

Linguagens de Programação

Programação Funcional e Haskell
Funções de Alta-Ordem

Thiago Alves

Introdução

Uma função é chamada de alta-ordem se ela recebe uma função como argumento ou retorna uma função como resultado.

```
add x y :: Int → (Int → Int)  
add x y = \x → (\y → x+y)
```



Funções currificadas retornam
funções como resultado.

Introdução

Uma função é chamada de alta-ordem se ela recebe uma função como argumento ou retorna uma função como resultado.

```
twice :: (a → a) → a → a  
twice f x = f (f x)
```

twice é de alta-ordem pois
recebe uma função como
primeiro argumento.

Introdução

Uma função é chamada de alta-ordem se ela recebe uma função como argumento ou retorna uma função como resultado.

```
> twice (*2) 3  
12
```

A Função Map

A função de alta-ordem map aplica uma função para todos elementos de uma lista.

```
map :: (a → b) → [a] → [b]
```

Exemplo:

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

A função map pode ser definida de uma maneira simples usando compreensão de listas:

```
map f xs = [f x | x ← xs]
```

Também pode ser definida usando recursão:

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

A função map pode ser aplicada nela mesma para processar lista de listas:

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

A Função Filter

A função de alta-ordem filter seleciona todo elemento da lista que satisfaz um predicado.

```
filter :: (a → Bool) → [a] → [a]
```

Exemplo:

```
> filter even [1..10]  
[2,4,6,8,10]  
> filter (/= ' ') "abc def"  
"abcdef"
```


Filter pode ser definida usando compreensão de listas:

```
filter p xs = [x | x ← xs, p x]
```

Pode ser definida usando recursão:

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

Podemos definir uma função que retorna a soma dos quadrados dos inteiros pares de uma lista:

```
sumsqreven :: [Int] → Int
```

Podemos definir uma função que retorna a soma dos quadrados dos inteiros pares de uma lista:

```
sumsqreven :: [Int] → Int  
sumsqreven ns = sum (map (^2) (filter even ns))
```

Vamos definir a função all que decide se todo elemento de uma lista satisfaz um predicado dado como argumento.

```
all :: (a → Bool) → [a] → Bool
```

Exemplo:

```
> all even [2,4,6,8,10]  
True
```

Vamos definir a função all que decide se todo elemento de uma lista satisfaz um predicado dado como argumento.

```
all :: (a → Bool) → [a] → Bool  
all p xs = and [p x | x ← xs]
```

Exemplo:

```
> all even [2,4,6,8,10]  
True
```

Vamos definir a função any que decide se pelo menos um elemento satisfaz um predicado.

```
any :: (a → Bool) → [a] → Bool
```

Exemplo:

```
> any (== ' ') "abc def"  
True
```

Vamos definir a função any que decide se pelo menos um elemento satisfaz um predicado.

```
any :: (a → Bool) → [a] → Bool  
any p xs = or [p x | x ← xs]
```

Exemplo:

```
> any (== ' ') "abc def"  
True
```

Vamos definir a função takeWhile que seleciona elementos de uma lista enquanto um predicado vale.

```
takeWhile :: (a → Bool) → [a] → [a]
```

Exemplo:

```
> takeWhile (/= ' ') "abc def"  
"abc"
```


Vamos definir a função takeWhile que seleciona elementos de uma lista enquanto um predicado vale.

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

Exemplo:

```
> takeWhile (/= ' ') "abc def"
"abc"
```

Vamos definir a função dropWhile que remove elementos enquanto um predicado vale.

```
dropWhile :: (a → Bool) → [a] → [a]
```

Exemplo:

```
> dropWhile (== ' ') " abc"  
"abc"
```

Vamos definir a função dropWhile que remove elementos enquanto um predicado vale.

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

Exemplo:

```
> dropWhile (== ' ') " abc"
"abc"
```

A Função Foldr

Várias funções em listas podem ser definidas usando o seguinte padrão simples de recursão:

$$f [] = v$$

$$f (x:xs) = x \oplus f xs$$

f mapeia a lista vazia para v , e qualquer lista não-vazia para a aplicação da função \oplus na cabeça e no resultado de f na sua cauda.

Por exemplo:

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

$v = 0$

$\oplus = +$

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

$v = 1$

$\oplus = *$

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

$v = \text{True}$

$\oplus = \&\&$

A função de alta-ordem foldr (fold right) encapsula esse padrão simples de recursão, com a função \oplus e o valor v como argumentos.

