Programação Funcional
6ª Aula — Funções de ordem superior

PROF. BONFIM AMARO JUNIOR

bonfimamaro@ufc.br

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:

```
applyTwice :: (a -> a) -> a -> a
applyTwice f x = f (f x)
```

```
ghci> applyTwice (+3) 10
16
ghci> applyTwice (++ " HAHA") "HEY"
"HEY HAHA HAHA"
ghci> applyTwice ("HAHA " ++) "HEY"
"HAHA HAHA HEY"
ghci> applyTwice (multThree 2 2) 9
144
ghci> applyTwice (3:) [1]
[3,3,1]
```

Porquê ordem superior?

- Permite definir padrões de computação comuns que podem ser facilmente re-utilizados.
- Facilita a definição de bibliotecas para domínios específicos:
  - processamento de listas;
  - formatação de texto;
  - “parsing”;
  - . . .
- Podemos provar propriedades gerais das funções de ordem superior que são válidas em qualquer use específico.

A função *map*

A função *map* aplica uma função a cada elemento da 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*

Exercício: Defina a Função map usando :

a) Lista por Compressão

b) Recursão

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
```

Esta forma será útil para provar propriedades usando indução.

Função *filter*

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

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

Exemplos:

```
> filter even [1..10]
[2,4,6,8,10]

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

Função *filter*

Exercício: Defina a Função filter usando :

a) Lista por Compressão

b) Recursão

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 de uma 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.)

Exercício: Defina as funções:

a) takeWhile

b) dropWhile

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

Definições recursivas de *takeWhile* e *dropWhile* (do prelúdio-padrão):

```
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 de uma lista.

any verifica se um predicado é verdadeiro para algum elementos de uma lista.

As duas funções têm o mesmo tipo:

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

As funções *all* e *any* (cont.)

Exemplos:

```
> all even [2,4,6,8]
True

> any odd [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)
```

A função *foldr*

Muitas funções sobre listas seguem o seguinte padrão de definição recursiva:

```
f []     = z
f (x:xs) = x ⊕ 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 $f\ xs$.

A função *foldr* (cont.)

Exemplos:

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

product []     = 1                z = 1
product (x:xs) = x * product xs   ⊕ = *

and []     = True                 z = True
and (x:xs) = x && and xs          ⊕ = &&

or []     = False                 z = False
or (x:xs) = x || or xs            ⊕ = ||

length []     = 0                 z = 0
length (x:xs)= 1 + length xs      ⊕ = \x n -> 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 (\x n->n+1) 0
```

A função *foldr* (cont.)

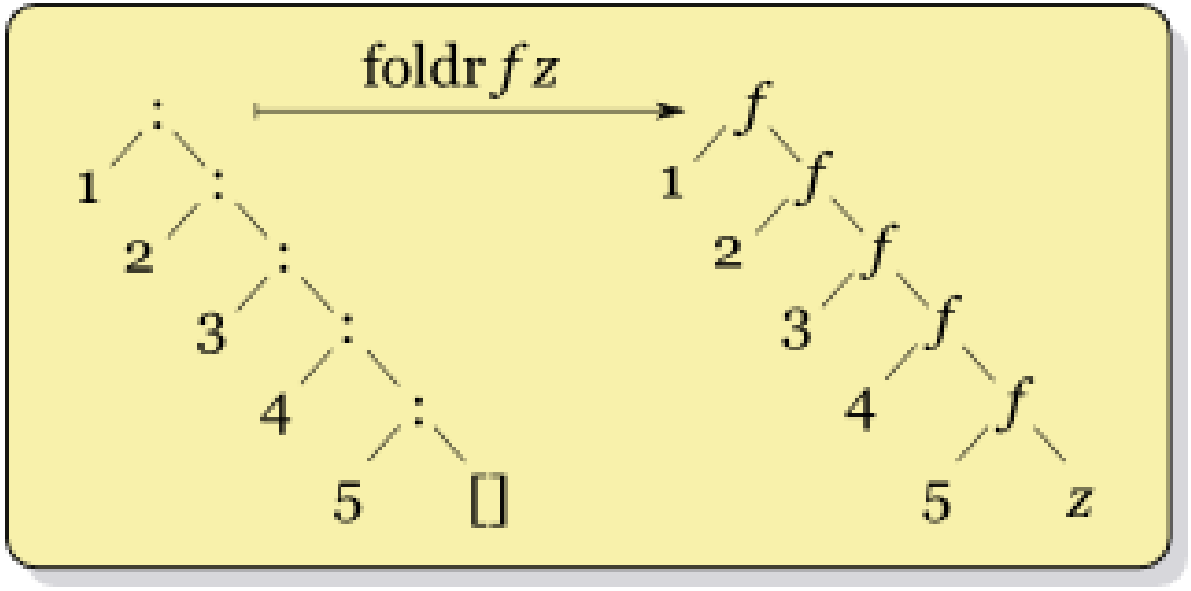
Definição recursiva de *foldr* (do prelúdio-padrão):

