The University for
business
and the professions

School of Mathematics, Computer Science & Engineering

BSc Computer Science

IN3043: Functional Programming
Part 3 examination Examination

January 2020

Answer THREE questions out of FOUR.
If more than THREE questions are answered, the best THREE will be counted.
Including working may provide the examiner with evidence to award partial credit for solutions that are incorrect.
A summary of selected standard Haskell functions and classes is attached for reference – These may be used in your solutions.

Division of marks: All questions carry equal marks
BEGIN EACH QUESTION ON A FRESH PAGE

Number of answer books to be provided: ONE
Calculators permitted: Casio FX-83/85 MS/ES/GT+ ONLY
Examination duration: 90 minutes
Dictionaries permitted: English translation and language dictionaries are permitted
Additional materials: None
Can question paper be removed from the examination room: No

Question 1

a) Suppose

```
frequencies :: [(String, Int)]
```

is a list of words with the number of times they occur in some document. (You may assume that there is only one entry in the list for each distinct word.) Give expressions for the following:

i) the number of different words in the original document. [5 Marks]

ii) The total number of words in the original document. [10 Marks]

iii) The number of words that occur exactly once each in the original document. [15 Marks]

b) Consider the function

```
foo :: Eq a => a -> [a] -> [Int]
foo x ys = [n | (n, y) <- zip [1..] ys, x == y]
```

i) Explain why the `Eq` constraint on the first line is required. [10 Marks]

ii) Give the value of `foo 'a' "abracadabra"`. [10 Marks]

iii) In general, how is the list returned by `foo` related to its argument? [5 Marks]

c) Consider the following function definition:

```
bar :: Ord a => a -> [a] -> [a]
bar x [] = [x]
bar x (y:ys)
  | x > y     = y : bar x ys
  | otherwise = x : y : ys
```

i) Give the value of the expression `bar 7 [5,3]`. [10 Marks]

ii) Give the value of the expression `bar 4 [1,6,2,7]`. [15 Marks]

iii) In general, how is the list returned by `bar` related to its argument? [5 Marks]

iv) Define an equivalent function, but without using recursion. [15 Marks]

Question 2

a) Write a function

```
letters :: String -> String
```

that returns the original string with all non-letters removed. [10 Marks]

b) Using the function `letters` from the previous part, and the library functions `getLine` and `putStrLn`, write a program fragment to read a line from the console and print a line consisting of the letters in that line. [15 Marks]

c) Consider the function

```
mystery [] = []
mystery (x:xs) = xs ++ [x]
```

i) Give a type signature for `mystery`. [5 Marks]

ii) Give the value of `mystery [4,5,6]`. [10 Marks]

iii) In general, how is the value returned by `mystery` related to its argument? [5 Marks]

d) Consider the definitions

```
f :: Integer -> Integer -> [Integer]
f a b = a : f b (a+b)

ns :: [Integer]
ns = f 1 2
```

i) Give the value of `take 5 ns`. [15 Marks]

ii) Give the value of `takeWhile (<30) (map (7*) ns)`. [15 Marks]

iii) What happens if you evaluate `filter (<7) ns` in the interpreter? [5 Marks]

e) Define a function

```
mapOdds :: (a -> a) -> [a] -> [a]
```

such that

$$\texttt{mapOdds f } [x_1, x_2, x_3, x_4, \ldots] = [\texttt{f } x_1, x_2, \texttt{f } x_3, x_4, \ldots]$$

[20 Marks]

Question 3

a) Give the values of the following expressions:

i) `[w | w <- words "Queen of Hearts", length w > 3]` [10 Marks]

ii) `[10*x + y | x <- [1..3], y <- [x..2*x]]` [20 Marks]

b) Give the values of the following expressions:

i) `map (2*) [1..5]` [5 Marks]

ii) `filter odd (map (+3) [1..6])` [10 Marks]

iii) `takeWhile (< 100) (iterate (*2) 1)` [10 Marks]

c) Suppose an implementation of sets uses ordered lists without duplicates. Write recursive definitions of the following functions, without using any library functions. You may assume that argument lists are in ascending order and contain no duplicates; lists produced by your functions should also be ordered and without duplicates. For full marks, your solutions should avoid repeated traversals of the lists.

