“Национальный Исследовательский Университет
“Московский Энергетический Институт”

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №2

Дмитрий Пугачев, студент группы А-05-19

Москва 2022 год.

Задание:

Разработать алгоритм и реализовать программу для грамматического анализа методом рекурсивного спуска.

Для каждого индивидуального варианта задаётся набор операторов (подмножество языка Паскаль).

Часть 1
Необходимо описать правила языка в форме БНФ. По данным правилам описать грамматику языка. Разработанную грамматику преобразовать к форме автоматной грамматики. Результаты показать преподавателю.

Часть 2
По заданной грамматике построить ДКА, распознающий грамматику, и только её. Результаты показать преподавателю.

Часть 3
Разработать алгоритм синтаксического анализа методом рекурсивного спуска. Предусмотреть тесты для проверки распознавания всех лексем.

В каждом варианте должен быть реализован следующий функционал:

- оператор присваивания
- арифметические операции + |- | * | /
- арифметические выражения
- Типы данных: Integer, Boolean

Выполнение:

Часть 1

Вариант 21
Program, var, begin, end, write, read, if, for, текстовые строки, правильные скобочные выражения

<prog> ::= program <program_name>; var <var_part> | program <program_name>; begin <statement_part>

<var_part> ::= <var_list>; begin <statement_part>

<statement_part> ::= <statement_list>; end. | end.

<program_name> ::= <identifier>

<bool> ::= false | true

<var> ::= <identifier_list> : <type>

<var_list> ::= <var> | <var>; <var_list>

<type> ::= Integer | Boolean | String

<identifier_list> ::= <identifier> | <identifier>, <identifier_list>

<statement_list> ::= <statement> | <statement>; <statement_list>

<statement> ::= <assignment> | <read> | <write> | <for> | <if> |
begin <statement_list> end | begin end

<assignment> ::= <identifier> := <expression>

<comparison_operator> ::= <> | < | > | <= | >= | =

<identifier> ::= <letter> | <letter><chars>

<string> ::= “<chars>”|”<chars>’ ‘<chars>”

<chars> ::= <char>|<char><chars>

<char> ::= <letter>|<digit>

<letter> ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q
| R | S | T | U | V | W | X | Y | Z | a | b | c | d | e | f | g | h | i | j | k
| l | m | n | o | p | q | r | s | t | u | v | w | x | y | z | _

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<read> ::= read(<identifier_list>) | read()

<write> ::= write(<identifier_list>) | write()

<int> ::= -<unsigned_int> | <unsigned_int>

<unsigned_int> ::= <digit> | <digit><unsigned_int>

<operation> ::= + | - | * | /

<formula> ::= <operand> | <formula><operation><formula>

<expression> ::= <formula> | <string> | <bool>

<operand> ::= <int> | <identifier> | (<formula>)

<for> ::= for <identifier> := <formula> to <formula> do <statement> | for
<assignment> downto <formula> do <statement>

<if> ::= if <bool_expr> then <statement>

<bool_expr> ::= (<bool_expr> and <bool_expr>) | (<bool_expr> or <bool_expr>) |
not <bool_expr> | <simple_bool_expr>

<simple_bool_expr> ::= <expression> |
<expression><comparison_operator><expression> | <bool>

Грамматика

1.Алфавит терминальных символов:

T={program, var, begin, end, read, write, if, then, else, for, to, do, integer, boolean, a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z, A, B, C, D, E, F, G, H, I, G, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, +, -, _, *, /, or, and, not, true, false, >=, =, =<, <, >, :=, ", ", ".", ";", ":", "(", ")"}

2.Алфавит нетерминальных символов:

N={<prog>, <var_part>, <statement_part>, <program_name>, <bool>, <var>, <var_list>, <i_b_type>, <identifier_list>, <statement_list>, <statement>, <assignment>, <comparison_operator>, <identifier>, <string>, <chars>, <char>, <letter>, <digit>, <read>, <write>, <int>, <unsigned_int>, <operation>, <formula>, <operand>, <for>, <if>, <bool_expr>, <simple_bool_expr>}

