
**UNIVERSIDADE FEDERAL DE SERGIPE**

**Disciplina: Programação Funcional**

# Projeto: Mapa

Márcio Henrique Matos de Freitas

Amanda Silva Araújo

Manoel Lourenço De Almeida Neto

João Pedro Natividade Dos Santos

São Cristovão-Sergipe
2023


**UNIVERSIDADE FEDERAL DE SERGIPE**

**Disciplina Programação Funcional**

Márcio Henrique Matos de Freitas

Amanda Silva Araújo

Manoel Lourenço De Almeida Neto

João Pedro Natividade Dos Santos

Professor(a): BEATRIZ TRINCHÃO ANDRADE DE CARVALHO

São Cristovão-Sergipe
2022

# Sumário

**1 Introdução** **5**

**2 Tipos de dados** **5**

**3 Funções Usadas** **5**

3.1 Funções Mapa Inicial e addMapa . . . . . 5
3.2 Função Estrada Existe . . . . . 6
3.3 Funções AtualizarEstradasRemovidas e removeCidade . . . . . 6
3.4 Funções addEstrada e removeEstrada . . . . . 6
3.5 Funções cidade Existe, coordenadas Existem, checarCidade . . . . . 7
3.6 Funções checarCidadeRepetida e checarCoordenadasRepetidas . . . . . 7
3.7 Função Manipular Mapa . . . . . 8
3.8 Função Manipular Mapa Opção 1 . . . . . 9
3.9 Função Manipular Mapa Opção 2 . . . . . 9
3.10 Função Manipular Mapa Opção 3 e 4 . . . . . 10
3.11 Função Manipular Mapa Opção 5,6 e 7 . . . . . 10
3.12 Função Manipular Mapa 6 . . . . . 11

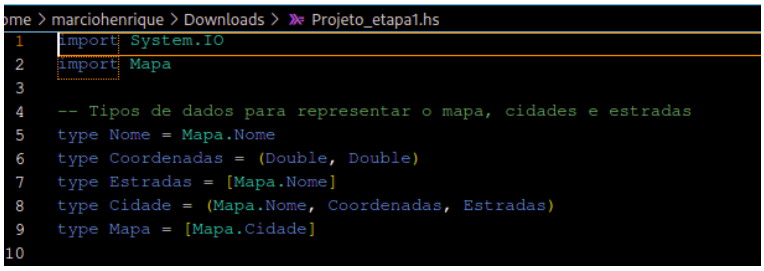
**4 Etapa 2 do Projeto** **11**

4.1 Função GetNome . . . . . 11
4.2 Função GetCoords . . . . . 11
4.3 Função Getcidade . . . . . 11
4.4 Função GetEstradas . . . . . 12
4.5 Função Addcidade *Correção da 1 etapa . . . . . 12
4.6 Função addEstrada e removeEstrada *Correção da 1 etapa . . . . . 13
4.7 Função calcularDistancia . . . . . 13
4.8 Função EstradaExiste . . . . . 14
4.9 Função RotaExiste . . . . . 14
4.10 Função EncontrarCaminho . . . . . 15
4.11 Função calcularSomaDistancias e calcularDistanciaRota . . . . . 16
4.12 Função checarEstradaRepetida . . . . . 17
4.13 Função ManipularMapa . . . . . 17

# 1 Introdução

No projeto Mapa da disciplina de Programação, no qual esse mapa é visto como um sistemas cartesiano x e y , onde podemos representar as cidades como ponto, assim cada cidade possui nome ,coordenada no mapa com conexões para outras cidades feitas por rota. O objetivo é entender como as cidades vão se relacionar e otimizar essas rotas. Um problema bem comum em planejamento urbanos.

# 2 Tipos de dados




Primeiramente, definimos os tipos que iremos usar em todo o projeto. O tipo "Nome" são as denominações que daremos as cidades. Ele está definido como "Mapa.Nome" para evitar conflitos entre os códigos, pois estava com as mesmas assinaturas que o módulo que serve para salvar arquivos. Utilizaremos essa técnica para os demais modelos.

Logo após, definimos o tipo "Coordenadas" como uma tupla de doubles. O tipo "Estrada" está projetado como uma lista de nomes que servem para verificar se as rotas entre as cidades estão conectadas. E com todas essas divisões, criamos o tipo "Cidade" que recebe cidade, coordenadas e estrada para assim categorizarmos "Mapa" como uma lista de cidades.

# 3 Funções Usadas

## 3.1 Funções Mapa Inicial e addMapa


A função addCidade destina-se a adicionar uma cidade a um mapa existente. Ela recebe dois parâmetros: o primeiro parâmetro é o mapa atual, mostrado como uma lista de cidades, e o segundo parâmetro é a cidade a ser adicionada, mostrada como uma tupla contendo o nome da cidade, suas coordenadas e uma lista de estradas vazias. A principal mudança introduzida por esta função é a adição da nova cidade ao mapa. Para fazer isso, o operador: (dois pontos) é usado para construir a nova lista. No código, a nova cidade é adicionada ao início do mapa existente e uma nova lista é gerada contendo a cidade recém-inserida, juntamente com todas as outras cidades já existentes no mapa. Quando essa função é executada, o mapa é atualizado com as cidades adicionadas e as informações sobre as cidades anteriores são mantidas.

