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

Departamento de Informática
Universidade do Minho

Junho de 2020

Grupo nr.	40
a87994	Daniel Ribeiro
a87986	Paulo Costa
a87989	Rui Baptista

1 Preâmbulo

A disciplina de Cálculo de Programas tem como objectivo principal ensinar a programação de computadores como uma disciplina científica. Para isso parte-se de um repertório de *combinadores* que formam uma álgebra da programação (conjunto de leis universais e seus corolários) e usam-se esses combinadores para construir programas *composicionalmente*, isto é, agregando programas já existentes.

Na sequência pedagógica dos planos de estudo dos dois cursos que têm esta disciplina, restringe-se a aplicação deste método à programação funcional em Haskell. Assim, o presente trabalho prático coloca os alunos perante problemas concretos que deverão ser implementados em Haskell. Há ainda um outro objectivo: o de ensinar a documentar programas, validá-los, e a produzir textos técnico-científicos de qualidade.

2 Documentação

Para cumprir de forma integrada os objectivos enunciados acima vamos recorrer a uma técnica de programação dita "literária" [**?**], cujo princípio base é o seguinte:

> *Um programa e a sua documentação devem coincidir.*

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

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

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

em que `lhs2tex` é um pre-processador que faz "pretty printing" de código Haskell em LaTeX e que deve desde já instalar executando

```
$ cabal install lhs2tex
```

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

```
$ ghci cp1920t.lhs
```

[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 **??** 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 **??** 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 **??** é dado um dicionário para testes, que corresponde à figura **??**. A implementação proposta deverá garantir as seguintes propriedades:

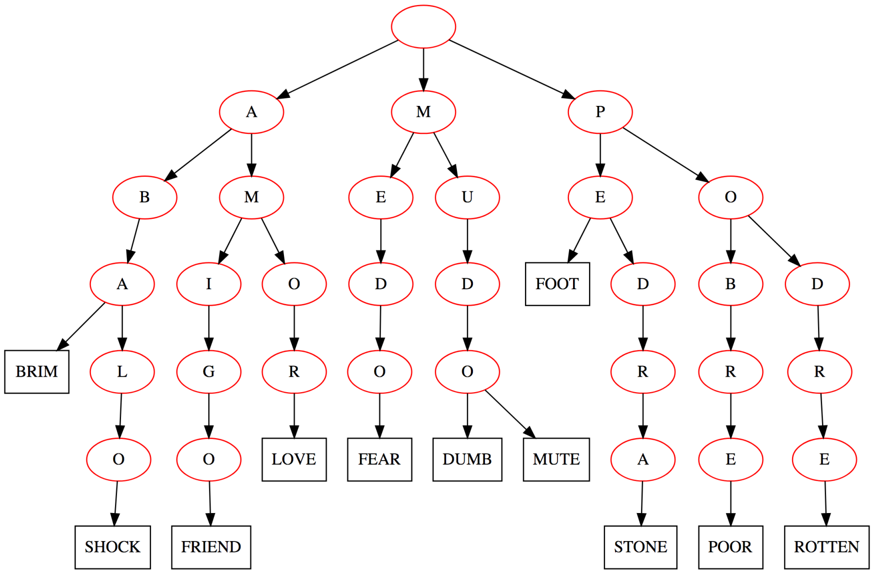


Figura 1: Representação em memória do dicionário dado para testes.

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

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

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

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

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

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

Problema 2

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

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

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

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

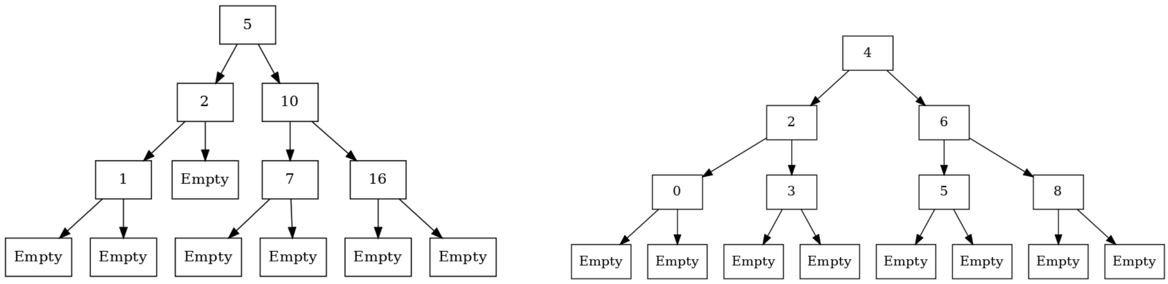


Figura 2: Duas árvores binárias de procura; a da esquerda vai ser designada por t_1 e a da direita por t_2.

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

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

Seguem alguns exemplos dos resultados que se esperam ao aplicar estas funções à árvore da esquerda ($t1$) e à árvore da direita ($t2$) da Figura **??**.

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

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

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

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

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

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

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

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

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

Para ambas as funções deve utilizar o que aprendeu sobre *catamorfismos e recursividade mútua*.
Sugestão: Se tiver problemas em implementar com base em catamorfismos estas duas últimas funções, tente implementar (com base em catamorfismos) as funções auxiliares

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

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

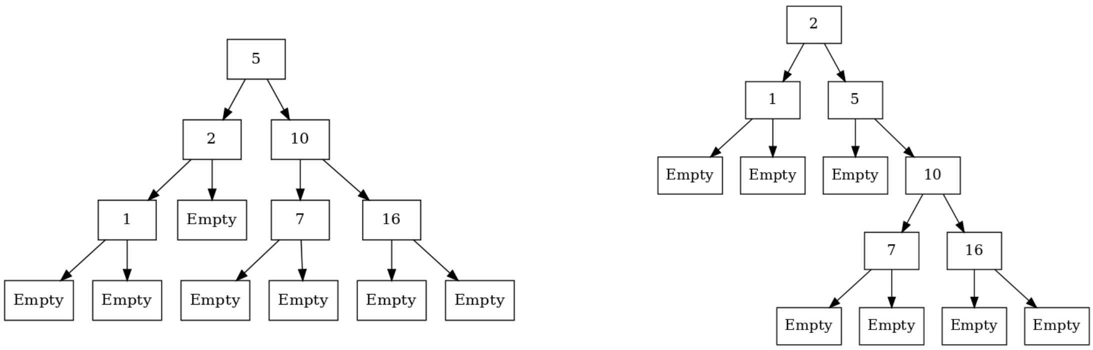


Figura 3: Exemplo de uma rotação à direita. A árvore da esquerda é a árvore original; a árvore da direita representa a rotação à direita correspondente.

