DIUM – LESI/LMCC
Métodos de Programação I
Exercícios

Carlos Bacelar, Luís Barbosa,
José Barros, Alcino Cunha, Maria João Frade,
Luís Neves, José Nuno Oliveira, Jorge Sousa Pinto

Dezembro 2004

Capítulo 1

Cálculo Não-recursivo

I – Produtos e Coprodutos

1. Use a definição $f \times g = \langle f \cdot \pi_1, g \cdot \pi_2 \rangle$ para provar que a propriedade

$$(g \cdot h) \times (i \cdot j) = (g \times i) \cdot (h \times j)$$

se verifica.

2. Aplique a *lei da troca*, $[\langle f, g\rangle, \langle h, k\rangle] = \langle [f, h], [g, k]\rangle$, à definição

$$undistr = [id \times i_1, id \times i_2]$$

3. Considere as declarações de tipo

$$\begin{aligned} f &: A \to B, \\ g &: C \to D, \\ i &: X + Y \to X', \\ j &: X + Y \to Y', \\ i' &: X \to X' \times Y', \\ j' &: Y \to X' \times Y' \end{aligned}$$

(a) Identifique a assinatura das seguintes funções:
   i. $f + \langle i, j\rangle \times g$
   ii. $f + [i', j'] \times g$

(b) Serão as duas funções iguais? Justifique.

4. Sejam dadas as seguintes funções, no contexto da biblioteca `Mpi.hs`:

```
f = either (const True) (not . snd)
g = either k (succ . fst)
```

onde `k` é uma função arbitrária. Identifique, justificando,

- o tipo de `[f,g]`, isto é, da lista contendo as funções `f` e `g`;
- o tipo de `(f,g)`, isto é, do par de funções `f` e `g`.

5. Considere o seguinte raciocínio:

$$
\begin{array}{cl}
& k = [f,g] \iff \left\{ \begin{array}{l} k \cdot i_1 = f \\ k \cdot i_2 = g \end{array} \right. \\
\equiv & \qquad \{ \ \ldots \text{(justifique)} \ \ldots \} \\
& h \cdot [i,j] = [f,g] \iff \left\{ \begin{array}{l} (h \cdot [i,j]) \cdot i_1 = f \\ (h \cdot [i,j]) \cdot i_2 = g \end{array} \right. \\
\equiv & \qquad \{ \ \ldots \text{(justifique)} \ \ldots \} \\
& h \cdot [i,j] = [f,g] \iff \left\{ \begin{array}{l} h \cdot ([i,j] \cdot i_1) = f \\ h \cdot ([i,j] \cdot i_2) = g \end{array} \right. \\
\equiv & \qquad \{ \ \ldots \text{(justifique)} \ \ldots \} \\
& h \cdot [i,j] = [f,g] \iff \left\{ \begin{array}{l} h \cdot i = f \\ h \cdot j = g \end{array} \right. \\
\equiv & \qquad \{ \ \ldots \text{(justifique)} \ \ldots \} \\
& h \cdot [i,j] = [h \cdot i, h \cdot j]
\end{array}
$$

Como se chama a propriedade de que o raciocínio partiu? Justifique cada passo e indique qual das leis que constam do anexo foi deduzida.

6. Indique qual das leis que constam do anexo é justificada pelo raciocínio que se segue,

$$
\begin{array}{cl}
 & \langle i,j\rangle \cdot h = \langle f,g\rangle \iff \left\{ \begin{array}{l} \pi_1 \cdot (\langle i,j\rangle \cdot h) = f \\ \pi_2 \cdot (\langle i,j\rangle \cdot h) = g \end{array} \right. \\
\equiv & \qquad \{ \ \ldots \text{(justifique)} \ldots \} \\
 & \langle i,j\rangle \cdot h = \langle f,g\rangle \iff \left\{ \begin{array}{l} (\pi_1 \cdot \langle i,j\rangle) \cdot h = f \\ (\pi_2 \cdot \langle i,j\rangle) \cdot h = g \end{array} \right. \\
\equiv & \qquad \{ \ \ldots \text{(justifique)} \ldots \} \\
 & \langle i,j\rangle \cdot h = \langle f,g\rangle \iff \left\{ \begin{array}{l} i \cdot h = f \\ j \cdot h = g \end{array} \right. \\
\equiv & \qquad \{ \ \ldots \text{(justifique)} \ldots \} \\
 & \langle i,j\rangle \cdot h = \langle i \cdot h, j \cdot h\rangle
\end{array}
$$

Justifique cada passo.

7. Considere a seguinte definição de uma função t, em Haskell:

```
t f g h k = [either (split f g)(split h k),
             split (either f h)(either g k)]
```

Qual é o tipo de t? Justifique convenientemente a sua resposta.

8. (a) Preencha as reticências do diagrama funcional que se segue:

$$
\begin{tikzcd}
 & & \cdots \\
\cdots & \cdots \arrow[l, "\pi_1"'] \arrow[r, "\pi_2"] & \cdots \arrow[u, "\pi_1"'] \arrow[d, "\pi_2"] \\
\cdots \arrow[u, "\pi_1"] \arrow[uurr, bend left, "\pi_2"] & \cdots \arrow[l, "\pi_1"] \arrow[u, "f"] \arrow[ur, "g"'] \arrow[r, "\pi_2"'] & \cdots
\end{tikzcd}
$$

   (b) Exprima g e f como *splits* envolvendo outras funções no diagrama. Em seguida, escreva f em Haskell com variáveis, isto é, sem recorrer ao construtor *split* e às projecções π_1 e π_2.

   (c) Que função f sua conhecida é definida pelo diagrama? Qual é a sua inversa? Justifique.

9. Calcule (caso exista) o tipo da função $\langle i_1, \pi_1\rangle$

10. Dadas as definições

$$\begin{array}{rcl} f & = & (h \cdot \pi_2) \cdot swap \\ g & = & \pi_1 \cdot (h \times (id^A \cdot ap)) \end{array}$$

represente f e g sob a forma de diagramas evidenciando o seu tipo, e mostre que $f = g$, para todo o h.

11. Apresente, justificando todos os passos, uma prova equacional do facto

$$(id \times \pi_2) \cdot assocr = \pi_1 \times id$$

II – Isomorfismos

12. Demonstre a seguinte igualdade

$$[id \times i_1, id \times i_2] = \langle [\pi_1, \pi_1], \pi_2 + \pi_2 \rangle$$

Qual o isomorfismo que esta função estabelece?

13. Seja *distr* (ler: *distribute right*) a bijecção que estabelece o isomorfismo $A \times (B+C) \cong A \times B + A \times C$.

(a) Preencha as reticências no diagrama que se segue por forma a ver nele especificada a bijeccão *distl* (ler: *distribute left*) que estabelece o isomorfismo $(B+C) \times A \cong B \times A + C \times A$:

$$\begin{tikzcd}
(B+C)\times A \arrow[r, "swap"] \arrow[rrr, bend right, "distl"'] & \cdots \arrow[r, "distr"] & \cdots \arrow[r, "\cdots"] & B\times A + C\times A
\end{tikzcd}$$

(b) Mostre que

$$[g,h] \times f \quad = \quad [g \times f, h \times f] \cdot distl$$

é uma propriedade válida sobre *distl*, aplicando, entre outras leis que conhece, as seguintes:

$$\begin{aligned} f \times [g,h] &= [f \times g, f \times h] \cdot distr \\ swap \cdot (f \times g) &= (g \times f) \cdot swap \end{aligned}$$

14. Por inferência de tipos, escolha a função que, de entre as seguintes,

$$\begin{gathered} [id, id] \\ [\langle \underline{True}, id \rangle, \langle \underline{False}, id \rangle] \\ [\langle \underline{True}, \underline{False} \rangle, id] \\ id + id \end{gathered}$$

estabelece o isomorfismo

$$2 \times A \quad \cong \quad A + A$$

da direita para a esquerda.

Aplique-lhe a *lei da troca* e codifique o resultado em Haskell.

15. Por analogia com $swap = \langle \pi_2, \pi_1 \rangle$, mostre que $[i_2, i_1]$ é a sua própria inversa. Qual o isomorfismo que esta função estabelece?

16. Relembre
$$assocr : A \times (B \times C) \to (A \times B) \times C$$
e escreva a sua dual, isto é, a bijecção que estabelece o isomorfismo
$$A + (B + C) \quad \cong \quad (A + B) + C$$
da esquerda para a direita. Codifique essa função em Haskell.

17. (a) Identifique ou defina as funções f e g que testemunham o isomorfismo
$$A \times 1 \cong A$$
da esquerda para a direita e da direita para a esquerda, respectivamente (nota: assuma $1 \cong \{()\}$).
(b) Recorrendo à função *swap*, como definiria a função que estabelece o isomorfismo $1 \times A \cong A$ da esquerda para a direita?

18. Considerando os isomorfismos bem conhecidos
    - $A^2 \cong A \times A$
    - $2 \times A \cong A + A$
    - $A \times (B \times C) \cong (A \times B) \times C$
    - $A \times (B + C) \cong (A \times B) + (A \times C)$

    sintetize, justificando, o seguinte isomorfismo
$$v : A \times (1 + X)^2 \to A + A \times X + A \times X + A \times X^2$$

19. Determine, usando as funções $swap :: X \times Y \to Y \times X$ e $distr :: X \times (Y + Z) \to X \times Y + X \times Z$, um isomorfismo entre $(A \times B + C) \times D$ e $A \times (D \times B) + D \times C$. Apresente ainda uma codificação dessa função em Haskell.

20. Defina em Haskell um isomorfismo entre os tipos `Maybe a` e `Either () a`.

21. Os tipos `(a,Maybe b)` e `Either a (b,a)` são isomorfos. Use as funções `swap`, `distrl` e as do exercício anterior para definir (em notação *pointfree*) as funções que testemunham esse isomorfismo.

III – Condicional

22. Prove a *lei de fusão* do condicional de McCarthy:

$$f \cdot (p \rightarrow g, h) \quad = \quad p \rightarrow f \cdot g, f \cdot h$$

23. Partindo da definição do *combinador condicional* de McCarthy e da propriedade

$$p? \cdot f \quad = \quad (f + f) \cdot (p \cdot f)?$$

prove a validade de

$$(p \rightarrow f, g) \cdot h \quad = \quad (p \cdot h) \rightarrow (f \cdot h), (g \cdot h)$$

IV – Exponenciais

24. Identifique quais das igualdades seguintes,

```
curry f a b  =  f(a,b)
curry g (f a) c = curry (g .\(a,c)->(f a, c)) a c
```

(expressas em sintaxe Haskell) são propriedades válidas, e identifique-as, desenhando o diagrama correspondente.

25. Identifique as funções que estabelecem o isomorfismo que se segue

$$B^{C\times A} \cong (B^A)^C$$

e defina-as.

