

Procesadores de Lenguaje: Práctica Anual
Ingeniería en Informática 4º C
Facultad de Informática UCM (2009-2010)

2º Entrega

Miguel Ángel Alonso

David García Higuera

Francisco Huertas Ferrer

1 Definición léxica del lenguaje

Letras:

a ≡ a
b ≡ b
c ≡ c
...
z ≡ z
A ≡ A
B ≡ B
C ≡ C
...
Z ≡ Z

Palabras reservadas

boolean ≡ {b}{o}{o}{l}{e}{a}{n}
character ≡ {c}{h}{a}{r}{a}{c}{t}{e}{r}
natural ≡ {n}{a}{t}{u}{r}{a}{l}
integer ≡ {i}{n}{t}{e}{g}{e}{r}
float ≡ {f}{l}{o}{a}{t}

and ≡ {a}{n}{d}
or ≡ {o}{r}
not ≡ {n}{o}{t}

if ≡ {i}{f}
then ≡ {t}{h}{e}{n}
else ≡ {e}{l}{s}{e}
null ≡ {n}{u}{l}{l}
record ≡ {r}{e}{c}{o}{r}{d}
array ≡ {a}{r}{r}{a}{y}
of ≡ {o}{f}
pointer ≡ {p}{o}{i}{n}{t}{e}{r}
tipo ≡ {t}{i}{p}{o}
procedure ≡ {p}{r}{o}{c}{e}{d}{u}{r}{e}
forward ≡ {f}{o}{r}{w}{a}{r}{d}
var ≡ {v}{a}{r}
dispose ≡ {d}{i}{s}{p}{o}{s}{e}
new ≡ {n}{e}{w}
while ≡ {w}{h}{i}{l}{e}
do ≡ {d}{o}
for ≡ {f}{o}{r}
to ≡ {t}{o}
-> ≡ {-}{>}
. ≡ {.}

(nat) ≡ {(){n}{a}{t}{}}
(int) ≡ {(){i}{n}{t}{}}
(char) ≡ {(){c}{h}{a}{r}{}}
(float) ≡ {(){f}{l}{o}{a}{t}{}}

true ≡ {t}{r}{u}{e}

false \equiv {f}{a}{l}{s}{e}

in \equiv {i}{n}

out \equiv {o}{u}{t}

& \equiv &

: \equiv :

; \equiv ;

=/= \equiv {=}{/}{=}

(\equiv \ (

) \equiv \)

< \equiv <

> \equiv >

+ \equiv \ +

- \equiv \ -

* \equiv \ *

/ \equiv \ /

% \equiv \ %

- (unario) \equiv \ -

\equiv

\ ' \equiv \ '

| \equiv \ |

[\equiv \ [

] \equiv \]

{ \equiv \ {

} \equiv \ }

Sentencias

letra \equiv [a..z | A..Z]

digito \equiv [0..9]

iden \equiv {letra} ({letra} | {digito}) *

booleanvalue \equiv {true} | {false}

natural \equiv 0 | ([1..9]{digito}*)

entero \equiv {-}? {natural}

real \equiv {natural} ((. {digito}* [1..9]) | ((E | e) -? {natural}) | (. {digito}* [1..9])

(E | e) -? {natural}))

exponente \equiv (E|e) -? {natural}

real \equiv {natural} ((0 | ({digito}* [1..9])) | ((E | e) -? {natural}) | ((0 | ({digito}* [1..9])) (E | e) -? {natural}))

Definición alternativa para entender la real de arriba (definición descompuesta)

decimal \equiv . (0 | ({digito}* [1..9]))

exponente \equiv (E|e) -? {natural}

real \equiv {natural} ({decimal} | {exponente} | ({decimal} {exponente}))

character \equiv \ (Cualquier elemento del conjunto de caracteres numéricos y alfabéticos de los cuales dispone una computadora) \ '

2 Definición sintáctica del lenguaje

2.1 Descripción de los operadores

Operadores Lógicos			
Operador	Aridad	Asociatividad	Prioridad
or	2	Izquierda	1
and	2	Izquierda	2
Not	1	---	4

Operadores Aritméticos			
Operador	Aridad	Asociatividad	Prioridad
+ (suma)	2	Izquierda	1
- (resta)	2	Izquierda	1
* (multiplicación)	2	Izquierda	2
/ (división)	2	Izquierda	2
% (modulo)	2	Izquierda	2

Operadores Relacionales			
Operador	Aridad	Asociatividad	Prioridad
< (menor que)	2	---	0
> (mayor que)	2	---	0
<= (menor igual que)	2	---	0
>= (mayor igual que)	2	---	0
!= (distinto)	2	---	0

Operadores de Desplazamiento			
Operador	Aridad	Asociatividad	Prioridad
<<	2	Derecha	3
>>	2	Derecha	3

Operadores de Conversión			
Operador	Aridad	Asociatividad	Prioridad
(float)	1	---	4
(int)	1	---	4
(nat)	1	---	4
valor	1	---	4
(char)	1	---	4

2.2 Formalización de la sintaxis

```
PROGRAMA ≡ {DECS} {&} {SENTS}
PROGRAMA ≡ {&} {SENTS}
DECS ≡ {DECS} ; {DEC}
DECS ≡ {DEC}
DEC ≡ {DECTIP}
DEC ≡ {DECVAR}
DEC ≡ {DECPROC}
DECVAR ≡ {iden} : { TIPOIDEN }
TIPOIDEN ≡ {boolean} | {carácter} | { natural} | {integer} | {float}|
{iden} |{record} {/{ }{CAMPOS} {/}} |{pointer} {TIPOIDEN}| {array} {[ ]
{natural} { } } {of} {TIPOIDEN}
CAMPOS ≡ {CAMPOS} {;} {CAMPO}
CAMPOS ≡ {CAMPO}
CAMPO ≡ {iden} : {TIPOIDEN}
DECTIP ≡ {tipo} {iden } {=} {TIPOIDEN}
DECPROC ≡ {proc} {iden} {DPARAMS} {PBLOQUE}
PBLOQUE ≡ {forward}
PBLOQUE ≡ { } {DECS} {&} {SENTS} { }
PBLOQUE ≡ { } {&} {SENTS} { }
DPARAMS ≡ { ( ) } {LISTAPARAMS} { }
DPARAMS ≡ {lambda}
LISTAPARAMS ≡ {LISTAPARAMS} { , } {PARAM}
LISTAPARAMS ≡ {PARAM}
PARAM ≡ {var} {iden} { : } {TIPOIDEN}
PARAM ≡ {iden} { : } {TIPOIDEN}

SENTS ≡ {SENTS} ; {SENT}
SENTS ≡ {SENT}
SENT ≡ {SASIG}
SENT ≡ {SWRITE}
SENT ≡ {SREAD}
SENT ≡ {SBLOQUE}
SENT ≡ {SIF}
SENT ≡ {SWHILE}
SENT ≡ {SFOR}
SENT ≡ {SNEW}
SENT ≡ {SDEL}
SWRITE ≡ {out} { ( ) } {EXP} { }
SREAD ≡ {in} { ( ) } {iden} { }
SASIG ≡ {mem} { := } {EXP}
SBLOQUE ≡ { } {SENTS} { }
SIF ≡ {if} {EXP} {then} {SENT} {PELSE}
PELSE ≡ {else} {SENT}
PELSE ≡ {lambdaλ}
SWHILE ≡ {while} {EXP} {do} {SENT}

SFOR ≡ {for} {mem} {=} {EXP} {to} {EXP} {do} {SENT}

SFOR ≡ {for} {iden} {=} {EXP} {FORC}
FORC ≡ {to} {EXP} {do} {SENT}

SCALL ≡ {iden} {PPARAMS}
PPARAMS ≡ { ( ) } {LPARAMS} { }
PPARAMS ≡ λ
LPARAMS ≡ {LPARAMS} { , } {EXP}
LPARAMS ≡ {EXP}
```

```

SNEW ≡ {new} {mem}
SDEL ≡ {dispose} {mem}MEM = {iden} {RMEM}
RMEM = λ
RMEM = {->} RMEM
RMEM = {[ ] {EXP} { ] } RMEM |
RMEM = { . } {iden} RMEM

EXP ≡ {EXP1} {OP0} {EXP1}
EXP ≡ {EXP1}
EXP1 ≡ {EXP1} {OP1} {EXP2}
EXP1 ≡ {EXP2}
EXP2 ≡ {EXP2} {OP2} {EXP3}
EXP2 ≡ {EXP3}
EXP3 ≡ {EXP4} {OP3} {EXP3}
(Nota: asocia a derechas)
{EXP3} ≡ {EXP4}
{EXP4} ≡ {OP4_1} {TERM}
//{EXP4} ≡ {OP4_2} {TERM} {OP4_2}
{EXP4} ≡ { | } {TERM} { | }
{EXP4} ≡ {TERM}
TERM ≡ mem
OP0 ≡ {<} | {>} | {<=} | {>=} | {=} | {/= }
OP1 ≡ {+} | {-} | {or}
OP2 ≡ {*} | {/} | {%} | {and}
OP3 ≡ {<<} | {>>}
OP4_1 ≡ {not} | {(nat)} | {(int)} | {(char)} | {(float)}
| ≡ { | }

```

3 Estructura y construcción de la tabla de símbolos

Posibles campos de la tabla de símbolos:

1. Nombre del identificador.
2. Dirección en tiempo de ejecución a partir de la cual se almacenará el contenido del identificador si es una variable. Si vamos a usar una estructura que funcione como memoria para el programa, su posición asignada al identificador (creo que el profesor comentó que no nos deberíamos preocupar mucho por los tamaños para las variables).
3. Tipo del identificador
4. .

3.1 Estructura de la tabla de símbolos

Estructura:

- Id: identificador de la tabla de símbolos (tipo string)
- Dirección: Donde se encuentra dentro de la memoria de variables
- Tipo: Indica el tipo de la variable.

Operaciones:

- **creaTS():TS** crea una tabla de símbolos vacía
- **creaTS(ts: TS):** crea una copia de la TS pasada como argumento
- **añadeID(ts: TS, id: String, tipo: String):** TS añade un id a la tabla de símbolos.
- **existeID(ts: TS, id: String): Boolean** Indica si existe un id en la tabla de símbolos.
- **existeID_var(ts: TS, id:String): Boolean** Indica si existe una variable en la tabla de símbolos cuyo identificador es id.
- **existeID_tip(ts: TS, id:String): Boolean** Indica si existe un tipo en la tabla de símbolos cuyo identificador es id.
- **existeID_proc(ts: TS, id:String): Boolean** Indica si existe un procedimiento en la tabla de símbolos cuyo identificador es id.
- **dameProps (ts:TS, id:String,clase):** Devuelve la lista de propiedades correspondiente al id.

3.2 Construcción de la tabla de símbolos

3.2.1 Funciones semánticas

3.2.2 Atributos semánticos

- ts: es un atributo sintetizado que corresponde a la tabla de símbolos
- tsh: atributo heredado con el cual se pasa la tabla de símbolos a partir de la cual la categoría sintáctica va a sintetizar ts.
- id: es un atributo sintetizado que guarda el nombre del identificador asociado a la declaración.
- props: es un atributo sintetizado que contiene el tipo asociado a la declaración ó expresión. ~~Sus posibles valores son: tBool, tChar, tNat, tInt y tFloat.~~
- errorDec: es un atributo sintetizado que nos informa acerca de posibles errores en la declaración de variables, tipos o procedimientos.
- errorSent: es un atributo sintetizado que nos informa acerca de posibles errores en el cuerpo del programa (sólo para la primera producción de la gramática).
- Nivel: indica el índice de anidamiento de los datos de la TS

3.2.3 Gramática de atributos

```
PROGRAMA ≡ {DECS} {&} {SENTS}
    PROGRAMA.error = DECS.errorDec OR SENTS.errorSent
    SENTS.tsh = DECS.ts
    DECS.tsph = creaTS()
    DECS.niv = 0

PROGRAMA ≡ {&} {SENTS}
    SENTS.tsph = creaTS()
    DECS.niv = 0

DECS ≡ {DECS} ; {DEC}
// Como viene en los apuntes con la TS ahora heredada ( sin contar el
tratamiento de errores)
    DECS_1.tsph=DECS_0.tsph
    DEC.tsph = DECS_1.tsp
    DECS_1.nivel = DECS_0.nivel = DEC.nivel
    DECS_0.tsp= añadeID(DECS_1.tsp, DEC.lexema, DEC.props,
        DEC.clase, DEC.nivel)
    DEC.dirh=DECS_1.dir

DECS ≡ {DEC}
    DEC.tsph = DECS.tsph
    DECS.tsp = añadeID(DECS.tsph, DEC.lexema, DEC.props,
        DEC.clase, DECS.nivel)
DECS.dir=DEC.tam
    DEC.dirh=0

DEC ≡ {DECVAR}    DECVAR.tsph = DEC.tsph
    DEC.clase = "variable"
    DEC.lexema = DECVAR.lexema
    DEC.props = DECVAR.props
    DEC.errorDec = DECVAR.error
    *En los apuntes añade aquí el nivel como parte de propiedades,
yo no lo he bajado tanto el nivel, ya que lo añadido en la producción
anterior
    DEC.tam = DECVAR.props.<tam: ¿?>
```

```

DECVAR ≡ {iden} : {TIPOIDEN}
    DECVAR.lexema = iden.lexema
    DECVAR.props = TIPOIDEN.props
    TIPOIDEN.tsph = DECVAR.tsph

TIPOIDEN ≡ {boolean}
    TIPODEN.props = <t: boolean> ++ <tam: 1>

TIPOIDEN ≡ { character }
    TIPODEN.props = <t: character > ++ <tam: 1>

TIPOIDEN ≡ { natural }
    TIPODEN.props = <t: natural > ++ <tam: 1>

TIPOIDEN ≡ { integer }
    TIPODEN.props = <t: integer > ++ <tam: 1>

TIPOIDEN ≡ { float }
    TIPODEN.props = <t: float > ++ <tam: 1>

TIPOIDEN ≡ { iden }
    TIPODEN.props = <t:ref> ++ <id: iden.lexema> ++ <tam: •
    dameProps(TIPOIDEN.tsph, iden.lexema,"variable").tam>

TIPOIDEN ≡ {record} {/{}}{CAMPOS} {/}}
    TIPOIDEN.props = <t:rec> ++ < campos: CAMPOS.props> ++ <tam:
    CAMPOS.tam>

TIPOIDEN ≡ {pointer} {TIPOIDEN}
    TIPOIDEN_0.props = <t:puntero> ++ <tbase: (<t:ref> ++
    <id:TIPOIDEN_1.lexema>)> ++ <tam: 1>

TIPOIDEN ≡ {array} {[]} {natural} {[]} {of} {TIPOIDEN}
    TIPOIDEN_0.props = <t:array> ++ <nelem: natural.valor> ++
    <tbase: TIPOIDEN_1.props> ++ <tam:
    (dameProps(TIPOIDEN.tsph, iden.lexema,"variable").tam *
    natural.valor)>

CAMPOS ≡ {CAMPOS} {;} {CAMPO}
    CAMPOS_0.props = CAMPOS_1.props++CAMPO.props
    CAMPOS_0.tam =CAMPOS_1.tam + CAMPO.tam
    CAMPO.desh = CAMPOS_1.tam

CAMPOS ≡ {CAMPO}
    CAMPOS.props = CAMPO.props
    CAMPOS.tam= CAMPO.tam
    CAMPO.desph= 0

CAMPO ≡ {iden} : {TIPOIDEN}
    CAMPO.props = <id: iden.lexema> ++ <t: _TIPOIDEN.props.t> ++
    <desp: CAMPO.desph>
    CAMPO.tam = dameProps(TIPOIDEN.tsph, iden.lexema,"variable").tam
    DEC ≡ {DECTIP}
    DEC.clase = "tipo"
    DEC.lexema = DECTIP.lexema
    DEC.props = DECTIP.props