## 3.2 Função Estrada Existe

```haskell
-- Verifica se duas cidades possuem conexão
estradaExiste :: Mapa.Mapa -> Mapa.Nome -> Mapa.Nome -> Bool
estradaExiste mapa cidadeOrigem cidadeDestino =
    cidadeDestino `elem` estradasOrigem && cidadeOrigem `elem` estradasDestino
    where
        estradasOrigem = getEstradas cidadeOrigem  mapa
        estradasDestino = getEstradas cidadeDestino  mapa
```

A função estradaExiste é de suma importância em sistemas que envolvem mapas e estradas, pois permite verificar se duas cidades estão conectadas por uma estrada direta. Isso é crucial para validar conexões, planejar rotas.

## 3.3 Funções AtualizarEstradasRemovidas e removeCidade

```haskell
atualizarEstradasRemovidas :: Mapa.Mapa -> Mapa.Nome -> Mapa.Mapa
atualizarEstradasRemovidas [] _ = []
atualizarEstradasRemovidas ((nome, coordenadas, estradas):outrasCidades) cidadeRemovida =
    (nome, coordenadas, filter (/= cidadeRemovida) estradas) : atualizarEstradasRemovidas outrasCidades cidadeRemovida


-- Remove uma cidade do mapa, pelo nome
removeCidade :: Mapa.Mapa -> Mapa.Nome -> Mapa.Mapa
removeCidade mapa nomeCidade =
    let mapaAtualizado = atualizarEstradasRemovidas mapa nomeCidade
    in [(nome, coordenadas, estradas) | (nome, coordenadas, estradas) <- mapaAtualizado, nome /= nomeCidade]
```

A função "atualizarEstradasRemovidas"desempenha um papel crucial ao atualizar as rotas entre cidades após a remoção de uma cidade específica do mapa. Ela utiliza a recursão para percorrer todas as cidades do mapa, removendo o nome da cidade excluída das estradas de outras cidades. Isso garante que as rotas sejam atualizadas de forma adequada, mantendo a consistência do mapa.

A função "removeCidade"faz uso da função anterior, "atualizarEstradasRemovidas", para efetivamente remover uma cidade do mapa. Ela cria um novo mapa onde a cidade removida não está mais presente e as estradas das outras cidades são atualizadas de acordo. Esse processo garante que a cidade seja removida de forma completa e que todas as conexões com outras localidades sejam devidamente ajustadas

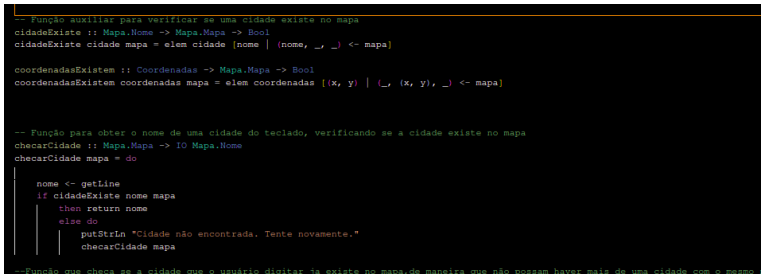
## 3.4 Funções addEstrada e removeEstrada

```haskell
addEstrada mapa cidadeOrigem cidadeDestino =
    if not (estradaExiste mapa cidadeOrigem cidadeDestino) then
        [(nome, coordenadas,atualizarEstradasAdicionadas nome estradas) |(nome, coordenadas, estradas) <- mapa]
        else mapa
        where
            atualizarEstradasAdicionadas nome estradas
                                        |cidadeOrigem == cidadeDestino = estradas
                                        |nome == cidadeOrigem = estradas ++ [cidadeDestino]
                                        |nome == cidadeDestino = estradas ++ [cidadeOrigem]
                                        |otherwise = estradas

-- Remove uma estrada entre duas cidades
removeEstrada :: Mapa.Mapa -> Mapa.Nome -> Mapa.Nome -> Mapa.Mapa
removeEstrada mapa cidadeOrigem cidadeDestino = if estradaExiste mapa cidadeOrigem cidadeDestino
    then
        [(nome, coordenadas, if nome == cidadeOrigem then [estrada | estrada <- estradas, estrada /= cidadeDestino] else
                             if nome == cidadeDestino then[estrada | estrada <- estradas, estrada /= cidadeOrigem]  else estradas) |
        (nome, coordenadas, estradas) <- mapa] else mapa
```

As funções de adição e remoção de estradas têm conceitos semelhantes. Ambas utilizam compreensão de listas, recebem um mapa e duas cidades como entrada, e retornam o mapa atualizado. Elas percorrem todas as cidades no mapa e verificam se o nome de cada cidade corresponde a uma das cidades de entrada. Se for igual à "cidadeOrigem", adicionam o nome da "cidadeDestino"às estradas dessa cidade; se for igual à "cidadeDestino", adicionam o nome da "cidadeOrigem"às estradas. Caso nenhuma dessas condições seja atendida, as estradas permanecem inalteradas. A função "removeEstrada"segue uma lógica semelhante, com a diferença de que remove as cidades da lista de estradas com base na "cidadeDestino". Essas funções são essenciais para atualizar as rotas entre duas cidades em um mapa.

