Laboratorium 4.

1. **Inferencja typu wyniku metody rekurencyjnej**

    1. W pliku `lab41.scala` wpisujemy:

```
object Appl41 {
  def factorial(n: Int) = {
    assert(n >= 0)
    if (n == 0 || n == 1) 1
    else n * factorial(n - 1)
  }

  def fibb(n: Int) = {
    assert(n >= 0)
    if (n == 0 || n == 1) n
    else fibb(n - 2) + fibb(n - 1)
  }

  def main(args: Array[String]) {
    println("5! = " + factorial(5))
    println("Fibb(10) = " + fibb(10))
  }
}
```

    2. Uruchamiamy kompilację i analizujemy jej wyniki

    3. Dodajemy typy wyników metod `factorial` i `fibb`
    4. Kompilujemy ponownie i uruchamiamy `Appl41`

2. **Elementarne algorytmy "tablicowe": *iteracja* vs. *rekursja***
   Uwaga: stosujemy technikę programistyczną TDD (*Test Driven Development*)

    1. W pliku `lab42.scala` wpisujemy:

```
object Appl42 {
  def checkPredicate(pred: Boolean,
                     predAsString: String,
                     prefix: String = "Checking if ") {
    if (pred) println(prefix + predAsString + ": OK")
    else println(prefix + predAsString + ": Fail")
  }
  def sumArrayIter(elems: Array[Int]) = {
    var sum = 0
    for (i <- elems) sum += i
    sum
  }
  def main(args: Array[String]) {
    val a1To5 = (1 to 5).toArray
    println("Testing with a1To5 = " +
            a1To5.mkString("Array(", ", ", ") ..."))
    val expectResult = 15
    checkPredicate(sumArrayIter(a1To5) == expectResult,
                   "sumArrayIter(0, a1To5) == " + expectResult)
    checkPredicate(sumArrayRec1(0, a1To5) == expectResult,
                   "sumArrayRec1(0, a1To5) == " + expectResult)
    checkPredicate(sumArrayRec2(a1To5) == expectResult,
                   "sumArrayRec2(a1To5) == " + expectResult)
  }
}
```

    2. Kompilujemy kod i analizujemy komunikaty kompilatora

    3. Dodajemy w obiekcie `Appl42` (np. nad metodą `main`) poniższe definicje:

```
def sumArrayRec1(i: Int, elems: Array[Int]): Int = 0
def sumArrayRec2(elems: Array[Int]) = 0
```

    4. Kompilujemy kod, uruchamiamy `Appl42` (`scala Appl42`) i analizujemy wyniki

    Zadanie: uzupełnić definicje metod `sumArrayRec1` i `sumArrayRec2` (na podstawie poniższych wskazówek):

```
def sumArrayRec1(i: Int, elems: Array[Int]): Int = {
  if (i < elems.size) elems(i) + sumArrayRec1(i + 1, elems)
  else ___
}

def sumArrayRec2(elems: Array[Int]) = {
  val size = elems.size
  def goFrom(i: Int): Int = {
    if (i < size) ___ + goFrom(___)
    else ___
  }
  goFrom(___)
}
```

    tak, aby wszystkie testy "przechodziły pozytywnie" (czyli miały status `OK`)

    Zadanie: dodać zestawy testów oraz implementacje metod:

    - `sumSqrArrayIter` + `sumSqrArrayRec1` + `sumSqrArrayRec2` (suma kwadratów elementów tablicy)|

- (*opcjonalne*) `prodArrayIter` + `prodArrayRec1` + `prodArrayRec2` (iloczyn elementów tablicy)
- (*opcjonalne*) `sumAbsArrayIter` + `sumAbsArrayRec1` + `sumAbsArrayRec2` (suma wartości bezwzględnych elementów tablicy)

Jak można uogólnić te rozwiązania (szczególne)?

