Cálculo de Programas
Trabalho Prático
MiEI+LCC — 2019/20

Departamento de Informática
Universidade do Minho

Junho de 2020

Grupo nr.	25
a83631	Filipa Alves dos Santos
a86579	Ivo Alexandre Pereira Baixo
a67656	Rui Alves dos Santos

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`[1] 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-processador que faz "pretty printing" de código Haskell em LaTeX 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
```

[1] 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:

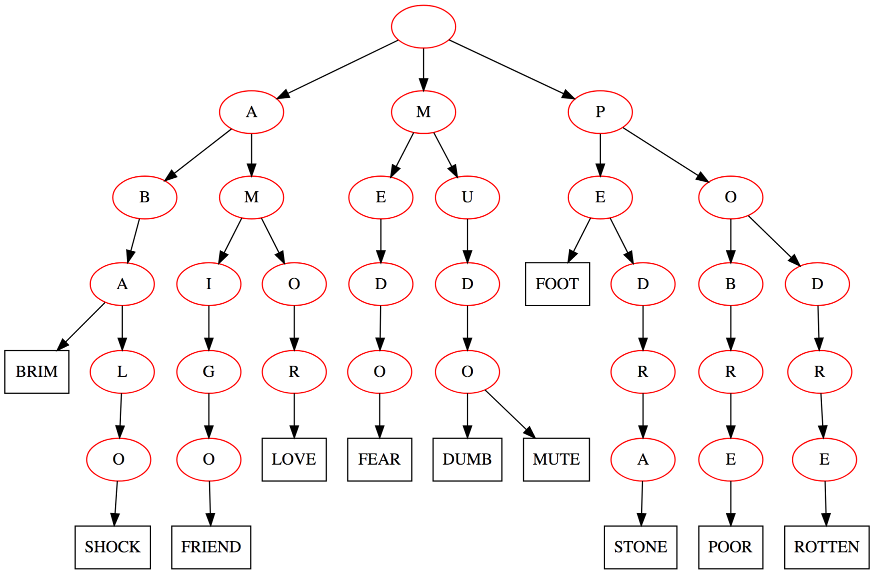


Figura 1: Representação em memória do dicionário dado para testes.

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

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

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

$$\begin{array}{l} prop\_dic\_red\ p\ s\ d \\ \quad \mid dic\_red\ p\ s\ d = dic\_imp\ d \equiv dic\_in\ p\ s\ (dic\_imp\ d) \\ \quad \mid otherwise = True \end{array}$$

Propriedade [QuickCheck] 3 *A operação dic_rd implementa a procura na correspondente exportação do dicionário:*

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

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.[2]

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 raíz tem o valor a, um filho s_1 à esquerda e um filho s_2 à direita. Assuma

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

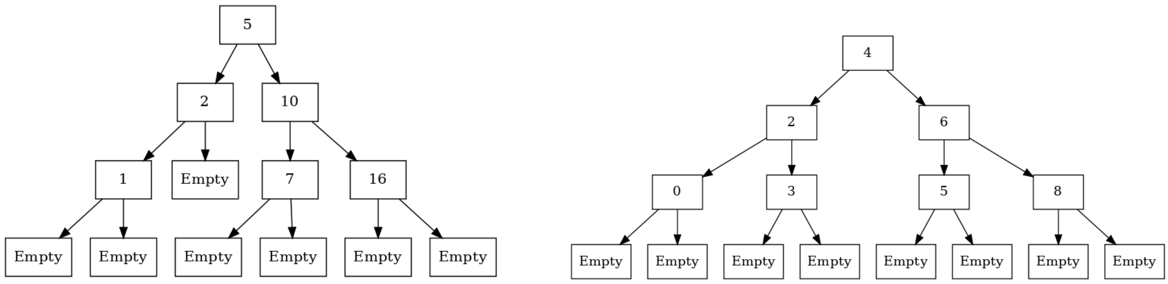


Figura 2: Duas árvores binárias de procura; a da esquerda vai ser designada por t_1 e a da direita por t_2.

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.

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

Seguem alguns exemplos dos resultados que se esperam ao aplicar estas funções à árvore da esquerda ($t1$) e à árvore da direita ($t2$) 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 $maisEsq$ e $maisDir$ são determinadas unicamente pela propriedade*

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

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:*

$$propEsq\ Empty = property\ Discard$$
$$propEsq\ x@(Node\ (a, (t, s))) = (maisEsq\ t) \not\equiv Nothing \Rightarrow (maisEsq\ x) \equiv maisEsq\ t$$

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

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

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

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

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

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

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

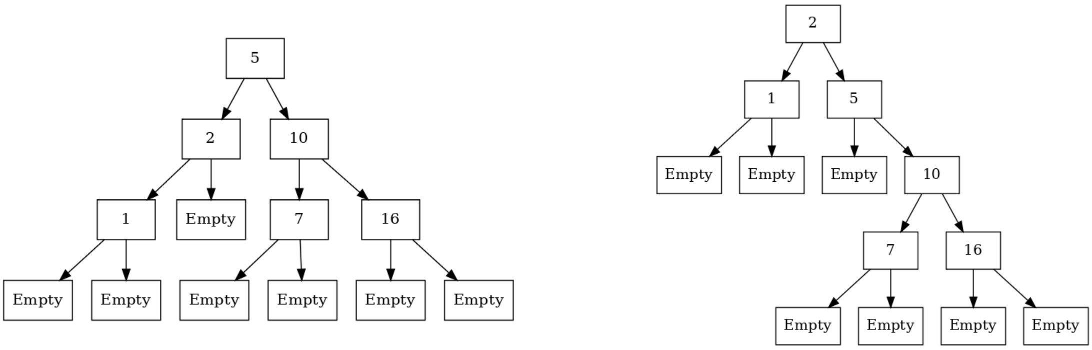


Figura 3: Exemplo de uma rotação à direita. A árvore da esquerda é a árvore original; a árvore da direita representa a rotação à direita correspondente.

