Laboratorium 5.

1. **Funkcje w Scali:** `trait FunctionN`

    1. W konsoli (REPL) definiujemy klasę `NumOps` :

    ```
    class NumOps {
      def succ(a: Int) = a + 1
      def pred(a: Int) = a - 1
    }
    ```

    2. Wpisujemy (w konsoli) kolejno poniższe linie:

    ```
    val numOps = new NumOps
    val after5 = numOps.succ(5)
    val before5 = numOps.pred(5)
    ```

    3. Wpisujemy (w konsoli) definicję klasy `Succ` :

    ```
    class Succ extends Function1[Int, Int] {
      override def apply(a: Int) = a + 1
    }
    ```

    4. Wpisujemy (w konsoli) kolejno poniższe linie:

    ```
    val succ = new Succ
    val after5 = succ.apply(5)
    val after5 = succ(5)
    ```

    Zadanie: zdefiniować klasę `Pred` odpowiadającą metodzie `pred` klasy `NumOps`

    Zadanie: w klasie `NumOps` dodać metodę (zwracającą maksymalną z trzech podanych liczb)

    ```
    def maxFrom3(d1: Double, d2: Double, d3: Double): Double = {
      /*...*/
    }
    ```

    a następnie zdefiniować klasę `MaxFrom3` z odpowiednią implementacją metody `apply` ; napisać kilka testów sprawdzających poprawność napisanego kodu

2. **Literały funkcyjne, obiekty funkcyjne** (*function literals*, *function values*)

    1. W konsoli (REPL) wpisujemy kolejno:

    ```
    val succ: (a: Int) = a + 1
    val succ = (a: Int) = a + 1
    val succ => (a: Int) = a + 1
    val succ: (a: Int) => Int = (a: Int) = a + 1
    val succ: (a: Int) => Int = (a: Int) => a + 1
    ```

    i analizujemy wyniki (komunikaty konsoli)

    2. W konsoli (REPL) definiujemy funkcję `succ` (obiekt funkcyjny/function value):

    ```
    val succ: (Int) => Int = (a: Int) => a + 1

    // uwaga: opuszczenie nawiasów możliwe tylko dla funkcji jednopa
    rametrowych
    val succ: Int => Int = (a: Int) => a + 1
    val succ = (a: Int) => a + 1
    ```

    Zadanie: (na podstawie powyższego kodu) zdefiniować funkcje: `pred` i `maxFrom3`

3. **Funkcje wyższego rzędu: funkcje jako argumenty**

    1. W pliku `lab531.scala` wpisujemy:

    ```
    object Appl531 {
      def genSumArray(elems: Array[Int], f: (Int) => Int) = {
        var sum = 0
        for (e <- elems) sum += f(e)
        sum
      }

      def sum(elems: Array[Int]) = genSumArray(elems, (e: Int) => e)

      def main(args: Array[String]) {
        val a = Array(1,2,3,4,-5)
        println("sum(a) = " + sum(a))
      }
    }
    ```

    2. Kompilujemy kod i uruchamiamy `Appl531`

    Zadanie: dodać metody obliczające sumy: elementów podniesionych do kwadratu (`sumSqr`), do potęgi 3 (`sumCube`) i wartości bezwzględnych elementów (`sumAbs`):

    ```
    def sumSqr(elems: Array[Int]) = genSumArray(elems, ___)
    ```

```
def sumCube(elems: Array[Int]) = genSumArray(elems, ___)
def sumAbs(elems: Array[Int]) = genSumArray(elems, ___)
```

napisać testy sprawdzające poprawność napisanego kodu

Zadanie (opcjonalne): w metodach: `sum`, `sumSqr`, `subCube`, `subAbs` dodać parametr `log: (String) => Unit`

```
def sum(elems: Array[Int], log: (String) => Unit)
def sumCube(elems: Array[Int], log: (String) => Unit)
//...
```

umożliwiający logowanie wykonanych operacji; dodać dwie funkcje:

```
val logToStdOut = (msg: String) => { println(msg) }
val noLogo = (msg: String) => {} // :)
```

i przetestować działanie napisanego kodu

4. **Funkcje wyższego rzędu: funkcje jako wyniki**

    1. W konsoli (REPL) dodajemy definicję metody `df` (obliczającej w sposób przybliżony pochodną funkcji):

    ```
    def df(f: Double => Double) = {
      (x: Double) => {
        val h = 1e-3
        (f(x + h) - f(x)) / h
      }
    }
    ```

