Министерство науки и высшего образования Российской Федерации

федеральное государственное автономное

образовательное учреждение высшего образования

«Национальный исследовательский университет

«Московский институт электронной техники»

институт системной и программной инженерии

и информационных технологий»

Курс «Теория алгоритмических языков и компиляторов»

Лабораторная работа № 5

Разработка интерпретатора.

Выполнил:

Студент группы ПИН-51Д

Джугели Дмитрий Александрович

Москва, 2025

Задание на лабораторную работу

Создать интерпретатор языка CBAS.

Входные данные: исходный текст программы на языке CBAS.

Выходные данные: консоль с результатами ввода-вывода исходной программы, либо сообщения об ошибках, присутствующих в программе

Код:

Класс Tokenizer разбивает исходную строку кода на токены (числа, операторы, ключевые слова, переменные)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text.RegularExpressions;

namespace Interpreter
{
    public class Tokenizer
    {
        public IEnumerable<Token> Tokenize(string code)
        {
            var allRules = Rules.GetAllRules();
            var regexPattern = String.Join("|", allRules.Select(x =>
            {
                if ("()*+".Contains(x))
                    return $"({@"\" + x})";
                return $"({x})";
            }));
            var regex = Regex.Matches(code, regexPattern, RegexOptions.Singleline);

            foreach (Match item in regex)
            {
                var token = item.Value;

                if (String.IsNullOrWhiteSpace(token)) { yield return new Token(token,
TokenType.Space); continue; }
                if (Rules.Functions.Contains(token)) { yield return new Token(token,
TokenType.Function); continue; }
                if (Rules.Sintaxis.Contains(token)) { yield return new Token(token,
TokenType.Sintaxis); continue; }
                if (Regex.Match(token, Rules.QuoteString).Success) { yield return new
Token(token, TokenType.QuoteString); continue; }
                if (Regex.Match(token, Rules.Digit).Success) { yield return new Token(token,
TokenType.Digit); continue; }
                if (Regex.Match(token, Rules.Character).Success) { yield return new
Token(token, TokenType.Character); continue; }
                if (Rules.Operators.Contains(token)) { yield return new Token(token,
TokenType.Operator); continue; }
                if (Rules.BoolOperations.Contains(token)) { yield return new Token(token,
TokenType.BoolOperation); continue; }
                if (Rules.Equal == (token)) { yield return new Token(token, TokenType.Equal);
continue; }
                if (Rules.Brackets.Contains(token)) { yield return new Token(token,
TokenType.Bracket); continue; }
                if (Rules.Punctuation.Contains(token)) { yield return new Token(token,
TokenType.Punctuation); continue; }
            }
        }

        public class Token
        {
            public Token(string token, TokenType identity)
            {
                TokenString = token;
                Type = identity;
            }
```

```
            public string TokenString { get; }
            public TokenType Type { get; }

            public override string ToString()
            {
                return $"{TokenString}|{Type.ToString()}";
            }
        }

        public static class Rules
        {
            static Rules()
            {
                Digit = "\\d+";
                Character = "[a-zA-Z_]+";
                Operators = "+-*/".Select(x => x.ToString()).ToArray();
                BoolOperations = new[] { "<", ">", "==", "!=" };
                Equal = "=";
                Brackets = "(){}".Select(x => x.ToString()).ToArray();
                QuoteString = "\"(.*?)\"";
                Punctuation = ";,".Select(x => x.ToString()).ToArray();
                Space = "[\n\r\t ]+";
                Functions = new[] { "scan", "print" };
                Sintaxis = new[] { "for", "to", "if", "else" };
            }

            public static IEnumerable<string> GetAllRules()
            {
                var list = new List<string>();
                list.AddRange(Functions);
                list.AddRange(Sintaxis);
                list.Add(QuoteString);
                list.Add(Digit);
                list.Add(Character);
                list.AddRange(Operators);
                list.AddRange(BoolOperations);
                list.Add(Equal);
                list.AddRange(Brackets);
                list.AddRange(Punctuation);
                list.Add(Space);
                return list;
            }

            public static string Digit { get; }
            public static string Character { get; }
            public static string[] Operators { get; }
            public static string[] BoolOperations { get; }
            public static string Equal { get; }
            public static string[] Brackets { get; }
            public static string QuoteString { get; }
            public static string[] Punctuation { get; }
            public static string Space { get; }
            public static string[] Functions { get; }
            public static string[] Sintaxis { get; }
        }

        public enum TokenType
        {
            Digit,
            Character,
            Operator,
            BoolOperation,
            Equal,
            Bracket,
            Punctuation,
```

