The University for
business
and the professions

School of Mathematics, Computer Science & Engineering

BSc in Computer Science

IN3043: Functional Programming
Part 3 examination Examination

January 2019

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) Give the values of the following expressions:

i) `[w | w <- words "Haskell is fun", odd (length w)]` [10 Marks]

ii) `[10*x + y | x <- [1..3], y <- [1..x]]` [15 Marks]

b) Define a function

```
acronym :: String -> String
```

that forms a word from the initials of the words in a string, for example

```
acronym "Call of Duty" = "CoD"
```

You may assume that the words in the string begin with letters and are separated by spaces. [15 Marks]

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

d) Consider the following function:

```
modify :: [a] -> [a]
modify xs = [x | (n, x) <- zip [1..] xs, even n]
```

i) Give the value of `modify "Haskell"`. [10 Marks]

ii) In general, how is the output of the function `modify` related to its input? [10 Marks]

e) Define a function

```
addLists :: Num a => [a] -> [a] -> [a]
```

that adds corresponding elements of two lists of the same type, with the extra elements of the longer list added at the end, e.g.

```
addLists [1,2,3] [1,3,5,7,9] = [2,5,8,7,9]
```

[15 Marks]

f) Generalize the previous function to a higher-order function that takes the combining function as an argument:

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

For example, `addLists` should be equivalent to `longZip (+)`. [10 Marks]

Question 2

a) Consider the following function definition:

```
mystery :: Eq a => a -> [a] -> [a]
mystery x [] = []
mystery x (y:ys)
  | x == y    = mystery x ys
  | otherwise = y : mystery x ys
```

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

ii) Give the value of the expression `mystery 2 [2,7,1,8,2,8]`. [10 Marks]

iii) What does the function do? [10 Marks]

b) Define a function

```
capitalPositions :: String -> [Int]
```

that returns the positions of the capital letters (counting from 0) in the original list, e.g.

```
capitalPositions "Wind in the Willows" = [0,12]
```

[20 Marks]

c) Give a type signature and definition for a function `interleave` that interleaves two lists of the same type, with the extra elements of the longer list added at the end, e.g.

```
interleave [1,3] [2,4,6,8] = [1,2,3,4,6,8]
interleave "LONGER" "tiny" = "LtOiNnGyER"
```

[25 Marks]

d) Consider the definition

```
f :: Integer -> [Integer]
f n = n : f (2*n + 1)
```

Give the values of

i) `take 5 (f 1)` [10 Marks]

ii) `takeWhile (<20) (map (*2) (f 1))`. [15 Marks]

Question 3

a) Write Haskell expressions for

i) the number of words in a string `s`. [10 Marks]

ii) the list of square numbers that lie between (but not including) the numbers `m` and `n`. (You may assume that $0 < \texttt{m} \leq \texttt{n}$.) [15 Marks]

b) Suppose a Haskell module contains a definition of

```
dictionary :: [String]
```

containing a list of words. Give expressions for

i) the list of words in the dictionary of 10 or more letters. [10 Marks]

ii) the number of words in the dictionary starting with the letter `'a'`. (You may assume the words contain at least one letter.) [10 Marks]

iii) the total length of all the words in the dictionary. [10 Marks]

c) Give list comprehensions (without higher-order functions) that are equivalent to the following expressions:

i) `filter (< 100) (filter (> 0) xs)` [10 Marks]

ii) `map (+2) (filter ((> 99) . (*5)) xs)` [15 Marks]

d) Write a definition of the function

```
unlines :: [String] -> String
```

which joins lines, after appending a newline character (`'\n'`) to each, for example

```
unlines ["Haskell", "is", "fun"] = "Haskell\nis\nfun\n"
```

[20 Marks]

Question 4

Consider the following tree type, which may be used to represent a search tree:

```
data Tree a = Empty
            | Node (Tree a) a (Tree a)
```

a) Give values of this type to represent:

   i) a tree containing only the value `6`. [6 Marks]

   ii) a tree containing only the values `2` and `6`. [9 Marks]

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

c) What is the type of `Node Empty True`? [10 Marks]

d) Write a function

```
isEmpty :: Tree a -> Bool
```

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

e) Write a function

```
singleton :: a -> Tree a
```

that constructs a tree that contains just the single value supplied as an argument. [10 Marks]

f) Write a function

```
count :: Tree a -> Int
```

that returns the number of nodes in the tree. [15 Marks]

g) In a search tree, each subtree `Node l x r` satisfies

   i) all the keys in `l` are smaller than `x`, and

   ii) all the keys in `r` are greater than `x`.

Write a recursive function

```
mkTree :: Ord a => [a] -> Tree a
```

that takes a list (which you may assume does not contain duplicate values) and constructs a search tree with the same elements, such that if the list is non-empty, the key in the top node of the resulting tree is the first element of the list. [25 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) `["Haskell", "fun"]` — 10 marks

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

ii) `[11, 21, 22, 31, 32, 33]` — 15 marks

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

b) — 15 marks

```
acronym s = map head (words s)
```

or equivalent, e.g.

```
acronym s = [c | (c:_) <- words s]
```

Equivalent long-winded recursive versions are equally acceptable.

c) — 15 marks

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

or

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

or the raw version

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

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

d) i) — 10 marks

```
zip [1..] "Haskell" =
    [(1,'H'),(2,'a'),(3,'s'),(4,'k'),(5,'e'),(6,'l'),(7,'l')]
```

so `modify "Haskell" = "akl"`. (5 marks if they include the numbers.)

ii) It returns the list containing every second element of its input list. — 10 marks

e) Allow anything equivalent to — 15 marks

```
addLists [] ys = ys
addLists xs [] = xs
addLists (x:xs) (y:ys) = (x+y):addLists xs ys
```

(four cases also fine). Marking: base cases [5], recursive case [10]. No penalty for continuing to recurse over the longer list, e.g.

```
addLists [] [] = []
addLists [] (y:ys) = y:addLists [] ys
addLists (x:xs) [] = x:addLists xs []
addLists (x:xs) (y:ys) = (x+y):addLists xs ys
```

f) 10 marks

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

Give full marks here to a generalization of the previous answer, even if that answer contained errors.

