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

Departamento de Informática
Universidade do Minho

Junho de 2021

Grupo nr.	51
a89557	Pedro Veloso
a89587	Carlos Preto
a93319	Catarina Morales

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 dependências 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.*

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

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

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

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 $eval$.*

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

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.

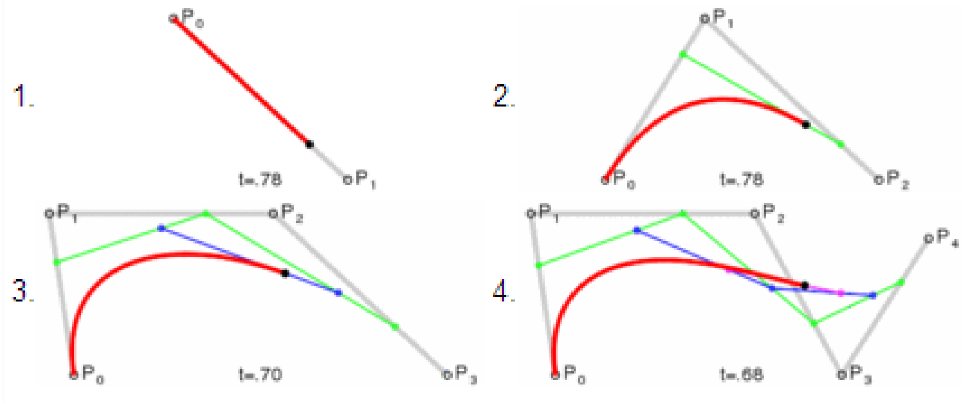

Figura 1: Exemplos de curvas de Bézier retirados da Wikipedia.

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:

$$
\begin{array}{l}
\mathit{linear1d} :: \mathbb{Q} \rightarrow \mathbb{Q} \rightarrow \mathit{OverTime}\ \mathbb{Q} \\
\mathit{linear1d}\ a\ b = \mathit{formula}\ a\ b\ \textbf{where} \\
\quad \mathit{formula} :: \mathbb{Q} \rightarrow \mathbb{Q} \rightarrow \mathit{Float} \rightarrow \mathbb{Q} \\
\quad \mathit{formula}\ x\ y\ t = ((1.0 :: \mathbb{Q}) - (\mathit{to}_{\mathbb{Q}}\ t)) * x + (\mathit{to}_{\mathbb{Q}}\ t) * y
\end{array}
$$

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.

$$\textbf{type}\ \mathit{NPoint} = [\mathbb{Q}]$$

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

$$
\begin{array}{l}
\mathit{p2d} = [1.2, 3.4] \\
\mathit{p3d} = [0.2, 10.3, 2.4]
\end{array}
$$

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

$$\textbf{type}\ \mathit{OverTime}\ a = \mathit{Float} \rightarrow a$$

O anexo C tem definida a função

$$\mathit{calcLine} :: \mathit{NPoint} \rightarrow (\mathit{NPoint} \rightarrow \mathit{OverTime}\ \mathit{NPoint})$$

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

$$\mathit{deCasteljau} :: [\mathit{NPoint}] \rightarrow \mathit{OverTime}\ \mathit{NPoint}$$

que implementa o algoritmo respectivo.

1. Implemente $\mathit{calcLine}$ como um catamorfismo de listas, testando a sua definição com a propriedade:

   Propriedade [QuickCheck] 8 *Definição alternativa.*

   $$
   \begin{array}{l}
   \mathit{prop\_calcLine\_def} :: \mathit{NPoint} \rightarrow \mathit{NPoint} \rightarrow \mathit{Float} \rightarrow \mathit{Bool} \\
   \mathit{prop\_calcLine\_def}\ p\ q\ d = \mathit{calcLine}\ p\ q\ d \equiv \mathit{zipWithM}\ \mathit{linear1d}\ p\ q\ d
   \end{array}
   $$

2. Implemente a função $\mathit{deCasteljau}$ como um hilomorfismo, testando agora a propriedade:

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

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

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,

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

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

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

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

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

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]

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

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:

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

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

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

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]:

$$oracle = [$$
$$1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, 208012, 742900, 2674440, 9694845,$$
$$35357670, 129644790, 477638700, 1767263190, 6564120420, 24466267020,$$
$$91482563640, 343059613650, 1289904147324, 4861946401452$$
$$]$$

Problema 3

Algoritmo:

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

Função auxiliar:

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

2D:

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

Modelo:

$$\mathbf{data}\ World = World\ \{points :: [NPoint]$$
$$, time :: Float$$
$$\}$$
$$initW :: World$$
$$initW = World\ [\,]\ 0$$

[10] Fonte: Wikipedia.

