Cálculo de Programas
Trabalho Prático
MiEI+LCC — 2018/19

Departamento de Informática
Universidade do Minho

Junho de 2019

Grupo nr.	1
a70565	Bruno Manuel Borlido Arieira
a73864	Joao Miguel Freitas Palmeira
a74264	Rafael Machado da Silva

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`[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-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
```

[1] O suffixo 'lhs' quer dizer *literate Haskell*.

Abra o ficheiro `cp1819t.lhs` no seu editor de texto preferido e verifique que assim é: todo o texto que se encontra dentro do ambiente

```
\begin{code}
...
\end{code}
```

vai ser seleccionado pelo GHCi para ser executado.

3 Como realizar o trabalho

Este trabalho teórico-prático deve ser realizado por grupos de três alunos. Os detalhes da avaliação (datas para submissão do relatório e sua defesa oral) são os que forem publicados na página da disciplina na *internet*.

Recomenda-se uma abordagem participativa dos membros do grupo de trabalho por forma a poderem responder às questões que serão colocadas na *defesa oral* do relatório.

Em que consiste, então, o *relatório* a que se refere o parágrafo anterior? É a edição do texto que está a ser lido, preenchendo o anexo D com as respostas. O relatório deverá conter ainda a identificação dos membros do grupo de trabalho, no local respectivo da folha de rosto.

Para gerar o PDF integral do relatório deve-se ainda correr os comando seguintes, que actualizam a bibliografia (com BibTEX) e o índice remissivo (com `makeindex`),

```
$ bibtex cp1819t.aux
$ makeindex cp1819t.idx
```

e recompilar o texto como acima se indicou. Dever-se-á ainda instalar o utilitário QuickCheck, que ajuda a validar programas em Haskell e a biblioteca Gloss para geração de gráficos 2D:

```
$ cabal install QuickCheck gloss
```

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

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

Qualquer programador tem, na vida real, de ler e analisar (muito!) código escrito por outros. No anexo C disponibiliza-se algum código Haskell relativo aos problemas que se seguem. Esse anexo deverá ser consultado e analisado à medida que isso for necessário.

Problema 1

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

Compiladores são normalmente programas complexos. Constam 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:

$$\mathbf{data}\ Expr = Num\ Int \mid Bop\ Expr\ Op\ Expr$$
$$\mathbf{data}\ Op = Op\ String$$

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:

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

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

     $$\begin{aligned}&prop\_neutro1 :: Expr \to Bool\\&prop\_neutro1 = calcula \cdot addZero \equiv calcula \textbf{ where}\\&\quad addZero\ e = Bop\ (Num\ 0)\ (Op\ \texttt{"+"})\ e\\&prop\_neutro2 :: Expr \to Bool\\&prop\_neutro2 = calcula \cdot addZero \equiv calcula \textbf{ where}\\&\quad addZero\ e = Bop\ e\ (Op\ \texttt{"+"})\ (Num\ 0)\end{aligned}$$

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

     $$\begin{aligned}&prop\_comuta = calcula \cdot mirror \equiv calcula \textbf{ where}\\&\quad mirror = (\!|[Num, g2]|\!)\\&\quad g2 = \widehat{\widehat{Bop}} \cdot (swap \times id) \cdot assocl \cdot (id \times swap)\end{aligned}$$

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

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

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

   - $show' :: Expr \to String$ - 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:*

     $$\begin{aligned}&prop\_inv :: Expr \to Bool\\&prop\_inv = \pi_1 \cdot head \cdot readExp \cdot show' \equiv id\end{aligned}$$

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:

$$\textbf{type } L2D = X\ Caixa\ Tipo$$

onde X é a estrutura de dados

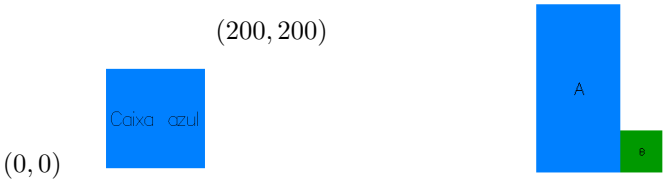


Figure 1: Caixa simples e caixa composta.

$$\textbf{data}\ X\ a\ b = Unid\ a \mid Comp\ b\ (X\ a\ b)\ (X\ a\ b)\ \textbf{deriving}\ Show$$

e onde:

$$\textbf{type}\ Caixa = ((Int, Int), (Texto, G.Color))$$
$$\textbf{type}\ Texto = String$$

Assim, cada caixa de texto é especificada pela sua largura, altura, o seu texto e a sua côr.[2] Por exemplo,

$$((200, 200), (\texttt{"Caixa azul"}, col\_blue))$$

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,

$$\textbf{data}\ Tipo = V \mid Vd \mid Ve \mid H \mid Ht \mid Hb$$

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

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

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).

[2] Pode relacionar *Caixa* com as caixas de texto usadas nos jornais ou com *frames* da linguagem HTML usada na Internet.

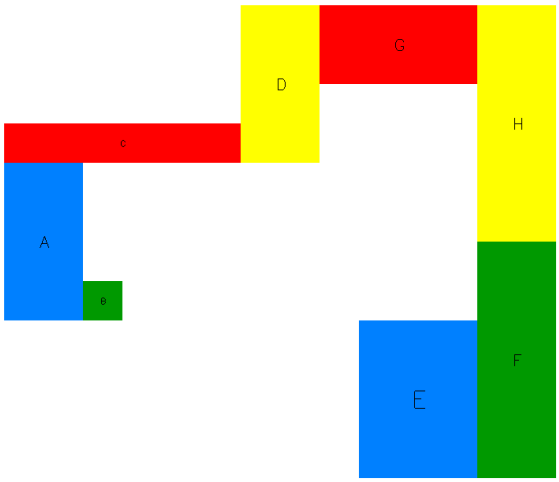


Figure 2: *Layout* feito de várias caixas coloridas.

$$\textbf{type } Fig = [(Origem, Caixa)]$$
$$\textbf{type } Origem = (Float, Float)$$

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

$$display :: G.Picture \rightarrow \mathsf{IO}\ ()$$

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

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

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 $\mathsf{F}\ X = 1 + X$) é fácil derivar-se da lei que foi estudada uma *regra de algibeira* que se pode ensinar a programadores que não tenham estudado Cálculo de Programas. Apresenta-se de seguida essa regra, tomando como exemplo o cálculo do ciclo-for que implementa a função de Fibonacci, recordar o sistema

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

Obter-se-á de imediato

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

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* coleccionam-se os resultados dos casos de base das funções, pela mesma ordem.

Mais um exemplo, envolvendo polinómios no segundo grau a $x^2 + bx + c$ em $\mathbb{N}_0$. Seguindo o método estudado nas aulas[5], 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:

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

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:

$$cos'\ x = \cdots \cdot \mathsf{for}\ loop\ init\ \mathbf{where}\ \cdots$$

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 π / 2:*

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

[3] Lei (3.94) em [**?**], página 98.
[4] Podem obviamente usar-se outros símbolos, mas numa primeiraleitura dá jeito usarem-se tais nomes.
[5] Secção 3.17 de [**?**].

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

Problema 4

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

$$\mathbf{data}\ FS\ a\ b = FS\ [(a, Node\ a\ b)]\ \mathbf{deriving}\ (Eq, Show)$$
$$\mathbf{data}\ Node\ a\ b = File\ b \mid Dir\ (FS\ a\ b)\ \mathbf{deriving}\ (Eq, Show)$$

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

$$\mathbf{type}\ Path\ a = [a]$$

Assumindo estes tipos de dados, o seguinte termo

