

# Cálculo de Programas

## Trabalho Prático

### MiEI+LCC — 2018/19

Departamento de Informática  
Universidade do Minho

Junho de 2019

| Grupo nr. | 63              |
|-----------|-----------------|
| a63129    | Duarte Freitas  |
| a75614    | Nadine Oliveira |
| a78029    | Catarina Sousa  |

## 1 Preâmbulo

A disciplina de **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, restringe-se a aplicação deste método à programação funcional em **Haskell**. 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, 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 `cp1819t.pdf` que está a ler é já um exemplo de **programação literária**: foi gerado a partir do texto fonte `cp1819t.lhs`<sup>1</sup> que encontrará no **material pedagógico** desta disciplina descompactando o ficheiro `cp1819t.zip` e executando

```
$ lhs2TeX cp1819t.lhs > cp1819t.tex
$ pdflatex cp1819t
```

em que **lhs2tex** é um pre-processor que faz “pretty printing” de código Haskell em **L<sup>A</sup>T<sub>E</sub>X** e que deve desde já instalar executando

```
$ cabal install lhs2tex
```

Por outro lado, o mesmo ficheiro `cp1819t.lhs` é executável e contém o “kit” básico, escrito em **Haskell**, para realizar o trabalho. Basta executar

```
$ ghci cp1819t.lhs
```

---

<sup>1</sup>O suffixo ‘lhs’ quer dizer *literate Haskell*.

Abra o ficheiro `cp1819t.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}
```

vai ser seleccionado pelo **GHCi** para ser executado.

### 3 Como realizar o trabalho

Este trabalho teórico-prático deve ser realizado por grupos de três 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 cp1819t.aux
$ makeindex cp1819t.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
```

Para testar uma propriedade **QuickCheck** *prop*, basta invocá-la com o comando:

```
> quickCheck prop'
+++ OK, passed 100 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.

### Problema 1

Um compilador é um programa que traduz uma linguagem dita de *alto nível* numa linguagem (dita de *baixo nível*) que seja executável por uma máquina. Por exemplo, o **GCC** compila C/C++ em código objecto que corre numa variedade de arquitecturas.

Compiladores são normalmente programas complexos. Constan essencialmente de duas partes: o *analisador sintático* que lê o texto de entrada (o programa *fonte* a compilar) e cria uma sua representação interna, estruturada em árvore; e o *gerador de código* que converte essa representação interna em código executável. Note-se que tal representação intermédia pode ser usada para outros fins, por exemplo, para gerar uma listagem de qualidade (*pretty print*) do programa fonte.

O projecto de compiladores é um assunto complexo que será assunto de outras disciplinas. Neste trabalho pretende-se apenas fazer uma introdução ao assunto, mostrando como tais programas se podem construir funcionalmente à custa de cata/ana/hilo-morfismos da linguagem em causa.

Para cumprirmos o nosso objectivo, a linguagem desta questão terá que ser, naturalmente, muito simples: escolheu-se a das expressões aritméticas com inteiros, *eg.*  $1+2$ ,  $3*(4+5)$  etc. Como representação interna adopta-se o seguinte tipo polinomial, igualmente simples:

```
data Expr = Num Int | Bop Expr Op Expr
data Op = Op String
```

1. Escreva as definições dos {cata, ana e hilo}-morfismos deste tipo de dados segundo o método ensinado nesta disciplina (recorde módulos como *eg.* `BTree` etc).

2. Como aplicação do módulo desenvolvido no ponto 1, defina como  $\{cata, ana \text{ ou } hilo\}$ -morfismo a função seguinte:

- $calcula :: Expr \rightarrow Int$  que calcula o valor de uma expressão;

**Propriedade QuickCheck 1** O valor zero é um elemento neutro da adição.

```
prop_neutro1 :: Expr → Bool
prop_neutro1 = calcula · addZero ≡ calcula where
  addZero e = Bop (Num 0) (Op "+") e
prop_neutro2 :: Expr → Bool
prop_neutro2 = calcula · addZero ≡ calcula where
  addZero e = Bop e (Op "+") (Num 0)
```

**Propriedade QuickCheck 2** As operações de soma e multiplicação são comutativas.

```
prop_comuta = calcula · mirror ≡ calcula where
  mirror = cataExpr [Num, g2]
  g2 =  $\widehat{\widehat{Bop}} \cdot (swap \times id) \cdot assocl \cdot (id \times swap)$ 
```

3. Defina como  $\{cata, ana \text{ ou } hilo\}$ -morfismos as funções

- $compile :: String \rightarrow Codigo$  - trata-se do compilador propriamente dito. Deverá ser gerado código posfixo para uma máquina elementar de **stack**. O tipo *Codigo* pode ser definido à escolha. Dão-se a seguir exemplos de comportamentos aceitáveis para esta função:

```
Tp4> compile "2+4" '
["PUSH 2", "PUSH 4", "ADD"] '
Tp4> compile "3*(2+4)" '
["PUSH 3", "PUSH 2", "PUSH 4", "ADD", "MUL"] '
Tp4> compile "(3*2)+4" '
["PUSH 3", "PUSH 2", "MUL", "PUSH 4", "ADD"] '
Tp4> '
```

- $show' :: Expr \rightarrow String$  - gera a representação textual de uma *Expr* pode encarar-se como o *pretty printer* associado ao nosso compilador

**Propriedade QuickCheck 3** Em anexo, é fornecido o código da função *readExp*, que é “inversa” da função *show'*, tal como a propriedade seguinte descreve:

```
prop_inv :: Expr → Bool
prop_inv =  $\pi_1 \cdot head \cdot readExp \cdot show' \equiv id$ 
```

**Valorização** Em anexo é apresentado código **Haskell** que permite declarar *Expr* como instância da classe *Read*. Neste contexto, *read* pode ser vista como o analisador sintático do nosso minúsculo compilador de expressões aritméticas.

Analise o código apresentado, corra-o e escreva no seu relatório uma explicação **breve** do seu funcionamento, que deverá saber defender aquando da apresentação oral do relatório.

Exprima ainda o analisador sintático *readExp* como um anamorfismo.

## Problema 2

Pretende-se neste problema definir uma linguagem gráfica “brinquedo” a duas dimensões (2D) capaz de especificar e desenhar agregações de caixas que contêm informação textual. Vamos designar essa linguagem por *L2D* e vamos defini-la como um tipo em **Haskell**:

```
type L2D = X Caixa Tipo
```

onde *X* é a estrutura de dados

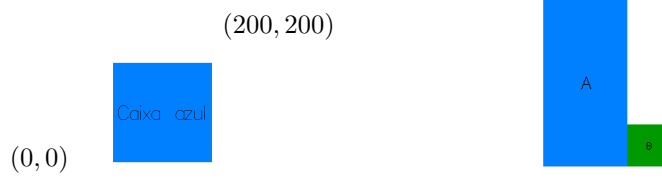


Figure 1: Caixa simples e caixa composta.

**data**  $X \ a \ b = Unid \ a \mid Comp \ b \ (X \ a \ b) \ (X \ a \ b)$  **deriving** *Show*

e onde:

**type**  $Caixa = ((Int, Int), (Texto, G.Color))$   
**type**  $Texto = String$

Assim, cada caixa de texto é especificada pela sua largura, altura, o seu texto e a sua cor.<sup>2</sup> Por exemplo,

$((200, 200), ("Caixa \ azul", col\_blue))$

designa a caixa da esquerda da figura 1.

O que a linguagem *L2D* faz é agregar tais caixas tipográficas umas com as outras segundo padrões especificados por vários “tipos”, a saber,

**data**  $Tipo = V \mid Vd \mid Ve \mid H \mid Ht \mid Hb$

com o seguinte significado:

- $V$  - agregação vertical alinhada ao centro
- $Vd$  - agregação vertical justificada à direita
- $Ve$  - agregação vertical justificada à esquerda
- $H$  - agregação horizontal alinhada ao centro
- $Hb$  - agregação horizontal alinhada pela base
- $Ht$  - agregação horizontal alinhada pelo topo

Como *L2D* instancia o parâmetro  $b$  de  $X$  com  $Tipo$ , é fácil de ver que cada “frase” da linguagem *L2D* é representada por uma árvore binária em que cada nó indica qual o tipo de agregação a aplicar às suas duas sub-árvores. Por exemplo, a frase

$ex2 = Comp \ Hb \ (Unid \ ((100, 200), ("A", col\_blue)))$   
 $\quad \quad \quad (Unid \ ((50, 50), ("B", col\_green)))$

deverá corresponder à imagem da direita da figura 1. E poder-se-á ir tão longe quando a linguagem o permita. Por exemplo, pense na estrutura da frase que representa o *layout* da figura 2.

É importante notar que cada “caixa” não dispõe informação relativa ao seu posicionamento final na figura. De facto, é a posição relativa que deve ocupar face às restantes caixas que irá determinar a sua posição final. Este é um dos objectivos deste trabalho: *calcular o posicionamento absoluto de cada uma das caixas por forma a respeitar as restrições impostas pelas diversas agregações*. Para isso vamos considerar um tipo de dados que comporta a informação de todas as caixas devidamente posicionadas (i.e. com a informação adicional da origem onde a caixa deve ser colocada).

<sup>2</sup>Pode relacionar *Caixa* com as caixas de texto usadas nos jornais ou com *frames* da linguagem HTML usada na Internet.



Figure 2: *Layout* feito de várias caixas coloridas.

```
type Fig = [(Origem, Caixa)]
type Origem = (Float, Float)
```

A informação mais relevante deste tipo é a referente à lista de “caixas posicionadas” (tipo  $(Origem, Caixa)$ ). Regista-se aí a origem da caixa que, com a informação da sua altura e comprimento, permite definir todos os seus pontos (consideramos as caixas sempre paralelas aos eixos).

1. Forneça a definição da função *calc\_origems*, que calcula as coordenadas iniciais das caixas no plano:

$$calc\_origems :: (L2D, Origem) \rightarrow X (Caixa, Origem) ()$$

2. Forneça agora a definição da função *agrup\_caixas*, que agrupa todas as caixas e respectivas origens numa só lista:

$$agrup\_caixas :: X (Caixa, Origem) () \rightarrow Fig$$

Um segundo problema neste projecto é *descobrir como visualizar a informação gráfica calculada por desenho*. A nossa estratégia para superar o problema baseia-se na biblioteca **Gloss**, que permite a geração de gráficos 2D. Para tal disponibiliza-se a função

$$crCaixa :: Origem \rightarrow Float \rightarrow Float \rightarrow String \rightarrow G.Color \rightarrow G.Picture$$

que cria um rectângulo com base numa coordenada, um valor para a largura, um valor para a altura, um texto que irá servir de etiqueta, e a cor pretendida. Disponibiliza-se também a função

$$display :: G.Picture \rightarrow IO ()$$

que dado um valor do tipo *G.picture* abre uma janela com esse valor desenhado. O objectivo final deste exercício é implementar então uma função

$$mostra\_caixas :: (L2D, Origem) \rightarrow IO ()$$

que dada uma frase da linguagem *L2D* e coordenadas iniciais apresenta o respectivo desenho no ecrã.

**Sugestão:** Use a função *G.pictures* disponibilizada na biblioteca **Gloss**.

## Problema 3

Nesta disciplina estudou-se como fazer **programação dinâmica** por cálculo, recorrendo à lei de recursividade mútua.<sup>3</sup>

Para o caso de funções sobre os números naturais ( $\mathbb{N}_0$ , com functor  $F X = 1 + X$ ) é fácil derivar-se da lei que foi estudada uma *regra de algibeira* que se pode ensinar a programadores que não tenham estudado **Cálculo de Programas**. Apresenta-se de seguida essa regra, tomando como exemplo o cálculo do ciclo-for que implementa a função de Fibonacci, recordar o sistema

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

Obter-se-á de imediato

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

usando as regras seguintes:

- O corpo do ciclo *loop* terá tantos argumentos quanto o número de funções mutuamente recursivas.
- Para as variáveis escolhem-se os próprios nomes das funções, pela ordem que se achar conveniente.<sup>4</sup>
- Para os resultados vão-se buscar as expressões respectivas, retirando a variável  $n$ .
- Em *init* colecionam-se os resultados dos casos de base das funções, pela mesma ordem.

Mais um exemplo, envolvendo polinómios no segundo grau a  $x^2 + bx + c$  em  $\mathbb{N}_0$ . Seguindo o método estudado nas aulas<sup>5</sup>, de  $f\ x = ax^2 + bx + c$  derivam-se duas funções mutuamente recursivas:

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

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

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

Qual é o assunto desta questão, então? Considerem fórmula que dá a série de Taylor da função coseno:

$$\cos x = \sum_{i=0}^{\infty} \frac{(-1)^i}{(2i)!} x^{2i}$$

Pretende-se o ciclo-for que implementa a função  $\cos' x\ n$  que dá o valor dessa série tomando  $i$  até  $n$  inclusivé:

$$\cos' x = \dots \text{for loop init where } \dots$$

**Sugestão:** Começar por estudar muito bem o processo de cálculo dado no anexo B para o problema (semelhante) da função exponencial.

**Propriedade QuickCheck 4** Testes de que  $\cos' x$  calcula bem o coseno de  $\pi$  e o coseno de  $\pi / 2$ :

$$\begin{aligned}prop\_cos1\ n = n \geq 10 &\Rightarrow abs\ (\cos\ \pi - \cos'\ \pi\ n) < 0.001 \\ prop\_cos2\ n = n \geq 10 &\Rightarrow abs\ (\cos\ (\pi / 2) - \cos'\ (\pi / 2)\ n) < 0.001\end{aligned}$$

<sup>3</sup>Lei (3.94) em [?], página 98.

<sup>4</sup>Podem obviamente usar-se outros símbolos, mas numa primeiraleitura dá jeito usarem-se tais nomes.

<sup>5</sup>Secção 3.17 de [?].

**Valorização** Transliterar *cos'* para a linguagem C; compilar e testar o código. Conseguia, por intuição apenas, chegar a esta função?

## Problema 4

Pretende-se nesta questão desenvolver uma biblioteca de funções para manipular *sistemas de ficheiros* genéricos. Um sistema de ficheiros será visto como uma associação de *nomes* a ficheiros ou *directorias*. Estas últimas serão vistas como sub-sistemas de ficheiros e assim recursivamente. Assumindo que *a* é o tipo dos identificadores dos ficheiros e directorias, e que *b* é o tipo do conteúdo dos ficheiros, podemos definir um tipo indutivo de dados para representar sistemas de ficheiros da seguinte forma:

```
data FS a b = FS [(a, Node a b)] deriving (Eq, Show)
data Node a b = File b | Dir (FS a b) deriving (Eq, Show)
```

Um caminho (*path*) neste sistema de ficheiros pode ser representado pelo seguinte tipo de dados:

```
type Path a = [a]
```

Assumindo estes tipos de dados, o seguinte termo

```
FS [("f1", File "01a"),
    ("d1", Dir (FS [("f2", File "01e"),
                    ("f3", File "01e")
                    ]))
    ]
```

representará um sistema de ficheiros em cuja raiz temos um ficheiro chamado *f1* com conteúdo "01a" e uma directoria chamada "d1" constituída por dois ficheiros, um chamado "f2" e outro chamado "f3", ambos com conteúdo "01e". Neste caso, tanto o tipo dos identificadores como o tipo do conteúdo dos ficheiros é *String*. No caso geral, o conteúdo de um ficheiro é arbitrário: pode ser um binário, um texto, uma colecção de dados, etc.

A definição das usuais funções *inFS* e *recFS* para este tipo é a seguinte:

```
inFS = FS · map (id × inNode)
inNode = [File, Dir]
recFS f = baseFS id id f
```

Suponha que se pretende definir como um *catamorfismo* a função que conta o número de ficheiros existentes num sistema de ficheiros. Uma possível definição para esta função seria:

```
conta :: FS a b → Int
conta = cataFS (sum · map ([1, id] · π₂))
```

O que é para fazer:

1. Definir as funções *outFS*, *baseFS*, *cataFS*, *anaFS* e *hyloFS*.
2. Apresentar, no relatório, o diagrama de *cataFS*.
3. Definir as seguintes funções para manipulação de sistemas de ficheiros usando, obrigatoriamente, catamorfismos, anamorfismos ou hilomorfismos:
  - (a) Verificação da integridade do sistema de ficheiros (i.e. verificar que não existem identificadores repetidos dentro da mesma directoria).

```
check :: FS a b → Bool
```

**Propriedade QuickCheck 5** A integridade de um sistema de ficheiros não depende da ordem em que os últimos são listados na sua directoria:

```
prop_check :: FS String String → Bool
prop_check = check · (cataFS (inFS · reverse)) ≡ check
```

- (b) Recolha do conteúdo de todos os ficheiros num arquivo indexado pelo *path*.

$tar :: FS\ a\ b \rightarrow [(Path\ a, b)]$

**Propriedade QuickCheck 6** O número de ficheiros no sistema deve ser igual ao número de ficheiros listados pela função *tar*.

$prop\_tar :: FS\ String\ String \rightarrow Bool$   
 $prop\_tar = length \cdot tar \equiv conta$

- (c) Transformação de um arquivo com o conteúdo dos ficheiros indexado pelo *path* num sistema de ficheiros.

$untar :: [(Path\ a, b)] \rightarrow FS\ a\ b$

**Sugestão:** Use a função *joinDupDirs* para juntar directorias que estejam na mesma pasta e que possuam o mesmo identificador.

**Propriedade QuickCheck 7** A composição *tar* · *untar* preserva o número de ficheiros no sistema.

$prop\_untar :: [(Path\ String, String)] \rightarrow Property$   
 $prop\_untar = validPaths \Rightarrow ((length \cdot tar \cdot untar) \equiv length)$   
 $validPaths :: [(Path\ String, String)] \rightarrow Bool$   
 $validPaths = (\equiv 0) \cdot length \cdot (filter\ (\lambda(a, -) \rightarrow length\ a \equiv 0))$

- (d) Localização de todos os *paths* onde existe um determinado ficheiro.

$find :: a \rightarrow FS\ a\ b \rightarrow [Path\ a]$

**Propriedade QuickCheck 8** A composição *tar* · *untar* preserva todos os ficheiros no sistema.

$prop\_find :: String \rightarrow FS\ String\ String \rightarrow Bool$   
 $prop\_find = curry\ \$$   
 $length \cdot \widehat{find} \equiv length \cdot \widehat{find} \cdot (id \times (untar \cdot tar))$

- (e) Criação de um novo ficheiro num determinado *path*.

$new :: Path\ a \rightarrow b \rightarrow FS\ a\ b \rightarrow FS\ a\ b$

**Propriedade QuickCheck 9** A adição de um ficheiro não existente no sistema não origina ficheiros duplicados.

$prop\_new :: ((Path\ String, String), FS\ String\ String) \rightarrow Property$   
 $prop\_new = ((validPath \wedge notDup) \wedge (check \cdot \pi_2)) \Rightarrow$   
 $(checkFiles \cdot \widehat{new})\ \mathbf{where}$   
 $validPath = (\neq 0) \cdot length \cdot \pi_1 \cdot \pi_1$   
 $notDup = \neg \cdot \widehat{elem} \cdot (\pi_1 \times ((fmap\ \pi_1) \cdot tar))$

**Questão:** Supondo-se que no código acima se substitui a propriedade *checkFiles* pela propriedade mais fraca *check*, será que a propriedade *prop\_new* ainda é válida? Justifique a sua resposta.

**Propriedade QuickCheck 10** A listagem de ficheiros logo após uma adição nunca poderá ser menor que a listagem de ficheiros antes dessa mesma adição.

$prop\_new2 :: ((Path\ String, String), FS\ String\ String) \rightarrow Property$   
 $prop\_new2 = validPath \Rightarrow ((length \cdot tar \cdot \pi_2) \leq (length \cdot tar \cdot \widehat{new}))\ \mathbf{where}$   
 $validPath = (\neq 0) \cdot length \cdot \pi_1 \cdot \pi_1$

- (f) Duplicação de um ficheiro.

$cp :: Path\ a \rightarrow Path\ a \rightarrow FS\ a\ b \rightarrow FS\ a\ b$

**Propriedade QuickCheck 11** A listagem de ficheiros com um dado nome não diminui após uma duplicação.

$prop\_cp :: ((Path\ String, Path\ String), FS\ String\ String) \rightarrow Bool$   
 $prop\_cp = length \cdot tar \cdot \pi_2 \leq length \cdot tar \cdot \widehat{cp}$





Figure 3: Exemplo de um sistema de ficheiros visualizado em Graphviz.

(g) Eliminação de um ficheiro.

$rm :: Path\ a \rightarrow FS\ a\ b \rightarrow FS\ a\ b$

**Sugestão:** Construir um anamorfismo  $nav :: (Path\ a, FS\ a\ b) \rightarrow FS\ a\ b$  que navegue por um sistema de ficheiros tendo como base o *path* dado como argumento.

**Propriedade QuickCheck 12** *Remover duas vezes o mesmo ficheiro tem o mesmo efeito que o remover apenas uma vez.*

$$prop\_rm :: (Path\ String, FS\ String\ String) \rightarrow Bool$$

$$prop\_rm = \widehat{rm} \cdot \langle \pi_1, \widehat{rm} \rangle \equiv \widehat{rm}$$

**Propriedade QuickCheck 13** *Adicionar um ficheiro e de seguida remover o mesmo não origina novos ficheiros no sistema.*

$$prop\_rm2 :: ((Path\ String, String), FS\ String\ String) \rightarrow Property$$

$$prop\_rm2 = validPath \Rightarrow ((length \cdot tar \cdot \widehat{rm} \cdot \langle \pi_1 \cdot \pi_1, \widehat{new} \rangle) \leq (length \cdot tar \cdot \pi_2)) \text{ where}$$

$$validPath = (\neq 0) \cdot length \cdot \pi_1 \cdot \pi_1$$

**Valorização** Definir uma função para visualizar em Graphviz a estrutura de um sistema de ficheiros. A Figura 3, por exemplo, apresenta a estrutura de um sistema com precisamente dois ficheiros dentro de uma directoria chamada "d1".

Para realizar este exercício será necessário apenas escrever o anamorfismo

$$cFS2Exp :: (a, FS\ a\ b) \rightarrow (Exp\ ()\ a)$$

que converte a estrutura de um sistema de ficheiros numa árvore de expressões descrita em Exp.hs. A função *dotFS* depois tratará de passar a estrutura do sistema de ficheiros para o visualizador.

# Anexos

## A Como exprimir cálculos e diagramas em LaTeX/lhs2tex

Estudar o texto fonte deste trabalho para obter o efeito:<sup>6</sup>

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

Os diagramas podem ser produzidos recorrendo à *package* L<sup>A</sup>T<sub>E</sub>X *xymatrix*, por exemplo:

$$\begin{array}{ccc}
 \mathbb{N}_0 & \xleftarrow{\text{in}} & 1 + \mathbb{N}_0 \\
 \downarrow \langle g \rangle & & \downarrow id + \langle g \rangle \\
 B & \xleftarrow{g} & 1 + B
 \end{array}$$

## B Programação dinâmica por recursividade múltipla

Neste anexo dão-se os detalhes da resolução do Exercício 3.30 dos apontamentos da disciplina<sup>7</sup>, onde se pretende implementar um ciclo que implemente o cálculo da aproximação até  $i = n$  da função exponencial  $\exp x = e^x$  via série de Taylor:

$$\exp x = \sum_{i=0}^{\infty} \frac{x^i}{i!} \quad (1)$$

Seja  $e\ x\ n = \sum_{i=0}^n \frac{x^i}{i!}$  a função que dá essa aproximação. É fácil de ver que  $e\ x\ 0 = 1$  e que  $e\ x\ (n+1) = e\ x\ n + \frac{x^{n+1}}{(n+1)!}$ . Se definirmos  $h\ x\ n = \frac{x^{n+1}}{(n+1)!}$  teremos  $e\ x$  e  $h\ x$  em recursividade mútua. Se repetirmos o processo para  $h\ x\ n$  etc obteremos no total três funções nessa mesma situação:

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

Segundo a *regra de algibeira* descrita na página 3 deste enunciado, ter-se-á, de imediato:

$$\begin{aligned}
 e'\ x &= prj \cdot \text{for loop init where} \\
 init &= (1, x, 2) \\
 loop\ (e, h, s) &= (h + e, x / s * h, 1 + s) \\
 prj\ (e, h, s) &= e
 \end{aligned}$$

<sup>6</sup>Exemplos tirados de [?].

<sup>7</sup>Cf. [?], página 102.

## C Código fornecido

### Problema 1

Tipos:

```
data Expr = Num Int
          | Bop Expr Op Expr deriving (Eq, Show)
data Op = Op String deriving (Eq, Show)
type Codigo = [String]
```

Functor de base:

```
baseExpr f g = id + (f × (g × g))
```

Instâncias:

```
instance Read Expr where
  readsPrec _ = readExp
```

Read para Exp's:

```
readOp :: String → [(Op, String)]
readOp input = do
  (x, y) ← lex input
  return ((Op x), y)

readNum :: ReadS Expr
readNum = (map (λ(x, y) → ((Num x), y))) · reads

readBinOp :: ReadS Expr
readBinOp = (map (λ((x, (y, z)), t) → ((Bop x y z), t))) ·
  ((readNum 'ou' (pcurvos readExp))
   'depois' (readOp 'depois' readExp))

readExp :: ReadS Expr
readExp = readBinOp 'ou' (
  readNum 'ou' (
    pcurvos readExp))
```

Combinadores:

```
depois :: (ReadS a) → (ReadS b) → ReadS (a, b)
depois _ _ [] = []
depois r1 r2 input = [((x, y), i2) | (x, i1) ← r1 input,
  (y, i2) ← r2 i1]

readSeq :: (ReadS a) → ReadS [a]
readSeq r input
  = case (r input) of
    [] → [([], input)]
    l → concat (map continua l)
    where continua (a, i) = map (c a) (readSeq r i)
      c x (xs, i) = ((x : xs), i)

ou :: (ReadS a) → (ReadS a) → ReadS a
ou r1 r2 input = (r1 input) ++ (r2 input)

senao :: (ReadS a) → (ReadS a) → ReadS a
senao r1 r2 input = case (r1 input) of
  [] → r2 input
  l → l

readConst :: String → ReadS String
readConst c = (filter ((≡ c) · π1)) · lex

pcurvos = parenthesis ' ( ' ' ) '
```

```

prectos = parenthesis ' [ ' ' ] '
chavetas = parenthesis ' { ' ' } '
parenthesis :: Char → Char → (ReadS a) → ReadS a
parenthesis _ _ _ [] = []
parenthesis ap pa r input
= do
  ((_, (x, _)), c) ← ((readConst [ap]) 'depois' (
    r 'depois' (
      readConst [pa]))) input
  return (x, c)

```

## Problema 2

Tipos:

```

type Fig = [(Origem, Caixa)]
type Origem = (Float, Float)

```

“Helpers”:

```

col_blue = G.azure
col_green = darkgreen
col_red = G.red
darkgreen = G.dark (G.dark G.green)

```

Exemplos:

```

ex1Caixas = G.display (G.InWindow "Problema 4" (400,400) (40,40)) G.white $
  crCaixa (0,0) 200 200 "Caixa azul" col_blue
ex2Caixas = G.display (G.InWindow "Problema 4" (400,400) (40,40)) G.white $
  caixasAndOrigin2Pict ((Comp Hb bbox gbox), (0.0,0.0)) where
    bbox = Unid ((100,200), ("A", col_blue))
    gbox = Unid ((50,50), ("B", col_green))
ex3Caixas = G.display (G.InWindow "Problema 4" (400,400) (40,40)) G.white mtest where
  mtest = caixasAndOrigin2Pict $ (Comp Hb (Comp Ve bot top) (Comp Ve gbox2 ybox2), (0.0,0.0))
  bbox1 = Unid ((100,200), ("A", col_blue))
  bbox2 = Unid ((150,200), ("E", col_blue))
  gbox1 = Unid ((50,50), ("B", col_green))
  gbox2 = Unid ((100,300), ("F", col_green))
  rbox1 = Unid ((300,50), ("C", G.red))
  rbox2 = Unid ((200,100), ("G", G.red))
  wbox1 = Unid ((450,200), ("", G.white))
  ybox1 = Unid ((100,200), ("D", G.yellow))
  ybox2 = Unid ((100,300), ("H", G.yellow))
  bot = Comp Hb wbox1 bbox2
  top = (Comp Ve (Comp Hb bbox1 gbox1) (Comp Hb rbox1 (Comp H ybox1 rbox2)))

```

A seguinte função cria uma caixa a partir dos seguintes parâmetros: origem, largura, altura, etiqueta e cor de preenchimento.

```

crCaixa :: Origem → Float → Float → String → G.Color → G.Picture
crCaixa (x,y) w h l c = G.Translate (x + (w / 2)) (y + (h / 2)) $ G.pictures [caixa, etiqueta] where
  caixa = G.color c (G.rectangleSolid w h)
  etiqueta = G.translate calc_trans_x calc_trans_y $
    G.Scale calc_scale calc_scale $ G.color G.black $ G.Text l
  calc_trans_x = -((fromIntegral (length l)) * calc_scale) / 2 * base_shift_x
  calc_trans_y = (-calc_scale / 2) * base_shift_y
  calc_scale = bscale * (min h w)

```

```

bscale = 1 / 700
base_shift_y = 100
base_shift_x = 64

```

Função para visualizar resultados gráficos:

```
display = G.display (G.InWindow "Problema 4" (400,400) (40,40)) G.white
```

## Problema 4

Funções para gestão de sistemas de ficheiros:

```

concatFS = inFS ·  $\widehat{(\text{++})}$  · (outFS × outFS)
mkdir (x, y) = FS [(x, Dir y)]
mkfile (x, y) = FS [(x, File y)]
joinDupDirs :: (Eq a) ⇒ (FS a b) → (FS a b)
joinDupDirs = anaFS (prepOut · (id × proc) · prepIn) where
  prepIn = (id × (map (id × outFS))) · sls · (map distr) · outFS
  prepOut = (map undistr) ·  $\widehat{(\text{++})}$  · ((map i1) × (map i2)) · (id × (map (id × inFS)))
  proc = concat · (map joinDup) · groupByName
  sls = ⟨lefts, rights⟩
joinDup :: [(a, [b])] → [(a, [b])]
joinDup = cataList [nil, g] where g = return · ⟨π1 · π1, concat · (map π2) ·  $\widehat{(\text{:})}$ ⟩
createFSfromFile :: (Path a, b) → (FS a b)
createFSfromFile ([a], b) = mkfile (a, b)
createFSfromFile (a : as, b) = mkdir (a, createFSfromFile (as, b))

```

Funções auxiliares:

```

checkFiles :: (Eq a) ⇒ FS a b → Bool
checkFiles = cataFS ( $\widehat{(\text{&})}$  · ⟨f, g⟩) where
  f = nr · (fmap π1) · lefts · (fmap distr)
  g = and · rights · (fmap π2)
groupByName :: (Eq a) ⇒ [(a, [b])] → [[(a, [b])]]
groupByName = (groupBy (curry p)) where
  p =  $\widehat{(\text{≡})}$  · (π1 × π1)
filterPath :: (Eq a) ⇒ Path a → [(Path a, b)] → [(Path a, b)]
filterPath = filter · (λp → λ(a, b) → p ≡ a)

```

Dados para testes:

- Sistema de ficheiros vazio:

```
efs = FS []
```

- Nível 0

```

f1 = FS [("f1", File "hello world")]
f2 = FS [("f2", File "more content")]
f00 = concatFS (f1, f2)
f01 = concatFS (f1, mkdir ("d1", efs))
f02 = mkdir ("d1", efs)

```

- Nível 1

```

f10 = mkdir ("d1", f00)
f11 = concatFS (mkdir ("d1", f00), mkdir ("d2", f00))
f12 = concatFS (mkdir ("d1", f00), mkdir ("d2", f01))
f13 = concatFS (mkdir ("d1", f00), mkdir ("d2", efs))

```

- Nível 2

```
f20 = mkdir ("d1", f10)
f21 = mkdir ("d1", f11)
f22 = mkdir ("d1", f12)
f23 = mkdir ("d1", f13)
f24 = concatFS (mkdir ("d1", f10), mkdir ("d2", f12))
```

- Sistemas de ficheiros inválidos:

```
ifs0 = concatFS (f1, f1)
ifs1 = concatFS (f1, mkdir ("f1", efs))
ifs2 = mkdir ("d1", ifs0)
ifs3 = mkdir ("d1", ifs1)
ifs4 = concatFS (mkdir ("d1", ifs1), mkdir ("d2", f12))
ifs5 = concatFS (mkdir ("d1", f1), mkdir ("d1", f2))
ifs6 = mkdir ("d1", ifs5)
ifs7 = concatFS (mkdir ("d1", f02), mkdir ("d1", f02))
```

Visualização em **Graphviz**:

```
dotFS :: FS String b → IO ExitCode
dotFS = dotpict · bmap "_" id · (cFS2Exp "root")
```

## Outras funções auxiliares

Lógicas:

```
infixr 0 ⇒
(⇒) :: (Testable prop) ⇒ (a → Bool) → (a → prop) → a → Property
p ⇒ f = λa → p a ⇒ f a

infixr 0 ⇔
(⇔) :: (a → Bool) → (a → Bool) → a → Property
p ⇔ f = λa → (p a ⇒ property (f a)) .&&. (f a ⇒ property (p a))

infixr 4 ≡
(≡) :: Eq b ⇒ (a → b) → (a → b) → (a → Bool)
f ≡ g = λa → f a ≡ g a

infixr 4 ≤
(≤) :: Ord b ⇒ (a → b) → (a → b) → (a → Bool)
f ≤ g = λa → f a ≤ g a

infixr 4 ∧
(∧) :: (a → Bool) → (a → Bool) → (a → Bool)
f ∧ g = λa → (f a) ∧ (g a)
```

Compilação e execução dentro do interpretador:<sup>8</sup>

```
run = do { system "ghc cp1819t"; system "./cp1819t" }
```

## D Soluções dos alunos

Os alunos devem colocar neste anexo as suas soluções aos 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 e/ou outras funções auxiliares que sejam necessárias.

<sup>8</sup>Pode ser útil em testes envolvendo **Gloss**. Nesse caso, o teste em causa deve fazer parte de uma função *main*.

## Problema 1

### In e Out

$inExpr :: Int + (Op, (Expr, Expr)) \rightarrow Expr$

$inExpr = [Num, (\cdot, (flip Bop))]$

$outExpr :: Expr \rightarrow Int + (Op, (Expr, Expr))$

$outExpr (Num a) = i_1 a$

$outExpr (Bop a x b) = i_2 (x, (a, b))$

### Catamorfismo, anamorfismo, hilomorfismo e Functor de Expr

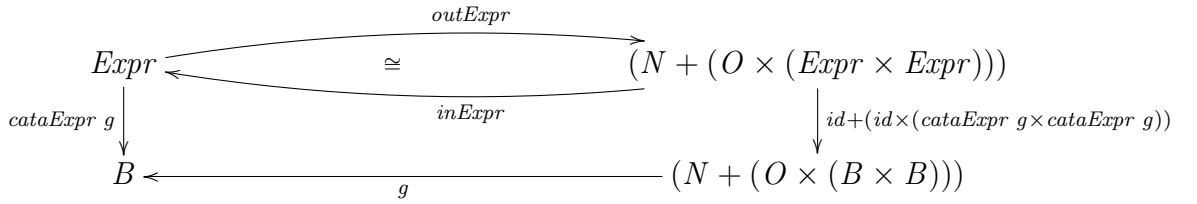


Diagrama do Catamorfismo do tipo Expr

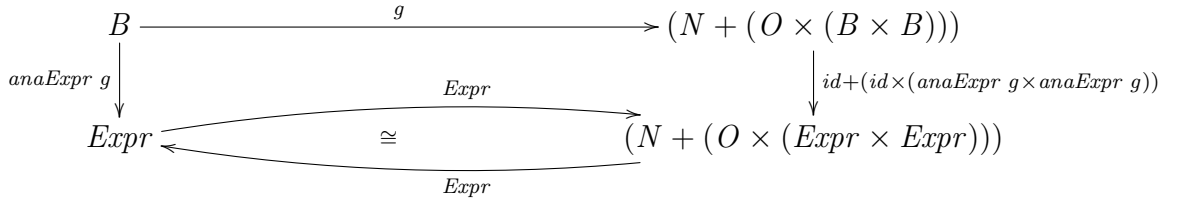


Diagrama do Anamorfismo do tipo Expr

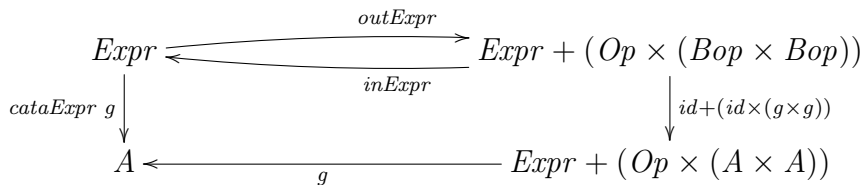
$recExpr\ f = baseExpr\ id\ f$

$cataExpr\ g = g \cdot (recExpr\ (cataExpr\ g)) \cdot outExpr$

$anaExpr\ g = inExpr \cdot (recExpr\ (anaExpr\ g)) \cdot g$

$hyloExpr\ h\ g = cataExpr\ h \cdot anaExpr\ g$

### Calcula



$calcula :: Expr \rightarrow Int$

$calcula = cataExpr\ [g1, g2]$

**where**  $g1\ a = a$

$$g2 \ (Op \ o, (x, y)) \mid (o \equiv "+") = x + y \\ \mid otherwise = x * y$$

### Show'

Função que usamos para representar o Expr como uma string. Para isso foi usado um catamorfismo (either g1 g2) sendo que o g1 corresponde ao caso simples (Num 1 = 1) e o g2 trata de gerar os parenteses de acordo com o tamanho da expressão aritmética.

$$show' :: Expr \rightarrow String \\ show' = cataExpr \ [g1, g2] \\ \textbf{where} \ g1 \ a = "(" \ ++ \ show \ (a) \ ++ \ ")" \\ g2 \ (Op \ o, (x, y)) \mid (\text{length} \ (x) > 1) = "(" \ ++ \ x \ ++ \ ")" \ ++ \ o \ ++ \ y \\ \mid (\text{length} \ (y) > 1) = x \ ++ \ o \ ++ \ "(" \ ++ \ y \ ++ \ ")" \\ \mid otherwise = x \ ++ \ o \ ++ \ y$$

### Compile

$$compile :: String \rightarrow Codigo \\ compile = cataExpr \ [convInt, convExp] \cdot \pi_1 \cdot head \cdot readExp \\ convInt :: Int \rightarrow Codigo \\ convInt \ i = ["PUSH" \ ++ \ show \ (i)] \\ convExp :: (Op, (Codigo, Codigo)) \rightarrow Codigo \\ convExp \ (Op \ o, (x, y)) \mid (o \equiv "\cdot") = x \ ++ \ y \ ++ \ ["MULL"] \\ \mid (o \equiv "+") = x \ ++ \ y \ ++ \ ["ADD"] \\ \mid otherwise = x \ ++ \ y \ ++ \ ["SUB"]$$

## Problema 2

### Diagrama L2D

$$\begin{array}{ccc} L2D & \begin{array}{c} \xrightarrow{outL2D} \\ \xleftarrow{inL2D} \end{array} & Caixa + (Tipo \times (L2D \times L2D)) \\ \begin{array}{c} \downarrow cataL2D \ g \\ A \end{array} & & \begin{array}{c} \downarrow id + (id \times (cataL2D \ (g) \times cataL2D \ (g))) \\ Caixa + (Tipo \times (A \times A)) \end{array} \\ & \xleftarrow{g} & \end{array}$$

Através do Diagrama representado anteriormente, e das definições de cata, ana e hylemorfismos, verificamos que:

$$inL2D :: a + (b, (X \ a \ b, X \ a \ b)) \rightarrow X \ a \ b \\ inL2D = [Unid, \widehat{\cdot} \cdot Comp] \\ outL2D :: X \ a \ b \rightarrow a + (b, (X \ a \ b, X \ a \ b)) \\ outL2D \ (Unid \ a) = i_1 \ a \\ outL2D \ (Comp \ b \ (x) \ (y)) = i_2 \ (b, (x, y)) \\ baseL2D \ f \ g \ h = f + (g \times (h \times h)) \\ recL2D \ f = baseL2D \ id \ id \ f \\ cataL2D \ g = g \cdot (recL2D \ (cataL2D \ g)) \cdot outL2D \\ anaL2D \ g = inL2D \cdot (recL2D \ (anaL2D \ g)) \cdot g \\ hylol2D \ h \ g = cataL2D \ h \cdot anaL2D \ g$$

**Resolução da Função collectLeafs** Para começar, foi necessário tipar a função *collectLeafs*, com o tipo definido em baixo. Esta função retorna todas as folhas de um



elemento do tipo  $X$  a  $b$ , tendo como output um array com a informação das folhas, sendo esta do tipo  $a$ .

Para tal, é necessário colocar todos os elementos nas folhas dentro da lista, e concatenar os outputs que vão sendo gerados. De seguida é apresando o diagram correspondente ao catamorfismo que permitiu definir a função apresentada.

$$\begin{array}{ccc}
 L2D & \begin{array}{c} \xrightarrow{\text{out}} \\ \xleftarrow{\text{in}} \end{array} & Caixa + (Tipo \times (L2D \times L2D)) \\
 \downarrow \text{cataL2D } g & & \downarrow id + (id \times (cataL2D(g) \times cataL2D(g))) \\
 [A] & \xleftarrow{g=[singl, conc \cdot \pi_2]} & Caixa + (Tipo \times ([A] \times [A]))
 \end{array}$$

$collectLeafs :: X \ a \ b \rightarrow [a]$   
 $collectLeafs = cataL2D [singl, conc \cdot \pi_2]$

**Resolução da função dimen** A função *dimen* tem como objectivo calcular as dimensões de um elemento do tipo  $X$  Caixa Tipo, com base na suas caixas e o posicionamento relativo das últimas.

Para definir a função *dimen*, foi necessário criar duas funções auxiliares:

- **toFloat**: Que converte as coordenadas (Int,Int) em (Float,Float)
- **compareDimen**: Que a partir do tipo fornecido calcula as dimensões com base no posicionamento das caixas.

Aplicando um catamorfismo L2D, com o gene que aplica a função *toFloat* ao elemento do tipo *Caixa*, e a função aos elementos do tipo  $(Tipo, (origem, ponto))$ , retorna a dimensão total da figura.

$toFloat :: (Int, Int) \rightarrow (Float, Float)$   
 $toFloat(a, b) = (fromInteger(toInteger a), fromInteger(toInteger b))$   
 $compareDimen :: String \rightarrow (Float, Float) \rightarrow (Float, Float) \rightarrow (Float, Float)$   
 $compareDimen \ t \ (a, b) \ (x, y) \mid t \equiv "V" \vee t \equiv "Vd" \vee t \equiv "Ve" = \text{if } (a \geq x) \text{ then } (a, b + y) \text{ else } (a + x, b)$   
 $\mid t \equiv "H" \vee t \equiv "Hb" \vee t \equiv "Ht" = \text{if } (b \geq y) \text{ then } (a + x, b) \text{ else } (a + x, y)$   
 $dimen :: X \ Caixa \ Tipo \rightarrow (Float, Float)$   
 $dimen = cataL2D [g1, g2]$   
**where**  $g1 \ a = toFloat(\pi_1 \ a)$   
 $g2 \ (V, (b, c)) = compareDimen "V" \ b \ c$   
 $g2 \ (Vd, (b, c)) = compareDimen "Vd" \ b \ c$   
 $g2 \ (Ve, (b, c)) = compareDimen "Ve" \ b \ c$   
 $g2 \ (H, (b, c)) = compareDimen "H" \ b \ c$   
 $g2 \ (Hb, (b, c)) = compareDimen "Hb" \ b \ c$   
 $g2 \ (Ht, (b, c)) = compareDimen "Ht" \ b \ c$

## Resolução da função calcOrigins

$calculaOrigens :: ((Tipo, (X \ Caixa \ Tipo, X \ Caixa \ Tipo)), Origem) \rightarrow (((), ((X \ Caixa \ Tipo, Origem)))$   
 $calculaOrigens((t, (t1, t2)), o) = (((), ((t1, o), (t2, o2))))$   
**where**  
 $o2 = calc \ t \ o \ (dimen \ t1)$   
 $calcOrigens :: ((X \ Caixa \ Tipo), Origem) \rightarrow X \ (Caixa, Origem) \ ()$

