Module Title: Informatics 1 — Functional Programming
Exam Diet (Dec/April/Aug): August 2017
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 :: String -> [Int] -> String
f cs ns = concat [ replicate n c | (c,n) <- zip cs ns ]

test1a =
  f "abcde" [3,1,2,0,4] == "aaabcceeee" &&
  f "call" [3,-2,1,2,7] == "cccl1l" &&
  f "raisin" [1,2,3,4] == "raaiiissss" &&
  f "moose" [2] == "mm" &&
  f "" [1,2,3] == ""



-- 1b

g :: String -> [Int] -> String
g "" _ = ""
g _ [] = ""
g (c:cs) (n:ns) | n<=0      = g cs ns
                | otherwise = c : (g (c:cs) (n-1:ns))

test1b =
  g "abcde" [3,1,2,0,4] == "aaabcceeee" &&
  g "call" [3,-2,1,2,7] == "cccl1l" &&
  g "raisin" [1,2,3,4] == "raaiiissss" &&
  g "moose" [2] == "mm" &&
  g "" [1,2,3] == ""

prop1 :: String -> [Int] -> Bool
prop1 cs ns = f cs (small ns) == g cs (small ns)
```

```
 where small ns = filter (<100) ns

-- Question 2

-- 2a

p :: String -> Bool
p cs = and [ odd (digitToInt c) | c <- cs, isDigit c ]

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

-- 2b

q :: String -> Bool
q [] = True
q (c:cs) | isDigit c = odd (digitToInt c) && q cs
         | otherwise = q cs

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

-- 2c

r :: String -> Bool
r cs = foldr (&&) True (map (odd . digitToInt) (filter isDigit cs))

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

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

-- 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

finalstate :: Command -> State -> State
finalstate Nil s = s
finalstate (com :#: mov) s = state mov (finalstate com s)

test3b =
  finalstate (Nil) (3,R)                                      == (3,R) &&
  finalstate (Nil :#: Go 3 :#: Turn :#: Go 4) (0,L)           == (1,R) &&
  finalstate (Nil :#: Go 3 :#: Turn :#: Dance :#: Turn) (0,R) == (3,R) &&
  finalstate (Nil :#: Go 3 :#: Turn :#: Go 2 :#: Go 1 :#: Turn :#: Go 4) (4,L)
                                                              == (0,L)

-- 3c

simplify :: Command -> Command
simplify com | simp com == com = com
             | otherwise       = simplify (simp com)

simp :: Command -> Command
simp Nil = Nil
simp (Nil :#: mov) = Nil :#: mov
simp (com :#: Turn :#: Turn) = simp com
simp (com :#: Go n :#: Go m) = simp (com :#: Go (n+m))
simp (com :#: mov1 :#: mov2) = simp (com :#: mov1) :#: mov2

test3c =
  simplify Nil
        == Nil &&
  simplify (Nil :#: Go 3 :#: Go 1 :#: Go 4 :#: Dance)
        == Nil :#: Go 8 :#: Dance &&
  simplify (Nil :#: Go 3 :#: Turn :#: Dance :#: Turn)
        == Nil :#: Go 3 :#: Turn :#: Dance :#: Turn &&
  simplify (Nil :#: Go 3 :#: Turn :#: Turn :#: Go 2 :#: Go 1 :#: Turn :#: Go 4)
        == Nil :#: Go 6 :#: Turn :#: Go 4

prop3 :: Command -> State -> Bool
prop3 com s = finalstate com s == finalstate (simplify com) s
```