

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

*Departamento de Informática*  
Universidade do Minho

Junho de 2017

## **Grupo nr. 28**

---

a78582	Hugo Brandão
a78296	Sérgio Alves
a78218	Tiago Alves

## **Conteúdo**

<b>1</b>	<b>Preâmbulo</b>	<b>2</b>
<b>2</b>	<b>Documentação</b>	<b>2</b>
<b>3</b>	<b>Como realizar o trabalho</b>	<b>3</b>
<b>A</b>	<b>Mónade para probabilidades e estatística</b>	<b>10</b>
<b>B</b>	<b>Definições auxiliares</b>	<b>11</b>
<b>C</b>	<b>Soluções propostas</b>	<b>11</b>

# 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`<sup>1</sup> 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-processor que faz “pretty printing” de código Haskell em **L<sup>A</sup>T<sub>E</sub>X** 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**:

```
{-# OPTIONS_GHC -XNPlusKPatterns#-}
import Cp
```

---

<sup>1</sup>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**<sup>2</sup> 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

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

---

<sup>2</sup>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:

```
wc_w :: [Char] -> Int
wc_w [] = 0
wc_w (c:l) =
  if ¬ (sep c) ∧ lookahead_sep l
  then wc_w l + 1
  else wc_w l
  where
    sep c = (c ≡ ' ' ∨ c ≡ '\n' ∨ c ≡ '\t')
    lookahead_sep [] = True
    lookahead_sep (c:l) = sep c

```

Re-implemente esta função segundo o modelo *worker/wrapper* onde *wrapper* deverá ser um catamorfismo 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:

```
data B-tree a = Nil | Block { leftmost :: B-tree a, block :: [(a, B-tree a)] } deriving (Show, Eq)

```

Por exemplo, a B-tree<sup>3</sup>

---

<sup>3</sup>Créditos: figura extraída de <https://en.wikipedia.org/wiki/B-tree>.



é representada no tipo acima por:

```

t = Block {
  leftmost = Block {
    leftmost = Nil,
    block = [(1, Nil), (2, Nil), (5, Nil), (6, Nil)]},
  block = [
    (7, Block {
      leftmost = Nil,
      block = [(9, Nil), (12, Nil)]}),
    (16, Block {
      leftmost = Nil,
      block = [(18, Nil), (21, Nil)]})
  ]}
  
```

Pretende-se, neste problema:

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

```

lsplitB_tree [] = i1 ()
lsplitB_tree [7] = i2 ([], [(7, [])])
lsplitB_tree [5, 7, 1, 9] = i2 ([1], [(5, []), (7, [9])])
lsplitB_tree [7, 5, 1, 9] = i2 ([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 *dotBTree t* para

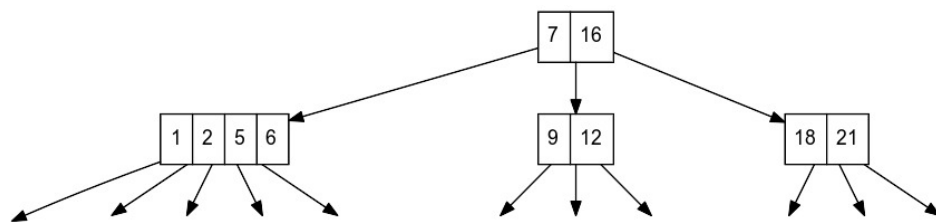
```

t = Node 6, (Node 3, (Node 2, (Empty, Empty)), Empty), Node 7, (Empty, Node 9, (Empty, Empty))))
  
```

obter-se-á a imagem



Escreva de forma semelhante uma função `dotB_tree` que permita mostrar em [Graphviz](#)<sup>4</sup> árvores B-tree tal como se ilustra a seguir,



para a árvore 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<sup>5</sup> no sistema:

**Variáveis:**  $A$  e  $B$

**Constantes:** nenhuma

**Axioma:**  $A$

**Regras:**  $A \rightarrow A B, B \rightarrow A$ .

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

<sup>4</sup>Como alternativa a instalar [Graphviz](#), podem usar [WebGraphviz](#) num browser.

<sup>5</sup>Ver [https://en.wikipedia.org/wiki/Aristid\\_Lindenmayer](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:

