Welcome to Programming Fundamentals 3

University of Luxembourg

2025/2026

Bachelor in Computer Sciences (BICS)
Semester 3

Lecture - Week #6

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

Important: Please remember to register your attendance in Moodle!

Recap of Lecture #5

- There are 3 mechanisms in Haskell to declare types:
  1. `type` for type synonyms (cannot be recursive);
  2. `data` for new type delcarations, possibly recursive and with multiple constructors;
  3. `newtype` for new types with a single constructor and single argument.
- Make type instances of typeclasses by either:
  1. Allowing the compiler to derive an instance of a particular typeclass;
  2. Declaring a type to be an instance of a typeclass by explicitly defining the required functions.
- Make our own typeclasses by defining the 'behavior' of types in that typeclass (i.e. which functions are available and what is their type signature).

Imperative programming approach:

```
procedure() {
    subroutine1();
    subroutine2();
    subroutine3();
}
```

The computer gets a list of commands and executes them in order. We essentially tell the computer what to do and when to do it.

Functional programming approach:

```
f x = f3 (f2 (f1 x))
```

Programming through composition of pure functions, with no side affects.

In a pure world, there's no side-effects.

- In a pure world, functions take **inputs** as explicit arguments and produce **outputs** as explicit results, with no side effects.
- In practice, programs are interactive and have side effects (some desirable, other not so much).
- Reading from keyboard, printing on the screen, relying on randomness are examples of side effects.
- Interactive programs by nature require the side effects of taking additional inputs and producing additional outputs while the program is runnning.
- In this lecture, we will learn to create interactive programs in Haskell and how a purely a functional language might deal with side-effects.

What is a side effect?

Side effects refer to any modification to the program's state that is observable outside of the function itself, such as:

- Modifying a global variable or data structure;
- Reading input from the keyboard or from a file;
- Writing output to the console or to a file;
- Making a network request;
- Printing something to the screen;
- Relying on randomness.

I/O

- Sometimes programs need to interact with the outside world.
- In Haskell, IO type represents computations that interact with the outside world, such as reading from the keyboard or writing to a file.
- Whenever an operation has side-effects, it is marked with IO, as an indicator of possible side-effect.
- This allows the developer to segregate the pure part of the code from the part that might produce side-effects.
- It also allows to compiler to check that side-effects do not happen when they shouldn't.

Finally, hello world!

- You may have notice in Lecture 1 that our `main` function has type `IO ()`:

```
main :: IO ()
main = putStrLn "Hello world!"
```

- To compile our program, we execute the command:

```
% ghc --make helloworld.hs
[1 of 1] Compiling Main             ( helloworld.hs,
helloworld.o )
Linking helloworld ...
```

- To run the program, we run:

```
% ./helloworld
Hello world!
```

- Alternatively, we can run the Haskell program directly by executing:

```
% runhaskell helloworld.hs
Hello world!
```

Demo GHCI

In [ ]:
```haskell
-- Demo GHCI

let luckyNumber = 7
:t luckyNumber

let luckyNumber = 7 :: Int
:t luckyNumber

-- wrap the Int 7 with IO
-- indicates that it performs an 'IO action' and contains an Int
:t return luckyNumber :: IO Int

-- imperative style, with do notation
welcomeName :: IO ()
welcomeName = do
    putStrLn "What's your name?"
    name <- getLine -- reads a line from standard input and binds it to variabl
    putStrLn ("Hello" ++ name)
```

```
luckyNumber :: forall {a}. Num a => a
```

```
luckyNumber :: Int
```

```
return luckyNumber :: IO Int :: IO Int
```

Do notation

To use an IO operation, we typically use the "do" notation, which allows us to sequence multiple IO actions in a block:

```
main :: IO ()
main = do
    putStrLn "What's your name?"
    name <- getLine
    putStrLn ("Hello, " ++ name ++ "!")
```

- `putStrLn` and `getLine` functions are **IO actions**.
- IO actions are sequenced using the **do notation**, in an imperative-style manner.
- `<-` operator binds the result of an IO action to a variable.

Another look at IO

In Haskell, an interactive program can be seen as a pure function that takes the *current state of the world* and returns a *modified state of the world* in which the program was executed and side-effects occurred.

*Olafur Eliasson - The Weather Project in the Turbine Hall, Tate Modern, 2003

Consider a new data type World:

```
data World = World
```

How would `putStrLn` and `getStr` might look like with our new World type?

```
putStr' :: String -> World -> World
putStr' = undefined

getLine' :: World -> (String, World)
getLine' = undefined
```

(How these functions are actually defined does not really matter.)

Let's redefine our `welcomeName` function in a pure way in terms of our new data type:

```
welcomeName' :: World -> World
welcomeName' w0 = w3
  where
    w1        = putStr' "What's your name?" w0
    (name,w2) = getLine' w1
    w3        = putStr' w2 ("Hello" ++ name)
```

Our definition has a small problem: We might end up in split worlds 😳

```
splitWorlds :: World -> (World,World)
splitWorlds w = (putStr' "Schrödinger's cat is alive" w,
                 putStr' "Schrödinger's cat is dead" w)
```

Maybe this is not that far-fetched in a quantum world, but it does not really capture how programs are executed in a classical computer.

