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

Departamento de Informática
Universidade do Minho

Junho de 2017

Grupo nr.	87
a76089	Etienne Costa
a78821	José Martins
a77049	Miguel Quaresma

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
import Nat
```

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

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

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

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

vai ser seleccionado pelo GHCi para ser executado.

3 Como realizar o trabalho

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

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

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

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

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

Problema 1

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

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

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

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

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

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

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

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].)

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

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:

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

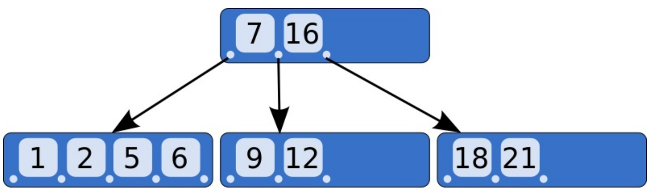
Re-implemente esta função segundo o modelo *worker/wrapper* onde *wrapper* deverá ser um catamorfismos de listas. Apresente os cálculos que fez para chegar a essa sua versão de $\mathit{wc\_w}$ e inclua testes em QuickCheck que verifiquem o funcionamento da sua solução. (**Sugestão:** aplique a lei de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

Problema 3

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

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

Por exemplo, a B-tree[3]



é representada no tipo acima por:

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

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

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 $\mathit{inordB\_tree} :: \mathsf{B\text{-}tree}\ t \to [t]$ que faça travessias "inorder"de árvores deste tipo.

3. Definir como um catamorfismo a função $\mathit{largestBlock} :: \mathsf{B\text{-}tree}\ a \to \mathit{Int}$ que detecta o tamanho do maior bloco da árvore argumento.

4. Definir como um anamorfismo a função $\mathit{mirrorB\_tree} :: \mathsf{B\text{-}tree}\ a \to \mathsf{B\text{-}tree}\ a$ 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:

