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

Departamento de Informática
Universidade do Minho

Junho de 2017

Grupo nr.	99 (preencher)
a78582	Hugo Brandão
a78296	Sérgio Alves
a78218	Tiago Alves

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:

$$\{\text{-\# OPTIONS}_{GHC} - XNPlusKPatterns\#-\}$$
$$\textbf{import } Cp$$

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

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

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

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

vai ser seleccionado pelo GHCi para ser executado.

3 Como realizar o trabalho

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

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

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

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

e recompilar o texto como acima se indicou. Dever-se-á ainda instalar o utilitário QuickCheck [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:

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

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 $lookahead\_sep$.)

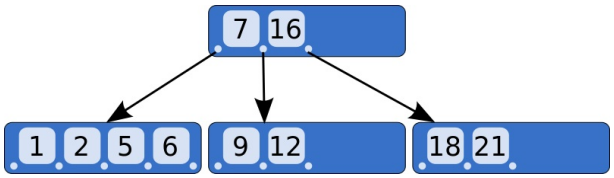
Problema 3

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

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

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:

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

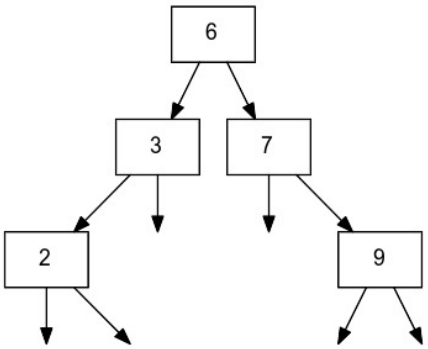
6. A biblioteca Exp permite representar árvores-expressão em formato DOT, que pode ser lido por aplicações como por exemplo Graphviz, produzindo as respectivas imagens. Por exemplo, para o caso de árvores BTree, se definirmos

```
-- dotBTree :: Show a =¿ BTree a -¿ IO ExitCode
-- dotBTree = dotpict . bmap Nothing (Just . show) . cBTree2Exp
```

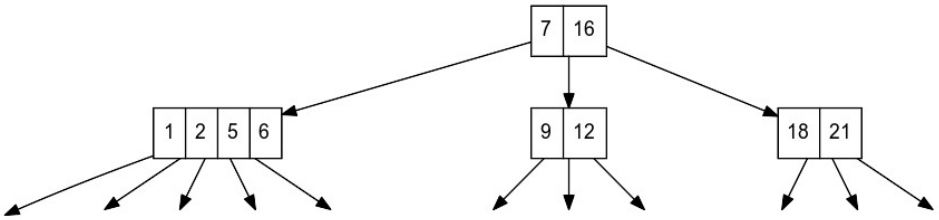
executando $dotBTree\ t$ para

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

obter-se-á a imagem



Escreva de forma semelhante uma função $dotB\_tree$ que permita mostrar em Graphviz[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:

$$\begin{aligned}&\textbf{type } Algae = A\\&\textbf{data } A = \text{NA} \mid A\ A\ B \textbf{ deriving } Show\\&\textbf{data } B = \text{NB} \mid B\ A \textbf{ deriving } Show\end{aligned}$$

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

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

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,

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

e a mesma coisa para os Bs:

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

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:

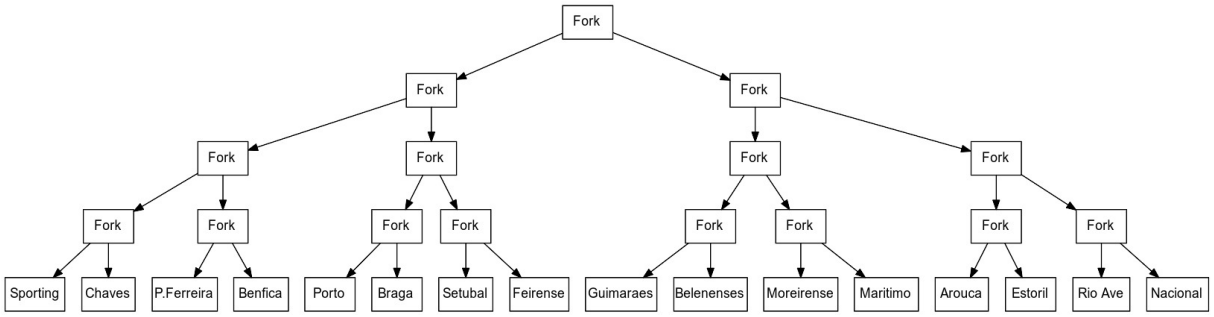
```
equipas :: [Equipa]
equipas = [
  "Arouca","Belenenses","Benfica","Braga","Chaves","Feirense",
  "Guimaraes","Maritimo","Moreirense","Nacional","P.Ferreira",
  "Porto","Rio Ave","Setubal","Sporting","Estoril"
  ]
```

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

$$\begin{array}{l} Arouca \\ \blacksquare\blacksquare\blacksquare\blacksquare\blacksquare\blacksquare\blacksquare\blacksquare\blacksquare\blacksquare\ 71.4\% \end{array} \qquad \begin{array}{l} \blacksquare\blacksquare\blacksquare\blacksquare\ 28.6\% Braga \end{array}$$

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

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

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:

$$\begin{array}{l} Porto \\ \blacksquare\ 21.4\% Benfica \\ \blacksquare\blacksquare\blacksquare\blacksquare\blacksquare\blacksquare\blacksquare\blacksquare\blacksquare\ 9.4\% Braga \\ \blacksquare\blacksquare\blacksquare\blacksquare\blacksquare\ 4.9\% Maritimo \\ \blacksquare\blacksquare\blacksquare\blacksquare\ 3.5\% Rio\ Ave \\ \blacksquare\blacksquare\blacksquare\ 1.9\% P.Ferreira \\ \blacksquare\blacksquare\blacksquare\ 1.4\% Estoril \\ \blacksquare\blacksquare\blacksquare\ 1.4\% Feirense \\ \blacksquare\ 0.4\% \end{array} \qquad \begin{array}{l} \blacksquare\ 21.7\% Sporting \\ \blacksquare\blacksquare\blacksquare\blacksquare\blacksquare\blacksquare\blacksquare\blacksquare\blacksquare\blacksquare\blacksquare\blacksquare\blacksquare\blacksquare\blacksquare\blacksquare\blacksquare\blacksquare\blacksquare\ 19.0\% Guimaraes \\ \blacksquare\blacksquare\blacksquare\blacksquare\blacksquare\ 5.1\% Nacional \\ \blacksquare\blacksquare\blacksquare\blacksquare\ 4.1\% Belenenses \\ \blacksquare\blacksquare\ 2.3\% Moreirense \\ \blacksquare\ 1.4\% Arouca \\ \blacksquare\ 1.4\% Setubal \\ \blacksquare\ 0.7\% Chaves \end{array}$$

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

$$\begin{aligned} &quem\_vence :: [Equipa] \to \mathsf{Dist}\ Equipa \\ &quem\_vence = eliminatoria \cdot sorteio \end{aligned}$$

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

O anamorfismo $sorteio :: [Equipa] \to \mathsf{LTree}\ Equipa$ 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

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

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

[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.

[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.

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

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

   $getR\ x$ 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

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

   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

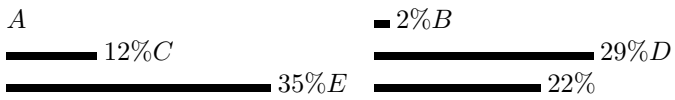
A Mónade para probabilidades e estatística

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

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

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

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



será representada pela distribuição

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

que o GHCi mostrará assim:

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

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

```
d2 = uniform (words "Uma frase de cinco palavras")
```

isto é

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

distribuição *normais*, eg.

