Welcome to Programming Fundamentals 3

University of Luxembourg

2025/2026

Bachelor in Computer Sciences (BICS)
Semester 3

Lecture - Week #5

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

Important: Please remember to register your attendance in Moodle!

Recap of Lecture #4

- Curried functions
- Partially applied functions
- Higher-order functions
  - `map`
  - `filter`
  - `takeWhile`, `dropWhile`
  - `all`, `any`
  - `foldr`, `foldl`
- Lambda functions
- Function application with `$`
- Pointfree style
- Function composition

Types and Typeclasses

Recall from Lecture #2

Typeclass	Description	Instances
Show	Members of Show can be presented as strings.	All Prelude types.
Read	The Read typeclass is essentially the opposite of Show: it defines functions that will take a String, parse it, and return data in any type that is a member of Read.	All Prelude types.
Enum	Defines operations on sequentially ordered types.	Integer, Int, Char, Bool, Float, Double
Bounded	Bounded members have an upper and a lower bound.	Int, Char, Bool

Typeclass	Description	Instances
Num	Numeric members.	Integer, Int, Float, Double
Integral	Integral includes only integral (whole) numbers.	Int, Integer
Floating	Floating includes only floating point numbers.	Float, Double
Eq	The Eq class provides equality (==) and inequality (/=) methods.	All Prelude types.

Types and Typeclasses (cont.)

If we don't remember which types are instances of a particular typeclass, or which methods a particular typeclass provides, we can ask GHCI:

In [3]:
```
:opt no-pager
```

In [1]:
```
(minBound :: Int)
(maxBound :: Int)
(maxBound :: Word)
(minBound :: Word)

--:info Ord
```

```
-9223372036854775808
9223372036854775807
18446744073709551615
0
```

Types and Typeclasses (cont.)

- In Lecture #2, we covered many types and typeclasses predefined in Haskell Prelude.
- In this lecture, we will introduce mechanisms for declaring new types.
- New types can take various forms, and can even be defined recursively.
- We will learn how to declare types as instances of predefined typeclasses, automatically and by hand.
- We will learn how to create our own classes.

Types and Typeclasses (cont.)

- There are 3 approaches to declare new data types in Haskell.

1. `type` declarations

- The simplest way of declaring a new type is to introduce a new name for an existing type.
- The `type` mechanism in Haskell can be used to define *synonyms*.
- A notable example you've already encountered is the type `String`, which is equivalent to `[Char]`.

```
type String = [Char]
```

Types and Typeclasses (cont.)

- Let's look at a few examples:

```
type Coordinate = (Int,Int)

type Transform = Coordinate -> Coordinate
```

- New type names must start with a capital letter.
- Type declaration can even be applied to functions.

Types and Typeclasses (cont.)

- However, `type` declarations cannot be recursive. The following example is **invalid**:

```
type Tree = (Int,[Tree])
```

- If needed, recursive types can be declared with `data`. (We will see this in a moment.)

Types and Typeclasses (cont.)

- Type declaration can also be parameterized by other types:

In [6]:
```
type Coordinate = (Int,Int)

type PairInt = (Int,Int)
type Pair a = (a,a)

x = ('a','b') :: Pair Char
s = ("Hello","World") :: Pair String

x
s
:type x
:type s
```

```
('a','b')
("Hello","World")
x :: Pair Char

s :: Pair String
```

Types and Typeclasses (cont.)

- Type declaration also admit multiple parameters.
- For example, lookup tables associate keys of one type with values of possibly another type.

```
type Assoc k v = [(k,v)]

find :: Eq k => k -> Assoc k v -> v
find k lookupTable = head [v | (k', v) <- lookupTable, k == k']
```

Types and Typeclasses (cont.)

2. `data` declarations

- New data types that are not synonyms of existing data types can be declared using the `data` mechanism.
- For example, the type `Bool` to represent boolean values `True` or `False` is declared in Haskell Prelude as follows:

```
data Bool = False | True
```

- In such declaration, the symbol `|` is read as *or* and the new values after the `=` are *value constructors* (in this case `True` and `False`).
- Like new types, value construtor names begin with capital letters.
- Constructor names must be unique.
- Just like the `type` mechanism, the `data` mechanism also admits parameters.

