Cálculo de Programas
Trabalho Prático
LEI+MiEI — 2021/22

Departamento de Informática
Universidade do Minho

Fevereiro de 2022

Grupo nr.	17
a89584	Miguel Ângelo Sousa Martins
a22222	Nome2 (preencher)
a33333	Nome3 (preencher)
a44444	Nome4 (preencher, se aplicável, ou apagar)

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 abodarem os problemas propostos no trabalho, os grupos devem ler com atenção o anexo A onde encontrarão as instruções relativas ao sofware 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 *LTree*s. Começa-se por, a partir da lista de transações, construir uma *LTree* cujas folhas são as transações:

$$list2LTree :: [a] \rightarrow \mathit{LTree}\ a$$

- *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 *LTree*s

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

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

   $$list2LTree :: [\,a\,] \rightarrow LTree\ a$$
   $$list2LTree = [\!(g\_list2LTree)\!]$$

   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]

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

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

   $$lTree2MTree :: Hashable\ a \Rightarrow LTree\ a \rightarrow FTree\ \mathbb{Z}\ (\mathbb{Z}, a)$$
   $$lTree2MTree = (\!| g\_lTree2MTree |\!)$$

3. Defina g_mroot por forma a

   $$mroot :: Hashable\ b \Rightarrow [\,b\,] \rightarrow \mathbb{Z}$$
   $$mroot = (\!| g\_mroot |\!) \cdot computeMerkleTree$$

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

[1]Para 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

$$classicMerkleTree :: Hashable\ a \Rightarrow [a] \rightarrow \mathit{FTree}\ \mathbb{Z}\ \mathbb{Z}$$

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

$$hyloNEList\ h\ g = cataNEList\ h \cdot anaNEList\ g$$

etc. Depois passe à definição do gene $g\_pairsList$ do anamorfismo de listas

$$pairsList :: [a] \rightarrow [(a, a)]$$
$$pairsList = [\!(g\_pairsList)\!]$$

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

$$getEvenBlock :: [a] \rightarrow [a]$$

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 catamorfimo por forma a

$$classicMerkleTree = (hyloNEList\ conquer\ divide) \cdot (\mathsf{map}\ \ Main.hash)$$

Para facilitar a definição do $conquer$, terá apenas de definir o gene $g\_mergeMerkleTree$ do catamorfismo de ordem superior

$$mergeMerkleTree :: \mathit{FTree}\ a\ p \rightarrow [\mathit{FTree}\ a\ c] \rightarrow \mathit{FTree}\ a\ c$$
$$mergeMerkleTree = (\!|\ g\_mergeMerkleTree\ |\!)$$

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 stan-
    dard  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:

$$
\begin{array}{l}
wc\_w :: [\,Char\,] \rightarrow Int \\
wc\_w\ [\,] = 0 \\
wc\_w\ (c : l) = \\
\quad \mathbf{if}\ \neg\ (sep\ c) \wedge lookahead\_sep\ l\ \mathbf{then}\ wc\_w\ l + 1\ \mathbf{else}\ wc\_w\ l \\
\qquad \mathbf{where} \\
\qquad\quad sep\ c = (c \equiv \texttt{'\ '} \vee c \equiv \texttt{'\textbackslash n'} \vee c \equiv \texttt{'\textbackslash t'}) \\
\qquad\quad lookahead\_sep\ [\,] = True \\
\qquad\quad lookahead\_sep\ (c : l) = sep\ c
\end{array}
$$

Por aplicação da lei de recursividade mútua

$$
\left\{ \begin{array}{l} f \cdot \mathsf{in} = h \cdot \mathsf{F}\ \langle f, g \rangle \\ g \cdot \mathsf{in} = k \cdot \mathsf{F}\ \langle f, g \rangle \end{array} \right. \quad \equiv \quad \langle f, g \rangle = (\!|\langle h, k \rangle|\!) \tag{2}
$$

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

$$
\begin{array}{l}
wc\_w\_final :: [\,Char\,] \rightarrow Int \\
wc\_w\_final = wrapper \cdot \underbrace{(\!|[g1\,, g2]|\!)}_{worker}
\end{array}
$$

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:

$$\mathbf{data}\ Unit\ a\ b = Image\ a \mid Text\ b\ \mathbf{deriving}\ Show$$

O tipo $Sheet$ (="página de jornal")

$$\mathbf{data}\ Sheet\ a\ b\ i = Rect\ (Frame\ i)\ (X\ (Unit\ a\ b)\ (Mode\ i))\ \mathbf{deriving}\ Show$$

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

$$\mathbf{data}\ Mode\ i = Hr\ i \mid Hl\ i \mid Vt\ i \mid Vb\ i\ \mathbf{deriving}\ Show$$

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

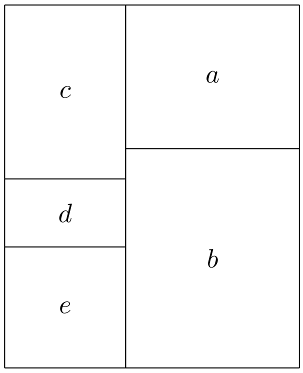
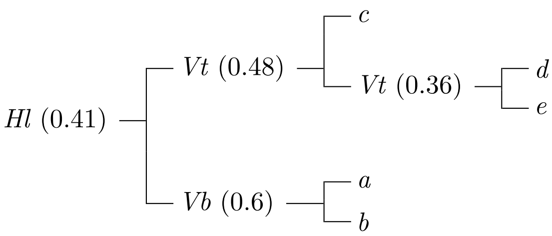


Figura 2: Layout de página de jornal.

