Re-exam – Introduction to Functional Programming

TDA555/DIT440, HT-21
Chalmers och Göteborgs Universitet, CSE

Day: 2022-01-03, *Time:* 14:00-18:00, *Place:* HB1 and HB3, Johanneberg

Course responsible

Alex Gerdes (031-772 6154). He will visit the exam room once between 15:00 and 15:30, and after that is 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; in total there are nine assignments.

- To pass the exam (with a 3 or a G) 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 or G! 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 definitions:

```
add (a, b, c) = a + b + c

combine (x:y:z:zs) = (x, y, z) : combine (y:z:zs)
combine _          = []
```

a) What does the expression `map add (combine [3,2,1,-3,5])` evaluate to? Write down the intermediate steps of your computation. If the type of your answer is incorrect then your solution will be considered incorrect.

b) What are the types (type signatures) of `add` and `combine`?

2

In this assignment you are going to define an own implementation of the *square root* function. You need to implement a function `approxSqrt` that can approximate $\sqrt{x}$ for any value x.

Consider the following two facts about the square root:

1. if y is a good approximation of $\sqrt{x}$ then $\frac{y+\frac{x}{y}}{2}$ is a better approximation,
2. the value 1 is an approximation of $\sqrt{x}$ (but not so good).

We will say that the approximation of $\sqrt{x}$ is *good enough* when y^2 is close to x. More precisely, when $|y^2 - x|$ is at most some given threshold.

a) Use the above two facts to implement a function:

```
approxSqrt :: Double -> Double -> Double
```

using *guards* such that `approxSqrt eps x` returns a value that is a good enough with respect to the given threshold `eps`. For example:

```
ghci> approxSqrt 0.001 5
2.2360688956433634
```

Hint: use a recursive helper function.

b) Maybe we don't know in advance yet when the approximation is good enough, and instead we just want a list of ever more precise approximations of $\sqrt{x}$. Write a function:

```
approxSqrts :: Double -> [Double]
```

that produces such a list.

3

Consider the `Prelude` function `iterate`:

```
iterate :: (a -> a) -> a -> [a]
iterate f x = let y = f x in y : iterate f y
```

which applies a function `f` repeatedly starting with the initial value `x` and returning the results of each application in an infinite list. For example:

```
ghci> take 10 (iterate (* 2) 1)
[1,2,4,8,16,32,64,128,256,512]
```

We want to create some useful variations on this function.

a) Write a function:

```
iterateN n f x = ...
```

that applies the function `f` a given number (`n`) of times, starting with the value `x`. For example:

```
ghci> iterateN 3 (*2) 1
8
```

You must define the `iterateN` function using explicit recursion (on `n`) and may *not* use the original `iterate` function.

b) Implement a function:

```
iterateWhile p f x = ...
```

that applies the given function `f` repeatedly while the predicate `p` (a function returning a boolean value) still holds. The iteration of function applications starts on the value `x`. For example:

```
> iterateWhile (< 100) (* 2) 1
128
```

Again, you are not allowed to use the original `iterate` function.

c) Give the types (type signatures) for `iterateN` and `iterateWhile`.

4

Consider the following data type definition that models an electronic billboard:

```
type Pixel = (Int, Int)

data BillBoard = BB { size :: (Int, Int), actives :: [Pixel] }
```

A billboard, which can be regarded as a matrix of pixels, consists of its size and a list of active pixels. The size field of a billboard is a tuple of integers where the first element is the number of rows, and the second element the number of columns. A pixel is a pair of integers which denotes its place (row and column) on the billboard. For example, `(0, 3)` is found on the first row and fourth column. We use zero-based indexing, that is, index `(0, 0)` points to the pixel on the first row and first column.

Using the above data definition we can make an example billboard:

```
lambda :: BillBoard
lambda = BB (4, 10) [(3,1),(2,2),(1,3),(0,4),(0,5),(2,4),(3,5),(4,6)]
```

Your task is to write a `Show` instance for the `BillBoard` data type, such that the billboard is displayed as a matrix of pixels. For example:

```
ghci> putStr (show lambda)
....##....
...#......
..#.#.....
.#...#....
```

An active pixel is represented by the character `'#'` and non-active pixels by a dot `'.'`.

Hint: use a nested list comprehension and define a help function for converting a pixel to a character.

5

Recall the `BillBoard` data type from the previous assignment:

```
type Pixel = (Int, Int)

data BillBoard = BB { size :: (Int, Int), actives :: [Pixel] }
```

We want to let a user to create a billboard of a specific size. You are going to define a function that first asks a user for the number of rows and number of columns. The function then creates an empty billboard (i.e., no active pixels) of the specified size, prints the empty billboard, and finally returns the newly created billboard.

Write the function above with the following type signature:

```
createBillBoard :: IO BillBoard
```

The following excerpt shows an example interaction:

```
ghci> bb <- createBillBoard
Number of rows?
> 2
Number of columns?
> 4
Created the following billboard:
....
....
```

You may assume a correct `Show` instance for the data type `BillBoard`, even if you have not implemented it. You can do this assignment independent from the other assignments.

6

The function `filter` from the `Prelude` takes a predicate and list as arguments and filters all elements from the list for which the predicate does not hold. In other words, the `filter` function returns the elements from an input list for which a given predicate holds. For example:

```
ghci> filter even [1..10]
[2,4,6,8,10]
```

Your task is to test the implementation of the `filter` function using QuickCheck for a given predicate (i.e., the properties will take a predicate as argument):

1. Write a property that checks that the result list cannot be longer than the input list.
2. Write a property that verifies that all elements in the result list satisfy the given predicate.