i) a function

```
insert :: Ord a => a -> [a] -> [a]
```

that adds an element to the input list, if not already present. [20 Marks]

ii) a function

```
intersection :: Ord a => [a] -> [a] -> [a]
```

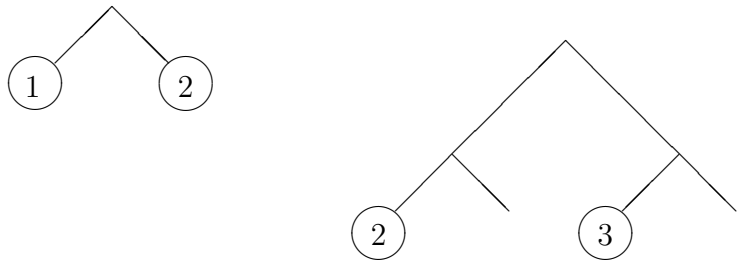
that returns the list of values present in both of the input lists. [25 Marks]

Question 4

Consider the following tree type:

```
data Tree a = Empty
            | Leaf a
            | Branch (Tree a) (Tree a)
```

a) Give values of this type to represent the following trees:



where blank leaves indicate empty subtrees. [15 Marks]

b) What are the types of `Empty` and `Branch`? [15 Marks]

c) What is the type of `Branch (Leaf True)`? [10 Marks]

d) Write a function

```
isLeaf :: Tree a -> Bool
```

that returns `True` if the tree is a leaf. [10 Marks]

e) Consider the following function:

```
foo :: Tree a -> Int
foo Empty = 0
foo (Leaf x) = 1
foo (Branch l r) = foo l + foo r
```

What is the result of applying the function `foo` to each of the two trees in the above diagram? [10 Marks]

f) Write a function

```
elements :: Tree a -> [a]
```

producing a list of the values in the tree, from left to right. The above trees would be mapped to the lists `[1, 2]` and `[2, 3]` respectively. [20 Marks]

g) Write a function

```
flipTree :: Tree a -> Tree a
```

returning the mirror image of the original tree, flipped left-to-right. [20 Marks]

Reference: selected standard functions

Basic functions

- `odd, even :: Integral a => a -> Bool`
  Test whether a number is odd or even
- `null :: [a] -> Bool`
  Test whether a list is empty
- `head :: [a] -> a`
  The first element of a non-empty list
- `tail :: [a] -> [a]`
  All but the first element of a non-empty list
- `last :: [a] -> a`
  The last element of a non-empty list
- `length :: [a] -> Int`
  The length of a list
- `reverse :: [a] -> [a]`
  the reversal of a finite list
- `(++) :: [a] -> [a] -> [a]`
  The concatenation of two lists.
- `zip :: [a] -> [b] -> [(a,b)]`
  List of pairs of corresponding elements of two lists, stopping when one list runs out.
- `take :: Int -> [a] -> [a]`
  The first n elements of the list if it has that many, otherwise the whole list.
- `drop :: Int -> [a] -> [a]`
  The list without the first n elements if it has that many, otherwise the empty list.
- `and :: [Bool] -> Bool`
  `and` returns `True` if all of the Booleans in the input list are `True`.
- `or :: [Bool] -> Bool`
  `or` returns `True` if any of the Booleans in the input list are `True`.
- `product :: Num a => [a] -> a`
  The product of a list of numbers.
- `sum :: Num a => [a] -> a`
  The sum of a list of numbers.
- `concat :: [[a]] -> [a]`
  The concatenation of a list of lists.

Higher order functions

- `map :: (a -> b) -> [a] -> [b]`
  `map f xs` is the list obtained by applying `f` to each element of `xs`:

$$\texttt{map f } [x_1, x_2, \ldots] = [\texttt{f } x_1, \texttt{f } x_2, \ldots\ ]$$

- `filter :: (a -> Bool) -> [a] -> [a]`
  `filter p xs` is the list of elements `x` of `xs` for which `p x` is `True`.

