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

Departamento de Informática
Universidade do Minho

Maio de 2020

Grupo nr.	57
a76861	Gonçalo Dias Camaz Moreira
a77278	Carlos José Lima Gonçalves
a78452	José Pedro dos Santos Ferreira

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" [**?**], 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:

$\{\text{-\# OPTIONS}_G HC - XNPlusKPatterns\#-\}$
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

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

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

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 [**?**].)

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 [**?**] e nos focarmos apenas na parte que implementa a opção `-w`, verificamos que a poderíamos escrever, em Haskell, da forma seguinte:

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

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 $lookahead\_sep$.)

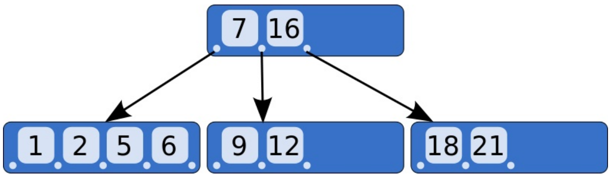
Problema 3

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

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

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:

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

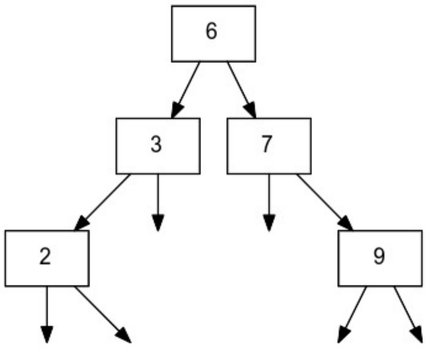
6. A biblioteca Exp permite representar árvores-expressão em formato DOT, que pode ser lido por aplicações como por exemplo Graphviz, produzindo as respectivas imagens. Por exemplo, para o caso de árvores BTree, se definirmos

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

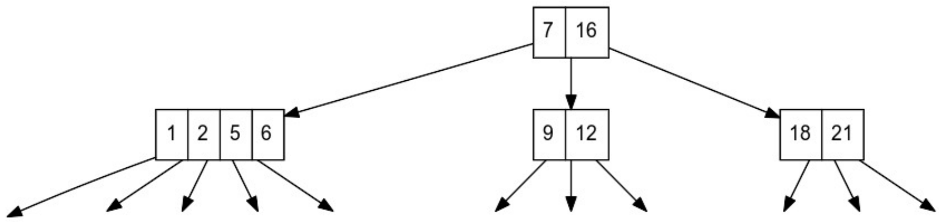
   executando $dotBTree\ t$ para

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

   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: $A \rightarrow A\ B, B \rightarrow A$.

Quer dizer, em cada iteração do "crescimento"da alga, cada A deriva num par $A\ B$ e cada B converte-se num A. Assim, ter-se-á, onde n é o número de iterações desse processo:

- $n = 0$: A
- $n = 1$: $A\ B$
- $n = 2$: $A\ B\ A$
- $n = 3$: $A\ B\ A\ A\ B$
- etc

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

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

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

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

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

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,

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

e a mesma coisa para os Bs:

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

Pretende-se, neste problema:

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

   $$\mathit{generateAlgae} :: \mathit{Int} \to \mathit{Algae}$$

   como anamorfismo de *Algae* e da função

   $$\mathit{showAlgae} :: \mathit{Algae} \to \mathit{String}$$

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

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

Problema 5

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

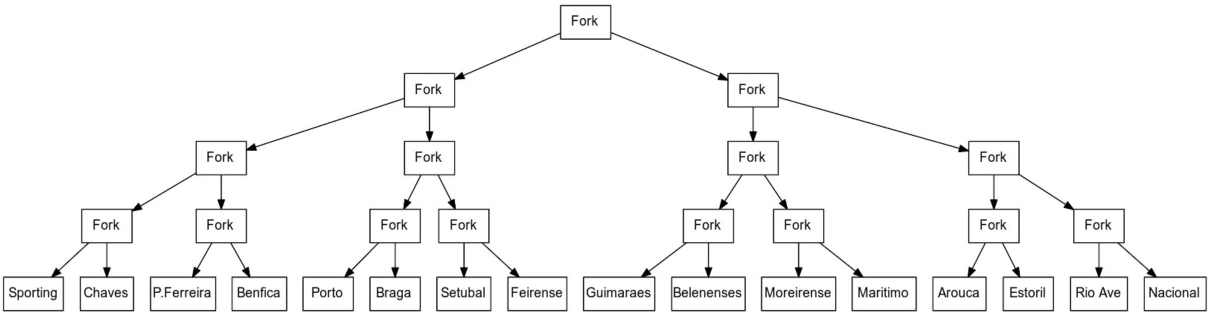
```
equipas :: [Equipa]
equipas = [
  "Arouca", "Belenenses", "Benfica", "Braga", "Chaves", "Feirense",
  "Guimaraes", "Maritimo", "Moreirense", "Nacional", "P.Ferreira",
  "Porto", "Rio Ave", "Setubal", "Sporting", "Estoril"
  ]
```

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 [?] — 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 [**?**].

