

# **PROCESADORES DE LENGUAJES**

**Curso 23/24**

## **Práctica: Desarrollo de procesador del lenguaje Tiny**

G01

Esther Babon Arcauz

Pablo Campo Gómez

Claudia López-Mingo Moreno

José Antonio Ruiz Heredia

# Índice

<b>Fase 1: Desarrollo de analizador léxico para Tiny(0) y Tiny</b>	<b>3</b>
1.1. Enumeración de las clases léxicas de Tiny(0):	3
1.2. Especificación formal del léxico del lenguaje mediante definiciones regulares:	4
1.3. Diseño de un analizador léxico para el lenguaje mediante un diagrama de transiciones:	5
1.4. Enumeración de las clases léxicas de Tiny:	6
1.5. Especificación formal del léxico del lenguaje mediante definiciones regulares:	7
<b>Fase 2: Desarrollo de analizador sintáctico para Tiny(0) y Tiny</b>	<b>9</b>
2.1. Desarrollo manual de un analizador sintáctico para Tiny(0).	9
2.1.1. Especificación sintáctica (gramática) para Tiny(0).	9
2.1.2. Acondicionamiento de la gramática para permitir la implementación de un analizador sintáctico descendente predictivo recursivo.	10
2.1.3. Directores de cada regla de la gramática acondicionada.	12
2.2. Desarrollo de analizadores sintácticos descendentes y ascendentes para Tiny.	16
2.2.1. Especificación sintáctica (gramática) para Tiny.	16
2.2.2. Acondicionamiento de la gramática para permitir la implementación de un analizador sintáctico descendente predictivo recursivo.	19
<b>Fase 3: Desarrollo de constructores de ASTs para Tiny</b>	<b>22</b>
3.1. Especificación de la sintaxis abstracta de Tiny mediante la enumeración de las firmas (cabeceras) de las funciones constructoras de ASTs.	22
3.2. Especificación del constructor de ASTs mediante una gramática s-atribuida.	26
3.3. Acondicionamiento de dicha especificación para permitir la implementación descendente.	32
3.4. Especificación de un procesamiento que imprima los tokens del programa leído	39
<b>Fase 4: Finalización del procesador para Tiny</b>	<b>46</b>
4.2 Especificación del procedimiento de comprobación de tipos.	55
4.3 Especificación del procesamiento de asignación de espacio.	68
4.4 Descripción del repertorio de instrucciones de la máquina-p necesario para soportar la traducción de Tiny a código-p.	74
4.5 Especificación del procesamiento de etiquetado.	77
4.6 Especificación del procesamiento de generación de código.	88

# Fase 1: Desarrollo de analizador léxico para Tiny(0) y Tiny

## 1.1. Enumeración de las clases léxicas de Tiny(0):

Aunque no exista una receta universal para identificar las clases léxicas, vamos a seguir algunas guías:

- Cada símbolo de puntuación y cada operador da lugar a una clase léxica univaluada separada.
- Cada palabra reservada da lugar a una clase léxica univaluada separada.
- Cada tipo literal a una clase léxica multivaluada.
- Lo mismo pasa con los identificadores: clase léxica multivaluada identificador.

En Tiny(0) encontramos las siguientes clases léxicas:

- **Variables:** Palabras reservadas que tienen asociadas un valor específico. Comienzan con una letra o un guión bajo, seguido de letras, dígitos o más guiones bajos.
- **Literales enteros:** Número incluido en el conjunto de los números enteros. Pueden incluir un signo opcional (+ o -) seguido de una secuencia de uno o más dígitos, sin ceros no significativos al principio.
- **Literales reales:** Número incluido en el conjunto de los números reales. Pueden tener las siguientes partes:
  - Parte entera. Sigue el formato de los literales enteros.
  - Parte decimal: Comienza con un punto seguido de uno o más dígitos.
  - Parte exponencial: Comienza con 'e' o 'E' seguido de uno o más dígitos.
- **Literales booleanos:** Representación de un valor de tipo booleano, es decir que se incluye dentro del conjunto de los valores 'true' y 'false'.
- **Operadores:** Símbolos y signos que se utilizan para realizar las siguientes operaciones:
  - Operadores aritméticos: suma, resta, multiplicación, división.
  - Operadores lógicos: and, not, or.
  - Operadores relacionales: mayor, menor, mayor igual, menor igual, igual, desigual.
  - Operador de asignación.
- **Símbolos de puntuación:** Elementos gramaticales utilizados para estructurar y organizar el código.
  - Paréntesis de apertura y paréntesis de cierre.
- **Palabras reservadas:** Palabras especiales que tienen un significado específico en el lenguaje y no pueden ser usadas como identificadores.

- Tipos de datos: int, real, bool.
- Valores: true, false.
- Operadores: and, not, or.

## 1.2. Especificación formal del léxico del lenguaje mediante definiciones regulares:

### • Alfabetos

- letra  $\equiv [a-z, A-Z, \_]$
- digitoPositivo  $\equiv [1-9]$
- digito  $\equiv \{\text{digitoPositivo}\} \mid 0$
- parteEntera  $\equiv (\{\text{digitoPositivo}\} \{\text{digito}\}^* \mid 0)$
- parteDecimal  $\equiv \backslash. (\{\text{digito}\}^* \{\text{digitoPositivo}\} \mid 0)$
- parteExponencial  $\equiv [\backslash e, \backslash E] [\backslash +, \backslash -]? (\{\text{digitoPositivo}\} \{\text{digito}\}^* \mid 0)$

### • Palabras reservadas

- int  $\equiv (i|I)(n|N)(t|T)$
- real  $\equiv (r|R)(e|E)(a|A)(l|L)$
- bool  $\equiv (b|B)(o|O)(o|O)(l|L)$
- true  $\equiv (t|T)(r|R)(u|U)(e|E)$
- false  $\equiv (f|F)(a|A)(l|L)(s|S)(e|E)$
- and  $\equiv (a|A)(n|N)(d|D)$
- not  $\equiv (n|N)(o|O)(t|T)$
- or  $\equiv (o|O)(r|R)$

### • Literales

- literalEntero  $\equiv [\backslash +, \backslash -]? \{\text{parteEntera}\}$
- literalReal  $\equiv \{\text{literalEntero}\} (\{\text{parteDecimal}\} \{\text{parteExponencial}\} | (\{\text{parteDecimal}\} \{\text{parteExponencial}\}) )$
- variable  $\equiv \{\text{letra}\} (\{\text{letra}\} \mid \{\text{digito}\})^*$

### • Operadores

- suma  $\equiv \backslash +$
- resta  $\equiv \backslash -$
- mul  $\equiv \backslash *$
- div  $\equiv /$
- mayor  $\equiv \backslash >$
- menor  $\equiv \backslash <$
- mayorIgual  $\equiv \backslash > =$
- menorIgual  $\equiv \backslash < =$
- igual  $\equiv ==$
- desigual  $\equiv !=$
- asig  $\equiv =$

- `parA`  $\equiv$  `\(`
- `parC`  $\equiv$  `\)`
- `arroba`  $\equiv$  `\@`
- `ampersand2`  $\equiv$  `\&\&`
- `puntoYComa`  $\equiv$  `\;`
- `llaveA`  $\equiv$  `\{`
- `llaveC`  $\equiv$  `\}`
- 

```

separador ≡ [ ,\t, \r, \b, \n]
comentario ≡ ##([^\n,EOF])*

```

[illegible]

## 1.4. Enumeración de las clases léxicas de Tiny:

Aunque no exista una receta universal para identificar las clases léxicas, vamos a seguir algunas guías:

- Cada símbolo de puntuación y cada operador da lugar a una clase léxica univaluada separada.
- Cada palabra reservada da lugar a una clase léxica univaluada separada.
- Cada tipo literal a una clase léxica multivaluada.
- Lo mismo pasa con los identificadores: clase léxica multivaluada identificador.

En Tiny encontramos las siguientes clases léxicas:

- **Variables:** Palabras reservadas que tienen asociadas un valor específico. Comienzan con una letra o un guión bajo, seguido de letras, dígitos o más guiones bajos.
- **Literales enteros:** Número incluido en el conjunto de los números enteros. Pueden incluir un signo opcional (+ o -) seguido de una secuencia de uno o más dígitos, sin ceros no significativos al principio.
- **Literales reales:** Número incluido en el conjunto de los números reales. Pueden tener las siguientes partes:
  - Parte entera: Sigue el formato de los literales enteros.
  - Parte decimal: Comienza con un punto seguido de uno o más dígitos.
  - Parte exponencial: Comienza con 'e' o 'E' seguido de uno o más dígitos.
- **Literales cadena:** Representa un valor de tipo string, comienza con comillas dobles (") seguidas por la cadena vacía o una cadena que contiene cualquier carácter que no sean comillas y termina con comillas dobles.
- **Operadores:** Símbolos y signos que se utilizan para realizar las siguientes operaciones:
  - Operadores aritméticos: suma, resta, multiplicación, división, módulo.
  - Operadores lógicos: and, not, or.
  - Operadores relacionales: mayor, menor, mayor igual, menor igual, igual, desigual.
  - Operador de asignación.
- **Símbolos de puntuación:** Elementos gramaticales utilizados para estructurar y organizar el código.

- Paréntesis de apertura y cierre, punto y coma, corchete de apertura y cierre, llave de apertura y cierre, punto, & , &&, arroba
- **Palabras reservadas:** Palabras especiales que tienen un significado específico en el lenguaje y no pueden ser usadas como identificadores.
  - Tipos de datos: int, real, bool, string, struct.
  - Valores: true, false, null.
  - Operadores: and, not, or.
  - Condicionales: if, else, while
  - Instrucciones: proc, new, delete, read, write, nl, type, call

## 1.5. Especificación formal del léxico del lenguaje mediante definiciones regulares:

- **Alfabetos**
  - letra  $\equiv [ a-z, A-Z, \_ ]$
  - digitoPositivo  $\equiv [ 1-9 ]$
  - digito  $\equiv \{ \text{digitoPositivo} \} \mid 0$
  - parteEntera  $\equiv \{ \text{digitoPositivo} \} \{ \text{digito} \}^* \mid 0$
  - parteDecimal  $\equiv \backslash. \{ \text{digito} \}^* \{ \text{digitoPositivo} \} \mid 0$
  - parteExponencial  $\equiv [ \backslash e, \backslash E ] [ \backslash +, \backslash - ]^? \{ \text{digitoPositivo} \} \{ \text{digito} \}^* \mid 0$
- **Palabras reservadas**
  - int  $\equiv (i|I)(n|N)(t|T)$
  - real  $\equiv (r|R)(e|E)(a|A)(l|L)$
  - bool  $\equiv (b|B)(o|O)(o|O)(l|L)$
  - string  $\equiv (s|S)(t|T)(r|R)(i|I)(n|N)(g|G)$
  - null  $\equiv (n|N)(u|U)(l|L)(l|L)$
  - proc  $\equiv (p|P)(r|R)(o|O)(c|C)$
  - if  $\equiv (i|I)(f|F)$
  - else  $\equiv (e|E)(l|L)(s|S)(e|E)$
  - while  $\equiv (w|W)(h|H)(i|I)(l|L)(e|E)$
  - struct  $\equiv (s|S)(t|T)(r|R)(u|U)(c|C)(t|T)$
  - new  $\equiv (n|N)(e|E)(w|W)$
  - delete  $\equiv (d|D)(e|E)(l|L)(e|E)(t|T)(e|E)$
  - read  $\equiv (r|R)(e|E)(a|A)(d|D)$
  - write  $\equiv (w|W)(r|R)(i|I)(t|T)(e|E)$
  - nl  $\equiv (n|N)(l|L)$
  - type  $\equiv (t|T)(y|Y)(p|P)(e|E)$
  - call  $\equiv (c|C)(a|A)(l|L)(l|L)$
  - true  $\equiv (t|T)(r|R)(u|U)(e|E)$
  - false  $\equiv (f|F)(a|A)(l|L)(s|S)(e|E)$
  - and  $\equiv (a|A)(n|N)(d|D)$
  - not  $\equiv (n|N)(o|O)(t|T)$
  - or  $\equiv (o|O)(r|R)$

- **Literales**

- $\text{identificador} \equiv \{\text{letra}\}\{\{\text{letra}\}\{\text{digito}\}\}^*$
- $\text{literalEntero} \equiv [\backslash +, \backslash -]^? \{\text{parteEntera}\}$
- $\text{literalReal} \equiv \{\text{literalEntero}\}(\{\text{parteDecimal}\}\{\text{parteExponencial}\}(\{\text{parteDecimal}\}\{\text{parteExponencial}\})$   
 $)$
- $\text{literalCadena} \equiv "([^\"])^*"$

- **Operadores**

- $\text{suma} \equiv \backslash +$
- $\text{resta} \equiv \backslash -$
- $\text{mul} \equiv \backslash *$
- $\text{div} \equiv \backslash /$
- $\text{mod} \equiv \backslash \%$
- $\text{mayor} \equiv \backslash >$
- $\text{menor} \equiv \backslash <$
- $\text{mayorIgual} \equiv \backslash >=$
- $\text{menorIgual} \equiv \backslash <=$
- $\text{igual} \equiv \backslash ==$
- $\text{desigual} \equiv \backslash !=$
- $\text{asig} \equiv \backslash =$
- $\text{parA} \equiv \backslash ($
- $\text{parC} \equiv \backslash )$
- $\text{puntoYComa} \equiv \backslash ;$
- $\text{punto} \equiv \backslash .$
- $\text{corcheteA} \equiv \backslash [$
- $\text{corcheteC} \equiv \backslash ]$
- $\text{llaveA} \equiv \backslash \{$
- $\text{llaveC} \equiv \backslash \}$
- $\text{arroba} \equiv \backslash @$
- $\text{ampersand} \equiv \backslash \&$
- $\text{ampersand2} \equiv \backslash \&\&$

- **Cadenas ignorables**

- $\text{separador} \equiv [ , \backslash t, \backslash r, \backslash b, \backslash n]$
- $\text{comentario} \equiv \#\# [^\n]^*$



## Fase 2: Desarrollo de analizador sintáctico para Tiny(0) y Tiny

### 2.1. Desarrollo manual de un analizador sintáctico para Tiny(0).

#### 2.1.1. Especificación sintáctica (gramática) para Tiny(0).

programa  $\rightarrow$  bloque EOF

bloque  $\rightarrow$  { declaraciones\_opt instrucciones\_opt }

declaraciones\_opt  $\rightarrow$  declaraciones &&

declaraciones\_opt  $\rightarrow \epsilon$

instrucciones\_opt  $\rightarrow$  instrucciones

instrucciones\_opt  $\rightarrow \epsilon$

declaraciones  $\rightarrow$  declaracion\_var ; declaraciones

declaraciones  $\rightarrow$  declaracion\_var

declaracion\_var  $\rightarrow$  tipo identificador

tipo  $\rightarrow$  int

tipo  $\rightarrow$  real

tipo  $\rightarrow$  bool

instrucciones  $\rightarrow$  instrucciones ; instruccion

instrucciones  $\rightarrow$  instruccion

instruccion  $\rightarrow$  @ E0

E0  $\rightarrow$  E1 = E0

E0  $\rightarrow$  E1

E1  $\rightarrow$  E1 op1 E2

E1  $\rightarrow$  E2

E2  $\rightarrow$  E2 + E3

E2  $\rightarrow$  E3 - E3

E2  $\rightarrow$  E3

E3  $\rightarrow$  E4 op3

E3  $\rightarrow$  E4

$E4 \rightarrow E4 \text{ op4 } E5$   
 $E4 \rightarrow E5$

$E5 \rightarrow \text{op5 } E5$   
 $E5 \rightarrow \text{literalEntero}$   
 $E5 \rightarrow \text{literalReal}$   
 $E5 \rightarrow \text{literalBool}$   
 $E5 \rightarrow \text{variable}$   
 $E5 \rightarrow ( E0 )$

$\text{op1} \rightarrow >$   
 $\text{op1} \rightarrow >=$   
 $\text{op1} \rightarrow <$   
 $\text{op1} \rightarrow <=$   
 $\text{op1} \rightarrow ==$   
 $\text{op1} \rightarrow !=$

$\text{op3} \rightarrow \text{and } E3$   
 $\text{op3} \rightarrow \text{or } E4$

$\text{op4} \rightarrow *$   
 $\text{op4} \rightarrow /$

$\text{op5} \rightarrow -$   
 $\text{op5} \rightarrow \text{not}$

2.1.2. Acondicionamiento de la gramática para permitir la implementación de un analizador sintáctico descendente predictivo recursivo.

