Cálculo de Programas
Trabalho Prático
MiEI+LCC — 2020/21

Departamento de Informática
Universidade do Minho

Junho de 2021

Grupo nr.	111
a93202	Francisco Alexandre Pinto Neves
a93166	João Pedro Rodrigues Carvalho
a93310	Joaquim Tiago Martins Roque

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 LaTeX 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
```

[1] 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 depêndencias 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 depêndencias externas serão instaladas automaticamente.

Para gerar o PDF, garanta que se encontra na diretoria *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,

$$
\begin{array}{l}
\mathbf{data}\ \mathit{ExpAr}\ a = X \\
\quad \mid N\ a \\
\quad \mid \mathit{Bin}\ \mathit{BinOp}\ (\mathit{ExpAr}\ a)\ (\mathit{ExpAr}\ a) \\
\quad \mid \mathit{Un}\ \mathit{UnOp}\ (\mathit{ExpAr}\ a) \\
\quad \mathbf{deriving}\ (\mathit{Eq}, \mathit{Show})
\end{array}
$$

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

$$
\begin{array}{l}
\mathbf{data}\ \mathit{BinOp} = \mathit{Sum} \\
\quad \mid \mathit{Product} \\
\quad \mathbf{deriving}\ (\mathit{Eq}, \mathit{Show}) \\
\mathbf{data}\ \mathit{UnOp} = \mathit{Negate} \\
\quad \mid E \\
\quad \mathbf{deriving}\ (\mathit{Eq}, \mathit{Show})
\end{array}
$$

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,

$$\mathit{Bin}\ \mathit{Sum}\ X\ (N\ 10)$$

designa $x + 10$ na notação matemática habitual.

1. A definição das funções $\mathit{inExpAr}$ e $\mathit{baseExpAr}$ para este tipo é a seguinte:

$$
\begin{array}{l}
\mathit{inExpAr} = [\underline{X}, \mathit{num\_ops}]\ \mathbf{where} \\
\quad \mathit{num\_ops} = [N, \mathit{ops}] \\
\quad \mathit{ops} = [\mathit{bin}, \widehat{\mathit{Un}}] \\
\quad \mathit{bin}\ (\mathit{op}, (a, b)) = \mathit{Bin}\ \mathit{op}\ a\ b \\
\mathit{baseExpAr}\ f\ g\ h\ j\ k\ l\ z = f + (g + (h \times (j \times k) + l \times z))
\end{array}
$$

   Defina as funções $\mathit{outExpAr}$ e $\mathit{recExpAr}$, e teste as propriedades que se seguem.

   Propriedade [QuickCheck] 1 *$\mathit{inExpAr}$ e $\mathit{outExpAr}$ são testemunhas de um isomorfismo, isto é, $\mathit{inExpAr} \cdot \mathit{outExpAr} = \mathit{id}$ e $\mathit{outExpAr} \cdot \mathit{idExpAr} = \mathit{id}$:*

$$
\begin{array}{l}
\mathit{prop\_in\_out\_idExpAr} :: (\mathit{Eq}\ a) \Rightarrow \mathit{ExpAr}\ a \to \mathit{Bool} \\
\mathit{prop\_in\_out\_idExpAr} = \mathit{inExpAr} \cdot \mathit{outExpAr} \equiv \mathit{id} \\
\mathit{prop\_out\_in\_idExpAr} :: (\mathit{Eq}\ a) \Rightarrow \mathit{OutExpAr}\ a \to \mathit{Bool} \\
\mathit{prop\_out\_in\_idExpAr} = \mathit{outExpAr} \cdot \mathit{inExpAr} \equiv \mathit{id}
\end{array}
$$

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.*

$$\begin{array}{l}
prop_sum_idr :: (Floating\ a, Real\ a) \Rightarrow a \rightarrow ExpAr\ a \rightarrow Bool \\
prop_sum_idr\ a\ exp = eval_exp\ a\ exp \stackrel{?}{=} sum_idr\ \mathbf{where} \\
\quad sum_idr = eval_exp\ a\ (Bin\ Sum\ exp\ (N\ 0)) \\
prop_sum_idl :: (Floating\ a, Real\ a) \Rightarrow a \rightarrow ExpAr\ a \rightarrow Bool \\
prop_sum_idl\ a\ exp = eval_exp\ a\ exp \stackrel{?}{=} sum_idl\ \mathbf{where} \\
\quad sum_idl = eval_exp\ a\ (Bin\ Sum\ (N\ 0)\ exp) \\
prop_product_idr :: (Floating\ a, Real\ a) \Rightarrow a \rightarrow ExpAr\ a \rightarrow Bool \\
prop_product_idr\ a\ exp = eval_exp\ a\ exp \stackrel{?}{=} prod_idr\ \mathbf{where} \\
\quad prod_idr = eval_exp\ a\ (Bin\ Product\ exp\ (N\ 1)) \\
prop_product_idl :: (Floating\ a, Real\ a) \Rightarrow a \rightarrow ExpAr\ a \rightarrow Bool \\
prop_product_idl\ a\ exp = eval_exp\ a\ exp \stackrel{?}{=} prod_idl\ \mathbf{where} \\
\quad prod_idl = eval_exp\ a\ (Bin\ Product\ (N\ 1)\ exp) \\
prop_e_id :: (Floating\ a, Real\ a) \Rightarrow a \rightarrow Bool \\
prop_e_id\ a = eval_exp\ a\ (Un\ E\ (N\ 1)) \equiv expd\ 1 \\
prop_negate_id :: (Floating\ a, Real\ a) \Rightarrow a \rightarrow Bool \\
prop_negate_id\ a = eval_exp\ a\ (Un\ Negate\ (N\ 0)) \equiv 0
\end{array}$$

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

$$\begin{array}{l}
prop_double_negate :: (Floating\ a, Real\ a) \Rightarrow a \rightarrow ExpAr\ a \rightarrow Bool \\
prop_double_negate\ a\ exp = eval_exp\ a\ exp \stackrel{?}{=} eval_exp\ a\ (Un\ Negate\ (Un\ Negate\ exp))
\end{array}$$

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

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

que se encontra na página 12 expressa como um hilomorfismo[2] e teste as propriedades:

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

$$\begin{array}{l}
prop_optimize_respects_semantics :: (Floating\ a, Real\ a) \Rightarrow a \rightarrow ExpAr\ a \rightarrow Bool \\
prop_optimize_respects_semantics\ a\ exp = eval_exp\ a\ exp \stackrel{?}{=} optmize_eval\ a\ exp
\end{array}$$

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:

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

[2] Qual é a vantagem de implementar a função $optimize\_eval$ utilizando um hilomorfismo em vez de utilizar um catamorfismo com um gene "inteligente"?

[3] 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.*

$$
\begin{array}{l}
prop\_const\_rule :: (Real\ a, Floating\ a) \Rightarrow a \rightarrow Bool \\
prop\_const\_rule\ a = sd\ (N\ a) \equiv N\ 0 \\
prop\_var\_rule :: Bool \\
prop\_var\_rule = sd\ X \equiv N\ 1 \\
prop\_sum\_rule :: (Real\ a, Floating\ a) \Rightarrow ExpAr\ a \rightarrow ExpAr\ a \rightarrow Bool \\
prop\_sum\_rule\ exp1\ exp2 = sd\ (Bin\ Sum\ exp1\ exp2) \equiv sum\_rule\ \mathbf{where} \\
\quad sum\_rule = Bin\ Sum\ (sd\ exp1)\ (sd\ exp2) \\
prop\_product\_rule :: (Real\ a, Floating\ a) \Rightarrow ExpAr\ a \rightarrow ExpAr\ a \rightarrow Bool \\
prop\_product\_rule\ exp1\ exp2 = sd\ (Bin\ Product\ exp1\ exp2) \equiv prod\_rule\ \mathbf{where} \\
\quad prod\_rule = Bin\ Sum\ (Bin\ Product\ exp1\ (sd\ exp2))\ (Bin\ Product\ (sd\ exp1)\ exp2) \\
prop\_e\_rule :: (Real\ a, Floating\ a) \Rightarrow ExpAr\ a \rightarrow Bool \\
prop\_e\_rule\ exp = sd\ (Un\ E\ exp) \equiv Bin\ Product\ (Un\ E\ exp)\ (sd\ exp) \\
prop\_negate\_rule :: (Real\ a, Floating\ a) \Rightarrow ExpAr\ a \rightarrow Bool \\
