Cálculo de Programas
Trabalho Prático
MiEI+LCC — Ano Lectivo de 2016/17

Departamento de Informática
Universidade do Minho

Junho de 2017

Grupo nr.	99 (preencher)
a78582	Hugo Brandão
a78296	Sérgio Alves
a78218	Tiago Alves

Conteúdo

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 ao desenvolvimento de programas funcionais na linguagem Haskell.

O presente trabalho tem por objectivo concretizar na prática os objectivos da disciplina, colocando os alunos perante problemas de programação que deverão ser abordados composicionalmente e 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" [3], cujo princípio base é o seguinte:

> *Um programa e a sua documentação devem coincidir.*

Por outras palavras, o código fonte e a sua documentação deverão constar do mesmo documento (ficheiro).

O ficheiro `cp1617t.pdf` que está a ler é já um exemplo de programação literária: foi gerado a partir do texto fonte `cp1617t.lhs`[1] que encontrará no material pedagógico desta disciplina descompactando o ficheiro `cp1617t.zip` e executando

```
lhs2TeX cp1617t.lhs > cp1617t.tex
pdflatex cp1617t
```

em que `lhs2tex` é um pre-processador que faz "pretty printing" de código Haskell em LaTeX e que deve desde já instalar a partir do endereço

> https://hackage.haskell.org/package/lhs2tex.

Por outro lado, o mesmo ficheiro `cp1617t.lhs` é executável e contém o "kit" básico, escrito em Haskell, para realizar o trabalho. Basta executar

```
ghci cp1617t.lhs
```

para ver que assim é:

```
GHCi, version 8.0.2: http://www.haskell.org/ghc/  :? for help
[ 1 of 11] Compiling Show             ( Show.hs, interpreted )
[ 2 of 11] Compiling ListUtils        ( ListUtils.hs, interpreted )
[ 3 of 11] Compiling Probability      ( Probability.hs, interpreted )
[ 4 of 11] Compiling Cp               ( Cp.hs, interpreted )
[ 5 of 11] Compiling Nat              ( Nat.hs, interpreted )
[ 6 of 11] Compiling List             ( List.hs, interpreted )
[ 7 of 11] Compiling LTree            ( LTree.hs, interpreted )
[ 8 of 11] Compiling St               ( St.hs, interpreted )
[ 9 of 11] Compiling BTree            ( BTree.hs, interpreted )
[10 of 11] Compiling Exp              ( Exp.hs, interpreted )
[11 of 11] Compiling Main             ( cp1617t.lhs, interpreted )
Ok, modules loaded: BTree, Cp, Exp, LTree, List, ListUtils, Main, Nat,
Probability, Show, St.
```

O facto de o interpretador carregar as bibliotecas do material pedagógico da disciplina, entre outras, deve-se ao facto de, neste mesmo sítio do texto fonte, se ter inserido o seguinte código Haskell:

$\{\text{-\# OPTIONS}_G HC - XNPlusKPatterns\# -\}$
import *Cp*

[1] O suffixo 'lhs' quer dizer *literate Haskell*.

```
import List
import Nat
import Exp
import BTree
import LTree
import St
import Probability hiding (cond)
import Data.List
import Test.QuickCheck hiding ((×))
import System.Random hiding ⟨·,·⟩
import GHC.IO.Exception
import System.IO.Unsafe
```

Abra o ficheiro `cp1617t.lhs` no seu editor de texto preferido e verifique que assim é: todo o texto que se encontra dentro do ambiente

```
\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 equilibrada e 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 suas 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 cp1617t.aux
makeindex cp1617t.idx
```

e recompilar o texto como acima se indicou. Dever-se-á ainda instalar o utilitário QuickCheck [2] que ajuda a validar programas em Haskell.

Problema 1

O controlador de um processo físico baseia-se em dezenas de sensores que enviam as suas leituras para um sistema central, onde é feito o respectivo processamento.

Verificando-se que o sistema central está muito sobrecarregado, surgiu a ideia de equipar cada sensor com um microcontrolador que faça algum pré-processamento das leituras antes de as enviar ao sistema central. Esse tratamento envolve as operações (em vírgula flutuante) de soma, subtracção, multiplicação e divisão.

Há, contudo, uma dificuldade: o código da divisão não cabe na memória do microcontrolador, e não se pretende investir em novos microcontroladores devido à sua elevada quantidade e preço.

Olhando para o código a replicar pelos microcontroladores, alguém verificou que a divisão só é usada para calcular inversos, $\frac{1}{x}$. Calibrando os sensores foi possível garantir que os valores a inverter estão entre $1 < x < 2$, podendo-se então recorrer à série de Maclaurin

$$\frac{1}{x} = \sum_{i=0}^{\infty} (1 - x)^i$$

para calcular $\frac{1}{x}$ sem fazer divisões. Seja então

$$inv\ x\ n = \sum_{i=0}^{n} (1 - x)^i$$

[2] Para uma breve introdução ver e.g. https://en.wikipedia.org/wiki/QuickCheck.

a função que aproxima $\frac{1}{x}$ com n iterações da série de MacLaurin. Mostre que $inv\ x$ é um ciclo-for, implementando-o em Haskell (e opcionalmente em C). Deverá ainda apresentar testes em QuickCheck que verifiquem o funcionamento da sua solução. (**Sugestão:** inspire-se no problema semelhante relativo à função ns da secção 3.16 dos apontamentos [4].)

Problema 2

Se digitar *man wc* na shell do Unix (Linux) obterá:

```
NAME
     wc -- word, line, character, and byte count

SYNOPSIS
     wc [-clmw] [file ...]

DESCRIPTION
    The wc utility displays the number of lines, words, and bytes contained in
    each input file,  or standard input (if no file is specified) to the stan-
    dard  output.  A line is defined as  a string of characters delimited by a
    <newline> character.  Characters beyond the final <newline> character will
    not be included in the line count.
    (...)
    The following options are available:
    (...)
        -w   The number of words in each input file is written to the standard
             output.
    (...)
```

Se olharmos para o código da função que, em C, implementa esta funcionalidade [2] e nos focarmos apenas na parte que implementa a opção `-w`, verificamos que a poderíamos escrever, em Haskell, da forma seguinte:

$$
\begin{array}{l}
wc\_w :: [\,Char\,] \rightarrow Int \\
wc\_w\ [\,] = 0 \\
wc\_w\ (c : l) = \\
\quad \mathbf{if}\ \neg\ (sep\ c) \wedge lookahead\_sep\ l \\
\quad \mathbf{then}\ wc\_w\ l + 1 \\
\quad \mathbf{else}\ wc\_w\ l \\
\qquad \mathbf{where} \\
\qquad\quad sep\ c = (c \equiv \texttt{' '} \vee c \equiv \texttt{'\textbackslash n'} \vee c \equiv \texttt{'\textbackslash t'}) \\
\qquad\quad lookahead\_sep\ [\,] = True \\
\qquad\quad lookahead\_sep\ (c : l) = sep\ c
\end{array}
$$

Re-implemente esta função segundo o modelo *worker/wrapper* onde *wrapper* deverá ser um catamorfismos de listas. Apresente os cálculos que fez para chegar a essa sua versão de wc_w e inclua testes em QuickCheck que verifiquem o funcionamento da sua solução. (**Sugestão:** aplique a lei de recursividade múltipla às funções wc_w e $lookahead\_sep$.)

