == ID){
            match(ID);
            match(SEMICOLON);
            return;
        }else if(actual->t == LBRACKET){
            match(LBRACKET);
            match(LNUM);
            match(RBRACKET);
            match(ID);
            match(SEMICOLON);
            return;
        }
    }
}

void Tipo(){
    if(actual->t == NUM){
        match(actual->t);
        return;
    }else if(actual->t == BOOL){
        match(actual->t);
        return;
    }else{
        std::cout << "Error Parser 669\n";
        exit(1);
    }
}

void Sentencias(){
    while(1){
        if(actual->t == RCURLY){
            return;
        }else{
            //auto sNext = "L" + std::to_string(mLabelCounter++);
```

```
            S("");
            //mCode << "\n imprimiendo \n";
            //mCode << sNext << ": ";
        }
    }
}

void S(std::string sNext){
    if(actual->t == ID){
        auto lRet = L();
        match(ASSIGN);
        auto bAddr = B();
        match(SEMICOLON);
        if(lRet.second == ""){
            mCode << lRet.first << " = " << bAddr << std::endl;
        }else{
            mCode << lRet.first << "[" << lRet.second << "]" << " = " << bAddr <<
std::endl;
        }
    }else if(actual->t == IF){
        auto sNext = "L" + std::to_string(mLabelCounter++);
        auto bTrue = "L" + std::to_string(mLabelCounter++);
        auto bFalse = "L" + std::to_string(mLabelCounter++);

        match(IF);
        match(LPAR);
        B(bTrue, bFalse);
        match(RPAR);
        mCode << bTrue << ": ";
        S(sNext);
        Sp(sNext, bFalse);
        mCode << sNext << ": ";
    }else if(actual->t == WHILE){
        auto sNext = "L" + std::to_string(mLabelCounter++);
        auto begin = "L" + std::to_string(mLabelCounter++);
        auto bTrue = "L" + std::to_string(mLabelCounter++);
        mCode << begin << ": ";
        match(WHILE);
        match(LPAR);
        B(bTrue, sNext);
        match(RPAR);
        mCode << bTrue << ": ";
        S(sNext);
        mCode << "goto " << begin << std::endl;
        mCode << sNext << ": ";
    }else if(actual->t == DO){
        auto sNext = "L" + std::to_string(mLabelCounter++);
        auto begin = "L" + std::to_string(mLabelCounter++);
        mCode << begin << ": ";
        match(DO);
        S(sNext);
        match(WHILE);
        match(LPAR);
        B(begin, sNext);
        match(RPAR);
        match(SEMICOLON);
        mCode << sNext << ": ";
    }else if(actual->t == BREAK){
        match(BREAK);
```

```
        match(SEMICOLON);
    }else{
        Bloque();
    }
}

void Sp(std::string sNext, std::string bFalse){
    if(actual->t == ELSE){
        mCode << "goto " << sNext << std::endl;
        match(ELSE);
        mCode << bFalse << ": ";
        S(sNext);
    }else if (actual->t == WHILE || actual->t ==ID || actual->t == IF || actual->t
== ELSE ||
        actual->t == DO || actual->t == BREAK || actual->t == LCURLY || actual->t
== RCURLY){
            mCode << bFalse << ": ";
            return;
    }
}

std::pair<std::string, std::string> L(){
    if(actual->t == ID){
        std::string l1Addr;
        auto w = getSizeType(getType(&idTable, actual->s));
        auto lBase = actual->s;
        match(ID);

        if(actual->t == LBRACKET){
            l1Addr = "t" + std::to_string(mTempCounter++);
            match(LBRACKET);
            auto bAddress = B();
            match(RBRACKET);
            mCode << l1Addr << " = " << w << " * " << bAddress << std::endl;
        }else if(   actual->t == ARITHOP || actual->t == RELOP || actual->t ==
LOGOP
                        || actual->t == ASSIGN || actual->t == RPAR || actual->t ==
RBRACKET
                        || actual->t == SEMICOLON){
            return std::make_pair(lBase, "");
        }

        while(1){
            auto nt = "t" + std::to_string(mTempCounter++);
            auto lAddr = "t" + std::to_string(mTempCounter++);
            if(actual->t == LBRACKET){
                match(LBRACKET);
                auto bAddress = B();
                match(RBRACKET);
                mCode << nt << " = " << w << " * " << bAddress << std::endl;
                mCode << lAddr << " = " << l1Addr << " + " << nt << std::endl;
                l1Addr = lAddr;
            }else if(   actual->t == ARITHOP || actual->t == RELOP || actual->t ==
LOGOP
                        || actual->t == ASSIGN || actual->t == RPAR || actual->t ==
RBRACKET
                        || actual->t == SEMICOLON){
                return std::make_pair(lBase, l1Addr);
            }else{
```

```
                std::cout << " 749\n";
                std::exit(1);
            }
        }
    }else{
        std::cout << "Error 754\n";
        std::exit(1);
    }
}

std::string B(std::string bTrue, std::string bFalse){
    auto r1Addr = R();
    std::string ope, rAddr, temp;

    while(1){
        if( ( (actual->t == LOGOP) && ( (actual->s == ("||")) || (actual->s ==
("&&")) ) ) ||
            ( (actual->t == RELOP) && ( (actual->s == ("==")) || (actual->s == ("!
=")) ) ) ){
           ope = actual->s;
           match(actual->t);
           temp = "t" + std::to_string(mTempCounter++);
           rAddr = R();
           //mCode << temp << " = " << r1Addr << " " << ope << " " << rAddr <<
std::endl;
        }
        else if(actual->t == RPAR || actual->t == RBRACKET || actual->t ==
SEMICOLON){
            mCode << "if " << r1Addr << " " << ope << " " << rAddr << " ";
            mCode << "goto " << bTrue << " goto " << bFalse << std::endl;
            r1Addr = temp;
            return "";
        }else{
            std::cout << "Error 771\n";
            std::exit(1);
        }
    }
}

std::string B(){
    auto r1Addr = R();

    while(1){
        if( ( (actual->t == LOGOP) && ( (actual->s == ("||")) || (actual->s ==
("&&")) ) ) ||
            ( (actual->t == RELOP) && ( (actual->s == ("==")) || (actual->s == ("!
=")) ) ) ){
           auto ope = actual->s;
           match(actual->t);
           auto temp = "t" + std::to_string(mTempCounter++);
           auto rAddr = R();
           mCode << temp << " = " << r1Addr << " " << ope << " " << rAddr <<
std::endl;
           r1Addr = temp;
        }
        else if(actual->t == RPAR || actual->t == RBRACKET || actual->t ==
SEMICOLON){
            return r1Addr;
        }else{
```

```
            std::cout << "Error 771\n";
            std::exit(1);
        }
    }
}

std::string R(){
    auto e1Addr = E();

    while(1){
        if((actual->t == RELOP) &&
            ((actual->s == ("<")) || (actual->s == ("<="))
            || (actual->s == (">")) || (actual->s == (">="))) ){
            auto ope = actual->s;
            match(actual->t);
            auto temp = "t" + std::to_string(mTempCounter++);
            auto eAddr = E();
            mCode << temp << " = " << e1Addr << " " << ope << " " << eAddr <<
std::endl;
            e1Addr = temp;
            return e1Addr;
        }
        else if(( (actual->t == LOGOP) && ( (actual->s == ("||")) || (actual->s ==
("&&")) ) ) ||
                ( (actual->t == RELOP) && ( (actual->s == ("==")) || (actual->s ==
("!=")) ) ) ||
                actual->t == RPAR || actual->t == RBRACKET || actual->t ==
SEMICOLON){
            return e1Addr;
        }else{
            std::cout << "Error 792\n" << stringToken(actual->t) << "\n" << actual-
>s << "\n";
            std::exit(1);
        }
    }
}

std::string E(){
    auto t1Addr = T();

    while(1){
        if((actual->t == ARITHOP) &&
            ((actual->s == ("+")) || (actual->s == ("-"))) ){
            auto ope = actual->s;
            match(actual->t);
            auto temp = "t" + std::to_string(mTempCounter++);
            auto tAddr = T();
            mCode << temp << " = " << t1Addr << " " << ope << " " << tAddr <<
std::endl;
            t1Addr = temp;
        }else if((actual->t == RELOP || actual->t == LOGOP || actual->t == RPAR
                || actual->t == RBRACKET || actual->t == SEMICOLON )){
            return t1Addr;
        }else{
            std::cout << "Error 810\n";
            std::exit(1);
        }
    }
}
```

