|  |  |
| --- | --- |
|  |  |
| МИНОБРНАУКИ РОССИИ | | |
| Федеральное государственное бюджетное образовательное учреждение  высшего профессионального образования  **«Московский технологический университет»**  **МИРЭА** | | |

Институт Информационных Технологий

Кафедра Корпоративных Информационных Систем

**ОТЧЕТ**

по Лабораторной Работе №4

на тему

«Синтаксическое дерево. Алгоритм сортировочной станции.»

по дисциплине

«ООП»

Выполнил студент группы ИСБО-11-16 Шайхуллин С.В.

Принял Cтарший преподаватель Мирзоян Д.И.

Выполнено «\_\_\_» \_\_\_\_\_\_\_\_\_2017 г.

Зачтено «\_\_\_» \_\_\_\_\_\_\_\_\_ 2017г.

Москва, 2017

**Теоретическая часть**

Простой интерпретатор анализирует и тут же выполняет (собственно интерпретация) программу покомандно (или построчно), по мере поступления её исходного кода на вход интерпретатора. Достоинством такого подхода является мгновенная реакция. Недостаток — такой интерпретатор обнаруживает ошибки в тексте программы только при попытке выполнения команды (или строки) с ошибкой.

Интерпретатор компилирующего типа — это система из компилятора, переводящего исходный код программы в промежуточное представление, например, в байт-код или p-код, и собственно интерпретатора, который выполняет полученный промежуточный код (так называемая виртуальная машина). Достоинством таких систем является большее быстродействие выполнения программ (за счёт выноса анализа исходного кода в отдельный, разовый проход, и минимизации этого анализа в интерпретаторе). Недостатки — большее требование к ресурсам и требование на корректность исходного кода. Применяется в таких языках, как Java, PHP, Tcl, Perl, REXX (сохраняется результат парсинга исходного кода[5]), а также в различных СУБД.

В случае разделения интерпретатора компилирующего типа на компоненты получаются компилятор языка и простой интерпретатор с минимизированным анализом исходного кода. Причём исходный код для такого интерпретатора не обязательно должен иметь текстовый формат или быть байт-кодом, который понимает только данный интерпретатор, это может быть машинный код какой-то существующей аппаратной платформы. К примеру, виртуальные машины вроде QEMU, Bochs, VMware включают в себя интерпретаторы машинного кода процессоров семейства x86.

Некоторые интерпретаторы (например, для языков Лисп, Scheme, Python, Бейсик и других) могут работать в режиме диалога или так называемого цикла чтения-вычисления-печати (англ. read-eval-print loop, REPL). В таком режиме интерпретатор считывает законченную конструкцию языка (например, s-expression в языке Лисп), выполняет её, печатает результаты, после чего переходит к ожиданию ввода пользователем следующей конструкции.

Уникальным является язык Forth, который способен работать как в режиме интерпретации, так и компиляции входных данных, позволяя переключаться между этими режимами в произвольный момент, как во время трансляции исходного кода, так и во время работы программ.[6]

Следует также отметить, что режимы интерпретации можно найти не только в программном, но и аппаратном обеспечении. Так, многие микропроцессоры интерпретируют машинный код с помощью встроенных микропрограмм, а процессоры семейства x86, начиная с Pentium (например, на архитектуре Intel P6), во время исполнения машинного кода предварительно транслируют его во внутренний формат (в последовательность микроопераций).

**Задание**

Релизовать приложение, вычисляющее значение выражения по его символьной записи.

**Алгоритм решения задачи**

Приложение должно принимать на вход математическое выражение в его текстовой записи. Должны быть реализованы вещественный и логический типы данных. Преобразование логического типа в вещественный — неяное, истина = 1.0, ложь = 0.0. Обратное преобразование запрещено. Необходимо реализовать следующие операции:

+ (сложение, вещественное)

- (вычитание, вещественное)

+ (унарный плюс, вещественное)

- (унарный минус, вещественное)

\* (умножение, вещественное)

/ (деление, вещественное)

% (остаток, вещественный)

\*\* (степень, вещественная)

= (сравнение, вещественное, результат логический)

> (больше, вещественное, результат логический)

< (меньше, вещественное, результат логический)

>= (больше либо равно, вещественное, результат логический)

<= (меньше либо равно, вещественное, результат логический)

!= (неравенство, вещественное, результат логический)

& (умножение, логическое [коньюнкция])

| (сложение, логическое [дизъюнкция])

^ (исключающее или, логическое)

! (отрицание, логическое)

Должны быть определены константы pi и e с точностью, доступной в среде разработки. Пользователя должна быть возможность добавлять собственные константы. Также должны быть реализованы следующие функции:

min (наименьшее из двух вещественных)

max (наибольшее из двух вещественных)

round (округление к ближайшему целому)

trunc (округление в сторону нуля)

floor (округление в меньшую сторону)

ceil (округление в большую сторону)

sin (синус)

cos (косинус)

tan (тангенс)

cotan (котангенс)

arcsin (арксинус)

arccos (арккосинус)

arctan(арктангенс)

ln (натуральный логарифм)

abs (модуль)

sign (знак, возвращает -1, 0 или 1 соответственно)

Режим работы тригонометрических функций (градусы/радианы) должен выбираться пользователем. Числа могут быть записаны как в десятичной, так и в научной форме.

**Тестирование**

using System;

using Microsoft.VisualStudio.TestTools.UnitTesting;

using static System.Math;

namespace Shaykhullin.Tests

{

[TestClass]

public class BinaryTests : InterpreterTester

{

[TestMethod]

public void ПоследовательностьОперацийВыполняется()

{

var result = Interpret<bool>("true & false | true ^ false");

Assert.IsTrue(result);

}

[TestMethod]

public void ЛогическиеОператорыИиИлиПрименяютсяВПравильнойПоследовательности()

{

var result1 = Interpret<bool>("true & false | true");

var result2 = Interpret<bool>("false | true & false");

Assert.IsTrue(result1);

Assert.IsFalse(result2);

}

[TestMethod]

public void ИсключающееИлиПрименяетсяСОтрицанием()

{

var result = Interpret<bool>("!(true ^ true)");

Assert.IsTrue(result);

}

[TestMethod]

public void ИсключающееИлиПрименяется()

{

var result = Interpret<bool>("true ^ true");

Assert.IsFalse(result);

}

[TestMethod]

public void ЛожьИлиИстинаРавнаИстине()

{

var result = Interpret<bool>("false | true");

Assert.IsTrue(result);

}

[TestMethod]

public void ЛожьИИстинаРавнаЛжи()

{

var result = Interpret<bool>("false & true");

Assert.IsFalse(result);

}

[TestMethod]

public void ЛожьРавнаЛжиСПодстановкой()

{

var result = Interpret<bool>("(1 == 0) == false");

Assert.IsTrue(result);

}

[TestMethod]

public void ЛожьРавнаЛжи()

{

var result = Interpret<bool>("false == false");

Assert.IsTrue(result);

}

[TestMethod]

public void ПравдаРавнаПравде()

{

var result = Interpret<bool>("true == true");

Assert.IsTrue(result);

}

[TestMethod]

public void ПравдаНеРавнаЛжи()

{

var result = Interpret<bool>("true != false");

Assert.IsTrue(result);

}

[TestMethod]

public void ОператорБольшеИлиРавноПрименяетсяКРавным()

{

var result = Interpret<bool>("2.23 >= 2.23");

Assert.IsTrue(result);

}

[TestMethod]

public void ОператорБольшеИлиРавноПрименяется()

{

var result = Interpret<bool>("1.11 >= 2.22");

Assert.IsFalse(result);

}

[TestMethod]

public void ОператорМеньшеИлиРавноПрименяетсяКРавным()

{

var result = Interpret<bool>("2.22 <= 2.22");

Assert.IsTrue(result);

}

[TestMethod]

public void ОператорМеньшеИлиРавноПрименяется()

{

var result = Interpret<bool>("1.11 <= 2.22");

Assert.IsTrue(result);

}

[TestMethod]

public void ОператорМеньшеПрименяетсяКВыражению()

{

var result = Interpret<bool>("1 + 2 < 2 - 100.12");

Assert.IsFalse(result);

}

[TestMethod]

public void ОператорБольшеПрименяетсяКВыраженям()

{

var result = Interpret<bool>("1 + 2 > 2 - 1");

Assert.IsTrue(result);

}

[TestMethod]

public void ОператорБольшеПрименяетсяКВыражению()

{

var result = Interpret<bool>("1 + 2 > 2");

Assert.IsTrue(result);

}

[TestMethod]

public void ОператорБольшеПрименяется()

{

var result = Interpret<bool>("1 > 2");

Assert.IsFalse(result);

}

[TestMethod]

public void ЛогическоеРавенствоПрименяетсяКЛогическому()

{

var result1 = Interpret<bool>("true == true");

Assert.IsTrue(result1);

}

[TestMethod]

public void ЛогическоеРавенствоПрименяется()

{

var result1 = Interpret<bool>("true == (1.2 == 1.2)");

Assert.IsTrue(result1);

}

[TestMethod]

public void ВещественноеРавенствоПрименяетсяСУмножением()

{

var result1 = Interpret<bool>("100 == 10\*10");

Assert.IsTrue(result1);

