

Cálculo de Programas

Trabalho Prático

MiEI+LCC — 2018/19

Departamento de Informática
Universidade do Minho

Junho de 2019

Grupo nr.	26
a85227	João Pedro Rodrigues Azevedo
a83719	Pedro Filipe Costa Machado
a85729	Paulo Jorge da Silva Araújo

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**” [1], 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`¹ 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^AT_EX** 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
```

¹O 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 $\{\text{cata}, \text{ana ou hilo}\}$ -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 =  $\widehat{\widehat{Bop}} \cdot (\text{swap} \times \text{id}) \cdot \text{assocl} \cdot (\text{id} \times \text{swap})$ 
```

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

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.² 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).

²Pode 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.³

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.⁴
- 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 aulas⁵, 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 π e o coseno de $\pi / 2$:

$$\begin{aligned}prop_cos1\ n = n \geq 10 &\Rightarrow abs\ (\cos\ \pi - \cos'\ \pi\ n) < 0.001 \\prop_cos2\ n = n \geq 10 &\Rightarrow abs\ (\cos\ (\pi / 2) - \cos'\ (\pi / 2)\ n) < 0.001\end{aligned}$$

³Lei (3.94) em [2], página 98.

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

⁵Secção 3.17 de [2].

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:⁶

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

Os diagramas podem ser produzidos recorrendo à *package* L^AT_EX *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 disciplina⁷, 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:

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

⁶Exemplos tirados de [2].

⁷Cf. [2], 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:

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

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 (\(x, y) → ((Num x), y))) · reads

readBinOp :: ReadS Expr
readBinOp = (map (\((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
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:⁸

```
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.

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

Problema 1

Pergunta 1 - Definições base para o tipo de dados

```

inExpr :: Int + (Op, (Expr, Expr)) → Expr
inExpr = [Num, bopCase]
  where bopCase (a, (b, c)) = Bop b a c
outExpr :: Expr → Int + (Op, (Expr, Expr))
outExpr (Num a) = i1 a
outExpr (Bop a op b) = i2 (op, (a, b))
recExpr f = baseExpr id f
cataExpr g = g · (recExpr (cataExpr g)) · outExpr
anaExpr g = inExpr · (recExpr (anaExpr g)) · g
hiloExpr h g = cataExpr h · anaExpr g

```

Pergunta 2 - Calcular o valor de uma expressão

Calcular um valor de uma expressão passa por dois casos, explicitados no tipo de dados. Ou temos um *Num* ou um *Bop* e trata-se de uma operação de redução desse tipo a um número, daí a solução passar por um catamorfismo.

Segue-se o diagrama mais apropriado para descrever este catamorfismo:

$$\begin{array}{ccc}
 \text{Expr} & \xrightarrow{\text{outExpr}} & \text{Num} + \text{Op} \times \text{Expr} \times \text{Expr} \\
 \downarrow f = \text{cataExpr } g & & \downarrow \text{id} + \text{id} \times f \times f \\
 \text{Int} & \xleftarrow{g = [\text{id}, \text{parseOp}]} & \text{Num} + \text{Op} \times \text{Int} \times \text{Int}
 \end{array}$$

Aqui *parseOp* auxilia a função *calcula* em converter uma operação e um par de Inteiros no resultado respetivo.

```

calcula :: Expr → Int
calcula = cataExpr [id, parseOp]

--
parseOp :: (Op, (Int, Int)) → Int
parseOp (Op op, (a, b)) | op == "+" = a + b
                        | op == "-" = a - b
                        | op == "*" = a * b
                        | op == "mod" = mod a b

```

Pergunta 3 - Pretty printer show' e a compile

Para converter um tipo de dados *Expr* na sua representação *String* basta-o converter para uma representação literal infixa. Um catamorfismo, mais uma vez, serviu para o caso.

```

show' = cataExpr [cnvNum, cnvPar]

--
cnvNum :: (Num a, Ord a, Show a) ⇒ a → String
cnvNum a | a < 0 = "(" ++ show a ++ ")"
          | otherwise = show a

--
cnvPar :: (Op, (String, String)) → String
cnvPar (Op op, (s1, s2)) = "(" ++ s1 ++ op ++ s2 ++ ")"

```

A operação de **compilar** uma *String* torna-se trivial após se conseguir obter a representação da *String* num *Expr*, tendo para isso, a função *readExp* fornecida.

Após isso, basta apenas fazer um catamorfismo sobre o tipo de modo a explorar a recursividade.

