**1. ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РАБОТУ**

**Задание:**  
С использованием метода рекурсивного спуска реализовать синтаксический анализатор для языка, цепочки которого задаются в лабораторной работе №1. Анализатор должен принимать на вход последовательность лексем, сформированную лексическим анализатором, восстанавливать дерево разбора (логически) и, в случае ошибок, выдавать информативное сообщение с указанием причины и места возникновения ошибки.

Дополнительные условия:

* Обеспечить поддержку конструкции типа

do until <логическое выражение> <операторы> loop

* Реализовать грамматику уровня 3, где логическое выражение имеет следующие альтернативы:

<логическое выражение> → <выражение сравнения>

| <унарная логическая операция> <выражение сравнения>

| <логическое выражение> <бинарная логическая операция> <выражение сравнения>

с последующим определением:

* + <выражение сравнения> → <операнд> | <операнд> <операция сравнения> <операнд>
  + <операция сравнения> → < | > | = | <>
  + <унарная логическая операция> → not
  + <бинарная логическая операция> → and | or
  + <операнд> → <идентификатор> | <константа>
  + <операторы> → <операторы>; <оператор> | <оператор>
  + <оператор> → <идентификатор> = <арифметическое выражение> | output <операнд>
  + <арифметическое выражение> → <операнд> | <операнд> <арифметическая операция> <арифметическое выражение>
  + <арифметическая операция> → + | - | / | \*

**2. ОПИСАНИЕ ГРАММАТИКИ И ЕЁ ПРЕОБРАЗОВАНИЕ**

**2.1 Грамматика исходного языка**

Исходный язык содержит следующие типы лексем, полученные на этапе лексического анализа:

* **Ключевые слова:**  
  do, until, loop, not, and, or, output
* **Идентификаторы:** (например, переменные, обозначаемые как *var*)
* **Константы:** (например, целые числа, обозначаемые как *const*)
* **Операторы:**
  + Оператор присваивания/сравнения: =
  + Операторы сравнения: <, >, <>
  + Арифметические операторы: +, -, \*, /
  + Разделитель: ;

**2.2 Преобразование грамматики**

Для реализации анализа методом рекурсивного спуска исходную грамматику необходимо привести к форме, удобной для написания процедур, то есть к расширенной нотации Бэкуса–Наура (ЭБНФ). Это включает:

1. **Устранение левой рекурсии.**  
   Например, исходное правило для логического выражения могло иметь вид:

<логическое выражение> → <выражение сравнения> | <логическое выражение> <бинарная логическая операция> <выражение сравнения>

Его преобразуют в:

<логическое выражение> → <выражение сравнения> { <бинарная логическая операция> <выражение сравнения> }

1. **Левая факторизация.**  
   Если несколько альтернатив начинаются одинаково, производится левая факторизация. Например, правило

<выражение сравнения> → <операнд> | <операнд> <операция сравнения> <операнд>

преобразуется в:

<выражение сравнения> → <операнд> [ <операция сравнения> <операнд> ]

1. **Запись правил в расширенной БНФ.**  
   Итоговая грамматика в ЭБНФ имеет следующий вид:
   * **Конструкция цикла:**

<DoUntilStatement> → do until <ЛогическоеВыражение> <Операторы> loop

* + **Логическое выражение:**

<ЛогическоеВыражение> → [ not ] <ВыражениеСравнения> { (and | or) [ not ] <ВыражениеСравнения> }

* + **Выражение сравнения:**

<ВыражениеСравнения> → <Операнд> [ (< | > | = | <>) <Операнд> ]

* + **Операнд:**

<Операнд> → var | const

* + **Операторы:**

<Операторы> → <Оператор> { ; <Оператор> }

* + **Оператор:**

<Оператор> → var = <АрифметическоеВыражение> | output <Операнд>

* + **Арифметическое выражение:**

<АрифметическоеВыражение> → <Операнд> { (+ | - | \* | /) <Операнд> }

**3. ОПИСАНИЕ АЛГОРИТМОВ И СТРУКТУР ДАННЫХ**

**3.1 Алгоритмы (процедуры и функции рекурсивного спуска)**

В программе для каждого нетерминального символа грамматики реализована отдельная функция. Например:

* **DoUntilStatement()**  
  Проверяет конструкцию цикла:  
  – Проверяет наличие лексемы do  
  – Проверяет наличие лексемы until  
  – Вызывает функцию LogicalExpression() для разбора логического выражения  
  – Вызывает функцию Operators() для разбора списка операторов  
  – Проверяет наличие лексемы loop  
  При ошибке выводится сообщение с указанием позиции.
* **LogicalExpression()**  
  Разбирает логическое выражение по правилу:

[ not ] <ВыражениеСравнения> { (and | or) [ not ] <ВыражениеСравнения> }

Функция фиксирует вход и выход, ведёт лог текущего токена.

