



Universidade do Minho
Escola de Engenharia

Cálculo de Programas

Trabalho Prático (2024/25)

Lic. em Engenharia Informática

Grupo G05

a103993 Júlia Bughi Corrêa da Costa
a104171 Gabriel Pereira Ribeiro
a104613 Luís Pinto da Cunha

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]
```

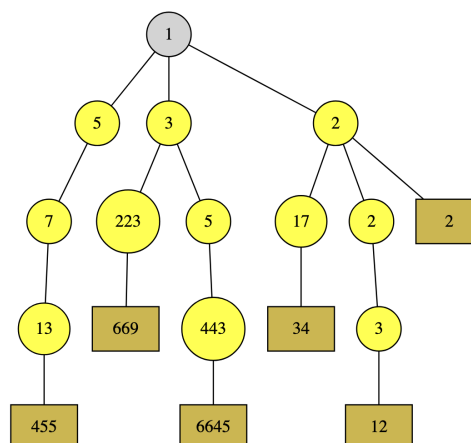
1. Implemente como anamorfismo de listas a função

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

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:

$$\text{prime_tree} :: [\mathbb{Z}] \rightarrow \text{Exp } \mathbb{Z} \ \mathbb{Z}$$

Sugestão: escreva o mínimo de código possível em *prime_tree* investigando cuidadosamente que funções disponíveis nas bibliotecas que são dadas podem ser reutilizadas.¹

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:

¹ 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:

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

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.¹ 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:

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

¹ 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](#)” [?], 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`¹ 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^AT_EX](#). (Ver também a `Makefile` que é disponibilizada.)

¹ 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^AT_EX](#). 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^AT_EX](#) 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_TE_X](#)) 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 ler¹ onde se obtém o efeito seguinte:²

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

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:

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

Exemplo de teste:

$\text{teste} = \text{evaluate}\ (\text{let_exp}\ f\ i) \equiv \text{Just}\ (26 / 245)$
where $f\ "x" = N\ 0; f\ "y" = N\ (1 / 7)$

¹ Procure e.g. por "sec:diagramas".

² 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.

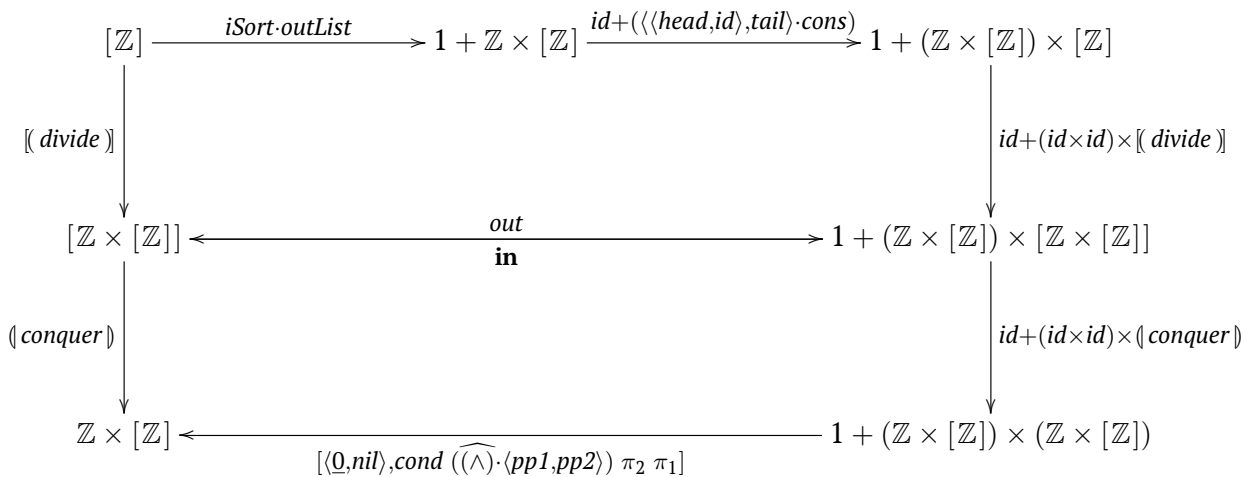
Problema 1

Para a resolução deste problema, utilizamos uma abordagem em duas fases, *divide* e *conquer*, típica dos hilomorfismos.

$$\begin{aligned} \text{divide} &:: [Int] \rightarrow () + ((Int, [Int]), [Int]) \\ \text{divide} &= (id + (\langle \langle head, id \rangle, tail \rangle \cdot cons)) \cdot outList \cdot iSort \end{aligned}$$

$$\begin{aligned} pp1 &= (\widehat{=}) \cdot \langle \pi_1 \cdot \pi_2, 0 \rangle \\ pp2 &= (\widehat{\leq}) \cdot \langle \pi_1 \cdot \pi_2, length \cdot \pi_2 \cdot \pi_2 \rangle \end{aligned}$$

$$\begin{aligned} \text{conquer} &:: () + ((Int, [Int]), (Int, [Int])) \rightarrow (Int, [Int]) \\ \text{conquer} &= [\langle 0, nil \rangle, cond (\widehat{(\wedge)} \cdot \langle pp1, pp2 \rangle) \pi_2 \pi_1] \end{aligned}$$

$$hindex = hyloList \text{conquer} \text{divide}$$


Problema 2

Primeira parte:

O *outNat* não é suficiente para este problema, logo desenvolvemos um *outPrimes* que junta os casos de 0, 1 e -1, e no outro caso faz o módulo do número que recebe.

divisorsList cria a lista de divisores de um número até à sua raiz quadrada (otimização). Esta é usada depois na função *isPrime* que verifica se um número é primo. Depois, *nextFactor* verifica qual o próximo fator primo de um número.

Por fim, *primes* é o anamorfismo que gera a lista de primos. A parte *aap · ÷ · nextFactor* recebe um inteiro *n* e divide-o pelo seu próximo fator primo. *aap* é uma função monádica que calcula *nextFactor n* e depois usa o resultado como argumento a *n ÷ nextFactor n*.

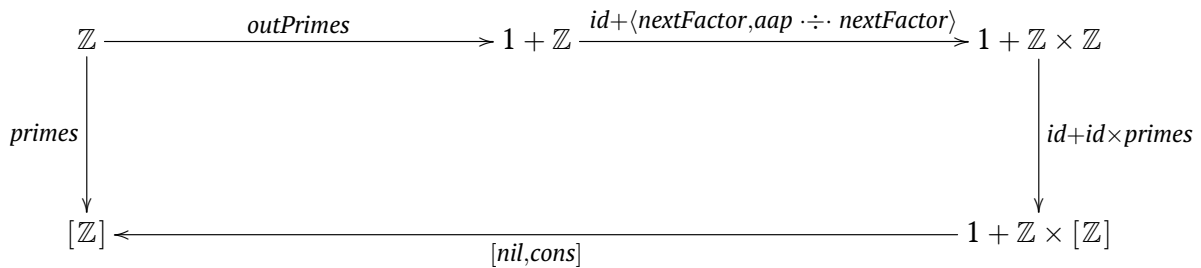
```
outPrimes 0 = i1 ()
outPrimes 1 = i1 ()
outPrimes (-1) = i1 ()
outPrimes n = i2 (abs n)
```

