

Exame 2022-02-02

[Voltar](#)

1. Apresente uma definição recursiva da função (pré-definida) `replicate :: Int -> a -> [a]` que dado um inteiro `n` e um elemento `x` constrói uma lista com `n` elementos, todos iguais a `x`. Por exemplo, `replicate 3 10` corresponde a `[10,10,10]`.

```
replicate :: Int -> a -> [a]
replicate 0 _ = []
replicate n x
  | n < 0 = []
  | otherwise = x : replicate (n - 1) x
```

2. Apresente uma definição recursiva da função `intersect :: Eq a => [a] -> [a] -> [a]` que retorna a lista resultante de remover da primeira lista os elementos que não pertencem à segunda. Por exemplo, `intersect [1,1,2,3,4] [1,3,5]` corresponde a `[1,1,3]`.

```
intersect :: Eq a => [a] -> [a] -> [a]
intersect [] _ = []
intersect (h:t) l
  | h `elem` l = h : intersect t l
  | otherwise = intersect t l
```

3. Recorde as declarações das *leaf trees* e *full trees*.

```
data LTree a = Tip a | Fork (LTree a) (LTree a)
data FTree a b = Leaf a | No b (FTree a b) (FTree a b)
```

Defina a função `conv :: LTree Int -> FTree Int Int` que recebe uma `LTree Int` e gera uma árvore `FTree Int Int` com a mesma forma, que

preserva o valor das folhas e coloca em cada nó a soma de todas as folhas da árvore com raiz nesse nó.

```
conv :: LTree Int -> FTree Int Int
conv = snd . convAux

convAux :: LTree Int -> (Int, FTree Int Int)
convAux (Tip x) = (x, Leaf x)
convAux (Fork l r) = (s, No s ll rr)
  where (sl, ll) = convAux l
        (sr, rr) = convAux r
        s = sl + sr
```

4. Considere o seguinte tipo `type Mat a = [[a]]` para representar matrizes. Defina a função `triSup :: Num a => Mat a -> Bool` que testa se uma matriz quadrada é triangular superior (i.e., todos os elementos abaixo da diagonal são nulos). Esta função deve devolver `True` para a matriz `[[1,2,3], [0,4,5], [0,0,6]]`.

```
triSup :: (Eq a, Num a) => Mat a -> Bool
triSup = all (all (== 0) . uncurry take) . zip [0..]
```

5. Considere o seguinte tipo de dados para representar subconjuntos de números reais.

```
data SReais = AA Double Double | FF Double Double
           | AF Double Double | FA Double Double
           | Uniao SReais SReais
```

`(AA x y)` representa o intervalo aberto `]x, y[`, `(FF x y)` representa o intervalo fechado `[x, y]`, `(AF x y)` representa `]x, y]`, `(FA x y)` representa `[x, y[` e `(Uniao a b)` a união de conjuntos.

a) Defina a `SReais` como instância da classe `Show`, de forma a que, por exemplo, a apresentação do termo `Uniao (Uniao (AA 4.2 5.5) (AF 3.1 7.0)) (FF (-12.3) 30.0)` seja `([4.2,5.5[U]3.1,7.0]) U [-12.3,30.0]`

```
instance Show SReais where
  show (AA a b) = "]" ++ show a ++ "," ++ show b ++ "["
  show (FF a b) = "[" ++ show a ++ "," ++ show b ++ "]"
  show (AF a b) = "]" ++ show a ++ "," ++ show b ++ "]"
  show (FA a b) = "[" ++ show a ++ "," ++ show b ++ "["
  show (Uniao a b) = "(" ++ show a ++ " U " ++ show b ++ ")"
```

b) Defina a função `tira :: Double -> SReais -> SReais` que retira um elemento de um conjunto.

```
tira :: Double -> SReais -> SReais
tira x (AA a b)
  | x > a && x < b = Uniao (AA a x) (AA x b)
  | otherwise = AA a b
tira x (FF a b)
  | x > a && x < b = Uniao (FA a x) (AF x b)
  | x == a = AF a b
  | x == b = FA a b
  | otherwise = FF a b
tira x (AF a b)
  | x > a && x < b = Uniao (AA a x) (AF x b)
  | x == b = AA a b
  | otherwise = AF a b
tira x (FA a b)
  | x > a && x < b = Uniao (FA a x) (AA x b)
  | x == a = AA a b
  | otherwise = FA a b
tira x (Uniao a b) = Uniao (tira x a) (tira x b)
```

6. Apresente uma definição alternativa da função `func`, usando recursividade explícita em vez de funções de ordem superior e fazendo uma única travessia da lista.

```
func :: Float -> [(Float,Float)] -> [Float]
func x l = map snd (filter ((>x) . fst) l)
```

```
func :: Float -> [(Float,Float)] -> [Float]
func _ [] = []
```

```
func x ((a,b):t)
  | a > x = b : func x t
  | otherwise = func x t
```

7. Defina a função `subseqSum :: [Int] -> Int -> Bool` tal que `subseqSum l k == True` se e só se existe uma sub-sequência da lista `l` cuja soma dos elementos é `k`. Por exemplo, `subseqSum [2,9,3,-4,2] 10 == True` e `subseqSum [2,9,3,4,2] 10 == False`.

```
subseqSum :: [Int] -> Int -> Bool
subseqSum [] _ = False
subseqSum l x = any ((== x) . sum) (inits l) || subseqSum (tail l) x
```

8. Defina a função `jogo :: Int -> (Int, Int) -> IO ()` tal que `jogo n (a,b)` gera uma lista aleatória de inteiros de tamanho `n` cujos valores estão compreendidos entre `a` e `b`, pede ao utilizador para indicar um número, verifica se a lista gerada tem uma sub-sequência cuja soma é esse número. No fim, escreve no ecrã a lista gerada e se a propriedade se verificou ou não. Pode assumir que a função da alínea anterior está definida. Sugestão: use a função `randomRIO :: Random a => (a,a) -> IO a`.

```
jogo :: Int -> (Int, Int) -> IO ()
jogo n (a,b) =
  sequence (replicate n (randomRIO (a,b)))
  >>= (\seq ->
    print seq
    >> putStrLn "Introduza um numero:"
    >> readLn
    >>= (\s ->
      if subseqSum seq s then
        putStrLn "A propriedade verifica-se."
      else
        putStrLn "A propriedade nao se verifica."
    )
  )
```

)

)

Made with  by RisingFisan.