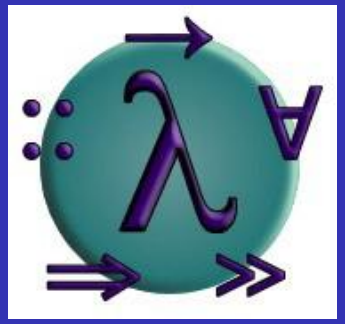
Programação Funcional



Aula 4

Definindo Funções

José Romildo Malaquias

Departamento de Computação
Universidade Federal de Ouro Preto

2011.2

Redefinindo funções do prelúdio

- O módulo `Prelude` é importado automaticamente em todos os módulos de uma aplicação em Haskell.
- Um nome que já tenha sido definido não pode ser redefinido.
- Como escrever uma definição usando um nome que já é utilizado em algum módulo?
- Podemos omitir alguns nomes ao importar um módulo, usando a declaração `import hiding`.

Redefinindo funções do prelúdio (cont.)

- Exemplo: para fazermos nossas próprias definições de `even` e `odd`, que já são definidas no módulo **`Prelude`**:

```haskell
import Prelude hiding (even, odd)

even n = mod n 2 == 0

odd n = not (even n)
```

Combinando funções

- A maneira mais simples de definir novas funções é simplesmente pela combinação de uma ou mais funções existentes.

Combinando funções (cont.)

- Exemplo: verificar se um caracter é um dígito decimal:

```
isDigit  :: Char -> Bool
isDigit c = c >= '0' && c <= '9'
```

Combinando funções (cont.)

- Exemplo: verificar se um número inteiro é par:

```haskell
even :: Integral a => a -> Bool
even n = n `mod` 2 == 0
```

Combinando funções (cont.)

- Exemplo: dividir uma lista em duas partes:

```
splitAt :: Int -> [a] -> ([a],[a])
splitAt n xs = (take n xs,drop n xs)
```

Combinando funções (cont.)

- Exemplo: calcular o recíproco de um número:

```
recip :: Fractional a => a -> a
recip n = 1/n
```

Expressões condicionais

- Uma expressão condicional tem a forma

```
if condição then exp1 else exp2
```

onde *condição* é uma expressão booleana (chamada predicado) e exp_1 (chamada consequência) e exp_2 (chamada alternativa) são expressões de um mesmo tipo.

- O valor da expressão condicional é o valor de exp_1 se a condição é verdadeira, ou o valor de exp_2 se a condição é falsa.
- Exemplos:

```
if True then 1 else 2          ⇒ 1
if False then 1 else 2         ⇒ 2
if 2>1 then "OK" else "FAIL"   ⇒ "OK"
```

Expressões condicionais (cont.)

- A expressão condicional é uma **expressão**, portanto sempre tem um valor.
- Assim uma expressão condicional pode ser usada dentro de outra expressão.
- Exemplos:

```
5 * (if True then 10 else 20)          ⇒ 50
5 * if True then 10 else 20            ⇒ 50
(if even 2 then 10 else 20) + 1        ⇒ 11
if even 2 then 10 else 20 + 1          ⇒ 10
length (if 2<=1 then "OK" else "FAIL") ⇒ 4
```

Expressões condicionais (cont.)

- A cláusula **`else`** não é opcional em uma expressão condicional. Omiti-la é um erro de sintaxe.
- Exemplos:

```
if True then 10 ⇒ ERRO DE SINTAXE
```

- Se fosse possível omiti-la, qual seria o valor da expressão quando a condição for falsa?

Expressões condicionais (cont.)

- Regra de inferência:

$$\frac{test :: \text{Bool} \qquad e_1 :: a \qquad e_2 :: a}{\text{if } test \text{ then } e_1 \text{ else } e_2 :: a}$$

- Observe que a consequência e a alternativa devem ser do mesmo tipo, que também é o tipo do resultado.
- Exemplos:

```
Prelude> :type if True then 10 else 20
if True then 10 else 20 :: Num a => a
Prelude> :type if 4>5 then "ok" else "bad"
if 4>5 then "ok" else "bad" :: [Char]
```

Expressões condicionais (cont.)

```
Prelude> if length [1,2,3] then "ok" else "bad"

<interactive>:0:4:
    Couldn’t match expected type ‘Bool’ with actual type ‘Int’
    In the return type of a call of ‘length’
    In the expression: length [1, 2, 3]
    In the expression: if length [1, 2, 3] then "ok" else "bad"
Prelude> if 4>5 then "ok" else ’H’

<interactive>:0:23:
    Couldn’t match expected type ‘[Char]’ with actual type ‘Char’
    In the expression: ’H’
    In the expression: if 4 > 5 then "ok" else ’H’
    In an equation for ‘it’: it = if 4 > 5 then "ok" else ’H’
```

Expressões condicionais (cont.)

