

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

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

Abra o ficheiro `cp1920t.lhs` no seu editor de texto preferido e verifique que assim é: todo o texto que se encontra dentro do ambiente

```
\begin{code}
...
\end{code}
```

vai ser seleccionado pelo **GHCI** para ser executado.

3 Como realizar o trabalho

Este trabalho teórico-prático deve ser realizado por grupos de três alunos. Os detalhes da avaliação (datas para submissão do relatório e sua defesa oral) são os que forem publicados na [página da disciplina](#) na *internet*.

Recomenda-se uma abordagem participativa dos membros do grupo de trabalho por forma a poderem responder às questões que serão colocadas na *defesa oral* do relatório.

Em que consiste, então, o *relatório* a que se refere o parágrafo anterior? É a edição do texto que está a ser lido, preenchendo o anexo **C** com as respostas. O relatório deverá conter ainda a identificação dos membros do grupo de trabalho, no local respectivo da folha de rosto.

Para gerar o PDF integral do relatório deve-se ainda correr os comando seguintes, que actualizam a bibliografia (com **BibTeX**) e o índice remissivo (com **makeindex**),

```
$ bibtex cp1920t.aux
$ makeindex cp1920t.idx
```

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

```
$ cabal install QuickCheck gloss
```

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

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

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

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

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

Problema 1

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

- *dic_rd* — procurar traduções para uma determinada palavra
- *dic_in* — inserir palavras novas (palavra e tradução)
- *dic_imp* — importar dicionários do formato “lista de pares palavra-tradução”
- *dic_exp* — exportar dicionários para o formato “lista de pares palavra-tradução”.

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

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



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

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

$$\text{prop_dic_rep } x = \text{let } d = \text{dic_norm } x \text{ in } (\text{dic_exp} \cdot \text{dic_imp}) d \equiv d$$

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

$$\begin{aligned} \text{prop_dic_red } p \ s \ d \\ | \text{ dic_red } p \ s \ d = \text{dic_imp } d \equiv \text{dic_in } p \ s \ (\text{dic_imp } d) \\ | \text{ otherwise} = \text{True} \end{aligned}$$

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

$$\text{prop_dic_rd } (p, t) = \text{dic_rd } p \ t \equiv \text{lookup } p \ (\text{dic_exp } t)$$

Problema 2

Árvores binárias (elementos do tipo **BTree**) são frequentemente usadas no armazenamento e procura de dados, porque suportam um vasto conjunto de ferramentas para procuras eficientes. Um exemplo de destaque é o caso das **árvores binárias de procura**, *i.e.* árvores que seguem o princípio de *ordenação*: para todos os nós, o filho à esquerda tem um valor menor ou igual que o valor no próprio nó; e de forma análoga, o filho à direita tem um valor maior ou igual que o valor no próprio nó. A Figura 2 apresenta dois exemplos de árvores binárias de procura.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 raiz tem o valor a , um filho s_1 à esquerda e um filho s_2 à direita. Assuma

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



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

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

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

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

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

Propriedade [QuickCheck] 4 As funções maisEsq e maisDir são determinadas unicamente pela propriedade

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

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

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

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

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

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

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

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

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

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

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



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



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

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

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

As árvores binárias providenciam uma boa maneira de reduzir o espaço de procura. Mas podemos fazer ainda melhor: podemos aproximar da raiz os elementos da árvore que são mais acedidos, reduzindo assim o espaço de procura na *dimensão vertical*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 raiz pela letra r . Se r não tem filhos à esquerda então simplesmente retornamos a árvore dada à entrada. Caso contrário,
2. designe o filho à esquerda pela letra l . A árvore que vamos retornar tem l na raiz, que mantém o filho à esquerda e adota r como o filho à direita. O orfão (*i.e.* o anterior filho à direita de l) passa a ser o filho à esquerda de r .

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

Comece então por implementar as funções

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

```

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

```

de rotação à direita e à esquerda.

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

```

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

```

De seguida implemente a operação de splaying

```

splay :: (Eq a) ⇒ [Bool] → (BTree a → BTree a)

```

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

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

```

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

```

Problema 3

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



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

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

```

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

```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Problema 4

Mónades são funtores com propriedades adicionais que nos permitem obter efeitos especiais em programação. Por exemplo, a biblioteca **Probability** oferece um mónade para abordar problemas de probabilidades. Nesta biblioteca, o conceito de distribuição estatística é captado pelo tipo

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

em que $ProbRep$ é um real de 0 a 1, equivalente a uma escala de 0 a 100%.

