Trabajo Practico 1

Facundo Emmanuel Messulam
Franco Ignacio Vallejos Vigier

22 de septiembre de 2021

1. Ejercicio 1

1.1. SINTAXIS ABSTRACTA

```
intexp ::= nat | var | u intexp
        | intexp + intexp
        | intexp b intexp
        | intexp × intexp
        | intexp ÷ intexp
boolexp ::= true | false
        | intexp == intexp
        | intexp 6= 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

```
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 ')'
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

```
EAssgn ::Variable -> Exp Int -> Exp Int
ESeq ::Exp Int -> Exp Int -> Exp Int
```

2.1. SINTAXIS ABSTRACTA

```
intexp ::= nat | var | u intexp
    | intexp + intexp
    | intexp b intexp
    | intexp × intexp
    | intexp ÷ intexp
    | var '=' intexp
    | intexp ',' intexp
```

2.2. SINTAXIS CONCRETA

```
intexp ::= nat
    | var
    | '-' intexp
    | intexp '+' intexp
    | intexp '-' intexp
    | intexp '*' intexp
    | intexp '/' intexp
    | '(' intexp ')'
    | var '=' intexp
    | intexp ',' intexp
```

3. Ejercicio 3

```
----------------------------------
--- Parser de expressiones enteras
----------------------------------
intexp :: Parser (Exp Int)
intexp = (ignoreparensint assignintexp) `chainl1` commaintexp
```

```
commaintexp :: Parser (Exp Int -> Exp Int -> Exp Int)
commaintexp =
    do
      reservedOp lis ","
      return (ESeq)
    <|> do
      sumintexp

assignintexp :: Parser (Exp Int)
assignintexp =
    try (do {
      valor <- identifier lis;
      reservedOp lis "=";
      y <- (ignoreparensint assignintexp) `chainl1` sumintexp;
      return (EAssgn valor y);
    })
    <|> do
      simpleintexp

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

simpleintexp :: Parser (Exp Int)
simpleintexp =
    do
      valor <- identifier lis
      return (Var valor)
    <|> do
      valor <- natural lis
      return (Const (fromIntegral valor))
    <|> try (do {
              reservedOp lis "-";
```

```
              ignoreparensint (do {
                                 valor <- natural lis;
                                  return (UMinus (Const (fromInteger valor)))
                               });
      })


ignoreparensint :: Parser (Exp Int) -> Parser (Exp Int)
ignoreparensint parser =
    do
      parens lis intexp
    <|> do
      parser
-----------------------------------
--- Parser de expressiones booleanas
-----------------------------------

boolexp :: Parser (Exp Bool)
boolexp = (ignoreParensBool notboolexp) `chainl1` orboolexp

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

notboolexp :: Parser (Exp Bool)
notboolexp =
    do
      reservedOp lis "!"
      x <- ignoreParensBool mediumboolexp
      return (Not x)
    <|> do
      x <- ignoreParensBool mediumboolexp
      return (x)

mediumboolexp :: Parser (Exp Bool)
mediumboolexp =
    do
      comparisonsboolexp
    <|> do
      simpleboolexp
```

```
comparisonsboolexp :: Parser (Exp Bool)
comparisonsboolexp =
    (do
      operando1 <- intexp
      (do
         reservedOp lis ">"
         operando2 <- intexp
         return (Gt operando1 operando2)
       <|> do
         reservedOp lis "<"
         operando2 <- intexp
         return (Lt operando1 operando2)
       <|> do
         reservedOp lis "!="
         operando2 <- intexp
         return (NEq operando1 operando2)
       <|> do
         (reservedOp lis "==")
         operando2 <- intexp
         return (Eq operando1 operando2)))


simpleboolexp :: Parser (Exp Bool)
simpleboolexp =
    do
      reservedOp lis "true"
      return (BTrue)
    <|> do
      reservedOp lis "false"
      return (BFalse)

ignoreParensBool :: Parser (Exp Bool) -> Parser (Exp Bool)
ignoreParensBool parser =
    do
      parens lis boolexp
    <|> do
      parser

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

comm :: Parser Comm
```

```
comm = (commaux) `chainl1` (do { reservedOp lis ";" ; return (Seq) })

commaux :: Parser Comm
commaux =
    do
      reservedOp lis "skip"
      return Skip
    <|> do
      reservedOp lis "if"
      operando1 <- boolexp
      operando2 <- braces lis comm
      (do
         reservedOp lis "else"
         operando3 <- braces lis comm
         return (IfThenElse operando1 operando2 operando3)
       <|> do
         return (IfThen operando1 operando2))
    <|> do
      reservedOp lis "repeat"
      operando1 <- braces lis comm
      reservedOp lis "until"
      operando2 <- boolexp
      return (Repeat operando1 operando2)
    <|> do
      nombre <- identifier lis
      reservedOp lis "="
      valor <- intexp
      return (Let nombre valor)




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


-------------------------------------
-- Función de testeo
-------------------------------------

test1 = parseComm "error" "skip;skip;skip"
test2 = parseComm "error" "if true { skip } else { skip; repeat { skip } until true}"
```

```
test3 = parseComm "error" "x = y = 3, 90 - 5 * 90, -3"
test4 = parseComm "error"
"if true && false || false {skip} else {skip; if 3 < 5 {skip} else {skip}}"
test5 = parseComm "error" "if false && true {skip}"
test6 = parseComm "error"
"if (false || (true || false)) {skip};if (((false || true) || false)) {skip}"
test7 = parseComm "error" "x = (y = 3), (90 - 5) * 90, -(3), (3 - 2) - 1, 3 - (2 - 1)"
test70 = parseComm "error" "x = (90 - 5) * 90"
test8 = parseComm "error" "x = 3 + (y = z = q = 4) "
test80 = parseComm "error" "x = y = z = 0"
test9 = parseComm "error" "x = x + y + 5 * (z = 4 * 12 + 1) / abcdefghi"
test90 = parseComm "error" "x = x + y + 5 * z = 4 * 12 + 1 / abcdefghi"
test901 = parseComm "error" "m = 5 * z = 4 * 12 + 1"
test91 = parseComm "error" "m = (x = 3) + (y = z = q = 4)"
test10 = parseComm "error" "x = x + y +zrfsfs"
test11 = parseComm "error" "x = x + y +23"
test12 = parseComm "error" "if b > a {c = 1} else{c = 0} ; a = a / 0"
testN = parseTest (comm >> Text.Parsec.Combinator.parserTrace "label")

right x = case x of
          Right y -> y
          _ -> undefined

superTest = do { test1; test2; test3; test4; test5 ; test6 ; test7 ; test70
; test8; test80 ;test9;test90;test901;test91; test10 ; test11; test12 }
```

