Paradigmas de Programação – Lista 7

Parte A - Funções de alta ordem

1. Usando apenas as funções das bibliotecas `Prelude` e `Data.Char` e funções de alta ordem, implemente, de forma tácita, as funções listadas abaixo.

(a) `count :: String -> Int`, que recebe uma string e retorne o número de palavras que terminem com letra minúscula.

```
ghci> count "Um exemplO de Contagem de PalavraS"    -- 4
```

(b) `inverses :: [Int] -> [Double]`, que recebe uma lista de inteiros e retorne uma lista com os inversos multiplicativos dos elementos não-nulos. Ex

```
ghci> inverses [0, 1, 0, 2, 0, 4]                   -- [1.0, 0.5, 0.25]
```

(c) `odds :: Int -> Int`, que recebe um inteiro e returne o número de dígitos ímpares deste número.

```
ghci> odds 12345                                    -- 3
```

(d) `palindromes :: String -> Int`, que recebe uma string e retorna o número de palavras que são palíndromos. Ex.:

```
ghci> palindromes "Aia mussum ou mirim"             -- 2
```

(e) `isOctNumber :: Int -> Bool`, que recebe um inteiro `x` e retorna verdadeiro se x representa um número em base octal. Ex.:

```
ghci> isOctNumber 1203577                           -- True
```

2. Usando uma única dobra à esquerda, implemente as funções abaixo. A implementação deve ser feita em uma única linha.

(a) `eval :: [Int] -> Int -> Int`, que recebe uma lista de inteiros `as` e um inteiro `x` e retorna o valor de $p(x)$, onde

$$p(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_N x^N$$

Ex.:

```
ghci> eval [1 -5 6] 2                               -- 0
```

Dica: utilize o algoritmo de Horner.

(b) `k_factorial :: Int -> Int -> Int`, que recebe os inteiros `n` e `k` e retorna o k-fatorial $n!_{(k)}$ de n, onde

$$n!_{(k)} = \begin{cases} n \cdot (n-k)!_{(k)}, & \text{se } n > 0; \\ 1, & \text{se } -k < n \leq 0; \\ 0, & \text{caso contrário} \end{cases}$$

Ex.:

```
ghci> k_factorial 6 2                               -- 48
```

(c) `mean :: [Int] -> Double`, que recebe uma lista de inteiros `xs` e retorna a média aritmética destes números. Ex.:

```
ghci> mean [1..4]                                   -- 2.5
```

(d) `arithmetic_progression_sum :: Int -> Int -> Int -> Int` que recebe o número de termos `n`, o primeiro termo `a` e a razão `q` de uma progressão aritmética e retorne a soma dos primeiros n termos desta progressão. Ex.:

```
ghci> arithmetic_progression_sum 10 1 2             -- 100
```

(e) `exp_approx :: Double -> Double`, que recebe um número `x` e retorna a aproximação da exponencial por série de potências até o 11° termo, isto é,

$$e^x \approx 1 + \frac{x}{1!} + \frac{x^2}{2!} + \ldots + \frac{x^{10}}{10!}$$

Ex.:

```
ghci> exp_approx 2                                    -- 7.388994708994708
```

Parte B - I/O

3. Resolva, no Beecrowd, o exercício indicado em Haskell. Para formatar a saída, use a função `printf` do módulo `Text.Printf`.

(a) `BEE 1007` – Diferença

(b) `BEE 1008` – Salário

(c) `BEE 1011` – Esfera

(d) `BEE 1013` – O Maior

(e) `BEE 1014` – Consumo