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

Lic. en Cs. de la Computación

Análisis de Lenguajes de Programación

Trabajo Práctico 1

Tomás Fernández De Luco F-3443/6
Ignacio Sebastián Moline M-6466/1

28 de septiembre de 2018

Ejercicio 1

El operador '?:' evalúa a la primera expresión entera si la condición booleana es verdadera y a la segunda en caso contrario. Por lo tanto, para representarlo en la sintaxis se necesitará una boolexp seguida de dos intexp. La sintaxis abstracta modificada para intexp será:

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

Y la sintaxis concreta correspondiente será:

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

Ejercicio 2

Se debe agregar un constructor más al tipo de IntExp con los tipos de datos mencionados previamente.

```haskell
-- Expresiones Aritmeticas
data IntExp = Const Integer
            | Var Variable
            | UMinus IntExp
            | Plus IntExp IntExp
            | Minus IntExp IntExp
            | Times IntExp IntExp
            | Div IntExp IntExp
            | TerCond BoolExp IntExp IntExp
 deriving Show
```

Ejercicio 3

El código del archivo Parser.hs es el siguiente:

```
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","skip","if",
                                                     "then","else","end",
                                                     "while","do", "repeat"]})

-----------------------------------
--- Parser de expressiones enteras
-----------------------------------
intexp :: Parser IntExp
intexp  = chainl1 term addopp

term = chainl1 factor multopp

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

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

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

-----------------------------------
--- Parser de expressiones booleanas
-----------------------------------
boolexp :: Parser BoolExp
boolexp  = chainl1 boolexp2 (try (do reservedOp lis "|"
                                     return Or))
```

```
boolexp2 = chainl1 boolexp3 (try (do reservedOp lis "&"
                                     return And))

boolexp3 = try (parens lis boolexp)
           <|> try (do reservedOp lis "~"
                       b <- boolexp3
                       return (Not b))
           <|> intcomp
           <|> boolvalue

intcomp = try (do i <- intexp
                  c <- compopp
                  j <- intexp
                  return (c i j))

compopp = try (do reservedOp lis "="
                  return Eq)
          <|> try (do reservedOp lis "<"
                      return Lt)
          <|> try (do reservedOp lis ">"
                      return Gt)

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

-----------------------------------
--- Parser de comandos
-----------------------------------
comm :: Parser Comm
comm = chainl1 comm2 (try (do reservedOp lis ";"
                              return Seq))

comm2 = try (do reserved lis "skip"
                return Skip)
        <|> try (do reserved lis "if"
                    cond <- boolexp
                    reserved lis "then"
                    case1 <- comm
                    reserved lis "else"
                    case2 <- comm
                    reserved lis "end"
                    return (Cond cond case1 case2))
        <|> try (do reserved lis "repeat"
                    c <- comm
                    reserved lis "until"
                    cond <- boolexp
                    reserved lis "end"
                    return (Repeat c cond))
        <|> try (do str <- identifier lis
                    reservedOp lis ":="
                    e <- intexp
