Командный проект по дисциплине «Функциональное и логическое программирование»

Задача 2038N (Исправление выражения)

Выражение - это строка из трёх символов, где первый и последний символы - цифры (от 0 до 9), а средний символ - знак сравнения (<, = или >).

Выражение считается истинным, если знак сравнения соответствует цифрам (например, если первая цифра строго меньше последней, то знак сравнения должен быть <).

Например, выражения 1<3, 4>2, 0=0 являются истинными, а 5>5, 7<3 - нет.

Вам дана строка s, представляющая собой выражение. Измените как можно меньше символов, чтобы s стало истинным выражением. Если s уже истинно, оставьте его без изменений.

Входные данные

Первая строка содержит целое число t ($1 \le t \le 300$) - количество тестовых случаев.

Каждый тестовый случай состоит из одной строки, содержащей строку s ($|s| = 3$, первый и последний символы s - цифры, второй символ - знак сравнения).

Выходные данные

Для каждого тестового случая выведите строку из 3 символов - истинное выражение, полученное изменением минимального количества символов в s. Если есть несколько возможных ответов, выведите любой из них.

Решение на императивном языке C#

```
public static class ExpressionFixer
{
    public static string FixExpression(string input)
    {
        char[] chars = input.ToCharArray();

        if (chars[0] < chars[2])
        {
            chars[1] = '<';
        }
        else if (chars[0] > chars[2])
        {
            chars[1] = '>';
        }
        else
        {
            chars[1] = '=';
        }

        return new string(chars);
    }
}
```

Описание решения на императивном языке

Метод FixExpression принимает входную строку и преобразует её в массив символов. Для преобразования строки в массив символов используется метод ToCharArray.

Затем он сравнивает первую цифру (chars[0]) с последней (chars[2]). Если первая цифра меньше последней, знак сравнения (chars[1]) устанавливается в '<'. Если первая цифра больше последней, знак устанавливается в '>'. Если цифры равны, знак становится '='.

После установки правильного знака массив преобразуется обратно в строку с помощью конструктора new string(chars)

Проверка работоспособности решения

Мои посылки

№	Когда	Кто	Задача	Язык	Вердикт	Время	Память
321982879	30.05.2025 03:30	rexandel	N - Fixing the Expression	C# 13	Полное решение	77 мс	0 КБ

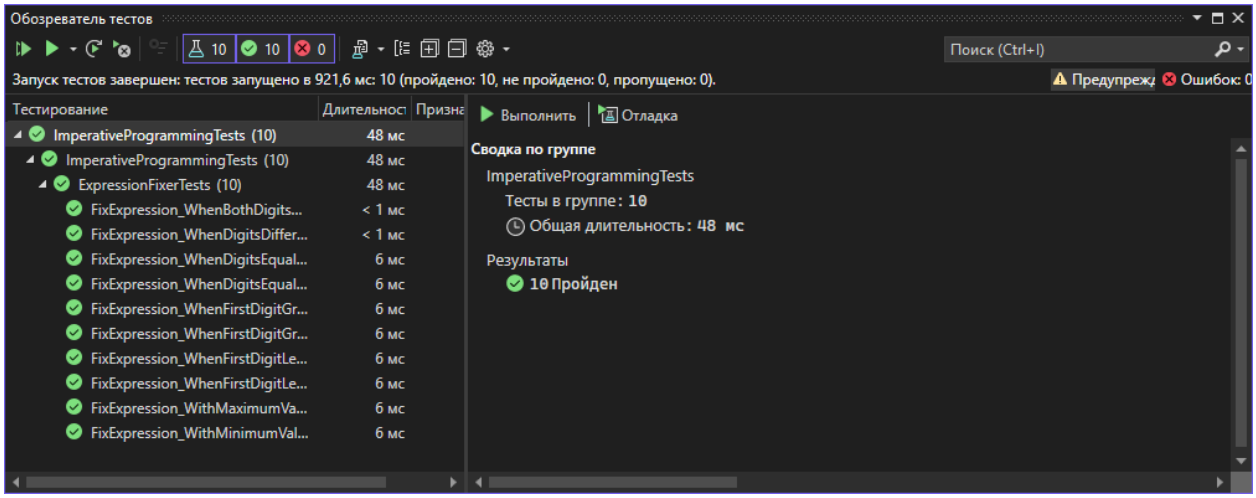
Написание автотестов

Для проверки корректности работы метода ExpressionFixer.FixExpression было написано 10 юнит-тестов с использованием фреймворка Microsoft Visual Studio Test Tools.

Список тестов:

1. Проверка корректного выражения с оператором '<'.
2. Исправление выражения с неправильным оператором '<'.
3. Проверка корректного выражения с оператором '>'.
4. Исправление выражения с неправильным оператором '>'.
5. Проверка корректного выражения с оператором '='.
6. Исправление выражения с неправильным оператором '='.
7. Проверка обработки минимальных значений цифр.
8. Проверка обработки максимальных значений цифр.
9. Проверка выражения с нулями и неправильным оператором.
10. Проверка выражений с цифрами, отличающимися на единицу.

Результат прохождения тестирования



Листинг автотестов

```
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace ImperativeProgrammingTests
{
    [TestClass]
    public class ExpressionFixerTests
    {
        // (1) Тест на корректное исправление выражения с оператором '<'
        [TestMethod]
        public void
FixExpression_WhenFirstDigitLessThanSecond_ShouldSetLessThanOperator()
        {
            // Arrange
            string input = "1<3"; // Здесь оператор правильный

            // Act
            string result = ExpressionFixer.FixExpression(input);

            // Assert
            Assert.AreEqual("1<3", result);
        }

        // (2) Тест на исправление выражения с неправильным оператором '<'
        [TestMethod]
        public void
FixExpression_WhenFirstDigitLessThanSecondWithWrongOperator_ShouldFixToLessThan()
        {
            // Arrange
            string input = "1>3"; // Здесь оператор неправильный

            // Act
            string result = ExpressionFixer.FixExpression(input);

            // Assert
            Assert.AreEqual("1<3", result);
        }

        // (3) Тест на корректное исправление выражения с оператором '>'
        [TestMethod]
        public void
FixExpression_WhenFirstDigitGreaterThanSecond_ShouldSetGreaterThanOperator()
        {
            // Arrange
            string input = "3>1"; // Здесь оператор правильный

            // Act
            string result = ExpressionFixer.FixExpression(input);

            // Assert
            Assert.AreEqual("3>1", result);
        }
```

```
        // (4) Тест на исправление выражения с неправильным оператором '>'
        [TestMethod]
        public void
FixExpression_WhenFirstDigitGreaterThanSecondWithWrongOperator_ShouldFixToGreaterTha
n()
        {
            // Arrange
            string input = "3=1"; // Здесь оператор неправильный

            // Act
            string result = ExpressionFixer.FixExpression(input);

            // Assert
            Assert.AreEqual("3>1", result);
        }

        // (5) Тест на корректное исправление выражения с оператором '='
        [TestMethod]
        public void FixExpression_WhenDigitsEqual_ShouldSetEqualOperator()
        {
            // Arrange
            string input = "5=5"; // Здесь оператор правильный

            // Act
            string result = ExpressionFixer.FixExpression(input);

            // Assert
            Assert.AreEqual("5=5", result);
        }

        // (6) Тест на исправление выражения с неправильным оператором '='
        [TestMethod]
        public void
FixExpression_WhenDigitsEqualWithWrongOperator_ShouldFixToEqualOperator()
        {
            // Arrange
            string input = "5<5"; // Здесь оператор неправильный

            // Act
            string result = ExpressionFixer.FixExpression(input);

            // Assert
            Assert.AreEqual("5=5", result);
        }

        // (7) Тест на обработку минимальных значений
        [TestMethod]
        public void FixExpression_WithMinimumValues_ShouldHandleCorrectly()
        {
            // Arrange
            string input = "0>9"; // 0 сильно меньше 9

            // Act
            string result = ExpressionFixer.FixExpression(input);

            // Assert
            Assert.AreEqual("0<9", result);
        }
```

```
        // (8) Тест на обработку максимальных значений
        [TestMethod]
        public void FixExpression_WithMaximumValues_ShouldHandleCorrectly()
        {
            // Arrange
            string input = "9<0"; // 9 сильно больше 0

            // Act
            string result = ExpressionFixer.FixExpression(input);

            // Assert
            Assert.AreEqual("9>0", result);
        }

        // (9) Тест на корректное исправление выражения с нулями и разными
операторами
        [TestMethod]
        public void
FixExpression_WhenBothDigitsZeroWithWrongOperator_ShouldFixToEqualOperator()
        {
            // Arrange
            string input = "0>0"; // Оба нуля, оператор неправильный

            // Act
            string result = ExpressionFixer.FixExpression(input);

            // Assert
            Assert.AreEqual("0=0", result);
        }

        // (10) Тест на обработку цифр с разницей в 1
        [TestMethod]
        public void FixExpression_WhenDigitsDifferByOne_ShouldSetCorrectOperator()
        {
            // Arrange
            string input1 = "4=5"; // 4 < 5
            string input2 = "5=4"; // 5 > 4
            string input3 = "5>4"; // Уже правильно
            string input4 = "4<5"; // Уже правильно

            // Act & Assert
            Assert.AreEqual("4<5", ExpressionFixer.FixExpression(input1));
            Assert.AreEqual("5>4", ExpressionFixer.FixExpression(input2));
            Assert.AreEqual("5>4", ExpressionFixer.FixExpression(input3));
            Assert.AreEqual("4<5", ExpressionFixer.FixExpression(input4));
        }
    }
}
```

