

# Cálculo de Programas

## Trabalho Prático

### MiEI+LCC — 2019/20

Departamento de Informática  
Universidade do Minho

Junho de 2020

**Grupo nr. 6**

---

a89588	João Pedro da Santa Guedes
a89601	Luís Pedro Oliveira de Castro Vieira
a89529	Pedro Miguel Soveral Pachebo Barbosa

## 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 à programação funcional em **Haskell**. Assim, o presente trabalho prático coloca os alunos perante problemas concretos que deverão ser implementados em **Haskell**. Há ainda um outro objectivo: o de ensinar a documentar programas, validá-los, 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*” [2], cujo princípio base é o seguinte:

*Um programa e a sua documentação devem coincidir.*

Por outras palavras, o código fonte e a documentação de um programa deverão estar no mesmo ficheiro.

O ficheiro `cp1920t.pdf` que está a ler é já um exemplo de **programação literária**: foi gerado a partir do texto fonte `cp1920t.lhs`<sup>1</sup> que encontrará no **material pedagógico** desta disciplina descompactando o ficheiro `cp1920t.zip` e executando

```
$ lhs2TeX cp1920t.lhs > cp1920t.tex
$ pdflatex cp1920t
```

em que **lhs2tex** é um pre-processor que faz “pretty printing” de código Haskell em **L<sup>A</sup>T<sub>E</sub>X** e que deve desde já instalar executando

```
$ cabal install lhs2tex
```

Por outro lado, o mesmo ficheiro `cp1920t.lhs` é executável e contém o “kit” básico, escrito em **Haskell**, para realizar o trabalho. Basta executar

```
$ ghci cp1920t.lhs
```

---

<sup>1</sup>O suffixo ‘lhs’ quer dizer *literate Haskell*.

Abra o ficheiro `cp1920t.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 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 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 cp1920t.aux
$ makeindex cp1920t.idx
```

e recompilar o texto como acima se indicou. Dever-se-á ainda instalar o utilitário **QuickCheck**, que ajuda a validar programas em **Haskell** e a biblioteca **Gloss** para geração de gráficos 2D:

```
$ cabal install QuickCheck gloss
```

Para testar uma propriedade **QuickCheck** *prop*, basta invocá-la com o comando:

```
> quickCheck prop
+++ OK, passed 100 tests.
```

Pode mesmo controlar o número de casos de teste e sua complexidade utilizando o comando:

```
> quickCheckWith stdArgs { maxSuccess = 200, maxSize = 10 } prop
+++ OK, passed 200 tests.
```

Qualquer programador tem, na vida real, de ler e analisar (muito!) código escrito por outros. No anexo **B** disponibiliza-se algum código **Haskell** relativo aos problemas que se seguem. Esse anexo deverá ser consultado e analisado à medida que isso for necessário.

### Problema 1

Pretende-se implementar um sistema de manutenção e utilização de um dicionário. Este terá uma estrutura muito peculiar em memória. Será construída uma árvore em que cada nodo terá apenas uma letra da palavra e cada folha da respectiva árvore terá a respectiva tradução (um ou mais sinónimos). Deverá ser possível:

- *dic\_rd* — procurar traduções para uma determinada palavra
- *dic\_in* — inserir palavras novas (palavra e tradução)
- *dic\_imp* — importar dicionários do formato “lista de pares palavra-tradução”
- *dic\_exp* — exportar dicionários para o formato “lista de pares palavra-tradução”.

A implementação deve ser baseada no módulo **Exp.hs** que está incluído no material deste trabalho prático, que deve ser estudado com atenção antes de abordar este problema.

No anexo **B** é dado um dicionário para testes, que corresponde à figura **1**. A implementação proposta deverá garantir as seguintes propriedades:



Figura 1: Representação em memória do dicionário dado para testes.

**Propriedade [QuickCheck] 1** Se um dicionário estiver normalizado (ver apêndice B) então não perdemos informação quando o representamos em memória:

$$\text{prop\_dic\_rep } x = \text{let } d = \text{dic\_norm } x \text{ in } (\text{dic\_exp} \cdot \text{dic\_imp}) d \equiv d$$

**Propriedade [QuickCheck] 2** Se um significado  $s$  de uma palavra  $p$  já existe num dicionário então adicioná-lo em memória não altera nada:

$$\begin{aligned} \text{prop\_dic\_red } p \ s \ d \\ | \text{ dic\_red } p \ s \ d = \text{dic\_imp } d \equiv \text{dic\_in } p \ s \ (\text{dic\_imp } d) \\ | \text{ otherwise} = \text{True} \end{aligned}$$

**Propriedade [QuickCheck] 3** A operação  $\text{dic\_rd}$  implementa a procura na correspondente exportação do dicionário:

$$\text{prop\_dic\_rd } (p, t) = \text{dic\_rd } p \ t \equiv \text{lookup } p \ (\text{dic\_exp } t)$$

## Problema 2

Árvores binárias (elementos do tipo **BTree**) são frequentemente usadas no armazenamento e procura de dados, porque suportam um vasto conjunto de ferramentas para procuras eficientes. Um exemplo de destaque é o caso das **árvores binárias de procura**, *i.e.* árvores que seguem o princípio de *ordenação*: para todos os nós, o filho à esquerda tem um valor menor ou igual que o valor no próprio nó; e de forma análoga, o filho à direita tem um valor maior ou igual que o valor no próprio nó. A Figura 2 apresenta dois exemplos de árvores binárias de procura.<sup>2</sup>

Note que tais árvores permitem reduzir *significativamente* o espaço de procura, dado que ao procurar um valor podemos sempre *reduzir a procura a um ramo* ao longo de cada nó visitado. Por exemplo, ao procurar o valor 7 na primeira árvore ( $t_1$ ), sabemos que nos podemos restringir ao ramo da direita do nó com o valor 5 e assim sucessivamente. Como complemento a esta explicação, consulte também os **vídeos das aulas teóricas** (capítulo ‘pesquisa binária’).

Para verificar se uma árvore binária está ordenada, é útil ter em conta a seguinte propriedade: considere uma árvore binária cuja raiz tem o valor  $a$ , um filho  $s_1$  à esquerda e um filho  $s_2$  à direita. Assuma

<sup>2</sup>As imagens foram geradas com recurso à função *dotBt* (disponível neste documento). Recomenda-se o uso desta função para efeitos de teste e ilustração.



Figura 2: Duas árvores binárias de procura; a da esquerda vai ser designada por  $t_1$  e a da direita por  $t_2$ .

que os dois filhos estão ordenados; que o elemento *mais à direita* de  $t_1$  é menor ou igual a  $a$ ; e que o elemento *mais à esquerda* de  $t_2$  é maior ou igual a  $a$ . Então a árvore binária está ordenada. Dada esta informação, implemente as seguintes funções como catamorfismos de árvores binárias.

$\text{maisEsq} :: \text{BTree } a \rightarrow \text{Maybe } a$   
 $\text{maisDir} :: \text{BTree } a \rightarrow \text{Maybe } a$

Seguem alguns exemplos dos resultados que se esperam ao aplicar estas funções à árvore da esquerda ( $t_1$ ) e à árvore da direita ( $t_2$ ) da Figura 2.

