Analisis de Lenguajes de Programación
Trabajo Práctico 1

Carlini Milton Vitali Franco

15 de Septiembre de 2016

1. Ejercicio 2.2.1

Para poder incluir a la operación "? : " agregamos la siguiente línea a la sintaxis abstracta de LIS:

$<intexp>$::= | $<boolexp>$? $<intexp>$: $<intexp>$

Además añadimos la siguiente línea a la sintaxis concreta:

$<intexp>$::= | $<boolexp>$ '?' $< intexp >$ ':' $<intexp>$

2. Ejercicio 2.4.1

Agregamos la siguiente línea a la semántica denotacional de expresiones enteras para poder incluir el operador ternario:

$$[\![p \ ? \ e_1 : e_2]\!]_{intexp^\sigma} = \begin{cases} [\![e_1]\!]_{intexp^\sigma} & si \ [\![p]\!]_{boolexp^\sigma} = true \\ [\![e_2]\!]_{intexp^\sigma} & si \ [\![p]\!]_{boolexp^\sigma} = false \end{cases}$$

3. Ejercicio 2.5.1

Para probar que que la relación de evaluación $\rightsquigarrow$ en un paso es determinista, debemos probar que si $\gamma \rightsquigarrow \gamma'$ y $\gamma \rightsquigarrow \gamma''$ entonces $\gamma' = \gamma''$, donde cada γ es un estado (una configuración terminal) o un comando junto con un conjunto de estados (una configuración no terminal).

Dem/ Por inducción en la derivación de $\gamma \rightsquigarrow \gamma'$. En cada paso de la derivación asumimos que la regla vale para las subderivaciones y procedemos por el análisis de la última regla de evaluación usada en la derivación.

Si la última regla usada en la derivación de $\gamma \rightsquigarrow \gamma'$ es ASS entonces sabemos que cada configuración es de la forma:

- $\gamma =$ <v ::= e, σ>
- $\gamma' = [\sigma | v : [\![e]\!]_{intexp^\sigma}]$

Luego vemos que la última regla aplicada en la derivación de $\gamma \rightsquigarrow \gamma''$ no puede haber sido otra diferente de ASS por la forma de γ que solo admite aplicar esa única regla de inferencia. Entonces claramente $\gamma' = \gamma''$.

Si la última regla usada en la derivación de $\gamma \rightsquigarrow \gamma'$ es SKIP entonces sabemos que cada configuración es de la forma:

- $\gamma =$ <skip, σ>
- $\gamma' = \sigma$

Luego vemos que la última regla aplicada en la derivación de $\gamma \rightsquigarrow \gamma''$ no puede haber sido otra diferente de SKIP por la forma de γ que solo admite aplicar esa única regla de inferencia. Entonces claramente $\gamma' = \gamma''$.

Si la última regla usada en la derivación de $\gamma \rightsquigarrow \gamma'$ es SEQ_1 entonces sabemos que cada configuración es de la forma:

- $\gamma = <c_0; c_1, \sigma>$
- $\gamma' = <c_1, \sigma'>$
- $<c_0, \sigma> \rightsquigarrow \sigma'$

Luego vemos que la última regla aplicada en la derivación de $\gamma \rightsquigarrow \gamma''$ pudo haber sido SEQ_2 o SEQ_1 por la forma de γ. Suponemos que en la derivación de γ'' se obtuvo a partir de SEQ_2, luego

- $\gamma = <c_0; c_1, \sigma>$
- $\gamma'' = <c_0^{'}; c_1, \sigma'>$
- $<c_0, \sigma> \rightsquigarrow <c_0^{'}, \sigma'>$

Llegamos a una contradicción ya que por HI sabemos que para toda subderivacion vale la propiedad, pero $<c_0, \sigma> \rightsquigarrow <c_0^{'}, \sigma'>$ y $<c_0, \sigma> \rightsquigarrow \sigma'$ donde $\sigma' \neq <c_0^{'}, \sigma'>$. Esta contradicción provino de suponer que la última regla aplicada fue SEQ_2. Luego, γ'' fue obtenida de la regla de inferencia SEQ_1 y por lo tanto $\gamma' = \gamma''(porHI)$.

