

Министерство образования Республики Беларусь Учреждение  
образования  
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ  
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ  
КАФЕДРА ИНФОРМАТИКИ

Отчёт по лабораторной работе №4

По теме “Семантический анализатор”

Выполнила: студентка гр. 053501 Шурко Т.А.  
Проверил: ассистент кафедры информатики Гриценко Н. Ю.

Минск 2023

## СОДЕРЖАНИЕ

1 Цель работы .....	3
2 Теория.....	4
3 Программа и комментарии.....	5
4 Демонстрация работы.....	6
5 Демонстрация нахождения ошибок .....	7
Вывод.....	9
Приложение А Код программы .....	10
Приложение Б Код программы.....	11
Приложение В Листинг кода .....	12
Приложение Г Текст программ .....	29

# **1 ЦЕЛЬ РАБОТЫ**

Освоение работы с существующими семантическими анализаторами.

Проведение семантического анализа; нахождение, обработка и вывод минимум 2 семантических ошибок.

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

## 2 ТЕОРИЯ

В процессе семантического анализа проверяется наличие семантических ошибок в исходной программе и накапливается информация о типах для следующей стадии – генерации кода.

При семантическом анализе используются иерархические структуры, полученные во время синтаксического анализа для идентификации операторов и операндов выражений и инструкций. Важным аспектом семантического анализа является проверка типов, когда компилятор проверяет, что каждый оператор имеет операнды допустимого спецификациями языка типа. Например, определение многих языков программирования требует, чтобы при использовании действительного числа в качестве индекса массива генерировалось сообщение об ошибке. В то же время спецификация языка может позволить определенное насильственное преобразование типов, например, когда бинарный арифметический оператор применяется к операндам целого и действительного типов. В этом случае компилятору может потребоваться преобразование целого числа в действительное.

В большинстве языков программирования имеет место неявное изменение типов (иногда называемое приведением типов(*coercion*)). Реже встречаются языки, подобные Ada, в которых большинство изменений типов должно быть явным. В языках со статическими типами, например C, все типы известны во время компиляции, и это относится к типам выражений, идентификаторам и литералам. При этом неважно, насколько сложным является выражение: его тип может определяться во время компиляции за определенное количество шагов, исходя из типов его составляющих. Фактически, это позволяет производить контроль типов во время компиляции и находить заранее (в процессе компиляции, а не во время выполнения программы!) многие программные ошибки.

### 3 ПРОГРАММА И КОММЕНТАРИИ

Программа написана на с++. Семантический анализ представляет из себя дополнительные проверки при приведении и обработке типов данных. Реализация осуществлялась путем расширения класса Parser (Приложение В) программного продукта.

Для семантического анализа происходило построение дополнительной таблицы, в с++ аналогом таблицы является map (см. рисунок 3.1), с названиями переменных и их типами данных.

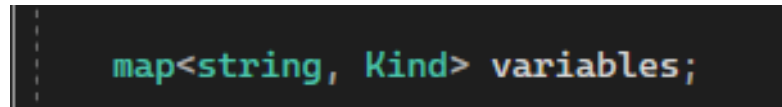
A screenshot of a code editor showing the C++ declaration `map<string, Kind> variables;`. The text is displayed in a dark-themed editor with syntax highlighting: `map` is blue, `<string, Kind>` is green, and `variables;` is orange.

Рисунок 3.1 –Таблица переменных

При необходимости вычислений осуществляются проверки на типы данных, там где это необходимо. При обнаружении несоответствия типов или невозможности проведения операции между ними, программный продукт выводит подробную ошибку и заканчивает свою работу.

## 4 ДЕМОНСТРАЦИЯ РАБОТЫ

Результат работы анализатора на рисунке 4.1 при наличии семантических ошибок в коде. При отсутствии каких-либо ошибок программный продукт продолжает свою работу.

