Universidad Nacional de Rosario

Facultad de ciencias exactas, ingeniería y agrimensura

Análisis de Lenguajes de Programación

Trabajo práctico 1

Alumnos:

Petruskevicius Ignacio - Trincheri Lucio

Julio 2021

ALP - Informe TP 1

Petruskevicius Ignacio - Trincheri Lucio, LCC UNR FCEIA 23/07/2021

1. Ejercicio 1

Para poder realizar el parser correctamente decidimos desambiguar la gramática, lo que ayudo en el desarrollo de las nuevas operaciones de asignación y el operador ",".

1.1. Sintaxis abstracta

Listing 1: Sintaxis abstracta

```
intexp ::= nat | var | -u intexp
        | intexp + intexp
        | intexp -b intexp
        | intexp * intexp
        | intexp / intexp
        | intexp ',' intexp
        | var = equal'
equal' ::= var = equal'
        | intexp'
intexp' ::= nat | var | -u intexp'
        | intexp' + intexp'
        | intexp' -b intexp'
        | intexp' * intexp'
        | intexp' / intexp'
        | intexp' ',' intexp'
boolexp ::= true | false
        | intexp == intexp
        | intexp != intexp
        | intexp < intexp
        | intexp > intexp
        | boolexp && boolexp
        | boolexp || boolexp
        | !boolexp
comm ::= skip
        | var = intexp
        | comm; comm
        | if boolexp then comm else comm
        | repeat comm until boolexp
```

1.2. Sintaxis concreta

Listing 2: Sintaxis concreta

```
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
        | '(' intexp ')'
        | intexp ',' intexp
        | var '=' equal'
equal' ::= var '=' equal'
        | intexp'
intexp' ::= nat | var | -u intexp'
        | intexp' '+' intexp'
        | intexp' '-' intexp'
        | intexp' '*' intexp'
        | intexp' '/' intexp'
        | intexp' ',' intexp'
        | '(' intexp ')'
boolexp ::= 'true' | 'false'
        | intexp '==' intexp
        | intexp '!=' intexp
        | intexp '<' intexp
        | intexp '>' intexp
        | boolexp '&&' boolexp
        | boolexp '||' boolexp
        | '!' boolexp
        | '(' boolexp ')'
comm ::= skip
        | var '=' intexp
        | comm ';' comm
        | 'if' boolexp '{' comm '}'
        | 'if' boolexp '{' comm '}' 'else' '{' comm '}'
        | 'repeat' comm 'until' boolexp 'end'
```

2. Ejercicio 2

Listing 3: Sintaxis abstracta Haskell

```haskell
module AST where

-- Identificadores de Variable
type Variable = String

-- Expresiones, aritmeticas y booleanas
data Exp a where
  -- Expresiones enteras
  Const :: Int -> Exp Int
  Var :: Variable -> 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
  ESeq :: Exp Int -> Exp Int -> Exp Int
  EAssgn :: Variable -> Exp Int -> Exp Int
  -- Expreseiones 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

deriving instance Show (Exp a)
deriving instance Eq (Exp a)

-- Comandos (sentencias)
-- Observar que solo se permiten variables de un tipo (entero)
data Comm
  = Skip
  | Let Variable (Exp Int)
  | Seq Comm Comm
  | IfThenElse (Exp Bool) Comm Comm
  | Repeat Comm (Exp Bool)
  deriving (Show, Eq)

pattern IfThen :: Exp Bool -> Comm -> Comm
pattern IfThen b c = IfThenElse b c Skip

data Error = DivByZero | UndefVar deriving (Eq, Show)
```

