Welcome to Programming Fundamentals 3

University of Luxembourg

2025/2026

Bachelor in Computer Sciences (BICS)
Semester 3

Lecture - Week #4

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

Important: Please remember to register your attendance in Moodle!

Recap of Lecture #3

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

Curried functions

- Although is may be helpful to think otherwise, every function in Haskell officially only takes one parameter.
- All the functions that accepted several parameters are in fact curried functions. What does it mean?

Function `max`

- First, notice that we can use parentheses without affecting the result.

```
max 7 8
8
```

```
(max 7) 8
8
```

Curried functions (cont.)

Function `max`

- To understand why, let's look at the type of `max` and `(max 7)`:

```
:t max
max :: Ord a => a -> a -> a

:t (max 7)
(max 7) :: (Ord a, Num a) => a -> a
```

- In other words, the signature of `max` could have parentheses and be read as a function that takes an `a` of typeclass `Ord` and returns a function from `a -> a`:

```
max :: Ord => a -> (a -> a)
max = \x -> (\y -> if x>=y then x else y)
```

- By calling `max` with only one argument, we obtain a **partially applied** function.

Curried functions (cont.)

Function `max'` with a single input

- We could have also defined `max'` taking as input a tuple.

```
max' :: Ord a => (a,a) -> a
max' (x,y) = if x>=y then x else y

max' (3,4)
4
```

Curried functions (cont.)

Function `max'` with a single input

- In this case, we cannot directly obtain a partially applied function, but the high-order library function `curry` converts a function on pairs into a curried function.

```
max' (3,4)
4

curry max' 3 4
4
```

```
:t curry
curry :: ((a, b) -> c) -> a -> b -> c
```

- Notice that the first argument of `curry` is a function.

Curried functions (cont.)

Function `uncurry`

- Conversely, `uncurry` converts a curried function with two arguments into a function on pairs.

```
uncurry max (3,4)
4
```

Curried functions (cont.)

What is a curried function?

- A curried function is a function of several arguments rewritten such that it accepts the first argument and returns a function that accepts the second argument and so on. This allows functions of several arguments to have some of their initial arguments partially applied.
- Fun fact: The "Curry" in "Currying" is a reference to logician **Haskell Curry**, who used the concept extensively.

Exercise 💪

- Prelude function `uncurry` converts a curried function with two arguments into a function on pairs.

```
uncurry max (3,4)
4
```

- Without asking GHCI, what is the type of `uncurry`?
- Define function `isUpperCase` that checks if a `Char` is an upper case character by partially applying function `elem`.

```
isUpperCase :: Char -> Bool
```

In [ ]:

```haskell
:opt lint

curry' :: ((a,b) -> c) -> a -> b -> c
curry' f x y = f (x,y)
```

```haskell
uncurry' :: (a -> b -> c) -> (a,b) -> c
uncurry' f (x,y) = f x y

:type curry
:type curry'
:type uncurry
:type uncurry'

isUpperCase :: Char -> Bool
isUpperCase c = (`elem` ['A'..'Z']) c

isUpperCase 'J'
```

```
curry :: forall a b c. ((a, b) -> c) -> a -> b -> c
curry' :: forall a b c. ((a, b) -> c) -> a -> b -> c
uncurry :: forall a b c. (a -> b -> c) -> (a, b) -> c
uncurry' :: forall a b c. (a -> b -> c) -> (a, b) -> c
```

```
True
```

Higher-order functions

- A higher-order function is a function that accepts a function as an argument or returns a function.
- `curry` and `uncurry` are two examples of functions that receive functions as arguments.
- Some higher-order functions defined in Prelude:
  - `map`
  - `filter`
  - `takeWhile`, `dropWhile`
  - `all`, `any`
  - `foldr`, `foldl`

Higher-order functions (cont.)

Function `map`

Function `map` applied a function to every element of a list.

```haskell
map :: (a -> b) -> [a] -> [b]
```

It can be defined as a list comprehension:

```haskell
map f xs = [f x | x <- xs]
```

Or recursively:

```haskell
map f [] = []
map f (x:xs) = f x : map f xs
```

In [2]:
```
:opt no-lint

map (^2) [1,2,3,4]

isUpperCase :: Char -> Bool
isUpperCase = (`elem` ['A'..'Z'])

map isUpperCase "Hello World!"
```

```
[1,4,9,16]
[True,False,False,False,False,False,True,False,False,False,False,False]
```

Higher-order functions (cont.)

