Exam – Introduction to Functional Programming

TDA555/DIT441 (DIT440), HT-24
Chalmers and Göteborgs Universitet, CSE

Day: 2024-10-29, *Time:* 14:00-18:00, *Place:* Johanneberg

Course responsible

Alex Gerdes (072-9744966). Daniël Apol will visit the exam room once around 15:00, and Alex 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 []     = 10
fun [x]    = x
fun (x:xs) = 1 + fun xs
```

a) What does the expression

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

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

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

Solution

```
{-
  fun [1,2,3]
= { def fun, third case }
  1 + fun [2,3]
= { def fun, third case }
  1 + 1 + fun [3]
= { addition }
  2 + fun [3]
= { def fun, second case }
  2 + 3
= { addition }
  5
-}

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

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

2

Implement the function:

```
interleave :: [a] -> [a] -> [a]
interleave xs     []     = xs
interleave []     ys     = ys
interleave (x:xs) (y:ys) = x : y : interleave xs ys
```

that *interleaves* two lists. The `interleave` function creates a new list where the elements of the two input lists are interleaved, that is, the first element from the first list is followed by the first element of the second list, then the second element of the first list, and so on. In other words, the elements of the two lists are alternated. If the lists are of unequal size, the remaining elements of the larger lists are just added to back of the resulting list.

Examples:

```
ghci> interleave "alex" "4321"
"a4l3e2x1"

ghci> interleave [1,2,3,4] [5,6,7]
[1,5,2,6,3,7,4]

ghci> interleave [1,2,3] []
[1,2,3]

ghci> interleave [] [4,5,6]
[4,5,6]
```

3

This assignment is about *modeling*. We have been using Discord in this course for discussions, queueing, announcements, and more. We are going to model an own group chat application in Haskell. We will call it: GHCHAT!

The group chat application GHCHAT consists of a server address (that can be modeled as a `String`), a list of users, and a list of channels. A user has a name and a particular role. An individual channel has a topic (a name for the channel) and a list of posts, where a post is a combination of a message (a `String`) and a user. Finally, a role can be an administrator, a member, or a bot.

Your task it to define a collection of data types that models GHCHAT 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`.

```haskell
data GhChat = GhChat
  { server   :: String
  , users    :: [User]
  , channels :: [Channel]
  } deriving Show

data User = User
  { name :: String
  , role :: Role
  } deriving Show

data Channel = Channel
  { topic   :: String
  , posts   :: [(String, User)]
  } deriving Show

data Role = Admin | Member | Bot deriving Show
```

4

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

```
readUser :: IO (String, String)
```

that reads the name and role of a user. This `IO` function should ask a user for its name and retrieves the user's role using the following given `IO` function:

```
getRole :: String -> IO String
getRole _ = return "Admin"
```

You do *not* need to write this function! But you need to *use* it in your definition of the `readUser` function. This function takes a user name as argument. So, after you have read the user name, you need to give it as an argument to the `getRole` function.

The `readUser` returns a tuple with the supplied name and retrieved role. For example:

```
ghci> readUser
Please give your user name:
> Alex
("Alex","Admin")
```

GHCi prints the result of the action `("Alex", "Admin")`, but your solution should not do this.

Suggested solution

```
readUser = do
  putStr "Please give your user name:\n> "
  name <- getLine
  role <- getRole name
  return (name, role)
```

5

Consider the following data type that can store either one or two elements of a particular type:

```
data Some a = One a | Two a a deriving (Eq, Show)
```

Write a function that converts a list of `Some`s to an ordinary list:

```
flatten :: [Some a] -> [a]
flatten somes = [x | some <- somes, x <- toList some]
 where
  toList (One x)   = [x]
  toList (Two x y) = [x, y]
```

The `flatten` function 'flattens' the list of `Some`s. The order of the elements in the original list should be preserved. *Hint*: write a simple help function that converts a `Some` to a list.

Examples:

```
ghci> flatten [One 1, Two 2 3, One 4, Two 5 6]
[1,2,3,4,5,6]

ghci> flatten [Two 'a' 'a', One 'l', Two 'e' 'e', One 'x']
"aaleex"

ghci> flatten []
[]
```

6

In an earlier assignment we introduced the `interleave` function, that interleaves two given lists. 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 task is to write a property that validates that the sum of two input integer lists is equal to the sum of the two interleaved lists:

```
prop_sum :: [Int] -> [Int] -> Bool
prop_sum xs ys = sum xs + sum ys == sum (interleave xs ys)
```

7

Consider the following data type for a *phone book*:

```
type Name      = String
type Number    = Int
data PhoneBook = Empty | Insert Name Number PhoneBook deriving Show
```

Such a data type is also known as a 'lookup table'. A phone book is either empty or we can add an entry using the `Insert` constructor. The `Insert` constructor takes a name and a number and adds it to another phone book. It is recursive data type.

For example, we can create a phone book as follows:

```
phoneBook :: PhoneBook
phoneBook = Insert "Alex" 6154 (Insert "Dave" 1059 Empty)
```

Your task is to define the higher-order function:

```
filterBook :: (Name -> Bool) -> PhoneBook -> PhoneBook
filterBook _ Empty = Empty
filterBook p (Insert name number book)
  | p name    = Insert name number (filterBook p book)
  | otherwise = filterBook p book
```

that takes a predicate (a function returning a `Bool`) and a phone book as argument, and returns a new phone book containing only the entries for which the predicate holds.