Original	Acondicionado
$\text{programa} \rightarrow \text{bloque EOF}$	
$\text{bloque} \rightarrow \{ \text{declaraciones\_opt instrucciones\_opt} \}$	
$\text{declaraciones\_opt} \rightarrow \text{declaraciones \&\&}$ $\text{declaraciones\_opt} \rightarrow \epsilon$	
$\text{instrucciones\_opt} \rightarrow \text{instrucciones}$ $\text{instrucciones\_opt} \rightarrow \epsilon$	

declaraciones $\rightarrow$ declaracion_var ; declaraciones declaraciones $\rightarrow$ declaracion_var	declaraciones $\rightarrow$ declaracion_var restodec restodec $\rightarrow$ ; declaraciones restodec $\rightarrow \epsilon$
declaracion_var $\rightarrow$ tipo identificador	
tipo $\rightarrow$ int tipo $\rightarrow$ real tipo $\rightarrow$ bool	
instrucciones $\rightarrow$ instrucciones ; instruccion instrucciones $\rightarrow$ instruccion	Instrucciones $\rightarrow$ Instruccion restolnst restolnst $\rightarrow$ ; Instruccion restolnst restolnst $\rightarrow \epsilon$
instruccion $\rightarrow$ @ E0	
E0 $\rightarrow$ E1 = E0 E0 $\rightarrow$ E1	E0 $\rightarrow$ E1 RE0 RE0 $\rightarrow$ = E0 RE0 $\rightarrow \epsilon$
E1 $\rightarrow$ E1 op1 E2 E1 $\rightarrow$ E2	E1 $\rightarrow$ E2 RE1 RE1 $\rightarrow$ op1 E2 RE1 RE1 $\rightarrow \epsilon$
E2 $\rightarrow$ E2 + E3 E2 $\rightarrow$ E3 - E3 E2 $\rightarrow$ E3	E2 $\rightarrow$ E3 RE2 E2' RE2 $\rightarrow$ - E3 RE2 $\rightarrow \epsilon$ E2' $\rightarrow$ + E3 E2' E2' $\rightarrow \epsilon$
E3 $\rightarrow$ E4 op3 E3 $\rightarrow$ E4	E3 $\rightarrow$ E4 RE3 RE3 $\rightarrow$ op3 RE3 $\rightarrow \epsilon$
E4 $\rightarrow$ E4 op4 E5 E4 $\rightarrow$ E5	E4 $\rightarrow$ E5 RE4 RE4 $\rightarrow$ op4 E5 RE4 RE4 $\rightarrow \epsilon$
E5 $\rightarrow$ op5 E5 E5 $\rightarrow$ literalEntero E5 $\rightarrow$ literalReal E5 $\rightarrow$ literalBool E5 $\rightarrow$ variable E5 $\rightarrow$ ( E0 )	

op1 → > op1 → >= op1 → < op1 → <= op1 → == op1 → != op3 → and E3 op3 → or E4 op4 → * op4 → / op5 → - op5 → not	
---	--

### 2.1.3. Directores de cada regla de la gramática acondicionada.

programa → bloque EOF

- Director: {

bloque → { declaraciones\_opt instrucciones\_opt }

- Director: {

declaraciones\_opt → declaraciones &&

- Dir: **int, real, bool**

declaraciones\_opt → ε

- Dir: ∅

instrucciones\_opt → instrucciones

- Dir: @

instrucciones\_opt → ε

- Dir: ∅

declaraciones → declaracion\_var restodec

- Dir: **int, real, bool**

restodec → ; declaraciones

- Dir: ;

restodec → ε

- Dir: ∅

declaracion\_var → tipo identificador

- Dir: **int, real, bool**

tipo  $\rightarrow$  int

- Dir: **int**

tipo  $\rightarrow$  real

- Dir: **real**

tipo  $\rightarrow$  bool

- Dir: **bool**

Instrucciones  $\rightarrow$  Instruccion restolnst

- Dir: **@**

restolnst  $\rightarrow$  ; Instruccion restolnst

- Dir: **;**

restolnst  $\rightarrow \epsilon$

- Dir:  **$\emptyset$**

instruccion  $\rightarrow$  @ E0

- Director: **@**

E0  $\rightarrow$  E1 RE0

- Dir: **-,not, literalEntero, literalReal, literalBool, variable, (**

RE0  $\rightarrow$  = E0

- Dir: **=**

RE0  $\rightarrow \epsilon$

- Dir:  **$\emptyset$**

E1  $\rightarrow$  E2 RE1

- Dir: **-,not, literalEntero, literalReal, literalBool, variable, (**

RE1  $\rightarrow$  op1 E2 RE1

- Dir: **>, >=, <, <=, ==, !=**

RE1  $\rightarrow \epsilon$

- Dir:  **$\emptyset$**

E2  $\rightarrow$  E3 RE2 E2'

- Dir: **-, not, literalEntero, literalReal, literalBool, variable, (**

RE2  $\rightarrow$  - E3

- Dir: **-**

$RE2 \rightarrow \varepsilon$

- Dir:  $\emptyset$

$E2' \rightarrow + E3 E2'$

- Dir: **+**

$E2' \rightarrow \varepsilon$

- Dir:  $\emptyset$

$E3 \rightarrow E4 RE3$

- Dir: **-, not, literalEntero, literalReal, literalBool, variable, (**

$RE3 \rightarrow op3$

- Dir: **and, or**

$RE3 \rightarrow \varepsilon$

- Dir:  $\emptyset$

$E4 \rightarrow E5 RE4$

- Dir: **-, not, literalEntero, literalReal, literalBool, variable, (**

$RE4 \rightarrow op4 E5 RE4$

- Dir: **\*, /**

$RE4 \rightarrow \varepsilon$

- Dir:  $\emptyset$

$E5 \rightarrow op5 E5$

- Dir: **-, not**

$E5 \rightarrow literalEntero$

- Dir: **literalEntero**

$E5 \rightarrow literalReal$

- Dir: **literalReal**

$E5 \rightarrow literalBool$

- Dir: **literalBool**

$E5 \rightarrow variable$

- Dir: **variable**

$E5 \rightarrow ( E0 )$

- Dir: **(**

op1  $\rightarrow$  >

- Dir: >

op1  $\rightarrow$  >=

- Dir: >=

op1  $\rightarrow$  <

- Dir: <

op1  $\rightarrow$  <=

- Dir: <=

op1  $\rightarrow$  ==

- Dir: ==

op1  $\rightarrow$  !=

- Dir: !=

op3  $\rightarrow$  and E3

- Dir: **and**

op3  $\rightarrow$  or E4

- Dir: **or**

op4  $\rightarrow$  \*

- Dir: \*

op4  $\rightarrow$  /

- Dir: /

op5  $\rightarrow$  -

- Dir: -

op5  $\rightarrow$  not:

- Dir: **not**

## 2.2. Desarrollo de analizadores sintácticos descendentes y ascendentes para Tiny.

### 2.2.1. Especificación sintáctica (gramática) para Tiny.

programa  $\rightarrow$  bloque EOF

bloque  $\rightarrow$  { declaraciones\_opt instrucciones\_opt }

declaraciones\_opt  $\rightarrow$  declaraciones &&

declaraciones\_opt  $\rightarrow \epsilon$

instrucciones\_opt  $\rightarrow$  instrucciones

instrucciones\_opt  $\rightarrow \epsilon$

declaraciones  $\rightarrow$  declaraciones ; declaracion

declaraciones  $\rightarrow$  declaracion

declaracion  $\rightarrow$  declaracion\_var

declaracion  $\rightarrow$  declaracion\_tipo

declaracion  $\rightarrow$  declaracion\_proc

declaracion\_var  $\rightarrow$  tipo identificador

declaracion\_tipo  $\rightarrow$  type tipo identificador

declaracion\_proc  $\rightarrow$  proc identificador ( parametros\_formales\_opt ) bloque

parametros\_formales\_opt  $\rightarrow$  parametros\_formales

parametros\_formales\_opt  $\rightarrow \epsilon$

parametros\_formales  $\rightarrow$  parametros\_formales , parametro\_formal

parametros\_formales  $\rightarrow$  parametro\_formal

parametro\_formal  $\rightarrow$  tipo and\_opt identificador

and\_opt  $\rightarrow$  &

and\_opt  $\rightarrow \epsilon$

tipo  $\rightarrow$  tipo [ literalEntero ]

tipo  $\rightarrow$  tipo1

tipo1  $\rightarrow$  ^ tipo1

tipo1  $\rightarrow$  tipo2



tipo2  $\rightarrow$  identificador  
tipo2  $\rightarrow$  struct { campos }  
tipo2  $\rightarrow$  int  
tipo2  $\rightarrow$  real  
tipo2  $\rightarrow$  bool  
tipo2  $\rightarrow$  string

campos  $\rightarrow$  campos , campo  
campos  $\rightarrow$  campo  
campo  $\rightarrow$  tipo identificador

instrucciones  $\rightarrow$  instrucciones ; instruccion  
instrucciones  $\rightarrow$  instruccion

instruccion  $\rightarrow$  @ E0  
instruccion  $\rightarrow$  if E0 bloque  
instruccion  $\rightarrow$  if E0 bloque else bloque  
instruccion  $\rightarrow$  while E0 bloque  
instruccion  $\rightarrow$  read E0  
instruccion  $\rightarrow$  write E0  
instruccion  $\rightarrow$  nl  
instruccion  $\rightarrow$  new E0  
instruccion  $\rightarrow$  delete E0  
instruccion  $\rightarrow$  call identificador ( parametros\_reales\_opt )  
instruccion  $\rightarrow$  bloque

parametros\_reales\_opt  $\rightarrow$  parametros\_reales  
parametros\_reales\_opt  $\rightarrow$   $\epsilon$

parametros\_reales  $\rightarrow$  parametros\_reales , E0  
parametros\_reales  $\rightarrow$  E0

E0  $\rightarrow$  E1 = E0  
E0  $\rightarrow$  E1

E1  $\rightarrow$  E1 op1 E2  
E1  $\rightarrow$  E2

E2  $\rightarrow$  E2 + E3  
E2  $\rightarrow$  E3 - E3  
E2  $\rightarrow$  E3

E3  $\rightarrow$  E4 op3  
E3  $\rightarrow$  E4

$E4 \rightarrow E4 \text{ op4 } E5$   
 $E4 \rightarrow E5$

$E5 \rightarrow \text{op5 } E5$   
 $E5 \rightarrow E6$

$E6 \rightarrow E6 \text{ op6}$   
 $E6 \rightarrow \text{literalEntero}$   
 $E6 \rightarrow \text{literalReal}$   
 $E6 \rightarrow \text{literalBool}$   
 $E6 \rightarrow \text{literalCadena}$   
 $E6 \rightarrow \text{identificador}$   
 $E6 \rightarrow \text{null}$   
 $E6 \rightarrow ( E0 )$

$\text{op1} \rightarrow >$   
 $\text{op1} \rightarrow >=$   
 $\text{op1} \rightarrow <$   
 $\text{op1} \rightarrow <=$   
 $\text{op1} \rightarrow ==$   
 $\text{op1} \rightarrow !=$

$\text{op3} \rightarrow \text{and } E3$   
 $\text{op3} \rightarrow \text{or } E4$

$\text{op4} \rightarrow *$   
 $\text{op4} \rightarrow /$   
 $\text{op4} \rightarrow \%$

$\text{op5} \rightarrow -$   
 $\text{op5} \rightarrow \text{not}$

$\text{op6} \rightarrow [ E0 ]$   
 $\text{op6} \rightarrow \text{.identificador}$   
 $\text{op6} \rightarrow ^$

2.2.2. Acondicionamiento de la gramática para permitir la implementación de un analizador sintáctico descendente predictivo recursivo.

Original	Acondicionado
programa $\rightarrow$ bloque EOF	
bloque $\rightarrow$ { declaraciones_opt instrucciones_opt } declaraciones_opt $\rightarrow$ declaraciones && declaraciones_opt $\rightarrow \epsilon$ instrucciones_opt $\rightarrow$ instrucciones instrucciones_opt $\rightarrow \epsilon$	
declaraciones $\rightarrow$ declaraciones ; declaracion declaraciones $\rightarrow$ declaracion declaracion $\rightarrow$ declaracion_var declaracion $\rightarrow$ declaracion_tipo declaracion $\rightarrow$ declaracion_proc	declaraciones $\rightarrow$ declaracion declaracion_extra declaracion_extra $\rightarrow$ ; declaraciones declaracion_extra $\rightarrow \epsilon$ declaracion $\rightarrow$ declaracion_var declaracion $\rightarrow$ declaracion_tipo declaracion $\rightarrow$ declaracion_proc
declaracion_var $\rightarrow$ tipo identificador declaracion_tipo $\rightarrow$ type tipo identificador declaracion_proc $\rightarrow$ proc identificador ( parametros_formales_opt ) bloque	
parametros_formales_opt $\rightarrow$ parametros_formales parametros_formales_opt $\rightarrow \epsilon$	
parametros_formales $\rightarrow$ parametros_formales , parametro_formal parametros_formales $\rightarrow$ parametro_formal	parametros_formales $\rightarrow$ parametro_formal parametros_formales' $\rightarrow$ , parametro_formal parametros_formales' $\rightarrow \epsilon$
parametro_formal $\rightarrow$ tipo and_opt identificador and_opt $\rightarrow$ & and_opt $\rightarrow \epsilon$	
tipo $\rightarrow$ tipo [ literalEntero ] tipo $\rightarrow$ tipo1	tipo $\rightarrow$ tipo1 tipo' tipo' $\rightarrow$ [literalEntero] tipo'

<p> <math>\text{tipo1} \rightarrow \wedge \text{tipo1}</math>  <math>\text{tipo1} \rightarrow \text{tipo2}</math>  <math>\text{tipo2} \rightarrow \text{identificador}</math>  <math>\text{tipo2} \rightarrow \text{struct } \{ \text{campos} \}</math>  <math>\text{tipo2} \rightarrow \text{int}</math>  <math>\text{tipo2} \rightarrow \text{real}</math>  <math>\text{tipo2} \rightarrow \text{bool}</math>  <math>\text{tipo2} \rightarrow \text{string}</math> </p>	<p> <math>\text{tipo}' \rightarrow \epsilon</math>  <math>\text{tipo1} \rightarrow \wedge \text{tipo1}</math>  <math>\text{tipo1} \rightarrow \text{tipo2}</math>  <math>\text{tipo2} \rightarrow \text{identificador}</math>  <math>\text{tipo2} \rightarrow \text{struct } \{ \text{campos} \}</math>  <math>\text{tipo2} \rightarrow \text{int}</math>  <math>\text{tipo2} \rightarrow \text{real}</math>  <math>\text{tipo2} \rightarrow \text{bool}</math>  <math>\text{tipo2} \rightarrow \text{string}</math> </p>
<p> <math>\text{campos} \rightarrow \text{campos} , \text{campo}</math>  <math>\text{campos} \rightarrow \text{campo}</math> </p>	<p> <math>\text{campos} \rightarrow \text{campo campos}'</math>  <math>\text{campos}' \rightarrow , \text{campo campos}'</math>  <math>\text{campos}' \rightarrow \epsilon</math> </p>
<p> <math>\text{campo} \rightarrow \text{tipo identificador}</math> </p>	
<p> <math>\text{Instrucciones} \rightarrow \text{instrucciones} ; \text{instruccion}</math>  <math>\text{Instrucciones} \rightarrow \text{instruccion}</math> </p>	<p> <math>\text{Instrucciones} \rightarrow \text{instruccion instrucciones}'</math>  <math>\text{instrucciones}' \rightarrow ; \text{instruccion instrucciones}'</math>  <math>\text{instrucciones}' \rightarrow \epsilon</math> </p>
<p> <math>\text{instruccion} \rightarrow @ \text{E0}</math>  <math>\text{instruccion} \rightarrow \text{if E0 bloque}</math>  <math>\text{instruccion} \rightarrow \text{if E0 bloque else bloque}</math>  <math>\text{instruccion} \rightarrow \text{while E0 bloque}</math>  <math>\text{instruccion} \rightarrow \text{read E0}</math>  <math>\text{instruccion} \rightarrow \text{write E0}</math>  <math>\text{instruccion} \rightarrow \text{nl}</math>  <math>\text{instruccion} \rightarrow \text{new E0}</math>  <math>\text{instruccion} \rightarrow \text{delete E0}</math>  <math>\text{instruccion} \rightarrow \text{call identificador ( parametros_reales_opt )}</math>  <math>\text{instruccion} \rightarrow \text{bloque}</math> </p>	<p> <math>\text{instruccion} \rightarrow \text{if E0 bloque if\_select}</math>  <math>\text{if\_select} \rightarrow \text{else bloque}</math>  <math>\text{if\_else} \rightarrow \epsilon</math>  <math>\text{instruccion} \rightarrow @ \text{E0}</math>  <math>\text{instruccion} \rightarrow \text{while E0 bloque}</math>  <math>\text{instruccion} \rightarrow \text{read E0}</math>  <math>\text{instruccion} \rightarrow \text{write E0}</math>  <math>\text{instruccion} \rightarrow \text{nl}</math>  <math>\text{instruccion} \rightarrow \text{new E0}</math>  <math>\text{instruccion} \rightarrow \text{delete E0}</math>  <math>\text{instruccion} \rightarrow \text{call identificador ( parametros_reales_opt )}</math>  <math>\text{instruccion} \rightarrow \text{bloque}</math> </p>
<p> <math>\text{parametros\_reales\_opt} \rightarrow \text{parametros\_reales}</math>  <math>\text{parametros\_reales\_opt} \rightarrow \epsilon</math> </p>	
<p> <math>\text{parametros\_reales} \rightarrow \text{parametros\_reales} , \text{E0}</math>  <math>\text{parametros\_reales} \rightarrow \text{E0}</math> </p>	<p> <math>\text{parametros\_reales} \rightarrow \text{E0 parametros\_reales}'</math>  <math>\text{parametros\_reales}' \rightarrow , \text{E0 parametros\_reales}'</math>  <math>\text{parametros\_reales}' \rightarrow \epsilon</math> </p>
<p> <math>\text{E0} \rightarrow \text{E1} = \text{E0}</math>  <math>\text{E0} \rightarrow \text{E1}</math> </p>	<p> <math>\text{E0} \rightarrow \text{E1 RE0}</math>  <math>\text{RE0} \rightarrow = \text{E0}</math> </p>

	$RE0 \rightarrow \epsilon$
$E1 \rightarrow E1 \text{ op1 } E2$ $E1 \rightarrow E2$	$E1 \rightarrow E2 RE1$ $RE1 \rightarrow \text{op1 } E2 RE1$ $RE1 \rightarrow \epsilon$
$E2 \rightarrow E2 + E3$ $E2 \rightarrow E3 - E3$ $E2 \rightarrow E3$	$E2 \rightarrow E3 RE2 E2'$ $RE2 \rightarrow - E3$ $RE2 \rightarrow \epsilon$ $E2' \rightarrow + E3 E2'$ $E2' \rightarrow \epsilon$
$E3 \rightarrow E4 \text{ and } E3$ $E3 \rightarrow E4 \text{ or } E4$ $E3 \rightarrow E4$	$E3 \rightarrow E4 RE3$ $RE3 \rightarrow \text{op3}$ $RE3 \rightarrow \epsilon$
$E4 \rightarrow E4 \text{ op4 } E5$ $E4 \rightarrow E5$	$E4 \rightarrow E5 RE4$ $RE4 \rightarrow \text{op4 } E5 RE4$ $RE4 \rightarrow \epsilon$
$E5 \rightarrow \text{op5 } E5$ $E5 \rightarrow E6$	
$E6 \rightarrow E6 \text{ op6}$ $E6 \rightarrow \text{literalEntero}$ $E6 \rightarrow \text{literalReal}$ $E6 \rightarrow \text{literalBool}$ $E6 \rightarrow \text{literalCadena}$ $E6 \rightarrow \text{identificador}$ $E6 \rightarrow \text{null}$ $E6 \rightarrow ( E0 )$	$E6 \rightarrow RE6 E6'$ $E6' \rightarrow \text{op6 } E6'$ $E6' \rightarrow \epsilon$ $RE6 \rightarrow \text{literalEntero}$ $RE6 \rightarrow \text{literalReal}$ $RE6 \rightarrow \text{literalBool}$ $RE6 \rightarrow \text{literalCadena}$ $RE6 \rightarrow \text{identificador}$ $RE6 \rightarrow \text{null}$ $RE6 \rightarrow ( E0 )$
$\text{op1} \rightarrow >$ $\text{op1} \rightarrow >=$	

op1 → < op1 → <= op1 → == op1 → != op3 → and E3 op3 → or E4 op4 → * op4 → / op4 → % op5 → - op5 → not op6 → [ E0 ] op6 → . identificador op6 → ^	
---	--

## Fase 3: Desarrollo de constructores de ASTs para Tiny

### 3.1. Especificación de la sintaxis abstracta de Tiny mediante la enumeración de las signatures (cabeceras) de las funciones constructoras de ASTs.