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



será representada pela distribuição

```
d1 :: Dist Char
d1 = D [('A', 0.02), ('B', 0.12), ('C', 0.29), ('D', 0.35), ('E', 0.22)]
```

que o **GHCI** mostrará assim:

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

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

```
d2 = uniform (words "Uma frase de cinco palavras")
```

isto é

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

distribuição *normais*, eg.

```
d3 = normal [10..20]
```

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

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

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

⁴Para mais detalhes ver o código fonte de **Probability**, que é uma adaptação da biblioteca **PHP** ("Probabilistic Functional Programming"). Para quem quiser souber mais recomenda-se a leitura do artigo [1].

respectiva a este problema.



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

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

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

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

```

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

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

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

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

Problema 5

Os **mosaicos de Truchet** são padrões que se obtêm gerando aleatoriamente combinações bidimensionais de ladrilhos básicos. Os que se mostram na figura 5 são conhecidos por ladrilhos de Truchet-Smith. A figura 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

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

Os diagramas podem ser produzidos recorrendo à *package* L^AT_EX *xymatrix*, por exemplo:

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

B Código fornecido

Problema 1

Função de representação de um dicionário:

$$\begin{aligned}
 dic_imp &:: [(String, [String])] \rightarrow Dict \\
 dic_imp &= Term " " \cdot \text{map } (bmap \text{ id singl}) \cdot \text{untar} \cdot \text{discollect}
 \end{aligned}$$

onde

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

Dicionário para testes:

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

Normalização de um dicionário (remoção de entradas vazias):

$$\begin{aligned}
 dic_norm &= collect \cdot filter \text{ p } \cdot \text{discollect} \textbf{ where} \\
 \text{p } (a, b) &= a > " " \wedge b > " "
 \end{aligned}$$

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

$$dic_red \text{ p s d } = (p, s) \in \text{discollect } d$$

⁵Exemplos tirados de [3].

Problema 2

Árvores usadas no texto:

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

Problema 3

Funções usadas para efeitos de teste:

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

Problema 5

Função de permutação aleatória de uma lista:

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

QuickCheck

Código para geração de testes:

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

```

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

```

Outras funções auxiliares

Lógicas:

```

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

```

Compilação e execução dentro do interpretador:⁶

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

C Soluções dos alunos

Os alunos devem colocar neste anexo as suas soluções aos exercícios propostos, de acordo com o “layout” que se fornece. Não podem ser alterados os nomes ou tipos das funções dadas, mas pode ser adicionado texto e/ou outras funções auxiliares que sejam necessárias.

Problema 1

discollect

A função discollect abaixo apresentada tira partido da utilização da notação do de modo a preencher cada par retornado com o respetivo valor.

```

discollect :: (Ord b, Ord a) => [(b, [a])] -> [(b, a)]
discollect d = Cp.cond null nil (do { (a, x) ← head; return [(a, b) | b ← x] ++ (discollect · tail) d }) d

```

⁶Pode 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:

```
discollectPW :: (Ord b, Ord a) => [(b, [a])] -> [(b, a)]
discollectPW [] = []
discollectPW ((a, x) : y) = [(a, b) | b <- x] ++ discollectPW y
```

A função de exportação do dicionário usa a já 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

```
dic_exp :: Dict -> [(String, [String])]
dic_exp = collect . tar
```

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 out (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 (id x map (|g|)) . 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 início do primeiro membro de cada um dos pares da lista k concatenada, daí ficamos com $g2(v, k) = \text{map } ((v++) \times id) (\text{concat } k)$.

```
tar = cataExp g where
  g = [g1, g2] where
    g1 = singl . ⟨" ", id⟩
    g2 (o, l) = map ((o++) × id) (concat l)
```

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 /textitString, que é como sabemos uma lista de /textitChar, logo a primeira ideia que ocorre seria o uso de um cataList que percorra a lista de /textitChar tal que quando a lista for vazia ele retorne as variáveis que se encontra nesse local do /textitDic. 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.

