



Universidade do Minho
Escola de Engenharia

Cálculo de Programas

Trabalho Prático (2023/24)

Lic. em Engenharia Informática

Grupo G06

a93323	Benjamim Meleiro Rodrigues
a95454	Lara Beatriz Pinto Ferreira
a95319	Matilde Maria Ferreira de Sousa Fernandes

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

1	→	2	→	3
				↓
4	→	5		6
↑				↓
7	←	8	←	9

1	→	2	→	3	→	4
						↓
5	→	6	→	7		8
↑						↓
9	←	10	←	11	←	12

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](#),

$$\tanh x = \frac{\sinh x}{\cosh x} \quad (1)$$

podendo estes ser definidos pelas seguintes [séries de Taylor](#):

$$\sum_{k=0}^{\infty} \frac{x^{2k+1}}{(2k+1)!} = \sinh x \quad (2)$$
$$\sum_{k=0}^{\infty} \frac{x^{2k}}{(2k)!} = \cosh x$$

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

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

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 passageiros. 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

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

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

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

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

$db :: [(Segment, Dist Delay)]$

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

$mkdist :: Eq a \Rightarrow [a] \rightarrow Dist a$

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

$delay :: Segment \rightarrow Dist Delay$

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

Finalmente, o objectivo principal é definir a função probabilística:

$pdelay :: Stop \rightarrow Stop \rightarrow Dist Delay$

$pdelay\ a\ b$ 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 $[a \dots b]$.

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.

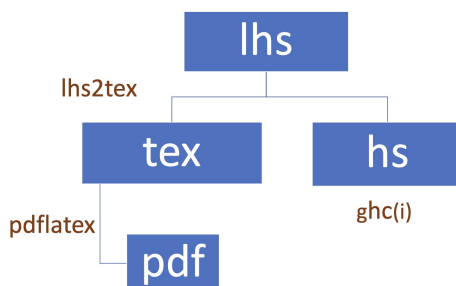
Para cumprir de forma integrada os objectivos do trabalho vamos recorrer a uma técnica de programação dita “[literária](#)” [?], cujo princípio base é o seguinte:

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 [L^AT_EX](#). (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 [L^AT_EX](#). 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-processor que faz “pretty printing” de código Haskell em [L^AT_EX](#) 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 [Bib_TE_X](#)) 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 ler¹ onde se obtém o efeito seguinte:²

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

Os diagramas podem ser produzidos recorrendo à *package* [xymatrix](#), por exemplo:

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

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:

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

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:

¹ Procure e.g. por "sec:diagramas".

² Exemplos tirados de [?].

³ Lei (3.95) em [?], 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* 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 aulas², 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:

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

F 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 $\text{Dist } a = D \{ \text{unD} :: [(a, \text{ProbRep})] \}$ (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

```
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:

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

² Secção 3.17 de [?] e tópico [Recursividade mútua](#) nos vídeos de apoio às aulas teóricas.


```
'D' 35.0%
'C' 29.0%
'E' 22.0%
'B' 12.0%
'A' 2.0%
```

É possível definir geradores de distribuições, por exemplo distribuições *uniformes*,

```
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.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 $g : A \rightarrow \text{Dist } B$ 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.

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 "Cristina Monteiro Carvalho Sequeira"
test1 = matrot m1 ≡ [1, 2, 3, 6, 9, 8, 7, 4, 5]
test2 = matrot m2 ≡ [1, 2, 3, 4, 8, 12, 11, 10, 9, 5, 6, 7]
test3 = matrot m3 ≡ "CristinaooarieuqeSCMonteirhlavra"
```

Problema 2

```
test4 = reverseVowels "" ≡ ""
test5 = reverseVowels "ácidos" ≡ "ocidás"
test6 = reverseByPredicate even [1..20] ≡ [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 [?].

Problema 3

Nenhum código é fornecido neste problema.