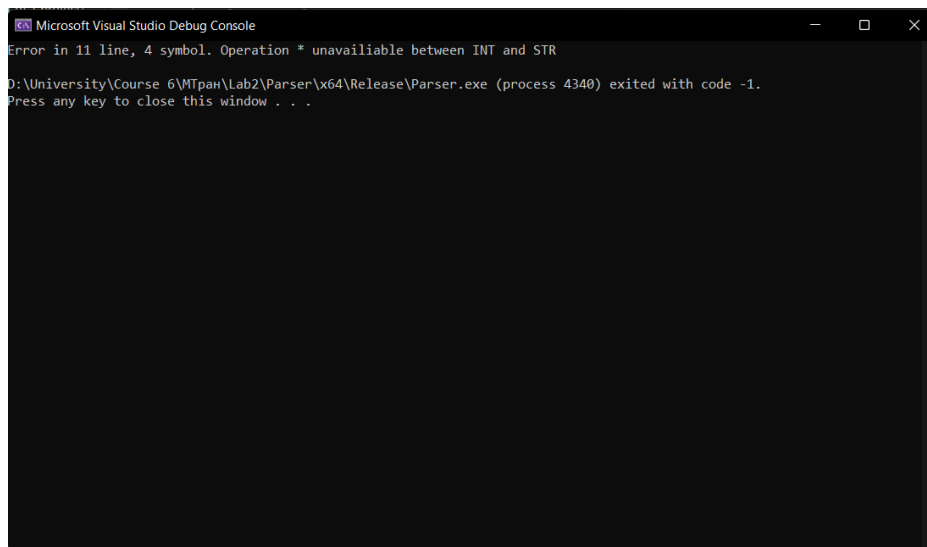


Рисунок 4.1 – Результат работы программного продукта

## 5 ДЕМОНСТРАЦИЯ НАХОЖДЕНИЯ ОШИБОК

На рисунке 5.1 семантическая ошибка в задаче 1 (приложение А): попытка провести недопустимую операцию между типами INT и STR. Одна из переменных имеет строковый тип, а другая целочисленный.

