Министерство образования Республики Беларусь Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

КАФЕДРА ИНФОРМАТИКИ

Отчёт по лабораторной работе №3

По теме “Синтаксический анализатор”

Выполнила: студентка гр. 053501 Шурко Т.А.

Проверил: ассистент кафедры информатики Гриценко Н. Ю.

Минск 2023

**СОДЕРЖАНИЕ**

[1 Цель работы 3](#_Toc131174104)

[2 Теория 4](#_Toc131174104)

[3 Программа и комментарии](#_Toc131174105) 5

[4 Демонстрация работы 6](#_Toc131174106)

[5 Демонстрация нахождения ошибок 7](#_Toc131174106)

[Вывод 9](#_Toc131174107)

[Приложение А Таблица 10](#_Toc131174107)

[Приложение Б Текст программы 12](#_Toc131174107)

[Приложение В Текст программы 28](#_Toc131174107)

[Приложение Г Синтаксическое дерево 29](#_Toc131174107)

# **ЦЕЛЬ РАБОТЫ**

Освоение работы с существующими синтаксическими анализаторами.

Разработать свой собственный синтаксический анализатор, выбранного подмножества языка программирования. Построить синтаксическое дерево. Определить минимум 4 возможных синтаксических ошибки и показать их корректное выявление. Основной целью работы является написание сценариев, которые задают синтаксические правила для выбранного подмножества языка.

В качестве анализируемого подмножества языка программирования будет использован язык программирования Python. Для написания анализатора использован язык программирования C++.

# **ТЕОРИЯ**

Синтаксический анализ — это процесс сопоставления линейной последовательности лексем (слов, токенов) естественного или формального языка с его формальной грамматикой. Результатом обычно является дерево разбора (синтаксическое дерево). Обычно применяется совместно с лексическим анализом. Синтаксический анализатор — это программа или часть программы, выполняющая синтаксический анализ. Задача синтаксического анализатора – проверить правильность записи выражения и разбить его на лексемы. Лексемой называется неделимая часть выражения, состоящая, в общем случае, из нескольких символов. Результатом синтаксического анализа является синтаксическое строение предложения, представленное либо в виде дерева зависимостей, либо в виде дерева составляющих, либо в виде некоторого сочетания первого и второго способов представления.

Типы алгоритмов:

● Нисходящий парсер — продукции грамматики раскрываются, начиная со стартового символа, до получения требуемой последовательности токенов, им соответствуют LL-грамматики;

● Восходящий парсер — продукции восстанавливаются из правых частей, начиная с токенов и кончая стартовым символом, им соответствуют LR-грамматики.

1. **ПРОГРАММА И КОММЕНТАРИИ**

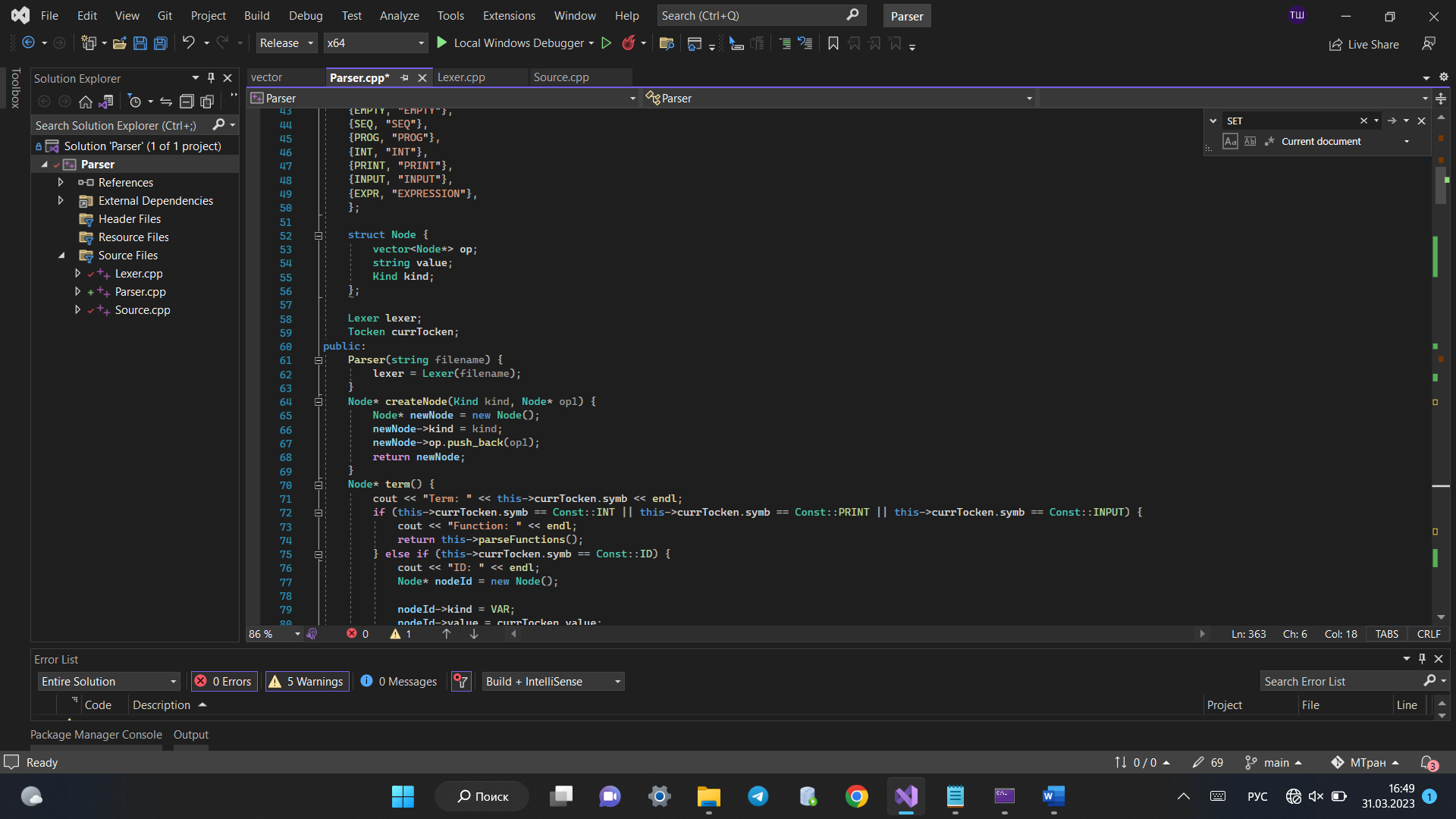
Программа написана на с++. Продукт представляет из себя нисходящий парсер, который рекурсивно разбивает задачу на узлы и на выходе формирует древовидную структуру. Разбиение происходит с помощью лексем, которые формируются в классе Lexer с помощью перечисления, значения которых отображены в таблице приложения А.

