Universidade do Minho
Escola de Engenharia

Cálculo de Programas

Trabalho Prático (2024/25)

Lic. em Engenharia Informática

Grupo G99

a104095	Rafael Airosa Pereira
a104532	Tomás Sousa Barbosa
a104272	João Miguel Freitas Rodrigues

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

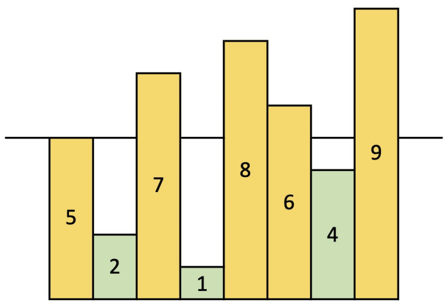
Esta questão aborda um problema que é conhecido pela designação '*H-index of a Histogram*' e que se formula facilmente:

> *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

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

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]
```

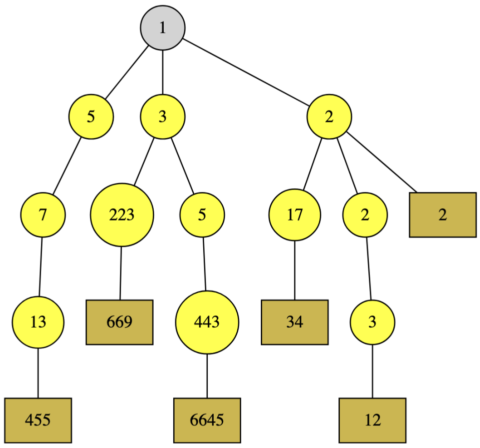
1. Implemente como anamorfismo de listas a função

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

   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,

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

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

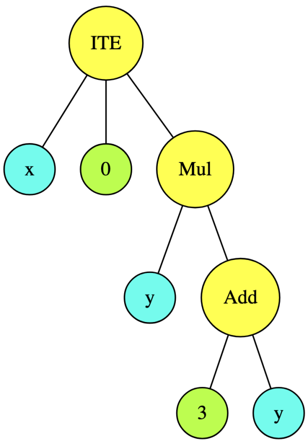
Proponha, explicando-a devidamente, uma solução sua para *convolve*. Valorizar-se-á a economia de código e o recurso aos combinadores *pointfree* estudados na disciplina, em particular a triologia *ana-cata-hilo* de tipos disponíveis nas bibliotecas dadas ou a definir.

Problema 4

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

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

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



representa a expressão

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

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

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

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

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

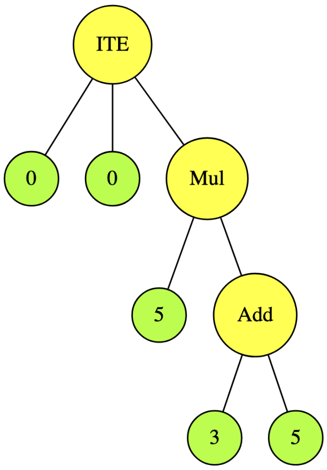
que, dada uma expressão com variáveis em a e uma função que a cada uma dessas variáveis atribui uma expressão ($a \rightarrow Expr\ c\ b$), faz a correspondente substituição.[1] Por exemplo, dada

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

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.

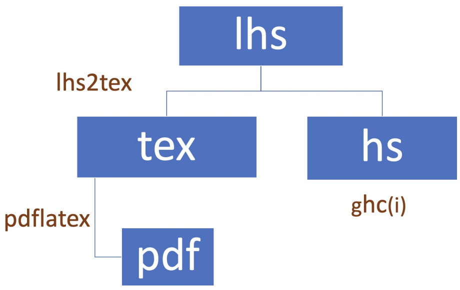
Para cumprir de forma integrada os objectivos do trabalho vamos recorrer a uma técnica de programação dita "literária" [1], cujo princípio base é o seguinte:

> *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]

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

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:

$$
\begin{array}{l}
inExpr :: a + (b + (Op, [Expr\ b\ a])) \rightarrow Expr\ b\ a \\
inExpr = [V, [N, \widehat{T}]] \\
baseExpr :: (a1 \rightarrow b1) \rightarrow (a2 \rightarrow b2) \rightarrow (a3 \rightarrow b3) \rightarrow a1 + (a2 + (b4, [a3])) \rightarrow b1 + (b2 + (b4, [b3])) \\
baseExpr\ g\ h\ f = g + (h + id \times \mathsf{map}\ \ f)
\end{array}
$$

Exemplos de expressões:

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

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

[2] Exemplos tirados de [2].

Exemplo de teste:

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

F Soluções dos alunos

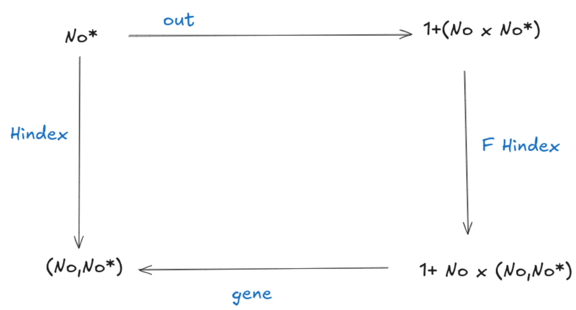