Question 2

a) i) The type of `x`, which is also the type of elements of the list `ys`, must belong to the `Eq` class, because the function uses `==` on these. (-2 if no mention of type.) — 10 marks

ii) They may give steps: — 10 marks

```
mystery 2 [2,7,1,8,2,8]  ⇝  mystery 2 [2,7,1,8,2,8]
                         ⇝  7 :  mystery 2 [1,8,2,8]
                         ⇝  7 :  1 :  mystery 2 [8,2,8]
                         ⇝  7 :  1 :  8 :  mystery 2 [2,8]
                         ⇝  7 :  1 :  8 :  mystery 2 [8]
                         ⇝  7 :  1 :  8 :  8 :  mystery 2 []
                         ⇝  7 :  1 :  8 :  8 :  []
                         ⇝  [7, 1, 8, 8]
```

but the final answer is sufficient.

iii) `mystery x ys` returns the list `ys` minus any elements equal to `x`. — 10 marks

b) Ideal version: — 20 marks

```
capitalPositions s =
    [n | (c, n) <- zip s [0..], isUpper c]
```

but long-winded recursive versions are also acceptable (if correct), e.g.

```
capitalPositions s = pos 0 s
  where pos :: Int -> String -> [Int]
        pos n [] = []
        pos n (c:cs)
          | isUpper c = n : pos (n+1) cs
          | otherwise = pos (n+1) cs
```

c) The type signature [6] should be — 25 marks

```
interleave :: [a] -> [a] -> [a]
```

For the definition, allow anything equivalent to

```
interleave [] ys = ys
interleave xs [] = xs
interleave (x:xs) (y:ys) = x:y:interleave xs ys
```

(four cases also fine) or

```
interleave [] ys = ys
interleave (x:xs) ys = x:interleave ys xs
```

Marking: base case(s) [7], recursive case [12]. No penalty for continuing to recurse over the longer list.

d) i) `[1, 3, 7, 15, 31]` — 10 marks

Marking Scheme: page 3

ii) They may give working, but the final answer is sufficient for full marks: 15 marks

- `f 1 = [1, 3, 7, 15, 31, ...]`
- `map (*2) (f 1) = [2, 6, 14, 30, 62, ...]`
- `takeWhile (<20) (map (*2) (f 1)) = [2, 6, 14]`

Question 3

a) i) `length (words s)` — 10 marks

ii) Anything equivalent to — 15 marks

```
takeWhile (< n) (dropWhile (<= m) (map (^2) [1..]))
```

including variants of

```
takeWhile (< n) (filter (> m) [x*x | x <- [1..]])
```

But only 12 for equivalents of

```
filter (< n) (filter (> m) [x^2 | x <- [1..]])
```

or

```
[x^2 | x <- [1..], x^2 > m, x^2 < n]
```

because they yield partial lists.

b) i) `[w | w <- dictionary, length w >= 10]` — 10 marks

ii) Either of — 10 marks

```
length [w | w <- dictionary, head w == 'a']
length [1 | ('a':_) <- dictionary]
```

or other equivalent.

iii) `sum [length w | w <- dictionary]` — 10 marks

c) i) `[x | x <- xs, x > 0, x < 100]` or `[x | x <- xs, x > 0 && x < 100]` — 10 marks

ii) Full marks for `[x+2 | x <- xs, x*5 > 99]` — 15 marks
9 marks for errors like `[5*x+2 | x <- xs, x > 99]`

d) Recursive version: — 20 marks

```
unlines [] = ""
unlines (x:xs) = x ++ "\n" ++ unlines xs
```

or non-recursive version:

```
unlines = concat . map (++ "\n")
```

Any equivalent is equally acceptable.

Question 4

a) i) 6 marks

```
Node Empty 6 Empty
```

ii) Any one of the following (don't worry about order at this stage): 9 marks

```
Node Empty 2 (Node Empty 6 Empty)
Node (Node Empty 2 Empty) 6 Empty
Node Empty 6 (Node Empty 2 Empty)
Node (Node Empty 6 Empty) 2 Empty
```

b) 15 marks

```
Empty :: Tree a
Node :: Tree a -> a -> Tree a -> Tree a
```

Marking: [5] and [10] respectively.

c) 10 marks

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

d) 10 marks

```
isEmpty Empty = True
isEmpty _ = False
```

`Empty` clause [5], other one [5] (`Node` clause also fine).

e) 10 marks

```
singleton x = Node Empty x Empty
```

f) 15 marks

```
count Empty = 0
count (Node l k r) = count l + 1 + count r
```

Marking: [5] and [10] respectively.

g) 25 marks

```
mkTree [] = Empty
mkTree (x:xs) = Node (mkTree (filter (< x) xs)) x
                     (mkTree (filter (> x) xs))
```

Marking: base case [6], filters [6], recursive calls [6], making node [7].