Exercícios em Programação Funcional

Compreensão de listas e vetores

Programação Funcional
Bacharelado em Sistemas de Informação
Maio - 2009

List Comprehension

Uma lista pode ser especificada pela **definição** de seus elementos. A *compreensão de listas (list comprehension)* é uma forma de definir uma construção considerando critérios:

```
novaLista :: [(Int,Int,Int)]
novaLista = [(i,j,i*j) | i <- [2,4..100],
                         j <- [3,6..100],
                         0 == ((i+j) `mod` 7)]
```

```
Main> novaLista
[(2,12,24),(2,33,66),(2,54,108),(2,75,150), ... ]
```

Os critérios são: i é par, j é múltiplo de 3 e a soma de i e j deve ser divisível por 7.

List Comprehension

Para construir um vetor com os 100 primeiros quadrados (ou cubos), podemos utilizar a *compreensão de listas*:

```
quadrados = array (1,100) [(i, i*i) | i <- [1..100]]

Main> quadrados

Array (1,100) [(1,1),(2,4),(3,9),(4,16),(5,25),(6,36),
(7,49),(8,64),....,(98,9604),(99,9801),(100,10000)]


cubos =  array (1, 100) [(i, i*i*i) |  i <- [1..100]]

Main> cubos

array (1,100) [(1,1),(2,8),(3,27),(4,64),(5,125),(6,216),
(7,343),(8,512),....,(99,970299),(100,1000000)]
```

List Comprehension

```
a = array (1,10) ((1,1) : [(i, i * a!(i-1))
                           | i <- [2..10]])
```

```
Main> a

array (1,10) [(1,1),(2,2),(3,6),(4,24),(5,120),
(6,720),(7,5040),(8,40320),(9,362880),(10,3628800)]
```

O operador (!) pode ser usado para a obtenção de elementos a partir de suas posições no vetor:

```
Main> a ! 3
6
Main> a ! 100
*** Exception: Error in array index
```

Construção de Vetor (*ListArray*)

A função *listArray* também pode ser utilizada para a construção de vetores. Seu primeiro parâmetro corresponde às dimensões do vetor, e o segundo parâmetro contém os valores a serem inseridos.

```
Main> listArray (0,2) "foo"
array (0,2) [(0,'f'),(1,'o'),(2,'o')]

Main> listArray (0,3) [True,False,False,True,False]
array (0,3) [(0,True),(1,False),(2,False),(3,True)]
```

Construção de Vetor (*Array*)

Para gerar os n primeiros números da sequência de Fibonnacci podemos escrever:

```
fibs     :: Int -> Array Int Int
fibs n   =  a
            where a = array (0,n) ([(0, 1), (1, 1)]
                      ++
                      [(i, a!(i-2) + a!(i-1))
                      | i <- [2..n]])
```

```
Main> fibs 8

array (0,8) [(0,1),(1,1),(2,2),(3,3),(4,5),(5,8),(6,13),
(7,21),(8,34)]
```

Exercícios: 1) Série Simples

Usando as técnicas de compreensão de listas e funções genéricas, calcule o resultado da aplicação da sequência abaixo à uma lista de valores:

$$\frac{X_1^2}{1} + \frac{X_2^2}{2} + \ldots + \frac{X_n^2}{n}$$

Parte 1: Gerar os números da Série

```
elemSerie::[Int]->[Float]
elemSerie xs = divSerie 1 (map (^2) xs)


divSerie::Int->[Int]->[Float]
divSerie b [] = []
divSerie b (x:xs) = ((/) (fromIntegral x)  (fromIntegral b)):
                    divSerie (b+1) xs
```

1) Série Simples - cont.

$$\frac{X_1^2}{1} + \frac{X_2^2}{2} + \ldots + \frac{X_n^2}{n}$$

Parte 2: Somar os números da série (gerados)

```
somaSerie xs = foldr1 (+) (elemSerie  xs)
```

```
Main> elemSerie [3,4,1,5,6]
[9.0,8.0,0.3333333,6.25,7.2]
Main> somaSerie [3,4,1,5,6]
30.78333
```

2) Série de Taylor

Faça uma função para calcular a expansão e^x definida pela série de Taylor :

$$1 + \frac{X}{1!} + \frac{X^2}{2!} + \frac{X^3}{3!} \ldots + \frac{X^n}{n!} = e^x$$

```
taylor::Int -> Int -> Double
taylor n exp = foldr1 (+) (elemsTaylor n exp)


elemsTaylor::Int -> Int -> [Double]
elemsTaylor n exp = 1:[ (/) (fromIntegral (exp^y))
                            (fromIntegral (fatorial y))
                      | y <- [1..n]]


Main> elemsTaylor 5 5
[1.0,5.0,12.5,20.8333333333333,26.0416666666667,26.0416666666667]

Main> taylor 10 5
146.380601025132

Main> taylor 5 5
91.4166666666667
```

3) Transposição de uma matriz (m x m)

Dada uma matriz (m x m), podemos definir uma função que retorne a matriz transposta correspondente:

```
mat:: Array (Int,Int) Int
mat = array ((1,1),(2,2)) [((1,1),2),((1,2),2),((2,1),3),
((2,2),4)]
```

```
transpose :: Array (Int,Int) Int ->  Array (Int,Int) Int
transpose a = array ((li,ui), (lj,uj))
              [((i,j),a!(j,i))
                | i <- [li..lj], j <- [ui..uj]]
              where ((li,ui), (lj,uj)) = bounds a
```

```
Main> transpose mat
array ((1,1),(2,2)) [((1,1),2),((1,2),3),((2,1),2),((2,2),4)]
```

4) Devolvendo o troco

Para calcular as combinações de notas para devolver o troco durante um pagamento, podemos definir a função :

```
notas:: [Int]
notas =  [1,2,5,10,20,50,100]

notasTroco :: Int -> [[Int]]
notasTroco 0 = [[]]
notasTroco valor = [v:vs | v <- notas, valor >= v,
                           vs <- notasTroco (valor-v) ]


Main> notasTroco 4
[[1,1,1,1],[1,1,2],[1,2,1],[2,1,1],[2,2]]
```

5) Listas Geradas

Mostre as listas geradas pelas expressões:

```
A)    [n*n | n<-[1..10], even n]
B)    [7 |n<-[1..4]]
C)    [ (x,y) | x<-[1..3], y<-[4..7]]
D)    [ (m,n) | m<-[1..3], n<-[1..m]]
E)    [j | i<-[1,-1,2,-2], i>0, j<-[1..i]]
F)    [a+b | (a,b)<-[(1,2),(3,4),(5,6)]]
```

6) Exercícios Propostos

A) Faça uma função (usando compreensão de listas) que calcula a quantidade de números negativos de uma lista:

```
listaNeg :: [Int] -> Int
listaNeg n = ...

> listaNeg [1,-3,-4,3,4,-5]
3
```

B) Crie um registro com no mínimo 20 nomes e idades:

```
Reg = [(15,”Ana”),(22,”Pedro”),...]
```

Implemente uma função para ordenar o registro (use o método da Bolha) considerando as idades.