Рисунок 3.1 – Структура узла

В каждом узле (см. рисунок 3.1) будет хранится вектор следующих узлов, тип узла и значение, если таковое имеется. Также, в классе Parser, который создает синтаксическое дерево присутствует объект класса Lexer, для получения токенов и текущий токен. Полный код класса, который парсит лексемы в приложении Б.

## **ДЕМОНСТРАЦИЯ РАБОТЫ**

Продемонстрируем работоспособность программного продукта на примере задачи 1 (Приложение B). В консоли отображается древовидная структура (см. рисунок 4.1), построенная в результате синтаксического анализы задачи. Полное дерево отображено в приложении Г.

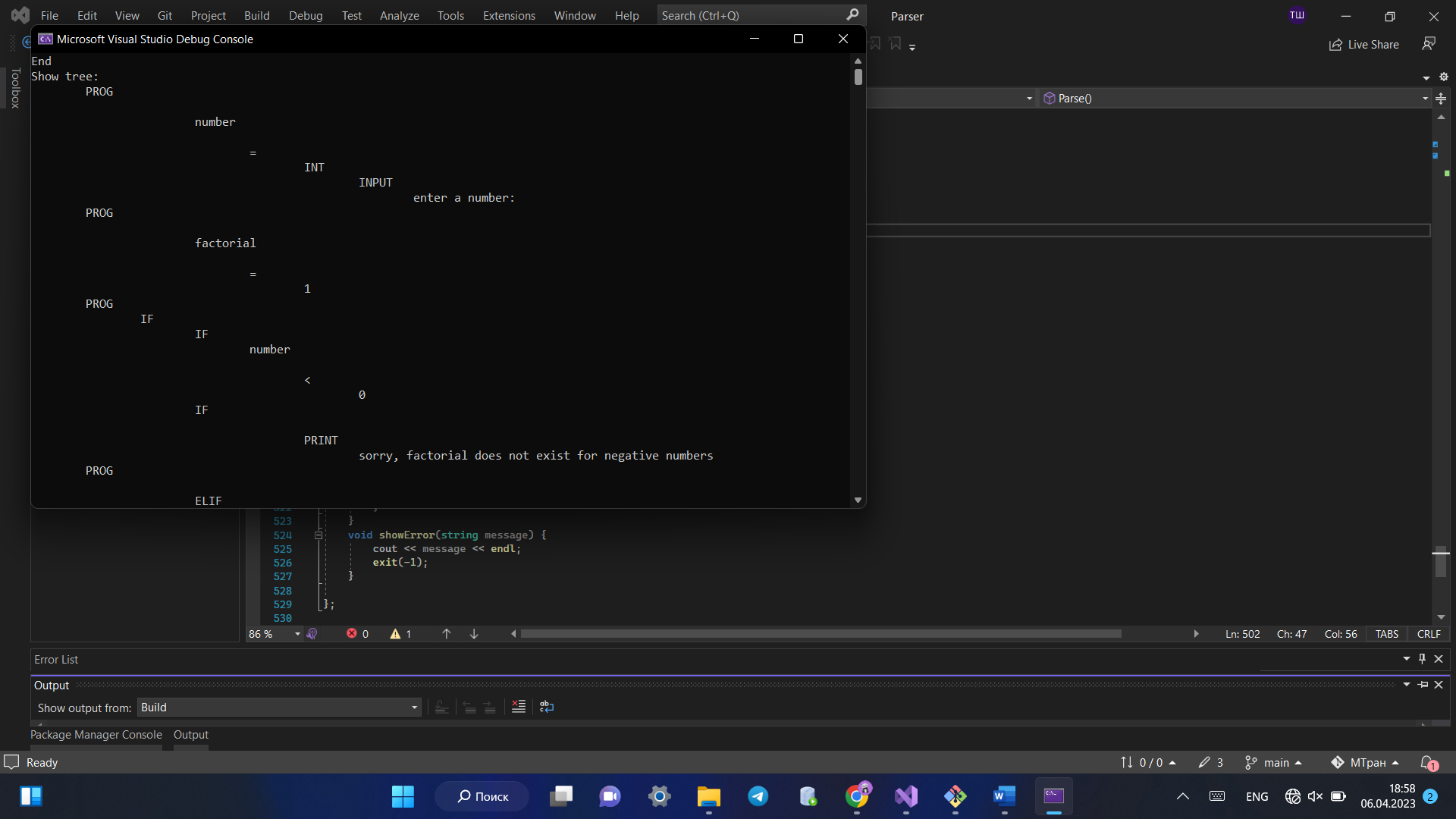


Рисунок 4.1 – Результат работы программного продукта

1. **ДЕМОНСТРАЦИЯ НАХОЖДЕНИЯ ОШИБОК**

На рисунке 5.1 ошибка в коде: отсутствующее двоеточие после If. Соответственно, синтаксическое дерево не может быть построено.

