Minicurso de Haskell

Tuplas
Listas
Maybe

Tuplas

Dupla de valores

Podem ser de tipos diferentes

Suporte por pattern matching

```haskell
fst :: (a, b) -> a
fst (x, _) = x
```

```haskell
snd :: (a, b) -> b
snd (_, y) = y
```

Pequeno Exercício

Defina as seguintes funções

```
appSnd :: (a, b) -> (b -> c) -> (a, c)
-- que transforma o segundo valor de uma tupla
```

Use pattern matching para separar os valores!

Listas

Lista Encadeada

Todos os elementos do mesmo tipo

Função (:) suportada por pattern matching

```
head :: [a] -> a
head (x:_) = x

tail :: [a] -> [a]
tail (_:xs) = xs

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

(:) :: a -> [a] -> [a]
(++) :: [a] -> [a] -> [a]
```

Pequeno Exercício

Defina as seguintes funções

```
any1 :: (a -> Bool) -> [a] -> Bool
-- Que retorna True caso qualquer
elemento da lista satisfaça a função
```

Use recusão!

Folding

Família de funções

"Comprime" os elementos de uma lista em um único valor

Em outras linguagens aparece como *reduce*

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

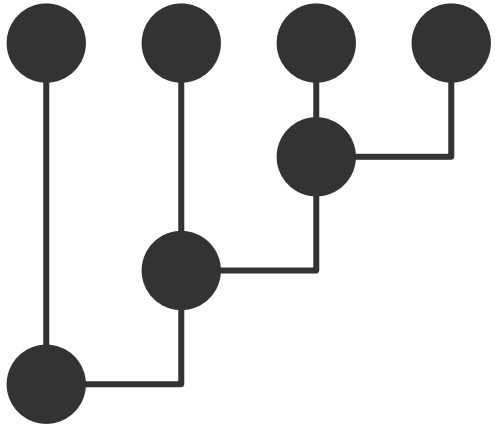
```haskell
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f b []       = b
foldr f b (x : xs) = f x (foldr f b xs)
```

```haskell
-- Não esquente com as definições, o importante é saber usar
```

foldr

foldr **g** a [b, c]

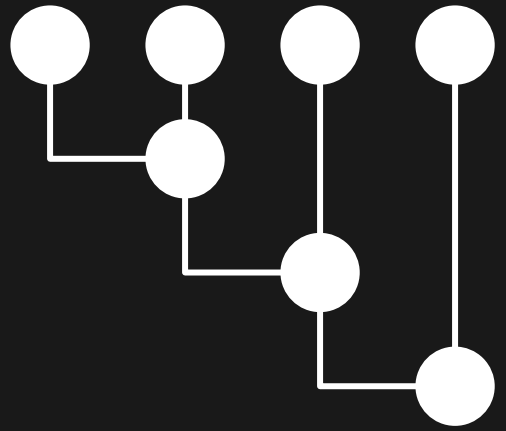
g b (**g** c a)



foldl

foldl g a [b, c]

g c (g a b)



Folding

Família de funções

Se a função for comutativa não há diferença

```haskell
sum :: [Int] -> Int
sum = foldl (+) 0
```

-- exemplo clássico

```haskell
and :: [Bool] -> Bool
and = foldl (&&) True
```

```haskell
or :: [Bool] -> Bool
or = foldl (||) False
```

```haskell
foldl1 :: (a -> a -> a) -> [a] -> a
foldl1 f (x:xs) = foldl f x xs
```

-- se você não quiser ter que passar um parâmetro inicial sempre

List Comprehension

```
[(x, y) | x <- [1..5], y <- ['a'..'d']]
```

-- Produto cartesiano

```
[1..]
```

-- Lista infinita (sim)

```
[1..100]
```

-- Lista de 1 a 100

```
doubleOdds xs = [x*2 | x <- xs, odd x]
```

Funções sobre listas

(!!)	[a] -> Int -> a
take	Int -> [a] -> [a]
drop	Int -> [a] -> [a]
filter	(a -> Bool) -> [a] -> [a]
scanl	(b -> a -> b) -> b -> [a] -> [a]
replicate	Int -> a -> [a]
reverse	[a] -> [a]
zipWith	(a -> b -> c) -> [a] -> [b] -> [c]