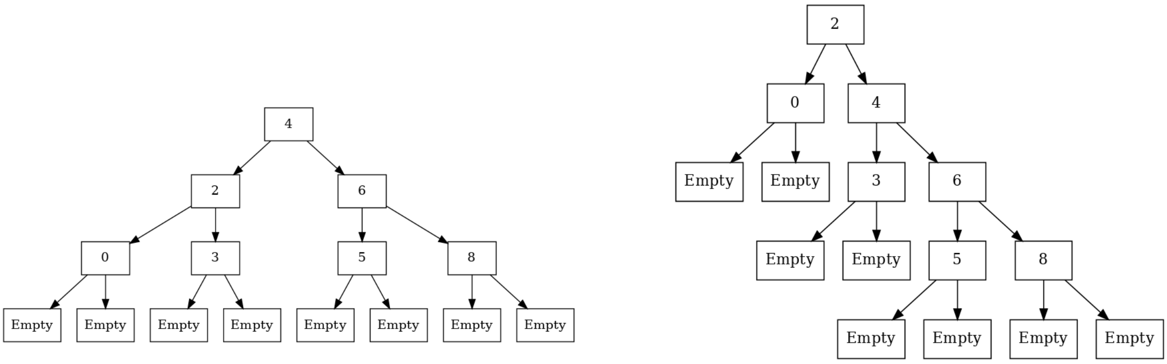


Figura 4: Exemplo de uma rotação à direita. A árvore da esquerda é a árvore original; a árvore da direita representa a rotação à direita correspondente.

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

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

As árvores binárias providenciam uma boa maneira de reduzir o espaço de procura. Mas podemos fazer ainda melhor: podemos aproximar da raíz os elementos da árvore que são mais acedidos, reduzindo assim o espaço de procura na *dimensão vertical*[3]. 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 raíz 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 raíz, que mantém o filho à esquerda e adopta 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 correspodente. De facto, podemos sempre aplicar uma *sequência* de rotações numa árvore de forma a mover um dado nó para a raíz (dando origem portanto à referida operação de splaying).

Começe então por implementar as funções

[3] Note que nas árvores de binária de procura a redução é feita na dimensão horizontal.

$$
\begin{array}{l}
rrot :: \mathsf{BTree}\ a \rightarrow \mathsf{BTree}\ a \\
lrot :: \mathsf{BTree}\ a \rightarrow \mathsf{BTree}\ a
\end{array}
$$

de rotação à direita e à esquerda.

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

$$
\begin{array}{l}
prop\_ord\_pres\_esq = forAll\ orderedBTree\ (isOrd \cdot lrot) \\
prop\_ord\_pres\_dir = forAll\ orderedBTree\ (isOrd \cdot rrot)
\end{array}
$$

De seguida implemente a operação de splaying

$$
splay :: [Bool] \rightarrow (\mathsf{BTree}\ a \rightarrow \mathsf{BTree}\ a)
$$

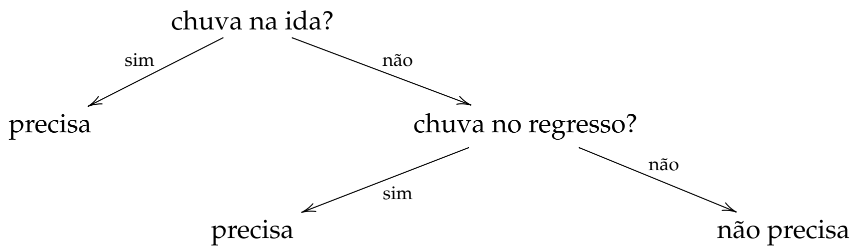
como um catamorfismo de listas. O argumento $[Bool]$ representa um caminho ao longo de uma árvore, em que o valor $True$ representa "seguir pelo ramo da esquerda"e o valor $False$ representa "seguir pelo ramo da direita". O caminho ao longo de uma árvore serve para *identificar* unicamente um nó dessa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

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

$$
\begin{array}{l}
prop\_ord\_pres\_splay :: [Bool] \rightarrow Property \\
prop\_ord\_pres\_splay\ path = forAll\ orderedBTree\ (isOrd \cdot (splay\ path))
\end{array}
$$

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 climatéricas. 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:

$$
\mathbf{data}\ Bdt\ a = Dec\ a \mid Query\ (String, (Bdt\ a, Bdt\ a))\ \mathbf{deriving}\ Show
$$

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. $extLTree : Bdt\ a \rightarrow \mathsf{LTree}\ a$ (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. $navLTree : \mathsf{LTree}\ a \rightarrow ([Bool] \rightarrow \mathsf{LTree}\ a)$ (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 àrvore 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.*

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

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

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

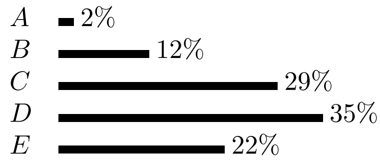
Problema 4

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

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

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 :: Dist a indica que a probabilidade de a é p, devendo ser garantida a propriedade de que todas as probabilidades de d somam 100%. Por exemplo, a seguinte distribuição de classificações por escalões de A a E,



será representada pela distribuição

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

que o GHCi mostrará assim:

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

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

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

isto é

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

distribuição *normais*, eg.

