Programação Funcional
Aula 3 — Definição de funções

Pedro Vasconcelos
DCC/FCUP

2021

Definição de funções

Podemos definir novas funções simples como expressões usando outras funções pré-definidas.

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

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

Expressões condicionais

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

```
-- valor absoluto: x se x>=0; -x se x<0
absoluto :: Float -> Float
absoluto x = if x>=0 then x else -x
```

As expressões condicionais podem ser imbricadas:

```
sinal :: Int -> Int
sinal x = if x>0 then 1 else
            (if x==0 then 0 else -1)
```

Ao contrário do C ou Java:

- o 'if...then ...else' é uma expressão e não um comando
- a alternativa 'else' é obrigatória

Alternativas com guardas

Podemos usar guardas em vez de expressões condicionais:

```
absoluto :: Float -> Float
absoluto | x>=0      = x
         | otherwise = -x


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

Alternativas com guardas (cont.)

Caso geral:

```
f x y ... | condição 1 = expressão 1
          | condição 2 = expressão 2
            ...
          | condição N = expressão N
```

- As condições são testada pela ordem indicada
- O resultado é dado pela expressão da primeira alternativa verdadeira
- A função é indefinida se nenhuma condição for verdadeira (dá erro durante a execução)
- A condição 'otherwise' é um sinónimo para `True`

Alternativas com guardas (cont.)

Definições locais abrangem todas as alternativas se a palavra 'where' estiver alinhada com as guardas.

Exemplo: raizes 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 variáveis locais usando 'let...in...'.
Neste caso o âmbito da definição não inclui 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)]
        -- r só está definido na expressão acima
     | delta==0  = [-b/(2*a)]
     | otherwise = []
     where delta = b^2 - 4*a*c
```

Encaixe de padrões

Podemos usar várias equações com padrões para distinguir casos.

```
not :: Bool -> Bool
not True = False
not False = True

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

Nota: estas funções estão pré-definidas no prelúdio-padrão.

Encaixe de padrões (cont.)

Uma definição alternativa:

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

- O padrão "_" encaixa qualquer valor
- As variáveis no padrão podem ser usadas no lado direito
- A definição acima ignora o segundo argumento se o primeiro for `False`

Encaixe de padrões (cont.)

Não podemos repetir variáveis nos padrões:

```
x && x = x                    -- ERRO
_ && _ = False
```

Alternativa: podemos usar uma guarda para impor a condição de igualdade.

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

Padrões sobre tuplos

Exemplos: as funções `fst` (*first*) e `snd` (*second*) dão-nos o primeiro e segundo elemento de um par.

```
fst :: (a,b) -> a
fst (x,_) = x

snd :: (a,b) -> b
snd (_,y) = y
```

Estas funções também estão pré-definidas no prelúdio-padrão.

Construtor de listas

Qualquer lista é construida acrescentando elementos um-a-um a uma lista vazia usando "`:`"[1].

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

[1] Lê-se "*cons*" de "*construtor*".

Padrões sobre listas

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

Necessitamos de parêntesis à volta do padrão (porque a aplicação têm maior precedência):

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

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

Padrões sobre inteiros

Exemplo: testar se um inteiro é 0, 1 ou 2.

```
small :: Int -> Bool
small 0    = True
small 1    = True
small 2    = True
small _    = False
```

A última equação encaixa todos os casos restantes.

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, a esta definição é equivalente à anterior:

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

Expressões-lambda

Podemos definir uma *função anónima* (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, o formalismo matemático que é a base teórica da programação funcional.

Expressões-lambda (cont.)

Podemos usar uma expressão-lambda aplicando-a a um valor (tal como o nome de uma função).

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

> (\x -> 2*x+1) 3
7
```

Para que servem as expressões-lambda?

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

Em particular: as expressões-lambda permitem compreender o uso de *"currying"* para funções de múltiplos argumentos.

Exemplo:

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

é equivalente a

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

Para que servem as expressões-lambda? (cont.)

As expressões-lambda são também úteis para evitar dar nomes a funções curtas.

Um exemplo usando `map` (que aplica uma função a todos os elementos duma lista): em vez de

```
quadrados = map f [1..10]
      where f x = x^2
```

podemos escrever

```
quadrados = map (\x->x^2) [1..10]
```

Operadores e secções

Qualquer operador binário (+, ∗, etc.) pode ser usado como função de dois argumentos colocando-o entre parentêsis.

Exemplo:

```
> 1+2
3

> (+) 1 2
3

> (++) "Abra" "cadabra!"
"Abracadabra!"
```

Operadores e secções (cont.)

- Expressões da forma `(x ⊗)` e `(⊗ x)` chamam-se secções
- Definem a função resultante de aplicar um dos argumentos do operador ⊗

```
> (+1) 2
3

> (/2) 1
0.5

> (++"!!!") "Bang"
"Bang!!!"
```

Exemplos

`(1+)`	sucessor
`(2*)`	dobro
`(^2)`	quadrado
`(1/)`	recíproco
`(++"!!")`	concatenar `"!!"` ao final

Assim podemos re-escrever o exemplo

```
quadrados = map (\x -> x^2) [1..10]
```

de forma ainda mais sucinta:

```
quadrados = map (^2) [1..10]
```