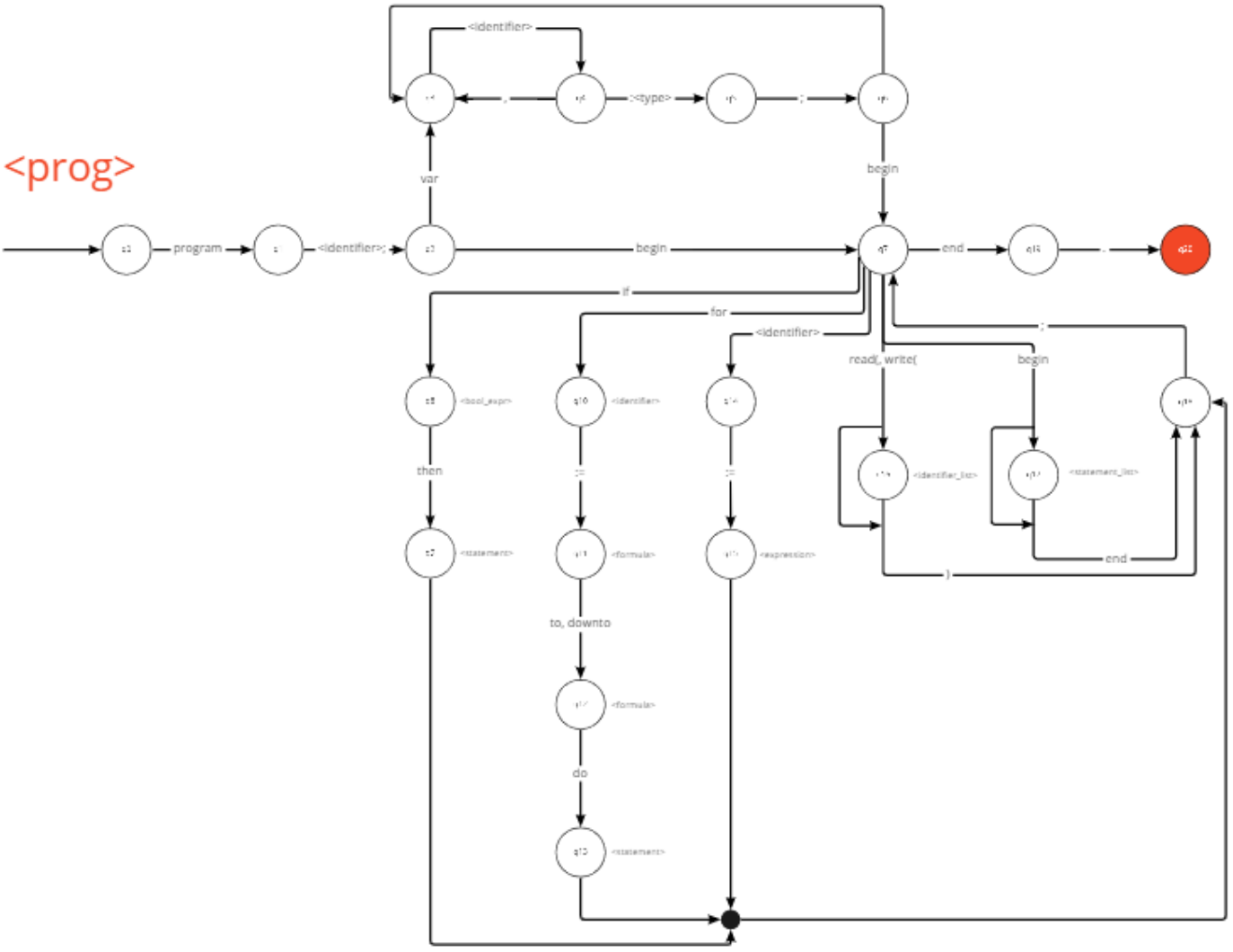
3.Начальный нетерминал

Z = { <prog> }

Часть 2

Построенный ДКА:

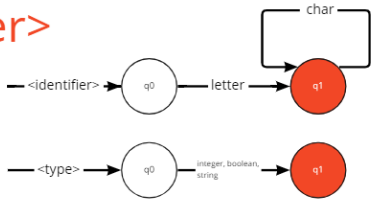
Начальный нетерминал <prog>:



Identifier, type:

<identifier>

<type>



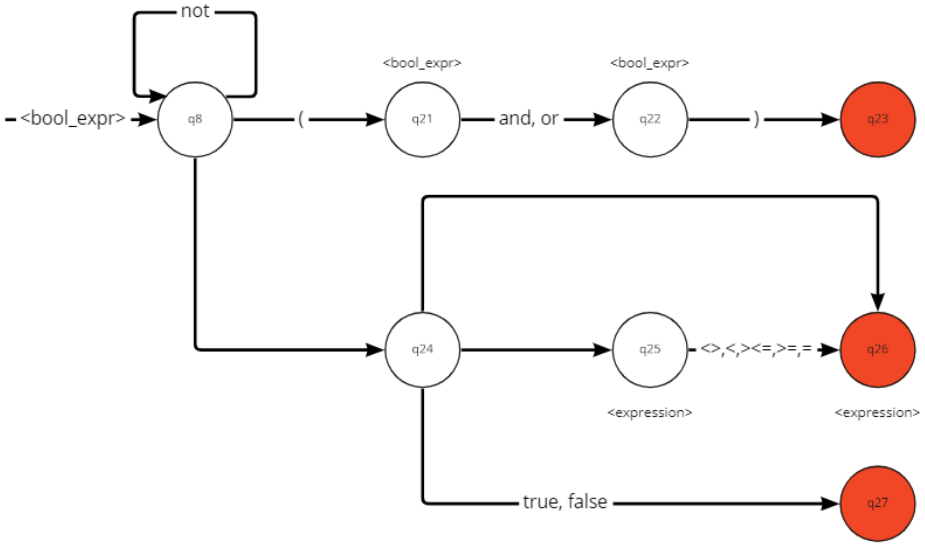
<letter> = A | B | C | ... | y | z | _

<digit> = 0 | 1 | ... | 9

<char> = <letter> | <digit>

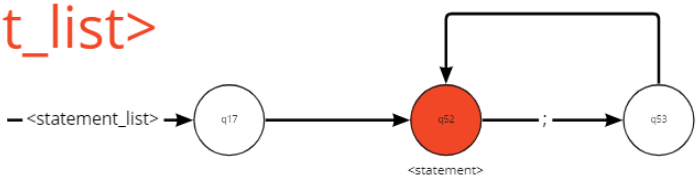
Bool expr:

<bool_expr>

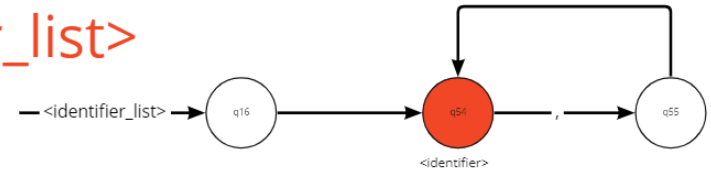


Statement list, identifier list:

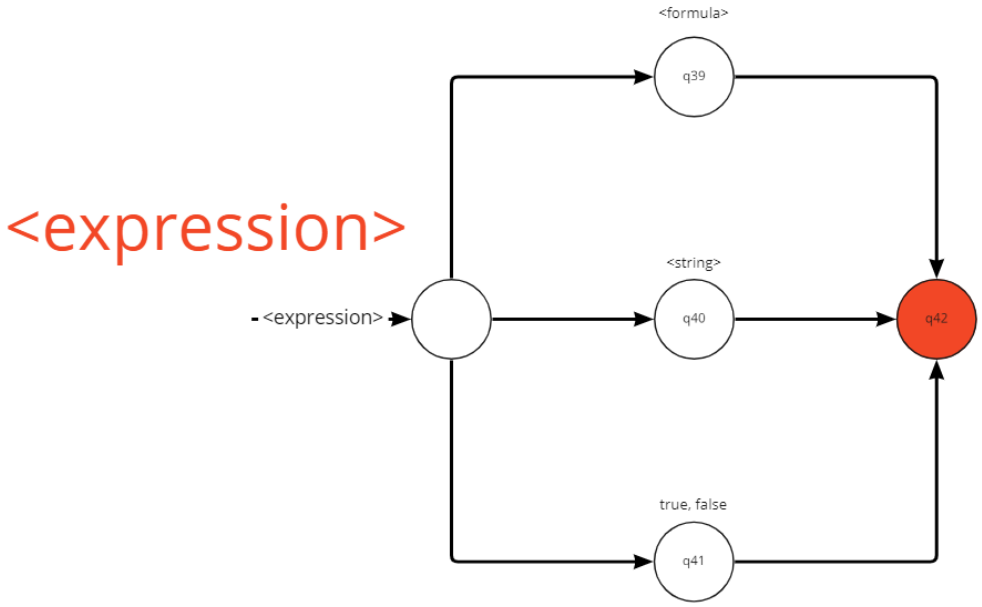
<statement_list>



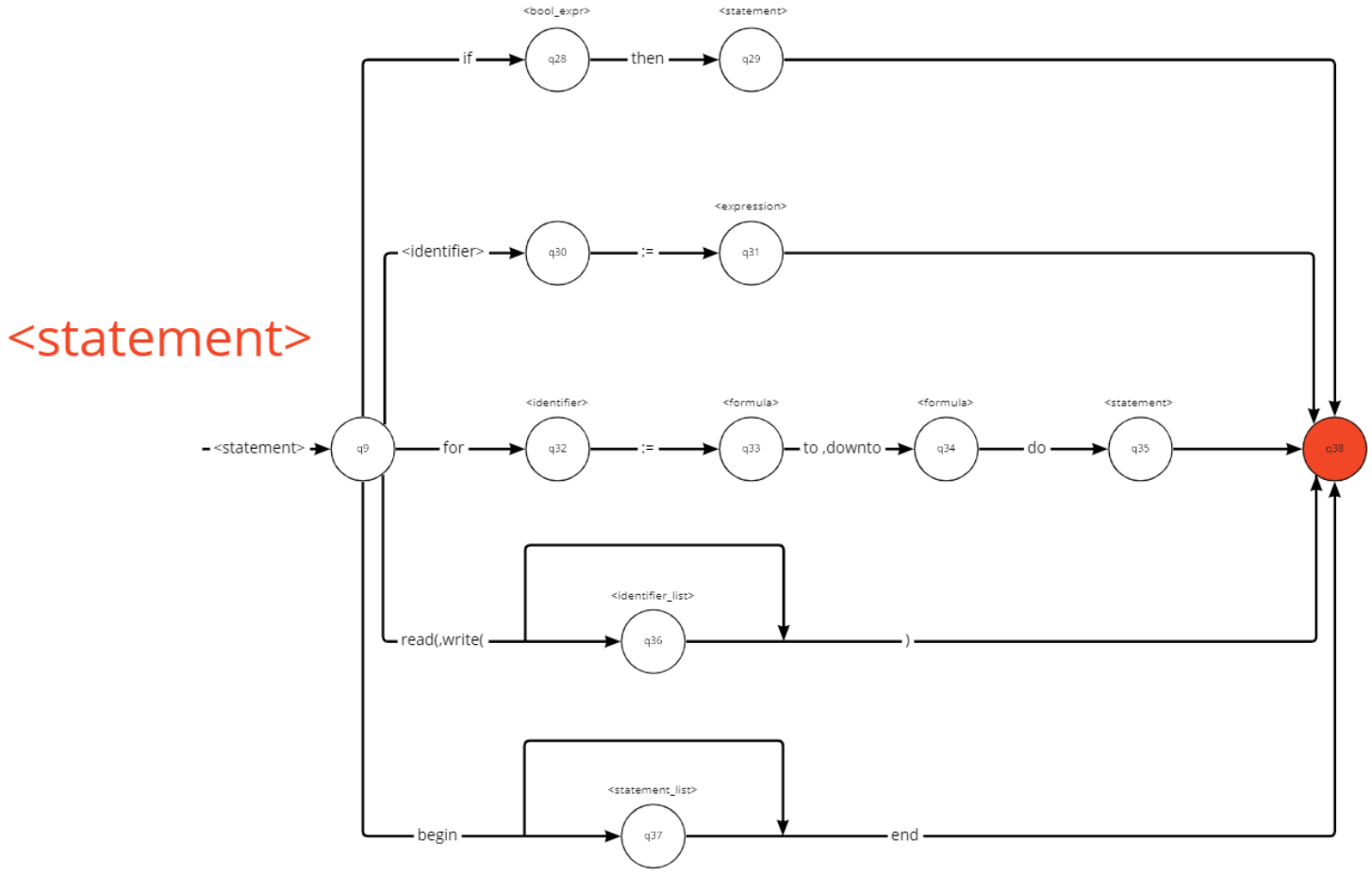
<identifier_list>



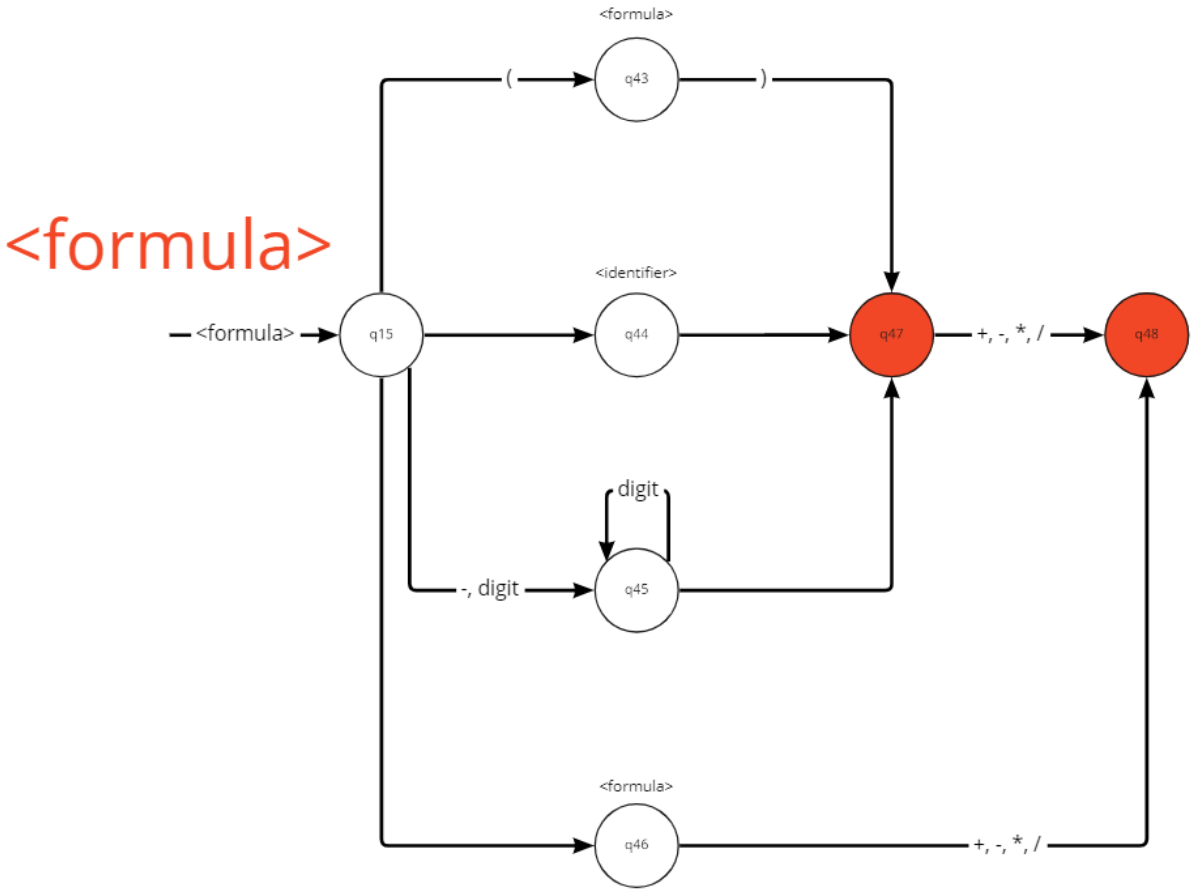
Expression:



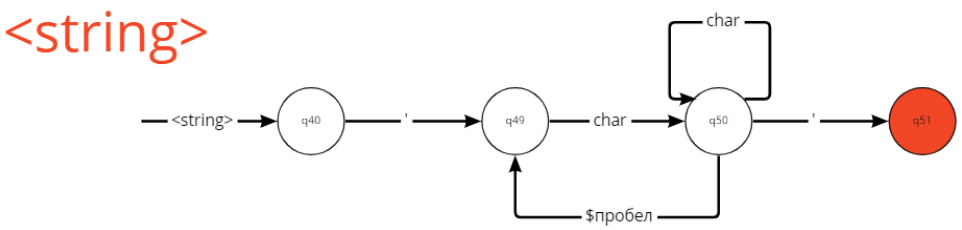
