



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

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 seleccionadas 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 neurais ([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 seleccionar que etapas da computação geram ou não *outputs* — obtendo-se assim o efeito 'many-to-one'. Ter-se-á, para esse efeito:

$$\begin{aligned} \text{mapAccumRfilter} &:: ((a, s) \rightarrow \text{Bool}) \rightarrow ((a, s) \rightarrow (c, s)) \rightarrow ([a], s) \rightarrow ([c], s) \\ \text{mapAccumLfilter} &:: ((a, s) \rightarrow \text{Bool}) \rightarrow ((a, s) \rightarrow (c, s)) \rightarrow ([a], s) \rightarrow ([c], s) \end{aligned}$$

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

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

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

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:

$$\pi_{loop} = \dots \text{ for loop inic where } \dots$$

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 .

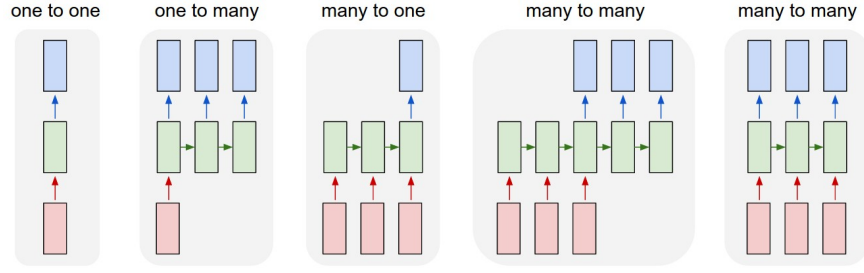


Figure 2: Várias tipologias de RNNs [1].

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 [(X1, 2), (X2, 1), (X3, 0)]
```

assumindo definido o tipo

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

e o “tipo-dimensão”:

```
data X = X1 | X2 | X3 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 → Vec Bool
mat X1 = V [(False, 1), (True, 0)]
mat X2 = V [(False, -1), (True, -3)]
mat X3 = 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

$$vec' = vec \gg mat$$

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

```
neg :: Bool → 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 $X \rightarrow Vec\ Bool$.

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 >>= 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.

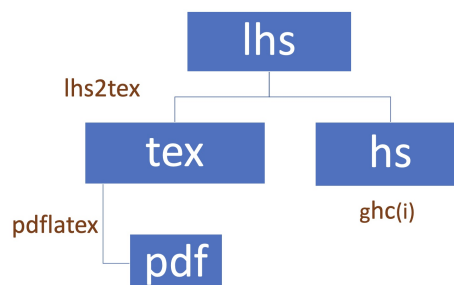
Para cumprir de forma integrada os objectivos do trabalho vamos recorrer a uma técnica de programação dita “[literária](#)” [2], cujo princípio base é o seguinte:

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 [L^AT_EX](#). (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 [L^AT_EX](#). 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-processor que faz “pretty printing” de código [Haskell](#) em [L^AT_EX](#) 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 ler¹ onde se obtém o efeito seguinte:²

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

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

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

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 $F X = 1 + X$) 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.

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

Obter-se-á de imediato

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

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* colecionam-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 aulas², 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:

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

F Código fornecido

Problema 1

Teste relativo à figura da página 1:

$$test_1 = mostwater\ hghts$$

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.

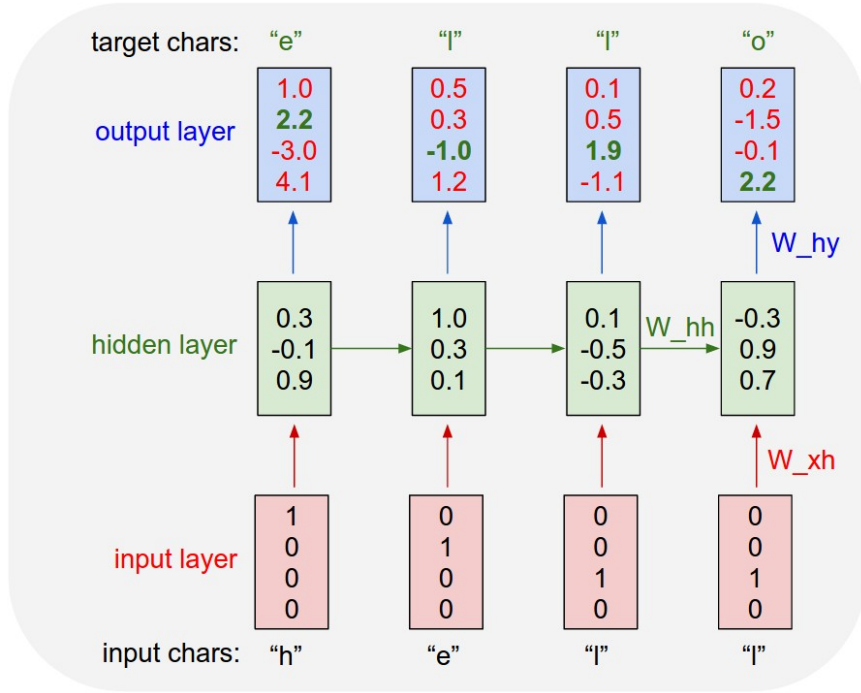


Figure 3: Exemplo *char seq* extraído de [1].

