Cálculo de Programas
Trabalho Prático
LCC+LEI — Ano Lectivo de 2014/15

Alfredo Gomes A71655
Octávio Maia A71369
Pedro Rocha A71802

Departamento de Informática
Universidade do Minho

Maio de 2015

Conteúdo

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 usa-se esses combinadores para construir programas *composicionalmente*, isto é, compondo programas já existentes.

Na sequência pedagógica dos planos de estudo dos dois cursos que têm esta disciplina, restringe-se a aplicação deste método ao desenvolvimento de programas funcionais na linguagem Haskell.

O presente trabalho tem por objectivo concretizar na prática os objectivos da disciplina, colocando os alunos perante problemas de programação que deverão ser abordados composicionalmente e implementados em Haskell. Há ainda um outro objectivo: o de ensinar a documentar programas e a produzir textos técnico-científicos de qualidade.

2 Documentação

Para cumprir de forma integrada e simples 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 sua documentação deverão constar do mesmo documento (ficheiro).

O ficheiro `cp1415t.pdf` que está a ler é já um exemplo de programação literária: foi gerado a partir do texto fonte `cp1415t.lhs`[1] que encontrará no material pedagógico desta disciplina descompactando o ficheiro `cp1415t.zip` e executando

```
lhs2TeX cp1415t.lhs > cp1415t.tex
pdflatex cp1415t
```

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

> `https://hackage.haskell.org/package/lhs2tex`.

Por outro lado, o mesmo ficheiro `cp1415t.lhs` é executável e contém o "kit" básico, escrito em Haskell, para realizar o trabalho. Basta executar

```
ghci cp1415t.lhs
```

para ver que assim é:

```
GHCi, version 7.8.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
[ 1 of 11] Compiling ListUtils        ( ListUtils.hs, interpreted )
[ 2 of 11] Compiling Cp               ( Cp.hs, interpreted )
[ 3 of 11] Compiling BTree            ( BTree.hs, interpreted )
[ 4 of 11] Compiling LTree            ( LTree.hs, interpreted )
[ 5 of 11] Compiling Exp              ( Exp.hs, interpreted )
[ 6 of 11] Compiling Nat              ( Nat.hs, interpreted )
[ 7 of 11] Compiling Show             ( Show.hs, interpreted )
[ 8 of 11] Compiling Probability      ( Probability.hs, interpreted )
[ 9 of 11] Compiling List             ( List.hs, interpreted )
[10 of 11] Compiling X3d              ( X3d.hs, interpreted )
[11 of 11] Compiling Main             ( cp1415t.lhs, interpreted )
Ok, modules loaded: List, Show, Nat, Exp, Cp, BTree, LTree, X3d,
Probability, Main, ListUtils.
```

O facto de o interpretador carregar as bibliotecas do material pedagógico da disciplina, entre outras, deve-se ao facto de, neste mesmo sítio do texto fonte, se ter inserido o seguinte código Haskell:

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

```
import Data.List
import System.Process
import Cp
import List
import Nat
import Exp
import BTree
import LTree
import X3d
import Control.Parallel.Strategies
import Probability hiding (· → ·, ·)
import System.Environment (getArgs)
```