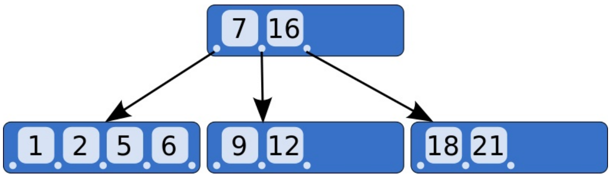
Problema 3

Uma "B-tree"é uma generalização das árvores binárias do módulo BTree a mais do que duas sub-árvores por nó:

$$\mathbf{data}\ \textsf{B-tree}\ a = Nil \mid Block\ \{\,leftmost :: \textsf{B-tree}\ a, block :: [\,(a, \textsf{B-tree}\ a)\,]\,\}\ \mathbf{deriving}\ (Show, Eq)$$

Por exemplo, a B-tree[3]

[3]Créditos: figura extraída de `https://en.wikipedia.org/wiki/B-tree`.



é representada no tipo acima por:

$$\begin{array}{l}
t = \mathit{Block}\ \{ \\
\quad \mathit{leftmost} = \mathit{Block}\ \{ \\
\quad\quad \mathit{leftmost} = \mathit{Nil}, \\
\quad\quad \mathit{block} = [(1, \mathit{Nil}), (2, \mathit{Nil}), (5, \mathit{Nil}), (6, \mathit{Nil})]\}, \\
\quad \mathit{block} = [\\
\quad\quad (7, \mathit{Block}\ \{ \\
\quad\quad\quad \mathit{leftmost} = \mathit{Nil}, \\
\quad\quad\quad \mathit{block} = [(9, \mathit{Nil}), (12, \mathit{Nil})]\}), \\
\quad\quad (16, \mathit{Block}\ \{ \\
\quad\quad\quad \mathit{leftmost} = \mathit{Nil}, \\
\quad\quad\quad \mathit{block} = [(18, \mathit{Nil}), (21, \mathit{Nil})]\}) \\
\quad\quad]\}
\end{array}$$

Pretende-se, neste problema:

1. Construir uma biblioteca para o tipo B-tree da forma habitual (in + out; ana + cata + hylo; instância na classe $\mathit{Functor}$).
2. Definir como um catamorfismo a função $\mathit{inordB\_tree}$ :: B-tree $t \rightarrow [t]$ que faça travessias "inorder"de árvores deste tipo.
3. Definir como um catamorfismo a função $\mathit{largestBlock}$ :: B-tree $a \rightarrow \mathit{Int}$ que detecta o tamanho do maior bloco da árvore argumento.
4. Definir como um anamorfismo a função $\mathit{mirrorB\_tree}$ :: B-tree $a \rightarrow$ B-tree a que roda a árvore argumento de 180º
5. Adaptar ao tipo B-tree o hilomorfismo "quick sort"do módulo BTree. O respectivo anamorfismo deverá basear-se no gene $\mathit{lsplitB\_tree}$ cujo funcionamento se sugere a seguir:

   $\mathit{lsplitB\_tree}\ [\,] = i_1\ ()$
   $\mathit{lsplitB\_tree}\ [7] = i_2\ ([\,], [(7, [\,])])$
   $\mathit{lsplitB\_tree}\ [5, 7, 1, 9] = i_2\ ([1], [(5, [\,]), (7, [9])])$
   $\mathit{lsplitB\_tree}\ [7, 5, 1, 9] = i_2\ ([1], [(5, [\,]), (7, [9])])$

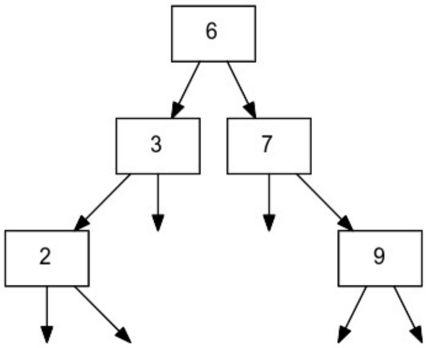
6. A biblioteca Exp permite representar árvores-expressão em formato DOT, que pode ser lido por aplicações como por exemplo Graphviz, produzindo as respectivas imagens. Por exemplo, para o caso de árvores BTree, se definirmos

   -- dotBTree :: Show a =¿ BTree a -¿ IO ExitCode
   -- dotBTree = dotpict . bmap Nothing (Just . show) . cBTree2Exp

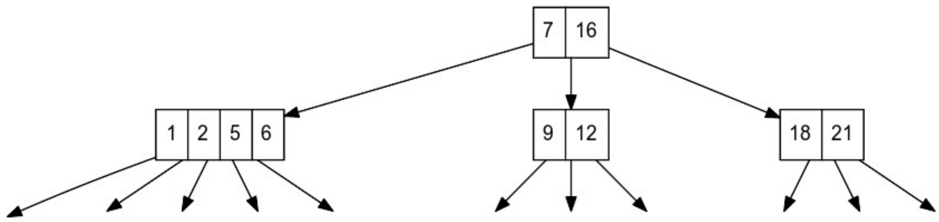
   executando $\mathit{dotBTree}\ t$ para

   $$t = \mathit{Node}\ (6, (\mathit{Node}\ (3, (\mathit{Node}\ (2, (\mathit{Empty}, \mathit{Empty})), \mathit{Empty})), \mathit{Node}\ (7, (\mathit{Empty}, \mathit{Node}\ (9, (\mathit{Empty}, \mathit{Empty}))))))$$

   obter-se-á a imagem



Escreva de forma semelhante uma função $dotB\_tree$ que permita mostrar em Graphviz[4] árvores B-tree tal como se ilustra a seguir,



para a árvora dada acima.