```
type Algae = A
data A = NA | A A B deriving Show
data B = NB | B A deriving Show
```

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

```
inA :: 1 + A × B → A
inA = [NA, A]
outA :: A → 1 + A × B
outA NA = i1 ()
outA (A a b) = i2 (a, b)
inB :: 1 + A → B
inB = [NB, B]
outB :: B → 1 + A
outB NB = i1 ()
outB (B a) = i2 a
```

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

$$\begin{aligned} \llbracket \cdot \cdot \rrbracket_A &:: (1 + c \times d \rightarrow c) \rightarrow (1 + c \rightarrow d) \rightarrow A \rightarrow c \\ \llbracket ga \ gb \rrbracket_A &= ga \cdot (id + \llbracket ga \ gb \rrbracket_A \times \llbracket ga \ gb \rrbracket_B) \cdot outA \end{aligned}$$

e a mesma coisa para os  $B$ s:

$$\begin{aligned} \llbracket \cdot \cdot \rrbracket_B &:: (1 + c \times d \rightarrow c) \rightarrow (1 + c \rightarrow d) \rightarrow B \rightarrow d \\ \llbracket ga \ gb \rrbracket_B &= gb \cdot (id + \llbracket ga \ gb \rrbracket_A) \cdot outB \end{aligned}$$

Pretende-se, neste problema:

1. A definição dos anamorfismos dos tipos  $A$  e  $B$ .
2. A definição da função

$$generateAlgae :: Int \rightarrow Algae$$

como anamorfismo de  $Algae$  e da função

$$showAlgae :: Algae \rightarrow String$$

como catamorfismo de  $Algae$ .

3. Use **QuickCheck** para verificar a seguinte propriedade:

$$length \cdot showAlgae \cdot generateAlgae = fib \cdot succ$$

## Problema 5

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

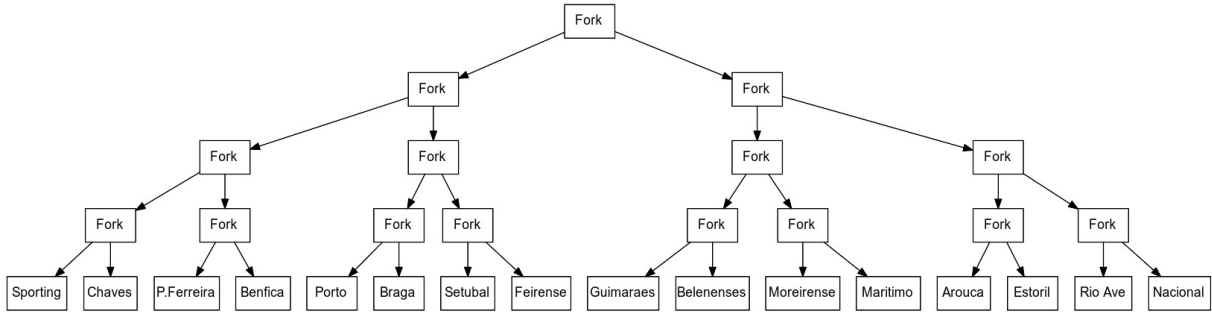
```
equipas :: [Equipa]
equipas = [
  "Arouca", "Belenenses", "Benfica", "Braga", "Chaves", "Feirense",
  "Guimaraes", "Maritimo", "Moreirense", "Nacional", "P.Ferreira",
  "Porto", "Rio Ave", "Setubal", "Sporting", "Estoril"
]
```

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.<sup>6</sup> Por exemplo,  $f(\text{"Arouca"}, \text{"Braga"})$  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  $\text{Dist } a$  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

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

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<sup>7</sup>, 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  $[Equipa] \rightarrow \text{Dist } Equipa$ ,

$quem\_vence :: [Equipa] \rightarrow \text{Dist } Equipa$   
