Trabajo Práctico
Número 1

Análisis de Lenguajes de Programación

Realizado por

Gianfranco Paoloni
Sebastián Zimmermann

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

FCEIA
FACULTAD DE
CIENCIAS EXACTAS,
INGENIERIA Y AGRIMENSURA

Ejercicio 1

Para este ejercicio, solo fue necesario agregar algunas operaciones a las expresiones aritméticas de enteros. Dando esto por resultado la siguiente Sintaxis Abstracta

Sintaxis Abstracta

```
intexp ::= nat
         | var
         | − intexp
         | intexp + intexp
         | intexp − intexp
         | intexp × intexp
         | intexp ÷ intexp
         | var = intexp
         | intexp , intexp

boolexp ::= true
          | false
          | intexp == intexp
          | intexp != intexp
          | intexp < intexp
          | intexp > intexp
          | boolexp ∧ boolexp
          | boolexp ∨ boolexp
          | ¬ boolexp

comm ::= skip
       | assignL
       | var = intexp
       | comm ; comm
       | if boolexp then comm
       | if boolexp then comm else comm
       | while boolexp do comm
```

Sintaxis Concreta

En el caso de la sintaxis concreta, el método fue similar.

```
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 ')'
         | var '=' intexp
         | intexp ',' intexp

boolexp ::= 'true'
          | 'false'
          | intexp '==' intexp
          | intexp '!=' intexp
          | intexp '<' intexp
          | intexp '>' intexp
          | boolexp '&&' boolexp
          | boolexp '||' boolexp
          | '!' boolexp
          | '(' boolexp ')'

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

Ejercicio 2

La resolución del ejercicio 2 constitió en un enfoque similar al ejercicio 1, dando por resultado el siguiente código de Haskell

```haskell
module AST where

-- Identificadores de Variable
type Variable = String

-- Expresiones Aritmeticas
data IntExp = Const Integer
            | Var Variable
            | UMinus IntExp
            | Plus IntExp IntExp
            | Minus IntExp IntExp
            | Times IntExp IntExp
            | Div IntExp IntExp
            | Assign Variable IntExp
            | Secuence IntExp IntExp
 deriving Show

-- Expresiones Booleanas
data BoolExp = BTrue
             | BFalse
             | Eq IntExp IntExp
             | NEq IntExp IntExp
             | Lt IntExp IntExp
             | Gt IntExp IntExp
             | And BoolExp BoolExp
             | Or BoolExp BoolExp
             | Not BoolExp
 deriving Show

-- Comandos (sentencias)