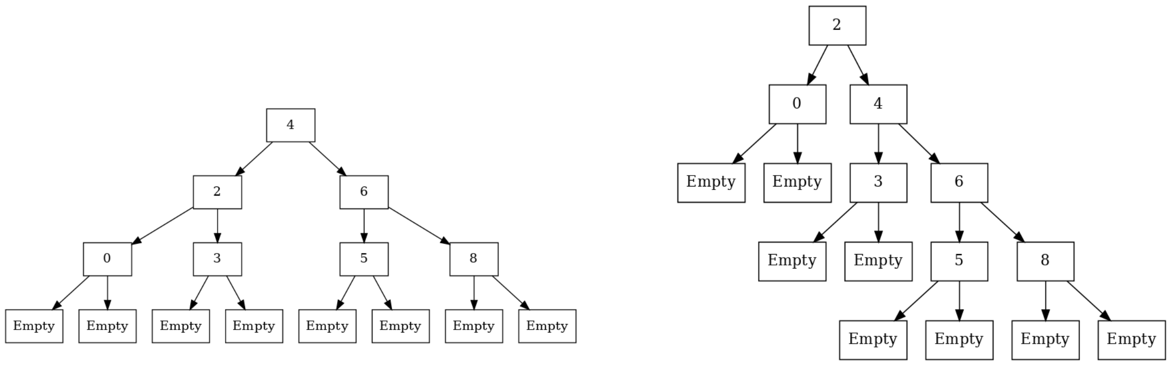


Figura 4: Exemplo de uma rotação à direita. A árvore da esquerda é a árvore original; a árvore da direita representa a rotação à direita correspondente.

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

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

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 **??** e **??** apresentam dois exemplos de rotações à direita. Note que em ambos os casos o valor 2 subiu um nível na árvore correspodente. De facto, podemos sempre aplicar uma *sequência* de rotações numa árvore de forma a mover um dado nó para a raíz (dando origem portanto à referida operação de splaying).

Começe então por implementar as funções

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

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

de rotação à direita e à esquerda.

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

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

De seguida implemente a operação de splaying

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

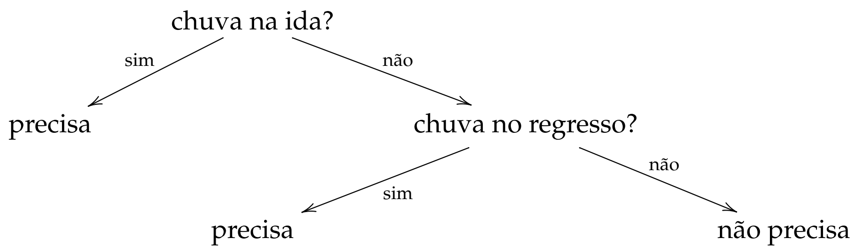
como um catamorfismo de listas. O argumento $[Bool]$ representa um caminho ao longo de uma árvore, em que o valor $True$ representa "seguir pelo ramo da esquerda"e o valor $False$ representa "seguir pelo ramo da direita". O caminho ao longo de uma árvore serve para *identificar* unicamente um nó dessa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

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

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

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:

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

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

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

1. Definir as funções $inBdt$, $outBdt$, $baseBdt$, $cataBdt$, e $anaBdt$.

2. Apresentar no relatório o diagrama de $anaBdt$.

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

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

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

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

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

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

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

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

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

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

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

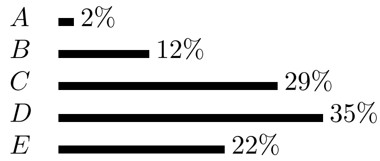
Problema 4

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 :: 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)

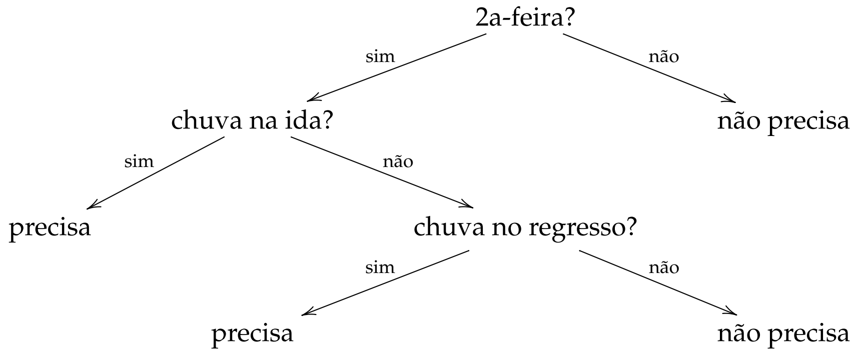
$$(f \bullet g)\ a = [(y, q * p) \mid (x, p) \leftarrow g\ a, (y, q) \leftarrow f\ x]$$

em que $g : A \rightarrow \mathsf{Dist}\ B$ e $f : B \rightarrow \mathsf{Dist}\ C$ são funções **monádicas** que representam *computações probabilísticas*. Este mónade é adequado à resolução de problemas de *probabilidades e estatística* usando programação funcional, de forma elegante e como caso particular da programação monádica. Vamos estudar a aplicação deste mónade ao exercício anterior, tendo em conta o facto de que nem sempre podemos responder com 100% de certeza a perguntas presentes em árvores de decisão.

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

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

respectiva a este problema.



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

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

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

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

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

Por fim, implemente como um catamorfismo de LTree a função

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

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

Problema 5

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

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 **??** é dada uma implementação da operação de permuta aleatória de uma lista que pode ser útil para resolver este exercício.

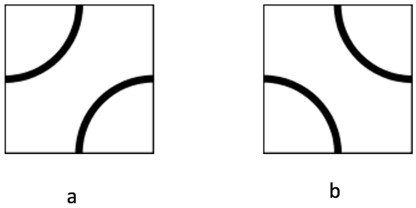


Figura 5: Os dois ladrilhos de Truchet-Smith.

