



R-313 ANÁLISIS DE LENGUAJES DE PROGRAMACIÓN

TRABAJO PRÁCTICO 1

Gianni Georg Weinand W-0528/2
Margarita Capretto C-6055/1

28 de Septiembre, 2018

1. Ejercicio 1

1.1. Sintaxis Abstracta

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

1.2. Sintaxis Concreta

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

2. Ejercicio 2

```
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
            | Tern BoolExp IntExp IntExp
            deriving Show
```

```

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

deriving Show

-- Comandos (sentencias)
-- Observar que solo se permiten variables de un tipo
-- (entero)
data Comm = Skip
          | Let Variable IntExp
          | Seq Comm Comm
          | Cond BoolExp Comm Comm
          | Repeat Comm BoolExp

deriving Show

```

3. Ejercicio 3

```

module Parser where

import Text.ParserCombinators.Parsec
import Text.ParserCombinators.Parsec.Expr
import Text.ParserCombinators.Parsec.Language
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 '='
, reservedOpNames = [":="]
, reservedNames =
    ["true","false","skip","if",
      "then","else","end",
      "while","do",
      "repeat",
      "until"]
})

```

----- --- Operadores -----

```

iOperators = [ [Prefix (reservedOp lis "-" >> return
  (UMinus))
  ], [Infix (reservedOp lis "*" >> return
    (Times)) AssocLeft,
    Infix (reservedOp lis "/" >> return (Div))
    AssocLeft]
  , [Infix (reservedOp lis "+" >> return
    (Plus)) AssocLeft,
    Infix (reservedOp lis "-" >> return
    (Minus)) AssocLeft]
  ]

bOperators = [ [Prefix (reservedOp lis "~" >> return (Not))
  ]
  , [Infix (reservedOp lis "&" >> return (And))
    AssocLeft,
    Infix (reservedOp lis "|" >> return (Or))
    AssocLeft]
  ]

```

----- --- Terminos -----

```

iTerm :: Parser IntExp
iTerm = try (parens lis intexp)
  <|> const
  <|> var
  <|> tern
  where const = do i <- integer lis
    return $ Const i
    var      = do id <- identifier lis
    return $ Var id
    tern     = do b <- boolexp
    i1 <- intexp
    i2 <- intexp

```

```

                                return $ Tern b i1 i2

bTerm :: Parser BoolExp
bTerm = try (parens lis boolexp)
      <|> btrue
      <|> bfalse
      <|> relexp
  where btrue  = (reserved lis "true"  >> return BTrue)
        bfalse = (reserved lis "false" >> return BFalse)

-----
--- Parser de expresiones enteras
-----

intexp :: Parser IntExp
intexp = buildExpressionParser i0operators iTerm

-----
--- Parser de expresiones de relacion
-----

relexp :: Parser BoolExp
relexp = do i1 <- intexp
          op <- relOp
          i2 <- intexp
          return $ op i1 i2
  where relOp = (reservedOp lis "=" >> return (Eq))
               <|> (reservedOp lis "<" >> return (Lt))
               <|> (reservedOp lis ">" >> return (Gt))

-----
--- Parser de expresiones booleanas
-----

boolexp :: Parser BoolExp
boolexp = buildExpressionParser b0operators bTerm

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

comm1 :: Parser Comm
comm1 = skip
      <|> ass
      <|> cond
      <|> repeat
  where skip  = reserved lis "skip" >> return Skip
        ass   = do v <- identifier lis
                  reservedOp lis ":@"
                  e <- intexp
                  return $ Let v e

```

```

cond      = do reserved lis "if"
              b <- boolexp
              reserved lis "then"
              c1 <- comm
              reserved lis "else"
              c2 <- comm
              reserved lis "end"
              return $ Cond b c1 c2
repeat = do reserved lis "repeat"
              c <- comm
              reserved lis "until"
              b <- boolexp
              reserved lis "end"
              return $ Repeat c b

comm :: Parser Comm
comm = do list <- sepBy comm1 $semi lis;
        return $ seqC list
      where seqC [x]      = x
            seqC (x:xs) = Seq x $seqC xs

-----
-- Funcion de parseo
-----

parseComm :: SourceName -> String -> Either ParseError Comm
parseComm = parse (totParser comm)

