Exam – Introduction to Functional Programming

TDA555/DIT441, HT-25
Chalmers and Göteborgs Universitet, CSE

Day: 2025-10-28, *Time:* 14:00-18:00, *Place:* Johanneberg

Course responsible

Alex Gerdes (072-9744966). Alex will visit the exam room once around 15:00, and Alex 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:

```
f (x:y:ys) = x : g ys
f _        = []

g (x:y:ys) = y : f ys
g _        = []
```

a) What does the expression

```
f [1,2,3,4]
```

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

b) What is the (most general) type[1] of `f`?

Solution

```
{-
  f [1,2,3,4]
= { def f, fist case }
  1 : g [3,4]
= { def g, first case }
  1 : 4 : f []
= { def f, second case }
  1 : 4 : []
= { list notation (optional) }
  [1, 4]
-}

f :: [a] -> [a]  -- also accepted: [Int] ->  [Int]
```

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

2

Implement the function:

```haskell
substitute :: [(Int, Int)] -> [Int] -> [Int]
substitute _   []     = []
substitute env (x:xs) = case lookup x env of
  Just y  -> y : substitute env xs
  Nothing -> x : substitute env xs

-- alternative
substitute' env = map (\x -> maybe x id (lookup x env))
```

The function `substitute` takes a list of integer pairs and a list of integers. Each pair `(x, y)` represents a substitution rule, meaning that every occurrence of `x` in the list should be replaced with `y`. If an element in the list does not appear as a key in the list of pairs, it should remain unchanged.

Your function should apply the substitutions to every element in the list and return the resulting list. *Hint*: use the `lookup` function.

Examples:

```
ghci> substitute [(1,10), (3,30)] [1,2,3,4]
[10,2,30,4]

ghci> substitute [] [1,2,3]
[1,2,3]

ghci> substitute [(5,0)] []
[]
```

3

A restaurant has a name, a list of reservations, and a Michelin star rating ranging from zero to three stars. A Michelin rating can only take those values, it should not be possible to assign, for example, five stars.

Each reservation records the guest's name, the number of guests, and the date and time of the booking, represented simply as a string. In addition, each reservation should include a list of dietary restrictions. A dietary restriction can be vegan, vegetarian, or lactose intolerant.

Your task is to define a collection of data types that models the restaurant booking system 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 Restaurant = Restaurant
  { name         :: String
  , stars        :: Stars
  , reservations :: [Reservation]
  } deriving Show

data Stars = None | One | Two | Three deriving Show

data Reservation = Reservation
  { guest  :: String
  , number :: Int
  , at     :: String
  , restrictions :: [Restriction]
  } deriving Show

data Restriction = Vegan | Vegetarian | Lactose deriving Show
```

4

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

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

The function should read input from the user: the name of the person making the reservation, the number of guests, and the desired date and time. The function does not need to ask for any dietary restrictions.

The `makeReservation` function should return a three-tuple containing the supplied name, number of guests, and date/time. For example:

```
ghci> makeReservation
Name: Alex
Number of guests: 5
Date and time: Tuesday at 19:00
("Alex",5,"Tuesday at 19:00")
```

Note that GHCi automatically prints the result of the action `("Alex",5,"Tuesday at 19:00")`. Your solution should not print this explicitly; it should return the three-tuple as the result of the `IO` action.

Suggested solution

```
makeReservation = do
  putStr "Name: "
  name <- getLine
  putStr "Number of guests: "
  n    <- readLn
  putStr "Date and time: "
  at   <- getLine
  return (name, n, at)
```

5

Write a function that drops every n-th element from a list:

```haskell
dropEvery :: Int -> [a] -> [a]
dropEvery n xs = [x | (x, i) <- zip xs [1..], i `mod` n /= 0]

-- Alternative using explicit recursion
dropEvery' n = go n
 where
  go _ [] = []
  go m (x:xs) | m <= 1    = go n xs
              | otherwise = x : go (m-1) xs
```

You may assume that the input integer n is positive (strictly greater than zero). Examples:

```
ghci> dropEvery 2 [1..10]
[1,3,5,7,9]

ghci> dropEvery 3 [1..10]
[1,2,4,5,7,8,10]

ghci> dropEvery 5 [1..10]
[1,2,3,4,6,7,8,9]

ghci> dropEvery 1 [1..10]
[]

ghci> dropEvery 2 []
[]
```

6

In an earlier assignment, we introduced the function `dropEvery`, which removes every *n*-th element from a list. We now want to ensure that this function behaves as expected by testing it with QuickCheck. You can complete this assignment even if you have not solved the earlier one.

Your task is to write a QuickCheck property that verifies that the result of `dropEvery` is always a subset of the input list. In other words, every element that appears in the output list should also appear in the original input list.