```
*Splay> maisDir t1
Just 16
*Splay> maisEsq t1
Just 1
*Splay> maisDir t2
Just 8
*Splay> maisEsq t2
Just 0
```

**Propriedade [QuickCheck] 4** As funções  $\text{maisEsq}$  e  $\text{maisDir}$  são determinadas unicamente pela propriedade

$\text{prop\_inv} :: \text{BTree } \text{String} \rightarrow \text{Bool}$   
 $\text{prop\_inv} = \text{maisEsq} \equiv \text{maisDir} \cdot \text{invBTree}$

**Propriedade [QuickCheck] 5** O elemento *mais à esquerda* de uma árvore está presente no ramo da esquerda, a não ser que esse ramo esteja vazio:

$\text{propEsq } \text{Empty} = \text{property } \text{Discard}$   
 $\text{propEsq } x@(Node(a, (t, s))) = (\text{maisEsq } t) \neq \text{Nothing} \Rightarrow (\text{maisEsq } x) \equiv \text{maisEsq } t$

A próxima tarefa deste problema consiste na implementação de uma função que insere um novo elemento numa árvore binária *preservando* o princípio de ordenação,

$\text{insOrd} :: (\text{Ord } a) \Rightarrow a \rightarrow \text{BTree } a \rightarrow \text{BTree } a$

e de uma função que verifica se uma dada árvore binária está ordenada,

$\text{isOrd} :: (\text{Ord } a) \Rightarrow \text{BTree } a \rightarrow \text{Bool}$

Para ambas as funções deve utilizar o que aprendeu sobre *catamorfismos e recursividade mútua*.

**Sugestão:** Se tiver problemas em implementar com base em catamorfismos estas duas últimas funções, tente implementar (com base em catamorfismos) as funções auxiliares

$\text{insOrd}' :: (\text{Ord } a) \Rightarrow a \rightarrow \text{BTree } a \rightarrow (\text{BTree } a, \text{BTree } a)$   
 $\text{isOrd}' :: (\text{Ord } a) \Rightarrow \text{BTree } a \rightarrow (\text{Bool}, \text{BTree } a)$

tais que  $\text{insOrd}' x = \langle \text{insOrd } x, \text{id} \rangle$  para todo o elemento  $x$  do tipo  $a$  e  $\text{isOrd}' = \langle \text{isOrd}, \text{id} \rangle$ .



Figura 3: Exemplo de uma rotação à direita. A árvore da esquerda é a árvore original; a árvore da direita representa a rotação à direita correspondente.



Figura 4: Exemplo de uma rotação à direita. A árvore da esquerda é a árvore original; a árvore da direita representa a rotação à direita correspondente.

**Propriedade [QuickCheck] 6** Inserir uma sucessão de elementos numa árvore vazia gera uma árvore ordenada.

$prop\_ord :: [Int] \rightarrow Bool$   
 $prop\_ord = isOrd \cdot (foldr insOrd Empty)$

As árvores binárias providenciam uma boa maneira de reduzir o espaço de procura. Mas podemos fazer ainda melhor: podemos aproximar da raiz os elementos da árvore que são mais acedidos, reduzindo assim o espaço de procura na *dimensão vertical*<sup>3</sup>. Esta operação é geralmente referida como *splaying* e é implementada com base naquilo a que chamamos *rotações à esquerda e à direita de uma árvore*.

Intuitivamente, a rotação à direita de uma árvore move todos os nós “uma casa para a sua direita”. Formalmente, esta operação define-se da seguinte maneira:

1. Considere uma árvore binária e designe a sua raiz pela letra  $r$ . Se  $r$  não tem filhos à esquerda então simplesmente retornamos a árvore dada à entrada. Caso contrário,
2. designe o filho à esquerda pela letra  $l$ . A árvore que vamos retornar tem  $l$  na raiz, que mantém o filho à esquerda e adota  $r$  como o filho à direita. O orfão (*i.e.* o anterior filho à direita de  $l$ ) passa a ser o filho à esquerda de  $r$ .

A rotação à esquerda é definida de forma análoga. As Figuras 3 e 4 apresentam dois exemplos de rotações à direita. Note que em ambos os casos o valor 2 subiu um nível na árvore correspondente. De facto, podemos sempre aplicar uma *sequência* de rotações numa árvore de forma a mover um dado nó para a raiz (dando origem portanto à referida operação de splaying).

Comece então por implementar as funções

<sup>3</sup>Note que nas árvores de binária de procura a redução é feita na dimensão horizontal.