Si la última regla usada en la derivación de $\gamma \rightsquigarrow \gamma'$ es SEQ_2, la prueba es análoga a la de SEQ_1.

Si la última regla usada en la derivación de $\gamma \rightsquigarrow \gamma'$ es IF_1 entonces sabemos que cada configuración es de la forma:

- $\gamma = <$if b then c_0 else $c_1, \sigma>$
- $\gamma' = <c_0, \sigma>$
- $[\![b]\!]_{boolexp^\sigma} = \text{true}$

Luego, última regla aplicada en la derivación de $\gamma \rightsquigarrow \gamma''$ pudo haber sido IF_2 o IF_1, pero como $[\![b]\!]_{boolexp^\sigma} = \text{true}$, podemos descartar IF_2. Por lo tanto $\gamma' = \gamma''$.

Si la última regla usada en la derivación de $\gamma \rightsquigarrow \gamma'$ es IF_2, la prueba es análoga a la de IF_1.

Si la última regla usada en la derivación de $\gamma \rightsquigarrow \gamma'$ es $WHILE_1$ entonces sabemos que cada configuración es de la forma:

- $\gamma = <$while b do $c, \sigma>$
- $\gamma' = <c;$ while b do $c, \sigma>$
- $[\![b]\!]_{boolexp^\sigma} = \text{true}$

Luego, la última regla aplicada en la derivación de $\gamma \rightsquigarrow \gamma''$ pudo haber sido $WHILE_2$ o $WHILE_1$, pero como $[\![b]\!]_{boolexp^\sigma} = \text{true}$, podemos descartar $WHILE_2$. Por lo tanto $\gamma' = \gamma''$.

Si la última regla usada en la derivación de $\gamma \rightsquigarrow \gamma'$ es $WHILE_2$, la prueba es análoga a la de $WHILE_1$.

4. Ejercicio 2.5.2

Para construir el árbol de prueba, reemplazamos cada configuración por una letra para simplificar la prueba.

$O = <x ::= x + 1;$ if $x > 0$ then $skip$ else $x ::= x - 1, [\sigma|x : 0]>$
$F = [\sigma|x : 1]$
$A = <x ::= x + 1, [\sigma|x : 0]> \rightsquigarrow [\sigma|x : 1]$
$I = <$if $x > 0$ then $skip$ else $x ::= x - 1, [\sigma|x : 1]>$
$B = [\![x > 0]\!]_{boolexp^{[\sigma|x:1]}} =$ true
$S = <$skip,$[\sigma|x : 1]>$

$$\dfrac{\dfrac{\dfrac{\overline{A}\ ASS}{O \rightsquigarrow I}\ SEQ_1 \quad \dfrac{\dfrac{B}{I \rightsquigarrow S}\ IF_1 \quad \dfrac{\overline{S \rightsquigarrow F}\ SKIP}{S \rightsquigarrow^* F}\ TR_1}{I \rightsquigarrow^* F}\ TR_1}{O \rightsquigarrow^* I}\ TR_1}{O \rightsquigarrow^* F}\ TR_2$$

5. Ejercicio 2.5.6

Regla de la semántica operacional para repeat:

(repeat c until b, $\sigma) \rightsquigarrow (c; while\ \neg b\ do\ c,\ \sigma)$

Para extender la gramática abstracta de LIS agregamos la siguiente regla a la sintaxis abstracta:

$<comm>$::= | repeat $<intexp>$ until $<boolexp>$

Parser.hs

```
1  module Parser where
2
3  import Text.ParserCombinators.Parsec
4  import Text.Parsec.Token
5  import Text.Parsec.Language (emptyDef)
6  import Data.Char
7  import AST
8
9  -----------------------------------------------------
10 --- Funcion para facilitar el testing del parser ---
11 -----------------------------------------------------
12 totParser :: Parser a -> Parser a
13 totParser p = do
14                   whiteSpace lis
15                   t <- p
16                   eof
17                   return t
18
19 -- Analizador de Tokens
20 lis :: TokenParser u
21 lis = makeTokenParser (emptyDef   { commentStart    = "/*"
22                                    , commentEnd      = "*/"
23                                    , commentLine     = "//"
24                                    , opLetter        = char '='
25                                    , reservedOpNames = ["+","-","*","/","?",">",
       "<","&","|","?"]
26                                    , reservedNames   = ["true","false","skip","
       if","repeat","until",
27                                                         "then","else","end", "
       while","do"]
28                                    })
29
30 ---------------------------------------
31 --- Parser de expressiones enteras ---
32 ---------------------------------------
33 {-
34 IntExpr   -> IntExpr('+' IntTerm | '-' IntTerm) | IntTerm
35 IntTerm   -> IntTerm('*' IntFactor | '/' IntFactor) | IntFactor
36 IntFactor -> <nat> | <var> | '-' <nat> | '(' IntExpr ')'
37               | BoolExpr '?' IntExpr ':' IntExpr
38 -}
39 intExp :: Parser IntExp
40 intExp = chainl1 intTerm addop
41
42 intTerm :: Parser IntExp
43 intTerm = chainl1 intFactor mulop
44
45 intFactor :: Parser IntExp
46 intFactor = do {d<-natural lis; return (Const d)}
47             <|> do {reservedOp lis "-"; e<-intFactor; return (UMinus e)}
48             <|> do {reservedOp lis "("; e<-intExp; reservedOp lis ")"; return e
       }
49             <|> do {e<-identifier lis; return (Var e)}
50
51 addop :: Parser (IntExp -> IntExp -> IntExp)
52 addop = do {reservedOp lis "+"; return Plus}
53         <|> do {reservedOp lis "-"; return Minus}
54
55
```

```
56  mulop :: Parser (IntExp -> IntExp -> IntExp)
57  mulop = do {reservedOp lis "*"; return Times}
58          <|> do {reservedOp lis "/"; return Div}
59
60  ----------------------------------------
61  --- Parser de expresiones booleanas ---
62  ----------------------------------------
