Práctica de Procesadores de Lenguaje
Segunda Parte

Fecha de entrega: **Viernes 28 de Mayo de 2010**

Número de grupo: 3
Componentes del Grupo : Ortiz Jaureguizar, Gonzalo
Sanjuán Redondo, Héctor
Tarancón Garijo, Rubén

Requisitos

El lenguaje a procesar

El lenguaje a procesar contiene todas las características fijadas para la primera entrega, más las siguientes

· **Secciones de declaraciones vacías:**
  · Se admiten secciones de declaraciones vacías (es decir, sin ninguna declaración).

· **Instrucciones de control**:
  · Instrucción *bloque*:
    - Formato: **{*I*0; *I*1; …; *I*n}**, donde cada I_i es una instrucción. El bloque debe contener, al menos, una instrucción.
    - Semántica operacional informal:
      · Ejecutar ***I*0**.
      · Ejecutar ***I*1**.
      …
      · Ejecutar ***I*n**.
  · Instrucción *if-then-else*.
    - Formato: **if** *Exp* **then** I_0 **else** I_1, o bien **if** *Exp* **then** *I* (la parte *else* es opcional).
    - La condición (*Exp*) es una expresión booleana. Tanto I_0 como I_1 como *I* son instrucciones.
    - La parte *else* siempre se corresponde con la parte *if* más cercana.
    - La semántica operacional informal de **if** *E* **then** I_0 **else** I_1 es:
      · Evaluar *E*
      · Si el resultado es *cierto*, ejecutar I_0
      · Si el resultado es *falso*, ejecutar I_1
    - La semántica operacional informal de **if** *E* **then** *I* es:
      · Evaluar *E*
      · Si el resultado es *cierto*, ejecutar *I*
      · Si el resultado es *falso*, no hacer nada
  · Instrucción *while*.
    - Formato: **while** *Exp* **do** *I*, con *Exp* una expresión booleana e *I* una instrucción.
    - Semántica operacional informal:
      · [*comienzo*] Evaluar *Exp*
      · Si el resultado es *cierto*
        - Ejecutar *I*
        - Volver a *comienzo*
      · Si el resultado es *falso*, no hacer nada
  · Instrucción *for*.
    - Formato: **for** *v*=*Exp*0 **to** *Exp*1 **do** *I*.
    - *Exp*0 y *Exp*1 son expresiones naturales o enteras, *v* es una variable cuyo tipo es compatible con el de *Exp*0 y *Exp*1, e *I* es una instrucción.
    - Semántica operacional informal:
      · Evaluar *Exp*0
      · Asignar a *v* el resultado
      · Evaluar *Exp*1 (sea *t* su valor)
      · [*comienzo*] Evaluar *v* £ *t*
      · si el resultado es *cierto*
        - Ejecutar *I*
        - *v* ¬ *v*+1
        - Volver a *comienzo*

· si el resultado es *falso*, no hacer nada

· **Evaluación en circuito corto de operadores booleanos**:
  · Los operadores booleanos **and** y **or** se evaluarán en circuito corto.
  · Semántica operacional informal de E_0 **and** E_1:
    - Evaluar E_0 (sea v_0 su valor)
    - si v_0 es *cierto*
      · Evaluar E_1 (sea v_1 su valor)
      · El valor de la expresión es v_1
    - si v_0 es *falso*, el valor de la expresión es v_0
  · Semántica operacional informal de E_0 **or** E_1:
    - Evaluar E_0 (sea v_0 su valor)
    - si v_0 es *falso*
      · Evaluar E_1 (sea v_1 su valor)
      · El valor de la expresión es v_1
    - si v_0 es *cierto*, el valor de la expresión es v_0

· **Tipos construidos**:
  · En las secciones de declaraciones se permitirá declarar tipos.
  · Cada declaración de tipo comenzará con la palabra reservada **tipo**. A continuación aparecerá un *identificador de tipo*, seguido del símbolo = y seguido de una *descripción de tipo*.
  · Las descripciones de tipo pueden ser:
    - Los tipos básicos contemplados en la primera entrega: boolean, character, natural, integer y float.
    - Otro identificador de tipo.
    - La descripción de un tipo *array*: **array** [*num*] **of** *DTipo*, con *num* un número natural, y *DTipo* una descripción de tipo (el *tipo base* del array; es decir, el tipo de los elementos).
    - La descripción de un tipo *registro*: **record** {C_0; …; C_n}.
      · Cada C_i es una descripción de campo. El formato de dicha descripción de campo es *DTipo NombreCampo*.
      · Debe haber, por lo menos, una descripción de campo.
      · No se permiten nombres de campo duplicados.
    - La descripción de un tipo *puntero*: **pointer** *DTipo*.
      · *DTipo* es la descripción del tipo base del puntero (el tipo de los objetos apuntados).
      · Se introduce el literal **null**. Este valor es compatible con cualquier tipo puntero, y denota un puntero que no apunta a ningún objeto.
      · Los punteros pueden compararse mediante = y =/=.
  · Como regla general, cuando, al describir un tipo, se utiliza un identificador de tipo, dicho identificador debe haber sido previamente declarado. La excepción a esta regla es en la descripción del tipo base en un tipo *puntero*: en este caso, se permite referir, además, cualquier otro identificador de tipo de los declarados en la misma sección de declaraciones.
  · Los tipos de las variables pueden ser descripciones de tipos arbitrarias.
  · Para decidir si un objeto puede asignarse a otro objeto, se llevará a cabo una comprobación estructural de sus tipos de acuerdo con las reglas siguientes:
    - Un objeto de tipo boolean únicamente puede asignarse a otro objeto de tipo boolean.
    - Un objeto de tipo character únicamente puede asignarse a

otro objeto de tipo character.
- Un objeto de tipo natural puede asignarse a otro objeto de tipo natural, a un objeto de tipo integer, o a un objeto de tipo float.
- Un objeto de tipo integer puede asignarse a otro objeto de tipo integer, o a un objeto de tipo float.
- Un objeto de tipo float puede asignarse únicamente a otro objeto de tipo float.
- Un objeto de tipo *array* puede asignarse únicamente a otro objeto de tipo *array*. Además, ambos objetos deben: (i) tener el mismo número de elementos, (ii) tener tipos base estructuralmente compatibles.
- Un objeto de tipo *registro* puede asignarse únicamente a otro objeto de tipo *registro*. Además:
  · Ambos registros deben tener exactamente el mismo número de campos.
  · Sea *n* el número de campos en ambos registros. Para cada *i* (1 £ *i* £ *n*), los campos que aparecen declarados en las posiciones *i*-esimas en los tipos de ambos registros han de tener el mismo nombre, y, además, tipos estructuralmente compatibles.
- Un objeto de tipo *puntero* puede asignarse únicamente a otro objeto de tipo *puntero* siempre y cuando los tipos base sean estructuralmente compatibles.
- Al considerar identificadores de tipo, se entenderá que dichos identificadores son equivalentes a sus descripciones de tipo asociadas.
- Si, durante el proceso de comprobación de compatibilidad estructural, se plantea más de una vez la compatibilidad estructural del mismo par de tipos, ambos tipos deben considerarse estructuralmente compatibles.

· A fin de acceder a los diferentes elementos de un objeto de tipo estructurado, pueden utilizarse *designadores*. Estos designadores se ajustan a la siguiente estructura:
- Una variable o parámetro formal (ver más adelante) es un designador. Su tipo será el tipo de la variable o parámetro.
- *d*[*Exp*], con *d* un designador de tipo *array* y *Exp* una expresión entera. El tipo de *d*[*Exp*] es el tipo base del array.
- *d*.*campo*, donde *d* es un designador de tipo *registro*, y *campo* es un campo de dicho registro.El tipo de *d*.*campo* es el tipo del campo.
- *d->*, con *d* un designador de tipo *puntero*. El tipo que resulta será el tipo base del puntero.

· Los designadores son una generalización de las variables en la versión anterior del lenguaje: pueden jugar tanto el papel de expresiones básicas, como aparecer en la parte izquierda de una asignación.
· Se añaden dos nuevos tipos de instrucciones:
- *Instrucción de reserva de memoria*. Comienza con la palabra reservada **new**, seguida de un designador de tipo *puntero*. Su efecto es crear un nuevo objeto del tipo base del puntero, e inicializar el puntero con la dirección de dicho objeto.
- *Instrucción de liberación de memoria*. Comienza con la palabra

reservada **dispose**, seguida de un designador de tipo *puntero*. Su efecto es liberar el espacio ocupado por el objeto apuntado por el puntero.

· **<u>Subprogramas</u>**:

· En las secciones de declaraciones se permitirá declarar procedimientos.

· Cada declaración de procedimiento constará de una *cabecera* y un *cuerpo*.

· Formato de la cabecera: **procedure** *nombreProcedimiento*(P_0, …, P_n)

- Cada P_i es un *parámetro formal*, que puede ser:
  · Por valor: *DTipo param*. *DTipo* es la descripción del tipo del parámetro, y *param* su nombre.
  · Por variable: **var** *DTipo param*.
- La lista de parámetros es opcional.

· Dos posibles clases de cuerpo:

- *Cuerpo definido*: **{**D_0; …; D_n & I_0; …; I_m**}**. Cada D_i es una declaración (se admite que no haya declaraciones). Cada I_j es una instrucción (ha de haber, al menos, una instrucción).
- *Cuerpo diferido*: palabra reservada **forward**.

· En una sección de declaraciones:

- Un mismo procedimiento puede tener, a lo sumo, dos declaraciones: una con **forward**, y otra con el cuerpo definido.
- En caso de que aparezca una declaración con **forward**, deberá aparecer obligatoriamente, y con posterioridad a la misma, una declaración con cuerpo definido.
- A todos los efectos, ambas declaraciones declaran un único procedimiento. La declaración con cuerpo **forward** sirve para permitir procedimientos mutuamente recursivos.

· No puede haber parámetros duplicados. Así mismo, los nombres de los parámetros deben ser diferentes de los nombres de variables, tipos y procedimientos declarados en la sección de declaraciones del procedimiento.

· El nombre del procedimiento no debe coincidir con el nombre de ninguno de los parámetros. Tampoco debe coincidir con ningún nombre de variable, tipo o procedimiento declarado en su sección de declaraciones.

· El lenguaje adoptará un convenio de *ámbito léxico* para asociar el uso de los nombres con sus declaraciones. Cada ocurrencia de un identificador en la descripción de un tipo o en una instrucción deberá corresponderse con una declaración de variable, tipo o procedimiento. Dicha declaración se buscará primeramente en la sección de declaraciones del bloque en el que está la ocurrencia. Si no aparece allí, se buscará en el bloque padre, … y así sucesivamente.

· Se introduce un nuevo tipo de instrucción: instrucción de *invocación de procedimiento*.

- Formato: *nombreProcedimiento* (E_0, …, E_n).
- Cada parámetro real E_i es una expresión.
- Una expresión se dice que tiene *modo var* si es, bien un designador, bien una expresión de la forma (*E*), siendo *E* una expresión en *modo var*.
- El número y el tipo de los parámetros reales deberá coincidir con el número y el tipo de los correspondientes parámetros formales.
- Así mismo, cuando el modo del parámetro formal es *var*, el

modo del correspondiente parámetro real también debe ser *var*, y el paso de parámetros se realizará *por variable*.

· Los procedimientos pueden invocarse recursivamente.

1. Definición léxica del lenguaje

Especificación formal del léxico del lenguaje (a veces llamado microsintaxis) utilizando definiciones regulares.

```
& ::= &
id ::= [a-zA-Z][a-zA-Z0-9]*
: ::= :
:= ::= :=
tipos ::= boolean | character | natural | integer | float
; ::= ;
< ::= <
> ::= >
<= ::= <=
>= ::= >=
= ::=  =
=/= ::= =/=
+ ::= \+
- ::= \-
* ::= \*
 / ::= /
% ::= %
and ::= and
or ::= or
not ::= not
<< ::= <<
>> ::= >>
(nat) ::= \(nat\)
(int) ::= \(int\)
(char) ::= \(char\)
(float) ::= \(float\)
litNat ::= ([1-9][0-9]*|0)
litFlo ::= ([1-9][0-9]*|0)(
                                \.0|
                                \.([0-9]*[1-9])|
                                \.([0-9]*[1-9])(e|E)-?([1-9][0-9]*|0)|
                                (e|E)-?[1-9][0-9]*|0
        )
litTrue ::= true
litFalse ::= false
litCha ::= '[a-zA-Z0-9]'
| ::= \|
( ::= \(
) ::= \)
comentario ::= #.*\\n
in := in
out := out
if ::= if
then ::= then
else ::= else
{ ::= \{
} ::= \}
while ::= while
do ::= do
for ::= for
to ::= to
tipo ::= tipo
array ::= array
of ::= of
record ::= record
pointer ::= pointer
null ::= null
```

^ ::= \^
new ::= new
dispose ::= dispose
procedure ::= procedure
var ::= var
forward ::= forward
- > ::= \- \>
. ::= \.
[::= \[
] ::= \]
, ::= \,

2. Definición sintáctica del lenguaje

2.1. Descripción de los operadores

Los operadores del lenguaje son los siguientes. Todos asocian a derechas, excepto los de nivel 3:

	Operador	**Aridad:**
Nivel 0 – menor prioridad		
	<	2
	>	2
	<=	2
	>=	2
	=	2
	=/=	2
Nivel 1		
	+	2
	-	2
	or	2
Nivel 2		
	*	2
	/	2
	%	2
	and	2
Nivel 3		
	>>	2
	<<	2
	Nivel 4 – mayor prioridad	
	not	1
	-	1
	(float)	1
	(int)	1
	(nat)	1
	(char)	1
	| |	1

2.2. Formalización de la sintaxis

Programa → Declaraciones & Instrucciones

Declaraciones → Declaraciones ; Declaracion
Declaraciones → Declaración
Declaracion → DeclaracionTipo
Declaracion → DeclaracionVariable

Declaracion → DeclaracionProcedimiento

DeclaracionTipo → tipo id = Tipo
DeclaraciónVariable → id : Tipo
DeclaracionProcedimiento → procedure id FParametros forward
DeclaracionProcedimiento → procedure id FParametros Bloque

Bloque → Declaraciones & Instrucciones
Bloque → Instrucciones
FParametros → (LFParametros)
FParametros → λ
LFParametros → LFParametros , FParametro
LFParametros → FParametro
FParametro → var Tipo id
FParametro → Tipo id

Tipo → Boolean
Tipo → character
Tipo → Float
Tipo → Natural
Tipo → id
Tipo → array [Expresion] of Tipo
Tipo → record { Campos }
Tipo → pointer Tipo

Campos → Campos ';' Campo
Campos → Campo
Campo → id : Tipo

Instrucciones → Instrucción ; Instrucciones
Instrucciones → Instrucción
Instrucción → InsProcedimiento
Instrucción → InsLectura
Instrucción → InsEscritura
Instrucción → InsAsignación
Instrucción → InsCompuesta
Instrucción → InsIf
Instrucción → InsWhile
Instrucción → InsFor
Instrucción → InsNew
Instrucción → InsDis

InsProcedimiento → id AParametros
AParametros → (LAParametros)
AParametros → λ
LAParametros → LAParametros , Expresion
LAParametros → Expresion

InsLectura → in(id)
InsEscritura → out(Expresion)
InsAsignación → Mem := Expresión

InsCompuesta → { Instrucciones }

InsIf → if Expresion then Instrucción Pelse
PElse → else Instrucción
PElse → λ

InsWhile → while Expresion do Instrucción

InsFor → for id=Expresion to Expresion do Instruccion

InsNew → new Mem

InsDis → dispose Mem

Mem → id
Mem → Mem->
Mem → Mem[Exp]
Mem → Mem.id

Expresión → ExpresiónNiv1 OpNiv0 ExpresiónNiv1
Expresión → ExpresiónNiv1

ExpresiónNiv1 → ExpresiónNiv1 OpNiv1 ExpresiónNiv2
ExpresiónNiv1 → ExpresiónNiv2

ExpresiónNiv2 → ExpresiónNiv2 OpNiv2 ExpresiónNiv3
ExpresiónNiv2 → ExpresiónNiv3

ExpresiónNiv3 → ExpresionNiv4 OpNiv3 ExpresiónNiv3
ExpresiónNiv3 → ExpresiónNiv4

ExpresiónNiv4 → OpNiv4 ExpresiónNiv4
ExpresiónNiv4 → | Expresión |
ExpresiónNiv4 → (Expresión)
ExpresiónNiv4 → Literal
ExpresionNiv4 → Mem

Literal → litNat
Literal → litFlo
Literal → litTrue
Literal → litFalse
Literal → litchi
Literal → litNull

OpNiv0 → <
OpNiv0 → >
OpNiv0 → <=
OpNiv0 → >=
OpNiv0 → =
OpNiv0 → =/=

OpNiv1 → +
OpNiv1 → -

```
OpNiv1 → or

OpNiv2 → *
OpNiv2 → /
OpNiv2 → %
OpNiv2 → and

OpNiv3 → >>
OpNiv3 → <<

OpNiv4 → not
OpNiv4 → -
OpNiv4 → (float)
OpNiv4 → (int)
OpNiv4 → (nat)
OpNiv4 → (char)
```

3. Estructura y construcción de la tabla de símbolos

Nuestro lenguaje está estructurado en bloques, por tanto nos es necesario una gestión de ámbitos para controlar si en un determinado momento una variable está disponible o no, ya que no todas las variables se declaran al inicio del programa, sino que pueden ser declaradas en procedimientos. Por este motivo hemos utilizado una pila de tabla de símbolos.

La información que necesitamos almacenar en cada tabla de símbolos es la siguiente:

El símbolo que identifica la variable

Una colección de propiedades asociadas a la variable, como clase, tipo, dir, nivel

Tabla de Símbolos	
“edad”	<clase:var, tipo: <t:integer>, dir:0, nivel:1>
“dni”	<clase:pvar, tipo: <t:float>, dir:1, nivel:1>
…	...

El tipo en este caso es bastante complejo y dependiendo de si es de un tipo u otro deberá llevar unos u otros parámetros.

Por ejemplo los booleanos son <t:boolean, tam:1>, los array son <t:array, nelem, tbase, tam>, etc...

3.1. Estructura de la tabla de símbolos

Descripción de las operaciones de la tabla de símbolos, definiendo la cabecera de dichas operaciones, así como describiendo informalmente su cometido, incluyendo el propósito de cada uno de sus parámetros.

creaGestor():GestorTS

El resultado es un gestor vacio.

creaTS():GestorTS

Añade una nueva tabla de símbolos al gestor

inserta(ts:GestorTS, id:String, props: PropsTs):GestorTs

El resultado es el gestor resultante de añadir id y sus propiedades a la tabla de símbolos de la

pila.

existe(ts: GestorTS, id:String):Boolean

El resultado es true si id aparece en alguna de las tablas de símbolos de la pila, false en caso contrario.

getProps(ts:GestorTS id:String):PropsTs

Te devuelve las propiedades del id que pasas como parámetro.

3.2. Construcción de la tabla de símbolos

3.2.1 Funciones semánticas

3.2.2 Atributos semánticos

Categoria Programa

ts: sintetizado. Es la tabla de símbolos que se va construyendo. Se sintetiza de las declaraciones.

tsh: heredado. Es la tabla de símbolos que se va pasando a cada una de las declaraciones de variables, tipo y procedimientos para ver si hay algo ya declarado. Se hereda a las declaraciones, parámetros, tipos.

Categoria Declaraciones

props: sintetizado. Es toda la información que se guarda de cada entrada de la tabla de simbolos. Se almacena el modo, el tipo, la dirección y el nivel.

Categoria FParametros

parámetros: sintetizado. Existe un atributo por procedimiento y guarda la información relativa a los parámetros (modo y tipo). Este atributo se sintetiza de LFParametros y LFParametro

Categoria Campo

campo: sintetizado. Guarda la información (id , tipo) de cada "campo" de un registro.

3.2.3 Gramática de atributos

Gramática de atributos que formaliza la construcción de la tabla de símbolos.

```
Programa → Declaraciones & Instrucciones
        Programa.ts          = Declaraciones.ts
        Instrucciones.tsh    = Declaraciones.ts
        Declaraciones.tsh    = creaTS()
        Declaraciones.nh     = 0

Declaraciones → Declaraciones ; Declaracion
        Declaraciones1.tsh   = Declaraciones0.tsh
        Declaracion.tsh      = Declaraciones1.ts
        Declaraciones0.ts    =
                inserta(Declaraciones1.ts, Declaracion.id, Declaracion.props)
        Declaraciones1.nh    = Declaraciones0.nh
        Declaracion.nh               = Declaraciones0.nh
        Declaracion0.n               = max(Declaraciones1.n,Declaracion.n)

Declaraciones → Declaración
        Declaracion.tsh      = Declaraciones.tsh
```

```
        Declaracion.nh              = Declaraciones.nh
        Declaraciones.ts    =
                inserta(Declaraciones.tsh, Declaracion.id, Declaracion.props)
        Declaraciones.n             = Declaracion.n

Declaracion → DeclaracionTipo
        Declaracion.id              = DeclaracionTipo.id
        Declaracion.props   = DeclaracionTipo.props
        DeclaracionTipo.tsh = Declaracion.tsh
        DeclaracionTipo.nh  = Declaracion.nh

Declaracion → DeclaracionVariable
        Declaracion.id                      = DeclaracionVariable.id
        Declaracion.props           = DeclaracionVariable.props
        DeclaracionVariable.tsh= Declaracion.tsh
        DeclaracionVariable.nh = Declaracion.nh

Declaracion → DeclaracionProcedimiento
        Declaracion.id                                  = DeclaracionProcedimiento.id
        Declaracion.props                       = DeclaracionProcedimiento.props
        DeclaracionProcedimiento.tsh    = Declaracion.tsh
        DeclaracionProcedimiento.nh     = Declaracion.nh

DeclaracionTipo →  tipo id = Tipo
        DeclaracionTipo.id          = id.lex
        DeclaracionTipo.props       =
                <clase:tipo, tipo:DeclaracionTipo.tipo, nivel: DeclaracionTipo.nh>
        DeclaracionTipo.tipo        = Tipo.tipo
        Tipo.tsh                            = DeclaracionTipo.tsh

DeclaracióńVarible → Tipo id
        DeclaracionVariable.id      = id.lex
        DeclaracionVariable.props   =
                <clase:var, tipo:DeclaracionVariable.tipo,
                        nivel:DeclaracionVariable.nh>
        DeclaracionVariable.tipo    = Tipo.tipo
        Tipo.tsh                                = DeclaracionVariable.tsh

DeclaracionProcedimiento →  procedure id Fparametros Forward
        DeclaracionProcedimiento.id     = id.lex
        DeclaracionProcedimiento.props  =
                <clase:forward, tipo: <t:proc, params: Fparametros.parametros>,
                nivel: DeclaracionProcedimiento.nh + 1>
        FParams.tsh                             = creaTS(DecProcedimiento.tsh)
    FParametros.nh                              = DeclaracionProcedimiento.nh +1

DeclaracionProcedimiento →  procedure id Fparametros Bloque
        DeclaracionProcedimiento.id     = id.lex
        DeclaracionProcedimiento.props  =
                <clase:proc, tipo: <t:proc, params: Fparametros.parametros>,
                nivel: DeclaracionProcedimiento.nh + 1>
        FParams.tsh                             = creaTS(DecProcedimiento.tsh)
    Bloque.tsh                                          =
```

```
      inserta(FParametros.ts, DeclaracionProcedimiento.id,  DeclaracionProcedimiento.props)
    FParametros.nh                              =
      Bloque.nh = DeclaracionProcedimiento.nh +1
```

Bloque → Declaraciones & Instrucciones

```
      Declaraciones.tsh   = Bloque.tsh
      Instrucciones.tsh   = Declaraciones.ts
      Bloque.ts                 = Declaraciones.ts
      Declaraciones.nh    = Bloque.nh
```

Bloque → Instrucciones

```
      Instrucciones.tsh   =  Bloque.tsh
```

FParametros → (LFParametros)

```
      LFParametros.tsh          = FParametros.tsh
      FParametros.ts                  = LFParametros.ts
      LFPatametros.nh           = FParametros.nh
      FParametros.parametros = LFParametros.parametros
```

FParametros → λ

```
      FParametros.ts                  = FParametros.tsh
      FParametros.parametros = []
```

LFParametros → LFParametros , FParametro

```
      LFParametros1.tsh         = LFParametros0.tsh
      LFParametros0.ts          =
            inserta (LFParametros1.ts, FParametro.id, FParametro.props)
      FParametro.nh                   = LFParametros1.nh = LFParametros0.nh
      LFParametros0.parametros =
            FParametros1.parametros ++ FParametro.parametro
```

LFParametros → FParametro

```
      LFParametros.ts           =
            inserta (LFParametros.tsh, FParametro.id, FParametro.props
      FParametro.nh                   = LFParametros.nh
      LFParametros.parametros= Fparametro.parametro
```

Fparametro → var Tipo id

```
      FParametro.id                   = id.lex
      FParametro.props          =
            <clase: pvar, tipo: Tipo.tipo, nivel: FParametro.nh>
      FParametro.parametro      = <modo: variable, tipo: Tipo.tipo>
```

Fparametro → Tipo id

```
      FParametro.id                   = id.lex
      FParametro.props          =
            <clase: var, tipo: Tipo.tipo, nivel: FParametro.nh>
      FParametro.parametro      = <modo: valor, tipo: Tipo.tipo>
```

Tipo → id

```
      Tipo.tipo    = <t:ref, id:id.lex>
```

Tipo → Boolean

```
      Tipo.tipo    = <t:bool>
```

Tipo → character

```
      Tipo.tipo    = <t:char>
```

Tipo → Float

```
      Tipo.tipo    = <t:float>
```

```
Tipo → Natural
       Tipo.tipo     = <t:nat>

Tipo → array [num] of Tipo
       Tipo0.tipo    = <t:array, nelems:num.lex, tbase:Tipo1.tipo>
       Tipo1.tsh     = Tipo0.tsh

Tipo → record { Campos }
       Tipo.tipo     = <t:reg, campos:Campos.campos>
       Campos.tsh    = Tipo.tsh

Tipo → pointer Tipo
       Tipo0.tipo    = <t:puntero, tbase:Tipo1.tipo>
       Tipo1.tsh     = Tipo0.tsh

Campos → Campos ';' Campo
       Campos0.campos       = Campos1.campos ++ Campo.campo
       Campo.tsh            = Campos1.tsh = Campos0.tsh

Campos → Campo
       Campos.campos        = [Campo.campo]
       Campo.tsh            = Campos.tsh

Campo → Tipo id
       Campo.campo   = <id:id.lex, tipo :Tipo.tipo>
     Tipo.tsh               = Campo.tsh
```

4. Especificación de las restricciones contextuales

Las restricciones contextuales relativas a esta práctica tienen que ver principalmente con que el tipo de las expresiones estén bien definidos y coincidan con el que permiten los operadores con las que van asociadas. Las restantes tienen que ver con la declaración de las variables y las instrucciones. Para las primeras he definido un atributo “tipo” y para las segundas un atributo “error”. Estos dos atributos tienen su enlace, ya que un posible valor del atributo “tipo” es error.

Las restricciones que tienen que ver con la declaración de variables tipos y procedimientos son:

- Solo se puede definir una variable o un tipo con el mismo nombre. En el caso de los procedimientos puede haber dos definiciones de procedimientos con el mismo nombre y el mismo nivel, en ese caso uno será con cuerpo definido y el otro con cuerpo diferido.

Las restricciones contextuales de la instrucción if (if Exp then I0 else I1), while (while Exp do I) son:

- La condición Exp de if y while tiene que ser una expresión booleana

Las restricciones contextuales del la instrucción for(for v=Exp0 to Exp1do I):

- Exp0 y Exp1 tienen que ser expresiones naturales o enteras y v es una variable cuyo tipo es compatible con el de Exp0 y Exp1.

La asignación lleva consigo muchas restricciones contextuales, dependiendo de cual sea el tipo al que queramos asignar:

- Un objeto de tipo boolean únicamente puede asignarse a otro objeto de tipo boolean.
- Un objeto de tipo character únicamente puede asignarse a otro objeto de tipo character.
- Un objeto de tipo natural puede asignarse a otro objeto de tipo natural, a un objeto de tipo integer, o a un objeto de tipo float.
- Un objeto de tipo integer puede asignarse a otro objeto de tipo integer, o a un objeto de tipo float.
- Un objeto de tipo float puede asignarse únicamente a otro objeto de tipo float.
- Un objeto de tipo *array* puede asignarse únicamente a otro objeto de tipo *array*. Además, ambos objetos deben: (i) tener el mismo número de elementos, (ii) tener tipos base estructuralmente compatibles.
- Un objeto de tipo *registro* puede asignarse únicamente a otro objeto de tipo *registro*. Además:
- Ambos registros deben tener exactamente el mismo número de campos.
- Sea *n* el número de campos en ambos registros. Para cada *i* (1 £ *i* £ *n*), los campos que aparecen declarados en las posiciones *i*-esimas en los tipos de ambos registros han de tener el mismo nombre, y, además, tipos estructuralmente compatibles.
- Un objeto de tipo *puntero* puede asignarse únicamente a otro objeto de tipo *puntero* siempre y cuando los tipos base sean estructuralmente compatibles.
- Al considerar identificadores de tipo, se entenderá que dichos identificadores son equivalentes a sus descripciones de tipo asociadas.
- Si, durante el proceso de comprobación de compatibilidad estructural, se plantea más de una vez la compatibilidad estructural del mismo par de tipos, ambos tipos deben considerarse estructuralmente compatibles.

Las restricciones las instrucciones que llaman a procedimientos son:
- Además de que el procedimiento ha tenido que ser declarado, los parámetros asociados a esta instrucción deben coincidir en tipo y numero a los de la declaración del procedimiento.

Las restricciones contextuales de las de las expresiones son las siguientes:
- Los operadores <, >, <=, >=, =/= son capaces de comparar entre si valores numéricos (natural con natural, natural con entero, natural con real, entero con natural, entero con entero, entero con real, real con natural con entero, real con real) caracteres y booleanos.
- Los operadores + y – operan solo sobre valores numéricos. El resultado será un valor real (siempre y cuando alguno de los operadores sea real), un valor entero (siempre y cuando alguno del os operandos sea entero y no haya operando reales) o un natural (solo cuando los dos operandos sean naturales).
- El operador **or** opera únicamente sobre valores booleanos.
- Los operadores *****y / operan solo sobre valores numéricos. El resultado será un valor real (siempre y cuando alguno de los operandos sea real), un valor entero (siempre y cuando alguno de los operandos sea entero y no haya operandos reales) o un natural (solo cuando los dos operandos sea naturales) o un natural (solo cuando los dos operandos sean naturales).
- En el operador **%** no opera sobre valores reales, caracteres y booleanos. El primer operando puede ser entero o natural, pero el segundo operando solo puede ser natural. El tipo del resultado será el mismo que el del primer operando.
- El operador **and** opera solo sobre valores booleanos.
- Los operadores <<y >>operan únicamente sobre valores naturales.
- El operador **not** opera solo sobre valores booleanos.
- El operador –opera solo sobre valores numéricos y el resultado es su operando cambiado de signo. Este resultado será real si el tipo del operando es real, y entero en otro caso.
- El operador **(float)** admite expresiones reales, enteras, de naturales y de caracteres. En todas

el resultado es real.

