Module Title: Informatics 1 — Functional Programming (morning sitting)
Exam Diet (Dec/April/Aug): December 2015
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

p :: [Int] -> Int
p xs = (duration `div` 60) `mod` 12 + 1
  where
    duration = sum [ x | x <- xs, x>=0 ]

test1a =
  p [] == 1 &&
  p [-30,-20] == 1 &&
  p [20,-30,30,14,-20] == 2 &&
  p [200,45] == 5 &&
  p [60,-100,360,-20,240,59] == 12 &&
  p [60,-100,360,-20,240,60] == 1

-- 1b

q :: [Int] -> Int
q xs = (d xs `div` 60) `mod` 12 + 1
  where
    d :: [Int] -> Int
    d [] = 0
    d (x:xs) | x>=0      = x + d xs
             | otherwise = d xs

test1b =
  q [] == 1 &&
  q [-30,-20] == 1 &&
  q [20,-30,30,14,-20] == 2 &&
  q [200,45] == 5 &&
  q [60,-100,360,-20,240,50] == 12 &&
  q [60,-100,360,-20,240,70] == 1

-- 1c

r :: [Int] -> Int
r xs = (duration ‘div‘ 60) ‘mod‘ 12 + 1
  where
    duration = foldr (+) 0 (filter (>=0) xs)

test1c =
  r [] == 1 &&
  r [-30,-20] == 1 &&
  r [20,-30,30,14,-20] == 2 &&
  r [200,45] == 5 &&
  r [60,-100,360,-20,240,50] == 12 &&
  r [60,-100,360,-20,240,70] == 1

prop1 :: [Int] -> Bool
prop1 xs = p xs == q xs && q xs == r xs

-- Question 2

-- 2a

f :: String -> String
f "" = ""
f (c:cs) = c:[ b | (a,b) <- zip (c:cs) cs, a /= b ]

test2a =
  f "Tennessee" == "Tenese" &&
  f "llama" == "lama" &&
  f "oooh" == "oh" &&
  f "none here" == "none here" &&
  f "nNnor hEere" == "nNnor hEere" &&
  f "A" == "A" &&
  f "" == ""

-- 2b

g :: String -> String
g [] = []
g [x] = [x]
g (x:y:xs) | x == y = g (x:xs)
           | otherwise = x : g (y:xs)

test2b =
  g "Tennessee" == "Tenese" &&
  f "llama" == "lama" &&
  g "oooh" == "oh" &&
  g "none here" == "none here" &&
  g "nNnor hEere" == "nNnor hEere" &&
  g "A" == "A" &&
  g "" == ""

prop2 :: String -> Bool
prop2 cs = f cs == g cs

-- Question 3

data Regexp = Epsilon
            | Lit Char
            | Seq Regexp Regexp
            | Or Regexp Regexp
        deriving (Eq, Ord)

-- turns a Regexp into a string approximating normal regular expression notation

showRegexp :: Regexp -> String
showRegexp Epsilon = "e"
showRegexp (Lit c) = [toUpper c]
showRegexp (Seq r1 r2) = "(" ++ showRegexp r1 ++ showRegexp r2 ++ ")"
showRegexp (Or r1 r2) = "(" ++ showRegexp r1 ++ "|" ++ showRegexp r2 ++ ")"

-- for checking equality of languages

equal :: Ord a => [a] -> [a] -> Bool
equal xs ys = sort xs == sort ys

-- For QuickCheck

instance Show Regexp where
    show  =  showRegexp

instance Arbitrary Regexp where
  arbitrary = sized expr
    where
      expr n | n <= 0 = oneof [elements [Epsilon]]
             | otherwise = oneof [ liftM Lit arbitrary
                                 , liftM2 Seq subform subform
                                 , liftM2 Or subform subform
                                 ]
             where
               subform = expr (n `div` 2)
```

```
r1 = Seq (Lit 'A') (Or (Lit 'A') (Lit 'A'))   -- A(A|A)
r2 = Seq (Or (Lit 'A') Epsilon)
         (Or (Lit 'A') (Lit 'B'))             -- (A|e)(A|B)
r3 = Seq (Or (Lit 'A') (Seq Epsilon
                            (Lit 'A')))
         (Or (Lit 'A') (Lit 'B'))             -- (A|(eA))(A|B)
r4 = Seq (Or (Lit 'A')
             (Seq Epsilon (Lit 'A')))
         (Seq (Or (Lit 'A') (Lit 'B'))
              Epsilon)                        -- (A|(eA))((A|B)e)
r5 = Seq (Seq (Or (Lit 'A')
                  (Seq Epsilon (Lit 'A')))
              (Or Epsilon (Lit 'B')))
         (Seq (Or (Lit 'A') (Lit 'B'))
              Epsilon)                        -- ((A|(eA))(e|B))((A|B)e)
r6 = Seq (Seq Epsilon Epsilon)
         (Or Epsilon Epsilon)                 -- (ee)(e|e)

-- 3a

language :: Regexp -> [String]
language Epsilon = [""]
language (Lit c) = [[c]]
language (Seq r1 r2) = nub [ s1++s2 | s1 <- language r1, s2 <- language r2 ]
language (Or r1 r2) = nub (language r1 ++ language r2)

test3a =
  language r1 `equal` ["AA"] &&                   -- A(A|A)
  language r2 `equal` ["AA","AB","A","B"] &&      -- (A|e)(A|B)
  language r3 `equal` ["AA","AB"] &&              -- (A|(eA))(A|B)
  language r4 `equal` ["AA","AB"] &&              -- (A|(eA))((A|B)e)
  language r5 `equal` ["AA","AB","ABA","ABB"] &&  -- ((A|(eA))(e|B))((A|B)e)
  language r6 `equal` [""]                        -- (ee)(e|e)

-- 3b

simplify :: Regexp -> Regexp
simplify (Seq r1 r2)
           | simplify r1 == Epsilon = simplify r2
           | simplify r2 == Epsilon = simplify r1
           | otherwise              = Seq (simplify r1) (simplify r2)
simplify (Or r1 r2)
           | simplify r1 == simplify r2 = simplify r1
           | otherwise                  = Or (simplify r1) (simplify r2)
simplify r = r

test3b =
  simplify r1 ==
      Seq (Lit 'A') (Lit 'A') && -- A(A|A) = AA
  simplify r2 == r2 &&           -- (A|e)(A|B) is already simplified
  simplify r3 ==
      Seq (Lit 'A')
          (Or (Lit 'A')
              (Lit 'B')) &&      -- (A|(eA))(A|B) = A(A|B)
  simplify r4 ==
      Seq (Lit 'A')
          (Or (Lit 'A')
              (Lit 'B')) &&      -- (A|(eA))((A|B)e) = A(A|B)
  simplify r5 ==
      Seq (Seq (Lit 'A')
               (Or Epsilon (Lit 'B')))
          (Or (Lit 'A') (Lit 'B')) &&
                                 -- ((A|(eA))(e|B))((A|B)e) = (A(e|B))(A|B)
  simplify r6 == Epsilon         -- (ee)(e|e) = e

simple :: Regexp -> Bool
simple (Seq Epsilon _) = False
simple (Seq _ Epsilon) = False
simple (Seq r1 r2) = simple r1 && simple r2
simple (Or r1 r2) | r1==r2 = False
                  | otherwise = simple r1 && simple r2
simple r = True

prop3 :: Regexp -> Bool
prop3 r = simple (simplify r) && language r `equal` language (simplify r)
```