

Cálculo de Programas

Trabalho Prático

LEI+MiEI — 2021/22

Departamento de Informática
Universidade do Minho

Janeiro de 2022

Grupo nr.	26
a89528	Luís Magalhães
a85829	Meriam Khammassi
a89578	Patícia Pereira
a88220	Xavier Mota

1 Preâmbulo

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 cursos que têm esta disciplina, opta-se pela aplicação deste método à programação em **Haskell** (sem prejuízo da sua aplicação a outras linguagens funcionais). 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, a validá-los e a produzir textos técnico-científicos de qualidade.

Antes de abordar os problemas propostos no trabalho, os grupos devem ler com atenção o anexo [A](#) onde encontrarão as instruções relativas ao software a instalar, etc.

Problema 1

Num sistema de informação distribuído, uma lista não vazia de transações é vista como um *blockchain* sempre que possui um valor de *hash* que é dado pela raiz de uma *Merkle tree* que lhe está associada. Isto significa que cada *blockchain* está estruturado numa *Merkle tree*. Mas, o que é uma *Merkle tree*?

Uma *Merkle tree* é uma *FTree* com as seguintes propriedades:

1. as folhas são pares (*hash*, transação) ou simplesmente o *hash* de uma transação;
2. os nodos são *hashes* que correspondem à concatenação dos *hashes* dos filhos;
3. o *hash* que se encontra na raiz da árvore é designado *Merkle Root*; como se disse acima, corresponde ao valor de *hash* de todo o bloco de transações.

(1)

Assumindo uma lista não vazia de transações, o algoritmo clássico de construção de uma *Merkle Tree* é o que está dado na Figura 1. Contudo, este algoritmo (que se pode mostrar ser um hilomorfismo de listas não vazias) é demasiadamente complexo. Uma forma bem mais simples de produzir uma *Merkle Tree* é através de um hilomorfismo de *LTrees*. Começa-se por, a partir da lista de transações, construir uma *LTree* cujas folhas são as transações:

$$list2LTree :: [a] \rightarrow LTree\ a$$

- Se a lista for singular, calcular o hash da transação.
- Caso contrário,
 1. Mapear a lista com a função hash.
 2. Se o comprimento da lista for ímpar, concatenar a lista com o seu último valor (que fica duplicado). Caso contrário, a lista não sofre alterações.
 3. Agrupar a lista em pares.
 4. Concatenar os hashes do par produzindo uma lista de (sub-)árvores nas quais a cabeça terá a respetiva concatenação.
 5. Se a lista de (sub-)árvores não for singular, voltar ao passo 2 com a lista das cabeças como argumento, preservando a lista de (sub-)árvores. Se a lista for singular, chegamos à Merkle Root. Contudo, falta compor a Merkle Tree final. Para tal, tendo como resultado uma lista de listas de (sub-)árvores agrupada pelos níveis da árvore final, é necessário encaixar sucessivamente os tais níveis formando a Merkle Tree completa.

Figura 1: Algoritmo clássico de construção de uma Merkle tree [4].

Depois, o objetivo é etiquetar essa árvore com os hashes,

$$lTree2MTree :: Hashable a \Rightarrow LTree a \rightarrow \underbrace{FTree \mathbb{Z} (\mathbb{Z}, a)}_{Merkle tree}$$

formando uma Merkle tree que satisfaça os três requisitos em (1). Em suma, a construção de um blockchain é um hilomorfismo de LTrees

$$\begin{aligned} computeMerkleTree &:: Hashable a \Rightarrow [a] \rightarrow FTree \mathbb{Z} (\mathbb{Z}, a) \\ computeMerkleTree &= lTree2MTree \cdot list2LTree \end{aligned}$$

1. Comece por definir o gene do anamorfismo que constrói LTrees a partir de listas não vazias:

$$\begin{aligned} list2LTree &:: [a] \rightarrow LTree a \\ list2LTree &= \llbracket g_list2LTree \rrbracket \end{aligned}$$

NB: para garantir que $list2LTree$ não aceita listas vazias deverá usar em $g_list2LTree$ o inverso $outNEList$ do isomorfismo

$$inNEList = [singl, cons]$$

2. Assumindo as seguintes funções hash e concHash:¹

$$\begin{aligned} hash &:: Hashable a \Rightarrow a \rightarrow \mathbb{Z} \\ hash &= toInteger \cdot (Data.Hashable.hash) \\ concHash &:: (\mathbb{Z}, \mathbb{Z}) \rightarrow \mathbb{Z} \\ concHash &= add \end{aligned}$$

defina o gene do catamorfismo que consome a LTree e produz a correspondente Merkle tree etiquetada com todos os hashes:

$$\begin{aligned} lTree2MTree &:: Hashable a \Rightarrow LTree a \rightarrow FTree \mathbb{Z} (\mathbb{Z}, a) \\ lTree2MTree &= \llbracket g_lTree2MTree \rrbracket \end{aligned}$$

3. Defina g_mroot por forma a

$$\begin{aligned} mroot &:: Hashable b \Rightarrow [b] \rightarrow \mathbb{Z} \\ mroot &= \llbracket g_mroot \rrbracket \cdot computeMerkleTree \end{aligned}$$

nos dar a Merkle root de um qualquer bloco $[b]$ de transações.

¹Para invocar a função hash, escreva $Main.hash$.

4. Calcule *mroot trs* da sequência de transações *trs* da no anexo e verifique que, sempre que se modifica (e.g. fraudulentamente) uma transação passada em *trs*, *mroot trs* altera-se necessariamente. Porquê? (Esse é exactamente o princípio de funcionamento da tecnologia **blockchain**.)

Valorização (não obrigatória): implemente o algoritmo clássico de construção de **Merkle trees**

```
classicMerkleTree :: Hashable a => [a] -> FTree Z Z
```

sob a forma de um hilomorfismo de listas não vazias. Para isso deverá definir esse combinador primeiro, da forma habitual:

```
hyloNEList h g = cataNEList h · anaNEList g
```

etc. Depois passe à definição do gene *g-pairsList* do anamorfismo de listas

```
pairsList :: [a] -> [(a, a)]
pairsList = [(g-pairsList)]
```

que agrupa a lista argumento por pares, duplicando o último valor caso seja necessário. Para tal, poderá usar a função (já definida)

```
getEvenBlock :: [a] -> [a]
```

que, dada uma lista, se o seu comprimento for ímpar, duplica o último valor.

