

Cálculo de Programas

Trabalho Prático

MiEI+LCC — 2020/21

Departamento de Informática
Universidade do Minho

Junho de 2021

Grupo nr.	95
a93325	Henrique Costa
a93163	José Pedro Fernandes
a93246	Matilde Bravo
a93272	Pedro Alves

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**” [?], 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`¹ 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-processador que faz “pretty printing” de código Haskell em **L^AT_EX** 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
```

¹O 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 ?? 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 ?? 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 módulos 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 ?? 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 ?? expressa como um hilomorfismo² 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:³

- Regra da soma:

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

²Qual é a vantagem de implementar a função *optimize_eval* utilizando um hilomorfismo em vez de utilizar um catamorfismo com um gene "inteligente"?

³Apesar 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.⁴

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

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

⁴Lei (3.94) em [?], página 98.

$$f\ 0 = 1$$

$$f\ (n + 1) = fib\ n + f\ n$$

Obter-se-á de imediato

$$fib' = \pi_1 \cdot \text{for loop init where}$$

$$\text{loop } (fib, f) = (f, fib + f)$$

$$\text{init} = (1, 1)$$

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.⁵
- 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 aulas⁶, de $f\ x = ax^2 + bx + c$ derivam-se duas funções mutuamente recursivas:

$$f\ 0 = c$$

$$f\ (n + 1) = f\ n + k\ n$$

$$k\ 0 = a + b$$

$$k\ (n + 1) = k\ n + 2\ a$$

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

$$f'\ a\ b\ c = \pi_1 \cdot \text{for loop init where}$$

$$\text{loop } (f, k) = (f + k, k + 2 * a)$$

$$\text{init} = (c, a + b)$$

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

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

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 ?? 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.

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

⁶Secção 3.17 de [?] e tópico **Recursividade mútua** nos vídeos das aulas teóricas.

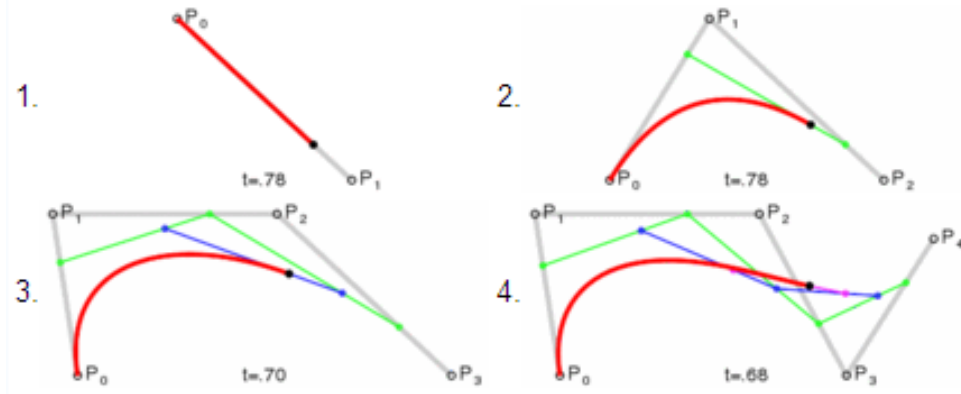


Figura 1: Exemplos de curvas de Bézier retirados da [Wikipedia](#).

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 ?? 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 trabalho⁷ 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 ,

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

onde $k = \text{length } x$. 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

$$\begin{aligned} \text{avg } [a] &= a \\ \text{avg } (a : x) &= \frac{1}{k+1} (a + \sum_{i=1}^k x_i) = \frac{a + k(\text{avg } x)}{k+1} \text{ para } k = \text{length } x \end{aligned}$$

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 :: Ord a ⇒ [a] → Property
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 ???. Para além disso, os grupos podem demonstrar o código na oral.

⁷A 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:⁸

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

Os diagramas podem ser produzidos recorrendo à *package* L^AT_EX *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 disciplina⁹, 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 ?? deste enunciado, ter-se-á, de imediato:

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

⁸Exemplos tirados de [?].

⁹Cf. [?], 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 (??):

