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] -> [String]
f [] = []
f (n:ns) = [ if i<j then "<" else ">", | (i,j) <- zip (n:ns) ns, i /= j ]

test1a =
  f [4,2,5,6,1,8] == [">","<","<",">","<"]
  && f [] == []
  && f [3] == []
  && f [3,3,1,-3] == [">",">"]

-- Question 2

-- 2a

isInitialism :: String -> Bool
isInitialism \ s = \text{length } s > 1 \land \text{and [ isUpper } c \mid c \leftarrow s ]

p :: [\text{String}] \rightarrow \text{Int}
p ss = \text{length [ } s \mid s \leftarrow ss, \text{isInitialism } s ]

test2a =
  \text{isInitialism } "A" == \text{False}
  \land \text{isInitialism } "AWOL" == \text{True}
  \land \text{isInitialism } "Ltd" == \text{False}
  \land p ["I","played","the","BBC","DVD","on","my","TV"] == 3
  \land p ["The","DUP","MP","is","not","OK"] == 3
  \land p ["The","SNP","won","in","South","Morningside"] == 1
  \land p [] == 0

-- 2b

isInitialism\' :: \text{String} \rightarrow \text{Bool}
isInitialism\' [] = \text{False}
isInitialism\' [c] = \text{False}
isInitialism\' (c:c':s) = \text{isCaps } (c:c':s)

isCaps :: \text{String} \rightarrow \text{Bool}
isCaps [] = \text{True}
isCaps (c:s) = \text{isUpper } c \land \text{isCaps } s

q :: [\text{String}] \rightarrow \text{Int}
q [] = 0
q (s:ss) | \text{isInitialism\' } s = 1 + q ss
  \text{otherwise} = q ss

test2b =
  \text{isInitialism\' } "A" == \text{False}
  \land \text{isInitialism\' } "AWOL" == \text{True}
  \land \text{isInitialism\' } "Ltd" == \text{False}
  \land q ["I","played","the","BBC","DVD","on","my","TV"] == 3
  \land q ["The","DUP","MP","is","not","OK"] == 3
  \land q ["The","SNP","won","in","South","Morningside"] == 1
  \land q [] == 0

-- 2c

r :: [\text{String}] \rightarrow \text{Int}
r ss = \text{foldr } (\_ \rightarrow \_n \rightarrow n+1) 0 (\text{filter } \text{isInitialism\' } ss)

test2c =
  r ["I","played","the","BBC","DVD","on","my","TV"] == 3
  \land r ["The","DUP","MP","is","not","OK"] == 3
  \land r ["The","SNP","won","in","South","Morningside"] == 1
&& 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 ]
    expr n | 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 8 ]

-- 3a

eval :: Expr -> Int -> Int
eval X i = i
eval (Const n) i = n
eval (p :+: q) i = eval p i + eval q i
eval (p :*: q) i = eval p i * eval q i
test3a =
  eval ((Const 3 :*: X) :+: (X :*: Const 0)) 2 == 6
  && eval ((Const 3 :+: Const 4) :*: X) 2 == 14
  && eval (Const 4 :+: (X :*: Const 3)) 3 == 13
  && eval (Const 2 :*: ((X :+: Const 1) :*: (Const 2 :*: Const 1))) 3 == 16

-- 3b

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

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

-- 3c

simplify :: Expr -> Expr
simplify X = X
simplify (Const n) = Const n
simplify (p :+: q) = (simplify p) :+: (simplify q)

where simplify' (p :+: Const n) = simplify (p :+: Const n)
simplify' p = p

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

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