```

```

DECTIP ≡ {tipo} {iden} {=} {TIPOIDEN}
    DECTIP.lexema = iden.lexema
    DECTIP.props = TIPOIDEN.props

DEC ≡ {DECPROC}
    DEC.clase = "procedimiento"
    DEC.lexema = DECPROC.lexema
    DEC.props = DECPROC.props
    DECPROC.tsph=DEC.tsph
    DECPROC.nivel = DEC.nivel
    DEC.errorDec = DECPROC.error

DECPROC ≡ {proc} {iden} {DPARAMS} {PBLOQUE}
    DECPROC.lexema = iden.lexema
    DECPROC.clase = "procedimiento"
    DECPROC.props = DPARAMS.<t:proc>++<params:DPARAMS.params>
    DPARAM.tsph = creaTS(DECPROC.tsph)
    PBLOQUE.tsph = añadeID(DPARAMS.tsph, DECPROC.lexema,
    DECPROC.props, DECPROC.nivel+1)
    DPARAMS.nivelh=PBLOQUE.nivelh=DECPROC.nivel+1

PBLOQUE ≡ {forward}

PBLOQUE ≡ {{{DECS}{&}{SENTS}{}}}
    DECS.tsph=PBLOQUE.tsph
    DECS.nivel=PBLOQUE.nivel
    SENT.tsh= DECS.ts
    PBLOQUE.ts=DECS.ts

PBLOQUE ≡ {{{&}{SENTS}{}}}
    SENTS.TS= PBLOQUE.tsph

DPARAMS ≡ {( ) {LISTAPARAMS} {}}
    DPARAMS.param=LISTAPARAMS.param
    DPARAMS.ts = LISTAPARAMS.ts
    LISTAPARAMS.tsph= DPARAMS.tsph
    LISTAPARAMS.nivelh = DPARAMS.nivelh

DPARAMS ≡ {lambda}
    DPARAMS.props = {}
    DPARAMS.ts= FPARAMS.tsph

LISTAPARAMS ≡ {LISTAPARAMS} {,} {PARAM}
    LISTAPARAMS_0.param = LISTAPARAMS_1.param ++ PARAM.param
    LISTAPARAMS_0.ts = añadeID(LISTAPARAMS_1.ts,
        PARAM.lexema,PARAM.props, PARAM.clase,
        LISTAPARAMS_0.nivelh)
    LISTAPARAMS.tsph = LISTAPARAMS.tsph

LISTAPARAMS ≡ {PARAM}
    LISTAPARAMS.param = PARAM.param
    LISTAPARAMS.ts = añadeID(LISTAPARAMS.tsph, PARAM.lexema,
        Para.props, PARAM,clase, LISTAPARAMS.nivelh)

PARAM ≡ {var} {iden} {:} {TIPOIDEN}
    PARAM.clase = "p_variable"
    PARAM.lexema = iden.lexema

```

```
PARAM.props = <t: TIPOIDEN.props>
PARAM.param = <modo : variable> <t: TIPOIDEN.props.t>

PARAM ≡ {iden} { : } {TIPOIDEN}
PARAM.id= iden.lexema
PARAM.clase = "variable"
PARAM.props = <t: TIPOIDEN.props.t>
PARAM.param = <modo:valor, t: TIPOIDEN.props>
```

4 Especificación de las restricciones contextuales

4.1 Descripción informal de las restricciones contextuales

1. Los identificadores utilizados en las instrucciones de la sección de código (tanto a la izquierda como a la derecha de la asignación) deben haber sido declarados previamente en la sección de declaraciones.
2. Un mismo identificador sólo puede aparecer una vez en la sección de declaraciones.
3. Los operadores de las expresiones deben ser del mismo tipo, o de tipos compatibles que permitan realizar la operación. Se aplica la misma restricción para las asignaciones.
4. No se permitirá la declaración de identificadores con el mismo nombre que alguna palabra reservada.
5. Las operaciones de desplazamiento sólo se pueden dar entre naturales.

4.2 Funciones semánticas

- **existeID(ts: TS, id: String): Boolean**
- **dameTipo(tipoId:String, tipoId:String, op:String): String** dados dos operandos y una operación te devuelve el tipo de la operación resultante, en caso de que no se pueda hacer la operación sobre los operadores concretos devuelve el tipo tError
- **dameTipo(tipoId:String, op:String): String** dados un operando y una operación te devuelve el tipo de la operación resultante, en caso de que no se pueda hacer la operación sobre los operadores concretos devuelve el tipo tError
- **esCompatibleAsig?(tipoId: String, tipoEXP: String): Boolean** Nos indica si el tipo del identificador de una asignación es compatible al de la expresión asociada.
- **dameTipoTS(tsh: TS, id: String): String** Devuelve el tipo asociado al identificador que se pasa por parámetro, recogiendo la información de la tabla de símbolos.

4.3 Atributos semánticos

Los atributos semánticos son los mismos que en el punto 3.2.2 a no ser que se indique lo contrario en las siguientes descripciones:

- Ts:
- tsh
- id
- tipo: ahora incluiremos el tipo erróneo para expresiones (tError)
- errorDec (Sólo para la primera producción de la gramática)
- errorSent
- op: operación asociada al operador (atributo sintetizado)

4.4 Gramática de atributos

```
PROGRAMA ≡ {DECS} {&} {SENTS}  
PROGRAMA.error = DECS.errorDec OR SENTS.errorSent  
SENTS.tsh = DECS.ts
```

```

    DECS.tsph = creaTS()
    DECS.niv = 0

PROGRAMA ≡ {&} {SENTS}
    PROGRAMA.error = SENTS.errorSent
    SENTS.tsph = creaTS()
    DECS.niv = 0

DECS ≡ {DECS} ; {DEC}
// Como viene en los apuntes con la TS ahora heredada ( sin contar el
tratamiento de errores)
    DECS_1.tsph=DECS_0.tsph
    DEC.tsph = DECS_1.tsph
    DECS_1.nivel = DECS_0.nivel = DEC.nivel
    DECS_0.errorDec = DEC.errorDec or DECS_1.errorDec
    si not DECS_0.errorDec entonces
        DECS_0.tsph= añadeID(DECS_1.tsph, DEC.lexema, DEC.props,
            DEC.clase, DEC.nivel)
    fin si

DECS ≡ {DEC}
    DECS.errorDec = DEC.errorDec
    DEC.tsph = DECS.tsph
    si DECS.errorDec entonces
        DECS.tsph = añadeID(DECS.tsph, DEC.lexema, DEC.props,
            DEC.clase, DECS.nivel)
    fin si

DEC ≡ {DECVAR}    DECVAR.tsph = DEC.tsph
    DEC.clase = "variable"
    DEC.lexema = DECVAR.lexema
    DEC.props = DECVAR.props
    DEC.errorDec = DECVAR.error

DECVAR ≡ {iden} : {TIPOIDEN}
    DECVAR.lexema = iden.lexema
    DECVAR.props = TIPOIDEN.props
    TIPOIDEN.tsph = DECVAR.tsph
    DECVAR.error = existeID(DECVAR.tsph, iden.lexema) or
        TIPOIDEN.error

TIPOIDEN ≡ {boolean}
    TIPOIDEN.error = FALSE
    TIPOIDEN.props = <t: boolean>

TIPOIDEN ≡ { caracter }
    TIPOIDEN.props = <t: caracter >
    TIPOIDEN.error = FALSE

TIPOIDEN ≡ { natural }
    TIPOIDEN.props = <t: natural >
    TIPOIDEN.error = FALSE

TIPOIDEN ≡ { integer }
    TIPOIDEN.props = <t: integer >
    TIPOIDEN.error = FALSE

TIPOIDEN ≡ { float }
    TIPOIDEN.props = <t: float >
    TIPOIDEN.error = FALSE

```

```

TIPOIDEN ≡ { iden }
    TIPOIDEN.props = <t:ref> ++ <id: iden.lexema>
    TIPOIDEN.error = not existeTipo(TIPOIDEN.tsph, iden.lexema)

TIPOIDEN ≡ {record} {/{ } {CAMPOS} {/}}
    TIPOIDEN.props = <t:rec> ++ < campos: CAMPOS.props>
    TIPOIDEN.error = CAMPOS.error

TIPOIDEN ≡ {pointer} {TIPOIDEN}
    TIPOIDEN_0.props = <t:puntero> ++ <tbase: (<t:ref> ++
        <id:TIPOIDEN_1.lexema>)>

TIPOIDEN ≡ {array} {[ ] {natural} [ ]} {of} {TIPOIDEN}
    TIPOIDEN_0.props = <t:array> ++ <nelem: natural.valor> ++
        <tbase: TIPOIDEN_1.props>
    TIPOIDEN_0.error = TIPOIDEN_1.error

CAMPOS ≡ {CAMPOS} {;} {CAMPO}
    CAMPO.tsph = CAMPOS.tsph
    CAMPOS_0.props = CAMPOS_1.props++CAMPO.props
    CAMPOS_0.error = CAMPOS_1.error or CAMPO.error

CAMPOS ≡ {CAMPO}
    CAMPO.tsph = CAMPOS.tsph
    CAMPOS.props = CAMPO.props
    CAMPOS.error = CAMPO.error

CAMPO ≡ {iden} : {TIPOIDEN}
    CAMPO.props = <id: iden.lexema> ++ <t: _TIPOIDEN.props.t>
    CAMPO.error = existeID(CAMPO.tsph, iden.lexema) or TIPOIDEN.error

DEC ≡ {DECTIP}
    DECTIP.tsph = DEC.tsph
    DEC.clase = "tipo"
    DEC.lexema = DECTIP.lexema
    DEC.props = DECTIP.props
    DEC.errorDec = DECTIP.error

DECTIP ≡ {tipo} {iden} {=} {TIPOIDEN}
    DECTIP.lexema = iden.lexema
    TIPOIDEN.tsph = DECTIP.tsph
    DECTIP.props = TIPOIDEN.props
    DECTIP.error = existeID(DECTIP.tsph, iden.lexema) or
        TIPOIDEN.error

DEC ≡ {DECPROC}
    DEC.clase = "procedimiento"
    DEC.lexema = DECPROC.lexema
    DEC.props = DECPROC.props
    DECPROC.tsph=DEC.tsph
    DECPROC.nivel = DEC.nivel
    DEC.errorDec = DECPROC.error

DECPROC ≡ {proc} {iden} {DPARAMS} {PBLOQUE}
    DECPROC.lexema = iden.lexema
    DECPROC.clase = "procedimiento"
    DECPROC.props = DPARAMS.<t:proc>++<params:DPARAMS.params>
    DPARAM.tsph = creaTS(DECPROC.tsph)
    PBLOQUE.tsph = añadeID(DPARAMS.tsph, DECPROC.lexema,
        DECPROC.props, DECPROC.nivel+1)

```

```

DPARAMS.nivelh=PBLOQUE.nivelh=DECPROC.nivel+1
DECPROC.error = existeId(iden) or DPARAMS.error or PBLOQUE.error

PBLOQUE ≡ {{}}{DECS}{&}{SENTS}{{}}
  DECS.tsph=PBLOQUE.tsph
  DECS.nivel=PBLOQUE.nivel
  SENT.tsh= DECS.tsp
  PBLOQUE.ts=DECS.tsp
  PBLOQUE.error = SENT.errorSent or DECS.errorDec

PBLOQUE ≡ {{}} {&} {SENTS}{{}}
  SENTS.ts= PBLOQUE.tsph
  PBLQUE.error = SENT.errorSent

DPARAMS ≡ {()} {LISTAPARAMS} {}
  DPARAMS.param=LISTAPARAMS.param
  DPARAMS.tsp = LISTAPARAMS.tsp
  LISTAPARAMS.tsph= DPARAMS.tsph
  LISTAPARAMS.nivelh = DPARAMS.nivelh
  DPARAMS.error = LISTAPARAMS.error

DPARAMS ≡ {lambda}
  DPARAMS.props = {}
  DPARAMS.tsp= DPARAMS.tsph
  DPARAMS.error = FALSE

LISTAPARAMS ≡ {LISTAPARAMS} {,} {PARAM}
  LISTAPARAMS_0.param = LISTAPARAMS_1.param ++ PARAM.param
  LISTAPARAMS_0.error = LISTAPARAMS_1.error or PARAM.error
  si not LISTAPARAMS_0.error entonces
    LISTAPARAMS_0.tsp = añadeID(LISTAPARAMS_1.tsp,
      PARAM.lexema,PARAM.props, PARAM.clase,
      LISTAPARAMS_0.nivelh)
  fin si
  LISTAPARAMS.tsph = LISTAPARAMS.tsph

LISTAPARAMS ≡ {PARAM}
  LISTAPARAMS.param = PARAM.param
  LISTAPARAMS.error = PARAM.error
  si not LISTAPARAMS.error entonces
    LISTAPARAMS.ts = añadeID(LISTAPARAMS.tsph, PARAM.lexema,
      PARA.props, PARAM,clase, LISTAPARAMS.nivelh)
  fin si

PARAM ≡ {var} {iden} {:} {TIPOIDEN}
  TIPOIDEN.tsph = PARAM.tsph
  PARAM.clase = "p_variable"
  PARAM.id= iden.lexema
  PARAM.props = <t: TIPOIDEN.props>
  PARAM.param = <modo : variable> <t: TIPOIDEN.props.t>
  PARAM.error = existeId(PARAN.tsph,iden.lexema) or TIPOIDEN.error

PARAM ≡ {iden} {:} {TIPOIDEN}
  TIPOIDEN.tsph = PARAM.tsph
  PARAM.id= iden.lexema
  PARAM.clase = "variable"
  PARAM.props = <t: TIPOIDEN.props.t>
  PARAM.param = <modo:valor, t: TIPOIDEN.props>
  PARAM.error = existeId(PARAN.tsph,iden.lexema) or TIPOIDEN.error

```



```

SENTS ::= SENTS ; SENT
    SENTS_1.tsh = SENT.tsh = SENTS_0.tsh
    SENTS_0.errorSent = SENTS_1.errorSent OR SENT.errorSent

SENTS ::= SENT
    SENT.tsh = SENTS.tsh
    SENTS.errorSent = SENT.errorSent

SENT ::= SWRITE
    SWRITE.tsh = SENT.tsh
    SENT.errorSent = SWRITE.errorSent

SENT ::= SREAD
    SREAD.tsh = SENT.tsh
    SENT.errorSent = SREAD.errorSent

SENT ::= SBLOQUE
    SBLOQUE.tsh = SENT.tsh
    SENT.errorSent = SBLOQUE.errorSent

SENT ::= SIF
    SIF.tsh = SENT.tsh
    SENT.errorSent = SIF.errorSent

SENT ::= SWHILE
    SWHILE.tsh = SENT.tsh
    SENT.errorSent = SWHILE.errorSent

SENT ::= SFOR
    SFOR.tsh = SENT.tsh
    SENT.errorSent = SFOR.errorSent

SWRITE ::= out ( EXP )
    EXP.tsh = SWRITE.tsh
    SWRITE.errorSent = (EXP.tipo = tError)

SREAD ::= in ( iden )    SREAD.errorSent = (NOT existeVar(SREAD.tsh,
iden.lexema))

```

(Faltaría añadir que el tipo de la variable ‘iden’, que se encuentra en la tabla de símbolos, debe ser compatible con el tipo de la cadena introducida por teclado)

```

SASIG ::= iden := EXP
    EXP.tsh = SASIG.tsh
    SASIG.errorSent = (EXP.tipo = tError) OR
        (NOT existeID(SASIG.tsh, iden.lexema)) OR
        (NOT esCompatibleAsig?(dameTipoTS(SASIG.tsh,
            iden.lexema), EXP.tipo))

SBLOQUE ::= { SENTS }
    SENTS.tsh = SBLOQUE.tsh
    SBLOQUE.errorSent = SENTS.errorSent

SIF ::= if EXP then SENT PELSE
    PELSE.tsh = SENT.tsh = EXP.tsh = SIF.tsh
    SIF.errorSent = (EXP.tipo <> tBool) OR SENT.errorSent OR
        PELSE.errorSent

```

```

PELSE ::= else SENT
    SENT.tsh = PELSE.tsh
    PELSE.errorSent = SENT.errorSent

PELSE ::= lambdaλ
    PELSE.errorSent = FALSE

SWHILE ::= while EXP do SENT
    SENT.tsh = EXP.tsh = SWHILE.tsh
    SWHILE.errorSent = (EXP.tipo <> tBool) OR SENT.errorSent

SFOR ::= for iden = EXP to EXP do SENT
    SENT.tsh = EXP_1.tsh = EXP_0.tsh = SFOR.tsh
    SFOR.errorSent = SENT.errorSent
        OR (NOT existeID(SFOR.tsh, iden.lexema))
        OR (NOT ((EXP_0.tipo = tNat) OR (EXP_0.tipo = tInt)))
        OR (NOT ((EXP_1.tipo = tNat) OR (EXP_1.tipo = tInt)))
        OR (NOT esCompatibleAsig?(dameTipoTS(SFOR.tsh,
            iden.lexema), EXP_0.tipo))
        OR (NOT esCompatibleAsig?(dameTipoTS(SFOR.tsh,
            iden.lexema), EXP_1.tipo))

SNEW ≡ {new} {mem}
    MEM.etqh = SNEW.etqh
    Si not Mem.tipo = puntero entonces
        SNEW.errorSent = TRUE

SDEL ≡ {dispose} {mem}
    MEM.etqh = SDEL.etqh
    Si not Mem.tipo = puntero entonces
        SDEL.errorSent = TRUE

SENT ::= {iden} {RSENT}
    RSENT.tsh = MEM.tsh
    RSENT.tipoh = dameTipo(SENT.tsh, iden.lexema)
    RSENT.tipo = RMEM.tipo
    SENT.errorSent = RSENT.errorSent

RSENT ::= {RMEM} {:=} {EXP}
    RMEM.tipoh = RSENT.tipoh
    RMEM.tsh = RSENT.tsh
    RSENT.errorSent = errorTipos(RMEM.tipo, EXP.tipo)
//      Or RMEM.tipo = tError or EXP.tipo = tError
// NO hace falta porque errorTipos te da error si alguno es tError

RSENT ::= {PPARAMS}
    PPARAMS.tsh = RSENT.tsh
    RSENT.errorSent = PPARAMS.error

PPARAMS ≡ {()}{LPARAMS} {}
    LPARAMS.procName = PPARAMS.procName
    LPARAMS.tsh = PPARAMS.tsh
    LPARAMS.nParamH = 0
    LPARAMS.dirH = 0
    EXP.tsh = LPARAMS_0.tsh
    PPARAMS.error = LPARAMS.error or
        damePropTS(PPARAMS.tsh, PPARAMS.procName).nParams <>
        LPARAMS.nParams

PPARAMS ≡ λ