prop\_negate\_rule\ exp = sd\ (Un\ Negate\ exp) \equiv Un\ Negate\ (sd\ exp)
\end{array}
$$

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 cálculando 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.*

   $$
   \begin{array}{l}
   prop\_congruent :: (Floating\ a, Real\ a) \Rightarrow a \rightarrow ExpAr\ a \rightarrow Bool \\
   prop\_congruent\ a\ exp = ad\ a\ exp \stackrel{?}{=} eval\_exp\ a\ (sd\ exp)
   \end{array}
   $$

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 $\mathsf{F}\ X = 1 + X$) 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

$$
\begin{array}{l}
fib\ 0 = 1 \\
fib\ (n+1) = f\ n
\end{array}
$$

[4] Lei (3.94) em [?], página 98.

$$f\ 0 = 1$$
$$f\ (n+1) = \mathit{fib}\ n + f\ n$$

Obter-se-á de imediato

$$\begin{array}{l} \mathit{fib}' = \pi_1 \cdot \mathsf{for}\ \mathit{loop}\ \mathit{init}\ \mathbf{where} \\ \quad \mathit{loop}\ (\mathit{fib}, f) = (f, \mathit{fib} + f) \\ \quad \mathit{init} = (1, 1) \end{array}$$

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 coleccionam-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[6], de $f\ x = ax^2 + bx + c$ derivam-se duas funções mutuamente recursivas:

$$\begin{array}{l} f\ 0 = c \\ f\ (n+1) = f\ n + k\ n \\ k\ 0 = a + b \\ k\ (n+1) = k\ n + 2\ a \end{array}$$

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

$$\begin{array}{l} f'\ a\ b\ c = \pi_1 \cdot \mathsf{for}\ \mathit{loop}\ \mathit{init}\ \mathbf{where} \\ \quad \mathit{loop}\ (f, k) = (f + k, k + 2 * a) \\ \quad \mathit{init} = (c, a + b) \end{array}$$

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

$$\mathit{cat} = \cdots \cdot \mathsf{for}\ \mathit{loop}\ \mathit{init}\ \mathbf{where}\ \cdots$$

que implemente esta função.

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

$$\mathit{prop\_cat} = (\geqslant 0) \Rightarrow (\mathit{catdef} \equiv \mathit{cat})$$

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 $\{P_0, ..., P_N\}$ 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.

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

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

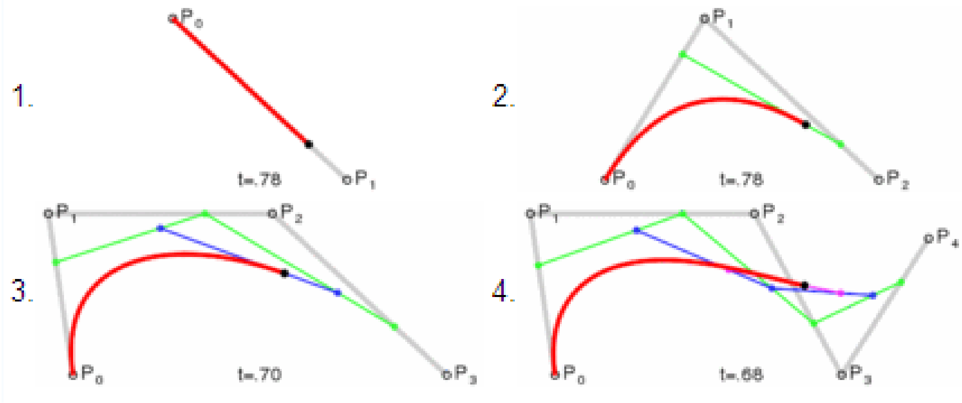

Figura 1: Exemplos de curvas de Bézier retirados da Wikipedia.

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:

$$\begin{aligned}
&linear1d :: \mathbb{Q} \to \mathbb{Q} \to OverTime\ \mathbb{Q}\\
&linear1d\ a\ b = formula\ a\ b\ \mathbf{where}\\
&\quad formula :: \mathbb{Q} \to \mathbb{Q} \to Float \to \mathbb{Q}\\
&\quad formula\ x\ y\ t = ((1.0 :: \mathbb{Q}) - (to_{\mathbb{Q}}\ t)) * x + (to_{\mathbb{Q}}\ t) * y
\end{aligned}$$

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.

$$\mathbf{type}\ NPoint = [\mathbb{Q}]$$

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

$$\begin{aligned}
&p2d = [1.2, 3.4]\\
&p3d = [0.2, 10.3, 2.4]
\end{aligned}$$

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

$$\mathbf{type}\ OverTime\ a = Float \to a$$

O anexo C tem definida a função

$$calcLine :: NPoint \to (NPoint \to OverTime\ NPoint)$$

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

$$deCasteljau :: [NPoint] \to OverTime\ NPoint$$

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.*

   $$\begin{aligned}
   &prop_calcLine_def :: NPoint \to NPoint \to Float \to Bool\\
   &prop_calcLine_def\ p\ q\ d = calcLine\ p\ q\ d \equiv zipWithM\ linear1d\ p\ q\ d
   \end{aligned}$$

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

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

$$
\begin{array}{l}
\mathit{prop\_bezier\_sym} :: [[\mathbb{Q}]] \rightarrow \mathit{Gen}\ \mathit{Bool} \\
\mathit{prop\_bezier\_sym}\ l = \mathit{all}\ (<\Delta) \cdot \mathit{calc\_difs} \cdot \mathit{bezs}\ \langle\$\rangle\ \mathit{elements}\ \mathit{ps}\ \mathbf{where} \\
\quad \mathit{calc\_difs} = (\lambda(x,y) \rightarrow \mathit{zipWith}\ (\lambda w\ v \rightarrow \mathbf{if}\ w \geqslant v\ \mathbf{then}\ w - v\ \mathbf{else}\ v - w)\ x\ y) \\
\quad \mathit{bezs}\ t = (\mathit{deCasteljau}\ l\ t, \mathit{deCasteljau}\ (\mathit{reverse}\ l)\ (\mathit{from}_{\mathbb{Q}}\ (1 - (\mathit{to}_{\mathbb{Q}}\ t)))) \\
\quad \Delta = 1\mathrm{e}{-}2
\end{array}
$$

3. Corra a função *runBezier* e aprecie o seu trabalho[7] clicando na janela que é aberta (que contém, a verde, um ponto inicila) 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,

$$\mathit{avg}\ x = \frac{1}{k}\sum_{i=1}^{k} x_i \tag{2}$$

onde $k = \mathit{length}\ x$. 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 facil de ver que

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

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 $\mathit{avg\_aux} = (\![[b, q]]\!)$ tal que $\mathit{avg\_aux} = \langle \mathit{avg}, \mathit{length} \rangle$ 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:*

$$
\begin{array}{l}
\mathit{prop\_avg} :: [\mathit{Double}] \rightarrow \mathit{Property} \\
\mathit{prop\_avg} = \mathit{nonempty} \Rightarrow \mathit{diff} \leqslant \underline{0.000001}\ \mathbf{where} \\
\quad \mathit{diff}\ l = \mathit{avg}\ l - (\mathit{avgLTree} \cdot \mathit{genLTree})\ l \\
\quad \mathit{genLTree} = [\![(\mathit{lsplit})]\!] \\
\quad \mathit{nonempty} = (>[\,])
\end{array}
$$

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.

[7] 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:[8]

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

Os diagramas podem ser produzidos recorrendo à *package* LaTeX xymatrix, por exemplo:

$$
\begin{tikzcd}
\mathbb{N}_0 \arrow[d, "(\!|g|\!)"'] & 1 + \mathbb{N}_0 \arrow[l, "\mathsf{in}"'] \arrow[d, "id + (\!|g|\!)"] \\
B & 1 + B \arrow[l, "g"]
\end{tikzcd}
$$

B Programação dinâmica por recursividade múltipla

Neste anexo dão-se os detalhes da resolução do Exercício 3.30 dos apontamentos da disciplina[9], 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:

$$
exp\ x = \sum_{i=0}^{\infty} \frac{x^i}{i!} \tag{3}
$$

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:

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

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

$$
\begin{array}{l}
e'\ x = prj \cdot \mathsf{for}\ loop\ init\ \mathbf{where} \\
\quad init = (1, x, 2) \\
\quad loop\ (e, h, s) = (h + e, x\ /\ s * h, 1 + s) \\
\quad prj\ (e, h, s) = e
