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

Departamento de Informática
Universidade do Minho

Junho de 2017

Grupo nr.	G19
a78985	Diana Costa
a78203	Paulo Mendes
a76945	Tânia Silva

Conteúdo

1 Preâmbulo

A disciplina de Cálculo de Programas tem como objectivo principal ensinar a programação de computadores como uma disciplina científica. Para isso parte-se de um repertório de *combinadores* que formam uma álgebra da programação (conjunto de leis universais e seus corolários) e usam-se esses combinadores para construir programas *composicionalmente*, isto é, agregando programas já existentes.

Na sequência pedagógica dos planos de estudo dos dois cursos que têm esta disciplina, restringe-se a aplicação deste método ao desenvolvimento de programas funcionais na linguagem Haskell.

O presente trabalho tem por objectivo concretizar na prática os objectivos da disciplina, colocando os alunos perante problemas de programação que deverão ser abordados composicionalmente e implementados em Haskell. Há ainda um outro objectivo: o de ensinar a documentar programas e a produzir textos técnico-científicos de qualidade.

2 Documentação

Para cumprir de forma integrada os objectivos enunciados acima vamos recorrer a uma técnica de programação dita "literária" [3], cujo princípio base é o seguinte:

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

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

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

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

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

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

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

```
ghci cp1617t.lhs
```

para ver que assim é:

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

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

import *Cp*
import *List*

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

import $\mathbb{N}_0$
import *Exp*
import BTree
import LTree
import *St*
import *Probability hiding* $(\cdot \to \cdot, \cdot)$
import *Data.List*
import *Test.QuickCheck hiding* $((\times))$
import *System.Random hiding* $\langle \cdot, \cdot \rangle$
import *GHC*.IO.*Exception*
import *System*.IO.*Unsafe*

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

```
\begin{code}
...
\end{code}
```

vai ser seleccionado pelo GHCi para ser executado.

3 Como realizar o trabalho

Este trabalho teórico-prático deve ser realizado por grupos de três alunos. Os detalhes da avaliação (datas para submissão do relatório e sua defesa oral) são os que forem publicados na página da disciplina na *internet*. Recomenda-se uma abordagem equilibrada e participativa dos membros do grupo de trabalho por forma a poderem responder às questões que serão colocadas na defesa oral do relatório.

Em que consiste, então, o *relatório* a que se refere o parágrafo anterior? É a edição do texto que está a ser lido, preenchendo o anexo C com as suas respostas. O relatório deverá conter ainda a identificação dos membros do grupo de trabalho, no local respectivo da folha de rosto.

Para gerar o PDF integral do relatório deve-se ainda correr os comando seguintes, que actualizam a bibliografia (com BibTEX) e o índice remissivo (com `makeindex`),

```
bibtex cp1617t.aux
makeindex cp1617t.idx
```

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

Problema 1

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

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

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

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

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

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

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

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

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

Problema 2

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

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

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

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

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

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

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

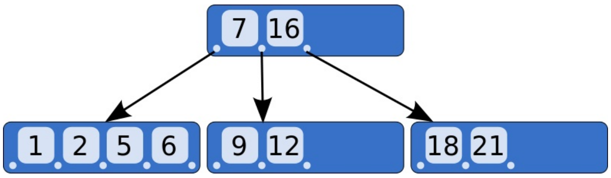
Problema 3

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

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

Por exemplo, a B-tree[3]

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



é representada no tipo acima por:

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

Pretende-se, neste problema:

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

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

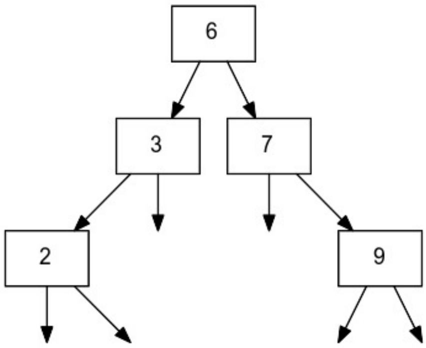
6. A biblioteca Exp permite representar árvores-expressão em formato DOT, que pode ser lido por aplicações como por exemplo Graphviz, produzindo as respectivas imagens. Por exemplo, para o caso de árvores BTree, se definirmos

$$
\begin{array}{l}
\mathit{dotBTree} :: \mathit{Show}\ a \Rightarrow \mathsf{BTree}\ a \rightarrow \mathsf{IO}\ \mathit{ExitCode}\\
\mathit{dotBTree} = \mathit{dotpict} \cdot \mathit{bmap}\ \mathit{nothing}\ (\mathit{Just} \cdot \mathit{show}) \cdot \mathit{cBTree2Exp}
\end{array}
$$

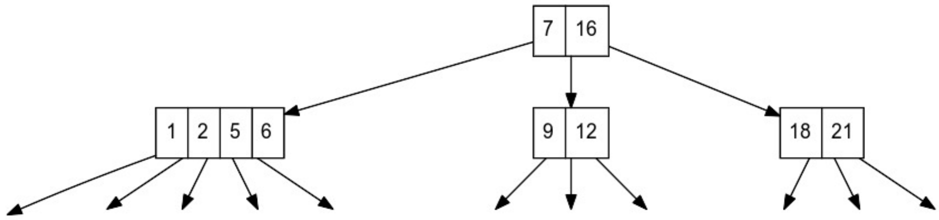
executando $\mathit{dotBTree}\ t$ para

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

obter-se-á a imagem



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



para a árvora dada acima.

Problema 4

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

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

Variáveis: A e B
Constantes: nenhuma
Axioma: A
Regras: $A \rightarrow A\ B, B \rightarrow A$.