Hiding the world with a world transformer.

```
type WorldT a = World -> (a,World)
```

World transformer `WorldT` (parameterized by type `a`) is a type synonym of a function that takes as input a state of the world and returns a value of type `a` and another state of the world.

Notice that `WorldT` hides `World` from the user.

Rewritten our functions `putStr` and `getLine` in terms of `WorldT`:

```
putStr' :: String -> World -> World
putStr' = undefined

getLine' :: World -> (String, World)
getLine' = undefined
```

→

```
--putStrT :: String -> World -> (a,World)
--putStrT s w = ((), putStr' s w)

putStrT :: String -> WorldT ()
putStrT s w = ((), putStr' s w)

getLineT :: WorldT String
getLineT = getLine'
```

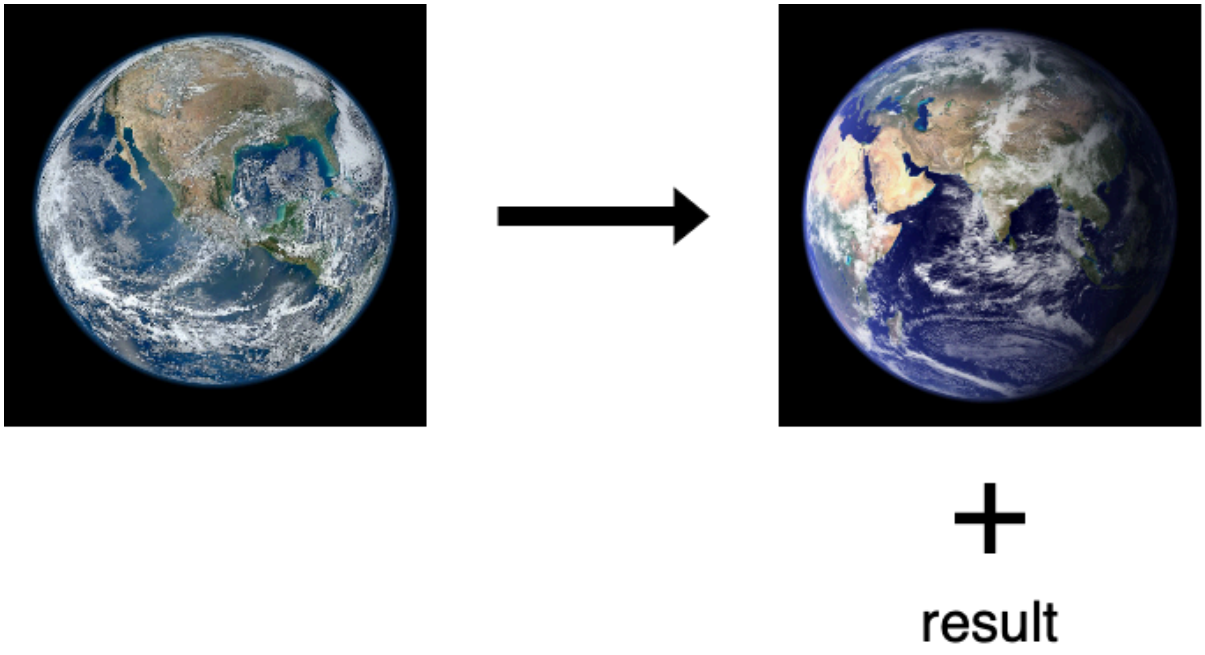
Type `IO` is just a synonym for `WorldT`.

```
type IO a = World -> (a,World)
```

Note: to use do notation with type `WorldT`, we must make it an instance of typeclass `Monad`. This is beyond the scope of this lecture and we refer to a future lecture of the course.

At this point, you might be worried about the feasability of passing around the entire state of the world when programming with actions. Don't worry, type `IO a` is provided as a primitive in Haskell.

IO is a world transformer



IO keeps *pure* and *impure* parts of the code separate.

- Inside `do` block, `if condition then I/O action else I/O action`.
- `return` is sort of the opposite to `<-`
- I/O action that's made up from a do block to have the result value of its last action

Functions `putChar`, `putStr`, `putStrLn` and `print`

In [5]:
```haskell
sayHello = do
    putChar 'H'
    putChar 'e'
    putChar 'l'

sayHello2 = do
    putStrLn "Hello"

sayHello2

putStr' :: String -> IO ()
putStr' [] = return ()
```

```
putStr' (x:xs) = do
    putChar x
    putStr' xs
```

```
Hello
```

Functions `getChar`, `getLine` and bindings

In [4]:
```
getName = do
    name <- getLine
    putStrLn ("Hello, " ++ name ++ "!")

getLine' :: IO String
getLine' = do
    c <- getChar
    if c == '\n' then return []
    else do
        rest <- getLine'
        return (c:rest)
```

Function `return`

- `return` is a function that takes a value and wraps it in a monadic context.
- In `IO` monad, `return` is used to wrap a pure value into an **IO action**.
- `return` does not exit a function early like in most imperative languages.
- In `do` notation, `<-` is kind of the reverse of `return`.

