

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

Departamento de Informática
Universidade do Minho

Junho de 2017

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

Contents

1	Preâmbulo	2
2	Documentação	2
3	Como realizar o trabalho	3
A	Mónade para probabilidades e estatística	10
B	Definições auxiliares	11
C	Soluções propostas	11

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-processor que faz “pretty printing” de código Haskell em **L^AT_EX** 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
```

¹O suffixo ‘lhs’ quer dizer *literate Haskell*.

```

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

```

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

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

²Para 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:

```
wc_w :: [Char] -> Int
wc_w [] = 0
wc_w (c:l) =
  if ¬ (sep c) ∧ lookahead_sep l
  then wc_w l + 1
  else wc_w l
  where
    sep c = (c ≡ ' ' ∨ c ≡ '\n' ∨ c ≡ '\t')
    lookahead_sep [] = True
    lookahead_sep (c:l) = sep c

```

Re-implemente esta função segundo o modelo *worker/wrapper* onde *wrapper* deverá ser um catamorfismo 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:

```
data B-tree a = Nil | Block { leftmost :: B-tree a, block :: [(a, B-tree a)] } deriving (Show, Eq)

```

Por exemplo, a B-tree³

³Créditos: figura extraída de <https://en.wikipedia.org/wiki/B-tree>.



é representada no tipo acima por:

```

t = Block {
  leftmost = Block {
    leftmost = Nil,
    block = [(1, Nil), (2, Nil), (5, Nil), (6, Nil)]},
  block = [
    (7, Block {
      leftmost = Nil,
      block = [(9, Nil), (12, Nil)]}),
    (16, Block {
      leftmost = Nil,
      block = [(18, Nil), (21, Nil)]})
  ]}

```

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* → [*t*] que faça travessias “inorder” de árvores deste tipo.
3. Definir como um catamorfismo a função *largestBlock* :: B-tree *a* → *Int* que detecta o tamanho do maior bloco da árvore argumento.
4. Definir como um anamorfismo a função *mirrorB_tree* :: B-tree *a* → B-tree *a* que roda a árvore argumento de 180°
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:

```

lsplitB_tree [] = i1 ()
lsplitB_tree [7] = i2 ([], [(7, [])])
lsplitB_tree [5, 7, 1, 9] = i2 ([1], [(5, []), (7, [9])])
lsplitB_tree [7, 5, 1, 9] = i2 ([1], [(5, []), (7, [9])])

```

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

```

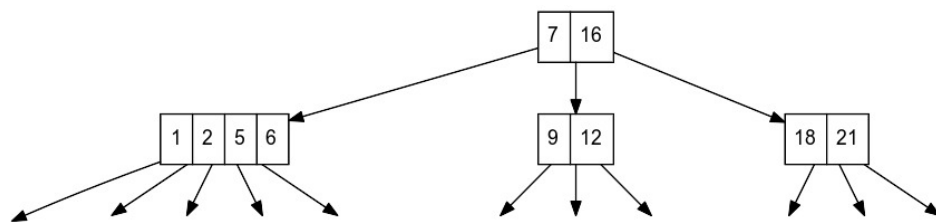
t = Node (6, (Node (3, (Node (2, (Empty, Empty)), Empty)), Node (7, (Empty, Node (9, (Empty, Empty))))))

```

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 árvore 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 Lindenmayer⁵ 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

⁴Como alternativa a instalar [Graphviz](#), podem usar [WebGraphviz](#) num browser.

⁵Ver 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:

```
type Algae = A
data A = NA | A A B deriving Show
data B = NB | B A deriving Show
```

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

