CITY
UNIVERSITY OF LONDON
EST 1894

The University for business and the professions

School of Mathematics, Computer Science & Engineering

BSc in Computer Science

IN3043: Functional Programming

January 2017

Answer THREE questions out of four.

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: 120 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 a definition of a function

```
capitalize :: String -> String
```

to capitalize the first letter of each word in a string. You may assume that words are separated by single spaces, and that each word begins with a letter. For example:

```
MyModule> capitalize "good day"
"Good Day"
```

[20 Marks]

b) Using the library functions `getLine` and `putStrLn`, write a program fragment to read two lines from the console and print the second line and then the first. [15 Marks]

c) Consider the functions

```
foo :: Eq a => [a] -> [a]
foo [] = []
foo (x:xs) = x : bar x xs

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

i) Explain the need for the `Eq` constraint in the types of `foo` and `bar`. [10 Marks]

ii) Give the value of `foo [1,2,2,1,1,3]`. [15 Marks]

iii) In general, how is the list returned by `foo` related to its argument? [10 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]

e) For `f` as in the previous part, what happens if you evaluate the expression `length (f 1)`? [5 Marks]

Question 2

a) Give the values of the following expressions:

   i) `[x*x | x <- [4,7,3]]` [5 Marks]

   ii) `[x*5-3 | x <- [1..8], even x]` [10 Marks]

   iii) `[y | x <- [1..3], y <- [x..x*2]]` [15 Marks]

b) Give a list comprehension (without higher-order functions) that is equivalent to the following expression:

```
map (+3) (filter ((> 20) . (*4)) xs)
```

[15 Marks]

c) Consider the following function definition

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

   i) Give the value of the expression `mystery 5 [4,2,7,6]`. [10 Marks]

   ii) Give the value of the expression `mystery 3 [2,7,1,8]`. [10 Marks]

   iii) Define an equivalent function, but without using recursion. [10 Marks]

d) Write a definition of the function

```
intersperse :: a -> [[a]] -> [a]
```

that appends together lists with a separator between them, for example

```
intersperse '-' ["rat", "cat", "dog"] = "rat-cat-dog"
intersperse 0 [[1,2], [3,4,5], [6]] = [1,2,0,3,4,5,0,6]
```

[25 Marks]

Question 3

a) Given a definition of a string of one or more words

```
text :: String
```

with punctuation already removed, write expressions for the following. (If you need to write any extra functions, give those too.)

i) the number of capital letters in `text`. [10 Marks]

ii) the number of words in `text`. [10 Marks]

iii) the number of words of at least 10 letters in `text`. [15 Marks]

iv) the list of words in `text` that occur twice in succession, e.g. "that" in "know that that had happened". The output words should be in the original order, and if the same word is repeated more than once, it should appear for each repetition. [20 Marks]

b) Define a function

```
deleteFirst :: (a -> Bool) -> [a] -> [a]
```

such that `deleteFirst p xs` is obtained from `xs` by deleting the first element `x` for which `p x` is `True`. If there is no such `x`, the result should be `xs`. [20 Marks]

c) Consider the definition

```
sums :: [Int] -> [Int]
sums [] = []
sums (x:xs) = x : map (+x) (sums xs)
```

Give the values of

i) `take 5 (sums [1..])` [10 Marks]

ii) `takeWhile (<20) (map (*3) (sums [1..]))` [15 Marks]

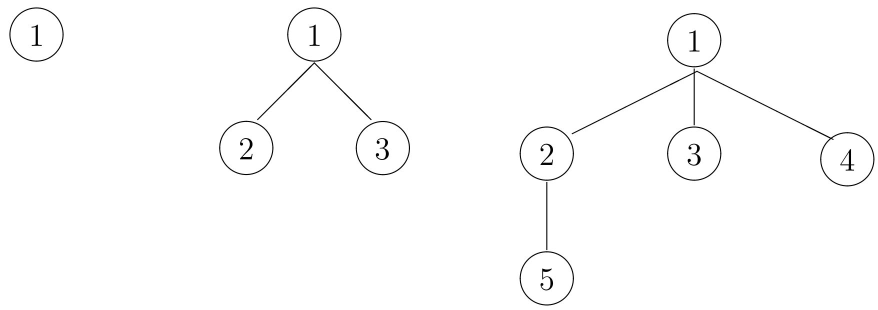
Question 4

Consider the following type definition, used to represent multiway trees:

```
data Tree a = Node a [Tree a]
```

A node consists of a value and a list of subtrees.

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



[20 Marks]

b) What is the type of `Node`? [10 Marks]

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

d) Write a function

```
root :: Tree a -> a
```

that returns the value at the root of the tree. Thus for each of the above trees it would return `1`. [10 Marks]

e) Write a function

```
leaf :: a -> Tree a
```

that constructs a tree of a single node, labelled by the argument. [10 Marks]

f) Write a function

```
sumTree :: Tree Int -> Int
```

that returns the sum of the integers in a tree. For example, the above trees would yield sums of 1, 6 and 15 respectively. [20 Marks]

g) Write a function

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

that returns the mirror image, flipped left-to-right, of the original tree.
[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) Any of the following or any equivalent variant are equally acceptable: **20 marks**