Exemplo:

```
sum = foldr (+) 0
```

```
product = foldr (*) 1
```

```
and = foldr (&&) True
```

foldr pode ser definida usando recursão:

```
foldr :: (a → b → b) → b → [a] → b
```

```
foldr f v [] = v
```

```
foldr f v (x:xs) = f x (foldr f v xs)
```

Entretanto, é melhor pensar em foldr não-recursivamente, como uma troca simultânea de cada $(:)$ em uma lista pela função dada, e $[]$ pelo valor dado.

Exemplo:

`sum [1,2,3]`
=
`foldr (+) 0 [1,2,3]`
=
`foldr (+) 0 (1:(2:(3:[])))`
=
`1+(2+(3+0))`
=
`6`

Troque cada `(:)`
por `(+)` e `[]` por
`0`.

Seja a função length:

```
length :: [a] → Int  
length []      = 0  
length (_:xs) = 1 + length xs
```

Vamos definir a função length usando a função foldr

Por exemplo:

`length [1,2,3]`
=
`length (1:(2:(3:[])))`
=
`1+(1+(1+0))`
=
`3`

Troque cada
(:) por $\lambda_n \rightarrow$
 $1+n$ e [] por 0.

Então, temos:

`length = foldr ($\lambda_n \rightarrow 1+n$) 0`

Seja a função reverse:

```
reverse []      = []  
reverse (x:xs) = reverse xs ++ [x]
```

Vamos definir a função reverse usando a função foldr

Seja a função reverse:

```
reverse [] = []  
reverse (x:xs) = reverse xs ++ [x]
```

Por exemplo:

```
reverse [1,2,3]  
=  
reverse (1:(2:(3:[])))  
=  
(([] ++ [3]) ++ [2]) ++ [1]  
=  
[3,2,1]
```

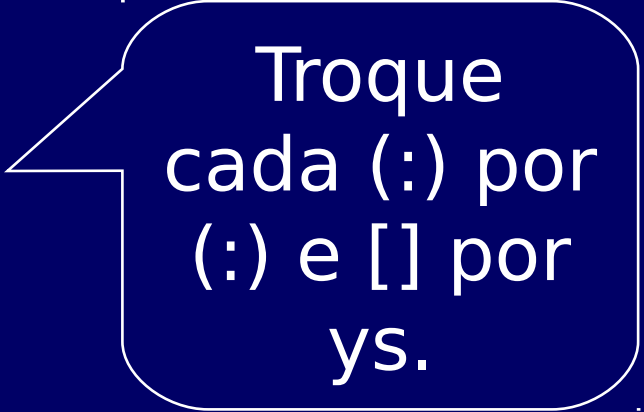
Troque cada (:) por
 $\lambda x \text{ xs} \rightarrow \text{xs} ++ [x]$
e [] por [].

Então, temos:

```
reverse = foldr ( $\lambda x\ xs \rightarrow xs ++ [x]$ ) []
```

Finalmente, podemos notar que a função concatena (`++`) tem uma definição compacta usando `foldr`:

```
(++ ys) = foldr (:) ys
```



Troque
cada `(:)` por
`(:)` e `[]` por
`ys`.

Operador de Composição

A função $(.)$ retorna a composição de duas funções como uma única função.

$$\begin{aligned} (.) &:: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c) \\ f . g &= \lambda x \rightarrow f (g x) \end{aligned}$$

Também pode ser vista como:

$$f . g \ x = f (g \ x)$$

Pode ser usada para simplificar aplicações aninhadas de funções

```
odd :: Int → Bool  
odd n = not (even x)
```

```
odd = not . even
```


Vamos definir a função `twice` usando composição

```
twice :: (a → a) → a → a  
twice f x = f (f x)
```

Vamos definir a função `twice` usando composição

```
twice :: (a → a) → a → a  
twice f x = f (f x)
```

```
twice f = f . f
```

Vamos definir a função abaixo usando composição

```
sumsqreven :: [Int] → Int
```

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

Vamos definir a função abaixo usando composição

```
sumsqreven :: [Int] → Int  
sumsqreven xs = sum (map (^2) (filter even xs))
```

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

Vamos definir a função que retorna a composição de uma lista de funções

```
compose :: [a → a] → (a → a)
```

Vamos definir a função que retorna a composição de uma lista de funções

```
compose :: [a → a] → (a → a)  
compose = foldr (.) (\x → x)
```