



| **UNR** Universidad
Nacional de Rosario

Universidad Nacional de Rosario

FACULTAD DE CIENCIAS EXACTAS, INGENIERÍA Y AGRIMENSURA

ESPECIFICACIÓN DEL LENGUAJE IMPERATIVO SIMPLE

*Primer trabajo practico
Análisis del Lenguajes de Programación*

Autor:
Caporalini, Joaquín
Arroyo, Joaquín

16 de septiembre de 2022

Índice

1. Ejercicio 1	2
2. Ejercicio 4	3
3. Ejercicio 5	3
4. Ejercicio 6	5
4.1. A	5
4.2. B	5
4.3. C	5
4.4. D	5
4.5. E	5
4.6. F	6
4.7. G	6
4.8. H	6
4.9. I	6
4.10. J	6
4.11. K	7
5. Ejercicio 10	7

1. Ejercicio 1

Extensión de la sintáxis abstracta con la regla de producción del operador ternario:

```

intexp ::= nat
        | var
        |  $\neg_a$  intexp
        | intexp + intexp
        | intexp  $\neg_b$  intexp
        | intexp * intexp
        | intexp / intexp
        | boolexpatom ? intexp : intexp
boolexpatom ::= true
              | false
              |  $\neg$ boolexpatom
              | boolexp
boolexp ::= intexp == intexp
          | intexp != intexp
          | intexp < intexp
          | intexp > intexp
          | intexp ^ intexp
          | intexp v intexp
          | boolexpatom
comm ::= skip
      | var = intexp
      | comm; comm
      | if boolexp then comm else comm
      | while boolexp do comm

```

Extensión de la sintáxis concreta con la regla de producción del operador ternario:

```

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

```

```

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

```

2. Ejercicio 4

Extensión de la semántica big-step de expresiones enteras con las reglas para el operador ternario:

$$\frac{\langle p_0, \sigma \rangle \Downarrow_{exp} \mathbf{true} \quad \langle e_0, \sigma \rangle \Downarrow_{exp} n_0}{\langle p_0 ? e_0 : e_1, \sigma \rangle \Downarrow_{exp} n_0} ?\mathbf{TRUE}$$

$$\frac{\langle p_0, \sigma \rangle \Downarrow_{exp} \mathbf{false} \quad \langle e_1, \sigma \rangle \Downarrow_{exp} n_1}{\langle p_0 ? e_0 : e_1, \sigma \rangle \Downarrow_{exp} n_1} ?\mathbf{FALSE}$$

3. Ejercicio 5

Queremos probar que la relación \rightsquigarrow es determinista, es decir que, si $t \rightsquigarrow v_1$ y $t \rightsquigarrow v_2 \Rightarrow v_1 = v_2$. Vamos a realizar la demostración bajo el supuesto de que \Downarrow_{exp} es determinista.

■ Si la última derivación fue ASS entonces

- 1. $\langle e, \sigma \rangle \Downarrow_{exp} n$
- 2. $t = \langle v := e, \sigma \rangle$
- 3. $v_1 = \langle skip, [\sigma|v:n] \rangle$

Como \Downarrow_{exp} es determinista por hipótesis, y la última regla que podemos aplicarle a t es ASS, $v_1 = v_2$.

■ Si la última derivación fue SEQ1 entonces

- 1. $t = \langle \mathbf{skip}; c1, \sigma \rangle$
- 2. $v_1 = \langle c1, \sigma \rangle$

Como la última regla que podemos aplicarle a t es SEQ1, $v_1 = v_2$.

■ Si la última derivación fue SEQ2 entonces

- 1. $\langle c0, \sigma \rangle \rightsquigarrow \langle c0', \sigma' \rangle$
- 2. $t = \langle c0; c1, \sigma \rangle$
- 3. $v_1 = \langle c0'; c1, \sigma' \rangle$

Como la última regla que podemos aplicarle a t es SEQ2 y por H.I $\langle c0, \sigma \rangle \rightsquigarrow \langle c0', \sigma' \rangle$ es determinista, no es posible aplicar SEQ2 con un antecedente diferente, por lo que $v_1 = v_2$.

