}{¿

# Cálculo de Programas

Trabalho Prático

LEI — 2022/23

*Departamento de Informática*

Universidade do Minho

Janeiro de 2023

| Grupo nr. | 21 |
|---|---|
| a85508 | Marco António Sampaio |
| a93313 | Diogo Miguel Araújo |

## 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 A 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 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 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 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]

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

[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.

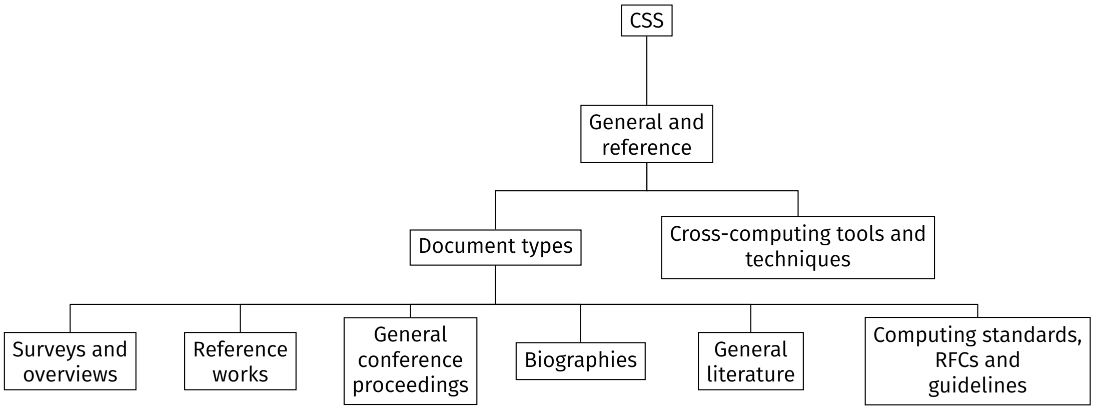



Figura 1: Fragmento de $acm\_ccs$ representado sob a forma de uma árvore do tipo Exp.

$$
\begin{array}{l}
tudo :: [String] \to [[String]] \\
tudo = post \cdot tax
\end{array}
$$

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 ítens).

Defina a função $post :: Exp\ String\ String \to [[String]]$ 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 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).

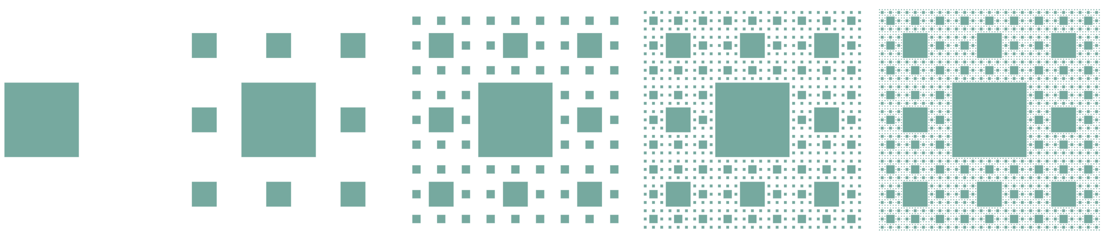


Figura 2: Construção do tapete de Sierpinski com profundidade 5.

**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.

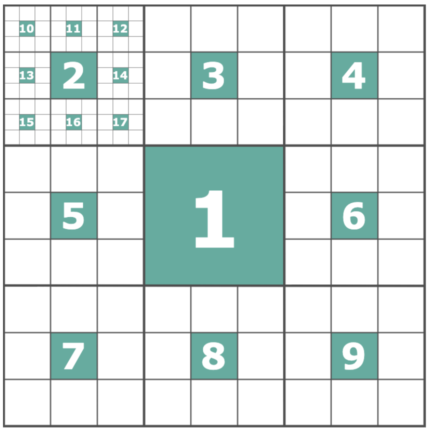



Figura 3: Tapete de Sierpinski com profundidade 2 e com os quadrados enumerados.

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

$$\begin{array}{l} \textbf{type}\ \mathit{Square} = (\mathit{Point}, \mathit{Side}) \\ \textbf{type}\ \mathit{Side} = \mathit{Double} \\ \textbf{type}\ \mathit{Point} = (\mathit{Double}, \mathit{Double}) \end{array}$$

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 poderá[4] corresponder à árvore da figura 4.