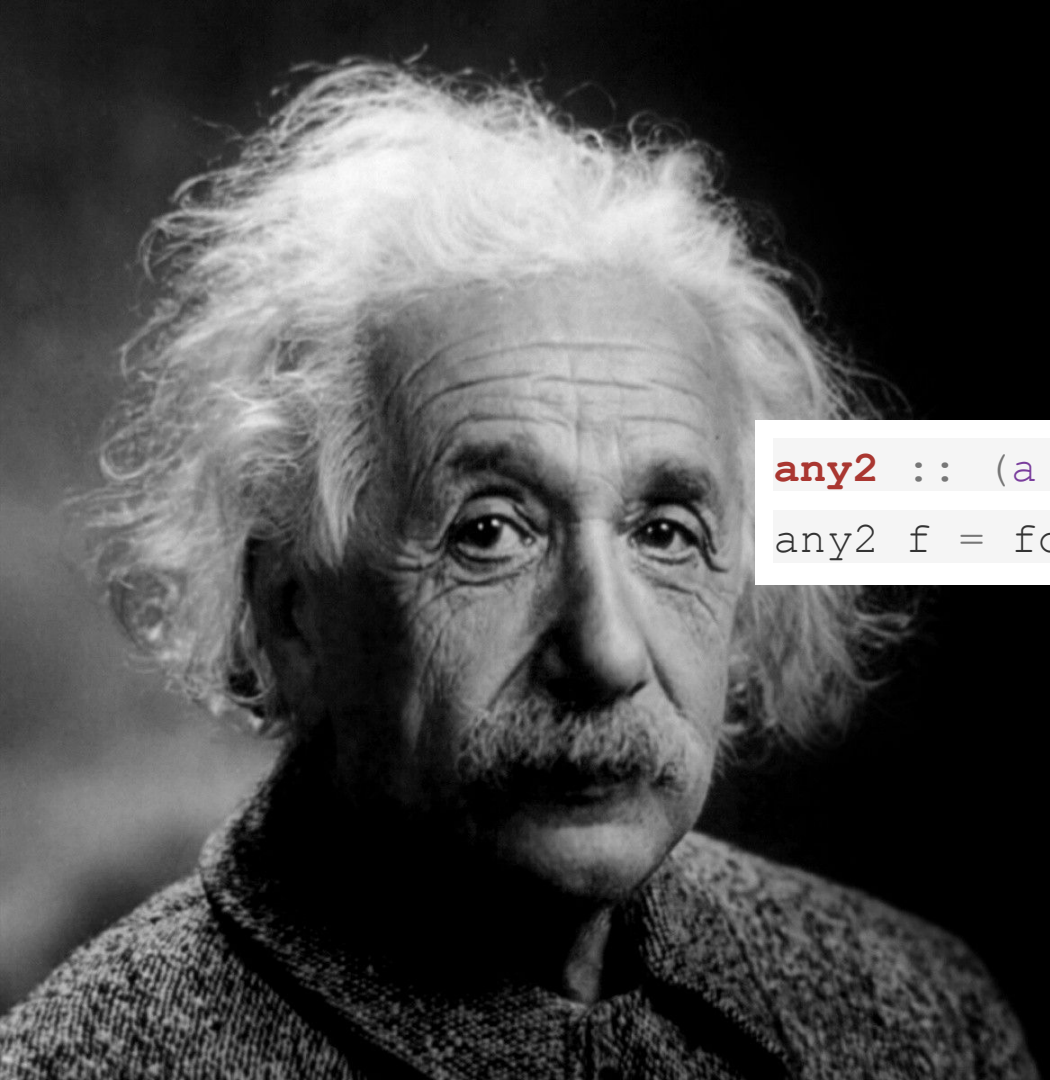
Maybe

Representa a possível ausência de valor

Dois construtores (Nothing e Just)

