Chalmers | Göteborgs Universitet 2018-04-05

Examiner: Thomas Hallgren, `hallgren@chalmers.se`, D&IT,
Answering questions at approx 15.00 (or by phone 0735 124 575)

Functional Programming TDA 452, DIT 142

2018-04-05 14.00 – 18.00 "Maskin"-salar (M)

0735 124 575

- There are 5 questions with maximum 8 + 10 + 6 + 7 + 9 = 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. *(8 points)* For each of the following definitions, give the most general type, or write "No type" if the definition is not correct in Haskell.

```
fa x   = x : x
fb x   = x ++ x
fc x y = show (x !! y)
fd     = filter fst
```

2. *(10 points)*

   (a) *(3 points)* Define a function `scramble` that takes a list and puts all the elements at even positions before the elements at odd position. Positions are numbered starting at `0`, so the first element is at an even position. The function should run in linear time and its type should be as general as possible.

   Examples:

   ```
   scramble [0..10] == [0,2,4,6,8,10,1,3,5,7,9]
   scramble "Functional Programming" == "Fntoa rgamnucinlPormig
   ```

   (b) *(3 points)* Define a function `unscramble` that is the inverse of `scramble`. The function should run in linear time and its type should be as general as possible.

   Examples:

   ```
   unscramble [0,2,4,6,8,10,1,3,5,7,9] == [0,1,2,3,4,5,6,7,8,9,10]
   unscramble "Fntoa rgamnucinlPormig" == "Functional Programming"
   ```

   (c) *(2 points)* Define a function `iter` that applies a function a given number of times to its argument.

   ```
   iter :: Int -> (a->a) -> a -> a
   ```

   Examples:

   ```
   iter 0 (*2) 1 = 1
   iter 5 (*2) 1 = 32
   ```

   (d) *(2 points)* Define a QuickCheck property that expresses that scrambling a list n times and then unscrambling it n times gives you the original list back. Make sure you avoid problems with negative numbers.

3. *(6 points)*

   (a) *(2 points)* Define a function that generates a random vowel (i.e. one of the letters in the string `"aeiouy"`), and a function that generates a random consonant (i.e. a letter a-z that is not a vowel).

   ```
   rVowel, rConsonant :: Gen Char
   ```

   (b) *(4 points)* Write a random password generator.

   ```
   randomPassword :: Gen String
   ```

   The passwords should be random sequences of letters, but to make them easier to remember, every other letter should be a consonant, and every other letter should be a vowel. Use `rConsonant` and `rVowel` from above to generate the consonants and vowels. The length of the passwords should vary randomly between 8 and 10.

   Here are a few examples of passwords generated in this way: `xazybenu`, `bilikocy`, `nohuruci`, `bisiqamotu`, `dixygahoba`, `huwamapun`.

4. *(7 points)* Write a simple spelling checker, i.e., a function that reads a dictionary and a text file and checks that all words in the text are spelled correctly according to the dictionary.

   ```
   checkSpelling :: FilePath -> IO [(String,Int)]
   ```

   The argument is the name of the file to check. The result is a list of misspelled words paired with the line number of the line they appeared on.

   Assume that there is a function that loads the dictionary and returns a pair of functions:

   ```
   loadDictionary :: IO (String->[String],String->Bool)
   ```

   The first function in the pair splits a string into words and removes punctuation. The second function checks if a word is spelled correctly.

   In addition to the functions listed at the back of this exam, the following library function might be useful:

   ```
   -- readFile reads the contents of a file
   readFile :: FilePath -> IO String

   -- File names are strings.
   type FilePath = String
   ```

5. *(9 points)*

   (a) *(3 points)* Define a recursive data type `Expr` to represent arithmetic expressions with numbers, variables, addition and multiplication. Let the numbers have type `Double` and the variable names be represented as strings.

   ```
   type Name = String

   data Expr = ...
   ```

   (b) *(3 points)* Define a function that computes the value of an expression, given an association list with the values of the variables. The function is allowed fail if it encounters an unknown variable.

   ```
   valueOfExpr :: [(Name,Double)] -> Expr -> Double
   ```

   (c) *(3 points)* Let a list of definitions $x_1 = e_1, ..., x_n = e_n$ be represented as a list of pairs of variable names ane expressions. Define a function that computes the values all the variables defined in a list of definitions.

   ```
   valuesOfDefinitions :: [(Name,Expr)] -> [(Name,Double)]
   ```

   Note that thanks to lazy evaluation, this definition can be very simple. You don't need to worry about in which order to compute the definitions, lazy evaluation takes care of that. You can assume that the list of definitions is free from circularities, e.g. $x = y, y = x$. (It's OK if the function crashes or loops if there are circularities.)

Example:

```
e1, e2, e3 :: Expr
e1 = ... -- the representation of x+1   in your data type
e2 = ... -- the representation of y+2*x in your data type
e3 = ... -- the representation of 3     in your data type
eqns = [("y",e1),("z",e2),("x",e3)]
prop_values = valuesOfDefinitions eqns == [("y",4),("z",10),("x",3)]
```

```
{-
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.
```