

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

*Departamento de Informática*  
Universidade do Minho

Junho de 2017

<b>Grupo nr.</b>	<b>37</b>
a67637	Célia Figueiredo
a67662	Adriana Pereira
a60993	Luís Pedro Fonseca

## 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**:

```
import Cp
import List
```

---

<sup>1</sup>O suffixo ‘lhs’ quer dizer *literate Haskell*.

```

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

```

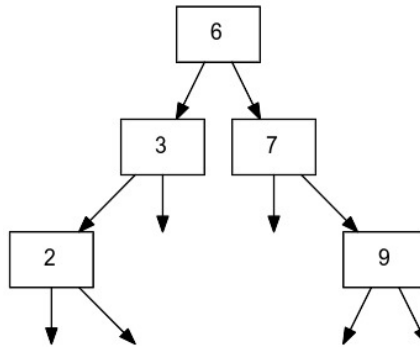
executando *dotBTree* *t* para

```

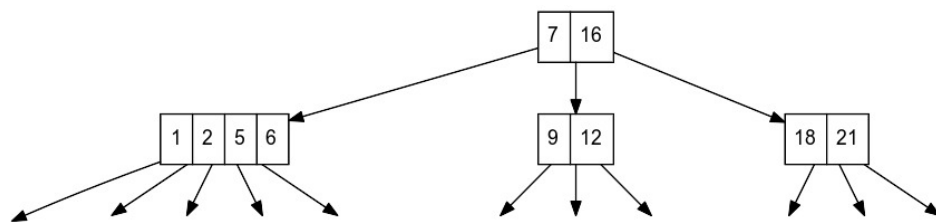
t = Node (6, (Node (3, (Node (2, (Empty, Empty)), Empty)), Node (7, (Empty, Node (9, (Empty, Empty))))))

```

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  $B$ ,

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

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:

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

```
generateAlgae :: Int -> Algae
```

como anamorfismo de  $Algae$  e da função

```
showAlgae :: Algae -> 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             21.7%  
 Sporting         21.4%  
 Benfica         19.0%  
 Guimaraes      9.4%  
 Braga           5.1%  
 Nacional        4.9%  
 Maritimo        4.1%  
 Belenenses     3.5%  
 Rio Ave         2.3%  
 Moreirense     1.9%  
 P.Ferreira      1.4%  
 Arouca         1.4%  
 Estoril         1.4%  
 Setubal         1.4%  
 Feirense       0.7%  
 Chaves         0.4%

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

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

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

<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.



O anamorfismo *sorteio* :: [Equipa] → LTree Equipa tem a seguinte arquitectura,<sup>8</sup>

$$\text{sorteio} = \text{anaLTree } \text{lsplit} \cdot \text{envia} \cdot \text{permuta}$$

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

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

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

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

*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

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

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].

## 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.

---

<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**.

# 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:

```
'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

```
"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>

$\text{Dist}$  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 saber 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

```
inv x n = π2 $ for (invAccum x) (0, 0) n
where invAccum x (it, accum) = (it + 1, accum + (1 - x) ** it)
```

A sua solução em C pode ser definida da seguinte forma:

```

#include <math.h>

float inv(float x, int n){
    float accum = 0;
    for(int i = 0; i < n; i++){
        accum += pow(1 - x, i);
    }
    return accum;
}

```

## Problema 2

```

-
wrapper :: [Int] -> Int wrapper = cataList (either (const 0) (uncurry (+)))
worker :: [Char] -> [Int] worker [] = [] worker [c] = [1] otherwise = [0] worker (c1:c2:cs)
= (sep c1) ^& sep (c2) = 1 : worker (c2 : cs) otherwise = 0 : worker (c2:cs)
-

```

```

wc_w_final :: [Char] -> Int
wc_w_final = wrapper . worker

wrapper = π2
worker :: [Char] -> (Bool, Int)
worker = cataList [(True, 0), wordAccum]

wordAccum :: (Char, (Bool, Int)) -> (Bool, Int)
wordAccum (a, (b, n))
| b ^& (¬ · sep) a = (sep a, succ n)
| otherwise = (sep a, n)
where sep c = c ∈ " \t\n "

```

```

wc_w_final = wrapper.worker
= { justificação ..... }
alpha · ⟨p · f, f⟩
= { justificação ..... }
alpha · (id × f) · ⟨p · f, id⟩ --- etc ---

```

## Problema 3

Para gerar o gráfico basta correr este comando: `dot -Tpng digraphG.dot -o diagrama.png`

```

inB_tree :: () + (B-tree a, [(a, B-tree a)]) -> (B-tree a)
inB_tree = [Nil, joinB_trees]

joinB_trees :: (B-tree a, [(a, B-tree a)]) -> B-tree a
joinB_trees (t1, ts) = Block { leftmost = t1, block = ts }

```

Uma versão alternativa seria:

```

inB_tree :: Either () (B_tree a, [(a, B_tree a)]) -> B_tree a
inB_tree = either (const Nil) (uncurry Block)

```

