WRITE ONLY ON ONE SIDE OF THE PAPER - the exams will be scanned in and only the front/odd pages will be read.

DO NOT WRITE WITH OTHER COLOUR THAN BLACK - coloured text may disappear during scanning.

PUT YOUR ID AND PAGE NUMBER ON EACH PAGE YOU SUBMIT - make sure that the amount of pages is equal to the amount you note on the front information page.

WRITE CLEARLY - if we cannot read you we cannot grade you.

PRELIMINARY AMOUNT OF POINTS : 6 (approx. one per question)
(-A list of selected functions from the Haskell modules: 
Prelude  
Data.List  
Data.Maybe  
Data.Char {-
-----------------------------------------------------  
-----------------------------------------------------

-- standard type classes  
Class Show a where  
  show :: a -> String

class Eq a where  
  (==), (/=) :: a -> a -> Bool

class (Eq a, Ord a) => Ord a where  
  (<), (<=), (>=), (>) :: a -> a -> Bool
  max, min :: a -> a -> a

class (Eq a, Show a) => Num a where  
  (+), (-), (*) :: a -> a -> a
  negate :: a -> a
  abs, signum :: a -> a
  fromInteger :: Integer -> a
  toInteger :: a -> Integer

class (Num a, Ord a) => Real a where  
  toRational :: a -> Rational

class (Real a, Enum a) => Integral a where  
  quot, rem :: a -> a -> a
  div, mod :: a -> a -> a
  toInteger :: a -> Integer

class (Num a) => Fractional a where  
  (/) :: a -> a -> a
  fromRational :: Rational -> a

class (Fractional a) => Floating a where  
  exp, log, sqrt :: a -> a
  sin, cos, tan :: a -> a

class (Real a, Fractional a) => RealFloat a where  
  truncate, round :: (Integral b) => a -> b
  ceiling, floor :: (Integral b) => a -> b


-- numerical functions

even, odd :: (Integral a) => a -> Bool

even n = n `rem` 2 == 0
odd = not . even

-- monadic functions

sequence :: Monad m => [m a] -> m [a]
sequence = foldr mcons (return [])

where mcons p q = do x <- p; xs <- q; return (x:xs)


sequence_ :: Monad m => [m a] -> m ()
sequence_ xs = do sequence xs; return ()


-----------------------------------------------------

-- functions on Functions

id :: a -> a
id x = x

const :: a -> b -> a
const x_ = x

(,) :: (b -> c) -> (a -> b) -> a -> c
f , g = \x -> f (g x)

flip :: (a -> b) -> a -> b
flip f x y = f y x

($) :: (a -> b) -> a -> b
f $ x = f x


-----------------------------------------------------

-- functions on Bools

data Bool = False | True

(&&), (||) :: Bool -> Bool -> Bool

True && x = x

False && _ = False

True || _ = True

False || x = x

not :: Bool -> Bool

not True = False

not False = True

-----------------------------------------------------

-- functions on Maybe

data Maybe a = Nothing | Just a

isJust :: Maybe a -> Bool
isJust (Just a) = True

isJust Nothing = False

isNothing :: Maybe a -> Bool
isNothing = not . isJust

fromJust :: Maybe a -> a
fromJust (Just a) = a

maybeToList :: Maybe a -> [a]

maybeToList Nothing = []

maybeToList (Just a) = [a]
listToMaybe :: [a] -> Maybe a
listToMaybe [] = Nothing
listToMaybe [a] = Just a

-- a hidden goodie

instance Monad [] where
  return x = [x]
x >>= f = concat (map f xs)

-- functions on pairs
fst :: (a, b) -> a
fst (x, y) = x
snd :: (a, b) -> b
snd (x, y) = y
curry :: ((a, b) -> c) -> a -> b -> c
curry f x y = f (x, y)
uncurry :: (a -> b -> c) -> (a, b) -> c
uncurry f p = f (fst p) (snd p)

-- functions on lists
map :: (a -> b) -> [a] -> [b]
map f xs = [ f x | x <- xs ]

(+>) :: [a] -> [a] -> [a]
xs ++ ys = foldr (:+) ys xs

filter :: (a -> Bool) -> [a] -> [a]
filter p xs = [ x | x <- xs, p x ]

concat :: [[a]] -> a
concat xs = foldr (:+) [] xs

concatMap :: (a -> [b]) -> [a] -> [b]
concatMap f = concat . map f

head, last :: [a] -> a
head (x:_:xs) = x
last (_:xs) = last xs

head (x:_:xs) = Just x
head (_:xs) = Nothing

init xs = x : init xs
init (x:xs) = init xs

null :: [a] -> Bool
null [] = True
null _:_ = False

length :: [a] -> Int
length [] = 0
length (_:_:l) = 1 + length l

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

foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z [] = z
foldr f z (x:xs) = f x (foldr f z xs)

foldl :: (a -> b -> a) -> [b] -> a
foldl f z [] = z
foldl f z (x:xs) = foldl f (f z x) xs

iterate :: (a -> a) -> a -> [a]
iterate f x = x : iterate f (f x)

repeat :: a -> [a]
repeat x = xs where xs = x:xs

replicate :: Int -> a -> [a]
replicate n x = take n (repeat x)