- Como na maioria das linguagens de programação, funções podem ser definidas usando expressões condicionais.
- Exemplo: valor absoluto

```
abs :: Int -> Int
abs n = if n >= 0 then n else -n
```

`abs` recebe um inteiro `n` e retorna `n` se ele é não-negativo, e `-n` caso contrário.

Expressões condicionais (cont.)

- Expressões condicionais podem ser aninhadas.
- Exemplo: sinal de um número:

```
signum  :: Int -> Int
signum n = if n < 0
             then -1
             else if n == 0
                    then 0
                    else 1
```

Expressões condicionais (cont.)

- Em Haskell, expressões condicionais sempre devem ter as duas alternativas, o que evita qualquer possível problema de ambigüidade com expressões condicionais aninhadas.

Equações com guardas

- Funções podem ser definidas através de equações com guardas, onde uma sequência de expressões lógicas chamadas **guardas** é usada para escolher um resultado.
- Uma **equação com guarda** é formada por uma sequência de cláusulas escritas logo após a lista de argumentos. Cada cláusula é introduzida por uma barra vertical (|) e consiste em uma condição chamada **guarda** e uma expressão (resultado), separados por =.

$$
\begin{aligned}
& f \;\; arg_1 \;\; \ldots \;\; arg_n \\
& \qquad\qquad | \;\; guarda_1 \;\; = \;\; exp_1 \\
& \qquad\qquad \ldots \\
& \qquad\qquad | \;\; guarda_m \;\; = \;\; exp_m
\end{aligned}
$$

- Cada guarda é uma expressão lógica.
- Os resultados devem ser todos do mesmo tipo.

Equações com guardas (cont.)

- Exemplo: valor absoluto

```
abs n | n >= 0 = n
      | n < 0  = -n
```

Nesta definição de `abs`, as guardas são `n >= 0` e `n < 0`, e as expressões associadas são `n` e `-n`, respectivamente.

Equações com guardas (cont.)

- Quando a função é aplicada, as guardas são verificadas em sequência. A primeira guarda verdadeira define o resultado.
- Assim no exemplo anterior o teste `n < 0` pode ser substituído pela constante **`True`**:

```
abs n | n >= 0 = n
      | True   = -n
```

Equações com guardas (cont.)

- A condição `True` pode também ser escrita como `otherwise`.
- Exemplo:

```
abs n | n >= 0    = n
      | otherwise = -n
```

- `otherwise` é uma condição que captura todas as outras situações que ainda não foram consideradas.
- `otherwise` é definida no prelúdio simplesmente como o valor verdadeiro:

```
otherwise :: Bool
otherwise  = True
```

Equações com guardas (cont.)

- Equações com guardas podem ser usadas para tornar definições que envolvem múltiplas condições mais fáceis de ler:
- Exemplo: determina o sinal de um número:

```
signum n | n < 0      = -1
         | n == 0     = 0
         | otherwise = 1
```

Equações com guardas (cont.)

- Exemplo: analisa o índice de massa corporal

```
analisaIMC imcs
  | imc <= 18.5 = "Voce esta abaixo do peso, seu emo!"
  | imc <= 25.0 = "Voce parece normal. Deve ser feio!"
  | imc <= 30.0 = "Voce esta gordo! Perca algum peso!"
  | otherwise   = "Voce esta uma baleia. Parabens!"
```

Equações com guardas (cont.)

- Se todas as guardas de uma equação forem falsas, a próxima equação é considerada. Se não houver uma próxima equação, ocorre um erro.
- Exemplo:

```
minhaFuncao x y | x > y = 1
                | x < y = -1
```

```
minhaFuncao 2 3 ⇒ -1
minhaFuncao 3 2 ⇒ 1
minhaFuncao 2 2 ⇒ ERRO
```

Equações com guardas (cont.)

- Um erro comum cometido por iniciantes é colocar um sinal de igual (=) depois do nome da função e parâmetros, antes da primeira guarda. Isso é um erro de sintaxe.

Exercícios

Em cada um dos exercícios a seguir:

- Defina a função solicitada de acordo com as instruções.
- Especifique o tipo mais geral desta função.
- Teste sua função no GHCi.

Exercício 1

Defina uma função chamada `media3` que recebe três valores e retorna a sua média aritmética.

Exercício 2

Defina uma função chamada `penultimo` que recebe uma lista e retorna o seu penúltimo elemento.

Exercício 3

Defina uma função chamada `maior2` que recebe dois valores e retorna o maior deles. Use expressões condicionais.

Exercícios (cont.)

Exercício 4

Defina uma função chamada `maior2'` que recebe dois valores e retorna o maior deles. Use equações com guardas.

Exercício 5

Defina uma função chamada `maior3` que recebe três valores e retorna o maior deles. Use expressões condicionais aninhadas.