* **ComparisonExpression()** (выражение сравнения)  
  Разбирается правило:

<Операнд> [ (< | > | = | <>) <Операнд> ]

* **Operand()**  
  Проверяет, что текущая лексема является идентификатором (var) или константой (const).
* **ArithExpr()**  
  Анализирует арифметическое выражение по правилу:

<Операнд> { (+ | - | \* | /) <Операнд> }

* **Operators()**  
  Разбирает список операторов, разделённых символом ;.
* **OperatorStmt()**  
  Определяет, является ли оператор присваиванием (например, var = <АрифметическоеВыражение>) или оператором вывода (output <Операнд>).

Каждая функция фиксирует вход и выход (с использованием логирования с отступами) для последующего отображения этапов разбора.

**3.2 Структуры данных**

* **Тип Token и перечисление TokenType.**  
  Для представления лексем используется класс Token, содержащий следующие поля:
  + Type – тип лексемы (например, lDo, lUntil, lLoop, lNot, lAnd, lOr, lOutput, lVar, lConst, lEqual, lLess, lGreater, lNotEqual, lPlus, lMinus, lMul, lDiv, lSemicolon).
  + Lexeme – строковое представление лексемы.
  + Position – позиция лексемы во входном тексте (для вывода сообщений об ошибке).
* **Коллекция токенов.**  
  Последовательность лексем хранится в виде списка (List<Token>), по которому осуществляется переход с помощью глобального указателя (индекса).

**3.3 Функции для работы с последовательностью лексем и вывода синтаксических ошибок**

* Функция Error(string msg, int pos) вызывается при обнаружении несоответствия ожидаемой лексеме и генерирует исключение с сообщением об ошибке и позицией.
* В процессе анализа глобальный указатель (индекс) последовательно перемещается по списку токенов, что позволяет точно определить место возникновения ошибки.

**4. ОПИСАНИЕ ИНТЕРФЕЙСА ПОЛЬЗОВАТЕЛЯ**

Приложение реализовано как WPF‑приложение на C#. Основные элементы интерфейса:

* **TextBox для ввода исходного текста.**  
  Пользователь вводит или редактирует исходный текст программы. По умолчанию в поле уже присутствует пример:

do until not x < 10 and y = 20 x = x + 1; output x loop

* **Кнопка "Разобрать".**  
  При нажатии на кнопку запускается лексический анализ (формирование списка токенов) и последующий синтаксический анализ.
* **TextBlock для вывода результата анализа.**  
  Здесь отображается сообщение об успешном разборе или сообщение об ошибке с указанием причины.
* **TextBox для отображения лога этапов разбора.**  
  Здесь пошагово выводится журнал работы синтаксического анализатора (вход/выход из функций, текущие токены и пр.), что позволяет проследить логику работы парсера.

**5. КОНТРОЛЬНЫЙ ПРИМЕР И РЕЗУЛЬТАТЫ ТЕСТИРОВАНИЯ**

**Пример входного текста:**

do until not x < 10 and y = 20 x = x + 1; output x loop

**Описание разборки:**

1. Парсер обнаруживает ключевое слово do, затем until.
2. Функция LogicalExpression() обрабатывает логическое выражение:  
   – При наличии опционального not оно потребляется.  
   – Разбирается выражение сравнения x < 10.  
   – Затем обнаруживается логическая операция and и далее разбирается выражение сравнения y = 20.
3. Функция Operators() разбирает список операторов:  
   – Первый оператор – присваивание x = x + 1 (арифметическое выражение разбирается с учетом оператора +).  
   – Второй оператор – оператор вывода output x.
4. После обработки операторов парсер ожидает ключевое слово loop, которое корректно обнаруживается.
5. Поскольку дополнительных токенов не остаётся, синтаксический анализ завершается успешно.

**Результат тестирования:**  
На форме в блоке результата отображается сообщение «Синтаксический анализ завершён успешно», а в дополнительном поле выводится подробный лог этапов разбора с отступами.

**6. ЛИСТИНГ ПРОГРАММЫ**

Ниже приведён полный листинг исходного кода проекта.

