Cálculo de Programas
Trabalho Prático
LEI — 2022/23

Departamento de Informática
Universidade do Minho

Janeiro de 2023

Grupo nr.	36
a83798	David António Vieira dos Santos Moura Neto
a81971	Marcelo Araújo de Sousa
a83693	Rui Filipe Chaves

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:

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

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

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

$f\ 1\ 2\ 3$ 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",
```

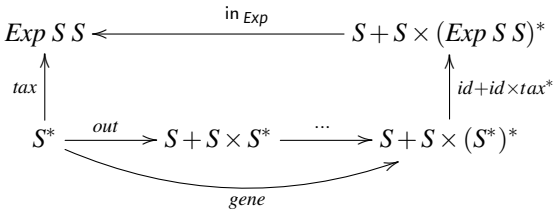
(10 primeiros ítens) etc., etc.[1]

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]:

   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]

[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.

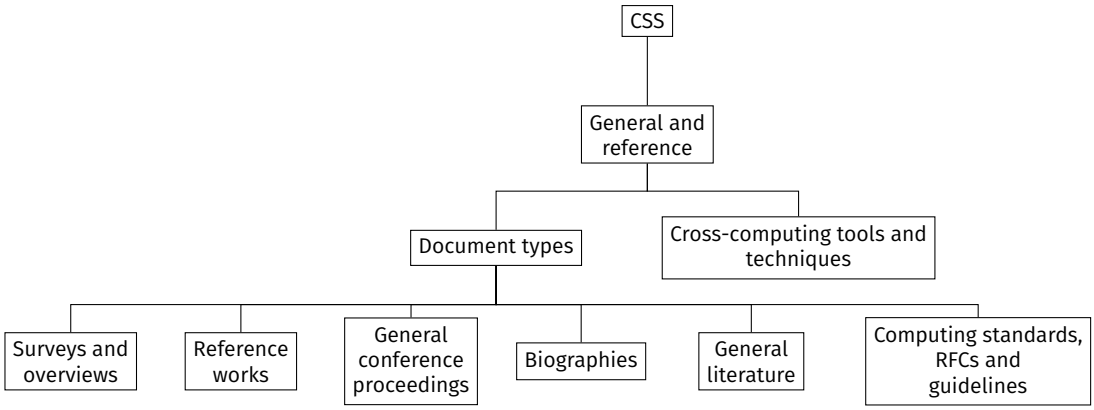


Figura 1: Fragmento de acm_ccs representado sob a forma de uma árvore do tipo Exp.

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

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

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 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. 2).

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

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

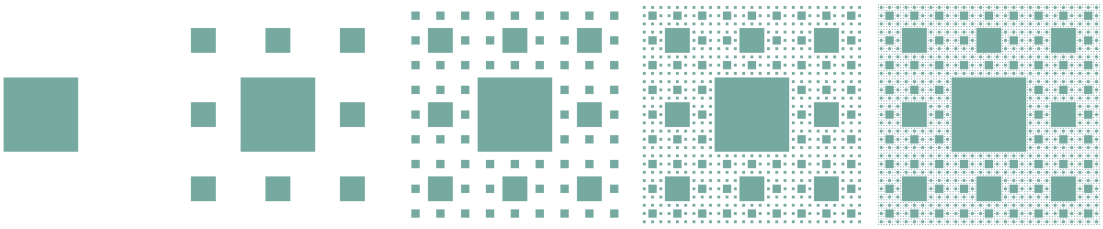


Figura 2: Construção do tapete de Sierpinski com profundidade 5.

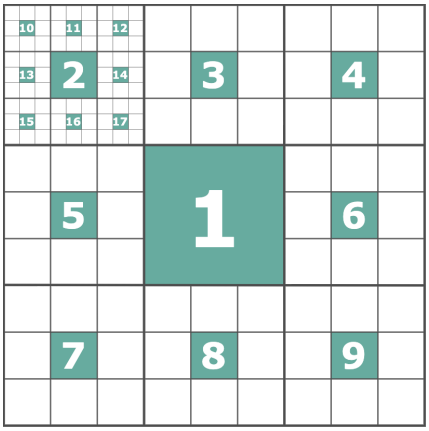


Figura 3: Tapete de Sierpinski com profundidade 2 e com os quadrados enumerados.