26. Mostre que se $f : A \to B$ é um isomorfismo então f^C tambem o é.

27. Use o resultado do exercício anterior para mostrar que $(A \times B)^C \cong (B \times A)^C$.

28. Demonstre as leis de *fusão* e *reflexão* da exponenciação, partindo da propriedade universal respectiva.

29. A função $distl : (A + B) \times C \to A \times C + B \times C$ (distributividade à esquerda) dispõe de uma definição *point-free* particularmente complicada:

$$distl = ap \cdot ([i_1^C \cdot \overline{id}, i_2^C \cdot \overline{id}] \times id)$$

(a) Preencha o diagrama seguinte que justifica o tipo atribuído à função.

$$\begin{tikzcd}
& & (A+B)\times C \arrow[dl, "\pi_1"'] \arrow[dr, "\pi_2"] & \\
\cdots \arrow[r, "\iota_1"] \arrow[d, "\overline{id}"'] & \cdots & \cdots \arrow[l, "\iota_2"'] \arrow[d, "\overline{id}"] & \cdots \arrow[d, "id"] \\
\cdots \arrow[r, "\iota_1^C"] & \cdots & \cdots \arrow[l, "\iota_2^C"'] & \cdots \\
& & \cdots \arrow[ul, "\pi_1"] \arrow[ur, "\pi_2"'] \arrow[d, "ap"] & \\
& & A\times C + B\times C &
\end{tikzcd}$$

(b) Conjecture a inversa *undistl* e apresente diagramas que justifiquem o seu tipo.

V – *Point-free* na definição de funções

30. O Haskell permite programar com funções constantes $\underline{c}$ – basta escrever `const c`. Verifique qual o tipo das expressões `f . const c` e `const (f c)`, para qualquer `f` e `c`. Que mais se pode dizer sobre estas expressões funcionais? Justifique.

31. Considere, em Haskell, a seguinte definição recursiva da função factorial,

```
fac 0 = 1
fac (n+1) = (n+1) * fac n
```

Mostre que essa definição pode ser convertida na seguinte definição *pointfree*:

$$fac \cdot [\underline{0}, succ] = [\underline{1}, mul \cdot (succ \times id)] \cdot (id + \langle id, fac \rangle)$$

onde *mul* é um operador *uncurried* de multiplicação (em Haskell, designa a versão *uncurried* de $*$ na class `Num`).

32. Considere a seguinte função em Haskell que calcula o quadrado de um número:

```
sq 0 = 0
sq (n+1) = 2*n+1 + sq n
```

Mostre que *sq* satisfaz a equação

$$sq \cdot in \quad = \quad [\underline{0}, add \cdot \langle odd, sq \rangle]$$

onde $in = [\underline{0}, succ]$, $\overline{add} = (+)$ e $odd = suc \cdot add \cdot \langle id, id \rangle$.
Sugestão: Recupere o código Haskell acima a partir da conversão da equação dada para notação com variáveis.

33. Derive a versão *pointwise* em Haskell da função f caracterizada pela seguinte equação

$$f \cdot [\underline{0}, \text{succ}] = [\langle \underline{0}, \underline{1} \rangle, \langle \pi_2, \text{uncurry } (+) \rangle] \cdot (id + f)$$

34. Considere a função

```
obsNat 0 = Left ()
obsNat (n+1) = Right (n+1,n)
```

Mostre que

$$obsNat \quad = \quad (id + \langle succ, id \rangle) \cdot out_{\mathbf{N}_0}$$

se verifica, onde $out_{\mathbf{N}_0}$ é o isomorfismo inverso de $in_{\mathbf{N}_0} = [\underline{0}, succ]$ em $\mathbf{N}_0 \cong 1 + \mathbf{N}_0$.
Sugestão: componha ambos os membros da igualdade com $in_{\mathbf{N}_0}$.

35. Considere a seguinte definição *point-free* de uma função f:

$$f = [\mathsf{succ} \cdot \underline{0}, \mathsf{plus} \cdot \langle f \cdot \mathsf{pred}, f \cdot \mathsf{pred2} \rangle] \cdot \mathsf{zeroOrOne}?$$

em que

$$\begin{array}{l} \mathsf{zeroOrOne}\ x = (x == 0)\ ||\ (x == 1) \\ \mathsf{plus} = \mathsf{uncurry}\ (+) \\ \mathsf{pred2} = \mathsf{pred} \cdot \mathsf{pred} \end{array}$$

Escreva uma definição de f no estilo *pointwise*, justificando todos os passos para a sua obtenção.

36. A seguinte função codifica um algoritmo de ordenação clássico, normalmente conhecido pelo nome de *heapsort*.

$$\mathsf{hsort} = [\mathsf{id}, \mathsf{cons} \cdot (\mathsf{id} \times (\mathsf{merge} \cdot (\mathsf{hsort} \times \mathsf{hsort}))) \cdot \mathsf{aux} \cdot \mathsf{cons}] \cdot \mathsf{out}_{\mathsf{RList}}$$

em que merge é a usual função de fusão de listas ordenadas; e aux é definida como se segue:

```
aux [h]   = (h,([],[]))
aux (h:t) = let (y,(l,r)) = aux t
            in if h<y then (h,(y:r,l))
                      else (y,(h:r,l))
```

Escreva uma definição de hsort no estilo *pointwise*, justificando todos os passos para a sua obtenção.

Capítulo 2

Cálculo Recursivo

I – Catamorfismos

1. Para descrever documentos Html usou-se o seguinte tipo:

   ```
   data DocHtml = S String | T String DocHtml | L [DocHtml]
   ```

   Por exemplo, o texto Html

   ```
   <html>
   <head>
   <title>Hello</title>
   </head>
   <body> Hello Word </body>
   </html>
   ```

   será representado pelo termo

   ```
   hello = T "html" (L [T "head" (L [T "title" (S "Hello") ]),
                        T "body" (L [S "Hello Word"]) ])
   ```

   (a) Desenhe o diagrama de definição de catamorfismos sobre `DocHtml`.

   (b) Defina, como um catamorfismo sobre `DocHtml`, a função

   ```
   imprime :: DocHtml -> String
   ```

2. Considere a função que se segue:

   ```
   f []    = ([],[])
   f (h:t) = let (l,r) = f t in (h:r,l)
   ```

Determine a sua assinatura, diga por palavras suas o que calcula a função, e exprima-a como um catamorfismo.

3. A seguinte versão linear do algoritmo de Fibonacci,

```
fib n = snd (f n)

f 0     = (0,1)
f (n+1) = let (a,b) = f n
          in (b,a+b)
```

é uma codificação em Haskell cuja função auxiliar f resultou do diagrama que se segue:

$$\begin{tikzcd} \mathbf{N} \arrow[d, "f"'] & 1+\mathbf{N} \arrow[l, "{[\underline{0},succ]}"'] \arrow[d, "id+f"] \\ \mathbf{N}\times\mathbf{N} & 1+\mathbf{N}\times\mathbf{N} \arrow[l, "g"] \end{tikzcd}$$

Caracterize o *gene* g do catamorfismo em causa.

4. Para representar expressões aritméticas com variáveis, pode usar-se o seguinte tipo:

```
data ExpVar = Var String | Const Int | Op(BinOp,ExpVar,ExpVar)
data BinOp = Mais | Menos | Vezes | Div
```

(a) Relembre a definição

```
data FTree a c = Unit c | Comp (a,(FTree a c,FTree a c))
```

e determine tipos `A` e `B` tais que `FTree A B` seja isomorfo a `ExpVar`. Defina em Haskell os isomorfismos entre esses dois tipos de dados.

(b) Defina como um catamorfismo a função `vars :: ExpVar -> [String]` que calcula a lista das variáveis que ocorrem numa expressão. Qual o valor de `vars (Op(Mais,Var "x",Var "x"))`?

(c) A função `subst :: String -> Exp -> Exp -> Exp` tem como objectivo substituir todas as ocorrências de uma variável de uma expressão por uma outra expressão. Por exemplo

```
subst "x"(Op(Mais,Const 3,Const 4)) (Op(Menos,Var "x",Var "y"))
```

deve dar como resultado a expressão

```
Op(Menos,Op(Mais,Const 3,Const 4),Var "y")
```

isto é (na notação habitual): substituindo por $3 + 4$ a variável x que ocorre em $x - y$, obtém-se a expressão $(3 + 4) - y$.

Defina como um catamorfismo sobre `ExpVar` a função `subst v e`.

5. No decorrer deste curso foi explicitada a relação entre a função `foldr` existente no prelúdio do HASKELL e os *catamorfismos* das listas. Em analogia com a função `foldr` das listas, considere a seguinte assinatura de função:

```
foldrX :: (String -> a -> a -> a) -> (Int -> a) -> X -> a
```

Conjecture um tipo de dados `X` para o qual a função `foldrX` possa ser entendida de forma análoga a `foldr` nas listas.

(a) Defina a função `foldrX` para o tipo de dados por si escolhido.

(b) Considere as funções seguintes,

```
f = foldrX (\x y z -> 1+y+z) (\x -> 0)
g = foldrX (\x y z -> y ++ x ++ z) (\x -> (show x))
```

Qual o tipo de cada uma das funções apresentadas? Diga resumidamente o que faz e para que serve cada uma. Dê exemplos de valores `x` e `y` de tal forma que `(f x)=2` e `(g y)="5 H 7"`.

(c) Diga como poderia utilizar o tipo de dados `X` por si definido para representar funções aritméticas simples. Como representaria a expressão `5+(3*2)`?

(d) Utilize agora a função `foldrX` para realizar a função de cálculo de expressões.

6. Uma das primeiras linguagens de programação funcional foi o Lisp, que apareceu há cerca de 40 anos. Nesta linguagem existe um único suporte para representação de dados, designado por *expressão-S* — abreviatura de *expressão simbólica*. Uma *expressão-S* é ou um valor atómico, ou uma sequência (possivelmente vazia) de *expressões-S*. Considera-se um *átomo* toda a unidade de informação indivisível, não-estruturada (i.e., *atómica*).

Por exemplo, são átomos os inteiros e os *strings* alfanuméricos, por exemplo `10`, $-$`5`, `a12`, `xyz`. Dão-se a seguir exemplos de *expressões-S* não atómicas, escritas na própria sintaxe concreta do Lisp:

```
()
(1)
(1 um 2 dois)
(1 (2 (3 (4))))
```

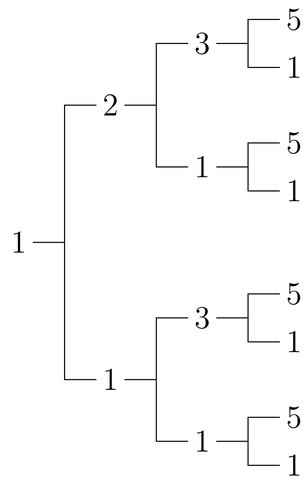
Seja

```
data SExp a = Atom a | Exp [ SExp a ]
```

a declaração de um tipo de dados em Haskell para descrever *expressões-S*.