- El operador **(int)** admite expresiones reales, enteras, de naturales y de caracteres. En todas el resultado es entero.
- El operador **(nat)** admite solo expresiones de naturales y de caracteres y en ambas el resultado es natural.
- El operador **(char)** admite solo expresiones de naturales y de caracteres y en ambas el resultado es caracter.
- El operador **valor absoluto** solo admite expresiones numéricas y el resultado es real y la expresión es real o natural si la expresión que evalúa es natural o entera.

4.1. Funciones semánticas

Descripción, si procede, de las funciones semánticas adicionales utilizadas en la especificación. Para cada función debe indicarse explícitamente su cabecera, así como informalmente su cometido, incluyendo el propósito de cada uno de sus parámetros.
Esta sección puede dejarse vacía si no se van a usar funciones semánticas adicionales.

```
refErronea(GestorTs gestor, TipoTs tipo):boolean
        devuelve tipo=ref /\ not existeID(gestor,tipo.id)

compatibles(TipoTs tipo1, TipoTs tipo2, GestorTs gestor):boolean
        si (tipo1 = natural)
                si (tipo2=natural) devuelve true
        si no si (tipo1 = integer)
                si(tipo2 = natural || tipo2 = integer) devuelve true
        si no si (tipo1 = float)
                si(tipo2 = natural || tipo2 = integer || tipo2 = float) devuelve true
        si no si ((tipo1 = boolean && tipo2 =boolean) || (tipo1 = carácter || tipo2 = carácter)
                        devuelve true
        si no si tipo1 = ref devuelve compatibles2(ts[tipo1.id].tipo,tipo2)
        si no si tipo2 = ref devuelve compatibles2(tipo1,ts[tipo2.id].tipo)
        si no si tipo1=tipo2=array Ù tipo1.nelems=tipo2.nelems
                devuelve compatibles2(tipo1.tbase,tipo2.tbase)
        si no si tipo1.t = tipo2.t = rec Ù |tipo1.campos| = |tipo2.campos|
                para i=1 hasta |tipo1.campos| hacer
                        si ¬compatibles2(tipo1.campos[i].tipo,tipo2.campos[i].tipo)
                                devolver false
                        fsi
                fpara
                devolver true
        si no si e1.t = e2.t = puntero
                devuelve compatible2(e1.tbase,e2.tbase)
        si no devuelve false

ref(TipoTs tipo, GestorTs gestor):TipoTs
        si tipo = ref entonces
                si existeID(ts,tipo.id)
                        devuelve ref(gestor[tipo.id].tipo , gestor)
                si no devuelve error
        si no devuelve tipo

esCompatibleConTipoBasico(TipoTs tipo, GestorTs gestor):boolean
        devuelve        compatible(tipo1,boolean,gestor) ||
                        compatible(tipo1,character,gestor) ||
                        compatible(tipo1,float,gestor) ||
                        compatible(tipo1,integer,gestor) ||
```

compatible(tipo1,natural,gestor)

existeCampo(ArrayList<Campo> campos, String id):boolean

```
para i=0 hasta |campos| hacer
    si (campos[i] = id)
        devuelve true;
devuelve false
```

4.2. Atributos semánticos

Para cada categoría sintáctica relevante en este procesamiento deben enumerarse sus atributos semánticos, indicando si son heredados o sintetizados, y describiendo informalmente su propósito.

Categoría Programa:

error: sintetizado. Sirve para indicar si hay errores contextuales del programa.

pend: sintetizado. Sirve para ver si en alguna declaración de variable se ha utilizado un tipo no definido anteriormente.

forward: sintetizado. Sirve para comprobar que todas las llamadas a procedimientos se han realizado con procedimientos declarados.

Categoría Declaraciones:

error: sintetizado. Indica si hay errores en las declaraciones.

pend: sintetizado. Contiene las declaraciones de variable cuyo tipo no ha sido declarado.

forward: sintetizado. Contiene las llamadas a procedimientos cuya declaración no ha sido realizada. La información viene de DeclaracionesVariable, DeclaracionesTipo y DeclaracionesProcedimiento.

forwardh: heredado. Contiene la misma información, pero en este caso se le pasa a las instrucciones de los procedimientos para que si alguna de ellas es una llamada a procedimiento, que tenga la información.

Categoria Bloque;

error: sintetizado. Indica si hay algún error en las declaraciones e instrucciones del procedimiento.

pend: sintetizado. Indica las declaraciones de variable cuyo tipo no ha sido definido en el procedimiento.

Categoría FParametros:

error: sintetizado. Indica los errores que hay en la declaración de los parámetros de un procedimiento. Este error es sintetizado de LFParametros y LFParametro.

Categoría Tipo:

error: sintetizado. Muestra si hay error en algunos de los tipos existentes.

Categoria Campo:

error: sintetizado. Indica si hay error en alguno de los campos de un registro.

Categoría Instrucciones:

error: sintetizado. Indica si hay errores en la sección de instrucciones.

tsh: heredado. Es la tabla de símbolos que proviene de las declaraciones y que heredan las categorías sintácticas Instrucciones, Instruccion, InsLectura, InsEscritura, InsAsignacion, InsFor, InsWhile, InsIf, InsCompuesta, InsNew, InsDel, AParametros, LAParametros, Expresion,

ExpresionNiv1, ExpresionNiv2, ExpresionNiv3 y ExpresionNiv4, Mem.

Categoría AParametros:

fparametrosh: heredado. Coge esta estructura de la tabla de símbolos, es una lista que guarda el modo y el tipo de cada parámetro. Este parámetro se pasará a la categoría sintáctica LAParametros.

nparametrosh: hereadado. Al igual que el anterior coge la información de la tabla de simbolos y guarda el numero de parámetros que debe tener el procedimiento. Este parámetro se le pasará a la categoría sintáctica LAParametros.

Categoría Expresión:

tipo: sintetizado. Contiene el tipo de la expresión y, en su caso, error si existe un error de tipos.

modo: sintetizado. Indica si se trata de una expresión por variable o por valor. Este valor viene sintetizado de las categorías ExpresionNiv1, ExpresionNiv2, ExpresionNiv3, ExpresionNiv4.

Categoria Mem:

tipo: sintetizado. Indica el tipo de Mem, si es de tipo error es que la expresión está mal definicida.

Categorías opNivX:

op: sintetizado. Contiene el tipo de operación.

4.4. Gramática de atributos

Gramática de atributos que formaliza la comprobación de las restricciones contextuales.

Programa → Declaraciones & Instrucciones

```
Programa.error      =
        Declaraciones.error v
        Instrucciones.error v
        Declaraciones.pend != ∅ v
        Declaraciones.forward !=  ∅
```

Declaraciones → Declaraciones ; Declaracion

```
Declaraciones0.error       =
        Declaraciones1.error v
        Declaracion.error v
                (existeID(Declaraciones1.ts,Declaracion.id)  ∧
                Declaraciones1.ts[Declaracion.id].nivel=Declaraciones0.nh)
Declaraciones0.pend =
        Declaraciones1.pend unión
        Declaracion.pend -
        (si  (Declaracion.props.clase = tipo)
                {Declaracion.id}
        sino ∅)
Declaraciones0.forward =   Declaraciones1.forward U
                           Declaracion.forward -
                                   si  (Declaracion.props.clase=proc)  {Declaracion.id}
```

Declaraciones → Declaracion

```
Declaraciones.error =
```

```
            Declaracion.error v
                  (existeID(Declaraciones.tsh,Declaracion.id)  ∧
                        Declaraciones.tsh[Declaracion.id].nivel = Declaraciones.nh)
      Declaraciones.pend  =
            Declaracion.pend -
            si (Declaracion.props.clase = tipo) {Declaracion.id}
      Declaraciones.forward = Declaracion.forward

Declaracion → DeclaracionTipo
      Declaracion.error   = DeclaracionTipo.error
      Declaracion.pend    = DeclaracionTipo.pend
      Declaracion.forward = {}


Declaracion → DeclaracionVariable
      Declaracion.error   = DeclaracionVariable.error
      Declaracion.pend    = DeclaracionVariable.pend
      Declaracion.forward = {}

Declaracion → DeclaracionProcedimiento
      DeclaracionProcedimiento.forwardh = Declaracion.forwardh
      Declaracion.error   = DeclaracionProcedimiento.error
      Declaracion.pend    = DeclaracionProcedimiento.pend
      Declaracion.forward = DeclaracionProcedimiento.forward

DeclaracionTipo → tipo id = Tipo
      DeclaracionTipo.error      =
            Tipo.error v
            existeID(DeclaracionTipo.tsh, id.lex) v
            ¬existeRef(DeclaracionTipo.tsh,Tipo.tipo)
      DeclaracionTipo.pend       = Tipo.pend

DeclaracióńVarible → Tipo id
      DeclaracionVariable.error  =
            Tipo.error v
            existeID(DeclaracionVariable.tsh, id.lex) v
      ¬existeRef(DeclaracionVariable.tsh,Tipo.tipo)
      DeclaracionVariable.pend   = Tipo.pend

DeclaracionProcedimiento → procedure id FParametros forward
      DeclaracionProcedimiento.error    =
            FParametros.error v
            (existeID(FParametros.ts, id.lex)  ∧
                  Fparametros.ts[id.lex].nivel=DeclaracionProcedimiento.nh+1)
      DeclaracionProcedimiento.pend = □
      DeclaracionProcedimiento.forward = {id}

DeclaracionProcedimiento → procedure id FParametros Bloque
      DeclaracionProcedimiento.error    =
            FParametros.error v
            Bloque.error v
            (existeID(FParametros.ts, id.lex)  ∧
                  Fparametros.ts[id.lex].nivel=DeclaracionProcedimiento.nh+1
```

```
        ^FParametros.ts[id.lex].clase != forward)
        DeclaracionProcedimiento.pend = Bloque.pend
        DeclaracionProcedimiento.forward = {}

Bloque → Declaraciones & Instrucciones
        Bloque.error  = Declaraciones.error v Instrucciones.error v
        Declaraciones.forward != {} v Declaraciones.pend !={}
        Bloque.pend   = Declaraciones.pend

Bloque → Instrucciones
        Bloque.error  = Instrucciones.error

FParametros → (LFParametros)
        FParametros.error    = LFParametros.error

FParametros → λ
        FParametros.error    = false

LFParametros → LFParametros , FParametro
        LFParametros0.error =
                LFParametros1.error v
                Fparametro.error v
        (existeID(LFParametros1.ts,FParametro.id)  ∧
        LFParametro1.ts[FParametro.id].nivel = LFParametros0.nh)

LFParametros → Fparametro
        LFParametros.error  =
                Fparametro.error v
        (existeID(LFParametros.ts,FParametro.id)  ∧
        LFParametros.ts[FParametro.id].nivel = LFParametros.nh)

FParametro → var Tipo id
        Fparametro.error           = Tipo.error

FParametro → Tipo id
        Fparametro.error    = Tipo.error

Tipo → id
        Tipo.error    =     si existeID(Tipo.tsh,id.lex)
                                    Tipo.tsh[id.lex].clase != tipo
                            sino
                                    false
        Tipo.pend     =     si (¬existeID(Tipo.tsh,id.lex))
                                    {id.lex}
                            sino
                                    ∅
Tipo → Boolean
        Tipo.error    = false
        Tipo.pend     = ∅

Tipo → character
        Tipo.error    = false
        Tipo.pend     = ∅
```

Tipo → Float

Tipo.error = false
Tipo.pend = ∅

Tipo → Natural

Tipo.error = false
Tipo.pend = ∅

Tipo → array [Expresion] of Tipo

$Tipo_0$.error =
Expresion.tipo.t != natural v
$Tipo_1$.error v
¬existeRef ($Tipo_0$.tsh , $Tipo_1$.tipo)
$Tipo_0$.pend = $Tipo_1$.pend

Tipo → record { Campos }

Tipo.error = Campos.error
Tipo.pend = Campos.pend

Tipo → pointer Tipo

$Tipo_0$.error = $Tipo_1$.error
$Tipo_0$.pend = $Tipo_1$.pend

Campos → Campos ';' Campo

$Campos_0$.error =
$Campos_1$.error v
existeCampo($Campos_1$.campos , Campo.id) v
Campo.error
$Campos_0$.pend = $Campos_1$.pend unión Campo.pend

Campos → Campo

Campos.error = Campo.error
Campos.pend = Campo.pend

Campo → Tipo id

Campo.error = Tipo.error v ¬existeRef(Campo.tsh , Tipo.tipo)
Campo.pend = Tipo.pend

Instrucciones → Instrucciones ; Instrucción

$Instrucciones_0$.error = $Instrucciones_1$.error v Instrucción.error
Instruccion.tsh = $Instrucciones_1$.tsh = $Instrucciónes_0$.tsh

Instrucciones → Instrucción

Instrucciones.error = Instrucción.error
Instrucción.tsh = Instrucciones.tsh

Instrucción → InsProcedimiento

Instrucción.error = InsProcedimiento.error
InsProcedimiento.tsh = Instrucción.tsh

Instrucción → InsLectura

```
        Instrucción.error    = InsLectura.error
        InsLectura.tsh       = Instrucción.tsh

Instrucción → InsEscritura
        Instrucción.error    = InsEscritura.error
        InsEscritura.tsh     = Instrucción.tsh

Instrucción → InsAsignacion
        Instrucción.error    = InsAsignacion.error
        InsAsignacion.tsh    = Instrucción.tsh

Instrucción → InsCompuesta
        Instrucción.error    = InsCompuesta.error
        InsCompuesta.tsh     = Instrucción.tsh

Instrucción → InsIf
        Instrucción.error    = InsIf.error
        InsIf.tsh            = Instrucción.tsh

Instrucción → InsWhile
        Instrucción.error    = InsWhile.error
        InsWhile.tsh         = Instrucción.tsh

Instrucción → InsFor
        Instrucción.error    = InsFor.error
        InsFor.tsh           = Instrucción.tsh

Instrucción → InsNew
        Instrucción.error    = InsNew.error

Instrucción → InsDis
        Instrucción.error    = InsDis.error

InsProcedimiento → id Aparametros
        InsProcedimiento.error     =
                ¬existeID(InsProcedimiento.tsh,id.lex) v
                InsProcedimiento.tsh[id.lex].clase != proc v
                Aparametros.error
        AParametros.tsh            = InsProcedimiento.tsh
        AParametros.fparametrosh   =
                InsProcedimiento.tsh[id.lex].tipo.parametros

AParametros → (LAParametros)
        AParametros.error          =
                LAParametros.error v
                |Aparametros.fparametrosh| != LAParametros.nparametros
        LAParametros.tsh           = Aparametros.tsh
        LAParametros.fparametrosh  = Aparametros.fparametrosh

AParametros → λ
        AParametros.error          = |AParametros.fparametrosh | > 0

LAParametros → LAParametros , Expresion
```

LAParametros0.error =
LAParametros$_1$.error v
Expresion.error v
LAParametros0.nparametros > | LAParametros0.fparametrosh | v
(LAParemtros0.fparametrosh[LAParemetros0.nparametros].modo=var ∧
Expresion.modo = var) v
¬compatibles(LAParametros0.fparametrosh[LAParametros0.nparametros].tipo,
Expresion.tipo, LAParametros0.tsh)
LAParametros1.tsh = Expresion.tsh = LAParemtros0.tsh
LAParametros0.nparametros = LAParemetros1.nparametros +1
LAParametros1.fparametrosh = LAParametros0.fparametrosh

LAParametros → Expresion
LAParametros.error =
| LAParemetros.fparametrosh | < 1 v
(LAParametros.fparametrosh[0].modo = var ∧
Expresion.modo = val) v
¬compatible(LAParametros.fparametrosh[1].tipo, Expresion.tipo,
LAParametros.tsh)
Expresion.tsh = LAParams.tsh
LAParametros.nparametros = 1

InsLectura → in(id)
InsLectura.error = NOT existeID(InsLectura.tsh,id.lex)

InsEscritura → out(Expresion)
InsEscritura.error = (Expresion.tipo = error)

InsAsignación → Mem := Expresión
InsAsignacion.error =
¬ esCompatible(Mem.tipo, Expresion.tipo, InsAsignacion.tsh)
Expresion.tsh = Mem.tsh = InsAsignacion.tsh

InsCompuesta → { Instrucciones }
InsCompuesta.error = Instrucciones.error
Instrucciones.tsh = InsCompuesta.tsh

InsIf → if Expresion then Instrucción Pelse
InsIf.error =
Expresion.tipo != <t:bool> v Instrucción.error v Pelse.error
Pelse.tsh = Instrucción.tsh = Expresion.tsh =InsIf.tsh

PElse → else Instrucción
Pelse.error = Instrucción.error
Instrucción.tsh = Pelse.tsh

PElse → λ
Pelse.error = false

InsWhile → while Expresion do Instrucción
InsWhile.error =
Expresion.tipo = <t:bool> v

```
            Instrucción.error
      Instrucción.tsh     = Expresion.tsh = InsWhile.tsh

InsFor → for id=Expresion to Expresion do Instruccion
      InsFor.error  =
            (Expresion0.tipo!=<t:natural> y Expresion0.tipo != <t:integer>) v
            (Expresion1.tipo!=<t:natural> y Expresion1.tipo != <t:integer>) v
            (id.tipo != <t:natural> y id.tipo != <t:integer>)
      Instrucción.tsh = Expresion1.tsh = Expresion0.tsh = InsFor.tsh

InsNew → new Mem
      InsNew.error  = Mem.tipo.t != puntero

InsDis → dispose Mem
      InsDis.error  = Mem.tipo.t != puntero

Mem → id
      Mem.tipo     =
            si existe(Mem.tsh , id.lex)
                  si Mem.tsh[id.lex].clase = var
                        ref!(Mem.tsh[id.lex].tipo, Mem.tsh)
                  sino
                        <t:error>
            sino
                  <t:error>

Mem → Mem^
      Mem0.tipo    =
            si Mem1.tipo.t = puntero
                  ref!(Mem1.tipo.tbase , Mem0.tsh)
            sino
                  <t:error>
      Mem1.tsh     = Mem0.tsh

Mem → Mem[Expresion]
      Mem0.tipo    =
            si Mem1.tipo.t =array ∧ Expresion.tipo.t = natural
                  ref!(Mem1.tipo.tbase, Mem0.tsh)
            sino
                  <t:error>
      Expresion.tsh = Mem.tsh

Mem → Mem.id
      Mem0.tipo    =
            si Mem1.tipo.t = rec
                  si campo?(Mem1.tipo.campos, id.lex)
                        ref!(Mem1.tipo.campos[id.lex].tipo, Mem0.tsh)
                  sino <t:error>
            sino <t:error>

Expresión → ExpresiónNiv1 OpNiv0 ExpresiónNiv1
      Expresion.tipo             =
            si     (ExpresionNiv1_0.tipo = <t:error> v
```

$ExpresionNiv1_1$.tipo = <t:error>) v
(ExpresionNiv1$_0$.tipo = <t:character> ∧
$ExpresionNiv1_1$.tipo =/= <t:character>) v
(ExpresionNiv1$_0$.tipo =/= <t:character> ∧
$ExpresionNiv1_1$.tipo = <t:character>) v
(ExpresionNiv1$_0$.tipo = <t:boolean> ∧
$ExpresionNiv1_1$.tipo =/= <t:boolean>) v
(ExpresionNiv1$_0$.tipo =/= <t:boolean> ∧
$ExpresionNiv1_1$.tipo = <t:boolean>))
<t:error>
sino
<t:boolean>
$ExpresionNiv1_1$.tsh = $ExpresionNiv1_0$.tsh = Expresion.tsh
Expresion.modo = val

Expresión → ExpresiónNiv1
Expresion.tipo = ExpresionNiv1.tipo
ExpresionNiv1.tsh = Expresion.tsh
Expresion.modo = ExpresionNiv1.modo

ExpresiónNiv1 → ExpresiónNiv1 OpNiv1 ExpresiónNiv2
$ExpresionNiv1_0$.tipo =
si (ExpresionNiv1$_1$.tipo = <t:error> v
ExpresionNiv2.tipo = <t:error> v
(ExpresionNiv1$_1$.tipo = <t:character> v
ExpresionNiv2.tipo =/= <t:character>) v
(ExpresionNiv1$_1$.tipo = <t:boolean> ∧
ExpresionNiv2.tipo =/= <t:boolean>) v
(ExpresionNiv1$_1$.tipo =/= <t:boolean> ∧
ExpresionNiv2.tipo = <t:boolean>))
<t:error>
sino
case (OpNiv1.op)
suma,resta:
si (ExpresionNiv1$_1$.tipo=<t:float> v
ExpresionNiv2.tipo = <t:float>)
<t:float>
sino si (ExpresionNiv1$_1$.tipo = <t:integer> v
ExpresionNiv2.tipo = <t:integer>)
<t:integer>
sino si (ExpresionNiv1$_1$.tipo =<t:natural> ∧
ExpresionNiv2.tipo = <t:natural>)
<t:natural>
sino
<t:error>
o:
si (ExpresionNiv1$_1$.tipo = <t:boolean> ∧
ExpresionNiv2.tipo = <t:boolean>)

```
                                    <t:boolean>
                             sino
                                    <t:error>
                      or:
                             si     (ExpresionNiv21.tipo = <t:boolean> ∧
                                    (ExpresionNiv21.tipo = <t:boolean>
                                    <t:boolean>
                             sino
                                    <t:error>

      ExpresionNiv2.tsh   = ExpresionNiv1_1.tsh = ExpresionNiv1_0.tsh
      ExpresionNiv1.modo  = val
```

ExpresiónNiv1 → ExpresiónNiv2

```
      ExpresionNiv1.tipo  = ExpresionNiv2.tipo
      ExpresionNiv2.tsh   = ExpresionNiv1.tsh
      ExpresionNiv1.modo  = ExpresionNiv2.modo
```

ExpresiónNiv2 → ExpresiónNiv2 OpNiv2 ExpresiónNiv3

```
      ExpresionNiv2_0.tipo =
            si     (ExpresionNiv2_1.tipo = <t:error> v
                   ExpresionNiv3.tipo = <t:error> v
                   ExpresionNiv2_1.tipo = <t:character> v
                   ExpresionNiv3.tipo = <t:character> v
                   (ExpresionNiv2_1.tipo = <t:boolean> ∧
                          ExpresionNiv3.tipo =/= <t:boolean> v
                   (ExpresionNiv2_1.tipo =/= <t:boolean> ∧
                          ExpresionNiv3.tipo = <t:boolean>))
                   <t:error>
            sino
                   case (OpNiv2.op)
                          multiplica,divide:
                                 si     (ExpresionNiv2_1.tipo=<t:float> v
                                 ExpresionNiv3.tipo = <t:float>)
                                        <t:float>
                                 sino   si     (ExpresionNiv2_1.tipo =<t:integer> v
                                        ExpresionNiv3.tipo = <t:integer>)
                                               <t:integer>
                                 sino   si     (ExpresionNiv2_1.tipo =<t:natural> ∧
                                        ExpresionNiv3.tipo=<t:natural>)
                                               <t:natural>
                                 sino
                                        <t:error>
                          modulo:
                                 si     (ExpresionNiv3.tipo = <t:natural> ∧
                                        (ExpresionNiv2_1.tipo=<t:natural> v
                                        ExpresionNiv2_1.tipo=<t:integer>))
                                 ExpresionNiv2_1.tipo
                                 sino
                                        <t:error>
```

y:
si (ExpresionNiv2$_1$.tipo = <t:boolean> ∧
ExpresionNiv3.tipo = <t:boolean>)
<t:boolean>
sino
<t:error>
and:
si (ExpresionNiv21.tipo = <t:boolean> ∧
(ExpresionNiv21.tipo = <t:boolean>
<t:boolean>
sino
<t:error>

ExpresionNiv3.tsh = ExpresionNiv2$_1$.tsh = ExpresionNiv2$_0$.tsh
ExpresionNiv2.modo = val

ExpresiónNiv2 → ExpresiónNiv3
ExpresionNiv2.tipo = ExpresionNiv3.tipo
ExpresionNiv3.tsh = ExpresionNiv2.tsh
ExpresionNiv2.modo = ExpresionNiv3.modo

ExpresiónNiv3 → ExpresionNiv4 OpNiv3 ExpresiónNiv3
ExpresionNiv3$_0$.tipo =
si (ExpresionNiv4.tipo = <t:error> v
ExpresionNiv3$_1$.tipo = <t:error> v
ExpresionNiv4.tipo =/= <t:natural> v
ExpresionNiv3$_1$.tipo =/= <t:natural>)
<t:error>
sino
<t:natural>
ExpresionNiv4.tsh = ExpresionNiv3$_1$.tsh = ExpresionNiv3$_0$.tsh
ExpresionNiv3.modo = val

ExpresiónNiv3 → ExpresiónNiv4
ExpresionNiv3.tipo = ExpresionNiv4.tipo
ExpresionNiv4.tsh = ExpresionNiv3.tsh
ExpresionNiv3.modo = ExpresionNiv4.modo

ExpresiónNiv4 → OpNiv4 ExpresiónNiv4
ExpresionNiv4$_0$.tipo =
si (ExpresionNiv4$_1$.tipo = <t:error>)
<t:error>
sino
case (OpNiv4.op)
no:
si (ExpresionNiv4$_1$.tipo=<t:boolean>)
<t:boolean>
sino
<t:error>
menos:

si (ExpresionNiv4$_1$.tipo=<t:float>)
<t:float>
sino si (ExpresionNiv4$_1$.tipo=<t:integer> v
ExpresionNiv4$_1$.tipo = <t:natural>)
<t:integer>
sino
<t:error>

cast-float:
si (ExpresionNiv4$_1$.tipo=/= <t:boolean>)
<t:float>
sino
<t:error>

cast-int:
si (ExpresionNiv4$_1$.tipo=/= <t:boolean>)
<t:integer>
sino
<t:error>

cast-nat:
si (ExpresionNiv4$_1$.tipo=<t:natural> v
ExpresionNiv4$_1$.tipo=<t:character>)
<t:natural>
sino
<t:error>

cast-char:
si (ExpresionNiv4$_1$.tipo=<t:natural> v
ExpresionNiv4$_1$.tipo=<t:character>)
<t:character>
sino
<t:error>

ExpresionNiv4$_1$.tsh = ExpresionNiv4$_0$.tsh
ExpresionNiv4.modo = val

ExpresiónNiv4 → | Expresión |

ExpresionNiv4.tipo =
si (Expresion.tipo = <t:error> v
Expresion.tipo=<t:boolean> v
Expresion.tipo=<t:character>)
<t:error>
sino si (Expresion.tipo = <t:float>)
<t:float>
sino si (Expresion.tipo = <t:natural> v
Expresion.tipo = <t:integer>)
<t:natural>
sino
<t:error>
Expresion.tsh = ExpresionNiv4.tsh
ExpresionNiv4.modo = Expresion.modo

ExpresiónNiv4 → (Expresión)

ExpresionNiv4.tipo = Expresion.tipo

```
        Expresion.tsh              = ExpresionNiv4.tsh
        ExpresionNiv4.modo  = Expresion.modo

ExpresiónNiv4 → Literal
        ExpresionNiv4.tipo  = Literal.tipo
        Literal.tsh         = ExpresiónNiv4.tsh
        ExpresionNiv4.modo  = var

ExpresionNiv4 → Mem
        ExpresionNiv4.tipo  = Mem.tipo
        Mem.tsh                    = ExpresionNiv4.tsh
        ExpresionNiv4.modo  = var

Literal → litNat
        Literal.tipo  = <t:natural>
Literal → litFlo
        Literal.tipo  = <t:float>
Literal → litTrue
        Literal.tipo  = <t:boolean>

Literal → litFalse
        Literal.tipo  = <t:boolean>

Literal → litCha
        Literal.tipo  = <t:character>
Literal → litNull
        Literal.tipo  = <t:integer>

OpNiv0 → <
        OpNiv0.op     = menor
OpNiv0 → >
        OpNiv0.op     = mayor
OpNiv0 → <=
        OpNiv0.op     = menor-ig
OpNiv0 → >=
        OpNiv0.op     = mayor-ig
OpNiv0 → =
        OpNiv0.op     = igual
OpNiv0 → =/=
        OpNiv0.op     = no-igual
OpNiv1 → +
        OpNiv1.op     = suma
OpNiv1 → -
        OpNiv1.op     = resta
OpNiv1 → or
        OpNiv1.op     = o
OpNiv2 → *
        OpNiv2.op     = multiplica
OpNiv2 → /
        OpNiv2.op     = divide
OpNiv2 → %
        OpNiv2.op     = modulo
OpNiv2 → and
```

```
        OpNiv2.op     = y
OpNiv3 → >>
        OpNiv3.op     = shl
OpNiv3 → <<
        OpNiv3.op     = shr
OpNiv4 → not
        OpNiv4.op     = no
OpNiv4 → -
        OpNiv4.op     = menos
OpNiv4 → (float)
        OpNiv4.op     = cast-float
OpNiv4 → (int)
        OpNiv4.op     = cast-int
OpNiv4 → (nat)
        OpNiv4.op     = cast-nat
OpNiv4 → (char)
        OpNiv4.op     = cast-char
```

5. Especificación de la traducción

5.1. Lenguaje objeto y máquina virtual

La máquina P que se adjunta, implementada en Java, consiste en:

· Una pila teóricamente infinita (su tamaño máximo real depende del sistema en el cual se ejecute) capaz de guardar datos.

· Una lista teóricamente infinita que representa el programa a ejecutarse.

· Una memoria de tamaño configurable donde se guardan la pila de llamadas y el heap dinámico.

· Un booleano "parar" que indica si la máquina debe o no leer la siguiente instrucción. Cuando valga cierto la máquina dejará de ejecutar el programa.

· Un entero "cp" cuya función es servir de índice dentro del programa, señalando la instrucción que se está ejecutando en cada momento.

