Listas, exercícios e provas EE1

Exercícios

Defina as seguintes funções

- fat :: Int -> Int que calcula o fatorial de um inteiro dado como argumento
- all4Equal :: Int -> Int -> Int -> Int -> Bool que compara se quatro valores inteiros são iguais
- equalCount :: Int -> Int -> Int -> Int que retorna quantos argumentos são iguais

13

```
fat :: Int -> Int
fat 1 = 1
fat n = n * fat (n-1)
by: Lucas
```

```
all4Equal :: Int -> Int -> Int -> Int -> Bool
all4Equal a b c d
| a == b && a == c && a == d = True
| otherwise = False
by: Lucas
```

```haskell
equalCount :: Int -> Int -> Int -> Int
equalCount a b c
| a == b && a == c = 3
| a == b || a == c || b == c = 2
| otherwise = 1
by: Lucas
```

Exercício

Exercício

Defina uma função que, dado um valor inteiro *s* e um número de semanas *n*, retorna quantas semanas de 0 a *n* tiveram vendas iguais a *s*.

16

```haskell
vendas 1 = 3
vendas 2 = 5
vendas 3 = 3
vendas 4 = 1

totalvendas :: Int -> Int -> Int
totalvendas s n
| s == vendas n && n == 1 = 1
| s /= vendas n && n == 1 = 0
```

```
| s == vendas n = 1 + totalvendas (s) (n-1)
| s /= vendas n = 0 + totalvendas (s) (n-1)

feito por Lucas
```

```
vendas 0 = 0
vendas 1 = 1
vendas 2 = 1
vendas 3 = 1
vendas 4 = 2
vendas 5 = 0

vendasDay :: Int -> Int -> Int
vendasDay m n
    |(m == vendas n) = vendas n
    |otherwise = 0


totalVendas :: Int -> Int -> Int
totalVendas s n
    | n == 0 = vendasDay s n
    | otherwise = (vendasDay s n) + (totalVendas s (n-1))
```

Exercícios

- Defina a funçãoo addEspacos que produz um string com uma quantidade n de espaços.
  addEspacos :: Int -> String
- Defina a função paraDireita utilizando a definição de addEspacos para adiciconar uma quantidade n de espaços à esquerda de um dado String, movendo o mesmo para a direita.
  paraDireita :: Int -> String -> String


```
addEspacos :: Int -> String
addEspacos x
| x == 0 = ""
| x /= 0 = " " ++  addEspacos (x-1)
by: Lucas
```

```
paraDireita :: Int -> String -> String
paraDireita e s = addEspacos (e) ++ s
by: Lucas
```

Exercícios

Escreva uma função para retornar, em forma de tabela, todas as vendas da semana 0 até a semana n, incluindo o total e a média de vendas no período. Usem as funções definidas previamente e defina novas funções que achar necessário.

Semana	Venda
0	12
1	14
2	15
Total	41
Média	13.6667

Exercícios

- Defina a função menorMaior que recebe três inteiros e retorna uma tupla com o menor e o maior deles, respectivamente.
- Defina a função ordenaTripla que recebe uma tripla de inteiros e ordena a mesma.

```
maxInt :: Int->Int->Int
maxInt a b
    | a<b = b
    |otherwise = a

minInt :: Int->Int->Int
minInt a b
    |a<b = a
    |otherwise = b

menorMaior :: Int -> Int -> Int -> (Int,Int)
menorMaior a b c = (maxInt (maxInt a b) c ,minInt (minInt
a b) c)
```

```
ordenaTripla :: (Int,Int,Int)->(Int,Int,Int)
ordenaTripla (x,y,z)
    | z>=y && y>=x = (x,y,z)
    | x>y = ordenaTripla(y,x,z)
    | y>z = ordenaTripla(x,z,y)
```

Exercícios

Uma linha pode ser representada da seguinte forma

```
type Ponto = (Float, Float)
type Reta  = (Ponto, Ponto)
```