```
compile = cataExpr [numToComp, bopToCompile] · π1 · head · readExp
  where numToComp a = ["PUSH " ++ show a]
        bopToCompile (op, (s1, s2)) = s1 ++ s2 ++ [opToCodigo op]
--
opToCodigo :: Op → String
opToCodigo (Op op) | op ≡ "+" = "ADD "
                  | op ≡ "-" = "SUB "
                  | op ≡ "*" = "MUL "
                  | op ≡ "mod" = "MOD "
--
```

Para melhor representar a operação de compile, ou seja, converter *Expr* em instruções **stack** segue-se o diagrama do catamorfismo implementado.

$$\begin{array}{ccc}
 Expr & \xrightarrow{\text{outExpr}} & Num + Op \times Expr \times Expr \\
 \downarrow f = \text{cataExpr } g & & \downarrow id + id \times f \times f \\
 [String] & \xleftarrow{g = [\text{numToComp}, \text{bopToCompile}]} & Num + Op \times [String] \times [String]
 \end{array}$$

Incluem-se aqui alguns testes aplicáveis às funções implementadas anteriormente:

```
exp_test :: Expr
exp_test = Bop (Bop (Num 5) (Op "*")) (Num 6)) (Op "+") (Num (-1))
string_codigo_1 :: String
string_codigo_1 = "2 + 3 * 4"
string_codigo_2 :: String
string_codigo_2 = "2 + 1 * 3 + 4"
```

Problema 2

Definições base do tipo de dados L2D

Este problema utilizava um tipo de dados que em muito se assemelhava com um tipo já conhecido por todos, as B-Trees, daí ter sido mais fácil interpretar a estrutura de dados. Deste modo, as operações de *in* e *out*, entre outras, são semelhantes.

O diagrama que representa um catamorfismo para este tipo é o seguinte:

$$\begin{array}{ccc}
 L2D & \xrightarrow{\text{outL2D}} & Unid + b \times L2D \times L2D \\
 \downarrow f = \text{cataL2D } g & & \downarrow id + id \times f \times f \\
 X & \xleftarrow{g} & Unid + b \times X \times X
 \end{array}$$

```
inL2D :: a + (b, (X a b, X a b)) → X a b
inL2D = [casoUnid, casoComp]
  where casoUnid a = Unid a
        casoComp (b, (l, r)) = Comp b l r
--
```



```

outL2D :: X a b → a + (b, (X a b, X a b))
outL2D (Unid a) = i1 a
outL2D (Comp b l r) = i2 (b, (l, r))

--
recL2D f = baseL2D id f
baseL2D f g = id + (f × (g × g))
cataL2D g = g · (recL2D (cataL2D g)) · outL2D
anaL2D g = inL2D · (recL2D (anaL2D g)) · g

```

Definições de funções dadas como undefined neste documento

Algumas das definições necessárias para a implementação daquela que seria a função principal *calcOrigins* foram disponibilizadas no ficheiro fornecido e, deste modo, serviram para facilitar o anamorfismo que definia esta função.

```

collectLeafs (Unid a) = [a]
collectLeafs (Comp b l r) = (collectLeafs l) ++ (collectLeafs r)