[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

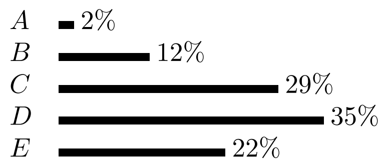
A Mónade para probabilidades e estatística

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

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

que o GHCi mostrará assim:

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

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

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

isto é

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

distribuição *normais*, eg.

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

etc.[10]

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

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

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

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

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

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

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

<=> { propriedades somatório }

$$\begin{cases} inv\ x\ 0 = 1 \\ inv\ x\ (n+1) = (1-x) \uparrow (n+1) + (inv\ x\ n) \end{cases}$$

<=> { função elevado }

$$\begin{cases} inv\ x\ 0 = 1 \\ inv\ x\ (n+1) = (elevado\ (1-x)\ (n+1)) + (inv\ x\ n) \end{cases}$$

<=> { propriedade aritmética - x$^{(n+1)} = x * x^n$}

$$\begin{cases} inv\ x\ 0 = 1 \\ inv\ x\ (n+1) = ((1-x) * (elevado\ (1-x)\ n)) + (inv\ x\ n) \end{cases}$$

<=> { definição elevado }

$$\begin{cases} \begin{cases} inv\ x\ 0 = 1 \\ inv\ x\ (n+1) = ((1-x) * (elevado\ (1-x)\ n)) + (inv\ x\ n) \end{cases} \\ \begin{cases} elevado\ (1-x)\ 0 = 1 \\ elevado\ (1-x)\ (n+1) = (1-x) * (elevado\ (1-x)\ n) \end{cases} \end{cases}$$

<=> { Def-comp, Def-const, Def-split, Def-proj, Def-(uncurry +), Def-((*) (1-x)), Def-succ }

$$\begin{cases} \begin{cases} (inv\ x \cdot \underline{0})\ n = \underline{1}\ n \\ (inv\ x \cdot \mathsf{succ})\ n = (\widehat{(+)} \cdot \langle ((*)\ (1-x)) \cdot elevado, inv\ x \rangle)\ n \end{cases} \\ \begin{cases} (elevado\ (1-x) \cdot \underline{0})\ n = \underline{1}\ n \\ (elevado\ (1-x) \cdot \mathsf{succ})\ n = (((*)\ (1-x)) \cdot \pi_1 \cdot \langle elevado, inv\ x \rangle)\ n \end{cases} \end{cases}$$

<=> { Igualdade extensional }

$$\begin{cases} \begin{cases} inv\ x \cdot \underline{0} = \underline{1} \\ inv\ x \cdot \mathsf{succ} = \widehat{(+)} \cdot \langle ((*)\ (1-x)) \cdot elevado, inv\ x \rangle \end{cases} \\ \begin{cases} elevado\ (1-x) \cdot \underline{0} = \underline{1} \\ elevado\ (1-x) \cdot \mathsf{succ} = ((*)\ (1-x)) \cdot \pi_1 \cdot \langle elevado, inv\ x \rangle \end{cases} \end{cases}$$

<=> { Absorção-x, Natural-id }

$$\begin{cases} \begin{cases} inv\ x \cdot \underline{0} = \underline{1} \\ inv\ x \cdot \mathsf{succ} = \widehat{(+)} \cdot (((*)\ (1-x)) \times id) \cdot \langle elevado, inv\ x \rangle \end{cases} \\ \begin{cases} elevado\ (1-x) \cdot \underline{0} = \underline{1} \\ elevado\ (1-x) \cdot \mathsf{succ} = ((*)\ (1-x)) \cdot \pi_1 \cdot \langle elevado, inv\ x \rangle \end{cases} \end{cases}$$

<=> { Cancelamento-+, Universal-+, Absorção-+, Natural-id }

$$\begin{cases} inv\ x \cdot [(\underline{0}), \mathsf{succ}] = [(\underline{1}), \widehat{(+)} \cdot (((*)\ (1-x)) \times id)] \cdot (id + \langle elevado, inv\ x \rangle) \\ elevado\ (1-x) \cdot [(\underline{0}), \mathsf{succ}] = [(\underline{1}), ((*)\ (1-x)) \cdot \pi_1] \cdot (id + \langle elevado, inv\ x \rangle) \end{cases}$$

