Expressões case

O Haskell tem ainda uma forma construir expressões que permite fazer análise de casos sobre a estrutura dos valores de um tipo. Essas expressões têm a forma:

```
case expressão of
     padrão -> expressão
     ...
     padrão -> expressão
```

Exemplos:

```
soma3 :: [Int] -> Int
soma3 l = case l of
            (x:y:z:t) -> x+y+z
            (x:y:t)   -> x+y
            (x:t)     -> x
            []        -> 0
```

```
null :: [a] -> Bool
null l = case l of
           []     -> True
           (x:xs) -> False
```

Funções recursivas sobre listas

- Como definir a função que calcula o comprimento de uma lista ?
  - Sabemos calcular o comprimento da lista vazia: é zero.
  - Se soubermos o comprimento da cauda da lista, também sabemos calcular o comprimento da lista completa: basta somar-lhe mais um.
- Como as listas são construídas unicamente à custa da lista vazia e de acrescentar um elemento à cabeça da lista, a definição da função `length` é muito simples:

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

Esta função é recursiva uma vez que se invoca a si própria.

- 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 função é aplicada à lista vazia.

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

Funções recursivas sobre listas

- **sum** calcula o somatório de uma lista de números.

```
sum :: Num a => [a] -> a
sum [] = 0
sum (x:xs) = x + sum xs
```

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

- **elem** testa se um elemento pertence a uma lista.

```
elem :: Eq a => a -> [a] -> Bool
elem x [] = False
elem x (y:ys) | x == y    = True
              | otherwise = elem x ys
```

```
elem 2 [1,2,3] = elem 2 [2,3]
               = True
```

Passo 1: a 1ª equação que faz *match* é a 2ª, mas como a guarda é falsa, usa a 3ª equação.

Passo 2: usa a 2ª equação porque faz *match* e a guarda é verdadeira.

Funções recursivas sobre listas

- **last** dá o último elemento de uma lista não vazia.

Note como a equação `last [x] = x` tem que aparecer em 1º lugar.

```
last :: [a] -> a
last [x] = x
last (_:xs) = last xs
```

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

O que aconteceria se trocássemos a ordem das equações?

Funções recursivas sobre listas

- **`init`** retira o último elemento de uma lista não vazia.

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

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

O que aconteceria se trocássemos a ordem das equações?

Funções recursivas sobre listas

- **`(++)`** faz a concatenação de duas listas.

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

Como a construção de listas é feita acrescentando elementos à esquerda da lista, vamos ter que definir a função fazendo a análise de casos sobre a lista da esquerda.

- Se a lista da esquerda for vazia `[] ++ l = l`
- Se a lista da esquerda não for vazia `(x:xs) ++ l = x : (xs ++ l)`

```
[1,2,3] ++ [4,5] = 1 : ([2,3] ++ [4,5])
                 = 1 : 2 : ([3] ++ [4,5])
                 = 1 : 2 : 3 : ([] ++ [4,5])
                 = 1 : 2 : 3 : [4,5]
                 = [1,2,3,4,5]
```

Haveria alguma diferença se trocássemos a ordem das equações?

Funções recursivas sobre listas

- **`reverse`** inverte uma lista.

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

Para acrescentar um elemento à direita da lista temos que usar `++[x]`

```
reverse [1,2,3] = (reverse [2,3]) ++ [1]
                = ((reverse [3]) ++ [2]) ++ [1]
                = (((reverse []) ++ [3]) ++ [2]) ++ [1]
                = [] ++ [3] ++ [2] ++ [1]
                = …
                = [3,2,1]
```

Funções recursivas sobre listas

- **`(!!)`** selecciona um elemento da lista numa dada posição.

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

```
> [6,4,3,1,5,7]!!2
3
> [6]!!2
*** Exception: Non-exhaustive patterns
> [6,4,3,1,5,7]!!(-3)
*** Exception: Non-exhaustive patterns
```

```
[6,4,3,1,5,7]!!2 = [4,3,1,5,7]!!1
                 = [3,1,5,7]!!0
                 = 3
```

Porquê ?

Funções recursivas sobre listas

Exemplo: a função que soma uma lista de pares, componente a componente

```
somas :: [(Int,Int)] -> (Int,Int)
somas l = (sumFst l, sumSnd l)
```

O padrão `((x,y):t)` permite extrair as componentes do par que está na cabeça da lista.

```
sumFst :: [(Int,Int)] -> Int
sumFst [] = 0
sumFst ((x,y):t) = x + sumFst t
```

```
sumSnd :: [(Int,Int)] -> Int
sumSnd [] = 0
sumSnd ((x,y):t) = y + sumSnd t
```

- Esta função recorre às funções `sumFst` e `sumSnd`, como funções auxiliares, para fazer o cálculo dos resultados parciais.
- Há no entanto desperdício de trabalho nesta implementação, porque se está a percorrer a lista duas vezes sem necessidade.
- Numa só travessia podemos ir somando os valores das respectivas componentes.

Tupling *(calcular vários resultados numa só travessia da lista)*

- Numa só travessia podemos ir somando os valores das respectivas componentes, mantendo um par que vamos construindo.

```
somas :: [(Int,Int)] -> (Int,Int)
somas [] = (0,0)
somas ((x,y):t) = (x + fst (somas t), y + snd (somas t))
```

Note que `(soma t)` devolve um par. Daí o uso das funções `fst` e `snd`.

Pode parecer que `(somas t)` está a ser calculada duas vezes, mas isso não é verdade. `(somas t)` só é calculado uma vez, já que o valor dos identificadores é imutável.

- Podemos fazer uma declaração local para tornar o código mais fácil de ler

```
somas :: [(Int,Int)] -> (Int,Int)
somas [] = (0,0)
somas ((x,y):t) = (x + a, y + b)
   where (a,b) = somas t
```

Estamos aqui a usar o padrão `(a,b)` para extrair as componentes do par devolvido por `(somas t)`.

Tupling *(calcular vários resultados numa só travessia da lista)*

```
somas :: [(Int,Int)] -> (Int,Int)
somas [] = (0,0)
somas ((x,y):t) = (x + fst (somas t), y + snd (somas t))
```

```
somas [(7,8),(1,2)] = (7+ fst (somas [(1,2)]), 8+ snd (somas [(1,2)]))
                    = (7+ fst (1+ fst (somas []), 2+ fst (somas [])) ,
                       8+ snd (1+ fst (somas []), 2+ snd (somas [])) )
                    = (7+ fst (1+ fst (0,0), 2+ fst (0,0)) ,
                       8+ snd (1+ fst (0,0), 2+ snd (0,0)) )
                    = (7+ fst (1+0, 2+0) , 8+ snd (1+0, 2+0) )
                    = (7+1+0 , 8+2+0)
                    = (8, 10)
```

Tupling *(calcular vários resultados numa só travessia da lista)*

```
somas :: [(Int,Int)] -> (Int,Int)
somas [] = (0,0)
somas ((x,y):t) = (x+a, y+b)
   where (a,b) = somas t
```

```
somas [(7,8),(1,2)] = (7+ …, 8+ …) = (7+1+0, 8+2+0) = (8,10)
    somas [(1,2)] = (1+ …, 2+ …)   = (1+0, 2+0)
        somas [] = (0,0)
```