Zadanie (opcjonalne): napisać odpowiednie metody dla tablicy dwuwymiarowej (`m x n`); zacząć od iteracji, potem rekursja

5. W pliku `lab43.scala` wpisujemy:

```
object Appl43 {
  def checkPredicate( /* ... */ ) { /* ... */ }
  def maxIter(elems: Array[Int]) = {
    var max = ___
    for (i <- elems) if (i > max) max = i
    max
  }
  def maxRec1(i: Int, maxEl: Int, elems: Array[Int]): Int = {
    if (i < elems.size) maxRec1(___, math.max(___, ___), elems)
    else ___
  }
  def maxRec2(elems: Array[Int]) = {
    def goFrom(i: Int, maxEl: Int, size: Int): Int = {
      if (i < size) goFrom(___, math.max(___, ___), size)
      else ___
    }
    goFrom(0, ___, ___)
  }
  def main(args: Array[String]) {
    val arr1 = Array(3, 12, 43, 4, 10)
    println("Testing with arr1 = " +
            arr1.mkString("Array(", ", ", ") ..."))
    val expectResult = 43
    checkPredicate(maxIter(arr1) == expectResult,
                   "maxIter(arr1) == " + expectResult)
    checkPredicate(maxRec1(0, ___, arr1) == expectResult,
                   "maxRec1(0, ___, arr1) == " + expectResult)
    checkPredicate(maxRec2(arr1) == expectResult,
                   "maxRec2(arr1) == " + expectResult)
  }
}
```

Zadanie: uzupełnić brakujące miejsca w kodzie, tak, aby wszystkie testy "przechodziły pozytywnie" (status `OK`); metody mają zwracać element o największej wartości

Zadanie (opcjonalne): zmodyfikować kod, aby metody zwracały najmniejszy element; jak można te rozwiązania uogólnić?

Zadanie (opcjonalne): napisać metody (odpowiadające powyższemu zestawowi) znajdujące DWA sąsiednie elementy, których suma jest największa

Zadanie (opcjonalne): napisać odpowiednie metody dla tablicy dwuwymiarowej (`m x n`); zacząć od iteracji, potem rekursja

3. **Rekursja - przepełnienie stosu**

1. W pliku `lab44.scala` wpisujemy:

```
object Appl44 {
  def sumArrayRec2(elems: Array[Int]) = {
    def goFrom(i: Int, size: Int, elms: Array[Int]): Int = {
      if (i < size) elms(i) + goFrom(i + 1, size, elms)
      else 0
    }
    goFrom(0, elems.size, elems)
  }

  def main(args: Array[String]) {
    println("sumArrayRec2 = " +
            sumArrayRec2((0 until 50000).toArray))
  }
}
```

2. Kompilujemy kod, uruchamiamy `Appl44` i analizujemy wyniki
Zadanie: wyznaczyć (eksperymentalnie) maksymalny rozmiar tablicy, który nie powoduje przepełnienia stosu

3. Modyfikujemy kod obiektu `Appl44` : dodajemy metodę `recurseTest` i zmieniamy metodę `main` :

```
def recurseTest(i: Int, j: Int, arr: Array[Int]): Int = {
  try {
    recurseTest(i + 1, j + 1, arr)
  } catch { case e: java.lang.StackOverflowError =>
    println("Recursion depth on this system is " + i + ".")
    i
  }
}
```

```
def main(args: Array[String]) {
  val recDepth = recurseTest(0, 0, Array(1))
  println("sumArrayRec2 = " +
```

```
          sumArrayRec2((0 until recDepth).toArray))
}
```

4. Kompilujemy kod, uruchamiamy `Appl44` i analizujemy wyniki

   Zadanie (opcjonalne): sprawdzić, czy otrzymany w wyniku powyższego testu rozmiar tablicy jest rzeczywiście maksymalny; czy otrzymywane w kolejnych urychomieniach wyniki są takie same?

