

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

*Departamento de Informática*  
Universidade do Minho

Junho de 2017

**Grupo nr.** 30

---

A78322	André Filipe Ferreira de Mira Vieira
A77048	Eduardo Gil Ribeiro Da Rocha
A78764	Ricardo André Araújo Neves

## Contents

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

# 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 `[fontsize=] 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 `[fontsize=] ghci cp1617t.lhs` para ver que assim é:

```
[fontsize=] 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] Compil-
ing 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**:

```
import Cp
import List
import Nat
import Exp
import BTree
import LTree
import St
import Probability hiding (cond, choose)
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

---

<sup>1</sup>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 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**), [fontsize=] 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, subacção, 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

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

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
```

<sup>2</sup>Para uma breve introdução ver e.g. <https://en.wikipedia.org/wiki/QuickCheck>.

```

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

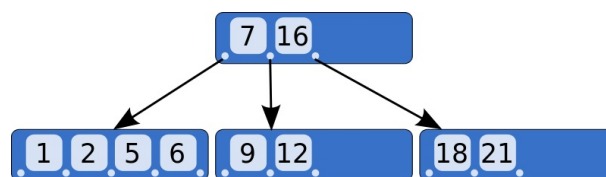
Uma “**B-tree**” é uma generalização das árvores binárias do módulo **BTree** a mais do que duas sub-árvores por nó:

```

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>



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,

```

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

```

    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 "inorder" 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
t1 = Node (6, (Node (3, (Node (2, (Empty, Empty))), Empty)), Node (7, (Empty, Node (9, (Empty, Empty)))))

```

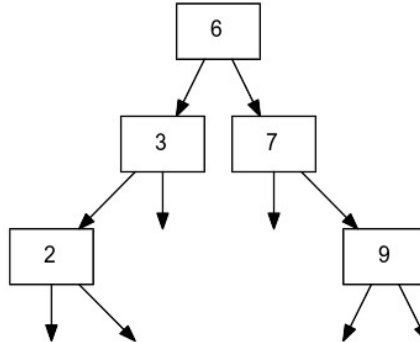
executando *dotBTree t* para

```

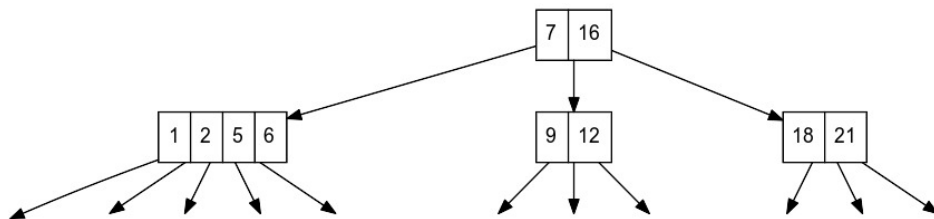
t = Node (6, (Node (3, (Node (2, (Empty, Empty))), Empty)), Node (7, (Empty, Node (9, (Empty, Empty)))))

```

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,



<sup>4</sup>Como alternativa a instalar **Graphviz**, podem usar **WebGraphviz** num browser.

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

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  $B$ ,

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

e a mesma coisa para os  $B$ s:

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

Pretende-se, neste problema:

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

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:

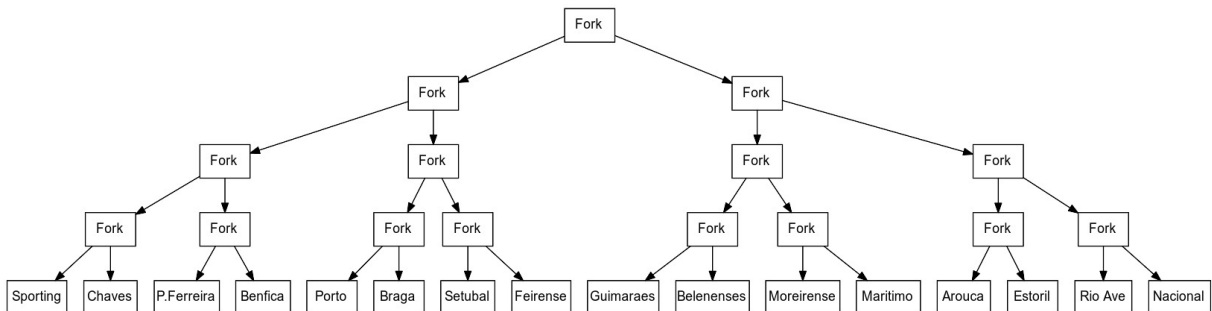
```
equipas :: [Equipa]
equipas = [
  "Arouca", "Belenenses", "Benfica", "Braga", "Chaves", "Feirense",
  "Guimaraes", "Maritimo", "Moreirense", "Nacional", "P.Ferreira",
  "Porto", "Rio Ave", "Setubal", "Sporting", "Estoril"
]
```