Desenhe o diagrama dos catamorfismos deste tipo e exprima a operação que conta o número de átomos presentes numa *expressão-S* como um desses catamorfismos.

7. Para encontrar todos os divisores de um dado número, é vulgar recorrer-se a uma árvore n-ária construída com base nos factores primos desse número, procedendo-se depois à multiplicação dos elementos que fazem parte dos diferentes *caminhos* da árvore. Por exemplo, ao número 30 corresponde a árvore seguinte:



A multiplicação dos elementos dos seus vários *caminhos* dá como resultado todos os divisores de 30, i.e.

$$1, 5, 3, 15, 2, 10, 6, 30$$

Considere agora o seguinte tipo indutivo que define a estrutura de uma árvore tal n-ária não vazia

```
data XTree a = XLeaf a | XCons (a , [XTree a])
```

a cujos catamorfismos corresponde o diagrama que se segue:

$$\begin{tikzcd}
\mathsf{XTree}\, A \arrow[d, "(\!|g|\!)"'] & A + A \times [\ \mathsf{XTree}\, A\ ] \arrow[l, "in"'] \arrow[d, "id + id \times map\ (\!|g|\!)"] \\
[\ [\ A\ ]\ ] & A + A \times [\ [\ [\ A\ ]\ ]\ ] \arrow[l, "g"]
\end{tikzcd}$$

(a) Complete a seguinte definição de uma função que deverá calcular todos os caminhos de uma árvore $\mathsf{XTree}\, A$:

```
traces = cataXTree (either ... ...)
```

(b) Suponha que alguém já programou uma função $f : \mathbf{N} \to \mathsf{XTree}\, \mathbf{N}$ que, dado um número $n \in \mathbf{N}$, constroi a respectiva árvore de primos. Defina então uma função $g : \mathbf{N} \to [\ \mathbf{N}\ ]$ que lhe permita obter a lista dos divisores de um dado número $n \in \mathbf{N}$.

8. Nesta disciplina estudou-se um método de programação que estende a tipos indutivos polinomiais algumas construções bem conhecidas, como por exemplo, `map` e `fold`. Contudo, em lugar de `fold`s falou-se de `cata`s. Isto porque de um `cata` se obtém facilmente o respectivo `fold` *desdobrando* o seu *gene* nos seus componentes, explicitando constantes e fazendo o *currying* dos operadores com mais de um argumento — por exemplo:

```
foldRList u f = cataRList (either (const u) (uncurry f))
```

Defina `foldLTree` e `foldBTree`.

9. Caracterize a função que é definida por $(\!|[\underline{[]}, h]|\!)$ para cada uma das seguintes definições de h:

$$
\begin{array}{rcl}
h(x,(y_1,y_2)) & = & y_1 +\!\!+ [x] +\!\!+ y_2 \\
h & = & app \cdot\ (singl \times app) \\
h & = & app \cdot\ (app \times singl) \cdot swap
\end{array}
$$

assumindo

$$
\begin{array}{l}
singl\ a = [a] \\
app\ \ = uncurry\ (+\!\!+\ )
\end{array}
$$

Qual é o tipo de dados em jogo? Justifique.

10. No contexto do exercício 33 do Cap. 1, defina f como um catamorfismo no tipo conveniente. Apresente os diagramas necessários.

11. Antes de resolver as duas alíneas desta questão analize com atenção a seguinte arquitectura para a função

$$mdc \quad = \quad mul \cdot fpc \cdot (fp \times fp)$$

que calcula o máximo divisor comum entre dois números naturais, conforme o diagrama

$$\mathbf{N} \times \mathbf{N} \xrightarrow{fp \times fp} \mathbf{N}^* \times \mathbf{N}^* \xrightarrow{fpc} \mathbf{N}^* \xrightarrow{mul} \mathbf{N}$$

onde

- fp calcula os factores primos de um número listados por ordem crescente, por exemplo $fp\ 60 = [2, 2, 3, 5]$ e $fp\ 42 = [2, 3, 7]$;
- fpc intersecta duas listas de factores primos (fpc abrevia *factores primos comuns*), por exemplo $fpc\ ([2, 2, 3, 5], [2, 3, 7]) = [2, 3]$;
- mul multiplica os factores da lista produzida por fpc, inferindo assim o máximo divisor comum – $mdc\ (60, 42) = 2 * 3 = 6$.

(a) Complete a seguinte definição, em Haskell, da função

```
fpc' [] r = .........
fpc' l [] = .........
fpc' (a:l) (b:r) | a == b = .........
                 | a <  b = .........
                 | a >  b = .........
```

que é a versão *curried* de fpc (isto é, $fpc' = \overline{fpc}$):

(b) Escreva mul como um catamorfismo (de listas).

12. A seguinte versão linear do algoritmo de Fibonacci,

```
fib n = snd (f n)

f 0     = (0,1)
f (n+1) = let (a,b) = f n
          in (b,a+b)
```

é uma codificação em Haskell cuja função auxiliar f é o catamorfismo de naturais representado no diagrama que se segue:

$$\begin{tikzcd}
\mathbf{N} \arrow[d, "f"'] & 1 + \mathbf{N} \arrow[l, "{[\underline{0}, succ]}"'] \arrow[d, "id+f"] \\
\mathbf{N} \times \mathbf{N} & 1 + \mathbf{N} \times \mathbf{N} \arrow[l, "g"]
\end{tikzcd}$$

Caracterize o *gene* g do catamorfismo em causa.

13. A seguinte função em Haskell, extraída de `Prelude.hs`,

```
lookup :: Eq a => a -> [(a,b)] -> Maybe b
lookup k []       = Nothing
lookup k ((x,y):xys)
      | k==x      = Just y
      | otherwise = lookup k xys
```

procura um dado elemento k numa lista de pares (x, y) e, se o encontrar, devolve o y correspondente.

(a) Escreva *lookup* k sob a forma de um catamorfismo de listas, isto é, calcule o gene g em

$$\mathit{lookup}\ k = (\!|g|\!)$$

Qual o seu comportamento para o caso da chave k estar repetida na lista argumento?

(b) Defina a função equivalente a *lookup* k para árvores binárias (em `BTree.hs`). Qual o seu comportamento para o caso da chave k vir repetida na árvore argumento?

14. Recorde a biblioteca `LTree.hs`, a que se acrescenta a função

```
f = cataLTree (inLTree . (id -|- swap))
```

Que "faz" esta função? Converta-a para notação Haskell *com* variáveis.

15. Considere a função definida como

$$\begin{array}{ll} \mathsf{media} & = (\mathsf{uncurry}(/)) \cdot \mathsf{aux} \\ \mathsf{where} & \mathsf{aux} = \langle (\!|[\underline{0}, \mathsf{uncurry}\ (+)]|\!)\ , \mathsf{length} \rangle \end{array}$$

(a) Defina aux como um catamorfismo.

(b) Poderá a função media ser definida como um catamorfismo? Justifique a sua resposta.

16. Considere a seguinte definição duma função que calcula a média de todos os inteiros que são folhas de uma árvore binária:

```
media :: LTree Integer -> Integer
media = (uncurry div) . calc
```

Defina `calc` como um catamorfismo.

17. Considere a seguinte função para calcular as posições de um elemento numa lista:

```
posicoes :: (Eq a) => a -> [a] -> [Int]
posicoes a l = [y | (x,y) <- zip l [1..], x == a]
```

   (a) Apresente uma versão recursiva desta função.
   (b) Reescreva a definição da função `posicoes x` definida na alínea anterior como um catamorfismo. Apresente os diagramas correspondentes.
   (c) Modifique o gene do catamorfismo da alínea anterior de forma a obter uma função que calcula quantas vezes um determinado elemento ocorre numa lista.

18. Considere a seguinte declaração de tipo em Haskell

```
data DT a = Vazia | Nodo (a, Bool -> DT a)
```

   (a) Demonstre que o tipo `DT a` é isomorfo ao tipo `BTree a`.
   (b) Defina em Haskell, como catamorfismos de `DT a` e `BTree a`, as funções testemunhas do isomorfismo `DT a` $\cong$ `BTree a`.
   (c) Considere a função

```
resp :: DT a -> [Bool] -> Maybe a
resp Vazia _ = Nothing
resp (Nodo (x,_)) [] = Just x
resp (Nodo (x,f)) (b:bs) = case (f b) of
                             Vazia -> Nothing
                             _     -> resp (f b) bs
```

   Explique por palavras suas porque é que, para um dado `t` do tipo `DT a`, a função (`resp t`) não pode ser expressa como um catamorfismo sobre listas.

19. Considere o tipo de dados

```
data  BTree a = Empty | Node (a,(BTree a, BTree a))
```

A função `alt :: BTree a -> Int` define-se como um catamorfimo de `BTree` da seguinte forma

$$\texttt{alt} = (\![\, [\underline{0}\,, succ\,.\,max\,.\,\pi_2]\,]\!)$$

sendo max a versão "uncurried" da função que calcula o máximo de dois inteiros.

Com base nas leis de catamorfismos traduza a função `alt` para a notação com variáveis.

20. Considere a seguinte função que testa se uma árvore binária está equilibrada

```
equi Empty = True
equi (Node (x,(e,d))) = equi e  &&  equi d  &&
                             abs (alt e - alt d) <= 1
```

A função `equi` poderá ser definida como um catamorfismo de `BTree` ? Justifique a sua resposta.

21. A lei seguinte

$$\langle (\!|i|\!)_{\mathsf{F}} , (\!|j|\!)_{\mathsf{F}} \rangle = (\!|(i \times j) \cdot \langle \mathsf{F}\ \pi_1, \mathsf{F}\ \pi_2 \rangle |\!)_{\mathsf{F}}$$

que é popularmente conhecida pelo nome de *banana-split*, permite combinar dois catamorfismos num só.

(a) Identique os tipos genéricos de entrada e saída da função

$$f \quad = \quad \langle (\!|[g1, g2]|\!) , (\!|[g1, g3]|\!) \rangle$$

onde

```
g1 = const []
g2(Left  a, l) = l
g2(Right b, l) = b:l
g3(Left  a, l) = a:l
g3(Right b, l) = l
```

Faça um diagrama explicativo e descreva (sucintamente) o que a função "faz" através de um exemplo.

(b) Aplique a lei *banana-split* à função f e exprima o resultado do seu cálculo em Haskell com variáveis. Qual é a vantagem desta versão da função em relação à original?

22. Defina como um catamorfismo sobre listas não vazias, uma função que agrupa elementos consecutivos iguais; note que, quando aplicada à lista `"aaabccdddd"`, por exemplo, esta função deve retornar `[('a',3),('b',1),('c',2),('d',4)]`.

II – Anamorfismos

23. Escreva sob a forma de um anamorfismo de listas a função que calcula a sequência de todos os inteiros pares não negativos inferiores a um dado número. Codifique o resultado em Haskell.