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:

   $pairL\ [1, 2] \equiv [(1, 2)]$,
   $pairL\ [1, 2, 3] \equiv [(1, 2), (2, 3)]$ e
   $pairL\ [1, 2, 3, 4] \equiv [(1, 2), (2, 3), (3, 4)]$

   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:

[2] A desenvolver de forma análoga a outras bibliotecas que conhece (*eg.* *LTree*, etc).
[3] Repare 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:

$$\textbf{data}\ Cell = Free \mid Blocked \mid Lft \mid Rght \mid Up \mid Down\ \textbf{deriving}\ (Eq, Show)$$
$$\textbf{type}\ Map = [[Cell]]$$

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, $putStr\ \$\ showM\ \$\ map_1$ 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 _ _ _ _ _ _ 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 $markMap :: [Pos] \rightarrow Map \rightarrow Map$, 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,

$$putStr\ \$\ showM\ \$\ markMap\ [(0,0),(0,1),(0,2),(1,2)]\ map_1$$

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 $[(Pos, Pos)] \rightarrow Map \rightarrow Map$ 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.*

[4] Ao 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 impossivel sair dela então todo o caminho que levou a esta posição deve ser removido (*backtracking*). Por exemplo:

$scout\ map_1\ (0,0)\ (2,0)\ 0 \equiv [[(0,0)]]$
$scout\ map_1\ (0,0)\ (2,0)\ 1 \equiv [[(0,0),(0,1)]]$
$scout\ map_1\ (0,0)\ (2,0)\ 4 \equiv [[(0,0),(0,1),(0,2),(1,2),(2,2)]]$
$scout\ map_2\ (0,0)\ (2,2)\ 2 \equiv [[(0,0),(0,1),(1,1)],[(0,0),(0,1),(0,2)]]$
$scout\ map_2\ (0,0)\ (2,2)\ 4 \equiv [[(0,0),(0,1),(1,1),(2,1),(2,2)],[(0,0),(0,1),(1,1),(2,1),(2,0)]]$

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 catamorfismos 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 LaTeX 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 BibTeX) 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:

[5] O 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 depêndencias 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]

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

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

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

[6]Como 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.
[7]Exemplos tirados de [3].

B Código fornecido

Problema 1

Sequência de transações para teste:

$trs = [(\texttt{"compra"}, \texttt{"20211102"}, -50),$
$\quad(\texttt{"venda"}, \quad \texttt{"20211103"}, 100),$
$\quad(\texttt{"despesa"}, \texttt{"20212103"}, -20),$
$\quad(\texttt{"venda"}, \quad \texttt{"20211205"}, 250),$
$\quad(\texttt{"venda"}, \quad \texttt{"20211205"}, 120)]$

$getEvenBlock :: [\,a\,] \rightarrow [\,a\,]$
$getEvenBlock\ l = \mathbf{if}\ (even\ (length\ l))\ \mathbf{then}\ l\ \mathbf{else}\ l +\!\!+ [\,last\ l\,]$

$firsts = [\pi_1, \pi_1]$

Problema 2

$wc\_test = \texttt{"Here is a sentence, for testing.\textbackslash nA short one."}$
$sp\ c = (c \equiv \texttt{' '} \vee c \equiv \texttt{'\textbackslash n'} \vee c \equiv \texttt{'\textbackslash t'})$

Problema 3

Tipos:

$\mathbf{data}\ X\ u\ i = XLeaf\ u \mid Node\ i\ (X\ u\ i)\ (X\ u\ i)\ \mathbf{deriving}\ Show$
$\mathbf{data}\ Frame\ i = Frame\ i\ i\ \mathbf{deriving}\ Show$

Funções da API[8]

$printJournal :: Sheet\ String\ String\ Double \rightarrow \mathsf{IO}\ ()$
$printJournal = write \cdot sheet2html$

$write :: String \rightarrow \mathsf{IO}\ ()$
$write\ s = \mathbf{do}\ writeFile\ \texttt{"jornal.html"}\ s$
$\quad putStrLn\ \texttt{"Output HTML written into file `jornal.html'"}$

Geração de HTML:

$sheet2html\ (Rect\ (Frame\ w\ h)\ y) = htmlwrap\ (x2html\ y\ (w, h))$

$x2html :: X\ (Unit\ String\ String)\ (Mode\ Double) \rightarrow (Double, Double) \rightarrow String$
$x2html\ (XLeaf\ (Image\ i))\ (w, h) = img\ w\ h\ i$
$x2html\ (XLeaf\ (Text\ txt))\ \_ = txt$
$x2html\ (Node\ (Vt\ i)\ x1\ x2)\ (w, h) = htab\ w\ h\ ($
$\quad tr\ (td\ w\ (h * i)\ (x2html\ x1\ (w, h * i))) +\!\!+$
$\quad tr\ (td\ w\ (h * (1 - i))\ (x2html\ x2\ (w, h * (1 - i))))$
$\quad)$
$x2html\ (Node\ (Hl\ i)\ x1\ x2)\ (w, h) = htab\ w\ h\ ($
$\quad tr\ (td\ (w * i)\ h\ (x2html\ x1\ (w * i, h)) +\!\!+$
$\qquad td\ (w * (1 - i))\ h\ (x2html\ x2\ (w * (1 - i), h)))$
$\quad)$
$x2html\ (Node\ (Vb\ i)\ x1\ x2)\ m = x2html\ (Node\ (Vt\ (1 - i))\ x1\ x2)\ m$
$x2html\ (Node\ (Hr\ i)\ x1\ x2)\ m = x2html\ (Node\ (Hl\ (1 - i))\ x1\ x2)\ m$

Funções auxiliares:

[8] API (="Application Program Interface").

```haskell
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:

```haskell
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:

```haskell
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:

$$map_1 = [[Free, Blocked, Free], [Free, Blocked, Free], [Free, Free, Free]]$$
$$map_2 = [[Free, Blocked, Free], [Free, Free, Free], [Free, Blocked, Free]]$$
$$map_3 = [[Free, Free, Free], [Free, Blocked, Free], [Free, Blocked, Free]]$$

Código para impressões de mapas e caminhos:

$showM :: Map \to String$
$showM = unlines \cdot (\mathsf{map}\ \ showL) \cdot reverse$

$showL :: [Cell] \to String$
$showL = (\![f_1, f_2]\!)$ **where**
$\quad f_1 = \underline{\texttt{""}}$
$\quad f_2 = \widehat{(+\!\!+)} \cdot (fromCell \times id)$

$fromCell\ Lft = \texttt{" > "}$
$fromCell\ Rght = \texttt{" < "}$
$fromCell\ Up = \texttt{" ^ "}$
$fromCell\ Down = \texttt{" v "}$
$fromCell\ Free = \texttt{" \_ "}$
$fromCell\ Blocked = \texttt{" X "}$

$toCell\ (x, y)\ (w, z) \mid x < w = Lft$
$toCell\ (x, y)\ (w, z) \mid x > w = Rght$
$toCell\ (x, y)\ (w, z) \mid y < z = Up$
$toCell\ (x, y)\ (w, z) \mid y > z = Down$

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

$ncols :: Map \to Int$
$ncols = [\underline{0}, length \cdot \pi_1] \cdot outList$

$nlines :: Map \to Int$
$nlines = length$

$isValidMap :: Map \to Bool$
$isValidMap = \widehat{(\wedge)} \cdot \langle isSquare, sameLength\rangle$ **where**
$\quad isSquare = \widehat{(\equiv)} \cdot \langle nlines, ncols\rangle$
$\quad sameLength\ [\,] = True$
$\quad sameLength\ [x] = True$
$\quad sameLength\ (x1 : x2 : y) = length\ x1 \equiv length\ x2 \wedge sameLength\ (x2 : y)$

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

$randomRIOL :: (Random\ a) \Rightarrow (a, a) \to Int \to \mathsf{IO}\ [a]$
$randomRIOL\ x = (\![f_1, f_2]\!)$ **where**
$\quad f_1 = \underline{return\ [\,]}$
$\quad f_2\ l = \mathbf{do}\ r1 \leftarrow randomRIO\ x$
$\qquad r2 \leftarrow l$
$\qquad return\ \$\ r1 : r2$

$buildMat :: Int \to Int \to \mathsf{IO}\ [[Int]]$
$buildMat\ n = (\![f_1, f_2]\!)$ **where**
$\quad f_1 = \underline{return\ [\,]}$
$\quad f_2\ l = \mathbf{do}\ x \leftarrow randomRIOL\ (0 :: Int, 3 :: Int)\ n$
$\qquad y \leftarrow l$
$\qquad return\ \$\ x : y$

$testWithRndMap :: \mathsf{IO}\ ()$
$testWithRndMap = \mathbf{do}$
$\quad dim \leftarrow randomRIO\ (2, 10) :: \mathsf{IO}\ Int$
$\quad out \leftarrow buildMat\ dim\ dim$
$\quad \mathsf{map}\ \leftarrow return\ \$\ \mathsf{map}\ \ (\mathsf{map}\ \ table)\ out$
$\quad putStr\ \$\ showM\ \mathsf{map}$
$\quad putStrLn\ \$\ \texttt{"Map of dimension "} +\!\!+ (show\ dim) +\!\!+ \texttt{"x"} +\!\!+ (show\ dim) +\!\!+ \texttt{"."}$

```
    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 $subst :: a \to Int \to [a] \to [a]$, que dado um valor x e um inteiro n, produz uma função $f : [a] \to [a]$ 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
```

$checkAround :: Map \to Pos \to [Pos]$, 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
```

$\textbf{infixr}\ 4 \leqslant$
$(\leqslant) :: \mathit{Ord}\ b \Rightarrow (a \rightarrow b) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow \mathit{Bool})$
$f \leqslant g = \lambda a \rightarrow f\ a \leqslant g\ a$
$\textbf{infixr}\ 4 \wedge$
$(\wedge) :: (a \rightarrow \mathit{Bool}) \rightarrow (a \rightarrow \mathit{Bool}) \rightarrow (a \rightarrow \mathit{Bool})$
$f \wedge g = \lambda a \rightarrow ((f\ a) \wedge (g\ a))$
$\textbf{instance}\ \mathit{Arbitrary}\ \mathit{Cell}\ \textbf{where}$
-- 1/4 chance of generating a cell 'Block'.
$\mathit{arbitrary} = \textbf{do}\ x \leftarrow \mathit{chooseInt}\ (0, 3)$
$\mathit{return}\ \$\ f\ x\ \textbf{where}$
$f\ x = \textbf{if}\ x < 3\ \textbf{then}\ \mathit{Free}\ \textbf{else}\ \mathit{Blocked}$

C Soluções dos alunos

Os alunos devem colocar neste anexo as suas soluções para os 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, diagramas e/ou outras funções auxiliares que sejam necessárias.

Valoriza-se a escrita de *pouco* código que corresponda a soluções simples e elegantes.

Problema 1

Listas vazias:

$$
\begin{array}{ll}
 & \mathit{outNEList} \cdot \mathit{inNEList} = \mathit{id} \\
\equiv & \quad \{\ \mathit{inNEList} := [\mathit{singl}, \mathit{cons}]\ \} \\
 & \mathit{outNEList} \cdot [\mathit{singl}, \mathit{cons}] = \mathit{id} \\
\equiv & \quad \{\ \text{Fusão-+, lei(20)}\ \} \\
 & [\mathit{outNEList} \cdot \mathit{singl}, \mathit{outNEList} \cdot \mathit{cons}] = \mathit{id} \\
\equiv & \quad \{\ \text{Universal-+, lei(17)}\ \} \\
 & \left\{ \begin{array}{l} \mathit{id} \cdot i_1 = \mathit{outNEList} \cdot \mathit{singl} \\ \mathit{id} \cdot i_2 = \mathit{outNEList} \cdot \mathit{cons} \end{array} \right. \\
\equiv & \quad \{\ \text{Natural-id, lei(1) ; Igualdade Extensional, lei(71)}\ \} \\
 & \left\{ \begin{array}{l} i_1\ x = (\mathit{outNEList} \cdot \mathit{singl})\ x \\ i_2\ (x, y) = (\mathit{outNEList} \cdot \mathit{cons})\ (x, y) \end{array} \right. \\
\equiv & \quad \{\ \text{Def-comp, lei(72)}\ \} \\
 & \left\{ \begin{array}{l} i_1\ x = \mathit{outNEList}\ (\mathit{singl}\ x) \\ i_2\ (x, y) = \mathit{outNEList}\ (\mathit{cons}\ (x, y)) \end{array} \right. \\
\equiv & \quad \{\ \text{singl x = [x] ; cons (x,y) = x:y}\ \} \\
 & \left\{ \begin{array}{l} i_1\ x = \mathit{outNEList}\ [x] \\ i_2\ (x, y) = \mathit{outNEList}\ x : y \end{array} \right. \\
\equiv & \quad \{\ \text{Propriedade simétrica da igualdade}\ \} \\
 & \left\{ \begin{array}{l} \mathit{outNEList}\ [x] = i_1\ x \\ \mathit{outNEList}\ x : y = i_2\ (x, y) \end{array} \right. \\
\square &
\end{array}
$$

$\mathit{outNEList}\ [a] = i_1\ a$
$\mathit{outNEList}\ (h : t) = i_2\ (h, t)$

$$
\begin{aligned}
&baseNEList\ f\ g = id + (f \times g)\\
&recNEList\ \ f\ = id + (id \times f)\\
&cataNEList\ g = g \cdot recNEList\ (cataNEList\ g) \cdot outNEList\\
&anaNEList\ g\ = inNEList \cdot recNEList\ (anaNEList\ g) \cdot g\\
&hyloNEList\ h\ g = cataNEList\ h \cdot anaNEList\ g
\end{aligned}
$$

Gene do anamorfismo:

$$
\begin{tikzcd}
LTree\ A & & A + (LTree\ A)^2 \arrow[ll, "\mathsf{in}={[Leaf, Fork]}"'] \\
A^* \arrow[u, "list2LTree"] \arrow[r, "outNEList"'] & A + A \times A^* \arrow[r, "g\_list2LTree"'] & A + A^* \times A^* \arrow[u, "id+list2LTree^2"']
\end{tikzcd}
$$

Nós sabemos que $g\_list2LTree = id + \langle f, g \rangle$. Então sobra determinar f e g. Para tal vamos analisar cada função individualmente.

