Lazy Evaluation *(call-by-name)*

```
dobro (triplo (snd (9,8))) ⇒ (triplo (snd (9,8)))+(triplo (snd (9,8)))
                           ⇒ (3*(snd (9,8))) + (triplo (snd (9,8)))
                           ⇒ (3*(snd (9,8))) + (3*(snd (9,8)))
                           ⇒ (3*8) + (3*(snd (9,8)))
                           ⇒ 24 + (3*(snd (9,8)))
                           ⇒ 24 + (3*8)
                           ⇒ 24 + 24
                           ⇒ 48
```

Com a estrategia *lazy* os parametros das funções só são calculados se o seu valor fôr mesmo necessário.

```
nl (triplo (dobro (7*45)) ⇒ '\n'
```

A *lazy evaluation* faz do Haskell uma linguagem **não estrita**. Esto é, uma função aplicada a um valor indefinido pode ter em Haskell um valor bem definido.

```
nl (3/0) ⇒ '\n'
```

A *lazy evaluation* também vai permitir ao Haskell lidar com *estruturas de dados infinitas*.

Podemos definir uma função recorrendo a várias equações.

Exemplo:

```
h :: (Char,Int) -> Int
h ('a',x) = 3*x
h ('b',x) = x+x
h (_,x) = x
```

Todas as equações têm que ser bem tipadas e de tipos coincidentes.

Cada equação é usada como regra de redução. Quando uma função é aplicada a um argumento, a equação que é selecionada como regra de redução é a 1ª equação (a contar de cima) cujo padrão que tem como argumento **concorda** com o argumento actual (***pattern matching***).

Exemplos:

```
h ('a',5) ⇒ 3*5 ⇒ 15
h ('b',4) ⇒ 4+4 ⇒ 8
h ('B',9) ⇒ 9
```

Note: Podem existir *várias* equações com padrões que concordam com o argumento actual. Por isso, a ordem das equações é importante, pois define uma prioridade na escolha da regra de redução.

O que acontece se alterar a ordem das equações que definem h ?

Funções Totais & Funções Parciais

Uma função diz-se **total** se está definida para todo o valor do seu domínio.

Uma função diz-se **parcial** se há valores do seu domínio para os quais ela não está definida (isto é, não é capaz de produzir um resultado no conjunto de chegada).

Exemplos:

```
conjuga :: (Bool,Bool) -> Bool
conjuga (True,True) = True
conjuga (x,y) = False
```

Função total

```
parc :: (Bool,Bool) -> Bool
parc (True,False) = False
parc (True,x) = True
```

Função parcial

Porquê ?

Tipos Simónimos

O Haskell pode renomear tipos através de declarações da forma:

```
type Nome p1 ... pn = tipo
```

parâmetros (*variáveis de tipo*)

Exemplos:

```
type Ponto = (Float,Float)
type ListaAssoc a b = [(a,b)]
```

Note que não estamos a criar tipos novos, mas apenas nomes novos para tipos já existentes. Esses nomes devem contribuir para a compreensão do programa.

Exemplo:

```
distOrigem :: Ponto -> Float
distOrigem (x,y) = sqrt (x^2 + y^2)
```

O tipo `String` é outro exemplo de um tipo sinónimo, definido no Prelude.

```
type String = [Char]
```

Definições Locais

Uma definição associa um nome a uma expressão.

Todas as definições feitas até aqui podem ser vistas como **globais**, uma vez que elas são visíveis no *módulo* do programa aonde estão. Mas, muitas vezes é útil reduzir o âmbito de uma declaração.

Em Haskell há duas formas de fazer definições **locais**: utilizando expressões `let ... in` ou através de cláusulas `where` junto da definição equacional de funções.

Exemplos:

```
let  c = 10
     (a,b) = (3*c, f 2)
     f x = x + 7*c
in f a + f b
```

⇒ 242

Porquê ?

```
testa y = 3 + f y + f a + f b
  where  c = 10
         (a,b) = (3*c, f 2)
         f x = x + 7*c
```

```
> testa 5
320

> c
Variable not in scope: `c'

> f a
Variable not in scope: `f'
Variable not in scope: `a'
```

As declarações locais podem ser de funções e de identificadores (fazendo uso de padrões).

Layout

Ao contrário de quase todas as linguagens de programação, o Haskell não necessita de marcas para delimitar as diversas declarações que constituem um programa.

Em Haskell a *identação do texto* (isto é, a forma como o texto de uma definição está disposto), tem um significado bem preciso.