In [7]:
```
main = do
  a <- return "Programming "
  b <- return "Fundamentals 3"
  putStrLn $ a ++ b

main
```

```
Programming Fundamentals 3
```

`If-then-else` and `let` inside `do` notation

In this example, it doesn't make much sense to taint strings "Programming" and "Fundamentals 3" with IO type contructor, just to temporarily *un-taint* the data inside an I/O action by binding it to variables `a` and `b`.

```
main = do
    a <- return "Programming"
    b <- return "Fundamentals 3"
    putStrLn $ a ++ " " ++ b
```

Instead, we could use `let`:

```
import Data.Char
```

```
main = do
    let a = map toUpper "Programming"
        b = map toUpper "Fundamentals 3"
    putStrLn $ a ++ " " ++ b
```

We can also use `if-then-else` inside `do` notation. Here is another example:

```
main = do
    line <- getLine
    if null line
        then return ()
        else do
            putStrLn $ reverseWords line
            main

reverseWords :: String -> String
reverseWords = unwords . map reverse . words
```

Sequencing

Function `sequence` takes a list of I/O actions and returns an I/O actions that will perform those actions one after the other.

```
sequence :: [IO a] -> IO [a]
```

In [ ]:
```
sequence (map print [1,2,3,4,5])
```

```
1
2
3
4
5
[(),(),(),(),()]
```

Note: When we evaluate an I/O action in GHCI, the action is performed and the result is printed. In this example, the action is printing the values `1`, `2`, `3`, `4` and `5` and each result is `()`. If the result is `()`, GHCI simply does not print it. But because of `sequence`, the result is actually `[(),(),(),(),()]`, which is printed.

Functions `getContents`, `interact` and lazy evaluation

`getContents`

- `getContents` is a function that returns all the input from the standard input stream.
- The function doesn't block waiting for input. It returns a `String` lazily as the input stream is read.
- Because the `String` returned by getContents is lazily evaluated, it's possible to process very large inputs without running out of memory.

`interact`

Observe the following pattern:

```
main = do
    c <- getContents
    print $ f c
```

- `interact` is a higher-order function that captures this pattern, taking a function of type `String -> String` as an argument, and returns an IO action that reads input from the standard input stream, passes it through the function, and then prints the output to the standard output stream.
- Again, the input is read lazily, and the output is generated lazily as well, which allows the function to process very large inputs and outputs without running out of memory.
- Lazy evaluation is a key feature of Haskell that allows computations to be deferred until their results are actually needed.

In [ ]:
```haskell
interact' :: (String -> String) -> IO ()
interact' f = do
    c <- getContents
    print $ f c
```

We could re-write the above code simply as (here, we give a more concrete example where the stdin is printed to the stdout in CAPSLOCK):

```
import Data.Char

main = interact f
    where f = map toUpper
```

Exercises 😵

(1) Warmup, but in 1 line!

HackerRank > Prepare > Algorithms > Warmup > Solve Me First

(2) Simple Array Sum

HackerRank > Prepare > Algorithms > Warmup > Simple Array Sum

(3) Time Conversion

HackerRank > Prepare > Algorithms > Warmup > Time Conversion

In [ ]:
```haskell
-- (1) Warmup, but in 1 line!
module Main where

main = interact $ show . sum . map read . words
```

```
-- "7\n3\n" ->
-- ["7","3"] ->
-- [7,3] ->
-- 10 ->
-- "10"
```

In [ ]:
```haskell
-- (2) Simple Array Sum
module Main where

main = interact $ show . sum . map read . tail . words

-- "6\n1 2 3 4 10 11\n" ->
-- ["6","1","2","3","4","10","11"] ->
-- ["1","2","3","4","10","11"] ->
-- [1,2,3,4,10,11] ->
-- 31 ->
-- "31"
```

In [ ]:
```haskell
module Main where

-- cabal install --lib split
-- Hoogle: Eq a => [a] -> [a] -> [[a]]
import Data.List.Split (splitOn)

main :: IO ()
main = interact convertTime

takeEnd :: Int -> String -> String
takeEnd n xs = drop (length xs - n) xs

dropTail :: Int -> String -> String
dropTail n xs = reverse $ drop n (reverse xs)

convertTime :: String -> String
convertTime twelveHourFormat
    | amPm == "AM" && hours == "12" = "00" ++ minSec ++ "\n"
    | amPm == "PM" && hours == "12" = "12" ++ minSec ++ "\n"
    | amPm == "AM"                  = hours ++ minSec ++ "\n"
    | otherwise                     = show (read hours + 12) ++ minSec ++ "\n"
  where
    amPm = takeEnd 2 twelveHourFormat
    splitFormat = splitOn ":" twelveHourFormat
    hours = splitFormat !! 0
    minutes = splitFormat !! 1
    secAmPm = splitFormat !! 2
    minSec = ":" ++ minutes ++ ":" ++ (dropTail 2 secAmPm)
```