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

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

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

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

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

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

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

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

e a mesma coisa para os Bs:

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

Pretende-se, neste problema:

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

   $$generateAlgae :: Int \to Algae$$

   como anamorfismo de *Algae* e da função

   $$showAlgae :: Algae \to String$$

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

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

Problema 5

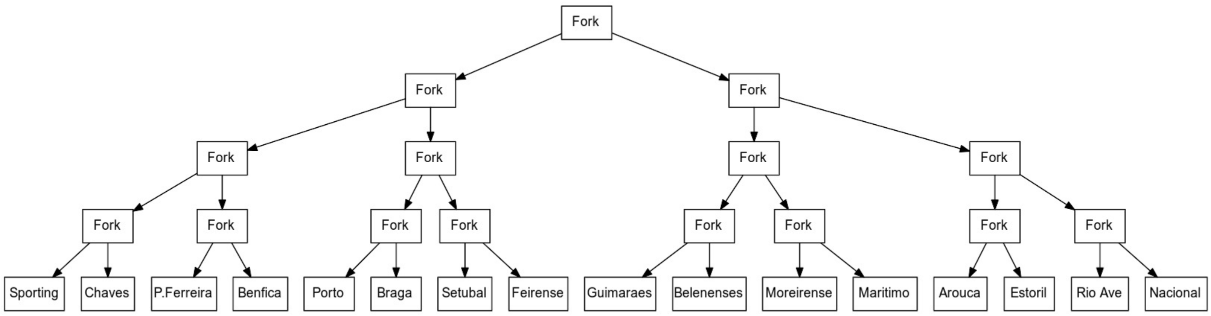
O ponto de partida deste problema é um conjunto de equipas de futebol, por exemplo:

$$\begin{array}{l} equipas :: [Equipa] \\ equipas = [\end{array}$$
```
  "Arouca","Belenenses","Benfica","Braga","Chaves","Feirense",
  "Guimaraes","Maritimo","Moreirense","Nacional","P.Ferreira",
  "Porto","Rio Ave","Setubal","Sporting","Estoril"
  ]
```

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

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

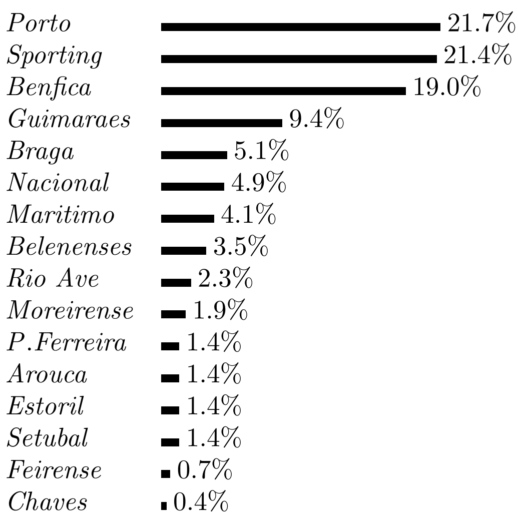
Para lidarmos com probabilidades vamos usar o mónade Dist a que vem descrito no apêndice A e que está implementado na biblioteca Probability [1] — ver definição (1) mais adiante. A primeira parte do problema consiste em sortear *aleatoriamente* os jogos das equipas. O resultado deverá ser uma LTree contendo,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



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

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

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



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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Referências

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

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

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

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

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

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

Anexos

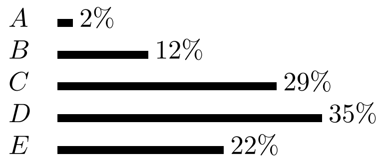
A Mónade para probabilidades e estatística

Mónades são functores com propriedades adicionais que nos permitem obter efeitos especiais em programação. Por exemplo, a biblioteca Probability oferece um mónade para abordar problemas de probabilidades. Nesta biblioteca, o conceito de distribuição estatística é captado pelo tipo

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

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

Cada par (a, p) numa distribuição $d :: \mathsf{Dist}\ a$ indica que a probabilidade de a é p, devendo ser garantida a propriedade de que todas as probabilidades de d somam 100%. Por exemplo, a seguinte distribuição de classificações por escalões de A a E,



será representada pela distribuição

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

que o GHCi mostrará assim:

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

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

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

isto é

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

distribuição *normais*, eg.

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

etc.[10]

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

Este mónade é adequado à resolução de problemas de *probabilidades e estatística* usando programação funcional, de forma elegante e como caso particular de programação monádica.

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

B Definições auxiliares

São dadas: a função que simula jogos entre equipas,

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

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

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

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

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

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

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

C Soluções propostas

Os alunos devem colocar neste anexo as suas soluções aos exercícios propostos, de acordo com o "layout" que se fornece. Não podem ser alterados os nomes das funções dadas, mas pode ser adicionado texto e / ou outras funções auxiliares que sejam necessárias.

Problema 1

A resolução deste problema consistiu essencialmente em três partes: a definição de *inv x* como uma função em *Haskell*, todo o raciocínio envolvente até chegar à solução final (com o auxílio da Lei *Fokkinga*), e a conversão da solução para um ciclo-*for*, tal como era pedido no enunciado.

Assim, e como primeira tarefa, resultou a seguinte definição de *inv x*:

$$
\begin{array}{l}
\mathit{inv1}\ x\ 0 = 1 \\
\mathit{inv1}\ x\ n = (\mathit{macL}\ x\ n) + (\mathit{inv1}\ x\ (n-1)) \\
\quad \mathbf{where} \\
\qquad \mathit{macL}\ x\ 0 = 1 \\
\qquad \mathit{macL}\ x\ n = (1-x) * \mathit{macL}\ x\ (n-1)
\end{array}
$$

Esta definição teve de ser posteriormente modificada, para ser mais fácil a resolução da segunda parte do problema, e para definir em *Point free*, sendo que daí resultou:

$$
\begin{array}{l}
\mathit{inv2}\ x\ 0 = 1 \\
\mathit{inv2}\ x\ (n+1) = (\mathit{macL}\ x\ (n+1)) + (\mathit{inv2}\ x\ (n)) \\
\quad \mathbf{where} \\
\qquad \mathit{macL}\ x\ 0 = 1 \\
\qquad \mathit{macL}\ x\ (n+1) = (1-x) * \mathit{macL}\ x\ (n)
\end{array}
$$

