



Universidade do Minho
Escola de Engenharia

Cálculo de Programas

Trabalho Prático (2025/26)

Lic. em Ciências da Computação

Lic. em Engenharia Informática

Grupo G99

axxxxxx Nome
axxxxxx Nome
axxxxxx Nome

Preâmbulo

Em [Cálculo de Programas](#) pretende-se ensinar a programação de computadores como uma disciplina científica. Para isso parte-se de um repertório de *combinadores* que formam uma álgebra da programação 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 abordarem os problemas propostos no trabalho, os grupos devem ler com atenção o anexo A onde encontrarão as instruções relativas ao *software* a instalar, etc.

Valoriza-se a escrita de *pouco* código que corresponda a soluções simples e elegantes que utilizem os combinadores de ordem superior estudados na disciplina.

Avaliação. Faz parte da avaliação do trabalho a sua defesa por parte dos elementos de cada grupo. Estes devem estar preparados para responder a perguntas sobre *qualquer* dos problemas deste enunciado. A prestação *individual* de cada aluno nessa defesa oral será uma componente importante e diferenciadora da avaliação.

Problema 1

Uma serialização (ou travessia) de uma árvore é uma sua representação sob a forma de uma lista. Na biblioteca *BTree* encontram-se as funções de serialização *inordt*, *preordt* e *postordt*, que fazem as travessias *in-order*, *pre-order* e *post-order*, respectivamente. Todas essas travessias são catamorfismos que percorrem a árvore argumento em regime *depth-first*.

Pretende-se agora uma função *bforder* que faça a travessia em regime *breadth-first*, isto é, por níveis. Por exemplo, para a árvore t_1 dada em anexo e mostrada na figura a seguir,



a função deverá dar a lista

[5, 3, 7, 1, 4, 6, 8]

em que se vê como os níveis 5, depois 3, 7 e finalmente 1, 4, 6, 8 foram percorridos.

Pretendemos propor duas versões dessa função:

1. Uma delas envolve um catamorfismo de *BTrees*:

```
bfsLevels :: BTTree a → [a]  
bfsLevels = concat · levels
```

Complete a definição desse catamorfismo:

```
levels :: BTTree a → [[a]]  
levels = (λ glevels)
```

2. A segunda proposta,

```
bft :: BTTree a → [a]
```

deverá basear-se num anamorfismo de listas.

Sugestão: estudar o artigo [2] cujo PDF está incluído no material deste trabalho. Quando fizer testes ao seu código pode, se desejar, usar funções disponíveis na biblioteca *Exp* para visualizar as árvores em GraphViz (formato .dot).

Justifique devidamente a sua resolução, que deverá vir acompanhada de diagramas explicativos. Como já se disse, valoriza-se a escrita de *pouco* código que corresponda a soluções simples e elegantes que utilizem os combinadores de ordem superior estudados na disciplina.

Problema 2

Considere a seguinte função em Haskell:

```
f x = wrapper · worker where  
    wrapper = head  
    worker 0 = start x  
    worker (n + 1) = loop x (worker n)  
loop x [s, h, k, j, m] =  
    [h / k + s, x ↑ 2 * h, k * j, j + m, m + 8]  
start x = [x, x ↑ 3, 6, 20, 22]
```

Pode-se provar pela lei de recursividade mútua que $f x n$ calcula o seno hiperbólico de x , $\sinh x$, para n aproximações da sua série de Taylor. Faça a derivação da função dada a partir da referida série de Taylor, apresentando todos os cálculos justificativos, tal como se faz para outras funções no capítulo respectivo do texto base desta UC [3].

Problema 3

Quem em Braga observar, ao fim da tarde, o tráfego onde a Avenida Clairmont Fernand se junta à N101, aproximadamente na coordenada [41°33'46.8"N 8°24'32.4"W](#) – ver as setas da figura que se segue – reparará nas sequências imparáveis (infinitas!) de veículos provenientes dessas vias de circulação.

Mas também irá observar um comportamento interessante por parte dos condutores desses veículos: por regra, *cada carro numa via deixa passar, à sua frente, exactamente outro carro da outra via*.



Este comportamento *civilizado* chama-se *fair-merge* (ou *fair-interleaving*) de duas sequências infinitas, também designadas *streams* em ciência da computação. Seja dado o tipo dessas sequências em Haskell,

```
data Stream a = Cons (a, Stream a) deriving Show
```

para o qual se define também:

```
out (Cons (x, xs)) = (x, xs)
```

O referido comportamento civilizado pode definir-se, em Haskell, da forma seguinte:¹

```
fair_merge :: (Stream a, Stream a) + (Stream a, Stream a) → Stream a
fair_merge = [h, k] where
  h (Cons (x, xs), y) = Cons (x, k (xs, y))
  k (x, Cons (y, ys)) = Cons (y, h (x, ys))
```

Defina *fair_merge* como um **anamorfismo** de *Streams*, usando o combinador

```
[(g)] = Cons · (id × [(g)]) · g
```

e a seguinte estratégia:

- Derivar a lei **dual** da recursividade mútua,

$$[f, g] = [(h, k)] \equiv \begin{cases} out \cdot f = F[f, g] \cdot h \\ out \cdot g = F[f, g] \cdot k \end{cases} \quad (1)$$

tal como se fez, nas aulas, para a que está no formulário.