24. Apresente as funções testemunhas do isomorfismo `DT a` $\cong$ `BTree a` do Ex. 18, agora definidas como anamorfismos.

25. Considere a função

$$g : [a] \times [b] \rightarrow 1 + (a \times b) \times (([a] \times [b]) \times ([a] \times [b]))$$

definida em Haskell como se segue (Nota: $null = (== [\,])$):

```
g = ((const ()) -|- (split g1 g2)) . (grd (null . p2))
    where g1 = (head >< head)
          g2 = (split (tail >< tail) (tail >< tail))
```

(a) Desenhe o diagrama do anamorfismo do qual g é gene. Qual é o propósito desta função?

(b) Calcule a função $[\!(g)\!]$ em notação com variáveis e exprima-a em Haskell.

26. A seguinte função em Haskell testa se uma dada lista (não vazia) está ou não ordenada.

```
ordenada :: (Ord a) => [a] -> Bool
ordenada = and . (map (uncurry (<=))) . (uncurry zip) . (split id tail)
```

(a) Mostre que esta função pode ser expressa como o seguinte anamorfismo de listas.

$$ordenada \quad = \quad and \cdot [\!(g \cdot \langle id, tail \rangle)\!]$$

Para isso defina a função g.

(b) Obtenha uma definição equivalente e recursiva de `ordenada`.

27. Considere a seguinte função `soma`:

```
soma ([],x) = x
soma (x,[]) = x
soma ((a:as),(b:bs)) = (a+b):(soma (as,bs))
```

Qual o tipo da função ? Exprima-a como um anamorfismo.

28. Considere as seguintes definições

```
data LTree a = Leaf a | Fork (LTree a) (LTree a)

unfold :: (a -> Maybe (b,a)) -> a ->  [b]
unfold f x = case (f x) of
             Nothing    -> []
             Just (h,y) -> h : (unfold f y)
```

(a) Defina a função `unfold` como um anamorfismo de listas, i.e., complete a seguinte definição

```
unfold f = anaList (g f)
     where g  ... = ...
```

(b) Por analogia com a função da primeira alínea (relembre também o Ex. 20 do Cap. 1), defina `unfoldLTree`

```
unfoldLTree :: ...
unfoldLTree f = anaLTree (h f)
     where h ... = ...
```

29. Escreva $[\!(\,(id + \langle odd, id\rangle) \cdot out_{\mathsf{F}}\,)\!]_{\mathsf{G}}$ em Haskell com variáveis e descreva o resultado da aplicação deste anamorfismo ao argumento $n = 3$, sendo odd a função

$$odd = suc \cdot add \cdot \langle id, id\rangle$$

e F, G os functores-base dos naturais e listas de naturais, respectivamente.

30. A função `cref :: (Eq a) => [a] -> [(a,Int)]` calcula, para cada elemento de uma lista, o número de vezes que ele ocorre nessa lista. Por exemplo,

```
cref [1,2,3,1,3,1] = [(1,3),(2,1),(3,2)]
```

(a) Defina a função `cref` como um anamorfismo.

(b) Considere agora uma variante desta função que, em vez de calcular o número de ocorrências, calcula as posições onde os elementos ocorrem. Explique por palavras suas por que é que esta função não pode ser descrita por uma modificação do gene da função da alínea anterior.

31. Considere a função seguinte:

```
altern :: (Int, Bool) -> [Char]
altern (0, _) = []
altern (n, b) | b = '+' : altern (n-1, False)
              | otherwise = '-' : altern (n-1, True)
```

Apresente a definição de `altern` como anamorfismo do tipo `[Char]`.

32. Considere a seguinte definição:

```
minDiv :: Integral a => a -> a
minDiv n = head [x | x <- [2..n] , n `mod` x == 0]
```

que calcula o mínimo divisor de um número natural (maior do que 1).

Use esta função na definição de um anamorfismo `factPrimos` que calcula a lista (não vazia) dos factores primos de um número inteiro, maior do que 1. (Note que o tipo de dados do resultado é "listas não vazias". Apresente o diagrama dos anamorfismos para este tipo).

III – Outros Exercícios sobre Anas e Catas

33. Relembre as definições de árvores binárias e listas:

```
data Lista a = Nil | Cons (a, Lista a)
data ArvBin a = Empty | Bin (a,(ArvBin a , ArvBin a))
```

Defina a função `posorder :: ArvBin a -> Lista a` como:

(a) um catamorfismo sobre árvores binárias

(b) um anamorfismo sobre listas

34. Considere a seguinte declaração de tipo em Haskell

```
data Nat = Zero | Succ Nat
```

(a) Defina as operações `cata` e `ana` para esse tipo de dados. Acompanhe essas definições com os diagramas respectivos.

(b) O tipo apresentado é isomorfo ao tipo primitivo do Haskell `[()]`. Apresente a definição em Haskell das funções que testemunham esse isomorfismo.

(c) Defina a função `comp :: [a] -> Nat` (determina o comprimento de uma lista) como um anamorfismo do tipo `Nat`.

35. Relembre as definições seguintes:

```
data BTree a = Empty | Node(a, (BTree a, BTree a))
data LTree a = Leaf a | Split (LTree a, LTree a)
```

Enquanto que na primeira há informação apenas em nodos intermédios (nas folhas encontram-se apenas `Empty`'s), na segunda a informação encontra-se só nas folhas. Considere agora uma outra variante de árvores binárias — *shape trees* — em que se guarda apenas a forma de uma árvore. O tipo `STree` das *shape trees* pode ser definido como:

```
data STree = V | N STree STree
```

(a) Defina em Haskell as usuais funções `in`, `out`, `rec`, `ana`, e `cata` para o tipo `STree`. Desenhe os correspondentes diagramas.

(b) Defina como anamorfismos em `STree` as funções

   i. `shapeB :: BTree a -> STree`
   ii. `shapeL :: LTree a -> STree`

que devolvem a forma das árvores em causa. Por exemplo, a forma da `BTree`

```
Node (3,(Empty,Node (4,(Node (5,(Empty,Empty)),Empty))))
```

e da `LTree`

```
Split (Leaf 'a',Split (Split (Leaf 'b',Leaf 'c'),Leaf 'd'))
```

é a `STree`

```
N V (N (N V V) V)
```

(c) Defina como um catamorfismo de `BTree`s a função

```
separa :: BTree a -> (STree, [a])
```

que, dada uma árvore binária de procura, devolve um par `(t,c)` em que `t` é a forma da árvore e `c` é a lista ordenada dos nodos da árvore.

(d) Considere a seguinte definição (`cataFTree` gera os catamorfismos do tipo *Full Tree*):

```
f = cataFTree (either v n)
      where v x = (Empty, Leaf x)
            n (r,((a,b),(x,y))) = (Node (r,(a,x)), Split (b,y))
```

Qual o tipo de `f`? Qual o resultado de `f (Comp (3,(Comp (2,(Unit 'a',Unit 'b')),Unit 'c')))`? O que calcula a função f?

36. Considere o tipo de dados indutivo das *listas não vazias*:

```
data NRList a = Sing a | Add  (a , NRList a)
```

(a) Comece por definir `inNRList`, `outNRList`, `recNRList`, `cataNRList` e `anaNRList`.

(b) Dada a função `f = (const ()) -|- id`, qual é o tipo de `(inRList . f)`, onde `inRList` é uma função que conhece do módulo `RList.hs`? Desenhe os diagramas justificativos da sua resposta.

(c) Desenhe o diagrama do catamorfismo `cataNRList (inRList . f)` e calcule a função recursiva definida por este catamorfismo ao nível da variável, explicando o que a função obtida faz.

(d) Defina como catamorfismos as funções `maxim` e `minim` que calculam, respectivamente, o maior e o menor elemento de uma lista não vazia.

(e) Defina como um catamorfismo a função `maxmin :: NRList a -> (a,a)` que calcula o maior e o menor elementos de uma lista não vazia.

(f) Defina como um anamorfismo de `NRList` a função

```
inits :: [a] -> NRList [a]
```

que calcula os segmentos iniciais de uma lista.

37. Considere as seguintes definições em Haskell:

```
data Nat = Zero | Suc Nat
data F x = P x | C (F x)
```

Para mostrar que (`Nat` $\times$ `A`) $\cong$ `F A` devemos definir os isomorfismos

```
nat2F :: (Nat,a) -> F a
f2Nat :: F a -> (Nat,a)
```

(a) Desenhe os diagramas dos catamorfismos e anamorfismos do tipo `F a`.

(b) Conjecture `nat2F` como um anamorfismo.

(c) Conjecture `f2Nat` como um catamorfismo.

(d) Mostre que a composição `f2Nat.nat2F` é na realidade a função identidade.

38. Considere o seguinte algoritmo *standard* para conversão de um número inteiro (não negativo) para base 2, expresso sob a forma de uma *lista de zeros ou uns*:

```
base2 :: Int -> [Int]
base2 0 = []
base2 x = let (q,r) = divMod x 2
          in (base2 q)++[r]
```

Por exemplo, $base2\,11 = 1011$ e de facto, se se avaliar `base2 11` em Haskell, obter-se-á a lista `[1,0,1,1]`.

(a) É possível definir $base2$ com base num anamorfismo:

$$base2 \;=\; reverse \cdot [\![g]\!]$$

Identifique o gene g desse anamorfismo.

(b) Seja agora $base10$ a inversa de $base2$, a função tal que

$$base2 \cdot base10 = id$$
$$base10 \cdot base2 = id$$

se verificam. Construa $base10$ com base num catamorfismo, i.e., identifique h em

$$base10 \quad = \quad (\!|h|\!).reverse$$

Sugestão: atente no exemplo seguinte:

$$base10\,[1,0,1,1] \quad = \quad (((1 \times 2) + 0) \times 2) + 1) \times 2 + 1$$

39. Considere o tipo de dados

```
data FList a b = Unit a | Add b (FList a b)
```

(a) Defina as operações `cata` e `ana` para esse tipo de dados. Acompanhe essas definições com os diagramas respectivos.

(b) O tipo apresentado é isomorfo ao tipo primitivo do Haskell `(a,[b])`. Defina as funções que testemunham esse isomorfismo respectivamente como um *catamorfismo* e um *anamorfismo* do tipo `FList`.

40. Considere as definições seguintes de um tipo de árvores binárias cujos nodos são etiquetados com números inteiros, e de uma função sobre essas árvores.

$$\mathsf{data\ IntBTree} = \mathsf{Empty} \mid \mathsf{Node}\ (\mathsf{Int}, (\mathsf{IntBTree}, \mathsf{IntBTree}))$$
$$\mathsf{heap2list} = (\![[\underline{[\,]}, \mathsf{cons} \cdot (\mathsf{id} \times \mathsf{merge})]]\!)_{\mathsf{IntBTree}}$$

em que $\mathsf{cons} = \mathsf{uncurry}\ (:)$ e $\mathsf{merge} : [\mathsf{Int}] \times [\mathsf{Int}] \to [\mathsf{Int}]$ é a função usual de fusão de listas ordenadas crescentemente.

