```haskell
import Data.Char
import Test.QuickCheck
import Control.Monad -- defines liftM, liftM2, used below

-- 1a

f :: [String] -> String
f xs = concat [ x | x <- xs, not (null x), isUpper (head x) ]

test1a =
    f ["This","Is","not","A","non","Test"]       ==  "ThisIsATest"
 && f ["noThing","beGins","uPPER"]               ==  ""
 && f ["Non-words","like","42","get","Dropped"]  ==  "Non-wordsDropped"
 && f ["An","Empty","Word","","gets","dropped"]  ==  "AnEmptyWord"

-- 1b

g :: [String] -> String
g []                                        =  ""
g (x:xs) | not (null x) && isUpper (head x)  =  x ++ g xs
         | otherwise                         =  g xs

test1b =
    g ["This","Is","not","A","non","Test"]       ==  "ThisIsATest"
 && g ["noThing","beGins","uPPER"]               ==  ""
 && g ["Non-words","like","42","get","Dropped"]  ==  "Non-wordsDropped"
 && g ["An","Empty","Word","","gets","dropped"]  ==  "AnEmptyWord"

test1 = test1a && test1b
prop_1 xs = f xs == g xs
check1 = quickCheck prop_1

-- 2a

p :: [(Int,Int)] -> Bool
p zs | not (null zs)
  = and [ v == x | ((u,v),(x,y)) <- zip zs (tail zs) ]

test2a =
    p [(1,2),(2,3),(3,4)]         ==  True
 && p [(9,5),(5,5),(5,7),(7,-2)]  ==  True
 && p [(1,2),(3,4)]               ==  False
 && p [(1,2),(2,3),(33,4)]        ==  False

-- 2b

q :: [(Int,Int)] -> Bool
q [(u,v)]           =  True
q ((u,v):(x,y):zs)  =  v == x && q ((x,y):zs)

test2b =
    q [(1,2),(2,3),(3,4)]         ==  True
 && q [(9,5),(5,5),(5,7),(7,-2)]  ==  True
 && q [(1,2),(3,4)]               ==  False
 && q [(1,2),(2,3),(33,4)]        ==  False

-- 2c

r :: [(Int,Int)] -> Bool
r zs | not (null zs)
  = foldr (&&) True (map (\ ((u,v),(x,y)) -> v == x) (zip zs (tail zs)))
```

```
test2c =
    r [(1,2),(2,3),(3,4)]         ==  True
 && r [(9,5),(5,5),(5,7),(7,-2)]  ==  True
 && r [(1,2),(3,4)]               ==  False
 && r [(1,2),(2,3),(33,4)]        ==  False

test2 = test2a && test2b && test2c
prop_2 xs = not (null xs) ==> p xs == q xs && q xs == r xs
check2 = quickCheck prop_2


-- 3

data Expr = Var String
          | Const Int
          | Expr :+: Expr
          | Expr :*: Expr
          deriving (Eq, Show)

-- code that enables QuickCheck to generate arbitrary values of type Expr

instance Arbitrary Expr where
  arbitrary = sized arb
    where
    arb 0          =  liftM Var arbitrary
    arb n | n > 0  =  oneof [liftM Const arbitrary,
                             liftM Var arbitrary,
                             liftM2 (:+:) sub sub,
                             liftM2 (:*:) sub sub]
      where
      sub = arb (n `div` 2)

-- 3a

isSimple :: Expr -> Bool
isSimple (Const m)              =  True
isSimple (Var x)                =  True
isSimple (Const m :+: Const n)  =  False
isSimple (Const 0 :+: a)        =  False
isSimple (a :+: Const 0)        =  False
isSimple (a :+: b)              =  isSimple a && isSimple b
isSimple (Const m :*: Const n)  =  False
isSimple (Const 0 :*: a)        =  False
isSimple (a :*: Const 0)        =  False
isSimple (Const 1 :*: a)        =  False
isSimple (a :*: Const 1)        =  False
isSimple (a :*: b)              =  isSimple a && isSimple b

test3a =
    isSimple (Var "x" :+: Var "y")                   ==  True
 && isSimple (Const 1 :+: Const 2)                   ==  False
 && isSimple (Const 0 :+: Var "x")                   ==  False
 && isSimple ((Var "y" :+: Const 1) :*: Var "x")     ==  True
 && isSimple (Var "y" :+: (Var "x" :*: Const 0))     ==  False
 && isSimple (Var "x" :*: (Const 3 :+: Const (-2)))  ==  False

-- 3b

simplify :: Expr -> Expr
simplify (Const m)  =  Const m
```

```
simplify (Var x)    =  Var x
simplify (a :+: b)  =  simplify a +++ simplify b
   where
   Const a +++ Const b  =  Const (a + b)
   Const 0 +++ a        =  a
   a +++ Const 0        =  a
   a +++ b              =  a :+: b
simplify (a :*: b)  =  simplify a *** simplify b
   where
   Const a *** Const b  =  Const (a * b)
   Const 0 *** a        =  Const 0
   a *** Const 0        =  Const 0
   Const 1 *** a        =  a
   a *** Const 1        =  a
   a *** b              =  a :*: b

test3b =
    simplify (Var "x" :+: Var "y")
         ==  (Var "x" :+: Var "y")
 && simplify (Const 1 :+: Const 2)
         ==  Const 3
 && simplify (Const 0 :+: Var "x")
         ==  Var "x"
 && simplify ((Var "y" :+: Const 1) :*: Var "x")
         ==  ((Var "y" :+: Const 1) :*: Var "x")
 && simplify (Var "y" :+: (Var "x" :*: Const 0))
         ==  Var "y"
 && simplify (Var "x" :*: (Const 3 :+: Const (-2)))
         ==  Var "x"

test3 = test3a && test3b
prop_3 a  =  isSimple (simplify a) && simplify (simplify a) == simplify a
check3 = quickCheck prop_3

test = test1 && test2 && test3
check = check1 >> check2 >> check3
```