}

[TestMethod]

public void ВещественноеРавенствоПрименяется()

{

var result1 = Interpret<bool>("100 == 100");

Assert.IsTrue(result1);

}

[TestMethod]

public void БинарноеСложениеПрименяется()

{

var result = Interpret<double>("2+2");

Assert.AreEqual(4, result);

}

[TestMethod]

public void БинарноеВычитаниеПрименяется()

{

var result = Interpret<double>("2-2+2");

Assert.AreEqual(2, result);

}

[TestMethod]

public void БинарноеУмножениеПрименяется()

{

var result1 = Interpret<double>("2\*2+2");

var result2 = Interpret<double>("2+2\*2");

Assert.AreEqual(6, result1);

Assert.AreEqual(6, result2);

}

[TestMethod]

public void БинарноеДелениеПрименяется()

{

var result1 = Interpret<double>("2/2+2");

var result2 = Interpret<double>("2+2/2");

Assert.AreEqual(3, result1);

Assert.AreEqual(3, result2);

}

}

}

using System;

using Microsoft.VisualStudio.TestTools.UnitTesting;

using static System.Math;

namespace Shaykhullin.Tests

{

[TestClass]

public class UnaryTests : InterpreterTester

{

[TestMethod]

public void КоинусВычисляетсяВВыражении()

{

var result = Interpret<double>("sin(cos(12.2))");

Assert.AreEqual(Sin(Cos(12.2)), result);

}

[TestMethod]

public void КоинусВычисляется()

{

var result = Interpret<double>("cos(12.2)");

Assert.AreEqual(Cos(12.2), result);

}

[TestMethod]

public void СинусВычисляетсяВВыражении()

{

var result = Interpret<double>("cos(sin(10))");

Assert.AreEqual(Cos(Sin(10)), result);

}

[TestMethod]

public void СинусВычисляется()

{

var result = Interpret<double>("sin(10)");

Assert.AreEqual(Sin(10), result);

}

[TestMethod]

public void ДвойноеВозведениеВСтепеньПрименяетя()

{

var result = Interpret<double>("2\*\*\*\*");

Assert.AreEqual(16, result);

}

[TestMethod]

public void ОстатокОтДеленияПрименяется()

{

var result = Interpret<double>("min((12 % 5), (1))");

Assert.AreEqual(1d, result);

}

[TestMethod]

public void ОкруглениеДоЦелогоПрименяется()

{

var result = Interpret<double>("trunc(1.19123123)");

Assert.AreEqual(1, result);

}

[TestMethod]

public void ОкруглениеКПлюсБесконечностиПрименяется()

{

var result = Interpret<double>("ceil(1.1)");

Assert.AreEqual(2, result);

}

[TestMethod]

public void ОкруглениеКМинусБесконечностиПрименяется()

{

var result = Interpret<double>("floor(1.99)");

Assert.AreEqual(1, result);

}

[TestMethod]

public void ОкруглениеКЧетномуПрименяется()

{

var result = Interpret<double>("round(1.5)");

Assert.AreEqual(2, result);

}

[TestMethod]

public void ВозведениеВСтепеньПрименяется()

{

var result = Interpret<double>("10\*\*");

Assert.AreEqual(100d, result);

}

[TestMethod]

public void УнарноеОтрицаниеПрименяетсяТрижды()

{

var result = Interpret<bool>("!!!true");

Assert.AreEqual(false, result);

}

[TestMethod]

public void УнарноеОтрицаниеПрименяетсяДважды()

{

var result = Interpret<bool>("!!true");

Assert.AreEqual(true, result);

}

[TestMethod]

public void УнарноеОтрицаниеПрименяется()

{

var result = Interpret<bool>("!true");

Assert.AreEqual(false, result);

}

[TestMethod]

public void УнарнаяФункцияПрименяетсяВнутриБинарной()

{

var result = Interpret<double>("max(sign(-10), sign(10))");

Assert.AreEqual(1d, result);

}

[TestMethod]

public void УнарныйМинусПримеряется()

{

var result = Interpret<double>("-1");

Assert.AreEqual(-1d, result);

}

[TestMethod]

public void УнарнаяФункцияПрименяется()

{

var result = Interpret<double>("sign(-100) + 1");

Assert.AreEqual(0d, result);

}

[TestMethod]

public void УнарныйМинусПримеряетсяНесколькоРаз()

{

var result = Interpret<double>("-----1");

Assert.AreEqual(-1d, result);

}

[TestMethod]

public void УнарныйМинусВместеСУнарнымПлюсом()

{

var result = Interpret<double>("+-+1");

Assert.AreEqual(-1d, result);

}

[TestMethod]

public void УнарныйМинусВместеСБинарнымПлюсом()

{

var result = Interpret<double>("-1+1");

Assert.AreEqual(0d, result);

}

[TestMethod]

public void УнарныйМинусПрименяетсяКПравильномуОперанду()

{

var result = Interpret<double>("min((-1), 2)");

Assert.AreEqual(-1d, result);

}

}

}

using Microsoft.VisualStudio.TestTools.UnitTesting;

using static System.Math;

namespace Shaykhullin.Tests

{

[TestClass]

public class ConstantTests : InterpreterTester

{

[TestMethod]

public void ТангенсВычисляетсяВВыражении()

{

var result = Interpret<double>("min(tan(1.12e-10), 10)");

Assert.AreEqual(Tan(1.12e-10), result);

}

[TestMethod]

public void ТангенсВычисляется()

{

var result = Interpret<double>("tan(1.2e-10)");

Assert.AreEqual(Tan(1.2e-10), result);

}

[TestMethod]

public void ЭкспоненциальнаяФормаЗаписиВычисляетсяВВыражении()

{

var result = Interpret<bool>("min(max(1.5e-10, 10.10e2), 1.4e-11) == 1.4e-11");

Assert.IsTrue(result);

}

[TestMethod]

public void НормализованнаяЭкспоненциальнаяФормаЗаписиВещественныхЧиселПоСтандартуIEEE754()

{

var result = Interpret<double>("1.2e-1");

Assert.AreEqual(0.12, result);

}

[TestMethod]

public void ВыражениеСКонстантамиВычисляется()

{

var result = Interpret<bool>("max(min(e, pi), min((e-pi), (pi\*\*))) == e");

Assert.IsTrue(result);

}

[TestMethod]

public void КонстантаEВычисляетсяВВыражении()

{

var result = Interpret<double>("e-1.111");

Assert.AreEqual(E-1.111, result, 0.00000001);

}

[TestMethod]

public void КонстантаEВычисляется()

{

var result = Interpret<double>("e");

Assert.AreEqual(E, result, 0.00000001);

}

[TestMethod]

public void КонстантаПиВычисляетсяВВыражении()

{

var result = Interpret<double>("pi+1.2");

Assert.AreEqual(PI + 1.2, result, 0.00000001);

}

[TestMethod]

public void КонстантаПиВычисляется()

{

var result = Interpret<double>("pi");

Assert.AreEqual(PI, result, 0.00000001);

}

}

}

namespace Shaykhullin.Tests

{

public class InterpreterTester

{

public TResult Interpret<TResult>(string expression)

{

return (TResult) new ExpressionInterpreter(

processor: new ExpressionSyntaxTreeProcessor(

formatter: new ExpressionSyntaxTreeFormatter(

sortProcessor: new ExpressionInfixSortStationProcessor(

lexingInterpreter: new ExpressionLexingInterpreter(expression)))))

.Interpret();

