```
import System.IO

-- 1

{-

LET True  = \ x y. x y

LET False = \ x y. y

LET NOR = \ a b. a (\i.False) (b (\k.False) True)

NOR True False =
(\ a b. a (\i.False) (b (\k.False) True)) True False ->Beta
(\ b. True (\i.False) (b (\k.False) True)) False ->Beta
True (\i.False) (False (\k.False) True) =
(\ x y. x y) (\i.False) (False (\k.False) True) ->Beta
(\ y. (\i.False) y) (False (\k.False) True) ->Beta
(\i.False) (False (\k.False) True) ->Beta
False

-}

-- 2

data ITree
  = INd Integer ITree ITree
  | ILf Integer
  deriving (Show,Eq)

mkFibTree :: Integer -> ITree
mkFibTree n
  | n==0 = ILf 0
  | n==1 = ILf 1
  | n >1 = INd (getVal im1 + getVal im2) im1 im2
  where
    im1 = mkFibTree (n-1)
    im2 = mkFibTree (n-2)

getVal :: ITree -> Integer
getVal (INd v _ _) = v
getVal (ILf v) = v


-- 3

{--

Def:
negAll [] = []
negAll (x:xs) = not x : negAll xs

Prove:
negAll xs = map not xs
for all finite xs

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

negAll [] = []                        -- 1
negAll (x:xs) = not x : negAll xs     -- 2

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



1)
xs==[]
negAll [] = map not []

L = negAll [] =|1
  = []
P = map f []  =|3
  = []
L = P

2)
xs=(a:as)
I.H. : negAll as = map not as

To prove:
negAll (a:as) = map not (a:as)

L = negAll (a:as)       =|2
  = not a : negAll as   =|I.H.
  = not a : map not as  =|4(<-)
  = map not (a:as)
  = P
```

```
Q.E.D.

--}


-- 4

data Tokens
  = IntVal Integer
  | Plus
  | Minus
  deriving (Show,Eq)


llex :: String -> [Tokens]
llex [] = []
llex (c:cs)
  | c=='+' = Plus : llex cs
  | c=='-' = Minus : llex cs
  | elm c ['0'..'9'] = getIVal [c] cs
  | elm c " \t" = llex cs
  | True = error "Incorrect symbol on input"

getIVal :: String -> String -> [Tokens]
getIVal is l@(c:cs) =
  if elm c ['0'..'9'] then getIVal (app is [c]) cs
     else IntVal ((read is)::Integer) : llex l
getIVal is [] = [IntVal ((read is)::Integer)]

elm :: Eq a => a -> [a] -> Bool
elm _ [] = False
elm x (y:ys) = x==y || elm x ys

app :: [a] -> [a] -> [a]
app [] ys = ys
app (x:xs) ys = x : app xs ys


-- 5

-- "Core" alternativa
getMinMaxL [l] = (length l,length l)
getMinMaxL (l:ls) =
  if ll<mi then (ll,ma)
  else if ll>ma then (mi,ll)
  else r
  where
    r@(mi,ma) = getMinMaxL ls
    ll = length l

-- "Prelude light" alternativa
getMinMaxL' ls = (mi,ma)
  where
    lls = map length ls
    mi = foldl1 min lls
    ma = foldl1 max lls

-- "Prelude" alternativa
getMinMaxL'' ls = (minimum lls,maximum lls)
  where
    lls = map length ls


prAno fname = do
  h <- openFile fname ReadMode
  c <- hGetContents h
  if null c then putStrLn "Empty file"
            else putStr $ unlines $ procc $ lines c
  hClose h

procc ls = map mkline ls
  where
    (mi,ma) = getMinMaxL' ls
    mkline l = show (llen-mi)++","++show (ma-llen)++":"++l
      where
        llen = length l


-- EOF
```