

# Listas

## Linguagem Haskell

Maria Adriana Vidigal de Lima

*Faculdade de Computação - UFU*

*Setembro - 2009*

1

## Listas em Haskell

- Noções sobre Listas
- Construção de Listas por Compreensão
- Funções Sobre Listas
- Processamento de Listas

# Fundamentos

- Uma lista é uma estrutura de dados que representa uma coleção de objetos homogêneos em sequência.
- Para alcançar qualquer elemento, todos os anteriores a ele devem ser recuperados.
- Em programação, uma lista vazia (representada por `[]` em Haskell) é a estrutura-base da existência de uma lista.

# Listas em Haskell

Uma lista é composta sempre de dois segmentos: cabeça (*head*) e corpo (*tail*). A cabeça da lista é sempre o primeiro elemento.

```
> ['a','b','c','d']
```

```
"abcd"
```

```
> 'a': ['b','c','d']
```

```
"abcd"
```

`['a', 'b', 'c', 'd']`  $\rightarrow$  `'a': ['b', 'c', 'd']`

*primeiro  
elemento*

*corpo da  
lista*

# Operador (:)

O símbolo (:) é o operador de construção de listas. Toda lista é construída através deste operador. Podemos verificar que este operador é polimórfico, usando o comando `type`:

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

A letra `a` indica uma família de tipos (numéricos, caracteres, tuplas, ...). O operador deve ser aplicado à argumentos de um mesmo tipo.

```
Hugs> 'a' : ['b', 'c', 'd']  
"abcd"  
Hugs> 2 : [4, 6, 8]  
[2, 4, 6, 8]
```

# Listas em Haskell

$[ 'a', 'b', 'c', 'd' ] \rightarrow 'a': [ 'b', 'c', 'd' ]$

|      {  
*primeiro*      *corpo da*  
*elemento*      *lista*

```
Hugs> 'a':['b','c','d']
```

```
"abcd"
```

```
Hugs> 1:[2,3]
```

```
[1,2,3]
```

```
Hugs> ['a','c','f'] == 'a':['c','f']
```

```
True
```

```
Hugs> [1,2,3] == 1:2:3:[]
```

```
True
```

```
Hugs> 1:[2,3] == 1:2:[3]
```

```
True
```

```
Hugs> "papel" == 'p':['a','p','e','l']
```

```
True
```

# Listas e Tipos

Uma lista é uma coleção de elementos de um dado tipo. Para todo tipo  $t$  existe uma lista  $[t]$  para seus elementos.

```
> [1,2,3] :: [Int]
[1,2,3]
```

```
> [True,True,False] :: [Bool]
[True,True,False]
```

```
> [(1,2),(4,5),(0,8)] :: [(Int,Int)]
[(1,2),(4,5),(0,8)]
```

```
> [[2,3,4],[5],[],[3,3]] :: [[Int]]
[[2,3,4],[5],[],[3,3]]
```

# Listas bem formadas

Alguns exemplos de listas bem formadas são:

```
Letras :: [Char]
```

```
Letras = ['a','b','c','z']
```

```
Inteiros :: [Int]
```

```
Inteiros = [5,23,4,66]
```

```
Booleanos :: [Bool]
```

```
Booleanos = [True, True, False]
```

```
Tuplas :: [(Int,Char)]
```

```
Tuplas = [(2,'v'),(3,'g'),(5,'d')]
```

```
Palavras :: [[Char]]
```

```
Palavras = ["ana", ['a','b','a']]
```



# Escrevendo Listas

Pode-se definir uma lista indicando os limites inferior e superior de um conjunto conhecido, onde existe uma relação de ordem entre os elementos, no seguinte formato:

*[ <limite-inferior> .. <limite-superior> ]*

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

```
> ['m'..'n']  
"mn"
```

```
> [1,3..6]  
[1,3,5]
```

```
> ['a','d'..'p']  
"adgjmp"
```

```
> [3.1..7]  
[3.1,4.1,5.1,6.1,7.1]
```

```
> [-1,-3..(-10)]  
[-1,-3,-5,-7,-9]
```

# Escrevendo Listas

Podemos definir qualquer progressão aritmética em uma lista utilizando a seguinte notação:

$[ \langle 1o. \text{ termo} \rangle, \langle 2o. \text{ termo} \rangle .. \langle \text{limite-superior} \rangle ]$

