```haskell
module Eval3 (eval) where

import AST

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

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

-- Ejercicio 3.a
-- Monada de Estado, Errores y Cantidad de Operaciones
-- Se puede haber hecho de otra forma, pero elegí si hay un error no mostrar la cuenta de operaciones, sí el entorno final.
newtype StateErrorTick a = StateErrorTick {runStateErrorTick :: Env -> (Maybe (a,Int),Env)}

-- 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 ()

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

-- Ejercicio 3.b
-- Clase para representar monadas que acumulan valores
class Monad m => MonadTick m where
    -- tick
    tick :: m ()

-- Ejercicio 3.d
-- instancia de MonadError para StateErrorTick
instance MonadError StateErrorTick where
    throw = StateErrorTick (\s -> (Nothing, s))

instance Monad StateErrorTick where
      return x = StateErrorTick (\s -> (Just (x,0),s))
      m >>= f  = StateErrorTick (\s -> case runStateErrorTick m s of
                                    (Nothing, s') -> (Nothing, s')
                                    (Just (v,n), s')  -> case runStateErrorTick (f v) s' of
                                                            (Nothing, s'') -> (Nothing, s'')
                                                            (Just (v',n'),s'') -> (Just (v',n
+n'),s'') )
-- Ejercicio 3.e
-- instancia de MonadState para StateErrorTick
instance MonadState StateErrorTick where
      lookfor v = StateErrorTick (\s -> (Just (lookfor' v s,0), s))
                  where lookfor' v ((u, j):ss) | v == u = j
                                               | v /= u = lookfor' v ss
      update v i = StateErrorTick (\s -> (Just ((),0), 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)

-- Ejercicio 3.c
-- instancia de MonadTick para StateErrorTick
instance MonadTick StateErrorTick where
      tick = StateErrorTick (\s -> (Just ((),1),s))

-- Evalua un programa en el estado nulo
eval :: Comm -> (Maybe ((),Int), Env)
eval c = runStateErrorTick (evalComm c) initState

-- Evalua un comando en un estado dado
evalComm :: (MonadState m, MonadError m, MonadTick m) => Comm -> m ()
evalComm Skip           = return ()
evalComm (Let v i)      = do ei <- evalIntExp i
```

```
                              update v ei
evalComm (Seq c1 c2)    = do evalComm c1
                              evalComm c2
evalComm (Cond b c1 c2) = do eb <- evalBoolExp b
                              if eb then evalComm c1
                                    else evalComm c2
evalComm (While b c)    = do eb <- evalBoolExp b
                              if eb then evalComm (Seq c (While b c))
                                    else evalComm Skip

-- Evalua una expresion entera, sin efectos laterales
evalIntExp :: (MonadState m, MonadError m, MonadTick m) => IntExp -> m Int
evalIntExp (Const i)   = return i
evalIntExp (Var v)     = lookfor v
evalIntExp (UMinus i)  = do u <- evalIntExp i
                            tick
                            return (-u)
evalIntExp (Plus n m)  = do arg1 <- evalIntExp n
                            arg2 <- evalIntExp m
                            tick
                            return (arg1 + arg2)
evalIntExp (Minus n m) = do arg1 <- evalIntExp n
                            arg2 <- evalIntExp m
                            tick
                            return (arg1 - arg2)
evalIntExp (Times n m) = do arg1 <- evalIntExp n
                            arg2 <- evalIntExp m
                            tick
                            return (arg1 * arg2)
evalIntExp (Div n m)   = do arg1 <- evalIntExp n
                            arg2 <- evalIntExp m
                            tick
                            if arg2 == 0 then throw else return (arg1 `div` arg2)


-- 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 n m)  = do arg1 <- evalIntExp n
                           arg2 <- evalIntExp m
                           return (arg1 == arg2)
evalBoolExp (Lt n m)  = do arg1 <- evalIntExp n
                           arg2 <- evalIntExp m
                           return (arg1 < arg2)
evalBoolExp (Gt n m)  = do arg1 <- evalIntExp n
                           arg2 <- evalIntExp m
                           return (arg1 > arg2)
evalBoolExp (And p q) = do arg1 <- evalBoolExp p
                           arg2 <- evalBoolExp q
                           return (arg1 && arg2)
evalBoolExp (Or p q)  = do arg1 <- evalBoolExp p
                           arg2 <- evalBoolExp q
                           return (arg1 || arg2)
evalBoolExp (Not p)   = do ep <- evalBoolExp p
                           return (not ep)
```