Міністерство освіти і науки, молоді та спорту України

Національний Технічний Університет України

“Київський Політехнічний Інститут”

Факультет прикладної математики

Кафедра СПіСКС

Розрахунково-графічна робота

з дисципліни

“Основи проектування трансляторів”

Тема: “ РОЗРОБКА СИНТАКТИЧНОГО АНАЛІЗАТОРА”

Варіант 21

Виконав:

Студент групи КВ-71

Янечко А.С.

**Варіант 21**

1. <signal-program> --> <program>

2. <program> --> PROGRAM <procedure-identifier> ; <block>.

3. <block> --> <variable-declarations> BEGIN <statements-list> END

4. <variable-declarations> --> VAR <declarationslist> | <empty>

5. <declarations-list> --> <declaration> <declarations-list> | <empty>

6. <declaration> --><variableidentifier>:<attribute><attributeslist> ;

7. <attributes-list> --> <attribute> <attributeslist> | <empty>

8. <attribute> --> INTEGER | FLOAT | [<range>]

9. <range> --> <unsigned-integer> .. <unsignedinteger>

10. <statements-list> --> <statement> <statementslist> | <empty>

11. <statement> --> <variable> := <expression> ; | LOOP <statements-list> ENDLOOP ;

12. <expression> --> <variable> | <unsigned-integer>

13. <variable> --> <variable-identifier><dimension>

14. <dimension> --> [ <expression> ] | <empty>

15. <variable-identifier> --> <identifier>

16. <procedure-identifier> --> <identifier>

17. <identifier> --> <letter><string>

18. <string> --> <letter><string> | <digit><string> | <empty>

19. <unsigned-integer> --> <digit><digits-string>

20. <digits-string> --> <digit><digits-string> | <empty>

21. <digit> --> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

22. <letter> --> A | B | C | D | ... | Z

**Лістинг програми**

**OPT\_Lexer.cpp**

#include <iostream>

#include "Lexer.h"

#include "SyntaxAnalyzer.h"

// #include "Translator.h"

#include <vector>

#include <string>

std::string getOutputPath(std::string& inputPath, std::string);

void start(std::string& path);

void generateTestFiles(int count);

std::string getNumberOfTest(int number);

int main(int argc, char\*\* argv) {

if (argc < 2 || argv[1] == nullptr) {

std::cout << "Wrong path!" << std::endl;

exit(0);

}

std::cout << "PATH: " << argv[1] << std::endl;

std::string path = argv[1];

start(path);

//generateTestFiles(14);

}

void generateTestFiles(int count) {

std::string root\_path = "E:\\GitHub\\Lexer\_OPT\\OPT\_Lexer\\tests";

for (int i = 1; i <= count; i++) {

std::string m\_path = root\_path + "\\test" + getNumberOfTest(i) + "\\input.sig";

start(m\_path);

}

}

std::string getNumberOfTest(int number) {

if (number < 10) {

return "0" + std::to\_string(number);

}

return std::to\_string(number);

}

void start(std::string& path) {

std::string analyzingFile = path;

std::string syntaxOutputFile = getOutputPath(analyzingFile, "output.txt");

std::string translatorOutputFile = getOutputPath(analyzingFile, "output.asm");

Lexer lexer;

Lexer::AnalyzeResult lexerAnalyzerResult = lexer.scanFile(analyzingFile);

std::cout << std::endl << std::endl << std::endl << "\t\t\t\t" << "RESULT" << std::endl;

// lexer.printScanResult();

std::vector<LexerResult> results = lexer.getResults();

SyntaxAnalyzer syntaxAnalizer(results);

syntaxAnalizer.analyze();

syntaxAnalizer.dumpTreeIntoFile(syntaxOutputFile, lexerAnalyzerResult.getErrorMassage());

std::cout << std::endl << std::endl;

// Translator translator = Translator(syntaxAnalizer.getResultTree());

// translator.analyze();

// translator.dumpIntoFile(translatorOutputFile);

}

std::string getOutputPath(std::string& inputPath, std::string output) {

std::string stringInputPath = inputPath;

int index = stringInputPath.find\_last\_of('\\', stringInputPath.size());

if (index < 0 || index >= stringInputPath.size()) return stringInputPath.append(output);

return stringInputPath.substr(0, index + 1).append(output);

}

**Lexer.h**

#pragma once

#include <vector>

#include <fstream>

#include <map>

#include "MultiSeparatedToken.h"

#include "LexerResult.h"

#include "TokenStatus.h"

class Lexer {

private:

// variables

int currentTokenState[6] = { 0, 0, 0, 0, 0, 0 };

std::string token = "";

int currentColumn = 0;

int currentRow = 1;

int savedColumn = currentColumn;

int savedRow = currentRow;

bool errorHappened = false;

std::string errorToken = "";

std::vector<int> lettersVector = {

'a', 'A', 'b', 'B', 'c', 'C', 'd', 'D', 'e', 'E', 'f', 'F', 'g', 'G', 'h', 'H',

'k', 'K', 'l', 'L', 'm', 'M', 'n', 'N', 'o', 'O', 'p', 'P', 'q', 'Q', 'r', 'R',

's', 'S', 't', 'T', 'u', 'U', 'v', 'V', 'w', 'W', 'x', 'X', 'y', 'y', 'z', 'Z',

'i', 'I'

};

std::vector<int> numbersVector = {48, 49, 50, 51, 52, 53, 54, 55, 56, 57};

std::vector<int> whiteSpacesVector = {9, 10, 13, 32};

std::vector<int> oneSeparatedTokens = {':', ';', '[',']','.'};

std::vector<MultiSeparatedToken> multiSeparatedTokens = {

MultiSeparatedToken(':', '='),

MultiSeparatedToken('.', '.')

