Resolução Lista 1 - Curso de Verão de Programação Funcional Pura e Aplicada em Haskell

1. Construa o list comprehension que gere:

a) [0, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50]

```
lista1A = [x * 5 | x <- [0 .. 10]]
```

b) Uma lista de 'a' a 'z' sem as vogais.

```
lista1B = [x | x <- ['a' .. 'z'], x `notElem` ['a', 'e', 'i', 'o', 'u']]
```

c) uma lista de 0 a 50 sem os números 2, 7, 13, 35 e 42.

```
lista1C = [x | x <- [0 .. 50], x `notElem` [2, 7, 13, 35, 42]]
```

d) [500.0, 250.0, 125.0, 62.5, 31.25, 15.625, 7.8125, 3.90625, 1.953125, 0.9765625]

```
lista1D = [500 / (2 ** x) | x <- [0 .. 9]]
```

e) uma lista com todas as coordenadas de um tabuleiro de damas 8x8 ([('a',1), ('a',2), ('a',3) ... ('h',7), ('h',8)]).

```
lista1E = [(x, y) | x <- ['a' .. 'h'], y <- [1 .. 8]]
```

2. Crie as funções da forma como solicitado:

a)Crie um função recursiva que verifique se um elemento esta presente em uma lista, deixe explicita a assinatura da função.

```
ehPresente :: Eq a => a -> [a] -> Bool
ehPresente _ [] = False
ehPresente elm (x : xs) = (elm == x) || ehPresente elm xs
```

b) Crie um função que receba 3 valores numéricos, de qualquer tipo, e considerando que cada valor é a medida de um lado de um triângulo, retorne um valor do tipo Bool informando se estas são medidas validas para formar um triângulo.

```
ehTriangulo :: Num a => Ord a => a -> a -> a -> Bool
ehTriangulo x y z = abs (y - z) < x && x < (y + z)
```

c) Crie uma função recursiva que tenha dois parâmetros númericos, de qualquer tipo, e calcule a potenciação, o primeiro parâmetro sendo a base e o segundo o expoente.

```
potencia :: Int -> Int -> Int
potencia b e
  | e == 0 = 1
  | e == 1 = b
  | otherwise = b * potencia b (e - 1)
```

3. Siga as intruções a baixo na sequência:

a) Crie um tipo Cor que possua um value constructor de mesmo nome que carrege 3 valores do tipo Int.

```
data Cor1 = Cor1 Int Int Int
```

b) Cada um dos valores do tipo Int são relativos aos valores RGB (red, green, blue), pra cada um desses valores crie (manualmente) uma função de projeçãoo nome das funções devem ser red, green e blue, respectvamente para cada um dos valores do tipo Int.

```
red1 :: Cor1 -> Int
red1 (Cor1 r _ _) = r

green1 :: Cor1 -> Int
green1 (Cor1 _ g _) = g

blue1 :: Cor1 -> Int
blue1 (Cor1 _ _ b) = b
```

c) Ao invés de criar as funções de projeção manualmente, crie o tipo Cor utilizando record syntax.

```
data Cor = Cor {red :: Int, green :: Int, blue :: Int} deriving (Show)
```

d) Crie uma função com o nome somaCor, que combine dois valores do tipo Cor, de forma que o resultado deve ser um novo valor, também do tipo Cor, onde os valores relativos ao RGB (red, green e blue), são relativos a soma dos valores RGB dos parâmetros da função, respectivamente. Caso algum valor relativo relativo ao valores RGB do resultado seja maior que 255, deve-se colocar 255 no lugar.

```
somaCanal :: Int -> Int -> Int
somaCanal x y
  | x > 255 || y > 255 = 255
  | otherwise = let s = x + y in if s > 255 then 255 else s

somaCor :: Cor -> Cor -> Cor
somaCor c1 c2 = Cor {red = red c1 `somaCanal` red c2, green = green c1 `somaCanal` green c2, blue = blu
```

e) Crie um operador <+> que faça a mesma coisa que a função somaCor, utilizi-se do conceito de currying para definir o operador. (Os símbolos de maior e menor fazem parte do operador)

```
(<+>) :: Cor -> Cor -> Cor
(<+>) = somaCor
```

f) Crie uma instância de Monoid para o tipo Cor, onde a operação bínaria seja o operador <+> e o elemento nêutro seja um valor do tipo Cor que contem os valores RGB como 0.

```
instance Semigroup Cor where
  (<>) = (<+>)

instance Monoid Cor where
  mempty = Cor {red = 0, green = 0, blue = 0}
```

g) A instância de Monoid criada é valida considerando as leis dos Monoids?

Sim, é uma instância válida pois é implementat ambas as funções `mappend` e `mempty`.

