Programação Funcional/Paradigmas da Programação I
1° Ano – LEI/LCC/LESI

Exame da 2ª Chamada

30 de Janeiro de 2007 – Duração: 2 horas

Parte I

Esta parte do exame representa 12 valores da cotação total. Cada uma das (sub-)alíneas está cotada em 2 valores.
A não obtenção de uma classificação mínima de 8 valores nesta parte implica a reprovação no exame.

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

   (b) Considere a seguinte função:

```
golosMarcados :: Jornada -> Int
golosMarcados j = sum (map soma) j
```

   Apresente uma Definição para a função `soma` de forma a que a função `golosMarcados` calcule o núnero total de golos marcados numa jornada, e reescreva a função `golosMarcados` sem utilizar as funções `sum` e `map`, e utilizando recursividade primitiva.

2. 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.

3. Considere a seguinte definição de um tipo de dados polimórfico para árvores binárias:

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

   Defina a função `folhas :: ArvBin a -> Int` que conta quantas folhas uma árvore. (Chamam-se folhas aos nódos que têm as duas sub-árvores vazias.)

4. Defina a função `ocorr :: String -> [String] -> [Int]` que, dada uma palavra e um texto (representado como uma lista de strings/palavras), devolve a lista com as posições em que essa palavra ocorre no texto. Por exemplo:

```
Prelude> ocorr "isto" ["isto","serve","para","ver","como","isto","funciona"]
[1,6]
```

5. Considere as seguintes definições:

```
type ListaCompras = [(Produto,Quantidade)]
type Produto = (Nome,PrecoKg)
type Nome = String
type PrecoKg = Float
type Quantidade = Float
```

   Escreva uma função `verificaSock :: ListaCompras -> ListaCompras -> Bool` que, dada uma lista de compras de um cliente e um valor do mesmo tipo representando as quantidades em *stock*, retorne um valor que indique se o pedido pode ou não ser satisfeito. Não assuma quaisquer pressupostos sobre o conteúdo das listas ou sobre a ordenação dos seus elementos.

Programação Funcional/Paradigmas da Programação I
1º Ano – LEI/LCC/LESI

Exame da 2ª Chamada

30 de Janeiro de 2007 – Duração: 2 horas

Parte II

1. Apresente uma definição para a função *merge* que funde os elementos de duas listas, ordenadas de forma crescente, numa única lista também ordenada de forma crescente:

```
merge :: (Ord a) => [a] -> [a] -> [a]
merge [1,3,5] [2,3,6]
=> [1,2,3,3,5,6]
```

2. Defina uma função `minsep` que recebe uma lista l e calcula um tuplo com o menor elemento da lista, e duas listas que contêm divididos entre elas os *restantes elementos* de l, por qualquer ordem. Os comprimentos das duas listas devem diferir no máximo numa unidade. A função deverá efectuar uma única travessia da lista.

```
minsep :: (Ord a) => [a] -> (a,[a],[a])
minsep [90,30,60,40,50]
=> (30,[90,60,],[40,50])
minsep [40,50]
=> (40,[50],[])
```

3. Considere o tipo indutivo de árvores binárias:

```
data BTree a = Empty | Node (a, BTree a, BTree a)
```

e a seguinte função que usa a função da alínea anterior:

```
list2btree :: (Ord a) => [a] -> BTree a
list2btree [] = Empty
list2btree l  = let (y,e,d) = minsep l
                in Node(y, list2btree e, list2btree d)
```

Que propriedades (ou *invariantes*) se pode afirmar que possuem todas as árvores construídas por esta função?

4. Defina a função `btree2list` que produz uma lista *ordenada* de forma crescente a partir de uma árvore construída pela função da alínea anterior.

```
btree2list :: (Ord a) => BTree a -> [a]
```

5. Utilizando como funções auxiliares apenas funções definidas nas alíneas anteriores, defina uma função de *ordenação* de listas

```
hsort :: (Ord a) => [a] -> [a]
```