```
FS [("f1", File "Ola"),
  ("d1", Dir (FS [("f2", File "Ole"),
    ("f3", File "Ole")
    ]))
  ]
```

representará um sistema de ficheiros em cuja raíz temos um ficheiro chamado *f1* com conteúdo `"Ola"` e uma directoria chamada `"d1"` constituída por dois ficheiros, um chamado `"f2"` e outro chamado `"f3"`, ambos com conteúdo `"Ole"`. 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:

$$inFS = FS \cdot \mathsf{map}\ (id \times inNode)$$
$$inNode = [File, Dir]$$
$$recFS\ f = baseFS\ id\ id\ f$$

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:

$$conta :: FS\ a\ b \to Int$$
$$conta = (\!|(sum \cdot \mathsf{map}\ ([\underline{1}, id] \cdot \pi_2))|\!)$$

O que é para fazer:

1. Definir as funções $outFS$, $baseFS$, $(\!|\cdot|\!)$, $[\!(\cdot)\!]$ e $(\!|\cdot|\!)$.
2. Apresentar, no relatório, o diagrama de $(\!|\cdot|\!)$.
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).
   $check :: FS\ a\ b \to Bool$

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

   $$prop\_check :: FS\ String\ String \to Bool$$
   $$prop\_check = check \cdot (\!|(inFS \cdot reverse)|\!) \equiv check$$

(b) Recolha do conteúdo de todos os ficheiros num arquivo indexado pelo *path*.
$tar :: FS\ a\ b \to [(Path\ a, b)]$

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

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

(c) Transformação de um arquivo com o conteúdo dos ficheiros indexado pelo *path* num sistema de ficheiros.
$untar :: [(Path\ a, b)] \to FS\ a\ b$
Sugestão: Use a função *joinDupDirs* para juntar directorias que estejam na mesma pasta e que possuam o mesmo identificador.

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

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

(d) Localização de todos os *paths* onde existe um determinado ficheiro.
$find :: a \to FS\ a\ b \to [Path\ a]$

Propriedade QuickCheck 8 *A composição* $tar \cdot untar$ *preserva todos os ficheiros no sistema.*

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

(e) Criação de um novo ficheiro num determinado *path*.
$new :: Path\ a \to b \to FS\ a\ b \to FS\ a\ b$

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

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

***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.*

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

(f) Duplicação de um ficheiro.
$cp :: Path\ a \to Path\ a \to FS\ a\ b \to FS\ a\ b$

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

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

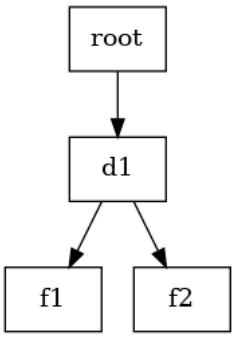


Figure 3: Exemplo de um sistema de ficheiros visualizado em Graphviz.

(g) Eliminação de um ficheiro.
$rm :: Path\ a \to FS\ a\ b \to FS\ a\ b$
Sugestão: Construir um anamorfismo $nav :: (Path\ a, FS\ a\ b) \to FS\ a\ b$ 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.*

$$prop\_rm :: (Path\ String, FS\ String\ String) \to Bool$$
$$prop\_rm = \widehat{rm} \cdot \langle \pi_1, \widehat{rm} \rangle \equiv \widehat{rm}$$

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

$$prop\_rm2 :: ((Path\ String, String), FS\ String\ String) \to Property$$
$$prop\_rm2 = validPath \Rightarrow ((\mathsf{length} \cdot tar \cdot \widehat{rm} \cdot \langle \pi_1 \cdot \pi_1, \widehat{\widehat{new}} \rangle)$$
$$\leqslant (\mathsf{length} \cdot tar \cdot \pi_2))\ \textbf{where}$$
$$validPath = (\not\equiv 0) \cdot \mathsf{length} \cdot \pi_1 \cdot \pi_1$$

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

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

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]

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

Os diagramas podem ser produzidos recorrendo à *package* LaTeX xymatrix, por exemplo:

$$
\begin{array}{ccc}
\mathbb{N}_0 & \xleftarrow{\ \mathsf{in}\ } & 1 + \mathbb{N}_0 \\
{\scriptstyle (\!|g|\!)}\downarrow & & \downarrow{\scriptstyle id + (\!|g|\!)} \\
B & \xleftarrow[\ g\ ]{} & 1 + B
\end{array}
$$

B Programação dinâmica por recursividade múltipla

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

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

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:

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

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

$$
\begin{array}{l}
e'\ x = prj \cdot \mathsf{for}\ loop\ init\ \mathbf{where} \\
\quad init = (1, x, 2) \\
\quad loop\ (e, h, s) = (h + e, x\ /\ s * h, 1 + s) \\
\quad prj\ (e, h, s) = e
\end{array}
$$

[6] Exemplos tirados de [?].
[7] Cf. [?], página 102.

C Código fornecido

Problema 1

Tipos:

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

Functor de base:

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

Instâncias:

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

Read para Exp's:

