Universidade do Minho
Escola de Engenharia

Cálculo de Programas

Trabalho Prático (2023/24)

Lic. em Engenharia Informática

Grupo G15

a100714 Diogo Santos
a100608 Luís Ribeiro
a100647 Martim Félix

janeiro 2024

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 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.

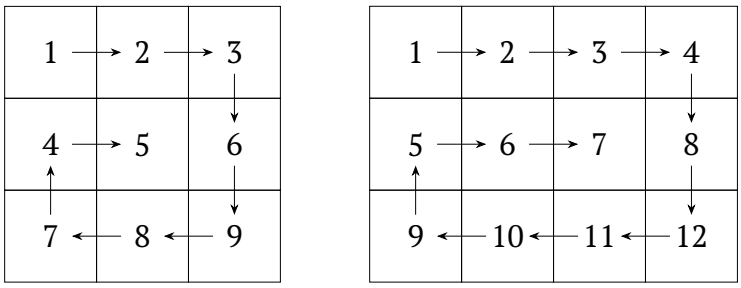
Antes de abodarem os problemas propostos no trabalho, os grupos devem ler com atenção o anexo A onde encontrarão as instruções relativas ao sofware a instalar, etc.

Valoriza-se a escrita de *pouco* código que corresponda a soluções simples e elegantes que utilizem os combinadores de ordem superior estudados na disciplina.

Problema 1

Este problema, retirado de um *site* de exercícios de preparação para entrevistas de emprego, tem uma formulação simples:

> *Dada uma matriz de uma qualquer dimensão, listar todos os seus elementos rodados em espiral.*
>
> *Por exemplo, dadas as seguintes matrizes:*



> *dever-se-á obter, respetivamente,* $[1,2,3,6,9,8,7,4,5]$ *e* $[1,2,3,4,8,12,11,10,9,5,6,7]$.
>
> □

Valorizar-se-ão as soluções *pointfree* que empreguem os combinadores estudados na disciplina, e.g. $f \cdot g$, $\langle f,g\rangle$, $f \times g$, $[f,g]$, $f+g$, bem como catamorfismos e anamorfismos.

Recomenda-se a escrita de *pouco* código e de soluções simples e fáceis de entender. Recomenda-se que o código venha acompanhado de uma descrição de como funciona e foi concebido, apoiado em diagramas explicativos. Para instruções sobre como produzir esses diagramas e exprimir raciocínios de cálculo, ver o anexo D.

Problema 2

Este problema, que de novo foi retirado de um *site* de exercícios de preparação para entrevistas de emprego, tem uma formulação muito simples:

Inverter as vogais de um string.

Esta formulação deverá ser generalizada a:

Inverter os elementos de uma dada lista que satisfazem um dado predicado.

Valorizam-se as soluções tal como no problema anterior e fazem-se as mesmas recomendações.

Problema 3

Sistemas como chatGPT etc baseiam-se em algoritmos de aprendizagem automática que usam determinadas funções matemáticas, designadas *activation functions* (AF), para modelar aspectos não lineares do mundo real. Uma dessas AFs é a tangente hiperbólica, definida como o quociente do seno e coseno hiperbólicos,

$$\tanh x = \frac{\sinh x}{\cosh x} \tag{1}$$

podendo estes ser definidos pelas seguintes séries de Taylor:

$$\begin{aligned} \sum_{k=0}^{\infty} \frac{x^{2k+1}}{(2k+1)!} &= \sinh x \\ \sum_{k=0}^{\infty} \frac{x^{2k}}{(2k)!} &= \cosh x \end{aligned} \tag{2}$$

Interessa que estas funções sejam implementadas de forma muito eficiente, desdobrando-as em operações aritméticas elementares. Isso pode ser conseguido através da chamada programação dinâmica que, em Cálculo de Programas, é feita de forma *correct-by-construction* derivando-se ciclos-**for** via lei de recursividade mútua generalizada a tantas funções quanto necessário — ver o anexo E.

O objectivo desta questão é codificar como um ciclo-for (em Haskell) a função

$$snh\ x\ i = \sum_{k=0}^{i} \frac{x^{2k+1}}{(2k+1)!} \tag{3}$$

que implementa $sinh\ x$, uma das funções de $tanh\ x$ (1), através da soma das i primeiras parcelas da sua série (2).

Deverá ser seguida a regra prática do anexo E e documentada a solução proposta com todos os cálculos que se fizerem.