<=> { Def-in, Def-F f }

$$\begin{cases} inv\ x \cdot \mathbf{in} = [(\underline{1}), \widehat{(+)} \cdot (((*)\ (1-x)) \times id)] \cdot \mathsf{F}\ \langle elevado, inv\ x \rangle \\ elevado\ (1-x) \cdot \mathbf{in} = [(\underline{1}), ((*)\ (1-x)) \cdot \pi_1] \cdot \mathsf{F}\ \langle elevado, inv\ x \rangle \end{cases}$$

<=> { Fokkinga }

$$\langle elevado\ (1-x), inv\ x \rangle = cataNat\ \langle [(\underline{1}), ((*)\ (1-x)) \cdot \pi_1], [(\underline{1}), \widehat{(+)} \cdot (((*)\ (1-x)) \times id)] \rangle$$

<=> { Lei da troca }

$$\langle elevado\ (1-x), inv\ x \rangle = cataNat\ [\langle \underline{1}, \underline{1} \rangle, \langle ((*)\ (1-x)) \cdot \pi_1, \widehat{(+)} \cdot (((*)\ (1-x)) \times id) \rangle]$$

Vemos que a função para ter a função inv apenas fazemos π_2 da função definida acima. A função definida acima é um ciclo for visto que é um catamorfismo de naturais.

$$\begin{array}{l} inv\_aux\ x\ 0 = 1 \\ inv\_aux\ x\ n = (elevado\ (1-x)\ n) + (inv\ x\ (n-1)) \end{array}$$

$$elevado\ x\ 0 = 1$$
$$elevado\ x\ n = x * (elevado\ x\ (n-1))$$

$$inv\ x = \pi_2 \cdot aux\ \textbf{where}\ aux = cataNat\ [\langle \underline{1}, \underline{1} \rangle, \langle ((*)\ (1-x)) \cdot \pi_1, \widehat{(+)} \cdot (((*)\ (1-x)) \times id) \rangle]$$
$$prop\_INV\ x\ n = ((n < 7000) \wedge (n \geqslant 0) \wedge (x > 1) \wedge (x < 2)) ==> (inv\ x\ n) \equiv (inv\_aux\ x\ n)$$
$$\quad \textbf{where}\ types = x :: Double$$
$$\qquad types1 = n :: Int$$

Problema 2

$$\begin{cases} wc\_w\ [\,] = 0 \\ wc\_w\ (c:l) = \textbf{if}\ ((\neg\ sep) \wedge (lookahead\ l))\ \textbf{then}\ (wc\_l\ l) + 1\ \textbf{else}\ wc\_w\ l \end{cases}$$

$<=>$ { função lookahead }

$$\begin{cases} \begin{cases} wc\_w\ [\,] = 0 \\ wc\_w\ (c:l) = \textbf{if}\ ((\neg\ sep) \wedge (lookahead\ l))\ \textbf{then}\ (wc\_l\ l) + 1\ \textbf{else}\ wc\_w\ l \end{cases} \\ \begin{cases} lookahead\ [\,] = True \\ lookahead\ (c:l) = sep\ c \end{cases} \end{cases}$$

$<=>$ { Def-comp, Def-x, Def-cond, Def-(uncurry $\wedge$), Def-proj, Def-cons, Def-succ }

$$\begin{cases} \begin{cases} wc\_w\ [\,] = 0 \\ wc\_w \cdot cons\ (c,l) = \begin{array}{l} \widehat{(\wedge)} \cdot ((\neg\ sep) \times (lookahead)) \rightarrow \\ \mathsf{succ} \cdot wc\_w \cdot \pi_2, \\ wc\_w \cdot \pi_2 \end{array} \quad (c,l) \end{cases} \\ \begin{cases} lookahead\ [\,] = True \\ lookahead \cdot cons\ (c,l) = sep \cdot \pi_1\ (c,l) \end{cases} \end{cases}$$

$<=>$ { Def-comp, Def-nil, Def-const }

$$\begin{cases} \begin{cases} wc\_w \cdot nil\ a = \underline{0}\ a \\ wc\_w \cdot cons\ (c,l) = \begin{array}{l} \widehat{(\wedge)} \cdot ((\neg\ sep) \times (lookahead)) \rightarrow \\ \mathsf{succ} \cdot wc\_w \cdot \pi_2, \\ wc\_w \cdot \pi_2 \end{array} \quad (c,l) \end{cases} \\ \begin{cases} lookahead \cdot nil\ a = \underline{True}\ a \\ lookahead \cdot cons\ (c,l) = sep \cdot \pi_1\ (c,l) \end{cases} \end{cases}$$

$<=>$ { Igualdade extensional }

$$\begin{cases} \begin{cases} wc\_w \cdot nil = \underline{0} \\ wc\_w \cdot cons = \begin{array}{l} \widehat{(\wedge)} \cdot ((\neg\ sep) \times (lookahead)) \rightarrow \\ \mathsf{succ} \cdot wc\_w \cdot \pi_2, \\ wc\_w \cdot \pi_2 \end{array} \end{cases} \\ \begin{cases} lookahead \cdot nil = \underline{True} \\ lookahead \cdot cons = sep \cdot \pi_1 \end{cases} \end{cases}$$