--
dimen :: X Caixa Tipo → (Float, Float)
dimen = cataL2D [casoUnid, casoComp]
casoUnid a = (fromIntegral (π1 (π1 a)), fromIntegral (π2 (π1 a)))
casoComp (Hb, ((x1, y1), (x2, y2))) = (x1 + x2, max y1 y2)
casoComp (H, ((x1, y1), (x2, y2))) = (x1 + x2, max y1 y2)
casoComp (Ht, ((x1, y1), (x2, y2))) = (x1 + x2, max y1 y2)
casoComp (V, ((x1, y1), (x2, y2))) = (max x1 x2, y1 + y2)
casoComp (Ve, ((x1, y1), (x2, y2))) = (max x1 x2, y1 + y2)
casoComp (Vd, ((x1, y1), (x2, y2))) = (max x1 x2, y1 + y2)

```

Segue-se, então, o diagrama que define um anamorfismo neste tipo de dados:

$$\begin{array}{ccc}
 X \text{ (Caixa, Origem) } () & \xleftarrow{\text{inL2D}} & (\text{Unid}, O) + ((), (X \text{ (Caixa, O) } (), X \text{ (Caixa, O) } ())) \\
 \uparrow f = \text{anaL2D} & & \uparrow \text{id} + \text{id} \times f \times f \\
 (L2D, O) & \xrightarrow{g} & (\text{Unid}, O) + ((), ((L2D, O), (L2D, O)))
 \end{array}$$

Onde "O" representa o tipo Origem.

Pergunta 1

Nesta pergunta era pedido uma função que dada uma origem e um conjunto de figuras em L2D fossem calculadas associações das mesmas com a origem respetiva, dados os diferentes tipo de associações.

O padrão de recursividade segue mais casos e traz uma ligeira complexidade ao tipo de dados, mas contudo, este assemelha-se muito com uma BTree.

Seguem-se as definições da pergunta 1:

```

calcOrigins :: ((X Caixa Tipo), Origem) → X (Caixa, Origem) ()
calcOrigins = anaL2D g
  where g (Unid a, or) = i1 (a, or)
        g (Comp b l r, o) = i2 ((), ((l, o), (r, calc b o (dimen l))))

```

Seguem-se algumas funções auxiliares que foram necessárias ser implementadas.

```

--
convCoord :: (Int, Int) → (Float, Float)
convCoord (x, y) = (fromIntegral x :: Float, fromIntegral y :: Float)
--

```

```

calc :: Tipo → Origem → (Float, Float) → Origem
calc (Hb) orig (box_x, box_y) = ((π1 orig) + box_x, π2 orig)
calc (Ht) orig (box_x, box_y) = ((π1 orig) + box_x, (π2 orig) + box_y)
calc (H) orig (box_x, box_y) = ((π1 orig) + box_x, (π2 orig)
    + (fromInteger $ round $ (box_y / 2) * (10 ↑ 5)) / (10.0 ↑ 5))
calc (V) orig (box_x, box_y) = ((π1 orig)
    + (fromInteger $ round $ (box_x / 2) * (10 ↑ 5)) / (10.0 ↑ 5),
    (π2 orig) + box_y)
calc (Ve) orig (box_x, box_y) = ((π1 orig), (π2 orig) + box_y)
calc (Vd) orig (box_x, box_y) = ((π1 orig) + box_x, (π2 orig) + box_y)

```

Pergunta 2 - Agrupar as caixas e as suas origens numa só lista

```

agrup_caixas :: X (Caixa, Origem) () → Fig
agrup_caixas (Unid (ca, or)) = [(or, ca)]
agrup_caixas (Comp b l r) = (agrup_caixas l) ++ (agrup_caixas r)

```

Display das caixas num gráfico 2D em gloss

Tendo as origens calculadas apenas é preciso agrupar as caixas com as suas origens, gerar as respetivas *G.Picture*, convertendo tudo para apenas uma *G.Picture* e apresentar na janela gloss o resultado. Para tal foi crucial ter funções como *crCaixa* e *display* para criar *G.Picture* e apresentá-las.

```

mostra_Caixas = display · caixasAndOrigin2Pict
caixasAndOrigin2Pict = G.pictures · criaPict · agrup_caixas · calcOrigins
criaPict :: Fig → [G.Picture]
criaPict [] = []
criaPict (h : t) = [(crCaixa or width height text col)] ++ (criaPict t)
  where or      = (π1 h)
        width   = (toFloat (π1 (π1 (π2 h))))
        height  = (toFloat (π2 (π1 (π2 h))))
        text    = (π1 (π2 (π2 h)))
        col     = (π2 (π2 (π2 h)))
        toFloat x = fromIntegral x :: Float

```

Incluem-se aqui alguns testes aplicáveis às funções implementadas anteriormente:

```

testeOrigens1 :: (X Caixa Tipo, Origem)
testeOrigens1 = (caixaBas1, (0, 0))
testeOrigens2 :: (X Caixa Tipo, Origem)
testeOrigens2 = (caixaBas2, (0, 0))
unidadeA = Unid ((100, 200), ("A", col_blue))
unidadeB = Unid ((50, 50), ("B", col_green))
caixaBas1 = Comp Hb unidadeA unidadeB
caixaBas2 = Comp Hb unidadeA caixaBas1

```

Problema 3

Série de Taylor do cosseno - Uma aproximação...

O objetivo do problema era implementar a série de Taylor através de uma aproximação de n em vez de um somatório infinito, utilizando recursividade mútua.

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

Primeiro consideramos o $\cos x n$ como sendo $c x n$, assim, podemos verificar, através do somatório que:

$$\begin{aligned}
c x 0 &= 1 \\
c x (n + 1) &= c x n + ((-1)^{n+1} * x^{2*n+2} / ((2 * n + 2)!)) = c x n + h x n \\
h x n &= ((-1)^{n+1} * x^{2*n+2}) / ((2 * n + 2)!) \\
h x 0 &= ((-x)^2) / 2 \\
h x (n + 1) &= (h x n) * ((-x)^2 / ((2 * n + 4) * (2 * n + 3))) = (h x n) * (-x)^2 / (s n) \\
s x n &= ((2 * n + 4) * (2 * n + 3)) \\
s 0 &= 12 \\
s (n + 1) &= s n + 8 * n + 18 = s n + f n \\
f x n &= 8 * n + 18 \\
f 0 &= 18 \\
f (n + 1) &= f n + 8
\end{aligned}$$

Agora, em termos de *Haskell*, podemos separar o problema em 3 funções, *loop* que representa o resultado recursivo para cada função (cujo funcionamento está descrito a cima), *init* que inicializa o caso base, ou seja, o caso de paragem de cada função e por fim *prj* que decide a função a apresentar quando se pretende mostrar o resultado da aproximação.

$$\begin{aligned}
\cos' x &= prj \cdot \text{for loop init where} \\
\text{loop } (c, h, s, f) &= (c + h, h * (-x \uparrow 2) / s, s + f, f + 8) \\
\text{init} &= (1, -x \uparrow 2 / 2, 12, 18) \\
prj (c, h, s, f) &= c
\end{aligned}$$

Problema 4

Triologia “ana-cata-hilo” - Sistema de ficheiros genéricos

Seguindo a recomendação do enunciado deste problema, seguem-se as definições básicas para o tipo **FS a b**:

$$\begin{aligned}
outFS &:: FS a b \rightarrow [(a, (b) + (FS a b))] \\
outFS (FS []) &= [] \\
outFS (FS ((x, y) : t)) &= [(x, outNode y)] ++ outFS (FS t) \\
outNode &:: Node a b \rightarrow b + (FS a b) \\
outNode (File b) &= i_1 (b) \\
outNode (Dir x) &= i_2 (x) \\
baseFS f g h &= \text{map } (f \times (g + h)) \\
cataFS &:: [(a, b + c)] \rightarrow c \rightarrow FS a b \rightarrow c \\
cataFS g &= g \cdot (recFS (cataFS g)) \cdot outFS \\
anaFS &:: (c \rightarrow [(a, b + c)]) \rightarrow c \rightarrow FS a b \\
anaFS g &= inFS \cdot (recFS (anaFS g)) \cdot g \\
hyloFS g h &= cataFS h \cdot anaFS g
\end{aligned}$$

E também o diagrama para um catamorfismo genérico para este tipo de dados:

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

Funções para manipulação de ficheiros

a) $check :: FS\ a\ b \rightarrow Bool$

```
-- Para verificar se o sistema de ficheiros é válido.
check :: (Eq a, Eq b) => FS a b -> Bool
check = cataFS geneCheck

geneCheck :: (Eq a, Eq b) => [(a, b + Bool)] -> Bool
geneCheck [] = True
geneCheck ((a, i1 b) : cauda) = naoTemRepetidos (passaLista (((a, i1 b) : cauda))) & geneCheck cauda
geneCheck ((a, i2 bool) : cauda) = bool & geneCheck cauda

-- Verifica se um par existe numa lista de pares.
containsPair :: (Eq a, Eq b) => (a, b) -> [(a, b)] -> Bool
containsPair (a, b) [] = False
containsPair (a, b) ((h, k) : t) | (a == h) & (b == k) = True
  | otherwise = containsPair (a, b) t