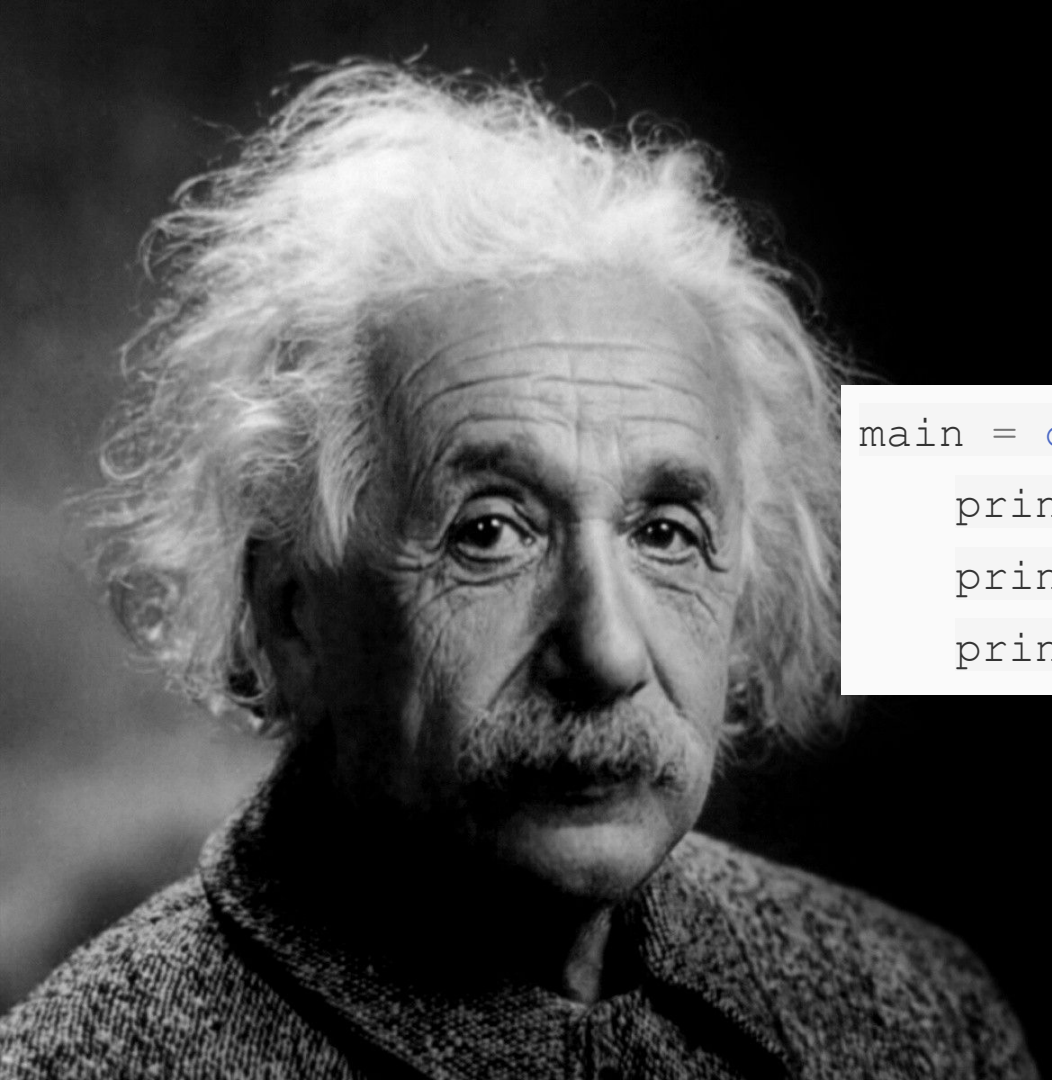
Em outras linguagens aparece como *Null*, *None*, *NaN*, *undefined* ou todos esses ao mesmo tempo

```
data Maybe a = Just a | Nothing
   -- Como se define um novo tipo

safeDiv :: Float -> Float -> Maybe Float
safeDiv a 0 = Nothing
safeDiv a b = Just $ a / b


safeHead :: [a] -> Maybe a
safeHead []     = Nothing
safeHead (x:_) = Just x
```



```haskell
any2 :: (a -> Bool) -> [a] -> Bool
any2 f = foldl (||) False . map f
```

Albert Einstein



```haskell
main = do
    print $ (*2) <$> [1..10]
    print $ (*2) <$> (7, 6)
    print $ (*2) <$> Just 3
```

Albert Einstein