Universidade do Minho
Escola de Engenharia

Cálculo de Programas
Trabalho Prático (2024/25)

Lic. em Ciências da Computação

Grupo G99

axxxxxx	Nome
axxxxxx	Nome
axxxxxx	Nome

junho 2025

Preâmbulo

Na UC de 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.

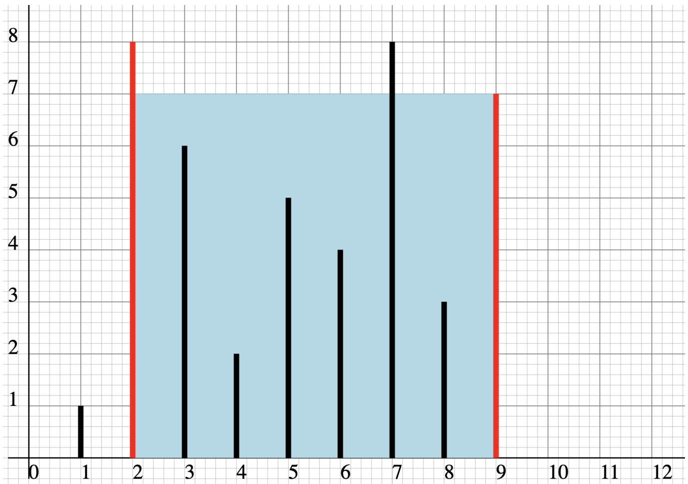
Na sequência pedagógica dos planos de estudo dos cursos que têm esta disciplina, opta-se pela aplicação deste método à programação em Haskell (sem prejuízo da sua aplicação a outras linguagens funcionais). Assim, o presente trabalho prático coloca os alunos perante problemas concretos que deverão ser 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.

Problema 1

Esta questão aborda um problema que é conhecido pela designação '*Container With Most Water*' e que se formula facilmente através do exemplo da figura seguinte:



A figura mostra a sequência de números

$$hghts = [1, 8, 6, 2, 5, 4, 8, 3, 7]$$

representada sob a forma de um histograma. O que se pretende é obter a maior área rectangular delimitada por duas barras do histograma, área essa marcada a azul na figura. (A "metáfora" *container with most water* sugere que as barras selecionadas delimitam um *container* com água.)

Pretende-se definida como um catamorfismo, anamorfismo ou hilomorfismo uma função em Haskell

$$mostwater :: [Int] \to Int$$

que deverá dar essa área. (No exemplo acima tem-se $mostwater\ [1, 8, 6, 2, 5, 4, 8, 3, 7] = 49$.) A resolução desta questão deverá ser acompanhada de diagramas elucidativos.

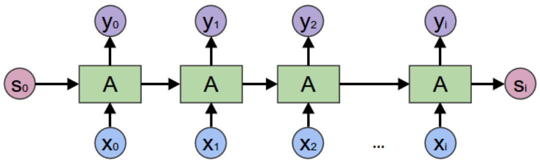


Figure 1: RNN vista como instância de um *accumulating map* [3].

Problema 2

Um dos problemas prementes da Computação na actualidade é conseguir, por engenharia reversa, interpretar as redes neuronais (RN) geradas artificialmente sob a forma de algoritmos compreensíveis por humanos.

Já foram dados passos que, nesse sentido, explicam vários padrões de RNs em termos de combinadores funcionais [3]. Em particular, já se mostrou como as RNNs (*Recurrent Neural Networks*) podem ser vistas como instâncias de *accumulating maps*, que em Haskell correspondem às funções *mapAccumR* e *mapAccumL*, conforme o sentido em que a acumulação se verifica (cf. figura 1).

A RNN que a figura 1 mostra diz-se *'one-to-one'* [1]. Há contudo padrões de RNNs mais gerais: por exemplo, o padrão *'many-to-one'* que se mostra na figura 2 extraída de [1].

