Лабораторна робота №4  
Цикли та рекурсії

* 1. Мета:

Отримання практичних навичок розробки циклічних програм та основ створення рекурсивних функцій.

* 1. Завдання:

1. Представити математичний запис фрагмента програми та обчислити значення змінної x після його виконання, де n – це номер варіанта. Фрагмент програми на рисунку:



Фрагмент програми

1. Скласти програму табулювання функції f(x) на відрізку [a; b] з кроком h Значення a, b, h вводити з клавіатури. Для другого варіанту завдання наступне:
2. Для заданих x, n, e, що вводяться з клавіатури:
   1. обчислити суму n доданків згідно варіанту
   2. обчислити суму тих доданків, які за абсолютним значенням більше e. (Завдання виконати для двох різних e, які відрізняються на порядок, для кожного випадку обчислити кількість доданків)
   3. Порівняти результати з "точним" значенням відповідної функції (сума визначає наближене значення) для x Є (-R,R)



* 1. Теоретична частина

У програмуванні значна кількість реальних завдань вимагає не лише застосування розгалужень, але й багаторазового повторення однотипних дій. Яскравим прикладом такої задачі є обчислення факторіала натурального числа n, що позначається як n! та визначається як добуток усіх чисел від 1 до n. За усталеною угодою, 0! = 1 та 1! = 1, тоді як факторіал від'ємного числа не визначений.

Одним зі способів реалізації багаторазових операцій є рекурсія. Факторіал можна визначити індуктивно як

У цьому індуктивному визначенні базовими випадками є 0! та 1!. Мовою Kotlin це реалізується за допомогою функції, яка викликає саму себе, але з меншим значенням аргументу. Наприклад, функція factorial(n) буде викликати factorial(n-1), поки умова if (n < 2) не стане істинною, що забезпечує завершення обчислень та запобігає нескінченній рекурсії.

Альтернативою рекурсивному підходу є ітеративне визначення, яке реалізується за допомогою циклів. У Kotlin найпоширенішим є цикл for, що ідеально підходить для таких завдань, як обчислення факторіала шляхом послідовного множення чисел від 1 до n. Ось конструкція:

for (i in 1..n) { ... }

Вона означає "для всіх i в інтервалі від 1 до n виконати...". У цій конструкції i є параметром циклу, який на кожній ітерації набуває нового значення від початкового до кінцевого. Тіло циклу розміщується всередині фігурних дужок і виконується n разів. Для зберігання проміжного результату, як, наприклад, у функції обчислення факторіала, необхідно використовувати мутуючу змінну, яка оголошується за допомогою ключового слова var (від "variable"). Це відрізняється від val (від "value"), що використовується для оголошення незмінних значень. Оператор result = result \* i виконує присвоювання мутуючій змінній нового значення, що є добутком її поточного значення та значення параметра циклу. Остаточний результат обчислення повертається оператором return result. Цикл for не обмежується лише перебором чисел в інтервалі min..max з кроком 1; він також може використовуватися для перебору чисел у заданій прогресії. Наприклад, 10 downTo 1 створює спадну прогресію з кроком 1, а 1..99 step 2 — висхідну прогресію з кроком 2.

У Kotlin існують модифікуючі оператори, які є скороченими формами запису для операцій присвоєння. Наприклад, оператор result = result \* i може бути замінений на result \*= i, що виконує домноження поточного значення змінної result на i. Аналогічно працюють оператори +=, -=, /=, %=, а також ++ (інкремент) та -- (декремент), які збільшують або зменшують значення змінної на одиницю.

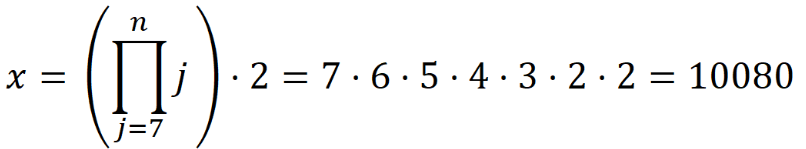
Для перевірки умов у циклічних програмах часто застосовується послідовна перевірка умов, як, наприклад, при визначенні, чи є натуральне число простим. Просте число ділиться без остачі тільки на одиницю і на себе. Прямолінійна перевірка передбачає ділення заданого числа n послідовно на числа в інтервалі від 2 до n-1. Для перевірки ділення націло використовується оператор залишка від ділення %. Якщо хоча б один раз число n ділиться без остачі, це означає, що воно не є простим, і в цьому випадку функція може достроково завершитися, повернувши false. Важливо розуміти, що для доведення простоти числа необхідно перевірити всі можливі дільники, і лише після завершення циклу, якщо жодного дільника не знайдено, можна повернути true. Для оптимізації перевірки простоти, достатньо обмежити інтервал перевірки дільників від 2 до квадратного кореня з n, оскільки якщо число n має дільник більший за √n, то воно обов'язково матиме і менший дільник. При цьому для обчислення квадратного кореня з цілого числа n необхідно спочатку перетворити його до типу Double за допомогою функції n.toDouble(), а потім результат обчислення кореня перетворити назад до цілого за допомогою .toInt(). Така форма запису, де аргумент функції стоїть перед її іменем через крапку, називається викликом функції через отримувача (receiver).