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

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

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

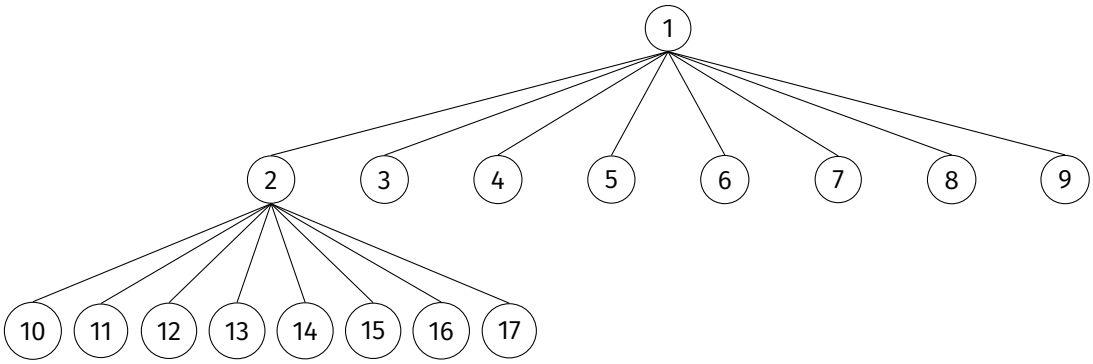


Figura 4: Possível árvore de suporte para a construção da fig. 3.

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.

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

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 D.

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

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

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

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 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:

$$\begin{aligned}
&\textbf{type } Team = String\\
&\textbf{type } Group = [Team]\\
&groups :: [Group]\\
&groups = [[\texttt{"Qatar"},\texttt{"Ecuador"},\texttt{"Senegal"},\texttt{"Netherlands"}],\\
&\quad [\texttt{"England"},\texttt{"Iran"},\texttt{"USA"},\texttt{"Wales"}],\\
&\quad [\texttt{"Argentina"},\texttt{"Saudi Arabia"},\texttt{"Mexico"},\texttt{"Poland"}],\\
&\quad [\texttt{"France"},\texttt{"Denmark"},\texttt{"Tunisia"},\texttt{"Australia"}],\\
&\quad [\texttt{"Spain"},\texttt{"Germany"},\texttt{"Japan"},\texttt{"Costa Rica"}],\\
&\quad [\texttt{"Belgium"},\texttt{"Canada"},\texttt{"Morocco"},\texttt{"Croatia"}],\\
&\quad [\texttt{"Brazil"},\texttt{"Serbia"},\texttt{"Switzerland"},\texttt{"Cameroon"}],\\
&\quad [\texttt{"Portugal"},\texttt{"Ghana"},\texttt{"Uruguay"},\texttt{"Korea Republic"}]]
\end{aligned}$$

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

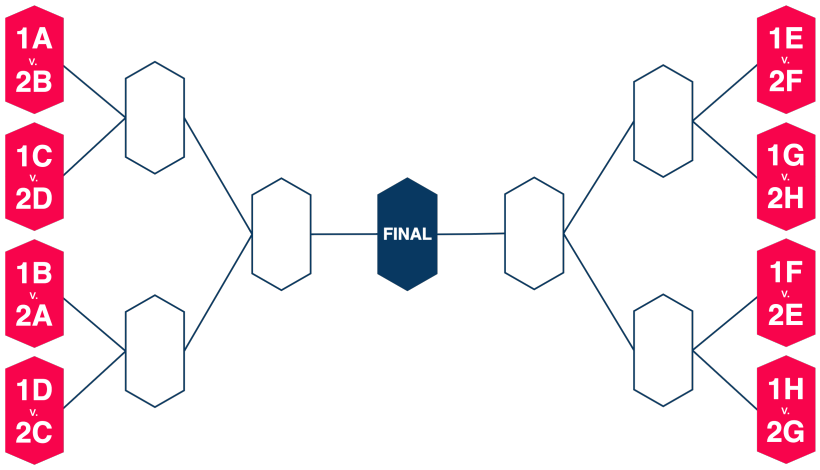


Figura 5: O “mata-mata”

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:

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

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

> $simulateGroupStage :: [[Match]] \to [[Team]]$
> $simulateGroupStage = \text{map}\ \ (groupWinners\ gsCriteria)$
>
> *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

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

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

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

etc.