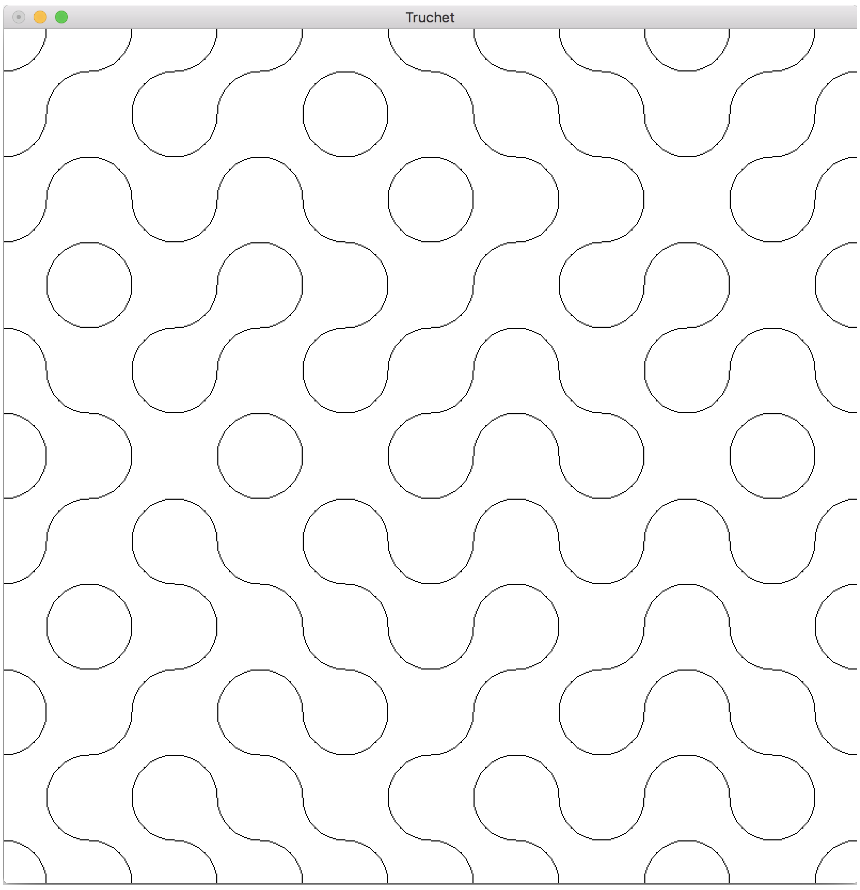


Figura 6: Um mosaico de Truchet-Smith.

Anexos

A Como exprimir cálculos e diagramas em LaTeX/lhs2tex

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

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

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

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

B Código fornecido

Problema 1

Função de representação de um dicionário:

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

onde

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

Dicionário para testes:

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

Normalização de um dicionário (remoção de entradas vazias):

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

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

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

[5] Exemplos tirados de [**?**].

Problema 2

Árvores usadas no texto:

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

Problema 3

Funções usadas para efeitos de teste:

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

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:

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

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

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

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

Outras funções auxiliares

Lógicas:

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

Compilação e execução dentro do interpretador:[6]

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

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

Começamos por definir a função discollect como um catamorfismo de listas, da seguinte forma:

$$discollect :: (Ord\ b, Ord\ a) \Rightarrow [(b, [a])] \to [(b, a)]$$
$$discollect = cataList\ g\ \textbf{where}$$
$$\quad g = [nil, k]$$
$$\quad k = \mathsf{conc} \cdot (discc \times id)$$
$$\quad discc\ (a, x) = [(a, b) \mid b \leftarrow x]$$

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

Para a função auxiliar tar do dic_exp usamos um catamorfismo de árvores de expressão, se estivermos numa Var v damos return a uma lista com o par [(' '),v], caso seja um Term vamos concatenar esse termo ao que já tinhamos do lado esquerdo do par.

$$
\begin{array}{l}
tar = cataExp\ g\ \textbf{where} \\
\quad g = [k, h] \\
\quad k\ v = [(\texttt{""}, v)] \\
\quad h = \widehat{f} \cdot (id \times concat) \\
\quad f\ x\ y = \mathsf{map}\ \ ((curry\ \mathsf{conc}\ x) \times id)\ y \\
dic\_exp :: Dict \rightarrow [(String, [String])] \\
dic\_exp = collect \cdot tar
\end{array}
$$

Para a função de procura usamos um hilomorfismo de árvores de expressão onde o anamorfismo vai "podar"a árvore, ou seja temos um Termo que não é prefixo da palavra que estamos a pesquisar, vamos colocar uma lista vazia, na lista das árvores de expressão desse tal termo, podando assim os ramos abaixo dele, se for prefixo, vamos procurar a palavra tirando esse prefixo, na lista desse Termo, podando quando necessário.

No final do anamorfismo vamos acabar com uma árvore onde as Variáveis dela são as traduções possíveis para uma dada palavra, assim usamos um catamorfismo para colocar essas Variáveis todas em uma lista.

É de notar que a definição do gene do nosso catamorfismo é exatamente igual à função expLeaves do módulo Exp.hs, por isso podiamos defininir esta função como a expLeaves após o anamorfismo antes referido.

$$
\begin{array}{l}
dic\_rd = curry\ (finaliza \cdot (hyloExp\ dic\_rd\_CataGen\ dic\_rd\_AnaGen)) \\
\quad \textbf{where}\ finaliza\ [\,] = Nothing \\
\qquad finaliza\ lista = (Just \cdot iSort \cdot nub)\ lista
\end{array}
$$

gene do catamorfismo

$$
dic\_rd\_CataGen = [singl, concat \cdot \pi_2]
$$

gene do anamorfismo

$$
\begin{array}{l}
dic\_rd\_AnaGen\ (\texttt{""}, Var\ a) = i_1\ a \\
dic\_rd\_AnaGen\ (p, Var\ \_) = i_2\ (p, [\,]) \\
dic\_rd\_AnaGen\ (p, Term\ j\ n) = \textbf{if}\ (isPrefix\ j\ p)\ \textbf{then}\ i_2\ (j, [(sufixo, o) \mid o \leftarrow n])\ \textbf{else}\ i_2\ (p, [\,]) \\
\quad \textbf{where}\ sufixo = drop\ (\mathsf{length}\ \ j)\ p \\
isPrefix\ x\ l = elem\ x\ (prefixes\ l)
\end{array}
$$

Para a função de inserção usamos um anamorfismo que vai cobrir todos os casos possíveis de inserções. Primeiro testamos se a palavra já existe no dicionário com a dic_rd, se isso se confimar, testamos se a tradução existe na lista de traduções da palavra procurada, se sim damos return ao dicionário. Caso contrário iremos inseri-la usando então o anamorfismo de árvores de expressão, passamos-lhe (dici,(p,t,True) ou seja o dicionário, a palavra a ser inserida, a sua respetiva tradução e um booleano True, este booleano é uma espécie de flag ao longo da nossa função que nos vai dizer se é nessa expressão que temos de inserir.