4. Ejercicio 4

4.1. Asignacion

$$\frac{\langle x, \sigma\rangle \Downarrow_{exp} \langle \sigma x, \sigma\rangle \quad \langle e_0, \sigma\rangle \Downarrow_{exp} \langle n_0, \sigma'\rangle}{\langle x = e_0; \sigma\rangle \Downarrow_{exp} \langle n_0; [\sigma' | x : n_0]\rangle}\ \text{EASSING}$$

4.2. Coma

$$\frac{\langle e_0, \sigma\rangle \Downarrow_{exp} \langle n_0, \sigma'\rangle \quad \langle e_1, \sigma\rangle \Downarrow_{exp} \langle n_1, \sigma''\rangle}{\langle e_0, e_1; \sigma\rangle \Downarrow_{exp} \langle n_1; \sigma''\rangle}\ \text{COMMA}$$

5. Ejercicio 5

Como objetivo tenemos que ver si:
$t \rightsquigarrow t' y t \rightsquigarrow t''$ entonces $t' = t''$

Para demostrar lo propuesto, haremos inducción sobre la derivación $t \rightsquigarrow t'$ Planteamos como HI, que todas las subderivaciones de t son determinístas. Por lo tanto al verificar la propiedad para cada nueva regla de derivación, se cumpliría la propiedad.

Si la última regla de $t \rightsquigarrow t'$ es ASS:

Sabemos que t es de la forma $\langle v = e, \sigma\rangle$:

1) Como las demás reglas de derivación viendo su forma no son aplicables, sabemos que t deriva a skip por lo tanto la ejecución se detendrá, no pudiendo seguir derivando.

