Міністерство освіти і науки України
Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Факультет прикладної математики
Кафедра системного програмування
і спеціалізованих комп'ютерних системи

Лабораторна робота №2

з дисципліни
«Основи проектування трансляторів»
На тему: "Розробка генератору коду"

Виконав:
студент групи КВ-84
Воєводін Ілля Петрович

Перевірив:

Київ – 2021

Завдання за варіантом 9

```
1. <signal-program> --> <program>
2. <program> --> PROGRAM <procedure-identifier> ;
        <block>.
3. <block> --> <declarations> BEGIN <statements-
        list> END
4. <statements-list> --> <empty>
5. <declarations> --> <math-function-declarations>
6. <math-function-declarations> --> DEFFUNC
        <function-list> |
        <empty>
7. <function-list> --> <function> <function-list>
        |
        <empty>
8. <function> --> <function-identifier> =
        <constant><function-characteristic> ;
9. <function-characteristic> --> \ <unsigned-
        integer> , <unsigned-integer>
10. <constant> --> <unsigned-integer>
11. <procedure-identifier> --> <identifier>
12. <function-identifier> --> <identifier>
13. <identifier> --> <letter><string>
14. <string> --> <letter><string> |
        <digit><string> |
        <empty>
15. <unsigned-integer> --> <digit><digits-string>
16. <digits-string> --> <digit><digits-string> |
        <empty>
17. <digit> --> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
18. <letter> --> A | B | C | D | ... | Z
```

Метод реалізації синтаксичного аналізатора: низхідний рекурсивний спуск.

Лістинг програми

Lexer.h

```
#ifndef _LEXER_H
#define _LEXER_H

#include <iostream>
#include <sstream>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <fstream>

using namespace std;

class Lexer {
public:
	int AllTableRowSize;
	int AllTableClmnSize;

	char** tableToken;
	int indexTableToken;

	int** TokenPosition;
```

```
        int positionIndex;

        char** constTable;
        int* constCodeTable;
        int indexConstTable;

        char** identifierTable;
        int* idnCodeTable;
        int indexIdentTable;

        string* log;
        int indexLog;

        int* SymbolCategories;
        char reservWord[5][20] = { "0 PROGRAM","1 BEGIN","2 END","3 DEFFUNC", NULL };
        int idnResWord[5] = { 401, 402, 403, 404, NULL };

        int Constant;
        int Idn;

        int clmn;
        int clmnCount;
        int rowCount;
        int CommRowPos;
        int CommClmnPos;

        char* resw;
        char* tok;

        Lexer(int row, int column);
        ~Lexer();

        void SymSort();
        void divideToken(char sym, ifstream& file);
};

#endif // _LEXER_H
```

