Programming Fundamentals 3

University of Luxembourg

Winter Semester 2023

Marjan Skrobot marjan.skrobot@uni.lu

Thanks to Agnese GINI

Outline

- Higher-order functions: map, filter, foldr
- Currying
- String transmitter program

map

Standard prelude definition:

```
map :: (a -> b) -> [a] -> [b]
map f xs = [ f x | x <- xs ]
```

map

Recursive definition of map:

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

filter

Standard prelude definition:

```
filter :: (a -> Bool) -> [a] -> [a]
filter p xs = [ x | x <- xs, p x]
```

filter

Recursive definition of filter:

```
filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs) | p x = x : filter p xs
                | otherwise = filter p xs
```

map, filter

Task 1: Show how the list comprehension `[ f x | x <- xs , p x ]` can be re-expressed using the higher-order functions **map** and **filter**?

Solution 1:

```
map f (filter p xs)
```

foldr

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

map, filter, foldr

Task 2: Redefine the functions **map f** and **filter p** using foldr.

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

filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs) | p x = x : filter p xs
                | otherwise = filter p xs
```

map, filter, foldr

Task 2: Redefine the functions **map f** and **filter p** using foldr.

Solution 2:

a) map f = foldr (\x acc -> (f x) : acc) []

b) filter p = foldr (\x acc -> if p x then x : acc else acc) []

map, filter, foldr

Task 3: Modify functions from Solution 2 to define:

a) filterEven :: [Integer] -> [Integer] that filters even values and

b) mapDiv2 :: Integral a => [a] -> [a] that divides a list by 2.

```
filter p = foldr (\x acc -> if p x then x : acc else acc ) []
map f = foldr (\x acc -> (f x) : acc ) []
```

map, filter, foldr

Task 3: Modify functions from Solution 2 to define:

a) filterEven :: [Integer] -> [Integer] that filters even values and

Solution 3a)

```
filterEven :: [Integer] -> [Integer]
filterEven = foldr (\x acc -> if p x then  x : acc else acc ) []
   where p = even
```

map, filter, foldr

Task 3: Modify functions from Solution 2 to define:

a) filterEven :: [Integer] -> [Integer] that filters even values and
b) mapDiv2 :: Integral a => [a] -> [a] that divides a list by 2.

Solution 3b)

```
mapDiv2 :: Integral a => [a] -> [a]
mapDiv2 = foldr (\x acc -> f x : acc ) []
    where f x  = div x 2
```

map, filter, foldr

Task 4: Define function

```
applyif :: (a -> Bool ) -> (a -> b) -> [a] -> [b]
```

that applies a function to elements list on if a given condition is True.

Solution 4a:

```
applyif :: (a -> Bool ) -> (a -> b) -> [a] -> [b]
applyif p f xs = map f (filter p xs)
```

map, filter, foldr

Task 4: Define function

```
applyif :: (a -> Bool ) -> (a -> b) -> [a] -> [b]
```

that applies a function to elements list on if a given condition is True.

Solution 4b:

```
applyif’ :: (a -> Bool ) -> (a -> b) -> [a] -> [b]
applyif’ p f = map f . filter p
```

Composition operator

The higher-order operator (.) returns the composition of two functions as a single function.

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

Currying

Transforming a function that takes multiple arguments in a tuple as its argument, into evaluating a sequence of functions, each with a single argument.

- `f :: a -> (b -> c)` is a curried form of `g :: (a, b) -> c`
- You can convert these two types in either directions with the Prelude functions curry and uncurry: `f = curry g` and `g = uncurry f`
- it holds that: `f x y = g (x,y)`

Currying

Task 5: Without looking at the standard prelude, define the higher-order library function curry that converts a function on pairs into a curried function, and, conversely, the function uncurry that converts a curried function with two arguments into a function on pairs.

```
curry (\(x,y) -> x + y) 1 8.
```

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