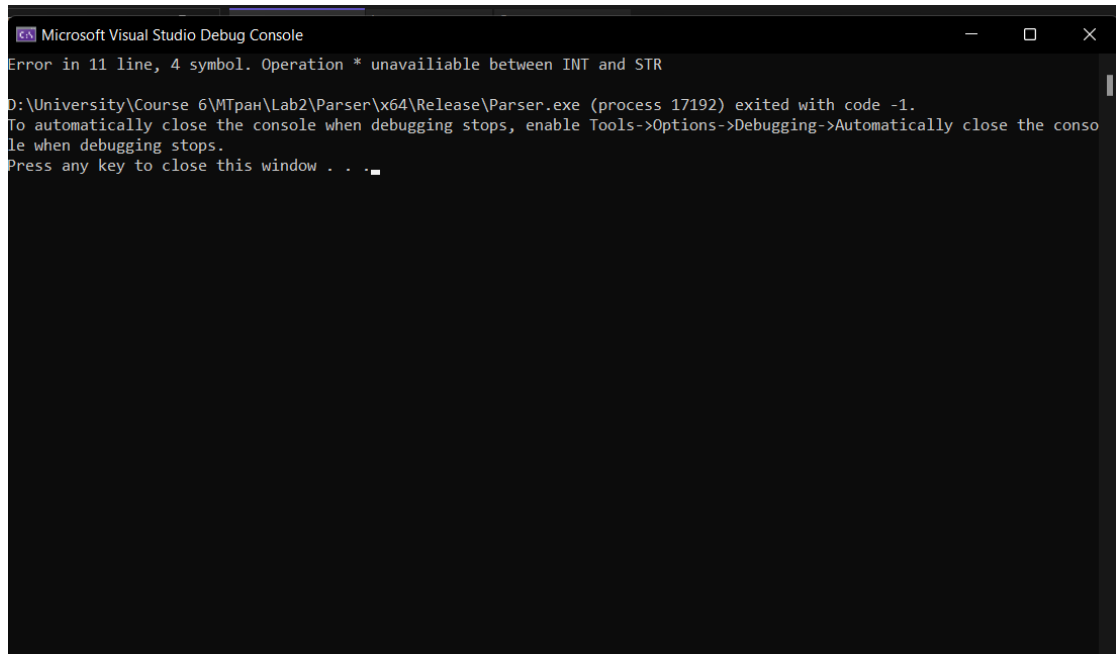


Рисунок 5.1 – Ошибка приведения типов

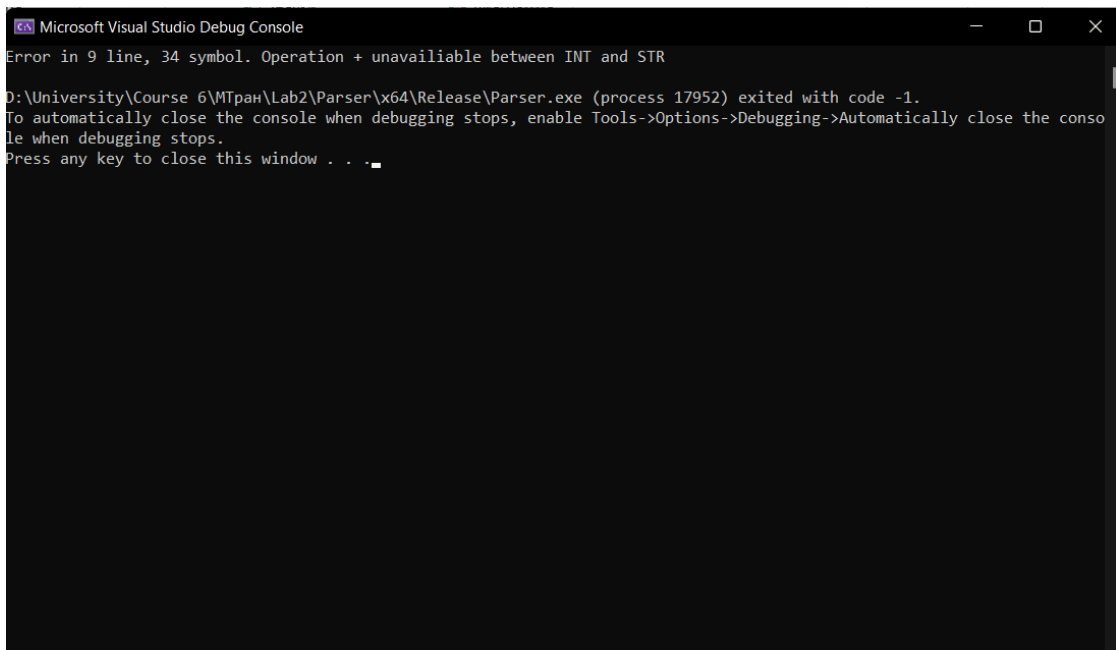


Рисунок 5.2 – Ошибка приведения типов

Данная ошибка возникает при отсутствии приведения введенного числа к типу `int` в задаче 1 (приложение Б), которое впоследствии прибавляется к целочисленному типу (см. рисунок 5.2). Исходный код задачи в приложении Г.



## **ВЫВОД**

В результате выполнения лабораторной работы была изучена теория о семантических анализаторах, разработан собственный семантический анализатор, выбранного подмножества языка программирования. Определены минимум 2 возможные семантические ошибки и показано их корректное выявление.