Por fim, defina os genes *divide* e *conquer* dos respetivos anamorfismo e catamorfismo por forma a

```
classicMerkleTree = (hyloNEList conquer divide) · (map Main.hash)
```

Para facilitar a definição do *conquer*, terá apenas de definir o gene *g-mergeMerkleTree* do catamorfismo de ordem superior

```
mergeMerkleTree :: FTree a p -> [FTree a c] -> FTree a c
mergeMerkleTree = [(g-mergeMerkleTree)]
```

que compõe a **FTree** (à cabeça) com a lista de **FTrees** (como filhos), fazendo um “merge” dos valores intermédios. Veja o seguinte exemplo de aplicação da função *mergeMerkleTree*:

```
> l = [Comp 3 (Unit 1, Unit 2), Comp 7 (Unit 3, Unit 4)]
>
> m = Comp 10 (Unit 3, Unit 7)
>
> mergeMerkleTree m l
Comp 10 (Comp 3 (Unit 1,Unit 2),Comp 7 (Unit 3,Unit 4))
```

NB: o *classicMerkleTree* retorna uma Merkle Tree cujas folhas são apenas o *hash* da transação e não o par (*hash*, transação).

Problema 2

Se se digitar **man wc** na shell do Unix (Linux) obtém-se:

NAME

wc -- word, line, character, and byte count

SYNOPSIS

wc [-clmw] [file ...]

DESCRIPTION

The wc utility displays the number of lines, words, and bytes contained in each input file, or standard input (if no file is specified) to the standard output. A line is defined as a string of characters delimited by a <newline> character. Characters beyond the final <newline> character will not be included in the line count.

(...)

```

The following options are available:
(...)
    -w    The number of words in each input file is written to the standard
           output.
(...)

```

Se olharmos para o código da função que, em C, implementa esta funcionalidade [1] e nos focarmos apenas na parte que implementa a opção `-w`, verificamos que a poderíamos escrever, em Haskell, da forma seguinte:

```

wc_w :: [Char] → Int
wc_w [] = 0
wc_w (c : l) =
  if ¬ (sep c) ∧ lookahead_sep l then wc_w l + 1 else wc_w l
  where
    sep c = (c ≡ ' ' ∨ c ≡ '\n' ∨ c ≡ '\t')
    lookahead_sep [] = True
    lookahead_sep (c : l) = sep c

```

Por aplicação da lei de recursividade mútua

$$\left\{ \begin{array}{l} f \cdot \text{in} = h \cdot F \langle f, g \rangle \\ g \cdot \text{in} = k \cdot F \langle f, g \rangle \end{array} \right. \equiv \langle f, g \rangle = \llbracket \langle h, k \rangle \rrbracket \quad (2)$$

às funções `wc_w` e `lookahead_sep`, re-implemente a primeira segundo o modelo *worker/wrapper* onde *worker* deverá ser um catamorfismo de listas:

```

wc_w_final :: [Char] → Int
wc_w_final = wrapper ·  $\underbrace{\llbracket [g1, g2] \rrbracket}_{\text{worker}}$ 

```

Apresente os cálculos que fez para chegar à versão `wc_w_final` de `wc_w`, com indicação dos genes h , k e $g = [g1, g2]$.

Problema 3

Neste problema pretende-se gerar o HTML de uma página de um jornal descrita como uma agregação estruturada de blocos de texto ou imagens:

```

data Unit a b = Image a | Text b deriving Show

```

O tipo *Sheet* (=“página de jornal”)

```

data Sheet a b i = Rect (Frame i) (X (Unit a b) (Mode i)) deriving Show

```

é baseado num tipo indutivo X que, dado em anexo (pág. 10), exprime a partição de um rectângulo (a página tipográfica) em vários subrectângulos (as caixas tipográficas a encher com texto ou imagens), segundo um processo de partição binária, na horizontal ou na vertical. Para isso, o tipo

```

data Mode i = Hr i | Hl i | Vt i | Vb i deriving Show

```

especifica quatro variantes de partição. O seu argumento deverá ser um número de 0 a 1, indicando a fracção da altura (ou da largura) em que o rectângulo é dividido, a saber:

- `Hr i` — partição horizontal, medindo i a partir da direita
- `Hl i` — partição horizontal, medindo i a partir da esquerda
- `Vt i` — partição vertical, medindo i a partir do topo
- `Vb i` — partição vertical, medindo i a partir da base

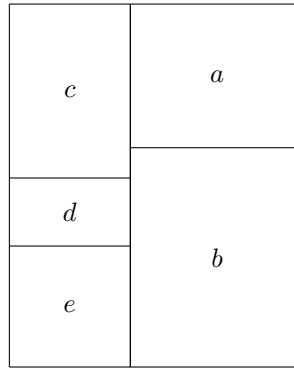
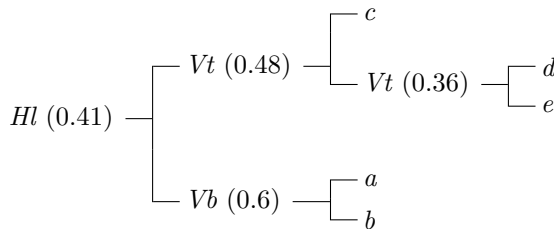


Figura 2: Layout de página de jornal.

Por exemplo, a partição dada na figura 2 corresponde à partição de um rectângulo de acordo com a seguinte árvore de partições:



As caixas delineadas por uma partição (como a dada acima) correspondem a folhas da árvore de partição e podem conter texto ou imagens. É o que se verifica no objecto *example* da secção B que, processado por *sheet2html* (secção B) vem a produzir o ficheiro `jornal.html`.

O que se pretende O código em **Haskell** fornecido no anexo B como “kit” para arranque deste trabalho não está estruturado em termos dos combinadores *cata-ana-hylo* estudados nesta disciplina. O que se pretende é, então:

1. A construção de uma biblioteca “pointfree”² com base na qual o processamento (“pointwise”) já disponível possa ser redefinido.
2. A evolução da biblioteca anterior para uma outra que permita partições n -árias (para *qualquer* n finito) e não apenas binárias.³

Problema 4

Este exercício tem como objectivo determinar todos os caminhos possíveis de um ponto A para um ponto B . Para tal, iremos utilizar técnicas de *brute force* e *backtracking*, que podem ser codificadas no mónade das listas (estudado na **aulas**). Comece por implementar a seguinte função auxiliar:

1. $pairL :: [a] \rightarrow [(a, a)]$ que dada uma lista l de tamanho maior que 1 produz uma nova lista cujos elementos são os pares (x, y) de elementos de l tal que x precede imediatamente y . Por exemplo:

$$\begin{aligned}
 pairL \ [1, 2] &\equiv [(1, 2)], \\
 pairL \ [1, 2, 3] &\equiv [(1, 2), (2, 3)] \text{ e} \\
 pairL \ [1, 2, 3, 4] &\equiv [(1, 2), (2, 3), (3, 4)]
 \end{aligned}$$

Para o caso em que $l = [x]$, i.e. o tamanho de l é 1, assuma que $pairL \ [x] \equiv [(x, x)]$. Implemente esta função como um *anamorfismo de listas*, atentando na sua propriedade:

²A desenvolver de forma análoga a outras bibliotecas que conhece (eg. **LTree**, etc).

