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

Departamento de Informática
Universidade do Minho

Junho de 2020

Grupo nr.	5
a86475	Paulo Pereira
a89616	Eduardo Coelho

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" [**?**], 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
```

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

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

```
\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:

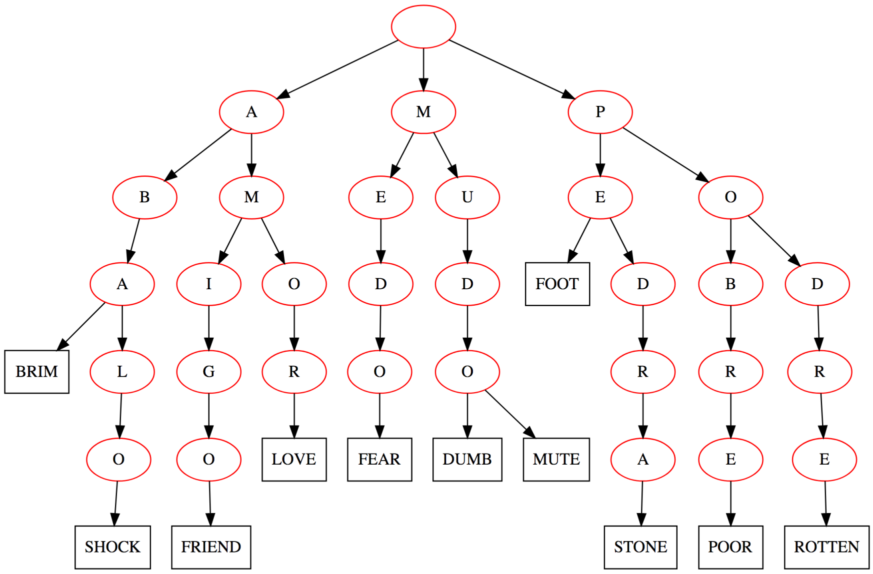


Figura 1: Representação em memória do dicionário dado para testes.

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.

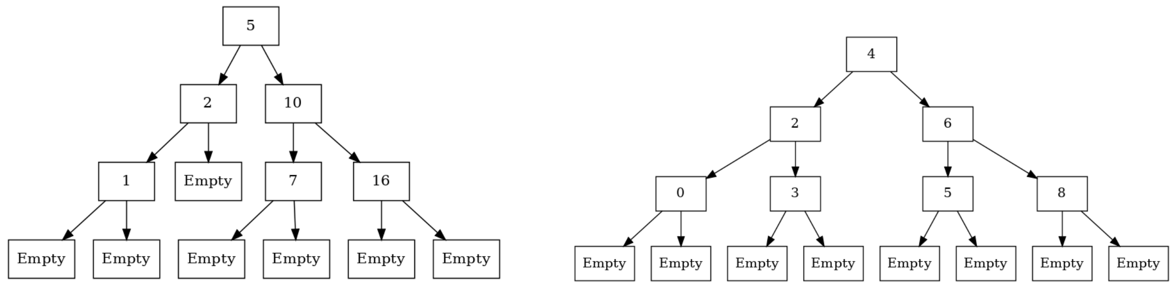


Figura 2: Duas árvores binárias de procura; a da esquerda vai ser designada por t_1 e a da direita por t_2.

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.

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

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*

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

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,

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

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

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

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

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

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

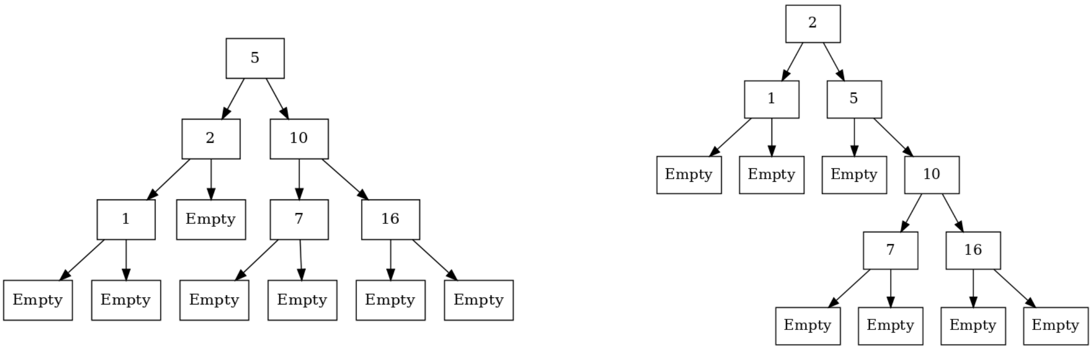


Figura 3: Exemplo de uma rotação à direita. A árvore da esquerda é a árvore original; a árvore da direita representa a rotação à direita correspondente.

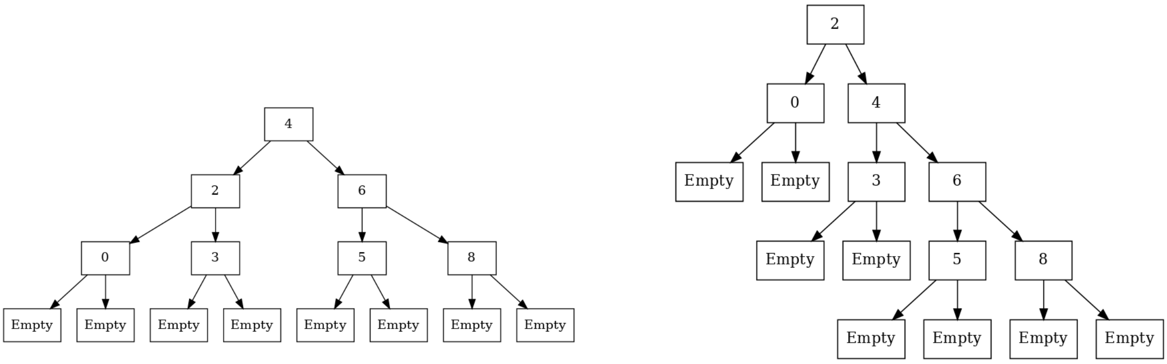


Figura 4: Exemplo de uma rotação à direita. A árvore da esquerda é a árvore original; a árvore da direita representa a rotação à direita correspondente.

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

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

As árvores binárias providenciam uma boa maneira de reduzir o espaço de procura. Mas podemos fazer ainda melhor: podemos aproximar da 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.

$rrot$ :: BTree $a \rightarrow$ BTree a
$lrot$ :: BTree $a \rightarrow$ BTree a

de rotação à direita e à esquerda.

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

$$prop\_ord\_pres\_esq = forAll\ orderedBTree\ (isOrd \cdot lrot)$$
$$prop\_ord\_pres\_dir = forAll\ orderedBTree\ (isOrd \cdot rrot)$$

De seguida implemente a operação de splaying

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

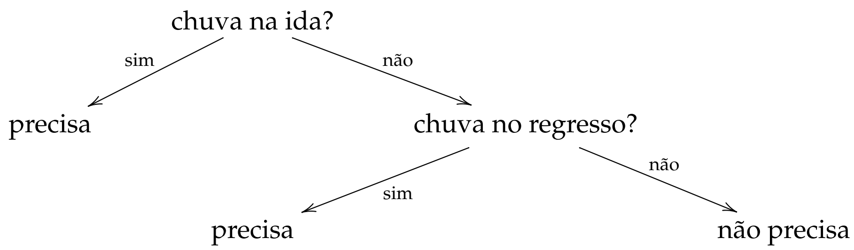
como um catamorfismo de listas. O argumento $[Bool]$ representa um caminho ao longo de uma árvore, em que o valor $True$ representa "seguir pelo ramo da esquerda"e o valor $False$ representa "seguir pelo ramo da direita". O caminho ao longo de uma árvore serve para *identificar* unicamente um nó dessa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

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

$$prop\_ord\_pres\_splay :: [Bool] \rightarrow Property$$
$$prop\_ord\_pres\_splay\ path = forAll\ orderedBTree\ (isOrd \cdot (splay\ path))$$

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 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:

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

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 $inBdt$, $outBdt$, $baseBdt$, $(\!|\cdot|\!)$, e $[\![\cdot]\!]$.

