

# Cálculo de Programas

## Trabalho Prático

### LEI — 2022/23

Departamento de Informática  
Universidade do Minho

Janeiro de 2023

Grupo nr.	49
A95191	João Afonso Alvim Oliveira Dias de Almeida
A97763	Daniel Du
A88220	Xavier Santos Mota

## 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 abordar os problemas propostos no trabalho, os grupos devem ler com atenção o anexo [A](#) onde encontrarão as instruções relativas ao software 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, ...

$f\ 1\ 2\ 3$  irá gerar a sequência:

1, 1, 3, 8, 17, 42, 100, 235, 561, 1331, ...

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

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

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",
```

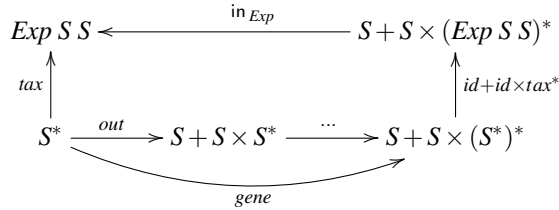
(10 primeiros itens) etc., etc.<sup>1</sup>

Pretende-se representar a mesma informação sob a forma de uma árvore de expressão, usando para isso a biblioteca [Exp](#) que consta do material pedagógico 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](#):

$$\begin{aligned} tax &:: [String] \rightarrow Exp\ String\ String \\ tax &= \llbracket gene \rrbracket_{Exp} \end{aligned}$$

Ou seja, defina o *gene* do anamorfismo, tendo em conta o seguinte diagrama<sup>2</sup>:



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).<sup>3</sup>

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

$$\begin{aligned} tudo &:: [String] \rightarrow [[String]] \\ tudo &= post \cdot tax \end{aligned}$$

para obter o efeito que se mostra na tabela 1.

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

<sup>1</sup>Informação obtida a partir do site [ACM CCS](#) seleccionando *Flat View*.

<sup>2</sup>*S* abrevia *String*.

<sup>3</sup>Para um exemplo de classificação de UC concreto, pf. ver a secção **Classificação ACM** na página pública de [Cálculo de Programas](#).

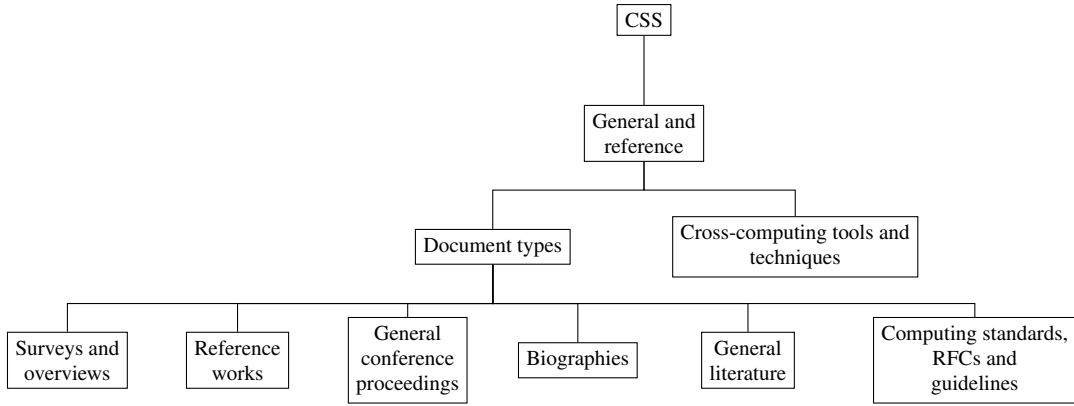


Figura 1: Fragmento de *acm\_ccs* representado sob a forma de uma árvore do tipo [Exp](#).

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 ítems).

**Sugestão:** Inspeccione as bibliotecas fornecidas à procura de funções auxiliares que possa re-utilizar para a sua solução ficar mais simples. Não se esqueça que, para o mesmo resultado, nesta disciplina “*ganha*” quem escrever menos código!

**Sugestão:** Para efeitos de testes intermédios não use a totalidade de *acm\_ccs*, que tem 2114 linhas! Use, por exemplo, *take 10 acm\_ccs*, como se mostrou acima.

### Problema 3

O [tapete de Sierpinski](#) é uma figura geométrica [fractal](#) em que um quadrado é subdividido recursivamente em sub-quadrados. A construção clássica do tapete de Sierpinski é a seguinte: assumindo um quadrado de lado  $l$ , este é subdividido em 9 quadrados iguais de lado  $l/3$ , removendo-se o quadrado central. Este passo é depois repetido sucessivamente para cada um dos 8 sub-quadrados restantes (Fig. 2).

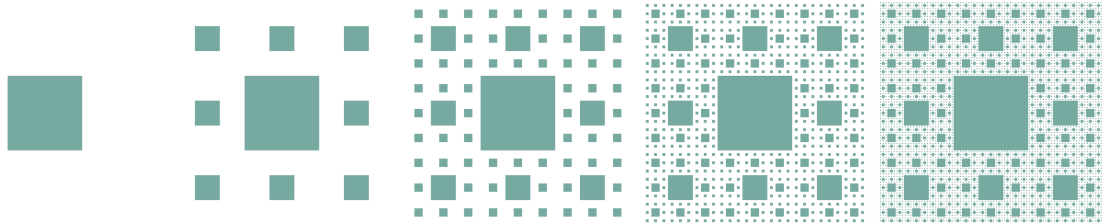


Figura 2: Construção do tapete de Sierpinski com profundidade 5.

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

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

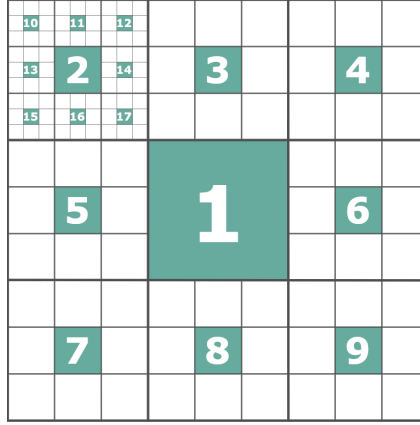


Figura 3: Tapete de Sierpinski com profundidade 2 e com os quadrados enumerados.

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

