

Cálculo de Programas

Trabalho Prático

MiEI+LCC — 2017/18

Departamento de Informática
Universidade do Minho

Junho de 2018

Grupo nr.	82 (preencher)
a73700	Sara Alexandra da Silva Pereira
a74155	Bruno Filipe da Silva Ferreira
a74813	André Filipe de Araújo Pereira de Sousa

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`¹ 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^AT_EX** 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

¹O 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:*

$$\text{prop1a} = \text{sort} \cdot \text{allTransactions} \equiv \text{sort} \cdot \text{allTransactions} \cdot \text{reverseChain}$$

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:*

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

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

$$\text{prop1c} = \text{sort} \cdot \text{ledger} \equiv \text{sort} \cdot \text{ledger} \cdot \text{reverseChain}$$

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:*

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

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

$$\text{prop1e} = \text{isValidMagicNr} \Rightarrow \text{isValidMagicNr} \cdot \text{reverseChain}$$

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 bits², 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:

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

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 rodam³, re-dimensionam⁴ e invertem as cores de uma quadtree⁵, 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"

```

²Cf. módulo *Data.Matrix*.

³Segundo um ângulo de 90° no sentido dos ponteiros do relógio.

⁴Multiplicando o seu tamanho pelo valor recebido.

⁵Um 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.



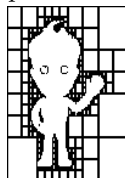
(f) Compressão de 2 níveis.



(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.⁶ 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 }.
```

⁶“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 μ (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* $\mu \cdot \mu = \mu \cdot \text{fmap } \mu$:

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

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*:

```

instance Applicative Bag where
  pure = return
  (< * >) = aap

```

O exemplo do texto:

```

bagOfMarbles = B [(Pink, 2), (Green, 3), (Red, 2), (Blue, 2), (White, 1)]

```

Um valor para teste (bags de bags de bags):

```

b3 :: Bag (Bag (Bag Marble))
b3 = B [(B [(B [(Pink, 2), (Green, 3), (Red, 2), (Blue, 2), (White, 1)], 5)
, (B [(Pink, 1), (Green, 2), (Red, 1), (Blue, 1)], 2)], 2)]

```

Outras funções auxiliares:

```

a ↦ b = (a, b)
consol :: (Eq b) ⇒ [(b, Int)] → [(b, Int)]
consol = filter nzero · map (id × sum) · col where nzero (_, x) = x ≠ 0
isempty :: Eq a ⇒ [(a, Int)] → Bool
isempty = all (≡ 0) · map π₂ · consol
col x = nub [k ↦ [d' | (k', d') ← x, k' ≡ k] | (k, d) ← x]
consolidate :: Eq a ⇒ Bag a → Bag a
consolidate = B · consol · unB

```

C Soluções dos alunos

Os alunos devem colocar neste anexo as suas soluções aos exercícios propostos, de acordo com o “layout” que se fornece. Não podem ser alterados os nomes ou tipos das funções dadas, mas pode ser adicionado texto e / ou outras funções auxiliares que sejam necessárias.

Problema 1

Sabendo o tipo da nossa estrutura podemos verificar que este ou contém um Block, denominado Bc , ou contém um tuplo com um Block e um BlockChain, sendo o tuplo denominado Bcs. Dessa maneira, dado um input, este será de um tipo ou do outro.

```

inBlockchain = [Bc, Bcs]

```

No caso do out queremos realizar a operação contrária, ou seja, criar a partir de uma BlockChain, um tipo que tanto pode ser um Bc ou um Bcs. Assim, queremos aplicar à nossa BlockChain a projeção à esquerda no caso desta ser um Bc ou a projeção à direita no caso contrário.

```

outBlockchain (Bc bc) = i₁ (bc)
outBlockchain (Bcs (bc, a)) = i₂ (bc, a)

```

Ao definir a função rec estamos a definir a função que trabalha sobre a soma que temos presente. Sendo dada uma função de transformação, esta será aplicada à parte do tipo que contem a informação do próximo elemento, neste caso sendo o BlockChain, presente no tuplo Bcs. Como verificado antes, temos um tipo que pode ser de dois tipos diferentes, assim para transformar esta soma é aplicado o co-produto na mesma, tendo assim criada a probabilidade de ser cada um dos tipos da soma. No entanto no lado direito da soma, temos presente um tipo que é um tuplo entre o tipo Bc e o tipo Bcs. Para tal é aplicado o produto neste tipo que transformaria o nosso tuplo no tuplo desejado.

```

recBlockchain f = id + id × f