 $quem\_vence = eliminatória \cdot sorteio$

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

O anamorfismo *sorteio*  $:: [Equipa] \rightarrow \text{LTree } Equipa$  tem a seguinte arquitectura,<sup>8</sup>

$sorteio = anaLTree \text{ lspl}it \cdot envia \cdot permuta$

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

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

A presença do mónade de IO tem a ver com a geração de números aleatórios<sup>9</sup>.

<sup>6</sup>Tratando-se de jogos eliminatórios, não há lugar a empates.

<sup>7</sup>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.

<sup>8</sup>A função *envia* não é importante para o processo; apenas se destina a simplificar a arquitectura monádica da solução.

<sup>9</sup>Quem estiver interessado em detalhes deverá consultar [System.Random](#).



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

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

*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

$$eliminatória :: LTree Equipa \rightarrow Dist Equipa$$

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. Ritchie. *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

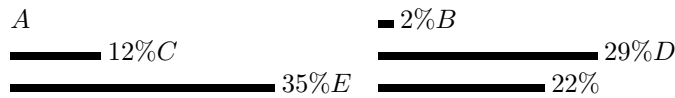
## A Mónade para probabilidades e estatística

Mónades são 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

$$\text{newtype Dist } a = D \{ \text{unD} :: [(a, \text{ProbRep})] \} \quad (1)$$

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  $E$ ,



será representada pela distribuição

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

que o **GHCI** mostrará assim:

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

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

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

isto é

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

distribuição *normais*, eg.

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

etc.<sup>10</sup>

$\text{Dist}$  forma um **mónade** cuja unidade é  $\text{return } a = D [(a, 1)]$  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  $g:A \rightarrow \text{Dist } B$  e  $f:B \rightarrow \text{Dist } C$  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.

<sup>10</sup>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,

