Universidade do Minho
Escola de Engenharia

Cálculo de Programas
Trabalho Prático (2024/25)

Lic. em Ciências da Computação

Grupo G02

a108395 José Pedro Flores Novais
a108840 Guilherme Dall'Agnol
a108653 João Manuel Pinto Alves

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.

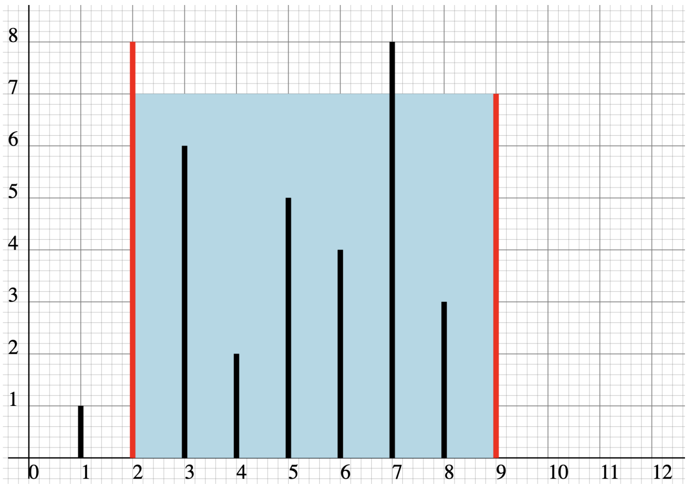
Na sequência pedagógica dos planos de estudo dos cursos que têm esta disciplina, opta-se pela aplicação deste método à programação em Haskell (sem prejuízo da sua aplicação a outras linguagens funcionais). Assim, o presente trabalho prático coloca os alunos perante problemas concretos que deverão ser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

$$mostwater :: [Int] \rightarrow Int$$

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.

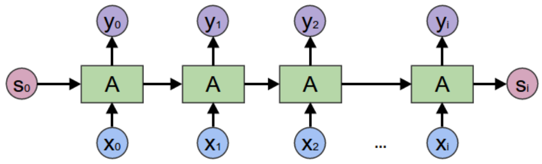


Figure 1: RNN vista como instância de um *accumulating map* [3].

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:

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

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 $\mathbb{N}_0$.

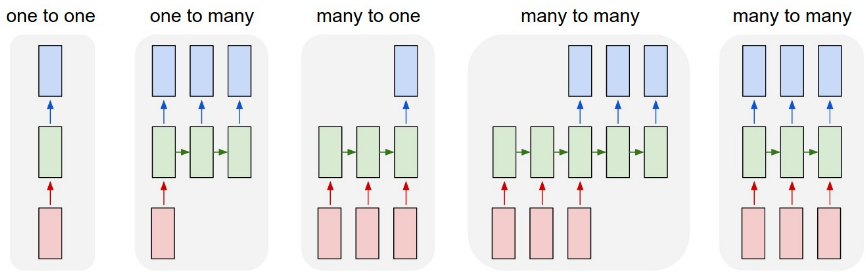


Figure 2: Várias tipologias de RNNs [1].

Problema 4

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

$$mat = \begin{bmatrix} 1 & -1 & 2 \\ 0 & -3 & 1 \end{bmatrix} \quad (2)$$

$$vec = \begin{bmatrix} 2 \\ 1 \\ 0 \end{bmatrix} \quad (3)$$

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

$$vec :: Vec\ X$$
$$vec = V\ [(X_1, 2), (X_2, 1), (X_3, 0)]$$

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:

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

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

$$neg :: Bool \to Vec\ Bool$$
$$neg\ False = V\ [(False, 0), (True, 1)]$$
$$neg\ True = V\ [(False, 1), (True, 0)]$$

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:

  instance $Functor\ Vec$ **where**
  fmap $f = ...$

- Instanciar Vec na class $Monad$ em Haskell:

  instance $Monad\ Vec$ **where**
  $x \ggg f = ...$
  $return\ a = ...$

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.

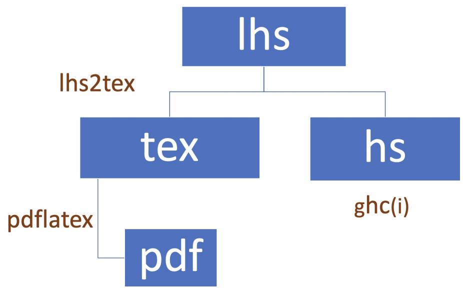
Para cumprir de forma integrada os objectivos do trabalho vamos recorrer a uma técnica de programação dita "literária" [2], cujo princípio base é o seguinte:

> *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.