- Defina funções que
  - retornem
    - a primeira coordenada de um ponto
    - a segunda coordenada de um ponto
  - indique se uma reta é vertical ou não ($x_1 = x_2$)


```
type Ponto = (Float, Float)
type Reta = (Ponto, Ponto)

exibiX :: Ponto -> Float
exibiX (x,_) = x

exibiY :: Ponto -> Float
exibiY (_,y) = y
```

```
retaVertical :: Reta -> Bool
retaVertical ((a,b),(c,d))
 | a == c = True
 | a /= c = False
```

Exercícios

Se uma reta é dada por

$(y - y_1)/(x - x_1) = (y_2 - y_1)/(x_2 - x_1)$,

defina uma função

pontoY :: Float –> Reta –> Float

que, dada uma coordenada x e uma reta, retorne a coordenada y, tal que o ponto (x, y) faça parte da reta.


```
whereIsX :: Float-> Reta -> Float
whereIsX a ((x,y),(v,w)) = ((a-y)/((w-y)/(v-x))) + x

whereIsY :: Float -> Reta -> Float
whereIsY a ((x,y),(v,w)) = ((a+x)*((w-y)/(v-x))) - y
```

```
*Main> :t [[2,3]]
[[2,3]] :: Num a => [[a]]
*Main>
```

Exercícios

- Quantos itens existem nas seguintes listas?
  - [2,3]
  - [[2,3]]
- Qual o tipo de [[2,3]] ?
- Qual o resultado da avaliação de
  - [2,4..9]
  - [2..2]
  - [2,7..4]
  - [10,9..1]
  - [10..1]
  - [2,9,8..1]


```
:t [[2,3]]

countList :: [a] -> Int --Funciona para qualquer tipo,
inclusive uma string
countList [] = 0
countList (x:xs) = 1 + countList(xs)
```

