

Cálculo de Programas

Trabalho Prático

LEI — 2022/23

Departamento de Informática
Universidade do Minho

Janeiro de 2023

Grupo nr.	4
a96766	Guilherme Sampaio
a96681	Luís Pereira
a95254	Rui Oliveira
a95104	Tiago Pereira

Preâmbulo

Cálculo de Programas tem como objectivo principal ensinar a programação de computadores como uma disciplina científica. Para isso parte-se de um repertório de *combinadores* que formam uma álgebra da programação (conjunto de leis universais e seus corolários) 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.

Problema 1

Suponha-se uma sequência numérica semelhante à sequência de Fibonacci tal que cada termo subsequente aos três primeiros corresponde à soma dos três anteriores, sujeitos aos coeficientes a , b e c :

$$\begin{aligned}f\ a\ b\ c\ 0 &= 0 \\f\ a\ b\ c\ 1 &= 1 \\f\ a\ b\ c\ 2 &= 1 \\f\ a\ b\ c\ (n+3) &= a * f\ a\ b\ c\ (n+2) + b * f\ a\ b\ c\ (n+1) + c * f\ a\ b\ c\ n\end{aligned}$$

Assim, por exemplo, $f\ 1\ 1\ 1$ irá dar como resultado a sequência:

1, 1, 2, 4, 7, 13, 24, 44, 81, 149, ...

$f\ 1\ 2\ 3$ irá gerar a sequência:

1, 1, 3, 8, 17, 42, 100, 235, 561, 1331, ...

etc.

A definição de f dada é muito ineficiente, tendo uma degradação do tempo de execução exponencial. Pretende-se otimizar a função dada convertendo-a para um ciclo *for*. Recorrendo à lei de recursividade mútua, calcule *loop* e *initial* em

$$fbl\ a\ b\ c = wrap \cdot for\ (loop\ a\ b\ c)\ initial$$

por forma a f e fbl serem (matematicamente) a mesma função. Para tal, poderá usar a regra prática explicada no anexo B.

Valorização: apresente testes de *performance* que mostrem quão mais rápida é fbl quando comparada com f .

Problema 2

Pretende-se vir a classificar os conteúdos programáticos de todas as UCs lecionadas no *Departamento de Informática* de acordo com o [ACM Computing Classification System](#). A listagem da taxonomia desse sistema está disponível no ficheiro Cp2223data, começando com

```
acm_ccs = ["CCS",
           "    General and reference",
           "        Document types",
           "            Surveys and overviews",
           "            Reference works",
           "            General conference proceedings",
           "            Biographies",
           "            General literature",
           "            Computing standards, RFCs and guidelines",
           "            Cross-computing tools and techniques",
```

(10 primeiros itens) etc., etc.¹

Pretende-se representar a mesma informação sob a forma de uma árvore de expressão, usando para isso a biblioteca [Exp](#) que consta do material pedagógico da disciplina e que vai incluída no zip do projecto, por ser mais conveniente para os alunos.

1. Comece por definir a função de conversão do texto dado em *acm_ccs* (uma lista de *strings*) para uma tal árvore como um anamorfismo de [Exp](#):

$$\begin{aligned} tax &:: [String] \rightarrow Exp\ String\ String \\ tax &= [(gene)]_{Exp} \end{aligned}$$

Ou seja, defina o *gene* do anamorfismo, tendo em conta o seguinte diagrama²:

$$\begin{array}{ccc} Exp\ S\ S & \xleftarrow{\text{in}_{Exp}} & S + S \times (Exp\ S\ S)^* \\ \uparrow tax & & \uparrow id + id \times tax^* \\ S^* & \xrightarrow{\text{out}} S + S \times S^* \xrightarrow{\dots} S + S \times (S^*)^* \\ & \searrow gene & \end{array}$$

Para isso, tome em atenção que cada nível da hierarquia é, em *acm_ccs*, marcado pela indentação de 4 espaços adicionais — como se mostra no fragmento acima.

Na figura 1 mostra-se a representação gráfica da árvore de tipo [Exp](#) que representa o fragmento de *acm_ccs* mostrado acima.

2. De seguida vamos querer todos os caminhos da árvore que é gerada por *tax*, pois a classificação de uma UC pode ser feita a qualquer nível (isto é, caminho descendente da raiz "CCS" até um subnível ou folha).³

¹Informação obtida a partir do site [ACM CCS](#) seleccionando *Flat View*.

² S abrevia *String*.

³Para um exemplo de classificação de UC concreto, pf. ver a secção **Classificação ACM** na página pública de [Cálculo de Programas](#).



Figura 1: Fragmento de *acm_ccs* representado sob a forma de uma árvore do tipo [Exp](#).

Precisamos pois da composição de *tax* com uma função de pós-processamento *post*,

$$\begin{aligned} tudo &:: [String] \rightarrow [[String]] \\ tudo &= post \cdot tax \end{aligned}$$

para obter o efeito que se mostra na tabela 1.

CCS			
CCS	General and reference		
CCS	General and reference	Document types	
CCS	General and reference	Document types	Surveys and overviews
CCS	General and reference	Document types	Reference works
CCS	General and reference	Document types	General conference proceedings
CCS	General and reference	Document types	Biographies
CCS	General and reference	Document types	General literature
CCS	General and reference	Cross-computing tools and techniques	

Tabela 1: Taxonomia ACM fechada por prefixos (10 primeiros ítems).

Defina a função *post* :: *Exp String String* \rightarrow $[[String]]$ da forma mais económica que encontrar.

Sugestão: Inspeccione as bibliotecas fornecidas à procura de funções auxiliares que possa re-utilizar para a sua solução ficar mais simples. Não se esqueça que, para o mesmo resultado, nesta disciplina “ganha” quem escrever menos código!

Sugestão: Para efeitos de testes intermédios não use a totalidade de *acm_ccs*, que tem 2114 linhas! Use, por exemplo, *take 10 acm_ccs*, como se mostrou acima.

Problema 3

O [tapete de Sierpinski](#) é uma figura geométrica [fractal](#) em que um quadrado é subdividido recursivamente em sub-quadrados. A construção clássica do tapete de Sierpinski é a seguinte: assumindo um quadrado de lado l , este é subdividido em 9 quadrados iguais de lado $l/3$, removendo-se o quadrado central. Este passo é depois repetido sucessivamente para cada um dos 8 sub-quadrados restantes (Fig. 2).

NB: No exemplo da fig. 2, assumindo a construção clássica já referida, os quadrados estão a branco e o fundo a verde.

A complexidade deste algoritmo, em função do número de quadrados a desenhar, para uma profundidade n , é de 8^n (exponencial). No entanto, se assumirmos que os quadrados a desenhar são os que estão a verde, a complexidade é reduzida para $\sum_{i=0}^{n-1} 8^i$, obtendo um ganho de $\sum_{i=1}^n \frac{100}{8^i} \%$. Por exemplo, para $n = 5$, o ganho é de 14.28%. O objetivo deste problema é a implementação do algoritmo mediante a referida otimização.

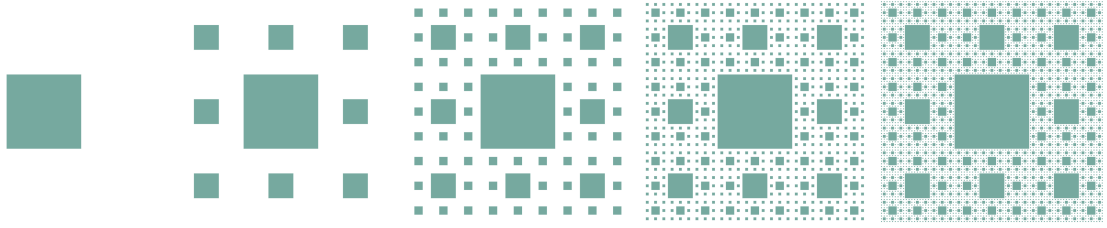


Figura 2: Construção do tapete de Sierpinski com profundidade 5.