■ Si la última derivación fue IF1 entonces

- 1. $\langle b, \sigma \rangle \Downarrow_{exp} true$
- 2. $t = \langle \mathbf{if } b \mathbf{ then } c0 \mathbf{ else } c1, \sigma \rangle$
- 3. $v_1 = \langle c0, \sigma \rangle$

Como \Downarrow_{exp} es determinista por hipótesis, y la última regla que podemos aplicarle a t es IF1, $v_1 = v_2$.

■ Análogo para IF2.

- Si la última derivación fue WHILE1 entonces

- 1. $\langle b, \sigma \rangle \Downarrow_{exp} true$
- 2. $t = \langle \mathbf{while} \ b \ \mathbf{do} \ c, \sigma \rangle$
- 3. $v1 = \langle c; \mathbf{while} \ b \ \mathbf{do} \ c, \sigma \rangle$

Como \Downarrow_{exp} es determinista por hipótesis, y la última regla que podemos aplicarle a t es WHILE1, $v1 = v2$.

- Si la última derivación fue WHILE2 entonces

- 1. $\langle b, \sigma \rangle \Downarrow_{exp} false$
- 2. $t = \langle \mathbf{while} \ b \ \mathbf{do} \ c, \sigma \rangle$
- 3. $v1 = \langle \mathbf{skip}, \sigma \rangle$

Como \Downarrow_{exp} es determinista por hipótesis, y la última regla que podemos aplicarle a t es WHILE2, $v1 = v2$.

4. Ejercicio 6

4.1. A

$$\frac{\frac{\langle x, [[\sigma|x:2]|y:2] \rangle \Downarrow_{exp} 2 \quad \text{VAR} \quad \frac{\langle 0, [[\sigma|x:2]|y:2] \rangle \Downarrow_{exp} 0 \quad \text{NVAL}}{\langle x > 0, [[\sigma|x:2]|y:2] \rangle \Downarrow_{exp} 2 > 0} \quad \text{LT}}{\frac{\langle \text{while } x > 0 \text{ do } x := x - y, [[\sigma|x:2]|y:2] \rangle \rightsquigarrow \langle x := x - y; \text{while } x > 0 \text{ do } x := x - y, [[\sigma|x:2]|y:2] \rangle}{\langle \text{while } x > 0 \text{ do } x := x - y, [[\sigma|x:2]|y:2] \rangle \rightsquigarrow^* \langle x := x - y; \text{while } x > 0 \text{ do } x := x - y, [[\sigma|x:2]|y:2] \rangle} \text{WHILE1 CLAUSURE}$$

4.2. B

$$\frac{\frac{\frac{\langle x, [[\sigma|x:2]|y:2] \rangle \Downarrow_{exp} 2 \quad \text{VAR} \quad \frac{\langle y, [[\sigma|x:2]|y:2] \rangle \Downarrow_{exp} 2 \quad \text{VAR}}{\langle x - y, [[\sigma|x:2]|y:2] \rangle \Downarrow_{exp} 0} \quad \text{MINUS}}{\langle x := x - y, [[\sigma|x:2]|y:2] \rangle \rightsquigarrow \langle \text{skip}, [[\sigma|x:0]|y:2] \rangle} \quad \text{ASS}}{\frac{\langle x := x - y; \text{while } x > 0 \text{ do } x := x - y, [[\sigma|x:2]|y:2] \rangle \rightsquigarrow \langle \text{skip}; \text{while } x > 0 \text{ do } x := x - y, [[\sigma|x:0]|y:2] \rangle}{\langle x := x - y; \text{while } x > 0 \text{ do } x := x - y, [[\sigma|x:2]|y:2] \rangle \rightsquigarrow^* \langle \text{skip}; \text{while } x > 0 \text{ do } x := x - y, [[\sigma|x:0]|y:2] \rangle} \text{SEQ2 CLAUSURE}$$

