



Universidade do Minho
Escola de Engenharia

Cálculo de Programas

Trabalho Prático (2023/24)

Lic. em Engenharia Informática

Grupo G14

a100594 João Manuel Machado Lopes
a100665 Tiago Nuno Magalhães Teixeira
a100693 Luís Vítor Lima Barros

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 ?? onde encontrarão as instruções relativas ao software a instalar, etc.

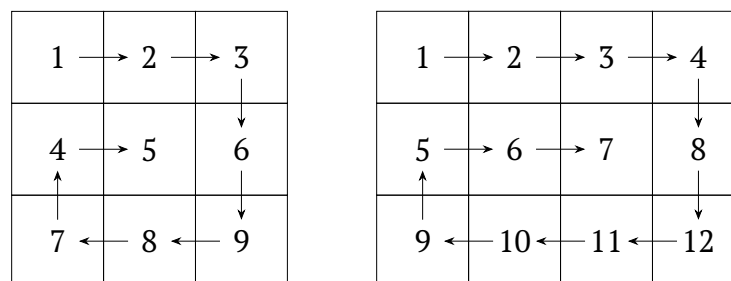
Valoriza-se a escrita de *pouco* código que corresponda a soluções simples e elegantes que utilizem os combinadores de ordem superior estudados na disciplina.

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:



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 ??.

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 ??.

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 $\tanh x$ (??), através da soma das i primeiras parcelas da sua série (??).

Deverá ser seguida a regra prática do anexo ?? 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 ??). 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 ??, página ??, 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.

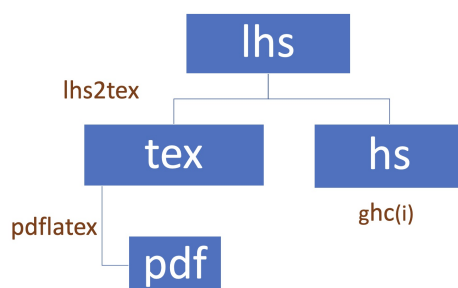
Para cumprir de forma integrada os objectivos do trabalho vamos recorrer a uma técnica de programação dita “[literária](#)” [?], cujo princípio base é o seguinte:

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 ?? 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 ?? 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 ?? 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* \Rightarrow [(*a*, *b*)] \rightarrow *a* \rightarrow *Maybe b*
mkf = *flip Prelude.lookup*

Ausência de qualquer atraso:

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

H Soluções dos alunos

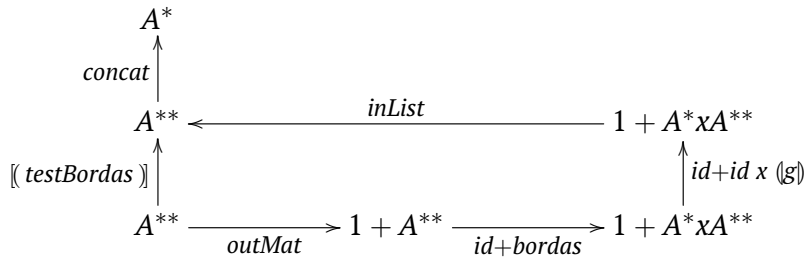
Os alunos devem colocar neste anexo as suas soluções para os exercícios propostos, de acordo com o “layout” que se fornece. Não podem ser alterados os nomes ou tipos das funções dadas, mas pode ser adicionado texto ao anexo, bem como diagramas e/ou outras funções auxiliares que sejam necessárias.

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

Problema 1

Para este problema partimos o problema em duas partes. Sabemos que a travessia em espiral é resultante da consecutiva travessia e remoção das bordas da matriz (primeira linhas -> ultima coluna -> última linha ao contrario -> primeira coluna de baixo para cima). Sendo de destacar que este processo termina quando a matriz estiver vazia, comportamento o qual que vai ser detetado pela função *outMat*. Dito isto, para a resolução deste problema, criamos uma função auxiliar que recebe uma matriz e devolve o par da lista correspondente à sua borda e a sua matriz interior. Para esta função funcionar, recorreremos a uma função que determina quando uma matriz é vazia *isEmpty*, de forma a determinar o ponto de paragem. Tendo as funções auxiliares todas devidamente definidas, a função principal *matrot* vai recorrer ao anamorfismo de listas, construindo assim, a partir da matriz inicial, uma lista com todas as suas bordas, por fim junta todos estes valores com *concat* de modo ao resultado corresponder a uma lista única que representa a rotação em espiral.