Se *mapAccumR* e *mapAccumL* juntam *maps* com *folds*, pretendemos agora combinadores que a isso acrescentem *filter*, por forma a selecionar que etapas da computação geram ou não *outputs* — obtendo-se assim o efeito *'many-to-one'*. Ter-se-á, para esse efeito:

$$\begin{array}{l} mapAccumRfilter :: ((a,s) \to Bool) \to ((a,s) \to (c,s)) \to ([a],s) \to ([c],s) \\ mapAccumLfilter :: ((a,s) \to Bool) \to ((a,s) \to (c,s)) \to ([a],s) \to ([c],s) \end{array}$$

Pretende-se a implementação de *mapAccumRfilter* e *mapAccumLfilter* sob a forma de ana / cata ou hilomorfismos em Haskell, acompanhadas por diagramas.

Como caso de uso, sugere-se o que se dá no anexo F que, inspirado em [1], recorre à biblioteca `Data.Matrix`.

Problema 3

Umas das fórmulas conhecidas para calcular o número π é a que se segue,

$$\pi = \sum_{n=0}^{\infty} \frac{(n!)^2 2^{n+1}}{(2n+1)!} \tag{1}$$

correspondente à função π_{calc} cuja implementação em Haskell, paramétrica em n, é dada no anexo F.

Pretende-se uma implementação eficiente de (1) que, derivada por recursividade mútua, não calcule factoriais nenhuns:

$$\pi_{loop} = \cdots \cdot \text{for } loop\ inic \ \textbf{where} \ \cdots$$

Sugestão: recomenda-se a **regra prática** que se dá no anexo E para problemas deste género, que podem envolver várias decomposições por recursividade mútua em $\mathbb{N}_0$.

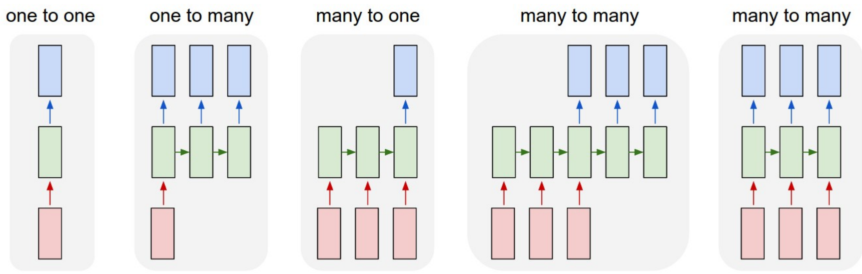


Figure 2: Várias tipologias de RNNs [1].

Problema 4

Considere-se a matriz e o vector que se seguem:

$$mat = \begin{bmatrix} 1 & -1 & 2 \\ 0 & -3 & 1 \end{bmatrix} \tag{2}$$

$$vec = \begin{bmatrix} 2 \\ 1 \\ 0 \end{bmatrix} \tag{3}$$

Em Haskell, podemos tornar explícito o espaço vectorial a que (3) pertence definindo-o da forma seguinte,

$$\begin{array}{l} vec :: Vec\ X \\ vec = V\ [(X_1, 2), (X_2, 1), (X_3, 0)] \end{array}$$

assumindo definido o tipo

$$\mathbf{data}\ Vec\ a = V\ \{outV :: [(a, Int)]\}\ \mathbf{deriving}\ (Ord)$$

e o "tipo-dimensão":

$$\mathbf{data}\ X = X_1 \mid X_2 \mid X_3\ \mathbf{deriving}\ (Eq, Show, Ord)$$

Da mesma forma que *tipamos* vec, também o podemos fazer para a matrix mat (2), cujas colunas podem ser indexadas por X também e as linhas por $Bool$, por exemplo:

$$\begin{array}{l} mat :: X \to Vec\ Bool \\ mat\ X_1 = V\ [(False, 1), (True, 0)] \\ mat\ X_2 = V\ [(False, -1), (True, -3)] \\ mat\ X_3 = V\ [(False, 2), (True, 1)] \end{array}$$

Quer dizer, matrizes podem ser encaradas como funções que dão vectores como resultado. Mais ainda, a multiplicação de mat por vec pode ser obtida correndo, simplesmente

$$vec' = vec \ggg mat$$

obtendo-se $vec' = V\ [(False, 1), (True, -3)]$ do tipo $Vec\ Bool$. Finalmente, se for dada a matrix

$$\begin{array}{l} neg :: Bool \to Vec\ Bool \\ neg\ False = V\ [(False, 0), (True, 1)] \\ neg\ True = V\ [(False, 1), (True, 0)] \end{array}$$

então a multiplicação de neg por mat mais não será que a matriz

$$neg \bullet mat$$

também do tipo $X \to Vec\ Bool$.