4.3. C

$$\frac{\text{A} \quad \text{B}}{\langle \text{while } x > 0 \text{ do } x := x - y, [[\sigma|x:2]|y:2] \rangle \rightsquigarrow^* \langle \text{skip}; \text{while } x > 0 \text{ do } x := x - y, [[\sigma|x:0]|y:2] \rangle} \text{TRANSITIVE}$$

4.4. D

$$\frac{\langle \text{skip}; \text{while } x > 0 \text{ do } x := x - y, [[\sigma|x:0]|y:2] \rangle \rightsquigarrow \langle \text{while } x > 0 \text{ do } x := x - y, [[\sigma|x:0]|y:2] \rangle}{\langle \text{skip}; \text{while } x > 0 \text{ do } x := x - y, [[\sigma|x:0]|y:2] \rangle \rightsquigarrow^* \langle \text{while } x > 0 \text{ do } x := x - y, [[\sigma|x:0]|y:2] \rangle} \text{SEQ1 CLAUSURE}$$

4.5. E

$$\frac{\text{C} \quad \text{D}}{\langle \text{while } x > 0 \text{ do } x := x - y, [[\sigma|x:2]|y:2] \rangle \rightsquigarrow^* \langle \text{while } x > 0 \text{ do } x := x - y, [[\sigma|x:0]|y:2] \rangle} \text{TRANSITIVE}$$

4.6. F

$$\begin{array}{c}
\frac{\langle x, [[\sigma|x:0]|y:2] \rangle \Downarrow_{exp} 0 \quad \mathbf{VAR}}{\langle x > 0, [[\sigma|x:0]|y:2] \rangle \Downarrow_{exp} 0 > 0} \quad \frac{\langle y, [[\sigma|x:2]|y:2] \rangle \Downarrow_{exp} 2 \quad \mathbf{VAR}}{\mathbf{LT}} \\
\frac{\langle \mathbf{while} \ x > 0 \ \mathbf{do} \ x := x - y, [[\sigma|x:0]|y:2] \rangle \rightsquigarrow \langle \mathbf{skip}, [[\sigma|x:0]|y:2] \rangle \quad \mathbf{WHILE2}}{\langle \mathbf{while} \ x > 0 \ \mathbf{do} \ x := x - y, [[\sigma|x:0]|y:2] \rangle \rightsquigarrow^* \langle \mathbf{skip}, [[\sigma|x:0]|y:2] \rangle} \mathbf{CLAUSURE}
\end{array}$$

4.7. G

$$\frac{\mathbf{E} \quad \mathbf{F}}{\langle \mathbf{while} \ x > 0 \ \mathbf{do} \ x := x - y, [[\sigma|x:2]|y:2] \rangle \rightsquigarrow^* \langle \mathbf{skip}, [[\sigma|x:0]|y:2] \rangle} \mathbf{TRANSITIVE}$$

4.8. H

$$\begin{array}{c}
\frac{\langle x, [[\sigma|x:1]|y:2] \rangle \Downarrow_{exp} 1 \quad \mathbf{VAR} \quad \frac{\langle y, [[\sigma|x:1]|y:2] \rangle \Downarrow_{exp} 2 \quad \mathbf{VAR}}{\mathbf{LT}}}{\langle x := x > y ? y * 2 : y, [[\sigma|x:1]|y:2] \rangle \Downarrow_{exp} 1 > 2} \quad \frac{\langle y, [[\sigma|x:1]|y:2] \rangle \Downarrow_{exp} 2 \quad \mathbf{VAR}}{\mathbf{?FALSE}} \\
\frac{\langle x := x > y ? y * 2 : y, [[\sigma|x:1]|y:2] \rangle \Downarrow_{exp} 2 \quad \mathbf{ASS}}{\langle x := x > y ? y * 2 : y, [[\sigma|x:1]|y:2] \rangle \rightsquigarrow \langle \mathbf{skip}, [[\sigma|x:2]|y:2] \rangle} \mathbf{SEQ2} \\
\frac{\langle x := x > y ? y * 2 : y; \mathbf{while} \ x > 0 \ \mathbf{do} \ x := x - y, [[\sigma|x:1]|y:2] \rangle \rightsquigarrow \langle \mathbf{skip}; \mathbf{while} \ x > 0 \ \mathbf{do} \ x := x - y, [[\sigma|x:2]|y:2] \rangle}{\langle x := x > y ? y * 2 : y; \mathbf{while} \ x > 0 \ \mathbf{do} \ x := x - y, [[\sigma|x:1]|y:2] \rangle \rightsquigarrow^* \langle \mathbf{skip}; \mathbf{while} \ x > 0 \ \mathbf{do} \ x := x - y, [[\sigma|x:2]|y:2] \rangle} \mathbf{CLAUSURE}
\end{array}$$

