GABRIEL MOISE

Question 3

(a)
```
> interleave :: [a] -> [a] -> [a]
> interleave (x:xs) ys = x: interleave ys xs
> interleave [] ys = ys
```

(b) The first 6 elements of the result of interleave [2,2] [1..] are 2,1,2,2,3,4 as we first alternate the first 2 elements of each list and then we print the rest of the infinite list.

(c)
```
> interleaveList :: [[a]] -> [a]
> interleaveList = foldr interleave []
```

As the function interleave ensures us that every element of an input list can be found after a finite number of steps, then the interleaveList function, which is built based on multiple applications of the interleave function, is ensured to have the same property.

Comment: Here, I feel that my explication lacks some more specific justification and it is more of an intuitive argumentation. How can I express myself more rigorously?

(d) The first 8 elements of the result of interleaveList [[1,2], [3,4], [5,6], [7,8], [9,10]...] are 1,3,2,5,4,7,6,9. Let's say that we got to the point where we have the elements of the accumulator x and y and we now interleave the accumulator with [9,10]. Therefore, the first 4 elements of the resulting list will be [9,x,10,y], according to the definition of interleave. Then, after interleaving the list with [7,8], the first 6 elements will be [7,9,8,x,10,y], then [5,7,6,9,8,x,10,y], then [3,5,4,7,6,9,8,x,10,y], and finally [1,3,2,5,4,7,6,9,8,x,10,y], so the first 8 elements are 1,3,2,5,4,7,6,9.

(e)
```
> allpairs :: [a] -> [b] -> [(a,b)]
> allpairs xs ys = [(x,y) | x <- xs, y <- ys]
```

The problem is that if the ys list is infinite, then there will be pairs which won't be reached in a finite number of steps, therefore we don't get a correct result. Let's imagine that xs is a list with at least 2 elements, the first being a and the second, b. Then, with the current definition of allpairs, the pairs of the form (a,y) with y from ys are returned first, and therefore no pair of the form (b,y) with y from ys will be reached in a finite number of steps, as there are infinitely many pairs before.

(f)
```
> allpairs2 :: [a] -> [b] -> [(a,b)]
> allpairs2 xs ys = interleaveList [ [(x,y) | y <- ys] | x <- xs]
```

By interleaving (with interleaveList) a finite or infinite number of lists (finite or infinite) that together form the Cartesian product of the two input lists, we are ensured (as we proved at (c)) that every pair can be found in a finite number of steps.

(g)
```
> data Tree = Nil | Fork Tree Tree
> alltrees :: [Tree]
> alltrees = Nil : rest
>     where rest = map (uncurry Fork) (allpairs2 alltrees alltrees)
```

This way, we create all the finite elements of the datatype Tree. We apply the (uncurry Fork) function to every pair of the Cartesian product of two alltrees arguments (therefore this definition is recursive) to obtain new trees which have the left subtree the first element of a specific pair from (allpairs2 alltrees alltrees) and the right subtree the second one.

Recall that we used uncurry defined as:

```
> uncurry :: (a -> b -> c) -> (a,b) -> c
> uncurry f (x,y) = f x y
```

FUNCTIONAL PROGRAMMING - 2012

GABRIEL MOISE

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 winners medal country
>   | null winners = 0
>   | (medal == m) && (country == c) = 1 + countmedals (tail winners) medal country
>   | otherwise = countmedals (tail winners) medal country
>     where (e, c, m) = head winners
```
(c)
```
> score :: Winners -> Country -> Int
> score winners country = 3 * countmedals winners Gold country +
>                         2 * countmedals winners Silver country +
>                             countmedals winners Bronze country
```
(d)
```
> ranking :: Winners -> Country -> Int
> ranking winners country = 1 + length (filter better winners)
>     where better (e, c, m) = score winners c > score winners country
```
This function calculates the ranking of a particular country by comparing its "score" with the "scores" of all the other countries from the list.
```
> rank :: Winners -> [Country] -> [Int]
> rank winners [] = []
> rank winners (c:cs) = ranking winners c : rank winners cs
```
(e)
```
> ljustify :: Country -> Int -> String
> ljustify country n = country ++ concat (take (n-ln) (repeat " "))
>     where ln = length country
```
We justify to the left the name of the countries, which appear in the first column
```
> rjustify :: String -> Int -> String
> rjustify number n = concat (take (n-ln) (repeat " ")) ++ number
>     where ln = length number
```
We justify to the right the data we obtain from the results (winners)

```
> table :: Winners -> [Country] -> String
> table winners [] = (ljustify "Country" 10) ++ " Gold Silver Bronze Rank\n"
> table winners (c:cs) =
>        table winners cs ++ (ljustify c 10) ++ "   " ++ (rjustify (show gold) 4) ++ "  "
>        ++ (rjustify (show silver) 6) ++ "  " ++ (rjustify (show bronze) 6) ++ "  "
>        ++ (rjustify (show pos) 4) ++ "\n"
>      where gold = countmedals winners Gold c
>            silver = countmedals winners Silver c
>            bronze = countmedals winners Bronze c
>            pos = ranking winners c
```

This function creates the string we want to print to obtain the "medaltable". We justify the countries to the left with 10 as we are told that is the maximum length of the name of a country and the numbers according to their column. We observe that we use the list of countries in the reversed order for us, in order to be able to write the headings, so we'll print the string with the list of countries reversed:

```
> medaltable :: Winners -> [Country] -> IO ()
> medaltable winners cs = putStr (table winners (reverse cs))
```