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

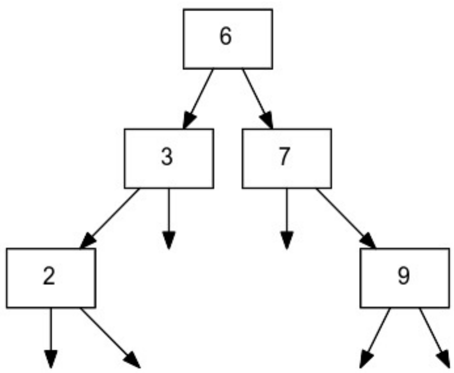
6. A biblioteca Exp permite representar árvores-expressão em formato DOT, que pode ser lido por aplicações como por exemplo Graphviz, produzindo as respectivas imagens. Por exemplo, para o caso de árvores BTree, se definirmos

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

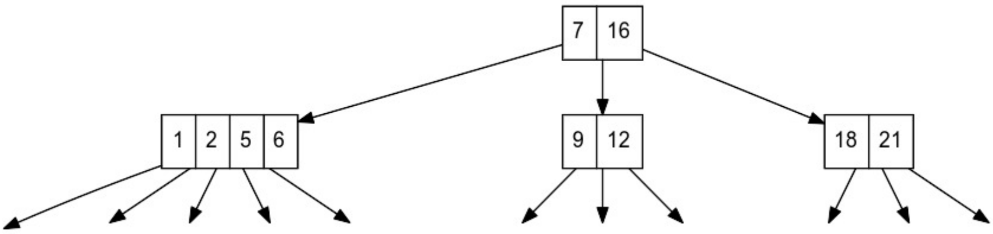
executando $\mathit{dotBTree}\ t$ para

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

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

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:

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

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

$$inA :: 1 + A \times B \rightarrow A$$
$$inA = [\underline{\mathrm{NA}}, \widehat{A}]$$
$$outA :: A \rightarrow 1 + A \times B$$
$$outA\ \mathrm{NA} = i_1\ ()$$
$$outA\ (A\ a\ b) = i_2\ (a, b)$$

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

$$
\begin{aligned}
&inB :: 1 + A \to B\\
&inB = [\underline{\mathrm{NB}}, B]\\
&outB :: B \to 1 + A\\
&outB\ \mathrm{NB} = i_1\ ()\\
&outB\ (B\ a) = i_2\ a
\end{aligned}
$$

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,

$$
\begin{aligned}
&\llparenthesis \cdot\ \cdot \rrparenthesis_A :: (1 + c \times d \to c) \to (1 + c \to d) \to A \to c\\
&\llparenthesis ga\ gb \rrparenthesis_A = ga \cdot (id + \llparenthesis ga\ gb \rrparenthesis_A \times \llparenthesis ga\ gb \rrparenthesis_B) \cdot outA
\end{aligned}
$$

e a mesma coisa para os Bs:

$$
\begin{aligned}
&\llparenthesis \cdot\ \cdot \rrparenthesis_B :: (1 + c \times d \to c) \to (1 + c \to d) \to B \to d\\
&\llparenthesis ga\ gb \rrparenthesis_B = gb \cdot (id + \llparenthesis ga\ gb \rrparenthesis_A) \cdot outB
\end{aligned}
$$

Pretende-se, neste problema:

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

   $$generateAlgae :: Int \to Algae$$

   como anamorfismo de $Algae$ e da função

   $$showAlgae :: Algae \to String$$

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

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

Problema 5

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

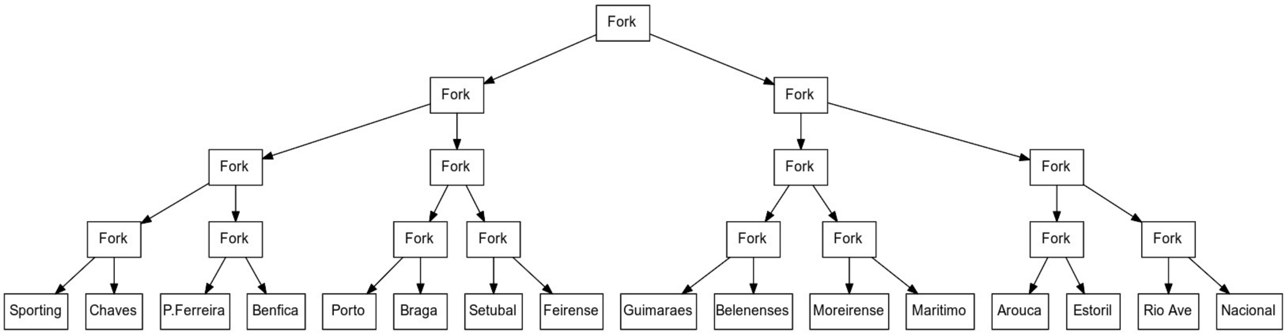
$equipas :: [Equipa]$
$equipas = [$
```
  "Arouca","Belenenses","Benfica","Braga","Chaves","Feirense",
  "Guimaraes","Maritimo","Moreirense","Nacional","P.Ferreira",
  "Porto","Rio Ave","Setubal","Sporting","Estoril"
  ]
```

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

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:

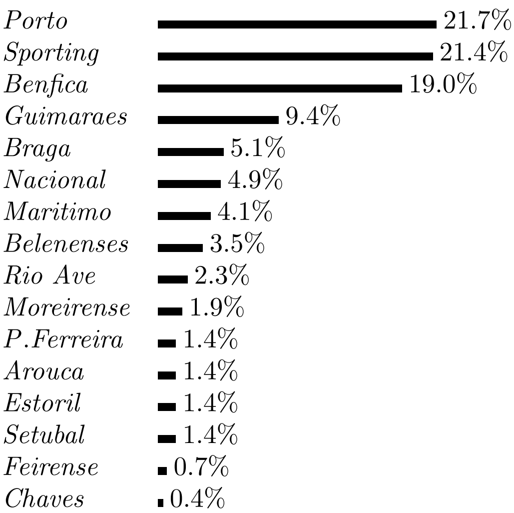
[6] Tratando-se de jogos eliminatórios, não há lugar a empates.