    2. Dodajemy (w konsoli) definicje funkcji:

    ```
    val sqr = (x: Double) => x * x
    val dSqrExact = (x: Double) => 2 * x
    ```

    3. Wpisujemy (w konsoli) (wyznaczamy przybliżenie pochodnej):

    ```
    val dSqrAppx = df(sqr)
    ```

    4. Wpisujemy w (w konsoli):

    ```
    for (x <- 0.0 to 2.0 by 0.2
         d = math.abs(dSqrAppx(x) - dSqrExact(x)))
      println(f"x = $x%2.1f, delta = $d")
    ```

    i analizujemy wyniki; jaka jest maksymalna wartość błędu?

    Zadanie (opcjonalne): znaleźć wartość `h`, dla której maksymalny błąd bezwzględny przybliżenia pochodnej w przedziale `[0,2]` jest mniejszy niż `1e-7`

    Zadanie (opcjonalne): sprawdzić dokładność następującego przybliżenia pochodnej w punkcje `x`, dla zadanego `h`:

    ```
    (f(x + h) - f(x - h)) / (2 * h)
    ```

    dla jakiej wartości “h” maksymalny błąd bezwzględny przybliżenia pochodnej w przedziale `[0,2]` jest mniejszy niż `1e-7` ?

    Zadanie (opcjonalne): napisać funkcję `d2f` obliczającą drugą pochodną funkcji `f` (wskazówka: druga pochodna to pochodna pierwszej pochodnej); przetestować jej działanie na wybranych przykładach funkcji

5. **Porównanie typów: `() => T` vs. `Unit => T` vs. `: => T`**

    1. W konsoli (REPL) wpisujemy:

    ```
    def f1(f: () => Int) = println("The number is: " + f())
    def f2(f: Unit => Int) = println("The number is: " + f(()))
    def f3(f: => Int) = println("The number is: " + f)
    ```

    2. Wpisujemy kolejno następujące linie i analizujemy wyniki:

    ```
    f1(1)
    f1({val x = 1; x})
    f1(() => 1)

    f2(1)
    f2({val x = 1; x})
    f2(() => 1)
    f2(_:Unit => 1)
    f2((_:Unit) => 1)

    f3(1)
    f3({val x = 1; x})
    f3(() => 1)
    f3(_:Unit => 1)
    f3((_:Unit) => 1)
    ```

6. **Porównanie: *call by value* vs. *call by name***

    1. W konsoli wpisujemy:

```
def f(argByVal: Long, argByName: => Long) {
  Thread.sleep(1000)
  println("In function: " + argByVal + ", " + argByName)
}
```

2. Ustawiamy (w konsoli) tryb `:paste` i wpisujemy:

```
println("Before: " + System.currentTimeMillis)
f(System.currentTimeMillis(), System.currentTimeMillis())
println("After: " + System.currentTimeMillis)
```

3. Wpisujemy nową definicję metody `f` - jedyna zmiana to usunięcie strzałki po `argByName`, wklejamy (w trybie `:paste`) kod z poprzedniego kroku i analizujemy wynik

4. W konsoli wpisujemy kolejno poniższe linie:

```
def printNTimes(toPrint: Int, n: Int) =
  for(i <- 1 to n) println(toPrint)

printNTimes(scala.util.Random.nextInt, 3)
```

```
def printNTimes(toPrint: => Int, n: Int) =
  for(i <- 1 to n) println(toPrint)

printNTimes(scala.util.Random.nextInt, 3)
```

i analizujemy wyniki.