Statement:



Formula:



String:

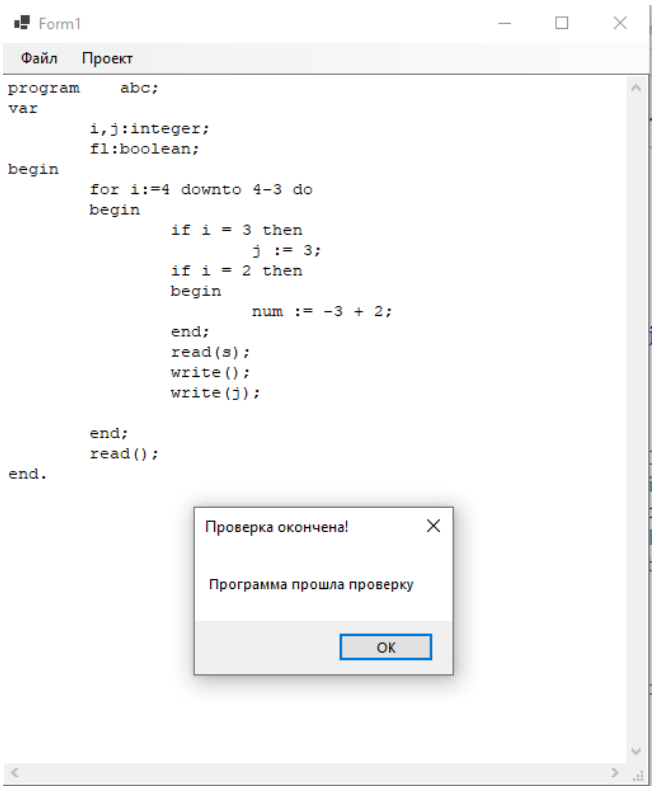


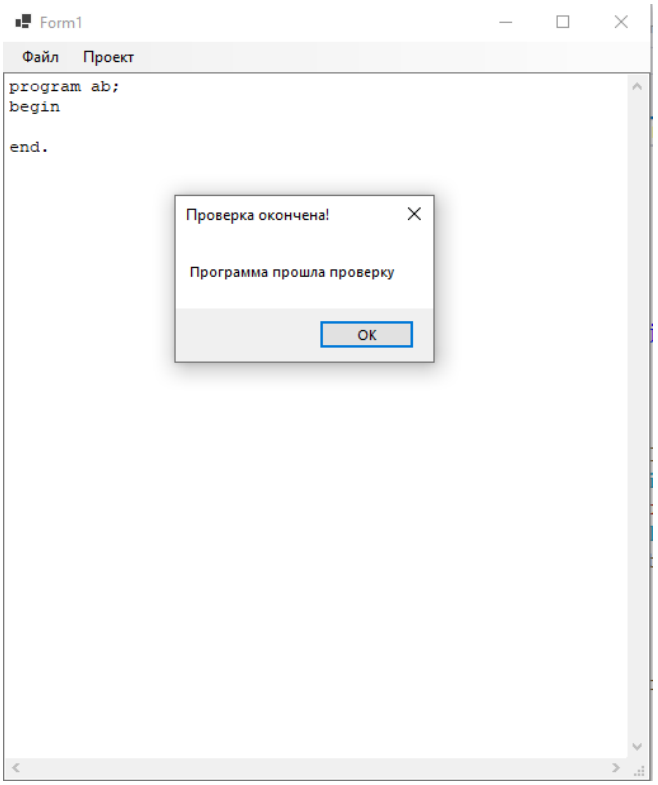
Полностью можно посмотреть по этой ссылке:
https://miro.com/app/board/uXjVOC5rPD0=/