```

rrot :: BTree a → BTree a
lrot :: BTree a → BTree a

```

de rotação à direita e à esquerda.

**Propriedade [QuickCheck] 7** As rotações à esquerda e à direita preservam a ordenação das árvores.

```

prop_ord_pres_esq = forAll orderedBTree (isOrd · lrot)
prop_ord_pres_dir = forAll orderedBTree (isOrd · rrot)

```

De seguida implemente a operação de splaying

```

splay :: [Bool] → (BTree a → BTree a)

```

como um catamorfismo de listas. O argumento `[Bool]` representa um caminho ao longo de uma árvore, em que o valor `True` representa "seguir pelo ramo da esquerda" e o valor `False` representa "seguir pelo ramo da direita". O caminho ao longo de uma árvore serve para *identificar* unicamente um nó dessa árvore.

**Propriedade [QuickCheck] 8** A operação de splay preserva a ordenação de uma árvore.

```

prop_ord_pres_splay :: [Bool] → Property
prop_ord_pres_splay path = forAll orderedBTree (isOrd · (splay path))

```

### Problema 3

Árvores de decisão binárias são estruturas de dados usadas na área de **machine learning** para codificar processos de decisão. Geralmente, tais árvores são geradas por computadores com base num vasto conjunto de dados e reflectem o que o computador "aprendeu" ao processar esses mesmos dados. Segue-se um exemplo muito simples de uma árvore de decisão binária:



Esta árvore representa o processo de decisão relativo a ser preciso ou não levar um guarda-chuva para uma viagem, dependendo das condições climáticas. Essencialmente, o processo de decisão é efectuado ao "percorrer" a árvore, escolhendo o ramo da esquerda ou da direita de acordo com a resposta à pergunta correspondente. Por exemplo, começando da raiz da árvore, responder `["não", "não"]` leva-nos à decisão "não precisa" e responder `["não", "sim"]` leva-nos à decisão "precisa".

Árvores de decisão binárias podem ser codificadas em **Haskell** usando o seguinte tipo de dados:

```

data Bdt a = Dec a | Query (String, (Bdt a, Bdt a)) deriving Show

```

Note que o tipo de dados `Bdt` é parametrizado por um tipo de dados `a`. Isto é necessário, porque as decisões podem ser de diferentes tipos: por exemplo, respostas do tipo "sim ou não" (como apresentado acima), a escolha de números, ou **classificações**.

De forma a conseguirmos processar árvores de decisão binárias em **Haskell**, deve, antes de tudo, resolver as seguintes alíneas:

1. Definir as funções `inBdt`, `outBdt`, `baseBdt`, `cataBdt`, e `anaBdt`.
2. Apresentar no relatório o diagrama de `anaBdt`.

Para tomar uma decisão com base numa árvore de decisão binária  $t$ , o computador precisa apenas da estrutura de  $t$  (*i.e.* pode esquecer a informação nos nós da árvore) e de uma lista de respostas “sim ou não” (para que possa percorrer a árvore da forma desejada). Implemente então as seguintes funções na forma de *catamorfismos*:

1.  $extLTree : Bdt\ a \rightarrow LTree\ a$  (esquece a informação presente nos nós de uma dada árvore de decisão binária).

**Propriedade [QuickCheck] 9** A função  $extLTree$  preserva as folhas da árvore de origem.

$$\begin{aligned} prop\_pres\_tips &:: Bdt\ Int \rightarrow Bool \\ prop\_pres\_tips &= tipsBdt \equiv tipsLTree \cdot extLTree \end{aligned}$$

2.  $navLTree : LTree\ a \rightarrow ([Bool] \rightarrow LTree\ a)$  (navega um elemento de  $LTree$  de acordo com uma sequência de respostas “sim ou não”. Esta função deve ser implementada como um catamorfismo de  $LTree$ . Neste contexto, elementos de  $[Bool]$  representam sequências de respostas: o valor  $True$  corresponde a “sim” e portanto a “segue pelo ramo da esquerda”; o valor  $False$  corresponde a “não” e portanto a “segue pelo ramo da direita”.

Seguem alguns exemplos dos resultados que se esperam ao aplicar  $navLTree$  a  $(extLTree\ bdtGC)$ , em que  $bdtGC$  é a árvore de decisão binária acima descrita, e a uma sequência de respostas.

