Eval3.hs

```
module Eval3 (eval) where

import AST

-- Estados
type State = [(Variable,Int)]
--Contador de operaciones: cantidad de sumas, cantidad de restas,
cantidad de multiplicaciones, cantidad de divisiones
type Op = (Int,Int,Int,Int)


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


--Contador de operaciones nulo
initOp :: Op
initOp = (0,0,0,0)


-- Busca el valor de una variable en un estado, asumo que aparece
exactamente una vez en el estado
lookfor :: Variable -> State -> Int
lookfor x [(_,i)] = i
lookfor x ((y,i):xs) = if x==y then i else lookfor x xs


-- Cambia el valor de una variable en un estado
update :: Variable -> Int -> State -> State
update x i [] = [(x,i)]
update x i ((y,j):xs) = if x==y then ((x,i):xs) else (y,j):(update x i
xs)


-- Evalua un programa en el estado y el contador de operaciones nulo
eval :: Comm -> Maybe (State,Op)
eval p = evalComm p (Just (initState,initOp) )


-- Evalua un comando en un estado dado
evalComm :: Comm -> Maybe (State,Op) -> Maybe (State, Op)
evalComm _ Nothing = Nothing
evalComm Skip s = s
evalComm (Let x ie) t@(Just (s,o)) = case (evalIntExp ie t) of
                        Nothing -> Nothing
                        Just (i,o1) -> Just ((update x i s),o1)

evalComm (Seq c1 c2) t = evalComm c2 (evalComm c1 t)

evalComm (Cond be c1 c2) t@(Just(s,op)) = case (evalBoolExp be t) of
```

```
                                Just (True,o1) -> evalComm c1 (Just
(s,o1))
                                Just (False,o2) -> evalComm c2 (Just
(s,o2))
                                Nothing -> Nothing
evalComm w@(While be c) t@(Just (s,o)) = case (evalBoolExp be t) of
                                Just (True,o1) -> evalComm (Seq c w)
(Just(s,o1))
                                Just (False,o2) -> (Just (s,o2))
                                Nothing -> Nothing


-- Evalua una expresion entera, sin efectos laterales
evalIntExp :: IntExp -> Maybe (State, Op) -> Maybe (Int,Op)
evalIntExp _ Nothing = Nothing
evalIntExp (Const i) (Just (s,op)) = Just (i,op)
evalIntExp (Var x) (Just (s,op)) = Just ((lookfor x s),op)
evalIntExp (UMinus ie) t = aplicU (evalIntExp ie t) ((-1)*)
evalIntExp (Plus ie1 ie2) (Just (s,op)) = let t = Just (s,initOp)
                                          in aplicBin (evalIntExp ie1 t)
(evalIntExp ie2 t) (+) (sumar op (1,0,0,0))
evalIntExp (Minus ie1 ie2) (Just (s,op)) = let t = Just (s,initOp)
                                           in aplicBin (evalIntExp ie1 t)
(evalIntExp ie2 t) (-) (sumar op (0,1,0,0))
evalIntExp (Times ie1 ie2) (Just (s,op)) = let t = Just (s,initOp)
                                           in aplicBin (evalIntExp ie1 t)
(evalIntExp ie2 t) (*) (sumar op (0,0,1,0))
evalIntExp (Div ie1 ie2) t@(Just (s,op)) = let t1 = Just (s,initOp)
                                           in aplicDiv (evalIntExp ie1 t)
(evalIntExp ie2 t1)


-- Evalua una expresion booleana, sin efectos laterales
evalBoolExp :: BoolExp -> Maybe (State,Op) -> Maybe (Bool,Op)
evalBoolExp _ Nothing = Nothing
evalBoolExp BTrue (Just (s,op)) = Just (True,op)
evalBoolExp BFalse (Just (s,op)) = Just (False,op)
evalBoolExp (Eq ie1 ie2) (Just (s,op)) = let t = Just(s,initOp)
                                         in aplicBin (evalIntExp ie1 t)
(evalIntExp ie2 t) (==) op
evalBoolExp (Lt ie1 ie2) (Just (s,op)) = let t = Just(s,initOp)
                                         in aplicBin (evalIntExp ie1 t)
(evalIntExp ie2 t) (<) op
evalBoolExp (Gt ie1 ie2) (Just (s,op)) = let t = Just(s,initOp)
                                         in aplicBin (evalIntExp ie1 t)
(evalIntExp ie2 t) (>) op
evalBoolExp (And be1 be2) (Just (s,op))= let t =  Just(s,initOp)
                                         in aplicBin (evalBoolExp be1 t)
(evalBoolExp be2 t) (&&) op
evalBoolExp (Or be1 be2) (Just (s,op))= let t =  Just(s,initOp)
                                        in aplicBin (evalBoolExp be1 t)
(evalBoolExp be2 t) (||) op
evalBoolExp (Not be) t= aplicU (evalBoolExp be t) (not)
```

```
aplicU :: Maybe (a,Op) -> (a -> a) -> Maybe (a,Op)
aplicU Nothing _ = Nothing
aplicU (Just (a,op)) f = Just ((f a),op)


aplicBin :: Maybe (a,Op) -> Maybe (a,Op) -> (a -> a -> b) -> Op->Maybe
(b,Op)
aplicBin Nothing _ _ _= Nothing
aplicBin _ Nothing _ _= Nothing
aplicBin (Just (a,op1)) (Just (b,op2)) f op= Just ((f a b),sumar (sumar
op1 op2) op)


aplicDiv :: Maybe (Int,Op) -> Maybe (Int,Op) -> Maybe (Int,Op)
aplicDiv Nothing _  = Nothing
aplicDiv _ Nothing  = Nothing
aplicDiv _ (Just (0,o))  = Nothing
aplicDiv (Just (a,op1)) (Just (b,op2)) = Just ((div a b), sumar (sumar
op1 op2) (0,0,0,1))

sumar :: Op -> Op -> Op
sumar (a,b,c,d) (a1,b1,c1,d1) = (a+a1,b+b1,c+c1,d+d1)
```