```

4. Ejercicio 4

$$\frac{\langle p, \sigma \rangle \Downarrow_{\text{boolexp}} \mathbf{true} \quad \langle e_0, \sigma \rangle \Downarrow_{\text{intexp}} n_0}{\langle p ? e_0 : e_1, \sigma \rangle \Downarrow_{\text{intexp}} n_0} \text{TERN}_1$$

$$\frac{\langle p, \sigma \rangle \Downarrow_{\text{boolexp}} \mathbf{false} \quad \langle e_1, \sigma \rangle \Downarrow_{\text{intexp}} n_1}{\langle p ? e_0 : e_1, \sigma \rangle \Downarrow_{\text{intexp}} n_1} \text{TERN}_2$$

5. Ejercicio 5

Sea $\langle t, \sigma \rangle \rightsquigarrow \langle t_1, \sigma_1 \rangle$ y $\langle t, \sigma \rangle \rightsquigarrow \langle t_2, \sigma_2 \rangle$. Se debe probar que $\langle t_1, \sigma_1 \rangle = \langle t_2, \sigma_2 \rangle$.

Se demostrará por inducción sobre la derivación de $\langle t_1, \sigma_1 \rangle$, teniendo en cuenta la última regla utilizada.

Si es la regla ASS, t es de la forma $v := e$. Luego

- $\langle e, \sigma \rangle \Downarrow_{intexp} n$, para algún n .
- $t_1 = \mathbf{skip}$ y $\sigma_1 = [\sigma|v : n]$.
- la última regla en la derivación de $\langle t_2, \sigma_2 \rangle$ no pudo ser SEQ₁, SEQ₂, IF₁, IF₂ ni REPEAT por la estructura de t . Por lo tanto, la última regla aplicada debió ser ASS. Entonces, usando que \Downarrow_{intexp} es determinista, se tiene que $t_2 = \mathbf{skip} = t_1$ y $\sigma_2 = [\sigma|v : n] = \sigma_1$.

Si es la regla SEQ₁, t es de la forma $\mathbf{skip}; c$. Luego

- $t_1 = c$ y $\sigma_1 = \sigma$.
- la última regla en la derivación de $\langle t_2, \sigma_2 \rangle$ no pudo ser ASS, IF₁, IF₂ ni REPEAT por la estructura de t . Pero tampoco pudo ser SEQ₂, ya que no hay una regla de derivación para \mathbf{skip} . Por lo tanto, la última regla aplicada debió ser también SEQ₁, entonces $t_2 = c = t_1$ y $\sigma_2 = \sigma = \sigma_1$.

Si es la regla SEQ₂, t es de la forma $c_0; c_1$. Luego

- $t_1 = c'_0; c_1$ y $\sigma_1 = \sigma'$.
- la última regla en la derivación de $\langle t_2, \sigma_2 \rangle$ no pudo ser ASS, IF₁, IF₂ ni REPEAT por la estructura de t . Por hipótesis inductiva se tiene que el paso $\langle c_0, \sigma \rangle \rightsquigarrow \langle c'_0, \sigma' \rangle$ es determinista. Se tiene entonces que c_0 no puede ser \mathbf{skip} y consecuentemente la última regla en la derivación de $\langle t_2, \sigma_2 \rangle$ no pudo ser SEQ₁. La única posibilidad es que haya sido también SEQ₂ y se concluye $t_2 = c'_0; c_1 = t_1$ y $\sigma_2 = \sigma' = \sigma_1$.

Si es la regla IF₁, t es de la forma $\mathbf{if } b \mathbf{ then } c_0 \mathbf{ else } c_1$. Luego

- $t_1 = c_0$ y $\sigma_1 = \sigma$.
- la última regla en la derivación de $\langle t_2, \sigma_2 \rangle$ no pudo ser ASS, SEQ₁, SEQ₂ ni REPEAT por la estructura de t . Por determinismo de $\Downarrow_{boolexp}$ se tiene que $b \Downarrow_{boolexp} \mathbf{true}$, lo que vuelve imposible el uso de IF₂. La única posibilidad es que haya sido también IF₁ y se concluye $t_2 = c_0 = t_1$ y $\sigma_2 = \sigma = \sigma_1$.