³Repare que é a falta desta capacidade expressiva que origina, no “kit” actual, a definição das funções auxiliares da secção B, por exemplo.

- Para todas as listas l de tamanho maior que 1, a lista `map π_1 (pairL l)` é a lista original l a menos do último elemento. Analogamente, a lista `map π_2 (pairL l)` é a lista original l a menos do primeiro elemento.

De seguida necessitamos de uma estrutura de dados representativa da noção de espaço, para que seja possível formular a noção de *caminho* de um ponto A para um ponto B , por exemplo, num papel quadriculado. No nosso caso vamos ter:

```
data Cell = Free | Blocked | Lft | Rght | Up | Down deriving (Eq, Show)
type Map = [[Cell]]
```

O terreno onde iremos navegar é codificado então numa *matriz* de células. Os valores *Free* and *Blocked* denotam uma célula como livre ou bloqueada, respectivamente (a navegação entre dois pontos terá que ser realizada *exclusivamente* através de células livres). Ao correr, por exemplo, `putStr $ showM $ map1` no interpretador irá obter a seguinte apresentação de um mapa:

```
— — —
— X —
— X —
```

Para facilitar o teste das implementações pedidas abaixo, disponibilizamos no anexo B a função `testWithRndMap`. Por exemplo, ao correr `testWithRndMap` obtivemos o seguinte mapa aleatoriamente:

```
— — — — — X — — X
— X — — — X — — —
— — — — — X — — —
— X — — — — — — X
— — — — — — — X —
— — — — — — — — —
— X X — — X — — — —
— — — — — — — — X —
— — — — — X — — X —
— — X — — — — — — X
Map of dimension 10x10.
```

De seguida, os valores *Lft*, *Rght*, *Up* e *Down* em *Cell* denotam o facto de uma célula ter sido alcançada através da célula à esquerda, direita, de cima, ou de baixo, respectivamente. Tais valores irão ser usados na representação de caminhos num mapa.

2. Implemente agora a função `markMap :: [Pos] → Map → Map`, que dada uma lista de posições (representante de um *caminho* de um ponto A para um ponto B) e um mapa retorna um novo mapa com o caminho lá marcado. Por exemplo, ao correr no interpretador,

```
putStr $ showM $ markMap [(0,0), (0,1), (0,2), (1,2)] map1
```

deverá obter a seguinte apresentação de um mapa e respectivo caminho:

```
> — —
^ X —
^ X —
```

representante do caso em que subimos duas vezes no mapa e depois viramos à direita. Para implementar a função `markMap` deverá recorrer à função `toCell` (disponibilizada no anexo B) e a uma função auxiliar com o tipo `[(Pos, Pos)] → Map → Map` definida como um *catamorfismo de listas*. Tal como anteriormente, anote as propriedades seguintes sobre `markMap`.⁴

- Para qualquer lista l a função `markMap l` é idempotente.
- Todas as posições presentes na lista dada como argumento irão fazer com que as células correspondentes no mapa deixem de ser *Free*.

⁴Ao implementar a função `markMap`, estude também a função `subst` (disponibilizada no anexo B) pois as duas funções tem algumas semelhanças.

Finalmente há que implementar a função $scout :: Map \rightarrow Pos \rightarrow Pos \rightarrow Int \rightarrow [[Pos]]$, que dado um mapa m , uma posição inicial s , uma posição alvo t , e um número inteiro n , retorna uma lista de caminhos que começam em s e que têm tamanho máximo $n + 1$. Nenhum destes caminhos pode conter t como elemento que não seja o último na lista (i.e. um caminho deve terminar logo que se alcança a posição t). Para além disso, não é permitido voltar a posições previamente visitadas e se ao alcançar uma posição diferente de t é impossível sair dela então todo o caminho que levou a esta posição deve ser removido (*backtracking*). Por exemplo:

```
scout map1 (0,0) (2,0) 0  $\equiv$  [[(0,0)]]
scout map1 (0,0) (2,0) 1  $\equiv$  [[(0,0), (0,1)]]
scout map1 (0,0) (2,0) 4  $\equiv$  [[(0,0), (0,1), (0,2), (1,2), (2,2)]]
scout map2 (0,0) (2,2) 2  $\equiv$  [[(0,0), (0,1), (1,1)], [(0,0), (0,1), (0,2)]]
scout map2 (0,0) (2,2) 4  $\equiv$  [[(0,0), (0,1), (1,1), (2,1), (2,2)], [(0,0), (0,1), (1,1), (2,1), (2,0)]]
```

3. Implemente a função

$scout :: Map \rightarrow Pos \rightarrow Pos \rightarrow Int \rightarrow [[Pos]]$

recorrendo à função *checkAround* (disponibilizada no anexo B) e de tal forma a que $scout\ m\ s\ t$ seja um catamorfismo de naturais *monádico*. Anote a seguinte propriedade desta função:

- Quanto maior for o tamanho máximo permitido aos caminhos, mais caminhos que alcançam a posição alvo iremos encontrar.

Anexos

A Documentação para realizar o trabalho

Para cumprir de forma integrada os objectivos Rdo trabalho vamos recorrer a uma técnica de programação dita “**literária**” [2], cujo princípio base é o seguinte:

Um programa e a sua documentação devem coincidir.