No terminal	Género
programa	Prog
bloque	Bloq
declaraciones_opt	DecsOpt
declaraciones	LDecs
instrucciones_opt	InsOpt
instrucciones	LIns
instruccion	Ins
declaracion	Dec
tipo	T

parametros_formales_opt	PFormOpt
parametros_formales	LPForm
parametro_formal	PForm
campos	LCamp
campo	Camp
parametros_reales_opt	PRealOpt
parametros_reales	LPReal
E0,E1,E2,E3,E4,E5,E6	Exp

GIC	Constructora de sintaxis abstracta
programa $\rightarrow$ bloque EOF	prog: Bloq x EOF $\rightarrow$ Prog
bloque $\rightarrow$ { declaraciones_opt instrucciones_opt }	bloq: DecsOpt x InsOpt $\rightarrow$ Bloq
declaraciones_opt $\rightarrow$ declaraciones && declaraciones_opt $\rightarrow \epsilon$	si_decs: LDecs $\rightarrow$ DecsOpt no_decs: $\rightarrow$ DecsOpt
instrucciones_opt $\rightarrow$ instrucciones instrucciones_opt $\rightarrow \epsilon$	si_ins: LIns $\rightarrow$ InsOpt no_ins: $\rightarrow$ InsOpt
declaraciones $\rightarrow$ declaraciones ; declaracion declaraciones $\rightarrow$ declaracion	muchas_decs: LDecs x Dec $\rightarrow$ LDecs una_dec: Dec $\rightarrow$ LDecs
declaracion $\rightarrow$ tipo identificador declaracion $\rightarrow$ type tipo identificador declaracion $\rightarrow$ proc identificador ( parametros_formales_opt ) bloque	dec_var: T x Iden $\rightarrow$ Dec dec_tipo: T x Iden $\rightarrow$ Dec dec_proc: Iden x PFormOpt x Bloq $\rightarrow$ Dec
parametros_formales_opt $\rightarrow$ parametros_formales parametros_formales_opt $\rightarrow \epsilon$	si_pform: LPForm $\rightarrow$ PFormOpt no_pform: $\rightarrow$ PFormOpt
parametros_formales $\rightarrow$ parametros_formales , parametro_formal parametros_formales $\rightarrow$ parametro_formal	muchas_pforms: LPForm x PForm $\rightarrow$ LPForm una_pform: PForm $\rightarrow$ LPForm
parametro_formal $\rightarrow$ tipo and_opt identificador and_opt $\rightarrow$ & and_opt $\rightarrow \epsilon$	pform_ref: $\rightarrow$ T x String $\rightarrow$ PForm pform_no_ref: $\rightarrow$ T x String $\rightarrow$ PForm

tipo → tipo [ literalEntero ] tipo → tipo1 tipo1 → ^ tipo1 tipo1 → tipo2 tipo2 → identificador tipo2 → struct { campos } tipo2 → int tipo2 → real tipo2 → bool tipo2 → string	array: T x literalEntero → T puntero: T → T iden: string → T struct: LCamp → T lit_ent: → T lit_real: → T lit_bool: → T lit_string: → T
campos → campos , campo campos → campo	muchos_camp: LCamp x Camp → LCamp un_camp: Camp → LCamp
campo → tipo identificador	camp: T x String → Camp
instrucciones → instrucciones ; instruccion instrucciones → instruccion	muchas_ins: LIns x Ins → LIns una_ins: Ins → LIns
instruccion → @ E0 instruccion → if E0 bloque instruccion → if E0 bloque else bloque instruccion → while E0 bloque instruccion → read E0 instruccion → write E0 instruccion → nl instruccion → new E0 instruccion → delete E0 instruccion → call identificador ( parametros_reales_opt ) instruccion → bloque	ins_asig: Exp → Ins ins_if: Exp x Bloq → Ins ins_if_else: Exp x Bloq x Bloq → Ins ins_while: Exp x Bloq → Ins ins_read: Exp → Ins ins_write: Exp → Ins ins_nl: → Ins ins_new: Exp → Ins ins_delete: Exp → Ins ins_call: string x PRealOpt → Ins ins_bloque: Bloq → Ins
parametros_reales_opt -> parametros_reales parametros_reales_opt → ε	si_preal: LPReal → PRealOpt no_preal: → PRealOpt
parametros_reales -> parametros_reales , E0 parametros_reales -> E0	muchos_preal: LPReal x Exp → LPReal un_preal: Exp → LPReal



<p> <math>E0 \rightarrow E1 = E0</math>  <math>E0 \rightarrow E1</math>  <math>E1 \rightarrow E1 \text{ op1 } E2</math>  <math>E1 \rightarrow E2</math>  <math>E2 \rightarrow E2 + E3</math>  <math>E2 \rightarrow E3 - E3</math>  <math>E2 \rightarrow E3</math>  <math>E3 \rightarrow E4 \text{ op3 } E5</math>  <math>E3 \rightarrow E4</math>  <math>E4 \rightarrow E4 \text{ op4 } E5</math>  <math>E4 \rightarrow E5</math>  <math>E5 \rightarrow \text{op5 } E5</math>  <math>E5 \rightarrow E6</math>  <math>E6 \rightarrow E6 \text{ op6 } E7</math>  <math>E6 \rightarrow E7</math>  <math>E7 \rightarrow \text{literalEntero}</math>  <math>E7 \rightarrow \text{literalReal}</math>  <math>E7 \rightarrow \text{literalBool}</math>  <math>E7 \rightarrow \text{literalCadena}</math>  <math>E7 \rightarrow \text{identificador}</math>  <math>E7 \rightarrow \text{null}</math>  <math>E7 \rightarrow ( E0 )</math>  <math>\text{op1} \rightarrow &gt;</math>  <math>\text{op1} \rightarrow &gt;=</math>  <math>\text{op1} \rightarrow &lt;</math>  <math>\text{op1} \rightarrow &lt;=</math>  <math>\text{op1} \rightarrow ==</math>  <math>\text{op1} \rightarrow !=</math>  <math>\text{op3} \rightarrow \text{and } E3</math>  <math>\text{op3} \rightarrow \text{or } E4</math>  <math>\text{op4} \rightarrow *</math>  <math>\text{op4} \rightarrow /</math>  <math>\text{op4} \rightarrow \%</math>  <math>\text{op5} \rightarrow -</math>  <math>\text{op5} \rightarrow \text{not}</math>  <math>\text{op6} \rightarrow [ E0 ]</math>  <math>\text{op6} \rightarrow \text{.identificador}</math>  <math>\text{op6} \rightarrow ^</math> </p>	<p> <math>\text{exp\_bin: Expbin} \rightarrow \text{Exp}</math>  <math>\text{exp\_un: Expun} \rightarrow \text{Exp}</math>  <math>\text{asig: Exp x Exp} \rightarrow \text{Expbin}</math>  <math>\text{mayor: Exp x Exp} \rightarrow \text{Expbin}</math>  <math>\text{menor: Exp x Exp} \rightarrow \text{Expbin}</math>  <math>\text{mayorIgual: Exp x Exp} \rightarrow \text{Expbin}</math>  <math>\text{menorIgual: Exp x Exp} \rightarrow \text{Expbin}</math>  <math>\text{igual: Exp x Exp} \rightarrow \text{Expbin}</math>  <math>\text{desigual: Exp x Exp} \rightarrow \text{Expbin}</math>  <math>\text{suma: Exp x Exp} \rightarrow \text{Expbin}</math>  <math>\text{resta: Exp x Exp} \rightarrow \text{Expbin}</math>  <math>\text{and: Exp x Exp} \rightarrow \text{Expbin}</math>  <math>\text{or: Exp x Exp} \rightarrow \text{Expbin}</math>  <math>\text{mul: Exp x Exp} \rightarrow \text{Expbin}</math>  <math>\text{div: Exp x Exp} \rightarrow \text{Expbin}</math>  <math>\text{mod: Exp x Exp} \rightarrow \text{Expbin}</math>  <math>\text{neg: Exp} \rightarrow \text{Expun}</math>  <math>\text{not: Exp} \rightarrow \text{Expun}</math>  <math>\text{acceso\_array: Exp x Exp} \rightarrow \text{Exp}</math>  <math>\text{acceso\_campo: Exp x String} \rightarrow \text{Exp}</math>  <math>\text{acceso\_puntero: Exp} \rightarrow \text{Exp}</math>  <math>\text{exp\_litEntero: N} \rightarrow \text{Exp}</math>  <math>\text{exp\_litReal: R} \rightarrow \text{Exp}</math>  <math>\text{exp\_litBoolTrue:} \rightarrow \text{Exp}</math>  <math>\text{exp\_litBoolFalse:} \rightarrow \text{Exp}</math>  <math>\text{exp\_litCadena: string} \rightarrow \text{Exp}</math>  <math>\text{exp\_identificador: string} \rightarrow \text{Exp}</math>  <math>\text{exp\_null:} \rightarrow \text{Exp}</math> </p>
---	---

### 3.2. Especificación del constructor de ASTs mediante una gramática s-atribuida.

**programa**  $\rightarrow$  **bloque EOF**  
programa.a = prog(bloque.a)

**bloque**  $\rightarrow$  { **declaraciones\_opt instrucciones\_opt** }  
bloque.a = bloq(declaraciones\_opt.a, instrucciones\_opt.a)

**declaraciones\_opt**  $\rightarrow$  **declaraciones &&**  
declaraciones\_opt.a = si\_decs(declaraciones.a)

**declaraciones\_opt**  $\rightarrow \epsilon$   
declaraciones\_opt.a = no\_decs()

**instrucciones\_opt**  $\rightarrow$  **instrucciones**  
instrucciones\_opt.a = si\_ins(instrucciones.a)

**instrucciones\_opt**  $\rightarrow \epsilon$   
instrucciones\_opt.a = no\_ins()

**declaraciones**  $\rightarrow$  **declaraciones ; declaracion**  
declaraciones0.a = muchas\_decs(declaraciones1.a, declaracion.a)

**declaraciones**  $\rightarrow$  **declaracion**  
declaraciones.a = una\_dec(declaracion.a)

**declaracion**  $\rightarrow$  **tipo identificador**  
declaracion.a = dec\_var(tipo.a, identificador.lex)

**declaracion**  $\rightarrow$  **type tipo identificador**  
declaracion.a = dec\_tipo(tipo.a, identificador.lex)

**declaracion**  $\rightarrow$  **proc identificador ( parametros\_formales\_opt ) bloque**  
declaracion.a = dec\_proc(identificador.lex, parametros\_formales\_opt.a, bloque.a)

**parametros\_formales\_opt**  $\rightarrow$  **parametros\_formales**  
parametros\_formales\_opt.a = si\_pform(parametros\_formales.a)

**parametros\_formales\_opt**  $\rightarrow \epsilon$   
parametros\_formales\_opt.a = no\_pform()

**parametros\_formales** → **parametros\_formales** , **parametro\_formal**

parametros\_formales0.a = muchas\_pforms(parametros\_formales1.a,  
parametro\_formal.a)

**parametros\_formales** → **parametro\_formal**

parametros\_formales.a = una\_pform(parametro\_formal.a)

**parametro\_formal** → **tipo and\_opt identificador**

parametro\_formal.a = pform(tipo.a, and\_opt.a, identificador.lex)

**and\_opt** → **&**

and\_opt.a = "ref"

**and\_opt** → **ε**

and\_opt.a = "no\_ref"

**tipo** → **tipo [ literalEntero ]**

tipo0.a = array(tipo1.a, literalEntero.lex)

**tipo** → **tipo1**

tipo.a = tipo1.a

**tipo1** → **^ tipo1**

tipo10.a = puntero(tipo11.a)

**tipo1** → **tipo2**

tipo1.a = tipo2.a

**tipo2** → **identificador**

tipo2.a = iden(identificador.lex)

**tipo2** → **struct { campos }**

tipo2.a = struct(campos.a)

**tipo2** → **int**

tipo2.a = lit\_ent()

**tipo2** → **real**

tipo2.a = lit\_real()

**tipo2** → **bool**

tipo2.a = lit\_bool()

**tipo2** → **string**

tipo2.a = lit\_string()

**campos** → **campos , campo**

campos0.a = muchos\_camp(campos1.a, campo.a)

**campos** → **campo**

campos.a = un\_camp(campo.a)

**campo** → **tipo identificador**

campo.a = camp(tipo.a, identificador.lex)

**instrucciones** → **instrucciones ; instruccion**

instrucciones0.a = muchas\_ins(instrucciones1.a, instruccion.a)

**instrucciones** → **instruccion**

instrucciones.a = una\_ins(instruccion.a)

**instruccion** → **@ E0**

instruccion.a = ins\_asig(E0.a)

**instruccion** → **if E0 bloque**

instruccion.a = ins\_if(E0.a, bloque.a)

**instruccion** → **if E0 bloque else bloque**

instruccion.a = ins\_if\_else(E0.a, bloque0.a, bloque1.a)

**instruccion** → **while E0 bloque**

instruccion.a = ins\_while(E0.a, bloque.a)

**instruccion** → **read E0**

instruccion.a = ins\_read(E0.a)

**instruccion** → **write E0**

instruccion.a = ins\_write(E0.a)

**instruccion** → **nl**

instruccion.a = ins\_nl()

**instruccion** → **new E0**

instruccion.a = ins\_new(E0.a)

**instruccion** → **delete E0**

instruccion.a = ins\_delete(E0.a)

**instruccion** → **call identificador ( parametros\_reales\_opt )**

instruccion.a = ins\_call(identificador.lex, parametros\_reales\_opt.a)

**instruccion** → **bloque**

instruccion.a = ins\_bloque(bloque.a)

**parametros\_reales\_opt** → **parametros\_reales**

parametros\_reales\_opt.a = si\_preal(parametros\_reales.a)

**parametros\_reales\_opt** →  $\epsilon$

parametros\_reales\_opt.a = no\_preal()

**parametros\_reales** → **parametros\_reales** , **E0**

parametros\_reales0.a = muchos\_preal(parametros\_reales1.a, E0.a)

**parametros\_reales** → **E0**

parametros\_reales.a = un\_preal(E0.a)

**E0** → **E1 = E0**

E00.a = asig(E1.a, E01.a)

**E0** → **E1**

E0.a = E1.a

**E1** → **E1 op1 E2**

E10.a = mkop2(op1.op, E11.a, E2.a)

**E1** → **E2**

E1.a = E2.a

**E2** → **E2 + E3**

E20.a = mkop2("+", E21.a, E3.a)

**E2** → **E3 - E3**

E2.a =mkop2("-", E30.a, E31.a)

**E2** → **E3**

E2.a = E3.a

**E3** → **E4 and E3**

E30.a = and(E4.a, E31.a)

**E3** → **E4 or E4**

E3.a = or(E40.a, E41.a)

**E3** → **E4**

E3.a = E4.a

**E4 → E4 op4 E5**

E40.a = mkop2(op4.op, E41, E5)

**E4 → E5**

E4.a = E5.a

**E5 → op5 E5**

E50.a = mkop1(op5.op, E51.a)

**E5 → E6**

E5.a = E6.a

**E6 → E6 [ E0 ]**

E60.a = acceso\_array(E61.a, E0.a)

**E6 → E6 . identificador**

E60.a = acceso\_campo(E61.a, identificador.lex)

**E6 → E6 ^**

E60.a = acceso\_puntero(E61.a)

**E6 → E7**

E6.a = E7.a

**E7 → literalEntero**

E7.a = exp\_litEntero(literalEntero.lex)

**E7 → literalReal**

E7.a = exp\_litReal(literalEntero.lex)

**E7 → literalTrue**

E7.a = exp\_litBoolTrue()

**E7 → literalFalse**

E7.a = exp\_litBoolFalse()

**E7 → literalCadena**

E7.a = exp\_litCadena(literalCadena.lex)

**E7 → identificador**

E7.a = exp\_litIdentificador(identificador.lex)

**E7 → null**

E7.a = exp\_null()

**E7** → ( E0 )

E7.a = E0.a

**op1** → >

op1.op = ">"

**op1** → >=

op1.op = ">="

**op1** → <

op1.op = "<"

**op1** → <=

op1.op = "<="

**op1** → ==

op1.op = "=="

**op1** → !=

op1.op = "!="

**op4** → \*

op4.op = "\*"

**op4** → /

op4.op = "/"

**op4** → %

op4.op = "%"

**op5** → -

op5.op = "-"

**op5** → not

op5.op = "not"

```

fun mkop2(op, opnd1, opnd2):
  op = "+" → return suma(opnd1, opnd2)
  op = "-" → return resta(opnd1, opnd2)
  op = "*" → return mul(opnd1, opnd2)
  op = "/" → return div(opnd1, opnd2)
  op = "%" → return mod(opnd1, opnd2)
  op = "=" → return asig(opnd1, opnd2)
  op = ">" → return mayor(opnd1, opnd2)
  op = "<" → return menor(opnd1, opnd2)
  op = ">=" → return mayorIgual(opnd1, opnd2)
  op = "<=" → return menorIgual(opnd1, opnd2)
  op = "==" → return igual(opnd1, opnd2)
  op = "!=" → return desigual(opnd1, opnd2)

```

```

fun mkop1(op, opnd1):
  op = "-" → return neg(opnd1)
  op = "not" → return not(opnd1)

```

```

fun pform(tipo, and_opt, id):
  and_opt = "ref" → return pform_ref(tipo, id)
  and_opt = "no_ref" → return pform_no_ref(tipo, id)

