Chalmers | Göteborgs Universitet 2018-01-11

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

Functional Programming TDA 452, DIT 142

2018-01-11 14.00 – 18.00 Samhällsbyggnad

- There are 5 questions with maximum $7 + 8 + 8 + 9 + 8 = 40$ 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. *(7 points)* The standard library function `lookup`,

   ```
   lookup :: Eq key => key -> [(key,value)] -> Maybe value
   ```

   lets you lookup the first value corresponding to a given key in a list of key-value pairs.

   (a) *(2 points)* Define a function that returns *all* values paired with the given key.

   ```
   lookupAll :: Eq key => key -> [(key,value)] -> [value]
   ```

   For 2 points, use a list comprehension. Other solutions will be longer and give at most 1 point.

   (b) *(2 points)* Use `lookupAll` to make a simple, non-recursive definition of the standard `lookup` function.

   (c) *(3 points)* Define a function `update`,

   ```
   update :: Eq key => key -> value -> [(key,value)] -> [(key,value)]
   ```

   that updates a list of key-value pairs. If the new key already exists in the list, the value should be replaced, otherwise a new pair should be added to the list. The order between the pairs in the list does not matter. Examples:

   ```
   update 'x' 5 [] == [('x',5)]
   update 'x' 5 [('y',3)] == [('y',3),('x',5)]
   update 'x' 5 [('y',3),('x',2),('z',4)] == [('y',3),('x',5),('z',4)]
   ```

Solution:

```
-- (a)
lookupAll key kvs = [v|(k,v)<-kvs,k==key]

-- (b)
lookup key = listToMaybe . lookupAll key

lookup_v2 key kvs = case lookupAll key kvs of
                      []  -> Nothing
                      v:_ -> Just v
-- (c)
-- If we assume that k does not appear multiple times in the list
update k v [] = [(k,v)]
update k v ((oldk,oldv):kvs) | oldk==k   = (oldk,v)    : kvs
                             | otherwise = (oldk,oldv) : update k v kvs

-- If we update all occurrences of k in the list
update_v2 k v kvs
  | null (lookupAll k kvs) = (k,v):kvs
  | otherwise              = [(k',if k'==k then v else v') | (k',v')<-kvs]

-- If we remove all previous occurrences of k
update_v3 k v kvs = (k,v):[(k',v')|(k',v')<-kvs,k'/=k]
```

2. *(8 points)* Consider the following data types for representing arithmetic expressions with multiplication, addition, subtraction and a variable X:

```
data Expr  = X | Num Int | Op BinOp Expr Expr  deriving (Eq,Show)
data BinOp = Add | Mul | Subtract              deriving (Eq,Show)

ex1 = Op Subtract (Num 100) X                  -- 100 - X
ex2 = Op Add (Num 100) (Op Mul (Num (-1)) X)   -- 100 + (-1)*X
```

(a) *(4 points)* Define a function that computes the value of an expression, given the value of the variable X. Example: `eval ex1 25 == 75`.

```
eval :: Expr -> Int -> Int
```

(b) *(4 points)* Although this data type can represent subtraction, it is not really needed since an expression such as, for example, 100 - X can be written as 100 + (-1) * X. Define a function

```
removeSub :: Expr -> Expr
```

which removes *all* subtraction operators in an expression by replacing them with a combination of addition and multiplication as in the above example. For example, `removeSub ex1 == ex2`.

Your definition should only remove the subtraction operators. It should not attempt to simplify or evaluate the expression in any way.

Solution:

```
-- (a)
eval X             x = x
eval (Num n)       x = n
eval (Op op e1 e2) x = evalOp op (eval e1 x) (eval e2 x)
    where evalOp Add      = (+)
          evalOp Mul      = (*)
          evalOp Subtract = (-)

eval_v2 X                   x = x
eval_v2 (Num n)             x = n
eval_v2 (Op Add e1 e2)      x = eval_v2 e1 x + eval_v2 e2 x
eval_v2 (Op Mul e1 e2)      x = eval_v2 e1 x * eval_v2 e2 x
eval_v2 (Op Subtract e1 e2) x = eval_v2 e1 x - eval_v2 e2 x


-- (b)
removeSub (Op op e1 e2) = binop op (removeSub e1) (removeSub e2)
    where binop Subtract e1 e2 = Op Add e1 (Op Mul (Num (-1)) e2)
          binop op       e1 e2 = Op op e1 e2
removeSub e             = e

removeSub_v2 (Op Add e1 e2) = Op Add (removeSub_v2 e1) (removeSub_v2 e2)
removeSub_v2 (Op Mul e1 e2) = Op Mul (removeSub_v2 e1) (removeSub_v2 e2)
removeSub_v2 (Op Subtract e1 e2) = Op Add r1 (Op Mul (Num (-1)) r2)
                                 where r1 = removeSub_v2 e1
                                       r2 = removeSub_v2 e2
removeSub_v2 e             = e
```

