

Programação Funcional

Linguagem Haskell: continuação

Profa. Andrea Schwertner Charão

ELC117 – Paradigmas de Programação
<http://www.inf.ufsm.br/~andrea/elc117>



Primeiro semestre 2011

- 1 Geração de listas com '..'
- 2 Avaliação preguiçosa (lazy evaluation)
- 3 Funções que geram listas infinitas
- 4 Listas por compreensão (list comprehension)
- 5 Composição de funções

Sumário

- 1 Geração de listas com '..'
- 2 Avaliação preguiçosa (lazy evaluation)
- 3 Funções que geram listas infinitas
- 4 Listas por compreensão (list comprehension)
- 5 Composição de funções

Geração de listas com '..'

- Notação '..': usada para gerar listas que representam domínios ordenados (números, caracteres, etc.)
- Formas:
 - Notação **[n..m]**: lista de n a m, com passo 1 (m é o limite superior)
 - Notação **[n,m..p]**: lista de n a p, com passo m-n (passo pode ser negativo)
 - Notação **[n..]**: lista infinita, com passo 1
 - Notação **[n,m..]**: lista infinita, com passo m-n

Notação [n..m]

Geração de listas

Notação **[n..m]**: Lista de n a m, com passo 1 (m é o limite superior)

Exemplos no interpretador:

```
> [1..5]  
[1,2,3,4,5]
```

```
> [5..1]  
[]
```

```
> [1.66..5] — 5 eh o limite  
[1.66,2.66,3.66,4.66]
```

```
> ['a'..'z']  
"abcdefghijklmnopqrstuvwxyz"
```

Notação [n,m..p]

Geração de listas

Notação **[n,m..p]**: Lista de n a p, com passo m-n (passo pode ser negativo)

Exemplos no interpretador:

```
> [2,4..10] — passo 4-2 = 2  
[2,4,6,8,10]
```

```
> [3,6..16] — passo 6-3 = 3  
[3,6,9,12,15]
```

```
> [10,9..1] — passo 9-10 = -1  
[10,9,8,7,6,5,4,3,2,1]
```

```
> [5, 4.5 .. 1] — passo 4.5-5 = -0.5  
[5.0,4.5,4.0,3.5,3.0,2.5,2.0,1.5,1.0]
```

Notação [n..]

Geração de listas

Notação [n..]: Lista infinita, com passo 1

Exemplos no interpretador:

```
> [10..]  
[10,11,12 ^C, Interrupted .
```

```
> [0.33..]  
[0.33,1.33,2.33 ^C, Interrupted .
```

A geração no interpretador pode ser interrompida com CONTROL-C.

Notação $[n,m..]$

Geração de listas

Notação $[n,m..]$: Lista infinita, com passo $m-n$

Exemplos no interpretador:

```
> [0.45,0.75..  
[0.45,0.75,1.05,1.35 ^C, Interrupted.
```

A geração no interpretador pode ser interrompida com CONTROL-C.

Sumário

- 1 Geração de listas com '..'
- 2 Avaliação preguiçosa (lazy evaluation)
- 3 Funções que geram listas infinitas
- 4 Listas por compreensão (list comprehension)
- 5 Composição de funções

Avaliação preguiçosa (lazy evaluation)

- Listas infinitas são possíveis porque Haskell trabalha com a noção de **avaliação preguiçosa** (lazy evaluation)
- Avaliação preguiçosa: retarda a avaliação de uma expressão até que seu valor seja realmente necessário
- Também conhecida como: call-by-need, non-strict evaluation

Exemplo:

```
> take 5 [11..]  
[11,12,13,14,15]
```

No exemplo acima:

- a função take retorna os 5 primeiros elementos da lista
- lazy evaluation em ação: a lista infinita não precisa ser totalmente gerada, basta gerar os 5 primeiros elementos

Avaliação preguiçosa (lazy evaluation)

Alguns benefícios da avaliação preguiçosa em relação à avaliação tradicional (strict):

- desempenho aumenta, pois evitam-se cálculos desnecessários
- uso de memória diminui, pois valores são criados quando necessário
- estruturas de dados potencialmente infinitas