Regras fundamentais:

1. Se uma linha começa mais à frente do que começou a linha anterior, então ela deve ser considerada como a continuação da linha anterior.
2. Se uma linha começa na mesma coluna que a anterior, então elas são consideradas definições independentes.
3. Se uma linha começa mais atrás do que a anterior, então essa linha não pretence à mesma lista de definições.

Ou seja: *definições do mesmo género devem começar na mesma coluna*

Exemplo:

```
exemplo :: Float -> Float -> Float
exemplo x 0 = x
exemplo x y = let a = x*y
                  b = if (x>=y) then x/y
                                else y*x
                  c = a-b
              in (a+b)*c
```


Operadores

Operadores infixos como o `+`, `*`, `&&`, ... , não são mais do que funções.

Um operador infixo pode ser usado como uma função vulgar (i.e., usando notação prefixa) se estiver entre parentesis.

Exemplo: `(+) 2 3` é equivalente a `2+3`

Note que `(+) :: Int -> Int -> Int`

Podem-se definir novos operadores infixos.

```
(+>) :: Float -> Float -> Float
x +> y = x^2 + y
```

Funções binárias podem ser usadas como um operador infixo, colocando o seu nome entre ``` `` ```.

Exemplo: `mod :: Int -> Int -> Int`

`3 `mod` 2` é equivalente a `mod 3 2`

Cada operador tem uma prioridade e uma associatividade estipulada.

Isto faz com que seja possível evitar alguns parentesis.

Exemplo: `x + y + z` é equivalente a `(x + y) + z`

`x + 3 * y` é equivalente a `x + (3 * y)`

A aplicação de funções tem prioridade máxima e é associativa à esquerda.

Exemplo: `f x * y` é equivalente a `(f x) * y`

É possível indicar a prioridade e a associatividade de novos operadores através de declarações.

```
infixl  num  op
infixr  num  op
infix   num  op
```

Funções com Guardas

Em Haskell é possível definir funções com alternativas usando **guardas**.

Uma guarda é uma expressão booleana. Se o seu valor for `True` a equação correspondente será usada na redução (senão tenta-se a seguinte).

Exemplos:

```
sig x y = if x > y then 1
                   else if x < y then -1
                                 else 0
```

é equivalente a

```
sig x y | x > y  = 1
        | x < y  = -1
        | x == y = 0
```

ou a

```
sig x y
    | x > y     = 1
    | x < y     = -1
    | otherwise = 0
```

`otherwise` é equivalente a `True`.

Exemplo: Raizes reais do polinómio $a\,x^2 + b\,x + c$

```
raizes :: (Double,Double,Double) -> (Double,Double)
raizes (a,b,c) = (r1,r2)
  where r1 = (-b + r) / (2*a)
        r2 = (-b - r) / (2*a)
        d = b^2 - 4*a*c
        r | d >= 0 = sqrt d
          | d < 0 = error "raizes imaginarias"
```

`error` é uma função pré-definida que permite indicar a mensagem de erro devolvida pelo interpretador. Repare no seu tipo

```
error :: String -> a
```

```
> raizes (2,10,3)
(-0.320550528229663,-4.6794494717703365)

> raizes (2,3,4)
*** Exception:  raizes imaginarias
```


Listas

`[T]` é o tipo das listas cujos elementos são todos do tipo `T` -- *listas homogéneas* .

```
[3.5^2, 4*7.1, 9+0.5] :: [Float]
[(253,"Braga"), (22,"Porto"), (21,"Lisboa")] :: [(Int,String)]
[[1,2,3], [1,4], [7,8,9]] :: [[Integer]]
```

Na realidade, as listas são construidas à custa de dois construtores primitivos:

- a lista vazia `[]`
- o construtor `(:)`, que é um operador infixo que dado um elemento `x` de tipo `a` e uma lista `l` de tipo `[a]`, constroi uma nova lista com `x` na 1ª posição seguida de `l`.

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

`[1,2,3]` é uma abreviatura de `1:(2:(3:[]))` que é igual a `1:2:3:[]` porque `(:)` é associativa à direita.

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

Os padrões do tipo lista são expressões envolvendo apenas os construtores `:` e `[]` (*entre parentesis*), ou a representação abreviada de listas.

```
head (x:xs) = x
```

```
tail (x:xs) = xs
```

```
null [] = True
null (x:xs) = False
```

```
soma3 :: [Integer] -> Integer
soma3 [] = 0
soma3 (x:y:z:t) = x+y+z
soma3 (x:y:t) = x+y
soma3 (x:t) = x
```

Qual o tipo destas funções ?