};

std::vector<LexerResult> lexerResultValues;

std::map<std::string, int> reservedWords;

std::map<std::string, int> constantTokensMap;

std::map<std::string, int> identifiersTokensMap;

std::map<std::string, int> oneSeparatedTokensMap;

std::map<std::string, int> multiSeparatedTokensMap;

// search ASCII code of param in lettersVector and if found return TRUE else FALSE

bool isLetter(int);

// search ASCII code of param in numbersVector and if found return TRUE else FALSE

bool isNumber(int);

// return TRUE if param is '('

bool isComment(int);

// search ASCII code of param in whiteSpacesVector and if found return TRUE else FALSE

bool isWhiteSpace(int);

// search ASCII code of param in oneSeparatedTokens and if found return TRUE else FALSE

bool isOneSeparated(int);

// search ASCII code of param in multiSeparatedTokens and if found return TRUE else FALSE

bool isMultiSeparated(int);

// search ASCII code of param in multiSeparatedTokens and if found return TRUE else FALSE

bool isSecondPartOfMultiSeparated(int, int);

// set status of token to 0

void resetTokenStatus();

// check status of token and depends on value add token as: ReservedWord, Identifier, OneSeparatedToken or MultiSeparatedToken

// invoke addTokenToResult to add token to result vector

void addToken(std::string&);

// add param of func to reservedWords

void addReservedWord(const char\*, int);

// add param of func to oneSeparatedTokensMap

void addOneSeparatedToken(const char\*, int);

// add param of func to multiSeparatedTokensMap

void addMultiSeparatedToken(const char\*, int);

// add token to identifiersTokensMap, returns code of just added identifier

int addIdentifier(std::string&);

// add token to constantTokensMap, returns code of just added constant

int addConstant(std::string&);

// add token to ResultsVector

void addTokenToResultVector(std::string&, TokenStatus, int, int);

// check if passed token (param of func) is contains in: \_ and return code if contains and -1 if not

// \_ = reservedWords

int isTokenReservedWord(std::string&);

// \_ = oneSeparatedTokensMap

int isTokenOneSeparatedToken(std::string&);

// \_ = multiSeparatedTokensMap

int isTokenMultiSeparatedToken(std::string&);

// \_ = identifiersTokensMap

int isTokenIdentifier(std::string&);

// \_ = constantTokensMap

int isTokenConstant(std::string&);

// return ASCII code of next character from file

int readCharacterFromFile(std::ifstream&);

// params of func is OneSeparated token

// this func check if param can be MultiSeparatedToken and if it is invoke caseMultiSeparated

// returns ASCII code of NOT letter OneSeparated, but before invoke addToken and add scanned token

int caseOneSeparated(int, std::ifstream&);

// params of func is OneSeparated token

// this func check if param can be MultiSeparatedToken and if it is invoke readCharacterFromFile

// if readed character is pair for MultiSeparatedToken add token and return nextLetter using readCharacterFromFile

// returns ASCII code of NOT MultiSeparated letter, but before invoke addToken and add scanned token

int caseMultiSeparated(int, std::ifstream&);

// params of func is letter token

// this func check if param can be ReservedWord or Ientifier

// returns ASCII code of NOT letter letter, but before invoke addToken and add scanned token

int caseLetter(int, std::ifstream&);

// params of func is letter token

// this func check next letter and if it isNumber == true add value to token, else

// returns ASCII code of NOT number letter, but before invoke addToken and add scanned token

int caseNumber(int, std::ifstream&);

// params of func is '(' letter

// check nextLetter and if it is not '\*' print ERROR

// else scan next letter untill ['\*',')'] will be scanned one by one

// returns ASCII code of letter that don't in comment block, but before invoke addToken and add scanned token

int caseComment(int, std::ifstream&);

void handleError(const char\* text, char letter, int row, int column);

public:

Lexer();

~Lexer() = default;

struct AnalyzeResult {

AnalyzeResult(bool success, std::string message) {

this->success = success;

this->errorMessage = message;

}

inline bool isSuccess() const {

return success;

}

inline std::string getErrorMassage() const {

return errorMessage;

}

private:

std::string errorMessage;

bool success;

};

AnalyzeResult scanFile(std::string&);

void printScanResult();

std::vector<LexerResult> getResults();

};

**Lexer.cpp**

#include "Lexer.h"

#include <string>

#include <iostream>

#include "ReservedWords.h"

typedef std::pair<std::string, int> MapPair;

int TOKEN\_STATUS\_CONSTANT = 0;

int TOKEN\_STATUS\_IDENTIFIER = 1;

int TOKEN\_STATUS\_RESERVED\_WORD = 2;

int TOKEN\_STATUS\_ONE\_SEPARATED\_TOKEN = 3;

int TOKEN\_STATUS\_MULTI\_SEPARATED\_TOKEN = 4;

int TOKEN\_STATUS\_COMMENT = 5;

int oneSymbolTokenIndex = 0;

int multiSymbolTokenIndex = 301;

int reservedWordsIndex = 401;

int constantsIndex = 501;

int variablesIndex = 1001;

bool Lexer::isLetter(int character) {

for (auto& c : lettersVector) {

if (c == character) {

return true;

}

}

return false;

}

bool Lexer::isNumber(int character) {

for (auto& number : numbersVector) {

if ((char) number == character) {

return true;

}

}

return false;

}

bool Lexer::isWhiteSpace(int character) {

for (auto& whiteSpaceId : whiteSpacesVector) {

if (whiteSpaceId == character) {

return true;

}

}

return false;

}

bool Lexer::isOneSeparated(int c) {

for (auto& oneSeparatedToken : oneSeparatedTokens) {

if (oneSeparatedToken == c) {

return true;

}

}

return false;

}

bool Lexer::isMultiSeparated(int c) {

for (auto& multiSeparatedToken : multiSeparatedTokens) {

if (multiSeparatedToken.getCode() == c) {

return true;

}

}

return false;

}

bool Lexer::isSecondPartOfMultiSeparated(int sepToken, int c) {

for (auto& multiSeparatedToken : multiSeparatedTokens) {

if (multiSeparatedToken.getCode() == sepToken && multiSeparatedToken.getPairCode() == c) {

return true;

}

}

return false;

}

void Lexer::resetTokenStatus() {

currentTokenState[0] = 0;

currentTokenState[1] = 0;

currentTokenState[2] = 0;

currentTokenState[3] = 0;

currentTokenState[4] = 0;

currentTokenState[5] = 0;

}

Lexer::Lexer() {

addReservedWord("PROGRAM", PROGRAM); //401

addReservedWord("BEGIN", BEGIN); //402

addReservedWord("END", END); //403

addReservedWord("VAR", VAR); //404

addReservedWord("FLOAT", FLOAT); //405

addReservedWord("LOOP", LOOP); //406

addReservedWord("ENDLOOP", ENDLOOP); //407

addReservedWord("INTEGER", INTEGER); //408

addOneSeparatedToken(":", COLON); //0

addOneSeparatedToken(";", SEMI\_COLON); //1

addOneSeparatedToken("[", LEFT\_SQUARE\_BRACKET); //2

addOneSeparatedToken("]", RIGHT\_SQUARE\_BRACKET); //3

addOneSeparatedToken(".", DOT); //4

addMultiSeparatedToken(":=", EQUALS); //301

addMultiSeparatedToken("..", DOUBLE\_DOT); //302

}

void Lexer::addToken(std::string&) {

if (currentTokenState[TOKEN\_STATUS\_CONSTANT] == 1) {

// std::cout << token << " is constant, row=" << savedRow << " column=" << savedColumn << std::endl;

addTokenToResultVector(token, CONSTANT, savedColumn, savedRow);

return;