-- Verifica se uma lista de pares não tem elementos repetidos.
naoTemRepetidos :: (Eq a, Eq b) => [(a, b)] -> Bool
naoTemRepetidos [x] = True
naoTemRepetidos (h : t) | containsPair h t = False
  | otherwise = naoTemRepetidos t

-- Cria uma lista de pares no qual a primeira componente é o nome da Dir/File e a segunda componente é o conteúdo.
passaLista :: [(a, b + bool)] -> [(a, b)]
passaLista [] = []
passaLista ((a, i1 b) : cauda) = [(a, b)] ++ passaLista cauda
passaLista ((a, i2 bool) : cauda) = passaLista cauda
```

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

```
tar :: FS a b -> [(Path a, b)]
tar = cataFS geneTar

geneTar :: [(a, b + [(Path a, b)])] -> [(Path a, b)]
geneTar [] = []
geneTar ((a, i1 b) : t) = [(a, b)] ++ geneTar t
geneTar ((a, i2 l) : t) = (addAllPaths a l) ++ (geneTar t)

-- Para adicionar uma diretoria à lista de diretorias existentes, onde temos uma par composto por um ficheiro e as diretorias.
addAllPaths :: a -> [(Path a, b)] -> [(Path a, b)]
addAllPaths a [] = []
addAllPaths a ((list, b) : t) = [(a, b)] ++ list ++ addAllPaths a t
```

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

```
untar :: (Eq a) => [(Path a, b)] -> FS a b
untar = joinDupDirs . (anaFS geneUntar)

--
geneUntar :: [(Path a, b)] -> [(a, b + [(Path a, b)])]
geneUntar [] = []
geneUntar (([], b) : cauda) = geneUntar cauda
geneUntar (([x], b) : cauda) = [(x, i1 b)] ++ geneUntar cauda
geneUntar (((h : t), b) : cauda) = [(h, i2 [(t, b)])] ++ geneUntar cauda
```

A função `untar` utiliza um anamorfismo que é descrito de seguida:

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

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

```
find :: (Eq a) => a -> FS a b -> [Path a]
find (a) (f) = findAux (a) (cataFS (gFind) (f))
```

-- Verifica se um elemento existe numa lista.

```
contains :: (Eq a) => a -> [a] -> Bool
contains a [] = False
contains a (h : t) | (a == h) = True
| otherwise = contains a t
```

-- A partir de uma lista devolve a lista de diretorias associadas ao ficheiro pretendido.

```
findAux :: (Eq a) => a -> [(a, Path a)] -> [Path a]
findAux a [] = []
findAux a ((a1, l) : t) | contains a l = [l] ++ (findAux a t)
| otherwise = (findAux a t)
```

-- Adiciona uma nova diretorio à lista de diretorias de um ficheiro.

```
addPathA :: a -> [(a, Path a)] -> [(a, Path a)]
addPathA a [] = []
addPathA a ((a1, l) : t) = [(a, [a] ++ l)] ++ addPathA a t
```

--

```
gFind :: [(a, b + [(a, Path a)])] -> [(a, Path a)]
gFind [] = []
gFind ((a, i1 b) : t) = [(a, [a])] ++ gFind t
gFind ((a, i2 p) : t) = (addPathA a p) ++ gFind t
```

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

```
new :: (Eq a) => Path a -> b -> FS a b -> FS a b
new p b fs = untar (ordenaPath (tar (fs)) (p, b))
```

-- Insere um Path e correspondente ficheiro na lista de diretorias do Sistema de ficheiros de forma ordenada.

```
ordenaPath :: (Eq a) => [(Path a, b)] -> (Path a, b) -> [(Path a, b)]
ordenaPath [] (l, b) = [(l, b)]
ordenaPath ((l, b) : t) (l1, b1) | (take ((length l) - 1) (l)) == (take ((length l) - 1) (l1)) = (l, b) : (l1, b1) : t
| otherwise = (l, b) : ordenaPath t (l1, b1)
```

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

```
cp :: (Eq a) => Path a -> Path a -> FS a b -> FS a b
cp list1 list2 (FS list3) = new list2 (encontraFile list1 (tar (FS list3))) (FS list3)
```

-- Devolve, se encontrar, o conteúdo de um ficheiro associado a um Path.

