Currying

- Funções curried
- Porque currying é útil?
- Convenções sobre currying
- Utilidade de expressões lambda
- Seções de operadores
- Utilidade de seções de operadores

Funções curried

- Em Haskell, podemos escolher como modelar funções de dois ou mais argumentos.
- Normalmente, as representamos no que é chamado de forma curried, onde elas apresentam seus argumentos um de cada vez.
- Isso é chamado de currying em homenagem a Haskell Curry, que foi um dos pioneiros do cálculo A e que deu nome a linguagem Haskell.

Funções curried

- Funções que recebem os seus argumentos um por vez são chamadas de funções curried,
- Por exemplo, uma função curried para multiplicar dois inteiros normalmente seria definida assim:
- multiplica :: Int -> Int -> Int
- multiplica x y = x*y

Funções curried

- enquanto uma versão uncurried pode ser dada agrupando os argumentos em um par, assim:
- multiplicaUC :: (Int, Int) -> Int
- multiplicaUC (x, y) = x * y

Funções curried

- Nessa opção, para passar vários argumentos em uma aplicação de função é formar uma estrutura de dados com os dados desejados
- e passar a estrutura como argumento.
- Neste caso, fica claro que haverá um único argumento, que é a estrutura de dados.

Funções curried

- Outro exemplo: usando uma tupla:
- somaPar :: (Int, Int) -> Int
- somaPar (x, y) = x + y
- A função somaPar recebe um único argumento que é um par, e resulta na soma dos componentes do par.
- Evidentemente este mecanismo não permite a aplicação parcial da função.

Funções curried

- Funções curried às vezes são chamadas de funções currificadas em português.

- Funções com mais de um argumento curried, resultando em funções aninhadas.

Porque currying é útil ?

- A notação é um pouco mais limpa:
- aplicamos uma função a um único argumento justapondo os dois, f x,
- e a aplicação a dois argumentos é feita estendendo isto assim: g x y.

Porque currying é útil ?

- Permite aplicação parcial.
- No caso da função multiplica, podemos escrever expressões como multiplica 2, que retorna uma função (2*y),
- embora isso não seja possível se os dois argumentos forem agrupados em um par, como é o caso de multiplicaUC.

Porque currying é útil ?

- Funções curried são mais flexíveis do que as funções com tuplas, porque muitas vezes funções úteis podem ser obtidas pela aplicação parcial de uma função curried.
- Por exemplo:

```
take 5 :: [a] -> [a]              -- função que seleciona os 5
                                  -- primeiros elementos de uma lista

drop 5 :: [a] -> [a]              -- função que descarta os 5
                                  -- primeiros elementos de uma lista

div 100 :: Integral a => a -> a -- função que divide 100 pelo seu argumento

elem 'a' :: String -> String      -- função que verifica se 'a' é
                                  -- elemento de uma lista
```

Convenções sobre currying

- Para evitar excesso de parênteses ao usar funções curried, duas regras simples foram adotadas na linguagem Haskell:
- A seta -> (construtor de tipos função) associa-se à direita.
- Exemplo:
- Int -> Int -> Int -> Int
- significa
- Int -> (Int -> (Int -> Int))

Convenções sobre currying

- A aplicação de função tem associatividade à esquerda.
- Exemplo:
- mult x y z
- significa
- ((mult x) y) z
- A menos que seja explicitamente necessário o uso de tuplas, todas as funções em Haskell são normalmente definidas na forma curried.

Utilidade de expressões lambda

- Expressões lambda podem ser usadas para dar um sentido formal para as funções definidas usando currying
  e para a aplicação parcial de funções.
- A função
- soma x y = x + y
- pode ser entendida como
- soma = \x -> (\y -> x + y)
- isto é, soma é uma função que recebe um argumento x e resulta em uma função que por sua vez recebe um argumento y e resulta em x+y.

Utilidade de expressões lambda

- Exemplos de uso:
- soma

==> \x -> (\y -> x + y)

- soma 2

==> (\x -> (\y -> x + y)) 2

==> \y -> 2 + y

- Expressões lambda também são úteis na definição de funções que retornam funções como resultados.

