

# Cálculo de Programas

## Trabalho Prático

### MiEI+LCC — 2020/21

Departamento de Informática  
Universidade do Minho

Junho de 2021

Grupo nr.	80
a84628	Carlos Beiramar
a80931	Jorge Silva
a52846	Tiago Azevedo

## 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 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 13 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 13 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.

$$f\ 0 = 1$$

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

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.<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:

$$f\ 0 = c$$

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

$$k\ 0 = a + b$$

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

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 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
```

- 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  $x$ ,

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

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.

- Recorra à lei de recursividade mútua para derivar a função `avg_aux = ([b, q])` tal que `avg_aux = (avg, length)` em listas não vazias.
- 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 [?].

<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 (4):

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

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 ==
(==) :: Eq b => (a -> b) -> (a -> b) -> (a -> Bool)
f == g =  $\lambda$ a -> f a == g a
infixr 4 <=
(<=) :: Ord b => (a -> b) -> (a -> b) -> (a -> Bool)
f <= g =  $\lambda$ a -> f a <= g a
infixr 4 ^&
(^&) :: (a -> Bool) -> (a -> Bool) -> (a -> Bool)
f ^& g =  $\lambda$ a -> ((f a) ^& (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)

```

Irão ser apresentadas agora as funções implementadas para a resolução do **Problema 1**.

1. A função *outExpAr* foi definida da seguinte forma:

```

outExpAr X = i1 ()
outExpAr num_ops = i2 (outNumOps num_ops)
outNumOps (N v) = i1 (v)
outNumOps (ops) = i2 (outOps ops)
outOps (Bin op a b) = i1 (outBinOps (Bin op a b))
outOps (Un op a) = i2 (outUnOps (Un op a))
outBinOps (Bin op a b) = (op, (a, b))
outUnOps (Un op a) = (op, a)

```

O objetivo desta expressão é “destruir” uma expressão do tipo *ExpAr*, ou seja, apresentando alguns exemplos para clarificar:

- **X** = Left()
- **(N v)** = Right(Left v)
- **(Bin ops a b)** = Right(Right(Left(ops,(a,b))))
- **(Un ops a)** = Right(Right(Right (ops,a)))

2. Seguidamente é apresentada a resolução para a função *recExpAr*. Nesta implementação seguiu-se as instruções dadas pelo professor numa das FAQ's presentes da página pública da UC e implementeu-se a função *baseExpAr* da seguinte forma:

$$baseExpAr' g f = baseExpAr id g id f f id f$$

Assim, a função *recExpAr f* segue o mesmo princípio que todas as estruturas leccionadas e, passa-se o *id* no primeiro argumento do *baseExpAr* e o *f* dado é passado como segundo argumento.

$$recExpAr f = baseExpAr' id f$$

3. De seguida implementeu-se a função *g\_eval\_exp* cujo objetivo é: Dada uma expressão aritmética e um escalar para substituir o *X*, esta função calcula o resultado da expressão.

```

g_eval_exp a = [left, right] where
  left = a
  right = [left_1, right_1] where
    left_1 n = n
    right_1 = [left_2, right_2] where
      left_2 (Sum, (exp1, exp2)) = exp1 + exp2
      left_2 (Product, (exp1, exp2)) = exp1 * exp2
      right_2 (Negate, exp1) = -exp1
      right_2 (E, x) = Prelude.exp x

```

4. Na função seguinte, é pretendido otimizar o cálculo do valor de uma expressão aritmética tirando proveito dos elementos absorventes de cada operação.

```
-- divide
clean (Bin Product _ (N 0)) = outExpAr (N 0)
clean (Bin Product (N 0) _) = outExpAr (N 0)
clean (Bin Sum a (N 0)) = outExpAr a
clean (Bin Sum (N 0) b) = outExpAr b
clean exp = outExpAr exp

-- conquer
gopt a = g_eval_exp a
```

Sabe-se que um **hilomorfismo** é decomposto por um **catamorfismo** após um **anamorfismo**. Sendo assim, na função *clean* foram definidos os casos para os elementos absorventes na soma e no produto das expressões aritméticas. Nesta função foi usada também a função *outExpAr* para “destruir” a expressão passada como argumento. Por fim, após esta otimização, definiu-se a função *gopt* que irá receber um valor como argumento e irá calcular o valor da expressão devolvida pelo **anamorfismo**.

5.  $sd\_gen :: Floating\ a \Rightarrow$   
 $() + (a + ((BinOp, ((ExpAr\ a, ExpAr\ a), (ExpAr\ a, ExpAr\ a))) + \cdot$   
 $(UnOp, (ExpAr\ a, ExpAr\ a)))) \rightarrow (ExpAr\ a, ExpAr\ a)$   
 $sd\_gen = \perp$
6.  $ad\_gen\ a\ exp = \perp$

## Problema 2

A fórmula dada para calcular o  $n$ -ésimo número de Catalan é

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

ou seja, calcula os 3 factoriais diferentes, o que, por si só é um desperdício de processamento pois efetua cálculos repetidos.