Desenhando o diagrama da função principal, obtemos a seguinte figura:



matrot :: *Eq a* ⇒ *[[a]]* → *[a]*

matrot = *concat* · *[[testBordas]]*

testBordas = (*id* + *bordas*) · *outMat*

outMat all@((_: _): _) = i₂ all

outMat _ = *i₁* ()

isEmpty :: *[[a]]* → *Bool*

isEmpty = [*true*, *false*] · *outMat*

applyConcFirst = (*conc* × *id*) · *assocl*

insertPair = (*cons* × *cons*) · *assocr* · (*assocl* × *id*) · ((*id* × *swap*) × *id*) · (*assocr* × *id*) · *assocl*

applyNotEmpty = *isEmpty* → *⟨nil, nil⟩*, ·

getLastColumn :: *[[a]]* → (*[a]*, *[[a]]*)

getLastColumn = *applyNotEmpty* (*g*)

where *g* = [*⟨nil, nil⟩*, *insertPair* · (*⟨last, init⟩* × *id*)]

getFirstColumnReversed :: *[[a]]* → (*[a]*, *[[a]]*)

getFirstColumnReversed = (*reverse* × *id*) · (*applyNotEmpty* (*g*))

where *g* = [*⟨nil, nil⟩*, *insertPair* · (*⟨head, tail⟩* × *id*)]

getLastReversed :: *[[a]]* → (*[a]*, *[[a]]*)

getLastReversed = *applyNotEmpty* (*reverse* · *last*, *init*)

getFirstLine :: *[[a]]* → (*[a]*, *[[a]]*)

getFirstLine = *applyNotEmpty* (*head*, *tail*)

bordas :: *[[a]]* → (*[a]*, *[[a]]*)

bordas = *applyConcFirst* · (*id* × *getFirstColumnReversed*) · *applyConcFirst* · (*id* × *getLastReversed*) · *applyConcFirst*

Problema 2

Face ao problema dado, começamos por fazer uma análise do mesmo e chegamos à conclusão de que a função *reverseVowels* é um caso específico da função *reverseByPredicate* cujo predicado se trata de uma função que avalia se um caracter é uma vogal. Logo, criamos a função *isVowel* e defenimos a primeira função através das duas funções mencionadas anteriormente.

Partindo então para a função genérica, começamos por inserir um indice da sua posição em cada elemento da lista de forma a manter informação das suas posições iniciais, resulando assim numa lista de pares: (*indice*, *valor*). Posteriormente, separamos os elementos cujo valor respeita ou não o predicado, recorrendo assim à função *splitByPredicate*, esta recebe uma função que avalia um elemento e uma lista, transofrmando esta num par de listas em que a primeira corresponde aos elementos que respeitam o predicado e a segunda os restantes. De seguida, invertemos a ordem dos valores da primeira lista, através da função *reverseP2*, e mantemos a outra inalterada. Por fim, juntamos as duas listas e reordenamos de acordo com os índices e retiramos os mesmos, recorrendo assim à função *sortOnAndRemoveP1*.

Diagrama da função *reverseP2*:

$$(A \times B)^* \xrightarrow{\text{unzip}} A^* \times B^* \xrightarrow{\text{id} \times \text{reverse}} A^* \times B^* \xrightarrow{\widehat{\text{zip}}} (A \times B)^*$$

Diagrama da função *splitByPredicate*:

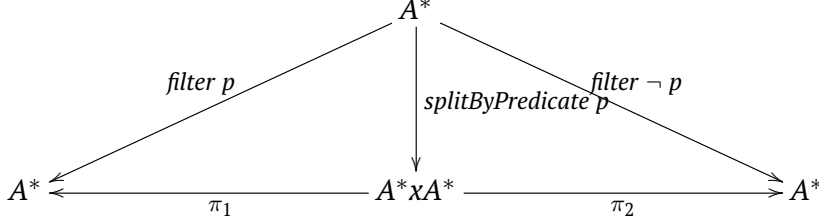
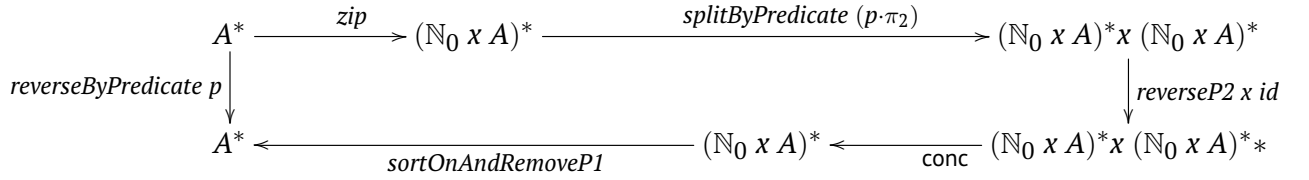


Diagrama da função principal:



reverseVowels :: *String* → *String*

reverseVowels = *reverseByPredicate isVowel*

isVowel = *flip elem* "áàãäéíouyÂÃÄÊËÏΟΥΥ"

reverseByPredicate :: (*a* → *Bool*) → [*a*] → [*a*]

reverseByPredicate *p* = *sortOnAndRemoveP1* · *conc* · (*reverseP2* × *id*) · *splitByPredicate* (*p* · *π2*) · *zip nat0*

reverseP2 = $\widehat{\text{zip}}$ · (*id* × *reverse*) · *unzip*

sortOnAndRemoveP1 = *map* *π2* · (*sortOn* *π1*)

splitByPredicate *p* = $\langle \text{filter } p, \text{filter } (\neg \cdot p) \rangle$

Problema 3

Para este problema, começamos por analisar as fórmulas matemáticas. Tanto no cosseno hiperbólico quanto no seno hiperbólico, é notável que a função objetivo apresenta uma forma semelhante.

Dito isto, se criarmos uma função auxiliar para ser aplicada a *k*, nomeadamente:

$$\text{senhk}(k) = 2k + 1 \quad (5)$$

$$\text{coshk}(k) = 2k \quad (6)$$

Dito isto, criamos uma função genérica para os dois casos:

$$f(x, j) = \frac{x^j}{j!} \quad (7)$$

Resultando assim nos seguintes somatórios:

$$\sum_{k=0}^{\infty} f(x, \sinh k(k)) = \sinh x \quad (8)$$

$$\sum_{k=0}^{\infty} f(x, \cosh k(k)) = \sinh x$$

De modo a simplificar os calculos é estabelecido um valor maximo que k pode atingir : i .

Dito isto, podemos partir qualquer um dos somatórios em duas casos diferentes, um caso de paragem quando temos $i = 0$, e um intermédio/inicial:

$$\sum_{k=0}^0 f(x, \sinh k(k)) = f(x, \sinh k(0)) = f(x, 2 * 0 + 1) = f(x, 1) = \frac{x^1}{1!} = x \quad (9)$$

$$\sum_{k=0}^i f(x, \cosh k(k)) = f(x, \sinh k(i)) + \sum_{k=0}^{i-1} f(x, \cosh k(k))$$

Partindo agora para a implementação em si, de modo a podermos utilizar o `for` predefinido, as funções de paragem e intermédias devolvem uma par que corresponde: valor de i e o valor do sumatório para k de 0 até i . Tendo isto em conta, estabelecemos o valor do caso de paragem na função `start` e o caso intermédio através da função `loop`, à qual acrescentamos como argumento a função `senhk`, de modo a este loop poder ser reutilizado para posteriormente construir a função do `cosh`.

Dito isto a função `loop`, além de receber o valor de x e a função a ser aplicada ao índice correspondente, vai receber também um par que corresponde ao valor do indice anterior e o sumatório até aquele instante. Com essas informações, incrementa o índice, utiliza a função f que representa a função objetivo para calcular a nova parcela e soma esta ao valor acumulado no sumatório até ao momento.