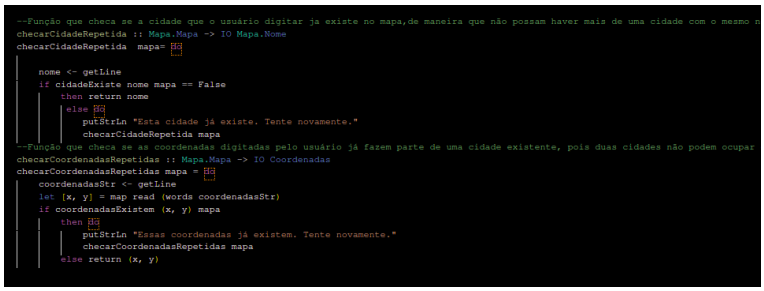
## 3.5 Funções cidade Existe, coordenadas Existem, checarCidade




Agora chegamos nas 5 funções de checagem do código que serão muito úteis para saber se o programa principal está autorizado a continuar o processo sem nenhum problema. A primeira, “cidadeExiste”, é uma função auxiliar para verificar, pelo nome, se uma cidade existe no mapa dado e retorna um booleano. Para isso, usamos a função “elem” que retorna True caso um elemento é igual a outro componente de uma lista, que nesse caso comparamos o nome da cidade com a lista de todos os nomes do mapa por meio de uma compreensão de listas. A mesma lógica é seguida em “coordenadasExistem”, mas usando os dados de coordenadas das cidades.

Após isso, temos a função “checarCidade”. Utiliza a função "cidadeExiste” para deixar o código mais robusto. Nessa função aparece duas informações novas. A primeira é que ela retorna “IO Mapa.Nome”. O “IO” significa que já começamos a trabalhar com informações do “Mundo” (seria algo como se pegássemos informações do “Mundo real” e colocássemos no “Mundo do computador ou vice-versa). Então, para pegar a informação do “Mundo real”, usamos “getLine” que serve justamente para ir ao mundo real e pedir uma informação e depois voltar com essa informação encapsulada. E para não comprometer a pureza do código funcional, utilizamos o recurso “<-”, que abre esse elemento encapsulado e coloca em “nome”. Sendo assim, se o nome da cidade existir no mapa, passa no teste, mas caso contrário, vai pedir de novo o nome de uma cidade que exista no mapa até que finalmente encontre a cidade. Essa verificação quase “infinita” ocorre por conta da recursão usada.

## 3.6 Funções checarCidadeRepetida e checarCoordenadasRepetidas




Por último, temos as funções para checar se as cidades ou as coordenadas já existem para não adicionar dados repetidos no mapa. “checarCidadeRepetida” usa a função “cidadeExiste” para analisar. Se a função auxiliar retornar false significa que não há o mesmo nome dado, então, a função retorna o nome da cidade que será usada. Caso contrário, usa uma recursão para retornar, novamente, a função, informando que há uma cidade com o mesmo nome, e, com isso, verifica de novo. E a função “checarCoordenadasRepetidas” examina se já existe a coordenada posta pelo cliente. Incrementamos as funções “map”, para usar a função “read” e “words”.

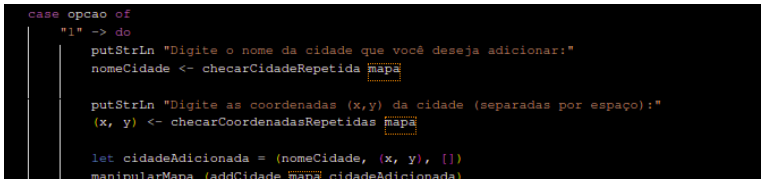
A função map serve para aplicar a função do primeiro argumento em todas os itens da lista, na qual é o segundo argumento. Já a função read serve para que as coordenadas, passadas inicialmente como string, sejam transformadas em doubles, assim podendo ser manipuladas no programa. E por último, a função words serve para separar uma string em uma lista de palavras quando se usa o espaço. Juntando as três funções, conseguimos transformar todas as coordenadas, separadas por espaço, em string para lê-las e assim informar se já existem no mapa ou não.

## 3.7 Função Manipular Mapa

```
92
93  -- Função principal para manipular o mapa
94  manipularMapa :: Mapa.Mapa -> IO ()
95  manipularMapa mapa = do
96      putStrLn "Mapa atual:"
97      print mapa
98      putStrLn "\nEscolha uma opção:"
99      putStrLn "1 - Adicionar cidade"
100     putStrLn "2 - Remover cidade"
101     putStrLn "3 - Adicionar estrada"
102     putStrLn "4 - Remover estrada"
103     putStrLn "5 - Salvar mapa"
104     putStrLn "6 - Carregar Mapa"
105     putStrLn "7 - Sair"
106     opcao <- getLine
```

A função “manipularMapa” é a responsável pela interação entre o programa e o cliente. Ela recebe um mapa e retorna IO (), já que, como dito anteriormente, utilizará dados pegos do mundo externo do computador, mas ela só tem a função de imprimir dados na tela. É por conta disso, que retorna IO vazio. No começo ela escreve o mapa atual que o usuário está utilizando e, também, diz todas as opções que o consumidor pode usar no programa. É usado a expressão “case of” para ativar a função desejada e para conseguir pegar a opção escolhida usamos, novamente, “getLine” e “<-”.