```

A função cata permite transformar um dado tipo de dados usando uma função que realiza esta transformação. Essa função de transformação permite transformar uma dada soma num tipo de dados unico. Para poder propagar a função pelo tipo é usada a função rec definida anteriormente, no entanto,

o tipo de dados recebido, Blockchain, não corresponde ao tipo recebido pela nossa rec. Para poder ultrapassar este problema é aplicada a função out definida, obtendo assim o tipo de dados pretendidos, e só após a função rec. A função que será aplicada a cada elemento é a própria função cata tendo como função de transformação a anterior, para que a mesma seja aplicada a cada elemento. Tendo feita a recursividade, podemos aplicar a função de transformação ao elemento atual.

$$cataBlockchain f = f \cdot recBlockchain (cataBlockchain f) \cdot outBlockchain$$

A função ana pretende fazer o contrário, sendo a mesma aplicada inicialmente ao tipo único recebido, permitindo transformar o tipo recebido numa soma. Depois da mesma ser aplicada iríamos realizar a recursividade, da mesma maneira feita no cata, usando a própria função ana e a função de transformação. Para depois poder obter o tipo de dados pretendido, é realizada a função in, que permite transformar a nossa soma num tipo de dados concreto.

$$anaBlockchain f = inBlockchain \cdot recBlockchain (anaBlockchain f) \cdot f$$

$$hyloBlockchain f g = cataBlockchain f \cdot anaBlockchain g$$

De seguida é apresentada a definição da função allTransactions, que permite obter a lista de Transações de uma dada blockchain. Para tal é aplicada a função cata à Blockchain sendo passada a função transformadora. Esta função é função de transformação da soma, sendo do primeiro lado aplicado a projeção 2, que obtem o tuplo (Time, Transactions) do block e após isso sendo a aplicada de novo a projeção 2 para se obter as Transactions. Do segundo lado da soma temos de aplicar a concatenação das Transactions de um elemento, sendo o mesmo obtido da mesma forma, com as Transactions já obtidas.

DIAGRAMA

$$\begin{aligned} allTransactions &= (cataBlockchain get_transaction) \\ \textbf{where } get_transaction &= [\pi_2 \cdot \pi_2, conc \cdot ((\pi_2 \cdot \pi_2) \times id)] \end{aligned}$$

Na função ledger pretendemos obter o Value de Cada Entity segundo uma dada Blockchain. Para tal é aplicada a função allTransactions, para obter a lista de todas as Transaction da Blockchain. Após, seria aplicada uma cata à lista de transactions, permitindo transformar os tuplos (Entity1, (Value, Entity2)) em ((Entity1, -Value), (Entity2, Value)). Sabendo que as listas não são de um tipo concreto, é usada a função either para identificar os dois casos. No caso de lista vazia, teremos também uma lista vazia. No outro caso teremos de transformar o nosso tuplo e concatenar o mesmo com os restantes. Para criar o nosso tuplo é aplicado um split ao mesmo, permitindo assim criar um tuplo de tuplos. A primeira função do split será um produto entre o id de Entity1 e o negate aplicado após a projeção 2 do tuplo presente dentro do tuplo inicial. Este negate serve para guardar a informação do Entity 1, receptor da transação. No caso da segunda função do split é aplicada a projeção 2 do tuplo criado apartir do produto de id com swap. Este swap permite obter a informação de forma a ser guardada pelo ledger.

Após o primeiro cata, é necessário criar a lista que guardaria os tuplos presentes em cada projeção dos tuplos da lista anterior. Para isto é aplicado outro cata à lista, que permitira transformar cada elemento de maneira a que a informação (Entity, Value) esteja presente na mesma. No caso de paragem novamente usado a lista vazia, e no outro caso como anteriormente temos de adicionar a informação de um elemento aos já presentes. Para isso é criada uma lista entre os dois elementos do tuplo sendo essa depois concatenada com a informação restante. Para poder agrupar a informação das várias Entity é usada a função col, permitindo agrupar os nossos tuplos como (Entity, [Value]).

