Programação Funcional

Questões dos Testes Práticos

LCC/LEI – 2008/2009

1. Defina uma função `quantos :: a -> [a] -> Int` que conta quantos elementos de uma lista são iguais a um dado elemento.
2. Defina uma função `maiores :: [a] -> Int` que, dada uma lista não vazia determina quantos elementos são maiores do que o primeiro.
3. Defina uma função `positivos :: [Int] -> Int` que calcula quantos elementos de uma lista de inteiros são positivos.
4. Defina uma função `contaMai :: [Char] -> Int` que determina quantos elementos de uma lista de caracteres são letras maiúsculas (use se precisar a função `isUpper :: Char -> Bool`).
5. Defina uma função `maiores :: [a] -> [a]` que, dada uma lista não vazia, determina a lista dos elementos que são maiores do que o primeiro.
6. Defina uma função `positivos :: [Int] -> [Int]` que, dada uma lista de inteiros, determina os números positivos dessa lista.
7. Defina uma função `impares :: [Int] -> [Int]` que, dada uma lista de inteiros, determina os números impares dessa lista.
8. Defina uma função `maior :: [Int] -> Int` que devolve o maior elemento de uma lista de inteiros não vazia.
9. Defina uma função `soConsoantes :: String -> String` que remove todas as vogais de uma `String`. Pode usar uma função pré-definida `vogal :: Char -> Bool` que testa se um carácter é uma vogal.
10. Defina a função `ordenada :: [Int] -> Bool` que testa se uma lista de inteiros está ordenada.
11. Defina uma função `pares :: [Int] -> Int` que, dada uma lista de inteiros, determina quantos números pares essa lista tem.
12. Defina uma função `contaAlg :: [Char] -> Int` que determina quantos elementos de uma lista de caracteres são algarismos (use, se precisar, a função `isDigit :: Char -> Bool`).
13. Considere o tipo `type Rectangulo = (Int,Int)` para representar rectângulos.
    (a) Defina uma função `quadrados :: [Rectangulo] -> Int` que, dada uma lista com rectângulos, conta quantos deles são quadrados.
    (b) Defina uma função `areaTotal :: [Rectangulo] -> Int` que calcula a área total de uma lista de rectângulos.
14. Defina a função `descomprime :: [(a,Int)] -> [a]` que replica cada elemento da lista de entrada o número de vezes especificado. Por exemplo, `descomprime [('a',2),('b',4)] == "aabbbb"`.

15. Defina a função `remove :: [a] -> Int -> [a]` que remove o elemento da lista que se encontra na posição especificada. Por exemplo, `remove [1,2,3,4] 2 == [1,2,4]`

16. Defina a função `copia :: [a] -> [Int] -> [a]` que copia para o resultado os elementos da primeira lista que se encontram nas posições indicadas pela segunda lista. Por exemplo, `copia "abcde" [1,3,1] == "bdb"`.

17. Defina a função (pré-definida em Haskell) `replicate :: Int -> a -> [a]` que constrói uma lista com n cópias de um elemento (por exemplo, `replicate 3 19 = [19,19,19]`).

18. Sem usar definições por compreensão, defina a função `gama :: Int -> Int -> [Int]` que constrói a lista dos números inteiros entre dois limites (por exemplo, `gama 3 9 = [3,4,5,6,7,8,9]`).

19. Defina a função `intercala :: a -> [a] -> [a]` que intercala um dado elemento entre cada dois elementos da lista. Por exemplo, `intercala ',' "abc" == "a,b,c"`.

20. Defina recursivamente a função `iguais :: [Int] -> [Int] -> Bool` que testa se duas listas são iguais.

21. Defina recursivamente a função `init :: [a] -> [a]` que elimina o último elemento de uma lista. Por exemplo, `init [1,2,3,4] == [1,2,3]`.

22. Defina recursivamente a função `splitAt :: Int -> [a] -> ([a],[a])` que parte uma lista numa determinada posição. Por exemplo, `splitAt 2 "abcde" == ("ab","cde")`.

23. Uma matriz em Haskell pode ser definida por uma lista de listas (uma lista para cada linha).

```
type Matriz = [Linha]
type Linha = [Float]
```

    (a) Defina uma função `maxMat :: Matriz -> Float` que calcula o maior elemento de uma matriz (não vazia).

    (b) Defina uma função `quantos :: Float -> Matriz -> Int` que calcula quantas vezes um determinado número ocorre numa matriz.