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

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

Todos 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:

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

$$pwcup = pknockoutStage \bullet pgroupStage$$

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

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

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 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" [2], 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]

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

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

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

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

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

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

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

[7] Exemplos tirados de [3].
[8] Lei (3.95) em [3], página 112.

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

Obter-se-á de imediato

$$
\begin{aligned}
&fib' = \pi_1 \cdot \text{for}\ loop\ init\ \textbf{where}\\
&\quad loop\ (fib, f) = (f, fib + f)\\
&\quad init = (1,1)
\end{aligned}
$$

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:

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

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

$$
\begin{aligned}
&f'\ a\ b\ c = \pi_1 \cdot \text{for}\ loop\ init\ \textbf{where}\\
&\quad loop\ (f,k) = (f + k, k + 2 * a)\\
&\quad init = (c, a + b)
\end{aligned}
$$

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

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

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

Cada par (a,p) numa distribuição $d ::$ 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

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

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 [3] 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.

$$d_3 = normal\ [10..20]$$

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

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

D Código fornecido

Problema 1

Alguns testes para se validar a solução encontrada:

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

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 [1].

[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:

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

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":

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

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:

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

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

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

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

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

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

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

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

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

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

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

Auxiliares:

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

Função binária monádica f:

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

Monadificação de uma função binária f:

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

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

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

