Aula prática 14

Expressão Lambda

Resumo

Expressões lambdas são funções anônimas que podem ser usadas como qualquer outro valor de primeira classe. Nesta aula vamos aprender sobre expressões lambda.

Sumário

1 Valores de primeira classe

1.1 Valores de primeira classe

- Tipo de **primeira classe**: não há restrições sobre como os seus valores podem ser usados.
- São valores de primeira classe:
  - números
  - caracteres
  - tuplas
  - listas
  - *funções*

  entre outros

1.2 Valores de primeira classe: Literais

- 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*

- Funções também podem ser escritas sem a necessidade de receber um nome:

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

1.3 Valores de primeira classe: Variáveis

- Valores de vários tipos podem ser *nomeados*:

```
matricula   = 456
sexo        = 'M'
aluno       = ("Ailton Mizuki Sato",101408,'M',"com")
disciplinas = ["BCC222","BCC221","MTM153","PRO300"]
livroTexto  = ("Programming in Haskell","G. Hutton",2007)
```

- Funções também podem ser nomeadas:

```
triplo = \x -> 3*x
```

Esta equação define a variável `triplo`, associando-a a um valor que é uma função.

Haskell permite escrever esta definição de forma mais sucinta:

```
triplo x = 3 * x
```

1.4 Valores de primeira classe: Argumentos

- Valores de vários tipos podem ser *argumentos* de funções:

```
sqrt 2.45
not True
length [1,6,4,5]
take 5 [1,8,6,10,23,0,0,100]
```

- Funções também podem ser argumentos de outras funções:

```
map triplo [1,2,3] ⇝ [3,6,9]
```

A função `triplo` é aplicada a cada elemento da lista `[1,2,3]`, resultando na lista `[3,6,9]`

1.5 Valores de primeira classe: Resultado

- Valores de vários tipos podem ser *resultados* de funções:

```
not False        ⇝ True
length [1,6,4,5] ⇝ 4
snd ("Ana",'F')  ⇝ 'F'
tail [1,6,4,5]   ⇝ [6,4,5]
```

- Funções também podem ser resultados de outras funções:

```
(abs . sin) (3*pi/2) ⇝ 1.0
(sqrt . abs) (-9)    ⇝ 3.0
```

O operador binário infixo `(.)` faz a composição de duas funções.

1.6 Valores de primeira classe: Componentes

- Valores de vários tipos podem ser *componentes* de outros valores:

```
("Ana",'F',18)
["BCC222","BCC221","MTM153","PRO300"]
[("Ailton",101408),("Lidiane",102408)]
```

- Funções também podem ser componentes de outros valores:

```
map (\g -> g (-pi)) [abs,sin,cos]
⇝ [3.141592653589793,-1.2246467991473532e-16,-1.0]
```

O segundo argumento de `map` é a lista das funções `abs`, `sin` e `cos`.

2 Expressão lambda

2.1 Expressões lambda

- Da mesma maneira que um número inteiro, uma string ou um par podem ser escritos sem ser nomeados, uma função também pode ser escrita sem associá-la a um nome.
- **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:

```
\padrão1 ... padrãon -> expressao
```

- O termo *lambda* provém do cálculo lambda (teoria de funções na qual as linguagens funcionais se baseiam), introduzido por Alonzo Church nos anos 1930 como parte de uma investigação sobre os fundamentos da Matemática.
- No cálculo lambda expressões lambdas são introduzidas usando a letra grega λ. Em Haskell usa-se o caracter `\`, que se assemalha-se um pouco com λ.

2.2 Exemplos de expressões lambda

Função anônima que calcula o dobro de um número:

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

O tipo desta expressão lambda é `Num a => a -> a`
Função anônima que mapeia um número x a $2x + 1$:

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

cujo tipo é `Num a => a -> a`
Função anônima que calcula o fatorial de um número:

```
\n -> product [1..n]
```

cujo tipo é `(Enum a, Num a) => a -> a`
Função anônima que recebe três argumentos e calcula a sua soma:

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

cujo tipo é `Num a => a -> a -> a -> a`
Definições de função usando expressão lambda:

```
f       = \x -> 2*x + 1
somaPar = \(x,y) -> x + y
fatorial = \n -> product [1..n]
```

é o mesmo que

```
f x           = 2*x + 1
somaPar (x,y) = x + y
fatorial n    = product [1..n]
```

2.3 Uso de expressões lambda

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

```
(\x -> 2*x + 1) 8
⇝ 17
```

```
(\a -> (a,2*a,3*a)) 5
⇝ (5,10,15)
```

```
(\x y -> sqrt (x*x + y*y)) 3 4
⇝ 5.0
```

```
(\xs -> let n = div (length xs) 2 in (take n xs,drop n xs)) "Bom dia"
⇝ ("Bom", " dia")
```

```
(\(x1,y1) (x2,y2) -> sqrt ((x2-x1)^2 + (y2-y1)^2)) (6,7) (9,11)
⇝ 5.0
```

2.4 Exercícios

Tarefa 1

Escreva uma função anônima que recebe uma tripla formada pelo nome, peso e altura de uma pessoa e resulta no seu índice de massa corporal, dado pela razão entre o peso e o quadrado da altura da pessoa.

