UNIVERSITY OF EDINBURGH

COLLEGE OF SCIENCE AND ENGINEERING

SCHOOL OF INFORMATICS

INFR08013 INFORMATICS 1 - FUNCTIONAL PROGRAMMING

Wednesday 19th December 2012

09:30 to 11:30

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 replaces every second element of a list, starting with the first, with a given item.

   The result list should have the same length as the argument list and should begin with the item, followed by the second element of the list, followed by the item, followed by the fourth element of the list, followed again by the item, and so on. For example:

```
f 0 [1,2,3,4,5]  =  [0,2,0,4,0]
f 0 [1,2,3,4]    =  [0,2,0,4]
f 0 []           =  []
f 0 [7]          =  [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 `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] -> Bool` that checks that every number in a list that is between 10 and 100 (inclusive) is even. For example:

```
p [1,12,153,84,64,9]  =  True
p [1,12,153,83,9]     =  False
p []                  =  True
p [1,151]             =  True
```

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] -> 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. [*16 marks*]

(c) Write a third function `r :: [Int] -> Bool` 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 disjunction ($\vee$, 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 truth value of the variable `X` returns the value of the formula. For example,

```
eval (Not T) True                  =  False
eval (Not X) False                 =  True
eval (Not X :|: Not (Not X)) True  =  True
eval (Not X :|: Not (Not X)) False =  True
eval (Not (Not X :|: F)) True      =  True
eval (Not (Not X :|: F)) False     =  False
```

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 use of the following laws:

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

For example,

```
simplify (Not X :|: Not (Not X))    =  Not X :|: X
simplify (Not (Not X :|: F))        =  X
simplify (Not T)                    =  F
simplify (Not F :|: X)              =  T
simplify (Not (Not (Not X) :|: X))  =  Not X
```

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