

# Cálculo de Programas

## Trabalho Prático

### MiEI+LCC — 2020/21

Departamento de Informática  
Universidade do Minho

Junho de 2021

Grupo nr.	47
a91695	Tiago dos Santos Silva Peixoto Carriço
a91637	Henrique Pinto Pires Lopes de Faria
a91635	Miguel Ângelo Alves de Freitas
a91697	Luís Filipe Fernandes Vilas

## 1 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 (conjunto de leis universais e seus corolários) e usam-se esses combinadores para construir programas *composicionalmente*, isto é, agregando programas já existentes.

Na sequência pedagógica dos planos de estudo dos dois 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.

## 2 Documentação

Para cumprir de forma integrada os objectivos enunciados acima 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 `cp2021t.pdf` que está a ler é já um exemplo de **programação literária**: foi gerado a partir do texto fonte `cp2021t.lhs`<sup>1</sup> que encontrará no **material pedagógico** desta disciplina descompactando o ficheiro `cp2021t.zip` e executando:

```
$ lhs2TeX cp2021t.lhs > cp2021t.tex
$ pdflatex cp2021t
```

em que **lhs2tex** é um pre-processor que faz “pretty printing” de código Haskell em **L<sup>A</sup>T<sub>E</sub>X** e que deve desde já instalar executando

```
$ cabal install lhs2tex --lib
```

Por outro lado, o mesmo ficheiro `cp2021t.lhs` é executável e contém o “kit” básico, escrito em **Haskell**, para realizar o trabalho. Basta executar

```
$ ghci cp2021t.lhs
```

---

<sup>1</sup>O suffixo ‘lhs’ quer dizer *literate Haskell*.

Abra o ficheiro `cp2021t.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.

### 3 Como realizar o trabalho

Este trabalho teórico-prático deve ser realizado por grupos de 3 (ou 4) 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 de trabalho por forma a poderem responder às questões que serão colocadas na *defesa oral* do relatório.

Em que consiste, então, o *relatório* a que se refere o parágrafo anterior? É a edição do texto que está a ser lido, preenchendo o anexo **D** 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 cp2021t.aux
$ makeindex cp2021t.idx
```

e recompilar o texto como acima se indicou. Dever-se-á ainda instalar o utilitário **QuickCheck**, que ajuda a validar programas em **Haskell** e a biblioteca **Gloss** para geração de gráficos 2D:

```
$ cabal install QuickCheck gloss --lib
```

Para testar uma propriedade **QuickCheck** *prop*, basta invocá-la com o comando:

```
> quickCheck prop
+++ OK, passed 100 tests.
```

Pode-se ainda controlar o número de casos de teste e sua complexidade, como o seguinte exemplo mostra:

```
> quickCheckWith stdArgs { maxSuccess = 200, maxSize = 10 } prop
+++ OK, passed 200 tests.
```

Qualquer programador tem, na vida real, de ler e analisar (muito!) código escrito por outros. No anexo **C** disponibiliza-se algum código **Haskell** relativo aos problemas que se seguem. Esse anexo deverá ser consultado e analisado à medida que isso for necessário.

#### 3.1 Stack

O **Stack** é um programa útil para criar, gerir e manter projetos em **Haskell**. Um projeto criado com o Stack possui uma estrutura de pastas muito específica:

- Os módulos auxiliares encontram-se na pasta *src*.
- O módulo principal encontra-se na pasta *app*.
- A lista de dependências externas encontra-se no ficheiro *package.yaml*.

Pode aceder ao **GHCI** utilizando o comando:

```
stack ghci
```

Garanta que se encontra na pasta mais externa **do projeto**. A primeira vez que correr este comando as dependências externas serão instaladas automaticamente.

Para gerar o PDF, garanta que se encontra na directoria *app*.

## Problema 1

Os tipos de dados algébricos estudados ao longo desta disciplina oferecem uma grande capacidade expressiva ao programador. Graças à sua flexibilidade, torna-se trivial implementar DSLs e até mesmo linguagens de programação.

Paralelamente, um tópico bastante estudado no âmbito de Deep Learning é a derivação automática de expressões matemáticas, por exemplo, de derivadas. Duas técnicas que podem ser utilizadas para o cálculo de derivadas são:

- *Symbolic differentiation*
- *Automatic differentiation*

*Symbolic differentiation* consiste na aplicação sucessiva de transformações (leia-se: funções) que sejam congruentes com as regras de derivação. O resultado final será a expressão da derivada.

O leitor atento poderá notar um problema desta técnica: a expressão inicial pode crescer de forma descontrolada, levando a um cálculo pouco eficiente. *Automatic differentiation* tenta resolver este problema, calculando o valor da derivada da expressão em todos os passos. Para tal, é necessário calcular o valor da expressão e o valor da sua derivada.

Vamos de seguida definir uma linguagem de expressões matemáticas simples e implementar as duas técnicas de derivação automática. Para isso, seja dado o seguinte tipo de dados,

```
data ExpAr a = X
  | N a
  | Bin BinOp (ExpAr a) (ExpAr a)
  | Un UnOp (ExpAr a)
  deriving (Eq, Show)
```

onde *BinOp* e *UnOp* representam operações binárias e unárias, respectivamente:

```
data BinOp = Sum
  | Product
  deriving (Eq, Show)
data UnOp = Negate
  | E
  deriving (Eq, Show)
```

O construtor *E* simboliza o exponencial de base *e*.

Assim, cada expressão pode ser uma variável, um número, uma operação binária aplicada às devidas expressões, ou uma operação unária aplicada a uma expressão. Por exemplo,

*Bin Sum X (N 10)*

designa  $x + 10$  na notação matemática habitual.

1. A definição das funções *inExpAr* e *baseExpAr* para este tipo é a seguinte:

```
inExpAr = [X, num_ops] where
  num_ops = [N, ops]
  ops = [bin, Un]
  bin (op, (a, b)) = Bin op a b
baseExpAr f g h j k l z = f + (g + (h × (j × k) + l × z))
```

Defina as funções *outExpAr* e *recExpAr*, e teste as propriedades que se seguem.

**Propriedade [QuickCheck] 1** *inExpAr* e *outExpAr* são testemunhas de um isomorfismo, isto é, *inExpAr* · *outExpAr* = *id* e *outExpAr* · *inExpAr* = *id*:

```
prop_in_out_idExpAr :: (Eq a) => ExpAr a -> Bool
prop_in_out_idExpAr = inExpAr · outExpAr ≡ id
prop_out_in_idExpAr :: (Eq a) => OutExpAr a -> Bool
prop_out_in_idExpAr = outExpAr · inExpAr ≡ id
```

2. Dada uma expressão aritmética e um escalar para substituir o  $X$ , a função

$$eval\_exp :: Floating a \Rightarrow a \rightarrow (ExpAr a) \rightarrow a$$

calcula o resultado da expressão. Na página 12 esta função está expressa como um catamorfismo. Defina o respectivo gene e, de seguida, teste as propriedades:

**Propriedade [QuickCheck] 2** A função *eval\_exp* respeita os elementos neutros das operações.

$$\begin{aligned} prop\_sum\_idr &:: (Floating a, Real a) \Rightarrow a \rightarrow ExpAr a \rightarrow Bool \\ prop\_sum\_idr a exp &= eval\_exp a exp \stackrel{?}{=} sum\_idr \textbf{ where} \\ sum\_idr &= eval\_exp a (Bin Sum exp (N 0)) \\ prop\_sum\_idl &:: (Floating a, Real a) \Rightarrow a \rightarrow ExpAr a \rightarrow Bool \\ prop\_sum\_idl a exp &= eval\_exp a exp \stackrel{?}{=} sum\_idl \textbf{ where} \\ sum\_idl &= eval\_exp a (Bin Sum (N 0) exp) \\ prop\_product\_idr &:: (Floating a, Real a) \Rightarrow a \rightarrow ExpAr a \rightarrow Bool \\ prop\_product\_idr a exp &= eval\_exp a exp \stackrel{?}{=} prod\_idr \textbf{ where} \\ prod\_idr &= eval\_exp a (Bin Product exp (N 1)) \\ prop\_product\_idl &:: (Floating a, Real a) \Rightarrow a \rightarrow ExpAr a \rightarrow Bool \\ prop\_product\_idl a exp &= eval\_exp a exp \stackrel{?}{=} prod\_idl \textbf{ where} \\ prod\_idl &= eval\_exp a (Bin Product (N 1) exp) \\ prop\_e\_id &:: (Floating a, Real a) \Rightarrow a \rightarrow Bool \\ prop\_e\_id a &= eval\_exp a (Un E (N 1)) \equiv expd 1 \\ prop\_negate\_id &:: (Floating a, Real a) \Rightarrow a \rightarrow Bool \\ prop\_negate\_id a &= eval\_exp a (Un Negate (N 0)) \equiv 0 \end{aligned}$$

**Propriedade [QuickCheck] 3** Negar duas vezes uma expressão tem o mesmo valor que não fazer nada.

$$\begin{aligned} prop\_double\_negate &:: (Floating a, Real a) \Rightarrow a \rightarrow ExpAr a \rightarrow Bool \\ prop\_double\_negate a exp &= eval\_exp a exp \stackrel{?}{=} eval\_exp a (Un Negate (Un Negate exp)) \end{aligned}$$

3. É possível otimizar o cálculo do valor de uma expressão aritmética tirando proveito dos elementos absorventes de cada operação. Implemente os genes da função

$$optimize\_eval :: (Floating a, Eq a) \Rightarrow a \rightarrow (ExpAr a) \rightarrow a$$

que se encontra na página 12 expressa como um hilomorfismo<sup>2</sup> e teste as propriedades:

**Propriedade [QuickCheck] 4** A função *optimize\_eval* respeita a semântica da função *eval*.

$$\begin{aligned} prop\_optimize\_respects\_semantics &:: (Floating a, Real a) \Rightarrow a \rightarrow ExpAr a \rightarrow Bool \\ prop\_optimize\_respects\_semantics a exp &= eval\_exp a exp \stackrel{?}{=} optimize\_eval a exp \end{aligned}$$

4. Para calcular a derivada de uma expressão, é necessário aplicar transformações à expressão original que respeitem as regras das derivadas:<sup>3</sup>

- Regra da soma:

$$\frac{d}{dx}(f(x) + g(x)) = \frac{d}{dx}(f(x)) + \frac{d}{dx}(g(x))$$

<sup>2</sup>Qual é a vantagem de implementar a função *optimize\_eval* utilizando um hilomorfismo em vez de utilizar um catamorfismo com um gene "inteligente"?

<sup>3</sup>Apesar da adição e multiplicação gozarem da propriedade comutativa, há que ter em atenção a ordem das operações por causa dos testes.

- Regra do produto:

$$\frac{d}{dx}(f(x)g(x)) = f(x) \cdot \frac{d}{dx}(g(x)) + \frac{d}{dx}(f(x)) \cdot g(x)$$

Defina o gene do catamorfismo que ocorre na função

$$sd :: Floating a \Rightarrow ExpAr a \rightarrow ExpAr a$$

que, dada uma expressão aritmética, calcula a sua derivada. Testes a fazer, de seguida:

**Propriedade [QuickCheck] 5** A função *sd* respeita as regras de derivação.

