```haskell
import System.IO
import System.Random

----------------------------
----------------------------

data LE a
    = Var a
    | App (LE a) (LE a)
    | Abs a (LE a)
    deriving (Show,Eq)

subst :: Eq a => LE a -> a-> LE a -> LE a
subst what var wher = mks [] what var wher

mks ::  Eq a => [a] -> LE a -> a-> LE a -> LE a
mks bound what var (Var a) =
    if a `elem` bound then Var a else what
mks bound what var (App e a) =
    App (mks bound what var e) (mks bound what var a)
mks bound what var (Abs v e) =
    Abs v (mks (v:bound) what var e)

----------------------------
----------------------------

length' a [] = a                            -- 1
length' a (_:xs) = length' (a+1) xs         -- 2

foldl' f a [] = a                           -- 3
foldl' f a (x:xs) = foldl' f (f a x) xs     -- 4

{-

length 0 xs = foldl (\ a _ -> a+1) 0 xs

1)

xs = []

L = length 0 [] =|1
  = 0
P = foldl (\ a -> a+1) 0 [] =|3
  = 0

L = P


2)

I.P.
forall k in N: length k as = foldl (\ a _ -> a+1) k as

xs = (a:as)

L = length 0 (a:as)  =|2
  = length (0+1) as  =|soucet
  = length (1) as    =|prebytecne zavorky
  = length 1 as      =|I.P.
  = foldl (\ a _ -> a+1) 1 as

P = foldl (\ a _ -> a+1) 0 (a:as)                 =|4
  = foldl (\ a _ -> a+1) ((\ a _ -> a+1) 0 a) as  =|beta_redukce
  = foldl (\ a _ -> a+1) ((\ _ -> 0+1) a) as      =|beta_redukce
  = foldl (\ a _ -> a+1) (0+1) as                 =|soucet
  = foldl (\ a _ -> a+1) (1) as                   =|prebytecne zavorky
  = foldl (\ a _ -> a+1) 1 as

L = P

Q.E.D.

-}

----------------------------
```

```
----------------------------

primes = 2:[x | x <- [3,5..], isPrime x primes]
    where
        isPrime x (p:ps) =
            (p*p > x) || (x `mod` p /= 0 && isPrime x ps)


-----------------------------
-----------------------------

mkR :: String -> IO ()
mkR file = do
    h <- openFile file ReadMode
    c <- hGetContents h
    let alllogs = lines c
    let empty = length $ filter (=="") alllogs
    let wempty = filter (/="") alllogs
    let notLogs = length $ filter (not . isLogin) wempty
    let correct = filter isLogin wempty
    putStrLn $ show (length correct) ++ "/" ++ show notLogs ++ "/" ++ show empty
    randLog <- genRand correct
    putStrLn $ unlines randLog
    hClose h

isLogin :: String -> Bool
isLogin x =
    length x == 8 && head x == 'x' &&
    (all (\x -> elem x ['a'..'z']) $ take 5 $ tail x) &&
    (all (\x -> elem x (['0'..'9']++['a'..'z'])) $ drop 6 x)

genRand :: [a] -> IO [a]
genRand [] = return []
genRand l = do
    ir <- randomRIO (0,length l - 1) :: IO Int
    let (h,t) = splitAt ir l
    let v = head t
    let r = tail t
    mkr <- genRand (h++r)
    return (v:mkr)

-- EOF
```