    (c) Defina uma função `ok :: Matriz -> Bool` que testa se uma matriz está bem formada. Em particular, a função deverá exibir o seguinte comportamento:

```
> ok [[1,2,3],[3,2,1]]
True
> ok [[1,2,3],[3,2]]
False
```

    (d) Defina uma função `zero :: Int -> Int -> Matriz` que, dadas as dimensões pretendidas, crie uma matriz de zeros.

```
> zero 3 2
[[0,0,0],[0,0,0]]
```

24. Considere a seguinte função que calcula o comprimento da `String` mais longa de uma lista.

```
compMaisLonga :: [String] -> Int
compMaisLonga [s]    = length s
compMaisLonga (s:ss) = max (length s) (compMaisLonga ss)
```

    Apresente uma definição alternativa desta função usando funções de ordem superior (como por exemplo, `map` ou `filter`) em vez de recursividade explícita. Pode usar também a função `maximum` que calcula o maior número de uma lista.

25. Considere a seguinte função que calcula quantos elementos de uma lista de `Strings` são nomes próprios (começam com uma maiúscula).

```
nomesProp :: [String] -> Int
nomesProp [] = 0
nomesProp (h:t) | proprio h = 1 + (nomesProp t)
                | otherwise = nomesProp t

proprio (c:_) = isUpper (c)
```

Apresente uma definição alternativa desta função usando funções de ordem superior (como por exemplo, `map` ou `filter`) em vez de recursividade explícita. Pode usar também a função `proprio` definida acima.

26. Considere a seguinte função que calcula quantas vezes um determinado elemento aparece numa lista.

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

Apresente uma definição alternativa desta função usando funções de ordem superior (como por exemplo, `map`, `filter` ou `zipWith`) em vez de recursividade explícita.

27. Considere a seguinte função que, dada uma lista de quadrados e rectângulos, calcula a área total dos quadrados.

```
areaQuadrados :: [(Int,Int)] -> Int
areaQuadrados [] = 0
areaQuadrados (h:t) | quadrado h = area h + areaQuadrados t
                    | otherwise = areaQuadrados t

quadrado (h,v) = h==v
area (h,v) = h*v
```

Apresente uma definição alternativa desta função usando funções de ordem superior (como por exemplo, `map`, `filter` ou `zipWith`) em vez de recursividade explícita. Pode também usar as funções `quadrado` e `area` definidas acima.

28. Defina a função `diferentes :: Eq a => [a] -> Bool` que testa se todos os elementos de uma lista são diferentes entre si.

29. Defina a função `leq :: String -> String -> Bool` que testa se a primeira lista antecede a segunda na ordem lexicográfica.

30. Assuma que os registos das temperaturas mínimas e máximas de vários dias estão guadados na estrutura de dados `TabTemp`. Considere as seguintes definições:

```
type TabTemp = [(Data,TempMin,TempMin)]
type Data = (Int,Int,Int)
type TempMin = Float
type TempMax = Float
```

(a) Defina a função `medias :: TabTemp -> [(Data,Float)]` calcule a temperatura média de cada dia.

(b) Defina a função `minMin :: TabTemp -> Float` que calcula a temperatura mínima mais baixa registada na tabela.

31. Assuma que a informação sobre os resultados dos jogos de uma jornada de um campeonato de futebol está guardada na seguinte estrutura de dados:

```
type Jornada = [Jogo]
type Jogo = ((Equipa,Golos),(Equipa,Golos))
type Equipa = String
type Golos = Int
```

(a) Defina a função `golosMarcados :: Jornada -> Int` calcule o número total de golos marcados numa jornada.

(b) Defina a função `pontos :: Jornada -> [(Equipa,Int)]` que calcula os pontos que cada equipa obteve na jornada (venceu - 3 pontos; perdeu - 0 pontos; empatou - 1 ponto)

32. Assuma que os registos das temperaturas mínimas e máximas de vários dias estão guadados na estrutura de dados `TabTemp`. Considere as seguintes definições:

```
type TabTemp = [(Data,TempMin,TempMin,Precipacao)]
type Data = (Int,Int,Int)
type TempMin = Float
type TempMax = Float
type Precipacao = Float
```

(a) Defina a função `amplTerm :: TabTemp -> [(Data,Float)]` calcule a amplitude termica (diferença entre temperatua máxima e mínima) de cada dia.