Exercícios

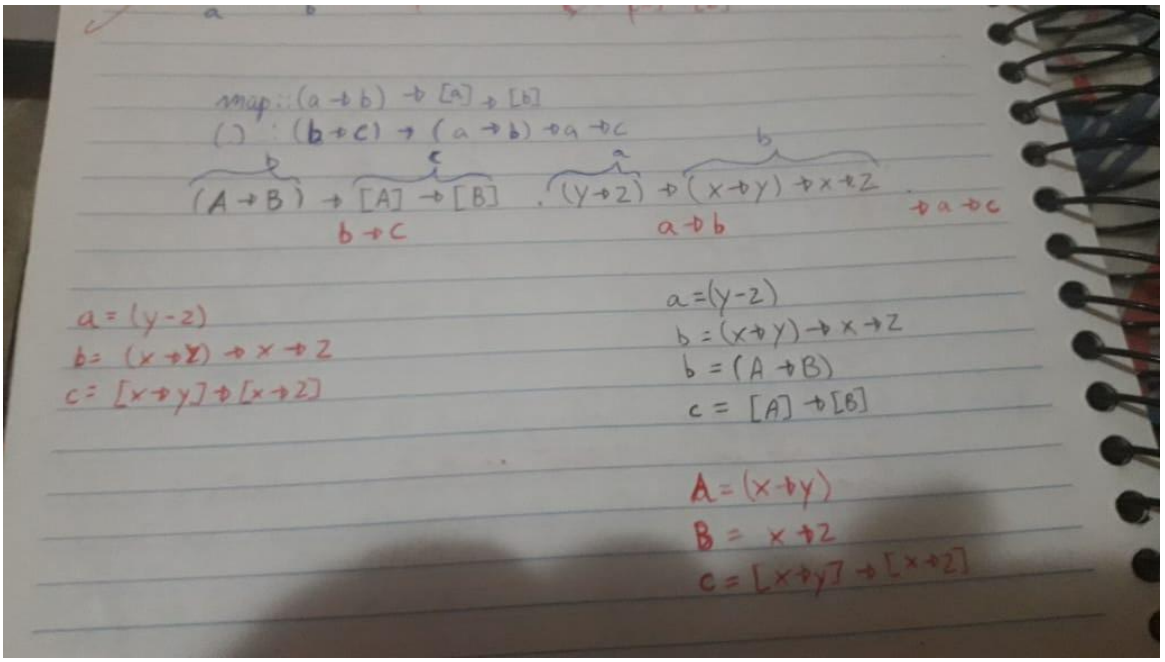
Defina funções sobre listas para

- dobrar os elementos de uma lista: double :: [Int] -> [Int]
- determinar se um valor faz parte de uma lista:
  member :: [Int] -> Int -> Bool
- filtrar apenas os números de uma lista:
  digits :: String -> String
- somar uma lista de pares: sumPairs :: [(Int,Int)]->[Int]


```
subListas :: [a] ->[[a]]
subListas [] = [[]]
subListas (x:xs) = [x:ys | ys <- subListas xs] ++
subListas xs
```

LISTA 01 - HASKELL

1. Determine se a expressão map.(.) está correta. Em caso negativo, justifique. Em caso positivo,indique o tipo resultante e justifique (derive) como você o determinou.



2. Defina uma função sublistas :: [a] -> [[a]] que retorna todas as sublistas de uma lista dada como argumento.

```
sublistas :: [a] -> [[a]]
sublistas [] = [[]]
sublistas (x:xs) = [ x:ys | ys <- sublistas (xs) ] ++
sublistas xs
```

3. Considere uma função polinomial de grau 2 (f(x) = ax2 + bx + c), onde a, b e c são os coeficientes do polinômio.

(a) Defina a função poli :: Integer -> Integer -> Integer -> Integer -> Integer que recebe como argumentos os coeficientes de uma função polinomial de grau 2 e devolve uma função de inteiro para inteiro (um polinômio).

```
poli :: Integer -> Integer -> Integer -> Integer ->
Integer
poli x y z = \a -> (a*a)*x + a*y + z
```

(b) Defina a função listaPoli :: [(
Integer,Integer,Integer)] -> [Integer->Integer] que aguarda uma lista de triplas de inteiros (coeficientes de um polinômio de segundo grau) e devolve uma lista de funções de inteiro para inteiro (polinômios) .

4. Dada uma matriz representada por uma lista de listas, defina fun ̧c ̃oes para:

(a) indicar se a mesma ́e uma matriz (se todas as linhas tˆem o mesmo tamanho)

```
countList :: [a] -> Int
countList [] = 0
countList (x:xs) = 1 + countList(xs)

isMatrix :: [[a]] -> Bool
isMatrix [[]] = True
isMatrix (x:y:xs) = (countList(x) == countList(y)) &&
isMatrix (y:xs)
isMatrix (x:xs) = True
```

(b) permutar a posição de duas linhas x e y, assumindo que x < y. Dica: pode-se utilizar as funções
init, take, drop e !! .c

```
permutar :: Int -> Int -> [a] -> [a]
permutar a b c
   | a == b = c
   | a<b = (take (a-1) c) ++ (take (1) (drop (b-1)
c)) ++ (take (b-1-a) (drop (a) c))++ (take (1) (drop
```

```
(a-1) c)) ++ drop b c
   | a>b = permutar b a c
--Peixa
```

```
init [ 1 , 2 , 3 ] = [ 1 , 2 ]
take 2 [ 1 , 2 , 3 ] = [ 1 , 2 ]
drop 2 [ 1 , 2 , 3 ] = [ 3 ]
[ 1 , 2 , 3 , 4 , 5 ] ! ! 2 = 3
map
zip
length
```

LISTA 02 HASKELL

1-

a)

```
f :: [Int] -> [Int]
f [] = []
f (x:[]) = []
f (x:y:xs)
    |x == y = x : (f (y:xs))
    | x /= y = f(y:xs)
```

b)

```
f :: [Int] -> [Int] --Com compressão de Lista
f [] = []
f (x:xs) = [x | (x,y) <- zip (x:xs) xs, x==y]
```