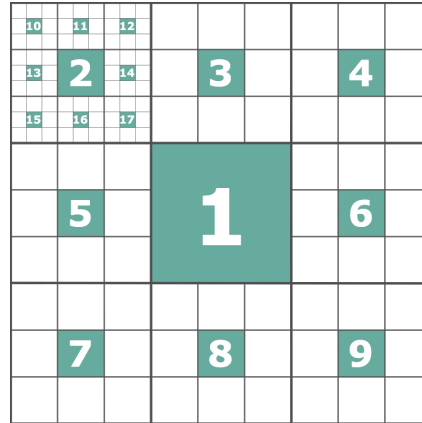


Figura 3: Tapete de Sierpinski com profundidade 2 e com os quadrados enumerados.

Assim, seja cada quadrado descrito geometricamente pelas coordenadas do seu vértice inferior esquerdo e o comprimento do seu lado:

type *Square* = (*Point*, *Side*)
type *Side* = *Double*
type *Point* = (*Double*, *Double*)

A estrutura recursiva de suporte à construção de tapetes de Sierpinski será uma [Rose Tree](#), na qual cada nível da árvore irá guardar os quadrados de tamanho igual. Por exemplo, a construção da fig. 3 poderá⁴ corresponder à árvore da figura 4.

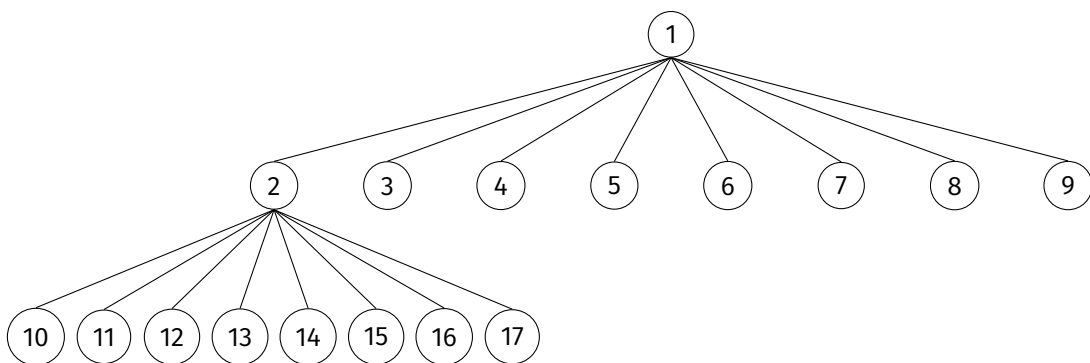


Figura 4: Possível árvore de suporte para a construção da fig. 3.

Uma vez que o tapete é também um quadrado, o objetivo será, a partir das informações do tapete (coordenadas do vértice inferior esquerdo e comprimento do lado), desenhar o quadrado central, subdividir o tapete nos 8 sub-tapetes restantes, e voltar a desenhar, recursivamente, o quadrado nesses 8 sub-tapetes. Desta forma, cada tapete determina o seu quadrado e os seus 8 sub-tapetes. No exemplo em cima, o tapete que contém o quadrado 1 determina esse próprio quadrado e determina os sub-tapetes que contêm os quadrados 2 a 9.

⁴A ordem dos filhos não é relevante.

Portanto, numa primeira fase, dadas as informações do tapete, é construída a árvore de suporte com todos os quadrados a desenhar, para uma determinada profundidade.

$$squares :: (Square, Int) \rightarrow Rose\ Square$$

NB: No programa, a profundidade começa em 0 e não em 1.

Uma vez gerada a árvore com todos os quadrados a desenhar, é necessário extrair os quadrados para uma lista, a qual é processada pela função *drawSq*, disponibilizada no anexo [D](#).

$$rose2List :: Rose\ a \rightarrow [a]$$

Assim, a construção de tapetes de Sierpinski é dada por um hilomorfismo de *Rose Trees*:

$$\begin{aligned} sierpinski &:: (Square, Int) \rightarrow [Square] \\ sierpinski &= \llbracket gr2l, gsq \rrbracket_r \end{aligned}$$

Trabalho a fazer:

1. Definir os genes do hilomorfismo *sierpinski*.
2. Correr

```
sierp4 = drawSq (sierpinski (((0,0),32),3))
constructSierp5 = do drawSq (sierpinski (((0,0),32),0))
  await
  drawSq (sierpinski (((0,0),32),1))
  await
  drawSq (sierpinski (((0,0),32),2))
  await
  drawSq (sierpinski (((0,0),32),3))
  await
  drawSq (sierpinski (((0,0),32),4))
  await
```

3. Definir a função que apresenta a construção do tapete de Sierpinski como é apresentada em *construcaoSierp5*, mas para uma profundidade $n \in \mathbb{N}$ recebida como parâmetro.

$$\begin{aligned} constructSierp &:: Int \rightarrow IO\ [] \\ constructSierp &= present \cdot carpets \end{aligned}$$

Dica: a função *constructSierp* será um hilomorfismo de listas, cujo anamorfismo *carpets* $:: Int \rightarrow [[Square]]$ constrói, recebendo como parâmetro a profundidade n , a lista com todos os tapetes de profundidade $1..n$, e o catamorfismo *present* $:: [[Square]] \rightarrow IO\ []$ percorre a lista desenhando os tapetes e esperando 1 segundo de intervalo.

Problema 4

Este ano ocorrerá a vigésima segunda edição do Campeonato do Mundo de Futebol, organizado pela Federação Internacional de Futebol (FIFA), a decorrer no Qatar e com o jogo inaugural a 20 de Novembro.

Uma casa de apostas pretende calcular, com base numa aproximação dos *rankings*⁵ das seleções, a probabilidade de cada seleção vencer a competição.

Para isso, o diretor da casa de apostas contratou o Departamento de Informática da Universidade do Minho, que atribuiu o projeto à equipa formada pelos alunos e pelos docentes de Cálculo de Programas.

⁵Os *rankings* obtidos [aqui](#) foram escalados e arredondados.

Para resolver este problema de forma simples, ele será abordado por duas fases:

1. versão acadêmica sem probabilidades, em que se sabe à partida, num jogo, quem o vai vencer;
2. versão realista com probabilidades usando o mónade *Dist* (distribuições probabilísticas) que vem descrito no anexo C.

A primeira versão, mais simples, deverá ajudar a construir a segunda.

Descrição do problema

Uma vez garantida a qualificação (já ocorrida), o campeonato consta de duas fases consecutivas no tempo:

1. fase de grupos;
2. fase eliminatória (ou “mata-mata”, como é habitual dizer-se no Brasil).

Para a fase de grupos, é feito um sorteio das 32 seleções (o qual já ocorreu para esta competição) que as coloca em 8 grupos, 4 seleções em cada grupo. Assim, cada grupo é uma lista de seleções.

Os grupos para o campeonato deste ano são:

```
type Team = String
type Group = [Team]
groups :: [Group]
groups = [ ["Qatar", "Ecuador", "Senegal", "Netherlands"],
  ["England", "Iran", "USA", "Wales"],
  ["Argentina", "Saudi Arabia", "Mexico", "Poland"],
  ["France", "Denmark", "Tunisia", "Australia"],
  ["Spain", "Germany", "Japan", "Costa Rica"],
  ["Belgium", "Canada", "Morocco", "Croatia"],
  ["Brazil", "Serbia", "Switzerland", "Cameroon"],
  ["Portugal", "Ghana", "Uruguay", "Korea Republic"] ]
```

Deste modo, *groups !! 0* corresponde ao grupo A, *groups !! 1* ao grupo B, e assim sucessivamente. Nesta fase, cada seleção de cada grupo vai defrontar (uma vez) as outras do seu grupo.

Passam para o “mata-mata” as duas seleções que mais pontuarem em cada grupo, obtendo pontos, por cada jogo da fase grupos, da seguinte forma:

- vitória — 3 pontos;
- empate — 1 ponto;
- derrota — 0 pontos.

Como se disse, a posição final no grupo irá determinar se uma seleção avança para o “mata-mata” e, se avançar, que possíveis jogos terá pela frente, uma vez que a disposição das seleções está desde o início definida para esta última fase, conforme se pode ver na figura 5.