```

```

PPARAMS.error = FALSE

LPARAMS ≡ {LPARAMS} {,} {EXP}
  LPARAMS_1.nParamH = LPARAMS_0.nParamH + 1
  LPARAMS_1.procName = LPARAMS_0.procName
  LPARAMS_1.tsh = LPARAMS.tsh
  EXP.tsh = LPARAMS_0.tsh
  LPARAMS_0.error = comparaTipos(LPARAMS_0.tsh, PPARAMS_0.procName,
                                LPARAMS_0.nParamH) or
                                LPARAMS_1.error or EXP.error
  LPARAM_0.nParam = LPARAM_1.nParam

LPARAMS ≡ {EXP}
  EXP.tsh = LPARAMS.tsh
  LPARAMS_0.error = comparaTipos(LPARAMS.tsh, PPARAMS.procName,
                                LPARAMSLPARAMS.nParamH) or
                                LPARAMS_1.error or EXP.error
  LPARAM.nParam = LPARAM.nParamH

EXP ::= EXP1 OP0 EXP1
  EXP1_0.tsh = EXP1_1.tsh = EXP.tsh
  EXP.tipo = dameTipo(EXP1_0.tipo, EXP1_1.tipo, OP0.op)

EXP ::= EXP1
  EXP1.tsh = EXP.tsh
  EXP.tipo = EXP1.tipo

EXP1 ::= EXP1 OP1 EXP2
  EXP1_1.tsh = EXP2.tsh = EXP1_0.tsh
  EXP1_0.tipo = dameTipo(EXP1_1.tipo, EXP2.tipo, OP1.op)

EXP1 ::= EXP2
  EXP2.tsh = EXP1.tsh
  EXP1.tipo = EXP2.tipo

EXP2 ::= EXP2 OP2 EXP3
  EXP2_1.tsh = EXP3.tsh = EXP2_0.tsh
  EXP2_0.tipo = dameTipo(EXP2_1.tipo, EXP3.tipo, OP2.op)

EXP2 ::= EXP3
  EXP3.tsh = EXP2.tsh
  EXP2.tipo = EXP3.tipo

EXP3 ::= EXP4 OP3 EXP3
  EXP3_1.tsh = EXP4.tsh = EXP3_0.tsh
  EXP3_0.tipo = dameTipo(EXP4.tipo, EXP3_1.tipo, OP3.op)

EXP3 ::= EXP4
  EXP4.tsh = EXP3.tsh
  EXP3.tipo = EXP4.tipo

EXP4 ::= OP4_1 TERM
  TERM.tsh = EXP4.tsh
  EXP4.tipo = dameTipo(TERM.tipo, OP4_1.op)

EXP4 ::= \ | TERM \ |
  (La barra '\\' es para escapar el símbolo '|' del valor absoluto)
  TERM.tsh = EXP4.tsh
  EXP4.tipo = dameTipo(TERM.tipo, "\"")

```

```

EXP4 ::= TERM
      TERM.tsh = EXP4.tsh
      EXP4.tipo = TERM.tipo

TERM ::= MEM
      TERM.tipo = MEM.tipo

TERM ::= boolean
      TERM.tipo = tBool

TERM ::= cadCaracteres
      TERM.tipo = tChar

TERM ::= natural
      TERM.tipo = tNat

TERM ::= entero
      TERM.tipo = tInt

TERM ::= real
      TERM.tipo = tFloat

TERM ::= ( EXP )
      EXP.tsh = TERM.tsh
      TERM.tipo = EXP.tipo

MEM = {iden} {RMEM}
      RMEM.tsh = MEM.tsh
      RMEM.tipoh = dameTipo(MEM.tsh, iden.lexema)
      MEM.tipo = RMEM.tipo

RMEM = λ
      RMEM.tipo = RMEM.tipoh

RMEM = {->} RMEM
      RMEM_1.tsh = RMEM_0.tsh
      RMEM_1.tipoh = tipoBase(RMEM_0.tsh, RMEM_0.tipoh.tbases)
      RMEM_0.tipo = RMEM_1.tipo

RMEM = {[ ] {EXP} [ ]} RMEM
      EXP.tsh = RMEM.tsh
      RMEM_1.tsh = RMEM_0.tsh
      Si RMEM_0.tipo = tArray and EXP.tipo = tEntero entonces
          RMEM_1.tipoh = tipoBase(RMEM_0.tsh, RMEM_0.tipoh.tbases)
          RMEM_0.tipo = RMEM_1.tipo
      Si no
          RMEM.tipo = tError

RMEM = { . } {iden} RMEM
      RMEM_1.tsh = RMEM_0.tsh
      RMEM_1.tipoh = tipoCampo(RMEM_0.tipoh, iden.lexema.RMEM_0.tsh)
      RMEM_0.tipo = RMEM_1.tipo

OP0 ::= < | > | <= | >= | = | /=

OP1 ::= +
      OP1.op = suma

OP1 ::= -
      OP1.op = resta

OP1 ::= or
      OP1.op = oLogica

```

```
OP2 ::= *
      OP2.op = multiplicacion
OP2 ::= /
      OP2.op = division
      OP2 ::= %
      OP2.op = resto
      OP2 ::= and
      OP2.op = yLogica
      OP3 ::= <<
      OP3.op = despIzq
      OP3 ::= >>
      OP3.op = despDer
      OP4_1 ::= not
      OP4_1.op = negLogica
      OP4_1 ::= -unario
      OP4_1.op = negArit
      OP4_1 ::= (nat)
      OP4_1.op = castNat
      OP4_1 ::= (int)
      OP4_1.op = castInt
      OP4_1 ::= (char)
      OP4_1.op = castChar
      OP4_1 ::= (float)
      OP4_1.op = castFloat
```

5 Especificación de la traducción

5.1 Lenguaje objeto

- apila(dato). Apila un dato en la pila. Este dato puede ser de cualquier tipo soportado.
- Desapila: Desapila un dato de la pila perdiendolo
- apilaDir(direccion). Apila un dato de memoria a la pila
- desapilaDir(direccion) Desapila un dato de la pila a memoria. Las operaciones son distintas dependiendo del dato primitivo (desapilaDirNat, desapilaDirBoolean...)
- mayor: compara los dos ultimos datos de la pila y guarda un booleano con el resultado de la comparación. Esta operación permite comparar enteros reales y naturales entre si. También caracteres entre si
- menor: compara los dos ultimos datos de la pila y guarda un booleano con el resultado de la comparación. Esta operación permite comparar enteros reales y naturales entre si. También caracteres entre si
- mayorIgual: compara los dos ultimos datos de la pila y guarda un booleano con el resultado de la comparación. Esta operación permite comparar enteros reales y naturales entre si. También caracteres entre si
- menorIgual: compara los dos ultimos datos de la pila y guarda un booleano con el resultado de la comparación- Esta operación permite comparar enteros reales y naturales entre si. También caracteres entre si
- disntinto: compara los dos ultimos datos de la pila y guarda un booleano con el resultado de la comparación. Esta operación permite comparar enteros reales y naturales entre si. También permite comprar caracteres entre si, y por último booleanos
- suma: Suma dos operandos, permite sumar reales, enteros y naturales entre si dando como resultado el tipo menos restrictivo de los que participen (Real, Entero y Natural en este orden), También permite sumar caracteres entre ellos
- resta: Resta dos operandos, permite restar reales, enteros y naturales entre si dando como resultado el tipo menos restrictivo de los que participen (Real, Entero y Natural en este orden) excepto la resta de dos naturales cullo resultado sea negativo, en ese caso el tipo será entero. También permite restar caracteres entre ellos
- producto: multiplica dos operandos, permite multiplicar reales, enteros y naturales entre si dando como resultado el tipo menos restrictivo de los que participen (Real, Entero y Natural en este orden).
- division: divide dos operandos, permite dividir reales, enteros y naturales entre si dando como resultado el tipo menos restrictivo de los que participen (Real, Entero y Natural en este orden).
- modulo: calcula el modulo de dos operandos, permite calcular el modulo entre enteros y naturales entre si dando como resultado el tipo menos restrictivo de los que participen (Real, Entero y Natural en este orden).
- yLogica: Realiza la y lógica entre dos booleanos
- oLogica: Realiza la o lógica entre dos booleanos
- negLogica: Realiza la negacion lógica entre dos operandos
- negArit: Cambia de signo un operando, los operando pueden ser reales o enteros. Esta operación esta en el repertorio de operaciones de nuestra MV pero la gramática no utiliza

- valorAbs: Devuelve el valor absoluto de un valor entero real o natural (aunque natural no tiene mucho sentido)
- despIzq: Desplaza n bytes a la izquierda
- despDer: Desplaza n bytes a la derecha
- castNat: Realiza el cast a natural
- castInt: Realiza el cast a entero
- castChar: Realiza el cast a Char
- castFloat: realiza el cast a Float
- leer: lee una entrada y la almacena en un buffer en la pila
- escribir: escribe una variable por pantalla
- apilaInd: Desapila un dato de memoria (a), y apila en la pila el valor contenido en la memoria en la posición a.
- desapilaInd: Desapila dos datos de la pila a memoria (“x”, correspondiente a la cima y “z” correspondiente a la sublima). La ejecución de esta instrucción tiene por resultado insertar en la memoria en la posición “z” el dato “x”. Las operaciones son distintas dependiendo del dato primitivo (desapilaDirNat, desapilaDirBoolean...)
- copia: Duplica el contenido de la cima de la pila.
- Mueve(tamaño): Desapila dos datos de la pila, la cima se corresponde con “origen” y la sublima con “destino”, esta instrucción copia “tamaño” celdas consecutivas de origen a destino.
- New(tamaño): crea “tamaño” numero de celdas consecutivas en la parte de la memoria correspondiente a la memoria dinámica, y deja en la cima de la pila la dirección de origen.
- Del(tamaño): Toma la cima de la pila, y la interpreta como la dirección a partir de la cual se eliminarán “tamaño” celdas consecutivas de la memoria correspondiente a la memoria dinámica)
- Ir_v(destino): Realiza un salto a la dirección indicada en destino cuando la cima de la pila sea igual a cierto.
- Ir_f(destino): Realiza un salto a la dirección indicada en destino cuando la cima de la pila sea igual a falso.
- Ir_a(destino): Realiza un salto incondicional a la instrucción indicada como parámetro.
- Ir_ind: Realiza un salto incondicional a la dirección que hay en la cima de la pila.
- stop: para la ejecución del programa

La maquina consta de 3 partes,

- Memoria: Un vector de objetos, dónde se puede definir su tamaño, la memoria tiene una parte estática (desde la posición 0 en adelante) y una parte dinámica (empieza en la posición final y va creciendo hacia el principio). Las operaciones que trabajan con al memoria son las instrucciones correspondientes a apilar, desafilas, new y del.
- Pila: Donde se van almacenando los datos para operar. Las operaciones quitando los apilas y desafilas trabajan con un numero determinandos de operadores que deben estar en la pila. Y apilan el resultado de las operaciones de nuevo en la pila.
- Codigo: Un vector de instrucciones, son secuenciales es decir, que en caso de que no haya saltos (en esta version de la maquina virtual no esta implementado) las operaciones se efectuaran crecientemente.

Tipos con los que trabaja la maquina virtual, Todos los tipos si no se dice lo contrario son serializables:

- MyBoolean: tipo correspondiente a un booleano
- MyNatural: Tipo natural
- MyInteger: Tipo entero
- MyFloat: Real
- MyChar: tipo carácter
- MyExecutionError: En caso de que una instrucción tiene la opción de lanzar este tipo con una descripción del error mas concreta para que pueda ser analizado. Este tipo no puede ser serializado. Esto se debe a que este tipo no se almacena nunca desde el traductor si no que son tipos que solo maneja la maquina virtual
- MyBuffer: Tipo que almacena un Buffer, útil para leer una cadena antes de transformarla a un valor dado. Este tipo tampoco es serializado ya que al igual que el anterior se trabaja con el directamente en la máquina virtual.

Control de errores:

El control de errores se realiza mediante el tipo `MyExecutionError`. En caso de error la ejecución fallará. (retornando un `false`) y se imprime el código de error en caso de existir)

Manejo de la memoria, pila y contador por parte de las instrucciones

Las instrucciones generan la interacción entre la pila y la memoria, de tal forma que las instrucciones van modificando la pila según sea la instrucción a ejecutar, y las instrucciones que acceden a memoria tanto para su modificación, consulta o eliminación de información, se hace mediante llamadas a procedimientos de la clase memoria.

Las instrucciones manejan directamente sobre la pila y memoria, e sitúan el contador de programa en la posición de la siguiente instrucción. En esta versión, al no soportar los saltos, siempre incrementara en uno la instrucción

La memoria:

Para ranurar la memoria se ha hecho que las declaraciones de memoria guarden datos basura en la memoria que indican cual es el tipo. Esto le da flexibilidad para una futura versión ya que en caso de destruir esa posición y poner un nuevo tipo hace que pueda reutilizarse esa posición de memoria dinámicamente.

5.2 Funciones semánticas

Modificaciones en las funciones de la tabla de símbolos:

- **añadeID(ts: TS, id: String, tipo: String, ps: Propiedades): TS** añade un id a la tabla de símbolos junto con su registro de propiedades.
- **damePropiedadesTS(ts: TS, id: String): Propiedades** Devuelve de la tabla de símbolos, el registro de propiedades asociado al identificador que se pasa por parámetro. Por ahora nos centraremos en la dirección asignada al identificador como propiedad ('dirProp').

Las funciones que usamos para generar código de forma automática son

- **Prologo (nivel, tamlocales):** Genera el código del prologo, el código de este es:

```
fun prologo(nivel,tamlocales)
  devuelve apila-dir(0)      ||
          apila(2)           ||
          suma               ||
          apila-dir(1+nivel) ||
          desapila-ind       ||
          apila-dir(0)       ||
          apila(3)           ||
          suma               ||
          desapila-dir(1+nivel) ||
          apila-dir(0)       ||
          apila(tamlocales+2) ||
          suma               ||
          desapila-dir(0)
ffun
```

- **epilogo(nivel):** Genera el código del epilogo de una función:

```
fun epilogo(nivel)
  devuelve apila-dir(1+nivel)
          apila(2)
          resta
          apila-ind
          apila-dir(1+nivel)
          apila(3)
          resta
          copia
          desapila-dir(0)
          apila(2)
          suma
          apila-ind
          desapila-dir(1+nivel)
```

- **Apila ret:** Esta función almacena la dirección de retorno en la posición del procedimiento para ello

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

5.3 ffunAtributos semánticos

- op: ya lo conocemos de la gramática de atributos anterior, pero ahora se amplía su uso para poder añadir las todas las operaciones del lenguaje de partida al código de la máquina a pila.
- dir: atributo sintetizado que va indicando cual es la siguiente dirección de memoria a asignar para los nuevos identificadores que se incorporen en la tabla de símbolos.
- cod: atributo sintetizado que va almacenando el código de la máquina a pila según se va realizando la traducción.

