# Cálculo de Programas
Trabalho Prático
LEI — 2022/23

*Departamento de Informática*
Universidade do Minho

Janeiro de 2023



| Grupo nr. | 05 |
|---|---|
| a96215 | João Martins |
| a97541 | Gonçalo Braga |
| a95019 | João Gonçalves |



## 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 abodarem os problemas propostos no trabalho, os grupos devem ler com atenção o anexo **??** onde encontrarão as instruções relativas ao sofware 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 $c$:

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

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

$$1, 1, 2, 4, 7, 13, 24, 44, 81, 149, \ldots$$

$f\ 1\ 2\ 3$ irá gerar a sequência:

$$1, 1, 3, 8, 17, 42, 100, 235, 561, 1331, \ldots$$

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

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

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

**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 ítens) 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 padagógico 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:

   $$tax :: [String] \to Exp\ String\ String$$
   $$tax = [\![ gene ]\!]_{Exp}$$

   Ou seja, defina o $gene$ do anamorfismo, tendo em conta o seguinte diagrama[2]:

   $$\begin{tikzcd}
Exp\ S\ S & & S + S \times (Exp\ S\ S)^* \arrow[ll, "\mathsf{in}_{Exp}"'] \\
S^* \arrow[u, "tax"] \arrow[r, "out"] \arrow[rr, bend right, "gene"'] & S + S \times S^* \arrow[r, "\cdots"] & S + S \times (S^*)^* \arrow[u, "id + id \times tax^*"']
\end{tikzcd}$$

   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 **??** 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]

[1] Informação obtida a partir do site ACM CCS selecionando *Flat View*.

[2] $S$ abrevia $String$.

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

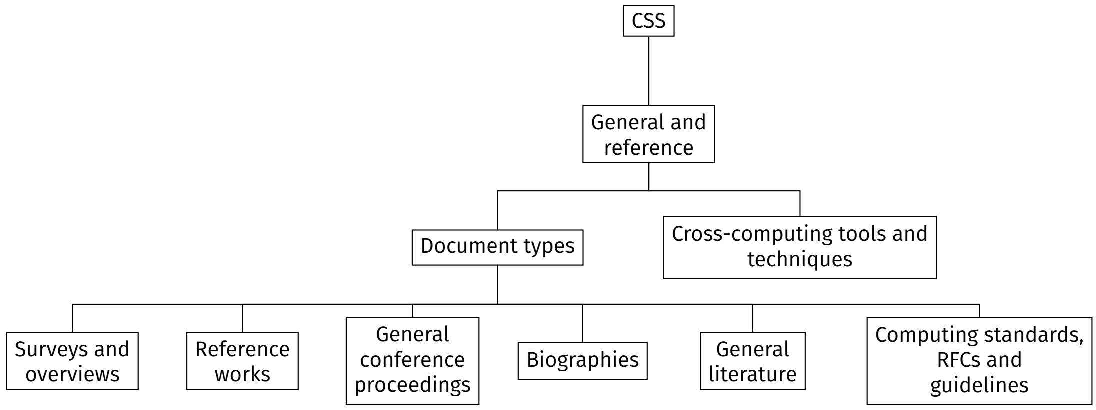



Figura 1: Fragmento de $acm\_ccs$ representado sob a forma de uma árvore do tipo Exp.

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

$$tudo :: [String] \rightarrow [[String]]$$
$$tudo = post \cdot tax$$

para obter o efeito que se mostra na tabela **??**.

| | | | |
|---|---|---|---|
| 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 ítens).

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

**Sugestão**: Inspecione 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 subdivido 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. **??**).

**NB**: No exemplo da fig. **??**, 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.

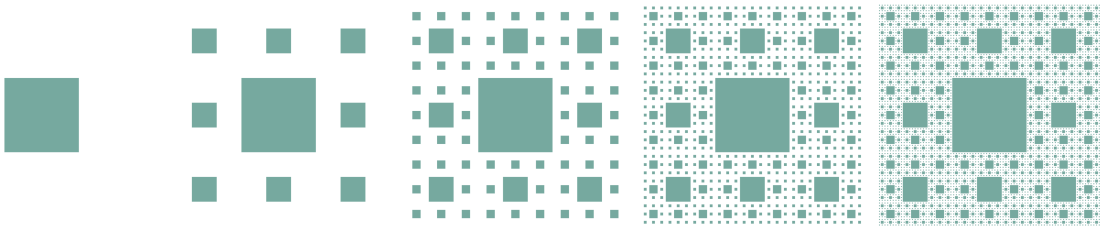


Figura 2: Construção do tapete de Sierpinski com profundidade 5.

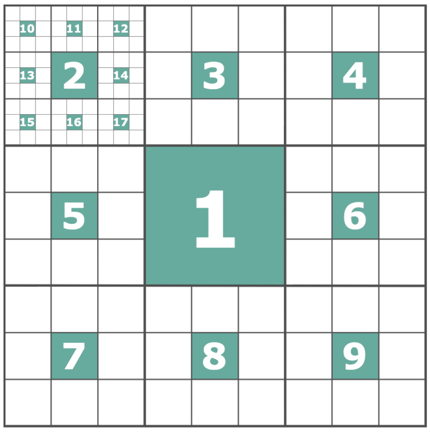



Figura 3: Tapete de Sierpinski com profundidade 2 e com os quadrados enumerados.

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

$$\textbf{type}\ \mathit{Square} = (\mathit{Point}, \mathit{Side})$$
$$\textbf{type}\ \mathit{Side} = \mathit{Double}$$
$$\textbf{type}\ \mathit{Point} = (\mathit{Double}, \mathit{Double})$$

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. **??** poderá[4] corresponder à árvore da figura **??**.

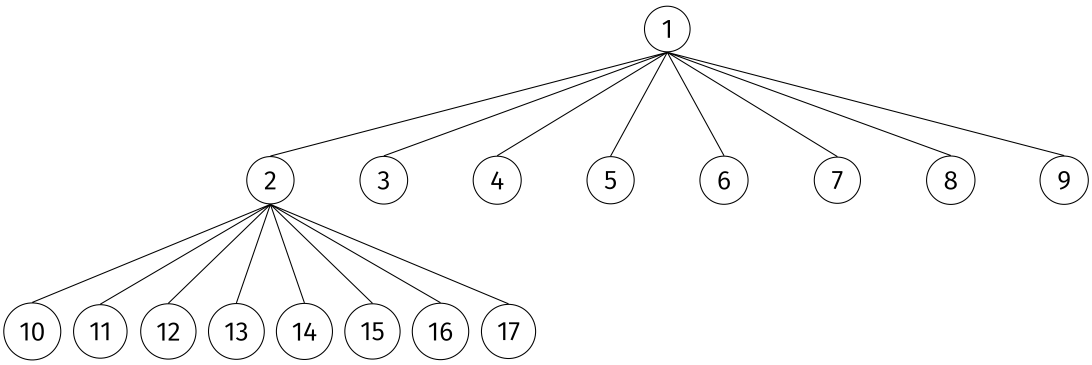



Figura 4: Possível árvore de suporte para a construção da fig. **??**.

