

|  |  |
| --- | --- |
| МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ  федеральное государственное бюджетное образовательное учреждение высшего образования  **«МОСКОВСКИЙ АВТОМОБИЛЬНО-ДОРОЖНЫЙ ГОСУДАРСТВЕННЫЙ**  **ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ (МАДИ)»** |  |

**Кафедра «Высшая математика»**

**Лабораторная работа № 2**

по дисциплине

«Структуры и алгоритмы обработки данных»

на тему:

**«**Работа с инфиксными/постфиксными выражениями**»**

**Выполнил:**

Учебная группа: 1бПМ   
ФИО: Греча К. П.

Подпись: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Руководитель лабораторной работы:**

Должность: старший преподаватель

Звание: б/з

ФИО: Кутейников И. А.

Подпись: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

«\_\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_20\_\_г.

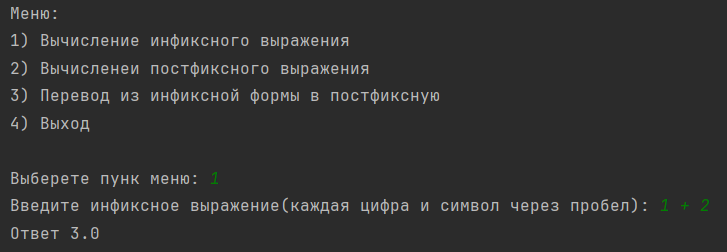
Цель:

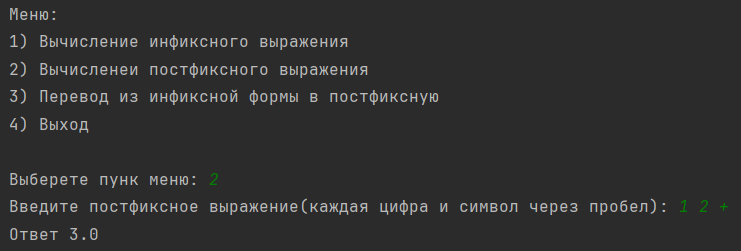
Разработать программу для вычисления инфиксного и постфиксного выражения введенного пользователем. Также предусмотреть перевод инфиксного выражения в постфиксное.

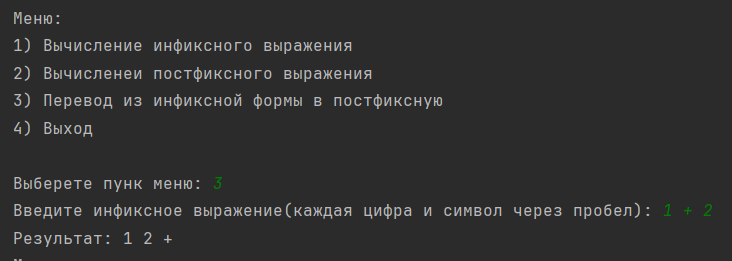
Алгоритм:

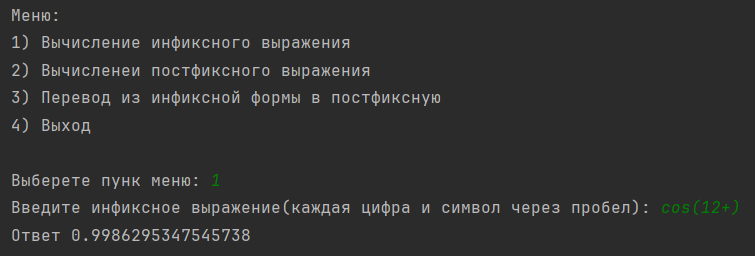
import java.util.Queue;  
import java.util.Scanner;  
import java.util.Stack;  
import java.util.LinkedList;  
import java.util.regex.Matcher;  
import java.util.regex.Pattern;  
import java.util.function.BiFunction;  
import java.util.HashMap;  
import java.util.\*;  
  
public class Main {  
  
 public static void main(String[] args) {  
  
 int ans;  
 String[] postfixExspression, infixExspression;  
  
 do {  
 Scanner in = new Scanner(System.*in*);  
 System.*out*.println("Меню:");  
 System.*out*.println("1) Вычисление инфиксного выражения");  
 System.*out*.println("2) Вычисленеи постфиксного выражения");  
 System.*out*.println("3) Перевод из инфиксной формы в постфиксную");  
 System.*out*.println("4) Выход\n");  
 System.*out*.print("Выберете пунк меню: ");  
 ans = in.nextInt();  
 switch (ans) {  
 case 1:  
 System.*out*.print("Введите инфиксное выражение(каждая цифра и символ через пробел): ");  
 in.nextLine(); //очищаем буфер  
 String inFixString = in.nextLine(); //считываем строку  
 infixExspression = inFixString.split(" ");//разбиваем на слова  
 *convFunction*(infixExspression);  
 System.*out*.println("Ответ " + *calcInfix*(infixExspression));  
 break;  
 case 2:  
 System.*out*.print("Введите постфиксное выражение(каждая цифра и символ через пробел): ");  
 in.nextLine(); // очищаем буфер  
 String postfixstr = in.nextLine();  
 postfixExspression = postfixstr.split(" ");  
 *convFunction*(postfixExspression);  
 System.*out*.println("Ответ " + *calcPostfix*(postfixExspression));  
 break;  
 case 3:  
 System.*out*.print("Введите инфиксное выражение(каждая цифра и символ через пробел): ");  
 in.nextLine(); // очищаем буфер  
 String infixstr = in.nextLine();  
 infixExspression = infixstr.split(" ");  
 *convFunction*(infixExspression);  
 System.*out*.println("Результат: " + *convInFixToPostFix*(infixExspression));  
 break;  
 }  
 } while (ans != 4);  
 }  
  