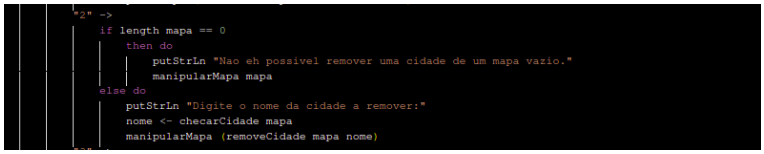
## 3.8 Função Manipular Mapa Opção 1




Na opção 1, que serve para adicionar uma cidade ao mapa, o código pede para adicionar o nome da cidade que deseja que já verificado pela função "checarCidadeRepetida" para não pôr o nome de uma cidade que já existe. É preferível que não use caracteres especiais, como acentos, pois, não utilizamos uma biblioteca para isso. Depois, as coordenadas, onde ocorre a mesma implementação, porém, usando a função "checarCoordenadasRepetidas".

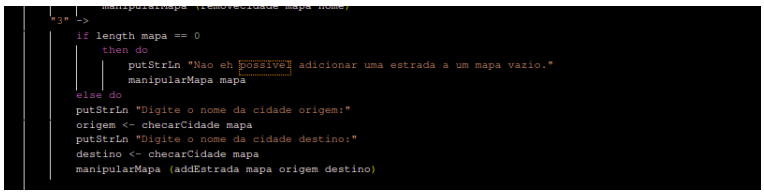
Quando todos os dados postos forem aceitos, chamamos os dados da nova cidade como "cidadeAdicionada" e, para continuar o looping, chamamos novamente a função "manipularMapa" já tendo a cidade sendo adicionada no mapa.

## 3.9 Função Manipular Mapa Opção 2



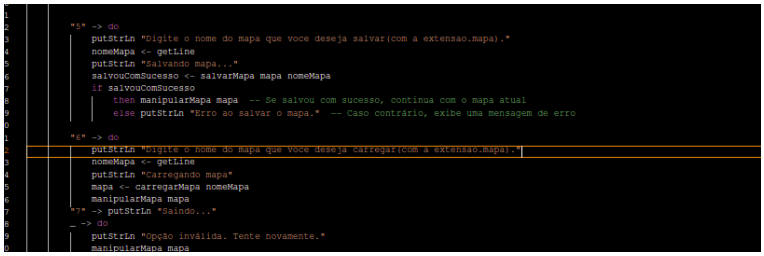

Na opção "2", que se refere à remoção de uma cidade do mapa, a verificação length mapa == 0 indica que, se o mapa estiver vazio (ou seja, não contém cidades), não faz sentido tentar remover uma cidade, pois não há cidades para remover. Nesse caso, o programa exibe uma mensagem informando que não é possível remover uma cidade de um mapa vazio e, em seguida, retorna ao menu principal, chamando manipularMapa mapa.

## 3.10 Função Manipular Mapa Opção 3 e 4




Em resumo, a verificação length mapa == 0 é comum às opções "3"e "4". Ela garante que as operações de adição e remoção de estradas, assim como de remoção de cidades, só sejam realizadas se houver pelo menos uma cidade no mapa. Se o mapa estiver vazio (sem cidades), não faz sentido executar essas operações, já que não há cidades nem estradas para modificar ou remover. Em ambos os casos, o programa exibe uma mensagem informando que não é possível executar a operação em um mapa vazio e retorna ao menu principal, chamando manipularMapa mapa

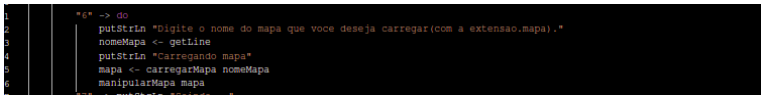
## 3.11 Função Manipular Mapa Opção 5,6 e 7




As funcionalidades 5 e 6 estão relacionadas a manipulação de mapas em arquivos. A primeira utiliza a função “salvarMapa”, que retorna um booleano, para saber se o processo de salvamento ocorreu do jeito esperado para assim utilizar o mapa na função principal do projeto ou informar se houve algum problema. Já em relação a sexta aplicação, usa a função “carregarMapa” e coloca no argumento “mapa” no qual já é chamado logo após.

Por último, caso nenhum dos números entre 1 e 7 sejam chamados, informaremos que é uma opção inválidas e retornaremos a função principal para manipular os mapas novamente. Sendo assim, utilizamos “Main” para já iniciar a função “manipularMapa” com um mapa vazio já quando entrar no programa.

### 3.12 Função Manipular Mapa 6




Em resumo, a opção "6"permite ao usuário carregar um mapa previamente salvo em um arquivo. O programa solicita o nome do arquivo, carrega os dados desse arquivo para criar um mapa em memória e, em seguida, permite que o usuário manipule esse mapa recém-carregado por meio da função manipularMapa. Essa opção é útil quando o usuário deseja continuar trabalhando com um mapa previamente salvo

## 4 Etapa 2 do Projeto

### 4.1 Função GetNome

```haskell
-- pega o nome da cidade
getNome :: Mapa.Cidade -> Mapa.Nome
getNome (nome,coordenadas,estradas) = nome
```

A função getNome é responsável por extrair o nome da cidade a partir da estrutura de dados da cidade. A função recebe como entrada uma cidade representada por uma tupla na qual o primeiro elemento é o nome da cidade, seguido pelas coordenadas e estradas. O objetivo desta função é retornar o nome da cidade, que é o primeiro elemento da tupla.