```
            QuoteString,
            Space,
            Function,
            Sintaxis
        }
    }
}
```

Класс Expression сложение и вычитание

```
using static Interpreter.Tokenizer;

namespace Interpreter
{
    public class Expression
    {
        public static int Parse(Interpreter.Context context)
        {
            var _context = context;
            var _stack = context.Tokens;
            var termResult = Term.Parse(_context);

            if (_stack.Count <= 0)
                return termResult;

            var @operator = _stack.Peek();
            if (@operator.Type == TokenType.Operator)
            {
                if (@operator.TokenString == "+")
                {
                    _stack.Pop();
                    return termResult + Parse(_context);
                }
                else if (@operator.TokenString == "-")
                {
                    _stack.Pop();
                    if (_stack.Count > 1)
                    {
                        var next = _stack.Pop();
                        var nextOperation = _stack.Peek();
                        _stack.Push(next);
                        bool isNextMultOperation = nextOperation.TokenString == "*" ||
nextOperation.TokenString == "/";
                        if ((next.Type == TokenType.Digit || next.Type == TokenType.Character)
&& !isNextMultOperation)
                        {
                            var result = termResult - Factor.Parse(_context);
                            var newToken = new Token(result.ToString(), TokenType.Digit);
                            _stack.Push(newToken);
                            return Parse(_context);
                        }
                    }
                    return termResult - Parse(_context);
                }
            }
            return termResult;
        }
    }
}
```

Класс Term умножение и деление

```
using static Interpreter.Tokenizer;

namespace Interpreter
{
    public class Term
    {
        public static int Parse(Interpreter.Context context)
        {
            var _context = context;
            var _stack = context.Tokens;
            var factorResult = Factor.Parse(context);

            if (_stack.Count <= 0)
                return factorResult;

            var @operator = _stack.Peek();
            if (@operator.Type == TokenType.Operator)
            {
                if (@operator.TokenString == "*")
                {
                    _stack.Pop();
                    return factorResult * Parse(_context);
                }
                else if (@operator.TokenString == "/")
                {
                    _stack.Pop();
                    return factorResult / Parse(_context);
                }
            }
            return factorResult;
        }
    }
}
```

Класс Factor обработка чисел, переменных и скобок

```
using System;
using static Interpreter.Tokenizer;

namespace Interpreter
{
    public class Factor
    {
        public static int Parse(Interpreter.Context context)
        {
            var _context = context;
            var _stack = context.Tokens;
            var _table = context.Variables;
            var token = _stack.Pop();
            var _type = token.Type;
            var _string = token.TokenString;
            if (_type == TokenType.Digit)
            {
                return Convert.ToInt32(_string);
            }
```

```
            if (_type == TokenType.Character)
            {
                var isInTable = _table.ContainsKey(_string);
                if (!isInTable)
                    throw new Exception($"Unexpected token {token.TokenString}.");
                return _table[_string];
            }