4.9. I

$$\frac{\langle \mathbf{skip}; \mathbf{while} \ x > 0 \ \mathbf{do} \ x := x - y, [[\sigma|x:2]|y:2] \rangle \rightsquigarrow \langle \mathbf{while} \ x > 0 \ \mathbf{do} \ x := x - y, [[\sigma|x:2]|y:2] \rangle \quad \mathbf{SEQ1}}{\langle \mathbf{skip}; \mathbf{while} \ x > 0 \ \mathbf{do} \ x := x - y, [[\sigma|x:2]|y:2] \rangle \rightsquigarrow^* \langle \mathbf{while} \ x > 0 \ \mathbf{do} \ x := x - y, [[\sigma|x:2]|y:2] \rangle} \mathbf{CLAUSURE}$$

4.10. J

$$\frac{\mathbf{H} \quad \mathbf{I}}{\langle x := x > y ? y * 2 : y; \mathbf{while} \ x > 0 \ \mathbf{do} \ x := x - y, [[\sigma|x:1]|y:2] \rangle \rightsquigarrow^* \langle \mathbf{while} \ x > 0 \ \mathbf{do} \ x := x - y, [[\sigma|x:2]|y:2] \rangle} \mathbf{TRANSITIVE}$$

4.11. K

$$\frac{\text{J} \quad \text{G}}{\langle x := x > y ? y * 2 : y; \text{while } x > 0 \text{ do } x := x - y, [[\sigma | x : 1] | y : 2] \rangle \rightsquigarrow^* \langle \text{skip}, [[\sigma | x : 0] | y : 2] \rangle} \text{TRANSITIVE}$$

K es la ultima parte del árbol

5. Ejercicio 10

Extensión de la sintáxis abstracta con la regla de producción del comando repeat:

$\text{comm} ::= \dots \mid \text{repeat } \text{comm} \text{ until } \text{boolexp}$

Extensión de la semantica con la regla para el comando repeat:

$$\frac{}{\langle \text{repeat } c \text{ until } b, \sigma \rangle \rightsquigarrow \langle c; \text{while } \neg b \text{ do } c, \sigma \rangle} \text{REPEAT}$$


```

1  module Parser where
2
3  import      Text.ParserCombinators.Parsec
4  import      Text.Parsec.Token
5  import      Text.Parsec.Language      ( emptyDef )
6  import      AST
7
8  -----
9  -- Funcion para facilitar el testing del parser.
10 totParser :: Parser a -> Parser a
11 totParser p = do
12   whiteSpace lis
13   t <- p
14   eof
15   return t
16
17 -- Analizador de Tokens
18 lis :: TokenParser u
19 lis = makeTokenParser
20   (emptyDef
21    { commentStart    = "/*"
22    , commentEnd      = "*/"
23    , commentLine     = "//"
24    , opLetter        = char '='
25    , reservedNames   = ["true", "false", "if", "else", "while", "skip", "do", "for"]
26    , reservedOpNames = [ "+",
27                          "-",
28                          "*",
29                          "/",
30                          "<",
31                          ">",
32                          "&&",
33                          "||",
34                          "!",
35                          "=",
36                          "==",
37                          "!=",
38                          ";",
39                          ",",
40                          "?",
41                          ":"
42    ]
43   }
44   )
45
46 -----
47 -- Parser de expresiones enteras
48 -----
49
50 addMinusOp = do {reservedOp lis "+"; return Plus}
51             <|> do {reservedOp lis "-"; return Minus}
52
53 timesDivOp = do {reservedOp lis "*"; return Times}
54             <|> do {reservedOp lis "/"; return Div}
55
56 intexp :: Parser (Exp Int)
57 intexp = ternParser
58
59 ternParser :: Parser (Exp Int)
60 ternParser = try (do b <- boolAtomParser
61                    reservedOp lis "?"