## ПРИЛОЖЕНИЕ А КОД ПРОГРАММЫ

```
number = int(input("Enter a number: "))
```

```
factorial = 'how'
```

```
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 Б КОД ПРОГРАММЫ

```
number = 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 В ЛИСТИНГ КОДА

```
#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, "INT"},
        {CONSTD, "FLOAT"},
        {CONSTS, "STR"},
        {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, ""},
};

map<string, Kind> variables;

struct Node {
    vector<Node*> op;
    string value;
    Kind kind;
};

Lexer lexer;
Token currToken;

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->currToken.symb == Const::INT || this->currToken.symb ==
Const::PRINT || this->currToken.symb == Const::INPUT || this->currToken.symb ==
Const::MAX) {

            return this->parseFunctions();
        } else if (this->currToken.symb == Const::ID) {
            Node* nodeId = new Node();

            nodeId->kind = VAR;
            nodeId->value = currToken.value;

            this->currToken = this->lexer.getNextToken();

            return nodeId;
        } else if (this->currToken.symb == Const::NUM) {
            Node* nodeNum = new Node();
            nodeNum->kind = CONSTN;
            nodeNum->value = this->currToken.value;

            this->currToken = this->lexer.getNextToken();

            return nodeNum;
        } else if (this->currToken.symb == Const::DNUM) {
            Node* nodeDoubleNum = new Node();
            nodeDoubleNum->kind = CONSTD;
            nodeDoubleNum->value = this->currToken.value;

            this->currToken = this->lexer.getNextToken();

            return nodeDoubleNum;
        } else if (this->currToken.symb == Const::STR) {

```

```

        Node* nodeStr = new Node();
        nodeStr->kind = CONSTS;
        nodeStr->value = this->currToken.value;

        this->currToken = this->lexer.getNextToken();

        return nodeStr;
    } else {
        return this->parseParentExpression();
    }
}

Node* parseFunctions() {
    if (this->currToken.symb == Const::INT) {
        Node* nodeInt = new Node();

        nodeInt->kind = INT;
        this->currToken = this->lexer.getNextToken();

        nodeInt->op.push_back(this->parseParentExpression());

        return nodeInt;

    } else if (this->currToken.symb == Const::PRINT) {
        Node* nodePrint = new Node();
        nodePrint->kind = PRINT;

        this->currToken = this->lexer.getNextToken();

        nodePrint->op.push_back(this->parseParentExpression());

        return nodePrint;

    } else if (this->currToken.symb == Const::INPUT) {
        Node* nodeInput = new Node();

```

```

nodeInput->kind = INPUT;

this->currToken = this->lexer.getNextToken();

nodeInput->op.push_back(this->parseParentExpression());

return nodeInput;
} else if (this->currToken.symb == Const::MAX) {
    Node* nodeMAX = new Node();
    nodeMAX->kind = MAX;

    this->currToken = this->lexer.getNextToken();

    if (this->currToken.symb != Const::LPAR) {
        this->showError("SYNTAX ERROR '(' EXPECTED!");
    }

    this->currToken = this->lexer.getNextToken();
    nodeMAX->op.push_back(this->parseArithmeticExpression());

    if (this->currToken.symb != Const::COMMA) {
        this->showError("EXPECTED COMMA IN MAX EXPRESSION");
    }
    this->currToken = this->lexer.getNextToken();
    nodeMAX->op.push_back(this->parseArithmeticExpression());

    if (this->currToken.symb != Const::RPAR) {
        this->showError("SYNTAX ERROR ')' EXPECTED!");
    }
    this->currToken = this->lexer.getNextToken();
    return nodeMAX;
}
}

Node* parseArithmeticExpression() {

```



```

Node* node = this->term();

    if (this->currToken.symb == Const::PLUS || this->currToken.symb ==
Const::MINUS ||
        this->currToken.symb == Const::MULTIPLY || this->currToken.symb
== Const::DIVIDE) {
        Node* arithmeticExpression = new Node();
        arithmeticExpression->kind = EXPR;
        arithmeticExpression->op.push_back(node);

        Kind kindTemp;
        while (this->currToken.symb == Const::PLUS || this-
>currToken.symb == Const::MINUS ||
            this->currToken.symb == Const::MULTIPLY || this-
>currToken.symb == Const::DIVIDE) {
            switch (this->currToken.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->currToken = this->lexer.getNextToken();

            Node* n = new Node();
            n->kind = kindTemp;
            n->op.push_back(this->term());
            arithmeticExpression->op.push_back(n);

