Re-exam – Introduction to Functional Programming

TDA555/DIT441, HT-24
Chalmers and Göteborgs Universitet, CSE

Day: 2025-01-07, *Time:* 14:00-18:00, *Place:* Johanneberg

Course responsible

Alex Gerdes (072-9744966). Alex will visit the exam room once around 15:00, and is after that available by phone.

Allowed aids

An English dictionary.

Grading

The exam consist of two parts: a part with seven small assignments and a part with two more advanced assignments; there are in total nine assignments.

- To pass the exam (with a 3) you need to give good enough answers for five out of the nine assignments. An answer with minor mistakes might be accepted, but this is at the discretion of the marker. An answer with large mistakes will be marked as incorrect.
- You do not need to solve the assignments from part II to pass the exam and you are happy with a 3! You are though encouraged to try the assignments from part II: they count to pass the exam, and you may get a higher grade.
- For a 4 you need to pass Part I (five out of seven assignments) and one assignment of your choice from Part II.
- For a 5 you need to pass Part I (five out of seven assignments) and both assignments from Part II.

Notes

- Begin each assignment on a new sheet and write your anonymous code on it.
- You may write your answers in Swedish and English.
- Excessively complicated answers might be rejected.
- *Write legibly!* Solutions that are difficult to read are marked as incorrect!
- 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

1

Given the following definition:

```
fun []     = 0
fun (x:xs) = if even x then x * fun xs else x + fun xs
```

a) What does the expression

```
fun [2, 21]
```

evaluate to? Use equational reasoning and write down the intermediate steps of the evaluation. You may freely choose the evaluation order.

b) What is the type[1] of `fun`?

Solution

```
{-
  fun [2, 21]
= { def fun, second case }
  if even 2 then 2 * fun [21] else 2 + fun [21]
= { app even }
  if True then 2 * fun [21] else 2 + fun [21]
= { def if, select true branch }
  2 * fun [21]
= { def fun, second case }
  2 * (if even 21 then 21 * fun [] else 21 + fun [])
= { app even }
  2 * (if False then 21 * fun [] else 21 + fun [])
= { def if, select false branch }
  2 * (21 + fun [])
= { def fun, first case }
  2 * (21 + 0)
= { addition and multiplication }
  42
-}

fun :: Integral a => [a] -> a  -- also accepted: [Int] ->  Int
```

[1] The type of a function is also called its *type signature*.

2

Implement the function:

```
chop :: Int -> [a] -> [[a]]
chop _ [] = []
chop n xs = let (ys, zs) = splitAt n xs in ys : chop n zs
```

that *chops* a list into sub-lists of a given size n. If the size of the list is not a multiple of n, then the remaining elements are just returned as the last sub-list. The given size n should be greater than zero, but you don't have to take this into account in your implementation. *Hint*: the `Prelude` function `splitAt` may come in handy.

Examples:

```
ghci> chop 2 [1,2,3,4,5,6]
[[1,2],[3,4],[5,6]]

ghci> chop 3 [1,2,3,4]
[[1,2,3],[4]]

ghci> chop 42 [1,2,3,4,5]
[[1,2,3,4,5]]
```

3

This assignment is about *modeling*. You are going to model a game in this exercise, which is called FUPSTER! The game is about building a timeline of computer science legends in chronological order of their birth year. The actual rules of this fictional game are not relevant to the assignment.

The FUPSTER game consists of a list of players and a deck of cards. A card holds the name of a computer science legend (such as Grace Hopper or Alan Turing), the speciality of the legend, and her/his year of birth. A speciality is: Programming, Logic and Types, Security, or Formal Methods. A player has a name, a list of cards, and a number of 'fupsters' that can be modelled as an integer.

Your task it to define a collection of data types that models FUPSTER according to the above description.

Suggested solution

Note, you don't need to use record syntax (which the suggested solution uses), nor have `deriving Show`.

```
data FUPster = FUPster
  { players :: [Player]
  , deck    :: [Card]
  } deriving Show

data Player = Player
  { name     :: String
  , cards    :: [Card]
  , fupsters :: Int
  } deriving Show

data Card = Card
  { legend     :: String
  , speciality :: Speciality
  , birthYear  :: Int
  } deriving Show

data Speciality = Programming | LogicTypes | Security | FormalMethods
  deriving Show
```

4

Now that the creation of FUPSTER (see the previous assignment) has started we want to support the developers by writing some code. You are going to define a function:

```
readCard :: IO (String, Int, String)
```

that reads the name of a legend, the year of birth, and her/his speciality. This `IO` function should ask a user for the name of the legend, the year of birth, and retrieve the legend's speciality using the following given `IO` function:

```
getSpeciality :: String -> IO String
getSpeciality _ = return "Programming"
```

You do *not* need to write this function! But you need to *use* it in your definition of the `readCard` function. This function takes a legend name as argument. So, after you have read the legend name, you need to give it as an argument to the `getSpeciality` function.

