Universidade do Minho
Escola de Engenharia

Cálculo de Programas
Trabalho Prático (2024/25)

Lic. em Engenharia Informática

Grupo 7

a95485	Miguel Torres Carvalho
a96587	Flávia Alexandra da Silva Araújo
a104001	Frederico Cunha Afonso

janeiro 2025

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

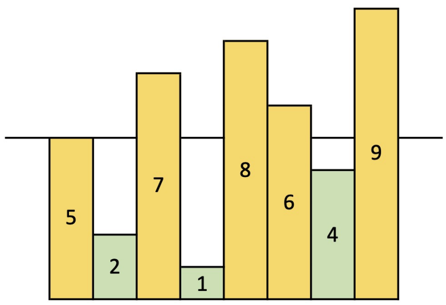
Esta questão aborda um problema que é conhecido pela designação '*H-index of a Histogram*' e que se formula facilmente:

> *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

$$hindex :: [Int] \to (Int, [Int])$$

tal que, para $(i, x) = hindex\ h$, 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. For exemplo,

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

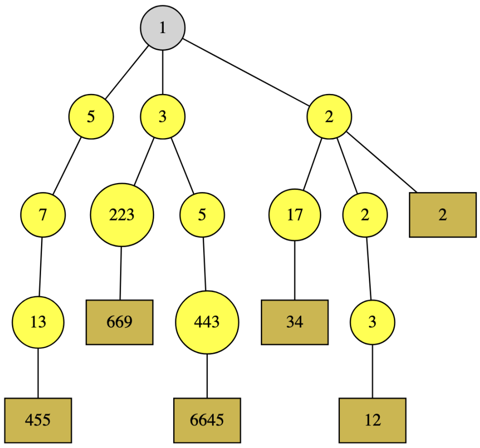
1. Implemente como anamorfismo de listas a função

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

   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.[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,

$$\begin{array}{l} \mathit{convolve} :: \mathit{Num}\ a \Rightarrow [a] \to [a] \to [a] \\ \mathit{convolve}\ \mathit{xs}\ \mathit{ys} = [\mathit{sum}\ \$\ \mathit{zipWith}\ (*)\ (\mathit{take}\ n\ (\mathit{drop}\ i\ \mathit{xs}))\ \mathit{ys} \mid i \leftarrow [0\,..\,(\mathit{length}\ \mathit{xs} - n)]] \\ \quad \textbf{where}\ n = \mathit{length}\ \mathit{ys} \end{array}$$

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

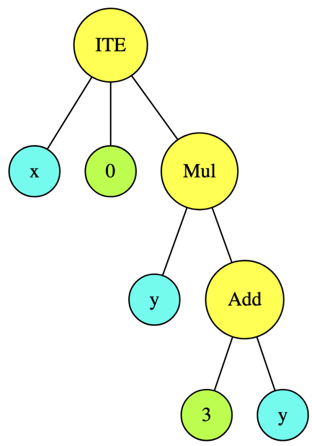
Proponha, explicando-a devidamente, uma solução sua para *convolve*. Valorizar-se-á a economia de código e o recurso aos combinadores *pointfree* estudados na disciplina, em particular a triologia *ana-cata-hilo* de tipos disponíveis nas bibliotecas dadas ou a definir.

Problema 4

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

$$\begin{array}{l} \textbf{data}\ \mathit{Expr}\ b\ a = V\ a \mid N\ b \mid T\ \mathit{Op}\ [\mathit{Expr}\ b\ a]\ \textbf{deriving}\ (\mathit{Show}, \mathit{Eq}) \\ \textbf{data}\ \mathit{Op} = \mathit{ITE} \mid \mathit{Add} \mid \mathit{Mul} \mid \mathit{Suc}\ \textbf{deriving}\ (\mathit{Show}, \mathit{Eq}) \end{array}$$

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



representa a expressão

$$\mathit{ite}\ (V\ \texttt{"x"})\ (N\ 0)\ (\mathit{multi}\ (V\ \texttt{"y"})\ (\mathit{soma}\ (N\ 3)\ (V\ \texttt{"y"}))) \tag{1}$$

— i.e. **if** x **then** 0 **else** $y * (3 + y)$ — assumindo as "helper functions":

$$\begin{array}{l} \mathit{soma}\ x\ y = T\ \mathit{Add}\ [x, y] \\ \mathit{multi}\ x\ y = T\ \mathit{Mul}\ [x, y] \\ \mathit{ite}\ x\ y\ z = T\ \mathit{ITE}\ [x, y, z] \end{array}$$

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 monádia, 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

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

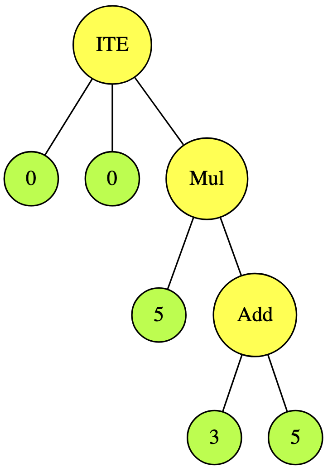
que, dada uma expressão com variáveis em a e uma função que a cada uma dessas variáveis atribui uma expressão ($a \rightarrow Expr\ c\ b$), faz a correspondente substituição.[1] Por exemplo, dada

$$\begin{array}{l} f\ \texttt{"x"} = N\ 0 \\ f\ \texttt{"y"} = N\ 5 \\ f\ \_ = N\ 99 \end{array}$$

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,

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

para f e e dadas acima.

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

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

[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.

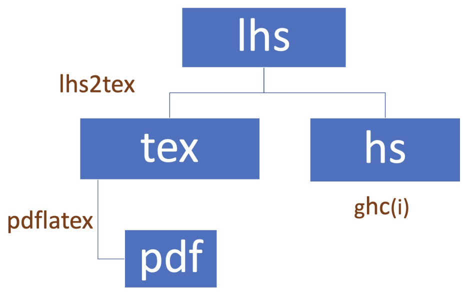
Para cumprir de forma integrada os objectivos do trabalho vamos recorrer a uma técnica de programação dita "literária" [**?**], cujo princípio base é o seguinte:

> *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 LaTeX. (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 LaTeX. 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-processador que faz "pretty printing" de código Haskell em LaTeX 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 BibTeX) 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[1] onde se obtém o efeito seguinte:[2]

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

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

$$
\begin{tikzcd}
\mathbb{N}_0 \arrow[d, "(\!|g|\!)"'] & 1 + \mathbb{N}_0 \arrow[l, "\mathsf{in}"'] \arrow[d, "id + (\!|g|\!)"] \\
B & 1 + B \arrow[l, "g"]
\end{tikzcd}
$$

E Código fornecido

Problema 1

$h :: [Int]$

Problema 4

Definição do tipo:

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

Exemplos de expressões:

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

Exemplo de teste:

$$
\begin{array}{l}
teste = evaluate\ (let\_exp\ f\ i) \equiv Just\ (26\ /\ 245) \\
\quad \textbf{where}\ f\ \texttt{"x"} = N\ 0; f\ \texttt{"y"} = N\ (1\ /\ 7)
\end{array}
$$

1 Procure e.g. por "`sec:diagramas`".

2 Exemplos tirados de [**?**].

F Soluções dos alunos

Problema 1

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

Para a resolução deste problema, recorreu-se a uma estratégia de divisão e conquista com recurso a um hilomorfismo de listas, definido na biblioteca List, com o anamorfismo responsável por ordenar a lista de *input* de forma crescente e o catamorfismo que, iterativamente, seleciona os elementos, incrementando o valor de *h-index*, em que n corresponde aos n elementos maiores ou iguais a n, sendo esta condição um invariante ao longo das iterações do catamorfismo. Graças à forma recursiva do catamorfismo, dada pelo functor do catamorfismo *conquer*, é possível iterar sobre a lista de forma decrescente, garantindo que o valor de *h-index* é o maior possível.

A estrutura de dados intermédia deste hilomorfismo é uma lista de inteiros, do tipo $[Int]$, esta correspondendo à saída do anamorfismo *divide* e à respetiva entrada do catamorfismo *conquer*.

$$
\begin{tikzcd}
Int^* \arrow[rr, bend left, "divide"] \arrow[d, "[\!(divide)\!]"'] \arrow[dd, bend right=60, "hindex"'] & & 1 + Int \times Int^* \arrow[d, "id+id\times[\!(divide)\!]"] \\
Int^* \arrow[rr, bend left, "out_{\mathrm{List}}"] \arrow[rr, phantom, "\cong"] \arrow[d, "(\!|conquer|\!)"'] & & 1 + Int \times Int^* \arrow[ll, bend left, "in_{\mathrm{List}}"] \arrow[d, "id+id\times(\!|conquer|\!)"] \\
Int \times Int^* & & 1 + Int \times (Int \times Int^*) \arrow[ll, bend left, "conquer"]
\end{tikzcd}
$$

$$
\begin{array}{l}
hindex = [\![\, conquer, \; divide \,]\!] \\
\quad \mathbf{where} \\
\qquad conquer = [g1, g2] \\
\qquad g1 = \underline{0, [\,]} \\
\qquad g2\ (h, t@(i, x)) \\
\qquad\quad \mid h > i = (i + 1, h : x) \\
\qquad\quad \mid otherwise = t \\
\qquad divide\ [\,] = i_1\ () \\
\qquad divide\ xs = \mathbf{let}\ m = minimum\ xs \\
\qquad\quad \mathbf{in}\ i_2\ (m, delete\ m\ xs)
\end{array}
$$

Para além da solução apresentada, foi também desenvolvida uma solução alternativa que recorre somente a um catamorfismo de listas, modificado de forma a que a função *out* seja capaz, aquando a lista de entrada não é vazia, de dividir a lista em duas partes, uma com o valor mínimo e outra com o restante da lista. Deste modo, a funcionalidade do anamorfismo *divide* é incorporada na função *out*.

A seguinte solução alternativa é 100% *pointfree*, com a função *g2* do gene do catamorfismo redesenhada, assim como a função *out*, esta, como supramencionado, equivalente à função *divide* da solução principal.

$$
\begin{tikzcd}
Int^* \arrow[r, "outSort_{List}"] \arrow[d, "hindex' = (\!| conquer |\!)"'] & 1 + Int \times Int^* \arrow[d, "id + id \times hindex'"] \\
Int \times Int^* & 1 + Int \times (Int \times Int^*) \arrow[l, "conquer"]
\end{tikzcd}
$$

$$
\begin{aligned}
&hindex' :: [Int] \rightarrow (Int, [Int]) \\
&hindex' = (\!| conquer |\!) \\
&\quad \mathbf{where} \\
&\qquad conquer = [g1, g2] \\
&\qquad g1 = \underline{0, [\,]} \\
&\qquad g2 = cond\ (\widehat{(>)} \cdot (id \times \pi_1)) \\
&\qquad\quad \langle \mathsf{succ}\ \cdot \pi_1 \cdot \pi_2, cons \cdot (id \times \pi_2) \rangle \\
&\qquad\quad \pi_2 \\
&outSortList = cond\ ([\,] \equiv) \\
&\quad (i_1 \cdot (!)) \\
&\quad (i_2 \cdot \langle \pi_1, \widehat{delete} \rangle \cdot \langle minimum, id \rangle) \\
&(\!| g |\!) = g \cdot recList\ (\!| g |\!) \cdot outSortList
\end{aligned}
$$

Problema 2

Para a descoberta dos fatores primos de um número, não negativo, foi implementado um anamorfismo de listas, *primes*, que, para um dado número, devolve a sua lista única de fatores primos.

Este anamorfismo é definido por um gene que, para um número n, verifica se este é menor que 2, caso em que devolve um valor à esquerda do tipo $1 + Int \times Int$ que indica o fim da iteração, ou, caso contrário, devolve um valor à direita do tipo $1 + Int \times Int$ cujo a primeira componente do par é o menor fator primo de n e a segunda componente é o quociente da divisão de n pelo seu menor fator primo, para que a iteração continue.

Para obtenção do menor fator primo de um número n, recorre-se a uma lista de inteiros por compreensão, $x \leftarrow [2\,..\,n]$, que, para cada x, verifica se $n \text{ `}mod\text{` } x \equiv 0$, ou seja, se x é um fator de n. Caso seja, x é o menor fator primo de n e devolve-se o par $(x, n \text{ `}div\text{` } x)$.

$$
\begin{tikzcd}
\mathbb{Z} \arrow[rr, bend left, "g"] \arrow[d, "primes = [\![g]\!]"'] & & 1 + \mathbb{Z} \times \mathbb{Z} \arrow[d, "id + id \times primes"] \\
\mathbb{Z}^* \arrow[rr, bend left, "out_{\mathrm{List}}"] \arrow[rr, phantom, "\cong"] & & 1 + \mathbb{Z} \times \mathbb{Z}^* \arrow[ll, bend left, "in_{\mathrm{List}}"]
\end{tikzcd}
$$

$$
\begin{array}{l}
primes = [\![g]\!] \\
\quad \textbf{where} \\
\qquad g\ n \\
\qquad\quad \mid n < 2 = i_1\ () \\
\qquad\quad \mid otherwise = \textbf{let}\ p = head\ [x \mid x \leftarrow [2\,..\,n], n \text{ `}mod\text{` } x \equiv 0] \\
\qquad\qquad \textbf{in}\ i_2\ (p, n \text{ `}div\text{` } p)
\end{array}
$$

Para a geração da árvore de primos de um conjunto de inteiros, optou-se por uma estratégia composta por dois passos principais:

1. Geração de uma de árvore de expressões após a obtenção da lista de fatores primos para cada número do conjunto de entrada, recorrendo ao anamorfismo de Exp$[\!(\, g \,)\!]$ e a função $\langle primes, id \rangle$, respetivamente.
2. Construção da árvore de primos, com recurso à função *mergeTrees*, que recebe uma lista de árvores de expressões e as funde numa única árvore.

$$\begin{tikzcd}
\mathbb{Z}^* \arrow[d, "map\ \langle primes{,}id \rangle"] \arrow[dd, bend right=60, "map\ ([\![g]\!] \cdot \langle primes{,}id \rangle)"'] \\
(\mathbb{Z}^* \times \mathbb{Z})^* \arrow[d, "map\ [\![g]\!]"] \\
(Exp\ \mathbb{Z}\ \mathbb{Z})^* \arrow[d, "mergeTrees"] \\
Exp\ \mathbb{Z}\ \mathbb{Z}
\end{tikzcd}$$

$$\begin{array}{l}
prime_tree = mergeTrees \cdot \mathsf{map}\ \ (anaExp\ g \cdot \langle primes, id \rangle) \\
\quad \textbf{where} \\
\qquad g\ ([\,], x) = i_1\ x \\
\qquad g\ (x : xs, y) = i_2\ (x, [(xs, y)]) \\
mergeTrees\ [tree] = tree \\
mergeTrees\ trees = Term\ 1\ (mergeSubtrees\ trees) \\
mergeSubtrees\ [\,] = [\,] \\
mergeSubtrees\ [tree] = [tree] \\
mergeSubtrees\ (Term\ o\ xs : Term\ o'\ ys : trees) \\
\quad \mid o \equiv o' = mergeSubtrees\ (Term\ o\ (xs +\!\!+\ ys) : trees) \\
\quad \mid otherwise = Term\ o\ xs : mergeSubtrees\ (Term\ o'\ ys : trees) \\
mergeSubtrees\ (tree : trees) = tree : mergeSubtrees\ trees
\end{array}$$

Problema 3

A convolução é uma operação muito utilizada em áreas como processamento de sinais, redes neurais e outras relacionadas à computação. Assim, consistindo em, partindo de duas listas a e b, combinar a informação de ambas as listas, através de multiplicações e somas, resultando na criação de uma nova lista.

De maneira a alcançar uma função que tivesse este comportamento, definimos a *convolve*.

Após verificar qual de ambas listas recebidas é a menor, esta irá criar uma lista de $N+M-1$ tuplos (onde N e M representam o tamanho da maior e menor lista recebida). Os tuplos vão ser constituídos pela maior lista recebida original à esquerda, e por uma versão alterada da menor lista recebida à direita (sendo cada versão criada única ao tuplo a que pertence).

Finalmente, com auxílio de um catamorfismo de listas, a lista será iterada, aplicando a cada tuplo uma função que multiplique todos os membros de ambas listas com um mesmo índice, e, mais tarde, faça o somatório da lista resultante, adicionando o resultado obtido a uma lista final.

$$
\begin{tikzcd}
A^* \times A^* \arrow[d, "divide"'] & \\
(A^* \times A^*)^* \arrow[r, "out_{List}"] \arrow[d, "(\!|conquer|\!)"'] & 1 + (A^* \times A^*) \times (A^* \times A^*)^* \arrow[d, "id + id \times (\!|conquer|\!)"] \\
A^* & 1 + (A^* \times A^*) \times A^* \arrow[l, "conquer"']
\end{tikzcd}
$$

$$
\begin{array}{l}
convolve :: Num\ a \Rightarrow [a] \rightarrow [a] \rightarrow [a] \\
convolve = (\!|\, conquer \,|\!)\ .:\ \overline{divide} \\
\quad \textbf{where} \\
\qquad (.:) :: (c \rightarrow d) \rightarrow (a \rightarrow b \rightarrow c) \rightarrow a \rightarrow b \rightarrow d \\
\qquad (.:) = (\cdot) \cdot (\cdot) \\
\qquad divide = \widehat{zipper} \cdot cond\ (\widehat{(>)} \cdot (length \times length))\ swap\ id \\
\qquad zipper\ n\ m = zip\ (replicate\ l\ n)\ (\mathrm{map}\ \ (h\ m)\ [1\mathinner{..}l]) \\
\qquad\quad \textbf{where} \\
\qquad\qquad l = length\ n + length\ m - 1 \\
\qquad h\ xs\ i = (+\!\!+)\ r \cdot reverse \cdot take\ (i - l)\ \$\ drop\ l\ xs \\
\qquad\quad \textbf{where} \\
\qquad\qquad l = max\ 0\ (i - length\ xs) \\
\qquad\qquad r = replicate\ l\ 0 \\
\qquad conquer = [g1, g2] \\
\qquad g1 = nil \\
\qquad g2 = cons \cdot (f \times id) \\
\qquad f = sum \cdot (\widehat{zipWith}\ (*))
\end{array}
$$

Como alternativa, foi desenvolvida uma versão desta função que é completamente *pointfree* e aproveita a inferência de tipos do Haskell.

Aqui, a função *convolve'*, ao invés de usar somente um catamorfismo de listas, baseia-se num hylomorfismo, este também de listas.

A função a ser usada pelo anamorfismo de listas vai recursivamente desenvolver o tuplo recebido, guardando neste um tuplo com a maior lista recebida e uma versão alterada da menor, e num segundo tuplo, a menor lista (original) e o número da iteração efetuada.

Por outro lado, função a ser usada pelo catamorfismo de listas será aquela que foi usada no fim da versão anterior de *convolve*, aplicando as anteriormente mencionadas multiplicações seguidas de um somatório da lista resultante a todos os tuplos ali presentes.

$$
\begin{tikzcd}
A^* \times A^* \arrow[d, "swapper"'] & \\
A^* \times A^* \arrow[d, "inFormat"'] & \\
B \arrow[r, bend left, "divide"] \arrow[d, "{[\!(divide)\!]}"'] \arrow[dd, bend right=60, "{[\![conquer, divide]\!]}"'] & 1 + B \times B \arrow[d, "id + id \times [\!(divide)\!]"] \\
B^* \arrow[r, bend left, "out_{List}"] \arrow[r, phantom, "\cong"] \arrow[d, "(\!|conquer|\!)"'] & 1 + B \times B^* \arrow[l, bend left, "in_{List}"] \arrow[d, "id + id \times (\!|conquer|\!)"] \\
A^* & 1 + B \times A^* \arrow[l, bend left, "conquer"]
\end{tikzcd}
$$

$$\text{where B} = (\text{A}^* \times A^*) \times (A^* \times Int)$$

$$
\begin{tikzcd}
A^* \times A^* \arrow[d, "{\langle sizeDiff, [\![conquer, divide]\!] \cdot inFormat \cdot swapper \rangle}"] \\
Int \times A^* \arrow[d, "\widehat{drop}"] \\
A^*
\end{tikzcd}
$$

$$
\begin{aligned}
& convolve' = \overline{\widehat{drop} \cdot \langle sizeDiff, [\![\, conquer, \ divide \,]\!] \cdot inFormat \cdot swapper \rangle} \\
& \quad \textbf{where} \\
& \qquad sizeDiff = abs \cdot \widehat{(-)} \cdot (length \times length) \\
& \qquad swapper = cond \ (\widehat{(>)} \cdot (length \times length)) \ id \ swap \\
& \qquad inFormat = \langle id, swap \cdot \langle flip \ (-) \ 1 \cdot (2*) \cdot length \cdot \pi_1, \pi_2 \rangle \rangle \\
& \qquad conquer = [nil, cons \cdot (applyMaths \times id)] \\
& \qquad \quad \textbf{where} \ applyMaths = sum \cdot (\widehat{zipWith \ (*)}) \cdot swap \cdot \pi_1 \\
& \qquad divide = cond \ ((0 \equiv) \cdot \pi_2 \cdot \pi_2) \ (i_1 \cdot (!)) \ (i_2 \cdot dup \cdot \langle \langle \pi_1 \cdot \pi_1, makeList \rangle, (id \times flip \ (-) \ 1) \cdot \pi_2 \rangle) \\
& \qquad \quad \textbf{where} \\
& \qquad \qquad makeList = buildPair \cdot \langle \pi_2, length \cdot \pi_1 \cdot \pi_1 \rangle \\
& \qquad \qquad buildPair = \widehat{drop} \cdot \langle \pi_2 \cdot \pi_1, \widehat{(+\!\!+)} \cdot \langle flip \ replicate \ 0 \cdot \pi_2, reverse \cdot \pi_1 \cdot \pi_1 \rangle \rangle
\end{aligned}
$$

Problema 4

Definição do tipo:

Para o cálculo de *outExpr*, partiu-se do isomorfismo de *outExpr* e *inExpr* e da definição de *inExpr*, concluindo-se a seguinte definição de *outExpr*:

$$
\begin{array}{ll}
 & outExpr \cdot inExpr = id \\
\equiv & \quad \{ \text{ Def. inExpr } \} \\
 & outExpr \cdot [V, [N, \widehat{T}]] = id \\
\equiv & \quad \{ \text{ 20: Fusão-+ (2x) } \} \\
 & [outExpr \cdot V, [outExpr \cdot N, outExpr \cdot \widehat{T}]] = id \\
\equiv & \quad \{ \text{ 17: Universal-+ (2x), 1: Natural-id (2x) } \} \\
 & \begin{cases} outExpr \cdot V = i_1 \\ outExpr \cdot N = i_2 \cdot i_1 \\ outExpr \cdot \widehat{T} = i_2 \cdot i_2 \end{cases} \\
\equiv & \quad \{ \text{ 72: Igualdade Extencional, 73: Def-comp, 86: Uncurry } \} \\
 & \begin{cases} outExpr\ (V\ v) = i_1\ v \\ outExpr\ (N\ n) = (i_2 \cdot i_1)\ n \\ outExpr\ (T\ o\ exprs) = (i_2 \cdot i_2)\ (o, exprs) \end{cases} \\
\Box &
\end{array}
$$

$$
\begin{array}{l}
outExpr\ (V\ v)\ = i_1\ v \\
outExpr\ (N\ n) = (i_2 \cdot i_1)\ n \\
outExpr\ (T\ o\ exprs) = (i_2 \cdot i_2)\ (o, exprs)
\end{array}
$$

Para a definição de *recExpr*, recorreu-se ao functor base *baseExpr* assim como tomou-se inspiração na definição de *recExp*, definida na biblioteca Exp. Deste modo, obteu-se que:

$$recExpr\ f\ = baseExpr\ id\ id\ f = id + (id + id \times \mathsf{map}\ \ f)$$

$$recExpr\ f = id + (id + id \times \mathsf{map}\ \ f)$$

Ana + cata + hylo:

Através do isomorfismo entre *inExpr* e *outExpr*, as respetivas leis de isomorfismos, ´Shunt-left' (33) e ´Shunt-right' (34), assim como da lei de indução, Universal-cata (46), da lei de coindução, Universal-ana (55), e da definição de hilomorfismos, obteve-se as seguintes definições:

$$
\begin{array}{l}
cataExpr\ g = g \cdot recExpr\ (cataExpr\ g) \cdot outExpr \\
anaExpr\ g\ \ = inExpr \cdot recExpr\ (anaExpr\ g) \cdot g \\
hyloExpr\ h\ g = cataExpr\ h \cdot anaExpr\ g
\end{array}
$$

Maps:

Para o tipo *Expr b* também foram desenvolvidas as instâncias de *Functor* e *BiFunctor*, com as definições de fmap e *bmap* respetivamente, estas derivadas da lei *Def-map-cata* (51).

$$\begin{array}{l} \textbf{instance}\ \mathit{Functor}\ (\mathit{Expr}\ b)\ \textbf{where} \\ \quad \mathsf{fmap}\ g = \mathit{cataExpr}\ (\mathit{inExpr} \cdot \mathit{baseExpr}\ g\ \mathit{id}\ \mathit{id}) \\ \textbf{instance}\ \mathit{BiFunctor}\ \mathit{Expr}\ \textbf{where} \\ \quad \mathit{bmap}\ h\ g = \mathit{cataExpr}\ (\mathit{inExpr} \cdot \mathit{baseExpr}\ g\ h\ \mathit{id}) \end{array}$$

Monad:

$$\begin{array}{l} \textbf{instance}\ \mathit{Monad}\ (\mathit{Expr}\ b)\ \textbf{where} \\ \quad \mathit{return} = V \\ \quad t \ggg g = (\mathit{muExpr} \cdot \mathsf{fmap}\ g)\ t \\ \mathit{muExpr} :: \mathit{Expr}\ b\ (\mathit{Expr}\ b\ a) \to \mathit{Expr}\ b\ a \\ \mathit{muExpr} = \mathit{cataExpr}\ [\mathit{id}, [N, \widehat{T}]] \\ \textbf{instance}\ \mathit{Applicative}\ (\mathit{Expr}\ b)\ \textbf{where} \\ \quad \mathit{pure} = \mathit{return} \\ \quad (<*>) = \mathit{aap} \end{array}$$

Let expressions:

Foram desenvolvidas duas versões da função *let_exp*, uma com recurso ao catamorfismo *cataExpr* com um gene similar a *inExpr* com exceção da aplicação da função f para as variáveis.

A outra versão é equivalente à principal, sendo esta uma versão *pointfree*, que recorre às funções *muExpr* e fmap.

$$\begin{array}{l} \mathit{let\_exp}\ f = \mathit{cataExpr}\ [f, [N, \widehat{T}]] \\ \mathit{let\_exp}' = \mathit{muExpr}\ .:\ \mathsf{fmap} \quad \text{-- equivalent to: (muExpr .) . fmap} \\ \quad \textbf{where}\ (.:) = (\cdot) \cdot (\cdot) \end{array}$$

TODO provar $\mathit{let\_exp} \equiv \mathit{let\_exp}'$

Catamorfismo monádico:

TODO Texto explicativo

$$\begin{array}{l} \mathit{mcataExpr}\ \mathit{phi}\ t = \textbf{do}\ \{b \leftarrow \mathit{traverseExpr}\ (\mathit{mcataExpr}\ \mathit{phi})\ (\mathit{outExpr}\ t); \mathit{phi}\ b\} \\ \mathit{traverseExpr}\ f = [\mathit{return} \cdot i_1, [\mathit{return} \cdot i_2 \cdot i_1, m]] \\ \quad \textbf{where}\ m\ (\mathit{op}, \mathit{es}) = \textbf{do}\ \{\mathit{cs} \leftarrow \mathit{mapM}\ f\ \mathit{es}; \mathit{return}\ \$\ i_2\ \$\ i_2\ (\mathit{op}, \mathit{return}\ \mathit{cs})\} \end{array}$$

Avaliação de expressões:

TODO Texto explicativo

$$\begin{array}{l} \mathit{evaluate} = \mathit{mcataExpr}\ \mathit{eval} \\ \quad \textbf{where} \\ \quad \mathit{eval} = [\underline{\mathit{Nothing}}, [\mathit{Just}, \mathit{evalTerm}]] \\ \quad \mathit{evalTerm}\ \mathit{op} = \textbf{case}\ \mathit{op}\ \textbf{of} \\ \quad\quad (\mathit{Add}, \mathit{Just}\ [x, y]) \to \mathit{Just}\ (x + y) \\ \quad\quad (\mathit{Mul}, \mathit{Just}\ [x, y]) \to \mathit{Just}\ (x * y) \\ \quad\quad (\mathit{Suc}, \mathit{Just}\ [x]) \to \mathit{Just}\ (x + 1) \\ \quad\quad (\mathit{ITE}, \mathit{Just}\ [x, y, z]) \to \textbf{if}\ x \not\equiv 0\ \textbf{then}\ \mathit{Just}\ y\ \textbf{else}\ \mathit{Just}\ z \\ \quad\quad \_ \to \mathit{Nothing} \end{array}$$