2) Como $\Downarrow_{exp}$ es determinista NO podemos tener que $\langle e, \sigma\rangle \Downarrow_{exp} \langle n, \sigma'\rangle$ y $\langle e, \sigma\rangle \Downarrow_{exp} \langle n', \sigma'\rangle$ con n distinto n'.

De (1) y (2) podemos decir que dada una segunda derivación $t \rightsquigarrow t''$, su última regla es ASS y llegando a un mismo conjunto de estados σ' que puede asignar solamente el valor n. Por lo tanto $t' = t''$

Si la última regla de $t \rightsquigarrow t'$ es SEQ_1: Sabemos que t es de la forma $\langle skip, t_1, \sigma\rangle$ Entonces, la premisa para aplicar SEQ_2 no puede ser posible, ya que toda ejecución que termina lo hace en skip, para algún estado σ. Por lo tanto, la última regla aplicada en la segunda derivación no puede ser SEQ_2, ya que la derivación $\langle skip, \sigma\rangle \rightsquigarrow \langle c_0', \sigma'\rangle$ no es posible. Además, por la forma de t, ninguna de las otras reglas presentes puede ser utilizada. Por ende t $t' = t''$

Si la última regla de $t \rightsquigarrow t'$ es SEQ_2: Sabemos que t es de la forma $\langle t_1, t_2, \sigma\rangle$ donde $\langle t_1, \sigma\rangle \rightsquigarrow \langle t_1', \sigma'\rangle$ (1). Debido a esto, la última regla aplicada en la segunda derivación no puede ser SEQ_1, porque tenemos como premisa (1) y si $t_1 = skip$ es ABS ya que t_1 tiene derivación en un paso por (1).Por HI, en la segunda derivación tendremos también que $\langle t_1, \sigma\rangle \rightsquigarrow \langle t_1', \sigma'\rangle$. Donde los comandos y los estados derivados coinciden con los de la primera derivación, por ser subderivaciones de t. Además, por la forma de t, ninguna de las otras reglas presentes puede ser utilizada. Por ende $t' = t''$

Si la última regla de $t \rightsquigarrow t'$ es IF_1: Entonces sabemos que t es de la forma $\langle if\ t_1\ then\ t_2\ else\ t_3, \sigma\rangle$ donde $\langle t_1, \sigma\rangle \Downarrow_{exp} \langle true, \sigma'\rangle$. Como $\Downarrow_{exp}$ es determinista entonces no puede pasar que

la última regla usada en $t \rightsquigarrow t''$ sea IF_2. Y dado a la forma de t, ninguna de las otras reglas presentes puede ser utilizada. Entonces la última regla en la segunda derivación es IF_1 por lo tanto $t' = t''$

Si la última regla de $t \rightsquigarrow t'$ es IF_2: Igual a IF_1

Si la última regla de $t \rightsquigarrow t'$ es $REPEAT$: Entonces sabemos que t es de la forma $\langle repeat\ c\ until\ b, \sigma \rangle$. Sabiendo esto podemos descartar el uso de la regla SEQ_1 dado que si la usamos estaríamos forzando que $\langle skip, \sigma \rangle \rightsquigarrow \langle c_0', \sigma' \rangle$ lo cual es ABS. Por lo cual la única regla capaz de finalizar la segunda derivación es REPEAT. Entonces $t_1 = t_2$

6. Ejercicio 6

$\langle y = 1, [[\sigma|x : 2]|y : 2]\rangle \Downarrow_{exp} \langle 1, [[\sigma|x : 1]|y : 2]\rangle$

$\langle y = 1, [[\sigma|x : 2]|y : 2]\rangle \rightsquigarrow \langle 1, [[\sigma|x : 1]|y : 2]\rangle$ A (EASSING,$\Downarrow_{exp}$ es determinista)

A, $\langle x = n, [[\sigma|x : 2]|y : 2]\rangle \rightsquigarrow \langle \mathbf{skip}, [[\sigma|x : n]|y : 1]\rangle$

$\langle x = y = 1, [[\sigma|x : 2]|y : 2]\rangle \rightsquigarrow^* \langle \mathbf{skip}, [[\sigma|x : 1]|y : 1]\rangle$ B(ASS y def $\rightsquigarrow *$)

