Cálculo de Programas
Trabalho Prático
LEI — 2022/23

Departamento de Informática
Universidade do Minho

Janeiro de 2023

Grupo nr.	12
a94913	José Manuel Antunes de Carvalho
a97031	Miguel Filipe Cidade da Silva
a96351	Luis Alberto Barreiro Araújo
a92926	Marisa Ferreira Soares

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",
```

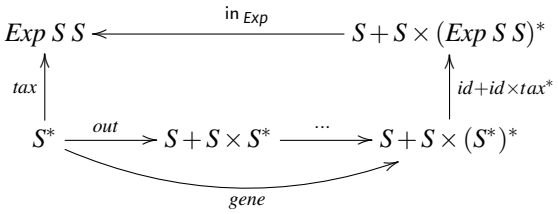
(10 primeiros ítens) etc., etc.[1]

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.

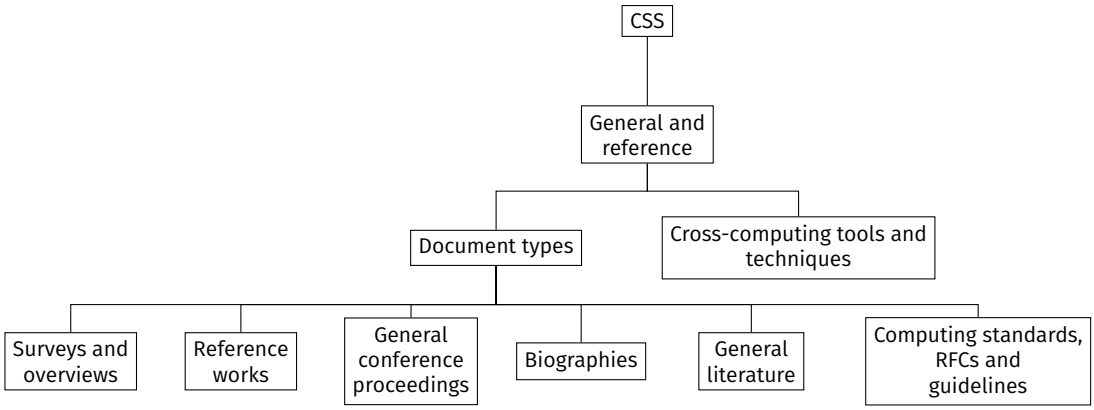


Figura 1: Fragmento de acm_ccs representado sob a forma de uma árvore do tipo Exp.

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.

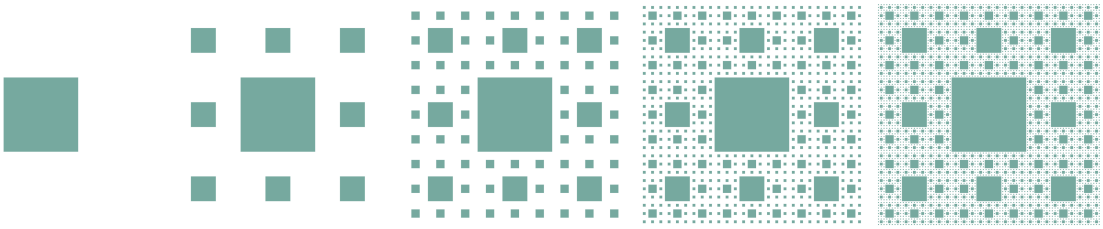


Figura 2: Construção do tapete de Sierpinski com profundidade 5.

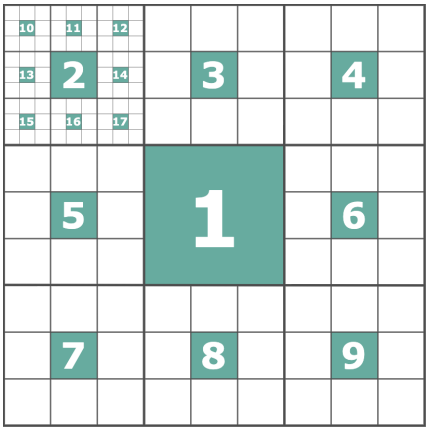


Figura 3: Tapete de Sierpinski com profundidade 2 e com os quadrados enumerados.

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.

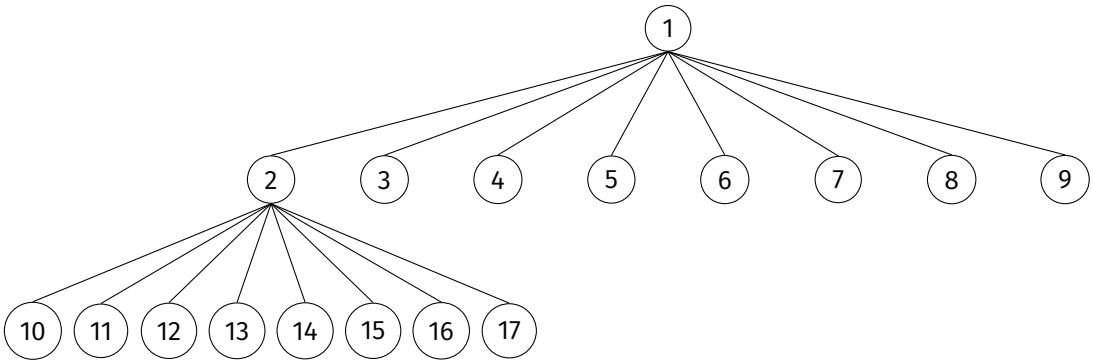


Figura 4: Possível árvore de suporte para a construção da fig. 3.

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

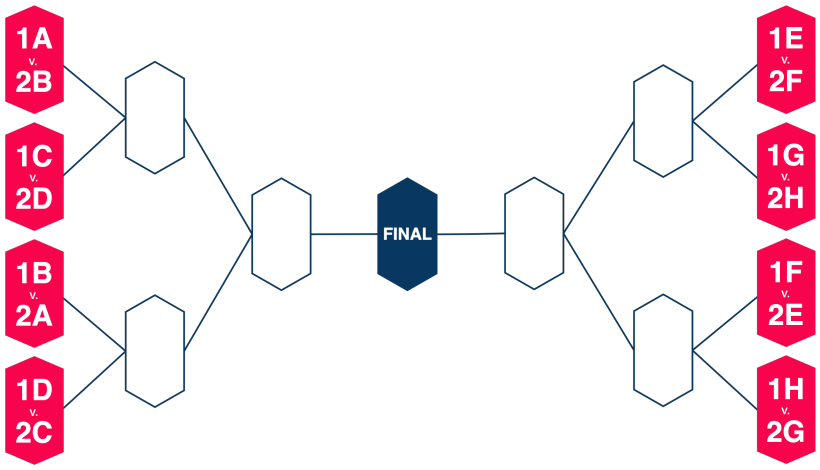


Figura 5: O “mata-mata”

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

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

Obter-se-á de imediato

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

usando as regras seguintes:

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

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

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

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

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

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

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.

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

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

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

D Código fornecido

Problema 1

Alguns testes para se validar a solução encontrada:

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

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 [**?**].

[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",4.0),
  ("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

Após analisar o contexto do problema apresentado, concluímos que se trata de um problema onde iremos recorrer à lei da recursividade mútua. É nos apresentado a seguinte função, derivada da sequência de Fibonacci usando um ciclo-for, tal que cada termo subsequente aos três primeiros corresponde à soma dos três anteriores:

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

O problema pede então que seja aplicada a seguinte regra:

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

Onde iremos obter o seguinte:

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

Deste modo podemos concluir o seguinte:

- O ciclo loop irá ter tantos argumentos quanto o número de funções mutuamente exclusivas. Neste caso serão 3 argumentos, pois temos 3 termos inciais.
- As funções serão respetivamente, g,h e f.
- Em init usam se os resultados dos casos base respetivamente, ((1,1),0).

E é nos apresentado o que devemos resolver:

$$fbl\ a\ b\ c = wrap\ \text{for}\ loop\ a\ b\ c\ initial$$

As funções f, h e g que levantamos foram as seguintes:

$$g\ 0 = 1$$
$$g\ (n+1) = c_1\ ((g\ n, h\ n), f\ n)$$
$$h\ 0 = 1$$

$$h\ (n+1) = g\ n$$
$$h\ (n+1) = c_2\ ((g\ n, h\ n), f\ n)$$
$$f\ 0 = 0$$
$$f\ (n+1) = h\ n$$
$$f\ (n+1) = c_3\ ((g\ n, h\ n), f\ n)$$

Como h (n+1) = g (n), c2 = (p1.p1) e como f(n+1) = h(n), c3 = (p2.p1).

Destas nossas funções a f é a função principal, sendo que esta pode ser traduzida para a seguinte forma:

$$f\ 0 = 0$$
$$f\ n = (\pi_2 \cdot \pi_1)\ ((f\ (n+1), f\ n), f\ (n-1))$$

Ou seja, em cada instante temos o resultado da função relativamente à etapa anterior, à etapa atual e à etapa seguinte. É desta forma que evitamos calcular novamente as mesmas coisas.

Casos base:

- f(0) = 0
- f(1) = (p2.p1) ((g 0, h 0), f 0) = (p2.p1) ((1,1),0) = 1
- f(2) = (p2.p1) ((g 1, h 1), f 1) = (p2.p1) ((?,g 0), h 0) = (p2.p1) ((?,1),1) = 1

Agora falta-nos apenas definir c1.

Partindo da definição da função, conseguimos reparar que ela utiliza o resultado das 3 etapas anteriores, Ora bem nós de certa forma já temos isso, como demonstado no seguinte esquema:

$$g\ (n+1) = c1\ ((g\ n,\ h\ n),\ f\ n)$$
$$\equiv \quad \{ \text{ subsituindo n+1 por n+3 } \}$$
$$g\ (n+3) = c1\ ((g\ (n+2),\ h\ (n+2)),\ f\ (n+2))$$
$$\equiv \quad \{ \text{ subsituindo f e h por g } \}$$
$$g\ (n+3) = c1\ ((g\ (n+2),\ g\ (n+1)),\ g\ (n))$$

Ou seja c1, acaba por ser uma simples função visto que nós já temos esses tais resultados. Para isso basta aplicar as operações aritméticas aos resultados já obtidos.

Partindo princípio que queremos manter os resultados de (g (n+2), g (n+1)), esses elementos aplicamos simplesmente umas funções de translação (c2 e c3) para a direita. O resultado de g(n) é apenas utilizado para o calculo de g (n+3) e posteriormente é deitado "fora".

Como a, b e c são parâmetros da função, a definição de c1 é:

$$add \cdot (add \times id) \cdot (((a*) \times (b*)) \times (c*))$$

Sendo assim o nosso sistema de equações ficou:

$$\begin{cases} g\ 0 = 1 \\ g\ (n+1) = (add \cdot (add \times id) \cdot (((a*) \times (b*)) \times (c*)))\ ((g\ n, h\ n), f\ n) \end{cases}$$

$$\begin{cases} h\ 0 = 1 \\ h\ (n+1) = (\pi_1 \cdot \pi_1)\ ((g\ n, h\ n), f\ n) \end{cases} \quad \begin{cases} f\ 0 = 1 \\ f\ (n+1) = (\pi_2 \cdot \pi_1)\ ((g\ n, h\ n), f\ n) \end{cases}$$

Primeiramente vamos multiplicar a, b e c pelo respetivo resultado, e depois posteriormente só são aplicadas adições dos resultados das tais multilicações.

Com isto, concluimos que a variável "initial", que vai ser definida por ((1,1),0), porque são os valores dos resultados iniciais dos coeficientes da sequência númerica, definidos no enunciado, que correspondem às funções também definidas inicialmente ((g,h),f).

$$initial = ((1,1),0)$$

De seguida definimos a função "wrap", que tem como funcionalidade ir buscar o segundo par de **for (loop a b c) initial**:

$$wrap = \pi_2$$

Por fim deduzimos a função "loop"simplesmente juntando c1, c2 e c3 numa única operação:

$$loop\ a\ b\ c = \langle\langle add \cdot (add \times id) \cdot (((a*) \times (b*)) \times (c*)), \pi_1 \cdot \pi_1\rangle, \pi_2 \cdot \pi_1\rangle$$