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 $[Equipa] \rightarrow \mathsf{Dist}\ Equipa$,

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Referências

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

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

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

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

Anexos

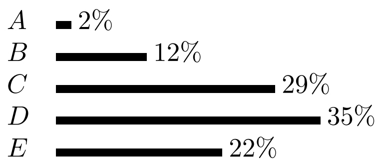
A Mónade para probabilidades e estatística

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

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

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 $g : A \to \mathsf{Dist}\ B$ e $f : B \to \mathsf{Dist}\ C$ 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,

```haskell
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*,

```haskell
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),

```haskell
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):

```haskell
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

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

-- quickCheck

-- property

$prop\_inv :: Double \rightarrow Int \rightarrow Bool$

$prop\_inv\ x\ n = abs\ (inv\ x\ n - 1\ /\ x) < 0.5$

-- test

$test\_inv = forAll\ (Test.QuickCheck.choose\ (1, 2))\ \$$

$\quad \lambda x \rightarrow forAll\ (Test.QuickCheck.choose\ (0, 1000))\ \$$

$\quad \lambda n \rightarrow prop\_inv\ x\ n$

Problema 2

$wc\_w\_final :: [\,Char\,] \rightarrow Int$

$wc\_w\_final = wrapper \cdot worker$

$wrapper = \pi_1$

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

$sep\ c = (c \equiv \texttt{'\ '} \vee c \equiv \texttt{'\textbackslash n'} \vee c \equiv \texttt{'\textbackslash t'})$

$prop\_wc :: [\,Char\,] \rightarrow Bool$

$prop\_wc\ s = (wc\_w\_final\ s) \equiv (wc\_w\ s)$

$test\_wc = \lambda s \rightarrow prop\_wc\ s$

$$\begin{cases} \begin{cases} wc\ [\,] = 0 \\ wc\ (c : l) = [\,wc\ l, [\mathsf{succ}\ \cdot wc\ l, wc\ l\,] \cdot (lookahead\ l)?\,] \cdot (sep\ c)? \end{cases} \\ \begin{cases} lookahead\ [\,] = True \\ lookahead\ (c : l) = sep\ c \end{cases} \end{cases}$$

<=> { Def-cond, Def-cons, Def-nil, Def-zero e Cancelamento-x }

$$\begin{cases} \begin{cases} wc \cdot nil = zero \\ wc \cdot cons\ (c, l) = [\,wc\ l, [\mathsf{succ}\ \cdot wc, wc\,] \cdot lookahead\ ?\ l\,] \cdot (sep\ c)? \end{cases} \\ \begin{cases} lookahead \cdot nil = \underline{True} \\ lookahead \cdot cons\ (c, l) = sep \cdot \pi_1\ (c, l) \end{cases} \end{cases}$$

<=> { Def-cond, Igualdade Extensional e Cancelamento-x }