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

Gloss:

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

Animação:

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

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]

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

[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:

$$
\begin{array}{l}
\mathbf{instance}\ \mathit{Arbitrary}\ \mathit{UnOp}\ \mathbf{where}\\
\quad \mathit{arbitrary} = \mathit{elements}\ [\mathit{Negate}, E]\\
\mathbf{instance}\ \mathit{Arbitrary}\ \mathit{BinOp}\ \mathbf{where}\\
\quad \mathit{arbitrary} = \mathit{elements}\ [\mathit{Sum}, \mathit{Product}]\\
\mathbf{instance}\ (\mathit{Arbitrary}\ a) \Rightarrow \mathit{Arbitrary}\ (\mathit{ExpAr}\ a)\ \mathbf{where}\\
\quad \mathit{arbitrary} = \mathbf{do}\\
\qquad \mathit{binop} \leftarrow \mathit{arbitrary}\\
\qquad \mathit{unop} \leftarrow \mathit{arbitrary}\\
\qquad \mathit{exp1} \leftarrow \mathit{arbitrary}\\
\qquad \mathit{exp2} \leftarrow \mathit{arbitrary}\\
\qquad a \quad\ \leftarrow \mathit{arbitrary}\\
\qquad \mathit{frequency} \cdot \mathsf{map}\ (\mathit{id} \times \mathit{pure})\ \$\ [(20, X), (15, N\ a), (35, \mathit{Bin}\ \mathit{binop}\ \mathit{exp1}\ \mathit{exp2}), (30, \mathit{Un}\ \mathit{unop}\ \mathit{exp1})]
\end{array}
$$

$$
\begin{array}{l}
\mathbf{infixr}\ 5 \stackrel{?}{=}\\
(\stackrel{?}{=}) :: \mathit{Real}\ a \Rightarrow a \rightarrow a \rightarrow \mathit{Bool}\\
(\stackrel{?}{=})\ x\ y = (\mathit{to}_{\mathbb{Q}}\ x) \equiv (\mathit{to}_{\mathbb{Q}}\ y)
\end{array}
$$

Outras funções auxiliares

Lógicas:

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

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:

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

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

Para se descobrir a definição de $outExpAr$, e uma vez que já se sabe a definição de $inExpAr$, recorreu-se à propriedade dos isomorfismos $out \cdot \mathbf{in} = id$.

$$outExpAr.inExpAr = id$$

$\equiv$ { inExpAr = [(const X), num_ops] }

$$outExpAr \cdot [\underline{X}, num\_ops] = id$$

$\equiv$ { (20) }

$$[outExpAr \cdot \underline{X}, outExpAr \cdot num\_ops] = id$$

$\equiv$ { (17) }

$$\begin{cases} id \cdot i_1 = outExpAr \cdot \underline{X} \\ id \cdot i_2 = outExpAr \cdot num\_ops \end{cases}$$

$\equiv$ { (1), Esquerda: (71) }

$$\begin{cases} i_1\ x = (outExpAr \cdot \underline{X})\ x \\ i_2 = outExpAr \cdot num\_ops \end{cases}$$

$\equiv$ { Esquerda: (72) }

$$\begin{cases} i_1\ x = outExpAr\ (\underline{X}\ x) \\ i_2 = outExpAr \cdot num\_ops \end{cases}$$

$\equiv$ { Esquerda: (74) }

$$\begin{cases} i_1\ () = outExpAr\ X \\ i_2 = outExpAr \cdot num\_ops \end{cases}$$

$\equiv$ { Direita: num_ops = [N, ops] }

$$\begin{cases} i_1\ () = outExpAr\ X \\ i_2 = outExpAr \cdot [N, ops] \end{cases}$$

$\equiv$ { Direita: (20), (17) }

$$\begin{cases} i_1\ () = outExpAr\ X \\ \begin{cases} i_2 \cdot i_1 = outExpAr \cdot N \\ i_2 \cdot i_2 = outExpAr \cdot ops \end{cases} \end{cases}$$

$\equiv$ { Primeira Direita: (71) }

$$\begin{cases} i_1\ () = outExpAr\ X \\ \begin{cases} (i_2 \cdot i_1)\ x = (outExpAr \cdot N)\ x \\ i_2 \cdot i_2 = outExpAr \cdot ops \end{cases} \end{cases}$$

$\equiv$ { Pimeira Direita: (72) }

$$\begin{cases} i_1\ () = outExpAr\ X \\ \begin{cases} i_2\ (i_1\ x) = outExpAr\ (N\ x) \\ i_2 \cdot i_2 = outExpAr \cdot ops \end{cases} \end{cases}$$

$\equiv$ { Segunda Direita: ops = [bin, (uncurry Un)] }

$$\begin{cases} i_1\ () = outExpAr\ X \\ \begin{cases} i_2\ (i_1\ x) = outExpAr\ (N\ x) \\ i_2 \cdot i_2 = outExpAr \cdot [\,bin, \widehat{Un}\,] \end{cases} \end{cases}$$