```
auxSequence :: [Maybe [String]] -> Maybe [String]
auxSequence = cataList [g1, g2]
  where g1 = nothing
        g2 (Just a, Just t) = Just (a ++ t)
        g2 (Nothing, Nothing) = Nothing
        g2 (Just a, Nothing) = Just a
        g2 (Nothing, Just a) = Just a
```

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

```
dic_rd = ([g1, g2])
= { lei-43 }
```

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

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. 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*.

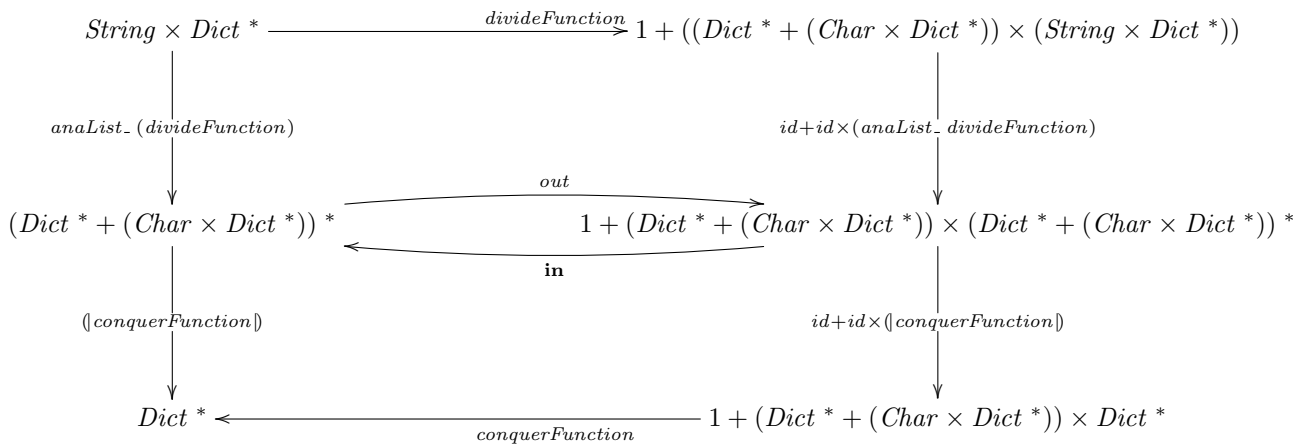
```

dic_rd = cataList [g1, g2]
where g1 (Var v) = Just [v]
      g1 (Term o l) = Nothing
      g2 (a, g2t) (Term o l) | o == "" = auxSequence (map (g2 (a, g2t)) l)
                             | o == [a] = auxSequence [b | b <- (map g2t l), Nothing <= b]
      g2 _ _ = Nothing

```

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.



A função desempenhada por `divideFunction` é recebendo a palavra a procurar no dicionário

