

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

Кафедра автоматизированных систем управления



ПОЯСНИТЕЛЬНАЯ ЗАПИСКА К КУРСОВОЙ РАБОТЕ

«Логические выражения языка C++»

дисциплина: «Теория формальных языков и компиляторов»

Выполнил:

Студент гр. АВТ-912, АВТФ

Мазуров А. В.

«__» _____ 20__ г.

(подпись)

Проверил:

д.т.н., профессор

Шорников Ю. В.

«__» _____ 20__ г.

(подпись)

Новосибирск 2022

СОДЕРЖАНИЕ

1 ПОСТАНОВКА ЗАДАЧИ	3
2 ПОРОЖДАЮЩАЯ ГРАММАТИКА.....	4
3 КЛАССИФИКАЦИЯ ГРАММАТИКИ	5
4 МЕТОД АНАЛИЗА.....	6
5 ДИАГНОСТИКА И НЕЙТРАЛИЗАЦИЯ ОШИБОК	7
6 ТЕСТИРОВАНИЕ ПРОГРАММЫ	8
7 ЛИСТИНГ	11
8 ЛИТЕРАТУРА.....	21

1 ПОСТАНОВКА ЗАДАЧИ

Выполнить программную реализацию алгоритма синтаксического анализа логических выражений языка C++.

Логическое выражение в программировании — конструкция языка программирования, результатом вычисления которой является «истина» или «ложь».

Над логическими выражениями и переменными возможны операции, результатом которых так же являются «истина» и «ложь»:

- дизъюнкция (или, ||);
- конъюнкция (и, &&);
- отрицание (не, !);
- исключающее или (^);
- эквиваленция (==);
- не эквиваленция (! =).

В рамках выполнения курсовой работы предлагается ограничиться двумя логическими операторами: дизъюнкцией (или, ||) и конъюнкцией (и, &&)

Таким образом, логическое выражение будет иметь следующий синтаксис:

$$id1 < \text{Оператор} > id2 \{ < \text{Оператор} > id3 \};$$

где id1, id2, id3 – идентификаторы, начинающиеся с буквы и состоящий из букв и(или) цифр,

<Оператор> - логический оператор, соответствующий дизъюнкции (или, ||) и конъюнкции (и, &&).

2 ПОРОЖДАЮЩАЯ ГРАММАТИКА

Порождающая грамматика логических выражений C++ имеет следующий вид:

$$G[< \text{ЛВ} >] = \{V_T, V_N, < \text{ЛВ} >, P\}$$

где

Множество правил вывода P :

$$1. < \text{ЛВ} > \rightarrow < \text{ИД} > < \text{ЛО} > < \text{ИД} > \{ < \text{ЛО} > < \text{ИД} > \}';'$$

$$2. < \text{ИД} > \rightarrow < \text{Б} > \{ < \text{Б} > \mid < \text{Ц} > \}$$

$$3. < \text{ЛО} > \rightarrow "||" \mid "&&"$$

$$4. < \text{Ц} > \rightarrow '0' \mid '1' \mid \dots \mid '9'$$

$$5. < \text{Б} > \rightarrow 'a' \mid 'b' \mid \dots \mid 'z' \mid 'A' \mid 'B' \mid \dots \mid 'Z'$$

Начальный нетерминал:

$$Z = < \text{ЛВ} >$$

Множество терминальных символов:

$$V_T = \{ 'a', 'b', \dots, 'z', 'A', 'B', \dots, 'Z', '0', '1', \dots, '9', ';', ' ', "||", "&&" \}$$

Множество нетерминальных символов:

$$V_N = \{ < \text{ЛВ} >, < \text{ЛО} >, < \text{Б} >, < \text{Ц} > \}$$

Условные обозначения:

- ЛВ – логическое выражение;
- ЛО – логический оператор;
- Б – буква;
- Ц – цифра.

3 КЛАССИФИКАЦИЯ ГРАММАТИКИ

Грамматика $G[Z]$ по классификации Хомского относится к контекстно-свободной и имеет вид:

$$A \rightarrow \alpha, \text{ где } A \in V_n, \alpha \in V^* [1].$$

В левой части допускаются только нетерминальные символы, а в правой в части могут присутствовать символы как терминального, так и нетерминального словарей.

Диаграмма состояний сканера лексем представлена на Рисунке 1:

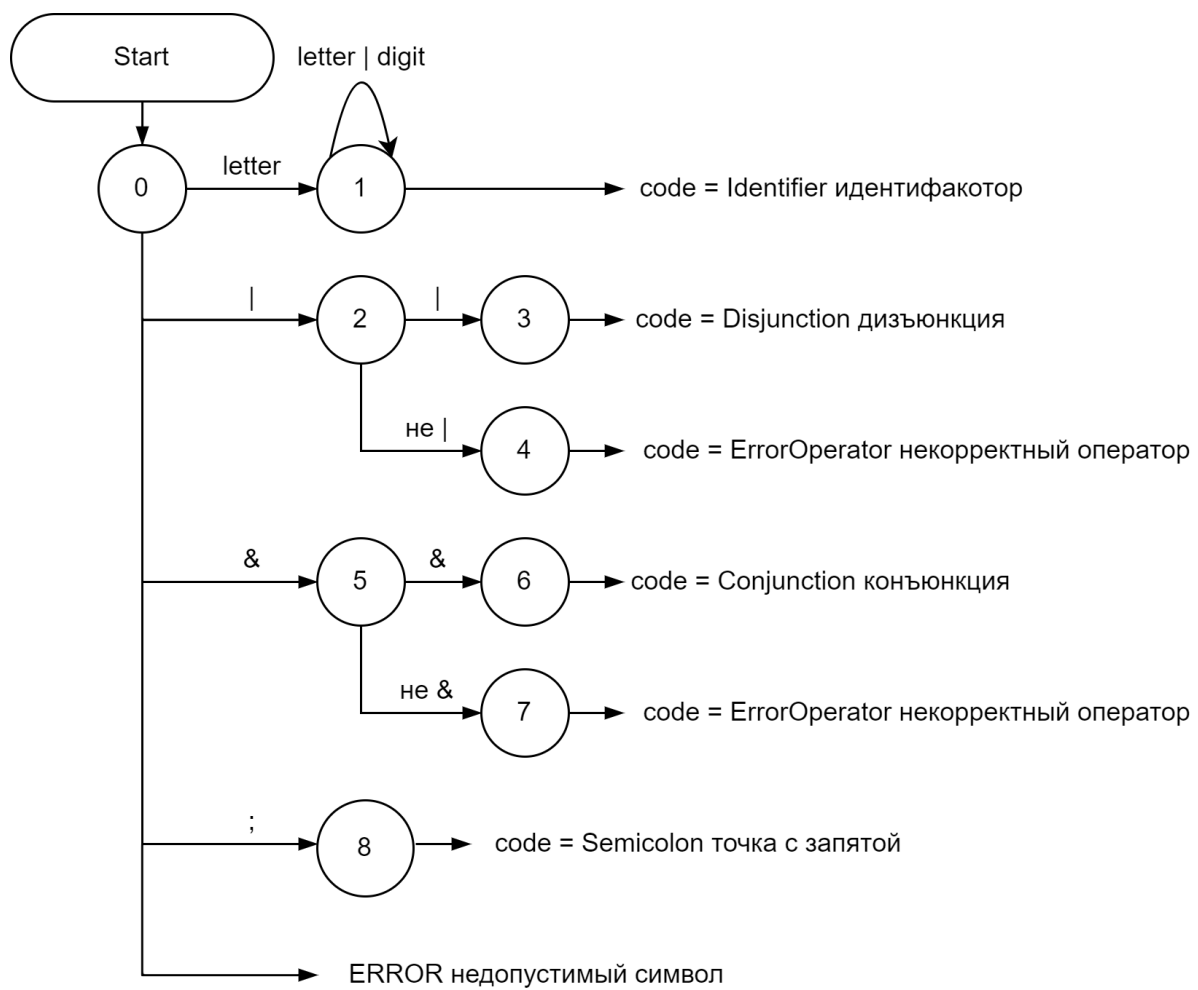


Рисунок 1 – диаграмма состояний сканера

4 МЕТОД АНАЛИЗА

Для грамматики $G[Z]$ был выбран метод рекурсивного спуска, так как он подходит для контекстно-свободных грамматик.

Основная идея метода рекурсивного спуска заключается в следующем: каждому нетерминальному символу грамматики ставится в соответствие своя функция, процедура или другая программная единица, задача которой – начиная с указанного места исходной цепочки найти подцепочку, которая выводится из этого нетерминала. Тело каждой такой функции задаётся в соответствии с правилами вывода, соответствующего нетерминала: терминалы из правой части распознаются самой функцией, а нетерминалы соответствуют вызовам функций. Функции могут вызывать сами себя [1].

Для решения данной задачи выбран язык программирования C#.

5 ДИАГНОСТИКА И НЕЙТРАЛИЗАЦИЯ ОШИБОК

Для данной грамматики производится диагностика и нейтрализация ошибок с их возможным исправлением. Нейтрализация ошибок осуществляется по методу Айронса: спускаясь по синтаксическому дереву без возврата по контексту, при обнаружении тупиковой ситуации отбрасываются те литеры (символы), которые привели в тупиковую ситуацию. На их место ставится типовое значение с целью дальнейшего корректного разбора.

6 ТЕСТИРОВАНИЕ ПРОГРАММЫ

Результаты тестирования программы приведены далее на Рисунках 2–7:

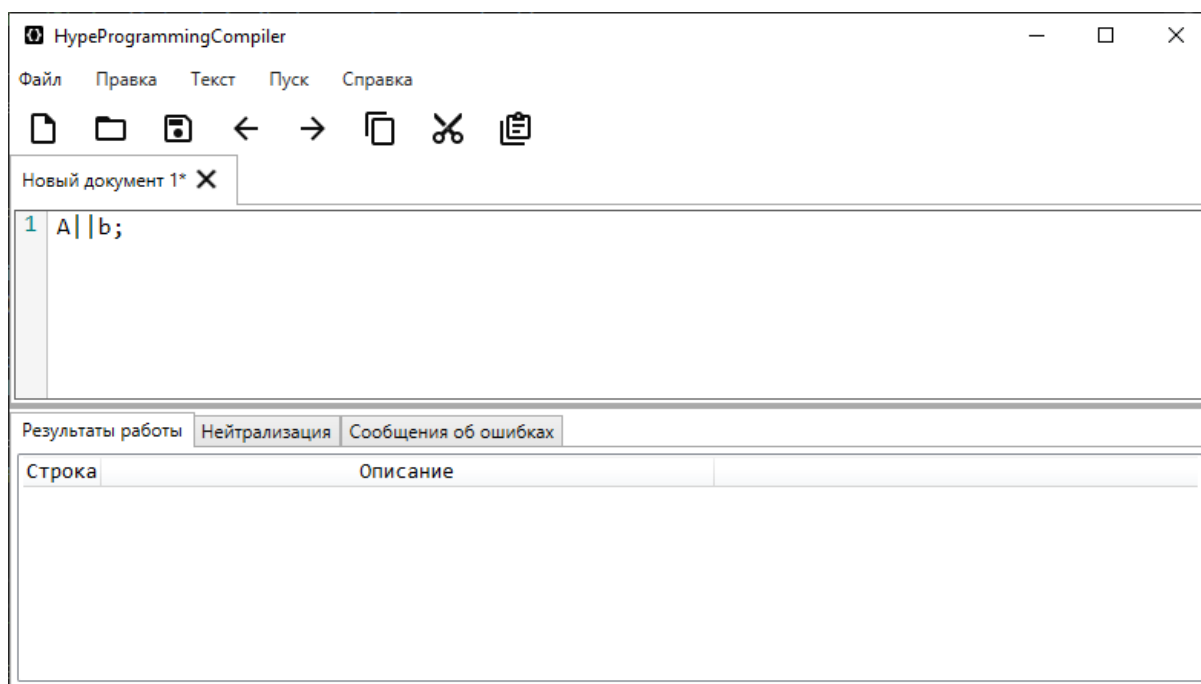


Рисунок 2 – ввод корректного логического выражения

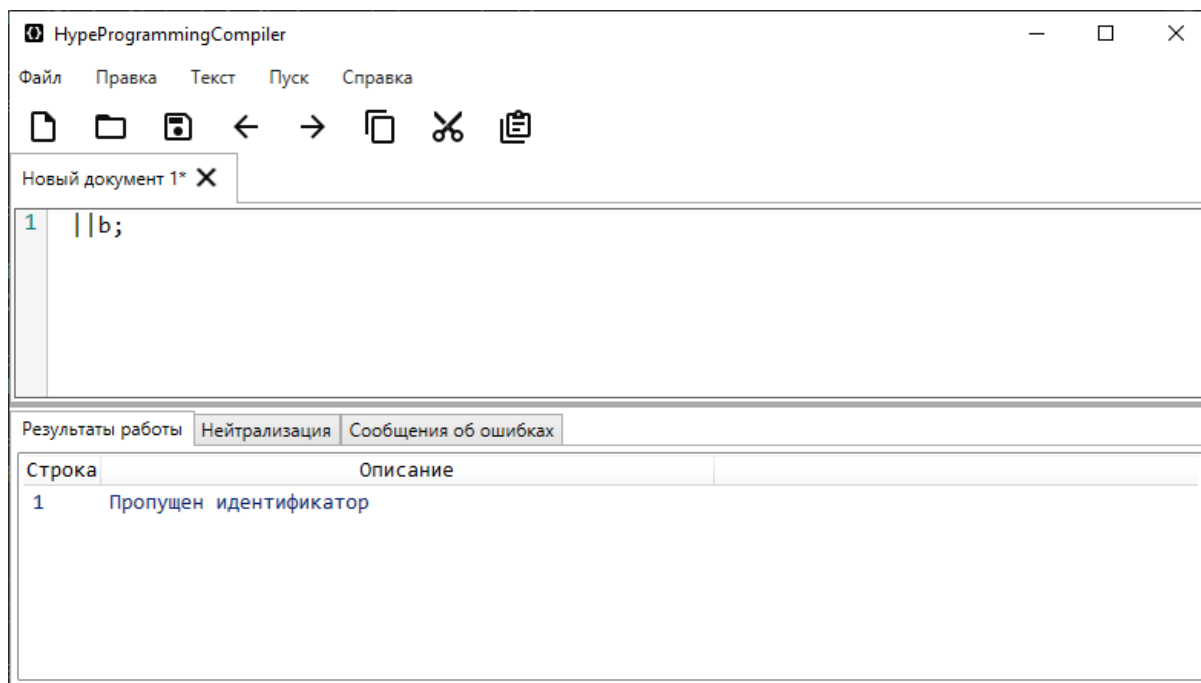


Рисунок 3 – пропуск идентификатора в логическом выражении

