Programação com Haskell – Listas

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

Listas

- Listas são coleções de elementos
  - em que a ordem é **significativa**
  - possivelmente com **elementos repetidos**

Listas em Haskell

- Uma lista em Haskell ou é **vazia** [ ] ou é **construida** usando o operador '**:**' que acresenta um elemento a uma outra lista

- Notação em extensão
  - Listar os elementos entre colchetes separados por vírgulas

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

Sequências Aritméticas (1/2)

- Podemos construir listas de sequências aritméticas usando expressões da forma [a..b] ou [a,b..c] (onde a, b e c são números)

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

> [1,3..10]
[1,3,5,7,9]

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

Sequências Aritméticas (2/2)

- Também podemos construir listas infinitas usando expressões [a..] ou [a,b..]

```
> take 10 [1,3..]
[1,3,5,7,9,11,13,15,17,19]
```

- Se tentarmos mostrar uma lista infinita o processo não termina, temos de interromper o interpretador (usando Ctrl-C)

```
> [1,3..]
[1,3,5,7,9,11,13,15,17,19,21,23,25,27,29,31,33,35,37,
39,41,43,45,47,49,51,53,55,57,59,61,63,65,67,69,71,73,
Interrupted
```

Notação em compreensão (1/2)

- Em matemática podemos usar notação em compreensão para definir um conjunto em função de outros
  - Exemplo:

$$\{x^2 \ : \ x \in \{1, 2, 3, 4, 5\}\}$$

define o conjunto

$$\{1, 4, 9, 16, 25\}$$

Notação em compreensão (2/2)

- Em Haskell podemos usar notação em compreensão para definir uma lista a partir de outras
  - Exemplo:

```
> [x^2 | x<-[1,2,3,4,5]]
[1, 4, 9, 16, 25]
```

Geradores

- O termo x<-[1,2,3,4,5] chama-se um **gerador** e determina os valores que a variável x assume, bem como a sua ordem

- Podemos usar múltiplos geradores para variáveis distintas

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

Ordem entre geradores

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

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

- As variáveis dos geradores posteriores mudam primeiro
- Analogia: loops ‘for’ aninhados

```
for(x=1; x<=3; x++)              for(y=4; y<=5; y++)
  for(y=4; y<=5; y++)    vs.       for(x=1; x<=3; x++)
     ...                              ...
```

Dependências entre geradores (1/2)

- Geradores posteriores podem depender dos valores de geradores anteriores (mas não o contrário)

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

> [(x,y) | y<-[x..3], x<-[1..3]]
ERRO: x não está definido
```

Dependências entre geradores (2/2)

- Um exemplo - a função ***concat*** (do prelude padrão) concatena uma lista de listas

```
> concat [[1,2,3],[4,5],[6,7]]
[1,2,3,4,5,6,7]
```

- Podemos definir usando uma lista em compreensão

```
concat :: [[a]] -> [a]
concat xss = [x | xs<-xss, x<-xs]
```

Guardas

- As definições em compreensão podem incluir condições lógicas, designadas ***guardas***, para filtrar os resultados

  - Exemplo: os inteiros x tal que x está entre 1 e 10 e x é par

```
> [x | x<-[1..10], x`mod`2==0]
[2,4,6,8,10]
```

Testar primos (1/3)

- Usando uma guarda, vamos definir uma função para listar todos os divisores de um inteiro positivo

```
divisores :: Int -> [Int]
divisores n = [x | x<-[1..n], n`mod`x==0]
```

  - Exemplo:

```
> divisores 15
[1,3,5,15]
```

Testar primos (2/3)

- Podemos agora definir uma função para testar primalidade: n é primo se e só se os seus divisores são exatamente 1 e n

```
primo :: Int -> Bool
primo n = divisores n == [1,n]
```

```
> primo 15
False

> primo 19
True
```

Testar primos (3/3)

- Vamos usar o teste de primalidade como guarda para listar todos os primos (de forma ineficiente) até um certo limite dado

```
primos :: Int -> [Int]
primos n = [x | x<-[2..n], primo x]
```

- Exemplo:

```
> primos 50
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47]
```

A função zip

- A função zip definida no prelude padrão combina duas listas na lista dos pares de elementos correspondentes