```
prop_const_rule :: (Real a, Floating a) => a -> Bool
prop_const_rule a = sd (N a) == N 0

prop_var_rule :: Bool
prop_var_rule = sd X == N 1

prop_sum_rule :: (Real a, Floating a) => ExpAr a -> ExpAr a -> Bool
prop_sum_rule exp1 exp2 = sd (Bin Sum exp1 exp2) == sum_rule where
  sum_rule = Bin Sum (sd exp1) (sd exp2)

prop_product_rule :: (Real a, Floating a) => ExpAr a -> ExpAr a -> Bool
prop_product_rule exp1 exp2 = sd (Bin Product exp1 exp2) == prod_rule where
  prod_rule = Bin Sum (Bin Product exp1 (sd exp2)) (Bin Product (sd exp1) exp2)

prop_e_rule :: (Real a, Floating a) => ExpAr a -> Bool
prop_e_rule exp = sd (Un E exp) == Bin Product (Un E exp) (sd exp)

prop_negate_rule :: (Real a, Floating a) => ExpAr a -> Bool
prop_negate_rule exp = sd (Un Negate exp) == Un Negate (sd exp)
```

5. Como foi visto, *Symbolic differentiation* não é a técnica mais eficaz para o cálculo do valor da derivada de uma expressão. *Automatic differentiation* resolve este problema calculando o valor da derivada em vez de manipular a expressão original.

Defina o gene do catamorfismo que ocorre na função

$$ad :: Floating a \Rightarrow a \rightarrow ExpAr a \rightarrow a$$

que, dada uma expressão aritmética e um ponto, calcula o valor da sua derivada nesse ponto, sem transformar manipular a expressão original. Testes a fazer, de seguida:

**Propriedade [QuickCheck] 6** Calcular o valor da derivada num ponto *r* via *ad* é equivalente a calcular a derivada da expressão e avalia-la no ponto *r*.

```
prop_congruent :: (Floating a, Real a) => a -> ExpAr a -> Bool
prop_congruent a exp = ad a exp == eval_exp a (sd exp)
```

## Problema 2

Nesta disciplina estudou-se como fazer **programação dinâmica** por cálculo, recorrendo à lei de recursividade mútua.<sup>4</sup>

Para o caso de funções sobre os números naturais ( $\mathbb{N}_0$ , com functor  $F X = 1 + X$ ) é fácil 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

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

<sup>4</sup>Lei (3.94) em [2], página 98.

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

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.<sup>5</sup>
- 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 aulas<sup>6</sup>, 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:

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

O que se pede então, nesta pergunta? Dada a fórmula que dá o *n*-ésimo **número de Catalan**,

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

derivar uma implementação de  $C_n$  que não calcule factoriais nenhuns. Isto é, derivar um ciclo-for

$$cat = \dots \cdot \text{for loop init where} \dots$$

que implemente esta função.

**Propriedade [QuickCheck] 7** A função proposta coincide com a definição dada:

$$prop\_cat = (\geq 0) \Rightarrow (catdef \equiv cat)$$

**Sugestão:** Começar por estudar muito bem o processo de cálculo dado no anexo B para o problema (semelhante) da função exponencial.

## Problema 3

As **curvas de Bézier**, designação dada em honra ao engenheiro **Pierre Bézier**, são curvas ubíquas na área de computação gráfica, animação e modelação. Uma curva de Bézier é uma curva paramétrica, definida por um conjunto  $\{P_0, \dots, P_N\}$  de pontos de controlo, onde  $N$  é a ordem da curva.

O algoritmo de *De Casteljau* é um método recursivo capaz de calcular curvas de Bézier num ponto. Apesar de ser mais lento do que outras abordagens, este algoritmo é numericamente mais estável, trocando velocidade por correção.

<sup>5</sup>Podem obviamente usar-se outros símbolos, mas numa primeira leitura dá jeito usarem-se tais nomes.

<sup>6</sup>Secção 3.17 de [2] e tópico **Recursividade mútua** nos vídeos das aulas teóricas.



Figura 1: Exemplos de curvas de Bézier retirados da [Wikipedia](#).

De forma sucinta, o valor de uma curva de Bézier de um só ponto  $\{P_0\}$  (ordem 0) é o próprio ponto  $P_0$ . O valor de uma curva de Bézier de ordem  $N$  é calculado através da interpolação linear da curva de Bézier dos primeiros  $N - 1$  pontos e da curva de Bézier dos últimos  $N - 1$  pontos.

A interpolação linear entre 2 números, no intervalo  $[0, 1]$ , é dada pela seguinte função:

```
linear1d :: ℚ → ℚ → OverTime ℚ
linear1d a b = formula a b where
  formula :: ℚ → ℚ → Float → ℚ
  formula x y t = ((1.0 :: ℚ) - (toℚ t)) * x + (toℚ t) * y
```

A interpolação linear entre 2 pontos de dimensão  $N$  é calculada através da interpolação linear de cada dimensão.

O tipo de dados *NPoint* representa um ponto com  $N$  dimensões.

```
type NPoint = [ℚ]
```

Por exemplo, um ponto de 2 dimensões e um ponto de 3 dimensões podem ser representados, respetivamente, por:

```
p2d = [1.2, 3.4]
p3d = [0.2, 10.3, 2.4]
```

O tipo de dados *OverTime a* representa um termo do tipo  $a$  num dado instante (dado por um *Float*).

```
type OverTime a = Float → a
```

O anexo C tem definida a função

```
calcLine :: NPoint → (NPoint → OverTime NPoint)
```

que calcula a interpolação linear entre 2 pontos, e a função

```
deCasteljau :: [NPoint] → OverTime NPoint
```

que implementa o algoritmo respectivo.

1. Implemente *calcLine* como um catamorfismo de listas, testando a sua definição com a propriedade:

**Propriedade [QuickCheck] 8** Definição alternativa.

```
prop_calcLine_def :: NPoint → NPoint → Float → Bool
prop_calcLine_def p q d = calcLine p q d ≡ zipWithM linear1d p q d
```

2. Implemente a função *deCasteljau* como um hilomorfismo, testando agora a propriedade:

**Propriedade [QuickCheck] 9** *Curvas de Bézier são simétricas.*

