5.1

```
take' :: Int -> [a] -> [a]
take' 0 _ = []          or for n ≤ 0.
take' _ [] = []
take' n (x:xs) = x : take' (n-1) xs
```

```
take' 5 [1..4] = [1,2,3,4]
take' 3 [1..6] = [1,2,3]
take' 2 [2..] = [2,3]
```

```
drop' : Int -> [a] -> [a]
drop' 0 xs = xs
drop' _ [] = []
drop' n (x:xs) = drop' (n-1) xs
```

```
drop' 5 [1..3] = []
drop' 5 [1..7] = [6,7]
drop' 2 [1..] = [3,4,5, ...
```

As the function take is defined the same way take' is, take n xs is strict in n, as if we had take' ⊥ xs, it would have to compare ⊥ with 0 from the first limit case, resulting in ⊥. However, take' is not strict in xs (and neither take because they behave in the same way) as for instance take' 0 ⊥ will return [], as the first limit case suggests. By supposing that there is a function that would not be strict neither in n, nor in xs, we would have take'' ⊥ xs ≠ ⊥ and take'' n ⊥ ≠ ⊥. But both of these instances need to be treated like limit cases, therefore we need to write them in some order. If we start with the first, then take'' n ⊥ will be ⊥ as it will have to compare ⊥ with xs and if we start with the second, then take'' ⊥ xs will be ⊥ as it will have to compare ⊥ with n. So take has to be strict in an argument.

– is there a different way?

5.2 Function map is defined as

```
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = f x : map f xs
```

Because of the way map is defined in Haskell, we can deduce that it is not strict in its first argument, as map ⊥ [] = [], but it is strict in its second, as the limit case would require ⊥ to be compared to [] (for an eventual map f ⊥).

Function map f is defined as

```
(map f) :: [a] -> [b]
```

So map f ⊥ = map f [] = [] but that's not enough.

This function takes a list, and by applying f to each of the elements from [a], it returns [b]. (map f) ⊥ = map f ⊥ and because we already proved that map is strict in its second argument, we can say that (map f) is strict.

```
(map f) [] = []
(map f) (x:xs) = f x : (map f) xs
```

5.3 ✓

```
evens :: [a] -> [a]
evens [] = []
evens [x] = [x]
evens (x:y:ys) = x : evens ys
```

evens ['a'..'z'] = "acegikmoqsuwy" ✓

```
odds :: [a] -> [a]
odds [] = []
odds (x:xs) = evens xs
```

odds ['a'..'z'] = "bdfhjlnprtvxz" ✓

```
alternates :: [a] -> ([a], [a])
alternates [] = ([], [])
alternates [x] = ([x], [])
alternates (x:y:l) = (x:l1, y:l2)
        where (l1, l2) = alternates l
```

✓

This function calculates the result in a single pass along the list as it takes two elements at a time, starting from the head (exception for the treated limit cases) and puts them in the correct lists, which are formed recursively. Derivation?

6.1

```
curry' :: ((a,b) -> c) -> a -> b -> c
curry' f x y = f (x,y)      -- (DEF. OF CURRY') = (A)

uncurry' :: (a -> b -> c) -> ((a,b) -> c)
uncurry' f (x,y) = f x y    -- (DEF. OF UNCURRY') = (B)
```

✓ ✓

By composing curry' with uncurry' we get two functions:

```
curry'. uncurry' :: (a -> b -> c) -> a -> b -> c
uncurry'. curry' :: ((a,b) -> c) -> (a,b) -> c
```

(i) Let's check that curry'. uncurry' = id. To do that we will apply to it the argument f, where f :: a -> b -> c and the arguments x and y, where x :: a, y :: b.

$$(curry'.\ uncurry')\ f\ x\ y = \Big(\big((curry'.\ uncurry')\ f\big)\ x\Big)\ y \overset{(*)}{=} \Big(\big(curry'\ (uncurry'\ f)\big)\ x\Big)\ y =$$

$$= \big(curry'\ (uncurry'\ f)\ x\big)\ y = curry'\ (uncurry'\ f)\ x\ y \overset{(A)}{=} (uncurry'\ f)\ (x,y) =$$

$$= uncurry'\ f\ (x,y) \overset{(B)}{=} f\ x\ y \Longrightarrow \boxed{curry'.\ uncurry' = id}$$ (for the (a -> b -> c) type)

(*) THE DEFINITION OF COMPOSITION

You might put in fewer parentheses.

(ii) Let's check that uncurry'. curry' = id. To do that we will apply to it the argument f, where f :: (a,b) -> c and the argument (x,y), where (x,y) :: (a,b).

(uncurry'. curry') f (x,y) = ((uncurry'. curry') f) (x,y) $\overset{*}{=}$ (uncurry' (curry' f)) (x,y) =

= uncurry' (curry' f) (x,y) $\overset{B}{=}$ (curry' f) x y = curry' f x y $\overset{A}{=}$ f (x,y) ⇒

⇒ **uncurry'. curry' = id** (for the ((a,b) -> c) type). ✓

From (i) and (ii) we can conclude that curry' and uncurry' are mutually inverse (and because they are defined exactly like curry and uncurry, then those two are mutually inverse too).

