Welcome to Programming Fundamentals 3

University of Luxembourg

2025/2026

Bachelor in Computer Sciences (BICS)
Semester 3

Lecture - Week #2

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

Important: Please remember to register your attendance in Moodle!

Recap of Week #1

Why functional programming in Haskell

- How to install GHCI and VS Code with Haskell language support
- Introduction to Haskell
  - Basic syntax
  - Functions
  - Precedence
  - Lists
  - List comprehension
  - Tuples
  - Pattern matching

GHCI Commands

command	shortcut	description
:load file	:l file	load a Haskell file
:reload	:r	reload changes
:edit	:e	edit current file
:set editor		set editor, e.g. `vim`
:type expression	:t expression	show the type of expression
:help	:h	show help menu
:quit	:q	quit GHCI

File extensions

The typical file extension for Haskell source code files is .hs or .lhs, for literate Haskell.

Source code files for a Haskell program are typically text files that contain one or more definitions of functions, data types, and other expressions written in the Haskell language. The files that contains these definitions can be named with the `.hs` or `.lhs` extension, and the common practice is to use `.hs` for normal source code and `.lhs` for literate Haskell source code.

Literate Haskell is a way of writing Haskell programs where the source code is interleaved with documentation written in plain text. This can make it more readable, especially for complex programs, because it allows you to include explanations of what the code does and why it is written the way it is.

If-then-else

If-then-else statements...

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

In [17]:
```
-- a few examples (1)

max 3.0 4.4

--max' :: Int -> Int -> Int
max' x y = if x >= y then x else y

isEven x = if x `mod` 2 == 0 then "Even" else "Odd"

ageGroup age = if age < 18 then "Junior"
               else if age < 65 then "Adult"
                    else "Senior"

ageGroup' age | age < 18 = "Junior"
              | age < 65 = "Adult"
              | otherwise = "Senior"

isEven 4
max' 3.0 4.4
ageGroup 19
```

```
4.4
"Even"
4.4
"Adult"
```

Guards

- As an alternative to using conditional expressions, functions can also be defined using guarded equations, in which a sequence of logical expressions called guards is used to choose between a sequence of results of the same type.
- When you have several conditions, guarded expressions make the code more readable.
- The previous `ageGroup` function could be written with guards as follows:

```
ageGroup age | age < 18 = "Junior"
             | age < 65 = "Adult"
             | otherwise = "Senior"
```

- Whereas patterns are a way of making sure a value conforms to some form and deconstructing it, guards are a way of testing whether some property is `True` or `False`.
- Conditions are tested from top to bottom.
- It is more common to use guards than "if-then-else" conditions, whenever possible.

- Another example:

```
abs n | n >= 0 = n
      | otherwise = -n
```

Case expressions

- Similar to `switch` in C and Java, in Haskell you might use case expressions.
- A case expressions allows you to match a value against several possible patterns and return a different result for each pattern.
- As in pattern matching, `_` is a catch-all pattern that matches any value not matched by the previous patterns.

Example

```
charToDigit :: Char -> Int
charToDigit c =
    case c of
        '0' -> 0
        '1' -> 1
        '2' -> 2
        '3' -> 3
        '4' -> 4
        '5' -> 5
        '6' -> 6
        '7' -> 7
        '8' -> 8
        '9' -> 9
        _   -> -1 -- character is not a digit
```

Comments

- Single line `-- comment`
- Multiple lines

```
{-
This function is commented out
mySum :: [Int] -> Int
```

```
mySum [] = 0
mySum (x:xs) = x + sum xs
-}
```

Infinite lists

- In Haskell, a list does not necessarily have a finite number of elements.
- It can be defined in a number of ways, such as:
  - by using recursion, where the list is defined in terms of itself;
  - by using the `iterate` function, which applies a function repeatedly to a seed value to generate an infinite list;
  - by using the `cycle` function, which repeats a finite list indefinitely;
  - by using ranges.
- Haskell is "lazy", meaning that expressions are only evaluated when needed. Therefore, we easily play with infinite lists.

