Examiner: David Sands `dave@chalmers.se`, D&IT

Functional Programming TDA 452, DIT 142

2016-01-14 14.00 – 18.00 "Maskin"-salar (M)

Ext 1059 / 031 772 1059

- There are 4 Questions with maximum $11 + 9 + 8 + 12 = 40$ 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!
  - 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.
  - 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.*
  - 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.

A computer once beat me at chess. But it was no match for me at kick boxing.

Question 1. *(11 points)*

(i) *(5 points)* The function xmas defined below prints a festive tree of the given size on the screen, so that typing `xmas 5` would create the following output:

```
    *
   * *
  * * *
 * * * *
* * * * *
```

The function below defines xmas:

```
xmas :: Int -> IO()
xmas n = doprint 1 where
  doprint m = if m > n then return ()
                   else do
                     printCopies (n - m) " "
                     printCopies m       " *"
                     putStrLn ""
                     doprint (m + 1)

     where printCopies k s = if k <= 0 then return ()
                            else do putStr s
                                    printCopies (k-1) s
```

The definition of `xmas` above is not considered to be in good Haskell style since (a) it does not make a good separation between IO and pure computation, and (b) it uses recursion where standard functions could be used instead. Give a new definition for `xmas` which fixes this. For full points your definition should not use recursion, and should do as much computation as possible outside of IO.

Solution

```
xmas' = putStr . tree
tree n = unlines [cr (n - m) " " ++ cr m " *"  | m <- [1..n]]
 where cr k = concat . replicate k
```

(ii) *(4 points)* Define a function

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

which splits a list into chunks at every element satisfying the given predicate. `prop_splitWhen0` should be `True` for your definition of `splitWhen`:

```
prop_splitWhen0 =
    splitWhen (== ';') "A;BB;;DDDD;" == ["A","BB","","DDDD",""]
 && splitWhen (>1)     [3,0,1,2,0,0] == [[],[0,1],[0,0]]
 && splitWhen (>1)     []            == [[]]
```

Hint: a recursive definition using `span` may be simplest. **Solution**

```
-- splitWhen p [] = [[]] -- This gives an extra []
splitWhen p xs = case span (not . p) xs of
                   (c, []) -> [c]
                   (c, r ) -> c : splitWhen p (drop 1 r)
```

(iii) *(2 points)* Describe the expected property of the expression `length (splitWhen p xs)` as a function

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

Solution

```
prop_splitWhen p xs =
    length (filter p xs) + 1 == length (splitWhen p xs)
```

(Note that quickCheck would need a more restricted type than this to be applicable to this function, but that is not important here.)

Question 2. *(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 l m n =  m `lookup` zip l n

fb [] a    = a
fb (b:c) a = fb c (b a)

fc (a:b) (c:d) = b /= c
fc _     e     = null e
```

(3 points) For the following function give its type, and give one example (on no more than one line) of what it does.

```
fd x = map (x:)
```

Solution

```
fa :: Eq a => [a] -> a -> [b] -> Maybe b
fb :: [t -> t] -> t -> t
fc :: Eq t => [t] -> [[t]] -> Bool
fd :: a -> [[a]] -> [[a]]
example = fd 1 [[],[2,3],[4,5]] == [[1],[1,2,3],[1,4,5]]
```

Question 3. *(8 points)* A Sudoku puzzle consists of a 9x9 grid. Some of the cells in the grid have digits (from 1 to 9), others are blank. The objective of the puzzle is to fill in the blank cells with digits from 1 to 9, in such a way that every row, every column and every 3x3 block has exactly one occurrence of each digit 1 to 9.

In lab 3 you represented a sudoku board using the type `data Sudoku = Sudoku [[Maybe Int]]`. In this question you are to use a similar type

```
data Sudoku = Sudoku [[Int]]
```

In this representation, `0` represents the blank square. An example sudoku is

```
ex = Sudoku
    [[3,6,0,0,7,1,2,0,0],[0,5,0,0,0,0,1,8,0],[0,0,9,2,0,4,7,0,0],
     [0,0,0,0,1,3,0,2,8],[4,0,0,5,0,2,0,0,9],[2,7,0,4,6,0,0,0,0],
     [0,0,5,3,0,8,9,0,0],[0,8,3,0,0,0,0,6,0],[0,0,7,6,9,0,0,4,3]]
```

(i) *(4 points)* Define a function