Obtém-se assim uma *álgebra linear tipada*. Contudo, para isso é preciso mostrar que Vec é um **mónade**, e é esse o tema desta questão, em duas partes:

- Instanciar Vec na class $Functor$ em Haskell:

  $$\begin{array}{l} \textbf{instance}\ Functor\ Vec\ \textbf{where} \\ \quad \mathsf{fmap}\ f = ... \end{array}$$

- Instanciar Vec na class $Monad$ em Haskell:

  $$\begin{array}{l} \textbf{instance}\ Monad\ Vec\ \textbf{where} \\ \quad x \ggg f = ... \\ \quad return\ a = ... \end{array}$$

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.

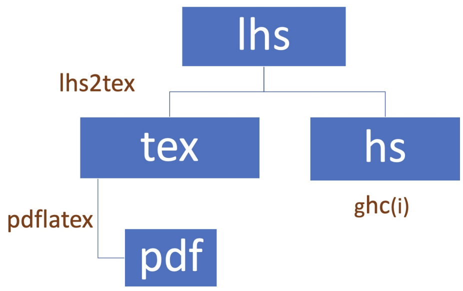
Para cumprir de forma integrada os objectivos do trabalho vamos recorrer a uma técnica de programação dita "literária" [2], cujo princípio base é o seguinte:

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 `cp2425t.pdf` que está a ler é já um exemplo de programação literária: foi gerado a partir do texto fonte `cp2425t.lhs`[1] que encontrará no material pedagógico desta disciplina descompactando o ficheiro `cp2425t.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:



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

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 `cp2425t.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 descarregar o referido zip com o código fonte do trabalho, basta executar os seguintes comandos:

```
$ docker build -t cp2425t .
$ docker run -v ${PWD}:/cp2425t -it cp2425t
```

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}:/cp2425t`) que permite que a diretoria em que se encontra na sua máquina local e a diretoria `/cp2425t` no container sejam partilhadas.

Pretende-se então que visualize/edite os ficheiros na sua máquina local e que os compile no container, executando:

```
$ lhs2TeX cp2425t.lhs > cp2425t.tex
$ pdflatex cp2425t
```

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 `cp2425t.lhs` é executável e contém o "kit" básico, escrito em Haskell, para realizar o trabalho. Basta executar

```
$ ghci cp2425t.lhs
```

Abra o ficheiro `cp2425t.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 cp2425t.aux
$ makeindex cp2425t.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 ler[1] onde se obtém o efeito seguinte:[2]

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

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

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

E Regra prática para a recursividade mútua em $\mathbb{N}_0$

Nesta disciplina estudou-se como fazer programação dinâmica por cálculo, recorrendo à lei de recursividade mútua.[3]

Para o caso de funções sobre os números naturais ($\mathbb{N}_0$, com functor $\mathsf{F}\ X = 1 + X$) pode derivar-se da lei que foi estudada uma *regra de algibeira* que se pode ensinar a programadores que não tenham estudado Cálculo de Programas. Apresenta-se de seguida essa regra, tomando como exemplo o cálculo do ciclo-for que implementa a função de Fibonacci, recordar o sistema:

[1] Procure e.g. por `"sec:diagramas"`.

[2] Exemplos tirados de [4].

[3] Lei (3.95) em [4], página 110.

$$\begin{array}{l} \mathit{fib}\ 0 = 1 \\ \mathit{fib}\ (n+1) = f\ n \end{array}$$

$$\begin{array}{l} f\ 0 = 1 \\ f\ (n+1) = \mathit{fib}\ n + f\ n \end{array}$$

Obter-se-á de imediato

$$\begin{array}{l} \mathit{fib}' = \pi_1 \cdot \mathsf{for}\ \mathit{loop}\ \mathit{init}\ \textbf{where} \\ \quad \mathit{loop}\ (\mathit{fib}, f) = (f, \mathit{fib} + f) \\ \quad \mathit{init} = (1, 1) \end{array}$$

usando as regras seguintes:

- 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.[1]
- Para os resultados vão-se buscar as expressões respectivas, retirando a variável n.
- Em *init* coleccionam-se os resultados dos casos de base das funções, pela mesma ordem.