```
prop_bezier_sym :: [[Q]] → Gen Bool
prop_bezier_sym l = all (<Δ) · calc_difs · bezs ($) elements ps where
  calc_difs = (λ(x, y) → zipWith (λw v → if w ≥ v then w - v else v - w) x y)
  bezs t = (deCasteljau l t, deCasteljau (reverse l) (fromQ (1 - (toQ t))))
  Δ = 1e-2
```

3. Corra a função `runBezier` e aprecie o seu trabalho<sup>7</sup> clicando na janela que é aberta (que contém, a verde, um ponto inicial) com o botão esquerdo do rato para adicionar mais pontos. A tecla `Delete` apaga o ponto mais recente.

## Problema 4

Seja dada a fórmula que calcula a média de uma lista não vazia  $x$ ,

$$avg\ x = \frac{1}{k} \sum_{i=1}^k x_i \quad (2)$$

onde  $k = length\ x$ . Isto é, para sabermos a média de uma lista precisamos de dois catamorfismos: o que faz o somatório e o que calcula o comprimento a lista. Contudo, é fácil de ver que

$$avg\ [a] = a$$

$$avg\ (a : x) = \frac{1}{k+1} (a + \sum_{i=1}^k x_i) = \frac{a + k(avg\ x)}{k+1} \text{ para } k = length\ x$$

Logo `avg` está em recursividade mútua com `length` e o par de funções pode ser expresso por um único catamorfismo, significando que a lista apenas é percorrida uma vez.

1. Recorra à lei de recursividade mútua para derivar a função `avg_aux = ([b, q])` tal que `avg_aux = (avg, length)` em listas não vazias.
2. Generalize o raciocínio anterior para o cálculo da média de todos os elementos de uma `LTree` recorrendo a uma única travessia da árvore (i.e. catamorfismo).

Verifique as suas funções testando a propriedade seguinte:

**Propriedade [QuickCheck] 10** *A média de uma lista não vazia e de uma `LTree` com os mesmos elementos coincide, a menos de um erro de 0.1 milésimas:*

```
prop_avg = nonempty ⇒ diff ≤ 0.000001 where
  diff l = avg l - (avgLTree · genLTree) l
  genLTree = ([lsplit])
  nonempty = (>[])
```

## Problema 5

(NB: Esta questão é **opcional** e funciona como **valorização** apenas para os alunos que desejarem fazê-la.)

Existem muitas linguagens funcionais para além do `Haskell`, que é a linguagem usada neste trabalho prático. Uma delas é o `F#` da Microsoft. Na directoria `fsharp` encontram-se os módulos `Cp`, `Nat` e `LTree` codificados em `F#`. O que se pede é a biblioteca `BTree` escrita na mesma linguagem.

Modo de execução: o código que tiverem produzido nesta pergunta deve ser colocado entre o `\begin{verbatim}` e o `\end{verbatim}` da correspondente parte do anexo `D`. Para além disso, os grupos podem demonstrar o código na oral.

<sup>7</sup>A representação em Gloss é uma adaptação de um `projeto` de Harold Cooper.



# Anexos

## A Como exprimir cálculos e diagramas em LaTeX/lhs2tex

Como primeiro exemplo, estudar o texto fonte deste trabalho para obter o efeito:<sup>8</sup>

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

Os diagramas podem ser produzidos recorrendo à *package* L<sup>A</sup>T<sub>E</sub>X *xymatrix*, por exemplo:

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

## B Programação dinâmica por recursividade múltipla

Neste anexo dão-se os detalhes da resolução do Exercício 3.30 dos apontamentos da disciplina<sup>9</sup>, onde se pretende implementar um ciclo que implemente o cálculo da aproximação até  $i = n$  da função exponencial  $\exp x = e^x$ , via série de Taylor:

$$\exp x = \sum_{i=0}^{\infty} \frac{x^i}{i!} \quad (3)$$

Seja  $e\ x\ n = \sum_{i=0}^n \frac{x^i}{i!}$  a função que dá essa aproximação. É fácil de ver que  $e\ x\ 0 = 1$  e que  $e\ x\ (n+1) = e\ x\ n + \frac{x^{n+1}}{(n+1)!}$ . Se definirmos  $h\ x\ n = \frac{x^{n+1}}{(n+1)!}$  teremos  $e\ x$  e  $h\ x$  em recursividade mútua. Se repetirmos o processo para  $h\ x\ n$  etc obteremos no total três funções nessa mesma situação:

$$\begin{aligned}
 e\ x\ 0 &= 1 \\
 e\ x\ (n+1) &= h\ x\ n + e\ x\ n \\
 h\ x\ 0 &= x \\
 h\ x\ (n+1) &= x / (s\ n) * h\ x\ n \\
 s\ 0 &= 2 \\
 s\ (n+1) &= 1 + s\ n
 \end{aligned}$$

Segundo a *regra de algibeira* descrita na página 3.1 deste enunciado, ter-se-á, de imediato:

$$\begin{aligned}
 e'\ x &= prj \cdot \text{for loop init where} \\
 init &= (1, x, 2) \\
 loop\ (e, h, s) &= (h + e, x / s * h, 1 + s) \\
 prj\ (e, h, s) &= e
 \end{aligned}$$

<sup>8</sup>Exemplos tirados de [2].

<sup>9</sup>Cf. [2], página 102.

## C Código fornecido

### Problema 1

```
expd :: Floating a => a -> a
expd = Prelude.exp
type OutExpAr a = () + (a + ((BinOp, (ExpAr a, ExpAr a)) + (UnOp, ExpAr a)))
```

### Problema 2

Definição da série de Catalan usando factoriais (1):

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

Oráculo para inspecção dos primeiros 26 números de Catalan<sup>10</sup>:

```
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
]
```

### Problema 3

Algoritmo:

```
deCasteljau :: [NPoint] -> OverTime NPoint
deCasteljau [] = nil
deCasteljau [p] = p
deCasteljau l = λpt -> (calcLine (p pt) (q pt)) pt where
  p = deCasteljau (init l)
  q = deCasteljau (tail l)
```

Função auxiliar:

```
calcLine :: NPoint -> (NPoint -> OverTime NPoint)
calcLine [] = nil
calcLine (p : x) = g p (calcLine x) where
  g :: (Q, NPoint -> OverTime NPoint) -> (NPoint -> OverTime NPoint)
  g (d, f) l = case l of
    [] -> nil
    (x : xs) -> λz -> concat $ (sequenceA [singl · linear1d d x, f xs]) z
```

2D:

```
bezier2d :: [NPoint] -> OverTime (Float, Float)
bezier2d [] = (0, 0)
bezier2d l = λz -> (fromQ × fromQ) · (λ[x, y] -> (x, y)) $ ((deCasteljau l) z)
```

Modelo:

```
data World = World { points :: [NPoint]
  , time :: Float
  }
initW :: World
initW = World [] 0
```

---

<sup>10</sup>Fonte: [Wikipedia](#).

