# ATV 2 - Paradigmas e Tecnologias Emergentes

## Capítulo 4

### Quest. 4.3 - Filtro de Pares e Impares

```
-- Filtra pares
filtraPares :: [Int] -> [Int]
filtraPares = filter even

-- Filtra ímpares
filtraImpares :: [Int] -> [Int]
filtraImpares = filter odd

-- Função principal
main :: IO ()
main = do
    let lista = [1..10]
    putStrLn $ "Lista original: " ++ show lista
    putStrLn $ "Pares: " ++ show (filtraPares lista)
    putStrLn $ "Ímpares: " ++ show (filtraImpares lista)
```

```
Lista original: [1,2,3,4,5,6,7,8,9,10]
Pares: [2,4,6,8,10]
Ímpares: [1,3,5,7,9]
```

### Quest. 4.4 - Filtro de Impares

```
ehPrimo :: Int -> Bool
ehPrimo n = null [x | x <- [2..n-1], n `mod` x == 0]

filtrarPrimos :: [Int] -> [Int]
filtrarPrimos = filter ehPrimo

-- Função principal
main :: IO ()
main = do
    let lista = [1..10]
    putStrLn $ "Lista original: " ++ show lista
    putStrLn $ "Primos: " ++ show (filtrarPrimos lista)
```

```
Lista original: [1,2,3,4,5,6,7,8,9,10]
Primos: [1,2,3,5,7]
```

### Quest. 4.6 - Reverter

```
reverter :: String -> String
reverter [] = []
reverter (x:xs) = reverter xs ++ [x]

main :: IO ()

palavraOriginal :: String
palavraOriginal = "Haskell"

main = do
    putStrLn $ "Palavra original: " ++ show palavraOriginal
    putStrLn $ "Palavra invertida: " ++ show (reverter palavraOriginal)
```

```
Palavra original: "Haskell"
Palavra invertida: "lleksaH"
```

### Quest. 4.7 - Filtrar Terças

```haskell
-- Define um tipo enumerado para os dias da semana
data Dia = Segunda | Terca | Quarta | Quinta | Sexta | Sabado | Domingo
    deriving (Eq, Show)

-- Função para filtrar as terças-feiras
filtrarTercas :: [Dia] -> [Dia]
filtrarTercas = filter (== Terca)

-- Lista de dias
dias :: [Dia]
dias = [Segunda, Terca, Quarta, Quinta, Sexta, Sabado, Domingo]

-- Função principal
main :: IO ()
main = do
    putStrLn $ "Lista original: " ++ show dias
    putStrLn $ "Lista filtrada: " ++ show (filtrarTercas dias)
```

```
Lista original: [Segunda,Terca,Quarta,Quinta,Sexta,Sabado,Domingo]
Lista filtrada: [Terca]
```