$<=>$ { Functor-x, Natural-π_1, Natural-π_2, Cancelamento-x, Natural-id, 2ª Lei de fusão do condicional }

$$\begin{cases} \begin{cases} wc\_w \cdot nil = \underline{0} \\ wc\_w \cdot cons = \begin{array}{l} \widehat{(\wedge)} \cdot ((\neg\ sep) \times (\pi_2)) \rightarrow \\ \mathsf{succ} \cdot \pi_1 \cdot \pi_2, \\ \pi_1 \cdot \pi_2 \end{array} \quad \cdot (id \times \langle wc\_w, lookahead \rangle) \end{cases} \\ \begin{cases} lookahead \cdot nil = \underline{True} \\ lookahead \cdot cons = sep \cdot \pi_1 \cdot (id \times \langle wc\_w, lookahead \rangle) \end{cases} \end{cases}$$

$<=>$ { Cancelamento-+, Universal-+, Def- in, Def-F f = id + id x f, Absorção-+ }

$$\left\{\begin{array}{l} wc\_w \cdot \mathbf{in} = [\underline{0}, (\begin{array}{l} \widehat{(\wedge)} \cdot ((\neg\ sep) \times (\pi_2)) \rightarrow \\ \mathsf{succ} \cdot \pi_1 \cdot \pi_2, \\ \pi_1 \cdot \pi_2 \end{array})] \cdot \mathsf{F}\ \langle wc\_w, lookahead \rangle \\ lookahead \cdot \mathbf{in} = [\underline{True}, sep \cdot \pi_1] \cdot \mathsf{F}\ \langle wc\_w, lookahead \rangle \end{array}\right.$$

<=> { Fokkinga }

$$\langle wc\_w, lookahead \rangle = cataList\ \langle [\underline{0}, (\begin{array}{l} \widehat{(\wedge)} \cdot ((\neg\ sep) \times (\pi_2)) \rightarrow \\ \mathsf{succ} \cdot \pi_1 \cdot \pi_2, \\ \pi_1 \cdot \pi_2 \end{array})], [\underline{True}, sep \cdot \pi_1] \rangle$$

<=> { Lei da troca }

$$\langle wc\_w, lookahead \rangle = cataList\ [\langle \underline{0}, \underline{True} \rangle, \langle (\begin{array}{l} \widehat{(\wedge)} \cdot ((\neg\ sep) \times (\pi_2)) \rightarrow \\ \mathsf{succ} \cdot \pi_1 \cdot \pi_2, \\ \pi_1 \cdot \pi_2 \end{array}), sep \cdot \pi_1 \rangle]$$

$$\begin{array}{l} wc\_w\_final :: [\,Char\,] \rightarrow Int \\ wc\_w\_final = wrapper \cdot worker \\ wrapper = \pi_1 \\ worker = cataList\ [\langle \underline{0}, \underline{True} \rangle, \langle cond\ (\widehat{(\wedge)} \cdot ((\neg \cdot sep) \times \pi_2))\ (\mathsf{succ} \cdot \pi_1 \cdot \pi_2)\ (\pi_1 \cdot \pi_2), sep \cdot \pi_1 \rangle] \\ \quad \mathbf{where}\ sep\ c = (c \equiv \texttt{' '}) \vee (c \equiv \texttt{'\textbackslash n'}) \vee (c \equiv \texttt{'\textbackslash t'}) \end{array}$$

Problema 3

Diagrama do catamorfismo

$$\begin{tikzcd} \textsf{B-tree}\ A \arrow[r, "outB\_tree"] \arrow[d, "cata\ g"'] & 1 + (\textsf{B-tree}\ A \times (A \times \textsf{B-tree}\ A)^*) \arrow[d, "id + (cata\ g) \times \mathsf{map}\ \ (id \times (cata\ g))"] \\ C & 1 + (C \times (A \times C)^*) \arrow[l, "g"] \end{tikzcd}$$

Diagrama do anamorfismo

$$\begin{tikzcd} \textsf{B-tree}\ A & 1 + (\textsf{B-tree}\ A \times (A \times \textsf{B-tree}\ A)^*) \arrow[l, "inB\_tree"'] \\ C \arrow[u, "ana\ g"] \arrow[r, "g"] & 1 + (C \times (A \times C)^*) \arrow[u, "id + (ana\ g) \times \mathsf{map}\ \ (id \times (ana\ g))"'] \end{tikzcd}$$

