Funções de ordem superior

- **break** é uma função do Prelude equivalente à função span invocada com a condição negada.

```
break :: (a -> Bool) -> [a] -> ([a],[a])
break p l = span (not . p) l
```

```
> break (>10) [3,4,5,30,8,12,9]
([3,4,5],[30,8,12,9])
```

Exemplo: A função **words** (do Prelude) que parte uma string numa lista de palavras.

```
> words "  \nabds\tbfsas\n26egd\n\n3673qw"
[“abds","bfsas","26egd","3673qw"]
```

```
words :: String -> [String]
words [] = []
words s = let l = dropWhile isSpace s
              (a,b) = break isSpace l
          in a : words b
```


foldr *(right fold)*

Considere as seguintes funções:

Estas funções fazem coisas distintas entre si, mas **a forma como operam é semelhante**: aplicam um operador binário à cabeça da lista e ao resultado de aplicar a função à cauda da lista, e quando a lista é vazia devolvem um determinado valor.

```
and [] =  True
and (b:bs) = b && (and bs)
```

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

Estas funções têm um **padrão de computação** comum. Apenas diferem no operador binário que é usado e no valor a devolver quando a lista é vazia.

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

```
concat [] = []
concat (l:ls) = l ++ (concat ls)
```

A função **foldr** do Prelude sintetiza este padrão de computação, abstraindo em relação ao operador binário que é usado e ao resultado a devolver quando a lista é vazia.

foldr *(right fold)*

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

foldr é uma função de ordem superior que recebe o operador **f** que é usado para construir o resultado, e o valor **z** a devolver quando a lista é vazia.

Exemplos:

```
and :: [Bool] -> Bool
and l = foldr (&&) True l
```

```
product :: Num a => [a] -> a
product xs = foldr (*) 1 xs
```

```
sum :: Num a => [a] -> a
sum l = foldr (+) 0 l
```

```
concat :: [[a]] -> [a]
concat l = foldr (++) [] l
```

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

Note que `foldr f z [x1,…,xn] == f x1 (… (f xn z)…)`
`== x1 `f` (… (xn `f` z)…)`
Ou seja, aplica `f` **associando à direita**.

foldr

Podemos olhar para a expressão `(foldr f z l)` como a substituição de cada `(:)` da lista por `f` , e de `[]` por `z`.

```
foldr f z (x1:x2:…:xn:[]) == x1 `f` (x2 `f` (… `f` (xn `f` z) …))
```

O resultado vai sendo construído a partir do lado direito da lista.

Exemplos:

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

```
foldr (*) 1 [1,2,3] == 1 * (2 * (3 * 1)))
```

```
foldr (&&) True [False,True,False] == False && (True && (False && True)))
```

```
foldr (++) [] [“abc”,“zzzz”,“bb”] == “abc” ++ (“zzzz” ++ (“bb” ++ []))
```

foldr

Muitas funções (mais do que à primeira vista poderia parecer) podem ser definidas usando o `foldr`.

Exemplo:

```
reverse :: [a] -> [a]
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]
```

`x` representa a cabeça da lista e `r` o resultado da chamada recursiva sobre a cauda.

Pode ser definida assim:

```
reverse l = foldr (\x r -> r++[x]) [] l
```

Exemplo:

```
length :: [a] -> Int
length [] = 0
length (x:xs) = 1 + length xs
```

`h` representa a cabeça da lista e `r` o resultado da chamada recursiva sobre a cauda.

Pode ser definida assim:

```
length = foldr (\h r -> 1+r) 0
```


foldl *(left fold)*

Existe no `Prelude` uma outra função, **`foldl`**, que vai construindo o resultado pelo lado esquerdo da lista.

```
foldl f z [x1,x2,…,xn] == (… ((z `f` x1) `f` x2) …) `f` xn
```

Exemplo:

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

A função **`foldl`** sintetiza um padrão de computação que corresponde a trabalhar com um acumulador. O `foldl` recebe como argumentos a função que combina o acumulador com a cabeça da lista, e o valor inicial do acumulador.