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

Obter-se-á de imediato

$$
\begin{aligned}
&\mathit{fib}' = \pi_1 \cdot \text{for}\ \mathit{loop}\ \mathit{init}\ \mathbf{where}\\
&\quad \mathit{loop}\ (\mathit{fib}, f) = (f, \mathit{fib} + f)\\
&\quad \mathit{init} = (1, 1)
\end{aligned}
$$

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:

$$
\begin{aligned}
&f\ 0 = c\\
&f\ (n+1) = f\ n + k\ n\\
&k\ 0 = a + b\\
&k\ (n+1) = k\ n + 2\ a
\end{aligned}
$$

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

$$
\begin{aligned}
&f'\ a\ b\ c = \pi_1 \cdot \text{for}\ \mathit{loop}\ \mathit{init}\ \mathbf{where}\\
&\quad \mathit{loop}\ (f, k) = (f + k, k + 2 * a)\\
&\quad \mathit{init} = (c, a + b)
\end{aligned}
$$

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.

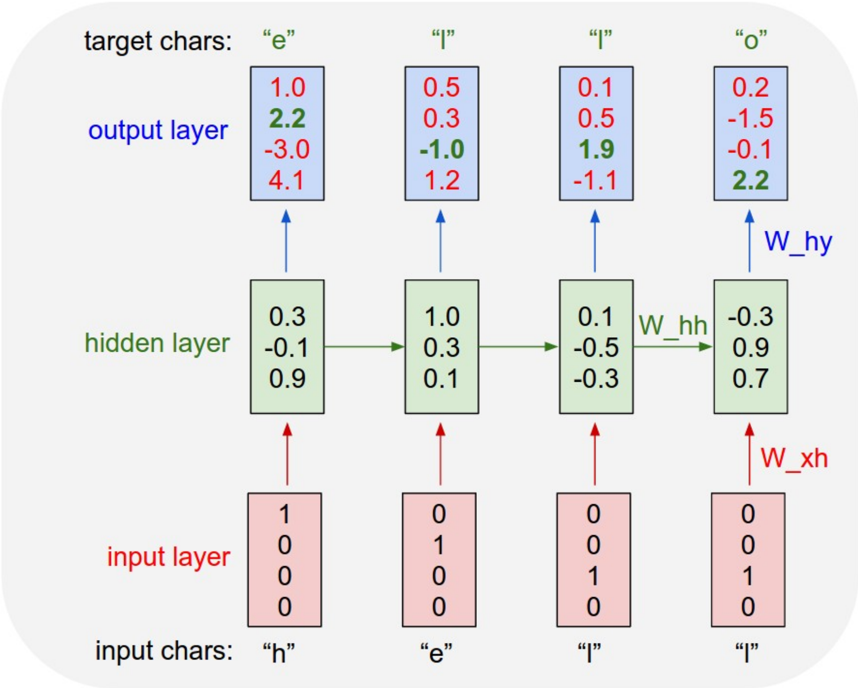


Figure 3: Exemplo *char seq* extraído de [1].

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

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:

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

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

Problema 3

Fórmula (1) em Haskell:

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

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*:

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

Outros detalhes da implementação de *Vec* em Haskell:

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

Funções auxiliares:

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

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

O problema 1 foi resolvido usando um hilomorfismo que consiste num anamorfismo que "faz pouco" e um catarmorfismo que "faz muito" ou seja um Easy Split/Hard Join.

começamos com o anamorfismo:

$$
\begin{aligned}
&\mathit{mysuffixes} :: [a] \to [[a]]\\
&\mathit{mysuffixes} = [\!(\, (id + \langle \mathit{cons}, \pi_2 \rangle) \cdot \mathit{outList} \,)\!]
\end{aligned}
$$

Agora podemos começar com a cabeça de cada sub-lista e fazer uma procura exaustiva pela lista de listas

$$
\begin{tikzcd}
\mathbb{N}_0^* \arrow[r, "mysuffixes"] & (\mathbb{N}_0^*)^* \arrow[r, "\mathsf{map}\ \langle head, tail \rangle"'] & \mathbb{N}_0 \times \mathbb{N}_0^*
\end{tikzcd}
$$