3. *(8 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 y = not (x && y)
fb (&) x y = not (x & y)
fc x y = (x+y)/2
fd x = [z | y<-x, z<-y]
```

Solution:

```
fa :: Bool -> Bool -> Bool
fb :: (a->b->Bool) -> a -> b -> Bool
fc :: Fractional a => a -> a -> a
fd :: [[a]] -> [a]
```

4. *(9 points)* Consider the following data type for representing rectangular grids:

```
data Grid a = Grid [[a]]  deriving (Eq,Show)

g1,g2 :: Grid Int    -- Example grids
g1 = Grid [[1,2],
           [3,4],
           [5,6]]

g2 = Grid [[5,3,1],
           [6,4,2]]
```

(a) *(2 points)* Define a function that a applies a function to every element of a grid.

```
mapGrid :: (a->b) -> Grid a -> Grid b
```

(b) *(2 points)* Define a function that rotates a grid 90 degrees clockwise.

```
rotateGrid :: Grid a -> Grid a    -- Example: rotateGrid g1 == g2
```

(c) *(3 points)* Define a QuickCheck test data generator for rectangular grids, including an instance in the `Arbitrary` class.

(d) *(2 points)* Define a property that expresses the fact that rotating a rectangular grid four times returns the grid you started with.

Hint: The above functions are easier to define by reusing suitable library functions than by using recursion on lists.

Solution:

```
-- (a)
mapGrid f = Grid . map (map f) . rows

-- (b)
rotateGrid = Grid . map reverse . transpose . rows

rotateGrid_v2 = Grid . transpose . reverse . rows

rows (Grid rs) = rs   -- a common helper function

-- (c)
instance Arbitrary a => Arbitrary (Grid a) where
  arbitrary = do height <- choose (1,10)
                 width <- choose (1,10)
                 rows <- vectorOf height (vectorOf width arbitrary)
                 return (Grid rows)

-- (d)
--prop_rotateGrid4 :: Eq a => Grid a -> Bool -- (*)
prop_rotateGrid4 :: Grid Int -> Bool
prop_rotateGrid4 g = (r . r . r . r) g == g
  where r = rotateGrid
-- (*) Both types are OK. The property holds for all types of grids, as long
-- as we can test that two grids are equal, but for testing with QuickCheck,
-- we need to choose a particular type.

prop_rotateGrid4_v2 g = iterate rotateGrid g !! 4 == g
```

5. *(8 points)* Write a Haskell function

```
checkHaskellFiles :: IO ()
```

that examines the Haskell source files in the current directory and reports the ones that contain lines that are too long. To be more specific:

- For each Haskell file that contains lines that are too long, the function should report how many lines are too long.
- Lines are too long if they contain more than 78 characters. A Haskell file that does not contain any lines that are too long is OK.
- The report should not mention Haskell files that are OK. The report should be empty if all Haskell files are OK.
- Haskell source files have names that end with `.hs`. Other files can be present, but they should be ignored.

Example: if the current directory contains the files `A.hs` (which is OK), `B.hs`, `C.hs`, `D.hs` and `E.pdf` (which is not a Haskell source file), the report might look like this:

```
B.hs: one line is too long
C.hs: 5 lines are too long
D.hs: 12 lines are too long
```

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

```
-- Functions to output text
putStr, putStrLn :: String -> IO ()

-- listDirectory returns the names of all the files in a directory
listDirectory :: FilePath -> IO [FilePath]

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

-- File names are strings. The name of the current directory is "."
type FilePath = String
```

Solution:

```
checkHaskellFiles =
  do files <- listDirectory "."
     sequence_ [checkHaskellFile f | f<-files,".hs" `isSuffixOf` f]
  where
    checkHaskellFile file =
      do s <- readFile file
         let long = length [l | l<-lines s, length l>78]
         case long of
           0 -> return ()
           1 -> putStrLn (file++": one line is too long")
           _ -> putStrLn (file++": "++show long++" lines are too long")

checkHaskellFiles_v2 =
  do hfs <- listHaskellFiles "."
     hfcs <- zip hfs <$> mapM readFile hfs
     putStr (concatMap reportHaskellFile hfcs)
  where
    listHaskellFiles dir =
      do files <- listDirectory "."
         return [f | f<-files,".hs" `isSuffixOf` f]

    reportHaskellFile :: (FilePath,String) -> String
    reportHaskellFile (filename,sourcecode) =
        case length [() | line<-lines sourcecode, length line>78] of
          0 -> ""
          1 -> filename++": one line is too long\n"
          n -> filename++": "++show n++" lines are too long\n"
```

```
{-
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.
```