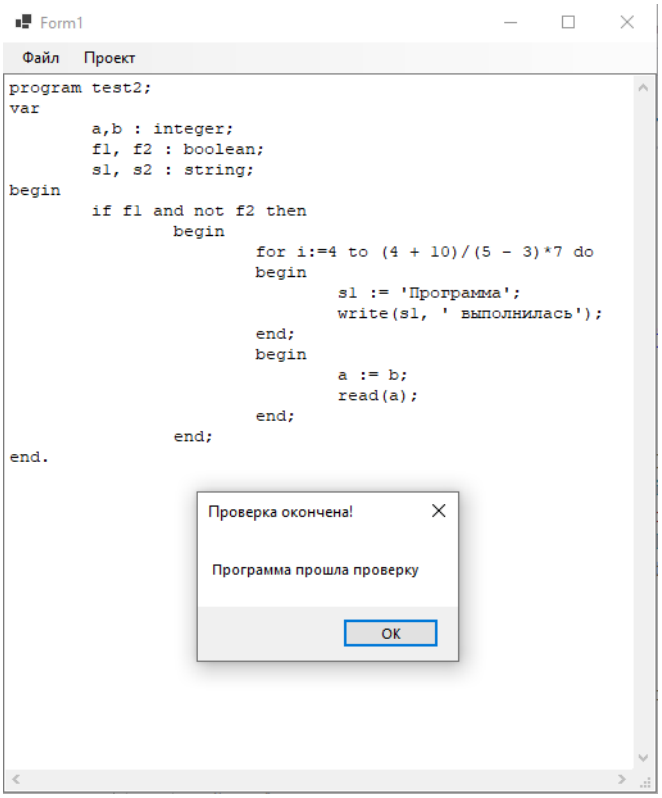
Часть 3

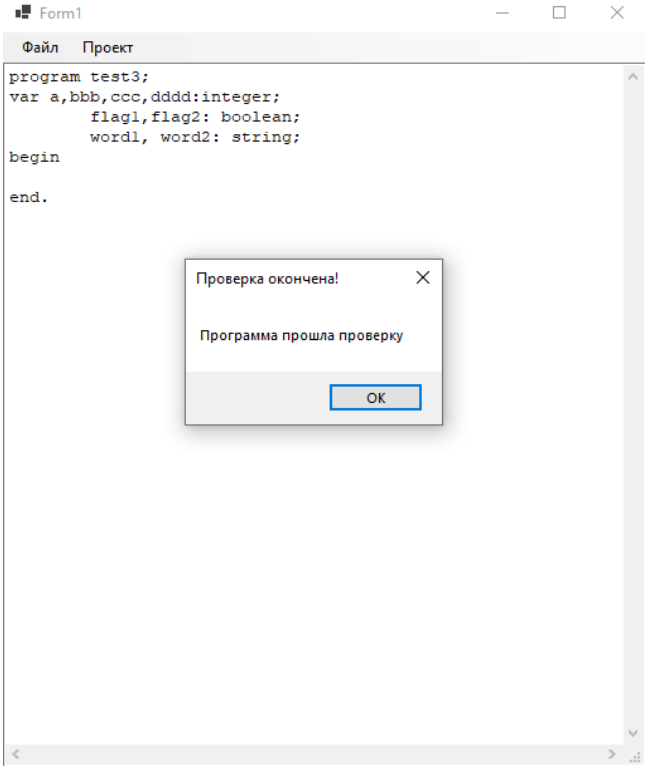
Программа создана на языке C# с использованием WinForms.

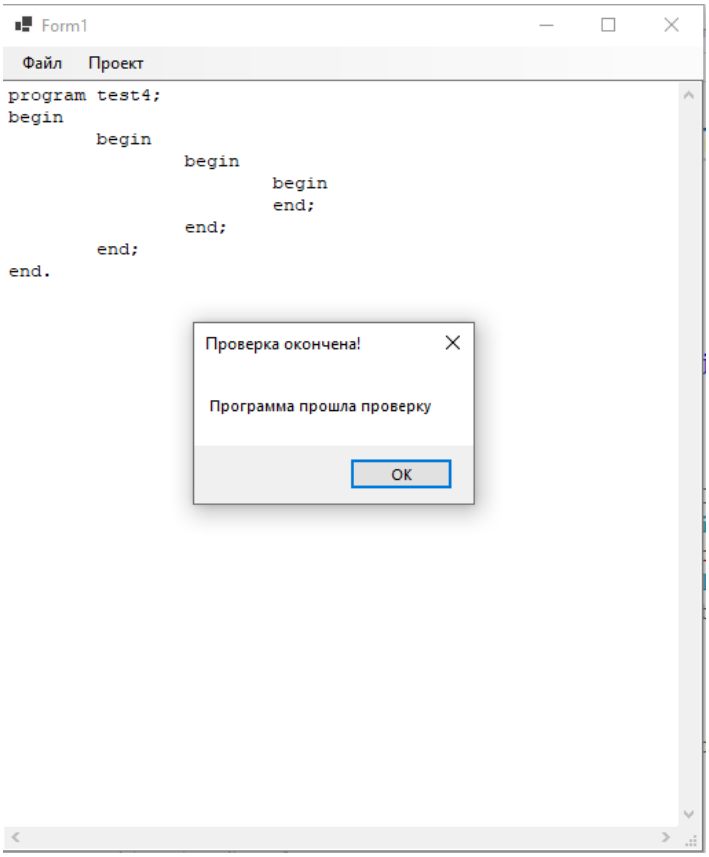
Тестирование:

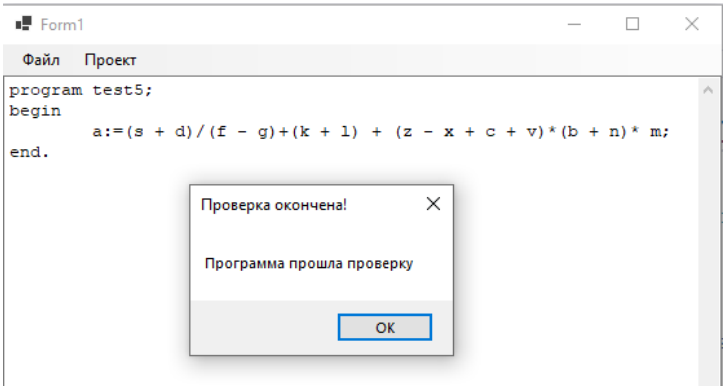


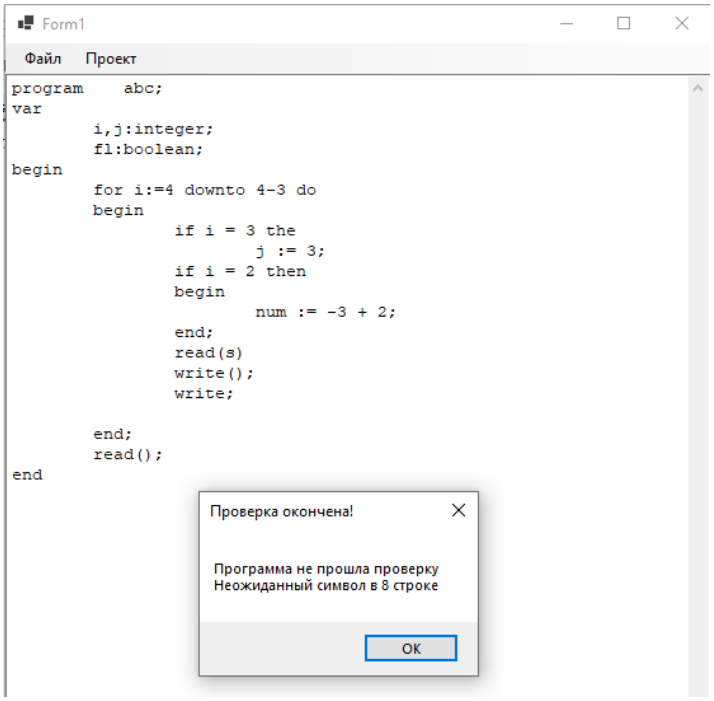


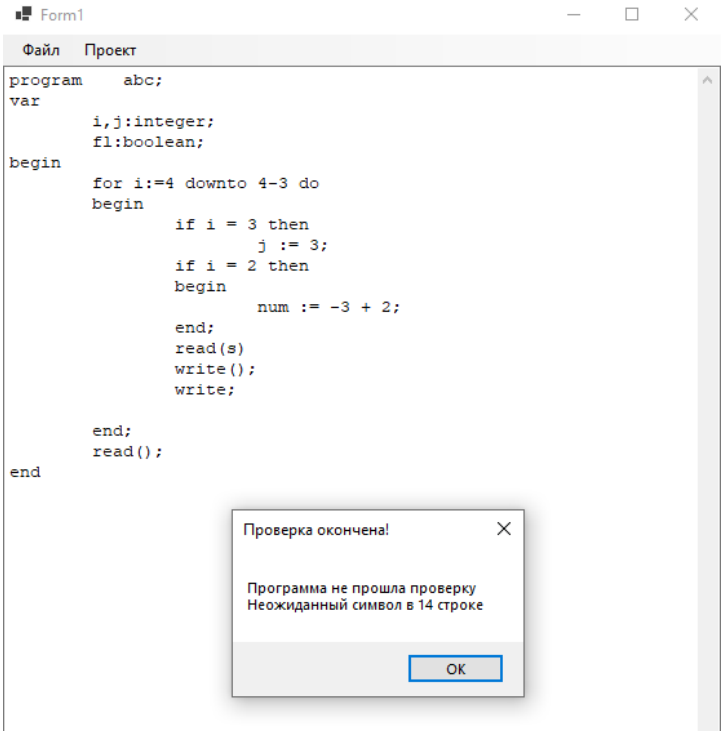












Класс Parser, выполняющий основную работу анализатора:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Lab_2._3 {
    class Parser {
```

```
        String codeBeingChecked = "";
        const String PROGRAM = "program";
        const String VAR = "var";
        const String BEGIN = "begin";
        const String END = "end";
        const String INTEGER = "integer";
        const String BOOLEAN = "boolean";
        const String STRING = "string";
        const String IF = "if";
        const String THEN = "then";
        const String ELSE = "else";
        const String READ = "read";
        const String WRITE = "write";
        const String FOR = "for";
        const String TO = "to";
        const String DOWNTO = "downto";
        const String DO = "do";
        const String OR = "or";
        const String AND = "and";
        const String NOT = "not";
        const String TRUE = "true";
        const String FALSE = "false";

        SortedSet<char> setSeparator = new SortedSet<char>(){' ','\t','\r','\n'};
        SortedSet<String> setRelations = new SortedSet<String>(){"<",">","<>","<=",">=","="};

        bool result = false;
        int mainIndex = 0;

        int indexError = 1;
        public int ErrorIndex { get { return indexError; } }
        int fixIndex = 0;

        public int indexOfNowStartLine = 0;
        public int indexOfNowEndLine = 0;

        public bool CheckProgramText(String textProgram) {
            codeBeingChecked = textProgram;
            result = prog();
            if (!result) {
                indexOfNowStartLine = indexOfNowEndLine;
                indexOfNowEndLine = Math.Max(fixIndex, mainIndex);
            }
            else {
                indexOfNowStartLine = 0;
                indexOfNowEndLine = 0;
            }
            return result;
        }

        bool nextToken() {
            if (mainIndex == 0) {
                if (codeBeingChecked[mainIndex] == '\n') {
                    indexError++;
                }
            }
            if (mainIndex < codeBeingChecked.Length) {
```