data Comm = Skip
          | Let Variable IntExp
          | Seq Comm Comm
          | IfThenElse BoolExp Comm Comm
          | While BoolExp Comm
 deriving Show
```

Ejercicio 3

Este ejercicio fue realizado en el archivo Parser.hs, presente en el **Anexo 2** al final de este trabajo.

Ejercicio 4

A continuación, mostramos la semántica Big-Step de expresiones enteras extendida, y la semántica de expresiones booleanas extendida, pues ahora pueden haber cambios en el estado σ. Más adelante veremos como repercuten dichas operaciones en la semántica de comandos.

$$\frac{\langle e, \sigma\rangle \Downarrow_{\text{intexp}} \langle n, \sigma\rangle}{\langle v := e, \sigma\rangle \Downarrow_{\text{intexp}} \langle n, [\sigma|v : e]\rangle} \ \textsc{Assign}$$

$$\frac{\langle e_0, \sigma\rangle \Downarrow_{\text{intexp}} \langle n_0, \sigma'\rangle \quad \langle e_1, \sigma'\rangle \Downarrow_{\text{intexp}} \langle n_1, \sigma''\rangle}{\langle e_0, e_1, \sigma\rangle \Downarrow_{\text{intexp}} \langle n_1, \sigma''\rangle} \ \textsc{Comma}$$

$$\frac{}{\langle nv, \sigma\rangle \Downarrow_{\text{intexp}} \langle nv, \sigma\rangle} \ \textsc{NVal} \qquad \frac{}{\langle v, \sigma\rangle \Downarrow_{\text{intexp}} \langle \sigma v, \sigma\rangle} \ \textsc{Var} \qquad \frac{\langle e, \sigma\rangle \Downarrow_{\text{intexp}} \langle n, \sigma'\rangle}{\langle -_u e, \sigma\rangle \Downarrow_{\text{intexp}} \langle -n, \sigma'\rangle} \ \textsc{UMinus}$$

$$\frac{\langle e_0, \sigma\rangle \Downarrow_{\text{intexp}} \langle n_0, \sigma'\rangle \quad \langle e_1, \sigma'\rangle \Downarrow_{\text{intexp}} \langle n_1, \sigma''\rangle}{\langle e_0 + e_1, \sigma\rangle \Downarrow_{\text{intexp}} \langle n_0 + n_1, \sigma''\rangle} \ \textsc{Plus} \qquad \text{(análogamente para } -_b,\ \times)$$

$$\frac{\langle e_0, \sigma\rangle \Downarrow_{\text{intexp}} \langle n_0, \sigma'\rangle \quad \langle e_1, \sigma'\rangle \Downarrow_{\text{intexp}} \langle n_1, \sigma''\rangle \quad n_1 \neq 0}{\langle e_0 \div e_1, \sigma\rangle \Downarrow_{\text{intexp}} \langle n_0 \div n_1, \sigma''\rangle} \ \textsc{Div}$$

$$\frac{\langle e_0, \sigma\rangle \Downarrow_{\text{intexp}} \langle n_0, \sigma'\rangle \quad \langle e_1, \sigma'\rangle \Downarrow_{\text{intexp}} \langle n_1, \sigma''\rangle}{\langle e_0 == e_1, \sigma\rangle \Downarrow_{\text{boolexp}} \langle n_0 = n_1, \sigma''\rangle} \ \textsc{Eq} \qquad \text{(análogamente para } <\ ,\ > \text{ y } \neq)$$

$$\frac{}{\langle bv, \sigma\rangle \Downarrow_{\text{boolexp}} \langle bv, \sigma\rangle} \ \textsc{BVal} \qquad \frac{\langle p, \sigma\rangle \Downarrow_{\text{boolexp}} \langle b, \sigma'\rangle}{\langle \neg p, \sigma\rangle \Downarrow_{\text{boolexp}} \langle \neg b, \sigma'\rangle} \ \textsc{Not}$$

$$\frac{\langle p_0, \sigma\rangle \Downarrow_{\text{boolexp}} \langle b_0, \sigma'\rangle \quad \langle p_1, \sigma'\rangle \Downarrow_{\text{boolexp}} \langle b_1, \sigma''\rangle}{\langle p_0 \vee p_1, \sigma\rangle \Downarrow_{\text{boolexp}} \langle b_0 \vee b_1, \sigma''\rangle} \ \textsc{Or} \qquad \text{(análogamente para } \wedge)$$

Ejercicio 5

Queremos probar que la relación $\rightsquigarrow$ es determinista, es decir que, si $t \rightsquigarrow v, t \rightsquigarrow v' \Rightarrow v = v'$
Vamos a probar esto por *inducción sobre la derivación* $t \rightsquigarrow v$, utilizando análisis por casos.
Además, asumiremos que las relaciones $\Downarrow_{\text{intexp}}$ y $\Downarrow_{\text{boolexp}}$ son deterministas

Demostración Realizaremos un **análisis por casos** para la derivación $t \rightsquigarrow v$

- Si la última derivación fue un ASS entonces:
  - $\langle e, \sigma \rangle \Downarrow_{\text{intexp}} \langle n, \sigma' \rangle$
  - $t = \langle v := e, \sigma \rangle$
  - $v = [\sigma | x : n]$

  Como $\Downarrow_{\text{intexp}}$ es determinista por hipótesis, la única regla que podemos aplicar es **ASS** luego $v = v'$

- Si la última derivación fue un $SKIP$ entonces:
  - $t = \langle$ **skip**, $\sigma \rangle$
  - $v = \sigma'$

  Como t comienza con un **skip**, la única regla que podemos aplicar es **SKIP** luego $v = v'$

- Si la última derivación fue un IF_1 entonces:
  - $\langle b, \sigma \rangle \Downarrow_{\text{boolexp}} \langle true, \sigma' \rangle$
  - $t = \langle if$ b then c_0, $\sigma \rangle$
  - $v = \langle c_0, \sigma' \rangle$

  Como t comienza con un `if`, existen 4 posibles reglas que utilizan un if, pero como t no posee un `else`, solo puedo aplicar o IF_1 o IF_2.

  Sin embargo, $\Downarrow_{\text{boolexp}}$ es determinista, si b evalúa **true** entonces no puede evaluar **false**, y por lo tanto la única regla que puedo utilizar es IF_1 y entonces $v = v'$

- Si la última derivación fue un IF_2 entonces:
  - $\langle b, \sigma \rangle \Downarrow_{\text{boolexp}} \langle false, \sigma' \rangle$
  - $t = \langle if$ b then c_0, $\sigma \rangle$
  - $v = \sigma'$

  De forma análoga al caso anterior, solamente podemos aplicar IF_1 o IF_2.

  Sin embargo, como $\Downarrow_{\text{boolexp}}$ es determinista, si b evalúa **false** entonces no puede evaluar **true**, y por lo tanto la única regla que puedo utilizar es IF_2 y por lo tanto $v = v'$

- Si la última derivación fue un IF_3 entonces:
  - $\langle b, \sigma \rangle \Downarrow_{\text{boolexp}} \langle true, \sigma' \rangle$
  - $t = \langle if$ b then c_0 else c_1, $\sigma \rangle$
  - $v = \langle c_0, \sigma' \rangle$

  De forma análoga al IF_1, como t comienza con un `if`, existen 4 posibles reglas que utilizan un if, pero como t posee `else`, solo puedo aplicar o IF_3 o IF_4.

  Sin embargo, $\Downarrow_{\text{boolexp}}$ es determinista, si b evalúa **true** entonces no puede evaluar **false**, y por lo tanto la única regla que puedo utilizar es IF_3 y entonces $v = v'$

- Si la última derivación fue un IF_4 entonces:
  - $\langle b, \sigma \rangle \Downarrow_{\text{boolexp}} \langle false, \sigma' \rangle$
  - $t = \langle if$ b then c_0 else c_1, $\sigma\rangle$
  - $v = \langle c_1, \sigma' \rangle$

  De forma análoga al IF_3, solo puedo aplicar o IF_3 o IF_4.

  Sin embargo, $\Downarrow_{\text{boolexp}}$ es determinista, si b evalúa **true** entonces no puede evaluar **false**, y por lo tanto la única regla que puedo utilizar es IF_4 y entonces $v = v'$

- Si la última derivación fue un $WHILE_1$ entonces:
  - $\langle b, \sigma \rangle \Downarrow_{\text{boolexp}} \langle true, \sigma' \rangle$
  - $t = \langle$*while* b *do* c, $\sigma\rangle$
  - $v = \langle$ c ; *while* b *do* c, $\sigma'\rangle$

