# Welcome to Software Foundations

## University of Luxembourg

2023/2024

Bachelor in Computer Sciences (BICS)
Semester 3

Lecture - Week #3

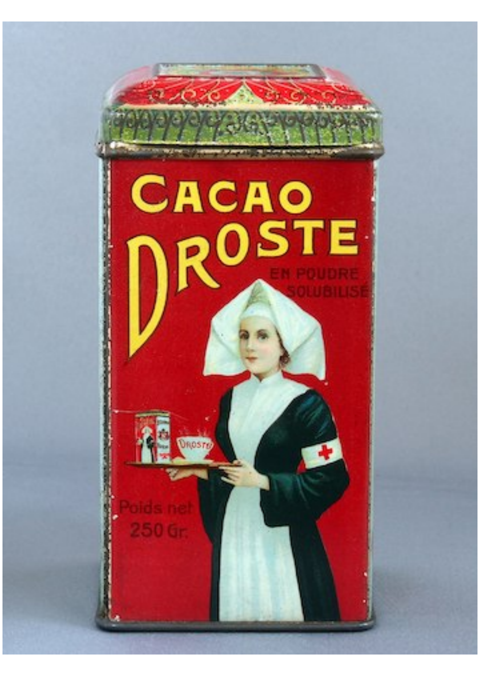
Dr. Afonso DELERUE ARRIAGA afonso.delerue@uni.lu

## Recap of Lecture #2

- Conditionals
  - if-then-else statements
  - guarded equations
  - case expressions
- Infinite lists and lazy evaluation
- Types
  - Bool, Char, Int (bounded), Integer (unbounded), Float, Double, ...
  - Type checking at compile time
  - Polymorphism
  - Type classes
- Local definitions
  - `where`
  - `let` ... `in` ...

## Recursion

Recursion plays a central role in Haskell, and is also used throughout computer science and mathematics. It is essentially a mechanism for looping.




A visual form of recursion known as the Droste effect, named after a Dutch brand of cocoa.

We have seen that many functions can naturally be defined in terms of other functions.

```
factorial n = product [1..n]
```

In Haskell, it is also permissible to define functions in terms of themselves, in which case the functions are called *recursive*. For example, we've also seen that the factorial function can be defined as follows:

```
factorial 0 = 1
factorial n = n * factorial (n-1)
```

- The first equation defines the base case.
- The second equation defines the function `factorial` in terms of itself.

### Why recursive functions? 🤷‍♂️

- We can't use cycles in a purely functional language because we can't modify variables.
- The only functional way to express repetition is to use recursion.
- Fortunately, any algorithm that can be written with cycles can also be written with recursive functions.
- Defining functions in terms of themselves can lead to elegant and concise solutions to many problems.
- Haskell has a strong type system, which helps ensure that recursive functions are well-defined and terminate, making it a safer language for recursive programming.

### Recursion over lists

- Recursion is not exclusive to functions on integers, but can also be used to define functions on lists.

**Examples**

```
product :: [Int] -> Int
product []     = 1
product (x:xs) = x * product xs

length :: [a] -> Int
length []      = 0
length (_:xs)  = 1 + length xs
```

### Recursive functions with mutiple arguments

Functions with multiple arguments can also be defined using recursion on more than one argument at the same time.

**Example**

```
zip :: [a] -> [b] -> [(a, b)]
zip _ [] = []
zip [] _ = []
zip (x:xs) (y:ys) = (x,y) : zip xs ys
```

### Multiple recursion

A function may also be applied more than once in its own definition, resulting in a multiple recursion definition.

**Example**

```
fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib n = fib (n-2) + fib (n-1)
```

### Mutual recursion

The classic example is the `isEven` and `isOdd` functions, which determine if a given number is even or odd.

```
isEven 0 = True
isEven n = isOdd (n-1)

isOdd 0 = False
isOdd n = isEven (n-1)
```

### Quicksort

- Quicksort is an efficient, general-purpose sorting algorithm.
- It can be defined recursively to sort a list:
  - If the list is **empty**, then the list is already sorted.
  - If the list is **not empty**, let `x` be the head of the list and `xs` the tail of the list in
    1. Recursively sort all element in `xs` that are **smaller** than `x`.
    2. Recursively sort all element in `xs` that are **greater** than `x`.
    3. Concatenate the two lists with `x` in the middle.

