Universidade do Minho
Escola de Engenharia

Cálculo de Programas

Trabalho Prático (2025/26)

Lic. em Ciências da Computação

Lic. em Engenharia Informática

Grupo G99

axxxxxx Nome
axxxxxx Nome
axxxxxx Nome

dezembro 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.

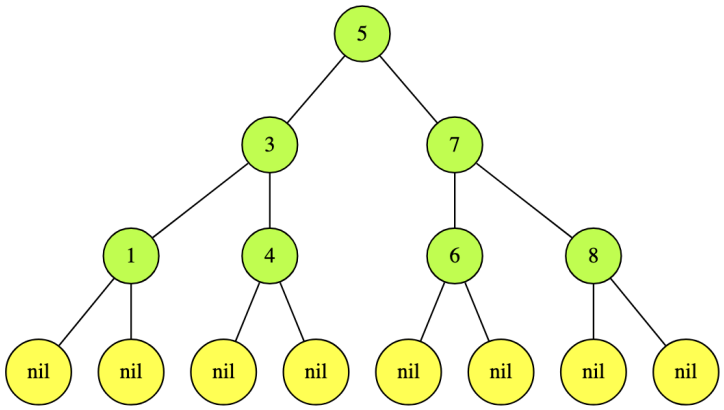
Valoriza-se a escrita de *pouco* código que corresponda a soluções simples e elegantes que utilizem os combinadores de ordem superior estudados na disciplina.

Avaliação. Faz parte da avaliação do trabalho a sua defesa por parte dos elementos de cada grupo. Estes devem estar preparados para responder a perguntas sobre *qualquer* dos problemas deste enunciado. A prestação *individual* de cada aluno nessa defesa oral será uma componente importante e diferenciadora da avaliação.

Problema 1

Uma serialização (ou travessia) de uma árvore é uma sua representação sob a forma de uma lista. Na biblioteca *BTree* encontram-se as funções de serialização *inordt*, *preordt* e *postordt*, que fazem as travessias *in-order*, *pre-order* e *post-order*, respectivamente. Todas essas travessias são catamorfismos que percorrem a árvore argumento em regime *depth-first*.

Pretende-se agora uma função *bforder* que faça a travessia em regime *breadth-first*, isto é, por níveis. Por exemplo, para a árvore t_1 dada em anexo e mostrada na figura a seguir,



a função deverá dar a lista

$$[5, 3, 7, 1, 4, 6, 8]$$

em que se vê como os níveis 5, depois $3, 7$ e finalmente $1, 4, 6, 8$ foram percorridos.

Pretendemos propor duas versões dessa função:

1. Uma delas envolve um catamorfismo de *BTree*s:

$$bfsLevels :: BTree\ a \to [a]$$
$$bfsLevels = concat \cdot levels$$

   Complete a definição desse catamorfismo:

$$levels :: BTree\ a \to [[a]]$$
$$levels = (\!|glevels|\!)$$

2. A segunda proposta,

$$bft :: BTree\ a \to [a]$$

   deverá basear-se num anamorfismo de listas.

Sugestão: estudar o artigo [**?**] cujo PDF está incluído no material deste trabalho. Quando fizer testes ao seu código pode, se desejar, usar funções disponíveis na biblioteca *Exp* para visualizar as árvores em GraphViz (formato .dot).

Justifique devidamente a sua resolução, que deverá vir acompanhada de diagramas explicativos. Como já se disse, 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 2

Considere a seguinte função em Haskell:

$$\begin{aligned}
&f\ x = wrapper \cdot worker\ \mathbf{where}\\
&\quad wrapper = head\\
&\quad worker\ 0 = start\ x\\
&\quad worker\ (n+1) = loop\ x\ (worker\ n)\\
&loop\ x\ [s,\quad h,\quad k,j,\quad m\ ] =\\
&\qquad [h\ /\ k+s, x\uparrow 2 * h, k * j, j+m, m+8]\\
&start\ x = [x, x\uparrow 3, 6, 20, 22]
\end{aligned}$$

Pode-se provar pela lei de recursividade mútua que $f\ x\ n$ calcula o seno hiperbólico de x, $sinh\ x$, para n aproximações da sua série de Taylor. Faça a derivação da função dada a partir da referida série de Taylor, apresentando todos os cálculos justificativos, tal como se faz para outras funções no capítulo respectivo do texto base desta UC [**?**].

Problema 3

Quem em Braga observar, ao fim da tarde, o tráfego onde a Avenida Clairmont Fernand se junta à N101, aproximadamente na coordenada 41°33'46.8"N 8°24'32.4"W — ver as setas da figura que se segue — reparará nas sequências imparáveis (infinitas!) de veículos provenientes dessas vias de circulação.

Mas também irá observar um comportamento interessante por parte dos condutores desses veículos: por regra, *cada carro numa via deixa passar, à sua frente, exactamente outro carro da outra via.*

Este comportamento *civilizado* chama-se *fair-merge* (ou *fair-interleaving*) de duas sequências infinitas, também designadas *streams* em ciência da computação. Seja dado o tipo dessas sequências em Haskell,

$$\textbf{data}\ Stream\ a = Cons\ (a, Stream\ a)\ \textbf{deriving}\ Show$$

para o qual se define também:

$$\text{out}\ (Cons\ (x, xs)) = (x, xs)$$

O referido comportamento civilizado pode definir-se, em Haskell, da forma seguinte:[1]

$$\begin{array}{l} fair\_merge :: (Stream\ a, Stream\ a) + (Stream\ a, Stream\ a) \to Stream\ a \\ fair\_merge = [h, k]\ \textbf{where} \\ \quad h\ (Cons\ (x, xs), y) = Cons\ (x, k\ (xs, y)) \\ \quad k\ (x, Cons\ (y, ys)) = Cons\ (y, h\ (x, ys)) \end{array}$$

Defina *fair_merge* como um **anamorfismo** de *Stream*s, usando o combinador

$$[\!(g)\!] = Cons \cdot (id \times [\!(g)\!]) \cdot g$$

e a seguinte estratégia:

- Derivar a lei **dual** da recursividade mútua,

$$[f, g] = [\!(\,[h, k]\,)\!] \quad \equiv \quad \begin{cases} out \cdot f = \mathsf{F}\ [f, g] \cdot h \\ out \cdot g = \mathsf{F}\ [f, g] \cdot k \end{cases} \tag{1}$$

  tal como se fez, nas aulas, para a que está no formulário.

- Usar (1) na resolução do problema proposto.

Justificar devidamente a resolução, que deverá vir acompanhada de diagramas explicativos.

Problema 4

Como se sabe, é possível pensarmos em catamorfismos, anamorfismos etc *probabilísticos*, quer dizer, programas recursivos que dão distribuições como resultados. Por exemplo, podemos pensar num combinador

$$pcataList :: (() + (a, b) \to \text{Dist}\ b) \to [a] \to \text{Dist}\ b$$

[1] O facto das sequências serem infinitas não nos deve preocupar, pois em Haskell isso é lidado de forma transparente por lazy evaluation.

que é muito parecido com

$$(\!|\cdot|\!) :: ((() + (a,b) \to b) \to [a] \to b$$

da biblioteca List. A principal diferença é que o gene de *pcataList* é uma função probabilística.

Como exemplo de utilização, recorde-se que $(\!|\,[zero, add]\,|\!)$ soma todos os elementos da lista argumento, por exemplo:

$$(\!|\,[zero, add]\,|\!)\ [20, 10, 5] = 35.$$

Considere-se agora a função *padd* (adição probabilística) que, com probabilidade 90% soma dois números e com probabilidade 10% os subtrai:

$$padd\ (a,b) = D\ [(a+b, 0.9), (a-b, 0.1)]$$

Se se correr

$$d4 = pcataList\ [pzero, padd]\ [20, 10, 5]\ \textbf{where}\ pzero = return \cdot zero$$

obter-se-á:

```
35  81.0%
25   9.0%
 5   9.0%
15   1.0%
```

Com base neste exemplo, resolva o seguinte

> ***Problema**: Uma unidade militar pretende enviar uma mensagem urgente a outra, mas tem o aparelho de telegrafia meio avariado. Por experiência, o telegrafista sabe que a probabilidade de uma palavra se perder (não ser transmitida) é 5%; e que, no final de cada mensagem, o aparelho envia o código* `"stop"`*, mas (por estar meio avariado), falha 10% das vezes.*
>
> *Qual a probabilidade de a palavra* `"atacar"` *da mensagem*
>
> $$words\ \texttt{"Vamos atacar hoje"}$$
>
> *se perder, isto é, o resultado da transmissão ser* `["Vamos","hoje","stop"]`*? E a de seguirem todas as palavras, mas faltar o* `"stop"` *no fim? E a da transmissão ser perfeita?*

Responda a estas perguntas encontrando *gene* tal que

$$transmitir = pcataList\ gene$$

descreve o comportamento do aparelho. Justificar devidamente a resolução, que deverá vir acompanhada de diagramas explicativos.

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.

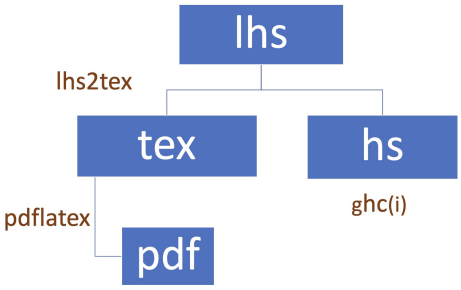
Para cumprir de forma integrada os objectivos do trabalho vamos recorrer a uma técnica de programação dita "literária" [**?**], cujo princípio base é o seguinte:

> *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 `cp2526t.pdf` que está a ler é já um exemplo de programação literária: foi gerado a partir do texto fonte `cp2526t.lhs`[1] que encontrará no material pedagógico desta disciplina descompactando o ficheiro `cp2526t.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 `cp2526t.zip`. Este container deverá ser usado na execução do GHCi e dos comandos relativos ao LaTeX. (Ver também a `Makefile` que é disponibilizada.)

Após instalar o Docker e descarregar o referido zip com o código fonte do trabalho, basta executar os seguintes comandos:

```
$ docker build -t cp2526t .
$ docker run -v ${PWD}:/cp2526t -it cp2526t
```

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}:/cp2526t`) que permite que a diretoria em que se encontra na sua máquina local e a diretoria `/cp2526t` no container sejam partilhadas.

Pretende-se então que visualize/edite os ficheiros na sua máquina local e que os compile no container, executando:

[1] O sufixo 'lhs' quer dizer *literate Haskell*.

```
$ lhs2TeX cp2526t.lhs > cp2526t.tex
$ pdflatex cp2526t
```

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 `cp2526t.lhs` é executável e contém o "kit" básico, escrito em Haskell, para realizar o trabalho. Basta executar

```
$ ghci cp2526t.lhs
```

Abra o ficheiro `cp2526t.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 G 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 cp2526t.aux
$ makeindex cp2526t.idx
```

e recompilar o texto como acima se indicou. (Como já se disse, pode fazê-lo correndo simplesmente `make` no container.)

No anexo F 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]

$$\begin{array}{ll} & id = \langle f, g \rangle \\ \equiv & \quad \{\ \text{universal property}\ \} \end{array}$$

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

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

$$
\begin{cases} \pi_1 \cdot id = f \\ \pi_2 \cdot id = g \end{cases}
$$

$$
\equiv \quad \{ \text{ identity } \}
$$

$$
\begin{cases} \pi_1 = f \\ \pi_2 = g \end{cases}
$$

$$
\square
$$

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 O mónade das distribuições probabilísticas

Mónades são functores com propriedades adicionais que nos permitem obter efeitos especiais em programação. Por exemplo, a biblioteca Probability oferece um mónade para abordar problemas de probabilidades. Nesta biblioteca, o conceito de distribuição estatística é captado pelo tipo

$$
\textbf{newtype}\ \text{Dist}\ a = D\ \{unD :: [(a, ProbRep)]\} \tag{2}
$$

em que $ProbRep$ é um real de 0 a 1, equivalente a uma escala de 0 a 100%.

Cada par (a, p) numa distribuição $d :: \text{Dist}\ a$ indica que a probabilidade de a é p, devendo ser garantida a propriedade de que todas as probabilidades de d somam 100%. Por exemplo, a seguinte distribuição de classificações por escalões de A a E,

A ▬ 2%
B ▬▬▬ 12%
C ▬▬▬▬▬▬▬▬ 29%
D ▬▬▬▬▬▬▬▬▬▬▬ 35%
E ▬▬▬▬▬▬▬ 22%

será representada pela distribuição

$$
\begin{array}{l}
d1 :: \text{Dist}\ Char \\
d1 = D\ [('\texttt{A}', 0.02), ('\texttt{B}', 0.12), ('\texttt{C}', 0.29), ('\texttt{D}', 0.35), ('\texttt{E}', 0.22)]
\end{array}
$$

que o GHCi mostrará assim:

```
'D'  35.0%
'C'  29.0%
'E'  22.0%
'B'  12.0%
'A'   2.0%
```

É possível definir geradores de distribuições, por exemplo distribuições *uniformes*,

$$
d2 = uniform\ (words\ \texttt{"Uma frase de cinco palavras"})
$$

isto é

```
   "Uma"  20.0%
 "cinco"  20.0%
    "de"  20.0%
 "frase"  20.0%
"palavras"  20.0%
```

distribuição *normais*, eg.

$$d3 = normal\ [10\,..\,20]$$

etc.[1] Dist forma um **mónade** cuja unidade é $return\ a = D\ [(a,1)]$ e cuja composição de Kleisli é (simplificando a notação)

$$(f \bullet g)\ a = [(y, q*p)\ |\ (x,p) \leftarrow g\ a, (y,q) \leftarrow f\ x]$$

em que $g : A \to$ Dist B e $f : B \to$ Dist C são funções **monádicas** que representam *computações probabilísticas*.

Este mónade é adequado à resolução de problemas de *probabilidades e estatística* usando programação funcional, de forma elegante e como caso particular da programação monádica.

F Código fornecido

Problema 1

Árvores exemplo:

```
t1 :: BTree Int
t1 = Node (5,(Node (3,(Node (1,(Empty,Empty)),Node (4,(Empty,Empty)))),
   Node (7,(Node (6,(Empty,Empty)),Node (8,(Empty,Empty))))))
t2 :: BTree Int
t2 =
   node 1
      (node 2 (node 4 Empty Empty) (node 5 Empty Empty))
      (node 3 (node 6 Empty Empty) (node 7 Empty Empty))
t3 :: BTree Char
t3 =
   node 'A'
      (node 'B' (node 'C' (node 'D' Empty Empty) Empty) Empty)
      (node 'E' Empty Empty)
t4 :: BTree Char
t4 =
   node 'A'
      (node 'B' (node 'C' (node 'D' Empty Empty) Empty) Empty)
      Empty
t5 :: BTree Int
t5 =
   node 1
```

[1] Para mais detalhes ver o código fonte de Probability, que é uma adaptação da biblioteca PFP ("Probabilistic Functional Programming"). Para quem quiser saber mais recomenda-se a leitura do artigo [**?**].

$(node\ 2\ (node\ 4\ Empty\ Empty)\ Empty)$
$(node\ 3\ Empty\ (node\ 5\ (node\ 6\ Empty\ Empty)\ Empty))$

$node\ a\ b\ c = Node\ (a, (b, c))$

G 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

Problema 1

Uma travessia em largura (breadth-first) de uma árvore processa os nós nível por nível, começando pela raiz. Existem diferentes formas de implementar esta travessia, cada uma explorando um combinador recursivo distinto.

Primeira abordagem: catamorfismo de árvores

A função *levels* agrupa elementos por profundidade, retornando uma lista de listas onde cada sublista representa um nível. Depois, *bfsLevels* concatena simplesmente estes níveis numa única lista, obtendo o resultado breadth-first.

Embora o resultado final seja breadth-first, o catamorfismo processa internamente a árvore de baixo para cima (visitando primeiro as subárvores). Quando chegamos a um nó, temos já processadas as subárvores esquerda e direita, recebendo listas de níveis para cada uma. O nó atual forma um novo nível no topo, e os restantes níveis são fundidos par a par através de *zipLevels*, alinhando elementos da mesma profundidade das duas subárvores.

Segunda abordagem: anamorfismo de listas

A função *bft* implementa uma travessia breadth-first através de um anamorfismo que utiliza uma fila de árvores. Ao contrário do catamorfismo que destrói a estrutura, o anamorfismo constrói incrementalmente a lista resultado.

O algoritmo funciona da seguinte forma: iniciamos com uma fila contendo apenas a raiz da árvore. Depois, repetidamente, removemos a árvore da frente da fila. Se for *Empty*, ignoramo-la e prosseguimos. Se for um nó *Node*, extraímos o seu valor (que é emitido para a lista resultado) e adicionamos os seus filhos no *fim* da fila. Este regime FIFO (first-in-first-out) garante que visitamos os nós exactamente na ordem breadth-first: primeiro todos os nós do nível 0 (raiz), depois nível 1, depois nível 2, e assim sucessivamente.

Implementação:

$glevels :: () + (a, ([[a]], [[a]])) \to [[a]]$
$glevels\ (i_1\ ()) = []$
$glevels\ (i_2\ (a, (ls, rs))) = [a] : zipLevels\ ls\ rs$
  where
    $zipLevels\ []\ rs = rs$

$$
\begin{aligned}
&\quad zipLevels\ ls\ [\,] = ls \\
&\quad zipLevels\ (l : ls)\ (r : rs) = (l +\!\!+ r) : zipLevels\ ls\ rs
\end{aligned}
$$

$$
\begin{aligned}
&bft :: BTree\ a \to [a] \\
&bft\ t = [\!(\ gbf\ )\!]\ [t] \\
&\quad \textbf{where} \\
&\qquad gbf\ [\,] = i_1\ () \\
&\qquad gbf\ (Empty : ts) = gbf\ ts \\
&\qquad gbf\ (Node\ (a, (l, r)) : ts) = i_2\ (a, ts +\!\!+ [l, r])
\end{aligned}
$$

Diagramas e Propriedades:

Catamorfismo *levels*:

$$
\begin{array}{ccc}
BTree\ a & \xleftarrow{outBTree} & 1 + a \times (BTree\ a \times BTree\ a) \\
{\scriptstyle levels}\downarrow & & \downarrow{\scriptstyle id+id\times(levels\times levels)} \\
[[a]] & \xleftarrow[glevels]{} & 1 + a \times ([[a]] \times [[a]])
\end{array}
$$

O diagrama mostra como *levels* decompõe a árvore (via *outBTree*), aplica *levels* recursivamente às subárvores, e depois combina o resultado usando *glevels*. O gene *glevels* trata dois casos: árvore vazia (retorna lista vazia) e nó com valor a e subárvores processadas (cria um novo nível com a e funde os restantes).

Anamorfismo *bft*:

$$
\begin{array}{ccc}
[BTree\ a] & \xleftarrow{gbf} & 1 + a \times [BTree\ a] \\
{\scriptstyle [\!(\ gbf\ )\!]}\downarrow & & \downarrow{\scriptstyle id+id\times[\!(\ gbf\ )\!]} \\
[a] & \xleftarrow[inList]{} & 1 + a \times [a]
\end{array}
$$

O gene *gbf* define o passo de construção da lista. A fila de árvores é consumida por três casos: se está vazia, o processo termina ($i_1\ ()$); se contém uma árvore vazia, ignoramo-la recursivamente; se contém um nó, emitimos o seu valor e continuamos com os restantes elementos da fila mais os seus filhos. A operação $ts +\!\!+ [l, r]$ é essencial: coloca os filhos no fim, não no início, mantendo a ordem breadth-first.

Problema 2

Problema 3

$$
fair\_merge' = [\!(\ \bot\ )\!]
$$

Problema 4

$$
\begin{aligned}
&pcataList = \bot \\
&gene = \bot
\end{aligned}
$$