Problema 4

Os atrasos, medidos em minutos, são inteiros:

type Delay = \mathbb{Z}

Amostra de dados apurados por passageiros:

```
dados = [((S0, S1), 0), ((S0, S1), 2), ((S0, S1), 0), ((S0, S1), 3), ((S0, S1), 3),  
         ((S1, S2), 0), ((S1, S2), 2), ((S1, S2), 1), ((S1, S2), 1), ((S1, S2), 4),  
         ((S2, S3), 2), ((S2, S3), 2), ((S2, S3), 4), ((S2, S3), 0), ((S2, S3), 5),  
         ((S3, S4), 2), ((S3, S4), 3), ((S3, S4), 5), ((S3, S4), 2), ((S3, S4), 0),  
         ((S4, S5), 0), ((S4, S5), 5), ((S4, S5), 0), ((S4, S5), 7), ((S4, S5), -1)]
```

“Funcionalização” de listas:

```
mkf :: Eq a => [(a, b)] -> a -> Maybe b  
mkf = flip Prelude.lookup
```

Ausência de qualquer atraso:

```
instantaneous :: Dist Delay  
instantaneous = D [(0, 1)]
```

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

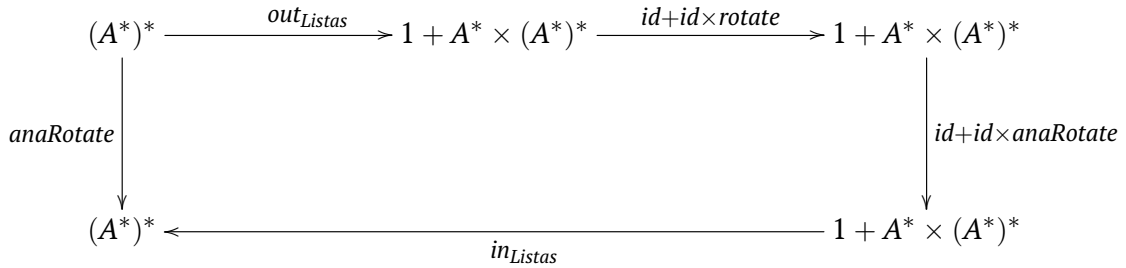
Para resolver este problema, chegamos à conclusão que tínhamos que guardar a primeira linha da matrix e depois rodá-la 90 graus no sentido anti-horário e repetir o processo até que a mesma ficasse vazia.

Primeiramente, definimos a função rotate, que é a base para a construção da espiral. Esta função realiza duas operações principais: a transposição da matriz (transpose), que troca as suas linhas por colunas, seguida pela inversão de cada nova linha (reverse). O resultado é uma rotação de noventa graus no sentido anti-horário.

```
rotate :: Eq a => [[a]] -> [[a]]  
rotate = reverse · transpose
```

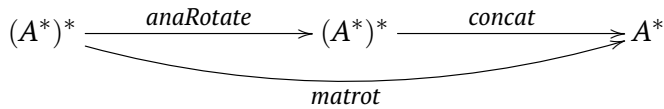
O núcleo da nossa estratégia é a função anaRotate, um anamorfismo que emprega a rotate em cada passo recursivo. Utilizando o combinador (id + id x rotate), que aplica a função rotate ao resto da

matriz após separar a primeira linha. A cada iteração, `anaRotate` acumula a primeira linha da matriz transformada, construindo assim uma lista de listas, onde cada sublista representa uma camada da espiral. O diagrama de anamorfismo abaixo visualiza este processo.



$\text{anaRotate} :: \text{Eq } a \Rightarrow [[a]] \rightarrow [[a]]$
 $\text{anaRotate} = \llbracket (\text{id} + \text{id} \times \text{Main.rotate}) \cdot \text{outList} \rrbracket$

Para alcançar a representação final da matriz em espiral, utilizámos a função `concat`, que concatena todas as sublistas numa única lista. O diagrama a seguir ilustra a aplicação de `concat` à estrutura produzida por `anaRotate`, resultando na lista final que pretendemos obter com a função `matrot`.



A função `matrot` é definida como a composição de `concat` e `anaRotate`. Esta única linha encapsula todo o processo de transformação da matriz original numa lista com os elementos rodados em espiral, como demonstrado pela equivalência a seguir.

$\text{matrot} :: \text{Eq } a \Rightarrow [[a]] \rightarrow [a]$
 $\text{matrot} = \text{concat} \cdot \text{anaRotate}$

Problema 2

Inicialmente o Problema 2 pede para inverter as vogais de uma string. Como tal, decidimos implementar a função `pre` que consiste em colocar todas as vogais no fim da string, mantendo a ordem original dos outros caracteres.

Para tal, utilizamos a função `conc` que concatena duas strings e a função `split` que separa uma string em duas, conforme o predicado que `lhe` é passado. Neste caso, o predicado é a função `isVowel` que verifica se um caracter é uma vogal.

A função `pre` é definida como a composicao de `conc` e `split`, juntamente com `filter isVowel`.

