Cálculo de Programas

Trabalho Prático

LEI — 2022/23

Departamento de Informática
Universidade do Minho

Janeiro de 2023

Grupo 48.	99 (preencher)
a95437	Beatriz Ribeiro Monteiro
a95719	João Pedro Machado Ribeiro
a96386	João Miguel Rodrigues da Cunha
a96626	João Carlos Fernandes Novais

Preâmbulo

Cálculo de Programas tem como objectivo principal ensinar a programação de computadores como uma disciplina científica. Para isso parte-se de um repertório de *combinadores* que formam uma álgebra da programação (conjunto de leis universais e seus corolários) e usam-se esses combinadores para construir programas *composicionalmente*, isto é, agregando programas já existentes.

Na sequência pedagógica dos planos de estudo dos cursos que têm esta disciplina, opta-se pela aplicação deste método à programação em Haskell (sem prejuízo da sua aplicação a outras linguagens funcionais). Assim, o presente trabalho prático coloca os alunos perante problemas concretos que deverão ser implementados em Haskell. Há ainda um outro objectivo: o de ensinar a documentar programas, a validá-los e a produzir textos técnico-científicos de qualidade.

Antes de abodarem os problemas propostos no trabalho, os grupos devem ler com atenção o anexo A onde encontrarão as instruções relativas ao sofware a instalar, etc.

Problema 1

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

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

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

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

$f\ 1\ 2\ 3$ irá gerar a sequência:

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

etc.

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

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

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

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

Problema 2

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

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

(10 primeiros ítens) etc., etc.[1]

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

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

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

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

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

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

   Na figura 1 mostra-se a representação gráfica da árvore de tipo Exp que representa o fragmento de acm_ccs mostrado acima.

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

[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.

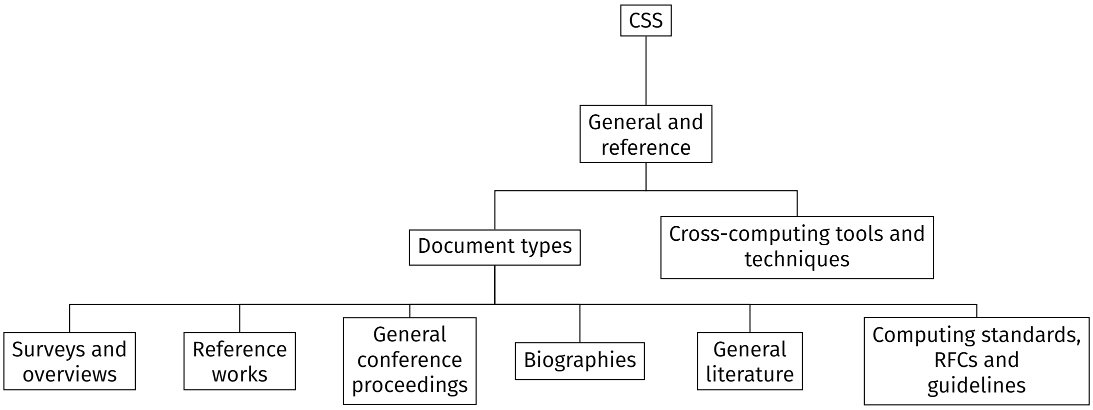


Figura 1: Fragmento de acm_ccs representado sob a forma de uma árvore do tipo Exp.

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.

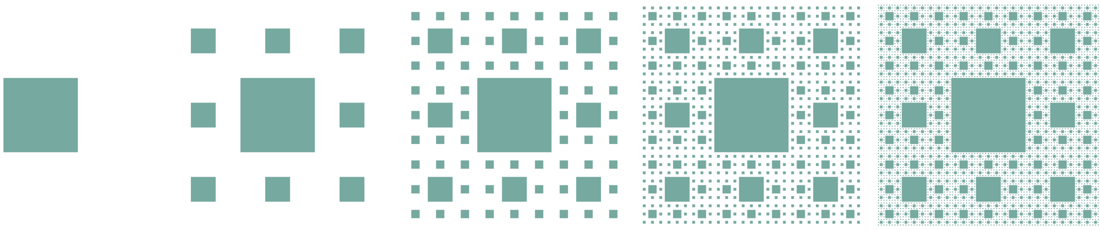


Figura 2: Construção do tapete de Sierpinski com profundidade 5.

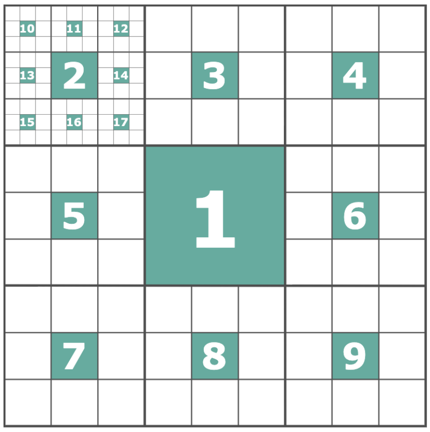


Figura 3: Tapete de Sierpinski com profundidade 2 e com os quadrados enumerados.

