

# Cálculo de Programas

## Trabalho Prático

### MiEI+LCC — 2018/19

Departamento de Informática  
Universidade do Minho

Junho de 2019

Grupo nr.	16
a83996	Filipe Fernandes
a84313	Pedro Fernandes
a84930	Rui Reis

## 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-processador que faz “pretty printing” de código Haskell em **LATeX** 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 sufixo ‘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 ?? 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 ?? 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 ou hilo}-morfismo a função seguinte:

- $\text{calcula} :: \text{Expr} \rightarrow \text{Int}$  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 =  $\widehat{\overline{Bop}} \cdot (\text{swap} \times \text{id}) \cdot \text{assocl} \cdot (\text{id} \times \text{swap})$ 
```

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

- $\text{compile} :: \text{String} \rightarrow \text{Codigo}$  - 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>
```

- $\text{show}' :: \text{Expr} \rightarrow \text{String}$  - 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 =  $\pi_1 \cdot \text{head} \cdot \text{readExp} \cdot \text{show}' \equiv \text{id}$ 
```

**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.

Expreia 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

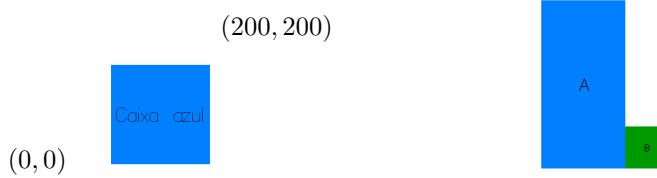


Figura 1: Caixa simples e caixa composta.

```
data X a b = Unid a | 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 ??.

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 | Vd | Ve | H | Ht | 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-árvore. Por exemplo, a frase

```
ex2 = Comp Hb (Unid ((100,200), ("A", col_blue)))
          (Unid ((50,50), ("B", col_green)))
```

deverá corresponder à imagem da direita da figura ?? . 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 ??.

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.

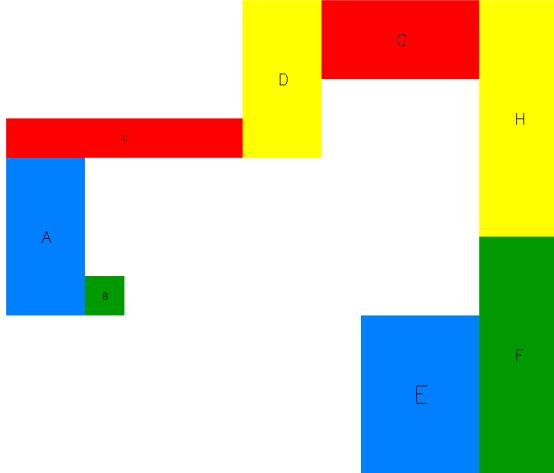


Figura 2: *Layout* feito de várias caixas coloridas.

```
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) → 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) () → 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 → Float → Float → String → G.Color → 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 → 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) → 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  $\mathbb{N}_0$ . Segundo 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 ?? 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  $\pi / 2$ :

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

<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 inductivo 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 "Olá"),
    ("d1", Dir (FS [("f2", File "Olé"),
                    ("f3", File "Olé")
                   ]))
]
```

representará um sistema de ficheiros em cuja raiz temos um ficheiro chamado *f1* com conteúdo "Olá" e uma directoria chamada "d1" constituída por dois ficheiros, um chamado "f2" e outro chamado "f3", ambos com conteúdo "Olé". 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 ([l, 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 identificados 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.*

$$\begin{aligned} prop\_tar &:: FS\ String\ String \rightarrow Bool \\ prop\_tar &= length \cdot tar \equiv conta \end{aligned}$$

- (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.*

$$\begin{aligned} prop\_untar &:: [(Path\ String, String)] \rightarrow Property \\ prop\_untar &= validPaths \Rightarrow ((length \cdot tar \cdot untar) \equiv length) \\ validPaths &:: [(Path\ String, String)] \rightarrow Bool \\ validPaths &= (\equiv 0) \cdot length \cdot (filter (\lambda(a, _) \rightarrow length a \equiv 0)) \end{aligned}$$

