



Universidade do Minho
Escola de Engenharia

Cálculo de Programas

Trabalho Prático (2023/24)

Lic. em Ciências da Computação

Grupo G19

a102499 Carlos Daniel Silva Fernandes
a102497 José Bernardo Moniz Fernandes
a102504 Pedro Augusto Camargo

Preâmbulo

Cálculo de Programas tem como objectivo principal 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 mediante a utilização dos combinadores de ordem superior estudados na disciplina. Recomenda-se ainda que o código venha acompanhado de uma descrição de como funciona e foi concebido, apoiado em diagramas explicativos. Para instruções sobre como produzir esses diagramas e exprimir raciocínios de cálculo, ver o anexo **D**.

Problema 1

No passado dia 10 de Março o país foi a eleições para a Assembleia da República. A lei eleitoral portuguesa segue, como as de muitos outros países, o chamado **Método de Hondt** para seleccionar os candidatos dos vários partidos, conforme os votos que receberam. E, tal como em anos anteriores, há sempre *notícias* a referir a quantidade de votos desperdiçados por este método. Como e porque é que isso acontece?

Pretende-se nesta questão construir em Haskell um programa que implemente o método de Hondt. A **Comissão Nacional de Eleições** descreve esse método [nesta página](#), que deverá ser estudada para resolver esta questão. O quadro que aí aparece,

Divisor	Partido			
	A	B	C	D
1	12000	7500	4500	3000
2	6000	3750	2250	1500
3	4000	2500	1500	1000
4	3000	1875	1125	750

mostra o exemplo de um círculo eleitoral que tem direito a eleger 7 deputados e onde concorrem às eleições quatro partidos *A*, *B*, *C* e *D*, cf:

data *Party* = *A* | *B* | *C* | *D* **deriving** (*Eq*, *Ord*, *Show*)

A votação nesse círculo foi

$[(A, 12000), (B, 7500), (C, 4500), (D, 3000)]$

sendo o resultado eleitoral

$$result = [(A, 3), (B, 2), (C, 1), (D, 1)]$$

apurado correndo

$$result = final\ history$$

que corresponde à última etapa da iteração:

$$history = [for\ step\ db\ i \mid i \leftarrow [0..7]]$$

Verifica-se que, de um total de 27000 votos, foram desperdiçados:

$$wasted = 9250$$

Completem no anexo G as funções que se encontram aí indefinidas¹, podendo adicionar funções auxiliares que sejam convenientes. No anexo F é dado algum código preliminar.

Problema 2

A biblioteca *LTree* inclui o algoritmo “mergesort” (*mSort*), que é um hilomorfismo baseado função

$$merge :: Ord\ a \Rightarrow ([a], [a]) \rightarrow [a]$$

que junta duas listas previamente ordenadas numa única lista ordenada.

Nesta questão pretendemos generalizar *merge* a *k*-listas (ordenadas), para qualquer *k* finito:

$$mergek :: Ord\ a \Rightarrow [[a]] \rightarrow [a]$$

Esta função deverá ser codificada como um hilomorfismo, a saber:

$$mergek = \llbracket f, g \rrbracket$$

1. Programe os genes *f* e *g* do hilomorfismo *mergek*.
2. Estenda *mSort* a

$$mSortk :: Ord\ a \Rightarrow Int \rightarrow [a] \rightarrow [a]$$

por forma a este hilomorfismo utilizar *mergek* em lugar de *merge* na etapa de “conquista”. O que se espera de *mSortk k* é que faça a partição da lista de entrada em *k* sublistas, sempre que isso for possível. (Que vantagens vê nesta nova versão?)

Problema 3

Considere-se a fórmula que dá o *n*-ésimo número de Catalan:

$$C_n = \frac{(2n)!}{(n+1)!(n!)} \tag{1}$$

No anexo F dá-se a função *catdef* que implementa a definição (10) em Haskell. É fácil de verificar que, à medida que *n* cresce, o tempo que *catdef n* demora a executar degrada-se.

¹ Cf. \perp no código.