Problema 4

Uma empresa de transportes urbanos pretende fornecer um serviço de previsão de atrasos dos seus autocarros que esteja sempre actual, com base em *feedback* dos seus paassageiros. Para isso, desenvolveu uma *app* que instala num telemóvel um botão que indica coordenadas GPS a um serviço central, de forma anónima, sugerindo que os passageiros o usem preferencialmente sempre que o autocarro onde vão chega a uma paragem.

Com base nesses dados, outra funcionalidade da *app* informa os utentes do serviço sobre a probabilidade do atraso que possa haver entre duas paragens (partida e chegada) de uma qualquer linha.

Pretende-se implementar esta segunda funcionalidade assumindo disponíveis os dados da primeira. No que se segue, ir-se-á trabalhar sobre um modelo intencionalmente *muito simplificado* deste sistema, em que se usará o mónade das distribuições probabilísticas (ver o anexo F). Ter-se-á, então:

- paragens de autocarro

  $$\textbf{data}\ Stop = S0 \mid S1 \mid S2 \mid S3 \mid S4 \mid S5\ \textbf{deriving}\ (Show, Eq, Ord, Enum)$$

  que formam a linha $[S0\,..\,S5]$ assumindo a ordem determinada pela instância de $Stop$ na classe $Enum$;

- segmentos da linha, isto é, percursos entre duas paragens consecutivas:

  $$\textbf{type}\ Segment = (Stop, Stop)$$

- os dados obtidos a partir da *app* dos passageiros que, após algum processamento, ficam disponíveis sob a forma de pares *(segmento, atraso observado)*:

  $$dados :: [(Segment, Delay)]$$

  (Ver no apêndice G, página 9, uma pequena amostra destes dados.)

A partir destes dados, há que:

- gerar a base de dados probabilística

  $$db :: [(Segment, \text{Dist}\ Delay)]$$

  que regista, estatisticamente, a probabilidade dos atrasos ($Delay$) que podem afectar cada segmento da linha. Recomenda-se aqui a definição de uma função genérica

  $$mkdist :: Eq\ a \Rightarrow [a] \to \text{Dist}\ a$$

  que faça o sumário estatístico de uma qualquer lista finita, gerando a distribuição de ocorrência dos seus elementos.

- com base em db, definir a função probabilística

  $$delay :: Segment \to \text{Dist}\ Delay$$

  que dará, para cada segmento, a respectiva distribuição de atrasos.

Finalmente, o objectivo principal é definir a função probabilística:

$$pdelay :: Stop \to Stop \to \text{Dist}\ Delay$$

$pdelay\ a\ b$ deverá informar qualquer utente que queira ir da paragem a até à paragem b de uma dada linha sobre a probabilidade de atraso acumulado no total do percurso $[a\,..\,b]$.

Valorizar-se-ão as soluções que usem funcionalidades monádicas genéricas estudadas na disciplina e que sejam elegantes, isto é, poupem código desnecessário.

Anexos

A Natureza do trabalho a realizar

Este trabalho teórico-prático deve ser realizado por grupos de 3 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.

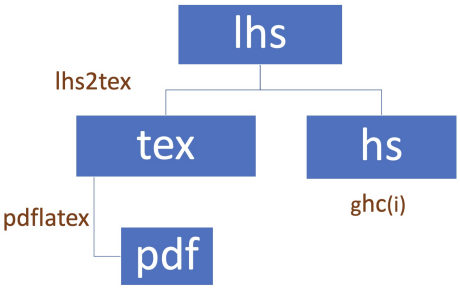
Para cumprir de forma integrada os objectivos do trabalho vamos recorrer a uma técnica de programação dita "literária" [2], cujo princípio base é o seguinte:

> *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 `cp2324t.pdf` que está a ler é já um exemplo de programação literária: foi gerado a partir do texto fonte `cp2324t.lhs`[1] que encontrará no material pedagógico desta disciplina descompactando o ficheiro `cp2324t.zip`.

Como se mostra no esquema abaixo, de um único ficheiro (*lhs*) gera-se um PDF ou faz-se a interpretação do código Haskell que ele inclui:



Vê-se assim que, para além do GHCi, serão necessários os executáveis pdflatex e lhs2TeX. Para facilitar a instalação e evitar problemas de versões e conflitos com sistemas operativos, é recomendado o uso do Docker tal como a seguir se descreve.

B Docker

Recomenda-se o uso do container cuja imagem é gerada pelo Docker a partir do ficheiro `Dockerfile` que se encontra na diretoria que resulta de descompactar `cp2324t.zip`. Este container deverá ser usado na execução do GHCi e dos comandos relativos ao LaTeX. (Ver também a `Makefile` que é disponibilizada.)

[1] O sufixo 'lhs' quer dizer *literate Haskell*.

