Module Title: Informatics 1 - Functional Programming
Exam Diet (Dec/April/Aug): August 2013
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, used below

-- Question 1

-- 1a

f :: [(Int,Int)] -> [Int]
f xs = [ if i `mod` 2 == 0 then a else b | (i,(a,b)) <- zip [0..] xs ]

test1a =
    f [(1,2),(5,7),(3,8),(4,9)]  ==  [1,7,3,9]
 && f [(1,2)]                    ==  [1]
 && f []                         ==  []

-- 1b

g :: [(Int,Int)] -> [Int]
g []                 =  []
g [(a,b)]            =  [a]
g ((a,b):(a',b'):xs) =  a : b' : g xs

test1b =
    g [(1,2),(5,7),(3,8),(4,9)]  ==  [1,7,3,9]
 && g [(1,2)]                    ==  [1]
 && g []                         ==  []

test1 = test1a && test1b
prop_1 xs = f xs == g xs
check1 = quickCheck prop_1

-- Question 2

-- 2a

isOdd :: Int -> Bool
isOdd i = i `mod` 2 == 1
```

```
p :: [Int] -> Int
p xs = product [ 3*x | x <- xs, isOdd x, x >= 0 ]

test2a =
    p [1,6,-15,11,-9]  ==  99
 && p [3,6,9,12,-9,9]  ==  6561
 && p []               ==  1
 && p [-1,4,-15]       ==  1

-- 2b

q :: [Int] -> Int
q []                         =  1
q (x:xs) | isOdd x && x >=0  =  3*x * q xs
         | otherwise         =  q xs

test2b =
    q [1,6,-15,11,-9]  ==  99
 && q [3,6,9,12,-9,9]  ==  6561
 && q []               ==  1
 && q [-1,4,-15]       ==  1

-- 2c

r :: [Int] -> Int
r xs = foldr (*) 1 (map (\x -> 3*x) (filter isPosOdd xs))
  where
  isPosOdd i = isOdd i && i >=0

test2c =
    r [1,6,-15,11,-9]  ==  99
 && r [3,6,9,12,-9,9]  ==  6561
 && r []               ==  1
 && r [-1,4,-15]       ==  1

test2 = test2a && test2b && test2c
prop_2 xs = p xs == q xs && q xs == r xs
check2 = quickCheck prop_2

-- Question 3

data Prop = X
          | F
          | T
          | Not Prop
          | Prop :<->: Prop
          deriving (Eq, Ord)
```

```
-- turns a Prop into a string approximating mathematical notation

showProp :: Prop -> String
showProp X           =  "X"
showProp F           =  "F"
showProp T           =  "T"
showProp (Not p)     =  "(~" ++ showProp p ++ ")"
showProp (p :<->: q) =  "(" ++ showProp p ++ "<->" ++ showProp q ++ ")"

-- For QuickCheck

instance Show Prop where
    show  =  showProp

instance Arbitrary Prop where
    arbitrary  =  sized prop
        where
          prop n | n <= 0     =  atom
                 | otherwise  =  oneof [ atom
                                       , liftM Not subform
                                       , liftM2 (:<->:) subform subform
                                       ]
                 where
                   atom = oneof [elements [X,F,T]]
                   subform  =  prop (n `div` 2)

-- 3a

eval :: Prop -> Bool -> Bool
eval X v           =  v
eval F _           =  False
eval T _           =  True
eval (Not p) v     =  not (eval p v)
eval (p :<->: q) v  =  (eval p v) == (eval q v)

test3a =
    eval (Not T) True                        ==  False
 && eval (Not X) False                       ==  True
 && eval (Not X :<->: Not (Not X)) True      ==  False
 && eval (Not X :<->: Not (Not X)) False     ==  False
 && eval (Not (Not X :<->: F)) True          ==  False
 && eval (Not (Not X :<->: F)) False         ==  True

-- 3 b

simplify :: Prop -> Prop
simplify X         =  X
```

```
simplify F          =  F
simplify T          =  T
simplify (Not p)    =  negate (simplify p)
  where
    negate T        =  F
    negate F        =  T
    negate (Not p)  =  p
    negate p        =  Not p
simplify (p :<->: q)  =  equalify (simplify p) (simplify q)
  where
    equalify T p              =  p
    equalify F p              =  Not p
    equalify p T              =  p
    equalify p F              =  Not p
    equalify p q | p == q     =  T
                 | otherwise  =  p :<->: q

test3b =
    simplify (Not F)                          ==  T
 && simplify (Not X :<->: Not (X :<->: T))  ==  T
 && simplify (Not (Not X :<->: Not T))      ==  Not X
 && simplify (Not (F :<->: Not (Not X)))    ==  X

test3 = test3a && test3b
prop_3 p =
    eval p True == eval (simplify p) True
    && eval p False == eval (simplify p) False
    && length (showProp p) >= length (showProp (simplify p))
check3 = quickCheck prop_3
```