$$\begin{cases} \begin{cases} wc \cdot nil = zero \\ wc \cdot cons\ (c, l) = [\,wc \cdot \pi_2, [\mathsf{succ}\ \cdot wc \cdot \pi_2, wc \cdot \pi_2\,] \cdot (lookahead \cdot \pi_2)?\,] \cdot (sep \cdot \pi_1)\ ?\ (c, l) \end{cases} \\ \begin{cases} lookahead \cdot nil = \underline{True} \\ lookahead \cdot cons = sep \cdot \pi_1 \end{cases} \end{cases}$$

<=> { Igualdade Extensional e Eq-+ }

$$\begin{cases} wc \cdot [\,nil, cons\,] = [\,zero, [\,wc \cdot \pi_2, [\mathsf{succ}\ \cdot wc \cdot \pi_2, wc \cdot \pi_2\,] \cdot (lookahead \cdot \pi_2)?\,] \cdot (sep \cdot \pi_1)?\,] \\ lookahead \cdot [\,nil, cons\,] = [\,\underline{True}, sep \cdot \pi_1\,] \end{cases}$$

<=> { Def-in }

$$\begin{cases} wc \cdot \mathbf{in} = [\,zero, [\,wc \cdot \pi_2, [\mathsf{succ}\ \cdot wc \cdot \pi_2, wc \cdot \pi_2\,] \cdot (lookahead \cdot \pi_2)?\,] \cdot (sep \cdot \pi_1)?\,] \\ lookahead \cdot \mathbf{in} = [\,\underline{True}, sep \cdot \pi_1\,] \end{cases}$$

<=> { Natural-p1, Natural-p2 e Cancelamento-x }

$$\begin{cases} wc \cdot \mathbf{in} = [\,zero, [\,wc \cdot \pi_2, [\mathsf{succ}\ \cdot \pi_1 \cdot \pi_2 \cdot (id \times <wc, look>), \pi_1 \cdot \pi_2 \cdot (id \times <wc, look>)\,] \cdot (\pi_2 \cdot \pi_2 \cdot (id \times <wc, look>))?\,] \cdot (sep \cdot \pi_1)?\,] \\ lookahead \cdot \mathbf{in} = [\,\underline{True}, \pi_1 \cdot (sep \times <wc, look>)\,] \end{cases}$$

<=> { Absorçao-+ e Natural-guarda; Natural-id e Functor-x }

$$\begin{cases} wc \cdot \mathbf{in} = [\,zero, [\,wc \cdot \pi_2, [\mathsf{succ}\ \cdot \pi_1 \cdot \pi_2, \pi_1 \cdot \pi_2\,] \cdot (\pi_2 \cdot \pi_2)\ ? \cdot (id \times <wc, look>)\,] \cdot (sep \cdot \pi_1)?\,] \\ lookahead \cdot \mathbf{in} = [\,\underline{True}, \pi_1 \cdot (sep \times id) \cdot (id \times <wc, look>)\,] \end{cases}$$

<=> { Natural-p2 e Cancelamento-x; Natural-const e Absorçao-+ }

$$\begin{cases} wc \cdot \mathbf{in} = [\,zero, [\,\pi_1 \cdot \pi_2 \cdot (id \times <wc, look>), [\mathsf{succ}\ \cdot \pi_1 \cdot \pi_2, \pi_1 \cdot \pi_2\,] \cdot (\pi_2 \cdot \pi_2)\ ? \cdot (id \times <wc, look>)\,] \cdot (sep \cdot \pi_1)?\,] \\ lookahead \cdot \mathbf{in} = [\,\underline{True}, \pi_1 \cdot (sep \times id)\,] \cdot (id + (id \times <wc, look>)) \end{cases}$$

<=> { Natural-p1; Functor-Listas, Natural-p1 }

$$\begin{cases} wc \cdot \mathbf{in} = [\,zero, [\,\pi_1 \cdot \pi_2 \cdot (id \times <wc, look>), [\mathsf{succ}\ \cdot \pi_1 \cdot \pi_2, \pi_1 \cdot \pi_2\,] \cdot (\pi_2 \cdot \pi_2)\ ? \cdot (id \times <wc, look>)\,] \cdot (\pi_1 \cdot (sep \times <wc, look>))?\,] \\ lookahead \cdot \mathbf{in} = [\,\underline{True}, sep \cdot \pi_1\,] \cdot F < wc, look> \end{cases}$$

