# Programação Funcional
# Funções de ordem superior, Listas infinitas

2022

# Funções de ordem superior

Uma função é de ordem superior se tem um argumento que é uma função ou um resultado que é uma função.

Exemplo: o primeiro argumento de *map* é uma função, logo *map* é uma função de ordem superior.

```
> map (^2) [1,2,3,4]
[1,4,9,16]
```

# Porquê ordem superior?

- Permite parametrizar funções passando-lhes *operações* e não apenas *dados*
- Permite definir padrões de computação comuns que podem ser facilmente re-utilizados
- Mais tarde veremos que podemos provar propriedades gerais de funções de ordem superior
  - exemplo: *map* mantém o comprimento da lista dada

# Nesta aula

Algumas funções de ordem superior definidas no prelúdio-padrão:

- `map`
- `filter`
- `takeWhile`, `dropWhile`
- `all`, `any`
- `foldr`, `foldl`
- `(.)` (composição)

# A função *map*

A função *map* aplica uma função a cada elemento duma lista.

```
map :: (a -> b) -> [a] -> [b]
```

## Exemplos

```
> map (+1) [1,3,5,7]
[2,4,6,8]

> map isLower "Hello!"
[False,True,True,True,True,False]
```

# A função *map* (cont.)

Podemos definir *map* usando uma lista em compreensão:

```
map f xs = [f x | x<-xs]
```

Também podemos definir *map* usando recursão:

```
map f []     = []
map f (x:xs) = f x : map f xs
```

(A forma recursiva será útil quando provarmos propriedades usando indução.)

# Função *filter*

A função *filter* seleciona elementos duma lista que satisfazem um predicado (uma função cujo resultado é um valor boleano).

```
filter :: (a -> Bool) -> [a] -> [a]
```

## Exemplos

```
> filter (\n->n`mod`2==0) [1..10]
[2,4,6,8,10]

> filter isLower "Hello, world!"
"elloworld"
```

# Função *filter* (cont.)

Podemos definir *filter* usando uma lista em compreensão:

```
filter p xs = [x | x<-xs, p x]
```

Também podemos definir *filter* usando recursão:

```
filter p [] = []
filter p (x:xs)
      | p x       = x : filter p xs
      | otherwise = filter p xs
```

# Funções *takeWhile* e *dropWhile*

*takeWhile* seleciona o maior prefixo duma lista cujos elementos verificam um predicado.

*dropWhile* remove o maior prefixo cujos elementos verificam um predicado.

As duas funções têm o mesmo tipo:

```
takeWhile, dropWhile :: (a -> Bool) -> [a] -> [a]
```

# Funções *takeWhile* e *dropWhile* (cont.)

## Exemplos

```
> takeWhile isLetter "Hello, world!"
"Hello"

> dropWhile isLetter "Hello, world!"
", world!"

> takeWhile (\n -> n*n<10) [1..5]
[1,2,3]

> dropWhile (\n -> n*n<10) [1..5]
[4,5]
```

# Funções *takeWhile* e *dropWhile* (cont.)

Poderiamos definir*takeWhile* e *dropWhile* de forma recursiva.

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

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

# As funções *all* e *any*

*all* verifica se um predicado é verdadeiro para todos os elementos duma lista.

*any* verifica se um predicado é verdadeiro para algum elemento duma lista.

As duas funções têm o mesmo tipo:

```
all, any :: (a -> Bool) -> [a] -> Bool
```

# As funções *all* e *any* (cont.)

## Exemplos

```
> all (\n -> n`mod`2==0) [2,4,6,8]
True

> any (\n -> n`mod`2/=0) [2,4,6,8]
False

> all isLower "Hello, world!"
False

> any isLower "Hello, world!"
True
```

# As funções *all* e *any* (cont.)

Podemos definir *all* e *any* usando *map*, *and* e *or*:

```
all p xs = and (map p xs)
any p xs = or (map p xs)
```

Também podemos definir por recursão:

```
all p []     = True
all p (x:xs) = p x && all p xs

any p []     = False
any p (x:xs) = p x || any p xs
```

# A função *foldr*

Muitas transformações sobre listas seguem o seguinte padrão de *recursão primitiva*:

$$\texttt{f []} \quad = z$$
$$\texttt{f (x:xs)} = \texttt{x} \oplus \texttt{f xs}$$

Ou seja, *f* transforma:

a lista vazia em $z$;

a lista não-vazia $x : xs$ usando uma operação $\oplus$ para combinar $x$ com o resultado da função para $xs$.

