module Dec2017 where

import Test.QuickCheck( quickCheck,
    Arbitrary( arbitrary ), Gen, suchThat,
    oneof, elements, sized, (==>) )

import Control.Monad -- defines liftM, liftM2, liftM3, used below
import Data.Char

-- Question 1

f :: [Int] -> [Int]
f [] = []
f (n:ns) = [ j-i | (i,j) <- zip (n:ns) ns, i < j ]

test1a =
    f [4,2,5,6,1,8] == [3,1,7]
    && f [] == []
    && f [3] == []
    && f [3,3,1,-3] == []

-- Question 2

-- 2a

isInitialism :: String -> Bool
isInitialism s = length s > 1 && and [ isUpper c | c <- s ]
p :: [String] -> Int
p ss = sum [ length s | s <- ss, isInitialism s ]

test2a =
isInitialism "A" == False
&& isInitialism "AWOL" == True
&& isInitialism "Ltd" == False
&& p ["I","played","the","BBC","DVD","on","my","TV"] == 8
&& p ["The","DUP","MP","is","not","OK"] == 7
&& p ["The","SNP","won","in","South","Morningside"] == 3
&& p [] == 0

-- 2b

isInitialism' :: String -> Bool
isInitialism' [] = False
isInitialism' [c] = False
isInitialism' (c:c’:s) = isCaps (c:c’:s)

isCaps :: String -> Bool
isCaps [] = True
isCaps (c:s) = isUpper c && isCaps s

q :: [String] -> Int
q [] = 0
q (s:ss) | isInitialism’ s = length s + q ss
          | otherwise = q ss

test2b =
isInitialism' "A" == False
&& isInitialism' "AWOL" == True
&& isInitialism' "Ltd" == False
&& q ["I","played","the","BBC","DVD","on","my","TV"] == 8
&& q ["The","DUP","MP","is","not","OK"] == 7
&& q ["The","SNP","won","in","South","Morningside"] == 3
&& q [] == 0

-- 2c

r :: [String] -> Int
r ss = foldr (+) 0 (map length (filter isInitialism’ ss))

test2c =
r ["I","played","the","BBC","DVD","on","my","TV"] == 8
&& r ["The","DUP","MP","is","not","OK"] == 7
&& r ["The","SNP","won","in","South","Morningside"] == 3
&& r [] == 0
prop2 ss = p ss == q ss && q ss == r ss

-- Question 3

data Expr = X                         -- variable
  | Const Int                       -- integer constant >=0
  | Expr :+: Expr                  -- addition
  | Expr :*: Expr                  -- multiplication
    deriving (Eq, Ord)

-- turns an Expr into a string approximating mathematical notation

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

-- For QuickCheck

instance Show Expr where
  show = showExpr

instance Arbitrary Expr where
  arbitrary = sized expr
    where
      expr n | n <= 0     = oneof [ return X
                             , liftM Const genPos ]
      | otherwise = oneof [ return X
                             , liftM Const genPos
                             , liftM2 (:+:) subform2 subform2
                             , liftM2 (:*:) subform2 subform2 ]

    where
      subform2 = expr (n `div` 2)
      genPos = oneof [ return 0 , return 1 , return 2 , return 3 , return 4 , return 5 , return 6 , return 7 , return 8 , return 9 ]

-- 3a

eval :: Expr -> Int -> Int
eval X i       = i
eval (Const n) _ = n
eval (p :+: q) i = eval p i + eval q i
eval (p :*: q) i = eval p i * eval q i

test3a =
eval ((X :*: Const 3) :+: (Const 0 :*: X)) 2 == 6
&& eval (X :*: (Const 3 :+: Const 4)) 2 == 14
&& eval (Const 4 :+: (Const 3 :*: X)) 3 == 13
&& eval (((Const 1 :*: Const 2) :*: (X :+: Const 1)) :*: Const 2) 3 == 16

-- 3b

isSimple :: Expr -> Bool
isSimple X = True
isSimple (Const _) = True
isSimple (p :+: q) = isSimple p && isSimple q
isSimple (Const _ :*: q) = False
isSimple (p :*: q) = isSimple p && isSimple q

test3b =
  isSimple ((X :*: Const 3) :+: (Const 0 :*: X)) == False
&& isSimple (X :*: (Const 3 :+: Const 4)) == True
&& isSimple (Const 4 :+: (Const 3 :*: X)) == False
&& isSimple (((Const 1 :*: Const 2) :*: (X :+: Const 1)) :*: Const 2) == False

-- 3c

simplify :: Expr -> Expr
simplify X = X
simplify (Const n) = Const n
simplify (p :+: q) = (simplify p) :+: (simplify q)
simplify (Const 0 :*: q) = Const 0
simplify (Const 1 :*: q) = simplify q
simplify (Const n :*: q) = (simplify q) :+: (simplify (Const (n-1) :*: q))
simplify (p :*: q) = simplify' (simplify p :*: simplify q)
where
  simplify' (Const n :*: q) = simplify (Const n :*: q)
simplify' p = p

test3c =
  simplify ((X :*: Const 3) :+: (Const 0 :*: X)) == (X :*: Const 3) :+: Const 0
&& simplify (X :*: (Const 3 :+: Const 4)) == X :*: (Const 3 :+: Const 4)
&& (simplify (Const 4 :+: (Const 3 :*: X)) == Const 4 :+: (X :+: (X :+: X)))
  || simplify (Const 4 :+: (Const 3 :*: X)) == Const 4 :+: ((X :+: X) :+: X))
&& simplify (((Const 1 :*: Const 2) :*: (X :+: Const 1)) :*: Const 2) ==
  ((X :+: Const 1) :+: (X :+: Const 1)) :+: Const 2

prop1_simplify :: Expr -> Bool
prop1_simplify p = isSimple (simplify p)

prop2_simplify :: Expr -> Int -> Bool
prop2_simplify p i = eval p i == eval (simplify p) i