- (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.*

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

- (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.*

$$\begin{aligned} prop\_new &:: ((Path\ String, String), FS\ String\ String) \rightarrow Property \\ prop\_new &= ((validPath \wedge notDup) \wedge (check \cdot \pi_2)) \Rightarrow \\ (checkFiles \cdot \widehat{\widehat{new}}) \text{ where} \\ validPath &= (\not\equiv 0) \cdot length \cdot \pi_1 \cdot \pi_1 \\ notDup &= \neg \cdot \widehat{elem} \cdot (\pi_1 \times ((fmap\ \pi_1) \cdot tar)) \end{aligned}$$

**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.*

$$\begin{aligned} prop\_new2 &:: ((Path\ String, String), FS\ String\ String) \rightarrow Property \\ prop\_new2 &= validPath \Rightarrow ((length \cdot tar \cdot \pi_2) \leqslant (length \cdot tar \cdot \widehat{\widehat{new}})) \text{ where} \\ validPath &= (\not\equiv 0) \cdot length \cdot \pi_1 \cdot \pi_1 \end{aligned}$$

- (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.*

$$\begin{aligned} prop\_cp &:: ((Path\ String, Path\ String), FS\ String\ String) \rightarrow Bool \\ prop\_cp &= length \cdot tar \cdot \pi_2 \leqslant length \cdot tar \cdot \widehat{\widehat{cp}} \end{aligned}$$

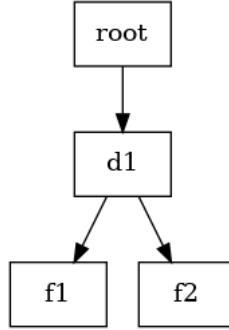


Figura 3: Exemplo de um sistema de ficheiros visualizado em [Graphviz](#).

(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.*

$$\begin{aligned} prop\_rm :: & (Path\ String, FS\ String\ String) \rightarrow Bool \\ prop\_rm = & \widehat{rm} \cdot \langle \pi_1, \widehat{rm} \rangle \equiv \widehat{rm} \end{aligned}$$

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

$$\begin{aligned} prop\_rm2 :: & ((Path\ String, String), FS\ String\ String) \rightarrow Property \\ prop\_rm2 = & validPath \Rightarrow ((length \cdot tar \cdot \widehat{rm} \cdot \langle \pi_1 \cdot \pi_1, \widehat{\text{new}} \rangle) \\ & \leqslant (length \cdot tar \cdot \pi_2)) \text{ where} \\ validPath = & (\neq 0) \cdot length \cdot \pi_1 \cdot \pi_1 \end{aligned}$$

**Valorização** Definir uma função para visualizar em [Graphviz](#) a estrutura de um sistema de ficheiros. A Figura ??, por exemplo, apresenta a estrutura de um sistema com precisamente dois ficheiros dentro de uma directória 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.

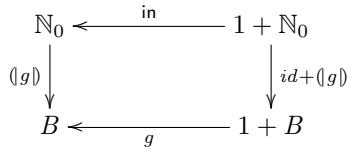
# Anexos

## 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:



## 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:

$$\exp x = \sum_{i=0}^{\infty} \frac{x^i}{i!} \tag{1}$$

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 ?? 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:

$$\text{baseExpr } f \ g = id + (f \times (g \times g))$$

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 = parentesis '()' '
```

```

precos = parentesis '[' ']'
chavetas = parentesis '{' '}'
parentesis :: Char → Char → (ReadS a) → ReadS a
parentesis _ _ _ [] = []
parentesis 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
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 côr 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{+}) · (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{+}) · ((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) · (:)⟩
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{Λ}) · ⟨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{≡}) · (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

De forma a definir a função *in* que constitui este tipo, chegou-se a conclusão de que o tipo para esta função deve partir de um *Either* sendo que, segundo o próprio tipo *Expr*, muito semelhante à árvore do tipo *FTree*, possui operações como nós e números sobre os quais são efetuadas ditas operações, como folhas. Daí que qualquer estrutura representativa de uma *Expr* terá de ser, ou um número (folha), ou uma operação com as duas expressões sobre as quais produz efeito.

Daí que a associação é então direta, sendo que

$$inExpr = [Num, f\ Bop]$$

Sendo que *f* corresponde a uma função capaz de atribuir de forma adequada, os argumentos ao construtor *Bop*, tal como se segue.