Na definição de f, desde logo notamos que a parte onde está $(n+3)$, assim como $(n+2)$ será um problema, pois não permitirá a definição da função como um catamorfirmo de Naturais, logo também não poderá ser definida como um ciclo for (já que este é um catamorfismo). Iremos então precisar de definir funções auxiliares, usando as leis da recursividade mútua, até chegarmos a definições de funções onde apenas se encontrem casos até $(n+1)$.

$$Let$$
$$f\ a\ b\ c\ (n+3) = f1\ a\ b\ c\ (n+1), logo$$
$$f\ a\ b\ c\ (n+2) = f1\ a\ b\ c\ n$$

Então:

$$f1\ a\ b\ c\ 0 = f\ a\ b\ c\ (0+2) = 0$$
$$f1\ a\ b\ c\ (n+1) = f\ a\ b\ c\ (n+3) = a * f\ a\ b\ c\ (n+2) + b * f\ a\ b\ c\ (n+1) + c * f\ a\ b\ c\ n$$

Sabemos que $f\ a\ b\ c\ (n+2) = f1\ a\ b\ c\ n$, então

$$f1\ a\ b\ c\ 0 = f\ a\ b\ c\ (0+2) = 0$$
$$f1\ a\ b\ c\ (n+1) = f\ a\ b\ c\ (n+3) = a * f1\ a\ b\ c\ n + b * f\ a\ b\ c\ (n+1) + c * f\ a\ b\ c\ n$$

Temos agora que definir mais uma função para eliminar o problema de $f\ a\ b\ c\ (n+1)$

$$Let$$
$$f\ a\ b\ c\ (n+1) = f2\ a\ b\ c\ n$$

Então:

$$f2\ a\ b\ c\ 0 = f\ a\ b\ c\ (0+1) = 1$$
$$f2\ a\ b\ c\ (n+1) = f\ a\ b\ c\ (n+2) = f1\ a\ b\ c\ n$$

Agora podemos renomear e simplificar a função original para apenas 2 cláusulas, porque f a b c (n + 1) = f2 a b c n, e depois mudamos o nome da função:

$$fnew\ a\ b\ c\ 0 = 0$$
$$fnew\ a\ b\ c\ (n+1) = f2\ a\ b\ c\ n$$

Funções auxiliares pedidas:

$$fnew\ a\ b\ c\ 0 = 0$$
$$fnew\ a\ b\ c\ (n+1) = f2\ a\ b\ c\ n$$
$$f1\ a\ b\ c\ 0 = 1$$
$$f1\ a\ b\ c\ (n+1) = a * f1\ a\ b\ c\ n + b * f2\ a\ b\ c\ n + c * fnew\ a\ b\ c\ n$$

$f2\ a\ b\ c\ 0 = 1$
$f2\ a\ b\ c\ (n+1) = f1\ a\ b\ c\ n$

Obtemos então 3 funções mutuamente recursivas, sendo a função *fnew* a que guarda o resultado. Aplicando então a regra prática do anexo B, chegamos de forma trivial à definição do *loop* e do *init* para chegar à definição de *fbl* (função matematicamente igual a f, mas mais eficiente)

$loop\ a\ b\ c\ ((f2,f1),fnew) = ((f1, a*f1+b*f2+c*fnew),f2)$
$initial = ((1,1),0)$
$wrap = \pi_2$

De seguida encontra-se a tabela 2 com os testes de performance, comparando os tempos de execução da função f e da função *fbl* (valores dos argumentos $a\ b\ c$ não influênciam os tempos de execução, logo a tabela pode ser considerada genérica para qualquer valor que estes possam tomar)

valor de n	tempo de execução de f (s)	tempo de execução de *fbl* (s)	tempo f / tempo *fbl*
5	0.01	0.01	1
10	0.01	0.01	1
15	0.01	0.01	1
20	0.08	0.01	8
25	1.42	0.01	142
30	29.63	0.01	2963

Tabela 2: Testes de Permormance.

Nota-se de imediato que no início não existe diferença, mas quando n começa a ter valores mais elevados a função *fbl* é muito mais rápida que f. Sendo que a razão entre os tempos de ambos vai aumentando exponencialmente. Isto ocorre devido ao facto da função f, fazer cálculos repetidos desnecessariamente por usar sempre a recursividade sobre elementos que já foram calculados antes, sendo um desperdício de tempo. Já a função *fbl*, por ser definida como um ciclo for, nunca faz cálculos repetidos, vai sempre guardando os resultados já calculados, que serão futuramente necessários para o cálculo dos elementos seguintes.

Problema 2

Gene de *tax*:

$contaespacos :: Num\ p \Rightarrow String \to p$
$contaespacos\ [\,] = 0$
$contaespacos\ (h:t) = \textbf{if}\ h \equiv \text{' '}\ \textbf{then}\ 1 + contaespacos\ t\ \textbf{else}\ 0$

$insertSeparators :: [String] \to [String]$
$insertSeparators\ [\,] = [\,]$
$insertSeparators\ [h] = \textbf{if}\ (contaespacos\ h \equiv 0)\ \textbf{then}\ h : [\texttt{"\$\$\$\$\$"}]\ \textbf{else}\ [h]$
$insertSeparators\ (h:h':t) = \textbf{if}\ (contaespacos\ h' \equiv 0)\ \textbf{then}\ h : \texttt{"\$\$\$\$\$"} : h' : insertSeparators\ (h':t)$
$\textbf{else}\ h : insertSeparators\ (h':t)$

$remove :: String \to [String] \to [String]$
$remove\ x\ [\,] = [\,]$