Após isso é apenas necessário adicionar a informação presente na lista de Value para obter o Ledger pretendido. Para isso é utilizada uma cata que transforma apenas a segunda parte do tuplo presente.

$$\begin{aligned} ledger &= (cataList h) \cdot col \cdot (cataList g) \cdot (cataList f) \cdot allTransactions \\ \textbf{where } f &= [nil, cons \cdot (((id \times negate \cdot \pi_2, \pi_2) \cdot (id \times swap)) \times id)] \\ g &= [nil, conc \cdot (conc \cdot ((singl \times singl)) \times id)] \\ h &= [nil, cons \cdot ((id \times sum) \times id)] \end{aligned}$$

Para verificar a validade dos Magic Numbers é necessário primeiramente obter uma lista de todos os Magic Numbers. Para isso é aplicada uma cata à Blockchain, que transformaria o primeiro elemento da soma num Magic Number e no segundo adicionava a sua informação à lista já presente. Após isso, duplicávamos a lista obtida, tendo um tuplo com a mesma lista. Depois disso, aplicávamos o produto ao

tuplo, tendo de um lado o tamanho da lista e o outro o tamanho da lista depois de remover os repetidos de uma lista. Apos isso, comparavamos os dois elementos do tuplo, usando a função uncurry (==)

$$\begin{aligned} isValidMagicNr &= (\widehat{=}) \cdot (\text{length} \times (\text{length} \cdot \text{nub})) \cdot \text{dup} \cdot \text{cataBlockchain} \, f \\ \text{where } f &= [\text{singl} \cdot \pi_1, \text{cons} \cdot (\pi_1 \times \text{id})] \end{aligned}$$

Problema 2

Tal como definido anteriormente temos para o tipo de dados pedido as funções básicas que trabalham sobre o mesmo. Estas seguem a mesma linha de pensamento, fazendo adaptações à estrutura presente agora.

$$\begin{aligned} inQTree \, (i_1 \, (a, (b, c))) &= \text{Cell} \, a \, b \, c \\ inQTree \, (i_2 \, (a, (b, (c, d)))) &= \text{Block} \, a \, b \, c \, d \end{aligned}$$

$$\begin{aligned} outQTree \, (\text{Cell} \, a \, b \, c) &= i_1 \, (a, (b, c)) \\ outQTree \, (\text{Block} \, a \, b \, c \, d) &= i_2 \, (a, (b, (c, d))) \end{aligned}$$

No caso da base deste tipo, são passadas duas funções, a função que é aplicada no tipo a da QTree e a função que será aplicada a cada uma das QTree presentes no block.

$$\text{baseQTree} \, g \, h = (g \times \text{id}) + (h \times (h \times (h \times h)))$$

$$\text{recQTree} \, f = \text{baseQTree} \, \text{id} \, f$$

$$\text{cataQTree} \, f = f \cdot \text{recQTree} \, (\text{cataQTree} \, f) \cdot \text{outQTree}$$

$$\text{anaQTree} \, f = \text{inQTree} \cdot \text{recQTree} \, (\text{anaQTree} \, f) \cdot f$$

$$\text{hyloQTree} \, f \, g = \text{cataQTree} \, f \cdot \text{anaQTree} \, g$$

instance Functor QTree where

$$\text{fmap} \, \text{gen} = \text{cataQTree} \, (\text{inQTree} \cdot \text{baseQTree} \, \text{gen} \, \text{id})$$

O rotate pretende rodar a Qtree existente em 90 ° graus. Para isso é necessário alterar o tamanho presente nas Cells. Para realizar o rotate é aplicada uma cata à Qtree, em que a depois de aplicada uma função f é aplicada a in da Qtree, para poder obter de novo Qtree. Esta função f é definida como uma soma sendo que do primeiro lado seria aplicado o produto entre id e swap, e do segundo lado teríamos de realizar um split para poder transformar o tuplo presente. Do primeiro lado iríamos colocar a projecção 1 após a projecção 2 após a projecção 2. Do segundo lado teríamos de realizar um novo split, para poder criar um tuplo neste local. Este tuplo seria criado com a projecção 1 em conjunto com um novo split. Este seria formado pela projecção 2 após a projecção 2 após a projecção 2 e pela projecção 1 após a projecção 2. Isto é necessário para criar o tuplo com as Qtree presentes no Block.

$$\begin{aligned} \text{rotateQTree} &= \text{cataQTree} \, (\text{inQTree} \cdot f) \quad \text{-- inQTree para converter o either do cata para Qtree} \\ \text{where } f &= g + h \quad \text{-- Co-produto para poder criar um either no fim} \\ g &= \text{id} \times \text{swap} \\ h &= \langle \pi_1 \cdot \pi_2 \cdot \pi_2, s1 \rangle \\ s1 &= \langle \pi_1, s2 \rangle \\ s2 &= \langle \pi_2 \cdot \pi_2 \cdot \pi_2, \pi_1 \cdot \pi_2 \rangle \end{aligned}$$