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

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

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

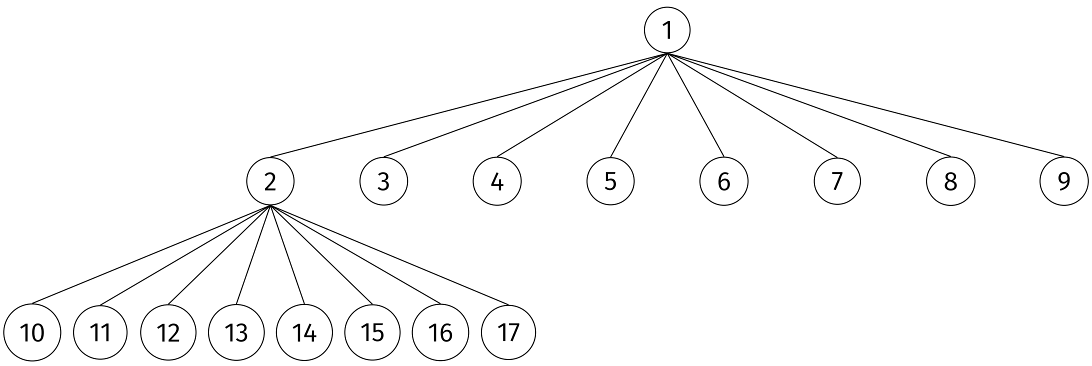


Figura 4: Possível árvore de suporte para a construção da fig. 3.

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

[4]A ordem dos filhos não é relevante.

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

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

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

Uma vez gerada a árvore com todos os quadrados a desenhar, é necessário extrair os quadrados para uma lista, a qual é processada pela função $drawSq$, disponibilizada no anexo D.

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

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

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

Trabalho a fazer:

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

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

3. Definir a função que apresenta a construção do tapete de Sierpinski como é apresentada em $constrаcaoSierp5$, mas para uma profundidade $n \in \mathbb{N}$ recebida como parâmetro.

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

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

Problema 4

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

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

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

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

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

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

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

Descrição do problema

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

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

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

Os grupos para o campeonato deste ano são:

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

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

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

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

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

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

Arquitetura proposta

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

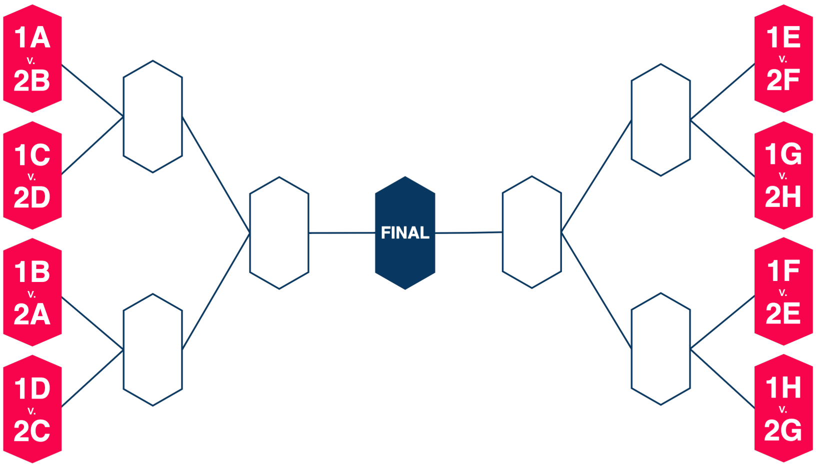


Figura 5: O "mata-mata"

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

Versão não probabilística

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

$$winner :: Team$$
$$winner = wcup\ groups$$
$$wcup = knockoutStage \cdot groupStage$$

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

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

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

> *Vamos dividir o processo em 3 partes:*
> - *gerar os jogos,*
> - *simular os jogos,*
> - *preparar o "mata-mata" gerando a árvore de jogos dessa fase (fig. 5).*
>
> *Assim:*
>
> $$groupStage :: [Group] \rightarrow \mathsf{LTree}\ Team$$
> $$groupStage = initKnockoutStage \cdot simulateGroupStage \cdot genGroupStageMatches$$
>
> *Comecemos então por definir a função $genGroupStageMatches$ que gera os jogos da fase de grupos:*
>
> $$genGroupStageMatches :: [Group] \rightarrow [[Match]]$$
> $$genGroupStageMatches = \mathsf{map}\ \ generateMatches$$
>
> *onde*
>
> $$\textbf{type}\ Match = (Team, Team)$$
>
> *Ora, sabemos que nos foi dada a função*
>
> $$gsCriteria :: Match \rightarrow Maybe\ Team$$
>
> *que, mediante um certo critério, calcula o resultado de um jogo, retornando $Nothing$ em caso de empate, ou a equipa vencedora (sob o construtor $Just$). Assim, precisamos de definir a função*

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

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

$$groupWinners\ criteria = best\ 2 \cdot consolidate \cdot (\gg\!\!= matchResult\ criteria)$$

Aqui está apenas em falta a definição da função $matchResult$*.*
Por fim, teremos a função $initKnockoutStage$ *que produzirá a LTree que a sub-equipa do "mata-mata" precisa, com as devidas posições. Esta será a composição de duas funções:*

$$initKnockoutStage = [\![glt]\!] \cdot arrangement$$

Trabalho a fazer:

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

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

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

Versão probabilística

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

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

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

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

etc.