The `readCard` returns a tuple with the supplied legend name, year of birth, and retrieved speciality. For example:

```
ghci> readCard
Please give the legend name:
> Alan Turing
Year of birth:
> 1912
("Alan Turing",1912,"Programming")
```

GHCi prints the result of the action `("Alan Turing",1912,"Programming")`, but your solution should not do this.

Suggested solution

```
readCard = do
  putStr "Please give the legend name:\n> "
  name <- getLine
  spec <- getSpeciality name
  putStr "Year of birth:\n> "
  year <- readLn
  return (name, year, spec)
```

5

Consider the following list data type that can store a sequence of integers:

```
data List = Empty | Skip List | Cons Int List deriving Show
```

The list data type is a bit special. We can, in addition to save an element with an integer, also have an element without one, that is, we can skip a place. For example:

```
xs :: List
xs = Cons 1 (Cons 2 (Skip (Cons 4 Empty)))

ys :: List
ys = Cons 1 (Skip (Cons 3 (Skip (Cons 5 Empty))))
```

Write a function that converts a list of type `List` to an ordinary list of `Maybes`.

```
toList :: List -> [Maybe Int]
toList l = case l of
  Empty    -> []
  Skip l   -> Nothing : toList l
  Cons x l -> Just x  : toList l
```

A skipped place must be translated to a `Nothing` and other elements to a `Just`.

Examples:

```
ghci> toList xs
[Just 1,Just 2,Nothing,Just 4]

ghci> toList ys
[Just 1,Nothing,Just 3,Nothing,Just 5]
```

6

In an earlier assignment we introduced the `chop` function, which divides a list into sub-lists of a given size. We want to make sure that this function works as expected by using QuickCheck. You can do this assignment even if you have not solved the earlier assignment.

Define a property that checks that the `chop` function returns sub-lists that have the given size n, with the possible exception of the last sub-list (which can be the first if there is only one sub-list). The last list may have a different size, but not greater than n.

```
prop_chop :: Int -> [Int] -> Bool
prop_chop n xs = let m = abs n + 1 in case chop m xs of
  []  -> null xs
  xss -> all ((== m) . length) (init xss) && length (last xss) <= m
```

Have a look at the `Prelude` functions `init` and `last` in the list of standard functions. Remember that these are partial functions, that is, they are not defined on the empty list. In addition, the first argument to the `chop` function must be a positive integer.

7

Your task is to define the *higher-order* function:

```
iterateWhile :: (a -> Bool) -> (a -> a) -> a -> a
iterateWhile p f x = let y = f x in if p y then iterateWhile p f y else y
```

that takes a predicate (a function returning a `Bool`) and another function as argument and applies the function repetitively as long as the predicate holds. In other words, the function is applied again and again on an initial value, until the predicate returns `False`, in which case the resulting value is returned.

Examples:

```
ghci> iterateWhile (< 10) (+1) 0
10

ghci> iterateWhile ((/= 0) . snd) (\ (x, y) -> (y, mod x y)) (48, 18)
(6,0)
```

Part II

8

Consider the following interface for lookup tables (also called a map), which we introduced in the course:

```
data Map k v = ...

empty    :: Map k v
insert   :: Ord k => k -> v -> Map k v -> Map k v
delete   :: Ord k => k -> Map k v -> Map k v
fromList :: Ord k => [(k, v)] -> Map k v
```

This `Map` data type has two type parameters: one for the keys (`k`) in the table and another one for the values (`v`). Using this `Map` data type we can efficiently lookup a key in a table and collect the associated value. We can create an empty map using `empty`, add and remove elements using `insert` and `delete`, respectively. The `fromList` takes a list of key/value pairs and converts it to a map.

We use the above `Map` data type to model a *matrix* as follows:

```
type Index = (Int, Int)

data Matrix a = Matrix
  { size  :: (Int, Int)
  , table :: Map Index a
  } deriving Show
```

We associate an *index*, defined as a combination of a row number and a column number, with a specific value in the matrix. To represent this, we introduced a type synonym for an index, which is a pair of integer values. This approach offers an advantage over representing a matrix as a list of lists: it allows for quick access to a value at a given index. In contrast, with a list of lists, accessing an element requires linear traversal, as Haskell lists do not support random access. In addition, the `Matrix` data type stores the size of the matrix using a pair of integers: the number of rows and the number of columns.

Your task is to define the following conversion function, which takes a list of lists and returns a `Matrix`:

```
fromLists :: [[a]] -> Matrix a
fromLists []  = Matrix (0, 0) M.empty
fromLists xss = Matrix (n, m) (M.fromList $ zip ixs (concat xss))
 where
  n   = length xss
  m   = length (head xss)
  ixs = [(x, y) | x <- [0 .. n - 1], y <- [0 .. m - 1]]
```

You may assume that the lists in the list of lists are of equal length. Note, the input list can be empty, and that we use zero-based indexing.