```
type Equipa = String
jogo :: (Equipa, Equipa) → Dist Equipa
jogo (e1, e2) = D [(e1, 1 - r1 / (r1 + r2)), (e2, 1 - r2 / (r1 + r2))] where
  r1 = rank e1
  r2 = rank e2
  rank = pap ranks
  ranks = [
    ("Arouca", 5),
    ("Belenenses", 3),
    ("Benfica", 1),
    ("Braga", 2),
    ("Chaves", 5),
    ("Feirense", 5),
    ("Guimaraes", 2),
    ("Maritimo", 3),
    ("Moreirense", 4),
    ("Nacional", 3),
    ("P.Ferreira", 3),
    ("Porto", 1),
    ("Rio Ave", 4),
    ("Setubal", 4),
    ("Sporting", 1),
    ("Estoril", 5)]
```

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

```
getR :: [a] → IO (a, [a])
getR x = do {
  i ← getStdRandom (randomR (0, length x - 1));
  return (x !! i, retira i x)
} where retira i x = take i x ++ drop (i + 1) x
```

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

```
presort :: (Ord a, Ord b) ⇒ (b → a) → [b] → [a]
presort f = map π2 · sort · (map (fork f id))
```

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

```
pap :: Eq a ⇒ [(a, t)] → a → t
pap m k = unJust (lookup k m) where unJust (Just a) = a
```

## 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.

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

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

$$\begin{aligned} invpw\ x\ 0 &= 1 \\ invpw\ x\ (n+1) &= maclaurin\ n + invpw\ x\ n \\ \textbf{where } maclaurin\ 0 &= 1 \\ maclaurin\ (n+1) &= (1-x) * maclaurin\ n \end{aligned}$$

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

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

$$\begin{aligned} &= \{ \text{in} = (\text{either } (\text{const } 0)\ \text{succ}); f = \text{inv}; g = \text{maclaurin}; F(\text{split}) \} \\ inv \cdot [\underline{0}, \text{succ}] &= h \cdot (id + \langle inv, maclaurin \rangle) \end{aligned}$$

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

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

$$\begin{aligned} inv2v\ x\ \underline{0} &= 1 \\ inv2v\ x\ \text{succ} &= add \cdot \langle maclaurin\ x, inv2v\ x \rangle \\ &= \{ \text{Universal-+} \} \\ inv2v\ x\ [\underline{0}, \text{succ}] &= [\underline{1}, add \cdot \langle maclaurin\ x, inv2v\ x \rangle] \\ &= \{ \text{Natural-id; Definição de maclaurin } x \} \\ inv2v\ x\ [\underline{0}, \text{succ}] &= [\underline{1} \cdot id, add \cdot \langle ((1-x)*)\ maclaurin\ x\ n, inv2v\ x \rangle] \\ &= \{ \text{Absorção-x} \} \\ inv2v\ x\ [\underline{0}, \text{succ}] &= [\underline{1} \cdot id, add \cdot (((1-x)*)) \times id \cdot \langle maclaurin\ x, inv2v\ x \rangle] \\ &= \{ \text{Absorção-+} \} \\ inv2v\ x\ [\underline{0}, \text{succ}] &= [\underline{1}, add \cdot (((1-x)*)) \times id] \cdot (id + \langle maclaurin\ x, inv2v\ x \rangle) \end{aligned}$$

Logo,

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

Obtendo então *h*, passamos à dedução de *k*:

$$\begin{aligned} maclaurin\ x\ \underline{0} &= 1 \\ maclaurin\ x\ \text{succ} &= (1-x) * (maclaurin\ x) \\ &= \{ \text{Universal-+} \} \\ maclaurin\ x\ [\underline{0}, \text{succ}] &= [\underline{1}, (1-x) * (maclaurin\ x)] \\ &= \{ \text{Natural-id; Cancelamento-x} \} \\ inv2v\ x\ [\underline{0}, \text{succ}] &= [\underline{1} \cdot id, ((1-x)*)) \cdot \pi_1 \cdot \langle maclaurin\ x, inv2v\ x \rangle] \\ &= \{ \text{Absorção-+} \} \end{aligned}$$

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

Assim,

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

Então, pela Lei de Fokkinga, temos que:

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

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

Com k encontrado e h definido acima, o catamorfismo de  $\text{inv } x$  apresenta-se:

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

Finalmente, a última tarefa consistia em provar que  $\text{inv } x$  seria um ciclo-for:

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

Finalmente, obtemos a solução, em que  $\text{add}$  foi substituído por  $\text{uncurry } (+)$  por erros de tipos:

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

Nota: Teste *QuickCheck*:

```
prop_invfokk :: Double -> Property
prop_invfokk num = (num > 1 & num < 2) ==> abs (a - b) <= 0.0000000000000009
  where a = inv num 500000
        b = 1 / num
```

## Problema 2

