EXAM

Functional Programming

TDA451/DIT141

DAY: December 16, 2008 TIME: 14.00 – 18.00 LOCATION: "Maskin-salar"

Responsible:	David Sands (0737 207 663)
Result:	Published 13 January, at the latest
Aids:	An English (or English-Swedish, or English-X) dictionary
Grade:	There are 4 Questions (with 12+9+17+12 = 50 points); a total of at least 23 points guarantees a pass

Please read the following guidelines carefully:

- Please read through all Questions before you start working on the answers
- Begin each Question on a new sheet
- Write clearly; unreadable = wrong!
- Full marks are given to solutions which are short, elegant, efficient, and correct. Less marks are given to solutions which are unnecessarily complicated or unstructured
- For each part Question, if your solution consists of more than 2 lines of Haskell code, include a short description of what your intention is with your solution
- You can use any standard Haskell function in your solution --- a list of some useful functions is attached
- 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!

Good Luck!

Question 1–List Programming (total 12p)

In this Question, you will solve some list programming exercises in Haskell.

(4p) **Part A** Define a function:

```
sameElems :: Eq a => [a] -> [a] -> Bool
```

that, given two lists, checks if the two lists have the same elements. Each element is supposed to occur equally often in both lists for the answer to be True.

Full points are only given if the type of the function is as general as given here.

Example:

```
Main>sameElems "apa" "aap"
True

Main>sameElems [1,2,3] [3,2,1,3]
False
```

(4p) **Part B** Define a function:

```
selections :: [a] -> [(a,[a])]
```

that, given a list, calculates all ways of removing one element from that list. The result is a list of pairs, where each pair contains the chosen element, plus the rest of the list. The chosen elements are taken in order, starting with the head of the list.

Full points are only given if the type of the function is as general as given here.

Examples:

```
Main>selections "cykel"
[('c',"ykel"),('y',"ckel"),('k',"cyel"),('e',"cykl"),('l',"cyke")]

Main>selections [4,1,0]
[(4,[1,0]),(1,[4,0]),(0,[4,1])]
```

(4p) **Part C** Define two "QuickCheck style" properties characterising the function `selections` above:

(1) Combining the first elements of all the result pairs should give back the original list. Example: Taking the first elements of the pairs in the second example above gives [4,1,0], which is the original list.

(2) For each pair in the selections, putting back the chosen element in the rest of the list should give a list with the same elements as the original list. Example: For the last pair (0,[4,1]) in the second example, putting back 0 in [4,1] gives [0,4,1], which has the same elements as the original list.

Hint: For the second property, you may want to use the function `sameElems`.

Question 2 – Programming Style (Total 9p)

The mobile phone operator "Telefart" charges for phone calls according to the following plan: *Each call incurs a 40 cent connection fee. In addition, after the first 10 seconds, each half-minute period costs 45 cents.*

Time is measured in whole seconds and prices are always in whole cents.

Telefart's top coder, Claus Koensson, has written the following Haskell function which calculates the cost of a given call:

```
callCost c | c <= 10   = 40
           | otherwise = 40 + 45 * (1 + (c - 11) `div` 30)
```

Claus's method to calculate the call cost is correct and probably as simple as it can be. Even so, the code is hard to maintain.

(3p) **Part A** Rewrite the code so that it would be easier for the rest of Telefart's programmers to understand and modify the code next time Telefart changes their price plan.

Now consider the following unrelated code:

```
letterTable :: [String] -> IO ()
letterTable []     = do return ()
letterTable (x:xs)
  | all isAlpha x  = do putStrLn (x ++ ": "
                                  ++ show (length x)
                                  ++ " letters")
                        letterTable xs
  | otherwise      = do letterTable xs
```

This is a function that, given a list of strings, filters out the ones that seem to be regular words (by looking if they only contain letters), and for those words prints a table with the word and how many letters that word has.

For example, executing `letterTable ["apa","+","bepa","<html>","cepa"]` produces:

```
apa: 3 letters
bepa: 4 letters
cepa: 4 letters
```

One bad property of the above code is that it mixes pure functional code with IO-instructions.

(4p) **Part B** Give a new implementation of the above function, where you try to separate IO-instructions from pure functions. The result should consist of only one function that produces IO-instructions (that is as small as possible), and one or more pure functions.

(2p) **Part C** Explain briefly why it is a good idea to separate IO-instructions from pure functions in your program. What things can be done with one that cannot be done with the other?

Question 3 – Expressions (Total 17p)

In this Question, we are going to look at datatypes for arithmetic expressions and functions that deal with these.

The simplest expressions we are considering are expressions consisting of numbers (integers), addition, and multiplication:

```
data Expr
  = Num Integer
  | Add Expr Expr
  | Mul Expr Expr
```

Later, we are going to add more constructors, but these will do for now.

(3p) **Part A** Implement an evaluation function for expressions:

```
eval :: Expr -> Integer
```

that calculates the values of expressions.
Example: eval `(Mul (Num 2) (Add (Num 3) (Num 4)))` returns 14.