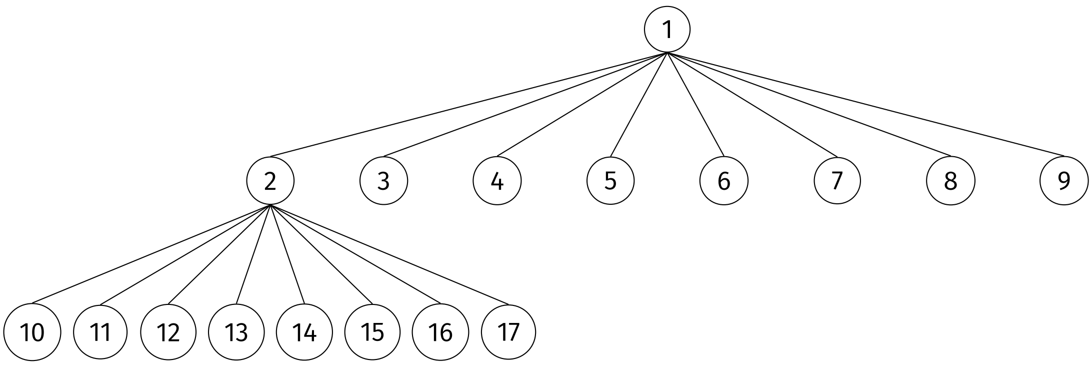



Figura 4: Possível árvore de suporte para a construção da fig. 3.

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.

[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.

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

**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*:

$$\begin{array}{l} sierpinski :: (Square, Int) \rightarrow [Square] \\ sierpinski = [\![ gr2l, gsq ]\!]_R \end{array}$$

**Trabalho a fazer:**

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

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

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.

$$\begin{array}{l} constructSierp :: Int \rightarrow \mathsf{IO}\ [()] \\ constructSierp = present \cdot carpets \end{array}$$

**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 $present :: [[Square]] \rightarrow \mathsf{IO}\ [()]$ 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 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

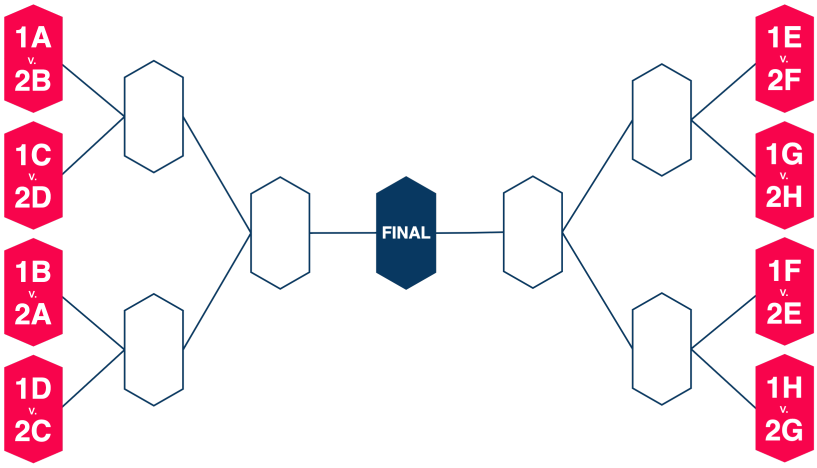



Figura 5: O "mata-mata"

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:

$$\begin{array}{l} winner :: Team \\ winner = wcup\ groups \end{array}$$

$$wcup = knockoutStage \cdot groupStage$$

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:*
>
> $$\begin{array}{l} groupStage :: [Group] \rightarrow \mathsf{LTree}\ Team \\ groupStage = initKnockoutStage \cdot simulateGroupStage \cdot genGroupStageMatches \end{array}$$
>
> *Comecemos então por definir a função $genGroupStageMatches$ que gera os jogos da fase de grupos:*
>
> $$\begin{array}{l} genGroupStageMatches :: [Group] \rightarrow [[Match]] \\ genGroupStageMatches = \mathsf{map}\ \ generateMatches \end{array}$$
>
> *onde*
>
> $$\mathbf{type}\ Match = (Team, Team)$$
>
> *Ora, sabemos que nos foi dada a função*
>
> $$gsCriteria :: Match \rightarrow Maybe\ Team$$
>
> *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*