**App.xaml**

<Application x:Class="RecursiveDescentParser.App"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

StartupUri="MainWindow.xaml">

<Application.Resources>

</Application.Resources>

</Application>

**App.xaml.cs**

using System.Windows;

namespace RecursiveDescentParser

{

public partial class App : Application

{

}

}

**MainWindow.xaml**

<Window x:Class="RecursiveDescentParser.MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

Title="Рекурсивный синтаксический анализатор" Height="600" Width="800">

<Grid Margin="10">

<Grid.RowDefinitions>

<RowDefinition Height="Auto"/>

<RowDefinition Height="2\*"/>

<RowDefinition Height="Auto"/>

<RowDefinition Height="Auto"/>

<RowDefinition Height="2\*"/>

</Grid.RowDefinitions>

<TextBlock Text="Введите исходный текст:" FontWeight="Bold" Grid.Row="0" Margin="0,0,0,5"/>

<TextBox Name="InputTextBox" Grid.Row="1" AcceptsReturn="True"

VerticalScrollBarVisibility="Auto" TextWrapping="Wrap"

Text="do until not x < 10 and y = 20 x = x + 1; output x loop" />

<Button Content="Разобрать" Grid.Row="2" Margin="0,10,0,10"

Width="100" HorizontalAlignment="Left" Click="ParseButton\_Click"/>

<TextBlock Name="ResultTextBlock" Grid.Row="3" TextWrapping="Wrap"

FontWeight="Bold" Foreground="DarkGreen" Margin="0,0,0,10"/>

<TextBox Name="ParseLogTextBox" Grid.Row="4" AcceptsReturn="True"

VerticalScrollBarVisibility="Auto" TextWrapping="Wrap"

IsReadOnly="True" Background="LightGray"/>

</Grid>

</Window>

**MainWindow.xaml.cs**

using System;

using System.Collections.Generic;

using System.Windows;

using System.Windows.Media;

namespace RecursiveDescentParser

{

public partial class MainWindow : Window

{

public MainWindow()

{

InitializeComponent();

}

private void ParseButton\_Click(object sender, RoutedEventArgs e)

{

string input = InputTextBox.Text;

List<Token> tokenList;

try

{

tokenList = Lexer.Tokenize(input);

}

catch (Exception ex)

{

ResultTextBlock.Text = "Ошибка лексического анализа: " + ex.Message;

ResultTextBlock.Foreground = Brushes.Red;

ParseLogTextBox.Text = "";

return;

}

try

{

bool result = Parser.Parse(tokenList);

if (result)

{

ResultTextBlock.Text = "Синтаксический анализ завершён успешно.";

ResultTextBlock.Foreground = Brushes.Green;

}

else

{

ResultTextBlock.Text = "Синтаксический анализ завершился с ошибками.";

ResultTextBlock.Foreground = Brushes.Red;

}

}

catch (Exception ex)

{

ResultTextBlock.Text = "Ошибка синтаксического анализа: " + ex.Message;

ResultTextBlock.Foreground = Brushes.Red;

}

ParseLogTextBox.Text = Parser.GetParseLog();

}

}

}

**Lexer.cs**

using System;

using System.Collections.Generic;

using System.Text;

namespace RecursiveDescentParser

{

public enum TokenType

{

lDo, // "do"

lUntil, // "until"

lLoop, // "loop"

lNot, // "not"

lAnd, // "and"

lOr, // "or"

lOutput, // "output"

lVar, // идентификатор (переменная)

lConst, // константа

lEqual, // "="

lLess, // "<"

lGreater, // ">"

lNotEqual, // "<>"

lPlus, // "+"

lMinus, // "-"

lMul, // "\*"

lDiv, // "/"

lSemicolon // ";"

}

public class Token

{

public TokenType Type { get; set; }

public string Lexeme { get; set; }

public int Position { get; set; }

public Token(TokenType type, string lexeme, int position)

{

Type = type;

Lexeme = lexeme;

Position = position;

}

public override string ToString() => $"{Lexeme} ({Type})";

}

public static class Lexer

{

public static List<Token> Tokenize(string input)

{

List<Token> tokens = new List<Token>();

int pos = 0;

while (pos < input.Length)

{

if (char.IsWhiteSpace(input[pos]))

{

pos++;

continue;

}

int start = pos;

char c = input[pos];

if (char.IsLetter(c))