### 4.2 Função GetCoords

```haskell
--pega as coordenadas da cidade
getCoords :: Mapa.Cidade -> Coordenadas
getCoords (_,coordenadas,_) =coordenadas
```

A função getCoords foi criada para obter as coordenadas geográficas da cidade a partir da representação na forma de tupla. A função recebe como entrada uma cidade representada por uma tupla, cujo segundo elemento contém as coordenadas da cidade. A função extrai essas coordenadas e as retorna como resultado.

### 4.3 Função Getcidade

```haskell
getCoords (_,coordenadas,_) =coordenadas
-- pega todos os dados da cidade a partir do nome
getCidade :: Mapa.Nome -> Mapa.Mapa -> Mapa.Cidade
getCidade nome mapa = head [cidade | cidade <- mapa,nome == getNome cidade]
```

A função getCidade desempenha um papel importante na recuperação de todas as informações sobre uma cidade a partir de seu nome. A função recebe dois argumentos: o nome da cidade que quer buscar e um mapa completo das cidades. A função examina o mapa procurando a cidade com o nome fornecido. Quando a cidade é encontrada, ela retorna em forma de uma tupla contendo todas as informações sobre a cidade, incluindo o nome, as coordenadas e as estradas.

## 4.4 Função GetEstradas

```
getCoords (_,coordenadas,_) = coordenadas
-- pega todos os dados da cidade a partir do nome
getCidade :: Mapa.Nome -> Mapa.Mapa -> Mapa.Cidade
getCidade nome mapa = head [cidade | cidade <- mapa,nome == getNome cidade]
```

A função getEstradas foi projetada para extrair as estradas conectadas a uma cidade com base em seu nome. A função aceita dois parâmetros: o nome da cidade que deseja analisar e o mapa geral da cidade. Para alcançar esse objetivo, a função utiliza a função interna getEstradasCidade, que extrai as estradas da cidade após obter a cidade correspondente usando a função getCidade. O resultado final é uma lista das estradas que ligam a cidade especificada a outras cidades do mapa.

## 4.5 Função Addcidade *Correção da 1 etapa

```
-- Adiciona uma cidade ao mapa
addCidade :: Mapa.Mapa -> Mapa.Cidade -> Mapa.Mapa
addCidade mapa (nome,coordenadas,estradas) =  (nome,coordenadas,[]) : mapa
```

A função addCidade destina-se a adicionar uma cidade a um mapa existente. Ela recebe dois parâmetros: o primeiro parâmetro é o mapa atual, mostrado como uma lista de cidades, e o segundo parâmetro é a cidade a ser adicionada, mostrada como uma tupla contendo o nome da cidade, suas coordenadas e uma lista de estradas vazias.

A principal mudança introduzida por esta função é a adição da nova cidade ao mapa. Para fazer isso, o operador: (dois pontos) é usado para construir a nova lista. No código, a nova cidade é adicionada ao início do mapa existente e uma nova lista é gerada contendo a cidade recém-inserida, juntamente com todas as outras cidades já existentes no mapa. Quando essa função é executada, o mapa é atualizado com as cidades adicionadas e as informações sobre as cidades anteriores são mantidas.

## 4.6 Função addEstrada e removeEstrada *Correção da 1 etapa

```haskell
-- Adiciona uma estrada entre duas cidades
addEstrada :: Mapa.Mapa -> Mapa.Nome -> Mapa.Nome -> Mapa.Mapa
addEstrada mapa cidadeOrigem cidadeDestino =
    if not (estradaExiste mapa cidadeOrigem cidadeDestino) then
        [(nome, coordenadas,atualizarEstradasAdicionadas nome estradas) |(nome, coordenadas, estradas) <- mapa]
        else mapa
        where
                atualizarEstradasAdicionadas nome estradas
                                                        |cidadeOrigem == cidadeDestino = estradas
                                                        |nome == cidadeOrigem = estradas ++ [cidadeDestino]
                                                        |nome == cidadeDestino = estradas ++ [cidadeOrigem]
                                                        |otherwise = estradas

-- Remove uma estrada entre duas cidades
removeEstrada :: Mapa.Mapa -> Mapa.Nome -> Mapa.Nome -> Mapa.Mapa
removeEstrada mapa cidadeOrigem cidadeDestino = if estradaExiste mapa cidadeOrigem cidadeDestino
    then
        [(nome, coordenadas, if nome == cidadeOrigem then [estrada | estrada <- estradas, estrada /= cidadeDestino] else
                            if nome == cidadeDestino then[estrada | estrada <- estradas, estrada /= cidadeOrigem]  else estradas) |
        (nome, coordenadas, estradas) <- mapa] else mapa
```

As funções addEstrada e removeEstrada têm conceitos semelhantes. Ambas utilizam três parâmetros: o mapa atual, o nome da cidade de origem e o nome da cidade de destino. Porém addEstrada é usada para adicionar uma estrada entre duas cidades e removeEstrada tem como finalidade remover uma estrada que conecta duas cidades no mapa. A mudança mais importante em relação ao texto anterior é a maneira como a função atualiza as rotas da cidade no mapa. Primeiro, essas funções usa a função "estradaExiste" para verificar se uma estrada entre duas cidades já existe no mapa. Se a estrada não existir, a função usa seu conhecimento de listas para criar uma nova lista de cidades.

