

# Cálculo de Programas Trabalho Prático MiEI+LCC — 2017/18

Departamento de Informática  
Universidade do Minho

Junho de 2018

Grupo nr.	62
a81451	Alexandre Rodrigues
a82145	Filipa Parente
a81403	Pedro Ferreira

## 1 Preâmbulo

A disciplina de **Cálculo de Programas** tem como objectivo principal ensinar a programação de computadores como uma disciplina científica. Para isso parte-se de um repertório de *combinadores* que formam uma álgebra da programação (conjunto de leis universais e seus corolários) e usam-se esses combinadores para construir programas *composicionalmente*, isto é, agregando programas já existentes.

Na sequência pedagógica dos planos de estudo dos dois cursos que têm esta disciplina, restringe-se a aplicação deste método à programação funcional em **Haskell**. Assim, o presente trabalho prático coloca os alunos perante problemas concretos que deverão ser implementados em **Haskell**. Há ainda um outro objectivo: o de ensinar a documentar programas e a produzir textos técnico-científicos de qualidade.

## 2 Documentação

Para cumprir de forma integrada os objectivos enunciados acima vamos recorrer a uma técnica de programação dita “**literária**” [?], cujo princípio base é o seguinte:

*Um programa e a sua documentação devem coincidir.*

Por outras palavras, o código fonte e a documentação de um programa deverão estar no mesmo ficheiro.

O ficheiro `cp1718t.pdf` que está a ler é já um exemplo de **programação literária**: foi gerado a partir do texto fonte `cp1718t.lhs`<sup>1</sup> que encontrará no **material pedagógico** desta disciplina descompactando o ficheiro `cp1718t.zip` e executando

```
$ lhs2TeX cp1718t.lhs > cp1718t.tex
$ pdflatex cp1718t
```

em que **lhs2tex** é um pre-processor que faz “pretty printing” de código Haskell em **L<sup>A</sup>T<sub>E</sub>X** e que deve desde já instalar executando

```
$ cabal install lhs2tex
```

Por outro lado, o mesmo ficheiro `cp1718t.lhs` é executável e contém o “kit” básico, escrito em **Haskell**, para realizar o trabalho. Basta executar

```
$ ghci cp1718t.lhs
```

Abra o ficheiro `cp1718t.lhs` no seu editor de texto preferido e verifique que assim é: todo o texto que se encontra dentro do ambiente

---

<sup>1</sup>O suffixo ‘lhs’ quer dizer *literate Haskell*.

```
\begin{code}
...
\end{code}
```

vai ser seleccionado pelo **GHCI** para ser executado.

### 3 Como realizar o trabalho

Este trabalho teórico-prático deve ser realizado por grupos de três alunos. Os detalhes da avaliação (datas para submissão do relatório e sua defesa oral) são os que forem publicados na [página da disciplina na internet](#).

Recomenda-se uma abordagem participativa dos membros do grupo de trabalho por forma a poderem responder às questões que serão colocadas na *defesa oral* do relatório.

Em que consiste, então, o *relatório* a que se refere o parágrafo anterior? É a edição do texto que está a ser lido, preenchendo o anexo **C** com as respostas. O relatório deverá conter ainda a identificação dos membros do grupo de trabalho, no local respectivo da folha de rosto.

Para gerar o PDF integral do relatório deve-se ainda correr os comando seguintes, que actualizam a bibliografia (com **BibTeX**) e o índice remissivo (com **makeindex**),

```
$ bibtex cp1718t.aux
$ makeindex cp1718t.idx
```

e recompilar o texto como acima se indicou. Dever-se-á ainda instalar o utilitário **QuickCheck**, que ajuda a validar programas em **Haskell**, a biblioteca **JuicyPixels** para processamento de imagens e a biblioteca **gloss** para geração de gráficos 2D:

```
$ cabal install QuickCheck JuicyPixels gloss
```

Para testar uma propriedade **QuickCheck** *prop*, basta invocá-la com o comando:

```
> quickCheck prop
+++ OK, passed 100 tests.
```

### Problema 1

Segundo uma [notícia do Jornal de Notícias](#), referente ao dia 12 de abril, “*apenas numa hora, foram transacionadas 1.2 mil milhões de dólares em bitcoins. Nas últimas 24 horas, foram transacionados 8,5 mil milhões de dólares, num total de 24 mil milhões de dólares referentes às principais criptomoedas*”.