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

`z` é o acumulador. `f` é usado para combinar o acumulador com a cabeça da lista.

`(f z x)` é o novo valor do acumulador.

foldl *(left fold)*

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

`z` é o acumulador. `f` é usado para combinar o acumulador com a cabeça da lista. `(f z x)` é o novo valor do acumulador.

Exemplo: Vejamos a relação entre função somatório implementada com um acumulador

```
sumAc [] ac = ac
sumAc (x:xs) ac = sumAc xs (ac+x)
```

```
sumAc [1,2,3] 0 = sumAc [2,3] (0+1)
                = sumAc [3] ((0+1)+2)
                = sumAc [] (((0+1)+2)+3)
                = ((0+1)+2)+3
                = 6
```

e o somatório implementado com um `foldl`

```
foldl (+) 0 [1,2,3] = foldl (+) (0+1) [2,3]
                    = foldl (+) ((0+1)+2) [3]
                    = foldl (+) (((0+1)+2)+3) []
                    = ((0+1)+2)+3
                    = 6
```


foldl

Muitas funções (mais do que à primeira vista poderia parecer) podem ser definidas usando o `foldl`.

Exemplo:

```
inverte :: [a] -> [a]
inverte l = inverteAc l []
  where inverteAc [] ac = ac
        inverteAc (x:xs) ac = inverteAc xs (x:ac)
```

Pode ser definida assim:

```
inverte l = foldl (\ac x -> x:ac)) [] l
```

`ac` representa o valor acumulado e `x` a cabeça da lista. `[]` é o valor inicial do acumulador.

Ou assim:

```
inverte l = foldl (flip (:)) [] l
```

Exemplo: A função `stringToInt :: String -> Int` definida anteriormente, com um parâmetro de acumulação, pode ser definida de forma equivalente assim:

```
stringToInt l = foldl (\ac x -> 10*ac + digitToInt x) 0 l
```

foldr vs foldl

Note que as expressões **(foldr f z xs)** e **(foldl f z xs)** só darão o mesmo resultado se a função **f** for comutativa e associativa, caso contrário dão resultados distintos.

Exemplo:

```
foldr (-) 8 [4,7,3,5] = 4 – (7 – (3 – (5 - 8)))
                      = 3
```

```
foldl (-) 8 [4,7,3,5] = (((8 - 4) – 7) – 3) – 5
                      = -11
```


Tipos algébricos

A construção de tipos algébricos dá à linguagem Haskell um enorme poder expressivo, pois permite a implementação de tipos enumerados, co-produtos (união disjunta de tipos), e tipos indutivos (recursivos).

O tipo das listas é um exemplo de um tipo indutivo (recursivo) :

```
data [a] = []
         | (:) a [a]
```

Uma lista,

- ou é vazia,
- ou tem um elemento e uma sub-estrutura que é também uma lista.

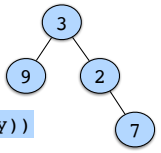
```
[1,2,3] = 1 : [2,3] = 1 : 2 : [3] = 1 : 2 : 3 : []
```

A noção de árvore binária expande este conceito.

Uma árvore binária,

- ou é vazia,
- ou tem um elemento e duas sub-estruturas que são também árvores.

```
data BTree a = Empty
             | Node a (BTree a) (BTree a)
```




```
Node 3 (Node 9 Empty Empty) (Node 2 Empty (Node 7 Empty Empty))
```


Árvores binárias

As árvores binárias são estruturas de dados muito úteis para organizar a informação.

```
data BTree a = Empty
             | Node a (BTree a) (BTree a)
  deriving (Show)
```

Os construtores da árvores são:

```
Empty :: BTree a

Node :: a -> (BTree a) -> (BTree a) -> (BTree a)
```

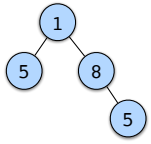
Empty representa a árvore vazia.