# A função *foldr* (cont.)

## Exemplos

```
sum []     = 0
sum (x:xs) = x + sum xs
```

$z = 0$

$\oplus = +$

```
product []     = 1
product (x:xs) = x * product xs
```

$z = 1$

$\oplus = *$

```
and []     = True
and (x:xs) = x && and xs
```

$z = \texttt{True}$

$\oplus = \texttt{\&\&}$

```
or []     = False
or (x:xs) = x || or xs
```

$z = \texttt{False}$

$\oplus = \texttt{||}$

```
length []     = 0
length (x:xs)= 1 + length xs
```

$z = 0$

$\oplus = \backslash\_\ n \to 1 + n$

# A função *foldr* (cont.)

A função de ordem superior *foldr* (fold right) abstrai este padrão de recursão; os seus argumentos são a operação $\oplus$ e o valor $z$.

```
sum     = foldr (+) 0

product = foldr (*) 1

and     = foldr (&&) True

or      = foldr (||) False

length  = foldr (\_ n->n+1) 0
```

# A função *foldr* (cont.)

A definição recursiva de *foldr* (do prelúdio-padrão) exprime o padrão de recursão.

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

# A função *foldr* (cont.)

É possível visualizar *foldr f z* como uma transformação sobre estruturas de listas:

- cada (:) transforma-se numa aplicação de *f*;
- [] transforma-se na constante *z*.

```
     foldr f z [1,2,3,4,5]
=    foldr f z (1:2:3:4:5:[])
=  f 1 (f 2 (f 3 (f 4 (f 5 z))))
```

# A função *foldr* (cont.)

## Exemplo

```
  sum [1,2,3,4]
=
  foldr (+) 0 [1,2,3,4]
=
  foldr (+) 0 (1:(2:(3:(4:[]))))
=
  1+(2+(3+(4+0)))
=
  10
```

# A função *foldr* (cont.)

## Outro exemplo

```
  product [1,2,3,4]
=
  foldr (*) 1 [1,2,3,4]
=
  foldr (*) 1 (1:(2:(3:(4:[]))))
=
  1*(2*(3*(4*1)))
=
  24
```

# A função *foldl*

A função *foldr* transforma uma lista usando uma operação associada à direita (*fold right*):

$$foldr\ (\oplus)\ z\ [x_1, x_2, \ldots, x_n] = x_1 \oplus (x_2 \oplus (\ldots (x_n \oplus z) \ldots))$$

Existe outra função *foldl* que transforma uma lista usando uma operação associada à esquerda (*fold left*):

$$foldl\ (\oplus)\ z\ [x_1, x_2, \ldots, x_n] = ((\ldots ((z \oplus x_1) \oplus x_2) \ldots) \oplus x_n)$$

# A função *foldl* (cont.)

Se *f* for *associativa* e *z* for o *elemento neutro*, então *foldr f z* e *foldl f z* dão o mesmo resultado.

```
  foldl (+) 0 [1,2,3,4]
=
  (((0+1)+2)+3)+4
=
  10
```

```
  foldr (+) 0 [1,2,3,4]
=
  1+(2+(3+(4+0)))
=
  10
```

# A função *foldl* (cont.)

Tal como *foldr*, a função *foldl* pode ser definida por recursão:

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

# A função *foldl* (cont.)

Pode ser mais fácil visualizar *foldl* como uma transformação sobre listas.

```
     foldl f z [1,2,3,4,5]
=    foldl f z (1:2:3:4:5:[])
= f (f (f (f (f z 1) 2) 3) 4) 5
```

# Composição

A função $(\cdot)$ é a composição de duas funções.