            if (_type == TokenType.Bracket && _string == "(")
            {
                var result = Expression.Parse(_context);
                ExpressionsHellper.CheckStack(_context);
                var closeBracket = _stack.Pop();
                if (closeBracket.TokenString == ")")
                    return result;
            }
            throw new Exception($"Unexpected simbol {_string}.");
        }
    }
}
```

Класс Interpreter основной класс – управление процессом

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using static Interpreter.Tokenizer;

namespace Interpreter
{
   public class Interpreter
   {
      public void Interpret(string code)
      {
         var tokinizer = new Tokenizer();
         var tokens = tokinizer.Tokenize(code);
         var onlyTokens = tokens.Where(x => x.Type != TokenType.Space);
         var context = new Context(onlyTokens);
         Interpret(context);
      }

      private void Interpret(Context context)
      {
         try
         {
            while (context.Tokens.Count > 0)
               Statement(context);
```

```
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.Message);
        }
    }

    private void Statement(Context context)
    {
        if (context.Tokens.Count == 0)
            return;
        var tokenFull = context.Tokens.Pop();
        var type = tokenFull.Type;
        var token = tokenFull.TokenString;
        if (token == "if")
        {
            If(context);
        }
        else if (token == "for")
        {
            For(context);
        }
        else if (token == "print")
        {
            Print(context);
        }
        else if (token == "scan")
        {
            Scan(context);
        }
        else if (type == TokenType.Character)
        {
            context.Tokens.Push(tokenFull);
            Assign(context);
        }
        else
            throw ExpressionsHellper.ThrowUnexpectedToken(tokenFull);
    }

    private void Assign(Context context)
    {
        ExpressionsHellper.CheckStack(context);
        var identifier = context.Tokens.Pop();
        var equalOperator = context.Tokens.Pop();
        if (equalOperator.Type != TokenType.Equal)
```

```
            throw ExpressionsHellper.ThrowUnexpectedToken(equalOperator);
        var expression = Expression.Parse(context);
        context.Variables[identifier.TokenString] = expression;
    }

    private void Scan(Context context)
    {
        ExpressionsHellper.CheckStack(context);
        var token = context.Tokens.Pop();
        if (token.Type != TokenType.Character)
            throw new Exception($"Invalid simbol for \"scan\"
{token.TokenString}.");
        var value = Int32.Parse(Console.ReadLine());
        context.Variables[token.TokenString] = value;
        ExpressionsHellper.CheckStack(context);
        token = context.Tokens.Pop();
        if (token.TokenString != ";")
            throw new Exception($"Invalid simbol for \"scan\"
{token.TokenString}.");
    }

    private void Print(Context context)
    {
        var result = PrintEnd(context);
        Console.WriteLine(result);
    }

    private string PrintEnd(Context context)
    {
        var stack = context.Tokens;
        ExpressionsHellper.CheckStack(context);
        var token = stack.Pop();
        if (token.Type == TokenType.QuoteString)
        {
            if (stack.Count > 0 && stack.Peek().TokenString == ",")
            {
                stack.Pop();
                return token.TokenString + " " + PrintEnd(context);
            }
            else
                return token.TokenString;
        }
        else if (token.Type == TokenType.Character || token.Type ==
TokenType.Digit)
        {
```

```
            context.Tokens.Push(token);
            var expression = Expression.Parse(context).ToString();
            bool isNextComma = stack.Count > 0 && stack.Peek().TokenString ==
",";
            if (stack.Count > 0 && isNextComma)
            {
               stack.Pop();
               return expression + " " + PrintEnd(context);
            }
            else
               return expression;
         }
         else
            throw new Exception("Invalid <print_end>.");
      }

