Universidad Nacional de Rosario
Facultad de Ciencias Exactas,
Ingeniería y Agrimensura

Lic. en Cs. de la Computación

Análisis de Lenguajes de Programación

Trabajo Práctico 4

Tomás Fernández De Luco F-3443/6
Ignacio Sebastián Moline M-6466/1

30 de noviembre de 2018

Ejercicio 1

a) Demostración de State como mónada:

```
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')

-- Monad 1
return y >>= f =
-- {Definicion de return}
State (\s -> (y, s)) >>= f =
-- {Definicion de >>=}
State (\s -> let (v, s') = runState (State (\s -> (y, s))) s
             in runState (f v) s') =
-- {Definicion de runState}
State (\s -> let (v, s') = (\s -> (y, s)) s
             in runState (f v) s') =
-- {Aplicacion de funcion anonima}
State (\s -> let (v, s') = (y, s)
             in runState (f v) s') =
-- {Sustitucion de let}
State (\s -> runState (f y) s) =
-- {Eta-reduccion}
State (runState (f y)) =
-- {Sea x = State z. Luego, State (runState x) = State z = x}
-- {Entonces, la composicion de State y runState es la identidad}
f y

-- Monad 2
m >>= return =
-- {Definicion de >>=}
State (\s -> let (v, s') = runState m s
             in runstate (return v) s') =
-- {Definicion de return}
State (\s -> let (v, s') = runState m s
             in runstate (State (\s -> (v, s))) s') =
-- {Definicion de runState}
State (\s -> let (v, s') = runState m s
             in (\s -> (v, s)) s') =
-- {Aplicacion de funcion anonima}
State (\s -> let (v, s') = runState m s
             in (v, s')) =
-- {Sustitucion de let}
State (\s -> runState m s) =
--{Eta-reduccion}
State (runState m) =
-- {La composicion de State y runState es la identidad}
m

-- Monad 3
(m >>= f) >>= g =
-- {Definicion de >>=}
(State (\s -> let (v, s') = runState m s
              in runState (f v) s')) >>= g =
-- {Definicion de >>=}
State (\s -> let (v, s') = runState (State (\s -> let (v, s') = runState m s
                                                  in runState (f v) s')) s
             in runState (g v) s') =
```

```haskell
-- {Definicion de runState}
State (\s -> let (v, s') = (\s -> let (v, s') = runState m s
                                  in runState (f v) s')) s
              in runState (g v) s') =
-- {Aplicacion de funcion anonima}
State (\s -> let (v, s') = (let (v, s') = runState m s
                            in runState (f v) s')
              in runState (g v) s') =
-- {Se reescribe el let de la siguiente forma:
-- let a = (let b = c in f(b)) in g(a)
-- let b = c
--     a = f(b)
--  in g(a)}
State (\s -> let (v, s')  = runState m s
                 (w, s'') = runState (f v) s'
             in runState (g w) s'')


m >>= (\x -> f x >>= g) =
-- {Definicion de >>=}
State (\s -> let (v, s') = runState m s
             in runState ((\x -> f x >>= g) v) s') =
-- {Aplicacion de funcion anonima}
State (\s -> let (v, s') = runState m s
             in runState (f v >>= g) s') =
-- {Definicion de >>=}
State (\s -> let (v, s') = runState m s
              in runState (State (\s -> let (v, s') = runState (f v) s
                                        in runState (g v) s')) s') =
-- {Definicion de runState}
State (\s -> let (v, s') = runState m s
              in (\s -> let (v, s') = runState (f v) s
                        in runState (g v) s') s') =
-- {Aplicacion de funcion anonima}
State (\s -> let (v, s') = runState m s
              in (let (v, s') = runState (f v) s'
                   in runState (g v) s')) =
-- {Se reescribe el let de la siguiente forma:
-- let a = b in (let c = f(a) in g(c))
-- let a = b
--     c = f(a)
--  in g(c)}
State (\s -> let (v, s')  = runState m s
                 (w, s'') = runState (f v) s'
             in runState (g w) s'')

-- {Ambos terminos son iguales a una misma expresion}
-- {Por transitividad, (m >>= f) >>= g = m >>= (\x -> f x >>= g)}
```