```
*ML> navLTree (extLTree bdtGC) []
Fork (Leaf "Precisa",Fork (Leaf "Precisa",Leaf "N precisa"))
*ML> navLTree (extLTree bdtGC) [False]
Fork (Leaf "Precisa",Leaf "N precisa")
*ML> navLTree (extLTree bdtGC) [False,True]
Leaf "Precisa"
*ML> navLTree (extLTree bdtGC) [False,True,True]
Leaf "Precisa"
*ML> navLTree (extLTree bdtGC) [False,True,True,True]
Leaf "Precisa"
```

**Propriedade [QuickCheck] 10** Percorrer uma árvore ao longo de um caminho é equivalente a percorrer a árvore inversa ao longo do caminho inverso.

$$\begin{aligned} prop\_inv\_nav &:: Bdt\ Int \rightarrow [Bool] \rightarrow Bool \\ prop\_inv\_nav\ t\ l &= \mathbf{let}\ t' = extLTree\ t\ \mathbf{in} \\ &\quad invLTree\ (navLTree\ t'\ l) \equiv navLTree\ (invLTree\ t')\ (fmap\ \neg\ l) \end{aligned}$$

**Propriedade [QuickCheck] 11** Quanto mais longo for o caminho menos alternativas de fim irão existir.

$$\begin{aligned} prop\_af &:: Bdt\ Int \rightarrow ([Bool],[Bool]) \rightarrow Property \\ prop\_af\ t\ (l1,l2) &= \mathbf{let}\ t' = extLTree\ t \\ &\quad f = \mathbf{length} \cdot tipsLTree \cdot (navLTree\ t') \\ &\quad \mathbf{in}\ isPrefixOf\ l1\ l2 \Rightarrow (f\ l1 \geq f\ l2) \end{aligned}$$

## Problema 4

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

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

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.<sup>4</sup> `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  $g : A \rightarrow \text{Dist } B$  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 da programação monádica. Vamos estudar a aplicação deste mónade ao exercício anterior, tendo em conta o facto de que nem sempre podemos responder com 100% de certeza a perguntas presentes em árvores de decisão.

Considere a seguinte situação: a Anita vai trabalhar no dia seguinte e quer saber se precisa de levar guarda-chuva. Na verdade, ela tem autocarro de porta de casa até ao trabalho, e portanto as condições meteorológicas não são muito significativas; a não ser que seja segunda-feira... Às segundas é dia de feira e o autocarro vai sempre lotado! Nesses dias, ela prefere fazer a pé o caminho de casa ao trabalho, o que a obriga a levar guarda-chuva (nos dias de chuva). Abaixo está apresentada a árvore de decisão

<sup>4</sup>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].



respectiva a este problema.



Assuma que a Anita não sabe em que dia está, e que a previsão da chuva para a ida é de 80% enquanto que a previsão de chuva para o regresso é de 60%. *A Anita deve levar guarda-chuva?* Para responder a esta questão, iremos tirar partido do que se aprendeu no exercício anterior. De facto, a maior diferença é que agora as respostas ("sim" ou "não") são dadas na forma de uma distribuição sobre o tipo de dados *Bool*. Implemente como um catamorfismo de *LTree* a função

$$bnavLTree :: LTree\ a \rightarrow ((BTree\ Bool) \rightarrow LTree\ a)$$

que percorre uma árvore dado um caminho, *não* do tipo  $[Bool]$ , mas do tipo  $BTree\ Bool$ . O tipo  $BTree\ Bool$  é necessário na presença de incerteza, porque neste contexto não sabemos sempre qual a próxima pergunta a responder. Teremos portanto que ter resposta para todas as perguntas na árvore de decisão.

Seguem alguns exemplos dos resultados que se esperam ao aplicar *bnavLTree* a  $(extLTree\ anita)$ , em que *anita* é a árvore de decisão acima descrita, e a uma árvore binária de respostas.

```

*ML> bnavLTree (extLTree anita) (Node(True, (Empty,Empty)))
Fork (Leaf "Precisa",Fork (Leaf "Precisa",Leaf "N precisa"))
*ML> bnavLTree (extLTree anita) (Node(True, (Node(True, (Empty,Empty)),Empty)))
Leaf "Precisa"
*ML> bnavLTree (extLTree anita) (Node(False, (Empty,Empty)))
Leaf "N precisa"

```

Por fim, implemente como um catamorfismo de *LTree* a função

$$pbnvLTree :: LTree\ a \rightarrow ((BTree\ (Dist\ Bool)) \rightarrow Dist\ (LTree\ a))$$

que deverá consistir na "monadificação" da função *bnavLTree* via a mónade das probabilidades. Use esta última implementação para responder se a Anita deve levar guarda-chuva ou não dada a situação acima descrita.

## Problema 5

Os **mosaicos de Truchet** são padrões que se obtêm gerando aleatoriamente combinações bidimensionais de ladrilhos básicos. Os que se mostram na figura 5 são conhecidos por ladrilhos de Truchet-Smith. A figura 7 mostra um exemplo de mosaico produzido por uma combinação aleatória de 10x10 ladrilhos *a* e *b* (cf. figura 5).

Neste problema pretende-se programar a geração aleatória de mosaicos de Truchet-Smith usando o mónade **Random** e a biblioteca **Gloss** para produção do resultado. Para uniformização das respostas, deverão ser seguidas as seguintes condições:

- Cada ladrilho deverá ter as dimensões 80x80
- O programa deverá gerar mosaicos de quaisquer dimensões, mas deverá ser apresentado como figura no relatório o mosaico de 10x10 ladrilhos.
- Valorizar-se-ão respostas elegantes e com menos linhas de código **Haskell**.

No anexo B é dada uma implementação da operação de permuta aleatória de uma lista que pode ser útil para resolver este exercício.



Figura 5: Os dois ladrilhos de Truchet-Smith.



Figura 6: Um mosaico de Truchet-Smith.

# Anexos

## A Como exprimir cálculos e diagramas em LaTeX/lhs2tex

Estudar o texto fonte deste trabalho para obter o efeito:<sup>5</sup>

$$\begin{aligned}
 id &= \langle f, g \rangle \\
 &\equiv \{ \text{universal property} \} \\
 &\quad \left\{ \begin{array}{l} \pi_1 \cdot id = f \\ \pi_2 \cdot id = g \end{array} \right. \\
 &\equiv \{ \text{identity} \} \\
 &\quad \left\{ \begin{array}{l} \pi_1 = f \\ \pi_2 = g \end{array} \right. \\
 &\square
 \end{aligned}$$

Os diagramas podem ser produzidos recorrendo à *package* L<sup>A</sup>T<sub>E</sub>X *xymatrix*, por exemplo:

$$\begin{array}{ccc}
 \mathbb{N}_0 & \xleftarrow{\text{in}} & 1 + \mathbb{N}_0 \\
 \downarrow \langle g \rangle & & \downarrow id + \langle g \rangle \\
 B & \xleftarrow{g} & 1 + B
 \end{array}$$

## B Código fornecido

### Problema 1

Função de representação de um dicionário:

```
dic_imp :: [(String, [String])] -> Dict
dic_imp = Term " " · map (bmap id singl) · untar · discollect
```

onde

```
type Dict = Exp String String
```

Dicionário para testes:

```
d :: [(String, [String])]
d = [("ABA", ["BRIM"]),
      ("ABALO", ["SHOCK"]),
      ("AMIGO", ["FRIEND"]),
      ("AMOR", ["LOVE"]),
      ("MEDO", ["FEAR"]),
      ("MUDO", ["DUMB", "MUTE"]),
      ("PE", ["FOOT"]),
      ("PEDRA", ["STONE"]),
      ("POBRE", ["POOR"]),
      ("PODRE", ["ROTTEN"])]
```

Normalização de um dicionário (remoção de entradas vazias):

```
dic_norm = collect · filter p · discollect where
  p (a, b) = a > " " ∧ b > " "
```

Teste de redundância de um significado *s* para uma palavra *p*:

```
dic_red p s d = (p, s) ∈ discollect d
```

---

<sup>5</sup>Exemplos tirados de [3].

## Problema 2

Árvores usadas no texto:

```
emp x = Node (x, (Empty, Empty))
t7 = emp 7
t16 = emp 16
t7_10_16 = Node (10, (t7, t16))
t1_2_nil = Node (2, (emp 1, Empty))
t' = Node (5, (t1_2_nil, t7_10_16))
t0_2_1 = Node (2, (emp 0, emp 3))
t5_6_8 = Node (6, (emp 5, emp 8))
t2 = Node (4, (t0_2_1, t5_6_8))
dotBt :: (Show a) => BTree a -> IO ExitCode
dotBt = dotpict · bmap Just Just · cBTree2Exp · (fmap show)
```

## Problema 3

Funções usadas para efeitos de teste:

```
tipsBdt :: Bdt a -> [a]
tipsBdt = cataBdt [singl, ( $\widehat{++}$ ) ·  $\pi_2$ ]
tipsLTree = tips
```

## Problema 5

Função de permutação aleatória de uma lista:

```
permuta [] = return []
permuta x = do { (h, t) ← getR x; t' ← permuta t; return (h : t') } where
  getR x = do { i ← getStdRandom (randomR (0, length x - 1)); return (x !! i, retira i x) }
  retira i x = take i x ++ drop (i + 1) x
```

## QuickCheck

Código para geração de testes:

```
instance Arbitrary a => Arbitrary (BTree a) where
  arbitrary = sized genbt where
    genbt 0 = return (inBTree $ i_1 ())
    genbt n = oneof [(liftM2 $ curry (inBTree · i_2))
      QuickCheck.arbitrary (liftM2 (,) (genbt (n - 1)) (genbt (n - 1))),
      (liftM2 $ curry (inBTree · i_2))
      QuickCheck.arbitrary (liftM2 (,) (genbt (n - 1)) (genbt 0)),
      (liftM2 $ curry (inBTree · i_2))
      QuickCheck.arbitrary (liftM2 (,) (genbt 0) (genbt (n - 1)))]
instance (Arbitrary v, Arbitrary o) => Arbitrary (Exp v o) where
  arbitrary = (genExp 10) where
    genExp 0 = liftM (inExp · i_1) QuickCheck.arbitrary
    genExp n = oneof [liftM (inExp · i_2 · ( $\lambda a \rightarrow (a, [])$ )) QuickCheck.arbitrary,
      liftM (inExp · i_1) QuickCheck.arbitrary,
      liftM (inExp · i_2 · ( $\lambda (a, (b, c)) \rightarrow (a, [b, c])$ ))
      $ (liftM2 (,) QuickCheck.arbitrary (liftM2 (,)
        (genExp (n - 1)) (genExp (n - 1)))),
      liftM (inExp · i_2 · ( $\lambda (a, (b, c, d)) \rightarrow (a, [b, c, d])$ ))
      $ (liftM2 (,) QuickCheck.arbitrary (liftM3 (,,)
```

```

    (genExp (n - 1)) (genExp (n - 1)) (genExp (n - 1)))
  ]
orderedBTree :: Gen (BTree Int)
orderedBTree = liftM (foldr insOrd Empty) (QuickCheck.arbitrary :: Gen [Int])
instance (Arbitrary a) => Arbitrary (Bdt a) where
  arbitrary = sized genbt where
    genbt 0 = liftM Dec QuickCheck.arbitrary
    genbt n = oneof [(liftM2 $ curry Query)
      QuickCheck.arbitrary (liftM2 (,) (genbt (n - 1)) (genbt (n - 1))),
      (liftM2 $ curry (Query))
      QuickCheck.arbitrary (liftM2 (,) (genbt (n - 1)) (genbt 0)),
      (liftM2 $ curry (Query))
      QuickCheck.arbitrary (liftM2 (,) (genbt 0) (genbt (n - 1)))]

```

## Outras funções auxiliares

Lógicas:

```

infixr 0 =>
  (=>) :: (Testable prop) => (a -> Bool) -> (a -> prop) -> a -> Property
  p => f = λa -> p a => f a
infixr 0 ⇔
  (⇔) :: (a -> Bool) -> (a -> Bool) -> a -> Property
  p ⇔ f = λa -> (p a => property (f a)) .&&. (f a => property (p a))
infixr 4 ≡
  (≡) :: Eq b => (a -> b) -> (a -> b) -> (a -> Bool)
  f ≡ g = λa -> f a ≡ g a
infixr 4 ≤
  (≤) :: Ord b => (a -> b) -> (a -> b) -> (a -> Bool)
  f ≤ g = λa -> f a ≤ g a
infixr 4 ∧
  (∧) :: (a -> Bool) -> (a -> Bool) -> (a -> Bool)
  f ∧ g = λa -> (f a) ∧ (g a)

```

Compilação e execução dentro do interpretador:<sup>6</sup>