$\equiv$ { Segunda Direita: (20), (17) }

$$\begin{cases} i_1\ () = outExpAr\ X \\ \begin{cases} i_2\ (i_1\ x) = outExpAr\ (N\ x) \\ \begin{cases} i_2 \cdot i_2 \cdot i_1 = outExpAr \cdot bin \\ i_2 \cdot i_2 \cdot i_2 = outExpAr \cdot \widehat{Un} \end{cases} \end{cases} \end{cases}$$

$\equiv$ { Segunda Direita: (71) }

$$\begin{cases} i_1\ () = outExpAr\ X \\ \begin{cases} i_2\ (i_1\ x) = outExpAr\ (N\ x) \\ \begin{cases} (i_2 \cdot i_2 \cdot i_1)\ (x, (y, z)) = (outExpAr \cdot bin)\ (x, (y, z)) \\ (i_2 \cdot i_2 \cdot i_2)\ (x, y) = (outExpAr \cdot \widehat{Un})\ (x, y) \end{cases} \end{cases} \end{cases}$$

$\equiv$ { Segunda Direita: (72) }

$$\begin{cases} i_1\ () = outExpAr\ X \\ \begin{cases} i_2\ (i_1\ x) = outExpAr\ (N\ x) \\ \begin{cases} i_2\ (i_2\ (i_1\ (x, (y, z)))) = outExpAr\ (Bin\ x\ y\ z) \\ i_2\ (i_2\ (i_2\ (x, y)) = outExpAr\ (Un\ x\ y)) \end{cases} \end{cases} \end{cases}$$

Assim, outExpAr pode ser definido como:

$$\begin{array}{l} outExpAr\ X = i_1\ () \\ outExpAr\ (N\ a) = i_2\ (i_1\ a) \\ outExpAr\ (Bin\ bp\ l\ r) = i_2\ (i_2\ (i_1\ (bp, (l, r)))) \\ outExpAr\ (Un\ up\ e) = i_2\ (i_2\ (i_2\ (up, e))) \end{array}$$

Uma vez descoberto o tipo de saida de $outExpAr$, torna-se bastante fácil definir a função $recExpAr$, visto que apenas nos interessa aplicar a recursividade aos tipos $ExpAr$, enquanto que nos restantes basta aplicar a função id, fazendo com que nunca se alterem.

$$recExpAr\ f = baseExpAr\ id\ id\ id\ f\ f\ id\ f$$

Realtivamente ao gene de $eval\_exp$, recorreu-se ao seguinte diagrama, de maneira a facilitar a compreensão dos tipos.

$$\begin{tikzcd} ExpAr\ a \arrow[r, bend left, "outExpAr"] \arrow[d, "eval\_exp"'] & 1 + (a + (BinOp \times (ExpAr\ a \times ExpAr\ a) + UnOp \times ExpAr\ a)) \arrow[l, bend left, "inExpAr"] \arrow[d, "id+(id+(id\times(eval\_exp\times eval\_exp)+id\times eval\_exp))"] \\ a & 1 + (a + (BinOp \times (a \times a) + UnOp \times a)) \arrow[l, "g\_eval\_exp"] \end{tikzcd}$$

Como se pode observar, o gene do catamorfismo corresponde à seta mais abaixo no diagrama. É fácil perceber que, quando o termo for vazio, basta devolver o valor escalar que nos é fornecido, e quando o termo já for um valor do tipo a, simplesmente se devolve o próprio elemento. Relativamente à parte recursiva, foram criadas duas funções auxiliares, sendo essas a $operationBin$ e a $operationUn$. A $operationBin$ trata os casos onde se recebe como primeiro elemento um par do tipo $BinOp$ e como segundo elemento um par de valores do tipo a. Caso $BinOp$ seja do tipo Sum, soma-se os dois valores, e caso seja do tipo $Product$, realiza-se a sua multiplicação. A $operationUn$ trata os casos onde se recebe um par com o primeiro elemento do tipo $UnOp$ e o segundo elemento um valor do tipo a. Aqui, caso $UnOp$ se trate de um $Negate$, devolve-se a negação do elemento, e caso se trate de E, devolve-se a exponenciação de a.

$$
\begin{aligned}
& g\_eval\_exp\ a = [\underline{a}, g\_val\_exp\_aux\ a] \\
& \quad \mathbf{where}\ g\_val\_exp\_aux\ a = [id, [operationBin, operationUn]]
\end{aligned}
$$

$$
\begin{aligned}
& operationBin\ (f, (a, b)) = \mathbf{if}\ f \equiv Sum\ \mathbf{then}\ a + b \\
& \quad \mathbf{else}\ a * b \\
& operationUn\ (f, a) = \mathbf{if}\ f \equiv Negate\ \mathbf{then} - a \\
& \quad \mathbf{else}\ expd\ a
\end{aligned}
$$

Para definir os genes da função $optmize\_eval$, foi necessário definir o seu diagrama, como um hilomorfismo, para perceber os tipos de entrada e saída de cada gene.