Utilidade de expressões lambda

- Exemplos de uso:
- soma 2 3

==> (\x -> (\y -> x + y)) 2 3

==> (\y -> 2 + y) 3

==> 2+3

==> 5

Seções de operadores

- Operadores
- Seções de operadores

Operadores

- Um operador binário infixo é uma função de dois argumentos escrita em notação infixa,
- isto é, entre os seus (dois) argumentos, ao invés de precedê-los.
- Por exemplo, a função (+) do prelúdio, para somar dois números, é um operador infixo, portanto deve ser escrita entre os operandos:
- 3 + 4

Operadores

- Lexicalmente, operadores consistem inteiramente de símbolos, em oposição aos identificadores normais que são alfanuméricos.
- Haskell não tem operadores prefixos, com exceção do menos (-), que pode ser tanto infixo (subtração) como prefixo (negação).
- Por exemplo:
- 3 - 4 ==> -1 {- operador infixo: subtração -}
- - 5 ==> -5 {- operador prefixo: negação -}

Operadores

- Um identificador alfanumérico pode ser usado como operador infixo quando escrito entre sinais de crase (`).
- Por exemplo, a função div do prelúdio calcula o quociente de uma divisão inteira:
- div 20 3 ==> 6
- Usando a notação de operador infixo:
- 20 `div` 3 ==> 6

Operadores

- Um operador infixo (escrito entre seus dois argumentos) pode ser convertido em uma função curried normal (escrita antes de seus dois argumentos) usando parênteses.
- Exemplos:
- (+) é a função que soma dois números.
- 1 + 2 ==> 3
- (+) 1 2 ==> 3

Operadores

- Mais exemplos:
- (>) é a função que verifica se o primeiro argumento é maior que o segundo.
- 100 > 200 ==> False
- (>) 100 200 ==> False
- (++) é a função que concatena duas listas.
- [1,2] ++ [30,40,50] ==> [1,2,30,40,50]
- (++) [1,2] [30,40,50] ==> [1,2,30,40,50]

Seções de operadores

- Como os operadores infixos são de fato funções, eles podem ser aplicados parcialmente.
- Haskell oferece uma notação especial para a aplicação parcial de um operador infixo, chamada de **seção do operador.**
- Uma seção de um operador é escrita colocando o operador e o argumento desejado entre parênteses.
- (1+)

Seções de operadores

- Exemplo:
- (1+)
- é a função que incrementa (soma um) ao seu argumento.
- É o mesmo que
- \x -> 1 + x
- (\x-> 1 + x) 8 ==> 9

- (1+) 8 ==> 9

Seções de operadores

- Exemplo:
- (*2)
- é a função que dobra (multiplica por 2) o seu argumento.
- É o mesmo que
- \x -> x * 2

- (*2) 8 ==> 16

Seções de operadores

- Exemplo:
- (100>)
- é a função que verifica se 100 é maior que o seu argumento.
- É o mesmo que
- \x -> 100 > x

- (100>) 8 ==> True

Seções de operadores

- Em geral, se ⊕ é um operador binário infixo, então as formas
- (⊕)
- (x ⊕)
- (⊕ y)
- são chamados de seções.

Seções de operadores

- Seções são equivalentes às definições com expressões lambdas:
- (⊕) = \x y -> x ⊕ y
- (x ⊕) = \y -> x ⊕ y
- (⊕ y) = \x -> x ⊕ y

Seções de operadores

- Nota:
- Como uma exceção, o operador binário - para subtração não pode formar uma seção direita
- (-x)
- porque isso é interpretado como negação unária na sintaxe Haskell.

Seções de operadores

- Nota:
- A função subtract do prelúdio é fornecida para este fim.
- Em vez de escrever (-x), você deve escrever
- (subtract x)

- (subtract 8) 10 ==> 2

Utilidade de seções de operadores

- Funções úteis às vezes podem ser construídas de uma forma simples, utilizando seções.
- Exemplos:

seção	descrição
`(1+)`	função sucessor
`(1/)`	função recíproco
`(*2)`	função dobro
`(/2)`	função metade

Utilidade de seções de operadores

- Seções são necessárias para anotar o tipo de um operador.
- Exemplos:

- (&&) :: Bool -> Bool -> Bool
- (+) :: Num a => a -> a -> a
- (:) :: a -> [a] -> [a]

Utilidade de seções de operadores

- Seções são necessárias para passar operadores como argumentos para outras funções.
- Exemplo:
- A função and do prelúdio, que verifica se todos os elementos de uma lista são verdadeiros, pode ser definida como:
- and :: [Bool] -> Bool
- and = foldr (&&) True
- onde foldr é uma função do prelúdio que reduz uma lista de valores a um único valor aplicando uma operação binária aos elementos da lista.