```haskell
module Eval2 (eval) where

import AST
import Control.Applicative (Applicative(..))
import Control.Monad       (liftM, ap)

-- Estados
type Env = [(Variable,Int)]

-- Estado nulo
initState :: Env
initState = []

-- Mónada estado
newtype StateError a = StateError { runStateError :: Env -> Maybe (a, Env) }

instance Monad StateError where
        return x          = StateError (\e -> Just (x,e))
        StateError h >>= f = StateError (\e -> case (h e) of
                                                 Nothing     -> Nothing
                                                 Just (a,e') -> runStateError (f a ↵
                                                 ) e' )
        -- h :: Env -> Maybe (a,Env)
        -- f :: a -> StateError b


-- Clase para representar mónadas con estado de variables
class Monad m => MonadState m where
    -- Busca el valor de una variable
    lookfor :: Variable -> m Int
    -- Cambia el valor de una variable
    update :: Variable -> Int -> m ()

instance MonadState StateError where
    lookfor v = StateError (\s -> Just (lookfor' v s, s))
                where lookfor' v ((u, j):ss) | v == u = j
                                             | v /= u = lookfor' v ss
        -- Suponemos que no se utilizan variables no declaradas en LIS
    update v i = StateError (\s -> Just ((), update' v i s))
                 where update' v i [] = [(v, i)]
                       update' v i ((u, _):ss) | v == u = (v, i):ss
                       update' v i ((u, j):ss) | v /= u = (u, j):(update' v i ss)

-- Para calmar al GHC
instance Functor StateError where
    fmap = liftM

instance Applicative StateError where
    pure   = return
    (<*>)  = ap


-- Clase para representar mónadas que lanzan errores
class Monad m => MonadError m where
    -- Lanza un error
    throw :: m a

instance MonadError StateError where
        throw = StateError (\e -> Nothing)

-- Evalua un programa en el estado nulo
-- CAMBIAMOS EL TIPO DE RETORNO RESPECTO AL Eval1, PARA PODER RETORNAR ERRORES.
eval :: Comm -> Maybe Env
eval p =  case (runStateError (evalComm p) initState) of
                Nothing    -> Nothing
                Just (a,e) -> Just e

-- Evalua un comando en un estado dado
```

```haskell
evalComm :: (MonadState m, MonadError m) => Comm -> m ()
evalComm Skip          = return ()
evalComm (Let v n)     = do n' <- evalIntExp n
                            update v n'
                            return ()

evalComm (Seq c1 c2)   = do evalComm c1
                            evalComm c2
                            return ()

evalComm (Cond b ct cf) = do  cond <- evalBoolExp b
                              if cond then (do {evalComm ct ; return () } )
                                      else (do {evalComm cf ; return () } )

evalComm w@(While b c)  = do  cond <- evalBoolExp b
                              if cond then evalComm (Seq c w)
                                      else return ()









-- Evalua una expresion entera, sin efectos laterales
evalIntExp :: (MonadState m, MonadError m) => IntExp -> m Int
evalIntExp (Const x)      = return x
evalIntExp (Var v)        = lookfor v
evalIntExp (UMinus e)     = do n <- evalIntExp e
                               return (-n)
evalIntExp (Plus e1 e2)   = do n1 <- evalIntExp e1
                               n2 <- evalIntExp e2
                               return (n1+n2)

evalIntExp (Minus e1 e2)  = do n1 <- evalIntExp e1
                               n2 <- evalIntExp e2
                               return (n1-n2)

evalIntExp (Times e1 e2)  = do n1 <- evalIntExp e1
                               n2 <- evalIntExp e2
                               return (n1*n2)

evalIntExp (Div e1 e2)    = do n1 <- evalIntExp e1
                               n2 <- evalIntExp e2
                               if n2==0 then throw else return (div n1 n2)

-- Evalua una expresion entera, sin efectos laterales
evalBoolExp :: (MonadState m, MonadError m) => BoolExp -> m Bool
evalBoolExp BTrue       = return True
evalBoolExp BFalse      = return False
evalBoolExp (Eq e1 e2)  = do n1 <- evalIntExp e1
                             n2 <- evalIntExp e2
                             return (n1==n2)
evalBoolExp (Lt e1 e2)  = do n1 <- evalIntExp e1
                             n2 <- evalIntExp e2
                             return (n1<n2)
evalBoolExp (Gt e1 e2)  = do n1 <- evalIntExp e1
                             n2 <- evalIntExp e2
                             return (n1>n2)
evalBoolExp (And b1 b2) = do b1' <- evalBoolExp b1
                             b2' <- evalBoolExp b2
                             return (b1' && b2')
evalBoolExp (Or b1 b2 ) = do b1' <- evalBoolExp b1
                             b2' <- evalBoolExp b2
                             return (b1' || b2')
evalBoolExp (Not b)     = do b' <- evalBoolExp b
                             return (not b')

```