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

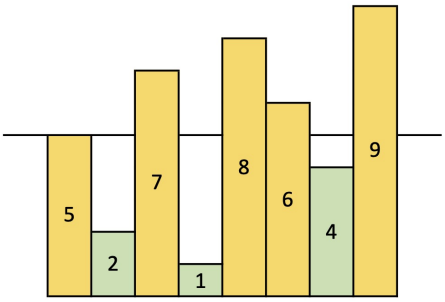
Esta questão aborda um problema que é conhecido pela designação '*H-index of a Histogram*' e que se formula facilmente:

> *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]
```

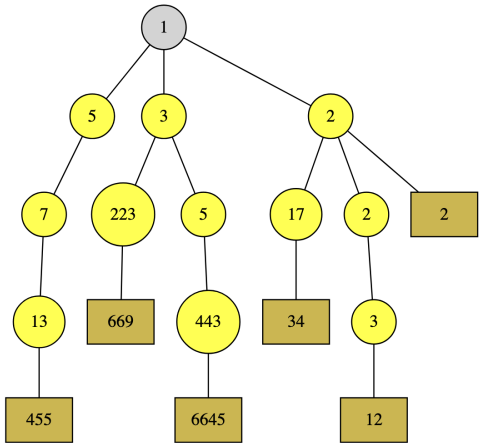
1. Implemente como anamorfismo de listas a função

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

   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:

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

   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,

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

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

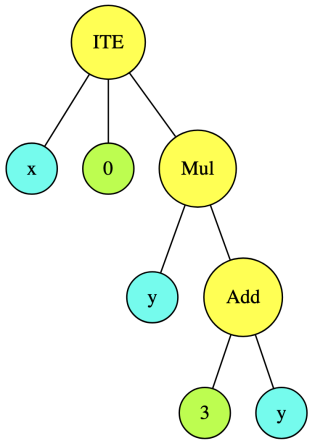
Proponha, explicando-a devidamente, uma solução sua para *convolve*. Valorizar-se-á a economia de código e o recurso aos combinadores *pointfree* estudados na disciplina, em particular a triologia *ana-cata-hilo* de tipos disponíveis nas bibliotecas dadas ou a definir.

Problema 4

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

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

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



representa a expressão

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

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

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

No anexo E propôe-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

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

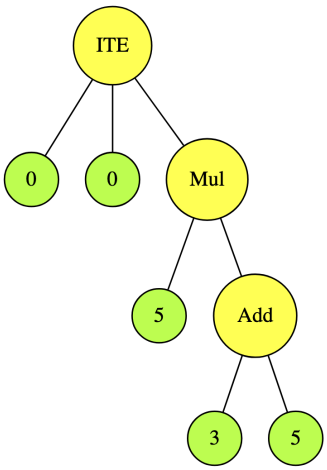
que, dada uma expressão com variáveis em a e uma função que a cada uma dessas variáveis atribui uma expressão ($a \rightarrow Expr\ c\ b$), faz a correspondente substituição.[1] Por exemplo, dada

$$f\ \texttt{"x"} = N\ 0$$
$$f\ \texttt{"y"} = N\ 5$$
$$f\ \_ = N\ 99$$

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,

$$evaluate\ e = Nothing$$
$$evaluate\ (let\_exp\ f\ e) = Just\ 40$$

para f e e dadas acima.

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

$$evaluate\ (T\ Add\ [N\ 2, N\ 3]) = Just\ 5$$
$$evaluate\ (T\ Mul\ [N\ 2]) = Nothing$$

[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.

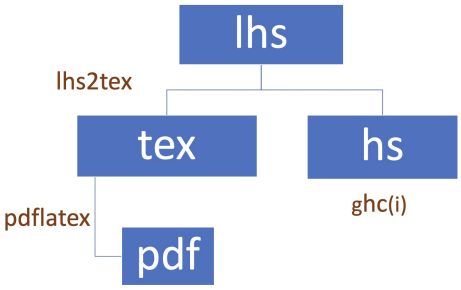
Para cumprir de forma integrada os objectivos do trabalho vamos recorrer a uma técnica de programação dita "literária" [1], cujo princípio base é o seguinte:

> *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`[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]

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

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

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

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:

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

Exemplo de teste:

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

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

[2] Exemplos tirados de [2].

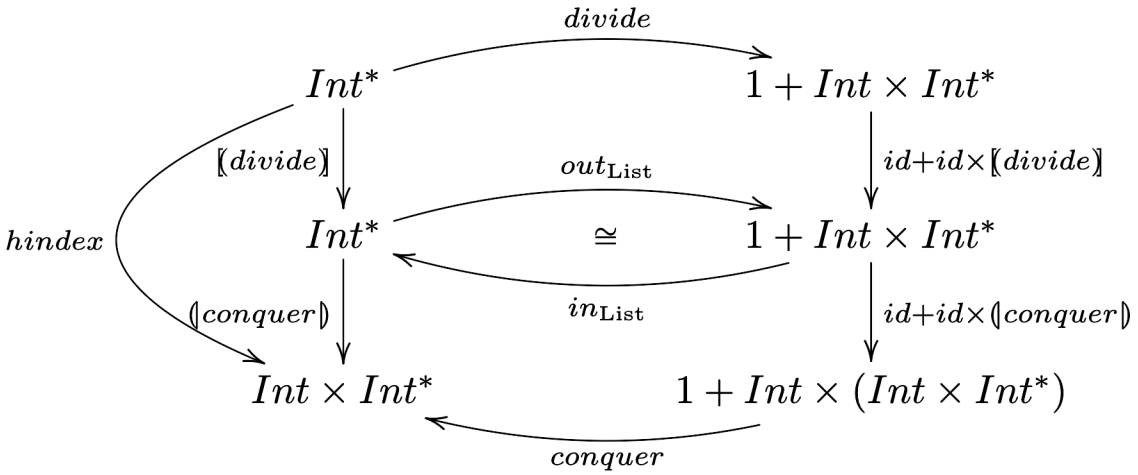
F Soluções dos alunos

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*.




```
hindex = ⟦ conquer, divide ⟧
  where
    conquer = [g1,g2]
    g1 = 0, []
    g2 (h,t@(i,x))
       | h > i = (i+1, h:x)
       | otherwise = t
    divide [] = i1 ()
    divide xs = let m = minimum xs
      in i2 (m, delete m xs)
```

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.

$$\begin{array}{ccc} Int^* & \xrightarrow{outSort_{List}} & 1 + Int \times Int^* \\ {\scriptstyle hindex' = (\!|conquer|\!)}\downarrow & & \downarrow{\scriptstyle id+id\times hindex'} \\ Int \times Int^* & \xleftarrow[conquer]{} & 1 + Int \times (Int \times Int^*) \end{array}$$

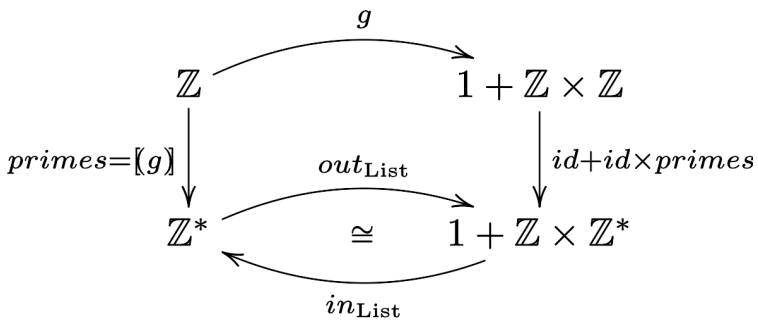
$$\begin{array}{l}
hindex' :: [Int] \to (Int, [Int]) \\
hindex' = (\!| conquer |\!) \\
\quad \textbf{where} \\
\qquad conquer = [g1, g2] \\
\qquad g1 = \underline{0, [\,]} \\
\qquad g2 = cond\ (\widehat{(>)} \cdot (id \times \pi_1)) \\
\qquad\quad \langle \mathsf{succ}\ \cdot \pi_1 \cdot \pi_2, cons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

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 $1 + Int \times Int$ cuja 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.

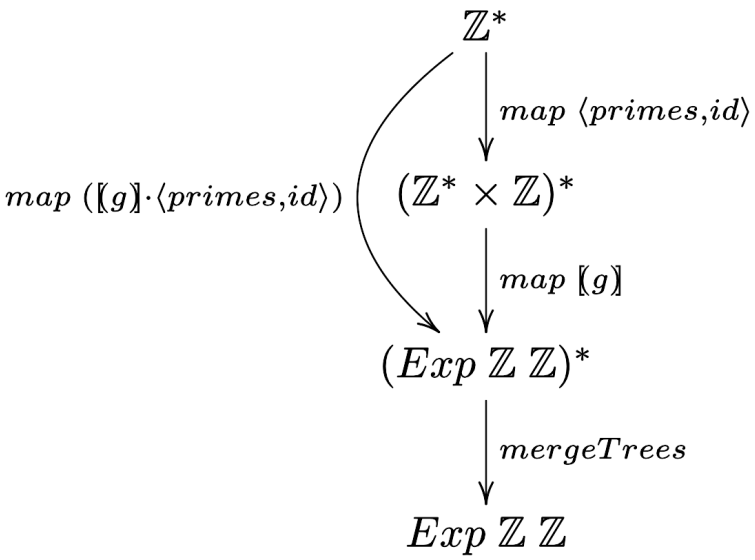
Para obtenção do menor fator primo de um número n, recorre-se a uma lista de inteiros por compreensão, $x \leftarrow [2\,..\,n]$, que, para cada x, verifica se n `` `mod` `` $x \equiv 0$, ou seja, se x é um fator de n. Caso seja, x é o menor fator primo de n e devolve-se o par $(x, n$ `` `div` `` $x)$.




```
primes = ⦇ g ⦈
  where
    g n
      | n < 2 = i1 ()
      | otherwise = let p = head [x | x ← [2 .. n], n `mod` x ≡ 0]
        in i2 (p, n `div` p)
```

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.



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

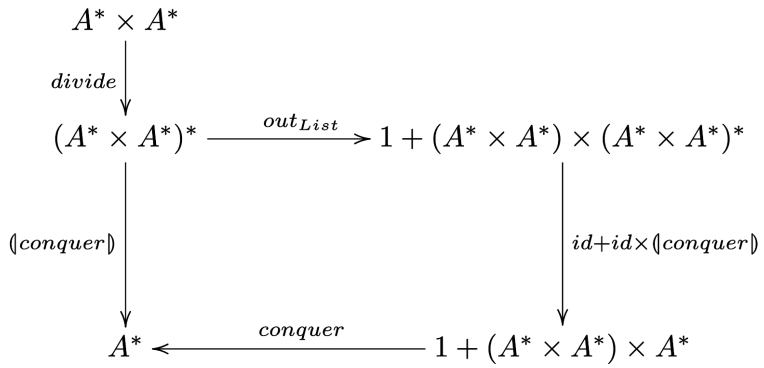
Problema 3

A convolução é uma operação muito utilizada em áreas como processamento de sinais, redes neurais e outras relacionadas à computação. Assim, partindo de duas listas a e b, combina-se 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 as listas recebidas é a menor, esta irá criar uma lista de $N + M - 1$ tuplos (onde N e M representam os tamanhos das listas). 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.



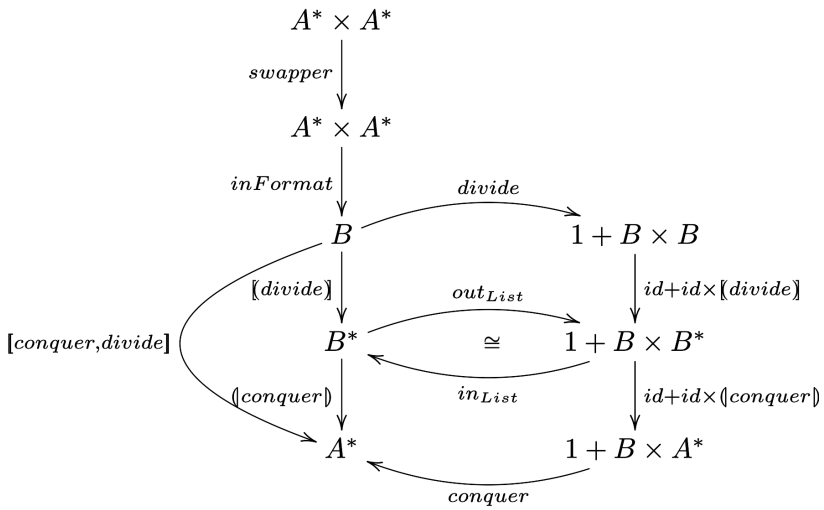

```
convolve :: Num a ⇒ [a] → [a] → [a]
convolve = (|conquer|) . : divide
  where
    (. :) :: (c → d) → (a → b → c) → a → b → d
    (. :) = (·) · (·)
    divide = zipper · cond ((>) · (length × length)) swap id
    zipper n m = zip (replicate l n) (map (h m) [1 . . l])
      where
        l = length n + length m − 1
    h xs i = (++) r · reverse · take (i − l) $ drop l xs
      where
        l = max 0 (i − length xs)
        r = replicate l 0
    conquer = [g1, g2]
    g1 = nil
    g2 = cons · (f × id)
    f = sum · (zipWith (∗))
```

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

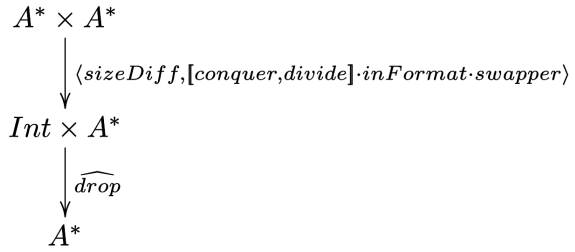
A função *convolve'*, ao invés de usar somente um catamorfismo de listas, baseia-se num hilomorfismo, este também de listas.

O gene do anamorfismo 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, o gene do catamorfismo será aquele usado na versão anterior de *convolve*, aplicando as anteriormente mencionadas multiplicações seguidas de um somatório da lista resultante a todos os tuplos ali presentes.



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



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

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*:

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

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

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:

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

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

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:

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

Maps:

Foram desenvolvidas as instâncias de *Functor* e *BiFunctor* para o tipo *Expr*, com fmap e bmap derivadas da lei *Def-map-cata* (51).

```
instance Functor (Expr b) where
  fmap g = cataExpr (inExpr · baseExpr g id id)
instance BiFunctor Expr where
  bmap h g = cataExpr (inExpr · baseExpr g h id)
```

Monad:

```
instance Monad (Expr b) where
  return = V
  t >>= g = (muExpr · fmap g) t
muExpr :: Expr b (Expr b a) → Expr b a
muExpr = cataExpr [id, [N, T̂]]
instance Applicative (Expr b) where
  pure = return
  (<*>) = aap
```

Let expressions:

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

```
let_exp f = cataExpr [f, [N, T̂]]
let_exp' = muExpr . : fmap    -- equivalent to: let_exp f = muExpr . fmap f
  where (. :) = (·) · (·)
```

$$\begin{array}{ccc} Expr\ C\ A & \xrightarrow{out_{Expr}} & A + (C + Op \times Expr\ C\ A) \\ {\scriptstyle let\_exp\ f = (\!|[f,[N,\widehat{T}]]|\!)}\downarrow & & \downarrow{\scriptstyle id+(id+id\times map\ (let\_exp\ f))} \\ Expr\ C\ B & \xleftarrow{[f,[N,\widehat{T}]]} & A + (C + Op \times Expr\ C\ B) \end{array}$$

$$\text{where f} :: \text{A} \to Expr\ C\ B$$

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

$$
\begin{aligned}
& let\_exp\ f = muExpr \cdot \text{fmap}\ f \\
\equiv & \quad \{\ \text{Def. muExpr, fmap f = T f}\ \} \\
& let\_exp\ f = (\!|\, [id, [N, \widehat{T}]] \,|\!) \cdot T\ f \\
\equiv & \quad \{\ \text{52: Absorção-cata}\ \} \\
& let\_exp\ f = (\!|\, [id, [N, \widehat{T}]] \cdot B\ (f, id) \,|\!) \\
\equiv & \quad \{\ B\ (f, id) = baseExpr\ f\ id\ id = f + (id + id \times \text{map}\ \ id)\ \} \\
& let\_exp\ f = (\!|\, [id, [N, \widehat{T}]] \cdot (f + (id + id \times \text{map}\ \ id)) \,|\!) \\
\equiv & \quad \{\ \text{map}\ \ id = id\text{, 22: Absorção-}+\text{, 1: Natural-id, 15: Functor-id-}\times\ \} \\
& let\_exp\ f = (\!|\, [f, [N, \widehat{T}]] \,|\!) \\
\square &
\end{aligned}
$$

Catamorfismo monádico:

O catamorfismo monádico é uma extensão do conceito clássico de catamorfismo, integrando efeitos monádicos no processo de redução de uma estrutura. No contexto do tipo *Expr*, ele permite combinar a manipulação de subestruturas com operações que podem falhar ou envolver efeitos computacionais.

A definição do *mcataExpr* recorre a função *traverseExpr* para percorrer a estrutura exterior de forma monádica e recursiva, após a travessia, aplica a função *phi* ao resultado, que encapsula a lógica final de transformação.

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

Avaliação de expressões:

A função *evaluate* utiliza o catamorfismo monádico para calcular o valor de uma expressão, considerando:

1. **Variáveis** — não avaliadas, retornam *Nothing*.
2. **Aridades** — *Just* com o valor.
3. **Termos compostos** — avaliados de acordo com operadores pré-definidos.

Os operadores suportados incluem:

- *Add* — adição de dois valores,
- *Mul* — multiplicação de dois valores,
- *Suc* — sucessor de um valor, equivalente a $+1$,
- *ITE* — *if-then-else*, controlado por um valor condicional.

Cada operador é aplicado a uma lista de operandos previamente avaliados. Se os operandos não forem válidos (exemplo: número insuficiente de argumentos), a avaliação falha com *Nothing*.

Esta definição segue um estilo declarativo, aproveitando o poder combinatório de *monads* para simplificar a lógica de processamento, abstraindo os detalhes da manipulação estrutural.

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

Index

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).