No caso da scale é necessário multiplicar o tamanho dado pelo tamanho presente em cada Cell. Para isso é usada a função ana. Esta permite que seja criada uma QTree dada uma função de geração de um tipo C, dado como input. Esta função de geração neste caso transforma a QTree usando a função out e aplica-lhe uma função f. Esta função f é definida pela soma entre uma função g e a identidade, visto

apenas queremos alterar a informação nas Cell. Esta função g é definida como um produto entre a identidade e um produto cujas funções são a multiplicação do Integer do Cell pelo Scale dado.

```
scaleQTree i = anaQTree (f · outQTree) -- Out para converter a Qtree em either para a função ana
  where f = g + id
        g = id × ((i*) × (i*))
```

Na função $invert$ queremos apenas alterar a informação presente no tipo a da Cell. Para tal é usada o funtor definido, sendo que este permite aplicar alteração apenas ao tipo da estrutura. A função criadora do novo pixel apenas altera o valor de cada pixel segundo a formula dada $(255 - c)$.

```
-- USAMOS A PixelRGBA8 porque é o monade que é guardado no  $a$  da QTree
invertQTree = fmap invert -- FMAP PARA APLICAR A FUNÇÃO A CADA UMA DAS FOLHAS
  where invert (PixelRGBA8 r g b a) = PixelRGBA8 (255 - r) (255 - g) (255 - b) (a)
```

No caso da $compress$ é criada uma função auxiliar $cataQTree'$ que permite fazer a decremantação do valor passado à $cata$, valor esse que depende da profundidade da Qtree e do valor passado. A função geradora recorre a $p2p$ para fazer a verificação. No caso de se verificar que o nível é menor ou igual a 0, mantemos a estrutura igual, no caso de ser maior realizamos o $prune$. A função $prune$ permite depois criar as novas cell com a informação das anteriores. Isto é feito através de um $cata$, que permite transformar a QTree presente nesse momento num unica Cell, no caso de este não ser já um cell. Na função transformadora é aplicada a $uncell$, função que cria a Cell dado um tuplo, após ser aplicada a $assocl$, sendo esta aplicada após a função f , que realiza a transformação do either para um tipo pretendido. Nesta função é aplicado a identidade do lado esquerdo e do lado direito é realizado um $split$, para poder ter a criação de num novo tuplo. Tuplo esse que a primeira parte é criada apartir da função $colorQTree$ - obtendo a cor - após realizar a projeção 1. No outro lado do tuplo seria adicionada a informação dos sizes da QTree a ser criada.

```
-- Criar um cata que faça a subtração do valor de  $k$ , para poder descer até onde se quer manter a arvore igual
compressQTree k qt = cataQTree' f (depthQTree qt - k) qt
  where f k = p2p (id, pruneQTree) (k ≤ 0) · inQTree
        cataQTree' f k = (f k) · recQTree (cataQTree' f (k - 1)) · outQTree

pruneQTree = cataQTree (uncell · assocl · f)
  where f = [id, ⟨colorQTree · π1, addSizes · getSizes⟩]
        uncell =  $\hat{\cdot} \$ Cell$ 
        addSizes ((x1, y1), (x2, y2)) = (x1 + x2, y1 + y2)
        getSizes = sizeQTree × (sizeQTree · π2 · π2)
        colorQTree = cataQTree $ [π1, π1]
```

No caso da $outline$, o que pretendemos fazer é criar uma Matrix de Bool, dada uma QTree e a função de verificação. Visto que apenas é necessário alterar a informação presente nas folhas da QTree podemos usar a função $fmap$, que permite transformar o tipo presente nas folhas, num novo tipo. Passamos assim a função f , recebida ao $fmap$. Após isso é necessário converter a QTree Bool para Matrix Bool, criando a borda nos casos necessários. Para tal é usada a função $convert$, que recebe uma soma, de um lado trata de

```
convert :: QTree Bool → Matrix Bool
convert = cataQTree (f)
  where f = [g, h]
        g (c, (i, j)) = if (c ∧ (i ≥ 3) ∧ (j ≥ 3)) then line i j else matrix j i c
        h (a, (b, (c, d))) = (a ↓ b) ↔ (c ↓ d)
        line i j = ((matrix 1 i true) ↔ ((matrix (j - 2) 1 true) ↓ ((matrix (j - 2) (i - 2) false) ↓ (matrix (j - 2) 1
outlineQTree f = convert · fmap f
```