- **etq**: atributo que representa una posición de código que va pasando entre las diferentes producciones.
- **longProlog**: constante con el tamaño del prólogo, en nuestro caso este valor es 13
- **longEpilogo**: constante con el tamaño del epilogo, en nuestro caso este valor es 13
- **longApilaRet**: constante con el tamaño del epilogo, en nuestro caso este valor es 5
-

5.4 Gramática de atributos

Modificaciones en relación al cuerpo de los programas, especialmente relacionadas al nuevo atributo sintetizado 'cod'. También se dan las actualizaciones del atributo sintetizado y heredado 'etq', los cambios añadidos están resaltados en cursiva:

```
PROGRAMA ≡ {DECS} {&} {SENTS}
    DECS.pendH = creaPendientes()
    DECS.tsph = creaTS()
    DECS.niv = 0
    DECS.etqh = 0
    DECS.callPendH = creaCallPends()
    SENTS.callPendH = DECS.callPendH    SENTS.etqh = DECS.etq
    SENTS.tsh = DECS.ts
    PROGRAMA.error = DECS.errorDec OR SENTS.errorSent OR
        DECS.pend.size <> 0 or SENTS.callPend.size <> 0
    PROGRAMA.cod = DECS.cod || SENTS.cod || stop

PROGRAMA ≡ {&} {SENTS}
    SENTS.tsph = creaTS()
    SENTS.callPendH = creaCallPends()
    PROGRAMA.error = SENTS.errorSent or SENTS.callPend.size <> 0
    PROGRAMA.cod = SENTS.cod || stop

DECS ≡ {DECS} ; {DEC}
// Como viene en los apuntes con la TS ahora heredada ( sin contar el
tratamiento de errores)
    DECS_1.callPendH = DECS_0.callPendH
    DEC.callPend = DECS_1.callPendH
    DECS.callPend = DEC.callPend
    DECS_1.pendH = DECS_0.pendH
    DECS_1.etqh = DECS_0.etq
    DEC.etqh = DECS_1.etq
    DECS_1.dirH = DECS_0.dirH
    DEC.dirH = DECS_1.dir
    DECS_1.tsph = DECS_0.tsph
    DEC.tsph = DECS_1.tsph
    DECS_1.nivel = DECS_0.nivel = DEC.nivel
    DECS_0.errorDec = DEC.errorDec or DECS_1.errorDec
    DEC.pendH = DECS_1.pend
    si not DECS_0.errorDec and DEC.clase = procedimiento entonces
        si not DEC.forward and entonces
            si existeID(DECS_1.tsp, DEC.lexema) and
                not (pendiente(DECS_1.pend, DEC.lexema) or
                    not tipoPendiente(DECS_1.pend, DEC.lexema)=proc)
            entonces
                DECS_0.errorSent = TRUE
            Si no si pendiente(DECS_0.pendH, DEC.lexema)
                DECS_0.pend = eliminaPendiente(DECS_1.pend,
                    DEC.lexema)
                Decs_0.errorSent = actualizaID(DECS_1.tsp,
```

```

DEC.lexema, DEC.props, DEC.clase,
DEC.nivel, DEC.etqH)

Si no
    DECS_0.pend = DECS_1.pend
fin si
si no si existeID(DECS_1.tsp, DEC.lexema) entonces
    DECS_0.errorSent = TRUE
Si no // no existe ID y es forward
    añadeID(DECS_1.tsp, DEC.lexema, DEC.props, DEC.clase,
            DEC.nivel, DEC.etqH)
    DECS_0.pend = añadePendientes(DECS_1.pend, DEC.lexema,
    proc)
fin si
si not DECS_0.errorDec // es un tipo o una var
entonces
    DECS_0.tsp= añadeID(DECS_1.tsp, DEC.lexema, DEC.props,
    DEC.clase, DEC.nivel, DEC.dirH)
    Si DEC.clase = "tipo" and existePendientes(DEC.lexema)
    Entonces
        DECS_0.pend=eliminaPendientes(DECS_1.pend, DEC.lexema)
    Si no
        DECS_0.pend = DECS_1.pend
Fin si
DECS.etq = DEC.etq
DECS_0.cod = DECS_1.cod || DEC.cod
DECS_0.dir = DEC.dir

DECS ≡ {DEC}
DEC.callPendH = DECS.callPendH
DECS.callPend = DEC.callPend
DEC.etqh = DECS.etqh
DEC.dirH = DECS.dirH
DECS.errorDec = DEC.errorDec
DEC.tsph = DECS.tsph
si not DECS_0.errorDec and DEC.clase = procedimiento entonces
    si not DEC.forward and entonces
        si existeID(DECS.tsph, DEC.lexema) and
            not (pendiente(DECS.pendH, DEC.lexema) or
            not tipoPendiente(DECS.pendH, DEC.lexema)=proc)
        entonces
            DECS.errorSent = TRUE
        Si no si pendiente(DECS.pendH, DEC.lexema)
            DECS.pend = eliminaPendiente(DECS.pendH,
            DEC.lexema)
            Decs.errorSent = actualizaID(DECS.tsph,
            DEC.lexema, DEC.props, DEC.clase,
            DEC.nivel, DEC.etqH)
        Si no
            DECS.pend = DECS.pendH
        fin si
    si no si existeID(DECS.tsph, DEC.lexema) entonces
        DECS.errorSent = TRUE
    Si no // no existe ID y es forward
        añadeID(DECS_1.tsp, DEC.lexema, DEC.props, DEC.clase,
            DEC.nivel, DEC.etqH)
        DECS.pend = añadePendientes(DECS.pendH,
            DEC.lexema, proc)
    fin si
si not DECS.errorDec // es un tipo o una var
entonces
    DECS.tsph= añadeID(RDECS_0.tsph, DEC.lexema, DEC.props,

```

```

        DEC.clase, DEC.nivel, DEC.dirH)
Si DEC.clase = "tipo" and existePendientes(DEC.lexema)
Entonces
        DECS.pend = eliminaPendientes(DECS.pendH, DEC.lexema)
Si no
        DECS.pend = DECS.pendH
Fin si
DECS.etq = DEC.etq
DECS.cod = DEC.cod
DECS.dir = DEC.dir+DEC.decSize

DEC ≡ {DECVAR}
DECVAR.tsph = DEC.tsph
DECVAR.pendH = DEC.pendH
DEC.pend = DECVAR.pend
DEC.dir = DEC.dirh
DEC.clase = "variable"
DEC.lexema = DECVAR.lexema
DEC.props = DECVAR.props
DEC.errorDec = DECVAR.error
DEC.etq = DEC.etqh
DEC.cod = "cadena vacia"
DEC.decSize = DECVAR.decSize

DECVAR ≡ {iden} : {TIPOIDEN}
TIPOIDEN.pendH = DECVAR.pendH
DECVAR.pend = TIPOIDEN.pend
TIPOIDEN.tsph = DECVAR.tsph
DECVAR.lexema = iden.lexema
DECVAR.props = TIPOIDEN.props
DECVAR.error = existeID(DECVAR.tsph, iden.lexema) or
        TIPOIDEN.error
DECVAR.decSize = TIPOIDEN.size

TIPOIDEN ≡ {boolean}
TIPOIDEN.pend = TIPOIDEN.pendH
TIPOIDEN.error = FALSE
TIPOIDEN.props = <t: boolean>
TIPOIDEN.size = 1

TIPOIDEN ≡ { caracter }
TIPOIDEN.pend = TIPOIDEN.pendH
TIPOIDEN.props = <t: caracter >
TIPOIDEN.error = FALSE
TIPOIDEN.size = 1

TIPOIDEN ≡ { natural }
TIPOIDEN.pend = TIPOIDEN.pendH
TIPOIDEN.props = <t: natural >
TIPOIDEN.error = FALSE
TIPOIDEN.size = 1

TIPOIDEN ≡ { integer }
TIPOIDEN.pend = TIPOIDEN.pendH
TIPOIDEN.props = <t: integer >
TIPOIDEN.error = FALSE
TIPOIDEN.size = 1

TIPOIDEN ≡ { float }
TIPOIDEN.pend = TIPOIDEN.pendH

```

```

TIPODEN.props = <t: float >
TIPOIDEN.error = FALSE
TIPOIDEN.size = 1

TIPOIDEN ≡ { iden }
    TIPOIDEN.pend = TIPOIDEN.pendH
    TIPODEN.props = <t:ref> ++ <id: iden.lexema>
    TIPOIDEN.error = not existeTipo(TIPOIDEN.tsph, iden.lexema)
    TIPOIDEN.size = iden.size

TIPOIDEN ≡ {record} {/{ } {CAMPOS} {/}}
    CAMPOS.pendH = TIPOIDEN.pendH
    TIPOIDEN.props = <t:rec> ++ < campos: CAMPOS.props>
    TIPOIDEN.error = CAMPOS.error
    TIPOIDEN.pend = CAMPOS.pend

TIPOIDEN ≡ {pointer} {TIPOIDEN}
    TIPOIDEN_1.pendH =TIPOIDEN_0.pendH
    TIPOIDEN_0.props = <t:puntero> ++ <tbase: ++ TIPOIDEN_1.props>
    TIPOIDEN.size = 1
    si TIPOIDEN_0.error and TIPOIDEN_1.tipo = ref and
        not existeID(TIPOIDEN_0.tsph, TIPOIDEN_1.props.id)
    entonces
        TIPOIDEN_0.pend = añadePendiente(TIPOPENDIENTE_1.pend,
            TIPOIDEN_1.props.id)
        TIPOIDEN_0.error = FALSE
    si no
        TIPOIDEN_0.pend = TIPOPENDIENTE_1.pend
    Fin si

TIPOIDEN ≡ {array} {[ ] {natural} [ ]} {of} {TIPOIDEN}
    TIPOIDEN_1.pendH =TIPOIDEN_0.pendH
    TIPOIDEN_0.props = <t:array> ++ <nelem: natural.valor> ++
        <tbase: TIPOIDEN_1.props>
    TIPOIDEN_0.error = TIPOIDEN_1.error
    TIPOIDEN_0.size = TIPOIDEN_1.size * natural.lexema
    TIPOIDEN_0.pend =TIPOIDEN_1.pend

CAMPOS ≡ {CAMPOS} {;} {CAMPO}
    CAMPOS_1.pendH = CAMPOS_0.pend
    CAMPO.pendH = CAMPOS_1.pend
    CAMPOS_0.pend = CAMPO.pend
    CAMPO.tsph = CAMPOS_0.tsph
    CAMPOS_1.tsph = CAMPOS.tsph
    CAMPOS_0.props = CAMPOS_1.props++CAMPO.props
    CAMPOS_0.error = CAMPOS_1.error or CAMPO.error
    CAMPOS_0.size = CAMPOS_1.size + CAMPO.size

CAMPOS ≡ {CAMPO}
    CAMPO.pendH = CAMPOS.pendH
    CAMPO.tsph = CAMPOS.tsph
    CAMPOS.props = CAMPO.props
    CAMPOS.error = CAMPO.error
    CAMPOS.size = CAMPO.size
    CAMPOS_0.pend = CAMPO.pend

CAMPO ≡ {iden} : {TIPOIDEN}
    CAMPO.props = <id: iden.lexema> ++ <t: _TIPOIDEN.props.t>
    CAMPO.error = existeID(CAMPO.tsph, iden.lexema) or

```

```

        TIPOIDEN.error
CAMPO.size = TIPOIDEN.size

DEC ≡ {DECTIP}
    DECTIP.tsph = DEC.tsph
    DEC.decSize = 0
    DEC.dir = DEC.dirH
    DEC.etq = DEC.etqh
    DEC.cod = "cadena vacia"
    DEC.clase = "tipo"
    DEC.lexema = DECTIP.lexema
    DEC.props = DECTIP.props
    DEC.errorDec = DECTIP.error

DECTIP ≡ {tipo} {iden} {=} {TIPOIDEN}
    TIPOIDEN.tsph = DECTIP.tsph
    DECTIP.lexema = iden.lexema
    DECTIP.props = TIPOIDEN.props
    DECTIP.error = existeID(DECTIP.tsph, iden.lexema) or
        TIPOIDEN.error

DEC ≡ {DECPROC}
    DECPROC.etqh = DEC.etqh
    DECPROC.tsph=DEC.tsph
    DECPROC.nivelH = DEC.nivelH
    DECPROC.callPendH = DEC.callPend
    DEC.callPend = DECPROC.callPend
    DEC.clase = "procedimiento"
    DEC.lexema = DECPROC.lexema
    DEC.props = DECPROC.props
    DEC.errorDec = DECPROC.error
    DEC.etq = DECPROC.etq
    DEC.cod = DECPROC.cod
    DEC.nivel = DECPROC.nivel

DECPROC ≡ {proc} {iden} {DPARAMS} {PBLOQUE}
    DPARAMS.tsph = creaTS(DECPROC.tsph)
    DPARAMS.dirH = 2
    DPARAMS.nivelh= DECPROC.nivel+1
    PBLOQUE.nivelh=DECPROC.nivel+1
    PBLOQUE.tsph = añadeID(DPARAMS.tsp, DECPROC.lexema,
        DECPROC.props, DECPROC.nivel+1)
    PBLOQUE.dirH = DPARAMS.dir
    PBLOQUE.etqh = DECPROC.etqh + longPrologo
    PBLOQUE.callPendH = DECPROC.callPendH
    DECPROC.callPend = PBLOQUE.callPend
    DECPROC.lexema = iden.lexema
    DECPROC.clase = "procedimiento"
    DECPROC.params =<t:proc>++<params:DPARAMS.params>

    DECPROC.cod = prologo(DPARAMS.nivelh, DPARAMS.size+
        PBLOQUE.dirH)||PBLOQUE.cod||epilogo (DPARAMS.nivelh) ||
        ir_ind()
    DECPROC.error = existeId(iden) or DPARAMS.error or PBLOQUE.error
    DECPROC.etq = PBLOQUE.etq + longEpilogo +1
        // el +1 es por el ir_ind

PBLOQUE ≡ {forward}
    PBLOQUE.ts = PBLOQUE.tsh

```

```

PBLOQUE.error = FALSE
PBLOQUE.etq = PBLOQUE.etqh
PBLOQUE.forward = TRUE
PBLOQUE.callPendH = PBLOQUE.callPend

PBLOQUE ≡ {{}}{DECS}{&}{SENTS}{{}}
  DECS.etqh = PBLOQUE.etqh
  DECS.tsph=PBLOQUE.tsph
  DECS.nivel=PBLOQUE.nivel
  DECS.dirH = PBLQUE.dirH
  DECS.callPendH = PBLOQUE.callPendH
  SENTS.callPendH = DECS.callPend
  SENTS.etqh = DECS.etq
  SENTS.tsh= DECS.tsp
  PBLOQUE.ts=DECS.tsp
  PBLOQUE.error = SENT.errorSent or DECS.errorDec
  PBLOQUE.etq = SENT.etqh
  PBLOQUE.forward = FALSE
  PBLOQUE.callPend = resolverPend(DECS.tsph, SENTS.callPend)

PBLOQUE ≡ {{}} {&} {SENTS}{{}}
  SENTS.etqh = PBLOQUE.etqh
  SENTS.ts= PBLOQUE.tsph
  SENTS.callPendH = PBLOQUE.callPendH
  PBLQUE.error = SENT.errorSent
  PBLOQUE.etq = SENT.etqh
  PBLOQUE.forward = FALSE
  PBLOQUE.params = PBLOQUE.paramsH ++ <forward : false>
  PBLOQUE.callPend = resolverPend(PBLOQUE.tsph, SENTS.callPend)

DPARAMS ≡ {()} {LISTAPARAMS} {}
  LISTAPARAMS .dirH = DPARAMS.dirH
  LISTAPARAMS.tsph= DPARAMS.tsph
  LISTAPARAMS.nivelh = DPARAMS.nivelh
  DPARAMS.size = LISTAPARAMS.size
  DPARAMS.param=LISTAPARAMS.param
  DPARAMS.tsp = LISTAPARAMS.tsp
  DPARAMS.error = LISTAPARAMS.error

DPARAMS ≡ {lambda}
  DPARAMS.dir = DPARAMS.dirH
  DPARAMS.props = {}
  DPARAMS.tsp= DPARAMS.tsph
  DPARAMS.error = FALSE

LISTAPARAMS ≡ {LISTAPARAMS} {,} {PARAM}
  LISTAPARAMS_1.dirH = LISTAPARAMS_0.dirH
  LISTAPARAMS_0.param = LISTAPARAMS_1.param ++ PARAM.param
  LISTAPARAMS_0.error = LISTAPARAMS_1.error or PARAM.error
  PARAM.dirH = LISTAPARAMS.dir
  si not LISTAPARAMS_0.error entonces
    LISTAPARAMS_0.tsp = añadeID(LISTAPARAMS_1.tsp,
      PARAM.lexema,PARAM.props, PARAM.clase,
      LISTAPARAMS_0.nivelh, LISTAPARAMS_1.dir)
  fin si
  LISTAPARAMS_0.dir = LISTAPARAMS_1.dir + PARAM.dirH

LISTAPARAMS ≡ {PARAM}
  LISTAPARAMS.param = PARAM.param
  LISTAPARAMS.error = PARAM.error
  si not LISTAPARAMS.error entonces