Problema 4

Nesta disciplina estudaram-se funções mutuamente recursivas e como lidar com elas. Os tipos indutivos de dados podem, eles próprios, ser mutuamente recursivos. Um exemplo dessa situação são os chamados L-Systems.

Um L-System é um conjunto de regras de produção que podem ser usadas para gerar padrões por re-escrita sucessiva, de acordo com essas mesmas regras. Tal como numa gramática, há um axioma ou símbolo inicial, de onde se parte para aplicar as regras. Um exemplo célebre é o do crescimento de algas formalizado por Lindenmayer[5] no sistema:

Variáveis: A e B

Constantes: nenhuma

Axioma: A

Regras: $A \to A\ B, B \to A$.

Quer dizer, em cada iteração do "crescimento"da alga, cada A deriva num par $A\ B$ e cada B converte-se num A. Assim, ter-se-á, onde n é o número de iterações desse processo:

- $n = 0$: A
- $n = 1$: $A\ B$
- $n = 2$: $A\ B\ A$
- $n = 3$: $A\ B\ A\ A\ B$
- etc

[4] Como alternativa a instalar Graphviz, podem usar WebGraphviz num browser.
[5] Ver `https://en.wikipedia.org/wiki/Aristid_Lindenmayer`.

Este L-System pode codificar-se em Haskell considerando cada variável um tipo, a que se adiciona um caso de paragem para poder expressar as sucessivas iterações:

$$\begin{array}{l}\textbf{type}\ Algae = A\\ \textbf{data}\ A = \mathrm{NA} \mid A\ A\ B\ \textbf{deriving}\ Show\\ \textbf{data}\ B = \mathrm{NB} \mid B\ A\ \textbf{deriving}\ Show\end{array}$$

Observa-se aqui já que A e B são mutuamente recursivos. Os isomorfismos in/out são definidos da forma habitual:

$$\begin{array}{l} inA :: 1 + A \times B \to A\\ inA = [\underline{\mathrm{NA}}, \widehat{A}]\\ outA :: A \to 1 + A \times B\\ outA\ \mathrm{NA} = i_1\ ()\\ outA\ (A\ a\ b) = i_2\ (a, b)\\ inB :: 1 + A \to B\\ inB = [\underline{\mathrm{NB}}, B]\\ outB :: B \to 1 + A\\ outB\ \mathrm{NB} = i_1\ ()\\ outB\ (B\ a) = i_2\ a\end{array}$$

O functor é, em ambos os casos, $\mathsf{F}\ X = 1 + X$. Contudo, os catamorfismos de A têm de ser estendidos com mais um gene, de forma a processar também os B,

$$\begin{array}{l} (\!|\cdot\ \cdot|\!)_A :: (1 + c \times d \to c) \to (1 + c \to d) \to A \to c\\ (\!|ga\ gb|\!)_A = ga \cdot (id + (\!|ga\ gb|\!)_A \times (\!|ga\ gb|\!)_B) \cdot outA\end{array}$$

e a mesma coisa para os Bs:

$$\begin{array}{l} (\!|\cdot\ \cdot|\!)_B :: (1 + c \times d \to c) \to (1 + c \to d) \to B \to d\\ (\!|ga\ gb|\!)_B = gb \cdot (id + (\!|ga\ gb|\!)_A) \cdot outB\end{array}$$

Pretende-se, neste problema:

1. A definição dos anamorfimos dos tipos A e B.
2. A definição da função

   $$generateAlgae :: Int \to Algae$$

   como anamorfismo de $Algae$ e da função

   $$showAlgae :: Algae \to String$$

   como catamorfismo de $Algae$.
3. Use QuickCheck para verificar a seguinte propriedade:

   $$\mathsf{length}\ \cdot showAlgae \cdot generateAlgae = fib \cdot \mathsf{succ}$$

Problema 5

O ponto de partida deste problema é um conjunto de equipas de futebol, por exemplo:

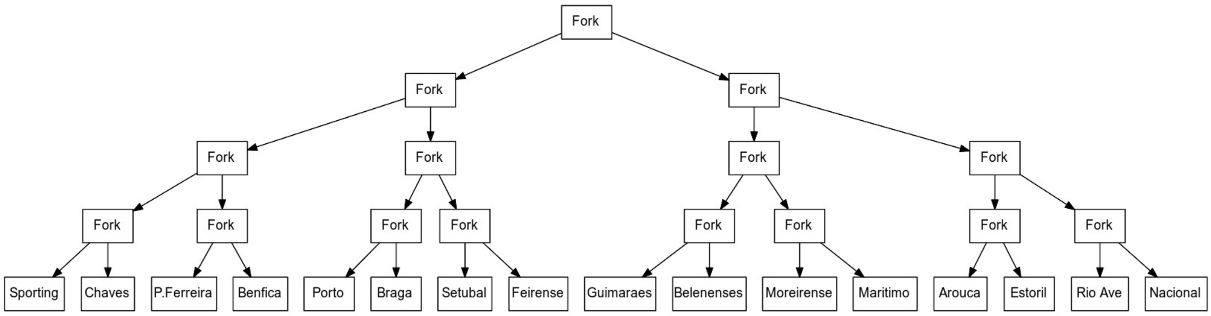
```
equipas :: [Equipa]
equipas = [
  "Arouca", "Belenenses", "Benfica", "Braga", "Chaves", "Feirense",
  "Guimaraes", "Maritimo", "Moreirense", "Nacional", "P.Ferreira",
  "Porto", "Rio Ave", "Setubal", "Sporting", "Estoril"
  ]
```

Assume-se que há uma função $f\ (e_1, e_2)$ que dá — baseando-se em informação acumulada historicamente, e.g. estatística — qual a probabilidade de e_1 ou e_2 ganharem um jogo entre si.[6] Por exemplo, f (`"Arouca"`, `"Braga"`) poderá dar como resultado a distribuição

$Arouca$ ▬ 28.6% $Braga$
▬ 71.4%

indicando que há 71.4% de probabilidades de `"Braga"` ganhar a `"Arouca"`.