      private void For(Context context)
      {
         ExpressionsHellper.CheckStack(context);
         var stack = context.Tokens;
         var charecter = stack.Pop();
         if (charecter.Type != TokenType.Character)
            throw ExpressionsHellper.ThrowUnexpectedToken(charecter);
         ExpressionsHellper.CheckStack(context);
         var equal = stack.Pop();
         if (equal.Type != TokenType.Equal)
            throw ExpressionsHellper.ThrowUnexpectedToken(equal);
         var left = Expression.Parse(context);
         ExpressionsHellper.CheckStack(context);
         var to = stack.Pop();
         if (to.TokenString != "to")
            throw ExpressionsHellper.ThrowUnexpectedToken(equal);
         var right = Expression.Parse(context);
         context.Variables[charecter.TokenString] = left;
         ForLoop(context, charecter, right);
      }

      private void ForLoop(Context context, Token charecter, int right)
      {
         var queue = context.Tokens;
         var commands = new List<Token>();
         int bracketCount = 1;
         var token = queue.Pop();
         if (token.TokenString != "{")
            throw ExpressionsHellper.ThrowUnexpectedToken(token);
```

```
        commands.Add(token);
        //take commands for For;
        while (queue.Count > 0 && bracketCount != 0)
        {
            token = queue.Pop();
            commands.Add(token);
            if (token.TokenString == "{")
                bracketCount++;
            if (token.TokenString == "}")
                bracketCount--;
        }
        if (bracketCount > 0)
            throw new Exception("Not enough }.");

        while (context.Variables[charecter.TokenString] < right)
        {
            var forContext = new Context(commands);
            forContext.Variables = context.Variables;
            BracketStatement(forContext);
            context.Variables[charecter.TokenString]++;
        }
    }

    private void If(Context context)
    {
        var queue = context.Tokens;
        if (IfBool(context))
        {
            BracketStatement(context);
            if (queue.Peek().TokenString == "else")
            {
                queue.Pop();
                var token = queue.Pop();
                int bracketCount = 1;
                if (token.TokenString != "{")
                    throw ExpressionsHellper.ThrowUnexpectedToken(token);
                while (queue.Count > 0 && bracketCount != 0)
                {
                    token = queue.Pop();
                    if (token.TokenString == "{")
                        bracketCount++;
                    if (token.TokenString == "}")
                        bracketCount--;
                }
                if (bracketCount > 0)
```

```
                throw new Exception("Not enough }.");
            }
        }
        else
        {
            int bracketCount = 1;
            var token = queue.Pop();
            if (token.TokenString != "{")
                throw ExpressionsHellper.ThrowUnexpectedToken(token);
            while (queue.Count > 0 && bracketCount != 0)
            {
                token = queue.Pop();
                if (token.TokenString == "{")
                    bracketCount++;
                if (token.TokenString == "}")
                    bracketCount--;
            }
            if (bracketCount > 0)
                throw new Exception("Not enough }.");
            Else(context);
        }
    }

    private void BracketStatement(Context context)
    {
        var queue = context.Tokens;
        ExpressionsHellper.CheckStack(context);
        var token = queue.Pop();
        if (token.TokenString != "{")
            throw ExpressionsHellper.ThrowUnexpectedToken(token);
        while (queue.Count > 0 && queue.Peek().TokenString != "}")
            Statement(context);
        ExpressionsHellper.CheckStack(context);
        token = queue.Pop();
        if (token.TokenString != "}")
            throw ExpressionsHellper.ThrowUnexpectedToken(token);
    }

    private void Else(Context context)
    {
        var queue = context.Tokens;
        if (queue.Count == 0)
            return;
        var token = queue.Pop();
        if (token.TokenString == "else")
```