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

etc.[4] Dist forma um **mónade** cuja unidade é $return\ a = D\ [(a,1)]$ e cuja composição de Kleisli é (simplificando a notação)

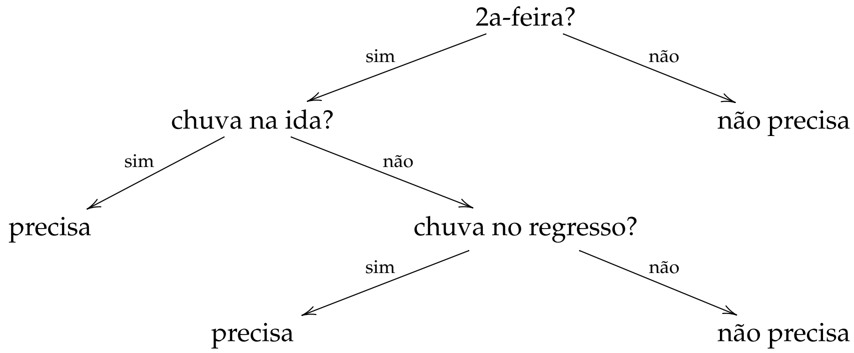
$$(f \bullet g)\ a = [(y, q * p) \mid (x,p) \leftarrow g\ a, (y,q) \leftarrow f\ x]$$

em que $g : A \to \mathsf{Dist}\ B$ e $f : B \to \mathsf{Dist}\ C$ 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 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

[4]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

$$bnavLTree :: \mathsf{LTree}\ a \rightarrow ((\mathsf{BTree}\ Bool) \rightarrow \mathsf{LTree}\ a)$$

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

$$pbnavLTree :: \mathsf{LTree}\ a \rightarrow ((\mathsf{BTree}\ (\mathsf{Dist}\ Bool)) \rightarrow \mathsf{Dist}\ (\mathsf{LTree}\ a))$$

que deverá consiste 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 6 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.

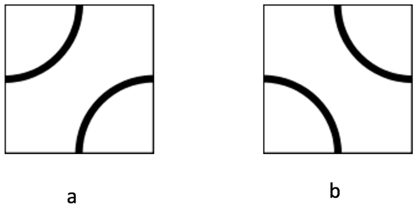


Figura 5: Os dois ladrilhos de Truchet-Smith.

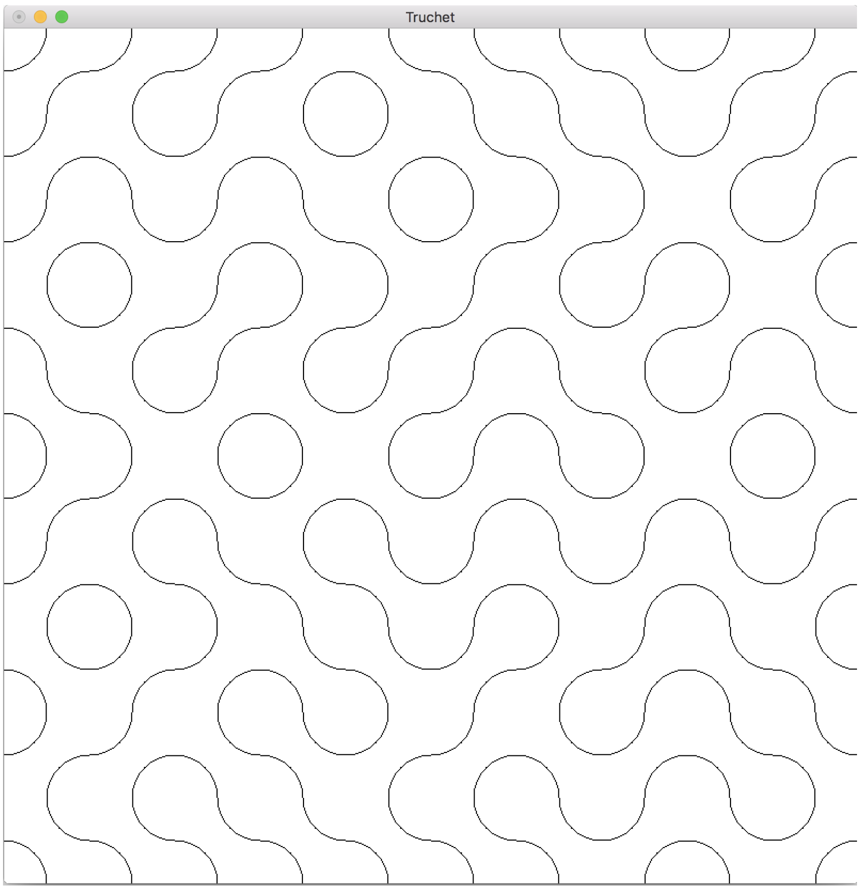


Figura 6: Um mosaico de Truchet-Smith.

