Trabajo práctico n° 1

Integrantes:

Ignacio Sebastián Moliné. Legajo M-6466/1

Sebastián Morales. Legajo M-6501/3

Ejercicio 1.

Extendemos la sintaxis abstracta y concreta de LIS para incluir asignaciones enteras, de la forma $x = e$ y el operador , para escribir una secuencia de expresiones enteras.

- *Sintaxis abstracta:*

$$
\begin{aligned}
intexp \;::=\; & \; nat \mid var \mid -_u \; intexp \\
\mid \; & \; var \; = \; intexp \\
\mid \; & \; intexp \; , \; intexp \\
\mid \; & \; intexp \; + \; intexp \\
\mid \; & \; intexp \; -_b \; intexp \\
\mid \; & \; intexp \; \times \; intexp \\
\mid \; & \; intexp \; \div \; intexp
\end{aligned}
$$

- *Sintaxis concreta:*

$$
\begin{aligned}
intexp \;::=\; & \; nat \\
\mid \; & \; var \\
\mid \; & \; '-' \; intexp \\
\mid \; & \; var \; '=' \; intexp \\
\mid \; & \; intexp \; ' , ' \; intexp \\
\mid \; & \; intexp \; '+' \; intexp \\
\mid \; & \; intexp \; '-' \; intexp \\
\mid \; & \; intexp \; '*' \; intexp \\
\mid \; & \; intexp \; '/' \; intexp \\
\mid \; & \; '(' \; intexp \; ')'
\end{aligned}
$$

Ejercicio 2.

Se deben agregar dos constructores más al tipo de dato Exp a, con los tipos de datos acordes a sus requisitos.

```
-- Expresiones, aritmeticas y booleanas
data Exp a where
  -- Expresiones enteras
  Const ::Int -> Exp Int
  Var ::Variable -> Exp Int
  EAssgn :: Variable -> Exp Int -> Exp Int
  ESeq :: Exp Int -> Exp Int -> Exp Int
  UMinus ::Exp Int -> Exp Int
  Plus ::Exp Int -> Exp Int -> Exp Int
  Minus ::Exp Int -> Exp Int -> Exp Int
  Times ::Exp Int -> Exp Int -> Exp Int
  Div ::Exp Int -> Exp Int -> Exp Int
  -- Expresiones booleanas
  BTrue ::Exp Bool
  BFalse ::Exp Bool
  Lt ::Exp Int -> Exp Int -> Exp Bool
  Gt ::Exp Int -> Exp Int -> Exp Bool
  And ::Exp Bool -> Exp Bool -> Exp Bool
  Or ::Exp Bool -> Exp Bool -> Exp Bool
  Not ::Exp Bool -> Exp Bool
  Eq ::Exp Int -> Exp Int -> Exp Bool
  NEq ::Exp Int -> Exp Int -> Exp Bool
```

Ejercicio 3.

El código del archivo Parser.hs es el siguiente.

```
intexp :: Parser (Exp Int)
intexp = chainl1 intexp' (try (do reservedOp lis ","
                                  return ESeq))

intexp' = chainl1 term addopp

term = chainl1 factor multopp

factor = try (parens lis intexp)
         <|> try (do reservedOp lis "-"
                     f <- factor
                     return (UMinus f))
         <|> try (do n <- integer lis
                     return (Const $ fromInteger n))
         <|> try (do str <- identifier lis
                     reservedOp lis "="
                     e <- intexp
                     return (EAssgn str e))
         <|> try (do str <- identifier lis
                     return (Var str))

addopp = do try (reservedOp lis "+")
            return Plus
         <|> do try (reservedOp lis "-")
                return Minus

multopp = do try (reservedOp lis "*")
             return Times
          <|> do try (reservedOp lis "/")
                 return Div

-------------------------------------
```

```
-------------------------------------
--- Parser de expressiones booleanas
-------------------------------------

boolexp :: Parser (Exp Bool)
boolexp = chainl1 boolexp' (try (do reservedOp lis "||"
                                    return Or))

boolexp' = chainl1 boolexp'' (try (do reservedOp lis "&&"
                                      return And))

boolexp'' = try (parens lis boolexp)
            <|> try (do reservedOp lis "!"
                        b <- boolexp''
                        return (Not b))
            <|> intComp
            <|> boolValue

intComp = try (do a <- intexp
                  c <- compOp
                  b <- intexp
                  return (c a b))

compOp = try (do reservedOp lis "=="
                 return Eq)
         <|> (do reservedOp lis "!="
                 return NEq)
         <|> (do reservedOp lis "<"
                 return Lt)
         <|> (do reservedOp lis ">"
                 return Gt)

boolValue = try (do reserved lis "true"
                    return BTrue)
            <|> try (do reserved lis "false"
                        return BFalse)
```

```haskell
comm :: Parser Comm
comm = chainl1 comm' (try (do reservedOp lis ";"
                              return Seq))

comm' = try (do reserved lis "skip"
                return Skip)
       <|> try (do reserved lis "if"
                   cond <- boolexp
                   symbol lis "{"
                   case1 <- comm
                   symbol lis "}"
                   reserved lis "else"
                   symbol lis "{"
                   case2 <- comm
                   symbol lis "}"
                   return (IfThenElse cond case1 case2))
       <|> try (do reserved lis "if"
                   cond <- boolexp
                   symbol lis "{"
                   case1 <- comm
                   symbol lis "}"
                   return (IfThen cond case1))
       <|> try (do reserved lis "while"
                   cond <- boolexp
                   symbol lis "{"
                   c <- comm
                   symbol lis "}"
                   return (While cond c))
       <|> try (do str <- identifier lis
                   reservedOp lis "="
                   e <- intexp
                   return (Let str e))
```