> $simulateGroupStage :: [[Match]] \to [[Team]]$
> $simulateGroupStage = \mathsf{map}\ \ (groupWinners\ gsCriteria)$
>
> *que simula a fase de grupos e dá como resultado a lista dos vencedores, recorrendo à função* $groupWinners$*:*
>
> $groupWinners\ criteria = best\ 2 \cdot consolidate \cdot (\gg\!= matchResult\ criteria)$
>
> *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 C, página 12 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":

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

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

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 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-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 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]

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

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:

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

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

[8] Lei (3.95) em [**?**], página 112.

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

Obter-se-á de imediato

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

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:

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

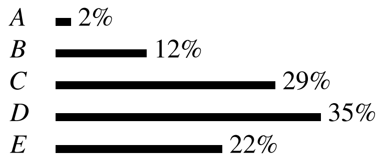
## C O mónade das distribuições probabilísticas

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

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

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

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

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 $g : A \to \mathsf{Dist}\ B$ e $f : B \to \mathsf{Dist}\ C$ 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:

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

## 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”:

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

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:

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

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

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

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

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

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

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

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

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

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

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

Auxiliares:

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

Função binária monádica $f$:

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

Monadificação de uma função binária $f$:

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

$$mbin :: Monad\ m \Rightarrow ((a,b) \rightarrow c) \rightarrow (m\ a, m\ b) \rightarrow m\ c$$
$$mbin = mmbin \cdot (return \cdot)$$

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

Funções auxiliares pedidas:

$$loop\ a\ b\ c = fblLoop\ a\ b\ c$$
$$initial = ((0,0),0)$$
$$wrap = \pi_2$$

### E.0.1 Primeira tentativa

Numa primeira abordagem a este problema, tentamos reduzir analiticamente a função dada, f, de maneira a que o seu nível de operação passasse de (n+3) para (n+1). Contudo, após algumas tentativas, não conseguimos deduzir as dependências que se traduziam nas funções mútuamente recursivas. Apesar disso, conseguimos perceber que há um pormenor que se manteve nesta tentativa de solução, o facto de haver necessidade de construir esta nova função através de três funções, tal como já nos sugeria a original ao ser constituída por três casos base, na sua definição (não é uma regra, mas pode induzir para o resultado correto).

### E.0.2 Segunda tentativa

Numa fase em que a solução não estava a ser muito fácil de calcular, tentamos seguir rumo pelo desenho de diagramas e uma pré-configuração de como o sistema se deveria comportar. Para tal, configuramos o seguinte diagrama para nos auxiliar neste processo.

$$\begin{tikzcd}
N \arrow[r, bend left, "outNat_0"] \arrow[d, "<<h,g>,f>"'] & 1+N \arrow[l, bend left, "inNat_0"] \arrow[d, "F<<h,g>,f>"] \\
(A \times B) \times C & F((A \times B) \times C) \arrow[l, "<<k,j>,i>"]
\end{tikzcd}$$

Como estamos a operar sobre números naturais, N, sabemos como se comportam os in e out dos mesmos.

```
in = [0, succ]
out 0 = Left ()
out (n+1) = Right n
```

Já que não conseguimos deduzir diretamente as funções necessárias, podemos atentar em como o sistema se deverá comportar para, desde o início, chegar à solução, recursivamente. Ora, preparando um esquema com dados relativos às respetivas soluções, chegamos ao seguinte:

```
                     c=1  b=1  a=1
N = 0    Left ()     ((_, _), 0)
N = 1    Right 0     ((_, _), 1)
N = 2    Right 1     ((_, _), 1)
N = 3    Right 2     ((_, _), 2)
N = 4    Right 3     ((_, _), 4)
N = 5    Right 4     ((_, _), 7)
N = 6    Right 5     ((_, _), 13)
N = 7    Right 6     ((_, _), 24)

                     c=3  b=2  a=1
N = 0    Left ()     ((_, _), 0)
N = 1    Right 0     ((_, _), 1)
N = 2    Right 1     ((_, _), 1)
N = 3    Right 2     ((_, _), 3)
N = 4    Right 3     ((_, _), 8)
N = 5    Right 4     ((_, _), 17)
N = 6    Right 5     ((_, _), 42)
N = 7    Right 6     ((_, _), 100)
```