```
std::string T(){
    auto f1Addr = F();

    while(1){
        if((actual->t == ARITHOP) &&
            (actual->s == ("*"))){
            auto ope = actual->s;
            match(actual->t);
            auto temp = "t" + std::to_string(mTempCounter++);
            auto fAddr = F();
            mCode << temp << " = " << f1Addr << " " << ope << " " << fAddr <<
std::endl;
            f1Addr = temp;
        }else if((actual->t == ARITHOP) &&
            (actual->s == ("/"))){
            auto ope = actual->s;
            match(actual->t);
            auto temp = "t" + std::to_string(mTempCounter++);
            auto fAddr = F();
            mCode << temp << " = " << f1Addr << " " << ope << " " << fAddr <<
std::endl;
            f1Addr = temp;
        }else if(actual->t == RELOP || actual->t == LOGOP || actual->t == RPAR
                || actual->t == RBRACKET || actual->t == SEMICOLON || ((actual->t
== ARITHOP) &&
            ((actual->s == ("+")) || (actual->s == ("-")))) ){
            return f1Addr;
        }else{
            std::cout << "Error 832\n";
            std::exit(1);
        }
    }
}

std::string F(){
    if(actual->t == LPAR){
        match(LPAR);
        auto bType = B();
        match(RPAR);
        return bType;
    }else if(actual->t == LNUM || actual->t == VERDAD || actual->t == FALSO){
        auto idLexVal = actual->s;
        match(actual->t);
        return idLexVal;
    }else{
        return L().first;
    }
}

void match(token t){
    if(actual->t == t){
        actual++;
        //std::cout << stringToken(actual->t) << "\n";
    }else{
        std::cout << "Error 858\n";
        std::exit(1);
    }
}
```

```
#include <string>
#include <iostream>
#include <sstream>

using std::cout;
using std::endl;
using std::string;
using std::to_string;
using std::stringstream;

typedef struct {
    string tag;
    string val;
} Token;

Token * look;

inline bool operator==(const Token& t, const string& s) {
    if((&t != NULL) && (t.tag == s))
        return true;
    return false;
}

typedef struct {
        int line = 1;
    char chtr = ' ';

    void iChar() {
        chtr = getchar();
    }

    bool iChar(char c) {
        iChar();
        if(chtr != c) {
            ungetc(chtr, stdin);
            return false;
        }
        chtr = ' ';
        return true;
    }

    Token* scan() {
        string numbr = "";
        do {
            iChar();
            if(chtr == ' ' || chtr == '\t') {
                continue;
            } else
            if(chtr == '\n') {
                line++;
            } else break;
        } while(chtr != EOF);
        switch(chtr) {
            case '{':   return new Token{"{", "{"};
            case '}':   return new Token{"}", "}"};
            case '[':   return new Token{"[", "["};
            case ']':   return new Token{"]", "]"};
            case '(':   return new Token{"(", "("};
            case ')':   return new Token{")", ")"};
```

```
    case '<':
        if(iChar('='))  return new Token{"<=", "<="};
        else return new Token{"<", "<"};
    case '>':
        if(iChar('='))  return new Token{"<=", ">="};
        else return new Token{">", ">"};
    case '%':   return new Token{"%", "%"};
    case ';':   return new Token{";", ";"};
    case '*':   return new Token{"*", "*"};
    case '/':   return new Token{"/", "/"};
    case '&':
        if(iChar('&'))  return new Token{"&&", "&&"};
        else return new Token{"Error", "Error"};
    case '|':
        if(iChar('|'))  return new Token{"||", "||"};
        else return new Token{"Error", "Error"};
    case '!':
        if(iChar('='))  return new Token{"!=", "!="};
        else return new Token{"Error", "Error"};
    case '=':
        if(iChar('='))  return new Token{"==", "=="};
        else return new Token{"=", "="};
    case '+':
        iChar();
        if(!isdigit(chtr)) {
            ungetc(chtr, stdin);
            return new Token{"+", "+"};
        }
        numbr += '+';
        break;
    case '-':
        iChar();
        if(!isdigit(chtr)) {
            ungetc(chtr, stdin);
            return new Token{"-", "-"};
        }
        numbr += '-';
        break;
    case EOF:
        return new Token{"EOF", "EOF"};
}

int state = 0;

if(chtr == '+' || chtr == '-' || isdigit(chtr)) {
    while(true) {
        switch(state) {
            case 0:
                if(chtr == '+' || chtr == '-')  state = 1;
                else if(chtr == '0')            state = 2;
                else if(isdigit(chtr))          state = 3;
                else {
                    ungetc(chtr, stdin);
                    return new Token{"Error", "Error"};
                }
                break;
            case 1:
                if(chtr == '0')                 state = 2;
                if(isdigit(chtr))               state = 3;
```

```
    else {
        ungetc(chtr, stdin);
        return new Token{"Error", "Error"};
    }
    break;
case 2:
    if(chtr == '.')                 state = 4;
    else if(chtr == 'e')            state = 6;
    else if(chtr == '0')          return NULL;
    else {
        ungetc(chtr, stdin);
        return new Token{"numbr", numbr};
    }
    break;
case 3:
    if(isdigit(chtr))               state = 3;
    else if(chtr == '.')            state = 4;
    else if(chtr == 'e')            state = 6;
    else  {
        ungetc(chtr, stdin);
        return new Token{"numbr", numbr};
    }
    break;
case 4:
    if(isdigit(chtr))               state = 5;
    else {
        ungetc(chtr, stdin);
        return new Token{"Error", "Error"};
    }
    break;
case 5:
    if(isdigit(chtr))               state = 5;
    else if(chtr == 'e')            state = 6;
    else  {
        ungetc(chtr, stdin);
        return new Token{"numbr", numbr};
    }
    break;
case 6:
    if(chtr == '+' || chtr == '-')  state = 7;
    else if(isdigit(chtr))          state = 8;
    else {
        ungetc(chtr, stdin);
        return new Token{"Error", "Error"};
    }
    break;
case 7:
    if(isdigit(chtr))               state = 8;
    else {
        ungetc(chtr, stdin);
        return new Token{"Error", "Error"};
    }
    break;
case 8:
    if(isdigit(chtr))               state = 8;
    else {
        ungetc(chtr, stdin);
        return new Token{"numbr", numbr};
    }
```

```
                    break;
                }
                numbr += chtr;
                iChar();
            }
        }
        if(chtr == '_' || isalpha(chtr)) {
            string s = "";
            do {
                s += chtr;
                iChar();
            } while(chtr == '_' || isalnum(chtr));
            ungetc(chtr, stdin);
            if(s == "if"   )return new Token{"if"   , s};
            if(s == "else" )return new Token{"else" , s};
            if(s == "do"   )return new Token{"do"   , s};
            if(s == "while")return new Token{"while", s};
            if(s == "break")return new Token{"break", s};
            if(s == "bool" )return new Token{"bool" , s};
            if(s == "num"  )return new Token{"num"  , s};
            if(s == "false")return new Token{"false", s};
            if(s == "true" )return new Token{"true" , s};
            return new Token{"id", s};
        }
        return new Token{"Error", "Error"};
    }
} Lxer;