No final ao fazer a procura e o calculo todo em todas as sublistas fazemos o catamorfismo mymaximum que retira o maior elemento de uma lista

$$
\begin{aligned}
&\mathit{mymaximum} :: [\mathit{Int}] \to \mathit{Int}\\
&\mathit{mymaximum} = (\!|\, [\underline{0}, \widehat{max}] \,|\!)
\end{aligned}
$$

que de todas as sublistas calculadas a área maxima temos uma lista de áreas maximas então calculamos o maximo

$$
\begin{tikzcd}
\mathbb{N}_0^* \arrow[r, "mymaximum"] & \mathbb{N}_0
\end{tikzcd}
$$

Agora precisamos de uma função *area* que a partir de um par head e tail faça o calculo tendo em conta sempre a cabeça

$$
\mathit{area} :: (\mathit{Int}, [\mathit{Int}]) \to \mathit{Int}
$$

$$
\begin{tikzcd}
\mathbb{N}_0 \times \mathbb{N}_0^* \arrow[rr, "area"] \arrow[d, "id \times ((\mathsf{map}\ swap) \cdot zip\ [1..])"'] & & \mathbb{N}_0 \\
\mathbb{N}_0 \times (\mathbb{N}_0 \times \mathbb{N}_0)^* \arrow[d, "\langle id \times len, \pi_2 \rangle"'] & & \\
(\mathbb{N}_0 \times \mathbb{N}_0) \times (\mathbb{N}_0 \times \mathbb{N}_0)^* \arrow[d, "(\widehat{replicate} \cdot swap) \times id"'] & & \\
\mathbb{N}_0^* \times (\mathbb{N}_0 \times \mathbb{N}_0)^* \arrow[r, "\widehat{zip}"'] & (\mathbb{N}_0 \times (\mathbb{N}_0 \times \mathbb{N}_0))^* \arrow[r, "\mathsf{map}\ auxarea"'] & \mathbb{N}_0^* \arrow[uuu, "mymaximum"']
\end{tikzcd}
$$

Onde temos como função auxiliar *auxarea* que faz o calculo e a alternativa qual dos extremos é o menor pra calcular a área.

$$auxarea :: (Int, (Int, Int)) \rightarrow Int$$

Quero lembrar que o do par $\mathbb{N}_0 \times (\mathbb{N}_0 \times \mathbb{N}_0)$ a **primeira componente** é a head ou seja uma das alturas que estamos a calcular, a **segunda componente** é a segunda altura e por fim a **ultima componente** é a respetiva distância entre as duas alturas. Agora basta verificar qual das alturas é a menor e multiplicar pela distância.

$$\begin{tikzcd}
\mathbb{N}_0 \times (\mathbb{N}_0 \times \mathbb{N}_0) \arrow[r, "assocl"] \arrow[d, "auxarea"'] & (\mathbb{N}_0 \times \mathbb{N}_0) \times \mathbb{N}_0 \arrow[r, "(grd\,\widehat{(>)})\times id"] & ((\mathbb{N}_0 \times \mathbb{N}_0) + (\mathbb{N}_0 \times \mathbb{N}_0)) \times \mathbb{N}_0 \arrow[dl, "{[\pi_2,\pi_1]\times id}"] \\
\mathbb{N}_0 & \mathbb{N}_0 \times \mathbb{N}_0 \arrow[l, "\widehat{(*)}"] &
\end{tikzcd}$$

Agora podemos exprimir *area* e *auxarea* em Haskell como:

$$auxarea = \widehat{(*)} \cdot ([\pi_2, \pi_1] \times id) \cdot (grd\ \widehat{(>)} \times id) \cdot assocl$$
$$area = mymaximum \cdot \mathsf{map}\ \ auxarea \cdot \widehat{zip} \cdot (\widehat{replicate} \cdot swap \times id) \cdot \langle id \times length, \pi_2 \rangle \cdot (id \times \mathsf{map}\ \ swap \cdot zip\ [1\,..])$$