   Zadanie (opcjonalne): przeanalizować kod bajtowy metod `sumArrayRec2` i `goFrom`

5. (*ćwiczenie opcjonalne*) Dodajemy w definicji obiektu kolejną metodę (implementacja funkcji Ackermana):

```
def ackerFun(m: Int, n: Int): Int = {
  assert(m >= 0 && n >= 0)
  if (m == 0) n + 1
  else if (n == 0) ackerFun(m - 1, 1)
  else ackerFun(m - 1, ackerFun(m, n - 1))
}
```

   Zadanie (opcjonalne): wyznaczyć maksymalne wartości `m` i `n`, dla których można obliczyć `ackerFun`

   Zadanie (opcjonalne): naszkicować zależności: `z1(y) = ackerFun(3, y) i z2(x) = ackerFun(x, 3)`

4. **Rekursja "ogonowa"/końcowa, adnotacja *@tailrec***

   1. W pliku `lab45.scala` wpisujemy:

```
object Appl45 {
  import scala.annotation.tailrec

  def sumArrayRec2(elems: Array[Int]) = {
    def goFrom(i: Int, size: Int, elms: Array[Int]): Int = {
      if (i < size) elms(i) + goFrom(i + 1, size, elms)
      else 0
    }
    goFrom(0, elems.size, elems)
  }

  def sumArrayRec3(elems: Array[Int]) = {
    @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, elems, 0)
  }

  def main(args: Array[String]) {
    println("sumArrayRec3 = " +
            sumArrayRec3((0 until 30000).toArray))
  }
}
```

   2. Kompilujemy kod i uruchamiamy `Appl45`. Dlaczego wywołanie (rekurencyjnej) metody `sumArrayRec3` nie powoduje przepełnienia stosu przy przetwarzaniu tak dużej tablicy?

   3. Dodajemy adnotację `@tailrec` w definicji metody `sumArrayRec2`

   4. Uruchamiamy kompilację i analizujemy jej wyniki. Czym różnią się metody `sumArrayRec2` i `sumArrayRec3`? Do czego służy parametr `acc`?

      Zadanie: w pliku `lab42.scala` dodać metodę `sumSqrArrayRec3` - z rekursją ogonową (dla potwierdzenia poprawności dodać `@tailrec`)

      Zadanie (opcjonalne): w pliku `lab41.scala` dodać nowe wersje `factorial` i `fibb` - z rekursją ogonową (dla potwierdzenia poprawności dodać `@tailrec`)

      Zadanie (opcjonalne): dodać odpowiednie metody (z rekursją ogonową) dla `prodArrayRec3` i `sumAbsArrayRec3` (dla potwierdzenia poprawności dodać `@tailrec`)

      `Zadanie (opcjonalne)`: przeanalizować kod bajtowy obu metod `goFrom` (wewnętrznych dla odpowiednio `sumArrayRec2` i `sumArrayRec3`)

5. **Usprawnianie procesów rekurencyjnych: "*memoization*" i wzorzec "*trampolina*"** (ćwiczenie opcjonalne)

   1. W pliku lab461.scala wpisujemy:

```
object fibbCalc {
  private val cache = collection.mutable.Map[Long, Long]()
  def fibb(n: Long): Long = {
    assert(n >= 0)
    if (n == 0 || n == 1) n
    else cache.getOrElseUpdate(n, fibb(n - 2) + fibb(n - 1))
  }
```

```
}

object Appl461 {
  def fibb(n: Long): Long = {
    assert(n >= 0)
    if (n == 0 || n == 1) n
    else fibb(n - 2) + fibb(n - 1)
  }

  def main(args: Array[String]) {
    val start1 = System.nanoTime
    val fibb50 = fibb(50)
    val end1 = System.nanoTime
    println("fibb(50) = " + fibb50 + ", elapsed: " +
            (end1 - start1) / 1e6 + " ms")

    val start2 = System.nanoTime
    val augmFibb50 = fibbCalc.fibb(50)
    val end2 = System.nanoTime
    println("Augmented fibb(50) = " + augmFibb50 +
            ", elapsed: " + (end2 - start2) / 1e6 + " ms")
  }
}
```

2. Kompilujemy kod, uruchamiamy `Appl461` i analizujemy wyniki (skąd wynika tak duża różnica w czasie wykonania tych metod?)

