Exam – Introduction to Functional Programming

TDA555/DIT441 (DIT440), HT-22
Chalmers och Göteborgs Universitet, CSE

Day: 2022-10-26, *Time:* 14:00-18:00, *Place:* Johanneberg

Course responsible

Alex Gerdes (031-772 6154). He will visit the exam room once between 15:30 and 16:00, and after that is 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 or a G) 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 or G! 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 number 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 definitions:

```
f x (y:z:zs) = y : x : f x (z:zs)
f _  xs      = xs

g x xs = concat (f x xs)
```

a) What does the expression

```
g " " ["intro", "func", "prog"]
```

evaluate to? Write down the intermediate steps of your computation. If the type of your answer is incorrect then your solution will be considered incorrect. Note that the `concat` function concatenates a list of lists into a single list.

b) What are the most general types (type signatures) of `f` and `g`?

2

Two natural numbers are *coprime* if the only common divisor is 1. In other words, there does not exist a number other than 1 that divides both numbers without leaving a remainder. For example, the pair of 4 and 9 are coprime. The number 4 has a factor 2, but 9 is not divisible by 2, and 4 is not divisible by 3, which is a factor of 9. The pair of 6 and 9, for example, are *not* coprime, since 3 divides both 6 and 9 without leaving a remainder.

To determine if a pair of numbers is coprime you should calculate their *greatest common divisor* (gcd) and check if this is equal to 1. The gcd can be calculated using the Euclidian algorithm, which works as follows: start with the pair of numbers (x, y) and repeatedly replace this by $(y, x \bmod y)$ until the pair is $(d, 0)$, where d is the greatest common divisor. For example:

$$\gcd(9,6) \to \gcd(6, 9 \bmod 6) \to \gcd(6,3) \to \gcd(3, 6 \bmod 3) \to \gcd(3,0)$$

so the gcd of 9 and 6 is 3. Note that the modulo operator (mod) returns the remainder of an integer division.

Your tasks are:

a) First, write a function that returns the greatest common divisor of two given integers:

```
gcd :: Int -> Int -> Int
```

b) Next, use the `gcd` function to implement the following function that checks if two numbers are coprime:

```
coprime :: Int -> Int -> Bool
```

3

Consider the following recursive datatype:

```
data DTree a
  = Decision a
  | Question String (DTree a) (DTree a)
  deriving (Eq, Show)
```

which models a *decision tree*. A decision tree can either be a `Decision`, where we store an item of type `a` denoting what to do when we reach that decision, or a `Question` where we store a question (as a string) together with two (sub)trees: one tree in case the answer to the question is positive, and a second tree if the answer is negative. For example, the following decision tree determines which mode of transport we should choose:

```
whatToDo :: DTree String
whatToDo = Question "Is it Raining?" yes no
 where
  yes = Decision "Take the bus"
  no  = Question "Is it more the 2km?" (Decision "Cycle") (Decision "Walk")
```

Another small example is to determine how much money to save:

```
howMuch :: DTree Int
howMuch = Question "Is the inflation above 2 percent?"
            (Question "Do you have a mortgage?" (Decision 1000) (Decision 400))
            (Decision 100)
```

Your task is to implement the higher-order function:

```
mapDecision :: (a -> b) -> DTree a -> DTree b
```

that applies a given function to every decision in a tree. For example:

```
ghci> mapDecision (++ "!!") whatToDo
Question "Is it Raining?" (Decision "Take the bus!!")
  (Question "Is it more the 2km?" (Decision "Cycle!!") (Decision "Walk!!"))
```

4

Recall the decision tree data type from the previous assignment:

```haskell
data DTree a
  = Decision a
  | Question String (DTree a) (DTree a)
  deriving (Eq, Show)
```

Your task is to define an `IO` function that asks a user the questions present in the tree based on a user's answers, and returns the appropriate decision:

```haskell
takeDecision :: DTree a -> IO a
```

The function should print the string stored in a question and read the reply from the user. If the reply is equal to the string `"y"` then the first decision tree in the `Question` constructor should be taken, otherwise the second decision tree.

The following excerpts show example interactions with the decision trees from the previous assignment:

```
ghci> takeDecision whatToDo
Is it Raining?  (answer y/n)
> n
Is it more the 2km?  (answer y/n)
> y
"Cycle"
```

and

```
ghci> takeDecision howMuch
Is the inflation above 2 percent?  (answer y/n)
> y
Do you have a mortgage?  (answer y/n)
> n
400
```

Note that the function *returns* the decision, it does not print it. It is GHCi that displays the result in the above excerpts.