A segunda parte do problema incluiria todo o racíocínio, com o auxílio da Lei de *Fokkinga*, para chegar ao catamorfismo correspondente à função *inv x*. (Os raciocínios terão uma linguagem e apresentação mais legível e "corriqueira", e a resposta exata ao problema terá o formato correto da UC de Cálculo de Programas.)

$$
\begin{array}{ll}
& \begin{cases} f \cdot \mathbf{in} = h \cdot F\ \langle f, g \rangle \\ g \cdot \mathbf{in} = k \cdot F\ \langle f, g \rangle \end{cases} \\
<=> & \quad \{\ \mathbf{in} = [\underline{0}, \mathsf{succ}];\ \text{f=inv};\ \text{g=macL};\ F\ \langle f, g \rangle \text{=} F\ \langle inv, macL \rangle \text{=} (id + \langle inv, macL \rangle)\ \} \\
& \begin{cases} inv \cdot [\underline{0}, \mathsf{succ}] = h \cdot (id + \langle inv, macL \rangle) \\ macL \cdot [\underline{0}, \mathsf{succ}] = k \cdot (id + \langle inv, macL \rangle) \end{cases}
\end{array}
$$

Para completar a lei de *Fokkinga*, é necessário deduzir h e k das funções inv e macL. Segue-se a dedução de h, a partir de *inv x*:

$$
\begin{array}{ll}
& \begin{cases} \mathit{inv2}\ x \cdot \underline{0} = 1 \\ \mathit{inv2}\ x \cdot \mathsf{succ} = add \cdot \langle macL\ x, \mathit{inv2}\ x \rangle \end{cases} \\
<=> & \quad \{\ \mathit{Universal}{-}{+}\ \} \\
& \mathit{inv2}\ x \cdot [\underline{0}, \mathsf{succ}] = [\underline{1}, add \cdot \langle macL\ x, \mathit{inv2}\ x \rangle] \\
<=> & \quad \{\ \mathit{Natural} - id;\ \text{Definição de macL x}\ \} \\
& \mathit{inv2}\ x \cdot [\underline{0}, \mathsf{succ}] = [\underline{1} \cdot id, add \cdot \langle ((1-x)*)\ macL\ x, \mathit{inv2}\ x \rangle] \\
<=> & \quad \{\ \mathit{Absorção} - x\ \} \\
& \mathit{inv2}\ x \cdot [\underline{0}, \mathsf{succ}] = [\underline{1} \cdot id, add \cdot (((1-x)*) \times id) \cdot \langle macL\ x, \mathit{inv2}\ x \rangle] \\
<=> & \quad \{\ \mathit{Absorção}{-}{+}\ \} \\
& \mathit{inv2}\ x \cdot [\underline{0}, \mathsf{succ}] = [\underline{1}, add \cdot (((1-x)*) \times id)] \cdot (id + \langle macL\ x, \mathit{inv2}\ x \rangle)
\end{array}
$$

Logo,

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

Do mesmo modo que se procedeu para h, segue-se a dedução de k:

$$
\begin{cases} macL\ x \cdot \underline{0} = 1 \\ macL\ x \cdot \mathsf{succ} = (1-x) * (macL\ x) \end{cases}
$$

$$
\begin{aligned}
<=> \quad & \{\ Universal{-}+\ \} \\
& macL\ x \cdot [\underline{0}, \mathsf{succ}] = [\underline{1}, (1-x) * (macL\ x)] \\
<=> \quad & \{\ Natural - id;\ Cancelamento - x\ \} \\
& macL\ x \cdot [\underline{0}, \mathsf{succ}] = [\underline{1} \cdot id, ((1-x)*) \cdot \pi_1 \cdot \langle macL\ x, inv2\ x \rangle] \\
<=> \quad & \{\ Absorção{-}+\ \} \\
& macL\ x \cdot [\underline{0}, \mathsf{succ}] = [\underline{1}, ((1-x)*) \cdot \pi_1] \cdot (id + \langle macL\ x, inv2\ x \rangle)
\end{aligned}
$$

Assim,

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

Ora, pela Lei de *Fokkinga*, podemos concluir que

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

$$
\begin{aligned}
<=> \quad & \{\ \text{Fokkinga}\ \} \\
& \langle inv, macL \rangle = (\!| \langle h, k \rangle |\!)
\end{aligned}
$$

Tendo h e k já definidos, chegamos ao catamorfismo de *inv x*:

$invcata\ x = \pi_2 \cdot (\!| \langle [\underline{(1)}, ((1-x)*) \cdot \pi_1], [\underline{(1)}, \widehat{(+)} \cdot (((1-x)*) \times id)] \rangle |\!)$

Finalmente, a última tarefa consistia em provar que *inv x* seria um ciclo-*for*. Bastou a definição de ciclo-*for* e o uso da Lei da Troca para este último passo:

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

Finalmente, obtemos a solução ao problema,

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

<u>Nota</u>: Teste *QuickCheck*.

Neste teste foi necessário restringir o x entre 1 e 2, como pedido no enunciado. Para além disso, como *inv x* dá uma aproximação de 1/x, era preciso ter em conta um erro pequeno de cálculo, daí o grupo ter utilizado o número 0.000000000000009. Por fim, foi decidido que seria melhor testar com 50000 iterações.

$prop\_Inv\ x = (x > 1 \wedge x < 2) ==> abs\ ((inv\ (inv\ x\ 50000)\ 50000) - x) < 0.000000000000009$

Problema 2

Para o problema 2 era requerido que fosse definida a função *wc c* segundo o modelo *worker/wrapper*, onde o *wrapper* seria um catamorfismo de listas. Para isto, como primeira instância, foram definidas as funções *wc c* e *lookahead sep* em *Point Free* para ajudar à resolução, compreensão e testes do exercício, e, de seguida, foi aplicada a Lei da Recursividade Múltipla (ou Fokkinga) às mesmas funções.

