Exam

1. Write a definition of \( \text{scanr} \), first using recursion, and then using \( \text{foldr} \).
   - Do the same for \( \text{scanl} \) first using recursion then \( \text{foldl} \).
   Provide type signature for both.
   A scan function accumulates a value like a fold, but returns a list of all intermediate values. E.g.
   \[
   \text{scanr} (+) 0 [1,2,3] = [6,5,3,0] \\
   \text{scanl} (+) 0 [1,2,3] = [0,1,3,6]
   \]
   - Use a fold (which one?) to define
     \[
     \text{reverse} :: [a] \rightarrow [a]
     \]
     which returns a list with the elements in reverse order.

2. Explain the \textit{subclass} relation between type classes in Haskell. Give an example as well.

3. Given the following definitions:
   \[
   \text{type} \; \text{ChurchNatural} \; a = (a \rightarrow a) \rightarrow (a \rightarrow a)
   \]
   \[
   \text{zeroC}, \text{oneC}, \text{twoC} :: \text{ChurchNatural} \; a
   \]
   \[
   \text{zeroC} \; f = \text{id} \quad \text{-- zeroC = const id} \\
   \text{oneC} \; f = f \quad \text{-- oneC = id} \\
   \text{twoC} \; f = f.f
   \]
   \[
   \text{succC n} \; f = f.(n \; f)
   \]
   \[
   \text{threeC} = \text{succC} \; \text{twoC} \\
   \text{fourC} = \text{succC} \; \text{threeC}
   \]
   \[
   \text{plusC} \; x \; y \; f = (x \; f).(y \; f)
   \]
   show that
   \[
   \text{twoC} \; \text{\textquote{plusC}} \; \text{twoC} = \text{fourC}
   \]
4. Given the following function:

```haskell
f x y = do
  a <- x
  b <- y
  return (a*b)
```

What is the type of `f`?
What is the value of `f [1,2,3] [2,4,8]`?
What is the value of `f (Just 5) Nothing`?
Is the expression `fmap (+2) (Just 5)` correct?
What is the type of expression `return 5`?

5. Explain the differences in behaviour of the following three functions:

```haskell
parSort2 (x:xs) = grtr `par` (lesser `par` (lesser ++ x:grtr))
  where lesser = parSort2 [y | y <- xs, y < x]
  grtr = parSort2 [y | y <- xs, y >= x]
parSort2 _ = []

parSort3 (x:xs) = grtr `par` (lesser 'pseq' (lesser ++ x:grtr))
  where lesser = parSort3 [y | y <- xs, y < x]
  grtr = parSort3 [y | y <- xs, y >= x]
parSort3 _ = []

parSort4 (x:xs) = force grtr `par` (force lesser 'pseq' (lesser ++ x:grtr))
  where lesser = parSort4 [y | y <- xs, y < x]
  grtr = parSort4 [y | y <- xs, y >= x]
parSort4 _ = []
```

6. In Haskore music is represented by the data type

```haskell
data Music = Note Pitch Dur [NoteAttribute] -- a note \ atomic
  | Rest Dur -- a rest / objects
  | Music :+: Music -- sequential composition
  | Music :=: Music -- parallel composition
  | ...
```

There is also a function which combines notes to lines of music:

```haskell
line = foldr (:+:) (Rest 0) :: [Music] -> Music
```

as well as the reverse function

```haskell
lineToList :: Music -> [Music]
lineToList n@(Rest 0) = []
lineToList (n :+: ns) = n : lineToList ns
```
(a) Let \( m_1 \) and \( m_2 \) be defined as
\[
m_1 = \text{Note (C,5) dur } \square \, :+: \, \text{Note (D,5) dur } \square, \, \text{Note (E,5) dur } \square
\]
\[
m_2 = \text{Note (C,5) dur } \square, \, \text{Note (D,5) dur } \square, \, \text{Note (E,5) dur } \square
\]

Explain how the values of expressions \( m_1 \) and \( m_2 \) differ.