A função f vai processar o *tuple* $A \times A^*$ de maneira a devolver o seu primeiro elemento à cabeça da primeira metade da lista presente na segunda parte do mesmo. Assim determina-mos que o fluxo lógico de ações para a função f seria primeiro determinar qual a metade da lista no *tuple* que lhe é passado. Através da função $half$ que gera um *tuple* cujo primeiro elemento é a primeira metade da lista e o segundo o resto conseguimos obter esse resultado. Como só queremos a primeira pare do *tuple* segue de forma composta a função π_1 para isolar esse elemento obtendo $\pi_1 \cdot half$. Como queremos preservar o primeiro elemento para o adicionar à cabeça da lista não lhe fazemos nada aplicando-lhe a função id obtendo $id \times (\pi_1 \cdot half)$. Para terminar como f retorna uma lista temos que a contruir utilizando o *tuple* e para tal fazemos uso do contrutor $cons$ obtendo $cons \cdot (id \times (\pi_1 \cdot half))$.

A função g age de forma muito semelhante à função f mas aquando da escolha da lista que deve retornar este deve escolher a segunda metade da lista. Assim acabamos com $\pi_2 \cdot half \cdot \pi_2$.

A primeira aproximação ao exercício foi a descrita acima. Mas uma dúvida surgiu entretanto. As folhas duma $MerkleTree$ devem estar todas ao mesmo nível? Depois de uma breve pesquisa concluímos que sim e portanto a *LTree* gerada aqui deveria ter as suas folhas todas ao mesmo nível. Acreditamos que este problema é mais facilmente resolvido aqui do que na geração da $MerkleTree$ mais tarde.

Este "novo"problema apesar de não ser muito diferente do anterior implica uma abordagem um pouco diferente. Primeiro como determinar as restantes folhas. Segundo o algoritmo de geração de $MerkleTree$'s apresentado no enunciado esta deve ser feita replicando o último elemento da lista a transformar quando necessário. E quando é que é necessário? Aqui o ideal é que a cada iteração, desde que a lista contenha mais do que um elemento que no final da divisão da lista que elas tenham o mesmo número de elementos e que esse número seja par. Porquê? Bem porque depois de alguns testes concluimos que a divisão em $Haskell$ é imprivisível e que a maneira mais segura de garantir um resultado satisfatório seria com estes passos.

Primeiro através de um *split* à entrada reconstruir a lista gerando um *tuple* de listas. De seguida dividir cada uma dessas listas a meio através da função auxliar $half$. A este ponto temos um *tuple* com dois *tuples* de listas como elementos em que a primeira lista é a primeira metade da lista total e a segunda bem, a segunda. Algo do género $((A^* \times A^*) \times (A^* \times A^*))$. A função $auxHalf$ iguala o número de elementos das duas listas duplicando o último elemento da lista mais pequena se necessário. As projeções π_1 e π_2 vão encontrar a primeira e segunda metade respetivamente. No final deste processo temos que verificar se a lista gerada contém mais do que um elemento. Aqui entre a condição $cond\ \ auxCond\ \ getEvenBlock\ \ id$ que basicamente verifica essa condição através função $auxCond$ e na eventualidade de se verificar (i.e. A lista contém mais do que um elemento.) aplica a função $getEvenBlock$ que coloca o número de elementos de uma lista a par. Se a cada iteração garantirmos este seguimento funcional, é garantida a contrução de uma *LTree* cujas folhas se encontram todas ao mesmo nível no final e que esse nível está completo.

Desta forma, nasce o novo gene do anamorfismo de $list2LTree$:

$id + \langle cond\ auxCond\ getEvenBlock\ id \cdot \pi_1 \cdot auxHalf \cdot half \cdot cons, cond\ auxCond\ getEvenBlock\ id \cdot \pi_2 \cdot auxHalf \cdot half \cdot cons \rangle$

$$
\begin{aligned}
&g\_list2LTree = (id + \langle cond\ auxCond\ getEvenBlock\ id \cdot \pi_1 \cdot auxHalf \cdot half \cdot cons, cond\ auxCond\ getEvenBlock\ id\\
&half :: [a] \to ([a],[a])\\
&half\ list = splitAt\ (length\ list\ `div`\ 2)\ list
\end{aligned}
$$

$$
\begin{array}{l}
auxCond :: [\,a\,] \to Bool \\
auxCond\ l = length\ l > 1 \\
auxHalf :: ([\,a\,],[\,a\,]) \to ([\,a\,],[\,a\,]) \\
auxHalf\ (l_1, l_2) \mid length\ l_1 < length\ l_2 = (l_1 +\!\!+ [\,last\ l_1\,], l_2) \\
\quad \mid length\ l_2 < length\ l_1 = (l_1, l_2 +\!\!+ [\,last\ l_2\,]) \\
\quad \mid otherwise = (l_1, l_2)
\end{array}
$$

Gene do catamorfismo:

$$
\begin{tikzcd}
LTree\ A \arrow[d, "lTree2MTree"'] & & A + (LTree\ A)^2 \arrow[ll, "\mathsf{in}{=}[Leaf, Fork]"'] \arrow[d, "id+lTree2MTree^2"] \\
FTree\ Z\ (Z \times A) & & A + (FTree\ Z\ (Z \times A))^2 \arrow[ll, "g\_lTree2MTree"]
\end{tikzcd}
$$

Nós sabemos que $g\_lTree2MTree = [g1, g2]$. Então sobra determinar $g1$ e $g2$. Para tal vamos analisar cada função individualmente.