Ejercicio 4.

Agregamos dos reglas para especificar tanto el comportamiento de la asignación de expresiones enteras como expresiones, así como el operador ‘,’ para secuencias de expresiones enteras:

$$\frac{<e_0, \sigma> \Downarrow_{exp} <n_0, \sigma'>}{<x := e_0, \sigma> \Downarrow_{exp} <n_0, [\sigma' | x : n_0]>} EASSGN$$

$$\frac{<e_0, \sigma> \Downarrow_{exp} <n_0, \sigma'> \quad <e_1, \sigma'> \Downarrow_{exp} <n_1, \sigma''>}{<e_0, e_1, \sigma> \Downarrow_{exp} <n_1, \sigma''>} ESEQ$$

Ejercicio 5.

La relación $\rightsquigarrow$ es determinista si

$$\text{si } t \rightsquigarrow t_1 \text{ y } t \rightsquigarrow t_2 \text{ entonces } t_1 = t_2$$

Suponemos que $\Downarrow$ es determinista (Sup) y demostramos por inducción. Si la última regla utilizada fue:

- ASS:
  - $\langle e, \sigma \rangle \Downarrow_{intexp} n$
  - $t = \langle v := e, \sigma \rangle$
  - $t_1 = [\sigma | v : n]$

  Por Sup y como ASS es la única regla aplicable en t, $t_1 = t_2$.
- SKIP:
  - $t = \langle \textbf{skip}, \sigma \rangle$
  - $t_1 = \sigma$

  Como SKIP es la única regla aplicable en t, $t_1 = t_2$.
- SEQ1:
  - $\langle c_0, \sigma \rangle \rightsquigarrow \sigma'$
  - $t = \langle c_0; c_1, \sigma \rangle$
  - $t_1 = \langle c_1, \sigma' \rangle$

  Por HI, $\langle c_0, \sigma \rangle \rightsquigarrow \sigma'$ es determinista, por lo que no puede ser $\langle c_0, \sigma \rangle \rightsquigarrow \langle c_0', \sigma' \rangle$. Luego, no puedo aplicar SEQ2. Como SEQ1 es la única regla aplicable a t, $t_1 = t_2$.

- SEQ2:
  - $\langle c_0, \sigma \rangle \rightsquigarrow \langle c_0', \sigma' \rangle$
  - $t = \langle c_0; c_1, \sigma' \rangle$
  - $t_1 = \langle c_0'; c_1, \sigma \rangle$

  Análogamente, no puedo usar SEQ1 por HI. Luego, como SEQ2 es la única regla aplicable en t, $t_1 = t_2$.
- IF1:
  - $\langle b, \sigma \rangle \Downarrow_{boolexp} \textbf{true}$
  - $t = \langle \textbf{if } b \textbf{ then } c_0, \sigma \rangle$
  - $t_1 = \langle c_0, \sigma \rangle$

  Por Sup, $\Downarrow_{boolexp}$ es determinista. Luego, no puede ser $\langle b, \sigma \rangle \Downarrow_{boolexp} \textbf{false}$ y no puedo aplicar IF2. Como IF1 es la única regla aplicable en t, $t_1 = t_2$.
- IF2: Análogamente a IF1.
- WHILE1:
  - $\langle b, \sigma \rangle \Downarrow_{boolexp} \textbf{true}$
  - $t = \langle \textbf{while } b \textbf{ do } c, \sigma \rangle$
  - $t_1 = \langle c; \textbf{while } b \textbf{ do } c, \sigma \rangle$

