

Programação Funcional

3ª Aula — Definição de funções

PROF. BONFIM AMARO JUNIOR

bonfimamaro@ufc.br

Definição de funções

Podemos definir novas funções simples usando funções pré-definidas.

```
minuscula :: Char -> Bool
minuscula c = c>='a' && c<='z'

fact :: Int -> Int
fact n = product [1..n]
```

Definição de funções

Ex) Definir uma função para calcular a média ponderada MP

Entrada: uma lista do tipo Float

Saída: Média (Float)

ENTRADA : [1 , 2 , 3 , 4] SAÍDA : 3.0
--

Definição de funções

Ex) Definir uma função para calcular a média ponderada MP

Código

```
1 auxNum::Int->[Float]->Float
2 auxNum _ [] = 0
3 auxNum i (x:xs) = x*fromIntegral(i) + (auxNum (i+1) xs)
4
5 getNumerador::[Float]->Float
6 getNumerador = auxNum 1
7
8 getDenominador::Int->Int
9
10 getDenominador tam = sum[1..tam]
11
12 mediaPond::[Float]->Float
13 mediaPond lista = getNumerador lista / fromIntegral (getDenominador (length lista))
```

Expressões condicionais

Podemos exprimir uma condição com duas alternativas usando 'if...then...else...'.

```
abs :: Float -> Float
abs x = if x>=0 then x else -x
```

As expressões condicionais podem ser:

```
senal :: Int -> Int
senal x = if x>0 then 1 else
           if x==0 then 0 else -1
```

Em Haskell, ao contrário do C/C++/Java, a alternativa 'else' é **obrigatória**.

Alternativas com guardas

Podemos usar **guardas** em vez de expressões condicionais:

```
sinal :: Int -> Int
sinal x | x>0      = 1
        | x==0     = 0
        | otherwise = -1
```

- Testa as condições pela ordem no programa.
- Seleciona a primeira alternativa verdadeira.
- Se nenhuma condição for verdadeira: erro de execução.
- A condição 'otherwise' é o caso de exceção

Alternativas com guardas (cont.)

Exemplo: as raízes de uma equação do 2º grau.

Definir uma função para estabelecer as raízes de uma equação

$$(a)x^2 + (b)x + c = 0$$

Retornar uma lista contendo a(s) raíze(s), caso exista(m)

$$x^2 - 5x + 6 = 0$$

$$\Delta = b^2 - 4ac$$

$$\Delta = (-5)^2 - 4 \cdot 1 \cdot 6$$

$$\Delta = 25 - 24$$

$$\Delta = 1$$

$$\begin{aligned}x &= \frac{-b \pm \sqrt{\Delta}}{2a} \\x &= \frac{-(-5) \pm \sqrt{1}}{2 \cdot 1} \\x' &= \frac{5+1}{2} = \frac{6}{2} = 3 \\x'' &= \frac{5-1}{2} = \frac{4}{2} = 2\end{aligned}$$

Alternativas com guardas (cont.)

Definições locais abrangem todas as alternativas se a palavra 'where' for indentada como as guardas.

Exemplo: as raízes de uma equação do 2º grau.

```
raizes :: Float -> Float -> Float -> [Float]
raizes a b c
  | delta>0    = [(-b+sqrt delta)/(2*a),
                  (-b-sqrt delta)/(2*a)]
  | delta==0   = [-b/(2*a)]
  | otherwise  = []
  where delta  = b^2 - 4*a*c
```


Alternativas com guardas (cont.)

Também podemos definir nomes locais a uma expressão usando 'let...in...'. Neste caso o âmbito da definição **não** inclui as outras alternativas.

```
raizes :: Float -> Float -> Float -> [Float]
raizes a b c
  | delta>0    = let r = sqrt delta
                  in [(-b+r)/(2*a), (-b-r)/(2*a)]
  | delta==0   = [-b/(2*a)]
  | otherwise  = []
where delta = b^2 - 4*a*c
```

Alternativas com guardas (cont.)

Exemplo_1)

Escreva uma função “returnElement” recebe uma lista e um valor (Ambos Inteiros) usando IF

Retorna a quantidade de elementos encontrados na lista de entrada

Exemplo_2)

Escreva uma função “returnElement” recebe uma lista e um valor (Genérico) usando Guarda

Retorna a quantidade de elementos encontrados na lista de entrada

Encaixe de padrões

Podemos usar **múltiplas equações com padrões** para distinguir argumentos.

```
not :: Bool -> Bool
```

```
not True = False
```

```
not False = True
```

```
(&&) :: Bool -> Bool -> Bool
```

```
True && True = True
```

```
True && False = False
```

```
False && True = False
```

```
False && False = False
```

Encaixe de padrões (cont.)

Uma definição alternativa:

```
(&&) :: Bool -> Bool -> Bool
True  && x = x
False && _ = False
```

Esta definição não avalia o segundo argumento se o primeiro for `False`.

- O padrão “`_`” encaixa qualquer valor.
- As variáveis no padrão podem ser usadas no lado direito.

