Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерного проектирования

Кафедра инженерной психологии и эргономики

Отчет

по практической работе №3

на тему:

**ФУНКЦИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ И ЛЯМБДА ВЫРАЖЕНИЯ**

Проверил \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Усенко Ф.В.

(подпись)

Выполнил \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Папко К.А.

(подпись)  гр. 214301

Минск, 2024

**Задание:** Напишите программу, которая позволяет пользователю добавлять новые арифметические операции в виде лямбда-выражений и применять их к выражениям. Программа должна поддерживать сохранение и загрузку этих операций.

1. ВЫПОЛНЕНИЕ ЗАДАНИЯ

import java.io.File  
import java.io.ObjectInputStream  
import java.io.ObjectOutputStream  
import kotlin.math.\*  
  
fun main() {  
 val calculator = DynamicCalculator()  
 val operationsFile = "operations\_multi.dat"  
  
 calculator.loadOperations(operationsFile)  
  
 while (true) {  
 println("\n--- Калькулятор ---")  
 println("1. Выполнить операцию")  
 println("2. Добавить новую операцию")  
 println("3. Сохранить операции")  
 println("4. Выйти")  
 print("Выберите действие: ")  
  
 when (readLine()?.toIntOrNull()) {  
 1 -> {  
 println("Доступные операции: ${calculator.listOperations()}")  
 print("Введите название операции: ")  
 val operation = readLine()  
  
 if (operation != null && calculator.hasOperation(operation)) {  
 val variables = calculator.getVariables(operation)  
 val values = mutableMapOf<String, Double>()  
  
 for (variable in variables) {  
 print("Введите значение для $variable: ")  
 val value = readLine()?.toDoubleOrNull()  
 if (value != null) {  
 values[variable] = value  
 } else {  
 println("Ошибка: некорректный ввод!")  
 break  
 }  
 }  
  
 val result = calculator.performOperation(operation, values)  
 if (result != null) {  
 println("Результат: $result")  
 } else {  
 println("Ошибка выполнения операции!")  
 }  
 } else {  
 println("Операция не найдена!")  
 }  
 }  
  
 2 -> {  
 print("Введите название операции: ")  
 val name = readLine()  
 print("Введите выражение для операции (например, a \* b + c / d - e): ")  
 val expression = readLine()  
  
 if (name != null && expression != null) {  
 try {  
 calculator.addOperation(name, expression)  
 println("Операция добавлена!")  
 } catch (e: IllegalArgumentException) {  
 println("Ошибка: ${e.message}")  
 }  
 } else {  
 println("Ошибка: некорректный ввод!")  
 }  
 }  
  
 3 -> {  
 calculator.saveOperations(operationsFile)  
 println("Операции сохранены!")  
 }  
  
 4 -> {  
 println("Выход из программы.")  
 break  
 }  
  
 else -> println("Ошибка: неверный выбор!")  
 }  
 }  
}  
  
class DynamicCalculator {  
 private val operations = mutableMapOf<String, String>()  
  
 fun addOperation(name: String, expression: String) {  
 val variables = extractVariables(expression)  
 if (variables.isEmpty()) {  
 throw IllegalArgumentException("Выражение должно содержать хотя бы одну переменную.")  
 }  
 operations[name] = expression  
 }  
  
 fun performOperation(name: String, values: Map<String, Double>): Double? {  
 val expression = operations[name] ?: return null  
 val parsedExpression = expression.replaceVariables(values)  
 return evaluateExpression(parsedExpression)  
 }  
  
 fun listOperations(): List<String> {  
 return operations.keys.toList()  
 }  
  
 fun hasOperation(name: String): Boolean {  
 return operations.containsKey(name)  
 }  
  
 fun getVariables(name: String): List<String> {  
 val expression = operations[name] ?: return emptyList()  
 return extractVariables(expression)  
 }  
  
 fun saveOperations(fileName: String) {  
 ObjectOutputStream(File(fileName).outputStream()).use **{** it.writeObject(operations) **}** }  
  
 fun loadOperations(fileName: String) {  
 val file = File(fileName)  
 if (file.exists()) {  
 ObjectInputStream(file.inputStream()).use **{** @Suppress("UNCHECKED\_CAST")  
 operations.putAll(it.readObject() as Map<String, String>)  
 **}** }  
 }  
  