Lexer.cpp

```
#include "Lexer.h"
```

```
Lexer::Lexer(int row, int column)
 {
        Constant = 501;
        Idn = 1001;
        rowCount = 0;
        clmn = 0;
        clmnCount = 0;

        Comm = '0';
        startComm = 0;
        endComm = 0;
        commCheck = false;
        openComm = false;

        digitString = false;
        alphaString = false;


        positionIndex = 0;
        indexTableToken = 0;
        indexConstTable = 0;
        indexIdentTable = 0;
        indexLog = 0;

        reservWord[0][9] = NULL;
        reservWord[1][7] = NULL;
        reservWord[2][5] = NULL;
        reservWord[3][9] = NULL;

        TokenPosition = new int* [row];
        tableToken = new char* [row];
        constTable = new char* [row];
        identifierTable = new char* [row];
        SymbolCategories = new int[128];

        resw = new char[row];
        tok = new char[row];
        log = new string[row];

        for (int i = 0; i < row; i++)
        {
                TokenPosition[i] = new int[column];
                tableToken[i] = new char[column];
                constTable[i] = new char[column];
                identifierTable[i] = new char[column];
        }

        for (int i = 0; i < row; i++)
        {
                for (int j = 0; j < column; j++)
                {
```

```
                        tableToken[i][j] = NULL;
                        TokenPosition[i][j] = NULL;
                        constTable[i][j] = NULL;
                        identifierTable[i][j] = NULL;
                }
        }
}

Lexer::~Lexer()
{
        for (int i = 0; i < 30; i++)
        {
                delete[] TokenPosition[i], tableToken[i], constTable[i], identifierTable[i],
SymbolCategories, resw, tok, log;
        }
}

void Lexer::SymSort() {

        for (int i = 0; i < 128; i++)
        {
                SymbolCategories[i] = 6;

                if (((8 <= i)&&(i <= 13)) || (i == 32))
                        SymbolCategories[i] = 0;

                if ((48 <= i)&&(i <= 57))
                        SymbolCategories[i] = 1;

                if (((i > 64) && (i < 91)) || ((i > 96) && (i < 123)))
                        SymbolCategories[i] = 2;

                if ((i == 59) || (i == 46) || (i == 61) || (i == 44) || (i == 92))
                        SymbolCategories[i] = 3;

                if (i == 40)
                        SymbolCategories[i] = 5;
        }
}

void Lexer::divideToken(char sym) {
        char num[1];
        stringstream fullErr;

        if (sym == '\n')
        {
                rowCount++;
                clmnCount = 0;
                if (openComm == true)
                        return;
        }
```

```
if (openComm == true)
{
	if (sym == '*')
	{
		endComm = clmnCount;
		clmnCount++;
		return;
	}
	if ((sym == ')') && (clmnCount - endComm == 1))
	{
		openComm = false;
		commCheck = false;
		clmnCount++;
		return;
	}

}

if ((openComm == true)&&(sym == EOF))
{
	try {
		throw (string)("Lexer: Error( You must close comment! )");
	}
	catch (string err) {
		log[indexLog] = err;
		indexLog++;
	}
	return;
}

if (openComm == false)
{
	if (commCheck == true)
	{
		if ((sym == '*') && (clmnCount - startComm == 1))
		{
			openComm = true;
			Comm = '0';

			clmnCount++;
			return;
		}
		else
		{
			commCheck = false;
		}
	}

	if (SymbolCategories[(int)sym] == 5) {
		commCheck = true;
		startComm = clmnCount;
		Comm = sym;
```

```
        clmnCount++;
}


if (alphaString == false) {
        if (digitString == false) {
                if (SymbolCategories[(int)sym] == 1) {
                        TokenPosition[positionIndex][1] = clmnCount + 1;
                        TokenPosition[positionIndex][0] = rowCount + 1;
                        TokenPosition[positionIndex][2] = Constant;

                        tableToken[indexTableToken][clmn] = sym;
                        constTable[indexConstTable][clmn] = sym;

                        positionIndex++;
                        clmnCount++;
                        clmn++;

                        digitString = true;
                }
        }
        else {
                if (SymbolCategories[(int)sym] == 1)
                {
                        tableToken[indexTableToken][clmn] = sym;
                        constTable[indexConstTable][clmn] = sym;

                        clmnCount++;
                        clmn++;
                }
                else {
                        indexConstTable++;
                        Constant++;
                        indexTableToken++;

                        clmn = 0;
                        digitString = false;
                }
        }
}

if ((SymbolCategories[(int)sym] == 0) && (sym != '\n')) {
        clmnCount++;
}


if (alphaString == false) {
        if (SymbolCategories[(int)sym] == 2) {
                TokenPosition[positionIndex][1] = clmnCount + 1;
                TokenPosition[positionIndex][0] = rowCount + 1;
```

```
                        tok[clmn] = sym;
                        tableToken[indexTableToken][clmn] = sym;

                        clmnCount++;
                        clmn++;

                        alphaString = true;
                }
        }
        else {
                if ((SymbolCategories[(int)sym] == 2) || (SymbolCategories[(int)sym] == 1))
{
                        tok[clmn] = sym;
                        tableToken[indexTableToken][clmn] = sym;

                        clmn++;
                        clmnCount++;
                }
                else {
                        bool f = false;
                        tok[clmn] = NULL;

                        for (int k = 0; reservWord[k][0]; k++)
                        {
                                for (int j = 0; reservWord[k][j]; j++)
                                {
                                        resw[j] = reservWord[k][j + 2];
                                }

                                if (strcmp(resw, tok) == 0)
                                {
                                        num[0] = { reservWord[k][0] };
                                        TokenPosition[positionIndex][2] =
idnResWord[atoi(num)];
                                        f = true;

                                        break;
                                }
                        }

                        if (f == false)
                        {
                                for (int j = 0; tok[j]; j++)
                                {
                                        identifierTable[indexIdentTable][j] = tok[j];
                                }
                                TokenPosition[positionIndex][2] = Idn;

                                indexIdentTable++;
                                Idn++;
                        }
```

```
                        alphaString = false;
                        clmn = 0;
                        indexTableToken++;
                        positionIndex++;
                }

        }

        if (SymbolCategories[(int)sym] == 3) {
                tableToken[indexTableToken][0] = sym;

                TokenPosition[positionIndex][1] = clmnCount + 1;
                TokenPosition[positionIndex][0] = rowCount + 1;
                TokenPosition[positionIndex][2] = (int)sym;

                clmnCount++;
                positionIndex++;
                indexTableToken++;
        }

        if ((SymbolCategories[(int)sym] == 6) || ((Comm=='(') && (commCheck == false)))
{
                try {
                        throw ((string)"Lexer: Error( Undefined symbol was found:");
                }
                catch (string err) {
                        if ((Comm == '(') && (commCheck == false)) {
                                fullErr << err << "^" << Comm << "^)(row: " << rowCount +
1 << ", column: " << startComm + 1 << ")";
                                log[indexLog] = fullErr.str();
                                fullErr.str("");
                                indexLog++;

                                Comm = '0';
                        }
                        if (SymbolCategories[(int)sym] == 6){
                                clmnCount++;
                                fullErr << err << "^" << sym << "^)(row: " << rowCount + 1
<< ", column: " << clmnCount << ")";

                                log[indexLog] = fullErr.str();
                                fullErr.str("");
                                indexLog++;

                        }
                }
        }
    }
    else {
        clmnCount++;
    }
}
#include "Lexer.h"
#define cnst 100

Lexer::Lexer(int row, int column)
```

```
{
	AllTableRowSize = row;
	AllTableClmnSize = clmn;

	Constant = 501;
	Idn = 1001;

	clmn = 0;
	clmnCount = 0;
	rowCount = 0;
	CommRowPos = 0;
	CommClmnPos = 0;

	indexTableToken = 0;
	indexConstTable = 0;
	indexIdentTable = 0;
	indexLog = 0;

	reservWord[0][9] = NULL;
	reservWord[1][7] = NULL;
	reservWord[2][5] = NULL;
	reservWord[3][9] = NULL;

	TokenPosition = new int* [row];
	tableToken = new char* [row];

	constTable = new char* [row];
	constCodeTable = new int[row]
		;
	identifierTable = new char* [row];
	idnCodeTable = new int[row];
	SymbolCategories = new int[128];

	resw = new char[row];
	tok = new char[row];
	log = new string[row];

	for (int i = 0; i < row; i++)
	{
		constCodeTable[i] = NULL;
		idnCodeTable[i] = NULL;

		TokenPosition[i] = new int[column];
		tableToken[i] = new char[column];
		constTable[i] = new char[column];
		identifierTable[i] = new char[column];
	}

	for (int i = 0; i < row; i++)
	{
		for (int j = 0; j < column; j++)
		{
			tableToken[i][j] = NULL;
			TokenPosition[i][j] = NULL;
			constTable[i][j] = NULL;
			identifierTable[i][j] = NULL;
		}
	}
}

Lexer::~Lexer()
{
	for (int i = 0; i < AllTableRowSize; i++)
	{
		delete[] TokenPosition[i], tableToken[i], constTable[i], identifierTable[i],
SymbolCategories, resw, tok, log;
	}
}
```

```
void Lexer::SymSort() {

	for (int i = 0; i < 128; i++)
	{
		SymbolCategories[i] = 6;

		if (((8 <= i)&&(i <= 13)) || (i == 32))
			SymbolCategories[i] = 0;

		if ((48 <= i)&&(i <= 57))
			SymbolCategories[i] = 1;

		if (((i > 64) && (i < 91)) || ((i > 96) && (i < 123)))
			SymbolCategories[i] = 2;

		if ((i == 59) || (i == 46) || (i == 61) || (i == 44) || (i == 92))
			SymbolCategories[i] = 3;

		if (i == 40)
			SymbolCategories[i] = 5;
	}
}

void Lexer::divideToken(char sym, ifstream& file) {
	char num[1];
	stringstream fullErr;

	if (sym == '\n')
	{
		rowCount++;
		clmnCount = 0;
		file.get(sym);
	}

	if (SymbolCategories[(int)sym] == 1)
	{
		char tok[cnst] = { NULL };
		TokenPosition[indexTableToken][0] = rowCount + 1;
		TokenPosition[indexTableToken][1] = clmnCount + 1;

		for (int i = 0; SymbolCategories[(int)sym] == 1; i++)
		{
			tableToken[indexTableToken][clmn] = sym;
			tok[clmn] = sym;
			file.get(sym);
			clmn++;
			clmnCount++;
		}
		tok[clmn] = NULL;
		clmn = 0;

		bool f = false;
		for (int i = 0; constTable[i][0]; i++)
		{
			for (int j = 0; constTable[i][j]; j++)
			{
				if ((tok[j] == constTable[i][j]) && ((tok[j] != NULL) &&
(constTable[i][j] != NULL)))
				{
					f = true;
				}
				else
				{
					f = false;
					break;
				}
			}
			if (f == true)
			{
```

```
                        TokenPosition[indexTableToken][2] = constCodeTable[i];
                        break;
                }
        }

        if (f == false)
        {
                TokenPosition[indexTableToken][2] = Constant;
                constCodeTable[indexConstTable] = Constant;

                for (int i = 0; tok[i]; i++)
                {
                        constTable[indexConstTable][i] = tok[i];
                }
                Constant++;
                indexConstTable++;
        }
        indexTableToken++;
    }

    if (SymbolCategories[(int)sym] == 2)
    {
        char tok[cnst] = { NULL };
        TokenPosition[indexTableToken][0] = rowCount + 1;
        TokenPosition[indexTableToken][1] = clmnCount + 1;

        for (int i = 0; (SymbolCategories[(int)sym] == 1) || (SymbolCategories[(int)sym] ==
2); i++)
        {
                tableToken[indexTableToken][clmn] = sym;
                tok[clmn] = sym;
                file.get(sym);
                clmn++;
                clmnCount++;
                if (file.eof())
                        break;
        }
        tok[clmn] = NULL;
        clmn = 0;

        bool res_f = false;
        for (int k = 0; reservWord[k][0]; k++)
        {
                for (int j = 0; reservWord[k][j]; j++)
                {
                        resw[j] = reservWord[k][j + 2];
                }

                if (strcmp(resw, tok) == 0)
                {
                        num[0] = { reservWord[k][0] };
                        TokenPosition[indexTableToken][2] = idnResWord[atoi(num)];
                        res_f = true;
                        break;
                }
        }

        if (res_f == false)
        {
                bool f = false;
                for (int i = 0; identifierTable[i][0]; i++)
                {
                        for (int j = 0; identifierTable[i][j]; j++)
                        {
                                if ((tok[j] == identifierTable[i][j]) && ((tok[j] != NULL) &&
(identifierTable[i][j] != NULL)))
                                        f = true;
                                else
                                {
```

```
                                f = false;
                                break;
                        }
                }
                if (f == true)
                {
                        TokenPosition[indexTableToken][2] = idnCodeTable[i];
                        break;
                }
        }

        if (f == false)
        {
                TokenPosition[indexTableToken][2] = Idn;
                idnCodeTable[indexIdentTable] = Idn;

                for (int i = 0; tok[i]; i++)
                {
                        identifierTable[indexIdentTable][i] = tok[i];
                }
                Idn++;
                indexIdentTable++;
        }
    }
    indexTableToken++;
}

if (SymbolCategories[(int)sym] == 3)
{
    TokenPosition[indexTableToken][0] = rowCount + 1;
    TokenPosition[indexTableToken][1] = clmnCount + 1;
    TokenPosition[indexTableToken][2] = (int)sym;

    tableToken[indexTableToken][0] = sym;
    indexTableToken++;
    clmnCount++;
}

if (SymbolCategories[(int)sym] == 5)
{
    clmnCount++;
    file.get(sym);
    CommRowPos = rowCount;
    CommClmnPos = clmnCount;

    if (sym == '*')
    {
        bool comm = false;
        clmnCount++;
        for (int i = 0; !file.eof(); i++)
        {
            file.get(sym);
            clmnCount++;
            if (sym == '*')
            {
                file.get(sym);
                clmnCount++;
                if (sym == ')')
                {
                    comm = true;
                    return;
                }
            }
            if (sym == '\n')
            {
                clmnCount = 0;
                rowCount++;
            }
        }
```

```
                        if (comm == false)
                        {
                                try {
                                        throw (string)("Lexer: Error( You must close comment! )");
                                }
                                catch (string err) {
                                        fullErr << err << "(row: " << CommRowPos + 1 << ", column: " <<
CommClmnPos << ")";

                                        log[indexLog] = fullErr.str();
                                        fullErr.str("");
                                        indexLog++;
                                }

                        }
                }
                else
                {
                        try {
                                throw ((string)"Lexer: Error( Undefined symbol was found:");
                        }
                        catch (string err) {
                                fullErr << err << "^" << sym << "^)(row: " << rowCount << ", column:
" << clmnCount << ")";

                                log[indexLog] = fullErr.str();
                                fullErr.str("");
                                indexLog++;
                        }
                }
        }

        if (SymbolCategories[(int)sym] == 0)
        {
                if (sym == '\n')
                {
                        rowCount++;
                }
                clmnCount++;
        }

        if (SymbolCategories[(int)sym] == 6)
        {
                try {
                        throw ((string)"Lexer: Error( Undefined symbol was found:");
                }
                catch (string err) {
                        fullErr << err << "^" << sym << "^)(row: " << rowCount + 1 << ", column: "
<< clmnCount + 1 << ")";

                        log[indexLog] = fullErr.str();
                        fullErr.str("");
                        indexLog++;
                }
                clmnCount++;
        }
}
```

Print.h

```
#ifndef _PRINT_H
#define _PRINT_H
#include <iostream>
#include <fstream>

using namespace std;

#pragma warning(disable : 4996)

int printCodingTable(int** tokenPosition, char** Token, string* ErrorLog, int indexErr, char*
testPath);
int printConst_IdnTable(char** ConstTable, char** IdnTable, char* testPath);
int printParserTable(char* testPath, struct treeSignal PrSig, struct treeProg MainProg, struct
treeBlock PrBlock, struct treeDeclarations PrDeclar, struct treeMathFunc PrMathFunc, struct
treeFuncList* PrFuncList);
char** testPath(char** argv);

#endif // !_PRINT_H
```

