Listas de Exercícios em Haskell

Lista 1 - Funções Básicas

1. Perímetro do Círculo

Enunciado: Crie uma função que recebe o raio (Double) de um círculo e retorna o perímetro dele.

```haskell
perimetroCirculo :: Double -> Double
perimetroCirculo r = 2 * pi * r
```

Explicação: A função calcula o perímetro usando a fórmula $2\pi r$, onde r é o raio fornecido.

2. Média de 3 Valores

Enunciado: Crie uma função que recebe 3 valores Float e calcula a média entre eles.

```haskell
media3 :: Float -> Float -> Float -> Float
media3 a b c = (a + b + c) / 3.0
```

Explicação: A função soma os três valores e divide por 3 para obter a média aritmética.

3. Verificar Dígito

Enunciado: Crie uma função que recebe um Char e indica se é um dígito ou não.

```haskell
ehDigito :: Char -> Bool
ehDigito c = c >= '0' && c <= '9'
```

Explicação: Verifica se o caractere está entre '0' e '9' na tabela ASCII.

4. Converter para Maiúscula

Enunciado: Crie uma função que converte uma letra minúscula (Char) para maiúscula.

```haskell
paraMaiuscula :: Char -> Char
paraMaiuscula c
    | c >= 'a' && c <= 'z' = toEnum (fromEnum c - 32) :: Char
    | otherwise = c
```

Explicação: Converte minúsculas para maiúsculas subtraindo 32 do valor ASCII. Outros caracteres permanecem inalterados.

5. Operador de Média

Enunciado: Crie um operador que calcula a média entre 2 valores do tipo Double.

```haskell
(</>) :: Double -> Double -> Double
a </> b = (a + b) / 2.0
```

Explicação: Operador personalizado que calcula a média aritmética de dois números.

6. Soma de Intervalo

Enunciado: Crie uma função que calcula a soma entre dois inteiros (conjunto fechado).

```haskell
somaIntervalo :: Int -> Int -> Int
somaIntervalo a b = sum [min a b .. max a b]
```

Explicação: Cria uma lista do menor ao maior número e calcula a soma dos elementos.

7. Operador de Média de Intervalo

Enunciado: Crie um operador que calcula a média dos números entre dois inteiros (conjunto fechado).

```
(<//>) :: Int -> Int -> Double
a <//> b = fromIntegral (somaIntervalo a b) / fromIntegral (abs (b - a) + 1)
```

Explicação: Usa a função somaIntervalo e divide pelo número de elementos no intervalo.

8. Potência

Enunciado: Crie uma função que calcula a potência entre dois inteiros (não use o operador ^).

```
potencia :: Int -> Int -> Int
potencia _ 0 = 1
potencia base exp = base * potencia base (exp - 1)
```

Explicação: Implementação recursiva da potência usando multiplicações sucessivas.

9. Operador de Resto

Enunciado: Crie um operador que calcula o resto da divisão entre inteiros positivos (não use as funções mod ou rem).

```
(<%>) :: Int -> Int -> Int
a <%> b
    | a < b = a
    | otherwise = (a - b) <%> b
```

Explicação: Implementação recursiva do resto da divisão usando subtração.

10. Máximo Divisor Comum

Enunciado: Crie uma função que calcula o mdc entre dois inteiros positivos (não use a função gcd).

```
mdc :: Int -> Int -> Int
mdc a 0 = a
mdc a b = mdc b (a <%> b)
```

Explicação: Implementação do algoritmo de Euclides usando o operador de resto definido anteriormente.

11. Mínimo Múltiplo Comum

Enunciado: Crie uma função que calcula o mmc entre dois inteiros positivos (não use a função lcm).

```
mmc :: Int -> Int -> Int
mmc a b = (a * b) `div` (mdc a b)
```

Explicação: Usa a relação matemática entre MDC e MMC: $MDC(a, b) \times MMC(a, b) = a \times b$.

12. Contar Algarismos

Enunciado: Crie uma função que recebe um Int e retorna quantos algarismos ele possui.

```
contaAlgarismos :: Int -> Int
contaAlgarismos n
    | n < 0 = contaAlgarismos (abs n)
    | n < 10 = 1
    | otherwise = 1 + contaAlgarismos (n `div` 10)
```

Explicação: Conta recursivamente os dígitos dividindo por 10 até chegar a um número menor que 10.