El interprete acepta los siguientes tipos de datos:

- Bool puede ser "true" o "false".
- Character es un dato que ocupa dos bytes que representa caracteres en formato UTF-8.
- Entero es un número entero en C2 de 32 bits.
- Natural es un número natural de 31 bits.
- Float es un número real en IEEE 754 de 32 bits.

Cada celda de la pila y de la memoria es capaz de contener cualquier tipo de dato aceptado por el interprete, independientemente de su tamaño en bits.
Cada celda de la lista que representa el programa es capaz de guardar cualquier tipo de instrucción independientemente de su tamaño.

5.1.2. Comportamiento interno de la máquina P

El funcionamiento de la máquina P es:

```
pila = pilaVacia();
cp ← 0
parar ← false
mientras(no parar)
```

```
instrucción = programa[cp]
si(ejecutar(instruccion))
        cp++
```

Esto se repite un número indeterminado de veces, parando cuando ejecute una instrucción "parar" o cuando se produzca un fallo en ejecucución (como podría ser una división por cero o encontrar tipos de valores incompatibles al realizar una operación).
Ejecutar una instrucción devolverá *false* y por tanto no se modificará el cp cuando esta instrucción modifique por su cuenta el cp (como, por ejemplo, las instrucciones de salto).

5.1.3. Repertorio de instrucciones del código P

El lenguaje objeto es un sencillo lenguaje de pila, es decir, las instrucciones apilan o desapilan datos (según indique su semántica) en una pila teóricamente infinita, teniendo como apoyo una memoria de acceso aleatorio donde almacenar la pila de llamadas y el heap dinámico.

La tabla siguiente muestra las instrucciones de este lenguaje.

· La columna "Código" indica el valor que tiene internamente la instrucción.

· Cada celda de la columna "Args" puede ser 0 (es decir, no tiene argumentos) o "<tipo del argumento> <nombre del argumento>", el nombre del argumento puede ser usado en las dos siguientes celdas.

· La columna "Interacción con la pila" sigue la siguiente sintaxis:

· Lo que se encuentre a la izquierda de → es el estado anterior de la pila y lo de la derecha el estado posterior (tras aplicar la instrucción).

· Las "," separan los elementos de la pila.

· Los tres puntos ("…") significan "el resto de la pila".

· La estructura "<tipo> <nombre>" especifica un valor en la pila.

· Los tipos pueden ser o bien los definidos en el lenguaje (Nat, Bool, Int, etc) o bien otra palabra en mayúsculas (por ejemplo "T"), en cuyo caso se tratará de un tipo genérico, pero todas las apariciones de esa palabra corresponderán al mismo tipo.

Por ejemplo: "…, T v1, T v2 → …, Bool res" indica que la instrucción desapila 2 variables y apila una variable booleana. Además v1 y v2 deben tener el mismo tipo.

Código	Nombre mnemotécnico	Args	Interacción con la pila	Descripción
0	Parar	0	… → …	Detiene la ejecución.
1	Apila	T dato	… → …, T dato	Apila en la pila el dato pasado como argumento.
2	Apila-dir	Nat dir	... → …, T M[dir]	Apila en la pila el contenido de la posición de memoria pasada como argumento.
3	Desapilar	0	…, T valor → …	Desapila el primer dato de la pila.
4	Desapilar-dir	Nat dir	…, T valor → …	Desapila el primer dato de la pila y lo almacena en la posición de memoria pasada como parámetro.
5	Menor	0	…, T v1, T v2 → …, Bool res	res = t1 < t2
6	Mayor	0	…, T v1, T v2 → …,	res = t1 > t2

			Bool res	
7	MenorIg	0	…, T v1, T v2 → …, Bool res	res = t1 <= t2
8	MayorIg	0	…, T v1, T v2 → …, Bool res	res = t1 >= t2
9	Igual	0	…, T v1, T v2 → …, Bool res	res = t1 == t2
10	No-Igual	0	…, T v1, T v2 → …, Bool res	res = t1 != t2
11	Sumar	0	…, T v1, T v2 → …, T res	res = v1 + v2. T tiene que ser un tipo numérico.
12	Restar	0	…, T v1, T v2 → …, T res	res = v1 - v2. T tiene que ser un tipo numérico.
13	Mul	0	…, T v1, T v2 → …, T res	res = v1 * v2. T tiene que ser un tipo numérico.
14	Div	0	…, T v1, T v2 → …, T res	res = v1 / v2. T tiene que ser un tipo numérico.
15	Mod	0	…, T v1, Nat v2 → …, T res	res = v1 % v2. T tiene que ser un tipo numérico.
16	Y	0	…, Bool v1, Bool v2 → …, Bool res	res = v1 && v2
17	O	0	…, Bool v1, Bool v2 → …, Bool res	res = v1 || v2
18	No	0	…, Bool v → …, Bool res	res = !v1
19	Negativo	0	…, T v → …, T res	res = -v. T tiene que ser un tipo numérico distinto a Nat
20	Shl	0	…, T v1, Nat v2 → …, T res	res = v1 << v2. T tiene que ser un tipo numérico.
21	Shr	0	…, T v1, Nat v2 → …, T res	res = v1 >> v2. T tiene que ser un tipo numérico.
22	CastInt	0	…, T v → ..., Int res	res = v como entero. Siguiendo la conversión especificada en este documento
23	CastChar	0	…, T v → ...,Char res	res = v como caracter. Siguiendo la conversión especificada en este documento
24	CastFloat	0	…, T v → …, Float res	res = v como float. Siguiendo la conversión especificada en este documento
25	CastNat	0	…, T v → …, Nat res	res = v como natural. Siguiendo la conversión especificada en este documento
26	Abs	0	…, T v → …, T res	res = | v | (excepto char y bool)

27	Salida	0	… → …, T res	Muestra por pantalla el valor del identificador
28	Entrada_Bool	0	… → …, Bool id	Entrada para identificadores de tipo booleano.
29	Entrada_Char	0	… → …, Char id	Entrada para identificadores de tipo caracter.
30	Entrada_Float	0	… → …, Float id	Entrada para identificadores de tipo real.
31	Entrada_Int	0	… → …, Integer id	Entrada para identificadores de tipo entero.
32	Entrada_Nat	0	… → …, Nat id	Entrada para identificadores de tipo natural.
33	Apilar_Ind	0	…, T d → …, T id	Id = memoria[d], siendo d un tipo que pueda castearse a natural
34	Desapila_Ind	0	…, T d, K v → …	memoria[d] = v, siendo d un tipo que pueda castearse a natural
35	Mueve	N t	…, T d, K o → …	mueve t celdas de o a d, siendo N natural o entero
36	New	Nat n	… → Nat dir	Reserva n celdas en el heap. dir = primera dirección de esas celdas
37	Delete	Nat n	…, Nat dir →	Elimna n celdas consecutivas del heap comenzando por dir
38	Seg	Int i	Φ → Φ	Determina la dirección de memoria donde acaba la memoria estática y comienza la dinámica
39	Ir_a	Nat dir	… → …	cp = dir return false
40	Ir_ind	0	…, T dir → …	cp = dir Dir debe ser un natural o entero return false
41	Ir_false	Nat dir	…, Bool b → …	Si b == false then cp = dir /\ return false
42	Ir_false	Nat dir	…, Bool b → …	Si b == true then cp = dir /\ return false
43	Copia	0	…, T t → …, T t, T t	Copia el valor de la cima de la pila

5.2. Funciones semánticas

Descripción, si procede, de las funciones semánticas adicionales utilizadas en la especificación. Para cada función debe indicarse explícitamente su cabecera, así como informalmente su cometido, incluyendo el propósito de cada uno de sus parámetros.

Esta sección puede dejarse vacía si no se van a usar funciones semánticas adicionales.

```
fun inicio(numNiveles : int, tamDatos : int)
    apila(numNiveles+2)
    desapila-dir(1)
    apila(1+numNiveles+tamDatos)
    desapila-dir(0)
ffun
cons longInicio = 4

fun apila-ret(ret)
    apila-dir 0  ||
    apila 1      ||
    suma         ||
    apila ret    ||
    desapila-ind
ffun
cons longApilaRet = 5

fun prologo(nivel,tamlocales)
    apila-dir 0           ||
    apila 2               ||
    suma                  ||
    apila-dir 1+nivel     ||
    desapila-ind          || // se reserva el display
    apila-dir 0           ||
    apila 3               ||
    suma                  ||
    desapila-dir 1+nivel  || // se fija el valor del actual
    apila-dir 0           ||
    apila tamlocales+2    ||
    suma                  ||
    desapila-dir 0           // se reserva espacio
ffun
cons longPrologo = 13

fun epilogo(nivel)
    apila-dir 1+nivel     ||
    apila 2               ||
    resta                 ||
    apila-ind             || // dir retorno
    apila-dir 1+nivel     ||
    apila 3               ||
    resta                 ||
    copia                 ||
    desapila-dir 0        || // cp
    apila 2               ||
    suma                  ||
    apila-ind             ||
                             // recuperado antiguo display
    desapila-dir 1+nivel
ffun
cons longEpilogo = 13

fun accesoVar(infoID)
    apila-dir 1+infoID.nivel  ||
    apila infoID.dir          ||
    suma
```

```
    si infoID.clase = pvar
        apila-ind
    si no
        λ
ffun

fun longAccesoVar(infoID)
    si infoID.clase = pvar entonces
        4
    si no
        3
ffun

fun inicio(numNiveles,tamDatos)
    apila numNiveles+2
    desapila-dir 1
    apila 1+numNiveles+tamDatos
    desapila-dir 0
ffun

cons inicio-paso = apila-dir(0) || apila(3) || suma

cons longInicioPaso = 3

cons fin-paso = desapila

cons longFinPaso = 1

fun direccionParFormal(pformal)
    devuelve
        apila(pformal.dir) ||
        suma
ffun

const longdireccionParFormal = 2

fun pasoParametro(modoReal,pformal)
    devuelve
        (si pformal.modo = val ∧ modoReal = var
            mueve(pformal.tipo.tam) // copia del valor en el caso de expr. mem
         si no desapila-ind        // copia del valor, o bien de la dirección )
ffun

const longPasoParametro = 1
```

5.3. Atributos semánticos

5.3.1 Atributos “cod”

El atributo sintetizado “cod” almacena la lista de instrucciones que van conformando el programa. La única salvedad al codigo es debida al uso de procedimientos forward. Al traducir llamadas a procedimientos forward no se conoce la dirección del salto, por lo que se deja con un nop. En estos

casos se añade un nop en la posición donde debería estar el salto, nop que luego será parseado en la declaración con cuerpo del procedimiento.

5.3.2 Atributo “etq” y “etqh”

El atributo heredado “etqh” almacena la etiqueta de cada bloque, es decir, la dirección absoluta dentro del programa que donde empezará el código que cada producción produzca.
El atributo sintetizado “etq” es similar a “etqh”, pero su valor es la dirección absoluta dentro del programa de la siguiente instrucción a la última producida. De esta manera:

$$longitudCodigo = etq - etqh$$

5.3.2 Atributo “dir” y “dirh”

En cada producción de declaración, el atributo heredado “dirh” es la posición dentro del marco de activación de la pila de llamadas que le correspondería a la siguiente variable que se declare (de haberla).
La producción “Declaraciones” tiene un atributo sintetizado “dir” que es el valor “dirh” de la siguiente declaración, de tal manera que:

$$dir = dirh + espacioDeclarado$$

Este atributo es propio de cada procedimiento. Comienza valiendo cero, aumenta con los parámetros del procedimiento y luego con sus variables internas.

5.3.3 Atributo “n” y “nh”

En las producciones de declaración, el atributo sintetizado “n” indica el número máximo de subprogramas que se declaran en esa rama del árbol de declaración.
Por otro lado el atributo heredado “nh” indica la profundidad del subprograma actual, de tal manera que:

$$n = nh + nuevosSubprogramasDeclarados$$

5.3.4 Atributo “tam”

El atributo sintetizado “tam” de los tipos indica el número de direcciones de memoria (de entradas en el marco de activación) que requiere el tipo declarado. Las producciones de campos y “Declaracion” también tienen un atributo sintetizado “tam” cuyo valor es la suma del tamaño de todos los tipos que se declaran por debajo.

5.3.5 Atributos “props” y “propsop”

Los atributos sintetizados “props” y “propsop” almacenan propiedades. Se trata de una tabla de propiedades que facilita la inserción posterior de estos valores en la tabla de símbolos.
El atributo “props” almacena propiedades de chequeo de errores como el nombre de la variable o su tipo, mientras que “propsop” almacena valores operacionales que se usarán en la traducción (como la dirección de la variable

5.3.6 Atributo “modo”

El atributo sintetizado “modo” puede valer “val” o “var”, indicando en el primer caso que se trata de un valor y en el segundo que se trata de una variable. Dependiendo del valor de este atributo el acceso a la memoria (en asignación o en llamadas a procedimientos) es diferente.

Por ejemplo, si la variable “a” vale 7, entonces la expresion “a” tiene modo “val”, sin embargo la expresión “a + 2” tiene modo “var”.

5.3.7 Atributo “parh”

El atributo heredado “parh” indica si el acceso a memoria es por valor o por referencia, de tal manera que si es false se entiende que se demanda la información por valor y por tanto se deja en la cima el valor de la expresión y en otro contrario (por variable) se deja la indirección.
Aunque “parh” y “modo” puedan parecerse, no son equivalentes. Principalmente porque mientras “modo” lo impone el acceso a memoria (sintetizado), “parh” lo impone la declaración del subprograma (si ese parámetro se declara o no por variable).

5.4. Gramática de atributos

Gramática de atributos que formaliza la traducción.

Programa → Declaraciones & Instrucciones

```
Programa.cod  =
    inicio(Declaraciones.n,Declaraciones.dir) ||
    ir-a(Declaraciones.etq)          ||
    Declaraciones.cod                ||
    Instrucciones.cod       ||
    stop
Declaraciones.etqh      = longInicio + 1
Instrucciones.etqh      = Declaraciones.etq
Declaraciones.dirh      = 0
Declaraciones.nh        = 0
```

Declaraciones → Declaraciones ; Declaracion

```
Declaraciones0.ts       =

   añadeID(Declaraciones1.ts,Declaracion.id,Declaracion.props)
Declaraciones1.dirh     = Declaraciones0.dirh
Declaraciones0.dir      = Declaraciones1.dir + Declaracion.tam
Declaracion.dirh        = Declaraciones1.dir
Declaraciones1.nh       = Declaraciones0.nh
   Declaracion.nh                   = Declaraciones0.nh
Declaraciones0.n        = max(Declaraciones1.n,Declaracion.n)
Declaraciones1.etqh     = Declaraciones0.etqh
Declaracion.etqh        = Declaraciones1.etq
Declaraciones0.etq      = Declaracion.etq
Declaraciones0.cod      = Declaraciones1.cod || Declaracion.cod
```

Declaraciones → Declaracion

```
Declaraciones.ts        =
          añadeID(Declaraciones.ts,Declaracion.id,Declaracion.props)
Declaraciones.dir       = Declaraciones.dirh + Declaracion.tam
Declaracion.dirh        = Declaraciones.dirh
Declaracion.nh          = Declaraciones.nh
Declaraciones.n         = Declaracion.n
Declaracion.etqh        = Declaraciones.etqh
Declaraciones.etq       = Declaracion.etq
Declaraciones.cod       = Declaracion.cod
```

Declaracion → DeclaracionTipo

```
Declaracion.props      = DeclaracionTipo.props ++ <>
Declaracion.tam         = 0
Declaracion.n           = Declaracion.nh
Declaracion.cod         = λ
Declaracion.etq         = Declaracion.etqh
```

Declaracion → DeclaracionVariable

```
DeclaracionVariable.props =
          DeclaracionVariable.props ++ <dir:Declaracion.dirh>
Declaracion.tam             = DeclaracionVariable.props.tipo.tam
Declaracion.n               = Declaracion.nh
Declaracion.cod             = λ
Declaracion.etq             = Declaracion.etqh
```

Declaracion → DeclaracionProcedimiento

```
Declaracion.tam                 = 0
DeclaracionProcedimiento.nh     = Declaracion.nh
Declaracion.n                   = DeclaracionProcedimiento.n
DeclaracionProcedimiento.etqh   = Declaracion.etqh
Declaracion.etq                 = DeclaracionProcedimiento.etq
Declaracion.cod                 = DeclaracionProcedimiento.cod
Declaracion.props               = DeclaracionProcedimiento.props
```

Tipo → Boolean

```
Tipo.tipo = <t:boolean,tam:1>
```

Tipo → Character

```
Tipo.tipo = <t:character,tam:1>
```

Tipo → Float

```
Tipo.tipo = <t:float,tam:1>
```

Tipo → Natural

```
Tipo.tipo = <t:natural,tam:1>
```

Tipo → Integer

```
Tipo.tipo = <t:integer,tam:1>
```

Tipo → iden

```
Tipo.tipo =
    <
        t:ref,
        id:iden.lex,
        tam:Tipo.tsh[iden.lex].tipo.tam
    >
```

Tipo → array [num] of Tipo

```
Tipo.tipo =
    <
        t:array,
```

```
            nelems:valorDe(num.lex),
            tbase:Tipo.tipo,
            tam:valorDe(num.lex)*Tipo1.tipo.tam
        >

Tipo → ^Tipo
    Tipo0.tipo =
        <
            t:puntero,
            tbase:Tipo1.tipo,
            tam:1
        >

Tipo → reg Campos freg
    Tipo.tipo =
        <
            t:array,
            campos:Campos.campos,
            tam:Campos.tam
        >


Campos → Campos ; Campo
    Campos0.tam = Campos1.tam + Campo.tam
    Campo.desh  = Campos1.tam

Campos → Campo
    Campos.tam = Campo.tam
    Campo.desh = 0


Campo → Tipo id
    Campo.campo =
        <
            id:iden.lex,
            tipo:Tipo.tipo,
            desp:Campo.desph
        >
    Campo.tam   = Tipo.tam

DeclaracionProcedimiento → proc id FParametros Bloque
    Bloque.dirh                     = FParametros.dir
    FParametros.nh                  = Bloque.nh = DeclaracionProcedimiento.nh + 1
    DeclaracionProcedimiento.n      = Bloque.n
    DeclaracionProcedimiento.cod    = Bloque.cod
    DeclaracionProcedimiento.props  = <inicio:Bloque.inicio>
    Bloque.etqh                     = DeclaracionProcedimiento.etqh
    DeclaracionProcedimiento.etq    = Bloque.etq
    Bloque.tsph                     =
        añadeID(
            FParametros.ts,
            DecProp.id,
            DeclaracionProcedimiento.props)
```

```
    )
```

FParametros → (LFParametros)

```
    FParametros.dir = LFParametros.dir
```

FParametros → λ

```
    FParametros.dir = 0
```

LFParametros → LFParametros, FParametro

```
    LFParametros0.ts    = añadeID(LFParametros1.ts,FParametro.id,
                            FParametro.props ⊗ <dir:LFParametros1.dir>)
    LFParametros0.dir   = LFParametros1.dir + FParametro.tam
    FParametro.dirh     = LFParametros1.dir
```

LFParametro → FParametro

```
    LFParametro.ts      = añadeID(LFParametro.tsh, FParametro.id,FParametro.props ⊗
<dir:0>)
    LFParametro.dir     = FParametro.tam
    FParametro.dirh     = 0
```

FParametro → var Tipo id

```
    FParametro.tam      = 1
    FParametro.param    =
              <modo: variable, tipo: Tipo.tipo, dir: FParametro.dirh>
```

FParametro → Tipo id

```
    FParametro.tam      = Tipo.tipo.tam
    FParametro.param    = <modo: valor, tipo: Tipo.tipo, dir: FParametro.dirh>
```

Bloque → Declaraciones && Instrucciones

```
    Declaraciones.dirh      = Bloque.dirh
    Declaraciones.nh        = Bloque.nh
    Bloque.n        = Declaraciones.n
    Declaraciones.etqh      = Bloque.etqh
    Bloque.inicio   = Declaraciones.etq
    Instrucciones.etqh       = Declaraciones.etq + longPrologo
    Bloque.etq      = Instrucciones.etq + longEpilogo + 1
    Bloque.cod      =
            Declaraciones.cod                   ||
            prologo(Bloque.nh,Declaraciones.dir) ||
            Instrucciones.cod                   ||
            epilogo(Bloque.nh)                  ||
            ir-ind
```

Bloque → Instrucciones

```
    Bloque.n        = Bloque.nh
    Bloque.cod      =
```

```
        prologo(Bloque.nh,Bloque.dirh)   ||
        Instrucciones.cod                ||
        epilogo(Bloque.nh)               ||
        ir-ind
    Instrucciones.etqh        = Bloque.etqh + longPrologo
    Bloque.inicio   = Bloque.etqh
    Bloque.etq      = Instrucciones.etq + longEpilogo + 1
```

Instrucciones → Instrucciones ; Instruccion

```
    Instrucciones0.cod  = Instrucciones1.cod || Instruccion.cod
    Instrucciones1.etqh = Instrucciones0.etqh
    Instruccion.etqh    = Instrucciones1.etq
    Instrucciones0.etq  = Instruccion.etq
```

Instrucciones → Instruccion

```
    Instrucciones.cod   = Instruccion.cod
    Instruccion.etqh    = Instrucciones.etqh
    Instrucciones.etq   = Instruccion.etq
```

Instruccion → InsAsignacion

```
    Instrucciones.cod       = InstruccionAsignacion.cod
    InsAsignacion.etqh      = Instruccion.etzqh
    Instruccion.etq         = InstruccionAsignacion.etq
```

Instruccion → InsLectura

```
    Instruccion.cod         = InsLectura.cod
    InsLectura.etqh         = Instruccion.etqh
    Instruccion.etq         = InsLectura.etq
```

Instruccion → InsEscritura

```
    Instruccion.cod         = InsEscritura.cod
    InsEscritura.etqh       = Instruccion.etqh
    Instruccion.etq         = InsEscritura.etq
```

Instruccion → InsCompuesta

```
    Instruccion.cod         = InsCompuesta.cod
    InsCompuesta.etqh       = Instruccion.etqh
    Instruccion.etq         = InsCompuesta.etq
```

Instruccion → InsIf

```
    Instruccion.cod         = InsIf.cod
    InsIf.etqh              = Instruccion.etqh
    Instruccion.etq         = InsIf.etq
```

Instruccion → InsWhile

```
    Instruccion.cod         = InsWhile.cod
    InsWhile.etqh           = Instruccion.etqh
    Instruccion.etq         = InsWhile.etq
```

Instruccion → InsFor

```
    Instruccion.cod         = InsFor.cod
    InsFor.etqh             = Instruccion.etqh
```

```
    Instruccion.etq         = InsFor.etq

Instruccion → InsNew
    Instruccion.cod         = InsNew.cod
    InsNew.etqh             = Instruccion.etqh
    Instruccion.etq         = InsNew.etq

Instruccion → InsDis
    Instruccion.cod         = InsDis.cod
    InsDis.etqh             = Instruccion.etqh
    Instruccion.etq         = InsDis.etq

Instruccion → InsProcedimiento
    InsProcedimiento.etqh   = Instruccion.etqh
    Instruccion.etq         = InsProcedimiento.etq
    Instruccion.cod         = InsProcedimiento.cod

InsLectura → in(id)
    InsLectura.cod  = in InsLectura.tsh[id.lex].dir
    InsLectura.etq  = InsLectura.etqh + 1


InsEscritura → out(Expresion)
    InsEscritura.cod   = Expresion.cod || out
    Expresion.etqh           = InsEscritura.etqh
    InsEscritura.etq   = Expresion.etq + 1
    Expresion.parh           = false


InsAsignación → Mem := Expresion
    InsAsignación.cod =
        si esCompatibleConTipoBasico(Mem.tipo,Expresion.tsh)
            Mem.cod || Expresion.cod || desapila-ind
        si no
            Mem.cod || Expresion.cod || mueve(Mem.tipo.tam)
    Mem.etqh          = InsAsignación.etqh
    Expresion.etqh          = Mem.etq
    InsAsignación.etq = Expresion.etq + 1
    Expresion.parh          = false


InsCompuesta → { Instrucciones }
    InsCompuesta.cod    = Instrucciones.cod
    Instrucciones.etqh  = InsCompuesta.etqh
    InsCompuesta.etq    = Instrucciones.etq


InsIf → if Expresion then Instruccion Pelse
    InsIf.cod           =
        Expresion.cod ||
              ir-f(Instruccion.etq + 1) ||
              Instruccion.cod ||
              ir-a(PElse.etq) ||
```

```
            PElse.cod
    Expresion.etqh      = InsIf.etqh
    Instruccion.etqh    = Expresion.etq + 1
    PElse.etqh          = Instruccion.etq + 1
    InsIf.etq           = PElse.etq
    Expresion.parh      = false
```

PElse → else Instruccion

```
    PElse.cod           = Instruccion.cod
    PElse.etq           = Instruccion.etq
    Instruccion.etqh    = PElse.etqh
```

PElse → λ

```
    PElse.cod  = λ
    PElse.etq  = PElse.etqh
```

InsWhile → while Expresion do Instruccion

```
    InsWhile.cod        =
        Expresion.cod               ||
        ir-f(Instruccion.etq + 1)   ||
        Instruccion.cod             ||
        ir-a(InsWhile.etqh)
    Expresion.etqh      = InsWhile.etqh
    Instruccion.etqh    = Expresion.etq + 1
    InsWhile.etq        = Instruccion.etq + 1
```

InsFor → for id=Expresion to Expresion do Instruccion

```
    InsFor.cod  =
        Expresion0.cod                      ||
        desapila-dir InsFor.tsh[id.lex].dir ||
        Expresion1.cod                      ||
        dup                                 ||
        apila-dir InsFor.tsh[id.lex].dir    ||
        igual                               ||
        ir-v( InsFor.etq - 1)               ||
        Instruccion.cod                     ||
        apila-dir InsFor.tsh[id.lex].dir    ||
        apilar 1                            ||
        sumar                               ||
        desapila-dir InsFor.tsh[id.lex].dir ||
        ir-a (Expresion1.etq)
        pop
    Expresion0.etqh   = InsFor.etqh
    Expresion1.etqh   = Expresion0.etq + 1
    Instruccion.etqh  = Expresion1.etq + 4
    InsFor.etq        = Instruccion.etq + 6
```

InsNew → new Mem

```
    InsNew.cod  =
```

```
        Mem.cod ||
        new(
            si Mem.tipo.tbase = ref
                InsNew.tsh[Mem.tipo.tbase.id].tam
            si no
                1
        )
        || desapila-ind
    Mem.etqh    = InsNew.etqh
    InsNew.etq  = Mem.etq + 2
```

InsDis → delete Mem

```
    InsDis.cod  =
        Mem.cod ||
        del(
            si Mem.tipo.tbase = ref
                InsDis.tsh[Mem.tipo.tbase.id].tam
            si no
                1
        )
    Mem.etqh    = InsDis.etqh
    InsDis.etq  = Mem.etq + 1
```

InsProcedimiento → iden AParametros

```
    InsProcedimiento.cod    =
      si (InsProcedimiento.ts[iden.lex].clase==forward)
              apila-ret(InsProcedimiento.etq)         ||
        AParametros.cod                           ||
              nop(iden) //parchear
      sino
        apila-ret(InsProcedimiento.etq)         ||
              AParametros.cod                           ||
        ir-a(InsProcedimiento.tsh[iden.lex].inicio)
    AParametros.etqh            = InsProcedimiento.etqh + longApilaRet
    InsProcedimiento.etq    = AParametros.etq + 1
```

AParametros → LAParametros

```
    AParametros.cod     = inicio-paso || LAParametros.cod || fin-paso
    LAParametros.etqh   = AParametros.etqh + longInicioPaso
    AParametros.etq     = LAParametros.etq + longFinPaso
```

AParametros → λ

```
    AParametros.cod     = λ
    AParametros.etq     = Aparametros.etqh
```

LAParametros ::= LAParametros, Expresion

```
    LAParametros0.cod   =
        LAParametros1.cod ||
        copia ||
        direccionParFormal(LAParametros0.fParametros[LAParametros0.nParametros]) ||
```

```
        Expresion.cod ||
        pasoParametro(Expresion.modo,LAParametros0.fParametros[LAParametros0.nParametros])
    LAParametros1.etqh  = LAParametros0.etqh
    Expresion.etqh  = LAParametros1.etq + 1 + longDireccionParFormal
    LAParametros0.etq   = Expresion.etq + longPasoParametro
    Expresion.parh  = LAParametros0.fParametrosh[LAParametros0.nParametros].modo == var

LAParametros ::= Expresion
    LAParametros.cod    =
        copia ||
        Expresion.cod ||
        pasoParametro(Expresion.modo,LAParametros.fParametros[0])
    Expresion.etqh  = LAParametros.etq + 1
    LAParametros.etq    = Expresion.etq + longPasoParametro
    Expresion.parh  = LAParametros.fParametrosh[1].modo == var

Mem → id
    Mem.cod    = accesoVar(Mem.tsh[id.lex])
    Mem.etq    = Mem.etqh + longAccesoVar(Mem.tsh[id.lex])

Mem → Mem->
    Mem0.cod   = Mem1.cod || apila-ind
    Mem1.etqh  = Mem0.etqh
    Mem0.etq   = Mem1.etq + 1

Mem → Mem[Expresion]
    Mem0.cod        =
        Mem1.cod ||
              Expresion.cod ||
              apila Mem1.tipo.tbase.tam ||
              multiplica ||
              suma
    Mem1.etqh       = Mem0.etqh
    Expresion.etqh  = Mem1.etq
    Mem0.etq        = Expresion.etq + 3

Mem → Mem.id
    Mem0.cod   = Mem1.cod || apila(Mem1.tipo.campos[iden.lex].desp) || suma
    Mem1.etqh  = Mem0.etqh
    Mem0.etq   = Mem1.etq + 2

Expresion → ExpresionNivel1 OpNiv0 ExpresionNivel1
    Expresion.cod         =
        ExpresionNivel10.cod ||
        ExpresionNivel11.cod ||
        case (OpNiv0.op)
            menor:
                menor
            mayor
                mayor
            menor-ig
                menorIg
            mayor-ig
```

```
                mayorIg
            igual
                igual
            no-igual
                no-igual
    ExpresionNivel10.etqh   = Expresion.etqh
    ExpresionNivel11.etqh   = ExpresionNivel10.etqh
    Expresion.etq           = ExpresionNivel11.etq +1
    Expresion.modo          = val
    ExpresionNivel10.parh   = ExpresionNivel11.parh = false

Expresion → ExpresionNivel1
    Expresion.cod           = ExpresionNivel1.cod
    ExpresionNivel1.etqh    = Expresion.etqh
    Expresion.etq           = ExpresionNivel1.etq
    Expresion.modo          = ExpresionNivel1.modo
    ExpresionNivel1.parh    = Expresion.parh

ExpresionNiv1 → ExpresionNiv1 or ExpresionNiv2
    ExpresionNiv10.cod  =
        ExpresionNiv11.cod ||
        dup ||
        ir-v(ExpresionNiv2.etq) ||
        desapila ||
        ExpresionNiv2.cod
    ExpresionNiv11.etqh = ExpresionNiv10.etqh
    ExpresionNiv2.etqh  = ExpresionNiv11.etq + 3
    ExpresionNiv10.etq  = ExpresionNiv2.etq
    ExpresionNiv1.modo  = val
    ExpresionNiv11.parh = ExpresionNiv2.parh = false

ExpresionNiv1 → ExpresionNiv1 OpNiv1 ExpresionNiv2
    ExpresionNiv11.etqh = ExpresionNiv10.etqh
    ExpresionNiv1.modo  = val
    ExpresionNiv11.parh = ExpresionNiv2.parh = false

    si (OpNiv1.op == or)
            ExpresionNiv10.cod =
             ExpresionNiv11.cod ||
             ExpresionNiv2.cod ||
             case (OpNiv1.op)
                 suma:
                     sumar
                 resta:
                     restar
           ExpresionNiv2.etqh  = ExpresionNiv11.etq
        ExpresionNiv10.etq  = ExpresionNiv2.etq + 1
      sino
        ExpresionNiv10.cod  =
            ExpresionNiv11.cod ||
               dup ||
                  ir-v(ExpresionNiv2.etq) ||
            desapila ||
```

```
            ExpresionNiv2.cod
        ExpresionNiv2.etqh  = ExpresionNiv11.etq + 3
        ExpresionNiv10.etq  = ExpresionNiv2.etq
```