Mais um exemplo, envolvendo polinómios do segundo grau $ax^2 + bx + c$ em $\mathbb{N}_0$. Seguindo o método estudado nas aulas[2], de $f\ x = ax^2 + bx + c$ derivam-se duas funções mutuamente recursivas:

$$\begin{array}{l} f\ 0 = c \\ f\ (n+1) = f\ n + k\ n \end{array}$$

$$\begin{array}{l} k\ 0 = a + b \\ k\ (n+1) = k\ n + 2\ a \end{array}$$

Seguindo a regra acima, calcula-se de imediato a seguinte implementação, em Haskell:

$$\begin{array}{l} f'\ a\ b\ c = \pi_1 \cdot \mathsf{for}\ \mathit{loop}\ \mathit{init}\ \textbf{where} \\ \quad \mathit{loop}\ (f, k) = (f + k, k + 2 * a) \\ \quad \mathit{init} = (c, a + b) \end{array}$$

F Código fornecido

Problema 1

Teste relativo à figura da página 1:

$$\mathit{test}_1 = \mathit{mostwater}\ \mathit{hghts}$$

Problema 2

Testes relativos a *mapAccumLfilter* e *mapAccumRfilter* em geral (comparar os *outputs*)

[1] Podem obviamente usar-se outros símbolos, mas numa primeira leitura dá jeito usarem-se tais nomes.

[2] Secção 3.17 de [4] e tópico Recursividade mútua nos vídeos de apoio às aulas teóricas.

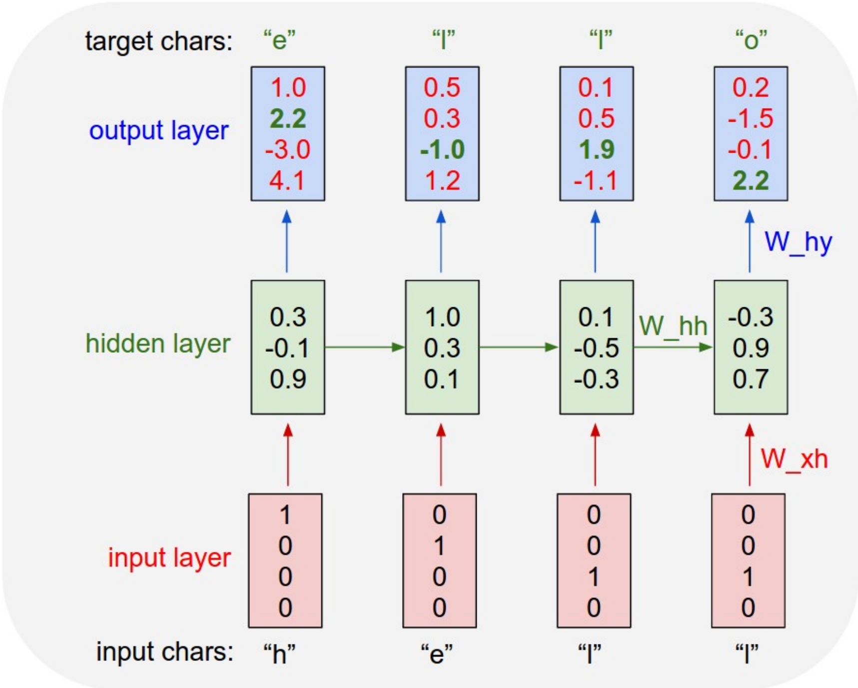


Figure 3: Exemplo *char seq* extraído de [1].

$$test_{2a} = mapAccumLfilter\ ((>10) \cdot \pi_1)\ f\ (odds\ 12, 0)$$
$$test_{2b} = mapAccumRfilter\ ((>10) \cdot \pi_1)\ f\ (odds\ 12, 0)$$

onde:

$$odds\ n = \mathsf{map}\ \ ((1+) \cdot (2*))\ [0\,..\,n-1]$$
$$f\ (a, s) = (s, a + s)$$