Assim, é necessário calcular os vencedores dos grupos sob uma distribuição probabilística. Uma vez calculadas as distribuições dos vencedores, é necessário colocá-las nas folhas de uma *LTree* de forma a fazer um *match* com a figura 5, entrando assim na fase final da competição, o tão esperado “mata-mata”. Para avançar nesta fase final da competição (i.e. subir na árvore), é preciso ganhar, quem perder é automaticamente eliminado (“mata-mata”). Quando uma seleção vence um jogo, sobe na árvore, quando perde, fica pelo caminho. Isto significa que a seleção vencedora é aquela que vence todos os jogos do “mata-mata”.

Arquitetura proposta

A visão composicional da equipa permitiu-lhe perceber desde logo que o problema podia ser dividido, independentemente da versão, probabilística ou não, em duas partes independentes — a da fase de grupos e a do “mata-mata”. Assim, duas sub-equipas poderiam trabalhar em paralelo, desde que se



Figura 5: O “mata-mata”

garantissem a composicionalidade das partes. Decidiu-se que os alunos desenvolveriam a parte da fase de grupos e os docentes a do “mata-mata”.

Versão não probabilística

O resultado final (não probabilístico) é dado pela seguinte função:

```
winner :: Team
winner = wcup groups
wcup = knockoutStage · groupStage
```

A sub-equipa dos docentes já entregou a sua parte:

```
knockoutStage = ([id, koCriteria])
```

Considere-se agora a proposta do *team leader* da sub-equipa dos alunos para o desenvolvimento da fase de grupos:

Vamos dividir o processo em 3 partes:

- gerar os jogos,
- simular os jogos,
- preparar o “mata-mata” gerando a árvore de jogos dessa fase (fig. 5).

Assim:

```
groupStage :: [Group] → LTree Team
groupStage = initKnockoutStage · simulateGroupStage · genGroupStageMatches
```

Começamos então por definir a função *genGroupStageMatches* que gera os jogos da fase de grupos:

```
genGroupStageMatches :: [Group] → [[Match]]
genGroupStageMatches = map generateMatches
```

onde

```
type Match = (Team, Team)
```

Ora, sabemos que nos foi dada a função

```
gsCriteria :: Match → Maybe Team
```

que, mediante um certo critério, calcula o resultado de um jogo, retornando *Nothing* em caso de empate, ou a equipa vencedora (sob o construtor *Just*). Assim, precisamos de definir a função

```
simulateGroupStage :: [[Match]] → [[Team]]
simulateGroupStage = map (groupWinners gsCriteria)
```

que simula a fase de grupos e dá como resultado a lista dos vencedores, recorrendo à função `groupWinners`:

```
groupWinners criteria = best 2 · consolidate · (>>=matchResult criteria)
```

Aqui está apenas em falta a definição da função `matchResult`.

Por fim, teremos a função `initKnockoutStage` que produzirá a [LTree](#) que a sub-equipa do “mata-mata” precisa, com as devidas posições. Esta será a composição de duas funções:

```
initKnockoutStage = [gl] · arrangement
```

Trabalho a fazer:

1. Definir uma alternativa à função genérica `consolidate` que seja um catamorfismo de listas:

```
consolidate' :: (Eq a, Num b) ⇒ [(a,b)] → [(a,b)]
consolidate' = [c]gene
```

2. Definir a função `matchResult :: (Match → Maybe Team) → Match → [(Team, Int)]` que apura os pontos das equipas de um dado jogo.
3. Definir a função genérica `pairup :: Eq b ⇒ [b] → [(b,b)]` em que `generateMatches` se baseia.
4. Definir o gene `gl`.

Versão probabilística

Nesta versão, mais realista, `gsCriteria :: Match → (Maybe Team)` dá lugar a

```
pgsCriteria :: Match → Dist (Maybe Team)
```

que dá, para cada jogo, a probabilidade de cada equipa vencer ou haver um empate. Por exemplo, há 50% de probabilidades de Portugal empatar com a Inglaterra,

```
pgsCriteria("Portugal", "England")
  Nothing  50.0%
  Just "England"  26.7%
  Just "Portugal"  23.3%
```

etc.

O que é `Dist`? É o mónade que trata de distribuições probabilísticas e que é descrito no anexo [C](#), página [11](#) e seguintes. O que há a fazer? Eis o que diz o vosso *team leader*:

O que há a fazer nesta versão é, antes de mais, avaliar qual é o impacto de `gsCriteria` virar monádica (em `Dist`) na arquitetura geral da versão anterior. Há que reduzir esse impacto ao mínimo, escrevendo-se tão pouco código quanto possível!

Todos lembraram algo que tinham aprendido nas aulas teóricas a respeito da “monadificação” do código: há que reutilizar o código da versão anterior, monadificando-o.

Para distinguir as duas versões decidiu-se afixar o prefixo ‘p’ para identificar uma função que passou a ser monádica.

A sub-equipa dos docentes fez entretanto a monadificação da sua parte:

```
pwinner :: Dist Team
pwinner = pwcup groups
```


$pwcup = pknockoutStage \bullet pgroupStage$

E entregou ainda a versão probabilística do “mata-mata”:

```
pknockoutStage = mcataLTree' [return,pkoCriteria]
mcataLTree' g = k where
  k (Leaf a) = g1 a
  k (Fork (x,y)) = mmbin g2 (k x,k y)
  g1 = g · i1
  g2 = g · i2
```

A sub-equipa dos alunos também já adiantou trabalho,

$pgroupStage = pinitKnockoutStage \bullet psimulateGroupStage \cdot genGroupStageMatches$

mas faltam ainda *pinitKnockoutStage* e *pgroupWinners*, esta usada em *psimulateGroupStage*, que é dada em anexo.

Trabalho a fazer:

- Definir as funções que ainda não estão implementadas nesta versão.
- **Valorização:** experimentar com outros critérios de “ranking” das equipas.

Importante: (a) código adicional terá que ser colocado no anexo E, obrigatoriamente; (b) todo o código que é dado não pode ser alterado.

Anexos

A Documentação para realizar o trabalho

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 `cp2223t.pdf` que está a ler é já um exemplo de [programação literária](#): foi gerado a partir do texto fonte `cp2223t.lhs`⁶ que encontrará no [material pedagógico](#) desta disciplina descompactando o ficheiro `cp2223t.zip` e executando:

```
$ lhs2TeX cp2223t.lhs > cp2223t.tex
$ pdflatex cp2223t
```

em que [lhs2tex](#) é um pré-processor que faz “pretty printing” de código Haskell em [L^AT_EX](#) e que deve desde já instalar utilizando o utilitário [cabal](#) disponível em [haskell.org](#).

Por outro lado, o mesmo ficheiro `cp2223t.lhs` é executável e contém o “kit” básico, escrito em [Haskell](#), para realizar o trabalho. Basta executar

```
$ ghci cp2223t.lhs
```

Abra o ficheiro `cp2223t.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.

⁶O sufixo ‘lhs’ quer dizer *literate Haskell*.

A.1 Como realizar o trabalho

Este trabalho teórico-prático deve ser realizado por grupos de 3 (ou 4) 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.