As funções são totais ou parciais?

```
> head [3,4,5,6]
3
> tail "HASKELL"
"ASKELL"
> head []
*** exception
> null [3.4, 6.5, -5.5]
False
> soma3 [5,7]
13
```

Em soma3 a ordem das equações é importante ? Porquê ?

Recorrência

Como definir a função que calcula o comprimento de uma lista ?

Temos dois casos:

- Se a lista fôr vazia o seu comprimento é zero.
- Se a lista não fôr vazia o seu comprimento é um mais o comprimento da cauda da lista.

```
length [] = 0
length (x:xs) = 1 + length xs
```

Esta função é **recursiva** uma vez que se invoca a si própria (aplicada à cauda da lista).

A função termina uma vez que as invocações recursivas são feitas sobre listas cada vez mais curtas, e vai chegar ao ponto em que a lista é vazia.

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

Em linguagens funcionais, a recorrência é a forma de obter ciclos.

Mais alguns exemplos de funções já definidas no módulo Prelude:

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

Qual o tipo destas funções ?

São totais ou parciais ?

Podemos trocar a ordem das equações ?

```
last [x] = x
last (_:xs) = last xs
```

```
elem x [] = False
elem x (y:ys) | x == y    = True
              | otherwise = elem x ys
```

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


Considere a função `zip` já definida no Perlude:

```
zip [] [] = []
zip [] (y:ys) = []
zip (x:xs) [] = []
zip (x:xs) (y:ys) = (x,y) : (zip xs ys)
```

Qual o seu tipo ? É total ou parcial ?
Podemos trocar a ordem das equações ?
Podemos dispensar alguma equação ?
Será que podemos definir `zip` com menos equações ?

Exercícios:

- Indique todos os passos de redução envolvidos no cálculo da expressão:

  `zip [1,2] "LCC"`

- Defina a função que faz o "zip" de 3 listas.

- Defina a função `unzip :: [(a,b)] -> ([a],[b])`

Padrões sobre números naturais.

O Haskell aceita como um padrão sobre números naturais, expressões da forma:

(*variável* + *número_natural*)

Exemplos:

```
fact 0 = 1
fact (n+1) = (n+1) * (fact n)
```

```
decTres (x+3) = x
```

```
> fact 4
24
> fact (-2)
*** Exception: Non-exhaustive patterns in function fact
```

```
> decTres 5
2
> decTres 10
7
> decTres 2
*** Exception: Non-exhaustive ...
```

Atenção: expressões como

(n*5), (x-4) ou (2+n)

não são padrões !

Mais alguma funções sobre listas pré-definidas no Prelude.

```
(x:_) !! 0  = x
(_:xs) !! (n+1) = xs !! n
```

```
init [x] = []
init (x:xs) = x : init xs
```

O que fazem estas funções ?

Qual o seu tipo ?

Estas funções serão totais ?

Trocando a ordem das equações, será que obtemos a mesma função ?

As funções take e drop estão pré-definidas no Prelude da seguinte forma:

```
take                :: Int -> [a] -> [a]
take n _   | n <= 0  = []
take _ []            = []
take n (x:xs)        = x : take (n-1) xs
```

```
drop                :: Int -> [a] -> [a]
drop n xs | n <= 0   = xs
drop _ []            = []
drop n (_:xs)        = drop (n-1) xs
```

Estas funções serão totais ?

Trocando a ordem das equações, será que obtemos a mesma função ?

Defina funções equivalentes utilizando padrões de números naturais.

nome@padrão

nome@padrão é uma forma de fazer uma definição local ao nível de um argumento de uma função.

Exemplos:

A função

```
fun :: (Int,String) -> (Char,(Int,String))
```

pode ser definida, equivalentemente, por:

```
fun (n,(x:xs)) = (x,(n,(x:xs))
```

ou

```
fun par@(n,(x:xs)) = (x,par)
```

ou

```
fun (n,(x:xs)) = let par = (n,(x:xs))
                 in (x,par)
```

```
{- Esta função vai retirando os elementos de uma lista até
encontrar um elemento não positivo -}

dropWhilePos []                       = []
dropWhilePos lis@(x:xs) | x > 0       = dropWhilePos xs
                        | otherwise = lis
```


Algoritmos de Ordenação

A ordenação de um conjunto de valores é um problema muito frequente, e muito útil na organização de informação.

Para o problema de ordenação de uma lista de valores, existem diversos algoritmos:

- ***Insertion Sort***
- ***Quick Sort***
- ***Merge Sort***
- ***...***

Vamos apresentar estes algoritmos, para *ordenar uma lista de valores por ordem crescente*, de acordo com os operadores relacionais <, <=, >, e >= (que implicítamente assumimos estarem definidos para os tipos desses valores).