Diagrama do hylomorfismo

$$\begin{tikzcd} C \arrow[r, "f"] \arrow[d, "ana\ f"'] & 1 + (C \times (A \times C)^*) \arrow[d, "id + (ana\ f) \times \mathsf{map}\ \ (id \times (ana\ f))"] \\ \textsf{B-tree}\ A \arrow[r, "outB\_tree"] \arrow[d, "cata\ g"'] & 1 + (\textsf{B-tree}\ A \times (A \times \textsf{B-tree}\ A)^*) \arrow[d, "id + (cata\ g) \times \mathsf{map}\ \ (id \times (cata\ g))"] \\ D & 1 + (D \times (A \times D)^*) \arrow[l, "g"] \end{tikzcd}$$

Como podemos ver o inB_tree terá de construir a árvore a partir de nada ou de uma árvore de A's e uma lista de pares de A's árvores de A's pelo que terá de ser um either. Para isto basta usar os construtores definidos aquando da definição do tipo B_tree, apenas tendo cuidado de usar const Nil para transformar numa função e uncurry Block para que receba pares.

A função outB_tree terá de receber uma B_tree de A's e devolver ou 1, ou um par de B_tree de A's lista de pares A B_tree de A's. Devolverá 1 se a árvore for nada(ou seja Nil) e os pares caso contrário.

$$\begin{array}{l} inB\_tree = [\underline{Nil}, \widehat{Block}] \\ outB\_tree\ Nil = i_1\ () \\ outB\_tree\ (Block\ a\ b) = i_2\ (a, b) \end{array}$$

Para definir o Bi-functor basta olhar para os tipos isomorfos nos diagramas. Vendo que B_tree A é isomorfo a 1 + B_tree A × (A × B_tree A)* percebemos logo que o bi-functor terá de partir de: B(A, B_tree A) isomorfo 1 + B_tree A × (A × B_tree A)*; e o bi-functor será: B(f,g) = id + g × map(f × g).

Para definir o functor basta utilizar o bi-functor usando a primeira função como id.

Para definir os anamorfismo, catamorfismo e hylomorfismo basta reparar nos diagramas acima.

$$
\begin{array}{l}
recB\_tree\ f = id + f \times \mathsf{map}\ \ (id \times f) \\
baseB\_tree\ g\ f = id + f \times \mathsf{map}\ \ (g \times f) \\
cataB\_tree\ g = g \cdot recB\_tree\ (cataB\_tree\ g) \cdot outB\_tree \\
anaB\_tree\ g = inB\_tree \cdot recB\_tree\ (anaB\_tree\ g) \cdot g \\
hyloB\_tree\ f\ g = (cataB\_tree\ f) \cdot (anaB\_tree\ g)
\end{array}
$$

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Functor}\ \textsf{B-tree} \\
\quad \textbf{where}\ \mathsf{fmap}\ f = cataB\_tree\ (inB\_tree \cdot (baseB\_tree\ f\ id))
\end{array}
$$

<u>Diagrama inorder B_tree</u>

$$
\begin{tikzcd}
\textsf{B-tree}\ A \arrow[r, "outB\_tree"] \arrow[d, "inorder"'] & 1 + (\textsf{B-tree}\ A \times (A \times \textsf{B-tree}\ A)^*) \arrow[d, "id+(inorder)\times\mathsf{map}\ \ (id\times(inorder))"] \\
A^* & 1 + (A^* \times (A \times A^*)^*) \arrow[l, "geneinorder"]
\end{tikzcd}
$$

Para definir a função apenas dependemos do gene, pelo que apenas temos de definir o gene. Para que façamos uma travessia inorder, basta concatenar o resultado da mais árvore à esquerda com a concatenação da construção de cada par pertencente na lista. Caso a árvore seja vazia retorna-se a lista vazia.

```
Em pointwise seria:


inorder (Nil) = []
inorder (Block a b) = (inorder a) ++ d
          where (c,e) = unzip b
                f = (map (inorder)) e
                l = zip c f
                j = (map cons) l
                d = concat j
```

$$
\begin{array}{l}
geneinorder = [nil, \widehat{(+\!\!+)} \cdot (id \times concat \cdot (\mathsf{map}\ \ cons))] \\
inordB\_tree = cataB\_tree\ geneinorder
\end{array}
$$

<u>Diagrama da largestBlock</u>

$$
\begin{tikzcd}
\textsf{B-tree}\ A \arrow[r, "outB\_tree"] \arrow[d, "largestBlock"'] & 1 + (\textsf{B-tree}\ A \times (A \times \textsf{B-tree}\ A)^*) \arrow[d, "id+(largestBlock)\times\mathsf{map}\ \ (id\times(largestBlock))"] \\
Nat & 1 + (Nat \times (A \times Nat)^*) \arrow[l, "g"]
\end{tikzcd}
$$