(a) Escreva uma definição *pointwise* de $\mathsf{heap2list}$, justificando todos os passos para a sua obtenção.

(b) A ideia subjacente à função $\mathsf{heap2list}$ é que quando aplicada a determinadas árvores, ela permite obter listas ordenadas de forma crescente. As árvores em questão (usualmente designadas por *heaps*) caracterizam-se pelo facto de o conteúdo de cada nó ser inferior ou igual aos conteúdos de todos os nós seus descendentes.

Tendo isto em conta, é possível escrever a função `heap2list` como um anamorfismo de listas. Escreva essa definição.

Sugestão: comece por definir uma função que permita combinar duas *heaps*.

IV – Hilomorfismos

41. Uma das formas de calcular n^2, o quadrado de um número natural n, é somar os n primeiros ímpares. De facto, $1^2 = 1$, $2^2 = 1 + 3$, $3^2 = 1 + 3 + 5$, etc. Em geral, $n^2 = (2n - 1) + (n - 1)^2$. De acordo com esta sugestão, exprima a função $sq\ n = n^2$ sob a forma de um hilomorfismo de listas.

42. Relembre a questão 38. Uma outra forma de definir a função `base2` é

```
base2 = reverse . expande
expande 0 = []
expande x = let (q,r) = divMod x 2
            in q:(expande r)
```

Esta definição é um hilomorfismo. Sobre que estrutura? Qual das duas definições considera ser mais eficiente?

43. A seguinte função calcula a $(n + 1)$-ésima linha do triângulo de Pascal

```
pascal 0 = [1]
pascal (n+1) = 1:(soma (pn,(tail pn)))
        where pn = pascal n
```

em que `soma` é a função definida no exercício 27. Por exemplo, `pascal 0` é a lista `[1]`, `pascal 2` é a lista `[1, 2, 1]`, *etc*, como se ilustra em baixo:

$$
\begin{array}{ccccccccccc}
 & & & & & 1 & & & & & \\
 & & & & 1 & & 1 & & & & \\
 & & & 1 & & 2 & & 1 & & & \\
 & & 1 & & 3 & & 3 & & 1 & & \\
 & 1 & & 4 & & 6 & & 4 & & 1 & \\
1 & & 5 & & 10 & & 10 & & 5 & & 1
\end{array}
$$

Uma outra forma de definir a função `pascal` seria:

```
pascal 0 = [1]
pascal (n+1) = 1:(soma ((pascal n) ,(tail(pascal n))))
```

Estas duas soluções correspondem a usar dois hilomorfismos sobre estruturas de dados diferentes. Quais são estas estruturas?

44. (a) Considere a seguinte definição duma função que testa se uma lista contém elementos repetidos:

```
dups :: (Eq a) => [a] -> Bool
dups [] = False
dups (h:t) = (elem h t) || (dups t)
```

Explique por palavras suas porque é que esta função não pode ser expressa como um catamorfismo sobre listas. Defina-a como um hilomorfismo.

(b) Na sequência da alínea anterior, defina em Haskell o tipo de dados intermédio usado na definição do hilomorfismo. Mostre qual o termo desse tipo que é usado no cálculo de `dups [1,2,3,1,2,3]`.

45. Considere a função

```
span :: (a->bool) -> [a] -> ([a],[a])
span pred [] = []
span pred (x:xs) | pred x = let (l,r) = span pred xs in (x:l,r)
                 | otherwise = ([],x:xs)
```

Mostre como se pode definir `span` como um *hilomorfismo* do tipo `FList` (cfr. Ex. 39).

46. Escreva a função *fpc* do Ex. 11 como um hilomorfismo.

47. Assumindo as definições

$$\begin{array}{rcl} in & = & [\underline{0}, succ] \\ out & = & in^{-1} \\ g & = & [\underline{1}, mul \cdot (succ \times id)] \end{array}$$

o functor "números naturais"

$$\begin{cases} id + id \times map\, X = 1 + X \\ id + id \times map\, f = id + f \end{cases}$$

e o functor "listas de naturais"

$$\begin{cases} \mathsf{G}\ X = 1 + \mathbf{N} \times X = id + id \times map\,(\mathbf{N} \times X) \\ \mathsf{G}\ f = id + id \times f = id + id \times map\,(id \times f) \end{cases}$$

complete as reticências no seguinte processo de transformação da definição *pointfree* de factorial do exercício 31 do Cap. 1 num hilomorfismo de listas:

$$
\begin{aligned}
& \mathit{fac} \cdot \mathit{in} = g \cdot \mathit{id} + \mathit{id} \times \mathit{map}\,\langle \mathit{id}, \mathit{fac} \rangle \\
= \;& \{\ldots\ldots\ldots\} \\
& \mathit{fac} = g \cdot \mathit{id} + \mathit{id} \times \mathit{map}\,\langle \mathit{id}, \mathit{fac} \rangle \cdot \mathit{out} \\
= \;& \{\ldots\ldots\ldots\} \\
& \mathit{fac} = g \cdot \mathit{id} + \mathit{id} \times \mathit{map}\,((\mathit{id} \times \mathit{fac}) \cdot \langle \mathit{id}, \mathit{id} \rangle) \cdot \mathit{out} \\
= \;& \{\ldots\ldots\ldots\} \\
& \mathit{fac} = g \cdot \mathit{id} + \mathit{id} \times \mathit{map}\,(\mathit{id} \times \mathit{fac}) \cdot \mathit{id} + \mathit{id} \times \mathit{map}\,\langle \mathit{id}, \mathit{id} \rangle \cdot \mathit{out} \\
= \;& \{\ldots\ldots\ldots\} \\
& \mathit{fac} = g \cdot (\mathsf{G}\ \mathit{fac}) \cdot \mathit{id} + \mathit{id} \times \mathit{map}\,\langle \mathit{id}, \mathit{id} \rangle \cdot \mathit{out} \\
= \;& \{\ldots\ldots\ldots\} \\
& \mathit{fac} = [\![g, \mathit{id} + \mathit{id} \times \mathit{map}\,\langle \mathit{id}, \mathit{id} \rangle \cdot \mathit{out}]\!] \\
= \;& \{\ldots\ldots\ldots\} \\
& \mathit{fac} = (\!| g |\!) \cdot [\!(\mathit{id} + \mathit{id} \times \mathit{map}\,\langle \mathit{id}, \mathit{id} \rangle \cdot \mathit{out})\!]
\end{aligned}
$$

48. Considere a seguinte definição de uma função em Haskell:

```
parte :: [Int] -> ([Int],[Int])
parte [] = ([],[])
parte (x:xs)
      | odd x = let (l,r) = parte xs
                in (x:l,r)
      | otherwise = ([],x:xs)
```

Defina a função `parte` como um hilomorfismo, tornando explícito o tipo da estrutura de dados virtual.

49. Considere a definição da função sq do exercício 32 do Capítulo 1. Complete as justificações do seguinte processo de cálculo que converte sq num hilomorfismo de

listas.

$$
\begin{array}{ll}
 & sq \cdot in = [\underline{0}, add \cdot \langle odd, sq \rangle] \\
\equiv & \{\ldots\ldots\ldots\} \\
 & sq \cdot in = [\underline{0}, add] \cdot (id + \langle odd, sq \rangle) \\
\equiv & \{\ldots\ldots\ldots\} \\
 & sq \cdot in = [\underline{0}, add] \cdot (id + odd \times sq) \cdot (id + \langle id, id \rangle) \\
\equiv & \{\ldots\ldots\ldots\} \\
 & sq \cdot in = [\underline{0}, add] \cdot (id + id \times sq) \cdot (id + odd \times id) \cdot (id + \langle id, id \rangle) \\
\equiv & \{\ldots\ldots\ldots\} \\
 & sq = [\underline{0}, add] \cdot (id + id \times sq) \cdot (id + \langle odd, id \rangle) \cdot out \\
\equiv & \{\ldots\ldots\ldots\} \\
 & sq = [\![[\underline{0}, add], (id + \langle odd, id \rangle) \cdot out]\!] \\
\equiv & \{\ldots\ldots\ldots\} \\
 & sq = (\![[\underline{0}, add]]\!) \cdot [\!((id + \langle odd, id \rangle) \cdot out)\!]
\end{array}
$$

50. Apresente uma possível definição da função `cref` do exercício 30 escrita como um hilomorfismo.

51. Defina a função `resposta = uncurry resp` (cfr. exercício 18) como um hilomorfismo.

52. A seguir apresenta-se uma versão do famoso algoritmo de Euclides para calcular o máximo divisor comum entre dois números inteiros positivos:

```
mdc (x,y) = if  (x == y) then  x
            else mdc (x',y')
    where x' = (max x y) - y'
          y' = min x y
```

   (a) Exprima esta função como um hilomorfismo, acompanhando a resposta de um diagrama elucidativo.
   (b) Defina em Haskell o tipo de dados intermédio que obteve na alínea anterior e apresente o elemento desse tipo que é calculado no cálculo de `mdc (12,18)`

53. No contexto do exercício 31, pretende-se escrever um *hilomorfismo*

```
alternPrint :: (Int, Bool) -> IO()
```

que imprima a sequência de caracteres gerada por `altern`. Para isso escreva um *catamorfismo* `c` tal que `alternPrint = c.altern` seja uma definiçao válida da função desejada.

54. Mostre que a versão *uncurried* da função de concatenação de listas que consta do `Prelude` do Haskell,

```
(++)              :: [a] -> [a] -> [a]
[]     ++ ys       = ys
(x:xs) ++ ys       = x : (xs ++ ys)
```

pode ser convertida num hilomorfismo (cujos genes não recorrem, obviamente, à função `(++)`).

55. Considere a seguinte função que testa se uma árvore binária está equilibrada

```
equi Empty = True
equi (Node (x,(e,d))) = equi e  &&  equi d  &&
                                abs (alt e - alt d) <= 1
```

Defina `equi` como um hilomorfismo.

56. Considere a seguinte função `bubble`:

```
bubble p [] = []
bubble p [x] = [x]
bubble p (a:b:t) | p a b = a : bubble p (b:t)
                 | otherwise = b : bubble p (a:t)
```

(a) Escreva `bubble` como um *hilomorfismo*. Defina em Haskell e em C o respectivo tipo de dados intermédio.

