Universidad Nacional de Rosario

Facultad de Ciencias Exactas, Ingeniería y Agrimensura

Especificación del Lenguaje Imperativo Simple

Primer trabajo practico
Análisis del Lenguajes de Programación

Autor:
Caporalini, Joaquín
Arroyo, Joaquín

16 de septiembre de 2022

Índice

1. Ejercicio 1

Extensión de la sintáxis abstracta con la regla de producción del operador ternario:

```
intexp      ::= nat
              | var
              | -_u intexp
              | intexp + intexp
              | intexp -_b intexp
              | intexp * intexp
              | intexp / intexp
              | boolexpatom ? intexp : intexp
boolexpatom ::= true
                | false
                | ¬boolexpatom
                | boolexp
boolexp     ::= intexp == intexp
              | intexp != intexp
              | intexp < intexp
              | intexp > intexp
              | intexp ∧ intexp
              | intexp ∨ intexp
              | boolexpatom
comm        ::=  skip
              | var = intexp
              | comm; comm
              | if boolexp then comm else comm
              | while boolexp do comm
```

Extensión de la sintáxis concreta con la regla de producción del operador ternario:

```
digit       ::= '0' | '1' | ... | '9'
letter      ::= 'a' | ... | 'z'
nat         ::= digit | digit nat
var         ::= letter | letter var
intexp      ::= nat
              | var
              | '-' intexp
              | intexp '+' intexp
              | intexp '-' intexp
              | intexp '*' intexp
              | intexp '/' intexp
              | boolexpatom '?' intexp ':' intexp
              | '(' intexp ')'
boolexpatom ::= true
                | false
                | '!' boolexpatom
                | '(' boolexp ')'
boolexp     ::= intexp '==' intexp
              | intexp '!=' intexp
              | intexp '<' intexp
              | intexp '>' intexp
              | intexp '&&' intexp
              | intexp '||' intexp
              | boolexpatom
```

```
comm    ::=  'skip'
           | var '=' intexp
           | comm ';' comm
           | 'if' boolexp '{' comm '}'
           | 'if' boolexp '{' comm '}' 'else' '{' comm '}'
           | 'while' boolexp '{' comm '}'
```

2. Ejercicio 4

Extensión de la semántica big-step de expresiones enteras con las reglas para el operador ternario:

$$\frac{\langle p_0,\sigma\rangle \Downarrow_{exp} \textbf{true} \quad \langle e_0,\sigma\rangle \Downarrow_{exp} n_0}{\langle p_0 ? e_0 : e_1,\sigma\rangle \Downarrow_{exp} \mathrm{n}_0} \text{?TRUE}$$

$$\frac{\langle p_0,\sigma\rangle \Downarrow_{exp} \textbf{false} \quad \langle e_1,\sigma\rangle \Downarrow_{exp} n_1}{\langle p_0 ? e_0 : e_1,\sigma\rangle \Downarrow_{exp} n_1} \text{?FALSE}$$

3. Ejercicio 5

Queremos probar que la relación $\rightsquigarrow$ es determinista, es decir que, si t $\rightsquigarrow$ v1 y t $\rightsquigarrow$ v2 $\Rightarrow$ v1 = v2. Vamos a realizar la demostración bajo el supuesto de que $\Downarrow_{exp}$ es determinista.

- Si la última derivación fue ASS entonces
  - 1. $\langle e, \sigma\rangle \Downarrow_{exp} n$
  - 2. t = $\langle v := e, \sigma\rangle$
  - 3. v1 = $\langle skip,\ [\sigma|\text{v:n}]\rangle$

  Como $\Downarrow_{exp}$ es determinista por hipótesis, y la última regla que podemos aplicarle a t es ASS, v1 = v2.

- Si la última derivación fue SEQ1 entonces
  - 1. t = $\langle \textbf{skip}; c1, \sigma\rangle$
  - 2. v1 = $\langle c1, \sigma\rangle$

  Como la última regla que podemos aplicarle a t es SEQ1, v1 = v2.