Anexos

A Como exprimir cálculos e diagramas em LaTeX/lhs2tex

Estudar o texto fonte deste trabalho para obter o efeito:[5]

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

Os diagramas podem ser produzidos recorrendo à *package* LaTeX xymatrix, por exemplo:

$$
\begin{tikzcd}
\mathbb{N}_0 \arrow[d, "(\!| g |\!)"'] & 1 + \mathbb{N}_0 \arrow[l, "\mathsf{in}"'] \arrow[d, "id + (\!| g |\!)"] \\
B & 1 + B \arrow[l, "g"]
\end{tikzcd}
$$

B Código fornecido

Problema 1

Função de representação de um dicionário:

$$
\begin{array}{l}
dic\_imp :: [(String, [String])] \rightarrow Dict \\
dic\_imp = Term \; \texttt{""} \cdot \mathsf{map} \; (bmap \; id \; singl) \cdot untar \cdot discollect
\end{array}
$$

onde

$$
\mathbf{type} \; Dict = Exp \; String \; String
$$

Dicionário para testes:

$$
\begin{array}{l}
d :: [(String, [String])] \\
d = [(\texttt{"ABA"}, [\texttt{"BRIM"}]), \\
\qquad (\texttt{"ABALO"}, [\texttt{"SHOCK"}]), \\
\qquad (\texttt{"AMIGO"}, [\texttt{"FRIEND"}]), \\
\qquad (\texttt{"AMOR"}, [\texttt{"LOVE"}]), \\
\qquad (\texttt{"MEDO"}, [\texttt{"FEAR"}]), \\
\qquad (\texttt{"MUDO"}, [\texttt{"DUMB"}, \texttt{"MUTE"}]), \\
\qquad (\texttt{"PE"}, [\texttt{"FOOT"}]), \\
\qquad (\texttt{"PEDRA"}, [\texttt{"STONE"}]), \\
\qquad (\texttt{"POBRE"}, [\texttt{"POOR"}]), \\
\qquad (\texttt{"PODRE"}, [\texttt{"ROTTEN"}])]
\end{array}
$$

Normalização de um dicionário (remoção de entradas vazias):

$$
\begin{array}{l}
dic\_norm = collect \cdot filter \; p \cdot discollect \; \mathbf{where} \\
\quad p \; (a, b) = a > \texttt{""} \wedge b > \texttt{""}
\end{array}
$$

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

$$
dic\_red \; p \; s \; d = (p, s) \in discollect \; d
$$

[5] Exemplos tirados de [3].

Problema 2

Árvores usadas no texto:

$$
\begin{aligned}
&emp\ x = Node\ (x, (Empty, Empty))\\
&t7 = emp\ 7\\
&t16 = emp\ 16\\
&t7\_10\_16 = Node\ (10, (t7, t16))\\
&t1\_2\_nil = Node\ (2, (emp\ 1, Empty))\\
&t' = Node\ (5, (t1\_2\_nil, t7\_10\_16))\\
&t0\_2\_1 = Node\ (2, (emp\ 0, emp\ 3))\\
&t5\_6\_8 = Node\ (6, (emp\ 5, emp\ 8))\\
&t2 = Node\ (4, (t0\_2\_1, t5\_6\_8))\\
&dotBt :: (Show\ a) \Rightarrow \mathsf{BTree}\ a \to \mathsf{IO}\ ExitCode\\
&dotBt = dotpict \cdot bmap\ Just\ Just \cdot cBTree2Exp \cdot (\mathsf{fmap}\ show)
\end{aligned}
$$

Problema 3

Funções usadas para efeitos de teste:

$$
\begin{aligned}
&tipsBdt :: Bdt\ a \to [a]\\
&tipsBdt = cataBdt\ [singl, \widehat{(+\!\!+)} \cdot \pi_2]\\
&tipsLTree = tips
\end{aligned}
$$

Problema 5

Função de permutação aleatória de uma lista:

$$
\begin{aligned}
&permuta\ [\,] = return\ [\,]\\
&permuta\ x = \mathbf{do}\ \{(h, t) \leftarrow getR\ x; t' \leftarrow permuta\ t; return\ (h : t')\}\ \mathbf{where}\\
&\quad getR\ x = \mathbf{do}\ \{i \leftarrow getStdRandom\ (randomR\ (0, \mathsf{length}\ \ x - 1)); return\ (x\ !!\ i, retira\ i\ x)\}\\
&\quad retira\ i\ x = take\ i\ x +\!\!+ drop\ (i + 1)\ x
\end{aligned}
$$

QuickCheck

Código para geração de testes:

$$
\begin{aligned}
&\mathbf{instance}\ Arbitrary\ a \Rightarrow Arbitrary\ (\mathsf{BTree}\ a)\ \mathbf{where}\\
&\quad arbitrary = sized\ genbt\ \ \mathbf{where}\\
&\quad\quad genbt\ 0 = return\ (inBTree\ \$\ i_1\ ())\\
&\quad\quad genbt\ n = oneof\ [(liftM2\ \$\ curry\ (inBTree \cdot i_2))\\
&\quad\quad\quad QuickCheck.arbitrary\ (liftM2\ (,)\ (genbt\ (n-1))\ (genbt\ (n-1))),\\
&\quad\quad\quad (liftM2\ \$\ curry\ (inBTree \cdot i_2))\\
&\quad\quad\quad QuickCheck.arbitrary\ (liftM2\ (,)\ (genbt\ (n-1))\ (genbt\ 0)),\\
&\quad\quad\quad (liftM2\ \$\ curry\ (inBTree \cdot i_2))\\
