Linguagens de Programação

Expressões Lambda e Funções de Alta Ordem

Samuel da Silva Feitosa

Aula 11

Expressões Lambda

Valores de Primeira Classe

- São valores de primeira classe:
  - Números, caracteres, tuplas, listas, **funções**, entre outros.
- Valores de vários tipos podem ser escritos literalmente, sem a necessidade de dar um nome a eles:

valor	tipo	descrição
`True`	`Bool`	o valor lógico *verdadeiro*
`'G'`	`Char`	o caracter *G*
`456`	`Num a => a`	o número 456
`2.45`	`Fractional a => a`	o número em ponto flutuante 2.45
`"haskell"`	`String`	a cadeia de caracteres *haskell*
`[1,6,4,5]`	`Num a => [a]`	a lista dos números 1, 6, 4, 5
`("Ana",False)`	`([Char],Bool)`	o par formado por *Ana* e *falso*

Valores de Primeira Classe

- Funções também podem ser escritas sem a necessidade de receber um nome:
  - Estas são conhecidas por **funções anônimas** ou **expressões *lambda***.

valor	tipo	descrição
`\x -> 3*x`	`Num a => a -> a`	função que calcula o triplo
`\n -> mod n 2 == 0`	`Integral a => a -> Bool`	função que verifica se é par
`\(p,q) -> p+q`	`Num a => (a,a) -> a`	função que soma par

- **Expressão *lambda*** é uma função anônima (sem nome), formada por uma seqüência de padrões representando os argumentos da função, e um corpo que especifica como o resultado pode ser calculado usando os argumentos:

Exemplos de expressões *lambda*

- Função anônima que calcula o dobro de um número:

```
\x -> x + x
```

- Função anônima que mapeia um número *x* a 2**x* + 1:

```
\x -> 2*x + 1
```

- Função anônima que recebe três argumentos e calcula a sua soma:

```
\a b c -> a + b + c
```

Uso de expressões *lambda*

- Apesar de não terem um nome, funções construídas usando expressões lambda podem ser usadas da mesma maneira que outras funções.

```
(\x -> 2*x + 1) 8
⇝ 17
```

Aplicação parcial de funções

- Uma função com múltiplos argumentos pode também ser considerada como uma função que retorna outra função como resultado.

```
f    :: Int -> Int -> Int
f x y = 2*x + y
```

- Ela pode ser aplicada a apenas um argumento, resultando em outra função, que espera o segundo argumento.
  - Isso permite a *aplicação parcial* da função.

Funções de alta ordem

Funções de alta ordem

- Uma **função de alta ordem** é uma função que:
  - tem outra **função** como argumento; ou
  - produz uma **função** como resultado.

A função *map*

- A função *map* recebe uma função e uma lista como argumentos e aplica esta função a cada um dos elementos da lista, resultando na lista de resultados.
- Exemplos:

```
map sqrt [0,1,4,9]                                ⇝ [0.0,1.0,2.0,3.0]
map succ "HAL"                                    ⇝ "IBM"
map head ["bom","dia","turma"]                    ⇝ "bdt"
map even [8,10,-3,48,5]                           ⇝ [True,True,False,True,False]
map isDigit "A18 B7"                              ⇝ [False,True,True,False,False,True]
map length ["ciênca", "da", "computação"]         ⇝ [6,2,10]
map (sqrt.abs.snd) [('A',100),('Z',-36)]          ⇝ [10,6]
```

Definição de *map*

```haskell
map          :: (a -> b) -> [a] -> [b]

map _ []     = []
map f (x:xs) = f x : map f xs
```

As funções *foldl* e *foldr*

- As funções *foldl* e *foldr* reduzem uma lista, usando uma função binária e um valor inicial.
  - *foldl* é associativa a esquerda e *foldr* é associativa a direta.
- Exemplos:

```
foldl (+) 0 []                                  ⇝ 0
foldl (+) 0 [1]                                 ⇝ 1
foldl (+) 0 [1,2]                               ⇝ 3
foldl (+) 0 [1,2,4]                             ⇝ 7
foldl (*) 1 [5,2,4,10]                          ⇝ 400
foldl (&&) True [2>0,even 6,odd 5,null []]      ⇝ True
foldl (||) False [2>3,even 6,odd 5,null []]     ⇝ True
```

Definições de *foldl* e *foldr*

```haskell
foldl           :: (a -> b -> a) -> a -> [b] -> a

foldl f z []     = z
foldl f z (x:xs) = foldl f (f z x) xs
```

```haskell
foldr           :: (a -> b -> b) -> b -> [a] -> b

foldr f z []     = z
foldr f z (x:xs) = f x (foldr f z xs)
```

A função *filter*

- A função *filter* recebe uma função e uma lista de argumentos e seleciona (filtra) os elementos da lista para os quais a função dada resulta em verdadeiro.
- Exemplos:

```
filter even [1,8,10,48,5,-3]           ⇝ [8,10,48]
filter odd [1,8,10,48,5,-3]            ⇝ [1,5,-3]
filter isDigit "A186 B70"              ⇝ "18670"
filter (not . null) ["abc","","ok",""] ⇝ ["abc", "ok"]
```

- A função *isDigit* não faz parte do prelude. É preciso importá-la.

```
import Data.Char
```

Definição de *filter*

```haskell
filter                    :: (a -> Bool) -> [a] -> [a]

filter _ []                = []
filter f (x:xs) | f x      = x : filter f xs
                | otherwise = filter f xs
```

Considerações Finais

- Nesta aula estudamos conceitos que subsidiam a programação funcional.
  - Diferentes tipos para valores de primeira classe, os quais podem ser argumentos.
  - As expressões lambda que permitem a criação e aplicação de funções anônimas.
  - E as funções de alta ordem, que permitem que funções sejam recebidas como argumentos para aplicar um determinada estrutura de dados.
- Para exemplificar, vimos exemplos das funções *map*, *fold* e *filter*.

Exercício

- Utilize uma **função de alta ordem** qualquer passando como parâmetro uma **expressão lambda** e uma lista de elementos.