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 \( f :: \text{[String]} \to \text{[String]} \) that replaces the first letter of each word in a list by the last letter of the next word. For example:

\[
\begin{align*}
  f \ ["pattern","matching","rules","ok"] &= \ ["gattern","satching","kules"] \\
  f \ ["word"] &= [] \\
  f \ ["almost","all","students","love","functional","programming"] &= \ ["llmost","sll","etudents","love","gunctional"] \\
  f \ ["make","love","not","war"] &= \ ["eake","tove","rot"]
\end{align*}
\]

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 \( g :: \text{[String]} \to \text{[String]} \) 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 of three capital letters, for example HRH and LOL.

(a) Write a function \( p :: \text{[String]} \to \text{Int} \) that counts the number of TLAs in a list of strings. For example:

\[
\begin{align*}
p ["I","played","the","BBC","DVD","in","the","USA"] &= 3 \\
p ["The","DUP","MP","travelled","to","LHR"] &= 2 \\
p ["The","SNP","won","in","South","Morningside"] &= 1 \\
p [] &= 0
\end{align*}
\]

Use *basic functions*, *list comprehension*, and *library functions*, but *not recursion*. Credit may be given for indicating how you have tested your function. \([12 \text{ marks}]\)

(b) Write a second function \( q :: \text{[String]} \to \text{Int} \) 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. \([12 \text{ marks}]\)

(c) Write a third function \( r :: \text{[String]} \to \text{Int} \) that also behaves like \( p \), this time using one or more of the following higher-order library functions:

\[
\begin{align*}
\text{map} &:: (a \to b) \to \text{[a]} \to \text{[b]} \\
\text{filter} &:: (a \to \text{Bool}) \to \text{[a]} \to \text{[a]} \\
\text{foldr} &:: (a \to b \to b) \to b \to \text{[a]} \to b
\end{align*}
\]

You may use *basic functions* and your function \( \text{isInitialism} \) 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. \([12 \text{ marks}]\)
3. The following data type represents arithmetic expressions over two variables, \( \text{X} \) and \( \text{Y} \):

\[
\text{data Expr} = \text{X} \quad \text{-- variable X} \\
| \text{Y} \quad \text{-- variable Y} \\
| \text{Const Int} \quad \text{-- integer constant} \\
| \text{Expr :+: Expr} \quad \text{-- addition} \\
| \text{Expr :*: Expr} \quad \text{-- multiplication}
\]

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

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

\[
\begin{align*}
\text{eval ((X :*: Const 3) :+: (Const 0 :*: Y)) 2 4} &= 6 \\
\text{eval (X :*: (Const 3 :+: Y)) 2 4} &= 14 \\
\text{eval (Y :+: (Const 1 :*: X)) 3 2} &= 5 \\
\text{eval (((Const 1 :*: Const 1) :*: (X :+: Const 1)) :*: Y) 3 4} &= 16
\end{align*}
\]

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

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

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

\[
\begin{align*}
\text{isSimple ((X :*: Const 3) :+: (Const 0 :*: Y))} &= \text{False} \\
\text{isSimple (X :*: (Const 3 :+: Y))} &= \text{True} \\
\text{isSimple (Y :+: (Const 1 :*: X))} &= \text{False} \\
\text{isSimple (((Const 1 :*: Const 1) :*: (X :+: Const 1)) :*: Y)} &= \text{False}
\end{align*}
\]

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

\textit{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:

- `Const 0 :*: e = Const 0`
- `Const 1 :*: e = e`
- `e :*: Const 0 = Const 0`
- `e :*: Const 1 = e`

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]