$\text{isVowel} = \text{oneOf} \text{ "AEIOUaeiou\AA\AA\AE\EE\I\I\O\O\U\U\aa\aa\ae\ee\ii\oo\ou\uu\i\i\U\U"}$

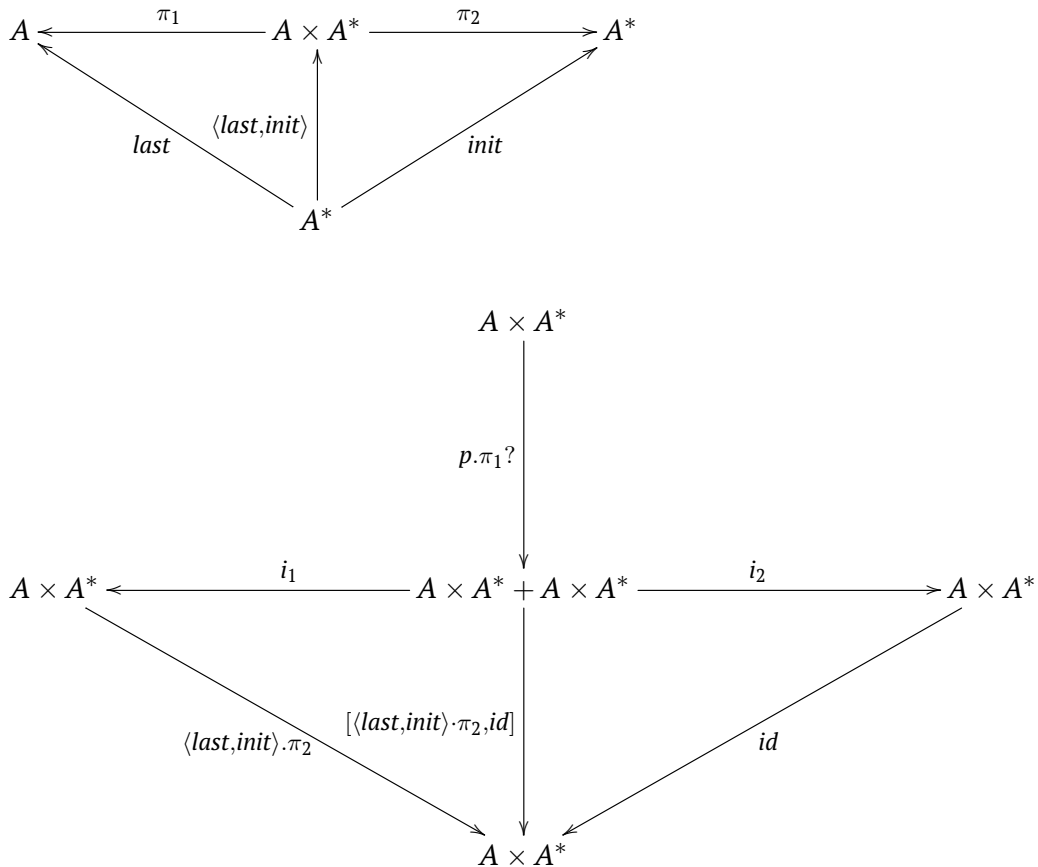
$\text{pre } p = \text{conc} \cdot \langle \text{id}, \text{filter } p \rangle$

Posteriormente, implementamos a função `anaReverse`, tendo em conta que a lista de entrada para esta função já é o resultado da função `pre`, ou seja, as vogais estão no fim da string.

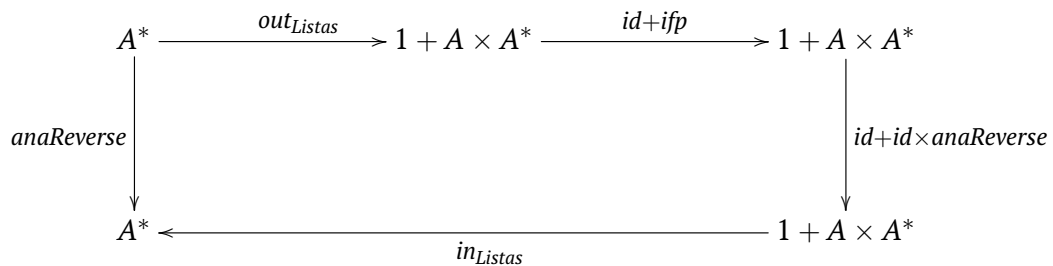
A *anaReverse* percorre essa lista e inverte elementos específicos, baseando-se num predicado *p*.

