EXAM
Introduction to Functional Programming
TDA555/DIT440

DAY: 2017-10-28 TIME: 14:00–18:00 PLACE: SB Multisal

Responsible:	David Sands 0737207663 [Will visit the exam rooms between 15.00 and 15.30]
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 larger 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
- *Tangentbordet är knepigt, men oftast har jag alt under ctrl.*

Good Luck!

Part I

You have to complete 5 out of the following 7 assignments to get a pass on the exam.

1

Given the following function

```
q1 :: [Int]  -> Int
q1 []                    = 0
q1 [x]                   = x
q1 (x:_:xs)              = max x (q1 xs)
```

what would the following expression give in ghci?

```
q1 (map abs [-1,-6,-5,7])
```

2

In this question you should assume that you have a function `rainfall` which computes the rainfall in Gothenburg for a given week (where weeks are numbered from 1 and upwards)

```
type WeekNumber = Int
rainfall :: WeekNumber -> Double    -- assume this function exists
```

A week is considered to be "dry" if the rainfall in that week is less than 5.

Complete the definition of the following function:

```
dryWeeks :: WeekNumber -> Int
dryWeeks n | n < 1     = 0
-- (complete this definition)
```

such that `dryWeeks n` (when `n > 0`) gives the number of dry weeks in the range 1 up to n.

Your solution must be recursive. Solutions that do not use recursion will be considered incorrect. Solutions which always return the value 0 (whether intended as a joke or otherwise) will also be considered incorrect!

In this question you should define a data type to represent a bus ticket of a certain kind described below.

A bus ticket is either a single ticket (a ticket valid for a certain number of minutes) or a period ticket (a ticket that lasts a number of whole days). A single ticket is marked with the date and time when it expires. A period ticket is marked with the date when it expires.

You should use the types `Date` and `Time` given below (although the details of their definitions are not important for this question):

```
type Year   = Int
type Month  = Int
type Day    = Int
type Hour   = Int
type Minute = Int

data Date = Date Year Month Day
data Time = Time Hour Minute
```

Your task is (only) to complete the following definition:

```
data BusTicket = ...
```

Note: you do not have to define any functions, or write any `deriving ...`.

4

The following data type represents arithmetic expressions with multiplication, addition, subtraction and a variable X:

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

data Op = Add | Mul | Subtract
  deriving (Eq,Show)
```

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, `100 - X` would be represented by the expression

```
ex4 = BinOp Subtract (Num 100) X
```

Then `removeSub ex4` should give

```
BinOp Add (Num 100) (BinOp Mul (Num (-1)) X)
```

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

Hint: a correct solution must use recursion for *every* sub-expression in order to remove *all* subtraction operators.

5

The standard function `isPrefixOf` tests whether a given list is a prefix of another. For example the following expression is true:

```
prop_isPrefixOf = "hell" `isPrefixOf` "hello"
               && []    `isPrefixOf` [1,2,3]
               && [1,2] `isPrefixOf` [1,2]
               && not ([2,3] `isPrefixOf` [1,2,3])
```

Define a quickCheck property

```
prop_take :: Int -> String -> Bool
```

which relates the function `isPrefixOf` with the function `take :: Int -> [a] -> [a]`. Your definition must use the two arguments as part of the test (so it is not OK to write a definition like `prop_isPrefixOf` which just gives a fixed number of examples).

6

Consider the following code:

```
data Suit = Hearts | Clubs | Diamonds | Spades
  deriving (Eq,Show)
data Rank = Numeric Int | Jack | Queen | King | Ace
  deriving (Eq,Show)
data Card = Card Rank Suit
  deriving (Eq,Show)

isRed, isDiamond :: Suit -> Bool
isRed s              = s == Hearts || s == Diamonds
isDiamond s          = s == Diamonds

isAce, isLow :: Rank -> Bool
isAce r              = r == Ace

isLow (Numeric n)    = n < 5
isLow _              = False

lowDiamonds cs = [Card r s | Card r s <- cs,  isLow r && isDiamond s ]

redAces cs     = [Card r s | Card r s <- cs,  isAce r && isRed s ]

lowRedCards cs = [Card r s | Card r s <- cs,  isLow r && isRed s ]
```

The last three functions in this code contain a lot of “cut-and-paste” repetition. Define a function