**Exemplo:** Vejamos o exemplo para o número 5 de Catalan.

$$\frac{(2 * 5)!}{(5 + 1)! * 5!} = \frac{(10 * 9 * 8 * 7 * 6 * 5 * 4 * 3 * 2 * 1)}{(6 * 5 * 4 * 3 * 2 * 1) * (5 * 4 * 3 * 2 * 1)} = \frac{(10 * 9 * 8 * 7 * 6 * 5!)}{(6 * 5!) * 5!} \quad (5)$$

Como é possível verificar, o  $5!$  é calculado 3 vezes separadas. Assim, com o intuito de tornar o cálculo mais eficiente, é possível cancelar algumas multiplicações com a divisão. Por exemplo, o  $(n+1)!$  do denominador pode ser retirado do  $2n!$  do numerador e o resultado obtido seria:

$$\frac{(2n * 2n - 1 * 2n - 2 ... 2n - (n + 2))}{n!} \quad (6)$$

Após isto, é possível verificar que o cálculo já se tornou mais eficiente

Tal como a sequência de *Fibonacci*, os números de *Catalan* satisfazem a "relação de recorrência", portanto podem ser definidos da seguinte forma:

$c_0 = 1$

$c_{(n+1)} = (2 * (2n+1) / (n+2)) * c_n$

Isto é o ideal pois como se quer implementar um ciclo for que começa a partir de um valor inicial (init), o crescimento dos valores é facilmente calculado a cada iteração. Como se pretende usa a divisão inteira, decidiu-se utilizar 3 funções recursivas entre si.

**Um contador:**  $f_n = n + 1$

**O Numerador:**  $g(0) = 1$ ;  $g(n+1) = 2 * (2 * x + 1) * n$

**O Denominador:**  $h(0) = 1$ ;  $h(n+1) = (n+2) * h(n)$

A divisão foi deixada para último passado, para evitar os números não inteiros, tal como é pedido no enunciado. Assim sendo, o ciclo inicia com um tuplo **(1,1,0)** e cada iteração do **loop** aplica a função a cada componente  $(d, n, x) = ((x+2) * d, 2 * (2 * x + 1) * n, 1 + x)$ . Após todas as iteração terminarem, a função **proj** faz a divisão do numerador pelo denominador e devolve o valor.

De seguida, é apresentado o código elaborado para a realização do exercício.

```
loop (d, n, x) = ((x + 2) * d, 2 * (2 * x + 1) * n, 1 + x)
inic = (1, 1, 0)
prj (a, b, c) = b ÷ a
```

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

A função **calcLine** recorre à função **linear1d** para determinar a interpolação entre dois valores. Como o objetivo é determinar a interpolação de várias coordenadas, faz sentido que esta função percorra duas listas cada uma com o conjunto de coordenadas de cada ponto. Em **Haskell** a função teria que ter este comportamento:

```
calcLineHaskell [] [] _ = []
calcLineHaskell (x : xs) [] _ = []
calcLineHaskell [] (x : xs) _ = []
calcLineHaskell (a : ax) (b : bx) d = linear1d a b d : calcLineHaskell ax bx d
```

Esta função irá percorrer duas listas, componente a componente, calcula o **linear1d** com o par de componentes e devolve uma lista com os valores das interpolações. Caso alguma ou ambas as listas estiverem vazias, o cálculo termina.

Para se definir como catamorfismo, este tem que fazer **outList** recursivamente até ao caso de paragem, consumir a lista de acordo com o gene e relacionar com a lista seguinte. Foi definido o catamorfismo em *pointwise* da seguinte forma:

A implementação do algoritmo de **deCasteljau** como hilomorfismo divide-se em duas partes. Tal como foi definido, o **hilomorfismo** é um **anamorfismo** seguido de um **catamorfismo**.

No **anamorfismo** irá ser criada uma estrutura "virtual" intermédia que, posteriormente, irá ser consumido pelo **catamorfismo**. Neste caso, a estrutura intermédia escolhida foi uma *Leaf Tree* pois o algoritmo original divide sucessivamente a lista de pontos em listas de comprimentos iguais (**init** e **tail**) até chegar ao ponto individual. Ou seja, a cada **init** e **tail** de uma lista é feito um **Fork**. Atingindo a lista de valor individual(ponto), é criada a *Leaf*.

Nas aulas foi analisado o **mergeSort** que consideramos em certos padrões semelhante nesta parte do anamorfismo.

Sendo assim, foi definido que

1. **divide** - Separa a lista com **init** e **tail** sucessivamente e transforma essas separações em Forks até chegar às Leafs que serão os pontos individuais.

```
separador = ⟨init, tail⟩
divide [] = i1 []
divide [x] = i1 x
divide l = i2 (separador l)
```

2. **conquer** - Nesta fase, ou seja no **catamorfismo**, o gene devolve o valor da *Leaf* ou, quando era *Fork* fazia o cálculo dos dois pontos usando a **calcLine** acima definido.

```
conquer = [g1, g2] where
  g1 x _ = x
  g2 (l, r) z = calcLine (l z) (r z) z
```