- Usar (1) na resolução do problema proposto.

Justificar devidamente a resolução, que deverá vir acompanhada de diagramas explicativos.

Problema 4

Como se sabe, é possível pensarmos em catamorfismos, anamorfismos etc *probabilísticos*, quer dizer, programas recursivos que dão distribuições como resultados. Por exemplo, podemos pensar num combinador

```
pcataList :: ((()) + (a, b) → Dist b) → [a] → Dist b
```

¹ O facto das sequências serem infinitas não nos deve preocupar, pois em Haskell isso é lidado de forma transparente por [lazy evaluation](#).

que é muito parecido com

$$(\cdot) :: ((a, b) \rightarrow b) \rightarrow [a] \rightarrow b$$

da biblioteca [List](#). A principal diferença é que o gene de *pcataList* é uma função probabilística.

Como exemplo de utilização, recorde-se que $([zero, add])$ soma todos os elementos da lista argumento, por exemplo:

$$([zero, add]) [20, 10, 5] = 35.$$

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

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

Se se correr

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

obter-se-á:

| | |
|----|-------|
| 35 | 81.0% |
| 25 | 9.0% |
| 5 | 9.0% |
| 15 | 1.0% |

Com base neste exemplo, resolva o seguinte

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

Qual a probabilidade de a palavra "atacar" da mensagem

words "Vamos atacar hoje"

se perder, isto é, o resultado da transmissão ser ["Vamos", "hoje", "stop"]? E a de seguirem todas as palavras, mas faltar o "stop" no fim? E a da transmissão ser perfeita?

Responda a estas perguntas encontrando gene tal que

transmitir = *pcataList gene*

descreve o comportamento do aparelho. Justificar devidamente a resolução, que deverá vir acompanhada de diagramas explicativos.

Anexos

A Natureza do trabalho a realizar

Este trabalho teórico-prático deve ser realizado por grupos de 3 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.

Para cumprir de forma integrada os objectivos do trabalho vamos recorrer a uma técnica de programação dita “[literária](#)” [1], 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 `cp2526t.pdf` que está a ler é já um exemplo de [programação literária](#): foi gerado a partir do texto fonte `cp2526t.lhs`¹ que encontrará no [material pedagógico](#) desta disciplina descompactando o ficheiro `cp2526t.zip`.

Como se mostra no esquema abaixo, de um único ficheiro (*lhs*) gera-se um PDF ou faz-se a interpretação do código [Haskell](#) que ele inclui:



Vê-se assim que, para além do [GHCi](#), serão necessários os executáveis [pdflatex](#) e [lhs2TeX](#). Para facilitar a instalação e evitar problemas de versões e conflitos com sistemas operativos, é recomendado o uso do [Docker](#) tal como a seguir se descreve.

B Docker

Recomenda-se o uso do [container](#) cuja imagem é gerada pelo [Docker](#) a partir do ficheiro `Dockerfile` que se encontra na diretoria que resulta de descompactar `cp2526t.zip`. Este [container](#) deverá ser usado na execução do [GHCi](#) e dos comandos relativos ao [LATEX](#). (Ver também a `Makefile` que é disponibilizada.)

Após [instalar o Docker](#) e descargar o referido zip com o código fonte do trabalho, basta executar os seguintes comandos:

```
$ docker build -t cp2526t .
$ docker run -v ${PWD}:/cp2526t -it cp2526t
```

NB: O objetivo é que o container seja usado *apenas* para executar o [GHCi](#) e os comandos relativos ao [LATEX](#). Deste modo, é criado um *volume* (cf. a opção `-v ${PWD}:/cp2526t`) que permite que a diretoria em que se encontra na sua máquina local e a diretoria `/cp2526t` no [container](#) sejam partilhadas.

Pretende-se então que visualize/editie os ficheiros na sua máquina local e que os compile no [container](#), executando:

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

```
$ lhs2TeX cp2526t.lhs > cp2526t.tex  
$ pdflatex cp2526t
```

[lhs2TeX](#) é o pre-processador que faz “pretty printing” de código Haskell em [LaTeX](#) e que faz parte já do [container](#). Alternativamente, basta executar

```
$ make
```

para obter o mesmo efeito que acima.

Por outro lado, o mesmo ficheiro cp2526t.lhs é executável e contém o “kit” básico, escrito em [Haskell](#), para realizar o trabalho. Basta executar

```
$ ghci cp2526t.lhs
```

Abra o ficheiro cp2526t.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.

C Em que consiste o TP

Em que consiste, então, o *relatório* a que se referiu acima? É a edição do texto que está a ser lido, preenchendo o anexo G 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 cp2526t.aux  
$ makeindex cp2526t.idx
```

e recompilar o texto como acima se indicou. (Como já se disse, pode fazê-lo correndo simplesmente [make](#) no [container](#).)

No anexo F disponibiliza-se algum código [Haskell](#) relativo aos problemas que são colocados. Esse anexo deverá ser consultado e analisado à medida que isso for necessário.

Deve ser feito uso da [programação literária](#) para documentar bem o código que se desenvolver, em particular fazendo diagramas explicativos do que foi feito e tal como se explica no anexo D que se segue.

D Como exprimir cálculos e diagramas em LaTeX/lhs2TeX

Como primeiro exemplo, estudar o texto fonte ([lhs](#)) do que está a ler¹ onde se obtém o efeito seguinte:²

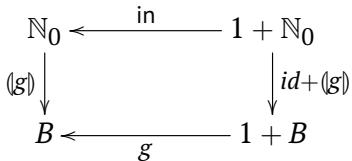
$$\begin{aligned} id &= \langle f, g \rangle \\ &\equiv \{ \text{ universal property } \} \end{aligned}$$

¹ Procure e.g. por "sec:diagramas".

² Exemplos tirados de [3].

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

Os diagramas podem ser produzidos recorrendo à package [xymatrix](#), por exemplo:



E O mónade das distribuições probabilísticas

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

newtype Dist $a = D \{ unD :: [(a, ProbRep)] \}$ (2)

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

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



será representada pela distribuição

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

que o [GHCi](#) mostrará assim:

```

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

```

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

$d2 = uniform (\text{words "Uma frase de cinco palavras"})$

isto é

```

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

```

distribuição *normais*, eg.

$$d3 = \text{normal}[10..20]$$

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

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

em que $g : A \rightarrow \text{Dist } B$ e $f : B \rightarrow \text{Dist } C$ são funções **monádicas** que representam *computações probabilísticas*.

Este mónade é adequado à resolução de problemas de *probabilidades e estatística* usando programação funcional, de forma elegante e como caso particular da programação monádica.

F Código fornecido

Problema 1

Árvores exemplo:

```

t1 :: BTTree Int
t1 = Node(5, (Node(3, (Node(1, (Empty, Empty)), Node(4, (Empty, Empty)))), 
    Node(7, (Node(6, (Empty, Empty)), Node(8, (Empty, Empty)))))))

t2 :: BTTree Int
t2 =
    node 1
        (node 2 (node 4 Empty Empty) (node 5 Empty Empty))
        (node 3 (node 6 Empty Empty) (node 7 Empty Empty))

t3 :: BTTree Char
t3 =
    node 'A'
        (node 'B' (node 'C' (node 'D' Empty Empty) Empty) Empty)
        (node 'E' Empty Empty)

t4 :: BTTree Char
t4 =
    node 'A'
        (node 'B' (node 'C' (node 'D' Empty Empty) Empty) Empty)
        Empty

