Tuan

**Question 5.1:**

> take' :: Int -> [a] -> [a]

> take' 0 xs = []

> take' n [] = []

> take' n (x : xs) = x : take' (n - 1) xs

> drop' :: Int -> [a] -> [a]

> drop' 0 xs = xs

> drop' n [] = []

> drop' n (x : xs) = drop' (n - 1) xs

The second and the third line of the function *take’* can be swapped and in the trivial cases it would still produce the same result. But if we write *take 0 undefined* in Haskell, the output would be []. So we know that the second line is indeed above the third line. In the same way we can reason that *drop’* should be defined in that way.

*take n xs* is strict in *n*, but not strict in *xs*. The function cannot be strict in neither, because it has to pattern match.

**Question 5.2:**

*map* is not is strict in its first value. If we wrote *map undefined []* the result would be *[]*. But *map f* is strict, because the way that the function is defined in the Prelude, it has to evaluate the second argument of *map*.

**Question 5.3:**

> evens :: [a] -> [a]

> evens [] = []

> evens [x] = [x]

> evens (x : y : xs) = x : evens xs

> odds :: [a] -> [a]

> odds [] = []

> odds xs = evens (tail xs)

> alternates2 :: [a] -> ([a], [a])

> alternates2 [] = ([], [])

> alternates2 [x] = ([x], [])

> alternates2 (x : y : xs) = (x : p, y : q)

> where (p, q) = alternates xs

**Question 6.1:**

> curry' :: ((a, b) -> c) -> (a -> b -> c)

> curry' f x y = f (x, y)

> uncurry' :: (a -> b -> c) -> (a, b) -> c

> uncurry' f (x, y) = f x y

Assume that the functions *func* and *funcc* are defined with types *func :: (a, b) -> c* and *funcc :: a -> b -> c*. I have to now prove that *curry’ . uncurry’ funcc (a, b) = funcc a b* and that *uncurry’ . curry’ func a b = func (a, b)*.

*(curry’ . uncurry’) funcc a b*

*=* {definition of functional composition}

*curry’ (uncurry’ funcc) a b*

*=* {definition of *uncurry’*}

*curry’ func a b*

*=* {definition of *curry’*}

*func (a, b)*

*=* {*func (a, b) = funcc a b*}

*funcc a b*

*(uncurry’ . curry’) func (a, b)*

= {definition of functional composition}

*uncurry’ (curry’ func) (a, b)*

*=* {definition of *curry’*}

*uncurry’ funcc (a, b)*

*=* {definition of *uncurry’*}

*funcc a b*

*=* {*funcc a b = func (a, b)*}

*func (a, b)*

Therefore, (*curry . uncurry*) and (*uncurry . curry*) are mutually inverse.

**Question 6.2:**

If the two equations were switched, any call of the function *zip* would pattern match with the first equation, and the result would always be the empty list.

> zip' :: [a] -> [b] -> [(a,b)]

> zip' [] bs = []

> zip' as [] = []

> zip' (a:as) (b:bs) = (a,b) : zip' as bs

With these set of equations the order of them does not matter because the patterns do not overlap.

**Question 6.3:**

If we want to defined *zipWtih* only with *zip* and predefined functions, the definition should be:

> zipWith' :: (a -> b -> c) -> [a] -> [b] -> [c]

> zipWith' f xs ys = map (uncurry f) (zip xs ys)

If we want the recursive definition, it would be:

> zipWith' :: (a -> b -> c) -> [a] -> [b] -> [c]

> zipWith' f [] ys = []

> zipWith' f xs [] = []

> zipWith' f (x : xs) (y : ys) = f x y : zipWith' f xs ys

So we can define *zip* with the function *zipWith*:

> zip2 :: [a] -> [b] -> [(a, b)]

> zip2 xs ys = zipWith' toTuple xs ys

> where toTuple x y = (x, y)

**Question 6.4:**

> split :: [a] -> [(a, [a])]

> split xs = unfold null' head' tail' (xs, [])

> where null' (xs, \_) = null xs

> head' (x : xs, ys) = (x , reverse ys ++ xs)

> tail' (x : xs, ys) = (xs, x : ys)

**Question 6.5:**

The definition for permutations using foldr is:

> permutations’ :: [a] -> [[a]]

> permutations’ = foldr helperFuncPerm [[]]

> helperFuncPerm :: a -> [[a]] -> [[a]]

> helperFuncPerm n xss = concat (map (include n) xss)

The definition for include using foldr is:

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

> include' n (x : xs) = foldr helperFunc [[n]] (x : xs)

> helperFunc :: a -> [[a]] -> [[a]]

> helperFunc n [[]] = [[n]]

> helperFunc n (xs : xss) = (head xs : n : tail xs) : map (n:) (xs : xss)

**Question 6.6:**

> unfold1 :: (a -> Bool) -> (a -> b) -> (a -> a) -> a -> [b]

> unfold1 null head tail x = if (not . null) x then head x : unfold null head tail (tail x) else []