                    return (Let str e))


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

Ejercicio 4

Se necesitarán dos reglas para representar el comportamiento esperado por el operador ternario '?:':

$$\frac{\langle p, \sigma\rangle \Downarrow_{boolexp} \textbf{true} \qquad \langle e_1, \sigma\rangle \Downarrow_{intexp} n}{\langle p \ ? \ e_1 : e_2, \sigma\rangle \Downarrow_{intexp} n} \ \textsc{TernTrue}$$

$$\frac{\langle p, \sigma\rangle \Downarrow_{boolexp} \textbf{false} \qquad \langle e_2, \sigma\rangle \Downarrow_{intexp} n}{\langle p \ ? \ e_1 : e_2, \sigma\rangle \Downarrow_{intexp} n} \ \textsc{TernFalse}$$

Ejercicio 5

Para demostrar el determinismo de la relación, se probará que si $t \rightsquigarrow t'$ y $t \rightsquigarrow t''$, entonces $t' = t''$, mediante inducción en la derivación $t \rightsquigarrow t'$. Para ello, se supondrá que las relaciones de evaluación big-step para expresiones enteras y booleanas son deterministas.

Si la última regla utilizada es ASS, t = $\langle v := e, \sigma\rangle$ y t' = $\langle skip, [\sigma|v : n]\rangle$, sabiendo que $\langle e, \sigma\rangle \Downarrow_{intexp} n$. Luego, en $t \rightsquigarrow t''$ no se puede haber aplicado como última regla SEQ_1, SEQ_2, IF_1, IF_2 o REPEAT, pues ellas requieren que el comando sea de un tipo distinto a la asignación. Por lo tanto, la última regla aplicada en $t \rightsquigarrow t''$ debe ser ASS. Como la evaluación de expresiones enteras es determinista, tenemos que si $\langle e, \sigma\rangle \Downarrow_{intexp} n$ y $\langle e, \sigma\rangle \Downarrow_{intexp} n'$, luego $n = n'$. Por lo tanto, $t'' = \langle skip, [\sigma|v : n]\rangle = t'$.

Si la última regla fue SEQ_1, t = $\langle skip; c_1, \sigma\rangle$ y $t' = \langle c_1, \sigma\rangle$. Entonces, en $t \rightsquigarrow t''$ no se puede haber aplicado ASS, IF_1, IF_2 o REPEAT, pues ellas requieren que el comando sea distinto de skip. Tampoco puede haber sido SEQ_2 ya que requeriría que $\langle skip, \sigma\rangle \rightsquigarrow \langle c', \sigma'\rangle$, pero no existen reglas que permitan derivarlo, pues es un valor. Por lo tanto, la última regla aplicada en $t \rightsquigarrow t''$ debe ser SEQ_1. Entonces, $t'' = \langle c_1, \sigma\rangle = t'$.

Si la última regla fue SEQ_2, t = $\langle c_0; c_1, \sigma\rangle$ y $t' = \langle c_0'; c_1, \sigma'\rangle$, sabiendo que $\langle c_0, \sigma\rangle \rightsquigarrow \langle c_0', \sigma'\rangle$. Entonces, en $t \rightsquigarrow t''$ no se puede haber aplicado ASS, IF_1, IF_2 o REPEAT, pues ellas requieren que el comando no sea una sucesión de dos subcomandos. Tampoco puede haber sido SEQ_1 ya que $\langle c_0, \sigma\rangle$ evalúa en un paso a $\langle c_0', \sigma'\rangle$, por la premisa de SEQ_2 en $t \rightsquigarrow t'$. Entonces, c_1 no puede ser skip, porque en ese caso, t no evaluaría a nada. Por lo tanto, la última regla aplicada en $t \rightsquigarrow t''$ debe ser SEQ_2. Sabemos entonces que $t'' = \langle c_0''; c_1, \sigma''\rangle$, con $\langle c_0, \sigma\rangle \rightsquigarrow \langle c_0'', \sigma''\rangle$. Luego, por hipótesis inductiva, se tiene que la evaluación de $\langle c_0, \sigma\rangle$ es determinista, por lo que tenemos que $\langle c_0', \sigma'\rangle = \langle c_0'', \sigma''\rangle$. Entonces, $t' = \langle c_0'; c_1, \sigma'\rangle = \langle c_0''; c_1, \sigma''\rangle = t''$.