Em que consiste, então, o *relatório* a que se refere o parágrafo anterior? É a edição do texto que está a ser lido, preenchendo o anexo E 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 cp2223t.aux
$ makeindex cp2223t.idx
```

e recompilar o texto como acima se indicou.

No anexo D, disponibiliza-se algum código [Haskell](#) relativo aos problemas apresentados. Esse anexo deverá ser consultado e analisado à medida que isso for necessário.

A.2 Como exprimir cálculos e diagramas em LaTeX/lhs2tex

Como primeiro exemplo, estudar o texto fonte deste trabalho para obter o efeito:⁷

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

Os diagramas podem ser produzidos recorrendo à *package* \LaTeX [xymatrix](#), por exemplo:

$$\begin{array}{ccc} \mathbb{N}_0 & \xleftarrow{\text{in}} & 1 + \mathbb{N}_0 \\ \text{\scriptsize $\langle g \rangle$} \downarrow & & \downarrow \text{\scriptsize $id + \langle g \rangle$} \\ B & \xleftarrow{g} & 1 + B \end{array}$$

B Regra prática para a recursividade mútua em \mathbb{N}_0

Nesta disciplina estudou-se como fazer [programação dinâmica](#) por cálculo, recorrendo à lei de recursividade mútua.⁸

Para o caso de funções sobre os números naturais (\mathbb{N}_0 , com functor $F X = 1 + X$) é fácil derivar-se da lei que foi estudada uma *regra de algibeira* que se pode ensinar a programadores que não tenham estudado [Cálculo de Programas](#). Apresenta-se de seguida essa regra, tomando como exemplo o cálculo do ciclo-for que implementa a função de Fibonacci, recordar o sistema:

$$\begin{aligned} fib\ 0 &= 1 \\ fib\ (n+1) &= f\ n \end{aligned}$$

⁷Exemplos tirados de [?].

⁸Lei (3.95) em [?], página 112.

$$f\ 0 = 1$$

$$f\ (n + 1) = fib\ n + f\ n$$

Obter-se-á de imediato

$$fib' = \pi_1 \cdot \text{for loop init where}$$

$$\text{loop } (fib, f) = (f, fib + f)$$

$$\text{init} = (1, 1)$$

usando as regras seguintes:

- O corpo do ciclo *loop* terá tantos argumentos quanto o número de funções mutuamente recursivas.
- Para as variáveis escolhem-se os próprios nomes das funções, pela ordem que se achar conveniente.⁹
- Para os resultados vão-se buscar as expressões respectivas, retirando a variável *n*.
- Em *init* colecionam-se os resultados dos casos de base das funções, pela mesma ordem.

Mais um exemplo, envolvendo polinómios do segundo grau $ax^2 + bx + c$ em \mathbb{N}_0 . Seguindo o método estudado nas aulas¹⁰, de $f\ x = ax^2 + bx + c$ derivam-se duas funções mutuamente recursivas:

$$f\ 0 = c$$

$$f\ (n + 1) = f\ n + k\ n$$

$$k\ 0 = a + b$$

$$k\ (n + 1) = k\ n + 2\ a$$

Seguindo a regra acima, calcula-se de imediato a seguinte implementação, em Haskell:

$$f'\ a\ b\ c = \pi_1 \cdot \text{for loop init where}$$

$$\text{loop } (f, k) = (f + k, k + 2 * a)$$

$$\text{init} = (c, a + b)$$

C O mónade das distribuições probabilísticas

Mónades são funtores 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

$$\text{newtype Dist } a = D\ \{ \text{unD} :: [(a, \text{ProbRep})] \}$$
 (1)

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,



será representada pela distribuição

$$d_1 :: \text{Dist Char}$$

$$d_1 = D\ [('A', 0.02), ('B', 0.12), ('C', 0.29), ('D', 0.35), ('E', 0.22)]$$

que o [GHCi](#) mostrará assim:

⁹Podem obviamente usar-se outros símbolos, mas numa primeira leitura dá jeito usarem-se tais nomes.
¹⁰Secção 3.17 de [?] e tópico [Recursividade mútua](#) nos vídeos de apoio às aulas teóricas.

```
'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*,

$$d_2 = \text{uniform}(\text{words "Uma frase de cinco palavras"})$$

isto é

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

distribuição *normais*, eg.

$$d_3 = \text{normal} [10..20]$$

etc.¹¹ Dist forma um **mónade** cuja unidade é $\text{return } a = D [(a, 1)]$ e cuja composição de Kleisli é (simplificando a notação)

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

em que $g : A \rightarrow \text{Dist } B$ e $f : B \rightarrow \text{Dist } C$ 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.

D Código fornecido

Problema 1

Alguns testes para se validar a solução encontrada:

```
test a b c = map (fbl a b c) x ≡ map (f a b c) x where x = [1..20]
test1 = test 1 2 3
test2 = test (-2) 1 5
```

Problema 2

Verificação: a árvore de tipo [Exp](#) gerada por

$$\text{acm_tree} = \text{tax acm_ccs}$$

deverá verificar as propriedades seguintes:

- $\text{expDepth acm_tree} \equiv 7$ (profundidade da árvore);
- $\text{length (expOps acm_tree)} \equiv 432$ (número de nós da árvore);
- $\text{length (expLeaves acm_tree)} \equiv 1682$ (número de folhas da árvore).¹²

O resultado final

$$\text{acm_xls} = \text{post acm_tree}$$

não deverá ter tamanho inferior ao total de nodos e folhas da árvore.

¹¹Para mais detalhes ver o código fonte de [Probability](#), que é uma adaptação da biblioteca [PHP](#) ("Probabilistic Functional Programming"). Para quem quiser saber mais recomenda-se a leitura do artigo [?].

¹²Quer dizer, o número total de nodos e folhas é 2114, o número de linhas do texto dado.

Problema 3

Função para visualização em SVG:

```
drawSq x = picd' [Svg.scale 0.44 (0,0) (x >>= sq2svg)]
sq2svg (p,l) = (color "#67AB9F" · polyg) [p,p ·+ (0,l),p ·+ (l,l),p ·+ (l,0)]
```

Para efeitos de temporização:

```
await = threadDelay 1000000
```

Problema 4

Rankings:

```
rankings = [
  ("Argentina",4.8),
  ("Australia",4.0),
  ("Belgium",5.0),
  ("Brazil",5.0),
  ("Cameroon",4.0),
  ("Canada",4.0),
  ("Costa Rica",4.1),
  ("Croatia",4.4),
  ("Denmark",4.5),
  ("Ecuador",4.0),
  ("England",4.7),
  ("France",4.8),
  ("Germany",4.5),
  ("Ghana",3.8),
  ("Iran",4.2),
  ("Japan",4.2),
  ("Korea Republic",4.2),
  ("Mexico",4.5),
  ("Morocco",4.2),
  ("Netherlands",4.6),
  ("Poland",4.2),
  ("Portugal",4.6),
  ("Qatar",3.9),
  ("Saudi Arabia",3.9),
  ("Senegal",4.3),
  ("Serbia",4.2),
  ("Spain",4.7),
  ("Switzerland",4.4),
  ("Tunisia",4.1),
  ("USA",4.4),
  ("Uruguay",4.5),
  ("Wales",4.3)]
```

Geração dos jogos da fase de grupos:

```
generateMatches = pairup
```

Preparação da árvore do “mata-mata”:

```
arrangement = (>>swapTeams) · chunksOf 4 where
  swapTeams [[a1,a2],[b1,b2],[c1,c2],[d1,d2]] = [a1,b2,c1,d2,b1,a2,d1,c2]
```

Função proposta para se obter o *ranking* de cada equipa:

$rank\ x = 4 ** (pap\ rankings\ x - 3.8)$

Cr terio para a simula  o n o probabil stica dos jogos da fase de grupos:

$gsCriteria = s \cdot \langle id \times id, rank \times rank \rangle$ **where**
 $s\ ((s_1, s_2), (r_1, r_2)) = \text{let } d = r_1 - r_2 \text{ in}$
if $d > 0.5$ **then** $Just\ s_1$
else if $d < -0.5$ **then** $Just\ s_2$
else $Nothing$

Cr terio para a simula  o n o probabil stica dos jogos do mata-mata:

$koCriteria = s \cdot \langle id \times id, rank \times rank \rangle$ **where**
 $s\ ((s_1, s_2), (r_1, r_2)) = \text{let } d = r_1 - r_2 \text{ in}$
if $d \equiv 0$ **then** s_1
else if $d > 0$ **then** s_1 **else** s_2

Cr terio para a simula  o probabil stica dos jogos da fase de grupos:

$pgsCriteria = s \cdot \langle id \times id, rank \times rank \rangle$ **where**
 $s\ ((s_1, s_2), (r_1, r_2)) =$
if $abs\ (r_1 - r_2) > 0.5$ **then** $fmap\ Just\ (pkoCriteria\ (s_1, s_2))$ **else** $f\ (s_1, s_2)$
 $f = D \cdot ((Nothing, 0.5) :) \cdot map\ (Just \times (/2)) \cdot unD \cdot pkoCriteria$

Cr terio para a simula  o probabil stica dos jogos do mata-mata:

$pkoCriteria\ (e_1, e_2) = D\ [(e_1, 1 - r_2 / (r_1 + r_2)), (e_2, 1 - r_1 / (r_1 + r_2))]$ **where**
 $r_1 = rank\ e_1$
 $r_2 = rank\ e_2$

Vers o probabil stica da simula  o da fase de grupos:¹³

$psimulateGroupStage = trim \cdot map\ (pgroupWinners\ pgsCriteria)$
 $trim = top\ 5 \cdot sequence \cdot map\ (filterP \cdot norm)$ **where**
 $filterP\ (D\ x) = D\ [(a, p) \mid (a, p) \leftarrow x, p > 0.0001]$
 $top\ n = vec2Dist \cdot take\ n \cdot reverse \cdot presort\ \pi_2 \cdot unD$
 $vec2Dist\ x = D\ [(a, n / t) \mid (a, n) \leftarrow x]$ **where** $t = sum\ (map\ \pi_2\ x)$

Vers o mais eficiente da *pwinner* dada no texto principal, para diminuir o tempo de cada simula  o:

$pwinner :: Dist\ Team$
 $pwinner = mbin\ f\ x \gg= pknockoutStage$ **where**
 $f\ (x, y) = initKnockoutStage\ (x ++ y)$
 $x = \langle g \cdot take\ 4, g \cdot drop\ 4 \rangle\ groups$
 $g = psimulateGroupStage \cdot genGroupStageMatches$

Auxiliares:

$best\ n = map\ \pi_1 \cdot take\ n \cdot reverse \cdot presort\ \pi_2$
 $consolidate :: (Num\ d, Eq\ d, Eq\ b) \Rightarrow [(b, d)] \rightarrow [(b, d)]$
 $consolidate = map\ (id \times sum) \cdot collect$
 $collect :: (Eq\ a, Eq\ b) \Rightarrow [(a, b)] \rightarrow [(a, [b])]$
 $collect\ x = nub\ [k \mapsto [d' \mid (k', d') \leftarrow x, k' \equiv k] \mid (k, d) \leftarrow x]$

Fun  o bin ria mon dica *f*:

$mmbin :: Monad\ m \Rightarrow ((a, b) \rightarrow m\ c) \rightarrow (m\ a, m\ b) \rightarrow m\ c$
 $mmbin\ f\ (a, b) = \text{do } \{x \leftarrow a; y \leftarrow b; f\ (x, y)\}$

Monadifica  o de uma fun  o bin ria *f*:

¹³Faz-se "trimming" das distribu  es para reduzir o tempo de simula  o.

$$mbin :: Monad\ m \Rightarrow ((a,b) \rightarrow c) \rightarrow (m\ a, m\ b) \rightarrow m\ c$$

$$mbin = mmbin \cdot (return \cdot)$$

Outras funções que podem ser úteis:

$$(f\ 'is'\ v)\ x = (f\ x) \equiv v$$

$$rcons\ (x,a) = x ++ [a]$$

E 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, diagramas e/ou outras funções auxiliares que sejam necessárias.

Valoriza-se a escrita de *pouco* código que corresponda a soluções simples e elegantes.

Problema 1

Por simplicidade, consideremos a função

$$f' = f\ a\ b\ c$$

como uma abreviatura da função original que se pretende otimizar. Consideremos, de igual modo, as seguintes funções

$$g\ n = f'\ (n + 1)$$

$$h\ n = g\ (n + 1)$$

Destas definições imediatamente segue que

$$g\ 0 = f'\ 1 = 1$$

$$g\ (n + 1) = h\ n$$

$$h\ 0 = g\ 1 = f'\ 2 = 1$$

$$h\ (n + 1) = g\ (n + 2) = f'\ (n + 3) = a * f'\ (n + 2) + b * f'\ (n + 1) + c * f'\ n = a * h\ n + b * g\ n + c * f'\ n$$

.

Estas três funções são mutuamente recursivas. Deste modo, tem-se que

$$\begin{aligned}
& \left\{ \begin{array}{l} g\ 0 = 1 \\ g\ (n+1) = h\ n \\ h\ 0 = 1 \\ h\ (n+1) = a * h\ n + b * g\ n + c * f'\ n \\ f'\ 0 = 0 \\ f'\ (n+1) = g\ n \end{array} \right\} \\
& \equiv \{ \text{ k a b c } ((x, y), z) = a * y + b * x + c * z \} \\
& \left\{ \begin{array}{l} g\ 0 = 1 \\ g\ (n+1) = \pi_2 \cdot \pi_1 ((g\ n, h\ n), f'\ n) \\ h\ 0 = 1 \\ h\ (n+1) = \text{ k a b c } ((g\ n, h\ n), f'\ n) \\ f'\ 0 = 0 \\ f'\ (n+1) = \pi_1 \cdot \pi_1 ((g\ n, h\ n), f'\ n) \end{array} \right\} \\
& \equiv \{ \text{ lei da recursividade mútua } \} \\
& \langle \langle g, h \rangle, f' \rangle = \langle \langle [1, \pi_2 \cdot \pi_1], [1, \text{ k a b c }], [0, \pi_1 \cdot \pi_1] \rangle \rangle \\
& \equiv \{ \text{ lei da troca } \} \\
& \langle \langle g, h \rangle, f' \rangle = \langle \langle [(1, 1), \langle \pi_2 \cdot \pi_1, \text{ k a b c } \rangle], [0, \pi_1 \cdot \pi_1] \rangle \rangle \\
& \equiv \{ \text{ lei da troca } \} \\
& \langle \langle g, h \rangle, f' \rangle = \langle \langle [(1, 1), 0], \langle \langle \pi_2 \cdot \pi_1, \text{ k a b c } \rangle, \pi_1 \cdot \pi_1 \rangle \rangle \rangle \\
& \equiv \{ \text{ definição de for } \} \\
& \langle \langle g, h \rangle, f' \rangle = \text{ for } \langle \langle \pi_2 \cdot \pi_1, \text{ k a b c } \rangle, \pi_1 \cdot \pi_1 \rangle \underline{\langle (1, 1), 0 \rangle} \\
& \square
\end{aligned}$$

Logo, conclui-se que

$$f\ a\ b\ c = \pi_2 \cdot \text{ for } \langle \langle \pi_2 \cdot \pi_1, \text{ k a b c } \rangle, \pi_1 \cdot \pi_1 \rangle \underline{\langle (1, 1), 0 \rangle}$$

Funções auxiliares pedidas:

$$\begin{aligned}
& \text{ k a b c } ((x, y), z) = a * y + b * x + c * z \\
& \text{ loop} = \langle \langle \pi_2 \cdot \pi_1, \text{ k a b c } \rangle, \pi_1 \cdot \pi_1 \rangle \\
& \text{ initial} = \underline{\langle (1, 1), 0 \rangle} \\
& \text{ wrap} = \pi_2
\end{aligned}$$

Comparação de desempenho

Dadas estas duas implementações da mesma função, comparou-se o seu desempenho. Para isso, criou-se um *Jupyter Notebook*, submetido juntamente com este relatório, que tratasse dessa mesma análise. Para medir o tempo de execução de cada função utilizou-se a biblioteca [Criterion](#); e para gerar os gráficos apresentados nas figuras 6 e 7 usou-se a biblioteca [Chart](#).

Os tempos calculados correspondem à média de 10 execuções de cada função para o respetivo argumento. Para os testes, usou-se $a = 2, b = 3, c = 4$.

Como se pode ver pelos gráficos, a implementação ineficiente de f apresenta um crescimento exponencial do tempo de execução, comparativamente à implementação eficiente, que tira proveito da recursividade mútua, que apresenta um crescimento linear.