Решение на функциональном языке Prolog

```
% Факты - утверждения, которые всегда истинны
% Правила - условные утверждения вида "если...то"
% Предикаты - (аналог функции из других языков)

% Программа на Prolog отвечает на вопросы (запросы), пытаясь найти решения, которые
удовлетворяют заданным правилам

% :- (Левая часть истинна, если правая часть истинна)

% ----------------------------------------------------------------------------

% Предикат для сравнения цифр и определения правильного оператора
compare_digits(D1, D2, '<') :- D1 < D2.  % Оператор '<' корректен между D1 и D2,
ЕСЛИ D1 меньше D2
compare_digits(D1, D2, '>') :- D1 > D2.  % Оператор '>' корректен между D1 и D2,
ЕСЛИ D1 больше D2
compare_digits(D, D, '=') !.  % Оператор '=' корректен между одной и той же
переменной

% ----------------------------------------------------------------------------

% Основной предикат для исправления выражения
fix_expression([D1, _, D2], [D1, Op, D2]) :-  % Результат [D1, Op, D2] корректен для
входа [D1, _, D2] ЕСЛИ
    % char_code - Встроенный предикат, который преобразует символ в ACII код
      char_code(D1, Code1),  % Можно получить Code1 из D1
    char_code(D2, Code2),  % Можно получить Code2 из D2
      % Предикат compare_digits используется для определения правильного ответа
    compare_digits(Code1, Code2, Op).  % Этот предикат является истинным

% ----------------------------------------------------------------------------

% Вспомогательные предикаты для работы со строками
string_to_list(String, List) :-
      % string_chars - встроенный предикат, который преобразует строку в список или
список в строку
    string_chars(String, List).

list_to_string(List, String) :-
      % string_chars - встроенный предикат, который преобразует строку в список или
список в строку
    string_chars(String, List).

% ----------------------------------------------------------------------------

% Главный предикат (интерфейс)
fix_expression_string(Input, Output) :-
    string_to_list(Input, List),
    fix_expression(List, FixedList),
    list_to_string(FixedList, Output).
```

Описание решения на функциональном языке Prolog

Основная логика программы реализована через предикат fix_expression, который принимает входной список символов вида [D1, _, D2] и возвращает исправленный список [D1, Op, D2]. Входной список состоит из двух цифр (D1 и D2) и произвольного среднего символа, который игнорируется (обозначен как _). Предикат преобразует символы цифр в их ASCII-коды с помощью встроенного предиката char_code, чтобы сравнить их числовые значения. Затем он использует вспомогательный предикат compare_digits для определения правильного знака сравнения (Op), который должен быть '<', '>' или '=' в зависимости от отношения между цифрами.

Предикат compare_digits определяет корректный знак сравнения на основе числовых значений цифр. Если ASCII-код первой цифры меньше кода второй, выбирается оператор '<'. Если первая цифра больше, выбирается '>'. Если цифры равны, выбирается '='. Это гарантирует, что выходной знак сравнения делает выражение истинным, например, для входа "1>3" результат будет "1<3", так как 1 меньше 3.

Для удобной работы со строками программа использует вспомогательные предикаты string_to_list и list_to_string, которые преобразуют входную строку в список символов и обратно с помощью встроенного предиката string_chars. Это позволяет обрабатывать входные данные в формате строки, как указано в задаче, и возвращать результат также в виде строки.

Главный предикат fix_expression_string служит интерфейсом программы. Он принимает входную строку, преобразует её в список символов, вызывает fix_expression для исправления списка, а затем преобразует результат обратно в строку. Например, для входной строки "5<5" программа вернёт "5=5", так как цифры равны, и правильным оператором является '='.

Проведение автотестов

Была использована та же логика, что и ранее для императивного языка.

```
[11/11] expression_fixer:invalid_length ..................................................... passed (0.000 sec)
% All 11 tests passed in 0.039 seconds (0.000 cpu)
true.
```

Решение на функциональном языке F#

```
open System

// Функция для определения правильного знака сравнения на основе двух цифр
let getCorrectOperator (left: char) (right: char) : char =
    match compare (int left) (int right) with
        | result when result < 0 -> '<'
        | result when result > 0 -> '>'
        | _ -> '='

// Функция для проверки истинности выражения
let isExpressionTrue (expression: string) : bool =
    let chars = expression.ToCharArray()
    let leftDigit = chars.[0]
    let operator = chars.[1]
    let rightDigit = chars.[2]
    let correctOperator = getCorrectOperator leftDigit rightDigit
    operator = correctOperator

// Основная функция для исправления выражения
let fixExpression (input: string) : string =
    match isExpressionTrue input with
        | true -> input  // Если выражение уже истинно, возвращаем без изменений
        | false ->
            let chars = input.ToCharArray() |> List.ofArray
            let leftDigit::op::rightDigit::_ = chars
            let correctOperator = getCorrectOperator leftDigit rightDigit
            // Создаем новое выражение
            [leftDigit; correctOperator; rightDigit]
            |> String.Concat

[<EntryPoint>]
let main (_: string[]) : int =
    // Чтение количества тестовых случаев
    let t = Console.ReadLine() |> int
    // Обработка каждого теста
    let testCases = [ for _ in 1..t -> Console.ReadLine() ]

    testCases
        |> List.map fixExpression
        |> List.iter (printfn "%s")
    0
```

Описание решения на функциональном языке F#

Функция getCorrectOperator отвечает за определение правильного знака сравнения на основе двух входных символов, представляющих цифры. Она преобразует символы в их числовые значения с помощью функции int, использует встроенную функцию compare для определения их отношения и возвращает соответствующий оператор: '<' если первая цифра меньше второй, '>' если больше, или '=' если они равны. Эта функция является чистой, так как не имеет побочных эффектов и зависит только от входных параметров, что соответствует принципам функционального программирования.

Функция isExpressionTrue проверяет, является ли входное выражение истинным. Она извлекает из строки три символа: первую цифру, оператор и вторую цифру, а затем сравнивает текущий оператор с правильным, вычисленным функцией getCorrectOperator. Если операторы совпадают, функция возвращает true, указывая, что выражение не требует изменений. Эта проверка, хотя и избыточна в данном контексте, добавляет явную логику для определения необходимости исправления, что делает код более читаемым, но менее оптимальным.

Основная функция fixExpression обрабатывает входное выражение и возвращает исправленную строку. Если выражение уже истинно, как определено функцией isExpressionTrue, оно возвращается без изменений. В противном случае строка преобразуется в список символов с помощью ToCharArray и List.ofArray, после чего список декомпозируется с использованием шаблона leftDigit::op::rightDigit::_, характерного для списков Черча. Первая и последняя цифры сохраняются, а оператор заменяется на правильный, вычисленный getCorrectOperator. Новый список из трёх символов преобразуется в строку с помощью String.Concat, обеспечивая минимальное изменение выражения, как требуется в задаче.

Функция main, помеченная атрибутом [EntryPoint], является точкой входа программы. Она читает количество тестовых случаев t с консоли, преобразует его в целое число и использует генератор списков для создания списка из t строк, считанных с помощью Console.ReadLine. Этот генератор заменяет императивный цикл, что делает программу более функциональной. Список тестовых случаев обрабатывается с помощью List.map, применяющего fixExpression к каждой строке, а результат передаётся в List.iter для вывода каждого исправленного выражения в консоль. Функция возвращает 0, обозначая успешное завершение.

Проверка работоспособности решения

Мои посылки

№	Когда	Кто	Задача	Язык	Вердикт	Время	Память
321987807	30.05.2025 05:29	rexandel	N - Fixing the Expression	F# 9	Полное решение	77 мс	0 КБ

Проведение тестирования

```
10
1<3
1>3
3>1
3=1
5=5
5<5
0>9
9<0
0>0
3<1
1<3
1<3
3>1
3>1
5=5
5=5
0<9
9>0
0=0
3>1
```

Задача 2038L (Ремонт мостов)

Недавно Монокарп стал директором парка, расположенного рядом с его домом. Парк довольно большой, и через него протекает небольшая река, разделяющая его на несколько зон. Через реку построено несколько мостов. Три из этих мостов особенно старые и требуют ремонта.

Все три моста имеют одинаковую длину, но различаются по ширине. Их ширина составляет 18, 21 и 25 единиц соответственно.

Во время ремонта парка Монокарпу нужно заменить старые доски, которые служили настилом мостов, на новые.

Доски продаются стандартной длины — 60 единиц. Монокарп уже знает, что для каждого моста нужно по n досок. Но поскольку мосты имеют разную ширину, ему нужно:

n досок длиной 18 для первого моста,

n досок длиной 21 для второго моста,

n досок длиной 25 для третьего моста.

Рабочие, занимающиеся ремонтом, могут без проблем разрезать доски на части, но отказываются склеивать доски, так как это создаёт слабые места и выглядит некрасиво.

Монокарп хочет купить как можно меньше стандартных досок, но ему сложно рассчитать нужное количество. Сможете ли вы ему помочь?

Входные данные

В первой и единственной строке содержится целое число n ($1 \le n \le 1000$) — количество досок, необходимых для каждого из трёх мостов.

Выходные данные

Выведите одно целое число — минимальное количество стандартных досок (длиной 60 единиц), которое нужно Монокарпу, чтобы покрыть все три моста, если доски можно разрезать на части.

Решение на императивном языке C#

```
using System;

class Program
{
    static void Main()
    {
        int n = int.Parse(Console.ReadLine());

        if (n == 1)
        {
            Console.WriteLine(2);
            return;
        }

        int result = n / 2;
        int remaining18 = n - (n / 2);

        if (n % 2 != 0)
        {
            result++;
            remaining18 -= 2;
        }

        if (remaining18 > 0)
        {
            result += remaining18 / 3;
            remaining18 %= 3;
        }

        result += (remaining18 + n + 1) / 2;

        Console.WriteLine(result);
    }
}
```

Описание решения на императивном языке

Алгоритм начинает с того, что определяет, сколько таких идеальных комбинаций можно составить. Поскольку каждая комбинация требует один кусок по 18 и два куска по 21, а всего нужно n кусков каждого типа, максимальное количество полных комбинаций равно n/2 (так как кусков по 21 должно хватить на пары). Каждая такая комбинация занимает ровно одну стандартную доску, поэтому базовое количество досок равно n/2.

