

Міністерство освіти і науки, молоді та спорту України

Національний технічний університет України

«Київський Політехнічний Інститут імені Ігоря Сікорського»

Факультет прикладної математики

Кафедра спеціалізованих комп’ютерних систем та системного програмування

**Розрахунково-графічна робота**

**З дисципліни «Інженерія програмного забезпечення. Основи проектування трансляторів»**

**На тему: «Розробка синтаксичного аналізатора»**

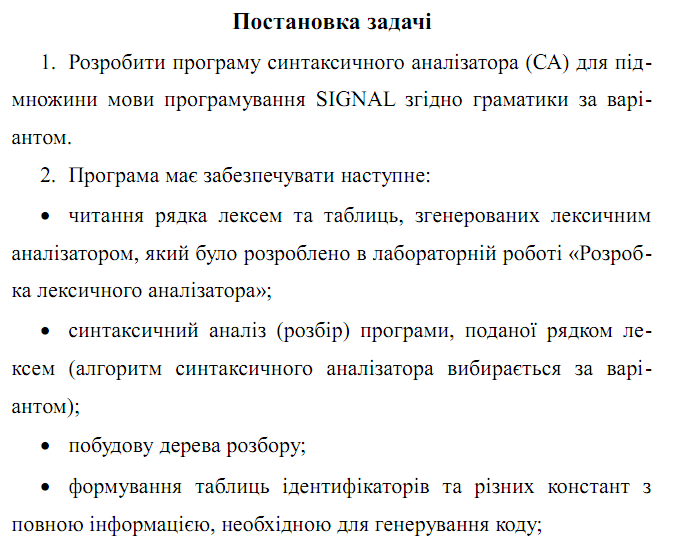
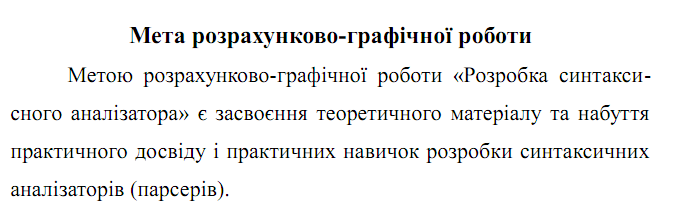
**Виконав:**

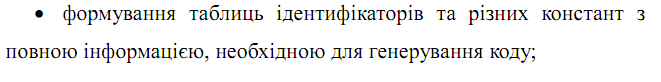
**Студент групи**

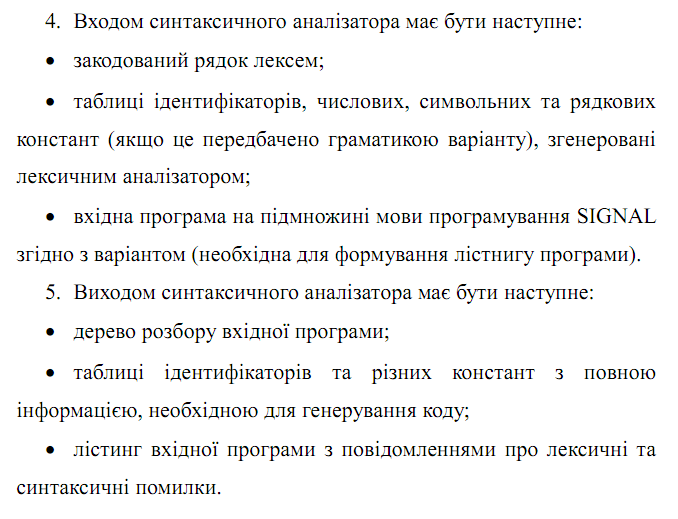
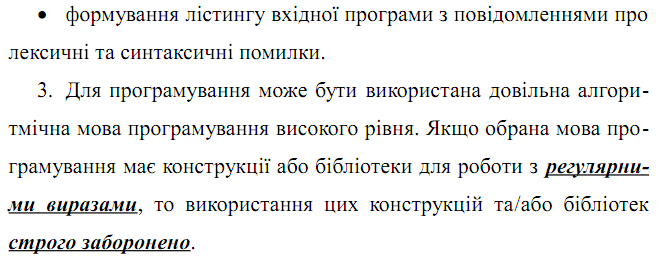
**КВ-52**

**Варіант: 14**

**Київ 2018**







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

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

<block>.