Print.cpp

```
#include "Print.h"
#include "ParserTree.h"

int printCodingTable(int** tokenPosition, char** Token, string* ErrorLog, int indexErr, char* testPath)
{
	ofstream outFile;

	outFile.open(testPath);
	if (outFile.is_open()) {
		cout << "File (generated.txt) OK!\n\n";
	}
	else {
		cout << "File (generated.txt) error!\n\n";
		return(0);
	}

	cout << "|  Row   | Column|TokCode| Token  |" << endl;
	outFile << "| Row |Column |TokCode |Token |\n";
	for (int i = 0; Token[i][0]; i++)
	{
		cout << "  " << tokenPosition[i][0] << "\t | " << tokenPosition[i][1] << "\t | " <<
tokenPosition[i][2] << "\t | ";
		outFile << "   " << tokenPosition[i][0] << " \t " << tokenPosition[i][1] << " \t " <<
tokenPosition[i][2] << " \t";
		for (int j = 0; Token[i][j]; j++)
		{
			cout << Token[i][j];
			outFile << Token[i][j];
		}
		cout << endl;
		outFile << endl;
	}
	outFile << endl;

	for (int i = 0; i < indexErr; i++)
```

```
	{
		cout << ErrorLog[i] << endl;
		outFile << ErrorLog[i].c_str() << endl;
	}
	outFile << endl;

	outFile.close();
}

int printConst_IdnTable(char** ConstTable, char** IdnTable, char* testPath)
{
	ofstream outFile;

	outFile.open(testPath, ios_base::app);
	if (outFile.is_open()) {
		cout << "File (generated.txt) OK!\n\n";
	}
	else {
		cout << "File (generated.txt) error!\n\n";
		return(0);
	}

	outFile << "Constant table \n\n";
	cout << "Constant table" << endl;

	for (int i = 0; ConstTable[i][0]; i++)
	{
		outFile << "\t";
		for (int j = 0; ConstTable[i][j]; j++)
		{
			cout << ConstTable[i][j];
			outFile << ConstTable[i][j];
		}
		cout << endl;
		outFile << endl;
	}
	outFile << "\n\n";

	outFile << "Identifier table \n\n";
	cout << "Identifier table" << endl;

	for (int i = 0; IdnTable[i][0]; i++)
	{
		outFile << "\t";
		for (int j = 0; IdnTable[i][j]; j++)
		{
			cout << IdnTable[i][j];
			outFile << IdnTable[i][j];
		}
		cout << endl;
		outFile << endl;
	}
	outFile << endl;

	outFile.close();
```

```
}


int printParserTable(char* testPath, treeSignal PrSig, treeProg MainProg, treeBlock PrBlock,
treeDeclarations PrDeclar, treeMathFunc PrMathFunc, treeFuncList* PrFuncList) {
	ofstream outFile;

	outFile.open(testPath);
	if (outFile.is_open()) {
		cout << "File (generated.txt) OK!\n\n";
	}
	else {
		cout << "File (generated.txt) error!\n\n";
		return(0);
	}

	cout << PrSig.SigName << endl;
	outFile << PrSig.SigName << endl;

	for (int i = 0; i < MainProg.treeLVL; i++)
	{
		cout << ".";
		outFile << ".";
	}
	cout << MainProg.ProgName << endl;
	outFile << MainProg.ProgName << endl;

	if (MainProg.Program.code != NULL)
	{
		for (int i = 0; i < MainProg.treeLVL + 2; i++)
		{
			cout << ".";
			outFile << ".";
		}
		cout << MainProg.Program.code << " ";
		outFile << MainProg.Program.code << " ";
		for (int j = 0; MainProg.Program.value[j]; j++)
		{
			cout << MainProg.Program.value[j];
			outFile << MainProg.Program.value[j];
		}
		cout << endl;
		outFile << endl;

		for (int i = 0; i < MainProg.treeLVL + 2; i++)
		{
			cout << ".";
			outFile << ".";
		}
		cout << MainProg.ProgIDN_Name << endl;
		outFile << MainProg.ProgIDN_Name << endl;

		for (int i = 0; i < MainProg.treeLVL + 4; i++)
		{
			cout << ".";
```

```
		outFile << ".";
	}
	cout << MainProg.IDN_Name << endl;
	outFile << MainProg.IDN_Name << endl;

	if (MainProg.ProgIDN.code != NULL) {
		for (int i = 0; i < MainProg.treeLVL + 6; i++)
		{
			cout << ".";
			outFile << ".";
		}
		cout << MainProg.ProgIDN.code << " ";
		outFile << MainProg.ProgIDN.code << " ";
		for (int j = 0; MainProg.ProgIDN.value[j]; j++)
		{
			cout << MainProg.ProgIDN.value[j];
			outFile << MainProg.ProgIDN.value[j];
		}
		cout << endl;
		outFile << endl;

		if (MainProg.Semicolon.code != NULL)
		{
			for (int i = 0; i < MainProg.treeLVL + 2; i++)
			{
				cout << ".";
				outFile << ".";
			}
			cout << MainProg.Semicolon.code << " " << MainProg.Semicolon.value[0]
<< endl;
			outFile << MainProg.Semicolon.code << " " <<
MainProg.Semicolon.value[0] << endl;
		}
		else
		{
			cout << MainProg.ErrorLog << endl;
			outFile << MainProg.ErrorLog << endl;
			outFile.close();
			return 0;
		}
	}
	else
	{
		cout << MainProg.ErrorLog << endl;
		outFile << MainProg.ErrorLog << endl;
		outFile.close();
		return 0;
	}
}
else
{
	cout << MainProg.ErrorLog << endl;
	outFile << MainProg.ErrorLog << endl;
	outFile.close();
	return 0;
```

```
}

for (int i = 0; i < PrBlock.treeLVL; i++)
{
	cout << ".";
	outFile << ".";
}
cout << PrBlock.BlockName << endl;
outFile << PrBlock.BlockName << endl;

for (int i = 0; i < PrDeclar.treeLVL; i++)
{
	cout << ".";
	outFile << ".";
}
cout << PrDeclar.DeclarationsName << endl;
outFile << PrDeclar.DeclarationsName << endl;

for (int i = 0; i < PrMathFunc.treeLVL; i++)
{
	cout << ".";
	outFile << ".";
}
cout << PrMathFunc.MathFuncName << endl;
outFile << PrMathFunc.MathFuncName << endl;

if (PrMathFunc.Deffunc.code != NULL)
{
	for (int i = 0; i < PrMathFunc.treeLVL + 2; i++)
	{
		cout << ".";
		outFile << ".";
	}
	cout << PrMathFunc.Deffunc.code << " ";
	outFile << PrMathFunc.Deffunc.code << " ";
	for (int j = 0; PrMathFunc.Deffunc.value[j]; j++)
	{
		cout << PrMathFunc.Deffunc.value[j];
		outFile << PrMathFunc.Deffunc.value[j];
	}
	cout << endl;
	outFile << endl;

	while (PrFuncList)
	{
		if (PrFuncList->AnotherFunc.FuncIDN.code != NULL)
		{
			for (int i = 0; i < PrFuncList->treeLVL; i++)
			{
				cout << ".";
				outFile << ".";
			}
			cout << PrFuncList->FuncListName << endl;
			outFile << PrFuncList->FuncListName << endl;
```

```
for (int i = 0; i < PrFuncList->AnotherFunc.treeLVL; i++)
{
	cout << ".";
	outFile << ".";
}
cout << PrFuncList->AnotherFunc.FuncName << endl;
outFile << PrFuncList->AnotherFunc.FuncName << endl;

for (int i = 0; i < PrFuncList->AnotherFunc.treeLVL + 2; i++)
{
	cout << ".";
	outFile << ".";
}
cout << PrFuncList->AnotherFunc.FuncIDN_Name << endl;
outFile << PrFuncList->AnotherFunc.FuncIDN_Name << endl;

for (int i = 0; i < PrFuncList->AnotherFunc.treeLVL + 4; i++)
{
	cout << ".";
	outFile << ".";
}
cout << PrFuncList->AnotherFunc.IDN_Name << endl;
outFile << PrFuncList->AnotherFunc.IDN_Name << endl;

for (int i = 0; i < PrFuncList->AnotherFunc.treeLVL + 6; i++)
{
	cout << ".";
	outFile << ".";
}
cout << PrFuncList->AnotherFunc.FuncIDN.code << " ";
outFile << PrFuncList->AnotherFunc.FuncIDN.code << " ";
for (int j = 0; PrFuncList->AnotherFunc.FuncIDN.value[j]; j++)
{
	cout << PrFuncList->AnotherFunc.FuncIDN.value[j];
	outFile << PrFuncList->AnotherFunc.FuncIDN.value[j];
}
cout << endl;
outFile << endl;

if (PrFuncList->AnotherFunc.Equ.code != NULL)
{
	for (int i = 0; i < PrFuncList->AnotherFunc.treeLVL + 6; i++)
	{
		cout << ".";
		outFile << ".";
	}
	cout << PrFuncList->AnotherFunc.Equ.code << " " << PrFuncList-
>AnotherFunc.Equ.value[0] << endl;
	outFile << PrFuncList->AnotherFunc.Equ.code << " " <<
PrFuncList->AnotherFunc.Equ.value[0] << endl;

	if (PrFuncList->AnotherFunc.Constant.code != NULL)
	{
		for (int i = 0; i < PrFuncList->AnotherFunc.treeLVL + 6;
i++)
```

```
{
	cout << ".";
	outFile << ".";
}
cout << PrFuncList->AnotherFunc.Constant.code << " ";
outFile << PrFuncList->AnotherFunc.Constant.code << " ";
for (int j = 0; PrFuncList->AnotherFunc.Constant.value[j];
j++)
{
	cout << PrFuncList->AnotherFunc.Constant.value[j];
	outFile << PrFuncList-
>AnotherFunc.Constant.value[j];
}
cout << endl;
outFile << endl;

for (int i = 0; i < PrFuncList-
>AnotherFunc.FuncCharacteristic.treeLVL; i++)
{
	cout << ".";
	outFile << ".";
}
cout << PrFuncList-
>AnotherFunc.FuncCharacteristic.FuncCharName << endl;
outFile << PrFuncList-
>AnotherFunc.FuncCharacteristic.FuncCharName << endl;

if (PrFuncList-
>AnotherFunc.FuncCharacteristic.BackSlash.code != NULL)
{
	for (int i = 0; i < PrFuncList-
>AnotherFunc.FuncCharacteristic.treeLVL + 2; i++)
	{
		cout << ".";
		outFile << ".";
	}
	cout << PrFuncList-
>AnotherFunc.FuncCharacteristic.BackSlash.code << " " << PrFuncList-
>AnotherFunc.FuncCharacteristic.BackSlash.value[0] << endl;
	outFile << PrFuncList-
>AnotherFunc.FuncCharacteristic.BackSlash.code << " " << PrFuncList-
>AnotherFunc.FuncCharacteristic.BackSlash.value[0] << endl;
	if (PrFuncList-
>AnotherFunc.FuncCharacteristic.FirstConst.code != NULL)
	{
		for (int i = 0; i < PrFuncList-
>AnotherFunc.FuncCharacteristic.treeLVL + 2; i++)
		{
			cout << ".";
			outFile << ".";
		}
		cout << PrFuncList-
>AnotherFunc.FuncCharacteristic.FirstConst.code << " ";
		outFile << PrFuncList-
>AnotherFunc.FuncCharacteristic.FirstConst.code << " ";
```

```
for (int j = 0; PrFuncList->AnotherFunc.FuncCharacteristic.FirstConst.value[j]; j++)
{
	cout << PrFuncList->AnotherFunc.FuncCharacteristic.FirstConst.value[j];
	outFile << PrFuncList->AnotherFunc.FuncCharacteristic.FirstConst.value[j];
}
cout << endl;
outFile << endl;

if (PrFuncList->AnotherFunc.FuncCharacteristic.Coma.code != NULL)
{
	for (int i = 0; i < PrFuncList->AnotherFunc.FuncCharacteristic.treeLVL + 2; i++)
	{
		cout << ".";
		outFile << ".";
	}
	cout << PrFuncList->AnotherFunc.FuncCharacteristic.Coma.code << " " << PrFuncList->AnotherFunc.FuncCharacteristic.Coma.value[0] << endl;
	outFile << PrFuncList->AnotherFunc.FuncCharacteristic.Coma.code << " " << PrFuncList->AnotherFunc.FuncCharacteristic.Coma.value[0] << endl;

	if (PrFuncList->AnotherFunc.FuncCharacteristic.SecondConst.code != NULL)
	{