Examples

- Let's look at a few examples

```
type Coordinate = (Int,Int)
data Direction = North | South | East | West

moveOne :: Direction -> Coordinate -> Coordinate
moveOne North (x,y) = (x,y+1)
moveOne South (x,y) = (x,y-1)
moveOne East (x,y) = (x+1,y)
moveOne West (x,y) = (x-1,y)
```

- Let's give it a try...

In [7]:
```
data Shape = Circle Float          -- radius
           | Rectangle Float Float -- height and width
    deriving (Show, Eq, Ord)

area :: Shape -> Float
area (Circle r)      = pi*r^2
area (Rectangle h w) = h*w

area (Circle 2)

Circle 2
:type Circle
:type Rectangle
```

```
Circle 2 == Circle 3
Circle 2 > Circle 1
```

```
12.566371
Circle 2.0
Circle :: Float -> Shape

Rectangle :: Float -> Float -> Shape

False
True
```

- Notice that value constructors are functions.
- This is why value constructors must be unique.

In [9]:
```
data Shape = Circle Float
           | Square Float

data Rectangle = Unequal Int Int
               | Square Int
```

```
<interactive>:4:18: error:
    Multiple declarations of ‘Square’
    Declared at: <interactive>:2:14
                 <interactive>:4:18
```

- Technically, this means we can map them or partially apply them.

```
map Circle [1.0,2.0,3.5,7]
[Circle 1.0,Circle 2.0,Circle 3.5,Circle 7.0]

map (Rectangle 1.0) [1.0,2.0,3.5,7]
[Rectangle 1.0 1.0,Rectangle 1.0 2.0,Rectangle 1.0 3.5,Rectangle
1.0 7.0]
```

- When a new data type is declared, it's not part of any typeclass.
- Therefore, we cannot even do simple things like printing or comparing values.
- When the keyword `deriving (Show, Eq)` is used after the `data` type declaration, Haskell automatically makes that type part of the `Show` and `Eq` typeclass.

Maybe

The Maybe type is defined as follows:

```
data Maybe a = Nothing | Just a
    deriving (Eq, Ord, Show)
```

This type is useful for operations that may fail. Simple examples including division by `0` or retrieving the head of an empty list.

```
div 5 0
*** Exception: divide by zero
```

```
head []
*** Exception: Prelude.head: empty list
```

We can use the Maybe type to define safe versions of these functions that do not fail.

```
safeDiv :: Integral a => a -> a -> Maybe a
safeDiv _ 0 = Nothing
safeDiv n m = Just (n `div` m)

safeHead :: [a] -> Maybe a
safeHead [] = Nothing
safeHead (x:xs) = Just x
```

We will see later on how to compose multiple such functions that may fail gracefully.

Alternative syntax

```
data Person = Person String String Int Float String deriving
(Show)

let guy = Person "Bob" "G." 2003 184.2 "+352 601234567"
```

The above syntax is kind of unreadable. We could use pattern-matching to define functions to access data of a Person, which helps understanding how to use the `Person` constructor:

```
firstname (Person x _ _ _ _) = x
lastname (Person _ x _ _ _) = x
year (Person _ _ x _ _) = x
```

But this is really tedeous and not so elegant.

Alternatively, we could write:

In [8]:
```
data Person = Person { firstName :: String
                     , lastName :: String
                     , year :: Int
                     , height :: Float
                     , phoneNumber :: Maybe String
                     } deriving (Show)

let girl = Person { firstName="Anne", lastName="F.", year=2002, height=170.5, p

firstName girl
```

```
"Anne"
```

Recursive types

- Types may also be recursive.
- Take the trivial example of a data type `Nat` for all natural numbers including `0`. It could be defined as follow.

```
data Nat = Zero | Succ Nat
```

- A value of type `Nat` is either `Zero` or of the form `Succ n` for some value `n` of type `Nat`.
- By successive applications of the constructor `Succ`, this declaration gives rise to an infinite sequence of values starting with `Zero`.

```
Zero
Succ Zero
Succ (Succ Zero)
Succ (Succ (Succ Zero))
...
```

The Peano axioms, named after 19th century Italian mathematician Giuseppe Peano, define the arithmetical properties of natural numbers.