```
                mainIndex++;
                if (mainIndex != codeBeingChecked.Length) {
                    if (codeBeingChecked[mainIndex] == '\n') {
                        if (mainIndex > fixIndex && mainIndex != indexOfNowEndLine) {
                            indexError++;
                            indexOfNowStartLine = indexOfNowEndLine;
                            indexOfNowEndLine = mainIndex;
                        }
                    }
                }
                return true;
            }

            return false;
        }

        bool nextTokenAndSkipSeparators() {
            bool fl = nextToken();
            if (fl) {
                while (mainIndex < codeBeingChecked.Length &&
(setSeparator.Contains(codeBeingChecked[mainIndex]))) {
                    mainIndex++;
                    if (codeBeingChecked[mainIndex] == '\n') {
                        if (mainIndex > fixIndex && mainIndex != indexOfNowEndLine) {
                            indexError++;
                            indexOfNowStartLine = indexOfNowEndLine;
                            indexOfNowEndLine = mainIndex;
                        }
                    }
                }
            }
            return fl;
        }

        char getToken(bool skipSeparator = false) {
            if (mainIndex == codeBeingChecked.Length)
                return '\0';
            if (skipSeparator && (setSeparator.Contains(codeBeingChecked[mainIndex]))) {
                nextTokenAndSkipSeparators();
            }
            return codeBeingChecked[mainIndex];

        }

        bool setIndex(int toChange) {
            if (toChange < codeBeingChecked.Length) {
                if (mainIndex >= fixIndex) {
                    fixIndex = mainIndex;
                }
                mainIndex = toChange;
                return true;
            }

            return false;
        }

        bool terminal(String word, bool skipSeparator = false) {
```

```
            if (skipSeparator) {
                getToken(true);
            }
            foreach (var i in word) {
                if (i == getToken() && nextToken()) {
                }
                else {
                    return false;
                }
            }
            return true;
        }

        bool prog() {
            return headProgram()
                && getToken(true) == ';'
                && nextToken()
                && block()
                && last();
        }

        bool last() {
            int index = mainIndex;
            if (getToken(true) == '.') {
                nextToken();
                while (mainIndex < codeBeingChecked.Length &&
(setSeparator.Contains(codeBeingChecked[mainIndex]))) {
                    mainIndex++;
                }
                if (mainIndex == codeBeingChecked.Length)
                    return true;
                return false;
            }
            return false;
        }

        bool headProgram() {
            return terminal(PROGRAM, true) && nextTokenAndSkipSeparators() && identifier();
        }

        bool block() {
            int index = mainIndex;
            return (descriptionsSect() && operatorsSect()) || (setIndex(index) &&
operatorsSect());
        }

        bool descriptionsSect() {
            return varsSect();
        }

        bool varsSect() {
            if (terminal(VAR, true) && nextTokenAndSkipSeparators() && descriptionVars() &&
getToken(true) == ';' && nextToken()) {
                int index = mainIndex;
                while (descriptionVars() && getToken(true) == ';' && nextToken()) {
                    index = mainIndex;
                }
```

```
                mainIndex = index;
                return true;
            }

            return false;
        }

        bool descriptionVars() {
            return listVarsNames() && getToken(true) == ':' && nextToken() && types();
        }

        bool types() {
            getToken(true);
            int index = mainIndex;
            if (terminal(INTEGER)
                    || (setIndex(index) && terminal(BOOLEAN))
                    || (setIndex(index) && terminal(STRING))) {
                return true;
            }
            return false;
        }

        bool listVarsNames() {
            if (identifier()) {
                int index = mainIndex;
                while (getToken(true) == ',' && nextToken() && identifier()) {
                    index = mainIndex;
                }
                mainIndex = index;
                return true;
            }
            return false;
        }

        bool identifier() {
            int index = mainIndex;
            int dopIndex;
            if (Char.IsLetter(getToken(true)) && nextToken()) {
                while ((Char.IsLetter(getToken()) || Char.IsNumber(getToken())) && nextToken()) {
                }
                dopIndex = mainIndex;
                if (!((setIndex(index) && (terminal(TRUE) || terminal(FALSE))))) {
                    mainIndex = dopIndex;
                    return true;
                }
                else
                    mainIndex = dopIndex;
            }
            return false;
        }

        bool operatorsSect() {
            if (terminal(BEGIN, true) && nextTokenAndSkipSeparators()) {
                int index = mainIndex;
                if ((operatorsList() && terminal(END, true)) || (setIndex(index) &&
terminal(END))) {
                    return true;
```

```
                }
            }

            return false;
        }

        bool operator_() {
            int index = mainIndex;
            if (operatorInputOutput() || (setIndex(index) && operatorIf()) || (setIndex(index) &&
operatorFor()) ||
                (setIndex(index) && operatorAssignment()) || (setIndex(index) &&
operatorCompound())) {
                return true;
            }
            return false;
        }

        bool lstWrite() {
            int index = mainIndex;
            if (setIndex(index) && (expres1() || getToken() == ')')) {
                index = mainIndex;
                char parse = getToken(true);
                if (parse != ')') {
                    if (parse == ',') {
                        nextToken();
                        while (parse == ',' && expres1()) {
                            index = mainIndex;
                            parse = getToken(true);
                            nextToken();
                        }
                        if (parse != ')')
                            return false;
                        return true;
                    }
                    if (expres1())
                        return true;
                    return false;
                }
                nextToken();
                return true;
            }
            else
                return false;
        }

        bool operatorInputOutput() {
            getToken(true);
            int index = mainIndex;
            if (terminal(WRITE) && getToken(true) == '(' && nextToken()) {
                index = mainIndex;
                if (getToken(true) == ')' && nextToken() || setIndex(index) && lstWrite()) {
                    return true;
                }
                return false;
            }
            else if (setIndex(index) && terminal(READ) && getToken(true) == '(' && nextToken()) {
                index = mainIndex;
```