		for (int i = 0; i < PrFuncList->AnotherFunc.FuncCharacteristic.treeLVL + 2; i++)
		{
			cout << ".";
			outFile << ".";
		}
		cout << PrFuncList->AnotherFunc.FuncCharacteristic.SecondConst.code << " ";
		outFile << PrFuncList->AnotherFunc.FuncCharacteristic.SecondConst.code << " ";
		for (int j = 0; PrFuncList->AnotherFunc.FuncCharacteristic.SecondConst.value[j]; j++)
		{
			cout << PrFuncList->AnotherFunc.FuncCharacteristic.SecondConst.value[j];
			outFile << PrFuncList->AnotherFunc.FuncCharacteristic.SecondConst.value[j];
		}
		cout << endl;
		outFile << endl;
	}
	else
	{
		cout << PrFuncList->AnotherFunc.FuncCharacteristic.ErrorLog << endl;
```

```
                                                                        outFile << PrFuncList-
>AnotherFunc.FuncCharacteristic.ErrorLog << endl;
                                                                        outFile.close();
                                                                        return 0;
                                                                }
                                                        }
                                                        else
                                                        {
                                                                cout << PrFuncList-
>AnotherFunc.FuncCharacteristic.ErrorLog << endl;
                                                                outFile << PrFuncList-
>AnotherFunc.FuncCharacteristic.ErrorLog << endl;
                                                                outFile.close();
                                                                return 0;
                                                        }
                                                }
                                                else
                                                {
                                                        cout << PrFuncList-
>AnotherFunc.FuncCharacteristic.ErrorLog << endl;
                                                        outFile << PrFuncList-
>AnotherFunc.FuncCharacteristic.ErrorLog << endl;
                                                        outFile.close();
                                                        return 0;
                                                }
                                        }
                                        else
                                        {
                                                cout << PrFuncList-
>AnotherFunc.FuncCharacteristic.ErrorLog << endl;
                                                outFile << PrFuncList-
>AnotherFunc.FuncCharacteristic.ErrorLog << endl;
                                                outFile.close();
                                                return 0;
                                        }
                                        if (PrFuncList->AnotherFunc.Semicolon.code != NULL)
                                        {
                                                for (int i = 0; i < PrFuncList->AnotherFunc.treeLVL
+ 2; i++)
                                                {
                                                        cout << ".";
                                                        outFile << ".";
                                                }
                                                cout << PrFuncList->AnotherFunc.Semicolon.code
<< " " << PrFuncList->AnotherFunc.Semicolon.value[0] << endl;
                                                outFile << PrFuncList-
>AnotherFunc.Semicolon.code << " " << PrFuncList->AnotherFunc.Semicolon.value[0] << endl;
                                        }
                                        else
                                        {
                                                cout << PrFuncList->AnotherFunc.ErrorLog <<
endl;
                                                outFile << PrFuncList->AnotherFunc.ErrorLog <<
endl;
                                                outFile.close();
```

```
                                        return 0;
                                }
                        }
                        else
                        {
                                cout << PrFuncList->AnotherFunc.ErrorLog << endl;
                                outFile << PrFuncList->AnotherFunc.ErrorLog << endl;
                                outFile.close();
                                return 0;
                        }
                }
                else
                {
                        cout << PrFuncList->AnotherFunc.ErrorLog << endl;
                        outFile << PrFuncList->AnotherFunc.ErrorLog << endl;
                        outFile.close();
                        return 0;
                }

        }
        else
        {
                if (PrFuncList->Empty != "")
                {
                        for (int i = 0; i < PrFuncList->treeLVL; i++)
                        {
                                cout << ".";
                                outFile << ".";
                        }
                        cout << PrFuncList->FuncListName << endl;
                        outFile << PrFuncList->FuncListName << endl;

                        for (int i = 0; i < PrFuncList->treeLVL + 2; i++)
                        {
                                cout << ".";
                                outFile << ".";
                        }
                        cout << PrFuncList->Empty << endl;
                        outFile << PrFuncList->Empty << endl;
                        break;
                }
                else
                {
                        cout << PrFuncList->AnotherFunc.ErrorLog << endl;
                        outFile << PrFuncList->AnotherFunc.ErrorLog << endl;
                        outFile.close();
                        return 0;
                }
        }
        PrFuncList = PrFuncList->AnotherList;
    }
}
else
{
    if (PrMathFunc.Empty != "")
```

```
		{
			for (int i = 0; i < PrMathFunc.treeLVL + 2; i++)
			{
				cout << ".";
				outFile << ".";
			}
			cout << PrMathFunc.Empty << endl;
			outFile << PrMathFunc.Empty << endl;
		}
		else
		{
			cout << PrMathFunc.ErrorLog << endl;
			outFile << PrMathFunc.ErrorLog << endl;
			outFile.close();
			return 0;
		}
	}

	if (PrBlock.Begin.code != NULL)
	{
		for (int i = 0; i < PrBlock.treeLVL + 2; i++)
		{
			cout << ".";
			outFile << ".";
		}
		cout << PrBlock.Begin.code << " ";
		outFile << PrBlock.Begin.code << " ";
		for (int j = 0; PrBlock.Begin.value[j]; j++)
		{
			cout << PrBlock.Begin.value[j];
			outFile << PrBlock.Begin.value[j];
		}
		cout << endl;
		outFile << endl;

		for (int i = 0; i < PrBlock.treeLVL + 2; i++)
		{
			cout << ".";
			outFile << ".";
		}
		cout << PrBlock.StatementsList << endl;
		outFile << PrBlock.StatementsList << endl;

		if (PrBlock.End.code != NULL)
		{
			for (int i = 0; i < PrBlock.treeLVL + 4; i++)
			{
				cout << ".";
				outFile << ".";
			}
			cout << PrBlock.Empty << endl;
			outFile << PrBlock.Empty << endl;

			for (int i = 0; i < PrBlock.treeLVL + 2; i++)
			{
```

```
                        cout << ".";
                        outFile << ".";
                }
                cout << PrBlock.End.code << " ";
                outFile << PrBlock.End.code << " ";
                for (int j = 0; j < PrBlock.End.value[j]; j++)
                {
                        cout << PrBlock.End.value[j];
                        outFile << PrBlock.End.value[j];
                }
                cout << endl;
                outFile << endl;
        }
        else
        {
                cout << PrBlock.ErrorLog << endl;
                outFile << PrBlock.ErrorLog << endl;
                outFile.close();
                return 0;
        }
    }
    else
    {
        cout << PrBlock.ErrorLog << endl;
        outFile << PrBlock.ErrorLog << endl;
        outFile.close();
        return 0;
    }

    if (MainProg.Point.code != NULL)
    {
        for (int i = 0; i < MainProg.treeLVL + 2; i++)
        {
                cout << ".";
                outFile << ".";
        }
        cout << MainProg.Point.code << " " << MainProg.Point.value[0] << endl;
        outFile << MainProg.Point.code << " " << MainProg.Point.value[0] << endl;
    }
    else
    {
        cout << MainProg.ErrorLog << endl;
        outFile << MainProg.ErrorLog << endl;
        outFile.close();
        return 0;
    }

    outFile.close();
    return 0;
}

char** testPath(char** argv)
{
    char** tablePath;
    tablePath = new char* [100];
```

```
	for (int k = 0; k < 100; k++)
	{
		tablePath[k] = new char[100];
		for (int i = 0; i < 100; i++)
		{
			tablePath[k][i] = NULL;
		}
	}

	int lastColumn;

	char Input[] = "input.sig";
	char Generated[] = "generated.txt";

	for (int i = 0, indexPath = 0; argv[i + 1]; i++, indexPath += 2)
	{
		for (int j = 0; argv[i + 1][j]; j++)
		{
			tablePath[indexPath][j] = argv[i + 1][j];
			tablePath[indexPath + 1][j] = argv[i + 1][j];
			lastColumn = j + 1;
		}

		for (int k = 0; Input[k]; k++)
		{
			tablePath[indexPath][lastColumn + k] = Input[k];
		}

		for (int l = 0; Generated[l]; l++)
		{
			tablePath[indexPath + 1][lastColumn + l] = Generated[l];
		}
	}

	return tablePath;
}
```