Pretende-se uma implementação mais eficiente de C_n que, derivada por recursividade mútua, não calcule factoriais nenhuns:

$cat = \dots$ for loop init **where** \dots

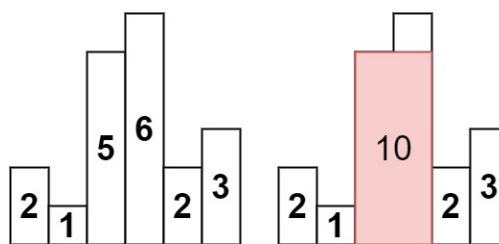
No anexo F é dado um oráculo que pode ajudar a testar cat . Deverá ainda ser comparada a eficiência da solução calculada cat com a de $catdef$.

Sugestão: Começar por estudar a regra prática que se dá no anexo E para problemas deste género.

Problema 4

Esta questão aborda um problema que é conhecido pela designação *Largest Rectangle in Histogram*. Precebe-se facilmente do que se trata olhando para a parte esquerda da figura abaixo, que mostra o histograma correspondente à sequência numérica:

$h :: [Int]$
 $h = [2, 1, 5, 6, 2, 3]$



À direita da mesma figura identifica-se o rectângulo de maior área que é possível inscrever no referido histograma, com área $10 = 2 * 5$.

Pretende-se a definição de uma função em Haskell

$lrh :: [Int] \rightarrow Int$

tal que $lrh\ x$ seja a maior área de rectângulos que seja possível inscrever em x .

Pretende-se uma solução para o problema que seja simples e estruturada num hilomorfismo baseado num tipo indutivo estudado na disciplina ou definido *on purpose*.

Anexos

A Natureza do trabalho a realizar

Este trabalho teórico-prático deve ser realizado por grupos de 3 alunos. Os detalhes da avaliação (datas para submissão do relatório e sua defesa oral) são os que forem publicados na [página da disciplina](#) na *internet*.

Recomenda-se uma abordagem participativa dos membros do grupo em **todos** os exercícios do trabalho, para assim poderem responder a qualquer questão colocada na *defesa oral* do relatório.

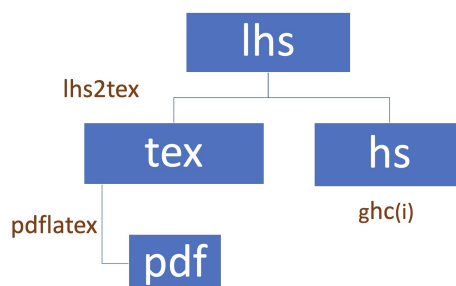
Para cumprir de forma integrada os objectivos do trabalho vamos recorrer a uma técnica de programação dita “[literária](#)” [1], cujo princípio base é o seguinte:

Um programa e a sua documentação devem coincidir.

Por outras palavras, o **código fonte** e a **documentação** de um programa deverão estar no mesmo ficheiro.

O ficheiro `cp2324t.pdf` que está a ler é já um exemplo de [programação literária](#): foi gerado a partir do texto fonte `cp2324t.lhs`¹ que encontrará no [material pedagógico](#) desta disciplina descompactando o ficheiro `cp2324t.zip`.

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



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

B Docker

Recomenda-se o uso do [container](#) cuja imagem é gerada pelo [Docker](#) a partir do ficheiro `Dockerfile` que se encontra na diretoria que resulta de descompactar `cp2324t.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.)

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

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

```
$ docker build -t cp2324t .  
$ docker run -v ${PWD}:/cp2324t -it cp2324t
```

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}:/cp2324t`) que permite que a diretoria em que se encontra na sua máquina local e a diretoria `/cp2324t` no [container](#) sejam partilhadas.