6.2 In the lecture, zip was defined as

```
zip :: [a] -> [b] -> [(a,b)]
zip (x:xs) (y:ys) = (x,y) : zip xs ys
zip _ _ = []
```

β-

If we switched the two equations, we will have zip _ _ = [] as first equation, so no matter what two lists we would want to zip, the result will always be []. The order is that way because our limit case happens only when one of the lists is empty (or both are), so we only get to the second equation in this case and the result here will always be [] because zip [x] [] = [], zip [] [x] = [] and zip [] [] = [], so that's why the underlines are used. The left-hand side patterns of the equations overlap, so we need a specific order for them, but if we want the order to be unimportant, we need them to not overlap.

Therefore we can create:

```
zip' :: [a] -> [b] -> [(a,b)]
zip' [] [] = []
zip' [x] [] = []
zip' [] [x] = []
zip' (x:xs) (y:ys) = (x,y) : zip' xs ys
```

But this will fail on, e.g.: zip' [] [1,2].

As each equation treats a distinct case, the order doesn't matter.

6.3

```
zipWith' :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith' f xs ys = map (uncurry f) (zip xs ys)
```

α ✓

If f was initially f :: a -> b -> c, then uncurry f :: (a,b) -> c, and therefore map (uncurry f) (zip xs ys) is a list of c, where c resulted from applying the (uncurry f) function to the (a,b) pair

However, zipWith is defined recursively as:

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
zipWith _ _ _ = []   -- the limit case that always returns []
```

Therefore, zip can be defined after zipWith:

```
zip = zipWith f
    where f x y = (x,y)
```

Or if we want to use a lambda expression to simplify:

```
zip = zipWith (\x y -> (x,y))
```

6.4

```
splits :: [a] -> [(a,[a])]
splits xs = [(xs !! p , take p xs ++ drop (p+1) xs | p <- [0..(length xs)-1]]
```

This way, for each element of the list (let's say x_i), we create the list without that element $[x_0, x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_{n-1}]$ by taking the first i elements and then concatening it with the list formed by dropping the first $(i+1)$ elements.

Thus, we can create:

```
permutations :: [a] -> [[a]]
permutations [x] = [[x]]
permutations xs = [x:zs | (x,ys) <- splits xs, zs <- permutations ys]
```

6.6

The definition of unfold, using recursivity would look like this:

```
unfold :: (a -> Bool) -> (a -> b) -> (a -> a) -> a -> [b]
unfold f g h x
  | f x = []
  | otherwise = g x : unfold f g h (h x)
```

And it uses the fact that when we want to unfold a result of a foldr, we start with taking away the last part that was added by foldr to the result, therefore we have g x, and we need to keep unfolding the rest, which will be h x. As in our lecture, g is a function similar to head (as behaviour) because it takes away the last part that was added, so we are left with h x (where h behaves like tail), giving the rest, which needs to be unfolded. The process terminates when the rest no longer respects f (which behaves like null for the case with head and tail).

10.4 Coming back here, we can recreate the function splits' as suggested by the task (by using unfold):

```
splits' :: [a] -> [(a, [a])]
splits' xs = unfold verify here next (0, xs)
```

Now we will define the three functions we use:

```
Verify :: (int, [a]) -> Bool
Verify (pos, xs) = pos > length xs - 1
```

This function behaves like null, checking if the position from where we want to take out x is between 0 and length xs - 1, meaning we have an element on that position (pos) in the list.

```
here :: (int, [a]) -> (a, [a])
here (pos, xs) = (xs!! pos, take pos xs ++ drop (pos+1) xs )
```

This function, given a valid position and a list, gets the element from that position out and forms the list as++ bs, where as = take pos xs, which are the elements before our position, and bs= drop (p+1) xs, which are the elements after our position.

```
next :: (int, [a]) -> (int, [a])
next (pos, xs) = (pos+1, xs)
```

This function behaves like tail, producing the next case that needs to be treated, until pos gets out of range.

However, in my honest opinion, the more natural way to create splits was the first one, but I tried to do it using unfold to get accustomed with folding and unfolding. is there a noticeable difference between splits and splits' that would determine one to be better than the other (time, memory)? Cause I don't understand why it was suggested to be done with unfold.

For example:

length (splits [1..50000]) needs 0.02 secs and 17,265,576 bytes to show 50000 and

length (splits' [1..50000]) needs 5.94 secs and 34,466,832 bytes to show 50000.

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]])]
```

{- map (\(z,zs) -> newElement x z (length ys - length acc - 1) zs) (splits ys)

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
```

= [zs | ys ← acc, zs ← include x ys]

{-

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.

-}