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

$$inExpr = [Num, \lambda(x, (y, z)) \rightarrow Bop\ y\ x\ z]$$

Quase que de forma análoga, *outExpr* efetua uma correspondência unívoca entre os diferentes construtores de *Expr*. Atribuindo os nós e folhas às correspondentes projeções no construtor *Either*.

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

$$outExpr (Num\ a) = i_1\ a$$

$$outExpr (Bop\ e_1\ o\ e_2) = i_2\ (o, (e_1, e_2))$$

### D.0.1 {Cata, ana e hilo}-morfismos

Utilizando uma estruturação muito semelhante àquela presente em *FTree*, foi possível chegar à presente estruturação de cata, ana e hilo-morfismos. Seguindo também a ideia base por detrás destas funções.

$$recExpr\ f = baseExpr\ id\ f$$

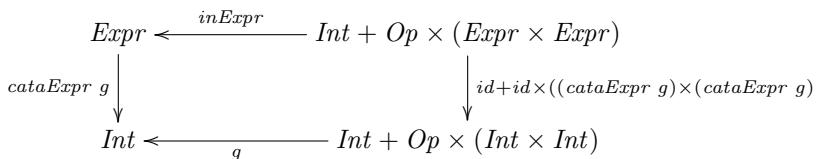
$$cataExpr\ g = g \cdot (recExpr\ (cataExpr\ g)) \cdot outExpr$$

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

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

### D.0.2 Calcula

Denote-se primeiro, que o cariz deste tipo de função é quase que obrigatoriamente um catamorfismo, pretende consumir está nossa estrutura *Expr* de forma a obter o resultado esperado por esta. Daí que, este catamorfismo visa então transformar esta estrutura em um inteiro, e pode ser resumida no seguinte diagrama:



Apartir deste diagrama acima, fica ímplicito que o gene deste catamorfismo terá de ser capaz de transformar o que tanto pode ser um só número em si próprio, ou indicar o resultado de uma operação sobre dois números. Daí que se chega à seguinte codificação final:

$$opCorresponde :: Op \rightarrow (Int, Int) \rightarrow Int$$

$$opCorresponde (Op\ "+") = \widehat{(+)}$$

$$opCorresponde (Op\ "*") = \widehat{(*)}$$

$$calcula :: Expr \rightarrow Int$$

$$calcula = cataExpr [id, opCorresponde]$$

### D.0.3 Compile

Tal como a função acima, esta é também necessariamente um catamorfismo, do qual se pretende obter uma lista de códigos. Esta lista de códigos nada mais é do que um representação textual destas entidades numéricas.

Após estudar com atenção os exemplos fornecidos, chegou-se a conclusão que a ordenação que se pretendia obter corresponde a uma travessia *post-order*. Daí que o gene deste catamorfismo deve ter em atenção a existência desta travessia.

```

compile :: String → Código
compile = cataExpr [singl · showInt, posOrd] · π₁ · head · readExp
  where posOrd = (++) · swap · ((singl · showOp) × (++)
    showInt = (++) "PUSH " · show
showOp :: Op → String
showOp (Op "+") = "ADD"
showOp (Op "*") = "MUL"

```

### D.0.4 Show

Como as demais, esta função possui também uma natureza de catamorfismo. Na qual se pretende obter a representação textual de uma *Expr*. Logo, o gene deste catamorfismo deve ser capaz de ou mostrar diretamente o número existente (caso este seja o elemento da *Expr*), ou representar textualmente a operação aritmética associada a um dado nó e correspondentes filhos.

```

show' = cataExpr [show, showAux]
  where showAux ((Op a), (s1, s2)) = "(" ++ s1 ++ " " ++ a ++ " " ++ s2 ++ ")"

```

## D.1 Problema 2

### D.1.1 {Cata, ana e hilo}-morfismos

Semelhante ao exercícios anterior, todos tipos de morfismos são iguais aos acima. Visto que continua a se tratar de um tipo de árvore específico.

```

inL2D :: a + (b, (X a b, X a b)) → X a b
inL2D = [Unid, λ(b, (x1, x2)) → Comp b x1 x2]
outL2D :: X a b → a + (b, (X a b, X a b))
outL2D (Unid a) = i₁ a
outL2D (Comp b x1 x2) = i₂ (b, (x1, x2))
baseL2D f g h = f + (g × (h × h))
recL2D f = baseL2D id id f
cataL2D g = g · recL2D (cataL2D g) · outL2D
anaL2D g = inL2D · recL2D (anaL2D g) · g

