Minicurso de Haskell

Funções de alta ordem
Type Variables
Sintaxe

Relembrando

Aplicação Parcial

```haskell
dobra :: Int -> Int
dobra x = 2 * x
```

```haskell
dobra :: Int -> Int
dobra = \x -> 2 * x
```

-- Função Lambda

```haskell
dobra :: Int -> Int
dobra = (2*)
```

-- Aplicação Parcial

Funções Infixas

- Caracteres especiais
- Definida entre parênteses

```
(*) :: Int -> Int -> Int
x * y = …
```

```
($) :: (a -> b) -> a -> b
f $ x = f x
```

-- Aplicação de função

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

-- Composição de funções

Tipe Variables

“Variáveis de tipo”

Geralmente representadas por alguma letra qualquer

Substituem qualquer tipo

Polimorfismo!

```
id :: a -> a
id x = x


($) :: (a -> b) -> a -> b
f $ x = f x


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

Pequeno Exercício

Defina as seguintes funções

```
(%) :: Int -> Int -> Int
-- que retorna o resto da
divisão entre dois números (use
a função mod, já existente)
```

```
(&) :: a -> (a -> b) -> b
-- que recebe um argumento
e aplica uma função nele
```

Point-free syntax

Em funções infixas

```haskell
(%) :: Int -> Int -> Int
x % y   = mod x y
(%) x y = mod x y
(%)     = mod
```

-- Todas essas definições estão corretas

```haskell
(&) = flip ($)
```

-- Para este caso o Haskell também já possui uma função

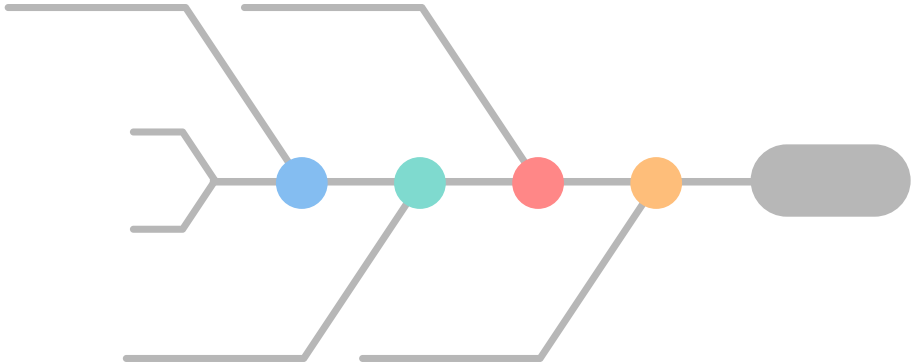
Compondo Funções

Anteriormente citamos a função *(.)*

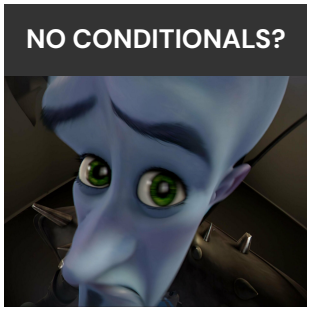
```
quadradoDoDobro      = (^2) . (*2)
raizQuadradaDoModulo = sqrt . abs
raizNDoModulo rad    = (**) recip rad . abs
```

-- (**) é a potenciação para Float e Double, (^) aceita apenas Int

-- recip x = 1 / x



Estruturas de Controle



Pattern Matching

Acontece de cima
para baixo

Substitui os IFs

```
safeDiv :: Float -> Float -> Float
safeDiv x 0 = 0
safeDiv x y = x / y
```

-- Se tentar dividir por zero
retorna zero (vamos melhorar isso
aqui mais tarde)

Case Expressions

Introduz um pattern matching dentro do corpo da função

```
safeDiv :: Float -> Float -> Float
safeDiv x y =
 case y of
   0 -> 0
   _ -> x / y
```

Bool

True ou False

Suporte por pattern matching

```
(&&) :: Bool -> Bool -> Bool
True && True = True
_    && _    = False
```

```
(||) :: Bool -> Bool -> Bool
False || False = False
_     || _     = True
```

```
not :: Bool -> Bool
not False = True
not True  = False
```

Pequeno Exercício

Implemente a seguinte função usando pattern-matching

Que realiza a operação lógica xor entre dois booleanos (True quando os dois forem diferentes).

```
xor :: Bool -> Bool -> Bool
```

Guards

Basicamente uma série de ifs

```
safeDiv :: Float -> Float -> Float
safeDiv x y
  | y == 0     = 0
  | otherwise = x / y
```

Pequeno Exercício

Estenda a seguinte função

Para que retorne uma mensagem diferente para os casos em que o aluno tirou 10 ou 0

```
avaliacao :: Float -> Float -> String
avaliacao x y
 | media < 7   = "Reprovado"
 | otherwise   = "Aprovado"
 where media = (x + y) / 2
```

Where

Define um valor no escopo da função

```
avaliacao3 :: Float -> Float -> String
avaliacao3 x y = strResultado ++ " com media " ++ show media
 where media = (x + y) / 2
       strResultado
         | media < 7 = "Reprovado"
         | otherwise = "Aprovado"
```

-- função show transforma coisas em string