Когда n четное, все куски по 21 аккуратно разбиваются на пары, и остается ровно n/2 кусков по 18, которые не вошли в базовые комбинации. Однако когда n нечетное, остается один лишний кусок по 21, который не может образовать пару. В этом случае алгоритм выделяет дополнительную доску специально для этого куска. Из доски длиной 60, отрезав кусок длиной 21, остается 39 единиц, из которых можно вырезать еще два куска по 18 единиц. Поэтому количество оставшихся кусков по 18 уменьшается на 2.

После базовых комбинаций и обработки нечетного случая могут остаться куски длиной 18, которые нужно разместить на дополнительных досках. Поскольку из одной доски можно вырезать максимум 3 куска по 18 единиц (3×18=54, остаток 6), алгоритм вычисляет необходимое количество дополнительных досок как remaining18/3. После этого может остаться от 0 до 2 кусков по 18, которые обрабатываются в финальной формуле.

Самая сложная часть алгоритма заключается в финальной формуле (remaining18 + n + 1) / 2. Эта формула одновременно учитывает все оставшиеся куски длиной 18 и все n кусков длиной 25. Логика заключается в том, что эти куски можно оптимально комбинировать на дополнительных досках: либо размещать по два куска длиной 25 на доске (2×25=50, остаток 10), либо комбинировать один кусок длиной 25 с одним куском длиной 18 (25+18=43, остаток 17). В среднем на каждой доске можно разместить эквивалент двух позиций, поэтому общее количество позиций (remaining18 + n) делится на 2 с округлением вверх.

Проверка работоспособности решения

Мои посылки

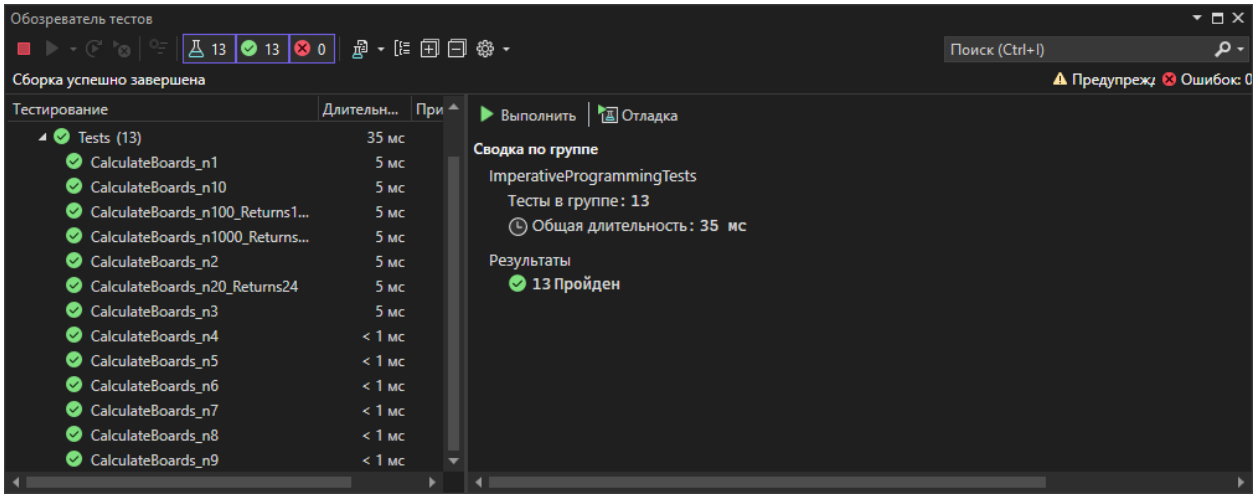
№	Когда	Кто	Задача	Язык	Вердикт	Время	Память
321992843	30.05.2025 06:41	rexandel	L - Bridge Renovation	C# 13	Полное решение	78 мс	0 КБ

Написание автотестов

Для проверки корректности работы метода BoardCalculator.CalculateBoards было написано 13 юнит-тестов с использованием фреймворка Microsoft Visual Studio Test Tools.

Список тестов:

1. Проверка минимального значения (n=1)
2. Проверка случая с n=2
3. Проверка случая с n=3
4. Проверка случая с n=4
5. Проверка случая с n=5
6. Проверка случая с n=6
7. Проверка случая с n=7
8. Проверка случая с n=8
9. Проверка случая с n=9
10. Проверка случая с n=10
11. Проверка случая с n=20
12. Проверка обработки значения n=100
13. Проверка обработки значения n=1000



Листинг автотестов

```
namespace ImperativeProgrammingTests
{
    [TestClass]
    public sealed class Tests
    {
        [TestMethod]
        public void CalculateBoards_n1()
        {
            Assert.AreEqual(2, BoardCalculator.CalculateBoards(1));
        }

        [TestMethod]
        public void CalculateBoards_n2()
        {
            Assert.AreEqual(3, BoardCalculator.CalculateBoards(2));
        }

        [TestMethod]
        public void CalculateBoards_n3()
        {
            Assert.AreEqual(4, BoardCalculator.CalculateBoards(3));
        }

        [TestMethod]
        public void CalculateBoards_n4()
        {
            Assert.AreEqual(5, BoardCalculator.CalculateBoards(4));
        }

        [TestMethod]
        public void CalculateBoards_n5()
        {
            Assert.AreEqual(6, BoardCalculator.CalculateBoards(5));
        }

        [TestMethod]
        public void CalculateBoards_n6()
        {
            Assert.AreEqual(7, BoardCalculator.CalculateBoards(6));
        }

        [TestMethod]
        public void CalculateBoards_n7()
        {
            Assert.AreEqual(9, BoardCalculator.CalculateBoards(7));
        }

        [TestMethod]
        public void CalculateBoards_n8()
        {
            Assert.AreEqual(10, BoardCalculator.CalculateBoards(8));
        }

        [TestMethod]
        public void CalculateBoards_n9()
        {
            Assert.AreEqual(11, BoardCalculator.CalculateBoards(9));
        }
```

```
        [TestMethod]
        public void CalculateBoards_n10()
        {
            Assert.AreEqual(12, BoardCalculator.CalculateBoards(10));
        }

        [TestMethod]
        public void CalculateBoards_n20_Returns24()
        {
            Assert.AreEqual(24, BoardCalculator.CalculateBoards(20));
        }

        [TestMethod]
        public void CalculateBoards_n100_Returns117()
        {
            Assert.AreEqual(117, BoardCalculator.CalculateBoards(100));
        }

        [TestMethod]
        public void CalculateBoards_n1000_Returns1167()
        {
            Assert.AreEqual(1167, BoardCalculator.CalculateBoards(1000));
        }
    }
}
```

Решение на функциональном языке Prolog

```
min_boards(N, MinBoards) :-
    UpperBound is ceiling(N * 18 / 60) + ceiling(N * 21 / 60) + ceiling(N * 25 /
60),

    between(1, UpperBound, MinBoards),
    can_cut_boards(MinBoards, N, 18, N, 21, N, 25),
    !.

% Проверяет, можно ли из заданного количества досок нарезать нужные части
can_cut_boards(TotalBoards, Need18, Len18, Need21, Len21, Need25, Len25) :-
    generate_cutting_combinations(TotalBoards, Need18, Len18, Need21, Len21, Need25,
Len25, [], Combinations),
    member(combination(Got18, Got21, Got25), Combinations),
    Got18 >= Need18,
    Got21 >= Need21,
    Got25 >= Need25.

% Генерирует все возможные комбинации нарезки досок
generate_cutting_combinations(0, _, _, _, _, _, _, Acc, Acc) :- !.

generate_cutting_combinations(BoardsLeft, Need18, Len18, Need21, Len21, Need25,
Len25, Acc, Result) :-
    BoardsLeft > 0,
    generate_single_board_cuts(60, Len18, Len21, Len25, Cut18, Cut21, Cut25),

    % Обновляем аккумулятор
    (   Acc = [] ->
        NewAcc = [combination(Cut18, Cut21, Cut25)]
    ;   update_combinations(Acc, Cut18, Cut21, Cut25, NewAcc)
    ),

    BoardsLeft1 is BoardsLeft - 1,
    generate_cutting_combinations(BoardsLeft1, Need18, Len18, Need21, Len21, Need25,
Len25, NewAcc, Result).

% Генерирует все возможные способы нарезки одной доски длиной 60
generate_single_board_cuts(BoardLength, Len18, Len21, Len25, Cut18, Cut21, Cut25) :-
    Max18 is BoardLength // Len18,
    between(0, Max18, Cut18),
    Remaining1 is BoardLength - Cut18 * Len18,

    Max21 is Remaining1 // Len21,
    between(0, Max21, Cut21),
    Remaining2 is Remaining1 - Cut21 * Len21,

    Max25 is Remaining2 // Len25,
    between(0, Max25, Cut25),

    % Проверяем, что вся длина использована корректно
    UsedLength is Cut18 * Len18 + Cut21 * Len21 + Cut25 * Len25,
    UsedLength =< BoardLength.

% Обновляет комбинации, добавляя новые отрезки
update_combinations([], Cut18, Cut21, Cut25, [combination(Cut18, Cut21, Cut25)]).

update_combinations([combination(Acc18, Acc21, Acc25)|Rest], Cut18, Cut21, Cut25,
                    [combination(New18, New21, New25)|NewRest]) :-
    New18 is Acc18 + Cut18,
    New21 is Acc21 + Cut21,
    New25 is Acc25 + Cut25,
    update_combinations(Rest, Cut18, Cut21, Cut25, NewRest).
```

Описание решения функциональном языке Prolog

Алгоритм начинает работу с определения верхней границы количества досок, которое потребуется в наихудшем случае. Эта граница вычисляется как сумма минимального количества досок для каждого моста отдельно, при условии, что каждый мост обслуживается независимо от остальных. Для этого используется формула округления вверх от деления общей длины досок каждого моста на длину стандартной доски.