```
wc_w_final :: [Char] -> Int
wc_w_final = wrapper . worker
wrapper = \pi2
worker = cataList [(\True, 0), (\sep . \pi1, cond (\widehat{(\wedge)} . ((\neg . sep) \times \pi1)) (\succ . \pi2 . \pi2) (\pi2 . \pi2))]
  where sep c = (c \equiv ' ' \vee c \equiv '\n' \vee c \equiv '\t')
```

Os cálculos do problema 2 são os seguintes:

$$\begin{aligned}
& lookahead\_sep [] = True \quad lookahead\_sep (c : l) = sep \ c \\
= & \quad \{ \text{Aplicar o [nil,cons] e Lei 73} \} \\
& lookahead\_sep \ nil = True \quad lookahead\_sep \ cons \ (c, l) = sep \ (c, l) \\
= & \quad \{ \text{Definição de in} \} \\
& look \cdot \mathbf{in} = [True, sep \cdot \pi_1]
\end{aligned}$$

id entre sep e p1 para podermos usar a lei 12

$$\begin{aligned}
= & \quad \{ \text{Lei 1- Natural-id} \} \\
& look \cdot \mathbf{in} = [True \cdot id, sep \cdot id \cdot \pi_1]
\end{aligned}$$

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

$$\begin{aligned}
& look \cdot \mathbf{in} = [True \cdot id, sep \cdot id \cdot \pi_1] \\
= & \quad \{ \text{Lei 12- Natural-p1} \} \\
& look \cdot \mathbf{in} = [True \cdot id, sep \cdot \pi_1 \cdot (id \times \langle lookahead\_sep, wc\_w \rangle)] \\
= & \quad \{ \text{Lei 22- Absorcao - +} \} \\
& look \cdot \mathbf{in} = [True, sep \cdot \pi_1] \cdot (id + (id \times \langle lookahead\_sep, wc\_w \rangle)) \\
= & \quad \{ \text{Lei 50 - Fokkinga} \} \\
& wc\_w \cdot \mathbf{in} = [0, x] \cdot (id + (id \times \langle lookahead\_sep, wc\_w \rangle))
\end{aligned}$$

Temos de determinar o  $x$

Aqui segue um esquema para a melhor compreensão da  $wc_w$  :

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

Tornamos o  $c$  assim para mais tarde conseguirmos torná-lo em  $(id \times (\text{split } lookahead\_sep \ wc\_w))$

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

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

$$\begin{aligned}
&= \{ \text{Lei 73 - Igualdade extensional} \} \\
&wc\_w \text{ nil} = 0 \\
&wc\_w \text{ cons} = (cond ((\wedge) \cdot \langle \neg \cdot sep \cdot \pi_1, \pi_1 \cdot \pi_2 \rangle) (succ \cdot \pi_2 \cdot \pi_2, \pi_2 \cdot \pi_2)) \cdot (id \times \langle lookahead\_sep, wc\_w \rangle) \\
&= \{ \text{Lei 10 - Igualdade extensional; Lei 1 - Natural-id} \} \\
&wc\_w \text{ nil} = 0 \cdot id \\
&wc\_w \text{ cons} = \\
&(cond ((\wedge) \cdot (\neg \cdot sep \times \pi_1)) (succ \cdot \pi_2 \cdot \pi_2, \pi_2 \cdot \pi_2)) \cdot (id \times \langle lookahead\_sep, wc\_w \rangle) \\
&= \{ \text{Definição de in} \} \\
&wc\_w \cdot \mathbf{in} = [0, id, (cond ((\wedge) \cdot (\neg \cdot sep \times \pi_1)) (succ \cdot \pi_2 \cdot \pi_2, \pi_2 \cdot \pi_2)) \cdot (id \times \langle lookahead\_sep, wc\_w \rangle)] \\
&= \{ \text{Lei 2 - Absorção-+} \} \\
&wc\_w \cdot \mathbf{in} = [0, cond ((\wedge) \cdot (\neg \cdot sep \times \pi_1)) (succ \cdot \pi_2 \cdot \pi_2, \pi_2 \cdot \pi_2)] \cdot (id + (id \times \langle lookahead\_sep, wc\_w \rangle)) \\
&= \{ \text{Ja temos ambas prontas para a regra de Fokkinga} \} \\
&look \cdot \mathbf{in} = [\underline{True}, sep \cdot \pi_1] \cdot (id + (id \times \langle lookahead\_sep, wc\_w \rangle)) \\
&wc\_w \cdot \mathbf{in} = \\
&[0, cond ((\wedge) \cdot (\neg \cdot sep \times \pi_1)) (succ \cdot \pi_2 \cdot \pi_2, \pi_2 \cdot \pi_2)] \cdot (id + (id \times \langle lookahead\_sep, wc\_w \rangle)) \\
&= \{ \text{Lei 50 - Fokkinga} \} \\
&cataList \langle [\underline{True}, sep] \cdot \pi_1, [0, cond ((\wedge) \cdot (\neg \cdot sep \times \pi_1)) (succ \cdot \pi_2 \cdot \pi_2, \pi_2 \cdot \pi_2)] \rangle \\
&= \{ \text{Lei 28 - Lei da troca} \} \\
&cataList [\langle \underline{True}, 0 \rangle, \langle sep \cdot \pi_1, cond ((\wedge) \cdot (\neg \cdot sep \times \pi_1)) (succ \cdot \pi_2 \cdot \pi_2, \pi_2 \cdot \pi_2) \rangle]
\end{aligned}$$