b) Implementación del evaluador para la mónada State:

```haskell
-- 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 var ie)      = do n <- evalIntExp ie
                                update var n
evalComm (Seq c1 c2)       = do evalComm c1
                                evalComm c2
evalComm (Cond be c1 c2) = do b <- evalBoolExp be
                                if b then (evalComm c1)
                                     else (evalComm c2)
evalComm (While be c)      = evalComm (Cond be (Seq c (While be c)) Skip)


-- Resuelve la evaluacion de dos expresiones enteras, aplicadas a un operador.
evalIntOp :: MonadState m => IntExp -> IntExp -> (Int -> Int -> a) -> m a
evalIntOp ie1 ie2 operator = do n1 <- evalIntExp ie1
                                n2 <- evalIntExp ie2
                                return (operator n1 n2)


-- Evalua una expresion entera, sin efectos laterales
evalIntExp :: MonadState m => IntExp -> m Int
evalIntExp (Const i)        = return i
evalIntExp (Var x)          = lookfor x
evalIntExp (UMinus ie)      = do n <- evalIntExp ie
                                 return (-n)
evalIntExp (Plus ie1 ie2)  = evalIntOp ie1 ie2 (+)
evalIntExp (Minus ie1 ie2) = evalIntOp ie1 ie2 (-)
evalIntExp (Times ie1 ie2) = evalIntOp ie1 ie2 (*)
evalIntExp (Div ie1 ie2)   = evalIntOp ie1 ie2 div


-- Resuelve la evaluacion de dos expresiones booleanas, aplicadas a un operador.
evalBoolOp :: MonadState m => BoolExp -> BoolExp -> (Bool -> Bool -> a) -> m a
evalBoolOp be1 be2 operator = do n1 <- evalBoolExp be1
                                 n2 <- evalBoolExp be2
                                 return (operator n1 n2)


-- Evalua una expresion booleana, sin efectos laterales
evalBoolExp :: MonadState m => BoolExp -> m Bool
evalBoolExp BTrue           = return True
evalBoolExp BFalse          = return False
evalBoolExp (Eq ie1 ie2)    = evalIntOp ie1 ie2 (==)
evalBoolExp (Lt ie1 ie2)    = evalIntOp ie1 ie2 (<)
evalBoolExp (Gt ie1 ie2)    = evalIntOp ie1 ie2 (>)
evalBoolExp (And be1 be2)   = evalBoolOp be1 be2 (&&)
evalBoolExp (Or be1 be2)    = evalBoolOp be1 be2 (||)
evalBoolExp (Not be)        = do b <- evalBoolExp be
                                 return (not b)
```

Ejercicio 2

a) Instancia de Monad para StateError:

```
instance Monad StateError where
    return x = StateError (\s -> Just (x, s))
    m >>= f = StateError (\s -> case runStateError m s of
                                    Nothing -> Nothing
                                    Just (v, s') -> runStateError (f v) s')
```

b) Instancia de MonadError para StateError:

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

c) Instancia de MonadState para StateError:

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

d) Implementación del evaluador utilizando la mónada StateError:

```haskell
-- Evalua un programa en el estado nulo
eval :: Comm -> Env
eval c = maybe initState snd (runStateError (evalComm c) initState)

-- Evalua un comando en un estado dado
evalComm :: (MonadState m, MonadError m) => Comm -> m ()
evalComm Skip               = return ()
evalComm (Let var ie)       = do n <- evalIntExp ie
                                 update var n
evalComm (Seq c1 c2)        = do evalComm c1
                                 evalComm c2
evalComm (Cond be c1 c2) = do b <- evalBoolExp be
                              if b then (evalComm c1)
                                   else (evalComm c2)
evalComm (While be c)       = evalComm (Cond be (Seq c (While be c)) Skip)


-- Resuelve la evaluacion de dos expresiones enteras, aplicadas a un operador.
evalIntOp :: (MonadState m, MonadError m) => IntExp -> IntExp -> (Int -> Int -> a)
    -> m a
evalIntOp ie1 ie2 operator = do n1 <- evalIntExp ie1
                                n2 <- evalIntExp ie2
                                return (operator n1 n2)


-- Evalua una expresion entera, sin efectos laterales
evalIntExp :: (MonadState m, MonadError m) => IntExp -> m Int
evalIntExp (Const i)         = return i
evalIntExp (Var x)           = lookfor x
evalIntExp (UMinus ie)       = do n <- evalIntExp ie
                                  return (-n)
evalIntExp (Plus ie1 ie2)    = evalIntOp ie1 ie2 (+)
evalIntExp (Minus ie1 ie2) = evalIntOp ie1 ie2 (-)
evalIntExp (Times ie1 ie2) = evalIntOp ie1 ie2 (*)
evalIntExp (Div ie1 ie2)     = do n1 <- evalIntExp ie1
                                  n2 <- evalIntExp ie2
                                  if n2 == 0 then throw
                                             else return (div n1 n2)


-- Resuelve la evaluacion de dos expresiones booleanas, aplicadas a un operador.
evalBoolOp :: (MonadState m, MonadError m) => BoolExp -> BoolExp -> (Bool -> Bool -> a)
    -> m a
evalBoolOp be1 be2 operator = do n1 <- evalBoolExp be1
                                 n2 <- evalBoolExp be2
                                 return (operator n1 n2)


-- Evalua una expresion booleana, sin efectos laterales
evalBoolExp :: (MonadState m, MonadError m) => BoolExp -> m Bool
evalBoolExp BTrue           = return True
evalBoolExp BFalse          = return False
evalBoolExp (Eq ie1 ie2)    = evalIntOp ie1 ie2 (==)
evalBoolExp (Lt ie1 ie2)    = evalIntOp ie1 ie2 (<)
evalBoolExp (Gt ie1 ie2)    = evalIntOp ie1 ie2 (>)
evalBoolExp (And be1 be2) = evalBoolOp be1 be2 (&&)
evalBoolExp (Or be1 be2)    = evalBoolOp be1 be2 (||)
evalBoolExp (Not be)        = do b <- evalBoolExp be
                                 return (not b)
```