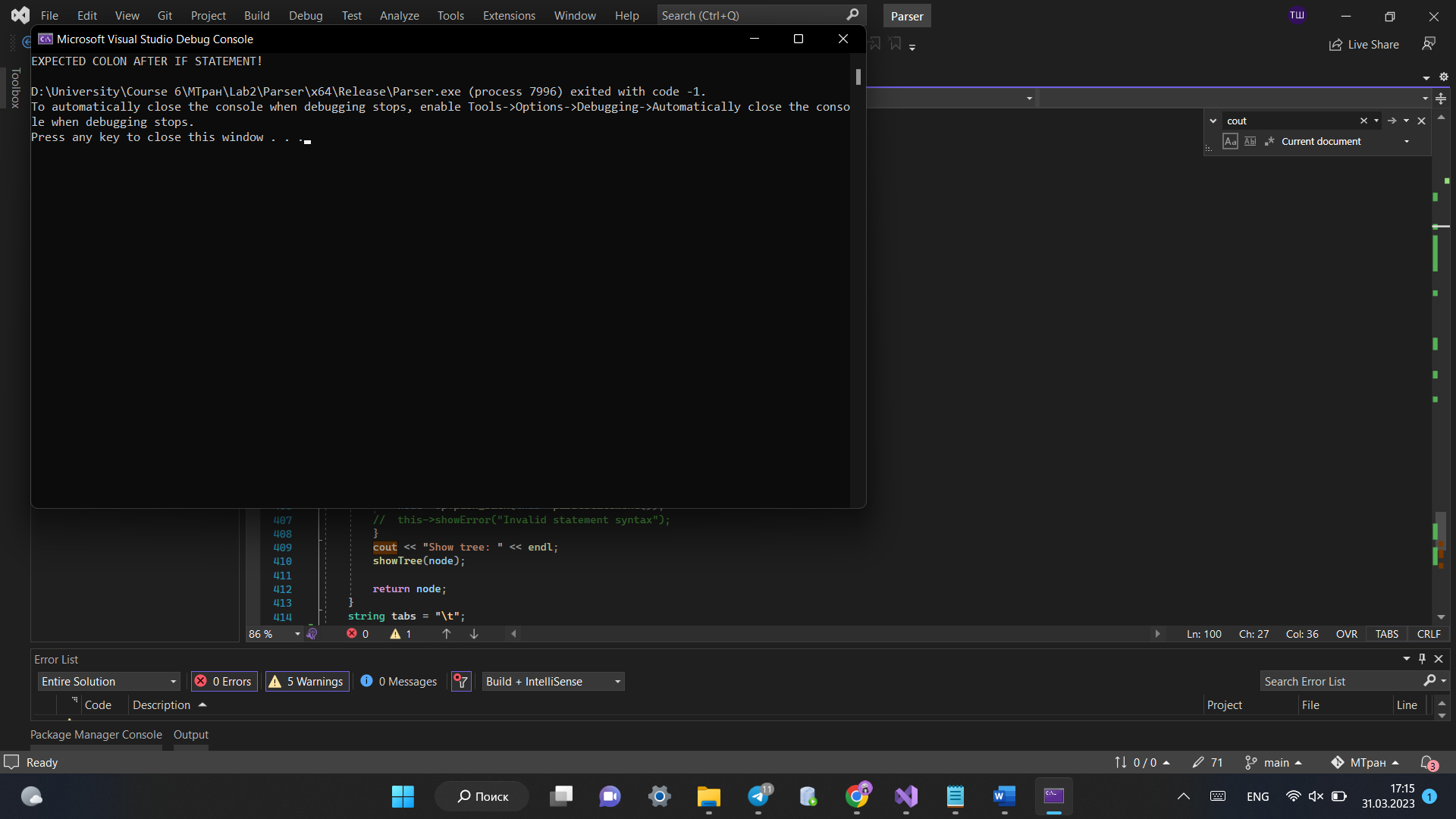


Рисунок 5.1 – Ошибка отсутствия ‘:’ после IF

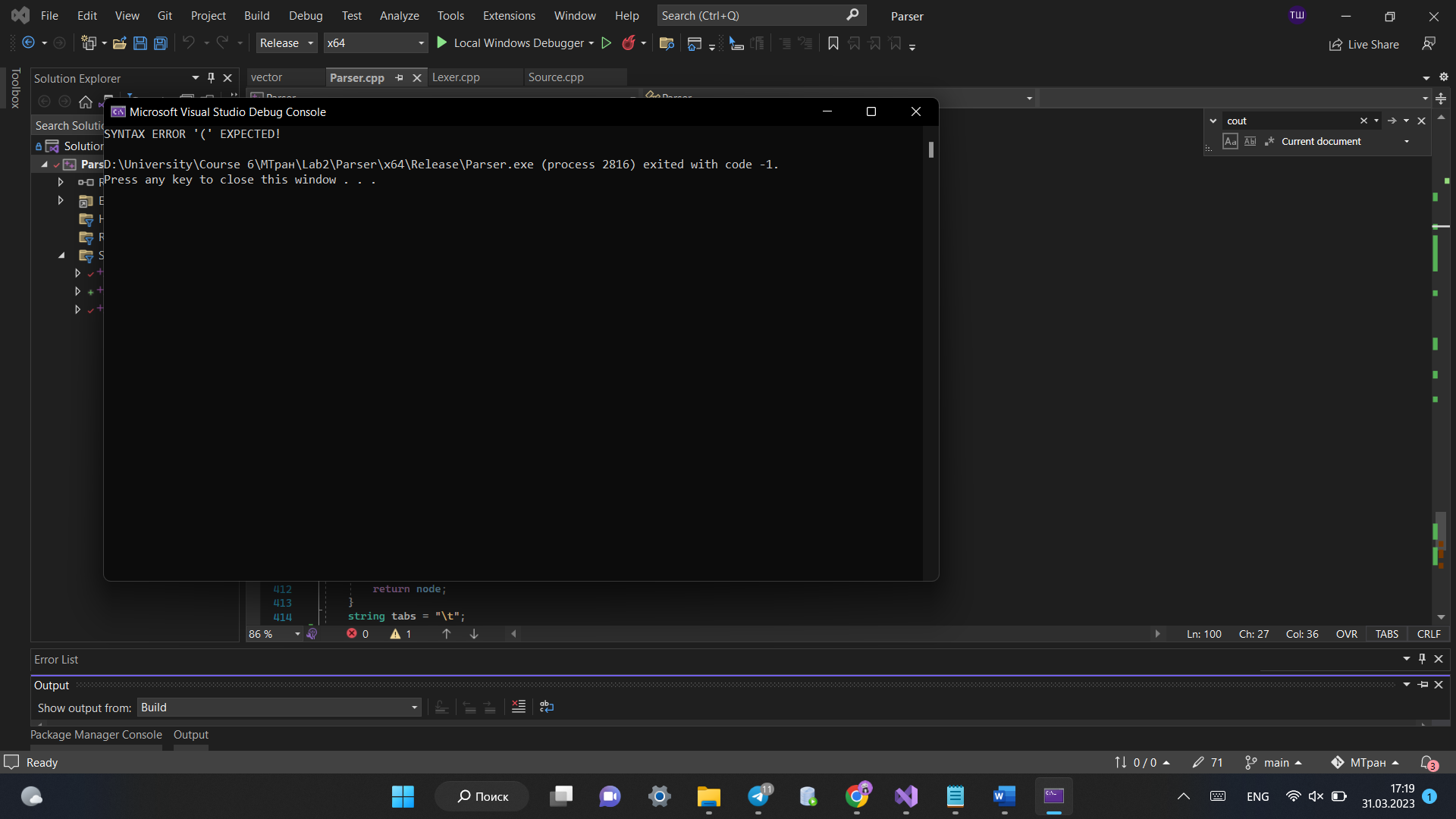


Рисунок 5.2 – Ошибка отсутствия ‘(’

Данная ошибка возникает при отсутствии выражения, которое ожидается в коде (см. рисунок 5.2). Например, If без выражения.

