Exam – Introduction to Functional Programming

TDA555/DIT441 (DIT440), HT-23
Chalmers and Göteborgs Universitet, CSE

Day: 2023-10-25, *Time:* 14:00-18:00, *Place:* Johanneberg

Course responsible

Alex Gerdes (072-9744966). He will visit the exam room once around 15:30, 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 [x]    = x
f (x:xs) = g x (f xs)
 where
  g x y | x >= y    = x
        | otherwise = y
```

a) What does the expression

```
f [4,7,1]
```

evaluate to? Use equational reasoning and write down the intermediate steps of the evaluation.

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

Solution

```
{-
  f [4,7,1]
= { def f }
  g 4 (f [7,1])
= { def f }
  g 4 (g 7 (f [1]))
= { def f }
  g 4 (g 7 1)
= { def g with x <= y }
  g 4 7
= { def g with x <= y }
  7
-}

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

2

Your task is to write a function:

```
removeDups :: [Int] -> [Int]
```

that removes all *duplicate* elements from a list. For example:

```
ghci> removeDups [1,6,2,1,4,5,4,6]
[1,6,2,4,5]
```

Note, the relative order of the remaining elements should be preserved.

Your tasks are:

a) First, write a function that removes all occurrences of a given element from a list:

```
remove :: Int -> [Int] -> [Int]
remove x xs = filter (/= x) xs
```

For example:

```
ghci> remove 1 [1,6,2,1,4,5,4,6]
[6,2,4,5,4,6]
```

Hint: remember the higher-order function `filter`.

b) Next, use the `remove` function to implement the `removeDups` function:

```
removeDups :: [Int] -> [Int]
removeDups []     = []
removeDups (x:xs) = x : removeDups (remove x xs)
```

3

This assignment is about *modeling* football (also called soccer) clubs. A football club consists of a name and a list of players. A player has a name, a field position, and a list of cards for foul play. A field position can be: a defender, a midfielder, or a striker. A card for foul play can be either yellow or red.

Define a collection of data types that models a football club according to the above description.

Suggested solution

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

```
data Player = Player
  { name     :: String
  , fouls    :: [Card]
  , position :: Position
  } deriving Show

data Card = Red | Yellow deriving Show

data Position = Striker | Midfielder | Defender deriving Show

data Club = Club
  { clubName :: String
  , players  :: [Player]
  } deriving Show

-- Example, not part of the assignment
ajax :: Club
ajax = Club "Ajax" [ Player "Marco van  Basten" [] Striker
                   , Player "Zlatan Ibrahimovic" [Yellow, Red] Defender
                   ]
```

4

Imagine you receive an assignment to write a function:

```
teamSelection :: IO [String]
```

that helps a football coach to make a selection of players for a particular game. This `IO` function should first ask the coach how many players she wants to select; let us call this number n. The function should then continue to ask n times the name of a selected player. The function should return a list of selected players . For example:

```
ghci> teamSelection
Welcome coach! How many players do you want to select?
> 3
What is the player's name?
> Zlatan Ibrahimovic
What is the player's name?
> Stina Blackstenius
What is the player's name?
> Marco van Basten
["Zlatan Ibrahimovic","Stina Blackstenius","Marco van Basten"]
```

Hint: write an `IO` help function that asks the name of a single player, and use the function

```
replicateM :: Int -> IO a -> IO [a]
```

to perform the help function n times.

Suggested solution

```
teamSelection = do
  putStr "Welcome coach! How many players do you want to select?\n> "
  n <- readLn
  replicateM n askPlayer
 where
  askPlayer = do
    putStr "What is the player's name?\n> "
    getLine
```

5

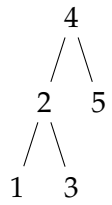
Consider the following data type definition for *non-empty* binary trees:

```
data Tree = Leaf Int | Node Tree Int Tree deriving (Eq, Show)
```

which stores a collection of integers. For example:

```
t :: Tree
t = Node (Node (Leaf 1) 2 (Leaf 3)) 4 (Leaf 5)
```

Which can be depicted as follows:



Your tasks are:

a) Define a function that returns the root (the top element) of a tree:

```
root :: Tree -> Int
root (Leaf x)     = x
root (Node _ x _) = x
```

Calling the function `root` on the example tree `t` will result in:

```
ghci> root t
4
```

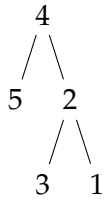
b) Write a recursive function that *mirrors* a tree:

```
mirror :: Tree -> Tree
mirror t = case t of
  Node l x r -> Node (mirror r) x (mirror l)
  _          -> t
```

That is, in every node the left subtree gets swapped with the right subtree. The result of applying `mirror` on the example tree `t` gives:

```
ghci> mirror t
Node (Leaf 5) 4 (Node (Leaf 3) 2 (Leaf 1))
```

Which can be drawn as follows:


4
5
2
3
1

6

In the previous assignment we introduced the `root` and `mirror` functions. We want to make sure that these functions work as aspected by defining some QuickCheck properties. You can do this assignment even if you have not solved the previous assignment.