&\quad\quad\quad QuickCheck.arbitrary\ (liftM2\ (,)\ (genbt\ 0)\ (genbt\ (n-1)))]\\
&\mathbf{instance}\ (Arbitrary\ v, Arbitrary\ o) \Rightarrow Arbitrary\ (Exp\ v\ o)\ \mathbf{where}\\
&\quad arbitrary = (genExp\ 10)\ \mathbf{where}\\
&\quad\quad genExp\ 0 = liftM\ (inExp \cdot i_1)\ QuickCheck.arbitrary\\
&\quad\quad genExp\ n = oneof\ [liftM\ (inExp \cdot i_2 \cdot (\lambda a \to (a, [\,])))\ QuickCheck.arbitrary,\\
&\quad\quad\quad liftM\ (inExp \cdot i_1)\ QuickCheck.arbitrary,\\
&\quad\quad\quad liftM\ (inExp \cdot i_2 \cdot (\lambda(a, (b, c)) \to (a, [b, c])))\\
&\quad\quad\quad \$\ (liftM2\ (,)\ QuickCheck.arbitrary\ (liftM2\ (,)\\
&\quad\quad\quad\quad (genExp\ (n-1))\ (genExp\ (n-1)))),\\
&\quad\quad\quad liftM\ (inExp \cdot i_2 \cdot (\lambda(a, (b, c, d)) \to (a, [b, c, d])))\\
&\quad\quad\quad \$\ (liftM2\ (,)\ QuickCheck.arbitrary\ (liftM3\ (,,)
\end{aligned}
$$

$$
\begin{array}{l}
\qquad\quad (genExp\ (n-1))\ (genExp\ (n-1))\ (genExp\ (n-1))))\\
\qquad]\\
orderedBTree :: Gen\ (\mathsf{BTree}\ Int)\\
orderedBTree = liftM\ (foldr\ insOrd\ Empty)\ (QuickCheck.arbitrary :: Gen\ [Int])
\end{array}
$$

$$
\begin{array}{l}
\mathbf{instance}\ (Arbitrary\ a) \Rightarrow Arbitrary\ (Bdt\ a)\ \mathbf{where}\\
\quad arbitrary = sized\ genbt\ \ \mathbf{where}\\
\quad\quad genbt\ 0 = liftM\ \ Dec\ QuickCheck.arbitrary\\
\quad\quad genbt\ n = oneof\ [(liftM2\ \$\ curry\ Query)\\
\quad\quad\quad QuickCheck.arbitrary\ (liftM2\ (,)\ (genbt\ (n-1))\ (genbt\ (n-1))),\\
\quad\quad\quad (liftM2\ \$\ curry\ (Query))\\
\quad\quad\quad QuickCheck.arbitrary\ (liftM2\ (,)\ (genbt\ (n-1))\ (genbt\ 0)),\\
\quad\quad\quad (liftM2\ \$\ curry\ (Query))\\
\quad\quad\quad QuickCheck.arbitrary\ (liftM2\ (,)\ (genbt\ 0)\ (genbt\ (n-1)))]
\end{array}
$$

Outras funções auxiliares

Lógicas:

$$
\begin{array}{l}
\mathbf{infixr}\ 0 \Rightarrow\\
(\Rightarrow) :: (Testable\ prop) \Rightarrow (a \rightarrow Bool) \rightarrow (a \rightarrow prop) \rightarrow a \rightarrow Property\\
p \Rightarrow f = \lambda a \rightarrow p\ \ a \Rightarrow f\ \ a\\
\mathbf{infixr}\ 0 \Leftrightarrow\\
(\Leftrightarrow) :: (a \rightarrow Bool) \rightarrow (a \rightarrow Bool) \rightarrow a \rightarrow Property\\
p \Leftrightarrow f = \lambda a \rightarrow (p\ \ a \Rightarrow property\ (f\ \ a))\ .\&\&.\ (f\ \ a \Rightarrow property\ (p\ \ a))\\
\mathbf{infixr}\ 4 \equiv\\
(\equiv) :: Eq\ b \Rightarrow (a \rightarrow b) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow Bool)\\
f \equiv g = \lambda a \rightarrow f\ \ a \equiv g\ \ a\\
\mathbf{infixr}\ 4 \leqslant\\
(\leqslant) :: Ord\ b \Rightarrow (a \rightarrow b) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow Bool)\\
f \leqslant g = \lambda a \rightarrow f\ \ a \leqslant g\ \ a\\
\mathbf{infixr}\ 4 \land\\
(\land) :: (a \rightarrow Bool) \rightarrow (a \rightarrow Bool) \rightarrow (a \rightarrow Bool)\\
f \land g = \lambda a \rightarrow ((f\ \ a) \land (g\ \ a))
\end{array}
$$

Compilação e execução dentro do interpretador:[6]

$$run = \mathbf{do}\ \{system\ \texttt{"ghc cp1920t"}; system\ \texttt{"./cp1920t"}\}$$

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

Função discollect

A *discollect* é utilizada na função *dic_imp* (que tem como objetivo importar dicionários do formato "lista de pares palavra-tradução"). Esta função *discollect* recebe uma lista de pares, em que cada par corresponde à palavra em português e a respectiva lista de traduções em inglês. Para cada elemento destas listas será criado um par individual, com a sua tradução portuguesa. A função devolve a lista com todos os estes novos pares.