Code_generator.h

```
#ifndef _CODE_GENERATOR_H
#define  _CODE_GENERATOR_H
#include "Parser.h"

class CodeGenerator {
public:
	int asmRow, asmClmn;
	char** Asm;

	int* IdnCheck;
	int checkIdx;

	string* CodeErrorLog;
	int errIdx;

	string* LexerErrLog;
	string ParserErrLog;

	CodeGenerator(int rows, int clmns, string* LexerErrors,string ParserError);
```

```
	void genProg(char* testPath, treeSignal PrSig, treeProg MainProg, treeBlock PrBlock,
treeDeclarations PrDeclar, treeMathFunc PrMathFunc, treeFuncList* PrFuncList);
};


#endif // !_CODE_GENERATOR_H
```

Code_generator.cpp

```
#include "Code_generator.h"

CodeGenerator::CodeGenerator(int rows, int clmns, string* LexerErrors, string ParserError)
{
	Asm = new char* [rows];
	asmRow = 0;
	asmClmn = 0;

	IdnCheck = new int [rows];
	checkIdx = 0;

	CodeErrorLog = new string[rows];
	errIdx = 0;

	LexerErrLog = LexerErrors;
	ParserErrLog = ParserError;

	for (int i = 0; i < rows; i++)
	{
		Asm[i] = new char[clmns];
		IdnCheck[i] = NULL;
		for (int j = 0; j < clmns; j++)
		{
			Asm[i][j] = NULL;
		}
	}
}

void CodeGenerator::genProg(char* testPath,treeSignal PrSig, treeProg MainProg, treeBlock
PrBlock, treeDeclarations PrDeclar, treeMathFunc PrMathFunc, treeFuncList* PrFuncList)
{
	ofstream lst;
	stringstream fullErr;
	bool flag;
	lst.open(testPath);


	if (MainProg.ErrorLog == "")
	{
		cout << "DATA_" << MainProg.ProgIDN.value << "\t SEGMENT byte\n";
		lst << "DATA_" << MainProg.ProgIDN.value << "\t SEGMENT byte\n";

		if ((PrMathFunc.Empty == "")&&(PrMathFunc.ErrorLog == ""))
		{
			treeFuncList* p = PrFuncList;
			while (p)
			{
				if ((p->Empty == "")&&(p->AnotherFunc.ErrorLog == "")&&(p-
>AnotherFunc.FuncCharacteristic.ErrorLog == ""))
				{
					flag = false;
					for (int i = 0; IdnCheck[i]; i++)
					{
						if (p->AnotherFunc.FuncIDN.code == IdnCheck[i])
						{
							fullErr << "Code Generator: Error( Name of math
function '" << p->AnotherFunc.FuncIDN.value << "' at [R:" << p->AnotherFunc.FuncIDN.row
```

```
                                                        << "][C:" << p->AnotherFunc.FuncIDN.clmn
<< "] must be unique!)\n";
                                                CodeErrorLog[errIdx] = fullErr.str();
                                                fullErr.str("");
                                                errIdx++;
                                                p->AnotherFunc.FuncIDN.code = -1;
                                                flag = true;
                                                break;
                                        }
                                }

                                if (atoi(p->AnotherFunc.FuncCharacteristic.SecondConst.value) <
1)
                                {
                                        fullErr << "Code Generator: Error( Value of second index
'" << p->AnotherFunc.FuncCharacteristic.SecondConst.value << "' at [R:" << p-
>AnotherFunc.FuncCharacteristic.SecondConst.row
                                                << "][C:" << p-
>AnotherFunc.FuncCharacteristic.SecondConst.clmn << "] can't be less then 1!)\n";
                                        CodeErrorLog[errIdx] = fullErr.str();
                                        fullErr.str("");
                                        errIdx++;
                                        p->AnotherFunc.FuncIDN.code = -1;
                                        flag = true;
                                        break;
                                }

                                if (flag == false)
                                {
                                        cout << "   " << p->AnotherFunc.FuncIDN.value << "\t\t "
<< atoi(p->AnotherFunc.FuncCharacteristic.SecondConst.value) + 1 << " DUP (?)\n";
                                        lst << "   " << p->AnotherFunc.FuncIDN.value << "\t\t "
<< atoi(p->AnotherFunc.FuncCharacteristic.SecondConst.value) + 1 << " DUP (?)\n";
                                }
                                IdnCheck[checkIdx] = p->AnotherFunc.FuncIDN.code;
                                checkIdx++;
                        }
                        p = p->AnotherList;
                }
        }

        cout << "DATA_"<< MainProg.ProgIDN.value <<"\t ENDS\n\nCODE_"<<
MainProg.ProgIDN.value <<"\t SEGMENT\n";
        lst << "DATA_" << MainProg.ProgIDN.value << "\t ENDS\n\nCODE_" <<
MainProg.ProgIDN.value <<"\t SEGMENT\n";

        if ((PrMathFunc.Empty == "")&&(PrMathFunc.ErrorLog == ""))
        {
                treeFuncList* f = PrFuncList;
                int label = 1;
                while (f)
                {
                        if ((f->Empty == "")&&(f->AnotherFunc.ErrorLog == "")&&(f-
>AnotherFunc.FuncCharacteristic.ErrorLog == "")&&(f->AnotherFunc.FuncIDN.code != -1))
                        {
                                if (atoi(f->AnotherFunc.FuncCharacteristic.FirstConst.value) >
atoi(f->AnotherFunc.FuncCharacteristic.SecondConst.value))
                                {
                                        fullErr << "Code Generator: Error( First index '" << f-
>AnotherFunc.FuncCharacteristic.FirstConst.value << "' at [R:" << f-
>AnotherFunc.FuncCharacteristic.FirstConst.row
                                                <<"][C:" << f-
>AnotherFunc.FuncCharacteristic.FirstConst.clmn << "] cant't be greater then second index)\n";
                                        CodeErrorLog[errIdx] = fullErr.str();
                                        fullErr.str("");
                                        errIdx++;
                                        break;
                                }
```

```
                                        else
                                        {
                                                cout << "\t MOV\t ECX," << f-
>AnotherFunc.FuncCharacteristic.FirstConst.value << endl;
                                                lst << "\t MOV\t ECX," << f-
>AnotherFunc.FuncCharacteristic.FirstConst.value << endl;

                                                cout << "\t MOV\t EAX," << f-
>AnotherFunc.FuncCharacteristic.SecondConst.value << endl;
                                                lst << "\t MOV\t EAX," << f-
>AnotherFunc.FuncCharacteristic.SecondConst.value << endl;
                                        }

                                        cout << "L" << label << "?:\n";
                                        lst << "L" << label << "?:\n";

                                        cout << "\t MOV\t " << f->AnotherFunc.FuncIDN.value << "[ECX *
2]," << f->AnotherFunc.Constant.value << endl;
                                        lst << "\t MOV\t " << f->AnotherFunc.FuncIDN.value << "[ECX *
2]," << f->AnotherFunc.Constant.value << endl;

                                        cout << "\t ADD\t ECX,1\n\t CMP\t ECX," << f-
>AnotherFunc.FuncCharacteristic.SecondConst.value << endl;
                                        lst << "\t ADD\t ECX,1\n\t CMP\t ECX, " << f-
>AnotherFunc.FuncCharacteristic.SecondConst.value << endl;

                                        cout << "\t JLE\t L"<< label << "?\n";
                                        lst << "\t JLE\t L"<<label<<"?\n";
                                        label++;
                                }
                                f = f->AnotherList;
                        }
                }

                if (PrBlock.Begin.code != NULL)
                {
                        cout << PrBlock.Begin.value << ":\n";
                        lst << PrBlock.Begin.value << ":\n";
                }

                cout << "CODE_" << MainProg.ProgIDN.value << "\t ENDS\n";
                lst << "CODE_" << MainProg.ProgIDN.value << "\t ENDS\n";

                if ((PrBlock.Begin.value != NULL)&&(PrBlock.End.value != NULL))
                {
                        cout << "\t\t " << PrBlock.End.value << "\t " << PrBlock.Begin.value <<
"\n";
                        lst << "\t\t " << PrBlock.End.value << "\t " << PrBlock.Begin.value << "\n";
                }
        }

        for (int i = 0; LexerErrLog[i] != ""; i++)
        {
                cout << LexerErrLog[i] << endl;
                lst << LexerErrLog[i] << endl;
        }

        if (ParserErrLog != "")
        {
                cout << ParserErrLog << endl;
                lst << ParserErrLog << endl;
        }

        for (int i = 0; CodeErrorLog[i] != ""; i++)
        {
                cout << CodeErrorLog[i];
                lst << CodeErrorLog[i];
        }
```

```
	lst.close();
}
```

Lab2.cpp

```
#include "Code_generator.h"
#include "Print.h"

int main(int argc, char** argv) {
	ifstream fileIn;
	const int tableRow = 100, tableColumn = 100;

	char** Path = testPath(argv);

	for (int var = 0; Path[var]; var++) {
		if (Path[var] == NULL)
			return 0;
		fileIn.open(Path[var]);
		if (fileIn.is_open()) {
			cout << "File " << (var + 2) / 2 << "(input.sig) OK!\n\n";
		}
		else {
			cout << "File " << (var + 2) / 2 << "(input.sig) error!\n\n";
			return(0);
		}
		var++;

		Lexer LA(tableRow, tableColumn);

		char ch = '0';
		LA.SymSort();

		while (ch) {
			fileIn.get(ch);
			if (fileIn.eof()) {
				LA.divideToken(EOF, fileIn);
				break;
			}
			else {
				LA.divideToken(ch, fileIn);
			}
		}

		Parser PA(tableRow, tableColumn, LA.TokenPosition, LA.tableToken, LA.rowCount,
LA.clmnCount);
		PA.Signal();

		//printCodingTable(LA.TokenPosition, LA.tableToken, LA.log, LA.indexLog,
Path[var]);
		//printConst_IdnTable(LA.constTable, LA.identifierTable, Path[var]);

		//printParserTable(Path[var], PA.Sig, PA.Prog, PA.MainBlock, PA.AllDeclarations,
PA.MathFunc, PA.FuncLists);

		CodeGenerator CG(tableColumn, tableRow, LA.log, PA.parserErrInfo.str());
		CG.genProg(Path[var], PA.Sig, PA.Prog, PA.MainBlock, PA.AllDeclarations,
PA.MathFunc, PA.FuncLists);

		fileIn.close();
	}

	return(0);
}
```

Parser.h

```
#ifndef _PARSER_H
```

```
#define _PARSER_H

#include "ParserTree.h"

class Parser {
public:
	int tokenCodePosition = 0;
	int** TableCodePosition;
	char** TokenTable;
	int LastRow;
	int LastClmn;
	stringstream parserErrInfo;

	treeSignal Sig;
	treeProg Prog;
	treeBlock MainBlock;
	treeDeclarations AllDeclarations;
	treeMathFunc MathFunc;
	treeFuncList* FuncLists;
	treeFunc FuncTMP;
	treeFuncChar FuncCharTMP;

	treeFuncList* LastList;
	treeFunc NullFunc;
	treeFuncChar NullFuncChar;

	treeFuncList* createHead(treeFuncList** List, bool EmptyFlag);
	treeFuncList* addNode(treeFuncList** List, bool EmptyFlag);
	bool funcExist;

	Parser(int row, int clmn, int** tokPosition, char** TokenCodTable, int lastRow, int
lastClmn);

	void Signal();
	bool ProgStart();
	bool Block();
	bool Declarations();
	bool MathFunction();
	bool FuncList();
	bool Func();
	bool FuncCharcteristic();
	bool Const();
	bool Identifier();
	bool Empty();
	bool EmptyMathFunc();
};

#endif // _PARSER_H
```

