Re-exam – Introduction to Functional Programming

TDA555/DIT441 (DIT440), HT-22
Chalmers and Göteborgs Universitet, CSE

Day: 2023-08-15, *Time:* 08:30-12:30, *Place:* Johanneberg

Course responsible

Alex Gerdes (072-9744966). He will visit the exam room once around 09:30, 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 number 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:

```
f bs = sum (g 0 bs)
 where
  g _ []       = []
  g n (b:bs) = b * 2^n : g (n+1) bs
```

a) What does the expression

```
f [1,0,1]
```

evaluate to? Write down the intermediate steps of your computation. You don't need to have separate steps for the built-in operators `(*)` and `(^)`.

b) What is the type (type signature) of `f`?

2

Your task is to write a function:

```
count :: String -> [(Char, Int)]
```

that counts how often a letter appears in a string. For example:

```
ghci> count "Abba 2 Babble."
[('a',3),('b',5),('e',1),('l',1)]
```

Note that the function should ignore other types of characters, such as spaces or punctuation marks. Furthermore, the function should ignore wheter a letter is in upper- och lower-case, they count as the same letter.

a) First, write a function that groups all letters in sublists:

```
groupLetters :: [Char] -> [[Char]]
```

For example:

```
ghci> groupLetters "Abba 2 Babble."
["aaa","bbbbb","e","l"]
```

Hint: you can compose the `group`, `map`, `isAlpha`, `toLower`, `filter`, and `sort` functions (also listed in the attached overview) to define this function. Note that you don't need to and are free to define this function however you want.

b) Next, use the `groupLetters` function to implement the `count` function:

```
count :: String -> [(Char, Int)]
```

3

This assignment is about modelling *boolean expressions*.

Your tasks are:

a) Design a data type `Truth` suitable for representing any boolean expression built from Boolean values (`True` and `False`), together with variables (represented as a single character) and binary operators `And` ($\land$), `Or` ($\lor$), `Implies` ($\Rightarrow$), and the unary operator `Negation` ($\neg$).

Your data type should not be able represent invalid expressions (e.g. containing operators not listed above). You are free to define additional helper types or type synonyms as needed.

b) Give the definition of the `Truth` expression representing $a \land (b \Rightarrow False)$

4

Consider the following datatype in Haskell that models a person:

```haskell
data Person = Person String Int Bool deriving Show
```

For example, `Person "Leia" 22 True` represents a person by the name "Leia", aged 22, who agrees to receive marketing material.

ChatGPT was asked to write an `IO` function that does the following: ask a user for a name, age, and whether they want marketing material. The function should read all these values, construct a value of type `Person` and return this. If at any step the user types an invalid input, the function should ask for that input again until a valid input is given. A valid age is a non-negative integer, and a valid name is any non empty string, and the answer to the marketing question should be "y" or "n".

ChatGPT produced something similar to the following valid Haskell code:

```haskell
readPerson :: IO Person
readPerson = do
  name <- readName
  age  <- readAge
  yn   <- yesNo
  return $ Person name age yn
 where
  readName = do
    putStr "Name: "
    a <- getLine
    if not (null a) then return a else do
      putStrLn "Invalid Input"
      readName
  readAge = do
    putStr "Age: "
    a <- getLine
    if all isDigit a  then return (read a) else do
      putStrLn "Invalid Input"
      readAge
  yesNo = do
    putStr "Would you like marketing material (y/n): "
    a <- getLine
    if a == "y" || a == "n" then return (convert a) else do
      putStrLn "Invalid Input"
      yesNo
    where
     convert "y" = True
     convert "n" = False
```

Notice how the three local functions all do a very similar task. Rewrite this code to remove this ugly "cut and paste" by creating a suitable higher-order function which (with the right parameters) can do the job of each of them.

5

In this assigment you are going to define an *abstract data type* for sets. Your solution should use the following internal representation for a set:

```
data Set a = Set [a]
```

The idea here is that the list contains the elements in the set.

