Funções de ordem superior

- Funções de ordem superior
- Um operador para aplicação de função
- Composição de funções
- Funções pré-definidas de ordem superior

Funções de ordem superior

- Uma função de ordem superior é uma função que
  - tem outra função como argumento, ou
  - produz uma função como resultado.
- No paradigma funcional, funções são valores de primeira classe.
- O que são valores de primeira classe ?

Um operador para aplicação de função

- O operador ($) definido no prelúdio se destina a substituir a aplicação de função normal

- O operador ($) é usado principalmente para eliminar o uso de parênteses nas aplicações de funções

- fun (x) ———- fun x ——— fun $ x

Um operador para aplicação de função

- O operador ($) apresenta uma precedência e associatividade diferente para ajudar a evitar parênteses.
- O operador ($) tem precedência zero e associa-se à direita.
- Já a aplicação de função normal tem precedência maior que todos os operadores e associa- se à esquerda.

Um operador para aplicação de função

- Exemplos de aplicação de função com ($)
- sqrt 36 —-> 6.0
- sqrt $ 36 —> 6.0
- ($) sqrt 36 —-> 6.0
- even (succ (abs (negate 36))) —-> False
- even $ succ $ abs $ negate 36 —-> False
- head (tail “asdf”) ——> ‘s’
- head $ tail $ “asdf” ——> ‘s’
- head $ tail “asdf” ——-> ’s’

Um operador para aplicação de função

- Definição de ($) :
- infixr 0 $
- ($) :: (a -> b) -> a -> b
- f $ x = f x

- ($) sqrt 36

Composição de funções

- Composição de funções é uma operação comum na Matemática.
- Dadas duas funções f e g, a função composta
- $f \circ g$ é definida por
- $(f \circ g)(x) = f(g(x))$
- Ou seja, quando a função composta $f \circ g$ é aplicada a um argumento x, primeiramente g é aplicada a x, e em seguida f é aplicada a este resultado gx.

Composição de funções

- A operação de composição de funções faz parte do prelúdio de Haskell.
- A função (.) recebe duas funções como argumento e resulta em uma terceira função que é a composição das duas funções dadas.
  - sqrt . abs
- A função (.) é um operador binário infixo de precedência e associatividade à esquerda.
- Observe que a operação (.) é uma função de ordem superior, pois recebe duas funções como argumento e resulta em outra função.

Composição de funções

- Exemplos de composição de funções

```
sqrt . abs                    ⇝ a função composta de sqrt e abs
(sqrt . abs) 9                ⇝ 3
(sqrt . abs) (16 - 25)        ⇝ 3
(sqrt . abs . sin) (3*pi/2)   ⇝ 1.0
(not . null) "abc"            ⇝ True
(sqrt . abs . snd) ('Z',-36)  ⇝ 6
```

Composição de funções

- Definição de (.)

```
infixr 9 .

(.) :: (b -> c) -> (a -> b) -> a -> c

f . g = h
  where h x = f (g x)
```

- sqrt . abs

Funções pré-definidas de ordem superior

- Existem funções pré-definidas em Haskell que servem para resolver folding, filtering e mapping.
- folding é a colocação de um operador entre os elementos de uma lista,
- filtering significa filtrar alguns elementos de uma lista
- mapping é a aplicação de funções a todos os elementos de uma lista.
- Os outros casos são combinações destes três, ou recursões primitivas.

Funções pré-definidas de ordem superior

- As principais são as funções foldr, map, e filter.
- Estas funções são todas polimórficas, ou seja, servem para listas de qualquer tipo
- e são também exemplos de funções de ordem superior

A função filter

- A função filter do prelúdio recebe uma função e uma lista como argumentos,
- e seleciona (**filtra**) os elementos da lista para os quais a função dada resulta em verdadeiro.
- filter even [1, 2, 6, 8, 11, 13] —-> [2, 6, 8]
- Note que filter é uma função de ordem superior, pois recebe outra função como argumento.

A função filter

- Exemplos de aplicação de filter

```
filter even [1,8,10,48,5,-3]              ⇝ [8,10,48]
filter odd [1,8,10,48,5,-3]               ⇝ [1,5,-3]
filter isDigit "A186 B70"                 ⇝ "18670"
filter (not . null) ["abc","","ok",""] ⇝ ["abc", "ok"]
```

Importando um módulo

- A função isDigit não faz parte do módulo Prelude, mas está definida no módulo Data.Char.
- Para usar isDigit é necessário importar o módulo Data.Char:
- no ambiente interativo use o comando: module (ou simplesmente: m):
- :m + Data.Char
- em um script e no ambiente interativo use a declaração
- import Data.Char

A função filter

- Definição de filter

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

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

A função map

- A função map do prelúdio recebe uma função e uma lista como argumentos,
- e **aplica** a função a cada um dos elementos da lista, resultando na lista dos resultados.
- map even [1, 4, 5, 11] —-> [False, True, False, False]
- map é uma função de ordem superior, pois recebe outra função como argumento.