Si la última regla utilizada es IF_1, t = $\langle$if b then c_0 else $c_1, \sigma\rangle$ y $t' = \langle c_0, \sigma\rangle$, sabiendo que $\langle b, \sigma\rangle \Downarrow_{boolexp} true$. Luego, la última regla aplicada en $t \rightsquigarrow t''$ no puede haber sido ASS, SEQ_1, SEQ_2 o REPEAT, pues ellas requieren que el comando sea de un tipo distinto a un if. Además, sabiendo que la relación de evaluación de expresiones booleanas es determinista, $\langle b, \sigma\rangle$ evalúa sólo a true, por lo que tampoco podría haberse usado IF_2. Por lo tanto, la última regla aplicada en $t \rightsquigarrow t''$ debe ser IF_1. Entonces, tenemos que $t'' = \langle c_0, \sigma\rangle = t'$. La demostración es análoga para el caso de IF_2.

Si la última regla fue REPEAT, t = $\langle repeat$ c $until$ b, $\sigma\rangle$ y $t' = \langle c$; if b then $skip$ else $repeat$ c $until$ b, $\sigma\rangle$. Entonces, en $t \rightsquigarrow t''$ no se puede haber aplicado ninguna otra regla, pues necesitan comandos distintos a un repeat. Por lo tanto, la última regla aplicada en $t \rightsquigarrow t''$ debe ser REPEAT. Entonces, $t'' = \langle c$; if b then $skip$ else $repeat$ c $until$ b, $\sigma\rangle = t'$.

Ejercicio 6

Para la resolución del ejercicio, se optó por dividir el árbol de derivación, dado que es demasiado grande para mostrarlo completo, y se considera que su desarrollo y explicación serán más claros de esta manera. Se comenzará con desarrollar la siguiente expresión:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\langle x, [\sigma|x:0]\rangle \Downarrow_{intexp} 0}\ \text{VAR} \quad \dfrac{}{\langle 1, [\sigma|x:0]\rangle \Downarrow_{intexp} 1}\ \text{NVAL}}{\langle x+1, [\sigma|x:0]\rangle \Downarrow_{intexp} 0+1=1}\ \text{PLUS}}{\langle x := x+1, [\sigma|x:0]\rangle \rightsquigarrow \langle skip, [\sigma|x:1]\rangle}\ \text{ASS}}{\langle x := x+1;\ \text{if } x>0 \text{ then } skip \text{ else } x := x-1, [\sigma|x:0]\rangle \rightsquigarrow \langle skip;\ \text{if } x>0 \text{ then } skip \text{ else } x := x-1, [\sigma|x:1]\rangle}\ \text{SEQ}_2$$

Sean $t_1 = \langle x := x+1;\ \text{if } x>0 \text{ then } skip \text{ else } x := x-1, [\sigma|x:0]\rangle$ y $t_2 = \langle skip;\ \text{if } x>0 \text{ then } skip \text{ else } x := x-1, [\sigma|x:1]\rangle$. Puede verse por el árbol anterior que $t_1 \rightsquigarrow t_2$. Luego,

$$\dfrac{}{t_2 \rightsquigarrow \langle \text{if } x>0 \text{ then } skip \text{ else } x := x-1, [\sigma|x:1]\rangle}\ \text{SEQ}_1$$

Ahora, sea $t_3 = \langle \text{if } x>0 \text{ then } skip \text{ else } x := x-1, [\sigma|x:1]\rangle$. Entonces,

$$\dfrac{\dfrac{\dfrac{}{\langle x, [\sigma|x:1]\rangle \Downarrow_{intexp} 1}\ \text{VAR} \quad \dfrac{}{\langle 0, [\sigma|x:0]\rangle \Downarrow_{intexp} 0}\ \text{NVAL}}{\langle x>0, [\sigma|x:1]\rangle \Downarrow_{boolexp} 1>0=true}\ \text{GT}}{t_3 \rightsquigarrow \langle skip, [\sigma|x:1]\rangle}\ \text{IF}_1$$

Donde $t_4 = \langle skip, [\sigma|x:1]\rangle$.
Como $\rightsquigarrow *$ es una clausura de $\rightsquigarrow$, tenemos que

$$\dfrac{t_1 \rightsquigarrow t_2}{t_1 \rightsquigarrow^* t_2}$$

$$\dfrac{t_2 \rightsquigarrow t_3}{t_2 \rightsquigarrow^* t_3}$$

$$\dfrac{t_3 \rightsquigarrow t_4}{t_3 \rightsquigarrow^* t_4}$$

Por último, como la relación es transitiva, se tiene que:

$$\dfrac{\dfrac{t_1 \rightsquigarrow^* t_2 \quad t_2 \rightsquigarrow^* t_3}{t_1 \rightsquigarrow^* t_3} \quad t_3 \rightsquigarrow^* t_4}{t_1 \rightsquigarrow^* t_4}$$

Completando así la prueba.

Ejercicio 7

Para el caso del error de división por cero, se deja que Haskell maneje el error y aborte la ejecución. Para las variables no definidas, se hizo un pattern matching no exhaustivo en la función *lookfor* así no habrá un comportamiento especificado en caso de no encontrar la variable en la lista de estados. El código del archivo Eval1.hs es el siguiente:

```
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
lookfor :: Variable -> State -> Integer
lookfor var ((x, value):xs) = if var == x
                              then value
                              else lookfor var xs

-- Cambia el valor de una variable en un estado
update :: Variable -> Integer -> State -> State
update var updateVal [] = [(var, updateVal)]
update var updateVal ((x, value):xs) =
    if var == x
    then (x, updateVal):xs
    else (x,value) : (update var updateVal 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 intE) state = update var (evalIntExp intE state) state
evalComm (Seq comm1 comm2) state = let state2 = (evalComm comm1 state)
                                   in evalComm comm2 state2
evalComm (Cond boolE commT commF) state = if (evalBoolExp boolE state)
                                          then evalComm commT state
                                          else evalComm commF state
evalComm (Repeat comm cond) state =
    evalComm (Seq comm (Cond cond Skip (Repeat comm cond))) state

-- Evalua una expresion entera, sin efectos laterales
evalIntExp :: IntExp -> State -> Integer
evalIntExp (Const int) state = int
evalIntExp (Var variable) state = lookfor variable state
evalIntExp (UMinus intE) state = -(evalIntExp intE state)
evalIntExp (Plus intE1 intE2) state =
    (evalIntExp intE1 state) + (evalIntExp intE2 state)
evalIntExp (Minus intE1 intE2) state =
    (evalIntExp intE1 state) - (evalIntExp intE2 state)
evalIntExp (Times intE1 intE2) state =
    (evalIntExp intE1 state) * (evalIntExp intE2 state)
evalIntExp (Div intE1 intE2) state =
    div (evalIntExp intE1 state) (evalIntExp intE2 state)
```

```
-- Evalua una expresion entera, sin efectos laterales
evalBoolExp :: BoolExp -> State -> Bool
evalBoolExp BTrue state = True
evalBoolExp BFalse state = False
evalBoolExp (Eq intE1 intE2) state =
    (evalIntExp intE1 state) == (evalIntExp intE2 state)
evalBoolExp (Lt intE1 intE2) state =
    (evalIntExp intE1 state) < (evalIntExp intE2 state)
evalBoolExp (Gt intE1 intE2) state =
    (evalIntExp intE1 state) > (evalIntExp intE2 state)
evalBoolExp (And boolE1 boolE2) state =
    (evalBoolExp boolE1 state) && (evalBoolExp boolE2 state)
evalBoolExp (Or boolE1 boolE2) state =
    (evalBoolExp boolE1 state) || (evalBoolExp boolE2 state)
evalBoolExp (Not boolE) state = not (evalBoolExp boolE state)
```

Ejercicio 8

El código del archivo Eval2.hs es el siguiente:

```
module Eval2 (eval) where

import AST

-- Estados
type State = [(Variable,Integer)]
data Error = DivByZero | UndefVar deriving Show

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

-- Busca el valor de una variable en un estado
lookfor :: Variable -> State -> Either Error Integer
lookfor var [] = Left UndefVar
lookfor var ((x, value):xs) = if var == x
                              then Right value
                              else lookfor var xs

-- Cambia el valor de una variable en un estado
update :: Variable -> Integer -> State -> State
update var updateVal [] = [(var, updateVal)]
update var updateVal ((x, value):xs) =
    if var == x
    then (x, updateVal):xs
    else (x,value) : (update var updateVal xs)

handleUnExpr :: Either Error a -> (a -> a) -> Either Error a
handleUnExpr (Left error) f = Left error
handleUnExpr (Right val) f = Right (f val)

handleBinExpr :: Either Error a -> Either Error a -> (a -> a -> b) -> Either Error b
handleBinExpr (Left error1) _ _ = Left error1
handleBinExpr _ (Left error2) _ = Left error2
handleBinExpr (Right val1) (Right val2) f = Right (f val1 val2)

