

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

Departamento de Informática
Universidade do Minho

Junho de 2018

Grupo nr.	99 (preencher)
A80760	Alexandre Pacheco
A82523	Diogo Sobral
A80741	José Pedro Pinto

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:*

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

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)
```



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:

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

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 quad-trees:

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):

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

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* $\mu \cdot \mu = \mu \cdot \text{fmap } \mu$:

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

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*:

```

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

```

inBlockchain = [Bc, Bcs]
outBlockchain (Bc b) = i1 b
outBlockchain (Bcs b) = i2 b
recBlockchain b = id + id × b
cataBlockchain g = g · recBlockchain (cataBlockchain g) · outBlockchain
anaBlockchain g = inBlockchain · recBlockchain (anaBlockchain g) · g
hyloBlockchain h g = cataBlockchain h · anaBlockchain g

```

A função *allTransactions* é bastante simples. Pode ser facilmente definida, como é sugerido no enunciado, por um catamorfismo (definido acima para o tipo de dados *Blockchain*) que percorre toda a estrutura de dados e concatena a lista de transações de cada *Block*.

$$\begin{array}{ccc}
 \text{Blockchain} & \xleftarrow{\text{inBlockchain}=[Bc, Bcs]} & \text{Block} + \text{Block} \times \text{Blockchain} \\
 \downarrow \text{allTransactions}=\langle g \rangle & & \downarrow \text{id} + \text{id} \times \text{allTransactions} \\
 \text{Transactions} & \xleftarrow{g=[\pi_2 \cdot \pi_2, \text{conc} \cdot (\pi_2 \cdot \pi_2 \times \text{id})]} & \text{Block} + \text{Block} \times \text{Transactions}
 \end{array}$$

```

allTransactions = cataBlockchain g
where g = [π2 · π2, conc · (π2 · π2 × id)]

```

A nossa implementação da função *ledger* faz uso do catamorfismo *allTransactions* para reduzir o *Blockchain* recebido a uma lista *Transactions*. Obtendo as *Transactions*, é processada a lista através de um catamorfismo que, para cada *Transaction*, verifica se as entidades envolvidas na mesma já estão no *ledger* acumulado até ao momento, registando-as com o valor referente à transação ou simplesmente somando (ou subtraindo) o valor ao par (*Entity*, *value*) já presente no *ledger*.

$$\begin{array}{ccc}
\text{Transactions} & \xleftarrow{[nil, cons]} & 1 + \text{Transaction} \times \text{Transactions} \\
\downarrow \llbracket [nil, sT] \rrbracket & & \downarrow id + id \times \llbracket [nil, sT] \rrbracket \\
\text{Ledger} & \xleftarrow{g=[nil, sT]} & 1 + (\text{Transaction} \times \text{Ledger})
\end{array}$$

$$sT = \widehat{somaTrans}$$

$$somaTrans :: \text{Transaction} \rightarrow \text{Ledger} \rightarrow \text{Ledger}$$

$$somaTrans (a, (b, c)) l = somaValor (a, -b) (somaValor (c, b) l)$$

$$somaValor :: (\text{Entity}, \text{Value}) \rightarrow \text{Ledger} \rightarrow \text{Ledger}$$

$$somaValor a l = \text{if } isNothing (look (\pi_1 a) l) \text{ then } a : l \\ \text{else } replacePair a l$$

$$replacePair :: (Eq a, Num b) \Rightarrow (a, b) \rightarrow [(a, b)] \rightarrow [(a, b)]$$

$$replacePair a [] = []$$

$$replacePair (a, b) ((x, y) : t) = \text{if } a \equiv x \text{ then } (a, b + y) : t \\ \text{else } (x, y) : replacePair (a, b) t$$

$$ledger = (cataList [nil, sT]) \cdot allTransactions$$

A nossa implementação da função *isValidMagicNr* começa com a aplicação de um *cataBlockchain* que reduz o *Blockchain* a uma lista *[MagicNo]*. Seguidamente é aplicado um anamorfismo que transforma essa lista numa lista *[Bool]* que indica, para cada valor da lista *[MagicNo]*, se esse valor se encontra repetido no resto da lista. Finalmente essa lista *[Bool]* é reduzida a um único *Bool* através de um catamorfismo que efetua a conjunção de todos os booleanos da lista.