Para lidarmos com probabilidades vamos usar o mónade Dist a que vem descrito no apêndice A e que está implementado na biblioteca Probability [1] — ver definição (1) mais adiante. A primeira parte do problema consiste em sortear *aleatoriamente* os jogos das equipas. O resultado deverá ser uma LTree contendo, nas folhas, os jogos da primeira eliminatória e cujos nós indicam quem joga com quem (vencendo), à medida que a eliminatória prossegue:



A segunda parte do problema consiste em processar essa árvore usando a função

$$jogo :: (Equipa, Equipa) \rightarrow \mathsf{Dist}\ Equipa$$

que foi referida acima. Essa função simula um qualquer jogo, como foi acima dito, dando o resultado de forma probabilística. Por exemplo, para o sorteio acima e a função *jogo* que é dada neste enunciado[7], a probabilidade de cada equipa vir a ganhar a competição vem dada na distribuição seguinte:

$Porto$
▬ 21.4% $Benfica$
▬ 9.4% $Braga$
▬ 4.9% $Maritimo$
▬ 3.5% $Rio\ Ave$
▬ 1.9% $P.Ferreira$
▬ 1.4% $Estoril$
▬ 1.4% $Feirense$
▬ 0.4%

▬ 21.7% $Sporting$
▬ 19.0% $Guimaraes$
▬ 5.1% $Nacional$
▬ 4.1% $Belenenses$
▬ 2.3% $Moreirense$
▬ 1.4% $Arouca$
▬ 1.4% $Setubal$
▬ 0.7% $Chaves$

Assumindo como dada e fixa a função *jogo* acima referida, juntando as duas partes obteremos um *hilomorfismo* de tipo $[Equipa] \rightarrow \mathsf{Dist}\ Equipa$,

$$quem\_vence :: [Equipa] \rightarrow \mathsf{Dist}\ Equipa$$
$$quem\_vence = eliminatoria \cdot sorteio$$

com características especiais: é aleatório no anamorfismo (sorteio) e probabilístico no catamorfismo (eliminatória).

O anamorfismo $sorteio :: [Equipa] \rightarrow \mathsf{LTree}\ Equipa$ tem a seguinte arquitectura, [8]

$$sorteio = anaLTree\ lsplit \cdot envia \cdot permuta$$

reutilizando o anamorfismo do algoritmo de "merge sort", da biblioteca LTree, para construir a árvore de jogos a partir de uma permutação aleatória das equipas gerada pela função genérica

$$permuta :: [a] \rightarrow \mathsf{IO}\ [a]$$

A presença do mónade de IO tem a ver com a geração de números aleatórios[9].

[6] Tratando-se de jogos eliminatórios, não há lugar a empates.

[7] Pode, se desejar, criar a sua própria função *jogo*, mas para efeitos de avaliação terá que ser usada a que vem dada neste enunciado. Uma versão de *jogo* realista teria que ter em conta todas as estatísticas de jogos entre as equipas em jogo, etc etc.

[8] A função *envia* não é importante para o processo; apenas se destina a simplificar a arquitectura monádica da solução.

[9] Quem estiver interessado em detalhes deverá consultar System.Random.

1. Defina a função monádica $permuta$ sabendo que tem já disponível

   $$getR :: [a] \rightarrow \mathsf{IO}\ (a, [a])$$

   $getR\ x$ dá como resultado um par (h, t) em que h é um elemento de x tirado à sorte e t é a lista sem esse elemento – mas esse par vem encapsulado dentro de IO.

2. A segunda parte do exercício consiste em definir a função monádica

   $$eliminatoria :: \mathsf{LTree}\ Equipa \rightarrow \mathsf{Dist}\ Equipa$$

   que, assumindo já disponível a função $jogo$ acima referida, dá como resultado a distribuição de equipas vencedoras do campeonato.

Sugestão: inspire-se na secção 4.10 (*'Monadification' of Haskell code made easy*) dos apontamentos [4].

Referências

[1] M. Erwig and S. Kollmansberger. Functional pearls: Probabilistic functional programming in Haskell. *J. Funct. Program.*, 16:21–34, January 2006.

[2] B.W. Kernighan and D.M. Richtie. *The C Programming Language*. Prentice Hall, Englewood Cliffs, N.J., 1978.

[3] D.E. Knuth. *Literate Programming*. CSLI Lecture Notes Number 27. Stanford University Center for the Study of Language and Information, Stanford, CA, USA, 1992.

[4] J.N. Oliveira. *Program Design by Calculation*, 2008. Draft of textbook in preparation. viii+297 pages. Informatics Department, University of Minho.

Anexos

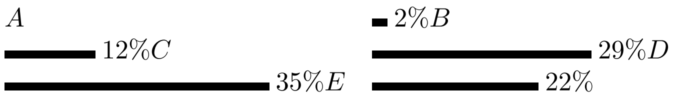
A Mónade para probabilidades e estatística

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

$$\textbf{newtype}\ \mathsf{Dist}\ a = D\ \{ \mathit{unD} :: [(a, \mathit{ProbRep})] \} \tag{1}$$

em que $\mathit{ProbRep}$ 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

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

que o GHCi mostrará assim:

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

É possível definir geradores de distribuições, por exemplo distribuições *uniformes*,

$$d2 = \mathit{uniform}\ (\mathit{words}\ \texttt{"Uma frase de cinco palavras"})$$

isto é

```
     "Uma"  20.0%
   "cinco"  20.0%
      "de"  20.0%
   "frase"  20.0%
"palavras"  20.0%
```

distribuição *normais*, eg.

$$d3 = \mathit{normal}\ [10 \mathinner{..} 20]$$

etc.[10]

Dist forma um **mónade** cuja unidade é $\mathit{return}\ a = D\ [(a, 1)]$ e cuja composição de Kleisli é (simplificando a notação)

$$(f \bullet g)\ a = [(y, q * p) \mid (x, p) \leftarrow g\ a, (y, q) \leftarrow f\ x]$$

em que $g : A \rightarrow \mathsf{Dist}\ B$ e $f : B \rightarrow \mathsf{Dist}\ C$ são funções **monádicas** que representam *computações probabilísticas*.

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.

[10] Para mais detalhes ver o código fonte de Probability, que é uma adaptação da biblioteca PHP ("Probabilistic Functional Programming"). Para quem quiser souber mais recomenda-se a leitura do artigo [1].

