Assim, mult pode ser usada para *gerar novas funções*.

**Exemplo:**

```
dobro = mult 2
triplo = mult 3
```

Qual é o seu tipo ?

Os operadores infixos também podem ser usados da mesma forma, isto é, aplicados a apenas um argumento, gerando assim uma nova função.

**Exemplo:**

```
(+) :: Integer -> Integer -> Integer
(<=) :: Integer -> Integer -> Bool
(*) :: Double -> Double -> Double
```

**(5+)** ≡ `(+) 5 :: Integer -> Integer`

**(0<=)** Qual é o tipo destas funções ?

**(3*)** Qual o valor das expressões: (0<=) 8
(3*) 5.7



# map

Considere as seguintes funções:

```
distancias :: [Ponto] -> [Float]
distancias [] = []
distancias (p:ps) = (distOrigem p) : (distancias ps)
```

```
minusculas :: String -> String
minusculas [] = []
minusculas (c:cs) = toLower c : minusculas cs
```

```
triplica :: [Double] -> [Double]
triplica [] = []
triplica (x:xs) = (3*x) : triplica xs
```

```
factoriais :: [Integer] -> [Integer]
factoriais [] = []
factoriais (n:ns) = fact n : factoriais ns
```

Todas estas funções têm um *padrão de computação* comum:

***aplicam uma função a cada elemento de uma lista, gerando deste modo uma nova lista.***



# map

Podemos definir uma função de ordem superior que aplica uma função ao longo de uma lista:

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

Note que `(map f lista)` é equivalente a `[ f x | x <- lista ]`

Podemos definir as funções do slide anterior à custa da função `map`, fazendo:

```
distancias lp = map distOrigem lp
minusculas s = map toLower s
triplica xs = map (3*) xs
factoriais ns = map fact ns
```

Ou então,

```
distancias = map distOrigem
minusculas = map toLower
triplica = map (3*)
factoriais = map fact
```

Porquê ?



# filter

Considere as seguites funções:

```
aprov :: [Int] -> [Int]
aprov [] = []
aprov (x:xs) = if (10<=x) then x:(aprov xs)
                          else (aprov xs)
```

```
filtraDigitos :: String -> String
filtraDigitos [] = []
filtraDigitos (c:cs)
       | isDigit c = c:(filtraDigitos cs)
       | otherwise = filtraDigitos cs
```

```
primQuad :: [Ponto] -> [Ponto]
primQuad [] = []
primQuad ((x,y):ps) | x>0 && y>0 = (x,y):(primQuad ps)
                    | otherwise  = primQuad ps
```

Todas estas funções têm um *padrão de computação* comum:

***dada uma lista, geram uma nova lista com os elementos da lista que satisfazem um determinado predicado.***

## filter

filter é uma função de ordem superior que filtra os elementos de uma lista que verificam um dado predicado (i.e. mantém os elementos da lista para os quais o predicado é verdadeiro).

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

Note que `(filter p lista)` é equivalente a `[ x | x <- lista , p x ]`

Podemos definir as funções do slide anterior à custa da função filter, fazendo:

```
aprov xs = filter (10<=) xs
```

```
filtraDigitos s = filter isDigit s
```

```
primQuad ps = filter aux ps
  where  aux (x,y) = 0<x && 0<y
```

Ou então,

```
aprov = filter (10<=)
```

```
filtraDigitos = filter isDigit
```

```
primQuad = filter aux
  where  aux (x,y) = 0<x && 0<y
```



## Funções anónimas

Em Haskell, é possível definir novas funções através de *abstrações lambda* ($\lambda$) da forma:

**`\x -> e`** representando uma função com argumento formal x e corpo da função e (a notação é inspirada no $\lambda$-*calculus* aonde isto se escreve $\lambda x.e$ )

**Exemplos:**

```
> (\x -> x+x) 5
10
```

```
> (\y -> y*3) 4
12
```

```
> (\x -> x:x^2:x^3:[]) 2
[2,4,8]
```

Funções com mais do que um argumento podem ser definidas de forma *abreviada* por:

**`\p1 ... pn -> e`** Além disso, os argumentos p1 ... pn podem ser padrões.

**Exemplos:**

```
> (\x y -> x+y) 5 3
8
```

```
> (\(x:xs) y -> y:xs) [3,4,5,2] 7
[7,4,5,2]
```

```
> (\(x1,y1) (x2,y2) -> (x1*x2,y1*y2)) (0,3) (5,2)
(0,6)
```

**Note que:** `\x y -> x+y` ≡ `\x -> (\y -> x+y)` Justifique com base no tipo.

Como ao definir estas funções não lhes associamos um nome, elas dizem-se **anónimas**.