Dentro do anamorfismo, a função *ifp* é usada com mecanismo de decisão que aplica o predicado *p* a cada elemento da lista, quando o elemento é uma vogal a função *ifp* aplica um *split*, substituindo o elemento em questão pelo último elemento da lista e removendo o mesmo. Isso resulta na inversão da posição das vogais dentro da lista.

Se o elemento não satisfizer o predicado, é mantido na sua posição original e a função procede para o próximo elemento da lista.



$$\begin{aligned}
 ifp &= [\langle last, init \rangle \cdot \pi_2, id] \cdot (p \cdot \pi_1) ? \\
 \equiv & \quad \{ \text{Def condicional de McCarthy} \} \\
 ifp &= p \cdot \pi_1 \rightarrow \langle last, init \rangle \cdot \pi_2, id
 \end{aligned}$$



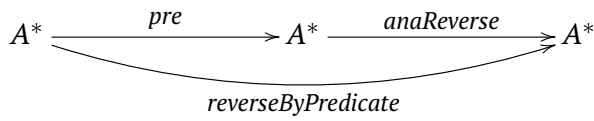
```

anaReverse :: (a2 → Bool) → [a2] → [a2]
anaReverse p = [(id + ifp) · outList]
  where
    ifp = p · π1 → ⟨last, init⟩ · π2, id

reverseVowels :: String → String
reverseVowels = reverseByPredicate isVowel

```

Na segunda parte do Problema 2 é pedido para generalizar a função inicial, de forma a inverter os elementos de uma dada lista que satisfazem um dado predicado.



```

reverseByPredicate :: (a → Bool) → [a] → [a]
reverseByPredicate p s = anaReverse p $ pre p s

```

Problema 3

Começamos por analisar a diferença entre duas iterações consecutivas do somatório:

$$\text{iteracao}k = \frac{x^{(2k+1)}}{(2k+1)!} \tag{5}$$

$$\text{iteracao}(k+1) = \frac{x^{(2(k+1)+1)}}{(2(k+1)+1)!} = \frac{x^{(2k+3)}}{(2k+3)!} = \frac{x^{(2k+1)} \times x^2}{(2k+3)(2k+2)(2k+1)!} \tag{6}$$

Como podemos ver, a diferença entre duas iterações consecutivas corresponde a multiplicação do elemento anterior com o valor $\frac{x^2}{(2k+3)(2k+2)}$, o que nos permite calcular o próximo elemento a partir do anterior de maneira eficiente.

Nesta função guardamos num acumulador o somatório até à iteração atual, o valor da iteração anterior e o número da iteração.

No caso de paragem temos a função start que retorna um acumulador com os valores iniciais para a iteração 0, ou seja, ((x,x),0).

```

start :: Num b1 ⇒ b2 → ((b2, b2), b1)
start x = ((x, x), 0)

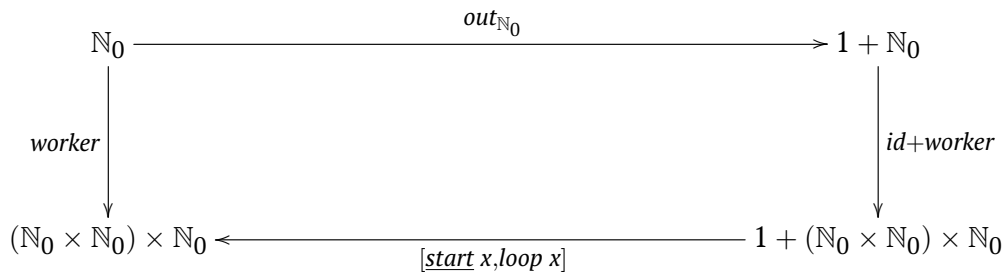
```

Nos restantes casos, a função `loop` recebe o acumulador e retorna um novo acumulador com os valores atualizados.

$den :: Num\ a \Rightarrow a \rightarrow a$
 $den\ i = (2 * i + 2) * (2 * i + 3)$

$num :: Num\ a \Rightarrow a \rightarrow a$
 $num\ x = x \uparrow 2$

$loop :: Fractional\ b \Rightarrow b \rightarrow ((b, b), b) \rightarrow ((b, b), b)$
 $loop\ x\ ((s, p), i) = ((s + conta, conta), i + 1)$
where $conta = p * (num\ x / den\ i)$



$worker = \text{for loop } x \text{ start } x$
 $\equiv \{ \text{Def ciclo-for} \}$
 $worker = \llbracket [start\ x, loop\ x] \rrbracket$

Para devolver o valor do somatório, a função `wrapper` recebe o acumulador e retorna o valor do somatório, ou seja, o primeiro elemento do acumulador.