B Definições auxiliares

São dadas: a função que simula jogos entre equipas,

$$
\begin{array}{l}
\textbf{type}\ \mathit{Equipa} = \mathit{String} \\
\mathit{jogo} :: (\mathit{Equipa}, \mathit{Equipa}) \rightarrow \mathsf{Dist}\ \mathit{Equipa} \\
\mathit{jogo}\ (e_1, e_2) = D\ [(e_1, 1 - \mathit{r1}\ /\ (\mathit{r1} + \mathit{r2})), (e_2, 1 - \mathit{r2}\ /\ (\mathit{r1} + \mathit{r2}))]\ \textbf{where} \\
\quad \mathit{r1} = \mathit{rank}\ e_1 \\
\quad \mathit{r2} = \mathit{rank}\ e_2 \\
\quad \mathit{rank} = \mathit{pap}\ \mathit{ranks} \\
\quad \mathit{ranks} = [\\
\qquad (\texttt{"Arouca"}, 5), \\
\qquad (\texttt{"Belenenses"}, 3), \\
\qquad (\texttt{"Benfica"}, 1), \\
\qquad (\texttt{"Braga"}, 2), \\
\qquad (\texttt{"Chaves"}, 5), \\
\qquad (\texttt{"Feirense"}, 5), \\
\qquad (\texttt{"Guimaraes"}, 2), \\
\qquad (\texttt{"Maritimo"}, 3), \\
\qquad (\texttt{"Moreirense"}, 4), \\
\qquad (\texttt{"Nacional"}, 3), \\
\qquad (\texttt{"P.Ferreira"}, 3), \\
\qquad (\texttt{"Porto"}, 1), \\
\qquad (\texttt{"Rio Ave"}, 4), \\
\qquad (\texttt{"Setubal"}, 4), \\
\qquad (\texttt{"Sporting"}, 1), \\
\qquad (\texttt{"Estoril"}, 5)]
\end{array}
$$

a função (monádica) que parte uma lista numa cabeça e cauda *aleatórias*,

$$
\begin{array}{l}
\mathit{getR} :: [a] \rightarrow \mathsf{IO}\ (a, [a]) \\
\mathit{getR}\ x = \textbf{do}\ \{ \\
\quad i \leftarrow \mathit{getStdRandom}\ (\mathit{randomR}\ (0, \mathsf{length}\ \ x - 1)); \\
\quad \mathit{return}\ (x\ !!\ i, \mathit{retira}\ i\ x) \\
\quad \}\ \textbf{where}\ \mathit{retira}\ i\ x = \mathit{take}\ i\ x +\!\!+\ \mathit{drop}\ (i + 1)\ x
\end{array}
$$

e algumas funções auxiliares de menor importância: uma que ordena listas com base num atributo (função que induz uma pré-ordem),

$$
\begin{array}{l}
\mathit{presort} :: (\mathit{Ord}\ a, \mathit{Ord}\ b) \Rightarrow (b \rightarrow a) \rightarrow [b] \rightarrow [b] \\
\mathit{presort}\ f = \mathsf{map}\ \ \pi_2 \cdot \mathit{sort} \cdot (\mathsf{map}\ \ (\mathit{fork}\ f\ \ \mathit{id}))
\end{array}
$$

e outra que converte "look-up tables"em funções (parciais):

$$
\begin{array}{l}
\mathit{pap} :: \mathit{Eq}\ a \Rightarrow [(a, t)] \rightarrow a \rightarrow t \\
\mathit{pap}\ m\ k = \mathit{unJust}\ (\mathit{lookup}\ k\ m)\ \textbf{where}\ \mathit{unJust}\ (\mathit{Just}\ a) = a
\end{array}
$$

C Soluções propostas

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 das funções dadas, mas pode ser adicionado texto e / ou outras funções auxiliares que sejam necessárias.

Problema 1

Neste problema a maior dificuldade foi descobrir como em pointfree conseguíamos aplicar a fórmula matemática de maclaurin dado um natural arbitrário. A primeira solução a que chegamos foi uma que se baseia numa equação matemática demasiado complicada para apresentar uma justificação, e por indicação do docente tentamos encontrar outra via.

$inv1\ x = \mathsf{for}\ ((1+) \cdot ((1-x)*))\ 1$

O ponto de partida foi então uma definição pointwise de inv:

$$
\begin{array}{l}
invpw\ x\ 0 = 1 \\
invpw\ x\ (n+1) = maclaurin\ n + invpw\ x\ n \\
\quad \mathbf{where}\ maclaurin\ 0 = 1 \\
\qquad maclaurin\ (n+1) = (1-x) * maclaurin\ n
\end{array}
$$

A partir de aí, com o auxílio da Lei de *Fokkinga*, tentamos chegar ao catamorfismo de *inv x*:

$$f \cdot \mathbf{in} = h \cdot F\ \langle f, g \rangle\ e\ g \cdot \mathbf{in} = k \cdot F\ \langle f, g \rangle$$

= { in=(either (const 0) succ); f = inv; g = maclaurin; F(split

$$inv \cdot [\underline{0}, \mathsf{succ}\ ] = h \cdot (id + \langle inv, maclaurin \rangle)$$

$$maclaurin \cdot [\underline{0}, \mathsf{succ}\ ] = k \cdot (id + \langle inv, maclaurin \rangle)$$

Deduzimos então as funções h e k de inv e de maclaurin para preencher a lei de *Fokkinga*:

$inv2v\ x\ \underline{0} = 1$

$inv2v\ x\ \mathsf{succ}\ = add \cdot \langle maclaurin\ x, inv2v\ x \rangle$

= { Universal-+ }

$inv2v\ x\ [\underline{0}, \mathsf{succ}\ ] = [\underline{1}, add \cdot \langle maclaurin\ x, inv2v\ x \rangle]$

= { Natural-id; Definição de maclaurin x }

$inv2v\ x\ [\underline{0}, \mathsf{succ}\ ] = [\underline{1} \cdot id, add \cdot \langle ((1-x)*)\ maclaurin\ x\ n, inv2v\ x \rangle]$

= { Absorção-x }

$inv2v\ x\ [\underline{0}, \mathsf{succ}\ ] = [\underline{1} \cdot id, add \cdot (((1-x)*) \times id) \cdot \langle maclaurin\ x, inv2v\ x \rangle]$

= { Absorção-+ }

$inv2v\ x\ [\underline{0}, \mathsf{succ}\ ] = [\underline{1}, add \cdot (((1-x)*) \times id)] \cdot (id + \langle maclaurin\ x, inv2v\ x \rangle)$

Logo,

$$h = [\underline{1}, add \cdot (((1-x)*) \times id)]$$

Obtendo então h, passamos à dedução de k:

$maclaurin\ x\ \underline{0} = 1$

$maclaurin\ x\ \mathsf{succ}\ = (1-x) * (maclaurin\ x)$

= { Universal-+ }

$maclaurin\ x\ [\underline{0}, \mathsf{succ}\ ] = [\underline{1}, (1-x) * (maclaurin\ x)]$