3. W pliku `lab462.scala` wpisujemy:

```
object Appl462 {
  def odd(n: Int): Boolean = {
    if (n == 0) false
    else even(n - 1) // indirect recursion
  }

  def even(n: Int): Boolean = {
    if (n == 0) true
    else odd(n - 1) // indirect recursion
  }

  def main(args: Array[String]) {
    val n = 999
    val str = if (even(n)) " is even." else " is odd."
    println(n.toString + str)
  }
}
```

4. Kompilujemy i uruchamiamy

   Zadanie: wyznaczyć maksymalną wartość `n`, która nie powoduje przepełnienia stosu

5. W pliku `lab463.scala` wpisujemy:

```
object Appl463 {
  import scala.util.control.TailCalls._

  private def isEven(n: Int): TailRec[Boolean] =
    if (n == 0) done(true) else tailcall(isOdd(n - 1))

  private def isOdd(n: Int): TailRec[Boolean] =
    if (n == 0) done(false) else tailcall(isEven(n - 1))

  def main(args: Array[String]) {
    val n = 99999
    val str = if (isEven(n).result) " is even." else " is odd."
    println(n.toString + str)
}
}
```

6. Kompilujemy i uruchamiamy `Appl463`. Dlaczego przy tak dużym `n` nie wystąpiło przepełnienie stosu?

6. **Odwzorowanie `switch - case` (z Javy) w Scali** (wariant z typem wyliczeniowym)

   1. W pliku JavaAppl.java wpisujemy:

```
class JavaAppl {
  enum WeekDay {
    Mon, Tue, Wed, Thu, Fri, Sat, Sun
  }
  private static void printDayName(WeekDay day) {
    switch (day) {
      case Mon: System.out.println("Monday");
        break;
      case Tue: System.out.println("Tuesday");
        break;
      case Wed: System.out.println("Wednesday");
        break;
      case Thu: System.out.println("Thursday");
        break;
      case Fri: System.out.println("Friday");
        break;
      case Sat: System.out.println("Saturday");
```

```
        break;
      case Sun: System.out.println("Sunday");
        break;
      default: System.out.println("What the he...?!");
    }
  }
  public static void main(String[] args) {
    for (WeekDay day : WeekDay.values()) {
      printDayName(day);
    }
  }
}
```

2. Kompilujemy kod (`javac`) i uruchamiamy `JavaAppl`

3. W pliku `lab470.scala` wpisujemy:

```
object Appl470 {
  object WeekDay extends Enumeration {
    type WeekDay = Value
    val Mon, Tue, Wed, Thu, Fri, Sat, Sun = Value
  }
  import WeekDay._
  def printDayName(day: WeekDay) = {
    day match {
      case Mon => println("Monday")
      case Tue => println("Tuesday")
      case Wed => println("Wednesday")
      case Thu => println("Thursday")
      case Fri => println("Friday")
      case Sat => println("Saturday")
      case Sun => println("Sunday")
      case _   => println("What the he..?!")
    }
  }
  def main(args: Array[String]) {
    for (day <- WeekDay.values) println(day)
  }
}
```

4. Kompilujemy kod i uruchamiamy `Appl470`

5. Modyfikujemy metodę `printDayName` w następujący sposób:

```
def printDayName(day: WeekDay) = day match {
  case Mon => println("Monday")
  //...
  case Sun => println("Sunday")
}
```

6. Ponownie kompilujemy kod i uruchamiamy `Appl470`

7. **Constant patterns (vs. *variable patterns*)**

   1. W konsoli (REPL) wpisujemy (tryb `:paste`):

```
def printContInfo(c: Any) = c match {
  case 1     => println("1 matched")
  case 1.1   => println("1.1 matched")
  case 1.1f  => println("1.1f matched")
  case "42"  => println("'42' matched")
  case true  => println("true matched")
  case false => println("false matched")
  case ()    => println("() matched")
  case _     => println("Unknown const")
}
```