```

```

62         i1 <- ternParser
63         reservedOp lis ":"
64         ECond b i1 <$> ternParser)
65     <|>
66     addMinusParser
67
68 addMinusParser :: Parser (Exp Int)
69 addMinusParser = timesDivParser `chainl1` addMinusOp
70
71 timesDivParser :: Parser (Exp Int)
72 timesDivParser = intFactorParser `chainl1` timesDivOp
73
74 intFactorParser :: Parser (Exp Int)
75 intFactorParser = parens lis intexp
76     <|>
77     try (do n <- natural lis
78           return (Const (fromIntegral n)))
79     <|>
80     try (do i <- identifier lis
81           return (Var i))
82     <|>
83     uMinusParser
84
85 uMinusParser :: Parser (Exp Int)
86 uMinusParser = do reservedOp lis "-"
87                 UMinus <$> intFactorParser
88
89 -----
90 -- Parser de expresiones booleanas
91 -----
92
93 boolCompOp = do {reservedOp lis "==" ; return Eq}
94             <|> do {reservedOp lis "!=" ; return NEq}
95             <|> do {reservedOp lis "<" ; return Lt}
96             <|> do {reservedOp lis ">" ; return Gt}
97
98 boolOp = do {reservedOp lis "&&" ; return And}
99           <|> do {reservedOp lis "||" ; return Or}
100
101 boolexp :: Parser (Exp Bool)
102 boolexp = andOrParser
103
104 andOrParser :: Parser (Exp Bool)
105 andOrParser = boolCompParser `chainl1` boolOp
106
107 boolCompParser :: Parser (Exp Bool)
108 boolCompParser = try (do i1 <- intexp
109                        b <- boolCompOp
110                        b i1 <$> intexp)
111     <|>
112     boolAtomParser
113
114 boolAtomParser :: Parser (Exp Bool)
115 boolAtomParser = try (parens lis boolexp)
116     <|>
117     do reserved lis "true"
118       return BTrue
119     <|>
120     do reserved lis "false"
121       return BFalse
122     <|>

```

```

123         do reservedOp lis "!"
124         Not <$> boolAtomParser
125
126     -----
127     -- Parser de comandos
128     -----
129
130     semicolon = do {reservedOp lis ";" ; return Seq}
131
132     comm :: Parser Comm
133     comm = (skipParser
134         <|> try letParser
135         <|> ifThenParser
136         <|> whileParser)
137         `chainr1` semicolon
138
139     skipParser :: Parser Comm
140     skipParser = do reserved lis "skip"
141                 return Skip
142
143     letParser :: Parser Comm
144     letParser = do v <- identifier lis
145                 reservedOp lis "="
146                 Let v <$> intexp
147
148     ifThenParser :: Parser Comm
149     ifThenParser = do reserved lis "if"
150                     b <- boolexp
151                     c1 <- braces lis comm
152                     IfThenElse b c1 <$> elseParser
153
154     elseParser :: Parser Comm
155     elseParser = do reserved lis "else"
156                 braces lis comm
157                 <|>
158                 return Skip
159
160     whileParser :: Parser Comm
161     whileParser = do reserved lis "while"
162                     b <- boolexp
163                     c <- braces lis comm
164                     return (While b c)
165
166     -----
167     -- Función de parseo
168     -----
169     parseComm :: SourceName -> String -> Either ParseError Comm
170     parseComm = parse (totParser comm)
171