Exercício 6

Defina uma função chamada `maior3'` que recebe três valores e retorna o maior deles. Use equações com guardas.

Exercício 7

Defina uma função chamada `maior3"` que recebe três valores e retorna o maior deles. Não use expressões condicionais e nem equações com guardas. Use a função `maior2` do exercício 3.

Exercícios (cont.)

Exercício 8

Defina uma função chamada `numRaizes` que recebe os três coeficientes de uma equação do segundo grau e retorna a quantidade de raízes reais distintas da equação. Assuma que a equação é não degenerada (isto é, o coeficiente do termo de grau 2 não é zero).

Exercício 9

Usando funções da biblioteca, defina a função `halve :: [a] -> ([a],[a])` que divide uma lista em duas metades. Por exemplo:

```
> halve [1,2,3,4,5,6]
([1,2,3],[4,5,6])

> halve [1,2,3,4,5]
([1,2],[3,4,5])
```

Exercícios (cont.)

Exercício 10

Determine o tipo da função definida a seguir e explique o que ela faz.

```
misterio m n p = not (m == n && n == p)
```

Casamento de padrão

- **Padrão** é uma frase que permite analisar um valor e associar variáveis aos dados que compõem o valor.
- **Casamento de padrão** é uma operação envolvendo um padrão e um valor que faz a correspondência (*casamento*) entre o padrão e o valor.
- O casamento de padrão pode suceder ou falhar, dependendo da forma do padrão e do valor envolvidos. Quando o casamento de padrão sucede as variáveis que ocorrem no padrão são associadas aos componentes correspondentes do valor.
- Existem várias formas de padrão. Na seqüência algumas delas serão apresentadas.

Expressão **case**

- A expressão **case** é uma forma de expressão que realiza casamento de padrão entre um valor e um ou mais padrões.
- A expressão **case** é da forma:

```
case exp of
  padrao_1 -> res_1
  ...
  padrao_n -> res_n
```

onde

- exp, res_1, ..., res_n são expressões
- $padrao_1$, ..., $padrao_n$ são padrões
- exp, $padrao_1$, ..., $padrao_n$ devem ser do mesmo tipo
- res_1, ..., res_n devem ser do mesmo tipo, que determina o tipo da expressão case.

Expressão **case** (cont.)

- Avaliação da expressão case:
  - é feita o casamento de padrão do valor de exp com os padrões, na seqüência em que foram escritos.
  - o primeiro padrão cujo casamento suceder é escolhido
  - o valor da expressão correspondente é o resultado da expressão case
  - o resultado é avaliado em um ambiente estendido com as associações de variáveis resultantes do casamento de padrão
  - se a expressão não casar com nenhum padrão, a expressão resulta em um erro
- Exemplos da expressão case serão apresentados junto com as formas de padrão, na seqüência.

Padrão constante

- O **padrão constante** é simplesmente uma constante.
- O casamento sucede se e somente se o padrão for idêntico ao valor.
- Nenhuma associação de variável é produzida.

Padrão constante (cont.)

- Exemplo: a expressão

```
case 3 - 2 + 1 of
  0 -> "zero"
  1 -> "um"
  2 -> "dois"
  3 -> "tres"
```

resulta em `"dois"`, pois o valor da expressão `3-2+1` é `2`, que casa com o terceiro padrão `2`, selecionando `"dois"` como resultado.

Padrão constante (cont.)

- Exemplo: a expressão

```
case 23 > 10 of
  True  -> "beleza!"
  False -> "oops!"
```

resulta em `"beleza!"`, pois o valor da expressão `23 > 10` é `True`, que casa com o primeiro padrão `True`, selecionando `"beleza!"` como resultado.

Padrão constante (cont.)

- Exemplo: a expressão

```
case toUpper (head "masculino") of
  'F' -> 10.2
  'M' -> 20.0
```

resulta em `"20.0"`, pois o valor da expressão `toUpper (head "masculino")` é `'M'`, que casa com o segundo padrão `'M'`, selecionando `20.0` como resultado.

Padrão constante (cont.)

- Exemplo: a expressão

```
case head "masculino" of
  'F' -> 10.2
  'M' -> 20.0
```

resulta em um erro em tempo de execução, pois o valor da expressão `head "masculino"` não casa com nenhum dos padrões.

Padrão constante (cont.)

- Exemplo: a expressão

```
case toUpper (head "masculino") of
  'F' -> "mulher"
  'M' -> 20.0
```

está incorreta, pois os resultados `"mulher"` e `20.0` não são do mesmo tipo.

Padrão constante (cont.)

- Exemplo: a expressão

```
case head "Masculino" == 'F' of
  True  -> "mulher"
  1     -> "homem"
```

está incorreta, pois os padrões `True` e `1` não são do mesmo tipo.

Padrão constante (cont.)

