Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерного проектирования

Кафедра проектирования информационно-компьютерных систем

Дисциплина Программирование мобильных информационных систем

Отчет

по лабораторной работе

на тему:

**«ФУНКЦИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ И ЛЯМБДА-ВЫРАЖЕНИЯ»**

Выполнил \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Булатевич В.Ю.

(подпись)

гр. 214301

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

(подпись)

Минск, 2024

**Цель работы:** Ознакомиться с основами функционального программирования в Kotlin, изучить лямбда-выражения, анонимные функции и замыкания. Научиться использовать эти концепции для написания более гибкого и читаемого кода.

**Задание**: Реализуйте программу, которая принимает математическое выражение в виде строки, разбивает его на компоненты и использует лямбда-выражения для его вычисления. Программа должна поддерживать сложные выражения с вложенными скобками.

**1 Реализация задания**

import import java.util.Stack  
  
fun main() {  
 *println*("Калькулятор. Введите 'exit' для выхода.")  
  
 while (true) {  
 *print*("Введите математическое выражение: ")  
 val input = *readlnOrNull*()?.*trim*()  
  
 if (input.*isNullOrEmpty*()) {  
 *println*("Пустой ввод. Попробуйте снова.")  
 continue  
 }  
  
 if (input.*equals*("exit", ignoreCase = true)) {  
 *println*("Выход из программы.")  
 break  
 }  
  
 try {  
 val result = *evaluateExpression*(input)  
 *println*("Результат: $result")  
 } catch (e: Exception) {  
 *println*("Ошибка: ${e.message}")  
 }  
 }  
}  
  
fun evaluateExpression(expression: String): Double {  
 val tokens = *tokenize*(expression)  
 return *parse*(tokens)  
}  
  
fun tokenize(expression: String): List<String> {  
 val tokens = *mutableListOf*<String>()  
 var numberBuffer = StringBuilder()  
  
 for (char in expression) {  
 when {  
 char.*isDigit*() || char == '.' -> numberBuffer.append(char)  
 char.*isWhitespace*() -> continue  
 else -> {  
 if (numberBuffer.*isNotEmpty*()) {  
 tokens.add(numberBuffer.toString())  
 numberBuffer = StringBuilder()  
 }  
 if (char in "+-\*/()") tokens.add(char.toString())  
 }  
 }  
 }  
  
 if (numberBuffer.*isNotEmpty*()) tokens.add(numberBuffer.toString())  
 return tokens  
}  
  
fun parse(tokens: List<String>): Double {  
 val stack = Stack<Double>()  
 val operatorStack = Stack<String>()  
 val precedence = *mapOf*(  
 "+" *to* 1, "-" *to* 1,  
 "\*" *to* 2, "/" *to* 2  
 )  
  
 fun applyOperation() {  
 if (stack.size < 2 || operatorStack.isEmpty()) {  
 throw IllegalArgumentException("Некорректное выражение")  
 }  
 val b = stack.pop()  
 val a = stack.pop()  
  
 val result = when (val operator = operatorStack.pop()) {  
 "+" -> a + b  
 "-" -> a - b  
 "\*" -> a \* b  
 "/" -> {  
 if (b == 0.0) throw ArithmeticException("Деление на ноль")  
 a / b  
 }  
 else -> throw IllegalArgumentException("Неизвестный оператор $operator")  
 }  
 stack.push(result)  
 }  
  
 for (i in tokens.*indices*) {  
 val token = tokens[i]  
 when {  
 token.*toDoubleOrNull*() != null -> stack.push(token.*toDouble*())  
 token in precedence.keys -> {  
 while (  
 operatorStack.*isNotEmpty*() &&  
 precedence[token]!! <= precedence[operatorStack.peek()]!!  
 ) {  
 applyOperation()  
 }  
 operatorStack.push(token)  
 }  
 token == "(" -> {  
 val subExpression = *mutableListOf*<String>()  
 var openBrackets = 1  
 var j = i + 1  
 while (j < tokens.size) {  
 val subToken = tokens[j]  
 if (subToken == "(") openBrackets++  
 if (subToken == ")") openBrackets--  
 if (openBrackets == 0) break  
 subExpression.add(subToken)  
 j++  
 }  
 if (openBrackets != 0) {  
 throw IllegalArgumentException("Пропущена закрывающая скобка")  
 }  
 stack.push(*parse*(subExpression))  
 }  
 token == ")" -> throw IllegalArgumentException("Неожиданная закрывающая скобка")  
 else -> throw IllegalArgumentException("Неизвестный токен $token")  
 }  
 }  
  