На рисунках 5.3-5.4 изображены ошибки в формировании выражения for.

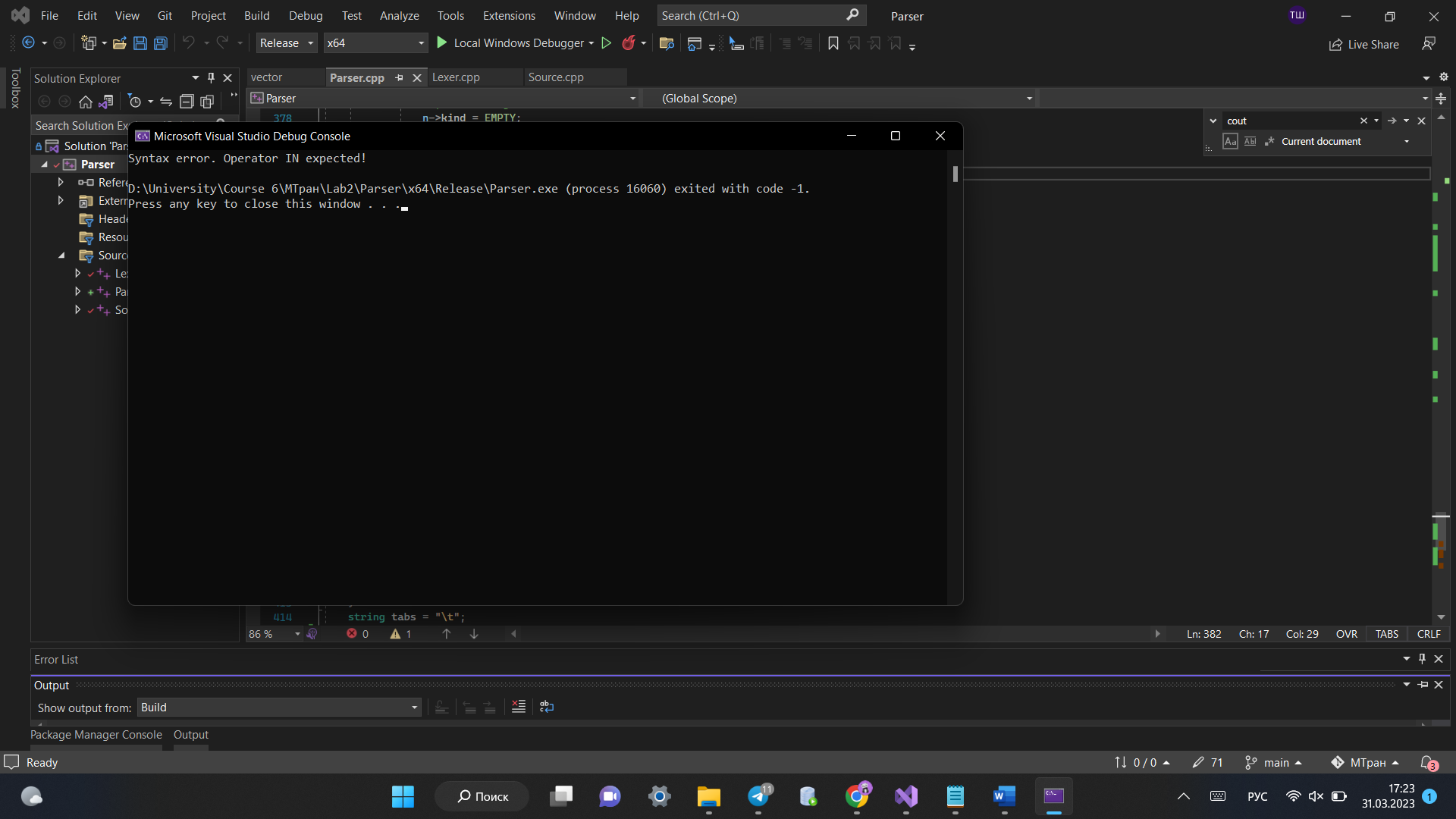


Рисунок 5.3 – Отсутствие In в выражении For

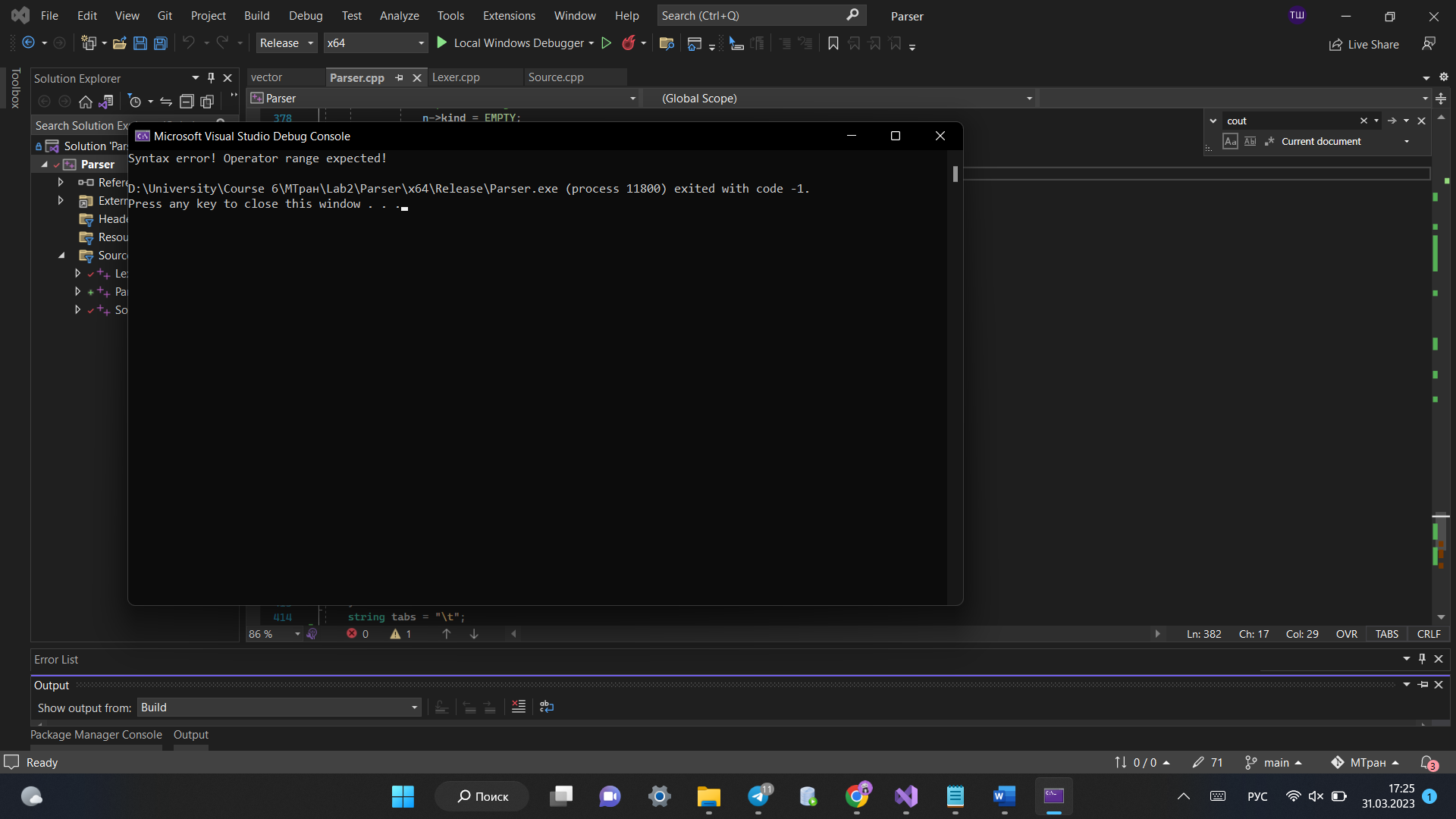


Рисунок 5.4 – Отсутствие Range в выражении For

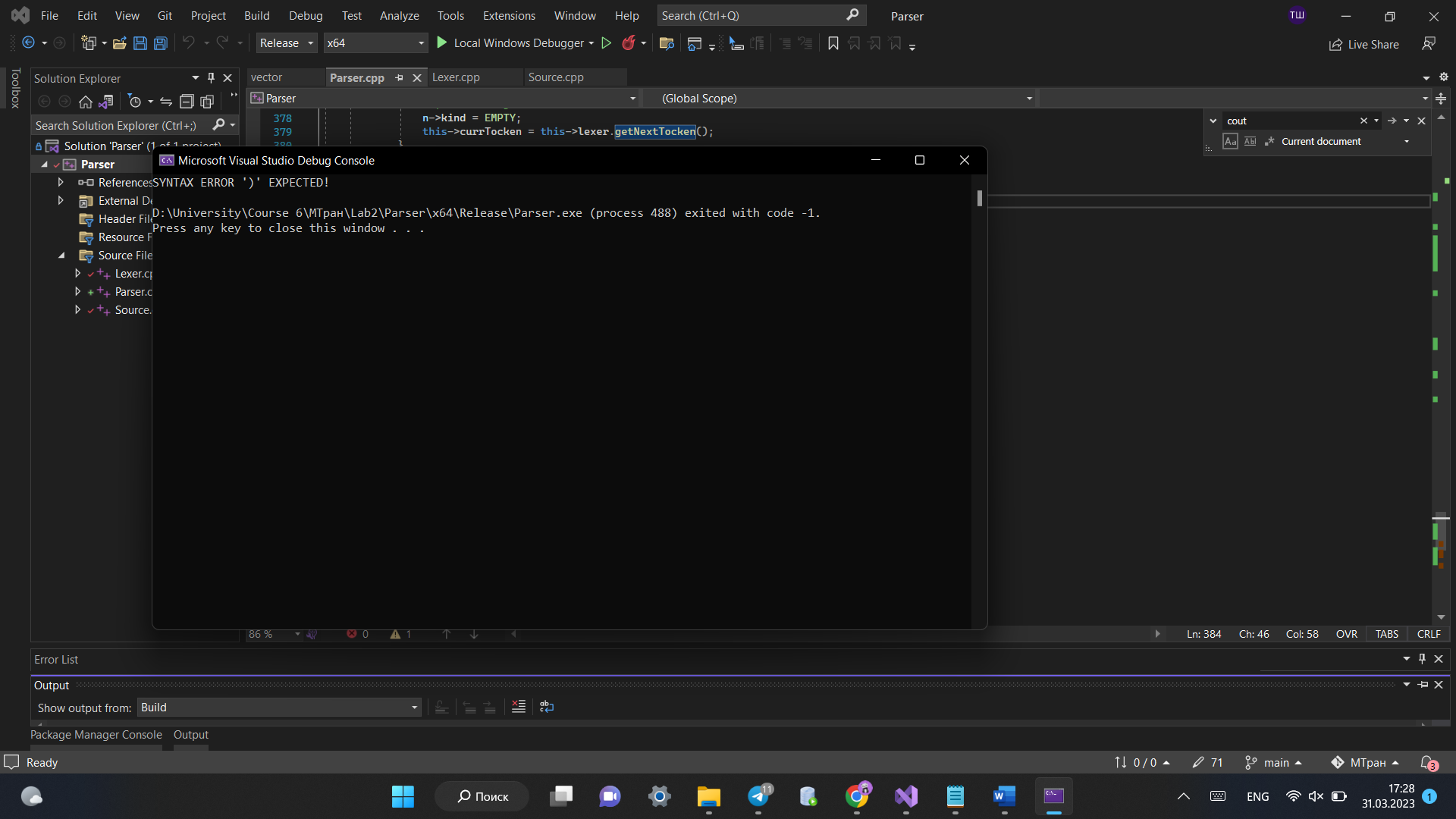
На рисунке 5.5 отсутствие закрывающейся скобки в выражении.

Рисунок 5.5 – Отсутствие закрывающейся скобки

## **ВЫВОД**

В результате выполнения лабораторной работы была изучена теория о синтаксических анализаторах, разработан собственный синтаксический анализатор, выбранного подмножества языка программирования, построено синтаксическое дерево. Определены минимум 4 возможных синтаксических ошибки и показано их корректное выявление.