}

if (currentTokenState[TOKEN\_STATUS\_IDENTIFIER] == 1 && currentTokenState[TOKEN\_STATUS\_RESERVED\_WORD] == 0) {

// std::cout << token << " is identifier, row=" << savedRow << " column=" << savedColumn << std::endl;

addTokenToResultVector(token, IDENTIFIER, savedColumn, savedRow);

return;

}

if (currentTokenState[TOKEN\_STATUS\_RESERVED\_WORD] == 1 && currentTokenState[TOKEN\_STATUS\_IDENTIFIER] == 0) {

// std::cout << token << " is reserved word, row=" << savedRow << " column=" << savedColumn << std::endl;

addTokenToResultVector(token, RESERVED\_WORD, savedColumn, savedRow);

return;

}

if (currentTokenState[TOKEN\_STATUS\_ONE\_SEPARATED\_TOKEN] == 1) {

// std::cout << token << " is one separated word, row=" << savedRow << " column=" << savedColumn << std::endl;

addTokenToResultVector(token, ONE\_SEPARATED\_TOKEN, savedColumn, savedRow);

return;

}

if (currentTokenState[TOKEN\_STATUS\_MULTI\_SEPARATED\_TOKEN] == 1) {

// std::cout << token << " is multi separated word, row=" << savedRow << " column=" << savedColumn << std::endl;

addTokenToResultVector(token, MULTI\_SEPARATED\_TOKEN, savedColumn, savedRow);

return;

}

if (currentTokenState[TOKEN\_STATUS\_COMMENT] == 1) {

// std::cout << token << " is comment, row=" << savedRow << " column=" << savedColumn << std::endl;

return;

}

int reservedWordCode = isTokenReservedWord(token);

if (reservedWordCode != -1) {

// std::cout << token << " is reserved word, row=" << savedRow << "column=" << savedColumn << std::endl;

addTokenToResultVector(token, RESERVED\_WORD, savedColumn, savedRow);

return;

} else {

// std::cout << token << " is identifier, row=" << savedRow << "column=" << savedColumn << std::endl;

addTokenToResultVector(token, IDENTIFIER, savedColumn, savedRow);

return;

}

}

int Lexer::addIdentifier(std::string& token) {

identifiersTokensMap.insert(MapPair(token, variablesIndex));

int code = variablesIndex;

variablesIndex++;

return code;

}

int Lexer::addConstant(std::string& word) {

constantTokensMap.insert(MapPair(word, constantsIndex));

int code = constantsIndex;

constantsIndex++;

return code;

}

void Lexer::addTokenToResultVector(std::string& token, TokenStatus status, int column, int row) {

int code = -1;

switch (status) {

case IDENTIFIER:

code = isTokenIdentifier(token);

if (code == -1) {

code = addIdentifier(token);

}

lexerResultValues.push\_back(LexerResult(token, code, row, column, lexerResultValues.size()));

break;

case RESERVED\_WORD:

code = isTokenReservedWord(token);

lexerResultValues.push\_back(LexerResult(token, code, row, column, lexerResultValues.size()));

break;

case ONE\_SEPARATED\_TOKEN:

code = isTokenOneSeparatedToken(token);

lexerResultValues.push\_back(LexerResult(token, code, row, column, lexerResultValues.size()));

break;

case MULTI\_SEPARATED\_TOKEN:

code = isTokenMultiSeparatedToken(token);

lexerResultValues.push\_back(LexerResult(token, code, row, column, lexerResultValues.size()));

break;

case CONSTANT:

code = isTokenConstant(token);

if (code == -1) {

code = addConstant(token);

}

lexerResultValues.push\_back(LexerResult(token, code, row, column, lexerResultValues.size()));

break;

}

}

void Lexer::addReservedWord(const char\* word, int code) {

reservedWords.insert(MapPair(word, code));

}

void Lexer::addOneSeparatedToken(const char\* word, int code) {

oneSeparatedTokensMap.insert(MapPair(word, code));

}

void Lexer::addMultiSeparatedToken(const char\* word, int code) {

multiSeparatedTokensMap.insert(MapPair(word, code));

}

int Lexer::isTokenReservedWord(std::string& word){

for (auto& resWord : reservedWords) {

if (resWord.first == word) {

return resWord.second;

}

}

return -1;

}

int Lexer::isTokenOneSeparatedToken(std::string& word) {

for (auto& oneSeparetadToken : oneSeparatedTokensMap) {

if (oneSeparetadToken.first == word) {

return oneSeparetadToken.second;

}

}

return -1;

}

int Lexer::isTokenMultiSeparatedToken(std::string& word) {

for (auto& multiSeparatedToken : multiSeparatedTokensMap) {

if (multiSeparatedToken.first == word) {

return multiSeparatedToken.second;

}

}

return -1;

}

int Lexer::isTokenIdentifier(std::string& word) {

for (auto& indentifier : identifiersTokensMap) {

if (indentifier.first == word) {

return indentifier.second;

}

}

return -1;

}

int Lexer::isTokenConstant(std::string& word) {

for (auto& constant : constantTokensMap) {

if (constant.first == word) {

return constant.second;

}

}

return -1;

}

int Lexer::readCharacterFromFile(std::ifstream &file) {

char currentCaracter = file.eof();

if (!file.eof()) {

file.get(currentCaracter);

currentColumn++;

}

return (int)currentCaracter;

}

int Lexer::caseMultiSeparated(int letter, std::ifstream& file) {

token += (char)letter;

while (!file.eof()) {

int nextLetter = readCharacterFromFile(file);

if (isSecondPartOfMultiSeparated(letter, nextLetter)) {

currentTokenState[TOKEN\_STATUS\_MULTI\_SEPARATED\_TOKEN] = 1;

currentTokenState[TOKEN\_STATUS\_ONE\_SEPARATED\_TOKEN] = 0;

token += (char)nextLetter;

} else {

if (!token.empty()) {

addToken(token);

}

token = "";

resetTokenStatus();

return nextLetter;

}

}

}

int Lexer::caseOneSeparated(int letter, std::ifstream& file) {

resetTokenStatus();

currentTokenState[TOKEN\_STATUS\_ONE\_SEPARATED\_TOKEN] = 1;

while (!file.eof()) {

if (isMultiSeparated(letter)) {

return caseMultiSeparated(letter, file);

} else {

token = (char)letter;

addToken(token);

token = "";

resetTokenStatus();

return readCharacterFromFile(file);

