1] As the class of ordered types is already declared, we can use it in order to write an instance declaration for ordered lists. We will define the (<=) operator as the other ones will be defined based on it (in the example from the sheet we can see that (>) = not (<=), (>=) = (==) || (not (<=)) and (<) = not ((==) || (not(<=))))

↑declared from Eq

α −

```
instance Ord a => Ord [a] where
        (<), (<=), (>), (>=) :: [a] -> [a] -> Bool
        [] <= _ = True     -- the limit case (this is the first one because [] <= [] is True)
        _ <= [] = False    -- the other limit case
        (x:xs) <= (y:ys) = (x<y) || ((x==y) && (xs <= ys))
```

α ✓

We apply the operator (<=) recursively for heads to create a lexicographical order between two lists: (x:xs) and (y:ys).

$\boxed{3.2}$ We start with $h\ x\ y = f\ (g\ x\ y)$. From the equality we can deduce the types of f, g and h:

α

$$\underset{A}{(h}\ \underset{B}{x)}\ \underset{C}{y} = \underset{D}{f}\ ((\underset{E}{g}\ \underset{B}{x})\ \underset{C}{y})$$

$h\ x \Rightarrow A = B \to F$

$(h\ x)\ y \Rightarrow F = C \to G$

$g\ x \Rightarrow E = B \to H$

$(g\ x)\ y \Rightarrow H = C \to i$

$f\ ((g\ x)\ y) \Rightarrow D = i \to J$

This is a bit laboured.

From the equality, we get that $G = J$.

$A = B \to F$	$D = i \to J$	$E = B \to H$
$A = B \to (C \to G)$	$D = i \to G$	$E = B \to (C \to i)$
$h :: a \to (b \to c)$	$f :: d \to c$	$g :: a \to (b \to d)$ ✓

Now, we can test the 1., 2. and 3. equalities:

1. $h = f . g$

That would mean f and g can be composed that way. However, g takes an argument a an returns a function $(b \to d)$, whereas f takes d as argument. Therefore, f.g doesn't make sense, according to the rules of composition. ✓

2. $h\ x = f \cdot g\ x$

In order to check if these two functions are equal, we use y as argument for them and see what they return:

$(h\ x)(y) = h\ x\ y = f\ (g\ x\ y)$ (from the initial equality)

$(f \cdot g\ x)(y) = (f \cdot (g\ x))\ (y) = f\ ((g\ x)\ (y)) = f\ (g\ x\ y)$ (the definition of compositi[on]

So, $h\ x$ and $f \cdot g\ x$ return the same value given an argument y, so they're equal.

3. $h\ x\ y = (f \cdot g)\ x\ y$

Again, we have $f \cdot g$, which we proved at 1. that cannot happen, so this equality is False

[.3] > $subst\ f\ g\ x = (f\ x)(g\ x)$

(-) We begin by allocating a type variable to each name:

$subst :: A$; $f :: B$; $g :: C$; $x :: \Delta$

$subst\ f \Rightarrow A = B \to E$

$subst\ f)\ g \Rightarrow E = C \to F$

$subst\ f)\ g)\ x \Rightarrow F = \Delta \to G$

$f\ x \Rightarrow B = \Delta \to H$

$g\ x \Rightarrow C = \Delta \to i$

$(f\ x)(g\ x) \Rightarrow H = i \to J$

Better not to introduce so many names – you have 13!
Better to work from the inner-most subexpression outwards.

From the equality we obtain $G = J$

$A = B \to E$

$A = (\Delta \to H) \to (C \to F)$

$A = (\Delta \to (i \to J)) \to ((\Delta \to i) \to (\Delta \to G))$

$A = (\Delta \to i \to G) \to ((\Delta \to i) \to \Delta \to G)$

So, $subst :: (\alpha \to \gamma \to \beta) \to ((\alpha \to \gamma) \to \alpha \to \beta)$

> $fix\ f = f\ (fix\ f)$

$fix :: A$; $f = B$

$fix\ f \Rightarrow A = B \to C$

$f(fix\ f) \Rightarrow B = C \to D$

From the equality we obtain $C = D$

$A = B \to C$

$A = (C \to D) \to C$

$A = (C \to C) \to C$

So, $fix :: (\alpha \to \alpha) \to \alpha$.

