Programação Funcional
Folha de Exercícios 02
Recursão

Prof. Wladimir Araújo Tavares

1. Quais são os tipos das seguintes funções:

   (a)
   ```
   remove x [] = []
   remove x (y:ys) = if x == y then ys else y: (remove x ys)
   ```
   ```
   remove :: Eq a => a -> [a] -> [a]
   ```

   (b)
   ```
   partes [] = [ [] ]
   partes (x:xs) = [ x:y | y <- partes xs] ++ partes xs
   ```

   (c)
   ```
   rota n xs = drop n xs ++ take n xs
   ```

   (d)
   ```
   swap (x,y) = (y,x)
   ```

   (e)
   ```
   twice f x = f (f x)
   ```

2. Explique o que cada função da questão anterior faz?
3. Defina cada uma das seguintes funções usando apenas funções pré-definidas em Haskell:

   (a) `prodIntervalo :: Int -> Int -> Int` que, dados dois valores inteiros n e m, retornar o produto de todos os valores inteiros entre n e m (inclusive). Por exemplo,
   `prodIntervalo 2 5 ==` $\Pi_{i=2}^{5} i$ `== 2 * 3 * 4 * 5 == 120`
   Dica: Use a função product.

   (b) `somaIntervalo :: Int -> Int -> Int` que, dados dois valores inteiros n e m, retornar o produto de todos os valores inteiros entre n e m (inclusive). Por exemplo,
   `somaIntervalo 2 5 ==` $\sum_{i=2}^{5} i$ `== 2 + 3 + 4 + 5 == 14`
   Dica: Use a função sum.

   (c) Defina a função `metades :: [a] -> ([a],[a])` tal que (metades xs) é formada pelo par formado por duas listas obtida pela divisão de xs em que a cardinalidade das listas diferem no máximo em um.
   `metades [2,3,5,7,9] == ([2,3],[5,7,9])`
   Dica: Use a função `splitAt :: Int -> [a] -> ([a], [a])`

4. Defina cada uma das seguintes funções usando recursão:

   (a) Defina a função `potencia :: Integer -> Integer -> Integer` tal que (potencia x n) é x elevado ao número natual n. Por exemplo,
   `potencia 2 3 == 8`

   (b) Defina a função `elemento :: Eq a => a -> [a] -> Bool` tal que (elemento x xs) verifica se x pertence a lista xs. Por exemplo, `elemento 3 [2,3,5] == True`

   (c) Defina a função `seleciona :: Int -> [a] -> a` tal que (seleciona n xs) é o n-ésimo elemento de xs. Por exemplo, `seleciona 2 [2,3,5,7] == 5`

   (d) Defina a função `refinada :: [Float] -> [Float]` tal que (refinada xs) é uma lista obtida intercalando dois elementos consecutivos de xs com a média aritmética deles. Por exemplo,
   ```
   refinada [2,7,1,8] == [2.0, 4.5, 7.0, 4.0, 1.0, 4.5, 8.0]
   refinada [2] == [2.0]
   refinada []  == []
   ```

   (e) Defina a função `merge :: [a] -> [a] -> [a]` tal que (merge xs ys) é uma lista ordenada obtida pela entrelaçamento de duas listas ordenadas xs e ys. Por exemplo,
   `merge [2,5,6] [1,3,4] == [1,2,3,4,5,6]`

   (f) Usando a função merge e metades, escreva uma definição recursiva da função `mergesort :: Ord a => [a] -> [a]` que implementa o método merge sort :
   - uma lista vazia ou com um só elemento já está ordenada;
   - para ordenar uma lista com dois ou mais elementos, partimos em duas metades, recursivamente ordenamos as duas parte e juntamos os resultados usando merge.

   (g) Defina uma função `ordenada :: Ord a => [a] -> Bool` tal que ordenada xs verifica se xs é uma lista ordenada. Por exemplo,
   ```
   ordenada [2,3,5] == True
   ordenada [2,5,3] == False
   ```

   (h) `subconjunto :: Eq a => [a] -> [a] -> Bool` tal que (subconjunto xs ys) verifica xs é um subconjunto de ys. Por exemplo,
   ```
   subconjunto [3,2,3] [2,5,3,5] == True
   subconjunto [3,2,3] [2,5,6,5] == False
   ```
   Dica: use a função `elem :: a -> [a] -> Bool` verifica se um elemento pertence a uma lista.

   (i) `union :: Eq a => [a] -> [a] -> [a]` tal que (union xs ys) é a união dos conjuntos xs e ys. Por exemplo,
   ```
   union [3,2,5] [5,7,3,4] == [3,2,5,7,4]
   ```
   Dica: use a função `notElem :: a -> [a] -> Bool` verifica se um elemento não pertence a uma lista.