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

etc.[10]

Dist forma um **mónade** cuja unidade é $return\ a = D\ [(a, 1)]$ e cuja composição de Kleisli é (simplificando a notação)

$$(f \bullet g)\ a = [(y, q * p)\ |\ (x, p) \leftarrow g\ a, (y, q) \leftarrow f\ x]$$

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

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

[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,

```
type Equipa = String
jogo :: (Equipa, Equipa) → Dist Equipa
jogo (e1, e2) = D [(e1, 1 − r1 / (r1 + r2)), (e2, 1 − r2 / (r1 + r2))] where
  r1 = rank e1
  r2 = rank e2
  rank = pap ranks
  ranks = [
    ("Arouca", 5),
    ("Belenenses", 3),
    ("Benfica", 1),
    ("Braga", 2),
    ("Chaves", 5),
    ("Feirense", 5),
    ("Guimaraes", 2),
    ("Maritimo", 3),
    ("Moreirense", 4),
    ("Nacional", 3),
    ("P.Ferreira", 3),
    ("Porto", 1),
    ("Rio Ave", 4),
    ("Setubal", 4),
    ("Sporting", 1),
    ("Estoril", 5)]
```

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

```
getR :: [a] → IO (a, [a])
getR x = do {
  i ← getStdRandom (randomR (0, length x − 1));
  return (x !! i, retira i x)
  } where retira i x = take i x ++ drop (i + 1) x
```

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

```
presort :: (Ord a, Ord b) ⇒ (b → a) → [b] → [b]
presort f = map π2 · sort · (map (fork f id))
```

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

```
pap :: Eq a ⇒ [(a, t)] → a → t
pap m k = unJust (lookup k m) where unJust (Just a) = a
```

C Soluções propostas

Os alunos devem colocar neste anexo as suas soluções aos exercícios propostos, de acordo com o "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

Neste problema a maior dificuldade foi descobrir como em pointfree conseguíamos aplicar a fórmula matemática de maclaurin dado um natural arbitrário. A primeira solução a que chegamos foi uma que se baseia numa equação matemática demasiado complicada para apresentar uma justificação, e por indicação do docente tentamos encontrar outra via.

$$inv1\ x = \mathsf{for}\ ((1+)\cdot((1-x)*))\ 1$$

O ponto de partida foi então uma definição pointwise de inv:

$$\begin{aligned}&invpw\ x\ 0 = 1\\&invpw\ x\ (n+1) = maclaurin\ n + invpw\ x\ n\\&\quad \mathbf{where}\ maclaurin\ 0 = 1\\&\qquad maclaurin\ (n+1) = (1-x) * maclaurin\ n\end{aligned}$$

A partir de aí, com o auxílio da Lei de *Fokkinga*, tentamos chegar ao catamorfismo de *inv x*:

$$f\cdot \mathbf{in} = h\cdot F\ \langle f, g\rangle\ e\ g\cdot \mathbf{in} = k\cdot F\ \langle f, g\rangle$$

$$\begin{aligned}=\quad & \{\ \text{in=(either (const 0) succ); f = inv; g = maclaurin; F(split f g) = F(split inv maclaurin)=(id + (split inv maclaurin}\\ & inv\cdot[\underline{0}, \mathsf{succ}\,] = h\cdot(id + \langle inv, maclaurin\rangle) maclaurin\cdot[\underline{0}, \mathsf{succ}\,] = k\cdot(id + \langle inv, maclaurin\rangle)\end{aligned}$$

