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

Departamento de Informática
Universidade do Minho

Junho de 2018

Grupo nr.	16
a77730	Sérgio Jorge
a77870	Vítor Castro
a79116	Marcos Pereira

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-processador que faz "pretty printing" de código Haskell em LaTeX 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

[1] 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.

$$\mathbf{data}\ Blockchain = Bc\ \{bc :: Block\} \mid Bcs\ \{bcs :: (Block, Blockchain)\}\ \mathbf{deriving}\ Show$$

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

$$\mathbf{type}\ Block = (MagicNo, (Time, Transactions))$$

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.

$$\mathbf{type}\ Transaction = (Entity, (Value, Entity))$$
$$\mathbf{type}\ Transactions = [Transaction]$$

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

$$\mathbf{type}\ Ledger = [(Entity, Value)]$$

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.

$\mathbf{type}\ MagicNo = String$
$\mathbf{type}\ Time = Int$ -- em milisegundos
$\mathbf{type}\ Entity = String$
$\mathbf{type}\ Value = Int$

Neste contexto, implemente as seguintes funções:

1. Defina a função $allTransactions :: Blockchain \to Transactions$, 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 $ledger :: Blockchain \to Ledger$, utilizando catamorfismos e/ou anamorfismos, que calcula o ledger (i.e., o valor disponível) de cada entidade numa uma 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:*

   $$prop1b = \mathsf{length} \cdot ledger \leqslant (2*) \cdot \mathsf{length} \cdot allTransactions$$

   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 $isValidMagicNr :: Blockchain \to Bool$, 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 quadtrees. Uma *quadtree* é uma árvore quaternária em que cada nodo tem quatro sub-árvores e cada folha representa um valor bi-dimensional.

$\mathbf{data}\ QTree\ a = Cell\ a\ Int\ Int \mid Block\ (QTree\ a)\ (QTree\ a)\ (QTree\ a)\ (QTree\ a)$
  $\mathbf{deriving}\ (Eq, Show)$

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

O anamorfismo $bm2qt$ converte um bitmap em forma matricial na sua codificação eficiente em quadtrees, e o catamorfismo $qt2bm$ executa a operação inversa:

[2] Cf. módulo $Data.Matrix$.

```
( 0 0 0 0 0 0 0 0 )'
( 0 0 0 0 0 0 0 0 )'
( 0 0 0 0 1 1 1 0 )'
( 0 0 0 0 1 1 0 0 )'
( 1 1 1 1 1 1 0 0 )'
( 1 1 1 1 1 1 0 0 )'
( 1 1 1 1 0 0 0 0 )'
( 1 1 1 1 0 0 0 1 )'
```

(a) Matriz de exemplo bm.

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

(b) Quadtree de exemplo qt.

Figura 1: Exemplos de representações de bitmaps.

$$
\begin{aligned}
&bm2qt :: (Eq\ a) \Rightarrow Matrix\ a \to QTree\ a \\
&bm2qt = anaQTree\ f\ \textbf{where} \\
&\quad f\ m = \textbf{if}\ one\ \textbf{then}\ i_1\ u\ \textbf{else}\ i_2\ (a,(b,(c,d))) \\
&\qquad \textbf{where}\ x = (nub \cdot toList)\ m \\
&\qquad\qquad u = (head\ x, (ncols\ m, nrows\ m)) \\
&\qquad\qquad one = (ncols\ m \equiv 1 \vee nrows\ m \equiv 1 \vee \mathsf{length}\ \ x \equiv 1) \\
&\qquad\qquad (a,b,c,d) = splitBlocks\ (nrows\ m\ `div`\ 2)\ (ncols\ m\ `div`\ 2)\ m
\end{aligned}
$$

$$
\begin{aligned}
&qt2bm :: (Eq\ a) \Rightarrow QTree\ a \to Matrix\ a \\
&qt2bm = cataQTree\ [f,g]\ \textbf{where} \\
&\quad f\ (k,(i,j)) = matrix\ j\ i\ \underline{k} \\
&\quad g\ (a,(b,(c,d))) = (a \updownarrow b) \leftrightarrow (c \updownarrow d)
\end{aligned}
$$

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 côr. 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:

$$
\begin{aligned}
whitePx &= PixelRGBA8\ 255\ 255\ 255\ 255 \\
blackPx &= PixelRGBA8\ 0\ 0\ 0\ 255 \\
redPx &= PixelRGBA8\ 255\ 0\ 0\ 255
\end{aligned}
$$

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

$$
\begin{aligned}
&readBMP :: FilePath \to \mathsf{IO}\ (Matrix\ PixelRGBA8) \\
&writeBMP :: FilePath \to Matrix\ PixelRGBA8 \to \mathsf{IO}\ ()
\end{aligned}
$$

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

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

Esta questão aborda operações de processamento de imagens utilizando quadtrees:

1. Defina as funções $rotateQTree :: QTree\ a \to QTree\ a$, $scaleQTree :: Int \to QTree\ a \to QTree\ a$ e $invertQTree :: QTree\ a \to QTree\ a$, como catamorfismos e/ou anamorfismos, que rodam[3], redimensionam [4] e invertem as cores de uma quadtree[5], 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"'
```

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

$$prop2c = rotateMatrix \cdot qt2bm \equiv qt2bm \cdot rotateQTree$$

[3] Segundo um ângulo de 90º no sentido dos ponteiros do relógio.
[4] Multiplicando o seu tamanho pelo valor recebido.
[5] 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.

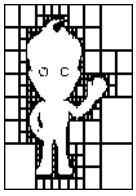
(d) Inversão de cores.

(e) Compresão de 1 nível.

(f) Compresão de 2 níveis.

(g) Compresão de 3 níveis.

(h) Compresão de 4 níveis.



(i) Bitmap de contorno.

(j) Bitmap com contorno.

Figura 2: Manipulação de uma figura bitmap utilizando quadtrees.

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

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

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

$$prop2e = shapeQTree \cdot invertQTree \equiv shapeQTree$$

2. Defina a função $compressQTree :: Int \to QTree\ a \to QTree\ a$, 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:*

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

3. Defina a função $outlineQTree :: (a \to Bool) \to QTree\ a \to Matrix\ Bool$, 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:*

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

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

$$teste2a = outlineQTree\ (\equiv 0)\ qt \equiv qtOut$$

Problema 3

O cálculo das combinações de n k-a-k,

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

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,

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

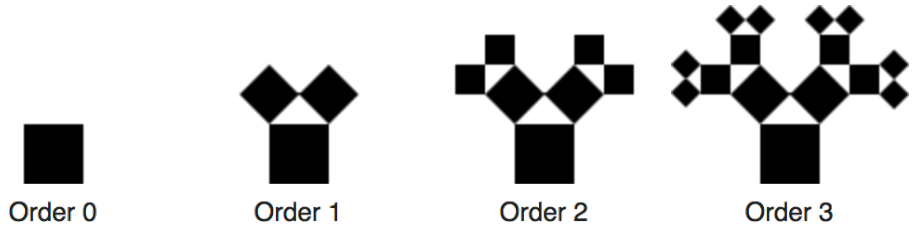


Figura 3: Passos de construção de uma árvore de Pitágoras de ordem 3.

onde

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

assumindo-se $d = n - k \geqslant 0$. 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

$$g\ 0 = 1$$
$$g\ (d+1) = \underbrace{(d+1)}_{s\ d} * g\ d$$
$$s\ 0 = 1$$
$$s\ (d+1) = s\ n + 1$$

A partir daqui alguém derivou a seguinte implementação:

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

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):*

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

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:

$$\textbf{data}\ FTree\ a\ b = Unit\ b \mid Comp\ a\ (FTree\ a\ b)\ (FTree\ a\ b)\ \textbf{deriving}\ (Eq, Show)$$
$$\textbf{type}\ PTree = FTree\ Square\ Square$$
$$\textbf{type}\ Square = Float$$

1. Defina a função $generatePTree :: Int \to PTree$, 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:*

   $$prop4a\ (SmallNat\ n) = (depthFTree \cdot generatePTree)\ n \equiv n$$

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

   $$prop4b\ (SmallNat\ n) = (isBalancedFTree \cdot generatePTree)\ n$$

2. Defina a função $drawPTree :: PTree \to [Picture]$, 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

$$\textbf{data}\ Marble = Red \mid Pink \mid Green \mid Blue \mid White\ \textbf{deriving}\ (Read, Show, Eq, Ord)$$

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":

$$\textbf{data}\ Bag\ a = B\ [(a, Int)]\ \textbf{deriving}\ (Ord)$$

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

$$\begin{aligned} &marbleWeight :: Marble \to Int \\ &marbleWeight\ Red\ \ = 3 \\ &marbleWeight\ Pink\ = 2 \\ &marbleWeight\ Green\ = 3 \\ &marbleWeight\ Blue\ = 6 \\ &marbleWeight\ White\ = 2 \end{aligned}$$

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

$$marbleWeights = \mathsf{fmap}\ marbleWeight\ bagOfMarbles$$

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

[6] "Marble" traduz para "berlinde" em português.

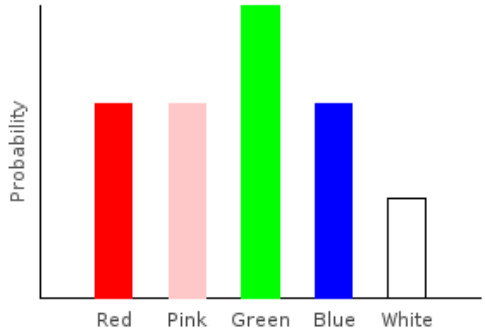


Figura 4: Distribuição de berlindes num saco.

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

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,

$$\begin{aligned}&\textbf{instance}\ Functor\ Bag\ \textbf{where}\\&\quad \mathsf{fmap}\ f = B \cdot \mathsf{map}\ (f \times id) \cdot unB\\&\textbf{instance}\ Monad\ Bag\ \textbf{where}\\&\quad x \gg\!= f = (\mu \cdot \mathsf{fmap}\ f)\ x\ \textbf{where}\\&\quad return = singletonbag\end{aligned}$$

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* $\mu \cdot return = id$*:*

   $$test5a = bagOfMarbles \equiv \mu\ (return\ bagOfMarbles)$$

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

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

   onde $b3$ *é um saco dado em anexo.*

Anexos

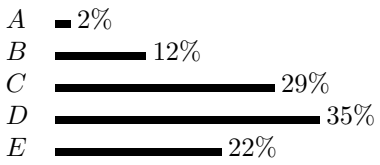
A Mónade para probabilidades e estatística

Mónades são functores 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

$$\textbf{newtype}\ \mathsf{Dist}\ a = D\ \{unD :: [(a, ProbRep)]\} \tag{2}$$

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

Cada par (a, p) numa distribuição $d :: \mathsf{Dist}\ a$ 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,



será representada pela distribuição

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

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 `ominus` b)
      ominus a b = a ++ neg b
      neg x = [(k,-i) | (k,i) <- x]
```

Ainda sobre o mónade Bag:

$$\begin{spec}
\mathbf{instance}\ Applicative\ Bag\ \mathbf{where}
  pure = return
  (<*>) = aap
\end{spec}$$

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):

$$\begin{spec}
b3 :: Bag\ (Bag\ (Bag\ Marble))
b3 = B\ [(B\ [(B\ [(Pink,2),(Green,3),(Red,2),(Blue,2),(White,1)],5)
  ,(B\ [(Pink,1),(Green,2),(Red,1),(Blue,1)],2)],2)]
\end{spec}$$

Outras funções auxiliares:

$$\begin{spec}
a \mapsto b = (a,b)
consol :: (Eq\ b) \Rightarrow [(b,Int)] \to [(b,Int)]
consol = filter\ nzero \cdot \mathsf{map}\ \ (id \times sum) \cdot col\ \mathbf{where}\ nzero\ (_,x) = x \not\equiv 0
isempty :: Eq\ a \Rightarrow [(a,Int)] \to Bool
isempty = all\ (\equiv 0) \cdot \mathsf{map}\ \ \pi_2 \cdot consol
col\ x = nub\ [k \mapsto [d' \mid (k',d') \leftarrow x, k' \equiv k] \mid (k,d) \leftarrow x]
consolidate :: Eq\ a \Rightarrow Bag\ a \to Bag\ a
consolidate = B \cdot consol \cdot unB
\end{spec}$$

