UNIVERSITY OF EDINBURGH

COLLEGE OF SCIENCE AND ENGINEERING

SCHOOL OF INFORMATICS

INFR08013 INFORMATICS 1 - FUNCTIONAL PROGRAMMING

Monday 15 th December 2014

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: notes and printed material are allowed, and USB sticks, but no electronic devices.**
4. **CALCULATORS MAY NOT BE USED IN THIS EXAMINATION**

Convener: D. K. Arvind
External Examiner: C. Johnson

THIS EXAMINATION WILL BE MARKED ANONYMOUSLY

1. (a) Write a function `f :: [Int] -> Bool` that, given a non-empty list of non-zero numbers, returns `True` if each successive number (except the first) is divisible by its predecessor in the list. The function should give an error if applied to the empty list; you may assume without test that all numbers are non-zero. For example:

```
f [1,1,-2,6,18,-18,180]   =  True
f [17]                    =  True
f [1,1,2,3,6,18]          =  False
f [1,2,6,3,9]             =  False
```

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 :: [Int] -> Bool` that behaves like `f`, 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*]

2. (a) Write a function `p :: [Int] -> Int` that computes the product of the squares of the negative numbers in a list. For example:

```
p [13]              = 1
p []                = 1
p [-3,3,1,-3,2,-1]  = 81
p [2,6,-3,0,3,-7,2] = 441
p [4,-2,-1,-3]      = 36
```

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 `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. [*12 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 arithmetic expressions over a single variable:

```
data Expr = X                        -- variable
          | Const Int                -- integer constant
          | Expr :+: Expr            -- addition
          | Expr :-: Expr            -- subtraction
          | Expr :*: Expr            -- multiplication
          | Expr :/: Expr            -- integer division
          | IfZero Expr Expr Expr    -- conditional expression
```

`IfZero p q r` represents the expression that would be written in Haskell as `if p==0 then q else r`.

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

(a) Write a function `eval :: Expr -> Int -> Int`, which given an expression and the value of the variable `X` returns the value of the expression. For example,

```
eval (X :+: (X :*: Const 2)) 3                        = 9
eval (X :/: Const 3) 7                                = 2
eval (IfZero (X :-: Const 3) (X:/:X) (Const 7)) 3    = 1
eval (IfZero (X :-: Const 3) (X:/:X) (Const 7)) 4    = 7
eval (Const 15 :-: (Const 7 :/: (X :-: Const 1))) 0 = 22
```

but both of the following should produce a divide-by-zero exception:

```
eval (Const 15 :-: (Const 7 :/: (X :-: Const 1))) 1
eval (X :/: (X :-: X)) 2
```

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

(b) Write a function `protect :: Expr -> Expr` that protects against divide-by-zero exceptions by "guarding" all uses of division with a test for a zero-valued denominator. In this case the result should be `maxBound` (the maximum value of type `Int`, which is platform dependent). Do *not* attempt to simplify the result by omitting tests that appear to be unnecessary. For example,

```
protect (X :+: (X :*: Const 2))
                = (X :+: (X :*: Const 2))
protect (X :/: Const 3)
              = IfZero (Const 3) (Const maxBound) (X :/: Const 3)
```

QUESTION CONTINUES ON NEXT PAGE

```
protect (IfZero (X :-: Const 3) (X:/:X) (Const 7))
              = IfZero (X :-: Const 3)
                       (IfZero X (Const maxBound) (X :/: X))
                       (Const 7)
protect (Const 15 :-: (Const 7 :/: (X :-: Const 1)))
              = (Const 15 :-: (IfZero (X :-: Const 1)
                                      (Const maxBound)
                                      (Const 7 :/: (X :-: Const 1))))
protect (X :/: (X :-: X))
              = IfZero (X :-: X) (Const maxBound) (X :/: (X :-: X))
```

which, when evaluated, give the following results:

```
eval (protect (X :+: (X :*: Const 2))) 3                    = 9
eval (protect (X :/: Const 3)) 7                            = 2
eval (protect (IfZero (X :-: Const 3) (X:/:X) (Const 7))) 3   = 1
eval (protect (IfZero (X :-: Const 3) (X:/:X) (Const 7))) 4   = 7
eval (protect (Const 15 :-: (Const 7 :/: (X :-: Const 1)))) 0 = 22
eval (protect (Const 15 :-: (Const 7 :/: (X :-: Const 1)))) 1
                                                 = (15-maxBound)
eval (protect (X :/: (X :-: X))) 2               = maxBound
```

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