Module Title: Informatics 1 — Functional Programming (first 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, liftM3, used below
import Data.Char

-- Question 1

-- 1a

divBy :: Int -> Int -> Bool
x `divBy` y  =  (x `mod` y == 0)

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

test1a =
  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

-- 1b

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

test1b =
  g [1,1,-2,6,18,-18,180]  ==  True &&
  g [17]                   ==  True &&
  g [1,1,2,3,6,18]         ==  False &&
  g [1,2,6,3,9]            ==  False

prop1 xs = not (null xs) && all (\x -> x/=0) xs ==> f xs == g xs
check1 = quickCheck prop1
```

```
-- Question 2

-- 2a

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

test2a =
  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

-- 2b

q :: [Int] -> Int
q []                  =  1
q (x:xs) | x<0        =  (x*x) * q xs
         | otherwise  =  q xs

test2b =
  q [13]               ==  1 &&
  q []                 ==  1 &&
  q [-3,3,1,-3,2,-1]   ==  81 &&
  q [2,6,-3,0,3,-7,2]  ==  441 &&
  q [4,-2,-1,-3]       ==  36

-- 2c

r :: [Int] -> Int
r xs = foldr (*) 1 (map (\x -> x*x) (filter (<0) xs))

test2c =
  r [13]               ==  1 &&
  r []                 ==  1 &&
  r [-3,3,1,-3,2,-1]   ==  81 &&
  r [2,6,-3,0,3,-7,2]  ==  441 &&
  r [4,-2,-1,-3]       ==  36

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

```
-- Question 3

data Expr = X
          | Const Int
          | Expr :+: Expr
          | Expr :-: Expr
          | Expr :*: Expr
          | Expr :/: Expr
          | IfZero 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 (p :/: q)  =  "(" ++ showExpr p ++ "/" ++ showExpr q ++ ")"
showExpr (IfZero p q r)  = "(if " ++ showExpr p ++ "=0 then "
                                  ++ showExpr q ++ " else "
                                  ++ showExpr r ++ ")"

-- 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
                                       , liftM2 (:/:) subform2 subform2
                                       , liftM3 (IfZero) subform3 subform3 subform3
                                       ]
                 where
                   subform2  =  expr (n `div` 2)
                   subform3  =  expr (n `div` 3)

-- 3a

eval :: Expr -> Int -> Int
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 (p :/: q) v       =  (eval p v) `div` (eval q v)
eval (IfZero p q r) v  =  if (eval p v)==0 then eval q v else eval r v

test3a =
  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

-- should produce exception: divide by zero
test3a' = eval (Const 15 :-: (Const 7 :/: (X :-: Const 1))) 1
test3a'' = eval (X :/: (X :-: X)) 2

-- 3 b

protect :: Expr -> Expr
protect X               =  X
protect (Const n)       =  (Const n)
protect (p :+: q)       =  (protect p) :+: (protect q)
protect (p :-: q)       =  (protect p) :-: (protect q)
protect (p :*: q)       =  (protect p) :*: (protect q)
protect (p :/: q)
   =  IfZero (protect q) (Const maxBound) ((protect p) :/: (protect q))
protect (IfZero p q r)
   =  IfZero (protect p) (protect q) (protect r)

test3b =
  protect (X :+: (X :*: Const 2)) == (X :+: (X :*: Const 2)) &&
  protect (X :/: Const 3)
     == IfZero (Const 3) (Const maxBound) (X :/: Const 3) &&
  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 :-: Co
  protect (X :/: (X :-: X))
     == IfZero (X :-: X) (Const maxBound) (X :/: (X :-: X))

test3b' =
  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

-- the following example requires
--   protect (p :/: q) = IfZero (protect q) ...
-- rather than
--   protect (p :/: q) = IfZero q ...

trickytest = X :/: (X :/: X)
test3b'' = eval (protect trickytest) 0 == 0

-- check equality to test that evaluation doesn't raise exception
-- this will fail
prop3 p n  =  eval p n == eval p n
check3 = quickCheck prop3

-- check equality to test that evaluation of protected expression
-- doesn't raise exception
-- this will succeed
prop3' p n  =  eval (protect p) n == eval (protect p) n
check3' = quickCheck prop3'
```