<=> { Functor-x, Natural-id, Absorçao-+ e Natural-guarda }

$$\begin{cases} wc \cdot \mathbf{in} = [\,zero, [\,\pi_1 \cdot \pi_2, [\mathsf{succ}\ \cdot \pi_1 \cdot \pi_2, \pi_1 \cdot \pi_2\,] \cdot (\pi_2 \cdot \pi_2)?\,] \cdot (sep \cdot \pi_1)\ ? \cdot (id \times <wc, look>)\,] \\ lookahead \cdot \mathbf{in} = [\,\underline{True}, sep \cdot \pi_1\,] \cdot F < wc, look> \end{cases}$$

<=> { Natural-const e Absorçao-+ }

$$\begin{cases} wc \cdot \mathbf{in} = [\,zero, [\,\pi_1 \cdot \pi_2, [\mathsf{succ}\ \ \pi_1 \cdot \pi_2, \pi_1 \cdot \pi_2\,] \cdot (\pi_2 \cdot \pi_2)?\,] \cdot (sep \cdot \pi_1)?\,] \cdot (id + (id \times <wc, look>)) \\ lookahead \cdot \mathbf{in} = [\,\underline{True}, sep \cdot \pi_1\,] \cdot F < wc, look> \end{cases}$$

<=> { Functor-Listas }

$$\begin{cases} wc \cdot \mathbf{in} = [zero, [\pi_1 \cdot \pi_2, [\mathsf{succ} \cdot \pi_1 \cdot \pi_2, \pi_1 \cdot \pi_2] \cdot (\pi_2 \cdot \pi_2)?] \cdot (sep \cdot \pi_1)?] \cdot F < wc, look> \\ lookahead \cdot \mathbf{in} = [\underline{True}, sep \cdot \pi_1] \cdot F < wc, look> \end{cases}$$

<=> { Fookinga }

$$<wc, lookahead> = (\!|<[zero, [\pi_1 \cdot \pi_2, [\mathsf{succ} \cdot \pi_1 \cdot \pi_2, \pi_1 \cdot \pi_2] \cdot (\pi_2 \cdot \pi_2)?] \cdot (sep \cdot \pi_1)?], [\underline{True}, sep \cdot \pi_1]>|\!)$$

<= { Cancelamento-x e Leibniz }

$$wc = \pi_1 \cdot (\!|<[zero, [\pi_1 \cdot \pi_2, [\mathsf{succ} \cdot \pi_1 \cdot \pi_2, \pi_1 \cdot \pi_2] \cdot (\pi_2 \cdot \pi_2)?] \cdot (sep \cdot \pi_1)?], [\underline{True}, sep \cdot \pi_1]>|\!)$$

Problema 3

$inB\_tree = [\underline{Nil}, \widehat{Block}]$
$outB\_tree\ Nil = i_1\ ()$
$outB\_tree\ (Block\ arv\ ((x, y) : t)) = i_2\ (arv, (x, y) : t)$
$recB\_tree\ f = baseB\_tree\ id\ f$
$baseB\_tree\ g\ f = id + (f \times \mathsf{map}\ \ (g \times f))$
$cataB\_tree\ g = g \cdot (recB\_tree\ (cataB\_tree\ g)) \cdot outB\_tree$
$anaB\_tree\ g = inB\_tree \cdot (recB\_tree\ (anaB\_tree\ g)) \cdot g$
$hyloB\_tree\ f\ g = cataB\_tree\ f \cdot anaB\_tree\ g$
instance $Functor$ B-tree
  where fmap $f = cataB\_tree\ (inB\_tree \cdot baseB\_tree\ f\ id)$
$inordB\_tree = cataB\_tree\ [nil, \mathsf{conc} \cdot (id \times (concat \cdot (\mathsf{map}\ \ (\mathsf{conc} \cdot (singl \times id)))))]$
$largestBlock = cataB\_tree\ [\underline{0}, \widehat{max} \cdot (id \times (\widehat{max} \cdot (\mathsf{length}\ \times maximum) \cdot unzip))]$
$mirrorB\_tree = anaB\_tree\ (((\underline{()} + (f2l \cdot (id \times reverse))) \cdot outB\_tree)$
$right2last :: [(a, b)] \rightarrow [(a, b)]$
$right2last\ [\,] = [\,]$
$right2last\ [a] = [a]$