(b) O tipo de dados intermédio obtido deve ser isomorfo a `(Maybe a, [(Bool,a)])`. Diga qual seria, neste tipo, a estrutura intermédia correspondente ao cálculo de

```
bubble (<) [2,1,4,3]
```

57. Seja

```
factores :: Integral a => a -> [(a,Int)]
```

a combinação das funções dos exercícios 32 e 22 num hilomorfismo (i.e., a função que calcula a decomposição em factores primos de um dado número inteiro maior do que 1). Calcule a versão *pointwise* dessa função, i.e., mostre que

```
factores n = if      m == n then [(n,1)]
             else if m == a then (a,b+1):ab
             else                (m,1):(a,b):ab
   where m          = minDiv n
         ((a,b):ab) = factores n `div` m
```

58. Sejam

$$g_a = (\underline{()} + \langle \mathsf{pred}, \mathsf{pred2} \rangle) \cdot \mathsf{zeroOrOne}?$$
$$g_c = [succ \cdot \underline{0}, \mathsf{plus}]$$
$$h = (\!| g_c |\!) \cdot [\!(g_a)\!]$$

(a) Desenhe o diagrama correspondente ao hilomorfismo h e declare em Haskell o seu tipo intermédio. Qual o valor deste tipo correspondente ao cálculo de h 4?

(b) Considere a seguinte definição recursiva *point-free*:

$$f = [\mathsf{succ} \cdot \underline{0}, \mathsf{plus} \cdot \langle f \cdot \mathsf{pred}, f \cdot \mathsf{pred2} \rangle] \cdot \mathsf{zeroOrOne}?$$

em que

$$\mathsf{zeroOrOne}\ x = (x == 0)\ ||\ (x == 1)$$
$$\mathsf{plus} = \mathsf{uncurry}\ (+)$$
$$\mathsf{pred2} = \mathsf{pred} \cdot \mathsf{pred}$$

Prove a equivalência $h = f$, justificando todos os passos.

59. A seguinte função em Haskell

```
sqrt’ p x = loop p x 1
         where loop p x r = let r’= (r + x / r) / 2
                            in if abs(r - r’) < p
                               then r’ else loop p x r’
```

calcula a raíz quadrada de um número x com erro p. Por exemplo,

```
> sqrt’ 0.01 2
1.41422 :: Double
>
```

(a) Represente *loop p x* como um hilomorfismo e faça um diagrama explicativo.

(b) Defina em Haskell o tipo de dados indutivo intermédio deste hilomorfismo (aquele que é saída do anamorfismo e entrada do catamorfismo) e represente o valor dessa estrutura para a situação em que *loop* é invocado de $sqrt'\ 1\,2$.

60. A seguinte função codifica um algoritmo de ordenação clássico, normalmente conhecido pelo nome de *heapsort*.

$$\mathsf{hsort} = [\mathsf{id}, \mathsf{cons} \cdot (\mathsf{id} \times (\mathsf{merge} \cdot (\mathsf{hsort} \times \mathsf{hsort}))) \cdot \mathsf{aux} \cdot \mathsf{cons}] \cdot \mathsf{out}_{\mathsf{RList}}$$

em que merge é a usual função de fusão de listas ordenadas; e aux é definida como se segue:

```
aux [h]   = (h,([],[]))
aux (h:t) = let (y,(l,r)) = aux t
            in if h<y then (h,(y:r,l))
                      else (y,(h:r,l))
```

(a) Defina hsort como um hilomorfismo e desenhe o diagrama correspondente.

(b) Declare em Haskell o tipo intermédio deste hilomorfismo e calcule o valor deste tipo que corresponde ao cálculo de $\mathsf{hsort}[10, 5, 9, 1, 8, 2, 4, 6, 7]$?

V– Hilomorfismos e Classes Algorítmicas *Standard*

61. O fragmento de código Haskell que se segue define parcialmente um componente de programação (módulo) Haskell que estudou:

```
inX = either Leaf Split

outX (Leaf a) = i1 a
outX (Split (t1,t2)) = i2 (t1,t2)

cataX a = a . (recX (cataX a)) . outX

hyloX a c = cataX a . anaX c
```

(a) Identifique, definindo-o, o tipo de dados paramétrico `X`, e acrescente ao fragmento dado as definições, que faltam, de `recX` e `anaX`.

(b) Complete a definição do operador functorial `map` associado ao tipo de dados `X`:

```
instance Functor X
        where map f = ...
```

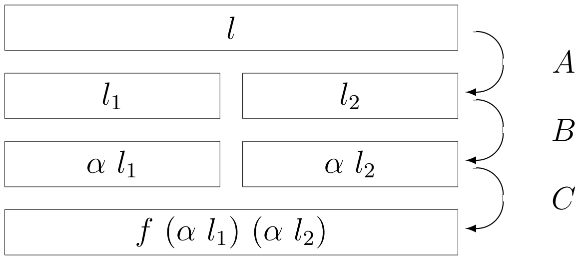
(c) Identifique quais dos seguintes algoritmos se definem nesse módulo como hilomorfismos do tipo `X`, identificando, quanto aos outros, os módulos a que pertencem:

Função	Algoritmo	Módulo (*.hs)
`mSort`	Merge sort	LTree.hs
`hanoi`	Torres de Hanoi	
`dfac`	Duplo factorial	
`qSort`	Quick sort	
`fac`	Factorial	
`fib`	Série de Fibonacci	
`iSort`	Odernação por inserção simples	

(d) Uma das vantagens de organizar o conhecimento algorítmico segundo o método estudado nesta disciplina é que se podem obter, dentro da mesma classe algorítmica, novos algoritmos pela simples combinação ou substituição de genes.

Que função se obtém de `mSort` substituindo-lhe um dos genes (qual?) pela função `cataLTree g`, onde `g = (either id (uncurry max))`? Faça um diagrama explicativo e converta-a, por cálculo, para Haskell com variáveis.

62. O seguinte desenho pretende descrever graficamente um algoritmo α de ordenação de listas bem conhecido:



(a) Identifique α, bem como as suas fases A, B e C e a função f. Codifique esta última em Haskell.

(b) Descreva o mesmo algoritmo sob a forma de um hilomorfismo de um particular tipo indutivo estudado nas aulas desta disciplina.

63. Pretende-se obter uma função que some os elementos de uma lista com eficiência semelhante à do algoritmo *quicksort*, i.e. do hilomorfismo

```
qSort = hyloBTree inord qsep
--    = (cataBTree inord) . (anaBTree qsep)
```

da biblioteca `BTree.hs`.

Qual a componente do hilomorfismo a modificar por forma a converter `qSort` na função pretendida? Justifique a sua resposta explicitando essa modificação.

64. (Resolva depois do Ex. 34 do Cap.1) Na biblioteca `RList.hs` a função factorial é apresentada como o hilomorfismo

```
fac = hyloRList (either (const 1) mul) obsNat
      where mul(x,y) = x*y
            obsNat 0 = Left ()
            obsNat (n+1) = Right (n+1,n)
```

Será a seguinte definição alternativa

```
fac' = hyloRList (either (const 1) g) obsNat'
   where g(n,m) = (n+1) * m
         obsNat' 0 = Left ()
         obsNat' (n+1) = Right (n,n)
```

equivalente a `fac`? Prove que o é, de facto, seguindo e justificando o raciocínio que se segue:

$$
\begin{array}{ll}
& fac' = [\![[\underline{1}, mul \cdot (succ \times id)], (id + \langle id, id \rangle) \cdot out_{\mathbf{N}_0}]\!] \\
\equiv & \{\ldots\ldots\ldots\} \\
& fac' = [\underline{1}, mul \cdot (succ \times id)] \cdot (id + id \times fac') \cdot (id + \langle id, id \rangle) \cdot out_{\mathbf{N}_0} \\
\equiv & \{\ldots\ldots\ldots\} \\
& fac' = [\underline{1}, mul] \cdot (id + succ \times id) \cdot (id + id \times fac') \cdot (id + \langle id, id \rangle) \cdot out_{\mathbf{N}_0} \\
\equiv & \{\ldots\ldots\ldots\} \\
& fac' = [\underline{1}, mul] \cdot (id + id \times fac') \cdot (id + succ \times id) \cdot (id + \langle id, id \rangle) \cdot out_{\mathbf{N}_0} \\
\equiv & \{\ldots\ldots\ldots\} \\
& fac' = [\underline{1}, mul] \cdot (id + id \times fac') \cdot (id + \langle succ, id \rangle) \cdot out_{\mathbf{N}_0} \\
\equiv & \{\ldots\ldots\ldots\} \\
& fac' = [\underline{1}, mul] \cdot (id + id \times fac') \cdot obsNat \\
\equiv & \{\ldots\ldots\ldots\} \\
& fac' = [\![[\underline{1}, mul], obsNat]\!] \\
\equiv & \{\ldots\ldots\ldots\} \\
& fac' = fac
\end{array}
$$

65. Considere o algoritmo *mergesort*. Neste contexto, indique o resultado de se aplicar ao resultado de `anaLTree lsplit [1,2,3,4]` a seguinte função:

```
f = cataLTree (inLTree . (id -|- swap))
```

66. Recorde a formulação como um hilomorfismo do algoritmo *quicksort*

```
qSort = hyloBTree inord qsep
--    = (cataBTree inord) . (anaBTree qsep)
```

que conhece da biblioteca `BTree.hs`, onde ocorrem os "genes"

```
inord = either (const []) join

qsep []    = Left ()
qsep (h:t) = Right (h,(s,l)) where (s,l) = part (<h) t
```

e as funções auxiliares

```
join(x,(l,r))=l++[x]++r

part p []                  = ([],[])
part p (h:t) | p h         = let (s,l) = part p t in (h:s,l)
             | otherwise   = let (s,l) = part p t in (s,h:l)
```

Pretende-se uma nova versão `qSort'` deste algoritmo que, para além de ordenar a lista argumento, lhe remove os elementos repetidos.

(a) Defina `qSort'` a partir do hilomorfismo `hyloBTree inord qsep`, alterando apenas o gene `qsep`.

(b) Repita a alínea anterior mudando agora apenas o gene `inord`.

(c) Comente a eficiência das duas versões alternativas das alíneas anteriores, sem se esquecer de abordar a situação seguinte: `qSort' l`, para `l` tal que `nub l = [1]` e `length l = 100`.

VI – Cálculo com Funções Recursivas

67. A igualdade que se segue é conhecida pelo nome de *banana-split* e traduz uma permutatividade célebre entre *splits* e catamorfismos.

$$\langle (\!| g |\!), (\!| k |\!) \rangle = (\!| \langle g \cdot \mathsf{F}\,\pi_1, k \cdot \mathsf{F}\,\pi_2 \rangle |\!)$$

   (a) Verifique que ambos os membros da igualdade exibem o mesmo tipo.

   (b) Mostre ainda que a lei se pode escrever da forma alternativa seguinte:

$$\langle (\!| g |\!), (\!| k |\!) \rangle = (\!| (g \times k) \cdot \langle \mathsf{F}\,\pi_1, \mathsf{F}\,\pi_2 \rangle |\!)$$

   (c) A função que calcula a média dos elementos de listas de números naturais

$$media \quad = \quad div \cdot \langle soma, comp \rangle$$

   combina dois catamorfismos: $soma = (\!| [\underline{0}, add] |\!)$ e $comp = (\!| [\underline{0}, succ \cdot \pi_2] |\!)$. Aplique à função *media* a lei anterior e traduza a função resultado para a notação com variáveis. Qual é a vantagem desta última em termos de eficiência?

