Métodos de Programação I

2.º Ano da LMCC (7003N5) + LESI (5303O7)
Ano Lectivo de 2002/2003

Exame (época normal, 2.ª chamada) — 4 de Fevereiro 2003
14h00
Salas 2201 a 2210

NB: *Esta prova consta de **8** alíneas que valem, cada uma, 2.5 valores. Responda no enunciado, preenchendo a sua identificação em todas as folhas.*

PROVA SEM CONSULTA (2 horas)

Questão 1 Os tipos `Maybe x` e `Either () x` são isomorfos.

1. Apresente em Haskell a definição das funções que testemunham este isomorfismo.
2. Os tipos `(a,Maybe b)` e `Either a (b,a)` são isomorfos. Apresente definições em **C** para estes tipos.
   Use as funções `swap`, `distrl` e as da alínea anterior para definir (em notação *pointfree*) as funções que testemunham esse isomorfismo.

Questão 2 Considere a função seguinte:

```
altern :: (Int, Bool) -> [Char]
altern (0, _) = []
altern (n, b) | b = '+' : altern (n-1, False)
              | otherwise = '-' : altern (n-1, True)
```

1. Defina-a como um *hilomorfismo*.
2. Altere a definição da função `altern` original para uma versão equivalente que possa ser definida como um *anamorfismo* do tipo `[Char]`. Apresente a definição de `altern` como anamorfismo.
3. Pretende-se escrever um *hilomorfismo* `alternPrint :: (Int, Bool) -> IO()` que imprima a sequência de caracteres gerada por `altern`. Para isso escreva um *catamorfismo* `c` tal que `alternPrint = c.altern` seja uma definiçao válida da função desejada.
   N.B: Caso não tenha respondido à alínea anterior poderá utilizar o anamorfismo da alínea (a).

Questão 3 Para captar o tipo das funções que transformam estado definiu-se o seguinte tipo de dados

```
data ST s a = ST { st :: s -> (a,s) }
```

1. Defina (`ST s`) como uma instância da classe `Monad`.
2. Usando o monade de transformação de estados construa uma função para etiquetar uma árvore binária, marcando cada nodo com um número correspondente à ordem porque o nodo é visitado numa travessia *inorder*.

Questão 4 Mostre que a versão "uncurried" da função de concatenação de listas que consta do `Prelude` do Haskell,

```
(++)            :: [a] -> [a] -> [a]
[]     ++ ys     = ys
(x:xs) ++ ys     = x : (xs ++ ys)
```

pode ser convertida num hilomorfismo.