Problema 3

Conversão de f k:

$$\begin{aligned}
& \left\{ \begin{array}{l} f \ k \ 0 = 1 \\ f \ k \ (d + 1) = (d + k + 1) * f \ k \ d \end{array} \right. \\
\equiv & \quad \{ \text{Igualdade extensional } \{73\} \times 2, \text{Def-comp } \{74\} \times 2 \} \\
& \left\{ \begin{array}{l} f \ k \cdot \underline{0} = \underline{1} \\ f \ k \cdot \text{succ} =_{mul} \langle f \ k, l \ k \rangle \end{array} \right. \\
\equiv & \quad \{ \text{Eq-+ } \{27\} \times 2, \text{Natural-id } \{1\} \times 2 \}
\end{aligned}$$

$$[f \ k \cdot 0, f \ k \cdot \text{succ}] = [1 \cdot id, mul \cdot \langle l \ k, f \ k \rangle]$$

$$\equiv \quad \{ \text{Fusão-+ } \{20\}, \text{Absorção-+ } \{22\} \}$$

$$f \ k \cdot [0, \text{succ}] = [1, mul] \cdot (id + \langle lk, fk \rangle)$$

□

Conversão l k :

$$\begin{aligned}
& \left\{ \begin{array}{l} l \ k \ 0 = k + 1 \\ l \ k \ (d + 1) = l \ k \ d + 1 \end{array} \right. \\
\equiv & \quad \{ \text{Igualdade extensional } \{73\} \times 2, \text{Def-comp } \{74\} \times 2 \}
\end{aligned}$$

$$\left\{ \begin{array}{l} l \ k \cdot \text{zero} = \text{succ} \\ l \ k \cdot \text{succ} =_{\text{succ}} \cdot l \ k \end{array} \right.$$

$$\equiv \quad \{ \text{Eq-+ } \}$$

$$[l \ k \cdot \text{zero}, l \ k \cdot \text{succ}] = [\text{succ}, \text{succ} \cdot l \ k]$$

$$\equiv \quad \{ \text{Fusão-+ (esq)}, \text{Absorção-+ (dir)} \}$$

$$l \ k \cdot [\text{zero}, \text{succ}] = [\text{succ}, \text{succ}] \cdot (id + lk)$$

□

Fokkinga:

$$\begin{aligned}
& \equiv \quad \{ \text{Fokkinga } \{50\} \} \\
& \langle f \ k, l \ k \rangle = \langle \langle [one, mul], [succ, succ] \rangle \rangle
\end{aligned}$$

□

$$\begin{aligned}
\text{flatet} &:: ((Integer, Integer), (Integer, Integer)) \rightarrow (Integer, Integer, Integer, Integer) \\
\text{flatet} \ ((a, b), (c, d)) &= (a, b, c, d)
\end{aligned}$$

$$\begin{aligned}
\text{unflatet} &:: (Integer, Integer, Integer, Integer) \rightarrow ((Integer, Integer), (Integer, Integer)) \\
\text{unflatet} \ (a, b, c, d) &= ((a, b), (c, d))
\end{aligned}$$

$$\begin{aligned}
\text{base} &= \text{flatet} \cdot \langle f, g \rangle \\
\text{where } f &= \langle one, \text{succ} \rangle \\
g &= \langle one, one \rangle
\end{aligned}$$

$$\begin{aligned}
-- \text{loop } (a,b,c,d) &= (a*b, b+1, c*d, d+1) \\
\text{loop} &= \text{flatet} \cdot f \cdot \text{unflatet} \\
\text{where } f &= \langle g, h \rangle \\
g &= \langle mul \cdot \pi_1, \text{succ} \cdot \pi_2 \cdot \pi_1 \rangle \\
h &= \langle mul \cdot \pi_2, \text{succ} \cdot \pi_2 \cdot \pi_2 \rangle
\end{aligned}$$