Teste

$$test_{2c} = mapAccumLfilter\ true\ step\ ([x_1, x_2, x_3, x_4], h_0)$$

baseado no exemplo de Karpathy [1] que a figura 3 mostra, usando os dados seguintes:

- Estado inicial:

$$h_0 = fromList\ 3\ 1\ [1.0, 1.0, 1, 0]$$

- *Step function*:

$$step\ (x, h) = (\alpha\ (wy * h), \alpha\ (wh * h + wx * x))$$

- Função de activação:

$$\alpha = \mathsf{fmap}\ \sigma\ \mathbf{where}\ \sigma\ x = (tanh\ x + 1)\ /\ 2$$

- *Input layer*:

$$inp = [x_1, x_2, x_3, x_4]$$
$$x_1 = fromList\ 4\ 1\ [1.0, 0, 0, 0]$$
$$x_2 = fromList\ 4\ 1\ [0, 1.0, 0, 0]$$
$$x_3 = fromList\ 4\ 1\ [0, 0, 1.0, 0]$$
$$x_4 = x_3$$

- Matrizes exemplo:

$$
\begin{aligned}
&wh = \mathit{fromList}\ 3\ 3\ [0.4, -0.2, 1.6, -3.1, 1.4, 0.1, 5.4, -2.7, 0.1]\\
&wy = \mathit{fromList}\ 4\ 3\ [2.1, 1.1, 0.8, 1.3, -6.4, -3.4, -2.7, -3.8, -1.3, -0.5, -0.9, -0.4]\\
&wx = \mathit{fromLists}\ [[0.0, -51.9, 0.0, 0.0], [0.0, 26.6, 0.0, 0.0], [-16.7, -5.5, -0.1, 0.1]]
\end{aligned}
$$

NB: Podem ser definidos e usados outros dados em função das experiências que se queiram fazer.

Problema 3

Fórmula (1) em Haskell:

$$
\begin{aligned}
&\pi_{calc}\ n = (\mathit{sum} \cdot \mathsf{map}\ f)\ [0\,..\,n]\ \mathbf{where}\\
&\quad f\ n = \mathit{fromIntegral}\ (n! * n! * (g\ n))\ /\ \mathit{fromIntegral}\ (d\ n)\\
&\quad g\ n = 2 \uparrow (n+1)\\
&\quad d\ n = (2 * n + 1)!
\end{aligned}
$$

Problema 4

Se pedirmos ao GHCi que nos mostre o vector *vec* obteremos:

```
{ X1 |-> 2 , X2 |-> 1 }
```

Este resultado aparece mediante a seguinte instância de *Vec* na classe *Show*:

$$
\begin{aligned}
&\mathbf{instance}\ (\mathit{Show}\ a, \mathit{Ord}\ a, \mathit{Eq}\ a) \Rightarrow \mathit{Show}\ (\mathit{Vec}\ a)\ \mathbf{where}\\
&\quad \mathit{show} = \mathit{showbag} \cdot \mathit{consol} \cdot \mathit{outV}\ \mathbf{where}\\
&\qquad \mathit{showbag} = \mathit{concat}\ \cdot\\
&\qquad\quad (+\!\!+[\texttt{"\ \}"}]) \cdot (\texttt{"\{\ "}:)\ \cdot\\
&\qquad\quad (\mathit{intersperse}\ \texttt{"\ ,\ "})\ \cdot\\
&\qquad\quad \mathit{sort}\ \cdot\\
&\qquad\quad (\mathsf{map}\ f)\ \mathbf{where}\ f\ (a, b) = (\mathit{show}\ a) +\!\!+ \texttt{"\ |->\ "} +\!\!+ (\mathit{show}\ b)
\end{aligned}
$$

Outros detalhes da implementação de *Vec* em Haskell:

$$
\begin{aligned}
&\mathbf{instance}\ \mathit{Applicative}\ \mathit{Vec}\ \mathbf{where}\\
&\quad \mathit{pure} = \mathit{return}\\
&\quad (<\!*\!>) = \mathit{aap}\\
&\mathbf{instance}\ (\mathit{Eq}\ a) \Rightarrow \mathit{Eq}\ (\mathit{Vec}\ a)\ \mathbf{where}\\
&\quad b \equiv b' = (\mathit{outV}\ b)\ `\mathit{lequal}`\ (\mathit{outV}\ b')\\
&\qquad \mathbf{where}\ \mathit{lequal}\ a\ b = \mathit{isempty}\ (a \ominus b)\\
&\qquad\quad a \ominus b = a +\!\!+ \overline{b}\\
&\qquad\quad \overline{x} = [(k, -i) \mid (k, i) \leftarrow x]
\end{aligned}
$$

