Министерство образования Республики Беларусь

Учреждение образования «Белорусский государственный университет информатики и радиоэлектроники»

Факультет компьютерного проектирования

Кафедра инженерной психологии и эргономики

Дисциплина: Программирование мобильных информационных систем

Отчёт

по лабораторной работе №3

на тему

**Функциональное программирование и лямбда-выражения**

Выполнил: Проверил:

ст. гр. 214302 Усенко Ф.В.

Ковзович Н.С.

Минск 2024

Задание: Функциональный парсер выражений: 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.

Листинг кода:

import java.util.\*  
import kotlin.math.\*  
  
fun main() {  
  
 println("\nЛогарифмы записываются: log\_a(b), где a - основание логарифма, b - аргумент логарифма.")  
 println("Натуральный логарифм: ln(b), где b - аргумент логарифма.")  
 println("Извлечь корень: sqrt(a), где a - неотрицательное число.")  
 println("PI: π - использовать для тригонометрических функций (π = 180 градусов), в остальных ситуациях: pi = 3,14... .")  
 println("Экспонента: e.")  
 println("Синус: sin(a), где a мб в радианах sin(2\*π), либо градусах sin(360)")  
 println("Косинус: cos(a), где a мб в радианах cos(2\*π), либо градусах cos(360)")  
 println("Тангенс: tan(a), где a мб в радианах tan(2\*π), либо градусах tan(360)")  
 println("Котангенс: cot(a), где a мб в радианах cot(π/2), либо градусах cot(90)")  
  
 while (true) {  
 println("\nВведите математическое выражение (или введите 0 для выхода): ")  
 val expression = readLine() ?: ""  
  
 if (expression == "0") break  
  
 try {  
 val result = evaluateExpression(expression)  
 println("Результат: $result")  
 } catch (e: Exception) {  
 println("Ошибка: ${e.message}")  
 }  
 }  
}  
  
val add: (Double, Double) -> Double = { a, b -> a + b }  
val subtract: (Double, Double) -> Double = { a, b -> a - b }  
val multiply: (Double, Double) -> Double = { a, b -> a \* b }  
val divide: (Double, Double) -> Double = { a, b ->  
 if (b == 0.0) throw IllegalArgumentException("Ошибка: Деление на ноль")  
 a / b  
}  
val power: (Double, Double) -> Double = { a, b -> a.pow(b) }  
val sqrtFunc: (Double) -> Double = { a ->  
 if (a < 0) throw IllegalArgumentException("Ошибка: Невозможно извлечь корень из отрицательного числа")  
 sqrt(a)  
}  
  
val epsilon = 1e-10 // Порог точности  
  
fun toRadians(degrees: Double): Double {  
 return Math.toRadians(degrees)  
}  
  
val sinFunc: (Double) -> Double = { value ->  
 sin(toRadians(value))  
}  
  
val cosFunc: (Double) -> Double = { value ->  
 cos(toRadians(value))  
}  
  
val tanFunc: (Double) -> Double = { value ->  
 if (abs(cos(toRadians(value))) < epsilon) throw IllegalArgumentException("Ошибка: Тангенс не определён для угла $value")  
 tan(toRadians(value))  
} // 90  
  
val cotFunc: (Double) -> Double = { value ->  
 if (abs(sin(toRadians(value))) < epsilon) throw IllegalArgumentException("Ошибка: Котангенс не определён для угла $value")  
 1 / tan(toRadians(value))  
} // 0  
  
val lnFunc: (Double) -> Double = { value -> ln(value) }  
val logFunc: (Double, Double) -> Double = { base, value ->  
 if (base <= 0 || base == 1.0) throw IllegalArgumentException("Некорректное основание логарифма: $base")  
 ln(value) / ln(base)  
}  
  
fun evaluateExpression(expr: String): Double {  
 val tokens = tokenize(expr.replace(" ", ""))  
 val result = parseExpression(tokens)  
 if (tokens.isNotEmpty()) throw IllegalArgumentException("Некорректное выражение")  
 return result  
}  
  