Parser.cpp

```
#include "Parser.h"

Parser::Parser(int row, int clmn, int** tokPosition, char** TokenCodTable, int lastRow, int lastClmn) {
	TableCodePosition = tokPosition;
	TokenTable = TokenCodTable;
	LastClmn = lastClmn;
	LastRow = lastRow;
	parserErrInfo.str() = "";
}

treeFuncList* Parser::createHead(treeFuncList** List, bool EmptyFlag)
{
	(*List) = new treeFuncList;
	(*List)->treeLVL = 10;
	(*List)->FuncListName = "<function-list>";
	if (EmptyFlag == false)
		(*List)->AnotherFunc = FuncTMP;
	else
		(*List)->Empty = "<empty>";
	(*List)->AnotherList = NULL;
	return (*List);
}

treeFuncList* Parser::addNode(treeFuncList** List, bool EmptyFlag) {

	if (*List)
	{
		treeFuncList* tmp = *List;
		while (tmp->AnotherList)
		{
			tmp = tmp->AnotherList;
		}
		tmp = new treeFuncList;
		tmp->treeLVL = 12;
		tmp->FuncListName = "<function-list>";
		tmp->AnotherList = NULL;
		if (EmptyFlag == false)
			tmp->AnotherFunc = FuncTMP;
		else
			tmp->Empty = "<empty>";
		(*List)->AnotherList = tmp;
		(*List) = tmp;
```

```
	}
	return 0;
}

void Parser::Signal() {
	Sig.SigName = "<signal-program>";
	ProgStart();
	Sig.StartProg = &Prog;
}

bool Parser::ProgStart() {
	TokenInfo tok;
	Prog.treeLVL = 2;
	Prog.ProgName = "<program>";
	if (TableCodePosition[tokenCodePosition][2] == 401)
	{
		tok.row = TableCodePosition[tokenCodePosition][0];
		tok.clmn = TableCodePosition[tokenCodePosition][1];
		tok.code = TableCodePosition[tokenCodePosition][2];
		for (int j = 0; TokenTable[tokenCodePosition][j]; j++)
		{
			tok.value[j] = TokenTable[tokenCodePosition][j];
		}
		Prog.Program = tok;
		tokenCodePosition++;

		Prog.ProgIDN_Name = "<procedure-identifier>";
		Prog.IDN_Name = "<identifier>";
		if (Identifier() == true)
		{
			tok.row = TableCodePosition[tokenCodePosition][0];
			tok.clmn = TableCodePosition[tokenCodePosition][1];
			tok.code = TableCodePosition[tokenCodePosition][2];

			for (int j = 0; TokenTable[tokenCodePosition][j]; j++)
			{
				tok.value[j] = TokenTable[tokenCodePosition][j];
				tok.value[j + 1] = NULL;
			}
			Prog.ProgIDN = tok;
			tokenCodePosition++;

			if (TableCodePosition[tokenCodePosition][2] == 59)
```

```
{
        tok.row = TableCodePosition[tokenCodePosition][0];
        tok.clmn = TableCodePosition[tokenCodePosition][1];
        tok.code = TableCodePosition[tokenCodePosition][2];
        tok.value[0] = TokenTable[tokenCodePosition][0];
        Prog.Semicolon = tok;
        tokenCodePosition++;

        if (Block() == true)
        {
                Prog.ProgBlock = &MainBlock;
                tokenCodePosition++;
                if (TableCodePosition[tokenCodePosition][2] == 46)
                {
                        tok.row = TableCodePosition[tokenCodePosition][0];
                        tok.clmn = TableCodePosition[tokenCodePosition][1];
                        tok.code = TableCodePosition[tokenCodePosition][2];
                        tok.value[0] = TokenTable[tokenCodePosition][0];
                        Prog.Point = tok;
                        return true;
                }
                else
                {
                        //err expected '.'
                        parserErrInfo << "Parser: Error( Symbol '.' expected, but '";

                        if (TableCodePosition[tokenCodePosition][0] != NULL) {
                                for (int j = 0; TokenTable[tokenCodePosition][j];
j++)
                                {
                                        parserErrInfo <<
TokenTable[tokenCodePosition][j];
                                }

                                parserErrInfo << "' found!)(row: " <<
TableCodePosition[tokenCodePosition][0] << ", column : " <<
TableCodePosition[tokenCodePosition][1] << ")";
                        }
                        else {
                                parserErrInfo << "Nothing' found! )(row: " <<
LastRow + 1 << ", column: " << LastClmn + 1 << ")";
                        }
                        Prog.ErrorLog = parserErrInfo.str();
                        return false;
```

```
                        }
                }
                else
                {
                        return false;
                }
        }
        else
        {
                //err expected ';'
                parserErrInfo << "Parser: Error( Symbol ';'  expected, but '";

                if (TableCodePosition[tokenCodePosition][0] != NULL) {
                        for (int j = 0; TokenTable[tokenCodePosition][j]; j++)
                        {
                                parserErrInfo << TokenTable[tokenCodePosition][j];
                        }

                        parserErrInfo << "' found!)(row: " <<
TableCodePosition[tokenCodePosition][0] << ", column : " <<
TableCodePosition[tokenCodePosition][1] << ")";
                }
                else {
                        parserErrInfo << "Nothing' found! )(row: " << LastRow + 1 << ",
column: " << LastClmn + 1 << ")";
                }
                Prog.ErrorLog = parserErrInfo.str();
                return false;
        }
}
else
{
        //err not PROGRAM Identifier
        parserErrInfo << "Parser: Error( Program identifier expected, but '";

        if (TableCodePosition[tokenCodePosition][0] != NULL) {
                for (int j = 0; TokenTable[tokenCodePosition][j]; j++)
                {
                        parserErrInfo << TokenTable[tokenCodePosition][j];
                }

                parserErrInfo << "' found!)(row: " <<
TableCodePosition[tokenCodePosition][0] << ", column : " <<
TableCodePosition[tokenCodePosition][1] << ")";
```

```
                        }
                        else {
                                parserErrInfo << "Nothing' found! )(row: " << LastRow + 1 << ", column: "
<< LastClmn + 1 << ")";
                        }
                        Prog.ErrorLog = parserErrInfo.str();
                        return false;
                }
        }
        else
        {
                //err not PROGRAM
                parserErrInfo << "Parser: Error( Reserved word 'PROGRAM' expected, but '";

                if (TableCodePosition[tokenCodePosition][0] != NULL) {
                        for (int j = 0; TokenTable[tokenCodePosition][j]; j++)
                        {
                                parserErrInfo << TokenTable[tokenCodePosition][j];
                        }

                        parserErrInfo << "' found!)(row: " << TableCodePosition[tokenCodePosition][0] <<
", column : " << TableCodePosition[tokenCodePosition][1] << ")";
                }
                else {
                        parserErrInfo << "Nothing' found! )(row: " << LastRow + 1 << ", column: " <<
LastClmn + 1 << ")";
                }
                Prog.ErrorLog = parserErrInfo.str();

                return false;
        }
}

bool Parser::Identifier() {
        if ((1001 <=
TableCodePosition[tokenCodePosition][2])&&(TableCodePosition[tokenCodePosition][2] <= 2001))
        {
                return true;
        }
        else {
                return false;
        }
}
```

```
bool Parser::Block() {
	TokenInfo tok;
	MainBlock.treeLVL = 4;
	MainBlock.BlockName = "<block>";

	if (Declarations() == true)
	{
		MainBlock.Declaration = &AllDeclarations;
		tokenCodePosition++;
		if (TableCodePosition[tokenCodePosition][2] == 402)
		{
			tok.row = TableCodePosition[tokenCodePosition][0];
			tok.clmn = TableCodePosition[tokenCodePosition][1];
			tok.code = TableCodePosition[tokenCodePosition][2];

			for (int j = 0; TokenTable[tokenCodePosition][j]; j++)
			{
				tok.value[j] = TokenTable[tokenCodePosition][j];
				tok.value[j + 1] = NULL;
			}
			MainBlock.Begin = tok;
			tokenCodePosition++;

			MainBlock.StatementsList = "<statements-list>";

			if (TableCodePosition[tokenCodePosition][2] == 403)
			{
				MainBlock.Empty = "<empty>";

				tok.row = TableCodePosition[tokenCodePosition][0];
				tok.clmn = TableCodePosition[tokenCodePosition][1];
				tok.code = TableCodePosition[tokenCodePosition][2];

				for (int j = 0; TokenTable[tokenCodePosition][j]; j++)
				{
					tok.value[j] = TokenTable[tokenCodePosition][j];
					tok.value[j + 1] = NULL;
				}
				MainBlock.End = tok;

				return true;
			}
			else
```

```
{
//err not END
parserErrInfo << "Parser: Error( Reserved word 'END' expected, but '";

if (TableCodePosition[tokenCodePosition][0] != NULL) {
for (int j = 0; TokenTable[tokenCodePosition][j]; j++)
{
parserErrInfo << TokenTable[tokenCodePosition][j];
}

parserErrInfo << "' found!)(row: " <<
TableCodePosition[tokenCodePosition][0] << ", column : " <<
TableCodePosition[tokenCodePosition][1] << ")";
}
else {
parserErrInfo << "Nothing' found! )(row: " << LastRow + 1 << ",
column: " << LastClmn + 1 << ")";
}
MainBlock.ErrorLog = parserErrInfo.str();
return false;
}
}
else
{
//err not BEGIN
parserErrInfo << "Parser: Error( Reserved word 'BEGIN' expected, but '";

if (TableCodePosition[tokenCodePosition][0] != NULL) {
for (int j = 0; TokenTable[tokenCodePosition][j]; j++)
{
parserErrInfo << TokenTable[tokenCodePosition][j];
}

parserErrInfo << "' found!)(row: " <<
TableCodePosition[tokenCodePosition][0] << ", column : " <<
TableCodePosition[tokenCodePosition][1] << ")";
}
else {
parserErrInfo << "Nothing' found! )(row: " << LastRow + 1 << ", column: "
<< LastClmn + 1 << ")";
}
MainBlock.ErrorLog = parserErrInfo.str();

return false;
```

```
		}
	}
	else
	{
		return false;
	}
}

bool Parser::Declarations() {
	AllDeclarations.treeLVL = 6;
	AllDeclarations.DeclarationsName = "<declarations>";

	if (MathFunction() == true)
	{
		AllDeclarations.MathFunctions = &MathFunc;
		return true;
	}
	else
	{
		return false;
	}
}

bool Parser::MathFunction() {
	TokenInfo tok;
	MathFunc.treeLVL = 8;
	MathFunc.MathFuncName = "<math-function-declarations>";
	if (TableCodePosition[tokenCodePosition][2] == 404)
	{
		tok.row = TableCodePosition[tokenCodePosition][0];
		tok.clmn = TableCodePosition[tokenCodePosition][1];
		tok.code = TableCodePosition[tokenCodePosition][2];

		for (int j = 0; TokenTable[tokenCodePosition][j]; j++)
		{
			tok.value[j] = TokenTable[tokenCodePosition][j];
			tok.value[j + 1] = NULL;
		}
		MathFunc.Deffunc = tok;
		tokenCodePosition++;

		if (FuncList() == true)
		{
```

```
                MathFunc.FunctionList = FuncLists;
                return true;
            }
            else
            {
                MathFunc.FunctionList = FuncLists;
                return false;
            }
        }
        else
        {
            if (Empty() == true)
            {
                MathFunc.Empty = "<empty>";
                tokenCodePosition--;
                return true;
            }
            else
            {
                //err not DEFFUNC & not empty
                parserErrInfo << "Parser: Error( Reserved word 'DEFFUNC' expected (or empty),
but '";

                if (TableCodePosition[tokenCodePosition][0] != NULL) {
                    for (int j = 0; TokenTable[tokenCodePosition][j]; j++)
                    {
                        parserErrInfo << TokenTable[tokenCodePosition][j];
                    }

                    parserErrInfo << "' found!)(row: " <<
TableCodePosition[tokenCodePosition][0] << ", column : " <<
TableCodePosition[tokenCodePosition][1] << ")";
                }
                else {
                    parserErrInfo << "Nothing' found! )(row: " << LastRow + 1 << ", column: "
<< LastClmn + 1 << ")";
                }
                MathFunc.ErrorLog = parserErrInfo.str();

                return false;
            }
        }

}
```

```
bool Parser::FuncList() {
	if (Empty() == true)
	{
		if (FuncLists == NULL)
		{
			createHead(&FuncLists, true);
			LastList = FuncLists;
		}
		else {
			addNode(&LastList, true);
		}
		tokenCodePosition--;
		return true;
	}
	else
	{
		funcExist = Func();

		if (FuncLists == NULL)
		{
			createHead(&FuncLists, false);
			LastList = FuncLists;
		}
		else {
			addNode(&LastList, false);
		}

		FuncTMP = NullFunc;
		FuncCharTMP = NullFuncChar;
	}
	tokenCodePosition++;

	if (funcExist == true)
	{
		bool funcEx = FuncList();
		return funcEx;
	}
	else
	{
		//err not empty & not identifier
		return false;
	}
```

```
}

bool Parser::Func() {
	TokenInfo tok;
	if (FuncLists == NULL)
		FuncTMP.treeLVL = 12;
	else
		FuncTMP.treeLVL = 14;
	FuncTMP.FuncName = "<function>";
	FuncTMP.FuncIDN_Name = "<function-identifier>";
	FuncTMP.IDN_Name = "<identifier>";

	if (Identifier() == true)
	{
		tok.row = TableCodePosition[tokenCodePosition][0];
		tok.clmn = TableCodePosition[tokenCodePosition][1];
		tok.code = TableCodePosition[tokenCodePosition][2];

		for (int j = 0; TokenTable[tokenCodePosition][j]; j++)
		{
			tok.value[j] = TokenTable[tokenCodePosition][j];
			tok.value[j + 1] = NULL;
		}
		FuncTMP.FuncIDN = tok;
		tokenCodePosition++;

		if (TableCodePosition[tokenCodePosition][2] == 61)
		{
			tok.row = TableCodePosition[tokenCodePosition][0];
			tok.clmn = TableCodePosition[tokenCodePosition][1];
			tok.code = TableCodePosition[tokenCodePosition][2];
			tok.value[0] = TokenTable[tokenCodePosition][0];
			FuncTMP.Equ = tok;
			tokenCodePosition++;

			if (Const() == true)
			{
				tok.row = TableCodePosition[tokenCodePosition][0];
				tok.clmn = TableCodePosition[tokenCodePosition][1];
				tok.code = TableCodePosition[tokenCodePosition][2];

				for (int j = 0; TokenTable[tokenCodePosition][j]; j++)
				{
```

```
                    tok.value[j] = TokenTable[tokenCodePosition][j];
                    tok.value[j + 1] = NULL;
                }
                FuncTMP.Constant = tok;
                tokenCodePosition++;

                if (FuncCharcteristic() == true)
                {
                    FuncTMP.FuncCharacteristic = FuncCharTMP;
                    tokenCodePosition++;
                    if (TableCodePosition[tokenCodePosition][2] == 59)
                    {
                        tok.row = TableCodePosition[tokenCodePosition][0];
                        tok.clmn = TableCodePosition[tokenCodePosition][1];
                        tok.code = TableCodePosition[tokenCodePosition][2];
                        tok.value[0] = TokenTable[tokenCodePosition][0];
                        FuncTMP.Semicolon = tok;
                        return true;
                    }
                    else
                    {
                        //err expected ';'
                        parserErrInfo << "Parser: Error( Symbol ';' expected, but '";

                        if (TableCodePosition[tokenCodePosition][0] != NULL) {
                            for (int j = 0; TokenTable[tokenCodePosition][j];
j++)
                            {
                                parserErrInfo <<
TokenTable[tokenCodePosition][j];
                            }

                            parserErrInfo << "' found!)(row: " <<
TableCodePosition[tokenCodePosition][0] << ", column : " <<
TableCodePosition[tokenCodePosition][1] << ")";
                        }
                        else {
                            parserErrInfo << "Nothing' found! )(row: " <<
LastRow + 1 << ", column: " << LastClmn + 1 << ")";
                        }
                        FuncTMP.ErrorLog = parserErrInfo.str();
                        return false;
                    }
                }
```

```
                else
                {
                        FuncTMP.FuncCharacteristic = FuncCharTMP;
                        return false;
                }
        }
        else
        {
                //err not CONST
                parserErrInfo << "Parser: Error( Constant expected, but '";

                if (TableCodePosition[tokenCodePosition][0] != NULL) {
                        for (int j = 0; TokenTable[tokenCodePosition][j]; j++)
                        {
                                parserErrInfo << TokenTable[tokenCodePosition][j];
                        }

                        parserErrInfo << "' found!)(row: " <<
TableCodePosition[tokenCodePosition][0] << ", column : " <<
TableCodePosition[tokenCodePosition][1] << ")";
                }
                else {
                        parserErrInfo << "Nothing' found! )(row: " << LastRow + 1 << ",
column: " << LastClmn + 1 << ")";
                }
                FuncTMP.ErrorLog = parserErrInfo.str();
                return false;
        }
}
else
{
        //err expected '='
        parserErrInfo << "Parser: Error( Symbol '=' expected, but '";

        if (TableCodePosition[tokenCodePosition][0] != NULL) {
                for (int j = 0; TokenTable[tokenCodePosition][j]; j++)
                {
                        parserErrInfo << TokenTable[tokenCodePosition][j];
                }

                parserErrInfo << "' found!)(row: " <<
TableCodePosition[tokenCodePosition][0] << ", column : " <<
TableCodePosition[tokenCodePosition][1] << ")";
        }
```

```
                else {
                        parserErrInfo << "Nothing' found! )(row: " << LastRow + 1 << ", column: "
<< LastClmn + 1 << ")";
                }
                FuncTMP.ErrorLog = parserErrInfo.str();
                return false;
        }
    }
    else
    {
        //err not Identifier
        parserErrInfo << "Parser: Error( Function identifier expected, but '";

        if (TableCodePosition[tokenCodePosition][0] != NULL) {
                for (int j = 0; TokenTable[tokenCodePosition][j]; j++)
                {
                        parserErrInfo << TokenTable[tokenCodePosition][j];
                }

                parserErrInfo << "' found!)(row: " << TableCodePosition[tokenCodePosition][0] <<
", column : " << TableCodePosition[tokenCodePosition][1] << ")";
        }
        else {
                parserErrInfo << "Nothing' found! )(row: " << LastRow + 1 << ", column: " <<
LastClmn + 1 << ")";
        }
        FuncTMP.ErrorLog = parserErrInfo.str();

        return false;
    }
}

bool Parser::Const() {
    if ((501 <=
TableCodePosition[tokenCodePosition][2])&&(TableCodePosition[tokenCodePosition][2] <= 1000))
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