A função $g1$ vai tratar do caso em que estamos a lidar com uma folha da *LTree* do *input*. Esta é bastante fácil de determinar pois os únicos passos que temos de realizar são gerar o *tuple* (*hash*,*input*) e aplicar o construtor $Unit$ do tipo *FTree*. O primeiro passo é atingido através do $\langle hash, id \rangle$ e o segundo é só compor este com $Unit$ obtendo $Unit \cdot \langle hash, id \rangle$.

Já a função $g2$ vai tratar do caso em que temos que processar *tuple* de duas $FTree's$. Aqui, a estratégia adotada foi gerar um *tuple* com a concatenação dos *hashes* como primeiro elemento e as duas $FTree's$ como segundo e depois aplicar a função $\widehat{Comp}$ de maneira a gerar a *FTree* final.

Assim de imediato aplicamos sobre o *tuple* deentrada um *split* de maneira a poder gerar um novo *tuple* como o descrito anteriormente. Como queremos preservar o *tuple* com as $FTree's$ no segundo elemento a segunda função do *split* será naturalmente a função identidade id.

Já a primeira função do *split* tem mais que se lhe diga. O objetivo desta é, como já foi mencionado, gerar a concatenação dos *hashes* das $FTree's$ que lhe são passadas. Para tal, primeiramente temos que isolar os respetivos *hashes*. Com isso em mente procedemos por aplicar a função out a cada uma das $FTree's$ efetivamente gerando uma um estrutura do tipo $(Z \times A) + (Z \times (FTree\ Z\ (Z \times A))^2)$ para cada. Como tudo o que precisamos obter a partir desta estrutura são os Z que constituem o primeiro elemento dos respetivos *tuples* a alternativa $[\pi_1, \pi_1]$ é a função necessária para o efeito. Como esta função já se encontra disponibilizada pelo nome *firsts* fazemos então uso dela. Assim para obter os valores dos *hashes* das $FTree's$ utilizamos a composição $firsts \cdot \mathsf{out}$ ilustrado no seguinte diagrama:

$$
\begin{tikzcd}
Z & (Z \times A) + (Z \times (FTree\ Z\ (Z \times A))^2 \arrow[l, "firsts"] & FTree\ Z\ (Z \times A) \arrow[l, "\mathsf{out}"]
\end{tikzcd}
$$

Sendo possível obter o valor dos *hashes* das $FTree's$ temos que gerar a concatenação dos mesmos. Para tal, fizemos uso da função já disponibilizada *concHash* que aceita um *tuple* de Z como entrada. Este será gerado aplicanda a função já definada a cada um dos elementos do *tuple* de entrada através do uso do compositor $\times$ efetivamente gerando a função $(firsts \cdot \mathsf{out}) \times (firsts \cdot \mathsf{out})$ com o comportamento definido no seguinte diagrama:

$$
\begin{tikzcd}
Z & & Z \times Z \arrow[ll, "\pi_1"'] \arrow[rr, "\pi_2"] & & Z \\
FTree\ Z\ (Z \times A) \arrow[u, "firsts \cdot \mathsf{out}"] & & (FTree\ Z\ (Z \times A))^2 \arrow[ll, "\pi_1"] \arrow[rr, "\pi_2"'] \arrow[u, "(firsts \cdot \mathsf{out}) \times (firsts \cdot \mathsf{out})"'] & & FTree\ Z\ (Z \times A) \arrow[u, "firsts \cdot \mathsf{out}"']
\end{tikzcd}
$$

Assim concluímos que para gerar a concatenação dos *hashes* das duas $FTree's$ como primeiro elemento do *tuple* teriamos que usar a função $concHash \cdot ((firsts \cdot \mathsf{out}) \times (firsts \cdot \mathsf{out}))$.

Finalmente após este processo todo acabamos com a função $\widehat{Comp} \cdot \langle concHash \cdot ((firsts \cdot \mathsf{out}) \times (firsts \cdot \mathsf{out})), id \rangle$ para $g2$.

$$
\begin{array}{l}
g\_lTree2MTree :: Hashable\ c \Rightarrow c + (FTree\ \mathbb{Z}\ (\mathbb{Z}, c), FTree\ \mathbb{Z}\ (\mathbb{Z}, c)) \to FTree\ \mathbb{Z}\ (\mathbb{Z}, c) \\
g\_lTree2MTree = [Unit \cdot \langle Main.hash, id \rangle, \widehat{Comp} \cdot \langle concHash \cdot ((firsts \cdot \mathsf{out}) \times (firsts \cdot \mathsf{out})), id \rangle]
\end{array}
$$

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

$$g\_mroot = firsts$$

Valorização:

$$
\begin{array}{l}
pairsList :: [\,a\,] \rightarrow [\,(a, a)\,] \\
pairsList = [\![g\_pairsList]\!] \cdot getEvenBlock \\
g\_pairsList = (id + \langle auxPairList, (drop\ 1) \cdot \pi_2 \rangle) \cdot outList \\
auxPairList :: (a, [\,a\,]) \rightarrow (a, a) \\
auxPairList\ (a, [\,]) = (a, a) \\
auxPairList\ (a, h : t) = (a, h) \\
classicMerkleTree :: Hashable\ a \Rightarrow [\,a\,] \rightarrow FTree\ \mathbb{Z}\ \mathbb{Z} \\
classicMerkleTree = (hyloNEList\ conquer\ divide) \cdot (\mathsf{map}\ \ Main.hash) \\
divide = \bot \\
conquer = [head, joinMerkleTree]\ \mathbf{where} \\
\quad joinMerkleTree\ (l, m) = mergeMerkleTree\ m\ (evenMerkleTreeList\ l) \\
\quad mergeMerkleTree = (\![\, [h_1, h_2] \,]\!) \\
\quad h_1\ c\ l = \bot \\
\quad h_2\ (c, (f, g))\ l = \bot \\
\quad evenMerkleTreeList = \bot
\end{array}
$$

Problema 2

$$
\begin{array}{l}
wc\_w\_final :: [\,Char\,] \rightarrow \mathbb{Z} \\
wc\_w\_final = wrapper \cdot worker \\
worker = (\![[g1, g2]]\!) \\
wrapper = \pi_1
\end{array}
$$

Gene de $worker$:

$$
\begin{array}{l}
g1 = \langle h_1, k_1 \rangle \\
g2 = \langle h_2, k_2 \rangle
\end{array}
$$