```

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<sup>4</sup> corresponder à árvore da figura 4.

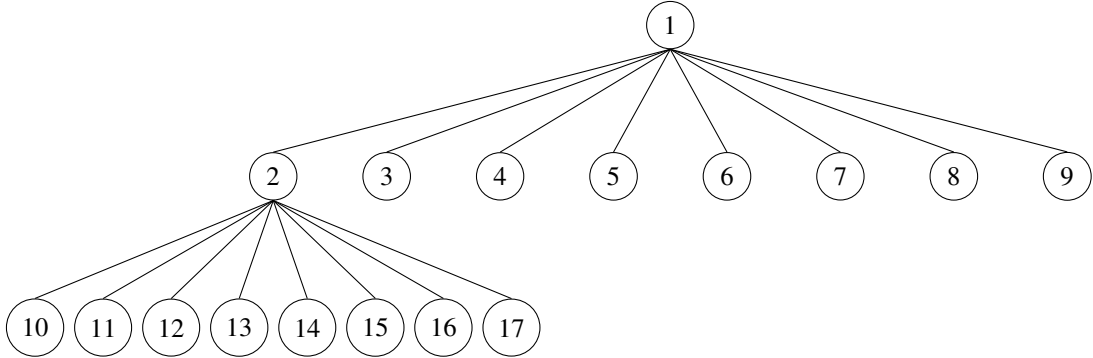


Figura 4: Possível árvore de suporte para a construção da fig. 3.

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.

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

```

squares :: (Square, Int) → 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 → [a]

```

<sup>4</sup>A ordem dos filhos não é relevante.

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

$$\begin{aligned} \text{sierpinski} &:: (\text{Square}, \text{Int}) \rightarrow [\text{Square}] \\ \text{sierpinski} &= \llbracket \text{gr2l}, \text{gsq} \rrbracket_k \end{aligned}$$

#### Trabalho a fazer:

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

```
sierp4 = drawSq (sierpinski (((0,0),32),3))
constructSierp5 = do drawSq (sierpinski (((0,0),32),0))
  await
  drawSq (sierpinski (((0,0),32),1))
  await
  drawSq (sierpinski (((0,0),32),2))
  await
  drawSq (sierpinski (((0,0),32),3))
  await
  drawSq (sierpinski (((0,0),32),4))
  await
```