После установления верхней границы алгоритм применяет метод перебора, проверяя каждое возможное количество досок от единицы до верхней границы. Для каждого проверяемого количества досок запускается процедура генерации всех возможных способов нарезки этих досок на отрезки нужных размеров.

Процесс генерации комбинаций работает рекурсивно, обрабатывая по одной доске за раз. Для каждой отдельной доски алгоритм определяет все возможные способы ее разрезания на отрезки длиной 18, 21 и 25 единиц. Это делается путем перебора всех допустимых количеств отрезков каждого типа, начиная с максимально возможного количества отрезков длиной 18, затем для оставшейся длины определяя максимальное количество отрезков длиной 21, и наконец, для оставшейся после этого длины вычисляя количество отрезков длиной 25.

При обработке каждой новой доски результаты ее нарезки добавляются к уже накопленным результатам от предыдущих досок. Это происходит через обновление аккумулятора, который хранит информацию о том, сколько отрезков каждого типа уже получено. Если аккумулятор пуст (обрабатывается первая доска), в него просто записывается результат нарезки текущей доски. В противном случае количества отрезков каждого типа от текущей доски добавляются к соответствующим значениям в каждой существующей комбинации.

После генерации всех возможных комбинаций нарезки заданного количества досок алгоритм проверяет, существует ли среди них хотя бы одна комбинация, которая обеспечивает достаточное количество отрезков каждого типа для всех трех мостов. Если такая комбинация найдена, текущее количество досок является решением задачи. Благодаря использованию оператора отсечения, алгоритм останавливается на первом найденном решении, которое гарантированно является минимальным из-за порядка перебора от меньшего к большему.

Проведение автотестов

Была использована та же логика, что и ранее для императивного языка.

```
?- run_tests.
[7/13] board_calculator:calculate_boards_n7 ..
```

Решение на функциональном языке F#

```
open System

// Функция для генерации всех возможных способов нарезки одной доски длиной 60
let generateSingleBoardCuts len18 len21 len25 =
    [ for cut18 in 0 .. (60 / len18) do
        let remaining1 = 60 - cut18 * len18
        for cut21 in 0 .. (remaining1 / len21) do
            let remaining2 = remaining1 - cut21 * len21
            for cut25 in 0 .. (remaining2 / len25) do
                if cut18 * len18 + cut21 * len21 + cut25 * len25 <= 60 then
                    yield (cut18, cut21, cut25) ]

// Проверяет, можно ли из totalBoards досок нарезать нужное количество кусков
let canCutBoards totalBoards need18 need21 need25 len18 len21 len25 =
    let rec checkCombinations boardsLeft need18 need21 need25 combinations =
        if boardsLeft = 0 then
            need18 <= 0 && need21 <= 0 && need25 <= 0
        else
            combinations
            |> List.exists (fun (cut18, cut21, cut25) ->
                checkCombinations (boardsLeft - 1) (need18 - cut18) (need21 - cut21)
(need25 - cut25) combinations)

    let possibleCuts = generateSingleBoardCuts len18 len21 len25
    checkCombinations totalBoards need18 need21 need25 possibleCuts

// Находит минимальное количество досок
let minBoards n =
    let len18, len21, len25 = 18, 21, 25
    let upperBound = ceil (float (n * len18) / 60.0) + ceil (float (n * len21) /
60.0) + ceil (float (n * len25) / 60.0) |> int
    [1 .. upperBound]
    |> List.find (fun totalBoards -> canCutBoards totalBoards n n n len18 len21
len25)

// Основная программа
[<EntryPoint>]
let main argv =
    let n = Console.ReadLine() |> int
    let result = minBoards n
    printfn "%d" result
    0
```

Описание решения на функциональном языке F#

Данный алгоритм решает задачу определения минимального количества досок длиной 60 единиц, из которых можно нарезать заданное количество кусков трех видов: длиной 18, 21 и 25 единиц. Каждый тип кусков требуется в количестве n штук. Алгоритм использует метод перебора возможных вариантов нарезки, чтобы найти оптимальное решение.

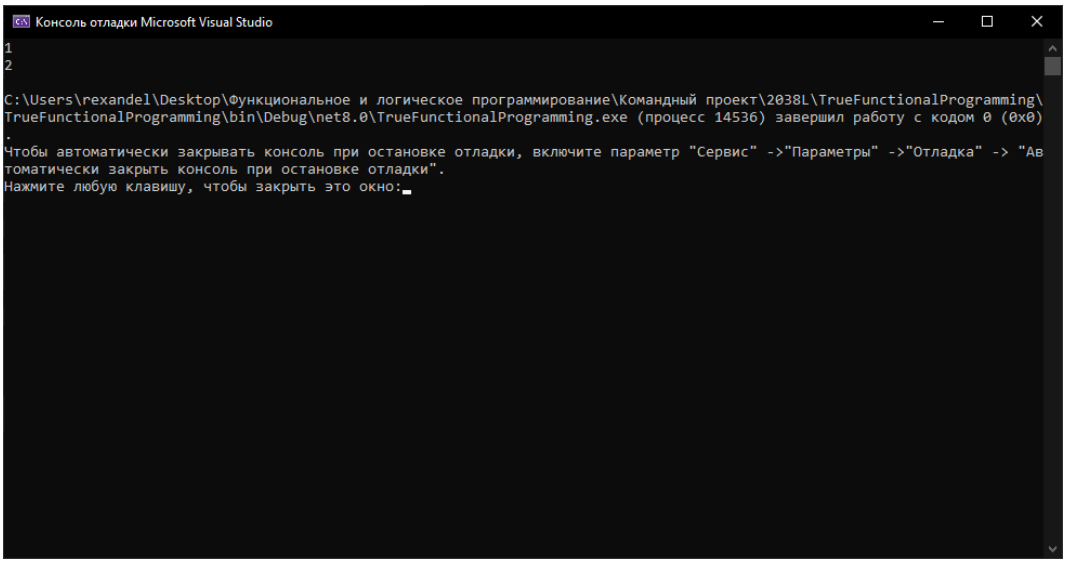
Алгоритм начинается с функции generateSingleBoardCuts, которая генерирует все возможные способы нарезки одной доски длиной 60 единиц на куски заданных длин. Для этого функция перебирает все допустимые комбинации количества кусков каждого типа (18, 21 и 25), которые могут быть получены из одной доски. Например, если из доски можно вырезать 2 куска по 18 и 1 кусок по 21, то такая комбинация (2, 1, 0) будет включена в список возможных вариантов.

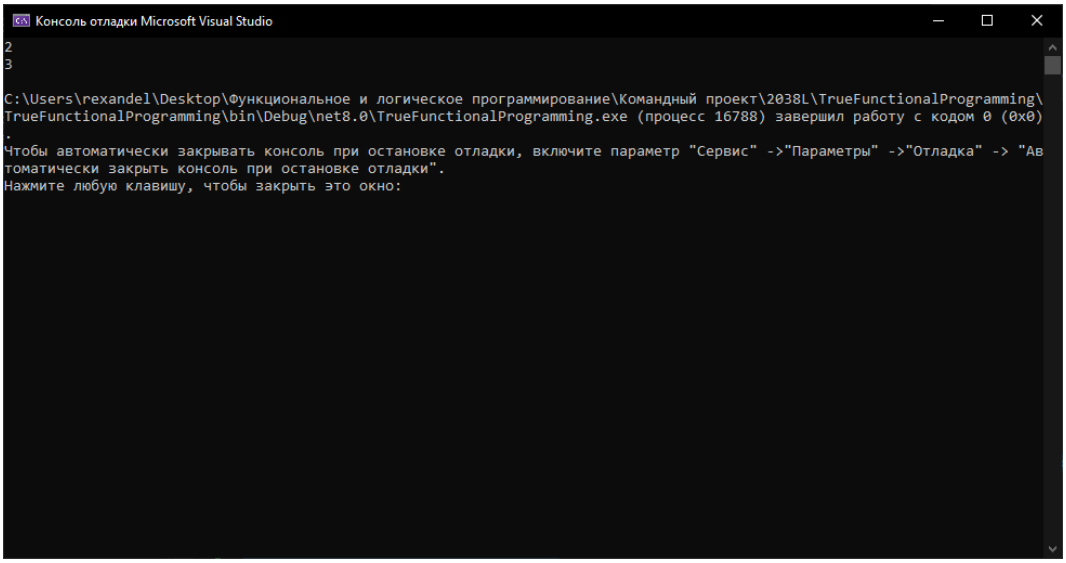
Следующая функция canCutBoards проверяет, можно ли из заданного количества досок (totalBoards) нарезать требуемое количество кусков (need18, need21, need25). Для этого используется рекурсивный подход: на каждом шаге проверяется, можно ли покрыть оставшуюся потребность в кусках, используя одну из возможных комбинаций нарезки. Если после использования всех досок потребность в кусках удовлетворена (или превышена), функция возвращает true, иначе — false.