Finalmente temos uma definição para o mostwater que pode ser expressa neste diagrama:

$$\begin{tikzcd}
\mathbb{N}_0^* \arrow[d, "mysuffixes"'] \arrow[ddrr, "mostwater"] & & \\
(\mathbb{N}_0^*)^* \arrow[d, "\mathsf{map}\ \ \langle head, tail\rangle"'] & & \\
(\mathbb{N}_0 \times \mathbb{N}_0^*)^* \arrow[r, "\mathsf{map}\ \ area"'] & \mathbb{N}_0^* \arrow[r, "mymaximum"'] & \mathbb{N}_0
\end{tikzcd}$$

Claro que para transformar num hylomorfismo de listas podemos utilizar algumas leis do Cálculo de Programas nomeadamente a *Composição de maps* e a *Absorção-Cata*

$$\begin{aligned}
& mostwater = mymaximum \cdot \mathsf{map}\ \ area \cdot \mathsf{map}\ \ \langle head, tail \rangle \cdot mysuffixes \\
\equiv \quad & \{ \text{ Functor-F } \} \\
& mostwater = mymaximum \cdot \mathsf{map}\ \ (area \cdot \langle head, tail \rangle) \cdot mysuffixes \\
\equiv \quad & \{ \text{ mymaximum-definition } \} \\
& mostwater = \llparenthesis\, [\underline{0}, \widehat{max}]\, \rrparenthesis \cdot \mathsf{map}\ \ (area \cdot \langle head, tail \rangle) \cdot mysuffixes \\
\equiv \quad & \{ \text{ Absorção-cata } \} \\
& mostwater = \llparenthesis\, [\underline{0}, \widehat{max}] \cdot (1 + ((area \cdot \langle head, tail \rangle) \times id))\, \rrparenthesis \cdot mysuffixes \\
\equiv \quad & \{ \text{ Absorção-+, mysuffixes-definition } \} \\
& mostwater = \llparenthesis\, [\underline{0}, \widehat{max} \cdot ((area \cdot \langle head, tail \rangle) \times id)]\, \rrparenthesis \cdot [\!(\, (id + \langle cons, \pi_2 \rangle) \cdot outList \,)\!] \\
\Box \quad &
\end{aligned}$$

Temos a definição de mostwater como um hylomorfismo:

$$mostwater = hyloList\ [\underline{0}, \widehat{max} \cdot ((area \cdot \langle head, tail \rangle) \times id)]\ ((id + \langle cons, \pi_2 \rangle) \cdot outList)$$

Podemos agora exprimir o mostwater no seu diagrama do hylomorfismo deixando claro o seu *divide* e o seu *conquer* também expondo a sua estrutura intermediária:

$$
\begin{tikzcd}
\mathbb{N}_0^* \arrow[r, "out_*"] \arrow[d, "divide"'] & 1 + \mathbb{N}_0 \times \mathbb{N}_0^* \arrow[r, "(id + \langle cons, \pi_2 \rangle)"] & 1 + \mathbb{N}_0^* \times \mathbb{N}_0^* \arrow[d, "id + id \times divide"] \\
(\mathbb{N}_0^*)^* \arrow[rr, bend left=10, "out_{**}"] \arrow[d, "conquer"'] & & 1 + \mathbb{N}_0^* \times (\mathbb{N}_0^*)^* \arrow[ll, bend left=10, "in_{**}"] \arrow[d, "id + id \times conquer"] \\
\mathbb{N}_0 & & 1 + \mathbb{N}_0^* \times \mathbb{N}_0 \arrow[ll, "{[\underline{0}, \widehat{max} \cdot ((area \cdot \langle head, tail \rangle) \times id)]}"]
\end{tikzcd}
$$

Problema 2

Pra resolver este problema foi criado uma estrutura para facilitar a visualização da resolução.

$$
\begin{tikzcd}
A^* \times S \arrow[r, bend left=10, "outLP"] & 1 \times S + A^* \times (A^* \times S) \arrow[l, bend left=10, "inLP"]
\end{tikzcd}
$$

Tal que o inLP é definido como a construção de uma lista mantendo o valor da segunda componente:

$$inLP = [nil \times id, (\widehat{(+\!\!+)} \times id) \cdot assocl]$$

E o outLP é definido como uma alternativa entre a primeira componente ser a lista vazia ou não. Assim colocando o unico valor numa lista (para efeitos futuros)

$$
\begin{array}{l}
outLP\ ([\,], s) = i_1\ ((), s) \\
outLP\ (h : t, s) = i_2\ (singl\ h, (t, s))
\end{array}
$$

Com o *inLP* e o *outLP* definidos podemos definir o seu funtor recursivo e com isso podemos admitir o seu catamorfismo e o seu anamorfismo.