$$\begin{array}{ccc}
\text{Blockchain} & \xleftarrow{inBlockchain=[Bc, Bcs]} & \text{Block} + \text{Block} \times \text{Blockchain} \\
\downarrow \llbracket [cons \cdot \langle \pi_1, nil \rangle, cons \cdot (\pi_1 \times id)] \rrbracket & & \downarrow id + id \times \llbracket [cons \cdot \langle \pi_1, nil \rangle, cons \cdot (\pi_1 \times id)] \rrbracket \\
[MagicNo] & \xleftarrow{[cons \cdot \langle \pi_1, nil \rangle, cons \cdot (\pi_1 \times id)]} & \text{Block} + \text{Block} \times [MagicNo]
\end{array}$$

$$\begin{array}{ccc}
[Bool] & \xleftarrow{inBlockchain=[Bc, Bcs]} & 1 + MagicNo \times [Bool] \\
\uparrow \llbracket (id + \langle \neg \cdot el, \pi_2 \rangle) \cdot outList \rrbracket & & \uparrow id + id \times \llbracket (id + \langle \neg \cdot el, \pi_2 \rangle) \cdot outList \rrbracket \\
[MagicNo] & \xleftarrow{[nil, cons]} & 1 + MagicNo \times [MagicNo]
\end{array}$$

$$\begin{array}{ccc}
[Bool] & \xleftarrow{inBlockchain=[Bc, Bcs]} & 1 + Bool \times [Bool] \\
\downarrow \llbracket [true, e] \rrbracket & & \downarrow id + id \times \llbracket [true, e] \rrbracket \\
Bool & \xleftarrow{[true, e]} & 1 + Bool \times [Bool]
\end{array}$$

$$el = \widehat{elem}$$

$$e = \widehat{(\wedge)}$$

$$isValidMagicNr = pim \cdot pam \cdot pum$$

$$\text{where } pum = cataBlockchain [cons \cdot \langle \pi_1, nil \rangle, cons \cdot (\pi_1 \times id)]$$

$$pam = anaList ((id + \langle \neg \cdot el, \pi_2 \rangle) \cdot outList)$$

$$pim = cataList [true, e]$$

Problema 2

```

aux1 (a, (b, c)) = Cell a b c
aux2 (a, (b, (c, d))) = Block a b c d
inQTree = [aux1, aux2]
outQTree (Cell a b c) = i1 (a, (b, c))
outQTree (Block a b c d) = i2 (a, (b, (c, d)))
baseQTree g f = (g × id) + (f × (f × (f × f)))
recQTree f = baseQTree id f
cataQTree g = g · (recQTree (cataQTree g)) · outQTree
anaQTree f = inQTree · (recQTree (anaQTree f)) · f
hyloQTree a c = cataQTree a · anaQTree c
node2p g f = (id × g) + (f × (f × (f × f)))
instance Functor QTree where
  fmap f = cataQTree (inQTree · baseQTree f id)

```

A função rotateQTree é resolvida recorrendo a um simples catamorfismo onde a função que, no caso de receber um Left, faz um swap do p2 do par recebido e, no caso de receber um Righth, troca as Qtree a para a ordem devida.

```

rotateaux (a, (b, (c, d))) = (c, (a, (d, b)))
rotateQTree = cataQTree (inQTree · ((id × swap) + rotateaux))

```

Diagrama rotateQTree

$$\begin{array}{ccc}
 QTree\ a & \xrightarrow{\quad outQTree \quad} & A \times (\mathbb{N}_0 \times \mathbb{N}_0) + (QTree\ a) \uparrow 4 \\
 \downarrow \scriptstyle (inQTree \cdot ((id \times swap) + rotateaux)) & & \downarrow \scriptstyle id + (inQTree \cdot ((id \times swap) + rotateaux)) \\
 QTree\ a & \xleftarrow{\quad inQTree \cdot ((id \times swap) + rotateaux) \quad} & A \times (\mathbb{N}_0 \times \mathbb{N}_0) + (QTree\ a) \uparrow 4
 \end{array}$$

A função scaleQTree é resolvida multiplicando o size da Árvore recebida pela taxa de aumento. Para isto recorreremos a um catamorfismo. O catamorfismo quando recebe um Left (a,(b,c)) multiplica ambos os valores da segunda componente pelo valor recebido.

```

scaleQTree tam = cataQTree (inQTree · node2p size id) where size = (*tam) × (*tam)

