Lab 2. High order functions

Objectives:

- implement and use **higher-order** functions. A **higher-order** function takes other functions as parameter or returns them
- implement **curry** and **uncurry** functions, and how they should be properly used (review lecture).

Create a new Scala worksheet to write your solutions

2.1 Intro. Functions as parameters

2.1.1 Write a function `apply` that takes an integer and return the result of the applied function on the given integer. Start from the code stub below:

```
def apply(n: Int, f: Int => Int): Int = {
   ???
}
```

2.1.2 Write a function `doubler` that returns a function that doubles the input it receives (an integer). Start from the code stub below:

```
def doubler(): Int => Int = {
   ???
}
```

2.1.3 Create a function `trycatch` that takes an integer and evaluates its value using the try function. If an error occurs (try function returns 0), the catch function will be called instead.

```
def trycatch(t: Int => Int, c: Int => Int)(x: Int): Int = {
   ???
}
```

2.1.4 Write a function `realtrycatch` where t and c take no parameters and produce a result upon evaluation. If an error occurs (try function returns 0), the catch function will be called instead.

```
def realtrycatch(t : => Int, c: => Int): Int  = {
   ???
}
```

2.2 Custom high order functions

2.2.1 Define the function `foldWith` which uses an operation `op` to reduce a range of integers to a value. For instance, given that `op` is addition (+), the result of folding the range 1 to 3 will be 1+2+3=6. `foldWith` should be curried (it will take the operation and return another function which expects the bounds).

```
def foldWith (op: (Int,Int) => Int)(start: Int, stop: Int): Int = {
  def tail_fold(crt: Int, acc: Int): Int  = ???
  ??
}
```

2.2.2 Define the function `foldConditional` which extends `foldWith` by also adding a predicate `p: Int ⇒ Boolean`. `foldConditional` will reduce only those elements of a range which satisfy the predicate.

```
def foldConditional(op: (Int,Int) => Int, p: Int => Boolean)(start: Int, stop: Int):
Int = ???
```

2.2.3 Write a function `foldMap` which takes values $a_1, a_2, \ldots, a_k$ from a range and computes $f(a_1)\ op\ f(a_2)\ op\ \ldots f(a_k)$. Use the `apply` and `foldWith` methods

```
def foldMap(op: (Int,Int) => Int, f: Int => Int)(start: Int, stop: Int): Int = ???
```

2.3 Curry vs Uncurry

2.3.1 Modify the function below so that it's curry and use it to calculate `5*3`

```
def multiply(x:Int, y:Int): Int => x * y
```

2.3.2 Modify the function below so that it's curry and use it to compare 3 numbers and return the maximum

```
def compare(x: Int, y: Int, z: Int): Int =
{
  if x > y && x > z then
    x
  else if y > x && y > z then
    y
  else
    z
}
```

2.4 Function transformations

The graph of a function can undergo different geometric transformation such as scaling, shifting, rotating, mirroring and so on. The result of those transformation will also be a function that looks similarly to the original. In this exercice we will particularly work with lines. A line is a linear equation of the form $f(x) = a * x + b$

2.4.1 Implement a function that shifts a line on Oy axis by a certain amount Δy

```
def shiftOY(line: Double => Double, delta_y: Double): Double => Double = {
   ???
}
```

2.4.2 Implement a function that shifts a line on Ox axis by a certain amount Δx

```
def shiftOX(line: Double => Double, delta_x: Double): Double => Double = {
   ???
}
```

2.4.3 Implement a function that checks if two lines intersect at an integer value from a given interval

```
def intersect(line1: Double => Double, line2: Double => Double)(start: Int, stop: In
t): Boolean = {
   ???
}
```