{

StringBuilder sb = new StringBuilder();

while (pos < input.Length && (char.IsLetterOrDigit(input[pos]) || input[pos] == '\_'))

{

sb.Append(input[pos]);

pos++;

}

string word = sb.ToString();

switch (word.ToLower())

{

case "do":

tokens.Add(new Token(TokenType.lDo, word, start));

break;

case "until":

tokens.Add(new Token(TokenType.lUntil, word, start));

break;

case "loop":

tokens.Add(new Token(TokenType.lLoop, word, start));

break;

case "not":

tokens.Add(new Token(TokenType.lNot, word, start));

break;

case "and":

tokens.Add(new Token(TokenType.lAnd, word, start));

break;

case "or":

tokens.Add(new Token(TokenType.lOr, word, start));

break;

case "output":

tokens.Add(new Token(TokenType.lOutput, word, start));

break;

default:

tokens.Add(new Token(TokenType.lVar, word, start));

break;

}

}

else if (char.IsDigit(c))

{

StringBuilder sb = new StringBuilder();

while (pos < input.Length && char.IsDigit(input[pos]))

{

sb.Append(input[pos]);

pos++;

}

tokens.Add(new Token(TokenType.lConst, sb.ToString(), start));

}

else

{

if (c == '<')

{

if (pos + 1 < input.Length && input[pos + 1] == '>')

{

tokens.Add(new Token(TokenType.lNotEqual, "<>", start));

pos += 2;

}

else

{

tokens.Add(new Token(TokenType.lLess, "<", start));

pos++;

}

}

else if (c == '=')

{

tokens.Add(new Token(TokenType.lEqual, "=", start));

pos++;

}

else if (c == '>')

{

tokens.Add(new Token(TokenType.lGreater, ">", start));

pos++;

}

else if (c == '+')

{

tokens.Add(new Token(TokenType.lPlus, "+", start));

pos++;

}

else if (c == '-')

{

tokens.Add(new Token(TokenType.lMinus, "-", start));

pos++;

}

else if (c == '\*')

{

tokens.Add(new Token(TokenType.lMul, "\*", start));

pos++;

}

else if (c == '/')

{

tokens.Add(new Token(TokenType.lDiv, "/", start));

pos++;

}

else if (c == ';')

{

tokens.Add(new Token(TokenType.lSemicolon, ";", start));

pos++;

}

else

{

pos++;

}

}

}

return tokens;

}

}

}

**Parser.cs**

using System;

using System.Collections.Generic;

using System.Text;

namespace RecursiveDescentParser

{

public static class Parser

{

private static List<Token> tokens;

private static int currentIndex;

private static StringBuilder parseLog = new StringBuilder();

private static int indentLevel = 0;

private static Token CurrentToken => currentIndex < tokens.Count ? tokens[currentIndex] : null;

private static void Log(string message)

{

parseLog.AppendLine(new string(' ', indentLevel \* 2) + message);

}

public static string GetParseLog()

{

return parseLog.ToString();

}

private static void Error(string msg, int pos)

{

throw new Exception($"Ошибка на позиции {pos}: {msg}");

}

public static bool DoUntilStatement()

{

Log("Вход в DoUntilStatement, текущий токен: " + (CurrentToken != null ? CurrentToken.ToString() : "null"));

indentLevel++;

if (CurrentToken == null || CurrentToken.Type != TokenType.lDo)

{

Error("Ожидалось 'do'", CurrentToken != null ? CurrentToken.Position : -1);

return false;

}

currentIndex++; // потребляем 'do'

if (CurrentToken == null || CurrentToken.Type != TokenType.lUntil)

{

Error("Ожидалось 'until'", CurrentToken != null ? CurrentToken.Position : -1);

return false;

}

currentIndex++; // потребляем 'until'

if (!LogicalExpression())

return false;

if (!Operators())

return false;

if (CurrentToken == null || CurrentToken.Type != TokenType.lLoop)

{

Error("Ожидалось 'loop'", CurrentToken != null ? CurrentToken.Position : -1);

return false;

}

currentIndex++; // потребляем 'loop'

indentLevel--;

Log("Выход из DoUntilStatement, результат: true, текущий токен: " + (CurrentToken != null ? CurrentToken.ToString() : "null"));

if (currentIndex < tokens.Count)

{

Error("Лишние токены после 'loop'", CurrentToken != null ? CurrentToken.Position : -1);

return false;

}

return true;

}

public static bool LogicalExpression()