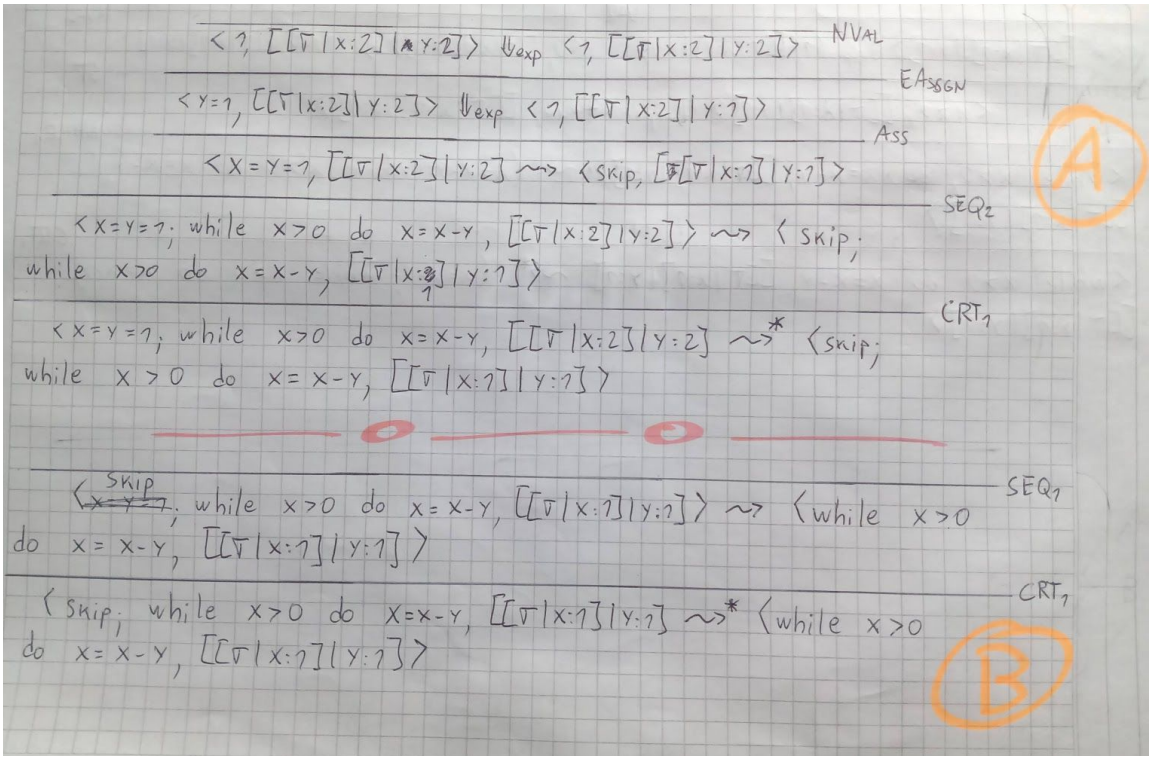
  Por Sup, $\Downarrow_{boolexp}$ es determinista, por lo que no puede ser $\langle b, \sigma \rangle \Downarrow_{boolexp} \textbf{false}$ y no puedo aplicar WHILE2. Como WHILE1 es la única regla aplicable en t, $t_1 = t_2$.
- WHILE2: Análogamente a WHILE1.

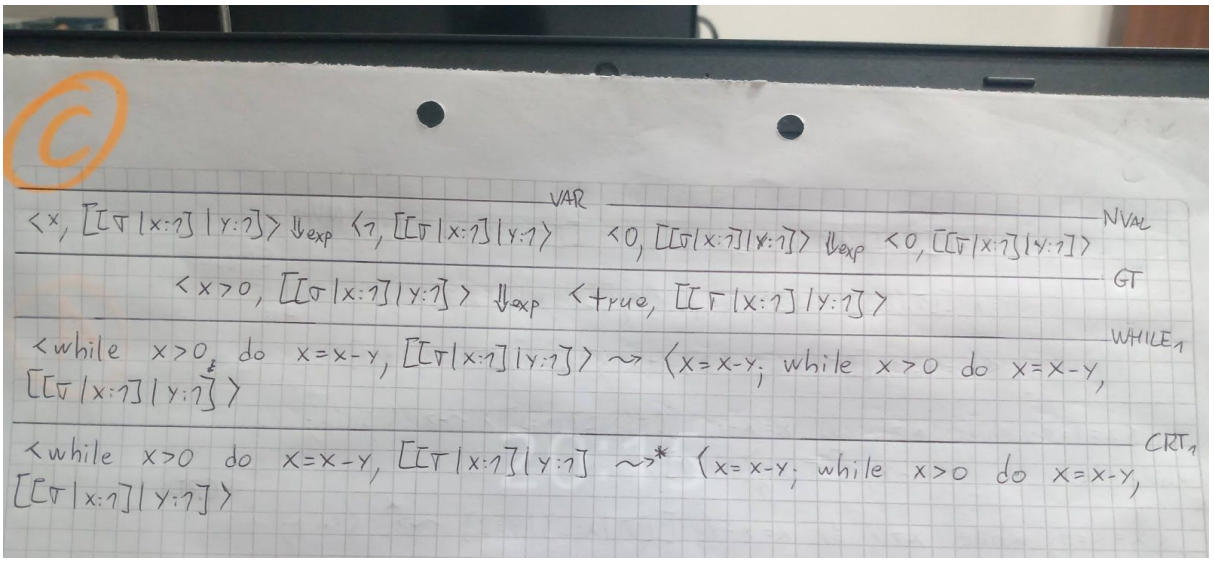
Ejercicio 6.