Deduzimos então as funções h e k de inv e de maclaurin para preencher a lei de *Fokkinga*:

$$\begin{aligned}& inv2v\ x\ \underline{0} = 1 inv2v\ x\ \mathsf{succ} = add\cdot(\langle maclaurin, \cdot\rangle \times inv2v\ x)\\=\quad & \{\ \text{Universal-+}\ \}\\& inv2v\ x\ [\underline{0}, \mathsf{succ}\,] = [\underline{1}, add\cdot\langle maclaurin\ x, inv2v\ x\rangle]\\=\quad & \{\ \text{Natural-id; Definição de maclaurin x}\ \}\\& inv2v\ x\ [\underline{0}, \mathsf{succ}\,] = [\underline{1}\cdot id, add\cdot\langle((1-x)*)\ maclaurin\ x\ n, inv2v\ x\rangle]\\=\quad & \{\ \text{Absorção-x}\ \}\\& inv2v\ x\ [\underline{0}, \mathsf{succ}\,] = [\underline{1}\cdot id, add\cdot(((1-x)*)\times id)\cdot\langle maclaurin\ x, inv2v\ x\rangle]\\=\quad & \{\ \text{Absorção-+}\ \}\\& inv2v\ x\ [\underline{0}, \mathsf{succ}\,] = [\underline{1}, add\cdot(((1-x)*)\times id)]\cdot(id + \langle maclaurin\ x, inv2v\ x\rangle)\end{aligned}$$

Logo,

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

Obtendo então h, passamos à dedução de k:

$$\begin{aligned}& maclaurin\ x\ \underline{0} = 1 maclaurin\ x\ \mathsf{succ} = (1-x) * (maclaurin\ x)\\=\quad & \{\ \text{Universal-+}\ \}\\& maclaurin\ x\ [\underline{0}, \mathsf{succ}\,] = [\underline{1}, (1-x) * (maclaurin\ x)]\\=\quad & \{\ \text{Natural-id; Cancelamento-x}\ \}\\& inv2v\ x\ [\underline{0}, \mathsf{succ}\,] = [\underline{1}\cdot id, ((1-x)*)\cdot\pi_1\cdot\langle maclaurin\ x, inv2v\ x\rangle]\\=\quad & \{\ \text{Absorção-+}\ \}\\& inv2v\ x\ [\underline{0}, \mathsf{succ}\,] = [\underline{1}, ((1-x)*)\cdot\pi_1]\cdot(id + \langle maclaurin\ x, inv2v\ x\rangle)\end{aligned}$$

Assim,

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

Então, pela Lei de *Fokkinga*, temos que:

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

$=$ { Fokkinga }

$\langle inv, maclaurin\rangle = cata\ \langle h, k\rangle$

Com k encontrado e h definido acima, o catamorfismo de *inv x* apresenta-se:

$$invc\ x = \pi_2 \cdot cataNat\ \langle [\underline{(1)}, ((1-x)*) \cdot \pi_1], [\underline{(1)}, add \cdot (((1-x)*) \times id)]\rangle$$

Finalmente, a última tarefa consistia em provar que *inv x* seria um ciclo-*for*:

$\langle h, k\rangle$

$=$ { Definição de h e k }

$\langle [\underline{1}, ((1-x)*) \times id], [\underline{1}, ((1-x)*) \cdot \pi_1]\rangle$

$=$ { Lei da Troca }

$[\langle \underline{1}, \underline{1}\rangle, \langle ((1-x)*) \times id, ((1-x)*) \cdot \pi_1\rangle]$

Finalmente, obtemos a solução, em que *add* foi substituido por *uncurry (+)* por erros de tipos:

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

Nota: Teste *QuickCheck*:

```
prop_invfokk :: Double → Property
prop_invfokk num = (num > 1 ∧ num < 2) ==> abs (a − b) ⩽ 0.000000000000009
  where a = inv num 500000
    b = 1 / num
```

Problema 2

```
wc_w_final :: [Char] → Int
wc_w_final = wrapper · worker
wrapper = π2
worker = cataList [⟨True, 0⟩, ⟨sep · π1, cond ((∧)) · ((¬ · sep) × π1)) (succ · π2 · π2) (π2 · π2)⟩]
  where sep c = (c ≡ ' ' ∨ c ≡ '\n' ∨ c ≡ '\t')
```

Os cálculos do problema 2 são os seguintes:

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

$=$ { Aplicar o [nil,cons] e Lei 73 }

$lookahead\_sep\ nil = True lookahead\_sep\ cons\ (c, l) = sep\ (c, l)$

$=$ { Definição de in }

$look \cdot \mathbf{in} = [\underline{True}, sep \cdot \pi_1]$

id entre sep e p1 para podermos usar a lei 12

$=$ { Lei 1- Natural-id }

$look \cdot \mathbf{in} = [\underline{True} \cdot id, sep \cdot id \cdot \pi_1]$

Como podemos colocar uma função g, utilizamos um split para termos já o Funtor definido para usarmos Fokkinga

$$p1.p2) -> succ.p2.p2, p2.p2).(c, (lookahead_s epl, wc_w l)) \;\%\; \lambda just = \{\, Lei\ 73 - Igualdade\ extensional \,\} \;\%\; wc_w ni$$
$$).(not.sep >< p1)) -> succ.p2.p2, p2.p2).(id >><< lookahead_s ep, wc_w >) \;\%\; \lambda just = \{\, Defini\c{c}ao\ de\ \mathbf{in} \,\} \;\%\; wc_w.i$$
$$).(not.sep >< p1)) -> succ.p2.p2, p2.p2)](id - -(id \times <look$$

$$).(not.sep >< p1)) -> succ.p2.p2, p2.p2)](id - -(id \times <look$$

```
prop_wc_w_final :: String -> Property
prop_wc_w_final str = forAll genSafeString $ λstr -> (toInteger (wc_w str) ≡ toInteger (wc_w_final str))
genSafeChar :: Gen Char
genSafeChar = elements (['a'..'z'] ++ [' ','\t','\n'])
genSafeString :: Gen String
genSafeString = listOf genSafeChar
```

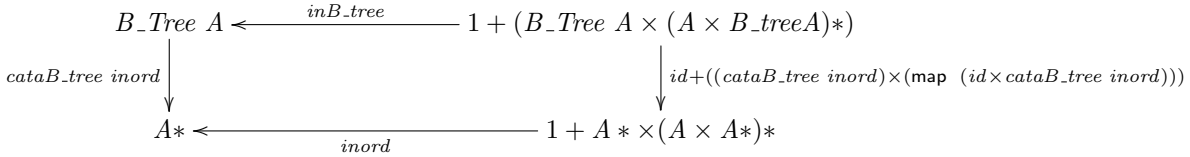
Problema 3

```
inB_tree :: () + (B-tree a, [(a, B-tree a)]) -> B-tree a
inB_tree = [Nil, Block]
outB_tree :: B-tree a -> () + (B-tree a, [(a, B-tree a)])
outB_tree (Nil) = i1 ()
outB_tree (Block {leftmost = l, block = b}) = i2 (l, b)
recB_tree f = baseB_tree id f
baseB_tree g f = id + (f × (map (g × f)))
cataB_tree g = g · (recB_tree (cataB_tree g)) · outB_tree
anaB_tree g = inB_tree · (recB_tree (anaB_tree g)) · g
hyloB_tree f g = cataB_tree f · anaB_tree g
instance Functor B-tree
  where fmap f = cataB_tree (inB_tree · baseB_tree f id)
```

Para as funções mais complexas apresentamos o diagrama correspondente que nos ajudou a resolver os problemas propostos.

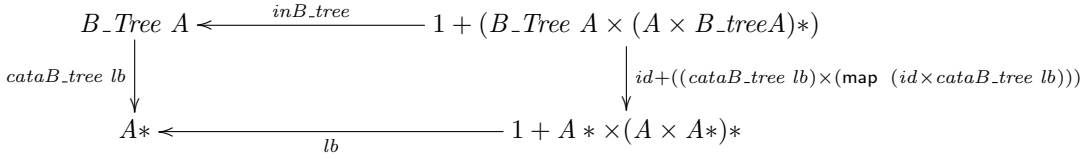
inordB_tree:

```
inordB_tree = cataB_tree inord
  where inord = [nil, conc · (id × aux)]
    aux = cataList [nil, conc · ((singl · π1) × id)]
```