3. <block> --> <declarations> begin <statements-

List> end

4. <declarations> --> <label-declarations>

5. <label-declarations> --> label <unsigned-inte-

Ger> <labels-list>; |

<empty>

6. <labels-list> --> , <unsigned-integer> <labels-

List> |

<empty>

7. <statements-list> --> <statement> <statements-

List> |

<empty>

8. <statement> --> <unsigned-integer> : <statement>|

Goto <unsigned-integer> ; |

<condition-statement> endif ;|

;

9. <condition-statement> --> <incomplete-condition-

Statement><alternative-part>

10. <incomplete-condition-statement> --> if

<conditional-expression> then

<statements-list>

11. <conditional-expression> --> <variable-identi-

Fier> = <unsigned-integer>

12. <alternative-part> --> else <statements-list> |

<empty>

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

14. <procedure-identifier> --> <identifier>

15. <identifier> --> <letter><string>

16. <string> --> <letter><string> |

<digit><string> |

<empty>

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

18. <digits-string> --> <digit><digits-string> |

<empty>

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

20. <letter> --> a | b | c | d | ... | z

Код програми

#include "SyntaxAnaliz.h"

SyntaxAnaliz::SyntaxAnaliz(string FileName)

{

ErrorStr = "";

ErrorCounter = 0;

Pos = 0;

LeksAnaliz LexAn;

LexAn.Analizator(FileName);

LexAn.MakeListing(FileName);

IdentMin = LexAn.MinIdent;

ConstMin = LexAn.MinConst;

ConstMax = LexAn.ConstMax;

Lexems = LexAn.Lexems;

IdentTable = LexAn.TableC;

ConstTable = LexAn.Table;

START();

if (ErrorCounter == 0) {

cout << "syntax analysis completed successfully" << endl;

}

}

bool SyntaxAnaliz::START()

{

Tree = InitTree();

return PROG(Tree);

}

bool SyntaxAnaliz::PROG(Node\* NodePtr)

{

Node\* N = NodePtr;

if (Pos >= Lexems.size()) {

Errors(1, Pos);

return false;

}

if (Lexems[Pos].Code == 401) {

N->Down = AddNode(N, Lexems[Pos].Code, Lexems[Pos].Name, Lexems[Pos].Line, Lexems[Pos].Column, "", true);

N = N->Down;

Pos++;

if (Pos >= Lexems.size()) {

Errors(1, Pos);

return false;

}

N->Next = AddNode(N, -1, "", -1, -1, "<procedure-identifier>", false);

N = N->Next;

if (PROCIDNT(N)){

Pos++;

}

else{

return false;

}

if (Lexems[Pos].Code == 59){

N->Next = AddNode(N, Lexems[Pos].Code, Lexems[Pos].Name, Lexems[Pos].Line, Lexems[Pos].Column, "", true);

Pos++;

N = N->Next;

N->Next = AddNode(N, -1, "", -1, -1, "<block>", false);

N = N->Next;

}

else{

Errors(2, Pos);

return false;

}

if (Pos >= Lexems.size()) {

Errors(1, Pos);

return false;

}

if (BLOCK(N)){

Pos++;

}

else{

return false;

}

if (Pos >= Lexems.size()) {

Errors(1, Pos);

return false;

}

if ((Pos < Lexems.size()) && (Lexems[Pos].Code == 46)){

N->Next = AddNode(N, Lexems[Pos].Code, Lexems[Pos].Name, Lexems[Pos].Line, Lexems[Pos].Column, "", true);

return true;

}

else

{

Errors(3, Pos);

return false;

}

}

else{

Errors(5, Pos);

return false;

}

}

bool SyntaxAnaliz::BLOCK(Node\* NodePtr)

{

if (Pos >= Lexems.size()) {

Errors(1, Pos);

return false;

}

Node\* N = NodePtr;

N->Down = AddNode(N, -1, "", -1, -1, "<declarations>", false);

N = N->Down;

if (DECLARATION(N)){

}

else

{

return false;

}

if (Pos >= Lexems.size()) {

Errors(1, Pos);

return false;

}

if (Lexems[Pos].Code == 405)