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 → IO [()]
constructSierp = present · carpets
```

**Dica:** a função *constructSierp* será um hilomorfismo de listas, cujo anamorfismo *carpets* ::  $\text{Int} \rightarrow [[\text{Square}]]$  constrói, recebendo como parâmetro a profundidade  $n$ , a lista com todos os tapetes de profundidade  $1..n$ , e o catamorfismo *present* ::  $[[\text{Square}]] \rightarrow \text{IO} [()]$  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*<sup>5</sup> 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.

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).

<sup>5</sup>Os *rankings* obtidos [aqui](#) foram escalados e arredondados.

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.

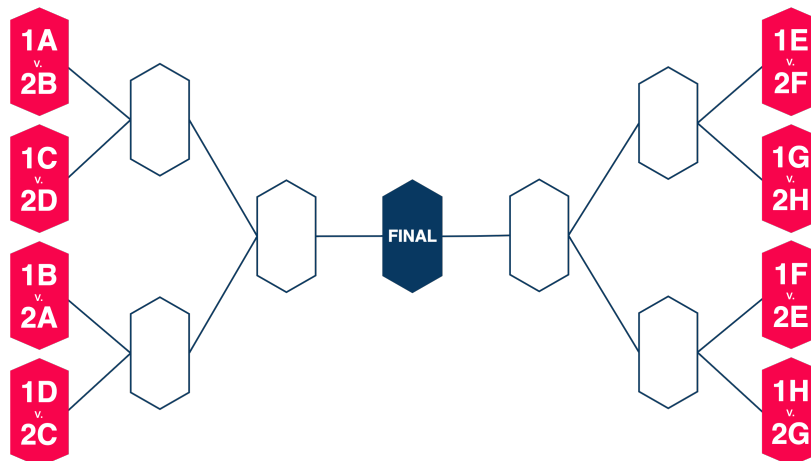


Figura 5: O “mata-mata”

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 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 · 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] → LTree Team
groupStage = initKnockoutStage · simulateGroupStage · genGroupStageMatches
```

Começamos então por definir a função *genGroupStageMatches* que gera os jogos da fase de grupos:

```
genGroupStageMatches :: [Group] → [[Match]]
genGroupStageMatches = map generateMatches
```

onde

```
type Match = (Team, Team)
```

Ora, sabemos que nos foi dada a função

```
gsCriteria :: Match → 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]] → [[Team]]
simulateGroupStage = 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 · consolidate · (>>=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 = ([gt]) · arrangement
```

Trabalho a fazer:

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

$$\begin{aligned} \text{consolidate}' &:: (Eq\ a, Num\ b) \Rightarrow [(a, b)] \rightarrow [(a, b)] \\ \text{consolidate}' &= \ll cgene \end{aligned}$$

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

### Versão probabilística

Nesta versão, mais realista,  $\text{gsCriteria} :: Match \rightarrow (Maybe\ Team)$  dá lugar a

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

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  $\text{Dist}$ ? 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  $\text{gsCriteria}$  virar monádica (em  $\text{Dist}$ ) na arquitetura geral da versão anterior. Há que reduzir esse impacto ao mínimo, escrevendo-se tão pouco código quanto possível!*

Todos relembaram 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 :: Dist Team
pwinner = pwcup groups
pwcup = pknockoutStage • pgroupStage
```

E entregou ainda a versão probabilística do “mata-mata”:

```
pknockoutStage = mcataLTree' [return, pkoCriteria]
mcataLTree' g = k where
  k (Leaf a) = g1 a
  k (Fork (x, y)) = mmbin g2 (k x, k y)
  g1 = g · i1
  g2 = g · i2
```

A sub-equipa dos alunos também já adiantou trabalho,

$$\text{pgroupStage} = \text{pinitKnockoutStage} \bullet \text{psimulateGroupStage} \cdot \text{genGroupStageMatches}$$

mas faltam ainda  $\text{pinitKnockoutStage}$  e  $\text{pgroupWinners}$ , esta usada em  $\text{psimulateGroupStage}$ , 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`<sup>6</sup> 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 utilitário `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.

### 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.

---

<sup>6</sup>O sufixo ‘lhs’ quer dizer *literate Haskell*.

## A.2 Como exprimir cálculos e diagramas em LaTeX/lhs2tex

Como primeiro exemplo, estudar o texto fonte deste trabalho para obter o efeito:<sup>7</sup>

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

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

$$\begin{array}{ccc}
 \mathbb{N}_0 & \xleftarrow{\text{in}} & 1 + \mathbb{N}_0 \\
 \downarrow \langle g \rangle & & \downarrow id + \langle g \rangle \\
 B & \xleftarrow{g} & 1 + B
 \end{array}$$

## 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.<sup>8</sup>

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:

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

Obter-se-á de imediato

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

usando as regras seguintes:

- O corpo do ciclo *loop* terá tantos argumentos quanto o número de funções mutuamente recursivas.
- Para as variáveis escolhem-se os próprios nomes das funções, pela ordem que se achar conveniente.<sup>9</sup>
- Para os resultados vão-se buscar as expressões respectivas, retirando a variável  $n$ .
- Em *init* colecionam-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<sup>10</sup>, de  $f\ x = ax^2 + bx + c$  derivam-se duas funções mutuamente recursivas:

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

<sup>7</sup>Exemplos tirados de [?].

<sup>8</sup>Lei (3.95) em [?], página 112.

<sup>9</sup>Podem obviamente usar-se outros símbolos, mas numa primeira leitura dá jeito usarem-se tais nomes.

<sup>10</sup>Secção 3.17 de [?] e tópico [Recursividade mútua](#) nos vídeos de apoio às aulas teóricas.

$$k\ 0 = a + b$$

$$k\ (n + 1) = k\ n + 2\ a$$

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