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.

[4] A ordem dos filhos não é relevante.

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

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

**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 **??**.

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

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

$$sierpinski :: (Square, Int) \to [Square]$$
$$sierpinski = [\![ gr2l, gsq ]\!]_R$$

**Trabalho a fazer:**

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

$$sierp4 = drawSq\ (sierpinski\ (((0,0),32),3))$$
$$\begin{aligned} constructSierp5 = \mathbf{do}\ & drawSq\ (sierpinski\ (((0,0),32),0)) \\ & await \\ & drawSq\ (sierpinski\ (((0,0),32),1)) \\ & await \\ & drawSq\ (sierpinski\ (((0,0),32),2)) \\ & await \\ & drawSq\ (sierpinski\ (((0,0),32),3)) \\ & await \\ & drawSq\ (sierpinski\ (((0,0),32),4)) \\ & await \end{aligned}$$

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.

$$constructSierp :: Int \to \mathsf{IO}\ [()]$$
$$constructSierp = present \cdot carpets$$

**Dica**: a função $constructSierp$ será um hilomorfismo de listas, cujo anamorfismo $carpets :: Int \to [[Square]]$ constrói, recebendo como parâmetro a profundidade $n$, a lista com todos os tapetes de profundidade $1..n$, e o catamorfismo $present :: [[Square]] \to \mathsf{IO}\ [()]$ 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.

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

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

1. versão académica 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 **??**.

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 **??**.

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 **??**, 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

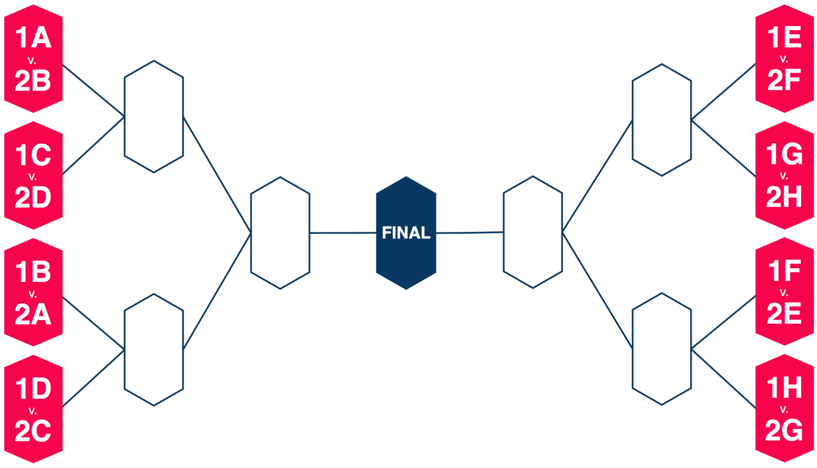



Figura 5: O “mata-mata”

garantisse 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:

$$
\begin{aligned}
&\mathit{winner} :: \mathit{Team}\\
&\mathit{winner} = \mathit{wcup}\ \mathit{groups}\\
&\mathit{wcup} = \mathit{knockoutStage} \cdot \mathit{groupStage}
\end{aligned}
$$

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

$$\mathit{knockoutStage} = (\![\mathit{id}, \mathit{koCriteria}]\!)$$

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. **??**).*

*Assim:*

$$
\begin{aligned}
&\mathit{groupStage} :: [\mathit{Group}] \to \mathsf{LTree}\ \mathit{Team}\\
&\mathit{groupStage} = \mathit{initKnockoutStage} \cdot \mathit{simulateGroupStage} \cdot \mathit{genGroupStageMatches}
\end{aligned}
$$

*Comecemos então por definir a função* $\mathit{genGroupStageMatches}$ *que gera os jogos da fase de grupos:*

$$
\begin{aligned}
&\mathit{genGroupStageMatches} :: [\mathit{Group}] \to [[\mathit{Match}]]\\
&\mathit{genGroupStageMatches} = \mathsf{map}\ \ \mathit{generateMatches}
\end{aligned}
$$

*onde*

$$\textbf{type}\ \mathit{Match} = (\mathit{Team}, \mathit{Team})$$

*Ora, sabemos que nos foi dada a função*

$$\mathit{gsCriteria} :: \mathit{Match} \to \mathit{Maybe}\ \mathit{Team}$$

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

> $simulateGroupStage :: [[Match]] \to [[Team]]$
> $simulateGroupStage = \mathsf{map}\ (groupWinners\ gsCriteria)$
>
> *que simula a fase de grupos e dá como resultado a lista dos vencedores, recorrendo à função* $groupWinners$:
>
> $groupWinners\ criteria = best\ 2 \cdot consolidate \cdot (\ggg= matchResult\ criteria)$
>
> *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:*
>
> $initKnockoutStage = [\![ glt ]\!] \cdot arrangement$

Trabalho a fazer:

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

   $consolidate' :: (Eq\ a, Num\ b) \Rightarrow [(a,b)] \to [(a,b)]$
   $consolidate' = (\![ cgene ]\!)$

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

### Versão probabilística

Nesta versão, mais realista, $gsCriteria :: Match \to (Maybe\ Team)$ dá lugar a

$pgsCriteria :: Match \to \mathsf{Dist}\ (Maybe\ Team)$

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 **??**, página **??** 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 relembraram 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:

$pwinner :: \mathsf{Dist}\ Team$
$pwinner = pwcup\ groups$

$$pwcup = pknockoutStage \bullet pgroupStage$$

E entregou ainda a versão probabilística do "mata-mata":

$$pknockoutStage = mcataLTree' \; [return, pkoCriteria]$$

$$\begin{aligned} & mcataLTree' \; g = k \; \textbf{where} \\ & \quad k \; (Leaf \; a) = g1 \; a \\ & \quad k \; (Fork \; (x,y)) = mmbin \; g2 \; (k \; x, k \; y) \\ & \quad g1 = g \cdot i_1 \\ & \quad g2 = g \cdot i_2 \end{aligned}$$

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 **??**, 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-processador que faz "pretty printing" de código Haskell em LaTeX e que deve desde já instalar utilizando o utiliário 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.