(b) Defina a função `maxChuva :: TabTemp -> (Data,Float)` que calcula o dia em que a precipitação foi mais elevada e o seu valor

33. Para armazenar as notas finais dos alunos definiram-se os seguintes tipos:

```
type Tabela = [(Num,Nota)]
type Num = String
type Nota = Int
```

(a) Defina a função `parte :: Tabela -> ([Num],[Num])` que dada a tabela de notas devolve um par de listas com o nome dos alunos reprovados e aprovados (isto é, com nota superior ou igual a 10).

(b) Defina a função `nota :: Num -> Tabela -> Maybe Nota`, que permite saber a nota de um dado aluno, caso exista.

34. Defina a função `splitAt :: Int -> [a] -> ([a],[a])` que recebe um número inteiro n e uma lista, e produz um par de listas, sendo a 1ª lista constituída pelos n primeiros elementos da lista de entrada e a 2ª lista constituída pelos restantes elementos. Por exemplo:

```
Prelude> splitAt 3 [5,3,4,2,1]
([5,3,4],[2,1])
```

35. Defina a função `maiorDoQue :: Int -> [Int] -> Maybe Int`, que dado um número e uma lista ordenada de números, devolve o primeiro número da lista que é maior do que esse número.

36. Defina uma função `filtragem :: (a->Bool) -> [a] -> ([a],[a])` que recebe um predicado e uma lista, e devolve um par que tem na 1ª componente os elementos da lista que satisfazem o predicado e na 2ª componente os elementos da lista que não satisfazem o predicado.

37. Considere as seguintes definições

```
data Avaliacao = A Float Float -- Teste, prática
               | B Float

type Aluno = (Int, String, Avaliacao) -- Numero, Nome, Av
type Turma = [Aluno]
```

    (a) Defina uma função `nota :: Avaliacao -> Maybe Int` que calcula a nota final de um aluno (deve retornar `Nothing` no caso de reprovar). Note que:
    - No método B, o aluno só tem aprovação se a nota do teste for superior ou igual a $9,5$.
    - No método A, o aluno só tem aprovação se a nota dos teste for superior ou igual a $8,0$ e a média pesada $(0.7 * t + 0.3 * p)$ for superior ou igual a $9,5$.

    (b) Sabendo que existe uma função `nota :: Avaliação -> Maybe Int` que calcula a nota final de um aluno (`Nothing` se reprovado), defina uma função `pauta:: Turma -> [(String, String, String)]` que produz a pauta.

    Por exemplo,

    `pauta [(123, "Joao", A 10 10), (124, "Maria", B 11), (125, "Manuel", A 5 10)]` deve produzir a lista

    `[("123", "Joao", "10"), ("124", "Maria", "11"), ("125", "Manuel", "Rep")]`

    (c) Defina uma função `melhorA :: Turma -> Maybe Int` que determina o número do aluno do método A que teve melhor nota no teste.

    (d) Sabendo que no método A a nota final do aluno é calculada por $0.7 * t + 0.3 * p$ defina uma função `finais :: Turma -> [(Int,Float)]` que calcula as notas dos alunos de uma turma.

    (e) Sabendo que no método A a nota final do aluno é calculada por $0.7 * t + 0.3 * p$ com nota mínima de 8 no teste, defina uma função `aprovado :: Aluno -> Bool` que testa se um dado aluno foi aprovado.

    (f) Sabendo que existe uma função `aprovado :: Bool` que testa se um dado aluno foi ou não aprovado, defina uma função `stat:: Turma -> (Float,Float)` que calcula os números de aprovados em cada um dos métodos de avaliação.

38. Considere a seguinte definição que calcula o comprimento da *string* mais longa de uma lista.

```
compMaisLonga :: [String] -> Int
compMaisLonga [s] = length s
compMaisLonga (s:ss) = max (length s) (compMaisLonga ss)
```

    Apresente uma versão alternativa desta função usando `map` (para calcular o comprimento de todas as *strings*) e `maximum`.

39. Considere a seguinte definição que calcula quantos elementos de uma lista de *strings* são nomes próprios (começam com uma maiúscula).

```
nomesProp :: [String] -> Int
nomesProp [] = 0
```

```
nomesProp (h:t) | proprio h = 1 + (nomesProp t)
                | otherwise = nomesProp t

proprio (c:_) = (isUpper (c))
```

Apresente uma versão alternativa desta função usando `filter` (para seleccionar os nomes próprios) e `length` (pode usar a função `proprio` definida acima).