Encaixe de padrões (cont.)

Os padrões para alternativa não podem repetir variáveis:

```
x && x = x  
_ && _ = False
```

-- ERRO

Podemos usar guardas para impor igualdade:

```
x && y | x==y = x  
_ && _       = False
```

-- OK

Padrões sobre tuplas

Exemplos: as projeções de pares (no prelúdio-padrão).

```
fst :: (a,b) -> a
```

```
fst (x,_) = x
```

```
snd :: (a,b) -> b
```

```
snd (_,y) = y
```

Padrões sobre listas

Qualquer lista é construída acrescentando elementos um-a-um à lista vazia usando o operador ':' (lê-se "*cons*").

```
[1, 2, 3, 4] = 1 : (2 : (3 : (4 : [])))
```

Podemos também usar um padrão `x:xs` para decompor uma lista.

```
head :: [a] -> a
```

```
head (x:_) = x -- 1º elemento
```

```
tail :: [a] -> [a]
```

```
tail (_:xs) = xs -- restantes elementos
```

Padrões sobre listas (cont.)

O padrão `x:xs` só encaixa **listas não-vazias**:

```
> head []  
ERRO
```

São necessários parêntesis à volta do padrão (aplicação têm maior precedência que operadores):

```
head x:_ = x -- ERRO
```

```
head (x:_) = x -- OK
```


Padrões sobre inteiros

Exemplo: testar se um inteiro é 0, 1 ou -1.

```
small :: Int -> Bool
small 0      = True
small 1      = True
small (-1)   = True
small _      = False
```

A última equação encaixa todos os restantes casos.

Expressões-case

Em vez de equações podemos usar 'case...of...':

Exemplo:

```
null :: [a] -> Bool
null xs = case xs of
    [] -> True
    (_:_) -> False
```

Expressões-case (cont.)

Os padrões são tentados pela ordem das alternativas.

Logo, esta definição é equivalente à anterior:

```
null :: [a] -> Bool
null xs = case xs of
            [] -> True
            _  -> False
```

Expressões-lambda

Podemos definir uma função anônima (i.e. sem nome) usando uma **expressão-lambda**.

Exemplo:

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

é a função que a cada x faz corresponder $2x + 1$.

Esta notação é baseada no *cálculo- λ* , um formalismo matemático que é a base da programação funcional.

Expressões-lambda (cont.)

Podemos aplicar a expressão-lambda a um valor (tal como uma função com nome).

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

```
3
```

```
> (\x -> 2*x+1) 3
```

```
7
```

Porquê usar expressões-lambda?

As expressões-lambda permitem definir **funções cujos resultados são outras funções**.

Em particular, usando expressões-lambda podemos definir formalmente a transformação de “*currying*”.

Exemplo:

```
soma x y = x+y
```

é equivalente a

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

Porquê usar expressões-lambda? (cont.)

Exercício 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.

Exercício 2)

Escreva uma expressão para seleccionar (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 seleccionados usando uma expressão lambda.

Porquê usar expressões-lambda? (cont.)

As expressões-lambda são convenientes para evitar dar nomes a expressões curtas usadas apenas uma vez.

Um exemplo: *map* aplica uma função a todos os elementos de listas.

Em vez de

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

podemos escrever

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


Porquê usar expressões-lambda? (cont.)

Exemplo: Área de um círculo

$$F \equiv \pi * r^2, \quad r \in \mathbb{N}$$

$$Q \equiv x * x, \quad x \in \mathbb{N}$$

$$H = F \circ Q \quad (\text{função composta})$$

$$H(r) = 3,14 * Q(r)$$

Abstração Lambda: $\lambda y. \pi * y * y$

Porquê usar expressões-lambda? (cont.)

Exemplo: Área de um círculo

$$F \equiv \pi * r^2, \quad r \in \mathbb{N}$$

$$Q \equiv x * x, \quad x \in \mathbb{N}$$

$$H = F \circ Q \quad (\text{função composta})$$

$$H(r) = 3,14 * Q(r)$$

Abstração Lambda: $\lambda y. \pi * y * y$

```
Em Haskell:      Main> (\y -> pi * y * y) 3  
                  28.2743338823081
```

Qualquer operador binário \oplus pode ser usado como função de dois argumentos escrevendo-o entre parentêsis (\oplus).

Exemplo:

```
> 1+2
```

```
3
```

```
> (+) 1 2
```

```
3
```

Seções (cont.)

Também podemos incluir um dos argumentos dentro do parêntesis para exprimir *uma função do outro argumento*.

```
> (+1) 2
```

```
3
```

```
> (2+) 1
```

```
3
```

Em geral: expressões da forma (\oplus) , $(x\oplus)$ e $(\oplus y)$ e \oplus designam-se **seções** e definem funções resultantes de aplicar parcialmente \oplus .

Seções (cont.)

Alguns exemplos:

$(1+)$	sucessor
$(2*)$	dobro
$(^2)$	quadrado
$(/2)$	metade fraccionária
$(\text{'div' } 2)$	metade inteira