```

tick :: Float → World → World
tick dt world = world { time = (time world) + dt }

actions :: Event → World → World
actions (EventKey (MouseButton LeftButton) Down _ p) world =
  world { points = (points world) ++ [(λ(x,y) → map toQ [x,y]) p] }
actions (EventKey (SpecialKey KeyDelete) Down _ _) world =
  world { points = cond (≡ []) id init (points world) }
actions _ world = world

scaleTime :: World → Float
scaleTime w = (1 + cos (time w)) / 2

bezier2dAtTime :: World → (Float, Float)
bezier2dAtTime w = (bezier2dAt w) (scaleTime w)

bezier2dAt :: World → OverTime (Float, Float)
bezier2dAt w = bezier2d (points w)

thicCirc :: Picture
thicCirc = ThickCircle 4 10

ps :: [Float]
ps = map fromQ ps' where
  ps' :: [Q]
  ps' = [0, 0.01 .. 1] -- interval

```

Gloss:

```

picture :: World → Picture
picture world = Pictures
  [ animateBezier (scaleTime world) (points world)
  , Color white · Line · map (bezier2dAt world) $ ps
  , Color blue · Pictures $ [ Translate (fromQ x) (fromQ y) thicCirc | [x,y] ← points world ]
  , Color green $ Translate cx cy thicCirc
  ] where
  (cx, cy) = bezier2dAtTime world

```

Animação:

```

animateBezier :: Float → [NPoint] → Picture
animateBezier _ [] = Blank
animateBezier _ [_] = Blank
animateBezier t l = Pictures
  [ animateBezier t (init l)
  , animateBezier t (tail l)
  , Color red · Line $ [a, b]
  , Color orange $ Translate ax ay thicCirc
  , Color orange $ Translate bx by thicCirc
  ] where
  a@(ax, ay) = bezier2d (init l) t
  b@(bx, by) = bezier2d (tail l) t

```

Propriedades e main:

```

runBezier :: IO ()
runBezier = play (InWindow "Bézier" (600,600) (0,0))
  black 50 initW picture actions tick

runBezierSym :: IO ()
runBezierSym = quickCheckWith (stdArgs { maxSize = 20, maxSuccess = 200 }) prop_bezier_sym

```

Compilação e execução dentro do interpretador:<sup>11</sup>

```

main = runBezier
run = do { system "ghc cp2021t"; system "./cp2021t" }

```

---

<sup>11</sup>Pode ser útil em testes envolvendo **Gloss**. Nesse caso, o teste em causa deve fazer parte de uma função *main*.

## QuickCheck

Código para geração de testes:

```
instance Arbitrary UnOp where
  arbitrary = elements [Negate, E]
instance Arbitrary BinOp where
  arbitrary = elements [Sum, Product]
instance (Arbitrary a) => Arbitrary (ExpAr a) where
  arbitrary = do
    binop <- arbitrary
    unop <- arbitrary
    exp1 <- arbitrary
    exp2 <- arbitrary
    a <- arbitrary
    frequency · map (id × pure) $ [(20, X), (15, N a), (35, Bin binop exp1 exp2), (30, Un unop exp1)]
infixr 5  $\stackrel{?}{=}$ 
( $\stackrel{?}{=}$ ) :: Real a => a -> a -> Bool
( $\stackrel{?}{=}$ ) x y = (to $_{\mathbb{Q}}$  x) == (to $_{\mathbb{Q}}$  y)
```

## Outras funções auxiliares

Lógicas:

```
infixr 0 =>
(=>) :: (Testable prop) => (a -> Bool) -> (a -> prop) -> a -> Property
p => f =  $\lambda$ a -> p a => f a
infixr 0 <=>
(<=>) :: (a -> Bool) -> (a -> Bool) -> a -> Property
p <=> f =  $\lambda$ a -> (p a => property (f a)) .&&. (f a => property (p a))
infixr 4  $\equiv$ 
( $\equiv$ ) :: Eq b => (a -> b) -> (a -> b) -> (a -> Bool)
f  $\equiv$  g =  $\lambda$ a -> f a  $\equiv$  g a
infixr 4  $\leq$ 
( $\leq$ ) :: Ord b => (a -> b) -> (a -> b) -> (a -> Bool)
f  $\leq$  g =  $\lambda$ a -> f a  $\leq$  g a
infixr 4  $\wedge$ 
( $\wedge$ ) :: (a -> Bool) -> (a -> Bool) -> (a -> Bool)
f  $\wedge$  g =  $\lambda$ a -> (f a)  $\wedge$  (g a)
```

## D 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, diagramas e/ou outras funções auxiliares que sejam necessárias.

Valoriza-se a escrita de *pouco* código que corresponda a soluções simples e elegantes.

### Problema 1

São dadas:

```
cataExpAr g = g · recExpAr (cataExpAr g) · outExpAr
anaExpAr g = inExpAr · recExpAr (anaExpAr g) · g
hyloExpAr h g = cataExpAr h · anaExpAr g
```