Para definir a largestBlock apenas dependemos do gene. Sabendo que a recursividade já está feita(pelo conceito de catamorfismo), temos apenas de ver qual é o máximo do segundo elemento da lista de pares, o tamanho da lista(que será o tamanho do bloco atual) e comparar estes dois com o do tamanho da lista da esquerda, escolhendo o máximo. Caso a árvore seja vazia retorna 0.

```
Em pointwise seria:


largestBlocks (Nil) = 0
largestBlocks (Block a b) = max (largestBlocks a) d
          where c = (map p2) b
```

```
                l = (map largestBlocks) c
                e = maximum l
                d = max e (length c)
```

$$largestBlock = cataB\_tree\ [\underline{0}, \widehat{max} \cdot (id \times mais)]\ \mathbf{where}\ mais = \widehat{max} \cdot \langle \mathsf{length}\ , maximum \cdot (\mathsf{map}\ \ \pi_2) \rangle$$

Diagrama da mirrorB tree

$$\begin{tikzcd}
\text{B-tree } A \arrow[d, leftarrow, "mirror"'] & 1 + (\text{B-tree } A \times (A \times \text{B-tree } A)^*) \arrow[l, "inB_tree"'] \arrow[d, leftarrow, "id + (mirror) \times \mathsf{map}\ \ (id \times (mirror))"'] \\
\text{B-tree } A \arrow[r, "g"] & 1 + (\text{B-tree } A \times (A \times \text{B-tree } A)^*)
\end{tikzcd}$$

Para definir a mirrorB_tree basta definir o gene do anamorfismo. Para realizar esta função apenas temos de fazer outB_tree e depois utilizar uma alternativa(+). No primeiro caso, em que a árvore é vazia, retornamos o mesmo. No segundo caso, temos de separar a lista de pares em duas listas, depois colocar a árvore da esquerda na cabeça da lista de árvores, reverter as duas listas, tirar a cabeça à das árvores(para ser a da esquerda) e depois juntar as outras duas em listas de pares.

```
Em pointwise seria:

mirror (Block a b) = Block j d
              where (c,e) = unzip b
                    f = (map mirror) e
                    g = cons (mirror a,f)
                    h = reverse g
                    j = head h
                    d = zip (reverse c) (tail h)
```

$$gera\_lista = \langle \pi_1 \cdot \pi_2, cons \cdot (id \times \pi_2) \rangle \cdot (id \times \langle \mathsf{map}\ \ \pi_1, \mathsf{map}\ \ \pi_2 \rangle)$$
$$gene\_aux\_mirror = (id + ((\langle head \cdot \pi_2, \widehat{zip} \cdot (id \times tail) \rangle \cdot (reverse \times reverse) \cdot gera\_lista)))$$
$$mirrorB\_tree = anaB\_tree\ (gene\_aux\_mirror \cdot outB\_tree)$$

Diagrama da quickSort

$$\begin{tikzcd}
A^* \arrow[r, "f"] \arrow[d, "ana\ f"'] & 1 + (A^* \times (A \times A^*)^*) \arrow[d, "id + (ana\ f) \times \mathsf{map}\ \ (id \times (ana\ f))"] \\
\text{B-tree } A \arrow[r, "outB_tree"] \arrow[d, "cata\ g"'] & 1 + (\text{B-tree } A \times (A \times \text{B-tree } A)^*) \arrow[d, "id + (cata\ g) \times \mathsf{map}\ \ (id \times (cata\ g))"] \\
A^* & 1 + (A^* \times (A \times A^*)^*) \arrow[l, "g"]
\end{tikzcd}$$

Para realizar esta função temos de partir a lista em um par de lista, com listas de pares de elemento lista. Nesta partição decidimos usar que cada lista(2ºelemento do par) ia ter 2 pares. Caso só existisse 1 elemento é trivial. Se existissem mais, pegaríamos nos 2 primeiros elementos e calcularíamos o máximo e mínimo de ambos, depois a lista do 1ºpar(1ºelemento) ia ser os elementos menores do que o mínimo. Depois a lista do primeiro par do 2ºelemento do par ia conter os elementos que estavam entre os 2 escolhidos e depois os maiores do que o máximo iam ficar no último par dp 2ºelemento do par.

Depois de ter uma B_tree ordenada, basta fazer inorder.

```
Em pointwise seria:

qSort [] = []
qSort t = (qSort a) ++ d
      where (a, b) = parteNTree t
            (e,c) = unzip b
```

```
        f = (map qSort) c
        g = zip e f
        d = concat ((map cons) g)
```

$$
\begin{array}{l}
lsplitB\_tree\ [\,] = i_1\ ()\\
lsplitB\_tree\ x = i_2\ (parteB\_tree\ x)\\
parteB\_tree :: (Ord\ a) \Rightarrow [a] \rightarrow ([a], [(a, [a])])\\
parteB\_tree\ [x] = ([\,], [(x, [\,])])\\
parteB\_tree\ (h1 : h2 : t) = (c, [(a, d), (b, e)])\\
\quad \textbf{where}\ a = min\ h1\ h2\\
\qquad b = max\ h1\ h2\\
\qquad c = filter\ (\lambda x \rightarrow x \leqslant a)\ t\\
\qquad d = filter\ (\lambda x \rightarrow (x > a) \wedge (x < b))\ t\\
\qquad e = filter\ (\lambda x \rightarrow x \geqslant b)\ t\\
qSortB\_tree :: (Ord\ a) \Rightarrow [a] \rightarrow [a]\\
qSortB\_tree = hyloB\_tree\ geneinorder\ lsplitB\_tree
\end{array}
$$