```

```

        }

        return arithmeticExpression;
    }

    return node;
}

Node* parseCompareExpression() {
    this->parseQuote();
    Node* n = this->parseArithmeticExpression();
    this->parseQuote();
    if (this->currToken.symb == Const::LESS) {
        this->currToken = this->lexer.getNextToken();
        Node* nodeComp = new Node();
        nodeComp->kind = LT;
        nodeComp->op.push_back(this->parseArithmeticExpression());
        n->op.push_back(nodeComp);
    } else if (this->currToken.symb == Const::GREATER) {
        this->currToken = this->lexer.getNextToken();
        Node* nodeComp = new Node();
        nodeComp->kind = GT;
        nodeComp->op.push_back(this->parseArithmeticExpression());
        n->op.push_back(nodeComp);
    } else if (this->currToken.symb == Const::GREATEREQUAL) {
        this->currToken = this->lexer.getNextToken();
        Node* nodeComp = new Node();
        nodeComp->kind = GE;
        nodeComp->op.push_back(this->parseArithmeticExpression());
        n->op.push_back(nodeComp);
    } else if (this->currToken.symb == Const::LESSEQUAL) {
        this->currToken = this->lexer.getNextToken();
        Node* nodeComp = new Node();
        nodeComp->kind = LE;
        nodeComp->op.push_back(this->parseArithmeticExpression());
        n->op.push_back(nodeComp);
    }
}

```

```

    } else if (this->currToken.symb == Const::EQUALEQUAL) {
        this->currToken = this->lexer.getNextToken();
        Node* nodeComp = new Node();
        nodeComp->kind = EQ;
        nodeComp->op.push_back(this->parseArithmeticExpression());
        n->op.push_back(nodeComp);
    } else if (this->currToken.symb == Const::NOTEQUAL) {
        this->currToken = this->lexer.getNextToken();
        Node* nodeComp = new Node();
        nodeComp->kind = EQ;
        nodeComp->op.push_back(this->parseArithmeticExpression());
        n->op.push_back(nodeComp);
    } else if (this->currToken.symb == Const::PLUSEQUAL) {
        this->currToken = this->lexer.getNextToken();
        Node* nodePlusEq = new Node();
        nodePlusEq->kind = PLUSEQ;
        nodePlusEq->op.push_back(this->parseArithmeticExpression());
        n->op.push_back(nodePlusEq);
    }

    return n;
}

Node* parseExpression() {

    if (this->currToken.symb != Const::ID) {
        return this->parseCompareExpression();
    }

    Node* n = this->parseCompareExpression();

    if (n->op.size() > 0 && n->kind == Kind::EXPR) {
        for (int i = 0; i < n->op.size(); i++) {
            if ((n->op[i])->kind == Kind::ADD || (n->op[i])->kind ==
Kind::SUB

```

```

        || (n->op[i])->kind == Kind::MULTIPLY || (n->op[i])->kind == Kind::DIVIDE) {
            for (int j = 0; j < ((n->op[i])->op).size(); j++) {
                if (((n->op[i])->op[j])->kind != Kind::VAR && ((n->op[i])->op[j])->kind != variables[n->op[0]->value]) {
                    this->showError("Operation " + symbolsKind[(n->op[i])->kind] + " unavailable between " + symbolsKind[((n->op[i])->op[j])->kind] + " and " + symbolsKind[variables[n->op[0]->value]]);
                }
            }
        }
        else if (((n->op[i])->op[j])->kind == Kind::VAR && variables[((n->op[i])->op[j])->value] != variables[n->op[0]->value]) {
            this->showError("Operation " + symbolsKind[(n->op[i])->kind] + " unavailable between " + symbolsKind[variables[((n->op[i])->op[j])->value]] + " and " + symbolsKind[variables[n->op[0]->value]]);
        }
    }
}

//
}
}
}

if (n->kind == VAR && this->currToken.symb == Const::EQUAL) {
    this->currToken = this->lexer.getNextToken();

    Node* nodeExpr = new Node();

    nodeExpr->kind = SET;
    nodeExpr->op.push_back(this->parseExpression());
    if (nodeExpr->op.size() > 0 && nodeExpr->op[0]->kind == Kind::INT)
    {
        variables.insert({ n->value, Kind::CONSTN });
    }
    else if (nodeExpr->op.size() > 0 && (nodeExpr->op[0]->kind == Kind::CONSTD || nodeExpr->op[0]->kind == Kind::CONSTN || nodeExpr->op[0]->kind == Kind::CONSTS)) {
        variables.insert({ n->value, nodeExpr->op[0]->kind });
    }
}