```

eval_exp :: Floating a => a -> (ExpAr a) -> a
eval_exp a = cataExpAr (g_eval_exp a)
optimize_eval :: (Floating a, Eq a) => a -> (ExpAr a) -> a
optimize_eval a = hyloExpAr (gopt a) clean
sd :: Floating a => ExpAr a -> ExpAr a
sd = π2 · cataExpAr sd_gen
ad :: Floating a => a -> ExpAr a -> a
ad v = π2 · cataExpAr (ad_gen v)

```

Definir:

A solução para  $outExpAr$  pode ser obtida através da igualdade  $outExpAr \cdot inExpAr = id$ :

$$\begin{aligned}
& outExpAr \cdot inExpAr = id \\
\equiv & \quad \{ \text{Def-}inExpAr \} \\
& outExpAr \cdot [\underline{X}, num\_ops] = id \\
\equiv & \quad \{ \text{Universal-+; Def-}num\_ops; \text{Fusão-+} \} \\
& \begin{cases} outExpAr \cdot \underline{X} = i_1 \\ outExpAr \cdot [N, ops] = i_2 \end{cases} \\
\equiv & \quad \{ \text{Universal-+; Def-}ops; \text{Fusão-+} \} \\
& \begin{cases} outExpAr \cdot \underline{X} = i_1 \\ outExpAr \cdot N = i_2 \cdot i_1 \\ outExpAr \cdot [bin, \widehat{Un}] = i_2 \cdot i_2 \end{cases} \\
\equiv & \quad \{ \text{Universal-+; Fusão-+} \} \\
& \begin{cases} outExpAr \cdot \underline{X} = i_1 \\ outExpAr \cdot N = i_2 \cdot i_1 \\ outExpAr \cdot bin = i_2 \cdot i_2 \cdot i_1 \\ outExpAr \cdot \widehat{Un} = i_2 \cdot i_2 \cdot i_2 \end{cases} \\
\equiv & \quad \{ \text{Introdução de variáveis} \} \\
& \begin{cases} outExpAr \ X = i_1 \ () \\ outExpAr \ (N \ x) = i_2 \ (i_1 \ x) \\ outExpAr \ (bin \ op \ a \ b) = i_2 \ (i_2 \ (i_1 \ (op, (a, b)))) \\ outExpAr \ (\widehat{Un} \ (x, y)) = i_2 \ (i_2 \ (i_2 \ (x, y))) \end{cases} \\
\equiv & \quad \{ \text{Def-}bin; \text{Def-}\widehat{\cdot} \}
\end{aligned}$$

```

outExpAr X = i1 ()
outExpAr (N x) = i2 (i1 x)
outExpAr (Bin op a b) = i2 (i2 (i1 (op, (a, b))))
outExpAr (Un x y) = i2 (i2 (i2 (x, y)))
--
recExpAr f = id + (id + ((id × (f × f)) + (id × f)))
--
g_eval_exp a = [a, num_ops] where
  num_ops = [id, ops]
  ops = [bina, unar]
  bina (Sum, par) = (+) par
  bina (Product, par) = (*) par
  unar Negate = negate
  unar E = expd
--
clean (Bin Product (N 0) b) = i2 (i1 0)

```

```

clean (Bin Product a (N 0)) = i2 (i1 0)
clean (Bin Product (N 1) b) = clean b
clean (Bin Product a (N 1)) = clean a
clean (Bin Sum (N 0) b) = clean b
clean (Bin Sum a (N 0)) = clean a
clean (Un Negate (Un Negate a)) = clean a
clean (Un E (N 0)) = i2 (i1 1)
clean a = outExpAr a
--
gopt a = g_eval_exp a

sd_gen :: Floating a =>
  () + (a + ((BinOp, ((ExpAr a, ExpAr a), (ExpAr a, ExpAr a))) + (UnOp, (ExpAr a, ExpAr a))))
  → (ExpAr a, ExpAr a)
sd_gen = [(X, N 1), q] where
  q = [constante, ops]
  constante = ⟨N · id, N · 0⟩
  ops = [binar, unar]
  unar (E, (x1, dx1)) = (Un E x1, Bin Product (Un E x1) dx1)
  unar (Negate, (x1, dx1)) = (Un Negate x1, Un Negate dx1)
  binar (Sum, ((x1, dx1), (x2, dx2))) = (Bin Sum x1 x2, Bin Sum dx1 dx2)
  binar x = (esq x, dir x)
  esq (Product, ((x1, dx1), (x2, dx2))) = Bin Product x1 x2
  dir (Product, ((x1, dx1), (x2, dx2))) = Bin Sum (Bin Product x1 dx2) (Bin Product dx1 x2)

ad_gen v = [b, q] where
  b () = (v, 1)
  q = [constante, ops]
  constante x = (x, 0)
  ops = [binar, unar]
  binar (Sum, ((x1, dx1), (x2, dx2))) = (x1 + x2, dx1 + dx2)
  binar (Product, ((x1, dx1), (x2, dx2))) = (x1 * x2, (x1 * dx2) + (dx1 * x2))
  unar (Negate, (x1, dx1)) = (-x1, -dx1)
  unar (E, (x1, dx1)) = (expd x1, dx1 * expd x1)

```

## Problema 2

Definir

```

loop = b × d where
  succ2 = (2+)
  b = ⟨succ · π1, mul · (succ × id)⟩
  d = ⟨succ2 · π1, mul · (mul · ⟨succ2, succ⟩ × id)⟩
inic = ((0, 1), (0, 1))
prj ((s1, facN), (s2, fac2N)) = fac2N ÷ (facN * (s1 + 1) * facN)

```

por forma a que