{

Log("Вход в LogicalExpression, текущий токен: " + (CurrentToken != null ? CurrentToken.ToString() : "null"));

indentLevel++;

if (CurrentToken != null && CurrentToken.Type == TokenType.lNot)

{

Log("Найден 'not'");

currentIndex++;

}

if (!ComparisonExpression())

return false;

while (CurrentToken != null &&

(CurrentToken.Type == TokenType.lAnd || CurrentToken.Type == TokenType.lOr))

{

Log("Найден логический оператор: " + CurrentToken.ToString());

currentIndex++; // потребляем and/or

if (CurrentToken != null && CurrentToken.Type == TokenType.lNot)

{

Log("Найден 'not' после логического оператора");

currentIndex++;

}

if (!ComparisonExpression())

return false;

}

indentLevel--;

Log("Выход из LogicalExpression, текущий токен: " + (CurrentToken != null ? CurrentToken.ToString() : "null"));

return true;

}

public static bool ComparisonExpression()

{

Log("Вход в ComparisonExpression, текущий токен: " + (CurrentToken != null ? CurrentToken.ToString() : "null"));

indentLevel++;

if (!Operand())

return false;

if (CurrentToken != null &&

(CurrentToken.Type == TokenType.lLess ||

CurrentToken.Type == TokenType.lGreater ||

CurrentToken.Type == TokenType.lEqual ||

CurrentToken.Type == TokenType.lNotEqual))

{

Log("Найден оператор сравнения: " + CurrentToken.ToString());

currentIndex++; // потребляем оператор сравнения

if (!Operand())

return false;

}

indentLevel--;

Log("Выход из ComparisonExpression, текущий токен: " + (CurrentToken != null ? CurrentToken.ToString() : "null"));

return true;

}

public static bool Operand()

{

Log("Вход в Operand, текущий токен: " + (CurrentToken != null ? CurrentToken.ToString() : "null"));

indentLevel++;

if (CurrentToken == null)

{

Error("Ожидался операнд (идентификатор или константа)", -1);

return false;

}

if (CurrentToken.Type == TokenType.lVar || CurrentToken.Type == TokenType.lConst)

{

Log("Найден операнд: " + CurrentToken.ToString());

currentIndex++; // потребляем операнд

indentLevel--;

Log("Выход из Operand, текущий токен: " + (CurrentToken != null ? CurrentToken.ToString() : "null"));

return true;

}

else

{

Error("Ожидался идентификатор или константа", CurrentToken.Position);

return false;

}

}

public static bool ArithExpr()

{

Log("Вход в ArithExpr, текущий токен: " + (CurrentToken != null ? CurrentToken.ToString() : "null"));

indentLevel++;

if (!Operand())

return false;

while (CurrentToken != null &&

(CurrentToken.Type == TokenType.lPlus ||

CurrentToken.Type == TokenType.lMinus ||

CurrentToken.Type == TokenType.lMul ||

CurrentToken.Type == TokenType.lDiv))

{

Log("Найден арифметический оператор: " + CurrentToken.ToString());

currentIndex++; // потребляем арифметический оператор

if (!Operand())

return false;

}

indentLevel--;

Log("Выход из ArithExpr, текущий токен: " + (CurrentToken != null ? CurrentToken.ToString() : "null"));

return true;

}

public static bool Operators()

{

Log("Вход в Operators, текущий токен: " + (CurrentToken != null ? CurrentToken.ToString() : "null"));

indentLevel++;

if (!OperatorStmt())

return false;

while (CurrentToken != null && CurrentToken.Type == TokenType.lSemicolon)

{

Log("Найден символ ';'");

currentIndex++; // потребляем ';'

if (!OperatorStmt())

return false;

}

indentLevel--;

Log("Выход из Operators, текущий токен: " + (CurrentToken != null ? CurrentToken.ToString() : "null"));

return true;

}

public static bool OperatorStmt()

{

Log("Вход в OperatorStmt, текущий токен: " + (CurrentToken != null ? CurrentToken.ToString() : "null"));

indentLevel++;

if (CurrentToken == null)

{

Error("Ожидался оператор", -1);

return false;

}

if (CurrentToken.Type == TokenType.lVar)

{

Log("Обнаружено присваивание, идентификатор: " + CurrentToken.ToString());

currentIndex++; // потребляем идентификатор

if (CurrentToken == null || CurrentToken.Type != TokenType.lEqual)