```
readOp :: String → [(Op, String)]
readOp input = do
  (x, y) ← lex input
  return ((Op x), y)
readNum :: ReadS Expr
readNum = (map (λ(x, y) → ((Num x), y))) · reads
readBinOp :: ReadS Expr
readBinOp = (map (λ((x, (y, z)), t) → ((Bop x y z), t))) ·
  ((readNum `ou` (pcurvos readExp))
    `depois` (readOp `depois` readExp))
readExp :: ReadS Expr
readExp = readBinOp `ou` (
  readNum `ou` (
  pcurvos readExp))
```

Combinadores:

```
depois :: (ReadS a) → (ReadS b) → ReadS (a, b)
depois _ _ [] = []
depois r1 r2 input = [((x, y), i2) | (x, i1) ← r1 input,
  (y, i2) ← r2 i1]
readSeq :: (ReadS a) → ReadS [a]
readSeq r input
  = case (r input) of
    [] → [([], input)]
    l → concat (map continua l)
      where continua (a, i) = map (c a) (readSeq r i)
        c x (xs, i) = ((x : xs), i)
ou :: (ReadS a) → (ReadS a) → ReadS a
ou r1 r2 input = (r1 input) ++ (r2 input)
senao :: (ReadS a) → (ReadS a) → ReadS a
senao r1 r2 input = case (r1 input) of
  [] → r2 input
  l  → l
readConst :: String → ReadS String
readConst c = (filter ((≡ c) · π1)) · lex
pcurvos = parentesis '(' ')'
```

```
precios = 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:

$$display = G.display\ (G.InWindow\ \texttt{"Problema 4"}\ (400,400)\ (40,40))\ G.white$$

Problema 4

Funções para gestão de sistemas de ficheiros:

$$concatFS = inFS \cdot \widehat{(+\!\!+)} \cdot (outFS \times outFS)$$
$$mkdir\ (x,y) = FS\ [(x, Dir\ y)]$$
$$mkfile\ (x,y) = FS\ [(x, File\ y)]$$
$$joinDupDirs :: (Eq\ a) \Rightarrow (FS\ a\ b) \to (FS\ a\ b)$$
$$joinDupDirs = [\!(prepOut \cdot (id \times proc) \cdot prepIn)\!]\ \textbf{where}$$
$$\quad prepIn = (id \times (\mathsf{map}\ \ (id \times outFS))) \cdot sls \cdot (\mathsf{map}\ \ distr) \cdot outFS$$
$$\quad prepOut = (\mathsf{map}\ \ undistr) \cdot \widehat{(+\!\!+)} \cdot ((\mathsf{map}\ \ i_1) \times (\mathsf{map}\ \ i_2)) \cdot (id \times (\mathsf{map}\ \ (id \times inFS)))$$
$$\quad proc = concat \cdot (\mathsf{map}\ \ joinDup) \cdot groupByName$$
$$\quad sls = \langle lefts, rights \rangle$$
$$joinDup :: [(a,[b])] \to [(a,[b])]$$
$$joinDup = cataList\ [nil, g]\ \textbf{where}\ g = return \cdot \langle \pi_1 \cdot \pi_1, concat \cdot (\mathsf{map}\ \ \pi_2) \cdot \widehat{(:)} \rangle$$
$$createFSfromFile :: (Path\ a, b) \to (FS\ a\ b)$$
$$createFSfromFile\ ([a], b) = mkfile\ (a, b)$$
$$createFSfromFile\ (a : as, b) = mkdir\ (a, createFSfromFile\ (as, b))$$

Funções auxiliares:

$$checkFiles :: (Eq\ a) \Rightarrow FS\ a\ b \to Bool$$
$$checkFiles = (\!|\widehat{(\wedge)} \cdot \langle f, g \rangle)\!|\ \textbf{where}$$
$$\quad f = nr \cdot (\mathsf{fmap}\ \pi_1) \cdot lefts \cdot (\mathsf{fmap}\ distr)$$
$$\quad g = and \cdot rights \cdot (\mathsf{fmap}\ \pi_2)$$
$$groupByName :: (Eq\ a) \Rightarrow [(a,[b])] \to [[(a,[b])]]$$
$$groupByName = (groupBy\ (curry\ p))\ \textbf{where}$$
$$\quad p = \widehat{(\equiv)} \cdot (\pi_1 \times \pi_1)$$
$$filterPath :: (Eq\ a) \Rightarrow Path\ a \to [(Path\ a, b)] \to [(Path\ a, b)]$$
$$filterPath = filter \cdot (\lambda p \to \lambda(a,b) \to p \equiv a)$$