```

### 3.3. Acondicionamiento de dicha especificación para permitir la implementación descendente.

Gramática s-atribuida	Acondicionado
<b>programa</b> → <b>bloque EOF</b> programa.a = prog(bloque.a)	
<b>bloque</b> → { <b>declaraciones_opt</b> <b>instrucciones_opt</b> } bloque.a = bloq(declaraciones_opt.a, instrucciones_opt.a)	
<b>declaraciones_opt</b> → <b>declaraciones &amp;&amp;</b> declaraciones_opt.a = si_decs(declaraciones.a) <b>declaraciones_opt</b> → <b>ε</b> declaraciones_opt.a = no_decs()	
<b>instrucciones_opt</b> → <b>instrucciones</b>	



instrucciones_opt.a = si_ins(instrucciones.a) <b>instrucciones_opt</b> → $\epsilon$ instrucciones_opt.a = no_ins()	
<b>declaraciones</b> → <b>declaraciones ; declaracion</b> declaraciones0.a = muchas_decs(declaraciones1.a, declaracion.a) <b>declaraciones</b> → <b>declaracion</b> declaraciones.a = una_dec(declaracion.a)	<b>declaraciones</b> → <b>declaracion declaraciones'</b> declaraciones'.h = una_dec(declaracion.a) declaraciones.a = declaraciones'.a <b>declaraciones'</b> → <b>; declaracion declaraciones'</b> declaraciones'1.h = muchas_decs(declaraciones'0.h, declaracion.a) declaraciones'0.a = declaraciones'1.a <b>declaraciones'</b> → $\epsilon$ declaraciones'.a = declaraciones'.h
<b>declaracion</b> → <b>tipo identificador</b> declaracion.a = dec_var(tipo.a, identificador.lex) <b>declaracion</b> → <b>type tipo identificador</b> declaracion.a = dec_tipo(tipo.a, identificador.lex) <b>declaracion</b> → <b>proc identificador</b> <b>(parametros_formales_opt ) bloque</b> declaracion.a = dec_proc(identificador.lex, parametros_formales_opt.a, bloque.a)	
<b>parametros_formales_opt</b> → <b>parametros_formales</b> parametros_formales_opt.a = si_pform(parametros_formales.a) <b>parametros_formales_opt</b> → $\epsilon$ parametros_formales_opt.a = no_pform()	
<b>parametros_formales</b> → <b>parametros_formales , parametro_formal</b> parametros_formales0.a = muchas_pforms(parametros_formales1.a, parametro_formal.a) <b>parametros_formales</b> → <b>parametro_formal</b> parametros_formales.a = una_pform(parametro_formal.a)	<b>parametros_formales</b> → <b>parametro_formal parametros_formales'</b> parametros_formales'.h = una_pform(parametro_formal.a) parametros_formales.a = parametros_formales'.a <b>parametros_formales'</b> → <b>, parametro_formal parametros_formales'</b> parametros_formales'1.h = muchas_pforms(parametros_formales'0.h, parametro_formal.a) parametros_formales'0.a = parametros_formales'1.a <b>parametros_formales'</b> → $\epsilon$ parametros_formales'.a = parametros_formales'.h

<b>parametro_formal</b> → <b>tipo</b> <b>and_opt</b> <b>identificador</b> parametro_formal.a = pform(tipo.a, and_opt.a, identificador.lex) <b>and_opt</b> → <b>&amp;</b> and_opt.a = pform_ref() <b>and_opt</b> → <b>ε</b> and_opt.a = pform_no_ref()	
<b>tipo</b> → <b>tipo</b> [ <b>literalEntero</b> ] tipo0.a = array(tipo1.a, literalEntero.lex) <b>tipo</b> → <b>tipo1</b> tipo.a = tipo1.a	<b>tipo</b> → <b>tipo1</b> <b>tipo'</b> tipo'.h = tipo1.a tipo.a = tipo'.a <b>tipo'</b> → [ <b>literalEntero</b> ] <b>tipo'</b> tipo'1.h = array(tipo'0.h, literalEntero.lex) tipo'0.a = tipo'1.a <b>tipo'</b> → <b>ε</b> tipo'.a = tipo'.h
<b>tipo1</b> → <b>^ tipo1</b> tipo10.a = puntero(tipo11.a) <b>tipo1</b> → <b>tipo2</b> tipo1.a = tipo2.a	
<b>tipo2</b> → <b>identificador</b> tipo2.a = iden(identificador.lex) <b>tipo2</b> → <b>struct { campos }</b> tipo2.a = struct(campos.a) <b>tipo2</b> → <b>int</b> tipo2.a = lit_ent() <b>tipo2</b> → <b>real</b> tipo2.a = lit_real() <b>tipo2</b> → <b>bool</b> tipo2.a = lit_bool() <b>tipo2</b> → <b>string</b> tipo2.a = lit_string()	
<b>campos</b> → <b>campos , campo</b> campos0.a = muchos_camp(campos1.a, campo.a) <b>campos</b> → <b>campo</b> campos.a = un_camp(campo.a)	<b>campos</b> → <b>campo</b> <b>campos'</b> campos'.h = un_camp(campo.a) campos.a = campos'.a <b>campos'</b> → <b>, campo</b> <b>campos'</b> campos'1.h = muchos_camp(campos'0.h, campo.a) campos'0.a = campos'1.a <b>campos'</b> → <b>ε</b> campos'.a = campos'.h

<b>campo</b> → <b>tipo identificador</b> campo.a = camp(tipo.a, identificador.lex)	
<b>instrucciones</b> → <b>instrucciones ; instruccion</b> instrucciones0.a = muchas_ins(instrucciones1.a, instruccion.a) <b>instrucciones</b> → <b>instruccion</b> instrucciones.a = una_ins(instruccion.a)	<b>instrucciones</b> → <b>instruccion instrucciones'</b> instrucciones'.h = una_ins(instruccion.a) instrucciones.a = instrucciones'.a <b>instrucciones'</b> → <b>; instrucción instrucciones'</b> instrucciones'1.h = muchas_ins(instrucciones'0.h, instrucción.a) instrucciones'0.a = instrucciones'1.a <b>instrucciones'</b> → $\epsilon$ instrucciones'.a = instrucciones'.h
<b>instruccion</b> → <b>@ E0</b> instruccion.a = ins_asig(E0.a) <b>instruccion</b> → <b>if E0 bloque</b> instruccion.a = ins_if(E0.a, bloque.a) <b>instruccion</b> → <b>if E0 bloque else bloque</b> instruccion.a = ins_if_else(E0.a, bloque0.a, bloque1.a) <b>instruccion</b> → <b>while E0 bloque</b> instruccion.a = ins_while(E0.a, bloque.a) <b>instruccion</b> → <b>read E0</b> instruccion.a = ins_read(E0.a) <b>instruccion</b> → <b>write E0</b> instruccion.a = ins_write(E0.a) <b>instruccion</b> → <b>nl</b> instruccion.a = ins_nl() <b>instruccion</b> → <b>new E0</b> instruccion.a = ins_new(E0.a) <b>instruccion</b> → <b>delete E0</b> instruccion.a = ins_delete(E0.a) <b>instruccion</b> → <b>call identificador ( parametros_reales_opt )</b> instruccion.a = ins_call(identificador.lex, parametros_reales_opt.a) <b>instruccion</b> → <b>bloque</b> instruccion.a = ins_bloque(bloque.a)	<b>instruccion</b> → <b>@ E0</b> instruccion.a = ins_asig(E0.a) <b>instruccion</b> → <b>if E0 bloque restoif</b> restoif.h = bloque.a restoif.ah = E0.a instruccion.a = restoif.a <b>restoif</b> → $\epsilon$ restoif.a = ins_if(restoif.ah, restoif.h) <b>restoif</b> → <b>else bloque</b> restoif.a = ins_if_else(restoif.ah, restoif.h bloque.a) <b>instruccion</b> → <b>while E0 bloque</b> instruccion.a = ins_while(E0.a, bloque.a) <b>instruccion</b> → <b>read E0</b> instruccion.a = ins_read(E0.a) <b>instruccion</b> → <b>write E0</b> instruccion.a = ins_write(E0.a) <b>instruccion</b> → <b>nl</b> instruccion.a = ins_nl() <b>instruccion</b> → <b>new E0</b> instruccion.a = ins_new(E0.a) <b>instruccion</b> → <b>delete E0</b> instruccion.a = ins_delete(E0.a) <b>instruccion</b> → <b>call identificador ( parametros_reales_opt )</b> instruccion.a = ins_call(identificador.lex, parametros_reales_opt.a) <b>instruccion</b> → <b>bloque</b> instruccion.a = ins_bloque(bloque.a)

<b>parametros_reales_opt</b> → <b>parametros_reales</b> parametros_reales_opt.a = si_preal(parametros_reales.a) <b>parametros_reales_opt</b> → ε parametros_reales_opt.a = no_preal()	
<b>parametros_reales</b> → <b>parametros_reales</b> , <b>E0</b> parametros_reales0.a = muchos_preal(parametros_reales1.a, E0.a) <b>parametros_reales</b> → <b>E0</b> parametros_reales.a = un_preal(E0.a)	<b>parametros_reales</b> → <b>E0 parametros_reales'</b> parametros_reales'.h = un_preal(E0.a) parametros_reales.a = parametros_reales'.a <b>parametros_reales'</b> → , <b>E0 parametros_reales'</b> parametros_reales'1.h = muchos_preal(parametros_reales'0.h, E0.a) parametros_reales'0.a = parametros_reales'1.a <b>parametros_reales'</b> → ε parametros_reales'.a = parametros_reales'.h
<b>E0</b> → <b>E1 = E0</b> E00.a = asig(E1.a, E01.a) <b>E0</b> → <b>E1</b> E0.a = E1.a	<b>E0</b> → <b>E1 asignacion</b> asignacion.h = E1.a E0.a = asignacion.a <b>asignacion</b> → = <b>E0</b> asignacion.a = asig(asignacion.h, E0.a) <b>asignacion</b> → ε asignacion.a = asignacion.h
<b>E1</b> → <b>E1 op1 E2</b> E10.a = mkop2(op1.op, E11.a, E2.a) <b>E1</b> → <b>E2</b> E1.a = E2.a	<b>E1</b> → <b>E2 E1'</b> E1'.h = E2.a E1.a = E1'.a <b>E1'</b> → <b>op1 E2 E1'</b> E1'1.h = mkop2(E1'0.h , op1.op, E2.a) E1'0.a = E1'1.a <b>E1'</b> → ε E1'.a = E1'.h

<b>E2 → E2 + E3</b> E20.a = mkop2("+", E21.a, E3.a) <b>E2 → E3 - E3</b> E2.a =mkop2("-", E30.a, E31.a) <b>E2 → E3</b> E2.a = E3.a	<b>E2 → E3 minus E2'</b> minus.h = E3.a E2'.h = minus.a E2.a = E2'.a <b>E2' → + E3 E2'</b> E2'1.h = mkop2(E2'1.h, "+", E3.a E2'0.a = E2'1.a <b>E2' → ε</b> E2'.a = E2'.h <b>minus → - E3</b> minus.a = mkop2(minus.h, "-", E3.a) <b>minus → ε</b> minus.a = minus.h
<b>E3 → E4 and E3</b> E30.a = and(E4.a, E31.a) <b>E3 → E4 or E4</b> E3.a = or(E40.a, E41.a) <b>E3 → E4</b> E3.a = E4.a	<b>E3 → E4 and_or</b> and_or.h = E4.a E3.a = and_or.a <b>and_or → and E3</b> and_or.a = and(and_or.h, E3.a) <b>and_or → or E4</b> and_or.a = or(and_or.h, E4.a) <b>and_or → ε</b> and_or.a = and_or.h
<b>E4 → E4 op4 E5</b> E40.a = mkop2(op4.op, E41.a, E5.a) <b>E4 → E5</b> E4.a = E5.a	<b>E4 → E5 E4'</b> E4'.h = E5.a E4.a = E4'.a <b>E4' → op4 E5 E4'</b> E4'1.h = mkop2(E4'0.h, op4.op, E5.a) E4'0.a = E4'1.a <b>E4' → ε</b> E4'.a = E4'.h
<b>E5 → op5 E5</b> E50.a = mkop1(op5.op, E51.a) <b>E5 → E6</b> E5.a = E6.a	
<b>E6 → E6 [ E0 ]</b> E60.a = acceso_array(E61.a, E0.a) <b>E6 → E6 . identificador</b> E60.a = acceso_campo(E61.a, identificador.lex) <b>E6 → E6 ^</b> E60.a = acceso_puntero(E61.a)	<b>E6 → E7 E6'</b> E6'.h = E7.a E6.a = E6'.a <b>E6' → resto6 E6'</b> E6'1.h = resto6.a

<b>E6 → E7</b> E6.a = E7.a	E6'0.a = E6'1.a <b>E6' → ε</b> E6'.a = E6'.h <b>resto6 → [ E0 ]</b> resto6.a = acceso_array(resto6.h, E0.a) <b>resto6 → . identificador</b> resto6.a = acceso_campo(resto6.h, identificador.lex) <b>resto6 → ^</b> resto6.a = acceso_puntero(resto6.h)
<b>E7 → literalEntero</b> E7.a = exp_litEntero(literalEntero.lex) <b>E7 → literalReal</b> E7.a = exp_litReal(literalReal.lex) <b>E7 → literalTrue</b> E7.a = exp_litBoolTrue() <b>E7 → literalFalse</b> E7.a = exp_litBoolFalse() <b>E7 → literalCadena</b> E7.a = exp_litCadena(literalCadena.lex) <b>E7 → identificador</b> E7.a = exp_litIdentificador(identificador.lex) <b>E7 → null</b> E7.a = exp_null() <b>E7 → ( E0 )</b> E7.a = E0.a	
<b>op1 → &gt;</b> op1.op = ">" <b>op1 → &gt;=</b> op1.op = ">=" <b>op1 → &lt;</b> op1.op = "<" <b>op1 → &lt;=</b> op1.op = "<=" <b>op1 → ==</b> op1.op = "==" <b>op1 → !=</b> op1.op = "!="	

<b>op4</b> → * op4.op = "*" <b>op4</b> → / op4.op = "/" <b>op4</b> → % op4.op = "%"	
<b>op5</b> → - op5.op = "-" <b>op5</b> → not op5.op = "not"	

### 3.4. Especificación de un procesamiento que imprima los tokens del programa leído

Procesamiento del lenguaje Tiny orientado a realizar una impresión bonita del programa. En concreto que:

- Las expresiones deben imprimirse con el mínimo número necesario de paréntesis
- Un token leído por cada línea
- Las palabras reservadas se escriben en minúsculas y con < >
- Las palabras reservadas son: int real bool string and or not null true false proc if else while struct new delete read write nl type call
- El fin de fichero se escribirá como <EOF>

```
imprimeOpnd(Opnd,MinPrior):
    if prioridad(Opnd) < MinPrior
        print "("
        imprime(Opnd)
    if prioridad(Opnd) < MinPrior
        print ")"
```

```
prioridad(asig(_,_)): return 0
prioridad(mayor(_,_)): return 1
prioridad(menor(_,_)): return 1
prioridad(mayorIgual(_,_)): return 1
prioridad(menorIgual(_,_)): return 1
prioridad(igual(_,_)): return 1
prioridad(desigual(_,_)): return 1
prioridad(suma(_,_)): return 2
prioridad(resta(_,_)): return 2
```

```
prioridad(and(_, _)): return 3
prioridad(or(_, _)): return 3
prioridad(mul(_, _)): return 4
prioridad(div(_, _)): return 4
prioridad(mod(_, _)): return 4
prioridad(neg(_)): return 5
prioridad(not(_)): return 5
prioridad(acceso_array(_, _)): return 6
prioridad(acceso_campo(_, _)): return 6
prioridad(acceso_puntero(_, _)): return 6
prioridad(exp_litEntero(_)): return 7
prioridad(exp_litReal(_)): return 7
prioridad(exp_litBoolTrue(_)): return 7
prioridad(exp_litBoolFalse(_)): return 7
prioridad(exp_litCadena(_)): return 7
prioridad(exp_identificador(_)): return 7
prioridad(exp_null(_)): return 7
```

```
imprime(prog(Bloq)):
    imprime(Bloq)
    print "<EOF>"
```

```
imprime(bloq(Decsopt, Insopt)):
    print "{"
    imprime(Decsopt)
    imprime(Insopt)
    print "}"
```

```
imprime(si_decs(LDecs)):
    imprime(LDecs)
    print "&&"
```

```
imprime(no_decs()):
    skip
```

```
imprime(muchas_decs(LDecs, Dec)):
    imprime(LDecs)
    print ";"
    imprime(Dec)
```

```
imprime(una_dec(Dec)):
    imprime(Dec)
```

```
imprime(dec_var(T, id)):
    imprime(T)
    imprime(iden(id))
```



```
imprime(dec_tipo(T, id)):
    print "<type>"
    imprime(T)
    imprime(iden(id))
```

```
imprime(dec_proc(id, PFormOpt, Bloq)):
    print "<proc>"
    imprime(iden(id))
    print "("
    imprime(PFormOpt)
    print ")"
    imprime(Bloq)
```

```
imprime(si_pform(LPForm)):
    imprime(LPForm)
```

```
imprime(no_pform()):
    skip
```

```
imprime(muchas_pforms(LPForm, PForm)):
    imprime(LPForm)
    print ","
    imprime(PForm)
```

```
imprime(una_pform(PForm)):
    imprime(PForm)
```

```
imprime(pform_ref(T, lden)):
    imprime(T)
    print "&"
    imprime(iden)
```

```
imprime(pform_no_ref(T, lden)):
    imprime(T)
    imprime(iden)
```

```
imprime(array(T, dim)):
    imprime(T)
    print "["
    imprime(dim)
    print "]"
```

```
imprime(puntero(T)):
    imprime(T)
    print "^"
```

```
imprime(iden(string)):
    print string
```

```
imprime(struct(LCamp)):
    print <struct>
    print "{"
    imprime(LCamp)
    print "}"
```

```
imprime(lit_ent(string)):
    print <int>
```

```
imprime(lit_real(string)):
    print <real>
```

```
imprime(lit_bool(string)):
    print<bool>
```

```
imprime(lit_string(string)):
    print<string>
```

```
imprime(muchos_camp(LCamp, Camp)):
    imprime(LCamp)
    print ","
    imprime(Camp)
```

```
imprime(un_camp(Camp)):
    imprime(Camp)
```

```
imprime(camp(T, lden)):
    imprime(T)
    imprime(liden)
```

```
imprime(Si_ins(LIns)):
    imprime(LIns)
```

```
imprime(no_ins()):
    skip
```

```
imprime(muchas_ins(LIns, Ins)):
    imprime(LIns)
    print ","
    imprime(Ins)
```

```
imprime(una_ins(Ins)):
    imprime(Ins)
```

```
imprime(ins_asig(Exp)):
    print "@"
    imprime(Exp)
    print ";
```

```
imprime(ins_if(Exp, Bloq)):
    print "<if>"
    imprime(Exp)
    imprime(Bloq)
```

```
imprime(ins_if_else(Exp, Bloq, Bloq)):
    print "<if>"
    imprime(Exp)
    imprime(Bloq)
    print "<else>"
    imprime(Bloq)
```

```
imprime(ins_while(Exp, Bloq)):
    print "<while>"
    imprime(Exp)
    imprime(Bloq)
```

```
imprime(ins_read(Exp)):
    print "<read>"
    imprime(Exp)
```

```
imprime(ins_write(Exp)):
    print "<write>"
    imprime(Exp)
```

```
imprime(ins_nl()):
    print "<nl>"
```

```
imprime(ins_new(Exp)):
    print "<new>"
    imprime(Exp)
```

```
imprime(ins_delete(Exp)):
    print "<delete>"
    imprime(Exp)
```

```
imprime(ins_call(Iden, PRealOpt)):
    print "<call>"
    imprime(Iden)
    print "("
    imprime(PRealOpt)
    print ")"
```

```
imprime(ins_bloque(Bloq)):
    imprime(Bloq)
```

```
imprime(si_preal(LPReal)):
    imprime(LPReal)
```

```
imprime(no_preal()):
    skip
```

```
imprime(muchos_preal(LPReal, Exp)):
    imprime(LPReal)
    print ","
    imprime(Exp)
```

```
imprime(un_preal(Exp)):
    imprime(Exp)
```

```
imprimeExpBin(Exp0, Exp1, string, p0, p1):
    imprimeOpnd(Exp0, p0)
    print string
    imprimeOpnd(Exp1, p1)
```

```
imprime(asig(Exp0, Exp1)):
    imprimeExpBin(Exp0, Exp1, "=", 1, 0)
```

```
imprime(mayor(Exp0, Exp1)):
    imprimeExpBin(Exp0, Exp1, ">", 1, 2)
```

```
imprime(menor(Exp0, Exp1)):
    imprimeExpBin(Exp0, Exp1, "<", 1, 2)
```

```
imprime(mayorIgual(Exp0, Exp1)):
    imprimeExpBin(Exp0, Exp1, ">=", 1, 2)
```

```
imprime(menorIgual(Exp0, Exp1)):
    imprimeExpBin(Exp0, Exp1, "<=", 1, 2)
```

```

imprime(igual(Exp0, Exp1)):
    imprimeExpBin(Exp0, Exp1, "==", 1, 2)

imprime(desigual(Exp0, Exp1)):
    imprimeExpBin(Exp0, Exp1, "!=", 1, 2)

imprime(suma(Exp0, Exp1)):
    imprimeExpBin(Exp0, Exp1, "+", 2, 3)

imprime(resta(Exp0, Exp1)):
    imprimeExpBin(Exp0, Exp1, "-", 3, 3)

imprime(and(Exp0, Exp1)):
    imprimeExpBin(Exp0, Exp1, "<and>", 4, 3)

imprime(or(Exp0, Exp1)):
    imprimeExpBin(Exp0, Exp1, "<or>", 4, 4)

imprime(mul(Exp0, Exp1)):
    imprimeExpBin(Exp0, Exp1, "*", 4, 5)

imprime(div(Exp0, Exp1)):
    imprimeExpBin(Exp0, Exp1, "/", 4, 5)

imprime(mod(Exp0, Exp1)):
    imprimeExpBin(Exp0, Exp1, "%", 4, 5)

imprimeExpUn(Exp, string, p):
    print string
    imprimeOpnd(Exp, p)

imprime(neg(Exp)):
    imprimeExpUn(Exp, "-", 5)

imprime(not(Exp)):
    imprimeExpUn(Exp, "<not>", 5)

imprime(acceso_array(Exp0, Exp1)):
    imprimeOpnd(Exp0, 6)
    print " [ "
    imprime(Exp1)
    print " ] "

```

```
imprime(acceso_campo(Exp, id)):
    imprimeOpnd(Exp, 6)
    print "."
    imprime(iden(id))
```

```
imprime(acceso_puntero(Exp)):
    imprimeOpnd(Exp, 6)
    print "^"
```

```
imprime(exp_litEntero(N)):
    print N
```

```
imprime(exp_litReal(R)):
    print R
```

```
imprime(exp_litCadena(Id)):
    print Id
```

```
imprime(exp_Identificador(Id)):
    print Id
```

```
imprime(exp_litBoolTrue()):
    print "<true>"
```

```
imprime(exp_litBoolFalse()):
    print "<false>"
```

```
imprime(exp_null()):
    print "<null>"
```

## Fase 4: Finalización del procesador para Tiny

### **4.1 Especificación del proceso de vinculación**

#### **4.1.1 Programa**

```
vincula(prog(Bloq)):
    ts = creaTS()
    vincula(Bloq)
```

#### 4.1.2 Declaraciones

```
vincula(Bloq(DecsOpt, InsOpt)):
    abreAmbito(ts)
    recolectaDecs1(DecsOpt)
    recolectaDecs2(DecsOpt)
    vincula1(InsOpt)
    cierraAmbito(ts)
```

```
recolectaDecs1(si_decs(LDecs)):
    recolectaDecs1(LDecs)
```

```
recolectaDecs2(si_decs(LDecs)):
    recolectaDecs2(LDecs)
```

```
recolectaDecs1(no_decs()):
    noop
```

```
recolectaDecs2(no_decs()):
    noop
```

```
recolectaDecs1(muchas_decs(LDecs, Dec)):
    recolectaDecs1(LDecs)
    recolectaDec1(Dec)
```

```
recolectaDecs2(muchas_decs(LDecs, Dec)):
    recolectaDecs2(LDecs)
    recolectaDec2(Dec)
```

```
recolectaDecs1(una_dec(Dec)):
    recolectaDec1(Dec)
```

```
recolectaDecs2(una_dec(Dec)):
    recolectaDec2(Dec)
```

```
recolectaDec1(dec_var(T, id)):
    vincula1(T)
    if contiene(ts,Id) then
        error
    else
        inserta(ts,Id,$)
    end if
```

```
recolectaDec2(dec_var(T, id)):
    vincula2(T)
```

```
recolectaDec1(dec_tipo(T, id)):
    vincula1(T)
    if contiene(ts,Id) then
        error
    else
        inserta(ts,Id,$)
    end if
```

```
recolectaDec2(dec_tipo(T, id)):
    vincula2(T)
```

```
recolectaDec1(dec_proc(id, PFormOpt, Bloq)):
    if contiene(ts,Id) then
        error
    else
        inserta(ts,Id,$)
    end if
    abreAmbito(ts)
    recolectaDecs1(PformOpt)
    recolectaDecs2(PformOpt)
    vincula(bloq)
    cierraAmbito(ts)
```

```
recolectaDec2(dec_proc(id, PFormOpt, Bloq)):
    noop
```

```
recolectaDecs1(si_pform(LPForm)):
    recolectaDecs1(LPForm)
```

```
recolectaDecs2(si_pform(LPForm)):
    recolectaDecs2(LPForm)
```

```
recolectaDecs1(no_pform()):
    noop
```

```
recolectaDecs2(no_pform()):
    noop
```

```
recolectaDecs1(muchas_pforms(LPForm, PForm)):
    recolectaDecs1(LPForm)
    recolectaDec1(PForm)
```

```
recolectaDecs2(muchas_pforms(LPForm, PForm)):
    recolectaDecs2(LPForm)
    recolectaDec2(PForm)
```



```

recolectaDecs1(una_pform(PForm)):
    recolectaDec(PForm)
recolectaDecs2(una_pform(PForm)):
    recolectaDec2(PForm)

```

```

recolectaDec1(pform_ref(T,id)):
    vincula1(T)
    if contiene(ts,Id) then
        error
    else
        inserta(ts,id,$)
    end if

```

```

recolectaDec2(pform_ref(T,id)):
    vincula2(T)

```

```

recolectaDec1(pform_no_ref(T,id)):
    vincula1(T)
    if contiene(ts,Id) then
        error
    else
        inserta(ts,id,$)
    end if

```

```

recolectaDec2(pform_no_ref(T,id)):
    vincula2(T)

```

#### 4.1.3 Tipos

```

vincula1(array(T, dim)):
    vincula1(T)

```

```

vincula2(array(T, dim)):
    vincula2(T)

```

```

vincula1(puntero(T)):
    if T != iden(_):
        vincula1(T)
    end if

```

```

vincula2(puntero(T)):
    if (T = iden(Id)):
        T.vinculo = vinculoDe(ts, Id)
        if (T.vinculo = -|):
            error
        end if
    else
        vincula2(T)
    end if

```

```
vincula1(iden(id)):
  if contiene(ts,id) then
    $.vinculo = vinculoDe(ts,id)
  else
    error
  end if
```

```
vincula2(iden(id)):
  noop
```

```
vincula1(struct(LCamp)):
  vincula1(LCamp)
```

```
vincula2(struct(LCamp)):
  vincula2(LCamp)
```

```
vincula1(lit_ent(string)):
  noop
```

```
vincula2(lit_ent(string)):
  noop
```

```
vincula1(lit_real(string)):
  noop
```

```
vincula2(lit_real(string)):
  noop
```

```
vincula1(lit_bool(string)):
  noop
```

```
vincula2(lit_bool(string)):
  noop
```

```
vincula1(lit_string(string)):
  noop
```

```
vincula2(lit_string(string)):
  noop
```

```
vincula1(muchos_camp(LCamp, Camp)):
  vincula1(LCamp)
  vincula1(Camp)
```

```
vincula2(muchos_camp(LCamp, Camp)):
  vincula2(LCamp)
```

vincula2(Camp)

vincula1(un\_camp(Camp)):  
vincula1(Camp)

vincula2(un\_camp(Camp)):  
vincula2(Camp)

vincula1(camp(T, Id)):  
vincula1(T)

vincula2(camp(T, Id)):  
vincula2(T)

#### **4.1.4 Instrucciones**

vincula1(Si\_ins(LIns)):  
vincula1(LIns)

vincula1(no\_ins()):  
noop

vincula1(muchas\_ins(LIns, Ins)):  
vincula1(LIns)  
vincula1(Ins)

vincula1(una\_ins(Ins)):  
vincula1(Ins)

vincula1(ins\_asig(Exp)):  
vincula1(Exp)

vincula1(ins\_if(Exp, Bloq)):  
vincula1(Exp)  
vincula(Bloq)

vincula1(ins\_if\_else(Exp, Bloq, Bloq)):  
vincula1(Exp)  
vincula(Bloq)  
vincula(Bloq)

vincula1(ins\_while(Exp, Bloq)):  
vincula1(Exp)  
vincula(Bloq)

```

vincula1(ins_read(Exp)):
    vincula1(Exp)
vincula1(ins_write(Exp)):
    vincula1(Exp)

vincula1(ins_nl()):
    noop

vincula1(ins_new(Exp)):
    vincula1(Exp)

vincula1(ins_delete(Exp)):
    vincula1(Exp)

vincula1(ins_call(Iden, PRealOpt)):
    if contiene(ts,Iden) then
        $.vinculo = vinculoDe(ts,Iden)
    else
        error
    end if
    vincula1(PRealOpt)

vincula1(ins_bloque(Bloq)):
    vincula(Bloq)

```

#### **4.1.5 Expresiones**

```

vincula1(si_preal(LPReal)):
    vincula1(LPReal)

vincula1(no_preal()):
    noop

vincula1(muchos_preal(LPReal, Exp)):
    vincula1(LPReal)
    vincula1(Exp)

vincula1(un_preal(Exp)):
    vincula1(Exp)

vincula1(asig(Exp0, Exp1)):
    vincula1(Exp0)
    vincula1(Exp1)

vincula1(mayor(Exp0, Exp1)):

```

```

vincula1(Exp0)
vincula1(Exp1)
vincula1(menor(Exp0, Exp1)):
    vincula1(Exp0)
    vincula1(Exp1)

vincula1(mayorIgual(Exp0, Exp1)):
    vincula1(Exp0)
    vincula1(Exp1)

vincula1(menorIgual(Exp0, Exp1)):
    vincula1(Exp0)
    vincula1(Exp1)

vincula1(igual(Exp0, Exp1)):
    vincula1(Exp0)
    vincula1(Exp1)

vincula1(desigual(Exp0, Exp1)):
    vincula1(Exp0)
    vincula1(Exp1)

vincula1(suma(Exp0, Exp1)):
    vincula1(Exp0)
    vincula1(Exp1)

vincula1(resta(Exp0, Exp1)):
    vincula1(Exp0)
    vincula1(Exp1)

vincula1(and(Exp0, Exp1)):
    vincula1(Exp0)
    vincula1(Exp1)

vincula1(or(Exp0, Exp1)):
    vincula1(Exp0)
    vincula1(Exp1)

vincula1(mul(Exp0, Exp1)):
    vincula1(Exp0)
    vincula1(Exp1)

vincula1(div(Exp0, Exp1)):
    vincula1(Exp0)

```

vincula1(Exp1)

vincula1(mod(Exp0, Exp1)):

vincula1(Exp0)

vincula1(Exp1)

vincula1(neg(Exp)):

vincula1(Exp)

vincula1(not(Exp)):

vincula1(Exp)

vincula1(acceso\_array(Exp0, Exp1)):

vincula1(Exp0)

vincula1(Exp1)

vincula1(acceso\_campo(Exp, id)):

vincula1(Exp)

vincula1(acceso\_puntero(Exp)):

vincula1(Exp)

vincula1(exp\_litEntero(N)):

noop

vincula1(exp\_litReal(R)):

noop

vincula1(exp\_litCadena(Id)):

noop

vincula1(exp\_Identificador(id)):

if contiene(ts,id) then

\$.vinculo = vinculoDe(ts,id)

else

error

end if

vincula1(exp\_litBoolTrue()):

noop

vincula1(exp\_litBoolFalse()):

noop

```
vincula1(exp_null()):  
    noop
```

## **4.2 Especificación del procedimiento de comprobación de tipos.**

```
ambos-ok(T0,T1) =  
    if T0 == ok and T1 == ok then  
        return ok  
    else  
        return error  
    end if
```

### **4.2.1 Comprobación de tipos de Programa.**

```
tipado(prog(Bloq) =  
    tipado(Bloq)  
    $.tipo = Bloq.tipo  
  
tipado(bloq(Decsopt,Insopt)) =  
    tipado(Decsopt)  
    tipado(Insopt)  
    $.tipo = ambos-ok(Insopt.tipo,Decsopt.tipo)
```

### **4.2.2 Comprobación de tipos de declaraciones.**

```
tipado(si_decs(LDecs)) =  
    tipado(LDecs)  
    $.tipo = LDecs.tipo  
  
tipado(no_decs()) =  
    $.tipo = OK  
  
tipado(una_dec(Dec)) =  
    tipado(Dec)  
    $.tipo = Dec.tipo  
  
tipado(muchas_decs(LDecs, Dec)) =  
    tipado(LDecs)  
    tipado(Dec)  
    $.tipo = ambos-ok(LDecs.tipo,Dec.tipo)  
  
tipado(dec_var(T, lden)) =  
    tipado(T)  
    $.tipo = T.tipo
```

```

tipado(dec_tipo(T, lden)) =
    tipado(T)
    $.tipo = T.tipo

tipado(dec_proc(liden,PFormOpt, Bloq)) =
    tipado(PFormOpt)
    tipado(Bloq)
    $.tipo = ambos-ok(PFormOpt.tipo, Bloq.tipo)

tipado(si_pform(LPForm)) =
    tipado(LPForm)
    $.tipo = LPForm.tipo

tipado(no_pform()) =
    $.tipo = OK

tipado(una_pform(PForm)) =
    tipado(PForm)
    $.tipo = PForm.tipo

tipado(muchas_PForms(LPForm, PForm)) =
    tipado(LPForm)
    tipado(PForm)
    $.tipo = ambos-ok(LPForm.tipo, PForm.tipo)

tipado(pform_no_ref(T, lden)) =
    tipado(T)
    $.tipo = T.tipo
tipado(pform_ref(T, lden)) =
    tipado(T)
    $.tipo = T.tipo

```

#### 4.2.3 Comprobación de tipos de tipos básicos.

```

tipado(lit_ent()) =
    $.tipo = ok

tipado(lit_real()) =
    $.tipo = ok

tipado(lit_bool()) =

```



```
$.tipo = ok
```

```
tipado(lit_string()) =  
    $.tipo = ok
```

```
tipado(array(T, litEntero)) =  
    si litEntero < 0:  
        $.tipo = error  
    sino:  
        tipado(T)  
        $.tipo = T.tipo
```

```
tipado(puntero(T)) =  
    tipado(T)  
    $.tipo = T.tipo
```

```
tipado(struct(LCamp)) =  
    si !hayRepetidos(LCamp):  
        tipado(LCamp)  
        $.tipo = LCamp.tipo  
    sino:  
        $.tipo = error
```

```
hayRepetidos(LCamp):  
    for(int i = 0; i < LCamp.length; i++):  
        for(int j = 0; j < LCamp.length; j++):  
            si LCamp[i] == LCamp[j] && i != j:  
                return true  
    return false
```

```
tipado(un_campo(Camp)) =  
    tipado(Camp)  
    $.tipo = C.tipo
```

```
tipado(muchos_campos(LCamp, Camp)) =  
    tipado(LCamp)  
    tipado(Camp)  
    $.tipo = ambos-ok(LCamp.tipo, Camp.tipo)
```

```
tipado(campo(id,T)) =  
    tipado(T)  
    $.tipo = T.tipo
```

```

tipado(iden(string)) =
    si $.vinculo = dec_tipo(T, id)añadir
        tipado(T)
        $.tipo = T.tipo

    si no:
        $.tipo = error