```
catdef n = (2 * n)! ÷ ((n + 1)! * n!)
```

Oráculo para inspecção dos primeiros 26 números de Catalan¹⁰:

```
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
```

¹⁰Fonte: [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:¹¹

```

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

```

¹¹Pode 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:

outExpAr

A implementação desta função é deduzida a partir da própria propriedade enunciada:

$outExpAr \cdot inExpAr == id$

Deste modo, para descobrirmos **outExpAr** basta fazermos sua composição com **inExpAr** e igualarmos ao **id**. Aplicando algumas propriedades, temos:

$$\begin{aligned}
& outExpAr \cdot inExpAr = id \\
\equiv & \quad \{ \text{def-inExpAr} \} \\
& outExpAr \cdot [\underline{X}, num_ops] = id \\
\equiv & \quad \{ \text{fusão-+} \} \\
& [outExpAr \cdot \underline{X}, outExpAr \cdot num_ops] = id \\
\equiv & \quad \{ \text{Universal-+, Natural-id} \} \\
& \begin{cases} outExpAr \cdot \underline{X} = i_1 \\ outExpAr \cdot num_ops = i_2 \end{cases} \\
\equiv & \quad \{ \text{def-num_ops} \} \\
& \begin{cases} outExpAr \cdot \underline{X} = i_1 \\ outExpAr \cdot [N, ops] = i_2 \end{cases} \\
\equiv & \quad \{ \text{fusão-+} \} \\
& \begin{cases} outExpAr \cdot \underline{X} = i_1 \\ [outExpAr \cdot N, outExpAr \cdot ops] = i_2 \end{cases} \\
\equiv & \quad \{ \text{Universal-+} \} \\
& \begin{cases} outExpAr \cdot \underline{X} = i_1 \\ outExpAr \cdot N = i_2 \cdot i_1 \\ outExpAr \cdot ops = i_2 \cdot i_2 \end{cases} \\
\equiv & \quad \{ \text{def-ops} \} \\
& \begin{cases} outExpAr \cdot \underline{X} = i_1 \\ outExpAr \cdot N = i_2 \cdot i_1 \\ outExpAr \cdot [bin, \widehat{Un}] = i_2 \cdot i_2 \end{cases} \\
\equiv & \quad \{ \text{fusao-+} \} \\
& \begin{cases} outExpAr \cdot \underline{X} = i_1 \\ outExpAr \cdot N = i_2 \cdot i_1 \\ [outExpAr \cdot bin, outExpAr \cdot \widehat{Un}] = i_2 \cdot i_2 \end{cases} \\
\equiv & \quad \{ \text{universal-+} \}
\end{aligned}$$

$$\begin{aligned}
& \begin{cases} outExpAr \cdot \underline{X} = i_1 \\ outExpAr \cdot N = i_2 \cdot i_1 \\ outExpAr \cdot bin = i_2 \cdot i_2 \cdot i_1 \\ outExpAr \cdot \widehat{Un} = i_2 \cdot i_2 \cdot i_2 \end{cases} \\
& \equiv \quad \{ \text{introdução de variáveis-+, def-comp} \} \\
& \begin{cases} outExpAr \underline{X} a = i_1 (a) \\ outExpAr (N a) = i_2 (i_1 (a)) \\ outExpAr (bin a) = i_2 (i_2 (i_1 (a))) \\ outExpAr \widehat{Un} a = i_2 (i_2 (i_2 (a))) \end{cases} \\
& \square
\end{aligned}$$

Transformando esse sistema de equações para notação de **Haskell**, temos a solução:

```

outExpAr :: ExpAr a → () + (a + ((BinOp, (ExpAr a, ExpAr a)) + (UnOp, ExpAr a)))
outExpAr (X) = i1 ()
outExpAr (N a) = i2 (i1 (a))
outExpAr (Un op a) = i2 (i2 (i2 (op, a)))
outExpAr (Bin op a b) = i2 (i2 (i1 (op, (a, b))))

```

recExpAr

Sabendo agora o *tipo de saída* do *outExpAr*, passamos a conhecer o *tipo de entrada* da função *recExpAr*. Como tal função será a responsável por chamar recursivamente o catamorfismo para as "*ExpAr*'s" presentes no tipo de entrada, basta separarmos a função nos casos específicos em que temos de invocar o catamorfismo recursivamente, isto é, caso o tipo de entrada for do tipo **(BinOp, (ExpAr a, ExpAr a))** ou **(UnOp, ExpAr a)**. Caso contrário, não haverá recursividade e basta invocarmos o **id**

Para uma melhor ilustração deste processo, segue um diagrama do estado da resolução até este momento:

$$\begin{array}{ccc}
ExpAr\ a & \xrightarrow{outExpAr} & 1 + (A + ((BinOp, (ExpAr\ a, ExpAr\ a)) + (UnOp, ExpAr\ a))) \\
cataExpAr\ (gene) \downarrow & & \downarrow recExpAr\ (cataExpAr\ (gene)) \\
(Floating)\ C & \xleftarrow{gene} & 1 + (A + ((BinOp, (C, C) + (UnOp, C)))
\end{array}$$

Concluindo, temos então *recExpAr* como:

```

recExpAr f = id + (id + ((f1 f) + (f2 f))) where
  f1 f (op, (a, b)) = (op, (f a, f b))
  f2 f (op, a) = (op, f a)

```

g_eval_exp

```

g_eval_exp a = [a, resto] where
  resto = [id, resto2]
  resto2 = [bin, un] where
    bin (Sum, (c, d)) = c + d
    bin (Product, (c, d)) = c * d
    un (Negate, c) = (-1) * c
    un (E, c) = expd c

```

clean

```

clean :: (Floating a, Eq a) => ExpAr a -> () + (a + ((BinOp, (ExpAr a, ExpAr a)) + (UnOp, ExpAr a)))
clean (X) = i1 ()
clean (N a) = i2 (i1 (a))
clean (Un op a) | (op == E) ^& a == (N 0) = i2 (i1 (1))
                  | otherwise = i2 (i2 (i2 (op, a)))
clean (Bin op a b) | (op == Product) ^& (a == (N 0) ^& b == (N 0)) = i2 (i1 (0))
                  | otherwise = i2 (i2 (i1 (op, (a, b))))
gopt a = g_eval_exp a

```

sd_gen

```

sd_gen :: Floating a =>
  () + (a + ((BinOp, ((ExpAr a, ExpAr a), (ExpAr a, ExpAr a))) + (UnOp, (ExpAr a, ExpAr a)))) -> (ExpAr a)
sd_gen = [(X, N . 1), resto] where
  resto = [(N . id, N . 0), resto2] where
    resto2 = [f1, f2] where
      f1 (Sum, ((a, b), (c, d))) = (Bin Sum a c, Bin Sum b d)
      f1 (Product, ((a, b), (c, d))) = (Bin Product a c, Bin Sum (Bin Product a d) (Bin Product b c))
      f2 (Negate, (a, b)) = (Un Negate a, Un Negate b)
      f2 (E, (a, b)) = (Un E a, Bin Product (Un E a) b)
      -- (a,b) originais (c,b) derivadas

```

ad_gen

```

ad_gen :: Floating a =>
  a -> () + (a + ((BinOp, ((ExpAr a, a), (ExpAr a, a))) + (UnOp, (ExpAr a, a)))) -> (ExpAr a, a)
ad_gen v = [(X, 1), resto] where
  resto = [(N . id, 0), resto2] where
    resto2 = [f1, f2] where
      f1 (Sum, ((a, b), (c, d))) = (Bin Sum a c, b + d)
      f1 (Product, ((a, b), (c, d))) = (Bin Product a c, ((eval_exp v a) * d) + (b * (eval_exp v c)))
      f2 (Negate, (a, b)) = (Un Negate a, (-1) * b)
      f2 (E, (a, b)) = (Un E a, (expd (eval_exp v a)) * b)

```

Problema 2

Definir

