Paradigmas da Programação I / Programação Funcional

ESI / MCC

Ano Lectivo de 2005/2006 (1a Chamada)

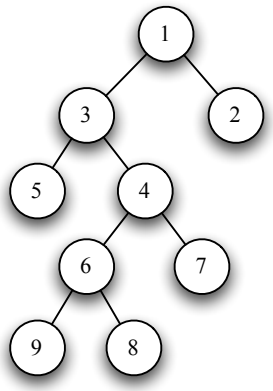
Questão 1 Para implementar fracções vamos usar o seguinte tipo de dados

```
data Frac = F Int Int  -- numerador denominador
```

1. Declare o tipo `Frac` como instância das classes `Eq` e `Ord` (assuma que as comparações são baseadas nos valores numéricos das fracções, i.e., $\frac{2}{3}$ é igual a $\frac{20}{30}$).
2. Defina uma função que simplifica uma fracção. Comece por definir a função `gcd :: Int -> Int -> Int` que calcula o máximo divisor comum entre dois inteiros.
3. Declare o tipo `Frac` como instância da classe `Show`. Tenha em consideração que, se se tratar de um número inteiro deve aparecer apenas o numerador. No outro caso o numerador e o denominador devem aparecer separados por um /.

Questão 2

Dada uma árvore binária, a cada elemento pode ser associado o seu *"caminho"*, que não é mais do que uma lista de Booleanos. Por exemplo, na árvore da direita, temos a seguinte correspondência entre nodos e caminhos:



- O caminho `[False,True,False]` corresponde ao nodo `6`.
- O caminho `[False,True,True]` corresponde ao nodo `7`.
- O caminho `[]` corresponde ao nodo `1` (a raíz da árvore).
- O caminho `[True,True]` não corresponde a nenhum nodo da árvore.

1. Defina uma função `cvalido :: BTree a -> [Bool] -> Bool` que, dada uma árvore e um caminho, diz se esse caminho é válido nessa árvore, i.e., se corresponde a algum nodo da árvore.
2. Defina uma outra função `selecciona :: BTree a -> [Bool] -> a` que, dada uma árvore e um caminho válido nessa árvore, retorna o elemento identificado por esse caminho.
3. Defina uma função `procura :: Eq a => BTree a -> a -> Maybe [Bool]` que, dado um elemento e uma árvore, procura esse elemento na árvore. Em caso de sucesso, a função deverá retornar `Just c` em que `c` é o caminho associado a esse elemento. Em caso de insucesso deve retornar `Nothing`.

Questão 3 Considere que se pretende resolver o seguinte problema.
Existe uma série de *cheques brinde*, cada um com um determinado valor. Para pagar uma determinada quantia pretende-se determinar qual a melhor combinação de cheques a usar. Esta deve ser tal que a soma dos seus valores seja superior ou igual à quantia em causa. Deve-se no entanto minimizar o valor desta soma.
Assumindo que a série de cheques brinde é modelada por uma lista de números, aquilo que se pretende é definir uma função `melhorEscolha :: [Int] -> Int -> ([Int],[Int])` que, dada a sequência de cheques e a quantia separa a sequência original em duas: os cheques a serem usados e os que sobram.

1. Comece por definir uma função `divide:: [Int] -> Int -> ([Int,Int])` que, dada uma lista de inteiros, e um inteiro, divide essa lista em duas de forma a que a soma dos elementos da primeira é superior ou igual ao inteiro dado. Por exemplo, `divide [1,2,3,4,5,6] 9 = ([1,2,3,4],[5,6])`.

2. Defina agora uma função que, dada uma lista com pares destes, calcula aquele cujo somatório da primeira componente é o menor. Defina ainda a função `melhorEscolha` referida acima. Assuma que existe já definida uma função `perms :: [a] -> [[a]]` que calcula, para uma dada lista, a lista de todas as suas permutações (por exemplo, `perms[1,2,3] = [[1,2,3],[2,1,3],[2,3,1],[1,3,2],[3,1,2],[3,2,1]]`).

3. Usando as funções acima, defina um programa `pagamento :: [Int] -> IO ([Int])` que recebe como parâmetro a lista dos valores dos cheques e que pergunta ao utilizador qual a quantia a pagar, imprime a lista dos cheques necessários ao pagamento e devolve a lista com os cheques não usados.