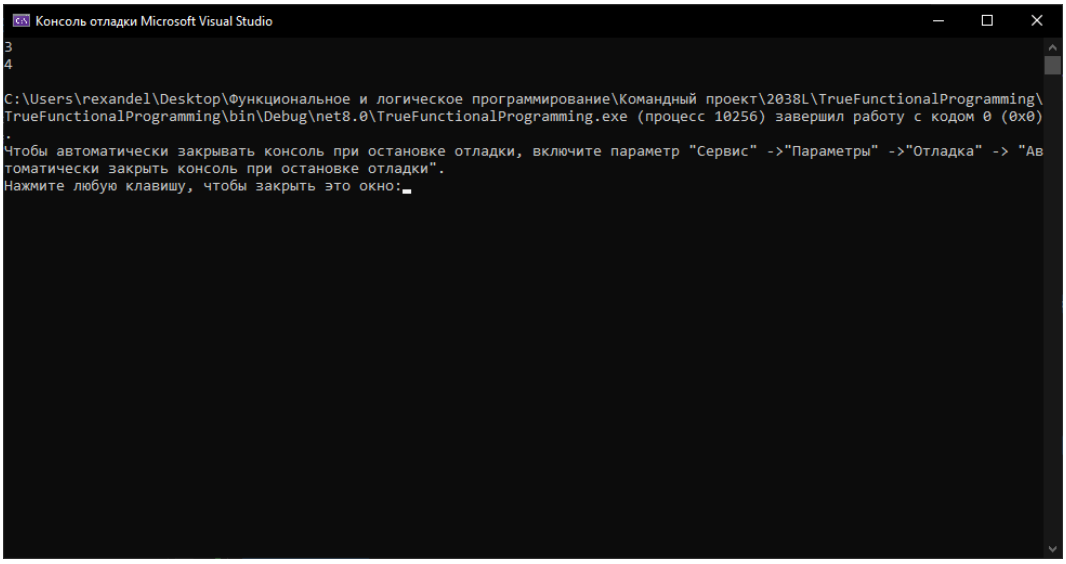
Функция minBoards определяет минимальное количество досок, необходимых для нарезки n кусков каждого типа. Сначала вычисляется верхняя граница количества досок, которая получается путем суммирования минимального количества досок, необходимых для каждого типа кусков в отдельности (это грубая оценка, но она гарантирует, что решение существует). Затем алгоритм перебирает все возможные количества досок от 1 до этой верхней границы, пока не найдет минимальное значение, при котором функция canCutBoards возвращает true.

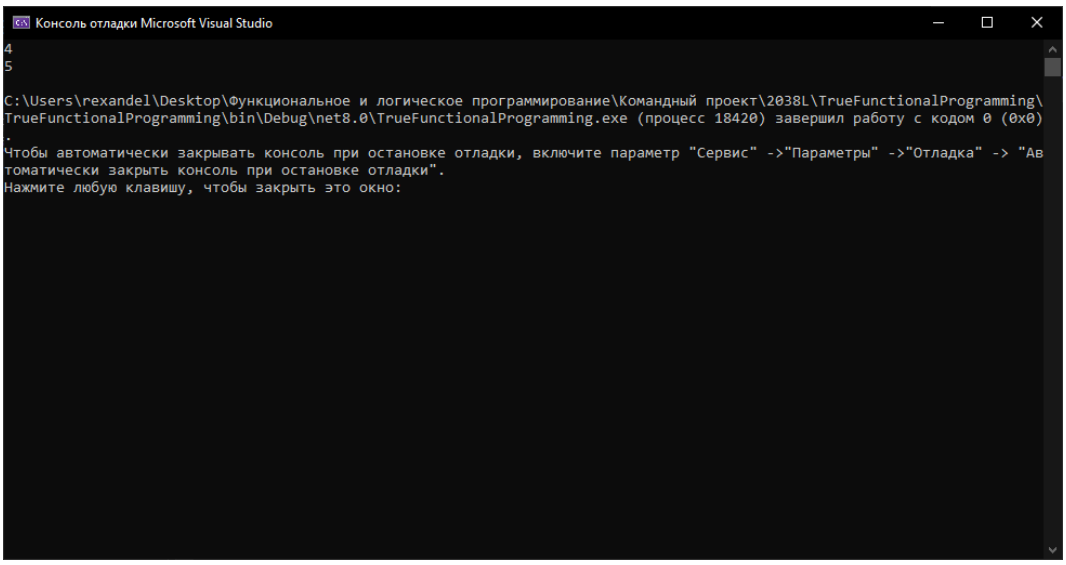
Основная программа (main) считывает входное значение n, вызывает функцию minBoards для вычисления результата и выводит его. Таким образом, алгоритм эффективно находит оптимальное решение, перебирая возможные варианты нарезки и проверяя их на соответствие требованиям.

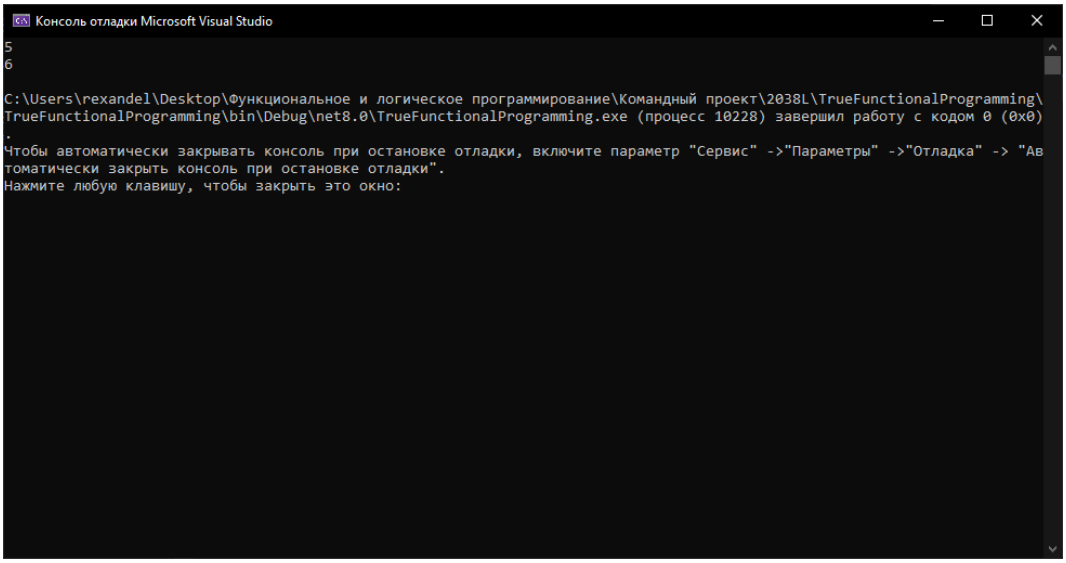
Проведение тестирования

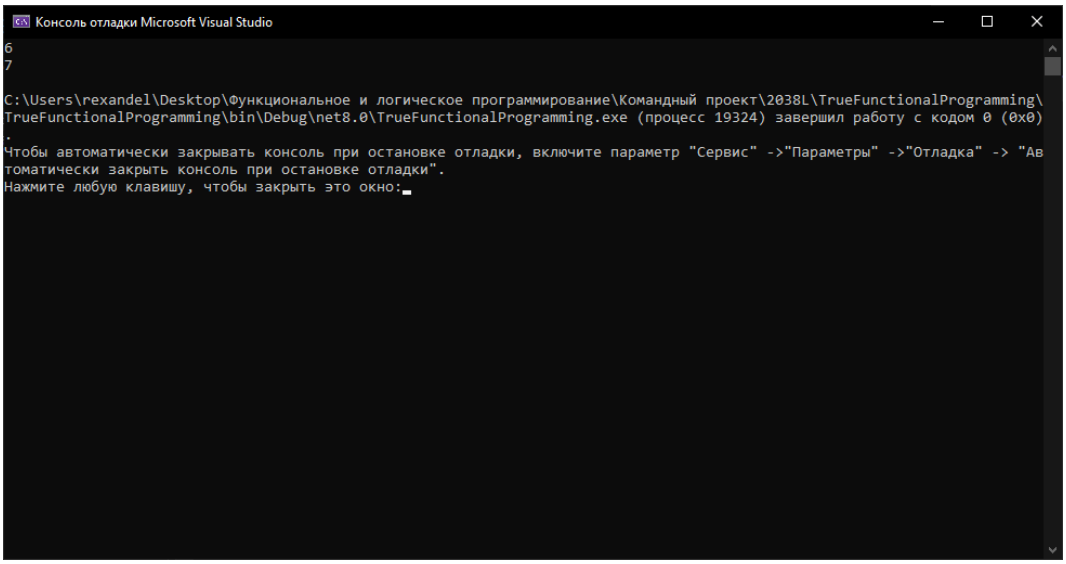







Консоль отладки Microsoft Visual Studio
4
5
C:\Users\rexandel\Desktop\Функциональное и логическое программирование\Командный проект\2038L\TrueFunctionalProgramming\TrueFunctionalProgramming\bin\Debug\net8.0\TrueFunctionalProgramming.exe (процесс 18420) завершил работу с кодом 0 (0x0).
Чтобы автоматически закрывать консоль при остановке отладки, включите параметр "Сервис" ->"Параметры" ->"Отладка" -> "Автоматически закрыть консоль при остановке отладки".
Нажмите любую клавишу, чтобы закрыть это окно:


Консоль отладки Microsoft Visual Studio
5
6
C:\Users\rexandel\Desktop\Функциональное и логическое программирование\Командный проект\2038L\TrueFunctionalProgramming\TrueFunctionalProgramming\bin\Debug\net8.0\TrueFunctionalProgramming.exe (процесс 10228) завершил работу с кодом 0 (0x0).
Чтобы автоматически закрывать консоль при остановке отладки, включите параметр "Сервис" ->"Параметры" ->"Отладка" -> "Автоматически закрыть консоль при остановке отладки".
Нажмите любую клавишу, чтобы закрыть это окно:


Консоль отладки Microsoft Visual Studio
6
7
C:\Users\rexandel\Desktop\Функциональное и логическое программирование\Командный проект\2038L\TrueFunctionalProgramming\TrueFunctionalProgramming\bin\Debug\net8.0\TrueFunctionalProgramming.exe (процесс 19324) завершил работу с кодом 0 (0x0).
Чтобы автоматически закрывать консоль при остановке отладки, включите параметр "Сервис" ->"Параметры" ->"Отладка" -> "Автоматически закрыть консоль при остановке отладки".
Нажмите любую клавишу, чтобы закрыть это окно:

Задача 2038J (Ожидание…)

Монокарп ждёт автобус на остановке. К сожалению, там много людей, которые тоже хотят уехать.

Вам дан список событий двух типов:

- b_i — на остановку прибывает автобус с b_i свободными местами;
- p_i — на остановку приходят p_i человек.

События перечислены в хронологическом порядке.