<u>Diagrama da cB_tree2Exp</u>

$$
\begin{tikzcd}
\text{B-tree } A \arrow[r, "outB\_tree"] \arrow[d, "cB\_tree2Exp"'] & 1 + (\text{B-tree } A \times (A \times \text{B-tree} A)^*) \arrow[d, "id + (cB\_tree2Exp) \times \mathsf{map}\ (f \times (cB\_tree2Exp))"] \\
Exp\ t^* A^* & 1 + (Exp\ t^* A^* \times (A \times Exp\ t^* A^*)^*) \arrow[l, "g"]
\end{tikzcd}
$$

Para a função dotB_tree apenas necessitamos de definir a função cB_tree2Exp. Mais uma vez para definir esta função apenas precisamos de definir o gene. Neste caso se for uma árvore vazia, dá-mos a árvore de expressões com a variável lista vazia, caso contrário dá-mos uma árvore de expressões em que a operação é cada primeiro elemento da lista e as restantes árvores são a mais à esquerda e os segundos elementos da lista.

$$
\begin{array}{l}
dotB\_tree :: Show\ a \Rightarrow \text{B-tree}\ a \rightarrow \mathsf{IO}\ ExitCode\\
dotB\_tree = dotpict \cdot bmap\ nothing\ (Just \cdot show) \cdot cB\_tree2Exp\\
cB\_tree2Exp = cataB\_tree\ (inExp \cdot (nil + \langle (\mathsf{map}\ \ \pi_1) \cdot \pi_2, cons \cdot (id \times (\mathsf{map}\ \ \pi_2)) \rangle))
\end{array}
$$

Problema 4

<u>Diagrama do anamorfismo A</u>

$$
\begin{tikzcd}
A & 1 + A \times B \arrow[l, "inA"] \\
C \arrow[u, "[\![ga\ gb]\!]_A"] \arrow[r, "ga"] & 1 + (C \times D) \arrow[u, "id + ([\![ga\ gb]\!]_A \times [\![ga\ gb]\!]_B)"']
\end{tikzcd}
$$

<u>Diagrama do anamorfismo B</u>

$$
\begin{tikzcd}
B & 1 + A \arrow[l, "inB"] \\
D \arrow[u, "[\![ga\ gb]\!]_B"] \arrow[r, "gb"] & 1 + C \arrow[u, "id + [\![ga\ gb]\!]_A"']
\end{tikzcd}
$$

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

<u>Diagrama da generateAlgae</u>

$$
\begin{tikzcd}
A & 1 + A \times B \arrow[l, "inA"] \\
Nat \arrow[u, "generateAlgae = [\![ga\ gb]\!]_A"] \arrow[r, "ga"] & 1 + (Nat \times Nat) \arrow[u, "id + ([\![ga\ gb]\!]_A \times [\![ga\ gb]\!]_B)"']
\end{tikzcd}
$$

$$\begin{tikzcd}
B & & 1 + A \arrow[ll, "inB"'] \\
Nat \arrow[u, "\llbracket (ga\ gb) \rrbracket_B"] \arrow[rr, "gb"] & & 1 + Nat \arrow[u, "id + \llbracket (ga\ gb) \rrbracket_A"']
\end{tikzcd}$$

Para realizar esta função temos de definir dois genes um para A e outro para B. O gene de B é trivial, pois é igual ao outNat. O gene de A é bastante parecido, diferindo apenas que em vez de darmos um natural, dá-mos dois iguais(pois estão ao mesmo nível).

```
Em pointwise seria:

genAlgae :: Integer -> Algae
genAlgae 0 = NA
genAlgae n = A (genAlgae (n-1)) (genBl (n-1))

genBl :: Integer -> B
genBl 0 = NB
genBl n = B (genAlgae (n-1))
```

$$outDuplo\ 0 = i_1\ ()$$
$$outDuplo\ (n) = i_2\ (n-1, n-1)$$
$$generateAlgae = \llbracket (outDuplo\ outNat) \rrbracket_A$$

Diagrama da showAlgae

$$\begin{tikzcd}
A \arrow[r, "outA"] \arrow[d, "showAlgae = (\!| ga\ gb |\!)_A"'] & 1 + A \times B \arrow[d, "id + showAlgae"] \\
String & 1 + (String \times String) \arrow[l, "ga"]
\end{tikzcd}$$

$$\begin{tikzcd}
B \arrow[r, "outB"] \arrow[d, "(\!| ga\ gb |\!)_B"'] & 1 + A \arrow[d, "id + (\!| ga\ gb |\!)_B"] \\
String & 1 + String \arrow[l, "gb"]
\end{tikzcd}$$

Para realizar esta função temos dois casos, o A e B.