Your tasks are:

a) Write a property that validates that if we mirror a tree twice we end up with the original tree:

```
prop_mirror :: Tree -> Bool
prop_mirror t = mirror (mirror t) == t
```

b) The root of a tree should remain the same when we mirror the tree. Write a property that checks this:

```
prop_root :: Tree -> Bool
prop_root t = root t == root (mirror t)
```

7

The Prelude function `zip` is a useful function that 'zips' together two lists. It pairwise combines the elements present at the same position in both lists, building a single list of pairs. For example:

```
ghci> zip [1,2,3] "alex"
[(1,'a'),(2,'l'),(3,'e')]
```

Your tasks are:

a) Define a function that 'zips' together *three* lists (instead of two):

```
zip3 :: [a] -> [b] -> [c] -> [(a, b, c)]
zip3 (x:xs) (y:ys) (z:zs) = (x, y, z) : zip3 xs ys zs
zip3 _      _      _      = []
```

b) Write a higher-order function:

```
zipMaybe :: (a -> b -> Maybe c) -> [a] -> [b] -> [c]
zipMaybe f xs ys = catMaybes (zipWith f xs ys)

-- Explicit recursive variant:
zipMaybe' f (x:xs) (y:ys) = let zs = zipMaybe' f xs ys in maybe zs (:zs) (f x y)
zipMaybe' _ _      _      = []

-- A more verbose solution
zm f (x:xs) (y:ys) = let zs = zm f xs ys in case f x y of
  Nothing -> zs
  Just z  -> z : zs
zm _ _ _   = []
```

which 'zips' two lists conditionally: only if the given argument function returns a `Just x` then the value `x` is returned in the resulting list. For example:

```
ghci> zipMaybe (\ x y -> if x < y then Just (x+y) else Nothing) [1,2,3] [0,3,5]
[5,8]
```

Hint: have a look at the `catMaybes` function.

Part II

8

Tic-tac-toe (sv: tre i rad) is a pencil-and-paper game for two players, who take turns marking the cells (with X or O) in a n by n grid (where n is usually 3). We model the game using the following data types:

```
data Board a = Board { size :: Int, rows :: [[a]] }

data Mark = X | O | B deriving Show

example :: Board Mark
example = Board 3 [ [ O, X, B ]
                  , [ X, O, B ]
                  , [ X, B, O ] ]
```

We introduce a general data type for boards, which can have any type of cell. We use the data type `Mark` for marking the cells (positions) on a board (where `B` is used for a blank cell).

Your tasks are:

a) Write a `Show` instance for the `Board` data type:

```
instance Show a => Show (Board a) where
  show (Board n rows) = unlines $ intersperse line (map showRow rows)
   where
    showRow = concat . intersperse "|" . map show
    line    = replicate (2 * n - 1) '-'

-- A more advanced version that takes care of cells of different length
-- (not required to pass)
showBoard (Board _ []) = ""
showBoard (Board n rs) = unlines $ intersperse line (map showRow rs)
   where
    showRow = concat . intersperse "|" . map (pad . show)
    line    = replicate (n * width + n - 1) '-'
    width   = maximum $ concatMap (map (length . show)) rs
    pad     = take width . (++ repeat ' ')
```

A board should be converted to a string as follows: the elements in a row are separated by a vertical bar and the rows are separated by a line of hyphens (sv: bindestreck), with the same length as the string representation for a row. For example:

```
ghci> example
O|X|B
-----
X|O|B
-----
```

```
X|B|O
```

You may assume that every board has n rows of length n. *Hint*: the functions `intersperse` and `unlines` could be useful here.

b) Define a QuickCheck generator for a board of `Marks` of a given size:

```
genBoard :: Int -> Gen (Board Mark)
genBoard n = do
  rows <- vectorOf n (vectorOf n (elements [X, O, B]))
  return (Board n rows)
```

9

In computer science we often need to *parse* data before we can work with it. When we, for example, read data from a file or receive input via the internet, it will very often be a string of characters. We need to convert the string to a value of a particular data type before we can work with it. This process is called parsing.