ExpresionNiv1 → ExpresionNiv2

```
    ExpresionNiv1.cod  = ExpresionNiv2.cod
    ExpresionNiv2.etqh = ExpresionNiv1.etqh
    ExpresionNiv1.etq  = ExpresionNiv2.etq
    ExpresionNiv1.modo = ExpresionNiv2.modo
    ExpresionNiv2.parh = ExpresionNiv1.parh
```

ExpresionNiv2 → ExpresionNiv2 OpNiv2 ExpresionNiv3

```
    ExpresionNiv21.etqh   = ExpresionNiv20.etqh
    ExpresionNiv20.modo   = val
    ExpresionNiv21.parh   = ExpresionNiv3.parh = false

       if (OpNiv2.op == and)
           ExpresionNiv20.cod    =
                  ExpresionNiv21.cod ||
                  ExpresionNiv3.cod  ||
                  case(OpNiv2.op)
                  Multiplica:
                         Mul
                Divide:
                    Div
                Modulo:
                    Mod
                y:
                    Y
           ExpresionNiv3.etqh    = ExpresionNiv21.etq
           ExpresionNiv20.etq    = ExpresionNiv3.etq +1
       sino
           ExpresionNiv20.cod    =
                    ExpresionNiv21.cod ||
               ir-f(ExpresionNiv2.etq + 1) ||
               ExpresionNiv2.cod ||
               ir-a(ExpresionNiv2.etq + 2) ||
               apila(0)
           ExpresionNiv2.etqh       = ExpresionNiv21.etq + 1
           ExpresionNiv20.etq    = ExpresionNiv2.etq + 2
```

ExpresionNiv2 → ExpresionNiv3

```
    ExpresionNiv3.etqh    = ExpresionNiv2.etqh
    ExpresionNiv2.etq     = ExpresionNiv3.etq
    ExpresionNiv2.cod     = ExNiv3.cod
    ExpresionNiv20.modo   = ExpresionNiv3.modo
    ExpresionNiv3.parh    = ExpresionNiv2.parh;
```

ExpresionNiv3 → ExpresionNiv4 OpNiv3 ExpresionNiv3

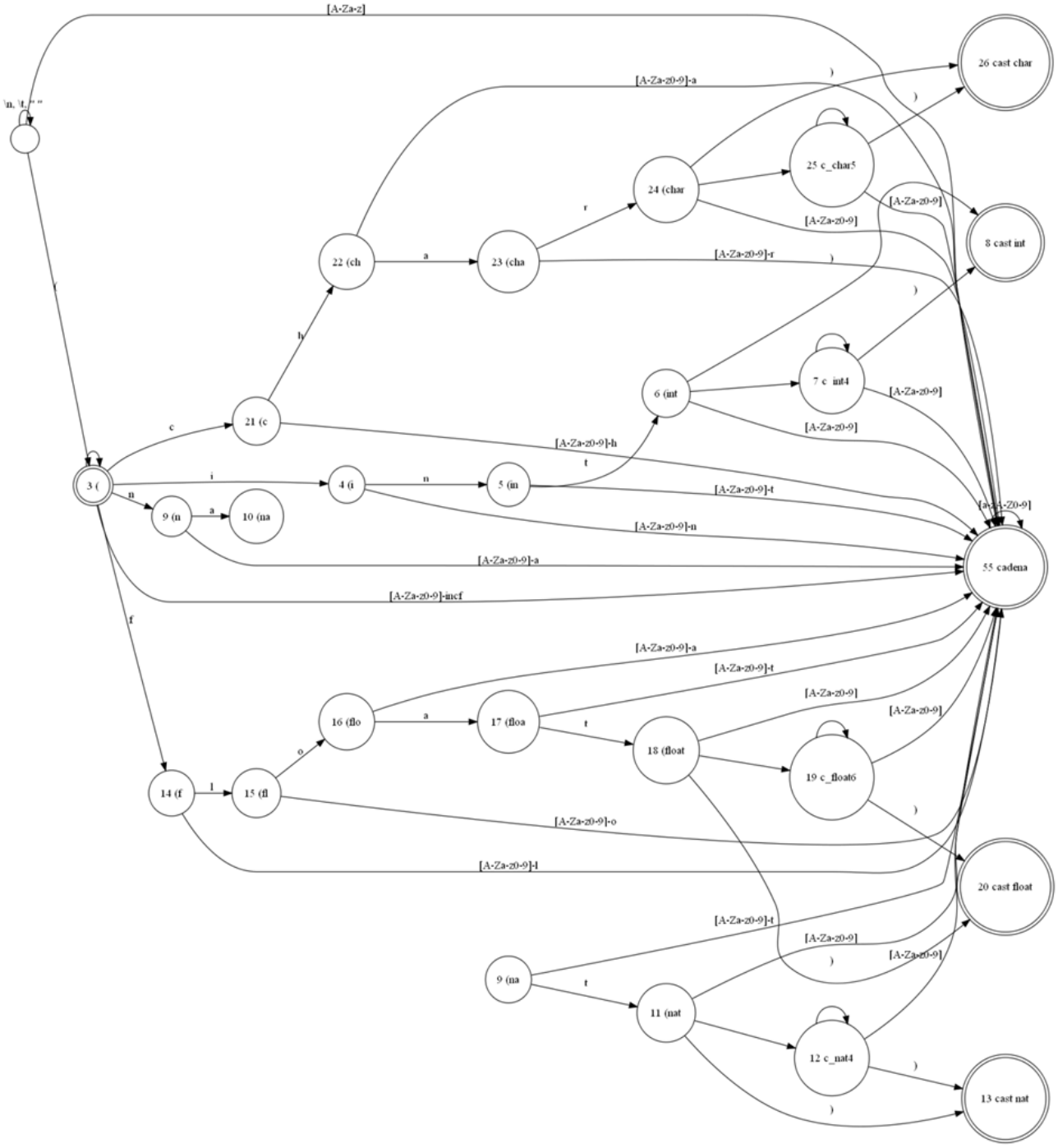
```
    ExpresionNiv30.cod  =
        case (OpNiv3.op)
            shl:
                ExpresionNiv4.cod || ExpresionNiv31.cod || shl
```

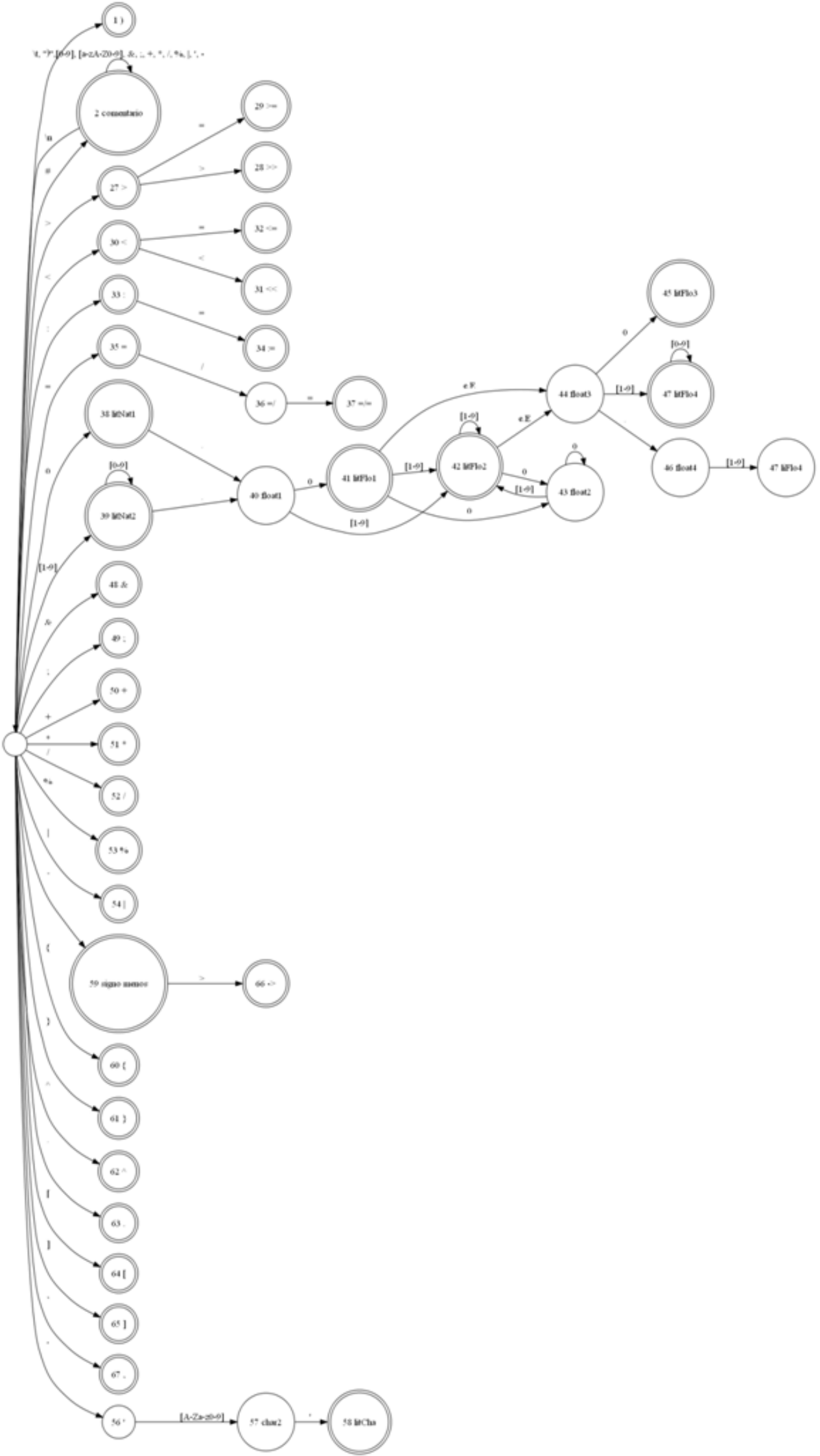
```
            shr:
                ExpresionNiv4.cod || ExpresionNiv31.cod || shr
    ExpresionNiv30.modo = val
    ExpresionNiv4.etqh  = ExpresionNiv30.etqh
    ExpresionNiv31.etqh = ExpresionNiv4.etq
    ExpresionNiv30.etqh = ExpresionNiv31.etq + 1
    ExpresionNiv4.parh  = ExpresionNiv31.parh = false

ExpresionNiv3 → ExpresionNiv4
    ExpresionNiv4.etqh = ExpresionNiv3.etqh
    ExpresionNiv3.etq  = ExpresionNiv4.etq
    ExpresionNiv3.cod  = ExpresionNiv4.cod
    ExpresionNiv3.modo = ExpresionNiv4.modo
    ExpresionNiv4.parh = ExpresionNiv3.parh

ExpresionNiv4 → OpNiv4 ExpresionNiv4
    ExpresionNiv40.cod =
        case (OpNiv4.op)
            no:
                ExpresionNiv41.cod || no
            negativo:
                ExpresionNiv41.cod || negativo
            cast-float:
                ExpresionNiv41.cod || CastFloat
            cast-int:
                ExpresionNiv41.cod || CastInt
            cast-nat:
                ExpresionNiv41.cod || CastNat
            cast-char:
                ExpresionNiv41.cod || CastChar
    ExpresionNiv40.modo = val
    ExpresionNiv41.etqh = ExpresionNiv4.etqh
    ExpresionNiv40.etq  = ExpresionNiv41.etq + 1
    ExpresionNiv41.parh = false

ExpresionNiv4 → | Expresion |
    ExpresionNiv4.cod  = Expresion.cod || abs
    ExpresionNiv4.modo = Expresion.val
    Expresion.etqh     = ExpresionNiv4.etqh
    ExpresionNiv4.etq  = Expresion.etq
    Expresion.parh     = false

ExpresionNiv4 → ( Expresion )
    ExpresionNiv4.cod  = Expresion.cod
    ExpresionNiv4.modo = Expresion.modo
    Expresion.etqh     = ExpresionNiv4.etqh
    ExpresionNiv4.etq  = Expresion.etq
    Expresion.parh     = ExpresionNiv4.parh

ExpresionNiv4 → Literal
    ExpresionNiv4.cod  = Literal.cod
    ExpresionNiv4.modo = var
    Literal.etqh = ExpresionNiv4.etqh
```

```
    ExpresionNiv4.etq  = Literal.etq

ExpresionNiv4 → Mem
    ExpresionNiv4.cod    =
        si esCompatibleConTipoBasico(Mem.tipo,ExpresionNiv4.tsh) /¥ not ExpresionNiv4.parh
            Mem.cod || apila-ind
        si no
            Mem.cod
    Mem.etqh        = ExpresionNiv4.etqh
    ExpresionNiv4.etq     =
        si esCompatibleConTipoBasico(Mem.tipo,ExpresionNiv4.tsh) /¥ not ExpresionNiv4.parh
            Mem.etq + 1
        si no
            Mem.etq
    ExpresionNiv4.modo    = var


Literal → litNat
    Literal.cod = apila LitNat.lex
    Literal.etq = Literal.etqh + 1

Literal → litFlo
    Literal.cod = apila litFlo.lex
    Literal.etq = Literal.etqh + 1

Literal → litTrue
    Literal.cod = apila true
    Literal.etq = Literal.etqh + 1

Literal → litFalse
    Literal.cod = apila false
    Literal.etq = Literal.etqh + 1

Literal → litCha
    Literal.cod = apila litCha.lex
    Literal.etq = Literal.etqh + 1
Literal → litNull
    Literal.cod = apila MIN_INT
    Literal.etq = Literal.etqh + 1
```

6. Diseño del analizador léxico

Diagrama de transición que caracterice el diseño del analizador léxico. La implementación del analizador léxico debe estar guiada por este diseño.




1)
\t, "\r", [0-9], [a-zA-Z0-9], &, _, +, *, /, %, |, ', -
2 comunque
29 >=
28 >>
27 >
32 <=
31 <<
30 <
34 :=
33 :
35 =
36 =/
37 =:=
38 litNat1
39 litNat2
40 float1
41 litFlo1
42 litFlo2
43 float2
44 float3
45 litFlo3
47 litFlo4
46 float4
47 litFlo4
48 &
49 ;
50 +
51 *
52 /
53 %
54 |
59 ogno numero
66 <>
60 (
61)
62 ^
63 .
64 [
65]
67 ,
56 '
57 char2
58 litCha
[0-9]
[1-9]
e.E
e.E
0

7. Acondicionamiento de las gramáticas de atributos

A continuación se presentan los acondicionamientos de las construcciones gramáticas de los puntos 2,3,4 que los necesitan, ya sea por ser recurrentes a izquierdas o por necesitar una factorización. Los acondicionamientos se han realizado ciñiéndonos a los esquemas proporcionados, incluso cuando no fuese estrictamente necesario, para facilitar una mejor compresión en el futuro y mantener una cierta coherencia entre la forma de proceder para las diferentes partes de las gramáticbahas.

7.1. Acondicionamiento de la Gramática para la Construcción de la tabla de símbolos

Declaraciones → Declaracion DeclaracionesRec

```
Declaracion.nh                  = Declaraciones.nh
DeclaracionesRec.nh = Declaraciones.nh
DeclaracionesRec.tsh        =
        inserta(Declaraciones.tsh,Declaracion.id, Declaracion.props)
Declaraciones.ts = DeclaracionesRec.ts
Declaraciones.n             = DeclaracionesRec.n
```

DeclaracionesRec → ';' Declaracion DeclaracionesRec

```
DeclaracionesRec1.tsh       =
        inserta(DeclaracionesRec0.tsh, Declaracion.id, Declaracion.props)
DeclaracionesRec0.ts        = DeclaracionesRec1.ts
Declaracion.nh                      = DeclaracionesRec0.nh
DeclaracionesRec1.nh        = max (DeclaracionesRec0.nh, Declaracion.n)
DeclaracionesRec0.n = DeclaracionesRec1.n
```

DeclaracionesRec → λ

```
DeclaracionesRec.ts = DeclaracionesRec.tsh
DeclaracionesRec.n          = DeclaracionesRec.nh
```

DeclaracionProcedimiento → procedure id FParametros DeclaracionProcFact

```
Fparametros.tsh = creaTS(DeclaracionProcedimiento.tsh)
Fparametros.nh = DeclaracionProcedimiento.nh +1

DeclaracionProcFact.nh = DeclaracionProcedimiento.nh +1
DeclaracionProcFact.tsh = FParametros.ts
DeclaracionProcFact.idh = id.lex
DeclaracionProcFact.paramsh = FParametros.parametros

DeclaracionProcedimiento.id= id.lex
DeclaracionProcedimiento.props    = DeclaracionProcFact.props
```

DeclaracionProcFact → Bloque

```
Bloque.tsh = inserta(DeclaracionProcFact.tsh, DeclaracionProcFact.idh,
                                    DeclaracionProcFact.propsh;
Bloque.nh = DeclaracionProcFact.nh

DeclaracionProcFact.props =
        <clase:proc, tipo: <t:proc, params: DeclaracionProcFact.paramsh>,
        nivel: DeclaracionProcFact.nh>
```

```
DeclaracionProcFact → Forward
        DeclaracionProcFact.props =
                <clase:forward, tipo: <t:proc, params: DeclaracionProcFact.paramsh>,
                nivel: DeclaracionProcFact.nh>

LFParametros → FParametro LFParametrosRec
        LFParametrosRec.tsh         =
                inserta(LFParametros.tsh, FParametro.id, FParametro.props)
        LFParametros.ts                     = LFParametrosRec.ts
        FParametro.nh                               = LFParametros.nh
        LFParametrosRec.nh                  = LFParametros.nh
        LFParametrosRec.parametrosh= FParametro.parametro
        LFParametros.parametros     = LFParametrosRec.parametros

LFParametrosRec → ',' FParametro LFParametrosRec
        LFParametrosRec1.tsh        =
                inserta(LFParametrosRec0.tsh, Fparametro.id, FParametro.props)
        LFParametrosRec0.ts  = LFParametrosRec1.ts
        FParametro.nh                       = LFParametrosRec0.nh
        LFParametrosRec1.nh  = LFParametrosRec0.nh
        LFParametrosRec1.parametrosh        =
                LFParametrosRec0.parametrosh ++ FParametro.parametro
        LFParametrosRec0.parametros         = LFParametrosRec1.parametros

LFParametrosRec → λ
        LFParametrosRec.ts                  = LFParametrosRec.tsh
        LFParametrosRec.parametros = LFParametrosRec.parametrosh

Campos → Campo CamposRec
        CamposRec.camposh    = [Campo.campo]
        Campos.campos                = CamposRec.campos
        Campo.tsh                    = Campos.tsh
        CamposRec.tsh        = Campos.tsh

CamposRec → ';' Campo CamposRec
        CamposRec1.camposh   = CamposRec0.camposh ++ Campo.campo
        CamposRec0.campos    = CamposRec1.campos
        Campo.tsh                    = CamposRec0.tsh
        CamposRec1.tsh               = CamposRec0.tsh

CamposRec → λ
        CamposRec.campos     = CamposRec.camposh
```

7.2. Acondicionamiento de la Gramática para la Comprobación de las Restricciones Contextuales

```
Declaraciones → Declaracion DeclaracionesRec
        DeclaracionesRec.errorh     =
                Declaracion.error  v
                (existeID(Declaraciones.tsh,Declaracion.id)  ∧
                        Declaraciones.tsh[Declaracion.id].nivel = Declaraciones.nh)
        DeclaracionesRec.pendh      =
```

```
            Declaracion.pend -
                  (si (Declaracion.props.clase = tipo) {Declaracion.id} sino {}
      DeclaracionesRec.forwardh = Declaracion.forward
      Declaraciones.error       = DeclaracionesRec.error
      Declaraciones.pend              = DeclaracionesRec.pend
      Declaraciones.forward     = DeclaracionesRec.forward
```

DeclaracionesRec → ';' Declaracion DeclaracionesRec

```
      DeclaracionesRec1.errorh   =
            DeclaracionesRec0.errorh v
            Declaracion.error v
            (existeID(DeclaracionesRec0.tsh,Declaracion.id)  ∧
                  DeclaracionesRec0.tsh[Declaracion.id].nivel=DeclaracionesRec0.nh)
      DeclaracionesRec1.pendh    =
            DeclaracionesRec0.pendh U
            Declaracion.pend -
            (si (Declaracion.props.clase = tipo) {Declaracion.id} sino {}

      DeclaracionesRec1.forwardh =     DeclaracionesRec0.forwardh U
                                       Declaracion.forward -
                                             si (Declaracion.props.clase == proc)
                                                   {Declaracion.id}

      DeclaracionesRec0.error    = DeclaracionesRec1.error
      DeclaracionesRec0.pend     = DeclaracionesRec1.pend
      DeclaracionesRec0.forward = DeclaracionesRec1.forward
```

DeclaracionesRec → λ

```
      DeclaracionesRec.error     = DeclaracionesRec.errorh
      DeclaracionesRec.pend            = DeclaracionesRec.pendh
      DeclaracionesRec.forward = DeclaracionesRec.forwardh
```

DeclaracionProcedimiento → procedure id FParametros DeclaracionProcFact

```
      DeclaracionProcFact.idh = id.lex
      DeclaracionProcFact.errorh = FParametros.error
      DeclaracionProcFact.tsh = FParametros.ts
      DeclaracionProcFact.nh = DeclaracionProcedimiento.nh + 1

      DeclaracionProcedimiento.pend = DeclaracionProcFact.pend
      DeclaracionProcedimiento.error = DeclaracionProcFact.error
      DeclaracionProcedimiento.forward = DeclaracionProcFact.forward
```

DeclaracionProcFact → forward

```
      DeclaracionProcFact.error =
            DeclaracionProcFact.errorh v
                  (existeID(DeclaracionProcFact.tsh,id.lex) ^
                        DeclaracionProcFact.tsh[DeclaracionProcFact.idh].nivel ==
                                          DeclaracionProcFact.nh + 1)
      DeclaracionProcFact.pend = {}
      DeclaracionProcFact.forward = {DeclaracionProcFact.idh}
```

DeclaracionProcFact → Bloque

```
      DeclaracionProcFact.error = DeclaracionProcFact.errorh v Bloque.error v
```

(existeID(DeclaracionProcFact.tsh,DeclaracionProcFact.idh)
^ DeclaracionProcFact.tsh[DeclaracionProcFact.idh].nivel==
DeclaracionProcFact.nh) ^
DeclaracionProcFact.tsh[DeclaracionProcFact.idh].props.t != forward)
DeclaracionProcFact.pend = Bloque.pend
DeclaracionProcFact.forward = {}

LFParametros → FParametro LFParametrosRec

LFParametrosRec.errorh =
Fparametro.error v
(exiteID(LFParametros.tsh, Fparametro.id) ∧
LFParametros.tsh[FParametro.id].nivel = LFParametros.nh)
LFParametros.error = LFParametrosRec.error

LFParametrosRec → ',' FParametro LFParametrosRec

LFParametrosRec1.errorh =
LFParametrosRec0.errorh v
FParametro.error v
(existeID(LFParametrosRec0.tsh,FParametro.id) ∧
LFParametrosRec0.tsh[FParametro.id].nivel = LFParametrosRec0.nh)
LFParametrosRec0.error = LFParametrosRec1.error

LFParametrosRec → λ

LFParametrosRec.error = LFParametros.errorh

Campos → Campo CamposRec

CamposRec.errorh = Campo.error
CamposRec.pendh = Campo.pend
Campos.error = CamposRec.error
Campos.pend = CamposRec.pend

CamposRec → ';' Campo CamposRec

CamposRec1.errorh =
CamposRec0.errorh v
existeCampo(CamposRec0.camposh, Campo.id)
CamposRec1.pendh = CamposRec0.pendh U Campo.pend
CamposRec0.pend = CamposRec1.pend
CamposRec0.error = CamposRec1.error

CamposRec → λ

CamposRec.error = CamposRec.error
CamposRec.pend = CamposRec.pendh

Instrucciones → Instruccion InstruccionesRec

Instruccion.tsh = Instrucciones.tsh
InstrucciónesRec.ts h = Instrucciones.tsh
InstruccionesRec.errorh= instrucción.error
Instrucciones.error = InstruccionesRec.error

InstruccionesRec → ; Instruccion InstruccionesRec

Instruccion.tsh = InstruccionesRec0.tsh
InstruccionesRec1.tsh = InstruccionesRec0.tsh
InstruccionesRec1.errorh =

```
            Instrucción.error v InstruccionesRec0.errorh
      InstruccionesRec0.error    = InstruccionesRec1.error
InstruccionesRec → λ
      InstruccionesRec.error     = InstruccionesRec.errorh

LAParametros → Expresion LAParametrosRec
      LAParametrosRec.nparametrosh      = 1
      Expresion.tsh                                  = LAParametros.tsh
      LAParametrosRec.tsh               = LAParametros.tsh
      LAParametrosRec.fparametrosh      = LAParametros.fparametrosh
      LAParametrosRec.errorh            =
            Expresion.error v
            |LAParametros.fparametrosh | < 1 v
            LAParametros.fparametrosh[0].modo = var ∧
                  Expresion.modo = val
            not compatibles(LAParametros.fparametrosh[0].tipo, Expresion.tipo,
      LAParametros.tsh)
      LAParametros.error = LAParametrosRec.error

LAParametrosRec → ',' Expresion LAParametrosRec
      LAParametrosRec1.nparametrosh     = LAParametrosRec0.nparametrosh + 1
      Expresion.tsh                                  = LAParametrosRec0.tsh
      LAParametrosRec1.tsh                     = LAParametrosRec0.tsh
      LAParametrosRec1.fparametrosh     = LAParametrosRec0.fparametrosh
      LAParametrosRec1.errorh           =
            LAParametrosRec0.errorh v
            Expresion.error v
            LAParameterosRec1.nparametrosh +1 > |LAParametros.fparametrosh|  v
      LAParametrosRec1.fparametrosh[LAParametrosRec1.nparametrosh].modo =
      var  ∧       Expresion.modo = val v
            not compatibles(
            LAParametrosRec1.fparametrosh[LAParametrosRec1.nparametrosh].tipo,
            Expresion.tipo, LAParametrosRec1.tsh
      LAParametrosRec0.error = LAParametrosRec1.error

LAParametrosRec → λ
      LAParametros.error                     = LAParametrosRec.errorh

Mem → id MemRec
      MemRec.tsh          = Mem.tsh
      MemRec.tipoh  =
            si existe(MemRec.tsh, id.lex)
                  si Mem.tsh[id.lex].clase = var
                        ref!(Mem.tsh[id.lex].tipo,Mem.tsh)
                  sino <t:error>
            sino
                  <t:error>
      Mem.tipo = MemRec.tipo

MemRec → ^ MemRec
      MemRec1.tsh   = MemRec0.tsh
      MemRec1.tipoh       =
            si (MemRec0.tipo.t = puntero
```

```
                ref!(MemRec0.tipo.tbase,MemRec0.tsh)
        sino
                <t:error>
    MemRec0.tipo  = MemRec1.tipo

MemRec → [Expresion] MemRec
    MemRec1.tsh   = MemRec0.tsh
    MemRec1.tipoh       =
        si MemRec0.tipo.t = array AND Expresion.tipo.t = num
                ref!(MemRec0.tipo.tbase,MemRec0.tsh)
        sino
                <t:error>
    MemRec0.tipo  = MemRec1.tipo

MemRec → .id MemRec
    MemRec1.tsh   = MemRec0.tsh
    MemRec1.tipoh       =
        si MemRec0.tipo.t = rec
                si campo?(MemRec0.tipo.campos, id.lex)
                        ref!(MemRec0.tipo.campos[id.lex].tipo, MemRec0.tsh)
                sino
                        <t:error>
    MemRec0.tipo = MemRec1.tipo

MemRec → λ
    MemRec.tipo = MemRec.tipoh

Expresion → ExpresiónNiv1 ExpresiónFact
    ExpresionFact.tipoh = ExpresionNiv1.tipo
    ExpresionNiv1.tsh   = Expresion.tsh
    ExpresionFact.tsh   = Expresion.tsh
    Expresion.tipo      = ExpresionFact.tipo
    Expresion.modo      = ExpresionFact.modo
    ExpresionFact.modoh = ExpresionNiv1.modo

ExpresionFact → OpNiv0 ExpresionNiv1
    ExpresionNiv1.tsh          = ExpresionFact.tsh
    ExpresionFact.tipo         =
        si     (ExpresionFact.tipoh = <t:error> v
               ExpresionNiv1.tipo = <t:error>) v
               (ExpresionFact.tipoh = <t:character> v
               ExpresionNiv1.tipo =/= <t:character>) v
               ( ExpresionFact.tipoh =/= <t:character> ∧
                       ExpresionNiv1.tipo = <t:character>)
               (ExpresionFact.tipoh = <t:boolean> ∧
                       ExpresionNiv1.tipo =/= <t:boolean>) v
               (ExpresionFact.tipoh =/= <t:boolean> ∧
                       ExpresionNiv1.tipo = <t:boolean>))
                       <t:error>
        sino
               <t:boolean>
    ExpresionFact.modo         = val
```