```

dic.in p s (Term "" v) = Term "" (hyloList (conquerFunction s) divideFunction (p, v))
divideFunction :: (String, [Dict]) → () + ([Dict] + (Char, [Dict]), (String, [Dict]))
divideFunction ("", []) = i1 ()
divideFunction ((h : t), []) = i2 ((i2 (h, [])), (t, []))
divideFunction ("", d) = i2 (i1 d, ("", []))
divideFunction ((h : t), ((Term k v) : ds)) | (k ≡ [h]) = i2 (i2 (h, ds), (t, v))
divideFunction ((h : t), (d : ds)) = ([i1 · id, i2 · ([i1 · (d:), i2 · (id × (d:))] × id)] · divideFunction) ((h : t), ds)
conquerFunction :: String → () + ([Dict] + (Char, [Dict]), [Dict]) → [Dict]
conquerFunction t l = [[Var t], (cFaux t) · swap] l
cFaux :: String → [Dict] → ([Dict] + (Char, [Dict])) → [Dict]
cFaux t l = [tt1 t l, tt2 t l]
  where tt1 t k d = (Var t) : d
        tt2 t k (l, r) = r ++ [Term [l] k]

```

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 raciocínio e as fórmulas de Cálculo de Programas usadas na resolução da função `maisDir` está apresentado em baixo.

$$\begin{aligned}
\text{maisDir} &= ([g1, g2]) \\
&= \{ \text{lei-43} \} \\
\text{maisDir} \cdot \text{inBTree} &= [g1, g2] \cdot F \text{ maisDir} \\
&= \{ \text{def inBTree ; def F} \} \\
\text{maisDir} \cdot [\text{Empty}, \text{Node}] &= [g1, g2] \cdot (id + (id \times (\text{maisDir} \times \text{maisDir}))) \\
&= \{ \text{lei-20 ; lei-22} \} \\
[\text{maisDir} \cdot \text{Empty}, \text{maisDir} \cdot \text{Node}] &= [g1, g2 \cdot (id \times (\text{maisDir} \times \text{maisDir}))] \\
&= \{ \text{lei-27} \} \\
&= \begin{cases} \text{maisDir} \cdot \text{Empty} = g1 \\ \text{maisDir} \cdot \text{Node} = g2 \cdot (id \times (\text{maisDir} \times \text{maisDir})) \end{cases} \\
&= \{ \text{inserindo variáveis} \} \\
&= \begin{cases} \text{maisDir} \cdot \text{Empty} () = g1 () \\ \text{maisDir} \cdot \text{Node} (a, (t1, t2)) = g2 \cdot (id \times (\text{maisDir} \times \text{maisDir})) (a, (t1, t2)) \end{cases} \\
&= \{ \text{lei-3 ; lei-75} \} \\
&= \begin{cases} \text{maisDir Empty} = g1 () \\ \text{maisDir (Node (a, (t1, t2)))} = g2 \cdot (a, ((\text{maisDir } t1), (\text{maisDir } t2))) \end{cases}
\end{aligned}$$

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.

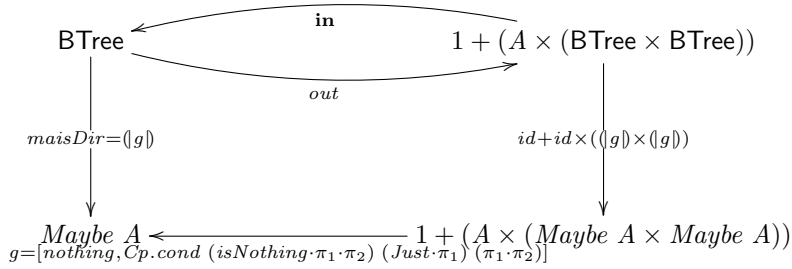
Por isso decidimos criar a função abaixo apresentada que retorna o `Bool True` caso receba um `Maybe` com valor `Nothing`.

$isNothing :: Maybe a \rightarrow Bool$
 $isNothing (Just a) = False$
 $isNothing (Nothing) = True$

Ficamos assim com a função definida abaixo.

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

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.



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

Para a resolução da função abaixo recorreremos 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 BTrees sendo o primeiro o uma forma de diferenciar em qual dos dois lados vai inserir retornando 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.

insOrd

$insOrd' x = cataBTree\ g$
 $\mathbf{where}\ g = \langle hFunction\ x, [Empty, k2] \rangle$
 $k2\ a, ((t11, t12), (t21, t22)) = Node\ (a, (t12, t22))$
 $insOrd\ a = (hFunction\ a) \cdot recBTree\ (insOrd'\ a) \cdot outBTree$
 $hFunction\ x = [h1\ x, h2\ x]$
 $\mathbf{where}\ h1\ x\ () = Node\ (x, (Empty, Empty))$
 $h2\ x\ a, ((t11, t12), (t21, t22)) \mid x \leq a = Node\ (a, (t11, t22))$
 $\mid otherwise = Node\ (a, (t12, t21))$

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* que 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.

```

isOrd' = cataBTree g
  where g = ⟨hFunc, [Empty, k2]⟩
        k2 (a, ((b1, t1), (b2, t2))) = Node (a, (t1, t2))
isOrd = hFunc · recBTree (isOrd') · outBTree
hFunc :: (Ord a) ⇒ () + (a, ((Bool, BTree a), (Bool, BTree a))) → Bool
hFunc = [h1, h2]
  where h1 () = True
        h2 (a, ((True, Empty), (True, Empty))) = True
        h2 (a, ((True, Empty), (True, t2))) = Just a ≤ (maisEsq t2)
        h2 (a, ((True, t1), (True, Empty))) = Just a ≥ (maisDir t1)
        h2 (a, ((True, t1), (True, t2))) = Just a ≤ (maisEsq t2) ∧ Just a ≥ (maisDir t1)
        h2 _ = False

```

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*.