Os alunos devem colocar neste anexo as suas soluções para os exercícios propostos, de acordo com o "layout" que se fornece. Não podem ser alterados os nomes ou tipos das funções dadas, mas pode ser adicionado texto ao anexo, bem como diagramas e/ou outras funções auxiliares que sejam necessárias.

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

Problema 1

$$
\begin{array}{l}
hindex = (\!|\, gene\, |\!) \cdot iSort \\
\quad \textbf{where} \\
\qquad gene :: () + (Int, (Int, [Int])) \rightarrow (Int, [Int]) \\
\qquad gene\ (i_1\ \_) = (0, [\,]) \\
\qquad gene\ (i_2\ (x, (h, l))) \\
\qquad\quad \mid x \geqslant length\ l + 1 = (length\ l + 1, x : l) \\
\qquad\quad \mid x \geqslant h = (h, x : l) \\
\qquad\quad \mid otherwise = (h, l)
\end{array}
$$

Este problema foi resolvido através de um catamorfismo da função gene. Foi resolvido através do seguinte diagrama:



A ideia por detrás desta resolução é a seguinte:

- 1. Organizamos o array que foi dado por ordem crescente.
- 2. Retiramos a cabeça do array e usámo-la como candidato a resposta.
- 3. Caso a length do array seja maior do que o candidato, então quer dizer que existem um número maior ou igual de números maiores que o candidato.
- 4. O candidato passa a ser a nossa nova resposta.
- 5. Repetimos o processo até o tamanho do array ser menor do que o valor do candidato. Quando isso acontecer, então devolvemos a resposta atual e o array final.

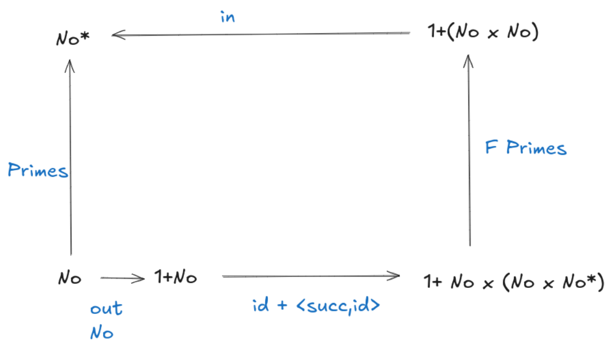
Problema 2

Primeira parte:

$$primes = [\!(\, gene \,)\!]$$

$$gene :: \mathbb{Z} \rightarrow () + (\mathbb{Z}, \mathbb{Z})$$
$$gene = (id + succ\_id) \cdot outN0$$
where
$$outN0 :: \mathbb{Z} \rightarrow () + \mathbb{Z}$$
$$outN0 = cond\ ({>}1)\ (i_2)\ \underline{i_1\ ()}$$
$$succ\_id :: \mathbb{Z} \rightarrow (\mathbb{Z}, \mathbb{Z})$$
$$succ\_id = \langle smallestPrimeFactor, \lambda n \rightarrow n \div smallestPrimeFactor\ n \rangle$$
$$smallestPrimeFactor :: \mathbb{Z} \rightarrow \mathbb{Z}$$
$$smallestPrimeFactor\ n = head\ [x \mid x \leftarrow [2\,..\,n], n \text{ `mod` } x \equiv 0]$$

Este problema foi resolvido através de um anamorfismo da função gene e do seu diagrama:



A função gene é a função geradora que cria a lista de inteiros a partir de um inteiro, sendo composta por duas funções:

- A função `outN0`, que verifica se o número é maior que 1, já que um número primo não pode ser negativo.
- A função `succ_id`, que divide o número pelo seu menor fator primo.

Além disso, implementamos mais uma função:
A função `smallestPrimeFactor`, uma função auxiliar que encontra o menor fator primo de um número.

Segunda parte:

$$prime\_tree\ xs = Term\ 1\ (untar\ (zip\ (\mathrm{map}\ \ primes\ xs)\ xs))$$

Esta solução foi alcançada depois de estudar a função untar que nos foi disponibilizada na biblioteca "Exp.hs".
Explicando melhor o código:

- A função `primes` é aplicada a cada elemento da lista xs, criando uma lista de listas de primos.
- A função `zip` junta as duas listas, criando uma lista de pares (lista de primos, valor inicial).
- A função `untar` transforma o par numa Exp tree.
- A função `Term1` coloca o 1 na raiz da árvore, este tem que ser posto "manualmente", pois ele não é um número primo.

Problema 3

$$
\begin{array}{l}
\textbf{type}\ State\ a = ([a],[a],Int)\\
matrixGen :: Num\ a \Rightarrow State\ a \rightarrow () + ([a], State\ a)\\
matrixGen\ (xs, ys, i)\\
\quad \mid i \geqslant length\ ys = i_1\ ()\\
\quad \mid otherwise = i_2\ (row, (xs, ys, i+1))\\
\quad \textbf{where}\ row = [x * (ys\ !!\ i) \mid x \leftarrow xs]\\
getDiagonal :: Int \rightarrow [[a]] \rightarrow [a]\\
getDiagonal\ k\ matrix\\
\quad \mid k < 0 = [\,]\\
\quad \mid otherwise = [matrix\ !!\ i\ !!\ j\\
\qquad \mid i \leftarrow [0\,..\,rows-1], j \leftarrow [0\,..\,cols-1], i+j \equiv k]\\
\quad \textbf{where}\\
\qquad rows = length\ matrix\\
\qquad cols = length\ (head\ matrix)\\
sumDiagonals :: Num\ a \Rightarrow [[a]] \rightarrow [a]\\
sumDiagonals\ matrix = \mathrm{map}\ \ (\lambda k \rightarrow sum\ (getDiagonal\ k\ matrix))\ [0\,..\,(rows+cols-2)]\\
\quad \textbf{where}\\
\qquad rows = length\ matrix\\
\qquad cols = length\ (head\ matrix)\\
convolve :: Num\ a \Rightarrow [a] \rightarrow [a] \rightarrow [a]\\
convolve\ xs\ ys = sumDiagonals\ [\!(\ matrixGen\ )\!]\ (xs, ys, 0)
\end{array}
$$

Decidimos implementar um type State a para facilitar o encapsulamento do estado necessário para gerar a matriz onde os dois primeiros argumentos apresentam as duas listas e o terceiro argumento um indice para controlar a iteração e indicar a linha da matriz a ser gerada.

Esta função foi implementada utilizando um anamorfismo da função `matrixGen`, seguido pela aplicação da função `sumDiagonals`.

O método de resolução foi inspirado no vídeo do canal 3Blue1Brown, que aborda a convolução de duas listas, relativamente à parte da utilização de uma matriz de modo a tornar a função mais eficiente (O(N.log(N))).

Explicação de forma simplificada:

- A função `matrixGen` é a função geradora responsável por criar uma única linha de uma matriz.
- O anamorfismo dessa função gera todas as linhas da matriz, formando a estrutura completa.
- Em seguida, a função `sumDiagonals` soma as diagonais da matriz e adiciona o resultado de cada soma a um array, que é então retornado como resultado final.

Problema 4

Resposta à Pergunta 1

$$
\begin{array}{l}
outExpr :: Expr\ b\ a \rightarrow a + (b + (Op, [Expr\ b\ a]))\\
outExpr\ (V\ a) = i_1\ a\\
outExpr\ (N\ b) = i_2\ (i_1\ b)\\
outExpr\ (T\ op\ l) = i_2\ (i_2\ (op, l))
\end{array}
$$

$$recExpr :: (a3 \to b3) \to b1 + (b2 + (b4, [a3])) \to b1 + (b2 + (b4, [b3]))$$
$$recExpr = baseExpr\ id\ id$$

Explicação :

- A função `outExpr` implementa o observador do tipo `Expr`, realizando a desconstrução da estrutura em um tipo soma `Either`:
  - Para variáveis (V a), retorna Left a
  - Para números (N b), retorna Right (Left b)
  - Para termos (T op l), retorna Right (Right (op, l))
- A função `recExpr` implementa a recursividade do tipo utilizando `baseExpr`, mantendo a estrutura original através de funções identidade.

Ana + *cata* + *hylo*:

$$cataExpr :: (b1 + (b2 + (Op, [b3])) \to b3) \to Expr\ b2\ b1 \to b3$$
$$cataExpr\ g = g \cdot recExpr\ (cataExpr\ g) \cdot outExpr$$
$$anaExpr :: (a3 \to a + (b + (Op, [a3]))) \to a3 \to Expr\ b\ a$$
$$anaExpr\ g = inExpr \cdot recExpr\ (anaExpr\ g) \cdot g$$
$$hyloExpr :: (b1 + (b2 + (Op, [c])) \to c) \to (a \to b1 + (b2 + (Op, [a]))) \to a \to c$$
$$hyloExpr\ h\ g = cataExpr\ h \cdot anaExpr\ g$$

Através da biblioteca `LTree.hs` e do seu catamorfismo, anamorfismo e hilomorfismo implementados,chegamos à construção das funções:

- `cataExpr`: Implementa o catamorfismo que percorre a estrutura recursivamente de baixo para cima:
  - Desconstrói a expressão usando `outExpr`
  - Aplica recursivamente a transformação usando `recExpr`
  - Finaliza aplicando a função de redução g
- `anaExpr`: Implementa o anamorfismo que constrói a estrutura recursivamente:
  - Usa a função gene g para criar a estrutura
  - Aplica recursivamente `recExpr`
  - Constrói a expressão final usando `inExpr`
- `hyloExpr`: Combina catamorfismo e anamorfismo em uma única transformação.

Resposta à Pergunta 2

Para declarar `Expr b` como instância da classe `Monad`, precisamos primeiro garantir que ela também seja instância de `Functor` e `Applicative`, já que estas são superclasses de `Monad` em Haskell.

Functor

A implementação do `Functor` é feita usando um catamorfismo:

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

Este `fmap` aplica a função `f` apenas às variáveis construtores (V), mantendo constantes (N) e operações (T) inalteradas.

Applicative

Para `Applicative`, aproveitamos as definições monádicas:

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Applicative}\ (\mathit{Expr}\ b)\ \textbf{where} \\
\quad \mathit{pure} = \mathit{return} \\
\quad (< * >) = \mathit{aap}
\end{array}
$$

Onde `aap` é a função auxiliar que converte operações monádicas em aplicativas.

Monad

A implementação da instância `Monad` é a parte central:

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Monad}\ (\mathit{Expr}\ b)\ \textbf{where} \\
\quad \mathit{return} = V \\
\quad (V\ a) \gg\!= f = f\ a \\
\quad (N\ b) \gg\!= f = N\ b \\
\quad (T\ op\ l) \gg\!= f = T\ op\ (\mathrm{map}\ \ (\gg\!= f)\ l)
\end{array}
$$

Vamos analisar cada componente:

- `return = V`: O construtor V serve como a função `return`, injetando valores puros na expressão
- `(V a) »= f`: Para variáveis, aplicamos a função `f` diretamente ao valor contido
- `(N b) »= f`: Constantes numéricas permanecem inalteradas
- `(T op l) »= f`: Para operações, mantemos o operador e aplicamos a transformação recursivamente a cada subexpressão na lista

Esta implementação satisfaz as leis monádicas:

1. Left : `return a »= f` = `f a`
2. Right : `m »= return` = `m`
3. Associatividade : `(m »= f) »= g` = `m »= (`
   `x -> f x »= g)`

A instância `Monad` permite usar expressões em contextos monádicos, facilitando operações como substituição de variáveis e avaliação de expressões em diferentes contextos.

Resposta à Pergunta 3

Maps: *Monad*: *Let expressions*:

$$\mathit{let\_exp}\ f = (\gg\!= f)$$

Esta implementação aproveita a instância `Monad` de `Expr`, onde:

- O operador »= (bind) aplica a função de substituição `f` às variáveis
- Para números, a estrutura é mantida enquanto a substituição é aplicada recursivamente
- A instância `Monad` garante que a substituição é feita de forma consistente em toda a expressão

Resposta à Pergunta 4

Catamorfismo monádico:

$$
\begin{array}{l}