```
divisorsList :: ℤ → [ℤ]
divisorsList n = [x | x ← [2..isqrt n], mod n x ≡ 0]
  where isqrt = floor · sqrt · fromIntegral
```

```
isPrime :: ℤ → Bool
isPrime = [false, null · divisorsList] · outPrimes
```

```
nextFactor :: ℤ → ℤ
nextFactor n = head [x | x ← 2 : [3, 5..n], mod n x ≡ 0 ∧ isPrime x]
```

```
primes = [(id + ⟨nextFactor, aap · ÷ · nextFactor⟩) · outPrimes]
```



Segunda parte:

Para criar a árvore precisamos criar os 'ramos', neste caso num par no formato (fatorização, número) com a função *buildPairs*.

Depois, fornecemos esses 'ramos' à função *untar* que irá criar a árvore do tipo *Exp ℤ ℤ*. Essa árvore vem numa lista, então apenas precisamos de usar *head* para a obter.

```
buildPairs :: [ℤ] → [( [ℤ], ℤ )]
buildPairs = map ⟨(:) 1 · primes, id⟩
prime_tree = head · untar · buildPairs
```

Problema 3

O primeiro passo para a implementação da convolução de 2 listas é acrescentar (*length l1* − 1) zeros ao início da segunda lista para que fique com tamanho igual a *length l1* + *length l2* − 1, que é o tamanho da lista resultante da convolução. Depois calculamos as sublistas com *sufixes* para simular o deslizar de uma lista sobre a outra. De seguida são aplicados 2 map, um escrito na forma de catamorfismo outro em anamorfismo para ser possível criar um hilomorfismo. Primeiro, multiplica-se os elementos

correspondentes das sublistas e da primeira lista invertida e de seguida reduz-se cada lista à sua soma tendo como resultado a lista correspondente à convolução.