Antes de mais, são apresentadas as definições das funções acima mencionadas, mais a definição de *sep*, que foram usadas para testes e para clarificar a linha de raciocínio do grupo antes da resolução do problema:

$$
\begin{array}{l}
\mathit{lh\_pointfree} :: [\,\mathit{Char}\,] \to \mathit{Bool} \\
\mathit{lh\_pointfree} = [\underline{\mathit{True}}, \mathit{sep} \cdot \pi_1] \cdot \mathit{outList} \\
\mathit{wc\_w\_pointfree} :: [\,\mathit{Char}\,] \to \mathit{Int} \\
\mathit{wc\_w\_pointfree} = [\underline{0}, \mathit{h2}] \cdot (\mathit{id} + \mathit{id} \times \langle \mathit{wc\_w\_pointfree}, \mathit{lh\_pointfree} \rangle) \cdot \mathit{outList} \\
\quad \mathbf{where}\ \mathit{h2} = \widehat{((\wedge)} \cdot ((\neg \cdot \mathit{sep}) \times \pi_2)) \to (\mathsf{succ} \cdot \pi_1 \cdot \pi_2), (\pi_1 \cdot \pi_2) \\
\mathit{sep} :: \mathit{Char} \to \mathit{Bool} \\
\mathit{sep}\ c = (c \equiv \texttt{'\ '} \vee c \equiv \texttt{'\textbackslash n'} \vee c \equiv \texttt{'\textbackslash t'})
\end{array}
$$

No que toca à resolução do problema, o grupo começou pela Lei de Fokkinga, como é apresentado a seguir. É de salientar a alteração do nome da função *wc w* para *wc* e da função *lookahead sep* para *lh*, por forma a facilitar a leitura e compreensão do racíocínio e cálculos.

$$
\begin{array}{ll}
& \begin{cases} f \cdot \mathbf{in} = h \cdot F\ \langle f, g \rangle \\ g \cdot \mathbf{in} = k \cdot F\ \langle f, g \rangle \end{cases} \\
<=> & \quad \{\ \mathbf{in} = [\mathit{nil}, \mathit{cons}];\ \text{f=wc; g=lh; } F\ \langle f, g \rangle\text{=}F\ \langle \mathit{wc}, \mathit{lh} \rangle\text{=}(\mathit{id} + \mathit{id} \times \langle \mathit{wc}, \mathit{lh} \rangle)\ \} \\
& \begin{cases} \mathit{wc} \cdot [\mathit{nil}, \mathit{cons}] = h \cdot (\mathit{id} + \mathit{id} \times \langle \mathit{wc}, \mathit{lh} \rangle) \\ \mathit{lh} \cdot [\mathit{nil}, \mathit{cons}] = k \cdot (\mathit{id} + \mathit{id} \times \langle \mathit{wc}, \mathit{lh} \rangle) \end{cases} \\
<=> & \quad \{\ \text{Def-+ (x2)}\ \} \\
& \begin{cases} \mathit{wc} \cdot [\mathit{nil}, \mathit{cons}] = h \cdot [i_1 \cdot \mathit{id}, (i_2 \cdot \mathit{id}) \times \langle \mathit{wc}, \mathit{lh} \rangle] \\ \mathit{lh} \cdot [\mathit{nil}, \mathit{cons}] = k \cdot [i_1 \cdot \mathit{id}, (i_2 \cdot \mathit{id}) \times \langle \mathit{wc}, \mathit{lh} \rangle] \end{cases} \\
<=> & \quad \{\ \text{Fusão-+};\ \mathit{Natural} - \mathit{id}\ \} \\
& \begin{cases} \mathit{wc} \cdot [\mathit{nil}, \mathit{cons}] = [h \cdot i_1, (h \cdot i_2) \times \langle \mathit{wc}, \mathit{lh} \rangle] \\ \mathit{lh} \cdot [\mathit{nil}, \mathit{cons}] = [k \cdot i_1, (k \cdot i_2) \times \langle \mathit{wc}, \mathit{lh} \rangle] \end{cases}
\end{array}
$$

Neste ponto, é necessário aplicar a Lei Eq-+ a ambas as condições do sistema. Comecemos pela primeira condição:

$$
\begin{array}{ll}
& [\mathit{wc} \cdot \mathit{nil}, \mathit{wc} \cdot \mathit{cons}] = [h \cdot i_1, (h \cdot i_2) \times \langle \mathit{wc}, \mathit{lh} \rangle] \\
<=> & \quad \{\ \text{Eq-+}\ \} \\
& \begin{cases} \mathit{wc} \cdot \mathit{nil} = h \cdot i_1 \\ \mathit{wc} \cdot \mathit{cons} = (h \cdot i_2) \cdot (\mathit{id} \times \langle \mathit{wc}, \mathit{lh} \rangle) \end{cases} \\
<=> & \quad \{\ \mathit{wc} \cdot \mathit{nil} = 0;\ \mathit{wc} \cdot \mathit{cons} = ((\neg \cdot \mathit{sep} \cdot \pi_1 \wedge \mathit{lh} \cdot \pi_2) \to ((\mathit{wc} \cdot \pi_2) + 1), (\mathit{wc} \cdot \pi_2));\ h = [\mathit{h1}, \mathit{h2}]\ \} \\
& \begin{cases} \mathit{h1} = 0 \\ \mathit{h2} \cdot (\mathit{id} \times \langle \mathit{wc}, \mathit{lh} \rangle) = \widehat{((\wedge)} \cdot \langle (\neg \cdot \mathit{sep} \cdot \pi_1)\ (\mathit{lh} \cdot \pi_2), \cdot \rangle) \to ((\mathit{wc} \cdot \pi_2) + 1), (\mathit{wc} \cdot \pi_2) \end{cases}
\end{array}
$$

Para descobrir h2 é necessária a 2ªLei de fusão do condicional e a Lei de Leibniz, usadas na seguinte prova:

$$h2 \cdot (id \times \langle wc, lh \rangle) = \widehat{(\wedge)} \cdot \langle \neg \cdot sep \cdot \pi_1, lh \cdot \pi_2 \rangle \rightarrow (wc \cdot \pi_2) + 1, wc \cdot \pi_2$$