Abra o ficheiro `cp1415t.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 equilibrada e 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, na 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 cp1415t.aux
makeindex cp1415t.idx
```

e recompilar o texto como acima se indicou.

4 Parte A

Nesta primeira parte do trabalho pretende-se averiguar a capacidade de utilização por parte dos alunos das bibliotecas fornecidas no material pedagógico da disciplina. Algumas respostas são validadas por testes unitários. Sempre que o resultado de um teste unitário for $False$, a solução proposta falha a validação e deve ser revista.

4.1 Biblioteca LTree

1. A seguinte função

$$balanced\ (Leaf\ \_) = True$$
$$balanced\ (Fork\ (t, t')) = balanced\ t \wedge balanced\ t' \wedge abs\ (depth\ t - depth\ t') \leqslant 1$$

testa se uma árvore binária está equilibrada ou não. Defina como catamorfismo em LTree a função auxiliar $depth$.

2. Seja dada:

$$t = Fork\ (Fork\ (Leaf\ 10, Fork\ (Leaf\ 2, Fork\ (Leaf\ 5, Leaf\ 3))), Leaf\ 23)$$

Testes unitários 1 *Verifique que árvore t está desequilibrada:*

$$test01 = balanced\ t \equiv False$$

3. Recorrendo a funções da biblioteca LTree, escreva numa única linha de Haskell a função

$$balance :: LTree\ a \to LTree\ a$$

que equilibra uma qualquer árvore binária.

Testes unitários 2 *Verifique que $balance\ t$ é uma árvore equilibrada:*

$$test02 = balanced\ (balance\ t) \equiv True$$

4.2 Biblioteca BTree

Pretende-se construir um anamorfismo que produza uma árvore binária de procura *equilibrada* que contenha o intervalo definido por dois inteiros (n, m):

$$abpe\ (n, m) = anaBTree\ qsplit\ (n, m)$$

Comece por definir o gene $qsplit$ e depois construa a árvore

$$t1 = abpe\ (20, 30)$$

que será precisa na secção 6.4.

Testes unitários 3 *Faça os testes seguintes:*

$$\begin{aligned}
&test03a = qsplit\ (4, 30) \equiv i_2\ (17, ((4, 16), (18, 30)))\\
&test03b = qsplit\ (4, 3) \equiv i_1\ ()\\
&test03c = qsplit\ (0, 0) \equiv i_1\ ()\\
&test03d = qsplit\ (1, 1) \equiv i_2\ (1, ((1, 0), (2, 1)))\\
&test03e = balBTree\ t1 \equiv True\\
&test03f = inordt\ t1 \equiv [20 \mathinner{..} 30]
\end{aligned}$$

4.3 Biblioteca para listas com sentinelas

Considere o tipo de dados que representa listas finitas com uma sentinela no fim:

$$\mathbf{data}\ SList\ a\ b = Sent\ b \mid Cons\ (a, SList\ a\ b)\ \mathbf{deriving}\ (Show, Eq)$$

1. Derive os isomorfismos $inSList$ e $outSList$, adicione-os a este ficheiro e passe aos testes que se seguem.

   Testes unitários 4 *Faça os testes seguintes:*

$$\begin{aligned}
&test04a = \mathbf{let}\ x = Cons\ (1, Sent\ \texttt{"end"})\ \mathbf{in}\ inSList\ (outSList\ x) \equiv x\\
&test04b = \mathbf{let}\ x = i_2\ (\texttt{"ola"}, Sent\ \texttt{"2"})\ \mathbf{in}\ outSList\ (inSList\ x) \equiv x
\end{aligned}$$

2. Derive os combinadores $cataSList$, $anaSList$ e $hyloSList$, e mostre que a função $merge$ da biblioteca LTree se pode escrever da forma seguinte,

$$\begin{aligned}
&merge' :: Ord\ a \Rightarrow ([a], [a]) \to [a]\\
&merge' = hyloSList\ [id, cons]\ mgen
\end{aligned}$$

   para um dado gene $mgen$ que deverá definir.

   Testes unitários 5 *Faça os seguintes testes:*

$$\begin{aligned}
&test05a = mgen\ ([0, 2, 5], [0, 6]) \equiv i_2\ (0, ([2, 5], [0, 6]))\\
&test05b = mgen\ ([0, 2, 5], [\,]) \equiv i_1\ [0, 2, 5]\\
&test05c = merge'\ ([\,], [0, 6]) \equiv [0, 6]
\end{aligned}$$

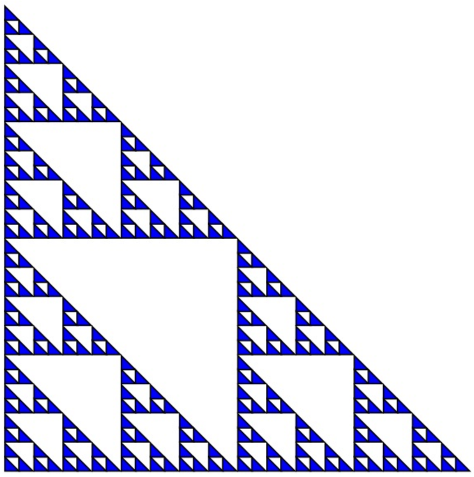


Figura 1: Um triângulo de Sierpinski

5 Parte B

O triângulo de Sierpinski é uma figura fractal que tem o aspecto da figura 1 e que se obtém da seguinte forma: considere-se um triângulo rectângulo e isósceles A cujos catetos têm comprimento s. A estrutura fractal é criada desenhando-se três triângulos no interior de A, todos eles rectângulos e isósceles e com catetos de comprimento $s/2$. Este passo é depois repetido para cada um dos triângulos desenhados, e assim sucessivamente. O resultado dos cinco primeiros passos é dado na Fig. 1.

Um triângulo de Sierpinski é gerado repetindo-se infinitamente o processo acima descrito. No entanto, para efeitos de visualização num monitor, cuja resolução é forçosamente finita, faz sentido escolher uma representação adequada do triângulo, parando o processo recursivo a um determinado nível. A figura a desenhar é constituída por um conjunto finito de triângulos todos da mesma dimensão (por exemplo, na figura 1 há 243 triângulos).

5.1 Criação de Triângulos de Sierpinski

Seja cada triângulo geometricamente descrito pelas coordenadas do seu vértice inferior esquerdo e o comprimento dos seus catetos:

$$\mathbf{type}\ \mathit{Tri} = (\mathit{Point}, \mathit{Side})$$

onde

$$\mathbf{type}\ \mathit{Side} = \mathit{Int}$$
$$\mathbf{type}\ \mathit{Point} = (\mathit{Int}, \mathit{Int})$$

A estrutura recursiva de (uma representação finita de) um triângulo de Sierpinski é captada por uma árvore ternária, em que cada nó é um triângulo com os respectivos três sub-triângulos:

$$\mathbf{data}\ \mathsf{TLTree} = \mathit{Tri}\ \mathit{Tri} \mid \mathit{Nodo}\ \mathsf{TLTree}\ \mathsf{TLTree}\ \mathsf{TLTree}$$

Nas folhas dessa árvore encontram-se os triângulos mais pequenos, todos da mesma dimensão, que deverão ser desenhados. Apenas estes conterão informação de carácter geométrico, tendo os nós da árvore um papel exclusivamente estrutural. Portanto, a informação geométrica guardada em cada folha consiste nas coordenadas do vértice inferior esquerdo e no lado dos catetos do respectivo triângulo. A função

$$\mathit{sierpinski} :: \mathit{Tri} \to \mathit{Int} \to [\,\mathit{Tri}\,]$$
$$\mathit{sierpinski}\ t = \mathit{apresentaSierp} \cdot (\mathit{geraSierp}\ t)$$

