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

Departamento de Informática
Universidade do Minho

Junho de 2017

Grupo nr.	33
a71841	André Pinho
a70363	Hugo Gonçalves
a72362	Miguel Costa

Contents

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*.

$\mathbf{import}\ \mathit{Exp}$
$\mathbf{import}\ \mathsf{BTree}$
$\mathbf{import}\ \mathsf{LTree}$
$\mathbf{import}\ \mathit{St}$
$\mathbf{import}\ \mathit{Probability\ hiding}\ (\mathit{cond})$
$\mathbf{import}\ \mathit{Data.List}$
$\mathbf{import}\ \mathit{Test.QuickCheck\ hiding}\ ((\times))$
$\mathbf{import}\ \mathit{System.Random\ hiding}\ \langle\cdot,\cdot\rangle$
$\mathbf{import}\ \mathit{GHC}.\mathsf{IO}.\mathit{Exception}$
$\mathbf{import}\ \mathit{System}.\mathsf{IO}.\mathit{Unsafe}$

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

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

[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:

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

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 $lookahead\_sep$.)

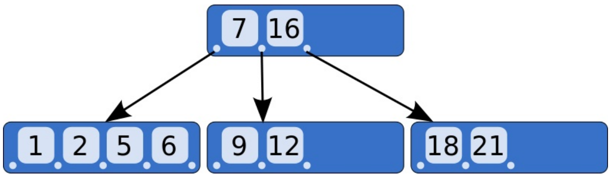
Problema 3

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:

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

Pretende-se, neste problema:

1. Construir uma biblioteca para o tipo B-tree da forma habitual (in + out; ana + cata + hylo; instância na classe $Functor$).
2. Definir como um catamorfismo a função $inordB\_tree$ :: B-tree $t \rightarrow [t]$ que faça travessias "inorder" de árvores deste tipo.
3. Definir como um catamorfismo a função $largestBlock$ :: B-tree $a \rightarrow Int$ que detecta o tamanho do maior bloco da árvore argumento.
4. Definir como um anamorfismo a função $mirrorB\_tree$ :: B-tree $a \rightarrow$ 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 $lsplitB\_tree$ cujo funcionamento se sugere a seguir:

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

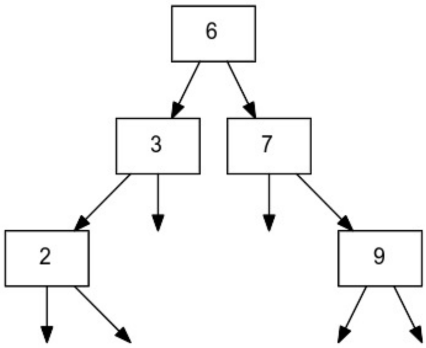
6. A biblioteca Exp permite representar árvores-expressão em formato DOT, que pode ser lido por aplicações como por exemplo Graphviz, produzindo as respectivas imagens. Por exemplo, para o caso de árvores BTree, se definirmos

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

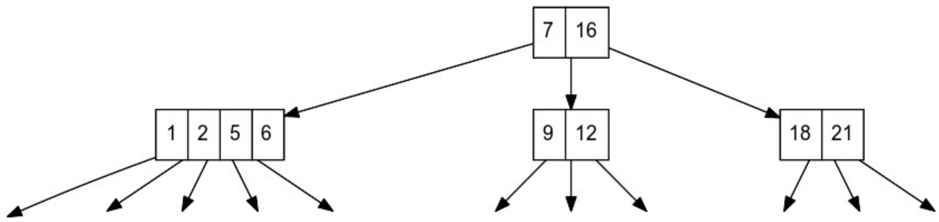
executando $dotBTree\ t$ para

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

obter-se-á a imagem



Escreva de forma semelhante uma função $dotB\_tree$ que permita mostrar em Graphviz[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: $A \to A\ B, B \to A$.

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:

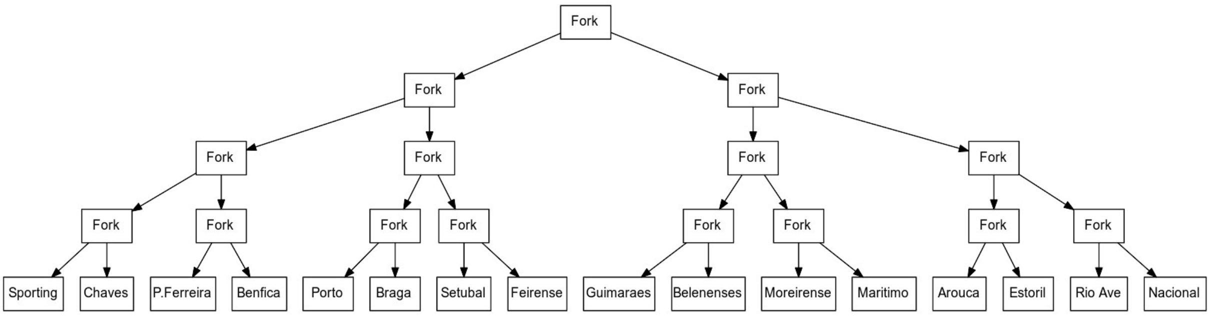
```
equipas :: [Equipa]
equipas = [
  "Arouca","Belenenses","Benfica","Braga","Chaves","Feirense",
  "Guimaraes","Maritimo","Moreirense","Nacional","P.Ferreira",
  "Porto","Rio Ave","Setubal","Sporting","Estoril"
  ]
```

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

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 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:

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

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

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

[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].

References

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

[2] B.W. Kernighan and D.M. 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

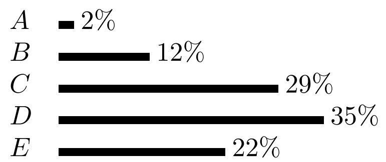
A Mónade para probabilidades e estatística

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

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

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.

$$d3 = normal\ [10\,..\,20]$$

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,

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

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

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

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

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

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

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

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

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

Problema 2

$$\begin{aligned}
&wc_w_final :: [\,Char\,] \rightarrow Int\\
&wc_w_final = wrapper \cdot worker\\
&wrapper = \pi_1\\
&worker = cataList\ \langle[\underline{0}, aux], [\underline{True}, sep \cdot \pi_1]\rangle\\
&\quad \textbf{where}\\
&\qquad sep = cond\ (\lambda c \rightarrow c \equiv \texttt{' '} \lor c \equiv \texttt{'\textbackslash n'} \lor c \equiv \texttt{'\textbackslash t'})\ true\ false\\
&\qquad aux = cond\ (\widehat{(\land)} \cdot \langle\neg \cdot sep \cdot \pi_1, \pi_2 \cdot \pi_2\rangle)\ (\mathsf{succ}\ \cdot \pi_1 \cdot \pi_2)\ (\pi_1 \cdot \pi_2)
\end{aligned}$$

Em primeiro lugar, começamos por definir as funções *wc_w* e *lookahead_sep* nas suas versões point free.

$wc\_w\ [\,] = 0$
$wc\_w\ (c : l) = \textbf{if}\ (\neg\ sep\ c) \land lookahead\_sep\ l\ \textbf{then}\ wc\_w\ l + 1\ \textbf{else}\ wc\_w\ l$

= { (74),(76),(79),(81), def succ, def nil, def cons }

$(wc\_w \cdot nil)\ x = \underline{0}\ x$
$(wc\_w \cdot cons)\ (c, l) = \textbf{if}\ (\neg\ sep \cdot \pi_1) \land (lookahead\_sep \cdot \pi_2)\ (c, l)\ \textbf{then}\ (\mathsf{succ} \cdot wc\_w \cdot \pi_2)\ (c, l)\ \textbf{else}\ (wc\_w \cdot \pi_2)\ (c, l)$

= { (76), def uncurry (and), (78), (80) }

$(wc\_w \cdot nil) = \underline{0}$
$(wc\_w \cdot cons)\ (c, l) = cond\ (\widehat{(\land)} \cdot \langle\neg \cdot sep \cdot \pi_1, lookahead\_sep \cdot \pi_2\rangle\ (\mathsf{succ}\ \cdot wc\_w \cdot \pi_2)\ (wc\_w \cdot \pi_2))\ (c, l)$

= { (73) }

$wc\_w \cdot nil = \underline{0}$
$wc\_w \cdot cons = cond\ (\widehat{(\land)} \cdot \langle\neg \cdot sep \cdot \pi_1, lookahead\_sep \cdot \pi_2\rangle\ (\mathsf{succ}\ \cdot wc\_w \cdot \pi_2)\ (wc\_w \cdot \pi_2))$

= { (1),(12),(7) }

$wc\_w \cdot nil = \underline{0}$
$wc\_w \cdot cons = cond\ (\widehat{(\land)} \cdot \langle\neg \cdot sep \cdot \pi_1 \cdot (id \times \langle wc\_w, lookahead\_sep\rangle), \pi_2 \cdot \langle wc\_w, lookahead\_sep\rangle \cdot \pi_2\rangle\ (\mathsf{succ} \cdot \pi_1 \cdot \langle wc\_w, lookahead\_sep\rangle \cdot \pi_2)\ (\pi_1 \cdot \langle wc\_w, lookahead\_sep\rangle \cdot \pi_2))$