63  {-
64  BoolExpr   -> BoolTerm '|' BoolExpr | BoolTerm
65  BoolTerm   -> BoolFactor '&' BoolTerm | BoolFactor
66  BoolFactor -> IntExpr( '=' IntExpr | '<' IntExpr | '>' IntExpr)
67                | 'not' BoolExpr | '(' BoolExpr ')' | BTrue | BFalse
68  -}
69  boolExp :: Parser BoolExp
70  boolExp = chainl1 boolTerm orOp
71
72  boolTerm :: Parser BoolExp
73  boolTerm = chainl1 boolFactor andOp
74
75  boolFactor :: Parser BoolExp
76  boolFactor = do {reservedOp lis "("; p<-boolExp; reservedOp lis ")"; return p}
77               <|> do {reservedOp lis "~"; p<-boolFactor; return (Not p)}
78               <|> do {reserved lis "true"; return (BTrue)}
79               <|> do {reserved lis "false"; return (BFalse)}
80               <|> do p<-intExp
81                      (do {reservedOp lis ">"; q<-intExp; return (Gt p q)}
82                       <|> do {reservedOp lis "<"; q<-intExp; return (Lt p q)}
83                       <|> do {reservedOp lis "="; q<-intExp; return (Eq p q)})
84
85  orOp :: Parser (BoolExp -> BoolExp -> BoolExp)
86  orOp = do {reservedOp lis "|"; return Or}
87
88  andOp :: Parser (BoolExp -> BoolExp -> BoolExp)
89  andOp = do {reservedOp lis "&"; return And}
90  --------------------------
91  --- Parser de comandos ---
92  --------------------------
93  {-
94  commExp  -> commTerm ';' commExp | commTerm
95  commTerm -> skip | 'if' boolExp 'then' commTerm 'else' commTerm 'end'
96              | 'while' boolExp 'do' commTerm 'end' | <var> ':=' intExp
97              | 'repeat' commTerm 'until' boolExp 'end'
98  -}
99  commExp :: Parser Comm
100 commExp = chainl1 commTerm commOp
101
102 commTerm :: Parser Comm
103 commTerm = do {reserved lis "skip";return Skip}
104           <|> do {reserved lis "if"; p<-boolExp
105                   ;reserved lis "then"; q<-commExp
106                   ;reserved lis "else"; r<-commExp
107                   ;reserved lis "end"; return (Cond p q r)}
108           <|> do {reserved lis "while"; p<-boolExp
109                   ;reserved lis "do"; q<-commExp
110                   ;reserved lis "end"; return (While p q)}
111           <|> do {p<-identifier lis; reservedOp lis ":=";
112                   ;q<-intExp; return (Let p q)}
113
114 commOp :: Parser (Comm -> Comm -> Comm)
115 commOp = do {reservedOp lis ";"; return Seq}
```

```
116
117 -------------------------
118 --- Funcion de parseo ---
119 -------------------------
120 parseComm :: SourceName -> String -> Either ParseError Comm
121 parseComm = parse (totParser commExp)
```

