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
Part 3 examination Examination

XXXday XXth XXX XXXX XXXX – XXXX

Answer THREE questions out of FOUR.
Including working may provide the examiner with evidence to award partial credit for solutions that are incorrect.
All questions carry equal marks
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) Consider a function

```
count :: Eq a => a -> [a] -> Int
```

that returns the number of times its first argument occurs in its second. For example:

- `count 'a' "abracadabra"` returns `5`.
- `count 15 [2,7,1,8,2,8]` returns `0`.

Write three separate implementations of `count`, using different techniques:

i) using a list comprehension [15 Marks]

ii) using a higher-order library function [15 Marks]

iii) as a recursive function [20 Marks]

b) In a certain programming language, an identifier is a string consisting of a letter followed by one or more letters, digits or underscore characters (`'_'`). Write a function

```
isIdentifier :: String -> Bool
```

that tests whether a string is such an identifier. [25 Marks]

c) Consider the function

```
while :: (a -> Bool) -> (a -> a) -> a -> a
while p f x = head (dropWhile p (iterate f x))
```

i) Give the value of `while (< 10) (*2) 1`. [5 Marks]

ii) The above implementation uses lists internally. Write an alternative definition of this function that uses recursion instead of lists. [20 Marks]

Question 2

a) Explain the difference between the types `String` and `IO String`, and give an example to show how the string in the second type may be accessed.

[15 Marks]

b) Describe the relationship between inputs and outputs for the following functions:

i)
```
incr :: [Int] -> [Int]
incr [] = []
incr [x] = [x]
incr (x1:x2:xs) = x1 : (x2+1) : incr xs
```

[15 Marks]

ii)
```
pick :: (a -> Bool) -> [a] -> [a]
pick p [] = []
pick p (x:xs)
  | p x       = pick p xs
  | otherwise = x : pick p xs
```

[20 Marks]

iii)
```
mklist :: Integer -> [Integer]
mklist x = 1 : map (x*) (mklist x)
```

[15 Marks]

c) Write a higher-order function

```
total :: (Int -> Int) -> [Int] -> Int
```

such that

$$\texttt{total}\ f\ [x_1, x_2, \ldots, x_n] = f\ x_1 + f\ x_2 + \cdots + f\ x_n$$

[15 Marks]

d) Write a higher-order function

```
compose :: [a -> a] -> a -> a
```

such that

$$\texttt{compose}\ [f_1, f_2, \ldots, f_n]\ x = f_1\ (f_2\ (\ldots (f_n\ x) \ldots))$$

[20 Marks]

Question 3

a) Give the values of the following expressions:

i) `zip "abc" [1..]`

[10 Marks]

ii) `[n*(n+1) | n <- [1..4]]`

[10 Marks]

iii) `[toUpper c | c <- "I like Haskell", isLower c]`

[15 Marks]

iv) `[[1..n] | n <- [1..3]]`

[15 Marks]

b) Write a function

```
pairs :: Int -> [(Int, Int)]
```

such that `pairs` n returns the list of pairs of numbers (x, y) such that $1 \leq x < y \leq n$. [15 Marks]

c) Write a function

```
replace :: Int -> Int -> [Int] -> [Int]
```

such that `replace` x y zs, replaces each occurrence of x in zs with y. For example,

```
replace 8 2 [2,7,1,8,2,8] = [2,7,1,2,2,2]
```

Marking: base case [3], test [3], right-hand sides [3 each]. [15 Marks]

d) Consider the following function:

```
foo :: Integer -> [Integer]
foo n = takeWhile ((< n) . (^2)) [1..]
```

i) Describe the relationship between inputs and outputs for this function. [15 Marks]

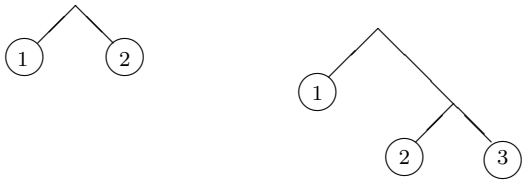
ii) How would the behaviour of the function change if we replaced the function `takeWhile` in the definition with `filter`? [5 Marks]

Question 4

Consider the following tree type:

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

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



[15 Marks]

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

c) What is the type of `Branch (Leaf 'a')`? [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:

```
choose :: Tree Int -> Int
choose (Leaf x) = x
choose (Branch l r) = choose r
```

What is the result of applying the function `choose` to each of the two trees in the above diagram? [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 3 and 6 respectively. [20 Marks]

g) Generalize the previous function to a higher order function that takes a function to use in place of the addition function:

```
foldTree :: (a -> a -> a) -> Tree a -> a
```

so that `sumTree = foldTree (+)`. [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) 15 marks

```
count x ys = length [y | y <- ys | x == y]
```

Anything else in place of the first `y` is also fine, as are equivalent solutions like

```
count x ys = sum [1 | y <- ys | x == y]
```

ii) 15 marks

```
count x ys = length (filter (== x) ys)
```

or even

```
count x = length . filter (== x)
```

iii) 20 marks

```
count x [] = 0
count x (y:ys)
  | x == y = count x ys + 1
  | otherwise = count x ys
```