```
run = do { system "ghc cp1920t"; system "./cp1920t" }
```

## C Soluções dos alunos

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 ou tipos das funções dadas, mas pode ser adicionado texto e/ou outras funções auxiliares que sejam necessárias.

### Problema 1

```

discollect :: (Ord b, Ord a) => [(b, [a])] -> [(b, a)]
discollect = set . (lstr .! id)

```

A função dic-exp tem a funcionalidade de transformar um dicionário numa lista de pares palavra, tradução. Deste modo fazemos um catamorfismo Exp, no qual temos um either que ou conserva o nosso Var ou então junta as letras dos diferentes nodos do dicionário que irão formar a respetiva palavra seguida da sua tradução. Por fim, como obtivemos uma lista de listas e apenas pretendíamos ter uma lista recorremos à função concat.

<sup>6</sup>Pode ser útil em testes envolvendo [Gloss](#). Nesse caso, o teste em causa deve fazer parte de uma função *main*.

```

dic_exp :: Dict → [(String, [String])]
dic_exp = collect · tar
tar = cataExp g where
  g = [conserva, junta]
  conserva x = [("", x)]
  junta (o, l) = map (auxJunta) (concat l) where
    auxJunta (x, y) = ((o ++ x), y)

```

A função `dic-rd` tem a funcionalidade de procurar traduções para uma dada palavra. Deste modo começamos por lhe aplicar a função `dic-exp` para passarmos a ter uma lista de pares palavra-tradução e de seguida aplicamos-lhe um catamorfismo de listas. Este catamorfismo é composto por um `either` `Nothing` ou uma função auxiliar que tem como objetivo comparar a palavra da qual pretendemos obter a tradução com as palavras do dicionário. Se encontrar essa palavra no dicionário retorna a sua tradução

```

dic_rd d = (cataList g) · dic_exp where
  g = [Nothing, auxDicRD] where
    auxDicRD ((p, tp), t) | d ≡ p = Just tp
    | otherwise = t
dic_in = ⊥

```

## Problema 2

```

auxMaisDir (h, (l, Nothing)) = Just h
auxMaisDir (−, (−, r)) = r
maisDir = cataBTree g
where g = [Nothing, auxMaisDir]

```

```

auxMaisEsq (h, (Nothing, r)) = Just h
auxMaisEsq (−, (l, −)) = l
maisEsq = cataBTree g
where g = [Nothing, auxMaisEsq]

```

```

auxInsOrd x (h, ((ll1, lr1), (rl1, rr1))) | x > h = (Node (h, (lr1, rl1)), Node (h, (lr1, rr1)))
| otherwise = (Node (h, (ll1, rr1)), Node (h, (lr1, rr1)))
insOrd' x = cataBTree g
where g = [(Node (x, (Empty, Empty)), Empty), auxInsOrd x]
insOrd x = π1 · insOrd' x

```

```

getRoot (Node (h, (l, r))) = h
auxIsOrd (a, ((b1, Empty), (b2, Empty))) = (True, Node (a, (Empty, Empty)))
auxIsOrd (a, ((b1, Empty), (b2, a2))) | a > getRoot (a2) = (False, Node (a, (Empty, a2)))
| otherwise = (True, Node (a, (Empty, a2)))
auxIsOrd (a, ((b1, a1), (b2, Empty))) | a < getRoot (a1) = (False, Node (a, (a1, Empty)))
| otherwise = (True, Node (a, (a1, Empty)))
isOrd' = cataBTree g
where g = [(True, Empty), auxIsOrd]
isOrd = π1 · isOrd'

```

```

auxRRot (h, (Empty, r)) = Node (h, (Empty, r))
auxRRot (h, ((Node (l, (ll, lr))), r)) = Node (l, (ll, Node (h, (lr, r))))
rrot = g · outBTree
where g = [Empty, auxRRot]

```

```

auxLRot (h, (l, Empty)) = Node (h, (l, Empty))
auxLRot (h, (l, (Node (r, (rl, rr))))) = Node (r, ((Node (h, (rl, l))), rr))
lrot = g · outBTree
  where g = [Empty, auxLRot]

```

### C.0.1 splay

Para conseguirmos perceber o que realmente é pretendido com a função `splay`, devemos averiguar primeiro o tipo que ela apresenta. Assim, podemos concluir que o nosso objetivo final deve ser retornar uma função, que dada uma BTree de Bool, nos devolve uma LTree. Ainda mais, devemos compreender que somos capazes de trabalhar com a versão curried desta função, isto é, trabalharemos a lista de Bool e a BTree como se fosse um par, permitindo assim saber qual o valor à cabeça da lista, e efetuar uma rotação da árvore como dito no enunciado: se o valor for True, vamos efetuar uma rotação à direita, se for False vamos efetuar uma rotação à esquerda. O nosso objetivo é através de múltiplas rotações colocarmos um elemento no topo da raiz que nos permita fazer uma busca vertical efetiva.

```

splay l t = (flip cataBTree t g) l
  where g = [λx → Empty, curry sp]
        sp ((a, (l, r)), []) = Node (a, (l [], r []))
        sp ((a, (l, r)), t) | head t ≡ True = rrot (Node (a, (l (tail t), r [])))
                           | otherwise = lrot (Node (a, (l [], r (tail t))))

```

$$\begin{array}{ccc}
\text{BTree } A & \xrightarrow{\text{outBTree}} & 1 + (A \times (\text{BTree } A \times \text{BTree } A)) \\
\text{cataBTree} \downarrow & & \downarrow \text{id} + \text{id} \times (\text{cataBTree} \times \text{cataBTree}) \\
(\text{BTree } A \uparrow \text{Bool}^*) & \xleftarrow{g} & 1 + A \times (((\text{BTree } A) \uparrow \text{Bool}^*) \times ((\text{BTree } A) \uparrow \text{Bool}^*))
\end{array}$$

### Problema 3

A `extLTree` tem como objetivo transformar uma Bdt numa LTree. Como dito no enunciado, devemos ignorar o conteúdo no nodo da árvore e guardar somente as bifurcações no Fork da LTree e a Dec na Leaf.