3. Ejercicio 3

Listing 4: Parser Haskell

```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", "repeat", "skip", "until"]
    , reservedOpNames = [ "+"
                        , "-"
                        , "*"
                        , "/"
                        , "<"
                        , ">"
                        , "&&"
                        , "||"
                        , "!"
                        , "="
                        , "=="
                        , "!="
                        , ";"
                        , ","
                        ]
    }
  )


x <||> y = (try x) <|> y
---------------------------------
```

```haskell
--- Parser de expressiones enteras
-----------------------------------
intexp :: Parser (Exp Int)
intexp = do chainl1 equalIntexp commasepIntexp
intexpOperators :: Parser (Exp Int)
intexpOperators = do chainl1 compTerm addsubIntexp <|> (do parens lis $ intexp)

compTerm :: Parser (Exp Int)
compTerm = do chainl1 (parens lis intexp <|> do { x <- natural lis ; return (↩
    Const (fromInteger x))} <|> do { x <- identifier lis ; return (Var x)} <|> ↩
    unarynegationIntexp) timesdivIntexp

commasepIntexp :: Parser (Exp Int -> Exp Int -> Exp Int)
commasepIntexp = do
                    reservedOp lis ","
                    return (ESeq)

equalIntexp :: Parser (Exp Int)
equalIntexp = do
                nombre <- identifier lis
                reservedOp lis "="
                x <- equalIntexp
                return (EAssgn nombre x)
                <|> intexpOperators

addsubIntexp :: Parser (Exp Int -> Exp Int -> Exp Int)
addsubIntexp = do
                 reservedOp lis "+"
                 return (Plus)
               <|> do
                 reservedOp lis "-"
                 return (Minus)
timesdivIntexp :: Parser (Exp Int -> Exp Int -> Exp Int)
timesdivIntexp = do
                    reservedOp lis "*"
                    return (Times)
                 <|> do
                    reservedOp lis "/"
                    return (Div)

unarynegationIntexp = do
                        reservedOp lis "-"
                        n <- intexp
                        return (UMinus n)
-----------------------------------
--- Parser de expressiones booleanas
-----------------------------------

boolexp :: Parser (Exp Bool)
boolexp = chainl1 (parens lis boolexp <|> do notBoolexp) orandBoolexp

```

```haskell
orandBoolexp :: Parser (Exp Bool -> Exp Bool -> Exp Bool)
orandBoolexp =  do
                  reservedOp lis "||"
                  return (Or)
                <|> do
                  reservedOp lis "&&"
                  return (And)

notBoolexp :: Parser (Exp Bool)
notBoolexp =
    do
      reservedOp lis "!"
      x <- parens lis boolexp <|> do caseBoolexp
      return (Not x)
    <||> do
      x <- caseBoolexp
      return (x)

caseBoolexp :: Parser (Exp Bool)
caseBoolexp =
    -- First case: comparators
    do
      termino1 <- intexp
      (do
         reservedOp lis ">"
         termino2 <- intexp
         return (Gt termino1 termino2)
       <|> do
         reservedOp lis "<"
         termino2 <- intexp
         return (Lt termino1 termino2)
       <|> do
         reservedOp lis "!="
         termino2 <- intexp
         return (NEq termino1 termino2)
       <|> do
         (reservedOp lis "==")
         termino2 <- intexp
         return (Eq termino1 termino2))
    -- Second case: value (lower priority)
    <||> do
      (do
         reservedOp lis "true"
         return (BTrue)
       <|> do
         reservedOp lis "false"
         return (BFalse))

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

```haskell

comm :: Parser Comm
comm = chainl1 comm' (do { reservedOp lis ";" ; return (Seq) })

comm' :: Parser Comm
comm' =
    do
      reservedOp lis "skip"
      return Skip
    <|> do
      nombre <- identifier lis
      reservedOp lis "="
      valor <- intexp
      return (Let nombre valor)
    <|>
      ifthen_elseComm
    <|> do
      reservedOp lis "repeat"
      termino1 <- braces lis comm
      reservedOp lis "until"
      termino2 <- boolexp
      return (Repeat termino1 termino2)