```
test2a = mapAccumLfilter ((>10) · π1) f (odds 12, 0)
test2b = mapAccumRfilter ((>10) · π1) f (odds 12, 0)
```

onde:

```
odds n = map ((1+) · (2*)) [0..n-1]
f (a, s) = (s, a + s)
```

Teste

```
test2c = mapAccumLfilter true step ([x1, x2, x3, x4], h0)
```

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

- Estado inicial:

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

- Step function:

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

- Função de activação:

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

- Input layer:

```
inp = [x1, x2, x3, x4]
x1 = fromList 4 1 [1.0, 0, 0, 0]
x2 = fromList 4 1 [0, 1.0, 0, 0]
x3 = fromList 4 1 [0, 0, 1.0, 0]
x4 = x3
```

- 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:

```
 $\pi_{calc} n = (sum \cdot map f) [0..n] \textbf{ where}$ 
 $f n = fromIntegral (n! * n! * (g n)) / fromIntegral (d n)$ 
 $g n = 2 \uparrow (n + 1)$ 
 $d n = (2 * n + 1)!$ 
```

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*:

```
instance (Show a, Ord a, Eq a) => Show (Vec a) where
  show = showbag · consol · outV where
    showbag = concat ·
      (++) [" "] · ("{" "·") ·
      (intersperse " , ") ·
      sort ·
      (map f) where f (a,b) = (show a) ++ " |-> " ++ (show b)
```

Outros detalhes da implementação de *Vec* em Haskell:

```
instance Applicative Vec where
  pure = return
  (< * >) = aap
instance (Eq a) => Eq (Vec a) where
  b ≡ b' = (outV b) 'lequal' (outV b')
  where lequal a b = isempty (a ⊖ b)
    a ⊖ b = a ++  $\bar{b}$ 
     $\bar{x} = [(k, -i) \mid (k, i) \leftarrow x]$ 
```

Funções auxiliares:

```
consol :: (Eq b) => [(b, Int)] -> [(b, Int)]
consol = filter nzero · map (id × sum) · col where nzero (_,x) = x ≠ 0
isempty :: Eq a => [(a, Int)] -> Bool
isempty = all (≡ 0) · map π2 · consol
col :: (Eq a, Eq b) => [(a, b)] -> [(a, [b])]
col x = nub [k ↦ [d' | (k', d') ← x, k' ≡ k] | (k, d) ← x] where a ↦ b = (a, b)
```

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:

$$\begin{aligned} \text{mysuffixes} &:: [a] \rightarrow [[a]] \\ \text{mysuffixes} &= \ll (id + \langle \text{cons}, \pi_2 \rangle) \cdot \text{outList} \rrbracket \end{aligned}$$

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

$$\mathbb{N}_0^* \xrightarrow{\text{mysuffixes}} (\mathbb{N}_0^*)^* \xrightarrow[\text{map } \langle \text{head}, \text{tail} \rangle]{} \mathbb{N}_0 \times \mathbb{N}_0^*$$

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

$$\begin{aligned} \text{mymaximum} &:: [Int] \rightarrow Int \\ \text{mymaximum} &= \ll [0, \widehat{\text{max}}] \rrbracket \end{aligned}$$

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

$$\mathbb{N}_0^* \xrightarrow{\text{mymaximum}} \mathbb{N}_0$$

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

$$\text{area} :: (Int, [Int]) \rightarrow Int$$

$$\begin{array}{ccc} \mathbb{N}_0 \times \mathbb{N}_0^* & \xrightarrow{\text{area}} & \mathbb{N}_0 \\ \downarrow id \times ((\text{map } \text{swap}) \cdot \text{zip } [1..]) & & \uparrow \text{mymaximum} \\ \mathbb{N}_0 \times (\mathbb{N}_0 \times \mathbb{N}_0)^* & & \\ \downarrow \langle id \times \text{len}, \pi_2 \rangle & & \\ (\mathbb{N}_0 \times \mathbb{N}_0) \times (\mathbb{N}_0 \times \mathbb{N}_0)^* & & \\ \downarrow (\widehat{\text{replicate}} \cdot \text{swap}) \times id & & \\ \mathbb{N}_0^* \times (\mathbb{N}_0 \times \mathbb{N}_0)^* & \xrightarrow[\widehat{\text{zip}}]{} (\mathbb{N}_0 \times (\mathbb{N}_0 \times \mathbb{N}_0))^* \xrightarrow[\text{map } \text{auxarea}]{} \mathbb{N}_0^* \end{array}$$

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.

$$\begin{array}{ccccc} \mathbb{N}_0 \times (\mathbb{N}_0 \times \mathbb{N}_0) & \xrightarrow{assocl} & (\mathbb{N}_0 \times \mathbb{N}_0) \times \mathbb{N}_0 & \xrightarrow{(grd \hat{>}) \times id} & ((\mathbb{N}_0 \times \mathbb{N}_0) + (\mathbb{N}_0 \times \mathbb{N}_0)) \times \mathbb{N}_0 \\ \downarrow auxarea & & & \nearrow [\pi_2, \pi_1] \times id & \\ \mathbb{N}_0 & \xleftarrow{(\widehat{*})} & \mathbb{N}_0 \times \mathbb{N}_0 & & \end{array}$$

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

$$auxarea = (\widehat{*}) \cdot ([\pi_2, \pi_1] \times id) \cdot (grd \hat{>} \times id) \cdot assocl$$

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

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

$$\begin{array}{ccc} \mathbb{N}_0^* & & \\ \downarrow mysuffixes & \searrow mostwater & \\ (\mathbb{N}_0^*)^* & & \\ \downarrow map \ \langle head, tail \rangle & & \\ (\mathbb{N}_0 \times \mathbb{N}_0^*)^* & \xrightarrow{map \ area} & \mathbb{N}_0^* \xrightarrow{mymaximum} \mathbb{N}_0 \end{array}$$

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*

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

Temos a definição de *mostwater* como um hylomorfismo:

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

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:

$$\begin{array}{ccccc}
 \mathbb{N}_0^* & \xrightarrow{out_*} & 1 + \mathbb{N}_0 \times \mathbb{N}_0^* & \xrightarrow{(id + \langle cons, \pi_2 \rangle)} & 1 + \mathbb{N}_0^* \times \mathbb{N}_0^* \\
 \downarrow divide & & & & \downarrow id + id \times divide \\
 (\mathbb{N}_0^*)^* & \xrightleftharpoons[in_{**}]{out_{**}} & 1 + \mathbb{N}_0^* \times (\mathbb{N}_0^*)^* & & \\
 \downarrow conquer & & & & \downarrow id + id \times conquer \\
