Module Title: Informatics 1 — Functional Programming (afternoon 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.

```haskell
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' 24) 'mod' 7 + 1
    where
        duration = sum [ x | x <- xs, x>=0 ]

    test1a =
        p [] == 1 &&
        p [-30,-20] == 1 &&
        p [12,-30,7,8,-20] == 2 &&
        p [90,15] == 5 &&
        p [90,-100,23,-20,54] == 7 &&
        p [90,-100,23,-20,55] == 1

-- 1b

q :: [Int] -> Int
q xs = (d xs 'div' 24) 'mod' 7 + 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 [12,-30,7,8,-20] == 2 &&
        q [90,15] == 5 &&
```

```
q [90, -100, 23, -20, 54] == 7 &&
q [90, -100, 23, -20, 55] == 1

-- 1c

r :: [Int] -> Int
r xs = (duration 'div' 24) 'mod' 7 + 1
  where
duration = foldr (+) 0 (filter (>=0) xs)

test1c =
r []  == 1 &&
r [-30,-20] == 1 &&
r [12,-30,7,8,-20] == 2 &&
r [90,15] == 5 &&
r [90,-100,23,-20,54] == 7 &&
r [90,-100,23,-20,55] == 1

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

-- Question 2

-- 2a

f :: String -> String
f "" = ""

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

test2a =
f "Tennessee" == "nse" &&
f "bookkeeper" == "oke" &&
f "llama hooves" == "lo" &&
f "www.dell.com" == "ww1" &&
f "ooooh" == "ooo" &&
f "nNnone here" == "" &&
f "" == ""

-- 2b

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

test2b =
g "Tennessee" == "nse" &&
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)
        (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 (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 (Lit 'B')
        (Seq (Lit 'A')
        (Or (Lit 'C') (Lit 'D'))) -- B(A(C|D))

r1' = Or (Seq (Lit 'A') (Lit 'A'))
        (Seq (Lit 'A') (Lit 'A')) -- (AA)|(AA)

r2' = Or (Seq (Or (Lit 'A') Epsilon)
        (Lit 'A'))
        (Seq (Or (Lit 'A') Epsilon)
        (Lit 'A'))) -- ((A|e)A)|((A|e)B)

r3' = Or (Seq (Or (Lit 'A')
        (Seq Epsilon (Lit 'A')))
        (Lit 'A'))
        (Seq (Or (Lit 'A')
        (Seq Epsilon (Lit 'A')))
        (Lit 'B'))) -- ((A|(eA))A | ((A|(eA))B)

r4' = r4

r5' = Seq (Or (Seq (Or (Lit 'A')
        (Seq Epsilon (Lit 'A'))))
        Epsilon)
        (Seq (Or (Lit 'A')
        (Seq Epsilon (Lit 'A'))))
        (Lit 'B'))) -- ((A|(eA))A | ((A|(eA))B)

r6' = Or (Seq (Lit 'B')
        (Seq (Lit 'A') (Lit 'C'))
        (Seq (Lit 'B')
        (Seq (Lit 'A') (Lit 'D'))) -- (B(AC))|(B(AD))

-- 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)
language r5 'equal' ["AA","AB","ABA","ABB"] && -- ((A|(eA))(e|B))(e|B)
language r6 'equal' ["BAC","BAD"] -- B(A(C|D))

**-- 3b**

flatten :: Regexp -> Regexp
flatten (Seq r1 (Or r2 r3)) = Or (flatten (Seq r1 r2)) (flatten (Seq r1 r3))
flatten (Seq r1 r2) | r1==r1' && r2==r2' = Seq r1 r2
| otherwise = flatten (Seq r1' r2')
where
r1' = flatten r1
r2' = flatten r2
flatten (Or r1 r2) = Or (flatten r1) (flatten r2)
flatten r = r

**test3b =**
flatten r1 == r1' && -- A(A|A) = (AA)|(AA)
flatten r2 == r2' && -- (A|e)(A|B) = ((A|e)A)|(A|e)B)
flatten r3 == r3' && -- (A|(eA))(A|B) = ((A|(eA))A)|(A|(eA)B)
flatten r4 == r4' && -- the left distributive law can’t be applied
flatten r5 == r5' && -- (A|(eA))(e|B)((A|B)e)
| -- = (((A|(eA))e)|((A|(eA))B)((A|B)e)
flatten r6 == r6' -- B(A(C|D)) = (BAC)|(B(AD))

flat :: Regexp -> Bool
flat (Seq _ (Or _ _)) = False
flat (Seq r1 r2) = flat r1 && flat r2
flat (Or r1 r2) = flat r1 && flat r2
flat r = True

**prop3 :: Regexp -> Bool**
prop3 r = flat (flatten r) && language r 'equal' language (flatten r)