}

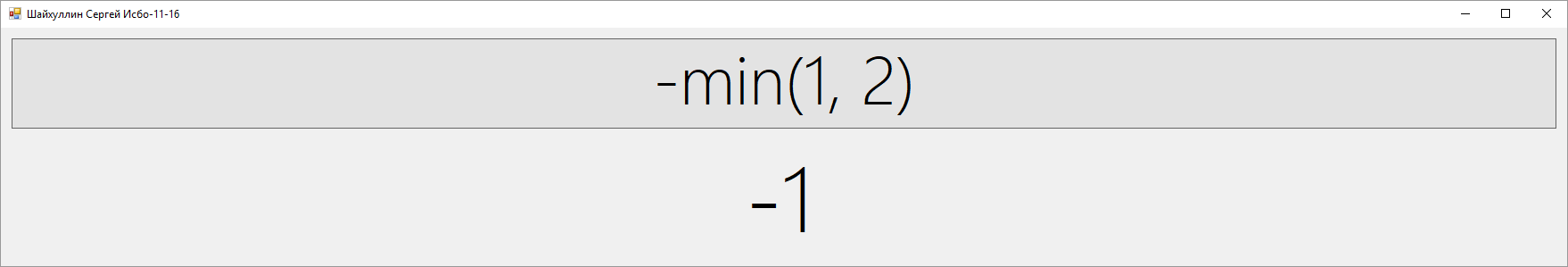
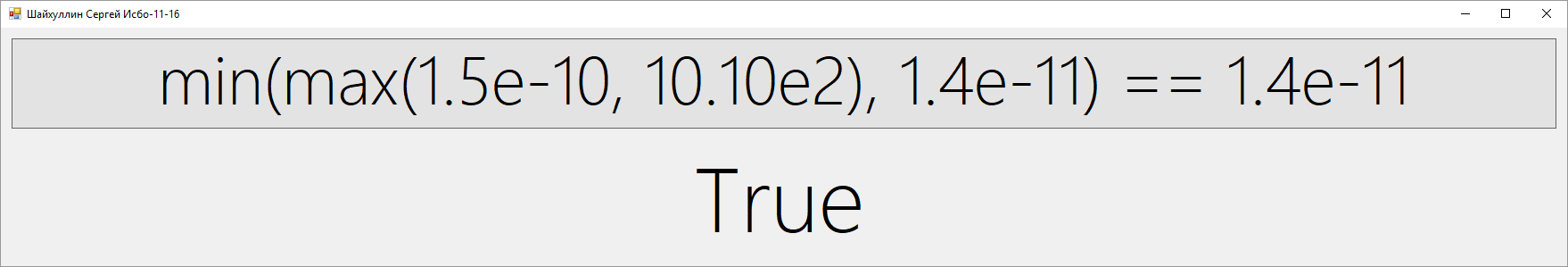
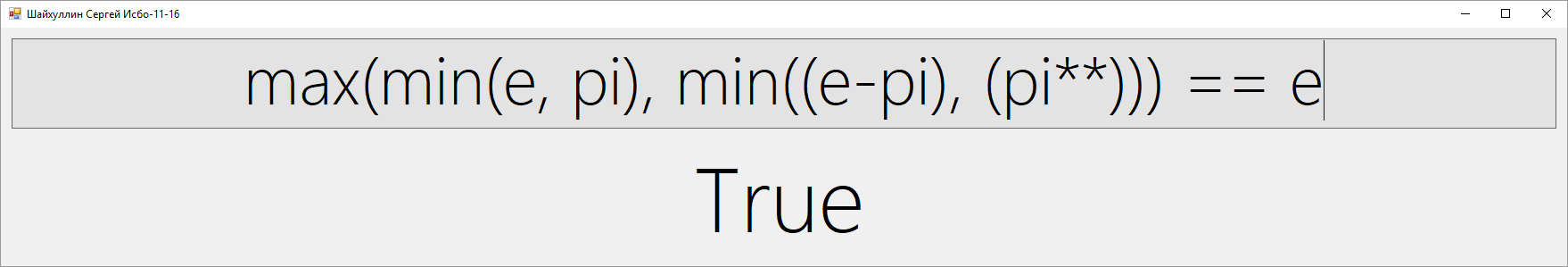
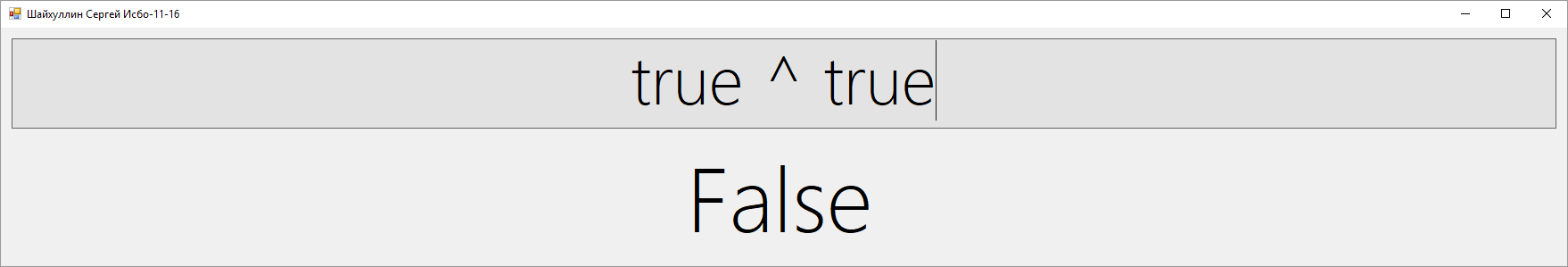
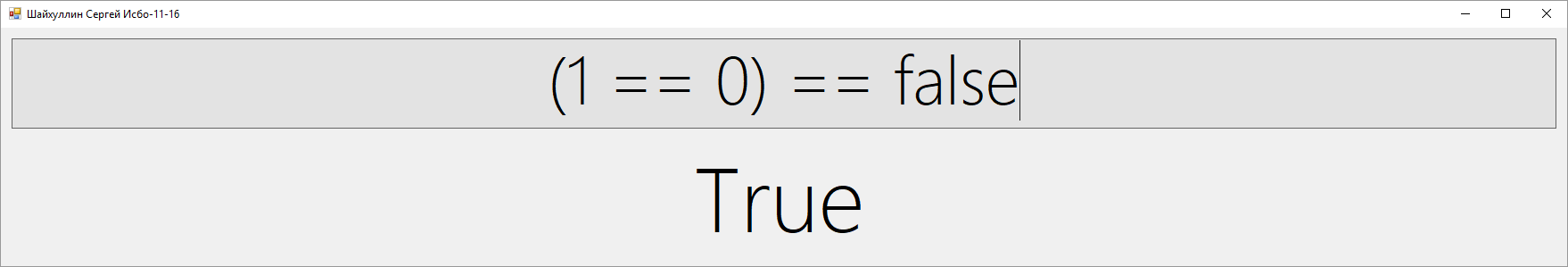
}

}

**Заключение**

В данной лабораторной работе мы научились выстраивать архитектуру интерпретатора, парсить выражения на лексемы, переводить их в постфиксную нотацию, создавать деревья операций и вычислять результат.

**Пример работы**

**Исходный код**

namespace Shaykhullin.Parsers

{

public interface ILexingContext

{

int Start { get; }

int Caret { get; set; }

char Current { get; }

string Expression { get; }

string SubExpression { get; }

}

}

using System;

using System.Collections.Generic;

using System.Linq;

using Shaykhullin.Lexemes;

using Shaykhullin.LexemeSorters;

namespace Shaykhullin

{

public class ExpressionInfixSortStationProcessor

{

private Queue<Lexeme> input;

private Queue<Lexeme> output;

private Stack<Lexeme> stack;

private IEnumerable<LexemeSorter> lexemeSorters;

public ExpressionInfixSortStationProcessor(ExpressionLexingInterpreter lexingInterpreter)

{

input = lexingInterpreter.LexExpression();

output = new Queue<Lexeme>();

stack = new Stack<Lexeme>();

lexemeSorters = typeof(LexemeSorter).Assembly.GetTypes()

.Where(type => typeof(LexemeSorter).IsAssignableFrom(type))

.Where(type => !type.IsAbstract)

.OrderByDescending(type => type.CalculateBaseClasses())

.Select(type => (LexemeSorter)Activator.CreateInstance(type))

.ToList();

}

public Queue<Lexeme> ToPostfixNotaition()

{

Lexeme prevLexeme = null;

while (input.Count > 0)

{

var lexeme = input.Dequeue();

// Первый сортер, попавший под условие и более конкретного к более базовому

var sorter = lexemeSorters.FirstOrDefault(s => s.IsSatisfied(lexeme))

?? throw new InvalidOperationException("Sorter not found");

sorter.Sort(lexeme, prevLexeme, input, output, stack);

prevLexeme = lexeme;

}

while (stack.Count > 0)

{

if (stack.Peek() is LeftParenthesisLexeme)

throw new InvalidOperationException("RIGHT Parenthesis");

output.Enqueue(stack.Pop());

}

return output;

}

}

}

using System;

using System.Linq;

using System.Collections.Generic;

using Shaykhullin.Lexemes;

using Shaykhullin.Parsers;

namespace Shaykhullin

{

public class ExpressionLexingInterpreter : ILexingContext

{

private Queue<Lexeme> lexemes;

public IEnumerable<LexemeParser> lexemeParsers;

public ExpressionLexingInterpreter(string expression)

{

lexemes = new Queue<Lexeme>();

lexemeParsers = typeof(LexemeParser).Assembly.GetTypes()

.Where(type => typeof(LexemeParser).IsAssignableFrom(type))

.Where(type => !type.IsAbstract)

.Select(type => (LexemeParser)Activator.CreateInstance(type))

.OrderByDescending(parser => parser.Name?.Length)

.ToList();

Expression = expression;

}

public string Expression { get; private set; }

public int Start { get; private set; }

public int Caret { get; set; }

public char Current => Expression.ElementAtOrDefault(Caret);

public string SubExpression =>

Caret > Expression.Length ? null : Expression.Substring(Start, Caret - Start);

public Queue<Lexeme> LexExpression()

{

do

{

Start = Caret;

var parser = lexemeParsers.FirstOrDefault(p => p.IsSatisfied(this))

?? throw new InvalidOperationException($"Error in {Caret} symbol");

var lexeme = parser.Parse(this);

if (lexeme != null && !(lexeme is EmptyLexeme))

{

lexemes.Enqueue(lexeme);

}

} while (Caret < Expression.Length);

return lexemes;

}

}

}

using System;

using System.Collections.Generic;

using Shaykhullin.Lexemes;

using Shaykhullin.Operations;

using Shaykhullin.OperationMappers;

using System.Linq;

namespace Shaykhullin

{

public class Tree<TData>

{

public TData Operation { get; set; }

public Tree<TData> Left { get; set; }

public Tree<TData> Right { get; set; }

}

public class ExpressionSyntaxTreeFormatter

{

private Queue<Lexeme> input;

private Stack<Tree<Operation>> operations;

private IEnumerable<LexemeOperationMapper> operationMappers;

public ExpressionSyntaxTreeFormatter(ExpressionInfixSortStationProcessor sortProcessor)

