Ejercicio 1

a)

- `return x >>= f`

  $\equiv \langle def.return \rangle$

  ```
  State (\s -> (x, s)) >>= f
  ```

  $\equiv \langle def. >>= \rangle$

  ```
  State (\s -> let (v, s') = runState (State (\s -> (x, s))) s
               in runState (f v) s')
  ```

  $\equiv \langle def.runState \rangle$

  ```
  State (\s -> let (v, s') = (\s -> (x, s)) s
               in runState (f v) s')
  ```

  $\equiv_{\beta}$

  ```
  State (\s -> let (v, s') = (x, s)
               in runState (f v) s')
  ```

  $\equiv \langle def.let \rangle$

  ```
  State (\s -> runState (f x) s)
  ```

  $\equiv_{\eta}$

  ```
  State (runState (f x))
  ```

  $\equiv \langle def.\circ \rangle$

  ```
  (State . runState) (f x)
  ```

  $\equiv \langle State \circ runState \equiv id \rangle$

  ```
  f x
  ```

- `State g >>= return`

  $\equiv \langle def.\ >>= \rangle$

  ```
  State (\s -> let (v, s') = runState (State g) s
               in runState (return v) s')
  ```

  $\equiv \langle def.runState \rangle$

  ```
  State (\s -> let (v, s') = g s
               in runState (return v) s')
  ```

  $\equiv \langle def.return \rangle$

  ```
  State (\s -> let (v, s') = g s
               in runState (State (\s -> (v, s))) s')
  ```

  $\equiv \langle def.runState \rangle$

  ```
  State (\s -> let (v, s') = g s
               in (\s -> (v, s)) s')
  ```

  $\equiv_{\beta}$

  ```
  State (\s -> let (v, s') = g s
               in (v, s'))
  ```

  $\equiv \langle def.let \rangle$

  ```
  State (\s -> g s)
  ```

  $\equiv_{\eta}$

  ```
  State g
  ```

- ((St t) >>= f) >>= g

  $\equiv \langle def.\ >>= \rangle$

```
St (\s1 -> let (x2, s2) = t s1
           in runState (f x2) s2) >>= g
```

  $\equiv \langle def.\ >>= \rangle$

```
St (\s3 -> let (x4, s4) = (\s1 -> let (x2, s2) = t s1
                                  in runState (f x2) s2) s3)
           in runState (g x4) s4)
```

  $\equiv_\beta$

```
St (\s3 -> let (x4, s4) = let (x2, s2) = t s3
                          in runState (f x2) s2
           in runState (g x4) s4)
```

  $\equiv \langle Lema \rangle$

```
St (\s3 -> let (x2, s2) = t s3
           in let (x4, s4) = runState (f x2) s2
              in runState (g x4) s4)
```

  $\equiv_\beta$

```
St (\s3 -> let (x2, s2) = t s3
           in (\s5 -> let (x4, s4) = runState (f x2) s5
                      in runState (g x4) s4) s2)
```

  $\equiv \langle runState \circ St \equiv id \rangle$

```
St (\s3 -> let (x2, s2) = t s3
           in runState (St (\s5 -> let (x4, s4) = runState (f x2) s5
                                   in runState (g x4) s4) s2)
```

  $\equiv \langle def.\ >>= \rangle$

```
St (\s3 -> let (x2, s2) = t s3
           in runState ((f x2) >>= g) s2)
```

  $\equiv_\beta$

```
St (\s3 -> let (x2, s2) = t s3
           in runState ((\w -> f w >>= g) x2) s2)
```

  $\equiv \langle def.\ >>= \rangle$

```
St t >>= (\w -> f w >>= g)
```

- Lema:

```
let x = let y = z
        in f y
in g x
```

$\equiv \langle y \notin FV\,(g\ x) \rangle$

```
let y = z
in let x = f y
   in g x
```