Ejercicio 3

a) Mónada propuesta:

```haskell
-- Monada estado
newtype StateErrorTick a =
        StateErrorTick {runStateErrorTick :: Env -> Maybe (a, Int, Env)}
```

b) Clase MonadTick:

```haskell
class Monad m => MonadTick m where
    -- Crea un contador en 1
    tick :: m ()
```

c) Instancia de MonadTick:

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

d) Instancia de MonadError:

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

e) Instancia de MonadState:

```haskell
instance MonadState StateErrorTick where
lookfor v = StateErrorTick (\s -> maybe (Nothing) (\u -> Just (u, 0, s)) (lookfor' v s))
                                  where lookfor' v [] = Nothing
                                        lookfor' v ((u,j):ss) | v == u    = Just j
                                                              | otherwise = lookfor' v ss
update v i = StateErrorTick (\s -> Just ((), 0, update' v i s))
             where update' v i [] = [(v, i)]
                   update' v i ((u, j):ss) | v == u    = (v, i):ss
                                           | otherwise = (u, j):(update' v i ss)
```

f) Implementación del evaluador utilizando la mónada StateErrorTick:

```haskell
-- Evalua un programa en el estado nulo
eval :: Comm -> (Int, Env)
eval c = maybe ((-1), [])
               (\(v, count, s) -> (count, s))
               (runStateErrorTick (evalComm c) initState)

-- Evalua un comando en un estado dado
evalComm :: (MonadState m, MonadError m, MonadTick m) => Comm -> m ()
evalComm Skip              = return ()
evalComm (Let var ie)      = do n <- evalIntExp ie
                                update var n
evalComm (Seq c1 c2)       = do evalComm c1
                                evalComm c2
evalComm (Cond be c1 c2) = do b <- evalBoolExp be
                              if b then (evalComm c1)
                                   else (evalComm c2)
evalComm (While be c)      = evalComm (Cond be (Seq c (While be c)) Skip)


-- Resuelve la evaluacion de dos expresiones enteras, aplicadas a un operador.
evalIntOp :: (MonadState m, MonadError m, MonadTick m) => IntExp -> IntExp ->
    (Int -> Int -> a) -> m a
evalIntOp ie1 ie2 operator = do n1 <- evalIntExp ie1
                                n2 <- evalIntExp ie2
                                return (operator n1 n2)


-- Evalua una expresion entera, sin efectos laterales.
evalIntExp :: (MonadState m, MonadError m, MonadTick m) => IntExp -> m Int
evalIntExp (Const i)         = return i
evalIntExp (Var x)           = lookfor x
evalIntExp (UMinus ie)       = do n <- evalIntExp ie
                                  return (-n)
evalIntExp (Plus ie1 ie2)    = tick >> evalIntOp ie1 ie2 (+)
evalIntExp (Minus ie1 ie2) = tick >> evalIntOp ie1 ie2 (-)
evalIntExp (Times ie1 ie2) = tick >> evalIntOp ie1 ie2 (*)
evalIntExp (Div ie1 ie2)     = do n1 <- evalIntExp ie1
                                  n2 <- evalIntExp ie2
                                  if n2 == 0 then throw
                                             else tick >> return (div n1 n2)

-- Resuelve la evaluacion de dos expresiones booleanas, aplicadas a un operador.
evalBoolOp :: (MonadState m, MonadError m, MonadTick m) => BoolExp -> BoolExp ->
    (Bool -> Bool -> a) -> m a
evalBoolOp be1 be2 operator = do n1 <- evalBoolExp be1
                                 n2 <- evalBoolExp be2
                                 return (operator n1 n2)

-- Evalua una expresion booleana, sin efectos laterales
evalBoolExp :: (MonadState m, MonadError m, MonadTick m) => BoolExp -> m Bool
evalBoolExp BTrue            = return True
evalBoolExp BFalse           = return False
evalBoolExp (Eq ie1 ie2)     = evalIntOp ie1 ie2 (==)
evalBoolExp (Lt ie1 ie2)     = evalIntOp ie1 ie2 (<)
evalBoolExp (Gt ie1 ie2)     = evalIntOp ie1 ie2 (>)
evalBoolExp (And be1 be2) = evalBoolOp be1 be2 (&&)
evalBoolExp (Or be1 be2)     = evalBoolOp be1 be2 (||)
evalBoolExp (Not be)         = do b <- evalBoolExp be
                                  return (not b)
```