ifthen_elseComm :: Parser Comm
ifthen_elseComm = do
                    reservedOp lis "if"
                    termino1 <- boolexp
                    termino2 <- braces lis comm
                    (do
                      reservedOp lis "else"
                      termino3 <- braces lis comm
                      return (IfThenElse termino1 termino2 termino3)
                     <|> do
                       return (IfThen termino1 termino2))
-------------------------------------
-- Funcion de parseo
-------------------------------------
parseComm :: SourceName -> String -> Either ParseError Comm
parseComm = parse (totParser comm)
```

4. Ejercicio 4

Las únicas modificaciones realizadas a la semántica al incluir la asignación y el operador "," son estas dos operaciones (el resto no son modificadas). Por lo tanto y como las semánticas del resto de operaciones ya estan realizadas por la cátedra (enunciado del TP 1 - Página 5), solo se realiza la semántica big-step para las nuevas operaciones.

$$\frac{\langle e, \sigma\rangle \Downarrow_{intexp} \langle n, \sigma'\rangle}{\langle v := e, \sigma\rangle \Downarrow_{intexp} \langle n, [\sigma'|v : e]\rangle}\ EAssgn$$

$$\frac{\langle e_0, \sigma\rangle \Downarrow_{intexp} \langle n_0, \sigma'\rangle \quad \langle e_1, \sigma'\rangle \Downarrow_{intexp} \langle n_1, \sigma''\rangle \quad ... \quad \langle e_m, \sigma^m\rangle \Downarrow_{intexp} \langle n_m, \sigma^{m+1}\rangle}{\langle e_0, e_1, ..., e_m, \sigma\rangle \Downarrow_{intexp} \langle n_m, \sigma^{m+1}\rangle}\ ESeq$$

5. Ejercicio 5

Efectivamente la relación de evaluación en un paso es determinista, para probarlo debemos mostrar que si $t \rightsquigarrow t'$ y $t \rightsquigarrow t''$ entonces $t' = t''$.

Para mostrarlo, haremos un desarrollo similar al visto en clase, utilizando inducción sobre la derivación $t \rightsquigarrow t'$. Tendremos como valido que la relación $\Downarrow$ es determinista (dado en el enunciado del ejercicio).

- Suponemos que se obtuvo $t' = \langle skip, [\sigma' \,|v : n]\rangle$ usando como última regla ASS Luego t es de la forma $\langle v = e, \sigma\rangle$ y la hipótesis es que $\langle e, \sigma\rangle \Downarrow \langle n, \sigma'\rangle$. Al ser la relación $\Downarrow$ determinista, no puede derivar a otro elemento. Por lo tanto, la única regla aplicable a t es ASS. Así resulta que $t' = t''$.

- Suponemos que se obtuvo $t' = \langle c_1, \sigma\rangle$ usando como última regla SEQ1. Luego t es de la forma $\langle skip; c_1, \sigma\rangle$. Entonces la única regla aplicable a t es SEQ1, por lo que debe suceder que $t' = t''$.

- Suponemos que se obtuvo $t' = \langle c_0'; c_1, \sigma'\rangle$ usando como última regla SEQ2 Luego t es de la forma $\langle c_0; c_1, \sigma\rangle$ y la hipótesis es que $\langle c_0, s\rangle \rightsquigarrow \langle c_0', \sigma'\rangle$. Por hipótesis inductiva, tenemos que la derivación $\langle c_0, \sigma\rangle \rightsquigarrow \langle c_0', s'\rangle$ es determinista, es decir, solo puede derivar a ese elemento. Como la única regla aplicable en t es SEQ2 (no podemos aplicar SEQ1 pues sabemos que c0 no es skip) tenemos que $t' = t''$.