A parser is a *function* that takes a string as input and produces output of a particular type:

```
type Parser a = String -> Maybe (a, String)
```

We use the `Maybe` data type to indicate that parsing may fail. Furthermore, we may not use (consume) all characters of the input string, and we return the remaining part of the input string as well. We use this to compose parsers: a parser consumes as much of the input as needed and leaves the remaining part of the input string for the next parser.

A common way to construct parsers is to use a so-called parser combinator library. The following collection of functions is such a library (though quite small):

```
-- Apply a function (type a -> b) to the result of another parser (type a)
app :: Parser (a -> b) -> Parser a -> Parser b

-- Make a parser that returns the given value and doesn't consume any input
ret :: a -> Parser a

-- Left-biased or: try the first and only if it fails consider the second
lor :: Parser a -> Parser a -> Parser a

-- Create a parser for a character, if the predicate holds for that character
sat :: (Char -> Bool) -> Parser Char

-- Apply a parser one or more times and return the results in a list
many1 :: Parser a -> Parser [a]

-- Run a parser on a string and return the result
run :: Parser a -> String -> Maybe a
```

With these few functions (also called combinators) we can create very powerful parsers. You don't need the definition for these functions, but they can be found in Appendix A. For example, the following parser recognizes a natural number:

```
pNat :: Parser Int
pNat = ret read `app` many1 (sat isDigit)
```

We try to recognize one or more characters that are digits, which results in a list of digit characters, to which we apply the `read` function, resulting in an `Int`. If we apply this parser on the string "123+321" we get:

```
ghci> pNat "123+321"
Just (123,"+321")
```

As you can see, the parser only consumes the first natural number from the input string, and returns the remainder.

Now, assume that you have a database of personal information stored as a text file in the following format:

```
Paul McCartney,46
John Lennon,45
George Harrison,44
Ringo Starr,54
```

Each line consists of a first and last name, followed by a comma, followed by that person's age. We want to create a parser for this database that converts this to a list of persons:

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

parseDB :: String -> Maybe [Person]
```

Your tasks are:

a) Write a parser that reads a first and last name that are separated by a single space:

```
pName :: Parser (String, String)
pName = retName `app` pString `app` pSpace `app` pString
 where
  pString = many1 (sat isAlpha)
  pSpace  = sat (== ' ')
  retName = ret (\first space last -> (first, last))
```

Hint: think about how to parse a string of letters with help of the `many1` and `sat` combinators. Then figure out how to recognize a space character. Use the `app` combinator to chain these parsers together, and finally use the `ret` combinator to lift a function that only returns the first and last name and ignores the recognized space.

b) Next, define a parser that reads a `Person`:

```
pPerson :: Parser Person
pPerson = retPerson `app` pName `app` pComma `app` pNat
 where
  pComma    = sat (== ',')
  retPerson = ret (\(first, last) _ age -> Person first last age)
```

Use the parser `pName` for reading a name, as well as the `pNat` parser. Take care of recognizing and ignoring the comma character between the name and age. Again, use the `ret` function to lift a function that takes care of this.

We can use the `pPerson` parser to create a parser for the database as follows:

```
parseDB = run $ many1 (ret const `app` pPerson `app` sat (== '\n'))
```

We use the `const` function to ignore the recognized newline. If we apply the `parseDB` function on the example text we get:

```
Just [Person "Paul" "McCartney" 46,Person "John" "Lennon" 45,Person "George"
↪  "Harrison" 44,Person "Ringo" "Starr" 54]
```

```
txt :: String
txt = unlines
  [ "Paul McCartney,46"
  , "John Lennon,45"
  , "George Harrison,44"
  , "Ringo Starr,54"
  ]
```

A

```haskell
app pf p = \inp -> do
  (f, xs) <- pf inp
  (x, ys) <- p xs
  return (f x, ys)

ret x = \inp -> Just (x, inp)

lor p1 p2 = \inp -> maybe (p2 inp) Just (p1 inp)

sat p (c:cs) | p c = Just (c, cs)
sat _ _            = Nothing

many1 p = ret (:) `app` p `app` (many1 p `lor` ret [])

run p = fmap fst . p
```