fun tokenize(expr: String): Queue<String> {  
 val tokens = LinkedList<String>()  
 var i = 0  
 while (i < expr.length) {  
 val c = expr[i]  
 when {  
 c.isDigit() -> {  
 val number = StringBuilder()  
 while (i < expr.length && (expr[i].isDigit() || expr[i] == '.')) {  
 number.append(expr[i])  
 i++  
 }  
 tokens.add(number.toString())  
 }  
 c == '+' || c == '-' || c == '\*' || c == '/' || c == '(' || c == ')' || c == '^' -> {  
 tokens.add(c.toString())  
 i++  
 }  
 c.isLetter() -> {  
 val function = StringBuilder()  
 while (i < expr.length && (expr[i].isLetter() || expr[i] == '\_' || expr[i].isDigit())) {  
 function.append(expr[i])  
 i++  
 }  
 tokens.add(function.toString())  
 }  
 else -> throw IllegalArgumentException("Некорректный символ: $c")  
 }  
 }  
 return tokens  
}  
  
fun parseExpression(tokens: Queue<String>): Double {  
 var result = parseTerm(tokens)  
 while (tokens.isNotEmpty() && (tokens.peek() == "+" || tokens.peek() == "-")) {  
 val operator = tokens.poll()  
 val nextTerm = parseTerm(tokens)  
 result = when (operator) {  
 "+" -> add(result, nextTerm)  
 "-" -> subtract(result, nextTerm)  
 else -> result  
 }  
 }  
 return result  
}  
  
fun parseTerm(tokens: Queue<String>): Double {  
 var result = parseFactor(tokens)  
 while (tokens.isNotEmpty() && (tokens.peek() == "\*" || tokens.peek() == "/")) {  
 val operator = tokens.poll()  
 val nextFactor = parseFactor(tokens)  
 result = when (operator) {  
 "\*" -> multiply(result, nextFactor)  
 "/" -> divide(result, nextFactor)  
 else -> result  
 }  
 }  
 return result  
}  
  
fun parseFactor(tokens: Queue<String>): Double {  
 var result = parseExponent(tokens)  
 while (tokens.isNotEmpty() && tokens.peek() == "^") {  
 tokens.poll()  
 val exponent = parseExponent(tokens)  
 result = power(result, exponent)  
 }  
 return result  
}  
  
fun parseExponent(tokens: Queue<String>): Double {  
 val token = tokens.poll() ?: throw IllegalArgumentException("Недостаточно аргументов")  
 return when {  
 token == "(" -> {  
 val result = parseExpression(tokens)  
 if (tokens.poll() != ")") throw IllegalArgumentException("Пропущена закрывающая скобка")  
 result  
 }  
 token.matches(Regex("-?\\d+(\\.\\d+)?")) -> token.toDouble()  
 token == "-" -> -parseExponent(tokens)  
 token == "π" -> 180.0  
 token == "pi" -> Math.PI  
 token == "e" -> Math.E  
 token == "sqrt" -> sqrtFunc(parseExponent(tokens))  
 token == "sin" -> sinFunc(parseExponent(tokens))  
 token == "cos" -> cosFunc(parseExponent(tokens))  
 token == "tan" -> tanFunc(parseExponent(tokens))  
 token == "cot" -> cotFunc(parseExponent(tokens))  
 token == "ln" -> lnFunc(parseExponent(tokens))  
 token.startsWith("log\_") -> {  
 val baseString = token.substring(4)  
 val base = baseString.toDoubleOrNull()  
 ?: throw IllegalArgumentException("Некорректное основание логарифма: $baseString")  
 logFunc(base, parseExponent(tokens))  
 }  
 else -> throw IllegalArgumentException("Некорректный токен: $token")  
 }  
}

Контрольные вопросы:

**что такое функция? что такое выражение? в чем разница?**

Функция — это блок кода, который принимает аргументы, выполняет определенные действия и возвращает результат. В Kotlin функции могут быть определены как стандартные (с ключевым словом `fun`) и анонимные (лямбда-функции).

Выражение — это часть кода, которая возвращает значение. Это может быть простое выражение, например, математическая операция, или более сложное, как вызов функции.

Разница:

- Функция — это самостоятельная конструкция, которая может содержать выражения и использоваться многократно.

- Выражение — это что-то, что всегда возвращает результат, и оно может быть частью функции.

**функции высокого порядка?**

Функции высокого порядка (high order function) — это функции, которые либо принимают функцию в качестве параметра, либо возвращают функцию, либо и то, и другое.

**лямбда-выражения?**

Лямбда-выражение — это функция без имени, которая может быть объявлена непосредственно в месте использования. Лямбды могут принимать параметры и возвращать значения. могут быть переданы как аргументы или возвращены из функций.

**анонимные функции?**

Анонимные функции похожи на лямбда-выражения, но могут иметь более сложное тело, включая несколько операторов. Анонимные функции определяются с использованием ключевого слова fun без имени функции. Они могут явно возвращать значения и указывать типы возвращаемых данных.