largestBlock:

```
largestBlock :: B-tree a → Int
largestBlock = cataB_tree lb
  where lb = [0, max^ · ⟨length · π2, max^ · (id × (maximum · aux))⟩]
        aux = cataList [nil, cons · (π2 × id)]
```



mirrorB_tree

```
mirrorB_tree = anaB_tree ((id + (join · inside · auxb2)) · outB_tree)
  where auxb2 = id × unzip
        inside = ⟨reverse · π1 · π2, reverse · cons · ⟨π1, π2 · π2⟩⟩
        join = ⟨head · π2, zip^ · ⟨π1, tail · π2⟩⟩


lsplitB_tree [] = i1 ()
lsplitB_tree (h : t) = i2 (s, [(h, l)]) where (s, l) = partB_tree (<h) t
partB_tree :: (a → Bool) → [a] → ([a], [a])
partB_tree p []                = ([], [])
partB_tree p (h : t) | p h = let (s, l) = partB_tree p t in (h : s, l)
   | otherwise = let (s, l) = partB_tree p t in (s, h : l)
qSortB_tree :: Ord a ⇒ [a] → [a]
qSortB_tree = hyloB_tree [nil, inord] lsplitB_tree
  where inord = conc · (id × (concat · (map cons)))
dotB_tree :: (Show a) ⇒ B-tree a → IO ExitCode
dotB_tree = dotpict · bmap nothing (Just · init · concat · (map (++" | ")) · (map show)) · cB_tree2Exp
cB_tree2Exp = cataB_tree [(Var "nil"), aux]
  where aux = (Term^) · (id × cons) · ⟨π1 · π2, ⟨π1, π2 · π2⟩⟩ · (id × unzip)
```

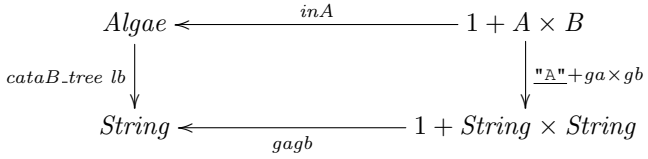
Problema 4

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

$$generateAlgae = [\![(id + \langle id, id\rangle) \cdot outNat\ outNat]\!]_A$$

generateAlgae:

```
showAlgae = (|ga gb|)_A
  where ga = ["A", conc]
        gb = ["B", id]
```



```
prop_4_6 :: Integer → Property
prop_4_6 n = (n ⩾ 0 ∧ n < 24) ==> lshG n ≡ fibsucc n
  where lshG = toInteger · length · showAlgae · generateAlgae · fromInteger
    fibsucc = fib · succ
```

Problema 5

```
permuta [] = return []
permuta l = do {(h, t) ← getR l; x ← permuta t; return (h : x)}
eliminatoria (Leaf x) = return x
eliminatoria (Fork (x, y)) = do {v ← eliminatoria x; z ← eliminatoria y; jogo (v, z)}
```

Quanto à primeira parte deste problema, fizemo-la, tal como pedido, com recurso ao monáde IO. Tendo em conta que a função *getR*, recebe uma lista com elementos de um determinado tipo, nos devolve um par com um elemento random dessa lista e com o resto da mesma sem esse elemento, então a nossa ideia foi chamar essa função até que chegasse ao fim da lista inicial, isto é, recursividade. Ficamos com o primeiro elemento de getR, e fazemos *permuta t*, em que t é o segundo elemento do par devolvido por getR, e assim sucessivamente.

Em relação à segunda parte do problema, elaboramo-la da mesma forma que elaborámos a primeira, com recursividade. Percorrendo todos os nodos de uma árvore, quando chegarmos às folhas de um Fork, chamamos a função jogo, fornecida no enunciado, tendo como argumentos as equipas que estão nessas folhas, devolvendo a probabilidade de cada equipa nesse jogo vencer. Realizando isto para toda a árvore, obtemos o resultado final esperado.

Índice