7. **Literały funkcyjne: udogodnienia w składni**

1. W konsoli wpisujemy:

```
def binaryOp(a: Int, b: Int, f: (Int, Int) => Int) = f(a, b)
```

2. Wpisujemy kolejno poniższe linie i analizujemy wyniki:

```
val aPlusB = binaryOp(6, 2, (a: Int, b: Int) => a + b)
val aPlusB = binaryOp(6, 2, (a, b) => a + b)
val aPlusB = binaryOp(6, 2, (_:Int) + (_:Int))
val aPlusB = binaryOp(6, 2, _ + _)
val aMinusB = binaryOp(6, 2, _ - _)
val aTimesB = binaryOp(6, 2, _ * _)
val aDivB = binaryOp(6, 2, _ / _)
```

Zadanie (opcjonalne): w pliku `lab531.scala` zmodyfikować definicje funkcji (sumujących) z wykorzystaniem uproszonych form literałów funkcyjnych (np. zamiast `(e: Int) => e) - e => e`)

8. **Domknięcie funkcji (*closure*), zmienne związane (*bound*) i "swobodne/niezwiązane" (*free*)**

1. W konsoli wpisujemy kolejno następujące linie i analizujemy wyniki:

```
var freeVar = 0
println("[1]: freeVar = " + freeVar)
val f1 = (boundVar: Int) => {
  println("[Inside f1]: freeVar = " +
          freeVar + ", boundVar = " + boundVar)
  freeVar += 1
}
println("[2]: freeVar = " + freeVar)

f1(5)
println("[3]: freeVar = " + freeVar)

freeVar += 10
println("[4]: freeVar = " + freeVar)

f1(10)
println("[5]: freeVar = " + freeVar)
```

2. W konsoli wpisujemy:

```
def df(f: Double => Double, h: Double): Double => Double = {
  (x: Double) => (f(x + h) - f(x - h)) / (2 * h)
}

val dSqrAppx = df((x: Double) => x * x, h = 1e-6)
val dSqrExact = (x: Double) => 2 * x
```

3. Wpisujemy (w konsoli) następującą linię i analizujemy wynik:

```
for (x <- 0.0 to 2.0 by 0.2
     d = math.abs(dSqrExact(x) - dSqrAppx(x)))
  println(f"x = $x%2.1f, delta = $d")
```

Zadanie (opcjonalne): napisać program znajdujący wartość taką `h`, dla której maksymalny błąd przybliżenia pochodnej w przedziale `[0,2]` jest najmniejszy

Zadanie (opcjonalne): zmienić poniższą funkcję w domknięcie (closure):

```
def sumArrayRec(elems: Array[Int]) = {
  @annotation.tailrec
  def goFrom(i: Int,
             size: Int,
             elms: Array[Int],
             acc: Int): Int = {
    if (i < size) goFrom(i + 1, size, elms, acc + elms(i))
    else acc
  }
  goFrom(0, elems.size, elms, 0)
}
```

uwaga: sygnatura `goFrom` powinna być następująca:

```
@annotation.tailrec
def goFrom(i: Int, acc: Int): Int
```

9. **Funkcje częściowe** (*partial functions*)

    1. W konsoli wpisujemy:

    ```
    val identForInts = new PartialFunction[Double, Double] {
      def apply(x: Double) = if (x == x.toInt) x
      def isDefinedAt(x: Double) = (x == x.toInt)
    }
    ```

    i analizujemy komunikat błędu.

    2. W konsoli wpisujemy:

    ```
    val identForInts = new PartialFunction[Double, Double] {
      def apply(x: Double) = if (x == x.toInt) x else Double.NaN
      def isDefinedAt(x: Double) = (x == x.toInt)
    }
    ```

    Dla jakich wartości parametru `x` funkcja `identForInts` jest określona?

    3. Wpisujemy (w konsoli) kolejno poniższe linie i analizujemy wyniki:

    ```
    identForInts.isDefinedAt(1.0)
    identForInts.isDefinedAt(1.3)
    identForInts.apply(1.0)
    identForInts(1.0)
    identForInts(1.5)
    identForInts.applyOrElse(1.0, (_:Double) => Double.NaN)
    identForInts.applyOrElse(1.5, (_:Double) => Double.NaN)
    ```

    4. Wpisujemy (w konsoli):

    ```
    val identForInts: PartialFunction[Double, Double] = {
      case x if x == x.toInt => x
    }
    ```