Para cada cidade no mapa, ela verifica se o nome da cidade corresponde à cidade de origem. Se sim, adiciona/remove a cidade de destino da lista de estradas dessa cidade. Da mesma forma, se o nome corresponder ao da cidade de destino, adiciona/remove a cidade de origem da lista de estradas dessa cidade. Para todas as outras cidades, as estradas permanecem inalteradas. Se a estrada já existir no addEstrada, a função retornará o mapa original inalterado e se a estrada não existe no removeEstrada, a função retorna o mapa original sem fazer nenhuma modificação.

## 4.7 Função calcularDistancia

```haskell
calcularDistancia :: Mapa.Nome -> Mapa.Nome -> Mapa.Mapa -> Double
calcularDistancia cidadeOrigem cidadeDestino mapa =
    if cidadeExiste cidadeOrigem mapa && cidadeExiste cidadeDestino mapa
    then
        sqrt((x1 - x2)^2 + (y1 - y2)^2)
    else
        error "Uma das cidades nao existe no mapa."
    where
        (x1, y1) = getCoords $ getCidade cidadeOrigem mapa
        (x2, y2) = getCoords $ getCidade cidadeDestino mapa
```

A função calcularDistancia é responsável por calcular a distância entre duas cidades no mapa, usando os nomes das cidades e o próprio mapa. Ela recebe três parâmetros: o nome da cidade de origem (cidadeOrigem) e da cidade de destino (cidadeDestino) para calcular a distância, também um mapa com informações sobre as cidades e suas coordenadas.

Em primeiro lugar, a função verifica se as cidades de origem e destino existem no mapa. Para fazer isso, ela usa a função cidadeExiste, que é definida anteriormente no código. Se uma das cidades não existir no mapa, a função lançará um erro com a mensagem "Uma das cidades não existe no mapa."

Se ambas as cidades existirem no mapa, a função continuará a calcular a distância entre elas. Para fazer isso, ela utiliza as coordenadas (x, y) das duas cidades. As coordenadas da cidade de origem são obtidas chamando getCoords com o resultado da função getCidade usando cidadeOrigem como argumento. As coordenadas da cidade de destino podem ser obtidas da mesma forma. Depois de obter as coordenadas das duas cidades, a função usa uma fórmula para determinar a distância euclidiana entre dois pontos no plano cartesiano: sqrt((x1 - x2)$^2$ + ($y1 - y2$)$^2$).

Nessa fórmula: (x1, y1) são as coordenadas da cidade de origem. (x2, y2) são as coordenadas da cidade de destino. O resultado desse cálculo é a distância entre as duas cidades, que é retornada como um valor do tipo Double.

## 4.8 Função EstradaExiste

```haskell
-- Verifica se duas cidades possuem conexão
estradaExiste :: Mapa.Mapa -> Mapa.Nome -> Mapa.Nome -> Bool
estradaExiste mapa cidadeOrigem cidadeDestino =
    cidadeDestino `elem` estradasOrigem && cidadeOrigem `elem` estradasDestino
    where
        estradasOrigem = getEstradas cidadeOrigem  mapa
        estradasDestino = getEstradas cidadeDestino  mapa
```

Essa função (estradaExiste) é responsável por verificar se há uma conexão direta (estrada) entre duas cidades no mapa. Ela recebe três parâmetros: o mapa que contém informações sobre as cidades e as estradas entre elas, o nome da cidade de origem (cidadeOrigem) e cidade destino (cidadeDestino) que se deseja verificar. A função começa obtendo as estradas das cidades de origem e destino do mapa. Para fazer isso, ela usa a função getEstradas, passando o nome da cidade como argumento. As estradas da cidade de origem são armazenadas em estradasOrigem, e as estradas da cidade de destino em estradasDestino.

Em seguida, a função verifica se a cidade de destino (cidadeDestino) está presente na lista de estradas da cidade de origem (estradasOrigem) e se a cidade de origem (cidadeOrigem) está presente na lista de estradas da cidade de destino (estradasDestino). Se ambas as condições forem verdadeiras, isso significa que há uma conexão direta (estrada) entre as duas cidades e a função retorna True. Caso contrário, ela retornará False.

## 4.9 Função RotaExiste

```haskell
-- Função para verificar se uma rota entre duas cidades existe
rotaExiste :: Mapa.Mapa -> Mapa.Nome -> Mapa.Nome -> Bool
rotaExiste mapa cidadeOrigem cidadeDestino = go [cidadeOrigem]  -- Inicia com a cidade de origem
  where
    go [] = False  -- Não há mais cidades para explorar
    go (atual:restante)
        | atual == cidadeDestino = True  -- Encontrou a cidade de destino
        | otherwise =
            let estradas = getEstradas atual mapa
                cidadesNaoVisitadas = filter (`notElem` restante) estradas
            in go (restante ++ cidadesNaoVisitadas)  -- Adiciona cidades não visitadas à fila
```

A função rotaExiste é responsável por verificar se há uma rota (uma série de estradas) entre duas cidades no mapa. Essa função é útil para determinar se é possível viajar de cidadeOrigem para cidadeDestino seguindo uma sequência de estradas no mapa. Ela recebe três parâmetros: o mapa que

contém as informações sobre as cidades e suas estradas, o nome da cidade de origem (cidadeOrigem) e cidade destino (cidadeDestino) que se deseja verificar.

