-- Problem 1
-- :
pgm = do
    putStrLn "Enter a list of ints > 
    line <- getLine
    let list = read line :: [Int]
    putStrLnln ("Read in, reversed: " ++ show (reverse list))
    putStrLnn $ "Has length " ++ show (length list) -- one solution
--putStrLnln ("Has length " ++ show (length list)) -- another solution
-- :

-- Problem 2
-- Treat itr n f = (((id . f) . f)... . f)
itr n f = foldl (. ) id [f | i <- [1..n]]

-- Problem 3 --- a couple of sample solutions
dropPrefix x y
    = let n = length x in if x == take n y then Just (drop n y) else Nothing

dropPrefix2 x y
    | x == take n y = Just (drop n y)
    | otherwise = Nothing
    where n = length x

----------------------------------
-- List problems (#'s 4 - 6)
data List a = None | Node a (List a) deriving (Show)

-- Problem 4
listEq :: Eq a => List a -> List a -> Bool
listEq None None = True
listEq (Node h1 t1) (Node h2 t2) = h1 == h2 && listEq t1 t2
listEq ___ = False

-- Problem 5
listLE :: Ord a => List a -> List a -> Bool
listLE None None = True
listLE None (Node ___) = True
listLE (Node ___) None = False
listLE (Node h1 t1) (Node h2 t2) = h1 < h2 || (h1 == h2 && listLE t1 t2)

-- Problem 6
instance (Eq a) => Eq (List a) where
    (==) = listEq

instance (Ord a) => Ord (List a) where
    (<=) = listLE

----------------------------------
-- Background for Problem 7
-- Regular expressions with only concatenation and alternation
data Pattern a
    = P_const a
    | P_or [Pattern a]
    | P_and [Pattern a] deriving (Eq, Read, Show)

match :: Eq a => (Pattern a) -> [a] -> Maybe [a]
match (P_const _) [] = Nothing
match (P_const x) (h:t) | h == x = Just t
    | otherwise = Nothing
match (P_or []) vals = Nothing
match (P_or (p:ps)) vals
= case match p vals of
   Nothing -> match (P_or ps) vals
   ok @ (Just leftover) -> ok

-- Problem 7: Add P_and clause

match (P_and []) vals = Just vals -- Code for
match (P_and (p:ps)) vals
   = case match p vals of
      Nothing -> Nothing
      ok @ (Just leftover) -> match (P_and ps) leftover