C Soluções dos alunos

Problema 1

Antes de proceder à resolução das alíneas, foi necessária a definição das funções relativas à manipulação de Blockchains:

$$inBlockchain = [Bc, Bcs]$$

O out da Blockchain foi calculado da seguinte forma:

$$\begin{spec}
    out \cdot \mathbf{in} = id
\equiv \quad \{\ \text{in = either Bc Bcs}\ \}
    out \cdot [Bc, Bcs] = id
\equiv \quad \{\ \text{Lei 20}\ \}
    [out \cdot Bc, out \cdot Bcs] = id
\equiv \quad \{\ \text{Lei 27}\ \}
    \begin{cases} out \cdot Bc = i_1 \\ out \cdot Bcs = i_2 \end{cases}
\equiv \quad \{\ \text{Passando para pointwise}\ \}
    \begin{cases} out\ (Bc\ a) = i_1\ a \\ out\ (Bcs\ (a,b)) = i_2\ (a,b) \end{cases}
\end{spec}$$

□

Resultando no seguinte código Haskell:

$$\begin{spec}
outBlockchain\ (Bc\ a) = i_1\ a
outBlockchain\ (Bcs\ (a,b)) = i_2\ (a,b)
\end{spec}$$

As restantes funções são:

$$recBlockchain\ f = id + (id \times f)$$
$$cataBlockchain\ g = g \cdot (recBlockchain\ (cataBlockchain\ g)) \cdot outBlockchain$$
$$anaBlockchain\ g = inBlockchain \cdot (recBlockchain\ (anaBlockchain\ g)) \cdot g$$
$$hyloBlockchain\ f\ g = cataBlockchain\ f \cdot anaBlockchain\ g$$

allTransactions

Para definir o allTransactions, que calcula uma lista de todas as transações de uma Blockchain, foi usado um catamorfismo de Blockchains:

$$\begin{array}{ccc} Blockchain & \xrightarrow{out} & Block + Block \times Blockchain \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle id+id\times f} \\ C & \xleftarrow[g]{} & Block + Block \times C \end{array}$$

Este catamorfismo simplesmente usa a função π_2 para extrair as listas de transações de cada bloco:

$$allTransactions = cataBlockchain\ [getTransaction, getTransactions]$$
$$getTransaction = \pi_2 \cdot \pi_2$$
$$getTransactions = \mathsf{conc} \cdot ((\pi_2 \cdot \pi_2) \times id)$$

É importante notar que *conc* gera uma lista concatenando um par que contém duas listas: $([a],[a]) \to [a]$

ledger

Para calcular o ledger de uma blockchain vamos fazer 3 passos:

1. Obter a lista das transações de uma blockchain (usando *allTransactions*);
2. Obter uma lista de entidades a partir da lista de transações (usando catamorfismo *getEntities*);
3. Eliminar repetições da lista de entidades;
4. Calcular o saldo de cada entidade fazendo map das entidades sobre uma função *getBalance*.

O diagrama seguinte mostra a composição de funções utilizada:

$$\begin{array}{c} Blockchain \\ \downarrow{\scriptstyle allTransactions} \\ [transaction] \\ \downarrow{\scriptstyle \langle getUniqueEntities, id\rangle} \\ ([entity],[transaction]) \\ \downarrow{\scriptstyle id\times getBalance} \\ ([entity],(getBalance\ [transaction])) \\ \downarrow{\scriptstyle \langle \pi_1, mapPair\cdot swap\rangle} \\ ([entity],[saldo]) \\ \downarrow{\scriptstyle zipPair} \\ [(entity, saldo)] \end{array}$$

A lista $[(entity, saldo)]$ no fim do diagrama anterior é o ledger que pretendemos calcular.

O código Haskell correspondente ao diagrama será:

$$ledger = zipPair \cdot \langle \pi_1, mapPair \cdot swap \rangle \cdot (id \times getBalance) \cdot \langle getUniqueEntities, id \rangle \cdot allTransactions$$

getEntities será um catamorfismo de listas que calcula uma lista de entidades a partir de uma de transações.

$$getEntities :: [Transaction] \to [String]$$
$$getEntities = cataList\ [nil, \mathsf{conc} \cdot ((pairToList \cdot \langle \pi_1, \pi_2 \cdot \pi_2 \rangle) \times id)]$$

Para transformar uma transação em uma lista com as duas entidades, usamos $pairToList \cdot \langle \pi_1, \pi_2 \cdot \pi_2 \rangle$.
No caso final, usamos nil para introduzir uma lista vazia.
É útil recordar que uma transação corresponde ao par $(origem, (valor, destino))$.

A função getUniqueEntities remove repetições de uma lista de entidades:

$$getUniqueEntities :: [Transaction] \to [String]$$
$$getUniqueEntities = remDup \cdot getEntities$$

getBalance será um catamorfismo de listas de transações que calcula o saldo de uma dada entidade.

$$getBalance :: [Transaction] \to String \to Int$$
$$getBalance\ transactions\ entity = (cataList\ [\underline{0}, addInt \cdot ((delta\ entity) \times id)])\ transactions$$

Esta função foi definida em pointwise uma vez que tem de receber a lista de transações antes da entidade, o que nos é útil para o map que vamos querer executar.

A função delta retorna a diferença de saldo resultante de uma transação para uma dada entidade (se a entidade não aparece na transação, delta será 0):

$$delta :: String \to Transaction \to Int$$
$$delta\ entity\ (a, (v, b))$$
$$\quad | \ entity \equiv a = -v$$
$$\quad | \ entity \equiv b = v$$
$$\quad | \ otherwise = 0$$

Foi necessário definir as funções auxiliares *mapPair* e *zipPair*:

$$mapPair :: (a \to b, [a]) \to [b]$$
$$mapPair\ (f, l) = \mathsf{map}\ \ f\ l$$

$$zipPair :: ([a], [b]) \to [(a, b)]$$
$$zipPair\ (x, y) = zip\ x\ y$$

A função addInt soma um par de Ints:

$$addInt :: (Int, Int) \to Int$$
$$addInt\ (a, b) = a + b$$

A função remDup Remove elementos duplicados de uma lista:

$$remDup :: (Ord\ a) \Rightarrow [a] \to [a]$$
$$remDup = \mathsf{map}\ \ head \cdot group \cdot sort$$

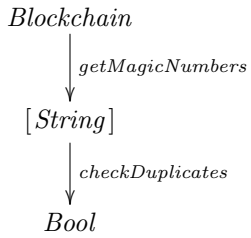
Esta função foi uma descoberta interessante, e tem um desempenho melhor que a equivalente nativa - *nub* é *O(N^2)* enquanto que *remDup* é *O(N log(N))*.

