# RE-EXAM

## Introduction to Functional Programming

TDA555/INN040

---

DAY: 19 January 2007 TIME: 8.30 -- 12.30 PLACE: M-salar

| | |
|---|---|
| **Responsible:** | Koen Lindström Claessen, Datavetenskap |
| **Result:** | Published 31 January, at the latest |
| **Aids:** | An English (or English-Swedish, or English-X) dictionary |
| **Grade:** | There are 4 assignments (with 18 + 21 + 18 + 3 = 60 points); a total of at least 27 points guarantees a pass |

**Please read the following guidelines carefully:**

- Note that the point-scale of this exam is different than the previous exam! You can get more points per question, but also more points are required to pass
- Please read through all assignments before you start working on the answers
- Answers can be given in Swedish or English
- Begin each assignment on a new sheet
- Write your personal number on each 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 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 of an earlier part of an assignment to help solve a later part --- even if you did not succeed in solving the earlier part!

Good Luck!

## Assignment 1 – List Programming (total 18p)

In this assignment, you will solve three list programming exercises in Haskell.

(4p) **Question A**

Define a function:

```
suffixes :: [a] -> [[a]]
```

that, given a list, produces all *suffixes* of the list. A list **as** is the suffix of a list **bs**, if the list **bs** ends with the elements of **as**, in that order.

Example:

```
Main> suffixes "abra"
["abra","bra","ra","a",""]
```

(5p) **Question B**

Define a function:

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

that, given two lists, checks if the second list is contained as a sublist in the first list. (A list **as** is called a *sublist* of a list **bs**, if **bs** can be written as **xs++as++ys**, for some lists **xs** and **ys**.)

Examples:

```
Main> "abracadabra" `contains` "cad"
True
Main> "abracadabra" `contains` "radab"
False
```

Hint : You might find it useful to use the function **suffixes**.

(5p) **Question C**

Define a function:

```
groups :: Int -> [a] -> [[a]]
```

that, given a number k and a list, chops the list up into sublists of length k.

Example:

```
Main> groups 2 [3,1,4,1,5,9]
[[3,1],[4,1],[5,9]]
Main> groups 3 "SEND MORE YOGURT"
["SEN","D M","ORE"," YO","GUR","T"]
```

(4p) **Question D**

Define two QuickCheck properties characterising the function **groups** above: (1) All groups should have the correct length (give special care to the length of the last group!); (2) Combining all groups together should give back the original string.

## Assignment 2 – Summarizing Replies to a Questionnaire (total 21p)

In this assignment, we will develop a Haskell program that can summarize replies to a questionnaire, for example a course questionnaire. The questionnaire is web-based, and stores all replies it has gotten so far in a special file.

An example course questionnaire might look as follows:

1. What is your general impression of the course?
2. What did you think of the exercises classes?
3. What did you think of the fact that there was only one lecture in week 2?
4. What did you think of the lab assignments?
5. How difficult was the course for you?

A reply from one person is modelled by a list of pairs. An example of a reply is the following:

```
aReply :: Reply
aReply = [(1,"very good"), (3,"bad"), (4,"okay")]
```

This models the fact that the person who entered the answers, answered "very good" on question 1, "bad" on question 3, and "okay" on question 4. Note that not all questions have to be answered by a person. For example, the above person did not answer question 2 and 5.

Here is an example of a list of replies:

```
someReplies = [ [(1,"very good"), (3,"bad"), (4,"okay")]
              , [(2,"good"), (3,"bad"), (4,"good"), (5,"difficult")]
              , [(4,"okay"), (5,"very difficult")]
              ]
```

(2p) **Question A**

Give a suitable type definition of the type **`Reply`**.

What is the type of the function **`someReplies`**?

(2p) **Question B**

A reply should not have two answers for the same question. Define a function:

```
validReply :: Reply -> Bool
```

that checks this.

(3p) **Question C**

Define a function:

```
questions :: [Reply] -> [Int]
```

that, given a list of replies, returns all question numbers that were answered in any of the replies. The list should not contain any duplicates, and should contain the question numbers in the right order.

Example:

```
Main> questions someReplies
[1,2,3,4,5]
```

(3p) **Question D**

Define a function:

```
answers :: Int -> [Reply] -> [String]
```

that, given a question number and a list of replies, gathers all answers to this question given in any of the replies. Note: This list *can* contain duplicates!

Examples:

```
Main> answers 3 someReplies
["bad","bad"]
Main> answers 5 someReplies
["difficult","very difficult"]
```

(4p) **Question E**

Define a function:

```
summary :: [Reply] -> [(Int,[(Int,String)])]
```

that, given a list of replies, produces a table, containing, for each question, all answers that were given to that question, and how many times that answer was given. The answers should be sorted in such a way that the most frequent answer comes first.

Example:

```
Main> summary someReplies
[ (1,[(1,"very good")]), (2,[(1,"good")]),
    (3,[(2,"bad")]), (4,[(2,"okay"),(1,"good")]),
      (5,[(1,"very difficult"),(1,"difficult")]) ]
```

We can see for example that the answers to question 3 were 2 "bad"s, and the answers to question 4 were: 2 "okay"s and 1 "good".