  Como t comienza con un `while`, existen 2 posibles reglas que utilizan un while, sin embargo, como $\Downarrow_{\text{boolexp}}$ es determinista, si b evalúa **true** no puede evaluar **false**, por lo tanto, la única regla posible de aplicar es $WHILE_1$ y entonces $v = v'$

- Si la última derivación fue un $WHILE_2$ entonces:
  - $\langle b, \sigma \rangle \Downarrow_{\text{boolexp}} \langle false, \sigma' \rangle$
  - $t = \langle$*while* b *do* c, $\sigma\rangle$
  - $v = \langle \sigma' \rangle$

  Análogamente a $WHILE_1$, existen 2 posibles evaluaciones que utilizan while, pero como $\Downarrow_{\text{boolexp}}$ es determinista, solo podemos utilizar $WHILE_2$ y por lo tanto, $v = v'$

- Si la última derivación fue un SEQ_1 entonces:
  - $a = \langle c_0, \sigma \rangle \rightsquigarrow \sigma'$
  - $t = \langle c_0 \ ; \ c_1 \ , \ \sigma \rangle$
  - $v = \langle c_1, \ \sigma' \rangle$

  Como t tiene forma de $\langle c_0 \ ; \ c_1 \ , \ \sigma \rangle$, solo puedo utilizar las reglas SEQ_1 y SEQ_2.

  Sin embargo, por **Hipótesis Inductiva** la derivación en a es determinista. Por lo tanto, no puede evaluar a una expresión $\langle c_0', \ \sigma' \rangle$ y luego la única regla posible de usar es SEQ_1.

  Resulta entonces, $v = v'$

- Si la última derivación fue un SEQ_2 entonces:
  - $a = \langle c_0, \sigma \rangle \rightsquigarrow \langle c_0', \sigma' \rangle$
  - $t = \langle c_0 \ ; \ c_1 \ , \ \sigma \rangle$
  - $v = \langle c_0' \ , \ c_1, \ \sigma' \rangle$

  Análogamente a SEQ_1, como t tiene forma de $\langle c_0 \ ; \ c_1 \ , \ \sigma \rangle$, solo puedo utilizar las reglas SEQ_1 y SEQ_2.

  Sin embargo, por **Hipótesis Inductiva** la derivación en a es determinista. Por lo tanto, no puede evaluar a una expresión σ' y luego la única regla posible de usar es SEQ_2.

  Resulta entonces, $v = v'$

Acabamos de probar entonces que la relación de evaluación en un paso $\rightsquigarrow$ es determinista para cada caso posible, y por lo tanto $\rightsquigarrow$ es determinista.

Ejercicio 6

Aclaración: Anexaremos las reglas de Semántica Operacional Estructural para Comandos que utilizaremos para este ejercicio al final del trabajo.

Queremos probar que

$$\langle x = x - 1 \ ; \ \mathit{while} \ x > 0 \ \mathit{do} \ x = x - 2, [\sigma|x:2]\rangle \rightsquigarrow^* [\sigma|x:-1]$$

Para esto vamos a estructurar la pureba en varios sub-árboles, que se demuestran a continuación:

Subarbol 1

T 1.1

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\langle x, [\sigma|x:2]\rangle \Downarrow_{\text{intexp}} \langle 2, [\sigma|x:2]\rangle}\textsc{Var} \quad \dfrac{}{\langle 1, [\sigma|x:2]\rangle \Downarrow_{\text{intexp}} \langle 1, [\sigma|x:2]\rangle}\textsc{NVal}}{\langle x-1, [\sigma|x:2]\rangle \Downarrow_{\text{intexp}} \langle 1, [\sigma|x:2]\rangle}\textsc{Minus}}{\langle x = x-1, [\sigma|x:2]\rangle \rightsquigarrow [\sigma|x:1]}\textsc{Ass}}{\langle x = x - 1 \ ; \ \mathit{while} \ x > 0 \ \mathit{do} \ x = x - 2, [\sigma|x:2]\rangle \rightsquigarrow \langle \mathit{while} \ x > 0 \ \mathit{do} \ x = x - 2, [\sigma|x:1]\rangle}\textsc{Seq1}}{\langle x = x - 1 \ ; \ \mathit{while} \ x > 0 \ \mathit{do} \ x = x - 2, [\sigma|x:2]\rangle \rightsquigarrow^* \langle \mathit{while} \ x > 0 \ \mathit{do} \ x = x - 2, [\sigma|x:1]\rangle}\text{R1}$$

T 1.2

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\langle x, [\sigma|x:1]\rangle \Downarrow_{\text{intexp}} \langle 1, [\sigma|x:1]\rangle}\textsc{Var} \quad \dfrac{}{\langle 0, [\sigma|x:1]\rangle \Downarrow_{\text{intexp}} \langle 0, [\sigma|x:1]\rangle}\textsc{NVal}}{\langle x > 0, [\sigma|x:1]\rangle \Downarrow_{\text{boolexp}} \langle \mathit{true}, [\sigma|x:1]\rangle}\textsc{Gt}}{\langle \mathit{while} \ x > 0 \ \mathit{do} \ x = x - 2, [\sigma|x:1]\rangle \rightsquigarrow \langle x = x - 2 \ ; \ \mathit{while} \ x > 0 \ \mathit{do} \ x = x - 2, [\sigma|x:1]\rangle}\textsc{While1}}{\langle \mathit{while} \ x > 0 \ \mathit{do} \ x = x - 2, [\sigma|x:1]\rangle \rightsquigarrow^* \langle x = x - 2 \ ; \ \mathit{while} \ x > 0 \ \mathit{do} \ x = x - 2, [\sigma|x:1]\rangle}\text{R1}$$

ST 1 Ahora vamos a demostrar nuestro subarbol 1

$$\dfrac{T1{,}1 \qquad\qquad T1{,}2}{\langle x = x - 1 \ ; \ \mathit{while} \ x > 0 \ \mathit{do} \ x = x - 2, [\sigma|x:2]\rangle \rightsquigarrow^* \langle x = x - 2 \ ; \ \mathit{while} \ x > 0 \ \mathit{do} \ x = x - 2, [\sigma|x:1]\rangle}\text{R3}$$

Subarbol 2

T 2.1

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\langle x, [\sigma|x:1]\rangle \Downarrow_{\text{intexp}} \langle 1, [\sigma|x:1]\rangle}\textsc{Var} \quad \dfrac{}{\langle 2, [\sigma|x:1]\rangle \Downarrow_{\text{intexp}} \langle 2, [\sigma|x:1]\rangle}\textsc{NVal}}{\langle x-2, [\sigma|x:1]\rangle \Downarrow_{\text{intexp}} \langle -1, [\sigma|x:1]\rangle}\textsc{Minus}}{\langle x = x-2, [\sigma|x:1]\rangle \rightsquigarrow [\sigma|x:-1]}\textsc{Ass}}{\langle x = x - 2 \ ; \ \mathit{while} \ x > 0 \ \mathit{do} \ x = x - 2, [\sigma|x:1]\rangle \rightsquigarrow \langle \mathit{while} \ x > 0 \ \mathit{do} \ x = x - 2, [\sigma|x:-1]\rangle}\textsc{Seq1}}{\langle x = x - 2 \ ; \ \mathit{while} \ x > 0 \ \mathit{do} \ x = x - 2, [\sigma|x:1]\rangle \rightsquigarrow^* \langle \mathit{while} \ x > 0 \ \mathit{do} \ x = x - 2, [\sigma|x:-1]\rangle}\text{R1}$$

T 2.2

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\langle x, [\sigma|x:-1]\rangle \Downarrow_{\text{intexp}} \langle -1, [\sigma|x:-1]\rangle}\textsc{Var} \quad \dfrac{}{\langle 0, [\sigma|x:-1]\rangle \Downarrow_{\text{intexp}} \langle 0, [\sigma|x:-1]\rangle}\textsc{NVal}}{\langle x > 0, [\sigma|x:-1]\rangle \Downarrow_{\text{boolexp}} \langle \mathit{false}, [\sigma|x:-1]\rangle}\textsc{Gt}}{\langle \mathit{while} \ x > 0 \ \mathit{do} \ x = x - 2, [\sigma|x:-1]\rangle \rightsquigarrow [\sigma|x:-1]}\textsc{While2}}{\langle \mathit{while} \ x > 0 \ \mathit{do} \ x = x - 2, [\sigma|x:-1]\rangle \rightsquigarrow^* [\sigma|x:-1]}\text{R1}$$

ST 2 Ahora vamos a demostrar nuestro subarbol 2

$$\dfrac{T2{,}1 \qquad\qquad\qquad\qquad T2{,}2}{\langle x = x - 2 \ ; \ while\ x > 0\ do\ x = x - 2, [\sigma|x:1]\rangle \rightsquigarrow^{*} [\sigma|x:-1]}\ \text{R3}$$

Árbol final Ahora, habiendo probado los árboles anteriores

$$\dfrac{ST1 \qquad\qquad\qquad\qquad ST2}{\langle x = x - 1 \ ; \ while\ x > 0\ do\ x = x - 2, [\sigma|x:2]\rangle \rightsquigarrow^{*} [\sigma|x:-1]}\ \text{R3}$$

Ejercicios 7 - 8 - 9

Estos ejercicios fueron realizados en los archivos Eval1.hs, Eval2.hs y Eval3.hs respectivamente. Dichos códigos están presentes en los **Anexos 3, 4 y 5** al final de este trabajo.

Ejercicio 10

Inicialmente agregamos el comando **for** a la sintaxis abstracta de LIS, dando como resultado el tipo de comandos de la forma:

```
comm ::= skip
       | assignL
       | var = intexp
       | comm ; comm
       | if boolexp then comm
       | if boolexp then comm else comm
       | while boolexp do comm
       | for intexp boolexp intexp comm
```

Luego, extendemos la Semántica Operacional agregando las siguientes reglas:

For$_{init}$

$$\dfrac{\langle e1, \sigma\rangle \Downarrow_{\text{intexp}} \langle n, \sigma'\rangle}{\langle for\ e1\ b\ e3\ c, \sigma\rangle \rightsquigarrow \langle e1; for\ n\ b\ e3\ c, \sigma'\rangle}\ \textsc{For}_{init}$$

For$_{true}$

$$\dfrac{\langle b, \sigma\rangle \Downarrow_{\text{boolexp}} \langle true, \sigma'\rangle \qquad \langle c, \sigma'\rangle \rightsquigarrow \sigma'' \qquad \langle e3, \sigma''\rangle \Downarrow_{\text{intexp}} \langle n2, \sigma'''\rangle}{\langle for\ n\ b\ e3\ c, \sigma\rangle \rightsquigarrow \langle for\ n\ b\ e3\ c, \sigma'''\rangle}\ \textsc{For}_{true}$$

For$_{false}$

$$\dfrac{\langle b, \sigma\rangle \Downarrow_{\text{boolexp}} \langle false, \sigma'\rangle}{\langle for\ n\ b\ e3\ c, \sigma\rangle \rightsquigarrow \sigma'}\ \textsc{For}_{false}$$

Anexo - Semántica Operacional Estructural para Comandos

Anexamos la Semántica Operacional Estructural para Comandos, modificando el cambio de estado, y son las reglas que emplearemos desde el Ejercicio 6 en adelante.

$$\frac{\langle e, \sigma\rangle \Downarrow_{\text{intexp}} \langle n, \sigma'\rangle}{\langle v := e, \sigma\rangle \rightsquigarrow [\,\sigma' \mid v\colon n\,]}\ \textsc{Ass} \qquad \frac{}{\langle \textbf{skip}, \sigma\rangle \rightsquigarrow \sigma}\ \textsc{Skip}$$

$$\frac{\langle c_0, \sigma\rangle \rightsquigarrow \sigma'}{\langle c_0; c_1, \sigma\rangle \rightsquigarrow \langle c_1, \sigma'\rangle}\ \textsc{Seq}_1 \qquad \frac{\langle c_0, \sigma\rangle \rightsquigarrow \langle c_0', \sigma'\rangle}{\langle c_0; c_1, \sigma\rangle \rightsquigarrow \langle c_0'; c_1, \sigma'\rangle}\ \textsc{Seq}_2$$

$$\frac{\langle b, \sigma\rangle \Downarrow_{\text{boolexp}} \langle true, \sigma'\rangle}{\langle if\ b\ then\ c_0, \sigma\rangle \rightsquigarrow \langle c_0, \sigma'\rangle}\ \textsc{If}_1 \qquad \frac{\langle b, \sigma\rangle \Downarrow_{\text{boolexp}} \langle false, \sigma'\rangle}{\langle if\ b\ then\ c_0, \sigma\rangle \rightsquigarrow \sigma'}\ \textsc{If}_2$$

$$\frac{\langle b, \sigma\rangle \Downarrow_{\text{boolexp}} \langle true, \sigma'\rangle}{\langle if\ b\ then\ c_0\ else\ c_1, \sigma\rangle \rightsquigarrow \langle c_0, \sigma'\rangle}\ \textsc{If}_3 \qquad \frac{\langle b, \sigma\rangle \Downarrow_{\text{boolexp}} \langle false, \sigma'\rangle}{\langle if\ b\ then\ c_0\ else\ c_1, \sigma\rangle \rightsquigarrow \langle c_1, \sigma'\rangle}\ \textsc{If}_4$$

$$\frac{\langle b, \sigma\rangle \Downarrow_{\text{boolexp}} \langle true, \sigma'\rangle}{\langle while\ b\ do\ c, \sigma\rangle \rightsquigarrow \langle c; while\ b\ do\ c, \sigma'\rangle}\ \textsc{While}_1 \qquad \frac{\langle b, \sigma\rangle \Downarrow_{\text{boolexp}} \langle false, \sigma'\rangle}{\langle while\ b\ do\ c, \sigma\rangle \rightsquigarrow \sigma'}\ \textsc{While}_2$$

Anexo 2 - Parser.hs