<=> { $Cancelamento - x$; Definição de succ }

$$h2 \cdot (id \times \langle wc, lh \rangle) = \widehat{(\wedge)} \cdot \langle \neg \cdot sep \cdot \pi_1, \pi_2 \cdot \langle wc, lh \rangle \cdot \pi_2 \rangle \rightarrow \mathsf{succ} \cdot \pi_1 \cdot \langle wc, lh \rangle \cdot \pi_2, \pi_1 \cdot \langle wc, lh \rangle \cdot \pi_2$$

<=> { $Fus\tilde{a}o - x$; $Reflex\tilde{a}o - x$; $Natural - \pi_1$; $Natural - \pi_2$; $Cancelamento - x$ }

$$h2 \cdot (id \times \langle wc, lh \rangle) = \widehat{(\wedge)} \cdot ((\neg\ sep) \times (\pi_2 \cdot \langle wc, lh \rangle)) \rightarrow \mathsf{succ} \cdot \pi_1 \cdot \pi_2 \cdot (id \times \langle wc, lh \rangle), \pi_1 \cdot \pi_2 \cdot (id \times \langle wc, lh \rangle)$$

<=> { $Functor - x$ }

$$h2 \cdot (id \times \langle wc, lh \rangle) = \widehat{(\wedge)} \cdot (((\neg\ sep \cdot \pi_1) \times \pi_2) \cdot (id \times \langle wc, lh \rangle)) \rightarrow \mathsf{succ} \cdot \pi_1 \cdot \pi_2 \cdot (id \times \langle wc, lh \rangle), \pi_1 \cdot \pi_2 \cdot (id \times \langle wc, lh \rangle)$$

<=> { 2ªLei de fusão do condicional; Lei de Leibniz }

$$h2 = (\widehat{(\wedge)} \cdot ((\neg \cdot sep) \times \pi_2)) \rightarrow (\mathsf{succ} \cdot \pi_1 \cdot \pi_2), (\pi_1 \cdot \pi_2)$$

Conclui-se assim que

$$h = [h1, h2]$$

<=> { Definição de h1 e h2 }

$$h = [\underline{0}, (\widehat{(\wedge)} \cdot ((\neg \cdot sep) \times \pi_2)) \rightarrow (\mathsf{succ} \cdot \pi_1 \cdot \pi_2), (\pi_1 \cdot \pi_2)]$$

Depois de tudo isto, falta ainda provar a segunda condição:

$$[lh \cdot nil, lh \cdot cons] = [k \cdot i_1, (k \cdot i_2) \times \langle wc, lh \rangle]$$

<=> { Eq-+; Como já provamos, k será $k = [k1, k2]$ }

$$\begin{cases} lh \cdot nil = k1 \\ lh \cdot cons = k2 \cdot (id \times \langle wc, lh \rangle) \end{cases}$$

<=> { Pelo enunciado, $lh \cdot nil = true$, $lh \cdot cons = sep \cdot \pi_1$ }

$$\begin{cases} k1 = true \\ k2 \cdot (id \times \langle wc, lh \rangle) = sep \cdot \pi_1 \end{cases}$$

Para descobrir k2 é necessária a Lei de Leibniz, usada na seguinte prova:

$$k2 \cdot (id \times \langle wc, lh \rangle) = sep \cdot \pi_1$$

<=> { $Natural - \pi_1$; $Natural - id$ }

$$k2 \cdot (id \times \langle wc, lh \rangle) = sep \cdot \pi_1 \cdot (id \times \langle wc, lh \rangle)$$

<=> { Lei de Leibniz }

$$k2 = sep \cdot \pi_1$$

Conclui-se assim que

$$k = [k1, k2]$$

$<=>$ $\{$ Definição de k1 e k2 $\}$

$$k = [\underline{True}, sep \cdot \pi_1]$$

Finalmente, segue-se a solução final deste problema e um exemplo (ou teste no terminal) de como o *worker/wrapper* funcionaria.

$$wc\_w\_final :: [Char] \rightarrow Int$$
$$wc\_w\_final = wrapper \cdot worker$$
$$wrapper = \pi_1$$
$$worker = cataList \; \langle [\underline{0}, h2], [\underline{True}, k2] \rangle$$
$$\textbf{where } h2 = (\widehat{(\wedge)} \cdot ((\neg \cdot sep) \times \pi_2)) \rightarrow (\mathsf{succ} \cdot \pi_1 \cdot \pi_2), (\pi_1 \cdot \pi_2)$$
$$k2 = sep \cdot \pi_1$$

Exemplo: *worker diana tania paulo* - (3,False) - *wrapper (3,False)* - 3

Nota: Teste *QuickCheck*.
Para este teste, foi necessário gerar uma *String* aleatória composta por caracteres de 'A' a 'Z', incluindo espaços, tabs e novas linhas. Deste modo, foram criadas as funções 'genSafeChar' e 'genSafeString' que, em conjunto com o *wrapper* para a String 'SafeString', geram as Strings necessárias ao teste da solução proposta.

$$genSafeChar :: Gen \; Char$$
$$genSafeChar = elements \; \$ \; [\texttt{'a'} \, .. \, \texttt{'z'}] +\!\!+ \texttt{"/n/t "}$$
$$genSafeString :: Gen \; String$$
$$genSafeString = listOf \; genSafeChar$$
$$\textbf{newtype } SafeString = SafeString \; \{unwrapSafeString :: String\}$$
$$\textbf{deriving } Show$$
$$prop\_wc = forAll \; genSafeString \; \$ \; \lambda str \rightarrow (wc\_w\_final \; str) \equiv (\mathsf{length} \; \$ words \; \$ \; str)$$