> twice f = f · f

Let's begin with f. Because f can be composed with itself, then its domain and codomain must be the same type. So,

$$f :: A \to A \quad \text{and} \quad f \cdot f :: A \to A$$

Now, if twice :: B, then

twice f $\Rightarrow B = (A \to A) \to C$

From the initial equality we have $C = A \to A$, so $B = (A \to A) \to (A \to A)$. Therefore,

twice :: $(\alpha \to \alpha) \to \alpha \to \alpha$ ✓

> selfie f = f f

selfie :: A, f :: B

selfie f $\Rightarrow A = B \to C$

f f $\Rightarrow B = B \to D$

From the equality we have $C = D \Rightarrow B = B \to C$

Now,

$$A = B \to C$$
$$A = (B \to C) \to C$$
$$A = ((B \to C) \to C) \to C$$
$$A = (((B \to C) \to C) \to C) \to C$$

....

And so on, we will never get to a finite result, so selfie cannot be defined. ✓

(In the ghci, after creating selfie, we get the error "cannot construct the infinite type: $t \sim t \to t_1$", selfie being defined selfie :: $(t \to t_1) \to t_1$, whereas f :: $t \to t_1$, so t corresponds to B and t_1 to C)

$\boxed{3.4}$ a) [] : xs = xs **FALSE**

B+ For this to be ~~syntactically~~ type correct, xs :: [[α]]. if xs = $[y_1, y_2, \dots, y_n]$, where $y_1, y_2, \dots, y_n$ :: [α], then [] : xs = [] : $[y_1, y_2, \dots, y_n]$ = $[[], y_1, y_2, \dots, y_n] \neq$ xs, so this equality is False. (it is also False for infinite lists, as head ([] : xs) = [], whereas head xs is not necessarily). For xs = ⊥, [] : ⊥ = [[] ⊥], as when we run it the program goes silent after showing [[]. — Does it ever hold? xs – infinite list of []s

b) xs : [] = [xs] **TRUE** ✓

The type of xs can be anything (xs :: α) as it will be treated as an element for [], so xs : [] = [xs] holds for all xs. Also, ⊥ : [] = [⊥]

c) [[]] ++ xs = xs **FALSE** – does it ever hold? Say it's type correct.

xs :: [[α]] and by concatenating it with [[]], we add [] to its elements. So, the equality doesn't hold for any xs, neither for ⊥.

d) [[]]++[xs] = [[],xs] TRUE

xs:: [α]. By concatenating [[]] with [xs] we get a list [[],xs] that contains the elem from the two lists. So, the equality holds for all xs (including ⊥) ✓

e) [] : xs = [[],xs] FALSE × badly typed.

xs::[[α]]. By adding [] to the list of elements from xs. If xs = [ys], where ys the sequence of elements (which are lists) of xs. So, [] : xs = [] : [ys] = [[], ys] ≠ [[],xs]. So, the equality doesn't hold for any xs (neither for ⊥)

f) xs : xs = [xs,xs] BADLY TYPED ✓

From xs:xs, we get that xs has the same type as the elements from itself, which is impossible, so the equality is syntactically wrong, so it might be badly typed. A solution to the problem is bracketing the second xs, like in case j) below.

g) [[]] ++ xs = [xs] FALSE × badly typed — No

xs:: [[α]]. Then, xs = [ys], where ys is a sequence of elements (which are lists), so [[]]++xs = [[]]++[ys] = [[],ys] ≠ [xs]. So, the equality doesn't hold for any xs (neither for ⊥)

h) [xs] ++ [] = [xs] TRUE ✓

As [] :: [α], xs :: α, so [xs]++[] returns the [xs] list unmodified (as we added nothing to it), so the equality holds for all xs (including ⊥)

i) xs : [] = xs FALSE × badly typed

From b) we understand why this equality doesn't hold for any xs (neither for ⊥).

j) xs : [xs] = [xs, xs] TRUE ✓

As xs:: α, it can be added to [xs] :: [α] and we get [xs,xs]. So the equality holds for all xs (for xs=⊥, we run ⊥ : [⊥], which returns [⊥])

k) [[]] ++ xs = [[], xs] FALSE × badly typed

As stated at d), [[]] + [xs] = [[],xs], so this equality doesn't hold for any xs (because xs ≠ [xs]) (for xs = ⊥, ghci returns the same thing as for d), which is [[],⊥], so ⊥ is an exception here)

l) [xs] ++ [xs] = [xs, xs] TRUE ✓

From the definition of concatenation, by concatenating two lists, we obtain a list which contains all the elements of the two initial lists. So, the equality holds for all xs (for xs = ⊥, we have [⊥] ++ [⊥] = [⊥]) Yes. length [⊥,⊥] = 2, length [⊥] = 1.