```

Diagrama da scaleQTree

$$\begin{array}{ccc}
 QTree\ a & \xrightarrow{\quad outQTree \quad} & A \times (\mathbb{N}_0 \times \mathbb{N}_0) + (QTree\ a) \uparrow 4 \\
 \downarrow \scriptstyle (inQTree \cdot node2p\ size\ id) & & \downarrow \scriptstyle id + (inQTree \cdot node2p\ size\ id) \\
 QTree\ a & \xleftarrow{\quad inQTree \cdot node2p\ size\ id \quad} & A \times (\mathbb{N}_0 \times \mathbb{N}_0) + (QTree\ a) \uparrow 4
 \end{array}$$

```

size = (*tam) * (*tam)

```

A função invertQTree tem uma resolução semelhante às anteriores. Para a resolver bastou recorrer a um catamorfismo que, quando recebe um Left (a,(b,c)), aplica no p1 a função invcor que é responsável por inverter as cores.

```

invcor (PixelRGBA8 a b c d) = PixelRGBA8 (255 - a) (255 - b) (255 - c) d
invertQTree = cataQTree (inQTree · baseQTree invcor id)

```

Diagrama da invertQTree

$$\begin{array}{ccc}
 QTree \text{ PixelRGBA8} & \xrightarrow{\text{outQTree}} & \text{PixelRGBA8} \times (\mathbb{N}_0 \times \mathbb{N}_0) + (QTree \ a) \uparrow 4 \\
 \downarrow \langle \text{inQTree} \cdot \text{baseQTree} \ \text{invcor} \ id \rangle & & \downarrow id + \langle \text{inQTree} \cdot \text{baseQTree} \ \text{invcor} \ id \rangle \\
 QTree \text{ PixelRGBA8} & \xleftarrow{\text{inQTree} \cdot \text{invcor} \ id} & \text{PixelRGBA8} \times (\mathbb{N}_0 \times \mathbb{N}_0) + (QTree \ a) \uparrow 4
 \end{array}$$

O problema da compressQTree fica resolvido cortando a árvore recebida e ficando esta com a altura no fim igual a altura inicial - n recebido. Para isto usamos um anamorfismo que recebe um par com a altura máxima que a árvore pode ter e a árvore inicial. A função invocada no anamorfismo no caso de receber um par (0,Block _ _ _) converte o block em cell e produz um i1 do tipo da árvore. No caso da primeira componente ser diferente de 0 a função produz um i2: um tuplo de pares com 4 entradas em que o p1 é o int com a altura inicial e o p2 um das árvores do bloco.

```

compressQTree n = anaQTree transformaTree . fl
  where fl = (nn × id) · ⟨depthQTree, id⟩
        nn = cond (>n) ((+) (negate n)) 0
transformaTree :: (Int, QTree a) → (a, (Int, Int)) + ((Int, QTree a), ((Int, QTree a), ((Int, QTree a), (Int, QTree a))))
transformaTree (_, Cell a b c) = i1 (a, (b, c))
transformaTree (0, a) = i1 (b, (d, c)) where (Cell b d c) = transfor a
transformaTree (size, Block a b c d) = let nsize = size - 1
  in i2 ((nsize, a), ((nsize, b), ((nsize, c), (nsize, d))))
transfor (Cell a b c) = Cell a b c
transfor a = Cell na (π1 size) (π2 size)
  where na = π1 $ maxsize a
        size = sizeQTree a
mymax :: [(a, (Int, Int))] → (a, (Int, Int))
mymax [x] = x
mymax (h1 : h2 : t) = let a1 = π2 h1
  a = π1 a1 * π2 a1
  b1 = π2 h2
  b = π1 b1 * π2 b1
  in if (a > b)
    then mymax (h1 : t)
    else mymax (h2 : t)
maxsize :: QTree a → (a, (Int, Int))
maxsize = cataQTree [id, f]
  where f (a, (b, (c, d))) = mymax [a, b, c, d]

```

Diagrama da compressQTree

$$\begin{array}{ccc}
 QTree \ a & \xleftarrow{\text{inQTree}} & A \times (\mathbb{N}_0 \times \mathbb{N}_0) + (QTree \ a) \uparrow 4 \\
 \uparrow \llbracket \text{transformaTree} \rrbracket & & \uparrow id + \llbracket \text{transformaTree} \rrbracket \\
 \mathbb{N}_0 \times QTree \ a & \xrightarrow{\text{transformaTree}} & A \times (\mathbb{N}_0 \times \mathbb{N}_0) + (\mathbb{N}_0 \times QTree \ a) \uparrow 4 \\
 \uparrow \langle ((-n), 0) \cdot (>n) ? \times id \rangle & & \\
 \mathbb{N}_0 \times QTree \ a & & \\
 \uparrow \langle \text{depthQTree}, id \rangle & & \\
 QTree \ a & &
 \end{array}$$