{

input = sortProcessor.ToPostfixNotaition();

operations = new Stack<Tree<Operation>>();

operationMappers = typeof(LexemeOperationMapper).Assembly.GetTypes()

.Where(type => typeof(LexemeOperationMapper).IsAssignableFrom(type))

.Where(type => !type.IsAbstract)

.OrderByDescending(type => type.CalculateBaseClasses())

.Select(type => (LexemeOperationMapper)Activator.CreateInstance(type))

.ToList();

}

public Tree<Operation> FormatOperation()

{

while (input.Count > 0)

{

var lexeme = input.Dequeue();

// Первый мапппер, попавший под условие от более конкретного к более базовому

var mapper = operationMappers.FirstOrDefault(s => s.IsSatisfied(lexeme))

?? throw new InvalidOperationException("Mapper not found");

operations.Push(mapper.Map(operations));

}

if (operations.Count == 1)

{

return operations.Pop();

}

else if (operations.Count == 0)

{

throw new InvalidOperationException("Empty expression");

}

else

{

throw new InvalidOperationException("Syntax error");

}

}

}

}

using System;

using System.Linq;

using System.Collections.Generic;

using Shaykhullin.Operations;

namespace Shaykhullin

{

public class ExpressionSyntaxTreeProcessor

{

private readonly Tree<Operation> input;

private readonly IEnumerable<OperationExecutor> operationExecutors;

public ExpressionSyntaxTreeProcessor(ExpressionSyntaxTreeFormatter formatter)

{

input = formatter.FormatOperation();

operationExecutors = typeof(OperationExecutor).Assembly.GetTypes()

.Where(t => typeof(OperationExecutor).IsAssignableFrom(t))

.Where(t => !t.IsAbstract)

.Select(t => (OperationExecutor)Activator.CreateInstance(t))

.ToArray();

}

public object ExecuteOperaion()

{

return ExecuteOperationRecursive(input);

object ExecuteOperationRecursive(Tree<Operation> node)

{

var executor = operationExecutors.SingleOrDefault(ex => ex.IsSatisfied(node.Operation))

?? throw new InvalidOperationException("Invalid operation");

return executor.ExecuteCore(node.Operation, node, ExecuteOperationRecursive);

}

}

}

}

using Shaykhullin.Lexemes;

namespace Shaykhullin.Parsers

{

public class AbsLexemeParser : LexemeParser<AbsLexeme>

{

public override string Name => "abs";

}

}

using Shaykhullin.Lexemes;

namespace Shaykhullin.Parsers

{

public class AndLexemeParser : LexemeParser<AndLexeme>

{

public override string Name => "&";

}

}

using Shaykhullin.Lexemes;

namespace Shaykhullin.Parsers

{

public class ArccosLexemeParser : LexemeParser<ArccosLexeme>

{

public override string Name => "arccos";

}

}

using Shaykhullin.Lexemes;

namespace Shaykhullin.Parsers

{

public class ArcsinLexemeParser : LexemeParser<ArcsinLexeme>

{

public override string Name => "arcsin";

}

}

using Shaykhullin.Lexemes;

namespace Shaykhullin.Parsers

{

public class ArctanLexemeParser : LexemeParser<ArctanLexeme>

{

public override string Name => "arctan";

}

}

using Shaykhullin.Lexemes;

namespace Shaykhullin.Parsers

{

public abstract class BooleanLexemeParser : LexemeParser<BooleanLexeme>

{

public override Lexeme Parse(ILexingContext context)

{

base.Parse(context);

return new BooleanLexeme(Name);

}

}

}

using Shaykhullin.Lexemes;

namespace Shaykhullin.Parsers

{

public class CeilLexemeParser : LexemeParser<CeilLexeme>

{

public override string Name => "ceil";

}

}

using Shaykhullin.Lexemes;

namespace Shaykhullin.Parsers

{

public class CommaLexemeParser : LexemeParser<CommaLexeme>

{

public override string Name => ",";

}

}

using Shaykhullin.Lexemes;

namespace Shaykhullin.Parsers

{

public class CosLexemeParser : LexemeParser<CosLexeme>

{

public override string Name => "cos";

}

}

using Shaykhullin.Lexemes;

namespace Shaykhullin.Parsers

{

public class CotanLexemeParser : LexemeParser<CotanLexeme>

{

public override string Name => "cotan";

}

}

using Shaykhullin.Lexemes;

namespace Shaykhullin.Parsers

{

public class DivideLexemeParser : LexemeParser<DivideLexeme>

{

public override string Name => "/";

}

}

using Shaykhullin.Lexemes;

namespace Shaykhullin.Parsers

{

public class DoubleLexemeParser : LexemeParser

{

public override bool IsSatisfied(ILexingContext context)

{

return char.IsDigit(context.Current) || context.Current == '.';

}

public override Lexeme Parse(ILexingContext context)

{

var hasSeparator = false;

var hasExp = false;

while (char.IsDigit(context.Current)

|| (context.Current == '.' && !hasSeparator)

|| ((context.Current == 'e' || context.Current == 'E') && !hasExp)

|| context.Current == '-' && hasExp)

{

if (context.Current == '.' && !hasSeparator)

hasSeparator = true;

if ((context.Current == 'e' || context.Current == 'E') && !hasExp)

hasExp = true;

context.Caret++;

}

return new DoubleLexeme(context.SubExpression);

}

}

}

using Shaykhullin.Lexemes;

namespace Shaykhullin.Parsers

{

public class EmptyLexemeParser : LexemeParser<EmptyLexeme>

{

public override string Name => " ";

}

}

using Shaykhullin.Lexemes;

namespace Shaykhullin.Parsers

{

public class EqualLexemeParser : LexemeParser<EqualLexeme>

{

public override string Name => "==";

}

}

using Shaykhullin.Lexemes;

namespace Shaykhullin.Parsers

{

public class ExpLexemeParser : LexemeParser<ExpLexeme>

{

public override string Name => "E";

}

}

namespace Shaykhullin.Parsers

{

public class FalseLexemeParser : BooleanLexemeParser

{

public override string Name => "False";

}

}

using Shaykhullin.Lexemes;

namespace Shaykhullin.Parsers

{

public class FloorLexemeParser : LexemeParser<FloorLexeme>

{

public override string Name => "floor";

}

}

using Shaykhullin.Lexemes;

namespace Shaykhullin.Parsers

{

public class GreaterLexemeParser : LexemeParser<GreaterLexeme>

{

public override string Name => ">";

public override bool IsSatisfied(ILexingContext context) =>

base.IsSatisfied(context) && context.Expression[context.Caret + 1] != '=';

}

}

using Shaykhullin.Lexemes;

namespace Shaykhullin.Parsers

{

public class GreaterOrEqualLexemeParser : LexemeParser<GreaterOrEqualLexeme>

{

public override string Name => ">=";

}

}

using Shaykhullin.Lexemes;

namespace Shaykhullin.Parsers

{

public class LeftParenthesisLexemeParser : LexemeParser<LeftParenthesisLexeme>

{

public override string Name => "(";

}

}

using Shaykhullin.Lexemes;

namespace Shaykhullin.Parsers

{

public class LessLexemeParser : LexemeParser<LessLexeme>

{

public override string Name => "<";

public override bool IsSatisfied(ILexingContext context) =>

base.IsSatisfied(context) && context.Expression[context.Caret + 1] != '=';

}

}

using Shaykhullin.Lexemes;

namespace Shaykhullin.Parsers

{

public class LessOrEqualLexemeParser : LexemeParser<LessOrEqualLexeme>

{

public override string Name => "<=";

}

}

using Shaykhullin.Lexemes;

namespace Shaykhullin.Parsers

{

public abstract class LexemeParser

{

public virtual string Name { get; }

public abstract Lexeme Parse(ILexingContext context);

public abstract bool IsSatisfied(ILexingContext context);

}

public abstract class LexemeParser<TLexeme> : LexemeParser

where TLexeme : Lexeme, new()

{

public override Lexeme Parse(ILexingContext context)

{

context.Caret += Name.Length;

return new TLexeme();

}

public override bool IsSatisfied(ILexingContext context)

{

if (context.Start + Name.Length > context.Expression.Length)

{

return false;

}

for (int index = context.Start; index < context.Start + Name.Length; index++)

{

if (char.ToLower(context.Expression[index]) != char.ToLower(Name[index - context.Start]))

{

return false;

}

}

return true;

}

}

}

using Shaykhullin.Lexemes;

namespace Shaykhullin.Parsers

{

public class LnLexemeParser : LexemeParser<ArctanLexeme>

{

public override string Name => "ln";

}

}

using Shaykhullin.Lexemes;

namespace Shaykhullin.Parsers

{

public class MaxLexemeParser : LexemeParser<MaxLexeme>

{

public override string Name => "max";

}

}

using Shaykhullin.Lexemes;

namespace Shaykhullin.Parsers

{

public class MinLexemeParser : LexemeParser<MinLexeme>

{

public override string Name => "min";