```
capitalize s =
  unwords [toUpper c : cs | (c:cs) <- words s]

capitalize s = unwords [capWord w | w <- words s]
  where capWord (c:cs) = toUpper c : cs

capitalize s = unwords (map capWord (words s))
  where capWord (c:cs) = toUpper c : cs

capitalize = unwords . map capWord . words
  where capWord (c:cs) = toUpper c : cs
```

Recursive versions are also acceptable (though they are unlikely to be correct).

b) **15 marks**

```
do
    s1 <- getLine
    s2 <- getLine
    putStrLn s2
    putStrLn s1
```

or any equivalent.

c) i) `bar` uses `==` on `x` and `y`, so their type must belong to `Eq`.[5] `foo` calls `bar`, and so inherits the constraints on the arguments from `bar`.[5] **10 marks**

ii) `[1,2,1,3]` **15 marks**

iii) It contains the elements of the original list, with adjacent repetitions collapsed to a single element. **10 marks**

d) i) `[1, 3, 7, 15, 31]` **10 marks**

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]`

e) No answer is produced because the infinite list has no end. Alternatively, any of: runs for ever, runs out of stack, runs out of memory. **5 marks**

Question 2

a) i) `[16,49,9]` **5 marks**

ii) `[7, 17, 27, 37]` is correct. 5 marks for `[2, 12, 22, 32]`. **10 marks**

iii) `[1, 2, 2, 3, 4, 3, 4, 5, 6]` **15 marks**

b) Full marks for `[x+3 | x <- xs, x*4 > 20]` **15 marks**

9 marks for `[4*x+3 | x <- xs, x > 20]`

c) i) They may give steps: **10 marks**

```
mystery 5 [4,2,7,6]  ⇝  mystery 5 [2,7,6]
                     ⇝  mystery 5 [7,6]
                     ⇝  [7,6]
```

but the final answer is sufficient.

ii) They may give steps: **10 marks**

```
mystery 3 [2,7,1,8]  ⇝  mystery 3 [7,1,8]
                     ⇝  [7,1,8]
```

but the final answer is sufficient.

iii) **10 marks**

```
mystery x ys = dropWhile (< x) ys
```

d) Recursive version: **25 marks**

```
intersperse x [] = []
intersperse x [y] = y
intersperse x (y:ys) = y ++ (x:intersperse x ys)
```

or non-recursive version:

```
intersperse x [] = []
intersperse x (y:ys) = y ++ concat (map (x:) ys)
```

Any equivalent is equally acceptable.

Question 3

a) i) As a list comprehension **10 marks**

```
length [c | c <- text, isUpper c]
```

or using a higher-order function:

```
length (filter isUpper text)
```

ii) `length (words text)` **10 marks**

iii) **15 marks**

```
length [w | w <- words text, length w >= 10]
```

or equivalently

```
length (filter ((>= 10) . length) (words text))
```

iv) **20 marks**

```
let ws = words text in
    [w | (w, prev) <- zip ws (tail ws), w == prev]
```

Marking: `words` [3], pairing the right words [10], selecting equals [7]. Versions like `dups (words text)` using a recursive function like

```
dups :: [String] -> [String]
dups [] = []
dups [x] = []
dups (x1:x2:xs)
  | x1 == x2 = x1 : dups (x2:xs)
  | otherwise = dups (x2:xs)
```

are equally acceptable.

b) Expected answer: **20 marks**

```
deleteFirst p [] = []
deleteFirst p (x:xs)
    | p x       = xs
    | otherwise = x : deleteFirst p xs
```

Marking: base case [5], terminating case [7], recursive case [8].
A non-recursive version like

```
deleteFirst p xs =
    takeWhile (not . p) ++ drop 1 (dropWhile (not . p)
```

is equally acceptable.

c) i) `[1, 3, 6, 10, 15]` **10 marks**

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

- `sums [1..]  = [1, 3, 6, 10, 15, 21, ...]`
- `map (*3) (sums [1..])  = [3, 9, 18, 30, 45, 63, ...]`
- `takeWhile (<20) (map (*3) (sums [1..]))  = [3, 9, 18]`

Question 4

a) 20 marks

```
Node 1 []
Node 1 [Node 2 [], Node 3 []]
Node 1 [Node 2 [Node 5 []], Node 3 [], Node 4 []]
```

Marking: [4], [7] and [9] respectively.

b) 10 marks

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

c) 10 marks

```
[Tree Bool] -> Tree Bool
```

d) 10 marks

```
root (Node x _) = x
```

e) 10 marks

```
leaf x = Node x []
```

f) 20 marks

```
sumTree (Node x ts) =
  x + sum (map sumTree ts)
```

A list comprehension version is equally acceptable:

```
sumTree (Node x ts) =
  x + sum [sumTree t | t <- ts]
```

g) 20 marks

```
flipTree (Node x ts) =
  Node x (reverse (map flipTree ts))
```

Again a list comprehension version would be fine (though less likely).