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

$\langle x = y = 1; \mathbf{repeat}\, x = x - y \,\mathbf{until}\, x == 0, [[\sigma|x : 2]|y : 2]\rangle \rightsquigarrow^* \langle x = y = 1; x = x - y; \mathbf{if}\, x == 0 \,\mathbf{then\ skip\ else\ repeat}\, x = x - y \,\mathbf{until}\, x == 0, [[\sigma|x : 1]|y : 1]\rangle$ C(SEQ$_2$, $def \rightsquigarrow^*$, REPEAT)

B, $\langle x = x - y, [[\sigma|x : 1]|y : 1]\rangle \rightsquigarrow \langle \mathbf{skip}, [[\sigma|x : 0]|y : 1]\rangle$

$\langle x = y = 1; x = x - y, [[\sigma|x : 2]|y : 2]\rangle \rightsquigarrow^* \langle \mathbf{skip}, [[\sigma|x : 0]|y : 1]\rangle$ D(SEQ$_2$ y def $\rightsquigarrow *$)

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

$\langle x = y = 1; \mathbf{repeat}\, x = x - y \,\mathbf{until}\, x == 0, [[\sigma|x : 2]|y : 2]\rangle \rightsquigarrow^* \langle \mathbf{if}\, x == 0 \,\mathbf{then\ skip\ else\ repeat}\, x = x - y \,\mathbf{until}\, x == 0, [[\sigma|x : 0]|y : 1]\rangle$ E(def $\rightsquigarrow *$)

$\langle x == 0, [[\sigma|x : 0]|y : 1]\rangle \Downarrow_{exp} \langle \mathbf{true}, [[\sigma|x : 0]|y : 1]\rangle$

$\langle x == 0, [[\sigma|x : 0]|y : 1]\rangle \rightsquigarrow \langle \mathbf{true}, [[\sigma|x : 0]|y : 1]\rangle$ F($\Downarrow_{exp}$ es determinista)

E, F, $\langle \mathbf{if}\, x == 0 \,\mathbf{then\ skip\ else\ repeat}\, x = x - y \,\mathbf{until}\, x == 0, [[\sigma|x : 0]|y : 1]\rangle \rightsquigarrow \langle \mathbf{if\ true\ then\ skip\ else\ repeat}\, x = x - y \,\mathbf{until}\, x == 0, [[\sigma|x : 0]|y : 1]\rangle, IF_1$

$\langle x = y = 1; \mathbf{repeat}\, x = x - y \,\mathbf{until}\, x == 0, [[\sigma|x : 2]|y : 2]\rangle \rightsquigarrow^* \langle \mathbf{skip}, [[\sigma|x : 0]|y : 1]\rangle$ H (IF$_1$, SEQ_2)

7. Ejercicio 7

```
-- Estados
type State = M.Map Variable Int

-- Estado nulo
-- Completar la definición
initState :: State
initState = M.empty

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

-- Cambia el valor de una variable en un estado
-- Completar la definición
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 definición
stepComm :: Comm -> State -> Pair Comm State
stepComm (Skip)             e = skip e
stepComm (IfThenElse b x y) e = let (t :!: e') = evalExp b e
                                in if t then stepComm x e' else stepComm y e'
stepComm (IfThen b x)       e = let (t :!: e') = evalExp b e
                                in if t then stepComm x e' else skip e'
stepComm (Repeat x b)       e =
stepComm (Seq (x) (IfThenElse b (Skip) (Repeat x b))) e
stepComm (Let n v)          e = let (i :!: e') = evalExp v e
                                in (Skip :!: update n i e')
stepComm (Seq Skip y)       e = stepComm y e
stepComm (Seq x y)          e = let (v :!: e') = (stepComm x e)
```

```
                                in stepComm y e'
skip e = (Skip :!: e)

-- Evalua una expresion
-- Completar la definición
evalExp :: Exp a -> State -> Pair a State

evalExp (BTrue)    e = (True :!: e)
evalExp (BFalse)   e = (False :!: e)

evalExp (Lt x y) e = evalBinExp (<) x y e
evalExp (Gt x y) e = evalBinExp (>) x y e
evalExp (Eq x y) e = evalBinExp (==) x y e
evalExp (NEq x y) e = evalBinExp (/=) x y e

--TODO cortocircuito?
evalExp (And x y) e = evalBinExp (&&) x y e
evalExp (Or x y) e = evalBinExp (||) x y e


evalExp (Const x) e = (x :!: e)
evalExp (Var x) e = (lookfor x e :!: e)

evalExp (UMinus x) e = let (r :!: e') = (evalExp x e)
                       in (-r :!: e')

evalExp (Plus x y) e = evalBinExp (+) x y e
evalExp (Minus x y) e = evalBinExp (-) x y e
evalExp (Times x y) e = evalBinExp (*) x y e
evalExp (Div x y) e = evalBinExp (div) x y e

evalExp (EAssgn x y) e = let (r :!: re) = evalExp y e
                         in (r :!: update x r re)
evalExp (ESeq x y) e = evalBinExp (\l r -> r) x y e


evalBinExp f x y e = let (r :!: re) = evalExp x e
                         (l :!: le) = evalExp y re
                     in ((f r l) :!: le)
```