Lxer lxer{};

class Parser {
    public:
        Parser();
        ~Parser();
        void parse();
    private:
        stringstream code;
        int labelCounter { 0 };
        int tmprlCounter { 0 };
        string label();
        string tmprl();
        string scode();
        string block();
        string stmts();
        string stmnt(const string& stmnt_next = string{ "" });
        //string elses();
        string assgn();
        string bexpr(
            const string& bexpr_true = string{ "" },
            const string& bexpr_flse = string{ "" });
        string expr();
        string term();
        string fctr();
        void emitc(string s);
        void emitl(string s);
        void match(string s);
        void error();
};
```

```
//unfinished
Parser::Parser() {
}

Parser::~Parser() {
}

//unfinished?
void Parser::parse() {
    look = lxer.scan();
    if(look == NULL) {
        error();
    } else {
        scode();
        if(*look == "EOF")
            //puts("SUCCESS");
            cout << code.str() << endl;
        else
            puts("ERROR");
    }
}

string Parser::label() {
    return "L" + to_string(labelCounter++);
}

string Parser::tmprl() {
    return "t" + to_string(tmprlCounter++);
}

//unfinished?
string Parser::scode() {
    block();
    return "";
}

//unfinished?
string Parser::block() {
    if(*look == "{") {
        match("{");
        //decls();
        stmts();
        match("}");
    } else error();
    return "";
}

//unfinished?
string Parser::stmts() {
    while (true) {
        if(
            *look == "if"    ||
            *look == "id"    ||
            *look == "while" ||
            *look == "do"    ||
            *look == "break" ||
            *look == "{") {
            string stmnt_next = label();
            stmnt(stmnt_next);
```

```
            emitl(stmnt_next);
        } else
        if(*look == "}") {
            return "";
        } else error();
    }
}

//unfinished?
//402
string Parser::stmnt(const string& stmnt_next) {
    if(*look == "if") {
        string bexpr_true = label();
        string bexpr_flse = label();
        match("if");
        match("(");
        bexpr(bexpr_true, bexpr_flse);
        match(")");
        emitl(bexpr_true);
        stmnt(stmnt_next);
        if(*look == "else") {
            match("else");
            emitc("goto " + stmnt_next);
            emitl(bexpr_flse);
            stmnt(stmnt_next);
        } else
        if(
            *look == "}"     ||
            *look == "if"    ||
            *look == "id"    ||
            *look == "while" ||
            *look == "do"    ||
            *look == "break" ||
            *look == "{") {
            emitl(bexpr_flse);
            return "";
        } else error();
    } else
    if(*look == "id") {
        string assgn_addr = assgn();
        match("=");
        string bexpr_addr = bexpr();
        match(";");
        emitc(assgn_addr + " = " + bexpr_addr);
    } else
    if(*look == "while") {
        string stmnt_begn = label();
        string bexpr_true = label();
        string bexpr_flse = stmnt_next;
        match("while");
        match("(");
        emitl(stmnt_begn);
        bexpr(bexpr_true, bexpr_flse);
        emitl(bexpr_true);
        match(")");
        stmnt(stmnt_begn);
        emitc("goto " + stmnt_begn);
    } else
    if(*look == "do") {
```

```
        match("do");
        stmnt();
        match("while");
        bexpr();
        match(";");
    } else
    if(*look == "break") {
        match("break");
        match(";");
    } else
    if(*look == "{") {
        block();
    } else error();
    return "";
}

//unfinished?
/*
string Parser::elses(string stmnt_next) {
    if(*look == "else") {
        match("else");
        emitc("goto " + stmnt_next);
         gen(' goto' 8. next)

        stmnt(stmnt_next);
    } else
    if(
        *look == "}"     ||
        *look == "if"    ||
        *look == "id"    ||
        *look == "while" ||
        *look == "do"    ||
        *look == "break" ||
        *look == "{") {
        return "";
    } else error();
}
*/

//unfinished?
//383
string Parser::assgn() {
    string assgn_addr;
    if(*look == "id") {
        assgn_addr = look->val;
        match("id");
        while (true) {
            if(*look == "[") {
                match("[");
                bexpr();
                match("]");
            } else
            if(
                *look == "="  ||
                *look == "*"  ||
                *look == "/"  ||
                *look == "+"  ||
                *look == "-"  ||
                *look == "<"  ||
```

```
                *look == "<=" ||
                *look == ">"  ||
                *look == ">=" ||
                *look == "==" ||
                *look == "!=" ||
                *look == "&&" ||
                *look == "||" ||
                *look == ")"  ||
                *look == ";"  ||
                *look == "]") {
                return assgn_addr;
            } else error();
        }
    } else error();
}

//unfinished?
//404
string Parser::bexpr(const string& bexpr_true, const string& bexpr_flse) {
    string exprA_addr = expr();
    string exprB_addr;
    string bexpr_addr = exprA_addr;
    while (true) {
        if(*look == "||") {
            match("||");
            if(bexpr_true != "" && bexpr_flse != "") {
                exprB_addr = expr();
                emitc("if " + exprA_addr + " || " + exprB_addr +
                    "\n\t\t" + "goto " + bexpr_true +
                    "\n\t\t" + "goto " + bexpr_flse);
            } else {
                bexpr_addr = tmprl();
                exprB_addr = expr();
                emitc(bexpr_addr + " = " + exprA_addr + " || " + exprB_addr);
            }
        } else
        if(*look == "&&") {
            match("&&");
            if(bexpr_true != "" && bexpr_flse != "") {
                exprB_addr = expr();
                emitc("if " + exprA_addr + " && " + exprB_addr +
                    "\n\t\t" + "goto " + bexpr_true +
                    "\n\t\t" + "goto " + bexpr_flse);
            } else {
                bexpr_addr = tmprl();
                exprB_addr = expr();
                emitc(bexpr_addr + " = " + exprA_addr + " && " + exprB_addr);
            }
        } else
        if(*look == "==") {
            match("==");
            if(bexpr_true != "" && bexpr_flse != "") {
                exprB_addr = expr();
                emitc("if " + exprA_addr + " == " + exprB_addr +
                    "\n\t\t" + "goto " + bexpr_true +
                    "\n\t\t" + "goto " + bexpr_flse);
            } else {
                bexpr_addr = tmprl();
                exprB_addr = expr();
```