- Si la última derivación fue SEQ2 entonces
  - 1. $\langle c0,\sigma\rangle \rightsquigarrow \langle c0', \sigma\text{'}\rangle$
  - 2. t = $\langle c0; c1, \sigma\rangle$
  - 3. v1 = $\langle c0'; c1, \sigma\text{'}\rangle$

  Como la última regla que podemos aplicarle a t es SEQ2 y por H.I $\langle c0, \sigma\rangle \rightsquigarrow \langle c0', \sigma'\rangle$ es determinista, no es posible aplicar SEQ2 con un antecedente diferente, por lo que v1 = v2.

- Si la última derivación fue IF1 entonces
  - 1. $\langle b, \sigma\rangle \Downarrow_{exp} true$
  - 2. t = $\langle \textbf{if}\ b\ \textbf{then}\ c0\ \textbf{else}\ c1, \sigma\rangle$
  - 3. v1 = $\langle c0, \sigma\rangle$

  Como $\Downarrow_{exp}$ es determinista por hipótesis, y la última regla que podemos aplicarle a t es IF1, v1 = v2.

- Análogo para IF2.

- Si la última derivación fue WHILE1 entonces
  - 1. $\langle b, \sigma \rangle \Downarrow_{exp} true$
  - 2. t = $\langle$**while** b **do** c, $\sigma\rangle$
  - 3. v1 = $\langle c;$**while** b **do** c, $\sigma\rangle$

  Como $\Downarrow_{exp}$ es determinista por hipótesis, y la última regla que podemos aplicarle a t es WHILE1, v1 = v2.

- Si la última derivación fue WHILE2 entonces
  - 1. $\langle b, \sigma \rangle \Downarrow_{exp} false$
  - 2. t = $\langle$**while** b **do** c, $\sigma\rangle$
  - 3. v1 = $\langle$**skip**, $\sigma\rangle$

  Como $\Downarrow_{exp}$ es determinista por hipótesis, y la última regla que podemos aplicarle a t es WHILE2, v1 = v2.

4. Ejercicio 6

4.1. A

$$
\dfrac{\dfrac{\dfrac{}{\langle x, [[\sigma|x:2\}|y:2\}\rangle \Downarrow_{exp} 2}\,\textbf{VAR} \quad \dfrac{}{\langle 0, [[\sigma|x:2\}|y:2\}\rangle \Downarrow_{exp} 0}\,\textbf{NVAL}}{\langle x>0, [[\sigma|x:2]|y:2]\rangle \Downarrow_{exp} 2>0}\,\textbf{LT}}{\dfrac{\langle \textbf{while } x>0 \textbf{ do } x:=x-y, [[\sigma|x:2]|y:2]\rangle \rightsquigarrow \langle x:=x-y; \textbf{while } x>0 \textbf{ do } x:=x-y, [[\sigma|x:2]|y:2]\rangle}{\langle \textbf{while } x>0 \textbf{ do } x:=x-y, [[\sigma|x:2]|y:2]\rangle \rightsquigarrow^* \langle x:=x-y; \textbf{while } x>0 \textbf{ do } x:=x-y, [[\sigma|x:2]|y:2]\rangle}\,\textbf{CLAUSURE}}\,\textbf{WHILE1}
$$

4.2. B

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\langle x, [[\sigma|x:2\}|y:2\}\rangle \Downarrow_{exp} 2}\,\textbf{VAR} \quad \dfrac{}{\langle y, [[\sigma|x:2\}|y:2\}\rangle \Downarrow_{exp} 2}\,\textbf{VAR}}{\langle x-y, [[\sigma|x:2]|y:2]\rangle \Downarrow_{exp} 0}\,\boldsymbol{MINUS}}{\langle x:=x-y, [[\sigma|x:2]|y:2]\rangle \rightsquigarrow \langle \textbf{skip}, [[\sigma|x:0]|y:2]\rangle}\,\textbf{ASS}}{\langle x:=x-y; \textbf{while } x>0 \textbf{ do } x:=x-y, [[\sigma|x:2]|y:2]\rangle \rightsquigarrow \langle \textbf{skip}; \textbf{while } x>0 \textbf{ do } x:=x-y, [[\sigma|x:0]|y:2]\rangle}\,\textbf{SEQ2}}{\langle x:=x-y; \textbf{while } x>0 \textbf{ do } x:=x-y, [[\sigma|x:2]|y:2]\rangle \rightsquigarrow^* \langle \textbf{skip}; \textbf{while } x>0 \textbf{ do } x:=x-y, [[\sigma|x:0]|y:2]\rangle}\,\textbf{CLAUSURE}
$$