```
f' a b c = π1 · for loop init where
  loop (f, k) = (f + k, k + 2 * a)
  init = (c, a + b)
```

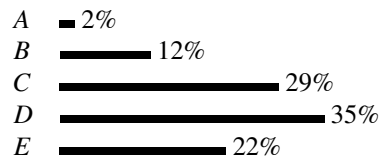
## C O mónade das distribuições probabilísticas

Mónades são funtores 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

**newtype** `Dist a = D { unD :: [(a, ProbRep)] }` (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  $d :: \text{Dist } a$  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

```
d1 :: Dist Char
d1 = D [( 'A', 0.02), ( 'B', 0.12), ( 'C', 0.29), ( 'D', 0.35), ( 'E', 0.22)]
```

que o [GHCI](#) mostrará assim:

```
'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*,

```
d2 = uniform (words "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.

```
d3 = normal [10..20]
```

etc.<sup>11</sup> `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  $g : A \rightarrow \text{Dist } B$  e  $f : B \rightarrow \text{Dist } C$  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.

<sup>11</sup>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 [?].

## D Código fornecido

### Problema 1

Alguns testes para se validar a solução encontrada:

```
test a b c = map (fbl a b c) x ≡ map (f a b c) x where x = [1..20]
test1 = test 1 2 3
test2 = test (-2) 1 5
```

Fizemos alguns testes tentamos gerar os primeiros 25 números da nossa sequência de fibonnaci.

Utilizando a função fornecida demora 5.53 secs, e usa map (fbl 1 1 1) [1..25] (5.53 secs, 3,187,080,168 bytes)  
map (f 1 1 1) [1..25] (0.01 secs, 533,240 bytes)

### 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).<sup>12</sup>

O resultado final

```
acm_xls = post acm_tree
```

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

### Problema 3

Função para visualização em [SVG](#):

```
drawSq x = picd'' [Svg.scale 0.44 (0,0) (x >>= sq2svg)]
sq2svg (p,l) = (color "#67AB9F" · 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),
```

---

<sup>12</sup>Quer dizer, o número total de nodos e folhas é 2114, o número de linhas do texto dado.

```

("England",4.7),
("France",4.8),
("Germany",4.5),
("Ghana",3.8),
("Iran",4.2),
("Japan",4.2),
("Korea Republic",4.2),
("Mexico",4.5),
("Morocco",4.2),
("Netherlands",4.6),
("Poland",4.2),
("Portugal",4.6),
("Qatar",3.9),
("Saudi Arabia",3.9),
("Senegal",4.3),
("Serbia",4.2),
("Spain",4.7),
("Switzerland",4.4),
("Tunisia",4.1),
("USA",4.4),
("Uruguay",4.5),
("Wales",4.3)]

```

Geração dos jogos da fase de grupos:

*generateMatches* = *pairup*

Preparação da árvore do “mata-mata”:

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

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

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

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

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

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

*koCriteria* =  $s \cdot \langle id \times id, rank \times rank \rangle$  **where**  
 $s((s_1, s_2), (r_1, r_2)) = \text{let } d = r_1 - r_2 \text{ in}$   
**if**  $d \equiv 0$  **then**  $s_1$   
**else if**  $d > 0$  **then**  $s_1$  **else**  $s_2$

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

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

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

*pkoCriteria*  $(e_1, e_2) = D[(e_1, 1 - r_2 / (r_1 + r_2)), (e_2, 1 - r_1 / (r_1 + r_2))]$  **where**  
 $r_1 = \text{rank } e_1$   
 $r_2 = \text{rank } e_2$

Versão probabilística da simulação da fase de grupos:<sup>13</sup>

```
psimulateGroupStage :: [[Match]] → Dist [[Team]]
psimulateGroupStage = trim · map (pgroupWinners pgsCriteria)
trim = top 5 · sequence · map (filterP · norm) where
  filterP (D x) = D [(a,p) | (a,p) ← x, p > 0.0001]
  top n = vec2Dist · take n · reverse · presort π2 · unD
  vec2Dist x = D [(a,n/t) | (a,n) ← x] where t = sum (map π2 x)
```

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

```
pwinner :: Dist Team
pwinner = mbin f x >>= pknockoutStage where
  f (x,y) = initKnockoutStage (x ++ y)
  x = ⟨g · take 4, g · drop 4⟩ groups
  g = psimulateGroupStage · genGroupStageMatches
```

Auxiliares:

```
best n = map π1 · take n · reverse · presort π2
consolidate :: (Num d, Eq d, Eq b) ⇒ [(b,d)] → [(b,d)]
consolidate = map (id × sum) · collect
collect :: (Eq a, Eq b) ⇒ [(a,b)] → [(a,[b])]
collect x = nub [k ↦ [d' | (k',d') ← x, k' ≡ k] | (k,d) ← x]
```

Função binária monádica *f*:

```
mmbin :: Monad m ⇒ ((a,b) → m c) → (m a, m b) → m c
mmbin f (a,b) = do { x ← a; y ← b; f (x,y) }
```

Monadificação de uma função binária *f*:

```
mbin :: Monad m ⇒ ((a,b) → c) → (m a, m b) → m c
mbin = mmbin · (return ·)
```

Outras funções que podem ser úteis:

```
(f 'is' v) x = (f x) ≡ 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.