2. Apresentar no relatório o diagrama de $[\![\cdot]\!]$.

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.*

   $$prop\_pres\_tips :: Bdt\ Int \rightarrow Bool$$
   $$prop\_pres\_tips = tipsBdt \equiv tipsLTree \cdot extLTree$$

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.*

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

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

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

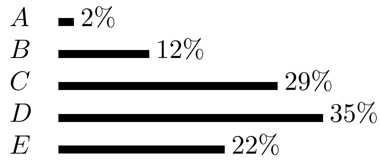
Problema 4

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.

$$d3 = normal\ [10..20]$$

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

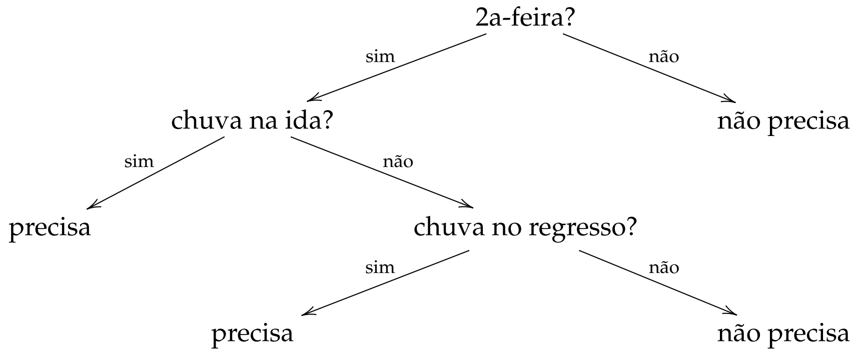
$$(f \bullet g)\ a = [(y, q * p) \mid (x, p) \leftarrow g\ a, (y, q) \leftarrow f\ x]$$

em que $g : A \rightarrow \mathsf{Dist}\ B$ e $f : B \rightarrow \mathsf{Dist}\ C$ são funções **monádicas** que representam *computações probabilísticas*. Este mónade é adequado à resolução de problemas de *probabilidades e estatística* usando programação funcional, de forma elegante e como caso particular 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 [?].

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

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

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

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

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.

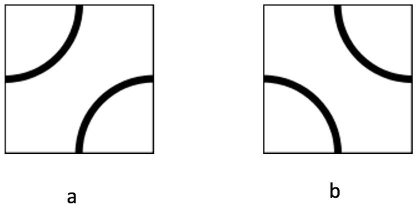


Figura 5: Os dois ladrilhos de Truchet-Smith.

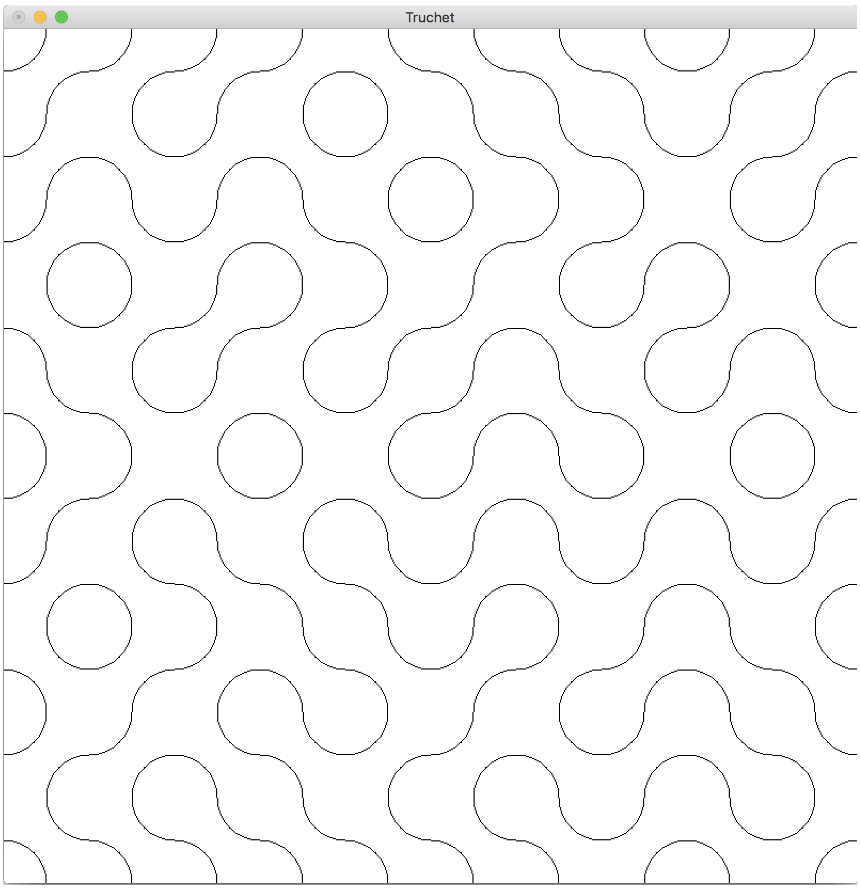


Figura 6: Um mosaico de Truchet-Smith.

Anexos

A Como exprimir cálculos e diagramas em LaTeX/lhs2tex

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

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

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

$$
\textbf{type}\ Dict = Exp\ String\ String
$$

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):

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

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 [?].

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:

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

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:

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

$$(genExp\ (n-1))\ (genExp\ (n-1))\ (genExp\ (n-1))))$$
$$]$$

$orderedBTree :: Gen\ (\mathsf{BTree}\ Int)$
$orderedBTree = liftM\ (foldr\ insOrd\ Empty)\ (QuickCheck.arbitrary :: Gen\ [Int])$

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

Outras funções auxiliares

Lógicas:

$\mathbf{infixr}\ 0 \Rightarrow$
$(\Rightarrow) :: (Testable\ prop) \Rightarrow (a \to Bool) \to (a \to prop) \to a \to Property$
$p \Rightarrow f = \lambda a \to p\ a \Rightarrow f\ a$

$\mathbf{infixr}\ 0 \Leftrightarrow$
$(\Leftrightarrow) :: (a \to Bool) \to (a \to Bool) \to a \to Property$
$p \Leftrightarrow f = \lambda a \to (p\ a \Rightarrow property\ (f\ a))\ .\&\&.\ (f\ a \Rightarrow property\ (p\ a))$

$\mathbf{infixr}\ 4 \equiv$
$(\equiv) :: Eq\ b \Rightarrow (a \to b) \to (a \to b) \to (a \to Bool)$
$f \equiv g = \lambda a \to f\ a \equiv g\ a$

$\mathbf{infixr}\ 4 \leqslant$
$(\leqslant) :: Ord\ b \Rightarrow (a \to b) \to (a \to b) \to (a \to Bool)$
$f \leqslant g = \lambda a \to f\ a \leqslant g\ a$

$\mathbf{infixr}\ 4 \wedge$
$(\wedge) :: (a \to Bool) \to (a \to Bool) \to (a \to Bool)$
$f \wedge g = \lambda a \to ((f\ a) \wedge (g\ a))$

Compilação e execução dentro do interpretador:[6]

$run = \mathbf{do}\ \{system$ `"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.

