# Estudos de caso

Programação Funcional

Prof. Rodrigo Ribeiro

# Setup

```haskell
module Aula08 where

import Data.Char (chr, ord, isLower)

main :: IO ()
main = return ()
```

# Objetivos

- Utilizar o conteúdo visto até o momento para solução de dois problemas: criptografia e serialização

# Cifra de César

- Algoritmo simples para criptografia de textos
- Consiste em deslocar caracteres em $n$ posições.
- Exemplo: `encode 2 "abc" == "cde"`.

# Cifra de César

- Convertendo um caractere em inteiro.

```
char2Int :: Char -> Int
char2Int c = ord c - ord 'a'

int2Char :: Int -> Char
int2Char n = chr (ord 'a' + n)
```

# Cifra de César

- Criptografando um caractere

```
shift :: Int -> Char -> Char
shift n c
  | isLower c = int2Char ((char2Int c + n) `mod` 26)
  | otherwise = c
```

# Cifra de César

- Criptografando uma string

```
encrypt :: Int -> String -> String
encrypt n s = map (shift n) s
```

# Cifra de César

- Descriptografando uma string

```
decrypt :: Int -> String -> String
decrypt n s = encrypt (- n) s
```

# Serialização

- Conversão de valores em sequências de bits.
- Como representar isso em Haskell?

# Serialização

- Função `zip`.

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

# Serialização

- Gerando uma lista infinita de valores.
  - Possível graças a *lazy evaluation*.

```
repeat :: a -> [a]
repeat x = x : repeat x

Prelude*> take 3 (repeat 2)
[2,2,2]
```

# Serialização

- Função `iterate`. - Também faz uso de *lazy evaluation*.

```
iterate :: (a -> a) -> a -> [a]
iterate f x = x : iterate (f x)
```

# Serialização

- Convertendo bits em números.

```
type Bit = Int

bin2Int :: [Bit] -> Int
bin2Int bs = sum [w * b | (w,b) <- zip weights bs]
         where
           weights = iterate (* 2) 1
```

# Serialização

- Convertendo números em bits.

```
int2Bin :: Int -> [Bit]
int2Bin 0 = []
int2Bin n = n `mod` 2 : int2Bin (n `div` 2)
```

# Serialização

- Gerando bytes

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

# Serialização

- Dividindo em listas de bytes

```
chop8 :: [Bit] -> [[Bit]]
chop8 [] = []
chop8 bs = take 8 bs : chop8 (drop 8 bs)
```

# Serialização

- Codificando uma string

```
encode :: String -> [Bit]
encode = concat . map (make8 . int2Bin . ord)
```

# Serialização

- Decodificando uma string

```
decode :: [Bit] -> String
decode = map (chr . bin2Int) . chop8
```

# Exercício

- Reimplemente a função `bin2Int` utilizando `foldr`.
- Um inconveniente da solução de serialização é a utilização do tipo `Bit` como um sinônimo. Idealmente, deveríamos utilizar um novo tipo, como se segue.

```
data Bit = O | I deriving Show
```

Modifique a implementação da serialização para utilizar essa versão do tipo `Bit`.