```
        emitc(bexpr_addr + " = " + exprA_addr + " == " + exprB_addr);
    }
} else
if(*look == "!=") {
    match("!=");
    if(bexpr_true != "" && bexpr_flse != "") {
        exprB_addr = expr();
        emitc("if " + exprA_addr + " != " + exprB_addr +
            "\n\t\t" + "goto " + bexpr_true +
            "\n\t\t" + "goto " + bexpr_flse);
    } else {
        bexpr_addr = tmprl();
        exprB_addr = expr();
        emitc(bexpr_addr + " = " + exprA_addr + " != " + exprB_addr);
    }
} else
if(*look == "<") {
    match("<");
    if(bexpr_true != "" && bexpr_flse != "") {
        exprB_addr = expr();
        emitc("if " + exprA_addr + " < " + exprB_addr +
            "\n\t\t" + "goto " + bexpr_true +
            "\n\t\t" + "goto " + bexpr_flse);
    } else {
        bexpr_addr = tmprl();
        exprB_addr = expr();
        emitc(bexpr_addr + " = " + exprA_addr + " < " + exprB_addr);
    }
} else
if(*look == "<=") {
    match("<=");
    if(bexpr_true != "" && bexpr_flse != "") {
        exprB_addr = expr();
        emitc("if " + exprA_addr + " <= " + exprB_addr +
            "\n\t\t" + "goto " + bexpr_true +
            "\n\t\t" + "goto " + bexpr_flse);
    } else {
        bexpr_addr = tmprl();
        exprB_addr = expr();
        emitc(bexpr_addr + " = " + exprA_addr + " <= " + exprB_addr);
    }
} else
if(*look == ">") {
    match(">");
    if(bexpr_true != "" && bexpr_flse != "") {
        exprB_addr = expr();
        emitc("if " + exprA_addr + " > " + exprB_addr +
            "\n\t\t" + "goto " + bexpr_true +
            "\n\t\t" + "goto " + bexpr_flse);
    } else {
        bexpr_addr = tmprl();
        exprB_addr = expr();
        emitc(bexpr_addr + " = " + exprA_addr + " > " + exprB_addr);
    }
} else
if(*look == ">=") {
    match(">=");
    if(bexpr_true != "" && bexpr_flse != "") {
        exprB_addr = expr();
```

```
                emitc("if " + exprA_addr + " >= " + exprB_addr +
                    "\n\t\t" + "goto " + bexpr_true +
                    "\n\t\t" + "goto " + bexpr_flse);
            } else {
                bexpr_addr = tmprl();
                exprB_addr = expr();
                emitc(bexpr_addr + " = " + exprA_addr + " >= " + exprB_addr);
            }
        } else
        if(
            *look == ")" ||
            *look == ";" ||
            *look == "]") {
        if(bexpr_true != "" && bexpr_flse != "" && exprB_addr.empty())
            emitc("if " + bexpr_addr + " == true" +
                "\n\t\t" + "goto " + bexpr_true +
                "\n\t\t" + "goto " + bexpr_flse);
            return bexpr_addr;
        } else error();
    }
}

string Parser::expr() {
    string termA_addr = term();
    string termB_addr;
    string expr_addr = termA_addr;
    while (true) {
        if(*look == "+") {
            expr_addr = tmprl();
            match("+");
            termB_addr = term();
            emitc(expr_addr + " = " + termA_addr + " + " + termB_addr);
            termA_addr = expr_addr;
        } else
        if(*look == "-") {
            expr_addr = tmprl();
            match("-");
            termB_addr = term();
            emitc(expr_addr + " = " + termA_addr + " - " + termB_addr);
            termA_addr = expr_addr;
        } else
        if(
            *look == "<"  ||
            *look == "<=" ||
            *look == ">"  ||
            *look == ">=" ||
            *look == "==" ||
            *look == "!=" ||
            *look == "&&" ||
            *look == "||" ||
            *look == ")"  ||
            *look == ";"  ||
            *look == "]") {
            return expr_addr;
        } else error();
    }
}

string Parser::term() {
```

```
    string fctrA_addr = fctr();
    string fctrB_addr;
    string term_addr = fctrA_addr;
    while (true) {
        if(*look == "*") {
            term_addr = tmprl();
            match("*");
            fctrB_addr = fctr();
            emitc(term_addr + " = " + fctrA_addr + " * " + fctrB_addr);
            fctrA_addr = term_addr;
        } else
        if(*look == "/") {
            term_addr = tmprl();
            match("/");
            fctrB_addr = fctr();
            emitc(term_addr + " = " + fctrA_addr + " / " + fctrB_addr);
            fctrA_addr = term_addr;
        } else
        if(
            *look == "+"  ||
            *look == "-"  ||
            *look == "<"  ||
            *look == "<=" ||
            *look == ">"  ||
            *look == ">=" ||
            *look == "==" ||
            *look == "!=" ||
            *look == "&&" ||
            *look == "||" ||
            *look == ")"  ||
            *look == ";"  ||
            *look == "]") {
            return term_addr;
        } else error();
    }
}

string Parser::fctr() {
    string fctr_addr;
    if(*look == "(") {
        match("(");
        fctr_addr = bexpr();
        match(")");
    } else
    if(*look == "id") {
        fctr_addr = assgn();
    } else
    if(*look == "numbr") {
        fctr_addr = look->val;
        match("numbr");
    } else
    if(*look == "false") {
        fctr_addr = "false";
        match("false");
    } else
    if(*look == "true") {
        fctr_addr = "true";
        match("true");
    } else error();
```

```
    return fctr_addr;
}

//unfinished?
void Parser::emitc(string s) {
    code << '\t' << s << endl;
}

//unfinished?
void Parser::emitl(string s) {
    code << s << ':';
}

//unfinished?
void Parser::match(string s) {
    if(*look == s) {
        *look = *lxer.scan();
        if(look == NULL) error();
    } else error();
}

void Parser::error() {
    puts("ERROR");
    exit(EXIT_FAILURE);
}

int main() {
    Parser parser;
    parser.parse();
}
```

```
#include <map>
#include <string>

#include "parser.h"
#include "lexer.h"

int main()
{
    Lexer lexer;

    typeTable tabla_simbolos;

    lexer.init(tabla_simbolos);

    Parser parser(lexer.getTokens());

    parser.semanticEnable = false;

    parser.connect(tabla_simbolos);

    parser.P();

    if(parser.getToken().compare("$")==0)
    {
        printf("Success\n");
    }
    else
    {
        printf("Error\n");
    }
    return 0;
}
```

```
#ifndef LEXER_H
#define LEXER_H

#include <iostream>
#include <map>
#include <set>
#include <string>
#include <utility>
#include <cctype>
#include <cstdio>
#include <cstdlib>
#include <queue>

using std::make_pair;
using std::string;
using std::pair;
using std::map;
using std::set;
using std::queue;

typedef map<unsigned int,pair<string,string>> typeTable;
typedef map<pair<string,string>,string> mapss;
typedef map<string,string> mss;
typedef queue<pair<string,unsigned int>> qpss;

struct DFA
{
    unsigned int startingState = { 0 };
    set<unsigned int> finalStates;
    map<pair<unsigned int,string>,unsigned int> transitionTable;
};

class Lexer
{
public:
    Lexer();
    ~Lexer();
    void init(typeTable& tabla_simbolos);
    void printTokens();
    void printTablaSimbolos();
    qpss& getTokens();

private:
    bool dfaStart(  DFA* dfa,
                    std::istream& inputStream,
                    bool no_alpha = true);
    void dfaAddTransition(  DFA* dfa,
                            unsigned int state,
                            string symbol,
                            unsigned int newState);
    void dfaAddFinalState(  DFA* dfa,
                            unsigned int state);
    bool new_line;
    unsigned int n_linea;
    int state_save;
    char symbol;
    string Address;
    string relopAddress;
    string symbol_in;
```

```
    string SGN_ANT;
    typeTable tabla_simbolos;
    qpss bolsa_token;
};

#endif // LEXER_H
```

```
#include "lexer.h"

Lexer::Lexer(){}
Lexer::~Lexer(){}

qpss& Lexer::getTokens()
{
    return bolsa_token;
}

void Lexer::init(typeTable& tabla_simbolos)
{
    n_linea = 1;

    new_line = true;