```

```

        LISTAPARAMS.ts = añadeID(LISTAPARAMS.tsph, PARAM.lexema,
                                PARAM.props,PARAM,clase,
                                LISTAPARAMS.nivelh, LISTAPARAMS.dirH)
    fin si
    LISTAPARAMS.dir = LISTAPARAMS.dirH + PARAM.size

PARAM ≡ {var} {iden} {:} {TIPOIDEN}
PARAM.clase = "p_variable"
PARAM.lexema = iden.lexema
PARAM.props = <t: TIPOIDEN.props>
PARAM.param = <modo : variable> <t: TIPOIDEN.props.t>
PARAM.size = 1

PARAM ≡ {iden} {:} {TIPOIDEN}
TIPOIDEN.tsph = PARAM.tsph
PARAM.id= iden.lexema
PARAM.clase = "variable"
PARAM.props = <t: TIPOIDEN.props.t>
PARAM.param = <modo:valor, t: TIPOIDEN.props>
PARAM.error = existeId(PARAM.tsph,iden.lexema) or TIPOIDEN.error
PARAM.size = TIPOIDEN.size

SENTS ::= SENTS ; SENT
SENTS_1.callPendH = SENTS_0.callPendH
SENTS_1.tsh = SENT.tsh = SENTS_0.tsh
SENTS_0.errorSent = SENTS_1.errorSent OR SENT.errorSent
SENTS_1.etqh = SENTS_0.etqh
SENT.etqh = SENTS_1.etq
SENT.callPendH = SENTS_1.callPend  SENTS_0.etq = SENT.etq
SENTS_0.cod = SENTS_1.cod || SENT.cod
SENTS.callPend = SENT.callPend

SENTS ::= SENT
SENT.callPendH = SENTS.callPendH    SENT.tsh = SENTS.tsh
SENTS.errorSent = SENT.errorSent
SENT.etqh = SENTS.etqh
SENTS.etq = SENT.etq
SENTS.cod = SENT.cod
SENTS.callPend = SENT.callPend

SENT ::= SWRITE
SWRITE.tsh = SENT.tsh
SWRITE.etqh = SENT.etqh
SENT.errorSent = SWRITE.errorSent
SENT.etq = SWRITE.etq
SENT.cod = SWRITE.cod

SENT ::= SREAD
SREAD.tsh = SENT.tsh
SREAD.etqh = SENT.etqh
SENT.errorSent = SREAD.errorSent
SENT.etq = SREAD.etq
SENT.cod = SREAD.cod

SENT ::= SBLOQUE
SBLOQUE.tsh = SENT.tsh
SBLOQUE.etqh = SENT.etqh
SENT.errorSent = SBLOQUE.errorSent
SENT.etq = SBLOQUE.etq
SENT.cod = SBLOQUE.cod

```



```

SENT ::= SIF
    SIF.tsh = SENT.tsh
    SIF.etqh = SENT.etqh
    SENT.errorSent = SIF.errorSent
    SENT.etq = SIF.etq
    SENT.cod = SIF.cod

SENT ::= SWHILE
    SWHILE.tsh = SENT.tsh
    SWHILE.etqh = SENT.etqh
    SENT.errorSent = SWHILE.errorSent
    SENT.etq = SWHILE.etq
    SENT.cod = SWHILE.cod

SENT ::= SFOR
    SFOR.tsh = SENT.tsh
    SFOR.etqh = SENT.etqh
    SENT.errorSent = SFOR.errorSent
    SENT.etq = SFOR.etq
    SENT.cod = SFOR.cod

SWRITE ::= out ( EXP )
    EXP.tsh = SWRITE.tsh
    EXP.etqh = SWRITE.etqh
    SWRITE.errorSent = (EXP.tipo = tError)
    SI SWRITE.errorSent
    ENTONCES
        SWRITE.cod = "Cadena vacía"
    SI NO
        SWRITE.cod = EXP.cod || escribir
    FIN SI
    SWRITE.etq = EXP.etq + 1

SREAD ::= in ( iden )
    SREAD.errorSent = (NOT existeVar(SREAD.tsh, iden.lexema))
    SI SREAD.errorSent
    ENTONCES
        SREAD.cod = "Cadena vacía"
    SI NO
        SREAD.cod = leer ||
            desapila_dir(damePropiedadesTS(SREAD.tsh
                , iden.lexema).dirProp)
    FIN SI
    SREAD.etq = 2

SBLOQUE ::= { SENTs }
    SENTs.tsh = SBLOQUE.tsh
    SENTs.etqh = SBLOQUE.etqh
    SBLOQUE.errorSent = SENTs.errorSent
    SBLOQUE.etq = SENTs.etq
    SBLOQUE.cod = SENTs.cod

SIF ::= if EXP then SENT PELSE
    EXP.etqh = SIF.etqh
    EXP.tsh = SIF.tsh
    PELSE.tsh = SIF.tsh
    PELSE.etqh = SENT.etq + 1
    SENT.tsh = SIF.tsh
    SENT.etqh = EXP.etq + 1

```

```

SIF.errorSent = (EXP.tipo <> tBool) OR SENT.errorSent OR
    PELSE.errorSent
SIF.cod = EXP.cod || ir-f(SENT.etq + 1) || SENT.cod ||
    ir-a(PELSE.etq) || PELSE.cod
SIF.etq = PELSE.etq

PELSE ::= else SENT
SENT.tsh = PELSE.tsh
SENT.etqh = PELSE.etqh
PELSE.errorSent = SENT.errorSent
PELSE.etq = SENT.etq
PELSE.cod = SENT.cod

PELSE ::= lambdaλ
PELSE.errorSent = FALSE
PELSE.cod = λ
PELSE.etq = PELSE.etqh

SWHILE ::= while EXP do SENT
EXP.tsh = SWHILE.tsh
EXP.etqh = SWHILE.etqh
SENT.etqh = EXP.etq + 1
SENT.etq = EXP.etq
SWHILE.errorSent = (EXP.tipo <> tBool) OR SENT.errorSent
SWHILE.cod = EXP.cod || ir-f(SENT.etq + 1) || SENT.cod
    || ir-a(SWHILE.etqh)
SWHILE.etq = SENT.etq + 1

SFOR ::= for MEM = EXP to EXP do SENT
MEM.tsh = SFOR.tsh
MEM.etqh = SFOR.etqh
EXP_0.tsh = SFOR.tsh
EXP_0.etqh = MEM.etq
EXP_1.tsh = SFOR.tsh
EXP_1.etqh = EXP_0.etq + 2
SENT.tsh = SFOR.tsh
SENT.etqh = EXP_1.etq + 2
SFOR.errorSent = SENT.errorSent
    OR (NOT existeID(SFOR.tsh, iden.lexema))
    OR (NOT ((EXP_0.tipo = tNat) OR (EXP_0.tipo = tInt)))
    OR (NOT ((EXP_1.tipo = tNat) OR (EXP_1.tipo = tInt)))
    OR (NOT esCompatibleAsig?(dameTipoTS(SFOR.tsh,
        iden.lexema), EXP_0.tipo))
    OR (NOT esCompatibleAsig?(dameTipoTS(SFOR.tsh,
        iden.lexema), EXP_1.tipo))
SFOR.cod = EXP_0.cod
    || desapila_dir(damePropiedadesTS(SFOR.tsh,
        iden.lexema).dirProp)
    || apila_dir(damePropiedadesTS(SFOR.tsh,
        iden.lexema).dirProp)
    || EXP_1.cod || menorIgual || ir-f(SENT.etq + 5) || SENT.cod
    || apila(damePropiedadesTS(SFOR.tsh, iden.lexema).dirProp)
    || apila(1) || suma
    || desapila_dir(damePropiedadesTS(SFOR.tsh,
        iden.lexema).dirProp)
    || ir-a(SFOR.etqh)
SFOR.etq = SENT.etq + 5

SNEW ≡ {new} {mem}
MEM.etqh = SNEW.etqh
MEM.tsh = SNEW.tsh

```

```

Si not Mem.tipo = puntero entonces
    SNEW.errorSent = TRUE
Si no
    si MEM.tipo.tBase = ref entonces
        Tam = dameProps(SNEW.tsh, MEM.tipo.tBase.id).tam
    Si no
        Tam = dameProps(SNEW.tsh, MEM.tipo.tBase).tam
    SNEW.cod = Mem.cod || new(tam)
    Desapila_ind()
    SNEW.etq = MEM.etq + 2

SDEL ≡ {dispose} {mem}
MEM.etqh = SDEL.etqh
MEM.tsh = SDEL.tsh
Si not Mem.tipo = puntero entonces
    SDEL.errorSent = TRUE
Si no
    si MEM.tipo.tBase = ref entonces
        Tam = dameProps(SDEL.tsh, MEM.tipo.tBase.id).tam
    Si no
        Tam = dameProps(SDEL.tsh, MEM.tipo.tBase).tam
    Fin si
    SNEW.cod = Mem.cod || del(tam)
    SNEW.etq = MEM.etq + 1
Fin si

SENT ::= {iden} {RSENT}
RSENT.tsh = MEM.tsh
RSENT.tipoh = dameTipo(SENT.tsh, iden.lexema)
RSENT.tipo = RMEM.tipo
RSENT.iden = iden.lexema
RSENT.etqh = SENT.etqh
RSENT.callPendH = SENT.callPendH
SENT.etq = RSENT.etq
SENT.errorSent = RSENT.errorSent
SENT.cod = RSENT.cod
SENT.callPend = RSENTS.callPend

RSENT ::= {RMEM} {:=} {EXP}
RMEM.tipoh = RSENT.tipoh
RMEM.etqh = EXP.etqh + 1
RMEM.tsh = RSENT.tsh
Si damePropsTS(RSENT.tsh, RSENT.iden).clase = pvar entonces
    codTemp = apila_ind()
    nCodTemp = 1
si no
    nCodTemp = 0
fin si

EXP.etqh = RSENT.etqh
RSENT.errorSent = errorTipos(RMEM.tipo, EXP.tipo)
si not RSENT.errorSent and EXP.modos = val entonces
    RSENT.cod = codTemp
    || EXP.cod
    || apila(damePropiedadesTS(RSENT.tsh, iden.lexema).dir)
    || RMEM.cod
    || desapila_ind()

```

```

        //copia mueve lo que hay en lo 2° a lo 1°

Si not si RSENT.errorSent and EXP.modos = var entonces
    size = dameSize(RSENT.tsh,RMEM.tipo)
    RSENT.cod= codTemp
        || EXP.cod
        || apila(damePropiedadesTS(RSENT.tsh,iden.lexema).dir)
        || RMEM.cod
        || mueve(size)
        //copia mueve lo que hay en lo 2° a lo 1°

Fin si
RSENT.etq = RMEM.etq+2+nCodTemp

RSENT ::= {PPARAMS}
PPARAMS.nivel = RSENT.nivel
PPARAMS.tsh = RSENT.tsh
PPARAMS.procName = RSENT.iden
RSENT.errorSent = PPARAMS.error
dirProc = dameDir(PPARAMS.tsh, RSENT.iden)
RSENT.cod = apila-ret(RSENT.etqh) ||
    || PPARAMS.cod
    || ir_a(dirProc) \\ la dirección esta en la pila
    || //codigo de postllamada
si dirProc = -1 entonces
    RSENT.callPend = añadeCallPend(RSENT.callPendH,RSENT.iden,
        PPARAMS.etq + 1)
RSENT.etq = PPARAMS.etq + longApilaRet + 1

PPARAMS ≡ {() {LPARAMS} {} }
LPARAMS.procName = PPARAMS.procName
LPARAMS.paramsSize = 0
LPARAMS.tsh = PPARAMS.tsh
LPARAMS.nParamH = 0
PPARAMS.error = LPARAMS.error or
    damePropTS(PPARAMS.tsh,PPARAMS.procName).nParams <>
    LPARAMS.nParams
PARAMS.cod = LPARAMS.cod

PPARAMS ≡ λ
PPARAMS.error = FALSE
PPARAMS.nParams = 0
PPARAMS.etqh = PPARAMS.etq
PPARAMS.cod = "cadena vacia"

LPARAMS ::= {EXP} {RLPARAMS}
EXP.etqh = LPARAMS.etqh
LPARAMS.nParams = RLPARAMS.nParams
RLPARAMS.nParamsH = 1
RLPARAMS.procName = LPARAMS.procName
LPARAMS.error =
    comparaParamFunc(LPARAMS.tsh,LPARAMS.procName,
        0, EXP.modos, EXP.tipo)
    or RLPARAMS.error or EXP.error
si not LPARAM_0.errorSent and (EXP.modos = val or
    parametroPorValor(LPARAMS_0.tsh,LPARAMS_0.procName,
        LPARAMS.nParams))
entonces
    RLPARAMS.paramsSizeH = 1
    LPARAMS_0.cod= EXP.cod

```

```

        || apila(0)
        || apila_ind()
        || apila(2)
        || suma
        || apila_ind()
        || RLPARAMS.cod
    RLPARAMS.etqh = EXP.etq+2
Si not si RSENT.errorSent entonces
    size = dameSize(RSENT.tsh,EXP.tipo)
    RLPARAMS.paramsSizeH = size
    LPARAMS_0.cod= EXP.cod
        || apila(0)
        || apila_ind()
        || apila(2)
        || suma
        || copia(size)
        || RLPARAMS.cod
        //copia mueve loque hay en lo 2° a lo 1°
    RLPARAMS.etqh = EXP.etq+2
    LPARAMS.paramsSize = RLPARAMS.paramsSize + size
Fin si
RLPARAMS ::= {,}{EXP} {RLPARAMS}
    RLPARAMS_0.nParams = RLPARAMS_1.nParams
    RLPARAMS_1.nParamsH = 1
    RLPARAMS_1.procName = RLPARAMS_0.procName
    EXP.etqh = RLPARAMS_0.etqh
    RLPARAMS_0.error =
        comparaParamFunc(RLPARAMS_0.tsh,RLPARAMS_0.procName,
            0, EXP.mod0, EXP.tipo)
        or RLPARAMS_1.error or EXP.error
si not LPARAM_0.errorSent and (EXP.mod0 = val or
    parametroPorValor(LPARAMS_0.tsh,LPARAMS_0.procName,
        LPARAMS.nParams))
Entonces
    RLPARAMS_1.paramsSizeH = RLPARAMS_0.paramsSizeH+1
    LPARAMS_0.cod= EXP.cod
        || apila(0)
        || apila_ind()
        || apila(2)
        || suma
        || apila_ind()
        || RLPARAMS_1.cod
    RLPARAMS.etqh = EXP.etq+2
    RLPARAMS_0.paramsSize = RLPARAMS_1.paramsSize + 1
Si not si RSENT.errorSent entonces
    size = dameSize(RSENT.tsh,EXP.tipo)
    RLPARAMS_1.paramsSizeH = RLPARAMS_0.paramsSizeH+size
    LPARAMS_0.cod= EXP.cod
        || apila(0)
        || apila_ind()
        || apila(2)
        || suma
        || copia(size)
        || RLPARAMS_1.cod
        //copia mueve loque hay en lo 2° a lo 1°
    RLPARAMS_1.etqh = EXP.etq+2
    RLPARAMS_0.paramsSize = RLPARAMS_1.paramsSize + size
Fin si

RLPARAMS ::= lambda
    RLPARAMS.error = FALSE

```

```

RLPARAMS.paramsSize = RLPARAMS.paramsSizeH
RLPARAMS.etq = RLPARAMS.etqh
RLPARAMS.nParams = RLPARAMS.nParamsH

```

```

EXP ::= EXP1 OP0 EXP1
    EXP1_0.tsh = EXP1_1.tsh = EXP.tsh
    EXP1_0.etqh = EXP.etqh
    EXP.mod0 = val
    EXP.tipo = dameTipo(EXP1_0.tipo, EXP1_1.tipo, OP0.op)
    SI (EXP.tipo = tError)
    ENTONCES
        EXP.cod = "Cadena vacía"
    SI NO
        nIns = 0
        Ins = "cadena vacia"
        Si EXP1_0.mod0 = var
            nIns = nIns+1
            Ins = Ins || EXP1_0.cod || apila_ind()
        Si no
            Ins = Ins || EXP1.cod
        Fin si
        EXP1_1.etqh = EXP1_0.eta + nIns
        nIns = 0
        Si EXP1_1.mod0 = var
            nIns = nIns +1
            Ins = Ins || EXP1_1.cod || apila_ind()
        Si no
            Ins = Ins || EXP1_1.cod
        Fin si
        EXP.cod = Ins || OP0.op
        nIns = nIns +1
        EXP.etq = EXP1_1.eta +nIns
    FIN SI
    EXP1_0.etqh = EXP.etqh
    EXP1_1.etqh = EXP1_0.etq
    EXP.etq = EXP1_1.etq + 1

EXP ::= EXP1
    EXP1.tsh = EXP.tsh
    EXP.mod0 = EXP1.mod0
    EXP.tipo = EXP1.tipo
    EXP1.etqh = EXP.etqh
    EXP.etq = EXP1.etq
    EXP.cod = EXP1.cod

EXP1 ::= EXP1 OP1 EXP2
    EXP1.mod0 = val
    EXP1_1.tsh = EXP2.tsh = EXP1_0.tsh
    EXP1_1.etqh = EXP1_0.etqh
    EXP1_0.tipo = dameTipo(EXP1_1.tipo, EXP2.tipo, OP1.op)
    SI (EXP1_0.tipo = tError)
    ENTONCES
        EXP1_0.cod = "Cadena vacía"
    SI NO si (OP1.op = oLogica)
        nIns = 0
        Ins = "cadena vacia"
        Si EXP1_1.mod0 = var
            nIns = nIns+1
            Ins = Ins || EXP1_1.cod || apila_ind()
        Si no