Las reglas denominadas CRTi, con i entre 1 y 3, corresponden a las reglas de la clausura reflexo-transitiva, enumeradas en el orden que muestra la siguiente imagen.

$$\frac{t \to t'}{t \to^* t'} \qquad \frac{}{t \to^* t} \qquad \frac{t \to^* t' \qquad t' \to^* t''}{t \to^* t''}$$

El árbol de derivación fue subdividido en varios árboles (primero, entre A y F; luego, entre I y III) para que la demostración sea más legible.





D

$\langle x, [[\top|x:2]|y:7]\rangle \Downarrow_{exp} \langle 7, [[\top|x:7]|y:7]\rangle$ VAR

$\langle y, [[\top|x:7]|y:7]\rangle \Downarrow_{exp} \langle 7, [[\top|x:7]|y:7]\rangle$ VAR

$\langle x-y, [[\top|x:7]|y:7]\rangle \Downarrow_{exp} \langle 0, [[\top|x:7]|y:7]\rangle$ BMINUS

$\langle x=x-y, [[\top|x:7]|y:7]\rangle \rightsquigarrow \langle skip, [[\top|x:0]|y:7]\rangle$ ASS

$\langle x=x-y; \text{while } x>0 \text{ do } x=x-y, [[\top|x:7]|y:7]\rangle \rightsquigarrow \langle skip; \text{while } x>0 \text{ do } x=x-y, [[\top|x:0]|y:7]\rangle$ SEQ2

$\langle x=x-y; \text{while } x>0 \text{ do } x=x-y, [[\top|x:7]|y:7]\rangle \rightsquigarrow^* \langle skip; \text{while } x>0 \text{ do } x=x-y, [[\top|x:0]|y:7]\rangle$ CRT1

$\langle skip; \text{while } x>0 \text{ do } x=x-y, [[\top|x:0]|y:7]\rangle \rightsquigarrow \langle \text{while } x>0 \text{ do } x=x-y, [[\top|x:0]|y:7]\rangle$ SEQ1

$\langle skip; \text{while } x>0 \text{ do } x=x-y, [[\top|x:0]|y:7]\rangle \rightsquigarrow^* \langle \text{while } x>0 \text{ do } x=x-y, [[\top|x:0]|y:7]\rangle$ CRT1

E

F

$\langle x, [[\top|x:0]|y:7]\rangle \Downarrow_{exp} \langle 0, [[\top|x:0]|y:7]\rangle$ VAR

$\langle 0, [[\top|x:0]|y:7]\rangle \Downarrow_{exp} \langle 0, [[\top|x:0]|y:7]\rangle$ NVAL

$\langle x>0, [[\top|x:0]|y:7]\rangle \Downarrow_{exp} \langle false, [[\top|x:0]|y:7]\rangle$ GT

$\langle \text{while } x>0 \text{ do } x=x-y, [[\top|x:0]|y:7]\rangle \rightsquigarrow \langle skip; [[\top|x:0]|y:7]\rangle$ WHILE2

$\langle \text{while } x>0 \text{ do } x=x-y, [[\top|x:0]|y:7]\rangle \rightsquigarrow^* \langle skip; [[\top|x:0]|y:7]\rangle$ CRT1