O que é Dist? É o mónade que trata de distribuições probabilísticas e que é descrito no anexo C, página 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:

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

$$pwcup = pknockoutStage \bullet pgroupStage$$

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

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

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

A sub-equipa dos alunos também já adiantou trabalho,

$$pgroupStage = pinitKnockoutStage \bullet psimulateGroupStage \cdot genGroupStageMatches$$

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

Trabalho a fazer:

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

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

Anexos

A Documentação para realizar o trabalho

Para cumprir de forma integrada os objectivos do trabalho vamos recorrer a uma técnica de programação dita "literária" [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]

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

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

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

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

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

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

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

[7] Exemplos tirados de [3].

[8] Lei (3.95) em [3], página 112.

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

Obter-se-á de imediato

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

usando as regras seguintes:

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

Mais um exemplo, envolvendo polinómios do segundo grau $ax^2 + bx + c$ em $\mathbb{N}_0$. Seguindo o método estudado nas aulas[10], de $f\ x = ax^2 + bx + c$ derivam-se duas funções mutuamente recursivas:

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

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

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

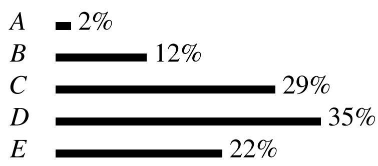
C O mónade das distribuições probabilísticas

Mónades são functores com propriedades adicionais que nos permitem obter efeitos especiais em programação. Por exemplo, a biblioteca Probability oferece um mónade para abordar problemas de probabilidades. Nesta biblioteca, o conceito de distribuição estatística é captado pelo tipo

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

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

Cada par (a, p) numa distribuição $d :: \mathsf{Dist}\ a$ indica que a probabilidade de a é p, devendo ser garantida a propriedade de que todas as probabilidades de d somam 100%. Por exemplo, a seguinte distribuição de classificações por escalões de A a E,



será representada pela distribuição

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

que o GHCi mostrará assim:

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

[10] Secção 3.17 de [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 $g : A \rightarrow \mathsf{Dist}\ B$ e $f : B \rightarrow \mathsf{Dist}\ C$ são funções **monádicas** que representam *computações probabilísticas*.

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

D Código fornecido

Problema 1

Alguns testes para se validar a solução encontrada:

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

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:

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

Para efeitos de temporização:

$$\mathit{await} = \mathit{threadDelay}\ 1000000$$

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:

$$\mathit{generateMatches} = \mathit{pairup}$$

Preparação da árvore do “mata-mata”:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Auxiliares:

$$best\ n = \mathsf{map}\ \ \pi_1 \cdot take\ n \cdot reverse \cdot presort\ \pi_2$$

$$\begin{array}{l} consolidate :: (Num\ d, Eq\ d, Eq\ b) \Rightarrow [(b,d)] \to [(b,d)] \\ consolidate = \mathsf{map}\ \ (id \times sum) \cdot collect \end{array}$$

$$\begin{array}{l} collect :: (Eq\ a, Eq\ b) \Rightarrow [(a,b)] \to [(a,[b])] \\ collect\ x = nub\ [k \mapsto [d' \mid (k',d') \leftarrow x, k' \equiv k] \mid (k,d) \leftarrow x] \end{array}$$

Função binária monádica f:

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

Monadificação de uma função binária f:

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

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

Outras funções que podem ser úteis:

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

E Soluções dos alunos

Os alunos devem colocar neste anexo as suas soluções para os exercícios propostos, de acordo com o "layout" que se fornece. Não podem ser alterados os nomes ou tipos das funções dadas, mas pode ser adicionado texto, diagramas e/ou outras funções auxiliares que sejam necessárias.

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

Problema 1

Funções auxiliares pedidas:

Para tornar fbl num *for*, é necessário primeiro definir algumas funções auxiliares, que nos permitam através das leis da recursividade mútua chegar a um catamorfismo, neste caso dos Números Naturais. Começamos por definir algumas funções iniciais (p, q e r), que se chamam recursivamente umas às outras, um dos princípios da recursividade mútua, e após continuarmos a desenvolver estes cálculos, conseguimos finalmente chegar a um Catamorfismo que pode ser transformado num *for*, conseguindo assim descobrir a definição de *loop* e *initial*.