Após instalar o Docker e descarregar o referido zip com o código fonte do trabalho, basta executar os seguintes comandos:

```
$ docker build -t cp2324t .
$ docker run -v ${PWD}:/cp2324t -it cp2324t
```

NB: O objetivo é que o container seja usado *apenas* para executar o GHCi e os comandos relativos ao LaTeX. Deste modo, é criado um *volume* (cf. a opção `-v ${PWD}:/cp2324t`) que permite que a diretoria em que se encontra na sua máquina local e a diretoria `/cp2324t` no container sejam partilhadas.

Pretende-se então que visualize/edite os ficheiros na sua máquina local e que os compile no container, executando:

```
$ lhs2TeX cp2324t.lhs > cp2324t.tex
$ pdflatex cp2324t
```

lhs2TeX é o pre-processador que faz "pretty printing" de código Haskell em LaTeX e que faz parte já do container. Alternativamente, basta executar

```
$ make
```

para obter o mesmo efeito que acima.

Por outro lado, o mesmo ficheiro `cp2324t.lhs` é executável e contém o "kit" básico, escrito em Haskell, para realizar o trabalho. Basta executar

```
$ ghci cp2324t.lhs
```

Abra o ficheiro `cp2324t.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.

C Em que consiste o TP

Em que consiste, então, o *relatório* a que se referiu acima? É a edição do texto que está a ser lido, preenchendo o anexo H 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 cp2324t.aux
$ makeindex cp2324t.idx
```

e recompilar o texto como acima se indicou. (Como já se disse, pode fazê-lo correndo simplesmente `make` no container.)

No anexo G disponibiliza-se algum código Haskell relativo aos problemas que são colocados. Esse anexo deverá ser consultado e analisado à medida que isso for necessário.

Deve ser feito uso da programação literária para documentar bem o código que se desenvolver, em particular fazendo diagramas explicativos do que foi feito e tal como se explica no anexo D que se segue.

D Como exprimir cálculos e diagramas em LaTeX/lhs2TeX

Como primeiro exemplo, estudar o texto fonte (lhs) do que está a ler[1] onde se obtém o efeito seguinte:[2]

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

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

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

E 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.[3]

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

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

Obter-se-á de imediato

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

usando as regras seguintes:

[1] Procure e.g. por `"sec:diagramas"`.

[2] Exemplos tirados de [3].

[3] Lei (3.95) em [3], página 110.

- 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.[1]
- 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[2], de $f\ x = ax^2 + bx + c$ derivam-se duas funções mutuamente recursivas:

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

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

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

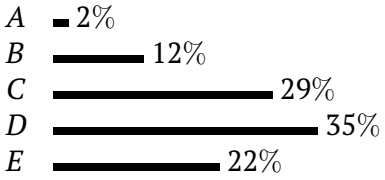
F O mónade das distribuições probabilísticas

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

$$\textbf{newtype}\ \text{Dist}\ a = D\ \{unD :: [(a, ProbRep)]\} \tag{4}$$

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

$$\begin{aligned} &d1 :: \text{Dist}\ Char \\ &d1 = D\ [('\texttt{A}', 0.02), ('\texttt{B}', 0.12), ('\texttt{C}', 0.29), ('\texttt{D}', 0.35), ('\texttt{E}', 0.22)] \end{aligned}$$

que o GHCi mostrará assim:

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

[2] 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*,

$$d2 = uniform\ (words\ \texttt{"Uma frase de cinco palavras"})$$

isto é