```

### D.1.2 Collect leafs

Esta função visa armazenar todas as folhas presentes nesta nossa estrutura. Daí que é quase que estritamente um catamorfismo. Sendo que se coleciona as folhas caso seja possível, caso contrário retorna-se o *nil*.

```
collectLeafs = cataL2D [singl, (++) · π₂]
```

### D.1.3 Dimensões

Após estudo, chegou-se à conclusão que há fatores em comum entre os diferentes tipos de colocação de figuras, pelo menos no que diz respeito às dimensões. Neste aspecto, todos os tipos na vertical partilham entre si uma característica, tal como todos os tipos na horizontal. Sendo que desta forma é possível de forma eficiente obter as dimensões reais da figura. Como se segue:

```

dimen :: X Caixa Tipo → (Float, Float)
dimen = cataL2D [(fromIntegral × fromIntegral) · π₁, getDem · (id × joinAxis)]
  where joinAxis = ⟨π₁ × π₁, π₂ × π₂⟩
getDem :: Tipo → ((Float, Float), (Float, Float)) → (Float, Float)
getDem V = (uMax × uAdd)
getDem Vd = (uMax × uAdd)
getDem Ve = (uMax × uAdd)
getDem _ = (uAdd × uMax)
uMax =  $\widehat{\max}$ 
uAdd =  $\widehat{(+)}$ 
```

### D.1.4 Calcula origens

Necessariamente um anamorfismo, sendo que se pretende algo aquilo uma estrutura específica. Daí que através de sucessivas aplicações se obtém uma árvore, tal que as suas folhas correspondem unicamente a todas as caixas e respetivas origens presentes no sistema.

```

calcOrigins :: ((X Caixa Tipo), Origem) → X (Caixa, Origem) ()
calcOrigins = anal2D [i₁ · id, i₂ · fty] · distl · (outL2D × id)
  where fty = λ((t, (x₁, x₂)), o) → (((), ((x₁, o), (x₂, calc t o (dimen x₁))))))
calc :: Tipo → Origem → (Float, Float) → Origem
calc V o (a, b) = sumTup o (a / 2, b)
calc Vd o b = sumTup o b
calc Ve o (a, b) = sumTup o (0, b)
calc H o (a, b) = sumTup o (a, b / 2)
calc Hb o (a, b) = sumTup o (a, 0)
calc Ht o b = sumTup o b
sumTup (a, b) (c, d) = (a + c, b + d)
```

### D.1.5 Finalização

De forma a finalizar este secção, e assumindo a existência das funções mencionadas no relatório, chegou-se às seguintes definições intuitivas sobre o que foi pedido.

```

agrup_caixas :: X (Caixa, Origem) () → Fig
agrup_caixas = cataList [nil,  $\widehat{(:)}$  · (swap × id)] · collectLeafs
caixasAndOrigin2Pict = G.pictures · cataList [nil,  $\widehat{(:)}$  · (makePic × id)] · agrup_caixas · calcOrigins
  where makePic (org, ((ii, ij), (t, c))) = crCaixa org (fromIntegral ii) (fromIntegral ij) t c
mostra_caixas = display · caixasAndOrigin2Pict
```

## Problema 3

Tal como descrito em exercícios semelhantes, apresenta-se de seguida a definição que permitiu alcançar esta solução.

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

Seja  $c x n = \sum_{i=0}^n \frac{(-1)^i}{(2i)!} x^{2i}$  a função que permite obter está aproximação, então resulta que  $c x 0 = 1$  e que  $c x (n+1) = c x n + \frac{(-1)^{n+1}}{(2n+2)!} x^{2n+2}$ . Esta função é passível de ser simplificada, definindo  $k x n = \frac{(-1)^{n+1}}{(2n+2)!} x^{2n+2}$  a partir do qual se obtém que  $k x 0 = -\frac{x^2}{2}$  e que  $k x (n+1) = \frac{-x^2}{2(n+2)(2n+3)} (k x n)$ . A qual é possível simplificar ainda mais assumindo  $s n = (n+2)(2n+3)$  do qual se obtém que  $s 0 = 6$  e que  $s (n+1) = 4n+9+s n$ , e, como as demais, é possível simplificar ainda mais definindo  $j n = 4n+9$ , que permite obter  $j 0 = 9$  e que  $j (n+1) = 4 + j n$ , a qual é impossível simplificar mais. Daí que se obtém a seguinte quatro funções mutuamente recursivas.