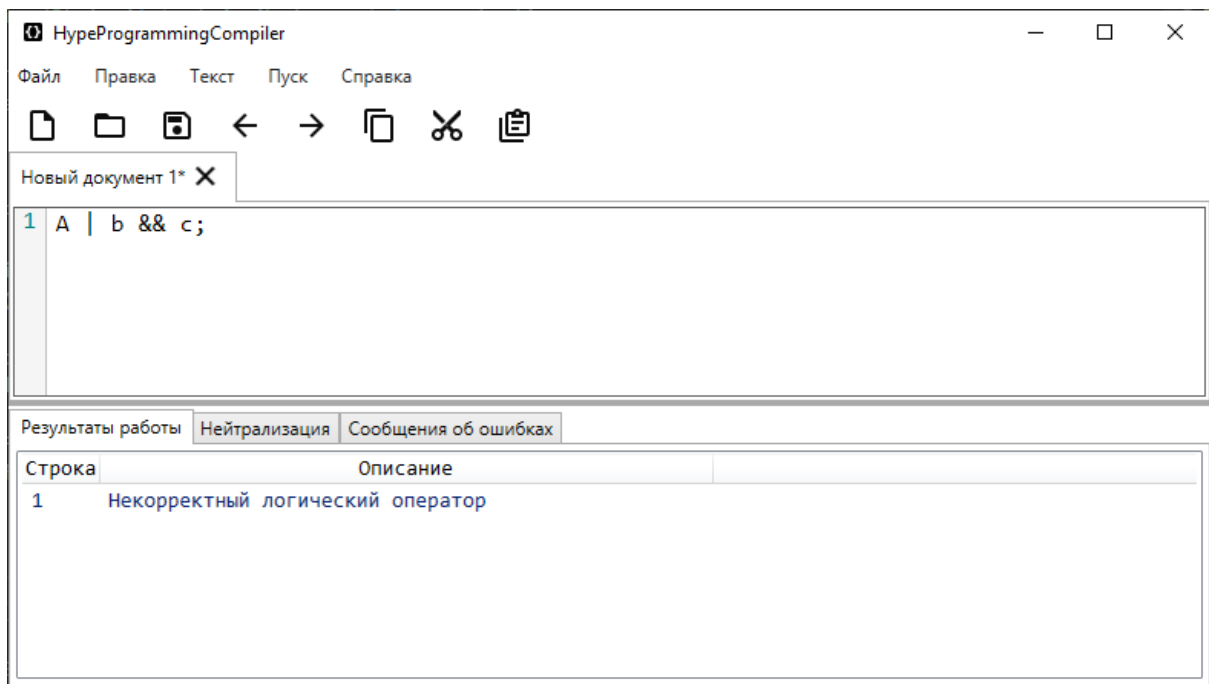


Рисунок 4 – ввод некорректного логического оператора

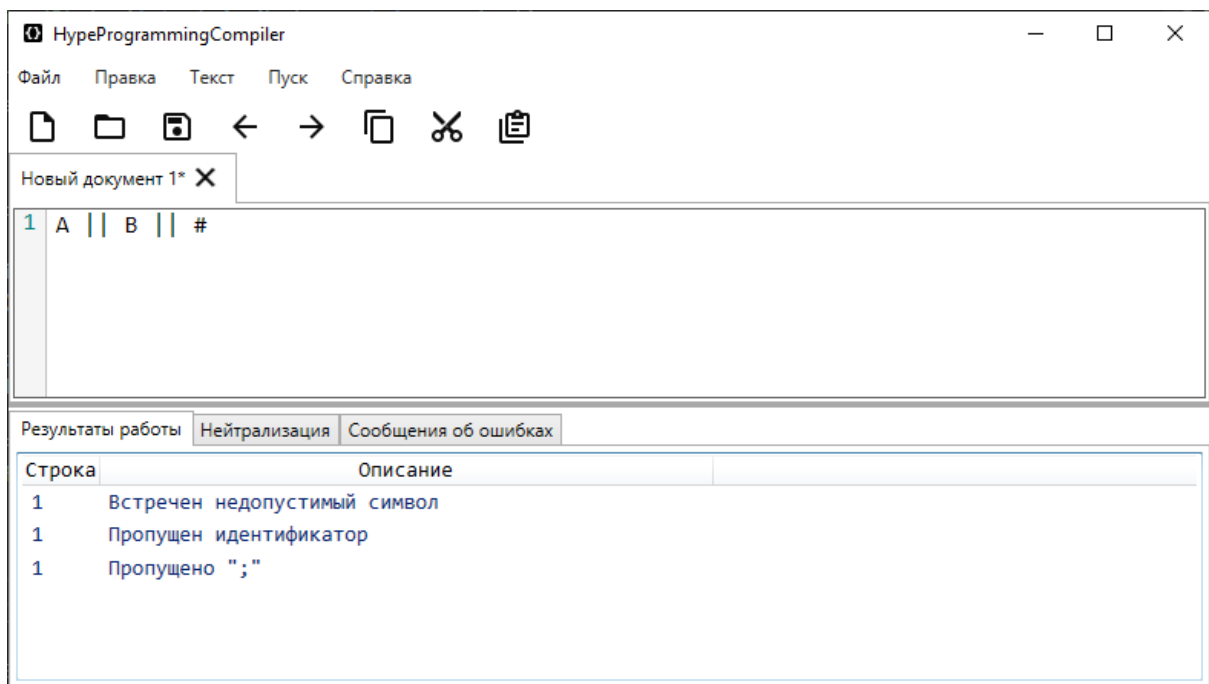


Рисунок 5 – пропуск идентификатора и точки с запятой, ввод недопустимого символа '#'

