Re-exam – Introduction to Functional Programming

TDA555/DIT441 (DIT440), HT-23
Chalmers and Göteborgs Universitet, CSE

Day: 2024-08-20, *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:

```
ack 0 n = n + 1
ack m 0 = ack (m - 1) 1
ack m n = ack (m - 1) (ack m (n - 1))
```

a) What does the expression

```
ack 1 1
```

evaluate to? Use equational reasoning and write down the intermediate steps of the evaluation. You should use an *outermost* evaluation strategy, that is, reduce the outermost reducible expression first. You may though reduce subtractions and additions whenever you want.

b) What is the type signature of `ack`?

Solution

```
{-
  ack 1 1
= { def ack, third case }
  ack (1 - 1) (ack 1 (1 - 1))
= { subtractions }
  ack 0 (ack 1 0)
= { def ack, first case on outermost ack call }
  (ack 1 0) + 1
= { def ack, second case }
  (ack (1 - 1) 1) + 1
= { subtraction }
  (ack 0 1) + 1
= { def ack, first case }
  (1 + 1) + 1
= { additions }
  3
-}

ack :: (Eq a, Num a) => a -> a -> a  -- also accepted: Int -> Int -> Int
```

2

Implement the function:

```
findIndex :: Eq a => a -> [a] -> Maybe Int
findIndex x xs = lookup x (zip xs [0..])

-- Alternative using explicit recursion
findIndex' x = go 0
 where
  go _ []       = Nothing
  go i (y:ys)
    | x == y    = Just i
    | otherwise = go (i+1) ys
```

that given an element `x` and a list `xs`, finds out at what position `x` occurs in the list. We start counting positions at 0. If there are multiple positions, the function `findIndex` always produces the first position `x` is at.

Examples:

```
ghci> findIndex 'a' "bepacepa"
Just 3

ghci> findIndex 11 [2,3,5,7,11,13]
Just 4

ghci> findIndex "hej" ["hej","hi","hola","hello","hoi"]
Just 0

ghci> findIndex 42 [1,2,3]
Nothing
```

The function uses the `Maybe` data type to indicate that the element `x` might not occur in the list. In this case, the `findIndex` should return `Nothing`, as the last example shows.

3

This assignment is about *modeling*. You are going to model a new kind of twitter in Haskell using data types. We will call it: Y! The name is an homage to the Y-combinator from the lambda calculus. Y is going to be the number one place for functional programmers to tweet[1] about functional programming, such as purity and higher-order functions.

Y consists of a list of users and a list of tweets. For every user we store their (real) name and a screen name (a nick name). A tweet consists of a user, who sent the tweet, a short message, and a date (which can be modeled as a `String`). In addition, a tweet stores the language in which the message of the tweet is written, which can be English, Swedish or Dutch.

Your task it to define a collection of data types that models Y 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 Y = Y
  { users  :: [User]
  , tweets :: [Tweet]
  }

data User = User
  { name       :: String
  , screenName :: String
  }

data Tweet = Tweet
  { by      :: User
  , message :: String
  , lang    :: Lang
  , date    :: Date
  }

type Date = String

data Lang = EN | SE | NL deriving Show
```

[1] A tweet is a short message, with a limited number of characters

4

Now that the creation of Y (see the previous assignment) has started we want to support the developers by writing some code. You are going to define a function:

```
readTweet :: IO (String, String, String)
```

that allows a user to compose a message (tweet). This `IO` function should ask a user for its name and the to be posted tweet. In addition, the function should read the current date using the following given function:

```
readDate :: IO String
```

You do *not* need to write this function! But you need to *use* it in your definition of the `readTweet` function. Note that we don't bother about the language of a tweet in this function.

The three parts (user name, message, and date) given by a user are then returned in a three-tuple. For example:

```
ghci> readTweet
What is your name?
> Alex
Please enter your message:
> Haskell rules!
("Alex","Haskell rules!","2024-08-20 11:05:26 CEST")
```

Suggested solution

```
readTweet = do
  putStr "What is your name?\n> "
  user <- getLine
  putStr "Please enter your tweet\n> "
  msg  <- getLine
  now  <- readDate
  return (user, msg, now)
```

5

Consider the following data type for the colors of the Dutch national flag:

```
data Color = Red | White | Blue deriving (Eq, Show)
```

Write a function that checks if a list of colors contains a sequence representing the Dutch national flag, which is red, white, and blue.

```
containsDutchFlag :: [Color] -> Bool
```

The sequence may contain repetitions of the same color, that is, the sequence should contain one or more red elements, directly followed by one or more white elements, directly followed by one or more blue elements. For example:

```
ghci> containsDutchFlag [White, Red, Red, White, White, White, Blue, Red]
True
```

The next example show a list that does not contain the Dutch national flag:

```
ghci> containsDutchFlag [Red, Red, White, Red, White, Red, Blue]
False
```

```
containsDutchFlag = go . map head . group
 where
  go [] = False
  go xs = take 3 xs == [Red, White, Blue] || go (tail xs)

-- Some alternatives:
containsDutchFlag' xs = (Red, White, Blue) `elem` combs
 where
   combs = zip3 ys (drop 1 ys) (drop 2 ys)  -- all running combinations
   ys    = map head (group xs)              -- remove duplicate adjacent colors

containsDutchFlag'' = red
 where
  red (Red:White:cs) = white cs
  red (_:cs)         = red cs
  red []             = False

  white (White:cs)   = white cs
  white (Blue:_)     = True
  white (c:cs)       = red (c:cs)
  white []           = False
```

6

In an earlier assignment we introduced the `findIndex` function, that returns the position of a given element in a given list. We want to make sure that this function works as expected by using QuickCheck. You can do this assignment even if you have not solved the earlier assignment.

Your tasks are:

a) Write a property that validates that if the `findIndex` returns an actual index, that the element at that index is indeed the given element we wanted to find:

```
prop_element :: Int -> [Int] -> Bool
prop_element x xs = case findIndex x xs of
  Just i  -> xs !! i == x
  Nothing -> x `notElem` xs
```

In case the `findIndex` function returns a `Nothing`, then you should validate that the given element is not in the list.

b) When using QuickCheck to validate the `prop_element` property it will generate random inputs for an element and a list. The probability that QuickCheck generates a random element that is an element of a random list is rather low.

How can we fix this? We want that the majority (but not all) of the generated tests validate the `findIndex` function with an element that is part of the input list.

You don't have to implement something, just explain in words how you would solve this issue.

One solution is to create an own generator:

```
genList :: Gen (Int, [Int])
genList = do
  xs <- listOf1 arbitrary
  x  <- elements xs
  frequency [ (10, return (x, xs))
            , ( 1, return (x, delete x xs))
            ]

prop_element' = forAll genList $ \(x, xs) ->
  classify (x `elem` xs) "Contains element" (prop_element x xs)
```

7

Consider the following data type for *lookup tables*:

```
data LookupTable k v = Empty | Insert k v (LookupTable k v) deriving Show
```

Such a data type is also known as a 'map'. A lookup table is either empty or we can add an entry using the `Insert` constructor. The data type is polymorphic and uses two type variables: `k` for the keys, and `v` for the values.

For example, we can create a phone book using the `LookupTable` data type:

```
phoneBook :: LookupTable String String
phoneBook = Insert "Alex" "033-7726154" $ Insert "Dave" "033-7721059" Empty
```

Your task is to define the higher-order function:

```
mapTable :: (v -> w) -> LookupTable k v -> LookupTable k w
mapTable f table = case table of
  Empty        -> Empty
  Insert k v t -> Insert k (f v) (mapTable f t)
```

that applies a given function (of type `v -> w`) to every *value* in a lookup table, the keys remain the same. We can, for instance, use the `mapTable` function to add a country code to all phone numbers in the example `phoneBook`:

```
ghci> mapTable (("+46-" ++) . tail) phoneBook
Insert "Alex" "+46-33-7726154" (Insert "Dave" "+46-33-7721059" Empty)
```

Part II

8

Consider the following alternative data type definition for lists:

```
data List a = Nil | Front a (List a) | Back (List a) a deriving (Eq, Show)
```

that can store a collection of elements of an arbitrary type. We also introduce two operators (*smart constructors*) along with their fixity and precedence:

```
(<|) :: a -> List a -> List a
x <| xs = Front x xs

infixr 5 <|

(|>) :: List a -> a -> List a
xs |> x = Back xs x

infixl 4 |>
```

This list definition allows you to add an element to the *front* of a list (just as ordinary lists from the `Prelude`), as well as an element to the *back* of a list. For example, the following list stores a collection of integers:

```
xs :: List Int
xs = 1 <| 2 <| Nil |> 3 |> 4
```

where an integer 2 is added to the front of an empty list, then 1 is added to the front of that list, followed by a 3 at the back, and finally a 4 is added to the back of the list. So, the list `xs` contains the following sequence: 1, 2, 3, 4. Note that we don't need to use any parentheses, because we have declared the fixity and precedence of the operators.

Your tasks are:

a) Write a function that converts a value of our list type to an ordinary list:

```
toList :: List a -> [a]
toList l = case l of
    Nil        -> []
    Front x xs -> x : toList xs
    Back  xs x -> toList xs ++ [x]
```

Note that the order of the elements must be preserved.

b) Define a function that *normalizes* a value of type `List`:

```
normalize :: List a -> List a
normalize = fromList . toList
 where
  fromList = foldr (<|) Nil
```

such that it no longer contains any `Back` constructors, but still has the same elements, in the same order. Calling the function `normalize` on the example list `xs` will result in:

```
ghci> normalize xs
Front 1 (Front 2 (Front 3 (Front 4 Nil)))
```

c) Write a function that folds a value of type `List a`:

```
foldList :: (a -> b -> b) -> b -> List a -> b
foldList f b = foldr f b . toList
```

9

Your task is to write a function:

```
sumSeqs :: Int -> [Int] -> [[Int]]
sumSeqs n xs = nub [ys | ys <- seqs xs, sum ys == n]
 where
  seqs []     = [[]]
  seqs (x:xs) = let xss = seqs xs in xss ++ map (x:) xss
```

that takes an integer (a sum) and a list of integers as input. The `sumSeqs` function calculates all possible combinations of integers from the list that add up to the given integer. For example:

```
ghci> sumSeqs 4 [1,1,2,2,3]
[[2,2],[1,3],[1,1,2]]
```

As you can see, the sum of all the combinations is 4. Here is another example:

```
ghci> sumSeqs 8 [1,2,3,4,5]
[[3,5],[1,3,4],[1,2,5]]
```

Note, that all combinations are unique, that is, we don't want duplicate combinations (or permutations of a combination). For instance, in the first example we can combine the first 1 with the three, and the second 1 with the three, but that leads to the same combination (`[1,3]` and `[1,3]`).

Hint: define a help function that returns all sub-sequences.

```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]

-- 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]         = []
```

```haskell
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"
```

```haskell
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 (,)

zip3              :: [a] -> [b] -> [c] -> [(a,b,c)]

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 ()
```