In [3]:
```haskell
-- a few examples (2)

ones = 1 : ones

take 100 $ ones

powersOfTwo = iterate (*2) 1

take 10 $ powersOfTwo

:type iterate

iterate' f v = v : iterate' f (f v)

powersOfTwo !! 8

weekWorkingDays = cycle ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday"

take 52 $ weekWorkingDays

cycle' xs = xs ++ cycle' xs

zip ['a','b'] [1,2,3]

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

-- example from haskell.org homepage
primes = filterPrime [2..]
  where filterPrime (p:xs) =
          p : filterPrime [x | x <- xs, x `mod` p /= 0]

take 3 primes
```

```
[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]
```

```
[1,2,4,8,16,32,64,128,256,512]
iterate :: forall a. (a -> a) -> a -> [a]
256
["Monday","Tuesday","Wednesday","Thursday","Friday","Monday","Tuesday","Wednesda
y","Thursday","Friday","Monday","Tuesday","Wednesday","Thursday","Friday","Monda
y","Tuesday","Wednesday","Thursday","Friday","Monday","Tuesday","Wednesday","Thu
rsday","Friday","Monday","Tuesday","Wednesday","Thursday","Friday","Monday","Tue
sday","Wednesday","Thursday","Friday","Monday","Tuesday","Wednesday","Thursda
y","Friday","Monday","Tuesday","Wednesday","Thursday","Friday","Monday","Tuesda
y","Wednesday","Thursday","Friday","Monday","Tuesday"]
[('a',1),('b',2)]
[2,3,5]
```

Question from previous editions of this course ❓

Question: What are the steps in the computation of list `primes`?

Answer:

```
primes = filterPrime [2..] where
  filterPrime (p:xs) =
    p : filterPrime [x | x <- xs, x `mod` p /= 0]

take 3 primes
= take 3 filterPrime [2..]
= take 3 filterPrime (2:[3..])
= take 3 (2 : filterPrime [x | x <- [3..], x `mod` 2 /= 0])
= take 3 (2 : 3 : filterPrime [x | x <- [5..], x `mod` 2 /= 0, x
`mod` 3 /= 0])
= take 3 (2 : 3 : 5 : filterPrime [x | x <- [7..], x `mod` 2 /=
0, x `mod` 3 /= 0, x `mod` 5 /= 0])

{At this point the list already has 3 elements and Haskell is
lazy.}

= [2,3,5]
```

Some new functions we've seen

- `take` -- `take n`, applied to a list `xs`, returns the prefix of `xs` of length `n`, or `xs` itself if `n >= length xs`.
- `takeWhile` -- `takeWhile`, applied to a predicate `p` and a list `xs`, returns the longest prefix (possibly empty) of `xs` of elements that satisfy `p`.
- `iterate` -- `iterate f x` returns an infinite list of repeated applications of `f` to `x`.
- `cycle` -- cycle ties a finite list into a circular one, or equivalently, the infinite repetition of the original list.
- `zip` -- `zip` takes two lists and returns a list of corresponding pairs.

Exercise #1: Fibonacci sequence 📝

In mathematics, the Fibonacci numbers, commonly denoted Fn, form a sequence, the Fibonacci sequence, in which each number is the sum of the two preceding ones. The sequence commonly starts from 1 and 2.

1,2,3,5,8,13,21,...

Exercise:

- Write a recursive function `fib` to compute the n-th element of the Fibonacci sequence. What is `fib 50`?
- Create an infinite Fibonacci sequence and take advantage of lazy evaluation. Then create a function that does the same as the above. What is `fib 50`?