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

O ficheiro `cp2122t.pdf` que está a ler é já um exemplo de **programação literária**: foi gerado a partir do texto fonte `cp2122t.lhs`⁵ que encontrará no **material pedagógico** desta disciplina descompactando o ficheiro `cp2122t.zip` e executando:

```
$ lhs2TeX cp2122t.lhs > cp2122t.tex
$ pdflatex cp2122t
```

em que **lhs2tex** é um pre-processador que faz “pretty printing” de código Haskell em **L^AT_EX** e que deve desde já instalar executando

```
$ cabal install lhs2tex --lib
$ cabal install --ghc-option=-dynamic lhs2tex
```

NB: utilizadores do macOS poderão instalar o *cabal* com o seguinte comando:

```
$ brew install cabal-install
```

Por outro lado, o mesmo ficheiro `cp2122t.lhs` é executável e contém o “kit” básico, escrito em **Haskell**, para realizar o trabalho. Basta executar

```
$ ghci cp2122t.lhs
```

Abra o ficheiro `cp2122t.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}
```

é seleccionado pelo **GHCi** para ser executado.

A.1 Como realizar o trabalho

Este trabalho teórico-prático deve ser realizado por grupos de 3 (ou 4) 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 em todos os exercícios do trabalho, para assim poderem responder a qualquer questão colocada na *defesa oral* do relatório.

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

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

```
$ bibtex cp2122t.aux
$ makeindex cp2122t.idx
```

e recompilar o texto como acima se indicou. Dever-se-á ainda instalar o utilitário **QuickCheck**, que ajuda a validar programas em **Haskell**:

```
$ cabal install QuickCheck --lib
```

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

⁵O sufixo ‘lhs’ quer dizer *literate Haskell*.


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

Pode-se ainda controlar o número de casos de teste e sua complexidade, como o seguinte exemplo mostra:⁶

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

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

Stack O **Stack** é um programa útil para criar, gerir e manter projetos em **Haskell**. Um projeto criado com o Stack possui uma estrutura de pastas muito específica:

- Os módulos auxiliares encontram-se na pasta *src*.
- O módulo principal encontra-se na pasta *app*.
- A lista de dependências externas encontra-se no ficheiro *package.yaml*.

Pode aceder ao **GHCI** utilizando o comando:

```
stack ghci
```

Garanta que se encontra na pasta mais externa **do projeto**. A primeira vez que correr este comando as dependências externas serão instaladas automaticamente. Para gerar o PDF, garanta que se encontra na diretoria *app*.

A.2 Como exprimir cálculos e diagramas em LaTeX/lhs2tex

Como primeiro exemplo, estudar o texto fonte deste trabalho para obter o efeito:⁷

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

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

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

⁶Como já sabe, os testes normalmente não provam a ausência de erros no código, apenas a sua presença (cf. [arquivo online](#)). Portanto não deve ver o facto de o seu código passar nos testes abaixo como uma garantia que este está livre de erros.

⁷Exemplos tirados de [3].

B Código fornecido

Problema 1

Sequência de transações para teste:

```
trs = [("compra", "20211102", -50),
       ("venda",   "20211103", 100),
       ("despesa", "20212103", -20),
       ("venda",   "20211205", 250),
       ("venda",   "20211205", 120)]
```

```
getEvenBlock :: [a] → [a]
getEvenBlock l = if (even (length l)) then l else l ++ [last l]
firsts = [π1, π1]
```

Problema 2

```
wc_test = "Here is a sentence, for testing.\nA short one."
sp c = (c ≡ ' ' ∨ c ≡ '\n' ∨ c ≡ '\t')
```

Problema 3

Tipos:

```
data X u i = XLeaf u | Node i (X u i) (X u i) deriving Show
data Frame i = Frame i i deriving Show
```

Funções da API⁸

```
printJournal :: Sheet String String Double → IO ()
printJournal = write · sheet2html
write :: String → IO ()
write s = do writeFile "jornal.html" s
          putStrLn "Output HTML written into file 'jornal.html' "
```

Geração de HTML:

```
sheet2html (Rect (Frame w h) y) = htmlwrap (x2html y (w, h))
x2html :: X (Unit String String) (Mode Double) → (Double, Double) → String
x2html (XLeaf (Image i)) (w, h) = img w h i
x2html (XLeaf (Text txt)) _ = txt
x2html (Node (Vt i) x1 x2) (w, h) = htab w h (
  tr (td w (h * i) (x2html x1 (w, h * i))) ++
  tr (td w (h * (1 - i)) (x2html x2 (w, h * (1 - i))))
)
x2html (Node (Hl i) x1 x2) (w, h) = htab w h (
  tr (td (w * i) h (x2html x1 (w * i, h))) ++
  td (w * (1 - i)) h (x2html x2 (w * (1 - i), h)))
)
x2html (Node (Vb i) x1 x2) m = x2html (Node (Vt (1 - i)) x1 x2) m
x2html (Node (Hr i) x1 x2) m = x2html (Node (Hl (1 - i)) x1 x2) m
```

Funções auxiliares:

⁸API (=“Application Program Interface”).

```

twoVtImg a b = Node (Vt 0.5) (XLeaf (Image a)) (XLeaf (Image b))
fourInArow a b c d =
  Node (Hl 0.5)
    (Node (Hl 0.5) (XLeaf (Text a)) (XLeaf (Text b)))
    (Node (Hl 0.5) (XLeaf (Text c)) (XLeaf (Text d)))

```

HTML:

```

htmlwrap = html · hd · (title "CP/2122 - sheet2html") · body · divt
html = tag "html" [] · ("<meta charset=\"utf-8\" />"++)
title t = (tag "title" [] t++)
body = tag "body" ["BGColor" ↦ show "#F4EFD8"]
hd = tag "head" []
htab w h = tag "table" [
  "width" ↦ show2 w, "height" ↦ show2 h,
  "cellpadding" ↦ show2 0, "border" ↦ show "1px"]
tr = tag "tr" []
td w h = tag "td" ["width" ↦ show2 w, "height" ↦ show2 h]
divt = tag "div" ["align" ↦ show "center"]
img w h i = tag "img" ["width" ↦ show2 w, "src" ↦ show i] ""
tag t l x = "<" ++ t ++ " " ++ ps ++ ">" ++ x ++ "</" ++ t ++ ">\n"
  where ps = unwords [concat [t, "=", v] | (t, v) ← l]
a ↦ b = (a, b)
show2 :: Show a ⇒ a → String
show2 = show · show

```

Exemplo para teste:

```

example :: (Fractional i) ⇒ Sheet String String i
example =
  Rect (Frame 650 450)
    (Node (Vt 0.01)
      (Node (Hl 0.15)
        (XLeaf (Image "cp2122t_media/publico.jpg"))
        (fourInArow "Jornal Público" "Domingo, 5 de Dezembro 2021" "Simulação para efe
      (Node (Vt 0.55)
        (Node (Hl 0.55)
          (Node (Vt 0.1)
            (XLeaf (Text
              "Universidade do Algarve estuda planta capaz de eliminar a doença do sol
            (XLeaf (Text
              "Organismo (semelhante a um fungo) ataca de forma galopante os montado
          (XLeaf (Image
            "cp2122t_media/1647472.jpg"))
        (Node (Hl 0.25)
          (twoVtImg
            "cp2122t_media/1647981.jpg"
            "cp2122t_media/1647982.jpg")
          (Node (Vt 0.1)
            (XLeaf (Text "Manchester United vence na estreia de Rangnick"))
            (XLeaf (Text "O Manchester United venceu, este domingo, em Old Trafford,

```

Problema 4

Exemplos de mapas:

```

map1 = [[Free, Blocked, Free], [Free, Blocked, Free], [Free, Free, Free]]
map2 = [[Free, Blocked, Free], [Free, Free, Free], [Free, Blocked, Free]]
map3 = [[Free, Free, Free], [Free, Blocked, Free], [Free, Blocked, Free]]

```

Código para impressões de mapas e caminhos:

```

showM :: Map → String
showM = unlines · (map showL) · reverse

showL :: [Cell] → String
showL = ([f1, f2]) where
  f1 = " "
  f2 = (++) · (fromCell × id)

fromCell Lft = " > "
fromCell Rght = " < "
fromCell Up = " ^ "
fromCell Down = " v "
fromCell Free = " _ "
fromCell Blocked = " x "

toCell (x, y) (w, z) | x < w = Lft
toCell (x, y) (w, z) | x > w = Rght
toCell (x, y) (w, z) | y < z = Up
toCell (x, y) (w, z) | y > z = Down

```

Código para validação de mapas (útil, por exemplo, para testes **QuickCheck**):

```

ncols :: Map → Int
ncols = [0, length · π1] · outList

nlines :: Map → Int
nlines = length

isValidMap :: Map → Bool
isValidMap = (∧) · ⟨isSquare, sameLength⟩ where
  isSquare = (≡) · ⟨nlines, ncols⟩
  sameLength [] = True
  sameLength [x] = True
  sameLength (x1 : x2 : y) = length x1 ≡ length x2 ∧ sameLength (x2 : y)

```

Código para geração aleatória de mapas e automatização de testes (envolve o mónade IO):

```

randomRIOL :: (Random a) ⇒ (a, a) → Int → IO [a]
randomRIOL x = ([f1, f2]) where
  f1 = return []
  f2 l = do r1 ← randomRIO x
            r2 ← l
            return $ r1 : r2

buildMat :: Int → Int → IO [[Int]]
buildMat n = ([f1, f2]) where
  f1 = return []
  f2 l = do x ← randomRIOL (0 :: Int, 3 :: Int) n
            y ← l
            return $ x : y

testWithRndMap :: IO ()
testWithRndMap = do
  dim ← randomRIO (2, 10) :: IO Int
  out ← buildMat dim dim
  map ← return $ map (map table) out
  putStr $ showM map
  putStrLn $ "Map of dimension " ++ (show dim) ++ "x" ++ (show dim) ++ " ."

```

```

putStr "Please provide a target position (must be different from (0,0)):"
t ← readLn :: IO (Int, Int)
putStr "Please provide the number of steps to compute:"
n ← readLn :: IO Int
let paths = hasTarget t (scout map (0,0) t n) in
  if length paths == 0
  then putStrLn "No paths found."
  else putStrLn $ "There are at least " ++ (show $ length paths) ++
    " possible paths. Here is one case: \n" ++ (showM $ markMap (head paths) map )
table 0 = Free
table 1 = Free
table 2 = Free
table 3 = Blocked
hasTarget y = filter (λl → elem y l)

```

Funções auxiliares $subst :: a \rightarrow Int \rightarrow [a] \rightarrow [a]$, que dado um valor x e um inteiro n , produz uma função $f : [a] \rightarrow [a]$ que dada uma lista l substitui o valor na posição n dessa lista pelo valor x :

```

subst :: a → Int → [a] → [a]
subst x = ([f1, f2]) where
  f1 = λl → x : tail l
  f2 f (h : t) = h : f t

```

$checkAround :: Map \rightarrow Pos \rightarrow [Pos]$, que verifica se as células adjacentes estão livres:

```

type Pos = (Int, Int)
checkAround :: Map → Pos → [Pos]
checkAround m p = concat $ map (λf → f m p)
  [checkLeft, checkRight, checkUp, checkDown]
checkLeft :: Map → Pos → [Pos]
checkLeft m (x, y) = if x == 0 ∨ (m !! y) !! (x - 1) == Blocked
  then [] else [(x - 1, y)]
checkRight :: Map → Pos → [Pos]
checkRight m (x, y) = if x == (ncols m - 1) ∨ (m !! y) !! (x + 1) == Blocked
  then [] else [(x + 1, y)]
checkUp :: Map → Pos → [Pos]
checkUp m (x, y) = if y == (nlines m - 1) ∨ (m !! (y + 1)) !! x == Blocked
  then [] else [(x, y + 1)]
checkDown :: Map → Pos → [Pos]
checkDown m (x, y) = if y == 0 ∨ (m !! (y - 1)) !! x == Blocked
  then [] else [(x, y - 1)]

```

QuickCheck

Lógicas:

```

infixr 0 ⇒
(⇒) :: (Testable prop) ⇒ (a → Bool) → (a → prop) → a → Property
p ⇒ f = λa → p a ⇒ f a
infixr 0 ⇔
(⇔) :: (a → Bool) → (a → Bool) → a → Property
p ⇔ f = λa → (p a ⇒ property (f a)) .&&. (f a ⇒ property (p a))
infixr 4 ≡
(≡) :: Eq b ⇒ (a → b) → (a → b) → (a → Bool)
f ≡ g = λa → f a == g a

```