2-

```
g::[Int] -> Bool
g [] = True
g a = foldr (==) True (map (not . odd) (filter (<=100) (filter(>=0)
a)))
```

3-

```haskell
type Fabricante = String
type Potencia = Float
data Lampada = Compacta Fabricante Potencia
                | Incandescente Fabricante Potencia
instance Show Lampada where
    show (Compacta f pitu ) = "Compacta "++ f ++" " ++
show(pitu)
    show (Incandescente f co) = "Incandescente " ++ f ++
" " ++ show(co)
instance Eq Lampada where
    (Compacta f1 p1) == (Compacta f2 p2) = f1==f2 && p1
==p2
    (Incandescente f1 p1) == (Incandescente f2 p2) =
f1==f2 && p1 ==p2
    _==_ = False
```

(1,5) 2. Implemente a função `filtrarEInserir :: [[Int]] -> Int -> ([[Int]], Int)` que retorna uma tupla. O primeiro elemento da tupla é constituído de listas de inteiros tais que a soma dos números ímpares é maior que a soma dos números pares. O segundo elemento consiste no produto entre o segundo argumento da função `filtrarEInserir` e a multiplicação da maior soma obtida das listas retornadas. Utilize obrigatoriamente `filter`.

> filtrarEInserir [[2,3,4,5,6], [1, 2, 3], [9]] 5
([[1,2,3], [9]], 45)

> filtrarEInserir [[2,3,4,5], []] 7
([[2,3,4,5]],56)

> filtrarEInserir [] 5
([],0)

```haskell
-- Questão 2 2017.2
countList1::[Int] -> Int
countList1 [] = 0
countList1 (x:xs)
    | mod x 2 == 1 = x + countList1 xs
    | otherwise = countList1 xs

countList2:: [Int] -> Int
countList2 [] = 0
countList2 (x:xs)
    | mod x 2 == 0 = x + countList2 xs
```

```
    | otherwise = countList2 xs

filtBool:: [Int] -> Bool
filtBool a
    |(countList1 a) > (countList2 a) = True
    |otherwise = False

maxInList:: [[Int]] -> Int
maxInList [[]] = 0
maxInList (x:[]) = countList1 x + countList2 x
maxInList (x:xs) = max (countList1 x + countList2 x)
(maxInList xs)

filtrarEInserir:: [[Int]] -> Int -> ([[Int]],Int)
filtrarEInserir [[]] a = ([[]],0)
filtrarEInserir (x:xs) a = (filter(\y -> filtBool y)
(x:xs) ,a*(maxInList(filter(\y -> filtBool y) (x:xs))))
```

(1,0) 3. Defina a função `altMap :: (a -> b) -> (a -> b) -> [a] -> [b]` que, de forma alternada, aplica as duas funções dadas como argumentos aos elementos sucessivos na lista, respeitando a ordem deles.

> altMap (+10) (+100) [0, 1, 2, 3, 4]
[10, 101, 12, 103, 14]

```
--Questão3

altMap :: (a->b) -> (a->b) -> [a] -> [b]
altMap f1 f2 [] = []
altMap f1 f2 (p:q:xs) = (f1 (p)): (f2 (q)): altMap f1 f2
xs
altMap f1 f2 (q:xs) = (f1 q): altMap f1 f2 xs
```

```
--Questão 4
--a)
poli:: Int -> Int -> Int -> Int -> Int
poli a b c = (\x -> (a*x*x) + (b*x) + c) --Recebe a b c,
```

```
aguarda receber x, e realiza a resposta
--b) Não funciona sozinha
listaPoli :: [(Int,Int,Int)] -> [Int -> Int]
listaPoli lp = [poli a b c | (a,b,c) <- lp]
--c)
appListaPoli:: [Int->Int]->[Int]->[Int]
appListaPoli lPoli lInt = [f x | (f,x) <-zip lPoli lInt]

--Questão 5
data Mobile = Pendente Int | Barra Mobile Mobile
batato = (Barra (Pendente 5) (Pendente 5))
peso :: Mobile -> Int
peso (Pendente n) = n
peso (Barra m p) = peso m + peso p
--b)
balanceado:: Mobile -> Bool
balanceado (Pendente _) = True --Se só tiver um, não
importa o que tiver, estará balanceado
--balanceado (Barra a b) = (peso a == peso b) --Isso
checaria se o total está balanceado
balanceado (Barra a b) = (peso a == peso b) && balanceado
a && balanceado b
-- Isso checaria para cada barra se está balanceado
```