```

outB_tree :: B-tree a -> () + (B-tree a, [(a, B-tree a)])
outB_tree Nil = i1 ()
outB_tree (Block left block) = i2 (left, block)

```

```

recB_tree :: (b -> d) -> z + (b, [(a, b)]) -> z + (d, [(a, d)])
recB_tree f = id + (f × (map (id × f)))

```

```
baseB_tree :: (a1 → b1) → (a → d) → b + (a, [(a1, a)]) → b + (d, [(b1, d)])
baseB_tree g f = id + (f × (map (g × f)))
```

```
cataB_tree :: ((() + (d, [(a, d)]) → d) → B-tree a → d)
cataB_tree g = g · (recB_tree (cataB_tree g)) · outB_tree
anaB_tree :: (b → () + (b, [(a, b)])) → b → B-tree a
anaB_tree g = inB_tree · (recB_tree (anaB_tree g)) · g
hyloB_tree :: ((() + (c, [(a, c)]) → c) → (b → () + (b, [(a, b)])) → b → c)
hyloB_tree f g = cataB_tree f · anaB_tree g
```

**instance** *Functor* B-tree

**where** fmap f = cataB\_tree (inB\_tree · baseB\_tree f id)

```
-- esq meio dir
inordB_tree :: B-tree t → [t]
inordB_tree = cataB_tree ordgene
ordgene = [[], join]
where join :: ([a], [(a, [a])]) → [a]
join (xs, ys) = xs ++ concat (map (λ(n, zs) → (n : zs)) ys)
```

```
largestBlock :: B-tree t → Int
largestBlock = cataB_tree [0, size]
where size :: (Int, [(a, Int)]) → Int
size (x, ys) = max x (length ys)
```

```
mirrorB_tree :: B-tree a → B-tree a
mirrorB_tree = cataB_tree (inB_tree · (id + (id × reverse)))
```

```
dotB_tree :: Show a ⇒ B-tree a → IO ExitCode
dotB_tree = dotpict · bmap nothing (Just · show) · cB_tree2Exp
cB_tree2Exp :: B-tree a → Exp [Char] [a]
cB_tree2Exp = cataB_tree [(Var "nil"), connect]
connect :: (Exp v [a], [(a, Exp v [a])]) → Exp v [a]
connect (x, xs) = Term (map π1 xs) (x : (map π2 xs))
```

```
qSortB_tree :: Ord a => [a] -> [a]
qSortB_tree = hyloB_tree ordgene lsplitB_tree
```

```
lsplitB_tree :: [a] -> Either () ([a], [(a, [a])])
lsplitB_tree [] = i1 ()
lsplitB_tree [x] = i2 ([], (x, []))
lsplitB_tree (x:xs) = undefined
```

```
qsep [] = Left ()
qsep (h:t) = Right (h, (s,l)) where (s,l) = part (<h) t
```

```
part :: (a -> Bool) -> [a] -> ([a], [a])
part p [] = ([], [])
part p (h:t) | p h = let (s,l) = part p t in (h:s,l)
               | otherwise = let (s,l) = part p t in (s,h:l)
```

## Problema 4

$$\begin{aligned} \llbracket \cdot \rrbracket_A &:: (a \rightarrow 1 + (a, b)) \rightarrow (b \rightarrow 1 + a) \rightarrow a \rightarrow A \\ \llbracket ga \text{ } gb \rrbracket_A &= inA \cdot (id + \llbracket ga \text{ } gb \rrbracket_A \times \llbracket ga \text{ } gb \rrbracket_B) \cdot ga \\ \llbracket \cdot \rrbracket_B &:: (a \rightarrow 1 + (a, b)) \rightarrow (b \rightarrow 1 + a) \rightarrow b \rightarrow B \\ \llbracket ga \text{ } gb \rrbracket_B &= inB \cdot (id + \llbracket ga \text{ } gb \rrbracket_A) \cdot gb \end{aligned}$$

```
generateAlgae :: Int → Algae
generateAlgae = \llbracket ga \text{ } gb \rrbracket_A \cdot succ
  where ga 0 = i1 ()
        ga n = i2 (pred n, pred n)
        gb 0 = i1 ()
        gb n = i2 (pred n)
showAlgae :: Algae → String
showAlgae = \llbracket ga \text{ } gb \rrbracket_A
  where ga = ["A", (++)]
        gb = ["B", id]
```

## Problema 5

```
permuta :: [a] → IO [a]
permuta [] = return []
permuta input = do
  (x, xs) ← getR input
  permuted ← permuta xs
  return (x : permuted)
```

```
eliminatoria :: LTree Equipa → Dist Equipa
eliminatoria = cataLTree g
  where g :: Equipa + (Dist Equipa, Dist Equipa) → Dist Equipa
        g = [return, distJogo]
        distJogo :: (Dist Equipa, Dist Equipa) → Dist Equipa
        distJogo (de1, de2) = do
          e1 ← de1
          e2 ← de2
          jogo (e1, e2)
```

# 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, 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