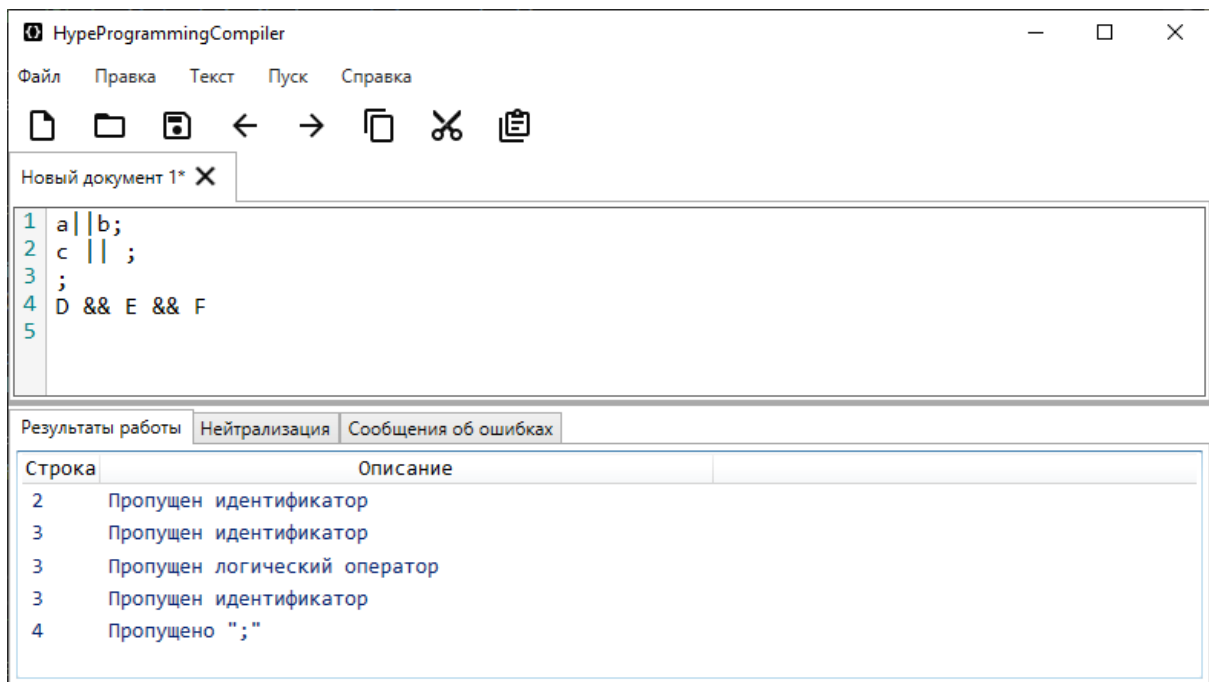


Рисунок 6 – ввод нескольких строк с логическими выражениями

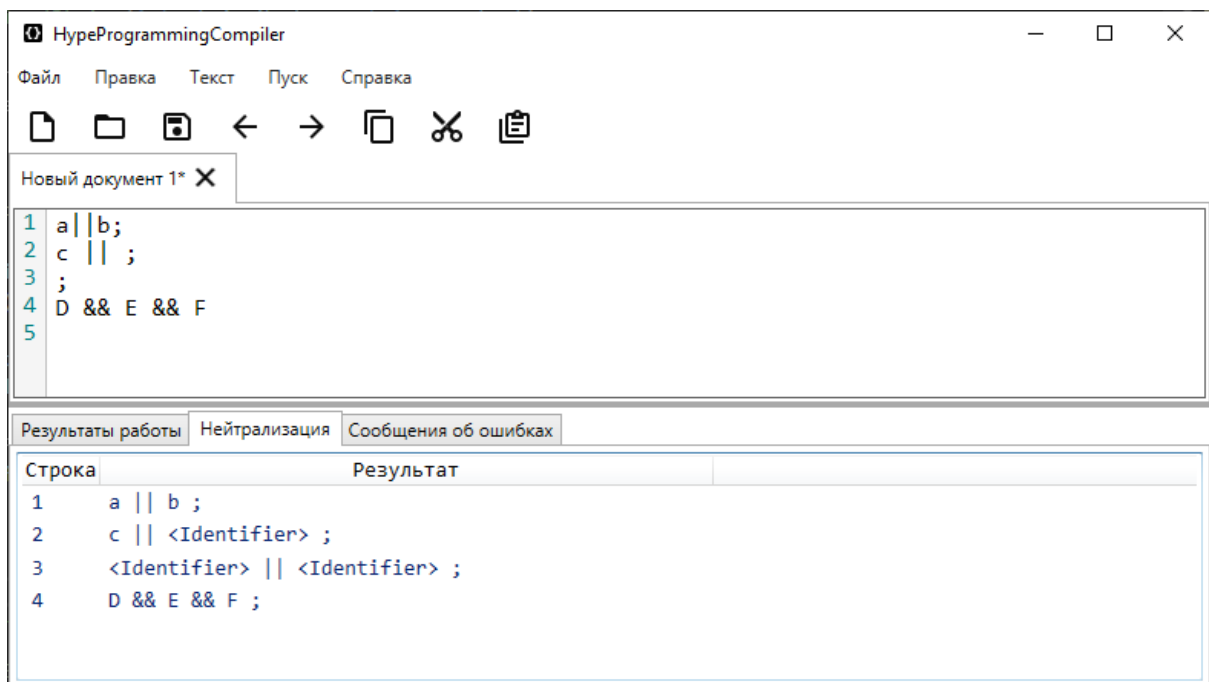


Рисунок 7 – Результат нейтрализации с исправлением ошибок для тех же выражений (см. Рисунок 6)

7 ЛИСТИНГ

Lexem.cs

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

namespace HypeProgrammingCompiler
{
    public enum LexemType // Перечень типов лексем
    {
        ErrorOperator = -1,
        ErrorToken = 0,
        Identifier = 1,
        Disjunction = 2,
        Conjunction = 3,
        Semicolon = 4
    }

    // Предоставляет хранение свойств токена и доступ к ним
    public class Lexem
    {
        public LexemType Type { get; set; }
        public string Symbol { get; set; }
        public int StringNumber { get; set; }
        public int StartPosition { get; set; }
        public int EndPosition { get; set; }

        public Lexem(LexemType type, string symbol, int stringNumber, int
startPosition, int endPosition)
        {
            Type = type;
            Symbol = symbol;
            StringNumber = stringNumber;
            StartPosition = startPosition;
            EndPosition = endPosition;
        }
    }
}
```

LexemList.cs

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

namespace HypeProgrammingCompiler
{
    // Хранит лист токенов
    class LexemList
    {
        public List<Lexem> lexems = new List<Lexem>();
        private int i = 0;

        public LexemList() { }
        public LexemList(List<Lexem> lexems) { this.lexems = lexems; }

        public Lexem Current // Возврат лексемы на которую указывает индекс
        {
            get
            {
                if (i >= lexems.Count)
                {
                    return new Lexem(0, null, lexems[i - 1].StringNumber, 0,
0);
                }
                return lexems[i];
            }
        }

        public bool Next() // Перемещение индекса вперёд
        {
            i++;
            if (i < lexems.Count)
            {
                return true;
            }
            else
                return false;
        }

        public Lexem Prev // Возврат предыдущей лексемы от текущего индекса
        {
            get
            {
                if (i > 0)
                    return lexems[i - 1];
            }
        }
    }
}
```

```

        else
            return new Lexem(0, null, 0, 0, 0);
    }
}

public void Add(Lexem lexem) // Вставка в конец
{
    lexems.Add(lexem);
}

public int Count // Возврат числа лексем в списке
{
    get
    {
        return lexems.Count;
    }
}

public void RemoveNext(Lexem lexem) // Удаление лексемы следующей за
текущей
{
    lexems.RemoveAt(lexems.IndexOf(lexem) + 1);
}

public void Insert(Lexem lexem) // Вставка лексемы по текущему индексу
{
    lexems.Insert(i , lexem);
}
}
}

```

Lexer.cs