```
convolve :: Num a => [a] -> [a] -> [a]
convolve l1 = hyloList f g . suffixes . flip padZeros (length l1 - 1)
  where padZeros l = ([l, 0:])
        f = [nil, cons . (sum × id)]
        g = (id + (zipWith (*) (reverse' l1) × id)) . outList
```

Diagrama *padZeros*:

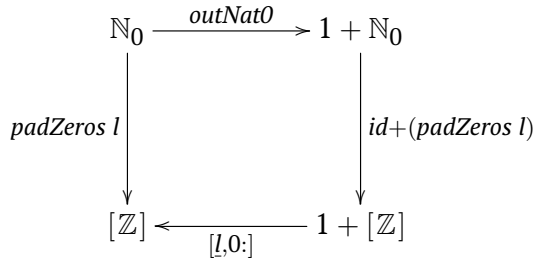
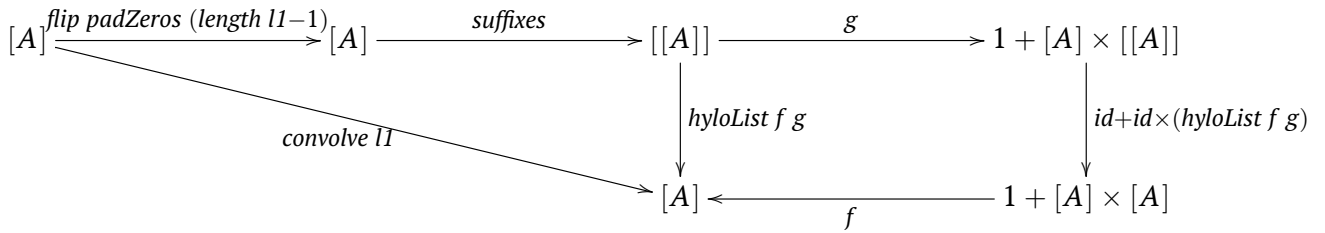


Diagrama *convolve*:



Problema 4

Definição do tipo:

```
outExpr (V a) = i1 a
outExpr (N b) = i2 (i1 b)
outExpr (T op l) = i2 (i2 (op, l))
recExpr f = baseExpr id id f
```

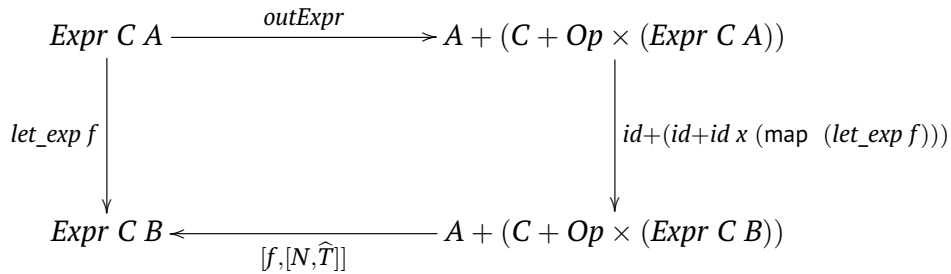
Ana + cata + hylo:

```
cataExpr g = g . recExpr (cataExpr g) . outExpr
anaExpr g = inExpr . recExpr (anaExpr g) . g
hyloExpr h g = cataExpr h . anaExpr g
```

```
instance Functor (Expr b) where
  fmap f = cataExpr (inExpr . baseExpr f id id)
instance Applicative (Expr b) where
  pure = return
  (< * >) = aap
instance Monad (Expr b) where
  return = V
  e >>= f = cataExpr [f, [N, T]] e
```

Maps: Monad: Monad: Let expressions:

$let_exp = flip (\gg=)$



Catamorfismo monádico:

$mcataExpr\ g = g \cdot ! (aux \cdot recExpr\ (mcataExpr\ g) \cdot outExpr)$
 $aux :: Monad\ m \Rightarrow a + (b + (Op, [m\ c])) \rightarrow m\ (a + (b + (Op, m\ [c])))$
 $aux = [return \cdot i_1, [return \cdot i_2 \cdot i_1, ret]]$
where $ret = return \cdot i_2 \cdot i_2 \cdot \langle \pi_1, sequence \cdot \pi_2 \rangle$

Avaliação de expressões:

$evaluate = cataExpr\ [nothing, [Just, g]]$
where $g\ (op, vals) = \text{case}\ (op, sequence\ vals)\ \text{of}$
 $(Add, Just\ [x, y]) \rightarrow Just\ (x + y)$
 $(Mul, Just\ [x, y]) \rightarrow Just\ (x * y)$
 $(ITE, Just\ [cond, t, e]) \rightarrow \text{if}\ cond > 0\ \text{then}\ Just\ t\ \text{else}\ Just\ e$
 $_ \rightarrow Nothing$

Index

\LaTeX , [5](#), [6](#)

bibtex, [6](#)

lhs2TeX, [5–7](#)

makeindex, [6](#)

pdflatex, [5](#)

xymatrix, [7](#)

Combinador “pointfree”

ana, [8](#)

 Listas, [9](#)

cata

 Naturais, [7](#), [10](#)

either, [7–11](#)

split, [7–9](#), [11](#)

Cálculo de Programas, [1](#), [5](#)

 Material Pedagógico, [5](#)

Docker, [5](#)

 container, [5](#), [6](#)

Função

π_1 , [7](#), [8](#), [11](#)

π_2 , [7](#), [8](#), [11](#)

length, [3](#), [8–10](#)

map, [7](#), [9](#), [11](#)

uncurry, [7](#), [8](#), [10](#), [11](#)

Haskell, [1](#), [5](#), [6](#)

 interpretador

 GHCi, [5](#), [6](#)

 Literate Haskell, [5](#)

Números naturais (\mathbb{N}), [7](#), [10](#)

Programação

 literária, [5](#), [7](#)

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](#)).