- Suponemos que se obtuvo $t' = \langle c_0, \sigma'\rangle$ usando como última regla a IF1. Luego t es de la forma $\langle if\, b\, then\, c_0\, else\, c_1,\ \sigma\rangle$ y la hipótesis es que $\langle b, k\rangle \Downarrow \langle true, \sigma'\rangle$. Entonces si se hubiera usado la regla IF2, luego $\langle b, k\rangle \Downarrow \langle false, \sigma\rangle$ lo cual se contradice porque true != false y sabemos que $\Downarrow$ es una relación determinista. Por lo tanto la única regla aplicable es IF1. Y resulta $t' = t''$.

- Suponemos que se obtuvo $t' = \langle c_0, \sigma'\rangle$ usando como última regla a IF2. Luego t es de la forma $\langle if\, b\, then\, c_1\, else\, c_0,\ \sigma\rangle$ y la hipótesis es que $\langle b, k\rangle \Downarrow \langle false, \sigma'\rangle$. Entonces no puede usar la regla IF1 ya que la relación $\Downarrow$ es determinista entonces $\langle b, k\rangle$ no deriva $\langle true, s\rangle$. Por lo tanto la única regla aplicable es IF2.

- Suponemos que se obtuvo $t' = \langle c; if\, b\, then\, skip\, else\, repeat\, b\, until\, c, \sigma\rangle$ usando como última regla REPEAT. Por lo tanto t es de la forma $\langle repeat\, c\, until\, b, \sigma\rangle$ y vemos que la unica regla aplicable es REPEAT, por lo que $t' = t''$. No tiene sentido intentar aplicar por ejemplo SEQ1 ya que c_0 debería derivar ser skip y resultaría que $\langle skip, \sigma\rangle \rightsquigarrow \langle c_0', \sigma'\rangle$ lo cual ninguna no se puede hacer con ninguna regla.

Luego de probarlo para cada regla, por inducción queda probado que la evaluación en un paso $\rightsquigarrow$ es determinista.

6. Ejercicio 6

Para hacer menos engorrosa la lectura, separamos la demostración en diferentes árboles de derivación.

Árbol 1:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\langle 1, [[\sigma|x:2]|y:2]\rangle \Downarrow_{intexp} \langle 1, [[\sigma|x:2]|y:2]\rangle}\, NVal}{\langle y := 1, [[\sigma|x:2]|y:2]\rangle \Downarrow_{intexp} \langle 1, [[\sigma|x:2]|y:1]\rangle}\, EAssgn}{\langle x := y = 1, [[\sigma|x:2]|y:1]\rangle \rightsquigarrow \langle skip, [[\sigma|x:1]|y:1]\rangle}\, Ass}{\langle x := y = 1; repeat\, x = x - y\, until\, x == 0, [[\sigma|x:2]|y:1]\rangle \rightsquigarrow \langle skip; repeat\, x = x - y\, until\, x == 0, [[\sigma|x:1]|y:1]\rangle}\, Seq2$$

$$\dfrac{\langle x := y = 1; repeat\, x = x - y\, until\, x == 0, [[\sigma|x:2]|y:1]\rangle \rightsquigarrow \langle skip; repeat\, x = x - y\, until\, x == 0, [[\sigma|x:1]|y:1]\rangle}{\langle x := y = 1; repeat\, x = x - y\, until\, x == 0, [[\sigma|x:2]|y:1]\rangle \rightsquigarrow^* \langle skip; repeat\, x = x - y\, until\, x == 0, [[\sigma|x:1]|y:1]\rangle}\, Clousure$$

Árbol 2:

$$\dfrac{\dfrac{}{\langle skip; repeat\, x = x - y\, until\, x == 0, [[\sigma|x:1]|y:1]\rangle \rightsquigarrow \langle repeat\, x = x - y\, until\, x == 0, [[\sigma|x:1]|y:1]\rangle}\, SEQ1}{\langle skip; repeat\, x = x - y\, until\, x == 0, [[\sigma|x:1]|y:1]\rangle \rightsquigarrow^* \langle repeat\, x = x - y\, until\, x == 0, [[\sigma|x:1]|y:1]\rangle}\, Clousure$$

Árbol 3:

$$\dfrac{\dfrac{}{\langle repeat\, x = x - y\, until\, x == 0, [[\sigma|x:1]|y:1]\rangle \rightsquigarrow \langle x = x - y; if\, x == 0\, then\, skip\, else\, repeat\, x = x - y\, until\, x == 0, [[\sigma|x:1]|y:1]\rangle}\, Repeat}{\langle repeat\, x = x - y\, until\, x == 0, [[\sigma|x:1]|y:1]\rangle \rightsquigarrow^* \langle x = x - y; if\, x == 0\, then\, skip\, else\, repeat\, x = x - y\, until\, x == 0, [[\sigma|x:1]|y:1]\rangle}\, Clousure$$

Árbol 4:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\langle x, [[\sigma|x:1]|y:1]\rangle \Downarrow_{intexp} \langle 1, [[\sigma|x:1]|y:1]\rangle}\, Var \quad \dfrac{}{\langle y, [[\sigma|x:1]|y:1]\rangle \Downarrow_{intexp} \langle 1, [[\sigma|x:1]|y:1]\rangle}\, Var}{\langle x - y, [[\sigma|x:1]|y:1]\rangle \Downarrow_{intexp} \langle 1 - 1, [[\sigma|x:1]|y:1]\rangle}\, Minus}{\langle x = x - y, [[\sigma|x:1]|y:1]\rangle \rightsquigarrow \langle skip, [[\sigma|x:0]|y:1]\rangle}\, Ass}{\langle x = x - y; if\ x == 0\ then\ skip\ else\ repeat\ x = x - y\ until\ x == 0, [[\sigma|x:1]|y:1]\rangle \rightsquigarrow \langle skip; if\ x == 0\ then\ skip\ else\ repeat\ x = x - y\ until\ x == 0, [[\sigma|x:0]|y:1]\rangle}\, SEQ2$$

$$\dfrac{}{\langle x = x - y; if\ x == 0\ then\ skip\ else\ repeat\ x = x - y\ until\ x == 0, [[\sigma|x:1]|y:1]\rangle \rightsquigarrow^* \langle skip; if\ x == 0\ then\ skip\ else\ repeat\ x = x - y\ until\ x == 0, [[\sigma|x:0]|y:1]\rangle}\, Clousure$$

Árbol 5:

$$\dfrac{\dfrac{}{\langle skip; if\ x == 0\ then\ skip\ else\ repeat\ x = x - y\ until\ x == 0, [[\sigma|x:0]|y:1]\rangle \rightsquigarrow \langle if\ x == 0\ then\ skip\ else\ repeat\ x = x - y\ until\ x == 0, [[\sigma|x:0]|y:1]\rangle}\, SEQ1}{\langle skip; if\ x == 0\ then\ skip\ else\ repeat\ x = x - y\ until\ x == 0, [[\sigma|x:0]|y:1]\rangle \rightsquigarrow^* \langle if\ x == 0\ then\ skip\ else\ repeat\ x = x - y\ until\ x == 0, [[\sigma|x:0]|y:1]\rangle}\, Clousure$$

Árbol 6:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\langle x, [[\sigma|x:0]|y:1]\rangle \Downarrow_{intexp} \langle 0, [[\sigma|x:0]|y:1]\rangle}\, Var \quad \dfrac{}{\langle 1, [[\sigma|x:0]|y:1]\rangle \Downarrow_{intexp} \langle 1, [[\sigma|x:0]|y:1]\rangle}\, NVal}{\langle x == 0, [[\sigma|x:0]|y:1]\rangle \Downarrow_{intexp} \langle 0 = 0, [[\sigma|x:0]|y:1]\rangle}\, EQ}{\langle if\ x == 0\ then\ skip\ else\ repeat\ x = x - y\ until\ x == 0, [[\sigma|x:0]|y:1]\rangle \rightsquigarrow \langle skip, [[\sigma|x:0]|y:1]\rangle}\, IF1}{\langle if\ x == 0\ then\ skip\ else\ repeat\ x = x - y\ until\ x == 0, [[\sigma|x:0]|y:1]\rangle \rightsquigarrow^* \langle skip, [[\sigma|x:0]|y:1]\rangle}\, Clousure$$

Árbol 7: Para no escribir los árboles de derivación en cada transitiva realizada, se dejará "Árbol n" como representante y "Tn" para los resultados.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{Arbol1 \quad Arbol2}{T1}\, Transitive \quad Arbol3}{T2}\, Transitive \quad Arbol4}{T3}\, Transitive \quad Arbol5}{T4}\, Transitive \quad Arbol6}{\langle x := y = 1; repeat\ x = x - y\ until\ x == 0, [[\sigma|x:2]|y:1]\rangle \rightsquigarrow^* \langle skip, [[\sigma|x:0]|y:1]\rangle}\, Transitive$$

7. Ejercicio 7

Listing 5: Parser Haskell

```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
-- Completar la definici n
initState :: State
initState = M.empty

-- Busca el valor de una variable en un estado
-- Completar la definici n
lookfor :: Variable -> State -> Int
lookfor v s = s M.! v

-- Cambia el valor de una variable en un estado
-- Completar la definici n
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
-- Completar la definici n
stepComm :: Comm -> State -> Pair Comm State
stepComm (Skip) s = (Skip :!: s)
stepComm (Let x y) s = let (n :!: s') = evalExp y s in (Skip :!: update x n s')
stepComm (Seq Skip n) s = (n :!: s)
stepComm (Seq s1 s2) s = let (e' :!: s') = stepComm s1 s in (Seq e' s2 :!: s')
stepComm (IfThenElse x y z) s = let (n :!: s') = evalExp x s
```

```
                                in if n then (y :!: s') else (z :!: s')
stepComm (IfThen x y) s = let (n :!: s') = evalExp x s
                          in if n then (y :!: s') else (Skip :!: s')
stepComm (Repeat x b) s = ((Seq x (IfThenElse b Skip (Repeat x b))) :!: s)


evalBinary x y s = (v1, v2, s'') where
                                (v1 :!: s') = evalExp x s
                                (v2 :!: s'') = evalExp y s'

-- Evalua una expresion
evalExp :: Exp a -> State -> Pair a State
--Int
evalExp (Const n) s = (n :!: s)
evalExp (Var n) s = (lookfor n s :!: s)
evalExp (UMinus n) s = ((-v) :!: s') where
                                    (v :!: s') = evalExp n s

evalExp (Plus x y) s = (v1 + v2 :!: s'') where
                                       (v1, v2, s'') = evalBinary x y s
evalExp (Minus x y) s = (v1 - v2 :!: s'') where
                                        (v1, v2, s'') = evalBinary x y s
evalExp (Times x y) s = (v1 * v2 :!: s'') where
                                        (v1, v2, s'') = evalBinary x y s
evalExp (Div x y) s = (div v1 v2 :!: s'') where
                                        (v1, v2, s'') = evalBinary x y s
evalExp (EAssgn xs y) s = (v :!: update xs v s') where
                                              (v :!: s') = evalExp y s

-- Bool
evalExp BTrue s = (True :!: s)
evalExp BFalse s = (False :!: s)
evalExp (Lt x y) s = (v1 < v2 :!: s'') where
                                     (v1, v2, s'') = evalBinary x y s

evalExp (Gt x y) s = (v1 > v2 :!: s'') where
                                     (v1, v2, s'') = evalBinary x y s

evalExp (Eq x y) s = ((v1 == v2) :!: s'') where
                                      (v1, v2, s'') = evalBinary x y s
evalExp (NEq x y) s = (v1 /= v2 :!: s'') where
                                       (v1, v2, s'') = evalBinary x y s
evalExp (And x y) s = ((v1 && v2) :!: s'') where
                                        (v1, v2, s'') = evalBinary x y s
evalExp (Or x y) s = ((v1 || v2) :!: s'') where
                                       (v1, v2, s'') = evalBinary x y s
evalExp (Not x) s = (not v1 :!: s') where
                                   (v1 :!: s') = evalExp x s
```

8. Ejercicio 8

Listing 6: Parser Haskell

```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
-- Completar la definici n
initState :: State
initState = M.empty

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

-- Cambia el valor de una variable en un estado
-- Completar la definici n
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
-- Completar la definici n
stepComm :: Comm -> State -> Either Error (Pair Comm State)
stepComm (Skip) s = Right (Skip :!: s)
```

```haskell
stepComm (Let x y) s = case evalExp y s of
                          Right (n :!: s') -> Right (Skip :!: update x n s')
                          Left e -> Left e
stepComm (Seq Skip n) s = Right (n :!: s)
stepComm (Seq s1 s2) s = case stepComm s1 s of
                            Right (e' :!: s') -> Right (Seq e' s2 :!: s')
                            Left e -> Left e
stepComm (IfThenElse x y z) s = case evalExp x s of
                                   Right (True :!: s') -> Right (y :!: s')
                                   Right (False :!: s') -> Right (z :!: s')
                                   Left e -> Left e
stepComm (IfThen x y) s = case evalExp x s of
                             Right (True :!: s') -> Right (y :!: s')
                             Right (False :!: s') -> Right (Skip :!: s')
                             Left e -> Left e
stepComm (Repeat x b) s = Right ((Seq x (IfThenElse b Skip (Repeat x b))) :!: s)


evalBinary x y f s = case evalExp x s of
                        Right (v1 :!: s') -> case evalExp y s' of
                                                Right (v2 :!: s'') -> Right ↩
                                                    (f v1 v2 :!: s'')
                                                Left e -> Left e
                        Left e -> Left e