A partir de um dado inteiro, conseguimos deduzir como as funções mutuamente recursivas se devem comportar.
Ora, o resultado já nos é apresentado pelo enunciado do problema, sendo colocado no último elemento da nossa configuração pela prestação definida da função **warp=p2**, cujo objetivo é retirar o resultado do problema.
Os inteiros e a sua configuração do **out** servem meramente como guias para nos orientarmos no processo de recursividade.
Passando para os primeiros três resultados, apesar dos diferentes fatores 'a', 'b' e 'c', percebemos que estes são constantes, o que faz sentido de acordo com a definição original da função f. Assim, basta-nos encontrar relações entre os valores, com o auxílio desses fatores, que, a partir de um nível anterior, consigam definir o resultado do nível seguinte.

Após um curto período de experimentação, chegamos às seguintes relações:

1. O primeiro elemento de um nível é o segundo do nível anterior.
2. O segundo elemento é o terceiro do nível anterior.
3. O terceiro elemento é o somatório de cada um dos elementos do nível anterior, multiplicado pelo seu fator.

Seguindo as regras apresentadas acima, chegamos à seguinte configuração (para os espaços em falta, sugere-se a aplicação das referidas regras, de maneira a perceber o processo intrínseco à configuração):

```
                     c=1  b=1  a=1
N = 0    Left ()     ((0, 0), 0)
N = 1    Right 0     ((0, 0), 1)
N = 2    Right 1     ((0, 1), 1)
N = 3    Right 2     ((1, 1), 2)
N = 4    Right 3     ((1, 2), 4)
N = 5    Right 4     ((2, 4), 7)
N = 6    Right 5     ((_, _), 13)
N = 7    Right 6     ((_, _), 24)

                     c=3  b=2  a=1
N = 0    Left ()     ((_, _), 0)
N = 1    Right 0     ((_, _), 1)
N = 2    Right 1     ((_, _), 1)
N = 3    Right 2     ((_, 1), 3)
N = 4    Right 3     ((1, 3), 8)
N = 5    Right 4     ((3, 8), 17)
N = 6    Right 5     ((8, 17), 42)
N = 7    Right 6     ((17, 42), 100)
```

Traduzindo as relações identificadas, chegamos à seguinte configuração:

-- initial = ((0, 0), 0)

-- fblH a b c ((ee, ed), d) = ed

$$fblH\ a\ b\ c = \pi_2 \cdot \pi_1$$

--

-- fblG a b c ((ee, ed), d) = d

$$fblG\ a\ b\ c = \pi_2$$

--

-- fblF a b c ((ee, ed), d) = a * d + b * ed + c * ee

$$fblF\ \_\ \_\ \_\ ((\_,\_),0) = 1$$
$$fblF\ \_\ \_\ \_\ ((\_,0),1) = 1$$
$$fblF\ a\ b\ c\ ((ee,ed),d) = a*d+b*ed+c*ee$$

--

-- <h, g >= <fblH, fblG >

$$fblHG\ a\ b\ c = \langle fblH\ a\ b\ c, fblG\ a\ b\ c\rangle$$

--

-- <<h, g >, f >= <fblHG, fblF >

$$fblLoop\ a\ b\ c = \langle fblHG\ a\ b\ c, fblF\ a\ b\ c\rangle$$

De maneira a justificar a necessidade de evoluir certos algoritmos para a sua versão em recursividade mútua, fizemos alguns testes com ambas as versões desta função, f e fbl. Para recolher as métricas utilizamos o comando ":set +s"para ativar as estatísticas de tempo e espaço, das quais conseguimos retirar informações como o tempo gasto pelo CPU e o número de *bytes* da memória necessários para executar os cálculos necessários. Na primeira tabela, as reticências devem-se a um tempo de espera tão demorado que não conseguimos obter os resultados. Na segunda, o único valor com o mesmo formato deve-se a ser um número tão grande que não se encaixa no formato da tabela.