$$f\ a\ b\ c = \langle\langle add \cdot (add \times id) \cdot (((a*) \times (b*)) \times (c*)), \pi_1 \cdot \pi_1\rangle, \pi_2 \cdot \pi_1\rangle$$

Definição do loop, point-free (chegamos à conclusão que é igual a f):

$$loop\ a\ b\ c = \langle\langle add \cdot (add \times id) \cdot (((a*) \times (b*)) \times (c*)), \pi_1 \cdot \pi_1\rangle, \pi_2 \cdot \pi_1\rangle$$

Valorização

$testa\ a\ b\ c = \text{map}\ \ (fbl\ a\ b\ c)\ x$ **where** $x = [1..20]$
$testb\ a\ b\ c = \text{map}\ \ (f\ a\ b\ c)\ x$ **where** $x = [1..20]$
$test11a = testa\ 1\ 2\ 3$
$test11b = testb\ 1\ 2\ 3$
$test22a = testa\ (-2)\ 1\ 5$
$test22b = testb\ (-2)\ 1\ 5$

$*Main> :set +s$
$*Main> test11a$
$[1,1,3,8,17,42,100,235,561,1331,3158,7503,17812,42292,100425,238445,566171,1344336,3192013,7579198]$
$(0.07\ secs, 472,216\ bytes)$
$*Main> test11b$
$[1,1,3,8,17,42,100,235,561,1331,3158,7503,17812,42292,100425,238445,566171,1344336,3192013,7579198]$
$(0.29\ secs, 152,214,104\ bytes)$
$*Main> test22a$
$[1,1,-1,8,-12,27,-26,19,71,-253,672,-1242,1891,-1664,-991,9773,-28857,62532,-105056,128359]$
$(0.07\ secs, 466,936\ bytes)$
$*Main> test22b$
$[1,1,-1,8,-12,27,-26,19,71,-253,672,-1242,1891,-1664,-991,9773,-28857,62532,-105056,128359]$
$(0.28\ secs, 152,580,536\ bytes)$

Assim é possível verificar que a função **fbl** é 4 vezes mais eficiente que a **f**, em termos de tempo e (+-)300 vezes mais eficiente em termos de memória usada.

Problema 2

Gene de *tax*:

A função out pode-se retirar do diagrama apresentado no enunciado. Basicamente se a lista tiver apenas um elemento, injetamo-lo à esquerda. Se a lista tiver uma cabeça e uma cauda, injetamos à direita o par (cabeça,[cauda]).

$outP2 :: [a] \to a + (a,[a])$
$outP2\ [x] = i_1\ x$
$outP2\ (h:t) = i_2\ (h,t)$

A função contaEspaços é apenas uma função auxiliar que conta o numero de espaços iniciais de uma String, que será aplicada à função que trata as diretorias e subdiretorias.

$$
\begin{aligned}
&contaEspacos :: String \to \mathbb{Z}\\
&contaEspacos\ "" = 0\\
&contaEspacos\ (' \ ' : t) = 1 + contaEspacos\ t\\
&contaEspacos\ \_ = 0
\end{aligned}
$$

Função auxiliar que apenas adiciona um elemento ao final de uma determinada matriz.

$$
\begin{aligned}
&addLast :: (String, \mathbb{Z}) \to [[(String, \mathbb{Z})]] \to [[(String, \mathbb{Z})]]\\
&addLast\ t\ l = (take\ (length\ l - 1)\ l) +\!\!+ [last\ (l) +\!\!+ [t]]
\end{aligned}
$$

A função a seguir recebe uma matriz de tuplos, que correspondem às diretorias e os respetivos número de espaços e um tuplo a ser inserido nessa mesma matriz. Ou seja a função vai determinar onde é inserido esse tuplo na matriz. Caso o número de espaços da cabeça do última lista da matriz for menor que o numero de espaços do elemento a inserir, então quer dizer que o tuplo (s,n) é subdiretoria desse elemento e adicionamos ao fim da lista com a função addLast acima descrita.

$$
\begin{aligned}
&funAux :: [[(String, \mathbb{Z})]] \to (String, \mathbb{Z}) \to [[(String, \mathbb{Z})]]\\
&funAux\ [\,]\ x = [[x]]\\
&funAux\ l\ (s,n) \mid \pi_2\ (head\ (last\ (l))) < n = addLast\ (s,n)\ l\\
&\quad \mid otherwise = l +\!\!+ [[(s,n)]]
\end{aligned}
$$

A seguir está um exemplo para melhor compreensão

Se passarmos a matriz como argumento

