**6.5**

newElement :: a -> a -> Int -> [a] -> [a]

newElement x z ln zs = take ln zs ++ [x] ++ [z] ++ drop ln zs

include :: a -> [a] -> [[a]]

include x ys =

(foldr (\(z,zs) acc -> (newElement x z (length ys - length acc - 1) zs) : acc )

[] (splits ys)) ++ [concat ([ys] ++ [[x]])]

{-

We created the include function using foldr this way:

* the lambda function used takes each pair formed by splits ys and the acc variable, which is the accumulator (it keeps the result) and adds a new element to acc, that element being result of the function newElement that will be interpreted below
* initially, the accumulator variable is [[]], as we want our result to be a list of lists
* we are going to work on the pairs of ys, the list where we want to include x, so we have the last argument as splits ys

Now, what does the function newElement do?

* the argument x is the thing we want to include in the list
* the argument z is the first element of each pair from splits ys, and it initially was in the list at the position ln + 1, and that's where we will introduce x and after that we will introduce z and then the rest of the list, as we can see :
* take ln takes the elements before the position where z was supposed to be
* then we add x on the position of z
* then we add z
* finally we add the rest of the list, which is after the position where z initially was ( drop ln zs )
* the argument ln is defined in the lambda function as being

ln = length ys - length acc - 1

because we create the result starting from the last element of splits ys, and then going backwards.

In order to know where to place x and z in a given list zs, we needed to correlate at what step we were in the process of creating the result in acc with the position where we insert x and z.

When acc is [[ ]], we start with the pair formed by the last element of ys and the first elements (init ys). Therefore we want to place x at the end and then to place z after. So we need to take the first (length ys -1) elements (because that's the length of init (ys)) and then add x, then add z, and then drop (length ys -1), which will be [].

This list will be added to the result, which is stored in acc.

After that, as we move on, the length of the result increases by 1 after each addition, and this tells us that x will need to be placed further back in the list given by the second element of the pair (z,zs).

Therefore, x has to be placed on the (length ys - length acc) position, so that's why we take the first (length ys - length acc -1) elements from zs, then we add x and z, and then we add the rest, which is drop (length ys - length acc -1) zs.

As a last thing, we can see that the last list, which is basically the ys list and then the element x added at the end, was not treated as a case, because we always put the element z after x. Therefore, at the end we add the concatenation of [ys] and [[x]], which is going to be the needed list, so then we add it to the result, as a list of lists.

-}

permutations :: [a] -> [[a]]

permutations [] = [[]]

permutations (x:xs) = [ zs | ys <- permutations xs, zs <- include x ys]

permutations' :: [a] -> [[a]]

permutations' xs = foldr (\x acc -> concat (map (include x) acc) ) [ [] ] xs

{-

Here we create the list of permutations by taking each element of xs and then applying the (include x) function to all of the elements from the result at that moment. Firstly, we will form [[[1]]], then we will concatenate it to [[1]], then we apply map (include 2) on it to get [[[2,1],[1,2]]], then we concatenate it to get [[1,2],[2,1]], and then we keep going, creating our permutations by "including" a new element each time in all of the elements of the current result (which is a list of lists). Because this method is very similar in behavior as the permutations implementation from above, the order will remain the same.

-}