Suppose we would like to augment our expression datatypes with two more "features":
(i) Variables: Expressions can contain variables with any name, represented as a String: **type** Name = String

(ii) Integer division: Apart from addition and multiplication, we would like to also be able to use integer division as an operation (division that rounds its result downwards).

(2p) **Part B** Extend the datatype with these two constructors

We are now going to adapt the evaluation function to the new datatype. Two extra things need to be taken care of:

(i) The value for each variable needs to be given as an argument. We will use a table `[(Name,Integer)]` for this that associates a value to each variable.

(ii) Evaluation of an expression may fail because of division by 0, which is not mathematically defined, or because the name is not present in the table. We will change the result type of `eval` into a `Maybe` type to deal with this -- the result `Nothing` means that division by 0 occurred or that a name has no associated value.

The final type of eval now becomes:

```
eval :: [(Name,Integer)] -> Expr -> Maybe Integer
```

Before we implement the new `eval`, there is a useful helper function we may make use of. For example, when evaluating `Add x y`, we will first evaluate x and y. However, either or both of these evaluations may result in `Nothing`, in which case

the result should also be `Nothing`. Only when both evaluations result in a `Just`, can we actually compute the final result, which is also a `Just`.

(2p) **Part C.** Implement a function

```
operMaybe :: (a -> b -> c) -> Maybe a -> Maybe b -> Maybe c
```

The expression `operMaybe f ma mb` results in `Just(f a b)` if `ma` is of the form `Just a` and `mb` is of the form `Just b`. Otherwise, if any of its arguments is `Nothing` the result is `Nothing` too.
Example: `operMaybe (+) (Just 4) (Just 5)` results in `Just 9`.

(4p) **Part D.** Implement a the adapted function

```
eval :: [(Name,Integer)] -> Expr -> Maybe Integer
```

Big-sum notation such as $\sum_{i=0}^{n} 2 * i$ represents (in this example) the mathematical expression *2*0 + 2*1 + 2*2 + … + 2*n*. For the final part of the Question you should define a new datatype `BigExpr` for expressions to represent "big-sum" expressions of the general form

$$\sum_{x=e}^{e'} e''$$

where e, e' and e'' are expressions that can be represented by the `Expr` datatype and the variable *x* can be represented by type `Name`.

(6p) **Part E.** Define the new datatype BigExpr, **and** its eval function

```
evalBig :: [(Name,Integer)] -> BigExpr -> Maybe Integer
```

Note that BigExpr does not need to be able to represent nested big sums. If you need to make any assumptions about the expressions that can be represented these assumptions should be clearly stated.

Question 4 – Family Trees (Total 12p)

A kind of family tree can be represented by the following Haskell datatype:

```
type Name   = String
type Born   = Int
data Family = Fam Name Born [Family]
```

Here is an example Family:

```
duck :: Family
duck = Fam "Uncle Scrooge" 1898
           [ Fam "Donald" 1932 []
           , Fam "Ronald" 1933
                [ Fam "Huey"  1968 []
                , Fam "Duey"  1968 []
                , Fam "Louie" 1968 []
                ]
           ]
```

This value represents the male line of the duck family where Uncle Scrooge, born in 1898, had two sons, Donald and Ronald. Ronald had three children. Neither Donald nor his nephews have any children.

(1p) **Part A** What is the type of the following function?

```
y = maximum . z
    where z (Fam _ born kids) = born : concatMap z kids
```

(2p) **Part B** Describe in a few words what it computes.

(3p) **Part C** Define a function `prop_Family :: Family -> Bool` which checks that every child in a Family is born after his or her parent.

(6p) **Part D** Define a function
`parent:: Name -> Family -> Maybe String`
which computes the name of the parent of a given family member.

Example `parent "Huey" duck` should give `Just "Ronald"`, and `parent "Dave" duck` should give `Nothing`.
For simplicity you may assume that any name occurs only once in a given family.

Appendix – Standard Haskell Functions

This is a list of selected functions from the standard Haskell modules: Prelude, Data.List, Data.Maybe,Data.Char. You may use these in your solutions.

```
--------------------------------------------------------------------------
-- 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

--------------------------------------------------------------------------
-- 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 ()

--------------------------------------------------------------------------
-- 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

------------------------------------------------------------------------
-- functions on pairs

fst             :: (a,b) -> a
fst(x,y)        =  x

snd              :: (a,b) -> b
snd (x,y)        =  y

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[]        = 0
length(_:l)     = 1 + length l

(!!)             :: [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'

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

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 xs       =  foldl1 max xs

minimum []       =  error "Prelude.minimum: empty list"
minimum xs       =  foldl1 min 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, y /= x ]

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]

partition        :: (a -> Bool) -> [a] -> ([a],[a])
partition p xs   = (filter p xs, filter (not . p) xs)

group           :: Eq a => [a] -> [[a]]
-- group "aapaabbbeee" == ["aa","p","aa","bbb","eee"]

isPrefixOf, isSuffixOf   :: Eq a => [a] -> [a] -> Bool
isPrefixOf []      _      =  True
isPrefixOf _       []     =  False
isPrefixOf (x:xs) (y:ys) =  x == y && isPrefixOf xs ys

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
--------------------------------------------------------------------------
```