 //функция с заменой значениями  
 public static void convFunction(String[] queue) {  
 String regular = "(sin|cos|tg|ctg|exp)\\(([^)]+)\\)";  
 Pattern pattern = Pattern.*compile*(regular);  
  
 for (int i = 0; i < queue.length; i++) {  
 Matcher matcher = pattern.matcher(queue[i]);  
 while (matcher.find()) {  
 String function = matcher.group(1);  
 String innerExpression = matcher.group(2);  
  
 // Преобразование строки в инфиксной форме в постфиксную.  
 String[] tokens = *tokenize*(innerExpression);  
 innerExpression = *convInFixToPostFix*(tokens);  
  
 String[] postfixArray = innerExpression.split(" ");  
 double evaluatedValue = *calcPostfix*(postfixArray);  
 double functionResult = *calculateFunction*(function, evaluatedValue);  
 queue[i] = String.*valueOf*(functionResult);  
 }  
 }  
 }  
  
 private static String[] tokenize(String expression) {  
 List<String> tokens = new ArrayList<>();  
  
 for (int i = 0; i < expression.length(); i++) {  
 char ch = expression.charAt(i);  
  
 // Если текущий символ является частью числа (цифра или точка) или оператором  
 if (Character.*isDigit*(ch) || ch == '.' || *isOperator*(ch)) {  
 tokens.add(Character.*toString*(ch)); // Добавляем каждый символ как отдельный токен  
 }  
 }  
  
 // Возвращаем полученные токены в виде массива  
 return tokens.toArray(new String[0]);  
 }  
  
 // Вспомогательный метод для проверки, является ли символ оператором  
 private static boolean isOperator(char ch) {  
 return ch == '+' || ch == '-' || ch == '\*' || ch == '/' || ch == '^';  
 }  
  
 private static double calculateFunction(String function, double argument) {  
 // Вычисление тригонометрических и экспоненциальной функций.  
 switch (function) {  
 case "sin":  
 return Math.*sin*(Math.*toRadians*(argument));  
 case "cos":  
 return Math.*cos*(Math.*toRadians*(argument));  
 case "tg":  
 return Math.*tan*(Math.*toRadians*(argument));  
 case "ctg":  
 return 1.0 / Math.*tan*(Math.*toRadians*(argument));  
 case "exp":  
 return Math.*exp*(argument);  
 default:  
 throw new IllegalArgumentException("Неизвестная функция: " + function);  
 }  
 }  
  
 public static double applyOperation(double operand1, double operand2, String operator) {  
 switch (operator) {  
 case "+":  
 return operand1 + operand2;  
 case "-":  
 return operand1 - operand2;  
 case "\*":  
 return operand1 \* operand2;  
 case "/":  
 if (operand2 == 0) {  
 throw new ArithmeticException("Нельзя делить на 0");  
 }  
 return operand1 / operand2;  
 case "^":  
 return Math.*pow*(operand1, operand2);  
 default:  
 throw new IllegalArgumentException("Неверный оператор: " + operator);  
 }  
 }  
  
 public static int priority(String operator) {  
 switch (operator) {  
 case "^":  
 return 3;  
 case "\*":  
 case "/":  
 return 2;  
 case "+":  
 case "-":  
 return 1;  
 default:  
 return 0;  
 }  
 }  
  
 // Метод для вычисления инфиксного выражения  
 public static double calcInfix(String[] tokens) {  
 Stack<Double> operands = new Stack<>(); // Стек операндов  
 Stack<String> operators = new Stack<>(); // Стек операторов  
 Map<String, BiFunction<Double, Double, Double>> operationMap = new HashMap<>(); // Словарь операций  
 // Заполнение словаря операций  
 operationMap.put("+", (a, b) -> a + b);  
 operationMap.put("-", (a, b) -> a - b);  
 operationMap.put("\*", (a, b) -> a \* b);  
 operationMap.put("/", (a, b) -> a / b);  
  