```

#### 4.2.4 Comprobación de tipos de instrucciones.

```
global st <- vacio()
```

*//Explicación: Si ya se ha hecho esta comprobación antes se devuelve true directamente. Si no se ha hecho se añaden y se mira si son compatibles (string solo con otro string, bool solo con otro bool etc). Si lo son se deja metido y sino se saca*

```

son_compatibles(T0,T1) =
    si estan(st, (T0, T1)):
        return true
    si no:
        añadir(st, (T0, T1))
    si ref!(T0) == int && ref!(T1) == int:
        return true
    si ref!(T0) == real && (ref!(T1) == int || ref!(T1) == real):
        return true
    si ref!(T0) == bool && ref!(T1) == bool:
        return true
    si ref!(T0) == string && ref!(T1) == string:
        return true
    si ref!(T0) == array(liden0, T2) && ref!(T1) == array(liden1, T3):
        si n0 == n1 && son_compatibles(T2, T3):
            return true
    si ref!(T0) == struct(lc1) && ref!(t1) == struct(lc2):
        si campos_compatibles(lc1, lc2):
            return true
    si ref!(T0) == puntero(T2) && T1 == null:
        return true
    si ref!(T0) == puntero(T2) && ref!(T1) == puntero(T3):
        si son_compatibles(T2, T3):
            return true
    eliminar(st, (T0, T1))

```

return false

*//Explicación: Dos campos solo son compatibles si son*

*1.un\_camp y tipos compatibles*

*2.muchos\_camp y tanto sus LCamp como los tipos lo son*

```
campos_compatibles(un_campo(campo(Iden1, T1)),  
un_campo(campo(Iden2, T2))) :  
    return son_compatibles(T1, T2)
```

```
campos_compatibles(un_campo(campo(Iden1, T1)),  
muchos_camp(LCamp, campo(Iden2, T2))) :  
    return false
```

```
campos_compatibles(muchos_camp(LCamp, camp(Iden1, T1)),  
un_camp(camp(Iden2, T2))) :  
    return false
```

```
campos_compatibles(muchos_camp(LCamp1,camp(Iden1,T1)),  
muchos_camp(LCamp2, camp(Iden2,T2))) :  
    return ambos-ok(campos_compatibles(LCamp1,LCamp2),  
son_compatibles(T1,T2))
```

```
tipado(si_ins(LIns)) :  
    Tipado(LIns)  
    $.tipo = LIns.tipo
```

```
tipado(no_ins()) :  
    $.tipo = OK
```

```
tipado(una_ins(Ins)):  
    tipado(Ins)  
    $.tipo = Ins.tipo
```

```
tipado(muchas_ins(LIns, Ins)) :  
    tipado(LIns)  
    tipado(Ins)  
    $.tipo = ambos-ok(LIns.tipo, Ins.tipo)
```

```
tipado(ins_asig(Exp)) :  
    si tipado(Exp) == ok:  
        $.tipo = ok  
    si no:
```

\$.tipo = ERROR

```
tipado(ins_if(Exp, Bloq)):  
  tipado(Exp)  
  si ref!(Exp.tipo) == bool:  
    si tipado(Bloq) == ok:  
      $.tipo = ok  
    si no:  
      $.tipo = ERROR
```

```
  si no:  
    $.tipo = ERROR
```

```
tipado(ins_if_else(Exp, Bloq1, Bloq2)) :  
  tipado(Exp)  
  si ref!(E.tipo) == bool  
    si tipado(Bloq1) == OK && tipado(Bloq2) == OK:  
      $.tipo = OK  
    si no:  
      $.tipo = ERROR  
  si no:  
    $.tipo = ERROR
```

```
tipado(ins_while(Exp, Bloq)) :  
  tipado(Exp)  
  si ref!(Exp.tipo) == bool:  
    si tipado(Bloq) == ok:  
      $.tipo = ok  
    si no:  
      $.tipo = ERROR  
  
  si no:  
    $.tipo = ERROR
```

```
tipado(ins_read(Exp)) :  
  tipado(Exp)  
  si (ref!(Exp.tipo) == int || ref!(Exp.tipo) == real || ref!(Exp.tipo) == string)  
&& es_desig(Exp):  
    $.tipo = OK  
  si no:  
    $.tipo = ERROR
```

```
es_desig(Exp):  
  si E == iden || E == acceso_array || E == acceso_puntero || E ==  
acceso_struct:  
    return true  
  return false
```

```

tipado(ins_write(Exp)) :
    tipado(Exp)
    si (ref!(Exp.tipo) == int || ref!(Exp.tipo) == real || ref!(Exp.tipo) == bool ||
ref!(Exp.tipo) == string):
        $.tipo = OK
    si no:
        $.tipo = ERROR

```

```

tipado(ins_nl):
    $.tipo = OK

```

```

tipado(ins_new(Exp)) :
    tipado(Exp)
    si ref!(Exp.tipo) == puntero:
        $.tipo = OK
    si no:
        $.tipo = ERROR

```

```

tipado(ins_delete(Exp)) :
    tipado(Exp)
    si ref!(Exp.tipo) == puntero:
        $.tipo = OK
    si no:
        $.tipo = ERROR

```

```

tipado(ins_bloque(Bloq)) :
    tipado(Bloq)
    $.tipo = Bloq.tipo

```

```

tipado(ins_call(string PRealOpt)):
    tipado(PRealOpt)
    let string.vinculo = dec_proc dec_proc = dec_proc(id,PForm,_)
    PRealOpt.tipo = chequeo_params(PForm, PRealOpt)
    si PRealOpt.tipo = ok:
        $.tipo = ok
    si no:
        $.tipo = error

```

```

chequeo_params(no_PForm, no_PReal): return ok

```

```

chequeo_params(un_PForm, un_PReal):
    return chequeo_params(E, PForm)

```

```

chequeo_params(Muchos_PReal(LPReal, Exp),
muchos_PForm(LPForm,PForm)):

    return ambos_ok(chequeo_params(LPReal,LPForm),
chequeo_parametro(Exp,PForm))

```

```

chequeo_params(Exp, PForm_no_ref(id, T)):
    tipado(Exp)
    tipado(PForm_no_ref)
    si son_compatibles(T, E.tipo):
        return ok
    si no:
        return error

chequeo_params(Exp, PForm_ref(id,T)) =
    tipado(Exp)
    tipado(PForm_ref)
    si es_desig(Exp) && son_compatibles(T, Exp.tipo)
        return ok
    si no:
        return error

```

#### 4.2.5 Comprobación de tipos de expresiones.

```

tipado(si_preal(LPReal)):
    tipado(LPReal)
    $.tipo = LPReal.tipo

tipado(no_preal()) :
    $.tipo = OK

tipado(un_preal(PReal)):
    tipado(PReal)
    $.tipo = PReal.tipo

tipado(muchos_PReal(LPReal, Exp)) :
    tipado(LPReal)
    tipado(Exp)
    $.tipo = ambos-ok(LPReal.tipo, Exp.tipo)

tipado(exp_litEntero(N)) :
    $.tipo = int

tipado(exp_litReal(R)) :
    $.tipo = real

tipado(exp_litEntero(N)):
    $.tipo = int

```

```
tipado(exp_litBoolTrue()) :  
    $.tipo = bool
```

```
tipado(exp_litBoolFalse()):  
    $.tipo = bool
```

```
tipado(exp_litCadena(String)) :  
    $.tipo = string
```

```
tipado(exp_identificador(String)) :  
    si $.vinculo == dec_var(String, T) || $.vinculo == pform_ref(T, String) ||  
    $.vinculo == pform_no_ref(T, String):  
        $.tipo = T  
    si no:  
        $.tipo = ERROR
```

```
tipado(exp_null()) :  
    $.tipo = null
```

```
tipado(menor(Exp0, Exp1)) :  
    tipado(Exp0)  
    tipado(Exp1)  
    $.tipo = tipo_relacional(E0.tipo, E1.tipo)
```

```
tipado(menorIgual(Exp0, Exp1)):  
    tipado(Exp0)  
    tipado(Exp1)  
    $.tipo = tipo_relacional(Exp0.tipo, Exp1.tipo)
```

```
tipado(mayor(Exp0, Exp1)):  
    tipado(Exp0)  
    tipado(Exp1)  
    $.tipo = tipo_relacional(Exp0.tipo, Exp1.tipo)
```

```
tipado(MayorIgual(Exp0, Exp1)):  
    tipado(Exp0)  
    tipado(Exp1)  
    $.tipo = tipo_relacional(Exp0.tipo, Exp1.tipo)
```

```

tipo_relacional(T0, T1):
    si (ref!(T0) == int || ref!(T0) == real) && (ref!(T1) == int || ref!(T1) == real):
        return bool
    si no:
        si ref!(T0) == bool && ref!(T1) == bool:
            return bool
        si no:
            si ref!(T0) == string && ref!(T1) == string:
                return bool
            si no:
                return error

```

```

tipado(igual(Exp0, Exp1)) :
    tipado(Exp0)
    tipado(Exp1)
    $.tipo = tipo_relacional2(Exp0.tipo, Exp1.tipo)

```

```

tipado(desigual(Exp0, Exp1)) :
    tipado(Exp0)
    tipado(Exp1)
    $.tipo = tipo_relacional2(Exp0.tipo, Exp1.tipo)

```

```

tipo_relacional2(T0, T1):
    si (ref!(T0) == int || ref!(T0) == real) && (ref!(T1) == int || ref!(T1) == real):
        return bool
    si no:
        si ref!(T0) == bool && ref!(T1) == bool:
            return bool
        si no:
            si ref!(T0) == string && ref!(T1) == string:
                return bool
            si ref!(T0) == puntero && ref!(T1) == puntero:
                return bool
        si no:
            si (ref!(T0) == puntero && ref!(T1) == null) || (ref!(T0) == null &&
                ref!(T1) == puntero):
                return bool
        si no:
            si ref!(T0) == null && ref!(T1) == null:
                return bool

```



```

        si no:
            return error
tipado(and(Exp0, Exp1)) :
    tipado(Exp0)
    tipado(Exp1)
    $.tipo = tipo_logico(Exp0.tipo, Exp1.tipo)

tipado(or(Exp0, Exp1)) :
    tipado(Exp0)
    tipado(Exp1)
    $.tipo = tipo_logico(Exp0.tipo, Exp1.tipo)

tipo_logico(T0, T1) :
    si ref!(T0) == bool && ref!(T1) == bool:
        return bool
    si no:
        return error

tipado(suma(Exp0, Exp1)) :
    tipado(Exp0)
    tipado(Exp1)
    $.tipo = tipo_binat(Exp0.tipo, Exp1.tipo)

tipado(resta(Exp0, Exp1)) :
    tipado(Exp0)
    tipado(Exp1)
    $.tipo = tipo_binat(Exp0.tipo, Exp1.tipo)

tipado(mul(Exp0, Exp1)) :
    tipado(Exp0)
    tipado(Exp1)
    $.tipo = tipo_binat(Exp0.tipo, Exp1.tipo)

tipado(div(Exp0, Exp1)) :
    tipado(Exp0)
    tipado(Exp1)
    $.tipo = tipo_binat(Exp0.tipo, Exp1.tipo)

```

```

tipo_binat(T0, T1) :
    si ref!(T0) == int && ref!(T1) == int:
        return int
    si no:
        si (ref!(T0) == real && (ref!(T1) == int || ref!(T1) == real)) || (ref!(T1)
== real && (ref!(T0) == int || ref!(T0) == real)):
            return real
        si no:
            return error

```

```

tipado(mod(Exp0, Exp1)) :
    tipado(Exp0)
    tipado(Exp1)
    tipo_mod_ent(Exp0.tipo, Exp1.tipo)
    si ref!(T0) == int && ref!(T1) == int:
        $.tipo = int
    si no:
        $.tipo = error

```

```

tipado(neg(Exp)) :
    tipado(Exp)
    si ref!(Exp.tipo) == int:
        $.tipo = int
    si no si ref!(Exp.tipo) == real:
        $.tipo = real
    si no:
        $.tipo = ERROR

```

```

tipado(not(Exp)) :
    tipado(Exp)
    si ref!(Exp.tipo) == bool:
        $.tipo = bool
    si no:
        $.tipo = ERROR

```

```

tipado(acceso_array(Exp0, Exp1)) :
    tipado(Exp0)
    tipado(Exp1)
    si ref!(Exp0.tipo) = array(n, tb)
        si ref!(Exp1.tipo) == int
            $.tipo = tb
        si no
            return error
    si no
        return error

```

```

tipado(acceso_campo(Exp, c)) :
    tipado(Exp)
    si ref!(Exp.tipo) == struct(LCampos):
        $.tipo = tipo_de(LCampos, c)
    si no:
        $.tipo = error

```

```

tipo_de(LCampos, c):
    i = 0
    mientras i < LCampos.length:
        sea LCampos[i] = campo(id, T):
            si c == id:
                return T
    return error

```

```

tipado(acceso_puntero(Exp)):
    tipado(Exp)
    si ref!(Exp.tipo) == puntero(T):
        $.tipo = T
    si no:
        $.tipo = error

```

```

tipado(asig(Exp0,Exp1)):
    tipado(Exp0)
    tipado(Exp1)
    if es_desig(Exp0) then
        if compatibles(Exp0.tipo, Exp1.tipo) then
            $.tipo = ok
        else:
            $.tipo = error
        end if
    else:
        $.tipo = error
    end if

```

### **4.3 Especificación del procesamiento de asignación de espacio.**

```

var dir = 0
var nivel = 0
var max_dir = 0
var desplazamiento = 0

asig_espacio(prog(Bloque)):
    asig_espacio(Bloque)

asig_espacio(bloque(Decsopt, Insopt)):
    dir_ant = dir
    asig_espacio(Decsopt)
    asig_espacio(Insopt)
    dir = dir_ant

asig_espacio(si_decs(LDecs)):
    asig_espacio1(LDecs)
    asig_espacio2(LDecs)

asig_espacio(no_decs()):
    skip

```

#### **4.3.1 Primera pasada en la sección de declaraciones**

```

asig_espacio1(muchas_decs(LDecs, Dec)):
    asig_espacio1(LDecs)
    asig_espacio1(Dec)

asig_espacio1(una_dec(Dec)):

```

```

        asig_espacio1(Dec)
asig_espacio1(dec_var(Tipo, Iden)):
    $.dir = dir
    $.nivel = nivel
    asig_tam1(Tipo)
    dir = dir + Tipo.tam

asig_espacio1(dec_tipo(Tipo, Iden)):
    asig_tam1(Tipo)

asig_espacio1(dec_proc(Iden, PFormOpt, Bloque)):
    dir_ant = dir
    max_dir_ant = max_dir
    nivel++
    $.nivel = nivel
    dir = 0
    max_dir = 0
    asig_espacio1(PFormOpt)
    asig_espacio2(PFormOpt)
    asig_espacio(Bloque)
    $.tam = dir
    dir = dir_ant
    max_dir = max_dir_ant
    nivel = nivel - 1

asig_espacio1(si_pform(LPForm)):
    asig_espacio1(LPForm)

asig_espacio1(no_pform()):
    skip

asig_espacio1(muchas_pforms(LPForm, PForm)):
    asig_espacio1(LPForm)
    asig_espacio1(PForm)

asig_espacio1(una_pform(PForm)):
    asig_espacio1(PForm)

asig_espacio1(pref(Tipo, Iden)):
    $.dir = dir
    $.nivel = nivel
    asig_tam1(Tipo)
    dir = dir + 1

```

```
asig_espacio1(pnoref(Tipo,Iden)):
    $.dir = dir
    $.nivel = nivel
    asig_tam1(Tipo)
    dir = dir + Tipo.tam
```

#### **4.3.2 Segunda pasada en la sección de declaraciones**

```
asig_espacio2(muchas_decs(LDecs, Dec)):
    asig_espacio2(LDecs)
    asig_espacio2(Dec)
```

```
asig_espacio2(una_dec(Dec)):
    asig_espacio2(Dec)
```

```
asig_espacio2(dec_var(Tipo, Iden)):
    asig_tam2(Tipo)
```

```
asig_espacio2(dec_tipo(Tipo, Iden)):
    asig_tam2(Tipo)
```

```
asig_espacio2(dec_proc(Iden, PFormOpt, Bloque)):
    skip
```

```
asig_espacio2(si_pform(LPForm)):
    asig_espacio2(LPForm)
```

```
asig_espacio2(no_pform()):
    skip
```

```
asig_espacio2(muchas_pforms(LPForm, PForm)):
    asig_espacio2(LPForm)
    asig_espacio2(PForm)
```

```
asig_espacio2(una_pform(PForm)):
    asig_espacio2(PForm)
```

```
asig_espacio1(pref(Tipo, Iden)):
    $.dir = dir
    $.nivel = nivel
    asig_tam1(Tipo)
    dir = dir + 1
```

```

asig_espacio1(pnoref(Tipo,Iden)):
    $.dir = dir
    $.nivel = nivel
    asig_tam1(Tipo)
    dir = dir + Tipo.tam

```

### 4.3.3 Asignación del tamaño de los tipos.

```

asig_tam1(int()):
    $.tam = 1

```

```

asig_tam2(int()):
    skip

```

```

asig_tam1(real()):
    $.tam = 1

```

```

asig_tam2(real()) :
    skip

```

```

asig_tam1(bool()):
    $.tam = 1

```

```

asig_tam2(bool()):
    skip

```

```

asig_tam1(string()):
    $.tam = 1

```

```

asig_tam2(string()):
    skip

```

```

asig_tam1(puntero(Tipo)):
    if Tipo != iden(string)
        asig_tam1(Tipo)
    $.tam = 1

```

```

asig_tam2(puntero(Tipo)):
    if Tipo = iden(string)
        sea Tipo.vinculo = dec_tipo(T', id) :
            Tipo.tam = T'.tam
    else
        asig_tam2(Tipo)

```

```
asig_tam1(iden(string)):
    sea $.vinculo = dec_tipo(T, id):
        $.tam = T.tam
```

```
asig_tam2(iden(string)):
    skip
```

```
asig_tam1(struct(LCampos)):
    $.tam = asig_tam1(LCampos)
```

```
asig_tam2(struct(LCampos)):
    asig_tam2(LCampos)
```

```
asig_tam1(un_campo(Campo)):
    sea Campo = campo(T, id):
        asig_tam1(T)
        Campo.desp = desplazamiento
    return desplazamiento + T.tam
```

```
asig_tam2(un_campo(Campo)):
    sea Campo = campo(T, id):
        Campo.desp = desplazamiento
    asig_tam2(T)
```

```
asig_tam1(muchos_campos(LCampos, Campo)):
    d_act = asig_tam1(LCampos)
    sea Campo = campo(T, id):
        asig_tam1(T)
        Campo.desp = d_act
    return d_act + T.tam
```

```
asig_tam2(muchos_campos(LCampos, Campo)) =
    asig_tam2(LCampos)
    sea Campo = campo(T, id):
        asigna_tam2(T)
```

```
asig_tam1(array(Tipo, n)):
    asig_tam1(Tipo)
    $.tam = val(n) * Tipo.tam
```

```
asig_tam2(array(Tipo, n)):
    asig_tam2(Tipo)
```



#### 4.3.4 Asignación del espacio de instrucciones.

```
asig_espacio(si_ins(LIns)):  
    asig_espacio(LIns)
```

```
asig_espacio(no_ins()):  
    skip
```

```
asig_espacio(una_ins(Ins)):  
    asig_espacio(Ins)
```

```
asig_espacio(muchas_ins(LIns, Ins)):  
    asig_espacio(LIns)  
    asig_espacio(Ins)
```

```
asig_espacio(ins_asig(E)):  
    skip
```

```
asig_espacio(ins_if(E, Bloque)):  
    asig_espacio(Bloque)
```

```
asig_espacio(ins_if_else(E, Bloque1, Bloque2)):  
    asig_espacio(Bloque1)  
    asig_espacio(Bloque2)  
asig_espacio(ins_while(E, Bloque)):  
    asig_espacio(Bloque)
```

```
asig_espacio(ins_read(E)):  
    skip
```

```
asig_espacio(ins_write(E)):  
    skip
```

```
asig_espacio(ins_nl()):  
    skip
```

```
asig_espacio(ins_new(E)):  
    skip
```

```
asig_espacio(ins_delete(E)):  
    skip
```

```
asig_espacio(ins_bloque(Bloque)):  
    asig_espacio(Bloque)
```

```
asig_espacio(ins_call(E)):  
    skip
```

#### **4.4 Descripción del repertorio de instrucciones de la máquina-p necesario para soportar la traducción de Tiny a código-p.**

Instrucciones que hacen falta incluir para extender la máquina-p y que así admita todas las operaciones de Tiny:

**apila\_int(I):** Apila un valor I de tipo int

**apila\_real(R):** Apila un valor R de tipo real

**apila\_bool(B):** Apila un valor B de tipo bool

**apila\_string(S):** Apila un valor S de tipo string

**apila\_dir(d):** Apila el valor que se encuentra en la dirección d de la memoria de datos

**desapila\_dir(d):** Desapila la cima y almacena su valor en la dirección d de la memoria de datos

**apila\_ind:** Desapila un valor d de la cima de la pila y lo usa como dirección de la memoria de datos cuyo valor apila.

**desapila\_ind:** Desapila un valor v de la cima de la pila y un segundo valor d de la subcima. Almacena el valor v en la posición d de la memoria de datos.

**apilad(n):** Apila el valor del display de nivel n

**desapilad(n):** Desapila el valor del display de nivel n

**suma\_int:** desapila dos valores enteros y apila el resultado de su suma

**resta\_int:** desapila dos valores enteros y apila el resultado de su resta (subcima - cima)

**mul\_int:** desapila dos valores enteros y apila el resultado de su multiplicación

**div\_int:** desapila dos valores enteros y apila el resultado de su división(subcima/cima)

**neg\_int:** desapila un valor entero y apila el resultado de su negación

**suma\_real:** desapila dos valores reales y apila el resultado de su suma

**resta\_real:** desapila dos valores reales y apila el resultado de su resta (subcima - cima)

**mul\_real:** desapila dos valores reales y apila el resultado de su multiplicación

**div\_real:** desapila dos valores reales y apila el resultado de su división(subcima/cima)

**neg\_real:** desapila un valor real y apila el resultado de su negación

**mod:** desapila dos valores y apila el resultado de "subcima % cima"

**not:** desapila un valor y apila el resultado de su negación lógica

**and:** desapila dos valores y apila el resultado de su conjunción lógica

**or:** desapila dos valores y apila el resultado de su disyunción lógica

**menor\_int:** desapila dos valores enteros y apila el resultado de “subcima < cima”

**menor\_real:** desapila dos valores reales y apila el resultado de “subcima < cima”

**menor\_bool:** desapila dos valores booleanos y apila el resultado de “subcima < cima”

**menor\_string:** desapila dos cadenas y apila el resultado de “subcima < cima”

**mayor\_int:** desapila dos valores enteros y apila el resultado de “subcima > cima”

**mayor\_real:** desapila dos valores reales y apila el resultado de “subcima > cima”

**mayor\_bool:** desapila dos valores booleanos y apila el resultado de “subcima > cima”

**mayor\_string:** desapila dos cadenas y apila el resultado de “subcima > cima”

**menor\_igual\_int:** desapila dos valores enteros y apila el resultado de “subcima <= cima”

**menor\_igual\_real:** desapila dos valores reales y apila el resultado de “subcima <= cima”

**menor\_igual\_bool:** desapila dos valores booleanos y apila el resultado de “subcima <= cima”

**menor\_igual\_string:** desapila dos cadenas y apila el resultado de “subcima <= cima”

**mayor\_igual\_int:** desapila dos valores enteros y apila el resultado de “subcima >= cima”

**mayor\_igual\_real:** desapila dos valores reales y apila el resultado de “subcima >= cima”

**mayor\_igual\_bool:** desapila dos valores booleanos y apila el resultado de “subcima >= cima”

**mayor\_igual\_string:** desapila dos cadenas y apila el resultado de “subcima >= cima”

**igual\_int:** desapila dos valores enteros y apila el resultado de “subcima == cima”

**igual\_real:** desapila dos valores reales y apila el resultado de “subcima == cima”

**igual\_bool:** desapila dos valores booleanos y apila el resultado de “subcima == cima”

**igual\_string:** desapila dos cadenas y apila el resultado de “subcima == cima”

**igual\_puntero:** desapila dos punteros y apila el resultado de “subcima == cima”

**igual\_null:** desapila dos tipos null y apila el resultado de “subcima == cima”

**igual:** desapila un valor null y un valor puntero y apila el resultado de “subcima == cima”

**desigual\_int:** desapila dos valores enteros y apila el resultado de “subcima != cima”

**desigual\_real:** desapila dos valores reales y apila el resultado de “subcima != cima”

**desigual\_bool:** desapila dos valores booleanos y apila el resultado de “subcima != cima”

**desigual\_string:** desapila dos cadenas y apila el resultado de “subcima != cima”

**desigual\_puntero:** desapila dos punteros y apila el resultado de “subcima != cima”

**desigual\_null:** desapila dos tipos null y apila el resultado de “subcima != cima”

**desigual:** desapila un valor null y un valor puntero y apila el resultado de “subcima != cima”

**mueve(n):** copia n celdas desde la dirección de origen, extraída de la cima de la pila, a la dirección destino, de la subcima.

**ir\_a(d):** salta de manera incondicional a la instrucción en la dirección d de memoria de programa

**ir\_v(d):** salta a la instrucción en la dirección d de la memoria de programa si el valor de la cima es true y desapila este valor

**ir\_f(d):** salta a la instrucción en la dirección d de la memoria de programa si el valor de la cima es false y desapila este valor

**ir\_ind():** desapila un valor d de la cima de la pila y salta a la instrucción en la dirección d de la memoria de programa.

**alloc(t):** con un tipo t. Se reserva una zona de memoria adecuada para almacenar valores del tipo . La operación en sí devuelve la dirección de comienzo de dicha zona de memoria.

**dealloc(d, t):** con d una dirección, y un tipo. Se notifica que la zona de memoria que comienza en d y que permite almacenar valores del tipo queda liberada.

**fetch(d):** on d una dirección. Devuelve el valor almacenado en la celda direccionada por d.

**store(d,v):** con d una dirección, y v un valor. Almacena v en la celda direccionada por d.

**copy(t):** Con t el tamaño de un tipo. Desapila 2 direcciones y copia el contenido de tamaño t empezando por d1 en un espacio que empieza en d2.

**indx(d,i, t):** con d una dirección, i un valor, y un tipo. Considera que, a partir de d, comienza un array cuyos elementos son valores del tipo , y devuelve la dirección de comienzo del elemento i-esimo de dicho array.

**acc(d,c, t):** con d una dirección, c un nombre de campo, y un tipo record. Considera que, a partir de d, está almacenado un registro de tipo , que contiene un campo c. Devuelve la dirección de comienzo de dicho campo.

**activa(n,t,d):** Reserva espacio en el segmento de pila de registros de activación para un procedimiento con nivel de anidamiento n y tamaño de datos locales t. Almacena la dirección d en la zona de control del registro como dirección de retorno. Almacena en la zona de control el valor del display de nivel n. Apila la dirección de comienzo de los datos en el registro

**desactiva(n,t):** Libera el espacio ocupado por el registro de activación actual, restaura el estado de la máquina. n es el nivel de anidamiento del procedimiento; t el tamaño de los datos locales. Apila la dirección de retorno; Restaura el valor del display de nivel n al antiguo valor guardado en el registro; Decrementa el puntero de pila de registros de activación en el tamaño ocupado por el registro

**dup():** Duplica la cima de la pila

**stop():** Detiene la ejecución de la máquina

**int2real():** desapila un valor entero y apila el mismo valor convertido a real

**leer\_entrada\_int():** lee un valor entero de la entrada estandar y lo apila

**leer\_entrada\_real():** lee un valor real de la entrada estandar y lo apila

**leer\_entrada\_string():** leerá una cadena de la entrada estandar y lo apila

**mostrar\_int():** imprime en salida estandar el dato entero de la cima y lo desapila.

**mostrar\_real():** imprime en salida estandar el dato real de la cima y lo desapila.

**mostrar\_bool():** imprime en salida estandar el dato booleano de la cima y lo desapila.

**mostrar\_string():** imprime en salida estandar la cadena de la cima y lo desapila.

**nl():** mostrará en la salida estándar un salto de línea.

**desechar():** desapila la cima y desecha el valor.

## **4.5 Especificación del procesamiento de etiquetado.**

### **4.5.1 Etiquetado para el programa.**

global procs = pila\_vacia()

global etq = 0

etiqueta(prog(Bloq)):

etiqueta(Bloq)

```
etiqueta(Bloq(Decsopt, Insopt)):
```

```
  $.prim = etq
  etiqueta(Decsopt)
  etq <- etq + 1
  etiqueta(Insopt)
  $.sig = etq
```

```
etiqueta(si_decs()):
```

```
  etiqueta(LDecs)
```

```
etiqueta(no_dec()): noop
```

```
etiqueta(muchas_decs(LDecs, Dec)):
```

```
  etiqueta(LDecs)
  etiqueta(Dec)
```

```
etiqueta(una_dec(Dec)):
```

```
  etiqueta(Dec)
```

#### **4.5.2 Etiquetado para las instrucciones.**

```
etiqueta(si_ins(LIns)):
```

```
  etiqueta(LIns)
```

```
etiqueta(no_ins()):
```

```
  noop
```

```
etiqueta(una_ins(Ins)) =
```

```
  etiqueta(Ins)
```

```
etiqueta(muchas_ins(LIns, Ins)) =
```

```
  etiqueta(LIns)
  etiqueta(Ins)
```

```
etiqueta(ins_asig(Exp)):
```

```
  etiqueta(Exp)
  t = ref!(Exp.tipo)
  si t == int || t == real || t == string
    si es_desig(Exp)
      etq <- etq + 1
    etq <- etq + 2
  sino
    etq <- etq + 1
```

```
etiqueta(ins_if(Exp, Bloq)):
```

```
    $.prim = etq
    etiqueta(Exp)
    si es_desig(Exp):
        etq <- etq + 1
    etq <- etq + 1
    etiqueta(Bloq)
    $.sig = etq
```

```
etiqueta(if_then_else(Exp, Bloq0, Bloq1)):
```

```
    $.prim = etq
    etiqueta(Exp)
    si es_desig(Exp):
        etq <- etq + 1
    etq <- etq + 1
    etiqueta(Bloq0)
    $.prim2 = etq
    etq <- etq + 1
    etiqueta(Bloq1)
    $.sig = etq
```

```
etiqueta(ins_call(Iden, Exp)):
```

```
    etiqueta(Exp)
    si es_desig(Exp):
        etq <- etq + 1
    etq <- etq + 1
```

```
etiqueta(while(Exp, Bloq)):
```

```
    $.prim = etq
    etiqueta(Exp)
    si es_desig(Exp):
        etq <- etq + 1
    etq <- etq + 1
    etiqueta(Bloq)
    etq <- etq + 1
    $.sig = etq
```

```
etiqueta(read(Exp)):
```

```
    etiqueta(Exp)
    t = ref!(Exp.tipo)
    si t == int || t == real || t == string
        etq <- etq + 2
```

```
etiqueta(write(Exp)):  
  etiqueta(Exp)  
  si es_desig(Exp):  
    etq <- etq + 1  
  etq <- etq + 1
```

```
etiqueta(ins_nl()):
```

```
etiqueta(new(Exp)):  
  etiqueta(E)  
  etq <- etq + 2
```

```
etiqueta(delete(Exp)):  
  etiqueta(Exp)  
  etq <- etq + 2
```

```
etiqueta(bloque(Exp)):  
  etiqueta(Exp)  
  etq <- etq
```

#### **4.5.3 Etiquetado para las expresiones.**

```
etiqueta(si_preal(LPRealOpt)):  
  etiqueta(LPReal)
```

```
etiqueta(no_preal()): noop
```

```
etiqueta(muchos_preal(LPReal, Exp)):  
  etiqueta(LPReal)  
  etiqueta(Exp)
```

```
etiqueta(un_preal(Exp)):  
  etiqueta(Exp)
```

```
etiqueta(exp_litEntero(N)):  
  etq <- etq + 1
```

```
etiqueta(exp_litReal(N)):  
  etq <- etq + 1
```

```
etiqueta(exp_litBoolTrue()):  
  etq <- etq + 1
```

```
etiqueta(exp_litBoolFalse()):
```



```

    etq <- etq + 1
etiqueta(exp_litCadena(N)):
    etq <- etq + 1

etiqueta(exp_identificador(Id)):
    etq <- etq + 1

etiqueta(exp_null()):
    etq <- etq + 1

```

#### 4.5.4 Operaciones

```

etiqueta(asig(Exp0,Exp1)):
    etiqueta(Exp0)
    etiqueta(Exp1)
    etq <- etq + 1
    if ref!(Exp1.tipo) == int & ref!(Exp0.tipo) == real
        if es_desig(Exp1)
            etq <- etq + 1
        etq <- etq + 2
    else
        if es_desig(Exp1)
            etq <- etq + 1

etiqueta(suma(Exp0,Exp1)):
    t0 = ref!(Exp0.tipo)
    t1 = ref!(Exp1.tipo)

    etiqueta(Exp0)
    if es_desig(Exp0)
        etq <- etq + 1
    if t0 == int & t1 == real
        etq <- etq + 1

    etiqueta(Exp1)
    if es_desig(Exp1)
        etq <- etq + 1
    else if t0==real & t1 ==int
        etq <- etq + 1

    etq <- etq + 1

```

```

etiqueta(resta(Exp0,Exp1)):
  t0 = ref!(Exp0.tipo)
  t1 = ref!(Exp1.tipo)

  etiqueta(Exp0)
  if es_desig(Exp0)
    etq <- etq + 1
  if t0 == int & t1 == real
    etq <- etq + 1

  etiqueta(Exp1)
  if es_desig(Exp1)
    etq <- etq + 1
  else if t0==real & t1 ==int
    etq <- etq + 1

  etq <- etq + 1

```

```

etiqueta(mul(Exp0,Exp1)):
  t0 = ref!(Exp0.tipo)
  t1 = ref!(Exp1.tipo)

  etiqueta(Exp0)
  if es_desig(Exp0)
    etq <- etq + 1
  if t0 == int & t1 == real
    etq <- etq + 1

  etiqueta(Exp1)
  if es_desig(Exp1)
    etq <- etq + 1
  else if t0==real & t1 ==int
    etq <- etq + 1

  etq <- etq + 1

```

```

etiqueta(div(Exp0,Exp1)):
    t0 = ref!(Exp0.tipo)
    t1 = ref!(Exp1.tipo)

    etiqueta(Exp0)
    if es_desig(Exp0)
        etq <- etq + 1
    if t0 == int & t1 == real
        etq <- etq + 1

    etiqueta(Exp1)
    if es_desig(Exp1)
        etq <- etq + 1
    else if t0==real & t1 ==int
        etq <- etq + 1

    etq <- etq + 1

etiqueta(mod(Exp0,Exp1)):
    t0 = ref!(Exp0.tipo)
    t1 = ref!(Exp1.tipo)

    etiqueta(Exp0)
    if es_desig(Exp0)
        etq <- etq + 1

    etiqueta(Exp1)
    if es_desig(Exp1)
        etq <- etq + 1

    etq <- etq + 1

```

```
etiqueta(and(Exp0,Exp1)):
  t0 = ref!(Exp0.tipo)
  t1 = ref!(Exp1.tipo)

  etiqueta(Exp0)
  if es_desig(Exp0)
    etq <- etq + 1

  etiqueta(Exp1)
  if es_desig(Exp1)
    etq <- etq + 1

  etq <- etq + 1
```

```
etiqueta(or(Exp0,Exp1)):
  t0 = ref!(Exp0.tipo)
  t1 = ref!(Exp1.tipo)

  etiqueta(Exp0)
  if es_desig(Exp0)
    etq <- etq + 1

  etiqueta(Exp1)
  if es_desig(Exp1)
    etq <- etq + 1

  etq <- etq + 1
```

```

etiqueta(mayor(Exp0,Exp1)):
    t0 = ref!(Exp0.tipo)
    t1 = ref!(Exp1.tipo)

    etiqueta(Exp0)
    if es_desig(Exp0)
        etq <- etq + 1
    if t0 == int & t1 == real
        etq <- etq + 1

    etiqueta(Exp1)
    if es_desig(Exp1)
        etq <- etq + 1
    else if t0==real & t1 ==int
        etq <- etq + 1

    etq <- etq + 1

etiqueta(menor(Exp0,Exp1)):
    t0 = ref!(Exp0.tipo)
    t1 = ref!(Exp1.tipo)

    etiqueta(Exp0)
    if es_desig(Exp0)
        etq <- etq + 1
    if t0 == int & t1 == real
        etq <- etq + 1

    etiqueta(Exp1)
    if es_desig(Exp1)
        etq <- etq + 1
    else if t0==real & t1 ==int
        etq <- etq + 1

    etq <- etq + 1