[6] O 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 **??** 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 **??**, 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]

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

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

$$
\begin{tikzcd}
\mathbb{N}_0 \arrow[d, "(\!| g |\!)"'] & 1 + \mathbb{N}_0 \arrow[l, "\mathsf{in}"'] \arrow[d, "id + (\!| g |\!)"] \\
B & 1 + B \arrow[l, "g"]
\end{tikzcd}
$$

# 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 $\mathsf{F}\ X = 1 + X$) 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:

$$
\begin{array}{l}
fib\ 0 = 1 \\
fib\ (n+1) = f\ n
\end{array}
$$

[7] Exemplos tirados de [**?**].
[8] Lei (3.95) em [**?**], página 112.

$$f\ 0 = 1$$
$$f\ (n+1) = fib\ n + f\ n$$

Obter-se-á de imediato

$$\begin{array}{l} fib' = \pi_1 \cdot \mathsf{for}\ loop\ init\ \mathbf{where} \\ \quad loop\ (fib, f) = (f, fib + f) \\ \quad init = (1, 1) \end{array}$$

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* coleccionam-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 aulas[10], de $f\ x = ax^2 + bx + c$ derivam-se duas funções mutuamente recursivas:

$$\begin{array}{l} f\ 0 = c \\ f\ (n+1) = f\ n + k\ n \\ k\ 0 = a + b \\ k\ (n+1) = k\ n + 2\ a \end{array}$$

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

$$\begin{array}{l} f'\ a\ b\ c = \pi_1 \cdot \mathsf{for}\ loop\ init\ \mathbf{where} \\ \quad loop\ (f, k) = (f + k, k + 2 * a) \\ \quad init = (c, a + b) \end{array}$$

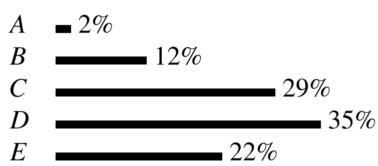
## C O mónade das distribuições probabilísticas

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

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

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

Cada par $(a, p)$ numa distribuição $d :: \mathsf{Dist}\ a$ 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

$$\begin{array}{l} d_1 :: \mathsf{Dist}\ Char \\ d_1 = D\ [(\texttt{'A'}, 0.02), (\texttt{'B'}, 0.12), (\texttt{'C'}, 0.29), (\texttt{'D'}, 0.35), (\texttt{'E'}, 0.22)] \end{array}$$

que o GHCi mostrará assim:

[9] Podem obviamente usar-se outros símbolos, mas numa primeira leitura dá jeito usarem-se tais nomes.
[10] Secção 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*,

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

isto é

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

distribuição *normais*, eg.

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

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

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

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

$$test\ a\ b\ c = \mathsf{map}\ (fbl\ a\ b\ c)\ x \equiv \mathsf{map}\ (f\ a\ b\ c)\ x\ \textbf{where}\ x = [1\mathinner{..}20]$$
$$test1 = test\ 1\ 2\ 3$$
$$test2 = test\ (-2)\ 1\ 5$$

## Problema 2

**Verificação**: a árvore de tipo Exp gerada por

$$acm\_tree = tax\ acm\_ccs$$

deverá verificar as propriedades seguintes:

- $expDepth\ acm\_tree \equiv 7$ (profundidade da árvore);
- $length\ (expOps\ acm\_tree) \equiv 432$ (número de nós da árvore);
- $length\ (expLeaves\ acm\_tree) \equiv 1682$ (número de folhas da árvore).[12]

O resultado final

$$acm\_xls = post\ acm\_tree$$

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

[11] Para 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 [**?**].

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

## Problema 3

Função para visualização em SVG:

$$drawSq\ x = picd''\ [Svg.scale\ 0.44\ (0,0)\ (x \ggg sq2svg)]$$
$$sq2svg\ (p,l) = (color\ \texttt{"\#67AB9F"} \cdot polyg)\ [p, p .+ (0,l), p .+ (l,l), p .+ (l,0)]$$

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”:

$$arrangement = (\ggg swapTeams) \cdot chunksOf\ 4\ \textbf{where}$$
$$swapTeams\ [[a_1,a_2],[b_1,b_2],[c_1,c_2],[d_1,d_2]] = [a_1,b_2,c_1,d_2,b_1,a_2,d_1,c_2]$$

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

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

Critério para a simulação não probabilística dos jogos da fase de grupos:

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

Critério para a simulação não probabilística dos jogos do mata-mata:

$$\begin{array}{l} koCriteria = s \cdot \langle id \times id, rank \times rank \rangle\ \mathbf{where} \\ \quad s\ ((s_1, s_2), (r_1, r_2)) = \mathbf{let}\ d = r_1 - r_2\ \mathbf{in} \\ \quad\quad \mathbf{if}\ d \equiv 0\ \mathbf{then}\ s_1 \\ \quad\quad\quad \mathbf{else\ if}\ d > 0\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2 \end{array}$$

Critério para a simulação probabilística dos jogos da fase de grupos:

$$\begin{array}{l} pgsCriteria = s \cdot \langle id \times id, rank \times rank \rangle\ \mathbf{where} \\ \quad s\ ((s_1, s_2), (r_1, r_2)) = \\ \quad\quad \mathbf{if}\ abs\ (r_1 - r_2) > 0.5\ \mathbf{then}\ \mathsf{fmap}\ Just\ (pkoCriteria\ (s_1, s_2))\ \mathbf{else}\ f\ (s_1, s_2) \\ \quad f = D \cdot ((Nothing, 0.5):) \cdot \mathsf{map}\ \ (Just \times (/2)) \cdot unD \cdot pkoCriteria \end{array}$$

Critério para a simulação probabilística dos jogos do mata-mata:

$$\begin{array}{l} pkoCriteria\ (e_1, e_2) = D\ [(e_1, 1 - r_2\ /\ (r_1 + r_2)), (e_2, 1 - r_1\ /\ (r_1 + r_2))]\ \mathbf{where} \\ \quad r_1 = rank\ e_1 \\ \quad r_2 = rank\ e_2 \end{array}$$

Versão probabilística da simulação da fase de grupos:[13]

$$\begin{array}{l} psimulateGroupStage = trim \cdot \mathsf{map}\ \ (pgroupWinners\ pgsCriteria) \\ trim = top\ 5 \cdot sequence \cdot \mathsf{map}\ \ (filterP \cdot norm)\ \mathbf{where} \\ \quad filterP\ (D\ x) = D\ [(a, p) \mid (a, p) \leftarrow x, p > 0.0001] \\ \quad top\ n = vec2Dist \cdot take\ n \cdot reverse \cdot presort\ \pi_2 \cdot unD \\ \quad vec2Dist\ x = D\ [(a, n\ /\ t) \mid (a, n) \leftarrow x]\ \mathbf{where}\ t = sum\ (\mathsf{map}\ \ \pi_2\ x) \end{array}$$