```

calcOrigins = anaL2D ((id + calculaOrigens) · distl · (outL2D × id))
calc :: Tipo → Origem → (Float, Float) → Origem
calc Hb (a, b) (x, y) = (x, b)
calc Ht (a, b) (x, y) = (a, b - y)
calc Vd (a, b) (x, y) = (a - x, y)
calc Ve (a, b) (x, y) = (a, b + y)
calc V (a, b) (x, y) = ((a / 2) - (x / 2), b)
calc H (a, b) (x, y) = (x, (b + y) / 2)

```

### Resolução da função agrup<sub>Caixa</sub>sto

$$\begin{array}{ccc}
X \text{ (Caixa, Origem) } () & \xrightleftharpoons[\text{in}]{\text{out}} & (\text{Caixa, Origem}) + (() \times (L2D \times L2D)) \\
\downarrow \text{cataL2D } g & & \downarrow \text{id} + (\text{id} \times (\text{cataL2D } (g) \times \text{cataL2D } (g))) \\
[(\text{Origem, Caixa})] & \xleftarrow{g = [\text{singl} \cdot \text{swap}, \text{conc} \cdot \pi_2]} & (\text{Caixa, Origem}) + (() \times [(\text{Origem, Caixa})] \times [(\text{Origem, Caixa})])
\end{array}$$

```

agrup_caixas :: X (Caixa, Origem) () → Fig
agrup_caixas = cataL2D [singl · swap, conc · π2]

```

**Resolução da função caixasAndOrigin2Pict** Esta função foi definida recorrendo a algumas funções previamente definidas. A  $\text{agrup}_{\text{caixas}}$  e  $\text{calcOrigins}$ . Primeiro foi aplicada a função  $\text{cataL2D}$  a  $\text{calcOrigins}$  :: ((X Caixa Tipo), Origem) -> X (Caixa, Origem) () Deseguida foi aplicada a função  $\text{agrup}_{\text{caixas}}$ .

### Problema 3

Solução:

```

cos' x = prj · for loop init where
  loop (c, h, s, f) = (h + c, (-(x ↑ 2) / s * h), f + s, 8 + f)
  init = (1, ((-0.5) * x ↑ 2), 12, 18)
  prj (c, h, s, f) = c

```

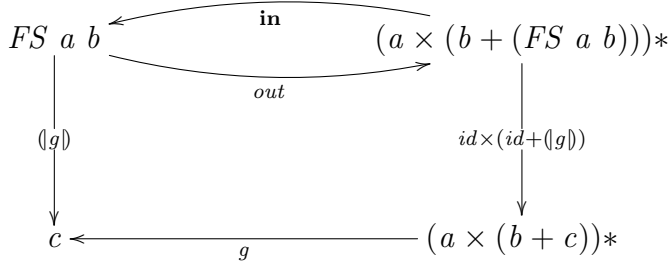
### Problema 4

Triologia “ana-cata-hilo”:

```

outFS (FS l) = map (id × outNode) l
outNode :: Node a b → b + (FS a b)
outNode (File b) = i1 b
outNode (Dir (FS l)) = i2 (FS l)
baseFS f g h = map (f × (g + h))
cataFS :: ([ (a, b + c) ] → c) → FS a b → c
cataFS g = g · recFS (cataFS g) · outFS
anaFS :: (c → [ (a, b + c) ]) → c → FS a b
anaFS f = inFS · recFS (anaFS f) · f
hyloFS g h = cataFS g · (anaFS h)

```



Outras funções pedidas:

$check :: (Eq\ a) \Rightarrow FS\ a\ b \rightarrow Bool$

$check = cataFS\ ((\widehat{\wedge}) \cdot (nr \times and) \cdot \langle fmap\ \pi_1, rights \cdot (fmap\ \pi_2) \rangle)$

$tar :: FS\ a\ b \rightarrow [(Path\ a, b)]$

$tar = cataFS\ (concat \cdot fmap\ ([singl \cdot (singl \times id), (fmap\ (\widehat{\times id}) \cdot (:)) \cdot distr])$

$untar :: (Eq\ a) \Rightarrow [(Path\ a, b)] \rightarrow FS\ a\ b$

$untar = anaFS\ (map\ h)$

**where**

$p = cond\ (null \cdot tail \cdot \pi_1)\ (i_1 \cdot \pi_2)\ (i_2 \cdot singl \cdot (tail \times id))$

$h = \langle head \cdot \pi_1, p \rangle$

$find :: (Eq\ a) \Rightarrow a \rightarrow FS\ a\ b \rightarrow [Path\ a]$

$find\ a = cataList\ [nil, cond\ ((\equiv a) \cdot last \cdot \pi_1 \cdot \pi_1)\ (cons \cdot (\pi_1 \times id))\ \pi_2] \cdot tar$

$new :: (Eq\ a) \Rightarrow Path\ a \rightarrow b \rightarrow FS\ a\ b \rightarrow FS\ a\ b$

$new\ p\ file\ fs = cond\ null\ \underline{fs}\ ((\cdot anaFS\ f)\ ((p, fs), file))\ \$\ p$

**where**

$f :: Eq\ a \Rightarrow ((Path\ a, FS\ a\ b), b) \rightarrow [(a, b + ((Path\ a, FS\ a\ b), b))]$

$f = cond\ (null \cdot tail \cdot \pi_1 \cdot \pi_1)\ (cons \cdot \langle head \cdot \pi_{11}, (\lambda(l, c) \rightarrow fmap\ [id_1, \lambda(t, y) \rightarrow (t, i_2$

$g :: Eq\ a \Rightarrow ((Path\ a, FS\ a\ b), b) \rightarrow [(a, b + ((Path\ a, FS\ a\ b), b))]$

$g\ al@(((x : xs), fs0)), file0) = \text{if}\ (null \cdot filter\ ((\equiv x) \cdot \pi_1) \cdot outFS\ \$\ fs0)\ \text{then}\ (((x, i_2$

$go :: Eq\ a \Rightarrow ((Path\ a, FS\ a\ b), b) \rightarrow [(a, b + ((Path\ a, FS\ a\ b), b))]$

$go\ (((x : xs), fs0)), file0) = fmap\ ([id_1, \lambda(a, b) \rightarrow \text{if}\ (a \equiv x)\ \text{then}\ (a, i_2\ ((xs, b), file0$

$cp :: (Eq\ a) \Rightarrow Path\ a \rightarrow Path\ a \rightarrow FS\ a\ b \rightarrow FS\ a\ b$

$cp\ from\ to = (cond\ (null \cdot \pi_1)\ \pi_2\ (\lambda([p], s) \rightarrow new\ to\ p\ s)) \cdot \langle (cataList\ [nil, cond\ ((\equiv from)$

$rm :: (Eq\ a) \Rightarrow (Path\ a) \rightarrow (FS\ a\ b) \rightarrow FS\ a\ b$

$rm\ p\ fs = cond\ (null \cdot \pi_1)\ \pi_2\ (anaFS\ f)\ \$\ (p, fs)$

**where**

$f :: Eq\ a \Rightarrow (Path\ a, FS\ a\ b) \rightarrow [(a, b + (Path\ a, FS\ a\ b))]$

$f = cond\ (null \cdot tail \cdot \pi_1)\ (\lambda((x : []), fs0) \rightarrow m\ ([x], fs0)\ (\neq x))\ (\lambda((x : xs), fs0) \rightarrow m\ ($

$m :: (Path\ a, FS\ a\ b) \rightarrow (a \rightarrow Bool) \rightarrow [(a, b + (Path\ a, FS\ a\ b))]$

$m\ (p, f)\ b = fmap\ (id \times (id + \langle p, id \rangle)) \cdot filter\ (b \cdot \pi_1) \cdot outFS\ \$\ f$

$auxJoin :: [(a, b + c), d] \rightarrow [(a, b + (d, c))]$

$auxJoin = \perp$

$cFS2Exp :: a \rightarrow FS\ a\ b \rightarrow (Exp\ ()\ a)$

$cFS2Exp\ r\ fs = anaExp\ f\ \$\ (r, fs)$

**where**

$f :: (a, FS\ a\ b) \rightarrow () + (a, [(a, FS\ a\ b)])$

$f = cond\ (null \cdot outFS \cdot \pi_2)\ (i_1 \cdot (!))\ (i_2 \cdot (id \times (fmap\ (filtro \cdot distr) \cdot outFS)))$

$filtro :: (a, b) + (a, FS\ a\ b) \rightarrow (a, FS\ a\ b)$

$filtro = [\lambda(a, \_) \rightarrow (a, FS\ [(a, Dir\ (FS\ []))]), id]$