FI

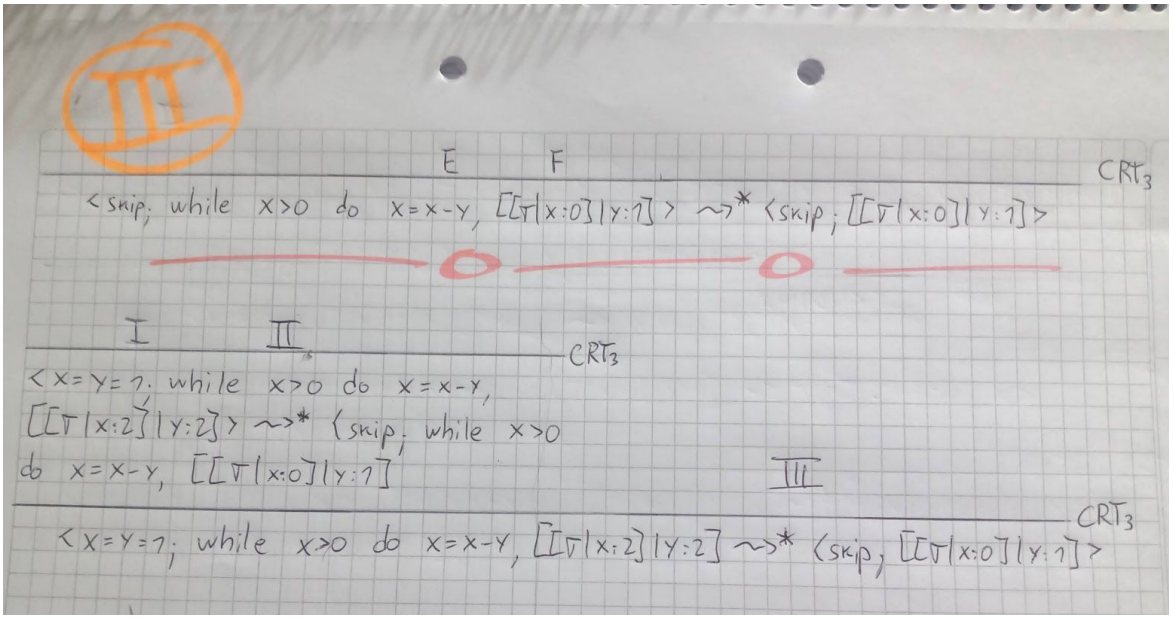
A B

$\langle x=y=7; \text{while } x>0 \text{ do } x=x-y, [[\top|x:2]|y:2]\rangle \rightsquigarrow^* \langle \text{while } x>0 \text{ do } x=x-y, [[\top|x:7]|y:7]\rangle$ CRT3

C D

$\langle \text{while } x>0 \text{ do } x=x-y, [[\top|x:7]|y:7]\rangle \rightsquigarrow^* \langle skip; \text{while } x>0 \text{ do } x=x-y, [[\top|x:0]|y:7]\rangle$ CRT3

G


III
E F
CRT3
<skip; while x>0 do x=x-y, [[⊤|x:0]|y:7]> ⇝* <skip; [[⊤|x:0]|y:7]>
I II
CRT3
<x=y=7; while x>0 do x=x-y,
[[⊤|x:2]|y:2]> ⇝* <skip; while x>0
do x=x-y, [[⊤|x:0]|y:7]
III
CRT3
<x=y=7; while x>0 do x=x-y, [[⊤|x:2]|y:2] ⇝* <skip, [[⊤|x:0]|y:7]>

Ejercicio 7.

Para el caso del error por división por cero, dejamos que Haskell maneje el error y aborte la ejecución. Para las variables no definidas se optó por hacer pattern matching exhaustivo en la función *lookfor* y manejar el error con un mensaje, dado que nos pareció más claro al momento de mostrar el error. Se podría haber optado también por dejar el pattern matching no exhaustivo y dejar que Haskell maneje el error, pero se prefirió claridad de mensajes. El código de Eval1.hs es:

```haskell
-- Estados
type State = M.Map Variable Int

-- Estado nulo
initState :: State
initState = M.empty

-- Busca el valor de una variable en un estado
lookfor :: Variable -> State -> Int
lookfor v s  = case M.lookup v s of
               Just x -> x
               Nothing -> error "Variable no definida dentro del entorno."

-- Cambia el valor de una variable en un estado
update :: Variable -> Int -> State -> State
update var i state = M.insert var i state

-- Evalua un programa en el estado nulo
eval :: Comm -> State
eval p = stepCommStar p initState

-- Evalua multiples pasos de un comnado en un estado,
-- hasta alcanzar un Skip
stepCommStar :: Comm -> State -> State
stepCommStar Skip s = s
stepCommStar c    s = Data.Strict.Tuple.uncurry stepCommStar $ stepComm c s

-- Evalua un paso de un comando en un estado dado
stepComm :: Comm -> State -> Pair Comm State
stepComm (Let var intE)                state = let (i :!: newState) = evalExp intE state
                                               in (Skip :!: (update var i newState))
stepComm (Seq comm1 comm2)             state = case (stepComm comm1 state) of
                                               (Skip :!: newState) -> (comm2 :!: newState)
                                               (c :!: newState) -> ((Seq c comm2) :!: newState)
stepComm (IfThenElse boolE comm1 comm2) state = case (evalExp boolE state) of
                                               (True :!: newState) -> (comm1 :!: newState)
                                               (False :!: newState) -> (comm2 :!: newState)
stepComm (While boolE comm)            state = case (evalExp boolE state) of
                                               (True :!: newState) -> ((Seq comm (While boolE comm)) :!: newState)
                                               (False :!: newState) -> (Skip :!: newState)
```

```haskell
-- Evalua una expresion
evalExp :: Exp a -> State -> Pair a State