 \mathbb{N}_0 & \xleftarrow{[0, \widehat{max} \cdot ((area \cdot \langle head, tail \rangle) \times id)]} & 1 + \mathbb{N}_0^* \times \mathbb{N}_0 & &
 \end{array}$$

Problema 2

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

$$A^* \times S \xrightleftharpoons[inLP]{outLP} 1 \times S + A^* \times (A^* \times S)$$

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)

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

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*:

$$\begin{array}{ccc}
 A^* \times S & \xrightarrow{outLP} & 1 \times S + A^* \times (A^* \times S) \\
 f \downarrow & & \downarrow id + (id \times f) \\
 A^* \times S & \xleftarrow{inLP} & 1 \times S + A^* \times (A^* \times S)
 \end{array}$$

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

$$\begin{array}{ccc} A^* \times S & \xrightarrow{outLP} & 1 \times S + A^* \times (A^* \times S) \\ mapAccumRfilter\ h\ f \downarrow & & \downarrow id + (id \times mapAccumRfilter\ h\ f) \\ B^* \times S & \xleftarrow{[nil \times id, auxR\ h\ f]} & 1 \times S + A^* \times (B^* \times S) \end{array}$$

tipo de `auxR`:

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

ou em Haskell:

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

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

$$\begin{array}{ccccc} A^* \times (B^* \times S) & \xrightarrow{head \times id} & A \times (B^* \times S) & & \\ & \downarrow \langle id \times \pi_2, assocl \cdot (id \times swap) \rangle & & & \\ & (A \times S) \times ((A \times S) \times B^*) & & & \\ & \downarrow (h \times (f \times id)) & & & \\ & 2 \times ((B \times S) \times B^*) & & & \\ & \downarrow grd\ \pi_1 & & & \\ & 2 \times ((B \times S) \times B^*) + 2 \times ((B \times S) \times B^*) & & & \\ \swarrow \pi_2 & & \downarrow choice & & \searrow \pi_2 \\ ((B \times S) \times B^*) & & B^* \times S & & ((B \times S) \times B^*) \\ \swarrow swap \cdot (id \times \widehat{(\cdot)}) \cdot assocr \cdot (swap \times id) & & & & \swarrow swap \cdot (\pi_2 \times id) \end{array}$$

Temos a definição de `auxR`:

$$auxR\ h\ f = choice \cdot grd\ \pi_1 \cdot (h \times (f \times id)) \cdot \langle id \times \pi_2, assocl \cdot (id \times swap) \rangle \cdot (head \times id) \textbf{ where}$$

$$choice = [swap \cdot (id \times \widehat{(\cdot)}) \cdot assocr \cdot (swap \times id) \cdot \pi_2, swap \cdot (\pi_2 \times id) \cdot \pi_2]$$

Agora faremos parecido para o auxL partindo de um anamorfismo:

$$\begin{array}{ccccc}
 B^* \times S & \xleftarrow{\text{inLP}} & 1 \times S + B^* \times (B^* \times S) \\
 \uparrow \text{mapAccumLfilter} & & \uparrow \text{is} + (\text{id} \times \text{mapAccumLfilter}) \\
 A^* \times S & \xrightarrow{\text{outLP}} 1 \times S + A^* \times (A^* \times S) \xrightarrow{\text{id} + \text{auxL } h f} & 1 \times S + C^* \times (A^* \times S)
 \end{array}$$

Sendo então o tipo da função auxL:

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

Em Haskell definido:

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

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

$$\begin{array}{ccc}
 A \times (S \times A^*) & \xleftarrow{(\text{head} \times \text{swap})} & A^* \times (A^* \times S) \\
 \downarrow \text{assocl} & & \searrow \text{auxL } h f \\
 (A \times S) \times A^* & & \\
 \downarrow (\langle h, f \rangle \times \text{id}) & & \\
 2 \times (C \times S) \times A^* & & \\
 \downarrow \text{assocr} & & \\
 2 \times ((C \times S) \times A^*) & & \\
 \downarrow (\text{id} \times ((\text{id} \times \text{swap}) \cdot \text{assocr})) & & \\
 2 \times (C \times (A^* \times S)) & \xrightarrow{\text{choice}} & C^* \times (A^* \times S) \\
 \downarrow \text{grd } \pi_1 & & \nearrow \text{id} \\
 2 \times ((C \times S) \times A^*) + 2 \times ((C \times S) \times A^*) & & \\
 \downarrow \pi_2 + \pi_2 & & \\