```

extLTree :: Bdt a → LTree a
extLTree = cataBdt g where
  g = [Leaf, Fork · π2]

```

$$\begin{array}{ccc}
\text{Bdt } A & \xrightarrow{\text{outBdt}} & \text{Dec} + \text{String} \times (\text{Bdt } A \times \text{Bdt } A) \\
\text{extLTree} \downarrow & & \downarrow \text{id} + \text{id} \times (\text{ex} \times \text{splay}) \\
(\text{BTree } A \uparrow \text{Bool}^*) & \xleftarrow{g} & \text{Dec} + A \times (((\text{BTree } A) \uparrow \text{Bool}^*) \times ((\text{BTree } A) \uparrow \text{Bool}^*))
\end{array}$$

```

inBdt :: a + (String, (Bdt a, Bdt a)) → Bdt a
inBdt = [Dec, Query]

```

```

outBdt :: Bdt a → a + (String, (Bdt a, Bdt a))
outBdt (Dec x) = i1 (x)
outBdt (Query (s, (t1, t2))) = i2 (s, (t1, t2))

```

```

baseBdt f g = id + (f × (g × g))
recBdt g = baseBdt id g

```

$$cataBdt\ g = g \cdot (recBdt\ (cataBdt\ g)) \cdot outBdt$$

$$anaBdt\ g = inBdt \cdot (recBdt\ (anaBdt\ g)) \cdot g$$

$$\begin{array}{ccc}
Bdt\ B & \xleftarrow{inBdt} & B\ (B, Bdt\ B) \\
\uparrow anaBdt\ g & & \uparrow B\ (id, anaBdt) \\
A & \xrightarrow{g} & B\ (B, A)
\end{array}$$

### C.0.2 navLTree

Para perceber o que realmente é pretendido com a função `navLTree`, devemos averiguar primeiro o tipo que ela apresenta. Assim, podemos concluir que o nosso objetivo final deve ser retornar uma função, que dada uma lista de `Bool`, nos devolve uma `LTree`. Ainda mais, devemos compreender que somos capazes de trabalhar com a versão curried desta função, isto é, trabalharemos a lista de `Bool` e a `LTree` como se fosse um par, permitindo assim saber qual o elemento à cabeça da lista de `Bool`, o que nos permite avançar conforme pede o enunciado: se for `True` avança para a esquerda, se for `False` avança para a direita.

```

navLTree :: LTree a → ([ Bool] → LTree a)
navLTree = cataLTree g
  where g = [flip Leaf, curry nav]
        nav ((l, r), []) = Fork (l [], r [])
        nav ((l, r), (h : t)) = if h then l t else r t

```

$$\begin{array}{ccc}
LTree\ A & \xrightarrow{outBTree} & A + (LTree\ A \times LTree\ A) \\
\downarrow navLTree & & \downarrow id + (navLTree \times navLTree) \\
(LTree\ A \uparrow Bool^*) & \xleftarrow{g} & A + (((LTree\ A) \uparrow Bool^*) \times ((LTree\ A) \uparrow Bool^*))
\end{array}$$

## Problema 4

### C.0.3 bnavLTree

De forma a perceber o que realmente é pretendido com a função `splay`, devemos averiguar primeiro o tipo que ela apresenta. Assim, podemos concluir que o nosso objetivo final deve ser retornar uma função, que dada uma `BTree` de `Bool`, nos devolve uma `LTree`. Ainda mais, devemos compreender que somos capazes de trabalhar com a versão curried desta função, isto é, trabalharemos a `BTree` de `Bool` e a `LTree` como se fosse um par, permitindo assim saber qual o valor no nodo da Árvore, e efetuar uma rotação da árvore como dito no enunciado: se o valor no nodo for `True` avança para o lado esquerdo, se for `False` avança para a direita.

```

bnavLTree = cataLTree g
  where g = [flip Leaf, curry bnav]
        bnav ((l1, r1), Empty) = Fork (l1 Empty, r1 Empty)
        bnav ((l1, r1), (Node (a, (Empty, r2)))) = if a then l1 Empty else r1 r2
        bnav ((l1, r1), (Node (a, (l2, Empty)))) = if a then l1 l2 else r1 Empty

```

$$\begin{array}{ccc}
LTree\ A & \xrightarrow{outBTree} & A + (LTree\ A \times LTree\ A) \\
\downarrow bnavLTree & & \downarrow id + (bnavLTree \times bnavLTree) \\
(LTree\ A \uparrow BTree\ Bool) & \xleftarrow{g} & A + (((LTree\ A) \uparrow (BTree\ Bool)) \times ((LTree\ A) \uparrow (BTree\ Bool)))
\end{array}$$



```
pbnavlTree = cataLTree g
  where g = ⊥
```

$$\begin{array}{ccc}
 \text{LTree } A & \xrightarrow{\text{outBTree}} & A + (\text{LTree } A \times \text{LTree } A) \\
 \downarrow \text{pbnavlTree} & & \downarrow \text{id} + (\text{pbnavlTree} \times \text{pbnavlTree}) \\
 (\text{Dist } (\text{LTree } A) \uparrow (\text{BTree } (\text{Dist } \text{Bool}))) & \xleftarrow{g} & A + ((\text{Dist } (\text{LTree } A) \uparrow (\text{BTree } (\text{Dist } \text{Bool})))) \uparrow 2
 \end{array}$$

## Problema 5