13. Contar Ocorrências

Enunciado: Crie uma função que recebe dois Int, sendo o primeiro um inteiro positivo ou negativo com qualquer quantidade de algarismos e o segundo um único algarismo (positivo). A função deve retornar quantas vezes o segundo algarismo aparece no primeiro inteiro.

```
contaOcorrencias :: Int -> Int -> Int
contaOcorrencias n d
    | n < 0 = contaOcorrencias (abs n) d
    | n < 10 = if n == d then 1 else 0
    | otherwise = (if (n `mod` 10) == d then 1 else 0) + contaOcorrencias (n `div` 10) d
```

Explicação: Verifica cada dígito do número usando módulo 10 e divisão por 10, contando as ocorrências.

14. Converter para Binário

Enunciado: Crie uma função que recebe um inteiro positivo e retorna uma String que corresponde ao inteiro convertido para binário.

```
paraBinario :: Int -> String
paraBinario 0 = "0"
paraBinario n
    | n < 0 = error "N mero negativo n o suportado"
    | otherwise = paraBinarioAux n ""
    where
        paraBinarioAux 0 str = str
        paraBinarioAux n str = paraBinarioAux (n `div` 2) (show (n `mod` 2) ++ str)
```

Explicação: Converte o número para binário usando divisões sucessivas por 2 e concatenando os restos.

15. Soma de Algarismos

Enunciado: Crie uma função que calcule a soma dos algarismos de um número inteiro.

```
somaAlgarismos :: Int -> Int
somaAlgarismos n
    | n < 0 = somaAlgarismos (abs n)
    | n < 10 = n
    | otherwise = (n `mod` 10) + somaAlgarismos (n `div` 10)
```

Explicação: Soma recursivamente cada dígito do número usando módulo 10 e divisão por 10.

16. Função de Ackermann

Enunciado: Crie uma função que calcula um elemento da série de Ackermann, que recebe dois inteiros não negativos.

```
ackermann :: Int -> Int -> Int
ackermann 0 n = n + 1
ackermann m 0 = ackermann (m - 1) 1
ackermann m n = ackermann (m - 1) (ackermann m (n - 1))
```

Explicação: Implementação direta da função de Ackermann, uma função recursiva não primitiva que cresce extremamente rápido.

Lista 2 - Trabalhando com Listas

1. Média de Lista

Enunciado: Implemente uma função que recebe uma lista do tipo Double e retorne a média dos elementos dessa lista.

```
mediaLista :: [Double] -> Double
mediaLista xs = sum xs / fromIntegral (length xs)
```

Explicação: Calcula a soma dos elementos e divide pelo número de elementos.

2. Inverter Lista

Enunciado: Implemente uma função que inverte os elementos de uma lista (não use a função reverse).

```
inverteLista :: [a] -> [a]
inverteLista [] = []
inverteLista (x:xs) = inverteLista xs ++ [x]
```

Explicação: Inverte a lista recursivamente movendo a cabeça para o final.

3. N-ésimo Elemento

Enunciado: Implemente uma função que informe o n-ésimo membro de uma lista (não use o operador !!).

```
enesimo :: Int -> [a] -> a
enesimo _ [] = error " ndice  fora da lista"
enesimo 0 (x:_) = x
enesimo n (_:xs) = enesimo (n - 1) xs
```

Explicação: Percorre a lista recursivamente até chegar ao índice desejado.

4. String para Maiúsculas

Enunciado: Implemente uma função que recebe uma String e a retorna com todas as letras em maiúscula.

```
paraMaiusculas :: String -> String
paraMaiusculas = map toUpper
    where
        toUpper c
            | c >= 'a' && c <= 'z' = toEnum (fromEnum c - 32) :: Char
            | otherwise = c
```

Explicação: Aplica a conversão para maiúsculas em cada caractere da string.

5. Capitalizar Frase

Enunciado: Implemente uma função que recebe uma frase (String) e a retorna com as letras iniciais de cada palavra em maiúscula.

```
capitalizaFrase :: String -> String
capitalizaFrase [] = []
capitalizaFrase (x:xs) = toUpper x : capitalizaPalavra xs
    where
        capitalizaPalavra [] = []
        capitalizaPalavra (x:xs)
            | x == ' '  = ' ' : capitalizaFrase xs
            | otherwise = x : capitalizaPalavra xs
```

Explicação: Capitaliza a primeira letra de cada palavra identificada por espaços.