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

-- Evalua un comando en un estado dado
evalComm :: Comm -> State -> Either Error State
evalComm Skip state = Right state
evalComm (Let var intE) state = case (evalIntExp intE state) of
                                    Left error -> Left error
                                    Right value -> Right (update var value state)
evalComm (Seq comm1 comm2) state = case (evalComm comm1 state) of
                                       Left error -> Left error
                                       Right state2 -> evalComm comm2 state2
evalComm (Cond boolE commT commF) state = case (evalBoolExp boolE state) of
                                              Left error -> Left error
                                              Right True -> evalComm commT state
                                              Right False -> evalComm commF state
evalComm (Repeat comm cond) state =
    evalComm (Seq comm (Cond cond Skip (Repeat comm cond))) state


-- Evalua una expresion entera, sin efectos laterales
```

```
evalIntExp :: IntExp -> State -> Either Error Integer
evalIntExp (Const int) state = Right int
evalIntExp (Var variable) state = lookfor variable state
evalIntExp (UMinus intE) state = handleUnExpr (evalIntExp intE state) (\x -> -x)
evalIntExp (Plus intE1 intE2) state =
    handleBinExpr (evalIntExp intE1 state) (evalIntExp intE2 state) (\x y -> x + y)
evalIntExp (Minus intE1 intE2) state =
    handleBinExpr (evalIntExp intE1 state) (evalIntExp intE2 state) (\x y -> x - y)
evalIntExp (Times intE1 intE2) state =
    handleBinExpr (evalIntExp intE1 state) (evalIntExp intE2 state) (\x y -> x * y)
evalIntExp (Div intE1 intE2) state =
     case (evalIntExp intE2 state) of
          Right 0 -> Left DivByZero
          x -> handleBinExpr (evalIntExp intE1 state) x (\x y -> div x y)

-- Evalua una expresion entera, sin efectos laterales
evalBoolExp :: BoolExp -> State -> Either Error Bool
evalBoolExp BTrue state = Right True
evalBoolExp BFalse state = Right False
evalBoolExp (Eq intE1 intE2) state =
    handleBinExpr (evalIntExp intE1 state) (evalIntExp intE2 state) (\x y -> x == y)
evalBoolExp (Lt intE1 intE2) state =
    handleBinExpr (evalIntExp intE1 state) (evalIntExp intE2 state) (\x y -> x < y)
evalBoolExp (Gt intE1 intE2) state =
    handleBinExpr (evalIntExp intE1 state) (evalIntExp intE2 state) (\x y -> x > y)
evalBoolExp (And boolE1 boolE2) state =
    handleBinExpr (evalBoolExp boolE1 state) (evalBoolExp boolE2 state) (\x y -> x && y)
evalBoolExp (Or boolE1 boolE2) state =
    handleBinExpr (evalBoolExp boolE1 state) (evalBoolExp boolE2 state) (\x y -> x || y)
evalBoolExp (Not boolE) state = handleUnExpr (evalBoolExp boolE state) (\x -> not x)
```

Ejercicio 9

Ahora *evalComm* devuelve un par cuyo primer elemento es un estado o error, y el segundo es la traza de asignaciones de variables hasta el final normal de la ejecución, o el error que la interrumpa. El código del archivo Eval3.hs es el siguiente:

```
module Eval3 (eval) where

import AST

-- Estados
type State = [(Variable,Integer)]
data Error = DivByZero | UndefVar deriving Show
type Trace = [String]

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

initTrace :: Trace
initTrace = []

-- Busca el valor de una variable en un estado
lookfor :: Variable -> State -> Either Error Integer
lookfor var [] = Left UndefVar
lookfor var ((x, value):xs) = if var == x
                              then Right value
                              else lookfor var xs

-- Cambia el valor de una variable en un estado
update :: Variable -> Integer -> State -> State
update var updateVal [] = [(var, updateVal)]
update var updateVal ((x, value):xs) =
    if var == x
    then (x, updateVal):xs
    else (x,value) : (update var updateVal xs)

handleUnExpr :: Either Error a -> (a -> a) -> Either Error a
handleUnExpr (Left error) f = Left error
handleUnExpr (Right val) f = Right (f val)

handleBinExpr :: Either Error a -> Either Error a -> (a -> a -> b) -> Either Error b
handleBinExpr (Left error1) _ _ = Left error1
handleBinExpr _ (Left error2) _ = Left error2
handleBinExpr (Right val1) (Right val2) f = Right (f val1 val2)