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,  $f("Arouca", "Braga")$  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  $\text{Dist } a$  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

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

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

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:

<i>Porto</i>	21.7%
<i>Sporting</i>	21.4%
<i>Benfica</i>	19.0%
<i>Guimaraes</i>	9.4%
<i>Braga</i>	5.1%
<i>Nacional</i>	4.9%
<i>Maritimo</i>	4.1%
<i>Belenenses</i>	3.5%
<i>Rio Ave</i>	2.3%
<i>Moreirense</i>	1.9%
<i>P.Ferreira</i>	1.4%
<i>Arouca</i>	1.4%
<i>Estoril</i>	1.4%
<i>Setubal</i>	1.4%
<i>Feirense</i>	0.7%
<i>Chaves</i>	0.4%

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

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

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 lsplit · envia · 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>.

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

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

*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

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

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].

<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 arquitetura monádica da solução.

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



## References

- [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  $E$ ,

$A$	■	2%
$B$	■	12%
$C$	■	29%
$D$	■	35%
$E$	■	22%

será representada pela distribuição

```
d1 :: Dist Char
d1 = D [( 'A' , 0.02), ( 'B' , 0.12), ( 'C' , 0.29), ( 'D' , 0.35), ( 'E' , 0.22)]
```

que o **GHCi** mostrará assim: `[fontsize=] '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 = uniform (words "Uma frase de cinco palavras")
```

isto é `[fontsize=] "Uma" 20.0"cinco" 20.0"de" 20.0"frase" 20.0"palavras" 20.0` distribuição *normais*, eg.

```
d3 = normal [10..20]
```

etc.<sup>10</sup>

`Dist` forma um **mónade** cuja unidade é `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.

## 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),
```

<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].

```

("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] → [b]
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 “lay-out” 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

$$\begin{aligned}
& \left\{ \begin{array}{l} (invAux\ x) \cdot in = [\underline{1-x}, mul \cdot \langle invAux\ x, \underline{1-x} \rangle] \\ (inv\ x) \cdot in = [\underline{1} \cdot id, add \cdot \langle invAux\ x, inv\ x \rangle] \end{array} \right. \\
& = \{ \text{Natural-id, Natural-const, Absorção-+} \} \\
& \left\{ \begin{array}{l} (invAux\ x) \cdot in = [\underline{1-x}, mul \cdot \langle id \cdot invAux\ x, \underline{1-x} \cdot inv\ x \rangle,] \\ (inv\ x) \cdot in = [\underline{1}, add] \cdot (id + \langle invAux\ x, inv\ x \rangle) \end{array} \right. \\
& = \{ \text{Natural-id, Def-x} \} \\
& \left\{ \begin{array}{l} (invAux\ x) \cdot in = [\underline{1-x} \cdot id, mul \cdot (id \times \underline{1-x}) \cdot \langle intAux\ x, inv\ x \rangle] \\ (inv\ x) \cdot in = [\underline{1}, add] \cdot (id + \langle invAux\ x, inv\ x \rangle) \end{array} \right.
\end{aligned}$$

$$= \{ \text{Natural-id, Def-x} \}$$

$$\begin{cases} (invAux\ x) \cdot \text{in} = [\underline{1-x}, mul \cdot (id \times \underline{1-x})] \cdot (id + \langle intAux\ x, inv\ x \rangle) \\ (inv\ x) \cdot \text{in} = [\underline{1}, add] \cdot (id + \langle invAux\ x, inv\ x \rangle) \end{cases}$$

$$= \{ \text{Fokkinga} \}$$

$$\langle invAux\ x, inv\ x \rangle = (\langle [\underline{1-x}, mul \cdot (id \times \underline{1-x})], [\underline{1}, add] \rangle)$$

$$= \{ \text{Lei da Troca} \}$$

$$\langle invAux\ x, inv\ x \rangle = (\langle [\underline{1-x}, \underline{1}], \langle mul \cdot (id \times (\underline{1-x})), add \rangle \rangle)$$

*soma* :: (Num a) ⇒ (a, a) → a  
*soma* (x, y) = x + y

*multiplica* :: (Num a) ⇒ (a, a) → a  
*multiplica* (x, y) = x \* y

*tiraNumero* :: Gen Float  
*tiraNumero* = choose (1, 1.99)

*arrDec* :: Float → Int → Float  
*arrDec* x n = (fromIntegral (floor (x \* (10 ↑ n)))) / (10 ↑ n)

*inv* x = for (multiplica · (id × (1 - x)), soma) (1 - x, 1)  
*retorna* x = π<sub>2</sub> · (inv x)  
*verificaInv* = **do** {  
     *x* ← *tiraNumero*;  
     return (arrDec (1 / x) 2 ≡ arrDec (retorna x 10000) 2)  
}

## Problema 2

*wc\_w\_final* :: [Char] → Int  
*wc\_w\_final* = wrapper · worker  
*wrapper* = ⊥  
*worker* = ⊥

## Problema 3

*inB\_tree* = [Nil, Block]

*out* · in = id

$$out \cdot [Nil, Block] = id$$

$$= \{ \text{Fusão-+} \}$$

$$[out \cdot Nil, out \cdot Block]$$

$$= \{ \text{Universal-+} \}$$

$$\begin{cases} id \cdot i_1 = out \cdot Nil \\ id \cdot i_2 = out \cdot Block \end{cases}$$

$$out \cdot Nil = i_1 \Rightarrow outNil = i_1()$$

$$out \cdot Block = i_2 \Rightarrow outBlockxy = i_2(x, y)$$