Define an abstract data type for a set in terms of the following functions:

```
empty        :: Set a                             -- create an empty set
mkSet        :: Eq a => [a] -> Set a              -- make a set from the elements in a list;
                                                  -- assume that the input is a finite list

elemSet      :: Eq a => a -> Set a -> Bool        -- tests whether an element is in the set
union        :: Eq a => Set a -> Set a -> Set a   -- take the union of two sets
intersection :: Eq a => Set a -> Set a -> Set a   -- take the intersection of two sets;
                                                  -- that is, a set with elements present
                                                  -- in both input sets
```

6

In assignment 2 we introduced the `count` function that counts the number of occurences of letters in a string. We want to make sure that the function works as aspected by defining some QuickCheck properties. You can do this assigment even if you have not solved assigment 2.

Your tasks are:

a) Write a property that validates that the sum of all the occurences of the different letters in a string is not larger than the total number of characters in that string:

```
prop_notLarger :: String -> Bool
```

b) The letters returned by the `count` function should of course be present in the input string, write a property that checks this:

```
prop_contains :: String -> Bool
```

Remember that the `count` function does not distinguish between capital and small letters.

7

Suppose we define a simple computer file system as a tree of folders:

```
type Name = String
type File = String

data Folder = Folder
  { name    :: Name
  , files   :: [File]
  , subdirs :: [Folder]
  } deriving Show
```

A folder has a name, a list of file names stored in that folder, and a list of sub-folders (also called sub-directories).

Your tasks are:

a) Define a higher-order function that applies a given function to every file in a file system:

```
mapFolder :: (File -> File) -> Folder -> Folder
```

b) Use the `mapFolder` function to write a function that converts all file names to lower case:

```
makeLower :: Folder -> Folder
```

Hint: the `toLower` function may be useful.

Part II

8

In this assignment you will implement (a part of) a *line editor*, which reads keyboard presses and produces a corresponding text string. Keyboard presses are represented by the following data type:

```
data Key = Chr Char | Del | GoLeft | GoRight | Copy | Paste
  deriving (Eq, Show)
```

The semantics of the keyboard presses are:

- `Chr`: represents a normal visible character,
- `Del`: represents the deletion key, which deletes the character to the left of the cursor (if possible),
- `GoLeft`: moves the cursor to the left (if possible),
- `GoRight`: moves the cursor to the right (if possible),
- `Copy`: copies the word to the *left* of the cursor and stores it in a clipboard. A word is a sequence of characters not including a space character.
- `Paste`: pastes the characters stored in the clipboard (if any) to the *right* of the cursor (the cursor remains at the same position).

Your task is to complete the following definition of function `run` by providing definitions for `State`, `startState`, `showState` and `runKey`:

```
data State = ...                     -- define the state of the editor

startState :: State                  -- the starting state
showState  :: State -> String        -- display the editor state
runKey     :: State -> Key -> State  -- update the editor state according to
                                     -- one key press
run :: [Key] -> String
run  = showState . foldl runKey startState  -- do not change this definition
```

so that given a list of keys, `run` computes the resulting text string. For example:

```
ghci> run [Chr '>', Chr ' ', Chr 'f', Chr 'p', Chr 'x', Del, Copy, GoLeft,
 ↪  Paste]
"> ffpp"
```

9

Recall the data definition for a simple computer file system from assignment 7:

```
type Name = string
type File = string

data Folder = Folder
  { name    :: Name
  , files   :: [File]
  , subdirs :: [Folder]
  } deriving Show
```

We continue to define some functions on this data type. Your tasks are:

a) define a 'fold' function for the `folder` data type:

```
foldFolder :: ((Name, [File]) -> b -> b) -> b -> Folder -> b
```

the general idea is that the sub-folders in the file system are combined from left to right (just like `foldr` on lists), and also from top to bottom.

b) use the `foldFolder` function to write a function that returns the all the files stored in the file system in a list:

```
allFiles :: Folder -> [File]
```

```haskell
{- Some standard functions from Prelude Data.List
   Data.Maybe Data.Char Control.Monad -}
---------------------------------------------
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, 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]
sequence      = foldr mcons (return [])
  where mcons p q = do x <- p
                       xs <- q
                       return (x:xs)

sequence_     :: Monad m => [m a] -> m ()
sequence_ xs = 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 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]         = []
init (x:xs)      = x : init xs
```

```haskell
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])
partition p xs = (filter p xs, filter (not . p) xs)
```

```haskell
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

------------------------------------------
-- Useful 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.
```