```

infixr 4 ≤
(≤) :: Ord b => (a → b) → (a → b) → (a → Bool)
f ≤ g = λa → f a ≤ g a

infixr 4 ∧
(∧) :: (a → Bool) → (a → Bool) → (a → Bool)
f ∧ g = λa → (f a) ∧ (g a)

instance Arbitrary Cell where
  -- 1/4 chance of generating a cell 'Block'.
  arbitrary = do x ← chooseInt (0,3)
  return $ f x where
    f x = if x < 3 then Free else Blocked

```

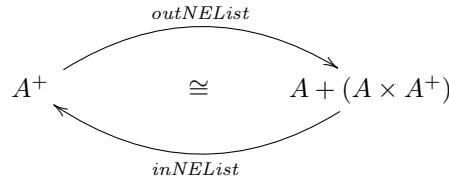
C Soluções dos alunos

Problema 1

Para garantir que *list2LTree* não aceita listas vazias é pedido para definir *outNEList*. Uma vez que conhecemos o *inNEList* podemos calcular o seu inverso da seguinte forma:

$$\begin{aligned}
& outNEList \cdot inNEList = id \\
\equiv & \{ \text{inNEList} \} \\
& outNEList \cdot [single, cons] = id \\
\equiv & \{ \text{Fusão-+ (20)} \} \\
& [outNEList \cdot single, outNEList \cdot cons] = id \\
\equiv & \{ \text{Universal-+ (17)} \} \\
& \begin{cases} id \cdot i_1 = outNEList \cdot single \\ id \cdot i_2 = outNEList \cdot cons \end{cases} \\
\equiv & \{ \text{Natural-id (1), Igualdade Extensional (69), Def-comp (70)} \} \\
& \begin{cases} outNEList (single a) = i_1 a \\ outNEList (cons (h, t)) = i_2 (h, t) \end{cases}
\end{aligned}$$

Assim o tipo do *outNEList* é:



Listas não vazias:

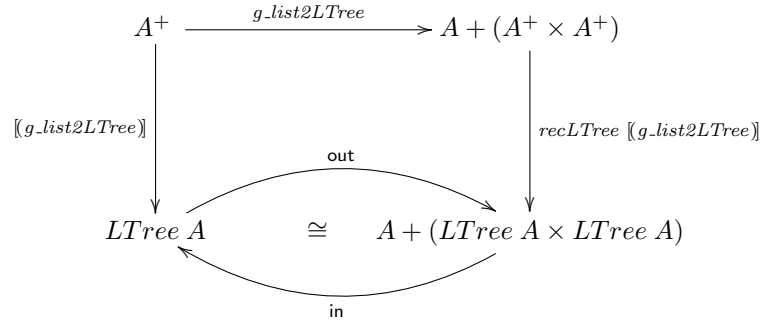
```

outNEList [a] = i1 a
outNEList (h : t) = i2 (h, t)
baseNEList f g = f + (f × g)
recNEList f = baseNEList id f
cataNEList g = g · recNEList (cataNEList g) · outNEList
anaNEList g = inNEList · recNEList (anaNEList g) · g
hyloNEList h g = cataNEList h · anaNEList g

```

Gene do anamorfismo:

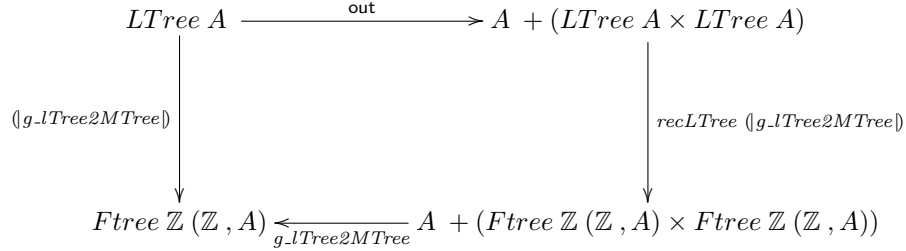
Para descobrir o gene do anamorfismo, recorreremos ao diagrama do anamorfismo das LTree.



Assim sendo percebemos que o gene recebe uma lista não vazia e produz duas listas não vazias, ou no caso da lista singular devolve-a. Uma vez que estamos a trabalhar com listas não vazias resolvemos aplicar o *outNEList* que definimos anteriormente. Como este produz em vez de duas listas um elemento e uma lista, decidimos tornar o único numa lista singular. Assim o resultado de *list2LTree* será uma árvore com o primeiro de cada sublista à esquerda

$$g_list2LTree = (id + singl \times id) \cdot outNEList$$

Gene do catamorfismo:



Da análise do diagrama retiramos que, este gene, no caso de receber um único elemento, produz um Unit, ou seja, uma folha, com um par em que o segundo elemento é a transação e o primeiro o seu valor de *hash*, que calculamos com a auxiliar fornecida *hash*. No caso de receber duas *FTree* necessitamos as juntar e concatenar os hashes com outra auxiliar fornecida, a *concHash*.

Após estudar o conteúdo do ficheiro *FTree.hs* e o enunciado concluímos que, em *FTree* a *c*, o *a* é concatenação dos *hash* e o *c* o par (hash, transação).

Para encontrar o valor de hash de uma *FTree* precisamos de definir as seguintes auxiliares:

$$\begin{aligned}
 getHash\ (Unit\ c) &= \pi_1\ c \\
 getHash\ (Comp\ a\ t) &= a \\
 calcHash\ (t1, t2) &= concHash\ (getHash\ t1, getHash\ t2)
 \end{aligned}$$

Assim para obter o resultado pretendido usando o *in* das *FTree* que recebe o tipo:

$$(\mathbb{Z} \times A) + (\mathbb{Z} \times (Ftree\ \mathbb{Z}\ (\mathbb{Z}, A) \times Ftree\ \mathbb{Z}\ (\mathbb{Z}, A)))$$

$$\begin{aligned}
 g_lTree2MTree &:: Hashable\ c \Rightarrow c + (Ftree\ \mathbb{Z}\ (\mathbb{Z}, c), Ftree\ \mathbb{Z}\ (\mathbb{Z}, c)) \rightarrow Ftree\ \mathbb{Z}\ (\mathbb{Z}, c) \\
 g_lTree2MTree &= in \cdot (\langle Main.hash, id \rangle + \langle calcHash, id \rangle)
 \end{aligned}$$

Gene de *mroot* ("get Merkle root"):

Uma vez que *computeMerkleTree* produz uma *MerkleTree* temos que o diagrama de *mroot* é:

$$\begin{array}{ccc}
 FTree \mathbb{Z} (\mathbb{Z}, A) & \xrightarrow{\text{out}} & (\mathbb{Z} \times A) + (\mathbb{Z} \times (FTree \mathbb{Z} (\mathbb{Z}, A) \times FTree \mathbb{Z} (\mathbb{Z}, A))) \\