Когда прибывает автобус, происходит следующее:

1. Все люди на остановке (кроме Монокарпа) пытаются сесть в автобус.
    - Если свободных мест хватает на всех, то все садятся.
    - Если мест недостаточно, то в автобус садятся только bi человек, а остальные остаются ждать.
2. Если после посадки всех людей (кроме Монокарпа) в автобусе остаётся хотя бы одно свободное место, то Монокарп может решить сесть в этот автобус (но он может и подождать следующий).

Для каждого автобуса определите, возможно ли Монокарпу уехать на нём.

`Входные данные`

Первая строка содержит целое число n ($1 \le n \le 10^3$) — количество событий.

Далее следуют n строк. В i-й строке содержится описание i-го события в одном из двух форматов:

- bi ($1 \le bi \le 10^6$) — прибытие автобуса с bi свободными местами;
- pi ($1 \le pi \le 10^6$) — прибытие pi человек на остановку.

`Выходные данные`

Для каждого события типа B выведите YES, если Монокарп может сесть в этот автобус, иначе выведите NO (регистр не важен).

Решение на императивном языке C#

```
using System;

class Program
{
    static void Main()
    {
        long w = 0;
        int t = int.Parse(Console.ReadLine());

        for (int i = 0; i < t; i++)
        {
            string[] input = Console.ReadLine().Split();
            string s = input[0];
            long x = long.Parse(input[1]);

            if (s == "B")
            {
                w -= x;
                Console.WriteLine(w < 0 ? "YES" : "NO");
                w = w > 0 ? w : 0;
            }
            else
            {
                w += x;
            }
        }
    }
}
```

Описание решения на императивном языке

В начале работы программы создаётся переменная w, которая хранит текущее количество людей на остановке. Изначально она равна нулю, потому что в самом начале на остановке никого нет.

Затем программа считывает число t, которое обозначает общее количество событий. События могут быть двух типов: либо прибытие автобуса (обозначается буквой B), либо прибытие новых людей (обозначается буквой P).

Далее программа в цикле обрабатывает каждое событие. Для этого она считывает строку, в которой указан тип события и число. Например, строка B 5 означает, что прибыл автобус с пятью свободными местами, а строка P 3 означает, что на остановку пришли три новых человека.

Если событие — это прибытие людей (P), то программа увеличивает переменную w на указанное количество людей. Например, если на остановке уже было два человека (w = 2), а пришло ещё три (P 3), то теперь w станет равно пяти.

Если событие — это прибытие автобуса (B), то программа выполняет несколько действий. Сначала она вычитает из w количество свободных мест в автобусе. Например, если на остановке было пять человек (w = 5), а автобус приехал с тремя местами (B 3), то после вычитания останется два человека (w = 2).

После этого программа проверяет, остались ли в автобусе свободные места. Если после вычитания w стало меньше нуля, это значит, что в автобусе есть хотя бы одно свободное место, и Монокарп может сесть в него. В этом случае программа выводит YES. Если же w осталось больше или равно нулю, значит, все места заняты, и Монокарп не сможет уехать — программа выводит NO.

Наконец, программа корректирует значение w. Если после посадки в автобус остались люди (w > 0), они остаются ждать следующий автобус, и значение w сохраняется. Если же все люди уехали (w <= 0), то w сбрасывается в ноль, потому что на остановке больше никого нет.

Проверка работоспособности решения

Мои посылки

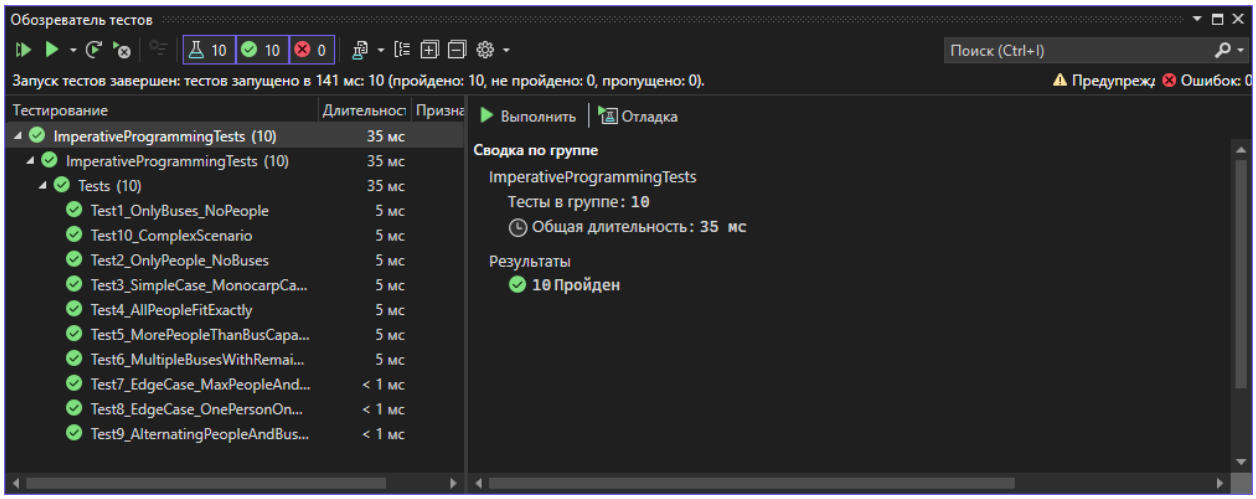
№	Когда	Кто	Задача	Язык	Вердикт	Время	Память
322001407	30.05.2025 08:21	rexandel	J - Waiting for...	C# 13	Полное решение	108 мс	0 КБ

Написание автотестов

Для проверки работы метода BusStopSimulator.ProcessEvents разработано 10 юнит-тестов с использованием MSTest.

Список тестов:

1. Только автобусы без людей
2. Только люди без автобусов
3. Простой случай с посадкой Монокарпа
4. Точное соответствие людей и мест
5. Людей больше, чем мест в автобусе
6. Несколько автобусов с оставшимися людьми
7. Граничный случай с максимальными значениями
8. Граничный случай с одним человеком и автобусом
9. Чередование людей и автобусов
10. Комплексный сценарий



Листинг автотестов

```
namespace ImperativeProgrammingTests
{
    [TestClass]
    public sealed class Tests
    {
        [TestMethod]
        public void Test1_OnlyBuses_NoPeople()
        {
            string input = "3\nB 2\nB 1\nB 3";
            string expected = "YES YES YES";
            string result = BusStopSimulator.ProcessEvents(input);
            Assert.AreEqual(expected, result);
        }

        [TestMethod]
        public void Test2_OnlyPeople_NoBuses()
        {
            string input = "2\nP 5\nP 3";
            string expected = "";
            string result = BusStopSimulator.ProcessEvents(input);
            Assert.AreEqual(expected, result);
        }

        [TestMethod]
        public void Test3_SimpleCase_MonocarpCanBoard()
        {
            string input = "4\nP 2\nB 3\nP 1\nB 2";
            string expected = "YES YES";
            string result = BusStopSimulator.ProcessEvents(input);
            Assert.AreEqual(expected, result);
        }

        [TestMethod]
        public void Test4_AllPeopleFitExactly()
        {
            string input = "3\nP 5\nB 5\nB 1";
            string expected = "NO YES";
            string result = BusStopSimulator.ProcessEvents(input);
            Assert.AreEqual(expected, result);
        }

        [TestMethod]
        public void Test5_MorePeopleThanBusCapacity()
        {
            string input = "3\nP 10\nB 4\nB 6";
            string expected = "NO NO";
            string result = BusStopSimulator.ProcessEvents(input);
            Assert.AreEqual(expected, result);
        }

        [TestMethod]
        public void Test6_MultipleBusesWithRemainingPeople()
        {
            string input = "5\nP 3\nB 2\nP 4\nB 3\nB 2";
            string expected = "NO NO NO";
            string result = BusStopSimulator.ProcessEvents(input);
            Assert.AreEqual(expected, result);
        }
```

```
        [TestMethod]
        public void Test7_EdgeCase_MaxPeopleAndBusCapacity()
        {
            string input = "2\nP 1000000\nB 1000000";
            string expected = "NO";
            string result = BusStopSimulator.ProcessEvents(input);
            Assert.AreEqual(expected, result);
        }

        [TestMethod]
        public void Test8_EdgeCase_OnePersonOneBus()
        {
            string input = "2\nP 1\nB 1";
            string expected = "NO";
            string result = BusStopSimulator.ProcessEvents(input);
            Assert.AreEqual(expected, result);
        }

        [TestMethod]
        public void Test9_AlternatingPeopleAndBuses()
        {
            string input = "6\nP 2\nB 1\nP 1\nB 2\nP 3\nB 3";
            string expected = "NO NO NO";
            string result = BusStopSimulator.ProcessEvents(input);
            Assert.AreEqual(expected, result);
        }

        [TestMethod]
        public void Test10_ComplexScenario()
        {
            string input = "7\nP 5\nB 3\nP 2\nB 4\nP 1\nB 2\nB 1";
            string expected = "NO NO YES YES";
            string result = BusStopSimulator.ProcessEvents(input);
            Assert.AreEqual(expected, result);
        }
    }
}
```

Решение на функциональном языке Prolog

```
% Предикат для обработки тестов
process_events(Input, ResultStr) :-
    % Разделяем входную строку на линии
    split_string(Input, "\n", "", [NLine|EventLines]),
    % Преобразуем количество событий в число
    atom_number(NLine, N),
    % Парсим события
    parse_events(EventLines, Events),
    % Решаем задачу
    solve(N, Events, Result),
    % Преобразуем результат в строку для assert_result
    atomic_list_concat(Result, ' ', ResultStr).

% Основной предикат для обработки задачи
solve(_N, Events, Result) :-
    process_events(Events, 0, [], Result).

% Обработка списка событий
% База: пустой список событий, возвращаем аккумулированный результат в обратном
порядке
process_events([], _, Acc, Result) :-
    reverse(Acc, Result).

% Событие: прибытие автобуса с Bi местами
process_events([b(Bi)|Rest], People, Acc, Result) :-
    % Проверяем, может ли Монокарп сесть
    (Bi > People -> CanBoard = 'YES' ; CanBoard = 'NO'),
    % Обновляем количество людей
    NewPeople is max(0, People - Bi),
    % Продолжаем обработку
    process_events(Rest, NewPeople, [CanBoard|Acc], Result).

% Событие: прибытие Pi человек
process_events([p(Pi)|Rest], People, Acc, Result) :-
    % Увеличиваем количество людей на остановке
    NewPeople is People + Pi,
    % Продолжаем обработку
    process_events(Rest, NewPeople, Acc, Result).

% Парсинг списка строк событий
parse_events([], []) :- !.
parse_events([Line|Rest], [Event|Events]) :-
    parse_event(Line, Event),
    parse_events(Rest, Events).

% Парсинг строки события: P N или B N
parse_event(Line, Event) :-
    split_string(Line, " ", "", [Type, NStr]),
    atom_number(NStr, N),
    ( Type = "P" -> Event = p(N)
    ; Type = "B" -> Event = b(N)
    ),
    !.
```

Описание решения функциональном языке Prolog

Основная идея заключалась в том, чтобы последовательно обрабатывать каждое событие из списка, отслеживая текущее количество людей на остановке. Для этого использовался аккумулятор, который передавался через рекурсивные вызовы. При обработке события типа "прибытие людей" значение аккумулятора увеличивалось на соответствующее количество человек.