[6]Pode ser útil em testes envolvendo Gloss. Nesse caso, o teste em causa deve fazer parte de uma função *main*.

$$discollect :: (Ord\ b, Ord\ a) \Rightarrow [(b, [a])] \rightarrow [(b, a)]$$
$$discollect\ [\,] = [\,]$$
$$discollect\ ((b, [\,]) : t) = discollect\ t$$
$$discollect\ ((b, (h : l)) : t) = (b, h) : discollect\ ((b, l) : t)$$

Função dic-exp

Já a dic_exp é composta por duas funções distintas: a $collect$, já definida, e a tar. A tar é a função principal, que exporta dicionários para o formato "lista de pares palavra-tradução", como é pedido na dic_exp. Já a $collect$ faz o contrário da $discollect$ explicada anteriormente: junta os pares que têm a mesma palavra em português num só, e coloca todas as suas traduções numa lista.

$$dic\_exp :: Dict \rightarrow [(String, [String])]$$
$$dic\_exp = collect \cdot tar$$

$$tar = cataExp\ g\ \textbf{where}$$
$$\quad g = [g1, g2]\ \textbf{where}$$
$$\quad\quad g1\ s = singl\ \langle nil, id\rangle\ s$$
$$\quad\quad g2\ (s, [\,]) = [\,]$$
$$\quad\quad g2\ (s, [\,] : t) = g2\ (s, t)$$
$$\quad\quad g2\ (s, ((h1, h2) : l) : t) = [(s + h1, h2)] + g2\ (s, (l : t))$$

Função dic-rd

Para a função dic_rd, foi pedido a implementação da procura de traducões para uma determinada palavra. A dic_exp é utilizada para transformar o dicionário numa lista de pares e de seguida, a dic_rd_aux itera esta lista até encontrar a palavra fornecida como argumento ou até chegar ao fim do dicionário sem a ter encontrado. No primeiro caso, é devolvido a lista de traduções associada a essa palavra ($Just\ l$) e no segundo, é devolvido $Nothing$.

$$dic\_rd\ p\ d = dic\_rd\_aux\ p\ (dic\_exp\ d)$$

$$dic\_rd\_aux :: String \rightarrow [(String, [String])] \rightarrow Maybe\ [String]$$
$$dic\_rd\_aux\ p\ [\,] = Nothing$$
$$dic\_rd\_aux\ p\ ((s, l) : t) = \textbf{if}\ p \equiv s\ \textbf{then}\ Just\ l\ \textbf{else}\ dic\_rd\_aux\ p\ t$$

Função dic-in

A função dic_in, que tem como objetivo inserir palavras novas (palavra e tradução), utiliza um raciocínio semelhante à dic_rd. É novamente utilizada a dic_exp de modo a facilitar a posterior inserção na dic_in_aux. Nesta última função, se a palavra já existir, a nova tradução é adicionada à lista do respetivo par. Caso contrário, é criado um novo par. No final, a dic_imp transforma a lista de pares palavra-tradução de volta para o tipo original $Dict$.

$$dic\_in :: String \rightarrow String \rightarrow Dict \rightarrow Dict$$
$$dic\_in\ p\ t\ d = dic\_imp\ (dic\_in\_aux\ p\ t\ (dic\_exp\ d)\ (dic\_exp\ d))$$

$$dic\_in\_aux :: String \rightarrow String \rightarrow [(String, [String])] \rightarrow [(String, [String])] \rightarrow [(String, [String])]$$
$$dic\_in\_aux\ p\ t\ o\ [\,] = ((p, [t]) : o)$$
$$dic\_in\_aux\ palavra\ traducao\ o\ ((s, t) : l) = \textbf{if}\ palavra \equiv s$$
$$\quad \textbf{then}\ ((s, (traducao : t)) : l)$$
$$\quad \textbf{else}\ dic\_in\_aux\ palavra\ traducao\ o\ l$$

Problema 2

Funções maisDir/maisEsq

As funcões *maisDir* e *maisEsq* retornam respetivamente os valores mais à direita e à esquerda de uma árvore do tipo BTree. No caso em que a árvore é vazia retornam Nothing e no caso de a árvore apenas ter um elemento retornam esse mesmo elemento.

$$
\begin{array}{l}
maisDir = cataBTree\ g\ \textbf{where} \\
\quad g = [g1, g2]\ \textbf{where} \\
\qquad g1\ () = Nothing \\
\qquad g2\ (a, (mba1, Nothing)) = Just\ a \\
\qquad g2\ (a, (mba1, mba2)) = mba2 \\
maisEsq = cataBTree\ g\ \textbf{where} \\
\quad g = [g1, g2]\ \textbf{where} \\
\qquad g1\ () = Nothing \\
\qquad g2\ (a, (Nothing, mba2)) = Just\ a \\
\qquad g2\ (a, (mba1, mba2)) = mba1
\end{array}
$$

Funções insOrd'/insOrd

Começamos por definir a função $insOrd'$ que insere um elemento numa BTree, retornando um par de BTree, sendo a da esquerda a BTree resultante de fazer a inserção e a da direita a BTree original. Assim, para definir a função *insOrd* basta escolher a árvore da esquerda do par resultante da função $insOrd'$.