(b) Define a function \( \text{line2} \) so that \( \text{line2} \, m_1 \) and \( \text{line2} \, m_2 \) is the same as \( m_2 \).

(c) Define a function \( \text{lineToList2} \) so that \( \text{lineToList2} \, (\text{line} \, m_1) \) and \( \text{lineToList2} \, (\text{line} \, m_2) \) is the same as \( \text{lineToList} \, (\text{line} \, m_2) \).

Good Luck! Lycka till!
A list of selected functions from the Haskell modules:

- Prelude
- Data.List
- Data.Maybe
- Data.Char

---

-- standard type classes

**class Eq a where**
**show :: a -> String**

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

**class (Eq a) => Integral a where**
**(+), (-), (*) :: a -> a -> a**
**max, min :: a -> a -> a**
**abs, signum :: a -> a**
**div, mod :: a -> a -> a**
**quot, rem :: a -> a -> a**

**class (Num a, Ord a) => Real a where**
**toInteger :: a -> Integer -> a**

**class (Real a, Enum a) => Fractional a where**
**(/) :: a -> a -> a**
**fromRational :: Rational -> a**

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

**class (Real a, Fractional a) => RealFrac a where**
**trunc, 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**

**sequence :: Monad m => [m a] -> m [a]**
**sequence xs = do x <- p; xs <- q; return (x:xs)**

---

-- monadic functions

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

---

-- functions on Booleans

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]
xs >>= 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 f x y = f (x, y)
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:_) = x
last [x] = x
last (_:xs) = last xs
tail, init :: [a] -> [a]
tail (_:xs) = xs
init [x] = []
init (x:xs) = x : init xs

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

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

takeWhile :: [a] -> [a] -> [a]
takeWhile p xs = xs |
  otherwise = []
takeWhile p xss |
  p x = xss |
  otherwise = []

dropWhile :: [a] -> [a] -> [a]
dropWhile p [] = []
dropWhile p xs@(x:xs') |
  p x = xs |
  otherwise = dropWhile p xs'

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

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

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

length l = |
take n xs = x : take (n-1) xs
drop n xs = xs |
take _ [] = []
take _ [x] = [x]
take n (x:xs) = x : take (n-1) xs

null (x:xs) = |
foldl f z [] = z
foldl f z (x:xs) = foldl f (f z x) xs

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

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

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

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

length l = |
length [] = 0
length (a:_ ) = 1 + length l

null [] = True
null (_:_ ) = False

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

takeWhile :: [a] -> [a] -> [a]
takeWhile p xs = xs |
  otherwise = []
takeWhile p xss |
  p x = xss |
  otherwise = []

dropWhile :: [a] -> [a] -> [a]
dropWhile p [] = []
dropWhile p xs@(x:xs') |
  p x = xs |
  otherwise = dropWhile p xs'

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

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

takeWhile :: [a] -> [a] -> [a]
takeWhile p xs = xs |
  otherwise = []
takeWhile p xss |
  p x = xss |
  otherwise = []

dropWhile :: [a] -> [a] -> [a]
dropWhile p [] = []

delete y (xs) = if x == y then xs else x : delete y xs

\( (\\) \quad \text{Eq a} \Rightarrow [a] \rightarrow [a] \) 
\( (\\) \quad = \text{foldl} (\text{flip delete}) \)

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

intersect :: Eq a => [a] -> [a] -> [a]
intersect xs ys = [x | x \&\& xs, x \text{ `elem` } ys ]

intersperse :: a -> [a] -> [a]
intersperse \( \emptyset \) \( [1,2,3,4] \) :: \([1,0,2,0,3,0,4] \)

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 "aaaabbbbxxx" :: ["aa","a","aa","b","b","b","x","x","x"]
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 x" reverse y

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

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

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

-- functions on Char

type String = [Char]
tolower, toUpper :: Char -> Char
tolower 'A' = 'a'
toUpper 'Z' = 'Z'
digitToChar :: Char -> Int
digitToChar '8' = 8

toIntDigit :: Int -> Char
toIntDigit 3 = '3'
          1 = '1'
          0 = '0'

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