}

}

}

int Lexer::caseLetter(int letter, std::ifstream& file) {

token += (char)letter;

currentTokenState[TOKEN\_STATUS\_IDENTIFIER] = 1;

currentTokenState[TOKEN\_STATUS\_RESERVED\_WORD] = 1;

while (!file.eof()) {

int nextLetter = readCharacterFromFile(file);

if (isNumber(nextLetter)) {

currentTokenState[TOKEN\_STATUS\_IDENTIFIER] = 1;

token += (char)nextLetter;

continue;

}

if (!isLetter(nextLetter)) {

if (!token.empty()) {

addToken(token);

}

token = "";

resetTokenStatus();

return nextLetter;

}

token += (char)nextLetter;

}

}

int Lexer::caseNumber(int letter, std::ifstream& file) {

currentTokenState[TOKEN\_STATUS\_CONSTANT] = 1;

token += (char)letter;

while (!file.eof()) {

int nextLetter = readCharacterFromFile(file);

if (!isNumber(nextLetter)) {

if (!isOneSeparated(nextLetter) && !isWhiteSpace(nextLetter)) {

handleError("Expected number or whitespace or separator, but got ", nextLetter, currentRow, currentColumn);

return nextLetter;

}

if (!token.empty()) {

addToken(token);

}

token = "";

resetTokenStatus();

return nextLetter;

}

token += (char)nextLetter;

}

}

int Lexer::caseComment(int letter, std::ifstream& file) {

resetTokenStatus();

currentTokenState[TOKEN\_STATUS\_COMMENT] = 1;

token += (char)letter;

bool isStartFound = false;

int nextLetter = readCharacterFromFile(file);

if (nextLetter == '\0') {

handleError("Unclosed comment", nextLetter, currentRow, currentColumn);

return nextLetter;

}

if (nextLetter != '\*') {

handleError("Comment must start with '(\*'", nextLetter, currentRow, currentColumn);

return nextLetter;

} else {

token += (char)nextLetter;

}

while (!file.eof()) {

nextLetter = readCharacterFromFile(file);

if (nextLetter == '\0') {

handleError("Unclosed comment", nextLetter, currentRow, currentColumn);

return nextLetter;

}

if (nextLetter == '\*') {

isStartFound = true;

} else if (nextLetter == ')' && isStartFound) {

token += (char)nextLetter;

addToken(token);

token = "";

resetTokenStatus();

return readCharacterFromFile(file);

} else {

isStartFound = false;

}

if (nextLetter == '\n') { currentRow++; currentColumn = 0; }

token += (char)nextLetter;

}

}

void Lexer::handleError(const char\* text, char letter, int row, int column) {

errorHappened = true;

std::cout << text << " character " << letter << " code: " << (int)letter << " (row: " << row << " column: " << column << ")" << std::endl;

errorToken = text;

errorToken.append(" character: ");

errorToken.append(std::to\_string(letter));

errorToken.append(" code: ");

errorToken.append(std::to\_string((int)letter));

errorToken.append(" (row: ");

errorToken.append(std::to\_string(row));

errorToken.append(" column: ");

errorToken.append(std::to\_string(column));

errorToken.append(")");

}

bool Lexer::isComment(int letter) {

return letter == '(';

}

Lexer::AnalyzeResult Lexer::scanFile(std::string& filePath) {

std::ifstream file;

file.open(filePath, std::ios::in);

int savedLetter = -1;

while (!file.eof()) {

int letter = savedLetter == -1 ? readCharacterFromFile(file) : savedLetter;

if (isLetter(letter)) {

savedLetter = -1;

savedColumn = currentColumn;

savedRow = currentRow;

letter = caseLetter(letter, file);

if (errorHappened) return AnalyzeResult(errorHappened, errorToken);

}

if (isOneSeparated(letter)) {

savedLetter = -1;

savedColumn = currentColumn;

savedRow = currentRow;

letter = caseOneSeparated(letter, file);

if (errorHappened) return AnalyzeResult(errorHappened, errorToken);

}

if (isNumber(letter)) {

savedLetter = -1;

savedColumn = currentColumn;

savedRow = currentRow;

letter = caseNumber(letter, file);

if (errorHappened) return AnalyzeResult(errorHappened, errorToken);

}

if (isComment(letter)) {

savedLetter = -1;

savedColumn = currentColumn;

savedRow = currentRow;

letter = caseComment(letter, file);

if (errorHappened) return AnalyzeResult(errorHappened, errorToken);

}

if (isWhiteSpace(letter)) {

savedColumn = currentColumn;

savedRow = currentRow;

savedLetter = -1;

if (!token.empty()) {

addToken(token);

token = "";

}

resetTokenStatus();

if (letter == '\n') {

currentColumn = 0;

currentRow++;

}

continue;

}

// to prevent endless loop when met undefined letter

if (letter == savedLetter) {

handleError("Undefined symbol", letter, currentRow, currentColumn);

return AnalyzeResult(errorHappened, errorToken);

}

savedLetter = letter;

}

return AnalyzeResult(errorHappened, errorToken);

}

void Lexer::printScanResult() {

for (auto& result : lexerResultValues) {

result.print();

}

}

std::vector<LexerResult> Lexer::getResults(){

return lexerResultValues;

}

**SyntaxAnalyzer.h**

#pragma once

#include "LexerResult.h"

#include <vector>

#include "Tree.h"

class SyntaxAnalyzer

{

public:

SyntaxAnalyzer(std::vector<LexerResult>&);

void analyze();

void dumpTreeIntoFile(std::string&, std::string);

Tree\* getResultTree();

private:

int currentIndex = 0;

std::vector<LexerResult> lexerResults;

Tree tree;

// for error handling

bool errorSyntaxHappened = false;

std::string errorMessageSyntax;

/\* reads from lexerResults symbol with index @param

if IndexOutOfBoundsException -> print exception "Unexpected end of file"

and end analyzing

\*/

LexerResult getItem(int);

//

void handleError(const char\*, LexerResult);

/\* start analyzing with 2-nd rule

return true if exception not hapenned and false if happened

during analyzing all program

\*/

bool caseProgram(int);

/\* each of this functions takes int

this is index of item in lexerResults to analyze

and each of this function return item next from lexerResults

that don't match rule of function

or LexerResult nullableResult("", -1, -1, -1, -1); \*/

LexerResult caseBlock(int);

LexerResult caseRange(int);

LexerResult caseVariable(int);

LexerResult caseVariableIdentifier(int);

LexerResult caseAttribute(int, Tree::TreeItem\*);

LexerResult caseStatement(int);

LexerResult caseDimension(int);

LexerResult caseIdentifier(int);

LexerResult caseExpression(int);

LexerResult caseDeclaration(int);

LexerResult caseAttributeList(int);

LexerResult caseDeclarationList(int, Tree::TreeItem \*);

LexerResult caseVariableDeclarations(int);