**ПРИЛОЖЕНИЕ А ТАБЛИЦА**

|  |  |
| --- | --- |
| Int-value | name |
| 0 | NUM |
| 1 | DNUM |
| 2 | STR |
| 3 | ID |
| 4 | IF |
| 5 | ELSE |
| 6 | WHILE |
| 7 | DO |
| 8 | BREAK |
| 9 | CONTINUE |
| 10 | LPAR |
| 11 | RPAR |
| 12 | LBR |
| 13 | RBR |
| 14 | PLUS |
| 15 | MINUS |
| 16 | MULTIPLY |
| 17 | DIVIDE |
| 18 | PLUSEQUAL |
| 19 | MINUSEQUAL |
| 20 | MULTIPLYEQUAL |
| 21 | DIVIDEEQUAL |
| 22 | LESS |
| 23 | GREATER |
| 24 | EQUAL |
| 25 | LESSEQUAL |
| 26 | GREATEREQUAL |
| 27 | NOT |
| 28 | NOTEQUAL |
| 29 | EQUALEQUAL |
| 30 | COLON |
| 31 | EF |
| 32 | QUOTE |
| 33 | COMMA |
| 34 | PRINT |
| 35 | MAX |
| 36 | AND |
| 37 | LEN |
| 38 | INPUT |
| 39 | INT |
| 40 | FOR |
| 41 | IN |
| 42 | RANGE |
| 43 | DEF |
| 44 | RETURN |
| 45 | ERR |

**ПРИЛОЖЕНИЕ Б ТЕКСТ ПРОГРАММЫ**

#include <iostream>

#include <vector>

#include <map>

#include "Lexer.cpp"

using namespace std;

class Parser {

private:

enum Kind {

VAR, CONSTN, CONSTD, CONSTS,

ADD, SUB, MULTIPLY, DIVIDE,

PLUSEQ,

LT, LE, GT, GE, EQ,

SET, RANGE,

IF, ELIF, ELSE,

WHILE, FOR,

EMPTY, SEQ, PROG,

INT, PRINT, INPUT, MAX,

EXPR

};

std::map<Kind, string> symbolsKind = {

{VAR, ""},

{CONSTN, "Number"},

{CONSTD, "Decimal Number"},

{CONSTS, "String"},

{ADD, "+"},

{SUB, "-"},

{MULTIPLY, "\*"},

{DIVIDE, "/"},

{LT, "<"},

{LE, "<="},

{GT, ">"},

{GE, ">="},

{EQ, "="},

{SET, "="},

{RANGE, "RANGE"},

{IF, "IF"},

{ELIF, "ELIF"},

{ELSE, "ELSE"},

{WHILE, "WHILE"},

{FOR, "FOR"},

{EMPTY, "EMPTY"},

{SEQ, "SEQ"},

{MAX, "MAX"},

{PROG, "PROG"},

{INT, "INT"},

{PRINT, "PRINT"},

{INPUT, "INPUT"},

{EXPR, ""},

};

struct Node {

vector<Node\*> op;

string value;

Kind kind;

};

Lexer lexer;

Tocken currTocken;

public:

Parser(string filename) {

lexer = Lexer(filename);

}

Node\* createNode(Kind kind, Node\* op1) {

Node\* newNode = new Node();

newNode->kind = kind;

newNode->op.push\_back(op1);

return newNode;

}

Node\* term() {

if (this->currTocken.symb == Const::INT || this->currTocken.symb == Const::PRINT || this->currTocken.symb == Const::INPUT || this->currTocken.symb == Const::MAX) {

return this->parseFunctions();

} else if (this->currTocken.symb == Const::ID) {

Node\* nodeId = new Node();

nodeId->kind = VAR;

nodeId->value = currTocken.value;

this->currTocken = this->lexer.getNextTocken();

return nodeId;

} else if (this->currTocken.symb == Const::NUM) {

Node\* nodeNum = new Node();

nodeNum->kind = CONSTN;

nodeNum->value = this->currTocken.value;

this->currTocken = this->lexer.getNextTocken();

return nodeNum;

} else if (this->currTocken.symb == Const::DNUM) {

Node\* nodeDoubleNum = new Node();

nodeDoubleNum->kind = CONSTD;

nodeDoubleNum->value = this->currTocken.value;

this->currTocken = this->lexer.getNextTocken();

return nodeDoubleNum;

} else if (this->currTocken.symb == Const::STR) {

Node\* nodeStr = new Node();

nodeStr->kind = CONSTS;

nodeStr->value = this->currTocken.value;

this->currTocken = this->lexer.getNextTocken();

return nodeStr;

} else {

return this->parseParentExpression();

}

}

Node\* parseFunctions() {

if (this->currTocken.symb == Const::INT) {

Node\* nodeInt = new Node();

nodeInt->kind = INT;

this->currTocken = this->lexer.getNextTocken();

nodeInt->op.push\_back(this->parseParentExpression());

return nodeInt;

} else if (this->currTocken.symb == Const::PRINT) {

Node\* nodePrint = new Node();

nodePrint->kind = PRINT;

this->currTocken = this->lexer.getNextTocken();

nodePrint->op.push\_back(this->parseParentExpression());

return nodePrint;

} else if (this->currTocken.symb == Const::INPUT) {

Node\* nodeInput = new Node();

nodeInput->kind = INPUT;

this->currTocken = this->lexer.getNextTocken();

nodeInput->op.push\_back(this->parseParentExpression());

return nodeInput;

} else if (this->currTocken.symb == Const::MAX) {

Node\* nodeMAX = new Node();

nodeMAX->kind = MAX;

this->currTocken = this->lexer.getNextTocken();

if (this->currTocken.symb != Const::LPAR) {

this->showError("SYNTAX ERROR '(' EXPECTED!");

}

this->currTocken = this->lexer.getNextTocken();

nodeMAX->op.push\_back(this->parseArithmeticExpression());

if (this->currTocken.symb != Const::COMMA) {

this->showError("EXPECTED COMMA IN MAX EXPRESSION");

}

this->currTocken = this->lexer.getNextTocken();

nodeMAX->op.push\_back(this->parseArithmeticExpression());

if (this->currTocken.symb != Const::RPAR) {

this->showError("SYNTAX ERROR ')' EXPECTED!");

}

this->currTocken = this->lexer.getNextTocken();

return nodeMAX;

}

}

Node\* parseArithmeticExpression() {

Node\* node = this->term();

if (this->currTocken.symb == Const::PLUS || this->currTocken.symb == Const::MINUS ||

this->currTocken.symb == Const::MULTIPLY || this->currTocken.symb == Const::DIVIDE) {

