

Examiner: Thomas Hallgren, D&IT,
Answering questions at approx 15.00 (or by phone)

Functional Programming TDA 452, DIT 142

2017-04-11 14.00 – 18.00 “Maskin”-salar (M)

- There are 6 questions with maximum $6 + 8 + 6 + 5 + 5 + 10 = 40$ points. Grading:
Chalmers: 3 = 20–26 points, 4 = 27–33 points, 5 = 34–40 points
GU: G = 20–33 points, VG = 34–40 points
- Results: latest approximately 10 days.
- **Permitted materials:**
 - Dictionary
- **Please read the following guidelines carefully:**
 - Read through all Questions before you start working on the answers.
 - Begin each Question on a new sheet.
 - Write clearly; unreadable = wrong!
 - For each part Question, if your solution consists of more than a few lines of Haskell code, use your common sense to decide whether to include a short comment to explain your solution.
 - You can use any of the standard Haskell functions *listed at the back of this exam document*.
 - **Full points** are given to solutions which are **short**, **elegant**, and **correct**. Fewer points may be given to solutions which are unnecessarily complicated or unstructured.
 - You are **encouraged to use** the solution to an **earlier part** of a Question to help solve a **later part** — even if you did not succeed in solving the earlier part.

1. (6 points) Given the following type for trees,

```
data Tree a = Leaf a | Branch (Tree a) (Tree a) deriving (Eq,Show)
```

```
t = Branch (Leaf 5) (Branch (Leaf 1) (Leaf 10)) -- an example tree
```

define

- a function that computes the sum of the values in a tree, and
- a function that finds the minimum element in a tree.

```
sumTree :: Num a => Tree a -> a -- Example: sumTree t == 16
```

```
minTree :: Ord a => Tree a -> a -- Example: minTree t == 1
```

While these functions can be defined as two separate recursive functions, they will be very similar, so this solution will give you at most 4 points. For full points, define a (recursive) helper function that captures the common pattern, and make `sumTree` and `minTree` two simple (non-recursive) functions that call the helper function.

Solution:

```
foldTree :: (a->a->a) -> Tree a -> a
foldTree f (Leaf a) = a
foldTree f (Branch l r) = f (foldTree f l) (foldTree f r)

sumTree = foldTree (+)
minTree = foldTree min

-- Alternative solution:
sumTree' (Leaf a) = a
sumTree' (Branch l r) = sumTree' l + sumTree' r

minTree' (Leaf a) = a
minTree' (Branch l r) = min (minTree' l) (minTree' r)

-- Another alternative solution:
toList :: Tree a -> [a]
toList (Leaf a) = [a]
toList (Branch l r) = toList l ++ toList r

sumTree'' :: Num a => Tree a -> a
minTree'' :: Ord a => Tree a -> a
sumTree'' = sum . toList
minTree'' = minimum . toList
```

2. (8 points)

- (a) (3 points) Reimplement the following function by using recursion directly, traversing the argument list only once.

```
splitAt :: Int -> [a] -> ([a],[a])
splitAt n xs = (take n xs,drop n xs)
```

Solution:

```
splitAt :: Int -> [a] -> ([a],[a])
splitAt n xs | n<=0 = ([],xs)
splitAt _ [] = ([],[])
splitAt n (x:xs) = (x:ys,zs) where (ys,zs) = splitAt (n-1) xs

-- For testing, not part of the solution
prop_splitAt n xs = splitAt n xs == (take n xs,drop n xs)
```

- (b) (3 points) Define a function `chunksOf` that splits a lists into chunks of the length indicated by the first argument.

```
chunksOf :: Int -> [a] -> [[a]]

-- Example: chunksOf 4 [1..10] == [[1,2,3,4],[5,6,7,8],[9,10]]
```

Solution:

```
chunksOf n [] = []
chunksOf n xs = xs1:chunksOf n xs2 where (xs1,xs2) = splitAt n xs
```

- (c) (2 points) Write a property that describes what the expected length of `chunksOf n xs` is, depending on `n` and `xs`.

Solution:

```
prop_length_chunksOf n xs =
  n>0 ==> length (chunksOf n xs) == (length xs+(n-1)) 'div' n
```

3. (6 points) For each of the following functions, give the most general type, or write No type if the definition is not type correct in Haskell.

```
fa x = [x,x]
fb x y = if x then y else -y
fc (f,g) (x,y) = (f x,g y)
```

Solution:

```
fa :: x -> [x]
fb :: Num a => Bool -> a -> a
fc :: (a->b,c->d) -> (a,c) -> (b,d)
```

4. (5 points) Given the following data types for playing cards and hands of playing cards,

```
data Card = ... deriving (Eq,Show) -- the details are unimportant
data Hand = Empty | Add Card Hand deriving (Eq,Show)
```

```
rCard :: Gen Card -- generates a random Card
```

define a QuickCheck test data generator

```
rHand :: Gen Hand
```

that generates a random hand of cards that contains at least one card, at most 6 cards, and doesn't contain the same card twice.