{

Error("Ожидалось '=' в присваивании", CurrentToken != null ? CurrentToken.Position : -1);

return false;

}

Log("Найден '='");

currentIndex++; // потребляем '='

if (!ArithExpr())

return false;

indentLevel--;

Log("Выход из OperatorStmt (присваивание), текущий токен: " + (CurrentToken != null ? CurrentToken.ToString() : "null"));

return true;

}

else if (CurrentToken.Type == TokenType.lOutput)

{

Log("Обнаружен оператор вывода: " + CurrentToken.ToString());

currentIndex++; // потребляем 'output'

if (!Operand())

return false;

indentLevel--;

Log("Выход из OperatorStmt (вывод), текущий токен: " + (CurrentToken != null ? CurrentToken.ToString() : "null"));

return true;

}

else

{

Error("Ожидалось присваивание или оператор вывода", CurrentToken.Position);

return false;

}

}

public static bool Parse(List<Token> tokenList)

{

tokens = tokenList;

currentIndex = 0;

parseLog.Clear();

indentLevel = 0;

Log("=== Начало синтаксического анализа ===");

bool result = DoUntilStatement();

Log("=== Конец синтаксического анализа ===");

return result;

}

}

}

**ЗАКЛЮЧЕНИЕ**

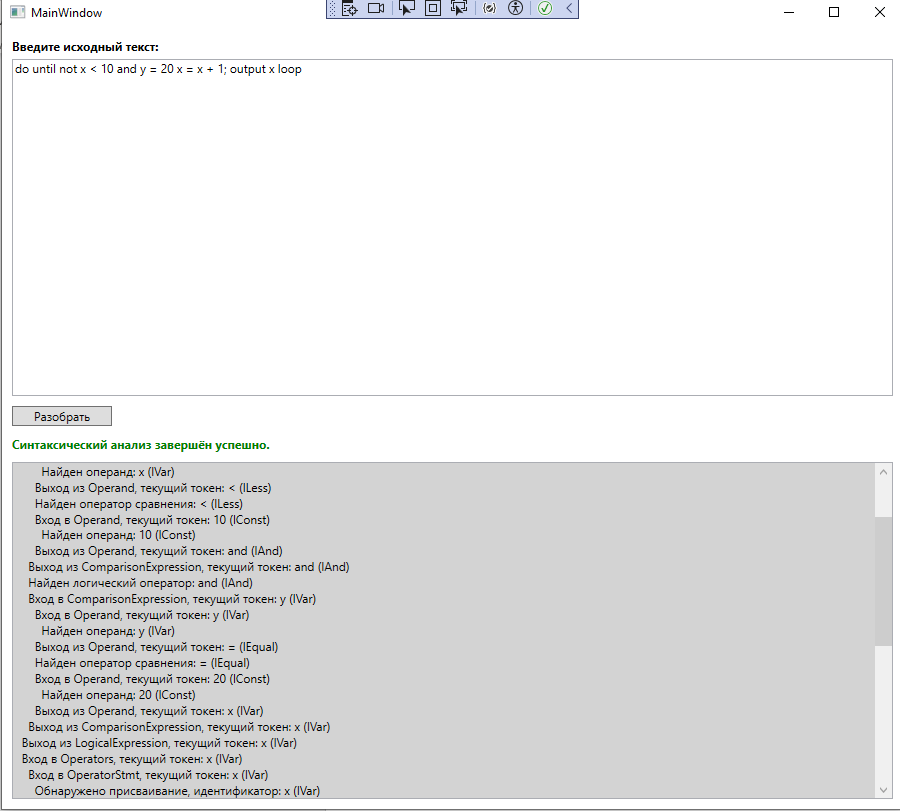
В ходе выполнения лабораторной работы №2 был разработан синтаксический анализатор методом рекурсивного спуска для языка, поддерживающего конструкцию

do until <логическое выражение> <операторы> loop

с грамматикой уровня 3 (учёт унарных и бинарных логических операций, выражений сравнения и арифметических выражений). Реализована обработка ошибок с выводом позиции и информативным сообщением. Также реализована подробная регистрация этапов разбора, что позволяет отслеживать работу парсера.

Контрольный пример:

do until not x < 10 and y = 20 x = x + 1; output x loop

при запуске приложения в WPF отображает сообщение об успешном разборе и подробный лог этапов синтаксического анализа. 

Для

do until not x < 10 and y = 20 x = x + 1; output x loop;

при запуске приложения в WPF отображает сообщение об ошибке.