De facto, é inquestionável que as criptomoedas, e em particular as bitcoin, vieram para ficar. Várias moedas digitais, e em particular as bitcoin, usam a tecnologia de block chain para guardar e assegurar todas as transações relacionadas com a moeda. Uma **block chain** é uma coleção de blocos que registam os movimentos da moeda; a sua definição em Haskell é apresentada de seguida.

```
data Blockchain = Bc { bc :: Block } | Bcs { bcs :: (Block, Blockchain) } deriving Show
```

Cada **bloco** numa block chain regista um número (mágico) único, o momento da execução, e uma lista de transações, tal como no código seguinte:

```
type Block = (MagicNo, (Time, Transactions))
```

Cada **transação** define a entidade de origem da transferência, o valor a ser transacionado, e a entidade destino (por esta ordem), tal como se define de seguida.

```
type Transaction = (Entity, (Value, Entity))
type Transactions = [ Transaction]
```

A partir de uma block chain, é possível calcular o valor que cada entidade detém, tipicamente designado de ledger:

```
type Ledger = [(Entity, Value)]
```

Seguem as restantes definições Haskell para completar o código anterior. Note que *Time* representa o momento da transação, como o número de **milisegundos** que passaram desde 1970.

```
type MagicNo = String
type Time = Int -- em milisegundos
type Entity = String
type Value = Int
```

Neste contexto, implemente as seguintes funções:

1. Defina a função *allTransactions :: Blockchain → Transactions*, como um catamorfismo, que calcula a lista com todas as transações numa dada block chain.

**Propriedade QuickCheck 1** *As transações de uma block chain são as mesmas da block chain revertida:*

$$prop1a = sort \cdot allTransactions \equiv sort \cdot allTransactions \cdot reverseChain$$

*Note que a função sort é usada apenas para facilitar a comparação das listas.*

2. Defina a função *ledger :: Blockchain → Ledger*, utilizando catamorfismos e/ou anamorfismos, que calcula o ledger (i.e., o valor disponível) de cada entidade numa dada block chain. Note que as entidades podem ter valores negativos; de facto isso acontecerá para a primeira transação que executarem.

**Propriedade QuickCheck 2** *O tamanho do ledger é inferior ou igual a duas vezes o tamanho de todas as transações:*

$$prop1b = length \cdot ledger \leq (2*) \cdot length \cdot allTransactions$$

**Propriedade QuickCheck 3** *O ledger de uma block chain é igual ao ledger da sua inversa:*

$$prop1c = sort \cdot ledger \equiv sort \cdot ledger \cdot reverseChain$$

3. Defina a função *isValidMagicNr :: Blockchain → Bool*, utilizando catamorfismos e/ou anamorfismos, que verifica se todos os números mágicos numa dada block chain são únicos.

**Propriedade QuickCheck 4** *A concatenação de uma block chain com ela mesma nunca é válida em termos de números mágicos:*

$$prop1d = \neg \cdot isValidMagicNr \cdot concChain \cdot \langle id, id \rangle$$

**Propriedade QuickCheck 5** *Se uma block chain é válida em termos de números mágicos, então a sua inversa também o é:*

$$prop1e = isValidMagicNr \Rightarrow isValidMagicNr \cdot reverseChain$$

## Problema 2

Uma estrutura de dados frequentemente utilizada para representação e processamento de imagens de forma eficiente são as denominadas **quadrees**. Uma *quadtree* é uma árvore quaternária em que cada nodo tem quatro sub-árvores e cada folha representa um valor bi-dimensional.

```
data QTree a = Cell a Int Int | Block (QTree a) (QTree a) (QTree a) (QTree a)
deriving (Eq, Show)
```

( 0 0 0 0 0 0 0 0 )	Block
( 0 0 0 0 0 0 0 0 )	(Cell 0 4 4) (Block
( 0 0 0 0 1 1 1 0 )	(Cell 0 2 2) (Cell 0 2 2) (Cell 1 2 2) (Block
( 0 0 0 0 1 1 0 0 )	(Cell 1 1 1) (Cell 0 1 1) (Cell 0 1 1) (Cell 0 1 1)))
( 1 1 1 1 1 1 0 0 )	(Cell 1 4 4)
( 1 1 1 1 1 1 0 0 )	(Block
( 1 1 1 1 0 0 0 0 )	(Cell 1 2 2) (Cell 0 2 2) (Cell 0 2 2) (Block
( 1 1 1 1 0 0 0 1 )	(Cell 0 1 1) (Cell 0 1 1) (Cell 0 1 1) (Cell 1 1 1)))

(a) Matriz de exemplo *bm*.