t5 :: BTTree Int
t5 =
    node 1

```

¹ Para mais detalhes ver o código fonte de [Probability](#), que é uma adaptação da biblioteca [PFP](#) (“Probabilistic Functional Programming”). Para quem quiser saber mais recomenda-se a leitura do artigo [?].

$$\begin{aligned}
 & (\text{node } 2 (\text{node } 4 \text{ Empty Empty}) \text{ Empty}) \\
 & (\text{node } 3 \text{ Empty } (\text{node } 5 (\text{node } 6 \text{ Empty Empty}) \text{ Empty})) \\
 & \text{node } a b c = \text{Node } (a, (b, c))
 \end{aligned}$$

G 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 ao anexo, bem como diagramas e/ou outras funções auxiliares que sejam necessárias.

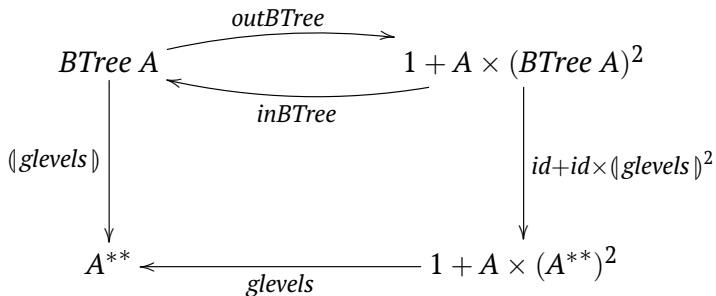
Importante: Não pode ser alterado o texto deste ficheiro fora deste anexo.

Problema 1

Na **primeira questão** do Problema 1, é-nos pedido uma implementação da função *levels* que coloca os elementos de uma árvore binária numa lista, tendo esta várias listas, cada uma relativa a um nível da árvore.

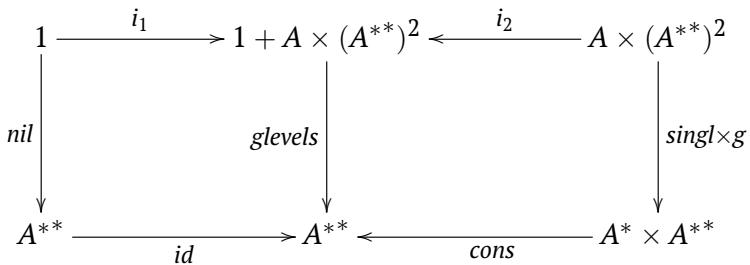
$$\begin{aligned}
 glevels &= [\text{nil}, \text{cons} \cdot (\text{singl} \times g)] \\
 \text{where} \\
 g ([], r) &= r \\
 g (l, []) &= l \\
 g ((l : ls), (r : rs)) &= (l ++ r) : g (ls, rs)
 \end{aligned}$$

O diagrama do catamorfismo *levels* é o seguinte:



O gene *glevels* deverá ser capaz de juntar os níveis de duas sub-árvores e adicionar a raiz da árvore à cabeça da lista resultante. Logo será necessário realizar uma espécie de *zip* aos elementos das listas. Esta operação está pré definida em Haskell, no entanto esta função remove elementos de uma das listas, caso não tenham o mesmo comprimento, então implementamos a função auxiliar *g* que concatena os elementos (listas) de duas listas, sem perder informação.

O diagrama do gene *glevels* é o seguinte:



Na **segunda questão** deste problema, o desafio baseia-se em implementar uma travessia em largura através de um anamorfismo de listas. A estratégia usada passa por usar uma lista de árvores, que corresponde à queue de nodos da árvore a explorar.

$$bft = concat \cdot [(id + g) \cdot outList] \cdot singl$$

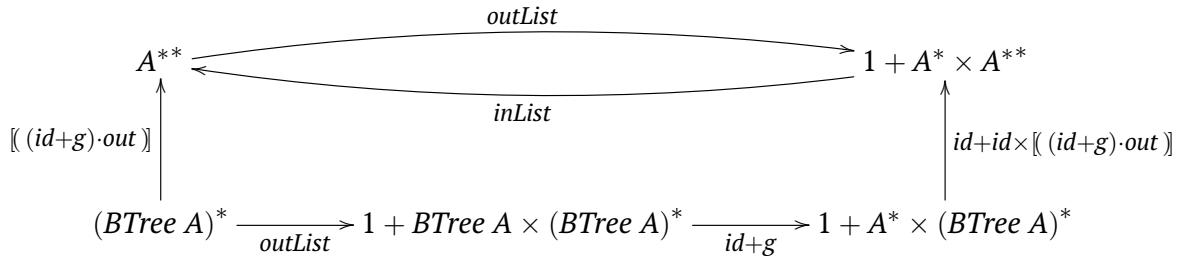
where

$$g :: (BTree a, [BTree a]) \rightarrow ([a], [BTree a])$$

$$g (Empty, rest) = ([], rest)$$

$$g (Node (x, (l, r)), rest) = ([x], rest ++ [l, r])$$

O diagrama deste anamorfismo é o seguinte:



Problema 2

Neste problema é-nos proposto derivar a definição do seno hiperbólico para n aproximações da sua série de Taylor.

$$\sinh(x) = \sum \frac{x \uparrow (2n+1)}{(2n+1)!}$$

O primeiro passo será resolver o somatório, transformando a definição de $\sinh x$ numa função recursiva:

$$\sinh x 0 = x$$

$$\sinh x (n+1) = x \uparrow (2 * n + 3) / (2 * n + 3)! + \sinh x n$$

O corpo desta função é dependente do seu input $n+1$, logo teremos que simplificar $\sinh x$. Podemos então introduzir duas funções:

$$f1 x n = x \uparrow (2 * n + 3)$$

$$f2 n = (2 * n + 3)!$$

Desta forma $\sinh x$ ficará dependente das definições de $f1$ e $f2$. Será necessário obter as definições recursivas primitivas destas duas funções, tal é alcançável através da substituição do valor de n por 0, de forma a obter a base da recursividade:

$$f1 x 0 = x \uparrow (2 * 0 + 3) = x \uparrow 3$$

$$f2 0 = (2 * 0 + 3)! = 6$$

A definição recursiva de $f1$ pode ser obtida através da divisão entre $f1 x (n+1)$ e $f1 x n$:

$$\frac{f1 x (n+1)}{f1 x n} = \frac{x \uparrow (2n+5)}{x \uparrow (2n+3)} = x \uparrow 2$$

Podemos aplicar o mesmo procedimento à função $f2$:

$$\frac{f2\ x\ (n+1)}{f2\ x\ n} = \frac{(2\ n+5)!}{(2\ n+3)!} = (2\ n+5) * (2\ n+4) = 4\ n \uparrow 2 + 18\ n + 20$$

Temos então as definições recursivas de $f1$ e $f2$:

$$\begin{aligned} f1\ x\ 0 &= x \uparrow 3 \\ f1\ x\ (n+1) &= x \uparrow 2 * f1\ x\ n \\ f2\ 0 &= 6 \\ f2\ (n+1) &= (4 * n \uparrow 2 + 18 * n + 20) * f2\ n \end{aligned}$$

Podemos verificar que $f2$ ainda depende de n , logo teremos de repetir o processo de forma a simplificar $f2$. Introduzimos $f3 = 4 * n \uparrow 2 + 18 * n + 20$. Por substituição obtemos o caso base de $f3$ e resolvendo a equação $f3\ (n+1) - f3\ n$ obtemos a definição recursiva:

$$\begin{aligned} f3\ 0 &= 20 \\ f3\ (n+1) &= 8 * n + 22 + f3\ n \end{aligned}$$

Será necessário repetir o processo uma última vez, pois $f3$ continua dependente de n . Para obter $f4$ usamos o mesmo processo apresentado anteriormente, isto é, tomamos $f4\ n = 8 * n$, substituímos n por 0, para determinar a base da recursividade, e calculamos $f4\ (n+1) - f4\ n$ para determinar o corpo da recursividade:

$$\begin{aligned} f4\ 0 &= 22 \\ f4\ (n+1) &= 8 + f4\ n \end{aligned}$$

Podemos então apresentar as versões primitivas das funções mencionadas anteriormente, que nos irão ajudar na dedução da função f :

$$\begin{aligned} f4\ 0 &= 22 \\ f4\ (n+1) &= 8 + f4\ n \\ f3'\ 0 &= 20 \\ f3'\ (n+1) &= f4\ n + f3'\ n \\ f2'\ 0 &= 6 \\ f2'\ (n+1) &= f3'\ n * f2'\ n \\ f1\ x\ 0 &= x \uparrow 3 \\ f1\ x\ (n+1) &= x \uparrow 2 * f1\ x\ n \\ sinh'\ x\ 0 &= x \\ sinh'\ x\ (n+1) &= f1\ x\ n / f2'\ n + sinh'\ x\ n \end{aligned}$$

Todas estas funções partilham uma característica fundamental para que seja possível derivar f , todas elas trabalham sobre o Functor dos Naturais. Através das seguintes regras, podemos derivar uma versão de f , que usa o catamorfismo for $b\ i$ dos Naturais:

- O corpo do ciclo loop terá tantos argumentos quanto o número de funções mutuamente recursivas.
- Para as variáveis escolhem-se os próprios nomes das funções, pela ordem que se achar conveniente.
- Para os resultados vão-se buscar as expressões respectivas, retirando a variável n .

- Em start coleccionam-se os resultados dos casos de base das funções, pela mesma ordem.

Seguindo as regras apresentadas e a **lei da recursividade mútua**, podemos obter a seguinte função (**Nota:** por simplicidade e elegância, em vez de se utilizar pares de elementos, usaremos uma lista e a função head):

$$\begin{aligned} f' x &= \text{head} \cdot \text{for loop start where} \\ \text{loop } [s, f1, f2, f3, f4] &= \\ [f1 / f2 + s, x \uparrow 2 * f1, f2 * f3, f3 + f4, 8 + f4] \\ \text{start } x &= [x, x \uparrow 3, 6, 20, 22] \end{aligned}$$

No entanto, a função f está definida em formato pointwise, logo será necessário aplicar a regra Universal-Cata a for loop start:

$$\begin{aligned} \text{worker} &= \text{for loop start} \\ \equiv & \quad \{ \text{Def. for, universal cata} \} \\ \text{worker} \cdot \text{in} &= [\text{start}, \text{loop}] \cdot F \text{ worker} \\ \equiv & \quad \{ F f = id + f, \text{Absorção-+}, \text{Natural-id} \} \\ \text{worker} \cdot \text{in} &= [\text{start}, \text{loop} \cdot \text{worker}] \\ \equiv & \quad \{ \text{Def. in, Fusão-+}, \text{Eq-+}, \text{pointwise}, \text{Def. composição} \} \\ & \quad \left\{ \begin{array}{l} \text{worker } 0 = \text{start } x \\ \text{worker } (n + 1) = \text{loop } (\text{worker } n) \end{array} \right. \end{aligned}$$

Através da definição pointwise de worker , e de algumas mudanças de nome às variáveis, derivamos a definição do seno hiperbólico de x , para n aproximações da sua série de Taylor:

$$\begin{aligned} f x &= \text{wrapper} \cdot \text{worker where} \\ \text{wrapper} &= \text{head} \\ \text{worker } 0 &= \text{start } x \\ \text{worker } (n + 1) &= \text{loop } x (\text{worker } n) \\ \text{loop } x [s, h, k, j, m] &= \\ [h / k + s, x \uparrow 2 * h, k * j, j + m, m + 8] \\ \text{start } x &= [x, x \uparrow 3, 6, 20, 22] \end{aligned}$$

Problema 3

O primeiro passo para resolver este problema será derivar a lei **dual** da recursividade mútua:

$$\begin{aligned} [f, g] &= [[h, k]] \\ \equiv & \quad \{ \text{universal ana} \} \\ \text{out} \cdot [f, g] &= F [f, g] \cdot [h, k] \\ \equiv & \quad \{ \text{Fusão-+ (twice)} \} \\ [\text{out} \cdot f, \text{out} \cdot g] &= [F [f, g] \cdot h, F [f, g] \cdot k] \\ \equiv & \quad \{ \text{Eq-+} \} \\ & \quad \left\{ \begin{array}{l} \text{out} \cdot f = F [f, g] \cdot h \\ \text{out} \cdot g = F [f, g] \cdot k \end{array} \right. \end{aligned}$$

De seguida iremos derivar um anamorfismo a partir do coproduto de h e k , definidos em *fair_merge*:

$$\begin{aligned}