Primeiro fazemos com o catamorfismo, faremos o catamorfismo identidade ou seja sendo seu gene o inLP:

$$
\begin{tikzcd}
A^* \times S \arrow[r, "outLP"] \arrow[d, "cataid"'] & 1 \times S + A^* \times (A^* \times S) \arrow[d, "id + (id \times cataid)"] \\
A^* \times S & 1 \times S + A^* \times (A^* \times S) \arrow[l, "inLP"]
\end{tikzcd}
$$

Podemos ver que está certo pelos tipos e sabemos que vai ser analogo ao anamorfismo, e com este diagrama podemos ver claramente o funtor recursivo da estrutura LP sendo:

$$recLP\ f = id + (id \times f)$$

Com o *recLP*, o *inLP* e o *outLP* ja podemos admitir catamorfismo e anamorfismos

$$cataLP\ g = g \cdot recLP\ (cataLP\ g) \cdot outLP$$

$$anaLP\ g = inLP \cdot recLP\ (anaLP\ g) \cdot g$$

Agora facilmente pela ordem natural de computação dos catamorfismos e anamorfismos posso definir as funções mapAccumRfilter e mapAccumLfilter como:

$$mapAccumRfilter\ h\ f = cataLP\ [nil \times id, auxR\ h\ f]$$
$$mapAccumLfilter\ h\ f = anaLP\ ((id + auxL\ h\ f) \cdot outLP)$$

Agora nos falta fazer um simples Cálculo de Programas para determinar auxR e auxL mas antes fazemos os diagramas para deixar explícito os tipos de ambas as funções

$$\begin{tikzcd}
A^* \times S \arrow[r, "outLP"] \arrow[d, "mapAccumRfilter\ h\ f"'] & 1 \times S + A^* \times (A^* \times S) \arrow[d, "id + (id \times mapAccumRfilter\ h\ f)"] \\
B^* \times S & 1 \times S + A^* \times (B^* \times S) \arrow[l, "{[nil \times id, auxR\ h\ f]}"]
\end{tikzcd}$$

tipo de auxR:

$$A^* \times (B^* \times S) \xrightarrow{auxR\ h\ f} B^* \times S$$

ou em Haskell:

$$auxR :: ((a, s) \to Bool) \to ((a, s) \to (c, s)) \to ([a], ([c], s)) \to ([c], s)$$

Agora desenvolveremos o diagrama a partir do tipo inicial tendo em conta as funções h (verificação) e f (transformação)

$$\begin{tikzcd}
A^* \times (B^* \times S) \arrow[r, "head \times id"] & A \times (B^* \times S) \arrow[d, "\langle id \times \pi_2, assocl \cdot (id \times swap) \rangle"'] & \\
& (A \times S) \times ((A \times S) \times B^*) \arrow[d, "(h \times (f \times id))"'] & \\
& 2 \times ((B \times S) \times B^*) \arrow[d, "grd\ \pi_1"'] & \\
& 2 \times ((B \times S) \times B^*) + 2 \times ((B \times S) \times B^*) \arrow[dl, "\pi_2"'] \arrow[dr, "\pi_2"] \arrow[dd, "choice"'] & \\
((B \times S) \times B^*) \arrow[dr, "swap \cdot (id \times \widehat{(:)}) \cdot assocr \cdot (swap \times id)"'] & & ((B \times S) \times B^*) \arrow[dl, "swap \cdot (\pi_2 \times id)"] \\
& B^* \times S &
\end{tikzcd}$$

Temos a definição de auxR:

$$auxR\ h\ f = choice \cdot grd\ \pi_1 \cdot (h \times (f \times id)) \cdot \langle id \times \pi_2, assocl \cdot (id \times swap) \rangle \cdot (head \times id)\ \mathbf{where}$$
$$\quad choice = [swap \cdot (id \times \widehat{(:)}) \cdot assocr \cdot (swap \times id) \cdot \pi_2, swap \cdot (\pi_2 \times id) \cdot \pi_2]$$