- Exemplo: a expressão

```
case head "Masculino" of
  True  -> "mulher"
  False -> "homem"
```

está incorreta, pois a expressão `head "Masculino"` e os padrões `True` e `False` não são do mesmo tipo.

Padrão constante (cont.)

- Exemplo: a expressão

```
case toUpper (head "masculino") of
   'F' -> "mulher"
 'M' -> 20.0
```

está incorreta, uma vez que não segue a regra de layout (os padrões não estão na mesma coluna).

Padrão variável

- O **padrão variável** é simplesmente um identificador de variável de valor (e como tal deve começar com letra minúscula).
- O casamento sucede sempre.
- A variável é associada ao valor.

Padrão variável (cont.)

- Exemplo: a expressão

```
case 3 - 2 + 1 of
  x -> 11 * x
```

resulta em `22`, pois o valor da expressão `3 - 2 + 1` é `2`, que casa com o primeiro padrão `x`, associando a variável `x` com o valor `2`, e selecionando `11 * x` como resultado

Padrão variável (cont.)

- Exemplo: a expressão

```
case mod 256 10 of
  7 -> 0
  n -> n * 1000
```

resulta em `6000`, pois o valor da expressão `mod 256 10` é `6`, que casa com o segundo padrão `n`, associando a variável `n` com o valor `6`, e selecionando `n * 1000` como resultado

Padrão variável (cont.)

- Exemplo: a expressão

```
case mod 257 10 of
  7 -> 0
  n -> n * 1000
```

resulta em `0`, pois `7` é o primeiro padrão que casa com o valor da expressão `mod 257 10`.

- Exemplo: a expressão

```
case mod 257 10 of
  n -> n * 1000
  7 -> 0
```

resulta em `7000`, pois `n` é o primeiro padrão que casa com o valor da expressão `mod 257 10`.

Padrão curinga

- O **padrão curinga** é escrito como um sublinhado (_).
- O casamento sucede sempre.
- Nenhuma associação de variável é produzida.
- _ é também chamado de variável anônima, pois casa com qualquer valor sem dar nome ao valor.

Padrão curinga (cont.)

- Exemplo: a expressão

```
case 46 - 2*20 of
  0 -> "zero"
  1 -> "um"
  2 -> "dois"
  3 -> "tres"
  4 -> "quatro"
  _ -> "maior que quatro"
```

resulta em `"maior que quatro"`, pois `_` é o primeiro padrão que casa com o valor da expressão `46 - 2*20`.

Padrão tupla

- Uma tupla de padrões também é um padrão

$$(\; padrao_1 , \; \ldots , \; padrao_n \;)$$

- O casamento sucede se e somente se cada um dos padrões casarem com os componentes correspondentes do valor.
- Observação: se as aridades do padrão tupla e do valor tupla forem diferentes, então ocorre um erro de tipo.

Padrão tupla (cont.)

- Exemplo: a expressão

```
case (3+2,3-2) of
  (0,0) -> 10
  (_,1) -> 20
  (x,2) -> x^2
  (x,y) -> x*y - 1
```

resulta em `20`, pois `(_,1)` é o primeiro padrão que casa com o valor da expressão `(3+2,3-2)`.

Listas

- **Lista** é uma seqüência de elementos de um mesmo tipo.
- Estruturalmente a forma de uma lista pode ser:
  - **lista vazia**
    - nenhum elemento na seqüência
    - constutor de dado: (constante)

```
[] :: [a]
```

  - **lista não vazia**
    - formada por dois componentes:
      cabeça: o primeiro elemento da lista
      cauda: lista de todos os demais elementos, a partir do segundo
    - construtor de dado: (dois argumentos)

```
infixr 5 :
(:) :: a -> [a] -> [a]
```

      O primeiro argumento é a cabeça da lista.
      O segundo argumento é a cauda da lista.

Listas (cont.)

- Exemplo:

```
[]
```

  - seqüência vazia (nenhum elemento)
  - não tem cabeça e nem cauda
  - tipo: `[a]`

Listas (cont.)

- Exemplo:

```
3 : []
```

  - seqüência: 3
  - cabeça: 3
  - cauda: []
  - tipo: `Num a => [a]`

Listas (cont.)

- Exemplo:

```
'a' : ('b' : [])
```

  - seqüência: `'a'`, `'b'`
  - cabeça: `'a'`
  - cauda: `'b' : []`
  - tipo: `[Char]`

Listas (cont.)

- Exemplo:

```
"bom" : ("dia" : ("brasil" : []))
```

  - seqüência: `"bom"`, `"dia"`, `"brasil"`
  - cabeça: `"bom"`
  - cauda: `"dia" : ("brasil" : [])`
  - tipo: `[[Char]]`

Listas (cont.)