& \left\{ \begin{array}{l} h(Cons(x, xs), y) = Cons(x, k(xs, y)) \\ k(x, Cons(y, ys)) = Cons(y, h(x, ys)) \end{array} \right. \\
\equiv & \quad \{ \text{Def-}\times, \text{Def. assocr, Def. aux} \} \\
& \left\{ \begin{array}{l} h((Cons \times id)((x, xs), y)) = Cons((id \times k)(assocr((x, xs), y))) \\ k((id \times Cons)(x, (y, ys))) = Cons((id \times h)(aux(x, (y, ys)))) \end{array} \right. \\
\equiv & \quad \{ \text{Def-comp, pointfree} \} \\
& \left\{ \begin{array}{l} h \cdot (Cons \times id) = Cons \cdot (id \times k) \cdot assocr \\ k \cdot (id \times Cons) = Cons \cdot (id \times h) \cdot aux \end{array} \right. \\
\equiv & \quad \{ \text{'Shunt-right'} \} \\
& \left\{ \begin{array}{l} out \cdot h \cdot (Cons \times id) = (id \times k) \cdot assocr \\ out \cdot k \cdot (id \times Cons) = (id \times h) \cdot aux \end{array} \right. \\
\Leftarrow & \quad \{ \text{Leibniz} \} \\
& \left\{ \begin{array}{l} out \cdot h \cdot (Cons \times id) \cdot (out \times id) = (id \times k) \cdot assocr \cdot (out \times id) \\ out \cdot k \cdot (id \times Cons) \cdot (id \times out) = (id \times h) \cdot aux \cdot (id \times out) \end{array} \right. \\
\equiv & \quad \{ \text{isomorfismo in/out} \} \\
& \left\{ \begin{array}{l} out \cdot h = (id \times k) \cdot assocr \cdot (out \times id) \\ out \cdot k = (id \times h) \cdot aux \cdot (id \times out) \end{array} \right. \\
\equiv & \quad \{ \text{Cancelamento-+ (twice)} \} \\
& \left\{ \begin{array}{l} out \cdot h = (id \times ([h, k] \cdot i_2)) \cdot assocr \cdot (out \times id) \\ out \cdot k = (id \times ([h, k] \cdot i_1)) \cdot aux \cdot (id \times out) \end{array} \right. \\
\equiv & \quad \{ \text{Functor-}\times \} \\
& \left\{ \begin{array}{l} out \cdot h = (id \times [h, k]) \cdot (id \times (i_2)) \cdot assocr \cdot (out \times id) \\ out \cdot k = (id \times [h, k]) \cdot (id \times (i_1)) \cdot aux \cdot (id \times out) \end{array} \right. \\
\equiv & \quad \{ \text{F } f = id \times f \} \\
& \left\{ \begin{array}{l} out \cdot h = F[h, k] \cdot (id \times (i_2)) \cdot assocr \cdot (out \times id) \\ out \cdot k = F[h, k] \cdot (id \times (i_1)) \cdot aux \cdot (id \times out) \end{array} \right. \\
\equiv & \quad \{ \text{Lei dual da recursividade mútua} \} \\
& [h, k] = [[(id \times (i_2)) \cdot assocr \cdot (out \times id), (id \times (i_1)) \cdot aux \cdot (id \times out)]]
\end{aligned}$$

Nota: Nos cálculos apresentados acima é referida uma função *aux*, esta função foi usada para simplificar a apresentação da função *fair_merge'*. Podemos observar na definição de *fair_merge*, mais precisamente na definição de *k*, que as instâncias *x* e *y* trocam de posições, logo a função *aux* efetua esta troca.

$$\begin{aligned}
& fair_merge' = [[l, r]] \\
& \text{where} \\
& \quad aux(x, (y, z)) = (y, (x, z)) \\
& \quad l = (id \times (i_2)) \cdot assocr \cdot (out \times id) \\
& \quad r = (id \times (i_1)) \cdot aux \cdot (id \times out)
\end{aligned}$$

O diagrama do anamorfismo *fair_merge'* é o seguinte:

$$\begin{array}{ccc}
 Stream A & \xleftarrow{\quad Cons \quad} & A \times Stream A \\
 \uparrow \llbracket [l,r] \rrbracket & & \uparrow id + \llbracket [l,r] \rrbracket \\
 (Stream A)^2 + (Stream A)^2 & \xrightarrow{\quad [l,r] \quad} & A \times (Stream A)^2 + (Stream A)^2
 \end{array}$$

O diagrama da alternativa $[l, r]$ é o seguinte:

$$\begin{array}{ccccc}
 (Stream A)^2 & \xrightarrow{i_1} & (Stream A)^2 + (Stream A)^2 & \xleftarrow{i_2} & (Stream A)^2 \\
 \downarrow out \times id & & \downarrow [l,r] & & \downarrow id \times out \\
 (A \times Stream A) \times Stream A & & & & Stream A \times (A \times Stream A) \\
 \downarrow assocr & & & & \downarrow aux \\
 A \times (Stream A)^2 & & & & A \times (Stream A)^2 \\
 & \searrow id \times (i_2) & & \swarrow id \times (i_1) & \\
 & & A \times (Stream A)^2 + (Stream A)^2 & &
 \end{array}$$

De forma a garantir que o comportamento da função *fair_merge'* é realmente o correto, definimos as seguintes instâncias de *Stream Int* e a função *takeStream*, dado que estamos perante um tipo de dados infinito, podemos tirar partido da *lazy evalution* do *Haskell*.