$remove\ x\ (h:t) = \textbf{if}\ (x \equiv h \wedge x\ != \texttt{"\$\$\$\$\$"})\ \textbf{then}\ remove\ h\ t\ \textbf{else}\ h : (remove\ h\ t)$

$remDup :: [String] \to [String]$
$remDup\ [\,] = [\,]$
$remDup\ (h:t) = h : (remDup\ (remove\ h\ t))$

$outList1\ [a] = i_1\ (a)$
$outList1\ (a:x) = i_2\ (a,x)$

$gene4 = id + id \times f\ \textbf{where}$
$f :: [String] \to [[String]]$
$f\ l = filter\ (!=[\texttt{""}])\ ((splitWhen\ (\equiv \texttt{"\$\$\$\$\$"})\ (remDup\ (insertSeparators\ (\text{map}\ \ (drop\ 4)\ l)))))$

$gene = gene4 \cdot outList1$

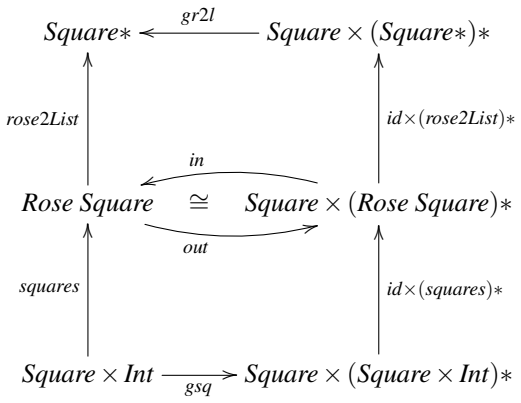
$$removeEspacos :: String \to String$$
$$removeEspacos\ [h] = [h]$$
$$removeEspacos\ (h:t) = \textbf{if}\ (h \equiv \text{' '})\ \textbf{then}\ removeEspacos\ (t)\ \textbf{else}\ h:t$$

Função de pós-processamento:

$$post = \perp$$

Problema 3

Diagrama do Hilomorfismo:



Para chegar ao gene do anamorfismo, temos de perceber como cada quadrado é desenhado em função do nível de profundidade em causa. No caso da profundidade ser 0 temos um quadrado central com um terço do comprimento de lado do quadrado representativo do tapete, logo serão essas as medidas do primeiro quadrado. E como este se encontra centrado, temos de deslocar as coordenadas referentes ao vértice inferior esquerdo também um terço do lado quadrado original (tapete),a lista dos próximos quadrados será obviamente vazia, por ser único quadrado e caso de paragem.

Agora, no caso da profundidade ser $n+1$, podemos pensar para o caso de profundidade 1 inicialmente para facilitar a perceção. Então o quadrado central vai ser construído da mesma forma que no caso 0, os outros 8 quadrados mais pequenos, têm que ser vistos como tapetes (que são quadrados com três vezes o lado desse quadrado mais pequeno,contendo-o no centro), assim repara-se que as coordenadas de x vão variar entre ($x, x+side\ /\ 3\ e\ x+2*side\ /\ 3$) e as coordenadas de y vão também variar entre ($y, y+side\ /\ 3\ e\ y+2*side\ /\ 3$), depois é só fazer todas as combinações possíveis(excluindo o caso correspondente ao quadrado centrar).

Depois observamos que os lados vão medir mais uma vez um terço do tapete original (mais uma vez estamos a olhar para cada quadrado incluíndo a parte branca como se fosse um sub-tapete). Por fim cada um desses sub-tapetes vai ficar emparelhado com n, para que na parte recurssiva do anamorfismo agora sim o quadrado central meça um terço desse subtapete. Podemos agora perceber que de forma genérica isto vai funcionar com qualquer profundidade. Pois trata-se de repetir o processo até chegar ao caso de paragem.

De certa forma o que o algoritmo faz é "olhar"para o tapete fazer o quadrado central, e chamar-se oito vezes de forma recursiva para cada oito sub-tapetes baixando a profundidade em cada um, e sempre assim até chegar à profundidade 0 e fazer apenas o tapete central.

$$squares = [\![\ gsq\ ]\!]_R$$
$$gsq :: (Square, Int) \to ((Square), [(Square, Int)])$$
$$gsq\ (((x,y), side), 0) = (((x+side\ /\ 3, y+side\ /\ 3), side\ /\ 3), [])$$
$$gsq\ (((x,y), side), n+1) = (((x+side\ /\ 3, y+side\ /\ 3), side\ /\ 3), [(((x,y), side\ /\ 3), n),$$
$$(((x+side\ /\ 3, y), side\ /\ 3), n), (((x+2*side\ /\ 3, y), side\ /\ 3), n), (((x, y+side\ /\ 3), side\ /\ 3), n),$$
$$(((x+2*side\ /\ 3, y+side\ /\ 3), side\ /\ 3), n), (((x, y+2*side\ /\ 3), side\ /\ 3), n),$$
$$(((x+side\ /\ 3, y+2*side\ /\ 3), side\ /\ 3), n), (((x+2*side\ /\ 3, y+2*side\ /\ 3), side\ /\ 3), n)])$$

O gene do catamorfismo *rose2List* descobre-se de forma muito trivial. Pois como partimos de um par *Squares*, lista de lista de *Squares*, e pretendemos juntar todos estes quadrados em apenas uma lista basta colocar o primeiro elemento do par à cabeça do concat da lista de lista de *Squares*,obtendo-se então a lista dos quadrados todos.