```
inA :: 1 + A × B → A
inA = [NA, A]
outA :: A → 1 + A × B
outA NA = i1 ()
outA (A a b) = i2 (a, b)
inB :: 1 + A → B
inB = [NB, B]
outB :: B → 1 + A
outB NB = i1 ()
outB (B a) = i2 a
```

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

$$\begin{aligned} \llbracket \cdot \cdot \rrbracket_A &:: (1 + c \times d \rightarrow c) \rightarrow (1 + c \rightarrow d) \rightarrow A \rightarrow c \\ \llbracket ga \ gb \rrbracket_A &= ga \cdot (id + \llbracket ga \ gb \rrbracket_A \times \llbracket ga \ gb \rrbracket_B) \cdot outA \end{aligned}$$

e a mesma coisa para os B s:

$$\begin{aligned} \llbracket \cdot \cdot \rrbracket_B &:: (1 + c \times d \rightarrow c) \rightarrow (1 + c \rightarrow d) \rightarrow B \rightarrow d \\ \llbracket ga \ gb \rrbracket_B &= gb \cdot (id + \llbracket ga \ gb \rrbracket_A) \cdot outB \end{aligned}$$

Pretende-se, neste problema:

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

$$generateAlgae :: Int \rightarrow Algae$$

como anamorfismo de $Algae$ e da função

$$showAlgae :: Algae \rightarrow String$$

como catamorfismo de $Algae$.

3. Use **QuickCheck** para verificar a seguinte propriedade:

$$length \cdot showAlgae \cdot generateAlgae = fib \cdot succ$$

Problema 5

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

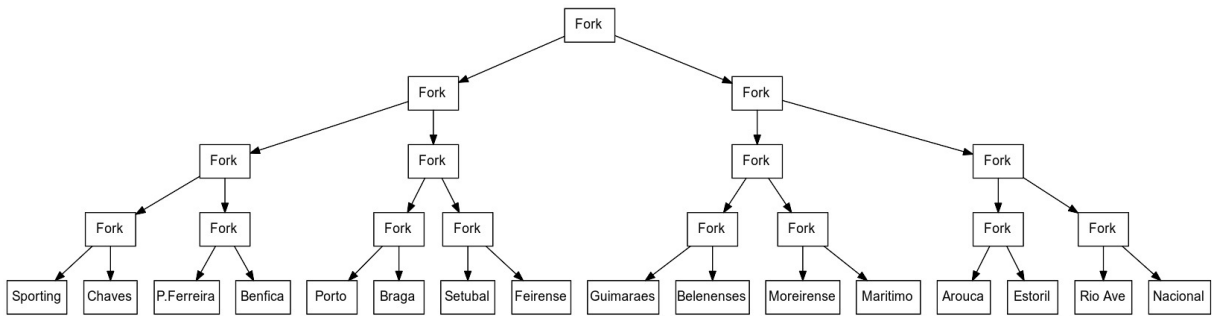
```
equipas :: [Equipa]
equipas = [
  "Arouca", "Belenenses", "Benfica", "Braga", "Chaves", "Feirense",
  "Guimaraes", "Maritimo", "Moreirense", "Nacional", "P.Ferreira",
  "Porto", "Rio Ave", "Setubal", "Sporting", "Estoril"
]
```

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, $f(\text{"Arouca"}, \text{"Braga"})$ 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 $\text{Dist } a$ 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

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

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 enunciado⁷, 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 $[Equipa] \rightarrow \text{Dist } Equipa$,

$quem_vence :: [Equipa] \rightarrow \text{Dist } Equipa$
 $quem_vence = eliminatória \cdot sorteio$

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

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

⁷Pode, 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 $\text{sorteio} :: [Equipa] \rightarrow \text{LTree } Equipa$ tem a seguinte arquitectura,⁸

$$\text{sorteio} = \text{anaLTree } \text{lsplit} \cdot \text{envia} \cdot \text{permuta}$$

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

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

A presença do mónade de IO tem a ver com a geração de números aleatórios⁹.