```
zip :: [a] -> [b] -> [(a,b)]
```

  - Exemplo:

```
> zip ['a','b','c'] [1,2,3,4]
[('a',1), ('b',2), ('c',3)]
```

- Se as listas tiverem comprimentos diferentes o resultado tem o comprimento da menor das duas

Usando a função zip (1/5)

- Combinando ***zip*** e ***tail***, vamos definir uma função para obter os pares consecutivos de elementos de uma lista

```
pares :: [a] -> [(a,a)]
pares xs = zip xs (tail xs)
```

- Justificativa:

$$
\begin{aligned}
\text{xs} &= x_1 : x_2 : \ldots : x_n : \ldots \\
\text{tail xs} &= x_2 : x_3 : \ldots : x_{n+1} : \ldots \\
\therefore \quad \text{zip xs (tail xs)} &= (x_1, x_2) : (x_2, x_3) : \ldots : (x_n, x_{n+1}) : \ldots
\end{aligned}
$$

Usando a função zip (2/5)

- Exemplos:

```
> pares [1,2,3,4]
[(1,2),(2,3),(3,4)]

> pares [1,1,2,3]
[(1,1),(1,2),(2,3)]

> pares [1,2]
[(1,2)]

> pares [1]
[]
```

Usando a função zip (3/5)

- Usando a função ***and :: [Bool] -> Bool*** do prelude padrão,

  $$\text{and } [b_1, b_2, \ldots, b_n] = b_1 \text{ ^ } b_2 \text{ ^ } \ldots \text{ ^ } b_n,$$

  e a função *pares,* podemos escrever uma função para verificar se uma lista está em ordem crescente

```
crescente :: Ord a => [a] -> Bool
crescente xs = and [x<=x' | (x,x')<-pares xs]
```

Usando a função zip (4/5)

- Alguns exemplos

```
> crescente [2,3]
True

> crescente [2,3,4,7,8]
True

> crescente [2,8,3,7,4]
False
```

- Qual será o resultado com uma lista de um só elemento? E com a lista vazia?

Usando a função zip (5/5)

- Podemos usar ***zip*** para combinar elementos com os seus índices na lista
  - Exemplo: procurar um valor em uma lista e obter todos os seus índices

```
indices :: Eq a => a -> [a] -> [Int]
indices x ys = [i | (i,y)<-zip [0..n] ys, x==y]
     where n = length ys - 1
```

  - Exemplo

```
> indices 'a' ['b','a','n','a','n','a']
[1,3,5]
```

Cadeias de caracteres (1/2)

- O tipo ***String*** é pré-definido no prelude padrão como um sinônimo de lista de carateres

```
type String = [Char]
```

- Por exemplo

```
"abc"
```

é equivalente a

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

Cadeias de caracteres (2/2)

- Como as cadeias são listas de carateres, podemos usar as funções de listas com cadeias de carateres

- Exemplos:

```
> length "abcde"
5

> take 3 "abcde"
"abc"

> zip "abc" [1,2,3,4]
[('a',1),('b',2),('c',3)]
```

Cadeias em compreensão

- Como as cadeias são listas, também podemos usar notação em compreensão com cadeias de carateres
  - Exemplo: contar letras minúsculas

```
minusculas :: String -> Int
minusculas txt = length [c | c<-txt, c>='a' && c<='z']
```

Processamento de listas e de caracteres (1/2)

- Muitas funções genéricas estão pré-definidas em bibliotecas (designadas **módulos** em Haskell)

- Podemos colocar uma declaração **import** para usar funções definidas num módulo. Exemplo:

```
import Char                                    -- usar o módulo Char

minusculas :: String -> Int
minusculas txt = length [c | c<-txt, isLower c]
        -- isLower testa se um caracter é uma letra minúscula
```

Processamento de listas e de caracteres (2/2)

- Um outro exemplo: converter uma cadeia de carateres em maiúsculas

```
import Char                                    -- usar o módulo Char

stringUpper :: String -> String
stringUpper txt = [toUpper c | c<-txt]
        -- toUpper converte letras minúsculas para maiúsculas
```