| Function | a | b | c | n | CPU Time (secs) | #Bytes of memory | Answer |
|---|---|---|---|---|---|---|---|
| f | 1 | 2 | 3 | 5 | 0,00 | 63,840 | 17 |
| f | 1 | 2 | 3 | 10 | 0,01 | 214,504 | 1331 |
| f | 1 | 2 | 3 | 15 | 0,02 | 3,356,080 | 100425 |
| f | 1 | 2 | 3 | 20 | 0,27 | 69,456,856 | 7579198 |
| f | 1 | 2 | 3 | 25 | 5,91 | 1,460,849,568 | 572024206 |
| f | 1 | 2 | 3 | 30 | 132.30 | 30,749,309,288 | 43172337417 |
| f | 1 | 2 | 3 | 35 | ... | ... | ... |

| Function | a | b | c | n | CPU Time (secs) | #Bytes of memory | Answer |
|---|---|---|---|---|---|---|---|
| fbl | 1 | 2 | 3 | 5 | 0,00 | 66,496 | 17 |
| fbl | 1 | 2 | 3 | 10 | 0,00 | 76,632 | 1331 |
| fbl | 1 | 2 | 3 | 15 | 0,00 | 86,768 | 100425 |
| fbl | 1 | 2 | 3 | 20 | 0,01 | 96,216 | 7579198 |
| fbl | 1 | 2 | 3 | 25 | 0,01 | 106,352 | 572024206 |
| fbl | 1 | 2 | 3 | 30 | 0,01 | 116,488 | 43172337417 |
| fbl | 1 | 2 | 3 | 100 | 0,01 | 266,768 | 8,4E+36 |
| fbl | 1 | 2 | 3 | 1000 | 0,03 | 2,721,432 | ... |

## Problema 2

Gene de *tax*:

$$gene = ggene$$

### E.0.3 Primeiro passo: hierarquias

O problema é-nos apresentado, desde logo, como contendo um tipo de hierarquia entre os seus elementos, sendo os mesmos *strings*. A hierarquia é definida pelo número de espaços em braco que contém no seu início, até alcançar um caracter diferente. A cada quatro espaços brancos, incrementa-se um nível.

```
    #espacos     #hierarquia
        0            0
        4            1
        8            2
      (...)        (...)
```

A partir daqui, podemos, desde já, definir as funções que arrecadarão com o trabalho de retirar o nível de hierarquia de cada uma das *strings*.

-- calculate the number of 'space' characters in the beginning of the string

$$
\begin{array}{l}
\mathit{initialSpaces}\ (h : t) \\
\quad \mid h \equiv \texttt{' '} = \mathsf{succ}\ \ (\mathit{initialSpaces}\ t) \\
\quad \mid \mathit{otherwise} = 0
\end{array}
$$

-- calculate the hierarchy level of current string

$$
\begin{array}{l}
\mathit{expHierarchyLevel}\ \mathit{str} = \mathit{n\_spaces} \div 4 \\
\quad \textbf{where}\ \mathit{n\_spaces} = \mathit{initialSpaces}\ \mathit{str}
\end{array}
$$

### E.0.4 Segundo passo: gene do anamorfismo

O diagrama apresentado no enunciado já nos mostra como parte do gene deste anamorfismo se comporta. Ao utilizar o out de listas não vazias, ou temos um único elemento, ou a cabeça da lista e a cauda da mesma, como na configuração do out aplicado a listas normais.

$$
S^* \xrightarrow{\quad out \quad} S + S \times S^* \xrightarrow{\quad \cdots \quad} S + S \times (S^*)^*
$$

A partir da linha acima exposta, percebemos que, quando aplicado, o nosso gene tem de preparar os valores que recebe com o formato correto para que o bifuntor de base opere na recursividade. Assim, a partir de ($S$ + $S$) temos de transformar no tipo ($S$ + $S$), enquanto no lado de $S^*$ precisamos transformar esta lista numa lista de listas, *(S*)**.

Tratando-se de uma disjunção, o nosso gene será obrigatoriamente do tipo (algo + algo).
Sendo uma execução em paralelo, do lado direito, o gene será, por sua vez, do tipo (algo + (algo $\times algo$)).