Vamos enumerar as linhas e os diferentes casos:

Caso estejamos numa Var

1 - Caso de quando a flag é falsa em uma Variável, damos return a essa Variável

2 - Caso a palavra for vazia acrescentamos a sua tradução a um termo novo pois Var "chinelo"== Term " [Var "chinelo"]

3 - Caso a palavra não seja vazia acrescentamos a palavra e a sua tradução, como um termo, dentro de outro com o que já tinha antes.

Caso estejamos num Termo
4 - Caso a flag seja false vamos dar return ao Termo com que tinhamos antes, sem alterar.
5 - Caso o operador j for prefixo de p e forem iguais então vamos dar return a um Termo novo com esse mesmo j , a lista de expressões do j concatenada com nova Variável inserida.
6 - Caso o operador j for prefixo da palavra, a palavra a inserir for diferente do operador j e a insertProbe do sufixo da palavra na lista dos Termos der True vamos então inserir no Termo que deu True, utilizando a função insertP
7 - Caso (5) e (6) não se cumprir vamos para o case 7 que vamos apenas concatenar o (Term sufixo [Var t],(,,False)) à lista em que estamos
8 - Caso o operador j não for prefixo de p mas a palavra p for prefixo do operador j então vamos dar return a um Termo com a a palavra p como operador e uma lista com a tradução t e com o resto dos Termos com o operador deles sendo o sufixo2 que é o operador j sem o p inicial
9 - Caso as palavras não tenham nada a ver (nenhuma ser prefixo da outra) vamos dar return a um Termo com o operador , com o que ja tinhamos antes e um novo Termo com lá dentro com a palavra e a tradução

```
dic_in p t dici = if (exists t (dic_rd p dici)) then dici else anaExp dic_in_AnaGen (dici, (p, t, True))
  where exists _ Nothing = False
    exists o (Just x) = elem o x
dic_in_AnaGen (Var a, (_, _, False)) = i1 a                                                   -- 1
dic_in_AnaGen (Var a, (p, t, _)) = if (p ≡ "")
  then i2 ("", [(Var a, ("", "", False)), (Var t, ("", "", False))])                          -- 2
  else i2 ("", [(Var a, ("", "", False)), (Term p [Var t], ("", "", False))])                 -- 3
dic_in_AnaGen (Term j n, (_, _, False)) = i2 (j, [(o, ("", "", False)) | o ← n])              -- 4
dic_in_AnaGen (Term j n, (p, t, True)) = if (isPrefix j p)
  then if (p ≡ j) then i2 (j, cons ((Var t, ("", "", False)),   [(o, ("", "", False)) | o ← n]))   -- 5
    else if (insertProbe sufixo n)
      then i2 (j, insertP sufixo t n)                                                         -- 6
        else i2 (j, cons ((Term sufixo [Var t], ("", "", False)), [(o, ("", "", False)) | o ← n]))  -- 7
  else if (isPrefix p j) then i2 (p, [(Var t, ("", "", False)), (Term sufixo2 n, ("", "", False))])  -- 8
  else i2 ("", [(Term p [Var t], ("", "", False)), (Term j n, (p, t, False))])                -- 9
            where sufixo = drop (length j) p
                  sufixo2 = drop (length p) j
```

A função insertProbe apenas verifica se pode inserir naquela lista.

```
insertProbe p [] = False
insertProbe p ((Var a) : xs) = False ∨ (insertProbe p xs)
insertProbe p ((Term j o) : xs) = (elem j (prefixes p)) ∨ (insertProbe p xs)
```

A função insertP, se a insertProbe se confirmar, verifica a mesma coisa mas desta vez constrói a lista pondo True ou False

```
insertP _ _ [] = []
insertP p t ((Var a) : xs) = (Var a, ("", "", False)) : (insertP p t xs)
insertP p t ((Term o n) : xs) = if (elem o (prefixes p))
  then (Term o n, (p, t, True)) : (insertP p t xs)
    else (Term o n, ("", "", False)) : (insertP p t xs)
```

Problema 2

Nesta função "maisDir", percorremos sempre para a direita, até verificarmos que a posição à direita está como "Empty"(aqui sabemos que nos encontramos na posição mais à direita possível). E quando assim o verificamos, retorna-se x (Just x).

$$
\begin{array}{l}
maisDir = cataBTree\ g\ \mathbf{where} \\
\quad g = [\underline{Nothing}, h] \\
\quad h\ (x, (\_, Nothing)) = Just\ x \\
\quad h\ (\_, (\_, r)) = r
\end{array}
$$

Esta próxima função é análoga à função anterior. Mas, obviamente, para o lado esquerdo.

$$
\begin{array}{l}
maisEsq = cataBTree\ g \\
\quad \mathbf{where}\ g = [\underline{Nothing}, h] \\
\qquad h\ (x, (Nothing, \_)) = Just\ x \\
\qquad h\ (\_, (l, \_)) = l
\end{array}
$$

Nesta função "insOrd", utilizamos um catamorfismo. De início, multiplicamos a árvore, devido à utilização da recursividade mútua ((l1,r1) representam funções). Aqui temos 2 casos, se tivermos Nil Nil, devolvemos um nó, com descendentes vazios. E o outro caso, verificamos se o elemento é igual, não inserimos, se for menor, vamos trabalhar na parte esquerda e obviamente, se o elemento for maior, trabalharemos na parte da direita.