```

prop_wc_w_final :: String → Property
prop_wc_w_final str = forAll genSafeString $ \astr → (toInteger (wc_w str) ≡ toInteger (wc_w_final str))

genSafeChar :: Gen Char
genSafeChar = elements ([ 'a' .. 'z' ] ++ [ ' ', '\t', '\n' ])

genSafeString :: Gen String
genSafeString = listOf genSafeChar

```

### Problema 3

```

inB_tree :: () + (B-tree a, [(a, B-tree a)]) → B-tree a
inB_tree = [Nil, Block]
outB_tree :: B-tree a → () + (B-tree a, [(a, B-tree a)])
outB_tree (Nil) = i1 ()
outB_tree (Block { leftmost = l, block = b }) = i2 (l, b)
recB_tree f = baseB_tree id f
baseB_tree g f = id + (f × (map (g × f)))
cataB_tree g = g · (recB_tree (cataB_tree g)) · outB_tree

```

$$\begin{array}{ccc}
B\_Tree\ A & \xleftarrow{\text{inB\_tree}} & 1 + (B\_Tree\ A \times (A \times B\_treeA)*) \\
\downarrow \text{cataB\_tree } g & & \downarrow \text{id} + (\text{cataB\_tree } g \times \text{map } (\text{id} \times \text{cataB\_tree } g)) \\
A* & \xleftarrow{f} & 1 + A* \times (A \times A*)*
\end{array}$$

$$\text{anaB\_tree } g = \text{inB\_tree} \cdot (\text{recB\_tree } (\text{anaB\_tree } g)) \cdot g$$

$$\begin{array}{ccc} B\_Tree \ A & \xrightarrow{\text{outB\_tree}} & 1 + (B\_Tree \ A \times (A \times B\_treeA)*) \\ \text{anaB\_tree } g \uparrow & & \uparrow \text{id} + (\text{anaB\_tree } g \times \text{map } (\text{id} \times \text{anaB\_tree } g)) \\ A* & \xrightarrow{f} & 1 + A* \times (A \times A*)* \end{array}$$

$$\text{hyloB\_tree } f \ g = \text{cataB\_tree } f \cdot \text{anaB\_tree } g$$

**instance** *Functor* **B-tree**

**where**  $\text{fmap } f = \text{cataB\_tree } (\text{inB\_tree} \cdot \text{baseB\_tree } f \ \text{id})$

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

**inordB.tree:**

$$\text{inordB\_tree} = \text{cataB\_tree } \text{inord}$$

**where**  $\text{inord} = [\text{nil}, \text{conc} \cdot (\text{id} \times \text{aux})]$

$\text{aux} = \text{cataList } [\text{nil}, \text{conc} \cdot ((\text{singl} \cdot \pi_1) \times \text{id})]$

$$\begin{array}{ccc} B\_Tree \ A & \xleftarrow{\text{inB\_tree}} & 1 + (B\_Tree \ A \times (A \times B\_treeA)*) \\ \text{cataB\_tree } \text{inord} \downarrow & & \downarrow \text{id} + ((\text{cataB\_tree } \text{inord}) \times (\text{map } (\text{id} \times \text{cataB\_tree } \text{inord}))) \\ A* & \xleftarrow{\text{inord}} & 1 + A* \times (A \times A*)* \end{array}$$

**largestBlock:**

$$\text{largestBlock} :: \text{B-tree } a \rightarrow \text{Int}$$

$$\text{largestBlock} = \text{cataB\_tree } \text{lb}$$

**where**  $\text{lb} = [0, \widehat{\text{max}} \cdot (\text{length} \cdot \pi_2, \widehat{\text{max}} \cdot (\text{id} \times (\text{maximum} \cdot \text{aux})))]$

$\text{aux} = \text{cataList } [\text{nil}, \text{cons} \cdot (\pi_2 \times \text{id})]$

