Tuan

**Question 3:**

**a)**

> interleave :: [a] -> [a] -> [a]

> interleave [] ys = ys

> interleave (x:xs) ys = x : (interleave ys xs)

**b)**

The first sex elements of interleave [2,2] [1..] are going to be 2, 1, 2, 2, 3, 4.

**c)**

> interleaveList :: [[a]] -> [a]

> interleaveList = foldr interleave []

**d)**

The first eight elements of interleaveList [[1,2], [3,4], [5,6], [7,8], [9,10]..] are going to be 1, 3, 2, 5, 4, 7, 6, 9.

**e)**

> allpairs :: [a] -> [b] -> [(a, b)]

> allpairs xs ys = [(x, y) | x <- xs, y <- ys]

This function will not always give the correct result because for the first element of xs it has to traverse all the elements of ys. Therefore, not each pair will be included in some initial section of the result.

**f)**

> allpairs2 :: [a] -> [b] -> [(a, b)]

> allpairs2 xs ys = interleaveList [map (\a -> (x, a)) ys | x <- xs]

**g)**

> alltrees :: [Tree]

> alltrees = Nil : otherTrees

> where otherTrees = interleaveList [map (\a -> Fork x a) alltrees

> | x <- alltrees]

**Question 4:**

**a)**

> type Event = String

> type Country = String

> data Medal = Gold | Silver | Bronze

> deriving (Eq, Show)

> type Winners = [(Event, Country, Medal)]

**b)**

> countmedals :: Winners -> Medal -> Country -> Int

> countmedals [] medal country = 0

> countmedals (x : xs) medal country =

> if (medal == m && country == c)

> then 1 + countmedals xs medal country

> else countmedals xs medal country

> where (e, c, m) = x

**c)**

> score :: Winners -> Country -> Int

> score win country = 3 \* countmedals win Gold country

> + 2 \* countmedals win Silver country

> + countmedals win Bronze country

**d)**

> rank :: Winners -> [Country] -> [Int]

> rank win xs = helperFunc win xs (map (score win) xs)

> helperFunc :: Winners -> [Country] -> [Int] -> [Int]

> helperFunc win [] ys = []

> helperFunc win (x : xs) ys = largerInList (score win x) ys

> : helperFunc win xs ys

> largerInList :: Int -> [Int] -> Int

> largerInList n [] = 1

> largerInList n (x : xs) = if x > n

> then 1 + largerInList n xs

> else largerInList n xs

largerInList searches in which place does a number rank in a list.

**e)**

> padLeft :: Int -> String -> String

> padLeft n str = str ++ replicate (n - length str) ' '

> padRight :: Int -> String -> String

> padRight n str = replicate (n - length str) ' ' ++ str

> medalTable :: Winners -> [Country] -> IO()

> medalTable win xs = putStr (padLeft 10 "Country" ++ " " ++

> padRight 4 "Gold" ++ " " ++

> padRight 6 "Silver" ++ " " ++

> padRight 6 "Bronze" ++ " " ++

> padRight 4 "Rank" ++ "\n" ++

> print' (zipping win (zip xs (rank win xs))))

> print' :: [(Country, Int, Int, Int, Int)] -> String

> print' [] = []

> print' (x : xs) = padLeft 10 c ++ " " ++

> padRight 4 (show g) ++ " " ++

> padRight 6 (show s) ++ " " ++

> padRight 6 (show b) ++ " " ++

> padRight 4 (show r) ++ "\n" ++ print' xs

> where (c, g, s, b, r) = x

> zipping :: Winners -> [(Country, Int)] ->

> [(Country, Int, Int, Int, Int)]

> zipping win [] = []

> zipping win (x : xs) = (country, gold, silver, bronze, rank)

> : zipping win xs

> where country = fst x

> gold = countmedals win Gold country

> silver = countmedals win Silver country

> bronze = countmedals win Bronze country

> rank = snd x

With zipping, I make a list of 5-tuples, the first element of which is the name of the country, the second element is the number of gold medals, the third – the number of silver medals, the fourth – the number of bronze medals, and the fifth element is the rank of the country.

With print’, from the list of 5-tuples I create a big string. Each line consists of the name of the country, the number of gold, silver, bronze medals and the rank of the country.

With medalTable, I just print the string.