```
bool Parser::FuncCharcteristic(){
	TokenInfo tok;
	if (FuncLists == NULL)
		FuncCharTMP.treeLVL = 18;
	else
		FuncCharTMP.treeLVL = 20;
	FuncCharTMP.FuncCharName = "<function-characteristic>";
	if (TableCodePosition[tokenCodePosition][2] == 92)
	{
		tok.row = TableCodePosition[tokenCodePosition][0];
		tok.clmn = TableCodePosition[tokenCodePosition][1];
		tok.code = TableCodePosition[tokenCodePosition][2];
		tok.value[0] = TokenTable[tokenCodePosition][0];
		FuncCharTMP.BackSlash = tok;
		tokenCodePosition++;

		if (Const() == true)
		{
			tok.row = TableCodePosition[tokenCodePosition][0];
			tok.clmn = TableCodePosition[tokenCodePosition][1];
			tok.code = TableCodePosition[tokenCodePosition][2];

			for (int j = 0; TokenTable[tokenCodePosition][j]; j++)
			{
				tok.value[j] = TokenTable[tokenCodePosition][j];
				tok.value[j + 1] = NULL;
			}
			FuncCharTMP.FirstConst = tok;
			tokenCodePosition++;

			if (TableCodePosition[tokenCodePosition][2] == 44)
			{
				tok.row = TableCodePosition[tokenCodePosition][0];
				tok.clmn = TableCodePosition[tokenCodePosition][1];
				tok.code = TableCodePosition[tokenCodePosition][2];
				tok.value[0] = TokenTable[tokenCodePosition][0];
				FuncCharTMP.Coma = tok;
				tokenCodePosition++;

				if (Const() == true)
				{
					tok.row = TableCodePosition[tokenCodePosition][0];
					tok.clmn = TableCodePosition[tokenCodePosition][1];
```