Управління виконанням циклів може здійснюватися за допомогою операторів break та continue. Оператор break дозволяє достроково перервати виконання циклу, переходячи до наступного оператора за межами циклу. Наприклад, у функції для визначення досконалого числа (яке дорівнює сумі своїх дільників, крім себе самого), якщо проміжна сума дільників вже перевищила саме число, подальші обчислення безглузді, і цикл можна перервати за допомогою break. Оператор continue перериває лише поточну ітерацію циклу, пропускаючи решту її тіла, та переходить до початку наступної ітерації. Це може бути використано, наприклад, щоб пропустити додавання чисел, які не є дільниками, до суми дільників.

Крім циклу for, Kotlin також пропонує цикли while та do..while, які застосовуються, коли кількість ітерацій невідома заздалегідь. Наприклад, для підрахунку кількості входжень певної цифри в десятковий запис числа, необхідно послідовно обробляти цифри числа, що вимагає циклу з невизначеною кількістю ітерацій. Щоб отримати молодшу цифру числа, використовують залишок від ділення на 10 (% 10), а для відкидання молодшої цифри — цілочисельне ділення на 10 (/= 10). Цикл while перевіряє умову, зазначену в дужках, перед кожною новою ітерацією (в тому числі перед першою); якщо умова хибна, цикл не виконується. Таким чином, цикл while може не виконатися жодного разу. Натомість цикл do..while перевіряє умову після кожної ітерації. Це означає, що тіло циклу do..while завжди виконується хоча б один раз. Через це while називається циклом з передумовою, а do..while — циклом з післяумовою. У конкретних випадках, таких як підрахунок цифр у числі 0, do..while може дати більш коректний результат, оскільки гарантує виконання хоча б однієї ітерації. Хоча будь-яке завдання можна вирішити за допомогою обох циклів, while на практиці зустрічається значно частіше. Варто зазначити, що для завдання з підрахунком цифр також можливе рекурсивне рішення, яке часто є більш коротким та витонченим у порівнянні з ітеративним.

* 1. Практична частина

1. На рисунку зображено математичний запис програми, що була у завданні і розрахунки для n = 2:



Математичний запис і розрахунки для 2 варіанту

1. Функції для вводу даних і табуляції функції:

fun var2z2() {

val (a,b,h) = readInputData()

tabulation(a,b,h)

}

fun readInputData(): Triple<Double, Double, Double> {

print("Введіть значення a: ")

val a = readln().toDouble()

print("Введіть значення b: ")

val b = readln().toDouble()

print("Введіть значення h: ")

val h = readln().toDouble()

return Triple(a,b,h)

}

fun tabulation(a: Double, b:Double, h:Double) {

if (h <= 0 || a > b) {

println("Некоректні значення: переконайтеся, що h > 0 і a <= b")

return

}

println("Таблиця значень функції f(x) = cbrt(x):")

println("x\t\tf(x)")

var x = a

while (x <= b) {

val y = cbrt(x)

println(String.format("%.2f\t%.2f", x, y))

x += h

}

}

1. Нижче зображений код програми, що реалізує функціонал для обчислення та аналізу числових рядів за заданими параметрами. Зокрема, програма обчислює суму n доданків ряду, де n є вхідним параметром, який вводиться користувачем. Додатково програма визначає суму доданків, абсолютне значення яких перевищує задану величину e. Цей розрахунок виконується для двох різних значень e, що відрізняються на порядок, при цьому для кожного випадку фіксується кількість доданків, що задовольняють цій умові. Нарешті, програма виконує порівняння отриманих наближених сум з "точним" значенням відповідної аналітичної функції для заданого x з інтервалу збіжності, що дозволяє оцінити точність виконаних обчислень.

fun var2z3() {

try {

print("Введіть значення x: ")

val x = readln().toDouble()

print("Введіть кількість доданків (n): ")

val n = readln().toInt()

print("Введіть значення e (точність): ")

val e1 = readln().toDouble()

val sumN = f(x, n)

println("Сума $n доданків: $sumN")

val (sumE1, countE1) = f(x, n, e1)

println("Сума доданків для e=$e1: $sumE1, кількість доданків: $countE1")

val e2 = e1 / 10

val (sumE2, countE2) = f(x, n, e2)

println("Сума доданків для e=$e2: $sumE2, кількість доданків: $countE2")

val exactValue = f(x)

println("Точне значення e^(-x^2): $exactValue")

println("Похибка для n доданків: ${abs(exactValue - sumN)}")

println("Похибка для e=$e1: ${abs(exactValue - sumE1)}")

println("Похибка для e=$e2: ${abs(exactValue - sumE2)}")

}

catch (e: Exception) {

println(e.message)

}

}

fun f(x:Double):Double {

return exp(-x.pow(2))

}

fun f(x: Double, n: Int): Double {

var sum = 0.0

for (i in 0 until n) {

val term = (-1.0).pow(i) \* x.pow(2 \* i) / factorial(i)

sum += term

}

return sum

}

fun f(x: Double, n: Int, e: Double): Pair<Double, Int> {

var sum = 0.0

var count = 0

for (i in 0 until n) {

val term = (-1.0).pow(i) \* x.pow(2 \* i) / factorial(i)

if (abs(term) > e) {

sum += term

count++

}

}

return Pair(sum, count)

}

fun factorial(n: Int): Double {

return if (n == 0 || n == 1) 1.0 else (2..n).fold(1.0) { res, i -> res \* i }

}

* 1. Висновки

У ході лабораторної роботи отримано практичні навички розробки циклічних програм та основи створення рекурсивних функцій.