### Quest. 4.8 - Conversão

```haskell
-- Define um tipo para representar Dinheiro em Real ou Dólar
data Dinheiro = Real Double | Dolar Double deriving (Show)

-- Função para converter Real para Dólar
converterParaDolar :: Dinheiro -> Dinheiro
converterParaDolar (Real valor) = Dolar (valor / 5)
converterParaDolar (Dolar valor) = Dolar valor

-- Função para converter uma lista de Dinheiro para Dólar
converterListaParaDolar :: [Dinheiro] -> [Dinheiro]
converterListaParaDolar = map converterParaDolar

-- Função para converter Real para Dólar
converterParaReal :: Dinheiro -> Dinheiro
converterParaReal (Real valor) = Real valor
converterParaReal (Dolar valor) = Real (valor * 5)

-- Função para converter uma lista de Dinheiro para Real
converterListaParaReal :: [Dinheiro] -> [Dinheiro]
converterListaParaReal = map converterParaReal

-- Função para filtrar apenas os Dólares de uma lista
filtrarDolares :: [Dinheiro] -> [Dinheiro]
filtrarDolares = filter ehDolar
    where
        ehDolar (Dolar _) = True
        ehDolar _ = False

-- Função para somar os valores em Dólar de uma lista
somarDolares :: [Dinheiro] -> Double
somarDolares xs = foldl somarValoresDolar 0 (filtrarDolares xs)
    where
        somarValoresDolar acc (Dolar valor) = acc + valor
        somarValoresDolar acc _ = acc

-- Função para contar a quantidade de Dólares em uma lista
contarDolares :: [Dinheiro] -> Int
contarDolares xs = foldl contarValoresDolar 0 (filtrarDolares xs)
    where
        contarValoresDolar acc (Dolar _) = acc + 1
        contarValoresDolar acc _ = acc

-- Função principal para testar as funções acima
main :: IO ()
main = do
    let listaDinheiro = [Real 10.0, Dolar 3.0, Real 20.0, Dolar 5.0]

    putStrLn $ "Lista original: " ++ show listaDinheiro
    putStrLn $ "Lista convertida para Dólar: " ++ show (converterListaParaDolar listaDinheiro)
    putStrLn $ "Lista convertida para Real: " ++ show (converterListaParaReal listaDinheiro)
    putStrLn $ "Somatório dos valores em Dólar: " ++ show (somarDolares listaDinheiro)
    putStrLn $ "Quantidade de Dólares: " ++ show (contarDolares listaDinheiro)
```

```
Lista original: [Real 10.0,Dolar 3.0,Real 20.0,Dolar 5.0]
Lista convertida para Dólar: [Dolar 2.0,Dolar 3.0,Dolar 4.0,Dolar 5.0]
Lista convertida para Real: [Real 10.0,Real 15.0,Real 20.0,Real 25.0]
Somatório dos valores em Dólar: 8.0
Quantidade de Dólares: 2
```

### Quest. 4.9 - Exercício ai

```haskell
-- Define um tipo para os dias da semana
data DiaSemana = Domingo | Segunda | Terca | Quarta | Quinta | Sexta | Sabado
    deriving (Show, Eq)

-- Função para contar quantos números negativos há em uma lista
contarNegativos :: [Int] -> Int
contarNegativos = foldl (\acc x -> if x < 0 then acc + 1 else acc) 0

-- Função para contar quantas vezes a letra 'P' aparece em uma string
contarLetrasP :: String -> Int
contarLetrasP = foldl (\acc x -> if x == 'P' then acc + 1 else acc) 0

-- Função para contar quantos sábados há em uma lista de dias
contarSabados :: [DiaSemana] -> Int
contarSabados = foldl (\acc dia -> if dia == Sabado then acc + 1 else acc) 0

-- Função para converter um dia da semana em um número inteiro
diaParaInt :: DiaSemana -> Int
diaParaInt Domingo = 1
diaParaInt Segunda = 2
diaParaInt Terca = 3
diaParaInt Quarta = 4
diaParaInt Quinta = 5
diaParaInt Sexta = 6
diaParaInt Sabado = 7

-- Função para somar os valores inteiros correspondentes aos dias de uma lista de dias
somaDias :: [DiaSemana] -> Int
somaDias = foldl (\acc dia -> acc + diaParaInt dia) 0

-- Função principal para testar as funções acima
main :: IO ()
main = do
    let numeros = [-1, 2, -3, 4, -5]
    let texto = "PPPPPPQQQPPP"
    let diasSemana = [Segunda, Sabado, Quarta, Sabado, Sexta, Sabado]

    putStrLn $ "Lista de números: " ++ show numeros
    putStrLn $ "Quantidade de negativos: " ++ show (contarNegativos numeros)

    putStrLn $ "Texto: " ++ texto
    putStrLn $ "Quantidade de letras 'P': " ++ show (contarLetrasP texto)

    putStrLn $ "Lista de dias da semana: " ++ show diasSemana
    putStrLn $ "Quantidade de sábados: " ++ show (contarSabados diasSemana)

    putStrLn $ "Soma dos valores inteiros dos dias da semana: " ++ show (somaDias diasSemana)
```

```
Lista de números: [-1,2,-3,4,-5]
Quantidade de negativos: 3
Texto: PPPPPPQQQPPP
Quantidade de letras 'P': 9
Lista de dias da semana: [Segunda,Sabado,Quarta,Sabado,Sexta,Sabado]
Quantidade de sábados: 3
Soma dos valores inteiros dos dias da semana: 33
```