cycle :: [a] -> [a]
cycle [] = error "Prelude.cycle: empty list"
cycle xs = xs' where xs' = xs++xs'

take, drop :: [Int] -> [a] -> [a]
take n | n <= 0 = []
take _ [] = []
take n (x:xs) = x : take (n-1) xs
drop n xs | n <= 0 = xs
drop _ [] = []
drop n (_:xs) = drop (n-1) xs

splitAt :: [a] -> [(a,[a])]
splitAt n xs = (take n xs, drop n xs)

takeWhile, dropWhile :: (a -> Bool) -> [a] -> [a]
takeWhile p [] = []
takeWhile p (x:xs) | p x = x : takeWhile p xs
  | otherwise = []
dropWhile p [] = []
dropWhile p xs@(x:xs') | p x = dropWhile p xs'
  | otherwise = xs
delete y (x:xs) = if x == y then xs else x : delete y xs

(\\) : Eq a => [a] -> [a]
(\\) = foldl (flip delete)

union : Eq a => [a] -> [a] -> [a]
union xs ys = xs ++ (ys \ ys)

intersperse : a -> [a] -> [a]
intersperse \ [1,2,3,4] = [1,0,2,0,3,0,4]

transpose : [[a]]->[a]
transpose [[1,2,3],[4,5,6]] = [[1,4],[2,5],[3,6]]

partition : (a -> Bool) -> [a] -> ([a],[a])
partition p xs = (filter p xs, filter (not . p) xs)

group : Eq a => [a] -> [[a]]
group "apaabbbbeee" = ["aa","p","aa","bb","ee"]

isPrefixOf, isSuffixOf : Eq a => [a] -> [a] -> Bool
isPrefixOf [] = True
isPrefixOf _ [] = False

isPrefixOf (x:xs) (y:ys) = x == y & & isPrefixOf xs ys

isSuffixOf x y = reverse x `isPrefixOf` reverse y

sort : (Ord a) => [a] -> [a]
sort = foldr insert []

insert : (Ord a) => a -> [a] -> [a]
insert x [] = [x]
insert x (y:xs) = if x <= y then x:y:xs else y:insert x xs

------------------------------------------------------------------

-- functions on Char

tolower, toUpper :: Char -> Char
-- toLower 'A' == 'a'
-- toUpper 'a' == 'A'

digitToInt :: Char -> Int
-- digitToInt '8' == 8

intToDigit :: Int -> Char
-- intToDigit 3 == '3'

ord :: Char -> Int
chr :: Int -> Char
Exam

1. Point-free notation
Rewrite the following two definitions into a point-free form (i.e., \( f = \ldots \), \( g = \ldots \)), using neither lambda-expressions nor list comprehensions nor enumeration nor \texttt{where} clause nor \texttt{let} clause:

\[
\begin{align*}
  f x y &= (3 - y) / x \\
  g x y &= [x z | z <- [1,3..y]]
\end{align*}
\]

2. Type derivation
Give the type of the following expressions:

(a) \((.)(::)\)
(b) \((::)(.)\)
(c) \(((()::)\))
(d) \((::())\)
(e) (Haskell swearing) \(\texttt{[[]\gg=]((\_\rightarrow[>=])})\)

3. Proving program properties
The \texttt{Functor} class is defined as follows:

```haskell
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

It is mandatory that all instances of \texttt{Functor} should obey:

\[
\begin{align*}
  \text{fmap id} &= \text{id} \\
  \text{fmap (p . q)} &= (\text{fmap p}) . (\text{fmap q})
\end{align*}
\]

Assume the following definition of lists as a functor instance:

```haskell
instance Functor [] where
  fmap g [] = []
  fmap g (x:xs) = g x : (fmap g xs)
```

Is this a correct definition of a functor instance? Why or why not? \textbf{Prove your claim.}
4. **Programming**

Give an example of a function with type

\((\texttt{[a]} \rightarrow \texttt{a} \rightarrow \texttt{b}) \rightarrow \texttt{[b]}\)

5. **Type classes**

Complete the following two instance declarations:

\[
\text{instance } (\texttt{Ord a}, \texttt{Ord b}) \Rightarrow \texttt{Ord (a,b)} \texttt{ where } \ldots \\
\text{instance } \texttt{Ord b} \Rightarrow \texttt{Ord [b]} \texttt{ where } \ldots
\]

where pairs and lists should be ordered lexicographically, like the words in dictionary.

6. **Monadic computations**

Given the following function:

\[
f x y = \text{do} \\
\text{a <- x} \\
\text{b <- y} \\
\text{return } (\texttt{a*b})
\]

(a) What is the type of \(f\)? (0.1)
(b) What is the value of \(f \ [1,2,3] \ [2,4,8]\)? (0.2)
(c) What is the value of \(f \ \texttt{(Just 5) Nothing}\)? (0.1)
(d) What is the type of expression \(\texttt{return 5}\)? (0.1)
(e) What is the value of expression \(\text{do} \ [1,2,3]; []; \ "abc"\)? (0.25)
(f) What is the value of expression \(\text{do} \ [1,2,3]; []\); \(\texttt{return \ "abc"}\)? (0.25)

**Good Luck!**