Когда встречалось событие типа "прибытие автобуса", вычислялось, сколько людей сможет сесть в автобус. Если количество людей на остановке меньше или равно количеству мест, все люди уезжают, и проверялось, осталось ли хотя бы одно свободное место для Монокарпа. Если мест было больше, чем людей, выводилось "YES". В противном случае выводилось "NO".

Если же количество людей превышало количество мест в автобусе, то после посадки людей оставалось ровно столько, сколько не поместилось. В этом случае для Монокарпа места не оставалось, и выводилось "NO".

Рекурсивная обработка событий продолжалась до конца списка, после чего программа завершалась. Для удобства ввода и вывода использовались стандартные предикаты чтения и записи, а также преобразование строк в числа.

Проведение автотестов

Была использована та же логика, что и ранее для императивного языка.

```
?- run_tests.
[10/10] bus_stop:test10_complex_scenario ..........
% All 10 tests passed in 0.037 seconds (0.016 cpu)
true.
```

Решение на функциональном языке F#

```
open System

type Event =
    | Bus of int
    | People of int

let parseEvent (s: string) =
    let parts = s.Split(' ')
    match parts.[0] with
    | "B" -> Bus (int parts.[1])
    | "P" -> People (int parts.[1])
    | _ -> failwith "Unknown event type"

let processEvents events =
    let rec loop remainingPeople acc = function
        | [] -> List.rev acc
        | event :: rest ->
            match event with
            | People p -> loop (remainingPeople + p) acc rest
            | Bus b ->
                let newPeople = max 0 (remainingPeople - b)
                let canBoard = if b > remainingPeople then "YES" else "NO"
                loop newPeople (canBoard :: acc) rest
    loop 0 [] events

[<EntryPoint>]
let main argv =
    let n = Console.ReadLine() |> int
    let events =
        List.init n (fun _ -> Console.ReadLine())
        |> List.map parseEvent
    let results = processEvents events
    results |> List.iter (printfn "%s")
    0
```

Описание решения функциональном языке F#

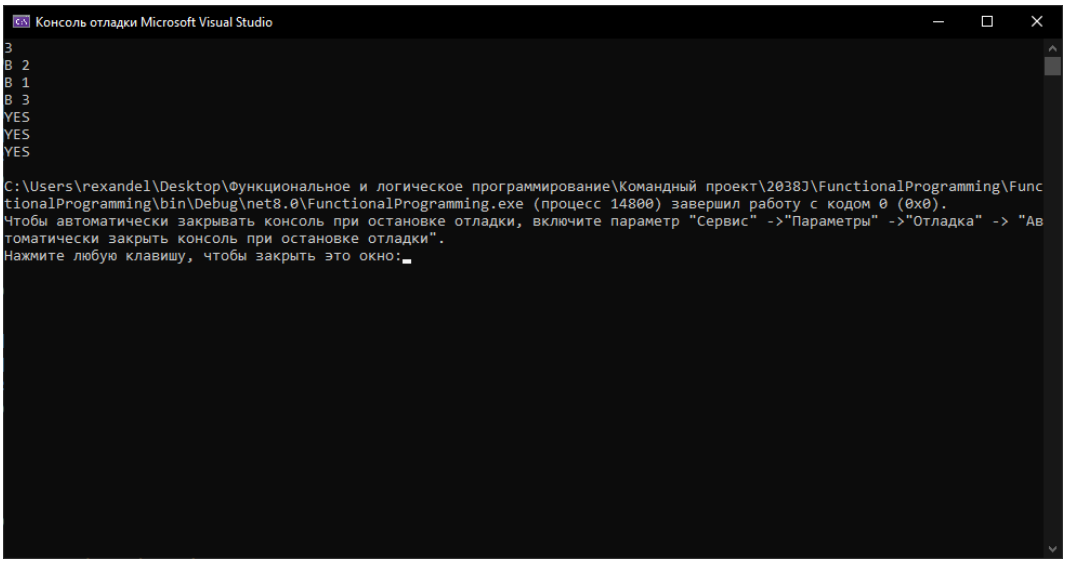
Для представления событий в программе определён дискриминированный союз Event, который моделирует два типа событий: прибытие автобуса с определённым количеством мест (Bus of int) и прибытие группы людей на остановку (People of int).

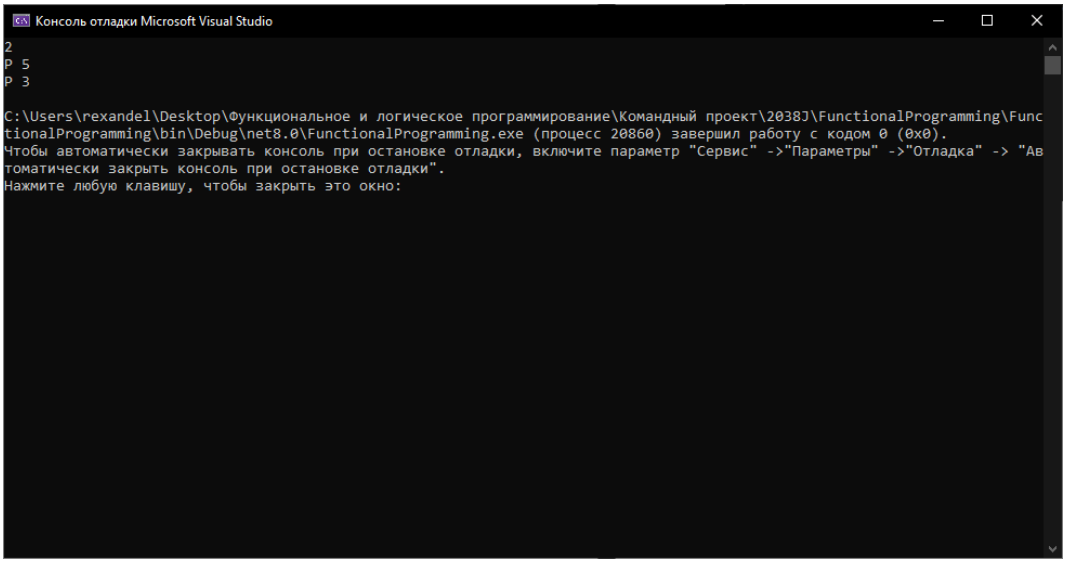
Функция parseEvent отвечает за преобразование строки ввода в значение типа Event. Каждая входная строка содержит два элемента: тип события («B» для автобуса или «P» для людей) и число. Функция разбивает строку по пробелу, выделяя тип события и числовое значение. Затем с помощью паттерн-матчинга она определяет, является ли событие Bus или People, преобразуя числовую часть строки в int с помощью функции int. Если встречается неизвестный тип события, функция вызывает исключение с помощью failwith.

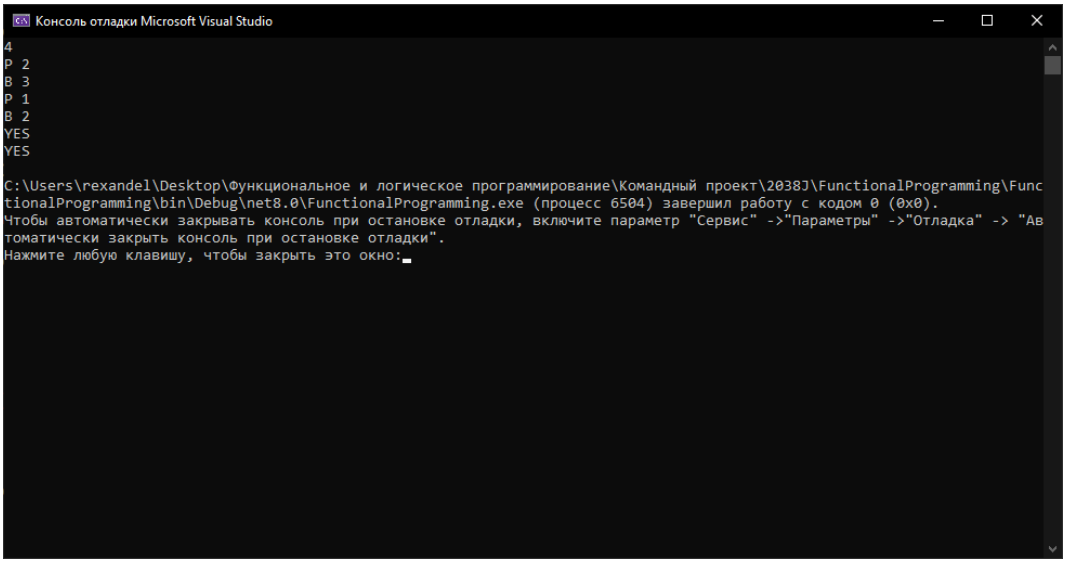
Основная логика решения реализована в функции processEvents, которая обрабатывает список событий и определяет, может ли Монокарп сесть на каждый автобус. Внутри этой функции определена вспомогательная рекурсивная функция loop, принимающая три параметра: remainingPeople (текущее количество людей, ожидающих на остановке), acc (аккумулятор для хранения результатов) и оставшиеся события для обработки. Использование вложенной рекурсивной функции — это стандартный функциональный подход, позволяющий изолировать рекурсивную логику и сохранить чистоту внешней функции. Функция loop реализована как хвостовая рекурсия, что обеспечивает эффективное использование памяти.