$$
\begin{array}{l}
insOrd'\ x = cataBTree\ g \\
\quad \mathbf{where}\ g = \bot \\
insOrd\ a\ x = (cataBTree\ g)\ x\ x\ a \\
\quad \mathbf{where}\ g = [g1, g2] \\
\qquad g1\ \_\ \_\ e = Node\ (e, (Empty, Empty)) \\
\qquad g2\ (e_1, (l1, r1))\ (Node\ (e_2, (l2, r2)))\ e = \mathbf{if}\ (e \equiv e_1)\ \mathbf{then}\ Node\ (e_2, (l2, r2)) \\
\qquad\quad \mathbf{else\ if}\ (e < e_1)\ \mathbf{then}\ Node\ (e_1, ((l1\ l2\ e), r2))\ \mathbf{else}\ Node\ (e_1, (l2, (r1\ r2\ e)))
\end{array}
$$

Esta próxima função, "isOrd", tem como intuito verificar se a árvore está ordenada. Como podemos verificar, no segundo caso, (g2), primeiro verificaremos a cabeça e depois os filhos, isto com o auxílio de duas funções, "checkHead"e "checkSons", respetivamente. A função "checkHead", usa ainda outra função "isBigger"para auxílio à parte esquerda da árvore, e a função "isSmaller"para a parte à direita. A função "checkSons", faz (l ll), para a parte esquerda e ainda (r rr), para a parte à direita.

$$
\begin{array}{l}
isOrd' = cataBTree\ g \\
\quad \mathbf{where}\ g = \bot \\
isOrd\ j = (cataBTree\ g)\ j\ j \\
\quad \mathbf{where}\ g = [g1, curry\ g2] \\
\qquad g1\ \_\ \_ = True \\
\qquad g2\ ((a, (l, r)), Node\ (aa, (ll, rr))) = checkHead \land checkSons \\
\qquad\quad \mathbf{where}\ checkHead = isBigger\ a\ (maisEsq\ ll) \land isSmaller\ a\ (maisDir\ rr) \\
\qquad\qquad checkSons = l\ ll \land r\ rr \\
isBigger\ x\ Nothing = True \\
isBigger\ x\ (Just\ j) = x > j \\
isSmaller\ x\ Nothing = True \\
isSmaller\ x\ (Just\ j) = x < j
\end{array}
$$

A função que se segue, ”rrot”, vai servir para efetuar uma rotação à direita. Como é óbvio, em primeiro caso, temos que, rodar Empty, devolverá automaticamente Empty (caso base). No caso de, (x a Empty c), algo que não poderá acontecer, devolvemos de forma igual. O outro caso, e o mais complicado a nosso ver de chegar a uma conclusão, é quando temos (x a (Node(aa,(bb,cc))) c) e aqui sim, já efetuaremos uma rotação como pretendido e que fica da forma (Node (aa, (bb,(Node (a,(cc,c)))))).

$$
\begin{array}{l}
\mathit{rrot}\ \mathit{Empty} = \mathit{Empty} \\
\mathit{rrot}\ (\mathit{Node}\ (a,(b,c))) = x\ a\ b\ c \\
\quad \textbf{where}\ x\ a\ \mathit{Empty}\ c = \mathit{Node}\ (a,(b,c)) \\
\qquad x\ a\ (\mathit{Node}\ (\mathit{aa},(\mathit{bb},\mathit{cc})))\ c = \mathit{Node}\ (\mathit{aa},(\mathit{bb},(\mathit{Node}\ (a,(\mathit{cc},c)))))
\end{array}
$$

Esta próxima função, funciona de forma análoga à anterior, mas esta, servir-nos-á para efetuar rotações à esquerda.

$$
\begin{array}{l}
\mathit{lrot}\ \mathit{Empty} = \mathit{Empty} \\
\mathit{lrot}\ (\mathit{Node}\ (a,(b,c))) = x\ a\ b\ c \\
\quad \textbf{where}\ x\ a\ b\ \mathit{Empty} = \mathit{Node}\ (a,(b,c)) \\
\qquad x\ a\ b\ (\mathit{Node}\ (\mathit{aa},(\mathit{bb},\mathit{cc}))) = \mathit{Node}\ (\mathit{aa},((\mathit{Node}\ (a,(b,\mathit{bb}))),\mathit{cc}))
\end{array}
$$

E por último, neste problema, chega a função ”splayCataGen”. Esta função, através de Bools, executará rotações (”True”refere-se às rotações para a esquerda e portanto ”False”, referir-se-á às rotações para a direita).

$$
\begin{array}{l}
\mathit{splay}\ l\ t = (\mathit{cataList}\ \mathit{splayCataGen})\ l\ t \\
\mathit{splayCataGen} = [\mathit{g1}, \mathit{curry}\ \mathit{g2}] \\
\quad \textbf{where}\ \mathit{g1}\ \_\ \mathit{bt} = \mathit{bt} \\
\qquad \mathit{g2}\ ((j,n),\mathit{bt}) = \textbf{if}\ (j)\ \textbf{then}\ n\ (\mathit{rrot}\ \mathit{bt})\ \textbf{else}\ n\ (\mathit{lrot}\ \mathit{bt})
\end{array}
$$

Problema 3

Definimos as funções da seguinte forma:

$$
\begin{array}{l}
\mathit{inBdt} = [\mathit{Dec}, \mathit{Query}] \\
\mathit{outBdt}\ (\mathit{Dec}\ a) = i_1\ a \\
\mathit{outBdt}\ (\mathit{Query}\ (s,(l,r))) = i_2\ (s,(l,r)) \\
\mathit{baseBdt}\ g\ h = \mathit{id} + (g \times (h \times h)) \\
\mathit{recBdt}\ g = \mathit{baseBdt}\ \mathit{id}\ g \\
\mathit{cataBdt}\ g = g \cdot (\mathit{recBdt}\ (\mathit{cataBdt}\ g)) \cdot \mathit{outBdt} \\
\mathit{anaBdt}\ g = \mathit{inBdt} \cdot (\mathit{recBdt}\ (\mathit{anaBdt}\ g)) \cdot g
\end{array}
$$

$$
\begin{tikzcd}
\mathit{Bdt} \arrow[r, "g"'] \arrow[d, "(\!|g|\!)"'] & \mathit{Dec} + A \times (\mathit{Bdt} \times \mathit{Bdt}) \arrow[d, "\mathit{id}+\mathit{id}\times((\!|g|\!)\times(\!|g|\!))"] \\
\mathit{Bdt} & \mathit{Dec} + A \times (\mathit{Bdt} \times \mathit{Bdt}) \arrow[l, "\mathit{inBdt}"]
\end{tikzcd}
$$