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

Problema 1

Atendendo ao tipo da função *discollect*, conseguimos percecer que ela consome o tipo de entrada para produzir o tipo de saída, mais especificamente, consome a lista que se encontra em cada par. Por isso, podemos definir a função com base num catamorfismo de listas, tal como se verifica em seguida.

$$
\begin{array}{l}
discollect :: (Ord\ b, Ord\ a) \Rightarrow [(b,[a])] \rightarrow [(b,a)] \\
discollect = cataList\ [nil, \mathsf{conc} \cdot (discollect\_pair \times id)]\ \mathbf{where} \\
\quad discollect\_pair\ (\_, [\,]) = [\,] \\
\quad discollect\_pair\ (b, (a : as)) = (b, a) : (discollect\_pair\ (b, as))
\end{array}
$$

$$
\begin{array}{l}
dic\_exp :: Dict \rightarrow [(String, [String])] \\
dic\_exp = collect \cdot tar
\end{array}
$$

Relativamente à definição da função *tar* à custa de um catamorfismo, olhemos primeiro para o seguinte diagrama que expressa o catamorfismo sobre o tipo *Exp* aplicado à função em causa.

$$
\begin{tikzcd}
Exp\ String\ String \arrow[r, "outExp"] \arrow[d, "tar"'] & Var\ String + Term\ String \times (Exp\ String\ String)^* \arrow[d, "id+id\times(\mathsf{map}\ \ tar)"] \\
(String, String)^* & Var\ String + Term\ String \times ((String, String)^*)^* \arrow[l, "{[g1,g2]}"]
\end{tikzcd}
$$

Uma vez que a função *untar* já estava definida, foi fácil encontrar o *g1*, para o caso em que o *outExp* retorna um Var *String*.

$$g1\ \ v = [(\texttt{""}, v)]$$

Para a função *g2*, no caso em que o *outExp* retorna um par $(Term\ String, [Exp\ String\ String])$, temos de ter em atenção que a parte recursiva do catamorfismo garante-nos que o elemento da direita do par já está tratado. Ou seja, aplicando $(\mathsf{map}\ \ tar)$ a $[Exp\ String\ String]$ obteremos uma lista de listas de pares $(String, String)$, onde no primeiro elemento do par falta a primeira letra, que neste caso é o *Term String* do par inicial. Então a função *g2* terá de ir a todas as sub-listas, e para cada sub-lista concatenar o *Term String* com a *String* da esquerda.

$$
\begin{array}{l}
tar = cataExp\ g\ \mathbf{where} \\
\quad g = [g1, g2]\ \mathbf{where} \\
\quad\quad g1\ \ v = [(\texttt{""}, v)] \\
\quad\quad g2\ (\_, [\,]) = [\,] \\
\quad\quad g2\ (s, (h : t)) = conc\_Term\ (s, h) +\!\!+\ g2\ (s, t)
\end{array}
$$

$$
\begin{array}{l}
conc\_Term :: (String, [(String, String)]) \rightarrow [(String, String)] \\
conc\_Term\ (\_, [\,]) = [\,] \\
conc\_Term\ (s, (s1, s2) : t) = (s +\!\!+\ s1, s2) : conc\_Term\ (s, t)
\end{array}
$$

Para definir a função *dic_rd*, tiramos partido da função $\widehat{lookup}$. Comecemos então por definir a função *uncurry_dic_rd*.

$$
\begin{array}{ll}
 & uncurry\_dic\_rd\ (p, t) = \widehat{lookup}\ (p, (dic\_exp\ \ t)) \\
\equiv & \quad \{\ (75)\ \} \\
 & uncurry\_dic\_rd\ (p, t) = \widehat{lookup}\ ((id \times dic\_exp)\ (p, t)) \\
\equiv & \quad \{\ (70)\ \} \\
 & uncurry\_dic\_rd\ (p, t) = (\widehat{lookup} \cdot (id \times dic\_exp))\ (p, t) \\
\equiv & \quad \{\ (69)\ \} \\
 & uncurry\_dic\_rd = (\widehat{lookup} \cdot (id \times dic\_exp))
\end{array}
$$

Concluimos então que:

$$dic\_rd = curry\ (\widehat{lookup} \cdot (id \times dic\_exp))$$

Por fim, definir a função dic_in tornou-se mais fácil após definir a função

$$\begin{array}{l}
insert_ps :: ((String, String), [(String, [String])]) \rightarrow [(String, [String])] \\
insert_ps\ ((p, s), [\,]) = [(p, [s])] \\
insert_ps\ ((p, s), (palavra, ls) : t) = \mathbf{if}\ (p \equiv palavra) \\
\quad \mathbf{then}\ (palavra, insert\ (s, ls)) : t\ \mathbf{else}\ (palavra, ls) : insert_ps\ ((p, s), t)\ \mathbf{where} \\
\quad insert\ (s, [\,]) = [s] \\
\quad insert\ (s, l@(h : t)) = \mathbf{if}\ (s \equiv h)\ \mathbf{then}\ l\ \mathbf{else}\ h : insert\ (s, t)
\end{array}$$

que dado um par de *strings* $(palavra, significado)$ e uma lista resultante de aplicar dic_exp, ou adiciona a palavra e o respetivo significado, caso seja uma palavra nova, ou atualiza a lista de significados da palavra, adicionando o respetivo significado.

Tendo em atenção que esta função é *uncurried*, uma vez feito o $insert\_ps$, podemos e devemos chamar a função dic_imp que retornará o dicionário com a adição pretendida. Como no procedimento anterior, comecemos por definir a função $uncurry\_dic\_in$.

$$\begin{array}{ll}
& uncurry_dic_in\ ((p, s), t) = (dic_imp\ (insert_ps\ ((p, s), dic_exp\ t))) \\
\equiv & \quad \{\ (75)\ \} \\
& uncurry_dic_in\ ((p, s), t) = (dic_imp\ (insert_ps\ ((id \times dic_exp)\ ((p, s), t)))) \\
\equiv & \quad \{\ (70)\ \} \\
& uncurry_dic_in\ ((p, s), t) = (dic_imp \cdot insert_ps \cdot (id \times dic_exp))\ ((p, s), t) \\
\equiv & \quad \{\ (69)\ \} \\
& uncurry_dic_in = dic_imp \cdot insert_ps \cdot (id \times dic_exp)
\end{array}$$

Concluimos então que:

$$dic\_in = curry\ (curry\ (dic\_imp \cdot insert\_ps \cdot (id \times dic\_exp)))$$

Contudo, apesar das funções dic_rd e dic_in estarem corretas, elas não tiram partido da representação em memória do Dicionário. Assim sendo, vamos redefinir as funções e vamos agora usar **hilomorfismos** sobre o tipo Exp para as redefinir.

O seguinte diagrama expressa o hilomorfismo auxiliar da função de procura dic_rd (uncurry).

$$\begin{tikzcd}
String \times Dict \arrow[r, "divide"] \arrow[d, "hylo_dic_rd"'] & (1 + String^*) + (String \times Dict)^* \arrow[d, "id + \mathsf{map}\ hylo_dic_rd"] \\
1 + String^* & (1 + String^*) + (1 + String^*)^* \arrow[l, "conquer"]
\end{tikzcd}$$

$$\begin{array}{l}
conquer_rd = [id, g]\ \mathbf{where} \\
\quad g\ l = \mathbf{if}\ \mathsf{length}\ (k\ l) \equiv 0\ \mathbf{then}\ i_1\ ()\ \mathbf{else}\ i_2\ (k\ l) \\
\quad k\ [\,] = [\,] \\
\quad k\ ((i_1\ ()) : t) = k\ t \\
\quad k\ ((i_2\ a) : t) = a +\!\!+\ k\ t \\
divide_rd = [alpha, beta] \cdot (distl + distl) \cdot distr \cdot (outList \times outExp) \\
alpha :: ((), String) + ((Char, String), String) \rightarrow (() + [String]) + [(String, Dict)] \\
alpha = [i_1 \cdot i_2 \cdot singl \cdot \pi_2, i_1 \cdot i_1 \cdot (!)] \\
beta :: ((), (String, [Dict])) + ((Char, String), (String, [Dict])) \rightarrow (() + [String]) + [(String, Dict)] \\
beta = [i_1 \cdot i_1 \cdot (!), k]\ \mathbf{where} \\
\quad k\ ((c, cs), (term, dicts)) = \mathbf{if}\ ([c] \equiv term \vee term \equiv \texttt{""}) \\
\quad\quad \mathbf{then}\ (i_2 \cdot discollect \cdot singl)\ (cs, dicts) \\
\quad\quad \mathbf{else}\ (i_1 \cdot i_1 \cdot (!))\ c
\end{array}$$

$$d\_rd :: (String, Dict) \rightarrow () + [String]$$
$$d\_rd = conquer\_rd \cdot (id + (\mathsf{map}\ \ d\_rd)) \cdot divide\_rd$$

$$toMaybe :: () + a \rightarrow Maybe\ a$$
$$toMaybe\ (i_1\ ()) = Nothing$$
$$toMaybe\ (i_2\ a) = Just\ a$$

$$hylo\_dic\_rd\ p\ d = (toMaybe \cdot d\_rd)\ (\texttt{"\ \ "} + p, d)$$

Em relação à função dic_in sob a forma de um hilomorfismo, tendo em conta que os dicionários estão normalizados, podemos assumir uma função de inserção do par (palavra, significado) na lista de sub-dicionários que sucede ao $Term$ `""`. Essa função é a função d_in, e os 3 casos possíveis de inserção são os seguintes:

1. A lista de dicionários chegou ao fim e a palavra é acrescentada no fim da lista como mais um dicionário;

2. A primeira letra da palavra não condiz com a do dicionário corrente e por isso há que preservar esse dicionário e continuar a percorrer a lista;

3. A primeira letra da palavra condiz com a letra corrente e por isso há que descer verticalmente para o resto da palavra e preservar o resto da lista de dicionários.

Foi também criada a função auxiliar $toDict$ que dado um par (palavra, significado) cria o respetivo Dicionário.

$$toDict :: (String, String) \rightarrow Dict$$
$$toDict = anaExp\ g\ \textbf{where}$$
$$\quad g\ (\texttt{""}, s) = i_1\ s$$
$$\quad g\ (p : ps, s) = i_2\ ([p], [(ps, s)])$$

$$d\_in :: ((String, String), [Dict]) \rightarrow [Dict]$$
$$d\_in\ ((p, s), []) = [toDict\ (p, s)]$$
$$d\_in\ ((\texttt{""}, s), d) = (Var\ s) : d$$
$$d\_in\ ((c : cs, s), (Var\ v) : d) = (Var\ v) : d\_in\ ((c : cs, s), d)$$
$$d\_in\ ((c : cs, s), (Term\ x\ xs) : d) = \textbf{if}\ ([c] \equiv x)$$
$$\quad \textbf{then}\ (Term\ x\ (d\_in\ ((cs, s), xs))) : d$$
$$\quad \textbf{else}\ (Term\ x\ xs) : d\_in\ ((c : cs, s), d)$$

$$hylo\_dic\_in\ p\ s\ (Term\ \texttt{""}\ d) = Term\ \texttt{""}\ (d\_in\ ((p, s), d))$$

Por último, atendendo ao facto da memória só ser 'habitada' por árvores de dicionários normalizados, após importação, e ao facto do Quickcheck não saber disso e começar a gerar representações em memória 'absurdas', foram implementadas as seguintes propriedades, para contornar esta situação.

$$valid\ t = t \equiv (dic\_imp \cdot dic\_norm \cdot dic\_exp)\ t$$

Propriedade [QuickCheck] 12 *Se um significado s de uma palavra p já existe num dicionário normalizado então adicioná-lo em memória não altera nada:*

$$prop\_dic\_red1\ p\ s\ d$$
$$\quad |\ d \not\equiv dic\_norm\ d = True$$
$$\quad |\ dic\_red\ p\ s\ d = dic\_imp\ d \equiv hylo\_dic\_in\ p\ s\ (dic\_imp\ d)$$
$$\quad |\ otherwise = True$$

Propriedade [QuickCheck] 13 *A operação dic_rd implementa a procura na correspondente exportação de um dicionário normalizado:*

$$prop\_dic\_rd1\ (p, t)$$
$$\quad |\ valid\ t = hylo\_dic\_rd\ p\ t \equiv lookup\ p\ (dic\_exp\ t)$$
$$\quad |\ otherwise = True$$

O resultado é satisfatório.

Problema 2

Para auxiliar a definição das funções $maisDir$ e $maisEsq$, foi definida a seguinte função

$$\begin{array}{l}
\mathit{isJust} :: \mathit{Maybe}\ a \rightarrow \mathit{Bool} \\
\mathit{isJust}\ (\mathit{Just}\ _) = \mathit{True} \\
\mathit{isJust}\ _ = \mathit{False}
\end{array}$$

que verifica se um dado a do tipo $Maybe$ existe na prática, ou seja, não é $Nothing$. Olhemos agora para o seguinte diagrama que expressa a função $maisDir$ como um catamorfismo de BTree.

$$\begin{tikzcd}
\mathsf{BTree}\ A \arrow[r, "outBTree"] \arrow[d, "maisDir"'] & 1 + A \times (\mathsf{BTree}\ A)^2 \arrow[d, "id + id \times maisDir^2"] \\
\mathit{Maybe}\ A & 1 + A \times (\mathit{Maybe}\ A)^2 \arrow[l, "{[g1, g2]}"]
\end{tikzcd}$$

Se a árvore for vazia ($Empty$), entao não há nenhum elemento mais à direita, pois não há nenhum elemento na sub-árvore, e portanto

$$g1 = \underline{\mathit{Nothing}}$$

Caso contrario, como a parte recursiva já trata de obter o elemento mais à direita de cada sub-árvore, basta apenas verificar se existe um elemento mais à direita na sub-árvore direita (daí a necessidade da função $isJust$) e caso exista, retorna-se esse mesmo valor. Caso contrário, a sub-árvore direita não tem nenhum elemento e, portanto, o valor mais à direita é a cabeça da árvore. Finalmente, a função $maisDir$ é dada por

$$\begin{array}{l}
\mathit{maisDir} = \mathit{cataBTree}\ g \\
\quad \mathbf{where}\ g = [\underline{\mathit{Nothing}}, k]\ \mathbf{where}\ k\ (a, (e, d)) = \mathbf{if}\ (\mathit{isJust}\ d)\ \mathbf{then}\ d\ \mathbf{else}\ \mathit{Just}\ a
\end{array}$$

A função $maisEsq$ define-se por analogia da função $maisDir$.

$$\begin{array}{l}
\mathit{maisEsq} = \mathit{cataBTree}\ g \\
\quad \mathbf{where}\ g = [\underline{\mathit{Nothing}}, k]\ \mathbf{where}\ k\ (a, (e, d)) = \mathbf{if}\ (\mathit{isJust}\ e)\ \mathbf{then}\ e\ \mathbf{else}\ \mathit{Just}\ a
\end{array}$$

As funções $insOrd$ e $isOrd$ são bastante fáceis de implementar uma vez definidas as funções $insOrd'$ e $isOrd'$ à custa de recursividade mútua. Definindo $insOrd'\ x = \langle insOrd\ x, id \rangle$ e $isOrd' = \langle isOrd, id \rangle$ então,

$$\begin{array}{l}
\mathit{insOrd}\ a\ x = \pi_1\ (\mathit{insOrd'}\ a\ x) \\
\mathit{isOrd} = \pi_1 \cdot \mathit{isOrd'}
\end{array}$$

Vamos agora definir as funções $insOrd'$ e $isOrd'$. Comecemos pela função $insOrd'$. Olhemos então para o seguinte diagrama que expressa o catamorfismo de BTree da função $insOrd'$.

$$\begin{tikzcd}
\mathsf{BTree}\ A \arrow[r, "outBTree"] \arrow[d, "insOrd'\ x"'] & 1 + A \times (\mathsf{BTree}\ A)^2 \arrow[d, "id + id \times (insOrd'\ x)^2"] \\
(\mathsf{BTree}\ A)^2 & 1 + A \times ((\mathsf{BTree}\ A)^2)^2 \arrow[l, "{[g1, g2]}"]
\end{tikzcd}$$

Para a função $g1$ (no caso em que a árvore é Empty), o resultado será um par onde o primeiro elemento do par é a árvore com o x inserido e o segundo elemento do par é a árvore inicial. É fácil concluir que

$$g1 = \langle \underline{(\mathit{emp}\ x)}, \underline{\mathit{Empty}} \rangle$$