```

```

1  module Eval1
2      ( eval
3        , State
4      )
5  where
6
7      import           AST
8      import qualified Data.Map.Strict           as M
9      import           Data.Strict.Tuple
10
11     -- Estados
12     type State = M.Map Variable Int
13
14     -- Estado nulo
15     initState :: State
16     initState = M.empty
17
18     -- Busca el valor de una variable en un estado
19     lookfor :: Variable -> State -> Int
20     lookfor v s = case M.lookup v s of
21         Just x -> x
22
23     -- Cambia el valor de una variable en un estado
24     update :: Variable -> Int -> State -> State
25     update = M.insert
26
27     -- Evalua un programa en el estado nulo
28     eval :: Comm -> State
29     eval p = stepCommStar p initState
30
31     -- Evalua multiples pasos de un comando en un estado,
32     -- hasta alcanzar un Skip
33     stepCommStar :: Comm -> State -> State
34     stepCommStar Skip s = s
35     stepCommStar c      s = Data.Strict.Tuple.uncurry stepCommStar $ stepComm c s
36
37     -- Evalua un paso de un comando en un estado dado
38     stepComm :: Comm -> State -> Pair Comm State
39     stepComm (Let v x) s = Skip :: (update v (evalExp x s) s)
40     stepComm (IfThenElse b c1 c2) s = if evalExp b s
41                                         then c1 :: s
42                                         else c2 :: s
43
44     stepComm (While b c) s = if evalExp b s
45                               then Seq c (While b c) :: s
46                               else Skip :: s
47
48     stepComm (Seq Skip c) s = c :: s
49     stepComm (Seq c cs) s = let newC :: newS = stepComm c s
50                               in stepComm (Seq newC cs) newS
51
52     -- Evalua una expresion
53     evalExp :: Exp a -> State -> a
54     evalExp (Const x) s = x
55     evalExp (Var v) s = lookfor v s
56     evalExp (UMinus e) s = -(evalExp e s)
57     evalExp (Plus x y) s = evalExp x s + evalExp y s
58     evalExp (Minus x y) s = evalExp x s - evalExp y s
59     evalExp (Times x y) s = evalExp x s * evalExp y s
60     evalExp (Div x y) s = evalExp x s `div` evalExp y s
61     evalExp (ECond b x y) s = if evalExp b s
62                               then evalExp x s
63                               else evalExp y s

```

```
62 evalExp BTrue s = True
63 evalExp BFalse s = False
64 evalExp (Lt x y) s = evalExp x s < evalExp y s
65 evalExp (Gt x y) s = evalExp x s > evalExp y s
66 evalExp (Eq x y) s = evalExp x s == evalExp y s
67 evalExp (NEq x y) s = evalExp x s /= evalExp y s
68 evalExp (And x y) s = evalExp x s && evalExp y s
69 evalExp (Or x y) s = evalExp x s || evalExp y s
70 evalExp (Not b) s = not (evalExp b s)
```

```

1  module Eval2
2      ( eval
3        , State
4      )
5  where
6
7      import           AST
8      import qualified Data.Map.Strict           as M
9      import           Data.Strict.Tuple
10
11     -- Estados
12     type State = M.Map Variable Int
13
14     -- Estado nulo
15     initState :: State
16     initState = M.empty
17
18     -- Busca el valor de una variable en un estado
19     lookfor :: Variable -> State -> Either Error Int
20     lookfor v s = case M.lookup v s of
21         Just x -> Right x
22         _      -> Left  UndefVar
23
24     -- Cambia el valor de una variable en un estado
25     update :: Variable -> Int -> State -> State
26     update = M.insert
27
28     -- Evalua un programa en el estado nulo
29     eval :: Comm -> Either Error State
30     eval p = stepCommStar p initState
31
32     -- Evalua multiples pasos de un comando en un estado,
33     -- hasta alcanzar un Skip
34     stepCommStar :: Comm -> State -> Either Error State
35     stepCommStar Skip s = return s
36     stepCommStar c      s = do
37         (c' ::!) s' <- stepComm c s
38         stepCommStar c' s'
39
40     -- Evalua un paso de un comando en un estado dado
41     stepComm :: Comm -> State -> Either Error (Pair Comm State)
42     stepComm (Let v x) s = case evalExp x s of
43         Right n -> Right (Skip ::! update v n s)
44         Left e  -> Left e
45     stepComm (IfThenElse b c1 c2) s = case evalExp b s of
46         Right True -> Right (c1 ::! s)
47         Right False -> Right (c2 ::! s)
48         Left e      -> Left e
49     stepComm (While b c) s = case evalExp b s of
50         Right True -> Right (Seq c (While b c) ::! s)
51         Right False -> Right (Skip ::! s)
52         Left e      -> Left e
53     stepComm (Seq Skip c) s = Right (c ::! s)
54     stepComm (Seq c cs) s = case stepComm c s of
55         Right (c' ::! s') -> stepComm (Seq c' cs) s'
56         Left e            -> Left e
57
58     -- Evalua una expresion
59     evalExp :: Exp a -> State -> Either Error a
60     evalExp (Const x) s = Right x
61     evalExp (Var v) s = case lookfor v s of