Dados para testes:

- Sistema de ficheiros vazio:

$$efs = FS\ []$$

- Nível 0

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

- Nível 1

$$f10 = mkdir\ (\texttt{"d1"}, f00)$$
$$f11 = concatFS\ (mkdir\ (\texttt{"d1"}, f00), mkdir\ (\texttt{"d2"}, f00))$$
$$f12 = concatFS\ (mkdir\ (\texttt{"d1"}, f00), mkdir\ (\texttt{"d2"}, f01))$$
$$f13 = concatFS\ (mkdir\ (\texttt{"d1"}, f00), mkdir\ (\texttt{"d2"}, efs))$$

- Nível 2

$$
\begin{aligned}
&f20 = mkdir\ (\texttt{"d1"}, f10)\\
&f21 = mkdir\ (\texttt{"d1"}, f11)\\
&f22 = mkdir\ (\texttt{"d1"}, f12)\\
&f23 = mkdir\ (\texttt{"d1"}, f13)\\
&f24 = concatFS\ (mkdir\ (\texttt{"d1"}, f10), mkdir\ (\texttt{"d2"}, f12))
\end{aligned}
$$

- Sistemas de ficheiros inválidos:

$$
\begin{aligned}
&ifs0 = concatFS\ (f1, f1)\\
&ifs1 = concatFS\ (f1, mkdir\ (\texttt{"f1"}, efs))\\
&ifs2 = mkdir\ (\texttt{"d1"}, ifs0)\\
&ifs3 = mkdir\ (\texttt{"d1"}, ifs1)\\
&ifs4 = concatFS\ (mkdir\ (\texttt{"d1"}, ifs1), mkdir\ (\texttt{"d2"}, f12))\\
&ifs5 = concatFS\ (mkdir\ (\texttt{"d1"}, f1), mkdir\ (\texttt{"d1"}, f2))\\
&ifs6 = mkdir\ (\texttt{"d1"}, ifs5)\\
&ifs7 = concatFS\ (mkdir\ (\texttt{"d1"}, f02), mkdir\ (\texttt{"d1"}, f02))
\end{aligned}
$$

Visualização em Graphviz:

$$
\begin{aligned}
&dotFS :: FS\ String\ b \to \mathsf{IO}\ ExitCode\\
&dotFS = dotpict \cdot bmap\ \underline{\texttt{""}}\ id \cdot (cFS2Exp\ \texttt{"root"})
\end{aligned}
$$

Outras funções auxiliares

Lógicas:

$$
\begin{aligned}
&\textbf{infixr}\ 0 \Rightarrow\\
&(\Rightarrow) :: (Testable\ prop) \Rightarrow (a \to Bool) \to (a \to prop) \to a \to Property\\
&p \Rightarrow f = \lambda a \to p\ a \Rightarrow f\ a\\
&\textbf{infixr}\ 0 \Leftrightarrow\\
&(\Leftrightarrow) :: (a \to Bool) \to (a \to Bool) \to a \to Property\\
&p \Leftrightarrow f = \lambda a \to (p\ a \Rightarrow property\ (f\ a))\ .\&\&.\ (f\ a \Rightarrow property\ (p\ a))\\
&\textbf{infixr}\ 4 \equiv\\
&(\equiv) :: Eq\ b \Rightarrow (a \to b) \to (a \to b) \to (a \to Bool)\\
&f \equiv g = \lambda a \to f\ a \equiv g\ a\\
&\textbf{infixr}\ 4 \leqslant\\
&(\leqslant) :: Ord\ b \Rightarrow (a \to b) \to (a \to b) \to (a \to Bool)\\
&f \leqslant g = \lambda a \to f\ a \leqslant g\ a\\
&\textbf{infixr}\ 4 \wedge\\
&(\wedge) :: (a \to Bool) \to (a \to Bool) \to (a \to Bool)\\
&f \wedge g = \lambda a \to ((f\ a) \wedge (g\ a))
\end{aligned}
$$

Compilação e execução dentro do interpretador:[8]

$$
run = \textbf{do}\ \{system\ \texttt{"ghc cp1819t"}; system\ \texttt{"./cp1819t"}\}
$$

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.

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

Problema 1

In, Out e Functor de Expr

$$inExpr :: Int + (Op, (Expr, Expr)) \to Expr$$
$$inExpr = [Num, (\widehat{\cdot} \cdot \widehat{flip\ Bop})]$$
$$outExpr :: Expr \to Int + (Op, (Expr, Expr))$$
$$outExpr\ (Num\ x) = i_1\ x$$
$$outExpr\ (Bop\ a\ x\ b) = i_2\ (x, (a, b))$$
$$recExpr\ f = id + (id \times (f \times f))$$

Catamorfismo, Anamorfismo e Hilomorfismo de Expr

```
  -- Catamorfismo de Expr
(|g|) = g . recExpr (|g|) . outExpr
  -- Anamorfismo de Expr
[(g)] = inExpr . recExpr [(g)] . g
  -- Hilomorfismo de Expr
(|f|) g = (|f|) . [(g)]
```

Resoluções

Resolução da função *calcula*

Para esta função, tal como especificado, recebe uma ou mais expressões aritméticas e, posteriormente retorna o valor relativo a essas expressões. Por exemplo, a expressão 4*(4+1) = 20.

```
calcAux :: (Op, (Int, Int)) -> Int
calcAux (x, (a, b)) | x == Op "+" = a + b
    | x == Op "*" = a * b
    | x == Op "-" = a - b
    | otherwise = a `div` b
calcula :: Expr -> Int
calcula = (|[id, calcAux]|)
```

Para o desenvolvimento desta função, decidimos criar uma função auxiliar onde fazemos o cálculo para uma expressão com dois inteiros e uma operação. Assim, verificamos qual o operador introduzido, e a partir deste é feito o cálculo com os dois inteiros. Sendo já sido feito o cálculo para uma expressão, agora na função calcula basta chamar a auxiliar tantas vezes quantas as operações. Para tal, decidimos usar um catamorfismo em que o seu gene é especificado para um número inteiro ou para uma expressão. Para testar esta expressão a forma correta de introduzir é, por exemplo, calcula (Bop (Num 3) (Op "+") (Num 4)).

