```haskell
import System.IO

-- ###################################################################
{-

LET True = \ x y . x y
Definujte IMPlikaci
Redukujte IMP True False = False
-----
LET False = \ x y . y
LET IMP = \ x y . x (\ z . y) True

IMP True False =
  (\ x y . x (\ z . y) True) True False =
  (\ y . True (\ z . y) True) False =
  True (\ z . False) True =
  (\ x y . x y) (\ z . False) True =
  (\ y . (\ z . False) y) True =
  (\ z . False) True =
  False

-}
-- ###################################################################

data Days
  = Su | Mo | Tu | We | Th | Fr | Sa
  deriving (Show,Eq,Read,Ord,Bounded,Enum)

weeks = [Su .. Sa] ++ weeks
months = [31,29,31,30,31,30,31,31,30,31,30,31]

mkDays ds = concat [[1..n] | n<-ds]
mkMonths ds = concat $ zipWith (\l n -> map (const n) l) [[1..n] | n<-ds] [1..12]

calendar2016 = zip3 ([Fr,Sa]++weeks) (mkDays months) (mkMonths months)

p13 [] = []
p13 (d@(Fr,13,_):xs) = d : p13 xs
p13 (_:xs) = p13 xs


-- ###################################################################

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

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

_foldl _ a [] = a                         -- 3
_foldl f a (x:xs) = _foldl f (f a x) xs   -- 4

{-

1: xs == []

L = _length 0 [] =|1
  = 0
P = _foldl (\a _ -> a+1) 0 [] =|3
  = 0
L = P

2: xs == (a:as)

I.H.
For all k:
_length k as = _foldl (\a _ -> a+1) k as

L = _length 0 (a:as)                =|2
  = _length (0+1) as                =|I.H.
  = _foldl (\a _ -> a+1) (0+1) as

P = _foldl (\a _ -> a+1) 0 (a:as)    =|4
```

```
  = _foldl (\a _ -> a+1) ((\a _ -> a+1) 0 a) as   =|BetaRed
  = _foldl (\a _ -> a+1) ((\_ -> 0+1) a) as       =|BetaRed
  = _foldl (\a _ -> a+1) (0+1) as
L = P

Q.E.D.

-}

prAno :: FilePath -> IO()
prAno f =  do
  h <- openFile f ReadMode
  c <- hGetContents h
  putStr $ unlines $ mkAll $ lines c
  hClose h

mkAll :: [String] -> [String]
mkAll lns =
  annoLn (getMinMax lns) lns

getMinMax :: [String] -> (Int,Int)
getMinMax [] = (0,0)
getMinMax (l:ls) = let
  (mi,ma) = getMinMax ls
  ml = length l
  in if ml<mi then (ml,ma)
  else if ml>ma then (mi,ml)
  else (mi,ma)

annoLn :: (Int,Int) -> [String] -> [String]
annoLn _ [] = []
annoLn m@(mi,ma) (l:ls) = let
  di = length l - mi
  da = ma - length l
  in ((show di)++"#"++(show da)++":"++l) : annoLn m ls

-- EOF
```