6. Filtrar Letras

Enunciado: Implemente uma função que recebe uma String e retorna outra somente com as letras.

```
filtraLetras :: String -> String
filtraLetras = filter (\c -> (c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z'))
```

Explicação: Filtra apenas caracteres que são letras (maiúsculas ou minúsculas).

7. Dividir Lista

Enunciado: Implemente uma função que recebe uma lista e um inteiro t e retorna uma lista de listas que corresponde a lista original dividida em listas de tamanho t.

```
divideLista :: Int -> [a] -> [[a]]
divideLista _ [] = []
divideLista t xs = take t xs : divideLista t (drop t xs)
```

Explicação: Divide a lista em sublistas de tamanho t usando take e drop.

8. Verificar Palíndromo

Enunciado: Implemente uma função que verifica se uma lista é ou não um palíndromo.

```
ehPalindromo :: Eq a => [a] -> Bool
ehPalindromo xs = xs == inverteLista xs
```

Explicação: Compara a lista com sua versão invertida para verificar se é palíndromo.

9. Números Primos

Enunciado: Implemente uma função que recebe um inteiro e retorna uma lista de todos os números primos iguais ou inferiores a esse inteiro.

```
primosAte :: Int -> [Int]
primosAte n = filter isPrime [2..n]
    where
        isPrime x = null [y | y <- [2..x-1], x `mod` y == 0]
```

Explicação: Filtra os números primos verificando se não têm divisores além de 1 e eles mesmos.

10. Concatenar Listas

Enunciado: Implemente uma função que recebe uma lista de listas e retorna uma lista normal com todos os elementos das listas internas (não use a função concat).

```
concatenaListas :: [[a]] -> [a]
concatenaListas [] = []
concatenaListas (xs:xss) = xs ++ concatenaListas xss
```

Explicação: Concatena recursivamente todas as sublistas em uma única lista.

11. Média de Listas

Enunciado: Implemente uma função que recebe uma lista de listas do tipo Double e retorna uma lista com a média dos elementos de cada lista interna.

```
mediaListas :: [[Double]] -> [Double]
mediaListas = map mediaLista
```

Explicação: Aplica a função mediaLista a cada sublista.

Lista 3 - Recursão e Funções Avançadas

1. Inversão de Lista com Recursão em Cauda

Enunciado: Crie uma função que inverte uma lista usando recursão em cauda.

```
inverteListaTail :: [a] -> [a]
inverteListaTail xs = inverteAux xs []
  where
    inverteAux [] acc     = acc
    inverteAux (x:xs) acc = inverteAux xs (x:acc)
```

Explicação: Usa um acumulador para construir a lista invertida de forma eficiente.

2. Resultado Final

Enunciado: Crie uma função que recebe uma lista de notas (Float) de alunos, calcula a média e retorna uma String como resultado.

```
resultadoFinal :: [Float] -> String
resultadoFinal notas
  | media < 4.0  = "REPROVADO"
  | media < 6.0  = "RECUPERACAO"
  | otherwise    = "APROVADO"
  where
    media = sum notas / fromIntegral (length notas)
```

Explicação: Calcula a média e retorna o resultado baseado em faixas de valores.

3. Sequência de Fibonacci

Enunciado: Crie uma função que recebe um inteiro n e retorna uma lista com os n primeiros elementos da série de Fibonacci.

```
fibonacci :: Int -> [Integer]
fibonacci n = take n fibSeq
  where
    fibSeq = 0 : 1 : zipWith (+) fibSeq (tail fibSeq)
```

Explicação: Gera a sequência de Fibonacci infinitamente usando lazy evaluation e pega os n primeiros elementos.

4. Equação do Segundo Grau

Enunciado: Crie a função equacao2grau a b c que retorna as raízes reais da equação do segundo grau ($ax^2 + bx + c = 0$).

```
equacao2grau :: Float -> Float -> Float -> [Float]
equacao2grau a b c
  | a == 0    = error "O coeficiente 'a' n o pode ser zero"
  | delta < 0 = []
  | otherwise = [(-b - sqrt delta) / (2*a), (-b + sqrt delta) / (2*a)]
  where
    delta = b^2 - 4*a*c
```

Explicação: Calcula as raízes usando a fórmula de Bhaskara, retornando lista vazia se não houver raízes reais.