A função *pairToList* cria um par a partir de uma lista com dois elementos:

$$pairToList :: (a, a) \to [a]$$
$$pairToList\ (x, y) = [x, y]$$

isValidMagicNr

Para definir *isValidMagicNr* foi definido um catamorfismo *getMagicNumbers*:



Ao que corresponde o seguinte código Haskell:

$$isValidMagicNr = checkDuplicates \cdot getMagicNumbers$$

O catamorfismo getMagicNumbers retorna a lista de números mágicos de uma Blockchain:

$$getMagicNumbers :: Blockchain \rightarrow [String]$$
$$getMagicNumbers = cataBlockchain\ [cons \cdot \langle \pi_1, nil \rangle, cons \cdot (\pi_1 \times id)]$$

O caso final, $cons \cdot \langle \pi_1, nil \rangle$, retorna uma lista que contém apenas o número mágico do último bloco.

A função checkDuplicates retorna *true* se uma lista não tem elementos duplicados, usando a função *remDup* definida anteriormente.

$$checkDuplicates :: (Ord\ a) \Rightarrow [a] \rightarrow Bool$$
$$checkDuplicates\ x = (remDup\ x) \equiv x$$

Foram também definidas duas Blockchains de teste de maneira a verificar as funcionalidades criadas para a resolução deste primeiro problema:

```
block1 = ("1234",
  (177777, [
    ("Marcos", (200, "Tarracho")),
    ("Antonio", (200, "Joao")),
    ("Tarracho", (200, "Marcos")),
    ("Marcos", (200, "Tarracho"))
  ]))
block2 = ("6789",
  (177888, [
    ("Marcos", (200, "Tarracho")),
    ("Antonio", (200, "Joao"))
  ]))
block3 = ("4444",
  (177888, [
    ("Maria", (200, "Matilde")),
    ("Matilde", (200, "Maria"))
  ]))
testBlockchain1 = Bcs (block1, Bc block2)
testBlockchain2 = Bcs (block3, Bcs (block1, Bc block2))
```

Problema 2

Antes de proceder à resolução das alíneas, foi necessária a definição das funções relativas à manipulação de QTrees.

$toCell\ (a,(b,c)) = Cell\ a\ b\ c$
$toBlock\ (a,(b,(c,d))) = Block\ a\ b\ c\ d$

Aplicando a definição de toBlock e toCell temos:

$inQTree = [toCell, toBlock]$

O out da QTree foi calculado da seguinte forma:

$$
\begin{aligned}
& outQTree \cdot inQTree = id \\
\equiv & \quad \{\ \text{Definição de inQTree}\ \} \\
& outQTree \cdot [toCell, toBlock] = id \\
\equiv & \quad \{\ \text{Lei da Fusão}\ \} \\
& [outQTree \cdot toCell, outQTree \cdot toBlock] = id \\
\equiv & \quad \{\ \text{Universal-+}\ \} \\
& \begin{cases} id \cdot i_1 = outQTree \cdot toCell \\ id \cdot i_2 = outQTree \cdot toBlock \end{cases} \\
\equiv & \quad \{\ \text{Lei 1, 73 e 74}\ \} \\
& \begin{cases} outQTree\ toCell\ (a,(b,c)) = i_1\ (a,(b,c)) \\ outQTree\ toBlock\ (a,(b,(c,d))) = i_2\ (a,(b,(c,d))) \end{cases}
\end{aligned}
$$

□
Resultando na seguinte definição em Haskell:

$outQTree\ (Cell\ a\ b\ c) = i_1\ (a,(b,c))$
$outQTree\ (Block\ a\ b\ c\ d) = i_2\ (a,(b,(c,d)))$

As restantes funções são:

$baseQTree\ g\ f = (g \times id) + (f \times (f \times (f \times f)))$
$recQTree\ f = baseQTree\ id\ f$
$cataQTree\ cata = cata \cdot recQTree\ (cataQTree\ cata) \cdot outQTree$
$anaQTree\ ana = inQTree \cdot recQTree\ (anaQTree\ ana) \cdot ana$
$hyloQTree\ f\ g = cataQTree\ f \cdot anaQTree\ g$
instance $Functor\ QTree$ **where**
  fmap $f = cataQTree\ (inQTree \cdot (baseQTree\ f\ id))$

RotateQTree

Para uma rotação de 90 graus da QTree, temos de reposicionar as células e os blocos. Usou-se um catamorfismo de QTree.

$$
\begin{array}{ccc}
QTree\ A & \xleftarrow{inQTree} & (A,(Int,Int)) + ((QTree\ A)\uparrow 4) \\
{\scriptstyle cataQTree\ r}\downarrow & & \downarrow{\scriptstyle recQTree\ (cataQTree\ r)} \\
QTree\ A & \xleftarrow[r]{} & (A,(Int,Int)) + ((QTree\ A)\uparrow 4)
\end{array}
$$

Por questões de legibilidade, colocou-se *r* ao invés do *either* de *rotateCell* e *rotateBlock*.
Por isso, rotateQTree vem:

$rotateQTree = cataQTree\ [rotateCell, rotateBlock]$
$rotateCell\ (a,(b,c)) = Cell\ a\ c\ b$
$rotateBlock\ (a,(b,(c,d))) = Block\ c\ a\ d\ b$

ScaleQTree

Para redimensionar, temos de multiplicar cada célula pelo fator de multiplicação. Usámos um catamorfismo de QTree. De notar que, devido à definição da *baseQTree*, é possível aplicação do *outQTree* apesar de termos um *Int* de entrada.