Problema 3

O problema 3 envolvia, em primeiro lugar, a construção de uma biblioteca para o tipo de dados *B-Tree*. Assim, com a ajuda da biblioteca da *Btree* comum, e com algum tempo e empenho, foram conseguidas as seguintes definições de *inB-Tree*, *outB-Tree*, catamorfismo de *B-Tree*, entre outros :

$$
\begin{array}{l}
inB\_tree\ (i_1\ ()) = Nil \\
inB\_tree\ (i_2\ (x, l)) = Block\ \{leftmost = x, block = l\} \\
outB\_tree\ Nil = i_1\ () \\
outB\_tree\ Block\ \{leftmost = x, block = l\} = i_2\ (x, l) \\
recB\_tree\ f = baseB\_tree\ id\ f \\
baseB\_tree\ g\ f = id + (f \times \mathsf{map}\ \ (g \times f))\ \ \text{\{-map porque é lista -\}} \\
cataB\_tree\ g = g \cdot (recB\_tree\ (cataB\_tree\ g)) \cdot outB\_tree \\
anaB\_tree\ g = inB\_tree \cdot (recB\_tree\ (anaB\_tree\ g)) \cdot g \\
hyloB\_tree\ f\ g = cataB\_tree\ f \cdot anaB\_tree\ g \\
\mathbf{instance}\ Functor\ \mathsf{B\text{-}tree} \\
\quad \mathbf{where}\ \mathsf{fmap}\ f = cataB\_tree\ (inB\_tree \cdot baseB\_tree\ f\ id)
\end{array}
$$

De seguida, era necessário definir a função *inorder*, adequada para este tipo de dados, como um catamorfismo. Através do seguinte diagrama, foi conseguido um raciocínio claro que permitiu chegar à solução, também apresentada a seguir:

$$
\begin{array}{l}
inordB\_tree = cataB\_tree\ inordB \\
inordB = [nil, join] \\
\quad \mathbf{where}\ join = \mathsf{conc} \cdot (id \times (concat \cdot (\mathsf{map}\ \ (cons))))
\end{array}
$$

$$
\begin{tikzcd}
B - Tree\ A \arrow[d, "(\!|inordB|\!)"'] & & 1 + (B - Tree\ A \times (A \times B - treeA)^{*}) \arrow[ll, "inB-tree"'] \arrow[d, "id+((\!|inordB|\!)\times\mathsf{map}\ \ (id\times(\!|inordB|\!)))"] \\
A^{*} & & 1 + A^{*} \times (A \times A^{*})^{*} \arrow[ll, "inordB"]
\end{tikzcd}
$$

Era também pedida a definição da função *largest Block* como um catamorfismo. Mais uma vez, através do auxílio de um diagrama, o problema foi resolvido, e ambos apresentam-se em baixo:

$$
\begin{array}{l}
largestBlock = cataB\_tree\ largestB \\
\quad \mathbf{where}\ largestB = [\underline{0}, \widehat{max} \cdot \langle \pi_1, maximum \cdot (cons \cdot \langle \mathsf{length}\ \ \cdot \pi_2, auxCata \cdot \pi_2 \rangle) \rangle] \\
\quad\quad auxCata = cataList\ [nil, cons \cdot (\pi_2 \times id)]
\end{array}
$$

$$
\begin{tikzcd}
B - Tree\ A \arrow[d, "(\!|largestB|\!)"'] & & 1 + (B - Tree\ A \times (A \times B - treeA)^{*}) \arrow[ll, "inB-tree"'] \arrow[d, "id+((\!|largestB|\!)\times\mathsf{map}\ \ (id\times(\!|largestB|\!)))"] \\
Int & & 1 + Int \times (A \times Int)^{*} \arrow[ll, "largestB"]
\end{tikzcd}
$$

Desta vez, era requerida a definição da função *mirror* como um anamorfismo. O anamorfismo foi conseguido através de várias funções auxiliares, resultando num código mais legível e de mais fácil compreensão. De seguida encontra-se a solução proposta, tal como um esquema que representa o raciocínio que o grupo teve.

$mirrorB\_tree = anaB\_tree\ ((id + (fim \cdot rever \cdot insere \cdot mir)) \cdot outB\_tree)$
  where $mir = id \times unzip$
    $insere = \langle \pi_1 \cdot \pi_2, cons \cdot \langle \pi_1, \pi_2 \cdot \pi_2 \rangle \rangle$
    $rever = \langle reverse \cdot \pi_1, reverse \cdot \pi_2 \rangle$
    $fim = \langle head \cdot \pi_2, \widehat{zip} \cdot \langle \pi_1, tail \cdot \pi_2 \rangle \rangle$

O raciocínio seguinte inclui a explicação passo a passo, usando as funções acima, que o grupo seguiu, a partir dos tipos de dados necessários para a resolução do exercício.

$$
\begin{array}{cl}
 & 1 + (\textsf{B-tree}\ a \times (a \times \textsf{B-tree}\ a)*) \\
= & \quad \{\ \text{mir}\ \} \\
 & 1 + (\textsf{B-tree}\ a \times (a * \times \textsf{B-tree}\ a*)) \\
= & \quad \{\ \text{insere}\ \} \\
 & 1 + (a * \times \textsf{B-tree}\ a*) \\
= & \quad \{\ \text{rever}\ \} \\
 & 1 + (\textsf{B-tree}\ a \times (a * \times \textsf{B-tree}\ a*)) \\
= & \quad \{\ \text{fim}\ \} \\
 & 1 + (\textsf{B-tree}\ a \times (a \times \textsf{B-tree}\ a)*)
\end{array}
$$

Quase no fim, seria necessário definir a função *quick sort* como um hilomorfismo. Isto foi conseguido através do desenho de um diagrama, e com as bibliotecas fornecidas pelos docentes. De seguida, encontra-se a solução a este problema, juntamente com o diagrama do hilomorfismo.