A função outlineQTree executa dois catamorfismos. O primeiro transforma a QTree a numa QTree Bool segundo a função recebida e o segundo, quando recebe um tipo Left, converte o elemento na devida matrix, altera-a ficando o centro da matrix a False e as bordas a False. No fim é executada a qt2bm na última matrix.

```

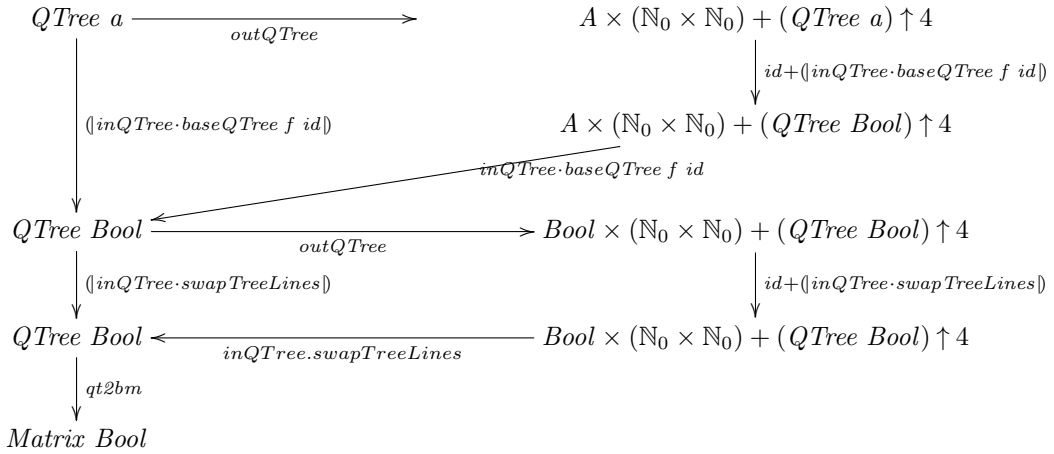
matrixtrns :: Matrix Bool → Matrix Bool
matrixtrns a = let (nrow, ncol) = ⟨nrows, ncols⟩ a
  cols = mapCol (\_x → True) 1 (mapCol (\_x → True) ncol a)
  rows = mapRow (\_x → True) 1 (mapRow (\_x → True) nrow cols)
in rows

swapTreeLines = cond typecheck h id
where h = cond (λ(i1 (a, (b, c))) → a ≡ True) (outQTree · bm2qt · matrixtrns · qt2bm · swapcel · inQTree) id
  swapcel (Cell a b c) = Cell False b c
  typecheck = [True, False]

outlineQTree f = qt2bm · cataQTree (inQTree · swapTreeLines) · cataQTree (inQTree · baseQTree f id)

```

Diagrama da outlineQTree



Problema 3

Com base na implementação descrita no enunciado, podemos perceber que a função *base* gera um t pulo com 4 elementos. Da  conclu mos que a fun  o *loop* recebe um t pulo de 4 elementos e devolve outro (o ciclo *for* tamb m devolve um t pulo assim). $for\ loop\ (base\ k) = (\llbracket base\ k, loop \rrbracket)$ Por enquanto, concentremo-nos em trabalhar com as leis da recursividade m ltipla:

$$\begin{aligned}
& fun \cdot \mathbf{in} = h \cdot F < fun, fun2 > \\
\equiv & \quad \{ \text{Def. in ; Def. (F f) nos naturais} \} \\
& fun \cdot [\underline{0}, succ] = h \cdot (id + \langle fun, fun2 \rangle) \\
\equiv & \quad \{ \text{Fus o-+ ; Universal-+} \} \\
& \begin{cases} fun \cdot \underline{0} = h \cdot (id + \langle fun, fun2 \rangle) \cdot i_1 \\ fun \cdot succ = h \cdot (id + \langle fun, fun2 \rangle) \cdot i_2 \end{cases} \\
\equiv & \quad \{ \text{Fus o const. ; Natural i1; Natural i2} \} \\
& \begin{cases} fun\ 0 = h \cdot i_1 \cdot id \\ fun \cdot succ = h \cdot i_2 \cdot \langle fun, fun2 \rangle \end{cases} \\
\equiv & \quad \{ \text{Natural id; Igualdade existencial; Def. comp.} \} \\
& \begin{cases} fun\ 0 = h \cdot i_1 \\ fun\ (succ\ x) = (h \cdot i_2) \cdot \langle fun, fun2 \rangle\ x \end{cases} \\
\equiv & \quad \{ \text{Def. succ; Def. split} \} \\
& \begin{cases} fun\ 0 = h \cdot i_1 \\ fun\ (x + 1) = (h \cdot i_2) \cdot ((fun\ x), (fun2\ x)) \end{cases}
\end{aligned}$$

