Re-exam – Introduction to Functional Programming

TDA555/DIT441, HT-25
Chalmers and Göteborgs Universitet, CSE

Day: 2026-08-18, *Time:* 14:00-18:00, *Place:* Johanneberg

Course responsible

Alex Gerdes (072-9744966). He will visit the exam room once around 15:30, and is after that available by phone.

Allowed aids

An English dictionary.

Grading

The exam consist of two parts: a part with seven small assignments and a part with two more advanced assignments; there are in total nine assignments.

- To pass the exam (with a 3) you need to give good enough answers for five out of the nine assignments. An answer with minor mistakes might be accepted, but this is at the discretion of the marker. An answer with large mistakes will be marked as incorrect.
- You do not need to solve the assignments from part II to pass the exam and you are happy with a 3! You are though encouraged to try the assignments from part II: they count to pass the exam, and you may get a higher grade.
- For a 4 you need to pass Part I (five out of seven assignments) and one assignment of your choice from Part II.
- For a 5 you need to pass Part I (five out of seven assignments) and both assignments from Part II.

Notes

- Begin each assignment on a new sheet and write your anonymous code on it.
- You may write your answers in Swedish and English.
- Excessively complicated answers might be rejected.
- *Write legibly!* Solutions that are difficult to read are marked as incorrect!
- You can make use of the standard Haskell functions and types given in the attached list (you have to implement other functions yourself if you want to use them). You do not have to import standard modules in your solutions. You do not have to copy any of the code provided.
- **Good luck!**

Part I

1

Given the following definition:

```
fun []     = ""
fun [x]    = show x
fun (x:xs) = show x ++ "," ++ fun xs
```

a) What does the expression

```
fun [1,2]
```

evaluate to? Use equational reasoning and write down the intermediate steps of the evaluation. You may choose any evaluation order.

b) What is the (most general) type[1] of `fun`?

Solution

```
{-
  fun  [1,2]
= { def fun, third case }
  show 1 ++ "," ++ fun [2]
= { apply show }
  "1" ++ "," ++ fun [2]
= { def fun, second case }
  "1" ++ "," ++ show 2
= { apply show }
  "1" ++ "," ++ "2"
= { apply first (++) }
  "1," ++ "2"
= { apply (++) }
  "1,2"
-}

fun :: Show a => [a] -> String  -- also accepted: [Int] -> String
```

[1] The type of a function is also called a *type signature*.

2

Implement the function:

```
indexWith :: [a] -> [Int] -> [a]
indexWith xs ixs = [x | Just x <- map (get xs) ixs]

-- Alternative:
indexWith' xs = mapMaybe (get xs)

get :: [a] -> Int -> Maybe a
get xs i | i >= 0 && i < length xs = Just (xs !! i)
         | otherwise               = Nothing
```

The function `indexWith` takes a list of values and a list of zero-based indices. For each valid index, it returns the element at that position in the first list. The order and repetition of indices are preserved. Indices outside the bounds of the list are ignored.

Hint: The `(!!)` operator crashes when given an invalid index. It may be useful to define an help function that safely selects an element from a list. The `Maybe` data type may come in handy.

For example:

```
ghci> indexWith ['a','b','c','d'] [2,0,2]
"cac"

ghci> indexWith [10,20,30] [1,5,-1,0]
[20,10]

ghci> indexWith [] [0,1]
[]
```

3

A package delivery service contains a collection of shipments. Each shipment has a tracking code, a sender, a package, and a delivery status.

A package is modelled by the recipient's name, its weight in grams, and whether or not it is fragile. The delivery status must be one of the following: registered, in transit, delivered, or failed. It should not be possible to assign any other status.

Your task is to define a collection of data types that models the package delivery service according to the above description.