```
[
  [
    ("General and References",0),("    Document Types",4),
    ("        Surveys and overviews",8),("        Reference works",8)
  ]
]
```

e o par

```
("        General conference proceedings",8)
```

o resultado final será:

```
[
  [
    ("General and References",0),("    Document Types",4),
    ("        Surveys and overviews",8),("        Reference works",8),
    ("        General conference proceedings",8)
  ]
]
```

Outro exemplo:

A matriz fornecida é:

```
[
  [
    ("Document Types",0),("    Surveys and overviews",4),
    ("    Reference works",4),("    General conference proceedings",4)
  ]
]
```

e o par é:

```
("Cross-computing tools and techniques",0)
```

Resultado:

```
[
  [
    ("Document Types",0),("    Surveys and overviews",4),
    ("    Reference works",4),("General conference proceedings",4)
  ],
  [
    ("Cross-computing tools and techniques",0)
  ]
]
```

Os resultados dos seguintes testes estão corretos.

$teste21 = expDepth\ acm\_tree \equiv 7$
$teste22 = length\ (expOps\ acm\_tree) \equiv 432$
$teste23 = length\ (expLeaves\ acm\_tree) \equiv 1682$

A função fun pega numa lista de strings e passa para uma matriz de Strings de acordo com as subdiretorias, isto é, cada lista de strings corresponde a uma diretoria. Uma diretoria é definida de acordo com o número de espaços e para isso primeiro contamos o número de espaços de 4 em 4 para criar as respetivas listas de strings. No fim apagamos os espaços que delimitavam as subdiretorias com a função drop.

$fun :: [String] \to [[String]]$
$fun = \mathsf{map}\ \ (\mathsf{map}\ \ (\lambda(s,n) \to drop\ 4\ s)) \cdot foldl\ funAux\ [\,] \cdot \mathsf{map}\ \ (\lambda s \to (s, contaEspacos\ s))$

O gene é facilmente retirado do diagrama apresentado no enunciado.

$$\begin{array}{ccc} S^* & \xrightarrow{out} & S + S \times S^* \\ & \searrow^{gene} & \downarrow{\scriptstyle id+(id\times fun)} \\ & & S + S \times (S^*)^* \end{array}$$

$gene = (id + (id \times fun)) \cdot outP2$

Função de pós-processamento:

$auxPost :: (String, [[[String]]]) \to [[String]]$
$auxPost\ (s, lis) = [s] : (((\mathsf{map}\ \ (\lambda l \to s : l)) \cdot concat)\ lis)$

A função post gera a matriz a partir da ExpTree gerada pela funçao tax

$post = (\!|\, [singl \cdot singl, auxPost] \,|\!)_{Exp}$

$$\begin{array}{ccc} (S^*)^* & \xleftarrow{\quad gene \quad} & S + S \times (S^*)^* \\ {\scriptstyle post}\uparrow & & \uparrow{\scriptstyle id+id\times post} \\ ExpSS & \xrightarrow[outExp]{} & S + S \times (S^*) \end{array}$$

Problema 3

Para resolver o problema 3, temos de tal como mencionado no enunciado, definir os genes do hilomofismo *sierpinski*, ou seja definir *gr2l* e *gsq* e correr a funcão *sierp4* depois, temos de definar as funções *present* e *carpets* de modo a podermos desenhar tapetes de Sierpinksi de profundidade n. Assim, de modo a alcançar a solução pretendida, codificamos as seguintes funções:

Primeiro começamos por definir a função *squares* como um anamorfismo de RoseTrees com gene *gsq*, esta função cria a árvore de suporte com todos os quadrados a desenhar, para uma dada profundidade.

$$squares = [\!(\, gsq \,)\!]_R$$

De modo a termos a lista dos nove quadrados gerados a partir de um quadrado inicial quando se faz uma iteração no tapete de Sierpinksi, tal como se pode ver nas figuras 2 e 3, em que um desses quadrados coincide com o quadrado a gerar no meio e os outros oito quadrados são colocados em torno do quadrado central, definimos a função *geraquadrados*. Esta função é bastante simples, pois apenas cria uma lista de quadrados, em que os quadrados gerados têm coordenadas diferentes do que o quadrado original, adicionando ou removendo um terço do *Side* do *Square*, passado como parâmetro. E alterando o valor do comprimento do lado do quadrado para um terço do valor original.