evalExp (Const int) state = (int :!: state)
evalExp (Var variable)      state = (lookfor variable state :!: state)
evalExp (EAssgn var intE)   state = let (i :!: newState) = evalExp intE state
                                    in  (i :!: update var i newState)
evalExp (ESeq intE1 intE2)  state = let (i1 :!: state1) = evalExp intE1 state
                                        (i2 :!: state2) = evalExp intE2 state1
                                     in (i2 :!: state2)
evalExp (UMinus intE)       state = let (i :!: newState) = evalExp intE state
                                    in  ((-i) :!: newState)
evalExp (Plus intE1 intE2)  state = let (i1 :!: state1) = evalExp intE1 state
                                        (i2 :!: state2) = evalExp intE2 state1
                                    in  ((i1 + i2) :!: state2)
evalExp (Minus intE1 intE2) state = let (i1 :!: state1) = evalExp intE1 state
                                        (i2 :!: state2) = evalExp intE2 state1
                                    in  ((i1 - i2) :!: state2)
evalExp (Times intE1 intE2) state = let (i1 :!: state1) = evalExp intE1 state
                                        (i2 :!: state2) = evalExp intE2 state1
                                    in  ((i1 * i2) :!: state2)
evalExp (Div intE1 intE2)   state = let (i1 :!: state1) = evalExp intE1 state
                                        (i2 :!: state2) = evalExp intE2 state1
                                    in  ((div i1 i2) :!: state2)
evalExp BTrue               state = (True :!: state)
evalExp BFalse              state = (False :!: state)
evalExp (Lt intE1 intE2)    state = let (i1 :!: state1) = evalExp intE1 state
                                        (i2 :!: state2) = evalExp intE2 state1
                                    in  ((i1 < i2) :!: state2)
evalExp (Gt intE1 intE2)    state = let (i1 :!: state1) = evalExp intE1 state
                                        (i2 :!: state2) = evalExp intE2 state1
                                    in  ((i1 > i2) :!: state2)
evalExp (And boolE1 boolE2) state = let (b1 :!: state1) = evalExp boolE1 state
                                        (b2 :!: state2) = evalExp boolE2 state1
                                    in  ((b1 && b2) :!: state2)
evalExp (Or boolE1 boolE2)  state = let (b1 :!: state1) = evalExp boolE1 state
                                        (b2 :!: state2) = evalExp boolE2 state1
                                    in  ((b1 || b2) :!: state2)
evalExp (Not boolE)         state = let (b :!: newState) = evalExp boolE state
                                    in  ((not b) :!: newState)
evalExp (Eq intE1 intE2)    state = let (i1 :!: state1) = evalExp intE1 state
                                        (i2 :!: state2) = evalExp intE2 state1
                                    in  ((i1 == i2) :!: state2)
evalExp (NEq intE1 intE2)   state = let (i1 :!: state1) = evalExp intE1 state
                                        (i2 :!: state2) = evalExp intE2 state1
                                    in  ((i1 /= i2) :!: state2)
```