Si es la regla IF₂, la prueba es análoga a IF₁.

Si es la regla REPEAT, t es de la forma $\mathbf{repeat } c \mathbf{ until } b$. Luego

- $t_1 = c; \mathbf{if } b \mathbf{ then skip else repeat } c \mathbf{ until } b$ y $\sigma_1 = \sigma$.
- la última regla en la derivación de $\langle t_2, \sigma_2 \rangle$ no pudo ser ASS, SEQ₁, SEQ₂, IF₁ ni IF₂ por la estructura de t . Por lo tanto, la última regla aplicada debió ser también REPEAT, entonces $t_2 = c; \mathbf{if } b \mathbf{ then skip else repeat } c \mathbf{ until } b = t_1$ y $\sigma_2 = \sigma = \sigma_1$.

6. Ejercicio 6

Sean $\sigma_0 = [\sigma|x : 0]$ y $\sigma_1 = [\sigma|x : 1]$. Por definición es claro que $\sigma_1 = [\sigma_0|x : 1]$.

Sean además $c_0 = x := x + 1, b_0 = x > 0, c_1 = x := x - 1,$

$c_2 = \text{if } b_0 \text{ then skip else } c_1.$

$$\frac{t \rightsquigarrow t'}{t \rightsquigarrow^* t'}$$

Dado que $t \rightsquigarrow^* t'$, se usará solamente \rightsquigarrow^* para acortar la prueba.

6.1. Árbol 1

$$\frac{\frac{\frac{\langle x, \sigma_0 \rangle \Downarrow_{\text{intexp}} 0 \text{ VAR} \quad \langle 1, \sigma_0 \rangle \Downarrow_{\text{intexp}} 1 \text{ NVAL}}{\langle x + 1, \sigma_0 \rangle \Downarrow_{\text{intexp}} 1 \text{ PLUS}}}{\langle x := x + 1, \sigma_0 \rangle \rightsquigarrow^* \langle \text{skip}, \sigma_1 \rangle \text{ ASS}}$$

6.2. Árbol 2

$$\frac{\frac{\langle x := x + 1, \sigma_0 \rangle \rightsquigarrow^* \langle \text{skip}, \sigma_1 \rangle \text{ ARBOL 1}}{\langle c_0; c_2, \sigma_0 \rangle \rightsquigarrow^* \langle \text{skip}; c_2, \sigma_1 \rangle \text{ SEQ}_2} \quad \frac{\langle \text{skip}; c_2, \sigma_1 \rangle \rightsquigarrow^* \langle c_2, \sigma_1 \rangle \text{ SEQ}_1}{\langle c_0; c_2, \sigma_0 \rangle \rightsquigarrow^* \langle c_2, \sigma_1 \rangle \text{ TRANSITIVIDAD}}$$

6.3. Árbol 3

$$\frac{\frac{\frac{\frac{\langle x, \sigma_1 \rangle \Downarrow_{\text{intexp}} 1 \text{ VAR} \quad \langle 0, \sigma_1 \rangle \Downarrow_{\text{intexp}} 0 \text{ NVAL}}{\langle b_0, \sigma_1 \rangle \Downarrow_{\text{boolexp}} \text{true} \text{ GT}}}{\langle c_2, \sigma_1 \rangle \rightsquigarrow^* \langle \text{skip}, \sigma_1 \rangle \text{ IF}_1}}{\langle c_0; c_2, \sigma_0 \rangle \rightsquigarrow^* \langle c_2, \sigma_1 \rangle \text{ ARBOL 2}} \quad \frac{\langle c_2, \sigma_1 \rangle \rightsquigarrow^* \langle \text{skip}, \sigma_1 \rangle \text{ TRANSITIVIDAD}}{\langle c_0; c_2, \sigma_0 \rangle \rightsquigarrow^* \langle \text{skip}, \sigma_1 \rangle}$$

7. Ejercicio 7

```
module Eval1 (eval) where

import AST

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

-- Estado nulo
initState :: State
initState = []

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