```haskell
-- Solution will run in Haskell Interpreted Mode
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","then",
                                                       "else", "while", "skip"]
                                    , reservedOpNames = ["-", "+", "=","*", "/", "==",
                                                         "!=", "&&", "||", "<", ">", "!", ";"]
                                    })
```

```haskell
-----------------------------------
--- Common Parsers
------------------------------------
parenParse  :: Parser p -> Parser p
parenParse p = do { symbol lis "("
                 ; x <- p
                 ; symbol lis ")"
                 ; return x }

-----------------------------------
--- Parser de expressiones enteras
------------------------------------
opParseTerm = do  { reservedOp lis "+" ; return (Plus) }
              <|> do { reservedOp lis "-" ; return (Minus) }

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

negativeParse :: Parser IntExp
negativeParse = do { reservedOp lis "-"
                  ; b <- factorParse
                  ; return (UMinus b) }

constParse :: Parser IntExp
constParse  = do { f <- natural lis
                 ; return (Const f) }

varParse :: Parser IntExp
varParse  = do x<- identifier lis
               return (Var x)

intexp :: Parser IntExp
intexp  = chainl1 termParse opParseTerm

termParse :: Parser IntExp
termParse  = chainl1 factorParse opParseFactor

factorParse :: Parser IntExp
factorParse  = negativeParse
               <|> try constParse
               <|> try varParse
               <|> parenParse intexp

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

boolPrimitive :: Parser BoolExp
boolPrimitive  = do { f <- reserved lis "true"; return (BTrue) }
                 <|> do { f <- reserved lis "false"; return (BFalse) }

compOpParse = do  { reservedOp lis "==" ; return (Eq) }
```

```
                <|> do { reservedOp lis "!=" ; return (NEq) }
                <|> do { reservedOp lis "<" ; return (Lt) }
                <|> do { reservedOp lis ">" ; return (Gt) }

boolOpParse = do  { reservedOp lis "&&" ; return (And) }
                <|> do { reservedOp lis "||" ; return (Or) }

negationParse :: Parser BoolExp
negationParse  = do { reservedOp lis "!"
                    ; b <- bTermParse  -- ACA FALTA UN EQUIVALENTE A FACTOR
                    ; return (Not b) }

comparisonParse :: Parser BoolExp
comparisonParse  = do { x <- intexp
                      ; f <- compOpParse
                      ; y <- intexp
                      ; return (f x y)}


boolexp :: Parser BoolExp
boolexp  = chainl1 bTermParse boolOpParse

bTermParse :: Parser BoolExp
bTermParse  = negationParse
                <|> try boolPrimitive
                <|> parenParse boolexp
                <|> comparisonParse