5

Consider the following data type definition that models an electronic billboard:

```
type Pixel = (Int, Int)

data BillBoard = BB { size :: (Int, Int), actives :: [Pixel] }
```

A billboard, which can be regarded as a matrix of pixels, consists of its size and a list of active pixels. The size field of a billboard is a tuple of integers where the first element is the number of rows, and the second element the number of columns. A pixel is a pair of integers which denotes its place (row and column) on the billboard. For example, `(0, 3)` is found on the first row and fourth column. We use zero-based indexing, that is, index `(0, 0)` points to the pixel on the first row and first column.

Using the above data definition we can make an example billboard:

```
lambda :: BillBoard
lambda = BB (4, 10) [(0,2),(1,3),(2,2),(2,4),(3,1),(3,5)]
```

which can be textually represented as follows:

```
..#.......
...#......
..#.#.....
.#...#....
```

where an active pixel is represented by the character `'#'`, a non-active pixel by a dot `'.'`.

Your tasks are:

a) Write a function that returns the number of inactive pixels:

```
inactives :: BillBoard -> Int
```

b) Define a function that *inverts* all the pixels in a billboard:

```
invert :: BillBoard -> BillBoard
```

All active pixels in an inverted billboard become inactive and all inactive pixels become active. For example, the following shows the textual representation of calling `invert` on `lambda`:

```
##.#######
###.######
##.#.#####
#.###.####
```

6

In assignment 2 we introduced the term *coprime*. We are going define two properties that all coprime number pairs should have.

Your tasks in this assignment are:

a) Write a property that validates that at least one of the numbers in a coprime pair is odd:

```
prop_odd :: (Int, Int) -> Bool
```

since if the numbers were both even then 2 would be a common divisor, which makes the pair not coprime.

b) The *least common multiple* of the numbers in a coprime pair (x, y) is equal to their product $x * y$. Implement a property that checks this:

```
prop_lcm :: (Int, Int) -> Bool
```

The `Prelude` contains a function `lcm` that calculates the least common multiple, which you may use.

7

In this question you should define a number of data types to model a *phone company*. A phone company consists of a list of customers and its VAT[1] number. A customer in a phony company has a name, a phone number, and a particular plan. A plan can either be a pre-paid[2] plan, which stores the amount of minutes left (for making phone calls), or it can be a subscription[3], which stores the amount of minutes left and the amount of data left (for surfing the web).

[1] MOMS
[2] svenska: kontantkort
[3] svenska: abonnemang

Part II

8

The *HyperText Markup Language* (HTML) is a language for describing documents. All webpages are written using HTML. Documents written in HTML have a structure that is determined by the use of *tags*. We can enclose a particular part of our document within certain tags, to indicate this structure. To enclose a part of a document in tags, we use matching open tags and close tags. For example:

- Text enclosed in boldface tags `<B> ... </B>` indicates that the text should be in boldface. Here, `<B>` is the open tag, and `</B>` is the corresponding close tag.
- Text enclosed in emphasize tags `<EM> ... </EM>` indicates that the text should be emphasized (often using italics).
- Text enclosed in paragraph tags `<P> ... </P>` indicates that the text forms a paragraph (often by having an empty line before and after).

(In reality, tags contain more information than just the tag name (such as `B`, `EM`, `P`, etc.), but for simplicity we do not deal with that here.) Here is an example of HTML code:

```
Welcome to my website!
<P>
  <B>
    My hobbies are <EM>Haskell</EM> programming and playing <EM>Myst</EM>.
  </B>
</P>
<P>
  Thanks for visiting! <EM>anna@gmail.com</EM>.
  <P>
    Bye, bye!
  </P>
</P>
```

and here is what it would look like in a browser:

Welcome to my website!

My hobbies are *Haskell* programming and playing *Myst*.

Thanks for visiting! *anna@gmail.com*

Bye, bye!