```

rrot (Node (a, ((Node (e1, (t11, t12))), t2))) = Node (e1, ((t11), (Node (a, (t12, t2)))))
rrot l = l
lrot (Node (a, (t1, (Node (e2, (t21, t22)))))) = Node (e2, ((Node (a, (t1, t21))), t22))
lrot t = t

```

splay

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

```

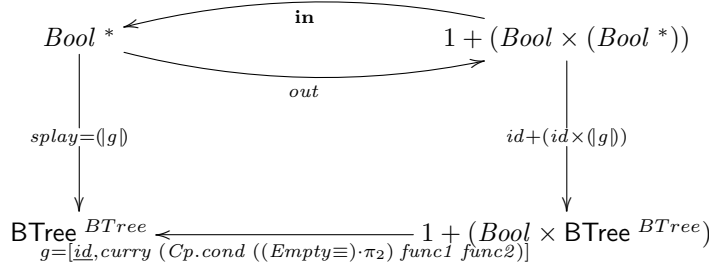
splay = ([g1, g2])
= { lei-43 }
splay · inList = [g1, g2] · F splay
= { def inBTree ; def F }
splay · [nil, cons] = [g1, g2] · (id + (id × splay))
= { lei-20 ; lei-22 }
[splay · nil (splay · cons), ·] = [g1, g2 · (id × splay)]
= { lei-27 }
{ splay · nil = g1
  splay · cons = g2 · (id × splay) }
= { inserindo variáveis }
{ splay · nil () = g1 ()
  splay · cons (h, t) = g2 · (id × splay) (h, t) }
= { lei-3 ; lei-75 }
{ splay [] = g1 ()
  splay (h : t) = g2 · (a, (splay t)) }

```

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 função 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:



```

splay = cataList [id, curry (Cp.cond ((Empty ≡) · π2) func1 func2)]
where
  func1 = id · π2
  func2 = Cp.cond (π1 · π1) (Cp.ap · (π2 × getNodEsq)) (Cp.ap · (π2 × getNodDir))

```

Em baixo temos uma versão pointwise mais simples de entender.

```

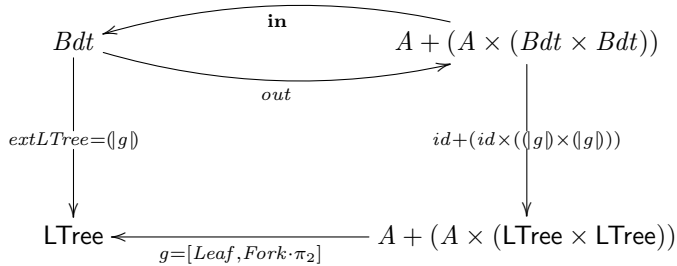
splayPW = cataList [id, g2]
where
  g2 _ Empty = Empty
  g2 (b1, sp2) (Node (a, (t1, t2))) | b1 ≡ True = sp2 t1
  | otherwise = sp2 t2

```

Problema 3

extLTree

O raciocínio para a *extLTree* foi percorrer a *Bdt* dada através de um cata 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:



```

extLTree :: Bdt a → LTree a
extLTree = cataBdt [g1, g2] where
  g1 = Leaf
  g2 = Fork · π2

```

Definições do tipo de dados

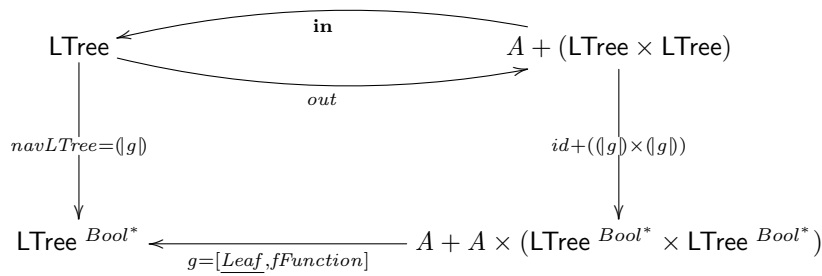
$$\begin{aligned} inBdt &= [Dec, Query] \\ outBdt (Dec\ c) &= i_1\ c \\ outBdt (Query\ d) &= i_2\ d \end{aligned}$$

Padrões de recursividade

$$\begin{aligned} baseBdt\ f\ g\ h &= f + g \times (h \times h) \\ recBdt\ h &= baseBdt\ id\ id\ h \\ cataBdt\ g &= g \cdot recBdt\ (cataBdt\ g) \cdot outBdt \\ anaBdt\ g &= inBdt \cdot recBdt\ (anaBdt\ g) \cdot g \end{aligned}$$

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:



```
navLTree :: LTree a -> ([ Bool ] -> LTree a)
navLTree = cataLTree g
  where g = [g1, fFunction]
        g1 = \. Leaf
fFunction :: ([ Bool ] -> LTree a), ([ Bool ] -> LTree a) -> [ Bool ] -> LTree a
fFunction = curry (Cp.cond (null . pi2) ff1 ff2)
  where ff1 = Fork . (Cp.ap . (pi1 x nil), Cp.ap . (pi2 x nil))
        ff2 = Cp.cond (head . pi2) (Cp.ap . (pi1 x tail)) (Cp.ap . (pi2 x tail))
```

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

```
navLTreePW :: LTree a -> ([ Bool ] -> LTree a)
navLTreePW = cataLTree g
  where g = [g1, fFunctionPW]
        g1 a = (Leaf a)
fFunctionPW :: ([ Bool ] -> LTree a), ([ Bool ] -> LTree a) -> [ Bool ] -> LTree a
fFunctionPW (t1, t2) [] = Fork ((t1 []), (t2 []))
fFunctionPW (t1, t2) (h : t) | h = t1 t
  | otherwise = t2 t
```

Problema 4

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

bnvalTree

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

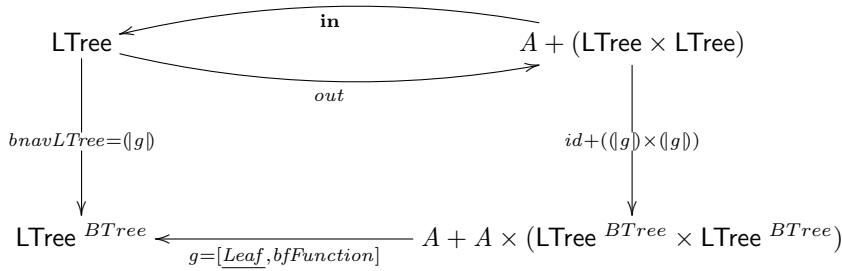
Para poder definir em pointfree decidimos criar as seguintes funções que apenas serão aplicadas a Node e que nos retornam cada um dos parametros deste como podemos ver abaixo.

```

getNodVal :: BTree a → a
getNodVal (Node (a, (t1, t2))) = a
getNodEsq :: BTree a → BTree a
getNodEsq (Node (a, (t1, t2))) = t1
getNodDir :: BTree a → BTree a
getNodDir (Node (a, (t1, t2))) = t2

```

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



```

bnavLTree = cataLTree g
  where g = [· · Leaf, bfFunction]
bfFunction :: ((BTree Bool → LTree a), (BTree Bool → LTree a)) → BTree Bool → LTree a
bfFunction = curry (Cp.cond ((Empty ≡) · π₂) func1 func2)
  where func1 = Fork · (Cp.ap · (π₁ × id), Cp.ap · (π₂ × id))
        func2 = Cp.cond (getNodVal · π₂) (Cp.ap · (π₁ × getNodEsq)) (Cp.ap · (π₂ × getNodDir))

```

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

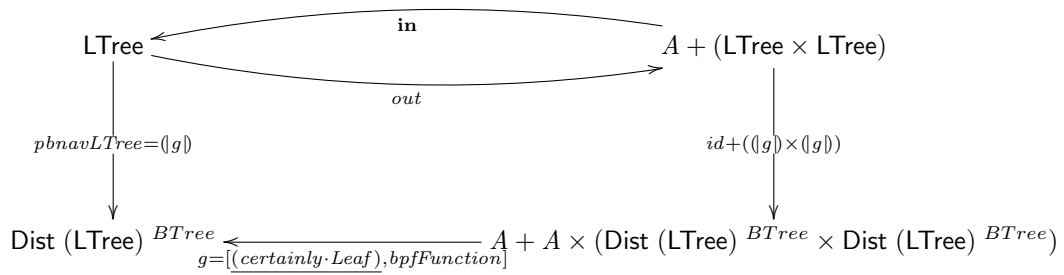
```