ExpresionFact → λ

```
ExpresionFact.modo         = ExpresionFact.modoh
ExpresionFact.tipo         = ExpresionFact.tipoh
```

ExpresiónNiv1 → ExpresiónNiv2 ExpresionNiv1Rec

```
ExpresionNiv2.tsh          = ExpresionNiv1.tsh
ExpresionNiv1Rec.tsh       = ExpresionNiv1.tsh
ExpresionNiv1Rec.tipoh = ExpresiónNiv2.tipo.
ExpresionNiv1Rec.modoh = ExpresionNiv2.modo
ExpresionNiv1.tipo         = ExpresiónNiv1Rec.tipo
ExpresionNiv1.modo         = ExpresionNiv1Rec.modo
```

ExpresiónNiv1Rec → OpNiv1 ExpresiónNiv2 ExpresiónNiv1Rec

```
ExpresionNiv2.tsh                 = ExpresionNiv1Rec0.tsh
ExpresionNiv1Rec1.tsh             = ExpresionNiv1Rec0.tsh
ExpresionNiv1Rec1.modoh    = val
ExpresionNiv1Rec1.tipoh    =
      si    (ExpresionNiv1Rec0.tipoh = <t:error> v
            ExpresionNiv2.tipo = <t:error> v
            ExpresionNiv1Rec0.tipoh = <t:character> v
            ExpresionNiv2.tipo = <t:character> v
            (ExpresionNiv1Rec0.tipoh = <t:boolean> ∧
                  ExpresionNiv2.tipo =/= <t:boolean>) v
            (ExpresionNiv1Rec0.tipoh =/= <t:boolean> ∧
                  ExpresionNiv2.tipo = <t:boolean>))
            <t:error>
      sino
            case (OpNiv1.op)
                  suma,resta:
                        si    (ExpresionNiv1Rec0.tipoh=<t:float> v
                              ExpresionNiv2.tipo = <t:float>)
                                    <t:float>
                        sino  si    (ExpresionNiv1Rec0.tipoh = <t:integer> v
                                    ExpresionNiv2.tipo = <t:integer>)
                                          <t:integer>
                  sino  si    (ExpresionNiv1Rec0.tipoh = <t:natural> ∧
                              ExpresionNiv2.tipo = <t:natural>)
                                          <t:natural>
                  sino
                              <t:error>
                  o:
                        si    (ExpresionNiv1Rec0.tipoh = <t:boolean> ∧
                              ExpresionNiv2.tipo = <t:boolean>)
                              <t:boolean>
                   sino
                              <t:error>
                  or:
                        si    (ExpresionNiv1Rec0.tipoh = <t:boolean> y
                              ExpresionNiv2.tipo = <t:boolean>)
                              <t:boolean>
                        sino
                              <t:error>
ExpresionNiv1Rec0.tipo            = ExpresionNiv1Rec1.tipo
```

ExpresionNiv1Rec → λ

ExpresionNiv1Rec.tipo = ExpresionNiv1Rec.tipoh
ExpresionNiv1Rec.modo = ExpresionNiv1Rec.modoh

ExpresiónNiv2 → ExpresionNiv3 ExpresiónNiv2Rec

ExpresionNiv2Rec.tipoh = ExpresiónNiv3.tipo
ExpresionNiv3.tsh = ExpresionNiv2.tsh
ExpresionNiv2Rec.tsh = ExpresionNiv2.tsh
ExpresionNiv2Rec.modoh = ExpresionNiv3.modo
ExpresionNiv2.tipo = ExpresiónNiv2Rec.tipo

ExpresiónNiv2Rec → OpNiv2 ExpresiónNiv3 ExpresiónNiv2Rec

$ExpresionNiv2Rec_1$.tsh = $ExpresionNiv2Rec_0$.tsh
ExpresionNiv3.tsh = $ExpresionNiv2Rec_0$.tsh
ExpresionNiv2Rec1.modoh = val
$ExpresionNiv2Rec_1$.tipoh =
si ExpresionNiv2Rec_0.tipoh = <t:error> v
ExpresionNiv3.tipo = <t:error> v
ExpresionNiv2Rec_0.tipoh = <t:character> v
ExpresionNiv3.tipo = <t:character> v
(ExpresionNiv2Rec_0.tipoh = <t:boolean> ∧
ExpresionNiv3.tipo =/= <t:boolean> v
(ExpresionNiv2Rec_0.tipoh =/= <t:boolean> ∧
ExpresionNiv3.tipo = <t:boolean>))
<t:error>
sino
case (OpNiv2.op)
multiplica,divide:
si (ExpresionNiv2Rec_0.tipoh=<t:float> v
ExpresionNiv3.tipo = <t:float>)
<t:float>
sino si (ExpresionNiv2Rec_0.tipoh = <t:integer> v
ExpresionNiv3.tipo = <t:integer>)
<t:integer
sino si (ExpresionNiv2Rec_0.tipoh = <t:natural> ∧
ExpresionNiv3.tipo= <t:natural>)
<t:natural>
sino
<t:error>
modulo:
si (ExpresionNiv3.tipo = <t:natural> ∧
(ExpresionNiv2Rec_0.tipoh= <t:natural> v
ExpresionNiv2Rec0.tipoh= <t:integer>))
ExpresionNiv2Rec_0.tipoh
sino
<t:error>
y:
si ExpresionNiv2Rec_0.tipoh = <t:boolean> ∧
ExpresionNiv3.tipo = <t:boolean>)
<t:boolean>
sino
<t:error>

```
                        and:
                              si     (ExpresionNiv2Rec0.tipoh = <t:boolean> y
                                     ExpresionNiv3.tipo = <t:boolean>
                                     <t:boolean>
                              sino
                                     <t:error>
      ExpresionNiv2Rec0.tipo         = ExpresionNiv2Rec1.tipo


ExpresiónNiv2Rec → λ
      ExpresionNiv2Rec.tipo          = ExpresionNiv2Rec.tipoh
      ExpresionNiv2Rec.modo          = ExpresionNiv2Rec.modh



(factorizamos)
ExpresionNiv3 → ExpresionNiv4 ExpresionNiv3Fact
      ExpresiónNiv4.tsh        = ExpresiónNiv3.tsh
      ExpresiónNiv3Fact.tsh    = ExpresiónNiv3.tsh
      ExpresionNiv3Fact.tipoh  = ExpresiónNiv4.tipo
      ExpresionNiv3Fact.modoh  = ExpresionNiv4.modo
      ExpresionNiv3.tipo       = ExpresionNiv3Fact.tipo

ExpresionNiv3Fact → OpNiv3 ExpresiónNiv3
      ExpresiónNiv3.tsh        = ExpresiónNiv3Fact.tsh
      ExpresionNiv3Fact.modo   = val
      ExpresionNiv3Fact.tipo   =
            si     (ExpresionNiv3Fact.tipoh = <t:error> v
                   ExpresionNiv3.tipo = <t:error> v
                   ExpresionNiv3Fact.tipoh =/= <t:natural> v
                   ExpresionNiv3.tipo =/= <t:natural>)
                   <t:error>
            sino
                   <t:natural>
ExpresionNiv3Fact → λ
      ExpresionNiv3Fact.tipo   = ExpresionNiv3Fact.tipoh
      ExpresionNiv3Fact.modo   = ExpresionNiv3Fact.modoh
```

7.3. Acondicionamiento de la Gramática para la Traducción

```
Declaraciones → Declaracion DeclaracionesRec
      Declaracion.etqh         = Declaraciones.etqh
      Declaracion.dirh         = Declaraciones.dir
      Declaracion.nh                 = Declaraciones.nh
      DeclaracionesRec.nh = Declaraciones.nh
      DeclaracionesRec.tsh     =
            inserta(Declaraciones.tsh,Declaracion.id, Declaracion.props)
      DeclaracionesRec.codh    = Declaracion.cod
      DeclaracionesRec.dirh    = Declaraciones.dirh + Declaracion.tam
      DeclaracionesRec.etqh    = Declaracion.etq
      Declaraciones.n          = DeclaracionesRec.n
      Declaraciones.dir        = DeclaracionesRec.dir
```

```
        Declaraciones.etq          = DeclaracionesRec.etq
        Declaraciones.ts           = DeclaracionesRec.ts
        Declaraciones.cod          = DeclaracionesRec.cod

DeclaracionesRec → ';' Declaracion DeclaracionesRec
        Declaracion.nh             = DeclaracionesRec0.nh
        Declaracion.etqh           = DeclaracionesRec0.etqh
        Declaracion.dir            = DeclaracionesRec0.dirh
        DeclaracionesRec1.etqh = Declaracion.etq
        DeclaracionesRec1.nh       = max (DeclaracionesRec0.nh, Declaracion.n)
        DeclaracionesRec1.dirh     = DeclaracionesRec0.dirh + Declaracion.tam
        DeclaracionesRec1.tsh      =
               inserta(DeclaracionesRec0.tsh, Declaracion.id, Declaracion.props)
        DeclaracionesRec1.codh     = DeclaracionesRec0.codh || Declaracion.cod
        DeclaracionesRec0.n        = DeclaracionesRec1.n
        DeclaracionesRec0.etq      = DeclaracionesRec1.etq
        DeclaracionesRec0.dir      = DeclaracionesRec1.dir
        DeclaracionesRec0.cod      = DeclaracionesRec1.cod
        DeclaracionesRec0.ts       = DeclaracionesRec1.ts

DeclaracionesRec → λ
        DeclaracionesRec.n         = DeclaracionesRec.nh
        DeclaracionesRec.etq       = DeclaracionesRec.etqh
        DeclaracionesRec.dir       = DeclaracionesRec.dirh
        DeclaracionesRec.ts = DeclaracionesRec.tsh
        DeclaracionesRec.cod       = DeclaracionesRec.codh

DeclaracionProcedimiento → proc id Fparametros DeclaracionProcFact
        FParametros.dirh = DeclaracionProcedimiento.dirh
        FParametros.nh = DeclaracionProcedimiento.nh + 1
        DeclaracionProcFact.dirh = FParametros.dir
        DeclaracionProcFact.nh = DeclaracionProcedimiento.nh +1
        DeclaracionProcFact.etq = DeclaracionProcedimiento.etqh

        DeclaracionProcedimiento.n = DeclaracionProcFact.n
        DeclaracionProcedimiento.dir = DeclaracionProcFact.dir
        DeclaracionProcedimiento.etq =    DeclaracionProcFact.etq

DeclaracionProcFact → Bloque
        Bloque.dirh = DeclaracionProcFact.dirh
        Bloque.nh = DeclaracionProcFact.nh
        Bloque.etqh =       DeclaracionProcFact.etqh
        Bloque.tsh = inserta(DeclaracionProcFact.tsh,
                             DeclaracionProcFact.idh,
                             <inicio:Bloque.inicio>)
        DeclaracionProcFact.props = <inicio:Bloque.inicio>
        DeclaracionProcFact.n = Bloque.n
        DeclaracionProcFact.cod = Bloque.cod
        DeclaracionProcFact.etq = Bloque.etq

DeclaracionProcFact → forward
        DeclaracionProcFact.n =    DeclaracionProcFact.nh
        DeclaracionProcFact.props = <>
```

```
        DeclaracionProcFact.cod = {}
        DeclaracionProcFact.etq =  DeclaracionProcFact.etqh

Campos → Campo CamposRec
        Campo.desh          = 0
        CamposRec.desh      = Campo.tam
        Campos.tam          = CamposRec.tam

CamposRec → ';' Campo CamposRec
        Campo.desh          = CamposRec0.desh
        CamposRec1.desh     = Campo.tam + CamposRec0.desh
        CamposRec0.tam      = CamposRec1.tam

CamposRec → λ
        CamposRec.tam       = CamposRec.desh

LFParametros -> FParametro LFParametrosRec
        LFParametrosRec.tsh =
                añadeID(LFParametros.tsh,FParametro.id, FParametro.props ++ <dir:0>)
        LFParametrosRec.dirh        = FParametro.tam.
        LFParametros.ts             = LFParametrosRec.ts
        LFParametros.dir            = LFParametrosRec.dir
        Fparametro.dirh                     = 0

LFParametrosRec -> FParametro , LFParametrosRec
        FParametro.dirh             = LFParametrosRec0.dirh
        LFParametrosRec1.dirh       = LFParametrosRec0.dirh + FParametro.tam
        LFParametrosRec1.tsh        =
                añadeID(      LFParametrosRec0,tsh,
                                FParametro.id,
                                FParametro.props ++ <dir:LFParametrosRec0.dirh>)
        LFParametrosRec0.ts  = LFParametrosRec1.ts
        LFParametrosRec0.dir        = LFPParametrosRec1.dir

LFParametrosRec -> λ
        LFParametrosRec.dir         = LFParametrosRec.dirh
        LFParametrosRec.ts          = LFParametrosRec.tsh

Instrucciones -> Instruccion InstruccionesRec
        Instruccion.etqh            = Instrucciones.etqh
        InstruccionesRec.etqh       = Instruccion.etq
        InstruccionesRec.codh       = Instruccion.cod
        Instrucciones.etq           = InstruccionesRec.etq
        Instrucciones.cod           = InstruccionesRec.cod

InstruccionesRec -> ; Instruccion InstruccionesRec
        Instruccion.etqh            = InstruccionesRec0.etqh
        InstruccionesRec1.etqh      = Instruccion.etq
        InstruccionesRec1.codh      = InstruccionesRec0.cod | | Instruccion.cod
        InstruccionesRec0.cod       = InstruccionesRec1.cod
        InstruccionesRec0.etq       = InstruccionesRec1.etq

InstruccionesRec -> λ
```

```
        InstruccionesRec.etq= InstruccionesRec.etqh
        InstruccionesRec.cod       = InstruccionesRec.codh


LAParametros → Expresion LAParametrosRec
        Expresion.etqh = LAParametros.etqh + 1
        Expresion.parh =
                LAParametros.fparametrosh[LAParametros.nparametrosh].modo == var
        LAParametrosRec.codh =
                copia ||
                Expresion.cod ||
                pasoParametro(Expresion.modo,LAParametros.fparametrosh[0])
        LAParametrosRec.etqh = Expresion.etq + longPaseoParametro

        LAParametros.cod = LAParametrosRec.cod
        LAParametros.etq = LAParametrosRec.etq

LAParametrosRec → ',' Expresion LAParametrosRec
        Expresion.etqh = LAParametrosRec0.etqh + 1 + longDireccionParFormal
        Expresion.parh =

        LAParametrosRec1.fParametrosh[LAParametros1Rec.nParametrosh].modo == var
        LAParametrosRec1.etqh = Expresion.etq + longPasoParametro
        LAParametrosRec1.codh =
                LAParametrosRec0.codh ||
                copia ||
direccionParFormal(LAParametrosRec0fparametrosh[LAParametrosRec0.nparametrosh]) ||
                Expresion.cod ||
                pasoParametro(      Expresion.modo,

        LAParametrosRec0.fparametrosh[LAParametrosRec0.nparametrosh])
        LAParametrosRec0.etq = LAParametrosRec1.etq
        LAParametrosRec0.cod = LAParametrosRec1.cod

LAParametrosRec → λ
        LAParametrosRec.cod = LAParametrosRec.codh
        LAParametrosRec.etq = LAParametrosRec.etqh


Mem → id MemRec
        MemRec.etqh   = logAcessoVar(Mem.tsh[id.lex]) + Mem.etqh
        Mem.etq       = MemRec.etq
        MemRec.codh   = accesoVar(Mem.tsh[id.lex])
        Mem.cod       = MemRec.cod

MemRec → ^ MemRec
        MemRec1.etqh  = MemRec0.etqh + 1
        MemRec0.codh  = MemRec0.codh || apila-ind
        MemRec0.etq   = MemRec1.etq
        Memrec0.cod   = MemRec1.cod

MemRec → [Expresion] MemRec
        Expresion.etq       = MemRec0.etqh
```

```
        MemRec1.etqh  = Expresion.etq + 3
        MemRec1.codh  =
                MemRec0.codh ||
                Exp.cod ||
                apila Mem1.tipo.tbase.tam ||
                multiplica ||
                suma
        MemRec0.etq   = MemRec1.etq
        Memrec0.cod   = MemRec1.cod

MemRec → .id MemRec
        MemRec1.etqh  = MemRec0.etqh + 2
        MemRec0.etq   = MemRec1.etq
        MemRec1.codh  =
                MemRec0.codh ||
                apila(MemRec0.tipo.campos[id.lex].desp ||
                suma
        Memrec0.cod   = MemRec1.cod

MemRec → λ
        MemRec.etq = MemRec.etqh
        MemRec.cod = MemRec.codh

Expresion -> ExpresionNiv1 ExpresionFact
        ExpresionNiv1.etqh = Expresion.etqh
        ExpresionNiv1.parh = Expresion.parh
        ExpresionFact.etqh = ExpresionNiv1.etq
        ExpresionFact.codh = ExpresionNiv1.cod
        ExpresionFact.modoh = ExpresionNiv1.modo
        Expresion.cod = ExpresionFact.cod
        Expresion.etq = ExpresionFact.etq
        Expresion.modo = ExpresionFact.modo

ExpresionFact-> OpNiv0 ExpresionNiv1
        ExpresionNiv1.parh= false
        ExpresionNiv1.etqh = ExpresionFact.etqh
        ExpresionFact.modo = val
        ExpresionFact.cod =
                ExpresionFact.codh ||
                ExpresionNiv1.cod
                case (OpNiv0)
                        menor: menor
                        mayor: mayor
                        menor-ig: menorIg
                        mayor-ig: mayorIg
                        igual : igual
                        no-igual: no-igual
        ExpresionFact.etq = ExpresionNiv1.etq + 1

ExpresionFact -> λ
        ExpresionFact.cod = ExpresionFact.codh
        ExpresionFact.modo = ExpresionFact.modoh
        ExpresionFact.etq = ExpresionFact.etqh
```

```
ExpresionNiv1 -> ExpresionNiv2 ExpresionNiv1Rec
	ExpresionNiv2.etqh = ExpresionNiv1.etqh
	ExpresionNiv2.parh = ExpresionNiv1.parh
	ExpresionNiv1Rec.etqh = ExpresionNiv2.etq
	ExpresionNiv1Rec.modoh = ExpresionNiv2.modo
	ExpresionNiv1Rec.codh = ExpresionNiv2.cod
	ExpresionNiv1.cod = ExpresionNiv1Rec.cod
	ExpresionNiv1.etq = ExpresionNiv1Rec.etq
	ExpresionNiv1.modo = ExpresionNiv1Rec.modo

ExpresionNiv1Rec -> OpNiv1 ExpresionNiv2 ExpresionNiv1Rec
	ExpresionNiv2.parh = false
	ExpresionNiv1Rec1.modoh = val
	if (opNiv1.op == or):
		ExpresionNiv2.etqh = ExpresionNiv1Rec0.etqh + 3
		ExpresionNiv1Rec1.codh =
			ExpresionNiv1Rec0.codh ||
			dup ||
			ir-v (ExpresionNiv2.etq) ||
			desapila ||
			ExpresionNiv2.cod
		ExpresionNiv1Rec1.etqh = ExpresionNiv2.etq

	else:
		ExpresionNiv2.etqh = ExpresionNiv1Rec0.etqh
		ExpresionNiv1Rec1.etqh = ExpresionNiv2.etq + 1
		ExpresionNiv1Rec1.codh =
			ExpresionNiv1Rec0.codh ||
			ExpresionNiv2.cod ||
			case (OpNiv1.op)
				suma: sumar
				resta: restar

	ExpresionNiv1Rec0.etq = ExpresionNiv1Rec1.etq
	ExpresionNiv1Rec0.cod = ExpresionNiv1Rec1.cod
	ExpresionNiv1Rec0.modo = ExpresionNiv1Rec1.modo

ExpresionNiv1Rec ->  λ
	ExpresionNiv1Rec.etq = ExpresionNiv1Rec.etqh
	ExpresionNiv1Rec.cod = ExpresionNiv1Rec.codh
	ExpresionNiv1Rec.modo = ExpresionNiv1Rec.modoh

ExpresionNiv2 -> ExpresionNiv3 ExpresionNiv2Rec
	ExpresionNiv3.etqh = ExpresionNiv2.etqh
	ExpresionNiv3.parh = ExpresionNiv2.parh
	ExpresionNiv2Rec.etqh = ExpresionNiv3.etq
	ExpresionNiv2Rec.codh = ExpresionNiv3.cod
	ExpresionNiv2Rec.modoh = ExpresionNiv3.modo
	ExpresionNiv2.cod = ExpresionNiv2Rec.cod
	ExpresionNiv2.modo = ExpresionNiv2Rec.modo
	ExpresionNiv2.etq = ExpresionNiv2Rec.etq
```

```
ExpresionNiv2Rec -> OpNiv2 ExpresionNiv3 ExpresionNiv2Rec
	ExpresionNiv2Rec1.modoh= val
	if (OpNiv2.op == and)
		ExpresionNiv2Rec1.cod =
			ExpresionNiv2Rec0.codh ||
			ir-f(ExpresionNiv3.etq + 1) ||
			ExpresionNiv3.cod ||
			ir-a(ExpresionNiv3.etq + 2) ||
			apila(0)
		ExpresionNiv3.etqh = ExpresionNiv2Rec0.etqh + 1
		ExpresionNiv2Rec1.etqh = ExpresionNiv3.etq + 2

	else:
		ExpresionNiv2Rec1.cod =
			ExpresionNiv2Rec0.codh ||
			ExpresionNiv3.cod ||
			case (OpNiv2.op)
				Multiplica: mul
				Divide: div
				Modulo: Mod
		ExpresionNiv3.etqh = ExpresionNiv2Rec0.etqh
		ExpresionNiv3Rec1.etqh = ExpresionNiv3.etq + 1

	ExpresionNiv2Rec0.cod = ExpresionNiv2Rec1.cod
	ExpresionNiv2Rec0.modo = ExpresionNiv2Rec1.modo
	ExpresionNiv2Rec0.etq = ExpresionNiv2Rec1.etq

ExpresionNiv2Rec -> λ
	ExpresionNiv2Rec.modo = ExpresionNiv2Rec.modoh
	ExpresionNiv2Rec.cod = ExpresionNiv2Rec.codh
	ExpresionNiv2Rec.etq = ExpresionNiv2Rec.etqh



ExpresionNiv3 -> ExpresionNiv4 ExpresionNiv3Fact
	ExpresionNiv4.etqh = ExpresionNiv3.etqh
	ExpresionNiv4.parh = ExpresionNiv3.h
	ExpresionNiv3Fact.codh = ExpresionNiv4.cod
	Expresionniv3Fact.etqh = ExpresionNiv4.etq
	ExpresionNiv3Fact.modoh = ExpresionNiv4.modo

	ExpresionNiv3.etq = ExpresionNiv3Fact.etq
	ExpresionNiv3.cod = ExpresionNiv3Fact.cod

ExpresionNiv3Fact -> OpNiv3 ExpresionNiv3
	ExpresionNiv3.etqh = ExpresionNiv3Fact.etqh
	ExpresionNiv3.parh = false
	ExpresionNiv3Fact.cod =
		case (OpNiv3.op)
			shl: ExpresionNiv3Fact.codh || ExpresionNiv3.cod || shl
			shr: ExpresionNiv3Fact.codh || ExpresionNiv3.cod || shr
	ExpresionNiv3Fact.modo=val
```

```
        ExpresionNiv3Fact.etq = ExpresionNiv3.etq + 1

ExpresionNiv3Fact -> λ
        ExpresionNiv3Fact.modo = ExpresionNiv3Fact.modoh
        ExpresionNiv3Fact.cod = {}
        ExpresionNiv3Fact.etq = ExpresionNiv3Fact.etqh
```