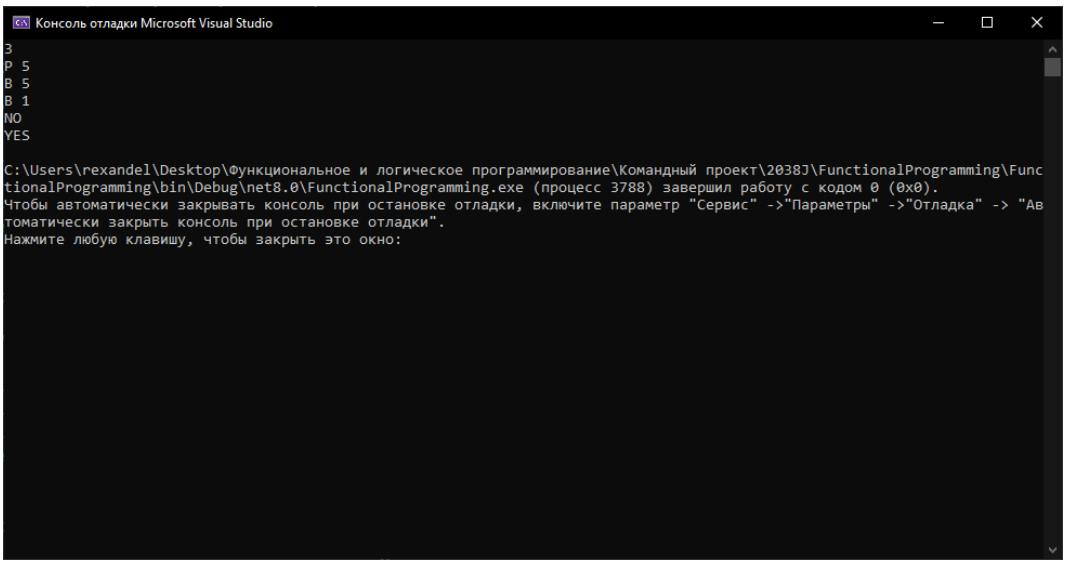
В функции loop используется паттерн-матчинг для обработки базового и рекурсивных случаев. Если список событий пуст, функция возвращает аккумулятор, предварительно перевернув его с помощью List.rev. Переворот необходим, поскольку результаты добавляются в начало аккумулятора в процессе рекурсии, что приводит к обратному порядку. Если список событий не пуст, функция анализирует первое событие. Для события People p она увеличивает количество ожидающих людей (remainingPeople) на p и рекурсивно вызывает loop с обновлённым количеством, тем же аккумулятором и оставшимися событиями. Для события Bus b функция вычисляет, может ли Монокарп сесть в автобус, проверяя, превышает ли количество мест (b) количество ожидающих людей (remainingPeople). Если мест больше, результат — «YES», иначе — «NO». Затем она вычисляет новое количество ожидающих людей, вычитая количество мест из текущего числа ожидающих, но не позволяя значению стать отрицательным (с помощью max 0). Результат для текущего автобуса добавляется в аккумулятор, и рекурсия продолжается с оставшимися событиями.

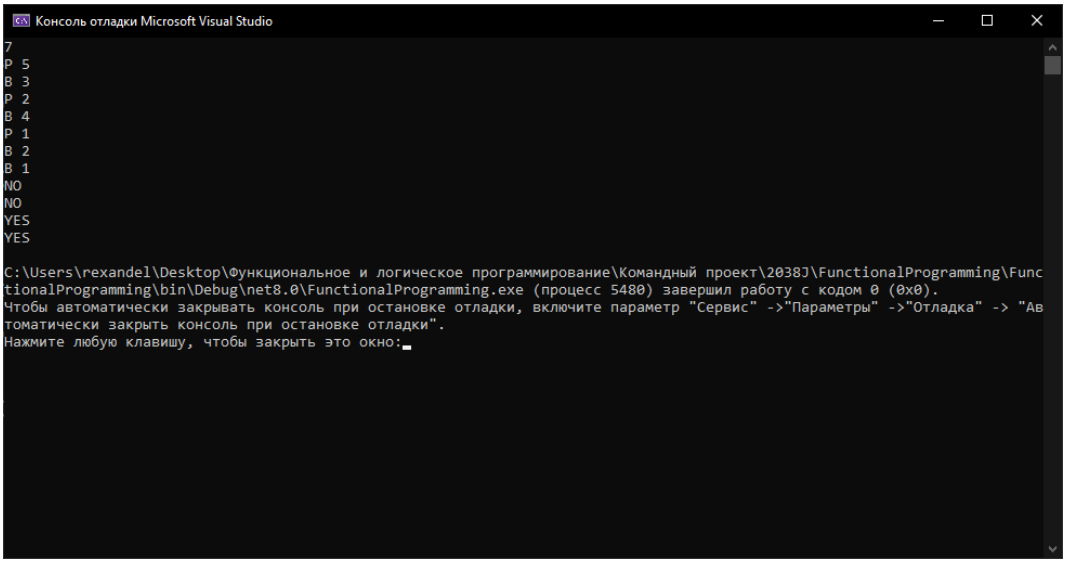
Проведение тестирования

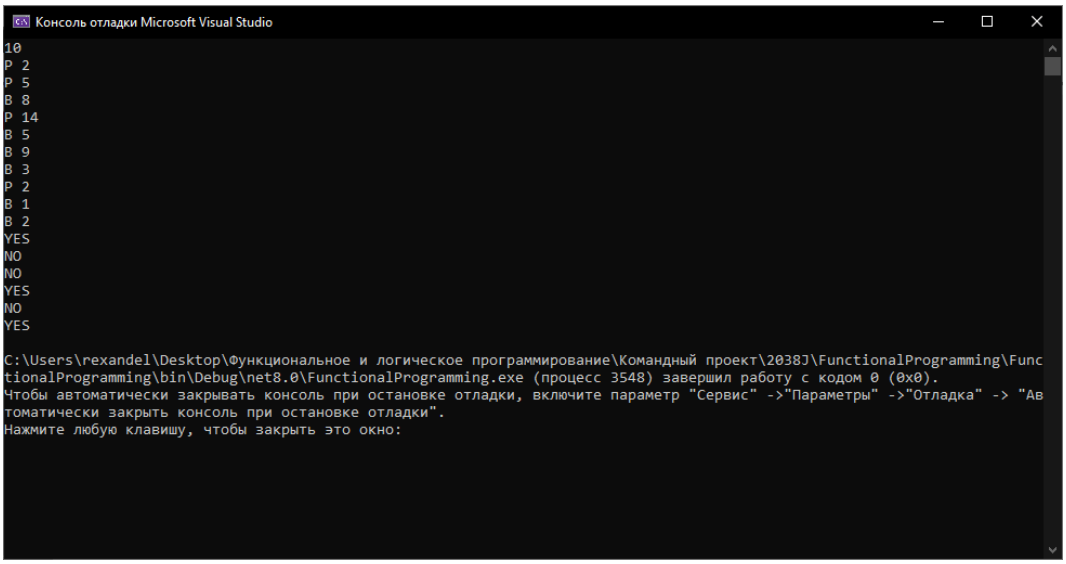







Консоль отладки Microsoft Visual Studio
3
P 5
B 5
B 1
NO
YES
C:\Users\rexandel\Desktop\Функциональное и логическое программирование\Командный проект\2038J\FunctionalProgramming\FunctionalProgramming\bin\Debug\net8.0\FunctionalProgramming.exe (процесс 3788) завершил работу с кодом 0 (0x0).
Чтобы автоматически закрывать консоль при остановке отладки, включите параметр "Сервис" ->"Параметры" ->"Отладка" -> "Автоматически закрыть консоль при остановке отладки".
Нажмите любую клавишу, чтобы закрыть это окно:


Консоль отладки Microsoft Visual Studio
7
P 5
B 3
P 2
B 4
P 1
B 2
B 1
NO
NO
YES
YES
C:\Users\rexandel\Desktop\Функциональное и логическое программирование\Командный проект\2038J\FunctionalProgramming\FunctionalProgramming\bin\Debug\net8.0\FunctionalProgramming.exe (процесс 5480) завершил работу с кодом 0 (0x0).
Чтобы автоматически закрывать консоль при остановке отладки, включите параметр "Сервис" ->"Параметры" ->"Отладка" -> "Автоматически закрыть консоль при остановке отладки".
Нажмите любую клавишу, чтобы закрыть это окно:


Консоль отладки Microsoft Visual Studio
10
P 2
P 5
B 8
P 14
B 5
B 9
B 3
P 2
B 1
B 2
YES
NO
NO
YES
NO
YES
C:\Users\rexandel\Desktop\Функциональное и логическое программирование\Командный проект\2038J\FunctionalProgramming\FunctionalProgramming\bin\Debug\net8.0\FunctionalProgramming.exe (процесс 3548) завершил работу с кодом 0 (0x0).
Чтобы автоматически закрывать консоль при остановке отладки, включите параметр "Сервис" ->"Параметры" ->"Отладка" -> "Автоматически закрыть консоль при остановке отладки".
Нажмите любую клавишу, чтобы закрыть это окно: