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

novembro 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.

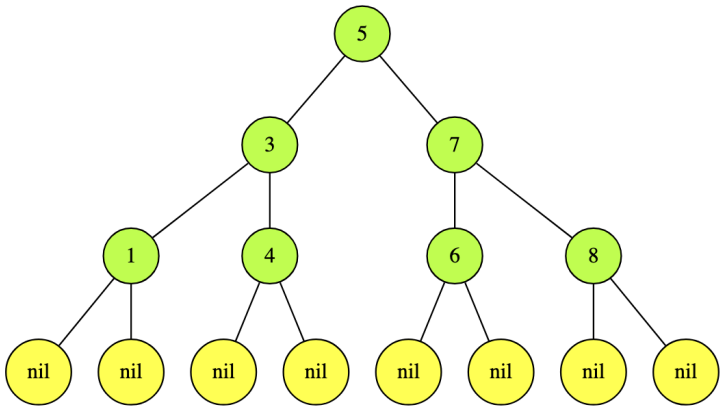
Valoriza-se a escrita de *pouco* código que corresponda a soluções simples e elegantes que utilizem os combinadores de ordem superior estudados na disciplina.

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 [2] 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:

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

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 [3].

Problema 3

A propor na 2ª edição deste enunciado.

Problema 4

A propor na 2ª edição deste enunciado.

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.

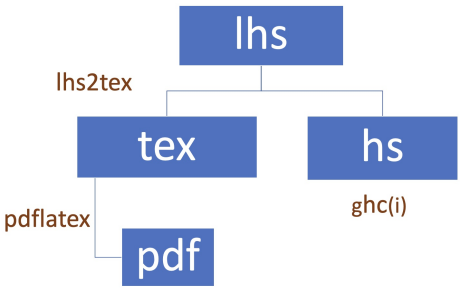
Para cumprir de forma integrada os objectivos do trabalho vamos recorrer a uma técnica de programação dita "literária" [1], cujo princípio base é o seguinte:

> *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 `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

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

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:

```
$ 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 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 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 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]

$$
\begin{aligned}
& id = \langle f, g\rangle \\
\equiv & \quad \{\ \text{universal property}\ \} \\
& \begin{cases} \pi_1 \cdot id = f \\ \pi_2 \cdot id = g \end{cases} \\
\equiv & \quad \{\ \text{identity}\ \} \\
& \begin{cases} \pi_1 = f \\ \pi_2 = g \end{cases} \\
\Box &
\end{aligned}
$$

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

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

E 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)
```

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

[2] Exemplos tirados de [3].

```
      (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
    (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))
```

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

```
concatPointWise :: [[a]] → [[a]] → [[a]]
concatPointWise [] ys = ys
concatPointWise xs [] = xs
concatPointWise (x : xs) (y : ys) = (x ++ y) : concatPointWise xs ys
concatPointFree :: [[a]] → [[a]] → [[a]]
concatPointFree = ⦇gene⦈
    where
      gene = [[g1, g2], [g3, g4]] · (distr + distr) · distl · (outList × outList)
      g1 = i1 ·
      g2 = i2 · (id × ⟨nil, id⟩) · π2
      g3 = i2 · (id × ⟨id, nil⟩) · π1
      g4 = i2 · ⟨conc · (π1 × π1), π2 × π2⟩
  -- Definição da Condicional (dada no enunciado)
  -- cond p f g = (either f g) . (grd p)
concatCondicional :: [[a]] → [[a]] → [[a]]
concatCondicional = ⦇gene⦈
    where
        -- O Gene: Uma cadeia de condicionais (If-Else-If...)
      gene = cond ambasVazias     -- if both empty
        stop     -- stop
        (cond listaEsquerdaVazia     -- if left list
          doListaDireita     -- Se só L1 vazia -> Processa L2
          (cond listaDireitaVazia     -- if right list
            doListaEsquerda     -- Se só L2 vazia -> Processa L1
            doAmbas))    -- Senão -> Processa ambas (caso x:xs, y:ys)
```

```
    -- 1. Predicados
  ambasVazias = (∧)^ · (null × null)    -- ([], [])?
  listaEsquerdaVazia = null · π1    -- ([], ...)?
  listaDireitaVazia = null · π2    -- (..., [])?

    -- 2. Ações (Os ramos)

    -- Caso Paragem: Devolve Left ()
  stop = i1 ·

    -- Caso L1 Vazia ([], y:ys): Produz y, estado ([], ys)
    -- Nota: Usamos head/tail porque o predicado garante que não é vazia
  doListaDireita = i2 · ⟨head · π2, ⟨nil, tail · π2⟩⟩

    -- Caso L2 Vazia (x:xs, []): Produz x, estado (xs, [])
  doListaEsquerda = i2 · ⟨head · π1, ⟨tail · π1, nil⟩⟩

    -- Caso Ambas Cheias (x:xs, y:ys): Produz x++y, estado (xs, ys)
  doAmbas = i2 · ⟨conc · (head × head), tail × tail⟩
-- Define helper e uma árvore de teste (cola isto no GHCi)
glevelsPointWise :: () + (a, ([[a]], [[a]])) → [[a]]
glevelsPointWise (i1 ()) = []
glevelsPointWise (i2 (a, (ls, rs))) = [a] : concatPointWise ls rs
glevelsPointFree :: () + (a, ([[a]], [[a]])) → [[a]]
glevelsPointFree = [nil, cons · (singl × concatPointFree)^]
genePointFree :: [BTree a] → () + (a, [BTree a])
genePointFree [] = i1 ()
genePointFree (Empty : ts) = genePointFree ts
genePointFree (Node (a, (l, r)) : ts) = i2 (a, ts ++ [l, r])
genePointWise :: [BTree a] → () + (a, [BTree a])
genePointWise [] = i1 ()
genePointWise (Empty : ts) = genePointWise ts
genePointWise (Node (a, (l, r)) : ts) = gNode (a, (ts, (l, r)))
  where
    gNode = i2
      · (id × conc)
      · (id × (id × conc))
      · (id × (id × (singl × singl)))
  -- g1 . cons.(Node >< id).assocr.(id >< swap) = i2. (id >< conc) . (id >< (id >< conc)) . (id >< (id >< (singl >< singl)))
bft t = ⟦ genePointWise ⟧ (singl t)
```

Problema 2

Problema 3

Problema 4

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] Chris Okasaki. Breadth-first numbering: lessons from a small exercise in algorithm design. In Martin Odersky and Philip Wadler, editors, *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000*, pages 131–136. ACM, 2000.

[3] 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).