68. Use a lei de fusão-cata

$$\begin{tikzcd}
\mu\mathsf{F} \arrow[d, "(\!|\alpha|\!)"'] \arrow[dd, bend right=60, "(\!|\beta|\!)"'] & \mathsf{F}\,(\mu\mathsf{F}) \arrow[l, "in"'] \arrow[d, "\mathsf{F}\,(\!|\alpha|\!)"] \\
B \arrow[d, "f"'] & \mathsf{F}\,B \arrow[l, "\alpha"] \arrow[d, "\mathsf{F}\,f"] \\
C & \mathsf{F}\,C \arrow[l, "\beta"]
\end{tikzcd}$$

   para provar a validade do seguinte facto, em Haskell:

   \texttt{ x + foldr (+) y = foldr (+) (x+y)}.

   Sugestão: interprete `foldr` como um catamorfismo de listas e faça $f = (x+)$.

69. Uma das operações conhecidas sobre listas é a da inversão:

```
invl [] = []
invl (a:l) = (invl l) ++ [a]
```

(a) Calcule a definição de $invl$, dada acima em Haskell, a partir do seguinte catamorfismo:

$$invl = (\!| \underbrace{[nil, uconc \cdot swap \cdot (singl \times id)]}_{g} |\!)$$

onde $nil = \underline{[\,]}$ e $uconc = uncurry(+\!+)$, apoiando a sua resposta por um diagrama explicativo.

(b) Converta para notação com variáveis a propriedade

$$invl \cdot uconc = uconc \cdot (invl \times invl) \cdot swap$$

e complete as igualdades seguintes por forma a exprimirem também propriedades válidas:

$$\begin{aligned} invl \cdot singl &= \ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ uconc \cdot (\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ ) &= cons \end{aligned}$$

em que $cons(a, l) = a : l$.

(c) Complete as justificações da seguinte prova da propriedade involutiva de $invl$:

$$\begin{aligned}
& invl \cdot invl = id \\
\Leftrightarrow \quad & \{\ldots\ldots\ldots\} \\
& invl \cdot (\!|g|\!) = (\!|in|\!) \\
\Leftrightarrow \quad & \{\ldots\ldots\ldots\} \\
& invl \cdot g = in \cdot (id + id \times invl) \\
\Leftrightarrow \quad & \{\ldots\ldots\ldots\} \\
& invl \cdot [nil, uconc \cdot swap \cdot (singl \times id)] = in \cdot (id + id \times invl) \\
\Leftrightarrow \quad & \{\ldots\ldots\ldots\} \\
& [invl \cdot nil, invl \cdot uconc \cdot swap \cdot (singl \times id)] = in \cdot (id + id \times invl) \\
\Leftrightarrow \quad & \{\ldots\ldots\ldots\} \\
& [nil, uconc \cdot (invl \times invl) \cdot swap \cdot swap \cdot (singl \times id)] = in \cdot (id + id \times invl) \\
\Leftrightarrow \quad & \{\ldots\ldots\ldots\} \\
& [nil, uconc \cdot (invl \cdot singl \times invl)] = in \cdot (id + id \times invl) \\
\Leftrightarrow \quad & \{\ldots\ldots\ldots\} \\
& [nil, uconc \cdot (singl \times invl)] = in \cdot (id + id \times invl) \\
\Leftrightarrow \quad & \{\ldots\ldots\ldots\} \\
& [nil, cons \cdot (id \times invl)] = in \cdot (id + id \times invl) \\
\Leftrightarrow \quad & \{\ldots\ldots\ldots\} \\
& [nil, cons] \cdot (id + id \times invl) = in \cdot (id + id \times invl) \\
\Leftrightarrow \quad & \{\ldots\ldots\ldots\} \\
& True
\end{aligned}$$

70. No contexto do exercício 40, seja $\mathsf{countNodes}$ a função assim definida:

```
countNodes Empty = 0
countNodes (Node(x,(e,d)) = succ ((countNodes e) + (countNodes d))
```

Prove, justificando todos os passos, a igualdade

$$\mathsf{countNodes} = \mathsf{length} \cdot \mathsf{heap2list}$$

N.B. Deverá aplicar a lei de fusão dos catamorfismos. Assuma como válida a equivalência $\mathsf{length} \cdot \mathsf{merge} = (\mathsf{uncurry}(+)) \cdot (\mathsf{length} \times \mathsf{length})$.

71. Considere a função que realiza a partição de uma lista s em duas outras listas que recolhem, respectivamente, os elementos de s que verificam ou falham um determinado predicado p:

$$\mathsf{partition}\,\mathsf{s} \;=\; \langle \mathsf{filter}\ \mathsf{p}, \mathsf{filter}\ \neg\mathsf{p} \rangle$$

Exprima esta função como um catamorfismo, por aplicação da respectiva lei de fusão.

VII – Outros Tópicos

72. Podemos representar os números interios (não negativos) como listas do tipo singular (i.e. o tipo `[()]` do *Haskell*).

    (a) Defina em *Haskell* as funções de representação e abstracção

    ```
    repres :: Int -> [()]
    abstr :: [()] -> Int
    ```

    (b) Codifique funções que calculem a *soma* e *multiplicação* de inteiros *na representação sugerida.*

73. Uma *árvore de pedigree* ($\mathsf{P}a$) descreve um animal (a) — por exemplo, um canídeo — e indica quais os seus ascendentes conhecidos (o pai e/ou a mãe), o seu pedigree, e assim sucessivamente:

$$\mathsf{P}a \;\cong\; a \times (1 + \mathsf{P}a) \times (1 + \mathsf{P}a)$$

    (a) Codifique o tipo $\mathsf{P}a$ em Haskell.

    (b) Uma possível alternativa para a codificação da informação requerida na questão anterior seria:

    ```
    data PedAlt x = N x
                  | SPai (x,PedAlt x)
                  | SMae (x,PedAlt x)
                  | Ambos (x,PedAlt x,PedAlt x)
    ```

    Codifique em `Halkell` as funções que estabelecem o isomorfismo entre ambos os tipos de dados.

74. A habitual definição da função factorial

$$\begin{array}{rcl} fac\ 0 & = & 1 \\ fac\ (n+1) & = & (n+1) \times (fac\ n) \end{array}$$

    pode ser calculada a partir do diagrama que se segue

$$\begin{tikzcd} \mathbf{N} \arrow[d, "fac"'] & 1 + \mathbf{N} \arrow[l, "in"'] \arrow[d, "id+\langle id,fac\rangle"] \\ \mathbf{N} & 1 + (\mathbf{N} \times \mathbf{N}) \arrow[l, "g"] \end{tikzcd}$$

onde $in = [\underline{0}, succ]$ e $g = [\underline{1}, mul \cdot (succ \times id)]$.

Repare que o diagrama é um tudo nada mais elaborado do que o habitual catamorfismo sobre **Nat**. De facto, é um caso particular de *paramorfismo*. No caso geral, dado um tipo indutivo $T \cong \mathsf{F}\ T$, o paramorfismo de g relativamente ao functor F, designado por $\langle\!|g|\!\rangle_{\mathsf{F}}$ é tal que

$$\begin{tikzcd} T \arrow[d, "\langle\!|g|\!\rangle"'] & \mathsf{F}\ T \arrow[l, "in"'] \arrow[d, "F\langle id, \langle\!|g|\!\rangle\rangle"] \\ C & \mathsf{F}\ (T \times C) \arrow[l, "g"] \end{tikzcd}$$

que traduz a seguinte propriedade universal:

$$h = \langle\!|g|\!\rangle \quad \Longleftrightarrow \quad h \cdot in = g \cdot \mathsf{F}\langle id, h\rangle$$

(a) Escreva a definição de *fac* como um paramorfismo.

(b) A partir da propriedade universal deduza a regra de **reflexão-para**:

$$id = \langle\!|in \cdot \pi_1|\!\rangle$$

Verificar; não será $id = \langle\!|in \cdot \mathsf{F}\ \pi_1|\!\rangle$??

(c) Sabendo que uma das formas de calcular n^2, o quadrado de um número natural n, é somar os n primeiros ímpares — $1^2 = 1$, $2^2 = 1 + 3$, $3^2 = 1 + 3 + 5$, etc, $n^2 = (2n - 1) + (n - 1)^2$ — exprima a função $sq\ n = n^2$ sob a forma de um paramorfismo em **Nat**.

75. Considere a seguinte função que utiliza um parâmetro de acumulação para calcular a média de uma lista de inteiros:

```
media' :: [Int] -> (Int, Int) -> Int
media' [] (a,b) = a/b
media' (h:t) (a,b) = media' t (h+a,b+1)
```

Note que a média de uma lista l é calculada como $\mathsf{media}'\ l\ (0, 0)$. Defina esta função media' como um catamorfismo (de ordem superior).

Capítulo 3

Monads

1. Na programação funcional é vulgar a ocorrência de funções parciais, i.e., funções indefinidas para algum dos seus argumentos. Por exemplo, a divisão é parcial pois $n/0$ é um valor indefinido, ou *excepção*. As excepções são vulgarmente assinaladas através de mensagens de erro, estendendo-se o codomínio da função por forma a fornecer *strings* explicativos. Em Haskell, por exemplo,

   ```
   (/) :: Double -> Double -> Double
   ```

   pode ser estendida a

   ```
   dv :: (Double,Double) -> Error Double
   dv(n,0) = Err "Nem pense em dividir por 0!"
   dv(n,m) = Ok (n / m)
   ```

   onde

   ```
   data Error a = Err String | Ok a deriving Show
   ```

   Outro exemplo ocorre no processamento de listas

   ```
   hd [] = Err "Lista vazia!"
   hd (a:_) = Ok a
   ```

   Para evitar a proliferação arbitrária de condições de teste de excepções e seu processamento pode definir-se um combinador de composição de funções parciais da forma seguinte:

   ```
   (.!) :: (a -> Error b) -> (c -> Error a) -> c -> Error b
   g .! f = errh . (fmap g) . f
   ```

   assumindo um combinador de excepções `errh` (=*error handler*)

```
errh (Err e) = Err e
errh (Ok  a) = a
```

e

```
instance Functor Error where
    fmap f (Ok a) = Ok (f a)
    fmap f (Err e) = Err e
```

(a) Qual o tipo de $errh$? Escreva a mesma função em notação sem variáveis, acompanhada de um diagrama explicativo, com base no isomorfismo

$$\mathtt{Error}\, a \cong \mathtt{String} + a$$

testemunhado pela função $in = [Err, Ok]$.