$snh :: (Integral\ a, Fractional\ b) \Rightarrow b \rightarrow a \rightarrow b$
 $snh\ x = wrapper \cdot worker$
where
 $worker = \text{for loop } x \text{ start } x$
 $wrapper = \pi_1 \cdot \pi_1$

Problema 4

Para auxiliar a geração da base de dados probabilística, criamos 3 funções auxiliares.

- A função `lka` recebe um segmento e uma lista de dados e devolve uma lista com os atrasos associados a esse segmento

$lka :: Eq\ a \Rightarrow a \rightarrow [(a, b)] \rightarrow [b]$
 $lka\ k = \text{map } \pi_2 \cdot \text{filter } ((\equiv k) \cdot \pi_1)$

- A função `mkdist` faz o sumário estatístico de uma qualquer lista finita, gerando a distribuição de ocorrência dos seus elementos. Começamos por aplicar a função `map`, de forma a obter uma lista de tuplos com o elemento e a sua percentagem de ocorrências. De seguida aplicamos a função `nub`, que remove os elementos repetidos da lista, e por fim aplicamos a `D` que transforma a lista de tuplos numa distribuição.

```
-- mkdist :: Eq a => [a] -> Dist a
mkdist l = D $ nub $ map ((id × divide) · Cp.dup) l
  where
    divide x = fromIntegral (n x l) / fromIntegral t
    t = length l
    n x = length · filter (≡ x)
```

- A função `mksd` começa por aplicar a função `lka` aos dados, de forma a obter uma lista com os atrasos de um determinado segmento, de seguida aplica a `mkdist` a essa lista, obtendo assim a distribuição de ocorrência dos atrasos desse segmento.

```
-- mksd :: Segment -> (Segment, Dist Delay)
mksd = ⟨id, mkdist · flip lka dados⟩
```

Por fim para gerar a base de dados probabilística, começamos por extrair os segmentos dos dados e eliminar os repetidos, de forma a obter uma lista de segmentos.

De seguida aplicamos a função `mksd` a todos os segmentos, devolvendo assim uma lista de tuplos com o segmento e a sua respetiva distribuição.

```
-- db :: [(Segment, Dist Delay)]
db = map mksd $ nub $ map π1 dados
```

Esta função começa por procurar o segmento na base de dados probabilística, e caso o encontre, devolve a sua distribuição de atrasos.

```
-- delay :: Segment -> Dist Delay
delay = fj · mkf db
```

```
fj :: Maybe a → a
fj (Just a) = a
```

A `pdelay` é uma função probabilística que 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]`.

Esta função recebe duas paragens e devolve a distribuição de atrasos acumulados entre as mesmas. De modo a realizarmos o pretendido, começamos por calcular a lista de segmentos de todas as paragens entre as duas paragens dadas.

Para tal, utilizámos a função `enumFromTo` que devolve uma lista com todas as paragens entre `a` e `b`, de seguida aplicamos um `split id tail` e juntamos para obter a lista de segmentos entre essas duas paragens.

Posteriormente, aplicamos a função `somatorio` que recebe a lista de segmentos e devolve a distribuição de atrasos acumulados entre os mesmos.

Esta função aplica a cada segmento a função delay e soma essa distribuição com a distribuição de atrasos do resto do percurso que foi calculada recursivamente pelo catamorfismo.

$$\begin{array}{ccc}
 \text{Segment}^* & \xrightarrow{\text{out}_{\text{Listas}}} & 1 + \text{Segment} \times \text{Segment}^* \\
 \downarrow \text{somatorio} & & \downarrow \text{id} + \text{id} \times \text{somatorio} \\
 \text{DistDelay} & \xleftarrow{[\text{instantaneous}, (\widehat{\text{joinWith } (+)} \cdot \text{delay})]} & 1 + \text{Segment} \times \text{DistDelay}
 \end{array}$$

$\text{somatorio} :: [\text{Segment}] \rightarrow \text{Dist Delay}$

$\text{somatorio} = \llbracket [\text{instantaneous}, (\widehat{\text{joinWith } (+)} \cdot \text{delay})] \rrbracket$

-- pdelay :: Stop -> Stop -> Dist Delay

$\text{pdelay } a \ b = \text{somatorio } \$ \widehat{\text{zip}} \$ \langle \text{id}, \text{tail} \rangle \$ \text{enumFromTo } a \ b$