```

```

62         Right x -> Right x
63         Left e -> Left e
64 evalExp (UMinus e) s = case evalExp e s of
65         Right x -> Right (-x)
66         Left e -> Left e
67 evalExp (Plus x y) s = evalExp' x y s (+)
68 evalExp (Minus x y) s = evalExp' x y s (-)
69 evalExp (Times x y) s = evalExp' x y s (*)
70 evalExp (Div x y) s = case evalExp x s of
71         Right x' -> case evalExp y s of
72             Right 0 -> Left DivByZero
73             Right y' -> Right (x' `div` y')
74             Left e1 -> Left e1
75         Left e2 -> Left e2
76 evalExp (ECond b x y) s = case evalExp b s of
77         Right True -> evalExp x s
78         Right False -> evalExp y s
79         Left e -> Left e
80 evalExp BTrue s = Right True
81 evalExp BFalse s = Right False
82 evalExp (Lt x y) s = evalExp' x y s (<)
83 evalExp (Gt x y) s = evalExp' x y s (>)
84 evalExp (Eq x y) s = evalExp' x y s (==)
85 evalExp (NEq x y) s = evalExp' x y s (/=)
86 evalExp (And x y) s = evalExp' x y s (&&)
87 evalExp (Or x y) s = evalExp' x y s (||)
88 evalExp (Not b) s = case evalExp b s of
89         Right x -> Right x
90         Left e -> Left e
91
92 -- Evaluador auxiliar de expresiones, recibe un operador y se lo aplica
93 -- a las dos expresiones recibidas.
94 evalExp' :: Exp a -> Exp a -> State -> (a -> a -> b) -> Either Error b
95 evalExp' x y s op = case evalExp x s of
96         Right x' -> case evalExp y s of
97             Right y' -> Right (op x' y')
98             Left e1 -> Left e1
99         Left e2 -> Left e2