Marking: : base case[5], structure of cons case[5], right-hand sides [5 each].

b) Several answers are possible: 25 marks

```
isIdentifier [] = False
isIdentifier (c:cs) = isAlpha c && and (map isIdChar cs)
  where isIdChar c = isAlpha c || isDigit c || c == '_'
```

but equivalent versions using list comprehensions or recursion are equally acceptable. Marking: empty case[5], first character[5], individual later characters[8], combination of tests[7].

c) i) An answer of 16 is sufficient for full marks. Working: 5 marks

```
    while (< 10) (*2) 1
=   head (dropWhile (< 10) (iterate (*2) 1))
=   head (dropWhile (< 10) [1, 2, 4, 8, 16, 32, ...])
=   head [16, 32, ...]
=   16
```

ii) 20 marks

```
while p f x
  | p x = while p f (f x)
  | otherwise = x
```

Question 2

a) A value of type `String` is a string, while one of type `IO String` is an I/O action to produce a string [10]. The string may be accessed using `do`-notation: 15 marks

```
do { c <- getLine; putStrLn (map toUpper c) }
```

or

```
do
    c <- getLine
    putStrLn (map toUpper c)
```

or the raw version

```
getLine >>= \ c -> putStrLn (map toUpper c)
```

or any similar example using other functions instead of `getLine` and `putStrLn` [5]. The argument of `putStrLn` is immaterial.

b) i) Adds one to every second element of the input list. 15 marks
Marking: maps a list to a list [3] of the same length [2], increment [4], which elements [6] (2 for all but first). (These may be implicit.)

ii) Same as `filter (not.p)`: select the elements of the input list that don't satisfy `p`. 20 marks
Marking: two arguments: a function and a list [8], returns a list [4], appropriate selection [8]. (These may be implicit.)

iii) The infinite list [5] of powers of `x` [10]. 15 marks

c) 15 marks

```
total f xs = sum (fmap f xs)
```

or just

```
total f = sum . fmap f
```

or list-comprehension form

```
total f xs = sum [f x | x <- xs]
```

or recorsive form

```
total f [] = 0
total f (x:xs) = f x + total f xs
```

Marking: sum [5], mapping [10].

d) Most likely answer is 20 marks

```
compose [] x = x
compose (f:fs) x = f (compose fs x)
```

Less likely, but also correct would be

```
compose = foldr (.) id
```

or even

```
compose fs x = foldr id x fs
```

or

```
compose = flip (foldr id)
```

Also equally acceptable is the more convoluted

```
compose [] x = x
compose fs x = compose (inits fs) (last fs x)
```

Marking: empty list [5] (3 if `[f]` is base case), non-empty list [15].

Question 3

a) i) `[('a',1),('b',2),('c',3)]` — 10 marks
Marking: 2 if all combinations.

ii) `[2,6,12,20]` — 10 marks

iii) `"LIKEASKELL"` — 15 marks
Marking: selection [7], conversion [8].

iv) `[[1],[1,2],[1,2,3]]` — 15 marks
Marking: 13 if all one list.

b) The best answer would be — 15 marks

```
pairs n = [(x, y) | x <- [1..n-1], y <- [x+1..n]]
```

but anything equivalent is equally acceptable, e.g.

```
pairs n = [(x, y) | x <- [1..n], y <- [1..n], x < y]
```

c) Any correct solution is equally acceptable, including — 15 marks

```
replace x y zs = [if z == x then y else z | z <- zs]

replace x y = map repl
  where
    repl z
      | z == x = y
      | otherwise = z

replace x y [] = []
replace x y (z:zs)
  | z == x = y : replace x y zs
  | otherwise = z : replace x y zs
```

d) i) `foo` n returns the list of positive numbers whose squares are less than n. — 15 marks
Marking: 7 for list of squares less than n.

ii) The function would return the same numbers [1], but would not terminate returning the end of the list, as `filter` would continue to examine the infinite input list [4]. (Do not penalize for values repeated from an incorrect answer to the previous part.) — 5 marks

Question 4

a) 15 marks

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

Marking: [6] and [9] respectively.

b) 15 marks

```
Leaf :: a -> Tree a
Branch :: Tree a -> Tree a -> Tree a
```

Marking: [6] and [9] respectively.

c) 10 marks

```
Tree Char -> Tree Char
```

d) 10 marks

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

`Leaf` clause [5], other one [5] (`Branch` clause also fine).

e) 2 and 3. 10 marks

f) 20 marks

```
sumTree (Leaf x) = x
sumTree (Branch l r) = sumTree l + sumTree r
```

Marking: [6] and [14] respectively.

g) 20 marks

```
foldTree f (Leaf x) = x
foldTree f (Branch l r) = f (foldTree f l) (foldTree f r)
```

Marking: [6] and [14] respectively.