

# Cálculo de Programas

## Trabalho Prático

### MiEI+LCC — 2020/21

Departamento de Informática  
Universidade do Minho

Junho de 2021

<b>Grupo nr.</b>	999 (preencher)
a89494	Rita Celeste Perucho Teixeira
a89476	Bruno Alexandre Martins Carvalho
a33333	Nome3 (preencher)
a44444	Nome4 (preencher, se aplicável, ou apagar)

## 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`<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 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 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.

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

**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 [?], página 98.

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

Obter-se-á de imediato

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

usando as regras seguintes:

- O corpo do ciclo *loop* terá tantos argumentos quanto o número de funções mutuamente recursivas.
- Para as variáveis escolhem-se os próprios nomes das funções, pela ordem que se achar conveniente.<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:

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

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 [?] 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 = ⊥ -- (¡[]) I DON'T KNOW BITCH SEE LATER!
```

## 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 [?].

<sup>9</sup>Cf. [?], 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
```

$eval\_exp :: Floating\ a \Rightarrow a \rightarrow (ExpAr\ a) \rightarrow a$   
 $eval\_exp\ a = cataExpAr\ (g\_eval\_exp\ a)$   
 $optimize\_eval :: (Floating\ a, Eq\ a) \Rightarrow a \rightarrow (ExpAr\ a) \rightarrow a$   
 $optimize\_eval\ a = hyloExpAr\ (gopt\ a)\ clean$   
 $sd :: Floating\ a \Rightarrow ExpAr\ a \rightarrow ExpAr\ a$   
 $sd = \pi_2 \cdot cataExpAr\ sd\_gen$   
 $ad :: Floating\ a \Rightarrow a \rightarrow ExpAr\ a \rightarrow a$   
 $ad\ v = \pi_2 \cdot cataExpAr\ (ad\_gen\ v)$

### D.0.1 Resolução e respetiva explicação

Para a maior parte do Problema 1, tivemos que nos familiarizar com os conceitos de Hilomorfismo, Anamorfismo e (o mais utilizado e trabalhado) Catamorfismo. A sua estrutura em diagrama generalizada é a seguinte:

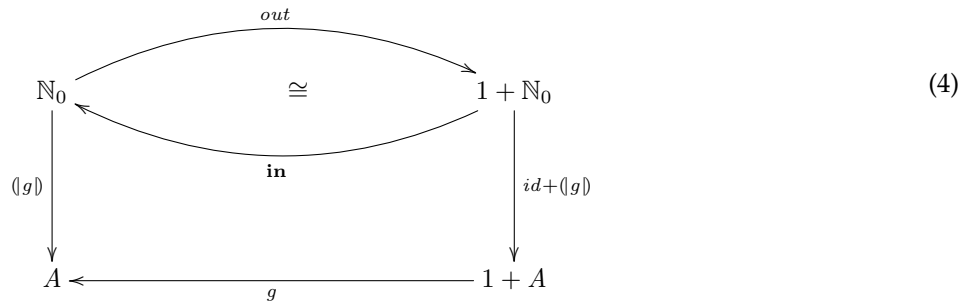
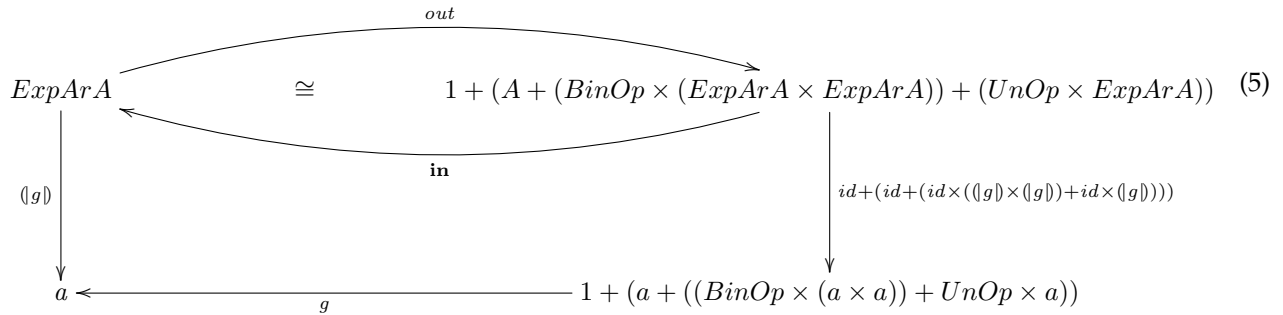