- 1 Geração de listas com '..'
- 2 Avaliação preguiçosa (lazy evaluation)
- 3 Funções que geram listas infinitas**
- 4 Listas por compreensão (list comprehension)
- 5 Composição de funções

Funções que geram listas infinitas

Alguns exemplos de funções:

- **repeat**
 - **repeat** :: $a \rightarrow [a]$
 - Recebe um argumento e gera uma lista repetindo o argumento infinitas vezes
- **cycle**
 - **cycle** :: $[a] \rightarrow [a]$
 - Recebe uma lista como argumento e gera uma lista repetindo o argumento infinitas vezes
- **iterate**
 - **iterate** :: $(a \rightarrow a) \rightarrow a \rightarrow [a]$
 - Recebe uma função e um argumento e retorna uma lista infinita, com sucessivas aplicações da função sobre o argumento
 - É uma função de **alta ordem**

Função repeat

Recebe um argumento e gera uma lista repetindo o argumento infinitas vezes

Exemplos:

```
> repeat 5  
[5,5,5,5 ^C, Interrupted.]
```

```
> take 10 (repeat 'a')  
"aaaaaaaaaa"
```

```
> take 10 (repeat "A")  
["A" ,"A" ,"A" ,"A" ,"A" ,"A" ,"A" ,"A" ,"A" ,"A" ]
```

```
> take 5 (repeat (0,0))  
[(0,0),(0,0),(0,0),(0,0),(0,0)]
```

Função cycle

Recebe uma lista como argumento e gera uma lista repetindo o argumento infinitas vezes

Exemplos:

```
> cycle [1..3]  
[1,2,3,1,2,3,1,2,3,1 ^C, Interrupted.]
```

```
> take 6 (cycle [1..3])  
[1,2,3,1,2,3]
```

```
> take 10 (cycle "ab")  
"ababababab"
```

```
> take 10 (repeat "ab") — repeat versus cycle  
["ab", "ab", "ab", "ab", "ab", "ab", "ab", "ab", "ab", "ab"]
```

Função iterate

Recebe uma função e um argumento e retorna uma lista infinita, com sucessivas aplicações da função sobre o argumento

Função iterate

```
iterate :: (a -> a) -> a -> [a]
```

Exemplos:

```
> iterate (2*) 1  
[1,2,4,8,16,32,54 ^C, Interrupted .
```

```
> take 3 (iterate (+0.2) 1.0)  
[1.0,1.2,1.4]
```

Sumário

- 1 Geração de listas com '..'
- 2 Avaliação preguiçosa (lazy evaluation)
- 3 Funções que geram listas infinitas
- 4 Listas por compreensão (list comprehension)**
- 5 Composição de funções

Listas por compreensão (list comprehension)

- Recurso de Haskell inspirado na notação de conjuntos em matemática
- Conjuntos: "por extensão" X "por compreensão"
 - Por extensão: enumera-se os elementos
 - Por compreensão: define-se uma regra de geração dos elementos
- Exemplo em matemática:
 - $S = \{2.x | x \in \mathbb{N}, x \leq 10\}$
 - S é o conjunto que contém os naturais menores ou iguais a 10, multiplicados por 2
 - Ou ainda: S é o conjunto dos 10 primeiros naturais pares

Exemplo em Haskell

```
> [x*2 | x <- [1..10]]  
[2,4,6,8,10,12,14,16,18,20]
```

Listas por compreensão (list comprehension)

Forma geral

```
[ exprsaida | gerador , ... , gerador ]
```

Exemplo

```
> [x*2 | x <- [1..10]]  
[2,4,6,8,10,12,14,16,18,20]
```

Onde:

- **gerador** tem a forma "*padrao* < -*expressao*" e representa uma lista de origem
- **exprsaida** é uma expressão avaliada sobre a(s) lista(s) gerada(s) do lado direito
- o símbolo "< -" pode ser lido como "pertence"

Listas por compreensão (list comprehension)

Exemplos no interpretador

```
> [x*2 | x <- [1..10]]  
[2,4,6,8,10,12,14,16,18,20]
```

```
> [x | x <- [1..10], even x]  
[2,4,6,8,10]
```

