EXAM
Introduction to Functional Programming
TDA555/DIT440

DAY: 2018-10-31 TIME: 14:00–18:00 PLACE: "Lindholmen"-salar

Responsible:	David Sands 0737207663 Will visit the exam rooms between 15:00 and 15:30, and again between 16.15 and 16.45]
Aids:	An English (or English-Swedish, or English-X) dictionary
Grade:	Completing Part I gives a 3 or a G; Part I and Part II are both needed for a 4, 5, or VG

This exam consists of two parts:

Part I (7 small assignments)

- Give good enough answers for 5 assignments here and you will get a 3 or a G
- (Points on Part II can be counted towards Part I if needed, but this is very unlikely to happen in practice.)

Part II (2 more advanced assignments)

- You do not need to solve this part if you are happy with a 3 or a G!
- Pass Part I and one assignment of your choice here and you will get a 4
- Pass Part I and both assignments here and you will get a 5 or a VG

Please read the following guidelines carefully:

- Begin each assignment on a new sheet
- Write your number on each sheet
- Write clearly; unreadable = wrong!
- Comments (if needed) can be given in Swedish or English
- You can make use of the standard Haskell functions and types given in the attached list (you have to implement other functions yourself if you want to use them)
- You do **not** have to import standard modules in your solutions. You do not have to copy any of the code provided.

Good Luck!

Part I

Correct answers to 5 out of the following 7 assignments gives a pass on the exam.

1

Given the following function

```
q1 :: Int -> [[Int]]
q1 n | n <= 0    = []
     | otherwise = replicate n 1 : q1 (n `div` 2)
```

What is the result of evaluating the following expression:

```
q1 4
```

You only need to write the answer. An answer with an incorrect type will be considered incorrect!

2

The following function which works a bit like `lookup` but returns a list of all the things found:

```
lookupAll :: Eq a => a -> [(a,b)] -> [b]
lookupAll key []          = []
lookupAll key ((k,v):kvs)
              | key == k  = v : lookupAll key kvs
              | otherwise =     lookupAll key kvs
```

For example

```
lookupAll "lucky" [("lucky",6),("unlucky",7),("lucky",8),("beastly",666)]
```

gives `[6,8]`.

Give a definition of `lookupAll` using just a list comprehension. You may use the equality operator `(==)` but no other standard functions. You must not use recursion.

3

In this question you should define a data type to represent a ticket for an amusement park (e.g. Liseberg). Each ticket has the following properties:

- A ticket is valid either for a child or for an adult; a child ticket states the age of the child.
- A ticket is either a season ticket or a day ticket; a day ticket states the date on which it is valid, and a season ticket states the year for which it is valid.

In this question you have to define two things. *Firstly* define a data type

```
data Ticket = ...
```

using the following given types to represent dates and years:

```
type Age   = Int
type Year  = Int
type Month = Int
type Day   = Int
```

and any other helper types that you find useful.

Your type should precisely represent the possible ticket types described above, and nothing more. For example, an adult ticket should *not* include an age (because that is only for child tickets), and a season ticket should not contain a date.

Secondly, give an example of a *value* of type `Ticket` which represents this year's season ticket for a child aged 13.

```
exampleTicket :: Ticket
exampleTicket = ...
```

4

The following data type `Expr` represents arithmetic expressions with multiplication, addition, and subtraction:

```
data Expr = Num Int | Add Expr Expr | Mul Expr Expr
 deriving (Eq,Show)
```

The data type `Ex` defined below also represents arithmetic expressions, but in a slightly different way:

```
data Ex = NumEx Int | BinEx Op Ex Ex

data Op = AddOp | MulOp
```

Define a function

```
convert :: Expr -> Ex
```

which converts from an expression of type `Expr` to the corresponding expression in the type `Ex`.

Hint: if your definition is not recursive then it is wrong.

5

Define a quickCheck property which relates the function `lookup` with the function `lookupAll` defined in question 2.

Note: There is more than one meaningful property that relates these two functions, but you should not just consider the special case when using `lookup` returns `Nothing`.

Note also that your property should never give an error for any inputs (because since the functions are correct, that can only mean that the property is wrong!).

6

Suppose that you have a function `guests :: DayNumber -> Int` that on a given day (a positive integer, where 1 is the first day of operation of the park) returns the number of visitors to an amusement park, where

```
type DayNumber = Int
```

In this question the definition of the function `guests` is not important.