Ejercicio 8.

El código del archivo Eval2.hs es:

```haskell
-- Estados
type State = M.Map Variable Int

-- Estado nulo
initState :: State
initState = M.empty

-- Busca el valor de una variable en un estado
lookfor :: Variable -> State -> Either Error Int
lookfor v s  = case M.lookup v s of
               Just x -> Right x
               Nothing -> Left UndefVar

-- Cambia el valor de una variable en un estado
update :: Variable -> Int -> State -> State
update var i state = M.insert var i state

-- Evalua un programa en el estado nulo
eval :: Comm -> Either Error State
eval p = stepCommStar p initState

-- Evalua multiples pasos de un comnado en un estado,
-- hasta alcanzar un Skip
stepCommStar :: Comm -> State -> Either Error State
stepCommStar Skip s = return s
stepCommStar c    s = do
  (c' :!: s') <- stepComm c s
  stepCommStar c' s'

-- Evalua un paso de un comando en un estado dado
stepComm :: Comm -> State -> Either Error (Pair Comm State)
stepComm (Let var intE)                state = case evalExp intE state of
                                               Left error -> Left error
                                               Right (i :!: newState) -> Right (Skip :!: (update var i newState))
stepComm (Seq comm1 comm2)             state = case (stepComm comm1 state) of
                                               Left error -> Left error
                                               Right (Skip :!: newState) -> Right (comm2 :!: newState)
                                               Right (c :!: newState) -> Right ((Seq c comm2) :!: newState)
stepComm (IfThenElse boolE comm1 comm2) state = case (evalExp boolE state) of
                                               Left error -> Left error
                                               Right (True :!: newState) -> Right (comm1 :!: newState)
                                               Right (False :!: newState) -> Right (comm2 :!: newState)
stepComm (While boolE comm)            state = case (evalExp boolE state) of
                                               Left error -> Left error
                                               Right (True :!: newState) -> Right ((Seq comm (While boolE comm)) :!: newState)
                                               Right (False :!: newState) -> Right (Skip :!: newState)
```

```haskell
-- ~ Dado un posible par y una operacion unaria, devuelve un posible par
-- ~ con la operación aplicada y el estado actualizado
handleUnExpr :: Either Error (Pair a State) -> (a -> a) -> Either Error (Pair a State)
handleUnExpr (Left error) _ = Left error
handleUnExpr (Right (val :!: state)) f = Right (f val :!: state)

-- ~ Dadas dos expreciones, el estado original y una operacion binaria,
-- ~ devuelve un posible par con la operación aplicada y el estado
-- ~ actualizado
handleBinExpr :: Exp a -> Exp a -> State -> (a -> a -> b) -> Either Error (Pair b State)
handleBinExpr exp1 exp2 state f = case evalExp exp1 state of
                                  Left error -> Left error
                                  Right (i1 :!: state1) -> case evalExp exp2 state1 of
                                                           Left error -> Left error
                                                           Right (i2 :!: state2) -> Right (f i1 i2 :!: state2)

-- Evalua una expresion
evalExp :: Exp a -> State -> Either Error (Pair a State)
evalExp (Const int) state = Right (int :!: state)
evalExp (Var variable)      state = case lookfor variable state of
                                    Left error -> Left error
                                    Right i -> Right (i :!: state)
evalExp (EAssgn var intE)   state = case evalExp intE state of
                                    Left error -> Left error
                                    Right (i :!: newState) -> Right (i :!: update var i newState)
evalExp (ESeq intE1 intE2)  state = handleBinExpr intE1 intE2 state (\a b -> b)
evalExp (UMinus intE)       state = handleUnExpr (evalExp intE state) (\x -> (-x))
evalExp (Plus intE1 intE2)  state = handleBinExpr intE1 intE2 state (\a b -> a + b)
evalExp (Minus intE1 intE2) state = handleBinExpr intE1 intE2 state (\a b -> a - b)
evalExp (Times intE1 intE2) state = handleBinExpr intE1 intE2 state (\a b -> a * b)
evalExp (Div intE1 intE2)   state = case evalExp intE1 state of
                                    Left error -> Left error
                                    Right (i1 :!: state1) -> case evalExp intE2 state1 of
                                                             Left error -> Left error
                                                             Right (0 :!: state2) -> Left DivByZero
                                                             Right (i2 :!: state2) -> Right ((div i1 i2) :!: state2)
evalExp BTrue               state = Right (True :!: state)
evalExp BFalse              state = Right (False :!: state)
evalExp (Lt intE1 intE2)    state = handleBinExpr intE1 intE2 state (\a b -> a < b)
evalExp (Gt intE1 intE2)    state = handleBinExpr intE1 intE2 state (\a b -> a > b)
evalExp (And boolE1 boolE2) state = handleBinExpr boolE1 boolE2 state (\a b -> a && b)
evalExp (Or boolE1 boolE2)  state = handleBinExpr boolE1 boolE2 state (\a b -> a || b)
evalExp (Not boolE)         state = handleUnExpr (evalExp boolE state) (\a -> not a)
evalExp (Eq intE1 intE2)    state = handleBinExpr intE1 intE2 state (\a b -> a == b)
evalExp (NEq intE1 intE2)   state = handleBinExpr intE1 intE2 state (\a b -> a /= b)
```

