Funções de ordem superior

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)
```

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

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

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

- As funções podem devolver outras funções como resultado

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

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

- **MAP**, consideraremos as seguintes funções:

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

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

--> Estas duas funções têm um padrão de computação comum, e apenas diferem na função que é aplicada a cada elemento da lista.

Ou seja, a forma como operam é semelhante: aplicam uma transformação a cada elemento da lista de entrada.

Assim, map é uma função de ordem superior que recebe a função f que é aplicada ao longo da lista:

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

Exemplos:

```
triplos :: [Int] -> [Int]

triplos l = map (3*) l
```

```
somapares :: [(Float,Float)] -> [Float]
somapares l = map aux l
        where aux (a,b) = a+b
                | {-Definir a função a ser aplicada a cada elemento.-}
```

- **FILTER**, consideraremos as seguintes funções

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

```
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.

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

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

```
    | p x       = x : filter p xs
    | otherwise = filter p xs
```

Exemplos:

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

```
positivos :: [Int] -> [Int]
positivos l = filter (>0) l
```

--> É possível definir a função filter através de listas por compreensão:

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

Funções Anónimas

Em haskell, é possível definir funções sem lhes dar nome através de expressões lambda. Por exemplo, `\x -> x+x`, é uma função anónima que recebe um número `x` e devolve como resultado `x+x`.

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

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

Em vez de:

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

Pode-se escrever:

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

- **FOLDR**

foldr é uma função de ordem superior que recebe o operador f que é usado para construir o resultado, e o valor z a devolver
quando a lista é vazia.

O seu padrão de computação é semelhante ao das seguintes funções:

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

product [] = 1
product (x:xs) = x * (product xs)
```

Apenas diferem no operário que é usado e no valor a devolver quando a lista é vazia.

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

Assim, `sum [1,2,3] = foldr (+) 0 [1,2,3]`.
Exemplo de ultilização:

```
reverse l = foldr (\x r -> r++[x]) [] l
length = foldr (\h r -> 1+r) 0
```

- **FOLDL**

Vai construindo o resultado pelo lado esquerdo da lista.
Deste modo, a função foldl sintetiza um padrão de computação que corresponde a trabalhar com um acumulador.
O foldl recebe como argumentos a função que combina o acumulador com a cabeça da lista, e o valor inicial do acumulador:

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

--> z é o acumulador.
f é usado para combinar o acumulador com a cabeça da lista.
(f z x) é o novo valor do acumulador.

Exemplos de ultilização:

```
reverse l = foldl (\ac x -> x:ac) [] l  -- > [] é o valor inicial do acumulador
sum l     = foldl (+) 0 l
```

#haskell
#SoftwareEngineering