$right2last\ ((x1, x2) : ((y1, y2) : t)) = (x1, y2) : right2last\ ((y1, x2) : t)$
$f2l = \langle \pi_2 \cdot head \cdot \pi_2, right2last \cdot cons \cdot \langle \langle \pi_1 \cdot head \cdot \pi_2, \pi_1 \rangle, tail \cdot \pi_2 \rangle \rangle$
$lsplitB\_tree\ [\,] = i_1\ ()$
$lsplitB\_tree\ (h : t) = i_2\ (partitionL\ h\ t)$
$partitionL\ h\ [\,] = ([\,], [(h, [\,])])$
$partitionL\ h\ (x : xs) = (lt, [(lP, ltg), (gP, gts)])$
  where
    $lP = min\ h\ x$
    $gP = max\ h\ x$
    $lt = filter\ (\leqslant lP)\ xs$
    $gt = filter\ (>lP)\ xs$
    $ltg = filter\ (<gP)\ gt$
    $gts = filter\ (>gP)\ gt$
$qSortB\_tree :: Ord\ a \Rightarrow [a] \rightarrow [a]$
$qSortB\_tree = inordB\_tree \cdot (anaB\_tree\ lsplitB\_tree)$
$dotB\_tree :: Show\ a \Rightarrow$ B-tree $a \rightarrow$ IO $ExitCode$
$dotB\_tree = dotpict \cdot bmap\ nothing\ (Just \cdot concat \cdot (intersperse\ \texttt{"|"}) \cdot (\mathsf{fmap}\ show)) \cdot cB\_tree2Exp$
$cB\_tree2Exp = cataB\_tree\ [\underline{(Var\ \texttt{"Nil"})}, \widehat{Term} \cdot \langle (\mathsf{fmap}\ \pi_1) \cdot \pi_2, cons \cdot \langle \pi_1, (\mathsf{fmap}\ \pi_2) \cdot \pi_2 \rangle \rangle]$

Problema 4

$[\!(ga\ gb)\!]_A = inA \cdot (id + ([\!(ga\ gb)\!]_A \times [\!(ga\ gb)\!]_B)) \cdot ga$
$[\!(ga\ gb)\!]_B = inB \cdot (id + [\!(ga\ gb)\!]_A) \cdot gb$

$generateAlgae = [\!(geraA\ geraB)\!]_A$

$geraA = (\underline{()} + \langle id, id \rangle) \cdot outNat$
$geraB = (\underline{()} + id) \cdot outNat$
$showAlgae = (\![\underline{\texttt{"A"}}, \mathsf{conc}]\ [\underline{\texttt{"B"}}, id]\!)_A$
-- QuickCheck
-- property
$prop\_Algae\ n = (\mathsf{length}\ \cdot showAlgae \cdot generateAlgae)\ n \equiv (fromInteger \cdot fib \cdot toInteger \cdot \mathsf{succ}\ )\ n$
-- test
$test\_Algae = forAll\ (Test.QuickCheck.choose\ (0, 20))\ \$\ \lambda n \rightarrow prop\_Algae\ n$

Problema 5

$permuta\ [\,] = return\ [\,]$
$permuta\ l = \mathbf{do}\ \{(r1, l1) \leftarrow getR\ l; l2 \leftarrow permuta\ l1; return\ (r1 : l2)\}$
$eliminatoria = \pi_2 \cdot (cataLTree\ [\langle id, certainly \rangle, \langle winner \cdot \pi_1, updateScores \rangle \cdot \langle jogo \cdot (\pi_1 \times \pi_1), \pi_2 \times \pi_2 \rangle])$
$updateScores\ r = D\ (r1 +\!\!+\ r2)$
where
$r1 = \mathsf{map}\ \ (id \times (*(\pi_2\ x)))\ l1$
$r2 = \mathsf{map}\ \ (id \times (*(\pi_2\ y)))\ l2$
$(l1, l2) = (\langle unD \cdot \pi_1, unD \cdot \pi_2 \rangle \cdot \pi_2)\ r$
$(x : (y : t)) = unD\ (\pi_1\ r)$
$winner = \widehat{(!!)} \cdot (id \times (aux \cdot \widehat{elemIndex} \cdot \langle maximum, id \rangle)) \cdot unzip \cdot unD$
where
$aux\ Nothing = 0$
$aux\ (Just\ x) = x$

Índice