-- Evalua una expresion
-- Completar la definici n
evalExp :: Exp a -> State -> Either Error (Pair a State)
-- Int
evalExp (Const n) e = Right (n :!: e)
evalExp (Var n) e = case lookfor n e of
                       Right x -> Right (x :!: e)
                       Left x -> Left x
evalExp (UMinus n) e = case evalExp n e of
                          Right (v :!: e') -> Right ((-v) :!: e')
                          Left x -> Left x
evalExp (EAssgn xs y) s = case evalExp y s of
                             Right (v :!: s') -> Right (v :!: update xs v s')
                             Left e -> Left e
evalExp (ESeq s1 s2) s = case evalExp s1 s of
                            Right (_ :!: s') -> evalExp s2 s'
                            Left e -> Left e
evalExp (Plus x y) s = evalBinary x y (+) s
evalExp (Minus x y) s = evalBinary x y (-) s
evalExp (Times x y) s = evalBinary x y (*) s
evalExp (Div x y) s = case evalExp x s of
                         Right (v1 :!: s') -> case evalExp y s' of
                                                 Right (0 :!: s'') -> Left ↩
                                                     DivByZero
                                                 Right (v2 :!: s'') -> Right ↩
                                                     (div v1 v2 :!: s'')
```

```
95                                                      Left e -> Left e
96                              Left e -> Left e
97
98  -- Bool
99  evalExp BTrue s = Right (True :!: s)
100 evalExp BFalse s = Right (False :!: s)
101 evalExp (Lt x y) s = evalBinary x y (<) s
102 evalExp (Gt x y) s = evalBinary x y (>) s
103 evalExp (Eq x y) s = evalBinary x y (==) s
104 evalExp (NEq x y) s = evalBinary x y (/=) s
105 evalExp (And x y) s = evalBinary x y (&&) s
106 evalExp (Or x y) s = evalBinary x y (||) s
107 evalExp (Not x) s = case evalExp x s of
108                       Right (v1 :!: s') -> Right (not v1 :!: s')
109                       Left e -> Left e
```

9. Ejercicio 9

Para almacenar la traza del programa, se agregó al estado una lista de tuplas en donde cada una representa un valor que tuvo cierta variable en algún momento de la ejecución. Por lo tanto al final, tenemos como resultado una lista de tuplas representando el valor final de cada variable (Map que ya se encontraba en el estado) y una lista que representa cronológicamente el valor que adoptó cada variable.

Listing 7: Parser Haskell

```
1  module Eval3
2    ( eval
3    , State
4    )
5  where
6
7  import           AST
8  import qualified Data.Map.Strict              as M
9  import           Data.Strict.Tuple
10
11 -- Estados
12 type State = (M.Map Variable Int, [(String, Int)])
13
14 -- Estado nulo
15 -- Completar la definici n
16 initState :: State
17 initState = (M.empty,[])
18
19 -- Busca el valor de una variable en un estado
20 -- Completar la definici n
21 lookfor :: Variable -> State -> Either Error Int
```

```haskell
lookfor v (s,_) = case M.lookup v s of
                  Just n -> Right n
                  Nothing -> Left UndefVar

-- Cambia el valor de una variable en un estado
-- Completar la definici n
update :: Variable -> Int -> State -> State
update v nv (s, xs) = (M.insert v nv s, (v, nv):xs)

-- Evalua un programa en el estado nulo
-- Hacemos un reverse de la traza para que quede en el orden del programa
eval :: Comm -> Either Error State
eval p = case stepCommStar p initState of
              Left e -> Left e
              Right (a, b) -> Right (a, reverse b)