Resolução da função *compile*

Este exercício trata da implementação de um compilador que gera o código introduzido para uma máquina género stack. Posto isto, aquando a introdução de um inteiro a sua representação é "PUSH inteiro" e para uma operação é realizada por extenso, por exemplo: compile "3+4" = ["PUSH 3", "PUSH 4", "ADD"]. Assim na função abaixo referida utilizamos um catamorfismo, onde o g1 é a representação de um número (["PUSH .."]) e o g2 a representação de uma ou mais expressões (["PUSH ..","PUSH ..","OP", ...].

```
compile :: String -> Codigo
compile = (|[g1, g2]|) . π1 . head . readExp
  where g1 a = ["PUSH " ++ show a]
    g2 (x, (a, b)) | x == Op "+" = a ++ b ++ ["ADD"]
        | x == Op "*" = a ++ b ++ ["MUL"]
        | x == Op "-" = a ++ b ++ ["SUB"]
        | otherwise = a ++ b ++ ["DIV"]
```

Resolução da função *show'*

Em relação a esta função, representa uma expressão do tipo Expr numa String. Assim sendo, fizemos tal conversão com o uso de um catamorfismo onde para g1 especificamos um inteiro (Num 3 = 3) e para g2 o caso de uma expressão completa com a respetiva operação entre dois inteiros. Por exemplo, com a introdução de uma expressão (Bop (Num 3) (Op "+") (Num 4)) é gerado um output "((3))+(4)".

```
show' :: Expr → String
show' = (|[g1, g2]|)
  where g1 a = showAux a
    g2 (Op a, (x, y)) | length (x) > 1 = "(" ++ x ++ ")" ++ a ++ y
       | length (y) > 1 = x ++ a ++ "(" ++ y ++ ")"
       | otherwise = "(  " ++ x ++ a ++ y ++ "  )"
showAux :: Int → String
showAux a | a < 0 = "(" ++ show a ++ ")"
  | otherwise = show a
```

Problema 2

In e Out de um L2D

```
inL2D :: a + (b, (X a b, X a b)) → X a b
inL2D = [Unid, comp] where comp (a, (b, c)) = Comp a b c
outL2D :: X a b → a + (b, (X a b, X a b))
outL2D (Unid a) = i1 a
outL2D (Comp a b c) = i2 (a, (b, c))
```

Anamorfismo, Catamorfismo e Hilomorfismo de uma L2D

```
  -- Functor
recL2D f = id + id × (f × f)
  -- Catamorfismo
(|g|) = g · recL2D (|g|) · outL2D
  -- Anamorfismo
[(g)] = inL2D · recL2D [(g)] · g
  -- Hylomorfismo
(|g|) h = (|g|) · [(h)]
```

Resoluções

Resolução da função *collectLeafs*

Como podemos ver na função seguinte decidiu-se utilizar um catamorfismo onde foi necessário uma função g1 que guarda uma Caixa numa lista e um g2 que agrega a Caixa às Caixas retiradas da estrutura de dados.

```
collectLeafs :: X a b → [a]
collectLeafs = (|[g1, g2]|)
  where g1 a = [a]
    g2 (b, (a, c)) = a ++ c
```

Resolução da função *dimen*

Relativamente a esta função é caracterizada por se utilizar um catamorfismo que trata da estrutura (X a b) onde existe um (Unid a) ou uma composição (Comp b (X a b) (X a b)) e cálcula as dimensões da Caixa ou das Caixas agregadas.

Relembrando a definição da Caixa:

$$Caixa = ((Int, Int), (Texto, G.Color))$$

Como se pode verificar no primeiro gene do catamorfismo, g1, são passadas as dimensões da Caixa de (Int,Int) para (Float,Float), já no caso do segundo gene, g2, são recebidas mais que uma Caixa e, para tal, é invocada a função compareDimen para calcular as agregações e, posteriormente, calcular as dimensões das agregações.

$$\begin{aligned}
&dimen :: X\ Caixa\ Tipo \to (Float, Float)\\
&dimen = (\!|[g1, g2]|\!)\\
&\quad \textbf{where}\ g1\ a = int2float\ (\pi_1\ a)\\
&\qquad g2\ (t, (b, c)) = compareDimen\ t\ b\ c
\end{aligned}$$

Resolução da função *compareDimen*

Como foi referido anteriormente, a função compareDimen, recebe uma combinação Comp b (X a b) (X a b), ou seja, recebe um Tipo, as dimensões de duas Caixas. Efetua o cálculo da agregação das duas Caixas segundo o Tipo recebido (V, Vd, Ve, H, Hb, Ht) e retorna as novas dimensões.

$$\begin{aligned}
&compareDimen :: Tipo \to (Float, Float) \to (Float, Float) \to (Float, Float)\\
&compareDimen\ V\ (a, b)\ (x, y) = \textbf{if}\ (a > x)\ \textbf{then}\ (a, b + y)\ \textbf{else}\ (x, b + y)\\
&compareDimen\ Vd\ (a, b)\ (x, y) = \textbf{if}\ (a > x)\ \textbf{then}\ (a, b + y)\ \textbf{else}\ (x, b + y)\\
&compareDimen\ Ve\ (a, b)\ (x, y) = \textbf{if}\ (a > x)\ \textbf{then}\ (a, b + y)\ \textbf{else}\ (x, b + y)\\
&compareDimen\ H\ (a, b)\ (x, y) = \textbf{if}\ (b > y)\ \textbf{then}\ (a + x, b)\ \textbf{else}\ (a + x, y)\\
&compareDimen\ Hb\ (a, b)\ (x, y) = \textbf{if}\ (b > y)\ \textbf{then}\ (a + x, b)\ \textbf{else}\ (a + x, y)\\
&compareDimen\ Ht\ (a, b)\ (x, y) = \textbf{if}\ (b > y)\ \textbf{then}\ (a + x, b)\ \textbf{else}\ (a + x, y)
\end{aligned}$$

Resolução da função *int2float*

Esta função foi elaborada para auxiliar à transformação das dimensões de uma caixa, que são valores inteiros, para floats.

$$\begin{aligned}
&int2float :: (Int, Int) \to (Float, Float)\\
&int2float\ (a, b) = (x, y)\\
&\quad \textbf{where}\ x = fromIntegral\ a :: Float\\
&\qquad y = fromIntegral\ b :: Float
\end{aligned}$$

Resolução da função *calcOrigins*

Tal como é pedido no enunciado, esta função tem o propósito de calcular as coordenadas de um conjunto de Caixas no plano, tendo em conta que a árvore de Caixas ficará com as folhas vazias no fim da execução da função. Para tal foi utilizado um anamorfismo que utiliza uma função g com os argumentos passados à calcOrigins. Esta função tem dois casos:

- No primeiro caso temos uma única caixa, logo vai-se aplicar a injeção 1 (i1), ficando (Caixa, Origem);
- No segundo caso temos um par de caixas, logo vai-se aplicar a injeção 2 (i2) colocando as folhas vazias e calculando a posição da próxima caixa. Para esse efeito, utiliza-se as dimensões das caixas obtidas através da função dimen e calculando o seu posicionamento consoante o tipo de agregação através da função calc.

```
calcOrigins :: ((X Caixa Tipo), Origem) → X (Caixa, Origem) ()
calcOrigins (c, o) = ⦇g⦈ (o, c)
  where g (o, (Unid a)) = i1 (a, o)
    g (o, (Comp a b c)) = i2 ((), ((o, b), (aux a, c)))
      where aux V = calc a z w
        aux Vd = calc a z w
        aux Ht = calc a z w
        aux H      | o ≡ (0.0, 0.0) = calc a z (0.0, π2 w)
                   | otherwise = calc a o z
        aux _      = calc a o z
        z = dimen b
        w = dimen c
```

Resolução da função *calc*

A função calc é uma das funções que auxilia a função calcOrigins e esta recebe como argumentos o Tipo da agregação, a origem da primeira Caixa e as dimensões da segunda Caixa. Deste modo para cada tipo de agregação, a função calcula as coordenadas (Origem) onde a segunda Caixa irá ficar colocada face a primeira Caixa.

```
calc :: Tipo → Origem → (Float, Float) → Origem
calc V (o1, o2) (x, y) = (n1, n2)
  where n1 = (o1 / 2) − (x / 2)
    n2 = o2
calc Vd (o1, o2) (x, y) = (n1, n2)
  where n1 = o2 − y
    n2 = o2
calc Ve (o1, o2) (x, y) = (n1, n2)
  where n1 = o1
    n2 = o2 + y
calc H (o1, o2) (x, y) | x ≡ 0.0 = (o1, (o2 / 2) − (y / 2))
  | (o1, o2) ≡ (0.0, 0.0) = (x, o2 + (y / 2))
  | otherwise             = (o1 + x, o2 + (y / 2))
calc Hb (o1, o2) (x, y) = (n1, n2)
  where n1 = x
    n2 = o2
calc Ht (o1, o2) (x, y) = (n1, n2)
  where n1 = o1
    n2 = o2 − y
```

Resolução da função *agrup_caixas*

Este exercício tem como âmbito a implementação de uma função com a finalidade de agrupa varias Caixas conforme as respetivas origens. Desta forma será possível construir o esquema das Caixas, isto é, a figura. Para construir esta função foi utilizado um catamorfismo onde g1 representa apenas uma Caixa (Origem,Caixa) e g2 um conjunto de tuplos (Origem,Caixa) representando várias Caixas.

```
agrup_caixas :: X (Caixa, Origem) () → Fig
agrup_caixas = (|[g1, g2]|)
  where g1 (a, b) = [(b, a)]
    g2 (b, (a, c)) = a ++ c
```

Resolução da função *caixas2Fig*

Esta função foi elaborada com o intuito de receber um tuplo com uma L2D seguida de uma Caixa seguido de um Tipo e uma Origem, de modo a converter este conjunto de Caixas numa Figura. Para tal esta função utiliza a calcOrigins que foi explicada anteriormente e utiliza a função agrupa_caixas com os dados gerados, permitindo gerar uma Figura como se pode ver no código seguinte.

$caixas2Fig :: ((X\ Caixa\ Tipo), Origem) \rightarrow Fig$
$caixas2Fig = agrup\_caixas \cdot calcOrigins$

Resolução da função *mostra_caixas*

Neste último exercício, o objetivo principal é partindo de uma frase da linguagem L2D e de uma Origem, ou seja, de um par de coordenadas iniciais, apresentar o desenho correspondente à agregação do conjunto de caixas da linguagem L2D no ecra. Para tal foi necessário desenvolver várias funções auxiliares que nos possibilita-se fazer o cálculo da Figura final. Primeiramente fornecemos a L2D e a Origem a função caixa2Fig que irá calcular a figura onde posteriormente, através da função fig2Pictures, a figura gerada vai ser convertida numa lista de G.Picture e, finalmente, irá ser feito o display (dado um valor do tipo G:picture abre uma janela com esse valor desenhado) dessa lista final para o ecra.

$mostra\_caixas :: (L2D, Origem) \rightarrow \mathsf{IO}\ ()$
$mostra\_caixas = display \cdot G.pictures \cdot fig2Pictures \cdot caixas2Fig$

Resolução da função *fig2Pictures*

A função fig2Pictures é uma função utilizada pela mostra_caixas com o intuito de transformar uma Figura numa lista com G.Picture. É uma função composta por dois casos: o caso em que não lhe foi fornecida figura e retorna uma lista vazia e, um segundo caso, onde recebe uma figura que é uma lista de tuplos (Origem,Caixa) em que recursivamente aplica a função crCaixaAux a cada tuplo da Figura.

$fig2Pictures :: Fig \rightarrow [G.Picture]$
$fig2Pictures\ [\,] = [\,]$
$fig2Pictures\ ((o, c) : t) = (crCaixaAux\ o\ (\pi_1\ c)\ (\pi_2\ c)) : (fig2Pictures\ t)$

Resolução da função *crCaixaAux*

É através desta função que a função fig2Pictures realiza os seus cálculos, isto é, a crCaixaAux recebe uma Origem, as coordenadas da Caixa e um tuplo composto pelo Texto e a G.Color (a cor da caixa) e cria a G.Picture. Com esta função é possível fornecer todos os dados necessários para a função fornecida crCaixa 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.

$crCaixaAux :: Origem \rightarrow (Int, Int) \rightarrow (String, G.Color) \rightarrow G.Picture$
$crCaixaAux\ o\ (x, y)\ (n, c) = crCaixa\ o\ a\ b\ n\ c$
  $\mathbf{where}\ a = fromIntegral\ x :: Float$
    $b = fromIntegral\ y :: Float$

Resolução da função *caixasAndOrigin2Pict*

Por fim, esta última função deste problema é uma sequência de funçoes criadas anteriormente. É recebido como argumento um tuplo (L2D,Origem) com o intuito de gerar uma G.Picture, para isso utiliza-se a função caixas2Fig de modo a gerar cada Figura para posteriormente convertê-las em G.Pictures através da função fig2Pictures.

$caixasAndOrigin2Pict :: ((X\ Caixa\ Tipo), Origem) \rightarrow G.Picture$
$caixasAndOrigin2Pict = G.pictures \cdot fig2Pictures \cdot caixas2Fig$

Problema 3

Considerando a fórmula que dá a série de Taylor da função coseno:

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

Substituindo o n por n+1 teremos:

$$cos\ x(n+1) = \frac{(-1)^{n+1}}{(2(n+1))!}x^{2(n+1)}$$

Obtemos assim h x n. Substituindo novamente o n por n+1 iremos obter o h x (n+1):

$$h\ x(n+1) = \frac{(-1)^{n+2}}{(2n+4)!}x^{2n+4} = \frac{(-1)^{n+1} * (-1) * x^{2n+2} * x^2}{(2n+2)! * (2n+3) * (2n+4)} = \frac{(-1)^{n+1}}{(2(n+1))!}x^{2(n+1)} * \frac{(-x)^2}{(2n+3) * (2n+4)}$$

Como resultado obtivemos de um lado o h x n e de outro resto do somatório inicial, daí é possível retirar:

$$s\ n = 2\ n + 3$$
$$s\ (n+1) = (2\ n+3) + 2$$
$$s\ 0 = 3$$
$$v\ n = 2\ n + 4$$
$$v\ (n+1) = (2\ n+4) + 2$$
$$v\ 0 = 4$$

Solução:

$$cos'\ x = prj \cdot \mathsf{for}\ loop\ init\ \mathbf{where}$$
$$loop\ (cs, h, s, v) = (h + cs, -x \uparrow 2\ /\ (s * v) * h, 2 + s, 2 + v)$$
$$init = (1, (-x \uparrow 2)\ /\ 2, 3, 4)$$
$$prj\ (cs, h, s, v) = cs$$

Problema 4

Triologia “ana-cata-hilo":

Out

$$outFS\ (FS\ l) = \mathsf{map}\ \ (id \times outNode)\ l$$
$$outNode\ (File\ b) = i_1\ b$$
$$outNode\ (Dir\ a) = i_2\ a$$

Anamorfismo, Catamorfismo e Hilomorfismo

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

-- Catamorfismo

$$(\!|\cdot|\!) :: ([(a, b + c)] \to c) \to FS\ a\ b \to c$$
$$(\!|g|\!) = g \cdot recFS\ (\!|g|\!) \cdot outFS$$

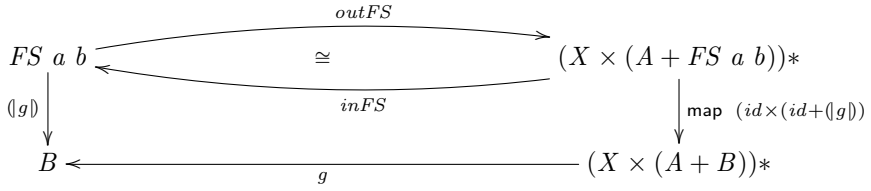
-- Anamorfismo

$$[\!(\cdot)\!] :: (c \to [(a, b + c)]) \to c \to FS\ a\ b$$
$$[\!(g)\!] = inFS \cdot recFS\ [\!(g)\!] \cdot g$$

-- Hylomorfismo

$$(\!|g|\!)\ h = (\!|g|\!) \cdot [\!(h)\!]$$

Diagrama de Catamorfismo



Outras funções pedidas:

Resolução da função *check*

A função check verifica a integridade do sistema de ficheiros, isto é, verifica que não existem identificadores repetidos dentro da mesma directoria. Primeiro foi criada a função checkRep de modo a verificar cada elemento de uma lista, retornando True caso não encontre repetidos ou retornando False caso encontre algum elemento repetido. A função check é composta por um catamorfismo onde aplicará a função anterior para ambos os lados da árvore, ou seja, percorrendo assim cada elemento da árvore (sistema de ficheiros), aplicando a função checkRep para verificar se o elemento em questão está repetido na árvore.

```
checkRep :: Eq a ⇒ [a] → Bool
checkRep [] = True
checkRep (h : t) = (¬ (elem h t)) ∧ (checkRep t)

check :: (Eq a) ⇒ FS a b → Bool
check = (|(checAux)|)
  where checAux [] = True
    checAux ((a, i1 b) : t) | (checkRep (map π1 ((a, i1 b) : t))) = True ∧ (checAux t)
      | otherwise = False
    checAux ((a, i2 b) : t) | b = True ∧ checAux t
      | otherwise = False
```

Resolução da função *tar*

Nesta função é necessário fazer a recolha do conteúdo de todos os ficheiros num arquivo indexado pelo path e, para tal, começamos por criar uma função auxiliar addToTuple que recebe um elemento e uma lista de tuplos em que o primeiro elemento de cada tuplo é uma lista e adiciona esse elemento ao longo da lista de tuplos. De seguida na função principal é composta por um catamorfismo onde no caso de ser o ramo da esquerda da árvore guarda o elemento numa lista, já no caso do ramo da direita aplica-se a função auxilar e concatena-se recursivamente o resultado obtido para o resto da lista.

```
addToTuple :: a → [([a], b)] → [([a], b)]
addToTuple _ [] = []
addToTuple x (([], _) : t) = addToTuple x t
addToTuple x ((l, a) : t) = ((x : l), a) : (addToTuple x t)

tar :: (Eq a) ⇒ FS a b → [(Path a, b)]
tar = (|(tarAux)|)
  where tarAux [] = []
    tarAux ((a, i1 b) : t) = ([a], b) : (tarAux t)
    tarAux ((a, i2 b) : t) = (addToTuple a b) ++ (tarAux t)
```

Resolução da função *untar*

Através desta função pertende-se transformar um arquivo com o conteúdo dos ficheiros indexado pelo path num sistema de ficheiros. Para isso decidiu-se utilizar a sugestão fornecida, ou seja, utilizar a função joinDupDirs para juntar directorias que estejam na mesma pasta e que possuam o mesmo identificador. Esta função baseada num anamorfismo irá realizar o inverso da função tar, usado para o efeito duas funções auxiliares, sendo que a primeira é usada no anamorfismo e a segunda é usada após o

anamorfismo (joinDupDirs). A função auxiliar faz com que isso aconteça devido a analisar todos os casos, ou seja, caso a lista seja vazia, continua vazia, caso a lista tenha apenas um elemento coloca-se o elemento à esquerda e concatena-se a lista do sistema de ficheiros, em último caso coloca-se o elemento à direita e concatena-se a lista do sistema de ficheiros.

```
untar :: (Eq a) ⇒ [(Path a, b)] → FS a b
untar = joinDupDirs · ⦇(untarAux)⦈
  where untarAux [] = []
    untarAux (((p : ps), b) : rl) | (ps ≡ []) = [(p, i1 b)] ++ untarAux rl
      | otherwise = [(p, i2 [(ps, b)])] ++ untarAux rl
```

Resolução da função *find*

Nesta função é necessário localizar todos os paths onde existe um determinado ficheiro e, para tal, começamos por criar uma função auxiliar addToList que irá adicionar um ficheiro a uma lista existente. Desta feita, utilizou-se um catamorfismo que verifica se caso a lista é vazia e mantém a lista vazia, verifica se caso o ficheiro é igual ao elemento à esquerda, mantendo-o e aplicando a função recursivamente ao resto da lista ou verifica se caso o elemento estiver à direta adiciona o elemento a lista e concatena com a chamada recursiva da função para o resto da lista.

```
addToList :: a → [[a]] → [[a]]
addToList _ [] = []
addToList x ([] : t) = (addToList x t)
addToList x (h : t) = (x : h) : (addToList x t)
find :: (Eq a) ⇒ a → FS a b → [Path a]
find a = (|(findAux a)|)
  where findAux file [] = []
    findAux file ((a, i1 b) : t) | (file ≡ a) = ([a]) : (findAux file t)
      | otherwise = findAux file t
    findAux file ((a, i2 b) : t) = (addToList a b) ++ (findAux file t)
```

Resolução da função *new*

```
new :: (Eq a) ⇒ Path a → b → FS a b → FS a b
new = ⊥
```

Resolução da função *cp*

```
cp :: (Eq a) ⇒ Path a → Path a → FS a b → FS a b
cp = ⊥
```

Resolução da função *rm*

```
rm :: (Eq a) ⇒ (Path a) → (FS a b) → FS a b
rm = ⊥
```

Resolução da função *auxJoin*

```
auxJoin :: ([(a, b + c)], d) → [(a, b + (d, c))]
auxJoin = ⊥
```

Resolução da função *cFS2Exp*

```
cFS2Exp :: a → FS a b → (Exp () a)
cFS2Exp = ⊥
```