# Welcome to Programming Fundamentals 3

## University of Luxembourg

2025/2026

Bachelor in Computer Sciences (BICS)
Semester 3

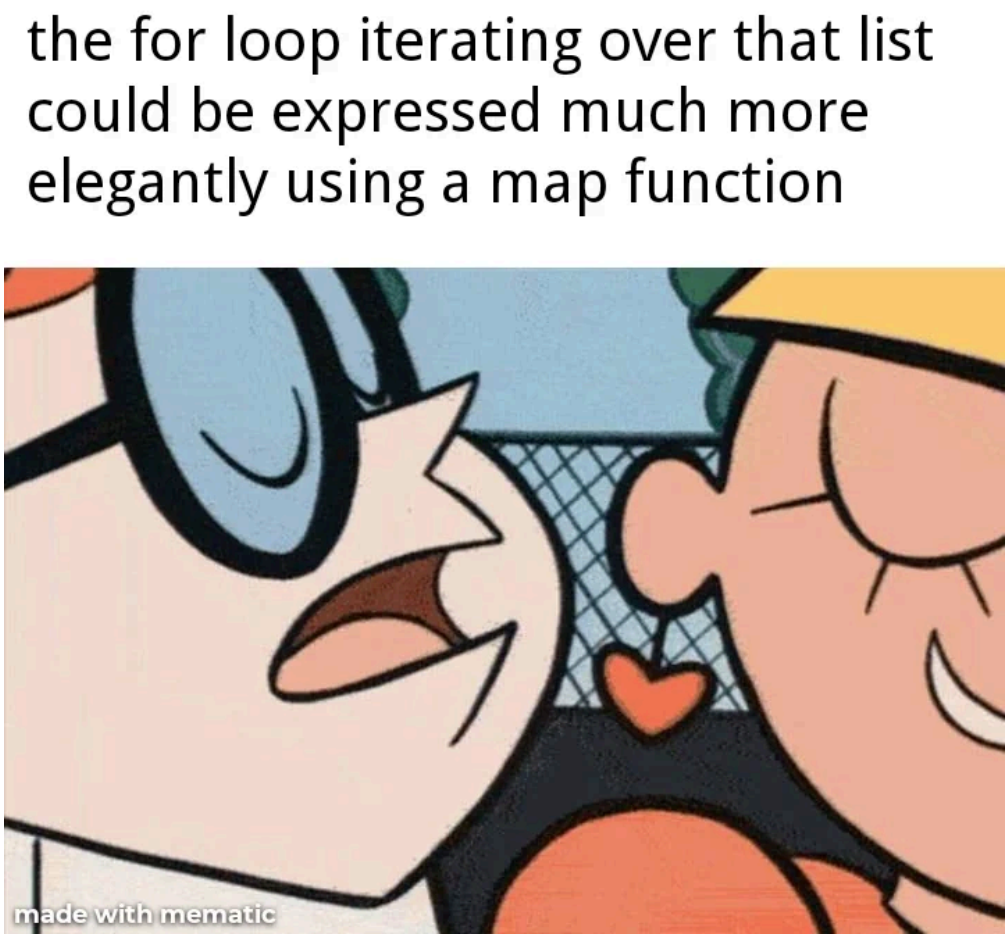
Lecture - Week #7

Dr. Afonso DELERUE ARRIAGA afonso.delerue@uni.lu

**Important**: Please remember to register your attendance in Moodle!

## Today: Midterm code challenge! 💪

Work in teams of 2-3 students to tackle the problems below.




## Problem #1: Sorting some

Given a list of objects of the same type you have to sort those having a specific property, according to a given rule, while leaving the others at their original positions. **Read every sub-problem before starting!**

### A. Sorting evens

Sort only the even numbers (from the smallest to the largest) inside a given list `l :: [Int]`.

```
sortEvens :: [Int] -> [Int]
```

*Input --> expected output:*

```
[5, 3, 2, 8, 1, 4] --> [5,3,2,4,1,8]
[5, 3, 1, 8, 0]    --> [5,3,1,0,8]
(reverse [1..10])  --> [2,9,4,7,6,5,8,3,10,1]
[]                 --> []
```

### B. Sorting vowels

Sort only the vowels with alphabetical order inside a given `String`.

```
sortVowels :: String -> String

map sortVowels ["house", "route" , "castle", "book"] -->
["heosu","reotu","castle","book"]
```

### C. Generic function

Write a generic function that takes a list of type `[a]` in typeclass `Ord` and a function `f :: a -> Bool`, and returns a list of objects of type `[a]`, whose elements that pass the predicate test and return `True` are sorted, while the other elements remain in place. The elements must be sorted by using the function `sort` from `Data.List`.

```
import Data.List (sort)

sortSome :: Ord a => (a -> Bool) -> [a] -> [a]
```

Notice that you can easily implement `sortVowels` and `sortEvens` in terms of `sortSome`. How?

## Problem #2: Expressions

Given the type declaration `data Expr = Val Int | Prod Expr Expr`, define a higher-order function

```
foldex :: (Int -> a) -> (a -> a -> a) -> Expr -> a
```

such that `foldex f g` replaces each `Val` constructor in an expression by the function `f`, and each `Prod` constructor by the function `g`.

**Example:**

```
let z = Prod (Val 7) (Prod (Val 3) (Val 1))
print $ foldex show (++) z
```

should return `"731"`, while

```
print $ foldex (+1) (-) z
```

should return `6`.

**Evaluate expressions**

Using `foldex`, define a function `eval :: Expr -> Int` that evaluates an expression to an integer value by evaluating the `Prod` as the product between integers.

## Problem #3: Grids

### A. Set of coordinates

Suppose that a coordinate grid of size $m \times n$ is given by the list of all pairs $(x, y)$ of integers such that $0 \leq x \leq m$ and $0 \leq y \leq n$. Using a list comprehension, define a function `grid :: Int -> Int -> Grid` that returns a coordinate grid `type Grid = [(Int,Int)]` of a given size.

**Example:**

`grid 1 2` should return `[(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2)]`.

Then, using a **list comprehension** and `grid`, define a function `diag :: Int -> Grid` returning the coordinates of the elements on the diagonal of a grid of squared size. For example `diag 2` should return `[(0,0),(1,1),(2,2)]`.

### B. Plotting grid

Consider the following function:

```
printGrid :: Int -> Int -> IO ()
printGrid r c = putStrLn $ unlines [ replicate c '*' | _ <-
[1..r] ]
```

For example `printGrid 4 30` should return

```
******************************
******************************
******************************
******************************
```

Modify this function in order to get a function that prints `*` for odd lines and `+` for even lines.

```
printGrid' : Int -> Int -> IO ()
```

For example, `printGrid' 4 30` should give

```
******************************
++++++++++++++++++++++++++++++
******************************
++++++++++++++++++++++++++++++
```

## Problem #4: Comparing trees

Consider a new data type `Tree` defined as follows:

```
data Tree = Node { val :: Int, left, right :: Maybe Tree }
```

Write a function `compare` that compares two `Maybe Tree` and returns `True` if an only if both trees have the same internal structure and nodes contain the same exact values. Note that function `compare` is already defined in `Prelude`, so you need to hide it in order to avoid conflicts.

```
import Prelude hiding (compare)

data Tree = Node { val :: Int, left, right :: Maybe Tree }

compare :: Maybe Tree -> Maybe Tree -> Bool
compare _ _ = undefined
```