Listas de Exercícios - Programação Funcional (Haskell)

Lista 1

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

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

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

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

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

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

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

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

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

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

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

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

7. 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)
```

8. 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)
```

9. 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
```

10. 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)
```

11. 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)
```

12. 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)
```

13. Crie uma função que retorna quantas vezes um dígito aparece em um número.

```
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
```

14. Crie uma função que recebe um inteiro positivo e retorna uma String com o número em binário.

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

15. Crie uma função que soma os 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)
```

16. Crie uma função que calcula o elemento da função de Ackermann.

```
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))
```