```
truchet1 = Pictures [put (0,80) (Arc (-90) 0 40), put (80,0) (Arc 90 180 40)]
truchet2 = Pictures [put (0,0) (Arc 0 90 40), put (80,80) (Arc 180 (-90) 40)]
-- janela para visualizar:
janela = InWindow
  "Truchet" -- window title
  (800,800) -- window size
  (100,100) -- window position
  -- defs auxiliares -----
put = Translate
--
```

De forma a gerarmos um mosaico de Truchet-Smith devemos pensar neste como se fosse uma matriz de imagens truchet1 e truchet2. Estas por sua vez devem ser vistas como simples zeros e uns. A partir daqui podemos gerar uma lista aleatória de uma dada dimensão N, formada por vários zeros e uns. Substituindo os zeros pela imagem truchet1 e os uns pela imagem truchet2 obtemos uma linha do mosaico a gerar. Assim só temos que repetir o processo de forma a gerar o restante número de linhas necessárias para obtermos a imagem completa. Assim, o nosso mosaico de Truchet-Smith pode ser visto como uma matriz de zeros e uns que são posteriormente substituídos pelas imagens associadas.

```
main :: IO ()
main = do let pics = [truchet1, truchet2]
  finalImage = ((render pics) (rmatriz 10 10))
  render pics = fmap (drawFullImage pics (-400) (-400))
  (display janela white) <=< (finalImage)

drawFullImage :: [Picture] → Float → Float → [[Int]] → Picture
drawFullImage pics _ _ [] = blank
drawFullImage pics x y (m : ms) = pictures [(drawList pics x y m), (drawFullImage pics (x + 80) y ms)]

drawList :: [Picture] → Float → Float → [Int] → Picture
drawList pics _ _ [] = blank
drawList pics x y (l : ls) = pictures [put (x, y) (pics !! l), drawList pics x (y + 80) ls]

rlista :: Int → IO [Int]
rlista x = replicateM x (randomRIO (0,1))

rmatriz :: Int → Int → IO [[Int]]
rmatriz 0 _ = return []
rmatriz _ 0 = return []
rmatriz x y = do
  m <- rlista y
  ms <- rmatriz (x - 1) (y)
  return (m : ms)
```

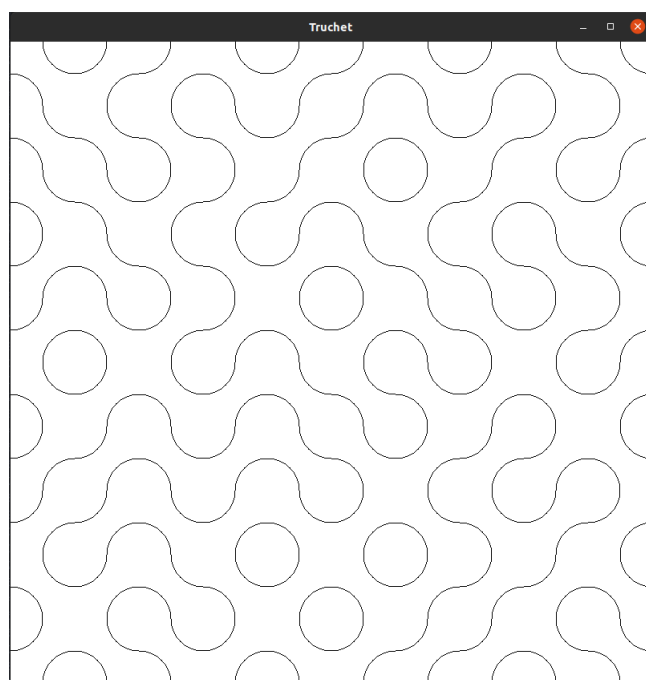


Figura 7: Um mosaico de Truchet-Smith.

# Índice

L<sup>A</sup>T<sub>E</sub>X, [1](#)

**bibtex**, [2](#)

    lhs2TeX, [1](#)

    makeindex, [2](#)

Combinador “pointfree”

*cata*, [11](#)

*either*, [12](#), [14](#), [15](#)

Cálculo de Programas, [1](#), [2](#)

    Material Pedagógico, [1](#)

    BTree.hs, [3](#)

Functor, [4](#), [6–9](#), [12–14](#)

Função

$\pi_1$ , [11](#)

$\pi_2$ , [11](#), [12](#), [14](#)

*length*, [7](#), [12](#)

*map*, [11](#)

*uncurry*, [12](#), [15](#)

Haskell, [1](#), [2](#), [6](#), [9](#)

    “Literate Haskell”, [1](#)

    Biblioteca

        PFP, [8](#)

        Probability, [7](#), [8](#)

    Gloss, [2](#), [9](#), [13](#)

    interpretador

        GHCi, [2](#), [8](#)

    Monad

        Random, [9](#)

    QuickCheck, [2](#)

Mosaico de Truchet, [9](#)

Números naturais ( $\mathbb{N}$ ), [11](#)

Programação literária, [1](#)

U.Minho

    Departamento de Informática, [1](#)

## Referências

- [1] M. Erwig and S. Kollmansberger. Functional pearls: Probabilistic functional programming in Haskell. *J. Funct. Program.*, 16:21–34, January 2006.
- [2] D.E. Knuth. *Literate Programming*. CSLI Lecture Notes Number 27. Stanford University Center for the Study of Language and Information, Stanford, CA, USA, 1992.
- [3] J.N. Oliveira. *Program Design by Calculation*, 2018. Draft of textbook in preparation. viii+297 pages. Informatics Department, University of Minho.