(b) Quadtree de exemplo *qt*.

Figura 1: Exemplos de representações de bitmaps.

Uma imagem monocromática em formato bitmap pode ser representada como uma matriz de bits<sup>2</sup>, tal como se exemplifica na Figura 1a.

O anamorfismo *bm2qt* converte um bitmap em forma matricial na sua codificação eficiente em quad-trees, e o catamorfismo *qt2bm* executa a operação inversa:

$bm2qt :: (Eq\ a) \Rightarrow Matrix\ a \rightarrow QTree\ a$	$qt2bm :: (Eq\ a) \Rightarrow QTree\ a \rightarrow Matrix\ a$
$bm2qt = anaQTree\ f\ \text{where}$	$qt2bm = cataQTree\ [f, g]\ \text{where}$
$f\ m = \text{if one then } i_1\ u\ \text{else } i_2\ (a, (b, (c, d)))$	$f\ (k, (i, j)) = matrix\ j\ i\ k$
$\text{where } x = (nub \cdot toList)\ m$	$g\ (a, (b, (c, d))) = (a \uparrow b) \leftrightarrow (c \uparrow d)$
$u = (head\ x, (ncols\ m, nrows\ m))$	
$one = (ncols\ m \equiv 1 \vee nrows\ m \equiv 1 \vee length\ x \equiv 1)$	
$(a, b, c, d) = splitBlocks\ (nrows\ m \div 2)\ (ncols\ m \div 2)\ m$	

O algoritmo *bm2qt* particiona recursivamente a imagem em 4 blocos e termina produzindo folhas para matrizes unitárias ou quando todos os píxeis de um sub-bloco têm a mesma cor. Para a matriz *bm* de exemplo, a quadtree correspondente *qt* = *bm2qt bm* é ilustrada na Figura 1b.

Imagens a cores podem ser representadas como matrizes de píxeis segundo o código de cores *RGBA*, codificado no tipo *PixelRGBA8* em que cada pixel é um quádruplo de valores inteiros (*red, green, blue, alpha*) contidos entre 0 e 255. Atente em alguns exemplos de cores:

```
whitePx = PixelRGBA8 255 255 255 255
blackPx = PixelRGBA8 0 0 0 255
redPx   = PixelRGBA8 255 0 0 255
```

O módulo *BMP*, disponibilizado juntamente com o enunciado, fornece funções para processar ficheiros de imagem bitmap como matrizes:

```
readBMP :: FilePath → IO (Matrix PixelRGBA8)
writeBMP :: FilePath → Matrix PixelRGBA8 → IO ()
```

Teste, por exemplo, no *GHCi*, carregar a Figura 2a:

```
> readBMP "cp1718t_media/person.bmp"
```

Esta questão aborda operações de processamento de imagens utilizando quadrees:

1. Defina as funções *rotateQTree* :: *QTree a* → *QTree a*, *scaleQTree* :: *Int* → *QTree a* → *QTree a* e *invertQTree* :: *QTree a* → *QTree a*, como catamorfismos e/ou anamorfismos, que rodam<sup>3</sup>, re-dimensionam<sup>4</sup> e invertem as cores de uma quadtree<sup>5</sup>, respectivamente. Tente produzir imagens similares às Figuras 2b, 2c e 2d:

```
> rotateBMP "cp1718t_media/person.bmp" "person90.bmp"
> scaleBMP 2 "cp1718t_media/person.bmp" "personx2.bmp"
> invertBMP "cp1718t_media/person.bmp" "personinv.bmp"
```

<sup>2</sup>Cf. módulo *Data.Matrix*.

<sup>3</sup>Segundo um ângulo de 90° no sentido dos ponteiros do relógio.

<sup>4</sup>Multiplicando o seu tamanho pelo valor recebido.

<sup>5</sup>Um pixel pode ser invertido calculando  $255 - c$  para cada componente *c* de cor RGB, exceptuando o componente alpha.



(a) Bitmap de exemplo.



(b) Rotação.



(c) Redimensionamento.



(d) Inversão de cores.



(e) Compressão de 1 nível.



(f) Compressão de 2 níveis.



(g) Compressão de 3 níveis.



