```
import System.IO

{-

LET True  = \ x y . x
LET False = \ x y . y
LET (?:)  = \ c t f . c t f

 Y E = k
 E k = k
 Y E = E k = E (Y E)

LET minus = Y (\ f x y . iszero x ? 0 : (iszero y ? x : f (prev x) (prev y) ))

-}

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

all' [] = True              -- 1
all' (x:xs) = x && all' xs  -- 2

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

{-

Ukazat:
foldr (&&) True xs = all xs

1)
xs = []

L = foldr (&&) True [] =|3
  = True

P = all [] =|1
  = True

L = P

2)
I.H.: foldr (&&) True as = all as
xs = (a:as)

L = foldr (&&) True (a:as) =|4
  = (&&) a (foldr (&&) True as) =|IH
  = (&&) a (all as) =|prefix->infix
  = a && (all as) =|priorita
  = a && all as

P = all (a:as) =|2
  = a && all as

L = P

Q.E.D.

-}


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

ins x [] = [x]
ins x l@(y:ys) =
    if x<y then x : l else y : ins x ys

sort l = foldr ins [] l

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

data DLog
    = Empty
```

```
    | IT Integer String
    deriving (Show,Eq)

pr :: [DLog] -> IO ()
pr [] = return ()
pr (Empty:xs) = pr xs
pr ((IT i s):xs) =
    if i `mod` 5 == 0
    then putStrLn (show i ++ ":" ++ s) >> pr xs
    else pr xs

pl :: String -> IO ()
pl fn = do
    h <- openFile fn ReadMode
    c <- hGetContents h
    pr $ proc $ lines c
    hClose h

proc :: [String] -> [DLog]
proc [] = []
proc (l:ls) =
    if null l
    then Empty : proc ls
    else (mk l) : proc ls

mk :: String -> DLog
mk l = IT ((read (takeWhile (/='#') l))::Integer)
          (tail $ dropWhile (/='#') l)

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

pl' :: String -> IO()
pl' fn = do
    h <- openFile fn ReadMode
    c <- hGetContents h
    pr $ map mkDL $ lines c
    hClose h

mkDL :: String -> DLog
mkDL [] = Empty
mkDL l = let (n,s) = span (/='#') l
         in IT ((read n)::Integer) (tail s)

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

data Tree a
    = Nd a (Tree a) (Tree a)
    | Lf
    deriving (Show,Eq)

takeLev :: Int -> Tree a -> Tree a
takeLev 0 _  = Lf
takeLev _ Lf = Lf
takeLev n (Nd v l r) =
    Nd v (takeLev (n-1) l) (takeLev (n-1) r)

initTree :: a -> Tree a
initTree val =
    Nd val (initTree val) (initTree val)

-- EOF
```