8. Ejercicio 8

```
    -- Estados
type State = M.Map Variable Int
```

```
-- Estado nulo
-- Completar la definición
initState :: State
initState = M.empty

-- Busca el valor de una variable en un estado
-- Completar la definición
lookfor :: Variable -> State -> Either Error Int
lookfor v s = lookfor' (s M.!? v)

lookfor' (Just x)  = Right x
lookfor' (Nothing) = Left UndefVar

-- Cambia el valor de una variable en un estado
-- Completar la definición
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 definición
stepComm :: Comm -> State -> Either Error (Pair Comm State)
stepComm (Skip)             e = skip e
stepComm (IfThenElse b x y) e = evalFailingExp (evalExp b e)
(\(t :!: e') -> if t then stepComm x e' else stepComm y e')
stepComm (IfThen b x)       e = evalFailingExp (evalExp b e)
(\(t :!: e') -> if t then stepComm x e' else skip e')
stepComm (Repeat x b)       e
= stepComm (Seq (x) (IfThenElse b (Skip) (Repeat x b))) e
stepComm (Let n v)          e = evalFailingExp (evalExp v e)
(\(i :!: e') -> Right  (Skip :!: update n i e'))
stepComm (Seq Skip y)       e = stepComm y e
stepComm (Seq x y)          e = evalFailingExp (stepComm x e)
```

```
(\(v :!: e') -> stepComm y e')
skip e = Right (Skip :!: e)

-- Evalua una expresion
-- Completar la definición
evalExp :: Exp a -> State -> Either Error (Pair a State)

evalExp (BTrue)    e = Right (True :!: e)
evalExp (BFalse)   e = Right (False :!: e)

evalExp (Lt x y) e = evalBinExp (<) x y e
evalExp (Gt x y) e = evalBinExp (>) x y e
evalExp (Eq x y) e = evalBinExp (==) x y e
evalExp (NEq x y) e = evalBinExp (/=) x y e

--TODO cortocircuito?
evalExp (And x y) e = evalBinExp (&&) x y e
evalExp (Or x y) e = evalBinExp (||) x y e


evalExp (Const x) e = Right (x :!: e)
evalExp (Var x) e =  evalFailingExp (lookfor x e) (\x -> Right (x :!: e))


evalExp (UMinus x) e = evalFailingExp (evalExp x e) (\(r :!: e') -> Right (-r :!: e))

evalExp (Plus x y) e = evalBinExp (+) x y e
evalExp (Minus x y) e = evalBinExp (-) x y e
evalExp (Times x y) e = evalBinExp (*) x y e
evalExp (Div x y) e = evalBinExp (div) x y e

evalExp (EAssgn x y) e = evalFailingExp (evalExp y e)
(\(r :!: re) -> Right (r :!: update x r re))
evalExp (ESeq x y) e = evalBinExp (\l r -> r) x y e


evalBinExp f x y e = let h r (l :!: le) = Right ((f r l) :!: le)
                         g (r :!: re) = evalFailingExp (evalExp y re) (h r)
                     in evalFailingExp (evalExp x e) g

evalFailingExp :: Either Error a -> (a -> Either Error b) -> Either Error b
evalFailingExp exp f = case exp of
                       Right x -> (f x)
                       Left y  -> Left y
```

9. Ejercicio 9

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

-- Estado nulo
-- Completar la definición
initState :: State
initState = (M.empty, "")

-- Busca el valor de una variable en un estado
-- Completar la definición
lookfor :: Variable -> State -> Either Error Int
lookfor v (m, s) = lookfor' (m M.!? v)

lookfor' (Just x)  = Right x
lookfor' (Nothing) = Left UndefVar

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

-- Agrega una traza dada al estado
-- Completar la definición
addTrace :: String -> State -> State
addTrace s (m, t) = (m, t ++ s)

addVarTrace :: Variable -> Int -> State -> State
addVarTrace v i e = addTrace ("Let " ++ v ++ " " ++ show i ++ "\n") e

tracingUpdate v i e = addVarTrace v i (update v i e)

-- 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 definición
stepComm :: Comm -> State -> Either Error (Pair Comm State)
stepComm (Skip)             e = skip e
stepComm (IfThenElse b x y) e = evalFailingExp (evalExp b e)
(\(t :!: e') -> if t then stepComm x e' else stepComm y e')
stepComm (IfThen b x)       e = evalFailingExp (evalExp b e)
(\(t :!: e') -> if t then stepComm x e' else skip e')
stepComm (Repeat x b)       e
= stepComm (Seq (x) (IfThenElse b (Skip) (Repeat x b))) e
stepComm (Let n v)          e = evalFailingExp (evalExp v e)
(\(i :!: e') -> Right  (Skip :!: tracingUpdate n i e'))
stepComm (Seq Skip y)       e = stepComm y e
stepComm (Seq x y)          e = evalFailingExp (stepComm x e)
(\(v :!: e') -> stepComm y e')
skip e = Right (Skip :!: e)

-- Evalua una expresion
-- Completar la definición
evalExp :: Exp a -> State -> Either Error (Pair a State)

evalExp (BTrue)    e = Right (True :!: e)
evalExp (BFalse)   e = Right (False :!: e)

evalExp (Lt x y) e = evalBinExp (<) x y e
evalExp (Gt x y) e = evalBinExp (>) x y e
evalExp (Eq x y) e = evalBinExp (==) x y e
evalExp (NEq x y) e = evalBinExp (/=) x y e

--TODO cortocircuito?
evalExp (And x y) e = evalBinExp (&&) x y e
evalExp (Or x y) e = evalBinExp (||) x y e


evalExp (Const x) e = Right (x :!: e)
evalExp (Var x) e =  evalFailingExp (lookfor x e) (\x -> Right (x :!: e))


evalExp (UMinus x) e = evalFailingExp (evalExp x e) (\(r :!: e') -> Right (-r :!: e))

evalExp (Plus x y) e = evalBinExp (+) x y e
evalExp (Minus x y) e = evalBinExp (-) x y e
evalExp (Times x y) e = evalBinExp (*) x y e
evalExp (Div x y) e = evalBinExp (div) x y e
```

```
evalExp (EAssgn x y) e = let value = (evalExp y e)
                             f (r :!: re) = Right (r :!: tracingUpdate x r re)
                         in evalFailingExp value f
evalExp (ESeq x y) e = evalBinExp (\l r -> r) x y e


evalBinExp f x y e = let h r (l :!: le) = Right ((f r l) :!: le)
                         g (r :!: re) = evalFailingExp (evalExp y re) (h r)
                     in evalFailingExp (evalExp x e) g

evalFailingExp :: Either Error a -> (a -> Either Error b) -> Either Error b
```

10. Ejercicio 10

10.1. Adicion a la gramatica abstracta

```
comm ::= skip
    | var '=' intexp
    | comm ';' comm
    | 'if' boolexp '{' comm '}'
    | 'if' boolexp '{' comm '}' 'else' '{' comm '}'
    | 'repeat' comm 'until' boolexp 'end'
    | 'for' (intexp;boolexp;intexp) comm
```

10.2. Adicion a la semantica operacional

A la semántica operacional de comandos se le agrega la siguiente regla:

$$\frac{\langle e1, \sigma\rangle \Downarrow_{exp} \langle n, \sigma'\rangle}{\langle for(e1; b; e2)c, \sigma\rangle \rightsquigarrow \langle if\ b\ \{repeat\ (c; e2)\ until\ b\}\ else\ \{skip\}, \sigma'\rangle}\ \text{FOR}$$