bnavLTreePW = cataLTree g
  where g = [g1, bfFunctionPW]
        g1 a = (Leaf a)
bfFunctionPW :: ((BTree Bool → LTree a), (BTree Bool → LTree a)) → BTree Bool → LTree a
bfFunctionPW (t1, t2) Empty = Fork ((t1 Empty), (t2 Empty))
bfFunctionPW (t1, t2) (Node (a, (tt1, tt2))) | a = t1 tt1
| otherwise = t2 tt2

```

pbnvalTree

Para a resolução desta função recorreremos ao uso de funções que encontramos enquanto estávamos a explorar o Probability.hs. Decidimos usar um catamorfismo para percorrer a LTree dada, em cada um dos nodos da BTree em que estávamos caso fosse Empty teríamos 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.



```

pbnavLTree = cataLTree g
  where g = [g1, pbfFunctionPW]
        g1 a = (certainly (Leaf a))
pbfFunctionPW (t1, t2) Empty = Probability.cond (choose 0.5 True False) (t1 Empty) (t2 Empty)
pbfFunctionPW (t1, t2) (Node (e, (l1, l2))) = Probability.cond e (t1 l1) (t2 l2)

```

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:

```

decisao =
  Query ("2a-feira?",
    ((Query ("chuva na ida?",
      ((Dec "precisa",
        (Query ("chuva na volta?",
          ((Dec "precisa"), (Dec "nao precisa"))))),
        (Dec "nao precisa")))))
  bTProbabilidade =
    Node ((D [(True, 1 / 7), (False, 6 / 7)]),
      (Node ((D [(True, 0.8), (False, 0.2)]),
        (Empty,
          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.

```

*Main> pbnavLTree (extLTree decisao) bTProbabilidade
Leaf "nao precisa"  86.9%
Leaf "precisa"     13.1%

```

Figura 7: Resposta ao problema de Anita.

Problema 5

```

truchet1 = Pictures [put (0, 80) (Arc (-90) 0 40), put (80, 0) (Arc 90 180 40)]
truchet2 = Pictures [put (0, 0) (Arc 0 90 40), put (80, 80) (Arc 180 (-90) 40)]
-- janela para visualizar:
janela = InWindow

```

```

"Truchet"    -- window title
(800,800)    -- window size
(100,100)    -- window position
-- defs auxiliares -----
put =  $\widehat{\text{Translate}}$ 
putTiles a b = (replicate a truchet1) ++ (replicate b truchet2)
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)), (fromIntegral (y * 80))) 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))
  r ← permuta (putTiles ((x * y) - n) n)
  display janela white (put ((-480), (-480)) (Graphics.Gloss.scale (10 / fromIntegral x) (10 / fromIntegral y) (P
--

```

Índice

L^AT_EX, [1](#)

bibtex, [2](#)

lhs2TeX, [1](#)

makeindex, [2](#)

Combinador “pointfree”

cata, [11](#), [14–19](#)

either, [12](#), [14–20](#)

Cálculo de Programas, [1](#), [2](#)

 Material Pedagógico, [1](#)

 BTree.hs, [3](#)

Functor, [4](#), [6–9](#), [12](#), [13](#), [15–17](#), [19](#), [20](#)

Função

π_1 , [11](#), [15](#), [16](#), [18](#), [19](#)

π_2 , [11](#), [12](#), [15–19](#)

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

map, [11](#), [14](#)

uncurry, [12](#), [14](#), [20](#)

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

 “Literate Haskell”, [1](#)

 Biblioteca

 PFP, [8](#)

 Probability, [7](#), [8](#)

 Gloss, [2](#), [9](#), [13](#)

 interpretador

 GHCi, [2](#), [8](#)

 Monad

 Random, [9](#)

 QuickCheck, [2](#)

Mosaico de Truchet, [9](#)

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

Programação literária, [1](#)

U.Minho

 Departamento de Informática, [1](#)

Referências

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