7

A URI (Uniform Resource Identifier) can be used to identify a particular, often electronic, resource. A URI consists of the following parts:

- a *scheme*, which can be one of the following: http, https, ldap, mailto, ftp or news,
- an *authority*, which in turn consists of the following components:
  - an optional *user name*,
  - a *host* name,
  - an optional *port number*,
- an optional *path*, which is a sequence of segments, where a segment is a string,
- an optional *query*, which is a collection of key-value pairs,
- and finally, an optional *fragment*, which is a string.

All non-optional parts are mandatory and need to be specified when constructing a URI. The next figure shows a number of example URIs:

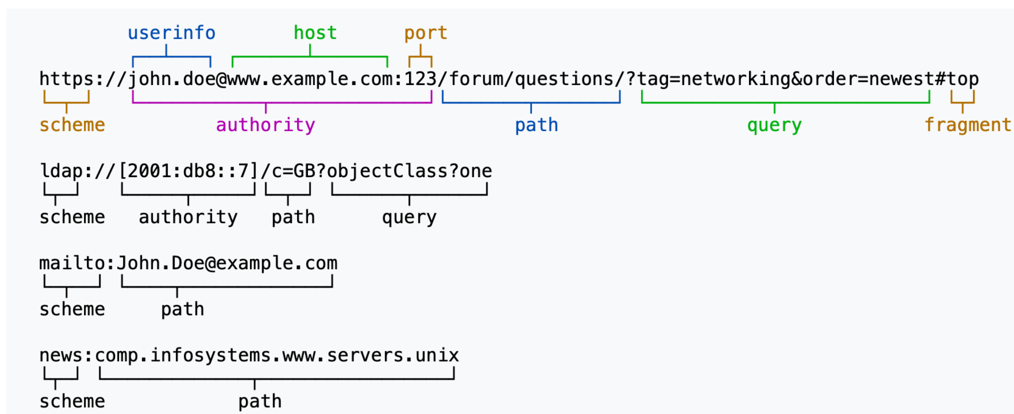

Figure 1: Example URIs (source: Wikipedia)

Your task is to define a collection of data types (and/or type synonyms) that models a URI and all its components as precisely as possible.

Part II

8

We have a radio-controlled car that accepts four different commands: forward, backward, turn left, and turn right. When the car turns left or right, it always turns exactly 90 degrees. Your assignment is to write a computer simulator for the car's movement.

We can model the four commands as a data type:

```
data Command
  = Forward  Int
  | Backward Int
  | TurnLeft
  | TurnRight
```

The integer argument to `Forward` and `Backward` denotes the distance the car should drive in the current direction.

Implement a function:

```
destination :: [Command] -> (Int, Int)
```

that, given a list of commands, computes the position of the car after following these commands. The original position of the car is `(x, y) == (0,0)`, and it is facing "upwards" in the sense that going forward from the start position will increase its `y` position.

For example:

```
ghci> destination [Forward 20, Backward 10, TurnRight, Forward 100]
(100,10)

ghci> destination [Forward 20, Backward 5, TurnLeft, Forward 100]
(-100,15)
```

9

Consider the following data types that model a small expression language with integer and boolean literals, addition, and multiplication. In addition, the expression language also supports a comparison operator (greater than) and an 'if-then-else' expression, which takes a condition (which is also an expression) and two subexpressions that model the 'then' and 'else' branch respectively.

```
data Expr
  = Lit Literal          -- Literal, either integer or boolean
  | Add Expr Expr        -- Addition
  | Mul Expr Expr        -- Multiplication
  | Gth Expr Expr        -- Comparison operator
  | If  Expr Expr Expr   -- if-then-else
```

```
data Literal
  = N Int               -- Integer literal
  | B Bool              -- Boolean literal
```

Using these data types we can define some example expressions:

```
-- Smart constructor for integers
num :: Int -> Expr
num = Lit . N

-- Smart constructor for booleans
bool :: Bool -> Expr
bool = Lit . B

-- Example expressions
e1, e2 :: Expr
e1 = (num 4 `Add` num 5) `Mul` num 2                     -- (4 + 5) * 2
e2 = If (e1 `Gth` (num 4)) (num 3) (num 4 `Add` num 1)    -- if e1 > 4 then 3 else 4 + 1
```

The above examples are well-formed, which means that they don't contain any type errors and we can evaluate these expressions and get a proper result. The expression language model also allows for ill-formed expressions, for example:

```
e3_bad, e4_bad, e5_bad, e6_bad :: Expr
e3_bad = num 42 `Add` bool False                         -- 42 + False
e4_bad = If e2 (num 3) (num 4)                           -- if e2 then 3 else 4
e5_bad = If (num 4 `Gth` num 2) (num 3) (bool True)      -- if 4 > 2 then 3 else True
e6_bad = If (bool False) (num 42) (num 3 `Mul` bool True)  -- if False then 42 else 3 * True
```

These expressions contain so-called *type errors*. For example, expression `e3_bad` adds an integer to a boolean value; in `e4_bad` the condition is of integer type where it should be a boolean; `e5_bad` has different types in the `then` and `else` branches; and finally in `e6_bad` we multiply an integer with a boolean (`3 * True`).

Your task is to write a type check function with the following type:

```
typecheck :: Expr -> Bool
```

This function checks if the expression is type correct. If the given expression contains a type error it will return `False` otherwise it will return `True`.

```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 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           :: 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 => (a -> b) -> m a -> m b
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
-----------------------------------------
-- * 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

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