Agora faremos parecido para o auxL partindo de um anamorfismo:

$$
\begin{tikzcd}
B^* \times S & & 1 \times S + B^* \times (B^* \times S) \arrow[ll, "inLP"'] \\
A^* \times S \arrow[u, "mapAccumLfilter"] \arrow[r, "outLP"'] & 1 \times S + A^* \times (A^* \times S) \arrow[r, "id + auxL\ h\ f"'] & 1 \times S + C^* \times (A^* \times S) \arrow[u, "is + (id \times mapAccumLfilter)"']
\end{tikzcd}
$$

Sendo então o tipo da função auxL:

$$A^* \times (A^* \times S) \xrightarrow[auxL\ h\ f]{} C^* \times (A^* \times S)$$

Em Haskell definido:

$$auxL :: ((a, s) \rightarrow Bool) \rightarrow ((a, s) \rightarrow (c, s)) \rightarrow ([a], ([a], s)) \rightarrow ([c], ([a], s))$$

Agora desenvolveremos o diagrama a partir do tipo inicial tendo em conta as funções h (verificação) e f (transformação) pra chegar numa conclusão do codigo do

$$
\begin{tikzcd}
A \times (S \times A^*) \arrow[d, "assocl"] & A^* \times (A^* \times S) \arrow[l, "(head \times swap)"'] \arrow[ddddr, "auxL\ h\ f"] & \\
(A \times S) \times A^* \arrow[d, "(\langle h, f \rangle \times id)"] & & \\
2 \times (C \times S) \times A^* \arrow[d, "assocr"] & & \\
2 \times ((C \times S) \times A^*) \arrow[d, "(id \times ((id \times swap) \cdot assocr))"] & & \\
2 \times (C \times (A^* \times S)) \arrow[rr, "choice"] \arrow[d, "grd\ \pi_1"] & & C^* \times (A^* \times S) \\
2 \times ((C \times S) \times A^*) + 2 \times ((C \times S) \times A^*) \arrow[d, "\pi_2 + \pi_2"] & & \\
(C \times S) \times A^* \arrow[d, "assocr + assocr"] & & \\
C \times (S \times A^*) + C \times (S \times A^*) \arrow[d, "(singl \times swap) + (nil \times swap)"] & & \\
C^* \times (A^* \times S) + C^* \times (A^* \times S) \arrow[r, "{[id, id]}"] & C^* \times (A^* \times S) \arrow[uuuur, "id"] &
\end{tikzcd}
$$

Com isso finalmente temos a definição do auxL em Haskell:

$auxL\ h\ f = choice \cdot (id \times ((id \times swap) \cdot assocr)) \cdot assocr \cdot (\langle h, f \rangle \times id) \cdot assocl \cdot (head \times swap)$ **where**
  $choice = [(singl \times id) \cdot \pi_2, (nil \times id) \cdot \pi_2] \cdot grd\ \pi_1$

Problema 3

O problema 3 foi resolvido por recursividade mutua, primeiro como objetivo: definir as funçoes que mutuamente realizam chamadas recursivas entre si. Para isto precisamos da forma recursiva do somatorio em haskell

$picalcRec :: Fractional\ a \Rightarrow \mathbb{Z} \rightarrow a$
$picalcRec\ 0 = 2$
$picalcRec\ n = fromIntegral\ (Nat.fac\ n) \uparrow 2 * 2 \uparrow (n+1)\ /\ fromIntegral\ (Nat.fac\ (2*n+1)) + \pi_{calc}\ (n-1)$

Podemos ver que nesta definição há várias dependencia de n, temos que aplicar a *regra da algibeira* para cada ocorrencia, ou seja, para cada ocorrencia de n temos que associar a outra função qualquer f (n-1)

$$
\begin{array}{ll}
 & n\,!\hat{}\ 2 * 2 \uparrow \{n+1\}\ /\ (2*n+1)\,! + \pi_{calc}\ (n-1) \\
\equiv & \qquad \{\ \mathbf{let}\ f\ (n-1) = n!\ \} \\
 & f\ (n-1) = n! \Rightarrow f\ n = (n+1)! \Rightarrow \left\{ \begin{array}{l} f\ 0 = 1 \\ f\ n = (n+1) * f\ (n-1) \end{array} \right. \\
\equiv & \qquad \{\ \mathbf{let}\ f2\ (n-1) = n+1\ \} \\
 & \left\{ \begin{array}{l} \left\{ \begin{array}{l} f\ 0 = 1 \\ f\ n = f2\ (n-1) * f\ (n-1) \end{array} \right. \\ \left\{ \begin{array}{l} f2\ 0 = 2 \\ f2\ n = n+2 \end{array} \right. \end{array} \right.
\end{array}
$$