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

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

$\text{getR } x$ 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

$$\text{eliminatória} :: \text{LTree } Equipa \rightarrow \text{Dist } Equipa$$

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. Ritchie. *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.

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

⁹Quem estiver interessado em detalhes deverá consultar **System.Random**.

Anexos

A Mónade para probabilidades e estatística

Mónades são funtores 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

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

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 ,

A	■	2%
B	■	12%
C	■	29%
D	■	35%
E	■	22%

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 $\text{return } a = D [(a, 1)]$ 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 $g:A \rightarrow \text{Dist } B$ e $f:B \rightarrow \text{Dist } C$ 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.

¹⁰Para mais detalhes ver o código fonte de **Probability**, que é uma adaptação da biblioteca **PHP** (“Probabilistic Functional Programming”). Para quem quiser saber 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

Em primeiro lugar escrevemos a função *inv* em pointwise, e seguimos a sugestão da secção 3.16 dos apontamentos para chegar à definição de *inv x* como um ciclo *for*.

$$\begin{aligned}
& \begin{cases} inv\ x\ 0 = 1 \\ inv\ x\ (n+1) = inv\ x\ n + aux\ x\ n \\ aux\ x\ 0 = (1-x) \\ aux\ x\ (n+1) = (1-x) * aux\ x\ n \end{cases} \\
\Longleftrightarrow & \{ (27), (73), (74), (76) \} \\
& \begin{cases} (inv \cdot [0, succ])\ x = [\underline{1}, \widehat{+}] \cdot \langle inv, aux \rangle \\ (aux \cdot [0, succ])\ x = [1-x, *(1-x)] \end{cases} \\
\Longleftrightarrow & \{ \text{def-inNat}, (22) \} \\
& \begin{cases} (inv \cdot inNat)\ x = [\underline{1}, add] \cdot F\ \langle inv, aux \rangle \\ (aux \cdot inNat)\ x = [1-x, *(1-x) \cdot \pi_2] \cdot F\ \langle inv, aux \rangle \end{cases} \\
\Longleftrightarrow & \{ (50) \} \\
& \langle inv, aux \rangle\ x = cataNat\ \langle [\underline{1}, \widehat{+}], [1-x, *(1-x)] \cdot \pi_2 \rangle \\
\Longleftrightarrow & \{ (28) \} \\
& \langle inv, aux \rangle\ x = cataNat\ [\langle \underline{1}, 1-x \rangle, \langle \widehat{+}, *(1-x) \rangle \cdot \pi_2] \\
\Longleftrightarrow & \{ \text{def for} \} \\
& \langle inv, aux \rangle\ x = \text{for}\ \langle \widehat{+}, *(1-x) \rangle \cdot \pi_2\ (1, (1-x))
\end{aligned}$$

Como apenas interessa o primeiro elemento do par, aplicamos $p1$ ao resultado do ciclo `for`, chegando assim à definição final.