    DFA dfa_id;
    DFA dfa_blanks;
    DFA dfa_num;
    DFA dfa_opr;
    DFA dfa_opa;
    DFA dfa_log;
    DFA dfa_agr;
    DFA dfa_scl;

    unsigned int posicion_tabla = 3;

    std::map<std::string, std::string> agrupadores;

    std::set<string> reservadas;

    tabla_simbolos.emplace(0,make_pair("NTyp","NaN"));
    tabla_simbolos.emplace(1,make_pair("BOOL","true"));
    tabla_simbolos.emplace(2,make_pair("BOOL","false"));

    reservadas.emplace("if");
    reservadas.emplace("else");
    reservadas.emplace("do");
    reservadas.emplace("while");
    reservadas.emplace("break");
    reservadas.emplace("num");
    reservadas.emplace("bool");
    reservadas.emplace("true");
    reservadas.emplace("false");

    agrupadores.emplace("{","LCURLY");
    agrupadores.emplace("}","RCURLY");
    agrupadores.emplace("(","LPAR");
    agrupadores.emplace(")","RPAR");
    agrupadores.emplace("[","LBRACKET");
    agrupadores.emplace("]","RBRACKET");

    dfaAddTransition(&dfa_blanks, 0, "\t", 1);
    dfaAddTransition(&dfa_blanks, 0, " " , 1);
    dfaAddTransition(&dfa_blanks, 1, "\t", 1);
    dfaAddTransition(&dfa_blanks, 1, " " , 1);

    dfaAddFinalState(&dfa_blanks, 1);
```

```
dfaAddTransition(&dfa_id,0,"letra",1);
dfaAddTransition(&dfa_id,0,  "_"  ,1);
dfaAddTransition(&dfa_id,1,  "_"  ,1);
dfaAddTransition(&dfa_id,1,"letra",1);
dfaAddTransition(&dfa_id,1,"digit",1);

dfaAddFinalState(&dfa_id, 1);

dfaAddTransition(&dfa_num,0,  "-"  ,1);
dfaAddTransition(&dfa_num,0,  "+"  ,1);
dfaAddTransition(&dfa_num,0,"digit",2);
dfaAddTransition(&dfa_num,1,"digit",2);
dfaAddTransition(&dfa_num,2,"digit",2);
dfaAddTransition(&dfa_num,2,  "."  ,3);
dfaAddTransition(&dfa_num,2,  "e"  ,5);
dfaAddTransition(&dfa_num,3,"digit",4);
dfaAddTransition(&dfa_num,4,"digit",4);
dfaAddTransition(&dfa_num,4,  "e"  ,5);
dfaAddTransition(&dfa_num,5,  "-"  ,6);
dfaAddTransition(&dfa_num,5,  "+"  ,6);
dfaAddTransition(&dfa_num,5,"digit",7);
dfaAddTransition(&dfa_num,6,"digit",7);
dfaAddTransition(&dfa_num,7,"digit",7);

dfaAddFinalState(&dfa_num, 1);
dfaAddFinalState(&dfa_num, 2);
dfaAddFinalState(&dfa_num, 4);
dfaAddFinalState(&dfa_num, 7);

dfaAddTransition(&dfa_opr,0, "<", 1);
dfaAddTransition(&dfa_opr,0, ">", 4);
dfaAddTransition(&dfa_opr,0, "=", 7);
dfaAddTransition(&dfa_opr,0, "!",10);
dfaAddTransition(&dfa_opr,1, "=", 2);
dfaAddTransition(&dfa_opr,4, "=", 5);
dfaAddTransition(&dfa_opr,7, "=", 8);
dfaAddTransition(&dfa_opr,10,"=",11);

dfaAddFinalState(&dfa_opr, 1);
dfaAddFinalState(&dfa_opr, 2);
dfaAddFinalState(&dfa_opr, 4);
dfaAddFinalState(&dfa_opr, 5);
dfaAddFinalState(&dfa_opr, 7);
dfaAddFinalState(&dfa_opr, 8);
dfaAddFinalState(&dfa_opr,11);

//dfaAddTransition(&dfa_opa,0, "+", 1);
//dfaAddTransition(&dfa_opa,0, "-", 1);
dfaAddTransition(&dfa_opa,0, "*", 1);
dfaAddTransition(&dfa_opa,0, "%", 1);
dfaAddTransition(&dfa_opa,0, "/", 1);

dfaAddFinalState(&dfa_opa, 1);

dfaAddTransition(&dfa_log,0, "&", 1);
dfaAddTransition(&dfa_log,1, "&", 2);
dfaAddTransition(&dfa_log,0, "|", 3);
dfaAddTransition(&dfa_log,3, "|", 4);
```

```
dfaAddFinalState(&dfa_log, 2);
dfaAddFinalState(&dfa_log, 4);

dfaAddTransition(&dfa_agr,0, "(", 1);
dfaAddTransition(&dfa_agr,0, ")", 1);
dfaAddTransition(&dfa_agr,0, "[", 1);
dfaAddTransition(&dfa_agr,0, "]", 1);
dfaAddTransition(&dfa_agr,0, "{", 1);
dfaAddTransition(&dfa_agr,0, "}", 1);

dfaAddFinalState(&dfa_agr, 1);

dfaAddTransition(&dfa_scl,0, ";", 1);

dfaAddFinalState(&dfa_scl, 1);


auto toUpper = []( std::string str ){
    std::string ret = "";
    std::locale loc;
    for (std::string::size_type i = 0; i < str.length(); ++i)
    {
        ret += std::toupper(str[i],loc);
    }
    return ret;
};

std::istream& inputStream(std::cin);

std::map<std::string,unsigned int> tab_norep;
while(inputStream.good())
{
    if(new_line)
    {
        new_line = false;
        dfaStart(&dfa_blanks,inputStream);
        symbol = inputStream.get();
        if (symbol == '\n')
        {
            n_linea++;
            new_line = true;
            continue;
        }
        else
        {
            inputStream.putback(symbol);
        }
    }
    dfaStart(&dfa_blanks,inputStream);

    if ( dfaStart(&dfa_id, inputStream) )
    {
        if(reservadas.count(Address) > 0)
        {
            if( Address.compare("true")==0 )
            {
                bolsa_token.emplace(toUpper(Address),1);
            }
```

```
                else if ( Address.compare("false")==0 )
                {
                    bolsa_token.emplace(toUpper(Address),2);
                }
                else
                {
                    bolsa_token.emplace(toUpper(Address),0);
                }
            }
            else
            {
                if ( tab_norep.count(Address) > 0)
                {
                    bolsa_token.emplace("ID",tab_norep[Address]);
                }
                else
                {
                    tab_norep.emplace(Address,posicion_tabla);
                    tabla_simbolos.emplace(posicion_tabla,make_pair("",Address));
                    bolsa_token.emplace("ID",posicion_tabla++);
                }
            }
        }
        else if ( dfaStart(&dfa_num, inputStream,false) )
        {
            if(SGN_ANT.compare("-")==0)
            {
                bolsa_token.emplace("OP_RES",0);
            }
            else if(SGN_ANT.compare("+")==0)
            {
                bolsa_token.emplace("OP_SUM",0);
            }
            else
            {
                if (state_save > 2)
                {

tabla_simbolos.emplace(posicion_tabla,make_pair("FLOAT",Address));
                    bolsa_token.emplace("LNUM",posicion_tabla++);
                }
                else
                {

tabla_simbolos.emplace(posicion_tabla,make_pair("INT",Address));
                    bolsa_token.emplace("LNUM",posicion_tabla++);
                }
            }
        }
        else if ( dfaStart(&dfa_opr, inputStream) )
        {
            if (SGN_ANT.compare("=") == 0 && state_save == 0)
            {
                bolsa_token.emplace("ASSIGN",0);
            }
            else
            {
                tabla_simbolos.emplace(
                    posicion_tabla,make_pair("RELOP",relopAddress));
```