$$
\begin{tikzcd}
ExpAr\ a \arrow[r, bend left, "gene\ de\ clean"] \arrow[d, "clean"'] & 1 + (a + (BinOp \times (ExpAr\ a \times ExpAr\ a) + UnOp \times ExpAr\ a)) \arrow[d, "id+(id+(id\times(clean\times clean)+id\times clean))"] \\
ExpAr\ a \arrow[r, bend left, "outExp"] \arrow[d, "gopt"'] & 1 + (a + (BinOp \times (ExpAr\ a \times ExpAr\ a) + UnOp \times ExpAr\ a)) \arrow[l, bend left, "inExpAr"] \arrow[d, "id+(id+(id\times(gopt\times gopt)+id\times gopt))"] \\
a & 1 + (a + (BinOp \times (a \times a) + UnOp \times a)) \arrow[l, "gene\ de\ gopt"]
\end{tikzcd}
$$

Ao analisar o diagrama, percebe-se que estamos perante um caso particular de um anamorfismo, onde a estrutura que o anamorfismo devolve é do mesmo tipo de entrada. Assim, $clean$ pode ser visto como um tipo especial de $outExpAr$, só que desta vez, quando se estiver perante o elemento absorvente da multiplicação, que é 0, ao invés de retornar a expressão da multiplicação, pode simplesmente devolver-se $N\ 0$, diminuindo assim o trabalho do catamorfismo. Também se considerou que, o exponencial e elevado a 0 dará sempre 1. A função auxiliar $anyZero$ verifica se algum elemento da esquerda ou da direita é $N\ 0$, devolvendo $True$ caso tal se verifique.

$$
\begin{aligned}
& clean\ X = i_1\ () \\
& clean\ (N\ a) = i_2\ (i_1\ a) \\
& clean\ (Bin\ bp\ l\ r) = \mathbf{if}\ (bp \equiv Product \wedge (anyZero\ (l, r)))\ \mathbf{then}\ i_2\ (i_1\ 0) \\
& \quad \mathbf{else}\ i_2\ (i_2\ (i_1\ (bp, (l, r)))) \\
& clean\ (Un\ up\ e) = \mathbf{if}\ (up \equiv E \wedge (e \equiv (N\ 0)))\ \mathbf{then}\ i_2\ (i_1\ 1) \\
& \quad \mathbf{else}\ i_2\ (i_2\ (i_2\ (up, e))) \\
& anyZero\ (l, r) = \mathbf{if}\ (l \equiv (N\ 0) \vee r \equiv (N\ 0))\ \mathbf{then}\ True \\
& \quad \mathbf{else}\ False
\end{aligned}
$$

Relativamente ao gene de $gopt$, é possível perceber que este é semelhante ao gene da função g_eval_exp, pelo que se reutilizará as funções $operationBin$ e $operationUn$.

$$
\begin{aligned}
& gopt\ a = [\underline{a}, gopt\_aux\ a] \\
& \quad \mathbf{where}\ gopt\_aux\ a = [id, [operationBin, operationUn]]
\end{aligned}
$$

Analisando o código fornecido para a função sd, verifica-se que o resultado se encontra no segundo elemento do par devolvido pelo cataExpAr sd_gen.

$$
\begin{tikzcd}
ExpAr\ a \arrow[r, bend left, "outExpAr"] \arrow[d, "(\!|sd\_gen|\!)"'] & 1 + (a + (BinOp \times (ExpAr\ a \times ExpAr\ a) + UnOp \times ExpAr\ a)) \arrow[l, bend left, "inExpAr"] \arrow[d, "id+(id+(id\times((\!|sd\_gen|\!)\times(\!|sd\_gen|\!))+id\times(\!|sd\_gen|\!)))"] \\
ExpAr\ a & 1 + (a + (BinOp \times (ExpAr\ a \times ExpAr\ a) + UnOp \times ExpAr\ a)) \arrow[l, "{[g1, g2]}"]
\end{tikzcd}
$$

No caso de paragem apenas se necessita de devolver um par onde no primeiro elemento está o termo X (visto que quando se aplica $outExpAr$ a X este dá o único habitante do tipo 1), e no segundo elemento a derivada de X (que será sempre $N\ 1$).

Quando o termo é do tipo a apenas se necessita de devolver um par onde no primeiro elemento está o termo $N\ a$, e no segundo elemento a respetiva derivada $N\ 0$.

Também se pode estar na presença de um par do tipo $(BinOp, ((a, b), (c, d)))$, onde é importante notar que o primeiro par (a, b) corresponde à primeira $ExpAr$ e à sua derivada, e o par (c, d) corresponde à segunda $ExpAr$ e à sua derivada.