```

loop (top, bot, n) = (2 * n * (2 * n - 1) * top, n * n * bot, 1 + n)
inic = (1, 1, 1)
prj (top, bot, n) = (top 'div' (n * bot))

```

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.

Começamos por separar o cálculo em duas partes, a parte do numerador e a do denominador. Se nos focamos no denominador, $(n + 1)!(n!)$, conseguimos determinar matematicamente uma forma de o calcular:

$$(n + 1)!(n!) = (n + 1)(n!)(n!) = (n + 1)(n!)^2$$

Conseguimos, então, criar uma função recursiva que calcula $(n!)^2$:

$$\begin{aligned} bot\ 0 &= 1 \\ bot\ (n + 1) &= (bot\ n) * n * n \end{aligned}$$

Podemos fazer o mesmo para o numerador, $(2n)!$:

$$(2n)! = (2n) * (2n - 1) * (2(n - 1))!$$

$$\begin{aligned} top\ 0 &= 1 \\ top\ (n + 1) &= (2 * n * (2 * n - 1)) * (top\ n) \end{aligned}$$

Utilizando isto, conseguimos determinar então uma função final que calcula o valor final:

$$cat\ n = (top\ n) \text{ 'div' } ((n + 1) * (bot\ n))$$

Podemos definir todas as funções necessárias para utilizar a regra de algibeira:

$$\begin{aligned} bot\ 0 &= 1 \\ bot\ (n + 1) &= (bot\ n) * (s\ n) * (s\ n) \\ top\ 0 &= 1 \\ top\ (n + 1) &= (2 * (s\ n) * (2 * (s\ n) - 1)) * (top\ n) \\ s\ 0 &= 1 \\ s\ (n + 1) &= 1 + s\ n \end{aligned}$$

Aplicando a regra de algibeira chegamos então à solução apresentada:

$$\begin{aligned} cat &= prj \cdot \text{for loop } inic \text{ where} \\ &\quad loop\ (top, bot, n) = (2 * n * (2 * n - 1) * top, n * n * bot, 1 + n) \\ &\quad inic = (1, 1, 1) \\ &\quad prj\ (top, bot, n) = (top \text{ 'div' } (n * bot)) \end{aligned}$$

Problema 3

$$\begin{aligned} calcLine &:: NPoint \rightarrow (NPoint \rightarrow OverTime\ NPoint) \\ calcLine &= cataList\ [\underline{nil}, g] \text{ where} \\ g &:: (\mathbb{Q}, NPoint \rightarrow OverTime\ NPoint) \rightarrow (NPoint \rightarrow OverTime\ NPoint) \\ g\ (d, f)\ l &= \text{case } l \text{ of} \\ &\quad [] \rightarrow nil \\ &\quad (x : xs) \rightarrow \lambda z \rightarrow concat\ \$\ (sequenceA\ [singl \cdot linearId\ d\ x, f\ xs])\ z \\ deCasteljau &:: [NPoint] \rightarrow OverTime\ NPoint \\ deCasteljau &= hyloAlgForm\ alg\ coalg \text{ where} \\ coalg &= divide \\ alg &= conquer \\ divide\ [] &= i_1\ [] \\ divide\ [a] &= i_1\ a \\ divide\ l &= i_2\ (init\ l, tail\ l) \\ quer\ (x, y) &= \lambda pt \rightarrow calcLine\ (x\ pt)\ (y\ pt)\ pt \\ conquer &= [_, quer] \\ hyloAlgForm\ f\ g &= \llbracket f \rrbracket \cdot \llbracket g \rrbracket \end{aligned}$$

Problema 4

Solução para listas não vazias:

$$avg = \pi_1 \cdot avg_aux$$

$$\begin{array}{ccc} A^+ & \xleftarrow{\text{in}} & A + A \times A^+ \\ \downarrow \langle avg, length \rangle & & \downarrow id + \langle avg, length \rangle \\ A \times \mathbb{N}_0 & \xleftarrow{g} & A + A \times (A \times \mathbb{N}_0) \end{array}$$

A partir do diagrama facilmente chegamos ao gene. Caso a sequência seja unitária então o resultado é o próprio número com `length` de 1. Se não, aplicamos a definição de `avg` e aplicamos a função `succ` à `length` que já tinha sido calculada.

