



Universidade do Minho
Escola de Engenharia

Cálculo de Programas

Trabalho Prático (2024/25)

Lic. em Engenharia Informática

Grupo G2

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

Preâmbulo

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

Na sequência pedagógica dos planos de estudo dos cursos que têm esta disciplina, opta-se pela aplicação deste método à programação em [Haskell](#) (sem prejuízo da sua aplicação a outras linguagens funcionais). Assim, o presente trabalho prático coloca os alunos perante problemas concretos que deverão ser implementados em [Haskell](#). Há ainda um outro objectivo: o de ensinar a documentar programas, a validá-los e a produzir textos técnico-científicos de qualidade.

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

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

Problema 1

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

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

Por exemplo, o histograma

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

que se mostra na figura



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

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

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

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

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

Problema 2

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

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

1. Implemente como anamorfismo de listas a função

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

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

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

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



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

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

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

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

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

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

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](#)” [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 [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

$hindex = \perp$

Problema 2

Primeira parte:

```
smallestPrimeFactor :: (ℤ, ℤ) → ℤ
smallestPrimeFactor = cond ((>)) · ((↑2) × id)) π2
  (cond ((≡ 0) · mod · swap) π1 (smallestPrimeFactor · (succ × id)))
g 1 = i1 ()
g n = i2 (smallestPrimeFactor (2, n), n ÷ smallestPrimeFactor (2, n))
primes = [(g)]
```

Segunda parte:

```
-- mergeTrees
mergeTrees :: (Exp ℤ ℤ, Exp ℤ ℤ) → Exp ℤ ℤ
mergeTrees (Var v1, Var v2) = Term 1 [Var v1, Var v2]
mergeTrees (Var v, Term o sub) = Term o (Var v : sub)
mergeTrees (Term o sub, Var v) = Term o (sub ++ [Var v])
mergeTrees (Term o1 sub1, Term o2 sub2)
  | o1 ≡ o2 = Term o1 (mergeSubtrees sub1 sub2)
  | otherwise = Term 1 [Term o1 sub1, Term o2 sub2]
mergeSubtrees :: [Exp ℤ ℤ] → [Exp ℤ ℤ] → [Exp ℤ ℤ]
mergeSubtrees [] t2 = t2
mergeSubtrees t1 [] = t1
mergeSubtrees (x : xs) ys =
  case find (sameRoot x) ys of
    Just y → mergeTrees (x, y) : mergeSubtrees xs (filter (≠ y) ys)
    Nothing → x : mergeSubtrees xs ys

-- Checks if two trees have the same root
sameRoot :: Exp ℤ ℤ → Exp ℤ ℤ → Bool
sameRoot (Term o1 _) (Term o2 _) = o1 ≡ o2
sameRoot _ _ = False

-- Generate a tree with a prime number as root and its prime factors as children
primeTree :: ℤ → Exp ℤ ℤ
primeTree n = Term 1 (myfoldr buildTree [Var n] (primes n))
where
  buildTree prime acc = [Term prime acc]
```

```

geneCata :: () + (Exp ℤ ℤ, Exp ℤ ℤ) → Exp ℤ ℤ
geneCata = [Term 1 [], mergeTrees]

geneAna :: [ℤ] → () + (Exp ℤ ℤ, [ℤ])
geneAna [] = i1 ()
geneAna (x : xs) = i2 (primeTree x, xs)
prime_tree = hyloList geneCata geneAna

```

Problema 3

```

convolve :: Num a ⇒ [a] → [a] → [a]
convolve = ⊥

```

Problema 4

Definição do tipo:

```

outExpr :: Expr b a → a + (b + (Op, [Expr b a]))
outExpr (V n) = i1 n
outExpr (N n) = (i2 · i1) n
outExpr (T op exprs) = (i2 · i2) (op, exprs)
recExpr :: (a → b1) → b2 + (b3 + (b4, [a])) → b2 + (b3 + (b4, [b1]))
recExpr = baseExpr id id

```

Ana + cata + hylo:

```

cataExpr :: (b2 + (b3 + (Op, [b1])) → b1) → Expr b3 b2 → b1
cataExpr g = g · recExpr (cataExpr g) · outExpr
anaExpr :: (a1 → a2 + (b + (Op, [a1]))) → a1 → Expr b a2
anaExpr g = inExpr · recExpr (anaExpr g) · g
hyloExpr :: (b2 + (b3 + (Op, [c])) → c) → (a → b2 + (b3 + (Op, [a]))) → a → c
hyloExpr h g = cataExpr h · anaExpr g

```

Monad:

```

instance Functor (Expr b) where
  fmap f (V a) = V (f a)
  fmap _ (N b) = N b
  fmap f (T op exprs) = T op (map (fmap f) exprs)
instance Applicative (Expr b) where
  pure :: a → Expr b a
  pure = V
  (V f) < * > x = fmap f x
  (N b) < * > _ = N b
  (T op fs) < * > x = T op (map (< * > x) fs)
instance Monad (Expr b) where
  return :: a → Expr b a
  return = pure
  (≫) :: Expr b a → (a → Expr b b1) → Expr b b1

```

$$\begin{aligned}
t \gg g &= \text{muExpr } (\text{fmap } g \ t) \\
\text{muExpr} &:: \text{Expr } b \ (\text{Expr } b \ a) \rightarrow \text{Expr } b \ a \\
\text{muExpr} &= \text{cataExpr } [\text{id}, \text{inExpr} \cdot i_2] \\
u &:: a \rightarrow \text{Expr } b \ a \\
u &= V
\end{aligned}$$

Maps: Monad: Let expressions:

$$\text{let_exp} = \perp$$

Catamorfismo monádico:

$$\text{mcataExpr } g = \perp$$

Avaliação de expressões:

$$\text{evaluate} = \perp$$

Index

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

bibtex, [4](#)

lhs2TeX, [3–5](#)

makeindex, [4](#)

pdflatex, [3](#)

xymatrix, [5](#)

Cálculo de Programas, [1](#), [3](#)

 Material Pedagógico, [3](#)

Combinador “pointfree”

cata

 Naturais, [5](#)

split, [5](#)

Docker, [3](#)

 container, [3](#), [4](#)

Função

π_1 , [5](#)

π_2 , [5](#)

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

 interpretador

 GHCi, [3](#), [4](#)

 Literate Haskell, [3](#)

Números naturais (\mathbb{N}), [5](#)

Programação

 literária, [3](#), [4](#)

References

- [1] D.E. Knuth. *Literate Programming*. CSLI Lecture Notes Number 27. Stanford University Center for the Study of Language and Information, Stanford, CA, USA, 1992.
- [2] J.N. Oliveira. Program Design by Calculation, 2024. Draft of textbook in preparation. First version: 1998. Current version: Sep. 2024. Informatics Department, University of Minho ([pdf](#)).