$$\begin{aligned}
&geraquadrados :: Square \to [Square] \\
&geraquadrados\ ((x,y),c) = [\\
&\quad ((x+a,y+a),a), \\
&\quad ((x,y+2*a),a), \\
&\quad ((x+a,y+2*a),a), \\
&\quad ((x+2*a,y+2*a),a), \\
&\quad ((x,y+a),a), \\
&\quad ((x+2*a,y+a),a), \\
&\quad ((x,y),a), \\
&\quad ((x+a,y),a), \\
&\quad ((x+2*a,y),a)] \\
&\quad \textbf{where} \\
&\qquad a = c\,/\,3
\end{aligned}$$

O gene *gsq*, tem como função gerar os quadrados a desenhar a partir de um quadrado original e uma dada profundidade.

Na forma como o definimos, utilizamos funções auxiliares, e para as explicarmos temos de observar o tipo dos argumentos com que trabalhamos. A primeira função que definimos foi, *paux*, esta função pega na segunda componente do tuplo passado como argumento (*Square*,*Int*), ou seja, no inteiro que representa o valor da profundidade e reduz esse valor uma unidade.

A função auxiliar *l*, também é bastante simples, simplesmente aplica a função *geraquadrados* para o quadrado passado no tuplo (*Square*,*Int*). Ou seja, *l* é a lista de nove quadrados gerados a partir do quadrado do argumento numa profundidade.

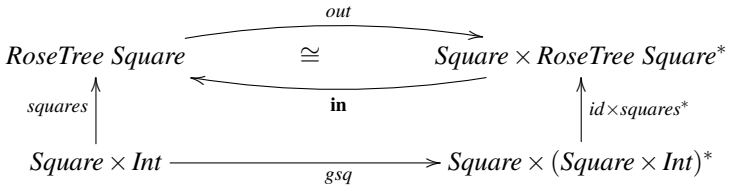
Assim, de modo a apresentar o tipo desejado no resultado, *(Square,[(Square,Int)])*, basta-nos pegar no primeiro quadrado gerado pela função *geraquadrados* que corresponde ao quadrado central, obtido através de *qc* e aplicar a *geraquadrados* para cada um dos quadrados gerados, decremento sempre a profundidade via *paux* e juntar num tuplo, ficando então com o resultado (quadrado inicial, lista de quadrados gerados a cada profundidade).

O aplicar da função que gera os quadrados a todos os quadrados gerados é garantido graças ao *map* que definimos, enquanto que a aplicação recursiva consoante o valor de *paux* é garantida por *qv*.

$$\begin{aligned}
&gsq :: (Square, Int) \to (Square, [(Square, Int)]) \\
&gsq\ t = (qc, \mathsf{map}\ \ (\lambda s \to (s, paux))\ qv) \\
&\quad \textbf{where} \\
&\qquad paux = (\pi_2\ t) - 1 \\
&\qquad l = geraquadrados\ (\pi_1\ t)
\end{aligned}$$

$$
\begin{aligned}
&qc = head\ l\\
&qv \mid paux \neq -1 = tail\ l\\
&\quad \mid otherwise = [\,]
\end{aligned}
$$

O esquema do anamorfismo *squares* é o seguinte:



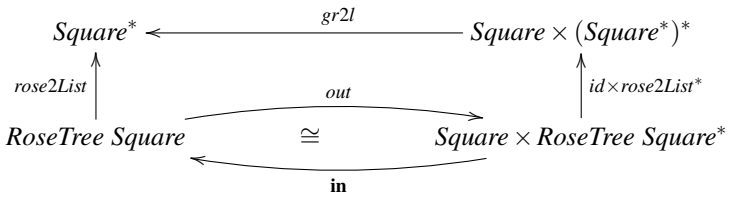
(2)

Podemos definir a função rose2 list através do catamorfismo de RoseTrees que tem como gene *gr2l*, esta função transforma uma RoseTree numa lista e será usada para extrair os quadrados da árvore auxiliar.

O gene *gr2l* é bastante simples e fizemo-lo de forma *point-free*. O que esta função faz é colocar à cabeça da lista a raiz da árvore e concatenar esse elemento à concatenação de todos os elementos das subarvores. o que se obtem aplicando *cons* ao produto de *id* e *concat*.

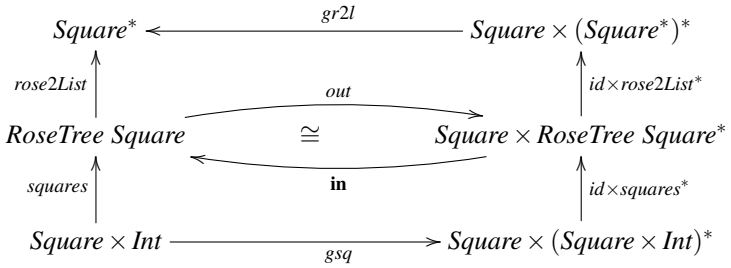
$$
\begin{aligned}
&rose2List = (\!|\, gr2l \,|\!)_R\\
&gr2l = cons \cdot (id \times concat)
\end{aligned}
$$

O esquema do catamorfismo *rose2List* tem o seguinte aspeto:



(3)

Logo, através dos esquemas já construidos, conseguimos elaborar o esquema do hilomofismo *sierpinski* da seguinte forma:



(4)

Por fim, como a função *constructSierp* é um hilomofismo de listas, é necessário definir o anamorfismo *carpets* e o catamorfismo *present* com os tipos presentes na dica do enunciado. A função *carpets* constroi a lista de todos os tapetes gerados até à profundidade passada como argumento. enquanto que a função *present* percorre a lista de quadrados gerados por *carpets*, desenhando-os esperando um segundo de intervalo.