Suggested solution

Note, you don't need to use record syntax, nor have deriving `Show`.

```
data DeliveryService = DeliveryService [Shipment] deriving Show

type Name = String

data Shipment = Shipment
  { trackingCode :: String
  , sender       :: Name
  , parcel       :: Package
  , status       :: DeliveryStatus
  } deriving Show

data Package = Package
  { recipient :: Name
  , weight    :: Int
  , fragile   :: Bool
  } deriving Show

data DeliveryStatus = Registered | InTransit | Delivered | Failed deriving Show
```

4

Now that we have a representation for a package delivery service (see the previous assignment), we would like to help its employees by writing some useful code. You are going to define the following function:

```
makePackage :: IO (String, Int, Bool)
```

The function `makePackage` should read the following input from the user: the recipient's name, the parcel's weight in grams, and whether or not the package is fragile. The last value should be entered as a Boolean value: `True` if the package is fragile, and `False` otherwise.

For example:

```
ghci> makePackage
Recipient:
> Alex Gerdes
Weight:
> 1500
Fragile:
> True
("Alex Gerdes",1500,True)
```

Note that GHCi automatically prints the result of the action `("Alex Gerdes",1500,True)`. Your solution should not print this explicitly; it should return the tuple as the result of the `IO` action.

Suggested solution

```
makePackage = do
  putStr "Recipient:\n> "
  recipient <- getLine
  putStr "Weight:\n> "
  weight    <- readLn
  putStr "Fragile:\n> "
  fragile   <- readLn
  return (recipient, weight, fragile)
```

5

Which of the following properties of the function `map` are true (for any functions `f`, `p` and lists `xs`)?

Select alternatives:

- ☐ `map f (reverse xs) == reverse (map f xs)`
- ☐ `map f (head xs) == head (map f xs)`
- ☐ `map f (tail xs) == tail (map f xs)`
- ☐ `map f (concat xs) == concat (map f xs)`
- ☐ `map f (filter p xs) == filter p (map f xs)`

You must select all values that are valid (if any). Selecting even one invalid value will cause you to fail this question.

Suggested solution

The first and third alternatives are correct.

```
f :: Int -> Int
f x = x * 3
p :: Int -> Bool
p x = x < 10

p1 xs = map f (reverse xs) == reverse (map f xs)
-- p2 xs = map f (head xs) == head (map f xs)  -- Type error
p3 xs = xs /= [] ==> map f (tail xs) == tail (map f xs)
-- p4 xs = map f (concat xs) == concat (map f xs)  -- Type error
p5 xs = map f (filter p xs) == filter p (map f xs)
```

6

In an earlier assignment, we introduced the function `indexWith`, which selects elements from a list using a list of indices. We now want to ensure that this function behaves as expected by testing it with QuickCheck. You can complete this assignment even if you have not solved the earlier one.

Your task is to write a QuickCheck property that verifies that the length of the list returned by `indexWith` is equal to the number of valid indices in the list of indices. An index i is valid for a list l if $0 \leq i < \text{length}(l)$.

```
prop_length :: [Int] -> [Int] -> Bool
prop_length xs ixs =
  length (xs `indexWith` ixs) == length [i | i <- ixs, i >= 0, i < length xs]
```

7

Define a higher-order function that conditionally transforms the elements of a list:

```
mapWhen :: (a -> Bool) -> (a -> a) -> [a] -> [a]
mapWhen p f = map (\x -> if p x then f x else x)
```

The function `mapWhen` takes a predicate, a function, and a list as arguments. The function should be applied only to elements for which the predicate returns `True`. All other elements should remain unchanged, and the order of the elements should be preserved.