= { (13) }

$wc\_w \cdot nil = \underline{0}$
$wc\_w \cdot cons = cond\ (\widehat{(\land)} \cdot \langle\neg \cdot sep \cdot \pi_1 \cdot (id \times \langle wc\_w, lookahead\_sep\rangle), \pi_2 \cdot \pi_2 \cdot (id \times \langle wc\_w, lookahead\_sep\rangle)\rangle\ (\mathsf{succ} \cdot \pi_1 \cdot \pi_2 \cdot (id \times \langle wc\_w, lookahead\_sep\rangle))\ (\pi_1 \cdot \pi_2 \cdot (id \times \langle wc\_w, lookahead\_sep\rangle)))$

= { (9) }

$wc\_w \cdot nil = \underline{0}$
$wc\_w \cdot cons = cond\ (\widehat{(\land)} \cdot (\langle\neg \cdot sep \cdot \pi_1, \pi_2 \cdot \pi_2\rangle \cdot (id \times \langle wc\_w, lookahead\_sep\rangle))\ (\mathsf{succ}\ \cdot \pi_1 \cdot \pi_2 \cdot (id \times \langle wc\_w, lookahead\_sep\rangle))\ (\pi_1 \cdot \pi_2 \cdot (id \times \langle wc\_w, lookahead\_sep\rangle)))$

= { (32) }

$wc\_w \cdot nil = \underline{0}$
$wc\_w \cdot cons = cond\ (\widehat{(\land)} \cdot (\langle\neg \cdot sep \cdot \pi_1, \pi_2 \cdot \pi_2\rangle\ (\mathsf{succ}\ \cdot \pi_1 \cdot \pi_2)\ (\pi_1 \cdot \pi_2) \cdot (id \times \langle wc\_w, lookahead\_sep\rangle)))$

$lookahead\_sep\ [\,] = True$
$lookahead\_sep\ (c : l) = sep\ c$

= { (81), def nil, def cons, (76) }

$lookahead\_sep \cdot nil = \underline{True}$
$(lookahead\_sep \cdot cons)\ (c, l) = (sep \cdot \pi_1)\ (c, l)$

$=\quad \{ \ (73) \ \}$

$lookahead\_sep \cdot nil = \underline{True}$
$lookahead\_sep \cdot cons = sep \cdot \pi_1$

Graças à definição de in de listas([nil,cons]),da lei 20 (Fusão +) e da lei 27 (Eq +), podemos definir cada uma das funcoes da seguinte forma:

$wc\_w \cdot \mathbf{in} = [\underline{0}, cond\ (\widehat{(\wedge)} \cdot (\langle \neg \cdot sep \cdot \pi_1, \pi_2 \cdot \pi_2 \rangle\ (\mathsf{succ}\ \cdot \pi_1 \cdot \pi_2)\ (\pi_1 \cdot \pi_2) \cdot (id \times \langle wc\_w, lookahead\_sep \rangle))))]$
$lookahaed\_sep \cdot \mathbf{in} = [\underline{True}, sep \cdot \pi_1]$

Por fim, como o Functor de listas é do tipo 1 + X, aplicamos a lei 50 (Fokkinga):

$wc\_w \cdot \mathbf{in} = [\underline{0}, cond\ (\widehat{(\wedge)} \cdot (\langle \neg \cdot sep \cdot \pi_1, \pi_2 \cdot \pi_2 \rangle\ (\mathsf{succ}\ \cdot \pi_1 \cdot \pi_2)\ (\pi_1 \cdot \pi_2) \cdot (id \times \langle wc\_w, lookahead\_sep \rangle))))] \cdot (id \times \langle wc\_w, lookahead\_sep \rangle)$
$lookahaed\_sep \cdot \mathbf{in} = [\underline{True}, sep \cdot \pi_1] \cdot (id \times \langle wc\_w, lookahead\_sep \rangle)$

$=\quad \{ \ (50) \ \}$

$\langle wc\_w, lookahead\_sep \rangle = cataList\ \langle [\underline{0}, cond\ (\widehat{(\wedge)} \cdot \langle \neg \cdot sep \cdot \pi_1, \pi_2 \cdot \pi_2 \rangle\ (\mathsf{succ}\ \cdot \pi_1 \cdot \pi_2)\ (\pi_1 \cdot \pi_2))], [\underline{True}, sep \cdot \pi_1] \rangle$

Desta forma chegamos à definição do worker, um catamorfismo de listas. Como apenas nos interessa o primeiro elemento do par, a função wrapper seleciona apenas o primeiro elemento, ficando assim com a definição de wc w final.

