UNIVERSITY OF EDINBURGH

COLLEGE OF SCIENCE AND ENGINEERING

SCHOOL OF INFORMATICS

INFORMATICS 1 - FUNCTIONAL PROGRAMMING

Monday 12$^{\underline{\text{th}}}$ December 2011

14:30 to 16:30

Convener: J Bradfield
External Examiner: A Preece

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.**

1. (a) Write a function `f :: [Int] -> Int` to find the maximum of the positive numbers in a list. If no number in the list is positive, it should return zero. For example,

```
f [1,2,3,4,5]     ==  5
f [-1,2,-3,4,-5]  ==  4
f [-1,-2,-3]      ==  0
f [2,42,-7]       ==  42
```

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` 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 sum of the products of adjacent elements in a list of even length, as shown below. The function should give an error if provided a list of odd length. For example:

```
p [1,2,3,4]         =  1*2 + 3*4              =  14
p [3,5,7,5,-2,4]    =  3*5 + 7*5 + (-2)*4     =  42
p []                                          =  0
p [1,2,3]                                     =  error
```

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]
foldr   :: (a -> b -> b) -> b -> [a] -> b
```

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

3. The following data type represents expressions built from variables, sums, and products.

```
data Expr = Var String
          | Expr :*: Expr
          | Expr :+: Expr
```

The template file includes code that enables QuickCheck to generate arbitary values of type `Expr`, to aid testing.

(a) Write two functions `isTerm, isNorm :: Expr -> Bool` that return true when the given expression is a term or is normal, respectively. We say that an expression is a *term* if it is a product of variables, that is, it is a variable or the product of two expressions that are terms. We say that an expression is *normal* if it is a sum of terms, that is, if it is a term or the sum of two expressions that are normal. For example,

```
isTerm (Var "x")                                          =  True
isTerm ((Var "x" :*: Var "y") :*: Var "z")                =  True
isTerm ((Var "x" :*: Var "y") :+: Var "z")                =  False
isTerm (Var "x" :*: (Var "y" :+: Var "z"))                =  False

isNorm (Var "x")                                          =  True
isNorm (Var "x" :*: Var "y" :*: Var "z")                  =  True
isNorm ((Var "x" :*: Var "y") :+: Var "z")                =  True
isNorm (Var "x" :*: (Var "y" :+: Var "z"))                =  False
isNorm ((Var "x" :*: Var "y") :+: (Var "x" :*: Var "z"))  =  True
isNorm ((Var "u" :+: Var "v") :*: (Var "x" :+: Var "y"))  =  False
isNorm (((Var "u" :*: Var "x") :+: (Var "u" :*: Var "y")) :+:
         ((Var "v" :*: Var "x") :+: (Var "v" :*: Var "y")))  =  True
```

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

(b) Write a function `norm :: Expr -> Expr` that converts an expression to an equivalent expression in normal form. An expression not in normal form may be converted to normal form by repeated application of the distributive laws,

$$\begin{aligned}(a+b)\times c &= (a\times c)+(b\times c)\\ a\times(b+c) &= (a\times b)+(a\times c)\end{aligned}$$

For example,

```
norm (Var "x")
  =  (Var "x")
norm ((Var "x" :*: Var "y") :*: Var "z")
  =  ((Var "x" :*: Var "y") :*: Var "z")
```

```
norm ((Var "x" :*: Var "y") :+: Var "z")
  =  ((Var "x" :*: Var "y") :+: Var "z")
norm (Var "x" :*: (Var "y" :+: Var "z"))
  =  ((Var "x" :*: Var "y") :+: (Var "x" :*: Var "z"))
norm ((Var "u" :+: Var "v") :*: (Var "x" :+: Var "y"))
  =  (((Var "u" :*: Var "x") :+: (Var "u" :*: Var "y")) :+:
      ((Var "v" :*: Var "x") :+: (Var "v" :*: Var "y")))
```

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