In [6]:
```
-- fib (3)
-- recursive, inefficient
fib :: Int -> Int
fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2)

fib 10

-- fast
-- create an infinite Fibonacci sequence and take advantage of lazy evaluation
fibs = gen 1 1
  where
    gen x y = x : gen y (x+y)

take 50 $ fibs

-- ChatGPT
fibs' :: [Int]
fibs' = 1 : 2 : zipWith (+) fibs' (tail fibs')
```

```
89
[1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,1597,2584,4181,6765,10946,17711,
28657,46368,75025,121393,196418,317811,514229,832040,1346269,2178309,3524578,570
2887,9227465,14930352,24157817,39088169,63245986,102334155,165580141,267914296,4
33494437,701408733,1134903170,1836311903,2971215073,4807526976,7778742049,125862
69025]
```

Question from previous editions of this course ❓

Question: Why we could not compute recursively Fibonacci of `50`?

Answer: Each evaluation of the function expands recursively into 2 other evaluations, so the number of evaluations grows exponentially.

```
fib 1 = 1
fib 2 = 2
fib n = fib (n-1) + fib (n-2)
```

Types

A type is a kind of label that every expression has. It tells us in which category of things that expression fits. The expression `True` is a `Bool`, `"hello"` is a `String`, etc.

Some types in Haskell:

- Bool
- Char
- Int (bounded)
- Integer (unbounded)
- Float
- Double

In [1]:
```
:type 'a'
:t True
:t "HELLO!"
:t (True, 'a')
:t 4 == 5
:t head

head' ::
head' [] = error
head' (x:xs) = x
```

```
'a' :: Char

True :: Bool

"HELLO!" :: String

(True, 'a') :: (Bool, Char)

4 == 5 :: Bool

head :: forall a. [a] -> a
```

Type Errors

Some operations only make sense with values of a specific type.

Some examples:

In [9]:
```
7 + True
--max "Programming Fundamentals" 3
--odd False

:type (+)
```

```
<interactive>:1:1: error: [GHC-39999]
    • No instance for ‘Num Bool’ arising from the literal ‘7’
    • In the first argument of ‘(+)’, namely ‘7’
      In the expression: 7 + True
      In an equation for ‘it’: it = 7 + True
```

Has Type Of

- In GHCI, to get the type of an expression, we use `:type` or `:t`.
- Double colon `::` reads "has type of".
- Functions also have types.
- Types allow the **compiler** to catch type errors, which are mistakes that occur when you try to use values in inappropriate ways.
- Often, types can also be inferred.
- However, it's generally a good idea to first define types and allow the compiler to do type checking at compile time, i.e. **before the execution**.
- C language is notoriously weakly typed because any pointer type is convertible to any other pointer type simply by casting.

In [10]:
```
let a = 3 :: Float
let b = 4 :: Float

:t max
:t max a b
```

```
max :: forall a. Ord a => a -> a -> a
```

```
max a b :: Float
```

Polymorphism

- Some functions can operate on multiple types of input. They are called polymorphic functions.

Example:

```
head :: [a] -> a
head (x:_) = x
```

The type signature of this function `head :: [a] -> a` states that the function takes an argument of a list of any type a and returns a value of that type.

Here are some examples of how `head` can be used:

In [10]:
```
head [1, 2, 3]
head "Hello"
head ['H','e','l','l','o']
```

```
1
'H'
'H'
```

And some other examples of polymorphic functions:

In [11]:
```
:t take
:t fst
:t zip
```

```
take :: forall a. Int -> [a] -> [a]
```

```
fst :: forall a b. (a, b) -> a
zip :: forall a b. [a] -> [b] -> [(a, b)]
```

Typeclasses

- Have you noticed that some functions have some functions have some type class constraints?

Example:

```
:t (==)
(==) :: Eq a => a -> a -> Bool

:t (>)
(>) :: Ord a => a -> a -> Bool
```

- A typeclass is a sort of interface that defines some behavior.
- Everything before the `=>` symbol is a class constraint.
- The equality function `(==)` takes any two values that are of the same type and returns a Bool. The type of those two values must be an instance of the `Eq` class.
- `Ord` typeclass is for types that have an ordering.