/\* this function takes 2 parametrs,

first one is default as for other functions

second is code of item

fucntion check code of item that have index of first param and if it is equal second param

return this item in other case continue it logic \*/

LexerResult caseStatementList(int, int);

};

**SyntaxAnalyzer.cpp**

#include "SyntaxAnalyzer.h"

#include <fstream>

#include "Rules.h"

#include "ReservedWords.h"

LexerResult nullableResult("", -1, -1, -1, -1);

SyntaxAnalyzer::SyntaxAnalyzer(std::vector<LexerResult>& res) {

this->lexerResults = res;

this->errorMessageSyntax = "";

this->errorSyntaxHappened = false;

}

void SyntaxAnalyzer::analyze(){

tree.addChild("<signal-program> --> <program>", -1, -1, SIGNAL\_PROGRAM, -1);

if (caseProgram(0)) {

// tree.print();

}

tree.print();

}

void SyntaxAnalyzer::dumpTreeIntoFile(std::string& filename, std::string startText) {

std::ofstream file;

file.open(filename);

file << startText << std::endl << std::endl;

if (errorSyntaxHappened) {

file << errorMessageSyntax << std::endl << std::endl;

}

tree.dumpIntoFile(file);

file.close();

}

Tree\* SyntaxAnalyzer::getResultTree() {

return &tree;

}

LexerResult SyntaxAnalyzer::getItem(int index){

if (lexerResults.size() <= index) {

errorSyntaxHappened = true;

std::cout << "Unexpected end of file" << std::endl;

errorMessageSyntax = "Unexpected end of file";

return nullableResult;

}

return lexerResults.at(index);

}

void SyntaxAnalyzer::handleError(const char\* message, LexerResult failedItem) {

errorSyntaxHappened = true;

std::cout << message << ", row = " << failedItem.getRowNumber() << " column = " << failedItem.getColumnNumber() << ", but got " << failedItem.getToken() << std::endl;

errorMessageSyntax = message;

errorMessageSyntax.append(", row = ");

errorMessageSyntax.append(std::to\_string(failedItem.getRowNumber()));

errorMessageSyntax.append(" column = ");

errorMessageSyntax.append(std::to\_string(failedItem.getColumnNumber()));

errorMessageSyntax.append(" but got ");

errorMessageSyntax.append(failedItem.getToken());

}

// 2. < program > --> PROGRAM <procedure - identifier>;

bool SyntaxAnalyzer::caseProgram(int index) {

Tree::TreeItem\* it = tree.addNext("<program> --> PROGRAM <procedure-identifier>;< block > .", -1, -1, PROGRAM\_RULE, -1);

if (getItem(index).getCode() != PROGRAM) {

handleError("PROGRAM expected", getItem(index));

return false;

}

tree.addNext("<procedure-identifier> --> <identifier>", -1, -1, PROCEDURE\_IDENTIFIER, -1);

LexerResult nextItem = caseIdentifier(index + 1);

tree.switchTo(it);

if (errorSyntaxHappened) return false;

if (nextItem.getCode() != SEMI\_COLON) {

handleError("; expected", nextItem);

return false;

}

tree.addNext("<block> --> <variable-declarations> BEGIN <statements-list> END", -1, -1, BLOCK, -1);

nextItem = caseBlock(nextItem.getIndexInResultVector() + 1);

return !errorSyntaxHappened;

}

// 3. < block > --> <variable - declarations> BEGIN <statements - list> END

LexerResult SyntaxAnalyzer::caseBlock(int index) {

Tree::TreeItem\* current = tree.getCurrent();

LexerResult item = caseVariableDeclarations(index);

tree.switchTo(current);

if (errorSyntaxHappened) return nullableResult;

if (item.getCode() != BEGIN) {

handleError("BEGIN expected", item);

return nullableResult;

}

tree.addChild(item.getToken(), item.getRowNumber(), item.getColumnNumber(), ADDING\_RESERVED\_WORD, BEGIN);

item = caseStatementList(item.getIndexInResultVector() + 1, END);

tree.switchTo(current);

if (errorSyntaxHappened) return nullableResult;

if (item.getCode() != END) {

handleError("END expected", item);

return nullableResult;

}

tree.addChild(item.getToken(), item.getRowNumber(), item.getColumnNumber(), ADDING\_RESERVED\_WORD, item.getCode());

return item;

}

// 4. <variable-declarations> --> VAR <declarations-list>|<empty>

LexerResult SyntaxAnalyzer::caseVariableDeclarations(int index) {

Tree::TreeItem\* it = tree.getCurrent();

tree.addNext("<variable-declarations> --> VAR <declarations-list>|<empty>", -1, -1, VARIABLE\_DECLARATIONS, -1);

LexerResult item = getItem(index);

if (item.getCode() != VAR) {

handleError("VAR expected", getItem(index));

return nullableResult;

}

tree.addChild(item.getToken(), item.getRowNumber(), item.getColumnNumber(), ADDING\_RESERVED\_WORD, item.getCode());

return caseDeclarationList(item.getIndexInResultVector() + 1, it);

}

// 5. <declarations-list> --> <declaration><declarations-list>|<empty>

LexerResult SyntaxAnalyzer::caseDeclarationList(int index, Tree::TreeItem\* root) {

tree.addNext("<declarations-list> --> <declaration><declarations-list>|<empty>", -1, -1, DECLARATIONS\_LIST, -1);

Tree::TreeItem\* it = tree.getCurrent();

LexerResult item = getItem(index);

// if begin it means that declaration-list is empty or ended

if (item.getCode() == BEGIN) {

tree.addNext("<empty>", -1, -1, EMPTY, -1);

tree.switchTo(root);

return item;

}

item = caseDeclaration(index);

if (errorSyntaxHappened) return nullableResult;

tree.switchTo(it);

return caseDeclarationList(item.getIndexInResultVector(), root);

}

// 6. <declaration> --> <variable-identifier>:<attribute><attributes-list>;

LexerResult SyntaxAnalyzer::caseDeclaration(int index) {

tree.addNext("<declaration> --><variable-identifier>:<attribute><attributes-list>;", -1, -1, DECLARATION, -1);

Tree::TreeItem\* it = tree.getCurrent();

tree.addNext("<variable-identifier> --> <identifier>", -1, -1, VARIABLE\_IDENTIFIER, -1);

LexerResult nextItem = caseIdentifier(index);

if (errorSyntaxHappened) return nullableResult;

// nextItem has index + 1

if (nextItem.getCode() != COLON) { // ':' code 0

handleError("':' expected", nextItem);

}

tree.switchTo(it);

tree.addChild(nextItem.getToken(), nextItem.getRowNumber(), nextItem.getColumnNumber(), ADDING\_SEPARATED, COLON);

if (errorSyntaxHappened) return nullableResult;