\end{array}
$$

[8] Exemplos tirados de [?].
[9] Cf. [?], página 102.

C Código fornecido

Problema 1

$$
\begin{aligned}
&expd :: Floating\ a \Rightarrow a \rightarrow a\\
&expd = Prelude.exp\\
&\mathbf{type}\ OutExpAr\ a = () + (a + ((BinOp, (ExpAr\ a, ExpAr\ a)) + (UnOp, ExpAr\ a)))
\end{aligned}
$$

Problema 2

Definição da série de Catalan usando factoriais (1):

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

Oráculo para inspecção dos primeiros 26 números de Catalan[10]:

$$
\begin{aligned}
&oracle = [\\
&\quad 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, 208012, 742900, 2674440, 9694845,\\
&\quad 35357670, 129644790, 477638700, 1767263190, 6564120420, 24466267020,\\
&\quad 91482563640, 343059613650, 1289904147324, 4861946401452\\
&\quad]
\end{aligned}
$$

Problema 3

Algoritmo:

$$
\begin{aligned}
&deCasteljau :: [NPoint] \rightarrow OverTime\ NPoint\\
&deCasteljau\ [\,] = nil\\
&deCasteljau\ [p] = \underline{p}\\
&deCasteljau\ l = \lambda pt \rightarrow (calcLine\ (p\ pt)\ (q\ pt))\ pt\ \mathbf{where}\\
&\quad p = deCasteljau\ (init\ l)\\
&\quad q = deCasteljau\ (tail\ l)
\end{aligned}
$$

Função auxiliar:

$$
\begin{aligned}
&calcLine :: NPoint \rightarrow (NPoint \rightarrow OverTime\ NPoint)\\
&calcLine\ [\,] = \underline{nil}\\
&calcLine\ (p : x) = \overline{g}\ p\ (calcLine\ x)\ \mathbf{where}\\
&\quad g :: (\mathbb{Q}, NPoint \rightarrow OverTime\ NPoint) \rightarrow (NPoint \rightarrow OverTime\ NPoint)\\
&\quad g\ (d, f)\ l = \mathbf{case}\ l\ \mathbf{of}\\
&\qquad [\,] \rightarrow nil\\
&\qquad (x : xs) \rightarrow \lambda z \rightarrow concat\ \$\ (sequenceA\ [singl \cdot linear1d\ d\ x, f\ xs])\ z
\end{aligned}
$$

2D:

$$
\begin{aligned}
&bezier2d :: [NPoint] \rightarrow OverTime\ (Float, Float)\\
&bezier2d\ [\,] = \underline{(0, 0)}\\
&bezier2d\ l = \lambda z \rightarrow (from_{\mathbb{Q}} \times from_{\mathbb{Q}}) \cdot (\lambda[x, y] \rightarrow (x, y))\ \$\ ((deCasteljau\ l)\ z)
\end{aligned}
$$

Modelo:

$$
\begin{aligned}
&\mathbf{data}\ World = World\ \{points :: [NPoint]\\
&\quad , time :: Float\\
&\quad \}\\
&initW :: World\\
&initW = World\ [\,]\ 0
\end{aligned}
$$

[10] Fonte: Wikipedia.

$$
\begin{array}{l}
tick :: Float \to World \to World \\
tick\ dt\ world = world\ \{time = (time\ world) + dt\} \\
actions :: Event \to World \to World \\
actions\ (EventKey\ (MouseButton\ LeftButton)\ Down\ \_\ p)\ world = \\
\quad world\ \{points = (points\ world) +\!\!+ [(\lambda(x,y) \to \mathsf{map}\ \ to_{\mathbb{Q}}\ [x,y])\ p]\} \\
actions\ (EventKey\ (SpecialKey\ KeyDelete)\ Down\ \_\ \_)\ world = \\
\quad world\ \{points = cond\ (\equiv [\,])\ id\ init\ (points\ world)\} \\
actions\ \_\ world = world \\
scaleTime :: World \to Float \\
scaleTime\ w = (1 + cos\ (time\ w))\ /\ 2 \\
bezier2dAtTime :: World \to (Float, Float) \\
bezier2dAtTime\ w = (bezier2dAt\ w)\ (scaleTime\ w) \\
bezier2dAt :: World \to OverTime\ (Float, Float) \\
bezier2dAt\ w = bezier2d\ (points\ w) \\
thicCirc :: Picture \\
thicCirc = ThickCircle\ 4\ 10 \\
ps :: [Float] \\
ps = \mathsf{map}\ \ from_{\mathbb{Q}}\ ps'\ \textbf{where} \\
\quad ps' :: [\mathbb{Q}] \\
\quad ps' = [0, 0.01\,..\,1] \quad \text{-- interval}
\end{array}
$$

Gloss:

$$
\begin{array}{l}
picture :: World \to Picture \\
picture\ world = Pictures \\
\quad [animateBezier\ (scaleTime\ world)\ (points\ world) \\
\quad , Color\ white \cdot Line \cdot \mathsf{map}\ \ (bezier2dAt\ world)\ \$\ ps \\
\quad , Color\ blue \cdot Pictures\ \$\ [Translate\ (from_{\mathbb{Q}}\ x)\ (from_{\mathbb{Q}}\ y)\ thicCirc \mid [x,y] \leftarrow points\ world] \\
\quad , Color\ green\ \$\ Translate\ cx\ cy\ thicCirc \\
\quad]\ \textbf{where} \\
\quad (cx, cy) = bezier2dAtTime\ world
\end{array}
$$

Animação:

$$
\begin{array}{l}
animateBezier :: Float \to [NPoint] \to Picture \\
animateBezier\ \_\ [\,] = Blank \\
animateBezier\ \_\ [\_] = Blank \\
animateBezier\ t\ l = Pictures \\
\quad [animateBezier\ t\ (init\ l) \\
\quad , animateBezier\ t\ (tail\ l) \\
\quad , Color\ red \cdot Line\ \$\ [a, b] \\
\quad , Color\ orange\ \$\ Translate\ ax\ ay\ thicCirc \\
\quad , Color\ orange\ \$\ Translate\ bx\ by\ thicCirc \\
\quad]\ \textbf{where} \\
\quad a@(ax, ay) = bezier2d\ (init\ l)\ t \\
\quad b@(bx, by) = bezier2d\ (tail\ l)\ t
\end{array}
$$

Propriedades e *main*:

$$
\begin{array}{l}
runBezier :: \mathsf{IO}\ () \\
runBezier = play\ (InWindow\ \texttt{"Bézier"}\ (600, 600)\ (0, 0)) \\
\quad black\ 50\ initW\ picture\ actions\ tick \\
runBezierSym :: \mathsf{IO}\ () \\
runBezierSym = quickCheckWith\ (stdArgs\ \{maxSize = 20, maxSuccess = 200\})\ prop\_bezier\_sym
\end{array}
$$

Compilação e execução dentro do interpretador:[11]

$$
\begin{array}{l}
main = runBezier \\
run = \textbf{do}\ \{system\ \texttt{"ghc cp2021t"}; system\ \texttt{"./cp2021t"}\}
\end{array}
$$

[11] 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:

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Arbitrary}\ \mathit{UnOp}\ \textbf{where} \\
\quad \mathit{arbitrary} = \mathit{elements}\ [\mathit{Negate}, E] \\
\textbf{instance}\ \mathit{Arbitrary}\ \mathit{BinOp}\ \textbf{where} \\
\quad \mathit{arbitrary} = \mathit{elements}\ [\mathit{Sum}, \mathit{Product}] \\
\textbf{instance}\ (\mathit{Arbitrary}\ a) \Rightarrow \mathit{Arbitrary}\ (\mathit{ExpAr}\ a)\ \textbf{where} \\
\quad \mathit{arbitrary} = \textbf{do} \\
\qquad \mathit{binop} \leftarrow \mathit{arbitrary} \\
\qquad \mathit{unop} \leftarrow \mathit{arbitrary} \\
\qquad \mathit{exp1} \leftarrow \mathit{arbitrary} \\
\qquad \mathit{exp2} \leftarrow \mathit{arbitrary} \\
\qquad a \quad\ \leftarrow \mathit{arbitrary} \\
\qquad \mathit{frequency} \cdot \mathsf{map}\ \ (\mathit{id} \times \mathit{pure})\ \$\ [(20, X), (15, N\ a), (35, \mathit{Bin}\ \mathit{binop}\ \mathit{exp1}\ \ \mathit{exp2}), (30, \mathit{Un}\ \mathit{unop}\ \mathit{exp1})] \\
\textbf{infixr}\ 5 \stackrel{?}{=} \\
(\stackrel{?}{=}) :: \mathit{Real}\ a \Rightarrow a \rightarrow a \rightarrow \mathit{Bool} \\
(\stackrel{?}{=})\ x\ y = (\mathit{to}_{\mathbb{Q}}\ x) \equiv (\mathit{to}_{\mathbb{Q}}\ y)
\end{array}
$$

Outras funções auxiliares

Lógicas:

$$
\begin{array}{l}
\textbf{infixr}\ 0 \Rightarrow \\