mcataExpr\ f\ (V\ a) = f\ (i_1\ a) \\
mcataExpr\ f\ (N\ b) = f\ (i_2\ (i_1\ b)) \\
mcataExpr\ f\ (T\ op\ l) = f\ (i_2\ (i_2\ (op, mapM\ (mcataExpr\ f)\ l)))
\end{array}
$$

O catamorfismo monádico `mcataExpr` é uma versão do catamorfismo adaptada para um contexto monádico. Ele permite transformar um valor de um tipo de dado da estrutura Expr em um tipo de dado monádico.

É uma função recursiva que aplica uma função f a cada elemento, através de um comportamento monádico.

- Para variáveis (V a): a função f é aplicada diretamente à variável a, encapsulado em i1
- Para números (N b): a função f é aplicada ao número b, encapsulado em i2 . i1
- Para termos (T op l):
  - Usa mapM para aplicar mcataExpr f recursivamente a cada elemento da lista
  - Combina o operador e a lista processada em um tuplo
  - Encapsula em i2 . i2 e aplica f

Resposta à Pergunta 5

Avaliação de expressões:

$$
\begin{array}{l}
evaluate\ (V\ \_) = Nothing \\
evaluate\ (N\ b) = Just\ b \\
evaluate\ (T\ Add\ [x, y]) = (+)\ \langle \$ \rangle\ evaluate\ x < * > evaluate\ y \\
evaluate\ (T\ Mul\ [x, y]) = (*)\ \langle \$ \rangle\ evaluate\ x < * > evaluate\ y \\
evaluate\ (T\ ITE\ [cond, thenExpr, elseExpr]) = \\
\quad \mathbf{case}\ evaluate\ cond\ \mathbf{of} \\
\qquad Just\ 0 \rightarrow evaluate\ elseExpr \\
\qquad Just\ \_ \rightarrow evaluate\ thenExpr \\
\qquad Nothing \rightarrow Nothing \\
evaluate\ \_ = Nothing
\end{array}
$$

A função `evaluate` avalia uma expressão do tipo Expr e retorna um valor do tipo `Maybe a` onde :

- `Nothing` indica que a expressão não é váilida (falhou)
- `Just a` contém o resultado da expressão caso este seja válida

A função avalia diferentes padrões de `Expr` de acordo com os seguintes casos:

- **Caso 1: Variáveis (V)**
  - `evaluate (V _)` retorna `Nothing`
  - Este caso cobre expressões contendo variáveis que não podem ser avaliadas diretamente
- **Caso 2: Valores Numéricos (N)**
  - `evaluate (N b)` retorna `Just b`

  - Um valor numérico (ou constante) pode ser avaliado diretamente
- **Caso 3: Soma (Add)**
  - Avalia as duas subexpressões x e y e aplica a função soma $+$ caso ambas sejam válidas
  - Caso alguma subexpressão não seja válida (`Nothing`), o resultado será `Nothing`
- **Caso 4: Multiplicação (`Mul`)**
  - Similar ao caso de soma, mas aplica a função multiplicação $*$
  - Exige exatamente duas subexpressões, qualquer número diferente resultará em `Nothing`
- **Caso 5: Condicional (`ITE`)**
  - `evaluate (T ITE [cond, thenExpr, elseExpr])`
  - Avalia a condição (cond) primeiro:
    * Se a condição for 0, avalia e retorna o valor de `elseExpr`
    * Se a condição for diferente de 0, avalia e retorna o valor de `thenExpr`
    * Se a condição for inválida (`Nothing`), o resultado será `Nothing`
- **Caso 6: Padrões Genéricos**
  - `evaluate _ = Nothing`
  - Este caso cobre situações inválidas, como operadores com número incorreto de subexpressões ou operadores desconhecidos
  - Por exemplo, `evaluate (T Add [N 2])` ou `evaluate (T ITE [N 1, N 2])` retornarão `Nothing`

Com isto ao testar a função com o exemplo do enunciado, concluimos que funcionava corretamente.

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).