Agora, podemos aplicar as funções que são referidas no enunciado. Primeiro apliquemos esta lei a f k e l k :

$$\begin{aligned}
& \left\{ \begin{array}{l} \left\{ \begin{array}{l} f \ k \ 0 = h1 \cdot i_1 \\ f \ k \ (x+1) = (h1 \cdot i_2) \cdot ((f \ k \ x), (l \ k \ x)) \\ l \ k \ 0 = h2 \cdot i_1 \\ l \ k \ (x+1) = (h2 \cdot i_2) \cdot ((f \ k \ x), (l \ k \ x)) \end{array} \right. \\ \equiv \quad \left\{ \text{Def. } f; h1 = [\underline{1}, mul]; h2 = [(k+1), succ \cdot \pi_2]; \text{Cancelamento-+} \right\} \\ \left\{ \begin{array}{l} \left\{ \begin{array}{l} f \ k \ 0 = 1 \\ f \ k \ (x+1) = mul \ (f \ k \ x) \ (l \ k \ x) \\ l \ k \ 0 = k+1 \\ l \ k \ (x+1) = succ \cdot l \ k \ x \end{array} \right. \end{array} \right.
\end{aligned}$$

Repetindo o processo para g e s , temos:

$$\{ \dots; h3 = [\underline{1}, mul]; h4 = [\underline{1}, succ \cdot \pi_2] \} \left\{ \begin{array}{l} \left\{ \begin{array}{l} g \ 0 = 1 \\ g \ (x+1) = mul \ (g \ x) \ (s \ x) \\ s \ 0 = 1 \\ s \ (x+1) = succ \cdot s \ x \end{array} \right. \end{array} \right.$$

Recordando que $h1 = [\underline{1}, mul]$, $h2 = [(k+1), succ \cdot \pi_2]$, $h3 = [\underline{1}, mul]$, e $h4 = [\underline{1}, succ \cdot \pi_2]$, podemos voltar à lei de Fokkinga e concluir que:

$$\begin{cases} \langle f \ k, l \ k \rangle = \langle \langle [\underline{1}, mul], [k+1, succ \cdot \pi_2] \rangle \rangle \\ \langle g, s \rangle = \langle \langle [\underline{1}, mul], [\underline{1}, succ \cdot \pi_2] \rangle \rangle \end{cases}$$

Como esta expressão sugere, podemos combinar estes dois catamorfismos seguindo a lei de banana-split :