For example:

```
ghci> mapWhen even (*10) [1,2,3,4,5]
[1,20,3,40,5]

ghci> mapWhen odd negate [1,2,3,4]
[-1,2,-3,4]
```

Part II

8

A robot moves on an infinite two-dimensional grid. Its position is represented by an x-coordinate and a y-coordinate. The robot also faces one of four directions:

```
data Direction = North | East | South | West deriving Show

data Robot = Robot Int Int Direction deriving Show
```

For example, `Robot 2 3 East` represents a robot at position $(2, 3)$, facing east. The robot can execute the following commands:

```
data Command
  = Forward Int
  | TurnTo Direction
  | Repeat Int [Command]
  deriving Show
```

The command `Forward n` moves the robot n units in its current direction. Moving north increases the y-coordinate, moving south decreases it, moving east increases the x-coordinate, and moving west decreases it.

The command `TurnTo direction` changes the direction in which the robot faces without changing its position. The command `Repeat n commands` executes the given list of commands n times. Repeated command blocks may themselves contain `Repeat` commands. You may assume that all distances and repetition counts are non-negative.

Your task is to define the following functions:

```
move :: Int -> Robot -> Robot

execute :: Robot -> [Command] -> Robot
```

The function `move` moves a robot forward without changing its direction. The function `execute` executes the commands from left to right. You may define additional help functions.

Suggested solution

```
move n (Robot x y direction) = case direction of
  North -> Robot x       (y+n) direction
  East  -> Robot (x+n)   y     direction
  South -> Robot x       (y-n) direction
  West  -> Robot (x-n)   y     direction

execute = foldl executeOne
 where
  executeOne robot@(Robot x y _) command = case command of
    Forward n        -> move n robot
```

```
    TurnTo direction -> Robot x y direction
    Repeat n cmds    -> execute robot $ concat $ replicate n cmds
```

Here is an initial robot along with some examples:

```
initial = Robot 0 0 North

ghci> move 3 initial
Robot 0 3 North

ghci> execute initial [Forward 2, TurnTo East, Forward 3]
Robot 3 2 East

ghci> execute initial [TurnTo West, Repeat 4 [Forward 1]]
Robot (-4) 0 West

ghci> execute initial [Forward 2, TurnTo East, Forward 3, Repeat 2 [TurnTo
 ↪  North, Forward 1]]
Robot 3 4 North
```

9

Consider the following alternative data type definition for trees:

```
data RoseTree a = Node a [RoseTree a] deriving Show
```

These are also called *rose trees*. Instead of having exactly two subtrees, as in binary trees, a rose tree node contains a list of subtrees. Note that there is no constructor for an empty tree, so every rose tree always contains at least one element.

Here is a smart constructor for creating a leaf node, along with an example rose tree:

```
leaf :: a -> RoseTree a
leaf x = Node x []

rt :: RoseTree Int
rt = Node 1 [Node 2 [leaf 4, leaf 5], Node 3 [leaf 6, leaf 7, leaf 8, leaf 9]]
```

Your tasks are:

a) Define a function that returns all values at a given depth in a rose tree:

```
atDepth :: Int -> RoseTree a -> [a]
atDepth n _ | n < 0   = []
atDepth 0 (Node x _)  = [x]
atDepth n (Node _ ts) = concatMap (atDepth (n-1)) ts
```

The root has depth zero, its children have depth one, and so on. Values should occur in the result from left to right. The function should return an empty list when the given depth is negative or exceeds the depth of the tree. For example:

```
ghci> atDepth 0 rt
[1]

ghci> atDepth 1 rt
[2,3]

ghci> atDepth 2 rt
[4,5,6,7,8,9]

ghci> atDepth 3 rt
[]
```

b) Define a function that groups the values of a rose tree by depth:

```
levels :: RoseTree a -> [[a]]
levels tree = takeWhile (not . null) [atDepth n tree | n <- [0..]]
```

The first list in the result should contain the root value, the second list should contain the values at depth one, and so on. Values at each depth should occur from left to right, and the result should not contain trailing empty lists. You may use `atDepth` from part a) in your solution.