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

-- Evalua un comando en un estado dado
evalComm :: Comm -> State -> Trace -> (Either Error State, Trace)
evalComm Skip state trace = (Right state, trace)
evalComm (Let var intE) state trace =
    case (evalIntExp intE state) of
    Left error -> (Left error, trace)
    Right value -> (Right (update var value state),
                    trace ++ ["Let " ++ var ++ " " ++ show(value)])
evalComm (Seq comm1 comm2) state trace =
    case (evalComm comm1 state trace) of
         (Left error, trace2) -> (Left error, trace2)
         (Right state2, trace2) -> evalComm comm2 state2 trace2
```

```
evalComm (Cond boolE commT commF) state trace =
    case (evalBoolExp boolE state) of
         Left error -> (Left error, trace)
         Right True -> evalComm commT state trace
         Right False -> evalComm commF state trace
evalComm (Repeat comm cond) state trace =
    evalComm (Seq comm (Cond cond Skip (Repeat comm cond))) state trace

-- Evalua una expresion entera, sin efectos laterales
evalIntExp :: IntExp -> State -> Either Error Integer
evalIntExp (Const int) state = Right int
evalIntExp (Var variable) state = lookfor variable state
evalIntExp (UMinus intE) state = handleUnExpr (evalIntExp intE state) (\x -> -x)
evalIntExp (Plus intE1 intE2) state =
    handleBinExpr (evalIntExp intE1 state) (evalIntExp intE2 state) (\x y -> x + y)
evalIntExp (Minus intE1 intE2) state =
    handleBinExpr (evalIntExp intE1 state) (evalIntExp intE2 state) (\x y -> x - y)
evalIntExp (Times intE1 intE2) state =
    handleBinExpr (evalIntExp intE1 state) (evalIntExp intE2 state) (\x y -> x * y)
evalIntExp (Div intE1 intE2) state =
    case (evalIntExp intE2 state) of
         Right 0 -> Left DivByZero
         x -> handleBinExpr (evalIntExp intE1 state) x (\x y -> div x y)

-- Evalua una expresion entera, sin efectos laterales
evalBoolExp :: BoolExp -> State -> Either Error Bool
evalBoolExp BTrue state = Right True
evalBoolExp BFalse state = Right False
evalBoolExp (Eq intE1 intE2) state =
    handleBinExpr (evalIntExp intE1 state) (evalIntExp intE2 state) (\x y -> x == y)
evalBoolExp (Lt intE1 intE2) state =
    handleBinExpr (evalIntExp intE1 state) (evalIntExp intE2 state) (\x y -> x < y)
evalBoolExp (Gt intE1 intE2) state =
    handleBinExpr (evalIntExp intE1 state) (evalIntExp intE2 state) (\x y -> x > y)
evalBoolExp (And boolE1 boolE2) state =
    handleBinExpr (evalBoolExp boolE1 state) (evalBoolExp boolE2 state) (\x y -> x && y)
evalBoolExp (Or boolE1 boolE2) state =
    handleBinExpr (evalBoolExp boolE1 state) (evalBoolExp boolE2 state) (\x y -> x || y)
evalBoolExp (Not boolE) state = handleUnExpr (evalBoolExp boolE state) (\x -> not x)
```

Ejercicio 10

De manera similar al ejercicio 1, se extiende la sintaxis abstracta del tipo *comm* para agregar un comando adicional. El resultado final será:

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

El esquema de reglas para la sintaxis operacional de *while* será muy similar a la del *repeat*. En caso de que la condición sea cierta, se realizará el comando seguido de volver a ejecutar el ciclo; si no, se terminará el comando con un *skip*.

$$\frac{}{\langle \text{while } b \text{ do } c, \sigma\rangle \rightsquigarrow \langle \text{if } b \text{ then } c; \text{ while } b \text{ do } c \text{ else } skip, \sigma\rangle}\ \text{WHILE}$$