= { Natural-id; Cancelamento-x }

$inv2v\ x\ [\underline{0}, \mathsf{succ}\ ] = [\underline{1} \cdot id, ((1-x)*) \cdot \pi_1 \cdot \langle maclaurin\ x, inv2v\ x \rangle]$

= { Absorção-+ }

$inv2v\ x\ [\underline{0}, \mathsf{succ}\,] = [\underline{1}, ((1-x)*) \cdot \pi_1] \cdot (id + \langle maclaurin\ x, inv2v\ x \rangle)$

Assim,

$k = [\underline{1}, ((1-x)*) \cdot \pi_1]$

Então, pela Lei de *Fokkinga*, temos que:

$$\begin{aligned} inv \cdot [\underline{0}, \mathsf{succ}\,] &= h \cdot (id + \langle inv, maclaurin \rangle) \\ maclaurin \cdot [\underline{0}, \mathsf{succ}\,] &= k \cdot (id + \langle inv, maclaurin \rangle) \end{aligned}$$

$$\begin{aligned} &= \quad \{ \text{ Fokkinga } \} \\ &\quad \langle inv, maclaurin \rangle = cata\ \langle h, k \rangle \end{aligned}$$

Com k encontrado e h definido acima, o catamorfismo de *inv x* apresenta-se:

$invc\ x = \pi_2 \cdot cataNat\ \langle [\underline{(1)}, ((1-x)*) \cdot \pi_1], [\underline{(1)}, add \cdot (((1-x)*) \times id)] \rangle$

Finalmente, a última tarefa consistia em provar que *inv x* seria um ciclo-*for*:

$$\begin{aligned} &\quad \langle h, k \rangle \\ &= \quad \{ \text{ Definição de h e k } \} \\ &\quad \langle [\underline{1}, ((1-x)*) \times id], [\underline{1}, ((1-x)*) \cdot \pi_1] \rangle \\ &= \quad \{ \text{ Lei da Troca } \} \\ &\quad [\langle \underline{1}, \underline{1} \rangle, \langle ((1-x)*) \times id, ((1-x)*) \cdot \pi_1 \rangle] \end{aligned}$$

Finalmente, obtemos a solução, em que *add* foi substituido por *uncurry (+)* por erros de tipos:

$inv\ x = \pi_2 \cdot (\mathsf{for}\ \langle ((1-x)*) \cdot \pi_1, \widehat{(+)} \cdot (((1-x)*) \times id) \rangle\ (1,1))$

Nota: Teste *QuickCheck*:

$prop\_invfokk :: Double \to Property$
$prop\_invfokk\ num = (num > 1 \wedge num < 2) ==> abs\ (a - b) \leqslant 0.000000000000009$
$\quad \textbf{where}\ a = inv\ num\ 500000$
$\qquad b = 1\ /\ num$

Problema 2

$wc\_w\_final :: [\,Char\,] \to Int$
$wc\_w\_final = wrapper \cdot worker$

$wrapper = \pi_2$

$worker = cataList\ [\langle \underline{True}, \underline{0} \rangle, \langle sep \cdot \pi_1, cond\ (\widehat{(\wedge)} \cdot ((\neg \cdot sep) \times \pi_1))\ (\mathsf{succ}\ \cdot \pi_2 \cdot \pi_2)\ (\pi_2 \cdot \pi_2) \rangle]$
$\quad \textbf{where}\ sep\ c = (c \equiv \texttt{' '} \vee c \equiv \texttt{'\textbackslash n'} \vee c \equiv \texttt{'\textbackslash t'})$

Os cálculos do problema 2 são os seguintes:

$$lookahead\_sep\ [\,] = True\ lookahead\_sep\ (c : l) = sep\ c$$

$=$ { Aplicar o [nil,cons] e Lei 73 }

$$lookahead\_sep\ nil = True lookahead\_sep\ cons\ (c, l) = sep\ (c, l)$$

$=$ { Definição de in }

$$look \cdot \mathbf{in} = [\underline{True}, sep \cdot \pi_1]$$

id entre sep e p1 para podermos usar a lei 12

$=$ { Lei 1- Natural-id }

$$look \cdot \mathbf{in} = [\underline{True} \cdot id, sep \cdot id \cdot \pi_1]$$

Como podemos colocar uma função g, utilizamos um split para termos já o Funtor definido para usarmos Fokkinga

$$look \cdot \mathbf{in} = [\underline{True} \cdot id, sep \cdot id \cdot \pi_1]$$

$=$ { Lei 12- Natural-p1 }

$$look \cdot \mathbf{in} = [\underline{True} \cdot id, sep \cdot \pi_1 \cdot (id \times \langle lookahead\_sep, wc\_w \rangle)]$$

$=$ { Lei 22- Absorcao - + }

$$look \cdot \mathbf{in} = [\underline{True}, sep \cdot \pi_1] \cdot (id + (id \times \langle lookahead\_sep, wc\_w \rangle))$$

$=$ { Lei 50 - Fokkinga }

$$wc\_w \cdot \mathbf{in} = [\underline{0}, x] \cdot (id + (id \times \langle lookahead\_sep, wc\_w \rangle))$$

Temos de determinar o x
Aqui segue um esquema para a melhor compreensão da wc$_w$:

$wc\_w\ nil = 0$
$wc\_w\ cons\ (c, l) = \mathbf{let}\ x = (c, (lookahead\_sep\ l, wc\_w\ l))$
$\mathbf{in\ if}\ (\neg \cdot sep\ c \wedge lookahead\_sep\ l)\ \mathbf{then}\ \pi_2 \cdot \pi_2 \cdot x + 1$
$\mathbf{else}\ \pi_2 \cdot \pi_2 \cdot x$

Tornamos o c assim para mais tarde conseguirmos torná-lo em (id x (split lookahead$_sepwc_w$))

$=$ { Lei 32 - 2 Lei da fusao do condicional }

$wc\_w\ nil = 0$
$wc\_w\ cons\ (c, l) =$
$(cond\ (\neg \cdot sep \cdot \pi_1 \wedge \pi_1 \cdot \pi_2)\ (\mathsf{succ}\ \cdot \pi_2 \cdot \pi_2, \pi_2 \cdot \pi_2)) \cdot (c, (lookahead\_sep\ l, wc\_w\ l))$