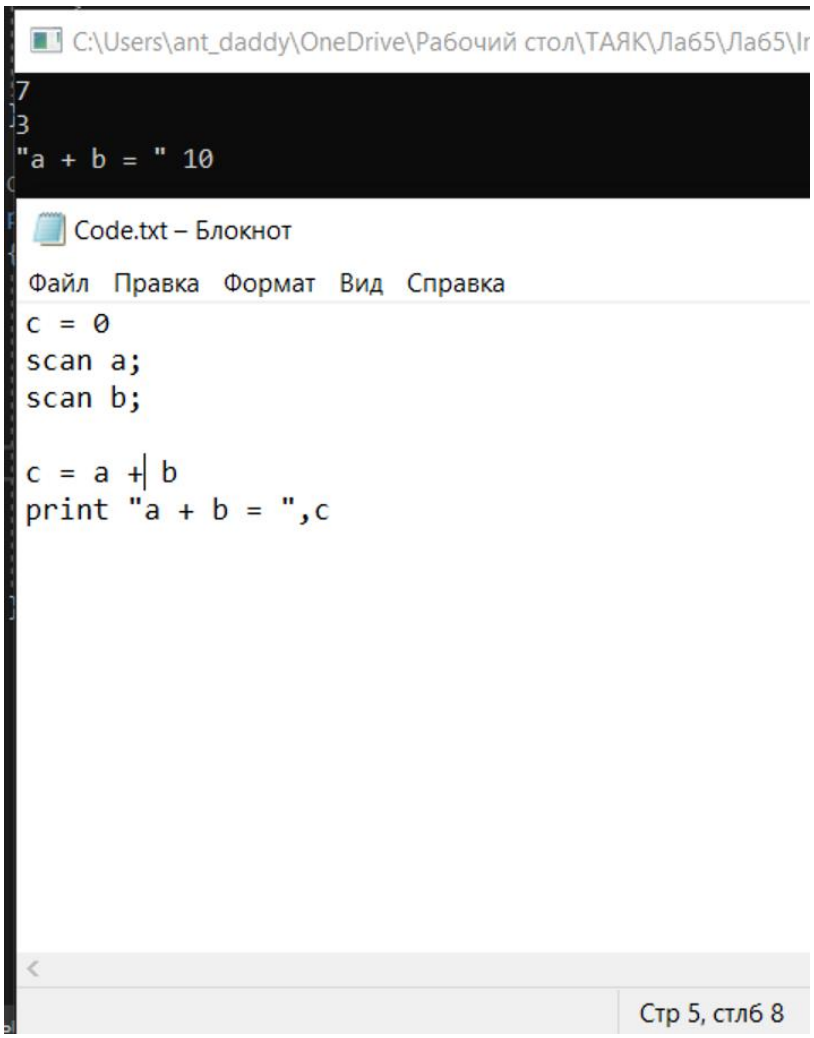
```
        {
            BracketStatement(context);
        }
    }

    private bool IfBool(Context context)
    {
        ExpressionsHellper.CheckStack(context);
        var left = Expression.Parse(context);
        ExpressionsHellper.CheckStack(context);
        var token = context.Tokens.Pop();
        var rigth = Expression.Parse(context);
        if (token.TokenString == ">")
        {
            return left > rigth;
        }
        else if (token.TokenString == "<")
        {
            return left < rigth;
        }
        else if (token.TokenString == "==")
        {
            return left == rigth;
        }
        else if (token.TokenString == "!=")
        {
            return left != rigth;
        }
        else
            throw ExpressionsHellper.ThrowUnexpectedToken(token);
    }

    public class Context
    {
        public Dictionary<string, int> Variables;
        public Stack<Token> Tokens;

        public Context(IEnumerable<Token> tokens)
        {
            Tokens = new Stack<Token>(tokens.Reverse());
            Variables = new Dictionary<string, int>();
        }
    }
  }
}
```

