

```

1  module Eval1 (eval) where
2
3  import AST
4  import Control.Applicative (Applicative(..))
5  import Control.Monad      (liftM, ap)
6
7  -- Estados
8  type Env = [(Variable,Int)]
9
10 -- Estado nulo
11 initState :: Env
12 initState = []
13
14 -- M nada estado
15 newtype State a = State { runState :: Env -> (a, Env) }
16
17 instance Monad State where
18     return x = State (\s -> (x, s))
19     m >= f = State (\s -> let (v, s') = runState m s in
20                             runState (f v) s')
21
22 -- Para calmar al GHC
23 instance Functor State where
24     fmap = liftM
25
26 instance Applicative State where
27     pure    = return
28     (<*>) = ap
29
30 -- Clase para representar m nadas con estado de variables
31 class Monad m => MonadState m where
32     -- Busca el valor de una variable
33     lookfor :: Variable -> m Int
34     -- Cambia el valor de una variable
35     update  :: Variable -> Int -> m ()
36
37 instance MonadState State where
38     lookfor v = State (\s -> (lookfor' v s, s))
39         where lookfor' v ((u, j):ss) | v == u = j
40                                     | v /= u = lookfor' v ss
41     update v i = State (\s -> ((), update' v i s))
42         where update' v i [] = [(v, i)]
43               update' v i ((u, _):ss) | v == u = (v, i):ss
44               update' v i ((u, j):ss) | v /= u = (u, j):(update' v i ss)
45
46 -- Evalua un programa en el estado nulo
47 eval :: Comm -> Env
48 eval p = snd (runState (evalComm p) initState)
49
50 -- Evalua un comando en un estado dado
51 evalComm :: MonadState m => Comm -> m ()
52 evalComm Skip      = return ()
53 evalComm (Let v n)  = do n' <- evalIntExp n
54                          update v n'
55                          return ()
56
57 evalComm (Seq c1 c2) = do evalComm c1
58                          evalComm c2
59                          return ()
60
61 evalComm (Cond b ct cf) = do cond <- evalBoolExp b
62                             if cond then (do {evalComm ct ; return ()} )
63                             else (do {evalComm cf ; return ()} )
64
65 evalComm w@(While b c) = do cond <- evalBoolExp b
66                             if cond then evalComm (Seq c w)
67                             else return ()
68

```

```

69
70 -- Evalua una expresion entera, sin efectos laterales
71 evalIntExp :: MonadState m => IntExp -> m Int
72 evalIntExp (Const x)      = return x
73 evalIntExp (Var v)        = lookfor v
74 evalIntExp (UMinus e)     = do n <- evalIntExp e
75                             return (-n)
76 evalIntExp (Plus e1 e2)   = do n1 <- evalIntExp e1
77                             n2 <- evalIntExp e2
78                             return (n1+n2)
79
80 evalIntExp (Minus e1 e2)  = do n1 <- evalIntExp e1
81                             n2 <- evalIntExp e2
82                             return (n1-n2)
83
84 evalIntExp (Times e1 e2)  = do n1 <- evalIntExp e1
85                             n2 <- evalIntExp e2
86                             return (n1*n2)
87
88 evalIntExp (Div e1 e2)    = do n1 <- evalIntExp e1
89                             n2 <- evalIntExp e2
90                             return (div n1 n2)
91
92 -- Evalua una expresion entera, sin efectos laterales
93 evalBoolExp :: MonadState m => BoolExp -> m Bool
94 evalBoolExp BTrue        = return True
95 evalBoolExp BFalse       = return False
96 evalBoolExp (Eq e1 e2)   = do n1 <- evalIntExp e1
97                             n2 <- evalIntExp e2
98                             return (n1==n2)
99 evalBoolExp (Lt e1 e2)    = do n1 <- evalIntExp e1
100                             n2 <- evalIntExp e2
101                             return (n1<n2)
102 evalBoolExp (Gt e1 e2)   = do n1 <- evalIntExp e1
103                             n2 <- evalIntExp e2
104                             return (n1>n2)
105 evalBoolExp (And b1 b2)  = do b1' <- evalBoolExp b1
106                             b2' <- evalBoolExp b2
107                             return (b1' && b2')
108 evalBoolExp (Or b1 b2 )  = do b1' <- evalBoolExp b1
109                             b2' <- evalBoolExp b2
110                             return (b1' || b2')
111 evalBoolExp (Not b)      = do b' <- evalBoolExp b
112                             return (not b')
113

```