```
showSudoku :: Sudoku -> String
```

such that running `putStrLn (showSudoku ex)` will display the sudoku above as:

```
3|6| | |7|1|2| |
-----------------
 |5| | | | |1|8|
-----------------
 | |9|2| |4|7| |
-----------------
 | | | |1|3| |2|8
-----------------
4| | |5| |2| | |9
-----------------
2|7| |4|6| | | |
-----------------
 | |5|3| |8|9| |
-----------------
 |8|3| | | | |6|
-----------------
 | |7|6|9| | |4|3
```

You may assume that the sudoku is well-formed. **Solution**

```
showSudoku (Sudoku s) = unlines $ intersperse hr $ map showRow s
           where hr = replicate (9*2-1) '-'
                 showRow = intersperse '|' . map showNum
                 showNum 0 = ' '
                 showNum n = head (show n)
```

(ii) *(4 points)* A sudoku contains nine 3×3 "blocks". Define a function

```
block :: (Int,Int) -> Sudoku -> [[Int]]
```

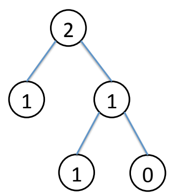
which returns the block corresponding to the two integer arguments (assumed to be in the range from 0 to 2).
For example `block (1,0) ex` should give `[[0,7,1],[0,0,0],[2,0,4]]`. **Solution**

```
block (x,y) (Sudoku s) = takeBlock . dropBlock y . map (takeBlock . dropBlock x) $ s
  where dropBlock z = drop (3 * z)
        takeBlock   = take 3
```

Question 4. *(12 points)* The following data type represents binary trees with elements of any type `a` at the nodes:

```
data Tree a = Leaf | Node a (Tree a) (Tree a)
  deriving Show
```

For example, the tree depicted below



could be represented in this data type by the expression

```
exTree = Node 2 (leafNode 1) (Node 1 (leafNode 1) (leafNode 0))
  where leafNode n = Node n Leaf Leaf
```

(i) *(4 points)* The *height* (also called the *depth*) of a binary tree is the number of nodes on a longest path from root to any leaf.

A binary tree is *balanced* (also called *height balanced*) if it is a leaf, or if it is a Node where the left and right sub-trees are balanced, and their heights differ by no more than one.

Define a function

```
hBalanced :: Tree a -> (Int,Bool)
```

which for any tree computes a pair of its height, and whether it is balanced. For example

```
prop_ex = hBalanced exTree == (3,True)
```

Solution

```
hBalanced Leaf          = (0,True)
hBalanced (Node _ t t') =
       let (h ,b ) = hBalanced t
           (h',b') = hBalanced t'
        in (1 + max h h', abs (h - h') <= 1 && b && b')
```

(ii) *(4 points)* A *path* of a nonempty tree is a list nodes on any path between the root and any leaf. For example, in tree `exTree` there are three paths: `[2,1]`, `[2,1,1]` and `[2,1,0]`. The tree `Leaf` has a single maximal path, `[]`. The tree `Node 0 Leaf (Node 1 Leaf Leaf)` has two paths, `[0]` and `[0,1]`.

Define a function

```
allPaths :: Tree a -> [[a]]
```

which computes a list of all the paths in the given tree. It is OK if the result of your function contains several occurrences of the same path, but all paths must be present in the result. **Solution**

```
allPaths Leaf           = [[]]
allPaths (Node a t1 t2) = map (a:) (allPaths t1 ++ allPaths t2)
```

(iii) *(4 points)* Define

```
balTree :: Gen (Tree Bool)
```

a quickCheck generator for arbitrary balanced trees of Booleans. Hint: it may be useful to define a function

```
bTree :: Int -> Gen (Tree Bool)
```

which generates balanced trees of a specified height. Note that there are three kinds of balanced trees of height $n > 0$: those with subtrees of equal height, those where the left subtree is one higher than the right, and vice-versa.

Solution

```
balTree = sized bTree

bTree n | n <= 0    = return Leaf
        | otherwise =
   do a <- arbitrary
      let m = n - 1
      (leftHeight,rightHeight) <- elements [(m,m-1), (m,m), (m-1,m)]
      left  <- bTree leftHeight
      right <- bTree rightHeight
      return $ Node a left right
```

```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 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)
-----------------------------------------
```

```haskell
-- 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)
```

```haskell
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.
```