Crivo de Eratóstenes

Um algoritmo mais eficiente para encontrar números primos, é o Crivo de Eratóstenes (assim chamado em honra ao matemático grego que o inventou), que permite obter todos os números primos até um determinados valor n. A ideia é a seguinte:

- Começa-se com a lista [2..n].
- Guarda-se o primeiro elemento da lista (pois é um número primo) e removem-se da cauda da lista todos os múltiplos desse primeiro elemento.
- Continua-se a aplicar o passo anterior à restante lista, até que a lista de esgote.

```
crivo []     = []
crivo (x:xs) = x : crivo [ n | n <- xs , n `mod` x /= 0 ]
```

```
primosAte n = crivo [2..n]
```

Lista infinita de números primos.

```
primos = crivo [2..]
```


Factorização em primos

O Teorema Fundamental da Aritmética (enunciado pela primeira vez por Euclides) diz que qualquer número inteiro (maior do que 1) pode ser decomposto num produto de números primos. Esta decomposição é única a menos de uma permutação.

Exemplo: Com o auxílio da lista de números primos, podemos definir uma função que dado um número (maior do que 1), calcula a lista dos seus factores primos.

```
factoriza :: Integer -> [Integer]
factoriza n = aux n primos
  where aux 1 _ = []
        aux n (x:xs)
              | n `mod` x /= 0 = aux n xs
              | otherwise      = x : aux (n `div` x) (x:xs)
```

```
> factoriza 94753
[19,4987]
> factoriza 9475312
[2,2,2,2,7,11,7691]
```


Funções de ordem superior

Em Haskell, as funções são entidades de primeira ordem. Ou seja,

- As funções podem receber outras funções como argumento.

```
twice :: (a -> a) -> a -> a
twice f x = f (f x)
```

Exemplos:

```
dobro :: Int -> Int
dobro x = x + x
```

```
quadruplo :: Int -> Int
quadruplo x = twice dobro x
```

```
quadruplo 5 = twice dobro 5
            = dobro (dobro 5)
            = (dobro 5) + (dobro 5)
            = (5+5) + (5+5)
            = 10 + 10
            = 20
```

```
retira2 :: [a] -> [a]
retira2 l = twice tail l
```

```
retira2 [4,5,7,0,9] = twice tail [4,5,7,0,9]
                    = tail (tail [4,5,7,0,9])
                    = tail [5,7,0,9]
                    = [7,0,9]
```


Funções de ordem superior

- As funções podem devolver outras funções como resultado.

```
mult :: Int -> Int -> Int
mult x y = x * y
```

O tipo é igual a `Int -> (Int -> Int)`, porque `->` é associativo à direita

Exemplos:

```
triplo :: Int -> Int
triplo = mult 3
```

`triplo` tem o mesmo tipo que `mult 3`

```
triplo 5 = mult 3 5
         = 3 * 5
         = 15
```

`mult 3 5 = (mult 3) 5`, porque a aplicação é associativa à esquerda

```
twice (mult 2) 5 = (mult 2) ((mult 2) 5) = mult 2 (mult 2 5)
                 = 2 * (mult 2 5)
                 = 2 * (2 * 5)
                 = 20
```

map

Consideremos as seguintes funções:

```
triplos :: [Int] -> [Int]
triplos [] = []
triplos (x:xs) = 3*x : triplos xs
```

Estas funções fazem coisas distintas entre si, mas **a forma como operam é semelhante**: aplicam uma transformação a cada elemento da lista de entrada.

```
maiusculas :: String -> String
maiusculas [] = []
maiusculas (x:xs) = toUpper x : maiusculas xs
```

Dizemos que estas funções têm um **padrão de computação** comum, e apenas diferem na função que é aplicada a cada elemento da lista.

```
somapares :: [(Float,Float)] -> [Float]
somapares [] = []
somapares ((a,b):xs) = a+b : somapares xs
```

A função **`map`** do `Prelude` sintetiza este padrão de computação, abstraindo em relação à função que é aplicada aos elementos da lista.

map

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

`map` é uma função de ordem superior que recebe a função `f` que é aplicada ao longo da lista.

Exemplos:

```
triplos :: [Int] -> [Int]
triplos l = map (3*) l
```

```
maiusculas :: String -> String
maiusculas xs = map toUpper xs
```

```
somapares :: [(Float,Float)] -> [Float]
somapares l = map aux l
     where aux (a,b) = a+b
```

```
triplos [1,2] = map (3*) [1,2]
              = 3*1 : map (3*) [2]
              = 3*1 : 3*2 : map (3*) []
              = 3*1 : 3*2 : []
              = 3:6:[] = [3,6]
```

Usando listas por compreensão, poderíamos definir a função `map` assim:

```
map f l = [ f x | x <- l ]
```


filter

Consideremos as seguintes funções:

```
pares :: [Int] -> [Int]
pares [] = []
pares (x:xs) = if even x
               then x : pares xs
               else pares xs
```

Estas funções fazem coisas distintas entre si, mas **a forma como operam é semelhante**: selecionam da lista de entrada os elementos que verificam uma dada condição.

```
positivos :: [Double] -> [Double]
positivos [] = []
positivos (x:xs)
        | x > 0     = x : positivos xs
        | otherwise = positivos xs
```

Estas funções têm um **padrão de computação** comum, e apenas diferem na condição com que cada elemento da lista é testado.

A função **`filter`** do `Prelude` sintetiza este padrão de computação, abstraindo em relação à condição com que os elementos da lista são testados.

filter

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

`filter` é uma função de ordem superior que recebe a condição `p` (um predicado) com que cada elemento da lista é testado.