Tarefa 2

Escreva uma expressão para selecionar (filtrar) os elementos múltiplos de 3 em uma lista de números. Utilize a função `filter :: (a -> Bool) -> [a] -> [a]` do prelúdio. Especifique a função que determina a propriedade a ser satisfeita pelos elementos selecionados usando uma expressão lambda.

Tarefa 3

Determine o tipo mais geral da seguinte expressão:

```
\a (m,n) -> if a then (m+n)^2 else (m+n)^3
```

Dica: do prelúdio tem-se
`(^) :: (Num a, Integral b) => a -> b -> a`.

Tarefa 4

Composição de funções é uma operação comum em Matemática, que a define como

$$(f \circ g)(x) = f(g(x))$$

Em Haskell podemos definir uma função para compor duas outras funções dadas como argumentos. O resultado é uma função: a função composta.

Definia a função `composta` que recebe duas funções como argumentos e resulta na função composta das mesmas. Use uma definição local para definir a função resultante:

```
composta f g = ···
  where
    ···
```

Tarefa 5

1. Escreva outra definição para a função `composta` usando uma expressão lambda para determinar o seu resultado. Nesta versão não use definições locais.
2. Determine o tipo mais geral da função `composta`.
3. Teste a função `composta` calculando o tipo e o valor da expressão

   ```
   (composta even length) "linguagens modernas"
   ```

Tarefa 6

O módulo **Prelude** define o operador binário `(.)` para fazer composição de funções. Este operador tem precedência 9 e associatividade à direira:

```
infixr 9 .
(.) :: (b -> c) -> (a -> b) -> a -> c
```

Determine o tipo e o valor das seguintes expressões que usam composição de funções e expressões lambda:

1. `(toUpper . head) ["maria","jose","silva"]`
2. `(not . odd . length) "felicidade"`
3. `(isLetter . head . head . reverse) ["maria","silva","pereira"]`
4. `(even . (\x -> x*2 + 3) . (\x -> div x 2) . snd) (9+4,9-4)`

3 Aplicação parcial de funções

3.1 Aplicação parcial de funções

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

3.2 Aplicação parcial de funções: exemplos

- Seja a seguinte função:

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

A função `f` recebe dois argumentos inteiros `x` e `y` e resulta na soma `2*x + y`.

- Alternativamente esta função pode ser definida em duas etapas:

```
f'  :: Int -> (Int -> Int)
f' x = h
  where h y = 2*x + y
```

A função `f'` recebe um argumento inteiro `x` e resulta na função `h`, que por sua vez recebe um argumento inteiro `y` e calcula `2*x + y`.

- Aplicando a função:

```
f' 2 3
⇝ (f' 2) 3
⇝ h 3
⇝ 2*2 + 3
⇝ 7
```

- As funções `f` e `f'` produzem o mesmo resultado final, mas `f` foi definida de uma forma mais breve.
- Podemos ainda definir a função usando uma expressão lambda:

```
f''  :: Int -> (Int -> Int)
f'' x =
  \y -> 2*x + y
```

Da mesma forma que `f'`, a função `f''` recebe um argumento inteiro `x` e resulta em uma função. Esta função recebe um argumento inteiro `y` e calcula `2*x + y`.

- Aplicando a função:

```
f'' 2 3
⇝ (f'' 2) 3
⇝ (\y -> 2*2 + y) 3
⇝ 2*2 + 3
⇝ 7
```

- Podemos ainda definir a função usando duas expressões lambda:

```
f''' :: Int -> (Int -> Int)
f'''  =
  \x -> (\y -> 2*x + y)
```

- Aplicando a função:

```
f''' 2 3
⇝ (\x -> (\y -> 2*x + y)) 2 3
⇝ (\y -> 2*2 + y) 3
⇝ 2*2 + 3
⇝ 7
```

- Todas as versões apresentadas para a função `f` (`f`, `f'`, `f''` e `f'''`) são equivalentes.

- Portanto a função `f` pode ser considerada como uma função que recebe um argumento e resulta em outra função que, por sua vez, recebe outro argumento e resulta na soma do dobro do primeiro argumento com o segundo argumento.

- Isto permite a *aplicação parcial* da função:

```
let g = f 5 in (g 8, g 1)
⇝ (18,11)

map (f 2) [1,8,0,19,5]
⇝ [5,12,4,23,9]

(f 2 . length) "entendeu?"
⇝ 13

filter (not . even . f 10) [1,8,0,19,5]
⇝ [1,19,5]
```

- Outro exemplo: multiplicação de três números:

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

A função `mult` recebe três argumentos e resulta no produto destes argumentos.