$$\begin{array}{ccc} B\_Tree \ A & \xleftarrow{\text{inB\_tree}} & 1 + (B\_Tree \ A \times (A \times B\_treeA)*) \\ \text{cataB\_tree } \text{lb} \downarrow & & \downarrow \text{id} + ((\text{cataB\_tree } \text{lb}) \times (\text{map } (\text{id} \times \text{cataB\_tree } \text{lb}))) \\ A* & \xleftarrow{\text{lb}} & 1 + A* \times (A \times A*)* \end{array}$$

**mirrorB.tree**

$$\text{mirrorB\_tree} = \text{anaB\_tree } ((\text{id} + (\text{join} \cdot \text{inside} \cdot \text{auxb2})) \cdot \text{outB\_tree})$$

**where**  $\text{auxb2} = \text{id} \times \text{unzip}$

$\text{inside} = \langle \text{reverse} \cdot \pi_1 \cdot \pi_2, \text{reverse} \cdot \text{cons} \cdot \langle \pi_1, \pi_2 \cdot \pi_2 \rangle \rangle$

$\text{join} = \langle \text{head} \cdot \pi_2, \widehat{\text{zip}} \cdot \langle \pi_1, \text{tail} \cdot \pi_2 \rangle \rangle$

$$\begin{array}{ccc} B\_Tree \ A & \xrightarrow{\text{outB\_tree}} & 1 + (B\_Tree \ A \times (A \times B\_treeA)*) \\ g = \text{anaB\_tree } (\text{id} + (\text{join} \cdot \text{inside} \cdot \text{auxb2})) \cdot \text{outB\_tree} \uparrow & & \uparrow \text{id} + (\text{anaB\_tree } g \times \text{map } (\text{id} \times \text{anaB\_tree } g)) \\ A* & \xrightarrow{f} & 1 + A* \times (A \times A*)* \end{array}$$

$$\text{lsplitB\_tree } [] = i_1 \ ()$$

$$\text{lsplitB\_tree } (h : t) = i_2 \ (s, [(h, l)]) \ \textbf{where } (s, l) = \text{partB\_tree } (<h) \ t$$

$$\text{partB\_tree} :: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow ([a], [a])$$

$$\text{partB\_tree } p \ [] = ([], [])$$

$$\text{partB\_tree } p \ (h : t) \mid p \ h = \text{let } (s, l) = \text{partB\_tree } p \ t \ \text{in } (h : s, l)$$

$$\mid \text{otherwise} = \text{let } (s, l) = \text{partB\_tree } p \ t \ \text{in } (s, h : l)$$

$$\text{qSortB\_tree} :: \text{Ord } a \Rightarrow [a] \rightarrow [a]$$

$$\text{qSortB\_tree} = \text{hyloB\_tree } [\text{nil}, \text{inord}] \ \text{lsplitB\_tree}$$

**where**  $\text{inord} = \text{conc} \cdot (\text{id} \times (\text{concat} \cdot (\text{map } \text{cons})))$



$$\begin{array}{ccc}
A * & \xrightarrow{\text{lsplitB\_tree}} & 1 + A * \times (A \times A) \\
\text{ana lsplitB\_tree} \downarrow & & \downarrow \text{id} + ((\text{ana lsplitB\_tree}) \times \text{map } (\text{id} \times \text{ana lsplitB\_tree})) \\
B & \xrightleftharpoons[\text{inB\_tree}]{\text{outB\_tree}} & 1 + B \times (A \times B) \\
\text{cata inordB\_tree} \downarrow & & \downarrow \text{id} + ((\text{cata inordB\_tree}) \times \text{map } (\text{id} \times \text{cata inordB\_tree})) \\
A & \xleftarrow{\text{inordB\_tree}} & 1 + A \times (A \times A)
\end{array}$$