A função utiliza uma abordagem de busca ampla para explorar as cidades e suas estradas em busca de uma rota entre cidadeOrigem e cidadeDestino. A função go é uma função interna que realiza a busca ampla. A função go usa uma lista contendo apenas cidadeOrigem como ponto de partida.

A cada passo da busca, a função verifica se a cidadeDestino foi alcançada (atual == cidadeDestino). Se a cidade de destino for encontrada, a função retorna True, indicando que uma rota foi encontrada. Caso contrário, a função obtém as estradas na cidade atual (estradas = getEstradas atual mapa) e cria uma lista chamada cidadesNaoVisitadas que contém as cidades para as quais ainda não foi explorada uma rota. Isso é feito filtrando as cidades não visitadas do restante da lista, incluindo as cidades que já foram visitadas ou navegadas.

Em seguida, a função adiciona as cidadesNaoVisitadas à lista restante e continua a busca, repetindo o processo até encontrar a cidadeDestino ou até não haver mais cidades para explorar (go [] = False). A função retorna False se a lista restante estiver vazia e nenhuma cidade de destino puder ser encontrada, o que significa que não há rota entre as duas cidades.

## 4.10 Função EncontrarCaminho

```haskell
-- Função para encontrar um caminho entre duas cidades
encontrarCaminho :: Mapa.Mapa -> Mapa.Nome -> Mapa.Nome -> [Mapa.Nome]
encontrarCaminho mapa cidadeOrigem cidadeDestino = go cidadeOrigem [cidadeOrigem] []
  where
    go atual _ caminho | atual == cidadeDestino = reverse (atual : caminho)
    go atual visitadas caminho =
        let estradas = getEstradas atual mapa
            cidadesNaoVisitadas = filter (`notElem` visitadas) estradas
        in case findPath cidadesNaoVisitadas of
            (proximaCidade:_) -> go proximaCidade (proximaCidade : visitadas) (atual : caminho)
            [] -> case caminho of
                [] -> []  -- Não há caminho possível
                _ -> go (head caminho) visitadas (tail caminho)  -- Retrocede

    findPath [] = []  -- Não há mais estradas a explorar
    findPath (proximaCidade:cidadesRestantes)
        | rotaExiste mapa proximaCidade cidadeDestino = [proximaCidade]
        | otherwise = findPath cidadesRestantes
```

A função encontrarCaminho é responsável por encontrar um caminho entre duas cidades no mapa, com base nos nomes das cidades e no próprio mapa. A função usa um método de busca em profundidade para encontrar o caminho. A função recebe três parâmetros: o mapa que contém as informações sobre as cidades e suas estradas, o nome da cidade de origem (cidadeOrigem) e cidade de destino (cidadeDestino) do caminho que se deseja encontrar.

A função go interna é usada para realizar uma pesquisa de profundidade. Ela inicia a busca na cidadeOrigem, com uma lista de visitadas contendo apenas a cidadeOrigem e uma lista vazia de caminho. A função go possui três casos: Se a cidadeAtual for igual à cidadeDestino, isso significa que encontramos o destino, e a função retorna o caminho encontrado até o momento, o qual é invertido usando reverse. No entanto, se a cidadeAtual não for igual à cidadeDestino, a função obtém as estradas da cidadeAtual usando

getEstradas e filtra as cidades que ainda não foram visitadas. Nesse ponto, a função findPath é chamada para encontrar uma cidade próxima que leve ao destino. Se encontrar uma cidade, a busca continua a partir dessa cidade ( go proximaCidade ...).

Se não encontrar uma cidade próxima, a função considera retroceder para a cidade anterior no caminho (go (head caminho) ...), eliminando a cidade atual do caminho (tail caminho). A função findPath é uma função auxiliar que busca na lista de cidades não visitadas por uma cidade que leva ao destino (cidadeDestino). Se encontrar uma cidade, ela a retorna como uma lista, caso contrário, retorna uma lista vazia. Assim a função encontrarCaminho retorna o caminho entre a cidade de origem e destino como uma lista de nomes de cidades.

## 4.11 Função calcularSomaDistancias e calcularDistanciaRota

```haskell
calcularSomaDistancias :: Mapa.Mapa -> [Mapa.Nome] -> Double
calcularSomaDistancias _ [] = 0.0
calcularSomaDistancias mapa [_] = 0.0
calcularSomaDistancias mapa (cidade1:cidade2:cidadesRestantes) =
                    calcularDistancia cidade1 cidade2 mapa + calcularSomaDistancias mapa (cidade2:cidadesRestantes

calcularDistanciaRota :: Mapa.Mapa -> Mapa.Nome -> Mapa.Nome -> Double
calcularDistanciaRota mapa origem destino
                                        |rotaExiste mapa origem destino = calcularSomaDistancias mapa caminho
                                        |otherwise = error "nao ha rota valida"
                                        where
                                            caminho = encontrarCaminho mapa origem destino
```

As funções calcularSomaDistancias e calcularDistanciaRota são usadas para calcular a soma das distâncias entre as cidades na rota e a distância total da rota entre essas duas cidades no mapa, respectivamente. A função calcularSomaDistancias recebe como argumentos o mapa e uma lista de nomes de cidades representando uma rota, enquanto a função calcularDistanciaRota recebe como argumentos o mapa, o nome da cidade de origem e o nome da cidade de destino.