}

}

using Shaykhullin.Lexemes;

namespace Shaykhullin.Parsers

{

public class MinusLexemeParser : LexemeParser<MinusLexeme>

{

public override string Name => "-";

}

}

using Shaykhullin.Lexemes;

namespace Shaykhullin.Parsers

{

public class ModulusLexemeParser : LexemeParser<ModulusLexeme>

{

public override string Name => "%";

}

}

using Shaykhullin.Lexemes;

namespace Shaykhullin.Parsers

{

public class MultiplyLexemeParser : LexemeParser<MultiplyLexeme>

{

public override string Name => "\*";

public override bool IsSatisfied(ILexingContext context) =>

base.IsSatisfied(context) && context.Expression[context.Caret + 1] != '\*';

}

}

using Shaykhullin.Lexemes;

namespace Shaykhullin.Parsers

{

public class NotEqualLexemeParser : LexemeParser<NotEqualLexeme>

{

public override string Name => "!=";

}

}

using Shaykhullin.Lexemes;

namespace Shaykhullin.Parsers

{

public class NotLexemeParser : LexemeParser<NotLexeme>

{

public override string Name => "!";

public override bool IsSatisfied(ILexingContext context) =>

base.IsSatisfied(context) && context.Expression[context.Caret + 1] != '=';

}

}

using Shaykhullin.Lexemes;

namespace Shaykhullin.Parsers

{

public class OrLexemeParser : LexemeParser<OrLexeme>

{

public override string Name => "|";

}

}

using Shaykhullin.Lexemes;

namespace Shaykhullin.Parsers

{

public class PiLexemeParser : LexemeParser<PiLexeme>

{

public override string Name => "Pi";

}

}

using Shaykhullin.Lexemes;

namespace Shaykhullin.Parsers

{

public class PlusLexemeParser : LexemeParser<PlusLexeme>

{

public override string Name => "+";

}

}

using Shaykhullin.Lexemes;

namespace Shaykhullin.Parsers

{

public class PowerLexemeParser : LexemeParser<PowerLexeme>

{

public override string Name => "\*\*";

}

}

using Shaykhullin.Lexemes;

namespace Shaykhullin.Parsers

{

public class RightParenthesisLexemeParser : LexemeParser<RightParenthesisLexeme>

{

public override string Name => ")";

}

}

using Shaykhullin.Lexemes;

namespace Shaykhullin.Parsers

{

public class RoundLexemeParser : LexemeParser<RoundLexeme>

{

public override string Name => "round";

}

}

using Shaykhullin.Lexemes;

namespace Shaykhullin.Parsers

{

public class SignLexemeParser : LexemeParser<SignLexeme>

{

public override string Name => "Sign";

}

}

using Shaykhullin.Lexemes;

namespace Shaykhullin.Parsers

{

public class SinLexemeParser : LexemeParser<SinLexeme>

{

public override string Name => "sin";

}

}

using Shaykhullin.Lexemes;

namespace Shaykhullin.Parsers

{

public class TanLexemeParser : LexemeParser<TanLexeme>

{

public override string Name => "tan";

}

}

namespace Shaykhullin.Parsers

{

public class TrueLexemeParser : BooleanLexemeParser

{

public override string Name => "True";

}

}

using Shaykhullin.Lexemes;

namespace Shaykhullin.Parsers

{

public class TruncLexemeParser : LexemeParser<TruncLexeme>

{

public override string Name => "trunc";

}

}

using Shaykhullin.Lexemes;

namespace Shaykhullin.Parsers

{

public class XorLexemeParser : LexemeParser<XorLexeme>

{

public override string Name => "^";

}

}

namespace Shaykhullin.Lexemes

{

public class BooleanLexeme : ValueLexeme<bool>

{

public BooleanLexeme() : base(false)

{

}

public BooleanLexeme(string value) : base(bool.Parse(value))

{

}

}

// DONE

}

namespace Shaykhullin.Lexemes

{

public class ComparisonOperationLexeme : OperationLexeme

{

public override int Order => 6;

}

public class EqualLexeme : ComparisonOperationLexeme

{

}

public class GreaterLexeme : ComparisonOperationLexeme { }

public class LessLexeme : ComparisonOperationLexeme { }

public class GreaterOrEqualLexeme : ComparisonOperationLexeme { }

public class LessOrEqualLexeme : ComparisonOperationLexeme { }

public class NotEqualLexeme : ComparisonOperationLexeme { }

public class NotLexeme : ComparisonOperationLexeme { }

}

using System;

namespace Shaykhullin.Lexemes

{

public class DoubleLexeme : ValueLexeme<double>

{

public DoubleLexeme(string value) : base(double.Parse(value))

{

}

}

public class PiLexeme : DoubleLexeme

{

public PiLexeme() : base(Math.PI.ToString()) { }

}

public class ExpLexeme : DoubleLexeme

{

public ExpLexeme() : base(Math.E.ToString()) { }

}

}

namespace Shaykhullin.Lexemes

{

// function

public class FunctionLexeme : Lexeme

{

public override int Order => 4;

}

public class MinLexeme : FunctionLexeme { }

public class MaxLexeme : FunctionLexeme { }

public class RoundLexeme : FunctionLexeme { }

public class TruncLexeme : FunctionLexeme { }

public class FloorLexeme : FunctionLexeme { }

public class CeilLexeme : FunctionLexeme { }

public class SinLexeme : FunctionLexeme { }

public class CosLexeme : FunctionLexeme { }

public class TanLexeme : FunctionLexeme { }

public class CotanLexeme : FunctionLexeme { }

public class ArcsinLexeme : FunctionLexeme { }

public class ArccosLexeme : FunctionLexeme { }

public class ArctanLexeme : FunctionLexeme { }

public class LnLexeme : FunctionLexeme { }

public class AbsLexeme : FunctionLexeme { }

public class SignLexeme : FunctionLexeme { }

}

using System;

namespace Shaykhullin.Lexemes

{

public abstract class Lexeme

{

public abstract int Order { get; }

}

}

namespace Shaykhullin.Lexemes

{

//operation

public class OperationLexeme : Lexeme

{

public override int Order => 4;

}

public class PlusLexeme : OperationLexeme

{

public override int Order => 5;

}

public class MinusLexeme : OperationLexeme

{

public override int Order => 3;

}

public class UnaryPlusLexeme : OperationLexeme

{

public override int Order => 3;

}

public class UnaryMinusLexeme : OperationLexeme

{

public override int Order => 3;

}

public class MultiplyLexeme : OperationLexeme

{

public override int Order => 2;

}

public class DivideLexeme : OperationLexeme

{

public override int Order => 2;

}

public class ModulusLexeme : OperationLexeme

{

public override int Order => 2;

}

public class PowerLexeme : OperationLexeme

{

public override int Order => 3;

}

public class AndLexeme : OperationLexeme { }

public class OrLexeme : OperationLexeme { }

public class XorLexeme : OperationLexeme { }

}

namespace Shaykhullin.Lexemes

{

// separator

public class SeparatorLexeme : Lexeme

{

public override int Order => 3;

}

public class EmptyLexeme : SeparatorLexeme { }

public class CommaLexeme : SeparatorLexeme { }

public class LeftParenthesisLexeme : SeparatorLexeme { }

public class RightParenthesisLexeme : SeparatorLexeme { }

}

namespace Shaykhullin.Lexemes

{

// value

public abstract class ValueLexeme : Lexeme

{

public object Value { get; }

public override int Order => 2;

protected ValueLexeme(object value)

{

Value = value;

}

}

public abstract class ValueLexeme<TValue> : ValueLexeme

{

public new TValue Value => (TValue)base.Value;

protected ValueLexeme(TValue value) : base(value)

{

}

}

}

using System.Collections.Generic;

using Shaykhullin.Lexemes;

namespace Shaykhullin.LexemeSorters

{

public class BooleanLexemeSorter : LexemeSorter<BooleanLexeme>

{

public override void Sort(Lexeme lexeme, Lexeme prevLexeme, Queue<Lexeme> input, Queue<Lexeme> output, Stack<Lexeme> stack)

{

output.Enqueue(lexeme);

}

}

}

using System;

using System.Collections.Generic;

using Shaykhullin.Lexemes;

namespace Shaykhullin.LexemeSorters

{

public class CommaLexemeSorter : LexemeSorter<CommaLexeme>

{

public override void Sort(Lexeme Lexeme, Lexeme prevLexeme, Queue<Lexeme> input, Queue<Lexeme> output, Stack<Lexeme> stack)

{

if (!(prevLexeme is ValueLexeme || prevLexeme is RightParenthesisLexeme))

{

throw new InvalidOperationException("Invalid Lexeme");

}

}

}

}

using System.Collections.Generic;

using Shaykhullin.Lexemes;

namespace Shaykhullin.LexemeSorters

{

public class DoubleLexemeSorter : LexemeSorter<DoubleLexeme>

{

public override void Sort(Lexeme Lexeme, Lexeme prevLexeme, Queue<Lexeme> input, Queue<Lexeme> output, Stack<Lexeme> stack)

{

output.Enqueue(Lexeme);