Ejercicio 9.

Ahora State es una tupla (diccionario,entero), cuya segunda componente acumula el trabajo descrito en las ecuaciones del enunciado. El código del archivo Eval3.hs es:

```haskell
-- Estado nulo
initState :: State
initState = (M.empty, 0)

-- Busca el valor de una variable en un estado
lookfor :: Variable -> State -> Either Error Int
lookfor v (s,counter)  = case M.lookup v s of
                         Just x -> Right x
                         Nothing -> Left UndefVar

-- Cambia el valor de una variable en un estado
update :: Variable -> Int -> State -> State
update var i (state,work) = (M.insert var i state,work)

-- Suma un costo dado al estado
addWork :: Integer -> State -> State
addWork n (s,w)= (s,w+n)

-- Evalua un programa en el estado nulo
eval :: Comm -> Either Error State
eval p = stepCommStar p initState

-- Evalua multiples pasos de un comnado en un estado,
-- hasta alcanzar un Skip
stepCommStar :: Comm -> State -> Either Error State
stepCommStar Skip s = return s
stepCommStar c    s = do
  (c' :!: s') <- stepComm c s
  stepCommStar c' s'

-- Evalua un paso de un comando en un estado dado
stepComm :: Comm -> State -> Either Error (Pair Comm State)
stepComm (Let var intE)                state = case evalExp intE state of
                                               Left error -> Left error
                                               Right (i :!: newState) -> Right (Skip :!: (update var i newState))
stepComm (Seq comm1 comm2)             state = case (stepComm comm1 state) of
                                               Left error -> Left error
                                               Right (Skip :!: newState) -> Right (comm2 :!: newState)
                                               Right (c :!: newState) -> Right ((Seq c comm2) :!: newState)
stepComm (IfThenElse boolE comm1 comm2) state = case (evalExp boolE state) of
                                               Left error -> Left error
                                               Right (True :!: newState) -> Right (comm1 :!: newState)
                                               Right (False :!: newState) -> Right (comm2 :!: newState)
stepComm (While boolE comm)            state = case (evalExp boolE state) of
                                               Left error -> Left error
                                               Right (True :!: newState) -> Right ((Seq comm (While boolE comm)) :!: newState)
                                               Right (False :!: newState) -> Right (Skip :!: newState)
```

```haskell
-- ~ Dado un posible par y una operacion unaria, devuelve un posible par
-- ~ con la operación aplicada y el estado actualizado
handleUnExpr :: Either Error (Pair a State) -> (a -> a) -> Either Error (Pair a State)
handleUnExpr (Left error)           = Left error
handleUnExpr (Right (val :!: state)) f = Right (f val :!: addWork 1 state)

-- ~ Dadas dos expreciones, el estado original y una operacion binaria,
-- ~ devuelve un posible par con la operación aplicada y el estado
-- ~ actualizado
handleBinExpr :: Exp a -> Exp a -> State -> (a -> a -> b) -> Either Error (Pair b State)
handleBinExpr exp1 exp2 state f = case evalExp exp1 state of
                                  Left error -> Left error
                                  Right (i1 :!: state1) -> case evalExp exp2 state1 of
                                                           Left error -> Left error
                                                           Right (i2 :!: state2) -> Right (f i1 i2 :!: addWork 1 state2)

-- Evalua una expresion y agrega el costo de su opéración correspondiente
evalExp :: Exp a -> State -> Either Error (Pair a State)
evalExp (Const int)         state = Right (int :!: state)
evalExp (Var variable)      state = case lookfor variable state of
                                    Left error -> Left error
                                    Right i -> Right (i :!: state)
evalExp (EAssgn var intE)   state = case evalExp intE state of
                                    Left error -> Left error
                                    Right (i :!: newState) -> Right (i :!: update var i newState)
evalExp (ESeq intE1 intE2)  state = handleBinExpr intE1 intE2 state (\a b -> b)
evalExp (UMinus intE)       state = handleUnExpr (evalExp intE state) (\x -> (-x))
evalExp (Plus intE1 intE2)  state = handleBinExpr intE1 intE2 state (\a b -> a + b)
evalExp (Minus intE1 intE2) state = handleBinExpr intE1 intE2 state (\a b -> a - b)
evalExp (Times intE1 intE2) state = handleBinExpr intE1 intE2 (addWork 1 state) (\a b -> a * b)
evalExp (Div intE1 intE2)   state = case evalExp intE1 state of
                                    Left error -> Left error
                                    Right (i1 :!: state1) -> case evalExp intE2 state1 of
                                                             Left error -> Left error
                                                             Right (0 :!: state2) -> Left DivByZero
                                                             Right (i2 :!: state2) -> Right ((div i1 i2) :!: addWork 2 state2)
evalExp BTrue               state = Right (True :!: state)
evalExp BFalse              state = Right (False :!: state)
evalExp (Lt intE1 intE2)    state = handleBinExpr intE1 intE2 state (\a b -> a < b)
evalExp (Gt intE1 intE2)    state = handleBinExpr intE1 intE2 state (\a b -> a > b)
evalExp (And boolE1 boolE2) state = handleBinExpr boolE1 boolE2 state (\a b -> a && b)
evalExp (Or boolE1 boolE2)  state = handleBinExpr boolE1 boolE2 state (\a b -> a || b)
evalExp (Not boolE)         state = handleUnExpr (evalExp boolE state) (\a -> not a)
evalExp (Eq intE1 intE2)    state = handleBinExpr intE1 intE2 state (\a b -> a == b)
evalExp (NEq intE1 intE2)   state = handleBinExpr intE1 intE2 state (\a b -> a /= b)
```

Ejercicio 10.
Agregamos la siguiente regla de producción a la gramática abstracta de LIS y extendemos la semántica operacional de comandos para el comando for:

- Sintaxis abstracta

  $comm$ ::= $skip$
  | $var = intexp$
  | $comm$; $comm$
  | $if\ boolexp\ then\ comm\ else\ comm$
  | $while\ boolexp\ do\ comm$
  | $for\ intexp\ boolexp\ intexp\ comm$

- Semántica operacional estructural para comandos

$$\frac{\langle e_1, \sigma\rangle \Downarrow_{intexp} \langle n, \sigma'\rangle}{\langle \textbf{for}\ e_1\ b\ e_2\ c, \sigma\rangle \rightsquigarrow \langle e_1;\ \textbf{for}\ n\ b\ e_2\ c, \sigma'\rangle}\ FOR1$$

$$\frac{\langle b, \sigma\rangle \Downarrow_{boolexp} \langle \textbf{true}, \sigma'\rangle \quad \langle e, \sigma'\rangle \rightsquigarrow \langle n_1, \sigma''\rangle}{\langle \textbf{for}\ n_2\ b\ e\ c, \sigma\rangle \rightsquigarrow \langle c;\ \textbf{for}\ n_2\ b\ e\ c, \sigma''\rangle}\ FOR2$$

$$\frac{\langle b, \sigma\rangle \Downarrow_{boolexp} \langle \textbf{false}, \sigma'\rangle}{\langle \textbf{for}\ n\ b\ e\ c, \sigma\rangle \rightsquigarrow \langle \textbf{skip}, \sigma'\rangle}\ FOR3$$