```

```

    }
    return n;
}

void parseQuote() {
    if (this->currToken.symb == Const::QUOTE) {
        this->currToken = this->lexer.getNextToken();
    }
}

Node* parseParentExpression() {
    if (this->currToken.symb != Const::LPAR) {
        this->showError("SYNTAX ERROR '(' EXPECTED!");
    }
    this->currToken = this->lexer.getNextToken();
    this->parseQuote();
    Node* n = this->parseExpression();

    this->parseQuote();

    if (this->currToken.symb != Const::RPAR) {
        this->showError("SYNTAX ERROR ')' EXPECTED!");
    }
    this->currToken = this->lexer.getNextToken();

    return n;
}

Node* parseStatementFor() {
    if (this->currToken.symb != Const::RANGE) {
        showError("Syntax error! Operator range expected!");
    }
    Node* nodeFor = new Node();
    nodeFor->kind = RANGE;

    this->currToken = this->lexer.getNextToken();

```

```

        if(this->currToken.symb != Const::LPAR) {
            this->showError("SYNTAX ERROR '(' EXPECTED!");
        }
        this->currToken = this->lexer.getNextToken();

        Node* n = this->parseExpression();
        nodeFor->op.push_back(n);

        if (this->currToken.symb == Const::COMMA) {
            this->currToken = this->lexer.getNextToken();
            Node* n = this->parseExpression();
            nodeFor->op.push_back(n);
        }

        if (this->currToken.symb != Const::RPAR) {
            this->showError("SYNTAX ERROR ')' EXPECTED!");
        }

        this->currToken = this->lexer.getNextToken();

        return nodeFor;
    }

Node* parseStatement() {
    Node* n = new Node();
    if (this->currToken.symb == Const::IF) {
        n->kind = Kind::IF;
        //if node
        Node* ifNode = new Node();
        ifNode->kind = Kind::IF;

        this->currToken = this->lexer.getNextToken();

        //parse expression

```

```

        if (this->currToken.symb == Const::LPAR) {
            ifNode->op.push_back(this->parseParentExpression());
        }
        else {
            ifNode->op.push_back(this->parseExpression());
        }
        //checks if a colon is present
        if (this->currToken.symb != Const::COLON) {
            this->showError("EXPECTED COLON AFTER IF STATEMENT!");
        }

        //parse statement
        this->currToken = this->lexer.getNextToken();

        while (this->currToken.symb == Const::TAB) {
            this->currToken = this->lexer.getNextToken();
            ifNode->op.push_back(this->parseStatement());
        }

        //append to node
        n->op.push_back(ifNode);
    } else if(this->currToken.symb == Const::ELIF){
        //same for (might be more than one)
        while (this->currToken.symb == Const::ELIF) {
            //creating node for elif
            Node* elifNode = new Node();
            elifNode->kind = Kind::ELIF;

            this->currToken = this->lexer.getNextToken();

            //parse expression
            if (this->currToken.symb == Const::LPAR) {
                elifNode->op.push_back(this->
>parseParentExpression());
            }
        }
    }