5. Dividir em Partes

Enunciado: Crie uma função que recebe um inteiro n e uma lista e retorna uma lista de listas que corresponde à lista original dividida em n partes.

```
divideEmPartes :: Int -> [a] -> [[a]]
divideEmPartes n xs
  | n <= 0    = error "N mero de partes deve ser positivo"
  | otherwise = divideAux n (length xs) xs
  where
    divideAux _ _ [] = []
    divideAux n len xs = take partSize xs : divideAux n len (drop partSize xs)
      where
        partSize = ceiling (fromIntegral len / fromIntegral n)
```

Explicação: Divide a lista em n partes aproximadamente iguais calculando o tamanho de cada parte.

Lista 4 - Tuplas e Tipos de Dados

1. Mínimo e Máximo

Enunciado: Crie uma função que recebe uma lista e retorna o menor e o maior valor dessa lista.

```
minMax :: Ord a => [a] -> (a, a)
minMax [] = error "Lista vazia"
minMax [x] = (x, x)
minMax (x:xs) = (min x minTail, max x maxTail)
  where
    (minTail, maxTail) = minMax xs
```

Explicação: Encontra recursivamente o menor e maior elemento da lista.

2. Verificar Primo

Enunciado: Crie uma função que verifica se um inteiro é primo.

```haskell
ehPrimo :: Int -> (Bool, Int)
ehPrimo n
  | n < 2     = (False, 1)
  | otherwise = verificaPrimo n 2
  where
    verificaPrimo n d
      | d*d > n      = (True, n)
      | n `mod` d == 0 = (False, d)
      | otherwise      = verificaPrimo n (d+1)
```

Explicação: Verifica se o número é primo testando divisores até sua raiz quadrada.

3. MDC e MMC

Enunciado: Crie uma função que recebe dois inteiros e retorna um tupla com o MDC e o MMC.

```haskell
mdcMmc :: Int -> Int -> (Int, Int)
mdcMmc a b = (mdc a b, mmc a b)
  where
    mdc x 0 = x
    mdc x y = mdc y (x `mod` y)
    mmc x y = (x * y) `div` mdc x y
```

Explicação: Usa o algoritmo de Euclides para MDC e a relação MDC×MMC para calcular o MMC.

4. Dias do Mês

Enunciado: Crie uma função que recebe um ano e um mês e retorna o número de dias do mês, considerando anos bissextos.

```haskell
diasMes :: Int -> Int -> Int
diasMes ano mes
  | mes == 2 && bissexto = 29
  | mes == 2             = 28
  | mes `elem` [4,6,9,11] = 30
  | otherwise            = 31
  where
    bissexto = (ano `mod` 400 == 0) || (ano `mod` 100 /= 0 && ano `mod` 4 == 0)
```

Explicação: Retorna os dias do mês considerando fevereiro em anos bissextos.

5. Tipo Data

Enunciado: Definição do tipo Data e funções para manipulação.

```haskell
type Data = (Int, Int, Int)  -- (dia, m s , ano)

datasIguais :: Data -> Data -> Bool
datasIguais (d1,m1,a1) (d2,m2,a2) = d1 == d2 && m1 == m2 && a1 == a2

diferencaDias :: Data -> Data -> Int
diferencaDias (d1,m1,a1) (d2,m2,a2) = abs (diasDesdeZero a1 m1 d1 - diasDesdeZero a2 m2
    d2)
  where
    diasDesdeZero ano mes dia = -- Implementa  o simplificada
      dia + (mes-1)*30 + (ano-1)*365
```

Explicação: Define um tipo para datas e funções básicas de comparação e diferença.