O grupo deverá visualizar/editar os ficheiros numa máquina local e compilá-los no [container](#), executando:

```
$ lhs2TeX cp2324t.lhs > cp2324t.tex  
$ pdflatex cp2324t
```

[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 `cp2324t.lhs` é executável e contém o “kit” básico, escrito em [Haskell](#), para realizar o trabalho. Basta executar

```
$ ghci cp2324t.lhs
```

O grupo deve abrir o ficheiro `cp2324t.lhs` num editor da sua preferência e verificar 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 [Bib_TE_X](#)) e o índice remissivo (com [makeindex](#)),

```
$ bibtex cp2324t.aux  
$ makeindex cp2324t.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.

Importante: o grupo deve evitar trabalhar fora deste ficheiro [lhs](#) que lhe é fornecido. Se, para efeitos de divisão de trabalho, o decidir fazer, deve **regularmente integrar** e validar as soluções que forem sendo obtidas neste [lhs](#), garantindo atempadamente a compatibilidade com este. Se não o fizer corre o risco de vir a submeter um ficheiro que não corre no GHCi e/ou apresenta erros na geração do PDF.

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* \LaTeX [xymatrix](#), por exemplo:

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

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:

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

¹ Procure e.g. por "sec:diagramas".

² Exemplos tirados de [2].

³ Lei (3.95) em [2], página 110.

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

Tipos básicos:

$$\begin{aligned} \text{type Votes} &= \mathbb{Z} \\ \text{type Deputies} &= \mathbb{Z} \end{aligned}$$

Dados:

$$\begin{aligned} db &:: [(Party, (Votes, Deputies))] \\ db &= \text{map } f\ \text{vote where } f\ (a, b) = (a, (b, 0)) \\ \text{vote} &= [(A, 12000), (B, 7500), (C, 4500), (D, 3000)] \end{aligned}$$

Apuramento:

$$\begin{aligned} final &= \text{map } (id \times \pi_2) \cdot last \\ total &= \text{sum } (\text{map } \pi_2\ \text{vote}) \\ wasted &= \text{waste history} \end{aligned}$$

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

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

Problema 3

Definição da série de Catalan usando factoriais (10):

$$catdef\ n = (2 * n)! \div ((n + 1)! * n!)$$

Oráculo para inspecção dos primeiros 26 números de Catalan¹:

```
oracle = [  
  1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, 208012, 742900, 2674440, 9694845,  
  35357670, 129644790, 477638700, 1767263190, 6564120420, 24466267020,  
  91482563640, 343059613650, 1289904147324, 4861946401452  
]
```

G Soluções dos alunos

Os grupos 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, diagramas e/ou outras funções auxiliares que sejam necessárias.

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

Problema 1

Ao analisar o problema, percebemos que a resolução do mesmo dependia do tipo de informação contida no history. Para isso começámos por definir o **step** e posteriormente o **waste**.

```
step [] = []  
step l = aux maxPair l  
where  
  maxPair = maximumBy (comparing ( $\pi_1 \cdot \pi_2$ )) l  
  aux _ [] = []  
  aux (maxP, (maxV, maxD)) ((p, (v, d)) : rest)  
    | maxP  $\equiv p \wedge$  maxD  $\equiv 0 = (p, (v \text{ 'div' } 2, \text{succ } d)) : rest$   
    | maxP  $\equiv p = (p, ((v * \text{succ } d) \text{ 'div' } (d + 2), \text{succ } d)) : rest$   
    | otherwise = (p, (v, d)) : aux maxPair rest
```

A cada iteração calculamos o partido com o maior número de votos e guardamos a informação em **maxPair**. Se tiver 0 deputados, para obter os “novos” votos, dividimos os votos totais desse partido por 2 e de seguida adicionamos um deputado. Se tiver mais que 0 deputados, os votos irão ser a divisão dos votos totais desse partido pelo seu número de deputados mais 2, de seguida aumentamos o número de deputados. Por fim, se não for o partido com mais votos, mantemos os votos e os deputados inalterados.

O **history** é a aplicação da função **for step db** em **i** vezes, que origina uma lista de listas possuindo a informação referente a cada patido (votos, deputados).

¹ Fonte: [Wikipedia](https://pt.wikipedia.org/wiki/Catalan).