Figura 6: Desempenho da implementação ineficiente de f

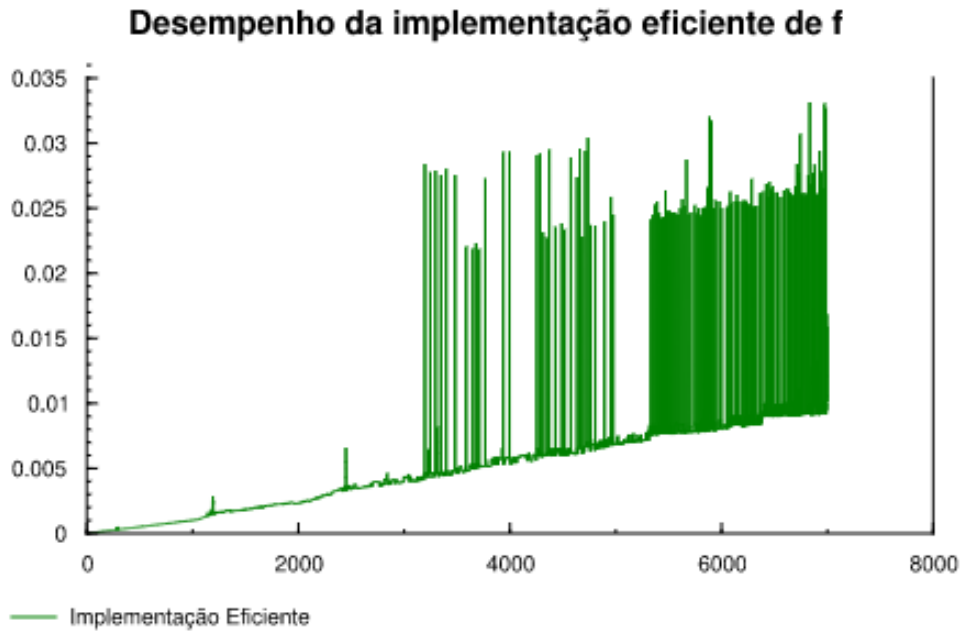


Figura 7: Desempenho da implementação eficiente de f

Esta melhoria de desempenho é notável, visto que torna uma função que não seria viável utilizar para valores relativamente pequenos do seu argumento, numa função capaz de processar valores muito superiores em intervalos de tempo bastante reduzidos.

Problema 2

Gene de *tax*:

$$gene = ((id) + (id \times ((groupBy \text{canTrim}) \cdot (\lambda verb \mid map \mid trim)))) \cdot out$$

Função de pós-processamento:

$$post = \llbracket [singl \cdot singl, (\lambda verb \mid map \mid) \cdot ((:) \times concat)] \rrbracket_{Exp}$$

gene

Desenhando o diagrama do anamorfismo:

$$\begin{array}{ccc} Exp \ String \ String & \xleftarrow{in_{Exp}} & String + String \times (Exp \ String \ String)^* \\ \uparrow \llbracket gene \rrbracket_{Exp} & & id + (id \times (map \llbracket gene \rrbracket_{Exp})) \uparrow \\ String^* & \xrightarrow{gene} & String + String \times String^{**} \end{array}$$

A função `gene` vai ser espessa em função do seu caso de paragem e caso geral. De notar que, se aplicarmos o funtor das listas não vazias (`out`) ao argumento da função, podemos definir o `gene` como uma soma de funções.

O lado esquerdo da soma - correspondente ao caso de paragem - será a identidade. Isto porque, caso a lista seja singular, pretende devolver-se esse elemento, que será uma folha na árvore de expressão.

O lado direito da soma é, como já tem sido hábito, bastante mais complexo. Como se deve tratar de uma função que recebe e devolve pares, vamos exprimi-la como um produto de outras duas funções. O fator do lado esquerdo deve ser, mais uma vez, a identidade, visto que se pretende preservar o elemento à cabeça da lista no nodo atual da árvore. O fator do lado direito deve ser uma função que, dada a cauda da lista, remova 4 espaços a todos os elementos (visto que estes elementos serão filhos na árvore, todos os elementos da lista serão *strings* que começam com, pelo menos, 4 espaços), e parta a lista resultante por subárvores a explorar recursivamente. Como se faz esta divisão? Simplesmente parte-se a lista sempre que há um elemento que não está indentado. Porquê nesses elementos? Porque esses elementos constituem as raízes das subárvores e, por isso, devem ser a cabeça das listas que serão recursivamente convertidas em árvores.

Deste modo, começa-se por um `map trim` à lista inicial. A função `trim` remove os primeiro quatro espaços de uma *string*. De seguida, essa função é composta com um `groupBy (const canTrim)`¹⁴. A função `groupBy` está definida no módulo de Haskell `Data.List` e parte uma lista sempre que a função argumento seja verdadeiro, colocando o elemento para o qual isso aconteceu à cabeça de uma nova lista.

Assim sendo, o `gene` do anamorfismo é definido por

$$gene = ((id) + (id \times ((groupBy \ canTrim) \cdot (\lambda verb \mid map \mid trim)))) \cdot out$$

com as seguintes funções auxiliares

$$\begin{aligned} trim (' ' : ' ' : ' ' : ' ' : r) &= r \\ canTrim (' ' : ' ' : ' ' : ' ' : _) &= True \\ canTrim _ &= False \end{aligned}$$

post

A função `post` recebe uma árvore de expressão e deve retornar uma lista de listas de *String*, que corresponde à tabela pretendida. Mais concretamente, deve, para cada elemento da árvore, criar uma lista correspondente à travessia na árvore desde a sua raiz até esse elemento.

Esta operação pode ser implementada como um catamorfismo sobre árvores de expressões. Fazendo o diagrama correspondente:

¹⁴Dada a assinatura da função `groupBy`, houve a necessidade de adicionar um primeiro argumento à função `canTrim`, que deve ser ignorado pela mesma, daí a utilização da função `const`

$$\begin{array}{ccc}
Exp \ String \ String & \xrightarrow{\text{out}_{Exp}} & String + String \times (Exp \ String \ String)^* \\
\downarrow \llbracket g \rrbracket_{Exp} & & \downarrow id + (id \times (\text{map } \llbracket g \rrbracket_{Exp})) \\
String^{**} & \xleftarrow{g} & String + String \times String^{***}
\end{array}$$

Com isto, basta apenas definir o gene do catamorfismo. Tendo em conta o facto da função receber uma soma e devolver uma lista, esta será definida como um `either`.

O caso de paragem (lado esquerdo do `either`) corresponde a uma folha da árvore de expressão. Nesse caso deve retornar-se uma lista singular, contendo outra lista singular contendo apenas esse valor. Isto faz-se com duas chamadas consecutivas de `singl`, ou seja, `singl . singl`.

No caso recursivo, recebe-se o elemento atual e o resultado das chamadas recursivas para os filhos. Aí, concatenam-se todos os resultados recursivos na mesma lista (para manter a noção de tabela) e, de seguida, adiciona-se o elemento à cabeça de cada uma dessas listas. Em notação *pointwise*:

$$\lambda verb \mid \text{map } \mid (s:) \$ \text{concat } l$$

Passando para *pointfree*:

$$(\widehat{\lambda verb \mid \text{map } \mid}) \cdot ((:) \times \text{concat})$$

Juntando tudo, temos a definição do gene e da função `post`:

$$\begin{aligned}
post &= \llbracket g \rrbracket_{Exp} \\
g &= [singl \cdot singl, (\widehat{\lambda verb \mid \text{map } \mid}) \cdot ((:) \times \text{concat})]
\end{aligned}$$

Problema 3

$$\begin{aligned}
squares &= \llbracket gsq \rrbracket_R \\
gsq &= \langle middleSquare \cdot \pi_1, [\widehat{nil}, \widehat{zip} \cdot (sideSquares \times repeat)] \cdot \text{distr} \rangle \cdot (id \times \text{outNat}) \\
rose2List &= \llbracket gr2l \rrbracket_R \\
gr2l &= \text{concat} \cdot \text{cons} \cdot (singl \times id) \\
carpets &= \text{map } (sierpinski \cdot \langle \text{defaultSquare}, id \rangle) [0..n] \\
present &= \text{sequence} \cdot (\text{map } ((\gg \text{await}) \cdot \text{drawSq}))
\end{aligned}$$

Funções auxiliares:

$$\begin{aligned}
middleSquare &= \langle \widehat{addTup} \cdot (id \times \text{dup}), \pi_2 \rangle \cdot (id \times (/3)) \\
sideSquares \ (xy, l) &= \text{map } \langle (\widehat{addTup} \ xy) \cdot (\text{scaleTup } l'), l' \rangle [(1,2), (2,2), (2,1), (2,0), (1,0), (0,0), (0,1), (0,2)] \\
\text{where } l' &= l / 3 \\
\text{addTup } (a,b) \ (c,d) &= (a+c, b+d) \\
\text{scaleTup } a \ (b,c) &= (a*b, a*c)
\end{aligned}$$

gsq

Façamos o esquema representativo dos anamorfismos sobre Rose Trees.

$$\begin{array}{ccc}
Rose \ Square & \xleftarrow{\text{in}} & Square \times Rose \ Square^* \\
\uparrow \llbracket gsq \rrbracket_R & & \uparrow id \times (\text{map } (\llbracket g \rrbracket_R)) \\
Square \times Int & \xrightarrow{gsq} & Square \times (Square \times Int)^*
\end{array}$$

Com isto, determina-se o tipo de `gsq`. O primeiro elemento do par corresponde ao quadrado a partir em subquadrados mais pequenos, e o segundo elemento é o número de partições restantes. Se este valor for 0, o quadrado deve ser preservado.