$$\begin{aligned}
& \langle \langle \langle h1, h2 \rangle \rangle, \langle \langle h3, h4 \rangle \rangle \rangle = \langle \langle \langle [\underline{1}, mul], [(k+1), succ \cdot \pi_2] \rangle \times \langle [\underline{1}, mul], [\underline{1}, succ \cdot \pi_2] \rangle \rangle \cdot \langle F \ \pi_1, F \ \pi_2 \rangle \rangle \\
& \equiv \quad \left\{ \text{Absorção-x ; Def. } F \text{ nos naturais} \right\} \\
& \langle \langle \langle h1, h2 \rangle \rangle, \langle \langle h3, h4 \rangle \rangle \rangle = \langle \langle \langle [\underline{1}, mul], [(k+1), succ \cdot \pi_2] \rangle \cdot (id + \pi_1) \times \langle [\underline{1}, mul], [\underline{1}, succ \cdot \pi_2] \rangle \cdot (id + \pi_2) \rangle \rangle \\
& \equiv \quad \left\{ \text{Absorção-x ; Absorção-+ ; Def. id} \right\} \\
& \langle \langle \langle h1, h2 \rangle \rangle, \langle \langle h3, h4 \rangle \rangle \rangle = \langle \langle \langle [\underline{1}, mul \cdot \pi_1], [(k+1), succ \cdot \pi_2 \cdot \pi_1] \rangle \times \langle [\underline{1}, mul \cdot \pi_2], [\underline{1}, succ \cdot \pi_2 \cdot \pi_2] \rangle \rangle \rangle \\
& \equiv \quad \left\{ \text{Lei da Troca} \right\} \\
& \langle \langle \langle h1, h2 \rangle \rangle, \langle \langle h3, h4 \rangle \rangle \rangle = \langle \langle \langle [\underline{1}, (k+1)], \langle mul \cdot \pi_1, succ \cdot \pi_2 \cdot \pi_1 \rangle \rangle \times \langle [\underline{1}, \underline{1}], \langle mul \cdot \pi_2, succ \cdot \pi_2 \cdot \pi_2 \rangle \rangle \rangle \rangle \\
& \equiv \quad \left\{ \text{Def.-Const; Fusão-x ; Fusão-+ ; Def.-x} \right\} \\
& \langle \langle \langle h1, h2 \rangle \rangle, \langle \langle h3, h4 \rangle \rangle \rangle = \langle \langle \langle [\underline{1}, \underline{k}+1], \langle mul, succ \cdot \pi_2 \rangle \rangle, [\underline{1}, \underline{1}], \langle mul, succ \cdot \pi_2 \rangle \rangle \rangle \\
& \equiv \quad \left\{ \text{Lei da Troca} \right\} \\
& \langle \langle \langle h1, h2 \rangle \rangle, \langle \langle h3, h4 \rangle \rangle \rangle = \underbrace{\langle \langle [\underline{1}, (k+1)], \langle \underline{1}, \underline{1} \rangle \rangle \rangle}_{base \ k} \underbrace{\langle \langle mul, succ \cdot \pi_2 \rangle, \langle mul, succ \cdot \pi_2 \rangle \rangle \rangle}_{loop}
\end{aligned}$$

Recordando a interpretação inicial, e que $\text{for } loop \ (base \ k) = \langle [base \ k, loop] \rangle$, temos então:

$$\langle [base \ k, loop] \rangle = \underbrace{\langle \langle [\underline{1}, (k+1)], \langle \underline{1}, \underline{1} \rangle \rangle \rangle}_{base \ k} \underbrace{\langle \langle mul, succ \cdot \pi_2 \rangle, \langle mul, succ \cdot \pi_2 \rangle \rangle \rangle}_{loop}$$

Obtendo esta expressão, podemos observar uma forma muito parecida com o pretendido, a partir da qual podemos já deduzir as funções $base \ k$ e $loop$. Basta olhar para o either no membro direito da equação. Note-se que a equação obtida funciona com um par de pares que, por simplicidade, foi definido como um túpulo de 4 elementos, pelo que as funções são definidas de modo *pointwise*.

$$\begin{aligned}
base \ k &= (1, k+1, 1, 1) \\
loop \ (a, x, b, y) &= (a * x, succ \ x, b * y, succ \ y)
\end{aligned}$$

Problema 4

```

ftreeunit b = Unit b
ftreeComp (a, (b, c)) = Comp a b c
inFTree = [ftreeunit, ftreeComp]
outFTree (Unit b) = i1 b
outFTree (Comp a b c) = i2 (a, (b, c))
baseFTree g f k = f + (g × (k × k))
recFTree f = baseFTree id id f
cataFTree g = g · (recFTree (cataFTree g)) · outFTree
anaFTree f = inFTree · (recFTree (anaFTree f)) · f
hyloFTree a c = cataFTree a · anaFTree c
instance Bifunctor FTree where
    bimap g f = cataFTree (inFTree · baseFTree g f id)