   2. Wywołujemy metodę `printContInfo` z kolejnymi stałymi `(1, 1.1, 1.1f, ...)` oraz z kilkoma innymi (dla których nie ma zdefiniowanych wzorców) np. `42`, `"abc"`, `3.14` i obserwujemy wyniki (uwaga: wywołanie dla `()` to `printContInfo(())`)

      Zadanie: dodać kilka innych stałych (o różnych typach) i sprawdzić działanie dopasowania

   3. Modyfikujemy definicję `printContInfo` w następujący sposób:

```
def printContInfo(c: Any) = c match {
  // ... jak poprzednio
  case ()  => println("() matched")
  case any =>
    println("Variable pattern matched, matched: " + any)
}
```

   4. Ponownie wywołujemy metodę `printContInfo` z tymi samymi stałymi. Co się zmieniło w działaniu metody?

   5. Przenosimy regułę `case any =>` do pierwszej linii (tzn. w konsoli wklejamy)

```
def printContInfo(c: Any) = c match {
  case any =>
    println("Variable pattern matched, matched: " + any)
  case 1 => println("1 matched")
```

```
  //...
}
```

6. Analizujemy wynik

8. **Constructor patterns**

   1. W pliku `lab471.scala` wpisujemy:

```
sealed abstract class Expr
case class Number(num: Double) extends Expr
case class UnOp(operator: String, arg: Expr) extends Expr
case class BinOp(operator: String, left: Expr, right: Expr)
extends Expr

object ExprEval {
  def simplify(e: Expr): Expr = e match {
    case UnOp("+", Number(num)) => Number(num)
    //...
    case _ => e
  }
  def evaluate(e: Expr): Double = ExprEval.simplify(e) match {
    case Number(num) => num
    case BinOp("+", left, right) =>
      evaluate(left) + evaluate(right)
    //...
    case _ =>
      println("Unmatched expression!"); 0
  }
}
object Appl471 {
  def main(args: Array[String]) {
    import ExprEval._

    println(simplify(UnOp("+", Number(10))))
    println(evaluate(BinOp("+",
                    BinOp("+", Number(1.5), Number(5.5)),
                    Number(3))))
  }
}
```

   2. Kompilujemy kod, uruchamiamy i analizujemy wyniki.

   3. W metodzie `simplify` dadajemy kolejną regułę upraszczającą (dla `0 + N = N`): `(BinOp("+", ___, ___) => __)` ; testujemy poprawność działania reguły (nowe wyrażenie w metodzie `main`)

   4. W metodzie `evaluate` dodajemy regułę odejmowania wyrażeń (`BinOp("-", ___, ___)`); testujemy poprawność działania reguły (nowe wyrażenie w metodzie `main`)

      Zadanie (opcjonalne): uzupełnić metody `simplify` i `evaluate` tak, aby można było obliczać wartości wyrażeń zawierających mnożenie i dzielenie.

9. **Variable binding and pattern guards** (ćwiczenie opcjonalne)

   1. W pliku `lab471.scala` uzupełniamy metodę `simplify` :

```
def simplify(e: Expr): Expr = e match {
  case UnOp("+", Number(num)) => Number(num)
  //...
  case UnOp("abs", e1 @ UnOp("abs", _)) => e1
  //...
  case _ => e
}
```

   2. W metodzie `main` dodajemy:

```
println(simplify(UnOp("abs", UnOp("abs", Number(-10)))))
```

   3. Kompilujemy kod, uruchamiamy `Appl471` i analizujemy wynik

   4. Ponownie dodajemy nową regułę w metodzie `simplify` :

```
// "if (num >= 0)" <- pattern guard
case UnOp("abs", Number(num)) if (num >= 0) => Number(num)
```

   5. W metodzie `main` dodajemy regułę sprawdzającą:

```
println(simplify(UnOp("abs", Number(10))))
```

   6. Kompilujemy kod, uruchamiamy `Appl471` i analizujemy wynik

10. **Typed patterns**

    1. W konsoli (REPL) wpisujemy (tryb `:paste`):

```
def typedPatterns(tp: Any) = tp match {
  case s: String   => println("String matched, str = " + s)
  case i: Int      => println("Int matched, i = " + i)
  case b: Boolean  => println("Boolean matched, b = " + b)
  case a: Array[_] => println("Array matched, a = " + a)
}
```

2. Testujemy poprawność działania wywołując `typedPatterns` kolejno z argumentami typu `String`, `Int`, `Boolean` i `Array`
3. Wywołujemy `typedPatterns` z argumentem typu `Double` i analizujemy wynik
4. Dopisujemy regułę (nowa definicja metody w REPL):

```
case _ => println("Default handler")
```

5. Wywołujemy `typedPatterns` z argumentem typu `Double` i analizujemy wynik

11. **Typ `Option`**

1. W konsoli (REPL) definiujemy dwie metody:

```
def wrap(s: String) = s match {
  case s if (s != null) => Some(s)
  case _ => None
}

import scala.language.implicitConversions
implicit def unwrap(opt: Option[String]) = opt match {
  case Some(s) => s
  case None => ""
}
```

2. Sprawdzamy, jaki jest typ wyniku metody `wrap`
3. W konsoli (REPL) wpisujemy kolejno:

```
scala> wrap("abc")
scala> wrap("")
scala> wrap(null)
```

4. Definiujemy tablicę `a1`:

```
val a1 = Array("s1", "s2", null, "s3")
```

5. Tworzymy tablicę, której elementami są liczby oznaczające długości kolejnych elementów tablicy `a1`:

```
val s1Lengts = for (s <- a1) yield s.length
```

6. Analizujemy komunikat błędu
7. Wpisujemy w konsoli (REPL) kolejno poniższe linie i analizujemy wyniki:

```
scala> val a1Safe = for (s <- a1) yield wrap(s)
scala> val s1Lengts1 = for (s <- a1Safe) yield unwrap(s).length
scala> val s1Lengts2 = for (s <- a1Safe) yield s.length
```

Dlaczego ostatnia linia nie spowodowała błędu?

Zadanie (opcjonalne): zdefiniować jeszcze raz metodę `unwrap`, ale tym razem bez `implicit` na początku, następnie wpisać w REPL jeszcze raz ostatnią z powyższych trzech linii (`val s1Lengts2 = ...`) i przeanalizować wynik

12. **Ekstraktory** (ćwiczenie opcjonalne)

1. W konsoli (REPL) wpisujemy (w trybie `:paste`):

```
abstract class Expr
class Number(val num: Double) extends Expr
class UnOp(val operator: String, val arg: Expr) extends Expr

object Number {
  def apply(num: Double) = new Number(num)
  def unapply(e: Number): Option[Double] =
    if (e != null) Some(e.num) else None
}

object UnOp {
  def apply(op: String, arg: Expr) = new UnOp(op, arg)
  def unapply(op: UnOp): Option[(String, Expr)] =
    if (op != null) Some((op.operator, op.arg)) else None
}

def handleExpr(e: Expr) = e match {
  case Number(num) => println("Number matched, num = " + num)
  case UnOp(op, _) => println("UnOp matched, operator = " + op)
  case _ => println("Default handler...")
}
```

2. W konsoli wpisujemy kolejno:

```
scala> handleExpr(Number(10))
scala> handleExpr(UnOp("+", Number(10)))
```

i analizujemy wyniki

Zadanie (opcjonalne): na podstawie powyższego przykładu przepisać kod z ćwiczenia *Constructor Patterns* bez użycia *case classes*.