```

inv x = π1 · for ⟨ $\widehat{+}$ ,  $*(1-x)$ ⟩ · π2 (1, (1-x))
genVal :: Gen (Float, Int)
genVal = do
  x ← Test.QuickCheck.choose (1, 2)
  y ← Test.QuickCheck.choose (1000, 1000)
  return (x, y)
prop_inv f =
  forAll genVal $ λ(x, y) →
    abs ((1 / x) - (inv x y)) < f
  where types = f :: Float
testInv f = quickCheck (prop_inv f)

```

Problema 2

```

wc_w_final :: [Char] → Int
wc_w_final = wrapper · worker
wrapper = π1
worker = cataList ⟨[0, aux], [True, sep · π1⟩⟩
  where
    sep = cond (λc → c ≡ ' ' ∨ c ≡ '\n' ∨ c ≡ '\t') true false
    aux = cond (( $\widehat{\wedge}$ ) · ⟨¬ · sep · π1, π2 · π2⟩) (succ · π1 · π2) (π1 · π2)
prop_wc x =
  wc_w_final x ≡ wc_w x
testWc = quickCheck prop_wc

```

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

$$\begin{aligned}
& \left\{ \begin{array}{l} wc_w [] = 0 \\ wc_w (c : l) = \text{if } (\neg sep\ c) \wedge lookahead_sep\ l \text{ then } wc_w\ l + 1 \text{ else } wc_w\ l \end{array} \right. \\
& \Leftrightarrow \{ (74),(76),(79),(81), \text{def succ, def nil, def cons} \} \\
& (wc_w \cdot nil)\ x = \underline{0}\ x \\
& (wc_w \cdot cons)\ (c, l) = \text{if } (\neg sep \cdot \pi_1) \wedge (lookahead_sep \cdot \pi_2)\ (c, l) \text{ then } (succ \cdot wc_w \cdot \pi_2)\ (c, l) \text{ else } (wc_w \cdot \pi_2)\ (c, l) \\
& \Leftrightarrow \{ (76), \text{def uncurry (and), (78), (80)} \} \\
& (wc_w \cdot nil) = \underline{0} \\
& (wc_w \cdot cons)\ (c, l) = \text{cond } ((\widehat{\wedge}) \cdot \langle \neg \cdot sep \cdot \pi_1, lookahead_sep \cdot \pi_2 \rangle) (succ \cdot wc_w \cdot \pi_2)\ (wc_w \cdot \pi_2)\ (c, l) \\
& \Leftrightarrow \{ (73) \} \\
& wc_w \cdot nil = \underline{0} \\
& wc_w \cdot cons = \text{cond } ((\widehat{\wedge}) \cdot \langle \neg \cdot sep \cdot \pi_1, lookahead_sep \cdot \pi_2 \rangle) (succ \cdot wc_w \cdot \pi_2)\ (wc_w \cdot \pi_2) \\
& \Leftrightarrow \{ (1),(12),(7) \} \\
& wc_w \cdot nil = \underline{0} \\
& wc_w \cdot cons = \text{cond } ((\widehat{\wedge}) \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) (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) \\
& \Leftrightarrow \{ (13) \} \\
& wc_w \cdot nil = \underline{0} \\
& wc_w \cdot cons = \text{cond } ((\widehat{\wedge}) \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) (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)) \\
& \Leftrightarrow \{ (9) \} \\
& wc_w \cdot nil = \underline{0} \\
& wc_w \cdot cons = \text{cond } ((\widehat{\wedge}) \cdot (\langle \neg \cdot sep \cdot \pi_1, \pi_2 \cdot \pi_2 \rangle \cdot (id \times \langle wc_w, lookahead_sep \rangle))) (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)) \\
& \Leftrightarrow \{ (32) \} \\
& wc_w \cdot nil = \underline{0} \\
& wc_w \cdot cons = \text{cond } ((\widehat{\wedge}) \cdot (\langle \neg \cdot sep \cdot \pi_1, \pi_2 \cdot \pi_2 \rangle) (succ \cdot \pi_1 \cdot \pi_2)\ (\pi_1 \cdot \pi_2) \cdot (id \times \langle wc_w, lookahead_sep \rangle))) \\
& \left\{ \begin{array}{l} lookahead_sep [] = True \\ lookahead_sep (c : l) = sep\ c \end{array} \right. \\
& \Leftrightarrow \{ (81), \text{def nil, def cons, (76)} \} \\
& \left\{ \begin{array}{l} lookahead_sep \cdot nil = \underline{True} \\ (lookahead_sep \cdot cons)\ (c, l) = (sep \cdot \pi_1)\ (c, l) \end{array} \right. \\
& \Leftrightarrow \{ (73) \} \\
& \left\{ \begin{array}{l} lookahead_sep \cdot nil = \underline{True} \\ lookahead_sep \cdot cons = sep \cdot \pi_1 \end{array} \right.
