```haskell
module Eval1 (eval) where

import AST

-- 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')

-- 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 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 => IntExp -> m Int
evalIntExp (Const i)   = return i
evalIntExp (Var v)     = lookfor v
evalIntExp (UMinus i)  = do u <- evalIntExp i
                            return (-u)
evalIntExp (Plus n m)  = do arg1 <- evalIntExp n
                            arg2 <- evalIntExp m
                            return (arg1 + arg2)
evalIntExp (Minus n m) = do arg1 <- evalIntExp n
                            arg2 <- evalIntExp m
                            return (arg1 - arg2)
evalIntExp (Times n m) = do arg1 <- evalIntExp n
                            arg2 <- evalIntExp m
                            return (arg1 * arg2)
evalIntExp (Div n m)   = do arg1 <- evalIntExp n
                            arg2 <- evalIntExp m
                            return (arg1 `div` arg2)

-- Evalua una expresion entera, sin efectos laterales
```

```
evalBoolExp :: MonadState 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)
```