(h) Compressão de 4 níveis.



(i) Bitmap de contorno.



(j) Bitmap com contorno.

Figura 2: Manipulação de uma figura bitmap utilizando quadrees.

**Propriedade QuickCheck 6** Rodar uma quadtree é equivalente a rodar a matriz correspondente:

$$\text{prop2c} = \text{rotateMatrix} \cdot \text{qt2bm} \equiv \text{qt2bm} \cdot \text{rotateQTree}$$

**Propriedade QuickCheck 7** Redimensionar uma imagem altera o seu tamanho na mesma proporção:

$$\text{prop2d} (\text{Nat } s) = \text{sizeQTree} \cdot \text{scaleQTree } s \equiv ((s*) \times (s*)) \cdot \text{sizeQTree}$$

**Propriedade QuickCheck 8** Inverter as cores de uma quadtree preserva a sua estrutura:

$$\text{prop2e} = \text{shapeQTree} \cdot \text{invertQTree} \equiv \text{shapeQTree}$$

2. Defina a função  $\text{compressQTree} :: \text{Int} \rightarrow \text{QTree } a \rightarrow \text{QTree } a$ , utilizando catamorfismos e/ou anamorfismos, que comprime uma quadtree cortando folhas da árvore para reduzir a sua profundidade num dado número de níveis. Tente produzir imagens similares (mas não necessariamente iguais) às Figuras 2e, 2f, 2g e 2h:

```
> compressBMP 1 "cp1718t_media/person.bmp" "person1.bmp"
> compressBMP 2 "cp1718t_media/person.bmp" "person2.bmp"
> compressBMP 3 "cp1718t_media/person.bmp" "person3.bmp"
> compressBMP 4 "cp1718t_media/person.bmp" "person4.bmp"
```

**Propriedade QuickCheck 9** A quadtree comprimida tem profundidade igual à da quadtree original menos a taxa de compressão:

$$\text{prop2f} (\text{Nat } n) = \text{depthQTree} \cdot \text{compressQTree } n \equiv (-n) \cdot \text{depthQTree}$$

3. Defina a função  $\text{outlineQTree} :: (a \rightarrow \text{Bool}) \rightarrow \text{QTree } a \rightarrow \text{Matrix Bool}$ , utilizando catamorfismos e/ou anamorfismos, que recebe uma função que determina quais os píxeis de fundo e converte uma quadtree numa matriz monocromática, de forma a desenhar o contorno de uma **malha poligonal** contida na imagem. Tente produzir imagens similares (mas não necessariamente iguais) às Figuras 2i e 2j:

```
> outlineBMP "cp1718t_media/person.bmp" "personOut1.bmp"
> addOutlineBMP "cp1718t_media/person.bmp" "personOut2.bmp"
```

**Propriedade QuickCheck 10** A matriz de contorno tem dimensões iguais às da quadtree:

$$\text{prop2g} = \text{sizeQTree} \equiv \text{sizeMatrix} \cdot \text{outlineQTree} (<0)$$

**Teste unitário 1** Contorno da quadtree de exemplo qt:

$$\text{teste2a} = \text{outlineQTree} (\equiv 0) \text{ qt} \equiv \text{qtOut}$$

## Problema 3

O cálculo das combinações de  $n$   $k$ -a- $k$ ,

$$\binom{n}{k} = \frac{n!}{k! * (n - k)!} \quad (1)$$

envolve três factoriais. Recorrendo à **lei de recursividade múltipla** do cálculo de programas, é possível escrever o mesmo programa como um simples ciclo-for onde se fazem apenas multiplicações e somas. Para isso, começa-se por estruturar a definição dada da forma seguinte,

$$\binom{n}{k} = h \ k \ (n - k)$$



Figura 3: Passos de construção de uma árvore de Pitágoras de ordem 3.

onde

$$h \ k \ d = \frac{f \ k \ d}{g \ d}$$

$$f \ k \ d = \frac{(d+k)!}{k!}$$

$$g \ d = d!$$

assumindo-se  $d = n - k \geq 0$ . É fácil de ver que  $f \ k$  e  $g$  se desdobram em 4 funções mutuamente recursivas, a saber

$$f \ k \ 0 = 1$$

$$f \ k \ (d+1) = \underbrace{(d+k+1)}_{l \ k \ d} * f \ k \ d$$

$$l \ k \ 0 = k+1$$

$$l \ k \ (d+1) = l \ k \ d + 1$$

e

$$g \ 0 = 1$$

$$g \ (d+1) = \underbrace{(d+1)}_{s \ d} * g \ d$$

$$s \ 0 = 1$$

$$s \ (d+1) = s \ d + 1$$

A partir daqui alguém derivou a seguinte implementação:

$$\binom{n}{k} = h \ k \ (n-k) \text{ where } h \ k \ n = \text{let } (a, -, b, -) = \text{for loop (base k) n in } a / b$$