 Map<String, Integer> precedenceMap = new HashMap<>(); // Словарь приоритетов операций  
 // Заполнение словаря приоритетов операций  
 precedenceMap.put("+", 1);  
 precedenceMap.put("-", 1);  
 precedenceMap.put("\*", 2);  
 precedenceMap.put("/", 2);  
 precedenceMap.put("^", 3);  
  
 for (String token : tokens) {  
 if (token.matches("\\-?\\d+(\\.\\d+)?")) { // Если токен - число  
 operands.push(Double.*parseDouble*(token));  
 } else if (token.equals("(")) { // Если токен - открывающая скобка  
 operators.push(token);  
 } else if (token.equals(")")) { // Если токен - закрывающая скобка  
 while (!operators.peek().equals("(")) {  
 if (operators.isEmpty()) {  
 throw new IllegalStateException("Некорректное количество скобок");  
 }  
 operands.push(*applyOperation*(operands.pop(), operands.pop(), operators.pop(), operationMap));  
 }  
 operators.pop();  
 } else if (precedenceMap.containsKey(token)) { // Если токен - оператор  
 while (!operators.isEmpty() && precedenceMap.get(token) <= precedenceMap.get(operators.peek())) {  
 operands.push(*applyOperation*(operands.pop(), operands.pop(), operators.pop(), operationMap));  
 }  
 operators.push(token);  
 }  
 }  
  
 while (!operators.isEmpty()) { // Выполнение оставшихся операций  
 operands.push(*applyOperation*(operands.pop(), operands.pop(), operators.pop(), operationMap));  
 }  
  
 return operands.pop(); // Возвращение результата  
 }  
  
 // Метод для применения операции к операндам  
 private static double applyOperation(double operand2, double operand1, String operator,  
 Map<String, BiFunction<Double, Double, Double>> operationMap) {  
 if (!operationMap.containsKey(operator)) {  
 throw new IllegalArgumentException("Invalid operator: " + operator);  
 }  
 return operationMap.get(operator).apply(operand1, operand2);  
 }  
  
 // Метод для вычисления постфиксного выражения  
 public static double calcPostfix(String[] queue) {  
 Stack<Double> stack = new Stack<>(); // Стек для выполнения операций  
 for (String item : queue) {  
 if (item.matches("\\-?\\d+(\\.\\d+)?")) { // Если элемент - число  
 stack.push(Double.*parseDouble*(item));  
 } else if (stack.size() < 2) { // Если в стеке недостаточно операндов  
 throw new IllegalArgumentException("Недостаточно операндов в стеке для операции " + item);  
 } else { // Если элемент - оператор  
 Double operand2 = stack.pop();  
 Double operand1 = stack.pop();  
 double result = *applyOperation*(operand1, operand2, item);  
 stack.push(result);  
 }  
 }  
 if (stack.size() != 1) { // Проверка на корректность вычислений  
 throw new IllegalStateException("В стеке должен остаться один элемент после вычисления");  
 }  
 return stack.pop(); // Возвращение результата  
 }  
  
 // Метод для преобразования инфиксного выражения в постфиксное  
 public static String convInFixToPostFix(String[] infixTokens) {  
 Stack<String> operators = new Stack<>(); // Стек операторов  
 Queue<String> postfix = new LinkedList<>(); // Очередь для постфиксного выражения  
 for (String token : infixTokens) {  
 if (token.matches("\\-?\\d+(\\.\\d+)?")) { // Если элемент - число  
 postfix.add(token);  
 } else if (token.equals("(")) { // Если токен - открывающая скобка  
 operators.push(token);  
 } else if (token.equals(")")) { // Если токен - закрывающая скобка  
 while (!operators.isEmpty() && !operators.peek().equals("(")) {  
 postfix.add(operators.pop());  
 }  
 if (operators.isEmpty()) {  
 throw new IllegalStateException("Стек операторов пуст");  
 }  
 operators.pop(); // Удаление открывающей скобки из стека  
 } else { // Если токен - оператор  
 while (!operators.isEmpty() && *priority*(token) <= *priority*(operators.peek())) {  
 postfix.add(operators.pop());  
 }  
 operators.push(token); // Добавление текущего оператора в стек  
 }  
 }  
 while (!operators.isEmpty()) { // Добавление оставшихся операторов в очередь  
 postfix.add(operators.pop());  
 }  
 return String.*join*(" ", postfix); // Возвращение постфиксного выражения в виде строки  
 }  
}

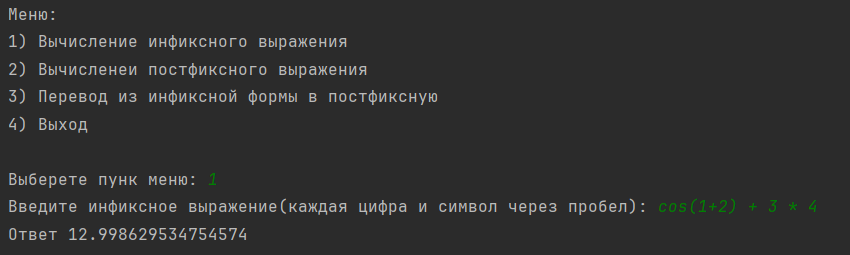
**Результат:**











**Заключение:**

В процессе выполнения лабораторной работы были изучены методы вычисления инфиксного, постфиксного выражения, перевод инфиксной формы в постфиксную. Была реализована передача функции аргументом инфиксное и постфиксное выражение, а также вычисления значения данной функции.