Node recebe um elemento e duas árvores, e constrói a árvore com esse elemento na raiz, uma árvore do lado esquerdo e outra do lado direito.

```
arv1 = Node 5 Empty Empty
```

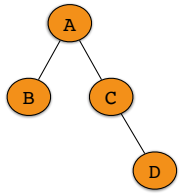

```
arv2 = Node 8 Empty arv1
```


```
arv3 = Node 1 arv1 arv2
```



Árvores binárias

Terminologia



- O nodo A é a **raiz** da árvore.
- Os nodos B e C são **filhos** (ou **descendentes**) de A.
- O nodo C é **pai** de D.
- B e D são **folhas** da árvore.
- O **caminho** ***(path)*** de um nodo é a sequência de nodos da raiz até esse nodo. Por exemplo, A,C,D é o caminho para o modo D.
- A **altura** da árvore é o comprimento do caminho mais longo. Esta árvore tem altura 3.

Árvores binárias

As funções definidas sobre tipos de dados recursivos, são geralmente funções recursivas, com padrões de recursividade semelhantes aos dos tipos de dados.

Exemplo: Calcular o número de nodos que tem uma árvore.

```
conta :: BTree a -> Int
conta Empty = 0
conta (Node x e d) = 1 + conta e + conta d
```

Exemplo: Somar todos de nodos de uma árvore de números .

```
sumBT :: Num a =>  BTree a -> a
sumBT Empty = 0
sumBT (Node x e d) = x + sumBT e + sumBT d
```

```
> sumBT (Node 2 Empty (Node 7 Empty Empty))
        = 2 + (sumBT Empty) + sumBT (Node 7 Empty Empty)
        = 2 + 0 + (7 + sumBT Empty + sumBT Empty)
        = 2 + 0 + (7 + 0 + 0)
        = 9
```


Árvores binárias

Exemplo: Calcular a altura de uma árvore.

```
altura :: BTree a -> Int
altura Empty = 0
altura (Node _ e d) = 1 + max (altura e) (altura d)
```

Exemplos: As funções map e zip para árvores binárias.

```
mapBT :: (a -> b) -> BTree a -> BTree b
mapBT f Empty = Empty
mapBT f (Node x e d) = Node (f x) (mapBT f e) (mapBT f d)
```

```
zipBT :: BTree a -> BTree b -> BTree (a,b)
zipBT (Node x1 e1 d1) (Node x2 e2 d2) =
                              Node (x1,x2) (zipBT e1 e2) (zipBT d1 d2)
zipBT _ _ = Empty
```


Travessias de árvores binárias

Uma árvore pode ser percorrida de várias formas. As principais estratégias são:

Travessia preorder: visitar a raiz, depois a árvore esquerda e a seguir a árvore direita.

```
preorder :: BTree a -> [a]
preorder Empty = []
preorder (Node x e d) = [x] ++ (preorder e) ++ (preorder d)
```

Travessia inorder: visitar árvore esquerda, depois a raiz e a seguir a árvore direita.

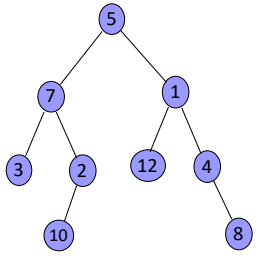
```
inorder :: BTree a -> [a]
inorder Empty = []
inorder (Node x e d) = (inorder e) ++ [x] ++ (inorder d)
```

Travessia postorder: visitar árvore esquerda, depois árvore direita, e a seguir a raiz..

```
postorder :: BTree a -> [a]
postorder Empty = []
postorder (Node x e d) = (postorder e) ++ (postorder d) ++ [x]
```


Travessias de árvores binárias

```
arv = (Node 5 (Node 7 (Node 3 Empty Empty)
                      (Node 2 (Node 10 Empty Empty) Empty)
              )
              (Node 1 (Node 12 Empty Empty)
                      (Node 4 Empty (Node 8 Empty Empty))
              )
      )
```




```
preorder arv  = [5,7,3,2,10,1,12,4,8]
```

```
inorder arv   = [3,7,10,2,5,12,1,4,8]
```

```
postorder arv = [3,10,2,7,12,8,4,1,5]
```