$$\begin{array}{ccc} Int \times QTree\ A & \xleftarrow{inQTree} & Int \times (A, (Int, Int)) + ((QTree\ A) \uparrow 4) \\ {\scriptstyle cataQTree\ s}\downarrow & & \downarrow{\scriptstyle recQTree\ (cataQTree\ s)} \\ QTree\ A & \xleftarrow[s]{} & Int \times (A, (Int, Int)) + ((QTree\ A) \uparrow 4) \end{array}$$

Por questões de legibilidade, colocou-se *s* ao invés do *either* de *scaleCell a* e *toBlock*.
Por isso, scaleQTree vem:

$scaleQTree\ a = cataQTree\ [scaleCell\ a, toBlock]$
$scaleCell\ mult\ (a, (b, c)) = Cell\ a\ (mult * b)\ (mult * c)$

InvertQTree

Para inverter as cores de uma QTree, temos de inverter a cor de cada pixel. Usámos um catamorfismo de QTree.

$$\begin{array}{ccc} QTree\ A & \xleftarrow{inQTree} & (A, (Int, Int)) + ((QTree\ A) \uparrow 4) \\ {\scriptstyle cataQTree\ i}\downarrow & & \downarrow{\scriptstyle recQTree\ (cataQTree\ i)} \\ QTree\ A & \xleftarrow[i]{} & (A, (Int, Int)) + ((QTree\ A) \uparrow 4) \end{array}$$

Por questões de legibilidade, colocou-se *i* ao invés do *either* de *invertCell* e *toBlock*.
Por isso, invertQTree vem:

$invertQTree = cataQTree\ [invertCell, toBlock]$
$invertCell\ ((PixelRGBA8\ r\ g\ b\ a), (x, y)) = Cell\ (PixelRGBA8\ (255 - r)\ (255 - g)\ (255 - b)\ a)\ x\ y$

CompressQTree

$$\begin{array}{ccc} QTree\ A & \xleftarrow{inQTree} & (A, (Int, Int)) + ((QTree\ A) \uparrow 4) \\ {\scriptstyle cataQTree\ i}\downarrow & & \downarrow{\scriptstyle id\ x\ id} \\ QTree\ A & \xleftarrow[i]{} & (A, (Int, Int)) + ((QTree\ A) \uparrow 4) \end{array}$$

$compressQTree\ a\ b = (anaQTree\ geneCompress)\ (a, b)$
$geneCompress :: (Int, QTree\ a) \to (a, (Int, Int)) + ((Int, QTree\ a), ((Int, QTree\ a),$
$((Int, QTree\ a), (Int, QTree\ a))))$
$geneCompress\ (x, (Cell\ a\ b\ c)) = i_1\ (a, (b, c))$
$geneCompress\ (x, t@(Block\ a\ b\ c\ d))$
$\quad | \ x \geqslant (depthQTree\ t) = i_1\ ((anyValue\ t), ((\pi_1\ (sizeQTree\ t)), (\pi_2\ (sizeQTree\ t))))$
$\quad | \ otherwise = i_2\ (((x, a), ((x, b), ((x, c), (x, d)))))$

Retorna um valor qualquer de uma QTree. Precisamos disto para a compress, para escolher um valor qualquer para o Block pai ao tirar os filhos.

$anyValue :: QTree\ a \to a$
$anyValue\ (Cell\ a\ b\ c) = a$
$anyValue\ (Block\ a\ b\ c\ d) = anyValue\ a$

OutlineQTree

Para fazer o *outline* da figura é preciso verificar se a célula, após aplicada a função dada, é de valor *True*. Se sim, utiliza-se a função *outlineBlock* que, dado um tamanho de bloco, procede ao contorno do mesmo. O cata apresentado é o que transforma uma *QTree* em a respetiva *QTree* de *Bool*. Após esta conversão, basta utilizar a função *qt2bm* para converter para *Matrix*, tal como pedido.

$$\begin{array}{ccc} QTree\ A & \xleftarrow{inQTree} & f \times (A,(Int,Int)) + ((QTree\ A) \uparrow 4) \\ {\scriptstyle cataQTree\ o}\downarrow & & \downarrow{\scriptstyle recQTree\ (cataQTree\ o)} \\ QTree\ Bool & \xleftarrow[o]{} & f \times (A,(Int,Int)) + ((QTree\ A) \uparrow 4) \end{array}$$

Por questões de legibilidade, colocou-se *o* ao invés do *either* de *outlineCell fun* e *toBlock*.
Por isso, outlineQTree vem:

$$\begin{array}{l} outlineQTree\ fun = qt2bm \cdot (cataQTree\ [outlineCell\ fun, toBlock]) \\ outlineCell\ fun\ (a,(b,c)) = \textbf{if}\ (fun\ a)\ \textbf{then}\ (outlineBlock\ b\ c)\ \textbf{else}\ (Cell\ (fun\ a)\ b\ c) \\ outlineBlock\ a\ b = Block \\ \quad (Block\ (Cell\ True\ 1\ 1) \\ \quad\quad (Cell\ True\ (a-2)\ 1) \\ \quad\quad (Cell\ True\ 1\ (b-2)) \\ \quad\quad (Cell\ False\ (a-2)\ (b-2))) \\ \quad (Cell\ True\ 1\ (b-1)) \\ \quad (Cell\ True\ (a-1)\ 1) \\ \quad (Cell\ True\ 1\ 1) \end{array}$$