$$\begin{aligned} c x 0 &= 1 \\ c x (n+1) &= (k x n) + (c x n) \\ k x 0 &= -(x \uparrow 2) / 2 \\ k x (n+1) &= (-x \uparrow 2 / (2 * (s n))) * (k x n) \\ s 0 &= 6 \\ s (n+1) &= (j n) + (s n) \\ j 0 &= 9 \\ j (n+1) &= 4 + (j n) \end{aligned}$$

As quais permitem obter a seguinte definição em Haskell:

Solução:

$$\begin{aligned} cos' x &= prj \cdot \text{for loop init where} \\ \text{loop } (c, k, s, j) &= (k + c, -(x \uparrow 2) / (2 * s) * k, j + s, 4 + j) \\ \text{init} &= (1, -(x \uparrow 2) / 2, 6, 9) \\ prj (c, k, s, j) &= c \end{aligned}$$

## Problema 4

Após estudo dos *data types* em questão chegou-se a conclusão de que, o *data type FS* pode ser associado ao seguinte functor base.

$$B (a, b, FS a b) = [(a, b + (FS a b))]$$

### D.1.6 outNode e outFS

Genericamente, a função *outNode* permite criar uma estrutura *Either* que contém ou o conteúdo de um dado ficheiro, ou a estrutura *FS* associada a uma dada diretória. Desta forma podendo contemplar ambas as entidades de uma só vez.

A função *outFS* aproveita esta definição, no sentido em que permite sintetizar, na formatação do functor, a estrutura *FS a b*, associado a cada identificador, ou o ficheiro a que corresponde, ou a estrutura/diretório *FS* a que diz respeito.

$$\begin{aligned} outNode :: Node a b &\rightarrow b + (FS a b) \\ outNode (File b) &= i_1 b \\ outNode (Dir fs) &= i_2 fs \\ outFS :: FS a b &\rightarrow [(a, b + (FS a b))] \\ outFS (FS l) &= \text{map } (id \times outNode) l \end{aligned}$$

### D.1.7 {Cata, ana e hilo}-morfismos

Assumindo a seguinte codificação do functor base, é possível fornecer uma definição à função *recFS* que será necessária mais à frente.

$$baseFS f g h = \text{map } (f \times (g + h))$$

Com o conhecimento do functor base, encontrar o correspondente catamorfismo é trivial, sendo que corresponde à típica técnica utilizada, que origina o seguinte diagrama, com a respetiva codificação associada:

$$\begin{array}{ccc}
 FS\ a\ b & \xrightarrow{\text{outFS}} & [a \times (b + (FS\ a\ b))] \\
 cataFS\ g \downarrow & & \downarrow \text{map } (id \times (id + (cataFS\ g))) \\
 C & \xleftarrow[g]{} & [a \times (b + C)]
 \end{array}$$

$$\begin{aligned}
 cataFS &:: ((a, b + c) \rightarrow c) \rightarrow FS\ a\ b \rightarrow c \\
 cataFS\ g &= g \cdot recFS\ (cataFS\ g) \cdot outFS
 \end{aligned}$$

De igual forma, é possível construir o seguinte anamorfismo:

$$\begin{array}{ccc}
 FS\ a\ b & \xleftarrow{\text{inFS}} & [a \times (b + (FS\ a\ b))] \\
 anaFS\ g \uparrow & & \uparrow \text{map } (id \times (id + (anaFS\ g))) \\
 C & \xrightarrow[g]{} & [a \times (b + C)]
 \end{array}$$

$$\begin{aligned}
 anaFS &:: (c \rightarrow [(a, b + c)]) \rightarrow c \rightarrow FS\ a\ b \\
 anaFS\ g &= inFS \cdot recFS\ (anaFS\ g) \cdot g
 \end{aligned}$$

Logo, é possível obter a seguinte definição do hilomorfismo:

$$hyloFS\ g\ h = cataFS\ g \cdot anaFS\ h$$

### D.1.8 Check

$$\begin{aligned}
 check &:: (Eq\ a) \Rightarrow FS\ a\ b \rightarrow Bool \\
 check &= cataFS\ (checkAll \cdot checkSub) \\
 \text{where } checkSub &= \text{map } (id \times [\underline{True}, id]) \\
 checkCur &= \widehat{(\equiv)} \cdot \langle nub, id \rangle \cdot \text{map } \pi_1 \\
 getMin &= cond\ (\equiv []) \underline{True} (minimum \cdot \text{map } \pi_2) \\
 checkAll &= \widehat{(\wedge)} \cdot \langle checkCur, getMin \rangle
 \end{aligned}$$