Caso tenhamos um único elemento, o construtor de *Exp Tree* consegue lidar com ele, há apenas necessidade de o preservar (identidade). O mesmo acontece para a cabeça de uma lista. Esta será um nodo pai na consequente árvore, restando apenas definir os seus filhos (identidade).
Ora, a partir disto, conseguimos definir algo como sendo do tipo (id + (id $\times algo$)).

Como o objetivo é pegar numa lista e transformá-la numa lista de listas, temos já uma dica de que é necessário agrupar valores. A questão é: Como? Com que restrições/fatores? Ora, temos a indicação de que temos de seguir uma determinada hierarquia e o cálculo da mesma já está preparado (primeiro passo). Assim, apenas precisamos de encontrar uma função que agrupe elementos tendo em conta uma determinada relação entre eles, algo que já existe: *groupBy*.

Desta maneira, agrupamos elementos de uma lista cujo nível hierárquico esteja em ordem crescente de valores!

-- after the out of a non-empty list, this is the gene

$$ggene = (id + (id \times groupBy\ (\lambda x\ y \rightarrow expHierarchyLevel\ x < expHierarchyLevel\ y))) \cdot out$$

Função de pós-processamento:

$$\begin{array}{l}
post :: Exp\ String\ String \rightarrow [[String]] \\
post\ (Var\ s) = [[s]] \\
post\ (Term\ a\ [\,]) = [[a]] \\
post\ (Term\ s\ es) = [[s]] +\!\!+ \mathsf{map}\ \ (s:)\ (concatMap\ post\ es)
\end{array}$$

## Problema 3

$$\begin{array}{l}
squares = [\![\, gsq \,]\!]_R \\
gsq = divide_ \\
rose2List = (\![ gr2l ]\!)_R \\
gr2l = (cons \cdot (id \times concat)) \\
carpets = reverse \cdot [\![ carpets_gene ]\!] \\
present :: [[Square]] \rightarrow \mathsf{IO}\ [()] \\
present = (\![ ppresent ]\!)
\end{array}$$

### E.0.5 Primeiro passo - anamorfismo de Rose Tree

Um hilomorfismo é constituído por um catamorfismo após um anamorfismo. Nesta sequência, precisamos preliminarmente de definir o anamorfismo que construirá uma *Rose tree* a partir do *input* que a função *squares* recebe: *(Square, Int)*.

Ora, traduzindo o tipo de entrada para um tipo mais genérico, podemos ter o seguinte:

Square = ((x, y), side) = ((A $\times B) \times C)$
$(Square, Int) = ((A \times B) \times C) \times P$

Sobre *rose trees* sabemos ainda que:

$$\begin{tikzcd}
Rose\ A \arrow[r, bend left, "out"] & A \times (Rose\ A)^* \arrow[l, bend left, "in"]
\end{tikzcd}$$

Assim, podemos preparar já o diagrama que define o anamorfismo a construir.

$$\begin{tikzcd}
(((A \times B) \times C) \times P) \arrow[rr, "divide"] \arrow[d, "{[(divide)]}"] & & ((A \times B) \times C) \times (((A \times B) \times C) \times P) \arrow[d, "{B(id,\ [(divide)])}"] \\
Rose((A \times B) \times C) \arrow[rr, bend left, "out"] & & ((A \times B) \times C) \times Rose((A \times B) \times C)^* \arrow[ll, bend left, "in"]
\end{tikzcd}$$

Com o diagrama feito, basta preparar os pontos. Para cada ponto de referência, temos de calcular os oito pontos relativos. Tivemos cuidado para ordenar esses pontos consoante a ordem relativa esquerda-direita, baixo-cima para produzir o efeito da ordem utilizada nas figuras fornecidas no enunciado do problema.