9

In this assignment you are going to define a pocket calculator that uses *reverse Polish notation* (RPN). Wikipedia describes RPN as follows:

> *In reverse Polish notation, the operators follow their operands. For example, to add 3 and 4 together, the expression is 3 4 + rather than 3 + 4. The conventional notation expression 3 - 4 + 5 becomes 3 4 - 5 + in reverse Polish notation: 4 is first subtracted from 3, then 5 is added to it.*
>
> *The advantage of reverse Polish notation is that it removes the need for order of operations and parentheses that are required by infix notation and can be evaluated linearly, left-to-right. For example, the infix expression (3 + 4) * (5 + 6) becomes 3 4 + 5 6 + * in reverse Polish notation.*

We start by defining a data type for the supported operations:

```
data Operation
  -- Binary operations
  = Add  -- Addition
  | Sub  -- Subtraction
  | Mul  -- Multiplication
  | Div  -- Division

  -- Unary operation
  | Neg  -- Negation

  -- Memory operations
  | MS   -- Store in memory
  | MR   -- Recall from memory
  deriving Show
```

The calculator has support for the self explanatory basic operations, such as addition and multiplication. In addition, it has support for saving an (single) intermediate result in memory, which can be reused in later calculations. This functionality is common in pocket calculators. See the examples below, which explain how you can use it.

We continue by defining a data type for the instructions we can give to the RPN pocket calculator:

```
data Instruction = Op Operation | Val Double deriving Show
```

An instruction is either an operation or a value. To use the pocket calculator we give it a list of instructions, which are processed from the start to the end of the list.

Your task is to define the following function:

```
calcRPN :: [Instruction] -> Double
calcRPN = go [] Nothing
 where
  go [v] _ [] = v
  go vs mem (instr:rest) = case instr of
```

```
    Val v -> go (v:vs) mem rest
    Op op -> case op of
      Add -> go (app2 vs (+)) mem rest
      Sub -> go (app2 vs (-)) mem rest
      Mul -> go (app2 vs (*)) mem rest
      Div -> go (app2 vs (/)) mem rest
      Neg -> go (app1 vs negate) mem rest
      MS  -> go vs (Just $ head vs) rest
      MR  -> maybe (go vs mem rest) (\v -> go (v:vs) mem rest) mem

app1 (x:vs)   f = f x   : vs
app2 (x:y:vs) f = f y x : vs
```

The function takes a list of instructions, which are in reverse polish notation, and calculates the result. You may assume correct input and don't need to do any error handling.

Here are the examples from Wikipedia:

```
ghci> calcRPN [Val 3, Val 4, Op Sub, Val 5, Op Add]
4.0

ghci> calcRPN [Val 3, Val 4, Op Add, Val 5, Val 6, Op Add, Op Mul]
77.0
```

Some examples using the memory instructions:

```
ghci> calcRPN [Val 1, Val 2, Op Add, Op MS, Op MR, Op Add, Op MR, Op Add]
9.0
```

In the above example we calculate $1 + 2$ and then store the result 3 in memory, we then retrieve the saved result and add it to the previous result, which results in 6, and than repeat the procedure to add 3 again, which finally leads to the end result 9.

Here is another example:

```
ghci> calcRPN [Val 1, Val 2, Op Add, Op MS, Val 3, Val 4, Op Add, Op Mul, Op
↪  MR, Op Mul]
63.0
```

```haskell
{- Some standard functions from Prelude Data.List
   Data.Maybe Data.Char Control.Monad -}
-----------------------------------------------
class Show a where
  show :: a -> String

class Read a where
  read :: String -> a

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, 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,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, sin, cos, tan :: a -> a

class (Real a, Fractional a) => RealFrac a where
  truncate, round, 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]
  -- eval actions from left to right, return results
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
  -- apply action to every element in a list
replicateM :: Monad m => Int -> m a -> m [a]
  -- performs action n times, and return a list

-- 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 Bool ------------------------
(&&), (||)       :: 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 -------------------------
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)

-- 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]         = []
```

```haskell
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 "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 (x:xs) | p x        = dropWhile p xs
                   | otherwise = x: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 ["ap","bep","cep"] == "ap\nbep\ncep"
-- unwords ["ap","bep","cep"] == "ap bep cep"
```

```haskell
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])
```

```haskell
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]

isSpace, isDigit, isAlpha :: Char -> Bool
toUpper, toLower :: Char -> Char

digitToInt :: Char -> Int
-- digitToInt '8' == 8

intToDigit :: Int -> Char
-- intToDigit 3 == '3'

ord :: Char -> Int
chr :: Int  -> Char

-- functions from Test.QuickCheck ---------------
arbitrary :: Arbitrary a => Gen a
-- generator used by quickCheck

choose :: Random a => (a, a) -> Gen a
-- 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 weighted list of generators

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 a size param

-- IO functions ---------------------------------
putStr, putStrLn :: String -> IO ()
getLine :: IO String
readLn :: Read a => IO a

type FilePath = String
readFile :: FilePath -> IO String
writeFile :: FilePath -> String -> IO ()
```