(\Rightarrow) :: (\mathit{Testable}\ \mathit{prop}) \Rightarrow (a \rightarrow \mathit{Bool}) \rightarrow (a \rightarrow \mathit{prop}) \rightarrow a \rightarrow \mathit{Property} \\
p \Rightarrow f = \lambda a \rightarrow p\ \ a \Rightarrow f\ \ a \\
\textbf{infixr}\ 0 \Leftrightarrow \\
(\Leftrightarrow) :: (a \rightarrow \mathit{Bool}) \rightarrow (a \rightarrow \mathit{Bool}) \rightarrow a \rightarrow \mathit{Property} \\
p \Leftrightarrow f = \lambda a \rightarrow (p\ \ a \Rightarrow \mathit{property}\ (f\ \ a))\ .\&\&.\ (f\ \ a \Rightarrow \mathit{property}\ (p\ \ a)) \\
\textbf{infixr}\ 4 \equiv \\
(\equiv) :: \mathit{Eq}\ b \Rightarrow (a \rightarrow b) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow \mathit{Bool}) \\
f \equiv g = \lambda a \rightarrow f\ \ a \equiv g\ \ a \\
\textbf{infixr}\ 4 \leqslant \\
(\leqslant) :: \mathit{Ord}\ b \Rightarrow (a \rightarrow b) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow \mathit{Bool}) \\
f \leqslant g = \lambda a \rightarrow f\ \ a \leqslant g\ \ a \\
\textbf{infixr}\ 4 \wedge \\
(\wedge) :: (a \rightarrow \mathit{Bool}) \rightarrow (a \rightarrow \mathit{Bool}) \rightarrow (a \rightarrow \mathit{Bool}) \\
f \wedge g = \lambda a \rightarrow ((f\ \ a) \wedge (g\ \ a))
\end{array}
$$

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, disgramas 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:

$$
\begin{array}{l}
\mathit{cataExpAr}\ g = g \cdot \mathit{recExpAr}\ (\mathit{cataExpAr}\ g) \cdot \mathit{outExpAr} \\
\mathit{anaExpAr}\ g = \mathit{inExpAr} \cdot \mathit{recExpAr}\ (\mathit{anaExpAr}\ g) \cdot g \\
\mathit{hyloExpAr}\ h\ g = \mathit{cataExpAr}\ h \cdot \mathit{anaExpAr}\ g
\end{array}
$$

$eval\_exp :: Floating\ a \Rightarrow a \rightarrow (ExpAr\ a) \rightarrow a$
$eval\_exp\ a = cataExpAr\ (g\_eval\_exp\ a)$

$optmize\_eval :: (Floating\ a, Eq\ a) \Rightarrow a \rightarrow (ExpAr\ a) \rightarrow a$
$optmize\_eval\ a = hyloExpAr\ (gopt\ a)\ clean$

$sd :: Floating\ a \Rightarrow ExpAr\ a \rightarrow ExpAr\ a$
$sd = \pi_2 \cdot cataExpAr\ sd\_gen$

$ad :: Floating\ a \Rightarrow a \rightarrow ExpAr\ a \rightarrow a$
$ad\ v = \pi_2 \cdot cataExpAr\ (ad\_gen\ v)$

Definir:

$outExpAr\ X = i_1\ ()$
$outExpAr\ (N\ a) = i_2\ (i_1\ (a))$
$outExpAr\ (Bin\ op\ a\ b) = i_2\ (i_2\ (i_1\ (op, (a, b))))$
$outExpAr\ (Un\ op\ a) = i_2\ (i_2\ (i_2\ (op, a)))$
--
$recExpAr\ h = id + (id + (id \times (h \times h) + id \times h))$
--
$g\_eval\_exp\ a = [\underline{a}, nums]$ **where**
$nums = [numb, ops]$
$ops = [bins, uns]$
$numb\ b = b$
$bins\ (Sum, (a, b)) = a + b$
$bins\ (Product, (a, b)) = a * b$
$uns\ (Negate, a) = a * (-1)$
$uns\ (E, a) = Prelude.exp\ a$
--
$clean\ X = outExpAr\ X$
$clean\ (N\ a) = outExpAr\ (N\ a)$
$clean\ (Bin\ Sum\ a\ b) = outExpAr\ (Bin\ Sum\ a\ b)$
$clean\ (Bin\ Product\ (N\ 0)\ b) = outExpAr\ (N\ 0)$
$clean\ (Bin\ Product\ a\ (N\ 0)) = outExpAr\ (N\ 0)$
$clean\ (Bin\ Product\ a\ b) = outExpAr\ (Bin\ Product\ a\ b)$
$clean\ (Un\ Negate\ a) = outExpAr\ (Un\ Negate\ a)$
$clean\ (Un\ E\ (N\ 0)) = outExpAr\ (N\ 1)$
$clean\ (Un\ E\ a) = outExpAr\ (Un\ E\ a)$
--
$gopt\ a = g\_eval\_exp\ a$

$sd\_gen :: Floating\ a \Rightarrow$
$() + (a + ((BinOp, ((ExpAr\ a, ExpAr\ a), (ExpAr\ a, ExpAr\ a))) + (UnOp, (ExpAr\ a, ExpAr\ a)))) \rightarrow (ExpAr\ a$
$sd\_gen = [derivx, nums]$ **where**
$nums = [numb, ops]$
$ops = [bins, uns]$
$derivx\ () = (X, N\ 1)$
$numb\ b = (N\ b, N\ 0)$
$bins\ (Sum, ((a1, a2), (b1, b2))) = (Bin\ Sum\ a1\ b1, Bin\ Sum\ a2\ b2)$
$bins\ (Product, ((a1, a2), (b1, b2))) = (Bin\ Product\ a1\ b1, Bin\ Sum\ (Bin\ Product\ a1\ b2)\ (Bin\ Product\ a2\ b1)$
$uns\ (E, (a, b)) = (Un\ E\ a, Bin\ Product\ (Un\ E\ a)\ b)$
$uns\ (Negate, (a, b)) = (Un\ Negate\ a, Un\ Negate\ b)$

$ad\_gen\ v = [derivx, nums]$ **where**
$nums = [numb, ops]$
$ops = [bins, uns]$
$derivx\ () = (v, 1)$
$numb\ b = (b, 0)$
$bins\ (Sum, ((a1, a2), (b1, b2))) = (a1 + b1, a2 + b2)$

$$bins\ (Product, ((a1, a2), (b1, b2))) = (a1 * b1, (a1 * b2) + (a2 * b1))$$
$$uns\ (E, (a, b)) = (Prelude.exp\ a, b * (Prelude.exp\ a))$$
$$uns\ (Negate, (a, b)) = (a * (-1), b * (-1))$$

Alínea 1 Resolução
Alinea 1:
Sabemos que por se tratar de um isomorfismo, $outExpAr \cdot inExpAr = id$, logo temos:

$$\begin{aligned}
& outExpAr \cdot inExpAr = id \\
\equiv \quad & \{ \text{ Def-inExpAr, Reflexão-+, Fusão-+ } \} \\
& [outExpAr \cdot \underline{X}, outExpAr \cdot num_ops] = [i_1, i_2] \\
\equiv \quad & \{ \text{ Eq-+, Def-}num_ops\text{, Natural-id } \} \\
& \begin{cases} outExpAr \cdot \underline{X} = i_1 \\ outExpAr \cdot [N, ops] = i_2 \cdot id \end{cases} \\
\equiv \quad & \{ \text{ Reflexão-+, Fusão-+ aplicada 2 vezes } \} \\
& \begin{cases} outExpAr \cdot \underline{X} = i_1 \\ [outExpAr \cdot N, outExpAr \cdot ops] = [i_2 \cdot i_1, i_2 \cdot i_2] \end{cases} \\
\equiv \quad & \{ \text{ Eq-+, Def-}ops\text{, Natural-id } \} \\
& \begin{cases} outExpAr \cdot \underline{X} = i_1 \\ outExpAr \cdot N = i_2 \cdot i_1 \\ outExpAr \cdot [bin, \widehat{Un}] = i_2 \cdot i_2 \cdot id \end{cases} \\
\equiv \quad & \{ \text{ Reflexão-+, Fusão-+ aplicada 2 vezes } \} \\
& \begin{cases} outExpAr \cdot \underline{X} = i_1 \\ outExpAr \cdot N = i_2 \cdot i_1 \\ [outExpAr \cdot bin, outExpAr \cdot \widehat{Un}] = [i_2 \cdot i_2 \cdot i_1, i_2 \cdot i_2 \cdot i_2] \end{cases} \\
\equiv \quad & \{ \text{ Eq-+ } \} \\
& \begin{cases} outExpAr \cdot \underline{X} = i_1 \\ outExpAr \cdot N = i_2 \cdot i_1 \\ outExpAr \cdot bin = i_2 \cdot i_2 \cdot i_1 \\ outExpAr \cdot \widehat{Un} = i_2 \cdot i_2 \cdot i_2 \end{cases} \\
\equiv \quad & \{ \text{ Igualdade Extensional } \} \\
& \begin{cases} outExpAr \cdot \underline{X}\ () = i_1\ () \\ outExpAr \cdot N\ a = i_2 \cdot i_1\ a \\ outExpAr \cdot bin\ (op, (a, b)) = i_2 \cdot i_2 \cdot i_1\ (op, (a, b)) \\ outExpAr \cdot \widehat{Un}\ (op, a) = i_2 \cdot i_2 \cdot i_2\ (op, a) \end{cases} \\
\equiv \quad & \{ \text{ Def-const, Def-Comp, Def-}bin\text{, Def-uncurry } \} \\
& \begin{cases} outExpAr\ \underline{X} = i_1\ () \\ outExpAr\ (N\ a) = i_2\ (i_1\ (a)) \\ outExpAr\ (Bin\ op\ a\ b) = i_2\ (i_2\ (i_1\ (op, (a, b)))) \\ outExpAr\ (Un\ op\ a) = i_2\ (i_2\ (i_2\ (op, a))) \end{cases}
\end{aligned}$$

