Module Title: Informatics 1 — Functional Programming (second sitting)
Exam Diet (Dec/April/Aug): December 2014
Brief notes on answers:

```
-- Full credit is given for fully correct answers.
-- Partial credit may be given for partly correct answers.
-- Additional partial credit is given if there is indication of testing,
-- either using examples or quickcheck, as shown below.

import Test.QuickCheck( quickCheck,
                        Arbitrary( arbitrary ),
                        oneof, elements, sized, (==>)  )
import Control.Monad -- defines liftM, liftM2, liftM4, used below
import Data.Char

-- Question 1

-- 1a

bigger :: Int -> Int -> Bool
x `bigger` y  =  x >= 2*y

f :: [Int] -> Bool
f xs | not (null xs) = and [ x' `bigger` x | (x,x') <- zip xs (tail xs) ]

test1a =
  f [1,2,7,18,47,180]  ==  True &&
  f [17]               ==  True &&
  f [1,3,5,16,42]      ==  False &&
  f [1,2,6,6,13]       ==  False

-- 1b

g :: [Int] -> Bool
g [x]        =  True
g (x:x':xs)  =  x' `bigger` x && g (x':xs)

test1b =
  g [1,2,7,18,47,180]  ==  True &&
  g [17]               ==  True &&
  g [1,3,5,16,42]      ==  False &&
  g [1,2,6,6,13]       ==  False

prop1 xs = not (null xs) ==> f xs == g xs
check1 = quickCheck prop1
```

```haskell
-- Question 2

-- 2a

p :: [Int] -> Int
p xs  =  sum [ x*x*x | x<-xs, x>0 ]

test2a =
  p [-13]               ==  0 &&
  p []                  ==  0 &&
  p [-3,3,1,-3,2,-1]    ==  36 &&
  p [2,6,-3,0,3,-7,2]   ==  259 &&
  p [4,-2,-1,-3]        ==  64

-- 2b

q :: [Int] -> Int
q []                  =  0
q (x:xs) | x>0        =  (x*x*x) + q xs
         | otherwise  =  q xs

test2b =
  q [-13]               ==  0 &&
  q []                  ==  0 &&
  q [-3,3,1,-3,2,-1]    ==  36 &&
  q [2,6,-3,0,3,-7,2]   ==  259 &&
  q [4,-2,-1,-3]        ==  64

-- 2c

r :: [Int] -> Int
r xs = foldr (+) 0 (map (\x -> x*x*x) (filter (>0) xs))

test2c =
  r [-13]               ==  0 &&
  r []                  ==  0 &&
  r [-3,3,1,-3,2,-1]    ==  36 &&
  r [2,6,-3,0,3,-7,2]   ==  259 &&
  r [4,-2,-1,-3]        ==  64

prop2 xs = p xs == q xs && q xs == r xs
check2 = quickCheck prop2
```

```
-- Question 3

data Expr = X
          | Const Integer
          | Expr :+: Expr
          | Expr :-: Expr
          | Expr :*: Expr
          | IfLt Expr Expr Expr Expr
          deriving (Eq, Ord)

-- turns an Expr into a string approximating mathematical notation

showExpr :: Expr -> String
showExpr X          =  "X"
showExpr (Const n)  =  show n
showExpr (p :+: q)  =  "(" ++ showExpr p ++ "+" ++ showExpr q ++ ")"
showExpr (p :-: q)  =  "(" ++ showExpr p ++ "-" ++ showExpr q ++ ")"
showExpr (p :*: q)  =  "(" ++ showExpr p ++ "*" ++ showExpr q ++ ")"
showExpr (IfLt p q r s)  = "(if " ++ showExpr p ++ "<"
                                  ++ showExpr q ++ " then "
                                  ++ showExpr r ++ " else "
                                  ++ showExpr s ++ ")"

-- For QuickCheck

instance Show Expr where
    show  =  showExpr

instance Arbitrary Expr where
    arbitrary  =  sized expr
        where
          expr n | n <= 0     =  oneof [elements [X]]
                 | otherwise  =  oneof [ liftM Const arbitrary
                                       , liftM2 (:+:) subform2 subform2
                                       , liftM2 (:-:) subform2 subform2
                                       , liftM2 (:*:) subform2 subform2
                                       , liftM4 (IfLt) subform4 subform4 subform4 sub
                                       ]
                 where
                   subform2  =  expr (n `div` 2)
                   subform4  =  expr (n `div` 4)

-- 3a

eval :: Expr -> Integer -> Integer
eval X v               =  v
eval (Const n) _       =  n
eval (p :+: q) v       =  (eval p v) + (eval q v)
```

```
eval (p :-: q) v       =  (eval p v) - (eval q v)
eval (p :*: q) v       =  (eval p v) * (eval q v)
eval (IfLt p q r s) v  =  if (eval p v) < (eval q v)
                          then eval r v else eval s v

test3a =
  eval (X :+: (X :*: Const 2)) 3 == 9 &&
  eval (X :-: (X :*: Const 3)) 0 == 0 &&
  eval (X :-: (X :*: Const 3)) 7 == -14 &&
  eval (X :+: X) 2 == 4 &&
  eval (Const 15 :+: (Const 7 :*: (X :-: Const 1))) 0 == 8 &&
  eval (X :-: (X :+: X)) 4 == -4

-- 3 b

protect :: Expr -> Expr
protect X              =  X
protect (Const n)      =  if n<0 then Const 0 else Const n
protect (p :+: q)      =  (protect p) :+: (protect q)
protect (p :-: q)
   =  IfLt (protect p) (protect q) (Const 0) ((protect p) :-: (protect q))
protect (p :*: q)      =  (protect p) :*: (protect q)
protect (IfLt p q r s)
   =  IfLt (protect p) (protect q) (protect r) (protect s)

test3b =
  protect (X :+: (X :*: Const 2))
     == (X :+: (X :*: Const 2)) &&
  protect (X :-: (X :*: Const 3))
     == IfLt X (X :*: Const 3) (Const 0) (X :-: (X :*: Const 3)) &&
  protect (X :+: X)
     == X :+: X &&
  protect (Const 15 :+: (Const 7 :*: (X :-: Const 1)))
     == Const 15 :+: (Const 7 :*: IfLt X (Const 1) (Const 0) (X :-: Const 1)) &&
  protect (X :-: (X :+: X))
     == IfLt X (X :+: X) (Const 0) (X :-: (X :+: X))

test3b' =
  eval (protect (X :+: (X :*: Const 2))) 3 == 9 &&
  eval (protect (X :-: (X :*: Const 3))) 0 == 0 &&
  eval (protect (X :-: (X :*: Const 3))) 7 == 0 &&
  eval (protect (X :+: X)) 2 == 4 &&
  eval (protect (Const 15 :+: (Const 7 :*: (X :-: Const 1)))) 0 == 15 &&
  eval (protect (X :-: (X :+: X))) 4 == 0

-- the following example requires
--   protect (p :-: q) = IfLt (protect p) (protect q) ...
-- rather than
```

```
--   protect (p :-: q) = IfLt p q ...
trickytest = ((((Const (-121)):-:(Const 11)):-:(Const (-187))):-:(Const 51))
test3b'' = eval (protect trickytest) 0 == 0

-- check that evaluation is never negative
-- this will fail
prop3 p n  =  n >= 0 ==> (eval p n >=0)
check3 = quickCheck prop3

-- check that evaluation of protected expression is never negative
-- this will succeed
prop3' p n  =  n >= 0 ==> eval (protect p) n >=0
check3' = quickCheck prop3'
```