b)

- ```
  evalComm :: MonadState m => Comm -> m ()
  evalComm (Skip) = return ()
  evalComm (Let v e) = do  x <- evalIntExp e;
                           update v x
  evalComm (Seq com1 com2) = (evalComm com1) >> (evalComm com2)
  evalComm (Cond expB com1 com2) = do b <- evalBoolExp expB;
                                      else evalComm com2
  evalComm (While expB com) = do b <- evalBoolExp expB;
                                 if b
                                 then evalComm (Seq com (While expB com))
                                 else  (evalComm Skip)
  ```

- ```
  evalIntExp :: MonadState m => IntExp -> m Int
  evalIntExp (Const n)     = return n
  evalIntExp (Var s)       = lookfor s
  evalIntExp (UMinus e)    = do x <- evalIntExp e;
                                return (negate x)
  evalIntExp (Plus e1 e2)  = abstract evalIntExp (+) e1 e2
  evalIntExp (Minus e1 e2) = abstract evalIntExp (-) e1 e2
  evalIntExp (Times e1 e2) = abstract evalIntExp (*) e1 e2
  evalIntExp (Div e1 e2)   = abstract evalIntExp div e1 e2
  ```

- ```
  evalBoolExp :: MonadState m => BoolExp -> m Bool
  evalBoolExp BTrue       = return True
  evalBoolExp BFalse      = return False
  evalBoolExp (Eq e1 e2)  = abstract evalIntExp (==) e1 e2
  evalBoolExp (Lt e1 e2)  = abstract evalIntExp (<) e1 e2
  evalBoolExp (Gt e1 e2)  = abstract evalIntExp (>) e1 e2
  evalBoolExp (And e1 e2) = abstract evalBoolExp (&&) e1 e2
  evalBoolExp (Or e1 e2)  = abstract evalBoolExp (||) e1 e2
  evalBoolExp (Not e)     = do b <- evalBoolExp e;
                               return (not b)
  ```

- ```
  abstract :: Monad m => (a -> m b) -> (b -> b -> c) -> a -> a -> m c
  abstract ev op e1 e2 = do x <- ev e1;
                            y <- ev e2;
                            return (op x y)
  ```

Ejercicio 2

a)
```
instance Monad StateError where
return x = StateError (\s -> Just (x,s))
t >>= f  = StateError (\s ->
                      (runStateError t s >>= \(x,s') -> (runStateError (f x) s')))
```

b)
```
instance MonadError StateError where
     throw = StateError (\s -> Nothing)
```

c)
```
instance MonadState StateError where
lookfor v = StateError (\s -> lookfor' v s s) where
           lookfor' _ [] _ = Nothing
           lookfor' v ((u, j):ss) s' | v == u = Just (j,s')
                                     | otherwise = lookfor' v ss s'

           update v i = StateError (\s -> update' v i s) where
                        update' v i [] = Just((),[(v,i)])
                        update' v i ((a,b):xs) | v == a = Just ((),(a,i):xs)
                                               | v /= a = update' v i xs >>=
                                                          \(x,y) -> Just ((),(a,b):y)
```

d)

- ```
  eval :: Comm -> Env
  eval p = case (runStateError (evalComm p) initState) of
                Nothing -> []
                Just (_,s) -> s
  ```

- ```
  evalComm :: (MonadState m, MonadError m) => Comm -> m ()
  evalComm (Skip) = return ()
  evalComm (Let v e) = do  x <- evalIntExp e;
                           update v x
  evalComm (Seq com1 com2) = (evalComm com1) >> (evalComm com2)
  evalComm (Cond expB com1 com2) = do b <- evalBoolExp expB;
                                      else evalComm com2
  evalComm (While expB com) = do b <- evalBoolExp expB;
                                 if b
                                 then evalComm (Seq com (While expB com))
                                 else  (evalComm Skip)
  ```

- ```
  evalIntExp :: (MonadState m, MonadError m) => IntExp -> m Int
  evalIntExp (Const n)     = return n
  evalIntExp (Var s)       = lookfor s
  evalIntExp (UMinus e)    = do x <- evalIntExp e;
                                return (negate x)
  evalIntExp (Plus e1 e2)  = abstract evalIntExp (+) e1 e2
  evalIntExp (Minus e1 e2) = abstract evalIntExp (-) e1 e2
  evalIntExp (Times e1 e2) = abstract evalIntExp (*) e1 e2
  evalIntExp (Div e1 e2)   = do x <- evalIntExp e2
                                if x == 0 then throw
                                else abstract evalIntExp div e1 e2
  ```
  ```