   (j) `diferencia :: Eq a => [a] -> [a] -> [a]` tal que (diferencia xs ys) é a diferença entre os conjuntos xs e ys. Por exemplo,
   ```
   diferencia [3,2,5,6] [5,7,3,4] == [2,6]
   diferencia [3,2,5] [5,7,3,2] == []
   ```

   (k) Defina a função `frequencia :: a -> [a] -> Int` tal que (`frequencia x xs`) devolve o número de ocorrências de x em u. Por exemplo,
   ```
   frequencia 5 [4,5,2,1,5,5,9] == 3
   ```

   (l) Defina a função `unico :: Eq a => a -> [a] -> Bool` tal que (`unico x xs`) devolve True se x ocorre exatamente uma vez em u e False, caso contrário.
   ```
   unico 2 [1,2,3,2] == False
   unico 2 [3,1] == False
   unico 2 [2] == True
   ```
   Dica: use a função `notElem :: Eq a => a -> [a] -> Bool`

   (m) Dado uma lista de números inteiros, definiremos o maior salto como o maior valor das diferenças (em valor absoluto) entre números consecutivos da lista. Por exemplo, dada uma lista [2,5,-3,7]
   - 3 (valor absoluto de 2 - 5)
   - 8 (valor absoluto de 5 - (-3))
   - 10 (valor absoluto de -3 - 7)

   Portanto o maior salto é 10. Não esta definido o maior salto para uma lista com menos de 2 elementos. Defina a função `maiorSalto :: [Integer] -> Integer` tal que maiorSalto xs é o maior salto da lista xs. Por exemplo,
   ```
   maiorSalto [1,5] == 4
   maiorSalto [10,-10,1,4,20,-2] == 22
   ```

   (n) Considere um polinômio $P(X) = c_0 + c_1 z + \ldots + c_n z^n$ representado pela lista dos seus coeficientes $[c0, c1, \ldots; c_n]$. Podemos calcular o valor do polinômio num ponto de forma eficiente usando a forma de Horner :

   $$P(z) = c0 + c_1 z + \ldots + c_n z_n = c_0 + z*(c_1 + z*(\ldots + z*(c_{n-1} + z*c_n)\ldots)) \quad (1)$$

   Note que usando a expressão não necessitamos de calcular potências: para calcular o valor dum polinômio de grau n usamos apenas n adições e n produtos.
   Complete a seguinte definição recursiva tal que horner cs z calcula o valor do polinômio com lista de coeficientes cs no ponto z usando a forma de Horner.
   ```
   horner :: [Double] -> Double -> Double
   horner [] z = 0
   horner (c:cs) z =
   ```

   (o) A função length, que computa o número de elementos de uma lista, pode ser definida do seguinte modo:
   ```
   length xs = length' 0 xs
               where length' n [] = n
   length' n (x:xs) = length' (n+1) xs
   ```
   Essa função usa a função auxiliar length', que possui um parâmetro adicional para acumular o resultado. A função `length'` é definida usando recursão de cauda, uma vez que a chamada recursiva `length' (n+1) xs`, usada no lado direito da definição, não ocorre dentro de nenhum argumento de outra função. Use essa técnica de recursão de cauda para definir as seguintes funções:
   i. `fac :: Int -> Int`, que computa o fatorial de um número natural
   ii. `reverse :: [a] -> [a]`, que inverte uma lista.

   (p) Defina uma função `remove :: Eq a => a -> [a] -> [a]` tal que (`remove x xs`) devolve uma lista obtida removendo todas as ocorrências de x em xs. Por exemplo,
   ```
   remove 2 [1,2,5,2,4,3,2] == [1,5,4,3]
   ```

   (q) Defina uma função `unique :: Eq a => [a] -> [a]` tal que (`unique xs`) devolve uma lista com os elementos de xs sem repetições.
   ```
   unique [1,2,5,2,5,7,2,5] == [1,2,5,7]
   ```

   (r) Defina uma função `inserir :: Ord a => a -> [a] -> [a]` tal que (`inserir x xs`) devolve uma lista ordenada ascendentemente, oriunda da inserção apropriada de x em xs. Por exemplo,
   `inserir 3 [2,7,12]`

   (s) Escreva uma função `permutations :: [a] -> [[a]]` para obter a lista com todas as permutações dos elementos uma lista. Assim, se xs tem comprimento n, então permutations xs tem comprimento n!.
   Exemplo: `permutations [1, 2, 3] = [[1, 2, 3], [2, 1, 3], [2, 3, 1], [1, 3, 2], [3, 1, 2], [3, 2, 1]]`
   Note que a ordem das permutações não é importante.