QUESTION: Is [⊥,⊥] different from [⊥]. Is [⊥,⊥] even possible?

$\boxed{4.1}$ If f and g are strict, then f ⊥ = ⊥ and g ⊥ = ⊥

B

Now, (f·g)⊥ = f (g ⊥) = f ⊥ = ⊥ ⇒ f·g is also strict. ✓

Converse: f·g is strict ⇒ f and g are strict

As a counterexample:

```
inf :: Integer
inf = 1 + inf
g :: a -> (a,a)
g x = (x,x)
f :: (a,a) -> a
f (x, y) = x
```

f has only one argument, which is a pair

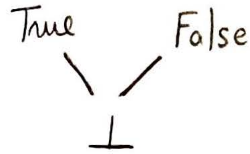
As we can see, (f·g) ⊥ = f (g ⊥) = f (⊥,⊥) = ⊥, so f·g is strict.

However, f is strict only to its first argument: f(⊥, y) = ⊥, but not for its second: f(x, ⊥) = x. So, f is not strict, so the converse is False. ×

f is strict

4.2 By counting $\bot$ as a Bool, there are 3 values of type Bool: $\bot$, False and True. So, a function f :: Bool -> Bool would have a domain consisting of 3 elements and the same codomain. So, the number of existing functions is $3^3 = 27$. ✓

α For a function f to be computable, it needs to respect the ordering: if $x \sqsubseteq y$, then $f\ x \sqsubseteq f\ y$. From the fact that $\bot \sqsubseteq$ False and $\bot \sqsubseteq$ True, we can conclude that $f\ \bot \sqsubseteq f$ True and $f\ \bot \sqsubseteq f$ False.



We distinguish 3 cases here:

1) $f\ \bot$ = False $\Rightarrow$ False $\sqsubseteq f$ False and False $\sqsubseteq f$ True

From the fact that False $\not\sqsubseteq$ True and False $\not\sqsubseteq \bot$, but False $\sqsubseteq$ False, we get to the conclusion that f True = False and f False = False, so f x = False, for all x in Bool. ✓

2) $f\ \bot$ = True

Reasoning in the same manner as we did in case 1), we obtain f x = True, for all x in Bool. ✓

3) $f\ \bot = \bot$

The only necessary things for f to be computable are $f\ \bot \sqsubseteq f$ True and $f\ \bot \sqsubseteq f$ False, so, by $f\ \bot = \bot$, they are both correct, for any values of f True and f False. For each of them we have 3 possibilities, so ~~by~~ using the product rule, we get 9 computable functions in this case.

In total, we have 1+1+9 = 11 computable functions ✓ could be clearer

For each computable function we can define in Haskell the 3 values for $f\ \bot$, f True and f False. For the cases where $f\ \bot = \bot$, we only define the value of f True and f False, as it will automatically mean that $f\ \bot = \bot$. For f1 x = False, we just define f like that and f2 x = True for case 2). So, all 11 computable functions are definable in Haskell. ✓

(tested the functions using inf :: Bool inf = not inf)

4.3 By trying every combination of two Bools in the ghci with the function (&&) we get:

α

1) False && False = False	4) True && False = False	undefined && False = undefined
2) False && True = False	5) True && True = True	undefined && True = undefined
3) False && undefined = False	6) True && undefined = undefined	undefined && undefined = undefined

A definition that would make it behave like this is:

```
(&&) :: Bool -> Bool -> Bool
False && x = False    -- this way the 1)-3) equalities work
True && x = x         -- this way the 4)-6) equalities work
```
✓

-- as we start with undefined for equalities 7)-9), we will automatically receive undefined as a result.