## Funções anónimas

É possível utilizar funções anónimas na definição de outras funções.

**Exemplos:**

```
dobro = \x->x+x
```

```
cauda = \(_:xs) -> xs
```

```
> dobro 5
10
> cauda [9,3,4,5]
[3,4,5]
```

As funções anónimas são úteis para evitar a declaração de funções auxiliares.

**Exemplos:**

```
trocaPares xs = map troca xs
  where troca (x,y) = (y,x)
```

```
trocaPares xs = map (\(x,y)->(y,x)) xs
```

```
primQuad = filter (\(x,y) -> 0<x && 0<y)
```

Os operadores infixos aplicados apenas a um argumento são uma forma abreviada de escrever funções anónimas.

**Exemplos:**

`(+y)` ≡ `\x -> x+y`

`(x+)` ≡ `\y -> x+y`

`(*5)` ≡ `\x -> x*5`



## foldr

Considere as seguintes funções:

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

`sum [3,5,8]` ⟹ `3 + (5 + (8+0))`

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

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

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

Todas estas funções têm um *padrão de computação* comum:

***aplicar um operador binário ao primeiro elemento da lista e ao resultado de aplicar a função ao resto da lista.***

O que se está a fazer é a extensão de uma operação binária a uma lista de operandos.

# foldr

Podemos capturar este padrão de computação fornecendo à função foldr o operador binário e o resultado a devolver para a lista vazia.

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

Note que `(foldr f z [x1,...,xn])` é igual a `(f x1 (... (f xn z)...))`
ou seja, `(x1 `f` (x2 `f` (... (xn `f` z)...)))` *(associa à direita)*

Podemos definir as funções do slide anterior à custa da função foldr, fazendo:

```
sum xs = foldr (+) 0 xs
product xs = foldr (*) 1 xs
and bs = foldr (&&) True bs
concat ls = foldr (++) [] ls
```

**Exemplos:**

```
(product [4,3,5])  ⇒  4 * (3 * (5 * 1))  ⇒  60
(concat [[3,4,5],[2,1],[7,8]])  ⇒  [3,4,5] ++ ([2,1] ++ ([7,8]++[]))
                                ⇒  [3,4,5,2,1,7,8]
```



# foldl

Podemos usar um padrão de computação semelhante ao do foldr, mas *associando à esquerda*, através da função `foldl`.

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

Note que `(foldl f z [x1,...,xn])` é igual a `(f (...(f z x1) ...) xn)`
ou seja, `((...((z `f` x1) `f` x2)...) `f` xn)` *(associa à esquerda)*

**Exemplos:**

```
sum xs = foldl (+) 0 xs
concat ls = foldl (++) [] ls
reverse xs = foldl (\t h -> h:t) [] xs
```

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

concat [[2,3],[8,4,7],[1]]  ⇒  (([]++[2,3]) ++ [8,4,7]) ++ [1]
                            ⇒  [2,3,8,4,7,1]

reverse [3,4]  ⇒  ((\t h -> h:t) ((\t h -> h:t) [] 3) 4)
               ⇒  4 : ((\t h -> h:t) [] 3)  ⇒  4:3:[]  ⇒  [4,3]
```



# foldr *vs* foldl

Note que (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
```

As funções foldr e foldl estão formemente relacionadas com as estratégias para contruir funções recursivas sobre listas que vimos atrás.

foldr está relacionada com a *recursividade primitiva.*

foldl está relacionada com o *uso de acumuladores.*

**Exercício:** Considere as funções

```
sumR xs = foldr (+) 0 xs
sumL xs = foldl (+) 0 xs
```

Escreva a cadeia de redução das expressões (sumR [1,2,3]) e (sumL [1,2,3])
e compare com o funcionamento da função somatório definida sem e com e acumuladores.



# Outras funções de ordem superior

Composição de funções

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

Trocar a ordem dos argumentos

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

Obter a versão *curried* de uma função

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

Obter a versão *uncurried* de uma função

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

**Exemplos:**

```
sextuplo = dobro . triplo
reverse xs = foldl (flip (:)) [] xs
quocientes pares = map (uncurry div) pares
```

```
sextuplo 5  ⇒  dobro (triplo 5)  ⇒  dobro 15  ⇒  30

quocientes [(3,4),(23,5),(7,3)]  ⇒  [0,4,2]
```