```
                bolsa_token.emplace("RELOP",posicion_tabla++);
            }
        }
        else if ( dfaStart(&dfa_opa, inputStream) )
        {
            if (SGN_ANT.compare("*")==0)
            {
                bolsa_token.emplace("OP_MUL",0);
            }
            else if (SGN_ANT.compare("/")==0)
            {
                bolsa_token.emplace("OP_DIV",0);
            }
        }
        else if ( dfaStart(&dfa_log, inputStream) )
        {
            tabla_simbolos.emplace(
                posicion_tabla,make_pair("LOGOP",relopAddress));
            bolsa_token.emplace("LOGOP",posicion_tabla++);
        }
        else if ( dfaStart(&dfa_agr , inputStream) )
        {
            bolsa_token.emplace(agrupadores[SGN_ANT],0);
        }
        else if ( dfaStart(&dfa_scl, inputStream) )
        {
            bolsa_token.emplace("SEMICOLON",0);
        }
        else if ( inputStream.get() == '\n' )
        {
            n_linea++;
            new_line = true;
        }
        else if ( inputStream.get() != -1 )
        {
            std::cout << "Error lexico en linea: " << n_linea << std::endl;
            exit(1);
        }
    }

    bolsa_token.emplace("$",0);
}

void Lexer::printTokens()
{
    while(!bolsa_token.empty())
    {
        std::cout << bolsa_token.front().first << '\n';

        bolsa_token.pop();
    }
}

void Lexer::printTablaSimbolos()
{
    for(auto i : tabla_simbolos)
    {
        std::cout << i.first << " Type: " << i.second.first
                    << " Address: " << i.second.second << '\n';
```

```
    }
}

void Lexer::dfaAddTransition(DFA* dfa, unsigned int state, std::string symbol,
                      unsigned int newState)
{
    dfa->transitionTable.emplace(std::make_pair(
        std::pair<unsigned int, std::string >(state, symbol), newState));
}

void Lexer::dfaAddFinalState(DFA* dfa, unsigned int state)
{
    dfa->finalStates.emplace(state);
}

bool Lexer::dfaStart(DFA* dfa, std::istream& inputStream, bool no_alpha)
{
    unsigned int currentState = dfa->startingState;

    Address = "";
    relopAddress = "";
    state_save = 0;

    while(inputStream.good())
    {
        symbol = inputStream.get();

        if ( isalpha(symbol) > 0 && no_alpha)
        {
            symbol_in = "letra"; Address+=symbol;
        }
        else if ( isdigit(symbol) > 0 )
        {
            symbol_in = "digit"; Address+=symbol;
        }
        else if ( symbol == '.' )
        {
            symbol_in = symbol; Address+=symbol;
        }
        else
        {
            if (symbol!=' ') relopAddress+=symbol;
            symbol_in = symbol;
        }

        std::pair<unsigned int, std::string> transition = {currentState,
symbol_in};

        if(dfa->transitionTable.count(transition) == 0 )
        {
            inputStream.putback(symbol);

            if(dfa->finalStates.count(currentState) > 0 )
            {
                return true;
            }
            else
            {
                return false;
```

```
            }
        }
        state_save = currentState;
        SGN_ANT = symbol;

        currentState = dfa->transitionTable[transition];
    }

    if(dfa->finalStates.count(currentState) > 0)
    {
        return true;
    }
    return false;
}
```

```
#ifndef PARSER_H
#define PARSER_H

#include <string>
#include <fstream>
#include <cstdlib>
#include <iostream>
#include <sstream>
#include <utility>
#include <queue>
#include <cctype>
#include <map>
#include <set>

using std::string;
using std::pair;
using std::queue;

typedef std::map< unsigned int , std::pair<string,string> > typeTable;
typedef std::pair<std::string,std::string> pss;
typedef std::set<std::pair<std::string,std::string>> spss;
typedef std::queue< std::pair<std::string,unsigned int>> qpss;
typedef std::map<std::pair<std::string,std::string>,std::string> mapss;
typedef std::map<std::string,std::string> mss;

#define debug_semantic(X,Y) std::cout<<"Type(1): "<<X<<" Type(2): "<<Y<<'\n'

class Parser
{
public:
      Parser();
      Parser(qpss& t);
      ~Parser();

      string getToken() const;
      void P();
      void connect(typeTable& ts);
      void printTablaSimbolos();

      bool semanticEnable;

private:
      void        Bloque();
      void Declaraciones();
      void             D();
      pss           Tipo();
      void    Sentencias();
      void             S(const string& snext = std::string{""});
      void            Sp(const string& snext = std::string{""},const string& bfalse
= std::string{""});
      pss                L();
      pss                B(const std::string& btrue = std::string{""}, const
std::string& bfalse=std::string{""});
      pss                R(const std::string& btrue = std::string{""}, const
std::string& bfalse=std::string{""});
      pss                E();
      pss                T();
      pss                F();
```

```
      unsigned int mTempCounter  = {0};
    unsigned int mLabelCounter = {0};

      void match(string cmp);
      void avanzarEntrada();

      qpss bolsa_token;
      pair<string,unsigned int> token;

      std::stringstream mCode;

      mapss tabla_inferencia_Mult;
      mapss tabla_inferencia_Arith;
      mapss tabla_inferencia_Relop;
    mapss tabla_inferencia_Logop;
    bool lvalue;

    typeTable tabla_simbolos;

    spss tabla_asign_valida;
};

#endif // PARSER_H
```

```
#include <string>
#include <utility>
#include <cstdio>
#include <cstdlib>

#include "parser.h"

using std::pair;
using std::string;
using std::make_pair;
using std::ofstream;
using std::to_string;

Parser::Parser(){}
Parser::~Parser(){}

void Parser::connect(typeTable& ts)
{
      tabla_simbolos = ts;
}

void Parser::avanzarEntrada()
{
      token = bolsa_token.front();
    bolsa_token.pop();
}

string Parser::getToken() const
{
      return token.first;
}

void Parser::P()
{
    if(token.first.compare("LCURLY") == 0)
    {
        Bloque();
    }
    else if (token.first.compare("$") == 0 )
    {

        return;
    }
    else
    {
        printf("Syntax error P\n");
        exit(1);
    }
}

void Parser::Bloque()
{
      if(token.first.compare("LCURLY") == 0 )
      {
            match("LCURLY");
            Declaraciones();

            Sentencias();
```

```
            ofstream outputStream("three_address_code.txt");
            outputStream << mCode.str();
            outputStream.close();
            match("RCURLY");
      }
      else
      {
            printf("Syntax error Bloque\n");
            exit(1);
      }
}

void Parser::Declaraciones()
{
      while(1)
      {
            if (token.first.compare("NUM")==0 || token.first.compare("BOOL")==0)
            {
                  D();
            }
            else if(    token.first.compare("IF")==0     ||
                              token.first.compare("RCURLY")==0 ||
                              token.first.compare("DO")==0     ||
                              token.first.compare("LCURLY")==0 ||
                        token.first.compare("BREAK")==0  ||
                        token.first.compare("WHILE")==0  ||
                        token.first.compare("ID")==0      )
            {
                  return;
            }
            else
            {
                  printf("Syntax error Declaraciones\n");
                  exit(1);
            }
      }
}

void Parser::D()
{

      pss retTipo = Tipo();

      if(token.first.compare("ID")==0)
      {
            tabla_simbolos[token.second].first = retTipo.first;
            match("ID");
            match("SEMICOLON");
      }
      else if (token.first.compare("LBRACKET")==0)
      {
            match("LBRACKET");
            match("LNUM");
            match("RBRACKET");
            tabla_simbolos[token.second].first = retTipo.first;
            match("ID");
            match("SEMICOLON");
      }
```