$$
\begin{array}{l}
insOrd'\ x = cataBTree\ g\ \textbf{where} \\
\quad g = [g1, g2]\ \textbf{where} \\
\qquad g1\ () = (Node\ (x, (Empty, Empty)), Empty) \\
\qquad g2\ (a, ((lx, l), (rx, r))) = \textbf{if}\ (x \geqslant a) \\
\qquad\quad \textbf{then}\ (Node\ (a, (l, rx)), Node\ (a, (l, r))) \\
\qquad\quad \textbf{else}\ (Node\ (a, (lx, r)), Node\ (a, (l, r))) \\
insOrd\ a\ x = \pi_1\ (insOrd'\ a\ x)
\end{array}
$$

Funções insOrd'/insOrd

De modo análogo à função *insOrd*, para definir a função *isOrd* também utilizamos uma função auxiliar $isOrd'$ que consiste num catamorfismo de BTree. A função $isOrd'$ retorna um par em que o primeiro elemento é um booleano que nos diz se a árvore (no segundo elemento do par) está ou não ordenada. Para definir a função *isOrd* basta selecionar o primeiro elemento do par resultante da função $isOrd'$.

$$
\begin{array}{l}
isOrd' = cataBTree\ g\ \textbf{where} \\
g = [g1, g2]\ \textbf{where} \\
\quad g1 = \langle \underline{True}, \underline{Empty} \rangle \\
\quad g2\ (a, ((b1, Empty), (b2, Empty))) = (True, Node\ (a, (Empty, Empty))) \\
\quad g2\ (a, ((b1, Empty), (b2, Node\ (y, (e, d))))) = ((a \leqslant y \wedge b2), Node\ (a, (Empty, Node\ (y, (e, d))))) \\
\quad g2\ (a, ((b1, Node\ (x, (e, d))), (b2, Empty))) = ((a \geqslant x \wedge b1), Node\ (a, (Node\ (x, (e, d)), Empty))) \\
\quad g2\ (a, ((b1, Node\ (x, (e_1, d1))), (b2, Node\ (y, (e_2, d2))))) = \\
\qquad ((a \geqslant x \wedge a \leqslant y \wedge b1 \wedge b2), Node\ (a, (Node\ (x, (e_1, d1)), Node\ (y, (e_2, d2))))) \\
isOrd = \pi_1 \cdot isOrd'
\end{array}
$$

Funções insOrd'/insOrd

As funções *rrot* e *lrot* fazem respetivamente uma rotação da BTree dada para a direita e para a esquerda.

$$
\begin{array}{l}
rrot\ Empty = Empty \\
rrot\ (Node\ (a, (Empty, Empty))) = (Node\ (a, (Empty, Empty))) \\
rrot\ (Node\ (a, (Empty, d))) = Node\ (a, (Empty, d))
\end{array}
$$

$$rrot\ (Node\ (a, (Node\ (e, (l, r)), Empty))) = Node\ (e, (l, Node\ (a, (r, Empty))))$$
$$rrot\ (Node\ (a, (Node\ (l, (l1, r1)), Node\ (r, (l2, r2))))) = Node\ (l, (l1, \quad Node\ (a, (r1, Node\ (r, (l2, r2))))))$$
$$lrot\ Empty = Empty$$
$$lrot\ (Node\ (a, (Empty, Empty))) = (Node\ (a, (Empty, Empty)))$$
$$lrot\ (Node\ (a, (l, Empty))) = Node\ (a, (l, Empty))$$
$$lrot\ (Node\ (a, (Empty, Node\ (d, (l, r))))) = Node\ (d, (Node\ (a, (Empty, l)), r))$$
$$lrot\ (Node\ (a, (Node\ (l, (l1, r1)), Node\ (r, (l2, r2))))) = Node\ (r, (Node\ (a, (Node\ (l, (l1, r1)), l2)), r2))$$

Funções insOrd'/insOrd

Começamos por tentar implementar a função splay como um catamorfismo de listas, mas sem efetivamente termos conseguido. Após consultarmos as FAQ's da UC decidimos optar por construír esta função com base num catamorfismo de BTree. Criamos uma função auxiliar $splay'$ que é um catamorfismo de BTree, e a função splay fica definida então da seguinte forma:

$$splay\ l\ t = splay'\ t\ l$$
$$splay' :: \mathsf{BTree}\ a \rightarrow [Bool] \rightarrow \mathsf{BTree}\ a$$
$$splay' = cataBTree\ g\ \mathbf{where}$$
$$\quad g = [g1, g2]\ \mathbf{where}$$
$$\quad\quad g1\ ()\ \_ = Empty$$
$$\quad\quad g2\ (a, (func1, func2))\ [\,] = Node\ (a, (func1\ [\,], func2\ [\,]))$$
$$\quad\quad g2\ (a, (func1, func2))\ (h : t) = \mathbf{if}\ h$$
$$\quad\quad\quad \mathbf{then}\ rrot\ (Node\ (a, (func1\ t, func2\ [\,])))$$
$$\quad\quad\quad \mathbf{else}\ lrot\ (Node\ (a, (func1\ [\,], func2\ t\ )))$$

Problema 3

Função inBdt

$$inBdt :: a + (String, (Bdt\ a, Bdt\ a)) \rightarrow Bdt\ a$$
$$inBdt = [Dec, Query]$$