> [7,6..3]

[7,6,5,4,3]

> [6,5..0]

[6,5,4,3,2,1,0]

> [-5,2..16]

[-5,2,9,16]

> [5,6..5]

[5]

> [1,1.1 .. 2]

[1.0,1.1,1.2,1.3,1.4,1.5,1.6,1.7,1.8,1.9,2.0]

# Listas por Compreensão

A descrição de uma lista pode ser feita em termos dos elementos de uma outra lista. Por exemplo, temos a lista  $L1 = [2, 4, 7]$ . Uma lista definida por compreensão pode ser escrita:

```
> [ 2 * n | n <- L1 ]  
[4,8,14]
```

A lista resultante contém todos os elementos da lista  $L1$ , multiplicados por 2. Assim, podemos ler:

*Obtenha todos os  $2*n$  dos elementos  $n$  contidos em  $L1 = [2, 4, 7]$ .*

# Listas por Compreensão

## Definição

A definição de listas por compreensão é feita por um construtor de listas que utiliza conceitos e notações da teoria dos conjuntos. Assim, para um conjunto  $A$  temos:

$$A = \{ E(x) \mid x \in C \wedge P_1(x) \wedge \dots \wedge P_n(x) \}$$

sendo  $E(x)$  uma expressão em  $x$ ,  $C$  um conjunto inicial para os valores de  $x$  e os vários  $P_i(x)$  são proposições em  $x$ .

O conjunto  $A$  é escrito em Haskell da seguinte forma:

$$A = [ E(x) \mid x \leftarrow \text{lista}, P_1(x), \dots, P_n(x) ]$$

# Listas por Compreensão

O conjunto dos quadrados dos números inteiros é definido pela expressão:

$$A = \{ x^2 \mid X \in \mathbb{N} \}$$

Em Haskell, podemos escrever  $A$  para listas finitas ou infinitas da seguinte forma:

```
listaQuad = [ x^2 | x <- [1..30] ]
```

```
listaQuadInf = [ x^2 | x <- [1..] ]
```

# Listas por Compreensão

```
listaQuad = [ x^2 | x <- [1..30] ]
```

```
> listaQuad
```

```
[1,4,9,16,25,36,49,64,81,100,121,144,169,196,225,256,289,  
324,361,400,441,484,529,576,625,676,729,784,841,900]
```

```
listaQuadInf = [ x^2 | x <- [1..] ]
```

```
> listaQuadInf
```

```
[1,4,9,16,25,36,49,64,81,100,121,144,169,196,225,256,289,  
324,361,400,441,484,529,576,625,676,729,784,841,900,961,  
1024,1089,1156,1225,1296,1369,1444,1521,1600 {Interrupted!}]
```

```
> elem 4 listaQuadInf
```

```
True
```

A função `elem` verifica se um elemento pertence à uma lista. Em caso positivo, retorna `True`, senão `False`.

# Gerador e Expressões Booleanas

- Na definição de lista por compreensão, o símbolo `<-` é chamado de **gerador** da lista, pois permite obter os dados através dos quais a nova lista será construída.
- Os geradores podem ser combinados com um ou mais testes, que são expressões booleanas.

```
listaQuadPares = [x^2 | x <- [1..20], even x]
```

```
Hugs> listaQuadPares
```

```
[4,16,36,64,100,144,196,256,324,400]
```

# Gerador e Expressões Booleanas

```
listaQuadParesSup = [x^2 | x <- [1..20], even x, x > 6]
```

```
Hugs> listaQuadParesSup  
[64,100,144,196,256,324,400]
```

```
listaPares = [even x | x <- [1..20] ]
```

```
Hugs> listaPares  
[False,True,False,True,False,True,False,True,  
False,True]
```



# Listas com mais de um Gerador

Adicionalmente, é possível que mais de um gerador seja utilizado na definição de uma lista por compreensão:

```
Hugs> [ x*y | x <- [1,2,3], y <- [3,7,9]]  
[3,7,9,6,14,18,9,21,27]
```

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

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

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

# Utilização de listas por compreensão

Podemos utilizar a técnica de construção de listas por compreensão na codificação de programas.

```
{- A função dobraPos duplica todos os números positivos  
maiores que zero de uma lista -}  
dobraPos :: [Int] -> [Int]  
dobraPos xs = [2*x | x<-xs, x>0]
```

```
>dobraPos [3,4,-1,0,5]  
[6,8,10]
```