```

```

        Ins = Ins || EXP1_1.cod
    Fin si
    Ins = Ins || copia || ir_v(EXP2.etq) ||
        desalipa
    Ins = Ins + 3
    EXP2.etqh = EXP1_1.etq + Ins
    nIns = 0
    Si EXP2.modo = var
        nIns = nIns+1
        Ins = Ins || EXP2.cod || apila_ind()
    Si no
        Ins = Ins || EXP2.cod
    Fin si
    EXP1_0.cod = Ins
    EXP1_0.etq = EXP2.etq + Ins
si no
    nIns = 0
    Ins = "cadena vacia"
    Si EXP1_1.modo = var
        nIns = nIns+1
        Ins = Ins || EXP1_1.cod || apila_ind()
    Si no
        Ins = Ins || EXP1_1.cod
    Fin si
    Si EXP2.modo = var
        nIns = nIns +1
        Ins = Ins || EXP2.cod || apila_ind()
    Si no
        Ins = Ins || EXP2.cod
    Fin si
    EXP1_0.cod = Ins || OP1.op
    nIns = nIns +1
    EXP1_0.etq = EXP2.etq + Ins
FIN SI
EXP1_1.etqh = EXP1_0.etqh

```

```

EXP1 ::= EXP2
    EXP1.modo = EXP2.modo
    EXP2.tsh = EXP1.tsh
    EXP1.tipo = EXP2.tipo
    EXP2.etqh = EXP1.etqh
    EXP1.etq = EXP2.etq
    EXP1.cod = EXP2.cod

EXP2 ::= EXP2 OP2 EXP3
    EXP2.modo = val
    EXP2_1.tsh = EXP3.tsh = EXP2_0.tsh
    EXP2_0.tipo = dameTipo(EXP2_1.tipo, EXP3.tipo, OP2.op)
    EXP2_1.etqh = EXP2_0.etqh
    SI (EXP2_0.tipo = tError)
    ENTONCES
        EXP2_0.cod = "Cadena vacía"
    si no si (OP2.op = yLogica)
        nIns = 0
        Ins = "cadena vacia"
        Si EXP2_1.modo = var
            nIns = nIns+1
            Ins = Ins || EXP2_1.cod || apila_ind()
        Si no
            Ins = Ins || EXP2_1.cod

```

```

Fin si
Ins = Ins || ir_f(EXP.etq + 2)
Ins = Ins + 1
EXP3.etqh = EXP2_1.etq + Ins
nIns = 0
Si EXP3.modo = var
    nIns = nIns+1
    Ins = Ins || EXP3.cod || apila_ind()
Si no
    Ins = Ins || EXP3.cod
Fin si
EXP2_0.cod = Ins || ir_a(EXP.etq+2) || apila (FALSE)
nIns = nIns + 2
EXP2_0.etq = EXP3.etq + nIns
SI NO
    nIns = 0
    Ins = "cadena vacia"
    Si EXP2_1.modo = var
        nIns = nIns+1
        Ins = Ins || EXP2_1.cod || apila_ind()
    Si no
        Ins = Ins || EXP2_1.cod
    Fin si
    Si EXP3.modo = var
        nIns = nIns +1
        Ins = Ins || EXP3.cod || apila_ind()
    Si no
        Ins = Ins || EXP3.cod
    Fin si
    EXP2_0.cod = Ins || OP2.op
    nIns = nIns +1
    EXP2_0.etq = EXP3.etq + Ins

FIN SI

```

```

EXP2 ::= EXP3
EXP2.modo = EXP3.modo
EXP3.tsh = EXP2.tsh
EXP2.tipo = EXP3.tipo
EXP3.etqh = EXP2.etqh
EXP2.etq = EXP3.etq
EXP2.cod = EXP3.cod

EXP3 ::= EXP4 OP3 EXP3
EXP3.modo = val
EXP4.etqh = EXP3_0.etqh
EXP3_1.tsh = EXP4.tsh = EXP3_0.tsh
EXP3_0.tipo = dameTipo(EXP4.tipo, EXP3_1.tipo, OP3.op)
SI (EXP4_0.tipo = tError)
    EXP3_0.cod = "Cadena vacia"
SI NO
    nIns = 0
    Ins = "cadena vacia"
    Si EXP4.modo = var
        nIns = nIns+1
        Ins = Ins || EXP4.cod || apila_ind()
    Si no
        Ins = Ins || EXP4.cod
    Fin si

```



```

EXP3_1.etqh = EXP4.eta + nIns
nIns = 0

Si EXP3_1.mod0 = var
    nIns = nIns + 1
    Ins = Ins || EXP3_1.cod || apila_ind()
Si no
    Ins = Ins || EXP3_1.cod
Fin si
EXP3_0.cod = Ins || OP3.op
nIns = nIns + 1
FIN SI
EXP3_0.etq = EXP3_1.etq + nIns

EXP3 ::= EXP4
EXP3.mod0 = EXP4.mod0
EXP4.tsh = EXP3.tsh
EXP3.tipo = EXP4.tipo
EXP4.etqh = EXP3.etqh
EXP3.etq = EXP4.etq
EXP3.cod = EXP4.cod

EXP4 ::= OP4_1 TERM
TERM.tsh = EXP4.tsh
TERM.etqh = EXP4.etqh
EXP4.mod0 = val
EXP4.tipo = dameTipo(TERM.tipo, OP4_1.op)
SI (EXP4.tipo = tError)
    ENTONCES
        EXP4.cod = "Cadena vacía"
    SI NO
        si TERM.mod0 = var
            EXP4.cod = TERM.cod || apila_ind() || OP4_1.op
            EXP4.etq = TERM.etq + 2
        Si no
            EXP4.cod = TERM.cod || OP4_1.cod
            EXP4.etq = TERM.etq + 1
        FIN SI
    FIN SI

EXP4 ::= \ | TERM \ |
EXP4.mod0 = val
EXP4.tipo = dameTipo(TERM.tipo, "|")
TERM.tsh = EXP4.tsh
SI (EXP4.tipo = tError)
    ENTONCES
        EXP4.cod = "Cadena vacía"
    SI NO
        si TERM.mod0 = var
            EXP4.cod = TERM.cod || apila_ind() || valorAbs
            EXP4.etq = TERM.etq + 2
        Si no
            EXP4.cod = TERM.cod || valorAbs
            EXP4.etq = TERM.etq + 1
        FIN SI
    FIN SI
    TERM.etqh = EXP4.etqh

EXP4 ::= TERM
TERM.tsh = EXP4.tsh
TERM.etqh = EXP4.etqh

```

```

    EXP4.modo = TERM.modo
    EXP4.tipo = TERM.tipo
    EXP4.etq = TERM.etq
    EXP4.cod = TERM.cod

TERM ::= MEM
    MEM.etqh = TERM.etqh
    MEM.tsh = TERM.tsh
    TERM.modo = var
    TERM.tipo = MEM.tipo
    TERM.etq = MEM.etq
    TERM.cod = MEM.cod
    TERM.etq = MEM.etq

TERM ::= boolean
    TERM.modo = val
    TERM.tipo = tBool
    TERM.cod = apila(valorDe(boolean.lexema))
    TERM.etq = TERM.etqh + 1

TERM ::= cadCaracteres
    TERM.modo = val
    TERM.tipo = tChar
    TERM.cod = apila(valorDe(cadCaracteres.lexema))
    TERM.etq = TERM.etqh + 1

TERM ::= natural
    TERM.modo = val
    TERM.tipo = tNat
    TERM.cod = apila(valorDe(natural.lexema))
    TERM.etq = TERM.etqh + 1

TERM ::= entero
    TERM.modo = val
    TERM.tipo = tInt
    TERM.cod = apila(valorDe(entero.lexema))
    TERM.etq = TERM.etqh + 1

TERM ::= real
    TERM.modo = val
    TERM.tipo = tFloat
    TERM.cod = apila(valorDe(real.lexema))
    TERM.etq = TERM.etqh + 1

TERM ::= ( EXP )
    EXP.etqh = TERM.etqh
    EXP.tsh = TERM.tsh
    TERM.modo = EXP.modo
    TERM.tipo = EXP.tipo
    TERM.etq = EXP.etq
    TERM.cod = EXP.cod

MEM = {iden} {RMEM}
    RMEM.tsh = MEM.tsh
    RMEM.etqh = MEM.etqh
    RMEM.tipoh = damePropiedadesTS(MEM.tsh, iden.lexema).tipo
    Si damePropsTS(MEM.tsh, iden.lexema).clase = pvar entonces
        RMEM.etqh = MEM.etqh + 2
        MEM.cod = apila(damePropiedadesTS(MEM.tsh, iden.lexema).dir)
        || apila_ind()

```

```

        || RMEM.cod
    si no
        RMEM.etqh = MEM.etqh + 1
        MEM.cod = apila(damePropiedadesTS(MEM.tsh, iden.lexema).dir)
        || RMEM.cod
    fin si
    MEM.etq = RMEM.etq
    MEM.tipo = RMEM.tipo

RMEM = λ
    RMEM.tipoh = RMEM.tipoh
    RMEM.cod = apila(RMEM.nivel+1)
        || suma // no necesitamos poner el +2 porque
        // la direccion en la TS comienza en 2
        || apila_ind()
    RMEM.etq = RMEM.etqh + 3
    Si no
        RMEM.cod = apila(RMEM.nivel+1)
        || suma
        RMEM.etq = RMEM.etqh + 2
    fin si

RMEM = {->} RMEM
    RMEM_1.tsh = RMEM_0.tsh
    RMEM_1.tipoh = tipoBase(RMEM_0.tsh, RMEM_0.tipoh)
    RMEM_0.tipo = RMEM_1.tipo
    RMEM_0.cod = Apila_ind() || RMEM_1.cod

RMEM = {[EXP]} RMEM
    EXP.tsh = RMEM.tsh
    EXP.etqh = RMEM_0.etqh
    RMEM_1.etqh = RMEM_0.etqh
    RMEM_1.tsh = RMEM_0.tsh
    Si RMEM_0.tipo = tArray and EXP.tipo = tEntero entonces
        RMEM_0.tipo = RMEM_1.tipo
        Size = damePropiedadesTS(RMEM.tsh, RMEM.tipoh).size
        RMEM_0.etq = RMEM_1.etq
        RMEM_1.tipoh = tipoBase(RMEM_0.tsh, RMEM_0.tipoh)
        RMEM_1.etqh = EXP.etq + 3
        RMEM_0.cod = EXP.cod || Apila(size) || multiplica || suma
        || RMEM_1.cod
        RMEM_0.etq = RMEM_1.etq
    Si no
        RMEM.tipo = tError

RMEM = {[iden]} RMEM
    RMEM_1.tsh = RMEM_0.tsh
    RMEM_1.etqh = RMEM_0.etqh + 2
    RMEM_1.tipoh = tipoCampo(RMEM_0.tipoh, iden.lexema.RMEM_0.tsh)
    RMEM_0.tipo = RMEM_1.tipo
    Si not RMEM_0.tipo = tError entonces
        Offset = dameOffsetCampo(RMEM_0.tsh, RMEM)
        RMEM_0.cod = apila(offset) || apila (suma) || RMEM_1.cod

OP0 ::= <
    OP0.op = menor

OP0 ::= >
    OP0.op = mayor

```

```
OP0 ::= <=
      OP0.op = menorIgual

OP0 ::= >=
      OP0.op = mayorIgual

OP0 ::= =
      OP0.op = igual

OP0 ::= /=
      OP0.op = distinto

OP1 ::= +
      OP1.op = suma

OP1 ::= -
      OP1.op = resta

OP1 ::= or
      OP1.op = oLogica

OP2 ::= *
      OP2.op = multiplicacion

OP2 ::= /
      OP2.op = division

OP2 ::= %
      OP2.op = resto

OP2 ::= and
      OP2.op = yLogica

OP3 ::= <<
      OP3.op = despIzq

OP3 ::= >>
      OP3.op = despDer

OP4_1 ::= not
        OP4_1.op = negLogica

OP4_1 ::= -unario
        OP4_1.op = negArit

OP4_1 ::= (nat)
        OP4_1.op = castNat

OP4_1 ::= (int)
        OP4_1.op = castInt

OP4_1 ::= (char)
        OP4_1.op = castChar

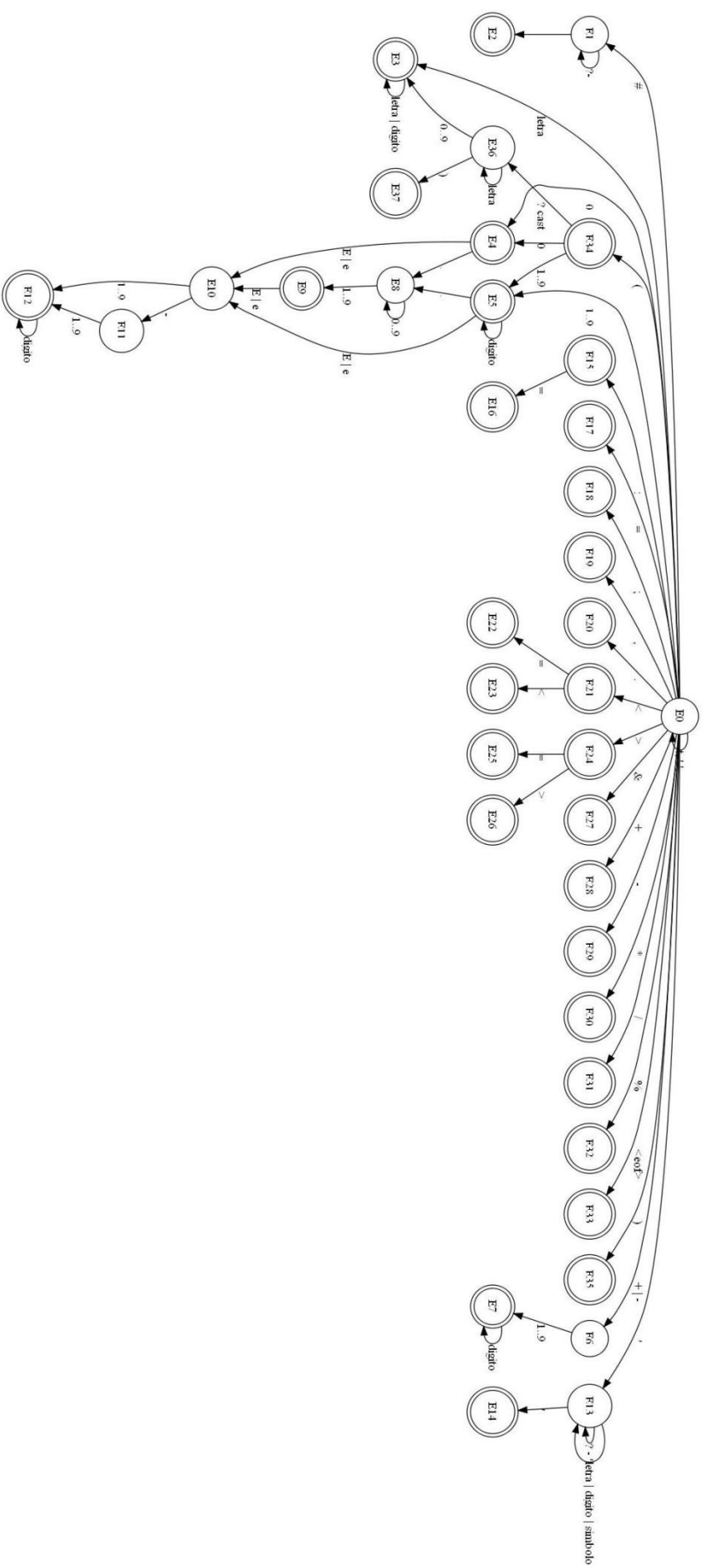
OP4_1 ::= (float)
        OP4_1.op = castFloat
```

6 Diseño del analizador léxico

Entendemos como:

- **Letra:** Cualquier letra [a..z | A..Z]
- **Digito:** Cualquier dígito [0..9]
- **¿:** Cualquier carácter alfanumérico
- **¿-(carácter):** Cualquier carácter alfanumérico excepto el especificado en (**carácter**)
- ' ': Espacio en blanco
- \n: Nueva línea
- \t: Tabulación