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

dacParse  = do  { reservedOp lis ";" ; return (Seq) }

skipParse :: Parser Comm
skipParse  = do { reserved lis "skip"
               ; return (Skip)}

letParse :: Parser Comm
letParse  = do { name <- identifier lis
               ; reservedOp lis "="
               ; value <- intexp
               ; return (Let name value)}

ifParse :: Parser Comm
ifParse  = do { reserved lis "if"
              ; cond <- boolexp
              ; reserved lis "then"
              ; symbol lis "{"
              ; thencmd <- comm
              ; symbol lis "}"
              ; elsecmd <- elseParse
              ; return (IfThenElse cond thencmd elsecmd)}
```

```haskell
elseParse :: Parser Comm
elseParse  = do { try (do { reserved lis "else"
                          ; symbol lis "{"
                          ; elsecmd <- comm
                          ; symbol lis "}"
                          ; return elsecmd})
                <|> return Skip}

whileParse :: Parser Comm
whileParse  = do { reserved lis "while"
                 ; cond <- boolexp
                 ; symbol lis "{"
                 ; cmd <- comm
                 ; symbol lis "}"
                 ; return (While cond cmd)}

commLine :: Parser Comm
commLine  = skipParse
            <|> ifParse
            <|> whileParse
            <|> try letParse

comm :: Parser Comm
comm = chainr1 commLine dacParse

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

Anexo 3 - Eval1.hs

```haskell
module Eval1 (eval) where

import AST

-- Estados
type State = [(Variable,Integer)]

-- Estado nulo
initState :: State
initState = [("t",0),("n",0),("i",0)]

-- Busca el valor de una variabl en un estado
-- Completar la definicion
lookfor :: Variable -> State -> Integer
lookfor var state = snd $ head $ filter (\(v,i) -> v==var) state

-- Cambia el valor de una variable en un estado
-- Completar la definicion
update :: Variable -> Integer -> State -> State
```

```haskell
update var int state = map (\(v,i) -> if (v == var) then (var, int) else (v,i)) state

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

-- Evalua un comando en un estado dado
evalComm :: Comm -> State -> State
evalComm Skip                 s = s
evalComm (Let v i)            s = update v (fst $ evalIntExp i s) s
evalComm (Seq x y)            s = let s' = evalComm x s
                                  in evalComm y s'
evalComm (IfThenElse b x y) s = case (evalBoolExp b s) of
                                  (True, s') -> evalComm x s'
                                  (_ ,   s') -> evalComm y s'
evalComm (While b c)          s = case (evalBoolExp b s) of
                                  (True, s') -> let s2 = evalComm c s'
                                                in evalComm (While b c) s2
                                  (_,    s') -> s'

-- Evalua una expresion entera, con efectos laterales

binop :: (Integer -> Integer -> b) -> IntExp -> IntExp -> State -> (b, State)
binop f x y s = let (x', s1) = evalIntExp x s
                    (y', s2) = evalIntExp y s1
                in (f x' y' , s2)

evalIntExp :: IntExp -> State -> (Integer, State)
evalIntExp (Const c)       s = (c, s)
evalIntExp (Var x)         s = (lookfor x s, s)
evalIntExp (UMinus x)      s = binop (*) x (Const (-1)) s
evalIntExp (Plus  x y)     s = binop (+) x y s
evalIntExp (Minus x y)     s = binop (-) x y s
evalIntExp (Times x y)     s = binop (*) x y s
evalIntExp (Div   x y)     s = binop div x y s

evalIntExp (Assign v i)    s = let (i' , s1) = evalIntExp i s
                               in (i', update v i' s1)
evalIntExp (Secuence x y) s = let (x', s') = evalIntExp x s
                               in evalIntExp y s'


-- Evalua una expresion booleana, con efectos laterales

binopBool :: (Bool -> Bool -> b) -> BoolExp -> BoolExp -> State -> (b, State)
binopBool f x y s = let (x', s1) = evalBoolExp x s
                        (y', s2) = evalBoolExp y s1
                    in (f x' y' , s2)

evalBoolExp :: BoolExp -> State -> (Bool, State)
evalBoolExp BTrue      s = (True,s)
evalBoolExp BFalse     s = (False,s)
evalBoolExp (Eq x y)   s = binop (==) x y s
```

```haskell
evalBoolExp (NEq x y) s = binop (/=) x y s
evalBoolExp (Lt x y)  s = binop (<) x y s
evalBoolExp (Gt x y)  s = binop (>) x y s
evalBoolExp (And x y) s = binopBool (&&) x y s
evalBoolExp (Or x y)  s = binopBool (||) x y s
evalBoolExp (Not x)   s = let (b, s') = (evalBoolExp x s)
                          in (not b, s')
```