```

outB_tree Nil = i_1 ()
outB_tree (Block x y) = i_2 (x, y)
baseB_tree x y = id + (x × map (y × x))
recB_tree x = baseB_tree x id
cataB_tree x = x · (recB_tree (cataB_tree x)) · outB_tree
anaB_tree x = inB_tree · (recB_tree (anaB_tree x)) · x
hyloB_tree x y = cataB_tree x · anaB_tree y
instance Functor B-tree
  where fmap f = cataB_tree (inB_tree · baseB_tree id f)
inordB_tree = ⊥
largestBlock = ⊥
mirrorB_tree = ⊥
lsplitB_tree = ⊥
qSortB_tree = ⊥
dotB_tree = ⊥
cB_tree2Exp = ⊥

```

#### Problema 4

```

@C=2cm [A] [d]_([g])[r]out1 + (A × [A])[d]F ([g])
B1 + (A × B)[l]_g
@C=2cm B [r]g[d][[g]]1 + (A × B)[d]F ([g])
[A]1 + (A × [A])[l]_in

[[ga gb]]A = inA · (id + ([[ga gb]]A × [[ga gb]]B)) · ga
[[ga gb]]B = inB · (id + [[ga gb]]A) · gb

@C=2cm Int [r]x[d][[x,y]]1 + Int × Int[d]F ([x,y])
Algae1 + A × B[l]_in_A

```

```

@C=2cm Int [r]y[d][(x,y)]1 + Int[d]F [(x,y)]
B1 + A[l]-inB
@C=2cm Algae [d]-[(x,y)][r]outA1 + (Algae × B)[d]F [(x,y)]
String1 + (String × String)[l]-x
@C=2cm B [d]-[(x,y)][r]outB1 + Algae[d]F [(x,y)]
String1 + String[l]-y

genA :: Int → () + (Int, Int)
genA 0 = i1 ()
genA x = i2 (x - 1, x - 1)

genB :: Int → () + Int
genB 0 = i1 ()
genB x = outNat x

generateAlgae = [(genA genB)]A

showA :: () + (String, String) → String
showA (i1 ()) = "A"
showA (i2 (x, y)) = x ++ y

showB :: () + (String) → String
showB (i1 ()) = "B"
showB (i2 x) = x

showAlgae = [(showA showB)]A

newFib :: Int → Int
newFib 0 = 1
newFib 1 = 1
newFib x = newFib (x - 1) + newFib (x - 2)

tiraNumero2 :: Gen Int
tiraNumero2 = choose (0, 20)

verifica :: Gen Bool
verifica = do {
  z ← tiraNumero2;
  return ((length (showAlgae (generateAlgae z))) ≡ (newFib (succ z)))
}

```

## Problema 5

```

permuta [] = return []
permuta x = do { (a, b) ← getR x; x ← permuta b; return (a : x) }

probEquipa1 :: Dist Equipa → Float
probEquipa1 (D ((x, y) : ys)) = y

flatten :: LTree Equipa → [Equipa]
flatten (Leaf a) = [a]
flatten (Fork (x, y)) = (flatten x) ++ (flatten y)

```