Consideremos F2:

id + $\langle\langle j,k\rangle,l\rangle$ = F $\langle\langle j,k\rangle,l\rangle$

$$\begin{cases} p\ a\ b\ c\ 0 = 1 \\ p\ a\ b\ c\ (n+1) = a * (p\ a\ b\ c\ n) + b * q\ a\ b\ c\ n + c * r\ a\ b\ c\ n \end{cases}$$

$$\begin{cases} q\ a\ b\ c\ 0 = 1 \\ q\ a\ b\ c\ (n+1) = p\ n \end{cases}$$

$$\begin{cases} r\ a\ b\ c\ 0 = 0 \\ r\ a\ b\ c\ (n+1) = q\ n \end{cases}$$

$\equiv$ $\{$ Def-const (75) * 6; Def succ; Def (a*); Def (b*); Def (c*); Def-comp(73) * 8; Def-split(77)*2 $\}$

$$\begin{cases} ((p\ a\ b\ c) \cdot 0)\ n = \underline{1}\ n \\ ((p\ a\ b\ c) \times \mathsf{succ}\ )\ n = add \cdot \langle add \cdot \langle (a*) \cdot (p\ a\ b\ c), (b*) \cdot (q\ a\ b\ c)\rangle, (c*) \cdot (r\ a\ b\ c)\rangle\ n \end{cases}$$

$$\begin{cases} ((q\ a\ b\ c) \cdot 0)\ n = \underline{1}\ n \\ ((q\ a\ b\ c) \cdot \mathsf{succ}\ )\ n = p\ a\ b\ c\ n \end{cases}$$

$$\begin{cases} ((r\ a\ b\ c) \cdot 0)\ n = \underline{0}\ n \\ ((r\ a\ b\ c) \cdot \mathsf{succ}\ )\ n = q\ a\ b\ c\ n \end{cases}$$

$\equiv$ $\{$ Igualdade estensional (72) * 6; Absorção-x (11) * 2 $\}$

$$\begin{cases} (p\ a\ b\ c) \cdot \underline{0} = \underline{1} \\ (p\ a\ b\ c) \times \mathsf{succ}\ = add \cdot (add \cdot ((a*) \times (b*)) \times (c*)) \cdot \langle\langle p\ a\ b\ c, q\ a\ b\ c\rangle, r\ a\ b\ c\rangle \end{cases}$$

$$\begin{cases} (q\ a\ b\ c) \cdot \underline{0} = \underline{1} \\ (q\ a\ b\ c) \cdot \mathsf{succ}\ = p\ a\ b\ c \end{cases}$$

$$\begin{cases} (r\ a\ b\ c) \cdot \underline{0} = \underline{0} \\ (r\ a\ b\ c) \cdot \mathsf{succ}\ = q\ a\ b\ c \end{cases}$$

$\equiv$ { Def in; Cancelamento-+ (18) * 6; p a b c = j; q a b c = k; r a b c = l }

$$\begin{cases} j \cdot \mathbf{in} \cdot i_1 = \underline{1} \\ j \cdot \mathbf{in} \cdot i_2 = add \cdot (add \cdot ((a*) \times (b*)) \times (c*)) \cdot \langle\langle j,k\rangle,l\rangle \end{cases}$$

$$\begin{cases} k \cdot \mathbf{in} \cdot i_1 = \underline{1} \\ k \cdot \mathbf{in} \cdot i_2 = j \end{cases}$$

$$\begin{cases} l \cdot \mathbf{in} \cdot i_1 = \underline{0} \\ l \cdot \mathbf{in} \cdot i_2 = k \end{cases}$$

$\equiv$ { Cancelamento-+ (18) * 3 }

$$\begin{cases} j \cdot \mathbf{in} = [\underline{1}, add \cdot (add \cdot ((a*) \times (b*)) \times (c*)) \cdot \langle\langle j,k\rangle,l\rangle] \\ k \cdot \mathbf{in} = [\underline{1}, j] \\ l \cdot \mathbf{in} = [\underline{0}, k] \end{cases}$$

$\equiv$ { Natural-id(1); Absorção-x(22); Cancelamento-x(7) * 4 }

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

$\equiv$ { Natural-id(1)*2; Absorção-x(22) * 2;F2 }

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

$\equiv$ { Eq-x(16); Fusão-x(9) * 2 }

$$\begin{cases} \langle j,k\rangle \cdot \mathbf{in} = \langle [\underline{1}, add \cdot (add \cdot ((a*) \times (b*)) \times (c*))], [\underline{1}, \pi_1 \cdot \pi_1]\rangle \cdot F\ \langle\langle j,k\rangle,l\rangle \\ l \cdot \mathbf{in} = [\underline{1}, \pi_2 \cdot \pi_1] \cdot F\ \langle\langle j,k\rangle,l\rangle \end{cases}$$

$\equiv$ { Lei da Troca(28) }

$$\begin{cases} \langle j,k\rangle \cdot \mathbf{in} = [\langle \underline{1}, \underline{1}\rangle, \langle add \cdot (add \cdot ((a*) \times (b*)) \times (c*)), \pi_1 \cdot \pi_1\rangle] \cdot F\ \langle\langle j,k\rangle,l\rangle \\ l \cdot \mathbf{in} = [\underline{1}, \pi_2 \cdot \pi_1] \cdot F\ \langle\langle j,k\rangle,l\rangle \end{cases}$$

$\equiv$ { Fokkinga(53) }

$$\langle\langle j,k\rangle,l\rangle = (\![[\langle\langle \underline{1}, \underline{1}\rangle, \underline{0}\rangle, \langle\langle add \cdot (add \cdot ((a*) \times (b*)) \times (c*)), \pi_1 \cdot \pi_1\rangle, \pi_2 \cdot \pi_1\rangle]]\!)$$