```
	else if(	token.first.compare("WHILE")==0  ||
				token.first.compare("LCURLY")==0 ||
				token.first.compare("DO")==0     ||
				token.first.compare("BREAK")==0  ||
				token.first.compare("IF")==0     ||
				token.first.compare("ID")==0     ||
				token.first.compare("NUM")==0    ||
				token.first.compare("BOOL")==0    )
	{
		return;
	}
	else
	{
		printf("Syntax error D\n");
		exit(1);
	}
}

pss Parser::Tipo()
{
	pss retTipo;

	if(token.first.compare("NUM")==0 || token.first.compare("BOOL")==0)
	{
		retTipo.first = token.first;
		match(token.first);
	}
	else
	{
		printf("Syntax error Tipo\n");
		exit(1);
	}
	return retTipo;
}

void Parser::Sentencias()
{
	while(1)
	{
		if (  token.first.compare("BREAK")==0 ||
				token.first.compare("DO")==0    ||
				token.first.compare("WHILE")==0 ||
				token.first.compare("IF")==0  ||
				token.first.compare("ID")==0    ||
				token.first.compare("LCURLY")==0 )
		{
			string snext = "L" + to_string(mLabelCounter);
		mLabelCounter++;

			S(snext);

			//mCode << snext +": "; se omite

		}
		else if (token.first.compare("RCURLY")==0)
		{
			return;
		}
		else
```

```
            {
                  printf("Syntax error Sentencias\n");
                  exit(1);
            }
      }
}

void Parser::S(const string& snext)
{
      //S->L=B;
      if(token.first.compare("ID")==0)
      {

            unsigned int id_addr = token.second;

            lvalue = true;
            pss retL = L();

            match("ASSIGN");
            lvalue = false;
            pss retB = B();

            mCode  << retL.second << "=" << retB.second << '\n';

            if ( tabla_asign_valida.count(make_pair(retL.first,retB.first)) == 0
                        && semanticEnable)
            {
                  if (retL.first.compare("")==0)
                  {
                        printf("Variable no declarada\n");
                  }
                  printf("Syntax error semantico S\n");
                  exit(1);
            }
            else
            {
                  tabla_simbolos[id_addr].first = retB.first;
            }
            match("SEMICOLON");
      }
      //S->if(B)SS'
      else if(token.first.compare("IF")==0)
      {
            string bfalse = "L" + to_string(mLabelCounter);
            mLabelCounter++;

            string btrue = "L" + to_string(mLabelCounter);
            mLabelCounter++;

            match("IF");
            match("LPAR");
            B(btrue,bfalse);
            match("RPAR");
            mCode << btrue << ": ";
            S(snext);
            Sp(snext,bfalse);

      }
      //S->while(B)S
```

```
	else if (token.first.compare("WHILE")==0)
	{
		string begin = "L" + to_string(mLabelCounter);
	  mLabelCounter++;
	  string btrue = "L" + to_string(mLabelCounter);
	  mLabelCounter++;

		match("WHILE");
		match("LPAR");

		mCode << begin << ": ";
		B(btrue,snext);
		match("RPAR");
		mCode << btrue << ": ";
		S(begin);
		mCode << "goto " + begin << '\n';
		mCode << snext << ": ";
	}
	//S->do S while(B);
	else if( token.first.compare("DO")==0)
	{
		match("DO");
		S();
		match("WHILE");
		match("LPAR");
		B();
		match("RPAR");
		match("SEMICOLON");
	}
	//S->BREAK;
	else if(token.first.compare("BREAK")==0)
	{
		match("BREAK");
		match("SEMICOLON");
	}
	//S->Bloque
	else if(token.first.compare("LCURLY")==0)
	{
		Bloque();
	}
	else if (token.first.compare("RCURLY")==0)
	{
		return;
	}
	else
	{
		printf("Syntax error S\n");
		exit(1);
	}
}

void Parser::Sp(const string& snext,const string& bfalse)
{
	if(token.first.compare("ELSE")==0)
	{
		mCode << "goto "+snext << ": \n";
		mCode << bfalse << ": ";
		match("ELSE");
		S(snext);
```

```
            mCode << snext << ": ";
      }
      else if(    token.first.compare("ID")==0     ||
                  token.first.compare("LCURLY")==0 ||
                  token.first.compare("DO")==0   ||
                  token.first.compare("BREAK")==0  ||
                  token.first.compare("IF")==0     ||
                  token.first.compare("WHILE")==0  ||
            token.first.compare("RCURLY")==0  )
      {
            mCode << bfalse << ": ";
            return;
      }
      else
      {
            printf("Syntax error Sp\n");
            exit(1);
      }
}

pss Parser::L()
{
      pss retL = tabla_simbolos[token.second];

      match("ID");

      string lbase = retL.second;
      string ltype = "Type("+lbase+")";

      string tmp{""};

      string tmpRet{""};

      bool uno = true;
      unsigned int contLevel = 0;
      while(1)
      {
            if (token.first.compare("LBRACKET")==0)
            {
                  contLevel++;
                  tmp = "t" + to_string(mTempCounter);
            mTempCounter++;
                  match("LBRACKET");
                  pss retB = B();
                  match("RBRACKET");

                  mCode << tmp << "=" << "w("+ltype+")*"+retB.second+"\n";
                  if (uno){tmpRet += tmp;uno=false;}
                  else tmpRet+="+"+tmp;
                  retL.second=tmpRet;
            }
            else if( token.first.compare("OP_MUL")==0    ||
token.first.compare("OP_DIV")==0   ||
                        token.first.compare("LOGOP")==0     ||
token.first.compare("ASSIGN")==0   ||
                        token.first.compare("RBRACKET")==0  ||
token.first.compare("RELOP")==0    ||
                        token.first.compare("SEMICOLON")==0 ||
token.first.compare("OP_RES")==0   ||
```

```
                        token.first.compare("OP_SUM")==0    ||
token.first.compare("RPAR")==0           )
            {
                  if ( contLevel > 1)
                  {
                        string tmp1 = "t" + to_string(mTempCounter);
                  mTempCounter++;
                        if (lvalue)
                        {
                              mCode << tmp1 << "=" << retL.second <<'\n';
                              retL.second = lbase+"["+tmp1+"]";
                        }
                        else
                        {
                              string tmp2 = "t" + to_string(mTempCounter);
                        mTempCounter++;
                        mCode << tmp1 << "=" << retL.second<<'\n';
                        mCode << tmp2 << "=" << lbase+"["+tmp1+"]" <<'\n';
                        retL.second = tmp2;
                        }
                  }
                  else if( contLevel == 1)
                  {
                        retL.second = lbase+"["+tmp+"]";
                  }
                  return retL;
            }
            else
            {
                  printf("Syntax error L\n");
                  exit(1);
            }
      }
}

pss Parser::B(const std::string& btrue, const std::string& bfalse)
{
      pss retR = R(btrue,bfalse);

      while(1)
      {
            if(token.first.compare("LOGOP")==0)
            {
                  string AddressRelop = tabla_simbolos[token.second].second;

                  match("LOGOP");

                  pss retR1 = R(btrue,bfalse);

                  mCode << "if " << retR.second << AddressRelop
                  << retR1.second << " goto " + btrue << "\ngoto " + bfalse <<'\n';

                  if
( tabla_inferencia_Logop.count(make_pair(retR.first,retR1.first))==0
                              && semanticEnable )
                  {
                        printf("Error semantico B\n");
                        exit(1);
                  }
```