Calculados os pontos, o anamorfismo faz todo o trabalho restante por nós, pegando nos pontos que criamos e transformando-os na respetiva *rose tree*.

$$
\begin{array}{l}
divide_{-}\,((((x,y),l),0)) = ((((x+(l\,/\,3),y+(l\,/\,3)),l\,/\,3),[\,]))\\
divide_{-}\,((((x,y),l),p)) = ((((x+sndLvl,y+sndLvl),sndLvl),x1 +\!\!+ x2 +\!\!+ x3))\\
\quad \textbf{where}\\
\quad sndLvl = (l\,/\,3)\\
\quad thdLvl = sndLvl * 2\\
\quad newP = p - 1\\
\quad x1 = (((x,y+thdLvl),sndLvl),newP):(((x+sndLvl,y+thdLvl),sndLvl),newP):(((x+thdLvl,y+thdLvl),sndLvl),ne\\
\quad x2 = (((x,y+sndLvl),sndLvl),newP):(((x+thdLvl,y+sndLvl),sndLvl),newP):[\,]\\
\quad x3 = (((x,y),sndLvl),newP):(((x+sndLvl,y),sndLvl),newP):(((x+thdLvl,y),sndLvl),newP):[\,]
\end{array}
$$

### E.0.6 Segundo passo - catamorfismo

O trabalho do catamorfismo *rose2List* é, como o nome indica, transformar uma *Rose tree* numa lista, contendo esta os quadrados a desenhar que se encontravam na árvore.

Partindo, novamente, de um diagrama, temos:

$$
\begin{tikzcd}
Rose((A\times B)\times C) \arrow[r, bend left, "outRose"] \arrow[d, "(|gr2l|)"'] & ((A\times B)\times C)\times(Rose(A\times B)\times C)^* \arrow[l, bend left, "inRose"] \arrow[d, "id+(|gr2l|)"] \\
((A\times B)\times C)^* & ((A\times B)\times C)\times Rose((A\times B)\times C)^* \arrow[l, "g=[\_,\_]"]
\end{tikzcd}
$$

Para formar uma lista a partir de uma cabeça e o resto do seu corpo, basta utilizarmos a função *cons*. Contudo, como temos listas de listas, temos de reduzir a complexidade desse tipo para apenas uma lista, logo:

```
g = cons . (id x concat)
```

Aqui, temos a construção de uma lista após preservar a sua cabeça e reduzir a complexidade da sua cauda.

### E.0.7 Terceiro Passo - anamorfismo *carpets*

Mais uma vez, é-nos pedida a construção de um hilomorfismo (catamorfismo após anamorfismo), desta vez para automatizar o precesso de construção dos gráficos com os quadrados de *Sierpisnki*.
Ora, o anamorfismo *carpets* recebe um inteiro e produz uma lista com listas de quadrados que definem cada um dos níveis de profundidade do tapete. Assim, conseguimos reproduzir o seguinte diagrama:

$$
\begin{tikzcd}
N \arrow[r, "outNat"] \arrow[d, "{[(carpets)]}"'] & 1+N \arrow[r, "id+<N{,}N>"] & 1+((A\times B)\times C)^*\times((A\times B)\times C)^* \arrow[d, "id+id\times{[(carpets)]}"] \\
(((A\times B)\times C)^*)^* & & 1+((A\times B)\times C)^*\times(((A\times B)\times C)^*)^* \arrow[ll, "inList"']
\end{tikzcd}
$$

Queremos, portanto, a partir de um número, criar recursivamente as listas que irão definir os *squares* de cada nível de profundidade do carpete. Como já temos uma função que faz esse trabalho, *Sierpinski*, basta atribuir-lhe um valor arbitrário:

-- then, we can use it as pointfree

$draw32\ n = sierpinski\ (((0,0),32),n)$
$carpets\_gene = (id + \langle draw32, id \rangle) \cdot outNat$

Aqui, não fazemos nada no nível de profundidade zero (preservar com identidade). Contudo, a partir de um inteiro natural processamos a respetiva lista de *squares* associado a esse nível e preservamos o mesmo valor para a próxima iteração da recursividade o voltar a processar. No fim, o construtor de listas será capaz de transformar este trabalho numa única lista com os tapetes de cada nível de profundidade.