```

lookfor var ((v,i):xs)
  | v == var = i
  | otherwise = lookfor var xs

-- Cambia el valor de una variable en un estado
-- Completar la definicion
update :: Variable -> Integer -> State -> State
update var int [] = [(var, int)]
update var int ((v,i):xs)
  | v == var = (var, int):xs
  | otherwise = (v,i) : (update var int xs)

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

-- Evalua un comando en un estado dado
-- Completar definicion
evalComm :: Comm -> State -> State
evalComm Skip state = state
evalComm (Let var int) state = update var (evalIntExp int
  state) state
evalComm (Seq com1 com2) state = evalComm com2 $ evalComm
  com1 state
evalComm (Cond bool com1 com2) state
  | evalBoolExp bool state = evalComm com1 state
  | otherwise = evalComm com2 state
evalComm rep@(Repeat com bool) state = evalComm (Seq com
  (Cond bool Skip rep))state

-- Evalua una expresion entera, sin efectos laterales
-- Completar definicion
evalIntExp :: IntExp -> State -> Integer
evalIntExp (Const int) _state = int
evalIntExp (Var var) state = lookfor var state
evalIntExp (UMinus int) state = negate $ evalIntExp int
  state
evalIntExp (Plus int1 int2) state = (evalIntExp int1 state)
  + (evalIntExp int2 state)
evalIntExp (Minus int1 int2) state = (evalIntExp int1
  state) - (evalIntExp int2 state)
evalIntExp (Times int1 int2) state = (evalIntExp int1
  state) * (evalIntExp int2 state)
evalIntExp (Div int1 int2) state = (evalIntExp int1 state)
  `div` (evalIntExp int2 state)
evalIntExp (Tern bool int1 int2) state
  | evalBoolExp bool state = evalIntExp int1 state
  | otherwise = evalIntExp int2 state

```

```

-- Evalua una expresion entera, sin efectos laterales
-- Completar definicion
evalBoolExp :: BoolExp -> State -> Bool
evalBoolExp BTrue _state      = True
evalBoolExp BFalse _state     = False
evalBoolExp (Eq int1 int2) state = (evalIntExp int1
    state) == (evalIntExp int2 state)
evalBoolExp (Lt int1 int2) state = (evalIntExp int1
    state) < (evalIntExp int2 state)
evalBoolExp (Gt int1 int2) state = (evalIntExp int1
    state) > (evalIntExp int2 state)
evalBoolExp (And bool1 bool2) state = (evalBoolExp bool1
    state) && (evalBoolExp bool2 state)
evalBoolExp (Or bool1 bool2) state = (evalBoolExp bool1
    state) || (evalBoolExp bool2 state)
evalBoolExp (Not bool) state      = not (evalBoolExp
    bool state)

```

8. Ejercicio 8

```

module Eval2 (eval) where

import AST

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

-- Errores
data Error = DivByZero | UndefVar Variable deriving Show

-- Estado nulo
initState :: State
initState = []

-- Busca el valor de una variable en un estado
-- Completar la definicion
lookfor :: Variable -> State -> Either Error Integer
lookfor var [] = Left $ UndefVar var
lookfor var ((v,i):xs)
    | v == var = Right i
    | otherwise = lookfor var xs

-- Cambia el valor de una variable en un estado
-- Completar la definicion
update :: Variable -> Integer -> State -> State
update var int [] = [(var, int)]
update var int ((v,i):xs)
    | v == var = (var, int):xs

```

```

    | otherwise = (v,i) : (update var int xs)

-- 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
evalComm :: Comm -> State -> Either Error State
evalComm Skip state = Right state
evalComm (Let var int) state =
    case (evalIntExp int state) of
        Left err -> Left err
        Right val -> Right $ update var val state
evalComm (Seq com1 com2) state =
    case evalComm com1 state of
        Left err -> Left err
        Right st -> evalComm com2 st
evalComm (Cond bool com1 com2) state =
    case evalBoolExp bool state of
        Left err -> Left err
        Right True -> evalComm com1 state
        Right False -> evalComm com2 state
evalComm rep@(Repeat com bool) state = evalComm (Seq com
    (Cond bool Skip rep)) state

