UNIVERSITY OF EDINBURGH

COLLEGE OF SCIENCE AND ENGINEERING

SCHOOL OF INFORMATICS

INFR08025 INFORMATICS 1 - INTRODUCTION TO COMPUTATION

Thursday 13$^{\underline{\text{th}}}$ December 2018

14:30 to 16:30

INSTRUCTIONS TO CANDIDATES

1. **Note that ALL QUESTIONS ARE COMPULSORY.**
2. **DIFFERENT QUESTIONS MAY HAVE DIFFERENT NUMBERS OF TOTAL MARKS. Take note of this in allocating time to questions.**
3. **This is an OPEN BOOK examination: notes and printed material are allowed, and USB sticks (read only), but no electronic devices.**
4. **CALCULATORS MAY NOT BE USED IN THIS EXAMINATION**

Convener: D.K.Arvind
External Examiner: J.Gibbons

THIS EXAMINATION WILL BE MARKED ANONYMOUSLY

1. (a) Write a function `f :: String -> Int` that computes the sum of the positions of the digits in a string, where positions begin with 0. For example:

```
f "" = 0
f "0 is the first position" = 0
f "I Love Functional Programming" = 0
f "2nite is 2 L8" = 21
f "0131 650 1000" = 66
f "1oTs & LoT5 of Num63r5" = 68
```

Use *basic functions*, *list comprehension*, and *library functions*, but *not recursion*. Credit may be given for indicating how you have tested your function. [*16 marks*]

(b) Write a second function `g :: String -> Int` that behaves like `f`, this time using *basic functions* and *recursion*, but *not list comprehension* or *library functions*. Credit may be given for indicating how you have tested your function. [*16 marks*]

2. (a) Write a function `p :: [(Int,Int)] -> Bool` that tests whether or not the sum of the products of the numbers in the pairs in a list is greater than zero, excluding pairs where the second component is negative. For example:

```
p [] = False
p [(-1,-2),(-3,-5)] = False
p [(4,5),(-7,3)] = False
p [(4,5),(-6,3),(2,-2)] = True
p [(4,5),(-6,3),(-2,2)] = False
p [(4,-5),(-3,2),(1,6),(-3,-1)] = False
```

Use *basic functions*, *list comprehension*, and *library functions*, but *not recursion.* Credit may be given for indicating how you have tested your function.

[*12 marks*]

(b) Write a function `q :: [(Int,Int)] -> Bool` that behaves like `p`, this time using *basic functions* and *recursion*, but *not list comprehension* or *library functions.* Credit may be given for indicating how you have tested your function. [*12 marks*]

(c) Write a function `r :: [(Int,Int)] -> Bool` that also behaves like `p`, this time using one or more of the following higher-order library functions:

```
    map     :: (a -> b) -> [a] -> [b]
    filter  :: (a -> Bool) -> [a] -> [a]
    foldr   :: (a -> b -> b) -> b -> [a] -> b
```

You may use basic functions but do *not* use *recursion*, *list comprehension* or *library functions* other than these three. Credit may be given for indicating how you have tested your function.

[**Hint:** When using `map` and `foldr`, don't forget that both `+` and `*` have type `Int -> Int -> Int`.] [*12 marks*]

3. The following data type represents binary trees with leaves of type `a`.

```
data Tree a = Lf a                    -- leaf
            | Tree a :+: Tree a       -- branch
```

The template file provides instances

```
(==) :: (Eq a) => Tree a -> Tree a -> Bool
show :: (Show a) => Tree a -> String
```

to compare two trees for equality (if their leaves can be compared for equality), and to convert a tree into a readable format (if its leaves can be converted into a readable format), It also provides code that enables QuickCheck to generate arbitrary values of type `Tree`, to aid testing.

(a) We call a tree *left-leaning* if the right branch of every subtree is a leaf. Write a function `left :: Tree a -> Bool` that determines whether a given tree is left leaning. For example,

```
left (Lf 1)                                 =  True
left (((Lf 1 :+: Lf 2) :+: Lf 3) :+: Lf 4)  =  True
left ((Lf 1 :+: Lf 2) :+: (Lf 3 :+: Lf 4))  =  False
left (Lf "a" :+: (Lf "b" :+: Lf "c"))       =  False
left ((Lf "a" :+: Lf "b") :+: Lf "c")       =  True
```

Credit may be given for indicating how you have tested your function. [*8 marks*]

(b) Write a function `leaves :: Tree a -> [a]` that returns a list of all the leaves in a tree, ordered from left to right. For example,

```
leaves (Lf 1)                                 =  [1]
leaves (Lf 1 :+: (Lf 2 :+: (Lf 3 :+: Lf 4)))  =  [1,2,3,4]
leaves ((Lf 1 :+: Lf 2) :+: (Lf 3 :+: Lf 4))  =  [1,2,3,4]
leaves (Lf "a" :+: (Lf "b" :+: Lf "c"))       =  ["a","b","c"]
leaves ((Lf "a" :+: Lf "b") :+: Lf "c")       =  ["a","b","c"]
```

Credit may be given for indicating how you have tested your function. [*8 marks*]

QUESTION CONTINUES ON NEXT PAGE

(c) Write a function `shift :: Tree a -> Tree` that converts a tree to a left-leaning tree, containing the same leaves in the same order. For example,

```
shift (Lf 1)
  =  (Lf 1)
shift (((Lf 1 :+: Lf 2) :+: Lf 3) :+: Lf 4)
  =  (((Lf 1 :+: Lf 2) :+: Lf 3) :+: Lf 4)
shift ((Lf 1 :+: Lf 2) :+: (Lf 3 :+: Lf 4))
  =  (((Lf 1 :+: Lf 2) :+: Lf 3) :+: Lf 4)
shift (Lf "a" :+: (Lf "b" :+: Lf "c"))
  =  ((Lf "a" :+: Lf "b") :+: Lf "c")
shift ((Lf "a" :+: Lf "b") :+: Lf "c")
  =  ((Lf "a" :+: Lf "b") :+: Lf "c")
```

Credit may be given for indicating how you have tested your function. [*16 marks*]