The five Peano axioms are:

1. Zero is a natural number.
2. Every natural number has a successor in the natural numbers.
3. Zero is not the successor of any natural number.
4. If the successor of two natural numbers is the same, then the two original numbers are the same.
5. If a set contains zero and the successor of every number is in the set, then the set contains the natural numbers.

The fifth axiom is known as the principle of induction because it can be used to establish properties for an infinite number of cases without having to give an infinite number of proofs.

In particular, given that P is a property and zero has P and that whenever a natural number has P its successor also has P, it follows that all natural numbers have P.

We could define conversion functions between types `Int` and `Nat`.

In [9]:
```
:opt no-lint

data Nat = Zero | Succ Nat
    deriving Show

nat2int :: Nat -> Int
nat2int Zero = 0
nat2int (Succ x) = 1 + nat2int x

--nat2int (Succ (Succ (Zero))) = 1 + nat2int (Succ (Zero)) = 1 + 1 + nat2int Ze

int2nat :: Int -> Nat
int2nat 0 = Zero
int2nat n | n >= 0 = Succ $ int2nat (n-1)
          | otherwise = error "Negative numbers not allowed."

(nat2int . int2nat) 3
```

```
3
```

Recursive type List

- We could also use the `data` mechanism to declare the built-in of lists.

  ```
  data List a = Nil | Cons a (List a)
      deriving Show
  ```

- A list is either empty, represented by constructor `Nil`, or it has a head with an element of type `a`, representented by `Cons a` and a tail, which is another list.

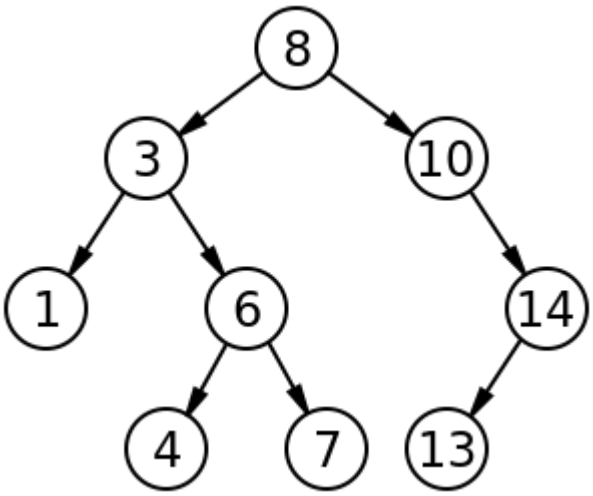
  ```
  (+++) :: List a -> List a -> List a
  (+++) Nil y = y
  (+++) (Cons x xs) y = Cons x (xs +++ y)
  ```

- Perhaps the type `List` becomes more familiar with an infix value constructor:

  ```
  data List a = Nil | a :-: (List a)
      deriving Show
  ```

Recursive type Tree

- Data type `tree`



In [19]:
```haskell
data Tree a = Empty | Branch a (Tree a) (Tree a) deriving Show

t = Branch 8 (Branch 3 (Branch 1 Empty Empty) (Branch 6 (Branch 4 Empty Empty)

occurs :: Eq a => a -> Tree a -> Bool
occurs _ Empty = False
occurs n (Branch x l r) | x == n = True
                        | otherwise = occurs n l || occurs n r
```

- Note that as in nature, trees come in different forms. Here are a few other examples, equally valid, depending on the application:

```haskell
-- nodes are not labelled
data Tree a = Leaf a | Node (Tree a) (Tree a)

-- leafs are not labelled
data Tree a = Leaf | Node (Tree a) a (Tree a)

-- leafs and nodes may the labelled with different types
data Tree a b = Leaf a | Node (Tree a b) b (Tree a b)

-- trees may have many branches
data Tree a = Node a [Tree a]
```

`newtype` declarations

- If the new type has a single constructor with a single argument, then it could also be declared with `newtype` mechanism.

```haskell
newtype Nat = N Int
```

- One might wonder what is the difference between new types declared with `newtype` mechanism compared to `type` and `data`.