**как обрабатывать ошибки с помощью анонимных функций и лямбда-выражений?**

Чтобы обработать ошибки с помощью анонимных функций и лямбда-выражений, можно использовать конструкцию `try-catch`, которая перехватывает ошибки и позволяет обработать их.

Анонимные функции:

Анонимная функция — это функция без имени, которая может включать `try-catch` прямо внутри себя для обработки ошибок.

val divide = fun(a: Int, b: Int): Int {

return try {

a / b // может вызвать ошибку, если b = 0

} catch (e: ArithmeticException) {

0 // обработка ошибки деления на ноль

}

}

Лямбда-выражения:

Лямбда-выражения — это короткая форма анонимных функций, в них тоже можно использовать `try-catch` для обработки ошибок.

val divide = { a: Int, b: Int ->

try {

a / b // может вызвать ошибку, если b = 0

} catch (e: ArithmeticException) {

0 // обработка ошибки деления на ноль

}

}

**в чем отличие анонимной функции от лямбда-выражений?**

В отличие от лямбда-выражений, анонимные функции могут содержать в себе несколько инструкций и поддерживают доступ к меткам возврата (return).

Отличия:

Синтаксис: Лямбда-выражения более краткие, анонимные функции могут содержать полный синтаксис с return и типом возвращаемого значения.

Типы: В анонимных функциях можно явным образом указывать возвращаемые типы, что иногда делает их более гибкими.

**чем лямбды и анонимные отличаются от функций высшего порядка?**

Лямбда и анонимные функции — это конкретные типы функций без имени, которые можно передавать как значения. Они являются строительными блоками для создания простых операций.

Функции высшего порядка — это функции, которые работают с другими функциями, принимая их как аргументы или возвращая их. Внутри этих функций можно использовать лямбды или анонимные функции.

Лямбды и анонимные функции — это способ определения функций,

а функции высшего порядка — это способ использования функций.

**что такое замыкание? что такое автозамыкание?**

Замыкание — это функция (включая **лямбда-выражения** или **анонимные функции**), которая сохраняет ссылку на переменные из своей внешней области видимости, даже если внешняя функция уже завершила выполнение. Эти переменные остаются доступными для замыкания, и оно может с ними работать.

Основное отличие между обычным замыканием и автозамыканием заключается в способе захвата и использования переменных из внешней области видимости:

Обычное замыкание:

- Определение: Обычное замыкание — это функция (лямбда-выражение или анонимная функция), которая захватывает переменные из своей внешней области видимости и может использовать их в своем теле.

- Контекст: При создании замыкания переменные сохраняются и могут быть использованы, но их состояние зависит от конкретного момента создания замыкания. Если замыкание не изменяет переменные, они остаются неизменными в своей области видимости.

Автозамыкание:

- Определение: Автозамыкание также захватывает переменные из внешней области видимости, но делает это автоматически и сохраняет их состояние между вызовами замыкания. При этом замыкание всегда имеет доступ к актуальному значению захваченных переменных.

- Контекст: Автозамыкание подразумевает, что переменные могут изменяться и их текущее состояние всегда будет доступно в замыкании, что позволяет динамически изменять поведение замыкания на основе текущих значений переменных.

Ключевые различия:

- Состояние переменных: В обычном замыкании переменные могут использоваться без изменения, тогда как в автозамыкании переменные изменяются и сохраняют свое состояние между вызовами.

- Динамичность: Автозамыкания обеспечивают более динамичное поведение, позволяя замыканию всегда иметь доступ к актуальным значениям захваченных переменных.

Таким образом, хотя оба типа замыканий могут захватывать переменные из внешней области видимости, автозамыкания обеспечивают автоматическое обновление состояния этих переменных между вызовами, в то время как обычные замыкания могут использовать переменные без изменения их состояния.

**что такое функциональные типы?**

Функциональные типы в Kotlin — это способ описания функций в виде значений, которые могут быть переданы, возвращены и хранимы в переменных.

Функциональные типы определяются как (A, B) -> C, где:

(A, B) — типы параметров, которые принимает функция.

C — тип возвращаемого значения.

Примеры:

() -> Unit — функция без параметров, не возвращающая значение.

(Int, Int) -> String — функция, принимающая два Int и возвращающая String.

**Именованные параметры** — это механизм, позволяющий указать названия параметров в объявлениях функциональных типов. Это делает код более читаемым и понятным, особенно когда функции принимают несколько параметров одного типа или когда важно понимать, что именно делает каждый параметр.

При помощи круглых скобок функциональные типы можно объединять: (Int) -> ((Int) -> Unit).