AST.hs

```
 1 module AST where
 2
 3 -- Identificadores de Variable
 4 type Variable = String
 5
 6 -- Expresiones Aritmeticas
 7 data IntExp = Const Int
 8             | Var Variable
 9             | UMinus IntExp
10             | Plus IntExp IntExp
11             | Minus IntExp IntExp
12             | Times IntExp IntExp
13             | Div IntExp IntExp
14             | TCond BoolExp IntExp IntExp -- Ternary conditional
15  deriving Show
16
17 -- Expresiones Booleanas
18 data BoolExp = BTrue
19              | BFalse
20              | Eq IntExp IntExp
21              | Lt IntExp IntExp
22              | Gt IntExp IntExp
23              | And BoolExp BoolExp
24              | Or BoolExp BoolExp
25              | Not BoolExp
26  deriving Show
27
28 -- Comandos (sentencias)
29 -- Observar que solo se permiten variables de un tipo (entero)
30 data Comm = Skip
31           | Let Variable IntExp
32           | Seq Comm Comm
33           | Cond BoolExp Comm Comm
34           | While BoolExp Comm
35           | Repeat Comm BoolExp
36  deriving Show
```

Eval1.hs

```
 1 module Eval1 (eval) where
 2 import AST
 3
 4 -- Estados
 5 -- Variable :: String
 6 type State = [(Variable, Int)]
 7
 8 -- Estado nulo
 9 initState :: State
10 initState = []
11
12 -- Busca el valor de una variable en un estado
13 -- Completar la definicion
```

```
14 lookfor :: Variable -> State -> Int
15 lookfor _ [] = error "Variable no encontrada"
16 lookfor v ((v',s'):xs)
17    | v == v' = s'
18    | otherwise = lookfor v xs
19
20 -- Cambia el valor de una variable en un estado
21 -- Completar la definicion
22 update :: Variable -> Int -> State -> State
23 update v s [] = [(v,s)]
24 update v s ((v',s'):xs)
25    | v == v' = [(v,s)] ++ xs
26    | otherwise = [(v',s')] ++ (update v s xs)
27
28 -- Evalua un programa en el estado nulo
29 eval :: Comm -> State
30 eval p = evalComm p initState
31
32 -- Evalua un comando en un estado dado
33 -- Completar definicion
34 evalComm :: Comm -> State -> State
35 evalComm comm xs = case comm of
36    Skip         -> xs
37    Let    a b   -> update a (evalIntExp b xs) xs
38    Seq    a b   -> evalComm b (evalComm a xs)
39    Cond   a b c -> if evalBoolExp a xs then evalComm b xs else evalComm c xs
40    While  a b   -> if evalBoolExp a xs then evalComm (Seq b d) xs else xs where
       d = While a b
41    Repeat a b   -> evalComm (Seq a (While (Not b) a)) xs
42
43 -- Evalua una expresion entera, sin efectos laterales
44 -- Completar definicion
45 evalIntExp :: IntExp -> State -> Int
46 evalIntExp exp xs = case exp of
47    Const  a     -> (fromInteger a)
48    Var    a     -> lookfor a xs
49    UMinus a     -> -(evalIntExp a xs)
50    Plus   a b   -> (evalIntExp a xs) + (evalIntExp b xs)
51    Minus  a b   -> (evalIntExp a xs) - (evalIntExp b xs)
52    Times  a b   -> (evalIntExp a xs) * (evalIntExp b xs)
53    Div    a b   -> (evalIntExp a xs) `div` (evalIntExp b xs)
54    TCond  a b c -> if (evalBoolExp a xs) then (evalIntExp b xs) else (
      evalIntExp c xs)
55
56 -- Evalua una expresion entera, sin efectos laterales
57 -- Completar definicion
58 evalBoolExp :: BoolExp -> State -> Bool
59 evalBoolExp exp xs = case exp of
60    BTrue   -> True
61    BFalse  -> False
62    Eq a b  -> (evalIntExp a xs) == (evalIntExp b xs)
63    Lt a b  -> (evalIntExp a xs) < (evalIntExp b xs)
64    Gt a b  -> (evalIntExp a xs) > (evalIntExp b xs)
65    And a b -> (evalBoolExp a xs) && (evalBoolExp b xs)
66    Or a b  -> (evalBoolExp a xs) || (evalBoolExp b xs)
67    Not a   -> not (evalBoolExp a xs)
```

