



Universidade do Minho
Escola de Engenharia

Cálculo de Programas

Trabalho Prático (2024/25)

Lic. em Ciências da Computação

Grupo G99

xxxxxxx	Nome
xxxxxxx	Nome
xxxxxxx	Nome

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.

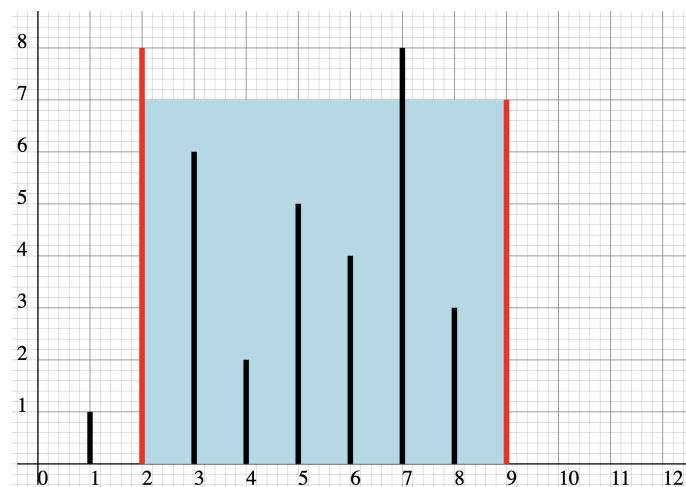
Na sequência pedagógica dos planos de estudo dos cursos que têm esta disciplina, opta-se pela aplicação deste método à programação em [Haskell](#) (sem prejuízo da sua aplicação a outras linguagens funcionais). Assim, o presente trabalho prático coloca os alunos perante problemas concretos que deverão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

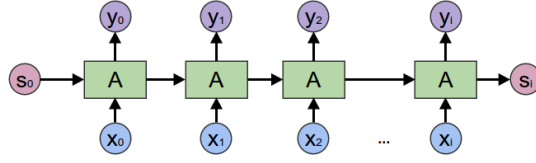


Figure 1: [RNN](#) vista como instância de um *accumulating map* [3].

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 .

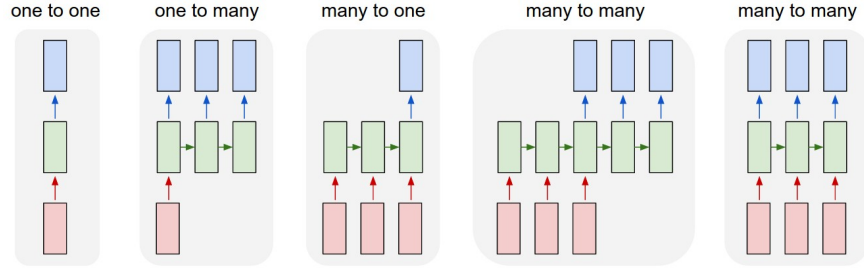


Figure 2: Várias tipologias de RNNs [1].

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.

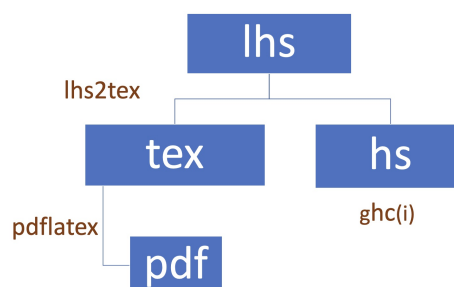
Para cumprir de forma integrada os objectivos do trabalho vamos recorrer a uma técnica de programação dita “[literária](#)” [2], cujo princípio base é o seguinte:

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

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

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:

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

F Código fornecido

Problema 1

Teste relativo à figura da página 1:

$$test_1 = \text{mostwater hghts}$$

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.

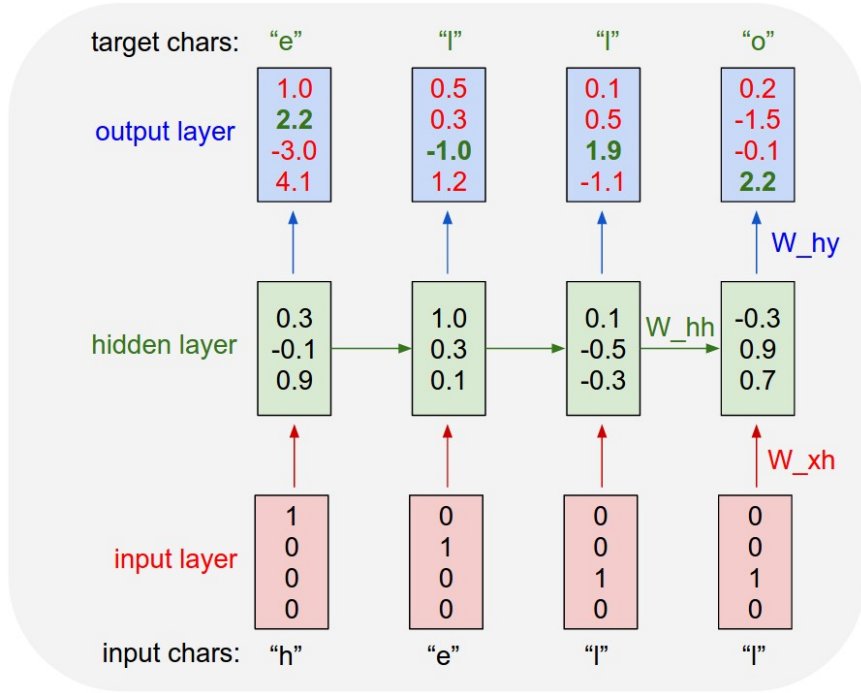


Figure 3: Exemplo *char seq* extraído de [1].

```
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:

```
h0 = fromList 3 1 [1.0, 1.0, 1, 0]
```

- Step function:

```
step (x, h) = (α (wy * h), α (wh * h + wx * x))
```

- Função de activação:

```
α = fmap σ where σ 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

$mostwater = \perp$

Problema 2

$mapAccumRfilter\ p\ f = \perp$

$mapAccumLfilter\ p\ f = \perp$

Problema 3

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

$worker = \text{for loop inic}$

$inic = \perp$

$loop = \perp$

$wrapper = \perp$

Problema 4

Functor:

instance Functor Vec where

$fmap\ f = \perp$

Monad:

instance Monad Vec where

$x \gg= f = \perp$

$return = \perp$

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: May 16, 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: May 16, 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](#)).