```
encontraFile :: (Eq a) => Path a -> [(Path a, b)] -> b
encontraFile list ((x, y) : t) | list == x = y
| otherwise = encontraFile list t
```

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

Sugestão de implementação, definição de *nav*:

```
nav :: (Eq a) => (Path a, FS a b) -> FS a b
nav = anaFS gNav
```

--

$$\begin{aligned}
gNav &:: (Eq\ a) \Rightarrow (Path\ a, FS\ a\ b) \rightarrow [(a, b + (Path\ a, FS\ a\ b))] \\
gNav\ ([], FS\ []) &= [] \\
gNav\ ((h : t), FS\ []) &= [] \\
gNav\ ([], FS\ ((a, File\ b) : t1)) &= [(a, i_1\ b)] ++ gNav\ ([], FS\ t1) \\
gNav\ ([], FS\ ((a, Dir\ f) : t1)) &= [(a, i_2\ ([], f))] ++ gNav\ ([], FS\ t1) \\
gNav\ ((h : t), FS\ ((a, File\ b) : t1)) &| (h \equiv a) = [(a, i_1\ b)] \\
&| otherwise = gNav\ (h : t, FS\ t1) \\
gNav\ ((h : t), FS\ ((a, Dir\ f) : t1)) &| (h \equiv a) = [(a, i_2\ (t, f))] \\
&| otherwise = gNav\ (h : t, FS\ t1) \\
-- \\
geneRemove &:: (Eq\ a) \Rightarrow (FS\ a\ b, FS\ a\ b) \rightarrow [(a, b + (FS\ a\ b, FS\ a\ b))] \\
geneRemove\ (FS\ [], -) &= [] \\
geneRemove\ (FS\ ((a, Dir\ f) : t1), FS\ []) &= [(a, i_2\ (f, FS\ []))] ++ geneRemove\ (FS\ t1, FS\ []) \\
geneRemove\ (FS\ ((a, File\ b) : t1), FS\ []) &= [(a, i_1\ b)] ++ geneRemove\ (FS\ t1, FS\ []) \\
geneRemove\ (FS\ ((a, File\ b) : t1), FS\ ((a1, File\ b1) : t2)) &| a \equiv a1 = [(a, i_2\ (FS\ t1, FS\ []))] \\
&| otherwise = [(a, i_1\ b)] \\
&+ geneRemove\ (FS\ t1, FS\ ((a1, File\ b1) : t2)) \\
geneRemove\ (FS\ ((a, Dir\ f1) : t1), FS\ ((a1, Dir\ f2) : t2)) &| a \equiv a1 = [(a, i_2\ (FS\ t1, FS\ []))] \\
&+ geneRemove\ (f1, f2) \\
&| otherwise = [(a, i_2\ (f1, FS\ []))] \\
&+ geneRemove\ (FS\ t1, FS\ ((a1, Dir\ f2) : t2)) \\
geneRemove\ (FS\ ((a, File\ b) : t1), FS\ ((a1, Dir\ f2) : t2)) &| a \not\equiv a1 = [(a, i_1\ b)] \\
&+ geneRemove\ (FS\ t1, FS\ ((a1, Dir\ f2) : t2)) \\
geneRemove\ (FS\ ((a, Dir\ f1) : t1), FS\ ((a1, File\ b) : t2)) &| a \not\equiv a1 = [(a, i_2\ (f1, FS\ []))] \\
&+ geneRemove\ (FS\ t1, FS\ ((a1, File\ b) : t2))
\end{aligned}$$

Definição de *rm*:

$$\begin{aligned}
rm &:: (Eq\ a) \Rightarrow (Path\ a) \rightarrow (FS\ a\ b) \rightarrow FS\ a\ b \\
rm\ p\ f &= anaFS\ (geneRemove)\ (f, nav\ (p, f))
\end{aligned}$$

Definições extras:

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

References

- [1] D.E. Knuth. *Literate Programming*. CSLI Lecture Notes Number 27. Stanford University Center for the Study of Language and Information, Stanford, CA, USA, 1992.
- [2] J.N. Oliveira. *Program Design by Calculation*, 2018. Draft of textbook in preparation. viii+297 pages. Informatics Department, University of Minho.