```
   "Uma"  20.0%
 "cinco"  20.0%
    "de"  20.0%
 "frase"  20.0%
"palavras"  20.0%
```

distribuição *normais*, eg.

$$d3 = normal\ [10\,..\,20]$$

etc.[1] Dist forma um **mónade** cuja unidade é $return\ a = D\ [(a,1)]$ e cuja composição de Kleisli é (simplificando a notação)

$$(f \bullet g)\ a = [(y, q*p)\mid (x,p) \leftarrow g\ a, (y,q) \leftarrow f\ x]$$

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

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

G Código fornecido

Problema 1

$$m1 = [[1,2,3],[4,5,6],[7,8,9]]$$
$$m2 = [[1,2,3,4],[5,6,7,8],[9,10,11,12]]$$
$$m3 = words\ \texttt{"Cristina Monteiro Carvalho Sequeira"}$$
$$test1 = matrot\ m1 \equiv [1,2,3,6,9,8,7,4,5]$$
$$test2 = matrot\ m2 \equiv [1,2,3,4,8,12,11,10,9,5,6,7]$$
$$test3 = matrot\ m3 \equiv \texttt{"CristinaooarieuqeSCMonteirhlavra"}$$

Problema 2

$$test4 = reverseVowels\ \texttt{""} \equiv \texttt{""}$$
$$test5 = reverseVowels\ \texttt{"ácidos"} \equiv \texttt{"ocidás"}$$
$$test6 = reverseByPredicate\ even\ [1\,..\,20] \equiv [1,20,3,18,5,16,7,14,9,12,11,10,13,8,15,6,17,4,19,2]$$

[1] Para mais detalhes ver o código fonte de Probability, que é uma adaptação da biblioteca PFP ("Probabilistic Functional Programming"). Para quem quiser saber mais recomenda-se a leitura do artigo [1].

Problema 3

Nenhum código é fornecido neste problema.

Problema 4

Os atrasos, medidos em minutos, são inteiros:

$$\textbf{type } \mathit{Delay} = \mathbb{Z}$$

Amostra de dados apurados por passageiros:

$$\begin{aligned}\mathit{dados} = [&((\mathit{S0},\mathit{S1}),0),((\mathit{S0},\mathit{S1}),2),((\mathit{S0},\mathit{S1}),0),((\mathit{S0},\mathit{S1}),3),((\mathit{S0},\mathit{S1}),3),\\ &((\mathit{S1},\mathit{S2}),0),((\mathit{S1},\mathit{S2}),2),((\mathit{S1},\mathit{S2}),1),((\mathit{S1},\mathit{S2}),1),((\mathit{S1},\mathit{S2}),4),\\ &((\mathit{S2},\mathit{S3}),2),((\mathit{S2},\mathit{S3}),2),((\mathit{S2},\mathit{S3}),4),((\mathit{S2},\mathit{S3}),0),((\mathit{S2},\mathit{S3}),5),\\ &((\mathit{S3},\mathit{S4}),2),((\mathit{S3},\mathit{S4}),3),((\mathit{S3},\mathit{S4}),5),((\mathit{S3},\mathit{S4}),2),((\mathit{S3},\mathit{S4}),0),\\ &((\mathit{S4},\mathit{S5}),0),((\mathit{S4},\mathit{S5}),5),((\mathit{S4},\mathit{S5}),0),((\mathit{S4},\mathit{S5}),7),((\mathit{S4},\mathit{S5}),-1)]\end{aligned}$$

"Funcionalização" de listas:

$$\begin{aligned}&\mathit{mkf} :: \mathit{Eq}\ a \Rightarrow [(a,b)] \to a \to \mathit{Maybe}\ b\\ &\mathit{mkf} = \mathit{flip}\ \mathit{Prelude.lookup}\end{aligned}$$

Ausência de qualquer atraso:

$$\begin{aligned}&\mathit{instantaneous} :: \mathsf{Dist}\ \mathit{Delay}\\ &\mathit{instantaneous} = D\ [(0,1)]\end{aligned}$$

H 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 ao anexo, bem como diagramas e/ou outras funções auxiliares que sejam necessárias.

Importante: Não pode ser alterado o texto deste ficheiro fora deste anexo.

Problema 1

Análise do problema

Neste problema decidimos analisar bem as imagens de exemplo que apareciam no enunciado, pois achamos que com uma perceção visual do problema conseguimos mais rapidamente chegar a uma solução.

Após a dita análise conseguimos observar um certo **padrão** na travessia pedida, lembrando que uma matriz é apenas uma lista de listas:

1. A cabeça da lista mantém-se (e retira-se)
2. Os próximos elementos são os últimos das seguintes listas (e retira-se)
3. Inverte-se a ordem dos elementos de todas as sub-listas restantes, da própria matriz e volta-se ao passo 1.

Testemos então este **padrão** com o exemplo dado no enunciado, a matriz

$$mat = [[1,2,3],[4,5,6],[7,8,9]]$$

Mantemos a lista resultado na variável *result*:

1. A cabeça da lista mantém-se (e retira-se):

   result = [1,2,3] e *mat* = [[4,5,6], [7,8,9]]

2. Os próximos elementos são os últimos das seguintes listas (e retira-se)

   result = [1,2,3,6,9] e *mat* = [[4,5], [7,8]]

3. Inverte-se a ordem dos elementos de todas as sub-listas restantes, da própria matriz e volta-se ao passo 1.

   result mantém-se. Primeiro inverter a ordem das sub-listas: *mat* = [[5,4], [8,7]]. Depois da própria matriz: *mat* = [[8,7], [5,4]]

Ao voltar-se ao passo 1. e repetir o processo chega-se ao resultado final de *result* = [1,2,3,6,9,8,7,4,5] que é o esperado.

Resolução

Partindo agora para a resolução do problema, começamos por definir um catamorfismo que tratasse de nos retornar os últimos elementos das sub-listas, originando este diagrama:

$$\begin{array}{ccc} (A^*)^* & \underset{\text{in}}{\overset{\text{out}}{\rightleftarrows}} & 1 + A^* \times (A^*)^* \\ {\scriptstyle lasts}\downarrow & & \downarrow{\scriptstyle id+id\times lasts} \\ A^* & \xleftarrow[[nil,cons\cdot(last\times id)]]{} & 1 + A^* \times A^* \end{array} \qquad lasts = (\!|\, [nil, cons \cdot (last \times id)] \,|\!)$$

Agora precisamos de algo que nos retorne as sub-listas da matriz sem os últimos elementos, pois partimos sempre da mesma matriz independentemente das operações que aplicamos, e o catamorfismo anterior e qualquer outro que usemos não alteram a "referência" da matriz.

Deu origem a este diagrama de um catamorfismo:

$$\begin{array}{ccc} (A^*)^* & \underset{\text{in}}{\overset{\text{out}}{\rightleftarrows}} & 1 + A^* \times (A^*)^* \\ {\scriptstyle myinits}\downarrow & & \downarrow{\scriptstyle id+id\times myinits} \\ A^* & \xleftarrow[[nil,cons\cdot(init\times id)]]{} & 1 + A^* \times A^* \end{array} \qquad myinits = (\!|\, [nil, cons \cdot (init \times id)] \,|\!)$$

Por fim é preciso algo que nos trate do ponto 3. do **padrão**. Um simples *reverse* não chega pois é preciso reverter as sub-listas também. De certa forma é preciso um "deep" reverse, que trate das duas coisas.

O diagrama do catamorfismo seguinte é apenas referente à parte de reverter as sublistas:

$$\begin{array}{ccc} (A^*)^* & \underset{\text{in}}{\overset{\text{out}}{\rightleftarrows}} & 1 + A^* \times (A^*)^* \\ {\scriptstyle aux}\downarrow & & \downarrow{\scriptstyle id+id\times aux} \\ A^* & \xleftarrow[[nil,cons\cdot(reverse\times id)]]{} & 1 + A^* \times A^* \end{array} \qquad deepreverse = reverse \cdot (\!|\, [nil, cons \cdot (reverse \times id)] \,|\!)$$

Com estas 3 funções definidas, podemos tratar de resolver por fim o problema:

$$
\begin{aligned}
&matrot :: Eq\ a \Rightarrow [[a]] \to [a]\\
&matrot\ [\,] = [\,]\\
&matrot\ (h : t) = h +\!\!+ lasts\ t +\!\!+ matrot\ (deepreverse\ (myinits\ t))\\
&\quad \textbf{where}\\
&\qquad lasts = (\!|\, [nil, cons \cdot (last \times id)]\,|\!)\\
&\qquad myinits = (\!|\, [nil, cons \cdot (init \times id)]\,|\!)\\
&\qquad deepreverse = reverse \cdot (\!|\, [nil, cons \cdot (reverse \times id)]\,|\!)
\end{aligned}
$$

NB: O grupo gostaria de ter utilizado mais conhecimento desta cadeira para tentar tornar tudo *pointfree* no entanto esta foi a solução que conseguimos obter.

Problema 2

Análise do problema

Neste problema, a informação de que se trata de uma questão popular em entrevistas de emprego ajudou para pesquisar mais sobre este desafio. Rapidamente se chegou a soluções implementadas noutro tipo de linguagens como **C**, **C++**, **Java** e **Python**.

A que era mais aceite e apreciada, de uma forma sucinta, era uma solução que envolve dois *pointers*, um no início da String e outro no fim, em que se iam aproximando do "centro" da String e trocavam de referências quando o predicado era verificado. Tentamos ver se conseguiamos tirar proveito dessa solução mas não foi o caso.

Pensamos então melhor e chegamos à conclusão que podíamos utilizar uma lista auxiliar que (no contexto do *reverseVowels*) teria todas as vogais presentes na String, já pela ordem inversa. Com esta lista auxiliar seria percorrer ambas e quando o predicado se verificar na principal, construa-se a lista final com a cabeça da lista auxiliar. Caso contrário constrói-se com a cabeça da principal

Resolução

Começa-se por definir o predicado usado para o caso das vogais:

$$
\begin{aligned}
&isVowel :: Char \to Bool\\
&isVowel = (\in \texttt{"aáàãâeéèêiíìîoóòõôuúùûAÁÀÃÂEÉÈÊIÍÌÎOÓÒÕÔUÚÙÛ"})
\end{aligned}
$$

A *reverseVowels* será apenas a mais genérica, *reverseByPredicate*, com o predicado definido anteriormente:

$$
\begin{aligned}
&reverseVowels :: String \to String\\
&reverseVowels = reverseByPredicate\ isVowel
\end{aligned}
$$

A parte principal e mais desafiante começa agora, que seria definir a função genérica. Implementar a solução discutida na secção de análise em Haskell não seria algo complicado, no entanto queriamos tentar envolver mais conceitos da cadeira.

Tentamos então fazer diagramas para tentar perceber o que se poderia utilizar aqui. Olhando para a definição de catamorfismo:

$$(\!|\, g \,|\!) = g \cdot F\ (\!|\, g \,|\!) \cdot \text{out}$$

Achamos que para o contexto do que queriamos obter, era o que mais se adequava. Não seria no entanto um catamorfismo já definido, teríamos que definir um "novo" pois estamos a trabalhar com as duas listas, principal e auxiliar.

Este foi o diagrama que, juntamente com a definição de catamorfismo, nos levou a essa conclusão:

$$\begin{array}{ccc} A^* \times A^* & \xrightarrow[out]{} & A^* + A \times (A^* \times A^*) \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle id+id\times f} \\ A^* & \xleftarrow[[id,cons]]{} & A^* + A \times A^* \end{array}$$

Este catamorfismo tem uma peculiaridade que é a necessidade de ser acompanhado pelo predicado requirido pelo problema. Isto levou-nos a considerar se seria válido chamar a esta solução um catamorfismo, no entanto, visto que segue a definição referida em cima, mantemos a nomenclatura.

Conseguimos então definir o BiFunctor:

$$B\ (f,g) = id + f \times g$$

Temos então a definição do *out* para esta estrutura de dados:

$$\begin{array}{l} outPredicateList :: (a \to Bool) \to ([a],[a]) \to [a] + (a,([a],[a])) \\ outPredicateList\ p\ ([\,],l) = i_1\ l \\ outPredicateList\ p\ (y:ys,x:xs) = \\ \quad \textbf{if}\ p\ x\ \textbf{then}\ i_2\ (y,(ys,xs))\ \textbf{else}\ i_2\ (x,(y:ys,xs)) \end{array}$$

Deixamos na primeira parte do *Either* a lista principal quando a auxiliar tiver sido toda percorrida. Na segunda parte, temos o conjunto com o elemento a "entrar" na lista resultado, e com as duas listas, principal e auxiliar.

Fica assim definido o catamorfismo:

$$cataPredicateList\ p\ g = g \cdot recList\ (cataPredicateList\ p\ g) \cdot outPredicateList\ p$$

Acabando então com a função genérica:

$$\begin{array}{l} reverseByPredicate :: (a \to Bool) \to [a] \to [a] \\ reverseByPredicate\ p = f\ p\ gene \cdot \langle reverse \cdot filter\ p, id \rangle \\ \quad \textbf{where} \\ \quad\quad f\ p\ gene = cataPredicateList\ p\ gene \\ \quad\quad gene = [id, cons] \end{array}$$

Antes do catamorfismo, criamos o par (lista auxiliar, lista principal). A lista auxiliar seria apenas filtrar a principal segundo o predicado e depois revertê-la.

Problema 3

Análise do Problema

Este é um desafio bastante interessante, porque possibilita um aumento em eficiência significativo quando comparado com a solução de forma recursiva. Ao observar a expressão matemática no corpo do somatório reparamos em algo interessante. Notamos que existe uma enorme semelhança com a definição da expressão do e^x, número de Euler:

$$e^x = \sum_{k=0}^{\infty} \frac{x^k}{k!} \tag{5}$$

Na verdade, se fixarmos $k = 2k + 1$ temos exatamente a expressão para $sinh\ x$ (2). Lembramo-nos então de um exercício que ao se estudar para os testes escritos se resolveu que tratava de um problema semelhante mas para o número de Euler[1].

NB: Não mostramos a resolução do exercício mencionado pois é equiparado ao pedido para resolver no enunciado, como referido.

Resolução

Facilmente conseguimos perceber o valor de snh para $k = 0$:

$$snh\ x\ 0 = \frac{x^{2\cdot 0+1}}{(2\cdot 0+1)!} = \frac{x^1}{1!} = x$$

Agora tem de se ver o caso para $k = k + 1$

$$\text{snh}\ \ x\ (k+1) = \sum_{i=0}^{k+1} \frac{x^{2(k+1)+1}}{(2(k+1)+1)!}$$

$$\equiv$$

$$\text{snh}\ \ x\ (k+1) = \sum_{i=0}^{k+1} \frac{x^{2k+3}}{(2k+3)!}$$

$$\equiv$$

$$\text{snh}\ \ x\ (k+1) = snh\ x\ k + \frac{x^{2k+3}}{(2k+3)!}$$

Como temos $snh\ x\ (k+1)$ a depender do termo $\frac{x^{2k+3}}{(2k+3)!}$ definimos $h\ x\ k = \frac{x^{2k+3}}{(2k+3)!}$

Repetimos agora o processo para $h\ x\ 0$:

$$h\ x\ 0 = \frac{x^{2\cdot 0+3}}{(2\cdot 0+3)!} = \frac{x^3}{6}$$

E para $h\ x\ (k+1)$:

$$\text{h}\ \ x\ (k+1) = \frac{x^{2(k+1)+3}}{(2(k+1)+3)!}$$

$$\equiv$$

$$\text{h}\ \ x\ (k+1) = \frac{x^{2k+3+2}}{(2k+3+2)!}$$

$$\equiv$$

$$\text{h}\ \ x\ (k+1) = \frac{x^{2k+3}\cdot x^2}{(2k+3)!\cdot(2k+4)\cdot(2k+5)}$$

$$\equiv$$

$$\text{h}\ \ x\ (k+1) = \frac{x^{2k+3}}{(2k+3)!}\cdot\frac{x^2}{(2k+4)\cdot(2k+5)}$$

$$\equiv$$

$$\text{h}\ \ x\ (k+1) = h\ x\ k\cdot\frac{x^2}{4k^2+18k+20}$$

[1] Exercício (3.33) em [3], página 118.

Acontece mais uma vez a mesma coisa que em $snh\ x\ (k+1)$, o denominador da segunda parcela da soma em $h\ x\ (k+1)$ depende de k. Criamos entao $f\ k = 4k^2 + 18k + 20$.

Para esta parte não é preciso muitos mais cálculos, pois no anexo E é nos dado um exemplo envolvendo polinómios do segundo grau, que é exatamente o que $f\ k$ trata. Seguindo a resolução que lá mostra temos:

$$\begin{cases} \begin{cases} f\ 0 & = 20 \\ f\ (k+1) & = f\ k + g\ k \end{cases} \\ \begin{cases} g\ 0 & = 4 + 18 = 22 \\ g\ (k+1) & = g\ k + 2 \cdot 4 = g\ k + 8 \end{cases} \end{cases}$$

Em suma, temos então:

$$\begin{cases} \begin{cases} snh\ x\ 0 & = x \\ snh\ x\ (k+1) & = snh\ x\ k + h\ x\ k \end{cases} \\ \begin{cases} h\ x\ 0 & = \frac{x^3}{6} \\ \mathrm{h}\ x\ (k+1) & = h\ x\ k \cdot \frac{x^2}{f\ k} \end{cases} \\ \begin{cases} f\ 0 & = 20 \\ f\ (k+1) & = f\ k + g\ k \end{cases} \\ \begin{cases} g\ 0 & = 22 \\ g\ (k+1) & = g\ k + 8 \end{cases} \end{cases}$$

Com isto podemos então aplicar a *regra de algibeira* do anexo E:

$$\begin{aligned} &snh\ x = wrapper \cdot worker\ \textbf{where} \\ &\quad worker = \text{for}\ (loop\ x)\ (start\ x) \\ &\quad wrapper\ (a, \_, \_, \_) = a \\ &loop\ x\ (snh', h, f, g) = (snh' + h, h * (x\ **\ 2\ /\ f), f + g, g + 8) \\ &start\ x = (x, x\ **\ 3\ /\ 6, 20, 22) \end{aligned}$$

Problema 4

Análise do Problema

Talvez o problema mais interessante, pelos seus efeitos práticos, mas também o que requeriu mais discussão para perceber por onde começar. Percebeu-se no entanto que o ínicio estaria pela implementação da função *mkdist*. O desafio aqui seria perceber como se iria calcular as probabilidades dos elementos. Basicamente será a probabilidade de um elemento occorrer na lista. Acabamos por perceber que se agruparmos os elementos iguais em sub-listas, podemos depois calcular a probabilidade desse elemento dividindo o tamanho da sub-lista desse elemento pelo tamanho da lista (inicial) total.

De seguida, com *mkdist* implementada, o próximo passo seria "preencher" a base de dados. Agora a tarefa é mais facilitada, teríamos que aplicar a função anterior aos conjuntos de segmentos obtidos em *dados*.

A parte mais simples seria a próxima, que era implementar *delay*. Com as funções *mkf* e *instantaneous* fornecidas no anexo G o nosso trabalho ficou facilitado se interpretarmos a *db* como uma espécie de **map** e *mkf* uma espécie de **get** dado uma key, para se obter o valor associado.

Por fim, teríamos que implementar a função "principal", *pdelay*. Aqui rapidamente percebemos que se, a partir das duas paragens dadas em argumento, obtivéssemos o caminho total (por exemplo, caminho de S0 a S3 seria [(S0, S1), (S1, S2), (S2, S3)]) bastaria fazer a probabilidade de *delay* acumulada.

Resolução

Vejamos agora como está implementado.

mkdist como referido usaria um função que calcula a probabilidade de cada elemento estar presente na lista:

```
calcularDistribuicaoProbabilidades :: Eq a ⇒ [a] → [(a, ProbRep)]
calcularDistribuicaoProbabilidades lista =
  let totalElementos = fromIntegral $ length lista
      contagem = map (λx → (head x, fromIntegral (length x) / totalElementos)) $ group lista
  in contagem
```

Restava então utilizar uma função definida na biblioteca de probabilidades fornecida, *mkD*, que cria uma distribuição dado uma lista de $(a, ProbRep)$:

```
mkdist = mkD · calcularDistribuicaoProbabilidades
```

Para construir a base de dados, utiliza-se a função *agrupar* na lista de dados, que agrupa os segmentos iguais em sub-listas. Depois cria-se então um par de segmento e a distribuição do seu atraso, com a *mkdist*.

NB: Usamos um sort em *agrupar* para ser possível os dados serem passados em qualquer ordem, embora os do enunciado estejam já ordenados por segmento:

```
db = f dados where
  f = map constroiDb · agrupar
  agrupar = groupBy (λx y → π1 x ≡ π1 y) · sort
  constroiDb = ⟨π1 · head, mkdist · map π2⟩
```

Para a função *delay* seria apenas verificar o caso do **Maybe** que o *mkf* retorna. É **importante** referir que o grupo assumiu que caso o resultado fosse **Nothing** o suposto seria usar a função *instantaneous*:

```
delay seg =
  case mkf db seg of
    Nothing → instantaneous
    Just distDelay → distDelay
```

Finalmente, para implementar *pdelay*, como foi referido na parte de análise seria preciso uma função que dadas duas paragens retorna o caminho de uma a outra:

```
caminho :: Stop → Stop → [Segment]
caminho start stop = zip stops (tail stops)
  where
    stops = enumFromTo start stop
```

Para a função de produzir a distribuição das probabilidades acumuladas, tiramos proveito do facto de **Dist** ser um Monad, facilitando e tornando elegante a implementação desta função:

```
distAccum :: Dist Delay → Dist Delay → Dist Delay
distAccum d1 d2 = do
  x ← d1
  y ← d2
  return (x + y)
```

Tendo agora a função *pdelay* concluída, que aplica *delay* nos vários segmentos do caminho de s1 a s2. Logo após, utilizamos *foldr1*, uma variante da muito famosa operação *foldr* que é uma função de ordem superior que é usada para reduzir uma lista a um único valor. Esta variante assume que a lista nunca é vazia não necessitando de inicialização.

$$pdelay\ s1\ s2 = foldr1\ distAccum\ \$\ \text{map}\ \ delay\ \$\ caminho\ s1\ s2$$

Index

LaTeX, 4, 5
    bibtex, 5
    lhs2TeX, 4–6
    makeindex, 5
    pdflatex, 4
    xymatrix, 6

Combinador “pointfree”
    cata
        Naturais, 6
    either, 1
    split, 1, 6
Cálculo de Programas, 1, 2, 4, 6
    Material Pedagógico, 4

Docker, 4
    container, 4, 5

Functor, 3, 6–9
Função
    π_1, 6, 7
    π_2, 6
    for, 6, 7, 10

Haskell, 1, 4, 5
    Biblioteca
        PFP, 8
        Probability, 7, 8
    interpretador
        GHCi, 4, 5, 7
    Literate Haskell, 4

Números naturais ($\mathbb{N}$), 6, 7

Programação
    dinâmica, 2, 6
    literária, 4, 6

Taylor series
    Maclaurin series, 2

References

[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*, 2018. Draft of textbook in preparation. viii+297 pages. Informatics Department, University of Minho.