Sabendo que *recExpAr* irá aplicar uma função que recebe como argumento a todos os elementos que resultam de *outExpAr* Assim, a partir de *baseExpAr* e do sseguinte diagrama:

$$\begin{tikzcd}
ExpAr\ a \arrow[rr, bend left, "outExpAr"] \arrow[r, phantom, "\cong"] & {} & 1 + (a + ((BinOp \times ExpAr\ a^2) + (UnOp \times ExpAr\ a))) \arrow[ll, bend left, "inExpAr"]
\end{tikzcd}$$

Temos que:

$$recExpArf = id + id + id \times f^2 + id \times f$$

Alínea 2:
(Nas alíneas seguintes $ExpAr\ a$ será substituido por $ExpAr$ apenas por simplificação)
Sabendo que a função g_eval_exp se trata do gene do catamorfismo $eval\_exp$ que tem como objetivo calcular o valor de uma $ExpAr$ (dado um valor para substituir a incognita), tendo ainda em conta a definição de $recExpAr$ temos o diagrama:

$$\begin{tikzcd}
ExpAr \arrow[d, "eval_exp"'] & 1 + (\mathbb{N}_0 + ((BinOp \times ExpAr \times ExpAr) + (UnOp \times ExpAr))) \arrow[l, "inExpAr"'] \arrow[d, "id+(id+(id\times eval_exp\times eval_exp+id\times eval_exp))"] \\
a & 1 + (a + ((BinOp \times a \times a) + (UnOp \times a))) \arrow[l, "g_eval_exp"]
\end{tikzcd}$$

Assim, prevendo as hipoteses existentes, temos que um $BinOp$ pode corresponder a uma soma ou a um produto e que um $UnOp$ pode corresponder a um $Negate$ ou a um E, tendo em conta as propriedades do calculo a que correspondem estas expressões teremos que o gene de $eval\_exp$ será:

$$g\_eval\_exp\ a = [\underline{a}, [numb, [bins, uns]]]$$

Com $numb$, $bins$ e uns, que devolvem o valor de um numero b, o resultado de uma operação binária e o resultado de uma operação unária respetivamente.

Alinea 3:
Nesta alínea pede-se que se implemente os genes da função $optimize\_eval$ que têm como objetivo calcular o valor de uma expressão mas de uma forma mais eficiente. Sendo $clean$ o gene do anamorfismo responsável por limpar os casos em que ocorre um elemento absorvente da operação e $gopt$ o gene do catamorfismo responsável por calcular o valor de uma expressão temos os diagramas:

$$\begin{tikzcd}
ExpAr \arrow[r, "clean"'] \arrow[d, "ana\ clean"'] & 1 + (a + ((BinOp \times ExpAr \times ExpAr) + (UnOp \times ExpAr))) \arrow[d, "id+(id+(id\times(ana\ clean)\times(ana\ clean)+id\times(ana\ clean)))"] \\
ExpAr \arrow[r, "outExpAr"] & 1 + (a + ((BinOp \times ExpAr \times ExpAr) + (UnOp \times ExpAr)))
\end{tikzcd}$$

$$\begin{tikzcd}
ExpAr \arrow[d, "(\!|gopt|\!)"'] & 1 + (\mathbb{N}_0 + ((BinOp \times ExpAr \times ExpAr) + (UnOp \times ExpAr))) \arrow[l, "inExpAr"'] \arrow[d, "id+(id+(id\times(\!|gopt|\!)\times(\!|gopt|\!)+id\times(\!|gopt|\!)))"] \\
a & 1 + (a + ((BinOp \times a \times a) + (UnOp \times a))) \arrow[l, "gopt"]
\end{tikzcd}$$

Assim sendo $clean$ poderá ser definida usando $outExpAr$. Tendo todas estas situações em conta, sabendo que o elemento absorvente da multiplicação é 0 e sabendo ainda que e^0 é sempre igual a 1 a função clean irá abordar estas situações, nas restantes irá aplicar apenas a função $outExpAr$.

Já a $(\!|gopt|\!)$ terá uma função semelhante à de $eval\_exp$, calcular o valor de uma $ExpAr$, logo podemos definir $gopt$ como sendo igual a g_eval_exp.

Alínea 4
Nesta alínea pretende-se definir o gene do catamorfismo de sd que tem como objetivo calcular a derivada de uma expressão. Sabemos que, de acordo com as regras das derivadas por vezes vamos precisar da expressão original (isto é, não derivada) para aplicar outras leis de derivação. Para isso pretende-se, ao calcular a derivada de uma expressão, possibilitar o acesso à expressão original, usando-se para isso, tal como sugerido pela forma como sd é definido no enunciado, um split em que o lado esquerdo corresponde â expressão original e o direito à derivada. Assim, temos o diagrama:

$$\begin{tikzcd}
ExpAr \arrow[d, "(\!|sd_gen|\!)"'] & 1 + (a + ((BinOp \times ExpAr \times ExpAr) + (UnOp \times ExpAr))) \arrow[l, "inExpAr"'] \arrow[d, "id+(id+(id\times(\!|sd_gen|\!)\times(\!|sd_gen|\!)+id\times(\!|sd_gen|\!)))"] \\
ExpAr^2 & 1 + (a + ((BinOp \times (ExpAr)^2 \times (ExpAr)^2) + (UnOp \times (ExpAr)^2))) \arrow[l, "sd_gen"]
\end{tikzcd}$$

Daqui temos:

$$sd\_gen = [\,derivx, [\,numb, [\,bins, uns\,]\,]\,]$$

Assim, tendo em conta as regras de derivação para as diferentes operações chegamos à definição apresentada. Com $derivx$ a calcular a derivada de uma incognita, $numb$ a calcular a derivada de um numero a, $bins$ e uns a calcularem a derivada das expressões binárias e unárias associadas ao tipo $ExpAr$.

Alinea 5
Por fim pede-se para determinar o gene do catamorfismo que calcula o valor (numerico) da derivada de uma expressão. De forma semelhante à alinea anterior é necessário guardar também o valor original da expressão derivada e ao mesmo tempo o valor da derivada. Assim temos o diagrama:

$$\begin{tikzcd}
ExpAr \arrow[d, "(\!|ad_gen|\!)"'] & 1 + (a + ((BinOp \times ExpAr \times ExpAr) + (UnOp \times ExpAr))) \arrow[l, "inExpAr"'] \arrow[d, "id+(id+(id\times(\!|ad_gen|\!)\times(\!|ad_gen|\!)+id\times(\!|ad_gen|\!)))"] \\
a^2 & 1 + (a + ((BinOp \times a^2 \times a^2) + (UnOp \times a^2))) \arrow[l, "ad_gen"]
\end{tikzcd}$$

Temos ainda:

$$sd\_gen = [\,derivx, [\,numb, [\,bins, uns\,]\,]\,]$$

Mais uma vez, tendo em conta as regras de derivação e que $ad = \pi_2 \cdot cataExpAr\ \ ad\_gen$ chegamos à solução apresentada em que $derivx$ calcula o valor da derivada de () que representa uma incogmita, $numb$ calcula o valor da derivada de um numero a e $bins$ e uns calculam o valor da derivada das expressões binárias e unárias associadas ao tipo $ExpAr$ da forma que é apresentado (mantendo sempre o valor da expressão original de forma a estar acessivel).

Problema 2

Definir

$$loop\ (a, b, c) = (quot\ (a * b)\ c, b + 4, c + 1)$$
$$inic = (1, 2, 2)$$
$$prj\ (a, b, c) = a$$

por forma a que

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

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

Definição do número n de Catalan:

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

Temos de imediato que $catal(0) = 1$. Devemos agora calcular $catal(n)$ como uma recursão, para podermos aplicar a *regra de algibeira* fornecida. Calculemos, então:

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

Logo

$$catal(n+1) = \frac{4n+2}{n+2} catal(n)$$