El Autómata se adjunta como una imagen aparte llamada automata.jpg



7 Acondicionamiento de las gramáticas de atributos

```
LPARAMS ≡ {LPARAMS} {EXP}  
LPARAMS ≡ {EXP}
```

Se cambia por

```
LPARAMS ::= {EXP} {RLPARAMS}  
    EXP.etqh = LPARAMS.etqh  
    LPARAMS.nParams = RLPARAMS.nParams  
    RLPARAMS.nParamsH = 1  
    RLPARAMS.procName = LPARAMS.procName  
    LPARAMS.error =  
        comparaParamFunc(LPARAMS.tsh,LPARAMS.procName,  
            0, EXP.mod0, EXP.tipo)  
        or RLPARAMS.error or EXP.error  
    si not LPARAM_0.errorSent and (EXP.mod0 = val or  
        parametroPorValor(LPARAMS_0.tsh,LPARAMS_0.procName,  
            LPARAMS.nParams))  
    entonces  
        RLPARAMS.paramsSizeH = 1  
        LPARAMS_0.cod= EXP.cod  
            || apila(0)  
            || apila_ind()  
            || apila(2)  
            || suma  
            || apila_ind()  
            || RLPARAMS.cod  
        RLPARAMS.etqh = EXP.etq+2  
        LPARAMS.paramsSize =  
RLPARAMS.paramsSize + 1  
    Si not si RSENT.errorSent entonces  
        size = dameSize(RSENT.tsh,EXP.tipo)  
        RLPARAMS.paramsSizeH = size  
        LPARAMS_0.cod= EXP.cod  
            || apila(0)  
            || apila_ind()  
            || apila(2)  
            || suma  
            || copia(size)  
            || RPARAMS.cod  
        //copia mueve lo que hay en lo 2° a lo 1°  
        RLPARAMS.etqh = EXP.etq+2  
        LPARAMS.paramsSize = RLPARAMS.paramsSize + size  
    Fin si  
RLPARAMS ::= {,}{EXP} {RLPARAMS}  
    RLPARAMS_0.nParams = RLPARAMS_1.nParams  
    RLPARAMS_1.nParamsH = 1  
    RLPARAMS_1.procName = RLPARAMS_0.procName  
    EXP.etqh = RLPARAMS_0.etqh  
    RLPARAMS_0.error =  
        comparaParamFunc(RLPARAMS_0.tsh,RLPARAMS_0.procName,  
            0, EXP.mod0, EXP.tipo)  
        or RLPARAMS_1.error or EXP.error  
    si not LPARAM_0.errorSent and (EXP.mod0 = val or  
        parametroPorValor(LPARAMS_0.tsh,LPARAMS_0.procName,  
            LPARAMS.nParams))  
    Entonces
```

```

RLPARAMS_1.paramsSizeH = RLPARAMS_0.paramsSizeH+1
LPARAMS_0.cod= EXP.cod
    || apila(0)
    || apila_ind()
    || apila(2)
    || suma
    || apila_ind()
    || RLPARAMS_1.cod
RLPARAMS.etqh = EXP.etq+2
RLPARAMS_0.paramsSize = RLPARAMS_1.paramsSize + 1
Si not si RSENT.errorSent entonces
    size = dameSize(RSENT.tsh,EXP.tipo)
    RLPARAMS_1.paraSizemsH = RLPARAMS_0.paramsSizeH+size
    LPARAMS_0.cod= EXP.cod
        || apila(0)
        || apila_ind()
        || apila(2)
        || suma
        || copia(size)
        || RLPARAMS_1.cod
        //copia mueve loque hay en lo 2° a lo 1°
    RLPARAMS_1.etqh = EXP.etq+2
    RLPARAMS_0.paramsSize = RLPARAMS_1.paramsSize + size
Fin si

RLPARAMS ::= lambda
    RLPARAMS.error = FALSE
    RLPARAMS.paramsSize = RLPARAMS.paramsSizeH
    RLPARAMS.etq = RLPARAMS.etqh
    RLPARAMS.nParams = RLPARAMS.nParamsH

```

Estas producciones tienen ambigüedad

```

LISTAPARAMS ≡ {LISTAPARAMS} {,} {PARAM}
LISTAPARAMS ≡ {PARAM}

```

Su transformación queda así:

```

LISTAPARAMS ::= {PARAM} {RLISTAPARAMS}
    PARAM.dirH = LISTAPARAMS.dirH
    RLISTAPARAMS.dirH = PARAM.dir
    RLISTAPARAMS.paramH = PARAM.param
    RLISTAPARAMS.nivelH = LISTAPARAMS.nivelH
    si not PARAM.error entonces
        RLISTAPARAMS.tsph = añadeID(LISTAPARAMS.tsph,
            PARAM.lexema,PARAM.props, PARAM.clase,
            LISTAPARAMS.nivelh, LISTAPARAMS.dirH)
        RLISTAPARAMS.dirH = LISTAPARAMS.dirH + PARAM.size
    fin si
    LISTAPARAMS.error = RLISTAPARAMS.error or PARAM.error
    LISTAPARAMS.dir = RLISTAPARAMS.dir
    LISTAPARAMS.param = RLISTAPARAMS.param

RLISTAPARAMS ::= {,} {PARAM} {RLISTAPARAMS}
    PARAM.dirH = RLISTAPARAMS_0.dirH
    RLISTAPARAMS_1.dirH = PARAM.dir
    RLISTAPARAMS_1.paramH = RLISTAPARAMS_0.paramH ++ PARAM.param
    RLISTAPARAMS_1.nivelH = RLISTAPARAMS_0.nivelH
    si not PARAM.error entonces
        RLISTAPARAMS_1.tsph = añadeID(RLISTAPARAMS_0.tsph,
            PARAM.lexema,PARAM.props, PARAM.clase,
            RLISTAPARAMS_0.nivelh, LISTAPARAMS.dirH)

```



```

        RLISTAPARAMS_1.dirH = RLISTAPARAMS_0.dirH + PARAM.size
    fin si
    RLISTAPARAMS_0.error = RLISTAPARAMS_1.error or PARAM.error
    RLISTAPARAMS_0.error = RLISTAPARAMS_1.error or PARAM.error
    RLISTAPARAMS_0.dir = RLISTAPARAMS_1.dir
    RLISTAPARAMS_0.param = RLISTAPARAMS_1.param

RLISTAPARAMS ::= lambda
    RLISTAPARAMS.error = FALSE
    RLISTAPARAMS.dir = RLISTAPARAMS.dirH
    RLISTAPARAMS.param = RLISTAPARAMS.paramH

```

Las producciones

```
CAMPOS ≡ {CAMPOS} {;} {CAMPO}
```

```
CAMPOS ≡ {CAMPO}
```

Queda trasformada en:

```

CAMPOS ≡ {CAMPO} {RCAMPOS}
    CAMPO.tsph = CAMPOS.tsph
    CAMPO.pendH = CAMPOS.pendH
    RCAMPOS.pendH = CAMPO.pend
    RCAMPOS.tsph = CAMPOS.tsph
    CAMPOS.props = CAMPO.props ++ RCAMPOS.props
    CAMPOS.error = CAMPO.erro or RCAMPOS.error
    CAMPOS.size = CAMPO.size + RCAMPOS.size
    CAMPOS.pend = RCAMPOS.pend

RCAMPOS ≡{;} {CAMPO} {RCAMPOS}
    CAMPO.tsph = RCAMPOS_0.tsph
    CAMPO.pendH = RCAMPOS_0.pendH
    RCAMPOS_1.pendH = CAMPO.pend
    RCAMPOS_1.tsph = RCAMPOS_0.tsph
    RCAMPOS_0.props = CAMPO.props ++ RCAMPOS_1.props
    RCAMPOS_0.error = CAMPO.erro or RCAMPOS_1.error
    RCAMPOS_0.size = CAMPO.size + RCAMPOS_1.size
    RCAMPOS_0.pend = RCAMPOS_1.pend

RCAMPOS ≡ lambda
    RCAMPOS.props = "cadena vacia"
    RCAMPOS.error = FALSE
    RCAMPOS.size = 0

```

La producción hay que adaptarla quitando recursión a izquierdas:

```
DECS ::= DECS ; DEC
```

Queda transformada en (Tras quitar recursion a izquierdas):

```

DECS ::= DEC RDECS
    DEC.etqH = DECS.etqH
    DEC.dirH = DECS.dirH
    DEC.nivel = DECS.nivel
    DEC.tsph = DECS.tsph
    DEC.pendH = RDECS_0.pendH
    DEC.callPendH = DECS.callPendH

```

```

RDECS.callPendH = DEC.callPend
RDECS.etqH = DEC.etq
RDECS.dirH = DEC.dir
RDECS.tsph = DEC.tsph
RDECS.nivel = DEC.nivel
DECS.errorDec = DEC.errorDec or RDECS.errorDec
si not DECS.errorDec and DEC.clase = procedimiento entonces
    si not DEC.forward and entonces
        si existeID(DECS.tsph, DEC.lexema) and
            not (pendiente(DEC.pend, DEC.lexema) or
                not tipoPendiente(DEC.pend, DEC.lexema)=proc)
        entonces
            DECS.errorSent = TRUE
        Si no si pendiente(DEC.pend, DEC.lexema)
            RDECS.pendH = eliminaPendiente(DEC.pend,
                DEC.lexema)
            DECS.errorDecs = compruebaCampos(DEC.tsph,
                DEC.lexema, DEC.props, DEC.clase,
                DEC.nivel, DEC.etqH)
            RDECS.tsph = actualizaID(DEC.tsph,
                DEC.lexema, DEC.etqH)
        Si no // no es un tipo "proc" de pendiente
            DECS.errorSent = TRUE
        fin si
    si no si existeID(RDECS_0.tsph, DEC.lexema) entonces
        DECS.errorSent = TRUE
    Si no // no existe ID y es forward
        RDECS.tsph = añadeID(DECS.tsph, DEC.lexema,
            DEC.props, DEC.clase, DEC.nivel, DEC.etqH)
        RDECS.pendH = añadePendientes(DEC.pend,
            DEC.lexema, proc)
    fin si
si not DECS.errorDec // es un tipo o una var
entonces
    si existeID(DECS.tsph, DEC.lexema) entonces
        DECS.errorDec = TRUE
    Si no
        RDECS.tsph= añadeID(DECS.tsph, DEC.lexema, DEC.props,
            DEC.clase, DEC.nivel, DEC.dirH)
        Si DEC.clase = "tipo"
            and existePendientes(DEC.pend, DEC.lexema)
        Entonces
            RDECS.pendH=eliminaPendientes(DEC.pend,
                DEC.lexema)
        Si no
            RDECS.pendH = DEC.pend
    Fin si
Fin si
DECS.pend = RDECS.pend
DECS.pend = RDECS.pend
DECS.etq = RDECS.etq
DECS.callPend = RDECS.callPend

```

```

RDECS ::= ; DEC RDECS
DEC.etqH = RDECS_0.etqH
DEC.tsph = RDECS_0.tsph
DEC.dirH = RDECS_0.dirH
DEC.nivel = RDECS_0.nivel

```

```

DEC.pendH = RDECS_0.pendH
DEC.callPendH = RDECS.callPendH
RDECS_1.callPendH = DEC.callPend
RDECS_1.etqH = DEC.etq
RDECS_1.dirH = DEC.dir
RDECS_1.tsph = DEC.tsph
RDECS_1.nivel = RDECS_0.nivel
RDECS_0.errorDec = DEC.errorDec or RDECS_1.errorDec
si not RDECS_0.errorDec and DEC.clase = procedimiento entonces
    si not DEC.forward and entonces
        si existeID(RDECS_0.tsph, DEC.lexema) and
            not (pendiente(DEC.pend,DEC.lexema) or
                not tipoPendiente(DEC.pend,DEC.lexema)=proc)
            entonces
                RDECS_0.errorSent = TRUE
            Si no si pendiente(DEC.pend,DEC.lexema)
                RDECS_0.errorDecs = compruebaCampos(DEC.tsph,
                    DEC.lexema, DEC.props,DEC.clase,
                    DEC.nivel, DEC.etqH)
                RDECS_1.pendH = eliminaPendiente(DEC.pend,
                    DEC.lexema)
                RDECS_1.tsph = actualizaID(DEC.tsph,
                    DEC.lexema,DEC.etqH)
            Si no // no es un tipo "proc" de pendiente
                RDECS_0.errorSent = TRUE
        si no si existeID(RDECS_0.tsph,DEC.lexema) entonces
            DECS_0.errorSent = TRUE
        Si no // no existe ID y es forward
            RDECS_1.tsph = añadeID(RDECS_0.tsph,DEC.lexema,
                DEC.props,DEC.clase, DEC.nivel, DEC.etqH)
            RDECS_1.pendH = añadePendientes(DEC.pend,
                DEC.lexema,proc)
        fin si
    si not DECS_0.errorDec // es un tipo o una var
        si existeID(RDECS_0.tsph, DEC.lexema) entonces
            DECS.errorDec = TRUE
        Si no
            RDECS_1.tsph= añadeID(RDECS_0.tsph,DEC.lexema,
                DEC.props,DEC.clase, DEC.nivel, DEC.dirH)
            Si DEC.clase = "tipo"
                and existePendientes(DEC.pend,DEC.lexema)
            Entonces
                RDECS_1.pendH=eliminaPendientes(DEC.pend,
                    DEC.lexema)
            Si no
                RDECS.pendH = DEC.pend
        Fin si
    Fin si
    DECS.pend = RDECS.pend
    RDECS_0.pend = RDECS_1.pend
    RDECS_0.etq = RDECS_1.etq
    RDECS_0.callPendH = RDECS_1.callPend

```

```

RDECS ::= λ
    RDECS.pend = RDECS.pendH
    RDECS.etq = RDECS.etqh
    RDECS.errorDec = FALSE
    RDECS.ts = RDECS.tsh
    RDECS.dir = RDECS.dirH
    RDECS.callPendH = RDECS.callPend

```

La producción hay que adaptarla quitando recursión a izquierdas:

```
SENTS ::= SENTS ; SENT
Queda transformada en:
SENTS ::= SENT RSENTS
    SENT.etqh = RSENTS.etq
    SENT.tsh = SENTS.tsh
    SENT.callPendH = SENTS_0.callPendh
    RSENTS.callPendH = SENT.callPend
    RSENTS.tsh = SENTS.tsh
    RSENTS.etqh = SENTS.etqh
    SENTS.etq = SENT.etq
    SENTS.errorSent = SENT.errorSent OR RSENTS.errorSent
    SENTS.cod = SENT.cod || RSENTS.cod
    SENTS.callPend = RSENTS.callPend

RSENTS ::= ; SENT RSENTS      SENT.etqh = RSENTS_0.etq
    SENT.tsh = SENTS.tsh
    SENT.callPendH = RSENTS_0.callPendh
    RSENTS_1.callPendH = SENT.callPend
    RSENTS_1.etqh = SENT.etq
    RSENTS_1.tsh = RSENTS_0.tsh
    RSENTS_0.etq = RSENT.etqh
    RSENTS_0.errorSent = SENT.errorSent OR RSENTS_0.errorSent
    RSENTS_0.cod = SENT.cod || RSENTS_1.cod
    RSENTS_0.callPend = RSENTS_1.callPend
RSENTS ::= λ
    RSENTS.callPend = RSENTS.callPendh
    RSENTS.etq = RSENTS.etqh
    RSENTS.errorSent = FALSE
    RSENTS.cod = "cadena vacia"
```

Las dos siguientes producciones empiezan por las mismas producciones no finales, es necesario factorizar

```
EXP ::= EXP1 OP0 EXP1
EXP ::= EXP1
```

El resultado es:

```
EXP ::= EXP1 REXP
    EXP1.tsh = EXP.tsh
    EXP1.etqh = EXP.etqh
    REXP.etqh = EXP1.etq
    REXP.tsh = EXP.tsh
    REXP.tipoH = EXP1.tipo
    REXP.modoh = EXP1.modoh
    REXP.codH = EXP1.cod
    EXP.modoh = REXP.modoh
    EXP.tipo = REXP.tipo
    EXP.cod = REXP.cod

REXP ::= OP0 EXP1
    EXP.modoh = val
    nIns = 0
    Ins = "cadena vacia"
    SI (REXP.tipoH = tError)
    ENTONCES
```

```

        EXP.cod = "Cadena vacía"
SI NO Si REXP.modoH = var
    nIns = nIns+1
    Ins = Ins || REXP.codH || apila_ind()
Si no
    Ins = Ins || REXP.codH
Fin si
EXP1.tsh = REXP.tsh
EXP1.etqh = REXP.etqh + nIns
nIns = 0
Si EXP1.modo = var
    nIns = nIns +1
    Ins = Ins || EXP1.cod || apila_ind()
Si no
    Ins = Ins || EXP1.cod
Fin si
EXP.cod = Ins || OP0.op
nIns = nIns +1
REXP.etq = EXP1.etq +nIns
REXP.etq = EXP1.etq + 1
REXP.tipo = dameTipo(REXP.tipoH, EXP1.tipo, OP0.op)

REXP ::= λ


---


    REXP.modo = REXP.modoH
    REXP1.etq = REXP1.etqh
    REXP.cod = REXP.codH
    REXP.tipo = REXP.tipoH

```

La producción hay que adaptarla quitando recursión a izquierdas:

```
EXP1 ::= EXP1 OP1 EXP2
```