Funções auxiliares:

$$
\begin{aligned}
&\mathit{consol} :: (\mathit{Eq}\ b) \Rightarrow [(b, \mathit{Int})] \rightarrow [(b, \mathit{Int})]\\
&\mathit{consol} = \mathit{filter}\ \mathit{nzero} \cdot \mathsf{map}\ (\mathit{id} \times \mathit{sum}) \cdot \mathit{col}\ \mathbf{where}\ \mathit{nzero}\ (\_, x) = x \not\equiv 0\\
&\mathit{isempty} :: \mathit{Eq}\ a \Rightarrow [(a, \mathit{Int})] \rightarrow \mathit{Bool}\\
&\mathit{isempty} = \mathit{all}\ (\equiv 0) \cdot \mathsf{map}\ \pi_2 \cdot \mathit{consol}\\
&\mathit{col} :: (\mathit{Eq}\ a, \mathit{Eq}\ b) \Rightarrow [(a, b)] \rightarrow [(a, [b])]\\
&\mathit{col}\ x = \mathit{nub}\ [k \mapsto [d' \mid (k', d') \leftarrow x, k' \equiv k] \mid (k, d) \leftarrow x]\ \mathbf{where}\ a \mapsto b = (a, b)
\end{aligned}
$$

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

$$mostwater = \perp$$

Problema 2

$$\begin{array}{l} mapAccumRfilter\ p\ f = \perp \\ mapAccumLfilter\ p\ f = \perp \end{array}$$

Problema 3

Reparemos que

$$\pi_n = \sum_{i=0}^{n} \frac{(i!)^2 2^{i+1}}{(2i+1)!} = 2 \sum_{i=0}^{n} \frac{(i!) \times (i!) 2^i}{(2i+1)!} = 2 \sum_{i=0}^{n} \frac{i! \times ((\cancel{2i}) \times \cancel{(2(i-1))} \times \cdots \times 2 \cdot 2 \times 2 \cdot 1)}{(2i+1) \times (\cancel{2i}) \times (2i-1) \times \cancel{(2(i-1))} \times \cdots \times 2 \times 1} =$$

$$= 2 \sum_{i=0}^{n} \frac{i!}{(2i+1)!!}$$

onde $n!!$ é o fatorial duplo.

Seja $f(n) = \sum_{i=0}^{n} \frac{i!}{(2i+1)!!} = 1 + \sum_{i=0}^{n-1} \frac{(i+1)!}{(2i+3)!!}$, $g(n) = \frac{(n+1)!}{(2n+3)!!}$ e $h(n) = \frac{n+2}{2n+5}$. É fácil reparar que

$$f(n) = 1 + \sum_{i=0}^{n-1} \frac{(i+1)!}{(2i+3)!!} = 1 + \sum_{i=0}^{n-1} g(i) \implies \begin{cases} f(0) = 1 \\ f(n+1) = f(n) + g(n) \end{cases}$$

$$g(n) = \frac{(i+1)!}{(2i+3)!!} = \frac{1}{3} \times \prod_{n=0}^{n-1} h(i) \implies \begin{cases} g(0) = \frac{1}{3} \\ g(n+1) = g(n) \times h(n) \end{cases}$$

e

$$h(n) = \frac{n+2}{2n+5} \implies \begin{cases} h(0) = \frac{2}{5} \\ h(n+1) = \frac{n+3}{2n+7} = \frac{\frac{n+2}{2n+5} - 1}{4\frac{n+2}{2n+5} - 3} = \frac{h(n)-1}{4h(n)-3} \end{cases}$$

Ao escrever as funções na forma *point-free* e recorrendo ás regras de cálculo usuais, tem-se

$$
\begin{aligned}
&f \cdot \mathsf{in} = [\underline{1}, add] \cdot (1 + \langle f, g \rangle) \\
&g \cdot \mathsf{in} = [\underline{1 / 3}, mul] \cdot (1 + \langle g, h \rangle) \\
&h \cdot \mathsf{in} = [\underline{2 / 5}, calc] \cdot (1 + h) \ \textbf{where} \\
&\quad calc\ n = (n - 1) / (4 * n - 3) \\
&\quad add\ (x, y) = x + y \\
&\quad mul\ (x, y) = x * y
\end{aligned}
$$

Em que $\mathbf{in} = [\underline{0}, suc]$.