- `iterate :: (a -> a) -> a -> [a]`
  `iterate f x` is the infinite list of repeated applications of `f` to `x`:

$$\texttt{iterate f x} = [\texttt{x}, \texttt{f x}, \texttt{f (f x)}, \ldots\ ]$$

- `takeWhile :: (a -> Bool) -> [a] -> [a]`
  `takeWhile p xs` is the longest prefix of `xs` consisting of elements `x` for which `p x` is `True`.

- `dropWhile :: (a -> Bool) -> [a] -> [a]`
  `dropWhile p xs` is the rest of `xs` after removing `takeWhile p xs`.

Text processing

- `words :: String -> [String]`
  breaks a string up into a list of words, which were delimited by white space.

- `lines :: String -> [String]`
  breaks a string up into a list of strings at newline characters. The resulting strings do not contain newlines.

- `unwords :: [String] -> String`
  joins words, adding separating spaces.

- `unlines :: [String] -> String`
  joins lines, after appending a terminating newline to each.

Character functions

- `isAlpha :: Char -> Bool`
  tests whether a character is alphabetic (i.e. a letter).

- `isUpper :: Char -> Bool`
  tests whether a character is an upper case letter.

- `isLower :: Char -> Bool`
  tests whether a character is a lower case letter.

- `isDigit :: Char -> Bool`
  tests whether a character is a digit.

- `toUpper :: Char -> Char`
  converts lower case letters to upper case, and preserves all other characters.

- `toLower :: Char -> Char`
  converts upper case letters to lower case, and preserves all other characters.

Input/Output

- `getLine :: IO String`
  an action that reads a line from the console.

- `putStrLn :: String -> IO ()`
  `putStrLn s` is an action that writes the string `s`, followed by a newline, to the console.

Selected standard classes

```
class  Eq a  where
    (==), (/=) :: a -> a -> Bool

    x /= y     =  not (x == y)
    x == y     =  not (x /= y)

class  (Eq a) => Ord a  where
    (<), (<=), (>=), (>) :: a -> a -> Bool

class  Show a  where
    show             :: a -> String

class  (Eq a) => Num a  where
    (+), (-), (*)    :: a -> a -> a
    fromInteger      :: Integer -> a
```

Marking Scheme

Question 1

a) i) `length frequencies` — 5 marks

ii) `sum [n | (w, n) <- frequencies]` — 10 marks

iii) `length [n | (w, n) <- frequencies, n == 1]` — 15 marks

b) i) The common type of `x` and the elements of the list `ys` must belong to the `Eq` class, because the function uses `==` on this type. (-2 if no mention of type.) — 10 marks

ii) — 10 marks

```
zip [1..] "abracadabra" =
    [(1,'a'), (2,'b'), (3,'r'), (4,'a'), (5,'c'), (6,'a'),
     (7,'d'), (8,'a'), (9,'b'), (10,'r'), (11,'a')]
```

`foo` picks the numbers paired with `'a'`:

```
foo 'a' "abracadabra" = [1, 4, 6, 8, 11]
```

(only the final answer is required for full marks, but give partial marks for partially correct working.)

iii) The positions (counting from 1) of occurrence of `x` in the list `ys`. — 5 marks

c) i) They may give steps: — 10 marks

```
bar 7 [5,3]  ⇝  5 :  bar 7 [3]
             ⇝  5 :  3 :  bar 7 []
             ⇝  5 :  3 :  [7]
             ⇝  [5, 3, 7]
```

but the final answer is sufficient.

ii) They may give steps: — 15 marks

```
bar 4 [1,6,2,7]  ⇝  1 :  bar 4 [6,2,7]
                 ⇝  1 :  4 :  [6,2,7]
                 ⇝  [1, 4, 6, 2, 7]
```

but the final answer is sufficient.

iii) `bar x ys` returns the list `ys` with `x` inserted before the first element greater than or equal to `x`. — 5 marks

iv) Anything equivalent to: — 15 marks

```
bar x ys = takeWhile (< x) ys ++ [x] ++ dropWhile (< x) ys
```

Marking: splitting [7], insertion [3], concatenation [5].