Para o caso em que a árvore é um $Node$, temos de ter em atenção que a parte recursiva do catamorfismo já trata de inserir o x nas duas sub-árvores. Neste ponto, temos de ver, mediante o valor da cabeça da árvore original, qual sub-árvore é que usamos para reconstruir a árvore agora com o x inserido. Depois de olhar para a função definida, é mais fácil entender a lógica.

$$
\begin{aligned}
&insOrd'\ x = cataBTree\ g\ \mathbf{where}\\
&\quad g = [\langle \underline{(emp\ x)}, \underline{Empty} \rangle, \langle k, Node \cdot (id \times (\pi_2 \times \pi_2)) \rangle]\\
&\quad k\ (h, ((lx, l), (rx, r))) = \mathbf{if}\ (x < h)\ \mathbf{then}\ Node\ (h, (lx, r))\ \mathbf{else}\ Node\ (h, (l, rx))
\end{aligned}
$$

Neste caso em concreto, $Node \cdot (id \times (\pi_2 \times \pi_2))$ trata da preservação da árvore. Ou seja, no par final, o elemento da direita é a árvore original. Relativamente à função k, recordemos que ela recebe a cabeça da árvore (h) e um par de pares. O primeiro par tem a sub-árvore esquerda com o x inserido (lx) e a sub-árvore esquerda inalterada (l). O segundo par tem a sub-árvore direita com o x inserido (rx) e a sub-árvore direita inalterada (r). Se o elemento a inserir é menor que a cabeça, então queremos construir uma árvore com h na cabeça e lx e r como sub-árvores. O caso contrário é análogo.

A função $isOrd'$ segue o mesmo padrão. Ela retorna um par do tipo ($Bool \times \mathsf{BTree}$), onde a árvore é preservada na direita do par. Portanto, a lóica por trás desta função é semelhante à $insOrd'$. O diagrama do catarmofismo é também muito parecido, pelo que deverá ser da seguinte forma

$$
\begin{tikzcd}
\mathsf{BTree}\ A \arrow[r, "outBTree"] \arrow[d, "isOrd'"'] & 1 + A \times (\mathsf{BTree}\ A)^2 \arrow[d, "id + id \times isOrd'^2"] \\
Bool \times (\mathsf{BTree}\ A) & 1 + A \times (Bool \times (\mathsf{BTree}\ A))^2 \arrow[l, "{[g1, g2]}"]
\end{tikzcd}
$$

Para a função $g1$ (no caso em que a árvore é Empty), é fácil concluir que

$$g1 = \langle \underline{True}, \underline{Empty} \rangle$$

pois uma árvore vazia está ordenada. Quanto à função $g2$, vamos olhar primeiro para o código e depois usar as variáveis usadas para o explicar (tal como foi feito na função $insOrd'$).

$$
\begin{aligned}
&isOrd' = cataBTree\ g\ \mathbf{where}\\
&\quad g = [\langle \underline{True}, \underline{Empty} \rangle, \langle k, Node \cdot (id \times (\pi_2 \times \pi_2)) \rangle]\\
&\quad k\ (h, ((b1, tl), (b2, tr))) = b1 \wedge b2 \wedge (h *\!>= tl) \wedge (h *\!<= tr)
\end{aligned}
$$

$Node \cdot (id \times (\pi_2 \times \pi_2))$ preserva a árvore inicial, tal como na função $insOrd'$. Relativamente à função k, ela recebe a cabeça da árvore h e um par de pares $((b1, tl), (b2, tr))$, onde $b1$ e $b2$ são os booleanos que definem se as sub-árvores estão ordenadas e tl e tr são as sub-árvores esquerda e direita respetivamente. Para verificar se a árvore inicial está ordenada, então $b1$ e $b2$ têm de ser $True$, e a cabeça h tem de ser maior ou igual que todos os elementos da sub-árvore tl e menor ou igual do que todos os elementos da sub-árvore tr. Para auxiliar esta última condição, forma criadas as seguintes funções

$$
\begin{aligned}
&(*\!>=) :: (Ord\ a) \Rightarrow a \rightarrow \mathsf{BTree}\ a \rightarrow Bool\\
&a *\!>= Empty = True\\
&a *\!>= (Node\ (h, (l, r))) = (a \geqslant h) \wedge (a *\!>= l) \wedge (a *\!>= r)\\
&(*\!<=) :: (Ord\ a) \Rightarrow a \rightarrow \mathsf{BTree}\ a \rightarrow Bool\\
&a *\!<= Empty = True\\
&a *\!<= (Node\ (h, (l, r))) = (a \leqslant h) \wedge (a *\!<= l) \wedge (a *\!<= r)
\end{aligned}
$$

Outra forma de definir a função $isOrd'$ seria por tirar partido das funções $maisDir$ e $maisEsq$ definidas em cima. A mudança seria, obviamente, na função k, onde em vez de verificar se h é maior ou igual que todos os elementos da sub-árvore tl, verificar apenas se h é maior ou igual que o elemento mais à direita de tl, e em vez de verificar se h é menor ou igual que todos os elementos da sub-árvore tr, verificar apenas se h é menor ou igual que o elemento mais à esquerda de tr. Para isso foi necessário criar uma função que compara um dado a com um $Maybe\ a$, como se pode ver em seguida.

$$
\begin{aligned}
&(.>.) :: (Ord\ a) \Rightarrow a \rightarrow Maybe\ a \rightarrow Bool\\
&a\ .>.\ Nothing = True\\
&a\ .>.\ (Just\ b) = (a \geqslant b)\\
&(.<.) :: (Ord\ a) \Rightarrow a \rightarrow Maybe\ a \rightarrow Bool\\
&a\ .<.\ Nothing = True\\
&a\ .<.\ (Just\ b) = (a \leqslant b)\\
&isOrd' = cataBTree\ g\ \mathbf{where}
\end{aligned}
$$

$$g = [\langle \underline{True}, \underline{Empty} \rangle, \langle k, Node \cdot (id \times (\pi_2 \times \pi_2)) \rangle]$$
$$k\ (h, ((b1, tl), (b2, tr))) = b1 \wedge b2 \wedge (h \mathbin{.>.} (maisDir\ tl)) \wedge (h \mathbin{.<.} (maisEsq\ tr))$$

Para definir as funções de rotação, recorremos às imagens e aos *gifs* que se encontram na página do *Wikipedia*.

$$rrot\ Empty = Empty$$
$$rrot\ a@(Node\ (\_, (Empty, \_))) = a$$
$$rrot\ (Node\ (b, (Node\ (a, (alpha, beta)), gamma))) = Node\ (a, (alpha, Node\ (b, (beta, gamma))))$$
$$lrot\ Empty = Empty$$
$$lrot\ a@(Node\ (\_, (\_, Empty))) = a$$
$$lrot\ (Node\ (a, (alpha, Node\ (b, (beta, gamma))))) = Node\ (b, (Node\ (a, (alpha, beta)), gamma))$$

Por fim, o seguinte diagrama expressa a função de *splay* à custa de um catamorfismo de BTree.

$$\begin{tikzcd}
\mathsf{BTree}\ A \arrow[r, "outBTree"] \arrow[d, "splay"'] & 1 + A \times (\mathsf{BTree}\ A)^2 \arrow[d, "id+id\times splay^2"] \\
(\mathsf{BTree}\ A)^{Bool^*} & 1 + A \times ((\mathsf{BTree}\ A)^{Bool^*})^2 \arrow[l, "{[g1, g2]}"]
\end{tikzcd}$$