```
                if (getToken(true) == ')' && nextToken() || setIndex(index) && listVarsNames() &&
getToken(true) == ')' && nextToken()) {
                    return true;
                }
                return false;
            }
            return false;
        }

        bool operatorIf() {
            if (terminal(IF, true) && boolExpr() && terminal(THEN, true) && operator_()) {
                int index = mainIndex;
                if (terminal(ELSE, true) && nextTokenAndSkipSeparators() && operator_()) {
                    index = mainIndex;
                }
                mainIndex = index;
                return true;
            }
            return false;
        }

        bool operatorFor() {
            if (terminal(FOR, true) && identifier() && terminal(":=", true) && expr()) {
                getToken(true);
                int index = mainIndex;
                if ((terminal(TO) || (setIndex(index) && terminal(DOWNTO))) && expr() &&
terminal(DO, true) && operator_()) {
                    return true;
                }
            }
            return false;
        }

        bool operatorCompound() {
            if (terminal(BEGIN, true) && nextTokenAndSkipSeparators()) {
                int index = mainIndex;
                if ((operatorsList() && terminal(END, true)) || (setIndex(index) &&
terminal(END))) {
                    return true;
                }
            }
            return false;
        }

        bool operatorsList() {
            if (operator_() && getToken() == ';') {
                int index = mainIndex;
                bool fl = true;
                while (fl && getToken(true) == ';' && nextToken()) {
                    index = mainIndex;
                    if (operator_() && getToken() == ';') {
                        index = mainIndex;
                    }
                    else {
                        fl = false;
                    }
                }
```

```
            mainIndex = index;
            return true;
        }
        return false;
    }

    bool operatorAssignment() {
        return var() && terminal(":=", true) && expr();
    }

    bool expres1() {
        int index = mainIndex;
        if (getToken(true) == ')' || getToken(true) == ',')
            return false;
        if ((getToken(true) != '\0' && texting())
                || (setIndex(index) && getToken(true) != '\0' && expr())
                || (setIndex(index) && getToken(true) != '\0' && var())) {
            return true;
        }
        return false;
    }

    bool texting() {
        char st = getToken(true);
        int index = mainIndex;
        if (st == '\'') {
            nextTokenAndSkipSeparators();
            while (Char.IsLetter(getToken()) || Char.IsNumber(getToken())) {
                nextTokenAndSkipSeparators();
            }
            if (getToken() == '\'') {
                nextTokenAndSkipSeparators();
                return true;
            }
        }
        return false;
    }

    bool expr() {
        int index = mainIndex;
        if (formula()
                || (setIndex(index) && boolExpr())
                || (setIndex(index) && texting())) {
            return true;
        }
        return false;
    }

    bool boolExpr() {
        int index = mainIndex;
        return relationship() || (setIndex(index) && simpleLogExpr());
    }

    bool simpleLogExpr() {
        if (boolTerm()) {
            int index = mainIndex;
            if (terminal(OR, true) && simpleLogExpr()) {
```

```
                    index = mainIndex;
                }
                mainIndex = index;
                return true;
            }
            return false;
        }

        bool boolTerm() {
            if (boolMultiplier()) {
                int index = mainIndex;
                if (terminal(AND, true) && boolTerm()) {
                    index = mainIndex;
                }
                mainIndex = index;
                return true;
            }
            return false;
        }

        bool boolMultiplier() {
            getToken(true);
            int index = mainIndex;
            if (boolConst()
                || (setIndex(index) && terminal(NOT) && boolMultiplier())
                || (setIndex(index) && var())
                || (setIndex(index) && getToken(true) == '(' && nextToken() && boolExpr() &&
getToken(true) == ')' && nextToken())
                ) {
                return true;
            }
            return false;
        }

        bool relationship() {
            int index = mainIndex;
            if ((formula() && operatorComparison() && formula())
                || (setIndex(index) && simpleLogExpr() && operatorComparison() &&
simpleLogExpr())) {
                return true;
            }
            return false;
        }

        bool formula() {
            int index = mainIndex;
            if (operatorAddition()) {
                index = mainIndex;
            }
            else {
                mainIndex = index;
            }
            if (term()) {
                index = mainIndex;
                while (operatorAddition() && term()) {
                    index = mainIndex;
                }
```

```
                mainIndex = index;
                return true;
            }
            return false;
        }

        bool term() {
            if (multiplier()) {
                int index = mainIndex;
                while (operatorMultiplication() && multiplier()) {
                    index = mainIndex;
                }
                mainIndex = index;
                return true;
            }
            return false;
        }

        bool multiplier() {
            getToken(true);
            int index = mainIndex;
            if (intNumber()
                || (setIndex(index) && var())
                || (setIndex(index) && getToken(true) == '(' && nextToken() && formula() &&
getToken(true) == ')' && nextToken())) {
                return true;
            }
            return false;
        }

        bool var() {
            return identifier();
        }

        bool intNumber() {
            char firstToken = getToken(true);
            bool isNumber = false;
            if (firstToken == '-' || firstToken == '+') {
                nextToken();
            }
            while (Char.IsNumber(getToken()) && nextToken()) {
                isNumber = true;
            }
            return isNumber;
        }

        bool operatorAddition() {
            getToken(true);
            int index = mainIndex;
            if ((getToken() == '+' && nextToken())
                || (setIndex(index) && (getToken() == '-' && nextToken()))
                ) {
                return true;
            }
            return false;
        }
```

```
        bool operatorMultiplication() {
            getToken(true);
            int index = mainIndex;
            if ((getToken() == '*' && nextToken())
                || (setIndex(index) && (getToken() == '/' && nextToken()))) {
                return true;
            }
            return false;
        }

        bool boolConst() {
            getToken(true);
            int index = mainIndex;
            if (terminal(TRUE)
                || (setIndex(index) && terminal(FALSE))) {
                return true;
            }
            return false;
        }

        bool operatorComparison() {
            getToken(true);
            int index = mainIndex;
            foreach (var i in setRelations) {
                if (setIndex(index) && terminal(i)) {
                    return true;
                }
            }
            return false;
        }
    }
}
```

Проект находится на GitHub по адресу: https://github.com/dilokil/OPT-Lab2