Consider the following functions:

```
closedDays, busyDays :: DayNumber -> DayNumber -> [DayNumber]

closedDays n m = [d | d <- [n..m], guests d == 0]
-- the days in the range for which the restaurant had no guests

busyDays n m   = [d | d <- [n..m], guests d > 2000]
-- the days in the range with more than 2000 guests
```

These functions look very similar. The question has two parts:

1. Define a more general function

   ```
   days :: (DayNumber -> Bool) -> DayNumber -> DayNumber -> [DayNumber]
   ```

   such that `days p n m` returns a list of the days in the range `n` to `m` for which the property `p` is true (where `p` is a function of type `DayNumber -> Bool`).

2. Define `closedDays` and `busyDays` using the more general function `days` (and any small helper functions that you find useful).

You are not required to write any comments or explanations, just Haskell code.

Write a QuickCheck generator

```
spam :: Gen String
```

which is a generator for a random email address such as "bob7@gmail.com", "alice99@hotmail.com", or "dave@gmail.com". You should assume two Haskell definitions

```
names, emailProviders :: [String]
```

For example, we might have

```
names          = ["alice", "bob", "dave"]
emailProviders = ["gmail","yahoo","hotmail"]
```

Note that these are just lists, not generators. Using these definitions together with the standard QuickCheck functions (listed in the appendix) you should generate email addresses like the examples above, combining names, email providers (all assumed to be `.com`) with zero, one or two digits after the name (so it should be possible to generate all of the names in the examples above).

Part II

You do not need to work on this part if you only want to get a 3 or a G (although a correct answer to part II can be used instead of a question in part I).

8

A decision tree is defined as follows:

```
data DTree = Decision Answer | Q Question DTree DTree

type Question = String
type Answer   = String
```

Define a function

```
attributes :: DTree -> [(Answer,[(Question,Bool)])]
```

which builds a table from a decision tree, where for every answer in the tree there is list of the questions which led to that answer, and whether the question was answered yes (`True`) or no (`False`).

For example consider the tree:

```
ex = Q "Is it Raining?" wet notWet
    where wet    = Decision "Take the bus"
          notWet = Q "Is it more the 2km?" (Decision "Cycle") (Decision "Walk")
```

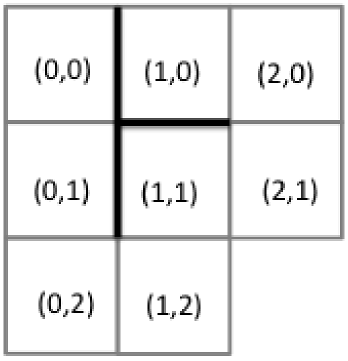
Then `attributes ex` should give the answer (reformatted for readability):

```
[
 ("Take the bus",[("Is it Raining?",True)]                            ),
 ("Cycle"       ,[("Is it Raining?",False),("Is it more the 2km?",True)] ),
 ("Walk"        ,[("Is it Raining?",False),("Is it more the 2km?",False)])
]
```

You may assume that each Answer appears at most once in a given decision tree.

A maze is a collection of squares, each given by an x and a y coordinate, and some walls between squares. Each wall is to the North, South, East, or West of a given square.

An example maze is given in the picture below where the thick black lines are the walls:



In this question we will represent a maze using the following types

```
type Maze = [(Position, [Direction])]
type Position = (Int,Int)

data Direction =  N | S | E | W
  deriving Eq
```

The maze pictured above can be represented as follows:

```
  exampleMaze = [((0,0),[E]), ((1,0),[S]), ((2,0),[])
                ,((0,1),[E]), ((1,1),[]),  ((2,1),[])
                ,((0,2),[]),  ((1,2),[])
                ]
```

Note that the order of the elements in the list are unimportant. If there is a wall between to adjacent positions e.g. (0,0) and (1,0), then it might either be represented by a wall on the east of (0,0), or on the west of (1,0), or both. You may, however, assume that a position appears at most once in a maze.

A *valid path* in a maze is a list of zero or more positions which are in the maze, and where you can get from one position to the next in the list by taking one step in a direction N, S, E or W, and where there is no wall in the way.

Here is an example of a valid path in `exampleMaze`, and three examples of non valid paths:

```
goodPath = [(0,0),(0,1),(0,2),(1,2),(1,1)]
badPath1 = [(0,0),(1,0)] -- crosses a wall
badPath2 = [(1,2),(2,2)] -- goes outside the maze
badPath3 = [(1,2),(2,1)] -- needs two steps
```

Define a Haskell function

```
validPath :: Maze -> [Position] -> Bool
```

which checks whether the given path is valid in the given maze.

```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,isNothing       :: Maybe a -> Bool
isJust (Just a)        =  True
isJust Nothing         =  False

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)
```

```haskell
-- 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\n"
-- 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.
------------------------------------------
-- Useful IO function
putStr, putStrLn  :: String -> IO ()
getLine       :: IO String
readFile      :: FilePath -> IO String
writeFile     :: FilePath -> String -> IO ()
```