Se a árvore for vazia, a função retorna a árvore vazia independentemente da lista de booleanos que recebe. Se a árvore não for vazia, então temos de ver os dois casos possiveis. O primeiro caso é quando a lista de booleanos é vazia. Neste caso, a função retorna a árvore inalterada, pois não há nenhuma rotação a fazer. O segundo caso é quando a lista de booleanos não é vazia. Desta forma, temos de de olhar primeiro para a cabeça da lista. Se a cabeça da lista for $True$, então o elemento ao qual queremos fazer *splay* está no filho da esquerda e portanto, tendo em conta que o catarmofismo já trata da parte recursiva por nós, ou seja, o elemento já está na cabeça do filho, temos apenas de fazer uma rotação à direita da árvore inicial. Obviamente, para que o catamorfismo trate da parte recursiva por nós, temos de passar a cauda da lista de booleanos ao filho da esquerda, pois é no filho da esquerda que está o elemento pretendido. O caso em que a cabeça da lista é $False$ é análogo. Assim, podemos definir a função *splay* da seguinte forma

$$splay\ l\ t = splaying\ t\ l$$
$$\begin{array}{l}
splaying = cataBTree\ [g1, g2]\ \mathbf{where} \\
\quad g1\ _\ _ = Empty \\
\quad g2\ (h, (f, g))\ [\,] = Node\ (h, (f\ [\,], g\ [\,])) \\
\quad g2\ (h, (f, g))\ (x : y) = \mathbf{if}\ x \\
\qquad \mathbf{then}\ rrot\ (Node\ (h, (f\ y, g\ [\,]))) \\
\qquad \mathbf{else}\ lrot\ (Node\ (h, (f\ [\,], g\ y)))
\end{array}$$

Problema 3

A estrutura de uma *árvore de decisão binária* é muito parecida com a estrutura de uma LTree. A diferença é que a *árvore de decisão binária* guarda uma $String$ na 'cabeça' do 'Fork', que neste caso tem o nome de *Query*. Ou seja, uma *árvore de decisão binária* tem decisões nas suas folhas e *queries* nos seus 'forks'. Portanto, é fácil perceber o **in** e o out de uma Bdt:

$$
\begin{tikzcd}
Bdt\ A \arrow[r, bend left, "outBdt"] & A + String \times (Bdt\ A)^2 \arrow[l, bend left, "inBdt"]
\end{tikzcd}
$$

Para a correta resolução dos exercícios deste problema, foi necessário implementar as seguintes funções:

$$inBdt = [Dec, Query]$$
$$outBdt\ (Dec\ a) = i_1\ a$$
$$outBdt\ (Query\ (s, (b1, b2))) = i_2\ (s, (b1, b2))$$
$$baseBdt\ f\ g = f + (id \times (g \times g))$$
$$recBdt\ f = baseBdt\ id\ f$$
$$(\!|g|\!) = g \cdot (recBdt\ (\!|g|\!)) \cdot outBdt$$
$$[\!(a)\!] = inBdt \cdot (recBdt\ [\!(a)\!]) \cdot a$$

O seguinte diagrama corresponde ao diagrama genérico de um anamorfismo de *árvores de decisão binárias*:

$$
\begin{tikzcd}
Bdt\ A & A + String \times (Bdt\ A)^2 \arrow[l, "inBdt"'] \\
C \arrow[u, "k = [\!(g)\!]"] \arrow[r, "g"'] & A + String \times C^2 \arrow[u, "id + id \times k^2"']
\end{tikzcd}
$$

Depois da analogia feita entre LTree e Bdt, é mais fácil definir a função $extLTree$, através de um catamorfismo de Bdt. Para complementar a explicação, olhemos para o seguinte diagrama que expressa a função $extLTree$ à custa de um catamorfismo de Bdt.

$$
\begin{tikzcd}
Bdt\ A \arrow[r, "outBdt"] \arrow[d, "extLTree"'] & A + String \times (Bdt\ A)^2 \arrow[d, "id + id \times extLTree^2"] \\
\mathsf{LTree}\ A & A + String \times (\mathsf{LTree}\ A)^2 \arrow[l, "{[Leaf, Fork \cdot \pi_2]}"]
\end{tikzcd}
$$

Partindo de *Bdt A*, se a função *outBdt* retornar um *A*, então estamos perante uma decisão (folha no caso das *LTree*), e é usada a função *Leaf*. Caso contrário, se a função $outBdt$ retornar um par com uma $String$ no lado esquerdo e um par de $Bdt\ A$ no lado direito, então estamos perante uma *query*. A parte recursiva do catamorfismo já tratou de transformar o par de Bdt do lado direito para um par de LTree. Então, temos apenas de ignorar a $String$, e fazer um $Fork$ das LTree geradas.

$$
\begin{array}{l}
extLTree :: Bdt\ a \to \mathsf{LTree}\ a \\
extLTree = (\!|g|\!)\ \mathbf{where} \\
\quad g = [Leaf, Fork \cdot \pi_2]
\end{array}
$$

A função $navLTree$ é um pouco mais complicada. Esta função está ***curried*** o que dificulta um pouco a definição do catamorfismo. Olhemos então para o seguinte diagrama que explica a função definida à custa de um catamorfismo de LTree.

$$
\begin{tikzcd}
\mathsf{LTree}\ A \arrow[r, "outLTree"] \arrow[d, "navLTree"'] & A + (\mathsf{LTree}\ A)^2 \arrow[d, "id + navLTree^2"] \\
(\mathsf{LTree}\ A)^{Bool^*} & A + ((\mathsf{LTree}\ A)^{Bool^*})^2 \arrow[l, "{[g1, g2]}"]
\end{tikzcd}
$$

É fácil pensar que esta função navega por uma LTree. Mediante a informação contida na lista de *booleanos*, ou navegamos para a direita ou para a esquerda. É também fácil concluir que caso estejamos perante uma folha, é impossível navegar nela, porque não existem mais ramificações. Basta pensar que as folhas são o final de um determinado caminho. Chegando a elas, não podemos avançar mais. Portanto, independentemente da lista de booleanos que recebemos (o caminho), a função retorna a própria folha. Temos então o primeiro caso tratado:

$$
\begin{array}{ll}
& \mathit{g1}\ (a, l) = \mathit{Leaf}\ a \\
\equiv & \quad \{\ (77)\ \} \\
& \mathit{g1}\ (a, l) = \mathit{Leaf}\ (\pi_1\ (a, l)) \\
\equiv & \quad \{\ (70)\ \} \\
& \mathit{g1}\ (a, l) = (\mathit{Leaf} \cdot \pi_1)\ (a, l) \\
\equiv & \quad \{\ (69)\ \} \\
& \mathit{g1} = \mathit{Leaf} \cdot \pi_1
\end{array}
$$

Mas a função $\mathit{g1}$ está *uncurried* e a função $\mathit{navLTree}$ está *curried*, logo,

$$\mathit{g1} = (\mathit{curry}\ (\mathit{Leaf} \cdot \pi_1))$$

Falta agora definir o $\mathit{g2}$. Tendo em conta o diagrama apresentado em cima, sabemos que a função $\mathit{g2}$ será do tipo:

$$((\mathsf{LTree}\ A)^{Bool^*})^2 \xrightarrow{\ \ g2\ \ } (\mathsf{LTree}\ A)^{Bool^*}$$

Sabendo que o par resulta de aplicar recursivamente a função $\mathit{navLTree}$, então o primeiro argumento de $\mathit{g2}$ são funções que indicam, para uma dada lista de escolhas, qual é a árvore entregue. Desta forma, foi definida a seguinte função.

$$
\begin{array}{l}
\mathit{nav} :: ([\mathit{Bool}] \rightarrow \mathsf{LTree}\ a, [\mathit{Bool}] \rightarrow \mathsf{LTree}\ a) \rightarrow [\mathit{Bool}] \rightarrow \mathsf{LTree}\ a \\
\mathit{nav}\ (f, g)\ [\,] = \mathit{Fork}\ (f\ [\,], g\ [\,]) \\
\mathit{nav}\ (f, g)\ (h : t) = \mathbf{if}\ h\ \mathbf{then}\ f\ t\ \mathbf{else}\ g\ t
\end{array}
$$