$\equiv$ { j = p a b c; k = q a b c; l = r a b c }

$$\langle\langle p\ a\ b\ c, q\ a\ b\ c\rangle, r\ a\ b\ c\rangle = (\![[\langle\langle \underline{1}, \underline{1}\rangle, \underline{0}\rangle, \langle\langle add \cdot (add \cdot ((a*) \times (b*)) \times (c*)), \pi_1 \cdot \pi_1\rangle, \pi_2 \cdot \pi_1\rangle]]\!)$$

$\square$

$$\langle\langle \underline{1}, \underline{1}\rangle, \underline{0}\rangle = \underline{((1,1),0)}$$

{ Igualdade estensional (72); Def-split(77)*2; Def-const (75) * 4 }

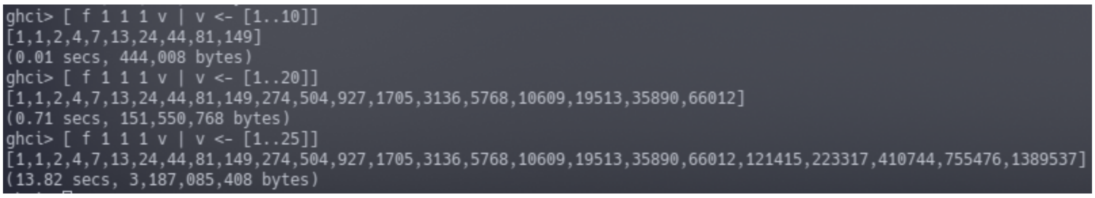
$$((1,1),0) = ((1,1),0)$$

$\square$

$$
\begin{aligned}
&loop\ a\ b\ c = \langle \langle add \cdot ((add \cdot ((a*) \times (b*))) \times (c*)), \pi_1 \cdot \pi_1 \rangle, \pi_2 \cdot \pi_1 \rangle \\
&initial = ((1,1),0) \\
&wrap = \pi_2
\end{aligned}
$$

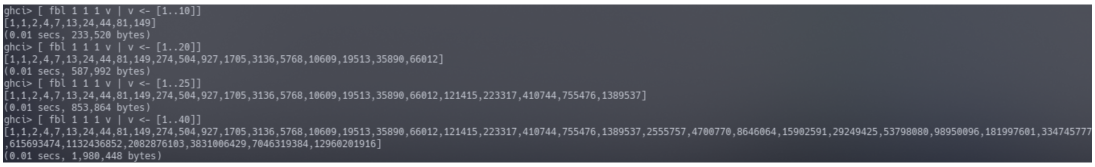
Testes de Performance

Apresenta-se agora duas imagens (figuras 6 e 7) que demonstram o quão mais eficiente fbl é em relação a f.



```
ghci> [ f 1 1 1 v | v <- [1..10]]
[1,1,2,4,7,13,24,44,81,149]
(0.01 secs, 444,008 bytes)
ghci> [ f 1 1 1 v | v <- [1..20]]
[1,1,2,4,7,13,24,44,81,149,274,504,927,1705,3136,5768,10609,19513,35890,66012]
(0.71 secs, 151,550,768 bytes)
ghci> [ f 1 1 1 v | v <- [1..25]]
[1,1,2,4,7,13,24,44,81,149,274,504,927,1705,3136,5768,10609,19513,35890,66012,121415,223317,410744,755476,1389537]
(13.82 secs, 3,187,085,408 bytes)
```

Figura 6: Execução de f 1 1 1 para diferentes tamanhos de input



```
ghci> [ fbl 1 1 1 v | v <- [1..10]]
[1,1,2,4,7,13,24,44,81,149]
(0.01 secs, 233,520 bytes)
ghci> [ fbl 1 1 1 v | v <- [1..20]]
[1,1,2,4,7,13,24,44,81,149,274,504,927,1705,3136,5768,10609,19513,35890,66012]
(0.01 secs, 587,992 bytes)
ghci> [ fbl 1 1 1 v | v <- [1..25]]
[1,1,2,4,7,13,24,44,81,149,274,504,927,1705,3136,5768,10609,19513,35890,66012,121415,223317,410744,755476,1389537]
(0.01 secs, 853,864 bytes)
ghci> [ fbl 1 1 1 v | v <- [1..40]]
[1,1,2,4,7,13,24,44,81,149,274,504,927,1705,3136,5768,10609,19513,35890,66012,121415,223317,410744,755476,1389537,2555757,4700770,8646064,15902591,29249425,53798080,98950096,181997601,334745777
,615693474,1132436852,2082876103,3831006429,7046319384,12960201916]
(0.01 secs, 1,980,448 bytes)
```

Figura 7: Execução de fbl 1 1 1 para diferentes tamanhos de input

É possível ver que para f (figura 6), à medida que o tamanho do input vai aumentando, o tempo de execução vai aumentando também, mas de forma exponencial, enquanto para fbl (figura 7) o tempo de execução se mantém igual (não mostramos mais porque já deu para verificar que é muito mais eficiente).