```

dotB_tree :: (Show a) => B-tree a -> IO ExitCode
dotB_tree = dotpict · bmap nothing (Just · init · concat · (map (++ " | ")) · (map show)) · cB_tree2Exp
cB_tree2Exp = cataB_tree [(Var "nil"), aux]
  where aux = (Term) · (id × cons) · ⟨π1 · π2, ⟨π1, π2 · π2⟩⟩ · (id × unzip)

```

## Problema 4

$$\begin{aligned}
\llbracket ga \text{ gb} \rrbracket_A &= inA \cdot (id + \llbracket ga \text{ gb} \rrbracket_A \times \llbracket ga \text{ gb} \rrbracket_B) \cdot ga \\
\llbracket ga \text{ gb} \rrbracket_B &= inB \cdot (id + \llbracket ga \text{ gb} \rrbracket_A) \cdot gb
\end{aligned}$$

$$generateAlgae = \llbracket (id + \langle id, id \rangle) \cdot outNat \ outNat \rrbracket_A$$

generateAlgae:

```

showAlgae = \ga gb => \ga gb
  where ga = ["A", conc]
        gb = ["B", id]

```

$$\begin{array}{ccc}
Algae & \xleftarrow{\text{inA}} & 1 + A \times B \\
\text{cataB\_tree lb} \downarrow & & \downarrow \text{"A"} + ga \times gb \\
String & \xleftarrow{\text{gagb}} & 1 + String \times String
\end{array}$$

```

prop_4_6 :: Integer -> Property
prop_4_6 n = (n ≥ 0 ∧ n < 24) ==> lshG n ≡ fibsucc n
  where lshG = toInteger · length · showAlgae · generateAlgae · fromInteger
        fibsucc = fib · succ

```

## Problema 5

```

permuta [] = return []
permuta l = do { (h, t) ← getR l; x ← permuta t; return (h : x) }
eliminatória (Leaf x) = return x
eliminatória (Fork (x, y)) = do { v ← eliminatória x; z ← eliminatória y; jogo (v, z) }

```

Quanto à primeira parte deste problema, fizemo-la, tal como pedido, com recurso ao monóide 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

- LaTeX, 2
  - lhs2TeX, 2
- B-tree, 4
- Cálculo de Programas, 3
  - Material Pedagógico, 2
    - BTree.hs, 4, 5
    - Exp.hs, 5
    - LTree.hs, 8
- Combinador “pointfree”
  - cata*, 7, 17
  - either*, 7, 12–17
- Função
  - $\pi_1$ , 12–17
  - $\pi_2$ , 11, 13–17
  - length*, 7, 11, 16, 17
  - map*, 11, 15–17
  - succ*, 7, 12–15, 17
  - uncurry*, 7, 13, 15–17
- Functor, 3, 5, 7–11, 17
- Graphviz, 5, 6
  - WebGraphviz, 6
- Haskell, 2, 3
  - “Literate Haskell”, 2
  - Biblioteca
    - PFP, 10
    - Probability, 8, 10
  - interpretador
    - GHCI, 3, 10
  - QuickCheck, 3, 4, 7
- L-system, 6, 7
- Programação literária, 2
- Taylor series
  - Maclaurin series, 3
- U.Minho
  - Departamento de Informática, 1
- Unix shell
  - wc*, 4
- Utilitário
  - LaTeX
    - bibtex*, 3
    - makeindex*, 3