-- Evalua una expresion entera, sin efectos laterales
-- Completar definicion
evalBinOp :: (a -> a -> b) -> Either Error a -> Either
    Error a -> Either Error b
evalBinOp _op (Left err) _e = Left err
evalBinOp _op _e (Left err) = Left err
evalBinOp op (Right e1) (Right e2) = Right $ op e1 e2

evalDiv :: Either Error Integer -> Either Error Integer ->
    Either Error Integer
evalDiv (Left err) _e = Left err
evalDiv _e (Left err) = Left err
evalDiv _e (Right 0) = Left DivByZero
evalDiv (Right e1) (Right e2) = Right $ div e1 e2

evalUniOp :: (a -> a) -> Either Error a -> Either Error a
evalUniOp _op (Left err) = Left err
evalUniOp op (Right e) = Right $ op e

evalIntExp :: IntExp -> State -> Either Error Integer
evalIntExp (Const int) _state = Right int
evalIntExp (Var var) state = lookfor var state

```

```

evalIntExp (UMinus int) state = evalUniOp negate $
    evalIntExp int state
evalIntExp (Plus int1 int2) state = evalBinOp (+)
    (evalIntExp int1 state) (evalIntExp int2 state)
evalIntExp (Minus int1 int2) state = evalBinOp (-)
    (evalIntExp int1 state) (evalIntExp int2 state)
evalIntExp (Times int1 int2) state = evalBinOp (*)
    (evalIntExp int1 state) (evalIntExp int2 state)
evalIntExp (Div int1 int2) state = evalDiv (evalIntExp int1
    state) (evalIntExp int2 state)
evalIntExp (Tern bool int1 int2) state =
    case evalBoolExp bool state of
        Left err    -> Left err
        Right True  -> evalIntExp int1 state
        Right False -> evalIntExp int2 state

-- Evalua una expresion entera, sin efectos laterales
-- Completar definicion
evalBoolExp :: BoolExp -> State -> Either Error Bool
evalBoolExp BTrue _state      = Right True
evalBoolExp BFalse _state     = Right False
evalBoolExp (Eq int1 int2) state = evalBinOp (==)
    (evalIntExp int1 state) (evalIntExp int2 state)
evalBoolExp (Lt int1 int2) state = evalBinOp (<)
    (evalIntExp int1 state) (evalIntExp int2 state)
evalBoolExp (Gt int1 int2) state = evalBinOp (>)
    (evalIntExp int1 state) (evalIntExp int2 state)
evalBoolExp (And bool1 bool2) state = evalBinOp (&&)
    (evalBoolExp bool1 state) (evalBoolExp bool2 state)
evalBoolExp (Or bool1 bool2) state = evalBinOp (||)
    (evalBoolExp bool1 state) (evalBoolExp bool2 state)
evalBoolExp (Not bool) state = evalUniOp not
    (evalBoolExp bool state)

```

9. Ejercicio 9

```

module Eval3 (eval) where

import AST

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

-- Errores
data Error = DivByZero | UndefVar Variable deriving Show

```

```

-- Estado nulo
initState :: State
initState = []

-- Traza nula
initTrace :: Trace
initTrace = []

-- Busca el valor de una variable en un estado
-- Completar la definicion
lookfor :: Variable -> State -> Either Error Integer
lookfor var [] = Left $ UndefVar var
lookfor var ((v,i):xs)
    | v == var = Right i
    | otherwise = lookfor var xs

-- Cambia el valor de una variable en un estado
-- Completar la definicion
update :: Variable -> Integer -> State -> State
update var int [] = [(var, int)]
update var int ((v,i):xs)
    | v == var = (var, int):xs
    | otherwise = (v,i) : (update var int xs)

-- Evalua un programa en el estado nulo
eval :: Comm -> Either Error StateTrace
eval p = case evalComm p (initState,initTrace) of
    Left err -> Left err
    Right (st,tr) -> Right (st, reverse tr)

-- Evalua un comando en un estado dado
-- Completar definicion
evalComm :: Comm -> StateTrace -> Either Error StateTrace
evalComm Skip stateTrace = Right stateTrace
evalComm (Let var int) (state,trace) =
    case (evalIntExp int state) of
        Left err -> Left err
        Right val -> Right (update var val state, (Let var
            $ Const val) : trace)
evalComm (Seq com1 com2) stateTrace =
    case evalComm com1 stateTrace of
        Left err -> Left err
        Right st -> evalComm com2 st
evalComm (Cond bool com1 com2) stateTrace@(state,trace) =
    case evalBoolExp bool state of
        Left err -> Left err
        Right True -> evalComm com1 stateTrace
        Right False -> evalComm com2 stateTrace

```