In [37]:
```
3.0 < 7.2

"alpha" `compare` "beta"

--:info Ord
```

```
True
LT
```

Typeclasses (cont.)

Other common typeclasses

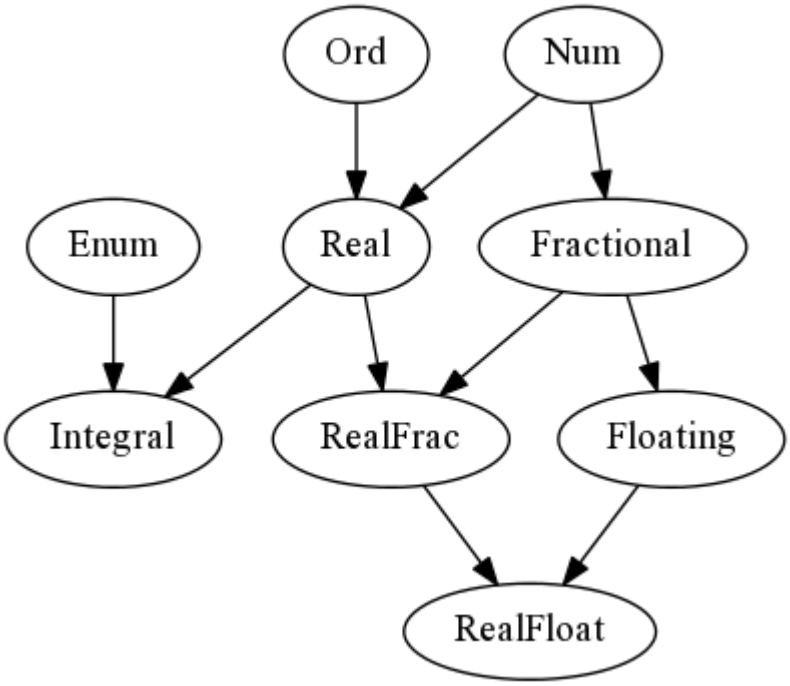
Typeclass	Description	Instances
Show	Members of Show can be presented as strings.	All Prelude types.
Read	The Read typeclass is essentially the opposite of Show: it defines functions that will take a String, parse it, and return data in any type that is a member of Read.	All Prelude types.
Enum	Defines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

minBound :: Int

In [12]:
```
(minBound :: Int)
(maxBound :: Int)

let b1 = abs (minBound :: Int)
let b2 = (maxBound :: Int)

2^64 == b1 + b2 + 1

(maxBound :: Char)

:t fromIntegral

let l = length [1,2,3,4]
fromIntegral l + 3.2
```

```
-9223372036854775808
9223372036854775807
True
'\1114111'
fromIntegral :: forall a b. (Integral a, Num b) => a -> b
7.2
```

Hierarchy of typeclasses



Source: Stephen Diehl

Exercise #2: Largest prime factor

The prime factors of 13195 are 5, 7, 13 and 29.

What is the largest prime factor of the number 600851475143 ?

Source: Project Euler, Problem #3

Optimize the solution below by looking for *candidates* prime factors of `n` only up to `sqrt(n)`.

Note: We haven't covered `filter` and lambda functions, but this should not prevent you from completing the exercise.