```
                  retR.first =
tabla_inferencia_Logop[make_pair(retR.first,retR1.first)];
            }
            else if(    token.first.compare("RPAR")==0           ||
                        token.first.compare("RBRACKET")==0  ||
                        token.first.compare("SEMICOLON")==0 )
            {
                  return retR;
            }
            else
            {
                  printf("Syntax error B\n");
                  exit(1);
            }
      }
}

pss Parser::R(const std::string& btrue, const std::string& bfalse)
{
      pss retE = E();

      while(1)
      {
            if (token.first.compare("RELOP")==0 )
            {
                  string AddressRelop = tabla_simbolos[token.second].second;

                  match("RELOP");

                  pss retE1 = E();

                  mCode << "if " << retE.second << AddressRelop
                  << retE1.second << " goto " + btrue << "\ngoto " + bfalse <<'\n';

                  if
( tabla_inferencia_Relop.count(make_pair(retE.first,retE1.first)) == 0
                              && semanticEnable )
                  {
                        if ( retE.first.compare("NUM")==0 ||
retE1.first.compare("NUM")==0 )
                        {
                              printf("Variable no inicializada\n");
                        }
                        printf("Syntax error semantico R\n");
                        exit(1);
                  }
                  retE.first =
tabla_inferencia_Relop[make_pair(retE.first,retE1.first)];
            }
            else if( token.first.compare("RPAR")==0    ||
                        token.first.compare("LOGOP")==0     ||
                        token.first.compare("RBRACKET")==0  ||
                        token.first.compare("SEMICOLON")==0 )
            {
                  return retE;
            }
            else
            {
                  printf("Syntax error R\n");
```

```
                  exit(1);
            }
      }
}

pss Parser::E()
{
      char signo;
      pss retT = T();

      while(1)
      {
            if(token.first.compare("OP_SUM")==0 || token.first.compare("OP_RES")==0
)
            {
                  string temp = "t" + to_string(mTempCounter);
            mTempCounter++;

            signo = (token.first.compare("OP_SUM")==0)?'+':'-';
                  match(token.first);

                  pss retT1 = T();

                  mCode << temp << "=" << retT.second << signo << retT1.second <<
'\n';

            retT.second = temp;

                  if
( tabla_inferencia_Arith.count(make_pair(retT.first,retT1.first)) == 0
                              && semanticEnable )
                  {
                        printf("Error semantico E\n");
                        exit(1);
                  }
                  retT.first =
tabla_inferencia_Arith[make_pair(retT.first,retT1.first)];
            }
            else if (   token.first.compare("LOGOP")==0     ||
                              token.first.compare("RELOP")==0           ||
                              token.first.compare("RPAR")==0      ||
                              token.first.compare("SEMICOLON")==0||
                              token.first.compare("RBRACKET")==0   )
            {
                  return retT;
            }
            else
            {
                  printf("Syntax error E\n");
                  exit(1);
            }
      }
}

pss Parser::T()
{
      pss retF = F();

      while(1)
```

```
      {
            if(token.first.compare("OP_MUL")==0)
            {
                  string temp = "t" + to_string(mTempCounter);
            mTempCounter++;

                  match(token.first);

                  pss retF1 = F();

                  mCode << temp << "=" << retF.second << "*" << retF1.second <<
'\n';

            retF.second = temp;

                  if
(tabla_inferencia_Mult.count(std::make_pair(retF.first,retF1.first))==0
                              && semanticEnable)
                  {
                        if (retF.first.compare("FLOAT")==0 &&
retF1.first.compare("FLOAT")==0)
                        {
                              printf("Operacion no permitida\n");
                        }
                        printf("Error semantico T1\n");
                        exit(1);
                  }
                  retF.first =
tabla_inferencia_Mult[std::make_pair(retF.first,retF1.first)];
            }
            else if(token.first.compare("OP_DIV")==0 )
            {
                  string temp = "t" + to_string(mTempCounter);
            mTempCounter++;

                  match(token.first);

                  pss retF1 = F();

                  mCode << temp << "=" << retF.second << "/" << retF1.second <<
'\n';

            retF.second = temp;

                  if
( tabla_inferencia_Arith.count(std::make_pair(retF.first,retF1.first)) == 0
                              && semanticEnable)
                  {
                        printf("Error semantico T2\n");
                        exit(1);
                  }
                  retF.first =
tabla_inferencia_Arith[std::make_pair(retF.first,retF1.first)];
            }
            else if(
                              token.first.compare("SEMICOLON")==0  ||
                              token.first.compare("RBRACKET")==0   ||
                              token.first.compare("OP_SUM")==0    ||
                              token.first.compare("OP_RES")==0     ||
```

```
                              token.first.compare("LOGOP")==0      ||
                              token.first.compare("RELOP")==0      ||
                              token.first.compare("RPAR")==0       )
            {

                  return retF;
            }
            else
            {
                  printf("Syntax error T\n");
                  exit(1);
            }
      }
}

pss Parser::F()
{
      pss retF;

      if (token.first.compare("LPAR") == 0)
      {
            match("LPAR");
            retF = B();
            match("RPAR");
      }
      else if (token.first.compare("ID")==0)
      {
            retF = L();
      }
      else if (token.first.compare("LNUM")==0  ||
                  token.first.compare("TRUE")==0  ||
                  token.first.compare("FALSE")==0  )
      {
            retF = tabla_simbolos[token.second];
            match(token.first);
      }
      else
      {
            printf("Syntax error F\n");
            exit(1);
      }
      return retF;
}
Parser::Parser(qpss& t)
{
      bolsa_token = t;
      token = bolsa_token.front();
    bolsa_token.pop();

    tabla_inferencia_Arith.emplace(
      make_pair("INT","INT"),"INT");
    tabla_inferencia_Arith.emplace(
      make_pair("INT","FLOAT"),"FLOAT");
    tabla_inferencia_Arith.emplace(
      make_pair("FLOAT","INT"),"FLOAT");
    tabla_inferencia_Arith.emplace(
      make_pair("FLOAT","FLOAT"),"FLOAT");

    tabla_inferencia_Mult.emplace(
```

```
      make_pair("INT","INT"),"INT");
    tabla_inferencia_Mult.emplace(
      make_pair("INT","FLOAT"),"FLOAT");
    tabla_inferencia_Mult.emplace(
      make_pair("FLOAT","INT"),"FLOAT");

    tabla_inferencia_Logop.emplace(
      make_pair("BOOL","BOOL"),"BOOL");

    tabla_inferencia_Relop.emplace(
      make_pair("INT","INT"),"BOOL");
    tabla_inferencia_Relop.emplace(
      make_pair("FLOAT","FLOAT"),"BOOL");
    tabla_inferencia_Relop.emplace(
      make_pair("INT","FLOAT"),"BOOL");
    tabla_inferencia_Relop.emplace(
      make_pair("FLOAT","INT"),"BOOL");

    tabla_asign_valida.emplace(
      make_pair("INT","INT"));
    tabla_asign_valida.emplace(
      make_pair("FLOAT","FLOAT"));
    tabla_asign_valida.emplace(
      make_pair("INT","FLOAT"));
    tabla_asign_valida.emplace(
      make_pair("FLOAT","INT"));
    tabla_asign_valida.emplace(
      make_pair("NUM","INT"));
    tabla_asign_valida.emplace(
      make_pair("NUM","FLOAT"));
    tabla_asign_valida.emplace(
      make_pair("BOOL","BOOL"));
}

void Parser::match(string cmp)
{
      if(token.first.compare(cmp)==0)
      {
            avanzarEntrada();
      }
      else
      {
            printf("Error match\n");
            exit(1);
      }
}

void Parser::printTablaSimbolos()
{
    for(auto i : tabla_simbolos)
    {
        std::cout << i.first << " Type:" << i.second.first
                  << "\t| Address: " << i.second.second << '\n';
    }
}
```