Programação Funcional
Laziness

André Santos

(a partir de *slides* elaborados por Fernando Castor e Márcio Cornélio)

Laziness

- *Lazy evaluation*
  - avaliação de uma expressão se dá apenas quando seu valor é necessário

```
f :: Int -> Int -> Int
f a b = a + b
f (9-3) (f 3 5) = (9-3) + (f 3 5)
                = 6 + (3+5)
                = 6 + 8
                = 14
```

Laziness

```
f1 :: Int -> Int -> Int
f1 a b = a + 12

f1 (9-3) (f1 34 3)
= (9 - 3) + 12
= 6 + 12
= 18
```

Laziness

```
f1 :: Int -> Int -> Int
f1 a b = a + 12
g :: Int -> Int
g c = c + g c

f1 3 (g 0) = ?
f1 (g 0) 3 = ?
```

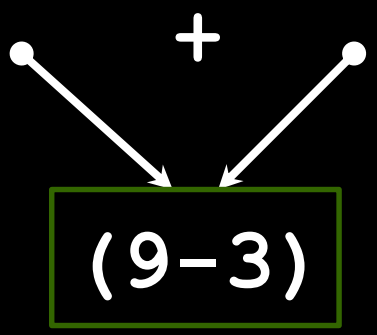
Laziness

```
troca :: Integer -> a -> a -> a
troca n x y
  | n > 0 = x
  | otherwise = y
```

Laziness

- Se a expressão se repete a partir de um nome compartilhado, NÃO é feita avaliação duplicada

```
f :: Int -> Int
f a = a + a
f (9-3) =
```



- Expressões são representadas como grafos

Laziness

- Atenção: se a expressão está repetida explícitamente, é feita avaliação duplicada

```
f :: Int -> Int -> Int
f a b = a + b
f (9-3) (9-3) = (9-3) + (9-3)
              =   6   + (9-3)
              =   6   +   6
              =   12
```

Laziness

- Argumentos não precisam ser avaliados por completo
  - apenas o necessário para a execução da função

```
f1 :: [Int] -> [Int] -> Int
f1 (a:as) (b:bs) = a + b

f1 [1..] [2..] = ?
```

Laziness

- Argumentos não precisam ser avaliados por completo
  - apenas o necessário para a execução da função

```
f12 :: [Int] -> [Int] -> Int
f12 [] ys = 0
f12 (x:xs) [] = 0
f12 (x:xs) (y:ys) = x + y
f12 [1..3] [1..3] = ?
```

Laziness

```
f12 [1.. 3 ] [1 .. 3]
= f12 (1: [2..3]) [1..3]
= f12 (1: [2..3]) (1: [2..3])
= 1 + 1
```

Recursão de Cauda

- Chamadas cujo resultado é diretamente "retornado" por quem faz a chamada:

```
fat n = tailFat n 1
tailFat 0 x = x
tailFat n x = tailFat(n-1)(n*x)
```

- **Tornam desnecessário que seja empilhado um *stack frame* por chamada**
  - **Evitam estouros de pilha**
- **O GHC transforma diversas chamadas "comuns" em chamadas de cauda**