-- 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
-- Completar la definici n
stepComm :: Comm -> State -> Either Error (Pair Comm State)
stepComm (Skip) s = Right (Skip :!: s)
stepComm (Let x y) s = case evalExp y s of
                           Right (n :!: s') -> Right (Skip :!: update x n s')
                           Left e -> Left e
stepComm (Seq Skip n) s = Right (n :!: s)
stepComm (Seq s1 s2) s = case stepComm s1 s of
                             Right (e' :!: s') -> Right (Seq e' s2 :!: s')
                             Left e -> Left e
stepComm (IfThenElse x y z) s = case evalExp x s of
                                      Right (True :!: s') -> Right (y :!: s')
                                      Right (False :!: s') -> Right (z :!: s')
                                      Left e -> Left e
stepComm (IfThen x y) s = case evalExp x s of
                               Right (True :!: s') -> Right (y :!: s')
                               Right (False :!: s') -> Right (Skip :!: s')
                               Left e -> Left e
stepComm (Repeat x b) s = Right ((Seq x (IfThenElse b Skip (Repeat x b))) :!: s)


evalBinary x y f s = case evalExp x s of
                         Right (v1 :!: s') -> case evalExp y s' of
                                                  Right (v2 :!: s'') -> Right ↩
                                                      (f v1 v2 :!: s'')
                                                  Left e -> Left e
```

```
72                          Left e -> Left e
73
74  -- Evalua una expresion
75  -- Completar la definici n
76  evalExp :: Exp a -> State -> Either Error (Pair a State)
77  -- Int
78  evalExp (Const n) e = Right (n :!: e)
79  evalExp (Var n) e = case lookfor n e of
80                          Right x -> Right (x :!: e)
81                          Left x -> Left x
82  evalExp (UMinus n) e = case evalExp n e of
83                             Right (v :!: e') -> Right ((-v) :!: e')
84                             Left x -> Left x
85  evalExp (EAssgn xs y) s = case evalExp y s of
86                                Right (v :!: s') -> Right (v :!: update xs v s')
87                                Left e -> Left e
88  evalExp (ESeq s1 s2) s = case evalExp s1 s of
89                                Right (_ :!: s') -> evalExp s2 s'
90                                Left e -> Left e
91  evalExp (Plus x y) s = evalBinary x y (+) s
92  evalExp (Minus x y) s = evalBinary x y (-) s
93  evalExp (Times x y) s = evalBinary x y (*) s
94  evalExp (Div x y) s = case evalExp x s of
95                             Right (v1 :!: s') -> case evalExp y s' of
96                                                      Right (0 :!: s'') -> Left ↩
                                                          DivByZero
97                                                      Right (v2 :!: s'') -> Right ↩
                                                          (div v1 v2 :!: s'')
98                                                      Left e -> Left e
99                             Left e -> Left e
100
101 -- Bool
102 evalExp BTrue s = Right (True :!: s)
103 evalExp BFalse s = Right (False :!: s)
104 evalExp (Lt x y) s = evalBinary x y (<) s
105 evalExp (Gt x y) s = evalBinary x y (>) s
106 evalExp (Eq x y) s = evalBinary x y (==) s
107 evalExp (NEq x y) s = evalBinary x y (/=) s
108 evalExp (And x y) s = evalBinary x y (&&) s
109 evalExp (Or x y) s = evalBinary x y (||) s
110 evalExp (Not x) s = case evalExp x s of
111                       Right (v1 :!: s') -> Right (not v1 :!: s')
112                       Left e -> Left e
```

10. Ejercicio 10

10.1. Ampliación de la gramática abstracta

Mostramos la parte de comandos de la gramática ya que es la única que se modifica.

Listing 8: Ampliación de la gramática abstracta

```
comm ::= skip
       | var = intexp
       | comm; comm
       | if boolexp then comm else comm
       | repeat comm until boolexp
       | for comm boolexp comm
```

10.2. Expansión semántica operacional

Para implementar el operador "FOR", hicimos uso de una secuencia evaluando "a" como primera instancia y luego realizando un repeat sobre "b" y "c".

$$\frac{}{\langle \textbf{for}\ (a; b; c), \sigma\rangle \rightsquigarrow \langle a; \textbf{if}\ b\ \textbf{then repeat}\ c\ \textbf{until NOT}\ b, \sigma\rangle}\ FOR$$