```
cat = prj · for loop inic
```

seja a função pretendida. **NB:** usar divisão inteira. Apresentar de seguida a justificação da solução encontrada.

### Justificação:

O valor de n! pode ser obtido pelas funções:

$$\begin{cases} facN\ 0 = 1 \\ facN\ (n + 1) = s1\ (n + 1) * facN\ n \\ s1\ 0 = 0 \\ s1\ (n + 1) = 1 + s1\ n \end{cases}$$

$$\begin{aligned}
&\equiv \{ \text{Igualdade Extensional} \} \\
&\quad \left\{ \begin{array}{l} facN \cdot zero = one \\ facN \cdot succ = mul \cdot (succ \times id) \cdot \langle s1, facN \rangle \\ s1 \cdot zero = zero \\ s1 \cdot succ = succ \cdot \pi_1 \cdot \langle s1, facN \rangle \end{array} \right. \\
&\equiv \{ \text{Eq-+; Def-in} \} \\
&\quad \left\{ \begin{array}{l} facN \cdot \mathbf{in} = [one, mul \cdot (succ \times id) \cdot \langle s1, facN \rangle] \\ s1 \cdot \mathbf{in} = [zero, succ \cdot \pi_1 \cdot \langle s1, facN \rangle] \end{array} \right. \\
&\equiv \{ \text{Absorção-+} \} \\
&\quad \left\{ \begin{array}{l} facN \cdot \mathbf{in} = [one, mul \cdot (succ \times id)] \cdot (id + \langle s1, facN \rangle) \\ s1 \cdot \mathbf{in} = [zero, succ \cdot \pi_1] \cdot (id + \langle s1, facN \rangle) \end{array} \right. \\
&\equiv \{ \text{Fokkinga} \} \\
&\quad \langle s1, facN \rangle = \langle \langle [zero, succ \cdot \pi_1 \cdot \langle s1, facN \rangle], [one, mul \cdot (succ \times id) \cdot \langle s1, facN \rangle] \rangle \rangle \\
&\equiv \{ \text{Lei da Troca} \} \\
&\quad \langle s1, facN \rangle = \langle \langle [zero, one], \langle succ \cdot \pi_1, mul \cdot (succ \times id) \rangle \rangle \rangle
\end{aligned}$$

Este mesmo processo pode ser usado para obter (2n)!:

(Usaremos uma auxiliar  $succ2 = (2+)$ )

$$\langle s2, fac2N \rangle = \langle \langle [zero, one], \langle succ2 \cdot \pi_1, mul \cdot (mul \cdot \langle succ2, succ \rangle \times id) \rangle \rangle \rangle$$

Temos então:

$$\begin{aligned}
&\quad \left\{ \begin{array}{l} \langle s1, facN \rangle = \langle \langle [zero, one], \langle succ \cdot \pi_1, mul \cdot (succ \times id) \rangle \rangle \rangle \\ \langle s2, fac2N \rangle = \langle \langle [zero, one], \langle succ2 \cdot \pi_1, mul \cdot (mul \cdot \langle succ2, succ \rangle \times id) \rangle \rangle \rangle \end{array} \right. \\
&\equiv \{ \text{Eq-x} \} \\
&\quad \left\{ \begin{array}{l} \langle \langle s1, facN \rangle, \langle s2, fac2N \rangle \rangle = \langle \langle [a, b], [a, d] \rangle \rangle \\ a = \langle zero, one \rangle \\ b = \langle succ \cdot \pi_1, mul \cdot (succ \times id) \rangle \\ d = \langle succ2 \cdot \pi_1, mul \cdot (mul \cdot \langle succ2, succ \rangle \times id) \rangle \end{array} \right. \\
&\equiv \{ \text{Banana-split} \} \\
&\quad \left\{ \begin{array}{l} \langle \langle s1, facN \rangle, \langle s2, fac2N \rangle \rangle = \langle \langle [a, b] \times [a, d], \langle F \pi_1, F \pi_2 \rangle \rangle \rangle \\ a = \langle zero, one \rangle \\ b = \langle succ \cdot \pi_1, mul \cdot (succ \times id) \rangle \\ d = \langle succ2 \cdot \pi_1, mul \cdot (mul \cdot \langle succ2, succ \rangle \times id) \rangle \end{array} \right. \\
&\equiv \{ \text{Absorção-x} \} \\
&\quad \left\{ \begin{array}{l} \langle \langle s1, facN \rangle, \langle s2, fac2N \rangle \rangle = \langle \langle [a, b] \cdot F \pi_1, [a, d] \cdot F \pi_2 \rangle \rangle \\ a = \langle zero, one \rangle \\ b = \langle succ \cdot \pi_1, mul \cdot (succ \times id) \rangle \\ d = \langle succ2 \cdot \pi_1, mul \cdot (mul \cdot \langle succ2, succ \rangle \times id) \rangle \end{array} \right. \\
&\equiv \{ \text{Definição do functor de números naturais} \} \\
&\quad \left\{ \begin{array}{l} \langle \langle s1, facN \rangle, \langle s2, fac2N \rangle \rangle = \langle \langle [a, b] \cdot (id + \pi_1), [a, d] \cdot (id + \pi_2) \rangle \rangle \\ a = \langle zero, one \rangle \\ b = \langle succ \cdot \pi_1, mul \cdot (succ \times id) \rangle \\ d = \langle succ2 \cdot \pi_1, mul \cdot (mul \cdot \langle succ2, succ \rangle \times id) \rangle \end{array} \right. \\
&\equiv \{ \text{Absorção-+} \} \\
&\quad \left\{ \begin{array}{l} \langle \langle s1, facN \rangle, \langle s2, fac2N \rangle \rangle = \langle \langle [a, b \cdot \pi_1], [a, d \cdot \pi_2] \rangle \rangle \\ a = \langle zero, one \rangle \\ b = \langle succ \cdot \pi_1, mul \cdot (succ \times id) \rangle \\ d = \langle succ2 \cdot \pi_1, mul \cdot (mul \cdot \langle succ2, succ \rangle \times id) \rangle \end{array} \right.
\end{aligned}$$

$$\begin{aligned}
&\equiv \{ \text{Lei da Troca} \} \\
&\quad \left\{ \begin{array}{l} \langle \langle s1, facN \rangle, \langle s2, fac2N \rangle \rangle = ([\langle a, a \rangle, \langle b \cdot \pi_1, d \cdot \pi_2 \rangle]) \\ a = \langle zero, one \rangle \\ b = \langle succ \cdot \pi_1, mul \cdot (succ \times id) \rangle \\ d = \langle succ2 \cdot \pi_1, mul \cdot (mul \cdot \langle succ2, succ \rangle \times id) \rangle \end{array} \right\} \\
&\equiv \{ \text{Def-for} \} \\
&\quad \left\{ \begin{array}{l} \langle \langle s1, facN \rangle, \langle s2, fac2N \rangle \rangle = \text{for } \langle b \cdot \pi_1, d \cdot \pi_2 \rangle ((0, 1), (0, 1)) \\ b = \langle succ \cdot \pi_1, mul \cdot (succ \times id) \rangle \\ d = \langle succ2 \cdot \pi_1, mul \cdot (mul \cdot \langle succ2, succ \rangle \times id) \rangle \end{array} \right\}
\end{aligned}$$

Daqui, podemos retirar que  $loop = \langle b \cdot \pi_1, d \cdot \pi_2 \rangle = b \times d$  e  $inic = ((0, 1), (0, 1))$ .

### Problema 3