# Utilização de listas por compreensão

A função `dobraPos` pode ser também escrita através do construtor de listas `(:)` de forma recursiva:

```
-- Função definida através de lista por compreensão
dobraPos :: [Int] -> [Int]
dobraPos xs = [2*x | x<-xs, x>0]

-- Função recursiva
dobraPosR :: [Int] -> [Int]
dobraPosR [] = []
dobraPosR (a:as) = if a > 0 then 2*a: dobraPosR as
                    else  dobraPosR as
```

# Utilização de listas por compreensão

A função `fatores` retorna cada um dos fatores de um número, e pode ser usada para verificar se um número é primo.

```
fatores n = [i | i<-[1..n], n `mod` i == 0]
```

```
> fatores 90  
[1,2,3,5,6,9,10,15,18,30,45,90]
```

```
primo n = if (fatores n) == [1,n] then True  
          else False
```

```
> primo 90  
False
```

# Funções Pré-definidas

A tabela abaixo relaciona algumas funções pré-definidas em Haskell para a manipulação de listas:

Função	Descrição	Exemplo
(++)	Concatena duas listas	<pre>&gt; [1,2,3]++[4,5,6] [1,2,3,4,5,6]</pre>
concat	Recebe uma lista de listas e as concatena	<pre>&gt; concat [[1,2],[3,4]] [1,2,3,4]</pre>
head	Retorna o primeiro elemento da lista	<pre>&gt; head "abc" 'a'</pre>
tail	Retorna o corpo da lista	<pre>&gt; tail "abc" "bc"</pre>
last	Retorna o último elemento da lista	<pre>&gt; last [4,3,2] 2</pre>

# Funções Pré-definidas

Função	Descrição	Exemplo
<code>elem</code>	Verifica se um elemento pertence à lista	<pre>&gt; elem 5 [1,5,10] True</pre>
<code>null</code>	Retorna verdadeiro (True) se uma lista é vazia	<pre>&gt; null [] True</pre>
<code>length</code>	Retorna o tamanho de uma lista	<pre>&gt; length "abcxyz" 6</pre>
<code>(!!)</code>	Operador de índice da lista, retorna o elemento mantido numa posição	<pre>&gt; [1,3,5,7,9] !!0 1 &gt; (!!)[ 'b', 'g', 'r', 'w' ] 3 'w'</pre>
<code>replicate</code>	Constrói uma lista pela replicação de um elemento	<pre>&gt; replicate 4 'c' "cccc"</pre>
<code>reverse</code>	Inverte os elementos de uma lista	<pre>&gt; reverse [4,5,2,2] [2,2,5,4]</pre>

# Funções Pré-definidas

Função	Descrição	Exemplo
<code>take</code>	Gera uma lista com os $n$ primeiros elementos da lista original	<pre>&gt; take 2 ['d','f','g','r'] "df"</pre>
<code>drop</code>	Retira $n$ elementos do início da lista	<pre>&gt; drop 3 [3,3,4,4,5,5] [4,5,5]</pre>
<code>takeWhile</code>	Retorna o maior segmento inicial de uma lista que satisfaz uma condição	<pre>&gt; takeWhile (&lt;10) [1,3,13,4] [1,3]</pre>
<code>dropWhile</code>	Retira o maior segmento inicial de uma lista que satisfaz uma condição	<pre>&gt; dropWhile (&lt;10) [1,3,13,4] [13,4]</pre>
<code>replicate</code>	Constrói uma lista pela replicação de um elemento	<pre>&gt; replicate 4 'c' "cccc"</pre>
<code>reverse</code>	Inverte os elementos de uma lista	<pre>&gt; reverse [4,5,2,2] [2,2,5,4]</pre>