Versão mais eficiente da *pwinner* dada no texto principal, para diminuir o tempo de cada simulação:

$$\begin{array}{l} pwinner :: \mathsf{Dist}\ Team \\ pwinner = mbin\ f\ x \gg\!= pknockoutStage\ \mathbf{where} \\ \quad f\ (x, y) = initKnockoutStage\ (x +\!\!+ y) \\ \quad x = \langle g \cdot take\ 4, g \cdot drop\ 4 \rangle\ groups \\ \quad g = psimulateGroupStage \cdot genGroupStageMatches \end{array}$$

Auxiliares:

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

Função binária monádica $f$:

$$\begin{array}{l} mmbin :: Monad\ m \Rightarrow ((a, b) \rightarrow m\ c) \rightarrow (m\ a, m\ b) \rightarrow m\ c \\ mmbin\ f\ (a, b) = \mathbf{do}\ \{x \leftarrow a; y \leftarrow b; f\ (x, y)\} \end{array}$$

Monadificação de uma função binária $f$:

[13] Faz-se “trimming” das distribuições para reduzir o tempo de simulação.

$$
\begin{array}{l}
mbin :: Monad\ m \Rightarrow ((a,b) \rightarrow c) \rightarrow (m\ a, m\ b) \rightarrow m\ c \\
mbin = mmbin \cdot (return \cdot)
\end{array}
$$

Outras funções que podem ser úteis:

$$
\begin{array}{l}
(f\ `is`\ v)\ x = (f\ x) \equiv v \\
rcons\ (x,a) = x +\!\!+ [a]
\end{array}
$$

# 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

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

De modo a resolver o primeiro problema, definimos duas funções auxiliares:

$$
\begin{array}{l}
g\ a\ b\ c\ n = f\ a\ b\ c\ (n+1) \\
j\ a\ b\ c\ n = f\ a\ b\ c\ (n+2)
\end{array}
$$

E podemos relacionar a função $g$ com a função $j$:

$$
j\ a\ b\ c\ n = g\ a\ b\ c\ (n+1) = f\ a\ b\ c\ (n+2)
$$

Logo, possuímos a seguinte recursividade mútua:

$$
\left\{
\begin{array}{l}
\left\{
\begin{array}{l}
j\ a\ b\ c\ 0 = 1 \\
j\ a\ b\ c\ (n+1) = a * j\ a\ b\ c\ n + b * g\ a\ b\ c\ n + c * f\ a\ b\ c\ n
\end{array}
\right. \\
\left\{
\begin{array}{l}
g\ a\ b\ c\ 0 = 1 \\
g\ a\ b\ c\ (n+1) = j\ a\ b\ c\ n
\end{array}
\right. \\
\left\{
\begin{array}{l}
f\ a\ b\ c\ 0 = 0 \\
f\ a\ b\ c\ (n+1) = g\ a\ b\ c\ n
\end{array}
\right.
\end{array}
\right.
$$

Através da lei da recursividade mútua, para os naturais, sabemos que:

$$\begin{cases} \begin{cases} (j\ a\ b\ c) \cdot \underline{0} = \underline{1} \\ (j\ a\ b\ c) \cdot \mathsf{succ} = add \cdot \langle add \cdot \langle (a*) \cdot \pi_1 \cdot \pi_1, (b*) \cdot \pi_2 \cdot \pi_1 \rangle, (c*) \cdot \pi_2 \rangle \cdot \langle \langle j\ a\ b\ c, g\ a\ b\ c \rangle, f\ a\ b\ c \rangle \end{cases} \\ \begin{cases} (g\ a\ b\ c) \cdot \underline{0} = \underline{1} \\ (g\ a\ b\ c) \cdot \mathsf{succ} = \pi_1 \cdot \pi_1 \cdot \langle \langle j\ a\ b\ c, g\ a\ b\ c \rangle, f\ a\ b\ c \rangle \end{cases} \\ \begin{cases} (f\ a\ b\ c) \cdot \underline{0} = \underline{0} \\ (f\ a\ b\ c) \cdot \mathsf{succ} = \pi_2 \cdot \pi_1 \cdot \langle \langle j\ a\ b\ c, g\ a\ b\ c \rangle, f\ a\ b\ c \rangle \end{cases} \end{cases}$$

$\equiv$ { 3x Eq-+; 3x Fusao-+ }

$$\begin{cases} (j\ a\ b\ c) \cdot [\underline{0}, \mathsf{succ}] = [\underline{1}, add \cdot \langle add \cdot \langle (a*) \cdot \pi_1 \cdot \pi_1, (b*) \cdot \pi_2 \cdot \pi_1 \rangle, (c*) \cdot \pi_2 \rangle \cdot \langle \langle j\ a\ b\ c, g\ a\ b\ c \rangle, f\ a\ b\ c \rangle] \\ (g\ a\ b\ c) \cdot [\underline{0}, \mathsf{succ}] = [\underline{1}, \pi_1 \cdot \pi_1 \cdot \langle \langle j\ a\ b\ c, g\ a\ b\ c \rangle, f\ a\ b\ c \rangle] \\ (f\ a\ b\ c) \cdot [\underline{0}, \mathsf{succ}] = [\underline{0}, \pi_2 \cdot \pi_1 \cdot \langle \langle j\ a\ b\ c, g\ a\ b\ c \rangle, f\ a\ b\ c \rangle] \end{cases}$$

$\equiv$ { Defenição de in; Natural-id }

$$\begin{cases} (j\ a\ b\ c) \cdot \mathbf{in} = [\underline{1} \cdot id, add \cdot \langle add \cdot \langle (a*) \cdot \pi_1 \cdot \pi_1, (b*) \cdot \pi_2 \cdot \pi_1 \rangle, (c*) \cdot \pi_2 \rangle \cdot \langle \langle j\ a\ b\ c, g\ a\ b\ c \rangle, f\ a\ b\ c \rangle] \\ (g\ a\ b\ c) \cdot \mathbf{in} = [\underline{1} \cdot id, \pi_1 \cdot \pi_1 \cdot \langle \langle j\ a\ b\ c, g\ a\ b\ c \rangle, f\ a\ b\ c \rangle] \\ (f\ a\ b\ c) \cdot \mathbf{in} = [\underline{0} \cdot id, \pi_2 \cdot \pi_1 \cdot \langle \langle j\ a\ b\ c, g\ a\ b\ c \rangle, f\ a\ b\ c \rangle] \end{cases}$$

$\equiv$ { 3x Absorção-+ }

$$\begin{cases} (j\ a\ b\ c) \cdot \mathbf{in} = [\underline{1}, add \cdot \langle add \cdot \langle (a*) \cdot \pi_1 \cdot \pi_1, (b*) \cdot \pi_2 \cdot \pi_1 \rangle, (c*) \cdot \pi_2 \rangle] \cdot 1 + \langle \langle j\ a\ b\ c, g\ a\ b\ c \rangle, f\ a\ b\ c \rangle \\ (g\ a\ b\ c) \cdot \mathbf{in} = [\underline{1}, \pi_1 \cdot \pi_1] \cdot 1 + \langle \langle j\ a\ b\ c, g\ a\ b\ c \rangle, f\ a\ b\ c \rangle \\ (f\ a\ b\ c) \cdot \mathbf{in} = [\underline{0}, \pi_2 \cdot \pi_1] \cdot 1 + \langle \langle j\ a\ b\ c, g\ a\ b\ c \rangle, f\ a\ b\ c \rangle \end{cases}$$