\end{aligned}$$

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:

$$\begin{aligned} wc_w \cdot \mathbf{in} &= [0, \text{cond } ((\widehat{\wedge}) \cdot (\langle \neg \cdot sep \cdot \pi_1, \pi_2 \cdot \pi_2 \rangle (\text{succ} \cdot \pi_1 \cdot \pi_2) (\pi_1 \cdot \pi_2) \cdot (id \times \langle wc_w, lookahead_sep \rangle)))] \\ lookahead_sep \cdot \mathbf{in} &= [\underline{True}, sep \cdot \pi_1] \end{aligned}$$

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

$$\begin{aligned} wc_w \cdot \mathbf{in} &= [0, \text{cond } ((\widehat{\wedge}) \cdot (\langle \neg \cdot sep \cdot \pi_1, \pi_2 \cdot \pi_2 \rangle (\text{succ} \cdot \pi_1 \cdot \pi_2) (\pi_1 \cdot \pi_2) \cdot (id \times \langle wc_w, lookahead_sep \rangle)))] \cdot \\ & (id \times \langle wc_w, lookahead_sep \rangle) \\ lookahead_sep \cdot \mathbf{in} &= [\underline{True}, sep \cdot \pi_1] \cdot (id \times \langle wc_w, lookahead_sep \rangle) \\ \Leftrightarrow & \quad \{ (50) \} \\ \langle wc_w, lookahead_sep \rangle &= \text{cataList } \langle [0, \text{cond } ((\widehat{\wedge}) \cdot (\langle \neg \cdot sep \cdot \pi_1, \pi_2 \cdot \pi_2 \rangle (\text{succ} \cdot \pi_1 \cdot \pi_2) (\pi_1 \cdot \pi_2)))] , [\underline{True}, sep \cdot \pi_1] \rangle \end{aligned}$$

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

Em primeiro lugar, começamos por construir o diagrama de um cataB_tree para construir a biblioteca do tipo.

$$\begin{array}{ccc} \text{B-tree } A & \xleftarrow{\text{inB_tree}} & 1 + \text{B-tree } A \times (A \times \text{B-tree } A)^* \\ \text{cataB_tree } g \downarrow & & \downarrow \text{id} + \text{cataB_tree } g \times (\text{map } (id \times \text{cataB_tree } g)) \\ B & \xleftarrow{g} & 1 + B \times (A \times B)^* \end{array}$$

Alinea 1

$$\begin{aligned} \text{inB_tree} &:: () + (\text{B-tree } a, [(a, \text{B-tree } a)]) \rightarrow \text{B-tree } a \\ \text{inB_tree} &= [\underline{Nil}, \widehat{Block}] \\ \text{outB_tree } Nil &= i_1 () \\ \text{outB_tree } (Block \{ \text{leftmost} = l, \text{block} = b \}) &= i_2 (l, b) \\ \text{recB_tree } f &= \text{baseB_tree } id \, f \\ \text{baseB_tree } g \, f &= id + f \times \text{map } (g \times f) \\ \text{cataB_tree } g &= g \cdot \text{recB_tree } (\text{cataB_tree } g) \cdot \text{outB_tree} \\ \text{anaB_tree } g &= \text{inB_tree} \cdot \text{recB_tree } (\text{anaB_tree } g) \cdot g \\ \text{hyloB_tree } f \, g &= \text{cataB_tree } f \cdot \text{anaB_tree } g \end{aligned}$$

Para definir o fmap para B_tree , usamos a lei 47, Def-map-cata.

$$\begin{aligned} \text{instance Functor B-tree} \\ \text{where fmap } f &= \text{cataB_tree } (\text{inB_tree} \cdot \text{baseB_tree } f \, id) \end{aligned}$$

Alinea 2

Para chegarmos à definição de inordB_tree , construímos o diagrama de catamorfismos:

$$\begin{array}{ccc} \text{B-tree } A & \xleftarrow{\text{inB_tree}} & 1 + \text{B-tree } A \times (A \times \text{B-tree } A)^* \\ \text{cataB_tree } g \downarrow & & \downarrow \text{id} + \text{cataB_tree } g \times (\text{map } (id \times \text{cataB_tree } g)) \\ A^* & \xleftarrow{g} & 1 + A^* \times (A \times A^*)^* \end{array}$$

No caso da árvore vazia, geramos a lista vazia. No outro caso:

$$(A \times A^*)^* \xrightarrow[\text{concat} \cdot \text{map} \text{ cons}]{\text{cons}} A^*$$

