**Chalmers** | Göteborgs Universitet 2017-01-12

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

# Functional Programming TDA 452, DIT 142

2017-01-12 14.00 – 18.00 "Maskin"-salar (M)

- There are 6 questions with maximum $6 + 9 + 7 + 6 + 6 + 6 = 40$ points; a total of 20 points definitely guarantees a pass.
- 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)*

   (a) *(3 points)* Give a define of the function

   ```
   findIndices :: (a->Bool) -> [a] -> [Int]
   ```

   that given a test and a list returns the *indexes* of the elements in the list that pass the test. Examples:

   ```
   findIndices isUpper "Hello World" == [0,6]
   findIndices isDigit "Hello World" == []
   filter isUpper "Hello World" == "HW"
   ```

   **Solution:**

   ```
   findIndices p xs = [i|(i,x)<-zip [0..] xs,p x]

   findIndices_v2 p = map fst . filter (p.snd) . zip [0..]
   ```

   (b) *(3 points)* Write a property to verify that the elements at the indexes returned by `findIndices` pass the test.

   ```
   prop_findIndices :: (a->Bool) -> [a] -> Bool
   ```

   **Solution:**

   ```
   prop_findIndices p xs = and [p (xs!!i) | i<-is]
     where is = findIndices p xs

   -- Also accepted, although it doesn't test exactly what was asked for...
   prop_findIndices_v2 p xs = [xs!!i | i<-findIndices p xs] == filter p xs
   ```

2. *(9 points)*

   (a) *(3 points)* The following function works as intended, but the code is not as simple and efficient as it could be:

   ```
   split :: [a] -> ([a],[a])
   split xs | length xs == 0 = ([],[])
            | length xs == 1 = (xs,[])
            | otherwise      = (head xs:fst p,head (tail xs):snd p)
                where p = split (drop 2 xs)
   ```

   Simplify the code as much as you can. In particular, use pattern matching instead of `fst`, `snd`, `length`, `head`, `tail` and `drop`.

   **Solution:**

   ```
   split' :: [a] -> ([a],[a])
   split' (x1:x2:xs) = (x1:xs1,x2:xs2) where (xs1,xs2) = split' xs
   split' xs = (xs,[])
   ```

   (b) *(3 points)* Define a function `merge` that merges two sorted lists into a sorted list.

```
merge     :: Ord a => [a] -> [a] -> [a]
```

**Solution:**

```
merge [] ys = ys
merge xs [] = xs
merge (x:xs) (y:ys) | x<=y      = x:merge xs (y:ys)
                    | otherwise = y:merge (x:xs) ys
```

(c) *(3 points)* Define a function `mergeSort` that sorts a list by splitting it into two parts, recursively sorting the parts, and merging the results.

```
mergeSort :: Ord a => [a] -> [a]
```

**Solution:**

```
mergeSort []  = []
mergeSort [x] = [x]
mergeSort xs  = merge (mergeSort xs1) (mergeSort xs2)
  where (xs1,xs2) = split xs
```

3. *(7 points)*

(a) *(2 points)* Define a data type `Tree a` for binary trees that have empty leaves and a value of type `a` in each internal node.

**Solution:**

```
data Tree a = Empty | Branch a (Tree a) (Tree a)
            deriving (Eq,Show)  -- optional
```

(b) *(3 points)* Define an `instance Functor Tree` with the expected behaviour for the standard class `Functor`, defined as

```
class Functor f where fmap :: (a->b) -> f a -> f b
```

**Solution:**

```
instance Functor Tree where
  fmap f Empty = Empty
  fmap f (Branch x l r) = Branch (f x) (fmap f l) (fmap f r)
```

(c) *(2 points)* Define a function `doubleTree` that doubles all the numbers in a tree of numbers.

```
doubleTree :: Num a => Tree a -> Tree a
```

**Solution:**

```
doubleTree = fmap (*2)
```

4. *(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 = x < y+1
fc x (y,z) = x y z
fd x y z = x (y z)
fa x = [x,x]
fb x y = if x then y else -y
fc (f,g) (x,y) = (f x,g y)
```

**Solution:**

5. *(6 points)*

   (a) *(2 points)* Given a function

   ```
   parse :: Parser a -> String -> Maybe (a,String)
   ```

   that applies a parser to an input string and returns `Nothing` if the parser fails, and `Just (a,s)` if the parser succeeds, where `a` is the result of the parser and `s` is the remaining unused input. Define

   ```
   completeParse :: Parser a -> String -> Maybe a
   ```

   that succeeds and returns `Just a` only if the parser accepted all of the input string (i.e., the remaining input is empty). (*Hint:* you don't need to know anything more about the `Parser` type in order to answer this question.)

   **Solution:**

   ```
   completeParse p s = case parse p s of
                         Just (a,"") -> Just a
                         _ -> Nothing
   ```

   (b) *(4 points)* The QuickCheck function `vectorOf` generates a list of a given length, using a given generator for the elements. Implement two versions of `vectorOf`:

   i. Using only standard library functions.

   ii. Using recursion directly.

   ```
   vectorOf_i,vectorOf_ii :: Int -> Gen a -> Gen [a]
   ```

**Solution:**

```
vectorOf_i n g = sequence (replicate n g)

vectorOf_ii 0 g = return []
vectorOf_ii n g = do x <- g
                     xs <- vectorOf_ii (n-1) g
                     return (x:xs)

vectorOf_ii' 0 g = return []
vectorOf_ii' n g = (:) <$> g <*> vectorOf_ii' (n-1) g
```

6. *(6 points)* The *Luhn algorithm* is used to check bank card numbers for simple errors such as mistyping a digit, and proceeds as follows:

   - consider each digit as a separate number;
   - moving left, double every other number from the second last;
   - subtract 9 from each number that is now greater than 9;
   - add all the resulting numbers together;
   - if the total is divisible by 10, the card number is valid

   Define a function `validCard` and suitable helper functions to check if a bank card number is valid according to the above algorithm.

   ```
   validCard :: Integer -> Bool
   ```

   Examples:

   ```
   validCard 1784 == True      -- 2*1 + 7 + 2*8-9 + 4 == 20
   validCard 1874 == False     -- 2*1 + 8 + 2*7-9 + 4 == 19
   ```

**Solution:**

```
-- Compact version
validCard = (==0) . (`mod` 10) . sum .
            map sub9 . zipWith (*) (cycle [1,2]) .
            reverse . map digitToInt . show
  where
    sub9 n = if n>9 then n-9 else n

-- More verbose, self-documenting variant
validCard_v2 :: Integer -> Bool
validCard_v2 n = luhn n `mod` 10 == 0
  where
    luhn = sum . map subtract9 . doubleEveryOther . integerToList

    subtract9 n = if n>9 then n-9 else n

    integerToList :: Integer -> [Int]
    integerToList = reverse . map (\x->read [x]) . show

    doubleEveryOther :: [Int] -> [Int]
    doubleEveryOther []         = []
    doubleEveryOther [d]        = [d]
    doubleEveryOther (d1:d2:ds) = d1:(2*d2):doubleEveryOther ds
```

```
{-
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 x <- p
                       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 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                  :: 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 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"
-- 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

-------------------------------------------
-- 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.
```