$= \quad \{$ Lei 73 - Igualdade extensional $\}$

$wc\_w\ nil = 0$
$wc\_w\ cons = (cond\ ((\wedge) \cdot \langle \neg \cdot sep \cdot \pi_1, \pi_1 \cdot \pi_2 \rangle)\ (\mathsf{succ}\ \cdot \pi_2 \cdot \pi_2, \pi_2 \cdot \pi_2)) \cdot (id \times \langle lookahead\_sep, wc\_w \rangle)$

$= \quad \{$ Lei 10 - Igualdade extensional; Lei 1 - Natural-id $\}$

$wc\_w\ nil = 0 \cdot id$
$wc\_w\ cons =$
$(cond\ ((\wedge) \cdot (\neg \cdot sep \times \pi_1))\ (\mathsf{succ}\ \cdot \pi_2 \cdot \pi_2, \pi_2 \cdot \pi_2)) \cdot (id \times \langle lookahead\_sep, wc\_w \rangle)$

$= \quad \{$ Definiçao de in $\}$

$wc\_w \cdot \mathbf{in} = [\underline{0} \cdot id, (cond\ ((\wedge) \cdot (\neg \cdot sep \times \pi_1))\ (\mathsf{succ}\ \cdot \pi_2 \cdot \pi_2, \pi_2 \cdot \pi_2)) \cdot (id \times \langle lookahead\_sep, wc\_w \rangle)]$

$= \quad \{$ Lei 2 - Absorçao-+ $\}$

$wc\_w \cdot \mathbf{in} = [\underline{0}, cond\ ((\wedge) \cdot (\neg \cdot sep \times \pi_1))\ (\mathsf{succ}\ \cdot \pi_2 \cdot \pi_2, \pi_2 \cdot \pi_2)] \cdot (id + (id \times \langle lookahead\_sep, wc\_w \rangle))$

$= \quad \{$ Ja temos ambas prontas para a regra de Fokkinga $\}$

$look \cdot \mathbf{in} = [\underline{True}, sep \cdot \pi_1] \cdot (id + (id \times < lookahead\_sep, wc\_w >))$
$wc\_w \cdot \mathbf{in} =$
$[\underline{0}, cond\ ((\wedge) \cdot (\neg \cdot sep \times \pi_1))\ (\mathsf{succ}\ \cdot \pi_2 \cdot \pi_2, \pi_2 \cdot \pi_2)] \cdot (id + (id \times \langle lookahead\_sep, wc\_w \rangle))$

$= \quad \{$ Lei 50 - Fokkinga $\}$

$cataList\ \langle [\underline{True}, sep] \cdot \pi_1, [\underline{0}, cond\ ((\wedge) \cdot (\neg \cdot sep \times \pi_1))\ (\mathsf{succ}\ \cdot \pi_2 \cdot \pi_2, \pi_2 \cdot \pi_2)] \rangle$

$= \quad \{$ Lei 28 - Lei da troca $\}$

$cataList\ [\langle \underline{True}, \underline{0} \rangle, \langle sep \cdot \pi_1, cond\ ((\wedge) \cdot (\neg \cdot sep \times \pi_1))\ (\mathsf{succ}\ \cdot \pi_2 \cdot \pi_2, \pi_2 \cdot \pi_2) \rangle]$

$prop\_wc\_w\_final :: String \rightarrow Property$
$prop\_wc\_w\_final\ str = forAll\ genSafeString\ \$\ \lambda str \rightarrow (toInteger\ (wc\_w\ str) \equiv toInteger\ (wc\_w\_final\ str))$
$genSafeChar :: Gen\ Char$
$genSafeChar = elements\ ([\texttt{'a'} \mathrel{..} \texttt{'z'}] +\!\!+ [\texttt{' '}, \texttt{'\textbackslash t'}, \texttt{'\textbackslash n'}])$
$genSafeString :: Gen\ String$
$genSafeString = listOf\ genSafeChar$

Problema 3

$inB\_tree :: () + (\mathsf{B\text{-}tree}\ a, [(a, \mathsf{B\text{-}tree}\ a)]) \rightarrow \mathsf{B\text{-}tree}\ a$
$inB\_tree = [\underline{Nil}, \widehat{Block}]$
$outB\_tree :: \mathsf{B\text{-}tree}\ a \rightarrow () + (\mathsf{B\text{-}tree}\ a, [(a, \mathsf{B\text{-}tree}\ a)])$
$outB\_tree\ (Nil) = i_1\ ()$
$outB\_tree\ (Block\ \{ leftmost = l, block = b \}) = i_2\ (l, b)$
$recB\_tree\ f = baseB\_tree\ id\ f$
$baseB\_tree\ g\ f = id + (f \times (\mathsf{map}\ \ (g \times f)))$
$cataB\_tree\ g = g \cdot (recB\_tree\ (cataB\_tree\ g)) \cdot outB\_tree$
$anaB\_tree\ g = inB\_tree \cdot (recB\_tree\ (anaB\_tree\ g)) \cdot g$
$hyloB\_tree\ f\ g = cataB\_tree\ f \cdot anaB\_tree\ g$
$\mathbf{instance}\ Functor\ \mathsf{B\text{-}tree}$
$\quad \mathbf{where}\ \mathsf{fmap}\ f = cataB\_tree\ (inB\_tree \cdot baseB\_tree\ f\ id)$

Para as funções mais complexas apresentamos o diagrama correspondente que nos ajudou a resolver os problemas propostos.

inordB_tree:

$$
\begin{array}{l}
inordB\_tree = cataB\_tree\ inord \\
\quad \mathbf{where}\ inord = [nil, \mathsf{conc} \cdot (id \times aux)] \\
\qquad aux = cataList\ [nil, \mathsf{conc} \cdot ((singl \cdot \pi_1) \times id)]
\end{array}
$$

$$
\begin{tikzcd}
B\_Tree\ A \arrow[d, "cataB\_tree\ inord"'] & & 1 + (B\_Tree\ A \times (A \times B\_treeA)*) \arrow[ll, "inB\_tree"'] \arrow[d, "id+((cataB\_tree\ inord)\times(\mathsf{map}\ \ (id\times cataB\_tree\ inord)))"] \\
A* & & 1 + A * \times (A \times A*)* \arrow[ll, "inord"]
\end{tikzcd}
$$

largestBlock:

$$
\begin{array}{l}
largestBlock :: \mathsf{B\text{-}tree}\ a \rightarrow Int \\
largestBlock = cataB\_tree\ lb \\
\quad \mathbf{where}\ lb = [\underline{0}, \widehat{max} \cdot \langle \mathsf{length}\ \cdot \pi_2, \widehat{max} \cdot (id \times (maximum \cdot aux))\rangle] \\
\qquad aux = cataList\ [nil, cons \cdot (\pi_2 \times id)]
\end{array}
$$

