

```

1  module Eval2 (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 -- Mónada estado
15 newtype StateError a = StateError { runStateError :: Env -> Maybe (a, Env) }
16
17 instance Monad StateError where
18     return x          = StateError (\e -> Just (x,e))
19     StateError h >=> f = StateError (\e -> case (h e) of
20                                         Nothing    -> Nothing
21                                         Just (a,e') -> runStateError (f a a
22                                                         ) e' )
23
24     -- h :: Env -> Maybe (a,Env)
25     -- f :: a -> StateError b
26
27 -- Clase para representar mónadas con estado de variables
28 class Monad m => MonadState m where
29     -- Busca el valor de una variable
30     lookfor :: Variable -> m Int
31     -- Cambia el valor de una variable
32     update :: Variable -> Int -> m ()
33
34 instance MonadState StateError where
35     lookfor v = StateError (\s -> Just (lookfor' v s, s))
36                 where lookfor' v ((u, j):ss) | v == u = j
37                                             | v /= u = lookfor' v ss
38     -- Suponemos que no se utilizan variables no declaradas en LIS
39     update v i = StateError (\s -> Just (((), update' v i s))
40                 where update' v i [] = [(v, i)]
41                       update' v i ((u, _) : ss) | v == u = (v, i) : ss
42                       update' v i ((u, j) : ss) | v /= u = (u, j) : (update' v i ss)
43
44 -- Para calmar al GHC
45 instance Functor StateError where
46     fmap = liftM
47
48 instance Applicative StateError where
49     pure = return
50     (<*>) = ap
51
52 -- Clase para representar mónadas que lanzan errores
53 class Monad m => MonadError m where
54     -- Lanza un error
55     throw :: m a
56
57 instance MonadError StateError where
58     throw = StateError (\e -> Nothing)
59
60 -- Evalua un programa en el estado nulo
61 -- CAMBIAMOS EL TIPO DE RETORNO RESPECTO AL Eval1, PARA PODER RETORNAR ERRORES.
62 eval :: Comm -> Maybe Env
63 eval p = case (runStateError (evalComm p) initState) of
64     Nothing    -> Nothing
65     Just (a,e) -> Just e
66
67 -- Evalua un comando en un estado dado

```

```

68 evalComm :: (MonadState m, MonadError m) => Comm -> m ()
69 evalComm Skip = return ()
70 evalComm (Let v n) = do n' <- evalIntExp n
71                       update v n'
72                       return ()
73
74 evalComm (Seq c1 c2) = do evalComm c1
75                           evalComm c2
76                           return ()
77
78 evalComm (Cond b ct cf) = do cond <- evalBoolExp b
79                             if cond then (do {evalComm ct ; return ()} )
80                                     else (do {evalComm cf ; return ()} )
81
82 evalComm w@(While b c) = do cond <- evalBoolExp b
83                             if cond then evalComm (Seq c w)
84                                     else return ()
85
86
87
88
89
90
91 -- Evalua una expresion entera, sin efectos laterales
92 evalIntExp :: (MonadState m, MonadError m) => IntExp -> m Int
93 evalIntExp (Const x) = return x
94 evalIntExp (Var v) = lookfor v
95 evalIntExp (UMinus e) = do n <- evalIntExp e
96                          return (-n)
97 evalIntExp (Plus e1 e2) = do n1 <- evalIntExp e1
98                             n2 <- evalIntExp e2
99                             return (n1+n2)
100
101 evalIntExp (Minus e1 e2) = do n1 <- evalIntExp e1
102                              n2 <- evalIntExp e2
103                              return (n1-n2)
104
105 evalIntExp (Times e1 e2) = do n1 <- evalIntExp e1
106                              n2 <- evalIntExp e2
107                              return (n1*n2)
108
109 evalIntExp (Div e1 e2) = do n1 <- evalIntExp e1
110                            n2 <- evalIntExp e2
111                            if n2==0 then throw else return (div n1 n2)
112
113 -- Evalua una expresion entera, sin efectos laterales
114 evalBoolExp :: (MonadState m, MonadError m) => BoolExp -> m Bool
115 evalBoolExp BTrue = return True
116 evalBoolExp BFalse = return False
117 evalBoolExp (Eq e1 e2) = do n1 <- evalIntExp e1
118                             n2 <- evalIntExp e2
119                             return (n1==n2)
120 evalBoolExp (Lt e1 e2) = do n1 <- evalIntExp e1
121                             n2 <- evalIntExp e2
122                             return (n1<n2)
123 evalBoolExp (Gt e1 e2) = do n1 <- evalIntExp e1
124                             n2 <- evalIntExp e2
125                             return (n1>n2)
126 evalBoolExp (And b1 b2) = do b1' <- evalBoolExp b1
127                              b2' <- evalBoolExp b2
128                              return (b1' && b2')
129 evalBoolExp (Or b1 b2) = do b1' <- evalBoolExp b1
130                             b2' <- evalBoolExp b2
131                             return (b1' || b2')
132 evalBoolExp (Not b) = do b' <- evalBoolExp b
133                        return (not b')
134

```