8. Esquema de traducción orientado a las gramáticas de atributos

```
Programa →
     {
       Declaraciones.etqh  = longInicio + 1
       Declaraciones.dirh  = 0
       Declaraciones.nh    = 0
       Declaraciones.tsh   = creaTS()

     }
     Declaraciones
     {
     }
     &
     {
       Instrucciones.etqh  = Declaraciones.etq
       Instrucciones.tsh   = Declaraciones.ts
     }
     Instrucciones
     {
       Programa.ts   = Declaraciones.ts
       Programa.error      =
              Declaraciones.error v
              Instrucciones.error  v
              Declaraciones.pend != □ v
              Declaraciones.forward !=  □
              Programa.cod  =
                     inicio(Declaraciones.n,Declaraciones.dir) ||
                     ir-a(Declaraciones.etq)                   ||
                     Declaraciones.cod                         ||
                     Instrucciones.cod                         ||
              stop;
     }


Declaraciones →
     {
       Declaracion.nh = Declaraciones.nh
       Declaracion.etqh = Declaraciones.etqh
       Declaracion.dirh = Declaraciones.dirh
     }
     Declaracion
     {
              DeclaracionesRec.codh = Declaracion.cod
              DeclaracionesRec.dirh = Declaraciones.dirh + Declaracion.tam
```

```
            DeclaracionesRec.etqh = Declaracion.etq
            DeclaracionesRec.nh = Declaracion.n
            DeclaracionesRec.tsh  = inserta(Declaraciones.tsh,Declaracion.id,
                        Declaracion.props)
            DeclaracionesRec.errorh =
                  Declaracion.error  v
                  (existeID(Declaraciones.tsh,Declaracion.id)  ∧
                        Declaraciones.tsh[Declaracion.id].nivel = Declaraciones.nh)
            DeclaracionesRec.pendh =
                  Declaracion.pend -
                        si (Declaracion.props.clase = tipo) {Declaracion.id} sino {}
            DeclaracionesRec.forwardh = Declaracion.forward
      }
      DeclaracionesRec
      {
        Declaraciones.ts = DeclaracionesRec.ts
        Declaraciones.error = DeclaracionesRec.error
        Declaraciones.pend = DeclaracionesRec.pend
        Declaraciones.n = DeclaracionesRec.n
        Declaraciones.dir = DeclaracionesRec.dir
        Declaraciones.etq = DeclaracionesRec.etq
        Declaraciones.ts = DeclaracionesRec.ts
        Declaraciones.cod = DeclaracionesRec.cod
        Declaraciones.forward      = DeclaracionesRec.forward
      }



DeclaracionesRec → ';'
      {
        Declaracion.nh = DeclaracionesRec0.nh
        Declaracion.etqh = DeclaracionesRec0.etqh
        Declaracion.dir = DeclaracionesRec0.dirh
      }
      Declaracion
      {
        DeclaracionesRec1.tsh =
              inserta(DeclaracionesRec0.tsh, Declaracion.id, Declaracion.props)
        DeclaracionesRec1.etqh = Declaracion.etq
        DeclaracionesRec1.nh = max (DeclaracionesRec0.nh, Declaracion.n)
        DeclaracionesRec1.dirh = DeclaracionRec0.dirh + Declaracion.tam
        DeclaracionesRec1.tsh =
              inserta(DeclaracionesRec0.tsh, Declaracion.id,
                                    Declaracion.props ++ Declaracion.propsop)
        DeclaracionesRec1.codh = DeclaracionesRec0.codh || Declaracion.cod

        DeclaracionesRec1.errorh =
              DeclaracionesRec0.errorh v
              Declaracion.error v
              (existeID(DeclaracionesRec0.tsh,Declaracion.id)  ∧
                    DeclaracionesRec0.tsh[Declaracion.id].nivel =  DeclaracionesRec0.nh)
        DeclaracionesRec1.pendh =
              DeclaracionesRec0.pendh U
```

```
            Declaracion.pend -
                  si (Declaracion.props.clase = tipo) {Declaracion.id} sino {}
      DeclaracionesRec1.forwardh =
            DeclaracionesRec0.forwardh U
            Declaracion.forward -
                  si (Declaracion.props.t == proc) {Declaracion.id}

    }
    DeclaracionesRec
    {
      DeclaracionesRec0.ts = DeclaracionesRec1.ts
      DeclaracionesRec0.error = DeclaracionesRec1.error
      DeclaracionesRec0.pend = DeclaracionesRec1.pend
      DeclaracionesRec0.n = DeclaracionesRec1.n
      DeclaracionesRec0.etq = DeclaracionesRec1.etq
      DeclaracionesRec0.dir = DeclaracionesRec1.dir
      DeclaracionesRec0.cod = DeclaracionesRec1.cod
      DeclaracionesRec0.forward = DeclaracionesRec1.forward
    }


DeclaracionesRec → λ
    {
      DeclaracionesRec.ts = DeclaracionesRec.tsh
      DeclaracionesRec.error = DeclaracionesRec.errorh
      DeclaracionesRec.pend = DeclaracionesRec.pendh
      DeclaracionesRec.n = DeclaracionesRec.nh
      DeclaracionesRec.etq = DeclaracionesRec.etqh
      DeclaracionesRec.dir = DeclaracionesRec.dirh
      DeclaracionesRec.cod = DeclaracionesRec.codh
      DeclaracionesRec.forward = DeclaracionesRec.forwardh
    }



Declaracion →
      {
      DeclaracionTipo.tsh = Declaracion.tsh
      DeclaracionTipo.nh = Declaracion.nh
      }
      DeclaracionTipo
      {
      Declaracion.tam         = 0
      Declaracion.n           = Declaracion.nh
      Declaracion.cod         = λ
      Declaracion.etq         = Declaracion.etqh
      Declaracion.id = DeclaracionTipo.id
      Declaracion.props = DeclaracionTipo.props ++ ◇
      Declaracion.error  = DeclaracionTipo.error
      Declaracion.pend = DeclaracionTipo.pend
      Declaracion.forward = {}
      }
```

```
Declaracion →
    {
      DeclaracionVariable.tsh = Declaracion.tsh
      DeclaracionVariable.nh = Declaracion.nh
    }
    DeclaracionVariable
    {
      Declaracion.id = DeclaracionVariable.id
      Declaracion.props   = DeclaracionVariable.props ++ <dir:Declaracion.dirh>
      Declaracion.error   = DeclaracionVariable.error
      Declaracion.pend    = DeclaracionVariable.pend
      Declaracion.tam     = DeclaracionVariable.props.tipo.tam
      Declaracion.n       = Declaracion.nh
      Declaracion.cod     = λ
      Declaracion.etq     = Declaracion.etqh
      Declaracion.forward = {}
    }


DeclaracionProcedimiento →
      procedure id
      {
            Fparametros.tsh = creaTS(DeclaracionProcedimiento.tsh)
            Fparametros.nh = DeclaracionProcedimiento.nh +1
            FParametros.dirh = DeclaracionProcedimiento.dirh
      }
      FParametros
      {
            DeclaracionProcFact.nh = DeclaracionProcedimiento.nh +1
            DeclaracionProcFact.tsh = FParametros.ts
            DeclaracionProcFact.idh = id.lex
            DeclaracionProcFact.paramsh = FParametros.parametros
            DeclaracionProcFact.errorh = FParametros.error
            DeclaracionProcFact.dirh = FParametros.dir
            DeclaracionProcFact.etq = DeclaracionProcedimiento.etqh
      }
      DeclaracionProcFact
      {
            DeclaracionProcedimiento.id= id.lex
            DeclaracionProcedimiento.props    = DeclaracionProcFact.props
            DeclaracionProcedimiento.pend = DeclaracionProcFact.pend
            DeclaracionProcedimiento.error = DeclaracionProcFact.error
            DeclaracionProcedimiento.forward = DeclaracionProcFact.forward
            DeclaracionProcedimiento.n = DeclaracionProcFact.n
            DeclaracionProcedimiento.dir = DeclaracionProcFact.dir
            DeclaracionProcedimiento.etq =    DeclaracionProcFact.etq
            DeclaracionProcedimiento.cod = DeclaracionProcFact.cod
      }


DeclaracionProcFact →
      {
            Bloque.tsh = inserta(DeclaracionProcFact.tsh,
```

```
            DeclaracionProcFact.idh, DeclaracionProcFact.propsh+
                        +<inicio:Bloque.inicio>;
            Bloque.nh = DeclaracionProcFact.nh
            Bloque.dirh = DeclaracionProcFact.dirh
            Bloque.etqh =       DeclaracionProcFact.etqh
      }
      Bloque
      {
            DeclaracionProcFact.props =
                  <clase:proc, tipo: <t:proc, params: DeclaracionProcFact.paramsh>,
                  <inicio: Bloque.inicio>,
            nivel: DeclaracionProcFact.nh>

            DeclaracionProcFact.error =
                  DeclaracionProcFact.errorh v
                  Bloque.error v
                  (existeID(DeclaracionProcFact.tsh,DeclaracionProcFact.idh)
                        ^  DeclaracionProcFact.tsh[DeclaracionProcFact.idh].nivel==
                                                DeclaracionProcFact.nh) ^
            DeclaracionProcFact.tsh[DeclaracionProcFact.idh].props.clase != forward)
            DeclaracionProcFact.pend = Bloque.pend
            DeclaracionProcFact.forward = {}
            DeclaracionProcFact.n = Bloque.n
            DeclaracionProcFact.cod = Bloque.cod
            DeclaracionProcFact.etq = Bloque.etq
      }
DeclaracionProcFact → Forward
      {
            DeclaracionProcFact.props =
                  <clase:forward, tipo: <t:proc, params:
            DeclaracionProcFact.paramsh>,nivel: DeclaracionProcFact.nh>
            DeclaracionProcFact.error =
                  DeclaracionProcFact.errorh v
                  (existeID(DeclaracionProcFact.tsh,id.lex) ^
                        DeclaracionProcFact.tsh[DeclaracionProcFact.idh].nivel ==
                                                DeclaracionProcFact.nh + 1)
            DeclaracionProcFact.pend = {}
            DeclaracionProcFact.forward = {DeclaracionProcFact.idh}
            DeclaracionProcFact.n =    DeclaracionProcFact.nh
            DeclaracionProcFact.props = <>
            DeclaracionProcFact.cod = {}
            DeclaracionProcFact.etq =  DeclaracionProcFact.etqh
      }


DeclaracionTipo → tipo id =
    {
            Tipo.tsh = DeclaracionTipo.tsh
    }
    Tipo
    {
            DeclaracionTipo.id = id.lex
            DeclaracionTipo.props =
```

```
                <clase:tipo, tipo:DeclaracionTipo.tipo, nivel: DeclaracionTipo.nh>
        DeclaracionTipo.tipo = Tipo.tipo
        DeclaracionTipo.error =
                Tipo.error v
                existeID(DeclaracionTipo.tsh, id.lex) v
                        ¬existeRef(DeclaracionTipo.tsh,Tipo.tipo)
    DeclaracionTipo.pend = Tipo.pend
}


DeclaraciónVariable →
{
    Tipo.tsh = DeclaracionVariable.tsh
}
Tipo id
{
    DeclaracionVariable.id = id.lex
    DeclaracionVariable.props =
        <clase:var, tipo: DeclaracionVariable.tipo, nivel:DeclaracionVariable.nh>
    DeclaracionVariable.tipo = Tipo.tipo
    DeclaracionVariable.error = Tipo.error v existeID(DeclaracionVariable.tsh, id.lex) v

    ¬existeRef(DeclaracionVariable.tsh,Tipo.tipo)
    DeclaracionVariable.pend = Tipo.pend
}


DeclaracionProcedimiento → procedure id
{
    FParams.tsh = creaTS(DeclaracionProcedimiento.tsh)
      FParametros.nh = DeclaracionProcedimiento.nh +1
}
FParametros
{
    Bloque.tsh =
      inserta(FParametros.ts, id.lex, <clase:proc, tipo: <t:proc, params:
                Fparametros.parametros>, nivel: DeclaracionProcedimiento.nh + 1>)
      Bloque.nh = DeclaracionProcedimiento.nh +1
}
Bloque
{
    DeclaracionProcedimiento.error =
        FParametros.error v
        Bloque.error v
        (existeID(FParametros.ts, id.lex)  ∧
                FParametros.ts[id.lex].nivel = DeclaracionProcedimiento.nh +1)
    DeclaracionProcedimiento.pend = Bloque.pend
    DeclaracionProcedimiento.id = id.lex
    DeclaracionProcedimiento.props =
        <clase:proc, tipo: <t:proc, params: Fparametros.parametros>,
                nivel: DeclaracionProcedimiento.nh + 1,inicio: Bloque.inicio>
}
```

```
Bloque →
    {
      Declaraciones.tsh   = Bloque.tsh
      Declaraciones.nh    = Bloque.nh
      Declaraciones.dirh  = Bloque.dirh
      Declaraciones.etqh  = Bloque.etqh

    }
    Declaraciones &
    {
      Instrucciones.etqh  = Declaraciones.etq + longPrologo
      Instrucciones.tsh   = Declaraciones.ts
    }
    Instrucciones
    {
      Bloque.ts    = Declaraciones.ts
      Bloque.error = Declaraciones.error v Instrucciones.error
      Bloque.pend  = Declaraciones.pend
      Bloque.n       = Declaraciones.n
      Bloque.inicio  = Declaraciones.etq
      Bloque.etq     = Instrucciones.etq + longEpilogo + 1
      Bloque.cod     =
               Declaracioness.cod                    ||
               prologo(Bloque.nh,Declaraciones.dir) ||
               Instrucciones.cod                     ||
               epilogo(Bloque.nh)          ||
               ir-ind
    }


Bloque →
    {
      Instrucciones.tsh =  Bloque.tsh
      Instrucciones.etqh       = Bloque.etqh + longPrologo
    }
    Instrucciones
    {
      Bloque.error = Instrucciones.error
      Bloque.n       = Bloque.nh
      Bloque.cod     =
          prologo(Bloque.nh,Bloque.dirh)  ||
          Instrucciones.cod               ||
          epilogo(Bloque.nh)              ||
          ir-ind

      Bloque.inicio  = Bloque.etqh
      Bloque.etq     = Instrucciones.etq + longEpilogo + 1
    }



FParametros →
```

```
    {
      LFParametros.tsh = FParametros.tsh
      LFPatametros.nh = FParametros.nh

    }
    (LFParametros)
    {
      FParametros.ts = LFParametros.ts
      FParametros.parametros = LFParametros.parametros
      FParametros.error = LFParametros.error
      FParams.dir = LFParams.dir
    }

FParametros → λ
    {
      FParametros.dir = 0
      FParametros.error = false
      FParametros.ts = FParametros.tsh
      FParametros.parametros = {}
    }

LFParametros →
    {
      FParametro.nh = LFParametros.nh
    }
    FParametro
    {
      LFParametrosRec.nh = LFParametros.nh
      LFParametrosRec.tsh = inserta(LFParametros.tsh, FParametro.id, FParametro.props ++
    <dir:0>)
      LFParametrosRec.parametrosh = FParametro.parametro
      LFParametrosRec.dirh = FParametro.tam
      LFParametrosRec.errorh = Fparametro.error v
                  existeID(LFParametros.tsh,FParametro.id)  ∧
                        LFParametros.tsh[FParametro.id].nivel = LFParametros.nh)
    }
    LFParametrosRec
    {
      LFParametros.ts = LFParametrosRec.ts
      LFParametros.parametros = LEParametrosRec.parametros
      LFParametros.error = LFParametrosRec.error
      LFParametros.dir = LFParametrosRec.dir
    }


LFParametrosRec → ','
    {

      FParametro.nh = LFParametrosRec0.nh
      FParametro.dirh = LFParametrosRec0.dirh
    }
    FParametro
    {
```

```
      LFParametrosRec1.nh = LFParametrosRec0.nh
      LFParametrosRec1.dirh = LFParametrosRec0.dirh + FParametro.tam
      LFParametrosRec1.tsh = inserta(LFParametrosRec0.tsh, Fparametro.id, FParametro.props)
      LFParametrosRec1.parametrosh = LFParametrosRec0.parametros ++ FParametro.parametro
      LFParametrosRec1.errorh = LFParametrosRec0.errorh v FParametro.error v

      (existeID(LFParametrosRec0.tsh,FParametro.id)  ∧
                                        LFParametrosRec0.tsh[FParametro.id].nivel =
    LFParametrosRec0.nh)
    }
    LFParametrosRec
    {
      LFParametrosRec0.ts = LFParametrosRec1.ts
      LFParametrosRec0.parametros = LFParametrosRec1.parametros
       LFParametrosRec0.error = LFParametrosRec1.error
      LFParametrosRec0.dir = LFParametrosRec1.dir
    }

LFParametrosRec → λ
    {
      LFParametrosRec.ts = LFParametrosRec.tsh
      LFParametrosRec.parametros= LFParametrosRec.parametrosh
      LFParametrosRec.error = LFParametros.errorh
      LFParametrosRec.dir = LFParametrosRec.dirh
    }


FParametro → var Tipo id
    {
      FParametro.tam      = 1
      Fparametro.parametro    = <modo: variable, tipo: Tipo.tipo, dir: Fparametro.dirh>
      FParametro.id = id.lex
      FParametro.props = <clase: pvar, tipo: Tipo.tipo, nivel: FParametro.nh>
      FParametro.error = Tipo.error
    }

FParametro → Tipo id
    {
      FParametro.tam = Tipo.tipo.tam
      Fparametro.parametro = <modo: valor, tipo: Tipo.tipo, dir: Fparametro.dirh>
      FParametro.id = id.lex
      FParametro.props = <clase: var, tipo: Tipo.tipo, nivel: FParametro.nh>
      Fparametro.error = Tipo.error
    }

Tipo → id
    {
        Tipo.tipo =
            <
                t:ref,
                id:id.lex,
                tam:Tipo.tsh[id.lex].tipo.tam
            >
```

```
      Tipo.error =  si existeID(Tipo.tsh,id.lex)
                    Tipo.tsh[id.lex].clase != tipo
                     sino
                    false
      Tipo.pend =  si (¬existeID(Tipo.tsh,id.lex))
                    {id.lex}
                     sino
                       ∅
    }

Tipo → Boolean
    {
      Tipo.tipo = <t:boolean,tam:1>
      Tipo.error = false
      Tipo.pend =  ∅

    }


Tipo → Character
    {
      Tipo.tipo = <t:character,tam:1>
      Tipo.error = false
      Tipo.pend = ∅
    }

Tipo → Float
    {
      Tipo.tipo = <t:float,tam:1>
      Tipo.error = false
      Tipo.pend = ∅
    }

Tipo → Natural
    {
      Tipo.tipo = <t:natural,tam:1>
      Tipo.error = false
      Tipo.pend = ∅
    }

Tipo → Integer
    {
      Tipo.tipo = <t:integer,tam:1>
      Tipo.error = false
      Tipo.pend = ∅
    }

Tipo → array [num] of Tipo
    {
        Tipo0.tipo =
            <
                t:array,
                nelems:valorDe(num.lex),
```

```
                tbase:Tipo1.tipo,
                tam:valorDe(num.lex)*Tipo1.tipo.tam
            >
      Tipo1.tsh = Tipo0.tsh
      Tipo0.error = Expresion.tipo.t != natural v Tipo1.error v ¬existeRef (Tipo0.tsh ,
    Tipo1.tipo)
      Tipo0.pend = Tipo1.pend
    }

Tipo → ^Tipo
    {
        Tipo0.tipo =
            <
                t:puntero,
                tbase:Tipo1.tipo,
                tam:1
            >
      Tipo1.tsh = Tipo0.tsh
      Tipo0.error = Tipo1.error
      Tipo0.pend = Tipo1.pend
    }


Tipo → reg
    {
      Campos.tsh = Tipo.tsh
    }
    Campos freg
    {
        Tipo.tipo =
            <
                t:array,
                campos:Campos.campos,
                tam:Campos.tam
            >
             Tipo.error = Campos.error
      Tipo.pend = Campos.pend
    }

Campos →
    {
      Campo.desh = 0
      Campo.tsh = Campos.tsh
    }
    Campo
    {
      CamposRec.camposh = [Campo.campo]
      CamposRec.errorh = Campo.error
      CamposRec.pendh = Campo.pend
      CamposRec.desh = Campo.tam
      CamposRec.tsh = Campos.tsh
```

```
    }
    CamposRec
    {
      Campos.campos = CamposRec.campos
      Campos.error = CamposRec.error
      Campos.pend = CamposRec.pend
      Campos.tam = CamposRec.tam
    }



CamposRec → ';'
    {
      Campo.desh=CamposRec0.desh
      Campo.tsh = CamposRec0.tsh
    }
    Campo
    {
      CamposRec1.tsh = CamposRec0.tsh
      CamposRec1.camposh = CamposRec0.camposh ++ Campo.campo
      CamposRec1.errorh = CamposRec0.errorh v existeCampo(CamposRec0.camposh, Campo.id)
      CamposRec1.pendh = CamposRec0.pendh U Campo.pend
      CamposRec1.desh = Campo.tam + CamposRec0.desh
    }
    CamposRec
    {
      CamposRec0.campos = CamposRec1.campos
      CamposRec0.pend = CamposRec1.pend
      CamposRec0.error = CamposRec1.error
      CamposRec0.tam = CamposRec1.tam
    }


CamposRec → λ
    {
      CamposRec.campos = CamposRec.camposh
      CamposRec.error =  CamposRec.error
      CamposRec.pend = CamposRec.pendh
      CamposRec.tam = CamposRec.desh
    }


Campo →
    {
      Tipo.tsh = Campo.tsh
    }
    Tipo id
    {
      Campo.campo =
            <
                id:iden.lex,
                tipo:Tipo.tipo,
                desp:Campo.desh
```

```
            >
      Campo.tam   = Tipo.tam
      Campo.error = Tipo.error v ¬existeRef(Campo.tsh , Tipo.tipo)
      Campo.pend = Tipo.pend
    }


Instrucciones →
    {
      Instruccion.tsh = Instrucciones.tsh
      Instruccion.etqh = Instrucciones.etqh
    }
    Instrucción
    {
      InstrucciónesRec.tsh = Instrucciones.tsh
      InstruccionesRec.errorh = Instrucción.error
      InstruccionesRec.etqh = Instruccion.etq
      InstruccionesRec.codh = Instruccion.cod
    }
    InstruccionesRec
    {
      Instrucciones.error = InstruccionesRec.error
      Instrucciones.etq = InstruccionesRec.etq
      Instrucciones.cod = InstruccionRec.cod
    }


InstruccionesRec → ;
    {
      Instruccion.tsh = InstruccionesRec0.tsh
      Instruccion.etqh = InstruccionesRec0.etqh
    }
    Instrucción
    {
      InstruccionesRec1.tsh = InstruccionesRec0.tsh
      InstruccionesRec1.errorh = Instrucción.error v InstruccionesRec0.errorh
      InstruccionesRec1.etqh = Instruccion.etq
      InstruccionesRec1.codh = InstruccionesRec0.cod | | Instruccion.cod
    }
    InstruccionesRec
    {
      InstruccionesRec0.error = InstruccionesRec1.error
      InstruccionesRec0.cod = InstruccionesRec1.cod
      InstruccionesRec0.etq = InstruccionesRec1.etq
    }


InstruccionesRec → λ
    {
      InstruccionesRec.error = InstruccionesRec.errorh
      InstruccionesRec.etq = InstruccionesRec.etqh
      InstruccionesRec.cod = InstruccionesRec.codh
    }
```

```
Instrucción →
    {
      InsProcedimiento.tsh = Instrucción.tsh
      InsProcedimiento.etqh = Instrucción.etqh
    }
    InsProcedimiento
    {
      Instrucción.error = InsProcedimiento.error
      Instrucción.cod = InstruccionProcedimiento.cod
      Instrucción.etq = InstruccionProcedimiento.etq
    }

Instrucción →
    {
      InsLectura.tsh = Instrucción.tsh
      InsLectura.etqh = Instrucción.eth
    }
    InsLectura
    {
      Instrucción.error = InsLectura.error
      Instrucción.cod = InsLectura.cod
      Instrucción.etq = InsLectura.etq
    }

Instrucción →
    {
      InsEscritura.tsh = Instrucción.tsh
      InsEscritura.etqh = Instrucción.etqh
    }
    InsEscritura
    {
      Instrucción.error = InsEscritura.error
      Instruccion.cod = InsEscritura.cod
      Instrucción.etq = InsEscritura.etq
    {

Instrucción →
    {
      InsAsignacion.tsh = Instrucción.tsh
      InsAsignacion.etqh = Instrucción.etqh
    }
    InsAsignación
    {
      Instrucción.error = InsAsignacion.error
       Instrucción.cod = InsAsignación.cod
       Instrucción.etq = InsAsignacion.etq
    }

Instrucción →
    {
      InsCompuesta.tsh = Instrucción.tsh
```

```
      InsCompuesta.etqh = Instrucción.etqh
    }
    InsCompuesta
    {
      Instrucción.error = InsCompuesta.error
      Instrucción.cod = InsCompuesta.cod
      Instrucción.etq = InsCompuesta.etq
    }

Instrucción →
    {
      InsIf.tsh = Instrucción.tsh
      InsIf.etqh = Instrucción.etqh
    }
    InsIf
    {
      Instrucción.error = InsIf.error
      Instrucción.cod = InsIf.cod
      Instrucción.etq = InsIf.etq
    }

Instrucción →
    {
      InsWhile.tsh = Instrucción.tsh
      InsWhile.etqh = Instrucción.etqh
    }
    InsWhile
    {
      Instrucción.error = InsWhile.error
      Instrucción.cod = InsWhile.cod
      Instrucción.etq = InsWhile.etq
    }

Instrucción →
    {
      InsFor.tsh = Instrucción.tsh
      InsFor.etqh = Instrucción.etqh
    }
    InsFor
    {
      Instrucción.error = InsFor.error
      Instrucción.cod = InsFor.cod
      Instrucción.etq = InsFor.etq
    }
Instrucción →
    {
      InsNew.etqh = instrucción.etqh
    }
    InsNew
    {
      Instrucción.error =InsNew.error
      Instrucción.cod = InsNew.cod
      Instrucción.etq = InsNew.etq
```

```
    }

Instrucción →
    {
      InsDis.etqh = Instrucción.etqh
    }
    InsDis
    {
      Instrucción.error = InsDis.error
      Instrucción.cod = InsDis.cod
      Instrucción.etq = InsDis.etq
    }


InsProcedimiento → id
    {
      AParametros.tsh = InsProcedimiento.tsh
      AParametros.fparametrosh = InsProcedimiento.tsh[id.lex].tipo.parametros
      AParametros.etqh           = InsProcedimiento.etqh + longApilaRet
    }
    AParametros
    {
      InsProcedimiento.error =
            ¬existeID(InsProcedimiento.tsh,id.lex) v
            InsProcedimiento.tsh[id.lex].clase != proc v
            Aparametros.error

      InsProcedimiento.cod    =
            si (InsProcedimiento.tsh[id.lex].props.clase==forward)
                apila-ret(InsProcedimiento.etq) ||
                Aparametros.cod                          ||
                nop(id.lex)
            sino
                apila-ret(InsProcedimiento.etq)         ||
                AParametros.cod                          ||
                ir-a(InsProcedimiento.tsh[iden.lex].inicio)
      InsProcedimiento.etq    = AParametros.etq + 1
    }



AParametros →
    {
      LAParametros.tsh = Aparametros.tsh
      LAParametros.fparametrosh = Aparametros.fparametrosh
    }
    (LAParametros)
    {
      AParametros.error =
            LAParametros.error v |Aparametros.fparametrosh| != LAParametros.nparametros
      AParametros.cod     = inicio-paso || LAParametros.cod || fin-paso
      AParametros.etq     = LAParametros.etq + longFinPaso
    }
```

```
AParametros → λ
    {
      AParametros.error = |AParametros.fparametrosh | > 0
      AParametros.cod     = λ
      AParametros.etq     = AParametros.etqh
    }



LAParametros →
    {
      Expresion.tsh = LAParametros.tsh
      Expresion.etqh = LAParametros.etqh + 1
      Expresion.parh =  LAParametros.fparametrosh[LAParametros.nparametros_h].modo == var
    }
    Expresion
    {
      LAParametrosRec.nparametrosh = 1
      LAParametrosRec.tsh = LAParametros.tsh
      LAParametrosRec.fparametrosh = LAParametros.fparametrosh
      LAParametrosRec.errorh =
            Expresion.error v
            | LAParametros.fparametrosh | < 1 v
            LAParametros.fparametrosh[0].modo = var ∧ Expresion.modo = val
            ¬compatibles(LAParametros.fparametrosh[0].tipo, Expresion.tipo,
                                                                  LAParametros.tsh)
            LAParametrosRec.codh =
                  copia ||
                  Expresion.cod ||
                  pasoParametro(Exp.modo,LAParametros.fparametrosh[0])
            LAParametrosRec.etqh = Expresion.etq + longPaseoParametro
    }
    LAParametrosRec
    {
      LAParametros.error = LAParametrosRec.error
            LAParametros.cod = LAParametrosRec.cod
      LAParametros.etq = LAParametrosRec.etq
    }
```

```
LAParametrosRec → ','
    {
      Expresion.tsh = LAParametrosRec0.tsh
      Expresion.etqh = LAParametrosRec0.etqh + 1 + longDireccionParFormal
      Expresion.parh =
            LAParametrosRec0.fParametrosh[LAParametros0Rec.nParametros_h].modo == var
    }
    Expresion
    {
      LAParametrosRec1.nparametrosh = LAParametrosRec0.nparametrosh + 1
      LAParametrosRec1.tsh = LAParametrosRec0.tsh
      LAParametrosRec1.fparametrosh = LAParametrosRec0.fparametrosh
      LAParametrosRec1.errorh =
            LAParametrosRec0.errorh v  Expresion.error v
            |LAParametros.fparametrosh|< LAParametrosRec0.nparametrosh + 1 v
            LAParametrosRec0.parametrosh[LAParametrosRec0.nparametrosh].modo = var  ∧
                  Expresion.modo = val v
            ¬compatibles(LAParametrosRec0.fparametrosh
               [LAParametrosRec0.nparametrosh].tipo, Expresion.tipo, LAParametrosRec0.tsh)

      LAParametrosRec1.etqh = Expresion.etq + longPasoParametro
      LAParametrosRec1.codh =
            LAParametrosRec0.codh ||
            copia ||

      direccionParFormal(LAParametrosRec0.fparametrosh[LAParametrosRec0.nparametros_h]) ||
            Expresion.cod ||
            pasoParametro(Expresion.modo,

      LAParametrosRec0.fparametrosh[LAParametrosRec0.nparametros_h])
    }
    LAParametrosRec
    {
      LAParametrosRec0.error = LAParametrosRec1.error
      LAParametrosRec0.etq = LAParametrosRec1.etq
      LAParametrosRec0.cod = LAParametrosRec1.cod
    }


LAParametrosRec → λ
    {
      LAParametros.error = LAParametrosRec.errorh
      LAParametrosRec.cod = LAParametrosRec.codh
      LAParametrosRec.etq = LAParametrosRec.etqh
    }

InsLectura →  in(id)
    {
      InsLectura.error    = NOT existeID(InsLectura.tsh,id.lex)
```

```
      InsLectura.cod      =
            accesoVar(InsLectura.tsh[id.lex].props) ||
            in     ||
            desapilaInd
      InsLectura.etq      = InsLectura.etqh + longAccesoVar + 2
    }

InsEscritura → out
    {
      Expresion.etqh = InsEscritura.etqh
      Expresion.parh = false
    }
    (Expresion)
    {
      InsEscritura.error = (Expresion.tipo = <t:error>)
      InsEscritura.etq   = Expresion.etq + 1
      InsEscritura.cod   = Expresion.cod || out
    }


InsAsignación →
    {
      Mem.etqh = InsAsignación.etqh
      Mem.tsh = InsAsignacion.tsh
    }
    Mem :=
    {
      Expresion.parh = false
      Expresion.etqh = Mem.etq
      Expresion.tsh = InsAsignacion.tsh
    }
    Expresión
    {
      InsAsignación.etq = Expresion.etq + 1
      InsAsignación.cod =
            si esCompatibleConTipoBasico(Mem.tipo,Expresion.tsh)
                  Mem.cod || Expresion.cod || desapila-ind
            si no
                  Mem.cod || Expresion.cod || mueve(Mem.tipo.tam)
      InsAsignacion.error = ¬ esCompatible(Mem.tipo, Expresion.tipo,
                                    InsAsignacion.tsh)
    }

InsCompuesta →
    {
      Instrucciones.tsh = InsCompuesta.tsh
      Instrucciones.etqh = InsCompuesta.etqh
    }
    '{' Instrucciones '}'
    {
      InsCompuesta.error  = Instrucciones.error
      InsCompuesta.cod = Instrucciones.cod
      InsCompuesta.etq = Instrucciones.etq
```

```
    }


InsIf → if
    {
      Expresion.tsh = InsIf.tsh
      Expresion.etqh = InsIf.etqh
      Expresion.parh = false
    }
    Expresion then
    {
      Instrucción.tsh = InsIf.tsh
      Instruccion.etqh = Expresion.etq + 1
    }
    Instrucción
    {
      PElse.tsh = InsIf.tsh
      PElse.etqh = Instruccion.etq + 1
    }
    PElse
    {
      InsIf.error = Expresion.tipo != <t:bool> v Instrucción.error v PElse.error
      InsIf.etq = PElse.etq
      InsIf.cod =
            Expresion.cod ||
            ir-f(Instruccion.etq + 1) ||
            Instruccion.cod ||
            ir-a(PElse.etq) ||
            PElse.cod
    }


PElse → else
    {
      Instrucción.tsh = PElse.tsh
      Instruccion.etqh = PElse.etqh
    }
    Instrucción
    {
      PElse.error = Instrucción.error
            PElse.cod = Instruccion.cod
            PElse.etq = Instruccion.etq
    }


PElse → λ
    {
      PElse.error = false
      PElse.cod  = λ
      PElse.etq  = PElse.etqh
    }
```

```
InsWhile → while
    {
      Expresion.etqh = InsWhile.etqh
      Expresion.tsh = InsWhile.tsh
    }
    Expresion do
    {
      Instrucción.tsh = InsWhile.tsh
      Instruccion.etqh = Expresion.etq + 1
    }
    Instrucción
    {
      InsWhile.error = Expresion.tipo != <t:bool> v Instrucción.error
      InsWhile.etq  = Instruccion.etq + 1
      InsWhile.cod  =
            Expresion.cod               ||
            ir-f(Instruccion.etq + 1)   ||
            Instruccion.cod             ||
            ir-a(InsWhile.etqh)
    }

InsFor → 'for' id=
    {
      Expresion0.tsh = InsFor.tsh
      Expresion0.etqh   = InsFor.etqh
    }
    Expresion 'to'
    {
      Expresion1.tsh = InsFor.tsh
      Expresion1.etqh   =
            Expresion0.etq + 1 + longAccesoVar(InsFor.tsh[id.lex].props)
    }
    Expresion 'do'
    {
      Instrucción.tsh = InsFor.tsh
      Instruccion.etqh  =
            Expresion1.etq + 4 + longAccesoVar(InsFor.tsh[id.lex].props)
    }
    Instruccion
    {
      InsFor.error =(Expresion0.tipo!=<t:natural> ∧ Expresion0.tipo !=<t:integer>) v
            (Expresion1.tipo != <t:natural> ∧ Expresion1.tipo != <t:integer>) v
            (id.tipo != <t:natural> ∧ id.tipo != <t:integer>)
      InsFor.etq = Instruccion.etq + 6
      InsFor.cod  =
                  accesoVar(InsFor.tsh[id.lex].props) ||
                      Expresion0.cod ||
                      desapila-ind ||
                      Expresion1.cod ||
                      Copia ||
                      accesoVar(InsFor.tsh[id.lex].props) ||
                  apila-ind                                ||
                  igual                               ||
```

```
                    ir-v( InsFor.etq - 1)                ||
                    Instruccion.cod                      ||
                    apila-dir InsFor.tsh[id.lex].dir     ||
                    apilar 1                             ||
                    sumar                                ||
                    desapila-dir InsFor.tsh[id.lex].dir ||
                    ir-a (Expresion1.etq)
                    desapila
    }


InsNew → new
    {
      Mem.tsh = InsNew.tsh
      Mem.etqh    = InsNew.etqh
    }
    Mem
    {
      InsNew.error = Mem.tipo.t != <t:puntero>
      InsNew.etq  = Mem.etq + 2
      InsNew.cod  =
            Mem.cod ||
            new(
                  si Mem.tipo.tbase = ref
                        InsNew.tsh[Mem.tipo.tbase.id].tam
                  si no
                        1
                   )
             || desapila-ind
    }


InsDis → dispose
    {
      Mem.tsh = InsDis.tsh
      Mem.etqh    = InsDis.etqh
    }
    Mem
    {
      InsDis.error = Mem.tipo.t != <t:puntero>
      InsDis.etq  = Mem.etq + 1
      InsDis.cod  =
            Mem.cod ||
            del(
                  si Mem.tipo.tbase = ref
                        InsDis.tsh[Mem.tipo.tbase.id].tam
                  si no
                        1
                   )
    }
```

```
Mem → id
     {
              MemRec.tsh = Mem.tsh
              MemRec.tipoh = si existe(Mem.tsh, id.lex)
                                        si Mem.tsh[id.lex].clase = var
                                               ref!(Mem.tsh[id.lex].tipo,Mem.tsh)
                                        sino <t:error>
                              sino <t:error>
       MemRec.etqh = Mem.etqh + longAcessoVar(Mem.tsh[id.lex])
       MemRec.codh = accesoVar(Mem.tsh[id.lex])
     }
     MemRec
     {
       Mem.tipo = MemRec.tipo
       Mem.cod =  MemRec.cod
       Mem.etq = MemRec.etq
     }


MemRec → '^'
     {
       MemRec1.tsh = MemRec0.tsh
       MemRec1.tipoh = si (MemRec0.tipoh.t = puntero
                                  ref!(MemRec0.tipoh.tbase,MemRec0.tsh)
                             sino <t:error>
       MemRec1.etqh = MemRec0.etqh + 1
       MemRec1.codh = MemRec0.codh || apila-ind
     }
     MemRec
     {
       MemRec0.tipo = MemRec1.tipo
       MemRec0.etq = MemRec1.etq
       MemRec0.cod = MemRec1.cod
     }

MemRec → '['
     {
       Expresion.tsh = MemRec0.tsh
       Expresion.etq = MemRec0.etqh
     }
     Expresion']'
     {
       MemRec1.tsh = MemRec0.tsh
       MemRec1.etqh = Expresion.etq + 3
       MemRec1.codh = MemRec0.codh || Exp.cod ||
                     apila  MemRec0.tipoh.tbase.tam || multiplica || suma
       MemRec1.tipoh = si MemRec0.tipo.t = array AND Expresion.tipo.t = num
                                  ref!(MemRec0.tipoh.tbase,MemRec0.tsh)
                             sino <t:error>
     }
     MemRec
     {
       MemRec0.tipo = MemRec1.tipo
```

```
      MemRec0.etq = MemRec1.etq
      Memrec0.cod = MemRec1.cod
    }


MemRec → '.'id
    {
      MemRec1.tsh = MemRec0.tsh
      MemRec1.etqh = MemRec0.etqh + 2
      MemRec1.tipoh =
            si MemRec0.tipoh.t = rec
                  si campo?(MemRec0.tipoh.campos, id.lex)
                        ref!(MemRec0.tipoh.campos[id.lex].tipo, MemRec0.tsh)
                  sino <t:error>
            sino <t:error>
      MemRec1.codh =
            MemRec0.codh || apila(MemRec0.tipo.campos[id.lex].desp || suma
    }
    MemRec
    {
      MemRec0.tipo = MemRec1.tipo
      MemRec0.etq = MemRec1.etq
      Memrec0.cod = MemRec1.cod
    }


MemRec → λ
    {
      MemRec.tipo = MemRec.tipoh
      MemRec.etq = MemRec.etqh
      MemRec.cod = MemRec.codh
    }



Expresion →
    {
      ExpresionNiv1.tsh = Expresion.tsh
      ExpresionNiv1.etqh = Expresion.etqh
      ExpresionNiv1.parh = Expresion.parh

    }
    ExpresiónNiv1
    {
      ExpresionFact.tipoh = ExpresionNiv1.tipo
      ExpresionFact.tsh = Expresion.tsh
      ExpresionFact.modoh = ExpresionNiv1.modo
      ExpresionFact.etqh = ExpresionNiv1.etq
      ExpresionFact.codh = ExpresionNiv1.cod
    }
    ExpresiónFact
    {
      Expresion.tipo = ExpresionFact.tipo
```

```
      Expresion.modo=ExpresionFact.modo
      Expresion.cod = ExpresionFact.cod
      Expresion.etq = ExpresionFact.etq
    }


ExpresionFact → OpNivO
    {
      ExpresionNiv1.tsh = ExpresionFact.tsh
      ExpresionNiv1.parh= false
      ExpresionNiv1.etqh = ExpresionFact.etqh
    }
    ExpresionNiv1
    {
      ExpresionFact.tipo = si    (ExpresionFact.tipoh = <t:error> v
                                 ExpresionNiv1.tipo = <t:error>) v
                                 ( ExpresionFact.tipoh = <t:character> v
                                       ExpresionNiv1.tipo =/= <t:character>) v
                                 ( ExpresionFact.tipoh =/= <t:character> ∧
                                       ExpresionNiv1.tipo = <t:character>)
                                 (ExpresionFact.tipoh = <t:boolean> ∧
                                       ExpresionNiv1.tipo =/= <t:boolean>) v
                                 ( ExpresionFact.tipoh =/= <t:boolean> ∧
                                 ExpresionNiv1.tipo = <t:boolean>))
                                  error
                    sino  boolean
      ExpresionFact.modo = val
      ExpresionFact.cod =
                  ExpresionFact.codh ||
                  ExpresionNiv1.cod ||
                  case (OpNivO)
                         menor: menor
                         mayor: mayor
                         menor-ig: menorIg
                         mayor-ig: mayorIg
                         igual : igual
                         no-igual: no-igual

      ExpresionFact.etq = ExpresionNiv1.etq + 1
    }

ExpresionFact → λ
    {
      ExpresionFact.modo = ExpresionFact.modoh
      ExpresionFact.tipo = ExpresionFact.tipoh
            ExpresionFact.cod = ExpresionFact.codh
      ExpresionFact.etq = ExpresionFact.etqh
    }

ExpresiónNiv1 →
    {
      ExpresionNiv2.tsh = ExpresionNiv1.tsh
```

```
    ExpresionNiv2.etqh = ExpresionNiv1.etqh
    ExpresionNiv2.parh = ExpresionNiv1.parh
  }
  ExpresiónNiv2
  {
    ExpresionNiv1Rec.tsh = ExpresionNiv1.tsh
    ExpresionNiv1Rec.tipoh = ExpresiónNiv2.tipo.
    ExpresionNiv1Rec.modoh = ExpresionNiv2.modo
    ExpresionNiv1Rec.etqh = ExpresionNiv2.etq
    ExpresionNiv1Rec.codh = ExpresionNiv2.cod
  }
  ExpresionNiv1Rec
  {
    ExpresionNiv1.tipo = ExpresiónNiv1Rec.tipo
    ExpresionNiv1.cod = ExpresionNiv1Rec.cod
    ExpresionNiv1.etq = ExpresionNiv1Rec.etq
    ExpresionNiv1.modo = ExpresionNiv1Rec.modo
  }


ExpresiónNiv1Rec → OpNiv1
  {
    ExpresionNiv2.tsh = ExpresionNiv1Rec0.tsh
    ExpresionNiv2.parh = false
    if (opNiv1.op == or):
          ExpresionNiv2.etqh = ExpresionNiv1Rec0.etqh + 3
    else
          ExpresionNiv2.etqh = ExpresionNiv1Rec0.etqh
  }
  ExpresiónNiv2
  {
    ExpresionNiv1Rec1.tsh = ExpresionNiv1Rec0.tsh
    ExpresionNiv1Rec1.tipoh =
          si     (ExpresionNiv1Rec0.tipoh = <t:error> v
                 ExpresionNiv2.tipo = <t:error> v
                 ExpresionNiv1Rec0.tipoh=<t:character> v
                 ExpresionNiv2.tipo=<t:character> v
                 (ExpresionNiv1Rec0.tipoh =<t:boolean> ∧
                        ExpresionNiv2.tipo =/= <t:boolean>) v
                 (ExpresionNiv1Rec0.tipoh=/=<t:boolean> ∧
                       ExpresionNiv2.tipo = <t:boolean>))
                 <t:error>
          sino case (OpNiv1.op)
                 suma,resta:
                       si     (ExpresionNiv1Rec0.tipoh=<t:float> v
                              ExpresionNiv2.tipo = <t:float>)
                              <t:float>
                       sino si       (ExpresionNiv1Rec0.tipoh = <t:integer> V
                                     ExpresionNiv2.tipo=<t:integer>)
                              <t:integer>
                       sino si       (ExpresionNiv1Rec0.tipoh = <t:natural> ∧
                                     ExpresionNiv2.tipo = <t:natural>)
                              <t:natural>
```

```
                        sino
                              <t:error>
                  o:
                        si    (ExpresionNiv1Rec0.tipoh = <t:boolean> ∧
                              ExpresionNiv2.tipo = <t:boolean>)
                              <t:boolean>
                        sino  <t:error>
                  or:
                        si    (ExpresionNiv1Rec0.tipoh = <t:boolean> y
                              ExpresionNiv2.tipo = <t:boolean>)
                              <t:boolean>
                        sino
                              <t:error>

      ExpresionNiv1Rec1.modoh = val
      ExpresionNiv1Rec1.tipoh = ExpresionNiv2.tipo
      if (opNiv1.op == or):
            ExpresionNiv1Rec1.codh =
                  ExpresionNiv1Rec0.codh ||
                  dup ||
                  ir-v (ExpresionNiv2.etq) ||
                  desapila ||
                  ExpresionNiv2.cod
            ExpresionNiv1Rec1.etqh = ExpresionNiv2.etq
      else:
            ExpresionNiv1Rec1.etqh = ExpresionNiv2.etq + 1
            ExpresionNiv1Rec1.codh =
                  ExpresionNiv1Rec0.codh ||
                  ExpresionNiv2.cod ||
                  case (OpNiv1.op)
                        suma: sumar
                        resta: restar
   }
   ExpresiónNiv1Rec
   {
      ExpresionNivlRec0.tipo = ExpresionNiv1Rec1.tipo
      ExpresionNiv1Rec0.modo = val
      ExpresionNiv1Rec0.etq = ExpresionNiv1Rec1.etq
      ExpresionNiv1Rec0.cod = ExpresionNiv1Rec1.cod
   }


ExpresionNiv1Rec → λ
{
      ExpresionNiv1Rec.tipo = ExpresionNiv1Rec.tipoh
      ExpresionNiv1Rec.modo = ExpresionNiv1Rec.modoh
      ExpresionNiv1Rec.etq = ExpresionNiv1Rec.etqh
      ExpresionNiv1Rec.cod = ExpresionNiv1Rec.codh
}

ExpresiónNiv2 →
   {
      ExpresionNiv3.tsh = ExpresionNiv2.tsh
```

```
    ExpresionNiv3.etqh = ExpresionNiv2.etqh
    ExpresionNiv3.parh = ExpresionNiv2.parh
  }
  ExpresionNiv3
  {
    ExpresionNiv2Rec.tipoh = ExpresiónNiv3.tipo
    ExpresionNiv2Rec.tsh = ExpresionNiv2.tsh
    ExpresionNiv2Rec.modoh = ExpresionNiv3.modo
    ExpresionNiv2Rec.etqh = ExpresionNiv3.etq
    ExpresionNiv2Rec.codh = ExpresionNiv3.cod
  }
  ExpresiónNiv2Rec
  {
    ExpresionNiv2.tipo = ExpresiónNiv2Rec.tipo
    ExpresionNiv2.cod = ExpresionNiv2Rec.cod
    ExpresionNiv2.modo = ExpresionNiv2Rec.modo
    ExpresionNiv2.etq = ExpresionNiv2Rec.etq
  }


ExpresiónNiv2Rec → OpNiv2
  {
    ExpresionNiv3.tsh = ExpresionNiv2Rec0.tsh
    if (OpNiv2.op == and)
          ExpresionNiv3.etqh = ExpresionNiv2Rec0.etqh + 1
    else
          ExpresionNiv3.etqh = ExpresionNiv2Rec0.etqh
  }
  ExpresiónNiv3
  {
    ExpresionNiv2Rec1.tsh = ExpresionNiv2Rec0.tsh
    ExpresionNiv2Rec1.modoh = val
    ExpresionNiv2Rec1.tipoh =
          si     (ExpresionNiv2Rec0.tipoh = <t:error> v
                 ExpresionNiv3.tipo = <t:error> v
                 ExpresionNiv2Rec0.tipoh = <t:character> v
                 ExpresionNiv3.tipo = <t:character> v
                 (ExpresionNiv2Rec0.tipoh = <t:boolean> ∧
                       ExpresionNiv3.tipo =/= <t:boolean> v
                 (ExpresionNiv2Rec0.tipoh =/= <t:boolean> ∧
                       ExpresionNiv3.tipo = <t:boolean>))
                    <t:error>
          sino case (OpNiv2.op)
                 multiplica,divide:
                          si   (ExpresionNiv2Rec0.tipoh=<t:float> v
                               ExpresionNiv3.tipo = <t:float>)
                               <t:float>
                          sino si     (ExpresionNiv2Rec0.tipoh = <t:integer> v
                                      ExpresionNiv3.tipo = <t:integer>)
                               <t:integer>
                          sino si     (ExpresionNiv2Rec0.tipoh = <t:natural> ∧
                                      ExpresionNiv3.tipo= <t:natural>)
```

```
                        <t:natural>
                    sino
                        <t:error>
            modulo:
                    si   (ExpresionNiv3.tipo = <t:natural> ∧
                        (ExpresionNiv2Rec0.tipoh= <t:natural> v
                        ExpresionNiv2Rec0.tipoh= <t:integer>))
                        ExpresionNiv2Rec0.tipoh
                    sino
                        <t:error>
            y:
                    si   (ExpresionNiv2Rec0.tipoh = <t:boolean> ∧
                        ExpresionNiv3.tipo = <t:boolean>)
                        <t:boolean>
                    sino
                        <t:error>
            and:
                        si     (ExpresionNiv2Rec0.tipoh = <t:boolean> y
                               ExpresionNiv3.tipo = <t:boolean>
                               <t:boolean>
                        sino
                               <t:error>
    if (OpNiv2.op == and)
          ExpresionNiv2Rec1.cod =
                ExpresionNiv2Rec0.codh ||
                ir-f(ExpresionNiv3.etq + 1) ||
                ExpresionNiv3.cod ||
                ir-a(ExpresionNiv3.etq + 2) ||
                apila(0)
          ExpresionNiv2Rec1.etqh = ExpresionNiv3.etq + 2

    else:
          ExpresionNiv2Rec1.cod =
                ExpresionNiv2Rec0.codh ||
                ExpresionNiv3.cod ||
                case (OpNiv2.op)
                      Multiplica: mul
                      Divide: div
                      Modulo: Mod
          ExpresionNiv3Rec1.etqh = ExpresionNiv3.etq + 1

  }
  ExpresiónNiv2Rec
  {
    ExpresionNiv2Rec0.tipo = ExpresionNiv2Rec1.tipo
    ExpresionNiv2Rec0.cod = ExpresionNiv2Rec1.cod
    ExpresionNiv2Rec0.modo = ExpresionNiv2Rec1.modo
    ExpresionNiv2Rec0.etq = ExpresionNiv2Rec1.etq
  }

ExpresiónNiv2Rec → λ
  {
    ExpresionNiv2Rec.tipo = ExpresionNiv2Rec.tipoh
```

```
      ExpresionNiv2Rec.modo = ExpresionNiv2Rec.modh
      ExpresionNiv2Rec.cod = ExpresionNiv2Rec.codh
      ExpresionNiv2Rec.etq = ExpresionNiv2Rec.etqh
    }

ExpresionNiv3 →
    {
      ExpresiónNiv4.tsh = ExpresiónNiv3.tsh
      ExpresionNiv4.etqh = ExpresionNiv3.etqh
      ExpresionNiv4.parh = ExpresionNiv3.parh
    }
    ExpresionNiv4
    {
      ExpresiónNiv3Fact.tsh = ExpresiónNiv3.tsh
      ExpresionNiv3Fact.tipoh = ExpresiónNiv4.tipo
      ExpresionNiv3Fact.modoh = ExpresionNiv4.modo
      ExpresionNiv3Fact.codh = ExpresionNiv4.cod
      Expresionniv3Fact.etqh = ExpresionNiv4.etq
    }
    ExpresionNiv3Fact
    {
      ExpresionNiv3.tipo = ExpresionNiv3Fact.tipo
      ExpresionNiv3.etq = ExpresionNiv3Fact.etq
      ExpresionNiv3.cod = ExpresionNiv3Fact.cod
    }


ExpresionNiv3Fact → OpNiv3
    {
      ExpresiónNiv3.tsh = ExpresiónNiv3Fact.tsh
      ExpresionNiv3.etqh = ExpresionNiv3Fact.etqh
      ExpresionNiv3.parh = false
    }
    ExpresiónNiv3
    {
      ExpresionNiv3Fact.modo = val
      ExpresionNiv3Fact.tipo =
            si (ExpresionNiv3Fact.tipoh = <t:error> v ExpresionNiv3.tipo = <t:error> v
                    ExpresionNiv3Fact.tipoh =/= <t:natural> v ExpresionNiv3.tipo =/=
    <t:natural>)
                  <t:error>
            sino <t:natural>
            ExpresionNiv3Fact.cod =
                  case (OpNiv3.op)
                        shl: ExpresionNiv3Fact.codh || ExpresionNiv3.cod || shl
                        shr: ExpresionNiv3Fact.codh || ExpresionNiv3.cod || shr
            ExpresionNiv3Fact.modo=val
      ExpresionNiv3Fact.etq = ExpresionNiv3.etq + 1
    }


ExpresionNiv3Fact → λ
    {
```

```
  ExpresionNiv3Fact.tipo = ExpresionNiv3Fact.tipoh
  ExpresionNiv3Fact.modo = ExpresionNiv3Fact.modoh
  ExpresionNiv3Fact.cod = {}
  ExpresionNiv3Fact.etq = ExpresionNiv3Fact.etqh
}
```