Question 2

a) Easy answer: 10 marks

```
letters = filter isAlpha
```

Equally acceptable is

```
letters cs = [c | c <- cs, isAlpha c]
```

or even a recursive version.

b) 15 marks

```
do { s <- getLine; putStrLn (letters s) }
```

or

```
do
    s <- getLine
    putStrLn (letters s)
```

or the raw version

```
getLine >>= \ s -> putStrLn (letters s)
```

2 marks for just `putStrLn (letters getLine)`.

c) i) `mystery :: [a] -> [a]` 5 marks

ii) `[5,6,4]` 10 marks

iii) It returns the list with the first element moved to the end. 5 marks

d) i) Sufficient answer: `[1, 2, 3, 5, 8]` 15 marks

Working (not required if answer correct):

ns = f 1 2
= 1 : f 2 3
= 1 : 2 : f 3 5
= 1 : 2 : 3 : f 5 8
= 1 : 2 : 3 : 5 : f 8 13
= 1 : 2 : 3 : 5 : 8 : f 13 21

ii) `[7, 14, 21]` 15 marks

iii) You would get `[1, 2, 3, 5` (3 marks) but then it waits forever looking for more answers (2 marks). 5 marks

e) 20 marks

```
mapOdds f [] = []
mapOdds f [x] = [f x]
mapOdds f (x1:x2:xs) =
    f x1 : x2 : mapOdds f xs
```

or

```
mapOdds f [] = []
mapOdds f (x:xs) = f x : mapEvens f xs

mapEvens f [] = []
mapEvens f (x:xs) = x : mapOdds f xs
```

Question 3

a) i) `["Queen", "Hearts"]` — 10 marks

6 marks for plain strings without showing it's a list.

ii) `[11, 12, 22, 23, 24, 33, 34, 35, 36]` — 20 marks

8 marks for the diagonal `[11, 22, 33]` or the product `[11, 12, 13, 21, 22, 23, 31, 32, 33]`.

b) i) `[2, 4, 6, 8, 10]` — 5 marks

ii) `[5,7,9]` — 10 marks

iii) `[1, 2, 4, 8, 16, 32, 64]` — 10 marks

c) i) — 20 marks

```
insert :: Ord a => a -> [a] -> [a]
insert x [] = [x]
insert x (y:ys)
  | x == y = y : ys
  | x < y = x : y : ys
  | otherwise = y : insert x ys
```

Marking: : 5 marks for each clause.

ii) The preferred answer is — 25 marks

```
intersection [] ys = []
intersection xs [] = []
intersection (x:xs) (y:ys)
  | x == y    = x : intersection xs ys
  | x <  y    = intersection xs (y:ys)
  | otherwise = intersection (x:xs) ys
```

Marking: : base cases [8], equal case [7], non-equal cases [10].
10 marks if they write a union function instead.

Question 4

a) 15 marks

```
Branch (Leaf 1) (Leaf 2)
Branch (Branch (Leaf 2) Empty)
       (Branch (Leaf 3) Empty)
```

Marking: [5] and [10] respectively.

b) 15 marks

```
Empty :: Tree a
Branch :: Tree a -> Tree a -> Tree a
```

Marking: [5] and [10] respectively.

c) 10 marks

```
Tree Bool -> Tree Bool
```

d) 10 marks

```
isLeaf (Leaf _) = True
isLeaf _ = False
```

`Leaf` clause [5], others [5] (three clauses also fine).

e) 2 in both cases. (5 marks each) 10 marks

f) 20 marks

```
elements Empty = []
elements (Leaf x) = [x]
elements (Branch l r) = elements l ++ elements r
```

Marking: [5], [5] and [10] respectively.

g) 20 marks

```
flipTree Empty = Empty
flipTree (Leaf x) = Leaf x
flipTree (Branch l r) = Branch (flipTree r) (flipTree l)
```

Marking: [5], [5] and [10] respectively.