Terminado assim o ciclo, pretendemos apenas devolver ao utilizador o resultado do sumatório, ou seja o segundo elemento do par calculado, utilizando assim o `wrapper`.

```

snh x = wrapper · worker where
  worker = for (loop x senhk) (start x senhk)
  wrapper = π2
  senhk = succ · (*2)
  incrementaIndice = (succ × id)
  loop x func = (id × myadd) · assocr · (⟨id, (f x) · func⟩ × id) · incrementaIndice
  myadd = (⊕)
  f x = (⌈/⌋) · (id × fromInteger) · ⟨Nat.exp x, Nat.fac⟩
  start x func = (0, f x (func 0))

```

Problema 4

Para este problema, começamos por criar a função `mkdist` que para uma dada lista, cria a sua distribuição, sendo as probabilidades determinadas apartir do número de vezes que um dado elemento encontra-se presente na lista. Para este efeito, a função começa por determinar a probabilidade de

cada elemento da lista, sem ter em conta repetidos. De seguida, utiliza a função *insereProb*, passando como argumento o par da probabilidade de cada elemento e a lista. Esta função auxiliar recebe um par de um elemento qualquer e uma lista e devolve uma lista em que os seus elementos são os mesmos que a lista inicial, contudo dentro de um par em que o segundo elemento foi o elemento passado como argumento. Estando assim os pares de elementos e a sua probabilidade definidos, usamos a função *mkD*, responsável por criar a estrutura de dados desejada devidamente normalizada.

De modo a criar a base de dados *db*, passamos os dados como argumento à função *criarBase*, esta será responsável por agrupar os dados de acordo com os segmentos e criar as respetivas distribuições dos seus atrasos, devolvendo assim uma lista de pares de **Segment** e **Dist**.

A função *delay* procura na base de dados a distribuição correspondente ao segmento, caso não encontre nada sobre o mesmo, podemos assumir que não houve atrasos ou seja a probabilidade de o atraso ser 0 é igual a 1.

Analisando o problema, é visível que o atraso é acumulativo, ou seja, de modo a combinar dois segmentos o atraso no primeiro vai afectar o atraso no segundo. Dito isto esta combinação é realizada pela função *combinaDelays* responsável por combinar duas distribuições, sendo que os respetivos valores são somados, representando assim a acumulação de atrasos. Seguindo o mesmo raciocínio, de modo a combinar uma lista de distribuições numa única distribuição, utilizamos a função *combinaListDelays* que corresponde ao catamorfismo de listas que calcula a distribuição final.

Por fim criamos uma função auxiliar *parteSegments* que determina todas as sequências intermédias entre dois **Enum** do mesmo tipo, obtendo assim o seguinte efeito para o nosso caso específico, determinando todos os segmentos interiores entre duas paragens.

$$parteSegments(S1, S4) = [(S1, S2), (S2, S3), (S3, S4)]$$

Recorrendo assim às funções auxiliares mencionadas anteriormente, podemos finalmente construir a função *pdelay*, responsável por calcular a distribuição de atrasos entre duas paragens, começando por criar a lista de segmentos seguida da transformação dos segmentos nas suas distribuições de atrasos. E por fim, combinar as distribuições da lista, utilizando assim a função *combinaListDelays*, utilizando a função *curry* de modo aos dois argumentos recebidos poderem ser passados como um par para a função *parteSegments* no início da sua composição.

```
db = criarBase dados
criarBase = (map (id × mkdist)) · (map ⟨π1 · head, (map π2)⟩) · (groupBy equalFirst)
mkdist = mkD · insereProb · ⟨(1/) · fromIntegral · length, id⟩
insereProb :: (b, [a]) → [(a, b)]
insereProb =  $\widehat{\cdot}$  $ (map · flip (,))
equalFirst :: Eq a ⇒ ((a, b), (a, c)) → Bool
equalFirst =  $\widehat{(\equiv)}$  · (π1 × π1)
combinaListDelays =  $\llbracket$  instantaneous, combinaDelays  $\rrbracket$ 
combinaDelays = (joinWith  $\widehat{(+)}$ )
delay = [instantaneous, id] · outMaybe · (mkf db)
parteSegments :: Enum a ⇒ (a, a) → [(a, a)]
parteSegments =  $\widehat{zip}$  · ⟨listEnums · (id × pred), listEnums · (succ × id)⟩
  where listEnums = enumFromTo
pdelay = combinaListDelays · (map delay) · parteSegments
```