Isto motiva a utilização do funtor dos naturais. Isso justifica o primeiro elemento da composição de funções que é a função `gsq`:

$$(id \times outNat)$$

Com isto, preserva-se o quadrado, mas diminui-se o natural numa unidade, possibilitando também a utilização de um `either` para separar o caso de paragem dos restantes casos.

Dado facto de a função receber um produto, está será expressa como um `split`. Expliquemos primeiro o lado direito do mesmo, responsável pela criação dos quadrados mais pequenos, que serão posteriormente processados recursivamente.

Foquemo-nos, inicialmente, no caso de paragem (lado esquerdo do `either`¹⁵). Este caso é bastante simples: deve ser retornada uma lista vazia, logo é utilizada a função `nil`.

O segundo caso é bastante mais complicado, e, também por isso, mais interessante. Por um lado, quer-se calcular todos os subquadrados gerados a partir do quadrado atual: para isso usa-se a função auxiliar `sideSquares`. Por outro lado, pretende-se juntar (`zip`) essa lista com uma lista contendo o valor inteiro fornecido. Para isso, podemos aproveitar a avaliação *lazy* de Haskell e a função `repeat`.

Sumariando, pretende-se pegar no par quadrado / número, calcular todos os subquadrados, e, em paralelo repetir o valor inteiro, juntando no fim essas listas. Isso é expresso pela expressão

$$\widehat{zip} \cdot (sideSquares \times repeat)$$

Passemos, agora, para o lado esquerdo do `split`, que devolve o quadrado que não será mais dividido. Este é bastante simples, basta extrair o primeiro elemento do par (o quadrado), e aplicar-lhe a função auxiliar `middleSquare`, que calcular o quadrado que deve ficar *no meio*. Assim sendo, o lado esquerdo corresponde a

$$(middleSquare \cdot \pi_1)$$

gr2l

Consideremos o diagrama do catamorfismo `rose2List`:

$$\begin{array}{ccc} Rose\ A & \xrightarrow{\quad out \quad} & A \times Rose\ A^* \\ \downarrow \langle \llbracket gr2l \rrbracket \rangle_R & & \downarrow id + \langle \llbracket gr2l \rrbracket \rangle_R \\ A^* & \xleftarrow{\quad gr2l \quad} & A \times A^{**} \end{array}$$

Derivamos, deste modo, o tipo da função `gr2l` pretendida. O que se pretende é que esta função concatene toda as listas que recebe como argumento no segundo elemento do par, e que adicione o elemento que recebe como primeiro elemento do par à cabeça da lista.

A solução concebida foi a seguinte: colocar o primeiro elemento como uma lista e juntá-lo à cabeça da lista de listas recebida como segundo argumento, e, de seguida, concatenar as listas interiores para formar uma só lista.

Este último passo é efetuado trivialmente pela função `concat`. O passo anterior recorre à função `cons`, que pega num par do tipo $(a, [a])$ e devolve uma lista em que o primeiro elemento foi acrescentado à lista. Para, neste caso, conseguir aplicar essa função, é necessário que o primeiro elemento do par seja, também ele, uma lista. Para isso usa-se a função `singl` multiplicada com a identidade.

¹⁵Foi aplicada a função `distr` antes do código aqui explicado, de forma a passar o `either` para fora do par, para facilitar a resolução. Isto apenas foi feito no lado direito do `split`, pois não era conveniente para o lado esquerdo, como será explicado de seguida

Assim sendo, a função `gr2l` é a composição de todas estas funções:

$$gr2l = concat \cdot cons \cdot (singl \times id)$$

carpets

Esta função deve receber um inteiro e devolver uma lista contendo os diferentes passos de aplicação do algoritmo. Ou seja, o primeiro elemento corresponde à lista de quadrados que se obtém rodando o algoritmo uma vez, o segundo elemento duas vezes, etc.

Para isso, pode-se aproveitar a função `sierpinski`, que recebe um par quadrado/inteiro, que corresponde ao quadrado inicial e ao número de iterações do algoritmo, respetivamente; e devolve os quadrados gerados.

Pretende aplicar-se, então, a função `sierpinski` com todos os limites de profundidade de 0 a n (em que n é o parâmetro de `carpets`). Isso motiva a aplicar um `map` da função `sierpinski` à lista `[0..n]`:

$$carpets\ n = \text{map } (sierpinski \cdot ???) [0..n]$$

No entanto, não é possível aplicar a função `sierpinski` a um inteiro. De facto, a função, como referido anteriormente, recebe um par. Esse inteiro deve ser passado à função como segundo elemento do par, sendo que o primeiro elemento é independente desse valor, sendo sempre o mesmo (expresso pela função `defaultSquare`). Isto motiva a utilização de um `split`, em que do lado direito se aplica a função identidade, e do lado esquerdo a constante `defaultSquare`.

Assim chega-se à definição final de `carpets`:

$$carpets\ n = \text{map } (sierpinski \cdot \langle defaultSquare, id \rangle) [0..n]$$

present

Finalmente, esta função deve receber uma lista como aquela devolvida pela função `carpets` e representá-la visualmente, devolvendo, assim, um `IO [()]`. Entre cada iteração, deve aguardar um valor fixo de tempo - algo que a função `await` já faz.

Esta função pode ser definida como um catamorfismo sobre listas. No entanto, o grupo desenvolveu uma solução mais simples recorrendo a um `map`. O raciocínio é bastante semelhante: pretende-se desenhar os quadrados e esperar; para cada elemento da lista recebida. Para isso, pode-se executar um `map` de uma função que faça o pretendido para cada elemento da lista. Mas como definir tal função?

O primeiro passo será sempre desenhar: por isso a função `drawSq` será sempre a primeira a ser chamada. De seguida é preciso esperar, recorrendo-se à função `>> await`. A combinação de `await` com a função `>>` de Haskell pode ser explicada pelo facto da função `drawSq` retornar um tipo monádico que deve ser ignorado por `await`. Compondo tudo:

No entanto, o tipo de retorno desta solução é `[IO ()]` e não `IO [()]`. Para resolver este problema, basta adicionar uma chamada à função `sequence`, que transforma uma lista de monades num monade de listas. Combinando tudo:

$$present = sequence \cdot (\text{map } ((\gg await) \cdot drawSq))$$

Problema 4

Versão não probabilística

Gene de `consolidate'`:

$$c_{gene} = \perp$$

Geração dos jogos da fase de grupos:

$$\begin{aligned}
\text{pairup} &= \text{concat} \cdot (\widehat{(\text{zipWith zip})}) \cdot \langle \text{repeat}, \text{tail} \cdot \text{suffixes} \rangle \\
\text{matchResult } f &= \widehat{\text{matchResults}} \cdot \langle \text{id}, f \rangle \\
\text{glt} &= (\text{id} + (\text{splitInHalf} \cdot \widehat{(\cdot)})) \cdot \text{out} \\
\text{matchResults} &= \overline{\text{cons} \cdot (\text{tr} \times \text{singl} \cdot \text{tr}) \cdot \langle \pi_1 \times \text{id}, \pi_2 \times \text{id} \rangle} \\
\text{tr} &= \widehat{\text{teamResult}} \\
\text{teamResult } t &= \text{maybe } (t, 1) \text{ (cond } (\equiv t) \text{ } \underline{t}, 3 \text{ } \underline{t}, 0)
\end{aligned}$$

Começamos pela função `pairup`, que deve, dado um grupo, devolver uma lista com todos os jogos desse grupo, sobre a forma de pares de equipas (*strings*).