```

outSList ([a]) = i1 (a)
outSList (a : b) = i2 (a, b)
cataSList g = g · (id + id × cataSList g) · outSList
avgAux = cataSList [⟨id, one⟩, ⟨k, succ · π2 · π2⟩]
  where
    k :: (Double, (Double, Integer)) → Double
    k (a, (b, c)) = (a + c' * b) / (c' + 1)
      where
        c' = fromIntegral c

```

Solução para árvores de tipo **LTree**:

```

avgLTree = π1 · (⟦ gene ⟧) where
  gene = [⟨id, one⟩, ⟨k, add · (π2 × π2)⟩]
  k :: ((Double, Integer), (Double, Integer)) → Double
  k ((a, b), (c, d)) = (a * b' + c * d') / (b' + d')
    where
      b' = fromIntegral b
      d' = fromIntegral d

```

Problema 5

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

F# mostrou ser uma linguagem bastante semelhante a Haskell, sendo a diferença principal a sintaxe (por exemplo, requerir o uso de *let* antes da definição de funções). A parte mais difícil desta "tradução" foi a diferença na omissão de argumentos, algo que, em geral, não é possível em **F#**.

```

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, (t1, t2)) -> i2 (a, (t1, t2))

// (2) Ana + cata + hylo -----
let baseBTree f g = id -|- (f >< (g >< g))
let recBTree g = baseBTree id g
let rec cataBTree g x = (g << recBTree (cataBTree g) << outBTree) x
let rec anaBTree g x = (inBTree << recBTree (anaBTree g) << g) x
let hyloBTree h g x = (cataBTree h << anaBTree g) x

// (3) Map -----
let fmap f x = cataBTree (inBTree << baseBTree f id) x

// (4) Examples -----

// (4.1) Immersion (mirror) -----
let invBTree x = cataBTree (inBTree << (id -|- (id >< swap))) x

```

```

// (4.2) Counting -----
let countBTree x = cataBTree (either (konst 0) (succ << (uncurry (+)) << p2)) x

// (4.3) Serialization -----
let inord y =
  let join (x, (l, r)) = l @ [x] @ r
  either nil join y
let inordt x = cataBTree inord x // in-order traversal

let preord y =
  let f (x, (l, r)) = x::(l @ r)
  either nil f y
let preordt x = cataBTree preord x // pre-order traversal

let postordt y = // post-order traversal
  let f (x, (l, r)) = l @ r @ [x]
  cataBTree (either nil f) y

// (4.4) Quicksort -----
let rec part p x =
  match x with
  | [] -> ([], [])
  | (h::t) -> let s, l = part p t
              if p h then (h::s,l) else (s,h::l)

let qsep x =
  match x with
  | [] -> i1 ()
  | (h::t) -> let s, l = part (fun n -> n < h) t
              i2 (h, (s, l))

let qSort x = hyloBTree inord qsep x

// (4.5) Traces -----
let rec union a b =
  match b with
  | [] -> a
  | h::t when List.contains h a -> union a t
  | h::t -> h::(union a t)

let tunion (a, (l, r)) = union (List.map (fun x -> a::x) l) (List.map (fun x -> a::x) r)
let traces x = cataBTree (either (konst [[]]) tunion) x

// (4.6) Towers of Hanoi -----
let present = inord
let strategy (d, n) = if n = 0 then i1 () else i2 ((n - 1, d), ((not d, n - 1), (not d, n - 1)))

let hanoi x = hyloBTree present strategy x

// The Towers of Hanoi problem comes from a puzzle marketed in 1883
// by the French mathematician Édouard Lucas, under the pseudonym
// Claus. The puzzle is based on a legend according to which
// there is a temple, apparently in Bramah rather than in Hanoi as
// one might expect, where there are three giant poles fixed in the
// ground. On the first of these poles, at the time of the world's
// creation, God placed sixty four golden disks, each of different
// size, in decreasing order of size. The Bramin monks were given

```

```

// the task of moving the disks, one per day, from one pole to another
// subject to the rule that no disk may ever be above a smaller disk.
// The monks' task would be complete when they had succeeded in moving
// all the disks from the first of the poles to the second and, on
// the day that they completed their task the world would come to
// an end!

