

Funções simples sobre listas

- `tail` retira o primeiro elemento de uma lista não vazia, isto é, dá a cauda da lista.

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

Pattern matching

```
> tail [1,2,3]
[2,3]
> tail [10,20,30,40,50]
[20,30,40,50]
> tail []
*** Exception: tail: empty list
```

x = 1 , xs = [2,3]

x = 10 , xs = [20,30,40,50]

Não há pattern matching

69

Funções simples sobre listas

- `null` testa se uma lista é vazia.

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

Pattern matching

```
> null [1,2,3]
False
> null []
True
```

Falha o pattern matching na 1^a equação.
Usa a 2^a equação com sucesso x=1, xs=[2,3]

Usa a primeira equação com sucesso

70

Funções simples sobre listas

Exemplo: a função que soma os 3 primeiros elementos de uma lista de inteiros pode ser definida assim

```
soma3 :: [Int] -> Int
soma3 l | length l <= 3 = sum l
| otherwise = sum (take 3 l)
```

Esta é uma definição pouco eficiente, pois temos que calcular o comprimento da lista, para depois somar apenas os seus 3 primeiros elementos.

Como poderemos definir essa função sem utilizar funções auxiliares e tirando partido do mecanismo de *pattern matching*?

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

Note que a ordem relativa das 3 primeiras equações tem que ser esta.

O que acontece se passarmos a 3^a equação para 1º lugar?

71

Funções simples sobre listas

Outra alternativa para a função soma3 pode ser assim

```
soma3 :: [Int] -> Int
soma3 [] = 0
soma3 [x] = x
soma3 [x,y] = x+y
soma3 [x,y,z] = sum (take 3 z)
```

[x] é uma lista com exactamente 1 elemento. [x]=(x:[])
[x,y] é uma lista com exactamente 2 elementos. [x,y]=(x:y:[])
[x,y,z] é uma lista qualquer mas a equação só irá ser usada com listas com mais de dois elementos, dada a sua posição relativa.

Não confundir os padrões aqui usados com os usados na versão anterior

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

(x:y:z:t) é uma lista com pelo menos 3 elementos.
(x:y:t) é uma lista com pelo menos 2 elementos.
(x:t) é uma lista com pelo menos 1 elemento.

72

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
```

73

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

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

75

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
```

74

Funções recursivas sobre listas

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

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

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

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

O que aconteceria se trocássemos a ordem das equações?

76

Funções recursivas sobre listas

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

```
init :: [a] -> [a]
init [] = []
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
- Se a lista da esquerda não for vazia

```
[] ++ l = l
(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

- `(!!)` seleciona 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]!!1
= [4,3,1,5,7]!!0
= [3,1,5,7]!!-1
= 3
```

Porquê?