Prova 2019.1

1. A função vendas :: Int -> Int retorna a quantidade semanal de vendas de uma loja. As semanas são numeradas em uma sequência 0, 1, 2, ... Implemente a função zeroVendas que se comporta da seguinte maneira: dado um número n que assumimos como não negativo, retorna o número de semanas na faixa 0, 1, ..., n em que a quantidade de itens

vendidos foi 0 (zero). Implemente definições de zeroVendas.

a) Usando compressão de listas e a função length

```
vendas :: Int -> Int
vendas 0 = 0
vendas 1 = 1
vendas 2 = 2
vendas 3 = 1
vendas 4 = 0
vendas _ = 0

genArray:: Int -> [Int]
genArray 0 = [y | y<-[vendas 0]]
genArray a = [y | y<-[vendas a]] ++ genArray (a-1)

zeroVendas :: Int -> Int
zeroVendas a = length(filter( == 0) (genArray a) )
--Texero

--outra forma : zeroVendas n = length ([x | x <- [0..n], 0 == vendas x]) --by Lion Kun

-- outra forma :
lista 0  = [vendas 0]
lista n  = vendas n : lista (n-1)
zeroVendas n = length([x | x <- lista n , x == 0 ])
```

b) Usando qualquer função padrão de Haskell, mas sem definir função recursiva, não usando foldr ou foldl, e sem usar compressão de lista.

```
zeroVendas :: Int -> Int
```

```
zeroVendas n = length(filter(==0) (map vendas [0..n]))
--by Lion Kun
```

c) Usando foldr, a lista [0..n] e sem uso de qualquer outra função recursiva. Pode ser necessária uma função auxiliar.

```
zeroVendas :: Int -> Int
zeroVendas n = foldr (+) 0 (map (+1)(filter( ==0 ) (map
vendas [0..n]) )) --Texero
```

2.

a) Defina um tipo algébrico (e tipos auxiliares que você achar necessário) para representar um bilhete de postagem. Um bilhete pode ser um destes:

- Um bilhete de trem de uma cidade para uma cidade, pode ser de primeira classe ou segunda classe
- Um bilhete de ônibus de uma cidade para uma cidade
- Um bilhete aéreo de uma cidade para uma cidade, que pode ser de classe econômica ou executiva

Cidades podem ser representadas por String

```
type Cidade = String
type Classe = String
data Bilhete = Trem Cidade Cidade Classe
             | Onibus Cidade Cidade
             | Aviao Cidade Cidade Classe
```

b) Para qualquer bilhete, a primeira cidade é chamada de cidade de origem; a segunda, de destino. Nós representamos uma viagem por uma lista de bilhetes.

Uma viagem é válida se para qualquer bilhetes consecutivos

na lista, a cidade de destino do primeiro bilhete é a de partida do segundo bilhete. Defina a função

```
valida :: [Bilhete] -> Bool
```

que determina se uma viagem é válida. Assuma que a lista não é vazia.

```haskell
partida :: Bilhete -> String
partida (Trem a b c) = a
partida (Onibus a b) = a
partida (Aviao a b c) = a

chegada :: Bilhete -> String
chegada (Trem a b c) = b
chegada (Onibus a b) = b
chegada (Aviao a b c) = b

valida :: [Bilhete] -> Bool
valida [] = True
valida (x:y:xs) = (chegada x) == (partida y) && valida (y:xs)
valida (x:[]) = True
```

```haskell
getOrigin:: [Bilhete] -> String
getOrigin ((Trem o _ _):x)  = o
getOrigin ((Aviao o _ _):x)  = o
getOrigin ((Onibus o _):x)  = o

valida::[Bilhete] -> Bool
valida [] = True
valida (x:[]) = True
valida((Trem _ d _):xs) = d == getOrigin xs && valida (xs)
valida((Aviao _ d _):xs) = d == getOrigin xs && valida (xs)
valida((Onibus _ d ):xs) = d == getOrigin xs && valida (xs)
```