Começando com calcularSomaDistancias, a função lida com três casos diferentes: se a lista estiver vazia ([ ]), a distância total é 0.0, pois não há cidades para calcular a distância entre elas. Se a lista tiver apenas um elemento ([ ]), também retorna 0.0, pois não há uma rota válida com apenas uma cidade. Se a lista tiver pelo menos duas cidades, a função começa calculando a distância entre as duas primeiras cidades da lista (cidade1 e cidade2) usando a função calcularDistancia. Em seguida, recursivamente, chama a função para calcular a distância entre cidade2 e o restante das cidades na lista (cidadesRestantes), adicionando o resultado ao resultado anterior. O processo continua até que todas as distâncias entre as cidades da lista tenham sido somadas, e o resultado final é a distância total da rota.

Já em calcularDistanciaRota a função verifica se existe uma rota válida entre a cidade de origem e a de destino usando a função rotaExiste. Se não houver uma rota válida, a função gera um erro com a mensagem "não há rota válida". Se existir uma rota válida, chama a função encontrarCaminho para obter o caminho (lista de cidades) entre a cidade de origem e a de destino. Em seguida, a função chama a função calcularSomaDistancias para calcular a distância total da rota, passando o mapa e o caminho como argumentos. O resultado é a distância total da rota entre as duas cidades.

## 4.12 Função checarEstradaRepetida

```haskell
checarEstradaRepetida :: Mapa.Mapa -> Mapa.Nome -> Mapa.Nome -> IO (Mapa.Nome, Mapa.Nome)
checarEstradaRepetida mapa origem destino = do
    if estradaExiste mapa origem destino
        then do
            putStrLn "Essa estrada já existe. Tente novamente."
            putStrLn "Digite novamente a cidade origem:"
            origemNova <- checarCidade mapa
            putStrLn "Digite novamente a cidade destino:"
            destinoNova <- checarCidade mapa
            checarEstradaRepetida mapa origemNova destinoNova
        else return (origem, destino)
```

A função “checarEstradaRepetida” serve para limitar a criação de estradas no mapa. Ela usa a função “estradaExiste” como função auxiliar para assim verificar as estradas e assim continuar o código sem problemas.

## 4.13 Função ManipularMapa

```haskell
-- Função principal para manipular o mapa
manipularMapa :: Mapa.Mapa -> IO ()
manipularMapa mapa = do
    putStrLn "Mapa atual:"
    print mapa
    putStrLn "\nEscolha uma opção:"
    putStrLn "1 - Adicionar cidade"
    putStrLn "2 - Remover cidade"
    putStrLn "3 - Adicionar estrada"
    putStrLn "4 - Remover estrada"
    putStrLn "5 - Calcular distancia entre duas cidades(pela rota)"
    putStrLn "6 - Mostrar Rota entre duas cidades"
    putStrLn "7 - Mostrar vizinhança de uma cidade"
    putStrLn "8 - Salvar mapa"
    putStrLn "9 - Carregar Mapa"
    putStrLn "10 - Sair"
    opcao <- getLine
```

A função "manipularMapa"teve algumas alterações. Primeiramente, adicionamos novas funções a ela. O método para calcular a distância entre duas cidades pelas rotas, outra que mostra a rota entre duas cidades, e a sétima revela a vizinhança de uma cidade. Em segundo lugar, fizemos alguns ajustes para um código mais robusto, como, por exemplo, verificar se o mapa é vazio antes de tentar remover uma cidade ou adicionar estradas.

A opção 5 do "case of"está relacionada à parte de calcular a distância da rota. Primeiramente, verifica se o mapa está vazio e, caso contrário, pede para o usuário informar as cidades. Se a rota entre as cidades fornecidas existir, o que verificamos com a função "rotaExiste", então informamos a distância por meio da função "calcularDistanciaRota". Se a função "rotaExiste"retornar falso, significa que não existem rotas entre as cidades e, consequentemente, não é possível calcular a distância da rota.

A opção 6 mostra o caminho entre uma cidade e outra. Primeiramente, verifica se existe uma rota entre as cidades novamente com a função "rotaExiste". Se existir, informa um dos possíveis percursos pela função "encontrarCaminho".

Por último, das funções novas, a opção 7 serve para informar todas as estradas nas quais uma determinada cidade inserida pelo usuário queira saber. Se o mapa não for vazio, exibe a lista de cidades por meio da função "getEstradas"

# Referências

[1] Beatriz, Professora. *Material de Aula de Programação Funcional*. 2023.

[2] LaTeX Project. *LaTeX - A document preparation system*. Disponível em: https://www.latex-project.org/.