Problema 3

$inB\_tree :: () + (\mathsf{B\text{-}tree}\ a, [(a, \mathsf{B\text{-}tree}\ a)]) \rightarrow \mathsf{B\text{-}tree}\ a$
$inB\_tree = [\underline{Nil}, \widehat{Block}]$
$outB\_tree\ Nil = i_1\ ()$
$outB\_tree\ (Block\ \{ leftmost = l, block = b \}) = i_2\ (l, b)$
$recB\_tree\ f = baseB\_tree\ id\ f$
$baseB\_tree\ g\ f = id + f \times \mathsf{map}\ \ (g \times f)$
$cataB\_tree\ g = g \cdot recB\_tree\ (cataB\_tree\ g) \cdot outB\_tree$
$anaB\_tree\ g = inB\_tree \cdot recB\_tree\ (anaB\_tree\ g) \cdot g$
$hyloB\_tree\ f\ g = cataB\_tree\ f \cdot anaB\_tree\ g$
$\mathbf{instance}\ Functor\ \mathsf{B\text{-}tree}$
$\quad \mathbf{where}\ \mathsf{fmap}\ f = cataB\_tree\ (inB\_tree \cdot baseB\_tree\ f\ id)$
$inordB\_tree = cataB\_tree\ [nil, \mathsf{conc} \cdot (id \times (concat \cdot \mathsf{map}\ \ cons))]$
$largestBlock = cataB\_tree\ ([\underline{0}, \widehat{max}] \cdot (id + id \times f))$
$\quad \mathbf{where}\ f = \widehat{max} \cdot \langle \mathsf{length}\ , maximum \rangle \cdot (\mathsf{map}\ \ \pi_2)$
$mirrorB\_tree = anaB\_tree\ ((id + ((g \cdot f) \cdot (id \times (unzip \cdot reverse)))) \cdot outB\_tree)$
$\quad \mathbf{where}$
$\quad\quad f\ (a, ([\,], [\,])) = (a, ([\,], [\,]))$
$\quad\quad f\ (a, (l, (h : t))) = (h, (l, t +\!\!+ [a]))$
$\quad\quad g = (id \times \widehat{zip})$
$lsplitB\_tree\ [\,] = i_1\ ()$

$lsplitB\_tree\ [h] = i_2\ ([\,], [(h, [\,])])$
$lsplitB\_tree\ (h1 : h2 : t) = i_2\ ((l, [(a, tMin), (b, tMax)]))$
where
$(a, b) = \langle \widehat{min}, \widehat{max} \rangle\ (h1, h2)$
$(tMin, tMax) = \langle filter\ (\widehat{(\wedge)} \cdot \langle >a, <b \rangle), filter\ (>b) \rangle\ t$
$l = filter\ (<a)\ t$

$qSortB\_tree :: Ord\ t \Rightarrow [t] \rightarrow [t]$
$qSortB\_tree = inordB\_tree \cdot anaB\_tree\ lsplitB\_tree$

$dotB\_tree :: Show\ a \Rightarrow \textsf{B-tree}\ a \rightarrow \textsf{IO}\ ExitCode$
$dotB\_tree = dotpict \cdot bmap\ nothing\ (Just \cdot show) \cdot cB\_tree2Exp$

$cB\_tree2Exp = cataB\_tree\ [\underline{(Var\ \texttt{"nil"})}, f \cdot (id \times unzip)]$
where
$f = \widehat{Term} \cdot (((id \times cons) \cdot assocr \cdot (swap \times id) \cdot assocl))$
-- f (a,(b,c)) = Term b (a:c)

Problema 4

{-Tendo em conta as definições dos catamorfismos dadas, definir os anamorfismos correspondentes é uma ta

$recA\ g\ h = baseA\ id\ g\ h$
$baseA\ f\ g\ h = f + g \times h$
$[\![ga\ gb]\!]_A = inA \cdot (recA\ [\![ga\ gb]\!]_A\ [\![ga\ gb]\!]_B) \cdot ga$
$recB\ f = baseB\ id\ f$
$baseB\ g\ f = g + f$
$[\![ga\ gb]\!]_B = inB \cdot (recB\ [\![ga\ gb]\!]_A) \cdot gb$

$generateAlgae = \bot$
$showAlgae = (\!|show\ show|\!)_A$

Problema 5

$permuta = \bot$
$eliminatoria = \bot$

Index