```haskell
module Eval3 (eval) where

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

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

newtype StateErrorTick a = StateErrorTick { runStateErrorTick :: Env -> (Maybe (a, Env),Int) }
--Agregamos un Int para contar la cantidad de operaciones. Si se produce un error
--contamos las operaciones hasta el error.

instance Monad StateErrorTick where
        return x              = StateErrorTick (\e -> (Just (x,e),0) )
        StateErrorTick h >>= f = StateErrorTick (\e -> case h e of
                                                        (Nothing,n)     -> (Nothing,n)
                                                        (Just (a,e'),n) -> let (ans,n') = runStateErrorTick (f a) e'
                                                                           in  (ans ,n' + n) )
        -- h :: Env -> (Maybe (a, Env),Int)
        -- f :: a -> StateErrorTick b


class Monad m => MonadTick m where
        tick :: m ()

instance MonadTick StateErrorTick where
        tick = StateErrorTick (\e -> (Just ( () , e) , 1))


class Monad m => MonadState m where
    lookfor :: Variable -> m Int
    update :: Variable -> Int -> m ()


instance MonadState StateErrorTick where
    lookfor v = StateErrorTick (\s -> (Just (lookfor' v s, s),0) )
                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 = StateErrorTick (\s -> (Just ((), update' v i s),0))
                 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)

class Monad m => MonadError m where
    throw :: m a

instance MonadError StateErrorTick where
        throw = StateErrorTick (\e -> (Nothing,0))


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

-- Evalua un programa en el estado nulo
eval :: Comm ->  (Maybe Env,Int)
eval p =  case (runStateErrorTick (evalComm p) initState) of
                (Nothing,n)    -> (Nothing,n)
                (Just (_,e),n) -> (Just e,n)

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

```haskell


evalComm :: (MonadState m, MonadError m, MonadTick 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 ()



evalIntExp :: (MonadState m, MonadError m, MonadTick m) =>  IntExp -> m Int
evalIntExp (Const x)      = return x
evalIntExp (Var v)        = lookfor v
evalIntExp (UMinus e)     = do n <- evalIntExp e
                               tick
                               return (-n)
evalIntExp (Plus e1 e2)   = do n1 <- evalIntExp e1
                               n2 <- evalIntExp e2
                               tick
                               return (n1+n2)

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

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

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

-- Evalua una expresion entera, sin efectos laterales
evalBoolExp :: (MonadState m, MonadError m, MonadTick 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')
```

```haskell
evalBoolExp (Not b)     = do b' <- evalBoolExp b
                             return (not b')
```