<u>Question 3.1:</u>

```
instance Ord a => Ord [a] where
 [] <= _ = True
 (x : xs) <= [] = False
 (x : xs) <= (y : ys) | x < y = True
                      | x == y = xs < ys
                      | otherwise = False
```

<u>Question 3.2:</u>

The first statement is false because it is assumed that the functions will take only one argument, but by definition they take two.
The second statement is true because f . g x y = f (g x y) (definition of functional composition).
The third statement is false because the function g has a type (a -> b -> c), and the function f has a type (c -> d), so the type of the output of g does not match the type of the input of the function f.

<u>Question 3.3:</u>

```
subst :: (a -> b -> c) -> (a -> b) -> a -> c
fix :: (a -> a) -> a
twice :: (a -> a) -> a -> a
selfie :: (a -> a) -> a -> a
```

<u>Question 3.4:</u>

a) [] : xs is well typed, provided that xs is a list of lists or the empty list. It that case the equation does not hold.
b) xs : [] holds for all xs.
c) [[]] ++ xs is badly typed.
d) [[]] ++ [xs] holds for all xs, which are lists.
e) [] : xs is well typed, in case that xs is a list of lists or the empty list. When xs = [], the equation does not hold, but when xs is a list of lists the equation holds.
f) In order xs : xs to be well typed, the xs has to be the empty list []. But when xs = [] the equation does not hold. It only holds when we the first list is a non-empty list, and the second one is the empty list.
g) [[]] ++ xs is well typed, provided xs is a list of lists. In that case the equation does not holds.
h) [xs] ++ [] holds for all xs, provided that xs is not a list.
i) xs : [] is well typed, but the equation does not hold.
j) xs : [xs] is well typed and the equation holds for all xs.
k) [[]] ++ xs is well typed but only when xs is a list of lists. In that case the equation does not hold.
l) The equation is well typed and it holds for all xs.

The following answers have been written assuming that xs can be anything. In case that is used the convention, which is used in the Richard Bird book, where xs is used to denote a list of elements, and xss to be a list of lists, then a), c), e), g), h), k) are badly typed.

Question 4.1:

The composition (f . g) is strict when (f . g) undefined = undefined.
(f . g) undefined
= {definition of functional composition}
f (g undefined)
= {the function g is strict}
f undefined
= {the function f is undefined}
undefined
Therefore, the composition (f. g) is strict, when the functions f and g are strict.

The converse is not true. Counterexample to that claim is:

```
g :: a -> Int
g _ = 0

f :: Int -> Int
f n = 1 `div` n
```

And 1 divided by 0 gives undefined.

Question 4.2:
When a type A -> B is given, the number of such functions are cardinality B to the power of cardinality A. So there are 27 functions that are of type Bool -> Bool. Out of those 27 functions, only 11 are computable – 9 functions in which f undefined = undefined, and f True, f False can be True, False or undefined, and the two functions, in which for no matter the input all of the outputs are True or all of the outputs are False. All of these functions are definable.

Question 4.3:

```
(&&) :: Bool -> Bool -> Bool
(&&) False x = False
(&&) True x = x
```

The remaining cases are (undefined &&& undefined), (undefined &&& True), (True &&& undefined). Because the function is computable, it means that all of these have to be undefined. If it was anything different from undefined, it would not be computable, because it would have to evaluate undefined, which would not be computable.

Question 4.4:

```
import Data.Char

units :: [String]
units = ["Two", "Three", "Four", "Five", "Six", "Seven", "Eight", "Nine", "Ten"]
```

```
--transforms a character to to lower case
canonical :: Char -> Char
canonical c | isLower c = c
            | isUpper c = toLower c
            | otherwise = c

line :: [Char]
line = "Went to mow a meadow \n"

line1 :: Int -> String
line1 1 = "One man went to mow \n"
line1 n = units !! (n - 2) ++ " men went to mow \n"

--turns all the letters in line3 (n - 1) to lower case and concatinates it to number ++ men
line3 :: Int -> String
line3 1 = "One man and his dog \n"
line3 n = units !! (n - 2) ++ " men, " ++ map canonical (line3 (n - 1))

verse :: Int -> String
verse n = line1 n ++ line ++ line3 n ++ line

song :: Int -> String
song 0 = ""
song 1 = verse 1
song n = song (n - 1) ++ "\n" ++ verse n
```

Problem sheet by Tuan Nguen