Eval2.hs

```
1  module Eval2 (eval) where
2  import AST
3
4  -- Estados
5  -- Variable :: String
6  type State = [(Variable, Int)]
7
8  -- Estado nulo
9  initState :: State
10 initState = []
11
12 -- Busca el valor de una variable en un estado
13 lookfor :: Variable -> State -> Int
14 lookfor _ [] = error "Variable no encontrada"
15
16 lookfor v ((v',s'):xs)
17    | v == v'   = s'
18    | otherwise = lookfor v xs
19
20 -- Cambia el valor de una variable en un estado
21 update :: Variable -> Int -> State -> State
22 update v s [] = [(v,s)]
23
24 update v s ((v',s'):xs)
25    | v == v'   = [(v,s)] ++ xs
26    | otherwise = [(v',s')] ++ (update v s xs)
27
28 -- Evalua un programa en el estado nulo
29 eval :: Comm -> ErrHandler
30 eval p = evalComm p initState
31
32 data ErrHandler = ERROR | OK State deriving Show
33
34 -- Evalua un comando en un estado dado
35 evalComm :: Comm -> State -> ErrHandler
36 evalComm Skip xs = OK xs
37
38 evalComm (Let v e) xs = case (evalIntExp e xs) of
39    Nothing -> ERROR
40    Just x -> OK (update v x xs)
41
42 evalComm (Seq c1 c2) xs = case (evalComm c1 xs) of
43    ERROR -> ERROR
44    OK state -> evalComm c2 state
45
46 evalComm (Cond b c1 c2) xs
47    | (evalBoolExp b xs) = evalComm c1 xs
48    | otherwise          = evalComm c2 xs
49
50 evalComm (While b c) xs
51    | not(evalBoolExp b xs) = OK xs
52    | otherwise = case (evalComm c xs) of
53       ERROR    -> ERROR
54       OK state -> evalComm (While b c) state
55
56 -- Evalua una expresion entera, sin efectos laterales
57 evalIntExp :: IntExp -> State -> Maybe Int
58 evalIntExp (Const a) xs  = Just a
59
```

```
60  evalIntExp (Var a) xs    = Just (lookfor a xs)
61
62  evalIntExp (UMinus a) xs = case (evalIntExp a xs) of
63     Nothing -> Nothing
64     Just x  -> Just ((-1)*x)
65
66  evalIntExp (Plus a b) xs = case (evalIntExp a xs) of
67     Nothing -> Nothing
68     Just x  -> case (evalIntExp b xs) of
69                  Nothing -> Nothing
70                  Just y  -> Just ((+) x y)
71
72  evalIntExp (Minus a b) xs = case (evalIntExp a xs) of
73     Nothing -> Nothing
74     Just x  -> case (evalIntExp b xs) of
75                  Nothing -> Nothing
76                  Just y  -> Just ((-) x y)
77
78  evalIntExp (Times a b) xs = case (evalIntExp a xs) of
79     Nothing -> Nothing
80     Just x  -> case (evalIntExp b xs) of
81                  Nothing -> Nothing
82                  Just y  -> Just ((*) x y)
83
84  evalIntExp (Div a b) xs = case (evalIntExp a xs) of
85     Nothing -> Nothing
86     Just x  -> case (evalIntExp b xs) of
87                  Nothing -> Nothing
88                  Just 0  -> Nothing
89                  Just y  -> Just ((div) x y)
90
91  -- Evalua una expresion entera, sin efectos laterales
92  evalBoolExp :: BoolExp -> State -> Bool
93  evalBoolExp exp xs = case exp of
94     BTrue     -> True
95     BFalse    -> False
96     Eq  e1 e2 -> (evalIntExp e1 xs) == (evalIntExp e2 xs)
97     Lt  e1 e2 -> (evalIntExp e1 xs) < (evalIntExp e2 xs)
98     Gt  e1 e2 -> (evalIntExp e1 xs) > (evalIntExp e2 xs)
99     And e1 e2 -> (evalBoolExp e1 xs) && (evalBoolExp e2 xs)
100    Or  e1 e2 -> (evalBoolExp e1 xs) || (evalBoolExp e2 xs)
101    Not e1    -> not (evalBoolExp e1 xs)
```