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

## Exemplo

```
par :: Int -> Bool
par x = x`mod`2 == 0

impar :: Int -> Bool
impar = not . par
```

# Composição (cont.)

A composição permite muitas vezes simplificar definições embricadas, omitido os parêntesis e o argumento.

## Exemplo

```
f xs = sum (map (^2) (filter even xs))
```

é equivalente a

```
f = sum . map (^2) . filter even
```

# Listas infinitas

Podemos usar listas para *sequências finitas*, por ex.:

```
[1,2,3,4] = 1:2:3:4:[]
```

Nesta aula vamos ver que podemos também usar listas para representar *sequências infinitas*, e.g.

```
[1..] =  1:2:3:4:5:...
```

Não podemos descrever uma lista infinita em extensão; usamos listas em compreensão ou definições recursivas.

# Exemplos

```haskell
-- todos os números naturais
nats :: [Integer]
nats = [0..]

-- todos os números pares não-negativos
pares :: [Integer]
pares = [0,2..]

-- a lista infinita 1, 1, 1,...
uns :: [Integer]
uns = 1 : uns

-- todos os inteiros a partir de um número
intsFrom :: Integer -> [Integer]
intsFrom n = n : intsFrom (n+1)
```

# Processamento de listas infinitas

Por causa da *lazy evaluation* as listas são calculadas à medida da necessidade e apenas até onde for necessário.

```
  head uns
=
  head (1:uns)
=
  1
```

# Processamento de listas infinitas (cont.)

Uma computação que necessite de percorrer toda a lista infinita não termina.

```
  length uns
=
  length (1:uns)
=
  1 + length uns
=
  1 + length (1:uns)
=
  1 + (1 + length uns)
=
  ⋮
```

não termina

# Produzir listas infinitas

Muitas funções do prelúdio-padrão produzem listas infinitas quando os argumentos são listas infinitas:

```
> map (2*) [1..]
[2, 4, 6, 8, 10, ...

> filter (\x->x`mod`2/=0) [1..]
[1, 3, 5, 7, 9, ...
```

Também podemos usar notação em compreensão:

```
> [2*x | x<-[1..]]
[2, 4, 6, 8, 10 ...

> [x | x<-[1..], x`mod`2/=0]
[1, 3, 5, 7, 9 ...
```

# Produzir listas infinitas (cont.)

Algumas funções do prelúdio-padrão produzem especificamente listas infinitas:

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

cycle :: [a] -> [a]
-- cycle xs = xs++xs++xs++...

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

(Note que *iterate* é de ordem-superior porque o primeiro argumento é uma função.)

# Produzir listas infinitas (cont.)

Podemos testar no interpretador pedido prefixos finitos:

```
> take 10 (repeat 1)
[1,1,1,1,1,1,1,1,1,1]

> take 10 (repeat 'a')
"aaaaaaaaaa"

> take 10 (cycle [1,-1])
[1,-1,1,-1,1,1,-1,1,-1,1]

> take 10 (iterate (2*) 1)
[1,2,4,8,16,32,64,128,256,512]
```

# Produzir listas infinitas (cont.)

As funções *repeat*, *cycle* e *iterate* estão definidas no prelúdio-padrão usando recursão:

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

cycle :: [a] -> [a]
cycle [] = error "empty list"
cycle xs = xs' where xs' = xs++xs'

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

# Porquê usar listas infinitas?

- Permite simplificar o processamento de listas finitas combinando-as com listas infinitas
- Permite separar a geração e o consumo de sequências
- Permite maior modularidade na decomposição dos programas

# Exemplo 1: Preenchimento de texto

Escrever uma função

```
preencher :: Int -> String -> String
```

que preenche uma cadeia com espaços de forma a perfazer *n* caracteres.

Se a cadeia já tiver comprimento *n* ou maior, deve ser truncada a *n* caracteres.