4.3. C

$$
\dfrac{\textbf{A} \qquad\qquad \textbf{B}}{\langle \textbf{while } x>0 \textbf{ do } x:=x-y, [[\sigma|x:2]|y:2]\rangle \rightsquigarrow^* \langle \textbf{skip}; \textbf{while } x>0 \textbf{ do } x:=x-y, [[\sigma|x:0]|y:2]\rangle}\,\textbf{TRANSITIVE}
$$

4.4. D

$$
\dfrac{\dfrac{}{\langle \textbf{skip}; \textbf{while } x>0 \textbf{ do } x:=x-y, [[\sigma|x:0\}|y:2\}\rangle \rightsquigarrow \langle \textbf{while } x>0 \textbf{ do } x:=x-y, [[\sigma|x:0\}|y:2\}\rangle}\,\textbf{SEQ1}}{\langle \textbf{skip}; \textbf{while } x>0 \textbf{ do } x:=x-y, [[\sigma|x:0\}|y:2\}\rangle \rightsquigarrow^* \langle \textbf{while } x>0 \textbf{ do } x:=x-y, [[\sigma|x:0\}|y:2\}\rangle}\,\textbf{CLAUSURE}
$$

4.5. E

$$
\dfrac{\textbf{C} \qquad\qquad \textbf{D}}{\langle \textbf{while } x>0 \textbf{ do } x:=x-y, [[\sigma|x:2]|y:2]\rangle \rightsquigarrow^* \langle \textbf{while } x>0 \textbf{ do } x:=x-y, [[\sigma|x:0\}|y:2\}\rangle}\,\textbf{TRANSITIVE}
$$

4.6. F

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\langle x, [[\sigma|x:0]|y:2]\rangle \Downarrow_{exp} 0}\;\textbf{VAR} \qquad \dfrac{}{\langle y, [[\sigma|x:2]|y:2]\rangle \Downarrow_{exp} 2}\;\textbf{VAR}}{\langle x > 0, [[\sigma|x:0]|y:2]]\rangle \Downarrow_{exp} 0 > 0}\;\textbf{LT}}{\langle \textbf{while } x > 0 \textbf{ do } x := x - y, [[\sigma|x:0]|y:2]]\rangle \rightsquigarrow \langle \textbf{skip}, [[\sigma|x:0]|y:2]\rangle}\;\textbf{WHILE2}}{\langle \textbf{while } x > 0 \textbf{ do } x := x - y, [[\sigma|x:0\}|y:2]]\rangle \rightsquigarrow^* \langle \textbf{skip}, [[\sigma|x:0]|y:2]\rangle}\;\textbf{CLAUSURE}$$

4.7. G

$$\dfrac{\textbf{E} \qquad\qquad \textbf{F}}{\langle \textbf{while } x > 0 \textbf{ do } x := x - y, [[\sigma|x:2]|y:2]\rangle \rightsquigarrow^* \langle \textbf{skip}, [[\sigma|x:0]|y:2]\rangle}\;\textbf{TRANSITIVE}$$