```
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.)

Podemos visualizar *foldr f z* como a transformação que substitui:

- cada (:) por *f*;
- [ ] por *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"):

$$\mathit{foldr}\ (\oplus)\ v\ [x_1, x_2, \ldots, x_n] = x_1 \oplus (x_2 \oplus (\ldots (x_n \oplus v) \ldots))$$

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

$$\mathit{foldl}\ (\oplus)\ v\ [x_1, x_2, \ldots, x_n] = ((\ldots ((v \oplus x_1) \oplus x_2) \ldots) \oplus x_n)$$

A função *foldl* (cont.)

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

sum = foldl (+) 0

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

sum = foldr (+) 0

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

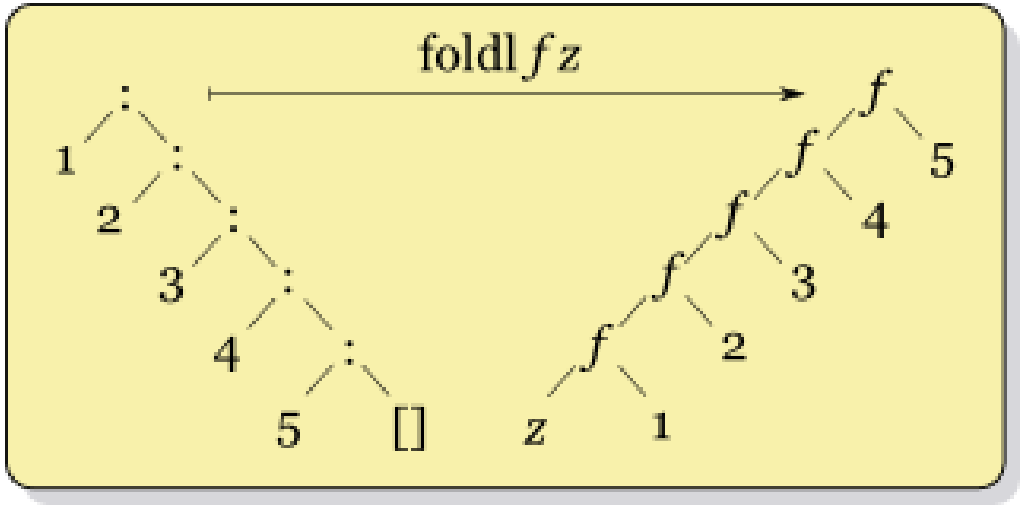
A função *foldl* (cont.)

Tal como *foldr*, a função *foldl* está definida no prelúdio-padrão usando 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.)

No entanto, pode ser mais fácil visualizar *foldl* como uma transformação sobre listas:



Fonte:

`http://en.wikipedia.org/wiki/Fold_(higher-order_function).`

Outras funções de ordem superior

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
```

Outras funções de ordem superior (cont.)

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

Exemplo:

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

é equivalente a

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