E seguidamente o diagrama do anamorfismo em árvores de decisão binárias

A função extLtree para esquecer a informação nos nós é definida como um catamorfismo de árvores de decisão binárias simples que quando recebe uma Decisão a, vai transformar em Leaf a, e quando recebe uma Query (s,(l,r)) vai transformar o (l,r) em um Fork, esquecendo assim o s (informação no nó)

$$
\begin{array}{l}
\mathit{extLTree} :: \mathit{Bdt}\ a \rightarrow \mathsf{LTree}\ a \\
\mathit{extLTree} = \mathit{cataBdt}\ g\ \mathbf{where} \\
\quad g = [\mathit{Leaf}, \mathit{Fork} \cdot \pi_2]
\end{array}
$$

A função de navegação em uma árvore de decisão, foi implementada através de um catamorfismo, de Leaf Trees, no caso base, se estivermos em uma folha e tivermos qualquer coisa na lista, vamos dar essa folha pois não podemos tomar nenhuma decisão em uma folha, no caso recursivo, se a lista for vazia, damos a LTree que estivermos a "olhar", caso não seja vazia, testamos o valor da cabeça da lista de bool, se for true vamos para a função do split do lado esquerdo (p1) com o resto da lista, se for false vamos para o lado direito.

$$
\begin{array}{l}
\mathit{navLTree} :: \mathsf{LTree}\ a \rightarrow ([\mathit{Bool}] \rightarrow \mathsf{LTree}\ a) \\
\mathit{navLTree}\ x\ y = \mathit{cataLTree}\ \mathit{navLTreeGen}\ x\ y \\
\mathit{navLTreeGen} = [\mathit{g1}, \mathit{curry}\ \mathit{g2}] \\
\quad \mathbf{where}\ \mathit{g1}\ x\ \_ = \mathit{Leaf}\ x \\
\qquad \mathit{g2}\ ((a, b), [\,]) = \mathit{Fork}\ (\langle a, b\rangle\ [\,]) \\
\qquad \mathit{g2}\ ((a, b), (x : \mathit{xs})) = \mathbf{if}\ (x)\ \mathbf{then}\ \pi_1\ \$\ g\ \mathbf{else}\ \pi_2\ \$\ g \\
\qquad\quad \mathbf{where}\ g = \langle a, b\rangle\ \mathit{xs}
\end{array}
$$

Problema 4

Esta função "bnavLTree", recebe uma LTreee uma BTree de Bools e devolvemos uma LTree. Aqui nesta função, temos (l1,r1) que funcionam como funções, (l2,r2) que funcionam como representação da árvore e ainda (l3,r3) que conforme o valor de b, se "b=True"então faremos (l1 l2 l3), no caso de "b=False"então teremos (r1 r2 r3).

$$
\begin{array}{l}
\mathit{bnavLTree}\ \mathit{lt}\ \mathit{bt} = (\mathit{cataLTree}\ \mathit{bnavLTreeGen})\ \mathit{lt}\ \mathit{lt}\ \mathit{bt} \\
\mathit{bnavLTreeGen} = [\mathit{g1}, \mathit{g2}] \\
\quad \mathbf{where}\ \mathit{g1}\ j\ \_\ \_ = \mathit{Leaf}\ j \\
\qquad \mathit{g2}\ \_\ \mathit{lt}\ \mathit{Empty} = \mathit{lt} \\
\qquad \mathit{g2}\ (\mathit{l1}, \mathit{r1})\ (\mathit{Fork}\ (\mathit{l2}, \mathit{r2}))\ (\mathit{Node}\ (b, (\mathit{l3}, \mathit{r3}))) = \mathbf{if}\ (b)\ \mathbf{then}\ \mathit{l1}\ \mathit{l2}\ \mathit{l3}\ \mathbf{else}\ \mathit{r1}\ \mathit{r2}\ \mathit{r3}
\end{array}
$$

A função pbnavLTree recebe uma LTree e uma BTree de distribuição e tem como objetivo devolver uma lista com as árvores próvaveis. Aqui, utilizamos um catamorfismo, começamos pela parte de baixo e vamos subindo na árvore, por isso a probabilidade individual de cada folha vai começar sempre a 100 por cento (1.0)[1], se a árvore de decisão for fazia, também colocamos 100 por cento na árvore em qual estamos [2]. Por fim, na chamada recursiva faz-se 2 pares, um para percorrer o ramo da esquerda e outro o ramo da direita[3].

$$
\begin{array}{l}
\mathit{pbnavLTree}\ \mathit{lt}\ \mathit{bt} = (\mathit{cataLTree}\ \mathit{pbnavLTreeGen})\ \mathit{lt}\ \mathit{lt}\ \mathit{bt} \\
\mathit{pbnavLTreeGen} = [\mathit{g1}, \mathit{g2}] \\
\quad \mathbf{where}\ \mathit{g1}\ j\ \_\ \_ = D\ [(\mathit{Leaf}\ j, 1.0)] \quad \text{-- 1} \\
\qquad \mathit{g2}\ \_\ \mathit{bt}\ \mathit{Empty} = D\ [(\mathit{bt}, 1.0)] \quad \text{-- 2} \\
\qquad \mathit{g2}\ (\mathit{l1}, \mathit{r1})\ (\mathit{Fork}\ (\mathit{l2}, \mathit{r2}))\ (\mathit{Node}\ (\mathit{dist}, (\mathit{l3}, \mathit{r3}))) = \\
\qquad\quad (D \cdot \mathsf{conc} \cdot (n \times n))\ ((\mathit{l1}\ \mathit{l2}\ \mathit{l3}, \mathit{probEsq}), (\mathit{r1}\ \mathit{r2}\ \mathit{r3}, \mathit{probDir})) \quad \text{-- 3}
\end{array}
$$

$$
\begin{aligned}
&\textbf{where } \mathit{probEsq} = \mathit{getTrueProb}\ (\mathit{getProbs})\\
&\quad \mathit{probDir} = \mathit{getFalseProb}\ (\mathit{getProbs})\\
&\quad \mathit{getProbs} = (\mathsf{map}\ \ (\mathit{id} \times \mathit{sum}) \cdot \mathit{collectProb} \cdot \mathit{unD})\ \mathit{dist}\\
&\quad n = \mathit{distProb} \cdot (\mathit{unD} \times \mathit{id})\\
&\quad \mathit{unD} = \mathit{Probability.unD}
\end{aligned}
$$