### D.1.9 Tar

$$\begin{aligned}
 tar &:: FS\ a\ b \rightarrow [(Path\ a, b)] \\
 tar &= cataFS\ (concat \cdot \text{map } (mapPath \cdot (id \times [filePath, id]))) \\
 \text{where } filePath &= singl \cdot \langle [], id \rangle \\
 mapPath\ (a, b) &= \text{map } ((:) a \times id) b
 \end{aligned}$$

### D.1.10 Untar

$$\begin{aligned}
 untar &:: (Eq\ a) \Rightarrow [(Path\ a, b)] \rightarrow FS\ a\ b \\
 untar &= joinDupDirs \cdot anaFS\ (\text{map } (cond\ ((\equiv 1) \cdot \text{length} \cdot \pi_1) (head_1) untarPath)) \\
 \text{where } untarPath &= \langle head \cdot \pi_1, i_2 \cdot singl \cdot \langle tail \cdot \pi_1, \pi_2 \rangle \rangle
 \end{aligned}$$

### D.1.11 *Find*

Após aplicar a função *tar* obtém-se todos os *paths* até todos os ficheiros, sendo que depois basta verificar que algum desses *paths* possui o identificador do ficheiro como diretória. E se de facto se verificar, então significa que aquele *path* é válido até ao ficheiro que se pretende.

$$\begin{aligned} find &:: (Eq\ a) \Rightarrow a \rightarrow FS\ a\ b \rightarrow [Path\ a] \\ find\ a &= cataList\ [[], keepCond\ a] \cdot tar \\ \text{where } keepCond\ a &= cond\ ((\equiv\ a) \cdot last \cdot \pi_1 \cdot \pi_1) (\widehat{(:)} \cdot (\pi_1 \times id)) \pi_2 \end{aligned}$$

### D.1.12 *New*

Adiciona na cabeça da representação em lista da estrutura *FS a b* o *path* indicado e o respetivo conteúdo do ficheiro. Aplicando de seguida a função *untar* que cria a estrutura *FS a b* com este novo ficheiro.

$$\begin{aligned} new &:: (Eq\ a) \Rightarrow Path\ a \rightarrow b \rightarrow FS\ a\ b \rightarrow FS\ a\ b \\ new\ p\ b &= untar \cdot ((p, b) :) \cdot tar \end{aligned}$$

### D.1.13 *Copy*

Após obter a representação em lista da estrutura *FS a b*, é efetuada uma procurar pelo *path* de origem indicado, e de seguida, se existir tal *path* adicionar este à estrutura *FS a b* utilizando a função *new* acima.

$$\begin{aligned} cp &:: (Eq\ a) \Rightarrow Path\ a \rightarrow Path\ a \rightarrow FS\ a\ b \rightarrow FS\ a\ b \\ cp\ orig\ dest &= cond\ (isJust \cdot \pi_1) (\widehat{(new\ dest)} \cdot (fromJust \times id)) \pi_2 \cdot \langle lookup\ orig \cdot tar, id \rangle \end{aligned}$$

### D.1.14 *Remove*

Devido à existência das funções *tar* e *untar* é possível criar é possível obter uma representação em lista da estrutura *FS a b* e de seguida remover desta estrutura o *path* indicado, e de seguida voltar a montar a estrutura *FS a b*.

$$\begin{aligned} rm &:: (Eq\ a) \Rightarrow (Path\ a) \rightarrow (FS\ a\ b) \rightarrow FS\ a\ b \\ rm\ p &= untar \cdot filter\ ((\not\equiv\ p) \cdot \pi_1) \cdot tar \end{aligned}$$

### D.1.15 Funções não construídas

$$\begin{aligned} auxJoin &::([(a, b + c)], d) \rightarrow [(a, b + (d, c))] \\ auxJoin &= \perp \\ cFS2Exp &:: a \rightarrow FS\ a\ b \rightarrow (Exp\ ()\ a) \\ cFS2Exp &= \perp \end{aligned}$$