}

}

}

using System;

using System.Collections.Generic;

using Shaykhullin.Lexemes;

namespace Shaykhullin.LexemeSorters

{

public class FunctionLexemeSorter : LexemeSorter<FunctionLexeme>

{

public override void Sort(Lexeme Lexeme, Lexeme prevLexeme, Queue<Lexeme> input, Queue<Lexeme> output, Stack<Lexeme> stack)

{

if (input.Count > 0 && input.Peek() is LeftParenthesisLexeme)

{

stack.Push(Lexeme);

}

else

{

throw new InvalidOperationException("Invalid Lexeme");

}

}

}

}

using System.Collections.Generic;

using Shaykhullin.Lexemes;

namespace Shaykhullin.LexemeSorters

{

public class LeftParenthesisLexemeSorter : LexemeSorter<LeftParenthesisLexeme>

{

public override void Sort(Lexeme Lexeme, Lexeme prevLexeme, Queue<Lexeme> input, Queue<Lexeme> output, Stack<Lexeme> stack)

{

stack.Push(Lexeme);

}

}

}

using System.Collections.Generic;

using Shaykhullin.Lexemes;

namespace Shaykhullin.LexemeSorters

{

public abstract class LexemeSorter

{

public abstract bool IsSatisfied(Lexeme Lexeme);

public abstract void Sort(Lexeme Lexeme, Lexeme prevLexeme, Queue<Lexeme> input, Queue<Lexeme> output, Stack<Lexeme> stack);

}

public abstract class LexemeSorter<TLexeme>

: LexemeSorter where TLexeme : Lexeme

{

public override bool IsSatisfied(Lexeme Lexeme)

{

return Lexeme is TLexeme;

}

}

}

using System.Collections.Generic;

using Shaykhullin.Lexemes;

namespace Shaykhullin.LexemeSorters

{

public class MinusOperationLexemeSorter : LexemeSorter<MinusLexeme>

{

public override void Sort(Lexeme lexeme, Lexeme prevLexeme, Queue<Lexeme> input, Queue<Lexeme> output, Stack<Lexeme> stack)

{

if (!(prevLexeme is ValueLexeme || prevLexeme is RightParenthesisLexeme))

{

stack.Push(new UnaryMinusLexeme());

}

else

{

while (stack.Count > 0 && stack.Peek() is OperationLexeme)

{

if (stack.Peek().Order >= lexeme.Order)

break;

output.Enqueue(stack.Pop());

}

stack.Push(lexeme);

}

}

}

}

using System.Collections.Generic;

using Shaykhullin.Lexemes;

namespace Shaykhullin.LexemeSorters

{

public class OperationLexemeSorter : LexemeSorter<OperationLexeme>

{

public override bool IsSatisfied(Lexeme lexeme)

{

return base.IsSatisfied(lexeme) && !(lexeme is PlusLexeme || lexeme is MinusLexeme);

}

public override void Sort(Lexeme Lexeme, Lexeme prevLexeme, Queue<Lexeme> input, Queue<Lexeme> output, Stack<Lexeme> stack)

{

while (stack.Count > 0 && stack.Peek() is OperationLexeme)

{

if (stack.Peek().Order >= Lexeme.Order)

break;

output.Enqueue(stack.Pop());

}

stack.Push(Lexeme);

}

}

}

using System.Collections.Generic;

using Shaykhullin.Lexemes;

namespace Shaykhullin.LexemeSorters

{

public class PlusOperationLexemeSorter : LexemeSorter<PlusLexeme>

{

public override void Sort(Lexeme lexeme, Lexeme prevLexeme, Queue<Lexeme> input, Queue<Lexeme> output, Stack<Lexeme> stack)

{

if (!(prevLexeme is ValueLexeme || prevLexeme is RightParenthesisLexeme))

{

stack.Push(new UnaryPlusLexeme());

}

else

{

while (stack.Count > 0 && stack.Peek() is OperationLexeme)

{

if (stack.Peek().Order >= lexeme.Order)

break;

output.Enqueue(stack.Pop());

}

stack.Push(lexeme);

}

}

}

}

using System;

using System.Collections.Generic;

using Shaykhullin.Lexemes;

namespace Shaykhullin.LexemeSorters

{

public class RightParenthesisLexemeSorter : LexemeSorter<RightParenthesisLexeme>

{

public override void Sort(Lexeme Lexeme, Lexeme prevLexeme, Queue<Lexeme> input, Queue<Lexeme> output, Stack<Lexeme> stack)

{

while (stack.Count > 0 && !(stack.Peek() is LeftParenthesisLexeme))

output.Enqueue(stack.Pop());

if (stack.Count > 0)

stack.Pop();

else

throw new InvalidOperationException("Closing Parenthesis");

if (stack.Count > 0 && stack.Peek() is FunctionLexeme)

output.Enqueue(stack.Pop());

}

}

}

//------------------------------------------------------------------------------

// <auto-generated>

// This code was generated by a tool.

// Runtime Version:4.0.30319.42000

//

// Changes to this file may cause incorrect behavior and will be lost if

// the code is regenerated.

// </auto-generated>

//------------------------------------------------------------------------------

using System;

using System.Reflection;

[assembly: System.Reflection.AssemblyCompanyAttribute("Shaykhullin.Lab4")]

[assembly: System.Reflection.AssemblyConfigurationAttribute("Debug")]

[assembly: System.Reflection.AssemblyFileVersionAttribute("1.0.0.0")]

[assembly: System.Reflection.AssemblyInformationalVersionAttribute("1.0.0")]

[assembly: System.Reflection.AssemblyProductAttribute("Shaykhullin.Lab4")]

[assembly: System.Reflection.AssemblyTitleAttribute("Shaykhullin.Lab4")]

[assembly: System.Reflection.AssemblyVersionAttribute("1.0.0.0")]

// Generated by the MSBuild WriteCodeFragment class.

using Shaykhullin.Operations;

using System;

namespace Shaykhullin

{

public class BinaryOperationExecutor : OperationExecutor<BinaryOperation>

{

public override object Execute(

BinaryOperation operation,

Tree<Operation> node,

Func<Tree<Operation>, object> executeRecursive)

{

return operation.ExecuteCore(executeRecursive(node.Left), executeRecursive(node.Right));

}

}

}

using Shaykhullin.Operations;

using System;

namespace Shaykhullin

{

public abstract class OperationExecutor

{

public abstract bool IsSatisfied(Operation operation);

public abstract object ExecuteCore(

Operation operation,

Tree<Operation> node,

Func<Tree<Operation>, object> executeRecursive);

}

public abstract class OperationExecutor<TOperation> : OperationExecutor

where TOperation : Operation

{

public override bool IsSatisfied(Operation operation)

{

return operation is TOperation;

}

public sealed override object ExecuteCore(

Operation operation,

Tree<Operation> node,

Func<Tree<Operation>, object> executeRecursive)

{

return Execute((TOperation)operation, node, executeRecursive);

}

public abstract object Execute(

TOperation operation,

Tree<Operation> node,

Func<Tree<Operation>, object> executeRecursive);

}

}

using Shaykhullin.Operations;

using System;

namespace Shaykhullin

{

public class UnaryOperationExecutor : OperationExecutor<UnaryOperation>

{

public override object Execute(

UnaryOperation operation,

Tree<Operation> node,

Func<Tree<Operation>, object> executeRecursive)

{

return operation.ExecuteCore(executeRecursive(node.Left ?? node.Right));

}

}

}

using Shaykhullin.Operations;

using System;

namespace Shaykhullin

{

public class ValueOperationExecutor

: OperationExecutor<ValueOperation>

{

public override object Execute(

ValueOperation operation,

Tree<Operation> node,

Func<Tree<Operation>, object> executeRecursive)

{

return operation.ExecuteCore();

}

}

}

using Shaykhullin.Lexemes;

using Shaykhullin.Operations;

namespace Shaykhullin.OperationMappers

{

public class BinaryAndLexemeOperationMapper

: BinaryLexemeOperationMapper<AndLexeme, BinaryAndOperation>

{

}

}

using Shaykhullin.Lexemes;

using Shaykhullin.Operations;

namespace Shaykhullin.OperationMappers

{

public class BinaryDivideLexemeOperationMapper

: BinaryLexemeOperationMapper<DivideLexeme, BinaryDivideOperation>

{

}

}

using Shaykhullin.Lexemes;

using Shaykhullin.Operations;

namespace Shaykhullin.OperationMappers

{

public class BinaryEqualLexemeOperationMapper

: BinaryLexemeOperationMapper<EqualLexeme, BinaryEqualOperation>

{

}

}

using Shaykhullin.Lexemes;

using Shaykhullin.Operations;

namespace Shaykhullin.OperationMappers

{

public class BinaryGreaterOrEqualLexemeOperationMapper

: BinaryLexemeOperationMapper<GreaterOrEqualLexeme, BinaryGreaterOrEqualOperation>