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

$$
\left\{
\begin{array}{l}
wc\_w \cdot \mathsf{in} = h \cdot \mathsf{F}\ \langle wc\_w, lookahead\_sep \rangle \\
lookahead\_sep \cdot \mathsf{in} = k \cdot \mathsf{F}\ \langle wc\_w, lookahead\_sep \rangle
\end{array}
\right.
\qquad (3)
$$

Determinar h:

$$
\begin{array}{cl}
 & wc\_w \cdot \mathsf{in} = h \cdot \mathsf{F}\ \langle wc\_w, lookahead\_sep \rangle \\
\equiv & \quad \{\ \mathsf{in} := [nil, cons]\ ;\ h := [h_1, h_2]\ ;\ \mathsf{F}\ f = id + id \times f\ \} \\
 & wc\_w \cdot [nil, cons] = [h_1, h_2] \cdot (id + id \times \langle wc\_w, lookahead\_sep \rangle) \\
\equiv & \quad \{\ \text{Fusão-+, lei(20) ; Absorção-+, lei(22)}\ \} \\
 & [wc\_w \cdot nil, wc\_w \cdot cons] = [h_1 \cdot id, h_2 \cdot (id + id \times \langle wc\_w, lookahead\_sep \rangle)] \\
\equiv & \quad \{\ \text{Eq-+, lei(27) ; Natural-id, lei(1)}\ \} \\
 & \left\{ \begin{array}{l} wc\_w \cdot nil = h_1 \\ wc\_w \cdot cons = h_2 \cdot (id \times \langle wc\_w, lookahead\_sep \rangle) \end{array} \right. \\
\equiv & \quad \{\ \text{Igualdade Extensional, lei(71) ; Def-comp, lei(72) ; Def-const, lei(74)}\ \} \\
 & \left\{ \begin{array}{l} wc\_w\ [\,] = h_1\ x \\ wc\_w\ (cons\ x) = h_2\ ((id \times \langle wc\_w, lookahead\_sep \rangle)\ x) \end{array} \right. \\
\equiv & \quad \{\ wc\_w\ [\,] = 0\ ;\ \text{x := (x,y) ; Def-x, lei(77)}\ \}
\end{array}
$$

$$\begin{cases} 0 = h_1\ x \\ wc\_w\ (cons\ (x, y)) = h_2\ (id\ x, \langle wc\_w, lookahead\_sep \rangle\ y) \end{cases}$$

$\equiv$ { Def-id, lei(73) ; Def-split, lei(76) ; cons (x,y) = (x:y) }

$$\begin{cases} 0 = h_1\ x \\ wc\_w\ (x : y) = h_2\ (x, (wc\_w\ y, lookahead\_sep\ y)) \end{cases}$$

$\equiv$ { Propriedade simétrica da igualdade ; $wc\_w\ (x : y) = \textbf{if}\ \neg\ (sp\ x) \wedge lookahead\_sep\ y\ \textbf{then}\ wc\_w\ y + 1\ \textbf{else}\ wc$

$$\begin{cases} h_1\ x = 0 \\ h_2\ (x, (wc\_w\ y, lookahead\_sep\ y)) = \textbf{if}\ \neg\ (sp\ x) \wedge lookahead\_sep\ y\ \textbf{then}\ wc\_w\ y + 1\ \textbf{else}\ wc\_w\ y \end{cases}$$

$\equiv$ { Def-cond, lei(78) }

$$\begin{cases} h_1\ x = 0 \\ h_2\ (x, (wc\_w\ y, lookahead\_sep\ y)) = cond\ (\neg\ (sp\ x) \wedge lookahead\_sep\ y)\ (wc\_w\ y + 1)\ (wc\_w\ y) \end{cases}$$

$\equiv$ { $auxWcW\ (x, y) = x \wedge y$; $\mathsf{succ}\ \ x = x + 1$ }

$$\begin{cases} h_1\ x = 0 \\ h_2\ (x, (wc\_w\ y, lookahead\_sep\ y)) = cond\ (auxWcW\ (\neg\ (sp\ x), lookahead\_sep\ y))\ (\mathsf{succ}\ \ (wc\_w\ y))\ (wc\_ \end{cases}$$

$\equiv$ { Def-comp, lei(72) }

$$\begin{cases} h_1\ x = 0 \\ h_2\ (x, (wc\_w\ y, lookahead\_sep\ y)) = cond\ (auxWcW\ ((\neg \cdot sp)\ x, lookahead\_sep\ y))\ (\mathsf{succ}\ \ (wc\_w\ y))\ (wc \end{cases}$$

$\equiv$ { Def-proj, lei(79) }

$$\begin{cases} h_1\ x = 0 \\ h_2\ (x, (wc\_w\ y, lookahead\_sep\ y)) = cond\ (auxWcW\ ((\neg \cdot sp)\ x, \pi_2\ (wc\_w\ y, lookahead\_sep\ y)))\ (\mathsf{succ}\ \ (\end{cases}$$

$\equiv$ { Def-x, lei(77), Def-comp, lei(72) }

$$\begin{cases} h_1\ x = 0 \\ h_2\ (x, (wc\_w\ y, lookahead\_sep\ y)) = cond\ ((auxWcW \cdot ((\neg \cdot sp) \times \pi_2))\ (x, (wc\_w\ y, lookahead\_sep\ y)))\ (\end{cases}$$

$\equiv$ { Def-x, lei(79), Def-comp, lei(72) }

$$\begin{cases} h_1\ x = 0 \\ h_2\ (x, (wc\_w\ y, lookahead\_sep\ y)) = cond\ ((auxWcW \cdot ((\neg \cdot sp) \times \pi_2))\ (x, (wc\_w\ y, lookahead\_sep\ y)))\ (\end{cases}$$

$\equiv$ { Def-cond, lei(78) }

$$\begin{cases} h_1\ x = 0 \\ h_2\ (x, (wc\_w\ y, lookahead\_sep\ y)) = (cond\ (auxWcW \cdot ((\neg \cdot sp) \times \pi_2))\ (\mathsf{succ}\ \cdot \pi_1 \cdot \pi_2)\ (\pi_1 \cdot \pi_2))\ (x, (wc\_ \end{cases}$$

$\square$

A partir do cálculo efetuado em cima concluímos que h_1 é a função constante $zero$ e h_2 é a função $cond\ (auxWcW \cdot ((\neg \cdot sp) \times \pi_2))\ (\mathsf{succ}\ \cdot \pi_1 \cdot \pi_2)\ (\pi_1 \cdot \pi_2)$.