Função outBdt

$$outBdt :: Bdt\ a \rightarrow a + (String, (Bdt\ a, Bdt\ a))$$
$$outBdt\ (Dec\ a) = i_1\ a$$
$$outBdt\ (Query\ (s, (b1, b2))) = i_2\ (s, (b1, b2))$$

Funções auxiliares baseBdt e recBdt

$$baseBdt\ f\ g = id + (f \times (g \times g))$$
$$recBdt\ g = baseBdt\ id\ g$$

Função cataBdt

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

Função anaBdt

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

Diagrama de anaBdt

$$\begin{tikzcd}
Bdt & & a + (String \times (Bdt\ a \times Bdt\ a)) \arrow[ll, "inBdt"'] \\
A \arrow[u, "[(g)]"] \arrow[rr, "g"] & & a + (String \times (A \times A)) \arrow[u, "id + id \times [(g)] \times [(g)]"']
\end{tikzcd}$$

Função extLTree

A função $extLTree$, tal como pedido, esquece a informação presente nos nós de uma dada árvore de decisão binária. Quando é apenas uma decisão (Dec), cria-se uma $Leaf$ com o conteúdo e, quando é uma $Query$, é descartada a pergunta e criado um $Fork$ com os filhos (Bdt).

$$\begin{array}{l}
extLTree :: Bdt\ a \rightarrow \mathsf{LTree}\ a \\
extLTree = cataBdt\ g\ \mathbf{where} \\
\quad g = [g1, g2]\ \mathbf{where} \\
\qquad g1\ a = Leaf\ a \\
\qquad g2\ (s, (l1, l2)) = Fork\ (l1, l2)
\end{array}$$

Função navLTree

A $navLTree$ navega um elemento de LTree de acordo com uma sequência de respostas ”sim ou não”. É um catamorfismo de LTree: se o h é $True$, vai para a esquerda e, caso contrário, vai para a direita.

$$\begin{array}{l}
navLTree :: \mathsf{LTree}\ a \rightarrow ([Bool] \rightarrow \mathsf{LTree}\ a) \\
navLTree = cataLTree\ g\ \mathbf{where} \\
\quad g = [g1, g2]\ \mathbf{where} \\
\qquad g1\ a\ _ = Leaf\ a \\
\qquad g2\ (func1, func2)\ [\,] = Fork\ (func1\ [\,], func2\ [\,]) \\
\qquad g2\ (func1, func2)\ (h : t) = \mathbf{if}\ h \\
\qquad\quad \mathbf{then}\ func1\ t \\
\qquad\quad \mathbf{else}\ \ func2\ t
\end{array}$$

Problema 4

Função bnavLTree

A função $bnavLTree$ percorre uma LTree dado um caminho que é do tipo BTree Bool. Definimos assim a função como um catamorfismo de LTree.

$$\begin{array}{l}
bnavLTree = cataLTree\ g\ \mathbf{where} \\
\quad g = [g1, g2]\ \mathbf{where} \\
\qquad g1\ a\ _ = Leaf\ a \\
\qquad g2\ (func1, func2)\ Empty = Fork\ (func1\ Empty, func2\ Empty) \\
\qquad g2\ (func1, func2)\ (Node\ (x, (bt1, bt2))) = \mathbf{if}\ x \\
\qquad\quad \mathbf{then}\ func1\ bt1 \\
\qquad\quad \mathbf{else}\ func2\ bt2
\end{array}$$

Função pbnavLTree

Definimos a função $pbnavLTree$ usando um catamorfismo de LTree e utilizando propriedades da monáde das probabilidades. Criamos também uma BTree (Dist Bool) 'anita' assim como um LTree [Char] 'lt-anita' para podermos responder à questão da Anita dever levar ou não guarda-chuva. A solução obtida foi a seguinte: 86,9 por cento probabilidade de precisar e 13,1 por cento de probabilidade de não precisar.

$pbnavLTree = cataLTree\ g$ **where**
  $g = [g1, g2]$ **where**
    $g1\ \ a\ \_ = D\ [((Leaf\ \ a), 1)]$
    $g2\ (func1, func2)\ Empty = (prod\ (func1\ \ Empty)\ (func2\ \ Empty)) \ggg return \cdot Fork$
    $g2\ (func1, func2)\ (Node\ (x, (bt1, bt2))) = Probability.cond\ x\ (func1\ \ bt1)\ (func2\ \ bt2)$
$anita = Node\ (D\ [(True, 1\ /\ 7), (False, 6\ /\ 7)], (Node\ (D\ [(True, 0.8), (False, 0.2)], (Empty, Node\ (D\ [(True, 0.6),$
$lt\_anita = Fork\ (Fork\ (Leaf$ `"Precisa"`$, Fork\ (Leaf$ `"Precisa"`$, Leaf$ `"Nao precisa"`$)), Leaf$ `"Nao preci`

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 ————-
$put = \widehat{Translate}$
  --

Índice

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.