Por fim, pode-se estar na presença de um par do tipo $(UnOp, (a, b))$, onde mais uma vez, o termo a tem o valor da $ExpAr$ e o termo b tem a derivada desse valor. Assim, com base na explicação de derivações presentes no documento, é possível produzir duas funções axiliares, $decideBin$ e $decideUn$, que tratam de derivar corretamente as expressões.

$$
\begin{aligned}
&sd\_gen :: Floating\ a \Rightarrow \\
&\quad () + (a + ((BinOp, ((ExpAr\ a, ExpAr\ a), (ExpAr\ a, ExpAr\ a))) + (UnOp, (ExpAr\ a, ExpAr\ a)))) \\
&\quad \to (ExpAr\ a, ExpAr\ a) \\
&sd\_gen = [\lambda() \to (X, N\ 1), sd\_gen\_aux1] \\
&\quad \textbf{where}\ sd\_gen\_aux1 = [\lambda a \to (N\ a, N\ 0), sd\_gen\_aux2]
\end{aligned}
$$

$$
\begin{aligned}
&sd\_gen\_aux2 = [decideBin, decideUn] \\
&decideBin\ (f, ((a, b), (c, d))) = \textbf{if}\ f \equiv Sum\ \textbf{then}\ (Bin\ Sum\ a\ c, Bin\ Sum\ b\ d) \\
&\quad \textbf{else}\ (Bin\ Product\ a\ c, Bin\ Sum\ (Bin\ Product\ a\ d)\ (Bin\ Product\ b\ c)) \\
&decideUn\ (f, (a, b)) = \textbf{if}\ f \equiv Negate\ \textbf{then}\ (Un\ Negate\ a, Un\ Negate\ b) \\
&\quad \textbf{else}\ (Un\ E\ a, Bin\ Product\ (Un\ E\ a)\ b)
\end{aligned}
$$

Por fim, foi necessário definir ad_gen. Analisando a função ad, percebe-se que esta devolve um par, em que o resultado da derivada no ponto se encontra no segundo elemento do par devolvido pelo cata-ExpAr $ad\_gen\ v$. Conclui-se assim que ad_gen irá devolver um par, onde no primeiro elemento estará o resultado da substituição do ponto na expressão original, e no segundo elemento estará o resultado da derivada no ponto.

Quando o termo é do tipo $(BinOp, ((a, b), (c, d)))$, no termo a estará o resultado da substituição do ponto na primeira $ExpAr$, no termo b estará a derivada dessa $ExpAr$ no ponto, no termo c estará o resultado da substituição do ponto na segunda $ExpAr$ e no termo d estará a derivada dessa $ExpAr$ no ponto. Assim, na função auxiliar dB, conforme o tipo de $BinOp$, é possível descobrir todos os valores necessários.

Por último, quando o termo é do tipo $(UnOp, (a, b))$, no termo a estará o valor do ponto pretendido e no termo b estará a derivada da $ExpAr$ no ponto. Assim, na função auxiliar dU, se estivermos perante um $Negate$, devolve-se um par com a negação do ponto e a negação da sua derivada. Se estivermos perante um E, então devolve-se um par com a exponencial do ponto e com a multiplicação da exponencial no ponto pela derivada da $ExpAr$ no ponto.

$$
\begin{aligned}
&ad\_gen\ v = [ad\_aux1\ v, ad\_aux2] \\
&ad\_aux1\ v\ () = (v, 1) \\
&ad\_aux2 :: Floating\ a \Rightarrow (a + ((BinOp, ((a, a), (a, a))) + (UnOp, (a, a)))) \to (a, a) \\
&ad\_aux2 = [ad\_aux3, ad\_aux4] \\
&ad\_aux3\ a = (a, 0) \\
&ad\_aux4 :: Floating\ a \Rightarrow ((BinOp, ((a, a), (a, a))) + (UnOp, (a, a))) \to (a, a) \\
&ad\_aux4 = [dB, dU] \\
&dB\ (f, ((a, b), (c, d))) = \textbf{if}\ f \equiv Sum\ \textbf{then}\ (a + c, b + d) \\
&\quad \textbf{else}\ (a * c, a * d + b * c) \\
&dU\ (f, (a, b)) = \textbf{if}\ f \equiv Negate\ \textbf{then}\ (negate\ (a), negate\ (b)) \\
&\quad \textbf{else}\ (expd\ (a), expd\ (a) * b)
\end{aligned}
$$

Problema 2

Para se descobrir como simplificar o algoritmo de $Catalan$, de maneira a não usar fatoriais,foi necessário desenvolver a fórmula fornecida pelos docentes. De seguida, apresenta-se a redução que foi feita na fórmula de $Catalan$:

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

$$C_n = \frac{(2*n)(2n-1)!}{(n+1)n!*n(n-1)!} \tag{5}$$

$$C_n = \frac{2(2n-1)!}{(n+1)n!*(n-1)!} \tag{6}$$

$$C_n = \frac{2(2n-1)(2n-2)!}{(n+1)n!*(n-1)(n-2)!} \tag{7}$$

$$C_n = \frac{2(2n-1)}{(n+1)} * \frac{(2n-2)!}{n!(n-1)(n-2)!} \tag{8}$$

$$C_n = \frac{2(2n-1)}{(n+1)} * \frac{2(n-1)(2n-3)!}{n!(n-1)(n-2)!} \tag{9}$$

$$C_n = \frac{2(2n-1)}{(n+1)} * \frac{2(2n-3)!}{n(n-1)!(n-2)!} \tag{10}$$

$$C_n = \frac{2(2n-1)}{(n+1)} * \frac{2(2n-3)(2n-4)!}{n(n-1)!(n-2)!} \tag{11}$$

$$C_n = \frac{2(2n-1)}{(n+1)} * \frac{2(2n-3)}{n} * ... \tag{12}$$

Através do desenvolvimento da fórmula, é possível perceber que um determinado número de $Catalan$ poderá ser calculado à custa do número de $Catalan$ anterior. Assim, após simplificação da fórmula, chegou-se à seguinte equação:

$$C_0 = 1, C_n = \frac{2(2n-1)}{n+1} C_{n-1} \tag{13}$$

Após descoberta a fórmula, tornou-se bastante simples definir a função por recursividade mútua. A função $inic$ será inicializada com o par $(1,1)$, enquante que a função $loop$ será chamada recursivamente. Desta forma, a função $loop$ terá 2 elementos, onde o primeiro elemento terá o índice da iteração atual, e o segundo elemento terá o número de $Catalan$ da iteração anterior.

$$
\begin{aligned}
&loop\ (a,b) = (a+1, b*(2*((2*a)-1)) \div a+1) \\
&inic = (1,1) \\
&prj = \pi_2 \\
&cat = prj \cdot \mathsf{for}\ loop\ inic
\end{aligned}
$$

Como um ciclo for é definido através de um catamorfismo de naturais, a função cat também poderá ser especificada como um catamorfismo de naturais (denominada como catalan), como ilustrado no seguinte diagrama:

$$
\begin{tikzcd}
\mathbb{N}_0 \arrow[r, bend left, "out"] \arrow[d, "catalan"'] & 1+\mathbb{N}_0 \arrow[l, bend left, "\mathsf{in}"] \arrow[d, "id+catalan"] \\
\mathbb{N}_0 \times \mathbb{N}_0 & 1+\mathbb{N}_0 \times \mathbb{N}_0 \arrow[l, "{[\underline{(1,1)}, loop]}"]
\end{tikzcd}
$$

Após obter como resultado o par, onde o primeiro elemento é o sucessor e o segundo elemento é o $Catalan$ do natural introduzido, apenas nos interessa o segundo elemento elemento do par, de modo a obter o mesmo resultado que o obtido na função cat.

Problema 3

Com base no código da função *calcLine* fornecido, foi possível definir este à custa de um catamorfismo. Numa primeira fase, definiu-se o diagrama do catamorfismo.

$$\begin{tikzcd}
NPoint \arrow[d, "calcLine"'] \arrow[r, bend left, "outList"] & 1 + \mathbb{Q} \times NPoint \arrow[l, bend left, "inList"] \arrow[d, "id + id \times calcLine"] \\
OverTime\ NPoint^{NPoint} & 1 + \mathbb{Q} \times OverTime\ NPoint^{NPoint} \arrow[l, "h"]
\end{tikzcd}$$

Como *NPoint* é uma lista de $\mathbb{Q}$, então cada elemento de *NPoint* será do tipo $\mathbb{Q}$. No caso de paragem, é devolvida a lista vazia, tal como se pode verificar na função auxiliar $calcLine\_aux1$.

A função auxiliar $calcLine\_aux2$ recebe como parámetros um par e um *NPoint*. O primeiro elemento do par tem o elemento à cabeça do primeiro *NPoint* e o segundo elemento do par tem uma função que recebe um *NPoint* e devolve um *OverTime NPoint*. Se o *NPoint* for nulo, então o resultado será dado pela aplicação da função que recebe um *NPoint* e devolve um *OverTime NPoint*, aplicada à lista vazia. Caso contrário, basta dar como argumentos da função *linear1d* o primeiro elemento do par e primeiro elemento da lista, e depois aplicar a função, que se encontra no segundo elemento do par, à cauda do segundo *NPoint*.

$$\begin{aligned}
&calcLine :: NPoint \rightarrow (NPoint \rightarrow OverTime\ NPoint)\\
&calcLine = cataList\ h\ \mathbf{where}\\
&\quad h = [calcLine_aux1, calcLine_aux2]\\
&calcLine_aux1\ () = \underline{nil}\\
&calcLine_aux2\ (r, f)\ [\,] = f\ [\,]\\
&calcLine_aux2\ (r, f)\ (h : t) = \lambda z \rightarrow concat\ \$\ (sequenceA\ [singl \cdot linear1d\ r\ h, f\ t])\ z
\end{aligned}$$

De maneira a definir a função de *deCasteljau* como um hilomorfismo, foi necessário definir um novo tipo de dados. Inicialmente, pensou-se na possibilidade de definir esta função como um hilomorfismo de LTree, porém, após alguns testes, percebeu-se que o algoritmo não funcionava para listas vazias. Assim, desenvolveu-se o seguinte tipo de dados:

$$\mathbf{data}\ AlgForm\ a = Vazio \mid Elemento\ a \mid Raiz\ (AlgForm\ a, AlgForm\ a)\ \mathbf{deriving}\ (Show, Eq)$$

O tipo de dados *AlgForm* foi baseado no tipo LTree, acrescentando a possibilidade de um dado elemento desse tipo poder ser *Vazio*. Além do *Vazio*, *AlgForm* também poderá ser formada por apenas um *Elemento*, ou então uma *Raiz* formada por um par de *AlgForm*'s, assemelhando-se assim a uma árvore, tal como pretendido.

