UNIVERSITY OF EDINBURGH

COLLEGE OF SCIENCE AND ENGINEERING

SCHOOL OF INFORMATICS

INFR08013 INFORMATICS 1 - FUNCTIONAL PROGRAMMING

Thursday 22 nd August 2013

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.**

Convener: J Bradfield
External Examiner: A Preece

THIS EXAMINATION WILL BE MARKED ANONYMOUSLY

1. (a) Write a function `f :: [(Int,Int)] -> [Int]` that takes a list of pairs of integers and returns a list containing the first element from each of the pairs in even-numbered positions and the second element from each of the pairs in odd-numbered positions, where numbering of list elements begins from 0. For example:

```
f [(1,2),(5,7),(3,8),(4,9)]   =   [1,7,3,9]
f [(1,2)]                     =   [1]
f []                          =   []
```

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

2. (a) Write a function `p :: [Int] -> Int` that computes the product of the results of multiplying each of the positive odd numbers in a list by three. If the list is empty or there are no positive odd numbers in the list, the function should produce `1`. For example:

```
p [1,6,-15,11,-9]   =  3*1 * 3*11        =  99
p [3,6,9,12,-9,9]   =  3*3 * 3*9 * 3*9   =  6561
p []                =  1
p [-1,4,-15]        =  1
```

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 `q :: [Int] -> 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. [*16 marks*]

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

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

Do not use recursion or list comprehension. Credit may be given for indicating how you have tested your function. [*12 marks*]

3. The following data type represents propositional formulas built from a single variable (`X`), constants true (`T`) and false (`F`), negation (`Not`) and bi-implication, also known as logical equivalence ($\leftrightarrow$, written using infix `:<->:`):

```
data Prop = X
          | F
          | T
          | Not Prop
          | Prop :<->: Prop
```

The template file includes a function (`showProp :: Prop -> String`) which converts formulas into a readable format and code that enables QuickCheck to generate arbitrary values of type `Prop`, to aid testing.

(a) Write a function `eval :: Prop -> Bool -> Bool`, which given a propositional formula and the value of the variable `X` returns the value of the formula. Recall the truth tables for negation and bi-implication:

P	`Not` P
F	T
T	F

P	Q	P `:<->:` Q
F	F	T
F	T	F
T	F	F
T	T	T

For example,

```
eval (Not T) True                        =  False
eval (Not X) False                       =  True
eval (Not X :<->: Not (Not X)) True      =  False
eval (Not X :<->: Not (Not X)) False     =  False
eval (Not (Not X :<->: F)) True          =  False
eval (Not (Not X :<->: F)) False         =  True
```

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

(b) Write a function `simplify :: Prop -> Prop` that converts a propositional formula to an equivalent simpler formula by repeated use of the following laws:

$$
\begin{aligned}
\texttt{Not T} &= \texttt{F} \\
\texttt{Not F} &= \texttt{T} \\
\texttt{Not (Not } p\texttt{)} &= p \\
\texttt{T :<->: } p &= p \\
\texttt{F :<->: } p &= \texttt{Not } p \\
p\texttt{ :<->: T} &= p \\
p\texttt{ :<->: F} &= \texttt{Not } p \\
p\texttt{ :<->: } p &= \texttt{T}
\end{aligned}
$$

QUESTION CONTINUES ON NEXT PAGE

For example,

```
simplify (Not F)                          =  T
simplify (Not X :<->: Not (X :<->: T))    =  T
simplify (Not (Not X :<->: Not T))        =  Not X
simplify (Not (F :<->: Not (Not X)))      =  X
```

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