$$h_1 = zero$$
$$h_2 = cond\ (auxWcW \cdot ((\neg \cdot sp) \times \pi_2))\ (\mathsf{succ}\ \cdot \pi_1 \cdot \pi_2)\ (\pi_1 \cdot \pi_2)$$
$$auxWcW :: (Bool, Bool) \rightarrow Bool$$
$$auxWcW\ (x, y) = x \wedge y$$

Determinar k:

$$lookahead\_sep \cdot \mathsf{in} = h \cdot \mathsf{F}\ \langle wc\_w, lookahead\_sep \rangle \tag{20}$$

$\equiv$ { $\mathsf{in} := [nil, cons]$; $k := [k_1, k_2]$; $\mathsf{F}\ f = id + id \times f$ }

$$lookahead\_sep \cdot [nil, cons] = [k_1, k_2] \cdot (id + id \times \langle wc\_w, lookahead\_sep \rangle) \tag{21}$$

$\equiv$ { Fusão-+, lei(20) ; Absorção-+, lei(22) }

$$[lookahead\_sep \cdot nil, lookahead\_sep \cdot cons] = [k_1 \cdot id, k_2 \cdot (id + id \times \langle wc\_w, lookahead\_sep \rangle)] \tag{22}$$

$\equiv$ { Eq-+, lei(27) ; Natural-id, lei(1) }

$$\left\{ \begin{array}{l} lookahead\_sep \cdot nil = k_1 \\ lookahead\_sep \cdot cons = k_2 \cdot (id \times \langle wc\_w, lookahead\_sep \rangle) \end{array} \right. \tag{23}$$

$\equiv$ { Igualdade Extensional, lei(71) ; Def-comp, lei(72) ; Def-const, lei(74) }

$$\left\{ \begin{array}{l} lookahead\_sep\ [\,] = k_1\ x \\ lookahead\_sep\ (cons\ x) = k_2\ ((id \times \langle wc\_w, lookahead\_sep \rangle)\ x) \end{array} \right. \tag{24}$$

$\equiv$ { $lookahead\_sep\ [\,] = True$; x := (x,y) ; Def-x, lei(77) }

$$\left\{ \begin{array}{l} True = k_1\ x \\ lookahead\_sep\ (cons\ (x, y)) = h_2\ (id\ x, \langle wc\_w, lookahead\_sep \rangle\ y) \end{array} \right. \tag{25}$$

$\equiv$ { Def-id, lei(73) ; Def-split, lei(76) ; cons (x,y) = (x:y) }

$$\left\{ \begin{array}{l} True = k_1\ x \\ lookahead\_sep\ (x : y) = k_2\ (x, (wc\_w\ y, lookahead\_sep\ y)) \end{array} \right. \tag{26}$$

$\equiv$ { $lookahead\_sep\ (x : y) = sp\ x$ }

$$\left\{ \begin{array}{l} True = k_1\ x \\ sp\ x = k_2\ (x, (wc\_w\ y, lookahead\_sep\ y)) \end{array} \right. \tag{27}$$

$\equiv$ { Propriedade simétrica da igualdade }

$$\left\{ \begin{array}{l} k_1\ x = True \\ k_2\ (x, (wc\_w\ y, lookahead\_sep\ y)) = sp\ x \end{array} \right. \tag{28}$$

$$\Box \tag{29}$$

A partir do cálculo efetuado em cima concluímos que k_1 é a função constante $\underline{True}$ e k_2 tem que ser a função $sp \cdot \pi_1$.

$$k_1 = \underline{True}$$
$$k_2 = sp \cdot \pi_1$$

Problema 3

$$inX :: u + (i, (X\ u\ i, X\ u\ i)) \to X\ u\ i$$
$$inX = [XLeaf, \widehat{\widehat{Node}} \cdot assocl]$$
$$outX\ (XLeaf\ u) = i_1\ u$$
$$outX\ (Node\ i\ l\ r) = i_2\ (i, (l, r))$$
$$baseX\ f\ h\ g = f + (h \times (g \times g))$$
$$recX\ f = id + (id \times (f \times f))$$
$$cataX\ g = g \cdot recX\ (cataX\ g) \cdot outX$$
$$outUnit\ (Image\ a) = i_1\ a$$
$$outUnit\ (Text\ b) = i_2\ b$$

Inserir a partir daqui o resto da resolução deste problema:
....

Problema 4

$$\begin{tikzcd}
(A \times A)^* & & 1 + (A \times A) \times (A \times A)^* \arrow[ll, "inList = {[Nil, Cons]}"'] \\
A^* \arrow[u, "pairL"] \arrow[r, "outList"'] & 1 + A \times A^* \arrow[r, "g"'] & 1 + (A \times A) \times A^* \arrow[u, "id + id \times pairL"']
\end{tikzcd}$$

Nós sabemos que $g = id + \langle f, g \rangle$. Então sobra determinar f e g. Para tal vamos analisar cada função individualmente.

A função f vai processar o $tuple$ derivado do $outList$ de maneira a que este produza o $tuple$ desejado. Este efeito é obtido através da função auxiliar $auxPairL$ que como podemos ver toma como tipo de entrada um $tuple$ igual ao originado por $outList$ e gera um novo $tuple$ como o desejado.

A função g quer apenas preservar o resto da lista portanto trata-se apenas da projeção π_2.

No final do anamorfismo temos uma estrutura quase semelhante à desejada com a única exceção de que se se tratar do caso em que a lista gerada tem mais de um elemento a lista teria um elemento a mais no final. Assim para resolver esse problema basta compor outra função auxiliar, $pairLAux$ que remove o último elemento de uma lista se esta tiver mais do que um elemento, com o resultado do anamorfismo originando o efeito pretendido.