- O construtor de lista não vazia `:` é um operador binário infixo com prioridade 5 e associatividade à direita.
- Isto significa que os parênteses no segundo argumento geralmente são desnecessários.
- Exemplo: o valor

```
100 : 200 : []
```

é idêntido a

```
100 : (200 : [])
```

Listas (cont.)

- Haskell oferece uma notação simplificada para listas:

  [exp_1, ..., exp_n]

  que é equivalente a

  exp_1 : ... : exp_n : []

- Exemplo: a lista

```
[10, 20, 30, 40, 50]
```

  é apenas uma abreviação para

```
10 : 20 : 30 : 40 : 50 : []
```

Padrões para listas

- Estruturalmente uma lista pode ser vazia ou não vazia.
- Para cada uma das formas há um padrão:
  - **padrão lista vazia**

```
[]
```

    - é um padrão constante
    - o casamento sucede se e somente se o valor for a lista vazia

  - **padrão lista não vazia**

pad_1 **:** pad_2

    - é formado por dois padrões pad_1 e pad_2
    - o casamento sucede se e somente se o valor for uma lista não vazia cuja cabeça casa com pad_1 e cuja cauda casa com pad_2

Padrões para listas (cont.)

- A forma

$$[\ padrao_1, \ \ldots, \ padrao_n\ ]$$

é uma abreviação sintática para

$$padrao_1 : \ldots : padrao_n : []$$

cujo casamento sucede somente se a lista tiver exatamente n elementos.

Padrões para listas (cont.)

- Exemplo: a expressão

```
case tail [10] of
  [] -> "vazia"
  _  -> "nao vazia"
```

resulta em `"vazia"`, pois o valor da expressão `tail [10]` casa com o padrão para lista vazia `[]`.

Padrões para listas (cont.)

- Exemplo: a expressão

```
case [10,20,30,40] of
  []   -> "lista vazia"
  x:xs -> "cabeca: " ++ show x ++ " cauda:
" ++ show xs
```

resulta em `"cabeca: 10 cauda: [20,30,40]"`, pois a lista `[10,20,30,40]` casa com o padrão para lista não vazia `x:xs`, associando `x` com `10` e `xs` com `[20,30,40]`.

Padrões para listas (cont.)

- Exemplo: a expressão

```
case [10..20] of
  x:y:z:_ -> x + y + z
  _       -> 0
```

resulta em `33`, pois a lista `[10..20]` casa com o padrão `x:y:z:_`, associando `x` com `10`, `y` com `11` e `z` com `12`.

Padrões para listas (cont.)

- Exemplo: a expressão

```
case [10,20] of
  x:y:z:_ -> x + y + z
  _       -> 0
```

resulta em `0`, pois a lista `[10,20]` não casa com o primeiro padrão `x:y:z:_`, mas casa com o segundo `_`.
Observe que o primeiro padrão casa somente com listas que tenham pelo menos três elementos.

Padrões para listas (cont.)

- Exemplo: a expressão

```
case [10,20,30] of
  [x1,_,x3] -> x1 + x3
  _         -> 0
```

resulta em `23`, pois a lista `[10,20,30]` casa com o primeiro padrão `[x1,_,x3]`.
Observe que este padrão casa somente com listas que tenham exatamente três elementos.

Expressão case com guardas

- Em uma expressão case cada padrão pode ser acompanhado de uma seqüência de cláusulas. Cada cláusula é introduzida por uma barra vertical (`|`) e consiste em uma condição (**guarda**) e uma expressão (resultado), separados por `->`.
- Para que o resultado de uma cláusula seja escolhido é necessário que o casamento de padrão suceda, e que a guarda correspondente seja verdadeira.

Expressão case com guardas (cont.)

- Exemplo:

```
case [100,20,3] of
  a:b:xs | a > b -> b:a:xs
         | a = b -> a:xs
  xs             -> xs
```

resulta em `[20,100,3]`, pois a lista `[100,20,3]` casa com o primeiro padrão `a:b:xs` e o primeiro elemento é maior do que o segundo.

Definindo funções com casamento de padrão

- A definição da função é formada por uma seqüência de equações.
- Os parâmetros usados em uma equação são padrões.
- Em uma aplicação da função o resultado será dado pela primeira equação cujos parâmetros casam com os respectivos argumentos, e cuja guarda (se houver) é verdadeira.
- Se em todas as equações os casamentos de padrão falharem ou todas as guardas forem falsas, ocorre um erro de execução.
- Muitas funções têm uma definição clara quando se usa casamento de padrão em seus argumentos.

Definindo funções com casamento de padrão (cont.)

- Exemplo:

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

A função `not` mapeia `False` a `True`, e `True` a `False`.

Definindo funções com casamento de padrão (cont.)

- Exemplo:

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

Definindo funções com casamento de padrão (cont.)

- Exemplo:

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

Definindo funções com casamento de padrão (cont.)