Problema 3

Tendo em conta o enunciado do problema 3, decidimos que a estratégia a adotar seria fazer o *split* de *f* e *l* e o *split* de *g* e *s*. Faz-se, então, a seguinte demonstração:

$$\begin{array}{ll} & \left\{ \begin{array}{l} fk\ 0 = 1 \\ fk\ (d+1) = (d+k+1) * fk\ d \end{array} \right. \quad \left\{ \begin{array}{l} lk\ 0 = 1 \\ lk\ (d+1) = lk\ d + 1 \end{array} \right. \\ \equiv & \quad \{\ \text{lei 73 (x2), lei 74 (x4), definiçao de (d+k+1), lei 76 (x2), lei 78}\ \} \\ & \left\{ \begin{array}{l} fk \cdot \underline{0} = \underline{1} \\ fk \cdot \mathsf{succ} = mul \cdot \langle lk, fk \rangle \end{array} \right. \quad \left\{ \begin{array}{l} lk \cdot \underline{0} = \underline{(k+1)} \\ lk \cdot \mathsf{succ} = \mathsf{succ} \cdot lk \end{array} \right. \\ \equiv & \quad \{\ \text{lei eq-+}\ \} \\ & \left\{ \begin{array}{l} [fk \cdot \underline{0}, fk \cdot \mathsf{succ}] = [\underline{1}, mul \cdot \langle lk, fk \rangle] \\ [lk \cdot \underline{0}, lk \cdot \mathsf{succ}] = [\underline{(k+1)}, \mathsf{succ} \cdot lk] \end{array} \right. \\ \equiv & \quad \{\ \text{definição de in dos naturais, lei da fusão (x2) e lei da absorção (x2)}\ \} \\ & \left\{ \begin{array}{l} fk \cdot \mathbf{in} = [\underline{1}, mul] \cdot (id + \langle lk, fk \rangle) \\ lk \cdot \mathbf{in} = [\underline{(k+1)}, \mathsf{succ}] \cdot (id + lk) \end{array} \right. \\ \equiv & \quad \{\ \text{definição de swap e lei do cancelamento-x}\ \} \\ & \left\{ \begin{array}{l} fk \cdot \mathbf{in} = ([\underline{1}, mul] \cdot swap) \cdot (id + \langle fk, lk \rangle) \\ lk \cdot \mathbf{in} = ([\underline{(k+1)}, \mathsf{succ} \cdot \pi_2] \cdot (id + \langle fk, lk \rangle)) \end{array} \right. \\ \equiv & \quad \{\ \text{fokkinga}\ \} \\ & \langle fk, lk \rangle = (\!|\langle [\underline{1}, mul \cdot swap], [\underline{(k+1)}, \mathsf{succ} \cdot \pi_2] \rangle \cdot |\!)_A \end{array}$$

□
Neste passo, faz-se a demonstração respetiva ao *g* e *s*:

$$
\begin{cases} g\ 0 = 1 \\ g\ (d+1) = (d+1) * g\ d \end{cases} \quad \begin{cases} s\ 0 = 1 \\ s\ (d+1) = s\ d + 1 \end{cases}
$$

$\equiv$ { lei 73 (x2), lei 74 (x4), lei 76 (x2), definição de (d+1) e lei 78 }

$$
\begin{cases} g \cdot \underline{0} = \underline{1} \\ g \cdot \mathsf{succ} = mul \cdot \langle s, g \rangle \end{cases} \quad \begin{cases} s \cdot \underline{0} = \underline{1} \\ s \cdot \mathsf{succ} = \mathsf{succ} \cdot s \end{cases}
$$

$\equiv$ { lei eq-+ }

$$
\begin{cases} [g \cdot \underline{0}, g \cdot \mathsf{succ}\ ] = [\underline{1}, mul \cdot \langle s, g \rangle] \\ [s \cdot \underline{0}, s \cdot \mathsf{succ}\ ] = [\underline{1}, \mathsf{succ} \cdot s] \end{cases}
$$

$\equiv$ { definição de in dos naturais, lei da fusão (x2) e lei da absorção (x2) }

$$
\begin{cases} g \cdot \mathbf{in} = [\underline{1}, mul] \cdot (id + \langle s, g \rangle) \\ s \cdot \mathbf{in} = ([\underline{(1)}, \mathsf{succ}\ ] \cdot \pi_1) \cdot (id + \langle s, g \rangle) \end{cases}
$$

$\equiv$ { propriedade do swap (x2) e cancelamento-x }

$$
\begin{cases} g \cdot \mathbf{in} = [\underline{1}, mul \cdot swap] \cdot (id + \langle g, s \rangle) \\ s \cdot \mathbf{in} = ([\underline{1}, \mathsf{succ} \cdot \pi_1 \cdot swap] \cdot (id + \langle g, s \rangle)) \end{cases}
$$

$\equiv$ { fokkinga }

$$
\langle g, s \rangle = (\!|\langle [\underline{1}, mul \cdot swap], [\underline{1}, \mathsf{succ} \cdot \pi_1 \cdot swap] \rangle \cdot |\!)_A
$$

□

Por fim, procedeu-se à combinação dos resultados obtidos através de um *banana-split*, tal como sugerido no enunciado.

$$
\begin{cases} (\!| i \cdot |\!)_A = (\!|\langle [\underline{1}, mul \cdot swap], [\underline{(k+1)}, \mathsf{succ} \cdot \pi_2] \rangle \cdot |\!)_A \\ (\!| j \cdot |\!)_A = (\!|\langle [\underline{1}, mul \cdot swap], [\underline{1}, \mathsf{succ} \cdot \pi_1 \cdot swap] \rangle \cdot |\!)_A \end{cases}
$$

$\equiv$ { lei banana-split }

$$
\langle (\!| i \cdot |\!)_A, (\!| j \cdot |\!)_A \rangle = (\!|\langle [\underline{1}, mul \cdot swap], [\underline{(k+1)}, \mathsf{succ} \cdot \pi_2] \rangle \times \langle [\underline{1}, mul \cdot swap], [\underline{1}, \mathsf{succ} \cdot \pi_1 \cdot swap] \rangle \cdot \langle F\ \pi_1, F\ \pi_2 \rangle \cdot |\!)_A
$$

$\equiv$ { lei da troca }

$$
\langle (\!| i \cdot |\!)_A, (\!| j \cdot |\!)_A \rangle = (\!|[\langle \underline{1}, \underline{(k+1)} \rangle, \langle mul \cdot swap, \mathsf{succ} \cdot \pi_2 \rangle] \times [\langle \underline{1}, \underline{1} \rangle, \langle mul \cdot swap, \mathsf{succ} \cdot \pi_1 \cdot swap \rangle] \cdot \langle F\ \pi_1, F\ \pi_2 \rangle \cdot |\!)_A
$$