Anexo 4 - Eval2.hs

```haskell
module Eval2 (eval) where

import AST

-- Estados
type State = [(Variable,Integer)]

-- Error
data Error = DivByZero | UndefVar deriving Show

-- Estado nulo
initState :: State
initState = [("t",0),("n",0),("i",0),("a",0)]

-- Busca el valor de una variabl en un estado
-- Completar la definicion
lookfor :: Variable -> State -> Either Error Integer
lookfor var state = let l = filter (\(v,i) -> v==var) state
                    in if (length l == 0) then Left UndefVar
                                          else Right $ snd $ head l

-- Cambia el valor de una variable en un estado
-- Completar la definicion
update :: Variable -> Integer -> State -> Either Error State
update var int state = case (lookfor var state) of
        Right _    -> Right $ map (\(v,i) -> if (v == var) then (var, int) else (v,i)) state
        otherwise -> Left UndefVar

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

-- Evalua un comando en un estado dado
-- Completar definicion
continue :: Comm -> Either Error State -> Either Error State
continue cmd (Left err) = Left err
continue cmd (Right s)  = evalComm cmd s

evalComm :: Comm -> State -> Either Error State
evalComm Skip               s = Right s
evalComm (Let v i)          s = case (evalIntExp i s) of
        Left err -> Left err
        Right (i', s') -> update v i' s'
```

```haskell
evalComm (Seq x y)          s = case (evalComm x s) of
        Left err -> Left err
        Right s' -> evalComm y s'
evalComm (IfThenElse b x y) s = case (evalBoolExp b s) of
        Left err -> Left err
        Right (True, s') -> evalComm x s'
        Right (False, s') -> evalComm y s'
evalComm (While b c)        s = case (evalBoolExp b s) of
        Left err -> Left err
        Right (True, s') -> continue (While b c) $ evalComm c s'
        Right (False, s') -> Right s'

-- Evalua una expresion entera, con efectos laterales
-- Completar definicion

binop :: (Integer -> Integer -> b) -> IntExp -> IntExp -> State -> Either Error (b, State)
binop f x y s = case (evalIntExp x s) of
        Left err       -> Left err
        Right (x', s') -> binop' f x' y s'

binop' :: (Integer -> Integer -> b) -> Integer -> IntExp -> State -> Either Error (b, State)
binop' f x y s = case (evalIntExp y s) of
        Left err       -> Left err
        Right (y', s') -> Right (f x y', s')

binopDiv :: IntExp -> IntExp -> State -> Either Error (Integer, State)
binopDiv x y s = case (evalIntExp x s) of
        Left err       -> Left err
        Right (x', s') -> binopDiv' x' y s'

binopDiv' x y s = case (evalIntExp y s) of
        Left err       -> Left err
        Right (0, s')  -> Left DivByZero
        Right (y', s') -> Right (div x y', s')


evalIntExp :: IntExp -> State -> Either Error (Integer, State)
evalIntExp (Const c)      s = Right (c, s)
evalIntExp (Var x)        s = case (lookfor x s) of
        Right x'  -> Right (x', s)
        otherwise -> Left UndefVar
evalIntExp (UMinus x)     s = binop (*) x (Const (-1)) s
evalIntExp (Plus  x y)    s = binop (+) x y s
evalIntExp (Minus x y)    s = binop (-) x y s
evalIntExp (Times x y)    s = binop (*) x y s
evalIntExp (Div   x y)    s = binopDiv x y s
evalIntExp (Assign v i)   s = case evalIntExp i s of
        Left err -> Left err
        Right (val, s1) -> case (update v val s1) of
                Left err -> Left err
                Right s2 -> Right (val, s2)
evalIntExp (Secuence x y) s = case (evalIntExp x s) of
        Left err -> Left err
```

```haskell
        Right (x', s') -> evalIntExp y s'

-- Evalua una expresion entera, sin efectos laterales
-- Completar definicion

binopBool :: (Bool -> Bool -> b) -> BoolExp -> BoolExp -> State -> Either Error (b, State)
binopBool f x y s = case (evalBoolExp x s) of
        Left err -> Left err
        Right (x', s') -> binopBool' f x' y s'

binopBool' :: (Bool -> Bool -> b) -> Bool -> BoolExp -> State -> Either Error (b, State)
binopBool' f x y s = case (evalBoolExp y s) of
        Left err -> Left err
        Right (y', s') -> Right (f x y', s')

evalBoolExp :: BoolExp -> State -> Either Error (Bool, State)
evalBoolExp BTrue     s = Right (True, s)
evalBoolExp BFalse    s = Right (False, s)
evalBoolExp (Eq x y)  s = binop (==) x y s
evalBoolExp (NEq x y) s = binop (/=) x y s
evalBoolExp (Lt x y)  s = binop (<) x y s
evalBoolExp (Gt x y)  s = binop (>) x y s
evalBoolExp (And x y) s = binopBool (&&) x y s
evalBoolExp (Or x y)  s = binopBool (||) x y s
evalBoolExp (Not x)   s = case (evalBoolExp x s) of
        Left err -> Left err
        Right (b, s') -> Right (not b, s')
```