```

0 [(A, (12000, 0)), (B, (7500, 0)), (C, (4500, 0)), (D, (3000, 0))]
1 [(A, (6000, 1)), (B, (7500, 0)), (C, (4500, 0)), (D, (3000, 0))]
2 [(A, (6000, 1)), (B, (3750, 1)), (C, (4500, 0)), (D, (3000, 0))]
3 [(A, (4000, 2)), (B, (3750, 1)), (C, (4500, 0)), (D, (3000, 0))]
4 [(A, (4000, 2)), (B, (3750, 1)), (C, (2250, 1)), (D, (3000, 0))]
5 [(A, (3000, 3)), (B, (3750, 1)), (C, (2250, 1)), (D, (3000, 0))]
6 [(A, (3000, 3)), (B, (2500, 2)), (C, (2250, 1)), (D, (3000, 0))]
7 [(A, (3000, 3)), (B, (2500, 2)), (C, (2250, 1)), (D, (1500, 1))]

```

Já o nosso **waste** soma todos os votos da última lista do **history**, que contém os votos desperdiçados.

$$\text{waste} = \text{sum} \cdot \text{map } (\pi_1 \cdot \pi_2) \cdot \text{last}$$

Problema 2

Para a descoberta dos genes **f** e **g** referentes ao hilomorfismo **mergek**, elaboramos o seguinte diagrama:

$$\begin{array}{ccc}
 (A^*)^* & \xrightarrow{\text{divide}} & 1 + A \times (A^*)^* \\
 \text{mergeK} \downarrow & & \downarrow \text{id} + \text{id} \times \text{mergeK} \\
 A^* & \xleftarrow{\text{conquer} = [\text{nil}, \text{cons}]} & 1 + A \times A^*
 \end{array}$$

Genes de *mergek*:

```

f = inList
g [] = i1 ()
g l = if filter (≠ []) l ≡ []
    then i1 ()
    else i2 (m, l') where
      m = minimum (map head x)
      l' = aux m x
      aux _ [] = []
      aux m (h : t) = if m ≡ head h
        then tail h : t
        else h : aux m t
      x = filter (≠ []) l

```

Na nossa resolução o "núcleo" do algoritmo é realizado em grande parte pelo anamorfismo, ficando como um catamorfismo que é o **inList**.

Extensão de *mSort*:

Utilizamos a função **chunksOf** para dividir a lista em k partes, assegurando que a divisão só ocorre se o comprimento da lista for superior a k. Desta forma, evitamos que o algoritmo entre em ciclo infinito. Posteriormente, utilizamos a função **mergek** para juntar as partes ordenadas.