In [ ]:
```
quicksort :: Ord a => [a] -> [a]
quicksort [] = []
quicksort (x:xs) = quicksort smallerThanX ++ [x] ++ quicksort biggerThanX
  where
    smallerThanX = [v | v <- xs, v < x]
    biggerThanX  = [v | v <- xs, v >= x]

quicksort [5,2,4,9,10,13]
quicksort "Programming Fundamentals 3"

-- mergesort: exercise at home
```

```
[2,4,5,9,10,13]
"  3FPaaadeggilmmmnnnorrstu"
```

### Step by step

Let's break down the execution step by step:

```
qsort [5,2,4,9,10,13]
= qsort [2,4] ++ [5] ++ qsort [9,10,13]
= (qsort [] ++ [2] ++ qsort [4]) ++ [5] ++ (qsort [] ++ [9] ++ qsort [10,13])
= (qsort [] ++ [2] ++ qsort [4]) ++ [5] ++ (qsort [] ++ [9] ++ (qsort [] ++ [10] ++
qsort [13]))
= [2,4,5,9,10,13]
```

### Recap of different recursive examples

- Recursion on integers
- Recursion on lists
- Recursive functions with multiple arguments
- Multiple recursion
- Mutual recursion

## How to do recursion

### The 5-step process to write recursive functions

1) Define the type of the function 2) Enumerate the cases 3) Define the simple cases 4) Define the other cases 5) Generalize and simplify

### Example 1

Let's try out the 5-step process by re-defining the Prelude function `drop` that removes a given number of elements from the start of a list.

### Example 1: Step 1

Let's try out the 5-step process by re-defining the Prelude function `drop` that removes a given number of elements from the start of a list.

**Step 1: Define the type of the function**

```
drop' :: Int -> [a] -> [a]
```

### Example 1: Step 2

Let's try out the 5-step process by re-defining the Prelude function `drop` that removes a given number of elements from the start of a list.

**Step 2: Enumerate the cases**

```
drop' :: Int -> [a] -> [a]
drop' 0 []     =
drop' 0 (x:xs) =
drop' n []     =
drop' n (x:xs) =
```

### Example 1: Step 3

Let's try out the 5-step process by re-defining the Prelude function `drop` that removes a given number of elements from the start of a list.

**Step 3: Define the simple cases**

```
drop' :: Int -> [a] -> [a]
drop' 0 []     = []
drop' 0 (x:xs) = (x:xs)
drop' n []     = []
drop' n (x:xs) =
```

### Example 1: Step 4

Let's try out the 5-step process by re-defining the Prelude function `drop` that removes a given number of elements from the start of a list.

**Step 4: Define the other cases**

```
drop' :: Int -> [a] -> [a]
drop' 0 []     = []
drop' 0 (x:xs) = (x:xs)
drop' n []     = []
drop' n (x:xs) = drop' (n-1) xs
```

### Example 1: Step 5

Let's try out the 5-step process by re-defining the Prelude function `drop` that removes a given number of elements from the start of a list.

**Step 5: Generalize and simplify**

Generalize

```
drop' :: Integral b => b -> [a] -> [a]
drop' 0 []     = []
drop' 0 (x:xs) = (x:xs)
drop' n []     = []
drop' n (x:xs) = drop' (n-1) xs
```

Simplify

```
drop' :: Integral b => b -> [a] -> [a]
drop' 0 xs     = xs
drop' _ []     = []
drop' n (_:xs) = drop' (n-1) xs
```

## How to do recursion (cont.)

### Example 2

Because we don't have loops in Haskell, it's really important to understand recursion for repetition. Let us give another try at the 5-step process by re-defining Prelude function `init` that removes the last element of a list.

### Example 2: Step 1

Because we don't have loops in Haskell, it's really important to understand recursion for repetition. Let us give another try at the 5-step process by re-defining Prelude functino `init` that removes the last element of a list.

**Step 1: Define the type of the function** Lists can be of any type.

```
init' :: [a] -> [a]
```

### Example 2: Step 2

Because we don't have loops in Haskell, it's really important to understand recursion for repetition. Let us give another try at the 5-step process by re-defining Prelude functino `init` that removes the last element of a list.