```
tok.code = TableCodePosition[tokenCodePosition][2];

for (int j = 0; TokenTable[tokenCodePosition][j]; j++)
{
	tok.value[j] = TokenTable[tokenCodePosition][j];
	tok.value[j + 1] = NULL;
}
FuncCharTMP.SecondConst = tok;
return true;
}
else
{
	//err not Constant
	parserErrInfo << "Parser: Error( Constant expected, but '";

	if (TableCodePosition[tokenCodePosition][0] != NULL) {
		for (int j = 0; TokenTable[tokenCodePosition][j]; j++)
		{
			parserErrInfo << TokenTable[tokenCodePosition][j];
		}

		parserErrInfo << "' found!)(row: " <<
TableCodePosition[tokenCodePosition][0] << ", column : " <<
TableCodePosition[tokenCodePosition][1] << ")";
	}
	else {
		parserErrInfo << "Nothing' found! )(row: " << LastRow + 1
<< ", column: " << LastClmn + 1 << ")";
	}
	FuncCharTMP.ErrorLog = parserErrInfo.str();
	return false;
}
}
else
{
	//err expected ','
	parserErrInfo << "Parser: Error( Symbol ',' expected, but '";

	if (TableCodePosition[tokenCodePosition][0] != NULL) {
		for (int j = 0; TokenTable[tokenCodePosition][j]; j++)
		{
			parserErrInfo << TokenTable[tokenCodePosition][j];
		}
```

```
                                        parserErrInfo << "' found!)(row: " <<
TableCodePosition[tokenCodePosition][0] << ", column : " <<
TableCodePosition[tokenCodePosition][1] << ")";
                                }
                                else {
                                        parserErrInfo << "Nothing' found! )(row: " << LastRow + 1 << ",
column: " << LastClmn + 1 << ")";
                                }
                                FuncCharTMP.ErrorLog = parserErrInfo.str();
                                return false;
                        }
                }
                else
                {
                        //err not Constant
                        parserErrInfo << "Parser: Error( Constant expected, but '";

                        if (TableCodePosition[tokenCodePosition][0] != NULL) {
                                for (int j = 0; TokenTable[tokenCodePosition][j]; j++)
                                {
                                        parserErrInfo << TokenTable[tokenCodePosition][j];
                                }

                                parserErrInfo << "' found!)(row: " <<
TableCodePosition[tokenCodePosition][0] << ", column : " <<
TableCodePosition[tokenCodePosition][1] << ")";
                        }
                        else {
                                parserErrInfo << "Nothing' found! )(row: " << LastRow + 1 << ", column: "
<< LastClmn + 1 << ")";
                        }
                        FuncCharTMP.ErrorLog = parserErrInfo.str();
                        return false;
                }
        }
        else
        {
                //err expected '\'
                parserErrInfo << "Parser: Error( Symbol '\' expected, but '";

                if (TableCodePosition[tokenCodePosition][0] != NULL) {
                        for (int j = 0; TokenTable[tokenCodePosition][j]; j++)
                        {
```

```
                        parserErrInfo << TokenTable[tokenCodePosition][j];
                    }

                    parserErrInfo << "' found!)(row: " << TableCodePosition[tokenCodePosition][0] <<
", column : " << TableCodePosition[tokenCodePosition][1] << ")";
            }
            else {
                    parserErrInfo << "Nothing' found! )(row: " << LastRow + 1<< ", column: " <<
LastClmn + 1<< ")";
            }
            FuncCharTMP.ErrorLog = parserErrInfo.str();

            return false;
    }
}

bool Parser::Empty() {
    if (TableCodePosition[tokenCodePosition][2] == 402)
            return true;
    else
            return false;
}
```

Приклади роботи програми

Test01

input.sig – Блокнот

Файл Правка Формат Вид Справка

```
PROGRAM PR1;
DEFFUNC F1=20\0,16;
BEGIN

END.
```

generated.txt – Блокнот

Файл Правка Формат Вид Справка

```
DATA_PR1        SEGMENT byte
  F1            17 DUP (?)
DATA_PR1        ENDS

CODE_PR1        SEGMENT
        MOV     ECX,0
        MOV     EAX,16
L1?:
        MOV     F1[ECX * 2],20
        ADD     ECX,1
        CMP     ECX, 16
        JLE     L1?
BEGIN:
CODE_PR1        ENDS
                END     BEGIN
```

Test02

input.sig – Блокнот

Файл Правка Формат Вид Сп

```
PROGRAM PR1;
DEFFUNC F1=20\0,16;
F2=34\3,456;
DFT=58\10,96;
BEGIN

END.
```

generated.txt – Блокнот

Файл Правка Формат Вид Справка

```
DATA_PR1        SEGMENT byte
  F1            17 DUP (?)
  F2            457 DUP (?)
  DFT           97 DUP (?)
DATA_PR1        ENDS

CODE_PR1        SEGMENT
        MOV     ECX,0
        MOV     EAX,16
L1?:
        MOV     F1[ECX * 2],20
        ADD     ECX,1
        CMP     ECX, 16
        JLE     L1?
        MOV     ECX,3
        MOV     EAX,456
L2?:
        MOV     F2[ECX * 2],34
        ADD     ECX,1
        CMP     ECX, 456
        JLE     L2?
        MOV     ECX,10
        MOV     EAX,96
L3?:
        MOV     DFT[ECX * 2],58
        ADD     ECX,1
        CMP     ECX, 96
        JLE     L3?
BEGIN:
CODE_PR1        ENDS
                END     BEGIN
```

Test03

input.sig – Блокнот

Файл Правка Формат Вид

```
PRO$GRAM PR1;
DEFFUNC F1=20\0,16;
BEGIN

END.
```

generated.txt – Блокнот

Файл Правка Формат Вид Справка

```
Lexer: Error( Undefined symbol was found:^$^)(row: 1, column: 4)
Parser: Error( Reserved word 'PROGRAM' expected, but 'PRO' found!)(row: 1, column : 1)
```

Test04

input.sig – Блокнот

Файл Правка Формат Вид Справ

```
PROGRAM PR1;
DEFFUNC F1=10\30,16;
BEGIN
(*egeg%$#*)
END.
```

generated.txt – Блокнот

Файл Правка Формат Вид Справка

```
DATA_PR1          SEGMENT byte
  F1              17 DUP (?)
DATA_PR1          ENDS

CODE_PR1          SEGMENT
BEGIN:
CODE_PR1          ENDS
                  END     BEGIN
Code Generator: Error( First index '30' at [R:2][C:15] cant't be greater then second index)
```

Test05

input.sig – Блокнот

Файл Правка Формат Вид

```
PROGRAM PR1;
DEFFUNC F1=20\12,24;
BEGIN ^

END
```

generated.txt – Блокнот

Файл Правка Формат Вид Справка

```
Lexer: Error( Undefined symbol was found:^^^)(row: 3, column: 7)
Parser: Error( Symbol '.' expected, but 'Nothing' found! )(row: 5, column: 5)
```

Test06

input.sig – Блокнот

Файл Правка Формат Вид Спр

```
PROGRAM 123;(*
DEFFUNC F1=20\0,16;
BEGIN

END.
```

generated.txt – Блокнот

Файл Правка Формат Вид Справка

```
Lexer: Error( You must close comment! )(row: 1, column: 13)
Parser: Error( Program identifier expected, but '123' found!)(row: 1, column : 9)
```

Test07

input.sig – Блокнот

Файл Правка Формат Вид Сп

```
PROGRAM PR1;
DEFFUNC F1=20\EQ,16;
F1=14\10,20;
BEGIN

END.
```

generated.txt – Блокнот

Файл Правка Формат Вид Справка

```
DATA_PR1          SEGMENT byte
DATA_PR1          ENDS

CODE_PR1          SEGMENT
CODE_PR1          ENDS

Parser: Error( Constant expected, but 'EQ' found!)(row: 2, column : 15)
```

Test08

input.sig – Блокнот

Файл Правка Формат Вид Справка

```
PROGRAM PR1;
DEFFUNC TEST=20\0,16; TEST=86\54,54; $=34\56,10;
BEGIN

END.
```

generated.txt – Блокнот

Файл Правка Формат Вид Справка

```
DATA_PR1        SEGMENT byte
  TEST          17 DUP (?)
DATA_PR1        ENDS

CODE_PR1        SEGMENT
        MOV     ECX,0
        MOV     EAX,16
L1?:
        MOV     TEST[ECX * 2],20
        ADD     ECX,1
        CMP     ECX, 16
        JLE     L1?
CODE_PR1        ENDS

Lexer: Error( Undefined symbol was found:^$^)(row: 2, column: 38)
Parser: Error( Function identifier expected, but '=' found!)(row: 2, column : 39)
Code Generator: Error( Name of math function 'TEST' at [R:2][C:23] must be unique!)
```

Test09

input.sig – Блокнот

Файл Правка Формат Ви

```
PROGRAM PR1;
DEFFUNC
BEGIN

END.
```

generated.txt – Блокнот

Файл Правка Формат Вид Справка

```
DATA_PR1        SEGMENT byte
DATA_PR1        ENDS

CODE_PR1        SEGMENT
BEGIN:
CODE_PR1        ENDS
                END     BEGIN
```

Test10

input.sig – Блокнот

Файл Правка Формат

```
PROGRAM PR1*;
F1=10\1,19;
BEGIN

END.
```

generated.txt – Блокнот

Файл Правка Формат Вид Справка

```
DATA_PR1          SEGMENT byte
DATA_PR1          ENDS

CODE_PR1          SEGMENT
CODE_PR1          ENDS

Lexer: Error( Undefined symbol was found:^*^)(row: 1, column: 12)
Parser: Error( Reserved word 'DEFFUNC' expected (or empty), but 'F1' found!)(row: 2, column : 1)
```

Test11

input.sig – Блокнот

Файл Правка Форма

```
PROGRAM PR1;
BEGIN

END.
```

generated.txt – Блокнот

Файл Правка Формат Вид Справка

```
DATA_PR1          SEGMENT byte
DATA_PR1          ENDS

CODE_PR1          SEGMENT
BEGIN:
CODE_PR1          ENDS
                  END     BEGIN
```

Test12

input.sig – Блокнот

Файл Правка Формат Вид

```
PROGRAM PR1;
DEFFUNC G2=4\25,0;
BEGIN

END.
```

generated.txt – Блокнот

Файл Правка Формат Вид Справка

```
DATA_PR1          SEGMENT byte
DATA_PR1          ENDS

CODE_PR1          SEGMENT
BEGIN:
CODE_PR1          ENDS
                  END     BEGIN
Code Generator: Error( Value of second index '0' at [R:2][C:17] can't be less then 1!)
```