# Funções Pré-definidas

Função	Descrição	Exemplo
<code>splitAt</code>	Divide uma lista num par de sub-listas fazendo a divisão numa determinada posição	<pre>&gt; splitAt 2 [3,4,2,1,5] ([3,4], [2,1,5])</pre>
<code>zip</code>	Recebe duas listas como entrada e retorna uma lista de pares	<pre>&gt; zip [1,2] ['a','b'] [(1,'a'), (2,'b')]</pre>
<code>sum</code>	Retorna a soma dos elementos da lista	<pre>&gt; sum [4,5,7,2,1] 19</pre>
<code>product</code>	Retorna o produto dos elementos da lista	<pre>&gt; product [5,3,6,1] 90</pre>
<code>maximum</code>	Retorna o maior elemento de uma lista	<pre>&gt; maximum [4,5,1,2] 5</pre>
<code>minimum</code>	Retorna o menor elemento de uma lista	<pre>&gt; minimum [5.2,0.3,7.2] 0.3</pre>



# Exemplos do uso de funções pré-definidas

```
geraPalindrome n = [1..n] ++ reverse [1..n]
```

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

```
fat n = product [1..n]
```

```
> fat 5  
120
```

```
fib n = fiblist !! n  
      where  
        fiblist = 1:1:[ a+b | (a,b) <-  
                          zip fiblist (tail fiblist) ]
```

```
> fib 4  
5
```

# Exemplos do uso de funções pré-definidas

Utilização da função `observe` para analisar os cálculos realizados por uma função:

```
import Hugs.Observe
fib n = (observe "fun" fiblist !! n)
      where
        fiblist = 1:1:[ a+b | (a,b) <-
                          (observe "gera"
                           zip fiblist (tail fiblist)) ]
```

```
Main> fib 7
```

```
21
```

```
>>>>>>> Observations <<<<<<<
```

```
fun
```

```
(1 : 1 : 2 : 3 : 5 : 8 : 13 : 21 : _)
```

```
gera
```

```
{ \ (1 : 1 : 2 : 3 : 5 : 8 : 13 : 21 : _)
```

```
  (1 : 2 : 3 : 5 : 8 : 13 : 21 : _)
```

```
  -> (1,1) : (1,2) : (2,3) : (3,5) : (5,8) : (8,13) : _
```

```
}
```

# Funções Recursivas

Para que possamos contar quantos elementos estão contidos numa lista, podemos escrever uma função recursiva:

```
conta :: [t] -> Int
conta [] = 0
conta (a:x) = 1 + conta x
```

```
> conta ['a','b','c']
3
```

Conta é uma função polimórfica, servindo para listas de qualquer tipo ("t" é uma variável de tipo, e pode ser substituída por qualquer tipo).

# Obter os N primeiros termos de uma lista

```
primeiros::Int-> [t] -> [t]  
primeiros 0 _ = []  
primeiros _ [] = []  
primeiros n (a:as) = a: primeiros (n-1) as
```

```
> primeiros 2 ['a','b','c']  
"ab"
```

# Função Pertence

```
pertence :: Eq t => t -> [t] -> Bool  
pertence a [] = False  
pertence a (x:z) = if (a == x) then True  
                    else pertence a z
```

```
> pertence 3 [4,5,2,1]  
False
```

# Encontrar o maior elemento da lista

```
maior [x] = x  
maior (x:y:resto) | x > y = maior (x: resto)  
                  | otherwise = maior (y: resto)
```

```
> maior [4,5,2,1]
```

```
5
```

# Função União Recursiva

```
pertence :: Eq t => t -> [t] -> Bool
pertence a [] = False
pertence a (x:z) = if (a == x) then True
                    else pertence a z

uniaoR :: Eq t => [t] -> [t] -> [t]
uniaoR [] l = l
uniaoR (x:xz) l = if pertence x l then uniaoR xz l
                  else x: uniaoR xz l
```

> uniaoR [2,3,4] [2,4,5,6]  
[3,2,4,5,6]

# Função União - por compreensão de listas

```
pertence :: Eq t => t -> [t] -> Bool
pertence a [] = False
pertence a (x:z) = if (a == x) then True
                    else pertence a z
```

```
uniao :: Eq t => [t] -> [t] -> [t]
uniao as bs = as ++ [b | b <- bs, not (pertence b as)]
```

```
> uniao [2,3,4] [2,4,5,6]
[2,3,4,5,6]
```



# Bibliografia

1. Haskell - Uma abordagem prática. Cláudio César de Sá e Márcio Ferreira da Silva. Novatec, 2006.
2. Haskell - The craft of functional programming. Simon Thompson. Pearson, 1999.