**какие высокоуровневые функции знаете и чем они друг от друга отличаются?**

map

Применяет заданную функцию ко всем элементам коллекции и возвращает новую коллекцию, содержащую результаты.

filter

Отбирает элементы коллекции, которые соответствуют условию, заданному в лямбда-выражении, и возвращает новую коллекцию с подходящими элементами.

reduce

Сокращает коллекцию до одного значения, используя заданную функцию. Начальное значение не требуется, так как оно вычисляется на основе первых двух элементов.

fold

Похоже на `reduce`, но требует начальное значение и может работать с любым типом результата.

forEach

Выполняет заданное действие для каждого элемента коллекции. Не возвращает новую коллекцию.

flatMap

Применяет заданную функцию к каждому элементу коллекции, а затем объединяет результаты в одну коллекцию.

any и all

- `any` возвращает `true`, если хотя бы один элемент коллекции удовлетворяет условию.

- `all` возвращает `true`, если все элементы коллекции удовлетворяют условию.

Ключевые отличия:

- Возвращаемое значение: Некоторые функции, как `map`, `filter`, `flatMap`, возвращают новую коллекцию, тогда как `forEach` не возвращает никакого значения.

- Начальное значение: `reduce` не требует начального значения, в то время как `fold` требует его.

- Условия: `any` и `all` используются для проверки условий, а не для трансформации данных.

**какие области видимости?** \* локальная, внешняя, глобальная

**что такое захват значения?**

Захват значения — это концепция, связанная с замыканиями, которая описывает, как анонимные функции или лямбда-выражения могут "захватывать" переменные из своей внешней области видимости. Это позволяет функциям сохранять доступ к переменным, даже после того, как внешняя функция завершила своё выполнение.

Как это работает?

Когда вы создаете лямбда-выражение или анонимную функцию внутри другой функции, оно может использовать переменные, объявленные в этой внешней функции. Эти переменные становятся частью контекста замыкания и могут быть доступны внутри лямбды даже после выхода из внешней функции.

**чем полезны функции вашего порядка в создании гибких расширяемых программ?**

Функции высшего порядка играют важную роль в создании гибких и легко расширяемых программ благодаря тому, что позволяют передавать функции как параметры и возвращать их как результаты. Этот подход обеспечивает высокую степень абстракции и переиспользования кода, улучшая читабельность и уменьшая дублирование логики.

**Основные преимущества функций высшего порядка:**

1. **Абстракция над действиями**:
   * Функции высшего порядка позволяют абстрагировать конкретные действия, передавая их как параметры. Например, функции map, filter и reduce могут выполнять операции над списками независимо от типа данных, что делает их универсальными.
   * Это освобождает от необходимости прописывать логику для каждого типа задачи. Например, можно создать общую функцию для обработки данных, передавая ей конкретные действия в виде лямбд.
2. **Повышение переиспользуемости**:
   * Функции высшего порядка позволяют избежать дублирования кода. Вместо того, чтобы повторять одну и ту же логику в разных местах программы, можно создать функцию высшего порядка, которая будет принимать на вход разный функционал (например, лямбда-выражения) и выполнять его.
   * Например, функция для обработки ошибок может быть написана так, чтобы принимать лямбду с конкретной логикой, избегая повторения одной и той же структуры кода.
3. **Функциональное программирование и композиция**:
   * Используя функции высшего порядка, можно объединять небольшие функции для создания более сложных операций.
   * Композиция позволяет строить функциональные цепочки, где результат одной функции передается в другую. Это упрощает логику и делает код более декларативным, так как можно легко читать последовательность действий.
4. **Гибкость и настройка поведения**:
   * Функции высшего порядка позволяют пользователю задавать поведение и логику выполнения на лету, передавая функции как аргументы.
   * Это полезно для создания API, библиотек или фреймворков, где пользователь может контролировать обработку данных или логику, не изменяя основной код.

**как с помощью лямбда выражения реализовать функцию map? (как map заменить лямбда выражением)? например, \*2**

val numbers = listOf(1, 2, 3, 4, 5)

val doubledNumbers = mutableListOf<Int>()

numbers.forEach { doubledNumbers.add(it \* 2) }

println(doubledNumbers) // Вывод: [2, 4, 6, 8, 10]

**как создать замыкание, которое при каждом вызове увеличивает счетчик на один?**

fun createCounter(): () -> Int {

var count = 0 // Переменная count захватывается замыканием

return {

count++ // Увеличиваем count на 1 при каждом вызове

}

}