$\equiv$ { Fokkinga }

$$\langle \langle j\ a\ b\ c, g\ a\ b\ c \rangle, f\ a\ b\ c \rangle = \llparenthesis \langle \langle [\underline{1}, add \cdot \langle add \cdot \langle (a*) \cdot \pi_1 \cdot \pi_1, (b*) \cdot \pi_2 \cdot \pi_1 \rangle, (c*) \cdot \pi_2 \rangle], [\underline{1}, \pi_1 \cdot \pi_1] \rangle, [\underline{0}, \pi_2 \cdot \pi_1] \rangle \rrparenthesis$$

$\square$

Mostrando assim que a podemos escrever a função $f$ a partir de um catamorfismo. Como obtemos o catamorfismo, podemos converter para um ciclo for de acordo com a definição:

$$\mathsf{for}\ b\ i = \llparenthesis [\underline{i}, b] \rrparenthesis$$

Assim sendo:

$$\llparenthesis \langle \langle [\underline{1}, add \cdot \langle add \cdot \langle (a*) \cdot \pi_1 \cdot \pi_1, (b*) \cdot \pi_2 \cdot \pi_1 \rangle, (c*) \cdot \pi_2 \rangle], [\underline{1}, \pi_1 \cdot \pi_1] \rangle, [\underline{0}, \pi_2 \cdot \pi_1] \rangle \rrparenthesis$$

$\equiv$ { Lei da troca }

$$\llparenthesis \langle [\langle \underline{1}, \underline{1} \rangle, \langle add \cdot \langle add \cdot \langle (a*) \cdot \pi_1 \cdot \pi_1, (b*) \cdot \pi_2 \cdot \pi_1 \rangle, (c*) \cdot \pi_2 \rangle, \pi_1 \cdot \pi_1 \rangle], [\underline{0}, \pi_2 \cdot \pi_1] \rangle \rrparenthesis$$

$\equiv$ { Lei da troca }

$$\llparenthesis [\langle \langle \underline{1}, \underline{1} \rangle, \underline{0} \rangle, \langle \langle add \cdot \langle add \cdot \langle (a*) \cdot \pi_1 \cdot \pi_1, (b*) \cdot \pi_2 \cdot \pi_1 \rangle, (c*) \cdot \pi_2 \rangle, \pi_1 \cdot \pi_1 \rangle, \pi_2 \cdot \pi_1 \rangle] \rrparenthesis$$

$\square$

Para sabermos o valor de inicialização do ciclo, igualamos a seguinte expressão:

$$
\begin{array}{cl}
 & \underline{i} = \langle\langle\underline{1},\underline{1}\rangle,\underline{0}\rangle \\
\equiv & \qquad \{ \text{ Universal-x } \} \\
 & \left\{ \begin{array}{l} \pi_1 \cdot \underline{i} = \langle\underline{1},\underline{1}\rangle \\ \pi_2 \cdot \underline{i} = \underline{0} \end{array} \right. \\
\equiv & \qquad \{ \text{ Universal-x; Fusão-const } \} \\
 & \left\{ \begin{array}{l} \left\{ \begin{array}{l} \pi_1 \cdot \pi_1 \cdot \underline{i} = \underline{1} \\ \pi_2 \cdot \pi_1 \cdot \underline{i} = \underline{1} \end{array} \right. \\ \underline{\pi_2 \cdot i} = \underline{0} \end{array} \right. \\
\equiv & \qquad \{ \text{ 2x Fusão-const; } \underline{a} = \underline{b} \text{ se } a = b \ \} \\
 & \left\{ \begin{array}{l} \left\{ \begin{array}{l} \underline{\pi_1 \cdot \pi_1 \cdot i} = \underline{1} \\ \underline{\pi_2 \cdot \pi_1 \cdot i} = \underline{1} \end{array} \right. \\ \pi_2 \cdot i = 0 \end{array} \right. \\
\equiv & \qquad \{ \text{ 2x } \underline{a} = \underline{b} \text{ se } a = b \ \} \\
 & \left\{ \begin{array}{l} \left\{ \begin{array}{l} \pi_1 \cdot \pi_1 \cdot i = 1 \\ \pi_2 \cdot \pi_1 \cdot i = 1 \end{array} \right. \\ \pi_2 \cdot i = 0 \end{array} \right. \\
\equiv & \qquad \{ \text{ Universal-x } \} \\
 & \left\{ \begin{array}{l} \pi_1 \cdot i = (1,1) \\ \pi_2 \cdot i = 0 \end{array} \right. \\
\equiv & \qquad \{ \text{ Universal-x } \} \\
 & i = ((1,1),0) \\
\square &
\end{array}
$$

Obtendo o *initial*.

Para apresentarmos a função do corpo do ciclo de forma mais amigável à preceção do utilizar vamos passá-la para a notação *point wise*, introduzindo variàveis:

$$
\begin{array}{ll}
 & \langle \langle add \cdot \langle add \cdot \langle (a*) \cdot \pi_1 \cdot \pi_1, (b*) \cdot \pi_2 \cdot \pi_1 \rangle, (c*) \cdot \pi_2 \rangle, \pi_1 \cdot \pi_1 \rangle, \pi_2 \cdot \pi_1 \rangle \; ((h,k),l) \\
\equiv & \quad \{ \text{ Def-split } \} \\
 & (\langle add \cdot \langle add \cdot \langle (a*) \cdot \pi_1 \cdot \pi_1, (b*) \cdot \pi_2 \cdot \pi_1 \rangle, (c*) \cdot \pi_2 \rangle, \pi_1 \cdot \pi_1 \rangle \; ((h,k),l), (\pi_2 \cdot \pi_1) \; ((h,k),l)) \\
\equiv & \quad \{ \text{ Def-split; Def-comp } \} \\
 & ((add \cdot \langle add \cdot \langle (a*) \cdot \pi_1 \cdot \pi_1, (b*) \cdot \pi_2 \cdot \pi_1 \rangle, (c*) \cdot \pi_2 \rangle \; ((h,k),l), (\pi_1 \cdot \pi_1) \; ((h,k),l)), \pi_2 \; (\pi_1 \; ((h,k),l))) \\
\equiv & \quad \{ \text{ 2x Def-comp; Def-proj } \} \\
 & ((add \; (\langle add \cdot \langle (a*) \cdot \pi_1 \cdot \pi_1, (b*) \cdot \pi_2 \cdot \pi_1 \rangle, (c*) \cdot \pi_2 \rangle \; ((h,k),l)), \pi_1 \; (\pi_1 \; ((h,k),l))), \pi_2 \; (h,k)) \\
\equiv & \quad \{ \text{ Def-split; 2x Def-proj } \} \\
 & ((add \; (add \cdot \langle (a*) \cdot \pi_1 \cdot \pi_1, (b*) \cdot \pi_2 \cdot \pi_1 \rangle \; ((h,k),l), ((c*) \cdot \pi_2) \; ((h,k),l)), \pi_1 \; (h,k)), k) \\
\equiv & \quad \{ \text{ 2x Def-comp; Def-proj } \} \\
 & ((add \; (add \; (((a*) \cdot \pi_1 \cdot \pi_1) \; ((h,k),l), ((b*) \cdot \pi_2 \cdot \pi_1) \; ((h,k),l)), (c*) \; (\pi_2 \; (h,l))), h), k) \\
\equiv & \quad \{ \text{ 2x Def-comp; Def-proj } \} \\
 & ((add \; (add \; (((a*) \cdot \pi_1) \; (\pi_1 \; ((h,k),l)), ((b*) \cdot \pi_2) \; (\pi_1 \; ((h,k),l))), (c*) \; l), h), k) \\
\equiv & \quad \{ \text{ 2x Def-proj; Definição de } * \} \\
 & ((add \; (add \; (((a*) \cdot \pi_1) \; (h,k), ((b*) \cdot \pi_2) \; (h,k)), c * l), h), k) \\
\equiv & \quad \{ \text{ 2x Def-comp } \} \\
 & ((add \; (add \; ((a*) \; (\pi_1 \; (h,k)), (b*) \; (\pi_2 \; (h,k))), c * l), h), k) \\
\equiv & \quad \{ \text{ 2x Def-proj } \} \\
 & ((add \; (add \; ((a*) \; h, (b*) \; k), c * l), h), k) \\
\equiv & \quad \{ \text{ 2x Definição de } * \} \\
 & ((add \; (add \; (a * h, b * k), c * l), h), k) \\
\equiv & \quad \{ \text{ Definição de } add \} \\
 & ((add \; (a * h + b * k, c * l), h), k) \\
\equiv & \quad \{ \text{ Definição de } add \} \\
 & ((a * h + b * k + c * l, h), k) \\
\Box &
\end{array}
$$

Funções auxiliares pedidas:

$$
\begin{array}{l}
loop \; a \; b \; c \; ((j,g),f) = ((a * j + b * g + c * f, j), g) \\
initial = ((1,1),0) \\
wrap = \pi_2
\end{array}
$$

## Problema 2

De forma a resolvermos o problema 2 é necessário recorrer ao seguinte diagrama de um anamorfismo:

$$
\begin{tikzcd}
Exp \; S \; S & & S + S \times (Exp \; S \; S)^* \arrow[ll, "\mathsf{in}_{Exp}"'] \\
S^* \arrow[u, "tax"] \arrow[r, "out"] \arrow[rr, bend right, "gene"'] & S + S \times S^* \arrow[r, "\cdots"] & S + S \times (S^*)^* \arrow[u, "id + id \times tax^*"']
\end{tikzcd}
$$

O gene será uma a composição de duas funções, a *out* e um função que é um $[\cdot,\cdot]$. Para podermos definir as duas componentes do $[\cdot,\cdot]$ é necessário perceber o resultado da função *out*. A função *out* é o *out* das listas não vazias. No caso da lista só com um elemento, esse é etiquetado à esquerda e neste caso aplicamos a função identidade, visto que, este elemento será uma variável na árvore de expressões. No caso da lista com mais de um elemento divide a cabeça e a causa e o par é etiquetado à direita. Para facilidade da interpertar o raciocínio apresentamos, através de um caso específico, como fica a saída:

```
("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"])
```

Neste segundo caso na cabeça ficará o primeiro elemento da lista, ou seja, a *String* não identada que é um operador, à qual basta aplicar a função identidade. Na cauda da lista ficam as *String* com identação. Para remover a identação é necessário remover os 4 espaços de todos os elementos e para isso usamos a função map $(take\ 4)$ ficando o par da seguinte forma:

```
("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"])
```

Em seguida é necessário dividir esta lista em sublistas, ou seja, fazer um agrupamento. Cada lista nova terá a cabeça, ou seja, a *String* sem identação e as *String* com hierarquia inferior a esta, com identação. Para isso usamos a seguinte função:

groupBy (\x y = take 4 y == "    ")

Esta condição permite que sejam geradas as lista da forma pretendida, visto que, a função groupBy mantem uma lista atual e faz a comparação do último elemento da lista atual com o próximo elemento da lista, se é verificada adiciona o elemento à lista atual, caso contrário insere a lista atual na lista de sublistas e adiciona uma nova lista atual com este elemento. Logo, no nosso caso em especifico:

```
("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"]])
```

Assim sendo, o gene de *tax*:

$$gene = (id + (id \times groupBy\ (\lambda x\ y \rightarrow take\ 4\ y \equiv \texttt{"}\qquad\texttt{"}) \cdot \mathsf{map}\ \ (drop\ 4))) \cdot out$$

A função de pós-processamento pode ser definida através de um catamorfismo com o seguinte diagrama:

$$
\begin{tikzcd}
Exp\ S\ S \arrow[d, "(\!|g|\!)_{Exp}"'] & & S + S \times (Exp\ S\ S)^* \arrow[ll, "\mathsf{in}_{Exp}"'] \arrow[d, "id + id \times (\!|g|\!)_{Exp}"] \\
(S^*)^* & & S + S \times ((S^*)^*)^* \arrow[ll, "gnp"]
\end{tikzcd}
$$

Depois de identificarmos o catamorfismo de árvores de expressão, passamos à fase de identificar qual o seu gene. No caso do lado esquerdo apenas precisamos de colocar a *String* dentro de uma lista e esta dentro de outra lista. No caso do lado direito é necessário perceber qual é o resultado do lado direito do par, que são as listas com os caminhos que é possível tomar a partir da *String* do lado esquerdo do par. Assim sendo, necessitamos de introduzir à cabeça de cada um desses caminhos a *String* do lado esquerdo. Para isso, primeiro a lista com as listas dos caminhos são concatenadas para possuírmos a lista com os caminhos. Em seguida, acrescentamos à cabeça de todos os caminhos a *String* do lado esquerdo e acrescentamos mais um caminho para que seja possível chegar ao nível desta *String*, visto que queremos todos os caminhos da árvore.

