Re-exam – Introduction to Functional Programming

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

Day: 2026-01-08, *Time:* 8:30-12:30, *Place:* Johanneberg

Course responsible

Alex Gerdes (072-9744966). He will visit the exam room once around 9:30, 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 True  (x:xs) = x : fun False xs
fun False (x:xs) = fun True xs
fun _     _      = []
```

a) What does the expression

```
fun False [1,2,3,4]
```

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 False [1,2,3,4]
= { def fun, second case }
  fun True [2,3,4]
= { def fun, first case }
  2 : fun False [3,4]
= { def fun, second case }
  2 : fun True [4]
= { def fun, first case }
  2 : 4 : fun False []
= { def fun, thrid case }
  2 : 4 : []
-}

fun :: Bool -> [a] -> [a]  -- also accepted: Bool -> [Int] ->  [Int]
```

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

2

Implement the function:

```haskell
unweave :: [a] -> ([a], [a])
unweave (x:y:zs) = let (xs, ys) = unweave zs in (x:xs, y:ys)
unweave xs       = (xs, [])

-- alternative
unweave' xs = let ix p = [x | (i, x) <- zip [0..] xs, p i] in (ix even, ix odd)
```

which splits a list into two lists by alternating elements. The function takes a list and returns a pair of lists. The first list contains the elements at even positions (indexes, starting from 0), and the second list contains the elements at odd positions. In other words, unweave is the inverse of an interleave function that alternates elements from two lists.

Behaviour:

- The first element of the input list goes into the first output list.
- The second element goes into the second output list.
- This pattern continues for the entire list.
- If the input list has an odd length, the first output list will have one more element than the second.

Examples:

```
ghci> unweave [1,2,3,4,5,6]
([1,3,5],[2,4,6])

ghci> unweave "alexgerdes"
("aegre","lxeds")

ghci> unweave [1,2,3]
([1,3],[2])

ghci> unweave [1]
([1],[])

ghci> unweave []
([],[])
```

3

A *learning management system* (like Canvas) contains a collection of courses. Each course has a name, a list of teachers, a list of enrolled students, and a course level, which can be Introductory, Intermediate, or Advanced. The course level must be one of these values; it should not be possible to assign any other level.

Each student is modelled by the student's name, age and whether or not she/he likes functional programming. Each teacher consists of a name and can be one of the following: Examiner or Teaching Assistant.

Your task is to define a collection of data types that models the learning management system according to the above description.

Suggested solution

Note, you don't need to use record syntax (which the suggested solution uses), nor have deriving `Show`.

```
data LMS = LMS [Course]

data Course = Course
  { name     :: String
  , teachers :: [Teacher]
  , students :: [Student]
  , level    :: Level
  } deriving Show

data Level = Introductory | Intermediate | Advanced deriving Show

type Name = String

data Teacher = Examiner Name | TA Name deriving Show

data Student = Student
  { sname   :: Name
  , age     :: Int
  , likesFP :: Bool
  } deriving Show
```

4

Now that we have a representation for a learning management system (see the previous assignment), we would like to help our administrators by writing some useful code. You are going to define the following function:

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

The function `makeStudent` should read the following input from the user: the student's name, the student's age, and whether or not the student likes Functional Programming (FP). The preference should be entered as a Boolean value: `True` if the student likes FP, and `False` otherwise.

For example:

```
ghci> makeStudent
Name:
> Alex Gerdes
Age:
> 47
Likes FP:
> True
("Alex Gerdes",47,True)
```

Note that GHCi automatically prints the result of the action `("Alex Gerdes",47,True)`. Your solution should not print this explicitly; it should return the tuple as the result of the `IO` action.

Suggested solution

```
makeStudent = do
  putStr "Name:\n> "
  name  <- getLine
  putStr "Age:\n> "
  age   <- readLn
  putStr "Likes FP:\n> "
  likes <- readLn
  return (name, age, likes)
```

5

Which of the following are values of data type:

```
data T = A Int | B (Maybe T)
```

Select one or more alternatives:

- ☐ `B (Just (B (A 3)))`
- ☐ `B (Just Nothing)`
- ☐ `B (Just 5)`
- ☐ `B (Just (A 5))`
- ☐ `B (Just (B Nothing))`

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 last two options are valid values.