- Exemplo:

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

Definindo funções com casamento de padrão (cont.)

- Exemplo:

```
(&&)        :: Bool -> Bool -> Bool
b && b = b
_ && _ = False
```

está incorreto, pois não é possível usar uma variável mais de uma vez nos padrões (princípio da linearidade).

Definindo funções com casamento de padrão (cont.)

- Exemplos:

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

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

Definindo funções com casamento de padrão (cont.)

- Exemplos:

```
test1           :: [Char] -> Bool
test1 ['a',_,_] = True
test1 _         = False
```

```
test2           :: [Char] -> Bool
test2 ('a':_) = True
test2 _       = False
```

Definindo funções com casamento de padrão (cont.)

- Exemplos:

```haskell
null       :: [a] -> Bool
null []     = True
null (_:_) = False

head       :: [a] -> a
head (x:_) = x

tail       :: [a] -> a
tail (_:xs) = xs
```

Exercícios

Exercício 11

Considere uma função `safetail` que se comporta da mesma maneira que `tail`, exceto que `safetail` mapeia a lista vazia para a lista vazia, enquanto que `tail` dá um erro neste caso. Definir `safetail` usando:

a) uma expressão condicional

b) equações com guardas

c) uma expressão case

d) equações com casamento de padrão

Dica: a função `null :: [a] -> Bool` do pelúdio pode ser usado para testar se uma lista é vazia.

Exercícios (cont.)

Exercício 12

Dê três possíveis definições para o operador lógico ou (`||`), utilizando casamento de padrão.

Exercícios (cont.)

Exercício 13

Redefina a seguinte versão do operador lógico e (&&) usando expressões condicionais ao invés de casamento de padrão:

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

Exercício 14

Redefina a seguinte versão do operador lógico e (&&) usando expressões condicionais ao invés de casamento de padrão:

```
True  && b = b
False && _ = False
```

Comente sobre o diferente número de expressões condicionais necessárias em comparação com o exercício 13.

Exercícios (cont.)

Exercício 15

Defina uma função que recebe dois pontos no espaço e retorna a distância entre eles. Considere que um ponto no espaço é representado por uma tripla de números que são as coordenadas do ponto. Use casamento de padrão.

Exercício 16

Analise a seguinte definição e apresente uma definição alternativa mais simples desta função.

```
opp :: (Int,(Int,Int)) -> Int
opp z = if fst z == 1
        then fst (snd z) + snd (snd z)
        else if fst z == 2
             then fst (snd z) - snd (snd z)
             else 0
```

Definições locais com **where**

- A palavra reservada **where** é utilizada para introduzir definições locais cujo contexto (ou **escopo**) é a expressão no lado direito e as guardas (quando houver) de uma equação.
- **Exemplo:**

```
f       :: Fractional a => (a,a) -> a
f (x,y) = (a + 1) * (a + 2) where a = (x+y)/2
```

```
f (2,3) ⇒ 15.75
f (5,1) ⇒ 20.0
```

Definições locais com **where** (cont.)

- Quando há duas ou mais definições locais, elas podem ser escritas em diferentes estilos.
- **Exemplos:**

```
f (x,y) = (a+1)*(b+2)
  where { a = (x+y)/2; b = (x+y)/3 }
```

```
f (x,y) = (a+1)*(b+2)
  where a = (x+y)/2; b = (x+y)/3
```

```
f (x,y) = (a+1)*(b+2)
  where a = (x+y)/2
        b = (x+y)/3
```

Neste último exemplo foi usada a regra de *layout*, que dispensa os símbolos `;`, `{` e `}` mas exige que cada definição local esteja alinhada em uma mesma coluna.

Definições locais com **where** (cont.)

- **Exemplo:**

```
square x = x * x

g x y | x <= 10 = x + a
      | x >  10 = x - a
  where
    a = square (y+1)
```

O escopo de a inclui os dois possíveis resultados, determinados pelas guardas.

Definições locais com **where** (cont.)

- As definições locais podem ser de funções e de variáveis, fazendo uso de padrões.
- **Exemplo:**

```
h y = 3 + f y + f a + f b
  where
    c = 10
    (a,b) = (3*c,f 2)
    f x = x + 7*c
```

```
h 5 ⇒ 320
```

O escopo de a inclui os dois possíveis resultados, determinados pelas guardas.

Definições locais com **where** (cont.)

- Exemplo: análise do índice de massa corporal

```
analisaIMC peso altura
  | imc <= 18.5 = "Voce esta abaixo do peso, seu emo!"
  | imc <= 25.0 = "Voce parece normal. Deve ser feio!"
  | imc <= 30.0 = "Voce esta gordo! Perca algum peso!"
  | otherwise   = "Voce esta uma baleia. Parabens!"
  where
    imc = peso / height ^2
```

Ou ainda:

```
analisaIMC peso altura
  | imc <= magro  = "Voce esta abaixo do peso, seu emo!"
  | imc <= normal = "Voce parece normal. Deve ser feio!"
  | imc <= gordo  = "Voce esta gordo! Perca algum peso!"
  | otherwise     = "Voce esta uma baleia. Parabens!"
  where
    imc    = peso / height ^2
    magro  = 18.5
    normal = 25.0
    gordo  = 30.0
```

Definições locais com **where** (cont.)