6-8. Agenda e Compromissos

Enunciado: Definição de tipos para compromissos e agenda, com funções de busca.

```haskell
type Compromisso = (String, Data)
type Agenda = [Compromisso]

compromissosNaData :: Agenda -> Data -> [String]
compromissosNaData agenda dataBusca =
  [desc | (desc, dataComp) <- agenda, datasIguais dataComp dataBusca]
```

```haskell

dataMaisCompromissos :: Agenda -> Data
dataMaisCompromissos agenda =
  fst $ foldl1 (\a b -> if snd a >= snd b then a else b) contagem
  where
    contagem = [(dataComp, length $ compromissosNaData agenda dataComp) | (_, dataComp)
    <- agenda]

diasParaCompromisso :: Agenda -> String -> Data -> (Data, Int)
diasParaCompromisso agenda descricao dataAtual =
  case [dataComp | (desc, dataComp) <- agenda, desc == descricao] of
    []       -> (dataAtual, 0)
    (dataComp:_) -> (dataComp, diferencaDias dataAtual dataComp)
```

Explicação: Implementa uma agenda simples com funções para buscar compromissos por data, encontrar a data com mais compromissos e calcular dias restantes para um compromisso.

Lista 5 - Funções de Alta Ordem

1. Implementação de foldl1

Enunciado: Implemente uma função que equivale à função foldl1.

```haskell
meuFoldl1 :: (a -> a -> a) -> [a] -> a
meuFoldl1 _ []     = error "Lista vazia"
meuFoldl1 _ [x]    = x
meuFoldl1 f (x:xs) = foldl f x xs
```

Explicação: Implementação de foldl1 que aplica a função da esquerda para a direita com o primeiro elemento como valor inicial.

2. Implementação de foldr1

Enunciado: Implemente uma função que equivale à função foldr1.

```haskell
meuFoldr1 :: (a -> a -> a) -> [a] -> a
meuFoldr1 _ []     = error "Lista vazia"
meuFoldr1 _ [x]    = x
meuFoldr1 f (x:xs) = f x (meuFoldr1 f xs)
```

Explicação: Implementação de foldr1 que aplica a função da direita para a esquerda.

3. Função algum

Enunciado: Implemente uma função que recebe uma função a-¿Bool e uma lista e retorna True se algum elemento dessa lista obtém True com a função de entrada.

```haskell
algum :: (a -> Bool) -> [a] -> Bool
algum _ [] = False
algum p (x:xs) = p x || algum p xs
```

Explicação: Verifica se pelo menos um elemento da lista satisfaz a condição.

4. Função primeiro

Enunciado: Implemente uma função que recebe uma função a-¿Bool e uma lista e retorna o primeiro elemento dessa lista que obtém True com a função recebida.

```haskell
primeiro :: (a -> Bool) -> [a] -> a
primeiro _ [] = error "Nenhum elemento satisfaz a condi  o"
primeiro p (x:xs)
    | p x       = x
    | otherwise = primeiro p xs
```

Explicação: Retorna o primeiro elemento que satisfaz a condição ou erro se nenhum satisfizer.

5. Aplicação em Cadeia

Enunciado: Implemente uma função que recebe um valor e uma lista de funções unárias e aplica cada função sobre o resultado da anterior.

```
aplicaCadeia :: a -> [a -> a] -> a
aplicaCadeia x [] = x
aplicaCadeia x (f:fs) = aplicaCadeia (f x) fs
```

Explicação: Aplica uma cadeia de funções a um valor inicial, passando o resultado de uma para a próxima.

6. Aplicação de Pares

Enunciado: Implemente uma função que recebe uma lista de funções unárias e uma lista de valores e retorna uma lista com os resultados correspondentes.

```
aplicaPares :: [a -> b] -> [a] -> [b]
aplicaPares [] _ = []
aplicaPares _ [] = []
aplicaPares (f:fs) (x:xs) = f x : aplicaPares fs xs
```

Explicação: Aplica cada função ao elemento correspondente na lista de valores.

7. Filtrar e Mapear

Enunciado: Implemente uma função que filtra e depois mapeia uma lista de inteiros.

```
filtraEMapeia :: (Int -> Bool) -> (Int -> Int) -> [Int] -> [Int]
filtraEMapeia p f xs = map f (filter p xs)
```

Explicação: Filtra os elementos que satisfazem a condição e depois aplica a função de mapeamento.

8. Cabeça da Lista

Enunciado: Implemente uma função que obtém a cabeça de uma lista usando a composição das funções last e reverse.

```
cabeca :: [a] -> a
cabeca = last . reverse
```

Explicação: Inverte a lista e pega o último elemento, que é equivalente ao primeiro da lista original.