```

ones :: Stream Int
ones = Cons (1, ones)
twos :: Stream Int
twos = Cons (2, twos)
takeStream :: Int → Stream a → [a]
takeStream 0 _ = []
takeStream (n + 1) (Cons (x, s)) = x : takeStream n s
  
```

Podemos usar esta função para verificar se *fair_merge* e *fair_merge'* têm o mesmo comportamento, mas é importante realçar que esta demonstração não é uma prova total.

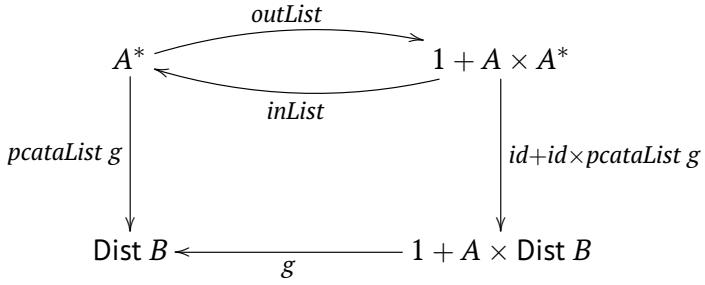
```

streamTest n = let
  input = i1 (ones, twos)
  l = takeStream n (fair_merge input)
  r = takeStream n (fair_merge' input)
  in l ≡ r
  
```

Problema 4

Para resolver este problema, iremos dividir a resolução em duas fases: (1) definir o operador *pcataList* e (2) definir *gene*. Podemos verificar que os tipos de *pcataList* e (\cdot) são semelhantes, e numa primeira

tentativa assumir que são iguais.



Analisando o diagrama verificamos que o tipo do gene g não encaixa no diagrama, pois o seu tipo é $g : 1 + A \times B \rightarrow \text{Dist } B$, logo não poderemos definir $pcataList$ como um catamorfismo de listas "normal". (É importante salientar que o diagrama apresentado é **inválido**, isto é, os tipos não são válidos.)

Embora o diagrama seja inválido, podemos tentar corrigi-lo através da definição de algumas funções, isto é, se formos capazes de definir uma composição de funções que satisfaça o tipo $1 + A \times \text{Dist } B \rightarrow \text{Dist } B$ e que envolva o gene g .

Sendo $\text{Dist } B$ um Monad, como podemos verificar na biblioteca [Probability](#) através da definição de `return` e de `>=`:

- $\text{return } x = D [(x, 1)]$
- $d \geqslant f = D [(y, q * p) \mid (x, p) \leftarrow \text{unD } d, (y, q) \leftarrow \text{unD } (f x)]$, e o gene $g : 1 + A \times B \rightarrow \text{Dist } B$

Podemos tentar definir uma composição monádica $g \bullet f \cdot F(\text{pcataList } g) \cdot \text{outList}$, com $f : 1 + A \times \text{Dist } B \rightarrow \text{Dist } (1 + A \times B)$. Conseguimos deduzir que f será uma alternativa, dado o seu tipo ser do género $A + B \rightarrow C$. O lado esquerdo da alternativa é simples de resolver, bastando efetuar uma injeção e colocar o resultado na "caixa" Dist , ou seja $f = [\text{return} \cdot i_1, \perp]$. Para o lado direito da alternativa teremos que emparelhar cada elemento b que está dentro do Monad $\text{Dist } B$ com o elemento A . Tal é possível através da notação `do` em Haskell, que nos permite "iterar" sobre os elementos de um Monad. Definimos então a função `attach` que pega num elemento puro e combina-o dentro de um Monad.

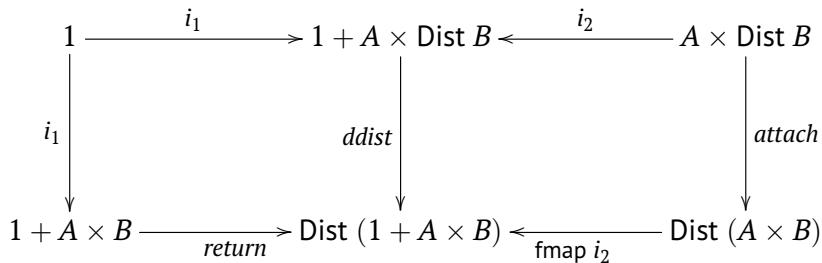
```

