Funções anónimas

As expressões lambda podem ser usadas na definição de funções. Por exemplo:

```
soma x y = x + y

soma1 = \x y -> x + y

soma2 = \x -> (\y -> x + y)

soma3 x = \y -> x + y
```

`soma`, `soma1`, `soma2` e `soma3` são funções equivalentes

Os operadores infixos aplicados apenas a um argumento (a que se dá o nome de secções), são uma forma abreviada de escrever funções anónimas.

Exemplos:

```
(+y)  ≡  \x -> x+y

(x+)  ≡  \y -> x+y

(*3)  ≡  \x -> x*3
```

Funções de ordem superior

- `(.)` composição de funções

$$\begin{tikzcd} A \arrow[r, "g"] \arrow[rr, bend right, "f\,.\,g"'] & B \arrow[r, "f"] & C \end{tikzcd}$$

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

Exemplo:

```
ultimo :: [a] -> a
ultimo = head . reverse
```

```
ultimo [1,2,3] = (head . reverse) [1,2,3]
               = head (reverse [1,2,3])
               = head [3,2,1]
               = 3
```

- `flip` troca a ordem dos argumentos de uma função binária.

```
flip :: (a -> b -> c) -> b -> a -> c
flip f x y = f y x
```

```
> (^) 3 2
9
> flip (^) 3 2
8
```

Exemplo:

```
mytake :: [a] -> Int -> [a]
mytake = flip take
```

```
mytake [1..10] 3 = flip take [1..10] 3
                 = take 3 [1..10]
                 = [1,2,3]
```

Funções de ordem superior

- `curry` transforma uma função que recebe como argumento um par, numa função equivalente que recebe um argumento de cada vez.

```
curry :: ((a,b) -> c) -> a -> b -> c
curry f x y = f (x,y)
```

- `uncurry` transforma uma função que recebe dois argumentos (um de cada vez), numa função equivalente que recebe um par.

```
uncurry :: (a -> b -> c) -> (a,b) -> c
uncurry f (x,y) = f x y
```

Exemplo:

```
quocientes :: [(Int,Int)] -> [Int]
quocientes l = map (\(x,y) -> div x y) l
```

Ou, em alternativa,

```
quocientes l = map (uncurry div) l
```

```
> quocientes [(10,3), (20,4)]
[3,5]
```

Funções de ordem superior

- `zipWith` constrói uma lista cujos elementos são calculados por uma função que é aplicada a argumentos que vêm de duas listas.

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f (x:xs) (y:ys) =  f x y : zipWith f xs ys
zipWith _ _ _ = []
```

```
> zipWith div [10,20..50] [1..]
[10,10,10,10,10]
```

```
> zipWith (^) [1..5] [2,2..]
[1,4,9,16,25]
```

```
> map (uncurry (^)) (zip [1..5] [2,2..])
[1,4,9,16,25]
```

Funções de ordem superior

- **takeWhile** recebe uma condição e uma lista e retorna o segmento inicial da lista cujos elementos satisfazem a condição dada.

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

```
> takeWhile (>3) [5,7,1,8,2]
[5,7]
```

- **dropWhile** recebe uma condição e uma lista e retorna a lista sem o segmento inicial de elementos que satisfazem a condição dada.

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

```
> dropWhile (>3) [5,7,1,8,2]
[1,8,2]
```

Funções de ordem superior

- **span** é uma função do `Prelude` que calcula simultamente o resultado das funções `takeWhile` e `dropWhile`. Ou seja, `span p l == (takeWhile p l, dropWhile p l)`

```
span :: (a -> Bool) -> [a] -> ([a],[a])
```

Exemplo: A função **lines** (do `Prelude`) que parte uma string numa lista de linhas.

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

```
lines :: String -> [String]
lines [] = []
lines s = let (l, r) = span (/='\n') s
          in case r of
               []   -> [l]
               x:xs -> l : lines xs
```

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:

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

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 revolvem um determinado valor.

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

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

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

```
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 o 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]
```

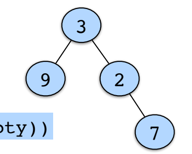
Uma lista,
- ou é vazia,
- ou tem um elemento e a uma sub-estrutura que é também uma lista.

```
[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 a 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))
```