Function `filter`

Filter function filters elements of a list that satisfy some predicate, i.e. a function that returns a Bool.

```
filter :: (a -> Bool) -> [a] -> [a]
```

As with `map`, we can easily define `filter` as a list comprehension:

```
filter p xs = [x | x <- xs, p x]
```

Or recursively:

```
filter p [] = []
filter p (x:xs) | p x = x : filter p xs
                | otherwise = filter p xs
```

In [4]:
```
filter odd [1..100]
filter isUpperCase "HELLO world"
```

```
[1,3,5,7,9,11,13,15,17,19,21,23,25,27,29,31,33,35,37,39,41,43,45,47,49,51,53,55,
57,59,61,63,65,67,69,71,73,75,77,79,81,83,85,87,89,91,93,95,97,99]
"HELLO"
```

Higher-order functions (cont.)

Functions `takeWhile` and `dropWhile`

Function `takeWhile` returns the longest prefix of a list that satisfies some predicate. In other words, `takeWhile` goes through a list, starting with the head and selects all elements until one is found that does't satisfy the predicate.

Conversely, `dropWhile` removes the longest prefix of a list that satisfies some predicate, returning the tail of the list thereafter.

```
takeWhile :: (a -> Bool) -> [a] -> [a]
```

```
dropWhile :: (a -> Bool) -> [a] -> [a]
```

We can define `takeWhile` recursively:

```
takeWhile p [] = []
takeWhile p (x:xs) | p x = x : takeWhile p xs
                   | otherwise = []
```

Higher-order functions (cont.)

Functions `takeWhile` and `dropWhile`

Similarly, we can also define `dropWhile` recursively:

```
dropWhile p [] = []
dropWhile p (x:xs) | p x = dropWhile p xs
                   | otherwise = (x:xs)
```

Alternatively, we can also define `dropWhile` with the help of `drop`, `length` and `takeWhile`

```
dropWhile' p xs = drop (length (takeWhile p xs)) xs
```

In [4]:
```
takeWhile (>5) [6,7,10,2,20]
dropWhile (>5) [6,7,10,2,20]

dropWhile' p xs = drop (length (takeWhile p xs)) xs
dropWhile' (>5) [6,7,10,2,20]

filter (>5) [6,7,10,2,20]
```

```
[6,7,10]
[2,20]
[2,20]
[6,7,10,20]
```

Higher-order functions (cont.)

Functions `all` and `any`

Function `all` only returns `True` if all elements of a list satisfy some predicate. Function `any` returns `True` if **any** element of a list satisfies some predicate, and returns `False` otherwise (i.e. no element of the list satisfies the predicate).

Functions `all` and `any` have the same type:

```
all :: (a -> Bool) -> [a] -> Bool

any :: (a -> Bool) -> [a] -> Bool
```

We can define `all` recursively:

```
all p [] = True
all p (x:xs) | p x = True && all p xs
             | otherwise = False
```

Higher-order functions (cont.)

Functions `all` and `any`

Similarly, we can also define `any` recursively:

```
any p [] = False
any p (x:xs) | p x = True
             | otherwise = False || any p xs
```

In [5]:
```
:opt no-lint

all (>5) [1,2,3,9,4,6]
all (odd) [3,5,7,11]

any (>5) [1,2,3,9,4,6]
any (\x -> x `mod` 2 == 0) [3,5,7,11]
any (even) [3,5,7,11]
```

```
False
True
True
False
False
```

Higher-order functions (cont.)

Folding

You've probably noticed by now that many functions over lists follow the same recursive pattern:

```
f [] = z
f (x:xs) = x # f xs
```

Function `f` maps the empty list to some value `z`. Some operator `#` is used to **combine** the head of the list and the result of recursively applying `f` to the tail of the list.

Higher-order functions (cont.)