- A função `even` retorna `True` se o argumento for par, ou `False` se o argumento for ímpar
- Em "`even x`", `x` representa cada elemento da lista
- A vírgula no gerador representa um "E" lógico

Listas por compreensão (list comprehension)

Com esta notação, pode-se facilmente gerar listas de tuplas

Exemplos no interpretador

```
> [(x,y) | x <- [1,2], y <- [1,2]]  
[(1,1),(1,2),(2,1),(2,2)]
```

```
> [(x,y) | x <- [1..2], y <- [3..6]]  
[(1,3),(1,4),(1,5),(1,6),(2,3),(2,4),(2,5),(2,6)]
```

Listas por compreensão (list comprehension)

Filtro para strings (.hs)

```
removeChar :: Char -> [Char] -> [Char]
removeChar c str = [x | x <- str, x /= c]
```

Uso

```
> removeChar 'a' "abababa"
"bbb"
```

Listas por compreensão (list comprehension)

Filtro para imagens (.hs)

```
filterImg :: [Int] -> [Int]
filterImg bitmap =
  [if pixel < 10 then 0 else pixel | pixel <- bitmap]
```

Uso

```
> filterImg [1,3,4,20,40,40,40,3,2]
[0,0,0,20,40,40,40,0,0]
```

Listas por compreensão (list comprehension)

Processa string (.hs)

```
removeNonUppercase :: [Char] -> [Char]
removeNonUppercase str =
    [ c | c <- str, c `elem` ['A'..'Z']]
```

Uso

```
> removeNonUppercase "afdAasfdA"
"AA"
```

Sumário

- 1 Geração de listas com '..'
- 2 Avaliação preguiçosa (lazy evaluation)
- 3 Funções que geram listas infinitas
- 4 Listas por compreensão (list comprehension)
- 5 Composição de funções

Composição de funções

- Recurso de Haskell inspirado na composição de funções matemáticas
- Exemplo em matemática: $f.g(x) = f(g(x))$
- Em Haskell, usa-se a função expressa por um ponto:
 $(.) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c$
- **Importante:** A primeira função recebe como argumento um valor do mesmo tipo de retorno da segunda função

Exemplos no interpretador

```
> (sqrt . last) [1,2,3]
```

```
1.7320508075688772
```

```
> (negate . (*3)) 5
```

```
-15
```

```
> map (negate . round) [5,-3,-6.8,-3.2,-19.24]
```

```
[-5,3,7,3,19]
```

Elefantes incomodam em Haskell

elefantes.hs

```
incomoda :: Int -> String
incomoda n
  | n == 1      = "1 elefante incomoda muita gente"
  | odd n       = show n ++ " elefantes incomodam muita gente"
  | otherwise   = show n ++ " elefantes " ++
                  concat (replicate n "incomodam ") ++ "muito mais"

main = sequence_ (map (putStrLn . incomoda) [1..6])
-- Outras opcoes:
-- main = sequence (map (putStrLn . incomoda) [1..6])
-- main = mapM_ (putStrLn . incomoda) [1..6]
-- main = mapM (putStrLn . incomoda) [1..6]
```

Elefantes incomodam em Haskell

Observações sobre o programa:

- A função `map` retorna uma lista com os comandos de saída na tela (**`putStrLn :: String -> IO ()`**)
- A função `sequence` executa os comandos da lista em sequência (parte imperativa do programa)
- A função **`mapM`** funcao lista é equivalente a **`sequence (map funcao lista)`**) para o caso em que a função representa uma ação (imperativa)
- As funções `mapM` e `sequence` mostram `[(),(),(),(),(),()]` no final da saída na tela, representando que a operação de saída transcorreu sem erro
- As funções com underscore ("`_`") no final são versões alternativas que desprezam o resultado da operação
- Mais sobre isso em:
<http://learnyouahaskell.com/input-and-output>

- A. R. Du Bois. **Programação Funcional com a Linguagem Haskell.**
- M. Lipovača. **Learn You a Haskell for Great Good!.** Disponível em: <http://learnyouahaskell.com/chapters>
- Haskell.org. **List comprehension - HaskellWiki.** Disponível em: http://www.haskell.org/haskellwiki/List_comprehension