  
 while (operatorStack.*isNotEmpty*()) {  
 applyOperation()  
 }  
  
 if (stack.size != 1) {  
 throw IllegalArgumentException("Некорректное выражение")  
 }  
  
 return stack.pop()  
}

**2 Ответы на контрользые вопросы**

1 Как объявить функцию в Kotlin? В чем разница между обычной функцией и однострочной функцией?  
Функция в Kotlin объявляется с помощью ключевого слова fun, за которым следует имя функции, параметры и тип возвращаемого значения. Обычная функция может содержать несколько инструкций в теле, заключённых в фигурные скобки. Однострочная функция, напротив, имеет компактный синтаксис с использованием символа = для указания результата выполнения.

2 Что такое функция высшего порядка, и как её использовать? Приведите пример.  
Функция высшего порядка — это функция, которая принимает другую функцию в качестве параметра или возвращает функцию. Пример: fun calculate(a: Int, b: Int, operation: (Int, Int) -> Int): Int { return operation(a, b) }.

3 Как передать функцию в качестве параметра другой функции? Приведите пример.  
Функцию можно передать в качестве параметра, указав её тип в сигнатуре функции. Например: val sum = { x: Int, y: Int -> x + y }; calculate(5, 3, sum).

4 Чем отличается лямбда-выражение от обычной функции?  
Лямбда-выражение — это анонимная функция, которая может быть объявлена "на месте" без имени. Оно имеет более компактный синтаксис и часто используется для передачи логики в другие функции. Обычная функция имеет имя и объявляется в контексте класса или файла.

5 Как создать лямбда-выражение с двумя параметрами? Приведите пример.  
Лямбда-выражение с двумя параметрами создаётся так: { x: Int, y: Int -> x + y }.

6 Что такое анонимная функция, и как она отличается от лямбда-выражения?  
Анонимная функция — это функция без имени, объявленная с использованием ключевого слова fun. Она может иметь явный тип возвращаемого значения, в отличие от лямбда-выражения.

7 Какие возможности предоставляют анонимные функции в Kotlin?  
Анонимные функции позволяют передавать сложные логики в качестве аргументов, определять тип возвращаемого значения и могут использоваться там, где лямбда-выражение неудобно из-за ограничения на типы или структуры.

8 Что такое замыкание, и как оно работает? Приведите пример использования.  
Замыкание — это функция, которая захватывает переменные из внешней области видимости и сохраняет их для дальнейшего использования. Пример: fun makeMultiplier(factor: Int): (Int) -> Int { return { x -> x \* factor } }.

9 Как лямбда-выражения могут захватывать переменные из внешней области видимости?  
Лямбда-выражения могут захватывать переменные из внешней области видимости, потому что они сохраняют ссылку на эту область. Например, лямбда, используемая внутри функции, может продолжать использовать локальные переменные этой функции даже после её завершения.

10 В чем преимущества использования функций высшего порядка в программировании?  
Функции высшего порядка делают код более гибким и повторно используемым, упрощают обработку данных, позволяют легко передавать логику в виде аргументов и способствуют более функциональному подходу к программированию.

**Вывод**

В ходе изучения функций в Kotlin были рассмотрены различные подходы к их созданию и использованию. Понято, что функции могут быть обычными и однострочными, где последние позволяют писать компактный код. Особое внимание уделено функциям высшего порядка, которые позволяют принимать или возвращать другие функции, делая код более универсальным и удобным для повторного использования.