Assim, obtém-se o seguinte código:

```
calcLine :: NPoint → (NPoint → OverTime NPoint)
calcLine = cataList h where
  h = [g1, g2]
  g1 x _ = nil
  g2 (a, b) [] z = b [] z
```



```

g2 (a, b) (x : xs) z = (linear1d a x z) : b xs z
deCasteljau :: [NPoint] → OverTime NPoint
deCasteljau = hyloAlgForm alg coalg where
    coalg = divide
    alg = conquer
hyloAlgForm = hyloLTree

```

**Notas:** Tal como indicado no enunciado, o algoritmo de **deCasteljau** não é mais eficiente pois efetua muitas interpolações entre pontos repetidos de forma a obter um valor mais exato. Como tal, o **quickCheck** com argumentos por defeito cria listas demasiado grandes para o computador processar em tempo útil. O tamanho do `maxSize` foi então reduzido para 20 ou 25. Sugere-se então que se altere nos argumentos do **quickCheck** o `maxSize` para 20 ou 25.

## Problema 4

Neste problema é necessário implementar um catamorfismo que calculasse a média de uma lista de números. Deste modo, é necessário elaborar duas coisas, obter o valor do somatório de todos os elementos presentes na lista e descobrir qual é o tamanho dessa mesma lista. Só será necessário percorrer a lista uma única vez.

Este problema foi dividido em pequenos problemas, de seguida irão ser abordadas cada uma das funções criadas.

1. A função *opDivisao* irá receber um par, **(Double, Double)**, e irá retornar o valor da **divisão** entre os dois elementos do par.

$$opDivisao = (\div)$$

2. A função *opSoma* irá receber um par, **(Double, Double)**, e irá retornar o valor da **soma** entre os dois elementos do par.

$$opSoma = (+)$$

3. A função *opMulti* irá receber um par, **(Double, Double)**, e irá retornar o valor da **multiplicação** entre os dois elementos do par.

$$opMulti = (*)$$

4. A função *opConvert* irá receber um par, **(Integer, Integer)**, e irá converter esse par para **(Double, Double)**, usando a função *realToFrac*.

$$opConvert = (realToFrac \times realToFrac)$$

5. A função *multi* irá receber um par, **(b, (g1, g2))**, e irá retornar um par **(x,y)** onde:

- $x = b$
- $y = (g1 * g2, g2)$

$$multi = (id \times \langle opMulti, \pi_2 \rangle)$$

6. A função *somas* irá receber um tuplo, **(b,(g1,g2))**, e irá retornar um par **(x,y)** onde:

- $x = opSoma (b,g1)$
- $y = g2$

$$somas = \langle opSoma \cdot \langle \pi_1, \pi_1 \cdot \pi_2 \rangle, \pi_2 \cdot \pi_2 \rangle$$

7. A função *incLen* irá receber um par, **(a,b)** e irá retornar um par **(x,y)** onde:

- $x = a$
- $y = b + 1$

$$incLen = (id \times (1+))$$

8. A função *divisao* irá receber **(a,b)** e irá retornar um par **(x,y)** onde:

- $x = opDivisao(a,b)$
- $y = b$

$$divisao = \langle opDivisao, \pi_2 \rangle$$

Solução para listas não vazias:

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

Na implementação na função é possível verificar que, o catamorfismo faz **outList** até chegar à lista vazia (**g1**) e, nesse momento, "cria" o par **(0,0)** sendo (media, length). Para além disso, neste passo, os valores são convertidos para **Double** para depois ser possível fazer a divisão evitando assim os conflitos entre tipos quando são obtidos valores fracionários.

De seguida, a cada passo do gene (**g2**), o catamorfismo aplica uma composição de operações aritméticas.

$$\begin{aligned} avg\_aux &= cataList\ g\ \textbf{where} \\ g &= [g1, g2]\ \textbf{where} \\ g1 &= opConvert \cdot \langle zero, zero \rangle \\ g2 &= (divisao \cdot incLen \cdot somas \cdot multi) \end{aligned}$$

Para generalizar esse cálculo para **LTree** foram feitas algumas alterações pois, não era possível seguir o raciocínio implementado anteriormente. Para tal, a árvore trabalhava com um triplo (**average,(sum,length)**) que calculava a nova média a cada Fork que irá ser iterado.

No caso de paragem, ou seja, em cada Leaf transformava o valor do triplo indicado acima. E no Fork as médias anteriores eram descartadas, soma-se os somatórios, os lengths e calcula-se a nova média, tornando a criar um triplo para continuar a "subir" na árvore.

Solução para árvores de tipo **LTree**:

$$\begin{aligned} avgLTree &= \pi_1 \cdot \langle gene \rangle\ \textbf{where} \\ gene &= [g1, g2]\ \textbf{where} \\ g1 &= (opDivisao \times id) \cdot \langle id, id \rangle \cdot (realToFrac \times realToFrac) \cdot \langle id, one \rangle \\ g2 &= (opDivisao \times id) \cdot \langle id, id \rangle \cdot (opSoma \times opSoma) \cdot \langle \pi_1 \times \pi_1, \pi_2 \times \pi_2 \rangle \cdot (\pi_2 \times \pi_2) \end{aligned}$$

## Problema 5

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