Problema 2

Neste exercício é pedido que apresentemos uma definição para o gene do anaformismo do tipo *Exp S S*, a partir de uma lista de String. Olhando para a definição de um anaformismo para este tipo de dados, vimos que o seu Functor aplica a um id lado esquerdo do + (ou seja, a tudo que apareça do tipo i1 a, é lhe aplicado um id), e do lado direito aplica um id ao lado esquerdo da multiplicação e faz um map com o anamorfismo que estejamos a definir do lado direito, pelo que na nossa definição de gene, decidimos tirar partido de out, que no caso de uma lista ser singular (só um elemento), devolve i1 com esse elemento, caso contrário coloca o primeiro elemento na primeira posição de um par, e o resto na segunda, aplicando ao resultado dessa função uma soma de funções, que a algo do tipo i1 lhe faz id, e se for um par separa a lista (lado direito) pelas diferentes sub-árvores com um nível de indentação acima do elemento da cabeça (primeiro elemento do par).

Gene de *tax*:

$$
\begin{aligned}
&gene = (id + separa) \cdot out \\
&separa\ (a,b) = (a, (groupBy\ (\backslash\_y \rightarrow nivelIdent\ y > identPai)\ b)) \\
&\quad \mathbf{where}\ identPai = 1 + nivelIdent\ a \\
&nivelIdent\ l = contaEspacos\ l \div 4 \\
&contaEspacos\ [\,] = 0 \\
&contaEspacos\ (h : t) = \mathbf{if}\ (h \not\equiv \text{'\ \ '})\ \mathbf{then}\ 0\ \mathbf{else}\ 1 + contaEspacos\ t
\end{aligned}
$$

$$
\begin{tikzcd}
S^* \arrow[rr, "gene"] \arrow[d, "tax"'] & & S^* + S^* \times (S^*)^* \arrow[d, "id + id \times map\, tax"] \\
Exp\ S\ S & & S^* + S^* \times Exp\ S\ S \arrow[ll, "\mathsf{in}_{Exp}"]
\end{tikzcd}
$$

Nesta segunda parte do exercício, é pedido para definir a função post, que recebe uma Exp S S e retorna uma lista de lista de String, em que cada uma dessas listas corresponderá a um caminho na árvore. Para a definir acabamos por definir duas funções auxiliares, uma que coloca um elemento numa lista de listas (aplicado a elementos da árvore do tipo Var), e outra que recebendo um par (elemento,lista de listas), coloca esse elemento na cabeça de todas as listas que estejam na lista que está do lado direito do par. Assim, resumidamente, esta função começa a desconstruir a árvore Exp S S pelas folhas, e começa a construir o resultado final através da adição das diferentes String de nível abaixo (considerando que um nível maior corresponde a maior indentação) à cabeça das folhas, criando os caminhos iterativamente, até que se chegue ao pai, tendo assim o resultado final.

Função de pós-processamento:

$$
\begin{array}{l}
post = [doubleSingl, cons \cdot (\langle \pi_1, fstACabeca \rangle \cdot (singleton \times (concatMap\ (post))))] \cdot \mathsf{out}_{\mathit{Exp}} \\
\quad \textbf{where}\ doubleSingl = singleton \cdot singleton \\
\qquad fstACabeca\ (x, l) = \mathsf{map}\ \ (x +\!\!+)\ l
\end{array}
$$

Problema 3

Neste exercício foi nos pedido para fazer um anamorfismo de Rose Tree Square, ou seja, uma Rose Tree em que a informação contida na mesma são os quadrados correspondentes aos diferentes níveis do tapete de sierpinski.

À semelhança do exercício anterior onde fizemos um anamorfismo, começamos por averiguar o funcionamento de anaRose, anamorfismo de Rose Tree's, e verificamos que neste caso o Functor é aplicado a um par, em que ao elemento do lado esquerdo é aplicada a identidade, e do lado direito é feito um map com o próprio anamorfismo, pelo que o gene deste anamorfismo teria que transformar o input dado em algo compatível com esse tipo.

O que nós fizemos foi uma função que, ao receber um quadrado e uma profundidade, se essa profundidade for 0, retornamos um par em que do lado esquerdo termos o quadrado que se localiza no meio do que é dado como argumento, e do lado direito uma lista vazia. Caso essa profundidade não seja 0, o comportamento para o lado esquerdo é idêntico ao explicado para caso a profundidade seja 0, mas do lado direito é retornada uma lista com os sub quadrados onde os próximos elementos da sequência devem ser desenhados, com profundidade decrementada.

$$
\begin{array}{l}
squares = [\!(\ gsq\ )\!]_{R} \\
gsq\ (((x, y), s), 0) = (((x + (s / 3), y + (s / 3)), s / 3), [\,]) \\
gsq\ (sq@((x, y), s), p + 1) = (((x + newSide, y + newSide), newSide), filhosFinal) \\
\quad \textbf{where}\ newSide = s / 3 \\
\qquad filhos = coordsFilhos\ sq \\
\qquad filhosFinal = \mathsf{map}\ \ (addDir\ p)\ filhos \\
coordsFilhos\ ((x, y), side) = [((x + xO, y + yO), nS) \mid xO \leftarrow offsets, yO \leftarrow offsets, (xO \not\equiv yO \vee x \not\equiv nS)] \\
\quad \textbf{where}\ nS = side / 3 \\
\qquad offsets = [0, nS, 2 * nS] \\
addDir\ v\ p = (p, v)
\end{array}
$$

$$
\begin{tikzcd}
Square \times Integer \arrow[r, "gsp"] \arrow[d, "squares"'] & Square \times (Square \times Integer)^* \arrow[d, "id \times mapsquares"] \\
Rose\ Square & Square \times (RoseSquare)^* \arrow[l, "inRose"]
\end{tikzcd}
$$

Em seguida, ainda na primeira parte do Exercício 3, é pedido um catamorfismo usado para converter uma Rose Tree em uma lista.