A função `matchResult` deve, dada uma função que devolve o resultado de um jogo e um jogo, devolver uma lista com a pontuação de cada uma das equipas nesse jogo. Recordemos que, no futebol, uma equipa ganha 3 pontos se ganhar, 1 se empatar, e 0 se perder.

Inicialmente, partiu-se o problema em duas partes mais pequenas. A primeira consiste em calcular, dados um jogo e o seu resultado, as pontuações de cada equipa; o que deu origem à função `matchResults`. A segunda parte consiste em calcular o resultado do jogo, e preparar essa informação para ser consumida pela função anterior.

A função `matchResults` é do tipo `Match -> Maybe Team -> [(Team, Int)]`. Em primeiro lugar, esta função vai ser *uncurried* para receber um par. Em segundo lugar, para obter os dados para esta função, basta aplicar um `split` (visto tratar-se de uma função que recebe um par), em que uma das funções é a identidade (porque pretendemos passar o jogo que recebemos, sem modificações), e a outra é a função que calcula o resultado do jogo, que foi recebida como argumento (`f`). Assim sendo, a função `matchResult` é definida por

$$\text{matchResult } f = \widehat{\text{matchResults}} \cdot \langle \text{id}, f \rangle$$

Passemos, agora, à primeira parte do problema: a função `matchResults`.

A função `glt` é o gene de um anamorfismo sobre *Leaf Trees*, o qual deve, a partir da lista das equipas que passaram à fase eliminatória em cada grupo, construir a árvore que representa o *mata-mata*. O anamorfismo é representado no seguinte diagrama.

$$\begin{array}{ccc}
\text{LTree Team} & \xleftarrow{\text{in LTree}} & \text{Team} + (\text{LTree Team} \times \text{LTree Team}) \\
\uparrow \llbracket \text{glt} \rrbracket & & \uparrow \text{id} + (\text{Leaf} \times \llbracket \text{glt} \rrbracket) \\
\text{Team}^* & \xrightarrow{\text{gr2l}} & \text{Team} + (\text{Team} \times \text{Team}^*)
\end{array}$$

Deriva-se, assim o tipo em Haskell da função `glt`: `[Team] -> Either Team (Team, [Team])`. Como o resultado é uma soma, esta função poderá ser expressa, provavelmente, como uma soma de funções. No entanto, o tipo de entrada não é uma soma, é uma lista. Mas, se aplicarmos o funtor das listas ao argumento de entrada já ficamos com uma soma: `Team + Team \times Team`.

Após aplicar o funtor, é necessário definir ambos os casos da soma. Caso a lista seja singular, deve devolver-se o próprio elemento, isto é, a função do lado esquerdo da soma deve ser a identidade. Caso a lista tenha mais elementos, deve-se dividi-la exatamente ao meio. Para isso, primeiro é preciso reconstruir a lista (utilizando o operador `:` de Haskell) e usar uma função, definida pelo grupo, para a dividir a meio, chamada `splitInHalf`.¹⁶

Assim sendo, o lado direito da soma é

$$(\text{splitInHalf} \cdot \widehat{(\cdot)})$$

e o gene `glt`

¹⁶Há já definida uma função no módulo `LTree`, chamada `lsplit`, que também parte a lista a meio. No entanto, esta não a parte exatamente a meio. Em vez disso, intercala os elementos entre as duas listas resultantes. Por isso, não foi usada.

$$glt = (id + (splitInHalf \cdot \widehat{(\cdot)})) \cdot out$$

Versão probabilística

```

pinitKnockoutStage = return · initKnockoutStage
pgroupWinners :: (Match → Dist (Maybe Team)) → [Match] → Dist [Team]
pgroupWinners criteria = (>>= return · best 2 · consolidate · concat) · multiProd · map (pmatchResult criteria)
pmatchResult criteria = ( $\widehat{(>>=)}$ ) · (criteria, return 'multiComp' matchResults)
multiComp = (·) · (·)

```

A definição da função `pinitKnockoutStage` é trivial quando feita à custa da função `initKnockoutStage`. De facto, a `pinitKnockoutStage` não depende em nada de probabilidades; a única diferença em relação à sua variante não monádica é o facto de ter de retornar um valor no monóide das probabilidades. Assim sendo, basta utilizar a função `return` do monóide após aplicar a `initKnockoutStage` original; ficando assim definida como

```

pinitKnockoutStage = return · initKnockoutStage

```

Começemos por derivar a definição da função `pgroupWinners`. O problema foi dividido em duas partes. A primeira consiste em obter a distribuição de probabilidade de cada conjunto de resultados dos jogos do grupo. Para fazer isso, aplica-se a função `pmatchResult criteria` a todos os elementos da lista de jogos. Ou seja, a primeira parte do problema consiste na função `map (pmatchResult criteria)`. Para converter os resultados de uma lista de monóides para um monóide de lista, usa-se a função `multiProd`. Assim sendo, esta primeira parte é, na verdade, resolvida através da função `multiProd . map (pmatchResult criteria)`.

A segunda parte consiste em, sabendo os resultados dos jogos, determinar os vencedores do grupo. Para isso, é necessário juntar as listas correspondentes a cada jogo numa só - usando, para isso, a função `concat` - e consolidar (`consolidate`) a lista, de forma a ficar com a pontuação de cada equipa. Finalmente, é necessário retornar os 2 melhores, utilizando, para isso, a função `best 2`. Como estamos a trabalhar num contexto monádico, é necessário utilizar a função `return` do monóide das distribuições de probabilidade para colocar o resultado dentro desse monóide.

De forma a combinar estas duas partes, utiliza-se o operador `>>=` de Haskell, para passar o resultado da primeira para a segunda. Assim sendo, o resultado é

```

pgroupWinners criteria = (>>= return · best 2 · consolidate · concat) · multiProd · map (pmatchResult criteria)

```

Finalmente, consideremos a função `pmatchResult`, que deve, para um dado encontro e uma função de probabilidade, retornar a probabilidade de cada resultado - sob a forma de pontuação de cada equipa. O primeiro passo é, naturalmente, aplicar a função de probabilidade ao encontro em questão. De seguida, deve converter-se em resultado em pontuações para cada equipa - que é o que faz a função `matchResult`. Assim sendo, esta função pode ser escrita, *pointwise*, da seguinte forma:

```

pmatchResult criteria m = criteria m >>= (return · matchResults m)

```

Convertendo esta função para uma definição *pointfree* temos

```

pmatchResult criteria = ( $\widehat{(>>=)}$ ) · (criteria, return 'multiComp' matchResults)
multiComp = (·) · (·)

```

Índice

- LaTeX, 10
 - bibtex, 10
 - lhs2TeX, 10
 - makeindex, 10
- Cálculo de Programas, 1, 3, 10, 11
 - Material Pedagógico, 9
 - Exp.hs, 2, 3, 13
 - LTree.hs, 6–8
 - Rose.hs, 4
- Combinador “pointfree”
 - either*, 7, 9
- Fractal, 3
 - Tapete de Sierpinski, 3
- Função
 - π_1 , 10, 11, 15
 - π_2 , 10, 15
 - for*, 2, 11
 - length*, 13
 - map*, 7, 8, 13–15
- Functor, 5, 8, 9, 11, 12, 15, 16
- Haskell, 1, 10
 - Biblioteca
 - PFP, 12
 - Probability, 12
 - interpretador
 - GHCi, 10, 12
 - Literate Haskell, 9
- Números naturais (\mathbb{N}), 11
- Programação
 - dinâmica, 11
 - literária, 9
- SVG (Scalable Vector Graphics), 13
- U.Minho
 - Departamento de Informática, 1, 2