$lsplitB\_tree\ [\,] = i_1\ ()$
$lsplitB\_tree\ (h : t) = i_2\ (s, [(h, l)])$ **where** $(s, l) = part1\ (<h)\ t$
$part1 :: (a \to Bool) \to [a] \to ([a], [a])$
$part1\ p\ [\,] \qquad = ([\,], [\,])$
$part1\ p\ (h : t) \mid p\ h = \textbf{let}\ (s, l) = part1\ p\ t\ \textbf{in}\ (h : s, l)$
  $\mid otherwise = \textbf{let}\ (s, l) = part1\ p\ t\ \textbf{in}\ (s, h : l)$
$qSortB\_tree :: Ord\ a \Rightarrow [a] \to [a]$
$qSortB\_tree = hyloB\_tree\ inordB\ lsplitB\_tree$

Diagrama:

$$
\begin{tikzcd}
A* \arrow[r, "lsplitBGene"] \arrow[d, "[\!(lsplitBGene)\!]"'] & 1 + A * \times (A \times A^*)^* \arrow[d, "id+([\!(lsplitBGene)\!]\times \mathsf{map}\ (id\times[\!(lsplitBGene)\!]))"] \\
B \arrow[r, shift left, "outB"] \arrow[d, "(\!|inordBGene|\!)"'] & 1 + B \times (A \times B)^* \arrow[l, shift left, "inB"] \arrow[d, "id+((\!|inordBGene|\!)\times \mathsf{map}\ (id\times(\!|inordBGene|\!)))"] \\
A^* & 1 + A^* \times (A \times A^*)^* \arrow[l, "inordBGene"]
\end{tikzcd}
$$

Finalmente, era requerido que fosse definida a função *dotB-tree*, que permitia mostrar em Graphviz árvores *B-tree*. Para esta função, já presente nas bibliotecas da Unidade Curricular, foi apenas necessário adaptar ao caso concreto da *B-tree* deste problema 3. A dificuldade centrou-se na função auxiliar *cB-tree2exp*, para a qual foi desenhado um diagrama e feito, passo a passo, um raciocínio de tipos para conseguir chegar à definição final da função auxiliar. Segue-se o catamorfismo que ajudou ao raciocínio, juntamente com a explicação passo a passo,a solução definitiva, e uma imagem de exemplo com o respetivo código.

Catamorfismo de cB-tree2exp:

$$\begin{tikzcd} B \arrow[d, "(\!|g|\!)"'] & & 1 + B \times (A \times B)^* \arrow[ll, "inB-tree"'] \arrow[d, "id+((\!|g|\!)\times \mathsf{map}\ (id\times(\!|g|\!)))"] \\ E & & 1 + E \times (A \times E)^* \arrow[ll, "g"] \end{tikzcd}$$

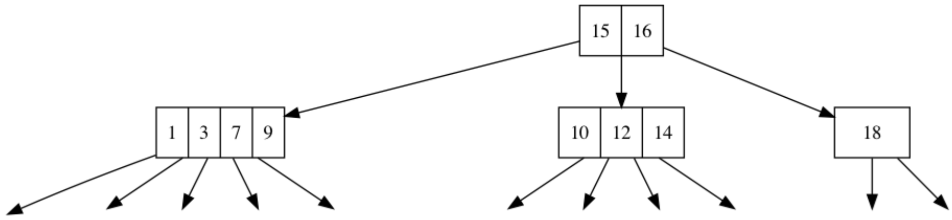
Explicação passo a passo do raciocínio até chegar a cB-tree2exp:

$$\begin{tikzcd} 1 + E \times (A \times E)^* \arrow[d, "id \times unzip"'] \\ E \times (A^* \times E^*) \arrow[d, "<\pi_1 \cdot \pi_2, <\pi_1, \pi_2 \cdot \pi_2 \gg"'] \\ A^* \times (E \times E^*) \arrow[d, "id \times cons"'] \\ A^* \times E^* \arrow[d, "(\widehat{Term})"'] \\ E \end{tikzcd}$$

Código:

$dotB\_tree :: (Show\ a) \Rightarrow \mathsf{B\text{-}tree}\ a \rightarrow \mathsf{IO}\ ExitCode$
$dotB\_tree = dotpict \cdot bmap\ nothing\ (Just \cdot init \cdot concat \cdot (\mathsf{map}\ (+\!\!+ \texttt{"|"})) \cdot (\mathsf{map}\ show)) \cdot cB\_tree2Exp$
$cB\_tree2Exp = cataB\_tree\ [\underline{(Var\ \texttt{"nil"})}, aux]$
  where $aux = (\widehat{Term}) \cdot (id \times cons) \cdot \langle \pi_1 \cdot \pi_2, \langle \pi_1, \pi_2 \cdot \pi_2 \rangle \rangle \cdot (id \times unzip)$

Imagem e código respetivo:



$bt = Block\ \{leftmost = Block\ \{leftmost = Nil, block = [(1, Nil), (3, Nil), (7, Nil), (9, Nil)]\},$
  $block = [(15, Block\ \{leftmost = Nil, block = [(10, Nil), (12, Nil), (14, Nil)]\}), (16, Block\ \{leftmost = Nil,$
  $block = [(18, Nil)]\})]\}$

Problema 4

Neste problema foram introduzidos conceitos de tipos de dados mutuamente recursivos, mais concretamente, os *L-Systems*. Era já fornecido o tipos de dados *Algae*, e os *in's* e *out's* respetivos, juntamente com os funtores e catamorfismos. Deste modo, era apenas necesário responder aos subproblemas propostos.

Como primeira instância, era necessária uma definição dos anamorfismos dos tipos A e B. Com o auxílio da definição de anamorfismo comum, presente nas bibliotecas da Unidade Curricular, foi apenas preciso adaptar a este caso concreto. Todo o raciocínio até à solução do primeiro problema é apresentado em baixo:

$$
\begin{array}{cl}
 & [\![g]\!] = \mathbf{in} \cdot (rec\ [\![g]\!]) \cdot g \\
<=> & \qquad \{ \text{ Adaptação do tipo geral anterior para o nosso } \} \\
 & \begin{cases} [\![g\ \cdot]\!]_A = inA \cdot (rec\ [\![g]\!]) \cdot g \\ [\![g\ \cdot]\!]_A = inA \cdot (rec\ [\![g]\!]) \cdot g \end{cases}
\end{array}
$$

Através dos catas já definidos (cataA e cataB), determinamos que o *rec* aplicado aos catas é, respetivamente:

$$
\begin{cases} id + (\!|\cdot\ \cdot|\!)_A \times (\!|\cdot\ \cdot|\!)_B \\ id + (\!|\cdot\ \cdot|\!)_A \end{cases}
$$

Também foi verificado que os anamorfismos teriam dois genes, espelhando os catamorfismos. Assim, e aplicando *rec* aos anamorfismos, temos que:

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

Como segunda instância foram requeridas duas definições: a definição da função *generateAlgae* como um anamorfismo de *Algae*, e a da função *showAlgae* com um catamorfismo de *Algae*. Ambas as funções foram conseguidas através do desenho de diagramas, que se apresentam a seguir, tal como as respetivas soluções.

