Lab 4. Data types in Scala

Objectives:

- get familiar with **algebraic data types**
- get familiar with **pattern matching** and **recursion** with them

4.1 Natural Numbers

Given the following implementation of the natural numbers, solve the next few exercises.

```
trait Nat
case object Zero extends Nat
case class Succ(x: Nat) extends Nat
```

4.1.1 Write a function which takes two natural numbers, and returns their sum.

```
def add(x: Nat, y: Nat): Nat = ???
```

4.1.2 Write a function which takes two natural numbers, and returns their product.

```
def multiply(x: Nat, y: Nat): Nat = ???
```

4.1.3 Write a function which takes an int and converts it to a Nat.

```
def toNat(x: Int): Nat = ???
```

4.2 Option

Option = carrier (like a box or a container) for a single or no element, of a given type. (Ex. `Some(_)` or `None`)

We use Option to write robust functions, in case they return null or fail to return an accepted value.

4.2.1 Let's revisit the function `realtrycatch` now that we have a type that represents the possibility of error. If an error occurs (try function returns `None`), the catch function will be called instead.

```
def realrealtrycatch(t: => Option[Int], c: => Int): Int = {
  ???
}
```

(!) 4.2.2 Refactor the function toNat(), so that it takes an integer (a positive or negative number) and returns a “container” of a Nat.

```
def toNatOpt(x: Int): Option[Nat] = ???
```

(!) 4.2.3 Refactor the function add(), so that it takes two “containers” of Nats and returns a “container” of a Nat.

```
def addOpt(x: Option[Nat], y: Option[Nat]): Option[Nat] = ???
```

4.3 Binary Trees

Given the following implementation of binary trees, solve the next few exercises.

```
trait BTree
case object EmptyTree extends BTree
case class Node(value: Int, left: BTree, right: BTree) extends BTree
```

4.3.1 Write a function which takes a BinaryTree and returns its depth.

```
def depth(tree: BTree): Int = ???
```

4.3.2 Write a function which takes a BinaryTree and returns the number of nodes in its subtree.

```
def subtree(tree: BTree): Int = ???
```

4.3.3 Write a function which takes a BinaryTree and returns the number of nodes with even number of children.

```
def evenChildCount(tree: BTree): Int = ???
```

4.3.4 Write a function which takes a BinaryTree and flattens it (turns it into a list containing the values of the nodes).

```
def flatten(tree: BTree): List[Int] = ???
```

4.3.5 Write a function which takes a BinaryTree and return the number of nodes whose values follow a certain rule.

```
def countNodes(tree: BTree, cond: Int => Boolean): Int = ???
```

4.3.6 Write a function which takes a BinaryTree and return mirrored BinaryTree.

```
def mirror(tree: BTree): BTree = ???
```

(!) 4.3.7 Write a function which takes two BinaryTree and tries to assign the second tree as a child of the first. It should return a “container” of a BinaryTree .

```
def append(tree1: BTree, tree2: BTree): Option[BTree] = ???
```

4.4 Expression evaluation

Given the following implementation of expressions, solve the next few exercises.

```
trait Expr
case class Atom(a: Int) extends Expr
case class Add(e1: Expr, e2: Expr) extends Expr
case class Mult(e1: Expr, e2: Expr) extends Expr
```

4.4.1 Write a function which takes an Expression and evaluates it.

```
def evaluate(e: Expr): Int = ???
```

4.4.2 Write a function which takes an Expression and simplifies it. (Ex. a * (b + c) → remove parentheses → ab + ac)

```
def simplify(e: Expr): Expr = ???
```

4.4.3 Write a function which takes an Expression and removes 'useless' operations. (Ex. a * 1 → a, a + 0 → a)

```
def optimize(e: Expr): Expr = ???
```

If you work outside of worksheets, you can define the trait as:

```
trait Expr {
    def + (that: Expr): Expr = Add(this, that)
    def * (that: Expr): Expr = Mul(this, that)
}
case class Atom(a: Int) extends Expr
case class Add(e1: Expr, e2: Expr) extends Expr
case class Mult(e1: Expr, e2: Expr) extends Expr
```

With the operators defined, you can create expressions writting:

```
(Atom(1) + Atom(2) * Atom(3)) + Atom(4)
```

instead of:

```
Add(Add(Atom(1), Mult(Atom(2), Atom(3))), Atom(4))
```

4.5 Matrix manipulation

We shall represent matrices as *lists of lists*, i.e. values of type `[ [Integer ] ]`. Each element in the outer list represents a line of the matrix. Hence, the matrix

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

will be represented by the list `[ [1,2,3],[4,5,6],[7,8,9] ]`.

To make signatures more legible, add the *type alias* to your code:

```
 type Matrix = List[List[Int]]
```

which makes the type-name `Matrix` stand for `[ [Integer] ]`.

4.5.1 Write a function that computes the scalar product with an integer:

$$2 * \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} = \begin{pmatrix} 2 & 4 & 6 \\ 8 & 10 & 12 \\ 14 & 16 & 18 \end{pmatrix}$$

```
def scalarProd(m: Matrix)(v: Int): Matrix = ???
```

4.5.2 Write a function which adjoins two matrices by extending rows (horizontally):

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} hjoin \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} = \begin{pmatrix} 1 & 2 & 5 & 6 \\ 3 & 4 & 7 & 8 \end{pmatrix}$$

```
def hJoin(m1: Matrix, m2: Matrix): Matrix = ???
```

4.5.3 Write a function which adjoins two matrices by adding new rows (vertically):

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} vjoin \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} = \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \\ 7 & 8 \end{pmatrix}$$

```
def vJoin(m1: Matrix, m2: Matrix): Matrix = ???
```

4.5.4 Write a function which adds two matrices, element by element:

```
def matSum(m1: Matrix, m2: Matrix): Matrix = ???
```