Se a lista for vazia (não existir caminho), então a função retorna a árvore original que foi partida pelo $\mathit{outLTree}$. Caso contrário, mediante o valor da cabeça da lista, ou retornamos a árvore da direita ou a da esquerda, tendo em conta que as funções f e g constituem a parte recursiva do catamorfismo e por isso é lhes passada a cauda da lista de *booleanos* como argumento. A função final é dada por

$$
\begin{array}{l}
\mathit{navLTree} :: \mathsf{LTree}\ a \rightarrow ([\mathit{Bool}] \rightarrow \mathsf{LTree}\ a) \\
\mathit{navLTree} = \mathit{cataLTree}\ g \\
\quad \mathbf{where}\ g = [\mathit{curry}\ (\mathit{Leaf} \cdot \pi_1), \mathit{nav}]
\end{array}
$$

Problema 4

A função *bnavLTree* é muito parecida com a função já definida *navLTree*. A única diferença é que a *bnavLTree* navega por uma LTree com base numa BTree de booleanos. Por isso, foi feita apenas uma pequena alteração na função. Mas antes, olhemos para o seguinte diagrama, também muito parecido ao diagrama da *navLTree*.

$$\begin{tikzcd}
\mathsf{LTree}\ A \arrow[r, "outLTree"] \arrow[d, "bnavLTree"'] & A + (\mathsf{LTree}\ A)^2 \arrow[d, "id+(bnavLTree)^2"] \\
(\mathsf{LTree}\ A)^{\mathsf{BTree}\ Bool} & A + ((\mathsf{LTree}\ A)^{\mathsf{BTree}\ Bool})^2 \arrow[l, "{[g1,g2]}"]
\end{tikzcd}$$

A função $g1$ faz exatamente o mesmo que a função $g1$ do catamorfismo da *navLTree*, e portanto

$$g1 = curry\ (Leaf \cdot \pi_1)$$

Para a função $g2$, seguimos o mesmo método usado no problema 3. Foi definida a seguinte função

$$\begin{aligned}
&bnav :: (\mathsf{BTree}\ Bool \to \mathsf{LTree}\ a, \mathsf{BTree}\ Bool \to \mathsf{LTree}\ a) \to \mathsf{BTree}\ Bool \to \mathsf{LTree}\ a\\
&bnav\ (f, g)\ Empty = Fork\ (f\ Empty, g\ Empty)\\
&bnav\ (f, g)\ (Node\ (h, (l, r))) = \mathbf{if}\ h\ \mathbf{then}\ f\ l\ \mathbf{else}\ g\ r
\end{aligned}$$

Se a cabeça da árvore for *True*, então seguimos pelo filho da esquerda com a sub-árvore BTree de booleanos esquerda, ignorando a sub-árvore direita. Caso contrário, seguimos pelo filho da direita com a sub-árvore BTree de booleanos direita, ignorando a sub-árvore esquerda. Obtemos então a função final dada por

$$\begin{aligned}
&bnavLTree = cataLTree\ g\\
&\quad \mathbf{where}\ g = [curry\ (Leaf \cdot \pi_1), bnav]
\end{aligned}$$

Por fim, olhemos para o seguinte diagrama que expressa a função *pbnavLTree* à custa de um catamorfismo de LTree.

$$\begin{tikzcd}
\mathsf{LTree}\ A \arrow[r, "outLTree"] \arrow[d, "pbnavLTree"'] & A + (\mathsf{LTree}\ A)^2 \arrow[d, "id+pbnavLTree^2"] \\
(\mathsf{Dist}\ (\mathsf{LTree}\ A))^{\mathsf{BTree}\ (\mathsf{Dist}\ Bool)} & A + ((\mathsf{Dist}\ (\mathsf{LTree}\ A))^{\mathsf{BTree}\ (\mathsf{Dist}\ Bool)})^2 \arrow[l, "{[g1,g2]}"]
\end{tikzcd}$$

A função $g1$ é o caso onde estamos perante uma *Leaf*, e, tendo em conta que é impossível navegar numa folha, a função retorna a própria folha com total certeza, independentemente da árvore de booleanos passada como argumento. Deste modo, a função $g1$ é dada por

$$g1\ \ a\ \ t = return\ (Leaf\ \ a)$$

que corresponde a

$$g1\ \ a\ \ t = D\ [(Leaf\ \ a, 1)]$$

Para a função $g2$, tal como nos exemplos anteriores, foi definida a seguinte função

$$\begin{aligned}
&pbnav\ (f, g)\ Empty = (prod\ (f\ \ Empty)\ (g\ \ Empty)) \ggg (return \cdot Fork)\\
&pbnav\ (f, g)\ (Node\ (h, (l, r))) = Probability.cond\ h\ (f\ \ l)\ (g\ \ r)
\end{aligned}$$

No caso em que a árvore de distribuições é *Empty*, o objetivo é retornar toda a LTree com total certeza. Para isso, usamos a função *prod*, o *monadic binding* e a função de unidade (*return*). O primeiro passo é combinar as distribuições de cada LTree resultante do *outLTree* e, para isso, é feito

$$prod\ (f\ \ Empty)\ (g\ \ Empty)$$

que terá o tipo $\mathsf{Dist}\ (\mathsf{LTree}\ A)^2$. O segundo e último passo é definir uma função 'Fork probabilística' e aplicá-la ao resultado anterior para obter algo do tipo $\mathsf{Dist}\ (\mathsf{LTree}\ A)$. Isso é feito à custa do *monadic* *binding* e da função de unidade ($return$). Para entender melhor como proceder a esta solução, vejamos o seguinte diagrama.

$$
\begin{tikzcd}
\mathsf{Dist}^2\ (\mathsf{LTree}\ A) \arrow[d, "\mu"'] & \mathsf{Dist}\ (\mathsf{LTree}\ A)^2 \arrow[l, "\mathsf{Dist}\ f"'] \arrow[dl, "\gg\!= f"] \arrow[d, dotted, dash] \\
\mathsf{Dist}\ (\mathsf{LTree}\ A) & (\mathsf{LTree}\ A)^2 \arrow[l, "f = return \cdot Fork"]
\end{tikzcd}
$$

Assim, é fácil concluir

$$pbnav\ (f, g)\ Empty = (prod\ (f\ \ Empty)\ (g\ \ Empty)) \gg\!= (return \cdot Fork)$$

Por fim, quando a árvore de distribuições é um $Node$, relembremos que as funções f e g são as funções resultantes de aplicar recursivamente $pbnavLTree$ às sub-árvores esquerda (l) e direita (r), respetivamente. Assim, estamos perante uma distribuição condicional, pelo que as probabilidades na cabeça da árvore devem ser multiplicadas pelas probabilidades dos filhos respetivos. A biblioteca **Probability.hs** já oferece essa função ($cond$).

Desta forma, a função final é dada por

$$
\begin{array}{l}
pbnavLTree = cataLTree\ g \\
\quad \mathbf{where}\ g = [g1, pbnav]\ \mathbf{where}\ g1\ \ a\ \ t = return\ (Leaf\ \ a)
\end{array}
$$

Atendendo ao problema da Anita precisar de guarda-chuva, foi necessário construir as árvores de decisão e de booleanos para poder responder à pergunta *Deve a Anita levar guarda-chuva dada a situação descrita?*

$$
\begin{array}{l}
bdtAnita = Query\ (\texttt{"2a feira?"}, (Query\ (\texttt{"chuva na ida?"}, \\
\quad (Dec\ \texttt{"precisa"}, Query\ (\texttt{"chuva no regresso?"}, \\
\qquad (Dec\ \texttt{"precisa"}, Dec\ \texttt{"nao precisa"})))), Dec\ \texttt{"nao precisa"})) \\
ltreeAnita = extLTree\ bdtAnita \\
btreeAnita = Node\ (D\ [(True, 0.1428), (False, 0.8572)], (Node\ (D\ [(True, 0.8), (False, 0.2)], \\
\quad (Empty, Node\ (D\ [(True, 0.6), (False, 0.4)], (Empty, Empty)))), Empty))
\end{array}
$$

O resultado foi o seguinte:

```
$ pbnavLTree ltreeAnita btreeAnita
$ Leaf "nao precisa" 86.9%
$     Leaf "precisa" 13.1%
```

Concluimos que, provavelmente, a Anita não irá precisar de guarda-chuva.