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

### Problema 1

Funções auxiliares pedidas:

```
fa a b c 0 = 0
fa a b c 1 = 1
fa a b c 2 = 1
fa a b c (n+3) = a * (fa a b c (n+2)) + b * (fa a b c (n+1)) + c * (fa a b c n)
initial = ((1,1),0)
```

<sup>13</sup>Faz-se “trimming” das distribuições para reduzir o tempo de simulação.

$loop\ a\ b\ c\ ((x,y),z) = ((a*x + b*y + c*z,x),y)$   
 $wrap = \pi_2$

Usando o comando `:set +s`, comando que inicia o cronómetro e conta o tempo de cada função executada, executamos alguns testes tentamos gerar os primeiros 25 números da nossa sequência de fibonnaci.

Utilizando a função fornecida pela equipa docente, a função demora 5.53 secs para gerar uma resposta.

```
map (fbl 1 1 1) [1..25]
(5.53 secs, 3,187,080,168 bytes)
```

Utilizando a nossa função, obtivemos uma resposta instantanea de 0.01 secs.

```
map (f 1 1 1) [1..25]
(0.01 secs, 533,240 bytes)
```

$$(2) \quad \left\{ \begin{array}{l} \cdot \text{ } lcbr(f0 = 1)(f(n+1) = (a * fn + b * gn) + c * bn)) \text{ } (lcbr(g0 = 1)(g(n+1) = fn)) \text{ } (lcbr(h0 = 1)(h(n+1) = gn)) \\ \cdot \end{array} \right.$$

## Problema 2

Gene de *tax*:

Esta foi a nossa primeira tentativa que fizemos desta função geradora. Foi feita sem tentar utilizar os conceitos dados na Unidade Curricular de Cálculo de Programas. A função *árvore* é o *tax*.

*ce* é uma função que conta os espaços seguidos no início de uma string.

$$\begin{array}{ccc} A^* & \xrightarrow{psi} & 1 + A \\ \downarrow ce & & \downarrow id+ce \\ \mathbb{N} & \xleftarrow{in_{Nat}} & 1 + \mathbb{N} \end{array}$$

$arvore\ [x] = Var\ x$   
 $arvore\ l = Term\ (Var\ a)\ filhos$   
**where**  $(a,b) = splitp\ l$   
 $filhos = [arvore\ pf \leftarrow b]$   
 $splitp\ (a:l) = (a, groupBy\ (\backslash y \rightarrow ce\ y > \gamma)\ l)$   
**where**  $\gamma = ce\ a + 4$  -- indentação do pai + 4

ou então em notação point free:

$$arvore = \widehat{Term} \cdot (Var \times (\text{map } arvore)) \cdot splitp$$

Apos isso, conseguimos reconhecer alguns o o Functor das Rose Trees ( $id \times map\ f$ ) e pudemos ver que a função *arvorea* é um anamorfismo das *Rose Tree*.

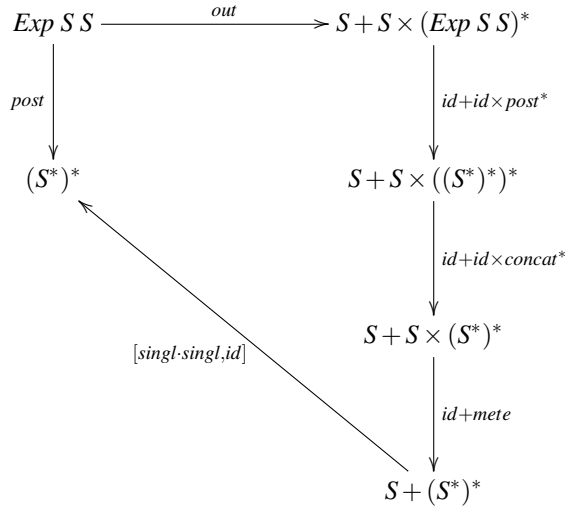
Versão final:

```

splitp (a,l) = (a,groupBy (>γ) · ce l)
  where
    γ = ce a + 4  -- indentação do pai + 4
    ce = anaNat ψ  -- conta espaços
    ψ (' ' : t) = i2 t
    ψ _ = i1 ()
    gene = (id + splitp) · out

```

Função de pós-processamento: Começamos por tentar escrever a função *post* com um catamorfismo de *Exp*.



