UNIVERSITY OF EDINBURGH

COLLEGE OF SCIENCE AND ENGINEERING

SCHOOL OF INFORMATICS

INFR08013 INFORMATICS 1 - FUNCTIONAL PROGRAMMING

Tuesday 14th August 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: I. Simpson
External Examiner: I. Gent

THIS EXAMINATION WILL BE MARKED ANONYMOUSLY

1. (a) Write a function `f :: [String] -> [String]` that replaces the first letter of each word in a list by the last letter of the next word. For example:

```
f ["pattern","matching","rules","ok"]
            = ["gattern","satching","kules"]
f ["word"]  = []
f ["almost","all","students","love","functional","programming"]
            = ["llmost","sll","etudents","love","gunctional"]
f ["make","love","not","war"]
            = ["eake","tove","rot"]
```

You should assume that the input list is not empty and that it contains no empty words. The output list will contain one word fewer than the input list. 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] -> [String]` that behaves like `f`, this time using *basic functions*, *recursion*, and *library functions*, but *not list comprehension*. Credit may be given for indicating how you have tested your function. [*16 marks*]

2. An *three-letter acronym* (TLA), is an abbreviation consisting three capital letters, for example HRH and LOL.

   (a) Write a function `p :: [String] -> Int` that counts the number of TLAs in a list of strings. For example:

   ```
   p ["I","played","the","BBC","DVD","in","the","USA"] = 3
   p ["The","DUP","MP","travelled","to","LHR"] = 2
   p ["The","SNP","won","in","South","Morningside"] = 1
   p [] = 0
   ```

   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 second function `q :: [String] -> Int` 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 third function `r :: [String] -> Int` 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 and your function `isInitialism` 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. [*12 marks*]

3. The following data type represents arithmetic expressions over two variables, X and Y:

```
data Expr = X                       -- variable X
          | Y                       -- variable Y
          | Const Int               -- integer constant
          | Expr :+: Expr           -- addition
          | Expr :*: Expr           -- multiplication
```

The template file includes a function `showExpr :: Expr -> String` which converts expressions into a readable format, and code that enables QuickCheck to generate arbitary values of type `Expr`, to aid testing.

(a) Write a function `eval :: Expr -> Int -> Int -> Int`, which given an expression and the values of the variable `X` and `Y`, in that order, returns the value of the expression. For example,

```
eval ((X :*: Const 3) :+: (Const 0 :*: Y)) 2 4 = 6
eval (X :*: (Const 3 :+: Y)) 2 4               = 14
eval (Y :+: (Const 1 :*: X)) 3 2               = 5
eval (((Const 1 :*: Const 1) :*: (X :+: Const 1)) :*: Y) 3 4
                                               = 16
```

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

(b) We call an expression *simple* if it contains no applications of multiplication where either argument is 0 or 1.

Write a function `isSimple :: Expr -> Bool` that determines whether or not an expression is simple. For example,

```
isSimple ((X :*: Const 3) :+: (Const 0 :*: Y)) = False
isSimple (X :*: (Const 3 :+: Y))               = True
isSimple (Y :+: (Const 1 :*: X))               = False
isSimple (((Const 1 :*: Const 1) :*: (X :+: Const 1)) :*: Y)
                                               = False
```

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

QUESTION CONTINUES ON NEXT PAGE

(c) Write a function `simplify :: Expr -> Expr` that converts an expression to an equivalent simple expression by use of the following laws:

$$\begin{aligned} \texttt{Const 0 :*: } e &= \texttt{Const 0} \\ \texttt{Const 1 :*: } e &= e \\ e \texttt{ :*: Const 0} &= \texttt{Const 0} \\ e \texttt{ :*: Const 1} &= e \end{aligned}$$

For example,

```
simplify ((X :*: Const 3) :+: (Const 0 :*: Y)) =
                 (X :*: Const 3) :+: Const 0
simplify (X :*: (Const 3 :+: Y)) =
                 X :*: (Const 3 :+: Y)
simplify (Y :+: (Const 1 :*: X)) =
                 Y :+: X
simplify (((Const 1 :*: Const 1) :*: (X :+: Const 1)) :*: Y) =
                 (X :+: Const 1) :*: Y
```

(Note that further simplifications are possible, using other laws, but `simplify` should do only those indicated above.) Credit may be given for indicating how you have tested your function. [*16 marks*]