Folding

Recall a few examples:

```
sum :: [Int] -> Int
sum [] = 0
```

```
sum (x:xs) = x + sum xs

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

Higher-order functions (cont.)

Folding

```
or :: [Bool] -> Bool
or [] = False
or (x:xs) = x || or xs

and :: [Bool] -> Bool
and [] = True
and (x:xs) = x && and xs
```

Higher-order functions (cont.)

`foldr`

The higher-order function `foldr` (abbreviating *fold right*) abstracts this pattern, with operator `#` and value `z` as arguments.

The previous examples can be rewritten with `foldr` as follows:

```
sum :: [Int] -> Int
sum xs = foldr (+) 0 xs

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

Higher-order functions (cont.)

`foldr`

```
or :: [Bool] -> Bool
or xs = foldr (||) False xs

and :: [Bool] -> Bool
and xs = foldr (&&) True xs
```

Higher-order functions (cont.)

`foldr`

The `foldr` function itself is part of Prelude and can be defined recursively, following the pattern that we initially observed:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z [] = z
foldr f z (x:xs) = x `f` (foldr f z xs)
```

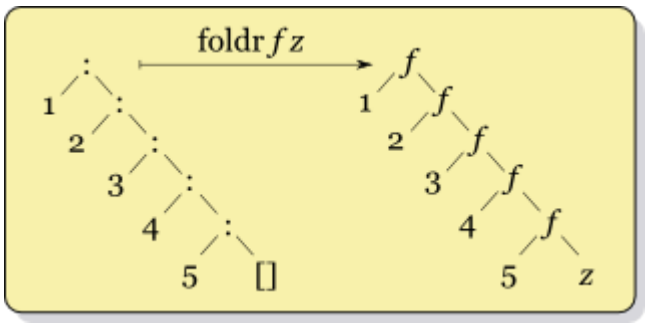
If we unroll `sum [1,2,3]` we see that the computation being performed is:

```
sum [1,2,3]
= foldr (+) 0 [1,2,3]
= 1 + (foldr (+) 0 [2,3])
= 1 + (2 + (foldr (+) 0 [3]))
= 1 + (2 + (3 + (foldr (+) 0 [])))
= 1 + (2 + (3 + (0)))
= 6
```

Higher-order functions (cont.)

`foldr`

A natural way to view folds is as a mechanism for replacing the structural components of a data structure with other functions and some value. Recall that `:` append an element to a list and that the empty list is represented by `[]`. We will learn in the next lecture all about data constructors, but for now, you can think that writting `[1,2,3]` is equivalent to writting `1 : (2 : (3 : []))`. Visually, this is how the `foldr` transformation looks like:

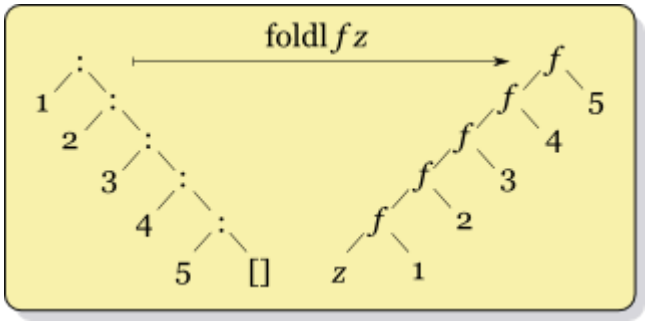


Source: Haskell Wiki

Higher-order functions (cont.)

`foldl`

Perhaps less intuitive, there's also a left fold `foldl` that works in a similar way to `foldr`, only the accumulator eats up the values from the *left*. Visually, this amounts to the following transformation:



Source: Haskell Wiki

Higher-order functions (cont.)

`foldl`

As with the right fold, `foldl` can also be defined resursively.

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl _ z [] = z
foldl f z (x:xs) = foldl f (z `f` x) xs
```

The left fold diagram suggests an easy way to reverse a list:

```
reverse' xs = foldl (flip (:)) [] xs
```

Here, `flip` is a function that takes another function as input (in our case the function `(:)`) and simply switches the order of the arguments.

```
:t flip
flip :: (a -> b -> c) -> b -> a -> c
```

Higher-order functions (cont.)