Solution:

```
rHand = do size <- choose (1,6)
          cards <- vectorOf size rCard
          return (toHand (nub cards))
where
  toHand :: [Card] -> Hand
  toHand [] = Empty
  toHand (c:cs) = Add c (toHand cs)
```

5. (5 points) Consider the following imperative style function that prints a string inside a frame of asterisks:

```
printFramed :: String -> IO ()
printFramed s = -- printFramed " Hello world "
  do putStars (n+2) --
    putStr "\n*" -- *****
    putStr s -- * Hello world *
    putStr "*\n" -- *****
    putStars (n+2)
    putStr "\n"
where
  n = length s

  putStars 0 = return ()
  putStars n = do putStr "*"
                 putStars (n-1)
```

Write a shorter, functional style implementation that produces the same output:

```
printFramed s = putStr (framed s)
framed :: String -> String
framed s = ...
```

To keep the definition short, use appropriate predefined functions instead of your own recursive functions.

Solution:

```
framed :: String -> String
framed s = stars ++ "\n*" ++ s ++ "*\n" ++ stars ++ "\n"
  where
    stars = replicate n '*'
    n = length s + 2
```

6. (10 points) Consider the following representation of a simple variant of XML:

```
type TagName = String      -- A few lowercase letters 'a'..'z'

data XML = Text String     -- Arbitrary text
         | Elem TagName [XML] -- A tagged element <t>...</t>
         deriving (Eq,Show)
```

(a) (3 points) Write a function

```
tableToXML :: [[String]] -> XML
```

that takes a table (represented as a list of rows of cells) and creates an HTML table for it (assuming here that HTML is an instance of XML). Example:

```
tableToXML [ ["A","B"], ["C","D"] ] ==
  Elem "table" [Elem "tr" [Elem "td" [Text "A"],Elem "td" [Text "B"]],
               Elem "tr" [Elem "td" [Text "C"],Elem "td" [Text "D"]]]
```

Solution:

```
tableToXML rows = Elem "table" (map rowToXML rows)
  where
    rowToXML row = Elem "tr" (map cellToXML row)
    cellToXML cell = Elem "td" [Text cell]
```

(b) (5 points) Write a function

```
showXML :: XML -> String
```

that outputs the textual representation of an XML document. Some characters in the text in the XML document need to be treated specially: since < starts a tag, any < in text should be output as <;, and any & should be output as &.

Examples:

```
showXML (Elem "td" [Text "hello"]) == "<td>hello</td>"
showXML (Text "this & that") == "this &amp; that"
showXML (Elem "td" [Text "4 < 5"]) == "<td>4 &lt; 5</td>"
```

Solution:

```
showXML (Text s) = escape s
showXML (Elem tag xml) = start tag++concatMap showXML xml++end tag

start,end :: TagName -> String
start tag = "<"+tag++>"
end tag = start ('/':tag)

escape :: String -> String
escape s = concatMap escape1 s
  where
    escape1 '<' = "&lt;"
    escape1 '&' = "&amp;"
    escape1 c = [c]
```

- (c) (2 points) Use the two functions above to define a function that creates the HTML code for a table containing values of an arbitrary type in the `Show` class:

```
renderTable :: Show a => [[a]] -> String
```

Example:

```
renderTable [[1,2],[3,4]] ==
  "<table><tr><td>1</td><td>2</td></tr>"+
  "<tr><td>3</td><td>4</td></tr></table>"
```

Solution:

```
renderTable = showXML . tableToXML . map (map show)
```

```

{-
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 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
    xs <- q
    return (x:xs)

sequence_ :: Monad m => [m a] -> m ()
sequence_ xs = do
  sequence xs
  return ()

liftM :: (Monad m) => (a1 -> r) -> m a1 -> m r
liftM f ml = do
  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 :: f x y
      => (a -> b -> c) -> b -> a -> c
flip f x y = f y x

($) :: (a -> b) -> a -> b
f $ x = f x

-- functions on Booleans
data Bool = False | True

(&&), (||) :: Bool -> Bool -> Bool
True && x = x
False && x = False
True || x = True
False || x = x

not :: Bool -> Bool
not True = False
not False = True

-- functions on Maybe
data Maybe a = Nothing | Just a

!isJust :: Maybe a -> Bool
isJust (Just a) = True
isJust Nothing = False

!isNothing :: Maybe a -> Bool
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 l = [x | Just x <- l]

-- 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 xs = [a] -> [a]
           = 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 (x: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) = x : takeWhile p xs

dropWhile p [] = []
dropWhile p (x:xs) = dropWhile p xs'
  where p x' = dropWhile p 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"
-- 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"
minimum (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 :: [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)
  | x == y then xs else x : delete y xs

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

```

```

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)
  | 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

-----
-- Signatures of some 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.

```