 \downarrow \llbracket g_mroot \rrbracket & & \downarrow \text{recFTree } \llbracket g_mroot \rrbracket \\
 \mathbb{Z} & \xleftarrow{g_mroot} & (\mathbb{Z} \times A) + \mathbb{Z} \times (\mathbb{Z} \times \mathbb{Z})
 \end{array}$$

Sendo a *Merkle Root* o valor de *hash* de todo o bloco, no caso de ser uma árvore singular o resultado será a hash da transação, caso contrário a hash do nodo mais exterior, ou seja a raiz. Como podemos verificar pelo diagrama, o gene será um either dos primeiros elementos, pelo que podemos usar a auxiliar fornecida *firts*.

$$g_mroot = \text{firts}$$

Quando executamos *mroot trs* o valor obtido é -13593070566482620546, correspondente ao valor de hash da *Merkle Root*, ou seja o valor de hash mais exterior ao executar *computeMerkleTree trs*. Ao alterar uma transação, este valor vai alterado, conforme a diferença entre os valores de hash que a mudança na transação causou.

Valorização:

Para descobrir o gene do anamorfismo *pairList* recorreremos ao seu diagrama.

$$\begin{array}{ccc}
 A^* & \xrightarrow{g_list2LTree} & 1 + ((A \times A) \times A^*) \\
 \downarrow \llbracket g_pairsList \rrbracket & & \downarrow \text{recList } \llbracket g_pairsList \rrbracket \\
 (A, A)^* & \xrightleftharpoons[\text{inList}]{\text{outList}} & 1 + ((A \times A) \times (A, A)^*)
 \end{array}$$

Pela análise do diagrama, concluímos que o gene *g_pairsList* recebe uma lista e produz no, ou uma lista vazia ou um par e uma lista. Tendo em conta que a lista tem de ser par, o primeiro passo foi utilizar a auxiliar *getEvenBlock*. Em seguida utilizamos uma vez o *outList*, olhando para o segundo elemento do either produzido por este, para obter o par precisavamos de mais um elemento da lista pelo que utilizamos um *split*, ficando com o tipo: $1 + (A \times (A \times A^*))$. Com este tipo simplesmente nos basta um passo para produzir o que pretendemos, sendo que usamos a *assocl* estudada nas aulas.

$$\begin{aligned}
 \text{pairsList} &:: [a] \rightarrow [(a, a)] \\
 \text{pairsList} &= \llbracket g_pairsList \rrbracket
 \end{aligned}$$

$$g_pairsList = (id + \text{assocl}) \cdot (id + id \times \langle \text{head}, \text{tail} \rangle) \cdot \text{outList} \cdot \text{getEvenBlock}$$

Mais uma vez recorremos a um diagrama para compreender os genes pedidos e os seus tipos. Para chegar à sua versão final tivemos de compreender o *conquer*, o que é um *either*, em que o lado esquerdo é o *head* de uma lista, que tendo em conta o que este produz, será uma lista com uma *MerkleTree* singular, isto é a *Merkle Root*. Conhecendo o tipo da *joinMerkleTree* foi possível obter o tipo do lado direito do *either*, e assim obter o resto do diagrama.

$$\begin{array}{ccc}
 \mathbb{Z}^+ & \xrightarrow{\text{divide}} & (FTree\mathbb{Z}\mathbb{Z})^+ + (FTree\mathbb{Z}\mathbb{Z})^+ \times \mathbb{Z}^+ \\
 \downarrow \llbracket \text{conquer} \rrbracket \cdot \llbracket \text{divide} \rrbracket & & \downarrow \text{recNEList } (\llbracket \text{conquer} \rrbracket \cdot \llbracket \text{divide} \rrbracket) \\
 FTree\mathbb{Z}\mathbb{Z} & \xleftarrow{\text{conquer}} & (FTree\mathbb{Z}\mathbb{Z})^+ + (FTree\mathbb{Z}\mathbb{Z})^+ \times (FTree\mathbb{Z}\mathbb{Z})
 \end{array}$$

Distinto da primeira parte deste problema, em *FTree* a *c* corresponde ao valor de *ache* de uma folha, da *Unit*, e o *a* corresponde à concatenação dos filhos.

Para resolver estes genes tivemos de perceber em que fase do algoritmo estes se encontram. Sendo que já temos a uma lista de pares a ser produzida, com a *pairsList*, resta-nos os passos 4 e 5 do algoritmo. Pelo diagrama anterior, concluímos que o *divide* será o passo 4, uma vez que este apresenta uma lista de sub-árvores como mencionado.

Para o *divide* primeiro aplicamos o *outNEList*, para ficar com um tipo idêntico ao pretendido, precisando agora apenas de transpor o elemento singular em *FTree* e