Aplicando a lei da recursividade múltipla para  $\langle f \ k, l \ k \rangle$  e para  $\langle g, s \rangle$  e combinando os resultados com a [lei de banana-split](#), derive as funções *base k* e *loop* que são usadas como auxiliares acima.

**Propriedade QuickCheck 11** Verificação que  $\binom{n}{k}$  coincide com a sua especificação (1):

$$\text{prop3 } (NonNegative \ n) \ (NonNegative \ k) = k \leq n \Rightarrow \binom{n}{k} \equiv n! / (k! * (n-k)!)$$

## Problema 4

**Fractais** são formas geométricas que podem ser construídas recursivamente de acordo com um conjunto de equações matemáticas. Um exemplo clássico de um fractal são as **árvores de Pitágoras**. A construção de uma árvore de Pitágoras começa com um quadrado, ao qual se unem dois quadrados redimensionados pela escala  $\sqrt{2}/2$ , de forma a que os cantos dos 3 quadrados coincidam e formem um triângulo rectângulo isósceles. Este procedimento é repetido recursivamente de acordo com uma dada ordem, definida como um número natural (Figura 3).

Uma árvore de Pitágoras pode ser codificada em Haskell como uma *full tree* contendo quadrados nos nodos e nas folhas, sendo um quadrado definido simplesmente pelo tamanho do seu lado:

```
data FTree a b = Unit b | Comp a (FTree a b) (FTree a b) deriving (Eq, Show)
type PTree = FTree Square Square
type Square = Float
```

1. Defina a função `generatePTree :: Int → PTree`, como um anamorfismo, que gera uma árvore de Pitágoras para uma dada ordem.

**Propriedade QuickCheck 12** Uma árvore de Pitágoras tem profundidade igual à sua ordem:

$$\text{prop4a } (\text{SmallNat } n) = (\text{depthFTree} \cdot \text{generatePTree}) \, n \equiv n$$

**Propriedade QuickCheck 13** Uma árvore de Pitágoras está sempre balanceada:

$$\text{prop4b } (\text{SmallNat } n) = (\text{isBalancedFTree} \cdot \text{generatePTree}) \, n$$

2. Defina a função `drawPTree :: PTree → [Picture]`, utilizando catamorfismos e/ou anamorfismos, que anima incrementalmente os passos de construção de uma árvore de Pitágoras recorrendo à biblioteca `gloss`. Anime a sua solução:

```
> animatePTree 3
```

## Problema 5

Uma das áreas em maior expansão no campo da informática é a análise de dados e `machine learning`. Esta questão aborda um *mónade* que ajuda a fazer, de forma simples, as operações básicas dessas técnicas. Esse *mónade* é conhecido por *bag*, *saco* ou *multi-conjunto*, permitindo que os elementos de um conjunto tenham multiplicidades associadas. Por exemplo, seja

```
data Marble = Red | Pink | Green | Blue | White deriving (Read, Show, Eq, Ord)
```

um tipo dado.<sup>6</sup> A lista `[Pink, Green, Red, Blue, Green, Red, Green, Pink, Blue, White]` tem elementos repetidos. Assumindo que a ordem não é importante, essa lista corresponde ao saco

```
{ Red |-> 2 , Pink |-> 2 , Green |-> 3 , Blue |-> 2 , White |-> 1 }
```

que habita o tipo genérico dos “bags”:

```
data Bag a = B [(a, Int)] deriving (Ord)
```

O *mónade* que vamos construir sobre este tipo de dados faz a gestão automática das multiplicidades. Por exemplo, seja dada a função que dá o peso de cada berlinde em gramas:

```
marbleWeight :: Marble → Int
marbleWeight Red   = 3
marbleWeight Pink  = 2
marbleWeight Green = 3
marbleWeight Blue  = 6
marbleWeight White = 2
```

Então, se quisermos saber quantos *berlindes* temos, de cada *peso*, não teremos que fazer contas: basta calcular

```
marbleWeights = fmap marbleWeight bagOfMarbles
```

onde `bagOfMarbles` é o saco de berlindes referido acima, obtendo-se:

```
{ 2 |-> 3 , 3 |-> 5 , 6 |-> 2 }.
```

---

<sup>6</sup>“Marble”traduz para “berlinde”em português.