(b) Calcule a definição em Haskell de $fmap$ a partir da interpretação do diagrama que se segue:

$$\begin{tikzcd} a \arrow[d, "f"] & \mathtt{Error}\, a \arrow[d, "fmap\, f"'] & \mathtt{String} + a \arrow[l, "in"'] \arrow[d, "id+f"] \\ b & \mathtt{Error}\, b & \mathtt{String} + b \arrow[l, "in"] \end{tikzcd}$$

(c) Preencha as reticências ...A... a ...F... na seguinte prova de functorialidade de $fmap$:

$$\begin{array}{l}
fmap\,(f \cdot g) = (fmap\, f) \cdot (fmap\, g) \\
\qquad\quad \equiv \ldots A \ldots \\
(fmap\,(f \cdot g)) \cdot in = (fmap\, f) \cdot (fmap\, g) \cdot in \\
\qquad\quad \equiv \ldots B \ldots \\
(fmap\,(f \cdot g)) \cdot in = (fmap\, f) \cdot in \cdot (id + g) \\
\qquad\quad \equiv \ldots C \ldots \\
(fmap\,(f \cdot g)) \cdot in = in \cdot (id + f) \cdot (id + g) \\
\qquad\quad \equiv \ldots D \ldots \\
(fmap\,(f \cdot g)) \cdot in = in \cdot (id + f \cdot g) \\
\qquad\quad \equiv \ldots E \ldots \\
(fmap\,(f \cdot g)) \cdot in = (fmap\,(f \cdot g)) \cdot in \\
\qquad\quad \equiv \ldots F \ldots \\
\text{True}
\end{array}$$

(d) Defina `Error` como instância da classe `Monad`, isto é, preencha as reticências em

```
instance Monad Error where

    return ............................

    ....... >>= .......................

    ...................................
```

2. Na interpretação de comandos é vulgar disponibilizar-se um modo de execução dito "verbose" sempre que uma função ou comando, ao executar, "explica" o que está a fazer. A opção "`-v`" é muitas vezes usada para esse efeito.

   A execução "verbose" é também muito utilizada em "debug", permitindo ao programador acompanhar o trajecto de uma execução até à ocorrência de um erro, por exemplo. Em programação imperativa (eg. C), a adição de instruções de saída tipo `printf` é uma forma primitiva de introduzir verbosidade. Em Haskell, esse processo pode ser sistematizado através do recurso a uma mónada particular que prevê "logs" (textos explicativos) anexos ao resultado de funções:

   ```
   data Verbose a = Verb (a, Text)

   type Text = [ String ]

   vap :: Text -> (a -> b) -> a -> Verbose b        -- verbose apply
   vap t f a = Verb(f a,t)
   ```

   (a) Com base no seguinte exemplo de composição (monádica) entre duas funções "verbose",

   ```
   Verbose> ((vap ["First succ: ok!"] succ) .!
             (vap ["Second succ: ok!"] succ)) 3
   Verb (5,["Second succ: ok!","First succ: ok!"])
   ```

   complete a seguinte definição da mónada subjacente:

   ```
   instance Monad Verbose where

       return ............................

       .......... >>= ....................

       ...................................

       ...................................
   ```

(b) Defina `Verbose` como instância da classe `Functor`.

3. Dê definições para as funções f_1 a f_9 da seguinte tabela, onde se apresentam três tipos paramétricos de dados em Haskell que são instâncias da classe `Monad`:

F a	`return`	`(>>=)`	μ
`Maybe a`	f_1	f_2	f_3
`[a]`	f_4	f_5	f_6
`Error a`	f_7	f_8	f_9

onde `data Error a = Err String | Ok a`.

4. Considere a seguinte definição de funções de transformação de estado.

```
data ST estado valor = ST estado -> (estado,valor)
```

(a) Complete a seguinte definição:

```
instance Monad (ST estado) where
   return a = ST (\s -> ...)
   (ST g) >>= f = ST (\s -> ...)
```

(b) Para esta instância de `Monad`, defina a multiplicação μ.

(c) Defina `ST s` como uma instância da classe `Functor`.

5. Considere o problema de se construir, a partir de uma lista arbitrária de tipo `a`, uma lista com o mesmo comprimento de tipo `[(a,Bool,Int)]` em que o booleano corresponde à verificação (ou não) de um predicado, e o inteiro corresponde a uma contagem dos elementos que verificaram (ou não) o predicado, *do fim para o início da lista.*

Por exemplo:

```
> ocorrencias (<5) [1,3,5,7,2,5,8,9,4]
[(1,True,4),(3,True,3),(5,False,5),(7,False,4),(2,True,2),
 (5,False,3),(8,False,2),(9,False,1),(4,True,1)]
```

O problema pode ser resolvido com o auxílio de um par auxiliar `(Int,Int)` em que se vai contando o número de elementos que verificaram ou não o predicado:

```
ocorrencias :: (a->Bool) -> [a] -> [(a,Bool,Int)]
ocorrencias p l = let (s,l') = ocorr p l (0,0) in l'

ocorr :: (a->Bool) -> [a] -> (Int,Int) -> ((Int,Int) , [(a,Bool,Int)])
ocorr p [] s = (s,[])
ocorr p (h:t) s = let ((st,sf), l) = ocorr p t s
                  in if (p h) then ((st+1,sf) , (h,True,st+1):l)
                              else ((st,sf+1) , (h,False,sf+1):l)
```

Uma solução alternativa para este problema recorre a uma mónade de estado:

```
data Trans a = Trans ((Int,Int) -> ((Int,Int) , a))
instance Monad Trans where ... -- [assuma definicao habitual]
```

(a) Complete a definição da versão monádica das funções `ocorr` e `ocorrencias`:

```
mocorr :: (a->Bool) -> [a] -> Trans [(a,Bool,Int)]
mocorr p [] = Trans (\(nt,nf) -> ((nt,nf) , []))
mocorr p (h:t) = do l <- ..........
                    n <- Trans f
                    return ((h, p h, n):l)
     where f = if (p h) then \(nt,nf) -> ((nt+1,nf),nt+1
                        else ..........

mocorrencias :: (a->Bool) -> [a] -> [(a,Bool,Int)]
mocorrencias p l = let (Trans f) = mocorr p l
                   in ..........
```

(b) Resolva uma variante deste problema em que a contabilização dos elementos que verificam / não verificam o predicado é feita do início para o fim da lista.

6. Para captar o tipo das funções que transformam estado definiu-se o seguinte tipo de dados

```
data ST s a = ST { st :: s -> (a,s) }
```

(a) Defina (`ST s`) como uma instância da classe `Monad`.

(b) Usando o monade de transformação de estados construa uma função para etiquetar uma árvore binária, marcando cada nodo com um número correspondente à ordem pela qual o nodo é visitado numa travessia *inorder*.

7. Mostre que, para toda a mónade F, se tem

$$(\mathsf{F}\ f)\ x = do\ \{\ a\ \leftarrow x\ ;\ return(f\ a)\ \}$$

8. Recorde o tipo de dados

```
data LTree a = Leaf a | Split (LTree a, LTree a)
```

que consta de uma biblioteca que estudou nesta disciplina.

(a) Defina como um catamorfismo deste tipo a função de *multiplicação*

$$\mu \;:\; LTree(LTree\ a) \rightarrow LTree\ a$$

que permita encarar $LTree$ como uma mónade.
NB: acompanhe a sua resposta com um diagrama explicativo.

(b) Qual a correspondente unidade (função `return` em Haskell)? Mostre que

$$\mu \cdot return = id$$

9. Complete a demonstração que se segue do facto

$$(\mathsf{F}\ f)\ x = do\ \{\ a\ \leftarrow x\ ;\ return(f\ a)\ \}$$

válido para toda a mónade F:

$$
\begin{array}{cl}
 & do\ \{\ a\ \leftarrow x\ ;\ return(f\ a)\ \} \\
= & \qquad \{\ \ldots \text{(justifique)} \ldots \} \\
 & x \mathbin{>\!\!>\!\!=} \lambda a.\ return(f\ a) \\
= & \qquad \{\ \ldots \text{(justifique)} \ldots \} \\
 & x \mathbin{>\!\!>\!\!=} (return \cdot f) \\
= & \qquad \{\ \ldots \text{(justifique)} \ldots \} \\
 & (\mu \cdot \mathsf{F}\ (return \cdot f))\ x \\
= & \qquad \{\ \ldots \text{(justifique)} \ldots \} \\
 & (\mu \cdot (\mathsf{F}\ return) \cdot (\mathsf{F}\ f))\ x \\
= & \qquad \{\ \ldots \text{(justifique)} \ldots \} \\
 & (id \cdot (\mathsf{F}\ f))x \\
= & \qquad \{\ \ldots \text{(justifique)} \ldots \} \\
 & (\mathsf{F}\ f)x
\end{array}
$$

10. O seguinte código diz respeito a uma função `label` que a partir de uma árvore de folhas de inteiros, produz uma outra árvore com a mesma forma, em que o conteúdo de cada folha é substituído pela sua soma com os conteúdos de todos as folhas à sua esquerda. Por exemplo, `label (Node (Leaf 1) (Node (Leaf 2) (Leaf 3)))` resulta em `Node (Leaf 1) (Node (Leaf 3) (Leaf 6))`. Assumindo que o tipo `Estado` foi já apropriadamente declarado como instância da classe Monad, complete o código em A e B.

```
data Estado a = E (Int -> (Int, a))

lab :: Tree Int -> Estado (Tree Int)
lab (Leaf x) = ............A...............
lab (Node esq dir) = do esq' <- lab esq
                        dir' <- lab dir
                        return (Node esq' dir')

label :: Tree a -> Tree Int
label t = ..........B..............
```

11. Se pedir ao GHC informações sobre a class `Monad`,

```
Prelude> :i Monad
```

obterá

```
-- Monad is a class
class Monad m :: (* -> *) where {
    (>>=) :: forall a b. m a -> (a -> m b) -> m b;
    (>>) :: forall a b. m a -> m b -> m b {- has default method -};
    return :: forall a. a -> m a;
    fail :: forall a. String -> m a {- has default method -};
    }
```

Idenfifique qual das funções disponibilizadas por esta classe corresponde à seguinte função f, em Haskell

```
f x y = (x >>= return) >>= (const y)
```

Sugestão: Poderá ser-lhe útil recordar das aulas teóricas, o diagrama

$$\begin{tikzcd} \mathsf{F}^2\,A \arrow[d, "\mu"'] & \mathsf{F}\,A \arrow[l, "u"'] \arrow[d, "\mathsf{F}\ u"] \arrow[dl, "id"] \\ \mathsf{F}\,A & \mathsf{F}^2\,A \arrow[l, "\mu"] \end{tikzcd}$$

12. A definição em Haskell que se segue

```
mfold k f [] = k
mfold k f (h:t) = do { b <- mfold k f t ; f h b }
```

estende o combinador `foldr` no contexto de uma mónade arbitrária. Qual o tipo mais geral de *mfold*? Complemente a sua resposta indicando instâncias da sua aplicação a habitantes dos tipos monádicos $[a]$ e $Maybe\ a$.