Diagrama do catamorfismo adaptado à função `recExpAr`:



Definir:

**outExpAr**

$$\begin{aligned}
 & out \cdot in = id \\
 \equiv & \quad \{ \text{pelo enunciado} \} \\
 & out \cdot [\underline{X}, num\_ops] = id \\
 \equiv & \quad \{ \text{pelo formulário Fusão-+} \} \\
 & [out \cdot \underline{X}, out \cdot num\_ops] = id \\
 \equiv & \quad \{ \text{Universal-+} \} \\
 & \left\{ \begin{array}{l} id \cdot i_1 = out \cdot \underline{X} \\ id \cdot i_2 = out \cdot num\_ops \end{array} \right. \\
 \equiv & \quad \{ \text{Natural-id} \} \\
 & \left\{ \begin{array}{l} i_1 = out \cdot \underline{X} \\ i_2 = out \cdot num\_ops \end{array} \right. \\
 \equiv & \quad \{ \text{Igualdade extensional e segundo o enunciado} \}
 \end{aligned}$$

$$\begin{aligned}
& \begin{cases} i_1 \ x1 = (out \cdot \underline{X}) \ x1 \\ i_2 = out \cdot [N \ ops, \cdot] \end{cases} \\
\equiv & \quad \{ \text{Def-comp}, \text{Def-const}, \text{Fusão-+} \} \\
& \begin{cases} i_1 = out \ X \\ i_2 = out \cdot [(out \cdot N) \ (out \cdot ops), \cdot] \end{cases} \\
\equiv & \quad \{ \text{Universal-+} \} \\
& \begin{cases} (i_2 \cdot i_1) \ x2 = (out \cdot N) \ x2 \\ i_2 \cdot i_2 = out \cdot ops \end{cases}
\end{aligned}$$

Finalmente pelo enunciado e pelo desenvolvimento repetitivo das regras obtemos a solução seguinte:

```

outExpAr X = i1 ()
outExpAr (N x) = i2 $ i1 x
outExpAr (Un unop x) = i2 $ i2 $ i2 (unop, x)
outExpAr (Bin binop a b) = i2 $ i2 $ i1 (binop, (a, b))
--

```

### recExpAr

```

recExpAr f = baseExpAr id id id f f id f

```

### g\_eval\_exp

De modo a resolver o seguinte código foi utilizado o exemplo de catamorfismo referido anteriormente com uma simples alteração da função, sendo esta a eval\_exp.

```

g_eval_exp f = [f, [g_eval, [g_eval_bin, g_eval_un]]]
g_eval = id
g_eval_bin (binop, (a, b)) = if (binop == Sum) then (a + b)
  else (a * b)
g_eval_un (unop, a) = if (unop == Negate) then (-a)
  else (expd a)

```

### clean

A função clean é uma função que tira proveito de outra resolvida anteriormente, no entanto vimonos forçados a tratar alguns casos específicos, nomeadamente a multiplicação por 0 e o caso de e^0.

```

clean (Bin Product _ (N 0)) = outExpAr $ N 0
clean (Bin Product (N 0) _) = outExpAr $ N 0
clean (Un E (N 0)) = outExpAr $ N 1
clean x = outExpAr x

```

### Optimize

```

gopt f = g_eval_exp f

```

### sd\_gen

Com o intuito de resolver o sd\_gen, foram utilizadas as regras da soma e do produto previamente fornecidas neste mesmo relatório. O objetivo desta função é aplicar tais fórmulas às diferentes formas como a ExpAr pode ser apresentada.

```

sd_gen :: Floating a =>
  () + (a + ((BinOp, ((ExpAr a, ExpAr a), (ExpAr a, ExpAr a))) + (UnOp, (ExpAr a, ExpAr a)))) -> (ExpAr a)
sd_gen = [f1, [f2, [f3, f4]]] where
  f1 _ = (X, N 1)
  f2 a = (N a, N 0)

```

$$\begin{aligned}
& f3 \text{ (binop, ((a, b), (c, d))) | (binop \equiv Sum) = ((Bin Sum a c), (Bin Sum b d)) \\
& \quad | \text{otherwise} = (Bin Product a c, Bin Sum (Bin Product a d) (Bin Product b c)) \\
& f4 \text{ (unop, (a, b)) | (unop \equiv Negate) = (Un Negate a, Un Negate b)} \\
& \quad | \text{otherwise} = (Un E a, Bin Product (Un E a) b)
\end{aligned}$$