$\equiv$ { conforme 3.90 a 3.95 dos apontamentos / lei 11) }

$$
\langle (\!| i \cdot |\!)_A, (\!| j \cdot |\!)_A \rangle = (\!|\langle [\langle \underline{1}, \underline{(k+1)} \rangle, \langle mul \cdot swap, \mathsf{succ} \cdot \pi_2 \rangle] \cdot F\ \pi_1, [\langle \underline{1}, \underline{(k+1)} \rangle, \langle mul \cdot swap, \mathsf{succ} \cdot \pi_2 \rangle] \cdot F\ \pi_2 \rangle \cdot |\!)_A
$$

$\equiv$ { lei da troca }

$$
\langle (\!| i \cdot |\!)_A, (\!| j \cdot |\!)_A \rangle = (\!|[\langle \langle \underline{1}, \underline{(k+1)} \rangle, \langle \underline{1}, \underline{1} \rangle \rangle, \langle \langle mul \cdot swap, \mathsf{succ} \cdot \pi_2 \rangle \cdot \pi_1, \langle mul \cdot swap, \mathsf{succ} \cdot \pi_1 \cdot swap \rangle \cdot \pi_2 \rangle] \cdot |\!)_A)
$$

$\equiv$ { definição de for b i }

$$
\begin{cases} b = \langle \langle mul \cdot swap, \mathsf{succ} \cdot \pi_2 \rangle \cdot \pi_1, \langle mul \cdot swap, \mathsf{succ} \cdot \pi_1 \cdot swap \rangle \cdot \pi_2 \rangle \\ i = \langle \langle \underline{1}, \underline{(k+1)} \rangle, \langle \underline{1}, \underline{1} \rangle \rangle \end{cases}
$$

□

A *base* foi extraída diretamente do cálculo anterior, enquanto que o *loop* precisou de uns ajustes de tipos internos, os quais foram realizados através das funções *tuple* e *untuple*.

$$
\begin{aligned}
&untuple\ ((a,b),(c,d)) = (a,b,c,d) \\
&tuple\ (a,b,c,d) = ((a,b),(c,d)) \\
&base = untuple \cdot \langle \langle \underline{1}, \mathsf{succ} \rangle, \langle \underline{1}, \underline{1} \rangle \rangle \\
&loop = untuple \cdot \langle \langle mul \cdot swap \cdot \pi_1, \mathsf{succ} \cdot \pi_2 \cdot \pi_1 \rangle \cdot tuple, \langle mul \cdot swap \cdot \pi_2, \mathsf{succ} \cdot \pi_1 \cdot swap \cdot \pi_2 \rangle \cdot tuple \rangle
\end{aligned}
$$

Problema 4

De maneira a resolver o problema 4, foi necessário definir as funções que facilitam a manipulação do tipo de dados *FTree*:

$inFTree$ usa os construtores de $FTree$, usando uma função auxiliar $toComp$ de maneira a poder converter um par recebido.

$$inFTree = [Unit, toComp]$$
$$toComp\ (a,(b,c)) = Comp\ a\ b\ c$$

$outFTree$ foi derivada de uma maneira semelhante à out das Blockchain (Problema 1):

$$outFTree\ (Unit\ a) = i_1\ a$$
$$outFTree\ (Comp\ a\ b\ c) = i_2\ (a,(b,c))$$

As restantes funções são:

$$baseFTree\ f\ g\ h = g + (f \times (h \times h))$$
$$recFTree\ f = baseFTree\ id\ id\ f$$
$$cataFTree\ g = g \cdot (recFTree\ (cataFTree\ g)) \cdot outFTree$$
$$anaFTree\ g = inFTree \cdot (recFTree\ (anaFTree\ g)) \cdot g$$
$$hyloFTree\ f\ g = cataFTree\ f \cdot anaFTree\ g$$

A partir da lei 47 (def-map-cata) foi definido:

$$\textbf{instance}\ Bifunctor\ FTree\ \textbf{where}$$
$$bimap\ f\ g = cataFTree\ (inFTree \cdot (baseFTree\ f\ g\ id))$$

generatePTree

generatePTree deve gerar uma árvore de Pitágoras para uma dada ordem, sendo definida como um anamorfismo.

Para este efeito, o grupo partiu do diagrama do anamorfismo de *FTree*, que pode ser usado em vez de *PTree* simplesmente porque:

type PTree = FTree Square Square

O diagrama é o seguinte:

$$\begin{array}{ccc} FTree\ A\ B & \xleftarrow{\textbf{in}} & B + A \times (FTree\ A\ B \times FTree\ A\ B) \\ {\scriptstyle f}\uparrow & & \uparrow{\scriptstyle id+id\times(f\times f)} \\ C & \xrightarrow[g]{} & B + A \times (C \times C) \end{array}$$

Neste diagrama, A e B ambos representam o tipo *Float*, para que a *FTree* seja convertível para uma *PTree*.

Na primeira tentativa, o grupo definiu um anamorfismo que recebe um *Int* igual ao rank da árvore de Pitágoras pretendida. Esse *Int* iria diminuindo, e o anamorfismo pararia quando este fosse igual a zero.

O gene deste primeiro anamorfismo era:

$$genePTree = (id + \langle \pi_2, \langle \pi_1, \pi_1 \rangle \rangle) \cdot (id + (pred \times id)) \cdot (id + \langle id, rankToMultiplier \rangle) \cdot (fromIntegral + id) \cdot oneToLeft$$

O que aconteceu foi que o anamorfismo criado gerava uma árvore invertida, pelo que seria necessário começar com o inteiro a zero e parar de iterar quando este fosse igual ao rank pretendido.

Para isso foi criado um segundo anamorfismo que recebe um par *(Int, Int)*, onde o primeiro inteiro representa o rank da iteração atual, e o segundo representa o rank final pretendido.