$$
\begin{tikzcd}
B\_Tree\ A \arrow[d, "cataB\_tree\ lb"'] & & 1 + (B\_Tree\ A \times (A \times B\_treeA)*) \arrow[ll, "inB\_tree"'] \arrow[d, "id+((cataB\_tree\ lb)\times(\mathsf{map}\ \ (id\times cataB\_tree\ lb)))"] \\
A* & & 1 + A * \times (A \times A*)* \arrow[ll, "lb"]
\end{tikzcd}
$$

mirrorB_tree

$$
\begin{array}{l}
mirrorB\_tree = anaB\_tree\ ((id + (join \cdot inside \cdot auxb2)) \cdot outB\_tree) \\
\quad \mathbf{where}\ auxb2 = id \times unzip \\
\qquad inside = \langle reverse \cdot \pi_1 \cdot \pi_2, reverse \cdot cons \cdot \langle \pi_1, \pi_2 \cdot \pi_2 \rangle\rangle \\
\qquad join = \langle head \cdot \pi_2, \widehat{zip} \cdot \langle \pi_1, tail \cdot \pi_2 \rangle\rangle
\end{array}
$$

$$
\begin{array}{l}
lsplitB\_tree\ [\,] = i_1\ () \\
lsplitB\_tree\ (h : t) = i_2\ (s, [(h, l)])\ \mathbf{where}\ (s, l) = partB\_tree\ (<h)\ t \\
partB\_tree :: (a \rightarrow Bool) \rightarrow [a] \rightarrow ([a], [a]) \\
partB\_tree\ p\ [\,] \qquad\qquad = ([\,], [\,]) \\
partB\_tree\ p\ (h : t) \mid p\ h = \mathbf{let}\ (s, l) = partB\_tree\ p\ t\ \mathbf{in}\ (h : s, l) \\
\quad \mid otherwise = \mathbf{let}\ (s, l) = partB\_tree\ p\ t\ \mathbf{in}\ (s, h : l) \\
qSortB\_tree :: Ord\ a \Rightarrow [a] \rightarrow [a] \\
qSortB\_tree = hyloB\_tree\ [nil, inord]\ lsplitB\_tree \\
\quad \mathbf{where}\ inord = \mathsf{conc} \cdot (id \times (concat \cdot (\mathsf{map}\ \ cons))) \\
dotB\_tree :: (Show\ a) \Rightarrow \mathsf{B\text{-}tree}\ a \rightarrow \mathsf{IO}\ ExitCode \\
dotB\_tree = dotpict \cdot bmap\ nothing\ (Just \cdot init \cdot concat \cdot (\mathsf{map}\ \ (+\!\!+ \texttt{"|"})) \cdot (\mathsf{map}\ \ show)) \cdot cB\_tree2Exp \\
cB\_tree2Exp = cataB\_tree\ [\underline{(Var\ \texttt{"nil"})}, aux] \\
\quad \mathbf{where}\ aux = \widehat{(Term)} \cdot (id \times cons) \cdot \langle \pi_1 \cdot \pi_2, \langle \pi_1, \pi_2 \cdot \pi_2 \rangle\rangle \cdot (id \times unzip)
\end{array}
$$

Problema 4

$$
\begin{array}{l}
(\!|ga\ gb|\!)_A = inA \cdot (id + (\!|ga\ gb|\!)_A \times (\!|ga\ gb|\!)_B) \cdot ga \\
(\!|ga\ gb|\!)_B = inB \cdot (id + (\!|ga\ gb|\!)_A) \cdot gb
\end{array}
$$

$$
generateAlgae = [\!(((id + \langle id, id \rangle) \cdot outNat\ outNat)\!]_A
$$

generateAlgae:

$$
\begin{aligned}
&showAlgae = (\!|ga\ gb|\!)_A\\
&\quad \textbf{where}\ ga = [\underline{\texttt{"A"}}, \mathsf{conc}]\\
&\qquad gb = [\underline{\texttt{"B"}}, id]
\end{aligned}
$$

$$
\begin{tikzcd}
Algae \arrow[d, "cataB\_tree\ lb"'] & & 1 + A \times B \arrow[ll, "inA"'] \arrow[d, "\underline{\texttt{"A"}} + ga \times gb"] \\
String & & 1 + String \times String \arrow[ll, "gagb"]
\end{tikzcd}
$$

$$
\begin{aligned}
&prop\_4\_6 :: Integer \rightarrow Property\\
&prop\_4\_6\ n = (n \geqslant 0 \wedge n < 24) ==> lshG\ n \equiv fibsucc\ n\\
&\quad \textbf{where}\ lshG = toInteger \cdot \mathsf{length}\ \cdot showAlgae \cdot generateAlgae \cdot fromInteger\\
&\qquad fibsucc = fib \cdot \mathsf{succ}
\end{aligned}
$$

Problema 5

$$
\begin{aligned}
&permuta\ [\,] = return\ [\,]\\
&permuta\ l = \textbf{do}\ \{(h, t) \leftarrow getR\ l; x \leftarrow permuta\ t; return\ (h : x)\}\\
&eliminatoria\ (Leaf\ x) = return\ x\\
&eliminatoria\ (Fork\ (x, y)) = \textbf{do}\ \{v \leftarrow eliminatoria\ x; z \leftarrow eliminatoria\ y; jogo\ (v, z)\}
\end{aligned}
$$

Quanto à primeira parte deste problema, fizemo-la, tal como pedido, com recurso ao monáde IO. Tendo em conta que a função *getR*, recebe uma lista com elementos de um determinado tipo, nos devolve um par com um elemento random dessa lista e com o resto da mesma sem esse elemento, então a nossa ideia foi chamar essa função até que chegasse ao fim da lista inicial, isto é, recursividade. Ficamos com o primeiro elemento de getR, e fazemos *permuta t*, em que t é o segundo elemento do par devolvido por getR, e assim sucessivamente.

Em relação à segunda parte do problema, elaboramo-la da mesma forma que elaborámos a primeira, com recursividade. Percorrendo todos os nodos de uma árvore, quando chegarmos às folhas de um Fork, chamamos a função jogo, fornecida no enunciado, tendo como argumentos as equipas que estão nessas folhas, devolvendo a probabilidade de cada equipa nesse jogo vencer. Realizando isto para toda a árvore, obtemos o resultado final esperado.

Índice