Para conseguirmos trabalhar com a função *carpets* foi necessário criarmos um quadrado de exemplo, que serve como ponto de partida para a criação da lista dos tapetes de Sierpinksi. Esse quadrado chama-se *squareExemplo* e caso desejamos criar tapetes a partir de outro ponto de partida, basta alterar o quadrado definido.

No caso da função *carpets*, a lista de quadrados gerados obtem-se aplicando o hilomofismo *sierpinski*, n vezes, uma vez para cada profundidade, o que se consegue através da lista com elementos de 0 até n.

A função *present* simplesmente aplica a função *drawSq* para todos os elementos da lista gerada por *carpets*, esperando depois 1 segundo graças à função await.

$$carpets :: Int \to [[Square]]$$
$$carpets\ n = \mathsf{map}\ \ (\lambda s \to (sierpinski\ (squareExemplo, s)))\ [0..n]$$
$$squareExemplo = ((0.0, 0.0), 32.0)$$
$$present :: [[Square]] \to \mathsf{IO}\ [()]$$
$$present = sequence \cdot \mathsf{map}\ \ (\lambda l \to \mathbf{do}\ \{(drawSq\ l); await; \})$$

Problema 4

Versão não probabilística

O gene de $consolidate'$ é:

$$cgene :: (Num\ b, Eq\ a) \Rightarrow () + ((a,b), [(a,b)]) \to [(a,b)]$$
$$cgene = [nil, funCgene]$$

A funCgene é definida da seguinte maneira:

$$funCgene :: (Eq\ a, Num\ b) \Rightarrow ((a,b), [(a,b)]) \to [(a,b)]$$
$$funCgene\ arg = \mathbf{case}\ lu\ \mathbf{of}$$
$$\quad Nothing \to cons\ arg$$
$$\quad Just\ n \to (cons \cdot ((id \times (+n)) \times (filter\ (\lambda tuplo \to (\pi_1\ tuplo) \not\equiv esq)))))\ arg$$
$$\quad \mathbf{where}$$
$$\quad\quad lu = ((\widehat{List.lookup}) \cdot (\pi_1 \times id))\ arg$$
$$\quad\quad esq = (\pi_1 \cdot \pi_1)\ arg$$

A nossa ideia com a funCgene foi primeiramente verificar se uma determinada "chave"existia numa lista de tuplos e se sim, qual era o valor associado. Para isso tivemos recurso à lookup definida no modulo List.hs . A função lookup desse modulo recebe a chave e a lista sem ser sub a forma de par. Devido a isso usamos a função uncurry.

Após esta fase temos 2 hipoteses, ou a chave foi encontrada e temos Just e o valor associado, ou não foi encontrada e temos Nothing.

Se tivermos Nothing o que esta função faz é invocar a função cons definida em Cp.hs e damos como argumento o par recebido.

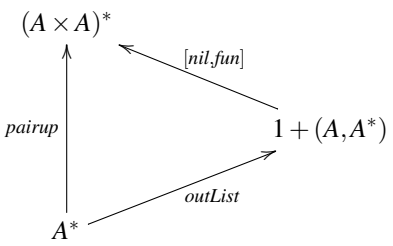
Caso o valor tenha sido encontrado, então vamos aplicar uma função à esquerda e uma função à direita no par recebido como argumento. À esquerda, mantemos a chave e aumentamos uma certa quantia no valor. Essa quantia foi o valor encontrado pela função lookup. Como nós não queremos ter chaves repetidas, ao lado esquerdo simplesmente invocamos a função filter para eliminar o elemento com a tal chave encontrado pela função lookup.

Após estas duas operações é invocada a função cons definida no modulo Cp.hs .

A função pairup definimos da seguinte forma:

$$pairup = ([nil, fun] \cdot outList)$$
$$\quad \mathbf{where}$$
$$\quad\quad fun = \mathsf{conc} \cdot \langle fun2, pairup \cdot \pi_2 \rangle$$
$$\quad\quad fun2\ (x,y) = \mathsf{map}\ \ (\lambda e \to (x,e))\ y$$

A nossa função pairup pode ser representada da seguinte forma:



O diagrama da nossa função auxiliar fun, é o seguinte:

$$(A, A^*) \xrightarrow{\langle fun2, pairup \cdot \pi_2 \rangle} (A^*, A^*) \xrightarrow{\text{conc}} (A \times A)^*$$

A nossa função *fun2* é muito similar à função zip. A função zip associa elementos de 2 listas, originando uma lista com tuplos das respetivas associações. A nossa função *fun2* simplesmente um elemento a cada um dos elementos de uma lista, originando na mesma tuplos.

Passando para a função, matchResult, numa primeira abordargem defimos a matchResult da seguinte forma:

```
matchResult f p = [(e1, pontos1), (e2, pontos2)]
where
  e1 = π1 p
  e2 = π2 p
  result = f p
  pontos1 = getpontuacao result e1
  pontos2 = getpontuacao result e2
```

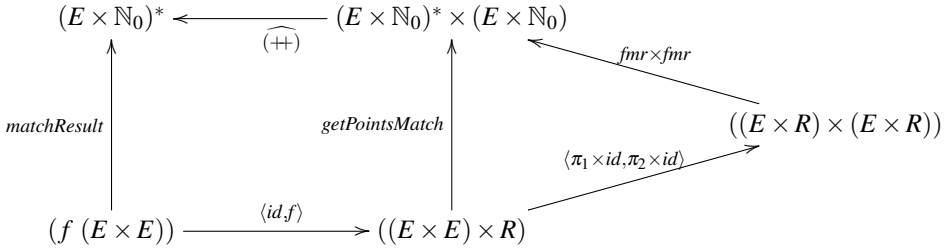
Esta função é bastante simples, mas não trabalha como uma função de point free. Após resolvermos todos os exercícios do projeto, proposemo-nos em adaptar certas funções para point free. A match result foi uma delas. Sendo assim a nossa função matchResult ficou definida da seguinte forma:

```
matchResult f = getPointsMatch · ⟨id, f⟩
```

As funções auxiliares a matchResult são:

```
getpontuacao :: (Team, Maybe Team) → ℤ
getpontuacao (e, r) = case r of
  Nothing → 1
  Just t → if t ≡ e then 3
    else 0
getPointsMatch :: (Match, Maybe Team) → [(Team, ℤ)]
getPointsMatch = conc · (fmr × fmr) · ⟨π1 × id, π2 × id⟩
  where
    fmr = singl · ⟨π1, getpontuacao⟩
```

Um diagrama como a função matchResult, junto com a getPointsMatch funcionam em conjunto:



A função fmr é representada da seguinte forma:

$$(E \times R) \xrightarrow{\langle \pi_1, getpontuacao \rangle} (E \times \mathbb{N}_0) \xrightarrow{singl} (E, Nat)^*$$

Passaremos a explicar cada passo desta nossa função matchResult. Primeiramente a nossa função pega na função recebida como argumento e aplica à partida em questão. De forma a não perder quais

as equipas que jogaram, é originado o par (Partida, Resultado) ou ((Equipa 1, Equipa 2), Resultado). A etapa seguinte é tentar organizar este tuplo em (equipa,resultado). Para isto utilizamos mais uma vez a função split que transforma o par ((Equipa 1, Equipa 2), Resultado) no par ((Equipa 1, Resultado), (Equipa 2, Resultado)). Quisemos organizar assim para ser mais fácil a utilização da nossa função getpontuacao. Após isto invocamos a função fmr que que transforma o último par em ([(Equipa 1, Pontuação 1)], [(Equipa 2, Pontuação 1)]), utilizando a função getpontuacao como auxiliar. Após esta fase basta apenas adaptar o tipo deste resultado para o pretendido que é [(Equipa 1, Pontuação 1), (Equipa 2, Pontuação 1)]. Para isso basta invocar a função conc definida no modulo Cp.hs.

Passando para a função glt, o diagrama da nossa função glt é o seguinte:

$$A^* \xrightarrow{glt} A + (A^* \times A^*) \tag{5}$$

A nossa função glt, quando recebe uma lista com apenas um único elemento, injeta esse mesmo elemento à esquerda numa alternativa. Caso contrário, invoca a função *splitAt* do prelude do Haskell. Esta função necessita de um indice para separar a lista. Este indice neste exercício será sempre o meio da lista.

$$\begin{aligned} &glt = (id + (\widehat{splitAt} \cdot \langle metade \cdot length, id \rangle \cdot cons)) \cdot outP2 \\ &\quad \textbf{where} \\ &\quad\quad metade\ n = n \div 2 \end{aligned}$$

Versão probabilística

Implementamos a função pinitKnockoutStage da seguinte forma:

$$\begin{aligned} &pinitKnockoutStage :: [[Team]] \to \text{Dist } (LTree\ Team) \\ &pinitKnockoutStage = (\!|\, [createDist, tupleDist] \,|\!) \cdot initKnockoutStage \\ &\quad \textbf{where} \\ &\quad\quad createDist\ t = D\ [(Leaf\ t, 1)] \\ &\quad\quad tupleDist = (\widehat{joinWith\ \overline{Fork}}) \end{aligned}$$

Esta função primeiramente invoca a função initKnockoutStage definida já na versão não probabilística.

Após isto nós obtemos uma LTree já com as partidas formadas. Agora basta apenas iniciar as probabilidades.

Para isto usamos a cataLTree defina no module LTree.hs. Esta função para as folhas, cria a distribuição com probabilidade igual a 100%. Como os nodos são apenas 1 equipa, a probabilidade de elas mesmas "ganharem"é 100%, dai atribuirmos essa probabilidade.

Para os nodos Fork, é aplicada a função tupleDist. Esta função cria o Fork, pois a cataLTree "destroi"a LTree e é necessário contrui-la outra vez. Depois disso, como as subarvores são distribuição de LTree, precisamos de utilizar a função joinWith definida em Probability.hs de forma a criar as combinações das distribuições.

Um esquema a demonstrar como esta função funciona:

$$\begin{array}{ccc} Dist(LTreeTeam) & \xleftarrow{[createDist,tupleDist]} & Team + (Dist\ (LTree\ Team) \times Dist\ (LTree\ Team)) \\ \uparrow (\!|\, [createDist,tupleDist] \,|\!) & & \uparrow id + pgroupWinners \times pgroupWinners \\ LTreeTeam & \xrightarrow[\text{out } LTree]{} & Team + (LTree\ Team \times LTree\ Team) \\ \uparrow initKnockoutStage & & \\ (Team^*)^* & & \end{array} \tag{6}$$

Partindo para a função pgroupWinners, esta nossa função ficou definida da seguinte forma:

$$pgroupWinners :: (Match \to \text{Dist } (Maybe\ Team)) \to [Match] \to \text{Dist } [Team]$$
$$pgroupWinners\ f = pmatchResult \cdot \langle sequence \cdot \text{map } (\lambda m \to (f\ m)), id \rangle$$

Esta função primeiramente gera as probabilidade do resultado de todas as partidas e de seguida invoca a função pmatchResult.

A função pmatchResult está implementada assim:

$$getPoints :: ([Match], [Maybe\ Team]) \to [(Team, \mathbb{Z})]$$
$$getPoints = (\!|\, [nil, \text{conc} \cdot (getPointsMatch \times id)]\, |\!) \cdot \widehat{zip}$$
$$pmatchResult :: (\text{Dist } [Maybe\ Team], [Match]) \to \text{Dist } [Team]$$
$$pmatchResult\ (resultados, partidas) = mapD\ (best\ 2 \cdot consolidate \cdot getPoints \cdot \langle \underline{partidas}, id \rangle)\ resultados$$

A função pmatchResult usa como função auxiliar getPoints. Esta função getPoints é praticamente igual à função getPointsMatch, até que não é por acaso que a utiliza. A diferença está nos argumentos, em que getPointsMatch recebe um tuplo (Match, Maybe Team) e getPoints recebe ([Match],[Maybe Team]).

A função pmatchResult simplesmente aplica uma sequência de funções às probabilidades geradas na função pgroupWinners. Primeiramente é gerado o tuplo ([Match],[Maybe Team]) para usarmos a função getPoints. Depois invocamos a função já definida previamente pela equipa docente, a consolidate, de forma a obtermos as pontuações finais de cada equipa no final do grupo. Depois invocamos mais uma vez uma função já previamente definida, a best, que vai buscar a n melhores equipas em cada grupo. Para aplicar estas funções às probabilidades, utilizamos a função mapD que aplica uma certa função ao conteúdo de uma distribuição.

A pedido da equipa docente vamos testar as funções pwinner e winner para rankings diferentes.

Sendo assim manipulamos os rankings para a seguinte forma:

```
rankings = [
  ("Argentina",4.4),
  ("Australia",4.0),
  ("Belgium",4.2),
  ("Brazil",4.7),
  ("Cameroon",4.0),
  ("Canada",4.0),
  ("Costa Rica",4.1),
  ("Croatia",4.7),
  ("Denmark",4.3),
  ("Ecuador",4.0),
  ("England",3.8),
  ("France",5.0),
  ("Germany",5.0),
  ("Ghana",3.9),
  ("Iran",4.0),
  ("Japan",4.1),
  ("Korea Republic",3.9),
  ("Mexico",4.5),
  ("Morocco",4.2),
  ("Netherlands",4.8),
  ("Poland",4.2),
  ("Portugal",5.5),
  ("Qatar",3.9),
  ("Saudi Arabia",3.9),
  ("Senegal",4.3),
  ("Serbia",3.8),
  ("Spain",4.0),
```

```
("Switzerland",4.4),
("Tunisia",4.1),
("USA",4.4),
("Uruguay",4.5),
("Wales",4.3)]
```

O resultado da execução de winner foi:

```
*Main> winner
"Portugal"
```

O resultado da execução de pwinner foi:

```
*Main> pwinner
  "Portugal"  33.7%
    "France"  14.7%
   "Germany"  14.0%
"Netherlands"  8.8%
   "Croatia"  5.6%
    "Mexico"  3.8%
    "Brazil"  3.8%
   "Uruguay"  2.8%
       "USA"  2.7%
   "Senegal"  2.0%
"Switzerland"  1.8%
 "Argentina"  1.6%
     "Wales"  1.6%
   "Denmark"  1.2%
   "Morocco"  0.7%
   "Tunisia"  0.3%
    "Poland"  0.3%
     "Spain"  0.2%
     "Japan"  0.1%
"Costa Rica"  0.1%
    "Canada"  0.1%
```

Tendo em conta o nosso novo ranking, os resultados são os esperados.

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.