ExpresiónNiv4 → OpNiv4

```
{
  ExpresionNiv41.tsh = ExpresionNiv4o.tsh
  ExpresionNiv41.modo = val
  ExpresionNiv41.etqh = ExpresionNiv4.etqh
  ExpresionNiv41.parh = false
}
```

ExpresiónNiv4

```
{
  ExpresionNiv40.etq  = ExpresionNiv41.etq + 1
        ExpresionNiv40.tipo =
              si (ExpresionNiv4_1.tipo = <t:error>)
                    <t:error>
              sino case (OpNiv4.op)
                    no:
                          si (ExpresionNiv4_1.tipo=<t:boolean>)
                                <t:boolean>
                          sino   <t:error>
                    menos:
                          si (ExpresionNiv4_1.tipo=<t:float>)
                            <t:float>
                          sino si (ExpresionNiv4_1.tipo=<t:integer> v
                              ExpresionNiv4_1.tipo = <t:natural>)
                              <t:integer>
                          sino <t:error>
                    cast-float:
                          si (ExpresionNiv4_1.tipo=/= <t:boolean>)
                            <t:float>
                          sino <t:error>
                    cast-int:
                          si (ExpresionNiv4_1.tipo=/= <t:boolean>)
                            <t:integer>
                          sino <t:error>
                    cast-nat:
                          si (ExpresionNiv4_1.tipo=<t:natural> v
                              ExpresionNiv4_1.tipo=<t:character>)
                           <t:natural>
                          sino <t:error>
                    cast-char:
                          si (ExpresionNiv4_1.tipo=<t:natural> v
                              ExpresionNiv4_1.tipo=<t:character>)
                           <t:character>
                          sino <t:error>
```

```
            ExpresionNiv40.cod =
                  case (OpNiv4.op)
                        no:
                              ExpresionNiv41.cod || no
                        negativo:
                              ExpresionNiv41.cod || negativo
                        cast-float:
                              ExpresionNiv41.cod || CastFloat
                        cast-int:
                              ExpresionNiv41.cod || CastInt
                        cast-nat:
                              ExpresionNiv41.cod || CastNat
                        cast-char:
                              ExpresionNiv41.cod || CastChar
    }
ExpresiónNiv4 → '|'
    {
      Expresion.tsh = ExpresionNiv4.tsh
      Expresion.etqh     = ExpresionNiv4.etqh
      Expresion.parh     = false
    }
    Expresión '|'
    {
            ExpresionNiv4.tipo =
                  si (Expresion.tipo = <t:error> v Expresion.tipo=<t:boolean> v
                                    Expresion.tipo=<t:character>)
                        <t:error>
                  sino si (Expresion.tipo = <t:float>)
                        <t:float>
                  sino si (Expresion.tipo = <t:natural> v Expresion.tipo =
                                                            <t:integer>)
                        <t:natural>
                  sino   <t:error>
            ExpresionNiv4.cod  = Expresion.cod || abs
            ExpresionNiv4.modo = Expresion.val
            ExpresionNiv4.etq  = Expresion.etq
    }

ExpresiónNiv4 → '('
    {
      Expresion.tsh = ExpresionNiv4.tsh
      Expresion.etqh     = ExpresionNiv4.etqh
      Expresion.parh     = ExpresionNiv4.parh
    }
    Expresión ')'
    {
      ExpresionNiv4.tipo = Expresion.tipo
      ExpresionNiv4.modo = Expresion.modo
      ExpresionNiv4.cod  = Expresion.cod
      ExpresionNiv4.etq  = Expresion.etq
    }

ExpresiónNiv4 →
```

```
    {
            Literal.etqh = ExpresionNiv4.etqh
            Literal.tsh = ExpresiónNiv4.tsh
    }
    Literal
    {
            ExpresionNiv4.cod  = Literal.cod
            ExpresionNiv4.modo = var
            ExpresionNiv4.etq  = Literal.etq
            ExpresionNiv4.tipo = Literal.tipo
    }

ExpresionNiv4 →
    {
      Mem.tsh = ExpresionNiv4.tsh
      Mem.etqh        = ExpresionNiv4.etqh
    }
    Mem
    {
      ExpresionNiv4.cod    =
          si esCompatibleConTipoBasico(Mem.tipo,ExpresionNiv4.tsh) AND
                                                              not ExpresionNiv4.parh
              Mem.cod || apila-ind
          si no
              Mem.cod
      ExpresionNiv4.etq     =
          si esCompatibleConTipoBasico(Mem.tipo,ExpresionNiv4.tsh) /¥ not
    ExpresionNiv4.parh
              Mem.etq + 1
          si no
             Mem.etq
      ExpresionNiv4.modo = var
      ExpresionNiv4.tipo = Mem.tipo
    }

Literal → litNat
    {
      Literal.cod = apila LitNat.lex
      Literal.etq = Literal.etqh + 1
      Literal.tipo = <t:natural>
    }
Literal → litFlo
    {
      Literal.cod = apila litFlo.lex
      Literal.etq = Literal.etqh + 1
      Literal.tipo = <t:float>
    }

Literal → litTrue
    {
      Literal.cod = apila true
      Literal.etq = Literal.etqh + 1
      Literal.tipo = <t:boolean>
```

```
    }

Literal → litFalse
    {
      Literal.cod = apila false
      Literal.etq = Literal.etqh + 1
      Literal.tipo = <t:boolean>
    }

Literal → litCha
    {
      Literal.cod = apila litCha.lex
      Literal.etq = Literal.etqh + 1
      Literal.tipo = <t:character>
    }

Literal → litNull
      {
      Literal.tipo  = <t:integer>
      Literal.etq = Literal.etqh + 1
      Literal.cod = apila MIN_INT
      }

OpNiv0 → <
      {OpNiv0.op = menor}
OpNiv0 → >
      {OpNiv0.op = mayor}
OpNiv0 → <=
      {OpNiv0.op = menor-ig}
OpNiv0 → >=
      {OpNiv0.op = mayor-ig}
OpNiv0 → =
      {OpNiv0.op = igual}
OpNiv0 → =/=
      {OpNiv0.op = no-igual}
OpNiv1 → +
      {OpNiv1.op = suma}
OpNiv1 → -
      {OpNiv1.op = resta}
OpNiv1 → or
      {OpNiv1.op = o}
OpNiv2 → *
      {OpNiv2.op = multiplica}
OpNiv2 → /
      {OpNiv2.op = divide}
OpNiv2 → %
      {OpNiv2.op = modulo}
OpNiv2 → and
      {OpNiv2.op = y}
OpNiv3 → >>
      {OpNiv3.op = shl}
OpNiv3 → <<
      {OpNiv3.op = shr}
```

```
OpNiv4 → not
        {OpNiv4.op = no}
OpNiv4 → -
        {OpNiv4.op = menos}
OpNiv4 → (float)
        {OpNiv4.op = cast-float}
OpNiv4 → (int)
        {OpNiv4.op = cast-int}
OpNiv4 → (nat)
        {OpNiv4.op = cast-nat}
OpNiv4 → (char)
        {OpNiv4.op = cast-char}
```

9. Esquema de traducción orientado al traductor predictivo – recursivo

9.1. Variables globales

- ts: Almacena la tabla de símbolos.
- numVars: Almacena la siguiente dirección de memoria libre.
- cod: Almacena el código que se va generando
- n: nivel anidamiento actual
- etq: posición de la siguiente instrucción en el código
- dir: dirección de memoria
- pend: declaraciones de tipo pendientes
- params: lista de parámetros.

9.2. Nuevas operaciones y transformación de ecuaciones semánticas

Las siguientes operaciones devuelven un booleano indicando si el Token actual corresponde a lo esperado. En función de ese resultado se puede decidir entre varias reglas de la gramática, o se pueden detectar errores en el código.

- **in()**: sirve para comprobar que lo siguiente es un token "in"
- **out()**: sirve para comprobar que lo siguiente es un token "out"
- **Identificador(out: lex)**: reconoce el token del identificador léxico y devuelve su lexema
- **dosPuntosIgual():** reconoce :=. Afecta a InsAsignación.
- **ampersand()**: reconoce &. Afecta a Programa.
- **puntoYComa()**: reconoce ; Afecta a Declaraciones e Instrucciones.
- **barraVertical()**: reconoce |. Afecta a Instrucciones (valor absoluto)
- **AbrePar(), CierraPar()**: reconocen los paréntesis. Afectan a Instrucciones y Declaraciones de Bloques en varias partes de la gramática.
- **abreCorchete(), cierraCorchete()**: los corchetes se reconocen al procesar declaración y accesos a arrays.
- **punto()**: se necesita cuando se procesa el acceso a campos de un registro.
- **coma()**: reconoce las comas que separan los parámetros de los procedimientos.
- **flecha()**: reconoce las fechas que acceden a lo apuntado por un puntero.
- **var()**: reconoce el token de la declaración de un parámetro por variable.
- **new(), dispose()**: reconoce el token de 'new' y 'dispose'
- **reg(), freg():** reconocen los token de inicio y fin de la declaración de un registro.
- **if(), then(), else():** reconocen los token relativos a la instrucción if.
- **for(), while():** reconocen los tokens relativos a las instrucciones de bucles.

Otras operaciones:

- **insertaCod(cod c, Instruccion i, int i):** reemplaza la instrucción en la posición 'i' del código 'c' con la instrucción 'i'. Se utiliza cuando hay que introducir alguna instrucción con datos que se conocen tras generar otras instrucciones a posteriori. En estos casos, se introduce una noop que se reemplaza con la instrucción buena cuando se conoce. Se ha optado por hacerlo de esta manera para permitir declarar el código como variable global. En otro caso, no habría sido necesario.