Temos, então, a função auxiliar $c(n) = \frac{4n+2}{n+2}$. Podemos decompor c em duas funções, $c1$ e $c2$, tais que: $c1(n) = 4n + 2$ e $c2(n) = n + 2$. Assim decomposta, é muito simples exprimir $c1$ e $c2$ à custa de si mesmas. Daqui, temos $c1(0) = c2(0) = 2$, $c1(n + 1) = 4 + c1(n)$ e $c2(n + 1) = 1 + c2(n)$.

$$catal\ 0 = 1$$
$$catal\ (n+1) = c1\ (n)\ catal\ (n)\ /\ c2\ (n)$$
$$c1\ 0 = 2$$
$$c1\ (n+1) = 4 + c1\ (n)$$
$$c2\ 0 = 2$$
$$c2\ (n+1) = 1 + c2\ (n)$$

Assim, estamos em condições de aplicar a *regra*, onde $inic = (1,\ 2,\ 2)$ e $loop(catal,\ c1,\ c2) = (quot\ (catal \cdot c1)\ \ c2,\ 4 + c1,\ 1 + c2)$.

Problema 3

$$\begin{array}{l}
calcLine :: NPoint \to (NPoint \to OverTime\ NPoint) \\
calcLine\ x1\ x2\ t = cataList\ h\ list\ \textbf{where} \\
\quad h = [nil, cons \cdot (ul1d\ t \times id)] \\
\quad list = zip\ x1\ x2 \\
\quad ul1d\ t\ p = \widehat{linear1d}\ p\ t \\
deCasteljau :: [NPoint] \to OverTime\ NPoint \\
deCasteljau\ [\,]\ _ = [\,] \\
deCasteljau\ l\ t = hyloAlgForm\ alg\ coalg\ l\ \textbf{where} \\
\quad coalg = divide \\
\quad alg = [id, aux] \\
\quad aux\ (a, b) = calcLine\ a\ b\ t \\
\quad divide\ [x] = i_1\ x \\
\quad divide\ l = i_2\ (init\ l, tail\ l) \\
hyloAlgForm = hyloLTree
\end{array}$$

Comecemos por simplificar a função fornecida $calcLine$, de modo a chegar a uma função equivalente mas mais simples, tornando o algoritmo em questão mais explícito. Temos:

$$\begin{array}{l}
calcLine :: NPoint \to (NPoint \to OverTime\ NPoint) \\
calcLine\ [\,] = \underline{nil} \\
calcLine\ (p : x) = \overline{g}\ p\ (calcLine\ x)\ \textbf{where} \\
\quad g :: (\mathbb{Q}, NPoint \to OverTime\ NPoint) \to (NPoint \to OverTime\ NPoint) \\
\quad g\ (d, f)\ l = \textbf{case}\ l\ \textbf{of} \\
\qquad [\,] \to nil \\
\qquad (x : xs) \to \lambda z \to concat\ \$\ (sequenceA\ [singl \cdot linear1d\ d\ x, f\ xs])\ z
\end{array}$$

Podemos, então, eliminar a função auxiliar g (ajustando nomes de variáveis):

$$\begin{array}{l}
calcLine :: NPoint \to (NPoint \to OverTime\ NPoint) \\
calcLine\ [\,] = \underline{nil} \\
calcLine\ (x1h : x1t) = \lambda x2\ \ t \to \textbf{case}\ x2\ \textbf{of} \\
\quad [\,] \to nil\ t \\
\quad x2h : x2t \to concat\ \$\ (sequenceA\ [singl \cdot linear1d\ x1h\ x2h, calcLine\ x1t\ x2t])\ t
\end{array}$$

Usando a definição de $sequenceA$, esta última linha fica da seguinte forma:

$$x2h : x2t \to concat\ [singl \cdot linear1d\ x1h\ x2h\ t, calcLine\ x1t\ x2t\ t]$$

Nesta situação, estamos a concatenar uma lista de um só elemento com uma outra, pelo que temos:

$$x2h : x2t \to cons\ (linear1d\ x1h\ x2h\ t, calcLine\ x1t\ x2t\ t)$$

Daqui, observamos que existem duas listas (em que cada uma representa um ponto) a serem consumidas em simultâneo. Isto porque a interpolação linear entre dois pontos só faz sentido para pontos com a mesma dimensão, ou seja, $length\ x1 == length\ x2$. No catamorfismo de listas, temos uma única lista a ser consumida. Para contornar este problema, podemos usar a função pré-definida do Haskell zip, cuja assinatura é:

$zip :: [\,a\,] \rightarrow [\,b\,] \rightarrow [\,(a, b)\,]$

Isto é possível porque as coordenadas correspondentes entre os dois pontos estão em posições iguais da respetiva lista. Portanto, em conformidade com o tipo de saída de zip, o catamorfismo em questão é aplicado a uma lista de pares. Como $NPoint$ é uma lista de $\mathbb{Q}$, serão pares de $\mathbb{Q}$.

$$\begin{tikzcd}
A^* \arrow[r, bend left, "outList"] \arrow[r, phantom, "\cong"] \arrow[d, "(\!|h|\!)"'] & 1 + A \times A^* \arrow[l, bend left, "inList"] \arrow[d, "id+id\times(\!|h|\!)"] \\
B & 1 + A \times B \arrow[l, "h"]
\end{tikzcd}$$

Já vimos que A será instanciando como $\mathbb{Q} \times \mathbb{Q}$, e pelo tipo de saída da função $calcLine$ temos que B é um $NPoint$, ou seja, $\mathbb{Q}^*$. Resta-nos calcular o gene h.

Para o caso da esquerda, respeitando a função dada, teremos nil. No caso da direita, temos o par cabeça e a cauda, já processada pelo algoritmo. Como $calcLine$ devolve um $NPoint$, devemos calcular a interpolação linear da cabeça usando a função $linear1d$ (*uncurried*) e usar $cons$ para a colocar na lista resultado. De notar ainda que $linear1d$ tem um parâmetro de entrada extra e portanto teremos de definir uma função auxiliar. Logo:

$h = [nil, cons \cdot (ul1d\ t \times id)]$
$\textbf{where}\ ul1d\ t\ p = \widehat{linear1d}\ p\ t$

Então:

$$\begin{tikzcd}
(\mathbb{Q} \times \mathbb{Q})^* \arrow[r, bend left, "outList"] \arrow[r, phantom, "\cong"] \arrow[d, "(\!|[nil, cons\cdot(ul1d\ t)\times id]|\!)"'] & 1 + (\mathbb{Q} \times \mathbb{Q}) \times (\mathbb{Q} \times \mathbb{Q})^* \arrow[l, bend left, "inList"] \arrow[d, "id+id\times(\!|[nil, cons\cdot(ul1d\ t)\times id]|\!)"] \\
\mathbb{Q}^* & 1 + (\mathbb{Q} \times \mathbb{Q}) \times \mathbb{Q}^* \arrow[l, "{[nil, cons\cdot(ul1d\ t)\times id]}"]
\end{tikzcd}$$

Temos, assim, a definição de $calcLine$ como um catamorfismo de listas.

$calcLine :: NPoint \rightarrow (NPoint \rightarrow OverTime\ NPoint)$
$calcLine\ x1\ x2\ t = cataList\ h\ list\ \textbf{where}$
$\quad h = [nil, cons \cdot (ul1d\ t \times id)]$
$\quad list = zip\ x1\ x2$
$\quad ul1d\ t\ p = \widehat{linear1d}\ p\ t$

O algoritmo $deCasteljau$ fornecido trata-se de um algoritmo do género *divide and conquer*, em que, para uma lista com um só elemento, o resultado é o elemento solitário da lista; para os restantes casos, ou seja, listas de N elementos, $\forall N > 1$, a função é chamada recursivamente para os primeiros $N - 1$ e para os últimos $N - 1$ elementos. O resultado de cada uma destas chamadas é depois calculado pela função $calcLine$. Nas aulas, foi estudado um algoritmo em tudo muito semelhante a este. Trata-se do *mergeSort*:

$mSort :: Ord\ a \Rightarrow [\,a\,] \rightarrow [\,a\,]$
$mSort\ [\,] = [\,]$
$mSort\ l = hyloLTree\ [singl, merge]\ lsplit\ l$
$merge\ (l, [\,]) = l$
$merge\ ([\,], r) = r$
$merge\ (x : xs, y : ys)$

$$
\begin{array}{l}
\quad \mid x < y \qquad = x : merge\ (xs, y : ys) \\
\quad \mid otherwise = y : merge\ (x : xs, ys) \\
lsplit\ [x] = i_1\ x \\
lsplit\ l = i_2\ (sep\ l)\ \mathbf{where} \\
\quad sep\ [\,] = ([\,], [\,]) \\
\quad sep\ (h : t) = \mathbf{let}\ (l, r) = sep\ t\ \mathbf{in}\ (h : r, l)