3. Dado o tipo algébrico Nat

```
data Nat = Zero | Succ Nat deriving (Eq.Show)
```

Defina:

(a) Uma função que converte números inteiros em números naturais.

(b) Uma função que converte números naturais em números inteiros.

(c) Defina a função soma que soma dois números naturais.

(d) Defina a função mult que multiplica dois números naturais.

```haskell
-- 3
data Nat = Zero | Succ Nat deriving (Show)

-- a
intToNat :: Int -> Nat
intToNat 0 = Zero
intToNat n = Succ (intToNat (n - 1))

-- b
natToInt :: Nat -> Int
natToInt Zero = 0
natToInt (Succ x) = 1 + natToInt x

-- c
soma :: Nat -> Nat -> Nat
soma Zero Zero = Zero
soma Zero (Succ c) = (Succ c)
soma (Succ c) Zero = (Succ c)
soma (Succ c) (Succ d) = intToNat (x + y)
    where
        x = (natToInt (Succ c))
        y = (natToInt (Succ d))

-- d
mult :: Nat -> Nat -> Nat
mult Zero Zero = Zero
mult Zero (Succ c) = Zero
mult (Succ c) Zero = Zero
mult (Succ c) (Succ d) = intToNat (x * y)
    where
        x = (natToInt (Succ c))
        y = (natToInt (Succ d))
```

2019.2

(2) um arquivo no formato pdf também com todas as respostas. Escrever o nome na primeira linha dos arquivos. O assunto do email deve estar no formato: Nome completo_1ee_plc_2019_2.

(1,0) 1. Defina operadores de seção s1 e s2 de maneira que

```
map s1 . filter s2
```

tenha o mesmo efeito que

```
filter (>5) . map (+2)
```

2. **Mergesort**

(1,0) (a) Defina a função recursiva merge :: **Ord** a =>[a] -> [a] -> [a] que une duas listas ordenadas e resulta em uma única lista ordenada.

(1,5) (b) Usando merge, defina a função msort :: **Ord** a =>[a] -> [a] que implementa merge sort, no qual uma lista vazia ou lista com um elemento está ordenada e qualquer outra lista é ordenada por mesclar as duas listas que resultam de ordenar as duas metades da lista separadamente.

(1,5) (c) Dica: defina metade :: [a] -> ([a],[a]) que divide uma lista em duas metades com tamanhos que diferem no máximo de 1.

3. A partir dos tipos

```
type Texto = String
type Id = String
type DataHoraPub = Int
```

podemos descrever *posts* e *thread* em uma rede social com os tipos

```
data Post = Post (Id, DataHoraPub) Texto deriving (Show, Eq)
data Thread = Nil | T Post (Thread)
```

(1,0) (a) Estabeleça que Thread é instância da classe **Show** de modo que a *thread*

T (Post ("Joao", 1) "asdf") (T (Post ("Marco", 2) "qwer") Nil) é exibida da seguinte forma:

(Joao 1 asdf)(Marco 2 qwer)

Implemente a função **show** para que a exibição se dê como solicitado.

(1,0) (b) Defina a função inserirPost que, dado um *post* e uma *thread*, devolve uma nova *thread* com o novo *post*.

(1,0) (c) Defina a função threadToList :: Thread -> [Post] que transforma uma *thread* em uma lista de *posts*.

(1,0) (d) Defina a função listToThread :: [Post] -> Thread que transforma uma lista de *posts* em um *thread*.