Figura 4: Distribuição de berlindes num saco.

Mais ainda, se quisermos saber o total de berlindes em *bagOfMarbles* basta calcular `fmap (!) bagOfMarbles` obtendo-se `{ () | -> 10 }`; isto é, o saco tem 10 berlindes no total.

Finalmente, se quisermos saber a probabilidade da cor de um berlinde que tiremos do saco, basta converter o referido saco numa distribuição correndo:

```
marblesDist = dist bagOfMarbles
```

obtendo-se a distribuição (graças ao módulo *Probability*):

```
Green  30.0%
Red    20.0%
Pink   20.0%
Blue   20.0%
White  10.0%
```

cf. Figura 4.

Partindo da seguinte declaração de *Bag* como um functor e como um mónade,

```
instance Functor Bag where
  fmap f = B · map (f × id) · unB
instance Monad Bag where
  x >>= f = (μ · fmap f) x where
    return = singletonbag
```

1. Defina a função  $\mu$  (multiplicação do mónade *Bag*) e a função auxiliar *singletonbag*.
2. Verifique-as com os seguintes testes unitários:

**Teste unitário 2** *Lei*  $\mu \cdot \text{return} = \text{id}$ :

$$\text{test5a} = \text{bagOfMarbles} \equiv \mu (\text{return bagOfMarbles})$$

**Teste unitário 3** *Lei*  $\mu \cdot \mu = \mu \cdot \text{fmap } \mu$ :

$$\text{test5b} = (\mu \cdot \mu) \, b3 \equiv (\mu \cdot \text{fmap } \mu) \, b3$$

onde *b3* é um saco dado em anexo.

# Anexos

## A Mónade para probabilidades e estatística

Mónades são funtores com propriedades adicionais que nos permitem obter efeitos especiais em programação. Por exemplo, a biblioteca **Probability** oferece um mónade para abordar problemas de probabilidades. Nesta biblioteca, o conceito de distribuição estatística é captado pelo tipo

$$\text{newtype Dist } a = D \{ \text{unD} :: [(a, \text{ProbRep})] \} \quad (2)$$

em que *ProbRep* é um real de 0 a 1, equivalente a uma escala de 0 a 100/.

Cada par  $(a, p)$  numa distribuição  $d :: \text{Dist } a$  indica que a probabilidade de  $a$  é  $p$ , devendo ser garantida a propriedade de que todas as probabilidades de  $d$  somam 100/. Por exemplo, a seguinte distribuição de classificações por escalões de  $A$  a  $E$ ,

$A$	■	2%
$B$	■	12%
$C$	■	29%
$D$	■	35%
$E$	■	22%

será representada pela distribuição

$$\begin{aligned} d1 &:: \text{Dist Char} \\ d1 &= D [( 'A', 0.02), ( 'B', 0.12), ( 'C', 0.29), ( 'D', 0.35), ( 'E', 0.22)] \end{aligned}$$

que o **GHCI** mostrará assim:

```
'D' 35.0%
'C' 29.0%
'E' 22.0%
'B' 12.0%
'A'  2.0%
```

Este mónade é adequado à resolução de problemas de *probabilidades e estatística* usando programação funcional, de forma elegante e como caso particular de programação monádica.

## B Definições auxiliares

Funções para mostrar *bags*:

```
instance (Show a, Ord a, Eq a) => Show (Bag a) where
  show = showbag . consol . unB where
    showbag = concat .
      ( ++ [ " } " ] ) . ( " { " : ) .
      ( intersperse " , " ) .
      sort .
      ( map f ) where f (a, b) = (show a) ++ " | -> " ++ (show b)
    unB (B x) = x
```

Igualdade de *bags*:

```
instance (Eq a) => Eq (Bag a) where
  b == b' = (unB b) 'lequal' (unB b')
  where lequal a b = isempty (a ⊖ b)
        ominus a b = a ++ neg b
        neg x = [(k, -i) | (k, i) <- x]
```

Ainda sobre o mónade *Bag*: