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 * 3^i | (x,i) <- zip (reverse xs) [0..] ] 
test1a = 
    f "201" == 19 && 
    f "12" == 5 && 
    f "1202" == 47 && 
    f "120221" == 430 

-- 1b 

g :: String -> Int 
g xs = g' 0 (reverse xs) 
where 
    g' i [] = 0 
    g' i (x:xs) = digitToInt x * 3^i + g' (i+1) xs 
test1b = 
    g "201" == 19 && 
    g "12" == 5 && 
    g "1202" == 47 && 
    g "120221" == 430 

base3 s = all (\c -> '0' <= c && c <= '2') s 
prop1 s = base3 s ==> f s == g s 
check1 = quickCheck prop1 

-- Question 2
-- 2a

divBy :: Int -> Int -> Bool
x `divBy` y = (x `mod` y == 0)

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

test2a =
  p [2,6,-3,0,18,-17,10] == True &&
  p [-13] == True &&
  p [-3,6,1,-3,9,18] == False &&
  p [5,-2,-6,3] == False

-- 2b

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

test2b =
  q [2,6,-3,0,18,-17,10] == True &&
  q [-13] == True &&
  q [-3,6,1,-3,9,18] == False &&
  q [5,-2,-6,3] == False

-- 2c

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

test2c =
  r [2,6,-3,0,18,-17,10] == True &&
  r [-13] == True &&
  r [-3,6,1,-3,9,18] == False &&
  r [5,-2,-6,3] == False

prop2 xs = not (null xs) && (head xs) /= 0
  ==> p xs == q xs && q xs == r xs
check2 = quickCheck prop2

-- Question 3

data Expr = X
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

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

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

-- 3 b

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

the :: [a] -> a
the [x] = x
the xs = error "ill-formed RPN"

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

-- should produce exception: ill-formed RPN

prop3 :: Expr -> Int -> Bool
prop3 p n = evalExpr p n == evalrpn (rpn p) n

check3 = quickCheck prop3