Node\* arithmeticExpression = new Node();

arithmeticExpression->kind = EXPR;

arithmeticExpression->op.push\_back(node);

Kind kindTemp;

while (this->currTocken.symb == Const::PLUS || this->currTocken.symb == Const::MINUS ||

this->currTocken.symb == Const::MULTIPLY || this->currTocken.symb == Const::DIVIDE) {

switch (this->currTocken.symb) {

case Const::PLUS:

kindTemp = ADD;

break;

case Const::MINUS:

kindTemp = SUB;

break;

case Const::MULTIPLY:

kindTemp = MULTIPLY;

break;

case Const::DIVIDE:

kindTemp = DIVIDE;

break;

}

this->currTocken = this->lexer.getNextTocken();

Node\* n = new Node();

n->kind = kindTemp;

n->op.push\_back(this->term());

arithmeticExpression->op.push\_back(n);

}

return arithmeticExpression;

}

return node;

}

Node\* parseCompareExpression() {

Node\* n = this->parseArithmeticExpression();

this->parseQuote();

if (this->currTocken.symb == Const::LESS) {

this->currTocken = this->lexer.getNextTocken();

Node\* nodeComp = new Node();

nodeComp->kind = LT;

nodeComp->op.push\_back(this->parseArithmeticExpression());

n->op.push\_back(nodeComp);

} else if (this->currTocken.symb == Const::GREATER) {

this->currTocken = this->lexer.getNextTocken();

Node\* nodeComp = new Node();

nodeComp->kind = GT;

nodeComp->op.push\_back(this->parseArithmeticExpression());

n->op.push\_back(nodeComp);

} else if (this->currTocken.symb == Const::GREATEREQUAL) {

this->currTocken = this->lexer.getNextTocken();

Node\* nodeComp = new Node();

nodeComp->kind = GE;

nodeComp->op.push\_back(this->parseArithmeticExpression());

n->op.push\_back(nodeComp);

} else if (this->currTocken.symb == Const::LESSEQUAL) {

this->currTocken = this->lexer.getNextTocken();

Node\* nodeComp = new Node();

nodeComp->kind = LE;

nodeComp->op.push\_back(this->parseArithmeticExpression());

n->op.push\_back(nodeComp);

} else if (this->currTocken.symb == Const::EQUALEQUAL) {

this->currTocken = this->lexer.getNextTocken();

Node\* nodeComp = new Node();

nodeComp->kind = EQ;

nodeComp->op.push\_back(this->parseArithmeticExpression());

n->op.push\_back(nodeComp);

} else if (this->currTocken.symb == Const::NOTEQUAL) {

this->currTocken = this->lexer.getNextTocken();

Node\* nodeComp = new Node();

nodeComp->kind = EQ;

nodeComp->op.push\_back(this->parseArithmeticExpression());

n->op.push\_back(nodeComp);

} else if (this->currTocken.symb == Const::PLUSEQUAL) {

this->currTocken = this->lexer.getNextTocken();

Node\* nodePlusEq = new Node();

nodePlusEq->kind = PLUSEQ;

nodePlusEq->op.push\_back(this->parseArithmeticExpression());

n->op.push\_back(nodePlusEq);

}

return n;

}

Node\* parseExpression() {

if (this->currTocken.symb != Const::ID) {

return this->parseCompareExpression();

}

Node\* n = this->parseCompareExpression();

if (n->kind == VAR && this->currTocken.symb == Const::EQUAL) {

this->currTocken = this->lexer.getNextTocken();

Node\* nodeExpr = new Node();

nodeExpr->kind = SET;

nodeExpr->op.push\_back(this->parseExpression());

n->op.push\_back(nodeExpr);

}

return n;

}

void parseQuote() {

if (this->currTocken.symb == Const::QUOTE) {

this->currTocken = this->lexer.getNextTocken();

}

}

Node\* parseParentExpression() {

if (this->currTocken.symb != Const::LPAR) {

this->showError("SYNTAX ERROR '(' EXPECTED!");

}

this->currTocken = this->lexer.getNextTocken();

this->parseQuote();

Node\* n = this->parseExpression();

this->parseQuote();

if (this->currTocken.symb != Const::RPAR) {

this->showError("SYNTAX ERROR ')' EXPECTED!");

}

this->currTocken = this->lexer.getNextTocken();

return n;

}

Node\* parseStatementFor() {

if (this->currTocken.symb != Const::RANGE) {

showError("Syntax error! Operator range expected!");

}

Node\* nodeFor = new Node();

nodeFor->kind = RANGE;

this->currTocken = this->lexer.getNextTocken();

if(this->currTocken.symb != Const::LPAR) {

this->showError("SYNTAX ERROR '(' EXPECTED!");

}

this->currTocken = this->lexer.getNextTocken();

Node\* n = this->parseExpression();

nodeFor->op.push\_back(n);

if (this->currTocken.symb == Const::COMMA) {

this->currTocken = this->lexer.getNextTocken();

Node\* n = this->parseExpression();

nodeFor->op.push\_back(n);

}

if (this->currTocken.symb != Const::RPAR) {

this->showError("SYNTAX ERROR ')' EXPECTED!");

}

this->currTocken = this->lexer.getNextTocken();

return nodeFor;

}

Node\* parseStatement() {

Node\* n = new Node();

if (this->currTocken.symb == Const::IF) {

n->kind = Kind::IF;

//if node

Node\* ifNode = new Node();

ifNode->kind = Kind::IF;

this->currTocken = this->lexer.getNextTocken();

//parse expression

if (this->currTocken.symb == Const::LPAR) {

ifNode->op.push\_back(this->parseParentExpression());

}

else {

ifNode->op.push\_back(this->parseExpression());

}

//checks if a colon is present

if (this->currTocken.symb != Const::COLON) {

this->showError("EXPECTED COLON AFTER IF STATEMENT!");

}

//parse statement

this->currTocken = this->lexer.getNextTocken();

while (this->currTocken.symb == Const::TAB) {

this->currTocken = this->lexer.getNextTocken();

ifNode->op.push\_back(this->parseStatement());

}

//append to node

n->op.push\_back(ifNode);

} else if(this->currTocken.symb == Const::ELIF){

//same for (might be more than one)

while (this->currTocken.symb == Const::ELIF) {

//creating node for elif

Node\* elifNode = new Node();

elifNode->kind = Kind::ELIF;

this->currTocken = this->lexer.getNextTocken();

