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

Departamento de Informática
Universidade do Minho

Junho de 2020

Grupo nr.	35
a89556	Marco Avelino Teixeira Pereira
a89549	Alexandre Ferreira Gomes
a69856	Manuel Jorge Mimoso Carvalho

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:

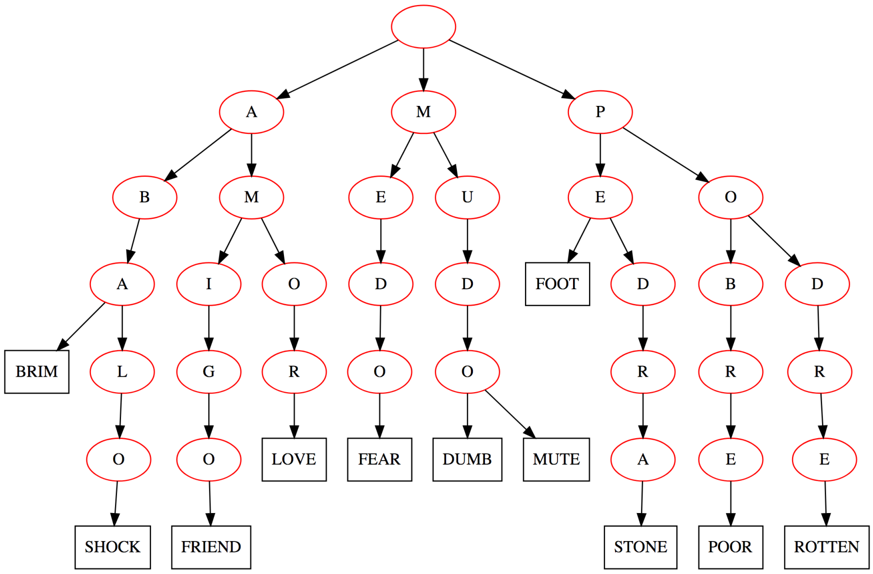


Figura 1: Representação em memória do dicionário dado para testes.

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:*

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

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

Propriedade [QuickCheck] 4 *Se um significado s de uma palavra p já existe num dicionário normalizado então adicioná-lo em memória não altera nada:*

$$\begin{array}{l} prop\_dic\_red1\ p\ s\ d \\ \quad \mid d \not\equiv dic\_norm\ d = True \\ \quad \mid dic\_red\ p\ s\ d = dic\_imp\ d \equiv dic\_in\ p\ s\ (dic\_imp\ d) \\ \quad \mid otherwise = True \end{array}$$

Propriedade [QuickCheck] 5 *A operação dic_rd implementa a procura na correspondente exportação de um dicionário normalizado:*

$$\begin{array}{l} prop\_dic\_rd1\ (p, t) \\ \quad \mid valid\ t = dic\_rd\ p\ t \equiv lookup\ p\ (dic\_exp\ t) \\ \quad \mid otherwise = True \end{array}$$

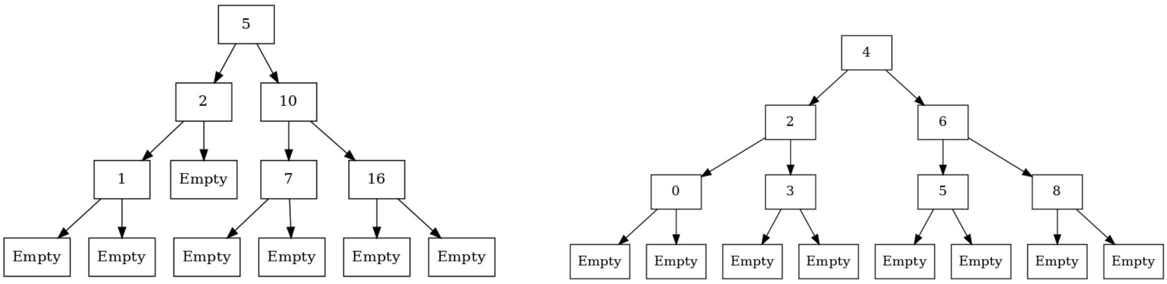


Figura 2: Duas árvores binárias de procura; a da esquerda vai ser designada por t_1 e a da direita por t_2.

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 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] 6 *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] 7 *O elemento mais à esquerda de uma árvore está presente no ramo da esquerda, a não ser que esse ramo esteja vazio:*

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

[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.

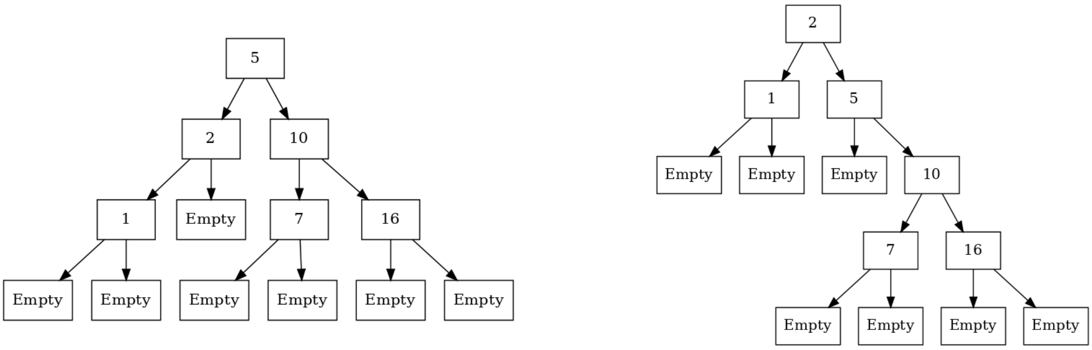


Figura 3: Exemplo de uma rotação à direita. A árvore da esquerda é a árvore original; a árvore da direita representa a rotação à direita correspondente.

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 $isOrd' = \langle isOrd, id \rangle$.

Propriedade [QuickCheck] 8 *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

$$rrot :: \mathsf{BTree}\ a \rightarrow \mathsf{BTree}\ a$$
$$lrot :: \mathsf{BTree}\ a \rightarrow \mathsf{BTree}\ a$$

de rotação à direita e à esquerda.

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

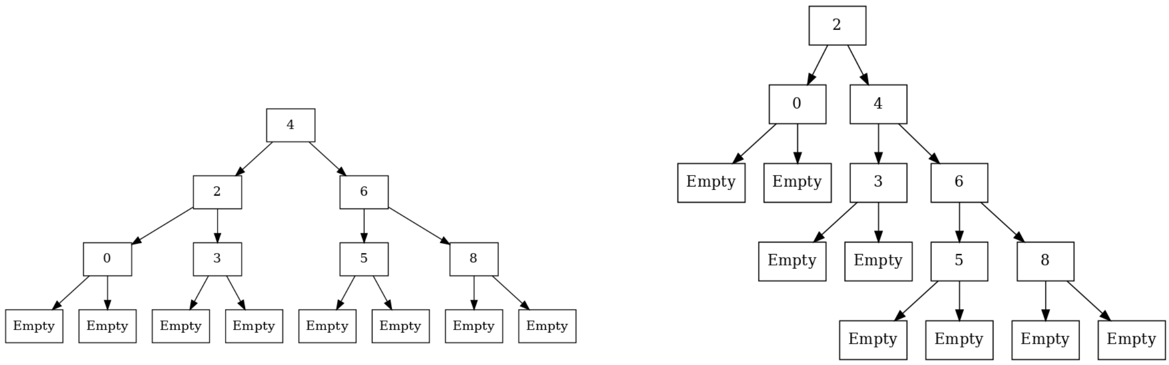


Figura 4: Exemplo de uma rotação à direita. A árvore da esquerda é a árvore original; a árvore da direita representa a rotação à direita correspondente.

Propriedade [QuickCheck] 9 *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

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

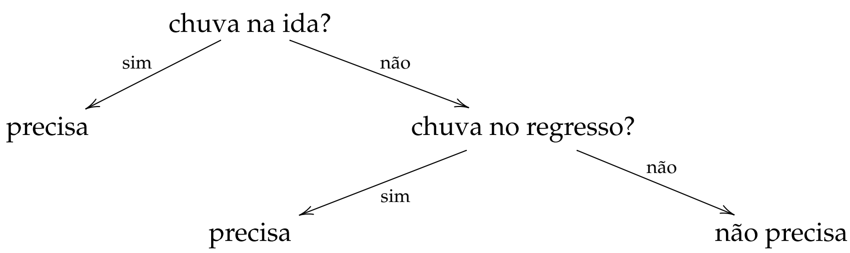
como um catamorfismo de listas. O argumento $[Bool]$ representa um caminho ao longo de uma árvore, em que o valor $True$ representa "seguir pelo ramo da esquerda"e o valor $False$ representa "seguir pelo ramo da direita". O caminho ao longo de uma árvore serve para *identificar* unicamente um nó dessa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

Propriedade [QuickCheck] 10 *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:

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

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] 11 *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] 12 *Percorrer uma árvore ao longo de um caminho é equivalente a percorrer a árvore inversa ao longo do caminho inverso.*

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

Propriedade [QuickCheck] 13 *Quanto mais longo for o caminho menos alternativas de fim irão existir.*

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

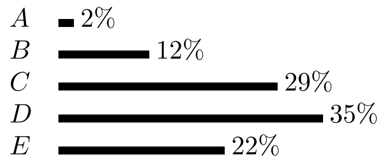
Problema 4

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



será representada pela distribuição

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

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)

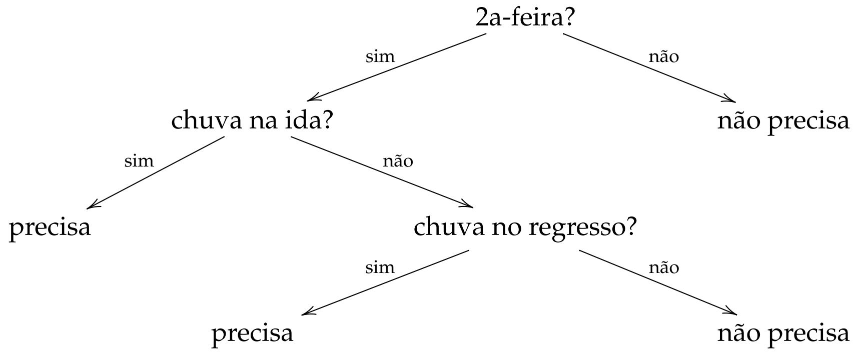
$$(f \bullet g)\ a = [(y, q * p) \mid (x, p) \leftarrow g\ a, (y, q) \leftarrow f\ x]$$

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

[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].

que a obriga a levar guarda-chuva (nos dias de chuva). Abaixo está apresentada a árvore de decisão 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 8 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.

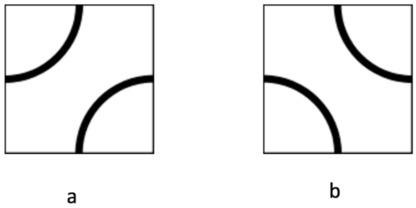


Figura 5: Os dois ladrilhos de Truchet-Smith.

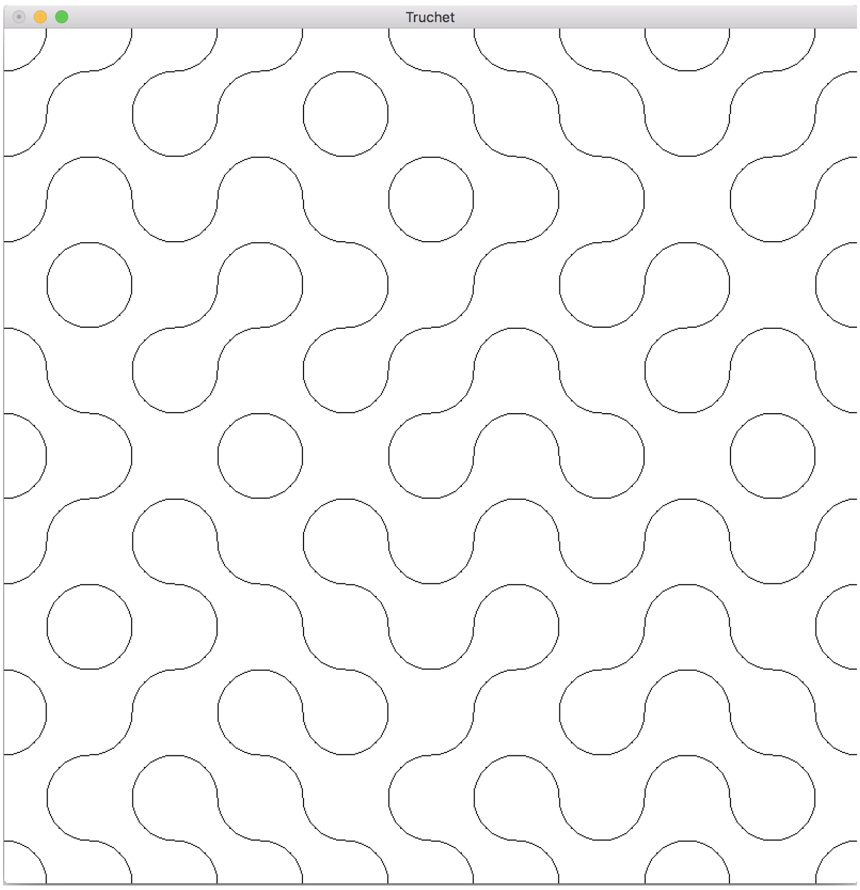


Figura 6: Um mosaico de Truchet-Smith.

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:

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

B Código fornecido

Problema 1

Função de representação de um dicionário:

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

onde

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

Dicionário para testes:

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

Normalização de um dicionário (remoção de entradas vazias):

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

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

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

[5] Exemplos tirados de [3].

Problema 2

Árvores usadas no texto:

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

Problema 3

Funções usadas para efeitos de teste:

$$tipsBdt :: Bdt\ a \rightarrow [a]$$
$$tipsBdt = cataBdt\ [singl, \widehat{(+\!\!+)} \cdot \pi_2]$$
$$tipsLTree = tips$$

Problema 5

Função de permutação aleatória de uma lista:

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

QuickCheck

Código para geração de testes:

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

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

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

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

Outras funções auxiliares

Lógicas:

$$\textbf{infixr}\ 0 \Rightarrow$$
$$(\Rightarrow) :: (Testable\ prop) \Rightarrow (a \to Bool) \to (a \to prop) \to a \to Property$$
$$p \Rightarrow f = \lambda a \to p\ a \Rightarrow f\ a$$
$$\textbf{infixr}\ 0 \Leftrightarrow$$
$$(\Leftrightarrow) :: (a \to Bool) \to (a \to Bool) \to a \to Property$$
$$p \Leftrightarrow f = \lambda a \to (p\ a \Rightarrow property\ (f\ a))\ .\&\&.\ (f\ a \Rightarrow property\ (p\ a))$$
$$\textbf{infixr}\ 4 \equiv$$
$$(\equiv) :: Eq\ b \Rightarrow (a \to b) \to (a \to b) \to (a \to Bool)$$
$$f \equiv g = \lambda a \to f\ a \equiv g\ a$$
$$\textbf{infixr}\ 4 \leqslant$$
$$(\leqslant) :: Ord\ b \Rightarrow (a \to b) \to (a \to b) \to (a \to Bool)$$
$$f \leqslant g = \lambda a \to f\ a \leqslant g\ a$$
$$\textbf{infixr}\ 4 \land$$
$$(\land) :: (a \to Bool) \to (a \to Bool) \to (a \to Bool)$$
$$f \land g = \lambda a \to ((f\ a) \land (g\ a))$$

Compilação e execução dentro do interpretador:[6]

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

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

discollect

A função discollect abaixo apresentada tira partido da utilização da "maquinaria monástica"modo a preencher cada par retornado com o respetivo valor.

$$discollect :: (Ord\ b, Ord\ a) \Rightarrow [(b, [a])] \to [(b, a)]$$
$$discollect\ d = Cp.cond\ null\ nil\ (head \ggg= \lambda(a, x) \to (return\ ([(a, b) \mid b \leftarrow x] +\!\!+ (discollect \cdot tail)\ d)))\ d$$

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

No entanto pode ser definida de uma forma mais "casual"e simples de entender como aquela abaixo apresentada:

$$
\begin{aligned}
& \mathit{discollectPW} :: (\mathit{Ord}\ b, \mathit{Ord}\ a) \Rightarrow [(b,[a])] \rightarrow [(b,a)] \\
& \mathit{discollectPW}\ [\,] = [\,] \\
& \mathit{discollectPW}\ ((a,x):y) = [(a,b) \mid b \leftarrow x] +\!\!+\ \mathit{discollectPW}\ y
\end{aligned}
$$

A função de exportação do dicionário usa a ja implementada função collect após a função tar ter retornado a lista com pares entre a palavra em português e uma lista de possíveis traduções desta.

dic_exp

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

tar

Como foi dito em cima aqui se encontra a função tar que utiliza como gene uma função que terá como input duas alternativas sendo elas o Var e o Term, logo o gene tem de ser um either. Sabendo isso temos apenas de encontrar os seus constituintes g1 e g2. O constituinte g1 será aplicado ao resultado de $\mathit{out}\ (\mathit{Var}\ v)$, tendo o seu retorno de ser uma lista com apenas um par em que o primeiro constituinte seria e o segundo seria a palavra traduzida a qual chegamos(v). O constituinte g2 será aplicado a $(\mathit{id} \times \mathsf{map}\ (|g|)) \cdot \mathit{out}$, a partir disto facilmente chegamos a conclusão que g2 será aplicado a (v,k) onde k é uma lista do tipo que temos de retornar e são os pares de traduções que se formam a partir daquele termo v, logo tudo o que temos de fazer é inserir v no inicio do primeiro membro de cada um dos pares da lista k concatenada, daí ficamos com $g2\ (v,k) = \mathsf{map}\ ((v+\!\!+) \times \mathit{id})\ (\mathit{concat}\ k)$.

$$
\begin{aligned}
& \mathit{tar} = \mathit{cataExp}\ g\ \mathbf{where} \\
& \quad g = [g1, g2]\ \mathbf{where} \\
& \qquad g1 = \mathit{singl} \cdot \langle \underline{\texttt{""}}, \mathit{id} \rangle \\
& \qquad g2\ (o,l) = \mathsf{map}\ ((o+\!\!+) \times \mathit{id})\ (\mathit{concat}\ l)
\end{aligned}
$$

dic_rd

Esta função foi o primeiro problema que tivemos de resolver sem receber informação alguma sobre que caminho seguir, pensando na tipagem desta função vemos que o primeiro argumento que ela recebe é uma *String*, que é como sabemos uma lista de *Char*, logo a primeira ideia que ocorre seria o uso de um cataList que percorra a lista de *Char* tal que quando a lista for vazia ele retorne as váriaveis que se encontra nesse local do *Dict*. No entanto para poder definir esta função decidimos criar uma outra auxiliar que seja parecida com a sequence mas que funcione com listas e as concatene ao invés de as juntar numa lista de listas e ainda que quando receba um Nothing, não faça com que o resultado seja automaticamente Nothing, ignorando-o apenas.

$$
\begin{aligned}
& \mathit{auxSequence} :: [\mathit{Maybe}\ [\mathit{String}]] \rightarrow \mathit{Maybe}\ [\mathit{String}] \\
& \mathit{auxSequence} = \mathit{cataList}\ [g1, g2] \\
& \quad \mathbf{where}\ g1 = \mathit{nothing} \\
& \qquad g2\ (\mathit{Just}\ a, \mathit{Just}\ t) = \mathit{Just}\ (a +\!\!+\ t) \\
& \qquad g2\ (\mathit{Nothing}, \mathit{Nothing}) = \mathit{Nothing} \\
& \qquad g2\ (\mathit{Just}\ a, \mathit{Nothing}) = \mathit{Just}\ a \\
& \qquad g2\ (\mathit{Nothing}, \mathit{Just}\ a) = \mathit{Just}\ a
\end{aligned}
$$

Tendo a nossa função auxiliar tivemos apenas que usar as definições aprendidas na UC para tornar fácil a descoberta de dic_rd.

$$
\begin{aligned}
& \quad \mathit{dic\_rd} = (|[g1, g2]|) \\
= & \quad \{ \text{ lei-43 } \}
\end{aligned}
$$

$$
\begin{aligned}
& dic\_rd \cdot inList = [g1, g2] \cdot F \; dic\_rd \\
= \quad & \{ \text{ def inBTree ; def F } \} \\
& dic\_rd \cdot [nil, cons] = [g1, g2] \cdot (id + (id \times dic\_rd)) \\
= \quad & \{ \text{ lei-20 ; lei-22 } \} \\
& [dic\_rd \cdot nil \; (splay \cdot cons), \cdot] = [g1, g2 \cdot (id \times dic\_rd)] \\
= \quad & \{ \text{ lei-27 } \} \\
& \begin{cases} dic\_rd \cdot nil = g1 \\ dic\_rd \cdot cons = g2 \cdot (id \times dic\_rd) \end{cases} \\
= \quad & \{ \text{ inserindo variáveis } \} \\
& \begin{cases} dic\_rd \cdot nil \; () = g1 \; () \\ dic\_rd \cdot cons \; (h, t) = g2 \cdot (id \times dic\_rd) \; (h, t) \end{cases} \\
= \quad & \{ \text{ lei-3 ; lei-75 } \} \\
& \begin{cases} dic\_rd \; [\,] = g1 \; () \\ dic\_rd \; (h : t) = g2 \cdot (h, (dic\_rd \; t)) \end{cases}
\end{aligned}
$$

Tendo chegado a isto temos agora que tentar deduzir as definições de g1 e g2, ora, g1 é como sempre o caso mais simples, caso estejamos perante uma lista vazia e um *Dict* Var vamos querer retornar o Just do singl da *String* que vem neste, caso contrário queremos retornar Nothing pois não há traduções desta palavra a menos que a String do termo seja uma String vazia. Quanto a g2 temos as mesmas hipóteses, caso esta receba uma Var vamos retornar Nothing pois não temos mais nada a percorrer no dicionário, caso receba um Term temos que verificar se a String deste é igual à cabeça da nossa lista de caracteres colocada em forma de lista, se isso for verdade mapeamos o segundo valor do par acima visto a cada um dos elementos da lista presente em Term caso contrário retornamos Nothing.

$$
\begin{aligned}
& dic\_rd = cataList \; [\underline{g1}, g2] \\
& \quad \textbf{where} \; g1 \; (Var \; v) = Just \; [v] \\
& \qquad g1 \; (Term \; o \; l) \mid o \equiv \texttt{""} = auxSequence \; (\mathsf{map} \;\; g1 \; l) \\
& \qquad\quad \mid otherwise = Nothing \\
& \qquad g2 \; (a, g2t) \; (Term \; o \; l) \mid o \equiv \texttt{""} = auxSequence \; (\mathsf{map} \;\; (g2 \; (a, g2t)) \; l) \\
& \qquad\quad \mid o \equiv [a] = auxSequence \; [b \mid b \leftarrow (\mathsf{map} \;\; g2t \; l), Nothing \not\equiv b] \\
& \qquad g2 \; \_ \; \_ = Nothing
\end{aligned}
$$

dic_in

A seguir temos a função de inserção no dicionário, esta função foi feita tirando partido da dica dada pelos docentes no FAQ, que foi o uso de um hylo. Abaixo encontra-se o diagrama de tipos de forma a fundamentar a explicação que iremos dar com mais coerência.

$$
\begin{tikzcd}
String \times Dict^{*} \arrow[r, "divideFunction"] \arrow[d, "{[\!(\,(divideFunction)\,)\!]}"'] & 1 + ((Dict^{*} + (Char \times Dict^{*})) \times (String \times Dict^{*})) \arrow[d, "id + id \times ([\!(divideFunction)\!])"] \\
(Dict^{*} + (Char \times Dict^{*}))^{*} \arrow[r, bend left, "out"] \arrow[d, "(\!|conquerFunction|\!)"'] & 1 + (Dict^{*} + (Char \times Dict^{*})) \times (Dict^{*} + (Char \times Dict^{*}))^{*} \arrow[l, bend left, "\mathbf{in}"] \arrow[d, "id + id \times (\!|conquerFunction|\!)"] \\
Dict^{*} & 1 + (Dict^{*} + (Char \times Dict^{*})) \times Dict^{*} \arrow[l, "conquerFunction"]
\end{tikzcd}
$$

A função desempenhada pelo anamorfismo é, recebendo a palavra a procurar no dicionário e a lista de vários dicionários onde adicionar, procurar onde inserir a palavra, retornando informações quanto a isso para que depois o catamorfismo tenha apenas de inserir a Var ou de criar os vários Term necessários para poder por fim inserir a Var.

A *divideFunction* vai fazer a maior parte do trabalho daí a sua grande extensão, irá inserir a direita um par em que o constituinte da esquerda é um Either tal que a inserção à direita deste significa que a palavra terminou e encontrou o lugar onde inserir e a inserção à esquerda significa que ainda não chegou ao fim a palavra mas encontrou o Term onde deverá inserir guardando o *Char* para o catamorfismo colocar no Term e a lista de dicionários que já não serão procurados pois já encontrou o que queria. O constituinte da direita é o resto da palavra e a lista de dicionários onde ainda procurar, reconstruindo o dicionário.

A *conquerFunction* irá receber a tradução da palavra e inserir consoante o outcome do anamorfismo, inserir simplesmente a Var caso não tenha de inserir Term e inserindo Term quando necessário através da auxiliar *cFaux*.

$$dic\_in\ p\ s\ (Term\ \texttt{""}\ v) = Term\ \texttt{""}\ (hyloList\ (conquerFunction\ s)\ divideFunction\ (p, v))$$

$$\begin{array}{l}
divideFunction :: (String, [Dict]) \to () + ([Dict] + (Char, [Dict]), (String, [Dict])) \\
divideFunction\ ((h : t), [\,]) = i_2\ ((i_2\ (h, [\,])), (t, [\,])) \\
divideFunction\ (\texttt{""}, d) = i_2\ (i_1\ d, (\texttt{""}, [\,])) \\
divideFunction\ ((h : t), ((Term\ k\ v) : ds))\ |\ (k \equiv [h]) = i_2\ (i_2\ (h, ds), (t, v)) \\
divideFunction\ ((h : t), (d : ds)) = ([i_1 \cdot id, i_2 \cdot ([i_1 \cdot (d{:}), i_2 \cdot (id \times (d{:}))] \times id)] \cdot divideFunction)\ ((h : t), ds)
\end{array}$$

$$conquerFunction :: String \to () + ([Dict] + (Char, [Dict]), [Dict]) \to [Dict]$$

$$conquerFunction\ t\ l = [[\underline{Var\ t}], (\widehat{cFaux\ t}) \cdot swap]\ l$$

$$\begin{array}{l}
cFaux :: String \to [Dict] \to ([Dict] + (Char, [Dict])) \to [Dict] \\
cFaux\ t\ l = [tt1\ t\ l, tt2\ t\ l] \\
\quad \mathbf{where}\ tt1\ t\ k\ d = (Var\ t) : d \\
\qquad tt2\ t\ k\ (l, r) = r +\!\!+ [Term\ [l]\ k]
\end{array}$$

Problema 2

maisDir e maisEsq

Para a resolução das funções maisDir e maisEsq utilizamos as várias regras aprendidas em Cálculo de Programas para obter a estrutura de g, que seria um either de g1 e g2. O raciocinio e as fórmulas de Cálculo de Programas usadas na resolução da função maisDir está apresentado em baixo.

$$\begin{array}{ll}
& maisDir = (\![g1, g2]\!) \\
= & \quad \{\ \text{lei-43}\ \} \\
& maisDir \cdot inBTree = [g1, g2] \cdot F\ maisDir \\
= & \quad \{\ \text{def inBTree ; def F}\ \} \\
& maisDir \cdot [\underline{Empty}, Node] = [g1, g2] \cdot (id + (id \times (maisDir \times maisDir))) \\
= & \quad \{\ \text{lei-20 ; lei-22}\ \} \\
& [maisDir \cdot \underline{Empty}, maisDir \cdot Node] = [g1, g2 \cdot (id \times (maisDir \times maisDir))] \\
= & \quad \{\ \text{lei-27}\ \} \\
& \left\{ \begin{array}{l} maisDir \cdot \underline{Empty} = g1 \\ maisDir \cdot Node = g2 \cdot (id \times (maisDir \times maisDir)) \end{array} \right. \\
= & \quad \{\ \text{inserindo variáveis}\ \} \\
& \left\{ \begin{array}{l} maisDir \cdot \underline{Empty}\ () = g1\ () \\ maisDir \cdot Node\ (a, (t1, t2)) = g2 \cdot (id \times (maisDir \times maisDir))\ (a, (t1, t2)) \end{array} \right.
\end{array}$$

$=$ { lei-3 ; lei-75 }

$$\begin{cases} maisDir\ Empty = g1\ () \\ maisDir\ (Node\ (a, (t1, t2))) = g2 \cdot (a, ((maisDir\ t1), (maisDir\ t2))) \end{cases}$$

A partir do sistema de equações ao qual chegamos conseguimos facilmente extrair g1 pois o node mais à direita de uma árvore vazia não existe logo Nothing.

Para g2 temos um contratempo pois não podemos apenas dizer que o resultado seria o obtido em *maisDir* t1 pois este pode retornar Nothing mas um simples pattern matching resolve essa situação.

Por isso decidimos criar a função abaixo apresentada que retorna o *Bool* True caso receba um *Maybe* com valor Nothing.

$$\begin{array}{l} isNothing :: Maybe\ a \rightarrow Bool \\ isNothing\ (Just\ a) = False \\ isNothing\ (Nothing) = True \end{array}$$

Ficamos assim com a função definida abaixo.

$$\begin{array}{l} maisDir = cataBTree\ g\ \mathbf{where} \\ \quad g = [g1, g2]\ \mathbf{where} \\ \qquad g1 = nothing \\ \qquad g2 = Cp.cond\ (isNothing \cdot \pi_2 \cdot \pi_2)\ (Just \cdot \pi_1)\ (\pi_2 \cdot \pi_2) \end{array}$$

Para a função maisEsq a forma de calcular é muito parecida mudando apenas o lado que vamos querer verificar se é Nothing, por isso decidimos apresentar apenas o diagrama de tipos para auxiliar na compreensão desta.

$$\begin{tikzcd} \mathsf{BTree} \arrow[d, "maisDir = (\!|g|\!)"'] \arrow[r, bend right, "out"'] & 1 + (A \times (\mathsf{BTree} \times \mathsf{BTree})) \arrow[l, bend right, "\mathbf{in}"'] \arrow[d, "id + id \times ((\!|g|\!) \times (\!|g|\!))"] \\ Maybe\ A & 1 + (A \times (Maybe\ A \times Maybe\ A)) \arrow[l, "g = [nothing, Cp.cond\ (isNothing \cdot \pi_1 \cdot \pi_2)\ (Just \cdot \pi_1)\ (\pi_1 \cdot \pi_2)]"] \end{tikzcd}$$

$$\begin{array}{l} maisEsq = cataBTree\ g\ \mathbf{where} \\ \quad g = [g1, g2]\ \mathbf{where} \\ \qquad g1 = nothing \\ \qquad g2 = Cp.cond\ (isNothing \cdot \pi_1 \cdot \pi_2)\ (Just \cdot \pi_1)\ (\pi_1 \cdot \pi_2) \end{array}$$

insOrd

Para a resolução da função abaixo recorremos ao enunciado, à equação dada que nos diz que

$$insOrd'\ x = \langle insOrd\ x, id \rangle$$

A partir desta conseguimos aplicar a lei de Fokkinga e outras para chegar a um estado em que se torna mais fácil a prespeção de o que cada função possa ser.

Como no formulário temos o nome que cada função toma como sendo h e k no lado direito da equação presente na fórmula de Fokkinga decidimos manter essa formulação criando assim hFunction. Dessa forma chegamos à definição que apresentamos abaixo e substituindo a definição apresentada de *insOrd'* por aquela que é dada no enunciado vemos que estas estão definidas com recursividade mútua. Sendo *insOrd'* uma função que retorna um par de *BTree* sendo o primeiro o uma forma de diferenciar em qual dos dois lados vai inserir retornarndo a *BTree* com a inserção no lado correto e o segundo a *BTree* que contem apenas os segundos elementos dos dois pares que recebe que são uma cópia do que era anteriormente para não perder informação. Desta forma em insOrd temos apenas que decidir qual dos lados queremos ao aplicar a mesma hFunction. Quando atingimos h com vazio sabemos que chegamos ao ponto onde devemos inserir o Node com a o valor a inserir e Empty nas suas duas árvores.

$$
\begin{array}{l}
insOrd'\ x = cataBTree\ g \\
\quad \textbf{where}\ g = \langle hFunction\ x, [\underline{Empty}, k2] \rangle \\
\qquad k2\ (a, ((t11, t12), (t21, t22))) = Node\ (a, (t12, t22)) \\
insOrd\ a = (hFunction\ a) \cdot recBTree\ (insOrd'\ a) \cdot outBTree \\
hFunction\ x = [h1\ x, h2\ x] \\
\quad \textbf{where}\ h1\ x\ () = Node\ (x, (Empty, Empty)) \\
\qquad h2\ x\ (a, ((t11, t12), (t21, t22)))\ |\ x \leqslant a = Node\ (a, (t11, t22)) \\
\qquad\quad |\ otherwise = Node\ (a, (t12, t21))
\end{array}
$$

isOrd

A função abaixo apresentada funciona de forma parecida à anterior usando *isOrd'* apenas para retornar um par em que o segundo elemento é uma forma de não perder informação sobre a árvore e o primeiro é o *Bool* qeu julgou se a árvore do seu par estava ou não ordenada. Da mesma forma que acima fizemos aqui podemos também simplesmente substituir a definição apresentada por *isOrd'* para que estas funções fiquem implementadas utilizando recursividade mútua de uma forma mais facilmente visível.

$$
\begin{array}{l}
isOrd' = cataBTree\ g \\
\quad \textbf{where}\ g = \langle hFunc, [\underline{Empty}, k2] \rangle \\
\qquad k2\ (a, ((b1, t1), (b2, t2))) = Node\ (a, (t1, t2)) \\
isOrd = hFunc \cdot recBTree\ (isOrd') \cdot outBTree \\
hFunc :: (Ord\ a) \Rightarrow () + (a, ((Bool, \mathsf{BTree}\ a), (Bool, \mathsf{BTree}\ a))) \rightarrow Bool \\
hFunc = [h1, h2] \\
\quad \textbf{where}\ h1\ () = True \\
\qquad h2\ (a, ((True, Empty), (True, Empty))) = True \\
\qquad h2\ (a, ((True, Empty), (True, t2))) = Just\ a \leqslant (maisEsq\ t2) \\
\qquad h2\ (a, ((True, t1), (True, Empty))) = Just\ a \geqslant (maisDir\ t1) \\
\qquad h2\ (a, ((True, t1), (True, t2))) = Just\ a \leqslant (maisEsq\ t2) \wedge Just\ a \geqslant (maisDir\ t1) \\
\qquad h2\ \_ = False
\end{array}
$$

rrot e lrot

A definição das próximas duas funções é bastante trivial tendo apenas que verificar se o lado que queremos fazer rotação tem um Node ou Empty.

$$
\begin{array}{l}
rrot\ (Node\ (a, ((Node\ (e_1, (t11, t12))), t2))) = Node\ (e_1, ((t11), (Node\ (a, (t12, t2))))) \\
rrot\ l = l \\
lrot\ (Node\ (a, (t1, (Node\ (e_2, (t21, t22)))))) = Node\ (e_2, ((Node\ (a, (t1, t21))), t22)) \\
lrot\ t = t
\end{array}
$$

splay

A definição de splay foi obtida por nós seguindo as formulas da Unidade Curricular sendo abaixo apresentada.

$$
\begin{array}{ll}
& splay = (\![[g1, g2]]\!) \\
= & \quad \{ \text{ lei-43 } \} \\
& splay \cdot inList = [g1, g2] \cdot F\ splay \\
= & \quad \{ \text{ def inBTree ; def F } \} \\
& splay \cdot [nil, cons] = [g1, g2] \cdot (id + (id \times splay)) \\
= & \quad \{ \text{ lei-20 ; lei-22 } \} \\
& [splay \cdot nil\ (splay \cdot cons), \cdot] = [g1, g2 \cdot (id \times splay)]
\end{array}
$$

$$
\begin{aligned}
= \quad & \{ \text{lei-27} \} \\
& \left\{ \begin{array}{l} splay \cdot nil = g1 \\ splay \cdot cons = g2 \cdot (id \times splay) \end{array} \right. \\
= \quad & \{ \text{inserindo variáveis} \} \\
& \left\{ \begin{array}{l} splay \cdot nil\ () = g1\ () \\ splay \cdot cons\ (h, t) = g2 \cdot (id \times splay)\ (h, t) \end{array} \right. \\
= \quad & \{ \text{lei-3 ; lei-75} \} \\
& \left\{ \begin{array}{l} splay\ [\,] = g1\ () \\ splay\ (h : t) = g2 \cdot (a, (splay\ t)) \end{array} \right.
\end{aligned}
$$

Depois de chegar a esta definição facilmente deduzimos que aplicar *splay* a uma lista vazia de *Bool* iremos retornar uma função que em nada altere a *BTree*, sendo essa funçao a identidade.

Para descobrir qual seria g2 tivemos de pensar um pouco mais mas seguindo a mesma ordem de pensamento conseguimos perceber que esta terá de retornar Empty a partir do momento em que a árvore que receba seja Empty e terá de retornar a função splay aplicada ao resto da lista para aplicar à *BTree* da esquerda caso a cabeça da lista seja o *Bool* true ou à *BTree* da direita caso seja false.

Podemos assim definir o diagrama de tipos que a seguir apresentamos:

$$
\begin{tikzcd}
Bool^{*} \arrow[r, bend right, "out"'] \arrow[d, "splay = (\!| g |\!)"'] & 1 + (Bool \times (Bool^{*})) \arrow[l, bend right, "\mathbf{in}"'] \arrow[d, "id + (id \times (\!| g |\!))"] \\
\mathsf{BTree}^{BTree} & 1 + (Bool \times \mathsf{BTree}^{BTree}) \arrow[l, "g = [\underline{id}, curry\ (Cp.cond\ ((Empty \equiv) \cdot \pi_2)\ func1\ func2)]"]
\end{tikzcd}
$$

$$
\begin{aligned}
& splay = cataList\ [\underline{id}, curry\ (Cp.cond\ ((Empty \equiv) \cdot \pi_2)\ func1\ func2)] \\
& \quad \mathbf{where} \\
& \quad\quad func1 = id \cdot \pi_2 \\
& \quad\quad func2 = Cp.cond\ (\pi_1 \cdot \pi_1)\ (Cp.ap \cdot (\pi_2 \times getNodEsq))\ (Cp.ap \cdot (\pi_2 \times getNodDir))
\end{aligned}
$$

Em baixo temos uma versão pointwise mais simples de entender.

$$
\begin{aligned}
& splayPW = cataList\ [\underline{id}, g2] \\
& \quad \mathbf{where} \\
& \quad g2\ \_\ Empty = Empty \\
& \quad g2\ (b1, sp2)\ (Node\ (a, (t1, t2)))\ \mid b1 \equiv True = sp2\ t1 \\
& \quad\quad \mid otherwise = sp2\ t2
\end{aligned}
$$

Problema 3

extLTree

O raciocinio para a *extLTree* foi percorrer a *Bdt* dada através de um catamorfismo destruindo a informação no primeiro membro do par Query guardando apenas no Fork as informações do segundo membro do par, e guardando em Leaf o representado em Dec. Desta forma ficamos com uma simples *LTree* onde apenas temos a decisão final consoante o caminho escolhido para atravessar a mesma. *extLTree* pode ser representada no seguinte diagrama:

$$\begin{tikzcd}
Bdt \arrow[d, "extLTree = (\!|g|\!)"'] \arrow[rr, bend right, "out"'] & & A + (A \times (Bdt \times Bdt)) \arrow[ll, bend right, "\mathbf{in}"'] \arrow[d, "id + (id \times ((\!|g|\!) \times (\!|g|\!)))"] \\
\mathsf{LTree} & & A + (A \times (\mathsf{LTree} \times \mathsf{LTree})) \arrow[ll, "g = [Leaf, Fork \cdot \pi_2]"']
\end{tikzcd}$$

$extLTree :: Bdt\ a \rightarrow \mathsf{LTree}\ a$
$extLTree = cataBdt\ [g1, g2]$ **where**
$\quad g1 = Leaf$
$\quad g2 = Fork \cdot \pi_2$

Definições do tipo de dados

$inBdt = [Dec, Query]$
$outBdt\ (Dec\ c) = i_1\ c$
$outBdt\ (Query\ d) = i_2\ d$

Padrões de recursividade

$baseBdt\ f\ g\ h = f + g \times (h \times h)$
$recBdt\ h = baseBdt\ id\ id\ h$
$cataBdt\ g = g \cdot recBdt\ (cataBdt\ g) \cdot outBdt$
$anaBdt\ g = inBdt \cdot recBdt\ (anaBdt\ g) \cdot g$

navLTree

A forma de pensar usada para esta função foi a utilização de um catamorfismo que irá percorrer a *LTree* dada, e consoante a lista de *Bool* a receber irá caso a cabeça desta lista seja true aplicar o resultado do catamorfismo da *LTree* da direita aplicado à cauda da lista de *Bool* à lista da direita e o contrário quando for False. Desta forma podemos definir navLTree no seguinte diagrama:

$$\begin{tikzcd}
\mathsf{LTree} \arrow[d, "navLTree = (\!|g|\!)"'] \arrow[rr, bend right, "out"'] & & A + (\mathsf{LTree} \times \mathsf{LTree}) \arrow[ll, bend right, "\mathbf{in}"'] \arrow[d, "id + ((\!|g|\!) \times (\!|g|\!))"] \\
\mathsf{LTree}^{Bool^*} & & A + A \times (\mathsf{LTree}^{Bool^*} \times \mathsf{LTree}^{Bool^*}) \arrow[ll, "g = [flip\ \underline{Leaf}, fFunction]"']
\end{tikzcd}$$

$navLTree :: \mathsf{LTree}\ a \rightarrow ([Bool] \rightarrow \mathsf{LTree}\ a)$
$navLTree = cataLTree\ g$
$\quad$ **where** $g = [g1, fFunction]$
$\qquad g1 = flip\ \underline{Leaf}$

$fFunction :: (([Bool] \rightarrow \mathsf{LTree}\ a), ([Bool] \rightarrow \mathsf{LTree}\ a)) \rightarrow [Bool] \rightarrow \mathsf{LTree}\ a$
$fFunction = curry\ (Cp.cond\ (null \cdot \pi_2)\ ff1\ ff2)$

$$
\begin{aligned}
&\textbf{where } \mathit{ff1} = \mathit{Fork} \cdot \langle \mathit{Cp.ap} \cdot (\pi_1 \times \mathit{nil}), \mathit{Cp.ap} \cdot (\pi_2 \times \mathit{nil}) \rangle \\
&\quad \mathit{ff2} = \mathit{Cp.cond}\ (\mathit{head} \cdot \pi_2)\ (\mathit{Cp.ap} \cdot (\pi_1 \times \mathit{tail}))\ (\mathit{Cp.ap} \cdot (\pi_2 \times \mathit{tail}))
\end{aligned}
$$

Em baixo temos uma versão pointwise que nos fornece um mais fácil entendimento do raciocinio.

$$
\begin{aligned}
&\mathit{navLTreePW} :: \mathsf{LTree}\ a \to ([\mathit{Bool}] \to \mathsf{LTree}\ a) \\
&\mathit{navLTreePW} = \mathit{cataLTree}\ g \\
&\quad \textbf{where } g = [\mathit{g1}, \mathit{fFunctionPW}] \\
&\qquad \mathit{g1}\ \ a = \underline{(\mathit{Leaf}\ \ a)}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{fFunctionPW} :: (([\mathit{Bool}] \to \mathsf{LTree}\ a), ([\mathit{Bool}] \to \mathsf{LTree}\ a)) \to [\mathit{Bool}] \to \mathsf{LTree}\ a \\
&\mathit{fFunctionPW}\ \ (\mathit{t1}, \mathit{t2})\ [\,] = \mathit{Fork}\ ((\mathit{t1}\ [\,]), (\mathit{t2}\ [\,])) \\
&\mathit{fFunctionPW}\ \ (\mathit{t1}, \mathit{t2})\ (h : t) \mid h = \mathit{t1}\ \ t \\
&\quad \mid \mathit{otherwise} = \mathit{t2}\ \ t
\end{aligned}
$$

Problema 4

Este problema consiste em colocar os alunos a prova com uma primeira função relativamente fácil de resolver pois segue o mesmo principio da função definida anteriormente *navLTree* e após isso aumentar a dificuldade para os alunos criarem uma mesma versão mas desta vez monadificada, usando a "maquinaria monástica".

bnvaLTree

Esta função foi realizada da mesma forma que a anterior sendo diferente apenas no facto que é uma *BTree* e não uma lista.

Para poder definir em pointfree decidimos criar as seguintes funções que apenas serão aplicadas a Node e que nos retornam cada um dos parametros deste como podemos ver abaixo.

$$
\begin{aligned}
&\mathit{getNodVal} :: \mathsf{BTree}\ a \to a \\
&\mathit{getNodVal}\ (\mathit{Node}\ (a, (\mathit{t1}, \mathit{t2}))) = a
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{getNodEsq} :: \mathsf{BTree}\ a \to \mathsf{BTree}\ a \\
&\mathit{getNodEsq}\ (\mathit{Node}\ (a, (\mathit{t1}, \mathit{t2}))) = \mathit{t1}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{getNodDir} :: \mathsf{BTree}\ a \to \mathsf{BTree}\ a \\
&\mathit{getNodDir}\ (\mathit{Node}\ (a, (\mathit{t1}, \mathit{t2}))) = \mathit{t2}
\end{aligned}
$$

Para complementar a nossa resposta encontra-se abaixo o diagrama de tipos:

$$
\begin{tikzcd}
\mathsf{LTree} \arrow[rr, bend right, "out"'] \arrow[d, "bnavLTree = (\!|g|\!)"'] & & A + (\mathsf{LTree} \times \mathsf{LTree}) \arrow[ll, bend right, "\mathbf{in}"'] \arrow[d, "id + ((\!|g|\!) \times (\!|g|\!))"] \\
\mathsf{LTree}^{BTree} & & A + A \times (\mathsf{LTree}^{BTree} \times \mathsf{LTree}^{BTree}) \arrow[ll, "g = [\mathit{flip}\ \underline{\mathit{Leaf}}, \mathit{bfFunction}]"]
\end{tikzcd}
$$

$$
\begin{aligned}
&\mathit{bnavLTree} = \mathit{cataLTree}\ g \\
&\quad \textbf{where } g = [\mathit{flip}\ \underline{\mathit{Leaf}}, \mathit{bfFunction}]
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{bfFunction} :: ((\mathsf{BTree}\ \mathit{Bool} \to \mathsf{LTree}\ a), (\mathsf{BTree}\ \mathit{Bool} \to \mathsf{LTree}\ a)) \to \mathsf{BTree}\ \mathit{Bool} \to \mathsf{LTree}\ a \\
&\mathit{bfFunction} = \mathit{curry}\ (\mathit{Cp.cond}\ ((\mathit{Empty}\ \equiv) \cdot \pi_2)\ \mathit{func1}\ \mathit{func2}) \\
&\quad \textbf{where } \mathit{func1} = \mathit{Fork} \cdot \langle \mathit{Cp.ap} \cdot (\pi_1 \times \mathit{id}), \mathit{Cp.ap} \cdot (\pi_2 \times \mathit{id}) \rangle \\
&\qquad \mathit{func2} = \mathit{Cp.cond}\ (\mathit{getNodVal} \cdot \pi_2)\ (\mathit{Cp.ap} \cdot (\pi_1 \times \mathit{getNodEsq}))\ (\mathit{Cp.ap} \cdot (\pi_2 \times \mathit{getNodDir}))
\end{aligned}
$$

Em baixo temos uma versão pointwise que permite uma compreensão mais simples e intuitiva do que aquela que apresentamos acima.

$$
\begin{array}{l}
bnavLTreePW = cataLTree\ g \\
\quad \mathbf{where}\ g = [g1, bfFunctionPW] \\
\qquad g1\ a = \underline{(Leaf\ a)}
\end{array}
$$

$$
\begin{array}{l}
bfFunctionPW :: ((\mathsf{BTree}\ Bool \to \mathsf{LTree}\ a), (\mathsf{BTree}\ Bool \to \mathsf{LTree}\ a)) \to \mathsf{BTree}\ Bool \to \mathsf{LTree}\ a \\
bfFunctionPW\ (t1, t2)\ Empty = Fork\ ((t1\ Empty), (t2\ Empty)) \\
bfFunctionPW\ (t1, t2)\ (Node\ (a, (tt1, tt2)))\ |\ a = t1\ tt1 \\
\quad |\ otherwise = t2\ tt2
\end{array}
$$

pbnavLTree

Para a resolução desta função recorremos ao uso de funções que encontramos enquanto estavamos a explorar o Probability.hs. Decidimos usar um catamorfismo para percorrer a *LTree* dada, em cada um dos nodos da *BTree* em que estavamos caso fosse Empty teriamos que retornar a mesma probabilidade a cada uma das possíveis decisões como seria de esperar e caso fosse um nodo utilizamos a função do módulo Probability que distribui as probabilidades para o lado esquerdo e direito da *LTree* consoante a probabilidade de o valor do nodo da *BTree* ser True ou False. Utilizando esta forma de pensar a definição da função torna-se simples sendo também simples definir o seu diagrama de tipos que a seguir apresentamos.

$$
\begin{tikzcd}
\mathsf{LTree} \arrow[rr, bend right, "out"'] \arrow[d, "pbnavLTree = (\!|g|\!)"'] & & A + (\mathsf{LTree} \times \mathsf{LTree}) \arrow[ll, bend right, "\mathbf{in}"'] \arrow[d, "id + ((\!|g|\!) \times (\!|g|\!))"] \\
\mathsf{Dist}\ (\mathsf{LTree})^{BTree} & & A + A \times (\mathsf{Dist}\ (\mathsf{LTree})^{BTree} \times \mathsf{Dist}\ (\mathsf{LTree})^{BTree}) \arrow[ll, "g = [flip\ \underline{(return \cdot Leaf)}, bpfFunction]"]
\end{tikzcd}
$$

$$
\begin{array}{l}
pbnavLTree = cataLTree\ g \\
\quad \mathbf{where}\ g = [g1, pbfFunctionPW] \\
\qquad g1 = flip\ \underline{(return \cdot Leaf)}
\end{array}
$$

$$
\begin{array}{l}
pbfFunctionPW\ (t1, t2)\ Empty = (\gg\!=)\ (prod\ (t1\ Empty)\ (t2\ Empty))\ (return \cdot Fork) \\
pbfFunctionPW\ (t1, t2)\ (Node\ (e, (l1, l2))) = Probability.cond\ e\ (t1\ l1)\ (t2\ l2)
\end{array}
$$

Tendo definida esta função podemos facilmente calcular as varias probabilidades e a resposta ao problema da Anita.

Seja decisao a *LTree* de decisão apresentada no início do problema e bTProbabilidade a *BTree* de probabilidades abaixo definidas:

$$
\begin{array}{l}
decisao = \\
\quad Query\ (\texttt{"2a-feira?"}, \\
\qquad ((Query\ (\texttt{"chuva na ida?"}, \\
\qquad\quad ((Dec\ \texttt{"precisa"}), \\
\qquad\quad (Query\ (\texttt{"chuva na volta?"}, \\
\qquad\qquad ((Dec\ \texttt{"precisa"}), (Dec\ \texttt{"nao precisa"})))))), \\
\qquad (Dec\ \texttt{"nao precisa"})))) \\
bTProbabilidade = \\
\quad Node\ ((D\ [(True, 1 / 7), (False, 6 / 7)]), \\
\qquad (Node\ ((D\ [(True, 0.8), (False, 0.2)]), \\
\qquad\quad (Empty,
\end{array}
$$

```
    Node ((D [(True, 0.6), (False, 0.4)]),
      (Empty, Empty)))),
  Empty))
```

Chamando o conjunto de funções obtemos o seguinte resultado tendo como podemos ver abaixo 86.9 % probabilidades de não precisar de levar guarda chuva maioritariamente devido a ter apenas de se preocupar com isso em 1 dos 7 dias da semana.

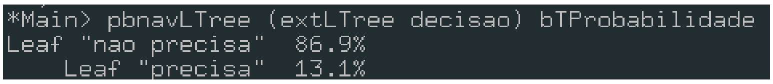


Figura 7: Resposta ao problema de Anita.

Problema 5

Pare resolver este problema, decidimos tirar proveito da função permuta que nos é fornecida.

```
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 ————-
```

$put = \widehat{Translate}$

Após o input do utilizador é gerado um número aleatório (n) entre 0 e o número total de ladrilhos necessários para fazer a imagem, depois é gerada uma lista com n ladrilhos do tipo truchet1 e total-n ladrilhos do tipo truchet2 a qual será aplicada a função permuta, ficamos assim com uma lista gerada aleatóriamente. Por fim criamos a função draw que é reponsável por desenhar cada peça no ponto na tela correto, esta função recebe para além da lista a desenhar as dimensões do mosaico e um acumulador que indica quando é necessário desenhar uma nova linha.

```
draw :: Int → Int → Int → [Picture] → [Picture]
draw _ _ _ [] = []
draw x y 0 l = (draw x (y − 1) x l)
draw x y i (h : t) = (put ((fromIntegral (i ∗ 80 − 480)), (fromIntegral (y ∗ 80 − 480))) h) : (draw x y (i − 1) t)
main :: IO ()
main = do
  putStrLn "Enter the width:"
  xi ← getLine
  putStrLn "Enter the height:"
  yi ← getLine
  let (x, y) = ((read xi :: Int), (read yi :: Int))
  n ← getStdRandom (randomR (0, x ∗ y))
  l ← permuta ((replicate ((x ∗ y) − n) truchet1) ++ (replicate n truchet2))
  display janela white (Pictures (draw x y x l))
  --
```

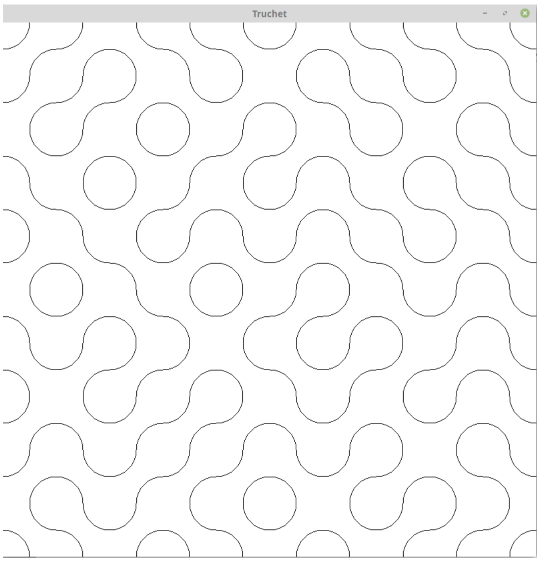


Figura 8: Resposta ao problema 5

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.