9.3. Esquema de traducción

```
Programa(out: error1) →
    {
      etq = longInicio+1
      dir = 0
      n = 0
      ts = creaTS()
      cod = {}
    }
    Declaraciones(out: error2)
    ampersand()
    {
      cod += inicio(n,dir)
      cod += ir-a(etq)
    }
    Instrucciones(out: error3)
    {
      cod += stop
      error1= error2 v error 3
    }


Declaraciones(out: error1) → Declaracion(out: error2, tam2, id2, props2)
    {
      dir += tam2
      errorh3 = error2  v  (existeID(ts,id2)  ∧ ts[id2].nivel = n)
      ts = inserta(ts,id2, props2)
      pend - = ( si (props2.clase = tipo) {id2} sino {}
    }
    DeclaracionesRec(In: errorh3; out error3)
    {
      error1 = error3
    }


DeclaracionesRec(In: errorh1; out error1) →
      puntoYcoma()
       Declaracion(out: error2, tam2, id2, props2)
    {
      dir += tam2
      errorh3 = errorh1 v error2 v (existeID(ts,id2) ∧ ts[id2].nivel = n)
      ts = inserta(ts, id2, props2)
      pend - = ( si (props2.clase = tipo) {id2} sino {}
```

```
    }
    DeclaracionesRec(In: errorh3; out error3)
    {
      error1 = error3
    }


DeclaracionesRec(In: errorh1; out error1) → λ
    {
      error1 = errorh1
    }


Declaracion(out: error1, tam1, id1, props1) → DeclaracionTipo(out: error2, id2,props2)
{
      tam         = 0
      error1 = error2
      id1 = id2
      props1 = props2;
}

Declaracion(out: error1, tam1, id1, props1) → DeclaracionVariable(out: error2, id2,
props2)
    {
      id1 = id2
      props1 = props2 ++ <dir:dir>
      error1 = error2
      tam1 = props2.tipo.tam
    }


Declaracion(out: error1, tam1, id1, props1) → DeclaracionProcedimiento(out:
error2,id2,props2)
    {
      error1 = error2
      tam1 = 0
      id1= id2
      props1 = props2
    }



DeclaracionTipo(out: error1, id1, props1) →
      tipo()
      id(out: lex)
      igual()
      Tipo(out: error2, tipo2;)
    {
            id1=lex
            props1=<clase:tipo, tipo: tipo2, nivel: n>
            error1 = error2 v existeID(ts, lex) v ¬existeRef(ts,tipo2)
    }
```

```
DeclaraciónVariable(out: error1, id1, props1) →
      Tipo(out: error2,tipo2)
      id(out:lex)
    {
      id1 = lex
      props1= <clase:var, tipo:tipo2, nivel: n>
      error1 = error2 v existeID(ts, lex) v ¬existeRef(ts,tipo2)
    }


DeclaracionProcedimiento(out: error1, id1, props1) →
      procedure()
      id(out:lex)
    {
      ts_aux = ts;
      ts = creaTS(ts_aux)
      dir_aux=dir
      n += 1;
    }
    FParametros(out:error2)
    {
      ts = inserta(ts, lex, <clase:proc, tipo: <t:proc, parametros: params>,  nivel: n>)
      params = {}
      dir=dir_aux+dir
    }
    Bloque(out: error3, inicio3)
    {
      error1 = error2 v error3 v (existeID(FParametros.ts, id.lex)  ^ ts[lex].nivel = n)
      id1 = lex
      props1 = <clase:proc, tipo: <t:proc, parametros: params>, nivel: n, inicio: inicio3>
      n - = 1;
      ts = ts_aux
    }


Bloque(out: error1, inicio1) →
      Declaraciones(out: error2)
      ampersand()
    {
      inicio = etq
      etq + = longPrologo
      cod += prologo(n, dir)
    }
    Instrucciones(out: error3)
    {
      error1 = error2 v error3
      etq += longEpilogo + 1
      cod += epilogo(n)
      cod += ir-ind
    }
```

```
Bloque(out: error1, inicio1) →
      ampersand()
    {
      inicio1 = etq
      etq += longPrologo
      cod += prologo(n,dir)
    }
    Instrucciones(out: error2)
    {
      error1 = error2
      cod += epilogo(n)
      cod += ir-ind
            etq += longEpilogo + 1
    }


FParametros(out:error1) →
      abrePar()
    LFParametros(out: error2)
    {
      error1 = error2
    }
    cierraPar()


FParametros(out:error1) → λ
    {
      error1 = false
    }

LFParametros(out:error1) →
    FParametro(out: error2, id2, props2, tam2)
    {
      errorh3 = error2 v existeID(ts,id2)  ∧ ts[id2].nivel = n)
      ts = inserta(ts, id2, FParametro.props ++ <dir:0>)
      dir += tam2
    }
    LFParametrosRec(in:errorh3; out: error3)
    {
      LFParametros.error = error3
    }


LFParametrosRec(in:errorh1; out: error1) →
    coma()
    FParametro(out: error2, id2, props2, tam2)
    {
      dir += tam2
      errorh3 = errorh1 v error2 v (existeID(ts,id2)  ∧ ts[id2].nivel = n)
      ts = inserta(ts, id2, props2)
    }
    LFParametrosRec(in:errorh3; out: error3)
    {
```

```
      error1 = error3
    }

LFParametrosRec(in:errorh1; out: error1) → λ
    {
      error1 = errorh1
    }


FParametro(out: error1, id1, props1, tam1) →
      var()
      Tipo(out: error2, tipo2)
      id(out: lex)
    {
      tam1      = 1
      params    += <modo: variable, tipo: tipo2, dir: dir>
      id = lex
      props1 = <clase: pvar, tipo: tipo2, nivel: n>
      error1 = error2
    }

FParametro(out: error1, id1, props1, tam1) →
      Tipo(out: error2, tipo2)
      id(out: lex)
    {
      tam1 = tipo2.tam
      params    += <modo: valor, tipo: tipo2, dir: dir>
      id1 = lex
      props1 = <clase: var, tipo: tipo2, nivel: n>
      error1 = error2
    }

Tipo(out: error1, tipo1) →
    id(out: lex)
    {
       tipo1 =
           <
               t:ref,
               id:lex,
               tam:ts[lex].tipo.tam
           >
      error1 =  si existeID(ts,lex)
                   ts[lex].clase != tipo
                    sino
                   false
      pend +=  si (¬existeID(ts,lex))
                   {lex}
                    sino
                      □
    }

Tipo(out: error1, tipo1) → Boolean
    {
```

```
      tipo1 = <t:boolean,tam:1>
      error1 = false
    }

Tipo(out: error1, tipo1) → Character
    {
      tipo1 = <t:character,tam:1>
      error1 = false
    }

Tipo(out: error1, tipo1) → Float
    {
      tipo1 = <t:float,tam:1>
      error1 = false
    }

Tipo(out: error1, tipo1) → Natural
    {
      tipo1 = <t:natural,tam:1>
      error1 = false
    }

Tipo(out: error1, tipo1) → Integer
    {
      tipo1 = <t:integer,tam:1>
      error1 = false
    }

Tipo(out: error1, tipo1) →
      array()
      abreCorchete()
      num(out:lex)
      cierraCorchete()
      of()
      Tipo(out: error2, tipo2)
    {
      tipo1 =
                  <
                t:array,
                nelems:valorDe(lex),
                tbase:tipo2,
                tam:valorDe(lex)*tipo2.tam
            >
      error1 = tipo2.error v ¬existeRef (ts ,tipo2
    }

Tipo(out: error1, tipo1) → ^Tipo(out: error2, tipo2)
    {
      tipo1 =
            <
                t:puntero,
                tbase:tipo2,
                tam:1
```

```
            >
      error1 = error2
    }


Tipo(out: error1, tipo1) →
      reg()
    Campos(out: error2, campos2, tam2)
    freg()
    {
      Tipo.tipo =
              <
                  t:array,
                  campos: campos2,
                  tam: tam2
              >
            error1 = error2
    }

Campos(out: error1, campos1, tam1) →
    {
      desh2 = 0
    }
    Campo(in: desh2; out: error2, id2, campo2, tam2)
    {
      camposh3 = [campo2]
      errorh3 = error2
      desh3 = tam2
    }
    CamposRec(in: errorh3, camposh3, desh3; out: error3, campos3, tam3)
    {
      error1 = error3
      campos1 = campos3
      tam1 = tam3
    }



CamposRec(in: errorh1, camposh1 desh1; out: error1, campos1, tam1) →
      puntoYComa()
    {
      desh2 = desh1
    }
    Campo(in: desh2; out: error2, id2, campo2, tam2)
    {
      camposh3 = camposh1 ++ campo2
      errorh3 = errorh1 v error2 v existeCampo(camposh1, id2)
      desh3 = tam2 + desh1
    }
    CamposRec(in: errorh3, camposh3, desh3; out: error3, campos3, tam3)
    {
      error1 = error3
      campos1 = campos3
```

```
      tam1 = tam3
    }


CamposRec(in: errorh1, camposh1 desh1; out: error1, campos1, tam1) → λ
    {
      error1= errorh1;
      campos1 = camposh1
      tam1 = desh1
    }


Campo(in: desh1; out: error1, id1, campo1, tam1) →
    Tipo(out: error2, tipo2)
      id(out: lex)
    {
        campo1 =
            <
                id:lex,
                tipo:tipo2,
                desp:desh1
            >
        tam1   = tipo2.tam
      error1 = error2 v ¬existeRef(ts,tipo2)
    }


Instrucciones(out: error1) →
    Instrucción(out: error2)
    {
      errorh3 = error2
    }
    InstruccionesRec(in: errorh3; out: error3)
    {
      error1 = error3
    }


InstruccionesRec(in: errorh1; out: error1) →
    puntoYComa()
    Instrucción(out: error2)
    {
      errorh3=error1 v errorh1
    }
    InstruccionesRec(in: errorh3; out: error3)
    {
      error1 = error3
    }



InstruccionesRec(in: error1h; out: error1) → λ
    {
```

```
      error1 = error1h
    }



Instrucción(out: error1) →
    InsProcedimiento(out: error 2)
    {
      error1 = error2
    }

Instrucción(out: error1) →
    InsLectura(out: error 2)
    {
      error1 = error2
    }

Instrucción(out: error1) →
    InsEscritura(out: error 2)
    {
      error1 = error2
    {

Instrucción(out: error1) →
    InsAsignación(out: error 2)
    {
      error1 = error2
    }

Instrucción(out: error1) →
    InsCompuesta(out: error 2)
    {
      error1 = error2
    }

Instrucción(out: error1) →
    InsIf(out: error 2)
    {
      error1 = error2
    }

Instrucción(out: error1) →
    InsWhile(out: error 2)
    {
      error1 = error2
    }

Instrucción(out: error1) →
    InsFor(out: error 2)
    {
      error1 = error2
    }
Instrucción(out: error1) →
```

```
    InsNew(out: error 2)
    {
      error1 = error2
    }

Instrucción(out: error1) →
    InsDis(out: error 2)
    {
      error1 = error2
    }

InsProcedimiento(out: error1) →
      id(out: lex)
    {
      params = ts[lex].tipo.parametros
      cod + = apilar-ret
      etq += longApilaRet
    }
    AParametros(out: error2)
    {
      error1 = error2 v ¬existeID(lex) v ts[lex].clase != proc
      cod += ir-a(ts[lex].inicio)
      etq += 1
    }

AParametros(out: error1) →
    abrePar()
    {
      etq += longInicioPaso
      cod += inicio-paso
    }
    LAParametros(out: error2)
    cierraPar()
    {
      error1 = error2
      cod += fin-paso
      etq += longFinPase
    }
AParametros(out: error1) → λ
    {
      error1 = |params| > 0
    }

LAParametros(out: error1) →
    {
      etq += longDireccionParFormal + 1
      cod += copia
      cod += direccionParFormal(fparamsh1[0])
             parh2 = params[0].modo == var
    }
    Expresion(in: parh2; out: tipo2, modo2)
    {
      nparam_h3 = 1
```

```
      errorh3 =     (tipo2 == <t:error>) v
                    (|params|) < 1 v (params[0].modo = var ∧ modo2 = val) v
                    NOT compatibles(params[0].tipo, tipo, ts)
      cod += pasoParametro(modo2,params[0])
      etq += longPaseoParametro
    }
    LAParametrosRec(nparamh3, errorh3; out: error3)
    {
      error1 = error3
    }


LAParametrosRec(in: nparamh1, errorh1; out: error1) →
    coma()
    {
      etq += longDireccionParFormal + 1
      cod += copia
      cod += direccionParFormal(fparamsh1[nparamh1])
      parh2 = params[nparamh1].modo == var
    }
    Expresion(in: parh2; out: tipo2, modo2)
    {
      nparamh3 = nparamh1 + 1
            errorh3 = errorh1 v tipo2==<t:error> v |params| < nparamh1 + 1 v
                        (params[nparamh1].modo ==var  ∧ modo2 == val) v
                        NOT compatibles(params[nparamh1].tipo, tipo2, ts)
            etq += longPasoParametro
            cod += pasoParametro(modo2,params[nparamsh1])
    }
    LAParametrosRec(in: nparamh3, errorh3; out: error3)
    {
      error1 = error3
    }


LAParametrosRec(nparamh3, errorh3; out: error3) → λ
    {
      error3= errorh3
    }

InsLectura(out: error1) →
      in()
      abrePar()
      id(out:lex)
      cierraPar()
    {
      error1 = NOT existeID(ts,lex)
      cod+= accesoVar(ts[lex].props);
      etq+=longAccesoVar
      cod += InsEntrada
      cod += desapilaInd
      etq += 2
    }
```

```
InsEscritura(out: error1) →
       out()
       abrePar()
     {

       parh2 = false

     }
     Expresion(in: parh2; out: tipo2, modo2)
     {
       error1 = (tipo2 = <t:error>)
       etq += 1
       cod += out
     }
     cierraPar()


InsAsignación(out: error1) →
     Mem(out: tipo2)
     dosPuntosIgual()
     {
       parh3 = false
     }
     Expresión(in: parh3; out: tipo3, modo3)
     {
       etq += 1
       error1 = ¬ esCompatible(tipo2, tipo3,ts)
       si esCompatibleConTipoBasico(tipo2, ts)
              cod += desapila-ind
       si no
              cod += mueve(tipo2.tam)
     }




InsCompuesta(out: error1) →
       abreCorchete()
     Instrucciones(out: error2)
     cierraCorchete()
     {
       error1 = error2
     }

InsIf(out: error1) →
       if ()
     {
       parh2 = false
     }
     Expresion(in: parh2; out: tipo2, modo2)
     then ()
     {
```

```
        etq += 1
        cod += noop
        aux1 = etq
    }
    Instrucción(out: error3)
    {
        insertar(cod, ir-f(etq+1), aux2 -1)
              etq += 1
              aux2=etq
              cod + = noop
    }
    Pelse(out: error4)
    {
        insertar(cod, ir-a(etq),aux2 -1)
        error1 = tipo2 != <t:bool> v error3 v error4
        InsIf.error = Expresion.tipo != <t:bool> v Instrucción.error v Pelse.error
    }

PElse(out: error1) →
        else()
    Instrucción(out: error2)
    {
        error1 = error2
    }

PElse(out: error1) → λ
    {
        error1 = false
    }

InsWhile(out:error1) →
        while ()
        {
              etq_while = etq
        }
    Expresion(in: parh2; out: tipo2, modo2)
    do ()
    {
        etq += 1
        aux = etq
        cod += noop
    }
    Instrucción(out: error3)
    {
        insertaCod(cod, ir-f(etq), aux -1)
        etq += 1
        cod+= ir-a(etq_while)
        error1 = tipo2 != <t:bool> v error3
    }

InsFor(out: error1) →
        for()
        {
```

```
            etq_for = etq
            parh2= false
      }
      id(out: lex)
      igual()
    Expresion(in: parh2; out: tipo2, modo2)
    to()
    {
      parh3 = false
      cod+= accesoVar(ts[lex].props);
      etq += 1 + longAccesoVar
      cod += desapila-ind
    }
    Expresion(in: parh3, out: tipo3,modo3)
    do()
    {
      cod+= copia
      cod += accesoVar(ts[lex].props);
      etq+=4 + longAccesoVar
      cod+= apilar-ind
      cod+= igual
      cod+= noop
      aux = etq - 1
    }
    Instruccion(out: error4)
    {
      error1 = error4 v (tipo2 != <t:natural> AND tipo2 != <t:integer>) v
                  (tipo3 != <t:natural> AND tipo2 != <t:integer>) v
                  (ts[lex].tipo != <t:natural> AND ts[lex].tipo != <t:integer>

      cod += apila-dir ts[lex].dir
      cod += apilar 1
      cod += sumar
      cod += desapila-dir ts[lex].dir
      cod += ir-a(aux)
      cod += desapila
      etq += 6
      insertaCod(cod, ir-v(etq ), aux)
    }

InsNew(out: erro1) →
      new()
    Mem(out: tipo2)(
    {
      error1 = tipo2 != <t:puntero>
      etq += 2
      cod += new
                  si tipo2.tbase = ref
                        ts[tipo2.tbase.id].tam
                  si no
                        tipo2.tbase.tam
                   )
            cod+= desapila-ind
```

```
    }

InsDis(out: error1) →
      dispose ()
    Mem(out: tipo2)
    {
      error1 = tipo2.t != puntero
      etq+=2
      cod+= apilaInd
      si tipo2.tbase == ref
            cod+= dis ts[tipo2.tbase.id].tam
      else
            cod+= dis tipo2.tbase.tam
    }


Mem(out: tipo1) → id(out: lex)
    {

      tipo2h = si existe(ts,lex)
                                si ts[lex].clase == var
                                      ref!(ts[lex].tipo,ts)
                                sino <t:error>
                  sino <t:error>
      etq+=longAccesoVar(ts[lex])
      cod+=accesoVar(ts[lex])
    }
    MemRec(in: tipo2h; out: tipo2)
    {
      tipo1=tipo2;
    }


MemRec(in: tipo1h; out: tipo1) → '^'
    {
      tipo2h = si (tipo1h.t = puntero
                                ref!(tipo1h.tbase,ts)
                sino <t:error>
      etq += 1;
      cod+= apila-ind
    }
    MemRec(in: tipo2h; out: tipo2)
    {
      tipo1 = tipo2
    }



MemRec(in: tipo1h; out: tipo1) →
      abreCorchete()
    {
      parh2=false
    }
```

```
    Expresion(in: parh2; out: tipo2,modo2)
    cierraCorchete()
    {
      etq += 3
      cod += apila tipo1h.tbase.tam
      cod += multiplica
      cod += suma
      tipo3h = si tipo1h.t == array AND tipo2.t = num
                          ref!(tipo1h.tbase,ts)
                 sino <t:error>
    }
    MemRec(in: tipo3h; out: tipo3)
    {
      tipo1 = tipo3
    }


MemRec(in: tipo1h; out: tipo1) →
      punto()
      id(out:lex)
    {
      etq += 2
      cod += apila(tipo1h.campos[id.lex].desp
      cod += suma
      tipo2h = si tipo1h.t = rec
                                 si campo?(tipo1h.campos,lex)
                                 ref!(tipo1h.campos[lex].tipo,ts)
                           sino <t:error>
                  sino <t:error>
    }
    MemRec(in: tipo2h; out: tipo2)
    {
      tipo1 = tipo2
    }

MemRec(in: tipo1h; out: tipo1) → λ
    {
      tipo1 = tipo1h
    }

Expresion(in: parh1; out: tipo1, modo1) →
    {
      parh2 = parh1
    }
    ExpresiónNiv1(in: parh2; out: tipo2, modo2)
    {
      tipoh3 = tipo2
      modoh3 = modo2
    }
    ExpresiónFact(in: tipoh3, modoh3; out: tipo3, modo3)
    {
      tipo1 = tipo3
      modo1 = modo3
```

```
    }

ExpresionFact(in:  tipoh1, modoh1; out: tipo1, modo1) →
      OpNiv0(out: op)
    {
      parh2 = false
    }
    ExpresionNiv1(in: parh2; out: tipo2, modo2)
    {
      tipo1 = si (tipoh1 = <t:error> v tipo2 = <t:error>) v
                        ( tipoh1 = <t:character> v tipo2 =/= <t:character>) v
                        ( tipoh1 =/= <t:character> ∧  tipo2 = <t:character>)
                        (tipoh1 = <t:boolean> ∧  tipo2 =/= <t:boolean>) v
                        ( tipoh1 =/= <t:boolean> ∧  tipo2 = <t:boolean>))
                              error
                    sino  boolean
      modo1 = val
      etq += 1
      cod +=
            case (op):
                        menor: menor
                        mayor: mayor
                        menor-ig: menorIg
                        mayor-ig: mayorIg
                        igual : igual
                        no-igual: no-igual
    }

ExpresionFact(in: tipoh1, modoh1; out: tipo1, modo1) → λ
    {
      modo1 = modoh1
      tipo1 = tipoh1
    }

ExpresiónNiv1(in: parh1; out: tipo1, modo1) →
    {
      parh2 = parh1
    }
    ExpresiónNiv2(in: parh2; out: tipo2, modo2)
    {
      tipoh3 = tipo2
      modoh3 = modo2
    }
    ExpresionNiv1Rec(in: tipoh3, modoh3; out: tipo3, modo3)
    {
      tipo1 = tipo3
      modo1 = modo3
    }


ExpresiónNiv1Rec(in: tipoh1, modoh1; out: tipo1, modo1) →
      OpNiv1(out: op)
    {
```

```
      parh2= false
      if (opNiv1.op == or):
            aux = etq +1
            etq += 3

            cod += dup
            cod += noop
            cod += desapila
    }
    ExpresiónNiv2(in: parh2; out: tipo2, modo2)
    {
      modoh3 = modo2
      tipoh3 = si (tipoh1 = <t:error> v tipo2 = <t:error> v
                  tipoh1 = <t:character> v tipo2 = <t:character> v
                  (tipoh1 = <t:boolean> ∧  tipo2 =/= <t:boolean>) v
                    (tipoh1 =/= <t:boolean> ∧  tipo2 = <t:boolean>))
                   <t:error>
            sino case (op)
                   suma,resta:
                            si (tipoh1=<t:float> v tipo2 = <t:float>)
                          <t:float>
                            sino si (tipoh1 = <t:integer> v tipo2 = <t:integer>)
                           <t:integer>
                            sino si (tipoh1 = <t:natural> ∧  tipo2 = <t:natural>)
                           <t:natural>
                              sino      <t:error>
                   o:
                            si (tipoh1 = <t:boolean> ∧  tipo2 = <t:boolean>)
                              <t:boolean>
                            sino <t:error>
      modoh3 =val
      if (opNiv1.op == or):
            insertaCod(cod,ir-v(etq), aux)
      else:
            etq += 1
            case (op)
                   suma: cod+= sumar
                   resta: cod+= restar
    }
    ExpresiónNiv1Rec(in: tipoh3, modoh3; out: tipo3, modo3)
    {
      tipo1 = tipo3
      modo1 = val
    }




ExpresionNiv1Rec(in: tipoh1, modoh1; out: tipo1, modo1) → λ
    {
      tipo1= tipoh1
      modo1 = modoh1
    }
```

```
ExpresiónNiv2(in: parh1; out: tipo1, modo1) →
    {
      parh2 = parh1
    }
    ExpresionNiv3(in: parh2; out: tipo2, modo2)
    {
      tipoh3 = tipo2
      modoh3 = modo2
    }
    ExpresiónNiv2Rec(in: tipoh3, modoh3; out: tipo3, modo3)
    {
      tipo1 = tipo3
      modo1 = modo3
    }


ExpresiónNiv2Rec(in: tipoh1, modoh1; out: tipo1, modo1) →
      OpNiv2 (out: op)
    {
      if (OpNiv2.op == and)
            aux = etq
            etq += 1
            cod += noop
      parh2 = false
    }
    ExpresiónNiv3(in: parh2; out: tipo2, modo2)
    {
      modoh3 = val
      tipoh3 =
            si (tipoh1 = <t:error> v tipo2 = <t:error> v
                          tipoh1 = <t:character> v tipo2 = <t:character> v
                (tipoh1 = <t:boolean> ∧  tipo2 =/= <t:boolean> v
                (tipoh1 =/= <t:boolean> ∧  tipo2 = <t:boolean>))
                     <t:error>
            sino case (op)
                  multiplica,divide:
                           si (tipoh1=<t:float> v tipo2 = <t:float>)
                               <t:float>
                           sino si (tipoh1 = <t:integer> v tipo2 = <t:integer>)
                               <t:integer>
                           sino si (tipoh1 = <t:natural> ∧  tipo2= <t:natural>)
                               <t:natural>
                           sino <t:error>
                  modulo:
                           si (tipo2 = <t:natural> ∧
                          (tipoh1= <t:natural> v tipoh1 = <t:integer>))
                               tipoh1
                           sino     <t:error>
                  y:
                           si (tipoh1 = <t:boolean> ∧ tipo2 = <t:boolean>)
                             <t:boolean>
                           sino <t:error>
```

```
      if (op == and)
            insertaCod(cod,ir-f(etq+1),aux)
            etq += 2
            cod += ir-a(etq + 2)
            cod += apila(false)
      else:

            cod += case(op)
                        Multiplica: mul
                        Divide: div
                        Modulo: Mod
            etq += 1
            modo1 = val
    }
    ExpresiónNiv2Rec(in: tipoh3, modoh3; out: tipo3, modo3)
    {
      tipo1 = tipo3
      modo1 = modo3
    }

ExpresiónNiv2Rec(in: tipoh1, modoh1; out: tipo1, modo1) → λ
    {
      tipo1 = tipoh1
      modo1 = modoh1
    }

ExpresionNiv3(in: parh1; out: tipo1, modo1) →
    {
      parh2 = parh1
    }
    ExpresionNiv4(in: parh2; out: tipo2, modo2)
    {
      tipoh3 = tipo2
      modoh3 = modo2
    }
    ExpresionNiv3Fact(in: tipoh3, modoh3; out: tipo3, modo3)
    {
      tipo1 = tipo3
      modo1  = modo3
    }


ExpresionNiv3Fact(in: tipoh1, modoh1; out: tipo1, modo1) →
      OpNiv3(out: op)
    {
      ExpresionNiv3.parh = false
    }
    ExpresiónNiv3(in: parh2; out: tipo2, modo2)
    {
      modo1 = val
      tipo1 =
            si (tipoh1 = <t:error> v tipo2 = <t:error> v tipoh1 =/= <t:natural> v tipo2
    =/= <t:natural>)
```

```
                    <t:error>
              sino <t:natural>
       etq += 1
              case (op)
                           shl: cod += shl
                           shr: cod += shr
    }

ExpresionNiv3Fact(in: tipoh1, modoh1; out: tipo1, modo1) → λ
    {
              tipo1 = tipoh1
              modo1 = modoh1
    }
ExpresiónNiv4(in: parh1; out: tipo1, modo1) →
       OpNiv4(out: op)
    {
              parh2 = false
    }
    ExpresiónNiv4(in: parh2; out: tipo2, modo2)
    {
       modo1 = val
              tipo1 =
                     si (tipo2 = <t:error>)
                               <t:error>
                     sino case (op)
                           no:
                                    si (tipo2 =<t:boolean>)
                                             <t:boolean>
                                    sino   <t:error>
                           menos:
                                  si ( tipo2 =<t:float>)
                                     <t:float>
                                    sino si (tipo2 =<t:integer> v tipo2 = <t:natural>)
                                        <t:integer>
                                    sino <t:error>
                           cast-float:
                                    si (tipo2 =/= <t:boolean>)
                                      <t:float>
                                    sino <t:error>
                           cast-int:
                                    si (tipo2 =/= <t:boolean>)
                                      <t:integer>
                                    sino <t:error>
                           cast-nat:
                                    si ( tipo2 =<t:natural> v tipo2 =<t:character>)
                                     <t:natural>
                                    sino <t:error>
                           cast-char:
                                    si (tipo2 =<t:natural> v tipo2 =<t:character>)
                                     <t:character>
                                    sino <t:error>
              etq += 1
              case (OpNiv4.op)
```

```
                    no:
                           cod += no
                    negativo:
                           cod += negativo
                    cast-float:
                           cod += CastFloat
                    cast-int:
                           cod += CastInt
                    cast-nat:
                           cod += CastNat
                    cast-char:
                           cod += CastChar
    }



ExpresiónNiv4(in: parh1; out: tipo1, modo1) →
      barraVertical()
    {
      parh2     = false
    }
    Expresión(in: parh2; out: tipo2, modo2)
    barraVertical()
    {
      tipo1 =
                    si (tipo2 = <t:error> v tipo2 =<t:boolean> v tipo2 = <t:character>)
                                 <t:error>
                    sino si (tipo2 = <t:float>)
                           <t:float>
                    sino si (tipo2 = <t:natural> v tipo2 = <t:integer>)
                           <t:natural>
                    sino   <t:error>
             cod += abs
             modo = val
    }

ExpresiónNiv4(in: parh1; out: tipo1, modo1) →
      abreParentesis()
    {
      parh2 = parh1
    }
    Expresión(in: parh2; out: tipo2, modo2)
    {
             tipo1 = tipo2
             modo1 = modo2
    }
    cierraParentesis()

ExpresiónNiv4(in: parh1; out: tipo1, modo1) →
    Literal(out: tipo2)
    {
             modo1 = var
             tipo1 = tipo2
```

```
    }

ExpresionNiv4(in: parh1; out: tipo1, modo1) →
    Mem(out: tipo2)
    {
      si esCompatibleConTipoBasico(tipo2,ts) /¥ NOT parh1
            cod += apila-ind
            etq += 1
            modo1 = var
      else modo = val
      tipo1 = tipo2
    }

Literal(out: tipo) → litNat(out: lex)
    {
      cod = apila lex
      etq += 1
      tipo = <t:natural>
    }
Literal(out: tipo) → litFlo(out: lex)
    {
      cod = apila lex
      etq += 1
      tipo = <t:float>
    }

Literal(out: tipo) → litTrue(out: lex)
    {
      cod = apila lex
      etq += 1
      tipo = <t:boolean>
    }

Literal(out: tipo) → litFalse(out: lex)
    {
      cod = apila lex
      etq += 1
      tipo = <t:boolean>
    }

Literal(out: tipo) → litCha(out: lex)
    {
      cod = apila lex
      etq += 1
      tipo = <t:character>
    }

OpNiv0(out: op) → <
      {op = menor}
OpNiv0(out: op) → >
      {op = mayor}
OpNiv0(out: op) → <=
      {op = menor-ig}
```

```
OpNiv0(out: op) → >=
        {op = mayor-ig}
OpNiv0(out: op) → =
        {op = igual}
OpNiv0(out: op) → =/=
        {op = no-igual}
OpNiv1(out: op) → +
        {OpNiv1.op = suma}
OpNiv1(out: op) → -
        {op = resta}
OpNiv1(out: op) → or
        {op = o}
OpNiv1(out: op) → *
        {op = multiplica}
OpNiv2(out: op) → /
        {op = divide}
OpNiv2(out: op) → %
        {op = modulo}
OpNiv2(out: op) → and
        {op = y}
OpNiv3(out: op) → >>
        {op = shl}
OpNiv3(out: op) → <<
        {op = shr}
OpNiv4(out: op) → not
        {op = no}
OpNiv4(out: op) → -
        {op = menos}
OpNiv4(out: op) → (float)
        {op = cast-float}
OpNiv4(out: op) → (int)
        {op = cast-int}
OpNiv4(out: op) → (nat)
        {op = cast-nat}
OpNiv4(out: op) → (char)
        {op = cast-char}
```

10. Formato de representación del código P

El código P se representa en binario. Cada instrucción de la máquina virtual se representa con un código (1 byte) seguido, si procede, del tipo de argumento (1 byte) y del valor en binario del argumento (4 bytes para números, 1 byte para char y booleanos). Las operaciones que se pueden realizar en este código están descritas en el punto 5.

11 Notas sobre la implementación

Descripción de la implementación realizada.

11.1. Descripción de archivos

Hemos divido el proyecto es 3 paquetes principales:

- **Compilador:** Contiene las clases relativas al traductor de lenguaje. Este paquete contiene 3 paquetes:
    1. Lexico: contiene la clase AnalizadorLexico, que recibe un flujo de entrada y devuelve un ArrayList de Tokens. Este paquete contiene un paquete Tokens con clases que corresponden a cada uno de los tipos de tokens que maneja el Analizador Léxico.
    2. TablaSimbolos: Contiene varias clases, la principal es GestorTS. Este gestor es una pila de Tablas de Símbolos(clase TablaSimbolos). A su vez esta tabla de símbolos tiene información como modo, tipo(clase TipoTs), dirección y nivel. El tipo se apoya en otras clases como Campo.
    3. Traductor: Contiene las clases necesarias para traducir un ArrayList de Tokens procedente del analizador léxico a un ArrayList de Objetos que contendrá el código binario. La clase que gestiona la traducción es Traductor y existen otras como ErrorTraductor que gestiona los errores.
- **Interfaz:** Contiene las interfaces utilizadas de cara al usuario. Como hemos implementado dos programas separados (compilador e intérprete) tenemos dos interfaces:
    1. Compilador: Contiene la interfaz del compilador. Esta interfaz da la opción de introducir el código en la propia interfaz o cargarla desde un fichero, una vez cargado puedes compilar y ver el código pila o ejecutar (en cuyo caso compilará y luego ejecutará el programa) También nos ofrece ejecutar el código en modo Traza (mostrando el contenido de la pila y la memoria en cada instrucción además de las entradas/salidas del programa) o en modo Normal (mostrando únicamente las entradas/salidas del programa)
    2. Pila: Contiene una interfaz que hemos utilizado para probar el intérprete a pila. Este panel hace de intermediario entre el bytecode del lenguaje a pila y el explicado en clase (con sentencias alfanuméricas como "apila 3" o "suma"). Al descompilar un archivo en bytecode este se mostrará como cadenas alfanuméricas. Al compilar, el texto escrito será traducido a lenguaje de pila, siempre y cuando su sintaxis sea correcta
- **Pila:** Contiene el intérprete encargado de simular la ejecución del código . Como tenemos dos tipos de código (código P y código J) este paquete contiene dos paquetes:
    1. Intérprete: Se encarga de ejecutar el código P generado por el compilador. Contiene 3 paquetes que se encargan de gestionar tanto los datos, como las instrucciones y excepciones que pueden surgir en el código y 3 clases que son las principales (EscritorPila, Interprete y LectorPila) que gestionan la entrada/salida de la ejecución y la propia ejecución. En este cuatrimestre hemos creado dos clases mas (Memoria y Huecos) para gestionar la memoria estática y el heap.