A função map

- Exemplos de aplicação de map

```
map sqrt [0,1,4,9]                                ⇝ [0.0,1.0,2.0,3.0]
map succ "HAL"                                    ⇝ "IBM"
map head ["bom","dia","turma"]                    ⇝ "bdt"
map even [8,10,-3,48,5]                           ⇝ [True,True,False,True,False]
map isDigit "A18 B7"                              ⇝ [False,True,True,False,False,True]
map length ["ciênca", "da", "computação"]         ⇝ [6,2,10]
map (sqrt.abs.snd) [('A',100),('Z',-36)]          ⇝ [10,6]
```

A função map

- Definição de map

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

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

A função zipWith

- A função zipWith recebe uma **função binária** e duas listas e retorna a lista formada pelos resultados da aplicação da função aos elementos correspondentes da listas dadas.
- zipWith (+) [1,2,4,5] [3,6,5,6] —-> [4,8,9,11]
- Se as listas forem de tamanhos diferentes, o tamanho do resultado é o menor tamanho.
- Observe que zipWith é uma função de ordem superior, pois recebe outra função como argumento.

A função zipWith

- Exemplos de aplicação de zipWith

```
zipWith (+) [] []                          ⇝ []
zipWith (+) [1,2,3,4,5] [3,3,4,1,5]        ⇝ [4,5,7,5,10]
zipWith (++) ["AB","cde"] ["?","123"]      ⇝ ["AB?","cd123"]
zipWith (^) [5,6,7,8] [2,3,4,5]            ⇝ [25,216,2401,32768]
zipWith (*) [5,6,7,8] [2,3]                ⇝ [10,18]
```

A função zipWith

- Definição de zipWith

```
zipWith                  :: (a -> b -> c) -> [a] -> [b] -> [c]

zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
zipWith _ _      _      = []
```

A função foldl

- A função foldl reduz uma lista, usando uma **função binária** e um **valor inicial**, de forma associativa à esquerda.

$$\texttt{foldl}\ (\oplus)\ e\ [x_0, x_1, \ldots, x_{n-1}]$$
$$\equiv$$
$$(\ldots((e \oplus x_0) \oplus x_1)\ \ldots) \oplus x_{n-1}$$

A função foldl

- Exemplos de aplicação de foldl

```
foldl (+) 0 []                                  ⇝ 0
foldl (+) 0 [1]                                 ⇝ 1
foldl (+) 0 [1,2]                               ⇝ 3
foldl (+) 0 [1,2,4]                             ⇝ 7
foldl (*) 1 [5,2,4,10]                          ⇝ 400
foldl (&&) True [2>0,even 6,odd 5,null []]      ⇝ True
foldl (||) False [2>3,even 6,odd 5,null []]     ⇝ True
```

A função foldl

- Definição de foldl

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

foldl f z []       = z
foldl f z (x:xs)   = foldl f (f z x) xs
```

A função foldr

- A função foldr reduz uma lista, usando uma função binária e um valor inicial, de forma associativa à direita.

$$\texttt{foldr}\ (\oplus)\ e\ [x_0, \ldots, x_{n-2}, x_{n-1}]$$
$$\equiv$$
$$x_0 \oplus (\ldots\ (x_{n-2} \oplus (x_{n-1} \oplus e))\ \ldots)$$

- foldr (+) 0 [1, 2, 4] —-> 7

A função foldr

- Exemplos de aplicação de foldr

```
foldr (+) 0 []                                  ⇝ 0
foldr (+) 0 [1]                                 ⇝ 1
foldr (+) 0 [1,2]                               ⇝ 3
foldr (+) 0 [1,2,4]                             ⇝ 7
foldr (*) 1 [5,2,4,10]                          ⇝ 400
foldr (&&) True [2>0,even 6,odd 5,null []]      ⇝ True
foldr (||) False [2>3,even 6,odd 5,null []]     ⇝ True
```

A função foldr

- Definição de foldr

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

foldr f z []        = z
foldr f z (x:xs) = f x (foldr f z xs)
```

A função foldl1

- A função foldl1 reduz uma lista não vazia usando uma função binária, de forma associativa à esquerda.
- foldl1 é uma variante de foldl que não tem valor inicial, e portanto deve ser aplicada a listas não-vazias.
- foldl1 (+) [1, 2, 4] —-> 7

A função foldl1

- Exemplos de aplicação de foldl1

```
foldl1 (+) []                                  ⇝ erro
foldl1 (+) [1]                                 ⇝ 1
foldl1 (+) [1,2,4]                             ⇝ 7
foldl1 (*) [5,2,4,10]                          ⇝ 400
foldl1 (&&) [2>0,even 6,odd 5,null []]         ⇝ True
foldl1 max [1,8,6,10,-48,5]                    ⇝ 10
```

A função foldl1

- Definição de foldl1

```
foldl1          :: (a -> a -> a) -> [a] -> a

foldl1 f (x:xs) = foldl f x xs
```

A função foldr1

- A função foldr1 reduz uma lista não vazia usando uma função binária, de forma associativa à esquerda.
- foldr1 é uma variante de foldr que não tem valor inicial, e portanto deve ser aplicada a listas não-vazias.

A função foldr1

- Exemplos de aplicação de foldr1

```
foldr1 (+) []                                  ⇝ erro
foldr1 (+) [1]                                 ⇝ 1
foldr1 (+) [1,2,4]                             ⇝ 7
foldr1 (*) [5,2,4,10]                          ⇝ 400
foldr1 (&&) [2>0,even 6,odd 5,null []]         ⇝ True
foldr1 max [1,8,6,10,-48,5]                    ⇝ 10
```

A função foldr1

- Definição de foldr1

```
foldr1          :: (a -> a -> a) -> [a] -> a

foldr1 _ [x]     = x
foldr1 f (x:xs) = f x (foldr1 f xs)
```