Programação Funcional com Haskell – Definição de funções

Prof. Dr. Eduardo Takeo Ueda
eduardo.tueda@sp.senac.br

Definição de funções

- Podemos definir novas funções simples usando as funções pré-definidas no prelude

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

  - fatorial :: Int -> Int
    fatorial n = product [1..n]

Expressões Condicionais

- Podemos expressar 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 aninhadas
  - sinal :: Int -> Int

    sinal x = if x>0 then 1 else

    if x==0 then 0 else -1

- A alternativa ‘else’ é obrigatória

Alternativas com guardas (1/2)

- Podemos usar alternativas com 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 sinônimo de True

Casamento de Padrões (1/3)

- Podemos usar múltiplas equações com padrões para descriminar os 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

Casamento de padrões (2/3)

- Uma definição alternativa da conjunção
  - (&&) :: Bool -> Bool -> Bool

    True && x = x

    False && _ = False
  - Esta definição não avalia o segundo argumento se o primeiro for False

- O padrão _ casa/encaixa com qualquer valor

- As variáveis no padrão podem ser usadas no lado direito

Casamento de padrões (3/3)

- Padrões não podem repetir variáveis
  - x && x = x **-- Erro**
  - _ && _ = False

- Podemos sempre usar guardas para impor uma condição
  - x && y | x==y = x **-- OK**
    _ && _ = False

Padrões sobre Tuplas

- Projeções de pares (Standard Prelude)
  - fst :: (a,b) -> a

    fst (x,_) = x

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

    snd (_,y) = y

Padrões sobre Listas (1/2)

- Qualquer lista é construida 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 -- elementos restantes

Padrões sobre Listas (2/2)

- O padrão x:xs só encaixa listas não-vazias
  - Hugs> head [ ]

    Program error: pattern match failure: head []

- São necessários parêntesis à volta do padrão porque a aplicação têm maior precedência do que os operadores
  - head x:_ = x --Erro
  - head (x:_) = x --Ok

Padrões sobre Inteiros (1/2)

- 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 casa/encaixa todos os casos restantes

Padrões sobre Inteiros (2/2)

- Padrões n+k (n é uma variável e k é uma constante)
  - pred :: Int -> Int

    pred (n+1) = n

- O padrão n+k só encaixa inteiros >= k
- Sintaxe descontinuada no padrão Haskell 2010
  - pred :: Int -> Int

    pred n | n>=1 = n-1

Expressões case (1/2)

- Em vez de equações podemos usar uma expressão **‘case’** para descrever vários padrões

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

Expressões case (2/2)

- Tal como nas equações, o casamento de padrões é feito pela ordem das alternativas. Logo, a definição seguinte é equivalente à anterior

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

Expressões lambda (1/2)

- Podemos definir uma função **anônima** (i.e. sem nome) usando uma expressão lambda
  - \x -> 2*x+1 é uma expressão que a cada x faz corresponder 2x+1
- Notação baseada no cálculo lambda, o formalismo matemático que é a fundação da programação funcional
- No computador: usamos o símbolo \ em vez da letra grega lambda

Expressões lambda (2/2)

- Podemos usar uma expressão lambda tal como uma função convencional

  - Hugs> (\x -> 2*x+1) 1
    3
  - Hugs> (\x -> 2*x+1) 3
    7
  - Hugs> (\x -> x^2) 5
    32

Por que usar expressões lambda?

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

  - soma x y = x+y

    é equivalente a

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

Por que usar expressões lambda?

- As expressões lambda permitem evitar dar nomes a funções referidas apenas uma vez

- Um exemplo usando map (que aplica uma função a todos os elementos de uma lista)
  - impares n = map f [0..n-1]
    where f x = 2*x+1

    pode ser simplificado para
  - impares n = map (\x->2*x+1) [0..n-1]