```

classicMerkleTree :: Hashable a => [a] -> FTree Z Z
classicMerkleTree = (hyloNEList conquer divide) · (map Main.hash)
divide = ((singl · Unit) + (Comp)) · outNEList
conquer = [head, joinMerkleTree] where
  joinMerkleTree (l, m) = mergeMerkleTree m (evenMerkleTreeList l)
  mergeMerkleTree = \[h1, h2] ->
    h1 c l = c
    h2 (c, (f, g)) l = (c, ((head l), last l))
    evenMerkleTreeList = getEvenBlock

```

Problema 2

```

wc_w_final :: [Char] -> Int
wc_w_final = wrapper · worker
worker = \[g1, g2] ->
  wrapper = ⊥

```

Gene de *worker*:

```

g1 = nil
g2 = ⊥

```

Genes $h = [h_1, h_2]$ e $k = [k_1, k_2]$ identificados no cálculo:

```

h1 = 0
h2 (c, l) = if (sp c) ∧ (π2 l) then (π1 l + 1) else (π1 l)
k1 = True
k2 = sp · π1

```

Problema 3

```

inX :: u + (i, (X u i, X u i)) -> X u i
inX = [XLeaf, n] where

```

$$\begin{aligned}
n(i, (l, r)) &= \text{Node } i \ l \ r \\
\text{outX } (\text{XLeaf } u) &= i_1 \ u \\
\text{outX } (\text{Node } i \ l \ r) &= i_2 \ (i, (l, r)) \\
\text{baseX } f \ h \ g &= f + (h \times (g \times g)) \\
\text{recX } f &= \text{baseX } \text{id} \ \text{id} \ f \\
\text{cataX } g &= g \cdot (\text{recX } (\text{cataX } g)) \cdot \text{outX}
\end{aligned}$$

Inserir a partir daqui o resto da resolução deste problema:

....

Problema 4

Sendo o **Pairl** um anamorfismo de listas podemos

$$\begin{aligned}
\text{pairL} &:: [a] \rightarrow [(a, a)] \\
\text{pairL} &= \llbracket g \rrbracket \text{ where} \\
g &= (\text{id} + \text{assocl}) \cdot (\text{id} + \text{id} \times \langle \text{head}, \text{id} \rangle) \cdot \text{outNEList}
\end{aligned}$$

Para conseguir chegar ao resultado de f_2 precisamos primeiro de entender O que um mapa representa e como modifica-lo. A primeira conclusão a que chegamos, pelo exemplo apresentado no problema 4 do Capítulo 1 da *markMap* é que a posição (0,0) a posição (0,1) a acima, ea posição (1,0) a da direita é a ultima linha da esquerda do mapa. Ou seja uma posição é do tipo (linha,coluna), De seguida, pela análise dos mapas fornecidos, percebemos que na matriz o primeiro elemento diz respeito à linha inferior do mapa, a segunda a linha a cima e assim adiante. Chegando a estas conclusões podemos afirmar que, após obter o resultado ao executar a *toCell* entre duas posições é:

- **Lft** iremos passar a atualizar a posição seguinte no mesmo elemnto da matriz
- **Rght** iremos passar a atualizar a posição anterior no mesmo elemnto da matriz
- **UP** iremos passar a mesma a posição no elemento seguinte da matriz
- **Down** iremos passar a mesma a posição no elemento anterior da matriz

$$\begin{aligned}
\text{markMap} &:: [\text{Pos}] \rightarrow \text{Map} \rightarrow \text{Map} \\
\text{markMap } l &= \llbracket [\text{id}, f_2] \rrbracket (\text{pairL } l) \text{ where} \\
f_2 &= \perp
\end{aligned}$$

$$\begin{aligned}
\text{scout} &:: \text{Map} \rightarrow \text{Pos} \rightarrow \text{Pos} \rightarrow \text{Int} \rightarrow [[\text{Pos}]] \\
\text{scout } m \ s \ t &= \llbracket [f_1, (\gg f_2 \ m \ s)] \rrbracket \text{ where} \\
f_1 &= \perp \\
f_2 &= \perp
\end{aligned}$$

Valorização (opcional) Completar as seguintes funções de teste no **QuickCheck** para verificação de propriedades das funções pedidas, a saber:

Propriedade [QuickCheck] 1 A lista correspondente ao lado esquerdo dos pares em $(\text{pairL } l)$ é a lista original l a menos do último elemento. Analogamente, a lista correspondente ao lado direito dos pares em $(\text{pairL } l)$ é a lista original l a menos do primeiro elemento:

$$\text{prop_reconst } l = (\text{map } \pi_1 (\text{pairL } l)) \equiv (\text{init } l) \wedge (\text{map } \pi_2 (\text{pairL } l)) \equiv (\text{tail } l)$$

Propriedade [QuickCheck] 2 Assuma que uma linha (de um mapa) é prefixa de uma outra linha. Então a representação da primeira linha também prefixa a representação da segunda linha:

$$\text{prop_prefix2 } l \ l' = (\text{tail } l') \equiv (\text{tail } l)$$

Propriedade [QuickCheck] 3 Para qualquer linha (de um mapa), a sua representação deve conter um número de símbolos correspondentes a um tipo célula igual ao número de vezes que esse tipo de célula aparece na linha em questão.

$$\begin{aligned} \text{prop_nmbrs } l \ c &= \perp \\ \text{count} &:: (Eq \ a) \Rightarrow a \rightarrow [a] \rightarrow Int \\ \text{count} &= \text{uncount} \text{ where} \\ \text{uncount } (a, []) &= 0 \\ \text{uncount } (a, l) &= \text{if } (a \equiv \text{head } l) \text{ then } \text{uncount } (a, \text{tail } l) + 1 \text{ else } \text{uncount } (a, \text{tail } l) \end{aligned}$$

Propriedade [QuickCheck] 4 Para qualquer lista l a função $\text{markMap } l$ é idempotente.

$$\begin{aligned} \text{inBounds } m \ (x, y) &= (\text{nlines } m) \equiv x \wedge (\text{ncols } m) \equiv y \\ \text{prop_idemp2 } l \ m &= \perp \end{aligned}$$

Propriedade [QuickCheck] 5 Todas as posições presentes na lista dada como argumento irão fazer com que as células correspondentes no mapa deixem de ser *Free*.

$$\text{prop_extr2 } l \ m = \perp$$

Propriedade [QuickCheck] 6 Quanto maior for o tamanho máximo dos caminhos mais caminhos que alcançam a posição alvo iremos encontrar:

$$\text{prop_reach } m \ t \ n \ n' = \perp$$

Índice

L^AT_EX, [8](#)

bibtex, [8](#)

lhs2TeX, [8](#)

makeindex, [8](#)

Blockchain, [1–3](#)

Combinador “pointfree”

ana

 Listas, [3, 14, 15](#)

cata, [4](#)

 Listas, [4, 12, 15](#)

 Naturais, [9, 12, 13, 15](#)

either, [2, 4, 10, 12–15](#)

Cálculo de Programas, [1, 8](#)

 Material Pedagógico, [8](#)

 FTree.hs, [1–3, 14](#)

 LTree.hs, [1, 2, 5](#)

Functor, [4, 10, 12, 13](#)

Função

π_1 , [6, 9, 10, 12](#)

π_2 , [6, 9](#)

length, [10, 12, 13](#)

map, [3, 6, 12–14](#)

uncurry, [12](#)

Haskell, [1, 5, 8, 9](#)

 interpretador

 GHCi, [8, 9](#)

 Literate Haskell, [8](#)

 QuickCheck, [8, 12, 16](#)

 Stack, [9](#)

Merkle tree, [1–3](#)

Mónade

 Listas, [5](#)

Números naturais (\mathbb{N}), [9](#)

Programação

 literária, [8](#)

U.Minho

 Departamento de Informática, [1](#)

Unix shell

wc, [3](#)

Referências

- [1] B.W. Kernighan and D.M. Ritchie. *The C Programming Language*. Prentice Hall, Englewood Cliffs, N.J., 1978.
- [2] D.E. Knuth. *Literate Programming*. CSLI Lecture Notes Number 27. Stanford University Center for the Study of Language and Information, Stanford, CA, USA, 1992.
- [3] J.N. Oliveira. *Program Design by Calculation*, 2018. Draft of textbook in preparation. viii+297 pages. Informatics Department, University of Minho.
- [4] SelfKey. What is a Merkle tree and how does it affect blockchain technology?, 2015. Blog: <https://selfkey.org/what-is-a-merkle-tree-and-how-does-it-affect-blockchain-techno>
Last read: 26 de Janeiro de 2022.