$pairL :: [a] \rightarrow [(a, a)]$
$pairL = pairLAux \cdot [\!(g)\!]$ **where**
$\quad g = (id + \langle auxPairL, \pi_2 \rangle) \cdot outList$

$pairLAux :: [(a, a)] \rightarrow [(a, a)]$
$pairLAux\ [x] = [x]$
$pairLAux\ l = init\ l$

$auxPairL :: (a, [a]) \rightarrow (a, a)$
$auxPairL\ (a, [\,]) = (a, a)$
$auxPairL\ (a, h : t) = (a, h)$

$$\begin{array}{ccc}
(Pos \times Pos)^* & \xleftarrow{inList=[nil,cons]} & 1 + (Pos \times Pos) \times (Pos \times Pos)^* \\
\Big\downarrow {\scriptstyle (\!|g|\!)} & & \Big\downarrow {\scriptstyle id + id \times (\!|g|\!)} \\
Map^{Map} & \xleftarrow[g=[\underline{id}, f_2]]{} & 1 + (Pos \times Pos) \times Map^{Map}
\end{array}$$

Para determinar f_2 temos primeiro que entender o que é que esta recebe à entrada. Como podemos ver no diagrama acima f_2 recebe à entrada um $tuple$ cujo primeiro elemento trata-se de um $tuple$ de posições e o segundo de uma função que gera um mapa quando receber um mapa. Depois, temos que entender o que é que f_2 tem que gerar. Mais uma vez pelo diagrama facilmente concluimos que f_2 tem que gerar também uma função que gera um mapa quando receber um mapa. Com isto podemos deduzir que o comportamento funcional de f_2 será o seguinte:

1. Processar o primeiro elemento do tuple e gerar uma função que gera um mapa quando receber um mapa;

2. Compor essa nova função com o segundo elemento do tuple e gerar uma nova função capaz de gerar um mapa quando receber um mapa.

À primeira vista bastante simples portanto comecemos pelo ponto número 1. Para este a estratégia adotada foi primeiro determinar a célula que iria substituir a já presente. Felizmente já havia disponível a função $toCell$ que trata desse processamento. No entanto esta toma como entrada duas posições e não um $tuple$ de posições mas nada que um $uncurry$ não consiga resolver. Assim a função responsável por gerar a nova célula é a função $auxToCell$ que se trata basicamente de $\widehat{toCell}$.

Depois temos que também saber onde é que precisaremos de substituir essa célula. Naturalmente essa substituição é feita na posição correspondete ao primeiro elemento do $tuple$ de posições. Precisamos então também de guardar essa informção através da projeção π_1.

Com isto o próximo passo será efetivamente gerar a função que é capaz de gerar um mapa quando receber um mapa. A função $substMatrix$ serve para esse mesmo propósito. Esta é semelhante à função $subst$ já disponibilizada e faz mesmo uso dela. $substMatrix$ recebe à entrada um elemento a substituir, a coluna e alinha onde substituílo. A função é nada menos que um catamorfismo dos naturais que percorre a linha como argumento até este ser 0 e aí efetua o $subst\ x\ coluna$ que substitui o elemento x na coluna $coluna$.

$$
\begin{tikzcd}
\mathbb{N}_0 \arrow[d, "substMatrix\ x\ coluna"'] & & 1 + \mathbb{N}_0 \arrow[ll, "\mathsf{in}"'] \arrow[d, "id + (substMatrix\ x\ coluna)"] \\
Map^{Map} & & 1 + Map^{Map} \arrow[ll, "g = [f_1, f_2]"]
\end{tikzcd}
$$

f_1, o caso quando a linha é 0 é a função constante que executa $subst\ x\ coluna$ à cabeça da lista que é passada como argumento e f_2 é o caso resursivo.

Assim, através da função $substMatrix$ somos capazes de gerar uma função que gera mapas a partir de mapas com recurso ao elemeto a substitui e à posição onde substitui-lo. Falta pensar em como lhe passar essa informação. Como foi dito em cima já é possível preservá-la através das funções $auxToCell$ e π_1. Estes se forem colocados num $split$ temos $\langle auxToCell, \pi_1 \rangle$ que devolve um $tuple$ com o elemento a substituir no primeiro elemento e a posição onde substituilo no segundo. Assim a função $substMatrix$ tem que poder aceitar esse tipo à entrada. Mais uma vez nada que alguns $uncurry$ e um $assocl$ não consiga resolver. Assim obtemos a função $auxMarkMap$ que se trata de $\widehat{\widehat{substMatrix}} \cdot assocl$ e gera uma função que gera mapas a partir de mapas recebendo um $tuple$ como o descrito à entrada.

Agora só falta o ponto número 2. Este é bastante mais simples que o 1 pois a dificuldade está em conseguir compor duas funções presentes num $tuple$. A função auxiliar $compX$ trata disso e assim obtemos uma função f_2 como a seguinte $compX \cdot (auxMarkMap \cdot \langle auxToCell, \pi_1 \rangle \times id)$.

$$
\begin{array}{l}
markMap :: [Pos] \rightarrow Map \rightarrow Map \\
markMap\ l = (\![\underline{id}, f_2]\!)\ (pairL\ l)\ \mathbf{where} \\
\quad f_2 = compX \cdot (auxMarkMap \cdot \langle auxToCell, \pi_1 \rangle \times id) \\
compX\ (f, g) = f \cdot g \\
auxMarkMap = \widehat{\widehat{substMatrix}} \cdot assocl \\
auxToCell = \widehat{toCell} \\
substMatrix :: a \rightarrow Int \rightarrow Int \rightarrow [[a]] \rightarrow [[a]] \\
substMatrix\ x\ coluna = (\![f_1, f_2]\!)\ \mathbf{where} \\
\quad f_1 = \lambda(h : t) \rightarrow subst\ x\ coluna\ h : t \\
\quad f_2\ f\ (h : t) = h : f\ t
\end{array}
$$

$$
\begin{array}{l}
scout :: Map \rightarrow Pos \rightarrow Pos \rightarrow Int \rightarrow [[Pos]] \\
scout\ m\ s\ t = (\![f_1, (\ggg f_2\ m\ s)]\!)\ \mathbf{where} \\
\quad f_1 = \perp \\
\quad f_2 = \perp
\end{array}
$$

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 $(pairL\ l)$ é a lista original l a menos do último elemento. Analogamente, a lista correspondente ao lado direito dos pares em $(pairL\ l)$ é a lista original l a menos do primeiro elemento:*

$$prop\_reconst\ l = \perp$$

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:*

$$prop\_prefix2\ l\ l' = \perp$$

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.*

$$prop\_nmbrs\ l\ c = \bot$$
$$count :: (Eq\ a) \Rightarrow a \to [\,a\,] \to Int$$
$$count = \bot$$

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

$$inBounds\ m\ (x, y) = \bot$$
$$prop\_idemp2\ l\ m = \bot$$

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

$$prop\_extr2\ l\ m = \bot$$

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

$$prop\_reach\ m\ t\ n\ n' = \bot$$

Índice

Referências

[1] B.W. Kernighan and D.M. Richtie. *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: 6 de fevereiro de 2022.