```

calcLine :: NPoint → (NPoint → OverTime NPoint)
calcLine l l2 d = (π1 · (cataList (g d)) · zip l) l2 where
  g d = [b d, q]
  b d () = ([], d)
  q ((a, b), (l, d)) = (linear1d a b d : l, d)
  --
deCasteljau :: [NPoint] → OverTime NPoint
deCasteljau l t = (hyloAlgForm t alg coalg) l where
  coalg [x] = i2 ((x, x), [])
  coalg [] = i1 ()
  coalg (x : y : []) = i2 ((x, y), [])
  coalg (x : y : t) = i2 ((x, y), y : t)
  alg t = [nil, algAux t]
  algAux t ((x, y), r) = calcLine x y t : r
hyloAlgForm t alg coalg = (auxCasteljau t) · hyloList (alg t) coalg where
  auxCasteljau a [x] = x
  auxCasteljau a l = (hyloAlgForm a alg coalg) l

```

### Problema 4

Solução para listas não vazias:

```

avg = π1 · avg_aux

avg_aux = cataList [b, q] where
  b () = (0, 0)
  q (x, (a, b)) = ((x + a * b) / (b + 1), b + 1)

```

Solução para árvores de tipo **LTree**:

```

avgLTree = π1 · ([gene]) where
  gene = [b, q]
  b = ⟨id, 1⟩
  q fs = (top fs / bottom fs, bottom fs)
  top (f1, f2) = (∗) f1 + (∗) f2
  bottom = (∗) · (π2 × π2)

```



## Problema 5

Inserir em baixo o código **F#** desenvolvido, entre `\begin{verbatim}` e `\end{verbatim}`:

```
module BTree

open Cp

// (1) Datatype Definition -----

type BTree<'a> = Empty | Node of 'a * (BTree<'a> * BTree<'a>)

let inBTree x = either (konst Empty) Node x

let outBTree x =
    match x with
    | Empty -> i1 ()
    | Node (a, (b1,b2)) -> i2 (a, (b1,b2))

// (2) Ana + cata + hylo -----

let baseBTree g f x = (id -|- (g >< (f >< f))) x

let recBTree f x = (baseBTree id f) x          // Isto é, id -|- (id >< (f >< f))

let rec cataBTree a x = (a << (recBTree (cataBTree a)) << outBTree) x

let rec anaBTree a x = (inBTree << (recBTree (anaBTree a)) << a) x

let hyloBTree a c x = (cataBTree a << anaBTree c) x

// (3) Map -----

// instance Functor BTree
//     where fmap f = cataBTree (inBTree . baseBTree f id)
let fmap f x = (cataBTree (inBTree << baseBTree f id)) x

// (4) Examples -----

// (4.1) Inversion (mirror) -----

let invBTree x = cataBTree (inBTree << (id -|- (id >< swap))) x

// (4.2) Counting -----

let countBTree x = cataBTree (either zero (succ << uncurry (+) << p2)) x

// (4.3) Serialization -----

let inord x =
    let join (x, (l,r)) = l @ [x] @ r
    in either nil join x

let inordt x = (cataBTree inord) x

let preord x =
    let join (x, (l,r)) = [x] @ l @ r
    in either nil join x
```

```

let preordt x = (cataBTree preord) x

let postordt x =
  let join (x,(l,r)) = l @ r @ [x]
  in (cataBTree (either nil join)) x

// (4.4) Quicksort -----

let rec part p x =
  match x with
  | [] -> ([],[])
  | (h::t) ->
    let (s,l) = part p t
    in if p h then (h::s,l) else (s,h::l)

let qsep x =
  match x with
  | [] -> i1 ()
  | (h::t) ->
    let y = part ((<) h) t
    in i2 (h,y)

let qSort x = (hyloBTree inord qsep) x

// (4.5) Traces -----

let cons a t = a::t

let union x y = x @ y

let traces x =
  let tunion (a,(l,r)) = (List.map (cons a) l) @ (List.map (cons a) r)
  in cataBTree (either (konst [[]]) tunion) x

// (4.6) Towers of Hanoi -----

let present x = inord x

let strategy x =
  match x with
  | (d,0) -> i1 ()
  | (d,n) -> i2 ((n-1,d),((not d,n-1),(not d,n-1)))

let hanoi x = (hyloBTree present strategy) x

// (5) Depth and balancing (using mutual recursion) -----

let balDepth x =
  let h (a,((b1,b2),(d1,d2))) = (b1 && b2 && abs(d1-d2)<=1,1+max d1 d2)
  in
    let f ((b1,d1),(b2,d2)) = ((b1,b2),(d1,d2))
    in
      let g x = (either (konst (true,1)) (h << (id >< f))) x
      in cataBTree g x

let balBTree x = (p1 << balDepth) x

let depthBTree x = (p2 << balDepth) x

```

```
// Fim do module BTree
```

# Índice

L<sup>A</sup>T<sub>E</sub>X, [1](#)

**bibtex**, [2](#)

    lhs2TeX, [1](#)

    makeindex, [2](#)

Cálculo de Programas, [1](#), [2](#), [5](#)

    Material Pedagógico, [1](#)

        BTree.hs, [8](#)

        Cp.hs, [8](#)

        LTree.hs, [8](#), [14](#)

        Nat.hs, [8](#)

Combinador “pointfree”

    cata, [8](#), [9](#)

    either, [3](#), [8](#), [13](#), [14](#)

Curvas de Bézier, [6](#), [7](#)

Deep Learning), [3](#)

DSL (linguagem específica para domínio), [3](#)

F#, [8](#), [14](#)

Função

$\pi_1$ , [6](#), [9](#), [14](#)

$\pi_2$ , [9](#), [13](#)

    for, [6](#), [9](#), [14](#)

    length, [8](#)

    map, [11](#), [12](#)

    uncurry, [3](#), [13](#), [14](#)

Functor, [5](#), [11](#)

Haskell, [1](#), [2](#), [8](#)

    Gloss, [2](#), [11](#)

    interpretador

        GHCi, [2](#)

    Literate Haskell, [1](#)

    QuickCheck, [2](#)

    Stack, [2](#)

Números de Catalan, [6](#), [10](#)

Números naturais ( $N$ ), [5](#), [6](#), [9](#)

Programação

    dinâmica, [5](#)

    literária, [1](#)

Racionais, [7](#), [8](#), [10–12](#)

U.Minho

    Departamento de Informática, [1](#)

## Referências

- [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.