Para o gene A vemos que no caso de paragem temos de dar a String ”A”, e no outro caso apenas temos de concatenar as duas Strings que já foram geradas recursivamente. Para o gene B no caso de paragem temos de retornar a String ”B”, e no outro caso apenas temos de retornar a Stirng gerada recursivamente.

```
Em pointwise seria:

showAlgae :: Algae -> String
showAlgae (NA) = "A"
showAlgae (A a b) = (showAlgae a) ++ (showBl b)

showBl :: B -> String
showBl (NB) = "B"
showBl (B a) = showAlgae a
```

$$my\_fib\ n = y\ \textbf{where}\ (x, y) = fib\_aux\ n$$
$$fib\_aux\ 0 = (1, 1)$$
$$fib\_aux\ (n+1) = (x + y, x)$$
$$\quad \textbf{where}\ (x, y) = fib\_aux\ n$$
$$showAlgae = (\!| [\underline{\texttt{"A"}}, \widehat{(+\!\!+)}]\ [\underline{\texttt{"B"}}, id] |\!)_A$$
$$prop\_LS\ n = ((n \geqslant 0) \land (n \leqslant 23)) ==> ((\mathsf{length}\ \cdot showAlgae \cdot generateAlgae)\ n) \equiv ((my\_fib \cdot \mathsf{succ})\ n)$$

Problema 5

A função permuta é uma função monádica, pelo que para chegar à sua definição começamos por defini-la como se estivesse com o mónade identidade e depois usamos as regras para chegar à sua real definição.

```
permuta [] = id []
permuta a = let (b,c) = getR a
                d = permuta c
             in id (b:c)
```

Utilizando as regras(id torna-se return e let torna-se do com o sinal = a passar a ←), chegamos à definição abaixo.

$$
\begin{array}{l}
permuta\ [\,] = return\ [\,] \\
permuta\ a = \mathbf{do}\ \{ \\
\quad (d, b) \leftarrow getR\ a; \\
\quad c \leftarrow permuta\ b; \\
\quad return\ (d : c) \\
\}
\end{array}
$$

Diagrama eliminatória

$$
\begin{tikzcd}
\mathsf{LTree}\ Equipa \arrow[r, "outLTree"] \arrow[d, "eliminatoria"'] & Equipa + (\mathsf{LTree}\ Equipa \times \mathsf{LTree}\ Equipa) \arrow[d, "id+(eliminatoria \times eliminatoria)"] \\
\mathsf{Dist}\ Equipa & Equipa + (\mathsf{Dist}\ Equipa \times \mathsf{Dist}\ Equipa)) \arrow[l, "geneEliminatoria"]
\end{tikzcd}
$$

Vendo no diagrama e como estamos a trabalhar com mónads, primeiramente utilizamos o código ¡penas com o mónad identidade e posteriormente tornamos monádico(neste caso com o mónad das Distribuições) com as regras vistas nas aulas.

O geneEliminatória terá de dar uma equipa, a partir de uma equipa ou de um par de equipas, logo terá de ser um either. Partindo de uma equipa para uma equipa apenas sabemos a função identidade. Partindo de um par de equipas e dando uma equipa(vencedora neste caso), teremos de realizar a função jogar(que realiza um jogo entre duas equipas e dá o vencedor).

```
jogar (e1, e2) = let a = e1
                     b = e2
                     c = jogar(a,b)
                  in id c

Pelo que será:
        geneEliminatoria = [id,jogar]
```

Utilizando as regras vistas, basta que o id se torne return e o let se torne do, com os sinais = a tornarem-se ←. Logo a função jogar dará a probabilidade de cada equipa ganhar(um mónad de Distribuições de probabilidade). Pelo que a definição das funções jogar e eliminatória são as seguintes.

$$
\begin{array}{l}
jogar :: (\mathsf{Dist}\ Equipa, \mathsf{Dist}\ Equipa) \rightarrow \mathsf{Dist}\ Equipa \\
jogar\ (e_1, e_2) = \mathbf{do}\ \{ \\
\quad a \leftarrow e_1; \\
\quad b \leftarrow e_2; \\
\quad c \leftarrow (jogo\ (a, b)); \\
\quad return\ c \\
\} \\
eliminatoria = cataLTree\ [return, jogar]
\end{array}
$$

Índice