\end{array}
$$

Podemos adaptá-lo para o problema em questão. Desde já, sabemos que o hilomorfismo a ser usado será o hilomorfismo das LTree. O gene do anamorfismo, como podemos observar, trata da parte *divide* do algoritmo. Como dissemos anteriormente, usamos a chamada recursiva para os primeiros e para os últimos $N - 1$ elementos. Vamos então defini-lo:

$$
\begin{array}{l}
coalg = divide \\
divide\ [x] = i_1\ x \\
divide\ l = i_2\ (init\ l, tail\ l)
\end{array}
$$

Agora, devemos encontrar o gene do catamorfismo. Para o caso em que o anamorfismo injeta à esquerda, temos um elemento solitário. Ora, para esse caso está trivialmente calculado o ponto a retornar (é o próprio ponto), logo usamos a função identidade. Já para o caso da direita, temos um par de pontos, resultantes da chamada recursiva do algoritmo para parte inicial e para a parte final da lista. A função recebe ainda um parâmetro extra, por isso usamos uma função auxiliar:

$$
\begin{array}{l}
alg = [id, aux] \\
aux\ (a, b) = calcLine\ a\ b\ t
\end{array}
$$

Assim, temos a solução para listas não vazias. Resta-nos o caso da lista vazia, que, respeitando o algoritmo, é a própria lista vazia:

$$
\begin{array}{l}
deCasteljau :: [NPoint] \rightarrow OverTime\ NPoint \\
deCasteljau\ [\,]\ \_ = [\,] \\
deCasteljau\ l\ t = hyloAlgForm\ alg\ coalg\ l\ \mathbf{where} \\
\quad coalg = divide \\
\quad alg = [id, aux] \\
\quad aux\ (a, b) = calcLine\ a\ b\ t \\
\quad divide\ [x] = i_1\ x \\
\quad divide\ l = i_2\ (init\ l, tail\ l)
\end{array}
$$

Problema 4

Solução para listas não vazias:

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

$$
\begin{array}{l}
inNL = [singl, cons] \\
outNL\ [a] = i_1\ (a) \\
outNL\ (a : x) = i_2\ (a, x) \\
cataNL\ g = g \cdot recList\ (cataNL\ g) \cdot outNL \\
avg\_aux = cataNL\ [f, g] \\
\quad \mathbf{where}\ f = \langle id, \underline{1} \rangle \\
\qquad g = \langle av, len \rangle \\
\qquad av\ (a, (b, c)) = (a + c * b)\ /\ (c + 1) \\
\qquad len\ (a, (b, c)) = c + 1
\end{array}
$$

Solução para árvores de tipo LTree:

$$
\begin{array}{l}
avgLTree = \pi_1 \cdot (\!|\ gene\ |\!)\ \mathbf{where} \\
\quad gene = [f, g] \\
\qquad \mathbf{where}\ f = \langle id, \underline{1} \rangle \\
\qquad\quad g = \langle av, len \rangle
\end{array}
$$

$$
\begin{aligned}
&av\ ((a,b),(c,d)) = (a * b + c * d)\ /\ (b + d)\\
&len\ ((a,b),(c,d)) = b + d
\end{aligned}
$$

Resolução:
Alínea 1.
A partir dos dados do problema, podemos inferir o seguinte diagrama:

$$
\begin{tikzcd}
T \arrow[r, bend left, "outNL"] \arrow[r, phantom, "\cong"] \arrow[d, "\llparenthesis g \rrparenthesis"'] & FT \arrow[l, bend left, "inNL"] \arrow[d, "F\ \llparenthesis g \rrparenthesis"] \\
A & FA \arrow[l, bend left, "FA"]
\end{tikzcd}
$$

Como estamos perante listas vazias, podemos também inferir será que $inNL = [singl, cons]$, pois, os dois únicos casos possíveis de uma lista deste tipo serão ou termos um elemento da lista, ou termos uma lista completa.
Falta-nos no entanto inferir $outNL$, no entanto, sabemos que, por se tratar de um isomorfismo, $outNL \cdot inNL = id$, logo de forma a obtermos $outNL$ vamos resolver essa equação:

$$
\begin{aligned}
& outNL \cdot inNL = id \\
\equiv \quad & \{ \text{ Def-}inNL \ \} \\
& out \cdot [singl, cons] = id \\
\equiv \quad & \{ \text{ Fusão-+ (lei 20) } \} \\
& [out \cdot singl, out \cdot cons] = id \\
\equiv \quad & \{ \text{ Universal-+ (lei 17) with k = id, f = out . singl, g = out . cons } \} \\
& \begin{cases} id \cdot i_1 = out \cdot singl \\ id \cdot i_2 = out \cdot cons \end{cases} \\
\equiv \quad & \{ \text{ Natural-id (lei 1) aplicada 2 vezes } \} \\
& \begin{cases} i_1 = out \cdot singl \\ i_2 = out \cdot cons \end{cases} \\
\equiv \quad & \{ \text{ Igualdade Extensional (lei 71) } \} \\
& \begin{cases} (out \cdot singl)\ x = i_1\ x \\ (out \cdot cons)\ x = i_2\ x \end{cases} \\
\equiv \quad & \{ \text{ Def-Comp (lei 72) aplicada 2 vezes } \} \\
& \begin{cases} out\ (singl\ x) = i_1\ x \\ out\ (cons\ (h,t)) = i_2\ (h,t) \end{cases} \\
\equiv \quad & \{ \text{ Def-singl, Def-cons } \} \\
& \begin{cases} out\ [x] = i_1\ x \\ out\ (h : t) = i_2\ (h,t) \end{cases}
\end{aligned}
$$

Tendo já definido $inNL$ e $outNL$, vamos agora procurar conhecer o catamorfismo.
Sabemos, de antemão, que o comprimento da lista será um natural positivo ou zero e que a média é um racional/double. Então podemos o seguinte diagrama:

$$
\begin{tikzcd}
L \arrow[r, bend left, "outNL"] \arrow[r, phantom, "\cong"] \arrow[d, "\llparenthesis g \rrparenthesis"'] & A + A \times L \arrow[l, bend left, "inNL"] \arrow[d, "id + id \times \llparenthesis g \rrparenthesis"] \\
(\mathbb{Q}, \mathbb{N}_0) & A + A \times (\mathbb{Q}, \mathbb{N}_0) \arrow[l, bend left, "g"]
\end{tikzcd}
$$

Podemos então observar que $\llparenthesis g \rrparenthesis = g \cdot (id + id \times \llparenthesis g \rrparenthesis) \cdot out$ e que o functor deste tipo de listas é dado por $F\ f = id + id \times f$.

Assim sendo, podemos ver que $(\!| g |\!) = g \cdot F\ (\!| g |\!) \cdot out$, o que, escrito em Haskell é dado por $cataNL = g \cdot recList\ (cataNL\ g) \cdot out$.
Temos, então, tudo o necessário para implementar a primeira alínea sendo que av faz o cálculo da média ponderada até ao nodo atual, len atualiza o comprimento da lista lido recursivamente, a representa o valor do nodo da lista atual, b a média calculada recursivamente e c o comprimento da lista já lido.

Alínea 2.
Tendo em conta que o catamorfismo já se encontra definido, não é necessário calculá-lo a ele, nem ao isomorfismo in/out, nem o functor. Logo, resta-nos apenas seguir a linha do pensamento anterior, para esta nova estrutura de dados.
Tal como na alínea anterior, av faz o cálculo da média ponderada até ao momento e len atualiza o número de valores lidos. A diferença surge apenas nas variáveis. Visto que estamos perante uma árvore, teremos de analisar dois lados ao invés de apenas um, então, como seria de esperar teremos duas variáveis que representam as médias ponderadas até ao nodo atual, a e c e outras duas que representam os tamanhos lidos recursivamente, b e d.

Problema 5

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