recebe a informação do triângulo exterior e o número de níveis pretendido, que funciona como critério de paragem do processo de construção do fractal. O seu resultado é a lista de triângulos a desenhar. Esta função é um hilomorfismo do tipo TLTree, i.e. a composição de duas funções: uma que gera TLTrees,

$$
\begin{array}{l}
\mathit{geraSierp} :: \mathit{Tri} \to \mathit{Int} \to \mathsf{TLTree} \\
\mathit{geraSierp}\ t\ 0 = \mathit{Tri}\ t \\
\mathit{geraSierp}\ ((x,y),s)\ n = \\
\quad \mathbf{let}\ s' = s \div 2 \\
\quad \mathbf{in}\ \mathit{Nodo} \\
\qquad\quad (\mathit{geraSierp}\ ((x,y),s')\ (n-1)) \\
\qquad\quad (\mathit{geraSierp}\ ((x+s',y),s')\ (n-1)) \\
\qquad\quad (\mathit{geraSierp}\ ((x,y+s'),s')\ (n-1))
\end{array}
$$

e outra que as consome:

$$
\begin{array}{l}
\mathit{apresentaSierp} :: \mathsf{TLTree} \to [\,\mathit{Tri}\,] \\
\mathit{apresentaSierp}\ (\mathit{Tri}\ t) = [\,t\,] \\
\mathit{apresentaSierp}\ (\mathit{Nodo}\ a\ b\ c) = (\mathit{apresentaSierp}\ a) +\!\!+ (\mathit{apresentaSierp}\ b) +\!\!+ (\mathit{apresentaSierp}\ c)
\end{array}
$$

5.2 Trabalho a realizar

Preparação:

1. Desenvolva a biblioteca "pointfree" `TLTree.hs` de forma análoga a outras bibliotecas que conhece (*eg.* BTree, LTree, etc) e que estão disponíveis no material pedagógico.

2. Defina como catamorfismos de TLTree as funções

$$
\begin{array}{l}
\mathit{tipsTLTree} :: \mathsf{TLTree}\ b \to [\,b\,] \\
\mathit{countTLTree} :: \mathsf{TLTree}\ b \to \mathit{Int} \\
\mathit{depthTLTree} :: \mathsf{TLTree}\ b \to \mathit{Int} \\
\mathit{invTLTree} :: \mathsf{TLTree}\ b \to \mathsf{TLTree}\ b
\end{array}
$$

   respectivamente semelhantes a tips, $\mathit{countLTree}$, depth e inv ("mirror") de LTree.

3. Exprima as funções $\mathit{geraSierp}$ e $\mathit{apresentaSierp}$ recorrendo a anamorfismos e catamorfismos, respectivamente, do tipo TLTree.

4. Defina a árvore

$$
\mathit{ts} = \mathit{geraSierp}\ \mathit{tri}\ 5\ \mathbf{where}\ \mathit{tri} = ((0,0),256)
$$

   e faça os testes seguintes:

   <u>Testes unitários</u> 6 *Verifique a profundidade da árvore gerada e o respectivo número de triângulos:*

$$
\begin{array}{l}
\mathit{test06a} = \mathit{depthTLTree}\ \mathit{ts} \equiv 6 \\
\mathit{test06b} = \mathit{countTLTree}\ \mathit{ts} \equiv 243 \\
\mathit{test06c} = \mathit{countTLTree}\ \mathit{ts} \equiv \mathit{length}\ (\mathit{tipsTLTree}\ \mathit{ts}) \\
\mathit{test06d} = \mathit{countTLTree}\ \mathit{ts} \equiv \mathit{countTLTree}\ (\mathit{invTLTree}\ \mathit{ts})
\end{array}
$$

Visualização: Para visualizarmos triângulos de Sierpinski vamos usar X3DOM, uma biblioteca "open-source" para construção e visualização de gráficos 3D no Web.[2] No pacote disponibilizado para a realização deste trabalho encontra a biblioteca $\mathit{X3d}$, que inclui a função $\mathit{drawTriangle}$ para geração de triângulos em 3D, usando X3DOM. Nesta abordagem, um ficheiro $\mathit{x3dom}$ é construído em dois passos:

- Desenham-se os triângulos, utilizando:

$$
\mathit{drawTriangle} :: ((\mathit{Int},\mathit{Int}),\mathit{Int}) \to \mathit{String}
$$

[2] Ver `http://examples.x3dom.org` para mais informação. Em `http://examples.x3dom.org/IG/buddha-anim/x3dom_imageGeometry.html`, por exemplo, pode ser visualizado um objecto gráfico com mais de um milhão de triângulos. Mais documentação em: `http://doc.x3dom.org/tutorials/index.html`.

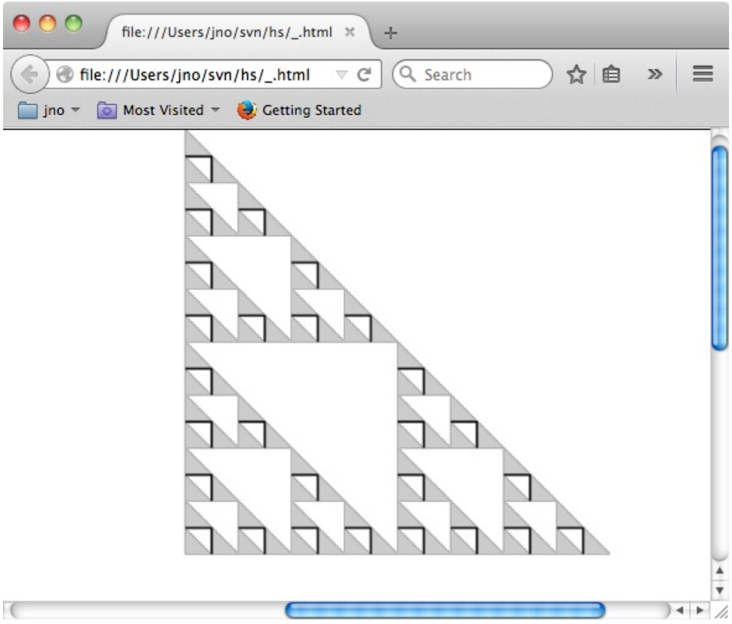


Figura 2: Um triângulo de Sierpinski em x3dom

- Finaliza-se o ficheiro com as tags de início e final:

  $$finalize :: String \rightarrow String$$

1. Usando estas funções e as que definiu anteriormente, faça a geração do HTML que representa graficamente o triângulo de Sierpinski definido por

   $$dados = (((0, 0), 32), 4)$$

   isto é, centrado na origem, com lado 32 e 4 níveis de recursividade. No anexo C sugere-se o recurso à função,

   $$render\ html = \textbf{do}\ \{ writeFile\ \texttt{"\_.html"}\ html; system\ \texttt{"firefox \_.html"} \}$$

   (adapte-a, se necessário) para visualizar o triângulo gerado num "browser". Espera-se que o resultado final seja como o que se mostra na Figura 2.

Valorização

Se tiver tempo, investigue como é que a sua resolução desta parte do trabalho evolui para o desenho, não de *triângulos* de Sierpinski, mas sim de *pirâmides* de Sierpinski — ver a imagem da figura 3. Pode recorrer, se desejar, às funções disponibilizadas no anexo B.1.

6 Parte C

6.1 Mónades

Os 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

$$\textbf{newtype}\ Dist\ a = D\ \{ unD :: [(a, ProbRep)] \}$$

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

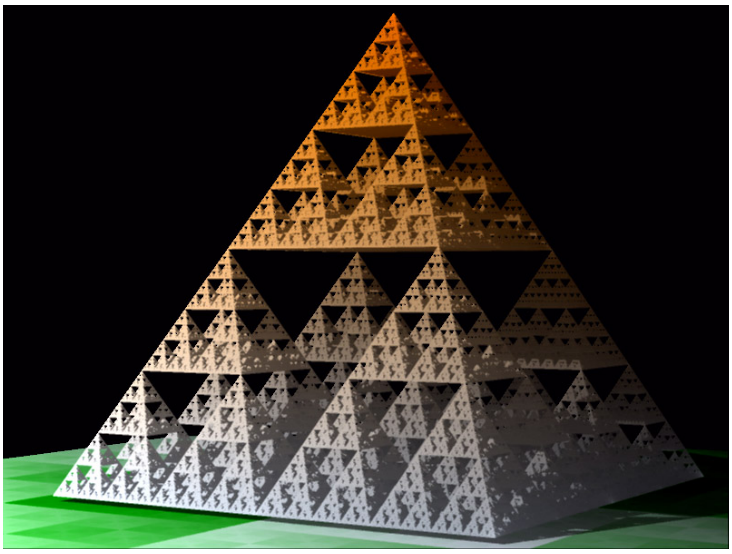
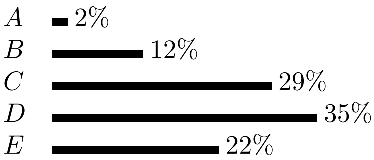


Figura 3: Uma pirâmide de Sierpinski

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

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

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.[3]

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

$Dist$ forma um **mónade** cuja unidade é $return\ a = D\ [(a, 1)]$ e cuja multiplicação é dada por (simplificando a notação)

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

em que $g : A \rightarrow Dist\ B$ e $f : B \rightarrow Dist\ C$ 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 de programação monádica. Vejamos um exemplo:

Problema: qual é a soma de faces mais provável quando lançamos dois dados num tabuleiro?

Assumindo que os dados não estão viciados, cada um oferece uma distribuição uniforme das suas faces (1 a 6). Basta correr a expressão monádica

$$\mathbf{do}\ \{x \leftarrow uniform\ [1\mathinner{..}6]; y \leftarrow uniform\ [1\mathinner{..}6]; return\ (x + y)\}$$

e obter-se-á:

```
*Main> do { x <- uniform [1..6] ;  y <- uniform [1..6] ; return(x+y) }
 7  16.7%
 6  13.9%
 8  13.9%
 5  11.1%
 9  11.1%
 4   8.3%
10   8.3%
 3   5.6%
11   5.6%
 2   2.8%
12   2.8%
```

A soma mais provável é 7, com 16.7%.

6.2 Trabalho a realizar

É possível pensarmos em catamorfismos, anamorfismos etc probabilísticos, quer dizer, programas recursivos que dão distribuições como resultados. Por exemplo, neste enunciado é dado o combinador

$$pcataList :: (Either\ ()\ (a, b) \rightarrow Dist\ b) \rightarrow [a] \rightarrow Dist\ b$$

que é muito parecido com

$$cataList :: (Either\ ()\ (a, b) \rightarrow b) \rightarrow [a] \rightarrow b$$

da biblioteca List. A única diferença é que o gene de $pcataList$ é uma função probabilística.

Exemplo de utilização: recorde-se que $cataList\ [zero, add]$ soma todos os elementos da lista argumento, por exemplo:

$$cataList\ [zero, add]\ [20, 10, 5] = 35.$$

Considere agora a função $padd$ (adição probabilística) que, com probabilidade 90% soma dois números e com probabilidade 10% os subtrai:

$$padd\ (a, b) = D\ [(a + b, 0.9), (a - b, 0.1)]$$

Se se correr

$$d4 = pcataList\ [pzero, padd]\ [20, 10, 5]\ \mathbf{where}\ pzero = return \cdot zero$$

obter-se-á:

```
35  81.0%
25   9.0%
 5   9.0%
15   1.0%
```

Com base nestes exemplos, resolva o seguinte

***Problema**: Uma unidade militar pretende enviar uma mensagem urgente a outra, mas tem o aparelho de telegrafia meio avariado. Por experiência, o telegrafista sabe que a probabilidade de uma palavra se perder (não ser transmitida) é 5%; no final de cada mensagem, o aparelho envia o código* `"stop"`*, mas (por estar meio avariado), falha 10% das vezes.*

Qual a probabilidade de a palavra `"atacar"` *da mensagem words* `"Vamos atacar hoje"` *se perder, isto é, o resultado da transmissão ser* `["Vamos","hoje","stop"]`*? e a de seguirem todas as palavras, mas faltar o* `"stop"` *no fim? E a da transmissão ser perfeita?*

Responda a todas estas perguntas encontrando g tal que

$$transmitir = pcataList\ gene$$

descreve o comportamento do aparelho.

Testes unitários 7 *Faça o seguinte teste unitário da sua versão para $gene$:*

$$test07 = gene\ (i_2\ (\texttt{"a"}, [\texttt{"b"}])) \equiv D\ [([\texttt{"a"}, \texttt{"b"}], 0.95), ([\texttt{"b"}], 0.05)]$$

Responda então às perguntas do problema acima correndo a expressão:

$$transmitir\ (words\ \texttt{"Vamos atacar hoje"})$$

6.3 Programação funcional paralela

Uma outra aplicação do conceito de mónade é a programação funcional paralela. A biblioteca Control.Parallel.Strategies, já carregada no início deste texto, implementa esse tipo de programação, que hoje está na ordem do dia. O mónade respectivo chama-se $Eval$ e disponibiliza duas funções,

$$rpar :: a \to Eval\ a$$
$$rseq :: a \to Eval\ a$$

conforme se deseja que uma dada computação seja efectuada em paralelo ou sequencialmente.[4] Por exemplo,

$$\begin{array}{l} parmap :: (a \to b) \to [a] \to Eval\ [b] \\ parmap\ f\ [\,] = return\ [\,] \\ parmap\ f\ (a : lt) = \mathbf{do} \\ \quad a' \leftarrow rpar\ (f\ a) \\ \quad lt' \leftarrow parmap\ f\ lt \\ \quad return\ (a' : lt') \end{array}$$

é um map monádico que usa $rpar$ para aplicar f a todos os elementos de uma lista *em paralelo*.

Se corrermos o map habitual em

$$map\ fib\ [20\,..\,30] = [10946, 17711, 28657, 46368, 75025, 121393, 196418, 317811, 514229, 832040, 1346269]$$

(cálculo dos números de Fibonacci do vigésimo ao trigésimo), o tempo que o cálculo vai demorar numa máquina com 2 cores[5] será da ordem de 1.1s. Já no caso de usar $parmap$ em vez de map, fará o mesmo cálculo em cerca de 60% desse tempo.

Para verificar esta diferença siga as instruções seguintes:[6]

1. Compile o presente enunciado correndo:

```
ghc -O2 cp1415t -rtsopts -threaded
```

2. De seguida execute numa "shell" o seguinte comando,

[4] Esta explicação é bastante simplista, mas serve de momento. Para uma abordagem completa e elucidativa ver a referência [3].
[5] Intel Core 2 Duo a 2.53 GHz.
[6] Ver detalhes em [3].