## Capítulo 5

### Quest. 5.01

```haskell
data TipoProduto = Escritorio | Informatica | Livro | Filme | Total deriving (Show, Eq)

data Produto = Produto { valor :: Double, tipo :: TipoProduto } | Nulo deriving (Show, Eq)

instance Semigroup Produto where
    (<>) :: Produto -> Produto -> Produto
    Nulo <> p = p
    p <> Nulo = p
    Produto v1 _ <> Produto v2 _ = Produto (v1 + v2) Total


instance Monoid Produto where
    mempty :: Produto
    mempty = Nulo

main :: IO ()
main = do
    let p1 = Produto 100.0 Informatica
    let p2 = Produto 50.0 Livro
    let p3 = Produto 30.0 Filme
    let nulo = Nulo

    -- Testando a combinação de dois produtos
    putStrLn "Combinação de dois produtos (Informatica e Livro):"
    print (p1 <> p2)

    -- Testando a combinação de produto com Nulo
    putStrLn "\nCombinação de produto com Nulo (Informatica e Nulo):"
    print (p1 <> nulo)

    -- Testando a combinação de três produtos
    putStrLn "\nCombinação de três produtos (Informatica, Livro e Filme):"
    print (p1 <> p2 <> p3)
```

```
Combinação de dois produtos (Informatica e Livro):
Produto {valor = 150.0, tipo = Total}

Combinação de produto com Nulo (Informatica e Nulo):
Produto {valor = 100.0, tipo = Informatica}

Produto {valor = 100.0, tipo = Informatica}
Produto {valor = 100.0, tipo = Informatica}

Combinação de três produtos (Informatica, Livro e Filme):
Produto {valor = 180.0, tipo = Total}
```

## Quest. 5.02

```
data Produto = Produto Double String | Nulo
    deriving (Show)

-- Torna Produto uma instância de Monoid
instance Semigroup Produto where
    Nulo <> p = p
    p <> Nulo = p
    Produto v1 n1 <> Produto v2 n2 = Produto (v1 + v2) (n1 ++ ", " ++ n2)

instance Monoid Produto where
    mempty = Nulo

-- Função para calcular o total geral de uma lista de produtos
totalGeral :: [Produto] -> Double
totalGeral produtos = case mconcat produtos of
    Nulo -> 0.0  -- Se o resultado for Nulo, o total é 0
    Produto v _ -> v  -- Se for um Produto, retorne o valor

-- Função principal para testar o comportamento
main :: IO ()
main = do
    let produtos = [Produto 10.0 "Caneta", Produto 20.0 "Lápis", Produto 15.0 "Caderno", Nulo]

    putStrLn $ "Lista de produtos: " ++ show produtos
    putStrLn $ "Total geral: " ++ show (totalGeral produtos)
```

```
Lista de produtos: [Produto 10.0 "Caneta",Produto 20.0 "L\225pis",Produto 15.0 "Caderno",Nulo]
Total geral: 45.0
```

### Quest. 5.03

```
-- Define o tipo Min
data Min = Min Int deriving (Show, Eq)

-- Torna Min uma instância de Ord para permitir comparação
instance Ord Min where
    (Min x) <= (Min y) = x <= y

-- Torna Min uma instância de Semigroup
instance Semigroup Min where
    (Min x) <> (Min y) = Min (min x y)

-- Torna Min uma instância de Monoid
instance Monoid Min where
    mempty = Min maxBound

-- Função para calcular o mínimo de uma lista de inteiros
calcularMinimo :: [Int] -> Int
calcularMinimo xs = let Min resultado = mconcat (map Min xs) in resultado

-- Função principal para testar o comportamento
main :: IO ()
main = do
    let numeros = [10, 3, 45, 2, 8, 12]
    putStrLn $ "Lista de números: " ++ show numeros
    putStrLn $ "Menor número da lista: " ++ show (calcularMinimo numeros)
```

```
Lista de números: [10,3,45,2,8,12]
Menor número da lista: 2
```