Culminando na função de pós-processamento:

$$
\begin{aligned}
&post = (\!| gnp |\!)_{Exp} \\
&\quad \mathbf{where}\ gnp = [gnp1, gnp2] \\
&\qquad gnp1\ x = [[x]] \\
&\qquad gnp2\ (x, l) = [x] : \mathsf{map}\ \ (x{:})\ (concat\ l)
\end{aligned}
$$

## Problema 3

A primeira parte do problema 3 consiste no cálculo dos gene das funções rose2List e squares.

Estes dois genes quando compostos formam o seguinte hilomorfismo:

$$
\begin{aligned}
&sierpinski :: (Square, Int) \to [Square] \\
&sierpinski = [\![ gr2l, gsq ]\!]_R
\end{aligned}
$$

Para resolvermos o hilomorfismo abordado anteriormente, nós começamos por descobrir o gene do catamorfismo rose2List.Sendo assim obtivemos o seguinte diagrama:

$$
\begin{tikzcd}
Rose\ A \arrow[d, "(\!|gr2l|\!)_R"'] & & A \times (Rose\ A)^* \arrow[ll, "inRose"'] \arrow[d, "(RecRose\ (\!|gr2l|\!)_R)"] \\
A^* & & A \times (A^*)^* \arrow[ll, "gr2l"]
\end{tikzcd}
$$

Tendo em conta o diagrama apresentado podemos reparar que o único ponto de falha existente na função é no cálculo do seu gene uma vez que tudo o resto já se encontra devidamente definido. Tendo em atenção aos tipos de entrada e de saída que o gene possui, conseguimos perceber claramente qual o trabalho desenvolvido por este.

Se repararmos o gene recebe um par constituído pela cabeça de uma lista, de tipo A, e uma lista de listas com elementos do tipo A. O gene devolve apenas uma lista,com elementos do tipo A.

Sendo assim conseguimos perceber que o gene irá ter de concatenar a lista de listas, formando apenas uma lista e colocando o elemento A á cabeça da lista criada.

Com base nesta explicação do diagrama conseguimos perceber que a função rose2List pode ser definida da seguinte forma:

$$
\begin{aligned}
&rose2List = (\!| gr2l |\!)_R \\
&gr2l\ (a, l) = a : concat\ l
\end{aligned}
$$

Tal como aconteceu para o cálculo do gene do catamorfismo, adotamos a mesma estratégia para o cálculo do gene do anamorfismo. Sendo assim o diagrama que corresponde ao anamorfismo é o seguinte:

$$
\begin{tikzcd}
\mathit{Rose\ Square} & & \mathit{Square} \times (\mathit{Rose\ Square})^* \arrow[ll, "\mathit{inRose}"'] \\
(\mathit{Square}, N_0) \arrow[u, "{[\![ \mathit{gsq} ]\!]_R}"] \arrow[r, "\mathit{outNat} \cdot \pi_2"] \arrow[rr, bend right, "\mathit{gsq}"'] & 1 + (N_0) \arrow[r, "\cdots"] & \mathit{Square} \times (\mathit{Square}, N_0)^* \arrow[u, "\mathit{RecRose}\ (\!| \mathit{gr2l} |\!)_R"']
\end{tikzcd}
$$

Visualizando o diagrama do anamorfismo, conseguimos perceber que apenas temos de calcular o gene do anamorfismo. Sendo assim, se repararmos no diagrama do anamorfismo, o gene recebe um par composto por (Square,N0) e tem de devolver um par composto por uma square e uma lista composta por (Square,$N_0$).

Com base no que foi dito anteriormente, conseguimos perceber que o gene tem de devolver uma square caso o natural seja igual a 0, ou então, devolve uma lista de pares (Square,N0) caso o N0 seja diferente de 0.

Esta lista, "contém"os quadrados do tapete de Sierpinski. Assim cada elemento da lista, tem diferentes coordenadas.

Sendo assim, conseguimos perceber qual o trabalho que o gene (gsq) irá ter de realizar.

$$
\begin{array}{l}
\mathit{squares} = [\![ \mathit{gsq} ]\!]_R \\
\mathit{gsq} = \langle \mathit{gsq1}, \mathit{gsq2} \rangle \\
\mathit{gsq1}\ (((x,y),z),w) = ((x+z/3, y+z/3), z/3) \\
\mathit{gsq2}\ (((x,y),z),w) = \mathbf{if}\ w > 0\ \mathbf{then}\ [(((x,y),z/3),w-1), \\
\quad (((x+z/3,y),z/3),w-1), \\
\quad (((x+(2*z/3),y),z/3),w-1), \\
\quad (((x,y+z/3),z/3),w-1), \\
\quad (((x+2*z/3,y+z/3),z/3),w-1), \\
\quad (((x,y+2*z/3),z/3),w-1), \\
\quad (((x+z/3,y+2*z/3),z/3),w-1), \\
\quad (((x+2*z/3,y+2*z/3),z/3),w-1)] \\
\quad \mathbf{else}\ [\,]
\end{array}
$$

Nesta fase do problema 3, surge uma nova abordagem ao problema. Para explicar esta nova abordagem temos o seguinte:

- Na primeira abordagem feita ao problema, nós temos de inserir a square inicial e a profundidade até á qual queremos simular;

- Nesta nova abordagem, nós apenas teremos de inserir a profundidade até á qual queremos simular e a função dar-nos-á os tapetes devidamente prontos;

Desta forma o resultado final, pode ser representado por este hilomorfismo:

Com base na útlima abordagem conseguimos perceber, que iremos ter duas funções: a carpets e a present. A função carpets recebe como input um inteiro, que representa a profundidade N e devolve uma lista de lista de Squares, que representam os tapetes de profundidade 0 até á profundidade N-1. Sendo assim, para uma profundidade igual a 0, não é criado qualquer tipo de tapete. Logo a lista resultado é uma lista vazia.

Para profundidades superiores a 0, é feito é o seguinte:

1º- Cálculo da carpets para profundidades inferiores á dada como parâmetro;

2º- Cálculo do sierpinski para profundidade inferior á dada como parâmetro.

Com isto percebemos como funciona a função carpets.

$$
\begin{array}{l}
\mathit{carpets}\ 0 = [\,] \\
\mathit{carpets}\ n = \mathit{carpets}\ (n-1) +\!\!+ [\mathit{sierpinski}\ (((0,0),32),n-1)]
\end{array}
$$

A função present consome o resultado produzido pela carpets uma vez que ambas compõem um hilomorfismo.

Assim o trabalho realizado pela função present é o seguinte:

-Cada elemento da lista [[Square]], que representa um tipo de tapete de sierpinski, é desenhado no ecrã(I∅) pela função drawSq;

-Entre o desenho de cada tapete no ecrã,o programa tem de "parar"durante um determinado intervalo de tempo, sendo assim é chamada a função await.