Uma vez definido o tipo de dados, foi necessário definir o **in** e o *out*, bem como o funtor deste tipo de dados, sendo que posteriormente definidos estes, se pode definir o *cataAlgForm*, *anaAlgForm* e *hiloAlgForm*.

$$\begin{aligned}
&inAlgForm = [\underline{Vazio}, [Elemento, Raiz]]\\
&outAlgForm\ Vazio = i_1\ ()\\
&outAlgForm\ (Elemento\ a) = i_2\ (i_1\ a)\\
&outAlgForm\ (Raiz\ (l, r)) = i_2\ (i_2\ (l, r))\\
&recAlgForm\ f = baseAlgForm\ id\ f\\
&baseAlgForm\ g\ f = id + (g + (f \times f))\\
&cataAlgForm\ a = a \cdot (recAlgForm\ (cataAlgForm\ a)) \cdot outAlgForm\\
&anaAlgForm\ f = inAlgForm \cdot (recAlgForm\ (anaAlgForm\ f)) \cdot f\\
&hyloAlgForm\ f\ g = cataAlgForm\ f \cdot anaAlgForm\ g
\end{aligned}$$

Uma vez definido tudo o que era necessário relativamente ao tipo de dados, foi necessário desenhar o diagrama do hilomorfismo associado ao $deCasteljau$, com base no novo tipo de dados.

$$
\begin{tikzcd}
NPoint* \arrow[r, bend left, "gene\ de\ coalg"] \arrow[d, "coalg"'] & 1 + NPoint + NPoint* \times NPoint* \arrow[d, "id+(id+(coalg \times coalg))"] \\
AlgForm\ NPoint* \arrow[r, bend left, "outAlgForm"] \arrow[d, "alg"'] & 1 + NPoint + AlgForm\ NPoint* \times AlgForm\ NPoint* \arrow[l, bend left, "inAlgForm"] \arrow[d, "id+(id+(alg \times alg))"] \\
OverTime\ NPoint & 1 + NPoint + OverTime\ NPoint \times OverTime\ NPoint \arrow[l, "gene\ de\ alg"]
\end{tikzcd}
$$

Como podemos reparar, se a lista for vazia, o gene do anamorfismo devolve o único habitante do tipo 1, que é o vazio. Se só tiver um elemento, o gene do anamorfismo devolve o próprio elemento, e, caso a lista tenha mais que um elemento, devolve um par constituido por duas listas, a primeira com todos os constituintes da lista excepto o último elemento, e a segunda com todos os constituintes da lista excepto o primeiro elemento. Relativamente ao gene do catamorfismo, este transformará o $AlgForm$ de lista de $NPoint$ num $OverTime\ NPoint$.

$$
\begin{array}{l}
deCasteljau :: [NPoint] \rightarrow OverTime\ NPoint \\
deCasteljau = hyloAlgForm\ alg\ coalg \\
coalg\ [\,] = i_1\ () \\
coalg\ [p] = i_2\ (i_1\ p) \\
coalg\ (l) = i_2\ (i_2\ (init\ l, tail\ l)) \\
alg = [algAux1, algAux2] \\
algAux1\ = \lambda() \rightarrow nil \\
algAux2 = [g1, g2] \\
g1\ \ a\ \ b = a \\
g2\ (l, r) = \lambda pt \rightarrow (calcLine\ (l\ pt)\ (r\ pt))\ pt
\end{array}
$$

Problema 4

Solução para listas não vazias:

A solução para listas não vazias dado por um catamorfismo de listas é representado pelo seguinte diagrama.

$$
\begin{tikzcd}
a* \arrow[r, bend left, "outList"] \arrow[d, "avg\_aux"'] & 1 + a \times a* \arrow[l, bend left, "inList"] \arrow[d, "id+id \times avg\_aux"] \\
a \times a & 1 + a \times (a \times a) \arrow[l, "{[avg\_aux1, avg\_aux2]}"]
\end{tikzcd}
$$

Assim, a função avg devolve o primeiro elemento do par devolvido pelo catamorfismo. Na função avg_aux considera-se que, no caso de paragem, é retornado o par (0,0), visto que no início, quer a média, quer o tamanho da lista são 0.

