



**Universidade do Minho**  
Escola de Engenharia

## **Cálculo de Programas**

### Trabalho Prático (2024/25)

Lic. em Engenharia Informática

#### **Grupo G2**

a104356 João d'Araújo Dias Lobo  
a90817 Mariana Rocha Cristino  
a104439 Rita da Cunha Camacho

## Preâmbulo

Em [Cálculo de Programas](#) pretende-se ensinar a programação de computadores como uma disciplina científica. Para isso parte-se de um repertório de *combinadores* que formam uma álgebra da programação e usam-se esses combinadores para construir programas *composicionalmente*, isto é, agregando programas já existentes.

Na sequência pedagógica dos planos de estudo dos cursos que têm esta disciplina, opta-se pela aplicação deste método à programação em [Haskell](#) (sem prejuízo da sua aplicação a outras linguagens funcionais). 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, a validá-los e a produzir textos técnico-científicos de qualidade.

Antes de abordarem os problemas propostos no trabalho, os grupos devem ler com atenção o anexo [A](#) onde encontrarão as instruções relativas ao *software* a instalar, etc.

Valoriza-se a escrita de *pouco* código que corresponda a soluções simples e elegantes que utilizem os combinadores de ordem superior estudados na disciplina.

## Problema 1

Esta questão aborda um problema que é conhecido pela designação '*H-index of a Histogram*' e que se formula facilmente:

*O h-index de um histograma é o maior número  $n$  de barras do histograma cuja altura é maior ou igual a  $n$ .*

Por exemplo, o histograma

$$h = [5, 2, 7, 1, 8, 6, 4, 9]$$

que se mostra na figura



tem *hindex*  $h = 5$  pois há 5 colunas maiores que 5. (Não é 6 pois maiores ou iguais que seis só há quatro.)

Pretende-se definida como um catamorfismo, anamorfismo ou hilomorfismo uma função em Haskell

$$\text{hindex} :: [Int] \rightarrow (Int, [Int])$$

tal que, para  $(i, x) = \text{hindex } h$ ,  $i$  é o H-index de  $h$  e  $x$  é a lista de colunas de  $h$  que para ele contribuem.

A proposta de *hindex* deverá vir acompanhada de um **diagrama** ilustrativo.

## Problema 2

Pelo [teorema fundamental da aritmética](#), todo número inteiro positivo tem uma única factorização prima. Por exemplo,

```
primes 455
[5,7,13]
primes 433
[433]
primes 230
[2,5,23]
```

1. Implemente como anamorfismo de listas a função

$primes :: \mathbb{Z} \rightarrow [\mathbb{Z}]$

que deverá, recebendo um número inteiro positivo, devolver a respectiva lista de factores primos.

A proposta de *primes* deverá vir acompanhada de um **diagrama** ilustrativo.

2. A figura mostra a “*árvore dos primos*” dos números [455, 669, 6645, 34, 12, 2].



Com base na alínea anterior, implemente uma função em Haskell que faça a geração de uma tal árvore a partir de uma lista de inteiros:

$prime\_tree :: [\mathbb{Z}] \rightarrow Exp\ \mathbb{Z}\ \mathbb{Z}$

**Sugestão:** escreva o mínimo de código possível em *prime\_tree* investigando cuidadosamente que funções disponíveis nas bibliotecas que são dadas podem ser reutilizadas.<sup>1</sup>

## Problema 3

A convolução  $a \star b$  de duas listas  $a$  e  $b$  — uma operação relevante em computação — está muito bem explicada [neste vídeo](#) do canal **3Blue1Brown** do YouTube, a partir de  $t = 6 : 30$ . Aí se mostra como, por exemplo:

<sup>1</sup> Pense sempre na sua produtividade quando está a programar — essa atitude será valorizada por qualquer empregador que vier a ter.

$$[1, 2, 3] \star [4, 5, 6] = [4, 13, 28, 27, 18]$$

A solução abaixo, proposta pelo chatGPT,

```
convolve :: Num a => [a] -> [a] -> [a]
convolve xs ys = [sum $ zipWith (*) (take n (drop i xs)) ys | i <- [0..(length xs - n)]]
  where n = length ys
```

está manifestamente errada, pois  $\text{convolve } [1, 2, 3] [4, 5, 6] = [32]$  (!).

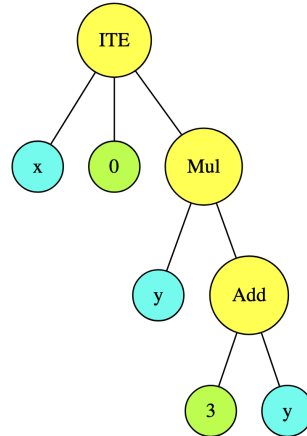
Proponha, explicando-a devidamente, uma solução sua para *convolve*. Valorizar-se-á a economia de código e o recurso aos combinadores *pointfree* estudados na disciplina, em particular a triologia *ana-cata-hilo* de tipos disponíveis nas bibliotecas dadas ou a definir.

## Problema 4

Considere-se a seguinte sintaxe (abstrata e simplificada) para **expressões numéricas** (em *b*) com variáveis (em *a*),

```
data Expr b a = V a | N b | T Op [Expr b a] deriving (Show, Eq)
data Op = ITE | Add | Mul | Suc deriving (Show, Eq)
```

possivelmente condicionais (cf. *ITE*, i.e. o operador condicional “if-then-else”). Por exemplo, a árvore mostrada a seguir



representa a expressão

$$\text{ite } (V \text{ "x"}) (N \ 0) (\text{multi } (V \text{ "y"}) (\text{soma } (N \ 3) (V \text{ "y"}))) \quad (1)$$

– i.e. **if** *x* **then** 0 **else** *y* \* (3 + *y*) – assumindo as “helper functions”:

```
soma x y = T Add [x, y]
multi x y = T Mul [x, y]
ite x y z = T ITE [x, y, z]
```

No anexo E propõe-se uma base para o tipo *Expr* (*baseExpr*) e a correspondente algebra *inExpr* para construção do tipo *Expr*.

1. Complete as restantes definições da biblioteca *Expr* pedidas no anexo F.
2. No mesmo anexo, declare *Expr b* como instância da classe *Monad*. **Sugestão:** relembre os exercícios da ficha 12.

3. Defina como um catamorfismo de *Expr* a sua versão monádica, que deverá ter o tipo:

$$mcataExpr :: Monad\ m \Rightarrow (a + (b + (Op, m\ [c])) \rightarrow m\ c) \rightarrow Expr\ b\ a \rightarrow m\ c$$

4. Para se avaliar uma expressão é preciso que todas as suas variáveis estejam instanciadas. Complete a definição da função

$$let\_exp :: (Num\ c) \Rightarrow (a \rightarrow Expr\ c\ b) \rightarrow Expr\ c\ a \rightarrow Expr\ c\ b$$

que, dada uma expressão com variáveis em *a* e uma função que a cada uma dessas variáveis atribui uma expressão ( $a \rightarrow Expr\ c\ b$ ), faz a correspondente substituição.<sup>1</sup> Por exemplo, dada

$$\begin{aligned} f\ "x" &= N\ 0 \\ f\ "y" &= N\ 5 \\ f\ \_ &= N\ 99 \end{aligned}$$

ter-se-á

$$let\_exp\ f\ e = T\ ITE\ [N\ 1, N\ 0, T\ Mul\ [N\ 5, T\ Add\ [N\ 3, N\ 1]]]$$

isto é, a árvore da figura a seguir:



5. Finalmente, defina a função de avaliação de uma expressão, com tipo

$$evaluate :: (Num\ a, Ord\ a) \Rightarrow Expr\ a\ b \rightarrow Maybe\ a$$

que deverá ter em conta as seguintes situações de erro:

- (a) *Variáveis* — para ser avaliada, *x* em *evaluate x* não pode conter variáveis. Assim, por exemplo,

$$\begin{aligned} evaluate\ e &= Nothing \\ evaluate\ (let\_exp\ f\ e) &= Just\ 40 \end{aligned}$$

para *f* e *e* dadas acima.

- (b) *Aridades* — todas as ocorrências dos operadores deverão ter o devido número de sub-expressões, por exemplo:

$$\begin{aligned} evaluate\ (T\ Add\ [N\ 2, N\ 3]) &= Just\ 5 \\ evaluate\ (T\ Mul\ [N\ 2]) &= Nothing \end{aligned}$$

<sup>1</sup> Cf. expressões **let ... in...**

**Sugestão:** de novo se insiste na escrita do mínimo de código possível, tirando partido da riqueza estrutural do tipo *Expr* que é assunto desta questão. Sugere-se também o recurso a diagramas para explicar as soluções propostas.

## Anexos

### A Natureza do trabalho a realizar

Este trabalho teórico-prático deve ser realizado por grupos de 3 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 em **todos** os exercícios do trabalho, para assim poderem responder a qualquer questão colocada na *defesa oral* do relatório.

Para cumprir de forma integrada os objectivos do trabalho 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 `cp2425t.pdf` que está a ler é já um exemplo de [programação literária](#): foi gerado a partir do texto fonte `cp2425t.lhs`<sup>1</sup> que encontrará no [material pedagógico](#) desta disciplina descompactando o ficheiro `cp2425t.zip`.

Como se mostra no esquema abaixo, de um único ficheiro (*lhs*) gera-se um PDF ou faz-se a interpretação do código [Haskell](#) que ele inclui:



Vê-se assim que, para além do [GHCi](#), serão necessários os executáveis [pdflatex](#) e [lhs2TeX](#). Para facilitar a instalação e evitar problemas de versões e conflitos com sistemas operativos, é recomendado o uso do [Docker](#) tal como a seguir se descreve.

### B Docker

Recomenda-se o uso do [container](#) cuja imagem é gerada pelo [Docker](#) a partir do ficheiro `Dockerfile` que se encontra na diretoria que resulta de descompactar `cp2425t.zip`. Este [container](#) deverá ser usado na execução do [GHCi](#) e dos comandos relativos ao [L<sup>A</sup>T<sub>E</sub>X](#). (Ver também a `Makefile` que é disponibilizada.)

---

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

Após [instalar o Docker](#) e descarregar o referido zip com o código fonte do trabalho, basta executar os seguintes comandos:

```
$ docker build -t cp2425t .  
$ docker run -v ${PWD}:/cp2425t -it cp2425t
```

**NB:** O objetivo é que o container seja usado *apenas* para executar o [GHCi](#) e os comandos relativos ao [L<sup>A</sup>T<sub>E</sub>X](#). Deste modo, é criado um *volume* (cf. a opção `-v ${PWD}:/cp2425t`) que permite que a diretoria em que se encontra na sua máquina local e a diretoria `/cp2425t` no [container](#) sejam partilhadas.

Pretende-se então que visualize/edite os ficheiros na sua máquina local e que os compile no [container](#), executando:

```
$ lhs2TeX cp2425t.lhs > cp2425t.tex  
$ pdflatex cp2425t
```

[lhs2TeX](#) é o pre-processor que faz “pretty printing” de código Haskell em [L<sup>A</sup>T<sub>E</sub>X](#) e que faz parte já do [container](#). Alternativamente, basta executar

```
$ make
```

para obter o mesmo efeito que acima.

Por outro lado, o mesmo ficheiro `cp2425t.lhs` é executável e contém o “kit” básico, escrito em [Haskell](#), para realizar o trabalho. Basta executar

```
$ ghci cp2425t.lhs
```

Abra o ficheiro `cp2425t.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}
```

é seleccionado pelo [GHCi](#) para ser executado.

## C Em que consiste o TP

Em que consiste, então, o *relatório* a que se referiu acima? É a edição do texto que está a ser lido, preenchendo o anexo [F](#) 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 [Bib<sub>T</sub>E<sub>X</sub>](#)) e o índice remissivo (com [makeindex](#)),

```
$ bibtex cp2425t.aux  
$ makeindex cp2425t.idx
```

e recompilar o texto como acima se indicou. (Como já se disse, pode fazê-lo correndo simplesmente `make` no [container](#).)

No anexo [E](#) disponibiliza-se algum código [Haskell](#) relativo aos problemas que são colocados. Esse anexo deverá ser consultado e analisado à medida que isso for necessário.

Deve ser feito uso da [programação literária](#) para documentar bem o código que se desenvolver, em particular fazendo diagramas explicativos do que foi feito e tal como se explica no anexo [D](#) que se segue.

## D Como exprimir cálculos e diagramas em LaTeX/lhs2TeX

Como primeiro exemplo, estudar o texto fonte ([lhs](#)) do que está a ler<sup>1</sup> onde se obtém o efeito seguinte:<sup>2</sup>

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

Os diagramas podem ser produzidos recorrendo à *package* [xymatrix](#), por exemplo:

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

## E Código fornecido

### Problema 1

$h :: [Int]$

### Problema 4

Definição do tipo:

$inExpr = [V, [N, \widehat{T}]]$   
 $baseExpr\ g\ h\ f = g + (h + id \times \text{map}\ f)$

Exemplos de expressões:

$e = ite\ (V\ "x")\ (N\ 0)\ (multi\ (V\ "y")\ (soma\ (N\ 3)\ (V\ "y")))$   
 $i = ite\ (V\ "x")\ (N\ 1)\ (multi\ (V\ "y")\ (soma\ (N\ (3 / 5))\ (V\ "y")))$

Exemplo de teste:

$teste = evaluate\ (let\_exp\ f\ i) \equiv Just\ (26 / 245)$   
**where**  $f\ "x" = N\ 0; f\ "y" = N\ (1 / 7)$

<sup>1</sup> Procure e.g. por "sec:diagramas".

<sup>2</sup> Exemplos tirados de [?].



## F Soluções dos alunos

Os alunos devem colocar neste anexo as suas soluções para os 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 ao anexo, bem como diagramas e/ou outras funções auxiliares que sejam necessárias.

**Importante:** Não pode ser alterado o texto deste ficheiro fora deste anexo.

### Problema 1

```

hindex = hyloBTree [f1,f2] qsep
f1 = 0, []
f2 :: (Int, ((Int, [Int]), (Int, [Int]))) → (Int, [Int])
f2 (n, ((-, ll), (-, lr))) = (hIndex, contributors)
  where
    list = lr ++ [n] ++ ll
    hIndex = myfoldr process 0 (zip [1..] list)
    process :: (Ord a) ⇒ ((a, a), a) → a
    process = cond ((≥) · swap · π1) (max · swap · (π1 × id)) π2
    contributors = filter (≥ hIndex) list

```

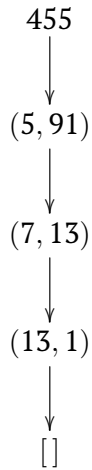
### Problema 2

A função `primes` é responsável por criar a lista de fatores primos de um dado número. De modo que, esta função pode ser definida como um anamorfismo de listas (*List*). Assim, o diagrama que representa a operação é o seguinte:

$$\begin{array}{ccc}
 \mathbb{Z} & \xrightarrow{g} & 1 + \mathbb{Z} \times \mathbb{Z} \\
 \llbracket g \rrbracket \downarrow & \text{in}_{List} \swarrow & \downarrow id + (id \times \llbracket g \rrbracket) \\
 \mathbb{Z}^* & \xleftarrow{out_{List}} & 1 + \mathbb{Z} \times \mathbb{Z}^*
 \end{array}$$

A implementação baseia-se em decompor o número repetidamente no seu menor fator primo, este processo repete-se até que o quociente resultante seja 1.

O processo pode ser representado graficamente como se segue para o número 455:



Assim,  $\text{primes } 455 = [5, 7, 13]$ .

A definição de *primes* como  $\llbracket g \rrbracket$  tira partido de que um anamorfismo constrói uma estrutura recursiva ao aplicar sucessivamente o gene  $g$  a um valor inicial. O gene  $g$  determina como cada passo da construção ocorre, neste caso  $g$  divide o número  $n$  no seu menor fator primo (calculado pela função *smallestPrimeFactor*) e no quociente resultante após a divisão. O processo termina quando  $n = 1$ , porque não existem mais fatores primos para serem determinados.

A função *smallestPrimeFactor* é responsável por determinar o menor fator primo de um número  $n$ , e é definida como um catamorfismo de naturais (*catNat*). Esta função aplica sucessivamente a lógica de "testar se um divisor  $d$  divide  $n$ " para valores  $d$  crescentes, assim inicia com o menor número primo (2).

O ciclo-for contém uma estrutura recursiva que verifica duas condições:

1. **Teste de primalidade:** Se  $d^2 > n$ : Nesse caso,  $n$  é primo e o seu menor fator primo é ele mesmo (o processo termina).

2. **Encontrar o menor fator primo:** Se  $n \bmod d = 0$ : Nesse caso,  $d$  é o menor fator primo de  $n$ .

**Caso contrário:** Incrementámos  $d$  e continuámos o processo.

**Fundamentação matemática:** A implementação baseia-se no Teorema Fundamental da Aritmética, que garante que todo o número inteiro positivo maior que 1 pode ser decomposto de forma única como um produto de fatores primos. O processo descrito no gene  $g$  utiliza esta propriedade para decompor iterativamente o  $n$  nos seus fatores primos, onde a divisibilidade é verificada e avançamos na procura do menor fator primo.

Primeira parte:

$$\begin{aligned}
 \text{smallestPrimeFactor } x &= \text{for } \lambda n \rightarrow \text{cond } ((\widehat{>}) \cdot ((\uparrow 2) \times \text{id})) \pi_2 \\
 &\quad (\text{cond } ((\equiv 0) \cdot \widehat{\text{mod}} \cdot \text{swap}) \pi_1 (\text{succ} \cdot \pi_1)) (n, x) \ 2 \ x \\
 g \ 1 &= i_1 \ () \\
 g \ n &= i_2 \ (\text{smallestPrimeFactor } n, n \div \text{smallestPrimeFactor } n) \\
 \text{primes} &= \llbracket g \rrbracket
 \end{aligned}$$

A função *prime\_tree* é responsável por criar a árvore dos primos de uma lista de inteiros, como se encontra ilustrado no enunciado. De modo que, esta função pode ser definida da seguinte forma:

Segunda parte:

$$prime\_tree = Term\ 1 \cdot untar \cdot map\ (\lambda n \rightarrow (primes\ n, n))$$

Inicialmente, adotamos uma abordagem extensiva para resolver o problema, com a definição de um hilomorfismo e todas as operações necessárias para construir a árvore. No entanto, durante este processo, reparámos na função *untar* da biblioteca *Exp.hs*, que efetua a operação necessária para transformar uma lista de pares numa estrutura do tipo  $[Exp\ v\ o]$ . Após compreendermos o comportamento e a definição da função *untar*, percebemos que era possível utilizá-la na construção da função *prime\_tree*, o que simplificou a implementação.

Explicação da função *prime\_tree*:

1. A função *primes* é aplicada a cada elemento da lista de inteiros e com o uso da expressão  $map\ (\lambda n \rightarrow (primes\ n, n))$ , obtemos uma lista de pares, onde o primeiro elemento é a lista de fatores primos de um número e o segundo elemento é o próprio número. Assim, no final da execução desta expressão, obtemos uma lista de pares do tipo  $[[[Z], Z]]$ .

2. Neste contexto, a função *untar* converte os fatores primos de um número e o próprio número numa representação de árvore onde os nós intermediários são os fatores e as folhas são os números originais,  $[Exp\ Z\ Z]$ . Esta conversão é realizada em três partes principais: a coalgebra, a base e a álgebra.

2.1. A coalgebra, representada pela função *c*, é responsável por decompor os dados, ou seja, separa os pares da lista inicial -  $[[[Z], Z]]$  - e transforma cada elemento para o formato  $Z + (Z, [[Z], Z])$ .

2.1.1. O  $map\ ((\pi_2 \dashv\!\!\! \dashv\! -\! assocr) \cdot distl \cdot (outList \times id))$  é aplicado a cada par da lista, onde:

- $outList \times id$  transforma a lista de fatores primos num tipo  $\cdot + \cdot$  e retorna o número original. Permitindo tratar em separado os fatores primos e os números.
- $distl$  distribui os elementos  $(a, b) + \cdot$  para o tipo  $(a + b, b)$ , separa os dados para facilitar o processamento posterior.
- $\pi_2 \dashv\!\!\! \dashv\! -\! assocr$  reorganiza os pares para agrupar corretamente os fatores primos associados a um número.

2.1.2. *sep* percorre a lista de elementos  $\cdot + \cdot$ , e separa os elementos  $i_1$  para o primeiro grupo e os  $i_2$  para o segundo grupo.

2.1.3.  $id \times collect$  aplica a função *id* ao primeiro valor do tuplo e *collect* ao segundo, de modo que a função *collect* é responsável por agrupar os fatores primos em listas separadas para cada número. Então, os números que partilham o mesmo fator primo são agrupados juntos.

2.1.4. *join* junta os valores numa lista única, recriando a estrutura necessária para representare os dados.

2.2. Após a coalgebra, avançamos para a base, esta aplica recursivamente a função *untar* a cada sublista e cria subárvores para cada conjunto de fatores. O tipo da função *base* é definido como:

$$base :: (Z \rightarrow Z) \rightarrow (Z \rightarrow Z) \rightarrow ([([Z], Z)] \rightarrow [Exp\ Z\ Z]) \rightarrow [Z + (Z, [[Z], Z])] \rightarrow [Z + (Z, [Exp\ Z\ Z])].$$

2.3. Para finalizar, a álgebra *a* organiza os dados processados na estrutura final, a operação *sort* organiza os nós, enquanto que o  $map\ inExp$  converte os elementos numa estrutura do tipo  $Exp\ Z\ Z$ . O seu tipo, neste contexto, é definido como:  $a :: [Z + (Z, [Exp\ Z\ Z])] \rightarrow [Exp\ Z\ Z]$ .

3. Por fim, a função *Term 1* é aplicada para adicionar a raiz da árvore com o valor 1, isto conecta todas as subárvores criadas pela função *untar*, construindo uma única árvore que representa a decomposição de todos os números da lista.

### Problema 3

$convolve :: Num\ a \Rightarrow [a] \rightarrow [a] \rightarrow [a]$   
 $convolve = \perp$

### Problema 4

Definição do tipo:

$outExpr :: Expr\ b\ a \rightarrow a + (b + (Op, [Expr\ b\ a]))$   
 $outExpr\ (V\ n) = i_1\ n$   
 $outExpr\ (N\ n) = (i_2 \cdot i_1)\ n$   
 $outExpr\ (T\ op\ exprs) = (i_2 \cdot i_2)\ (op, exprs)$   
 $recExpr :: (a \rightarrow b1) \rightarrow b2 + (b3 + (b4, [a])) \rightarrow b2 + (b3 + (b4, [b1]))$   
 $recExpr = baseExpr\ id\ id$

Ana + cata + hylo:

$cataExpr\ g = g \cdot recExpr\ (cataExpr\ g) \cdot outExpr$   
 $anaExpr\ g = inExpr \cdot recExpr\ (anaExpr\ g) \cdot g$   
 $hyloExpr\ h\ g = cataExpr\ h \cdot anaExpr\ g$

Monad:

Para declarar  $Expr\ b$  como instância da classe *Monad*, foram implementadas as instâncias de *Functor*, *Applicative* e *Monad* do tipo  $Expr\ b$ . A abordagem utilizada foi guiada pelo exercício 4 da ficha 12, adaptando ao contexto específico de  $Expr\ b$ .

Começamos pelo *Functor*, onde a função *fmap* aplica uma transformação às variáveis da expressão, mantendo as restantes estruturas (*N* e *T*) inalteradas. Esta operação é realizada de forma recursiva usando o catamorfismo - *cataExpr* -, que reconstrói a expressão após aplicar a *f* às variáveis. A composição com a função *inExpr* e a base do catamorfismo (*baseExpr*) assegura que a estrutura é processada corretamente.

**instance** *Functor* ( $Expr\ b$ )  
**where**  $fmap\ f = cataExpr\ (inExpr \cdot baseExpr\ f\ id\ id)$

De seguida, definimos a instância *Applicative*, onde a função *pure* cria uma expressão com uma variável (*V*) com o valor fornecido, a função  $\langle * \rangle$  considera três casos:

- se a expressão é uma variável ( $V\ f$ ), aplica *fmap* para mapear função sobre a outra expressão;
- se a expressão é um número ( $N\ b$ ), mantém o número inalterado;
- se a expressão é uma operação ( $T\ op\ fs$ ), aplica a função a cada subexpressão da operação.

**instance** *Applicative* ( $Expr\ b$ ) **where**  
 $pure :: a \rightarrow Expr\ b\ a$   
 $pure = V$   
 $(V\ f) \langle * \rangle x = fmap\ f\ x$   
 $(N\ b) \langle * \rangle \_ = N\ b$   
 $(T\ op\ fs) \langle * \rangle x = T\ op\ (map\ (\langle * \rangle x)\ fs)$

Por fim, a instância *Monad* foi definida, a definição *return* é equivalente a *pure*, cria uma variável. A operação  $\gg$  aplica a função *g* a cada variável da expressão, usando a função auxiliar *muExpr* para processar a expressão resultante.

```
instance Monad (Expr b) where
  return :: a → Expr b a
  return = pure
  (≫) :: Expr b a → (a → Expr b b1) → Expr b b1
  t ≫ g = muExpr (fmap g t)
  muExpr :: Expr b (Expr b a) → Expr b a
  muExpr = cataExpr [id, inExpr · i2]
  u :: a → Expr b a
  u = V
```

Provemos que *Expr b* é uma instância de *Monad*:

- $u = V$  e  $V = \text{inExpr} \cdot i_1$ , logo  $u = \text{inExpr} \cdot i_1$ ;
- $\text{muExpr} = \text{cataExpr} [\text{id}, \text{inExpr} \cdot i_2]$ .

Provar a lei monádica Unidade (63):

$$\begin{aligned}
& \text{mu} \cdot u = \text{mu} \cdot T u \\
\equiv & \quad \{ \text{definição de mu; definição de u} \} \\
& \llbracket [\text{id}, \text{in} \cdot i_2] \rrbracket \cdot \text{in} \cdot i_1 = \llbracket [\text{id}, \text{in} \cdot i_2] \rrbracket \cdot T u \\
\equiv & \quad \{ \text{absorção-cata} \} \\
& \llbracket [\text{id}, \text{in} \cdot i_2] \rrbracket \cdot \text{in} \cdot i_1 = \llbracket [\text{id}, \text{in} \cdot i_2] \cdot B(u, \text{id}) \rrbracket \\
\equiv & \quad \{ B(f,g) = f + G g, \text{absorção-+}, \text{natural-id}, \text{functor-id-F} \} \\
& \llbracket [\text{id}, \text{in} \cdot i_2] \rrbracket \cdot \text{in} \cdot i_1 = \llbracket [u, \text{in} \cdot i_2] \rrbracket \\
\equiv & \quad \{ \text{definição de u, cancelamento-cata} \} \\
& [\text{id}, \text{in} \cdot i_2] \cdot F \text{mu} \cdot i_1 = \llbracket [\text{in} \cdot i_1, \text{in} \cdot i_2] \rrbracket \\
\equiv & \quad \{ \text{fusão-+}, \text{reflexão-+}, \text{reflexão-cata}, \text{base-cata}, B(\text{id}, \text{mu}) = \text{id} + G \text{mu} \} \\
& [\text{id}, \text{in} \cdot i_2] \cdot (\text{id} + G \text{mu}) \cdot i_1 = \text{id} \\
\equiv & \quad \{ \text{natural-i1}, \text{natural-id} \} \\
& [\text{id}, \text{in} \cdot i_2] \cdot i_1 = \text{id} \\
\equiv & \quad \{ \text{cancelamento-+} \} \\
& \text{id} = \text{id} \\
\equiv & \quad \{ \text{P.R.I.} \} \\
& \text{True} \\
& \square
\end{aligned}$$

Provar a lei monádica Multiplicação (62):

$$\begin{aligned}
& \text{mu} \cdot \text{mu} = \text{mu} \cdot T \text{mu} \\
\equiv & \quad \{ \text{definição de mu} \} \\
& \text{mu} \cdot \text{mu} = \llbracket [\text{id}, \text{in} \cdot i_2] \rrbracket \cdot T \llbracket [\text{id}, \text{in} \cdot i_2] \rrbracket
\end{aligned}$$

$$\begin{aligned}
&\equiv \{ \text{absorção-cata} \} \\
&\quad mu \cdot mu = \langle [id, \mathbf{in} \cdot i_2] \cdot (\langle [id, \mathbf{in} \cdot i_2] \rangle + G id) \rangle \\
&\equiv \{ \text{Functor-id-F, natural-id, absorção-+} \} \\
&\quad mu \cdot mu = \langle \langle [id, \mathbf{in} \cdot i_2] \rangle, \mathbf{in} \cdot i_2 \rangle \\
&\equiv \{ \text{definição de mu} \} \\
&\quad mu \cdot \langle [id, \mathbf{in} \cdot i_2] \rangle = \langle \langle [id, \mathbf{in} \cdot i_2] \rangle, \mathbf{in} \cdot i_2 \rangle \\
&\Leftarrow \{ \text{fusão-cata} \} \\
&\quad mu \cdot [id, \mathbf{in} \cdot i_2] = \langle \langle [id, \mathbf{in} \cdot i_2] \rangle, \mathbf{in} \cdot i_2 \rangle \cdot (id + G mu) \\
&\equiv \{ \text{fusão-+, absorção-+, natural-id, eq-+} \} \\
&\quad [id, \mathbf{in} \cdot i_2] \cdot i_1 = id \\
&\equiv \{ \text{cancelamento-+} \} \\
&\quad \begin{cases} mu = mu \\ mu \cdot \mathbf{in} \cdot i_2 = \mathbf{in} \cdot i_2 \cdot G mu \end{cases} \\
&\equiv \{ p \ \& \ \text{True} = \text{True}, \text{definição de mu, cancelamento-cata, base-cata} \} \\
&\quad [id, \mathbf{in} \cdot i_2] \cdot (id + G mu) \cdot i_2 = \mathbf{in} \cdot i_2 \cdot G mu \\
&\equiv \{ \text{natural-i2, cancelamento-+} \} \\
&\quad \mathbf{in} \cdot i_2 \cdot G mu = \mathbf{in} \cdot i_2 \cdot G mu \\
&\equiv \{ \text{P.R.I} \} \\
&\quad \text{True} \\
&\square
\end{aligned}$$

*Maps: Monad: Let expressions:*

A função *let\_exp* é responsável por substituir todas as variáveis numa expressão *Expr* pelas suas correspondentes expressões atribuídas por uma função fornecida como argumento.

A definição da função *let\_exp* utiliza o catamorfismo *cataExpr*. No caso de encontrar uma variável (*V*), a função *f* é usada para substituir essa variável pela expressão correspondente. Para valores numéricos (*N*), a função mantém o valor inalterado. Para operadores (caso *T*), a função constrói uma nova expressão que combina os resultados das subexpressões recursivamente.

Em suma, *let\_exp* avalia uma expressão ao substituir todas as variáveis pelas expressões correspondentes, garantindo que a estrutura da expressão original é mantida. De seguida, o diagrama mostra como a operação do catamorfismo percorre e transforma a expressão.

$$\begin{array}{ccc}
Expr \ C \ A & \xleftarrow{\mathbf{in}_{Expr}} & A + (C + (Op \times (Expr \ C \ A)^*)) \\
\downarrow \text{let\_exp } f & \xrightarrow{\text{out}_{Expr}} & \downarrow id + (id + (id \times \text{map } (\text{let\_exp } f))) \\
Expr \ C \ B & \xleftarrow{[f, [N, \hat{T}]}} & A + (C + (Op \times (Expr \ C \ B)^*))
\end{array}$$

$$let\_exp \ f = cataExpr \ [f, [N, \hat{T}]]$$

Catamorfismo monádico:

$$\begin{aligned}
mcataExpr\ g &= g \cdot (dl' \cdot recExpr\ (mcataExpr\ g) \cdot outExpr) \\
dl' &:: Monad\ m \Rightarrow a + (b + (Op, [m\ c])) \rightarrow m\ (a + (b + (Op, m\ [c]))) \\
dl' &= [return \cdot i_1, [return \cdot i_2 \cdot i_1, aux]] \\
\textbf{where}\ aux\ (op, ms) &= \textbf{do}\ m \leftarrow \textit{lamb}\ ms; (return \cdot i_2 \cdot i_1)\ (op, return\ m)
\end{aligned}$$

Avaliação de expressões:

A função *evaluate* avalia expressões do tipo *Expr* e retorna o resultado da avaliação com o tipo *Maybe*. Esta função tem em conta dois cenários de erro: variáveis não instanciadas e operadores aplicados a um número incorreto de argumentos.

A função *evaluate* utiliza o catamorfismo *mcataExpr* para avaliar a expressão. Este conceito generaliza o conceito de catamorfismo simples para permitir trabalhar em contextos monádicos. O ponto central deste conceito é que combina a lógica de transformação (*g*) com a recursão da estrutura, enquanto lida automaticamente com contextos monádicos. Assim, evitámos tratar manualmente de cada contexto monádico *Maybe* em cada passo.

No caso do *evaluate*, o gene *gene* define como é que se processa cada nó da estrutura *Expr*. O gene *gene* lida com três casos principais:

- *V* : Para uma variável, retornamos *Nothing*, porque as variáveis não podem ser avaliadas.
- *N* : Para um número, retornamos o próprio número com *Just*.
- *T* : Para um operador, utilizamos a função auxiliar *aux* que:
  - avalia todos os argumentos no contexto *Maybe*, isto é, verifica se todos os argumentos são válidos;
  - aplica a função *result* para calcular o resultado final, caso todos os argumentos sejam válidos.

A função *result* define o comportamento esperado para cada operador e valida a aridade, assim garante que apenas os operadores com a aridade correta sejam avaliados. Caso contrário, a avaliação falha e retorna *Nothing*.

$$\begin{aligned}
evaluate &= mcataExpr\ gene \\
gene &:: (Num\ a, Ord\ a) \Rightarrow b + (a + (Op, Maybe\ [a])) \rightarrow Maybe\ a \\
gene &= [Nothing, [Just, aux]] \\
\textbf{where}\ aux\ (op, args) &= \textbf{do}\ argsR \leftarrow args; result\ op\ argsR \\
result &:: (Num\ a, Ord\ a) \Rightarrow Op \rightarrow [a] \rightarrow Maybe\ a \\
result\ Add\ [x, y] &= Just\ (x + y) \\
result\ Mul\ [x, y] &= Just\ (x * y) \\
result\ Suc\ [x] &= Just\ (x + 1) \\
result\ ITE\ [cond, t, f] &= \textbf{if}\ cond \neq 0 \textbf{ then}\ Just\ t \textbf{ else}\ Just\ f \\
result\ \_ \_ &= Nothing
\end{aligned}$$

# Index

$\LaTeX$ , [3](#), [4](#)

**bibtex**, [4](#)

**lhs2TeX**, [3–5](#)

**makeindex**, [4](#)

**pdflatex**, [3](#)

**xymatrix**, [5](#)

Cálculo de Programas, [1](#), [3](#)

    Material Pedagógico, [3](#)

Combinador “pointfree”

*cata*

        Naturais, [5](#)

*split*, [5](#)

Docker, [3](#)

    container, [3](#), [4](#)

Função

$\pi_1$ , [5](#)

$\pi_2$ , [5](#)

Haskell, [1](#), [3](#), [4](#)

    interpretador

        GHCi, [3](#), [4](#)

    Literate Haskell, [3](#)

Números naturais ( $\mathbb{N}$ ), [5](#)

Programação

    literária, [3](#), [4](#)



## 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, 2024. Draft of textbook in preparation. First version: 1998. Current version: Sep. 2024. Informatics Department, University of Minho ([pdf](#)).