Module Title: Informatics 1 — Functional Programming
Exam Diet (Dec/April/Aug): August 2016
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, liftM3, used below
import Data.List
import Data.Char

-- Question 1

-- 1a

f :: String -> Int
f xs = sum [ digitToInt x * 2^i | (x,i) <- zip (reverse xs) [0..] ]

test1a =
  f "101" == 5 &&
  f "11" == 3 &&
  f "1101" == 13 &&
  f "110111" == 55

-- 1b

g :: String -> Int
g xs = g' 0 (reverse xs)
  where
    g' i [] = 0
    g' i (x:xs) = digitToInt x * 2^i + g' (i+1) xs

test1b =
  g "101" == 5 &&
  g "11" == 3 &&
  g "1101" == 13 &&
  g "110111" == 55

binary s = all (\c -> '0' <= c && c <= '1') s

prop1 s = binary s ==> f s == g s

-- Question 2
```

```
-- 2a

div3 :: Int -> Bool
div3 x = x `mod` 3 == 0

p :: [Int] -> Bool
p xs = and [ odd x | x <- xs, div3 x ]

test2a =
    p [1,15,153,83,64,9]  ==  True &&
    p [1,12,153,83,9]     ==  False &&
    p []                  ==  True &&
    p [2,151]             ==  True

-- 2b

q :: [Int] -> Bool
q []                              = True
q (x:xs) | div3 x && not(odd x) = False
         | otherwise              = q xs

test2b =
    q [1,15,153,83,64,9]  ==  True &&
    q [1,12,153,83,9]     ==  False &&
    q []                  ==  True &&
    q [2,151]             ==  True

-- 2c

r :: [Int] -> Bool
r xs = foldr (&&) True (map odd (filter div3 xs))

test2c =
    r [1,15,153,83,64,9]  ==  True &&
    r [1,12,153,83,9]     ==  False &&
    r []                  ==  True &&
    r [2,151]             ==  True

prop2 xs = p xs == q xs && q xs == r xs

-- 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   =  if eval p v then eval q v else True

test3a =
    eval (Not T) True                      ==  False &&
    eval (Not X) False                     ==  True &&
    eval (Not X :->: Not (Not X)) True     ==  True &&
    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)    =  negify (simplify p)
simplify (p :->: q) =  implify (simplify p) (simplify q)

negify :: Prop -> Prop
negify T        =  F
negify F        =  T
negify (Not p)  =  p
negify p        =  Not p

implify :: Prop -> Prop -> Prop
implify T p     =  p
implify F p     =  T
implify p T     =  T
implify p F     =  negify p
implify p q     =  p :->: q

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

prop3 p =
    eval p True == eval (simplify p) True
    && eval p False == eval (simplify p) False
    && length (showProp p) >= length (showProp (simplify p))
```