We can, for instance, use the `filterBook` function to extract all the persons who's name starts with an 'A' in the example `phoneBook`:

```
ghci> filterBook (isPrefixOf "A") phoneBook
Insert "Alex" 6154 Empty
```

Part II

8

Consider the familiar data type(s) for a simple expression language:

```
data Expr
  = Val Value
  -- Integer operations
  | Add Expr Expr
  | Mul Expr Expr
  -- Boolean operations
  | And Expr Expr
  | Or  Expr Expr
  -- If-then-else
  | If  Expr Expr Expr
  deriving Show

data Value = Num Int | Bool Bool deriving Show
```

We have moved the literal values to a separate data type, and it can be either an integer (`Num`) or a boolean (`Bool`) value. We have extended the expression data type with two logical operators (conjunction and disjunction) on boolean values, and we introduce an if-then-else constructor. The first field of the `If` constructor is the condition, the second field is the 'then' branch, and finally the third field is the 'else' branch.

In addition, we define some small help functions to create integer and boolean expression values:

```
int :: Int -> Expr
int n = Val (Num n)

true, false :: Expr
true  = Val (Bool True)
false = Val (Bool False)
```

Your task is to write an evaluation function for the `Expr` data type:

```
eval :: Expr -> Value
eval expr = case expr of
  Val v    -> v
  Add x y  -> num (+) x y
  Mul x y  -> num (*) x y
  And x y  -> bool (&&) x y
  Or  x y  -> bool (||) x y
  If c t e -> case eval c of
    Bool b -> eval (if b then t else e)
    _      -> error "Type error: the condition must be a boolean."

num :: (Int -> Int -> Int) -> Expr -> Expr -> Value
num op l r = case (eval l, eval r) of
```

```
    (Num x, Num y) -> Num (x `op` y)
    _              -> error "Type error: expected two integer values."

bool :: (Bool -> Bool -> Bool) -> Expr -> Expr -> Value
bool op l r = case (eval l, eval r) of
    (Bool x, Bool y) -> Bool (x `op` y)
    _                -> error "Type error: expected two boolean values"
```

Note that we can only evaluate the binary operators if they are applied to two values of the correct type. The `Add` and `Mul` operators can only be applied to integer values, and the `And` and `Or` operators to boolean values. We cannot, for example, add a boolean to an integer, or apply the logical 'or'-operator on two integers. In addition, the condition of the if-then-else construct should evaluate to a boolean. If any of the aforementioned restrictions are not met in an expression, then the types do not match and we cannot evaluate it. The evaluation function should report this with an error message by calling the `error` function with an appropriate error message.

Some examples:

```
ghci> eval (If true (Add (int 1) (int 2)) (int 42))
Num 3

ghci> eval (If (And true false) (Add (int 1) (int 2)) (int 42))
Num 42

ghci> eval (Add (int 10) false)
*** Exception: Type error: expected two integer values.

ghci> eval (Or true false)
Bool True
```

9

Consider the following alternative data type definition for lists:

```
data Tree a = Leaf a | Node (Tree a) a (Tree a) deriving Show
type List a = [(Tree a, Int)]
```

that can store a collection of elements of an arbitrary type. This particular list type stores the elements as a *list of trees*; each tree is paired with the size of the tree. This data structure can be used to implement random access lists, since it is possible to access every element in logarithmic time. We are not going to develop that in this question, but we are going to use the data structure and define some functions on it.

We also introduce a function that creates an empty list of type `List a`, and a function that adds an element to a list:

```
empty :: List a
empty = []

add :: a -> List a -> List a
x `add` ((l, n) : (r, m) : ts) | n == m = (Node l x r, 2 * n + 1) : ts
x `add` ts                              = (Leaf x, 1) : ts
```

The `add` function checks if the first two trees are of equal size, and if so, then it creates a new tree with the `Node` constructor. The node stores the to be added element, and the first two lists become the left and right subtrees of the node. This way we make sure we have always perfectly balanced trees, and it is a cheap function. Moreover, using the paired sizes, we exactly know in which tree to look when we index into the list. Cunning, don't you agree?

Here are some examples:

```
ghci> add 1 empty
[(Leaf 1,1)]

ghci> add 2 (add 1 empty)
[(Leaf 2,1),(Leaf 1,1)]

ghci> add 3 (add 2 (add 1 empty))
[(Node (Leaf 2) 3 (Leaf 1),3)]

ghci> add 4 (add 3 (add 2 (add 1 empty)))
[(Leaf 4,1),(Node (Leaf 2) 3 (Leaf 1),3)]
```

Your tasks are:

a) Write a function that counts the number of elements in a value of type `List a`:

```
size :: List a -> Int
size = sum . map snd
```

Hint: remember that every tree is paired with its size.

b) Write a function that converts an ordinary Haskell list to our new list type:

```
fromList :: [a] -> List a
fromList = foldr add empty
```

You must use the `empty` and `add` functions to implement the `fromList` function.

c) Define a function returns the head of a list of type `List a`:

```
hd :: List a -> a
hd trees = case fst (head trees) of
  Leaf x     -> x
  Node _ x _ -> x
```

You don't need to take care of an empty list.

```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 (,)

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])
```

```haskell
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 ()
```