Feito para o n!. Agora aplicamos a mesma regra a todas as ocorrencia de n e por fim extraimos os seus valores iniciais e operações para o *ciclo for*

$$
\begin{array}{ll}
 & f\ (n-1) \uparrow 2 * 2 \uparrow \{n+1\}\ /\ (2*n+1)\,! + picalcRec\ (n-1) \\
\equiv & \qquad \{\ \mathbf{let}\ t\ (n-1) = 2 \uparrow \{n+1\}\ \} \\
 & t\ (n-1) = 2 \uparrow \{n+1\} \Rightarrow t\ n = 2 \uparrow \{n+2\} \Rightarrow \left\{ \begin{array}{l} t\ 0 = 4 \\ t\ n = 2 * t\ (n-1) \end{array} \right.
\end{array}
$$

$$
\begin{array}{ll}
 & f\ (n-1) \uparrow 2 * t\ (n-1)\ /\ (2*n+1)\,! + \pi_{calc}\ (n-1) \\
\equiv & \qquad \{\ \mathbf{let}\ g\ (n-1) = (2*n+1)!\ \} \\
 & g\ (n-1) = (2*n+1)! \Rightarrow g\ n = (2*n+3)! \Rightarrow \left\{ \begin{array}{l} g\ 0 = 6 \\ g\ n = (2*n+2) * (2*n+3) * g\ (n-1) \end{array} \right. \\
\equiv & \qquad \{\ \mathbf{let}\ g1\ (n-1) = 2*n+2\ and\ g2\ (n-1) = (2*n+3)\ \} \\
 & \left\{ \begin{array}{l} \left\{ \begin{array}{l} g1\ 0 = 4 \\ g1\ n = 2*n+4 \end{array} \right. \\ \left\{ \begin{array}{l} g2\ 0 = 5 \\ g2\ n = 2*n+5 \end{array} \right. \end{array} \right. \quad \left\{ \begin{array}{l} g\ 0 = 6 \\ g\ n = g1\ (n-1) * g2\ (n-1) * g\ (n-1) \end{array} \right.
\end{array}
$$

Agora finalmente temos todas as operações e valores inicias tendo a nossa equação do picalc neste modelo

$$\begin{cases} \pi_{calc}\ 0 = 2 \\ f\ (n-1) \uparrow 2 * t\ (n-1)\ /\ g\ (n-1) + \pi_{calc}\ (n-1) \end{cases}$$

Vamos agora construir piloop com as informações que nos calculamos anteriormente (para simplificação chamaremos s = picalcRec)

Vamos agorar fazer as operações:

Operação do s:

$$inicS = 2$$
$$op\_S\ s\ f\ g\ t = fromIntegral\ (f \uparrow 2 * t)\ /\ fromIntegral\ g + s$$

Operação do g:

$$inicG = 6$$
$$op\_G\ g\ g1\ g2 = g * g1 * g2$$

Operação do t:

$$inicT = 4$$
$$op\_T\ t = 2 * t$$

Operação do f:

$$inicF = 1$$
$$op\_F\ f\ f2 = f * f2$$

operação do f2

$$inicF2 = 2$$
$$op\_F2\ f\ f2 = \mathsf{succ}\ \ f2$$

Operação do g1:

$$inicG1 = 4$$
$$op\_G1\ g1 = g1 + 2$$

Operação do g2:

$$inicG2 = 5$$
$$op\_G2\ g2 = g2 + 2$$

Temos agora tudo para construir o *ciclo for*

$$worker = \mathsf{for}\ loop\ inic$$

Os nossos valores iniciais vão ser os casos bases de cada função que definimos

$$inic = (inicS, inicG, inicT, inicF, inicF2, inicG1, inicG2)$$

E as operações ja definimos também:

$$
\begin{array}{l}
loop\ (s,g,t,f,f2,g1,g2) = \\
\quad (op\_S\ s\ f\ g\ t, \\
\qquad op\_G\ g\ g1\ g2, \\
\qquad op\_T\ t, \\
\qquad op\_F\ f\ f2, \\
\qquad op\_F2\ f\ f2, \\
\qquad op\_G1\ g1, \\
\qquad op\_G2\ g2 \\
\qquad)
\end{array}
$$

por fim precisamos filtrar somente o nosso resultado que vai ser a primeira componente do tuplo:

$$wrapper\ (x,\_,\_,\_,\_,\_,\_) = x$$

Por fim temos a nossa função feita baseada num *ciclo for*

$$\pi_{loop} = wrapper \cdot worker$$

Problema 4

Primeiro resolvendo o functor que se aplica em Vec terá os tipos:

$$Vec\ A \xrightarrow{\mathsf{fmap}\ f} Vec\ B$$

Portanto facilmente chegamos numa definição deste functor explicita neste diagrama:

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

e em Haskell como:

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

Agora temos que definir o (»=) e o return deste Monad, para tal efeito precisaremos de ter μ para a facilitação de cálculos e completude do problema.

Nosso μ terá obviamente os tipos:

$$Vec\ (Vec\ A) \xrightarrow{\mu} Vec\ A$$

partiremos então com a resolução do diagrama nomeando nossa função μ como *miuV*:

$$
\begin{tikzcd}
Vec\ (Vec\ A) \arrow[d, "outV"'] & \\
(Vec\ A \times Int)^* \arrow[d, "(outV \times id)"'] \arrow[ddr, bend left, "concatMap\ (miuaux \cdot (outV \times id))"] & \\
((A \times Int)^* \times Int)^* \arrow[d, "\mathsf{map}\ \ miuaux"'] & \\
(A \times Int)^{**} \arrow[r, "concat"'] & (A \times Int)^* \arrow[d, "V"'] \\
& Vec\ A
\end{tikzcd}
$$

também teremos que fazer o *miuaux* que fará a multiplicação de todos os elementos do vetor associado ao número.

$$miuaux :: ([(a, Int)], Int) \rightarrow [(a, Int)]$$

$$
\begin{tikzcd}
(A \times Int)^* \times Int \arrow[r, "\langle id, length \rangle \times id"] \arrow[d, "miuaux"'] & ((A \times Int)^* \times \mathbb{N}_0) \times Int \arrow[r, "assocr"] & (A \times Int)^* \times (\mathbb{N}_0 \times Int) \arrow[d, "(id \times \widehat{replicate})"] \\
(A \times Int)^* & ((A \times Int) \times Int)^* \arrow[l, "\mathsf{map}\ \ ((id \times \widehat{(*)}) \cdot assocr)"] & (A \times Int)^* \times Int^* \arrow[l, "\widehat{zip}"]
\end{tikzcd}
$$

Temos agora definido as duas funções em haskell.

$$
\begin{aligned}
&miuaux = \mathsf{map}\ \ ((id \times \widehat{(*)}) \cdot assocr) \cdot \widehat{zip} \cdot (id \times \widehat{replicate}) \cdot assocr \cdot (\langle id, length \rangle \times id) \\
&miuV :: Vec\ (Vec\ a) \rightarrow Vec\ a \\
&miuV = V \cdot concatMap\ (miuaux \cdot (outV \times id)) \cdot outV
\end{aligned}
$$

Agora com o *miuV* definido podemos facilmente calcular (x »= f) de acordo com a (88) do formulário de Cálculo de Programas

$$
\begin{aligned}
& x \gg\!= f = (\mu \cdot T\ f)\ x \\
\equiv \quad & \qquad \{\ substituindo\ pelo\ o\ nosso\ contexto\ \} \\
& x \gg\!= f = (miuV \cdot \mathsf{fmap}\ f)\ x
\end{aligned}
$$

Agora nos falta calcular o return. Sabemos que miuV . return = id então o return será

$$return = V \cdot singl \cdot \langle id, \underline{1} \rangle$$

Associamos ao numero 1 pois 1 é o elemento neutro da multiplicação então quando aplicamos miuv isto multiplica todos os elementos por 1 resultando no mesmo vetor. Exprimos em haskell o *binding* e o *return* como:

Monad:

```haskell
instance Monad Vec where
  x >>= f = miuV (fmap f x)
  return = V . singl . ⟨id, 1⟩
```

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).