- Na verdade `mult` recebe um argumento de cada vez. Ou seja, `mult` recebe um inteiro `x` e resulta em uma função que por sua vez recebe um inteiro `y` e resulta em outra função, que finalmente recebe um inteiro `z` e resulta no produto `x * y * z`.

- Este entendimetno fica claro quando usamos expressões lambda para definir a função de maneira alternativa:

```
mult' :: Int -> (Int -> (Int -> Int))
mult'  = \x -> \y -> \z -> x * y * z
```

4 *Currying*

4.1 Funções *curried*

- Outra 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.
- 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 que *recebem os seus argumentos um por vez* são chamadas de **funções *curried***[1], celebrando o trabalho de **Haskell Curry** no estudo de tais funções.
- Funções com mais de um argumento *curried*, resultando em funções aninhadas.

4.2 Por que *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
```

4.3 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))
```

- 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*.

[1] Funções *curried* às vezes são chamadas de **funções currificadas** em português.

5 Seções de operadores

5.1 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
```

- 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  -}
```

- 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
```

- 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
```

  – `(>)` 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]
```

5.2 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.
- **Exemplo:**

```
(1+)
```

  é a função que incrementa (soma um) ao seu argumento. É o mesmo que

```
\x -> 1 + x
```

```
(1+) 8 ⇝ 9
```

- **Exemplo:**

```
(*2)
```

é a função que dobra (multiplica por 2) o seu argumento. É o mesmo que

```
\x -> x * 2
```

```
(*2) 8 ⇝ 16
```

- **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
```

- **Exemplo:**

```
(<0)
```

é a função que verifica se o seu argumento é negativo. É o mesmo que

```
\x -> x < 0
```

```
(<0) 8 ⇝ False
```

- Outros **Exemplos** de aplicação de seções de operador:

```
(1+) 2 ⇝ 3
(+1) 2 ⇝ 3
```

```
(100>) 200 ⇝ False
(>100) 200 ⇝ True
```

```
([1,2]++) [30,40,50] ⇝ [1,2,30,40,50]
(++[1,2]) [30,40,50] ⇝ [30,40,50,1,2]
```

- Em geral, se $\oplus$ é um operador binário infixo, então as formas

$$(\oplus)$$
$$(x\ \oplus)$$
$$(\oplus\ y)$$

são chamados de **seções**.

- Seções são equivalentes às definições com expressões lambdas:

```
(⊕) = \x y -> x ⊕ y
```

```
(x ⊕) = \y -> x ⊕ y
```

```
(⊕ y) = \x -> x ⊕ y
```

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

```
(subtract x)
```

```
(subtract 8) 10 ⇝ 2
```

6 Utilidade de expressões lambda

6.1 Por que seções são úteis?

- *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

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

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

- 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.

6.2 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*.
- **Exemplo**:
  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`.

```
soma
⇝ \x -> (\y -> x + y)
```

```
soma 2
⇝ (\x -> (\y -> x + y)) 2
⇝ \y -> 2 + y
```

```
soma 2 3
⇝ (\x -> (\y -> x + y)) 2 3
⇝ (\y -> 2 + y) 3
⇝ 2 + 3
⇝ 5
```

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

- **Exemplo:**
  A função `const` definida na biblioteca retorna como resultado uma função constante, que sempre resulta em um dado valor:

```
const    :: a -> b -> a
const x _ = x
```

```
const 6 0  ⇝ 6
const 6 1  ⇝ 6
const 6 2  ⇝ 6
const 6 9  ⇝ 6
const 6 75 ⇝ 6
```

```
h = const 6 ⇝ \_ -> 6
```

```
h 0  ⇝ 6
h 4  ⇝ 6
h 75 ⇝ 6
```

  A função `const` pode ser definida de uma maneira mais natural usando expressão lambda, tornando explícito que o resultado é uma função:

```
const  :: a -> (b -> a)
const x = \_ -> x
```

- Expressões lambda podem ser usadas para evitar a nomeação de funções que são *referenciados apenas uma vez*.

- **Exemplo:**
  A função

```
impares n = map f [0..n-1]
  where
    f x = x*2 + 1
```

  que recebe um número n e retorna a lista dos n primeiros números ímpares, pode ser simplificada:

```
impares n = map (\x -> x*2 + 1) [0..n-1]
```

6.3 Exercícios

Tarefa 7

Para cada uma das seguintes funções:

- descreva a função
- determine o tipo mais geral da função
- reescreva a função usando expressões lambda ao invés de seções de operadores

a `('c':)`

b `(:"fim")`

c `(==2)`

d `(++"\n")`

e `(^3)`

f `(3^)`

g `(`elem` "AEIOU")`

Tarefa 8

Determine o valor da expressão:

```
let pares = [(1,8),(2,5),(0,1),(4,4),(3,2)]
    h = sum . map (\(x,y) -> x*y-1) . filter (\(x,_) -> even x)
in h pares
```

Tarefa 9

Mostre como a definição de função *curried*

```
mult x y z = x * y * z
```

pode ser entendida em termos de expressões lambda.
Dica: Redefina a função usando expressões lambda.