### Quest. 5.04

```haskell
-- Define o tipo Min
data Min = Min Int deriving (Show, Eq)

-- Torna Min uma instância de Ord para permitir comparação
instance Ord Min where
    (Min x) <= (Min y) = x <= y

-- Torna Min uma instância de Semigroup
instance Semigroup Min where
    (Min x) <> (Min y) = Min (min x y)

-- Torna Min uma instância de Monoid
instance Monoid Min where
    mempty = Min maxBound

-- Função para calcular o menor valor de uma lista de Min
minAll :: [Min] -> Min
minAll [] = mempty
minAll xs = foldl mappend mempty xs

-- Função principal para testar o comportamento
main :: IO ()
main = do
    let valores = [Min 10, Min 3, Min 45, Min 2, Min 8, Min 12]
    putStrLn $ "Lista de valores Min: " ++ show valores
    putStrLn $ "Menor valor: " ++ show (minAll valores)
```

```
Lista de valores Min: [Min 10,Min 3,Min 45,Min 2,Min 8,Min 12]
Menor valor: Min 2
```

### Quest. 5.05

```haskell
data Paridade = Par | Impar deriving (Show, Eq)

class ParImpar a where
    decide :: a -> Paridade

instance ParImpar Int where
    decide n
        | even n    = Par
        | otherwise = Impar

instance ParImpar [a] where
    decide xs
        | even (length xs) = Par
        | otherwise        = Impar

instance ParImpar Bool where
    decide False = Par
    decide True  = Impar

-- Classe para calcular a média
class Media a where
    mean :: a -> Double

-- Instância para listas de números
instance Media [Double] where
    mean xs
        | null xs   = 0  -- Caso a lista esteja vazia
        | otherwise = sum xs / fromIntegral (length xs)

-- Exemplo de uso
main :: IO ()
main = do
    -- Testando a função decide
    print $ decide (5 :: Int)           -- Saída: Impar
    print $ decide [1, 2, 3, 4]         -- Saída: Par
    print $ decide False                -- Saída: Par

    -- Testando a função mean
    print $ mean ([1.0, 2.0, 3.0, 4.0] :: [Double])   -- Saída: 2.5
    print $ mean ([] :: [Double])                     -- Saída: 0.0
```

```
Impar
Par
Par
2.5
0.0
```

## Quest. 5.07

```haskell
data Arvore a = Folha a | No (Arvore a) (Arvore a) deriving (Show)

-- Função mapa que aplica uma função a todos os elementos da árvore
mapa :: (a -> b) -> Arvore a -> Arvore b
mapa f (Folha x) = Folha (f x)
mapa f (No esquerda direita) = No (mapa f esquerda) (mapa f direita)

-- Exemplo de uso
main :: IO ()
main = do
    let arvore = No (Folha 1) (No (Folha 2) (Folha 3)) -- Árvore de exemplo
    print arvore                                       -- Saída: No (Folha 1) (No (Folha 2) (Folha 3))

    let dobrada = mapa (*2) arvore                     -- Mapeando a função (*2)
    print dobrada                                      -- Saída: No (Folha 2) (No (Folha 4) (Folha 6))

    let paraString = mapa show arvore                  -- Convertendo elementos para String
    print paraString                                   -- Saída: No (Folha "1") (No (Folha "2") (Folha "3"))
```

```
No (Folha 1) (No (Folha 2) (Folha 3))
No (Folha 2) (No (Folha 4) (Folha 6))
No (Folha "1") (No (Folha "2") (Folha "3"))
```

### Quest. 5.08

```haskell
somar5 :: Int -> Int
somar5 x = x + 5

main :: IO ()
main = do
    let resultado = somar5 10
    putStrLn ("O resultado da soma é: " ++ show resultado)
```

```
O resultado da soma é: 15
```