We can represent HTML documents in Haskell as a list of tags:

```
type HTML = [Tag]
```

There are three different kinds of tags: a piece of text, an open tag, and a close tag.

```haskell
data Tag
  = Text  String
  | Open  String
  | Close String
  deriving (Eq, Show)
```

The example piece of HTML above can be represented by the following Haskell expression:

```haskell
annasSida :: HTML
annasSida =
  [ Text "Welcome to my website!"
  , Open "P"
    , Open "B"
      , Text "My hobbies are ", Open "EM", Text "Haskell", Close "EM"
      , Text " programming and playing ", Open "EM", Text "Myst", Close "EM"
      , Text "."
    , Close "B"
  , Close "P"
  , Open "P"
    , Text "Thanks for visiting! ", Open "EM", Text "anna@gmail.com", Close "EM"
    , Open "P", Text ". Bye, bye!", Close "P"
  , Close "P"
  ]
```

Your tasks are:

a) Define a function

```haskell
render :: HTML -> String
```

that converts an HTML document to a string containing HTML code. The function should write out opening and closing tags enclosed in angled brackets. For example:

```
ghci> toHtml annasSida
"Welcome to my website!<P><B>My hobbies are <EM>Haskell</EM> programm..."
```

The resulting string is truncated to fit on the page, but contains the entire HTML document and has the same content as the example code on the previous page.

b) Write a function:

```haskell
checkHtml :: HTML -> Bool
```

that validates that every open tag has a corresponding close tag. In an HTML document the tags can be nested (e.g., `<P><B>Hej!</B></P>`), but not intertwined (`<P><B>Hej!</P></B>`). So, after an open tag the first close tag in the remainder of the list should match the open tag.

9

Consider the following data type that models a small symbolic expression language with integer literals, addition, and multiplication. In addition, the expression language also supports a 'let-expression', which allows for a local definition that can be used in (sub)expression. In a 'let-expression' you can *bind* an expression to a variable and use this variable in another expression. For example: `let x = 123 in x + x`, which would evaluate to 246.

```haskell
type Name = String

data Expr
  = Num Int             -- Literal integer
  | Add Expr Expr       -- Addition
  | Mul Expr Expr       -- Multiplication division
  | Let Name Expr Expr  -- let x = e1 in e2
  | Var Name            -- Variable
```

Using this data type we can define some example expressions:

```haskell
-- We make 'Expr' an instance of 'Num' for short hand notation
instance Num Expr where
  (+) = Add
  (*) = Mul
  fromInteger = Num . fromInteger

-- Example expressions and their string representation (in comment)
(x, y, z) = (Var "x", Var "y", Var "z")

e1, e2, e3, e4, e5, e6, e7 :: Expr
e1 = (4 + 5) * 2                        -- (4 + 5) * 2
e2 = Let "x" 3 (x + x)                  -- let x = 3 in x + x
e3 = 5 + 4 * x                          -- 5 + 4 * x
e4 = Let "x" 1 (Let "y" 2 (x + y))      -- let x = 1 in let y = 2 in x + y
e5 = Let "x" (2 * 3) (x * x) + 12       -- let x = (2 * 3) in x * x + 12
e6 = Let "x" 2 (x * Let "x" 4 (x * x))  -- let x = 2 in x * let x = 4 in x * x
e7 = Let "x" 5 (x + y + z)              -- let x = 5 in x + y + z
```

Write a function that evaluates an expression:

```haskell
eval :: Expr -> Maybe Int
```

The evaluation may fail in case there is an unbound variable (a variable not bound be a `Let`). For example, the evaluation of `e6` results in `Just 48`, whereas the evaluation of `e3` results in `Nothing` because `x` is not bound. Note that a variable can be *shadowed* by another variable, this happens if you have nested 'let-expressions' that bind the *same* variable name. The evaluation should use the innermost binding. For example, in `e6` in the subexpression `(x * x)`, the variable `x` refers to 4 (the innermost binding of `x`) and not 2.

```haskell
{- This is a list of selected functions from the
   standard Haskell modules: Prelude Data.List
   Data.Maybe Data.Char Control.Monad         -}
-----------------------------------------
-- * Standard type classes

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           :: 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 ()

liftM   :: Monad m => (a -> b) -> m a -> m b
liftM f m1   = do x1 <- m1
                  return (f x1)
-----------------------------------------
-- * 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,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]         = []
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 "Prelude.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 xs@(x:xs')
            | p x       =  dropWhile p xs'
            | otherwise =  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 ["apa","bepa","cepa"]
--       == "apa\nbepa\ncepa\n"
-- unwords ["apa","bepa","cepa"]
--       == "apa bepa cepa"

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]]
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]

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
-----------------------------------------
-- * Useful functions from Test.QuickCheck

arbitrary :: Arbitrary a => Gen a
-- the generator for values of a type
-- in class Arbitrary, used by quickCheck

choose :: Random a => (a, a) -> Gen a
-- Generates 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 list of generators with
-- weighted random distribution.

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 on
-- the size parameter.
-----------------------------------------
-- * Useful IO function
putStr, putStrLn  :: String -> IO ()
getLine           :: IO String

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