```

mSortk k [] = []
mSortk k l
  | length l ≤ 1 = l
  | otherwise = let chunks = chunksOf (if length l ≤ k then 1 else k) l
                in mergek (map (mSortk k) chunks)

```

Com uma abordagem que divide a lista em k partes, é possível reduzir significativamente o número de comparações e operações necessárias para ordenar a lista, tornando o algoritmo mais eficiente. Este método beneficia do paralelismo, uma vez que permite dividir a lista em k partes e ordená-las em simultâneo.

Cada parte pode ser ordenada de forma independente e, posteriormente, combinada para formar a lista ordenada final. Isto não só distribui a carga de trabalho, como também reduz o tempo total de execução. Esta abordagem é escalável, podendo ser ajustada para diferentes tamanhos de listas.

Problema 3

Para a simplificação do algoritmo de Catalan sem recurso ao cálculo dos factoriais é necessário reescrever a fórmula. A simplificação é a seguinte descrita:

$$C_n = \frac{(2n)!}{(n+1)!(n!)} \quad (2)$$

$$C_n = \frac{(2n)(2n-1)!}{(n+1)n! * n(n-1)!} \quad (3)$$

$$C_n = \frac{2(2n-1)!}{(n+1)n! * (n-1)!} \quad (4)$$

$$C_n = \frac{2(2n-1)(2n-2)!}{(n+1)n! * (n-1)(n-2)!} \quad (5)$$

...

A continuação da simplificação matemática mostraria que era possível obter o número de Catalan atual, usando o número de Catalan anterior.

Pelo link da Wikipédia disponibilizada pela equipa docente sabemos que:

$$C_0 = 1 \quad (6)$$

$$C_n = \frac{2(2n-1)}{n+1} * C_{n-1} \quad (7)$$

Que iremos simplificar para

$$C_n = \frac{4n-2}{n+1} * C_{n-1} \quad (8)$$

Para reduzir o número de operações.

Sabendo como se escreve

$$C_n \quad (9)$$

através de

$$C_{n-1} \quad (10)$$

é possível escrever a recursividade mútua do algoritmo.

O nosso **inic** será composto por um par (a,b) que será inicializado com (1,1). O loop é responsável por aplicar a função succ ao primeiro elemento e a executar a chamada recursiva no segundo elemento do **inic**. A primeira componente serve para guardar a iteração atual, já a segunda guarda o número de Catalan. Sendo assim, ficamos com o seguinte algoritmo:

$$cat = prj \cdot \text{for loop } inic$$

onde:

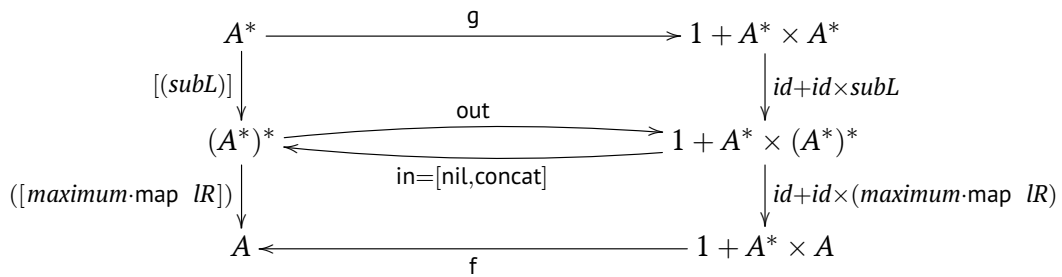
$$\text{loop } (a, b) = (a + 1, b * ((4 * a) - 2) \text{ 'div' } (a + 1))$$

$$inic = (1, 1)$$

$$prj = \pi_2$$

Problema 4

Diagrama para a resolução do problema:



$$lrh = (\text{maximum} \cdot \text{map } lR) \cdot \text{subL}$$

$$\text{subL } [] = []$$

$$\text{subL } xs = xs : \text{subL } (\text{tail } xs)$$

$$lR [] = 0$$

$$lR [x] = x$$

$$lR (x : y : xs)$$

$$| x > y = x$$

$$| x \leq y = x + lR (x : xs)$$

Index

\LaTeX , [4](#), [5](#)

bibtex, [5](#)

lhs2TeX, [4–6](#)

makeindex, [5](#)

pdflatex, [4](#)

Combinador “pointfree”

hylo

 Listas, [2](#)

split, [6](#)

Comissão Nacional de Eleições, [1](#)

 Método de Hondt, [1](#)

Cálculo de Programas, [1](#), [4](#), [6](#)

 Material Pedagógico, [4](#)

 LTree.hs, [2](#)

 mSort (‘merge sort’), [2](#)

Docker, [4](#)

 container, [4](#), [5](#)

Functor, [6](#)

Função

π_1 , [6–9](#)

π_2 , [6–9](#), [11](#)

for, [2](#), [3](#), [7](#), [11](#)

length, [10](#)

map, [7](#), [9–11](#)

succ, [8](#)

Haskell, [1](#), [4](#), [5](#)

 interpretador

 GHCi, [4](#), [5](#)

 Literate Haskell, [4](#)

Números de Catalan, [2](#), [8](#)

Números naturais (\mathbb{N}), [6](#), [7](#)

Programação

 dinâmica, [6](#)

 literária, [4](#), [6](#)

References

- [1] D.E. Knuth. *Literate Programming*. CSLI Lecture Notes Number 27. Stanford University Center for the Study of Language and Information, Stanford, CA, USA, 1992.
- [2] J.N. Oliveira. *Program Design by Calculation*, 2018. Draft of textbook in preparation. viii+297 pages. Informatics Department, University of Minho.