Recorrendo à lei de absorção $+$ e $\times$ e com auxilío da função assocl, temos

$$
\begin{aligned}
&f \cdot \mathsf{in} = [\underline{1}, add \cdot \pi_1 \cdot assocl] \cdot (1 + \langle f, \langle g, h \rangle \rangle) \\
&g \cdot \mathsf{in} = [\underline{1 / 3}, mul \cdot \pi_2] \cdot (1 + \langle f, \langle g, h \rangle \rangle) \\
&h \cdot \mathsf{in} = [\underline{2 / 5}, calc \cdot \pi_2 \cdot \pi_2] \cdot (1 + \langle f, \langle g, h \rangle \rangle) \ \textbf{where} \\
&\quad calc\ n = (n - 1) / (4 * n - 3) \\
&\quad add\ (x, y) = x + y \\
&\quad mul\ (x, y) = x * y
\end{aligned}
$$

Podemos unir as equações numa só através do Eq-$\times$ e Fusão-$\times$. Portanto, unindo as duas últimas equações e em seguida aplicando a lei da troca, temos

$$
\begin{aligned}
&f \cdot \mathsf{in} = [\underline{1}, add \cdot \pi_1 \cdot assocl] \cdot (1 + \langle f, \langle g, h \rangle \rangle) \\
&\langle g, h \rangle \cdot \mathsf{in} = [\langle \underline{1 / 3}, \underline{2 / 5} \rangle, \langle mul \cdot \pi_2, calc \cdot \pi_2 \cdot \pi_2 \rangle] \cdot (1 + \langle f, \langle g, h \rangle \rangle) \ \textbf{where} \\
&\quad calc\ n = (n - 1) / (4 * n - 3) \\
&\quad add\ (x, y) = x + y \\
&\quad mul\ (x, y) = x * y
\end{aligned}
$$

Repetindo o mesmo raciocínio de forma análoga, tem-se por fim

$$
\begin{aligned}
&\langle f, \langle g, h \rangle \rangle \cdot \mathsf{in} = [\underline{(1, (1 / 3, 2 / 5))}, \langle add \cdot \pi_1 \cdot assocl, \langle mul \cdot \pi_2, calc \cdot \pi_2 \cdot \pi_2 \rangle \rangle] \cdot (1 + \langle f, \langle g, t \rangle \rangle) \ \textbf{where} \\
&\quad calc\ n = (n - 1) / (4 * n - 3) \\
&\quad add\ (x, y) = x + y \\
&\quad mul\ (x, y) = x * y
\end{aligned}
$$

Ora, pela lei de universal-cata, tem-se que a função $\langle f, \langle g, h \rangle \rangle$ é da forma for *loop inic*. Portanto, se $worker = \langle f, \langle g, h \rangle \rangle$, tem-se

$$
\begin{aligned}
&worker = \mathsf{for}\ loop\ inic \\
&loop = \langle add \cdot \pi_1 \cdot assocl, \langle mul \cdot \pi_2, calc \cdot \pi_2 \cdot \pi_2 \rangle \rangle \\
&inic = \underline{(1, (1 / 3, 2 / 5))} \ \textbf{where} \\
&\quad calc\ n = (n - 1) / (4 * n - 3) \\
&\quad add\ (x, y) = x + y \\
&\quad mul\ (x, y) = x * y
\end{aligned}
$$

Finalmente, como $\pi_{loop} = (2*) \cdot \pi_1 \cdot \langle f, \langle g, h \rangle \rangle$, temos

$$
\begin{aligned}
&\pi_{loop} = wrapper \cdot worker \\
&worker = \mathsf{for}\ loop\ inic \\
&loop = \langle add \cdot \pi_1 \cdot assocl, \langle mul \cdot \pi_2, calc \cdot \pi_2 \cdot \pi_2 \rangle \rangle \ \textbf{where} \\
&\quad calc\ n = (n - 1) / (4 * n - 3)
\end{aligned}
$$

$$
\begin{array}{l}
\quad add\ (x, y) = x + y \\
\quad mul\ (x, y) = x * y \\
inic = \underline{(1, (1\ /\ 3, 2\ /\ 5))} \\
wrapper = (2*) \cdot \pi_1
\end{array}
$$

Problema 4

Para fazer o funtor, vamos explorar melhor o in e o out do Vec.

$V :: [(a, Int)] \rightarrow Vec\ a$

$outV :: Vec \rightarrow [(a, Int)]$

Como o $outV$ e V usam lista como input e output, podemos usar funções de listas para auxiliar nas nossas funções de Vec.

Functor:

fmap :: $(a \rightarrow b) \rightarrow Vec\ a \rightarrow Vec\ b$

Utilizando o $outV$ e o map , podemos definir o seguinte diagrama:

$$
\begin{tikzcd}
Vec\ A \arrow[d, "outV"'] \\
(A \times Int)^* \arrow[d, "\mathsf{map}\ \ (f \times id)"'] \\
(B \times Int)^* \arrow[d, "V"'] \\
Vec\ B
\end{tikzcd}
$$

O que nos permite definir o fmap assim:

$$
\begin{array}{l}
\textbf{instance}\ Functor\ Vec\ \textbf{where} \\
\quad \mathsf{fmap}\ f = V \cdot (\mathsf{map}\ \ (f \times id)) \cdot outV
\end{array}
$$

Monad:

Para o monad, vamos definir o μ (miu) e o υ ($return$) para facilitar na definição de outras funções

$return :: a \rightarrow Vec\ a$

$return\ a = V\ [(a, 1)]$

para qualquer a, fazemos um *singleton*, associado com 1, porque é o elemento neutro da multiplicação.

$miu :: Vec\ (Vec\ a) \rightarrow Vec\ a$

ou seja

$$\begin{array}{c}
Vec\ (Vec\ A) \\
\Big\downarrow{\scriptstyle outV} \\
(Vec\ A \times Int)^{*} \\
\Big\downarrow{\scriptstyle \mathsf{map}\ (outV \cdot \widehat{mulV})} \\
(A \times Int)^{**} \\
\Big\downarrow{\scriptstyle concat} \\
A^{*} \\
\Big\downarrow{\scriptstyle V} \\
Vec\ A
\end{array}$$

sendo o $mulV$ o produto escalar de vetores.

$$mulV :: Vec\ a \to Int \to Vec\ a$$
$$mulV\ v\ x = V\ (\mathsf{map}\ \ (id \times (x*))\ (outV\ v))$$

e assim definimos:

$$miu = V \cdot concat \cdot \mathsf{map}\ \ (outV \cdot \widehat{mulV}) \cdot outV$$

falta apenas definir $(\gg\!=) :: Vec\ a \to (a \to Vec\ b) \to Vec\ b$, com o miu e fmap definido, fica simples definir:

$$\begin{array}{c}
Vec\ A \\
\Big\downarrow{\scriptstyle \mathsf{fmap}\ f} \\
Vec\ (Vec\ B) \\
\Big\downarrow{\scriptstyle miu} \\
Vec\ B
\end{array}$$

E assim concluimos que

$$\textbf{instance}\ Monad\ Vec\ \textbf{where}$$
$$\quad x \gg\!= f = miu\ (\mathsf{fmap}\ f\ x)$$
$$\quad return\ a = V\ [(a, 0)]$$

Index

References

[1] A. Karpathy. The unreasonable effectiveness of recurrent neural networks, 2015. Blog: http://karpathy.github.io/2015/05/21/rnn-effectiveness, last read: June 11, 2025.

[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] C. Olah. Neural networks, types, and functional programming, 2015. Blog: http://colah.github.io/posts/2015-09-NN-Types-FP/, last read: June 11, 2025.

[4] 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).