//parse expression

if (this->currTocken.symb == Const::LPAR) {

elifNode->op.push\_back(this->parseParentExpression());

}

else {

elifNode->op.push\_back(this->parseExpression());

}

//checks if a colon is present

if (this->currTocken.symb != Const::COLON) {

this->showError("EXPECTED COLON AFTER IF STATEMENT!");

}

//parse statement

this->currTocken = this->lexer.getNextTocken();

while (this->currTocken.symb == Const::TAB) {

this->currTocken = this->lexer.getNextTocken();

elifNode->op.push\_back(this->parseStatement());

}

//append to node

n->op.push\_back(elifNode);

}

//same for else

}

else if (this->currTocken.symb == Const::ELSE) {

//new node for else

Node\* elseNode = new Node();

elseNode->kind = Kind::ELSE;

this->currTocken = this->lexer.getNextTocken();

//checks if a colon is present

if (this->currTocken.symb != Const::COLON) {

this->showError("EXPECTED COLON AFTER IF STATEMENT!");

}

//parse statement

this->currTocken = this->lexer.getNextTocken();

while (this->currTocken.symb == Const::TAB) {

this->currTocken = this->lexer.getNextTocken();

elseNode->op.push\_back(this->parseStatement());

}

//append to node

n->op.push\_back(elseNode);

}

else if (this->currTocken.symb == Const::WHILE) {

n->kind = Kind::WHILE;

this->currTocken = this->lexer.getNextTocken();

//parse expression(might be in brackets)

if (currTocken.symb == Const::LPAR) {

n->op.push\_back(this->parseParentExpression());

}

else {

n->op.push\_back(this->parseExpression());

}

//checks if a colon is present

if (this->currTocken.symb != Const::COLON) {

this->showError("EXPECTED COLON AFTER IF STATEMENT!");

}

this->currTocken = this->lexer.getNextTocken();

while (this->currTocken.symb == Const::TAB) {

this->currTocken = this->lexer.getNextTocken();

}

n->op.push\_back(this->parseStatement());

}

else if (this->currTocken.symb == Const::FOR) {

n->kind = Kind::FOR;

this->currTocken = this->lexer.getNextTocken();

//cannot be in brackets

//maybe there shoud be different parse function for expr

n->op.push\_back(this->term());

if (this->currTocken.symb != Const::IN) {

showError("Syntax error. Operator IN expected!");

}

this->currTocken = this->lexer.getNextTocken();

n->op.push\_back(this->parseStatementFor());

this->currTocken = this->lexer.getNextTocken();

while (this->currTocken.symb == Const::TAB) {

this->currTocken = this->lexer.getNextTocken();

n->op.push\_back(this->parseStatement());

}

}

else if (this->currTocken.symb == Const::SEMICOLON) {

//; do nothing

n->kind = EMPTY;

this->currTocken = this->lexer.getNextTocken();

}

else if (this->currTocken.symb == Const::LBRA) {

// block { }

n->kind = EMPTY;

this->currTocken = this->lexer.getNextTocken();

while (this->currTocken.symb != Const::RBRA) {

n->kind = SEQ;

n->op.push\_back(n);

n->op.push\_back(parseStatement());

}

this->currTocken = this->lexer.getNextTocken();

}

else {

n->kind = EXPR;

if (this->currTocken.symb == Const::TAB) {

while (this->currTocken.symb == Const::TAB) {

this->currTocken = this->lexer.getNextTocken();

}

}

n->op.push\_back(this->parseExpression());

}

return n;

}

Node\* Parse() {

currTocken = this->lexer.getNextTocken();

Node\* node = createNode(PROG, this->parseStatement());

while (this->currTocken.symb != Const::EF) {

node->op.push\_back(this->parseStatement());

}

cout << "Show tree: " << endl;

showTree(node);

return node;

}

string tabs = "\t";

void showTree(Node\* root) {

if (root->value != "") {

cout << tabs << root->value << endl;

}

for (int i = 0; i < root->op.size(); i++) {

cout << tabs << symbolsKind[root->kind] << endl;

tabs += '\t';

if (root->op.size() == 0) {

}

showTree(root->op[i]);

tabs.pop\_back();

}

}

void showError(string message) {

cout << message << endl;

exit(-1);

}

};

**ПРИЛОЖЕНИЕ В ТЕКСТ ПРОГРАММЫ**

1. Нахождения n чисел Фибоначчи, введенного пользователем

n = int(input("Enter positive number: "))  
  
if n <= 0:  
 print("You enter negative number or 0")  
else:  
 num\_1 = 0  
 num\_2 = 1  
   
 print(num\_2)  
   
 for i in range(n-1):  
 res = num\_1 + num\_2  
 num\_1 = num\_2  
 num\_2 = res  
   
 print(num\_2)

1. Дана последовательность натуральных чисел, завершающаяся числом 0. Определите, какое наибольшее число подряд идущих элементов этой последовательности равны друг другу.

prev = -1  
curr\_rep\_len = 0  
max\_rep\_len = 0  
  
element = int(input())  
  
while element != 0:  
 if prev == element:  
 curr\_rep\_len += 1  
 else:  
 prev = element  
 max\_rep\_len = max(max\_rep\_len, curr\_rep\_len)  
 curr\_rep\_len = 1  
   
 element = int(input())  
   
max\_rep\_len = max(max\_rep\_len, curr\_rep\_len)  
print(max\_rep\_len)

1. Вычислить факториал введенного пользователем числа.

number = int(input("Enter a number: "))  
factorial = 1  
  
if number < 0:  
 print("Sorry, factorial does not exist for negative numbers")  
elif number == 0:  
 print("The factorial of 0 is 1")  
else:  
 for i in range(1, number + 1):  
 factorial = factorial\*i  
 print(factorial)

**ПРИЛОЖЕНИЕ Г СИНТАКСИЧЕСКОЕ ДЕРЕВО**

PROG

number

=

INT

INPUT

enter a number:

PROG

factorial

=

1

PROG

IF

IF

number

<

0

IF

PRINT

sorry, factorial does not exist for negative numbers

PROG

ELIF

number

=

0

ELIF

PRINT

the factorial of 0 is 1

PROG

ELSE

FOR

i

FOR

RANGE

1

RANGE

number

+

1

FOR

factorial

=

factorial

\*

i

FOR

PRINT

factorial