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

namespace HypeProgrammingCompiler
{
    class Lexer
    {
        public LexemList lexemList = new LexemList(); // Список лексем для
        // заполнения
        private List<string> textStrings = new List<string>();
        private int i = 0; // Индекс символа
        private int stringNumber = 0; // Номер строки

        public Lexer(string text)
        {
            textStrings = text.Split("\n").ToList();
            textStrings[textStrings.Count - 1] += "\r";
        }

        public string Print()
        {
            string result = "";

            do
            {
                result += lexemList.Current.Type + " - ";
                result += lexemList.Current.Symbol + " - ";
                result += "S: " + (lexemList.Current.StringNumber) + " - ";
                result += "P: " + lexemList.Current.StartPosition + "-";
                result += lexemList.Current.EndPosition + ";\n";
            }
            while (lexemList.Next());

            return result;
        }

        public void Analyze()
        {
            foreach (string textString in textStrings)
            {
                AnalyzeString(textString);
                stringNumber++;
            }
        }

        private void AnalyzeString(string textString)
        {
            i = 0;
            for (; i < textString.Length; i++)
            {
                // Пропуск пробелов перед лексемой
            }
        }
    }
}
```

```

        while (textString[i] == ' ') i++;

        // Если лексема начинается с буквы, то парсим идентификатор
        if (char.IsLetter(textString[i]))
        {
            MatchIdentifier(textString);
            if (textString.Length <= i)
                break;
        }
        // Иначе парсим дизъюнкцию
        else if (textString[i] == '|')
        {
            MatchDisjunction(textString);
            if (textString.Length <= i)
                break;
        }
        // Иначе парсим конъюнкцию
        else if (textString[i] == '&')
        {
            MatchConjunction(textString);
            if (textString.Length <= i)
                break;
        }
        // Иначе парсим точку с запятой
        else if (textString[i] == ';')
        {
            MatchSemicolon();
            if (textString.Length <= i)
                break;
        }
        // Иначе парсим неизвестный символ
        else
        {
            if (textString[i] != '\r')
                MatchErrorToken(textString);
        }
    }
}

private void MatchIdentifier(string textString)
{
    string identifier = "";
    int startPosition = i + 1;

    while (char.IsLetterOrDigit(textString[i]))
    {
        identifier += textString[i];
        i++;

        if (textString.Length <= i)
            break;
    }
    int endPosition = i;

```