{

N->Next = AddNode(N, Lexems[Pos].Code, Lexems[Pos].Name, Lexems[Pos].Line, Lexems[Pos].Column, "", true);

N = N->Next;

Pos++;

N->Next = AddNode(N, -1, "", -1, -1, "<statements-list>", false);

N = N->Next;

}

else

{

Errors(8, Pos);

return false;

}

if (Pos >= Lexems.size()) {

Errors(1, Pos);

return false;

}

if (STATMLIST(N)) {

Pos++;

}

else return false;

if (Lexems[Pos].Code == 411)

{

N->Next = AddNode(N, Lexems[Pos].Code, Lexems[Pos].Name, Lexems[Pos].Line, Lexems[Pos].Column, "", true);

return true;

}

else

{

//Errors(9, Pos);

return false;

}

}

bool SyntaxAnaliz::PROCIDNT(Node\* NodePtr) {

if (Pos >= Lexems.size()) {

Errors(1, Pos);

return false;

}

//int HelpPos = Pos;

if (Lexems[Pos].Code >= IdentMin)

{

Node\* N = NodePtr;

N->Down = AddNode(N, Lexems[Pos].Code, Lexems[Pos].Name, Lexems[Pos].Line, Lexems[Pos].Column, "", true);

return true;

}

else

{

Errors(6, Pos);

return false;

}

}

bool SyntaxAnaliz::DECLARATION(Node\* NodePtr) {

if (Pos >= Lexems.size()) {

Errors(1, Pos);

return false;

}

Node\* N = NodePtr;

N->Down = AddNode(N, -1, "", -1, -1, "<label-declarations>", false);

N = N->Down;

if (LABELDECL(N)) {

return true;

}

else return false;

}

bool SyntaxAnaliz::STATMLIST(Node\* NodePtr) {

if (Pos >= Lexems.size()) {

Errors(1, Pos);

return false;

}

Node\* N = NodePtr;

N->Down = AddNode(N, -1, "", -1, -1, "<statement>", false);

if (STATM(N->Down)) {

N = N->Down;

N->Next = AddNode(N, -1, "", -1, -1, "<statements-list>", false);

N = N->Next;

Pos++;

}

else {

N->Down = AddNode(N, -1, "", -1, -1, "<empty>", false);

Pos--;

return true;

}

if (STATMLIST(N)) {

}

return true;

}

bool SyntaxAnaliz::STATM(Node\* NodePtr) {

if (Pos >= Lexems.size()) {

Errors(1, Pos);

return false;

}

Node\* N = NodePtr;

N->Down = AddNode(N, -1, "", -1, -1, "<unsigned-integer>", false);

if (CONST(N->Down)) {

N = N->Down;

Pos++;

if (Lexems[Pos].Code == ':') {

N->Next = AddNode(N, Lexems[Pos].Code, Lexems[Pos].Name, Lexems[Pos].Line, Lexems[Pos].Column, "", true);

N = N->Next;

Pos++;

N->Next = AddNode(N, -1, "", -1, -1, "<statement>", false);

N = N->Next;

if (STATM(N)) {

return true;

}

else {

return false;

}

}

else {

return false;

}

}

else {

if (Lexems[Pos].Code == 415) {

N->Down = AddNode(N, Lexems[Pos].Code, Lexems[Pos].Name, Lexems[Pos].Line, Lexems[Pos].Column, "", true);

N = N->Down;

Pos++;

N->Next = AddNode(N, -1, "", -1, -1, "<unsigned-integer>", false);

N = N->Next;

if (CONST(N)) {

Pos++;

if (Lexems[Pos].Code == ';') {

N->Next = AddNode(N, Lexems[Pos].Code, Lexems[Pos].Name, Lexems[Pos].Line, Lexems[Pos].Column, "", true);

return true;

}

else {

Errors(11,Pos);

return false;

}

}

else {

return false;

}

}

else {

if (Lexems[Pos].Code == ';') {

N->Down = AddNode(N, Lexems[Pos].Code, Lexems[Pos].Name, Lexems[Pos].Line, Lexems[Pos].Column, "", true);

return true;

}

else {

N->Down = AddNode(N, -1, "", -1, -1, "<condition-statement>", false);