Problema 4

```

inFTree (i1 b) = Unit b
inFTree (i2 (a, (b, c))) = Comp a b c
outFTree (Unit b) = i1 b
outFTree (Comp a b c) = i2 (a, (b, c))
baseFTree f g h = g + (f × (h × h))
recFTree f = baseFTree id id f
cataFTree f = f · recFTree (cataFTree f) · outFTree
anaFTree f = inFTree · recFTree (anaFTree f) · f
hyloFTree f g = cataFTree f · anaFTree g

instance Bifunctor FTree where
  bimap f g = cataFTree (inFTree · baseFTree f g id)

```

De maneira a gerar uma árvore de Pitágoras de uma dada ordem, recebendo a mesma como argumento é necessário aplicar a função **outNat** para gerar um *Either* que será o argumento de um anamorfismo aplicado à ordem recebida. Sendo assim, a função *g* devolve, também, um *Either*, pois é o tipo da nossa *Ftree*. Como sabemos que uma *Unit* corresponde ao maior quadrado que detém lado igual a 1, o lado esquerdo da soma corresponde ao Float 1.0. Ao lado direito queremos multiplicar $\sqrt{2}/2$ elevado à ordem + 1.

```

generatePTree = anaFTree (g · outNat)
  where g = (1.0) + ⟨((sqrt (2) / 2)↑) · succ, ⟨id, id⟩⟩
drawPTree = ⊥

```

Problema 5

A função singleton, dada uma cor cria um Bag com um berlinde dessa mesma cor. Para isso, foi aplicado um split de maneira a ser criado o tuplo (cor, número de berlindes dessa cor). De seguida, é aplicado o *singl* a esse resultado para criar uma lista com o tuplo, após isso basta aplicar o construtor do Bag.

```

-- Dada uma cor constroi um bag com um berlinde dessa cor
singletonbag = B · singl · ⟨id, (1)⟩

```

Para a multiplicação de Bags é necessário o desdobramento do tipo recebido pela **muB** (*Bag (Bag (Bag Marble))*) de maneira a ser obtido apenas uma lista do tipo [(Bag a, Int)] através da aplicação da função **unB** a todos os tuplos da lista. Seguidamente é, ainda, necessário uma nova aplicação da função **unB** para ficar com o tipo desejado. Assim, conseguimos efetuar a multiplicação do inteiro por todos os inteiros de todos os Bags (caso existam) através de um **map**. No final, de maneira a ser devolvido novamente um *Bag a*, é preciso concatenar a lista de listas devolvida pelo map e aplicar o construtor do Bag.

```

-- Multiplicação do monade
μ = B · concat · (map (f) · unB · fmap unB)
  where f (a, b) = map (id × (*b)) a

```

Como é referido no enunciado, um exemplo de uma Distribuição é dada por $d1 = D [(A', 0.02), (B', 0.12), (C', 0.29), (D', 0.35), (E', 0.22)]$. Logo, a um *bagOfMarbles* é aplicado uma *unB* de maneira a ser retirada apenas a lista de tuplos que constitui um Bag, de seguida efetuamos um somatório das quantidades de berlindes de cada cor e geramos um tuplo ([a,Int], Total de berlindes). Agora, a cada elemento da lista, através de um **map** é dividido o número total de berlindes pelo número de berlindes de cada cor e gerados os tuplos constituintes de uma distribuição.

```

dist = D · f · ⟨id, sum · map (π2)⟩ · unB
  where f (a, b) = map (λ(c, d) → (c, toFloat d / toFloat b)) a

```


D Como exprimir cálculos e diagramas em LaTeX/lhs2tex

Estudar o texto fonte deste trabalho para obter o efeito:⁷

$$\begin{aligned}
 & id = \langle f, g \rangle \\
 \equiv & \quad \{ \text{universal property} \} \\
 & \left\{ \begin{array}{l} \pi_1 \cdot id = f \\ \pi_2 \cdot id = g \end{array} \right. \\
 \equiv & \quad \{ \text{identity} \} \\
 & \left\{ \begin{array}{l} \pi_1 = f \\ \pi_2 = g \end{array} \right. \\
 & \square
 \end{aligned}$$

Os diagramas podem ser produzidos recorrendo à *package* L^AT_EX *xymatrix*, por exemplo:

$$\begin{array}{ccc}
 \mathbb{N}_0 & \xleftarrow{\text{in}} & 1 + \mathbb{N}_0 \\
 \downarrow \langle g \rangle & & \downarrow id + \langle g \rangle \\
 B & \xleftarrow{g} & 1 + B
 \end{array}$$

⁷Exemplos tirados de [?].