Nas funções auxiliares descritas em baixo, o getTrueProb[4] vai buscar a probabilidade de ser True sendo análoga à getFalseProb[5], a distProb[6] vai multiplicar as probabilidades para sabermos a probabilidade encadeada e a collectProb[7] vai colecionar probabilidades que são iguais, colocando em um par o booleano do lado esquerdo, e do outro lado a lista das respetivas probabilidades, após isso o (map (id >< sum) irá somar essa lista, ficamos assim só com 0-2 pares, na variável getProbs.

$$
\begin{aligned}
&\mathit{getTrueProb} = \mathit{cataList}\ g \quad \text{-- 4}\\
&\quad \textbf{where } g = [\mathit{g1}, \mathit{g2}]\\
&\qquad \mathit{g1}\ \_ = 0.0\\
&\qquad \mathit{g2}\ ((j, n), o) = \textbf{if } (j) \textbf{ then } n \textbf{ else } o\\
&\mathit{getFalseProb} = \mathit{cataList}\ g \quad \text{-- 5}\\
&\quad \textbf{where } g = [\mathit{g1}, \mathit{g2}]\\
&\qquad \mathit{g1}\ \_ = 0.0\\
&\qquad \mathit{g2}\ ((j, n), o) = \textbf{if } (\neg\, j) \textbf{ then } n \textbf{ else } o\\
&\mathit{distProb}\ (x, y) = \mathsf{map}\ \ (\mathit{id} \times (*y))\ x \quad \text{-- 6}\\
&\mathit{collectProb}\ x = \mathit{set}\ [k \mapsto [d' \mid (k', d') \leftarrow x, k' \equiv k] \mid (k, d) \leftarrow x] \quad \text{-- 7}
\end{aligned}
$$

Problema 5

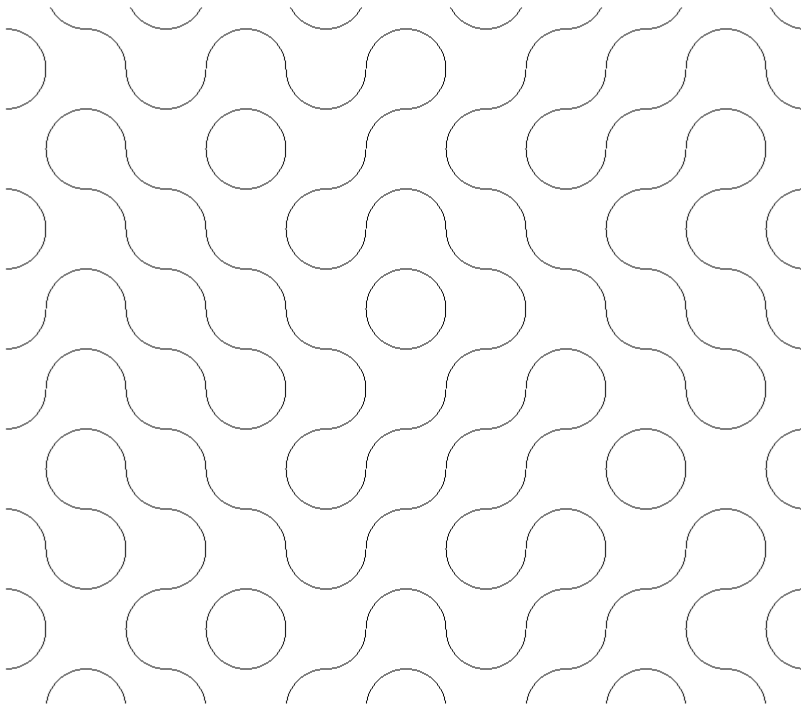


Figura 7: Exemplo gerado com a função drawTruchet 10 10.

Os truchet tiles usados como base

$$truchet1 = Pictures\ [put\ (0, 80)\ (Arc\ (-90)\ 0\ 40), put\ (80, 0)\ (Arc\ 90\ 180\ 40)]$$
$$truchet2 = Pictures\ [put\ (0, 0)\ (Arc\ 0\ 90\ 40), put\ (80, 80)\ (Arc\ 180\ (-90)\ 40)]$$
$$put = \widehat{Translate}$$

A função drawTruchet x y é a principal, para a usar colocamos quantos tiles queremos no x e quantos tiles queremos no y, a janela vai ser gerada à medida, e na pic vamos ter a informação para gerar a imagem.

$drawTruchet\ x\ y = display\ janela1\ white\ pic$
  where $pic = mconcat\ (mapGen\ x\ y)$
    $janela1 = InWindow$ `"Truchet"` $(80 * x, 80 * y)\ (100, 100)$

A função mapGen vai dar return a uma lista de listas de monads Picture, que depois são concatenados usando o mconcat na drawTruchet.

Nas mapCoords vamos ter as coordenadas criadas a partir de uma lista de compreensão, é uma lista de listas, que onde as listas interiores são 4 pontos que fazem um quadrado, e podemos manipular o ponto 2 e 3 para colocar um Truchet 1 e o ponto 1 e 4 para colocar um Truchet 2.