N = N->Down;

if (CONDSTATM(N)) {

Pos++;

if (Lexems[Pos].Code == 421) {

N->Next = AddNode(N, Lexems[Pos].Code, Lexems[Pos].Name, Lexems[Pos].Line, Lexems[Pos].Column, "", true);

N = N->Next;

Pos++;

if (Lexems[Pos].Code == ';') {

N->Next = AddNode(N, Lexems[Pos].Code, Lexems[Pos].Name, Lexems[Pos].Line, Lexems[Pos].Column, "", true);

return true;

}

else {

return false;

}

}

else {

return false;

}

}

else {

return false;

}

}

}

}

}

bool SyntaxAnaliz::CONDSTATM(Node\* NodePtr) {

if (Pos >= Lexems.size()) {

Errors(1, Pos);

return false;

}

Node\* N = NodePtr;

N->Down = AddNode(N, -1, "", -1, -1, "<incomplete-condition-statement>", false);

N = N->Down;

if (INCOMCONDSTATM(N)) {

Pos++;

}

else {

return false;

}

N->Next = AddNode(N, -1, "", -1, -1, "<alternative-part>", false);

N = N->Next;

if (ALTPART(N)) {

return true;

}

else return false;

}

bool SyntaxAnaliz::INCOMCONDSTATM(Node\* NodePtr) {

if (Pos >= Lexems.size()) {

Errors(1, Pos);

return false;

}

Node\* N = NodePtr;

if (Lexems[Pos].Code == 441)

{

N->Down = AddNode(N, Lexems[Pos].Code, Lexems[Pos].Name, Lexems[Pos].Line, Lexems[Pos].Column, "", true);

N = N->Down;

Pos++;

N->Next = AddNode(N, -1, "", -1, -1, "<conditional-expression>", false);

N = N->Next;

if (CONDEXPR(N)) {

Pos++;

if (Lexems[Pos].Code == 455) {

N->Next = AddNode(N, Lexems[Pos].Code, Lexems[Pos].Name, Lexems[Pos].Line, Lexems[Pos].Column, "", true);

N = N->Next;

Pos++;

N->Next = AddNode(N, -1, "", -1, -1, "<statement-list>", false);

N = N->Next;

if (STATMLIST(N)) {

return true;

}

else {

return false;

}

}

else {

Errors(12,Pos);

return false;

}

}

else {

return false;

}

}

else {

return false;

}

}

bool SyntaxAnaliz::ALTPART(Node\* NodePtr) {

if (Pos >= Lexems.size()) {

Errors(1, Pos);

return false;

}

Node\* N = NodePtr;

if (Lexems[Pos].Code == 425) {

N->Down = AddNode(N, Lexems[Pos].Code, Lexems[Pos].Name, Lexems[Pos].Line, Lexems[Pos].Column, "", true);

N = N->Down;

Pos++;

N->Next = AddNode(N, -1, "", -1, -1, "<statement-list>", false);

N = N->Next;

}

else {

N->Down = AddNode(N, -1, "", -1, -1, "<empty>", false);

Pos--;

return true;

}

if (STATMLIST(N)) {

return true;

}

else {

return false;

}

}

bool SyntaxAnaliz::CONDEXPR(Node\* NodePtr) {

if (Pos >= Lexems.size()) {

Errors(1, Pos);

return false;

}

Node\* N = NodePtr;

N->Down = AddNode(N, -1, "", -1, -1, "<variable-identifier>", false);

N = N->Down;

if (PROCIDNT(N)) {

Pos++;

if (Lexems[Pos].Code == '=') {

N->Next = AddNode(N, Lexems[Pos].Code, Lexems[Pos].Name, Lexems[Pos].Line, Lexems[Pos].Column, "", true);

N = N->Next;

Pos++;

N->Next = AddNode(N, -1, "", -1, -1, "<unsigned-integer>", false);

N = N->Next;

if (CONST(N)) {

return true;

}

else {

return false;

}

}

else {

return false;

}

}

else {

return false;

}

}

bool SyntaxAnaliz::LABELDECL(Node\* NodePtr) {

if (Pos >= Lexems.size()) {

Errors(1, Pos);

return false;

}

Node\* N = NodePtr;

if (Lexems[Pos].Code == 407) {

N->Down = AddNode(N, Lexems[Pos].Code, Lexems[Pos].Name, Lexems[Pos].Line, Lexems[Pos].Column, "", true);