- Definições locais com **where** não são compartilhadas entre corpos de funções de diferentes padrões nas equações.
- **Exemplo:**

```
saudacao           :: String -> String
saudacao "Joana"    = saudacaoLegal ++ " Joana!"
saudacao "Ferando"  = saudacaoLegal ++ " Fernando!"
saudacao nome       = saudacaoInfeliz ++ " " ++ nome
  where
    saudacaoLegal   = "Ola! Que bom encontrar voce, "
    saudacaoInfeliz = "Oh! Pfft. E voce, "
```

Esta definição de função está incorreta. Para corrigi-la, transforme as definições locais de `saudacaoLegal` e `saudacaoInfeliz` em definições globais.

Expressoes **let**

- Uma **expressão let** é formada por uma lista de declarações mutuamente recursivas, e por uma expressão:

```
let definições in expressão
```

- O escopo das declarações é a expressão e o lado direito das declarações. Portanto os nomes introduzidos nas declarações só podem ser usados nas próprias declarações e na expressão.
- O tipo da expressão let é o tipo da expressão que aparece no seu corpo.
- O valor da expressão let é o valor da expressão que aparece no seu corpo, calculado em um contexto que inclui as variáveis introduzidas nas declarações.
- A expressão let se estende à direita tanto quanto possível.

Expressoes **let** (cont.)

- **Exemplos:**

```
let a = 5 in (a - 1)*a + 1
⇒ 21
```

Expressoes **let** (cont.)

```
let { y = 1 + 2; z = 4 + 6 } in y + z
⇒ 21
```

Expressoes **let** (cont.)

```
let r = 3; s = 6 in r^2 + s^2
⇒ 45
```

Expressoes **let** (cont.)

```
let a = 1
    b = -5
    c = 6
    r = sqrt (b^2 - 4*a*c)
    k = a/2
in ((-b + r)/k,(-b - r)/k)
⇒ (12.0,8.0)
```

Expressoes **let** (cont.)

```
f a b = let y   = a*b
            g x = (x+y)/y
        in g (2*a) + g (3*b)
```

```
f 3 4
⇒ 3.5
```

Expressoes **let** (cont.)

```
4 * let x = 5-2 in x * x
⇒ 36
```

Expressoes **let** (cont.)

```
(let x = 5-2 in x * x) ^ 2
⇒ 81
```

Expressoes **let** (cont.)

```
[ let square x = x*x in (square 5,square 3, square 2) ]
⇒ [(25,9,4)]
```

Expressoes **let** (cont.)

```
( let a = 100; b = 200; c = 300 in a*b*c
, let foo = "Hey "; bar = "there!" in foo ++ bar
)
⇒ (6000000,"Hey there!")
```

Expressoes **let** (cont.)

```
let (a,b,c) = (1,2,3) in a + b + c
⇒ 6
```

Expressoes **let** (cont.)

```
let a = 1
in let b = 2
   in a + b
⇒ 3
```

Expressoes **let** (cont.)

```
let x = 7 in (let x = "foo" in x, x)
⇒ ("foo",7)
```

Expressoes **let** (cont.)

```
let c = 10
    (a,b) = (3*c, f 2)
    f x = x + 7*c
in f a + f b
⇒ 242
```

Expressoes **let** (cont.)

- **Exemplo:**
  Cálculo da área da superfície de um cilindro:

```
areaSuperfCil    :: Double -> Double -> Double
areaSuperfCil r h = let areaLado = 2 * pi * r * h
                        areaBase = pi * r^2
                    in areaLado + 2*areaBase
```

Diferenças entre `let` e `where`

- Com **where** as definições são colocadas no final, e com **let** elas são colocadas no início.
- **let** é uma expressão e pode ser usada em qualquer lugar onde se espera uma expressão.
- Já **where** não é uma expressão, podendo ser usada apenas para fazer definições locais em uma definição de função.

Exercício 17

Defina uma função para calcular as raízes reais do polinômio

$$ax^2 + bx + c$$

Faça duas versões, usando:

- expressão **let** para calcular o discrimante
- definição local com **where** para calcular o discriminante.

Teste suas funções no GHCi.
Dica: Use a função `error :: String ->` a do prelúdio, que exibe uma mensagem de erro e termina o programa, para exibir uma mensagem quando não houver raízes reais.