`newtype` vs. `type`

```
newtype Nat = N Int
type Nat = Int
```

- `type` is for synonyms, in which case requiring type `Nat` or type `Int` is the same.
- `newtype` declares a new type. In the above example, types `Nat` and `Int` are not interchangeable. `Nat` also has a value constructor `N`.

`newtype` vs. `data`

```
newtype Nat = N Int
data Nat = N Int
```

- Whatever we can declare with `newtype` we could also declare with `data` mechanism.
- The reverse is not true has `newtype` is limited to new types with single value constructors and single arguments.
- `newtype` is prefered whenever possible for efficiency reasons. Value constructors of `newtype` are removed by the compiler after type checking.

Make types instances of typeclasses

- Only for new types declared via `data` and `newtype`.
- By default, a new type has no methods associated with it.
- If we try to compare two `Circle`, it simply won't work out-of-the-box:

In [14]:
```
data Shape = Circle Float
           | Square Float

let x = Circle 2.0
let y = Circle 3.0

x == y
```

```
<interactive>:1:3: error: [GHC-39999]
    • No instance for ‘Eq Shape’ arising from a use of ‘==’
      There are instances for similar types:
        instance [safe] Eq Ghci346.Shape -- Defined at <interactive>:3:19
        instance [safe] Eq Ghci260.Shape -- Defined at <interactive>:3:21
    • In the expression: x == y
      In an equation for ‘it’: it = x == y
```

Technically, we can't even print a value of a type that is not an instance of class `Show`.

In [15]:
```
data Shape = Circle Float
           | Square Float

Square 1.0
```

```
<interactive>:1:1: error: [GHC-39999]
    • No instance for ‘Show Shape’ arising from a use of ‘print’
      There are instances for similar types:
        instance [safe] Show Ghci346.Shape -- Defined at <interactive>:3:13
        instance [safe] Show Ghci260.Shape -- Defined at <interactive>:3:15
    • In a stmt of an interactive GHCi command: print it
```

Derived instances

We can, however, ask the Haskell compiler to make our new type an instance of a certain typeclass, automatically, by using the keyword `deriving`.

For example, we could declare the new type `Shape` and derive instances of classes `Show` and `Eq`. This allows to check syntatic equality and print types *as is*.

In [30]:
```haskell
data Shape = Circle Float
           | Square Float
           deriving (Show, Eq)

Square 1.0

Circle 2.0 == Circle 3.0
```

```
Square 1.0
False
```

How to make types instances of classes by hand

Types can also be made instances of classes manually, by explicitly defining functions that make a particular type an instance of that class.

Example:

In [14]:
```haskell
data TrafficLight = Red | Yellow | Green

instance Eq TrafficLight where
    Red == Red = True
    Green == Green = True
    Yellow == Yellow = True
    _ == _ = False

instance Show TrafficLight where
    show Red = "Red light"
    show Yellow = "Yellow light"
    show Green = "Green light"

Green
```

```
Green light
```

Creating our own typeclasses

Let's define a `Measurable` typeclass that provides a way to calculate the size of a data type.

```
class Measurable a where
    size :: a -> Int
```

The size function takes a value of type a and returns an integer representing its size. To make a type an instance of Measurable, you need to define an implementation for size:

```
instance Measurable Int where
    size _ = 8

instance Measurable Char where
    size _ = 1

instance Measurable a => Measurable [a] where
    size xs = sum [size x | x <- xs]
```

This declares `Int`, `Char`, and `[a]` (where `a` is an instance of `Measurable`) as instances of the Measurable typeclass.

Types and Typeclasses (cont.)