Para chegarmos à lista final, basta aplicar a função *conc*, versão uncurried da função *(++)*.

$$A^* \times A^* \xrightarrow[(++)]{\text{conc}} A^*$$

Por fim chegamos à definição de *inordB_tree*.

$$\text{inordB_tree} = \text{cataB_tree} [\text{nil}, \text{conc} \cdot (\text{id} \times (\text{concat} \cdot \text{map} \text{ cons}))]$$

Alinea 3

Para fazermos a função *largestBlock*, fizemos um diagrama de catamorfismos.

$$\begin{array}{ccc} \text{B-tree } A & \xleftarrow{\text{inB_tree}} & 1 + \text{B-tree } A \times (A \times \text{B-tree } A)^* \\ \text{cataB_tree } g \downarrow & & \downarrow \text{id} + \text{cataB_tree } g \times (\text{map } (\text{id} \times \text{cataB_tree } g)) \\ \text{Int} & \xleftarrow{g} & 1 + \text{Int} \times (A \times \text{Int})^* \end{array}$$

Para chegarmos à definição do gene, fomos por partes. A primeira foi obter apenas uma lista com apenas o segundo elemento do par. Depois, criar um par com o tamanho dessa lista e o valor do maior elemento da lista. Por fim, guardar o maior dos dois.

$$A \times \text{Int}^* \xrightarrow{\text{map } \pi_2} \text{Int}^* \xrightarrow{\widehat{\text{max}} \cdot \langle \text{length}, \text{maximum} \rangle} \text{Int}$$

Definindo a função anterior como *f*:

$$1 + \text{Int} \times (A \times \text{Int})^* \xrightarrow{\text{id} + \text{id} \times f} 1 + \text{Int} \times \text{Int} \xrightarrow{[0, \widehat{\text{max}}]} \text{Int}$$

Chegando assim à definição do gene do catamorfismo.

$$\begin{aligned} \text{largestBlock} &= \text{cataB_tree} ([0, \widehat{\text{max}}] \cdot (\text{id} + \text{id} \times f)) \\ \text{where } f &= \widehat{\text{max}} \cdot \langle \text{length}, \text{maximum} \rangle \cdot (\text{map } \pi_2) \end{aligned}$$

Alinea 4

Para chegar à definição de *mirrorB_tree*, construímos um diagrama de anamorfismos.

$$\begin{array}{ccc} \text{B-tree } A & \xrightarrow{g} & 1 + \text{B-tree } A \times (A \times \text{B-tree } A)^* \\ \text{anaB_tree } g \downarrow & & \downarrow \text{id} + \text{anaB_tree } g \times (\text{map } (\text{id} \times \text{anaB_tree } g)) \\ \text{B-tree } A & \xleftarrow{\text{inB_tree}} & 1 + \text{B-tree } A \times (A \times \text{B-tree } A)^* \end{array}$$

O gene do anamorfismo é representado nos seguintes diagramas:

$$\text{B-tree } A \xrightarrow{\text{outB_tree}} 1 + \text{B-tree } A \times (A \times \text{B-tree } A)^* \xrightarrow{\text{id} + \text{id} \times (\text{unzip} \cdot \text{reverse})} 1 + \text{B-tree } A \times ((A)^* \times (\text{B-tree } A)^*)$$

Chegando à definição final de *mirrorB_tree*.

$$\begin{aligned} \text{mirrorB_tree} &= \text{anaB_tree} ((\text{id} + ((g \cdot f) \cdot (\text{id} \times (\text{unzip} \cdot \text{reverse})))) \cdot \text{outB_tree}) \\ \text{where} \\ f(a, ([], [])) &= (a, ([], [])) \\ f(a, (l, (h : t))) &= (h, (l, t ++ [a])) \\ g &= (\text{id} \times \widehat{\text{zip}}) \end{aligned}$$

Alinea 5

Para a definir o quick sort de *B_tree*, faltava definir o gene do anamorfismo. Com base nas dicas dadas pelo professor, chegamos à conclusão que a função tem 3 casos, o de lista vazia, o de lista singular ou lista com mais de 1 elemento. Com o uso da função *filter*, conseguimos a definição de *lsplitB_tree*, que

analisa os dois elementos à cabeça da lista, e filtra os restantes elementos da lista para as respetivas posições.

