

Programação Funcional

3ª Aula — Definição de funções

Sandra Alves
DCC/FCUP

2019/20

Definição de funções

- ▶ Um script em Haskell é um conjunto de definições de funções e valores
- ▶ Um valor é uma definição da forma
 $v = e$
onde e é uma expressão em Haskell
- ▶ Funções podem ser definidas como
 $f\ p_{11} \dots p_{1k} = e_1$
 \dots
 $f\ p_{n1} \dots p_{nk} = e_n$
onde p_{ij} é um padrão e e_i é uma expressão

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]
```

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 embricadas:

```
sinal :: Int -> Int
sinal 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 I

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 um sinónimo de True.

Alternativas com guardas II

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 III

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
```

Encaixe de padrões I

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 II

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 III

Os padrões numa alternativa não podem repetir variáveis:

```
x && x = x
```

-- ERRO

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

Podemos usar guardas para impor igualdade:

```
x && y | x==y = x
```

-- OK

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

Padrões sobre tuplos

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 I

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 II

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

```
> head []  
ERRO
```

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

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

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

Padrões sobre inteiros I

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.

Padrões sobre inteiros II

Padrões $n+k$ (n é uma variável e k é uma constante).

```
anterior :: Int -> Int  
anterior (n+1) = n
```

- ▶ O padrão $n+k$ só encaixa inteiros $\geq k$
- ▶ É necessário usar parentêsis em torno do padrão

Não suportada a partir do Haskell 2010; alternativa:

```
anterior :: Int -> Int  
anterior n | n>=1 = n-1
```

Expressões-case I

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

Exemplo:

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


Expressões-case II

Os padrões são tentados pela ordem das alternativas.

Logo, a esta definição é equivalente à anterior:

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

Expressões-lambda I

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

Exemplo:

$$\backslash x \rightarrow 2 * x + 1$$

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 II

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? I

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 = $\backslash x \rightarrow (\backslash y \rightarrow x+y)$

Porquê usar expressões-lambda? II

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 duma lista.

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]
```

Seções I

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 II

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 III

Alguns exemplos:

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

Exercícios:

- ▶ Indique três definições possíveis para o operador lógico *or* usando encaixe de padrão
- ▶ Implemente a função *sinal* usando:
 - ▶ expressões com guardas
 - ▶ expressões case
 - ▶ encaixe de padrão