```
// (c) MP-I (1998/9-2006/7) and CP (2005/6-2016/7)

module BTree

open Cp

// (1) Datatype definition -----------------------------------------------------

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

let inBTree x = either (konst Empty) Node x

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

// (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 c a x = ((cataBTree c) << (anaBTree a)) x

// (3) Map ---------------------------------------------------------------------

let fmap f t = cataBTree (inBTree << baseBTree f id) t

// (4) Examples ----------------------------------------------------------------

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

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

```
// (4.2) Counting --------------------------------------------------------------

let countBTree x = cataBTree (either (konst 0) (succ << (uncurry (+)) << p2)) x

// (4.3) Serialization ---------------------------------------------------------
let singl x = [x]

let inord t =
    let join(x,(l,r)) = l @ (singl x) @ r
    in (either nil join) t

let inordt t = cataBTree inord t

let preord t =
    let f(x,(l,r)) = x :: l @ r
    in (either nil f) t

let preordt t = cataBTree preord t

let postord t =
    let f(x,(l,r)) = l @ r @ (singl x)
    in (either nil f) t

let postordt t = cataBTree postord t

// (4.4) Quicksort -------------------------------------------------------------

let app a l = a :: l

let rec part h t = //pivot / list
    match t with
    | [] -> ([],[])
    | x::xs -> if x < h then ((app x) >< id) (part h xs)
                         else (id >< (app x)) (part h xs)

let qsep list =
    match list with
    | [] -> i1 ()
    | h::t -> let (s,l) = part h t
              in i2 (h,(s,l))

let qSort x = hyloBTree inord qsep x

// (4.5) Traces ----------------------------------------------------------------

let rec union l1 l2 =
    match l2 with
    | [] -> l1
    | h::t -> if List.exists (fun e -> e = h) l1 then union l1 t else union (l1 @ [h]

let tunion (a,(l,r)) = union (List.map (app a) l) (List.map (app a) r)

let traces x = (cataBTree (either (konst [[]]) tunion)) x

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

let strategy l =
```

```
    match l with
    | (d,0) -> i1 ()
    | (d,n) -> i2 ((n-1,d),((not d,n-1),(not d, n-1)))

let hanoi x = hyloBTree inord strategy x

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

let baldepth x =
    let h (a,((b1,b2),(d1,d2))) = (b1 && b2 && abs(d1-d2)<=1, 1 + (max d1 d2))
    let f ((b1,d1),(b2,d2)) = ((b1,b2),(d1,d2))
    let g g1 = either (konst (true, 1)) (h << (id >< f)) g1
    in (cataBTree g) x

let balBTree x = (p1 << baldepth) x

let depthBTree x = (p2 << baldepth) x

// (6) Going polytipic --------------------------------------------------------

let tnat f x =
    let theta (a,b) = a @ b
    in  (either (konst []) (theta << (f >< theta))) x

let monBTree f a = cataBTree (tnat f) a

// alternative to (4.3) serialization ----------------------------------------

let preordt' t = monBTree singl t

// (7) Zipper ------------------------------------------------------------------

type Deriv<'a> =
    | Dr of bool * ('a  * BTree<'a>)

type Zipper<'a> = List<Deriv<'a>>

let rec plug l t =
    match l with
    | [] -> t
    | (Dr (false,(a,l))::z)  -> Node (a,(plug z t,l))
    | (Dr (true,(a, r))::z)  -> Node (a,(r,plug z t))
```