### ad\_gen

A função `ad_gen` tem o mesmo objetivo da função criada anteriormente, porém já são aplicados valores específicos (não nos ficamos pelas expressões matemáticas).

$$\begin{aligned}
& ad\_gen \ v \ (i_1 \ ()) = (v, 1) \\
& ad\_gen \ v \ (i_2 \ (i_1 \ a)) = (a, 0) \\
& ad\_gen \ v \ (i_2 \ (i_2 \ (i_1 \ (binop, ((a, b), (c, d))))) | (binop \equiv Sum) = (a + c, b + d) \\
& \quad | \text{otherwise} = (a * c, (a * d) + (b * c)) \\
& ad\_gen \ v \ (i_2 \ (i_2 \ (i_2 \ (unop, (a, b))))) | (unop \equiv Negate) = (-a, -b) \\
& \quad | \text{otherwise} = (expd \ a, (expd \ a) * b)
\end{aligned}$$

## Problema 2

Definir

$$\begin{aligned}
& loop \ (a, auxd, auxc, b, auxb, c, auxc) = (auxd * auxc * a, 2 + auxd, 2 + auxc, auxb * b, 1 + auxb, auxc * c, 1 + auxc) \\
& inic = (1, 2, 1, 1, 2, 1, 1) \\
& prj \ (a, auxd, auxc, b, auxb, c, auxc) = a \div b * c
\end{aligned}$$

por forma a que

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

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

## Problema 3

Utilizando a função `calcLine` para calcular as curvas de Bézier, necessitamos de mais que um ponto que constitui a mesma. Assim, para construirmos a função `calcLine` tivemos que considerar isso mesmo: caso fosse fornecido apenas um ponto, a função iria retornar nulo; caso contrário, utilizamos uma função auxiliar que segue o código fornecido no relatório pelos docentes. De modo a compreender melhor a função auxiliar, esta é feita como uma versão do `cataList` da classe `List.hs`.

$$\begin{aligned}
& calcLine :: NPoint \rightarrow (NPoint \rightarrow OverTime \ NPoint) \\
& calcLine = cataList \ h \ \textbf{where} \\
& \quad h = [\lambda \text{ ponto} \rightarrow nil, auxCalcLine] \\
& auxCalcLine \ (d, f) \ l = \textbf{case } l \ \textbf{of} \\
& \quad [] \rightarrow nil \\
& \quad (x : xs) \rightarrow \lambda z \rightarrow concat \$ (sequenceA [singl \cdot linear1d \ d \ x, f \ xs]) \ z
\end{aligned}$$

Tomando como ponto de partida as `LTrees`, uma vez que são próximas daquilo que é o retorno pedido e na classe `LTree.hs`, constatámos a existência da função `mSort`, que é um `divide and conquer`, podemos seguir então esse método. Começando então por fazer uma função auxiliar do gene do anamorfismo (`alg`) e, posteriormente, o gene do catamorfismo (`coAlg`), este último através do nos é fornecido pelos docentes, utilizando o `calcLine`.

Por fim, obtemos o resultado seguinte:

$$\begin{aligned}
& deCasteljau :: [NPoint] \rightarrow OverTime \ NPoint \\
& deCasteljau = hyloAlgForm \ alg \ coalg \ \textbf{where} \\
& \quad coalg = \llbracket castelCoalgAux \rrbracket \\
& \quad alg = \llbracket castelAlgAux \rrbracket \\
& castelAlgAux \ [] = i_1 \ nil \\
& castelAlgAux \ [p] = i_1 \ p
\end{aligned}$$

```

castelAlgAux l = i2 ((init l), (tail l))
castelCoalgAux = [aux1, aux2]
aux1 l = l
aux2 (p, q) = λpt → (calcLine (p pt) (q pt)) pt
hyloAlgForm f g = g · f

```

#### Problema 4

Solução para listas não vazias:

$$avg = \pi_1 \cdot avg\_aux$$

$$avg\_aux = \perp$$

Solução para árvores de tipo **LTree**:

$$avgLTree = \pi_1 \cdot \langle gene \rangle \textbf{ where}$$

$$gene = \perp$$

#### Problema 5

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