Valores de primeira classe

- Dizemos que um tipo de **primeira classe** é um tipo para o qual não há restrições sobre como os seus valores podem ser usados.

Valores de primeira classe (cont.)

- Valores de vários tipos podem ser escritos diretamente, sem a necessidade de dar um nome a eles:

valor	tipo	descrição
`456`	`Num a => a`	o número 456
`2.45`	`Floating a => a`	o número em ponto flutuante 2.45
`True`	`Bool`	o valor lógico *verdadeiro*
`'G'`	`Char`	o caracter *G*
`"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*

Valores de primeira classe (cont.)

- 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. Hut-
ton" 2007
```

Valores de primeira classe (cont.)

- 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]
```

Valores de primeira classe (cont.)

- 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]
```

Valores de primeira classe (cont.)

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

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

Valores de primeira classe (cont.)

- Em Haskell funções são valores de primeira de classe:
  - funções podem ser escritas sem a necessidade de receberem um nome

    ```
    \x -> 3*x
    ```

    função anônima que mapeia um número ao seu triplo.
  - funções podem ser nomeadas

    ```
    f = \x -> 3*x
    g x = 3 * x
    ```

    as funções `f` e `g` são idênticas.
  - funções podem ser argumentos de outras funções

    ```
    map f [1,2,3] ⇒ [2,4,6]
    ```

    a função `f` é aplicada a cada elemento da lista, retornando a lista dos resultados.
  - funções podem ser resultados de outras funções

    ```
    abs . sin
    ```

    composição das funções `abs` e `sin`.

Valores de primeira classe (cont.)

- funções podem ser componentes de outros valores

```
[abs,sin,cos]
```

lista das funções `abs`, `sin` e `cos`.

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:

```
\padrão_1 ... padrão_n -> 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.

Expressões lambda (cont.)

- **Exemplo**:
  função anônima que calcula o dobro de um número:

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

  O tipo desta expressão lambda é `Num a => a -> a`

Expressões lambda (cont.)

- **Exemplo**:
  função anônima que mapeia um número x a $2x + 1$:

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

  cujo tipo é `Num a => a -> a`

Expressões lambda (cont.)

- **Exemplo**:
  função anônima que calcula o fatorial de um número:

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

  cujo tipo é `(Enum a, Num a) => a -> a`

Expressões lambda (cont.)

- **Exemplo**:
  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`

Expressões lambda (cont.)

- **Exemplo**:
  definições de função usando expressão lambda:

```
f       = \x -> 2*x + 1
cauda   = \(_:xs) -> xs
fatorial = \n -> product [1..n]
```

  é o mesmo que

```
f x           = 2*x + 1
cauda (_:xs) = xs
fatorial n    = product [1..n]
```

Expressões lambda (cont.)

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

Expressões lambda (cont.)

- **Exemplos**:
  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
```

```
(\(x:xs) -> x) "Bom dia"
⇒ 'B'
```

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

Uso 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.

Uso de expressões lambda (cont.)

- **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`.

Uso de expressões lambda (cont.)

```
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
```

Uso de expressões lambda (cont.)

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

Uso de expressões lambda (cont.)

- **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
```

Uso de expressões lambda (cont.)

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
```

Uso de expressões lambda (cont.)

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

Uso de expressões lambda (cont.)

- **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]
```

Operadores

- Um **operador 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.

Operadores (cont.)

- 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  -}
```

Operadores (cont.)

- 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
```

Operadores (cont.)

- 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]
```

Seções de operadores

- Como os operadores infixos são de fato funções, faz sentido ser capaz de aplicá-las parcialmente também.
- Em Haskell a aplicação parcial de um operador infixo é chamada de **seção** e deve ser escrita entre parênteses.

Seções de operadores (cont.)

- **Exemplo:**

```
(1+)
```

é a função que incrementa (soma 1 a) o seu argumento. É o mesmo que

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

Seções de operadores (cont.)

- **Exemplo:**

```
(*2)
```

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

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

Seções de operadores (cont.)

- **Exemplo:**

```
(100>)
```

é a função que verifica se 100 é maior que o seu argumento. É o mesmo que

```
\x -> 100 > x
```

Seções de operadores (cont.)

- **Exemplo:**

```
(<0)
```

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

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

Seções de operadores (cont.)

- **Exemplos** de aplicação:

```
(1+) 2 ⇒ 3
(+2) 1 ⇒ 3
```

```
(100>) 200 ⇒ False
(>200) 100 ⇒ False
```

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

Seções de operadores (cont.)

- Em geral, se $\oplus$ é um operador binário, 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
```

Seções de operadores (cont.)

- **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)
    ```

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

Por que seções são úteis? (cont.)

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

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

Por que seções são úteis? (cont.)

- 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:

```haskell
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.

Exercícios

Exercício 18

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")`

Exercícios (cont.)

Exercício 19

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.

Fim