Maybe a

```
data Maybe a = Nothing | Just a
```

Exemplo: uma função para somar valores do tipo `Maybe Int` pode ser definida assim

```
myAdd :: Maybe Int -> Maybe Int -> Maybe Int
myAdd Nothing Nothing = Nothing
myAdd Nothing (Just y) = Nothing
myAdd (Just x) Nothing = Nothing
myAdd (Just x) (Just y) = Just (x+y)
```

Esta função pode ser definida de forma mais compacta. Como?

```
myAdd :: Maybe Int -> Maybe Int -> Maybe Int
myAdd (Just x) (Just y) = Just (x+y)
myAdd _ _ = Nothing
```

_ representa uma variável anónima. O GHCi gera automaticamente um nome novo para a variável. Costuma usar-se quando a variável não é utilizada no lado direito da equação.

Nota: poderíamos definir, por exemplo, esta equação assim:
`myAdd x y = Nothing`

Funções com guardas

Recorde a definição da função factorial

```
fact :: Integer -> Integer
fact 0 = 1
fact n = n * fact (n-1)
```

```
> fact 5
120
> fact 20
2432902008176640000
> fact (-1)
*** Exception: stack overflow
```

Porque a computação não termina e enche a *stack*.

Podemos definir `fact` usando uma guarda (condição) na segunda equação

```
fact :: Integer -> Integer
fact 0 = 1
fact n | n > 0 = n * fact (n-1)
```

```
> fact (-1)
*** Exception: Non-exhaustive patterns in
function fact
```

A guarda é uma expressão Booleana. A equação só pode ser usada se a condição for verdadeira.

Funções com guardas

Uma definição alternativa para `fact` poderá ser

```
fact :: Integer -> Integer
fact 0 = 1
fact n | n > 0 = n * fact (n-1)
       | otherwise = error "Não está definida."
```

Aqui temos 2 equações com guardas.

A guarda `otherwise` corresponde a `True`.

A função `error :: String -> a` do `Prelude` permite alterar a mensagem de erro devolvida.

```
> fact (-1)
*** Exception: Não está definida.
```


Funções com guardas

- As expressões condicionais podem ser aninhadas.

```
sinal :: Int -> Int
sinal x = if x<0 then -1
          else if x==0 then 0
               else 1
```

- As equações guardadas podem ser usadas para tornar definições que envolvam if's aninhados mais fáceis de ler.

```
sinal x | x<0  = -1
        | x==0 = 0
        | otherwise = 1
```

- O uso de equações guardadas é também uma forma de contornar o facto de as expressões condicionais em Haskell terem obrigatoriamente o ramo `else`.

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 parêntesis.

```
> 3 + 2
5
> (+) 3 2
5
```

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

```
> div 10 3
3
> 10 `div` 3
3
```

- Podemos definir novos operadores infixos

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

e indicar a prioridade e a associatividade através de declarações

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


Listas

- As listas são sequências de tamanho variável de elementos do mesmo tipo.
- As listas podem ser representadas colocando os seus elementos, separados por vírgulas, entre parêntesis rectos. Mas isso é açúcar sintáctico.

```
[1,2,3] :: [Int]
```

- Na realidade as listas são um tipo algébrico, cujos elementos são construídos à custa dos seguintes construtores:

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

[ ] representa a lista vazia.

(:) é o constructor infixo que recebe um elemento e uma lista, e acrescenta o elemento à cabeça da lista (isto é, do lado esquerdo da lista) .
Nota: **(:)** é associativo à direita.

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

```
> 1:2:3:[]
[1,2,3]
> (2,3):(0,-1):[]
[(2,3),(0,-1)]
> 'B':"om dia!"
"Bom dia!"
```


Funções simples sobre listas

- **head** dá o primeiro elemento de uma lista não vazia, isto é, a cabeça da lista.

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

(x:xs) é um padrão que representa uma lista com pelo menos um elemento. **x** é o primeiro elemento da lista e **xs** é a restante lista.

Um padrão que é argumento de uma função tem que estar **entre parêntesis**, excepto se for uma variável ou uma constante atómica.

Pattern matching

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

`x = 1 , xs = [2,3]`

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

Não há *pattern matching*

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*

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 1ª equação.
Usa a 2ª equação com sucesso `x=1, xs=[2,3]`

Usa a primeira equação com sucesso

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 3ª equação para 1º lugar?

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 l = sum (take 3 l)
```

`[x]` é uma lista com exactamente 1 elemento. `[x]==(x:[])`
`[x,y]` é uma lista com exactamente 2 elementos. `[x,y]==(x:y:[])`
`l` 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.

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.

```
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?

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?