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

Para fazer a função *carpets*, que constrói todos os tapetes de profundidade 1..n, basta para o caso de profundidade 0 criar uma lista de listas apenas com a lista [(0,0),32], que se trata do quadrado base central no tapete de profundidade 0. Quando se trata de $n+1$, basta chamar a função *sierpinski* usando como argumentos o mesmo quadrado central e $n+1$ de profundidade, obtendo-se uma lista com todos os quadrados dessa profundidade. Depois coloca-se essa lista dentro de outra lista e junta-se com a lista que resulta da chamada recursiva de *carpets n*.

$$
\begin{aligned}
&carpets :: Int \to [[Square]] \\
&carpets\ 0 = [[((0,0),32)]] \\
&carpets\ (n+1) = [sierpinski\ (((0,0),32),n+1)] +\!\!+ carpets\ n
\end{aligned}
$$

Inicialmente, para facilitar, começamos por criar uma função que recebe uma Carpete (Lista de *Squares*) e apresenta a imagem num ficheiro html (A função *present1Carpet*). Para isso basta recorrer à função *drawSq* utilizando como argumento o quadrado central base ((0,0),32), mas falta ainda saber a profundidade a que corrresponde essa Carpete. Foi então necessário criar outra função auxiliar que em função do nº de quadrados na lista vai indicar a profundidade respetiva (a função *seqFormula*, que é simplesmente uma fórmula matemática para esta sequência), então passamos mais o argumento *seqFormula* da *length* da lista à função *drawSq*, obtendo a profundidade respetiva. Assim a função *present1Carpet* já apresenta o ficheiro html recebendo apenas auma lista de quadrados.

Para apresentar as imagens de todas as Carpetes até uma certa profundidade basta primeiro correr a função *carpets*, para se obter a lista de Carpetes e utilizar essa lista como argumento na função *present*, que vai aplicar o monadic map *presente1Carpet* a todas as carpetes (mas só depois de fazer reverse à lista, para os ficheiros surgirem por ordem crescente de profundidade)

$$
\begin{aligned}
&seqFormula\ x = fromIntegral\ (round\ ((logBase\ 8\ (7*x+1))-1)) \\
&present1Carpet\ (h:t) = \mathbf{do} \\
&\quad x \leftarrow drawSq\ (sierpinski\ (((0,0),32),(seqFormula\ (fromIntegral\ (length\ (h:t))))))) \\
&\quad await \\
&present\ llSquares = mapM\ present1Carpet\ (reverse\ llSquares)
\end{aligned}
$$

Problema 4

Versão não probabilística

Diagrama do Catamorfismo:

$$
\begin{array}{ccc}
(Team \times Int)* & \overset{in}{\underset{out}{\cong}} & 1 + (Team \times Int) \times (Team \times Int)* \\
\Big\downarrow{\scriptstyle consolidate} & & \Big\downarrow{\scriptstyle id+id\times consolidate} \\
(Team \times Int)* & \xleftarrow[cgene]{} & 1 + (Team \times Int) \times (Team \times Int)*
\end{array}
$$

A função *consolidade* é um catamorfismo, que recebe uma lista de pares $Team \times Int$, sendo que cada par especifica os pontos que a respetiva equipa obteve num jogo, como a lista reflete todos os pontos de todas os jogos de todas as equipas de um grupo, vai obrigatoriamente ter equipas repetidas, com outros pontos associados a outros jogos. O objetivo da função é devolver a lista já com os pontos

todos que a equipa obteve acumulados num só par, para todas as equipas, sendo agora uma lista de 4 elementos.