4.8. H

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\langle x, [[\sigma|x:1]|y:2]\rangle \Downarrow_{exp} 1}\;\textbf{VAR} \qquad \dfrac{}{\langle y, [[\sigma|x:1]|y:2]\rangle \Downarrow_{exp} 2}\;\textbf{VAR}}{\langle x := x > y \;?\; y*2 \;:\; y, [[\sigma|x:1]|y:2]\rangle \Downarrow_{exp} 1 > 2}\;\textbf{LT} \qquad \dfrac{}{\langle y, [[\sigma|x:1]|y:2]\rangle \Downarrow_{exp} 2}\;\textbf{VAR}}{\langle x := x > y \;?\; y*2 \;:\; y, [[\sigma|x:1]|y:2]\rangle \Downarrow_{exp} 2}\;\textbf{?FALSE}}{\langle x := x > y \;?\; y*2 \;:\; y, [[\sigma|x:1]|y:2]\rangle \rightsquigarrow \langle \textbf{skip}, [[\sigma|x:2]|y:2]\rangle}\;\textbf{ASS}}{\langle x := x > y \;?\; y*2 \;:\; y; \textbf{while } x > 0 \textbf{ do } x := x - y, [[\sigma|x:1]|y:2]\rangle \rightsquigarrow \langle \textbf{skip}; \textbf{while } x > 0 \textbf{ do } x := x - y, [[\sigma|x:2]|y:2]\rangle}\;\textbf{SEQ2}}{\langle x := x > y \;?\; y*2 \;:\; y; \textbf{while } x > 0 \textbf{ do } x := x - y, [[\sigma|x:1]|y:2]\rangle \rightsquigarrow^* \langle \textbf{skip}; \textbf{while } x > 0 \textbf{ do } x := x - y, [[\sigma|x:2]|y:2]\rangle}\;\textbf{CLAUSURE}$$

4.9. I

$$\dfrac{\dfrac{}{\langle \textbf{skip}; \textbf{while } x > 0 \textbf{ do } x := x - y, [[\sigma|x:2]|y:2]\rangle \rightsquigarrow \langle \textbf{while } x > 0 \textbf{ do } x := x - y, [[\sigma|x:2]|y:2]\rangle}\;\textbf{SEQ1}}{\langle \textbf{skip}; \textbf{while } x > 0 \textbf{ do } x := x - y, [[\sigma|x:2]|y:2]\rangle \rightsquigarrow^* \langle \textbf{while } x > 0 \textbf{ do } x := x - y, [[\sigma|x:2]|y:2]\rangle}\;\textbf{CLAUSURE}$$

4.10. J

$$\dfrac{\textbf{H} \qquad\qquad \textbf{I}}{\langle x := x > y \;?\; y*2 \;:\; y; \textbf{while } x > 0 \textbf{ do } x := x - y, [[\sigma|x:1]|y:2]\rangle \rightsquigarrow^* \langle \textbf{while } x > 0 \textbf{ do } x := x - y, [[\sigma|x:2]|y:2]\rangle}\;\textbf{TRANSITIVE}$$

4.11. K

$$\frac{\mathbf{J} \qquad\qquad\qquad\qquad \mathbf{G}}{\langle x := x > y\ ?\ y * 2\ :\ y; \mathbf{while}\ x > 0\ \mathbf{do}\ x := x - y, [[\sigma|x:1]|y:2]\rangle \rightsquigarrow^* \langle \mathbf{skip}, [[\sigma|x:0]|y:2]\rangle}\ \mathbf{TRANSITIVE}$$

K es la ultima parte del árbol

5. Ejercicio 10

Extensión de la sintáxis abstracta con la regla de producción del comando repeat:

comm ::= ... | **repeat** comm **until** boolexp

Extensión de la semantica con la regla para el comando repeat:

$$\frac{}{\langle \mathbf{repeat}\ c\ \mathbf{until}\ b, \sigma\rangle \rightsquigarrow \langle c; \mathbf{while}\ \neg b\ \mathbf{do}\ c, \sigma\rangle}\ \text{REPEAT}$$

```haskell
module Parser where

import             Text.ParserCombinators.Parsec
import             Text.Parsec.Token
import             Text.Parsec.Language           ( emptyDef )
import             AST

-----------------------
-- Funcion para facilitar el testing del parser.
totParser :: Parser a -> Parser a
totParser p = do
  whiteSpace lis
  t <- p
  eof
  return t

-- Analizador de Tokens
lis :: TokenParser u
lis = makeTokenParser
  (emptyDef
    { commentStart    = "/*"
    , commentEnd      = "*/"
    , commentLine     = "//"
    , opLetter        = char '='
    , reservedNames   = ["true", "false", "if", "else", "while", "skip" , "do", "for"]
    , reservedOpNames = [ "+"
                        , "-"
                        , "*"
                        , "/"
                        , "<"
                        , ">"
                        , "&&"
                        , "||"
                        , "!"
                        , "="
                        , "=="
                        , "!="
                        , ";"
                        , ","
                        , "?"
                        , ":"
                        ]
    }
  )

-----------------------------------
-- Parser de expressiones enteras
-----------------------------------

addMinusOp = do {reservedOp lis "+"; return Plus}
         <|> do {reservedOp lis "-"; return Minus}

timesDivOp = do {reservedOp lis "*"; return Times}
        <|> do {reservedOp lis "/"; return Div}

intexp :: Parser (Exp Int)
intexp = ternParser

ternParser :: Parser (Exp Int)
ternParser = try (do b <- boolAtomParser
                     reservedOp lis "?"
```

```haskell
                     i1 <- ternParser
                     reservedOp lis ":"
                     ECond b i1 <$> ternParser)
             <|>
             addMinusParser

addMinusParser :: Parser (Exp Int)
addMinusParser = timesDivParser `chainl1` addMinusOp

timesDivParser :: Parser (Exp Int)
timesDivParser  = intFactorParser  `chainl1` timesDivOp

intFactorParser :: Parser (Exp Int)
intFactorParser = parens lis intexp
                  <|>
                  try (do n <- natural lis
                          return (Const (fromIntegral n)))
                  <|>
                  try (do i <- identifier lis
                          return (Var i))
                  <|>
                  uMinusParser

uMinusParser :: Parser (Exp Int)
uMinusParser = do reservedOp lis "-"
                  UMinus <$> intFactorParser

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

boolCompOp = do {reservedOp lis "==" ; return Eq}
         <|> do {reservedOp lis "!=" ; return NEq}
         <|> do {reservedOp lis "<" ; return Lt}
         <|> do {reservedOp lis ">" ; return Gt}

boolOp = do {reservedOp lis "&&" ; return And}
     <|> do {reservedOp lis "||" ; return Or}

boolexp :: Parser (Exp Bool)
boolexp = andOrParser

andOrParser :: Parser (Exp Bool)
andOrParser = boolCompParser `chainl1` boolOp

boolCompParser :: Parser (Exp Bool)
boolCompParser = try (do i1 <- intexp
                         b <- boolCompOp
                         b i1 <$> intexp)
                 <|>
                 boolAtomParser

boolAtomParser :: Parser (Exp Bool)
boolAtomParser = try (parens lis boolexp)
                 <|>
                 do reserved lis "true"
                    return BTrue
                 <|>
                 do reserved lis "false"
                    return BFalse
                 <|>
```

```haskell
                  do reservedOp lis "!"
                     Not <$> boolAtomParser

-----------------------------------
-- Parser de comandos
-----------------------------------

semicolon = do {reservedOp lis ";" ; return Seq}

comm :: Parser Comm
comm = (skipParser
        <|> try letParser
        <|> ifThenParser
        <|> whileParser)
       `chainr1` semicolon

skipParser :: Parser Comm
skipParser = do reserved lis "skip"
                return Skip

letParser :: Parser Comm
letParser = do v <- identifier lis
               reservedOp lis "="
               Let v <$> intexp

ifThenParser :: Parser Comm
ifThenParser = do reserved lis "if"
                  b <- boolexp
                  c1 <- braces lis comm
                  IfThenElse b c1 <$> elseParser

elseParser :: Parser Comm
elseParser = do reserved lis "else"
                braces lis comm
             <|>
             return Skip

whileParser :: Parser Comm
whileParser = do reserved lis "while"
                 b <- boolexp
                 c <- braces lis comm
                 return (While b c)

------------------------------------
-- Función de parseo
------------------------------------
parseComm :: SourceName -> String -> Either ParseError Comm
parseComm = parse (totParser comm)
```

```haskell
module Eval1
  ( eval
  , State
  )
where

import           AST
import qualified Data.Map.Strict               as M
import           Data.Strict.Tuple

-- 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

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

-- 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 v x) s = Skip :!: (update v (evalExp x s) s)
stepComm (IfThenElse b c1 c2) s = if evalExp b s
                                  then c1 :!: s
                                  else c2 :!: s
stepComm (While b c) s = if evalExp b s
                         then Seq c (While b c) :!: s