4. Siga as intruções abaixo em sequência:

a) Crie um tipo Cofre que possua uma type variable, e um value constructor de mesmo nome que o tipo, que carregue um valor que seja um lista do tipo da type variable.

```
newtype Cofre a = Cofre [a] deriving (Show)
```

b) Crie uma instância do typeclass Functor para esse tipo.

```
instance Functor Cofre where
  fmap f (Cofre xs) = Cofre (fmap f xs)
```

c) Crie uma instância do typeclass Applicative para esse tipo.

```
instance Applicative Cofre where
  pure x = Cofre [x]
  (Cofre fs) <*> (Cofre xs) = Cofre [f x | x <- xs, f <- fs]
```

5. Siga as intruções a baixo na sequência:

a) Crie o tipo Automovel que possua dois value constructors, um para representar o carro, e o outro para representar a moto. (os value constructors não carregam nenhum valor)

```
data Automovel = Carro | Moto
```

b) Crie um tipo Veiculo que possua um value constructor de mesmo nome que carregue 2 valores, o primeiro de nome automovel do tipo Automovel, o segundo de nome placa do tipo String. (Utilize record syntax)

```
data Veiculo = Veiculo {automovel :: Automovel, placa :: String}
```

c) Crie um typeclass EhCarro, para tipos de kind *, que possua uma função com nome ehCarro que receba um valor do tipo do type parameter do typeclass, e retorne um valor do tipo Bool informando se é um carro.

```
class EhCarro c where
  ehCarro :: c -> Bool
```

d) Crie uma instância de EhCarro para o tipo Veiculo.

```
instance EhCarro Veiculo where
  ehCarro (Veiculo Carro _) = True
  ehCarro (Veiculo Moto _) = False
```

e) Crie uma instância de EhCarro para o tipo Int (Nenhum número é um carro).

```
instance EhCarro Int where
  ehCarro _ = False
```

6. Seguindo os aximos de Peano:

a) Zero é um número natural, e todo sucessor de um número natural também é um número natural. Crie o tipo Natural, que possua dois value contructors, um para representar o valor zero, e o outro para representar a sucessão de um valor do tipo Natural. (Dica: este é um tipo recursivo)

```
data Natural = Zero | Proximo Natural deriving (Show)
```

b) Com o conceito de sucessão contido na definição do tipo Natual, podemos representar o valor 1 como o sucessor de zero, o valor 2 como o sucessor do sucessor de zero, e assim por diante. Crie uma função somaNatural, que receba dois parâmetros do tipo Natural que você criou e retorne um valor do tipo Natural que seja a soma dos dois parâmetros.

```
somaNatural :: Natural -> Natural -> Natural
somaNatural Zero x = x
```

```
somaNatural (Proximo x) y = Proximo (somaNatural x y)
```

c) Crie uma função que converta de Natural para Int.

```
paraInt :: Natural -> Int
paraInt Zero = 0
paraInt (Proximo x) = 1 + paraInt x
```

7. Crie uma função que receba 3 parâmetros, sendo eles duas funções (referenciadas aqui no enunciado como f e g) e um valor (referenciado aqui no enunciado como x), e retorne uma tupla de duas posições contendo na primeira posição o resultado de f composta em g aplicado em x e na segunda o resultado de g composta em f aplicado em x. Construa a função da forma mais genérica que for possível de criar uma função com essas caracteristicas e deixe explicita a assinatura da função. Faça uma breve explicação sobre a forma como vc pensou para criar essa função.

```
compFunc :: (a -> a) -> (a -> a) -> a -> (a, a)
compFunc f g x = (f . g $ x, g . f $ x)
```

8. Crie um tipo JoKenPo que possua três value constructors, representando as jogadas predra, papel e tesoura. Crie um tipo Resultado que possua três value constructors, representado os resultados da vitória do jogador 1, a vitória do jogador 2 e o empate. Crie uma função jogar, que recebe duas jogadas, do tipo JoKenPo, e retorne o resultado, do tipo Resultado. Considere o primeiro parâmetro como jogador 1 e o segundo como jogador 2.

```
data JoKePo = Pedra | Papel | Tesoura deriving (Show)

data Result = Jogador1 | Jogador2 | Empate deriving (Show)

jogar :: JoKePo -> JoKePo -> Result
jogar Pedra Tesoura = Jogador1
jogar Pedra Papel = Jogador2
jogar Pedra Pedra = Empate
jogar Tesoura Papel = Jogador1
jogar Tesoura Pedra = Jogador2
jogar Tesoura Tesoura = Empate
jogar Papel Pedra = Jogador1
jogar Papel Tesoura = Jogador2
jogar Papel Papel = Empate
```