We create the function (&&&), which satisfies:

False &&& y = False

x &&& False = False

True &&& True = True

Now, we'll make a table of values from this information

&&&	$\perp$	F	T
$\perp$	?	F	?
F	F	F	F
T	?	F	T

Given that (&&&) is computable, then if $(x,y) \sqsubseteq (z,t) \Rightarrow x \&\&\& y \sqsubseteq z \&\&\& t$

So, $(\perp, T) \sqsubseteq (F,T) \Rightarrow \perp \&\&\& T \sqsubseteq F \Rightarrow \perp \&\&\& T \in \{\perp, F\}$

But, $(\perp, T) \sqsubseteq (T,T) \Rightarrow \perp \&\&\& T \sqsubseteq T \Rightarrow \perp \&\&\& T \in \{\perp, T\}$ $\Rightarrow \perp \&\&\& T = \perp$

Same way, $(T,\perp) \sqsubseteq (T,F) \Rightarrow T \&\&\& \perp \sqsubseteq F \Rightarrow T \&\&\& \perp \in \{\perp, F\}$

$(T,\perp) \sqsubseteq (T,T) \Rightarrow T \&\&\& \perp \sqsubseteq T \Rightarrow T \&\&\& \perp \in \{\perp, T\}$ $\Rightarrow T \&\&\& \perp = \perp$

Finally, $(\perp,\perp) \sqsubseteq (T,\perp) \Rightarrow \perp \&\&\& \perp \sqsubseteq \perp \Rightarrow \perp \&\&\& \perp = \perp$

Now we can finish the table of values for (&&&) and see there is only one computable function with the given properties:

&&&	$\perp$	F	T
$\perp$	$\perp$	F	$\perp$
F	F	F	F
T	$\perp$	F	T

However, this function cannot be defined in Haskell as the cases where False &&& undefined = False and undefined &&& False = False cannot both happen without getting to define (&&&) as being always False (and this is not the function we want!). Suppose both cases are ok to be defined. Then we need a definition for False &&& undefined, which needs to be **False &&& y = False** and a definition for undefined &&& False, which needs to be **x &&& False = False**. However, by having both definitions, we will need to write them in a chosen order. If we start with the first, the second case will transform into undefined &&& False = undefined (as the program will need to compare undefined with False) and if we start with the second definition, the first case will fail for the same reason. The problem is solved if (&&&) always returns False, but that would mean True &&& True would be False (so that is not correct).

In conclusion, (&&&) is unique (as it is computable), but it can't be defined in Haskell.

αβ

4.4

VOCABULARY PART

```
upperNumbers :: [String]
upperNumbers = ["Zero","One","Two","Three","Four","Five","Six","Seven","Eight","Nine","Ten"]
lowerNumbers :: [String]
lowerNumbers = ["zero","one","two","three","four","five","six","seven","eight","nine","ten"]
```

No need for both.

CASES FOR SINGULAR & PLURAL

```
plural :: Int -> String
plural 0 = ""
plural 1 = " man"
plural n = " men"
```

FIRST LINE

```
line1 :: Int -> String
line1 0 = ""
line1 n = (upperNumbers !! n) ++ (plural n) ++ rest
      where rest = " went to mow"
```

← No need for an extra definition

THIRD LINE WITHOUT THE FIRST TWO WORDS

```
restOfLine3 :: Int -> String
restOfLine3 0 = ""
restOfLine3 1 = "one man and his dog"
restOfLine3 n = (lowerNumbers !! n) ++ plural n ++ ", " ++ restOfLine3 (n-1)
```

FIRST TWO WORDS OF THE THIRD LINE

```
line3 :: Int -> String
line3 0 = ""
line3 1 = "One man and his dog"
line3 n = (upperNumbers !! n) ++ (plural n) ++ ", " ++ restOfLine3 (n-1)
```

repeated code.

CREATING EACH VERSE FROM LINES

```
verse :: Int -> String
verse 0 = ""
verse n = (line1 n) ++ "\n" ++ line ++ "\n" ++ (line3 n) ++ "\n" ++ line ++ "\n"
      where line = "Went to mow a meadow"
```

CREATING THE SONG FROM VERSES

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

GABRIEL MOISE – SHEET 2