- ```
  evalBoolExp :: (MonadState m, MonadError m) => BoolExp -> m Bool
  evalBoolExp BTrue       = return True
  evalBoolExp BFalse      = return False
  evalBoolExp (Eq e1 e2)  = abstract evalIntExp (==) e1 e2
  evalBoolExp (Lt e1 e2)  = abstract evalIntExp (<) e1 e2
  evalBoolExp (Gt e1 e2)  = abstract evalIntExp (>) e1 e2
  evalBoolExp (And e1 e2) = abstract evalBoolExp (&&) e1 e2
  evalBoolExp (Or e1 e2)  = abstract evalBoolExp (||) e1 e2
  evalBoolExp (Not e)     = do b <- evalBoolExp e;
                               return (not b)
  ```
- ```
  abstract :: Monad m => (a -> m b) -> (b -> b -> c) -> a -> a -> m c
  abstract ev op e1 e2 = do x <- ev e1;
                            y <- ev e2;
                            return (op x y)
  ```

Ejercicio 3

a)
```
newtype StateErrorTick a = StateErrorT
                           { runStateError :: Env -> Maybe (a, Env, Int) }
```

b)
```
class Monad m => MonadTick m where
    tick :: m ()
```

c)
```
instance MonadTick StateErrorTick where
    tick = StateErrorT (\s -> Just ((),s,1))
```

d)
```
instance MonadError StateErrorTick where
    throw = StateErrorT (\s -> Nothing)
```

e)
```
instance MonadState StateErrorTick where
lookfor v = StateErrorT (\s -> lookfor' v s s) where
            lookfor' _ [] _ = Nothing
            lookfor' v ((u, j):ss) s' | v == u = Just (j,s',0)
                                      | otherwise = lookfor' v ss s'

            update v i = StateErrorT (\s -> update' v i s) where
                         update' v i [] = Just((),[(v,i)], 0)
                         update' v i ((a,b):xs) | v == a = Just ((), (a,i):xs, 0)
                                                | v /= a = update' v i xs >>=
                                                           \(x,y,n) -> Just ((),(a,b):y, n)
```

f)

- ```
  eval :: Comm -> (Env,Int)
  eval p = case (runStateError (evalComm p) initState) of
           Nothing -> ([],0)
           Just (_,s,n) -> (s,n)
  ```

- ```
  evalComm :: (MonadState m, MonadError m, MonadTick m) => Comm -> m ()
  evalComm (Skip) = return ()
  evalComm (Let v e) = do  x <- evalIntExp e;
                           update v x
  evalComm (Seq com1 com2) = (evalComm com1) >> (evalComm com2)
  evalComm (Cond expB com1 com2) = do b <- evalBoolExp expB;
                                      else evalComm com2
  evalComm (While expB com) = do b <- evalBoolExp expB;
                                 if b
                                 then evalComm (Seq com (While expB com))
                                 else (evalComm Skip)
  ```

- ```
  evalIntExp :: (MonadState m, MonadError m, MonadTick m) => IntExp -> m Int
  evalIntExp (Const n)     = return n
  evalIntExp (Var s)       = lookfor s
  evalIntExp (UMinus e)    = do x <- evalIntExp e;
                                return (negate x)
  evalIntExp (Plus e1 e2)  = abstract evalIntExp (+) e1 e2 True
  evalIntExp (Minus e1 e2) = abstract evalIntExp (-) e1 e2 True
  evalIntExp (Times e1 e2) = abstract evalIntExp (*) e1 e2 True
  evalIntExp (Div e1 e2)   = do x <- evalIntExp e2
                                if x == 0 then throw
                                else abstract evalIntExp div e1 e2 True
  ```

- ```
  evalBoolExp ::(MonadState m, MonadError m, MonadTick m) => BoolExp -> m Bool
  evalBoolExp BTrue       = return True
  evalBoolExp BFalse      = return False
  evalBoolExp (Eq e1 e2)  = abstract evalIntExp (==) e1 e2 False
  evalBoolExp (Lt e1 e2)  = abstract evalIntExp (<) e1 e2 False
  evalBoolExp (Gt e1 e2)  = abstract evalIntExp (>) e1 e2 False
  evalBoolExp (And e1 e2) = abstract evalBoolExp (&&) e1 e2 False
  evalBoolExp (Or e1 e2)  = abstract evalBoolExp (||) e1 e2 False
  evalBoolExp (Not e)     = do b <- evalBoolExp e;
                               return (not b)
  ```

- ```
  abstract ev op e1 e2 b = do x <- ev e1;
                              y <- ev e2;
                              if b then do tick;
                                           return (op x y)
                              else return (op x y)
  ```
  ```