```

```

else {
    elifNode->op.push_back(this->parseExpression());
}

//checks if a colon is present
if (this->currToken.symb != Const::COLON) {
    this->showError("EXPECTED      COLON      AFTER      IF
STATEMENT!");
}

//parse statement
this->currToken = this->lexer.getNextToken();

while (this->currToken.symb == Const::TAB) {
    this->currToken = this->lexer.getNextToken();
    elifNode->op.push_back(this->parseStatement());
}

//append to node
n->op.push_back(elifNode);
}
//same for else
}
else if (this->currToken.symb == Const::ELSE) {
    //new node for else
    Node* elseNode = new Node();
    elseNode->kind = Kind::ELSE;

    this->currToken = this->lexer.getNextToken();
    //checks if a colon is present
    if (this->currToken.symb != Const::COLON) {
        this->showError("EXPECTED      COLON      AFTER      IF
STATEMENT!");
    }

    //parse statement

```



```

        this->currToken = this->lexer.getNextToken();

        while (this->currToken.symb == Const::TAB) {
            this->currToken = this->lexer.getNextToken();
            elseNode->op.push_back(this->parseStatement());
        }

        //append to node
        n->op.push_back(elseNode);
    }
else if (this->currToken.symb == Const::WHILE) {
    n->kind = Kind::WHILE;
    this->currToken = this->lexer.getNextToken();

    //parse expression(might be in brackets)
    if (currToken.symb == Const::LPAR) {
        n->op.push_back(this->parseParentExpression());
    }
    else {
        n->op.push_back(this->parseExpression());
    }

    //checks if a colon is present
    if (this->currToken.symb != Const::COLON) {
        this->showError("EXPECTED COLON AFTER IF STATEMENT!");
    }
    this->currToken = this->lexer.getNextToken();
    while (this->currToken.symb == Const::TAB) {
        this->currToken = this->lexer.getNextToken();
    }
    n->op.push_back(this->parseStatement());
}
else if (this->currToken.symb == Const::FOR) {
    n->kind = Kind::FOR;

```

```

this->currToken = this->lexer.getNextToken();
//cannot be in brackets
//maybe there should be different parse function for expr
n->op.push_back(this->term());

variables.insert({ n->op[0]->value, Kind::CONSTN });
if (this->currToken.symb != Const::IN) {
    showError("Syntax error. Operator IN expected!");
}
this->currToken = this->lexer.getNextToken();

n->op.push_back(this->parseStatementFor());

this->currToken = this->lexer.getNextToken();

while (this->currToken.symb == Const::TAB) {
    this->currToken = this->lexer.getNextToken();
    n->op.push_back(this->parseStatement());
}
}
else if (this->currToken.symb == Const::SEMICOLON) {
    //; do nothing
    n->kind = EMPTY;
    this->currToken = this->lexer.getNextToken();
}
else if (this->currToken.symb == Const::LBRA) {
    // block { }
    n->kind = EMPTY;
    this->currToken = this->lexer.getNextToken();
    while (this->currToken.symb != Const::RBRA) {
        n->kind = SEQ;
        n->op.push_back(n);
        n->op.push_back(parseStatement());
    }
    this->currToken = this->lexer.getNextToken();
}

```

```

    }
    else {
        n->kind = EXPR;
        if (this->currToken.symb == Const::TAB) {
            while (this->currToken.symb == Const::TAB) {
                this->currToken = this->lexer.getNextToken();
            }
        }
        n->op.push_back(this->parseExpression());
    }
    return n;
}

```

```

Node* Parse() {
    currToken = this->lexer.getNextToken();

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

    while (this->currToken.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 << "Error in " << lexer.getLine() << " line, "
<<lexer.getSymbolCounter() << " symbol. " << message << endl;
    exit(-1);
}

int getLine() {
    return lexer.getLine();
}

int getSymbolCounter() {
    return lexer.getSymbolCounter();
}

};

```

## ПРИЛОЖЕНИЕ Г ТЕКСТ ПРОГРАММ

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)
```

2. 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)
```

3. 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)
```