Temos que o gene da função *post* é dado por:

$$\begin{aligned}
gpost &= [singl \cdot singl, id] \cdot (id + mete) \cdot (id + id \times concat) \\
&\equiv \{ (22), (1) \times 2 \} \\
gpost &= [singl \cdot singl, mete] \cdot (id + id \times concat) \\
&\equiv \{ (22), (1) \times 2 \} \\
gpost &= [singl \cdot singl, mete \cdot id \times concat] \\
&\square
\end{aligned}$$

*mete* é uma função que coloca dentro da lista e à cabeça de todos os elementos um outro elemento.

```

mete (a,b) = [a] : map (a:) b
gpost = [singl · singl, mete · (id × concat)]
post = (gpost)Exp

```

Após uma análise às bibliotecas fornecidas encontramos uma função muito semelhante à *mete*, a função *prefixes*. Chegamos a conclusão que poderíamos tornar mais económica a nossa função se utilizássemos a *prefixes*. Chegamos a seguinte definição de *post*, a função *tail* serve para remover a lista vazia à cabeça da lista:

```

post = tail · prefixes · nodes

```



### Problema 3

Fizemos o seguinte diagrama da função squares que é um anamorfismo de RoseTrees

$$\begin{array}{ccc}
 S \times \mathbb{N} & \xrightarrow{\text{quadrado}} & S \times (S \times \mathbb{N})^* \\
 \downarrow \text{square} & & \downarrow \text{id} \times \text{square}^* \\
 \text{RoseT } S \ S & \xleftarrow{\text{in}_R} & S \times (\text{RoseT } S \ S)^*
 \end{array}$$

A nossa função quadrado gera as nova coordenadas dos proximos quadrados e desenha o quadrado do meio. Esta é a nossa primeira defenição da função quadrado. Quando não há mais níveis para desenhar quando  $n == 0$ , retornamos lista vazia. Se não, retornamos uma lista com todos os quadrados do próximo nível.

```

quadrado2 (((x,y),l),n)
  | n == 0 = (meio, [])
  | n > 0 = (meio, lista)
where t = l / 3
      meio = ((x+t,y+t),t)
      lista =

```

FIXME Criamos uma função auxiliar beta que dado um elemento e uma lista cria pares e coloca a direita de todos os elementos esse tal elemento

$$\beta = \text{map} \cdot \text{flip } \overline{\text{id}}$$

Exemplo:

```

β "asd" [1..5]
[(1, "asd"), (2, "asd"), (3, "asd"), (4, "asd"), (5, "asd")]

```

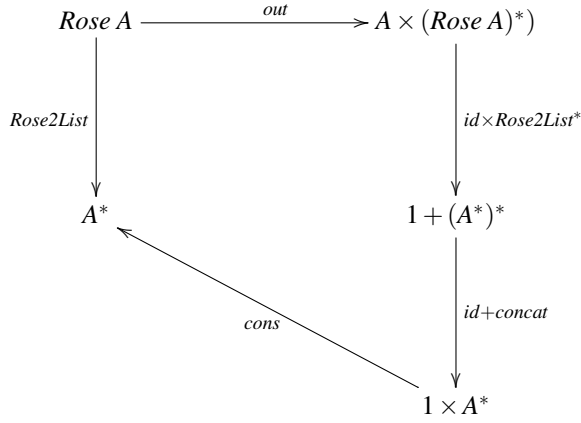
Conseguimos substituir muitos dos elementos repetidos do lado direito dos tuplos com a função beta. Por exemplo se tenho uma lista de coordenadas do próximo nível de quadrados, posso utilizar essa função para colocar a direita de todos os elementos dessa lista a largura, formando uma lista de quadrados.

```

quadrado (((x,y),l),n)
  | n == 0 = (meio, [])
  | n > 0 = (meio, β (n-1) $ β t $ β y k ++ β b k ++ β c (init k))
where t = l / 3
      meio = ((x+t,c),t)
      fun = [(+(2*t)), (+t)]
      k = x : map ($x) fun
      [b,c] = map ($y) fun
gsq = quadrado
squares = [gsq]_R

```

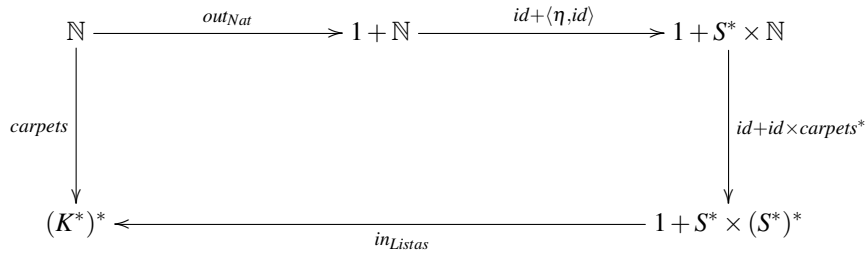
Diagrama da função catamorfismo que converte RoseTrees para lista