Considerando que uma Rose Tree é constituída por um elemento e uma lista de Rose Tree, apenas aplicamos a identidade a esse elemento que ficará na cabeça, e concatenamos a lista de Rose Tree (que quando é concatenada ja foi transformada numa lista de listas de elementos).

$$rose2List = (\!| gr2l |\!)_R$$
$$gr2l = cons \cdot (id \times concat)$$

$$\begin{tikzcd}
Rose\ A \arrow[r, "outRose"] \arrow[d, "rose2List"'] & A \times (RoseA)^* \arrow[d, "id \times rose2List"] \\
{[A]} & A \times (A^*)^* \arrow[l, "gr2l"]
\end{tikzcd}$$

Já na segunda parte do exercício, foi nos pedido para definir, numa primeira parte, um anamorfismo, que construirá, recebendo um inteiro n, a lista com os tapetes de profundidade 1 até n.

No gene deste anamorfismo começamos por primeiro aplicar o outNat, para poder transformar em algo que o anamorfismo pudesse usar. Ao resultado dessa função aplicamos uma soma de funções, que a algo do tipo i1 () aplica a identidade, e a algo do tipo i2 a, aplicámos um split que cria um par que do lado esquerdo terá uma lista de Square (um tapete) e do lado direito terá a profundidade do próximo tapete a gerar.

$$sqInicial = \langle \underline{(0,0), 32}, id \rangle$$
$$carpets = [\!(geneCarpets)\!]$$
$$geneCarpets = ((id + \langle sierpinski \cdot sqInicial, id \rangle) \cdot outNat)$$

$$\begin{tikzcd}
\mathbb{N}_0 \arrow[r, "genecarperts"] \arrow[d, "carpets"'] & 1 + Square^* \times \mathbb{N}_0 \arrow[d, "id + id \times carpets"] \\
(Square^*)^* & 1 + Square^* \times (Square^*)^* \arrow[l, "inList"]
\end{tikzcd}$$

Em seguida foi pedido para definir um catamorfismo que percorre uma lista de listas de Square, desenhando todos os elementos, com 1 segundo de intervalo entre os mesmos.

Este exercício foi complicado pois adiciona uma componente monádica, sendo preciso termos em conta isso para o retorno da função. No gene do catamorfismo, se for recebido algo do tipo i1 () apenas retornamos uma lista vazia, e caso tenhamos algo do tipo i2 (a,b), aplicamos a função consIO que, recebendo um par ([Square], IO [()]), retorna apenas IO [()], e que, antes de fazer essa junção, desenha o tapete e espera 1 segundo.

É de reforçar que antes de correr o catamorfismo aplicamos um reverse à lista dada como input, visto que a lista retornada por carpets tem primeiro os tapetes de profundidade mais elevada, e nós aqui queremos desenhar primeiro os de profundidade mais baixa. Possivelmente esta não é a melhor alternativa, mas foi a única maneira que (até agora) arranjamos de resolver este problema.

$$ignora1\ a = return\ [\,]$$
$$apresentaSq :: [Square] \rightarrow \mathsf{IO}\ ()$$
$$apresentaSq\ a = drawSq\ a \gg await \ggg return$$
$$consIO\ (h, t) = \mathbf{do}\ \{ r_1 \leftarrow apresentaSq\ h; r_2 \leftarrow t; return\ (r_2 +\!\!+ [r_1]) \}$$
$$genePresent :: () + ([Square], \mathsf{IO}\ [()]) \rightarrow \mathsf{IO}\ [()]$$
$$genePresent = [ignora1, consIO]$$
$$present :: [[Square]] \rightarrow \mathsf{IO}\ [()]$$
$$present = (\!| genePresent |\!) \cdot reverse$$

Problema 4

Antes de partir para a resolução do exercício em si, queríamos apenas clarificar que apenas resolvemos a parte não probabilística deste exercício.

Versão não probabilística

Na primeira parte desta versão do exercício, é nos proposta a definição de cgene, o gene do catamorfismo consolidate', que é usado para calcular os pontos de cada equipa no final da simulação da fase de grupos.

Para a resolução desse problema definimos uma função auxiliar que recebe um par ((equipa, pontos), [(equipa, pontos)]), e que devolve [(equipa, pontos)], que soma todos os pontos que a equipa passa no primeiro par do argumento tem.

No próprio gene do catamorfismo, apenas aplicamos um either em que, algo do tipo i1 () é transformado numa lista vazia, a algo do tipo i2 a é aplicada a função explicada acima.