```

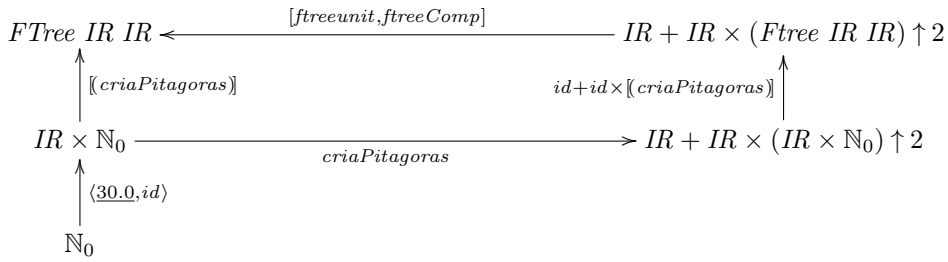
O problema relativo a gerar a árvore de pitagoras foi resolvido recorrendo a um anamorfismo que recebe um par Float × Int em que o Int representa a altura que a árvore pode ter e o float o tamanho do quadrado. A função que é passada ao anamorfismo no caso de receber a componente do par responsável pela altura com o valor 0 produz um i₁ com o valor da primeira componente. No caso de ser diferente de 0 é criado um i₂ (A,(b,b)) em que o A tem o valor da primeira componente e o B = (A*sqrt(2) / 2, altura -1)

```

criaPitagoras :: (Square, Int) → Square + (Square, ((Square, Int), (Square, Int)))
criaPitagoras (a, 0) = i1 (a)
criaPitagoras (a, b) = i2 (a, ((a * razao, nb), (a * razao, nb))) where nb = pred b
    razao = sqrt (2) / 2
generatePTree = anaFTree criaPitagoras · initype
where initype = ⟨30.0, id⟩

```

Diagrama da generatePTree



A função drawPTree pode ser partida em várias partes. Consiste em transformar a árvore inicial numa árvore de pictures onde as imagens já se encontram desenhadas e nos sítios certos do referencial. Seguidamente corremos um hylomorfismo que transforma uma árvore de pictures numa lista de (int, Picture) em que o int representa a ordem da árvore. Posteriormente, esta lista é transformada agrupando os valores da mesma ordem e juntando todas as imagens.

```

criaFRPic :: ((Bool, Float, (Float, Float)), FTree Float Float) → Picture + (Picture, (((Bool, Float, (Float, Float)),
criaFRPic ((s, a, (x, y)), Unit b) = i1 (createImage (s, (b, (x, y))))
criaFRPic ((s, a, (x, y)), Comp size f1 f2) = i2 (createImage (s, (size, (x, y))), (b, c))
where vd = rotateV a ⟨/2, id⟩ size
    ve = rotateV a ⟨(negate) · (/2), id⟩ size
    ang = atan 1
    na = (ang + a, a - ang)
    b = ((¬ s, π1 na, (x + π1 ve, y + π2 ve)), f1)
    c = ((¬ s, π2 na, (x + π1 vd, y + π2 vd)), f2)
createImage :: (Bool, (Float, (Float, Float))) → Picture