// There is a wellknown inductive solution to the problem given
// by the pseudocode below. In this solution we make use of the fact
// that the given problem is symmetrical with respect to all three
// poles. Thus it is undesirable to name the individual poles. Instead
// we visualize the poles as being arranged in a circle; the problem
// is to move the tower of disks from one pole to the next pole in
// a specified direction around the circle. The code defines  $H\ n\ d$ 
// to be a sequence of pairs  $(k, d')$  where  $n$  is the number of disks,
//  $k$  is a disk number and  $d$  and  $d'$  are directions. Disks are numbered
// from 0 onwards, disk 0 being the smallest. (Assigning number 0
// to the smallest rather than the largest disk has the advantage
// that the number of the disk that is moved on any day is independent
// of the total number of disks to be moved.) Directions are boolean
// values, true representing a clockwise movement and false an anticlockwise
// movement. The pair  $(k, d')$  means move the disk numbered  $k$  from
// its current position in the direction  $d'$ . The semicolon operator
// concatenates sequences together,  $[]$  denotes an empty sequence
// and  $[x]$  is a sequence with exactly one element  $x$ . Taking the pairs
// in order from left to right, the complete sequence  $H\ n\ d$  prescribes
// how to move the  $n$  smallest disks onebyone from one pole to the
// next pole in the direction  $d$  following the rule of never placing
// a larger disk on top of a smaller disk.

//  $H\ 0\ d = []$ ,
//  $H\ (n+1)\ d = H\ n\ d ; [ (n, d) ] ; H\ n\ d$ .

// (excerpt from R. Backhouse, M. Fokkinga / Information Processing
// Letters 77 (2001) 71--76)

// (5) Depth and balancing (using mutual recursion) -----
let baldepth n =
  let h (a, ((b1, b2), (d1, d2))) = (b1 && b2 && abs (d1 - d2) <= 1, 1 + max d1 d2)
  let f ((b1, d1), (b2, d2)) = ((b1, b2), (d1, d2))
  let g x = either (konst (true, 1)) (h << (id >< f)) x
  cataBTree g n

let balBTree x = (p1 << baldepth) x
let depthBTree x = (p2 << baldepth) x

```

Índice

L^AT_EX, [1](#)

bibtex, [2](#)

 lhs2TeX, [1](#)

 makeindex, [2](#)

Combinador “pointfree”

 cata, [8](#), [9](#)

 either, [3](#), [8](#), [13–16](#)

Curvas de Bézier, [6](#), [7](#)

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

 Material Pedagógico, [1](#)

 BTree.hs, [8](#)

 Cp.hs, [8](#)

 LTree.hs, [8](#), [16](#)

 Nat.hs, [8](#)

Deep Learning), [3](#)

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

F#, [8](#), [17](#)

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

Função

π_1 , [6](#), [9](#), [16](#)

π_2 , [9](#), [13](#), [16](#)

 for, [6](#), [9](#), [15](#)

 length, [8](#), [16](#)

 map, [11](#), [12](#)

 succ, [16](#)

 uncurry, [3](#), [13](#), [14](#)

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

 Gloss, [2](#), [11](#)

 interpretador

 GHCi, [2](#)

 Literate Haskell, [1](#)

 QuickCheck, [2](#)

 Stack, [2](#)

Números de Catalan, [6](#), [10](#)

Números naturais (I

\mathbb{N}), [5](#), [6](#), [9](#), [16](#)

Programação

 dinâmica, [5](#)

 literária, [1](#)

Racionais, [7](#), [8](#), [10–12](#), [16](#)

U.Minho

 Departamento de Informática, [1](#)

Referências