```
prop_subset :: Int -> [Int] -> Bool
prop_subset n xs
  | n > 0     = all (`elem` xs) (dropEvery n xs)
  | otherwise = True
```

Note that your property should take into account that `dropEvery` is only defined for positive integers (its first argument).

7

Define a higher-order function that maps two functions over a list of pairs:

```
tmap :: (a -> b) -> (c -> d) -> [(a, c)] -> [(b, d)]
tmap f g = map (\(x, y) -> (f x, g y))
```

The first argument, a function of type `a -> b`, is applied to the first element of a pair (tuple with two elements), whereas the second argument, a function of type `c -> d`, is applied to the second element. This is done for every element in the list.

For example:

```
ghci> tmap (+1) (*2) [(1,1),(2,2),(3,3)]
[(2,2),(3,4),(4,6)]

ghci> tmap even show [(1,1),(2,2),(3,3)]
[(False,"1"),(True,"2"),(False,"3")]
```

Part II

8

Consider the following alternative data type definition for trees:

```
data RoseTree a = Node a [RoseTree a] deriving Show
```

These are also called *rose trees*. Instead of having exactly two subtrees, as in binary trees, a rose tree node contains a list of subtrees. Note that there is no constructor for an empty tree, so every rose tree always contains at least one element.

Here is a smart constructor for creating a leaf node, along with an example rose tree:

```
leaf :: a -> RoseTree a
leaf x = Node x []

rt :: RoseTree Int
rt = Node 1 [Node 2 [leaf 4, leaf 5], Node 3 [leaf 6, leaf 7]]
```

Your tasks are:

a) Write a function that converts a rose tree into a list:

```
flatten :: RoseTree a -> [a]
flatten (Node x forest) = x : concatMap flatten forest
```

You may choose any traversal order for the elements, although a pre-order traversal is the most natural.

For example:

```
ghci> flatten rt
[1,2,4,5,3,6,7]
```

b) Write a function that folds a rose tree:

```
foldTree :: (a -> [b] -> b) -> RoseTree a -> b
foldTree f (Node x forest) = x `f` map (foldTree f) forest
```

This higher-order function takes a function that combines the value stored in a node with the results of folding all its subtrees. Note that the `foldTree` function does not take a base value; this is unnecessary since a rose tree is always non-empty.

For example:

```
ghci> foldTree (\ x xs -> x + sum xs) rt
28
```

9

Consider the following data type that models a small symbolic expression language with integer literals, addition, and multiplication. In addition, the expression language also supports a 'let-expression', which allows for a local definition that can be used in (sub)expression. In a 'let-expression' you can *bind* an expression to a variable and use this variable in another expression. We have defined an auxiliary data type for definitions. For example: `let x = 123 in x + x`, which would evaluate to 246.

```
type Name = String
data Def  = Def Name Expr

data Expr
  = Num Int         -- Literal integer
  | Add Expr Expr   -- Addition
  | Mul Expr Expr   -- Multiplication division
  | Let Def  Expr   -- let x = e1 in e2
  | Var Name        -- Variable
```

Using this data type we can define some example expressions:

```
-- Example expressions and their string representation (in comments)
(x, y, z) = (Var "x", Var "y", Var "z")

e1, e2, e3, e4 :: Expr
e1 = Mul (Add (Num 4) (Num 5)) (Num 2)    -- (4 + 5) * 2
e2 = Let (Def "x" (Num 3)) (Add x x)      -- let x = 3 in x + x
e3 = Let (Def "x" (Num 1)) (Add x y)      -- let x = 1 in x + y
e4 = Let (Def "x" (Num 2))                -- let x = 2 in x * let x = 4 in x * x
         (Mul x (Let (Def "x" (Num 4)) (Mul x x)))
```

Write a function that evaluates an expression:

```
eval :: Expr -> Maybe Int
eval = go []
 where
  go env expr = case expr of
    Num n     -> return n
    Var v     -> lookup v env
    Add x y   -> lift (+) env x y
    Mul x y   -> lift (*) env x y
    Let def e -> do
      env' <- add def env
      go env' e

  lift op env x y = do
    x' <- go env x
    y' <- go env y
    return (x' `op` y')

  add (Def n e) env = let newEnv = [d | d <- env, fst d /= n] in do
    x <- go newEnv e
```

```
    return ((n, x) : newEnv)

-- Alternative using applicative/monadic style
eval' = go []
 where
  go env expr = case expr of
    Num n     -> return n
    Var v     -> lookup v env
    Add x y   -> (+) <$> go env x <*> go env y
    Mul x y   -> (*) <$> go env x <*> go env y
    Let def e -> add def env >>= flip go e

  add (Def n e) env = (\x -> (n, x) : env') <$> go env' e
   where
    env' = [d | d <- env, fst d /= n]
```

The evaluation may fail in case there is an unbound variable (a variable not bound in a `Let`). For example, the evaluation of `e2` results in `Just 6`, whereas the evaluation of `e3` results in `Nothing` because `y` is not bound. Note that a variable can be *shadowed* by another variable, this happens if you have nested 'let-expressions' that bind the *same* variable name. The evaluation should use the innermost binding. For example, in `e4` in the subexpression `(x * x)`, the variable `x` refers to 4 (the innermost binding of `x`) and not 2.

The evaluation of the example expressions:

```
ghci> eval e1
Just 18

ghci> eval e2
Just 6

ghci> eval e3
Nothing

ghci> eval e4
Just 32
```

```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]

maybe :: b -> (a -> b) -> Maybe a -> b
maybe x f m = case m of Just y -> f y; Nothing -> x

-- 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"

reverse          :: [a] -> [a]
reverse          =  foldl (flip (:)) []
```

```haskell
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]

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 ()
```