Inicialmente foi feita uma função auxiliar *consolAux*, que recebendo um par $Team \times Int$ e a lista de todos os pares $Team \times Int$, vai juntar a sua pontuação com a do primeiro elemento da lista com a mesma equipa e mantém o resto da lista inalterada.

Agora para o gene em si, repara-se que será um either de 1 ou $(Team \times Int) \times (Team \times Int)*$. No lado esquerdo apenas se retorna a lista vazia, e no direito caso a lista seja vazia, apenas se faz singl ao par, para o transformar numa lista, se a lista não for vazia basta chamar a função auxiliar definida para agrupar os pontos da equipa na lista com todos os pontos de todas as equipas, uma vez que podemos pensar que a parte recursiva da lista já foi feita,estando já os pontos agrupados pelas equipas.

Gene de $consolidate'$:

$$
\begin{aligned}
&cgene :: (Eq\ a, Num\ b) \Rightarrow () + ((a,b),[(a,b)]) \rightarrow [(a,b)]\\
&cgene = [g1, g2]\ \textbf{where}\\
&\quad g1\ () = [\,]\\
&\quad g2\ (a,[\,]) = [a]\\
&\quad g2\ (a,h:t) = consolAux\ a\ (h:t)\\
&consolAux\ x\ [\,] = [x]\\
&consolAux\ x\ (h:t) = \textbf{if}\ \pi_1\ x \equiv \pi_1\ h\ \textbf{then}\ ((\pi_1\ x, \pi_2\ x + \pi_2\ h) : t)\\
&\quad \textbf{else}\ h : consolAux\ x\ t
\end{aligned}
$$

Primeiramente para gerar uma lista com os jogos, foi feita a função auxiliar *pairUpAux*. Esta função recebendo a lista das equipas vai criar a lista de todos os jogos que a equipa que se encontra à cabeça vai disputar, retornando assim uma lista de pares. Na função final *pairup* basta chamar a função auxiliar com a lista toda como argumento e concatenar com a chamada recursiva da função pairup para a cauda.

Geração dos jogos da fase de grupos:

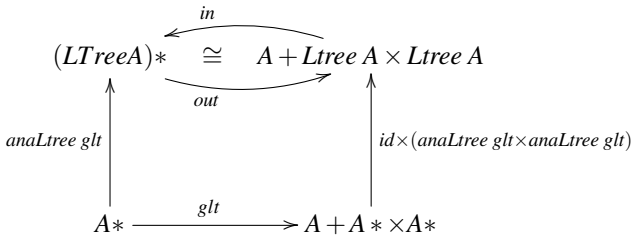
$$
\begin{aligned}
&pairup\ [\,] = [\,]\\
&pairup\ (h:t) = pairupAux\ (h:t) +\!\!+ pairup\ t\\
&pairupAux\ [\,] = [\,]\\
&pairupAux\ [x] = [\,]\\
&pairupAux\ (x:y:xs) = (x,y) : pairupAux\ (x:xs)
\end{aligned}
$$

A função matchResult define-se de forma muito trivial, bastando ir por casos e devolver a pontuação devida para ambas as equipas.

Definição da função MatchResult

$$
\begin{aligned}
&matchResult :: (Match \rightarrow Maybe\ Team) \rightarrow Match \rightarrow [(Team, Int)]\\
&matchResult\ f\ (t1,t2) = \textbf{if}\ f\ (t1,t2) \equiv Nothing\ \textbf{then}\ [(t1,1),(t2,1)]\\
&\quad \textbf{else if}\ f\ (t1,t2) \equiv Just\ t1\ \textbf{then}\ [(t1,3),(t2,0)]\ \textbf{else}\ [(t1,0),(t2,3)]
\end{aligned}
$$

Diagrama do Anamorfismo:



Para definir a função $(\!|\ glt\ |\!)$ que recebe a lista das Equipas apuradas e coloca-as numa Ltree, apenas será necessário definir o gene *glt*. A ideia inicial que surge é de dividir a lista em duas partes iguais (ou uma com mais um elemento que a outra caso o nº de elementos da lista seja ímpar), se a lista só contiver um elemento então basta injeta-lo à esquerda, se tiver mais então injeta-se do lado direito

o par com a lista partida em duas. Para obter apenas a primeira metade da lista foi definida a função auxiliar *glttake* (que fica com metade dos elementos arredondado para baixo) e a função *gltdrop*(que fica com metade dos elementos arredondado para cima).