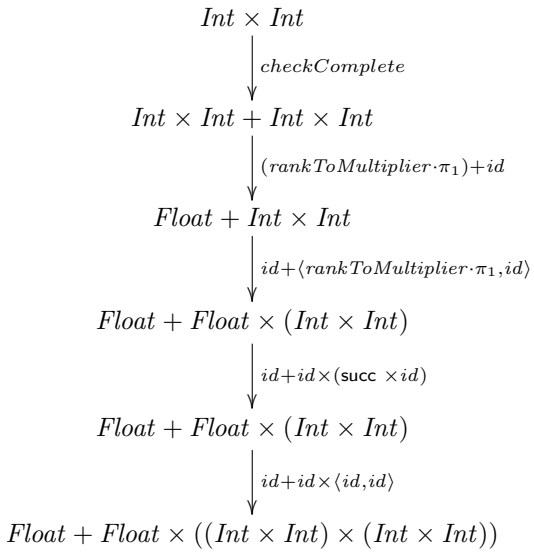
Ao encaixar uma função $\langle \underline{1}, id \rangle$ antes do anamorfismo, consegue-se manter esta mudança invisível para o utilizador. Ficamos então com uma função *generatePTree*:

$$generatePTree = anaFTree\ genePTree \cdot \langle \underline{0}, id \rangle$$

Em que *genePTree* é o gene do anamorfismo:

$$genePTree = (id + (id \times \langle id, id \rangle)) \cdot (id + (id \times (\mathsf{succ}\ \times id))) \cdot (id + \langle rankToMultiplier \cdot \pi_1, id \rangle) \cdot ((rankToMultiplier \cdot \pi_1) + id) \cdot checkComplete$$

Este gene é representado no seguinte diagrama:



De seguida são definidas as funções usadas pelo anamorfismo:

rankToMultiplier retorna o multiplicador de uma PTree para um dado Rank. Por exemplo, o multiplicador de ordem 0 é 1, o de ordem 1 é $\frac{\sqrt{2}}{2}$, e o de ordem 2 é $(\frac{\sqrt{2}}{2})^2$.

```
rankToMultiplier :: Int → Float
rankToMultiplier a = (((sqrt 2) / 2) ↑ a)
```

checkComplete executa i_1 sobre um par de inteiros se estes forem iguais, ou i_2 se forem diferentes. Esta função é importante para determinar quando chegamos à última iteração do anamorfismo.

```
checkComplete :: (Int, Int) → (Int, Int) + (Int, Int)
checkComplete (a, b)
  | b < 0 = i1 (a, 0)
  | a ≡ b = i1 (a, b)
  | otherwise = i2 (a, b)
```

O caso

$$b < 0 = i_1\ (a, 0)$$

evita um loop infinito no caso de ser pedida uma *PTree* com rank negativo.

drawPTree

O grupo não definiu a função drawPTree.

$$drawPTree = \bot$$

Problema 5

Para a realização deste exercício, foi necessário compreender qual a função de *singletonbag* e *muB*. A primeira permite, tendo um determinado objeto, inseri-lo num dado *Bag*. Quanto à segunda, dado um determinado *Bag* com *Bags* e o respetivo número dos mesmos, deve permitir criar apenas um *Bag* com o conteúdo de todos os interiores.

singletonbag

Para a realização de *singletonbag*, devemos primeiro criar um tuplo com o elemento que recebemos, e o respetivo número de elementos, ou seja, um.

$$A \xrightarrow{toTuple} A \times 1$$

Depois de ter o tuplo, este deve ser inserido dentro de uma lista, uma vez que o construtor de *Bag* recebe tal. Para isso, usou-se *singl*. Posteriormente, basta aplicar o construtor *B*. Abaixo, mostram-se as definições realizadas.

$$singletonbag = B \cdot singl \cdot toTuple$$
$$toTuple\ a = (a, 1)$$

μ

Para fazer o μ, tendo em conta o seu objetivo, optou-se por, desde logo, realizar um *fmap unB*. Isto vai permitir fazer *unBag* dos *Bags* interiores do *Bag* fornecido. Desta forma, dentro do *Bag* inicial vamos ter tuplos (cujo primeiro elemento é uma lista de elementos depois do *unBag*, e o segundo é o respetivo número de sacos iguais existentes).

$$Bag\ (Bag\ A, Int)^{*} \xrightarrow{\mathsf{fmap}\ unB} Bag\ ((A, Int)^{*}, Int)^{*}$$

Depois de termos a *Bag* com tuplos, em que o primeiro elemento do mesmo é uma lista de tuplos do *Bag* interior, e em que o segundo é o número respetivo de *Bags* existentes à *priori*, é necessário retirar este último. Para retirar sem perder informação, devemos multiplicar o número de elementos de cada tipo em cada um dos *bags* interiores, pelo respetivo número de *bags* daquele tipo. Isso foi feito com recurso a um *map multBags*, após um *unBag*.

Faz-se então o *unB*:

$$Bag\ ((A, Int)^{*}, Int)^{*} \xrightarrow{unB} ((A, Int)^{*}, Int)^{*}$$

E agora aplicar a função *multBags* a toda a lista, através de um *map*.

$$((A, Int)^{*}, Int)^{*} \xrightarrow{\mathsf{map}\ \ multBags} ((A, Int)^{*})^{*}$$

Tendo feito todos os passos anteriores, basta concatenar as listas numa só, através da função *concat*. Naturalmente, posteriormente, aplica-se o construtor *B*, obtendo-se a *Bag* final.

$$\mu = B \cdot concat \cdot (\mathsf{map}\ \ multBags) \cdot unB \cdot (\mathsf{fmap}\ unB)$$
$$multBags :: ([(a, Int)], Int) \to [(a, Int)]$$
$$multBags\ ([], c) = []$$
$$multBags\ (((a, b) : tail), c) = [(a, b * c)] +\!\!+ (multBags\ (tail, c))$$

dist

O *dist*, após recebido um *Bag* tem, naturalmente, que fazer o seu *unBag*. Tendo a lista de tuplos, em que o primeiro elemento é um elemento específico e o segundo o número de existentes desse elemento. Feito isto, é usada a *marbleReplication* (com recurso a *map*), para transformar tuplos de A $\times$ Int em uma lista de A, como se demonstra.

$$(A, Int)^{*} \xrightarrow{\mathsf{map}\ \ marbleReplication} A^{**}$$

Tendo agora uma lista com as listas formadas após replicação, resta fazer o concat das mesmas. Esta lista fica assim pronta a ser processada pela função *uniform*, obtendo-se a distribuição pedida.

$$dist = uniform \cdot concat \cdot \mathsf{map}\ \ marbleReplication \cdot unB$$
$$marbleReplication = \widehat{replicate} \cdot swap$$