```

```
etiqueta(menorIgual(Exp0,Exp1)):
```

```
  t0 = ref!(Exp0.tipo)
```

```
  t1 = ref!(Exp1.tipo)
```

```
  etiqueta(Exp0)
```

```
  if es_desig(Exp0)
```

```
    etq <- etq + 1
```

```
  if t0 == int & t1 == real
```

```
    etq <- etq + 1
```

```
  etiqueta(Exp1)
```

```
  if es_desig(Exp1)
```

```
    etq <- etq + 1
```

```
  else if t0==real & t1 ==int
```

```
    etq <- etq + 1
```

```
  etq <- etq + 1
```

```
etiqueta(mayorIgual(Exp0,Exp1)):
```

```
  t0 = ref!(Exp0.tipo)
```

```
  t1 = ref!(Exp1.tipo)
```

```
  etiqueta(Exp0)
```

```
  if es_desig(Exp0)
```

```
    etq <- etq + 1
```

```
  if t0 == int & t1 == real
```

```
    etq <- etq + 1
```

```
  etiqueta(Exp1)
```

```
  if es_desig(Exp1)
```

```
    etq <- etq + 1
```

```
  else if t0==real & t1 ==int
```

```
    etq <- etq + 1
```

```
  etq <- etq + 1
```

```

etiqueta(desigual(Exp0,Exp1)):
    t0 = ref!(Exp0.tipo)
    t1 = ref!(Exp1.tipo)

    etiqueta(Exp0)
    if es_desig(Exp0)
        etq <- etq + 1
    if t0 == int & t1 == real
        etq <- etq + 1

    etiqueta(Exp1)
    if es_desig(Exp1)
        etq <- etq + 1
    else if t0==real & t1 ==int
        etq <- etq + 1

    etq <- etq + 1

etiqueta(igual(Exp0,Exp1)):
    t0 = ref!(Exp0.tipo)
    t1 = ref!(Exp1.tipo)

    etiqueta(Exp0)
    if es_desig(Exp0)
        etq <- etq + 1
    if t0 == int & t1 == real
        etq <- etq + 1

    etiqueta(Exp1)
    if es_desig(Exp1)
        etq <- etq + 1
    else if t0==real & t1 ==int
        etq <- etq + 1

    etq <- etq + 1

etiqueta(neg(Exp0)):
    t0 = ref!(Exp0.tipo)

    etiqueta(Exp0)
    if es_desig(Exp0)
        etq <- etq + 1

```

```

    etq <- etq + 1
etiqueta(not(Exp0)):
    t0 = ref!(Exp0.tipo)

    etiqueta(Exp0)
    if es_desig(Exp0)
        etq <- etq + 1

    etq <- etq + 1

etiqueta(acceso_array(Exp0,Exp1)):
    etiqueta(Exp0)
    etiqueta(Exp1)

    if es_desig(ref!(Exp0))
        etq <- etq + 1
    etq <- etq + 1

etiqueta(acceso_campo(Exp, Campo)):
    etiqueta(Exp)

etiqueta(acceso_puntero(Exp)):
    etiqueta(Exp)

```

## **4.6 Especificación del procesamiento de generación de código.**

### **4.6.1 Generación de código para programa.**

```

global procs = pila_vacia()

gen-cod(prog(Bloq)):
    gen-cod(Bloq)

```



```

gen-cod(Bloq(DecsOpt, InsOpt)):
  recolecta_procs(DecsOpt)
  gen-cod(InsOpt)
  emit stop()
  mientras(!es_vacia(procs)):
    sub = cima(sub_pendientes)
    desapila(sub_pendientes)
    let sub = dec_proc(Iden,Param,LDecs,Is) in
      emit desapilad(sub.nivel)
      recolecta_subs(LDecs)
      gen-cod(Ins)
      emit desactiva(sub.nivel,sub.tam)
      emit ir-ind()
    end let

```

```

gen-cod(si_decs()):
  recolecta_procs(LDecs)

```

```

gen-cod(no_dec()):
  skip

```

```

gen-cod(una_dec(Dec)):
  recolecta_procs(Dec)

```

```

gen-cod(muchas_decs(LDecs, Dec)):
  recolecta_procs(LDecs)
  recolecta_procs(Dec)

```

```

gen-cod(dec_var(T,Iden)):
  skip

```

```

gen-cod(dec_tipo(T,Iden)):
  skip

```

#### **4.6.2 Generación de código para declaraciones.**

```

gen-cod(dec_proc(Iden, PFormOpt, Bloq)):
  gen-cod(Bloq)
  emit desactiva(nivel, tam)
  emit ir_ind()

```

#### **4.6.3 Generación de código para instrucciones.**

```

gen-cod(si_ins(LIns)):
  gen-cod(LIns)

```

```
gen-cod(no_ins()):  
    skip
```

```
gen-cod(una_ins(Ins)) =  
    gen-cod(Ins)
```

```
gen-cod(muchas_ins(LIns, Ins)) =  
    gen-cod(LIns)  
    gen-cod(Ins)
```

```
gen-cod(ins_asig(Exp)):  
    gen-cod(Exp)  
    emit desecha()
```

```
gen-acc-val(Exp):  
    si es_desig(Exp):  
        v = emit fetch(r)  
    sino:  
        v = r
```

```
gen-cod(ins_if(Exp, Bloq)):  
    gen-cod(Exp)  
    gen-acc-val(Exp)  
    emit ir-f($.sig)  
    gen-cod(Bloq)
```

```
gen-cod(if_then_else(Exp, Bloq0, Bloq1)):  
    gen-cod(Exp)  
    gen-acc-val(Exp)  
    emit ir-f($.sig)  
    gen-cod(Bloq0)  
    emit ir-a($.sig)  
    gen-cod(Bloq1)
```

```
gen-cod(while(Exp, Bloq)):  
    gen-cod(Exp)  
    gen-acc-val(Exp)  
    emit ir-f($.sig)  
    gen-cod(Bloq)  
    emit ir-a($.prim)
```

```

gen-cod(read(Exp)):
    t = ref!(Exp.tipo)
    r = gen-cod(Exp)
    emit leer_entrada_<t>
    store(r, t)

```

```

gen-cod(write(Exp)):
    r = gen-cod(Exp)
    gen-acc-val(Exp)
    emite mostrar_<v>

```

```

gen-cod(ins_nl()):
    skip

```

```

gen-cod(new(Exp)):
    emite store(gen-cod(Exp), emite alloc(ref!(Exp.tipo)))

```

dealloc(d, ), si d = -1. Error de ejecución si d = -1

```

gen-cod(delete(Exp)):
    d = gen-cod(Exp)
    si d != -1:
        emite dealloc(d, ref!(E.tipo))
    sino:
        ERROR

```

```

gen-cod(ins_call(string, PRealOpt)):
    let id = string.vinculo, id.vinculo = dec_proc(_, PFormOpt , _) in
        emit activa($.vinculo.nivel, $.vinculo.tam, $.dir_sig)
        gen-paso-pf(PFormOpt, PRealOpt)
        emit ir-a($.vinculo.dir_inic)
    end let

```

```

gen-cod(ins_bloque(Bloque)):
    gen-cod(Bloq)

```

#### 4.6.4 Generación de código para expresiones

```

gen-cod(exp_litEntero(N)):
    emit apila_int(N)

```

```

gen-cod(exp_litReal(N)):
    emit apila_real(N)

```

```

gen-cod(exp_litBoolTrue()):
    emit apila_bool(True)

gen-cod(exp_litBoolFalse()):
    emit apila_bool(False)

gen-cod(exp_litCadena(N)):
    emit apila_string(N)

gen-cod(exp_identificador(Id)):
    if $.vinculo.nivel != 0:
        emit apilad($.vinculo.nivel)
        emit apila_int($.vinculo.dir)
        emit suma_int
        if $.vinculo == pform_ref(T, id)
            emit apila_ind
    sino:
        emit apila_int($.vinculo.dir)

gen-cod(exp_null()):
    emit apila_int(-1)))

gen-acc-val(Exp):
    if es_desig(ref!(Exp))
        emit apila_ind()

gen-cod(asig(Exp0,Exp1)):
    gen-cod(Exp0)
    gen-cod(Exp1)
    if ref!(Exp1.tipo) == int & ref!(Exp0.tipo) == real
        if es_desig(Exp1)
            emit apila_ind
            emit int2real
            emit desapila_ind
    else
        if es_desig(Exp1)
            emit mueve(ref!(Exp0.tipo).tam)
        else
            emit desapila_ind

```

```

gen-cod(suma(Exp0,Exp1)):
    t0 = ref!(Exp0.tipo)
    t1 = ref!(Exp1.tipo)
    if t0 == int & t1 == real
        gen-cod(Exp0)
        gen-acc-val(Exp0)
        emit int2real
        gen-cod(Exp1)
        gen-acc-val(Exp1)
        emit suma_real
    else
        gen-cod(Exp0)
        gen-acc-val(Exp0) //si es designador apila_ind sino nada
        gen-cod(Exp1)
        gen-acc-val(Exp1)

    if t0 == int & t1 == int
        emit suma_int
    else if t0 == real & t1 == real
        emit suma_real
    else if t0==real & t1 ==int
        emit int2real
        emit suma_real

```

```

gen-cod(resta(Exp0,Exp1)):
    t0 = ref!(Exp0.tipo)
    t1 = ref!(Exp1.tipo)
    if t0 == int & t1 == real
        gen-cod(Exp0)
        gen-acc-val(Exp0)
        emit int2real
        gen-cod(Exp1)
        gen-acc-val(Exp1)
        emit resta_real
    else
        gen-cod(Exp0)
        gen-acc-val(Exp0) //si es designador apila_ind sino nada
        gen-cod(Exp1)
        gen-acc-val(Exp1)

    if t0 == int & t1 == int
        emit resta_int
    else if t0 == real & t1 = real
        emit resta_real
    else if t0==real & t1 ==int
        emit int2real
        emit resta_real

```

```

gen-cod(mul(Exp0,Exp1)):
    t0 = ref!(Exp0.tipo)
    t1 = ref!(Exp1.tipo)
    if t0 == int & t1 == real
        gen-cod(Exp0)
        gen-acc-val(Exp0)
        emit int2real
        gen-cod(Exp1)
        gen-acc-val(Exp1)
        emit mul_real
    else
        gen-cod(Exp0)
        gen-acc-val(Exp0) //si es designador apila_ind sino nada
        gen-cod(Exp1)
        gen-acc-val(Exp1)

    if t0 == int & t1 == int
        emit mul_int
    else if t0 == real & t1 == real
        emit mul_real
    else if t0==real & t1 ==int
        emit int2real
        emit mul_real

```

```

gen-cod(div(Exp0,Exp1)):
    t0 = ref!(Exp0.tipo)
    t1 = ref!(Exp1.tipo)
    if t0 == int & t1 == real
        gen-cod(Exp0)
        gen-acc-val(Exp0)
        emit int2real
        gen-cod(Exp1)
        gen-acc-val(Exp1)
        emit div_real
    else
        gen-cod(Exp0)
        gen-acc-val(Exp0) //si es designador apila_ind sino nada
        gen-cod(Exp1)
        gen-acc-val(Exp1)

    if t0 == int & t1 == int
        emit div_int
    else if t0 == real & t1 = real
        emit div_real
    else if t0==real & t1 ==int
        emit int2real
        emit div_real

gen-cod(mod(Exp0,Exp1)):
    gen-cod(Exp0)
    gen-acc-val(Exp0)
    gen-cod(Exp1)
    gen-acc-val(Exp1)
    emit mod

gen-cod(and(Exp0,Exp1)):
    gen-cod(Exp0)
    gen-acc-val(Exp0)
    gen-cod(Exp1)
    gen-acc-val(Exp1)
    emit and

gen-cod(or(Exp0,Exp1)):
    gen-cod(Exp0)
    gen-acc-val(Exp0)
    gen-cod(Exp1)
    gen-acc-val(Exp1)

```



```

    emit or
gen-cod(mayor(Exp0,Exp1)):
    t0 = ref!(Exp0.tipo)
    t1 = ref!(Exp1.tipo)
    if t0 == int & t1 == real
        gen-cod(Exp0)
        gen-acc-val(Exp0)
        emit int2real
        gen-cod(Exp1)
        gen-acc-val(Exp1)
        emit mayor_real
    else
        gen-cod(Exp0)
        gen-acc-val(Exp0) //si es designador apila_ind sino nada
        gen-cod(Exp1)
        gen-acc-val(Exp1)
    if t0 == t1
        if t1 ==real
            emit mayor_real
        else if t1 == int
            emit mayor_int
        else if t1 == bool
            emit mayor_bool
        else if t1 == string
            emit mayor_string

    else if t0==real & t1 ==int
        emit int2real
        emit mayor_real

```

```

gen-cod(menor(Exp0,Exp1)):
    t0 = ref!(Exp0.tipo)
    t1 = ref!(Exp1.tipo)
    if t0 == int & t1 == real
        gen-cod(Exp0)
        gen-acc-val(Exp0)
        emit int2real
        gen-cod(Exp1)
        gen-acc-val(Exp1)
        emit menor_real
    else
        gen-cod(Exp0)
        gen-acc-val(Exp0) //si es designador apila_ind sino nada
        gen-cod(Exp1)
        gen-acc-val(Exp1)
    if t0 == t1
        if t1 ==real
            emit menor_real
        else if t1 == int
            emit menor_int
        else if t1 == bool
            emit menor_bool
        else if t1 == string
            emit menor_string
    else if t0==real & t1 ==int
        emit int2real
        emit menor_real

```

```

gen-cod(menorIguar(Exp0,Exp1)):
    t0 = ref!(Exp0.tipo)
    t1 = ref!(Exp1.tipo)
    if t0 == int & t1 == real
        gen-cod(Exp0)
        gen-acc-val(Exp0)
        emit int2real
        gen-cod(Exp1)
        gen-acc-val(Exp1)
        emit menorIguar_real
    else
        gen-cod(Exp0)
        gen-acc-val(Exp0) //si es designador apila_ind sino nada
        gen-cod(Exp1)
        gen-acc-val(Exp1)
    if t0 == t1
        if t1 ==real
            emit menorIguar_real
        else if t1 == int
            emit menorIguar_int
        else if t1 == bool
            emit menorIguar_bool
        else if t1 == string
            emit menorIguar_string
    else if t0==real & t1 ==int
        emit int2real
        emit menorIguar_real

```

```

gen-cod(mayorIgual(Exp0,Exp1)):
    t0 = ref!(Exp0.tipo)
    t1 = ref!(Exp1.tipo)
    if t0 == int & t1 == real
        gen-cod(Exp0)
        gen-acc-val(Exp0)
        emit int2real
        gen-cod(Exp1)
        gen-acc-val(Exp1)
        emit mayorIgual_real
    else
        gen-cod(Exp0)
        gen-acc-val(Exp0) //si es designador apila_ind sino nada
        gen-cod(Exp1)
        gen-acc-val(Exp1)
    if t0 == t1
        if t1 ==real
            emit mayorIgual_real
        else if t1 == int
            emit mayorIgual_int
        else if t1 == bool
            emit mayorIgual_bool
        else if t1 == string
            emit mayorIgual_string
    else if t0==real & t1 ==int
        emit int2real
        emit mayorIgual_real

```

```

gen-cod(desigual(Exp0,Exp1)):
    t0 = ref!(Exp0.tipo)
    t1 = ref!(Exp1.tipo)
    if t0 == int & t1 == real
        gen-cod(Exp0)
        gen-acc-val(Exp0)
        emit int2real
        gen-cod(Exp1)
        gen-acc-val(Exp1)
        emit desigual_real
    else
        gen-cod(Exp0)
        gen-acc-val(Exp0) gen-cod(Exp1)
        gen-acc-val(Exp1)
    if t0 == t1
        if t1 ==real
            emit desigual_real
        else if t1 == int
            emit desigual_int
        else if t1 == bool
            emit desigual_bool
        else if t1 == string
            emit desigual_string
        else if t1 == puntero
            emit desigual_puntero
        else if t1 == null
            emit desigual_null
    else if t0==real & t1 ==int
        emit int2real
        emit desigual_real
    else if t0 == puntero & t1 == null || t1 == null & t1 == puntero
        emit desigual_pn

```

```

gen-cod(igual(Exp0,Exp1)):
    t0 = ref!(Exp0.tipo)
    t1 = ref!(Exp1.tipo)
    if t0 == int & t1 == real
        gen-cod(Exp0)
        gen-acc-val(Exp0)
        emit int2real
        gen-cod(Exp1)
        gen-acc-val(Exp1)
        emit igual_real
    else
        gen-cod(Exp0)
        gen-acc-val(Exp0) //si es designador apila_ind sino nada
        gen-cod(Exp1)
        gen-acc-val(Exp1)
    if t0 == t1
        if t1 ==real
            emit igual_real
        else if t1 == int
            emit igual_int
        else if t1 == bool
            emit igual_bool
        else if t1 == string
            emit igual_string
        else if t1 == puntero
            emit igual_puntero
        else if t1 == null
            emit igual_null
    else if t0==real & t1 ==int
        emit int2real
        emit igual_real
    else if t0 == puntero & t1 == null || t1 == null & t1 == puntero
        emit igual_pn

```

```

gen-cod(neg(Exp0)):
    t0 = ref!(Exp0.tipo)
    gen-cod(Exp0)
    gen-acc-val(Exp0)
    if t0 == int
        emit neg_int
    else if t0 == real
        emit neg_real

```

```
gen-cod(not(Exp0)):
    gen-cod(Exp0)
    gen-acc-val(Exp0)
    emit not
```

```
gen-cod(acceso_array(Exp0,Exp1)):
    gen-cod(Exp0)
    gen-cod(Exp1)
    gen-acc-val(Exp1)
    sea ref!(Exp1.tipo) = array (Tipo, d)
        emit apila_int(Tipo.tam)
    emit mul_int
    emit suma_int
```

```
gen-cod(acceso_campo(Exp, Campo)):
    gen-cod(Exp)
    let ref!(Exp.tipo) = struct(LCampos)
        emit apila_int(desplazamiento(LCampos, Campo))
    emit suma_int
```

```
gen-cod(acceso_puntero(Exp)):
    gen-cod(Exp)
    emit apila_ind()
```