```

```

1  module Eval3
2      ( eval
3      , State
4      )
5  where
6
7      import           AST
8      import qualified Data.Map.Strict           as M
9      import           Data.Strict.Tuple
10
11  -- Estados
12  type State = (M.Map Variable Int, Integer)
13
14  -- Estado nulo
15  initState :: State
16  initState = (M.empty, 0)
17
18  -- Busca el valor de una variable en un estado
19  lookfor :: Variable -> State -> Either Error Int
20  lookfor v (s, _) = case M.lookup v s of
21      Just x -> Right x
22      _      -> Left UndefVar
23
24  -- Cambia el valor de una variable en un estado
25  update :: Variable -> Int -> State -> State
26  update x v (s, w) = (M.insert x v s, w)
27
28  -- Suma un costo dado al estado
29  addWork :: Integer -> State -> State
30  addWork n (s, w) = (s, w+n)
31
32  -- Evalua un programa en el estado nulo
33  eval :: Comm -> Either Error State
34  eval p = stepCommStar p initState
35
36  -- Evalua multiples pasos de un comando en un estado,
37  -- hasta alcanzar un Skip
38  stepCommStar :: Comm -> State -> Either Error State
39  stepCommStar Skip s = return s
40  stepCommStar c      s = do
41      (c' ::! s') <- stepComm c s
42      stepCommStar c' s'
43
44  -- Evalua un paso de un comando en un estado dado
45  stepComm :: Comm -> State -> Either Error (Pair Comm State)
46  stepComm (Let v x) s = case evalExp x s of
47      Right (n ::! s') -> Right (Skip ::! (update v n s'))
48      Left e          -> Left e
49  stepComm (IfThenElse b c1 c2) s = case evalExp b s of
50      Right (True ::! s') -> Right (c1 ::! s')
51      Right (False ::! s') -> Right (c2 ::! s')
52      Left e              -> Left e
53  stepComm (While b c) s = case evalExp b s of
54      Right (True ::! s') -> Right (Seq c (While b c) ::! s')
55      Right (False ::! s') -> Right (Skip ::! s')
56      Left e              -> Left e
57  stepComm (Seq Skip c) s = Right (c ::! s)
58  stepComm (Seq c cs) s = case stepComm c s of
59      Right (c' ::! s') -> stepComm (Seq c' cs) s'
60      Left e            -> Left e
61

```



```

62  -- Evalua una expresion
63  evalExp :: Exp a -> State -> Either Error (Pair a State)
64  evalExp (Const x) s = Right (x ::! (addWork 0 s))
65  evalExp (Var v) s = case lookfor v s of
66      Right x -> Right (x ::! (addWork 0 s))
67      Left e -> Left e
68  evalExp (UMinus e) s = case evalExp e s of
69      Right (x ::! s') -> Right (-x ::! (addWork 1 s'))
70      Left e -> Left e
71  evalExp (Plus x y) s = evalExp' x y s 2 (+)
72  evalExp (Minus x y) s = evalExp' x y s 2 (-)
73  evalExp (Times x y) s = evalExp' x y s 3 (*)
74  evalExp (Div x y) s = case evalExp x s of
75      Right (x' ::! s1) -> case evalExp y s1 of
76          Right (0 ::! _) -> Left DivByZero
77          Right (y' ::! s2) -> Right (x' `div` y' ::! (addWork 3 s2))
78          Left e1 -> Left e1
79      Left e2 -> Left e2
80  evalExp (ECond b x y) s = case evalExp b s of
81      Right (True ::! s') -> evalExp x (addWork 1 s')
82      Right (False ::! s') -> evalExp y (addWork 1 s')
83      Left e -> Left e
84  evalExp BTrue s = Right (True ::! s)
85  evalExp BFalse s = Right (False ::! s)
86  evalExp (Lt x y) s = evalExp' x y s 2 (<)
87  evalExp (Gt x y) s = evalExp' x y s 2 (>)
88  evalExp (Eq x y) s = evalExp' x y s 2 (==)
89  evalExp (NEq x y) s = evalExp' x y s 2 (/=)
90  evalExp (And x y) s = evalExp' x y s 2 (&&)
91  evalExp (Or x y) s = evalExp' x y s 2 (||)
92  evalExp (Not b) s = case evalExp b s of
93      Right (x ::! s') -> Right (x ::! (addWork 1 s'))
94      Left e -> Left e
95
96  -- Evaluador auxiliar de expresiones, ademas recibe el trabajo de aplicarle el operador
97  -- a las dos expresiones recibidas y lo agrega al estado.
98  evalExp' :: Exp a -> Exp a -> State -> Integer -> (a -> a -> b) -> Either Error (Pair b State)
99  evalExp' x y s w op = case evalExp x s of
100      Right (x' ::! s1) -> case evalExp y s1 of
101          Right (y' ::! s2) -> Right ((op x' y') ::! (addWork w s2))
102          Left e1 -> Left e1
103      Left e2 -> Left e2

```