Definição do gene glt

$$
\begin{aligned}
&glttake\ [x] = [x]\\
&glttake\ (h:t) = take\ length\ (h:t) \div 2\ (h:t)\\
&gltdrop\ [x] = [x]\\
&gltdrop\ (h:t) = drop\ length\ (h:t) \div 2\ (h:t)\\
&glt :: [a_1] \to a_1 + ([a_1],[a_1])\\
&glt\ [x] = i_1\ x\\
&glt\ (h:t) = i_2\ (glttake\ (h:t), gltdrop\ (h:t))
\end{aligned}
$$

Versão probabilística

$$
\begin{aligned}
&pinitKnockoutStage = \bot\\
&pgroupWinners :: (Match \to \mathsf{Dist}\ (Maybe\ Team)) \to [Match] \to \mathsf{Dist}\ [Team]\\
&pgroupWinners = \bot
\end{aligned}
$$

Inicialmente definimos a função auxiliar probAux, que recebendo uma função do tipo $(Match \to \mathsf{Dist}\ (Maybe\ Team))$,que será sempre a função *pgsCriteria*, e um jogo vai retornar a lista apenas com as probabilidades. Uma vez que a apresentação das probabilidades segue sempre a mesma ordem, primeiro a probabilidade correspondente ao empate (caso a lista tenha 3 elementos), depois a probabilidade de vitória da equipa do lado esquerdo do par e por última a probabilidade de vitória da equipa da esquerda, podemos explorar esta propriedade mais tarde.

Tendo isso em conta definimos uma outra função simples que retorna o n-ésimo elemento de uma lista *listNelem*. Agora com estas funções já podemos definir a função pmatchResult, que vai receber o jogo em causa, e com base no nº de elementos da lista da função auxiliar vai criar distribuições para cada equipa, sabendo sempre a qual associar devido à propriedade mostramos.

$$
\begin{aligned}
&pmatchResult :: Match \to \mathsf{Dist}\ [(Team, ProbRep)]\\
&pmatchResult\ (t1,t2) = \textbf{if}\ (length\ (probAux\ pgsCriteria\ (t1,t2)) \equiv 3)\ \textbf{then}\\
&D\ [([(t1,1),(t2,1)], listNelem\ 1\ (probAux\ pgsCriteria\ (t1,t2))),\\
&([(t1,3),(t2,0)], listNelem\ 2\ (probAux\ pgsCriteria\ (t1,t2))),\\
&([(t1,0),(t2,3)], listNelem\ 3\ (probAux\ pgsCriteria\ (t1,t2)))]\ \textbf{else}\\
&D\ [([(t1,3),(t2,0)], listNelem\ 1\ (probAux\ pgsCriteria\ (t1,t2))),\\
&([(t1,0),(t2,3)], listNelem\ 2\ (probAux\ pgsCriteria\ (t1,t2)))]\\
&probAux :: (Match \to \mathsf{Dist}\ (Maybe\ Team)) \to Match \to [ProbRep]\\
&probAux\ f\ (a,b) = \mathsf{map}\ \ \pi_2\ (unD\ (f\ (a,b)))\\
&listNelem :: (Integral\ a_1) \Rightarrow a_1 \to [ProbRep] \to ProbRep\\
&listNelem\ 1\ (h:t) = h\\
&listNelem\ (n+1)\ (h:t) = listNelem\ n\ t
\end{aligned}
$$

Índice

Referências

[1] M. Erwig and S. Kollmansberger. Functional pearls: Probabilistic functional programming in Haskell. *J. Funct. Program.*, 16:21–34, January 2006.

[2] D.E. Knuth. *Literate Programming*. CSLI Lecture Notes Number 27. Stanford University Center for the Study of Language and Information, Stanford, CA, USA, 1992.

[3] J.N. Oliveira. Program Design by Calculation, 2022. Textbook in preparation, 310 pages. Informatics Department, University of Minho. Current version: Sept. 2022.