Exemplos:

```
pares :: [Int] -> [Int]
pares l = filter even l
```

```
positivos :: [Double] -> [Double]
positivos xs = filter (>0) xs
```

```
pares [1,2,3,4] = filter even [1,2,3,4]
                = filter even [2,3,4]
                = 2 : filter even [3,4]
                = 2 : filter even [4]
                = 2 : 4 : filter even []
                = 2:4:[] = [2,4]
```

Usando listas por compreensão, poderíamos definir a função `filter` assim:

```
filter p l = [  x | x <- l, p x ]
```

Funções anónimas

Em Haskell é possível definir funções sem lhes dar nome, através expressões lambda.

Por exemplo, `\x -> x+x`

É uma **função anónima** que recebe um número x e devolve como resultado x+x.

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

Uma expressão lambda tem a seguinte forma (a notação é inspirada no *λ-calculus*):

\padrão ... padrão -> expressão

Exemplos:

```
> (\x y -> x+y) 3 8
11
```

```
> (\(x1,y1) (x2,y2) -> (x1+x2,y1+y2)) (3,2) (7,9)
(10,11)
```

```
> (\(x:xs) -> xs) [1,2,3]
[2,3]
```

```
> (\(x:xs) y -> y:xs) [1,2,3] 9
[9,2,3]
```


Funções anónimas

As expressões lambda são úteis para evitar declarações de pequenas funções auxiliares.

Exemplo: Em vez de

```
trocapares :: [(a,b)] -> [(b,a)]
trocapares l = map troca l
   where troca (x,y) = (y,x)
```

pode-se escrever

```
trocapares l = map (\(x,y)->(y,x)) l
```

Exemplo:

```
multiplosDe :: Int -> [Int] -> [Int]
multiplosDe n xs = filter (\x -> mod x n == 0) xs
```


Funções anónimas

As expressões lambda podem ser usadas na definição de funções. Por exemplo:

```
soma x y = x + y
soma1 = \x y -> x + y
soma2 = \x -> (\y -> x + y)
soma3 x = \y -> x + y
```

`soma`, `soma1`, `soma2` e `soma3` são funções equivalentes

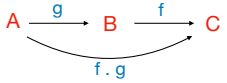
Os operadores infixos aplicados apenas a um argumento (a que se dá o nome de secções), são uma forma abreviada de escrever funções anónimas.

Exemplos:

```
(+y)  ≡  \x -> x+y
(x+)  ≡  \y -> x+y
(*3)  ≡  \x -> x*3
```


Funções de ordem superior

- `(.)` composição de funções




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

Exemplo:

```
ultimo :: [a] -> a
ultimo = head . reverse
```

```
ultimo [1,2,3] = (head . reverse) [1,2,3]
               = head (reverse [1,2,3])
               = head [3,2,1]
               = 3
```

- `flip` troca a ordem dos argumentos de uma função binária.

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

```
> (^) 3 2
9
> flip (^) 3 2
8
```

Exemplo:

```
mytake :: [a] -> Int -> [a]
mytake = flip take
```

```
mytake [1..10] 3 = flip take [1..10] 3
                 = take 3 [1..10]
                 = [1,2,3]
```

Funções de ordem superior

- **curry** transforma uma função que recebe como argumento um par, numa função equivalente que recebe um argumento de cada vez.

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

- **uncurry** transforma uma função que recebe dois argumentos (um de cada vez), numa função equivalente que recebe um par.

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

Exemplo:

```
quocientes :: [(Int,Int)] -> [Int]
quocientes l = map (\(x,y) -> div x y) l
```

Ou, em alternativa,

```
quocientes l = map (uncurry div) l
```

```
> quocientes [(10,3), (20,4)]
[3,5]
```


Funções de ordem superior

- **zipWith** constrói uma lista cujos elementos são calculados por uma função que é aplicada a argumentos que vêm de duas listas.

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f (x:xs) (y:ys) =  f x y : zipWith f xs ys
zipWith _ _ _ = []
```

```
> zipWith div [10,20..50] [1..]
[10,10,10,10,10]
```

```
> zipWith (^) [1..5] [2,2..]
[1,4,9,16,25]
```

```
> map (uncurry (^)) (zip [1..5] [2,2..])
[1,4,9,16,25]
```


Funções de ordem superior

- **takeWhile** recebe uma condição e uma lista e retorna o segmento inicial da lista cujos elementos satisfazem a condição dada.

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

```
> takeWhile (>3) [5,7,1,8,2]
[5,7]
```

- **dropWhile** recebe uma condição e uma lista e retorna a lista sem o segmento inicial de elementos que satisfazem a condição dada.

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

```
> dropWhile (>3) [5,7,1,8,2]
[1,8,2]
```


Funções de ordem superior

- **span** é uma função do `Prelude` que calcula simultaneamente o resultado das funções `takeWhile` e `dropWhile`. Ou seja, `span p l == (takeWhile p l, dropWhile p l)`

```
span :: (a -> Bool) -> [a] -> ([a],[a])
```

Exemplo: A função **lines** (do `Prelude`) que parte uma string numa lista de linhas.

```
> lines "  \nabds\tbfsas\n26egd\n\n3673qw"
["  ","abds\tbfsas","26egd","","3673qw"]
```

```
lines :: String -> [String]
lines [] = []
lines s = let (l, r) = span (/='\n') s
          in case r of
               []  -> [l]
               x:xs -> l : lines xs
```