```

lsplitB_tree [] = i1 ()
lsplitB_tree [h] = i2 ([], [(h, [])])
lsplitB_tree (h1 : h2 : t) = i2 ((l, [(a, tMin), (b, tMax)]))
  where
    (a, b) = (min, max) (h1, h2)
    (tMin, tMax) = (filter ((< a) · (> b)), filter (> b)) t
    l = filter (< a) t
qSortB_tree :: Ord t => [t] -> [t]
qSortB_tree = inordB_tree · anaB_tree lsplitB_tree

```

Alinea 6

Para a definição de dotB_tree, verificamos como era feito para as BTrees, sendo o funcionamento semelhante.

```

dotB_tree :: Show a => B-tree a -> IO ExitCode
dotB_tree = dotpict · bmap nothing (Just · show) · cB_tree2Exp
cB_tree2Exp = cataB_tree [(Var "nil"), f · (id × unzip)]
  where
    f = Term · (((id × cons) · assoc · (swap × id) · assocl))
    -- f (a,(b,c)) = Term b (a:c)

```

Quickcheck

Fizemos ainda alguns testes para verificar se as funções que definimos estavam correctas.

```

checkList xs = and $ zipWith (≤) xs $ tail xs
prop_ord l =
  checkList x ≡ True
  where x = qSortB_tree l
prop_mirror l =
  checkList (invl ((inordB_tree · mirrorB_tree · (anaB_tree lsplitB_tree)) l)) ≡ True

```

Problema 4

Alinea 1

Tendo em conta as definições dos catamorfismos dadas e dos funtores, definir os anamorfismos correspondentes é uma tarefa bastante simples:

```

recA g h = baseA id g h
baseA f g h = f + g × h
[[· ·]]_A :: (a -> 1 + a × d) -> (d -> 1 + a) -> a -> A
[[ga gb]]_A = inA · (recA [[ga gb]]_A [[ga gb]]_B) · ga
recB f = baseB id f
baseB g f = g + f
[[· ·]]_B :: (c -> 1 + c × d) -> (d -> 1 + c) -> d -> B
[[ga gb]]_B = inB · (recB [[ga gb]]_A) · gb

```


Alinea 2

Para resolvermos a `generateAlgae` construímos os dois diagramas:

$$\begin{array}{ccc}
 Int & \xrightarrow{(id + \langle id, id \rangle) \cdot outNat} & 1 + Int \times Int \\
 \downarrow \llbracket ga \ gb \rrbracket_A & & \downarrow id + \llbracket ga \ gb \rrbracket_A \times \llbracket ga \ gb \rrbracket_B \\
 A & \xleftarrow{inA} & 1 + A \times B
 \end{array}$$

$$\begin{array}{ccc}
 Int & \xrightarrow{outNat} & 1 + Int \\
 \downarrow \llbracket ga \ gb \rrbracket_B & & \downarrow id + \llbracket ga \ gb \rrbracket_A \\
 B & \xleftarrow{inB} & 1 + A
 \end{array}$$

Daqui inferimos que para o `geneB` seria apenas necessário fazer o `out` dos naturais, e chegaríamos ao tipo esperado. Para o `geneA`, tínhamos de criar um par de inteiros no lado direito da alternativa, portanto após o `out`, é feito um `split` de identidades chegando ao tipo desejado.

$$generateAlgae = \llbracket (id + \langle id, id \rangle) \cdot outNat \ outNat \rrbracket_A$$

Para resolvermos a `showAlgae` construímos os dois diagramas:

$$\begin{array}{ccc}
 A & \xleftarrow{inA} & 1 + A \times B \\
 \downarrow \llbracket ga \ gb \rrbracket_A & & \downarrow id + \llbracket ga \ gb \rrbracket_A \times \llbracket ga \ gb \rrbracket_B \\
 S & \xleftarrow{["A", conc]} & 1 + S \times S
 \end{array}$$

$$\begin{array}{ccc}
 B & \xleftarrow{inB} & 1 + B \\
 \downarrow \llbracket ga \ gb \rrbracket_B & & \downarrow id + \llbracket ga \ gb \rrbracket_A \\
 S & \xleftarrow{["B", id]} & 1 + S
 \end{array}$$

Com base nos diagramas, verificamos que ambos os genes seriam alternativas. No lado esquerdo da alternativa, o `geneA` aplica a função constante à string "A" e o `geneB` aplica a mesma função à string "B". No lado direito da alternativa, o `geneA` faz `(++)` uncurried, o `geneB` apenas aplica a identidade.

$$showAlgae = \llbracket ["A", conc] \ ["B", id] \rrbracket_A$$

Alinea 3