Temos que o gene do catamorfismo é:

$$\begin{aligned}
 \text{gr2l} &= \text{cons} \cdot (\text{id} \times \text{concat}) \\
 \text{rose2List} &= \llbracket \text{gr2l} \rrbracket_R
 \end{aligned}$$

Diagrama da construção da lista de listas de quadrados da função carpets



A função eta gera uma lista de quadrados para uma determinada profundidade.

$$\begin{aligned}
 \eta &= \text{sierpinski} \cdot \overline{\text{id}} \ ((0,0), 32) \\
 \text{carpets} &= \llbracket (\text{id} + \langle \eta, \text{id} \rangle) \cdot \text{outNat} \rrbracket
 \end{aligned}$$

Para definir a função carpets começamos por definir uma função recursiva daquilo que queríamos fazer também fizemos um função auxiliar theta. Theta desenha um nível de quadrados e espera 1 segundo.

```

θ l = do
  drawSq l
  await
present [] = return []
present (h:t) = do
  present t
  θ h
  return []

```

Depois tentamos fazer um função present mas mudamos assinatura porque não percebemos o porque de a função devolver IO[()] achamos mais pertinente retornar IO().

$$\begin{array}{ccc}
(K^*)^* & \xrightarrow{out_{Listas}} & 1 + K^* \times (K)^* \\
\downarrow present2 & & \downarrow id + id \times present2 \\
IO() & \xleftarrow{[return, \theta \cdot \pi_1]} & 1 + K^* \times IO()
\end{array}$$

$\theta = (>>await) \cdot (drawSq)$   
 $present2 :: [Square] \rightarrow IO ()$   
 $present2 = \llbracket [return, \theta \cdot \pi_1] \rrbracket$

Esta função não funciona porque como o haskell é preguiçoso, quando fazemos p1 ele não vai calcular o resultado da cauda, assim se pedir para imprimir vários níveis ele só imprime o último. Para colocar isto a funcionar teríamos que o forçar a não deitar fora a cauda. Esta experiência foi o suficiente para desboquelar como fazer com o resultado de saída a ser IO[()].

Usando o tipo IO[()] podemos resolver este problema no gene da função juntando a cabeça (do tipo IO()) com a cauda recursiva já calculada ficando com o tipo IO[()] Precisamos então de arranjar uma função que junte um IO() com IO[()]: um cons monádico, a função cons $\flat$ .

Também invertamos a lista que passamos à função presente para quando for desenhar, desenhar pela ordem certa, do menor nível para o maior.

Aqui está o diagrama que mostra a função present em formato catamorfismo:

$$\begin{array}{ccc}
(S^*)^* & \xrightarrow{out} & 1 + S^* \times (S^*)^* \\
\downarrow present & & \downarrow id + id \times present \\
IO[()] & & 1 + S^* \times IO[()] \\
\uparrow [return \cdot return, id] & & \downarrow id + teta \times id \\
1 + IO[()] & \xleftarrow{1 + consb} & 1 + IO() \times IO[()]
\end{array}$$

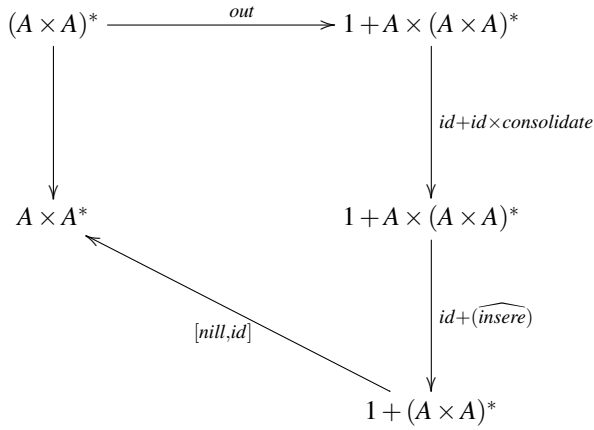
$\theta = (>>await) \cdot (drawSq)$   
 $present = \llbracket [return \cdot return, consb \cdot (\theta \times id)] \rrbracket \cdot reverse$   
 $consb(a, b) = \mathbf{do}$   
 $\quad x \leftarrow a$   
 $\quad y \leftarrow b$   
 $\quad return \$ cons(x, y)$

Utilizamos return após return porque no caso de retornar o tipo 1 é preciso monadificá-lo 2 vezes, usando o monad da listas e depois com o monad IO.