// attribute for index + 2

nextItem = caseAttribute(nextItem.getIndexInResultVector() + 1, it);

if (errorSyntaxHappened) return nullableResult;

nextItem = caseAttributeList(nextItem.getIndexInResultVector());

if (errorSyntaxHappened) return nullableResult;

tree.switchTo(it);

return nextItem;

}

// 7. <attributes-list> --> <attribute> <attributes-list> | <empty>

LexerResult SyntaxAnalyzer::caseAttributeList(int index) {

tree.addNext("<attributes-list> --> <attribute> <attributes-list> | <empty>", -1, -1, ATTRIBUTES\_LIST, -1);

Tree::TreeItem\* it = tree.getCurrent();

LexerResult item = getItem(index);

if (item.getCode() == SEMI\_COLON) { // ';' code 1

// attributr-list is empty or ended

tree.addChild("<empty>", -1, -1, EMPTY, -1);

return getItem(index + 1);

}

item = caseAttribute(index, it);

tree.switchTo(it);

return errorSyntaxHappened ? nullableResult : caseAttributeList(item.getIndexInResultVector());

}

// 8. < attribute > --> INTEGER | FLOAT | [<range>]

LexerResult SyntaxAnalyzer::caseAttribute(int index, Tree::TreeItem\* root) {

tree.addNext("< attribute > -- > INTEGER | FLOAT | [<range>]", -1, -1, ATTRIBUTE, -1);

LexerResult item = getItem(index);

if (item.getCode() == INTEGER || item.getCode() == FLOAT) {

tree.addChild(item.getToken(), item.getRowNumber(), item.getColumnNumber(), ADDING\_RESERVED\_WORD, item.getCode());

tree.switchTo(root);

return getItem(index + 1);

}

if (item.getCode() == LEFT\_SQUARE\_BRACKET) { // '[' code 2

tree.addChild(item.getToken(), item.getRowNumber(), item.getColumnNumber(), RANGE\_SEPARATED\_SYMBOL, item.getCode());

Tree::TreeItem\* it = tree.getCurrent();

item = caseRange(index + 1);

if (errorSyntaxHappened) return nullableResult;

if (item.getCode() != RIGHT\_SQUARE\_BRACKET) { // ']' code 3

handleError("expected ]", item);

return nullableResult;

}

tree.switchTo(it);

tree.addChild(item.getToken(), item.getRowNumber(), item.getColumnNumber(), RANGE\_SEPARATED\_SYMBOL, item.getCode());

tree.switchTo(root);

return getItem(item.getIndexInResultVector() + 1);

}

handleError("expected INTEGER | FLOAT | [<range>]", item);

return nullableResult;

}

// 9. <range> --> <unsigned-integer> .. <unsigned-integer>

LexerResult SyntaxAnalyzer::caseRange(int index) {

tree.addNext("<range> --> <unsigned-integer> .. <unsigned-integer>", -1, -1, RANGE, -1);

Tree::TreeItem\* it = tree.getCurrent();

LexerResult item = getItem(index);

if (item.getCode() < 501 || item.getCode() >= 1001) {

handleError("<unsigned - integer> expected", item);

return nullableResult;

}

tree.addNext("<unsigned-integer>", -1, -1, UNSIGNED\_INTEGER, -1);

int code = item.getCode();

tree.addChild(item.getToken(), item.getRowNumber(), item.getColumnNumber(), ADDING\_CONSTANT, code);

tree.switchTo(it);

item = getItem(index + 1);

if (item.getCode() != DOUBLE\_DOT) { // '..' code 302

handleError(".. expected", item);

return nullableResult;

}

tree.addChild(item.getToken(), item.getRowNumber(), item.getColumnNumber(), RANGE\_SEPARATED\_SYMBOL, DOUBLE\_DOT);

item = getItem(index + 2);

if (item.getCode() < 501 || item.getCode() >= 1001) {

handleError("<unsigned - integer> expected", item);

return nullableResult;

}

tree.addNext("<unsigned-integer>",-1, -1, UNSIGNED\_INTEGER, -1);

tree.addChild(item.getToken(), item.getRowNumber(), item.getColumnNumber(), ADDING\_CONSTANT, item.getCode());

tree.switchTo(it);

return getItem(index + 3);

}

// 10. <statements-list> --> <statement> <statements-list> | <empty>

LexerResult SyntaxAnalyzer::caseStatementList(int index, int code) {

tree.addNext("<statements-list> --> <statement> <statements-list> | <empty>", -1, -1, STATEMENT\_LIST, -1);

Tree::TreeItem\* it = tree.getCurrent();

LexerResult item = getItem(index);

// if met END or ENDLOOP means

// end

if (item.getCode() == code) {

tree.addNext("<empty>", -1, -1, EMPTY, -1);

return item;

}

item = caseStatement(item.getIndexInResultVector());

tree.switchTo(it);

if (errorSyntaxHappened) return nullableResult;

return caseStatementList(item.getIndexInResultVector(), code);

}

// 11. <statement> --> <variable> := <expression> ; | LOOP <statements-list> ENDLOOP ;

LexerResult SyntaxAnalyzer::caseStatement(int index) {

tree.addNext("<statement> --> <variable> := <expression> ; | LOOP <statements-list> ENDLOOP;", -1, -1, STATEMENT, -1);

LexerResult item = getItem(index);

Tree::TreeItem\* current = tree.getCurrent();

if (item.getCode() >= 1001) {

// variable

item = caseVariable(item.getIndexInResultVector());

if (item.getCode() != EQUALS) {

handleError(":= expected", item);

return nullableResult;

}

tree.switchTo(current);

tree.addChild(item.getToken(), item.getRowNumber(), item.getColumnNumber(), ADDING\_RESERVED\_WORD, item.getCode());

item = caseExpression(item.getIndexInResultVector() + 1);

if (errorSyntaxHappened) return nullableResult;

// LOOP

} else if (item.getCode() == LOOP) { // LOOP - 406

tree.addChild(item.getToken(), item.getRowNumber(), item.getColumnNumber(), ADDING\_RESERVED\_WORD, item.getCode());

item = caseStatementList(item.getIndexInResultVector() + 1, ENDLOOP);

tree.switchTo(current);

if (errorSyntaxHappened) return nullableResult;

if (item.getCode() != ENDLOOP) { //ENDLOOP - 407

handleError("ENDLOOP expected", item);

return nullableResult;

}

tree.addChild(item.getToken(), item.getRowNumber(), item.getColumnNumber(), ADDING\_RESERVED\_WORD, item.getCode());

item = getItem(item.getIndexInResultVector() + 1);

} else {

handleError("<variable> or LOOP expected", item);

return nullableResult;