```

evalComm rep@(Repeat com bool) stateTrace = evalComm (Seq
    com (Cond bool Skip rep)) stateTrace

-- Evalua una expresion entera, sin efectos laterales
-- Completar definicion
evalBinOp :: (a -> a -> b) -> Either Error a -> Either
    Error a -> Either Error b
evalBinOp _op (Left err) _e = Left err
evalBinOp _op _e (Left err) = Left err
evalBinOp op (Right e1) (Right e2) = Right $ op e1 e2

evalDiv :: Either Error Integer -> Either Error Integer ->
    Either Error Integer
evalDiv (Left err) _e = Left err
evalDiv _e (Left err) = Left err
evalDiv _e (Right 0) = Left DivByZero
evalDiv (Right e1) (Right e2) = Right $ div e1 e2

evalUniOp :: (a -> a) -> Either Error a -> Either Error a
evalUniOp _op (Left err) = Left err
evalUniOp op (Right e) = Right $ op e

evalIntExp :: IntExp -> State -> Either Error Integer
evalIntExp (Const int) _state = Right int
evalIntExp (Var var) state = lookfor var state
evalIntExp (UMinus int) state = evalUniOp negate $
    evalIntExp int state
evalIntExp (Plus int1 int2) state = evalBinOp (+)
    (evalIntExp int1 state) (evalIntExp int2 state)
evalIntExp (Minus int1 int2) state = evalBinOp (-)
    (evalIntExp int1 state) (evalIntExp int2 state)
evalIntExp (Times int1 int2) state = evalBinOp (*)
    (evalIntExp int1 state) (evalIntExp int2 state)
evalIntExp (Div int1 int2) state = evalDiv
    (evalIntExp int1 state) (evalIntExp int2 state)
evalIntExp (Tern bool int1 int2) state =
    case evalBoolExp bool state of
        Left err -> Left err
        Right True -> evalIntExp int1 state
        Right False -> evalIntExp int2 state

-- Evalua una expresion entera, sin efectos laterales
-- Completar definicion
evalBoolExp :: BoolExp -> State -> Either Error Bool
evalBoolExp BTrue _state = Right True
evalBoolExp BFalse _state = Right False
evalBoolExp (Eq int1 int2) state = evalBinOp (==)
    (evalIntExp int1 state) (evalIntExp int2 state)

```

```

evalBoolExp (Lt int1 int2) state      = evalBinOp (<)
    (evalIntExp int1 state) (evalIntExp int2 state)
evalBoolExp (Gt int1 int2) state      = evalBinOp (>)
    (evalIntExp int1 state) (evalIntExp int2 state)
evalBoolExp (And bool1 bool2) state = evalBinOp (&&)
    (evalBoolExp bool1 state) (evalBoolExp bool2 state)
evalBoolExp (Or bool1 bool2) state  = evalBinOp (||)
    (evalBoolExp bool1 state) (evalBoolExp bool2 state)
evalBoolExp (Not bool) state         = evalUniOp not
    (evalBoolExp bool state)

```

10. Ejercicio 10

10.1. Sintaxis Abstracta

```

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

```

10.2. Semántica Operacional de Comandos

$$\frac{\langle b, \sigma \rangle \Downarrow_{\text{boolexp}} \mathbf{true}}{\langle \mathbf{while} \ b \ \mathbf{do} \ c, \sigma \rangle \rightsquigarrow \langle c; \mathbf{while} \ b \ \mathbf{do} \ c, \sigma \rangle} \text{ WHILE}_1$$

$$\frac{\langle b, \sigma \rangle \Downarrow_{\text{boolexp}} \mathbf{false}}{\langle \mathbf{while} \ b \ \mathbf{do} \ c, \sigma \rangle \rightsquigarrow \langle \mathbf{skip}, \sigma \rangle} \text{ WHILE}_2$$