40. Considere a seguinte definição da função `(!!) :: [a] -> Int -> a` pré definida em Haskell, de selecção de um elemento da lista.

```
l !! n = head (drop n l)
```

Apresente uma definição alternativa (recursiva) sem usar a função `drop`.

41. Considere a seguinte definição de uma função que retira um elemento a uma lista

```
allBut :: [a] -> Int -> [a]
allBut l n = (a,tail b)
   where (a,b) = splitAt n l
```

Apresente uma definição alternativa (recursiva) sem usar a função `splitAt`.

42. Relembre a definição da função `split :: a -> [a] -> ([a],[a])` usada pelo *quicksort* para partir uma lista em duas:

```
split a l = ([x | x<-l, x<=a], [x | x<-l, x > a])
```

Apresente uma definição alternativa (recursiva) da função que não use definições de listas por compreensão e que percorra a lista uma única vez.

43. Considere a seguinte definição da função `extractMultiples :: [Int] -> Int -> ([Int], [Int])` que, dada uma lista de inteiros e um inteiro calcula duas listas: uma com todos os elementos da lista que são múltiplos do inteiro dado, e a outra com os restantes elementos.

```
extractMultiples l n = ([x | x<-l, mod x n == 0], [x | x<-l, mod x n /= 0])
```

Apresente uma definição alternativa (recursiva) da função que não use definições de listas por compreensão e que percorra a lista uma única vez.

44. Considere o seguinte tipo para representar valores opcionais:

```
data Maybe a = Nothing | Just a
```

Defina a função `catMaybes :: [Maybe a] -> [a]` que extrai todo o conteúdo da lista de entrada.

45. Considere o seguinte tipo para representar árvores com valores nas folhas:

```
data Tree a = Leaf a | Fork (Tree a) (Tree a)
```

(a) Defina a função `folhas :: Tree a -> [a]` que devolve todas as folhas da árvore.

(b) Defina a função `altura :: Tree a -> Int` que calcula a altura de uma árvore.

46. Considere o seguinte tipo para representar números naturais:

```
data Nat = Zero | Succ Nat
```

Defina a função `toInt :: Nat -> Int` que converte um natural para o inteiro correspondente.

47. Relembre a definição da função `lookup :: (Eq a) => [(a,b)] -> a -> Maybe b`.

```
lookup _ [] = Nothing
lookup x ((a,b):t)
    | x == a    = Just b
    | otherwise = lookup x t
```

Defina a função `lookupSort :: (Ord a) => [(a,b)] -> a -> Maybe b` que, dando o mesmo resultado da função `lookup`, assume (e tira partido de) que a lista está ordenada por ordem crescente (na primeira componente).

48. Considere a seguinte definição do tipo das árvores binárias.

```
data BTree a = Vazia | Nodo a (BTree a) (BTree a)
```

(a) Defina uma função `lookupBST :: (Ord a) => (BTree (a,b)) -> a -> Maybe b` que faz a procura numa árvore binária de procura (ordenada pela primeira componente).

(b) Defina uma função `lookupBT :: (Eq a) => (BTree (a,b)) -> a -> Maybe b` que faz a procura numa árvore binária não necessariamente de procura.

(c) Relembre a definição da função `map :: (a -> b) -> [a] -> [b]`.
Defina uma função semelhante para árvores binárias, i.e., a função `mapBT: (a->b) -> (BTree a) -> BTree b`.

(d) Defina uma função `deProcura :: (Ord a) => (BTree a) -> Bool` que testa se uma árvore binária é de procura.

(e) Defina uma função `eHeap :: (Ord a) => (BTree a) -> Bool` que testa se uma árvore binária é uma *heap* (uma **heap** é uma árvore em que a raiz é o menor elemento da árvore e as suas su-bárvores também são heaps).

(f) Defina uma função `balanceada :: (BTree a) -> Bool` que testa se uma árvore binária é balanceada (uma árvore é balanceada sse a diferença de altura entre as suas sub-árvores é no máximo 1, e essas sub-árvores também são balanceadas).

(g) Defina uma função `equilibrada:: (BTree a) -> Bool` que testa se uma árvore binária é equilibrada (uma árvore é **equilibrada** sse a diferença de número de elementos entre as suas sub-árvores é no máximo 1, e essas sub-árvores também são equilibradas).