Lund University
Department of Computer Science

EDAN40: Functional Programming
16th December 2013, 8.00–13.00

# Exam

1. 
   - Write a definition of `scanr`, first using recursion, and then using `foldr`.
   - Do the same for `scanl` first using recursion then `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.

   ```
   scanr (+) 0 [1,2,3] = [6,5,3,0]
   scanl (+) 0 [1,2,3] = [0,1,3,6]
   ```

   - Use a fold (which one?) to define

     ```
     reverse :: [a] -> [a]
     ```

     which returns a list with the elements in reverse order.

2. Explain the *subclass* relation between type classes in Haskell. Give an example as well.

3. Given the following definitions:

   ```
   type ChurchNatural a = (a -> a) -> (a -> a)

   zeroC, oneC, twoC :: ChurchNatural a
   zeroC f = id         -- zeroC = const id
   oneC  f = f          -- oneC  = id
   twoC  f = f.f

   succC n f =  f.(n f)

   threeC    = succC twoC
   fourC     = succC threeC

   plusC x y f   = (x f).(y f)
   ```

   show that

   ```
   twoC `plusC` twoC = fourC
   ```

4. Given the following function:

```
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:

```
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

```
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:

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

as well as the reverse function

```
lineToList :: Music -> [Music]
lineToList n@(Rest 0) = []
lineToList (n :+: ns) = n : lineToList ns
```

(a) Let `m1` och `m2` be defined as

```
m1 = [Note (C,5) dur [] :+: Note (D,5) dur [], Note (E,5) dur []]
m2 = [Note (C,5) dur [], Note (D,5) dur [], Note (E,5) dur []]
```

Explain how the values of expressions `line m1` and `line m2` differ.

(b) Define a function `line2` so that `line2 m1` and `line2 m2` is the same as `line m2`.

(c) Define a function `lineToList2` so that `lineToList2 (line m1)` and `lineToList2 (line m2)` is the same as `lineToList (line m2)`.

Good Luck! Lycka till!

```haskell
{-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 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

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) => RealFrac 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 -> c) -> b -> a -> c
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]
  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       :: ((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 xss       = foldr (++) [] xss

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 (_:l)     = 1 + length l

(!!)             :: [a] -> Int -> a
(x:_)  !! 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) -> 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              :: Int -> [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
```

```haskell
lines, words         :: String -> [String]
-- lines "apa\nbepa\ncepa\n" == ["apa","bepa","cepa"]
-- words "apa bepa\n cepa"   == ["apa","bepa","cepa"]

unlines, unwords     :: [String] -> String
-- unlines ["apa","bepa","cepa"] == "apa\nbepa\ncepa"
-- unwords ["apa","bepa","cepa"] == "apa bepa cepa"

and, or              :: [Bool] -> Bool
and                  = foldr (&&) True
or                   = foldr (||) False

any, all             :: (a -> Bool) -> [a] -> Bool
any p            = or . map p
all p            = and . map p

elem, notElem    :: (Eq a) => a -> [a] -> Bool
elem x           = any (== x)
notElem x        = all (/= x)

lookup           :: (Eq a) => a -> [(a,b)] -> Maybe b
lookup key []    = Nothing
lookup key ((x,y):xys)
    | key == x   = Just y
    | otherwise  = lookup key xys

sum, product     :: (Num a) => [a] -> a
sum              = foldl (+) 0
product          = foldl (*) 1

maximum, minimum :: (Ord a) => [a] -> a
maximum []       = error "Prelude.maximum: empty list"
maximum xs       = foldl1 max xs

minimum []       = error "Prelude.minimum: empty list"
minimum xs       = foldl1 min xs

zip              :: [a] -> [b] -> [(a,b)]
zip              = zipWith (,)

zipWith          :: (a->b->c) -> [a]->[b]->[c]
zipWith z (a:as) (b:bs)
                 = z a b : zipWith z as bs
zipWith _ _ _    = []

unzip            :: [(a,b)] -> ([a],[b])
unzip            = foldr (\(a,b) ~(as,bs) -> (a:as,b:bs)) ([],[])

nub              :: (Eq a) => [a] -> [a]
nub []           = []
nub (x:xs)       = x : nub [ y | y <- xs, x /= y ]

delete           :: Eq a => a -> [a] -> [a]
delete y []      = []
delete y (x:xs)  = if x == y then xs else x : delete y xs

(\\)             :: Eq a => [a] -> [a]-> [a]
(\\)             = foldl (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 `elem` ys ]

intersperse              :: a -> [a] -> [a]
-- intersperse 0 [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 "aapaabbbeee"   == ["aa","p","aa","bbb","eee"]

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

type String = [Char]

toUpper, toLower  :: Char -> Char
-- toUpper 'a'    == 'A'
-- toLower 'Z'    == 'z'

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

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

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