# Exemplo 1: Preenchimento de texto (cont.)

## Exemplos

```
> preencher 10 "Haskell"
"Haskell   "

> preencher 10 "Haskell B. Curry"
"Haskell B."
```

# Exemplo 1: Preenchimento de texto (cont.)

- Uma solução que calcula e acrescenta o número correto de espaços testando uma condição:

```
preencher n xs
    | k < n     = xs ++ replicate (n-k) ' '
    | otherwise = take n xs
    where k = length xs
```

- Mas há uma solução mais simples usando *take* e uma lista infinita:

```
preencher n xs = take n (xs++repeat ' ')
```

# Exemplo 2: Aproximação da raiz quadrada

Calcular uma aproximação de $\sqrt{q}$ pelo *método babilónico*:

1. Começamos com $x_0 = q$
2. Em cada passo, melhoramos a aproximação tomando

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{q}{x_n}\right)$$

3. Critérios de paragem:

   número de iterações parar ao fim de um certo número de iterações

   erro absoluto parar quando a distância entre aproximações é inferior a $\epsilon$:

$$|x_{n+1} - x_n| < \epsilon$$

# Exemplo 2: Aproximação da raiz quadrada (cont.)

```haskell
-- sucessão infinita de aproximações à raiz quadrada
aproximações :: Double -> [Double]
aproximações q = iterate (\x->0.5*(x+q/x)) q

-- critério de paragem por erro absoluto
erroAbsoluto :: [Double] -> Double -> Double
erroAbsoluto xs eps
  = head [x' | (x,x')<-zip xs (tail xs), abs(x-x')<eps]
```

# Exemplo 2: Aproximação da raiz quadrada (cont.)

## Exemplos para calcular $\sqrt{2}$

```
> aproximações 2.0
[2.0,1.5,1.4166667,1.4142157, 1.4142135, 1.4142135, ...

> aproximações 2.0 !! 5
1.4142135

> (aproximações 2.0) `erroAbsoluto` 0.01
1.4166667

> (aproximações 2.0) `erroAbsoluto` 0.001
1.4142135
```

# Exemplo 3: A sucessão de Fibonacci

A sucessão de Fibonacci:

- começa com 0, 1;
- cada valor seguinte é a *soma dos dois anteriores*.

$$0 : 1 : 1 : 2 : 3 : 5 : 8 : 13 : \ldots : a : b : a{+}b : \ldots$$

# Exemplo 3: A sucessão de Fibonacci (cont.)

Solução em Haskell: uma lista infinita definida recursivamente.

```
fibs :: [Integer]
fibs = 0 : 1 : [a+b | (a,b)<-zip fibs (tail fibs)]
```

Alternativa usando *zipWith* em vez de lista em compreensão (ver folha de exercícios):

```
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

# Exemplo 3: A sucessão de Fibonacci (cont.)

Os primeiros dez números de Fibonacci:

```
> take 10 fibs
[0,1,1,2,3,5,8,13,21,34]
```

O nono número Fibonacci (índices começam em zero):

```
> fibs!!8
21
```

O primeiro Fibonacci superior a 100:

```
> head (dropWhile (<=100) fibs)
144
```

# Exemplo 4: O crivo de Eratóstenes

Gerar *todos* os números primos usando o *crivo de Eratóstenes*.

1. Começar com a lista $[2, 3, 4, \ldots]$;
2. Marcar o primeiro número $p$ na lista como primo;
3. Remover da lista $p$ e todos os seus múltiplos;
4. Repetir o passo 2.

Observar que o passo 3 envolve processar uma lista infinita.

# Exemplo 4: O crivo de Eratóstenes (cont.)

## Em Haskell

```
primos :: [Integer]
primos = crivo [2..]

crivo :: [Integer] -> [Integer]
crivo (p:xs) = p : crivo [x | x<-xs, x`mod`p/=0]
```

# Exemplo 4: O crivo de Eratóstenes (cont.)

Os primeiros 10 primos:

```
> take 10 primos
[2,3,5,7,11,13,17,19,23,29]
```

Quantos primos são inferiores a 100?

```
> length (takeWhile (<100) primos)
25
```