 private fun extractVariables(expression: String): List<String> {  
 val regex = Regex("[a-zA-Z]+")  
 return regex.findAll(expression).map **{** it.value **}**.toSet().toList()  
 }  
  
 private fun String.replaceVariables(values: Map<String, Double>): String {  
 var newExpression = this  
 for ((variable, value) in values) {  
 newExpression = newExpression.replace(variable, value.toString())  
 }  
 return newExpression  
 }  
  
 private fun evaluateExpression(expression: String): Double {  
 val regex = Regex("[0-9.]+|[+\\-\*/^()]")  
 val tokens = regex.findAll(expression).map **{** it.value **}**.toList()  
 val valuesStack = mutableListOf<Double>()  
 val operatorsStack = mutableListOf<Char>()  
  
 val precedence = mapOf(  
 '+' to 1,  
 '-' to 1,  
 '\*' to 2,  
 '/' to 2,  
 '^' to 3  
 )  
  
 for (token in tokens) {  
 when {  
 token.isNumber() -> valuesStack.add(token.toDouble())  
 token == "(" -> operatorsStack.add('(')  
 token == ")" -> {  
 while (operatorsStack.isNotEmpty() && operatorsStack.last() != '(') {  
 performOperationOnStacks(valuesStack, operatorsStack)  
 }  
 operatorsStack.removeAt(operatorsStack.size - 1)  
 }  
 token.isOperator() -> {  
 while (operatorsStack.isNotEmpty() && operatorsStack.last() != '(' &&  
 precedence[token[0]]!! <= precedence[operatorsStack.last()]!!) {  
 performOperationOnStacks(valuesStack, operatorsStack)  
 }  
 operatorsStack.add(token[0])  
 }  
 }  
 }  
  
 while (operatorsStack.isNotEmpty()) {  
 performOperationOnStacks(valuesStack, operatorsStack)  
 }  
  
 return valuesStack.last()  
 }  
  
 private fun performOperationOnStacks(valuesStack: MutableList<Double>, operatorsStack: MutableList<Char>) {  
 val b = valuesStack.removeAt(valuesStack.size - 1)  
 val a = valuesStack.removeAt(valuesStack.size - 1)  
 val operator = operatorsStack.removeAt(operatorsStack.size - 1)  
 val result = applyOperator(a, b, operator)  
 valuesStack.add(result)  
 }  
  
 private fun applyOperator(a: Double, b: Double, operator: Char): Double {  
 return when (operator) {  
 '+' -> a + b  
 '-' -> a - b  
 '\*' -> a \* b  
 '/' -> a / b  
 '^' -> Math.pow(a, b)  
 else -> throw IllegalArgumentException("Неизвестный оператор: $operator")  
 }  
 }  
  
 private fun String.isNumber(): Boolean = this.toDoubleOrNull() != null  
 private fun String.isOperator(): Boolean = this.length == 1 && this[0] in "+-\*/^"  
}

1. ОТВЕТЫ НА ВОПРОСЫ

* Как объявить функцию в Kotlin? В чем разница между обычной функцией и однострочной функцией?  
  Функция в Kotlin объявляется с использованием ключевого слова fun, за которым следуют имя функции, параметры и тело. Обычная функция может содержать несколько строк кода, заключенных в фигурные скобки, а однострочная функция имеет компактный вид и возвращает результат выражения без использования фигурных скобок.
* Что такое функция высшего порядка, и как её использовать?  
  Функция высшего порядка — это функция, которая принимает другую функцию в качестве параметра или возвращает функцию как результат. Она используется для повышения гибкости и повторного использования кода.
* Как передать функцию в качестве параметра другой функции?  
  Для передачи функции в качестве параметра нужно указать её тип в сигнатуре принимающей функции. Например, fun calculate(a: Int, b: Int, operation: (Int, Int) -> Int).
* Чем отличается лямбда-выражение от обычной функции?  
  Лямбда-выражение — это компактная анонимная функция, которая не требует имени, объявления с fun, или явного указания возвращаемого типа, если его можно вывести автоматически.
* Как создать лямбда-выражение с двумя параметрами?  
  Лямбда-выражение с двумя параметрами создается с использованием синтаксиса { параметр1, параметр2 -> тело\_выражения }.

1. ВЫВОД

В ходе выполнения практической работы №3 были изучены основы функционального программирования на языке Kotlin, в том числе создание и использование лямбда-выражений и функций высшего порядка.