```

        lexemList.Add(new Lexem(LexemType.Identifier, identifier, string-
Number + 1, startPosition, endPosition));
        i--;
    }

    private void MatchDisjunction(string textString)
    {
        if (textString.Length != i + 1)
        {
            if (textString[i + 1] == '|')
            {
                lexemList.Add(new (LexemType.Disjunction, "||", string-
Number + 1, i + 1, i + 2));
                i++;
                return;
            }
        }
        // Парсим некорректный оператор
        lexemList.Add(new (LexemType.ErrorOperator,
textString[i].ToString(), stringNumber + 1, i + 1, i + 1));
    }

    private void MatchConjunction(string textString)
    {
        if (textString.Length != i + 1)
        {
            if (textString[i + 1] == '&')
            {
                lexemList.Add(new (LexemType.Disjunction, "&&", string-
Number + 1, i + 1, i + 2));
                i++;
                return;
            }
        }
        // Парсим некорректный оператор
        lexemList.Add(new (LexemType.ErrorOperator,
textString[i].ToString(), stringNumber + 1, i + 1, i + 1));
    }

    private void MatchSemicolon()
    {
        lexemList.Add(new (LexemType.Semicolon, ";", stringNumber + 1, i +
1, i + 1));
    }

    private void MatchErrorToken(string textString)
    {
        lexemList.Add(new (LexemType.ErrorToken, textString[i].ToString(),
stringNumber + 1, i + 1, i + 1));
    }
}

```


Parser.cs

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

namespace HypeProgrammingCompiler
{
    class Parser
    {
        // Исходный текст
        private string text;
        // Список ошибок для вывода и нейтрализации
        public ObservableCollection<Error> errorList = new ObservableColle-
tion<Error>();
        // Список лексем - результат декомпозиции текста
        private LexemList lexemList = new LexemList();
        // Исправленные строки
        public List<FixedString> FixedStrings = new List<FixedString>();
        public struct FixedString
        {
            public int StringNumber { get; set; }
            public string Content { get; set; }

            public FixedString(int stringNumber, string content)
            {
                StringNumber = stringNumber;
                Content = content;
            }
        }
        public void PrepareFixedStrings()
        {
            string fixedString = "";
            foreach (var lexem in lexemList.lexems)
            {
                fixedString += lexem.Symbol + " ";
                if (lexem.Symbol == ";"")
                {
                    FixedStrings.Add(new FixedString(lexem.StringNumber,
fixedString));
                    fixedString = "";
                }
            }
        }
        // Перечисление кодов ошибок
        public enum ErrorCode
        {
            NoIdentifier = 1,
            NoOperator = 2,
            NoSemicolon = 3,
            InvalidOperator = 4,
            InvalidCharacter = 5
        }
    }
}
```

```

//Класс Error - содержит информацию об ошибке и её код
public class Error
{
    public string Info { get; set; }
    public ErrorCode Code { get; set; }
    public string Symbol { get; set; }
    public int StringNumber { get; set; }
    public int StartPosition { get; set; }
    public int EndPosition { get; set; }

    public Error(string info, ErrorCode code, string symbol, int
stringNumber, int startPosition, int endPoition)
    {
        Info = info;
        Code = code;
        Symbol = symbol;
        StringNumber = stringNumber;
        StartPosition = startPosition;
        EndPosition = endPoition;
    }
    public Error(string info, ErrorCode code, int stringNumber, int
startPosition, int endPoition)
    {
        Info = info;
        Code = code;
        StringNumber = stringNumber;
        StartPosition = startPosition;
        EndPosition = endPoition;
    }
}

public Parser(string text)
{
    this.text = text;
    errorList.CollectionChanged += ErrorList_CollectionChanged;
}

// Нейтрализация ошибок при их обнаружении
private void ErrorList_CollectionChanged(object sender, System Collec-
tions.Specialized.NotifyCollectionChangedEventArgs e)
{
    Error error = e.NewItems[0] as Error;

    switch (error.Code)
    {
        case ErrorCode.NoIdentifier:
            lexemList.Insert(new Lexem(LexemType.Identifier, "<Identi-
fier>", lexemList.Current.StringNumber, lexemList.Current.StartPosition, 0));
            break;
        case ErrorCode.NoOperator:
            lexemList.Insert(new Lexem(LexemType.Conjunction, "||",
lexemList.Current.StringNumber, lexemList.Current.StartPosition, 0));
            break;
        case ErrorCode.NoSemicolon:

```

```

        lexemList.Insert(new Lexem(LexemType.Semicolon, ";", lex-
emList.Current.StringNumber, lexemList.Current.StartPosition, 0));
        break;
        case ErrorCode.InvalidOperator:
            if (lexemList.Current.Symbol == "|")
                lexemList.Insert(new Lexem(LexemType.Conjunction,
"||", lexemList.Current.StringNumber, lexemList.Current.StartPosition, 0));
            else
                lexemList.Insert(new Lexem(LexemType.Conjunction,
"&&", lexemList.Current.StringNumber, lexemList.Current.StartPosition, 0));

            lexemList.RemoveNext(lexemList.Current);
            break;
    }
}

private bool IsErrorLexem(Lexem lexem) // Предикат определения
недопустимых лексем и их удаления
{
    if (lexem.Type == LexemType.ErrorToken)
    {
        errorList.Add(new Error("Встречен недопустимый символ", Error-
Code.InvalidCharacter, lexem.Symbol, lexem.StringNumber, lexem.StartPosition,
lexem.EndPosition));
        return true;
    }
    return false;
}

public void Parse()
{
    Lexer lexer = new Lexer(text);
    lexer.Analyze(); // Декомпозиция текста на лексемы
    lexemList = lexer.lexemList;

    // Нейтрализация недопустимых символов
    lexemList.lexems.RemoveAll(IsErrorLexem);

    if (lexemList.Count > 0)
    {
        do
        {
            // Если пропущен идентификатор
            if (lexemList.Current.Type != LexemType.Identifier)
                errorList.Add(new Error("Пропущен идентификатор", Er-
rorCode.NoIdentifier, lexemList.Current.StringNumber, lexemList.Cur-
rent.StartPosition, lexemList.Current.EndPosition));

            lexemList.Next();
            // Если оператор пропущен или некорректен
            if (lexemList.Current.Type != LexemType.Disjunction &&
                lexemList.Current.Type != LexemType.Conjunction)
            {
                if (lexemList.Current.Type == LexemType.ErrorOperator)

```

```

        errorList.Add(new Error("Некорректный логический
оператор", ErrorCode.InvalidOperator, lexemList.Current.Symbol, lexemList.Current.StringNumber, lexemList.Current.StartPosition, lexemList.Current.EndPosition));

        else
            errorList.Add(new Error("Пропущен логический
оператор", ErrorCode.NoOperator, lexemList.Prev.StringNumber, lexemList.Current.StartPosition, lexemList.Current.EndPosition));
    }

    lexemList.Next();
    //Если пропущен идентификатор
    if (lexemList.Current.Type != LexemType.Identifier)
        errorList.Add(new Error("Пропущен идентификатор", ErrorCode.NoIdentifier, lexemList.Current.StringNumber, lexemList.Current.StartPosition, lexemList.Current.EndPosition));

    // Итерация * (Замыкание Клини)
    while (lexemList.Next() && lexemList.Current.Type != LexemType.Semicolon)
    {
        if (lexemList.Current.Type != LexemType.Disjunction &&
            lexemList.Current.Type != LexemType.Conjunction)
        {
            if (lexemList.Current.Type == LexemType.ErrorOperator)
            {
                errorList.Add(new Error("Некорректный
логический оператор", ErrorCode.InvalidOperator, lexemList.Current.Symbol, lexemList.Current.StringNumber, lexemList.Current.StartPosition, lexemList.Current.EndPosition));
            }
            else
            {
                errorList.Add(new Error("Пропущен логический
оператор", ErrorCode.NoOperator, lexemList.Prev.StringNumber, lexemList.Current.StartPosition, lexemList.Current.EndPosition));
            }
        }

        lexemList.Next();
        if (lexemList.Current.Type != LexemType.Identifier)
            errorList.Add(new Error("Пропущен идентификатор", ErrorCode.NoIdentifier, lexemList.Current.StringNumber, lexemList.Current.StartPosition, lexemList.Current.EndPosition));
    }

    // Если пропущено ";"
    if (lexemList.Current.Type != LexemType.Semicolon)
        errorList.Add(new Error("Пропущено \";\"", ErrorCode.NoSemicolon, lexemList.Current.StringNumber, lexemList.Current.StartPosition, lexemList.Current.EndPosition));
    }
    while (lexemList.Next()); // Цикл до последней лексемы
    ВКЛЮЧИТЕЛЬНО
}
PrepareFixedStrings();
}
}
}

```

8 ЛИТЕРАТУРА

1. Шорников Ю.В. Теория и практика языковых процессоров: Учеб. пособие. – Новосибирск: Изд-во НГТУ, 2004. – 208 с.
2. Руководство по WPF // METANIT.COM сайт о программировании : [электронный ресурс]. – 2021. – URL: <https://metanit.com/sharp/wpf/> (дата обращения: 14.04.2022).
3. Полное руководство по языку программирования C# 10 и платформе .NET 6 // METANIT.COM сайт о программировании : [электронный ресурс]. – 2022. – URL: <https://metanit.com/sharp/wpf/> (дата обращения: 9.03.2022).
4. Теория формальных языков и компиляторов [электронный ресурс] . – 2021. – URL: <https://dispace.edu.nstu.ru/didesk/course/show/8594> (дата обращения: 04.03.2022)