O mmap é utilizado para aplicar o monade de I∅a cada elemento da lista dada como input. Desta forma, conseguimos perceber o funcionamento da função present que é composta por um monade de I∅.

$$present = mmap\ (\lambda l \rightarrow \mathbf{do}\ \{drawSq\ l; await\})$$

Depois de calculadas estas duas funções, conseguimos por fim finalizar o hilomorfismo inicial, que constroi tapetes de Sierpinski precisando apenas de uma determinada profundidade.

## Problema 4

Neste problema 4, é nos pedido que façamos uma espécie de simulação do campeonato do mundo que ocorreu este ano. Para isso vamos ter duas partes:

-Uma versão não probabilística em que se sabe à partida para um dado jogo, quem o vai vencer.

-Uma versão com probabilidades, onde se usa o mónade Dist(que usa distribuições probabilísticas).

### Versão não probabilística

Nesta primeira fase não probabilística, primeiramente, é nos pedido para definir uma alternativa à função genérica consolidate, que seja um catamorfismo de listas. Para isso, tivemos de definir o gene do catamorfismo que está indicado em baixo.

Gene de $consolidate'$:

$$\begin{array}{l}
cgene :: (Eq\ a, Num\ b) \Rightarrow () + ((a,b),[(a,b)]) \rightarrow [(a,b)] \\
cgene = [cgene1, cgene2] \\
\quad \mathbf{where}\ cgene1 = nil \\
\qquad cgene2\ ((a,b),[\,]) = [(a,b)] \\
\qquad cgene2\ ((a,b),(a_1,b_1):l) = \mathbf{if}\ a \equiv a_1\ \mathbf{then}\ (a,b+b_1):l \\
\qquad\quad \mathbf{else}\ (a_1,b_1):cgene2\ ((a,b),l)
\end{array}$$

Geração dos jogos da fase de grupos:

Para decidir quem é o vencedor de um determinado jogo, tivemos de definir a função matchResult, que recebe como argumentos a função gsCriteria que mediante um certo critério, calcula o resultado de um jogo, e um argumento de tipo Match, que é um par de Teams.

A função encontra-se definida em baixo.

$$\begin{array}{l}
matchResult\ gsCriteria\ (t1,t2)\ |\ gsCriteria\ (t1,t2) \equiv Just\ t1 = [(t1,3),(t2,0)] \\
\quad |\ gsCriteria\ (t1,t2) \equiv Just\ t2 = [(t1,0),(t2,3)] \\
\quad |\ otherwise = [(t1,1),(t2,1)]
\end{array}$$

De seguida, foi nos pedido para definir a função pairup, em que generateMatches se baseia. Esta função, basicamente recebe uma lista de elementos de um certo tipo b, e cria uma lista de pares de elementos desse tipo.

$$pairup = \perp$$

Por último, foi nos pedido para definir o gene glt, que será usado na função initKnockoutStage. O gene encontra-se definido em baixo.

$$glt = \perp$$

### Versão probabilística

Para a versão probabilística, como é dito anteriormento, primereiro avaliamos o impacto de $gsCriteria$ virar monádica. Para isso as funções que sofrem alterações são a $pinitKnockoutStage$, $pgroupWinners$ e $pmatchResult$. Na $pmatchResult$ verificamos que como o criterio mudou é necessário monadificar o código do matchResult:

$$
\begin{array}{l}
matchResult\ gsCriteria\ (t1,t2) \mid gsCriteria\ (t1,t2) \equiv Just\ t1 = [(t1,3),(t2,0)] \\
\quad \mid gsCriteria\ (t1,t2) \equiv Just\ t2 = [(t1,0),(t2,3)] \\
\quad \mid otherwise = [(t1,1),(t2,1)]
\end{array}
$$

Para monadificar é feita usamos a notação **let**:

$$
\begin{array}{l}
matchResult\ gsCriteria\ (t1,t2) = \mathbf{let}\ x = gsCriteria\ (t1,t2)\ \mathbf{in}\ f\ x \\
\quad \mathbf{where}\ f\ Nothing = [(t1,1),(t2,1)] \\
\qquad f\ (Just\ t) = \mathbf{if}\ t \equiv t1\ \mathbf{then}\ [(t1,3),(t2,0)]\ \mathbf{else}\ [(t1,0),(t2,3)]
\end{array}
$$

Agora passamos para a versão monádica:

$$
\begin{array}{l}
pmatchResult\ criteria\ (t1,t2) = \mathbf{do}\ \{ r \leftarrow criteria\ (t1,t2); return\ (f\ r) \} \\
\quad \mathbf{where}\ f\ Nothing = [(t1,1),(t2,1)] \\
\qquad f\ (Just\ t) = \mathbf{if}\ t \equiv t1\ \mathbf{then}\ [(t1,3),(t2,0)]\ \mathbf{else}\ [(t1,0),(t2,3)]
\end{array}
$$

Passamos agora à função $pgroupWinners$, e esta necessita de ser alterada em alguns aspetos. Primeiro temos de descobrir os resultados de cada partida através do novo critério, e para isso aplicamos o map monádico, o $mmap$. Depois temos de aplicar o conjunto de funções necessárias para descobrir as duas melhores equipas do grupo, e para isso é necessário serem executadas no interior do monade e para isso usamos o fmap. Dando origem à seguinte definição:

$$
\begin{array}{l}
pgroupWinners :: (Match \to \mathsf{Dist}\ (Maybe\ Team)) \to [Match] \to \mathsf{Dist}\ [Team] \\
pgroupWinners\ criteria = \mathsf{fmap}\ (best\ 2 \cdot consolidate \cdot concat) \cdot mmap\ (pmatchResult\ criteria)
\end{array}
$$

Por fim é necessário monadificar a função $initKnockoutStage$. Neste caso, esta é usada após a composição monádica, como indicado:

$$
\begin{array}{ll}
& pgroupStage = pinitKnockoutStage \bullet psimulateGroupStage \cdot genGroupStageMatches \\
\equiv & \quad \{\ (66)\ \} \\
& pgroupStage = \lambda mu \cdot T\ pinitKnockoutStage \cdot psimulateGroupStage \cdot genGroupStageMatches \\
\square &
\end{array}
$$

Assim sendo, pinitKnockoutStage vai ser aplicada da seguinte forma:

$$
\begin{tikzcd}
Dist(Team^*)^* \arrow[d, "\mathsf{Dist}\ pinitKnockoutStage"'] \\
\mathsf{Dist}\ (LTree\ Team)
\end{tikzcd}
$$

Concluindo o que é necessário fazer é aplicar a função de return à função de $initKnockoutStage$:

$$
pinitKnockoutStage\ l = \mathbf{do}\ \{ return\ (initKnockoutStage\ l) \}
$$

# Índice