```
selectCards :: (Rank -> Bool) -> (Suit -> Bool) -> [Card] -> [Card]
```

which generalises these three functions, so that the following property holds:

```
prop_selectCards cs = lowDiamonds cs == selectCards isLow isDiamond cs
                   && redAces     cs == selectCards isAce isRed cs
                   && lowRedCards cs == selectCards isLow isRed cs
```

Note: to check such a property with quickCheck we would need to define generators for cards. You do not need to worry about that.

7

Give the definition of a QuickCheck generator

```
quadlist :: Gen [Integer]
```

for lists of Integers, where for every list generated, the length of the list is a multiple of 4. I.e., the generated lists contain 0 numbers, or 4 numbers, or 8 numbers, or 12 numbers, and so on. Hint: QuickCheck function `vectorOf :: Int -> Gen a -> Gen [a]` which generates a list of a specific length, as well as the generator `arbitrary` may be useful, but `replicate` or `sized` should probably *not* be used.

Hints: (i) don’t make the common mistake of trying to apply a function of type `Integer -> a` to something of type `Gen Integer`, and (ii) don’t forget that you can work with things of type `Gen Integer` using do-notation.

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

The following definitions represent a shape composed of coloured squares arranged in a grid. This can be modelled as a list-of-lists, one for each row:

```
data Shape = S [Row]
type Row = [Square]

type Square = Maybe Colour
data Colour = Black | Red | Green   deriving Eq
```

For example, a black L-shape might be represented by the following:

```
lshape = [[x,o]
         ,[x,o]
         ,[x,x]] where x = Just Black
                       o = Nothing
```

An alternative way to represent a shape is to use a coordinate system. Each coloured part of the shape is represented by a coordinate of a position in the grid, and the colour at that coordinate:

```
type AltShape = [Point]
data Point = P Colour (Int,Int)  deriving Eq
```

In this representation, the L-shape above could be written:

```
altLshape = map (P Black) [(0,0),(0,1),(0,2),(1,2)]
```

Note that the alternative definition is not exactly equivalent, as does not give us a way to represent the blank squares; for example, all completely blank shapes, whether large or small, will be represented by the empty list. We will not worry about this minor difference in this question.

Define a function

```
fromShape :: Shape -> AltShape
```

that converts from a `Shape` to an `AltShape`, so that for example

```
fromShape lshape == altLshape
```

If your solution produces the correct points but listed in a different order that is also acceptable. You may assume, if necessary, that every row in the original shape has the same number of elements.

Consider the following definition of a binary tree

```
data Tree a = Branch (Tree a) a (Tree a)
            | Leaf
            deriving Eq
```

When using a tree to represent data it is often good if the tree is *balanced*, which means that there are roughly the same number of things in the left sub tree as there are in the right sub tree, for every branch in the tree.

We define the *skew* of a tree to be a measure of how unbalanced it is. Let us first define the *skew of a branch* in a tree: the skew of a branch (a non-negative number) is the difference between the number of things in the left sub-tree compared to the right sub-tree. Now we define the skew of a tree to be zero if the tree is a leaf, and the largest skew of all the branches in the tree otherwise.

For example, consider `tree1 :: Tree String`

```
tree1 = Branch (Branch Leaf "left" Leaf) "top" Leaf
```

The skew of the top branch is 1 and the skew of the left branch is 0, so the skew of `tree1` is 1. Consider `tree2`:

```
tree2 = Branch (Branch Leaf "left" (Branch Leaf "lr" Leaf)) "top" Leaf
```

there are three different branches, with skews 2, 1 and 0, respectively. So the skew of the whole tree is the maximum of these, namely 2.

Define a function

```
skew :: Tree a -> Int
```

which computes the skew of a tree. You may compute the skew in any way you like (i.e. it should be equivalent to the definition given above but it does not have to be defined in the same way).

```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 ()
```