                         else Skip :!: s
stepComm (Seq Skip c) s = c :!: s
stepComm (Seq c cs) s = let newC :!: newS = stepComm c s
                        in stepComm (Seq newC cs) newS

-- Evalua una expresion
evalExp :: Exp a -> State -> a
evalExp (Const x) s = x
evalExp (Var v) s = lookfor v s
evalExp (UMinus e) s = -(evalExp e s)
evalExp (Plus x y) s = evalExp x s + evalExp y s
evalExp (Minus x y) s = evalExp x s - evalExp y s
evalExp (Times x y) s = evalExp x s * evalExp y s
evalExp (Div x y) s = evalExp x s `div` evalExp y s
evalExp (ECond b x y) s = if evalExp b s
                          then evalExp x s
                          else evalExp y s
```

```
evalExp BTrue s = True
evalExp BFalse s = False
evalExp (Lt x y) s = evalExp x s < evalExp y s
evalExp (Gt x y) s = evalExp x s > evalExp y s
evalExp (Eq x y) s = evalExp x s == evalExp y s
evalExp (NEq x y) s = evalExp x s /= evalExp y s
evalExp (And x y) s = evalExp x s && evalExp y s
evalExp (Or x y) s = evalExp x s || evalExp y s
evalExp (Not b) s = not (evalExp b s)
```

```haskell
module Eval2
  ( eval
  , State
  )
where

import           AST
import qualified Data.Map.Strict               as M
import           Data.Strict.Tuple

-- 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
                   _  -> Left UndefVar

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

-- 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 v x) s = case evalExp x s of
                        Right n -> Right (Skip :!: update v n s)
                        Left e -> Left e
stepComm (IfThenElse b c1 c2) s = case evalExp b s of
                                    Right True -> Right (c1 :!: s)
                                    Right False -> Right (c2 :!: s)
                                    Left e -> Left e
stepComm (While b c) s = case evalExp b s of
                          Right True -> Right (Seq c (While b c) :!: s)
                          Right False -> Right (Skip :!: s)
                          Left e -> Left e
stepComm (Seq Skip c) s = Right (c :!: s)
stepComm (Seq c cs) s = case stepComm c s of
                          Right (c' :!: s') -> stepComm (Seq c' cs) s'
                          Left e -> Left e

-- Evalua una expresion
evalExp :: Exp a -> State -> Either Error a
evalExp (Const x) s = Right x
evalExp (Var v) s = case lookfor v s of
```

```
                      Right x -> Right x
                      Left e -> Left e
evalExp (UMinus e) s = case evalExp e s of
                        Right x -> Right (-x)
                        Left e -> Left e
evalExp (Plus x y) s = evalExp' x y s (+)
evalExp (Minus x y) s = evalExp' x y s (-)
evalExp (Times x y) s = evalExp' x y s (*)
evalExp (Div x y) s = case evalExp x s of
                        Right x' ->  case evalExp y s of
                                      Right 0  -> Left DivByZero
                                      Right y' -> Right (x' `div` y')
                                      Left e1  -> Left e1
                        Left e2 -> Left e2
evalExp (ECond b x y) s = case evalExp b s of
                            Right True -> evalExp x s
                            Right False -> evalExp y s
                            Left e -> Left e
evalExp BTrue s = Right True
evalExp BFalse s = Right False
evalExp (Lt x y) s = evalExp' x y s (<)
evalExp (Gt x y) s = evalExp' x y s (>)
evalExp (Eq x y) s = evalExp' x y s (==)
evalExp (NEq x y) s = evalExp' x y s (/=)
evalExp (And x y) s = evalExp' x y s (&&)
evalExp (Or x y) s = evalExp' x y s (||)
evalExp (Not b) s = case evalExp b s of
                      Right x -> Right x
                      Left e -> Left e

-- Evaluador auxiliar de expresiones, recibe un operador y se lo aplica
-- a las dos expresiones recibidas.
evalExp' :: Exp a -> Exp a -> State -> (a -> a -> b) -> Either Error b
evalExp' x y s op = case evalExp x s of
                      Right x' ->  case evalExp y s of
                                    Right y' -> Right (op x' y')
                                    Left e1 -> Left e1
                      Left e2 -> Left e2
```

```haskell
module Eval3
  ( eval
  , State
  )
where

import           AST
import qualified Data.Map.Strict               as M
import           Data.Strict.Tuple

-- Estados
type State = (M.Map Variable Int, Integer)

-- 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, _) = case M.lookup v s of
                    Just x -> Right x
                    _ -> Left UndefVar

-- Cambia el valor de una variable en un estado
update :: Variable -> Int -> State -> State
update x v (s, w) = (M.insert x v s, w)

-- 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 v x) s = case evalExp x s of
                          Right (n :!: s') -> Right (Skip :!: (update v n s'))
                          Left e -> Left e
stepComm (IfThenElse b c1 c2) s = case evalExp b s of
                                    Right (True :!: s') -> Right (c1 :!: s')
                                    Right (False :!: s') -> Right (c2 :!: s')
                                    Left e -> Left e
stepComm (While b c) s = case evalExp b s of
                          Right (True :!: s') -> Right (Seq c (While b c) :!: s')
                          Right (False :!: s') -> Right (Skip :!: s')
                          Left e -> Left e
stepComm (Seq Skip c) s = Right (c :!: s)
stepComm (Seq c cs) s = case stepComm c s of
                          Right (c' :!: s') -> stepComm (Seq c' cs) s'
                          Left e -> Left e