Problema 5

Assumindo um mosaico de dimensões x por y, a estratégia consiste em produzir uma lista com $x * y$ ladrilhos, alternando os ladrilhos, permutar a lista, criar uma matriz x por y com a lista já 'baralhada', transladar os elementos da matriz para as respetivas posições no mosaico e, por fim, desenhar.

Desta forma, foram definidas as seguintes funções: $truchetList$ que cria uma lista de ladrilhos alternados; $truchetMatrix$ que, dada uma lista, transforma essa lista numa matriz; $truchetTranslateX$ que translada os ladrilhos para a posição correta no eixo do x; $truchetTranslateY$ que translada os ladrilhos para a posição correta no eixo do y; $mosaicMatrix$ que permuta a lista e constroi a matriz; $mosaic$ que aplica as translações à matriz, otendo o mosaico final; $display'$ que transforma o mosaico no tipo $Picture$ para poder ser desenhado.

$$truchet1 = Pictures\ [put\ (0,80)\ (Arc\ (-90)\ 0\ 40), put\ (80,0)\ (Arc\ 90\ 180\ 40)]$$

$$truchet2 = Pictures\ [put\ (0,0)\ (Arc\ 0\ 90\ 40), put\ (80,80)\ (Arc\ 180\ (-90)\ 40)]$$

$dimX :: Int$
$dimX = 10$

$dimY :: Int$
$dimY = 10$

-- janela para visualizar:

$janela = InWindow$
`"Truchet"` -- window title
$(80 * dimX, 80 * dimY)$ -- window size
$(100, 100)$ -- window position

-- defs auxiliares:

$put = \widehat{Translate}$

$truchetList\ 0 = [\,]$
$truchetList\ 1 = [truchet1]$
$truchetList\ (n+2) = truchet1 : truchet2 : truchetList\ n$

$truchetMatrix\ \_\ [\,] = [\,]$
$truchetMatrix\ n\ l = take\ n\ l : truchetMatrix\ n\ (drop\ n\ l)$

$truchetTranslateX\ \_\ \_\ \_\ [\,] = [\,]$
$truchetTranslateX\ n\ i\ j\ [h] = [put\ (80 * i, 0)\ h]$
$truchetTranslateX\ n\ i\ j\ (h1 : h2 : t) =$
$\quad put\ (i * 80, 0)\ h1 : put\ (j * 80, 0)\ h2 : truchetTranslateX\ (n-2)\ (i+1)\ (j-1)\ t$

$truchetTranslateY\ \_\ \_\ \_\ [\,] = [\,]$
$truchetTranslateY\ n\ i\ j\ [h] = [\mathsf{map}\ (put\ (0, 80 * i))\ h]$
$truchetTranslateY\ n\ i\ j\ (h1 : h2 : t) =$
$\quad \mathsf{map}\ (put\ (0, i * 80))\ h1 : \mathsf{map}\ (put\ (0, j * 80))\ h2 : truchetTranslateY\ (n-2)\ (i+1)\ (j-1)\ t$

-- mosaico de x * y ladrilhos:

$mosaicMatrix = \mathbf{do}\ \{m \leftarrow permuta\ (truchetList\ (dimX * dimY)); return\ (truchetMatrix\ dimX\ m)\}$

$mosaic = \mathbf{do}\ \{m \leftarrow mosaicMatrix;$
$\quad return\ (truchetTranslateY\ dimX\ 0\ (-1)\ (\mathsf{map}\ (truchetTranslateX\ dimX\ 0\ (-1))\ m))\}$

-- display:

$display' = \mathbf{do}\ \{m \leftarrow mosaic; return\ (pictures\ (concat\ m))\}$

$main :: \mathsf{IO}\ ()$
$main = \mathbf{do}\ \{d \leftarrow display'; display\ janela\ white\ d\}$

Executando a função $main$ com $dimX$ = $dimY$ = 10, o resultado será a figura 7. Por fim, como já foi tocado, o problema 5 é **escalável**, ou seja, foram definidas 'variáveis globais', $dimX$ e $dimY$, pelo que o resultado de gerar, por exemplo, um mosaico 6x8 será a figura 8.

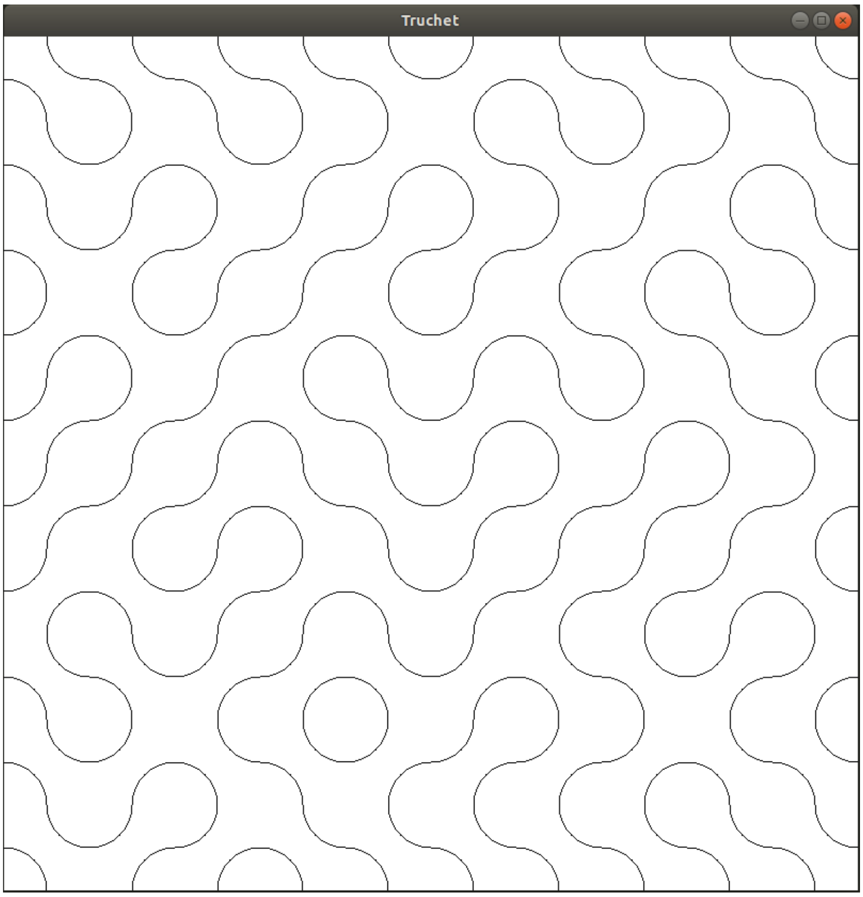


Figura 7: Um mosaico de Truchet-Smith de dimensões 10x10

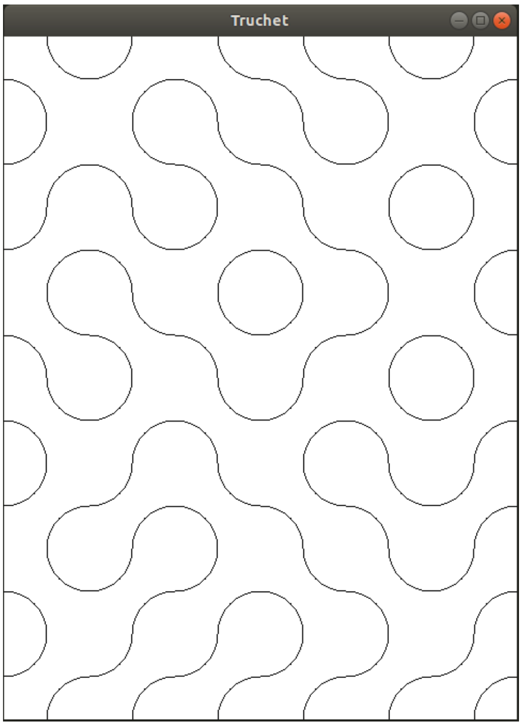


Figura 8: Um mosaico de Truchet-Smith de dimensões 6x8