```
uncurry (+) (1, 8)
```

```
uncurry :: (a -> b -> c) -> (a , b) -> c
uncurry f (a , b) = f a b
```

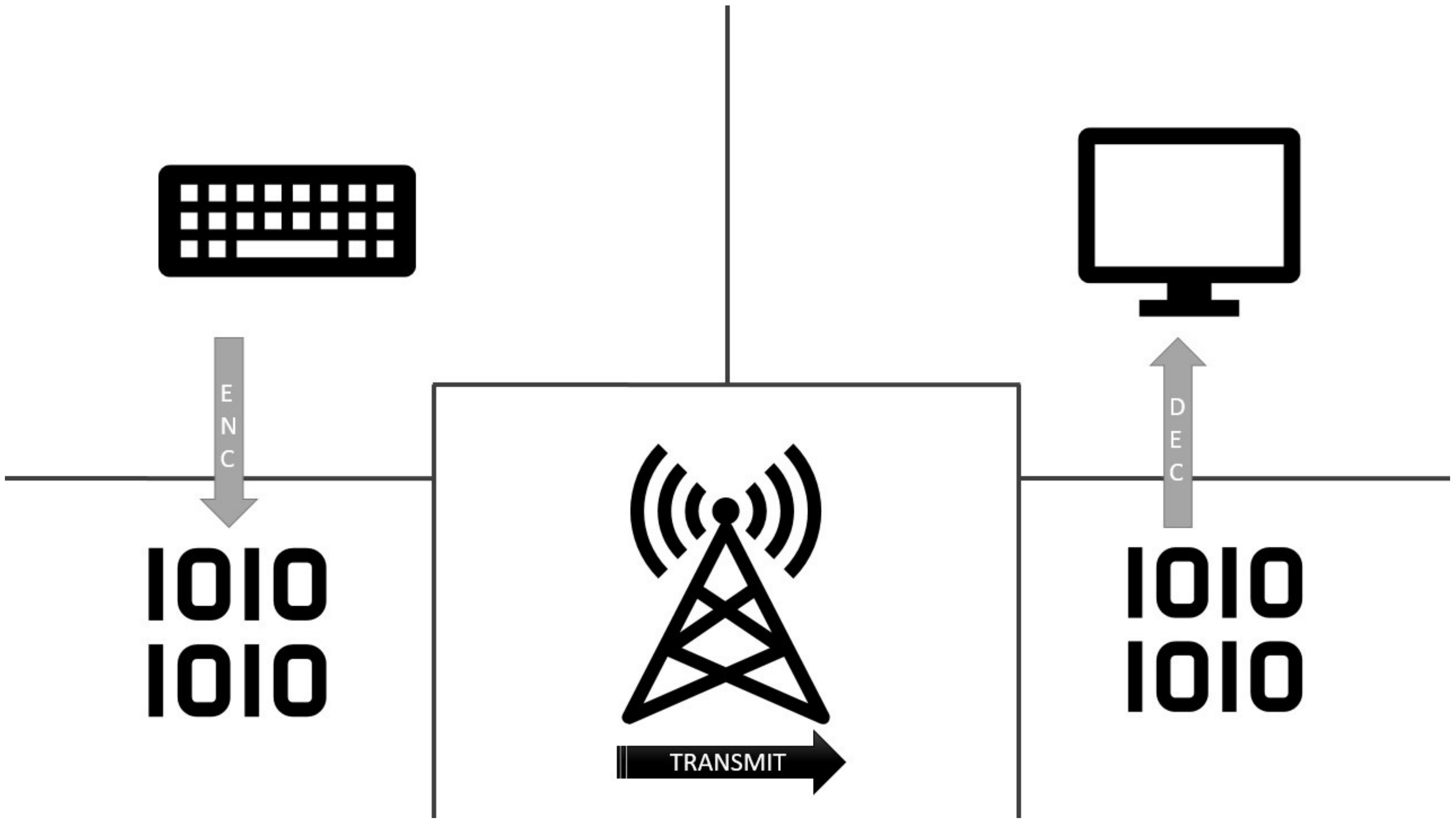
Currying

```
curry (\(x,y) -> x + y) 1 8

curry :: ( ( a , b ) -> c) -> a -> b -> c
curry f a b = f (a , b)

(curry f) x y = f (x,y)
((curry f) x) y = f (x,y)
curry f = \x -> \y -> f (x,y)
```

String transmitter program



String transmitter program

Main Task: simulating the transmission of a string of characters in low-level form as a list of binary digits.

Key-points:

- encode a string as a list of binary digits
- simulate a channel
- decode the list of binary digits

Encode and Decode

A string is list of characters. So, to encode a string as a number we can use the functions from Data.Char library.

`ord :: Char -> Int` converts a character to a Unicode number.

`chr : : Int -> Char` converts a Unicode number to a character.

Bit to Int

```
import Data.Char
type Bit = Int

bin2int :: [Bit] -> Int
bin2int bits = sum [w*b | (w,b) <- zip weights bits]
    where weights = iterate (*2) 1
```

Bit to Int

import Data.Char

type Bit = Int

simplification is possible : a + 2*(b + 2*(c + 2*d))

```
bin2int :: [Bit] -> Int
bin2int = foldr (\x y -> x+2*y) 0
```

Int to Bit

```
int2bin :: Int -> [Bit]
int2bin 0 = []
int2bin n = n `mod` 2 : int2bin (n `div` 2)
```

[Bit] into Byte

We will ensure that all our binary numbers have the same length, in this case a byte (=eight bits), by using a function make8 that truncates or extends a binary number as appropriate to make it precisely 8 bits:

```
type Byte = [Bit]

make8 :: [Bit] -> Byte
make8 bits = take 8 (bits ++ repeat 0)
```

Encoding

We can finally encode string for low-level transmission.