Примеры работы:



C:\Users\ant_daddy\OneDrive\Рабочий стол\ТАЯК\Лаб5\Л

```
"2 + 3 * 4 = " 14
"(2 + 3) * 4 = " 20
```

Code.txt – Блокнот

Файл Правка Формат Вид Справка

```
a = 2 + 3 * 4
b = (2 + 3) * 4
print "2 + 3 * 4 = ", a
print "(2 + 3) * 4 = ", b
```

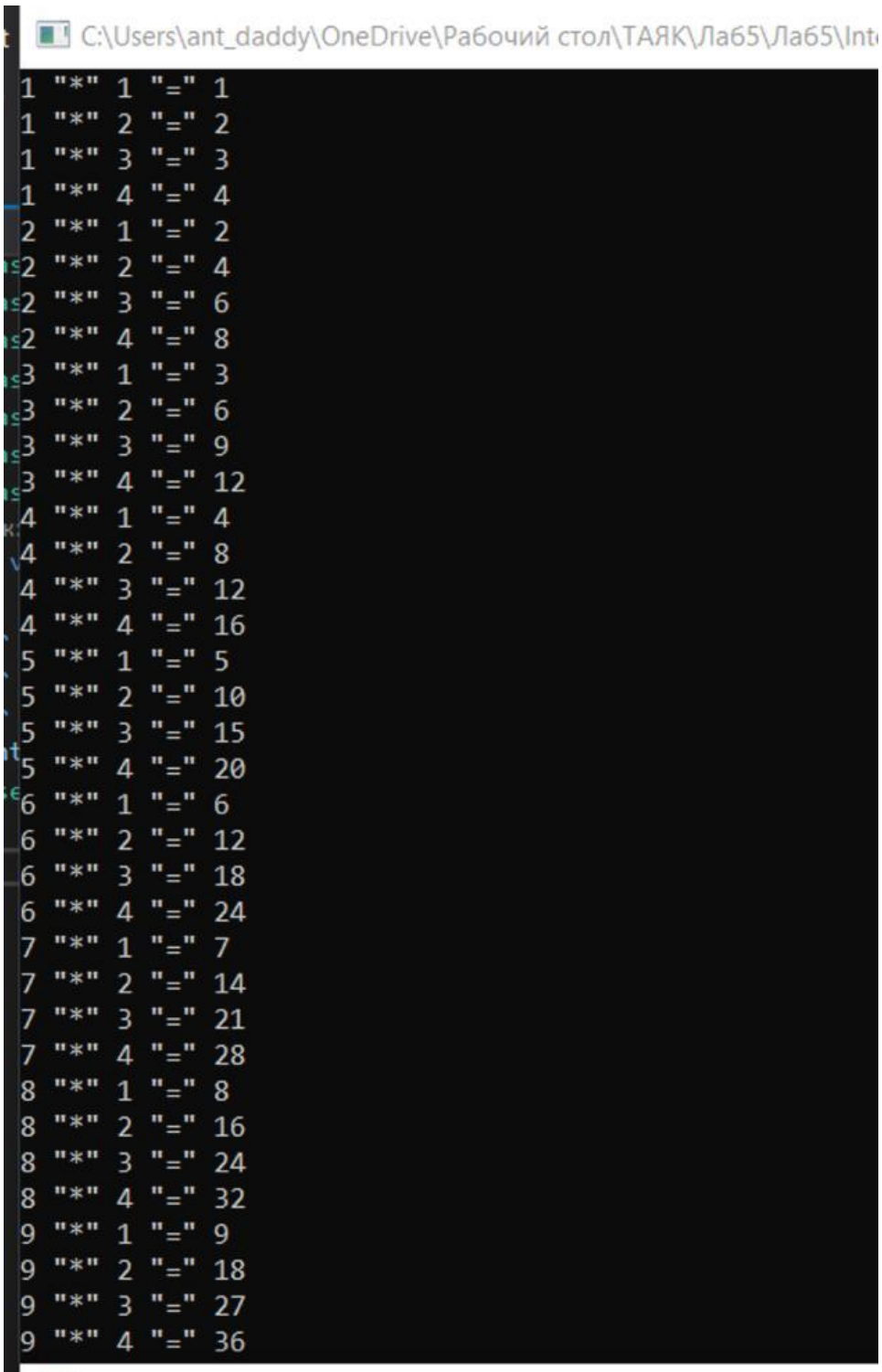
C:\Users\ant_daddy\OneDrive\Рабочий стол\ТАЯК

```
"i=" 1 "sum=" 1
"i=" 2 "sum=" 3
"i=" 3 "sum=" 6
"i=" 4 "sum=" 10
"result sum: " 10
```

Code.txt – Блокнот

Файл Правка Формат Вид Справка

```
sum = 0
for i = 1 to 5 {
    sum = sum + i
    print "i=", i, "sum=", sum
}
print "result sum: ", sum
```



Code.txt – Блокнот

Файл Правка Формат Вид Справка

```
for i = 1 to 10 {
    for j = 1 to 5 {
        result = i * j
        print i, "*", j, "=", result
    }
}
```

C:\Users\ant_daddy\OneDrive\Рабочий стол\TA

```
"a не больше b"
"a равно 10"
"b равно 20"
```

Code.txt – Блокнот

Файл Правка Формат Вид Справка

```
a = 10
b = 20

if a > b {
    print "a больше b"
} else {
    print "a не больше b"
}

if a == 10 {
    print "a равно 10"
}

if b != 20 {
    print "b не равно 20"
} else {
    print "b равно 20"
}
```

C:\Users\ant_daddy\OneDrive\Рабочий стол\ТАЯК\Лаб5\Лаб5\Interpreter\Inter

```
"Введите начальное число:"
1
"Введите конечное число:"
5
"Сумма чисел от " 1 " до " 5 " = " 10
```

Code.txt – Блокнот

Файл Правка Формат Вид Справка

```
print "Введите начальное число:"
scan start;
print "Введите конечное число:"
scan end;

sum = 0
for i = start to end {
    sum = sum + i
}

print "Сумма чисел от ", start, " до ", end, " = ", sum
```

Code.txt – Блокнот

Файл Правка Формат Вид Справка

```
a = 10
b = 3
c = a / b
d = 7 / 2
print "10 / 3 = ", c
print "7 / 2 = ", d
```