**Step 2: Enumerate the cases** We cannot remove the last element of a list if the list if already empty. Therefore, we only have one case.

```
init' :: [a] -> [a]
init' (x:xs) =
```

### Example 2: Step 3

Because we don't have loops in Haskell, it's really important to understand recursion for repetition. Let us give another try at the 5-step process by re-defining Prelude functino `init` that removes the last element of a list.

**Step 3: Define the simple cases**

If `x` is the last element of the list, return an empty list.

```
init' :: [a] -> [a]
init' (x:xs) | null xs = []
             | otherwise =
```

### Example 2: Step 4

Because we don't have loops in Haskell, it's really important to understand recursion for repetition. Let us give another try at the 5-step process by re-defining Prelude functino `init` that removes the last element of a list.

**Step 4: Define the other cases**

Otherwise we keep `x` and remove the last element from `xs`.

```
init' :: [a] -> [a]
init' (x:xs) | null xs = []
             | otherwise = x : init' xs
```

### Example 2: Step 5

Because we don't have loops in Haskell, it's really important to understand recursion for repetition. Let us give another try at the 5-step process by re-defining Prelude functino `init` that removes the last element of a list.

**Step 5: Generalize and simplify**

Type is already generalized, so nothing to do in that regard.

```
init' :: [a] -> [a]
init' (x:xs) | null xs = []
             | otherwise = x : init' xs
```

Instead of using guards, we could enumerate the case where the list only has 1 element.

```
init' :: [a] -> [a]
init' [_] = []
init' (x:xs) = x : init' xs
```

## Exercises 🧩

### (1) Growth of a Population

- https://www.codewars.com/kata/563b662a59afc2b5120000c6

### (2) Multiples of 3 or 5

- https://projecteuler.net/problem=1

In [ ]:
```
-- problem 1
nbYears :: Int -> Float -> Int -> Int -> Int
nbYears p0 percent aug p
    | p0 >= p = 0
    | otherwise = 1 + nbYears (floor (fromIntegral p0 * (1 + percent/100)) + aug) percent aug p

nbYears 1000 2 50 1200

-- problem 2
sum [x |x  <- [1..999], (x `mod` 3 == 0) || (x `mod` 5 == 0)]
```

```
3
233168
```

## Questions from the class ❓

Question: The generalized signature type of `quicksort` is `quicksort :: Ord a => [a] -> [a]`. Why generic type `a` has to be constrained on typeclass `Ord`?

Answer: The `Ord` class is used for totally ordered datatypes. Data types that are instances of `Ord` are guaranteed to be applicable to the following functions:

```
class Eq a => Ord a where
    compare :: a -> a -> Ordering
    (<) :: a -> a -> Bool
    (<=) :: a -> a -> Bool
    (>) :: a -> a -> Bool
    (>=) :: a -> a -> Bool
    max :: a -> a -> a
    min :: a -> a -> a
```

Our `quicksort` function makes use of `(<=)` and `(>)`, thus `a` has to be constrained on class `Ord`. In simple terms: we can only sort 'things' that have an order. It happens that all data types we've seen so far are instances of class `Ord`, but we will see later on that we can create our own data types and comparison is not granted unless we make those new data types instances of `Ord`.

Question: How is `quicksort` different from `mergesort`?

Answer: `quicksort` takes the head of the list and splits the tail into two sublist of elements based on the critea 'less or equal' or 'greater than' the head, and then recursively applies itself to these sublists and places the head in between these two sorted lists. In the base case, the empty list is always sorted.

`mergesort` splits the list to be sorted into two halves, recursively applies itself to the sublists and merges the two sublists together, which is an easy task since these are now sorted. In the base case, an empty list and a list with a single element is always sorted. Here is a possible implementation of `mergesort`:

```
mergesort :: Ord a => [a] -> [a]
mergesort [] = []
mergesort [x] = [x]
mergesort xs = let (left, right) = split xs
               in merge (mergesort left) (mergesort right)

split :: [a] -> ([a], [a])
split xs = splitAt (length xs `div` 2) xs

merge :: Ord a => [a] -> [a] -> [a]
merge [] ys = ys
merge xs [] = xs
merge (x:xs) (y:ys)
    | x <= y    = x : merge xs (y:ys)
    | otherwise = y : merge (x:xs) ys
```