In [ ]:
```
factors :: Integral a => a -> [a]
factors n = filter (\x -> (n `mod` x) == 0) [1..n]

isPrime :: Integral a => a -> Bool
isPrime n = factors n == [1,n]

primeFactors :: Integral a => a -> [a]
primeFactors n = filter isPrime candidates
    where candidates = filter (\x -> (n `mod` x) == 0) [2..floor(sqrt(fromInteg
--  where candidates = filter (\x -> (n `mod` x) == 0) [2..n] -- it suffices to

primeFactors 13195
primeFactors 600851475143
```

```
[5,7,13,29]
[71,839,1471,6857]
```

We only need to look for prime factors of `n` up to the square root of `n` because, if `n` has a prime factor greater than the square root of `n`, then it must also have a corresponding prime factor less than or equal to the square root of `n`. For example, if `n` is a composite number and `ab=n` where `a` and `b` are prime numbers and `a>sqrt(n)` and `b>sqrt(n)` then `ab > sqrt(n)*sqrt(n)=n` which is a contradiction. Therefore, at least one of the factors of `n` must be less than or equal to the square root of `n`.

Local definitions

Where

- Local definitions cover all alternatives if the word `where` is indented like guards.

Example

```
bmiTell :: Float -> Float -> String
bmiTell weight height
    | bmi <= 18.5 = "Underweight"
    | bmi <= 25.0 = "Healthy weight"
    | bmi <= 30.0 = "Overweight"
    | otherwise   = "Obese"
    where bmi = weight / height ^ 2
```

(Categories by the WHO for adults over 20 years old.)

Local definitions (cont.)

Let ... in ...

- Similarly, we can also use keywords 'let' and 'in' to define local values.

Example

```
bmiCategory :: Float -> Float -> String
bmiCategory weight height =
    let bmi = weight / (height ^ 2)
        category
            | bmi <= 18.5 = "Underweight"
            | bmi <= 25.0 = "Healthy weight"
            | bmi <= 30.0 = "Overweight"
            | otherwise = "Obese"
    in category
```

(Categories by the WHO for adults over 20 years old.)

Reserved keywords in Haskell

```
case class data default deriving do else
if import in infix infixl infixr instance
let module newtype of then type where
```

Questions from the class ❓

Question: What are the steps in the computation of list `primes`?

Answer:

```
primes = filterPrime [2..] where
  filterPrime (p:xs) =
    p : filterPrime [x | x <- xs, x `mod` p /= 0]

take 3 primes
= take 3 filterPrime [2..]
= take 3 filterPrime (2:[3..])
= take 3 (2 : filterPrime [x | x <- [3..], x `mod` 2 /= 0])
= take 3 (2 : 3 : filterPrime [x | x <- [5..], x `mod` 2 /= 0, x
`mod` 3 /= 0])
= take 3 (2 : 3 : 5 : filterPrime [x | x <- [7..], x `mod` 2 /=
0, x `mod` 3 /= 0, x `mod` 5 /= 0])

{At this point the list already has 3 elements and Haskell is
lazy.}

= [2,3,5]
```

Question: Why we could not compute recursively Fibonacci of `50`?

Answer: Each evaluation of the function expands recursively into 2 other evaluations, so the number of evaluations grows exponentially.

```
fib 1 = 1
fib 2 = 2
fib n = fib (n-1) + fib (n-2)
```

Question: Why the signature of `sqrt` function is `sqrt :: Floating a => a -> a` and not `sqrt :: (Num a, Floating b) => a -> b`.

Answer: The precision of the input (`Float` vs `Double`) determines the precision of the output.

Function `fromIntegral` converts a value of type `a` which is constrained to be an instance of class `Integral` into a more general type `b` constrained only on class `Num`.

```
fromIntegral :: (Integral a, Num b) => a -> b
```

In the following example, `fromIntegral` takes an `Int` and converts it to a more general type that is instance of `Num`. Then, `sqrt` expects a type of class `Floating` and infers that the output of `fromIntegral` is an instance of class `Floating`.

```
sqrt $ fromIntegral (3 :: Int)
```

We can explicitely tell what is the type output by `fromIntegral` so that Haskell does not have to infer the types.

```
sqrt $ (fromIntegral (3 :: Int) :: Double)
```

By having the same input/output type, the precision of the input type determines the precision of the output.

```
sqrt :: Floating a => a -> a
```