For example:

```
ghci> levels rt
[[1],[2,3],[4,5,6,7,8,9]]

ghci> levels (Node 1 [Node 2 [leaf 3], leaf 4])
[[1],[2,4],[3]]
```

```haskell
{- Some standard functions from Prelude Data.List
   Data.Maybe Data.Char Control.Monad -}
-----------------------------------------------
class Show a where
  show :: a -> String

class Read a where
  read :: String -> a

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, 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,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, sin, cos, tan :: a -> a

class (Real a, Fractional a) => RealFrac a where
  truncate, round, 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]
  -- eval actions from left to right, return results
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
  -- apply action to every element in a list
replicateM :: Monad m => Int -> m a -> m [a]
  -- performs action n times, and return a list

-- 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 Bool -------------------------
(&&), (||)       :: 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 ---------------------------
isJust, isNothing      :: Maybe a -> Bool
isJust (Just a)        =  True
isJust Nothing         =  False
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

catMaybes              :: [Maybe a] -> [a]
catMaybes ls           = [x | Just x <- ls]

maybe :: b -> (a -> b) -> Maybe a -> b
maybe x f m = case m of Just y -> f y; Nothing -> x

mapMaybe :: (a -> Maybe b) -> [a] -> [b]

-- functions on pairs ----------------------------
fst              :: (a,b) -> a
fst (x,y)        =  x

snd              :: (a,b) -> b
snd (x,y)        =  y

swap             :: (a,b) -> (b,a)
swap (a,b)       = (b,a)

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            = foldr (const (1+)) 0

(!!)              :: [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 "cycle: empty list"
cycle xs          =  xs' where xs' = xs ++ xs'

tails             :: [a] -> [[a]]
tails xs          =  xs : case xs of
                            []      -> []
                            _ : xs' -> tails 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 (x:xs) | p x        = dropWhile p xs
                   | otherwise  = x:xs

span :: (a -> Bool) -> [a] -> ([a], [a])
span p as = (takeWhile p as, dropWhile p as)

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 ["ap","bep","cep"] == "ap\nbep\ncep"
-- unwords ["ap","bep","cep"] == "ap bep cep"
```

```
reverse           :: [a] -> [a]
reverse           =  foldl (flip (:)) []

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 (x:xs) = foldl max x xs

minimum [] = error "Prelude.minimum: empty list"
minimum (x:xs) = foldl min x 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]]
```

```haskell
group = groupBy (==)

groupBy :: (a -> a -> Bool) -> [a] -> [[a]]
groupBy _  []     =  []
groupBy eq (x:xs) =  (x:ys) : groupBy eq zs
             where (ys,zs) = span (eq x) xs

isPrefixOf :: Eq a => [a] -> [a] -> Bool
isPrefixOf []     _      = True
isPrefixOf _      []     = False
isPrefixOf (x:xs) (y:ys) = x==y && isPrefixOf xs ys

isSuffixOf :: Eq a => [a] -> [a] -> Bool
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]

isSpace, isDigit, isAlpha :: Char -> Bool
toUpper, toLower :: Char -> Char

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

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

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

-- functions from Test.QuickCheck ---------------
arbitrary :: Arbitrary a => Gen a
-- generator used by quickCheck

choose :: Random a => (a, a) -> Gen a
-- a random element in the given inclusive range

oneof :: [Gen a] -> Gen a
-- Randomly uses one of the given generators
frequency :: [(Int, Gen a)] -> Gen a
-- Chooses from weighted list of generators

elements :: [a] -> Gen a
-- Generates one of the given values.

listOf :: Gen a -> Gen [a]
-- Generates a list of random length.
vectorOf :: Int -> Gen a -> Gen [a]
-- Generates a list of the given length.

sized :: (Int -> Gen a) -> Gen a
-- construct generators that depend a size param

-- IO functions ---------------------------------
putStr, putStrLn :: String -> IO ()
getLine :: IO String
readLn :: Read a => IO a

type FilePath = String
readFile :: FilePath -> IO String
writeFile :: FilePath -> String -> IO ()
```