}

tree.switchTo(current);

if (item.getCode() == SEMI\_COLON) { // ';' code 1

// attributr-list is empty or ended

tree.addChild(item.getToken(), item.getRowNumber(), item.getColumnNumber(), ADDING\_SEPARATED, item.getCode());

return getItem(item.getIndexInResultVector() + 1);

}

handleError("; expected", item);

return nullableResult;

}

// 12. <expression> --> <variable> | <unsigned-integer>

LexerResult SyntaxAnalyzer::caseExpression(int index) {

tree.addNext("<expression> --> <variable> | <unsigned-integer>", -1, -1, EXPRESSION, -1);

LexerResult item = getItem(index);

if (item.getCode() >= 501 && item.getCode() < 1001) {

tree.addNext("<unsigned-integer>", -1, -1, UNSIGNED\_INTEGER, -1);

tree.addChild(item.getToken(), item.getRowNumber(), item.getColumnNumber(), ADDING\_CONSTANT, item.getCode());

return getItem(index + 1);

}

item = caseVariable(item.getIndexInResultVector());

if (errorSyntaxHappened) return nullableResult;

return item;

}

// 13. <variable> --> <variable-identifier><dimension>

LexerResult SyntaxAnalyzer::caseVariable(int index) {

Tree::TreeItem\* cur = tree.getCurrent();

Tree::TreeItem\* it = tree.addNext("<variable> --> <variable-identifier><dimension>", -1, -1, VARIABLE, -1);

LexerResult item = caseVariableIdentifier(index);

tree.switchTo(it);

if (errorSyntaxHappened) return nullableResult;

if (item.getCode() == LEFT\_SQUARE\_BRACKET) {

item = caseDimension(item.getIndexInResultVector());

if (errorSyntaxHappened) return nullableResult;

return item;

} else {

tree.addNext("<dimension> --> [ <expression> ] | <empty>", -1, -1, DIMENSION, -1);

tree.addChild("<empty>", -1, -1, EMPTY, -1);

}

tree.switchTo(cur);

return item;

}

LexerResult SyntaxAnalyzer::caseVariableIdentifier(int index) {

tree.addNext("<variable-identifier> --> <identifier>", -1, -1, VARIABLE\_IDENTIFIER, -1);

return caseIdentifier(index);

}

// 14. <dimension> --> [ <expression> ] | <empty>

LexerResult SyntaxAnalyzer::caseDimension(int index) {

tree.addNext("<dimension> --> [ <expression> ] | <empty>", -1, -1, DIMENSION, -1);

Tree::TreeItem\* it = tree.getCurrent();

LexerResult item = getItem(index);

if (item.getCode() != LEFT\_SQUARE\_BRACKET) {

handleError("[ expected", item);

return nullableResult;

}

tree.addChild(item.getToken(), -1, -1, RANGE\_SEPARATED\_SYMBOL, LEFT\_SQUARE\_BRACKET);

item = caseExpression(item.getIndexInResultVector() + 1);

tree.switchTo(it);

if (item.getCode() != RIGHT\_SQUARE\_BRACKET) {

handleError("] expected", item);

return nullableResult;

}

tree.addChild(item.getToken(), item.getRowNumber(), item.getColumnNumber(), RANGE\_SEPARATED\_SYMBOL, RIGHT\_SQUARE\_BRACKET);

return getItem(item.getIndexInResultVector() + 1);

}

// 16,16,17. <identifier> --> <letter><string>

LexerResult SyntaxAnalyzer::caseIdentifier(int index) {

tree.addNext("<identifier> --> <letter><string>", -1, -1, IDENTIFIER\_RULE, -1);

LexerResult item = getItem(index);

tree.addChild(item.getToken(), item.getRowNumber(), item.getColumnNumber(), ADDING\_INDENTIFIER, item.getCode());

if (item.getCode() >= 1001) {

return getItem(index + 1);

}

handleError("<identifier> expected", item);

return nullableResult;

}

**Tree.h**

#pragma once

#include <vector>

#include <iostream>

#include <string>

#include <fstream>

#include "Rules.h"

class Tree {

public:

Tree();

~Tree() = default;

struct TreeItem {

public:

TreeItem(std::string& data, int row, int column, Rules rule, int value) {

this->data = data;

this->value = value;

this->rule = rule;

this->row = row;

this->column = column;

}

~TreeItem() = default;

inline std::vector<TreeItem\*> getChilds() {

return childs;

}

inline Rules getRule() const {

return rule;

}

inline std::string getStringData() const {

return data;

}

inline int getData() const {

return value;

}

inline int getRow() const {

return row;

}

inline int getColumn() const {

return column;

}

TreeItem\* add(TreeItem\* item) {

childs.push\_back(item);

return item;

}

void print(int tabulation) {

std::cout << "(" << tabulation << ")";

int tab = tabulation;

while (tab > 0) {

std::cout << "\*";

tab--;

}

std::cout << data;

if (value != -1) {

std::cout << " " << value;

}

std::cout << std::endl;

for (auto& child : childs) {

child->print(tabulation + 2);

}

}

void dumpIntoFile(std::ofstream& fileStream, int tabulation) {

fileStream << "(" << tabulation << ")";

int tab = tabulation;

while (tab > 0) {

fileStream << "\*";

tab--;

}

fileStream << data;

if (value != -1) {

fileStream << " " << value;

}

fileStream << std::endl;

for (auto& child : childs) {

child->dumpIntoFile(fileStream, tabulation + 2);

}

}

private:

// contains child of current rule

std::vector<TreeItem\*> childs;

// contains rule as string (for output)

std::string data;

// contains id of rule

Rules rule;

// contains id of main resource (such as id of variable or identifier)

int value;

// row position

int row;

// column position

int column;

};

/\* add child to lastModified

DON'T modify the lastModified

return pointer to added item

\*/

TreeItem\* addChild(std::string, int, int, Rules, int);

TreeItem\* addChild(const char\*, int, int, Rules, int);

TreeItem\* addChild(int, int, int, Rules, int);

/\* add next to lastModified

lastModified become added TreeItem

return pointer to added item

\*/

TreeItem\* addNext(std::string, int, int, Rules, int);

TreeItem\* addNext(const char\*, int, int, Rules, int);

TreeItem\* addNext(int, int, int, Rules, int);

TreeItem\* getCurrent();

/\* lastModifiend = param

TRUE -> if switch is successfull

FALSE -> if not

\*/

bool switchTo(TreeItem\*);

void print();

void dumpIntoFile(std::ofstream&);

TreeItem\* getRoot();

private:

TreeItem\* root;

TreeItem\* lastModified;

int depth;

int elements;

TreeItem\* add(std::string&, int, int, Rules, int, bool);