```

genPos :: Gen Int
genPos = do
  n <- Test.QuickCheck.choose (0,20)
  return (n)

myfib :: Int -> Int
myfib = hyloLTree [1, (+)] fibd

prop_LSystems n =
  (length . showAlgae . generateAlgae) n ≡ (myfib . succ) n

testLSystems = quickCheck $ forAll genPos $ \n -> prop_LSystems n

```

Problema 5

Alinea 1

Para resolver a alinea 1, o grupo seguiu a secção 4.10 dos apontamentos, chegando em primeiro lugar a uma definição da função `permuta`:

```

permuta [] = id []
permuta l = let ((h, t) = getR l) (res = permuta t) in id (h : res)

```

“Monidificando” o código, obtemos a função permuta.

```

permuta [] = return []
permuta l = do {
  (h, t) ← getR l;
  res ← permuta t;
  return (h : res)
}

```

Alinea 2

Para resolver a alinea 2, o grupo construiu o seguinte diagrama de catamorfismo:

$$\begin{array}{ccc}
 \text{LTree } Equipa & \xleftarrow{\text{inLTree}} & Equipa + \text{LTree } Equipa \times \text{LTree } Equipa \\
 \downarrow \text{cataLTree } g & & \downarrow \text{id} + (\text{cataLTree } g) \\
 Equipa & \xleftarrow{g} & Equipa + Equipa \times Equipa
 \end{array}$$

Após o diagrama, verificamos que o gene será [id,jogo]. De seguida, passamos a função eliminatória a pointwise:

```

eliminatória (Leaf a) = a
eliminatória (Fork (e, d)) = jogo (eliminatória e, eliminatória d)

<=> { Inserção de let, (1) }

eliminatória (Leaf a) = id a
eliminatória (Fork (e, d)) = let (x, y) = (eliminatória e, eliminatória d) in jogo (x, y)

```

Desta forma, rapidamente inserimos return no lugar de id e trocamos o let por do, chegando à definição final de eliminatória.

```

eliminatória (Leaf a) = return a
eliminatória (Fork (e, d)) = do {
  x ← eliminatória e;
  y ← eliminatória d;
  (jogo (x, y))
}

```

QuickCheck

```

prop_jog e =
  abs (sumP (unD (quem_vence equipas)) - 1) < e
testJog e = quickCheck (prop_jog e)

```

Index

- LaTeX, 2
 - lhs2TeX, 2
- B-tree, 4
- Cálculo de Programas, 3
 - Material Pedagógico, 2
 - BTree.hs, 4, 5
 - Exp.hs, 5
 - LTree.hs, 8, 9
- Combinador “pointfree”
 - cata*, 7, 17
 - either*, 7, 12, 14–17
- Função
 - π_1 , 12–14
 - π_2 , 11–15
 - length*, 7, 11, 15, 17
 - map*, 11, 14, 15
 - succ*, 7, 12–14, 17
 - uncurry*, 7, 12–17
- Functor, 3, 5, 7–11, 16
- Graphviz, 5, 6
 - WebGraphviz, 6
- Haskell, 2, 3
 - “Literate Haskell”, 2
 - Biblioteca
 - PFP, 10
 - Probability, 8, 10
 - interpretador
 - GHCi, 3, 10
 - QuickCheck, 3, 4, 7
- L-system, 6, 7
- Programação literária, 2
- Taylor series
 - Maclaurin series, 3
- U.Minho
 - Departamento de Informática, 1
- Unix shell
 - wc*, 4
- Utilitário
 - LaTeX
 - bibtex*, 3
 - makeindex*, 3