Diagramas da função *generateAlgae*:

$$
\begin{tikzcd}
Algae\ A \arrow[r, "outA"] & 1 + A \times B \\
\mathbb{N}_0 \arrow[u, "generate\ A"] \arrow[r, "(id+\langle id,id\rangle)\cdot outNat"'] & 1 + \mathbb{N}_0 \times \mathbb{N}_0 \arrow[u, "id+generateA\ A\times generateA\ B"']
\end{tikzcd}
$$

Explicação do raciocínio para chegar a *genA*:

$$
\begin{tikzcd}
\mathbb{N}_0 \arrow[d, "outNat"'] \\
1 + \mathbb{N}_0 \arrow[d, "id+<id,id>"'] \\
1 + \mathbb{N}_0 \times \mathbb{N}_0
\end{tikzcd}
$$

$$
\begin{tikzcd}
B \arrow[r, "outB"] & 1 + A \\
\mathbb{N}_0 \arrow[u, "generateB"] \arrow[r, "outNat"'] & 1 + \mathbb{N}_0 \arrow[u, "id + generateB\ A"']
\end{tikzcd}
$$

Função *generateAlgae*:

$$
\begin{array}{l}
generateAlgae = [\![genA\ genB]\!]_A \\
\quad \textbf{where } genA = (id + \langle id, id \rangle) \cdot outNat \\
\qquad genB = outNat
\end{array}
$$

Diagramas da função *showAlgae*:

$$
\begin{tikzcd}
Algae\ A \arrow[d, "showA"'] & 1 + A \times B \arrow[l, "inA"'] \arrow[d, "id + ShowA\ A \times ShowA\ B"] \\
String & 1 + String \times String \arrow[l, "{[\underline{\texttt{"A"}}, \mathsf{conc}]}"]
\end{tikzcd}
$$

$$
\begin{tikzcd}
B \arrow[d, "showB"'] & 1 + A \arrow[l, "inB"'] \arrow[d, "id + ShowB\ A"] \\
String & 1 + String \arrow[l, "{[\underline{\texttt{"B"}}, id]}"]
\end{tikzcd}
$$

Função *showAlgae*:

$$
\begin{array}{l}
showAlgae = (\!| ginA\ ginB |\!)_A \\
\quad \textbf{where } ginA = [\underline{\texttt{"A"}}, \mathsf{conc}] \\
\qquad ginB = [\underline{\texttt{"B"}}, id]
\end{array}
$$

Nota: Teste *QuickCheck*.

Neste teste foi necessário usar as funções 'fromIntegral' e 'toInteger' para garantir a compatibilidade de tipos durante o teste. Também se chegou à conclusão de que seria melhor restringir o x entre 1 e 25 já que com valores mais elevados, os testes não seriam efetuados em tempo útil, ou seja, poderiam demorar dias a ser completados.

$$
\begin{array}{l}
prop\_sg\ x = (x > 1 \wedge x < 25) ==> (\mathsf{length}\ \cdot showAlgae \cdot generateAlgae)\ x \equiv \\
\quad (fromIntegral \cdot fib \cdot \mathsf{succ} \cdot toInteger)\ x
\end{array}
$$

Problema 5

O problema 5 tem como base probabilidades, e, consequentemente, o uso de mónades. Como primeira instância, era necessário definir a função *permuta* para construir a árvore de jogos a partir de uma permutação aleatória das equipas. A segunda parte do exercício envolvia a definição de outra função, *eliminatoria*, que dá como resultado a distribuição de equipas vencedoras do campeonato.

O raciocínio até chegar à solução da primeira função, - *permuta* - envolveu duas fases: inicialmente, a função foi definida em *pointwise*, e por fim foi convertida para "forma"de mónades, que é a solução final.

Definição *pointwise* de *permuta*:

$getNotMon :: [a] \rightarrow (a, [a])$
$getNotMon\ (a : b) = (a, b)$

$permuta1 :: [a] \rightarrow [a]$
$permuta1\ [\,] = [\,]$
$permuta1\ \ a = (c : b)$
$\quad \textbf{where}\ (c, d) = getNotMon\ (a)$
$\qquad b = permuta1\ \ (d)$

Solução final:

$permuta\ [\,] = return\ [\,]$
$permuta\ x = \textbf{do}\ \{(a, b) \leftarrow getR\ x; c \leftarrow permuta\ \ b; return\ (a : c)\}$

Para a segunda função - *eliminatoria* procedeu-se do mesmo modo. Assim, primeiro, esta foi definida em *pointwise*, e de seguida surgiu a solução final.

Definição *pointwise* de *eliminatoria*:

$jogoNotMon :: (a, a) \rightarrow a$
$jogoNotMon\ (a, b) = b$

$eliminatoria1 :: \mathsf{LTree}\ Equipa \rightarrow Equipa$
$eliminatoria1\ \ (Leaf\ \ a) = a$
$eliminatoria1\ \ (Fork\ \ (e, d)) = jogoNotMon\ (eliminatoria1\ \ e, eliminatoria1\ \ d)$

Solução final:

$eliminatoria\ (Leaf\ \ z) = return\ z$
$eliminatoria\ (Fork\ \ (a, b)) = \textbf{do}\ \{sortA \leftarrow eliminatoria\ \ a; sortB \leftarrow eliminatoria\ \ b; jogo\ \ (sortA, sortB)\}$

Índice