};

**Tree.cpp**

#include "Tree.h"

Tree::Tree() {

this->root = nullptr;

this->lastModified = nullptr;

this->elements = 0;

this->depth = 0;

}

Tree::TreeItem\* Tree::addChild(std::string data, int row, int column, Rules rules, int value) {

return add(data, row, column, rules, value, false);

}

Tree::TreeItem\* Tree::addChild(const char\* data, int row, int column, Rules rules, int value) {

std::string str = data;

return add(str, row, column, rules, value, false);

}

Tree::TreeItem\* Tree::addChild(int data, int row, int column, Rules rules, int value) {

std::string str = std::to\_string(data);

return add(str, row, column, rules, value, false);

}

Tree::TreeItem\* Tree::addNext(std::string data, int row, int column, Rules rules, int value) {

depth++;

return add(data, row, column, rules, value, true);

}

Tree::TreeItem\* Tree::addNext(const char\* data, int row, int column, Rules rules, int value) {

std::string str = data;

return add(str, row, column, rules, value, true);

}

Tree::TreeItem\* Tree::addNext(int data, int row, int column, Rules rules, int value) {

std::string str = std::to\_string(data);

return add(str, row, column, rules, value, true);

}

Tree::TreeItem\* Tree::add(std::string& data, int row, int column, Rules rules, int value, bool isModify) {

elements++;

TreeItem\* item = new TreeItem(data, row, column, rules, value);

if (root == nullptr) {

root = item;

lastModified = root;

return lastModified;

}

item = lastModified->add(item);

lastModified = isModify ? item : lastModified;

return item;

}

bool Tree::switchTo(TreeItem\* item) {

if (item == nullptr) {

return false;

}

lastModified = item;

return true;

}

Tree::TreeItem\* Tree::getCurrent() {

return lastModified;

}

void Tree::dumpIntoFile(std::ofstream& outputStream) {

if (root == nullptr) return;

root->dumpIntoFile(outputStream, 0);

}

Tree::TreeItem\* Tree::getRoot(){

return root;

}

void Tree::print() {

if (root == nullptr) return;

root->print(0);

}

**TokenStatus.h**

#pragma once

enum TokenStatus {

IDENTIFIER,

RESERVED\_WORD,

ONE\_SEPARATED\_TOKEN,

MULTI\_SEPARATED\_TOKEN,

CONSTANT

};

**Rules.h**

#pragma once

enum Rules{

SIGNAL\_PROGRAM = 1,

PROGRAM\_RULE = 2,

BLOCK = 3,

VARIABLE\_DECLARATIONS = 4,

DECLARATIONS\_LIST = 5,

DECLARATION = 6,

ATTRIBUTES\_LIST = 7,

ATTRIBUTE = 8,

RANGE = 9,

STATEMENT\_LIST = 10,

STATEMENT = 11,

EXPRESSION = 12,

VARIABLE = 13,

DIMENSION = 14,

VARIABLE\_IDENTIFIER = 15,

PROCEDURE\_IDENTIFIER = 16,

IDENTIFIER\_RULE = 17,

STRING = 18,

UNSIGNED\_INTEGER = 19,

EMPTY = 20,

RANGE\_SEPARATED\_SYMBOL = 21,

ADDING\_RESERVED\_WORD = 22,

ADDING\_INDENTIFIER = 23,

ADDING\_SEPARATED = 24,

ADDING\_CONSTANT = 25

};

/\* RULES

1. <signal-program> --> <program>

2. <program> --> PROGRAM <procedure-identifier>;

<block>.

3. <block> --> <variable-declarations> BEGIN <statements-list> END

4. <variable-declarations> --> VAR <declarations-list>|<empty>

5. <declarations-list> --> <declaration><declarations-list>|<empty>

6. <declaration> --><variable-identifier>:<attribute><attributes-list> ;

7. <attributes-list> --> <attribute> <attributes-list> | <empty>

8. <attribute> --> INTEGER | FLOAT | [<range>]

9. <range> --> <unsigned-integer> .. <unsigned-integer>

10. <statements-list> --> <statement> <statements-list> | <empty>

11. <statement> --> <variable> := <expression> ; | LOOP <statements-list> ENDLOOP ;

12. <expression> --> <variable> | <unsigned-integer>

13. <variable> --> <variable-identifier><dimension>

14. <dimension> --> [ <expression> ] | <empty>

15. <variable-identifier> --> <identifier>

16. <procedure-identifier> --> <identifier>

----------------------------------------------------------------------------------------------

17. <identifier> --> <letter><string>

18. <string> --> <letter><string> | <digit><string> | <empty>

19. <unsigned-integer> --> <digit><digits-string>

20. <digits-string> --> <digit><digits-string> | <empty>

21. <digit> --> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

22. <letter> --> A | B | C | D | ... | Z

\*/

**ReservedWords.h**

#pragma once

enum ReserverWords {

COLON = 0,

SEMI\_COLON = 1,

LEFT\_SQUARE\_BRACKET = 2,

RIGHT\_SQUARE\_BRACKET = 3,

DOT = 4,

PROGRAM = 401,

BEGIN = 402,

END = 403,

EQUALS = 301,

DOUBLE\_DOT = 302,

VAR = 404,

FLOAT = 405,

LOOP = 406,

ENDLOOP = 407,

INTEGER = 408

};

**MultiSeparatedToken.h**

#pragma once

class MultiSeparatedToken

{

public:

MultiSeparatedToken(int \_code, int \_pairCode) {

code = \_code;

pairCode = \_pairCode;

}

~MultiSeparatedToken() = default;

inline int getCode() { return code; }

inline int getPairCode() { return pairCode; }

private:

int code;

int pairCode;

};

**LexerResult.h**

#pragma once

#include <string>

#include <iostream>

class LexerResult

{

public:

LexerResult(std::string token, int code, int rowNumber, int columnNumber, int indexInResultVector) {

this->token = token;

this->code = code;

this->columnNumber = columnNumber;

this->rowNumber = rowNumber;

this->indexInResultVector = indexInResultVector;

}

~LexerResult() = default;

inline std::string getToken() const { return token; }

inline int getCode() const { return code; }

inline int getRowNumber() const { return rowNumber; }

inline int getColumnNumber() const { return columnNumber; }

inline int getIndexInResultVector() const { return indexInResultVector; }

void print() {

// std::cout << token << ", code=" << code << ", column=" << columnNumber << ", row=" << rowNumber << std::endl;

std::cout << rowNumber << " " << columnNumber << " " << code << " " << token << "position in resultVector: " << indexInResultVector << std::endl;

}

private:

std::string token;

int code;

int rowNumber;

int columnNumber;

int indexInResultVector;

};

**Тести**