(4p) **Question F**

Define a function:

```
createSummary :: FilePath -> IO ()
```

that, given the name of a file containing the replies, prints out a summary of the replies in the file. The format of the replies in the file is in the standard Haskell format. You may assume the existence of a function `read :: String -> [Reply]` that converts a string into a list of replies.

Example: (when given a file with replies, called `someReplies.txt`)

```
Main> createSummary "someReplies.txt"
Q1: 1 very good
Q2: 1 good
Q3: 1 bad
Q4: 2 okay, 1 good
Q5: 1 very difficult, 1 difficult
```

(3p) **Question G**

To get a better overview of the actual answers that are given, we would like to produce an additional summary of the results, where answers like "good" and "very good" (or "difficult" and "very difficult") are grouped together. In this way, it is easier to see what percentage of answers are on the "right side".

Define a function:

```
mild :: [Reply] -> [Reply]
```

that transforms all answers in the given replies into *milder* answers, by removing the word "very" from them.

Example: (after applying the function `mild` to the contents of the file `someReplies.txt`, resulting in the file `someMildReplies.txt`)

```
Main> createSummary "someMildReplies.txt"
Q1: 1 good
Q2: 1 good
Q3: 1 bad
Q4: 2 okay, 1 good
Q5: 2 difficult
```

## Assignment 3 – People and Companies (total 18p)

In this assignment, we will write some Haskell code that can help us manage a company, where people work and get a salary.

We start by modelling people using a Haskell datatype. We define the following:

```
data Person = MkPerson String Double
```

Here, **`MkPerson`** is a constructor with two arguments: the name of the person as a string, and their monthly salary measured in Swedish krona.

(1p) **Question A**

Define a function:

```
anna :: Person
```

that models a person called Anna, who earns 25.000 SEK per month.

(2p) **Question B**

Define a function:

```
showPerson :: Person -> String
```

that represents a person as a string, showing their name and their salary.

Example:

```
Main> showPerson anna
"Anna, 25000.0 SEK per month"
```

The next step is to model a whole company where people work. Companies are usually divided up into *divisions.* Each division has a manager (a person), and possibly a number of divisions that he or she manages. We use the following recursive datatype:

```
data Division = MkDivision Person [Division]
```

A division can consist of only one person, in which case the list of divisions under that person is empty. For example, the division **`MkDivision anna []`** only consists of Anna, by herself. People at the company that are not managing anyone are called *workers.* In the example, Anna would be called a worker.

An example of a larger division is:

```
example :: Division
example = MkDivision angela [ MkDivision anna []
                            , MkDivision per [ MkDivision kalle [] ]
                            ]
```

The above describes a division with one manager, Angela, who manages two sub-divisions: Anna, who works by herself, and Per, who in turn is the manager over the division consisting of Kalle. The workers in the above example are Anna and Kalle.

A whole *company* is simply seen as one big division.

```
type Company = Division
```

You are allowed to make use of the following function.

```
persons :: Company -> [Person]
```

This function, given a company, produces a list of all people in the company.

(3p) **Question C**

Define a function **`showCompany :: Company -> String`** that represents all persons working at the company as a string. Here, each person occurs on one line.

```
Main> putStr (showCompany example)
Angela, 73000.0 SEK per month
Anna, 25000.0 SEK per month
Per, 31000.0 SEK per month
Kalle, 21000.0 SEK per month
```

(4p) **Question D**

Define a function **`workers :: Company -> [Person]`** that produces a list of all workers in a given company. (Remember that a worker is a manager of a division without any other people.)

(4p) **Question E**

Define a function **`giveRaise :: Double -> Company -> Company`** that increases everyone's salary by a given factor. For example, **`giveRaise 1.1`** should give everyone a salary raise of 10%.

(4p) **Question F**

Functions like **`persons`** and **`workers`** are quite common, and their definitions are very much alike. Define a higher-order function **`gather`** that can gather all kinds of information from a company. The function is parameterized by a function that tells it what information to gather.

```
gather :: (Division -> [a]) -> Company -> [a]
```

So, **`gather what`** gathers information from each division in the whole company that is specified by the function **`what`**. An example of the use of **`gather`** is the following alterative definition of the function **`persons`**:

```
persons :: Company -> [Person]
persons comp = gather person comp
  where
    person (MkDivision p _) = [p]
```

Here, the function **`person`** specifies what kind of information to gather from each division.

## Assignment 4 -- Background Knowledge (total 3p)

In this assignment, you have the chance to show us what background knowledge you have picked up during the course.

Please only answer *one* of the following questions. You can choose which one! (Do not pick more than one – I will only look at the first of these questions you decide to answer.)

(3p) **Question A**

Discuss the difference between functions of type `A -> B` and functions of type `A -> IO B`. Are there things that you can do with one that you cannot do with the other? What is the point of separating these two kinds of functions? How does this difference influence the design of your program?

(3p) **Question B**

What are the main differences between the programming language Haskell and the programming language Erlang? What are the main similarities? You may discuss programming language "features" as well as the difference in purpose behind the two.

(3p) **Question C**

In a *sequential* or *imperative* programming language, a programmer expresses a sequence of instructions that should be carried out by the computer, one step at a time. Discuss what the disadvantage of this principle is in a parallel setting (where many things happen at the same time), for example when using a dual core processor. What can the advantage of functional programming be in this context?

## 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

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