`foldl`

If we unroll `reverse' [1,2,3]` we get the following steps:

```
reverse' [1,2,3]
= foldl (flip (:)) [] [1,2,3]
= foldl (flip (:)) ([] `flip (:)` 1) [2,3]
= foldl (flip (:)) [1] [2,3]
= foldl (flip (:)) ([1] `flip (:)` 2) [3]
= foldl (flip (:)) [2,1] [3]
= foldl (flip (:)) ([2,1] `flip (:)` 3) []
= foldl (flip (:)) [3,2,1] []
= [3,2,1]
```

Lambda functions

Sometimes it is convenient to define a function locally, on the spot. To do so, we can take advantage of *lambda functions* (sometimes also called *anonymous functions*).

Syntax

```
\arguments -> function body
```

Example

Perhaps the previous example was not totally clear. Let's rewrite it with a lambda function:

```
reverse'' xs = foldl (\zs x -> x : zs) [] xs

--func zs x = x : zs

reverse'' [1,2,3]
[3,2,1]
```

Function application with `$`

The dollar sign `$` is just an operator to replace parentheses and render the code more intelligible!

It's a syntatic trick to write functions like

```
sumsqreven ns = sum (map (^2) (filter even ns))
```

as

```
sumsqreven ns = sum $ map (^2) $ filter even ns
```

Under the hood, it's a little more than that. It's a higher-order function with the lowest precendence and right associative.

```
infixr 0
($) :: (a -> b) -> a -> b
f $ a = f a
```

Meaning that everything on the right side of `$` is executed first and applied as one final argument.

Pointfree style

It is very common for functional programmers to write functions as a composition of other functions, never mentioning the actual arguments they will be applied to. For example, compare:

```
sum = foldr (+) 0
```

with:

```
sum' xs = foldr (+) 0 xs
```

These functions perform the same operation, however, the former is more compact and cleaner.

Function composition `(.)`

In mathematics, the compoistion of two functions, say `f` and `g` results in a new function `(f . g)(x) = f(g(x))`.

In Haskell, function composition is done in the same way with operator `(.)`, which is simply a higher-order function that takes 2 functions as input and outputs another function combining both.

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
f . g = \x -> f (g x)
```

Function composition can be used to simplify nested function application, by reducing parentheses and avoiding the need to explicitely refer to the argument when combined with a pointfree style.

Function composition `(.)` (cont.)

Example

```
odd n = not (even n)

twice f x = f (f x)

sumsqreven ns = sum (map (^2) (filter even ns))
```

The above functions could be rewritten as

```
odd = not . even

twice f = f . f

sumsqreven = sum . map (^2) . filter even
```

Which versions are easier to read?

Exercises 🧩

(1) Recall the definition of `length` we've seen in lecture #3

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

Redefine the function using `foldr` and a lambda function.

(2) Define `maxList` as a foldr

```
maxList :: Ord a => [a] -> a
maxList = undefined
```

(3) Define `flattenr` as a foldr and `flattenl` as a foldl

```
flattenr :: [[a]] -> [a]
flattenr = undefined

flattenl :: [[a]] -> [a]
flattenl = undefined
```

In [ ]:

```
:opt lint
length2 xs = foldr (\x acc -> acc + 1) 0 xs

length2 [1,2,3]

maxList :: Ord a => [a] -> a
maxList (x:xs) = foldr (\y acc -> max y acc) x xs

maxList [1,2,3,7,5]

flattenr :: [[a]] -> [a]
flattenr xs = foldr (\x acc -> x ++ acc) [] xs

flattenr2 :: [[a]] -> [a]
flattenr2 xs = foldr (\y acc -> (++) y acc) [] xs

flattenr3 :: [[a]] -> [a]
flattenr3 xs = foldr (++) [] xs

flattenr4 :: [[a]] -> [a]
flattenr4 = foldr (++) []

flattenr [[1,2],[3],[4,5]]
flattenr2 [[1,2],[3],[4,5]]
flattenr3 [[1,2],[3],[4,5]]

flattenl :: [[a]] -> [a]
flattenl xs = foldl (\acc y -> acc ++ y) [] xs

flattenl2 :: [[a]] -> [a]
flattenl2 = foldl (++) []

flattenl [[1,2],[3],[4,5]]
flattenl2 [[1,2],[3],[4,5]]
```

```
7
[1,2,3,4,5]
[1,2,3,4,5]
[1,2,3,4,5]
[1,2,3,4,5]
[1,2,3,4,5]
```

Exercises 🧩 (cont.)

(4) Concat me like a fold

The `concat` function on lists has the following type:

```
concat :: [[a]] -> [a]
```

Simply put, it flattens a list:

```
concat [[1, 2], [3, 4], [], [5]] == [1, 2, 3, 4, 5]
```

Write a function `foldConcat`, equivalent to `concat`, with two `foldr`:

```
foldConcat :: [[a]] -> [a]
foldConcat = foldr (f1 (foldr f2)) acc
    where f1 = undefined
          f2 = undefined
          acc = undefined
```

If you want a more challenging variation, try to solve Concat me like a fold 💪💪💪

In [ ]:
```haskell
foldConcat :: [[a]] -> [a]
foldConcat xs = foldr (f1 (foldr f2)) acc xs
    where f1 = undefined
          f2 = undefined
          acc :: [a]
          acc = undefined

foldConcatV1 :: [[a]] -> [a]
foldConcatV1 xs = foldr (\x acc -> x ++ acc) [] xs

foldConcatV2 :: [[a]] -> [a]
foldConcatV2 xs = foldr (++) [] xs

foldConcatV3 :: [[a]] -> [a]
foldConcatV3 xs = foldr (+++) [] xs
    where (+++) :: [a] -> [a] -> [a]
          (+++) xs ys = foldr (\x acc -> x : acc) ys xs

foldConcatV4 :: [[a]] -> [a]
foldConcatV4 xs = foldr (+++) [] xs
    where (+++) :: [a] -> [a] -> [a]
          (+++) xs ys = foldr (:) ys xs

foldConcatV5 :: [[a]] -> [a]
foldConcatV5 xs = foldr (+++) [] xs
    where (+++) :: [a] -> [a] -> [a]
```

```
          (+++) xs ys = flip (foldr (:)) xs ys

foldConcatV6 :: [[a]] -> [a]
foldConcatV6 xs = foldr (+++) [] xs
    where (+++) :: [a] -> [a] -> [a]
          (+++) = flip (foldr (:))

foldConcatV7 :: [[a]] -> [a]
foldConcatV7 = foldr (f1 (foldr f2)) acc
    where f1 = flip
          f2 = (:)
          acc = []
```

Exercises 🧩 (cont.)

(5) Define `foldr` and `foldl` functions in Python.

Yes, I really meant Python! 🐍🙃 Then, write `reverse` in terms of `foldl`.

1. Python has a `foldl` function:

   ```
   import functools

   def foldl(func, acc, xs):
     return functools.reduce(func, xs, acc)
   ```

2. Notice that `foldr` is very similar to `foldl`:

   ```
   foldl :: (acc -> elem -> acc) -> acc -> [elem] -> acc
   foldr :: (elem -> acc -> acc) -> acc -> [elem] -> acc
   ```

3. We can define `foldr` in terms of `foldl` by reversing the list and flipping the arguments of the function:

   ```
   foldr' func acc xs = foldl (\acc elem -> func elem acc) (reverse
   xs)

   foldr'' func acc xs = foldl (flip func) (reverse xs)
   ```

4. Now, we can apply the same strategy in Python:

   ```
   import functools
   import operator

   def foldl(func, acc, xs):
     return functools.reduce(func, xs, acc)

   def flip(func):
       @functools.wraps(func)
       def newfunc(x, y):
           return func(y, x)
       return newfunc
   ```

```
    def foldr(func, acc, xs):
        return functools.reduce(flip(func), reversed(xs), acc)

    # tests
    print(foldl(operator.sub, 0, [1,2,3])) # -6
    print(foldl(operator.add, 'L', ['1','2','3'])) # 'L123'
    print(foldr(operator.sub, 0, [1,2,3])) # 2
    print(foldr(operator.add, 'R', ['1','2','3'])) # '123R'
```

Credits: André Burgaud