N = N->Down;

N->Next = AddNode(N, -1, "", -1, -1, "<unsigned-integer>", false);

N = N->Next;

Pos++;

}

else {

N->Down = AddNode(N, -1, "", -1, -1, "<empty>", false);

return true;

}

if (CONST(N)) {

N->Next = AddNode(N, -1, "", -1, -1, "<label-list>", false);

N = N->Next;

Pos++;

}

else return false;

if (LABELLIST(N)) {

}

else return false;

if ((Pos < Lexems.size()) && (Lexems[Pos].Code == 59)) {

N->Next = AddNode(N, Lexems[Pos].Code, Lexems[Pos].Name, Lexems[Pos].Line, Lexems[Pos].Column, "", true);

Pos++;

return true;

}

else {

Errors(2, Pos);

return false;

}

}

bool SyntaxAnaliz::CONST(Node\* NodePtr) {

if (Pos >= Lexems.size()) {

Errors(1, Pos);

return false;

}

if ((Lexems[Pos].Code >= ConstMin) && (Lexems[Pos].Code <= ConstMax))

{

Node\* N = NodePtr;

N->Down = AddNode(N, Lexems[Pos].Code, Lexems[Pos].Name, Lexems[Pos].Line, Lexems[Pos].Column, "", true);

N = N->Down;

return true;

}

else

{

Errors(10, Pos);

return false;

}

}

bool SyntaxAnaliz::LABELLIST(Node\* NodePtr) {

if (Pos >= Lexems.size()) {

Errors(1, Pos);

return false;

}

Node\* N = NodePtr;

if (Lexems[Pos].Code == 44) {

N->Down = AddNode(N, Lexems[Pos].Code, Lexems[Pos].Name, Lexems[Pos].Line, Lexems[Pos].Column, "", true);

N = N->Down;

N->Next = AddNode(N, -1, "", -1, -1,"<unsigned-integer>", false);

N = N->Next;

Pos++;

}

else {

N->Down = AddNode(N, -1, "", -1, -1, "<empty>", false);

return true;

};

if (CONST(N)) {

N->Next = AddNode(N, -1, "", -1, -1, "<label-list>", false);

N = N->Next;

Pos++;

}

else return false;

if (LABELLIST(N)) {

}

else return false;

}

void SyntaxAnaliz::Errors(int p, int pos) {

switch (p) {

case 1:

cout << "Syntactic error: Unexpected end of the file!" << endl;

ErrorStr += "Syntactic error: Unexpected end of the file!";

ErrorCounter++;

break;

case 2:

cout << "Syntactic error on Line " << Lexems[pos].Line << " Column " << Lexems[pos].Column << ": \";\" is passed" << endl;

ErrorStr += "Syntactic error on Line ";

ErrorStr += to\_string(Lexems[pos].Line);

ErrorStr += " Column ";

ErrorStr += to\_string(Lexems[pos].Column);

ErrorStr += ": \";\" is passed";

ErrorCounter++;

break;

case 3:

cout << "Syntactic error on Line " << Lexems[pos].Line << " Column " << Lexems[pos].Column << ": \".\" is passed" << endl;

ErrorStr += "Syntactic error on Line ";

ErrorStr += to\_string(Lexems[pos].Line);

ErrorStr += " Column ";

ErrorStr += to\_string(Lexems[pos].Column);

ErrorStr += ": \".\" is passed";

ErrorCounter++;

break;

//case 4:

// cout << "Syntactic error: \".\" is passed after the \"END\"" << endl;

// break;

case 5:

cout << "Syntactic error on Line " << Lexems[pos].Line << " Column " << Lexems[pos].Column << ": Keyword \"PROGRAM\" is passed" << endl;

ErrorStr += "Syntactic error on Line ";

ErrorStr += to\_string(Lexems[pos].Line);

ErrorStr += " Column ";

ErrorStr += to\_string(Lexems[pos].Column);

ErrorStr += ": Keyword \"PROGRAM\" is passed";

ErrorCounter++;

break;

case 6:

cout << "Syntactic error on Line " << Lexems[pos].Line << " Column " << Lexems[pos].Column << ": Must be procedure identifier" << endl;

ErrorStr += "Syntactic error on Line ";

ErrorStr += to\_string(Lexems[pos].Line);