{

}

}

using Shaykhullin.Lexemes;

using Shaykhullin.Operations;

namespace Shaykhullin.OperationMappers

{

public class BinaryGreaterLexemeOperationMapper

: BinaryLexemeOperationMapper<GreaterLexeme, BinaryGreaterOperation>

{

}

}

using Shaykhullin.Lexemes;

using Shaykhullin.Operations;

namespace Shaykhullin.OperationMappers

{

public class BinaryLessOrEqualLexemeOperationMapper

: BinaryLexemeOperationMapper<LessOrEqualLexeme, BinaryLessOrEqualOperation>

{

}

}

using Shaykhullin.Lexemes;

using Shaykhullin.Operations;

namespace Shaykhullin.OperationMappers

{

public class BinaryLessLexemeOperationMapper

: BinaryLexemeOperationMapper<LessLexeme, BinaryLessOperation>

{

}

}

using System;

using System.Collections.Generic;

using Shaykhullin.Lexemes;

using Shaykhullin.Operations;

namespace Shaykhullin.OperationMappers

{

public abstract class BinaryLexemeOperationMapper<TLexeme, TOperation>

: LexemeOperationMapper<TLexeme, TOperation>

where TLexeme : Lexeme

where TOperation : BinaryOperation, new()

{

public override Tree<Operation> Map(Stack<Tree<Operation>> operations)

{

if (operations.Count < 2)

{

throw new InvalidOperationException("Invalid binary operation");

}

return new Tree<Operation>

{

Operation = new TOperation(),

Right = operations.Pop(),

Left = operations.Pop()

};

}

}

}

using Shaykhullin.Lexemes;

using Shaykhullin.Operations;

namespace Shaykhullin.OperationMappers

{

public class BinaryMinLexemeOperationMapper

: BinaryLexemeOperationMapper<MinLexeme, BinaryMinOperation>

{

}

public class BinaryMaxLexemeOperationMapper

: BinaryLexemeOperationMapper<MaxLexeme, BinaryMaxOperation>

{

}

public class RoundLexemeOperationMapper

: UnaryLexemeOperationMapper<RoundLexeme, UnaryRoundOperation>

{

}

public class TruncLexemeOperationMapper

: UnaryLexemeOperationMapper<TruncLexeme, UnaryTruncOperation>

{

}

public class FloorLexemeOperationMapper

: UnaryLexemeOperationMapper<FloorLexeme, UnaryFloorOperation>

{

}

public class CeilLexemeOperationMapper

: UnaryLexemeOperationMapper<CeilLexeme, UnaryCeilOperation>

{

}

public class SinLexemeOperationMapper

: UnaryLexemeOperationMapper<SinLexeme, UnarySinOperation>

{

}

public class CosLexemeOperationMapper

: UnaryLexemeOperationMapper<CosLexeme, UnaryCosOperation>

{

}

public class TanLexemeOperationMapper

: UnaryLexemeOperationMapper<TanLexeme, UnaryTanOperation>

{

}

public class CotanLexemeOperationMapper

: UnaryLexemeOperationMapper<CotanLexeme, UnaryCotanOperation>

{

}

public class ArcsinLexemeOperationMapper

: UnaryLexemeOperationMapper<ArcsinLexeme, UnaryArcsinOperation>

{

}

public class ArccosLexemeOperationMapper

: UnaryLexemeOperationMapper<ArccosLexeme, UnaryArccosOperation>

{

}

public class ArctanLexemeOperationMapper

: UnaryLexemeOperationMapper<ArctanLexeme, UnaryArctanOperation>

{

}

public class LnLexemeOperationMapper

: UnaryLexemeOperationMapper<LnLexeme, UnaryLnOperation>

{

}

public class AbsLexemeOperationMapper

: UnaryLexemeOperationMapper<AbsLexeme, UnaryAbsOperation>

{

}

public class SignLexemeOperationMapper

: UnaryLexemeOperationMapper<SignLexeme, UnarySignOperation>

{

}

}

using Shaykhullin.Lexemes;

using Shaykhullin.Operations;

namespace Shaykhullin.OperationMappers

{

public class BinaryMinusLexemeOperationMapper

: BinaryLexemeOperationMapper<MinusLexeme, BinaryMinusOperation>

{

}

}

using Shaykhullin.Operations;

using Shaykhullin.Lexemes;

namespace Shaykhullin.OperationMappers

{

public class BinaryModulusOperationMapper

: BinaryLexemeOperationMapper<ModulusLexeme, BinaryModulusOperation>

{

}

}

using Shaykhullin.Lexemes;

using Shaykhullin.Operations;

namespace Shaykhullin.OperationMappers

{

public class BinaryMultiplyLexemeOperationMapper

: BinaryLexemeOperationMapper<MultiplyLexeme, BinaryMultiplyOperation>

{

}

}

using Shaykhullin.Lexemes;

using Shaykhullin.Operations;

namespace Shaykhullin.OperationMappers

{

public class BinaryNotEqualLexemeOperationMapper

: BinaryLexemeOperationMapper<NotEqualLexeme, BinaryNotEqualOperation>

{

}

}

using Shaykhullin.Lexemes;

using Shaykhullin.Operations;

namespace Shaykhullin.OperationMappers

{

public class BinaryOrLexemeOperationMapper

: BinaryLexemeOperationMapper<OrLexeme, BinaryOrOperation>

{

}

}

using Shaykhullin.Lexemes;

using Shaykhullin.Operations;

namespace Shaykhullin.OperationMappers

{

public class BinaryPlusLexemeOperationMapper

: BinaryLexemeOperationMapper<PlusLexeme, BinaryPlusOperation>

{

}

}

using Shaykhullin.Lexemes;

using Shaykhullin.Operations;

namespace Shaykhullin.OperationMappers

{

public class BinaryXorLexemeOperationMapper

: BinaryLexemeOperationMapper<XorLexeme, BinaryXorOperation>

{

}

}

using System;

using System.Collections.Generic;

using Shaykhullin.Lexemes;

using Shaykhullin.Operations;

namespace Shaykhullin.OperationMappers

{

public abstract class UnaryLexemeOperationMapper<TLexeme, TOperation>

: LexemeOperationMapper<TLexeme, TOperation>

where TLexeme : Lexeme

where TOperation : UnaryOperation, new()

{

public override Tree<Operation> Map(Stack<Tree<Operation>> operations)

{

if (operations.Count < 1)

{

throw new InvalidOperationException("Invalid unary operation");

}

return new Tree<Operation>

{

Operation = new TOperation(),

Right = operations.Pop()

};

}

}

}

using Shaykhullin.Lexemes;

using Shaykhullin.Operations;

namespace Shaykhullin.OperationMappers

{

public class UnaryMinusLexemeOperationMapper

: UnaryLexemeOperationMapper<UnaryMinusLexeme, UnaryMinusOperation>

{

}

}

using Shaykhullin.Lexemes;

using Shaykhullin.Operations;

namespace Shaykhullin.OperationMappers

{

public class UnaryNotLexemeOperationMapper

: UnaryLexemeOperationMapper<NotLexeme, UnaryNotOperation>

{

}

}

using Shaykhullin.Lexemes;

using Shaykhullin.Operations;

namespace Shaykhullin.OperationMappers

{

public class UnaryPlusLexemeOperationMapper

: UnaryLexemeOperationMapper<UnaryPlusLexeme, UnaryPlusOperation>

{

}

}

using Shaykhullin.Lexemes;

using Shaykhullin.Operations;

namespace Shaykhullin.OperationMappers

{

public class UnaryPowerLexemeOperationMapper

: UnaryLexemeOperationMapper<PowerLexeme, UnaryPowerOperation>

{

}

}

using System.Collections.Generic;

using Shaykhullin.Lexemes;

using Shaykhullin.Operations;

namespace Shaykhullin.OperationMappers

{

public class BooleanLexemeOperationMapper

: LexemeOperationMapper<BooleanLexeme, DoubleValue>

{

private Lexeme Lexeme;

public override bool IsSatisfied(Lexeme Lexeme)

{

return base.IsSatisfied((this.Lexeme = Lexeme));

}

public override Tree<Operation> Map(Stack<Tree<Operation>> operations)

{

return new Tree<Operation>

{

Operation = new BooleanValue(((BooleanLexeme)Lexeme).Value)

};

}

}

}

using System.Collections.Generic;

using Shaykhullin.Lexemes;

using Shaykhullin.Operations;

namespace Shaykhullin.OperationMappers

{

public class DoubleLexemeOperationMapper

: LexemeOperationMapper<DoubleLexeme, DoubleValue>

{

private Lexeme Lexeme;

public override bool IsSatisfied(Lexeme Lexeme)

{

return base.IsSatisfied((this.Lexeme = Lexeme));

}

public override Tree<Operation> Map(Stack<Tree<Operation>> operations)

{

return new Tree<Operation>

{

Operation = new DoubleValue(((DoubleLexeme)Lexeme).Value)

};

}

}

}

using System.Collections.Generic;

using Shaykhullin.Lexemes;

using Shaykhullin.Operations;

namespace Shaykhullin.OperationMappers

{

public abstract class ValueLexemeOperationMapper<TLexeme, TOperation>

: LexemeOperationMapper<TLexeme, TOperation>

where TLexeme : Lexeme

where TOperation : ValueOperation, new()

{

public override Tree<Operation> Map(Stack<Tree<Operation>> operations)

{

return new Tree<Operation>

{

Operation = new TOperation()

};

}

}

}

using System.Collections.Generic;

using Shaykhullin.Lexemes;

using Shaykhullin.Operations;

namespace Shaykhullin.OperationMappers

{

public abstract class LexemeOperationMapper

{

public abstract bool IsSatisfied(Lexeme Lexeme);

public abstract Tree<Operation> Map(Stack<Tree<Operation>> operations);