    5. Powtarzamy punkt 3. (wpisujemy w konsoli) kolejno linie i analizujemy wyniki)

10. **Partially applied functions**

    1. Definiujemy w konsoli metodę `sum` :

    ```
    def sum(a: Int, b: Int, c: Int) = a + b + c
    ```

    2. Wpisujemy (w konsoli) kolejno poniższe linie i analizujemy wyniki inferencji typu

    ```
    sum(1, 2, 3)
    val sumABC = sum _
    val aPlus = sumABC(1, _: Int, _:Int)
    val aPlusBPlus = aPlus(2, _:Int)
    val aPlusBPlusC = aPlusBPlus(3)
    ```

11. **Składanie funkcji**

    1. Definiujemy (w konsoli) poniższe metody:

    ```
    def f(x: Int) = x + 3
    def g(x: Int) = x * x
    ```

    2. Wpisujemy kolejno poniższe linie i analizujemy wyniki:

    ```
    val f_o_g = f _ compose g _
    f_o_g(5)

    val fAndThenG = f _ andThen g _
    fAndThenG(5)
    ```

    Co oznacza `f _` ?

12. **Rozwijanie funkcji** (*currying*)

    1. Definiujemy w konsoli metodę `sumABC` :

    ```
    def sumABC(a: Int, b: Int, c: Int) = a + b + c
    ```

2. Wpisujemy (w konsoli) kolejno poniższe linie i analizujemy wyniki

```
sumABC(1,2,3)
val currSumABC = (sumABC _).curried
currSumABC(1,2,3)
currSumABC(1)(2)(3)
currSumABC{1}{2}{3}
currSumABC{1}(2){3}
```

3. Definiujemy (w konsoli) metodę `mCurrSumABC`

```
def mCurrSumABC(a: Int)(b: Int)(c: Int) = a + b + c
```

   i analizujemy wynik inferencji typu.

4. Wywołujemy metodę `mCurrSumABC` z argumentami `1` , `2` i `3`

5. Definiujemy w konsoli funkcję `fCurrSumABC` :

```
val fCurrSumABC = (a: Int) => (b: Int) => (c: Int) => a + b + c
```

6. Wpisujemy jeszcze raz powyższą definicję, tym razem z pełną specyfikacją typu:

```
val fCurrSumABC: Int => (Int => (Int => Int)) =
  (a: Int) => (b: Int) => (c: Int) => a + b + c
```

7. Wpisujemy kolejno poniższe linie i analizujemy komunikaty błędów:

```
val fCurrSumABC =
  (a: Int) => ((b: Int) => (c: Int)) => a + b + c
```

```
val fCurrSumABC =
 ((a: Int) => ((b: Int) => (c: Int))) => a + b + c
```

8. W konsoli wpisujemy poniższą definicję:

```
def f(ints: Int*)(doubles: Double*)(strings: String*) = {
  for (i <- ints) print(i + " ")
  println()
  for (d <- doubles) print(d + " ")
  println()
  for (s <- strings) print(s + " ")
}
```

9. Wywołujemy metodę `f` :

```
f(1, 2, 3, 4)(6.1, 7.1, 8.1, 9.5)("abc", "def", "ghi")
```

10. W konsoli wpisujemy poniższą definicję:

```
def catchingAllExceptions(tryBlock: => Unit) = {
  try {
    tryBlock
  } catch {
    case _: Throwable =>
  }
}
```

11. Wywołujemy (w konsol, tryb `:pastei`) powyższą metodę

```
catchingAllExceptions {
  val x = 4
  println("x = " + x)
  val y = 0
  println("y = " + y)
  val z = x / y
  println("z = " + z)
}
```

   i analizujemy wynik.

12. (*Opcjonalne*) Wpisujemy w konsoli następujący kod:

```
import scala.language.implicitConversions
implicit def defaultExceptionHdlr(ex: Throwable) =
  println(ex.getMessage)

def catchingAllExceptions(tryBlock: => Unit)
    (implicit defaultExHdlr: (Throwable) => Unit) = {
  try {
    tryBlock
  } catch {
    case ex: Throwable => defaultExHdlr(ex)
  }
}

catchingAllExceptions {
  val x = 4
  println("x = " + x)
  val y = 0
  println("y = " + y)
```

```
  val z = x / y
  println("z = " + z)
}
```

i analizujemy wynik.