```
encode :: String -> [Bit]
encode = concat . map (make8 . int2bin . ord)
```

Decoding

To decode a list of bits we first need to chop it into 8-bit binary numbers:

```
chop8 :: [Bits] -> [Byte]
chop8 [] = []
chop8 bits = take 8 bits : chop8 (drop 8 bits)
```

```
decode :: [Bit] -> String
decode = map (chr . bin2int) . chop8
```

Transmission

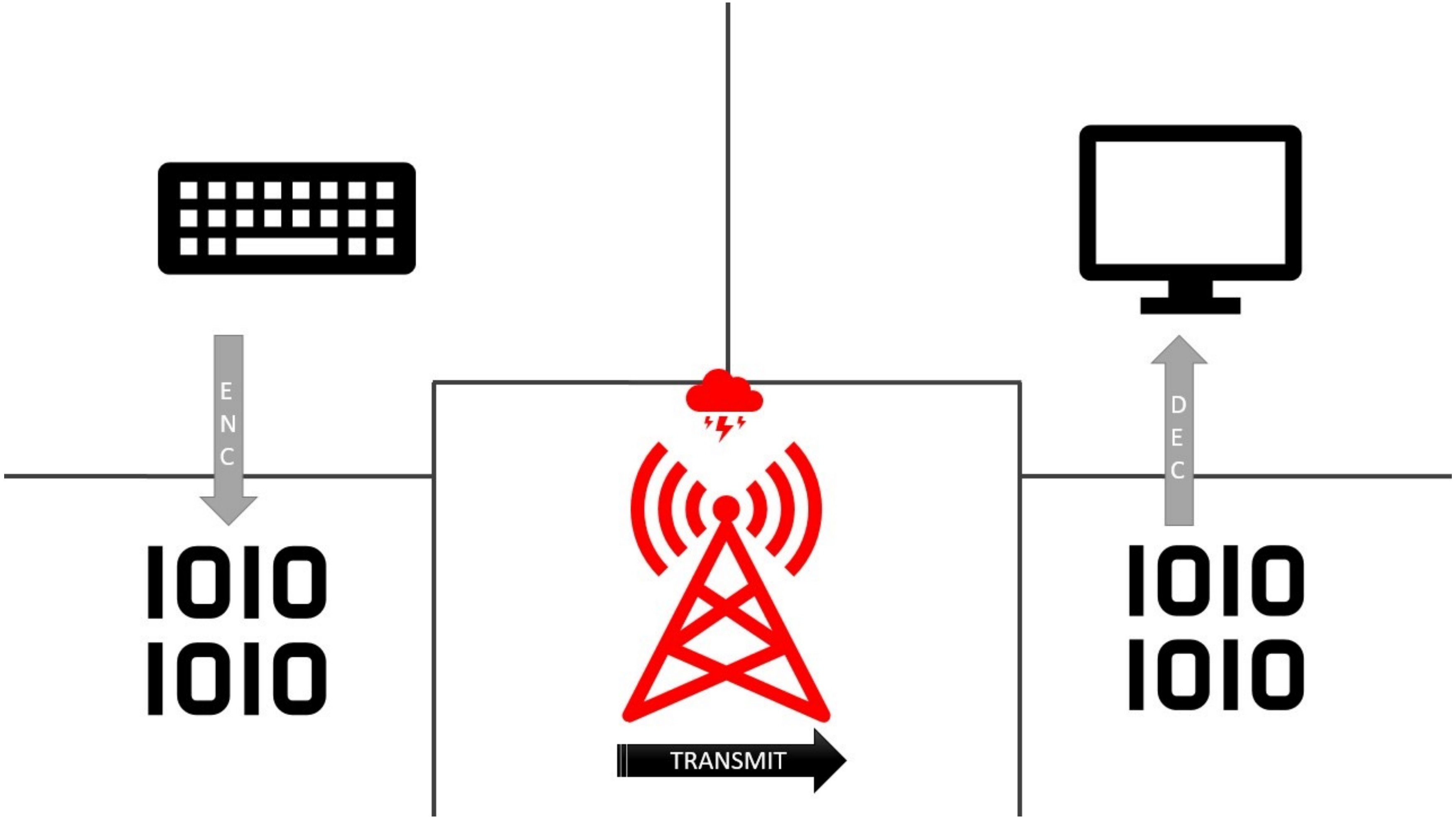
A transmission is sending a string via a given channel by transmitting bits. For instance, we can model *perfect communication* with the identity function.

```
channel :: [Bit] -> [Bit]
channel = id
```

```
transmit :: String -> String
transmit = decode . channel . encode
```

transmit "A good programmer is someone who always looks both ways before crossing a one-way street. - Doug Linder, computer scientist"

Faulty Transmission



Faulty Transmission

Exercise: Modify the binary string transmitter example to detect simple transmission errors using the concept of parity bits. That is, each eight- bit binary number produced during encoding is extended with a parity bit, set to one if the number contains an odd number of ones, and to zero otherwise. In turn, each resulting nine-bit binary number consumed during decoding is checked to ensure that its parity bit is correct, with the parity bit being discarded if this is the case, and a parity error being reported otherwise.

Faulty Transmission

Hint: the library function error :: String -> a displays the given string as an error message and terminates the program; the polymorphic result type ensures that error can be used in any context.