```
./cp1415t exemplo seq +RTS -s -N2
```

onde o 2 em $N2$ indica 2 *cores* (se a máquina em questão tiver mais *cores*, este número deverá ser actualizado). Como pode ver inspecionando o código da função $main$ na secção A, o que vai ser executado é

$$putStrLn \cdot show \cdot (map\ fib)\ \$\ [20\,..\,30]$$

Das estatísticas que lhe aparecem no écran retenha esta:

```
Total   time    1.41s  (  1.11s elapsed)
```

Em particular, o campo $elapsed$ apresenta o tempo decorrido desde o início da execução do programa até ao respectivo fim.

3. De seguida execute

```
./cp1415t exemplo par +RTS -s -N2
```

que irá chamar, desta vez

$$putStrLn \cdot show \cdot runEval \cdot (parmap\ fib)\ \$\ [20\,..\,30]$$

A estatística correspondente à de cima será, desta vez, da ordem seguinte:

```
Total   time    1.13s  (  0.69s elapsed)
```

Em suma, a versão paralela é cerca de 1.61x mais rápida ($\frac{1.11}{0.69}$) que a sequencial.

6.4 Trabalho a realizar

Com base na definição de $parmap$ acima, defina a função

$$parBTreeMap :: (a \rightarrow b) \rightarrow (BTree\ a) \rightarrow Eval\ (BTree\ b)$$

que implemente o "map paralelo" sobre BTree's.

De seguida, corra testes semelhantes aos apresentados acima para apurar o ganho em *performance* da aplicação da função fib a todos os números da árvore $t1$ da secção 4.2, em duas versões:

1. $fmap\ fib$ (sem paralelismo, usando a função definida em BTree), ou
2. usando $parBTreeMap\ fib$.

Em máquinas mais rápidas e/ou com mais "cores" deve usar números maiores para obter uma melhor distinção entre as duas versões.

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] S. Marlow. *Parallel and Concurrent Programming in Haskell*. O'Reilly, 2013.

Anexos

A Programa principal

$$
\begin{array}{l}
\mathit{main} :: \mathit{IO}\ ()\\
\mathit{main} = \mathit{getArgs} \ggg\!= (\neg \cdot \mathit{null}) \rightarrow \mathit{exemp\_or\_exer}, \mathit{errInvArgs}\\
\quad \textbf{where}\\
\qquad \mathit{exemp\_or\_exer} = (((\equiv)\ \texttt{"exemplo"}) \cdot \mathit{head}) \rightarrow \mathit{exemp}, \mathit{exer}\\
\qquad \mathit{exemp} = (((\equiv)\ 2) \cdot \mathit{length}) \rightarrow \mathit{execExemp}, \mathit{errInvArgs}\\
\qquad \mathit{execExemp} = \mathit{isPar} \rightarrow \mathit{execExempPar}, \mathit{execExempSeq}\\
\qquad \mathit{exer} = (((\equiv)\ 3) \cdot \mathit{length}) \rightarrow \mathit{execExer}, \mathit{errInvArgs}\\
\qquad \mathit{execExer} = \mathit{isPar} \rightarrow \mathit{execExerPar}, \mathit{execExerSeq}\\
\qquad \mathit{execExempSeq} = \underline{(\mathit{putStrLn} \cdot \mathit{show} \cdot (\mathit{fmap}\ \mathit{fib})\ \$\ \mathit{abpe}\ (20, 30))}\\
\qquad \mathit{execExempPar} = \underline{(\mathit{putStrLn} \cdot \mathit{show} \cdot \mathit{runEval} \cdot (\mathit{parBTreeMap}\ \mathit{fib})\ \$\ \mathit{abpe}\ (20, 30))}
\end{array}
$$

B Bibliotecas e código auxiliar

$$
\begin{array}{l}
\mathit{errInvArgs} :: a \rightarrow \mathit{IO}\ ()\\
\mathit{errInvArgs} = \underline{\ }\ \$\ \mathit{putStrLn}\ \mathit{msgInvArgs}\\
\quad \textbf{where}\\
\quad \mathit{msgInvArgs} = \texttt{"Invalid arguments"}\\
\mathit{execExerPar} :: [\mathit{String}] \rightarrow \mathit{IO}\ ()\\
\mathit{execExerPar} = \perp\\
\mathit{execExerSeq} :: [\mathit{String}] \rightarrow \mathit{IO}\ ()\\
\mathit{execExerSeq} = \perp\\
\mathit{isPar} :: [\mathit{String}] \rightarrow \mathit{Bool}\\
\mathit{isPar} = (((\equiv)\ \texttt{"par"}) \cdot \mathit{head} \cdot \mathit{tail}) \rightarrow \underline{\mathit{True}}, \underline{\mathit{False}}\\
\mathit{pcataList}\ g = \mathit{mfoldr}\ (\mathit{curry}\ (g \cdot i_2))\ ((g \cdot i_1)\ ())\ \textbf{where}\\
\quad \mathit{mfoldr}\ f\ d\ [\,] = d\\
\quad \mathit{mfoldr}\ f\ d\ (a : x) = \textbf{do}\ \{ y \leftarrow \mathit{mfoldr}\ f\ d\ x; f\ a\ y \}
\end{array}
$$

B.1 "Easy X3DOM access"

Defina-se a seguinte composição de funções

$$
\mathit{x3dom} = \mathit{html} \cdot \mathit{preamble} \cdot \mathit{body} \cdot \mathit{x3d} \cdot \mathit{scene} \cdot \mathit{items}
$$

para gerar um texto HTML que represente um objecto gráfico em X3DOM. Esta função usa as seguintes funções auxiliares:

$$
\begin{array}{l}
\mathit{html} = \mathit{tag}\ \texttt{"html"}\ [\,]\\
\mathit{preamble} = \mathit{headx}\ \text{`}\mathit{with}\text{`}\ [\mathit{title}\ \texttt{"CP/X3DOM generation"}, \mathit{links}, \mathit{script}]\\
\mathit{body} = \mathit{tag}\ \texttt{"body"}\ [\,]\\
\mathit{x3d} = \mathit{tag}\ \texttt{"x3d"}\ [(\texttt{"width"}, \texttt{"\textbackslash"500px\textbackslash""}), (\texttt{"height"}, \texttt{"\textbackslash"400px\textbackslash""})]\\
\mathit{scene} = \mathit{tag}\ \texttt{"scene"}\ [\,]\\
\mathit{items} = \mathit{concat}\\
\mathit{links} = \mathit{ctag}\ \texttt{"link"}\ [\\
\quad (\texttt{"rel"}, \mathit{quote}\ \texttt{"stylesheet"}), (\texttt{"type"}, \mathit{quote}\ \texttt{"text/css"}),\\
\quad (\texttt{"href"}, \mathit{quote}\ \texttt{"http://www.x3dom.org/x3dom/release/x3dom.css"})]\\
\mathit{script} = \mathit{ctag}\ \texttt{"script"}\ [\\
\quad (\texttt{"type"}, \mathit{quote}\ \texttt{"text/javascript"}),
\end{array}
$$

$$
\begin{array}{l}
\quad (\texttt{"src"}, \mathit{quote}\ \texttt{"http://www.x3dom.org/x3dom/release/x3dom.js"})] \\
\mathit{ctag}\ t\ l = \mathit{tag}\ t\ l\ \texttt{""}
\end{array}
$$

onde

$$
\begin{array}{l}
\mathit{tag}\ t\ l\ x = \texttt{"<"} +\!\!+\ t +\!\!+ \texttt{" "} +\!\!+ \mathit{ps} +\!\!+ \texttt{">"} +\!\!+ x +\!\!+ \texttt{"</"} +\!\!+ t +\!\!+ \texttt{">"} \\
\quad \textbf{where}\ \mathit{ps} = \mathit{unwords}\ [\mathit{concat}\ [t, \texttt{"="}, v] \mid (t, v) \leftarrow l] \\
\mathit{headx} = \mathit{tag}\ \texttt{"head"}\ []
\end{array}
$$

De seguida dão-se mais algumas funções auxiliares facilitadoras:

$$
\begin{array}{l}
\mathit{transform}\ (x, y, z) = \mathit{tag}\ \texttt{"transform"}\ [(\texttt{"translation"}, \mathit{quote}\ (\mathit{show3D}\ (x, y, x)))] \\
\mathit{groupx}\ (x, y, z) = (\mathit{tag}\ \texttt{"group"}\ [(\texttt{"bboxSize"}, \mathit{quote}\ (\mathit{show3D}\ (x, y, z)))]) \cdot \mathit{items} \\
\mathit{shapex} = \mathit{tag}\ \texttt{"shape"}\ [] \\
\mathit{title} = \mathit{tag}\ \texttt{"title"}\ [] \\
\mathit{appearance} = \mathit{tag}\ \texttt{"appearance"}\ [] \\
\mathit{show3D}\ (x, y, z) = \mathit{show}\ x +\!\!+ \texttt{" "} +\!\!+ \mathit{show}\ y +\!\!+ \texttt{" "} +\!\!+ \mathit{show}\ z \\
t\ \text{`}\mathit{with}\text{`}\ l = ((t\ \$\ \mathit{items}\ l) +\!\!+) \\
\mathit{quote}\ s = \texttt{"\textbackslash""} +\!\!+ s +\!\!+ \texttt{"\textbackslash""} \\
\mathit{prime}\ s = \texttt{"'"} +\!\!+ s +\!\!+ \texttt{"'"} \\
\mathit{box}\ p\ \mathit{col} = (\mathit{transform}\ p \cdot \mathit{shapex} \cdot \mathit{items})\ [\mathit{color}\ \mathit{col}, \mathit{ctag}\ \texttt{"box"}\ [(\texttt{"size"}, \mathit{prime}\ \texttt{"2,2,2"})]] \\
\mathit{cone}\ p\ \mathit{col}\ b\ h = (\mathit{transform}\ p \cdot \mathit{shapex} \cdot \mathit{items}) \\
\quad [\mathit{color}\ \mathit{col}, \\
\quad\quad \mathit{ctag}\ \texttt{"cone"}\ [(\texttt{"bottomRadius"}, \mathit{prime}\ (\mathit{show}\ b)), (\texttt{"height"}, \mathit{prime}\ (\mathit{show}\ h))]] \\
\mathit{color}\ c = \mathit{appearance}\ (\mathit{ctag}\ \texttt{"material"}\ [(\texttt{"diffuseColor"}, \mathit{prime}\ c)])
\end{array}
$$

C Soluções propostas

Os alunos devem colocar neste anexo as suas soluções aos exercícios propostos, de acordo com o "layout" que se fornece. Podem ser adicionadas outras funções auxiliares que sejam necessárias.

Secção 4.1

C.0.1 Função depth:

Esta função permite auxiliar a função *balanced*, definida no enunciado, de maneira a retornar a profundidade de uma *LTree*, isto é, quantos níveis tem a árvore. Para isso tivemos de definir esta função usando o catamorfismo em *LTree*. Concluimos que o resultado ou ia ser 1 (one) ou então teriamos de descobrir onde (na sub-árvore da esquerda ou direita) existe o máximo de níveis e depois somar um a esse número devolvendo o resultado final ($\mathit{succ} \cdot \widehat{(\mathit{max})}$)

$$
\begin{array}{l}
\mathit{depth} :: \mathit{LTree}\ a \to \mathit{Integer} \\
\mathit{depth} = \mathit{cataLTree}\ [\mathit{one}, \mathit{succ} \cdot \widehat{(\mathit{max})}]
\end{array}
$$

C.0.2 Função balance:

Neste caso tivemos de refazer a função *balance* apenas numa linha recorrendo a funções da biblioteca *LTree*. Para isso tivemos de fazer o anamorfismo da *lsplit* que permite organizar a árvore de maneira a ela ficar balanceada após receber a lista com os elementos da árvore. Elementos estes que são providenciados pelo catamorfismo que guarda o valor de um nodo numa lista, seguido da destruição do mesmo.

$$
\begin{array}{l}
\mathit{balance} :: \mathit{LTree}\ a \to \mathit{LTree}\ a \\
\mathit{balance} = \mathit{anaLTree}\ (\mathit{lsplit}) \cdot \mathit{cataLTree}\ [\mathit{singl}, \widehat{(+\!\!+)}]
\end{array}
$$

Secção 4.2

C.0.3 Função qsplit:

Nesta secção tivemos de começar por definir o gene qsplit em haskell. O gene $qsplit$, dado um par de inteiros, cujo par não pode conter números iguais a 0, devolve uma árvore gerada com base nos inteiros fornecidos. Se o primeiro número for menor ou igual que o segundo, a árvore é construída colocando na raiz um s ($soma\ de\ ambos\ os\ numeros\ a\ dividir\ por\ dois$), e colocando numa sub-árvore cuja raiz é o nodo esquerdo da árvore original, o primeiro elemento do par e $s-1$, enquanto que na outra sub-árvore coloca $s+1$ e o segundo elemento do par, caso contrario não é criada a árvore.

$$
\begin{array}{l}
qsplit :: Integral\ a \Rightarrow (a, a) \rightarrow Either\ ()\ (a, ((a, a), (a, a))) \\
qsplit\ (x, y) = \mathbf{if}\ (x \equiv 0 \vee y \equiv 0)\ \mathbf{then}\ i_1\ () \\
\qquad\qquad\qquad\quad \mathbf{else\ let}\ s = x + y \div 2 \\
\qquad\qquad\qquad\qquad \mathbf{in\ if}\ x \leqslant y\ \mathbf{then}\ i_2\ (s, ((x, (s-1)), ((s+1), y))) \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{else}\ i_1\ ()
\end{array}
$$

Secção 4.3

C.0.4 Funções base da SList:

Este conjunto de funções é semelhante a outras bibliotecas usadas na disciplina, sendo estas adaptadas para a estrutura de dados $SList$.

A função $inSList$ faz a separação dos construtores Sent (se a lista apenas possuir a sentinela) e Cons (se houver uma lista completa finalizada por uma sentinela), a função $outSList$ faz o oposto.

A função $recSList$ assegura a recursividade das listas, $anaSList$ constroi uma $SList$, a $cataSList$ devolve um valor depois de fazer a recursividade, e por fim a $hyloSList$ é a composição das funções cataSList após a $anaSList$.

$$
\begin{array}{l}
inSList :: Either\ a\ (a1, SList\ a1\ a) \rightarrow SList\ a1\ a \\
inSList = [Sent, Cons] \\
outSList :: SList\ b\ a \rightarrow Either\ a\ (b, SList\ b\ a) \\
outSList\ (Sent\ a) = i_1\ a \\
outSList\ (Cons\ (b, a)) = i_2\ (b, a) \\
recSList\ f = id + (id \times f) \\
anaSList :: (c \rightarrow Either\ a\ (b, c)) \rightarrow c \rightarrow SList\ b\ a \\
anaSList\ f = inSList \cdot (recSList\ (anaSList\ f)) \cdot f \\
cataSList :: (Either\ b\ (a, d) \rightarrow d) \rightarrow SList\ a\ b \rightarrow d \\
cataSList\ a = a \cdot (recSList\ (cataSList\ a)) \cdot outSList \\
hyloSList :: (Either\ b\ (d, c) \rightarrow c) \rightarrow (a \rightarrow Either\ b\ (d, a)) \rightarrow a \rightarrow c \\
hyloSList\ a\ c = cataSList\ a \cdot anaSList\ c
\end{array}
$$

C.0.5 Função mgen:

A função $mgen$ recebe um par de listas ordenadas. Caso uma das listas seja vazia devolve a outra lista. Caso contrário devolve um tuplo em que o primeiro elemento é o maior da comparação entre as cabeças das listas, sendo o segundo um tuplo de listas identicas às recebidas como parâmetro excepto que irá ter o menor elemento da mesma removido.

$$
\begin{array}{l}
mgen :: Ord\ a \Rightarrow ([a], [a]) \rightarrow Either\ [a]\ (a, ([a], [a])) \\
mgen\ (l, [\,]) = i_1\ l \\
mgen\ ([\,], r) = i_1\ r \\
mgen\ ((x : xs), (y : ys)) = \mathbf{if}\ (x \leqslant y)\ \mathbf{then}\ i_2\ (x, (xs, y : ys)) \\
\quad \mathbf{else}\ i_2\ (y, (x : xs, ys))
\end{array}
$$

Secção 5.2

C.0.6 Funções base de TLTree:

Este conjunto de funções é semelhante a outras bibliotecas usadas na disciplina, sendo estas adaptadas para a estrutura de dados TLTree. A explicação destas funções são semelhantes às apresentadas anteriormente para a biblioteca SList.

$$inTLTree = [L, N]$$
$$outTLTree\ (L\ d) = i_1\ d$$
$$outTLTree\ (N\ (a, (b, c))) = (i_2\ (a, (b, c)))$$
$$baseTLTree\ g\ f = g + (f \times (f \times f))$$
$$recTLTree\ f = id + (f \times (f \times f))$$
$$anaTLTree\ f = (inTLTree \cdot (recTLTree\ (anaTLTree\ f) \cdot f))$$
$$cataTLTree\ a = a \cdot (recTLTree\ (cataTLTree\ a)) \cdot outTLTree$$
$$hyloTLTree\ a\ c = cataTLTree\ a \cdot anaTLTree\ c$$

C.0.7 Funções tips, count, depth e inv da TLTree:

A função $tipsTLTree$ devolve um lista com todos os elementos de uma TLTree. Para isto, aplicamos um catamorfismo que irá percorrer toda a árvore recursivamente, dando uma lista apenas com um elemento ($singl$) ou concatena as três sub-árvores através do conc.

A função $countTLTree$ conta quantos nós tem uma árvore. Aplicando o catamorfismo a função retorna 1 ($\underline{1}$) se a árvore apresentar só um nó ou retorna a soma de todos os nós presentes nas três sub-árvores caso contrário.

A função $depthTLTree$ calcula a profundidade da árvore TLTree.

Por último, a função $invTLTree$ inverte uma TLTree. Para fazer esta função, tivemos de fazer uma função auxiliar $swapT$ que aplica o swap a 3 elementos (x,(y,z)) = (z,(y,x))

$$tipsTLTree :: \mathsf{TLTree}\ b \to [b]$$
$$tipsTLTree = cataTLTree\ [singl, \mathsf{conc}]$$
$$\quad\quad\quad\quad \mathbf{where}\ \mathsf{conc}\ (l, (x, y)) = l + x + y$$
$$countTLTree :: \mathsf{TLTree}\ b \to Int$$
$$countTLTree = cataTLTree\ [\underline{1}, add]$$
$$\quad \mathbf{where}\ add\ (l, (x, y)) = l + x + y$$
$$depthTLTree :: \mathsf{TLTree}\ b \to Int$$
$$depthTLTree = cataTLTree\ [\underline{(1)}, succ \cdot \widehat{(max)} \cdot (id \times \widehat{(max)})]$$
$$invTLTree :: \mathsf{TLTree}\ b \to \mathsf{TLTree}\ b$$
$$invTLTree = cataTLTree\ (inTLTree \cdot (id + swapT))$$
$$\quad\quad\quad \mathbf{where}\ swapT :: (a, (b, c)) \to (c, (b, a))$$
$$\quad\quad\quad\quad swapT\ (x, (y, z)) = (z, (y, x))$$

C.0.8 Funções gera, geraSierp e apresentaSierp:

A função $gera$ recebe como parâmetros o triangulo original e um inteiro que simboliza o número de divisões que iremos fazer (o número de recursões da função).

A função $geraSierp$ utiliza o anamorfismo após a junção do Tri e Int num par, fazendo a recursividade na função $gera$.

A função $apresentaSierp$ recebe como parâmetro uma TLTree e devolve uma lista de elementos Tri.

$$gera :: (Tri, Int) \to Either\ Tri\ ((Tri, Int), (((Tri, Int), (Tri, Int))))$$
$$gera\ (t, 0) = i_1\ (t)$$
$$gera\ (((x, y), s), n) = \mathbf{let}\ s' = s \div 2$$
$$\quad\quad\quad\quad \mathbf{in}\ i_2\ ((((x, y), s'), n - 1), ((((x + s', y), s'), n - 1), (((x, y + s'), s'), n - 1)))$$
$$geraSierp :: Tri \to Int \to \mathsf{TLTree}\ Tri$$
$$geraSierp = curry\ (anaTLTree\ gera)$$

$$apresentaSierp :: \mathsf{TLTree}\ Tri \to [\,Tri\,]$$
$$apresentaSierp = cataTLTree\ [singl, \widehat{(++)} \cdot (id \times \widehat{(++)})]$$

C.0.9 Função rep e draw:

A função draw permite gerar a visualização do triângulo, num browser. A função *rep* tem como objetivo gerar o triângulo de Sierpinski, usando a função *gerar* , função esta que irá transformar a TLTree numa lista usando a função *apresentaSierp*. No final desta conversão a lista irá ser convertida num ficheiro *html* pela função *drawTriangle*, utilizando por fim a função *finalize* para ser possível ver o ficheiro *html* num browser.

$$\begin{array}{l} draw = render\ html\ \textbf{where} \\ \quad html = rep\ dados \\ rep = finalize \cdot concat \cdot (map\ drawTriangle) \cdot apresentaSierp \cdot gerar \\ \quad \textbf{where}\ gerar\ (tri, num) = geraSierp\ tri\ (fromIntegral\ num) \end{array}$$

Secção 6.2

A função *gene* executa um $[perderStop, perderPalavraLista]$ que devolve a probabilidade de se perder apenas a última palavra, "stop", a partir da função *perderStop*. Devolve também a probabilidade de se perder qualquer uma das palavras que fazem parte da frase, usando para este fim a função *perderPalavraLista*.

A função *gene* é utilizada na função $transmitir = pcataList\ gene$.

$$\begin{array}{l} gene = [perderStop, perderPalavraLista] \\ \quad \textbf{where} \\ \quad\quad perderPalavraLista\ (a, b) = D\ [((a : b), 0.95), (b, 0.05)] \\ \quad\quad perderStop\ a = (D\ [([\,], 0.1), ([\texttt{"stop"}], 0.9)]) \end{array}$$

Obtivemos o seguinte output ao executar o comando $transmitir\ (words$ `"Vamos atacar hoje"`$)$:

```
["Vamos","atacar","hoje","stop"]  77.2%
       ["Vamos","atacar","hoje"]   8.6%
        ["atacar","hoje","stop"]   4.1%
       ["Vamos","atacar","stop"]   4.1%
         ["Vamos","hoje","stop"]   4.1%
              ["Vamos","atacar"]   0.5%
                ["Vamos","hoje"]   0.5%
               ["atacar","hoje"]   0.5%
                ["Vamos","stop"]   0.2%
               ["atacar","stop"]   0.2%
                 ["hoje","stop"]   0.2%
                      ["atacar"]   0.0%
                       ["Vamos"]   0.0%
                        ["hoje"]   0.0%
                        ["stop"]   0.0%
                              []   0.0%
```

Podemos então verificar que:

- Existe 77.2% de probabilidade de transmitir a frase completa.
- Existe 8.6% de probabilidade de transmitir todos os elementos da frase excepto o *stop*.
- Existe 4.1% de probabilidade de perder qualquer uma das seguintes palavras: *vamos*,*atacar* ou *hoje*, embora nunca perdamos o *stop*.

Secção 6.4

A função $parBTreeMap$ tem como objetivo aplicar a função previamente definida $rpar$ a todos os elementos de uma $BTree$ através da invocação da função $parBTreeMap$ recursiva pelo ciclo **do**.

$$
\begin{array}{l}
parBTreeMap\ t\ (Empty) = return\ Empty;\\
parBTreeMap\ t\ (Node\ (r, (e, d))) = \mathbf{do}\\
\quad r' \leftarrow rpar\ (t\ r)\\
\quad e' \leftarrow parBTreeMap\ t\ e\\
\quad d' \leftarrow parBTreeMap\ t\ d\\
\quad return\ (Node\ (r', (e', d')))
\end{array}
$$

Obtivemos os seguinte resultados após executar a versão sequencial e paralela, respetivamente:

```
Total   time    1.33s  (  0.84s elapsed)
Total   time    0.70s  (  0.38s elapsed)
```

Em suma, a versão paralela é cerca de 2.21x mais rápida ($\frac{0.84}{0.38}$) que a sequencial.

Índice