UNIVERSITY OF EDINBURGH

COLLEGE OF SCIENCE AND ENGINEERING

SCHOOL OF INFORMATICS

INFORMATICS 1 - FUNCTIONAL PROGRAMMING

Wednesday 15th August 2012

09:30 to 11:30

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.**

THIS EXAMINATION WILL BE MARKED ANONYMOUSLY

1. (a) Write a function `f :: [String] -> String` to concatenate every word in a list that begins with an upper case letter. For example,

```
f ["This","Is","not","A","non","Test"]          =  "ThisIsATest"
f ["noThing","beGins","uPPER"]                   =  ""
f ["Non-words","like","42","get","Dropped"]     =  "Non-wordsDropped"
f ["An","Empty","Word","","gets","dropped"]     =  "AnEmptyWord"
```

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 :: [String] -> String` that behaves like `f`, this time using *basic functions*, *recursion*, and the *library function to append two lists*, 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)] -> Bool` that given a non-empty list of pairs returns true if the second component of each pair (save the last) is equal to the first component of the next pair. The function should give an error if given the empty list. For example:

```
p [(1,2),(2,3),(3,4)]          =  True
p [(9,5),(5,5),(5,7),(7,-2)]   =  True
p [(1,2),(3,4)]                =  False
p [(1,2),(2,3),(33,4)]         =  False
p []                           =  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)] -> 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,Int)] -> Bool` 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 comprehension 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 constants, variables, sums, and products.

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

(a) Write a function `isSimple :: Expr -> Bool` that returns true when the given expression is simple. We say that an expression is *simple* if it does not contain any of the following:

- a sum of two constants,
- a product of two constants,
- a sum where one of the summands is the constant zero,
- a product where one of the factors is the constant zero or one.

For example,

```
isSimple (Var "x" :+: Var "y")                 =  True
isSimple (Const 1 :+: Const 2)                 =  False
isSimple (Const 0 :+: Var "x")                 =  False
isSimple ((Var "y" :+: Const 1) :*: Var "x")   =  True
isSimple (Var "y" :+: (Var "x" :*: Const 0))   =  False
isSimple (Var "x" :*: (Const 3 :+: Const (-2)))  =  False
```

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

(b) Write a function `simplify :: Expr -> Expr` that converts an expression to an equivalent simple expression. To simplify an expression, compute the sum and products of constants, and apply the laws stating that 0 is an identity for sums, 1 is an identity for multiplication, and 0 is a zero for multiplication.

$$\begin{aligned} 0 + a &= a = a + 0 \\ 1 \times a &= a = a \times 1 \\ 0 \times a &= 0 = a \times 0 \end{aligned}$$

For example,

```
simplify (Var "x" :+: Var "y")
      =  (Var "x" :+: Var "y")
simplify (Const 1 :+: Const 2)
      =  Const 3
simplify (Const 0 :+: Var "x")
      =  Var "x"
simplify ((Var "y" :+: Const 1) :*: Var "x")
      =  ((Var "y" :+: Const 1) :*: Var "x")
```

```
simplify (Var "y" :+: (Var "x" :*: Const 0))
      =  Var "y"
simplify (Var "x" :*: (Const 3 :+: Const (-2)))
      =  Var "x"
```

Credit may be given for indicating how you have tested your function. The template file includes code that enables QuickCheck to generate arbitary values of type `Expr`, to aid testing. [*16 marks*]