6

In an earlier assignment, we introduced the function `unweave`, which splits a list into two lists by alternating elements. 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 sum of the lengths of the two lists returned by `unweave` is equal to the length of the input list.

```haskell
prop_length :: [Int] -> Bool
prop_length xs = let (ys, zs) = unweave xs in length xs == length ys + length zs
```

7

Consider the following recursive data type:

```
data Steps = Stop | GoLeft Int Steps | GoRight Int Steps deriving Show
```

Define a higher-order function that applies a given function to all integer values stored in the constructors of a value of type `Steps`:

```
mapSteps :: (Int -> Int) -> Steps -> Steps
mapSteps f steps = case steps of
  Stop           -> Stop
  GoLeft  n next -> GoLeft  (f n) (mapSteps f next)
  GoRight n next -> GoRight (f n) (mapSteps f next)
```

The function should preserve the structure of the `Steps` value, modifying only the integer components. For example:

```
ghci> mapSteps (+1) (GoLeft 2 (GoRight 3 Stop))
GoLeft 3 (GoRight 4 Stop)

ghci> mapSteps (\ x -> 42) (GoRight 3 (GoLeft 2 (GoRight 1 Stop)))
GoRight 42 (GoLeft 42 (GoRight 42 Stop))
```

Part II

8

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 the leaves of a rose tree:

```
leaves :: RoseTree a -> [a]
leaves (Node x [])     = [x]
leaves (Node _ forest) = concatMap leaves forest
```

A leaf is a tree without any subtrees. For example:

```
ghci> leaves rt
[4,5,6,7,8,9]
```

b) Write a function:

```
paths :: RoseTree a -> [[a]]
paths (Node x ts) = case ts of
  [] -> [[x]]
  _  -> map (x:) (concatMap paths ts)
```

that returns all root-to-leaf paths. A path is a sequence of node values starting at the root and following parent–child links down to a leaf. For example, if the root is 1 and it has a child 2 which has leaf child 5, then `[1,2,5]` is one path.

For example:

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

9

A *set* is a collection of elements, for example a set of natural numbers $\mathbb{N}$ or the set of card suits $\{\clubsuit, \blacklozenge\}$. The order of elements in a set does not matter; for instance, $\{1, 2\}$ and $\{2, 1\}$ represent the same set.

One of the most fundamental operations on a set is to determine whether an element belongs to the set. In set theory this is written using the symbol $\in$, for example $4 \in \mathbb{N}$. Instead of representing a set using a list of elements or a binary search tree, we will model a set as a *function*. Specifically, we use the following type synonym:

```
type Set a = a -> Bool
```

A value of type `Set a` is a function that takes an element and returns `True` if the element belongs to the set, and `False` otherwise.

Using this representation, common set operations can be expressed in a concise and elegant way. For example:

```
singleton :: Eq a => a -> Set a
singleton x = \y -> x == y

member :: a -> Set a -> Bool
member x s = s x
```

The `singleton` function constructs a set containing exactly one element. Its implementation returns a function that checks whether its argument is equal to the given element. The `member` function checks whether a given value is an element of a set by applying the set function to that value.

Here are some examples:

```
ghci> member 2 (singleton 1)
False

ghci> member 'a' (singleton 'a')
True
```

Your task is to define the following set operations using this functional representation:

```
-- Create the empty set
empty :: Set a

-- Insert an element into a set
insert :: Eq a => a -> Set a -> Set a

-- The union of two sets: returns True for elements that belong
-- to either of the sets
union :: Set a -> Set a -> Set a

-- Set difference: returns a set containing all elements of the
-- first set that do not belong to the second set
```

```
difference :: Set a -> Set a -> Set a
```

9.1 Solutions

```
empty _ = False

insert x = union (singleton x)

union p q = \x -> p x || q x

difference p q = \x -> p x && not (q x)
```

Each function should behave according to the standard mathematical definition of the corresponding set operations. All operations can, and should, be implemented using a single line of code. If you find yourself needing multiple lines, you are likely not on the right track and may not have fully understood the chosen representation.

```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

-- 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
```

```haskell
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 (:)) []
```

```haskell
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]]
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 ()
```