Module Title: Informatics 1 - Functional Programming, SECOND SITTING
Exam Diet (Dec/April/Aug): December 2013
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, liftM2, used below
import Data.Char

-- Question 1

-- 1a

f :: String -> Int
f xs = sum [ digitToInt x * 4^i | (x,i) <- zip (reverse xs) [0..] ]

test1a =
  f "203" == 35 &&
  f "13" == 7 &&
  f "1302" == 114 &&
  f "130321" == 1849

-- 1b

g :: String -> Int
g xs = g' 0 (reverse xs)
  where
    g' i [] = 0
    g' i (x:xs) = digitToInt x * 4^i + g' (i+1) xs

test1b =
  g "203" == 35 &&
  g "13" == 7 &&
  g "1302" == 114 &&
  g "130321" == 1849

base4 s = all (\c -> '0' <= c && c <= '2') s

prop1 s = base4 s ==> f s == g s
check1 = quickCheck prop1

-- Question 2
```

```
-- 2a

muchBigger :: Int -> Int -> Bool
x `muchBigger` y  =  x >= 2*y

p :: [Int] -> Bool
p (a:xs)  =  and [ x `muchBigger` a | x <- xs, x >= 0 ]

test2a =
  p [2,6,-3,18,-7,10]  ==  True &&
  p [13]               ==  True &&
  p [-3,6,1,-6,9,18]   ==  True &&
  p [5,-2,-6,7]        ==  False

-- 2b

q :: [Int] -> Bool
q (a:xs)  =  q' xs
  where
    q' []  =  True
    q' (x:xs) | x >= 0     =  x `muchBigger` a && q' xs
              | otherwise  =  q' xs

test2b =
  q [2,6,-3,18,-7,10]  ==  True &&
  q [13]               ==  True &&
  q [-3,6,1,-6,9,18]   ==  True &&
  q [5,-2,-6,7]        ==  False

-- 2c

r :: [Int] -> Bool
r (a:xs)  =  foldr (&&) True (map (`muchBigger` a) (filter (>= 0) xs))

test2c =
  r [2,6,-3,18,-7,10]  ==  True &&
  r [13]               ==  True &&
  r [-3,6,1,-6,9,18]   ==  True &&
  r [5,-2,-6,7]        ==  False

prop2 xs = not (null xs) ==> p xs == q xs && q xs == r xs
check2 = quickCheck prop2

-- Question 3

data Expr = X
          | Const Int
```

```
          | Neg Expr
          | Expr :+: Expr
          | Expr :*: Expr
          deriving (Eq, Ord)

-- turns an Expr into a string approximating mathematical notation

showExpr :: Expr -> String
showExpr X          =  "X"
showExpr (Const n)  =  show n
showExpr (Neg p)    =  "(-" ++ showExpr p ++ ")"
showExpr (p :+: q)  =  "(" ++ showExpr p ++ "+" ++ showExpr q ++ ")"
showExpr (p :*: q)  =  "(" ++ showExpr p ++ "*" ++ showExpr q ++ ")"

-- evaluate an Expr, given a value of X

evalExpr :: Expr -> Int -> Int
evalExpr X v          =  v
evalExpr (Const n) _  =  n
evalExpr (Neg p) v    =  - (evalExpr p v)
evalExpr (p :+: q) v  =  (evalExpr p v) + (evalExpr q v)
evalExpr (p :*: q) v  =  (evalExpr p v) * (evalExpr q v)

-- For QuickCheck

instance Show Expr where
    show  =  showExpr

instance Arbitrary Expr where
    arbitrary  =  sized expr
        where
          expr n | n <= 0     =  oneof [elements [X]]
                 | otherwise  =  oneof [ liftM Const arbitrary
                                       , liftM Neg subform
                                       , liftM2 (:+:) subform subform
                                       , liftM2 (:*:) subform subform
                                       ]
                 where
                   subform  =  expr (n `div` 2)

-- 3a

ppn :: Expr -> [String]
ppn X  =  ["X"]
ppn (Const n)  =  [show n]
ppn (Neg p)    =  "-" : ppn p
ppn (p :+: q)  =  "+" : ppn p ++ ppn q
ppn (p :*: q)  =  "*" : ppn p ++ ppn q
```

```
test3a =
  ppn (X :*: Const 3) == ["*", "X", "3"] &&
  ppn (Neg (X :*: Const 3)) == ["-", "*", "X", "3"] &&
  ppn ((Const 5 :+: Neg X) :*: Const 17) == ["*", "+", "5", "-", "X", "17"] &&
  ppn ((Const 15 :+: Neg (Const 7 :*: (X :+: Const 1))) :*: Const 3)
        == ["*", "+", "15", "-", "*", "7", "+", "X", "1", "3"]

-- 3 b

evalppn :: [String] -> Int -> Int
evalppn s n = the (foldr step [] s)
    where
      step "+" (x:y:ys)  =  (y+x):ys
      step "*" (x:y:ys)  =  (y*x):ys
      step "-" (x:ys)    =  (-x):ys
      step "X" ys        =  n:ys
      step m ys | all (\c -> isDigit c || c=='-') m
                          =  (read m :: Int):ys
                | otherwise  =  error "ill-formed PPN"
      the :: [a] -> a
      the [x] = x
      the xs = error "ill-formed PPN"

test3b =
  evalppn ["*", "X", "3"] 10 == 30 &&
  evalppn ["-", "*", "X", "3"] 20 == -60 &&
  evalppn ["*", "+", "5", "-", "X", "17"] 10 == -85 &&
  evalppn ["*", "+", "15", "-", "*", "7", "+", "X", "1", "3"] 2 == -18

-- should produce exception: ill-formed PPN
test3b' =
  evalppn ["+", "-", "*", "X", "3"] 20

prop3 :: Expr -> Int -> Bool
prop3 p n =  evalExpr p n == evalppn (ppn p) n

check3 = quickCheck prop3
```