```haskell
module Eval1 (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 State a = State { runState :: Env -> (a, Env) }

instance Monad State where
    return x = State (\s -> (x, s))
    m >>= f = State (\s -> let (v, s') = runState m s in
                           runState (f v) s')

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

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

-- 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 State where
    lookfor v = State (\s -> (lookfor' v s, s))
                where lookfor' v ((u, j):ss) | v == u = j
                                             | v /= u = lookfor' v ss
    update v i = State (\s -> ((), 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)

-- Evalua un programa en el estado nulo
eval :: Comm -> Env
eval p = snd (runState (evalComm p) initState)

-- Evalua un comando en un estado dado
evalComm :: MonadState 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 => 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
                               return (div n1 n2)

-- Evalua una expresion entera, sin efectos laterales
evalBoolExp :: MonadState 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')

```