Module Title: Informatics 1 — Functional Programming (morning sitting)
Exam Diet (Dec/April/Aug): December 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, (==>), Property )
import Control.Monad -- defines liftM, liftM3, used below
import Data.List
import Data.Char

-- Question 1

-- 1a

f :: [Int] -> [Int] -> Int
f ns ms = sum [ n | (n,m) <- zip ns ms, m `divides` n ]

divides :: Int -> Int -> Bool
m `divides` n = n `mod` m == 0

test1a =
  f [6,9,2,7] [2,3,5,1] == 22 &&
  f [6,9,2] [2,3,5,1] == 15 &&
  f [1,2,3,4,5] [5,4,3,2,1] == 12 &&
  f [10,20,30,40] [3,4,5,6,7] == 50

-- 1b

g :: [Int] -> [Int] -> Int
g [] _ = 0
g _ [] = 0
g (n:ns) (m:ms) | m `divides` n = n + g ns ms
                | otherwise     = g ns ms

test1b =
  g [6,9,2,7] [2,3,5,1] == 22 &&
  g [6,9,2] [2,3,5,1] == 15 &&
  g [1,2,3,4,5] [5,4,3,2,1] == 12 &&
  g [10,20,30,40] [3,4,5,6,7] == 50


prop1 :: [Int] -> [Int] -> Property
```

```
prop1 ns ms = and [ m /=0 | m <- ms ] ==> f ns ms == g ns ms

-- Question 2

-- 2a

p :: String -> Int
p cs = maximum (0:[ digitToInt c | c <- cs, isDigit c ])

test2a =
  p "Inf1-FP" == 1 &&
  p "Functional" == 0 &&
  p "1+1=2" == 2 &&
  p "3.157/3 > 19" == 9

-- 2b

q :: String -> Int
q [] = 0
q (c:cs) | isDigit c = max (digitToInt c) (q cs)
         | otherwise = q cs

test2b =
  q "Inf1-FP" == 1 &&
  q "Functional" == 0 &&
  q "1+1=2" == 2 &&
  q "3.157/3 > 19" == 9

-- 2c

r :: String -> Int
r cs = foldr max 0 (map digitToInt (filter isDigit cs))

test2c =
  r "Inf1-FP" == 1 &&
  r "Functional" == 0 &&
  r "1+1=2" == 2 &&
  r "3.157/3 > 19" == 9

prop2 :: String -> Bool
prop2 cs = p cs == q cs && q cs == r cs

-- Question 3

data Move =
     Go Int            -- move the given distance in the current direction
   | Turn              -- reverse direction
   | Dance             -- dance in place, without changing direction
  deriving (Eq,Show)   -- defines obvious == and show

data Command =
     Nil                        -- do nothing
   | Command :#: Move           -- do a command followed by a move
  deriving Eq                   -- defines obvious ==

instance Show Command where   -- defines show :: Command -> String
  show Nil = "Nil"
  show (com :#: mov) = show com ++ " :#: " ++ show mov

type Position = Int
data Direction = L | R
  deriving (Eq,Show)          -- defines obvious == and show
type State = (Position, Direction)

-- For QuickCheck

instance Arbitrary Move where
  arbitrary = sized expr
    where
      expr n | n <= 0 = elements [Turn, Dance]
             | otherwise = liftM (Go) arbitrary

instance Arbitrary Command where
  arbitrary = sized expr
    where
      expr n | n <= 0 = oneof [elements [Nil]]
             | otherwise = oneof [ liftM2 (:#:) subform arbitrary
                                 ]
             where
               subform = expr (n-1)

instance Arbitrary Direction where
  arbitrary = elements [L,R]

-- 3a

state :: Move -> State -> State
state (Go d) (n,L) = (n - d, L)
state (Go d) (n,R) = (n + d, R)
state Turn (c,L) = (c, R)
state Turn (c,R) = (c, L)
state Dance p = p

test3a =
  state (Go 3) (0,R) == (3,R) &&
  state (Go 3) (0,L) == (-3,L) &&
  state Turn (-2,L) == (-2,R) &&
  state Dance (4,R) == (4,R)

-- 3b

trace :: Command -> State -> [State]
trace Nil s = [s]
trace (com :#: mov) s = t ++ [state mov (last t)]
    where t = trace com s

test3b =
  trace (Nil) (3,R)
               == [(3,R)] &&
  trace (Nil :#: Go 3 :#: Turn :#: Go 4) (0,L)
               == [(0,L),(-3,L),(-3,R),(1,R)] &&
  trace (Nil :#: Go 3 :#: Dance :#: Turn :#: Turn) (0,R)
               == [(0,R),(3,R),(3,R),(3,L),(3,R)] &&
  trace (Nil :#: Go 3 :#: Turn :#: Go 2 :#: Go 1 :#: Turn :#: Go 4) (4,L)
               == [(4,L),(1,L),(1,R),(3,R),(4,R),(4,L),(0,L)]

-- 3c

samepos :: State -> [State] -> Bool
samepos (p,s) ss = p ‘elem‘ (map fst ss)

dancify :: Command -> Command
dancify Nil = Nil
dancify (com :#: Dance) = (dancify com) :#: Dance
dancify (com :#: m) | samepos (state m (last t)) t = (dancify com) :#: m :#: Dance
                    | otherwise                    = (dancify com) :#: m
         where t = trace com (0,R)

test3c =
  dancify Nil
         == Nil &&
  dancify (Nil :#: Go 3 :#: Turn :#: Go 4)
         == Nil :#: Go 3 :#: Turn :#: Dance :#: Go 4 &&
  dancify (Nil :#: Go 3 :#: Dance :#: Turn :#: Turn)
         == Nil :#: Go 3 :#: Dance :#: Turn :#: Dance :#: Turn :#: Dance &&
  dancify (Nil :#: Go 3 :#: Turn :#: Go 2 :#: Go 1 :#: Turn :#: Go 4)
         == Nil :#: Go 3 :#: Turn :#: Dance :#: Go 2 :#: Go 1 :#: Dance
                                            :#: Turn :#: Dance :#: Go 4
```