```

```

createImage = cond ( $\pi_1$ ) rot nrot
  where rot ( $\_$ , ( $size$ , ( $x$ ,  $y$ ))) = translate  $x$   $y$  (rotate (45.0) (rectangleSolid size size))
        nrot ( $\_$ , ( $size$ , ( $x$ ,  $y$ ))) = translate  $x$   $y$  (rectangleSolid size size)

loopAna :: [[ $a$ ]]  $\rightarrow$  [[ $a$ ]]
loopAna [] = []
loopAna [x] = [x]
loopAna (h : t) = h : loopAna ((h ++ a) : tail t)
  where a = head t

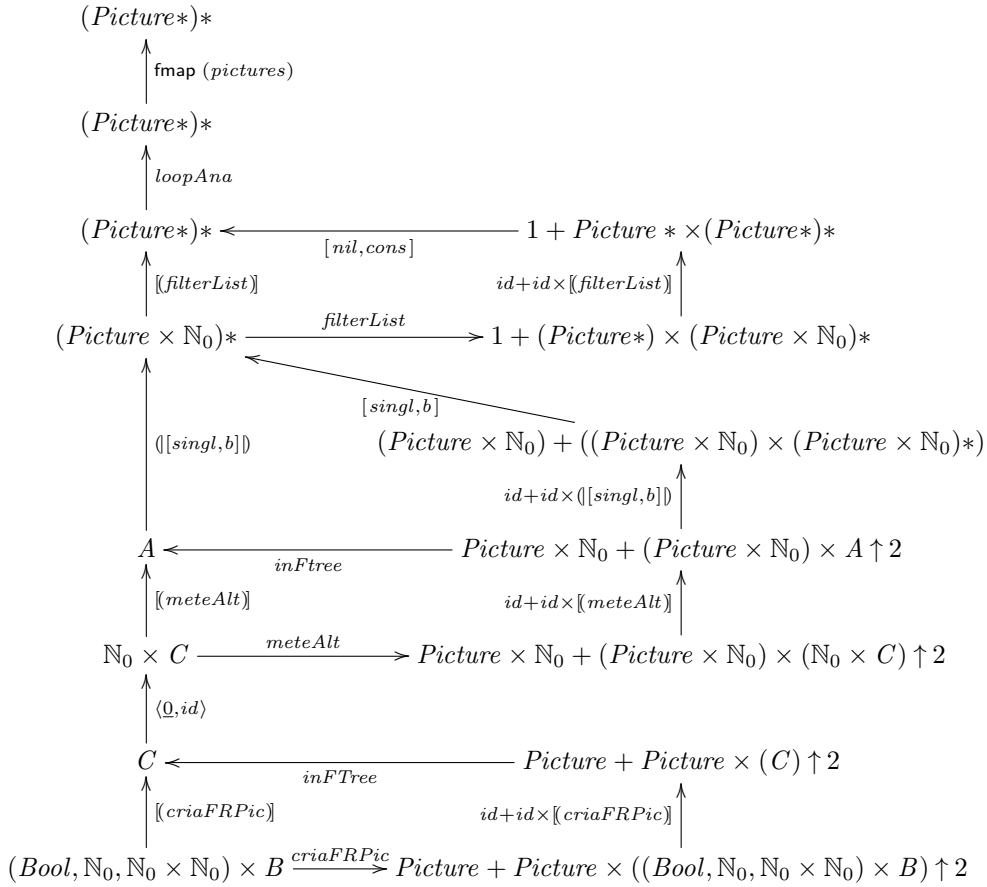
meteAlt :: (Int, FTree a b)  $\rightarrow$  (b, Int) + ((a, Int), ((Int, FTree a b), (Int, FTree a b)))
meteAlt (x, Unit b) =  $i_1$  (b, x)
meteAlt (x, Comp a f1 f2) =  $i_2$  ((a, x), ((x - 1, f1), (x - 1, f2)))

filterList :: [(a, Int)]  $\rightarrow$  [a] + ([a], [(a, Int)])
filterList [] =  $i_1$  []
filterList t =  $i_2$  (f1 t, f2 t) where f1 = map ( $\pi_1$ )  $\cdot$  filter ( $\lambda k \rightarrow \pi_2 k \equiv alt$ )
        alt =  $\pi_2$  (head t)
        f2 = filter ( $\lambda k \rightarrow \pi_2 k \not\equiv alt$ )

breadthFirst = fmap (pictures)  $\cdot$  loopAna
drawPTree = breadthFirst  $\cdot$  anaList filterList  $\cdot$  hyloFTree [singl, b] (meteAlt)  $\cdot$  h
  where h = ( $\mathbb{Q}$ , k)
        b (a, (b, c)) = [a] ++ b ++ c
        k = anaFTree (criaFRPic)  $\cdot$  initype
        initype a = ((False, 0, (0, 0)), a)

```

Diagrama da drawPTree



$b(a, (d, c)) = [a] ++ d ++ c$

$A = FTree (Picture \times \mathbb{N}_0) (Picture \times \mathbb{N}_0)$

$B = FTree Float Float$

$C = FTree Picture Picture$

```
main :: IO ()
main = do putStrLn ("Número de níveis da árvore?")
        ni ← getLine
        let n = read ni :: Int
        animatePTree n
```

Problema 5

Diagrama da μB

$$\begin{array}{ccc}
 B(C) & \xleftarrow{\text{fmap } unB} & B(B(A)) \\
 \mu \downarrow & & \downarrow B.juntaBag \cdot unB \cdot \text{fmap } unB \\
 C & \xrightarrow{B.juntaBag} & B(A)
 \end{array}$$

```
singletonbag = B · singl · ⟨id, 1⟩
μ = B · juntaBag · unB · fmap unB
transforBagBagList :: [(t, Int), Int] → [(t, Int)] + [(t, Int), [(t, Int), Int]]
transforBagBagList [] = i1 []
transforBagBagList list = i2 (ele, tail list)
  where (listmod, x) = head list
        ele = map (id × (*x)) listmod
juntaBag = hyloList [nil, h] transforBagBagList
  where h (a, b) = a ++ b
dist b = D list
  where size = sum · map π2 $ unB b
        list = fmap (id × odds) $ unB b
        odds x = toFloat x / toFloat size
```

D Como exprimir cálculos e diagramas em LaTeX/lhs2tex

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

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

□

⁷Exemplos tirados de [?].

Os diagramas podem ser produzidos recorrendo à *package* \LaTeX **xymatrix**, por exemplo:

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