attach :: (Monad m) => (a, m b) -> m (a, b)
attach (b, x) = do { a <- x; return (b, a) }
  
```

Por fim basta injetar cada elemento de $\text{Dist}(A \times B)$ através de $\text{fmap } i_2$:

$$ddist = [\text{return} \cdot i_1, \text{fmap } i_2 \cdot attach]$$

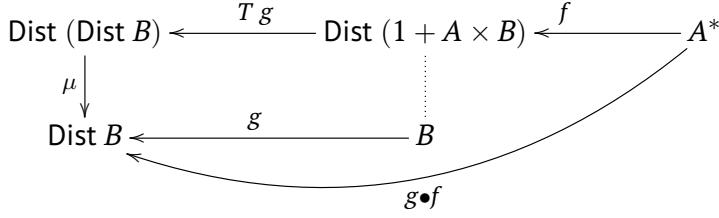
Podemos então apresentar o diagrama de $ddist$:



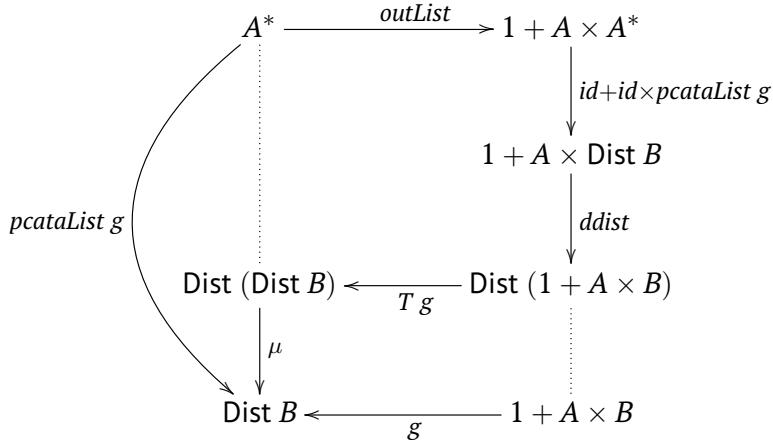
Por fim, podemos definir o operador $pcataList$ através da composição monádica descrita anteriormente:

$$pcataList g = g .! (ddist \cdot recList (pcataList g) \cdot \text{outList})$$

O diagrama da composição monádica é o seguinte (abreviando $f = ddist \cdot recList (pcataList g) \cdot outList$):



Apresentamos então o diagrama de *pacataList*:



Na segunda fase deste problema temos de definir *gene*, que simula o comportamento descrito no exemplo. Esta função será uma alternativa entre duas funções, uma delas que trabalha sobre o único habitante do tipo 1, que corresponde a uma lista vazia neste caso, ou seja, o fim da mensagem. A palavra "stop" deverá ser transmitida no fim de cada mensagem, com 10% de probabilidade de falhar, logo terá 90% de ser enviada. Seguindo a mesma estratégia da função apresentada *padd*, podemos definir *pstop* da seguinte forma:

$$pstop_ = D [(["stop"], 0.9), ([], 0.1)]$$

Sabemos ainda que a probabilidade de uma palavra ser enviada é de 95%, dado que a probabilidade de se perder é de 5%. Logo, considerando uma mensagem como uma lista de palavras não vazia, a função que descreve este comportamento é a seguinte:

$$plose (w, xs) = D [(w : xs, 0.95), (xs, 0.05)]$$

Definidas as funções que tratam do fim e do corpo de uma mensagem, podemos então definir *gene*, como a alternativa entre estas:

$$gene = [pstop, plose]$$

Através de *pcataList* e *gene*, podemos ver quais são as probabilidades associadas à transmissão da mensagem *words* "Vamos atacar hoje", através da execução de *transmitir*:

| | |
|-------------------------------------|-------|
| ["Vamos", "atacar", "hoje", "stop"] | 77.2% |
| ["Vamos", "atacar", "hoje"] | 8.6% |
| ["atacar", "hoje", "stop"] | 4.1% |
| ["Vamos", "atacar", "stop"] | 4.1% |

| | |
|---------------------------|------|
| ["Vamos", "hoje", "stop"] | 4.1% |
| ["Vamos", "atacar"] | 0.5% |
| ["Vamos", "hoje"] | 0.5% |
| ["atacar", "hoje"] | 0.5% |
| ["Vamos", "stop"] | 0.2% |
| ["atacar", "stop"] | 0.2% |
| ["hoje", "stop"] | 0.2% |
| ["atacar"] | 0.0% |
| ["Vamos"] | 0.0% |
| ["hoje"] | 0.0% |
| ["stop"] | 0.0% |
| [] | 0.0% |

Vamos então responder às perguntas:

- Qual a probabilidade de o resultado da transmissão ser ["Vamos", "hoje", "stop"]? Terá uma probabilidade de 4.1%.
- Qual a probabilidade de seguirem todas as palavras, mas faltar "stop" no fim? Terá uma probabilidade de 8.6%.
- Qual a probabilidade de o resultado da transmissão ser perfeito? Terá uma probabilidade de 77.2%.

De forma a garantir que estamos a lidar com uma distribuição válida, vamos recorrer às funções *checkD* e *sumP* definidas na biblioteca [Probability](#) que validam uma distribuição e determinam o somatório das probabilidades, respetivamente, para verificar a sua validade. Através da execução da seguinte equação:

$$checkedD = checkD(transmitir(words \text{ "Vamos atacar hoje"}))$$

Verificamos que *checkedD* não apresenta nenhuma mensagem de erro e tem o mesmo valor que *transmitir* (*words* "Vamos atacar hoje").

$$valueD = (sumP \cdot unD \cdot transmitir \cdot words) \text{ "Vamos atacar hoje"}$$

Provamos também que *valueD* = 1.0, logo estamos perante uma distribuição válida.

Index

\LaTeX , 5, 6
 bibtex, 6
 lhs2TeX, 5, 6
 makeindex, 6
 pdflatex, 5
 xymatrix, 7

Combinador “pointfree”

ana, 3
 cata
 Naturais, 7
 either, 3, 4
 split, 6

Cálculo de Programas, 1, 4

 Material Pedagógico, 5
 List.hs, 4

Docker, 5

container, 5, 6

Functor, 3, 7, 8

Função

π_1 , 7
 π_2 , 7

Haskell, 1, 5, 6

 Biblioteca

 PFP, 8

 Probability, 7, 8

 interpretador

 GHCi, 5–7

 Lazy evaluation, 3

 Literate Haskell, 5

Números naturais (\mathbb{N}), 7

Programação

 literária, 5, 6

References

- [1] D.E. Knuth. *Literate Programming*. CSLI Lecture Notes Number 27. Stanford University Center for the Study of Language and Information, Stanford, CA, USA, 1992.
- [2] Chris Okasaki. Breadth-first numbering: lessons from a small exercise in algorithm design. In Martin Odersky and Philip Wadler, editors, *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000*, pages 131–136. ACM, 2000.
- [3] J.N. Oliveira. Program Design by Calculation, 2024. Draft of textbook in preparation. First version: 1998. Current version: Sep. 2024. Informatics Department, University of Minho ([pdf](#)).