Eval3.hs

```
1  module Eval3 (eval) where
2  import AST
3
4  -- Estados
5  -- Variable :: String
6  type State = [(Variable, Int)]
7
8  -- Estado nulo
9  initState :: State
10 initState = []
11
12 -- Busca el valor de una variable en un estado
13 lookfor :: Variable -> State -> Int
14 lookfor _ [] = error "Variable no encontrada"
15
16 lookfor v ((v',s'):xs)
```

```
17     | v == v'   = s'
18     | otherwise = lookfor v xs
19
20  -- Cambia el valor de una variable en un estado
21  update :: Variable -> Int -> State -> State
22  update v s [] = [(v,s)]
23
24  update v s ((v',s'):xs)
25     | v == v'   = [(v,s)] ++ xs
26     | otherwise = [(v',s')] ++ (update v s xs)
27
28  -- Evalua un programa en el estado nulo
29  eval :: Comm -> (ErrHandler, Int)
30  eval p = evalComm p (initState, 0)
31
32  data ErrHandler = ERROR | OK State deriving Show
33
34  -- Evalua un comando en un estado dado
35  evalComm :: Comm -> (State, Int) -> (ErrHandler, Int)
36  evalComm Skip (xs,i) = (OK xs, i)
37
38  evalComm (Let v e) (xs,i) = case eval of
39        Nothing -> (ERROR, c)
40        Just x  -> (OK (update v x xs), c)
41     where (eval, c) = evalIntExp e (xs,i)
42
43  evalComm (Seq c1 c2) xs = case state of
44        ERROR -> (ERROR, c)
45        OK st -> evalComm c2 (st, c)
46     where (state, c) = evalComm c1 xs
47
48  evalComm (Cond b c1 c2) (xs, i)
49     | (evalBoolExp b (xs,i)) = evalComm c1 (xs,i)
50     | otherwise              = evalComm c2 (xs,i)
51
52  evalComm (While b c) (xs,i)
53     | not(evalBoolExp b (xs,i)) = (OK xs, i)
54     | otherwise = case eval of
55           ERROR    -> (ERROR, n)
56           OK state -> evalComm (While b c) (state, n)
57        where (eval, n) = evalComm c (xs,i)
58
59  -- Evalua una expresion entera, sin efectos laterales
60  evalIntExp :: IntExp -> (State, Int) -> (Maybe Int, Int)
61  evalIntExp (Const a) (xs,i)  = (Just (fromInteger a), i)
62  evalIntExp (Var a) (xs,n)    = (Just (lookfor a xs), n)
63
64  evalIntExp (UMinus e) (xs,n) = case eval of
65        Nothing -> (Nothing, i+n)
66        Just x  -> (Just ((-1)*x), i+n+1)
67     where (eval, i) = evalIntExp e (xs,0)
68
69  evalIntExp (Plus e1 e2) (xs,n) = case eval1 of
70        Nothing -> (Nothing, i1+n)
71        Just x  -> case eval2 of
72           Nothing -> (Nothing, i1+i2+n)
73           Just y  -> (Just ((+) x y), i1+i2+n+1)
74     where (eval1, i1) = evalIntExp e1 (xs,0)
75           (eval2, i2) = evalIntExp e2 (xs,0)
76
```

```
77  evalIntExp (Minus e1 e2) (xs,n) = case eval1 of
78        Nothing -> (Nothing, i1+n)
79        Just x  -> case eval2 of
80           Nothing -> (Nothing, i1+i2+n)
81           Just y  -> (Just ((-) x y), i1+i2+n+1)
82     where (eval1, i1) = evalIntExp e1 (xs,0)
83           (eval2, i2) = evalIntExp e2 (xs,0)
84
85  evalIntExp (Times e1 e2) (xs,n) = case eval1 of
86        Nothing -> (Nothing, i1+n)
87        Just x  -> case eval2 of
88           Nothing -> (Nothing, i1+i2+n)
89           Just y  -> (Just ((*) x y), i1+i2+n+1)
90     where (eval1, i1) = evalIntExp e1 (xs,0)
91           (eval2, i2) = evalIntExp e2 (xs,0)
92
93  evalIntExp (Div e1 e2) (xs,n) = case eval1 of
94        Nothing -> (Nothing, i1+n)
95        Just x  -> case eval2 of
96           Nothing -> (Nothing, i1+i2+n)
97           Just 0  -> (Nothing, i1+i2)
98           Just y  -> (Just ((div) x y), i1+i2+n+1)
99     where (eval1, i1) =  evalIntExp e1 (xs,0)
100          (eval2, i2) =  evalIntExp e2 (xs,0)
101
102 -- Evalua una expresion entera, sin efectos laterales
103 evalBoolExp :: BoolExp -> (State,Int) -> Bool
104 evalBoolExp exp xs = case exp of
105    BTrue     -> True
106    BFalse    -> False
107    Eq  e1 e2 -> (evalIntExp e1 xs) == (evalIntExp e2 xs)
108    Lt  e1 e2 -> (evalIntExp e1 xs) < (evalIntExp e2 xs)
109    Gt  e1 e2 -> (evalIntExp e1 xs) > (evalIntExp e2 xs)
110    And e1 e2 -> (evalBoolExp e1 xs) && (evalBoolExp e2 xs)
111    Or  e1 e2 -> (evalBoolExp e1 xs) || (evalBoolExp e2 xs)
112    Not e1    -> not (evalBoolExp e1 xs)
```