### E.0.8 Quarto passo - catamorfismo *present*

O catamorfimos final deve processar a lista de tapetes criada anteriormente e, percorrendo cada tapete, deve desenhá-lo, respeitando um intervalo de um segundo entre cada gráfico gerado.
Assumindo que nunca vamos ter uma lista sem pontos (i.e., usamos um catamorfismo de listas não vazias), o gene deste catamorfismo tem de ser capaz de lidar com dois casos: quando a lista tem um único tapete, ou quando tem um tapete à cabeça de uma lista e os restantes tapetes na cauda da mesma. Tratando-se de uma disjunção, definimos dois casos para um gene *ppresent* que se limita a utilizar a função fornecida para desenho de carpetes:

$ppresent\ (i_1\ a) = \mathbf{do}\ drawSq\ a; await; return\ [\,]$
$ppresent\ (i_2\ (h,t)) = \mathbf{do}\ drawSq\ h; await; \mathbf{do}\ t$

## Problema 4

### Versão não probabilística

Gene de $consolidate'$:

$pairConsolidate\ (a,b)\ [\,] = [(a,b)]$
$pairConsolidate\ (a,b)\ ((h1,h2):t)$
$\quad \mid a \equiv h1 = ((h1,h2+b):t)$
$\quad \mid otherwise = (h1,h2) : pairConsolidate\ (a,b)\ t$
$pairConsolidate2\ ((a,b),[\,]) = [(a,b)]$
$pairConsolidate2\ ((a,b),((h1,h2):t))$
$\mid a \equiv h1 = (h1,h2+b):t$
$\mid otherwise = (h1,h2) : pairConsolidate2\ ((a,b),t)$

-- cgene1 = nil
-- cgene2 :: (Eq a, Num b) =¿ ((a, b), [(a, b)]) -¿ [(a, b)]
-- cgene2 = pairConsolidate2

$cgene :: (Eq\ a, Num\ b) \Rightarrow c + ((a,b),[(a,b)]) \rightarrow [(a,b)]$
$cgene = [nil, pairConsolidate2]$

Geração dos jogos da fase de grupos:

$pairupEsq\ (team,[\,]) = [\,]$
$pairupEsq\ (team,h:[\,]) = [(team,h)]$
$pairupEsq\ (team,h:t) = (team,h) : pairupEsq\ (team,t)$
$pairupGene = (id + \langle pairupEsq, \pi_2 \rangle) \cdot outList$
$pairup = concat \cdot (\!| pairupGene |\!)$
$matchResult :: (Match \rightarrow Maybe\ Team) \rightarrow Match \rightarrow [(Team, Int)]$
$matchResult\ criteria\ (m1,m2) = [(m1,r_1),(m2,r_2)]\ \mathbf{where}$
$\quad winner = criteria\ (m1,m2)$
$\quad r_1 = \mathbf{if}\ winner \equiv Nothing\ \mathbf{then}\ 0\ \mathbf{else\ if}\ winner \equiv Just\ m1\ \mathbf{then}\ 3\ \mathbf{else}\ 1$
$\quad r_2 = \mathbf{if}\ r_1 \equiv 0\ \mathbf{then}\ 0\ \mathbf{else\ if}\ r_1 \equiv 3\ \mathbf{then}\ 1\ \mathbf{else}\ 3$

-- arrangement = (¿¿= swapTeams) . chunksOf 4 where

-- swapTeams [[a1,a2],[b1,b2],[c1,c2],[d1,d2]] = [a1,b2,c1,d2,b1,a2,d1,c2]

$glt = \bot$

$gltg\ (a, [b]) = (Leaf\ a, Leaf\ b)$
$gltg\ (a, b : c : d : e) = (Fork\ (Leaf\ a, Leaf\ b), Fork\ (Leaf\ c, Leaf\ d))$

$gltgene = (id + \langle Fork \cdot (Leaf \times Leaf \cdot head), tail \cdot \pi_2 \rangle) \cdot outList$

$l2LT :: [a] \rightarrow \mathsf{LTree}\ a$
$l2LT = \bot$

$b = simulateGroupStage\ (genGroupStageMatches\ groups)$
$a = arrangement\ b$

### Versão probabilística

$pinitKnockoutStage = \bot$

$pgroupWinners :: (Match \rightarrow \mathsf{Dist}\ (Maybe\ Team)) \rightarrow [Match] \rightarrow \mathsf{Dist}\ [Team]$
$pgroupWinners = \bot$

$pmatchResult = \bot$