 (C \times S) \times A^* & & \\
 \downarrow \text{assocr} + \text{assocr} & & \\
 C \times (S \times A^*) + C \times (S \times A^*) & & \\
 \downarrow (\text{singl} \times \text{swap}) + (\text{nil} \times \text{swap}) & & \\
 C^* \times (A^* \times S) + C^* \times (A^* \times S) & \xrightarrow{[\text{id}, \text{id}]} & C^* \times (A^* \times S)
 \end{array}$$

Com isso finalmente temos a definição do auxL em Haskell:

$$\begin{aligned}
 \text{auxL } h f &= \text{choice} \cdot (\text{id} \times ((\text{id} \times \text{swap}) \cdot \text{assocr})) \cdot \text{assocr} \cdot (\langle h, f \rangle \times \text{id}) \cdot \text{assocl} \cdot (\text{head} \times \text{swap}) \textbf{ where} \\
 \text{choice} &= [(\text{singl} \times \text{id}) \cdot \pi_2, (\text{nil} \times \text{id}) \cdot \pi_2] \cdot \text{grd } \pi_1
 \end{aligned}$$

Problema 3

O problema 3 foi resolvido por recursividade mutua, primeiro como objetivo: definir as funções 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)

$$\begin{aligned}
 & n! \uparrow 2 * 2 \uparrow \{n + 1\} / (2 * n + 1)! + \pi_{calc} (n - 1) \\
 \equiv & \quad \{ \text{let } f (n - 1) = n! \} \\
 & f (n - 1) = n! \Rightarrow f n = (n + 1)! \Rightarrow \begin{cases} f 0 = 1 \\ f n = (n + 1) * f (n - 1) \end{cases} \\
 \equiv & \quad \{ \text{let } f2 (n - 1) = n + 1 \} \\
 & \left\{ \begin{cases} f 0 = 1 \\ f n = f2 (n - 1) * f (n - 1) \end{cases} \right. \\
 & \left\{ \begin{cases} f2 0 = 2 \\ f2 n = n + 2 \end{cases} \right.
 \end{aligned}$$

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

$$\begin{aligned}
 & f (n - 1) \uparrow 2 * 2 \uparrow \{n + 1\} / (2 * n + 1)! + picalcRec (n - 1) \\
 \equiv & \quad \{ \text{let } t (n - 1) = 2 \uparrow \{n + 1\} \} \\
 & t (n - 1) = 2 \uparrow \{n + 1\} \Rightarrow t n = 2 \uparrow \{n + 2\} \Rightarrow \begin{cases} t 0 = 4 \\ t n = 2 * t (n - 1) \end{cases} \\
 & f (n - 1) \uparrow 2 * t (n - 1) / (2 * n + 1)! + \pi_{calc} (n - 1) \\
 \equiv & \quad \{ \text{let } g (n - 1) = (2 * n + 1)! \} \\
 & g (n - 1) = (2 * n + 1)! \Rightarrow g n = (2 * n + 3)! \Rightarrow \begin{cases} g 0 = 6 \\ g n = (2 * n + 2) * (2 * n + 3) * g (n - 1) \end{cases} \\
 \equiv & \quad \{ \text{let } g1 (n - 1) = 2 * n + 2 \text{ and } g2 (n - 1) = (2 * n + 3) \} \\
 & \left\{ \begin{cases} g1 0 = 4 \\ g1 n = 2 * n + 4 \\ g2 0 = 5 \\ g2 n = 2 * n + 5 \end{cases} \right. \quad \begin{cases} g 0 = 6 \\ g n = g1 (n - 1) * g2 (n - 1) * g (n - 1) \end{cases}
 \end{aligned}$$

Agora finalmente temos todas as operações e valores iniciais 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 $s = \text{picalcRec}$)

Vamos agora fazer as operações:

Operação do s :

$$\begin{aligned} \text{inicS} &= 2 \\ \text{op_S } s \ f \ g \ t &= \text{fromIntegral } (f \uparrow 2 * t) / \text{fromIntegral } g + s \end{aligned}$$

Operação do g :

$$\begin{aligned} \text{inicG} &= 6 \\ \text{op_G } g \ g1 \ g2 &= g * g1 * g2 \end{aligned}$$

Operação do t :

$$\begin{aligned} \text{inicT} &= 4 \\ \text{op_T } t &= 2 * t \end{aligned}$$

Operação do f :

$$\begin{aligned} \text{inicF} &= 1 \\ \text{op_F } f \ f2 &= f * f2 \end{aligned}$$

operação do $f2$

$$\begin{aligned} \text{inicF2} &= 2 \\ \text{op_F2 } f \ f2 &= \text{succ } f2 \end{aligned}$$

Operação do $g1$:

$$\begin{aligned} \text{inicG1} &= 4 \\ \text{op_G1 } g1 &= g1 + 2 \end{aligned}$$

Operação do $g2$:

$$\begin{aligned} \text{inicG2} &= 5 \\ \text{op_G2 } g2 &= g2 + 2 \end{aligned}$$

Temos agora tudo para construir o *ciclo for*

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

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

$$\text{inic} = (\text{inicS}, \text{inicG}, \text{inicT}, \text{inicF}, \text{inicF2}, \text{inicG1}, \text{inicG2})$$

E as operações já definimos também:

```

loop (s, g, t, f, f2, g1, g2) =
  (op_S s f g t,
   op_G g g1 g2,
   op_T t,
   op_F f f2,
   op_F2 f f2,
   op_G1 g1,
   op_G2 g2
  )

```

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

```

miuaux :: ([ (a, Int)], Int) → [(a, Int)]
miuaux = map ((id ×  $\widehat{(*)}$ ) · assocr) ·  $\widehat{zip}$  · (id ×  $\widehat{replicate}$ ) · assocr · ((id, length) × id)
miuV :: Vec (Vec a) → Vec a
miuV = V · concatMap (miuaux · (outV × id)) · outV

```

Functor:

```

instance Functor Vec where
  fmap f = V · map (f × id) · outV

```

Monad:

```

instance Monad Vec where
  x  $\gg=$  f = miuV (fmap f x)
  return = V · singl ·  $\langle id, \underline{1} \rangle$ 

```

Index

\LaTeX , 4

bibtex, 4

lhs2TeX, 3–5

makeindex, 4

pdflatex, 3

xymatrix, 5

Combinador “pointfree”

cata

 Naturais, 5

mapAccumL, 2

mapAccumR, 2

split, 5

Cálculo de Programas, 1, 3

 Material Pedagógico, 3

Docker, 4

 container, 4, 5

Função

π_1 , 5

π_2 , 5

map, 6

Haskell, 1–5

 Hackage

 Data.Matrix, 2

 interpretador

 GHCi, 3, 4

 Literate Haskell, 3

Números naturais (\mathbb{N}), 5

Programação

 literária, 3, 5

Rede neuronal, 2

 RNN, 2, 3

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 10, 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 10, 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](#)).