Quando a função avg_aux2 receber o par $(c, (a, x))$, o termo c corresponde ao primeiro elemento da lista, a corresponde à média da lista e x ao número de elementos da lista. Assim, para obter a nova média, basta somar o termo c com o somatório da lista (dado pela multiplicação da média pelo número de elementos), e dividir tudo pelo novo tamanho, que corresponde ao incremento de uma unidade no tamanho anterior.

$$
\begin{array}{l}
avg = \pi_1 \cdot avg\_aux \\
avg\_aux = cataList\ [avg\_aux1, avg\_aux2] \\
avg\_aux1\ \ a = (0, 0) \\
avg\_aux2\ (c, (a, x)) = ((c + (x * a))\ /\ (x + 1), x + 1)
\end{array}
$$

Solução para árvores de tipo LTree:

Definir a função avg para árvores do tipo LTree é bastante semelhante a definir a função avg, porém, torna-se mais fácil compreeender esta função após se definir o seu diagrama.

$$
\begin{tikzcd}
\mathsf{LTree}\ a \arrow[rr, bend left, "outLTree"] \arrow[d, "(\!|gene|\!)"'] & & a + \mathsf{LTree}\ a \times \mathsf{LTree}\ a \arrow[ll, bend left, "inLTree"] \arrow[d, "id + (\!|gene|\!) \times (\!|gene|\!)"] \\
a \times a & & a + ((a \times a) \times (a \times a)) \arrow[ll, "{[avgLTree\_aux1, avgLTree\_aux2]}"]
\end{tikzcd}
$$

A função $avgLTree$ retorna o primeiro elemento do par devolvido pelo catamorfismo. Podemos verificar que, após aplicada a recursividade, uma LTree a se transforma num par (a, a), onde o primeiro elemento corresponde ao somatório dos elementos da árvore e o segundo elemento corresponde ao número de elementos dessa mesma árvore. Assim, para saber a média final, basta somar a soma dos elementos da árvore da esquerda com a soma dos elementos da árvore da direita e dividir tudo pelo tamanho total das duas árvores.

$$
\begin{array}{l}
avgLTree = \pi_1 \cdot (\!| gene |\!) \ \mathbf{where} \\
\quad gene = [avgLTree\_aux1, avgLTree\_aux2] \\
avgLTree\_aux1\ \ a = (a, 1) \\
avgLTree\_aux2\ ((a1, a2), (b1, b2)) = ((a1 * a2 + b1 * b2) \,/\, (a2 + b2), a2 + b2)
\end{array}
$$

Problema 5

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

```
module BTree

open Cp

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

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

let inBTree x = either (konst Empty) (Node) x

let outBTree x =
     match x with
     | Empty -> i1 ()
     | Node (a,(t1,t2)) -> i2 (a,(t1,t2))

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

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

let recBTree g x = (baseBTree id g) x

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

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

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

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

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

// (4.0) Inversion (mirror) -----------------------------------------

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

// (4.1) Counting ---------------------------------------------------

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

// (4.2) Serialization ----------------------------------------------

let join (x,(l,r)) = l@[x]@r

let inord x = either nil join x

let inordt x = cataBTree inord x                //in-order traversal

let f1 (x,(l,r)) = x::l@r

let preord x = (either nil f1) x

let preordt x = cataBTree preord x              //pre-order traversal

let f2 (x,(l,r)) = l@r@[x]

let postordt x = cataBTree (either nil f2) x    //post-order traversal


//-- (4.4) Quicksort ------------------------------------------------
let rec part p list =
   match list with
   | head :: tail -> if p head then let (s,l) = part p tail in (head::s,l)
                     else let (s,l) = part p tail in (s,head::l)
   | [] -> ([],[])


let qsep list =
   match list with
   | head :: tail -> i2 (head, (part ((>) head) tail))
   | [] -> i1 ()


let qSort x = hyloBTree inord qsep x

//-- (4.5) Traces ---------------------------------------------------
let rec membro a list =
   match list with
   | [] -> false
   | x::xs -> if a=x then true else membro a xs

let rec union list1 list2 =
    match list2 with
    | [] -> list1
    | x::xs when membro  x list1 -> union list1 xs
    | x::xs -> (union list1 xs)@[x]
```

```
let tunion(a,(l,r)) = union (List.map (fun x -> a::x) l)
(List.map (fun x -> a::x) r)

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

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

let present x = inord x

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

let hanoi x = hyloBTree present strategy x

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

let hbaldepth (a, ((b1,b2),(d1,d2))) = (b1 && b2 && (abs(d1-d2) <= 1), 1
+ (max d1 d2))

let fbaldepth ((b1,d1),(b2,d2)) = ((b1,b2),(d1,d2))

let baldepth x = cataBTree (either (konst(true,1)) (hbaldepth <<
   (id >< fbaldepth)) ) x

let balBTree x = let (s,l) = baldepth x in s

let depthBTree x = let (s,l) = baldepth x in l

//---------------------------- end of library ----------------------------
```