Recap

- Typeclasses are like interfaces. A typeclass defines some behavior (like comparing for equality, comparing for ordering, enumeration). Types that can behave in that way are made instances of that typeclass.
- The behavior of typeclasses is achieved by defining functions or just type declarations that we then implement.
- When we say that a type is an instance of a typeclass, we mean that we can use the functions that the typeclass defines with that type.

Exercises 🧩

(1) Say Hi - Person type

Write a function sayHi which will take a Person record as input and will return a string:

"Hi, i'am {firstName} {lastName} and it is nice to meet You."

You will need to define the record type `Person`, with fields `firstName` and `lastName`.

https://www.codewars.com/kata/57a852c353ba334961001480

In [10]:
```
data Person = Person { firstName :: String, lastName :: String }

sayHi :: Person -> String
sayHi p = "Hi, i'am " ++ (firstName p) ++ " " ++ (lastName p) ++ " and it is ni

let you = Person { firstName = "Cristiano", lastName = "R."}

sayHi you
```

```
"Hi, i'am Cristiano R. and it is nice to meet You."
```

(2) Binary Tree Search (not BST)

Given a number and a binary tree (not a Binary Search Tree!):

- return True if the given number is in the tree
- return False if it isn't

https://www.codewars.com/kata/5acc79efc6fde7838a0000a0

```
data Tree a = Nil | Node (Tree a) a (Tree a) deriving (Show)
```

In [11]:
```haskell
{-
-- Sample Tests has incorrect indentation, but attempt work.
-- Fix indentation to Test:

main = hspec spec
spec = do
  describe "The search function" $ do
    it "should work for some examples" $ do
      search 1 Nil `shouldBe` False
      search 2 (Node Nil 2 Nil) `shouldBe` True
      search 2 (Node Nil 1 Nil) `shouldBe` False
-}

data Tree a = Nil | Node (Tree a) a (Tree a) deriving (Show)

search :: Int -> Tree Int -> Bool
search x Nil = False
search x (Node left y right) | x == y    = True
                             | otherwise = search x left || search x right

t1 = Node Nil 3 (Node Nil 4 Nil)

search 4 t1
```

```
True
```

(3) Binary Tree Compare

Write a function compare(a, b) which compares the two trees defined by Nodes a and b and returns true if they are equal in structure and in value and false otherwise.

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

https://www.codewars.com/kata/55847fcd3e7dadc9f800005f

In [ ]:
```haskell
-- NOT COMPLETED IN CLASS

import Prelude hiding (compare) -- hide Prelude function `compare` to avoid amb

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


compare :: Maybe Tree -> Maybe Tree -> Bool
compare Nothing Nothing = True
compare Nothing _       = False
compare _       Nothing = False
compare (Just t1) (Just t2)
```

```
    | val t1 /= val t2  = False
    | otherwise        = compare (left t1) (left t2) && compare (right t1) (ri
```

(4) Data Structure #0 True or False

Implement functions for data type Bool:

- Boolean and;
- Boolean or;
- Boolean not;
- To make guards more readable, `otherwise` is the same as `True`.

https://www.codewars.com/kata/5bba2a9940196db222000022

In [ ]:

```haskell
-- NOT COMPLETED IN CLASS

import Prelude hiding (Bool (..), (&&), (||), not, otherwise)

--import TrueOrFalse.Prelude (Bool (..))
data Bool = True | False deriving (Eq, Show)

-- Boolean and
infixr 3 &&
(&&) :: Bool -> Bool -> Bool
(&&) True True = True
(&&) _ _ = False

-- Boolean or
infixr 2 ||
(||) :: Bool -> Bool -> Bool
(||) False False = False
(||) _ _ = True

-- Boolean not
not :: Bool -> Bool
not True = False
not False = True

-- otherwise :: Bool
otherwise :: Bool
otherwise = True

-- bool x y p evaluates to x when p is False, and evaluates to y when p is True
bool :: a -> a -> Bool -> a
bool _ y True = y
bool x _ False = x
```