```

eliminatoria tree = do {
   $x \leftarrow D$  [
    ( $e_1, ((\text{probEquipa1 } (\text{jogo } (e_1, e_2))) * ((\text{probEquipa1 } (\text{jogo } (e_1, e_3))) + (\text{probEquipa1 } (\text{jogo } (e_1, e_4)))) * ((\text{probEquipa1 } (\text{jogo } (e_2, e_1))) * ((\text{probEquipa1 } (\text{jogo } (e_2, e_3))) + (\text{probEquipa1 } (\text{jogo } (e_2, e_4)))) * ((\text{probEquipa1 } (\text{jogo } (e_3, e_4))) * ((\text{probEquipa1 } (\text{jogo } (e_3, e_1))) + (\text{probEquipa1 } (\text{jogo } (e_3, e_2)))) * ((\text{probEquipa1 } (\text{jogo } (e_4, e_3))) * ((\text{probEquipa1 } (\text{jogo } (e_4, e_1))) + (\text{probEquipa1 } (\text{jogo } (e_4, e_2)))) * ((\text{probEquipa1 } (\text{jogo } (e_5, e_6))) * ((\text{probEquipa1 } (\text{jogo } (e_5, e_7))) + (\text{probEquipa1 } (\text{jogo } (e_5, e_8)))) * ((\text{probEquipa1 } (\text{jogo } (e_6, e_5))) * ((\text{probEquipa1 } (\text{jogo } (e_6, e_7))) + (\text{probEquipa1 } (\text{jogo } (e_6, e_8)))) * ((\text{probEquipa1 } (\text{jogo } (e_7, e_8))) * ((\text{probEquipa1 } (\text{jogo } (e_7, e_5))) + (\text{probEquipa1 } (\text{jogo } (e_7, e_6)))) * ((\text{probEquipa1 } (\text{jogo } (e_8, e_7))) * ((\text{probEquipa1 } (\text{jogo } (e_8, e_5))) + (\text{probEquipa1 } (\text{jogo } (e_8, e_6)))) * ((\text{probEquipa1 } (\text{jogo } (e_9, e_{10}))) * ((\text{probEquipa1 } (\text{jogo } (e_9, e_{11}))) + (\text{probEquipa1 } (\text{jogo } (e_9, e_{12})))) * ((\text{probEquipa1 } (\text{jogo } (e_{10}, e_9))) * ((\text{probEquipa1 } (\text{jogo } (e_{10}, e_{11}))) + (\text{probEquipa1 } (\text{jogo } (e_{10}, e_{12})))) * ((\text{probEquipa1 } (\text{jogo } (e_{11}, e_{12}))) * ((\text{probEquipa1 } (\text{jogo } (e_{11}, e_9))) + (\text{probEquipa1 } (\text{jogo } (e_{11}, e_{10})))) * ((\text{probEquipa1 } (\text{jogo } (e_{12}, e_{11}))) * ((\text{probEquipa1 } (\text{jogo } (e_{12}, e_9))) + (\text{probEquipa1 } (\text{jogo } (e_{12}, e_{10})))) * ((\text{probEquipa1 } (\text{jogo } (e_{13}, e_{14}))) * ((\text{probEquipa1 } (\text{jogo } (e_{13}, e_{15}))) + (\text{probEquipa1 } (\text{jogo } (e_{13}, e_{16})))) * ((\text{probEquipa1 } (\text{jogo } (e_{14}, e_{13}))) * ((\text{probEquipa1 } (\text{jogo } (e_{14}, e_{15}))) + (\text{probEquipa1 } (\text{jogo } (e_{14}, e_{16})))) * ((\text{probEquipa1 } (\text{jogo } (e_{15}, e_{16}))) * ((\text{probEquipa1 } (\text{jogo } (e_{15}, e_{13}))) + (\text{probEquipa1 } (\text{jogo } (e_{15}, e_{14})))) * ((\text{probEquipa1 } (\text{jogo } (e_{16}, e_{15}))) * ((\text{probEquipa1 } (\text{jogo } (e_{16}, e_{13}))) + (\text{probEquipa1 } (\text{jogo } (e_{16}, e_{14})))) * ((\text{probEquipa1 } (\text{jogo } (e_{16}, e_{15}))))
  ]
  return  $x$  }
  where
    ( $e_1 : e_2 : e_3 : e_4 : e_5 : e_6 : e_7 : e_8 : e_9 : e_{10} : e_{11} : e_{12} : e_{13} : e_{14} : e_{15} : e_{16} : ys$ ) = flatten tree$ 
```

## Index

- LaTeX, 2
  - lhs2TeX, 2
- B-tree, 4
- Cálculo de Programas, 3
  - Material Pedagógico, 2
  - BTree.hs, 4, 5
  - Exp.hs, 5
  - LTree.hs, 8, 9
- Combinador “pointfree”
  - cata*, 7
  - either*, 7
- Função
  - $\pi_2$ , 11
  - length*, 7, 11
  - map*, 11
  - succ*, 7
  - uncurry*, 7
- Functor, 3, 5, 7–11
- 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