```

```haskell
-- Evalua una expresion
evalExp :: Exp a -> State -> Either Error (Pair a State)
evalExp (Const x) s = Right (x :!: (addWork 0 s))
evalExp (Var v) s = case lookfor v s of
                      Right x -> Right (x :!: (addWork 0 s))
                      Left e -> Left e
evalExp (UMinus e) s = case evalExp e s of
                        Right (x :!: s') -> Right (-x :!: (addWork 1 s'))
                        Left e -> Left e
evalExp (Plus x y) s = evalExp' x y s 2 (+)
evalExp (Minus x y) s = evalExp' x y s 2 (-)
evalExp (Times x y) s = evalExp' x y s 3 (*)
evalExp (Div x y) s = case evalExp x s of
                        Right (x' :!: s1) ->  case evalExp y s1 of
                                                Right (0 :!: _) -> Left DivByZero
                                                Right (y' :!: s2) -> Right (x' `div` y' :!: (addWork 3 s2))
                                                Left e1 -> Left e1
                        Left e2 -> Left e2
evalExp (ECond b x y) s = case evalExp b s of
                            Right (True :!: s') -> evalExp x (addWork 1 s')
                            Right (False :!: s') -> evalExp y (addWork 1 s')
                            Left e -> Left e
evalExp BTrue s = Right (True :!: s)
evalExp BFalse s = Right (False :!: s)
evalExp (Lt x y) s = evalExp' x y s 2 (<)
evalExp (Gt x y) s = evalExp' x y s 2 (>)
evalExp (Eq x y) s = evalExp' x y s 2 (==)
evalExp (NEq x y) s = evalExp' x y s 2 (/=)
evalExp (And x y) s = evalExp' x y s 2 (&&)
evalExp (Or x y) s = evalExp' x y s 2 (||)
evalExp (Not b) s = case evalExp b s of
                      Right (x :!: s') -> Right (x :!: (addWork 1 s'))
                      Left e -> Left e

-- Evaluador auxiliar de expresiones, ademas recibe el trabajo de aplicarle el operador
-- a las dos expresiones recibidas y lo agrega al estado.
evalExp' :: Exp a -> Exp a -> State -> Integer -> (a -> a -> b) -> Either Error (Pair b State)
evalExp' x y s w op = case evalExp x s of
                        Right (x' :!: s1) -> case evalExp y s1 of
                                               Right (y' :!: s2) -> Right ((op x' y') :!: (addWork w s2))
                                               Left e1 -> Left e1
                        Left e2 -> Left e2
```