## Problema 4

### Versão não probabilística

Defenimos o consolidate com um catamorfismo



Gene de *consolidate'*:

O gene do catamorfismo de consolidate é:

$$\begin{aligned}
 cgene &= [\text{nil}, \text{id}] \cdot \text{id} + \widehat{\text{insere}} \\
 &\equiv \{ (22); (1) \times 2 \} \\
 cgene &= [\text{nil}, \widehat{\text{insere}}] \\
 &\square
 \end{aligned}$$

*insere* é uma função que insere um par numa lista de pares onde se já existir um com a primeira componente do tuplo igual soma a segunda componente com uma já existente

```

insere a [] = [a]
insere (a,b) ((c,d):e)
  | a == c = (a,b+d):e -- insere (a,b) e
  | otherwise = (c,d):insere (a,b) e
-- fazer isto como catalist FIXME
cgene = [nil, insere]

```

Geração dos jogos da fase de grupos:

```

pairup [] = []
pairup (a:l) = beta a l ++ pairup l
pontos (a,b) = maybe [(a,1),(b,1)] vs where
  vs x | x == a = [(a,3),(b,0)]
       | x == b = [(a,0),(b,3)]
matchResult :: (Match -> Maybe Team) -> Match -> [(Team,Int)]
matchResult f m = pontos m $ f m

```

Pensamos logo em usar o gene do ana do mergesort, *lsplit* que parte uma lista em 2 mas não funciona porque queremos manter a mesma ordem relativa entre os elementos quando parte, então criamos uma função *half* que parte uma lista em 2

$$\text{half} = \widehat{\text{splitAt}} \cdot \langle \text{flip} \cdot \div \cdot 2 \cdot \text{length}, \text{id} \rangle$$

Ou então usando o *bind* dos monads

$$\text{half} = \text{splitAt} \ll \text{flip} \cdot \div \cdot 2 \cdot \text{length}$$

$$\begin{array}{ccccc}
A^* & \xrightarrow{\text{out}} & A + A \times A^* & \xrightarrow{id + half \cdot cons} & A + A^* \times A^* \\
\downarrow \text{ana} & & & & \downarrow id + ana^2 \\
LTreeA & \xleftarrow{inLTree} & & & A + LTreeA^2
\end{array}$$

$$glt = (id + half \cdot cons) \cdot out$$

## Versão probabilística

```

pinitKnockoutStage = return · initKnockoutStage
-- psimulateGroupStage :: [[Match]] -> Dist [[Team]]
-- groupWinners criteria = best 2 . consolidate . (λc= matchResult criteria)
-- pgsCriteria :: Match -> Dist (Maybe Team)
-- groupWinners :: (Match -> Maybe Team) -> [Match] -> [Team]
-- groupWinners criteria = best 2 . consolidate . (λc= matchResult criteria)
-- pgroupWinners :: (Match -> Dist (Maybe Team)) -> [Match] -> Dist [Team]
pgroupWinners pcriteria = ⊥ -- return (best 2 . consolidate) . (λc= pmatchResult pcriteria)
-- pgsCriteria :: Match -> Dist (Maybe Team)
-- pontos :: (Num b, Eq a) => (a, a) -> Maybe a -> [(a, b)]
-- pmatchResult = undefined
pmatchResult :: (Match → Dist (Maybe Team)) → Match → Dist [(Team, Int)]
pmatchResult criteria m = do
  e ← criteria m -- Dist (Maybe Team)
  pontos m e -- [(Team, Int)]
-- return e

```

# Índice

L<sup>A</sup>T<sub>E</sub>X, [10](#)

    bibtex, [10](#)

    lhs2TeX, [10](#)

    makeindex, [10](#)

Cálculo de Programas, [1](#), [3](#), [10](#), [11](#)

    Material Pedagógico, [9](#)

        Exp.hs, [2](#), [3](#), [13](#)

        LTree.hs, [6–8](#)

        Rose.hs, [4](#)

Combinador “pointfree”

*either*, [7](#), [9](#)

Fractal, [3](#)

    Tapete de Sierpinski, [3](#)

Função

$\pi_1$ , [10](#), [11](#), [15](#)

$\pi_2$ , [10](#), [15](#)

*for*, [2](#), [11](#)

*length*, [13](#)

*map*, [7](#), [8](#), [13–15](#)

Functor, [5](#), [8](#), [9](#), [11](#), [12](#), [15](#), [16](#)

Haskell, [1](#), [10](#)

    Biblioteca

        PFP, [12](#)

        Probability, [12](#)

    interpretador

        GHCi, [10](#), [12](#)

    Literate Haskell, [9](#)

Números naturais ( $\mathbb{N}$ ), [11](#)

Programação

    dinâmica, [11](#)

    literária, [9](#)

SVG (Scalable Vector Graphics), [13](#)

U.Minho

    Departamento de Informática, [1](#), [2](#)