}

public abstract class LexemeOperationMapper<TLexeme, TOperation> : LexemeOperationMapper

where TLexeme : Lexeme

where TOperation : Operation

{

public override bool IsSatisfied(Lexeme Lexeme)

{

return Lexeme is TLexeme;

}

}

}

namespace Shaykhullin.Operations

{

public class BinaryAndOperation

: BinaryOperation<bool, bool>

{

public override bool Calculate(bool arg1, bool arg2)

{

return arg1 & arg2;

}

}

}

namespace Shaykhullin.Operations

{

public class BinaryDivideOperation

: BinaryOperation<double, double>

{

public override double Calculate(double arg1, double arg2)

{

return arg1 / arg2;

}

}

}

namespace Shaykhullin.Operations

{

public class BinaryEqualOperation

: BinaryOperation<object, bool>

{

public override bool Calculate(object arg1, object arg2)

{

return arg1.Equals(arg2);

}

}

}

namespace Shaykhullin.Operations

{

public class BinaryGreaterOperation

: BinaryOperation<double, bool>

{

public override bool Calculate(double arg1, double arg2)

{

return arg1 > arg2;

}

}

}

namespace Shaykhullin.Operations

{

public class BinaryGreaterOrEqualOperation

: BinaryOperation<double, bool>

{

public override bool Calculate(double arg1, double arg2)

{

return arg1 >= arg2;

}

}

}

namespace Shaykhullin.Operations

{

public class BinaryLessOperation

: BinaryOperation<double, bool>

{

public override bool Calculate(double arg1, double arg2)

{

return arg1 < arg2;

}

}

}

namespace Shaykhullin.Operations

{

public class BinaryLessOrEqualOperation

: BinaryOperation<double, bool>

{

public override bool Calculate(double arg1, double arg2)

{

return arg1 <= arg2;

}

}

}

namespace Shaykhullin.Operations

{

public class BinaryMaxOperation

: BinaryOperation<double, double>

{

public override double Calculate(double arg1, double arg2)

{

return arg1 > arg2 ? arg1 : arg2;

}

}

}

namespace Shaykhullin.Operations

{

public class BinaryMinOperation

: BinaryOperation<double, double>

{

public override double Calculate(double arg1, double arg2)

{

return arg1 > arg2 ? arg2 : arg1;

}

}

}

namespace Shaykhullin.Operations

{

public class BinaryMinusOperation

: BinaryOperation<double, double>

{

public override double Calculate(double arg1, double arg2)

{

return arg1 - arg2;

}

}

}

namespace Shaykhullin.Operations

{

public class BinaryModulusOperation

: BinaryOperation<double, double>

{

public override double Calculate(double arg1, double arg2)

{

return arg1 % arg2;

}

}

}

namespace Shaykhullin.Operations

{

public class BinaryMultiplyOperation

: BinaryOperation<double, double>

{

public override double Calculate(double arg1, double arg2)

{

return arg1 \* arg2;

}

}

}

namespace Shaykhullin.Operations

{

public class BinaryNotEqualOperation

: BinaryOperation<object, bool>

{

public override bool Calculate(object arg1, object arg2)

{

return !(arg1.Equals(arg2));

}

}

}

namespace Shaykhullin.Operations

{

public abstract class BinaryOperation : Operation

{

public abstract object ExecuteCore(object arg1, object arg2);

}

public abstract class BinaryOperation<TArgument, TReturn> : BinaryOperation

{

public sealed override object ExecuteCore(object arg1, object arg2)

{

return Calculate((TArgument)arg1, (TArgument)arg2);

}

public abstract TReturn Calculate(TArgument arg1, TArgument arg2);

}

}

namespace Shaykhullin.Operations

{

public class BinaryOrOperation

: BinaryOperation<bool, bool>

{

public override bool Calculate(bool arg1, bool arg2)

{

return arg1 | arg2;

}

}

}

namespace Shaykhullin.Operations

{

public class BinaryPlusOperation

: BinaryOperation<double, double>

{

public override double Calculate(double arg1, double arg2)

{

return arg1 + arg2;

}

}

}

namespace Shaykhullin.Operations

{

public class BinaryXorOperation

: BinaryOperation<bool, bool>

{

public override bool Calculate(bool arg1, bool arg2)

{

return arg1 ^ arg2;

}

}

}

using System;

namespace Shaykhullin.Operations

{

public class UnaryAbsOperation

: UnaryOperation<double, double>

{

public override double Calculate(double arg)

{

return Math.Abs(arg);

}

}

}

using System;

namespace Shaykhullin.Operations

{

public class UnaryArccosOperation : UnaryOperation<double, double>

{

public override double Calculate(double arg)

{

return Math.Acos(arg);

}

}

}

using System;

namespace Shaykhullin.Operations

{

public class UnaryArcsinOperation

: UnaryOperation<double, double>

{

public override double Calculate(double arg)

{

return Math.Asin(arg);

}

}

}

using System;

namespace Shaykhullin.Operations

{

public class UnaryArctanOperation

: UnaryOperation<double, double>

{

public override double Calculate(double arg)

{

return Math.Atan(arg);

}

}

}

using System;

namespace Shaykhullin.Operations

{

public class UnaryCeilOperation

: UnaryOperation<double, double>

{

public override double Calculate(double arg)

{

return Math.Ceiling(arg);

}

}

}

using System;

namespace Shaykhullin.Operations

{

public class UnaryCosOperation

: UnaryOperation<double, double>

{

public override double Calculate(double arg)

{

return Math.Cos(arg);

}

}

}

using System;

namespace Shaykhullin.Operations

{

public class UnaryCotanOperation

: UnaryOperation<double, double>

{

public override double Calculate(double arg)

{

return 1d / Math.Tan(arg);

}

}

}

using System;

namespace Shaykhullin.Operations

{

public class UnaryFloorOperation

: UnaryOperation<double, double>

{

public override double Calculate(double arg)

{

return Math.Floor(arg);

}

}

}

using System;

namespace Shaykhullin.Operations

{

public class UnaryLnOperation

: UnaryOperation<double, double>

{

public override double Calculate(double arg)

{

return Math.Log(arg);

}

}

}

namespace Shaykhullin.Operations

{

public class UnaryMinusOperation

: UnaryOperation<double, double>

{

public override double Calculate(double arg)

{

return -arg;

}

}

}

namespace Shaykhullin.Operations

{

public class UnaryNotOperation

: UnaryOperation<bool, bool>

{

public override bool Calculate(bool arg)

{

return !arg;

}

}

}

namespace Shaykhullin.Operations

{

public abstract class UnaryOperation : Operation

{

public abstract object ExecuteCore(object arg);

}

public abstract class UnaryOperation<TArgument, TReturn> : UnaryOperation

{

public sealed override object ExecuteCore(object arg)

{

return Calculate((TArgument)arg);

}

public abstract TReturn Calculate(TArgument arg);

}

}

namespace Shaykhullin.Operations

{

public class UnaryPlusOperation

: UnaryOperation<double, double>

{

public override double Calculate(double arg)

{

return arg;

}

}

}

using System;

namespace Shaykhullin.Operations

{

public class UnaryPowerOperation

: UnaryOperation<double, double>

{

public override double Calculate(double arg)

{

return Math.Pow(arg, 2);

}

}

}

using System;

namespace Shaykhullin.Operations

{

public class UnaryRoundOperation

: UnaryOperation<double, double>

{

public override double Calculate(double arg)

{

return Math.Round(arg);

}

}

}

using System;

namespace Shaykhullin.Operations

{

public class UnarySignOperation

: UnaryOperation<double, double>

{

public override double Calculate(double arg)

{

return Math.Sign(arg);

}

}

}

using System;

namespace Shaykhullin.Operations

{

public class UnarySinOperation

: UnaryOperation<double, double>

{

public override double Calculate(double arg)

{

return Math.Sin(arg);

}

}

}

using System;

namespace Shaykhullin.Operations

{

public class UnaryTanOperation

: UnaryOperation<double, double>

{

public override double Calculate(double arg)

{

return Math.Tan(arg);

}

}

}

using System;

namespace Shaykhullin.Operations

{

public class UnaryTruncOperation

: UnaryOperation<double, double>

{

public override double Calculate(double arg)

{

return Math.Truncate(arg);

}

}

}

namespace Shaykhullin.Operations

{

public class BooleanValue : ValueOperation<bool>

{

public BooleanValue(bool value) : base(value)

{

}

}

}

namespace Shaykhullin.Operations

{

public class DoubleValue : ValueOperation<double>

{

public DoubleValue(double value) : base(value)

{

}

}

}

namespace Shaykhullin.Operations

{

public abstract class ValueOperation : Operation

{

public abstract object ExecuteCore();

}

public abstract class ValueOperation<TValue> : ValueOperation

{

private TValue value;

protected ValueOperation(TValue value)

{

this.value = value;

}

public sealed override object ExecuteCore()

{

return Calculate();

}

public TValue Calculate()

{

return value;

}

}

}

namespace Shaykhullin.Operations

{

public abstract class Operation

{

}

}