Compreensão de Listas e Funções de Alta Ordem

Profa. Juliana Kaizer Vizzotto

juvizzotto@gmail.com

Disciplina de Paradigmas de Programação - Juliana Kaizer Vizzotto

Compreensão de Listas (*List Comprehensions*)

- É uma maneira de se descrever uma lista inspirada na notação de conjuntos. Por exemplo, se a lista `list` é [1,7,3], pode-se duplicar o valor dos elementos da lista da seguinte maneira:

  ```
  [2 * a | a <- list]
  ```

  que terá valor:

  ```
  [2,14,6]
  ```

  ou

  ```
  Hugs> [2 * a | a <- [1,7,3]]
  [2,14,6]
  ```

- Na `list Comprehension` o `a <- list` é chamdo de gerador da lista,

Compreensão de Listas (*List Comprehensions*)

- Os geradores podem ser combinados com predicados que são funções que devolvem valores booleanos (`a -> Bool`):

```
Hugs> [a | a <- list, even a]
```

- A função `even` devolve o valor booleano True se o seu argumento for um número par. Então, a compreensão de listas devolve apenas os números pares da lista `list`.
- Pode-se também trabalhar com qualquer tipo de padrão:

```
somaPares :: [(Int,Int)]->[Int]
somaPares lista = [a+b | (a,b) <- lista]

Hugs> somaPares [(2,3),(3,7),(4,5)]
[5,10,9]
```

Compreensão de Listas (*List Comprehensions*)

- Quando se trabalha com mais de um gerador, o primeiro valor da primeira lista é gerado e mantido enquanto se avalia os valores da lista seguinte:

```
pares :: [t] -> [u]-> [(t,u)]
pares n m = [(a,b) | a <- n, b <- m]
```

- Crie um filtro para strings utilizando compreensão de listas:

```
filtro1 :: Char -> [Char] -> [Char]
```

- Considere a seguinte lista de tuplas, em que cada tupla tem-se o número do aluno, o nome aluno e sua nota:

```
baseDeDados :: [(Int, String, Float)]
baseDeDados = [(1, "Juliana", 10.0),
               (2, "Carlos", 6.8),
               (3, "Maria",7.0)]
```

Compreensão de Listas (*List Comprehensions*)

- Implemente uma função utilizando compreensão de listas que transforma uma `baseDeDados` em uma lista de nomes de alunos:

  ```
  nomes : [(Int, String,Float)] ->[String]
  ```

Funções de Alta Ordem

- Funções que podem receber **funções** como argumentos!
- Em geral, em linguagens funcionais, as funções podem receber funções como argumentos e também retornar funções.
- Funções desse tipo são chamadas de *funções de alta ordem*.

Funções de Alta Ordem para trabalhar com listas

- A maioria das definições sobre listas se encaixam em três casos:
- `folding`, que é a colocação de um operador entre os elementos de uma lista;
- `filtering`, que significa filtrar alguns elementos, e
- `map`, que é a aplicação de funções a todos os elementos da lista.
- Existem funções pré-definidas em Haskell que servem para resolver estes casos: `foldr1`, `map` e `filter`.

Definições

- A função `foldr1` tem o seguinte tipo e definição

```
foldr1 :: (t->t->t) -> [t]-> t

foldr1 f [a] = a
foldr1 f (a:b:x) = f a (foldr1 f (b:x))
```

- Por exemplo:

```
Hugs> foldr1 (&&) [True, False,True]
False

Hugs> foldr1 (++) ["Concatenar", "uma", "lista"]
"Concatenar uma lista"
```

Definições

- A função `map` aplica uma função a todos os elementos da lista

```
map :: (t->u) -> [t]-> [u]

map f [] = []
map f (a:x) = f a : map f x
```

- Por exemplo:

```
Hugs> map length ["Haskell", "Hugs", "GHC"]
[7,4,3]

Hugs> map (2*) [1,2,3]
[2,4,6]
```

Definições

- A função `filter` filtra a lista através de um predicado ou propriedade. Um predicado é uma função que tem tipo `t -> Bool`, como por exemplo:

```
par :: Int -> Bool
par n = (n `mod` 2 == 0)

filter :: (t-> Bool) -> [t]-> [t]

filter p [] = []
filter f (a:x)
     | p a       = a:filter p x
     |otherwise = filter p x
```

- Por exemplo:

```
Hugs> filter par [2,4,5,6,10,11]
[2,4,6,10]
```

Definições

- Defina `filter` com compreensão de listas.
- Estude para que servem as funções: `takeWhile`, `dropWhile` e `zip`.

Listas Infinitas

- A linguagem Haskell, assim como todas as linguagens funcionais puras, são chamadas de *non-strict languages*, elas trabalham com *lazy evaluation*, ou seja, os argumentos de uma função são avaliados somente quando necessário. Se temos, por exemplo a função:

  `f(x) = 7`

  e passamos o seguinte argumento:

  `f((21 +33)* 8) = 7`

  é realmente necessário realmente perder-se tempo computacional avaliando-se a expressão: `((21+33) * 8)`, se a avaliação da função sempre gera o valor 7, independente do argumento?
- Em linguagens imperativas como C e Pascal, os argumentos sempre são avaliados antes de serem passados para as funções.

Listas Infinitas

- Como é a avaliação em java?
- A *lazy evaluation* nos permite trabalhar com estruturas infinitas
- Na *lazy evaluation* apenas partes de uma estrutura de dados são avaliados
- Imagine a seguinte lista:

```
uns = 1 :uns
```

- Passe esta estrutura para o interpretador.
- Agora, considere a seguinte função:

```
somaOsDoisPrimeiros :: [Int] -> Int
somaOsDoisPrimeiros (a:b:x) = a + b
```

- Temos

```
Hugs> somaOsDoisPrimeiros uns
2
```

Listas Infinitas

- Estude a função `iterate` do Haskell.
- Defina uma função que calcule as potências ímpares de um número inteiro.