(1,0) (e) Defina a função removerPost :: (Id, DataHoraPub) -> Thread -> Thread que remove um *post* identificado pelo par (Id, DataHoraPub) de uma *thread*. Utilize a função **filter** na implementação.

1 - Defina operadores de seção s1 e s2 de maneira que:
map s1 . filter s2
tenha o mesmo efeito que
filte (>5) . map + 2
2 - Mergesort

(a) Defina a função recursiva merge :: Ord a =>[a]->[a]->[a] que une duas listas ordenadas e resulta em uma única lista ordenada

```
merge:: [Int] -> [Int] -> [Int]
merge (x:xs) [] = (x:xs)
merge [] (x:xs) = (x:xs)
merge (x:xs) (y:ys)
  |x<=y = x : merge xs (y:ys)
  |y<x = y : merge ys (x:xs)
```

(b) Usando merge, defina a função msort :: Ord a => [a] -> [a] que implementa merge sort, no qual uma lista vazia ou lista com um elemento está ordenada e qualquer outra lista é ordenada por mesclar as duas listas que resultam de ordenar as duas metades da lista separadamente.

```
msort:: [Int] -> [Int]
msort [] = []
msort x
  | length x == 1 = x
  | otherwise = merge(msort(fst (half x))) (msort(snd (half x)))
```

(c) Dica: defina metade ::[a] ->([a],[a]) que divide uma lista em duas metades com tamanhos que diferem no máximo 1.

```
half:: [Int] -> ([Int],[Int])
half a = (take (div (length a) 2) a, drop (div (length a) 2) a)
```

3. A partir dos tipos:
type Texto = String
type Id = String
type DataHoraPub = Int
podemos descrever posts e thread em uma rede social com os tipos
data Post = Post (Id,DataHoraPub) Texto deriving (Show,Eq)
data Thread = Nil | T Post (Thread)

```
type Texto = String
type Id = String
type DataHoraPub = Int
data Post = Post (Id,DataHoraPub) Texto deriving (Show,Eq)
data Thread = Nil | T Post (Thread)
batato = Post ("batato",2) "penes" --Variaveis para ajudar
cenoro = Post ("cenoro",3) "clitoros"
bataT = (T batato (T cenoro Nil))
```

(a) Estabeleça que Thread é instância da classe Show de modo que a thread T(Post (“Joao”,1)”asdf”) (T(Post(“Marco”,2)”qwer”)Nil) é exibida da seguinte forma:
(Joao 1 asdf)(Marco 2 qwer)
Implemente a função show para que a exibição se dê como solicitado.

```
instance Show (Thread)
  where
     show (Nil) = ""
     show (T (Post(a,b) c) nextT) = "("++a++" "++ show b ++"
"++c++")"++show nextT
```

(b) Defina a função inserirPost que dado um post e uma thread, devolve uma nova thread com o novo post.

```
inserirPost:: Post -> Thread -> Thread
inserirPost a b = T a b
```

(c) Defina a função threadToList :: Thread->[Post] que

transforma uma thread em uma lista de posts.

```
threadToList :: Thread ->[Post]
threadToList Nil = []
threadToList (T a (aux)) = a : threadToList aux
```

(d) Defina a função ListToThread::[Post] -> Thread que transforma uma lista de posts em um thread.

```
listToThread:: [Post] -> Thread
listToThread [] = Nil
listToThread (x:xs) = T x (listToThread xs)
```

(e) Defina a função removerPost:: (Id,DataHoraPost) -> Thread -> Thread que remove um post identificado pelo par (Id,DataHoraPost) de uma thread Utilize a função filter na implementação

```
getInfo:: Post -> (Id,DataHoraPub)
getInfo (Post a t) = a

removerPost:: (Id,DataHoraPub)->Thread->Thread
removerPost a Nil = Nil
removerPost a (T e d)
  |a == (getInfo e) = d
  |otherwise = T e (removerPost a d)
```