Anexo 5 - Eval3.hs

```haskell
module Eval3 (eval) where

import AST

-- Estados
type State = [(Variable,Integer)]

-- Trabajo
type Work = Integer

-- Error
data Error = DivByZero | UndefVar deriving Show

-- Estado nulo
initState :: State
initState = [("t",0),("n",0),("i",0),("a",0)]

-- Busca el valor de una variabl en un estado
-- Completar la definicion
lookfor :: Variable -> State -> Either Error Integer
lookfor var state = let l = filter (\(v,i) -> v==var) state
                    in if (length l == 0) then Left UndefVar
                                          else Right $ snd $ head l
```

```haskell
-- Cambia el valor de una variable en un estado
-- Completar la definicion
update :: Variable -> Integer -> State -> Either Error State
update var int state = case (lookfor var state) of
        Right _   -> Right $ map (\(v,i) -> if (v == var) then (var, int) else (v,i)) state
        otherwise -> Left UndefVar

-- Evalua un programa en el estado nulo
eval :: Comm -> Either Error (Work, State)
eval p = evalComm p (0, initState)

-- Evalua un comando en un estado dado
-- Completar definicion
continue :: Comm -> Either Error (Work, State) -> Either Error (Work, State)
continue cmd (Left err)      = Left err
continue cmd (Right (w,s))  = evalComm cmd (w, s)

evalComm :: Comm -> (Work, State) -> Either Error (Work, State)
evalComm Skip              (w, s) = Right (w, s)
evalComm (Let v i)         (w, s) = case (evalIntExp i s) of
        Left err              -> Left err
        Right (w', (i', s')) -> let ret = update v i' s'
                                in calcwork (w+w') ret
                where calcwork _ (Left err) = Left err
                      calcwork w (Right s)  = Right (w, s)
evalComm (Seq x y)         (w, s) = case (evalComm x (w, s)) of
        Left err        -> Left err
        Right (w', s') -> evalComm y (w', s')
evalComm (IfThenElse b x y) (w, s) = case (evalBoolExp b s) of
        Left err                 -> Left err
        Right (w', (True, s'))  -> evalComm x (w+w', s')
        Right (w', (False, s')) -> evalComm y (w+w', s')
evalComm (While b c)        (w, s) = case (evalBoolExp b s) of
        Left err                 -> Left err
        Right (w', (True, s'))  -> continue (While b c) $ evalComm c (w+w', s')
        Right (w', (False, s')) -> Right (w+w', s')

-- Evalua una expresion entera, con efectos laterales
-- Completar definicion

binop :: (Integer -> Integer -> b) -> IntExp -> IntExp
         -> State -> Either Error (Work, (b, State))
binop f x y s = case (evalIntExp x s) of
        Left err        -> Left err
        Right (w1, (x', s')) -> binop' w1 f x' y s'

binop' :: Work -> (Integer -> Integer -> b) -> Integer -> IntExp
          -> State -> Either Error (Work, (b, State))
binop' w1 f x y s = case (evalIntExp y s) of
        Left err        -> Left err
        Right (w2, (y', s')) -> Right (w1+w2+1, (f x y', s'))
```

```haskell
binopMul :: IntExp -> IntExp -> State -> Either Error (Work, (Integer, State))
binopMul x y s = case (evalIntExp x s) of
        Left err       -> Left err
        Right (w1, (x', s')) -> binopMul' w1 x' y s'

binopMul' :: Work -> Integer -> IntExp -> State -> Either Error (Work, (Integer, State))
binopMul' w1 x y s = case (evalIntExp y s) of
        Left err       -> Left err
        Right (w2, (y', s')) -> Right (w1+w2+2, ( x * y', s'))

binopDiv :: IntExp -> IntExp -> State -> Either Error (Work, (Integer, State))
binopDiv x y s = case (evalIntExp x s) of
        Left err       -> Left err
        Right (w1, (x', s')) -> binopDiv' w1 x' y s'

binopDiv' w1 x y s = case (evalIntExp y s) of
        Left err       -> Left err
        Right (_, (0, s'))  -> Left DivByZero
        Right (w2, (y', s')) -> Right (w1+w2+2, (div x y', s'))

evalIntExp :: IntExp -> State -> Either Error (Work, (Integer, State))
evalIntExp (Const c)      s = Right (0, (c, s))
evalIntExp (Var x)        s = case (lookfor x s) of
        Right x'  -> Right (0, (x', s))
        otherwise -> Left UndefVar
evalIntExp (UMinus x)     s = binop (*) x (Const (-1)) s
evalIntExp (Plus  x y)    s = binop (+) x y s
evalIntExp (Minus x y)    s = binop (-) x y s
evalIntExp (Times x y)    s = binopMul x y s
evalIntExp (Div   x y)    s = binopDiv x y s
evalIntExp (Assign v i)   s = case evalIntExp i s of
        Left err -> Left err
        Right (w, (val, s1)) -> case (update v val s1) of
                Left err -> Left err
                Right s2 -> Right (w, (val, s2))
evalIntExp (Secuence x y) s = case (evalIntExp x s) of
        Left err -> Left err
        Right (w, (x', s')) -> let ret = evalIntExp y s'
                               in calcwork w ret
                where calcwork w1 (Left err)             = Left err
                      calcwork w1 (Right (w2, (x2, s2))) = Right (w1+w2,(x2, s2))
-- Evalua una expresion entera, sin efectos laterales
-- Completar definicion

binopBool :: (Bool -> Bool -> b) -> BoolExp -> BoolExp
             -> State -> Either Error (Work, (b, State))
binopBool f x y s = case (evalBoolExp x s) of
        Left err -> Left err
        Right (w1, (x', s')) -> binopBool' w1 f x' y s'

binopBool' :: Work -> (Bool -> Bool -> b) -> Bool -> BoolExp
              -> State -> Either Error (Work, (b, State))
binopBool' w1 f x y s = case (evalBoolExp y s) of
```

```
        Left err -> Left err
        Right (w2, (y', s')) -> Right (w1+w2+1, (f x y', s'))

evalBoolExp :: BoolExp -> State -> Either Error (Work, (Bool, State))
evalBoolExp BTrue      s = Right (0, (True, s))
evalBoolExp BFalse     s = Right (0, (False, s))
evalBoolExp (Eq x y)   s = binop (==) x y s
evalBoolExp (NEq x y)  s = binop (/=) x y s
evalBoolExp (Lt x y)   s = binop (<) x y s
evalBoolExp (Gt x y)   s = binop (>) x y s
evalBoolExp (And x y)  s = binopBool (&&) x y s
evalBoolExp (Or x y)   s = binopBool (||) x y s
evalBoolExp (Not x)    s = case (evalBoolExp x s) of
        Left err -> Left err
        Right (w, (b, s')) -> Right (w+1, (not b, s'))
```