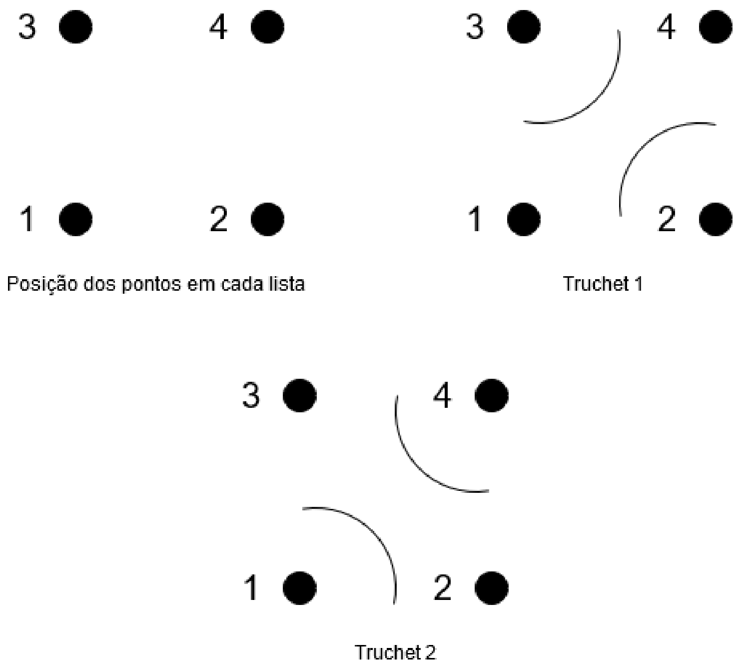


Figura 8: Geração dos pontos

Essa lista é criada toda no primeiro quadrante de um referencial x y, ou seja vamos precisar de centra-las, para esse efeito usamos um map de map onde o map interior vai fazer leftshift ao x e um downshift ao y de metade do comprimento e metade da largura. Ficamos assim com uma lista de listas das coordenadas já centradas em mapCenter

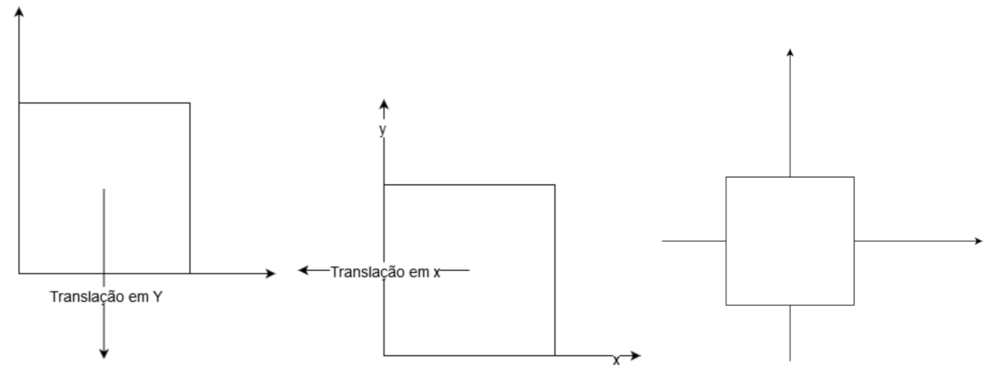


Figura 9: Exemplo do funcionamento do leftshift e do downshift

Agora vamos calcular nTruchet1 que vai ser a area / 2 e o nTruchet2 que vai ser o que sobra, com isto vamos ter porções mais ou menos iguais de Truchet1 e Truchet2, a razão disto é para a imagem ficar mais bonita e equilibrada, podiamos também implementar uma aleatoriedade aqui, escolhendo um número de [0,área] para os Truchet1 o o Truchet2 seria o resto Após isso genTruchetList vai criar uma lista com nTruchet1 1 e nTruchet2 2 Depois a função genTruchet vai permutar essa lista usando a função permuta disponibilizada e vai usar um unsafePerformIO para podermos trabalhar com esse IO

$$
\begin{array}{l}
mapGen\ comp\ lar = truchetSmithGen\ genTruchet\ mapCenter\\
\quad \mathbf{where}\ mapCenter = \mathsf{map}\ \ (\mathsf{map}\ \ (((+leftshift) \cdot realToFrac) \times ((+downshift) \cdot realToFrac)))\ mapCoords\\
\qquad mapCoords = [[(x * 80, y * 80), (x * 80 + 80, y * 80), (x * 80, y * 80 + 80), (x * 80 + 80, y * 80 + 80)] \mid\\
\qquad\quad x \leftarrow [0 \mathinner{\ldotp\ldotp} comp - 1], y \leftarrow [0 \mathinner{\ldotp\ldotp} lar - 1]]\\
\qquad leftshift = (-((realToFrac\ comp) \mathbin{/} 2 * 80))\\
\qquad downshift = (-((realToFrac\ lar) \mathbin{/} 2 * 80))\\
\qquad genTruchet = unsafePerformIO\ (permuta\ genTruchetList)\\
\qquad genTruchetList = \mathsf{conc}\ (\qquad replicate\ nTruchet1\ 1, replicate\ nTruchet2\ 2)\\
\qquad area = comp * lar\\
\qquad nTruchet1 = area \div 2\\
\qquad nTruchet2 = (nTruchet1 + (mod\ area\ 2))
\end{array}
$$

A função truchetSmithGen vai receber uma lista de 1's e 2's onde o 1 representa Truchet1 e o 2 representa o Truchet 2, o tamanho da lista é o número de truchets totais na imagem (x * y), esta lista já foi permutada, e também recebe uma lista de coordenadas "lcoords"onde tem então os 4 pontos.

A função vai usar um catamorfismo e a recursividade mútua, percorrendo assim a lista de 1's e 2's ao mesmo tempo que percorre a lista das coordenadas, se encontrar o número 1, vai colocar nessas coordenadas o Truchet 1, caso encontre um 2 coloca o Truchet 2.

$$
\begin{array}{l}
truchetSmithGen\ lts\ lcoords = (cataList\ g)\ lts\ lts\ lcoords\\
\quad \mathbf{where}\ g = [g1, g2]\\
\qquad g1\ \_\ \_\ \_ = [\,]\\
\qquad g2\ (1, j)\ (1 : o)\ (l : ls) = \mathsf{conc}\ ([Pictures\ [put\ (l\, !!\, 2)\ (Arc\ (-90)\ 0\ 40),\\
\qquad\quad put\ (l\, !!\, 1)\ (Arc\ 90\ 180\ 40)]], j\ \ o\ \ ls)\\
\qquad g2\ (2, j)\ (2 : o)\ (l : ls) = \mathsf{conc}\ ([Pictures\ [put\ (l\, !!\, 0)\ (Arc\ 0\ 90\ 40),\\
\qquad\quad put\ (l\, !!\, 3)\ (Arc\ 180\ (-90)\ 40)]], j\ \ o\ \ ls)
\end{array}
$$