ErrorStr += " Column ";

ErrorStr += to\_string(Lexems[pos].Column);

ErrorStr += ": Must be procedure identifier";

ErrorCounter++;

ErrorCounter++;

break;

case 7:

cout << "Syntactic error on Line " << Lexems[pos].Line << " Column " << Lexems[pos].Column << ": \")\" is passed" << endl;

ErrorStr += "Syntactic error on Line ";

ErrorStr += to\_string(Lexems[pos].Line);

ErrorStr += " Column ";

ErrorStr += to\_string(Lexems[pos].Column);

ErrorStr += ": \")\" is passed";

ErrorCounter++;

break;

case 8:

cout << "Syntactic error on Line " << Lexems[pos].Line << " Column " << Lexems[pos].Column << ": Keyword \"BEGIN\" is passed" << endl;

ErrorStr += "Syntactic error on Line ";

ErrorStr += to\_string(Lexems[pos].Line);

ErrorStr += " Column ";

ErrorStr += to\_string(Lexems[pos].Column);

ErrorStr += ": Keyword \"BEGIN\" is passed";

ErrorCounter++;

break;

case 9:

cout << "Syntactic error on Line " << Lexems[pos].Line << " Column " << Lexems[pos].Column << ": Keyword \"END\" is passed" << endl;

ErrorStr += "Syntactic error on Line ";

ErrorStr += to\_string(Lexems[pos].Line);

ErrorStr += " Column ";

ErrorStr += to\_string(Lexems[pos].Column);

ErrorStr += ": Keyword \"END\" is passed";

ErrorCounter++;

break;

case 11:

cout << "Syntactic error on Line " << Lexems[pos].Line << " Column " << Lexems[pos].Column << ": \",\" is passed" << endl;

ErrorStr += "Syntactic error on Line ";

ErrorStr += to\_string(Lexems[pos].Line);

ErrorStr += " Column ";

ErrorStr += to\_string(Lexems[pos].Column);

ErrorStr += ": \",\" is passed";

ErrorCounter++;

break;

case 12:

cout << "Syntactic error on Line " << Lexems[pos].Line << " Column " << Lexems[pos].Column << ": \"THEN\" is passed" << endl;

ErrorStr += "Syntactic error on Line ";

ErrorStr += to\_string(Lexems[pos].Line);

ErrorStr += " Column ";

ErrorStr += to\_string(Lexems[pos].Column);

ErrorStr += ": \"THEN\" is passed";

ErrorCounter++;

break;

}

}

Node\* SyntaxAnaliz::InitTree()

{

Node\* Help = new Node;

Help->Code = -1;

Help->Column = -1;

Help->Down = NULL;

Help->IsTerminal = false;

Help->Line = -1;

Help->Name = "";

Help->Next = NULL;

Help->Func = "<program>";

return Help;

}

Node\* SyntaxAnaliz::AddNode(Node\* Nd, int code, string name, int line, int column, string Func, bool isterminal)

{

Node\* Help = new Node;

Help->Code = code;

Help->Column = column;

Help->Down = NULL;

Help->IsTerminal = isterminal;

Help->Line = line;

Help->Name = name;

Help->Next = NULL;

Help->Func = new char[255];

Help->Func = Func;

return Help;

}

Node\* SyntaxAnaliz::RemoveNode(Node\* N)

{

if (N != NULL) free(N);

return NULL;

}

void SyntaxAnaliz::WriteTree(Node\* Root, string sps)

{

while (Root != NULL)

{

if (Root->IsTerminal)

{

TreeStr += sps + "'" + Root->Name + "'" + "\n";

}

else

{

TreeStr += sps + Root->Func + "\n";

}

if (Root->Down != NULL) {

for (int i = 0; i < 6; i++) {

sps.push\_back(' ');

}

WriteTree(Root->Down, sps);

for (int i = 0; i < 6; i++) {

sps.pop\_back();

}

}

Root = Root->Next;

}

}

void SyntaxAnaliz::TreeListing(string FileName) {

TreeStr = "";

FileName = FileName + ".lst";

fstream f;

f.open(FileName, std::ios::app);

if (ErrorCounter > 0) {

f << ErrorStr << endl;

}

else {

WriteTree(Tree, "");

f << TreeStr << endl;

}

}