Gene de $consolidate'$:

$$cgene :: (Eq\ a, Num\ b) \Rightarrow () + ((a,b),[(a,b)]) \rightarrow [(a,b)]$$
$$cgene = [nil, adiciona]$$
$$adiciona\ (p,[\,]) = singleton\ p$$
$$adiciona\ ((a,b),(a_1,b_1):t) = \textbf{if}\ a \equiv a_1\ \textbf{then}\ (a,b+b_1):t\ \textbf{else}\ (a_1,b_1): adiciona\ ((a,b),t)$$

$$\begin{tikzcd}
(A\times B)^* \arrow[r, "outList"] \arrow[d, "consolidate'"'] & 1+(A\times B)\times(A\times B)^* \arrow[d, "id+id\times consolidate'"] \\
A\times B^* & 1+(A\times B)\times(A\times B)^* \arrow[l, "cgene"]
\end{tikzcd}$$

Em seguida é nos solicitado que apresentemos uma definição para a função pairup que, recebendo uma lista de equipas, retorna uma lista com todos os jogos que devem ser realizados entre essas equipas.

Depois pedem ainda que seja definida uma função que retorna uma lista com os pontos associados a duas equipas que realizam um jogo entre si, e ainda um gene para o anamorfismo que é usado para transformar a lista dos vencedores da fase de grupos em uma LTree de equipas, referente à fase eliminatória do campeonato.

Vamos agora explicar brevemente como pensamos para resolver cada um dos requisitos:

- pairup: recebe uma lista de equipas e adiciona à lista final todos os jogos que a equipa a cabeça pode fazer com o resto das equipas, chamando-se a si própria recursivamente para o resto das equipas.
- matchResult: aplicamos a função de critério ao jogo em questão, sabendo que o resultado disso é Maybe Team, definimos um outMaybe, que recebendo um Maybe a retorna i1 () caso seja Nothing e i2 a caso seja Just a, aplicando em seguida um either que retorna a lista correspondente aos pontos de cada equipa.
- glt: sendo este o gene de um anamorfismo de LTree A, tendo neste caso que A é do tipo Team. Definimos uma função auxiliar chamada outTeamList que, recebendo uma lista de equipas, se esta só tiver um elemento, retorna i1 a, sendo a esse elemento, e caso contenha mais que um elemento retorna i2 (a,b), em que a e b são duas metades da lista.

Geração dos jogos da fase de grupos:

$$pairup\ [\,] = [\,]$$
$$pairup\ (x:xs) = \mathsf{map}\ \ \langle \underline{x}, id\rangle\ xs +\!\!+ pairup\ xs$$
$$outMaybe\ Nothing = i_1\ ()$$
$$outMaybe\ (Just\ a) = i_2\ a$$
$$empate\ (a,b)\ \_ = [(a,1),(b,1)]$$
$$vencedor\ (a,b)\ v = [(vencedor,3),(derrotado,0)]$$
$$\quad \textbf{where}\ (vencedor, derrotado) = \textbf{if}\ a \equiv v\ \textbf{then}\ (a,b)\ \textbf{else}\ (b,a)$$
$$matchResult\ func\ jogo = ([empate\ jogo, vencedor\ jogo] \cdot outMaybe \cdot func)\ jogo$$

$$
\begin{array}{l}
outTeamList\ [t] = i_1\ t \\
outTeamList\ l = \mathbf{let}\ splitPoint = length\ l \div 2 \\
\quad sl = splitAt\ splitPoint\ l \\
\quad \mathbf{in}\ i_2\ sl
\end{array}
$$

$$glt = outTeamList$$

$$
\begin{tikzcd}
A^* \arrow[rr, "glt"] \arrow[d, "(\!|g|\!)"] & & A + A^* \times A^* \arrow[d, "id + (\!|glt|\!) \times (\!|glt|\!)"] \\
LTree A & & A + \textit{LTree}\ A \times \textit{LTree}\ A \arrow[ll, "\mathsf{in}_{\,LTree}"]
\end{tikzcd}
$$

Versão probabilística

Como já referimos, acabamos por não fazer esta parte do trabalho, apenas definimos a função pmatchResult, que recebendo o critério e um jogo nos dá um Dist ([(Team,Int)]), com a probabilidade dos pontos das equipas no final do jogo serem esses.

$$pinitKnockoutStage = \bot$$

$$getMatch\ (D\ l) = (\langle head, head \cdot tail \rangle \cdot (\mathsf{map}\ \ \pi_1) \cdot \pi_1 \cdot head)\ l$$

$$sameMatch\ (a,b)\ (a_1,b_1) = (a \equiv b \wedge a_1 \equiv b_1) \vee (a \equiv b_1 \wedge b \equiv a_1)$$

$$
\begin{array}{l}
pgroupWinners :: (Match \rightarrow \mathsf{Dist}\ (Maybe\ Team)) \rightarrow [Match] \rightarrow \mathsf{Dist}\ [Team] \\
pgroupWinners\ pcriteria = \bot
\end{array}
$$

$$pmatchResult\ func\ match = \mathsf{fmap}\ ([empate\ match, vencedor\ match] \cdot outMaybe)\ (func\ match)$$

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.