El resultado es:

```
EXP1 ::= EXP2 REXP1
    EXP2.tsh = EXP1.tsh
    EXP2.etqh = EXP1.etqh
    REXP1.modoH = EXP2.modo
    REXP1.etqh = EXP2.etq
    REXP1.tsh = EXP1.tsh
    EXP1.modo = REXP1.modo
    SI (EXP2.tipo = tError)
    ENTONCES
        EXP1.tipo = tError
        EXP1.cod = "Cadena vacía"
    SI NO
        REXP1.tipoH = EXP2.tipo
        REXP1.codH = EXP2.cod
        EXP1.tipo = REXP1.tipo
        EXP1.cod = REXP1.cod
    FIN SI

REXP1 ::= OP1 EXP2 REXP1
    REXP1.modo = val
    nIns = 0
    Ins = "cadena vacia"
    SI (REXP1_0.tipoH = tError)
    ENTONCES
        REXP1_0.cod = "Cadena vacía"
    SI NO
        Si REXP1_0.modoH = var
            nIns = nIns+1
            Ins = Ins || REXP1_0.codH || apila_ind()
        Si no
            Ins = Ins || REXP1_0.codH
        Fin si
        si (OP1.op = oLogica)
            Ins = Ins + 3
            Ins = Ins || copia || ir_v(EXP2.etq) || desalipa
        Sin si
            EXP2.tsh = REXP1_0.tsh
        EXP2.etqh = REXP1_0.etqh + nIns
        nIns = 0
        Si EXP2.modo = var
            nIns = nIns+1
            Ins = Ins || EXP2.cod || apila_ind()
        Si no
            Ins = Ins || EXP2.cod
        Fin si
        si not (OP1.op = oLogica)
            nIns = nIns +1
            Ins = Ins || Op1.op
        Fin si
        REXP1_1.etqh = EXP2.etq
        REXP1_1.tsh = EXP2_0.tsh
        REXP1_1.tipoH = dameTipo(REXP1_0.tipoH, EXP2.tipo, OP1.op)
        REXP1_1.modoH = val
        REXP1_1.codH = Ins
        REXP1_1.etqH = EXP2.etq + nIns
    FIN SI
    REXP1_0.cod = REXP1_1.cod
```

```
REXP1_0.etq = REXP1_1.etq
```

```
REXP1 ::= λ
  REXP1.mod0 = REXP1.mod0H
  REXP1.etq = REXP1.etqh
  REXP1.tipo = REXP1.tipoH
  REXP1.cod = REXP1.codH
```

La producción hay que adaptarla quitando recursión a izquierdas:

```
EXP2 ::= EXP2 OP2 EXP3
El resultado es:
EXP2 ::= EXP3 REXP2
  REXP3.mod0 = EXP3.mod0H
  EXP2.mod0 = REXP2.mod0
  EXP3.tsh = EXP2.tsh
  REXP2.tsh = EXP2.tsh
  EXP3.etqh = EXP2.etqh
  REXP2.etqh = EXP3.etq
  SI (EXP3.tipo = tError OR REXP2.tipo = tError)
  ENTONCES
    EXP2.tipo = tError
    EXP2.cod = "Cadena vacía"
  SI NO
    REXP2.tipoH = EXP3.tipo
    REXP2.codH = EXP3.cod
    EXP2.tipo = REXP2.tipo
    EXP2.cod = REXP2.cod
  FIN SI

REXP2 ::= OP2 EXP3 REXP2
  REXP2_0.mod0 = val
  REXP2_1.mod0H = val
  EXP3.tsh = REXP2_0.tsh
  REXP2_1.tsh = EXP3_0.tsh
  REXP2_1.tipoH = dameTipo(REXP2_0.tipoH, EXP3.tipo, OP2.op)
  REXP2_0.tipo = REXP2_1.tipo
  OP2.finExpresion = REXP2_1.etq
  EXP3.etqh = REXP2_0.etqh + 1
  REXP2_1.etqh = EXP3.etq
  REXP2_0.etq = REXP2_1.etq
  REXP2_1.mod0H = val
  SI (REXP2_0.tipo = tError)
  ENTONCES
    REXP2_0.cod = "Cadena vacía"
  SI NO si (OP2.op = yLogica)
    nIns = 0
    Ins = "cadena vacía"
    Si REXP2_0.mod0H = var
      nIns = nIns+1
      Ins = Ins || REXP2_0.codH || apila_ind()
    Si no
      Ins = Ins || REXP2_0.codH
  Fin si

  Ins = Ins || ir_f(EXP3.etq+2)
  Ins = Ins + 1
  EXP3.etqh = REXP2_0.etqh + Ins
  nIns = 0
  Si EXP3.mod0 = var
    nIns = nIns+1
```

```

        Ins = Ins || EXP3.cod || apila_ind()
Si no
    Ins = Ins || EXP3.cod
Fin si
REXP2_1.codH = Ins || ir_f(EXP3.etq+2) || apila(false)
REXP2_1.etqh = EXP3.etq + nIns + 2
si no
    nIns = 0
    Ins = "cadena vacia"
    Si REXP2_0.modoh = var
        nIns = nIns+1
        Ins = Ins || EXP2_0.codH || apila_ind()
    Si no
        Ins = Ins || EXP2_0.codH
    Fin si
    Si EXP3.modoh = var
        nIns = nIns +1
        Ins = Ins || EXP3.cod || apila_ind()
    Si no
        Ins = Ins || EXP3.cod
    Fin si
    REXP2_1.codH = Ins || OP2.op
    nIns = nIns +1
    REXP2_1.etqh = EXP3.etq + nIns
FIN SI
REXP1_0.etq = EXP1_1.etq
REXP1_0.cod = REXP1_1.cod

REXP2 ::= λ
    REXP2.modoh = REXP2.modoh
    REXP2.etq = REXP2.etqh
    REXP2.tipo = REXP2.tipoH
    REXP2.cod = REXP2.codH

```

Las dos siguientes producciones empiezan por las mismas producciones no finales, es necesario factorizar

```

EXP3 ::= EXP4 OP3 EXP3
EXP3 ::= EXP4
El resultado es:
EXP3 ::= EXP4 REXP3
    EXP4.tsh = REXP3.tsh = EXP3.tsh
    EXP4.etqh = REXP3.etqh
    REXP3.modoh = EXP4.modoh
    REXP3.etqh = EXP4.etq
    SI (EXP4.tipo = tError) ENTONCES
        EXP3.tipo = tError
        EXP3.cod = "Cadena vacía"
    SI NO
        REXP3.codH = EXP4.cod
        REXP3.tipoH = EXP4.tipo
        EXP3.cod = REXP3.cod
        EXP3.tipo = REXP3.tipo
        EXP3.modoh = REXP3.modoh
    FIN SI

REXP3 ::= OP3 EXP3
    EXP3.tsh = REXP3.tsh
    EXP3.etqh = REXP3.etqh

    SI (REXP3.tipoH = tError)

```



```

    REXP3.cod = "Cadena vacía"
SI NO
    nIns = 0
    Ins = "cadena vacia"
    Si REXP3.modoH = var
        nIns = nIns+1
        Ins = Ins || REXP3.codH || apila_ind()
    Si no
        Ins = Ins || REXP3.codH
    Fin si
    EXP3.etqh = REXP3.etq + nIns
    nIns = 0

    Si EXP3.modo = var
        nIns = nIns +1
        Ins = Ins || EXP3.cod || apila_ind()
    Si no
        Ins = Ins || EXP3.cod
    Fin si
    REXP3.cod = Ins || OP3.op
    nIns = nIns +1
FIN SI
REXP3.etq = EXP3.etq + nIns
REXP3.modo = val
REXP3.etq = EXP3.etq + 1
REXP3.tipo = dameTipo(REXP3.tipoH, EXP3.tipo, OP3.op)

REXP3 ::= λ
    REXP3.modo = REXP3.modoH
    REXP3.etq = REXP3.etqh
    REXP3.cod = REXP3.codH
    REXP3.tipo = REXP3.tipoH

```

8 Esquema de traducción orientado a las gramáticas de atributos

```
PROGRAMA ::=
    DECS.pendH = creaPendientes()
    DECS.tsph = creaTS()
    DECS.niv = 0
    DECS.etqh = 0
    DECS.callPendH = creaCallPends()
    {DECS}
    SENTs.callPendH = DECS.callPendH
    SENTs.etqh = DECS.etq
    SENTs.tsh = DECS.ts
    {&}
    {SENTs}
    PROGRAMA.error = DECS.errorDec OR SENTs.errorSent OR
        DECS.pend.size <> 0 or SENTs.callPend.size <> 0
    PROGRAMA.cod = DECS.cod || SENTs.cod || stop

PROGRAMA ::=
    SENTs.tsph = creaTS()
    SENTs.callPendH = creaCallPends()
    {&}
    {SENTs}
    PROGRAMA.error = SENTs.errorSent or SENTs.callPend.size <> 0
    PROGRAMA.cod = SENTs.cod || stop

DECS ::=

    DEC.etqH = DECS.etqH
    DEC.dirH = DECS.dirH
    DEC.nivel = DECS.nivel
    DEC.tsph = DECS.tsph
    DEC.pendH = RDECS_0.pendH
    DEC.callPendH = DECS.callPendH
    {DEC}
    RDECS.callPendH = DEC.callPend
    RDECS.etqH = DEC.etq
    RDECS.dirH = DEC.dir
    RDECS.tsph = DEC.tsph
    RDECS.nivel = DECS.nivel
    DECS.errorDec = DEC.errorDec or RDECS.errorDec
    si not DECS.errorDec and DEC.clase = procedimiento entonces
        si not DEC.forward and entonces
            si existeID(DECS.tsph, DEC.lexema) and
                not (pendiente(DEC.pend, DEC.lexema) or
                    not tipoPendiente(DEC.pend, DEC.lexema)=proc)
            entonces
                DECS.errorSent = TRUE
            Si no si pendiente(DEC.pend, DEC.lexema)
                RDECS.pendH = eliminaPendiente(DEC.pend,
                    DEC.lexema)
                DECS.errorDecs = compruebaCampos(DEC.tsph,
                    DEC.lexema, DEC.props, DEC.clase,
                    DEC.nivel, DEC.etqH)
```

```

        RDECS.tsph = actualizaID(DEC.tsph,
                                DEC.lexema, DEC.etqH)
    Si no // no es un tipo "proc" de pendiente
        DECS.errorSent = TRUE
    fin si
    si no si existeID(RDECS_0.tsph, DEC.lexema) entonces
        DECS.errorSent = TRUE
    Si no // no existe ID y es forward
        RDECS.tsph = añadeID(DEC.tsph, DEC.lexema,
                            DEC.props, DEC.clase, DEC.nivel, DEC.etqH)
        RDECS.pendH = añadePendientes(DEC.pend,
                                    DEC.lexema, proc)
    fin si
    si not DECS.errorDec // es un tipo o una var
    entonces
        si existeID(DEC.tsph, DEC.lexema) entonces
            DECS.errorDec = TRUE
        Si no
            RDECS.tsph = añadeID(DEC.tsph, DEC.lexema, DEC.props,
                                DEC.clase, DEC.nivel, DEC.dirH)
            Si DEC.clase = "tipo"
                and existePendientes(DEC.pend, DEC.lexema)
            Entonces
                RDECS.pendH = eliminaPendientes(DEC.pend,
                                                DEC.lexema)
            Si no
                RDECS.pendH = DEC.pend
        Fin si
    Fin si
    {RDECS}
    DECS.pend = RDECS.pend
    DECS.pend = RDECS.pend
    DECS.etq = RDECS.etq
    DECS.callPend = RDECS.callPend

```

```

RDECS ::= ;
DEC.etqH = RDECS_0.etqH
DEC.tsph = RDECS_0.tsph
DEC.dirH = RDECS_0.dirH
DEC.nivel = RDECS_0.nivel
DEC.pendH = RDECS_0.pendH
DEC.callPendH = RDECS.callPendH
{DEC}
RDECS_1.callPendH = DEC.callPend
RDECS_1.etqH = DEC.etq
RDECS_1.dirH = DEC.dir
RDECS_1.tsph = DEC.tsph
RDECS_1.nivel = RDECS_0.nivel
RDECS_0.errorDec = DEC.errorDec or RDECS_1.errorDec
si not RDECS_0.errorDec and DEC.clase = procedimiento entonces
    si not DEC.forward and entonces
        si existeID(RDECS_0.tsph, DEC.lexema) and
            not (pendiente(DEC.pend, DEC.lexema) or
                not tipoPendiente(DEC.pend, DEC.lexema) = proc)
        entonces
            RDECS_0.errorSent = TRUE
        Si no si pendiente(DEC.pend, DEC.lexema)
            RDECS_0.errorDecs = compruebaCampos(DEC.tsph,

```

```

        DEC.lexema, DEC.props, DEC.clase,
        DEC.nivel, DEC.etqH)
    RDECS_1.pendH = eliminaPendiente(DEC.pend,
        DEC.lexema)
    RDECS_1.tsph = actualizaID(DEC.tsph,
        DEC.lexema, DEC.etqH)
    Si no // no es un tipo "proc" de pendiente
        RDECS_0.errorSent = TRUE
    si no si existeID(RDECS_0.tsph, DEC.lexema) entonces
        DECS_0.errorSent = TRUE
    Si no // no existe ID y es forward
        RDECS_1.tsph = añadeID(RDECS_0.tsph, DEC.lexema,
            DEC.props, DEC.clase, DEC.nivel, DEC.etqH)
        RDECS_1.pendH = añadePendientes(DEC.pend,
            DEC.lexema, proc)
    fin si
si not DECS_0.errorDec // es un tipo o una var
    si existeID(RDECS_0.tsph, DEC.lexema) entonces
        DECS.errorDec = TRUE
    Si no
        RDECS_1.tsph = añadeID(RDECS_0.tsph, DEC.lexema,
            DEC.props, DEC.clase, DEC.nivel, DEC.dirH)
        Si DEC.clase = "tipo"
            and existePendientes(DEC.pend, DEC.lexema)
        Entonces
            RDECS_1.pendH = eliminaPendientes(DEC.pend,
                DEC.lexema)
        Si no
            RDECS.pendH = DEC.pend
    Fin si
Fin si
{RDECS}
RDECS_0.pend = RDECS_1.pend
RDECS_0.pend = RDECS_1.pend
RDECS_0.etq = RDECS_1.etq
RDECS_0.callPendH = RDECS_1.callPend

```

```

RDECS ::=
    RDECS.pend = RDECS.pendH
    RDECS.etq = RDECS.etqh
    RDECS.errorDec = FALSE
    RDECS.ts = RDECS.tsh
    RDECS.dir = RDECS.dirH
    RDECS.callPendH = RDECS.callPend

```

```

DEC ≡
    DECVAR.tsph = DEC.tsph
    DECVAR.pendH = DEC.pendH
    {DECVAR}
    DEC.pend = DECVAR.pend
    DEC.dir = DEC.dirh
    DEC.clase = "variable"
    DEC.lexema = DECVAR.lexema
    DEC.props = DECVAR.props
    DEC.errorDec = DECVAR.error
    DEC.etq = DEC.etqh
    DEC.cod = "cadena vacia"

```

```

DEC.decSize = DECVAR.decSize

DECVAR ≡
  {iden}
  TIPOIDEN.tsph = DECVAR.tsph
  TIPOIDEN.pendH = DECVAR.pendH
  {TIPOIDEN}
  DECVAR.pend = TIPOIDEN.pend
  DECVAR.lexema = iden.lexema
  DECVAR.props = TIPOIDEN.props
  DECVAR.error = existeID(DECVAR.tsph, iden.lexema) or
    TIPOIDEN.error
  DECVAR.decSize = TIPOIDEN.size

TIPOIDEN ≡
  {boolean}
  TIPOIDEN.pend = TIPOIDEN.pendH
  TIPOIDEN.error = FALSE
  TIPOIDEN.props = <t: boolean>
  TIPOIDEN.size = 1

TIPOIDEN ≡
  { character }
  TIPOIDEN.pend = TIPOIDEN.pendH
  TIPOIDEN.props = <t: character >
  TIPOIDEN.error = FALSE
  TIPOIDEN.size = 1

TIPOIDEN ≡
  { natural }
  TIPOIDEN.pend = TIPOIDEN.pendH
  TIPOIDEN.props = <t: natural >
  TIPOIDEN.error = FALSE
  TIPOIDEN.size = 1

TIPOIDEN ≡
  { integer }
  TIPOIDEN.pend = TIPOIDEN.pendH
  TIPOIDEN.props = <t: integer >
  TIPOIDEN.error = FALSE
  TIPOIDEN.size = 1

TIPOIDEN ≡
  { float }
  TIPOIDEN.pend = TIPOIDEN.pendH
  TIPOIDEN.props = <t: float >
  TIPOIDEN.error = FALSE
  TIPOIDEN.size = 1

TIPOIDEN ≡
  { iden }
  TIPOIDEN.pend = TIPOIDEN.pendH
  TIPOIDEN.props = <t:ref> ++ <id: iden.lexema>
  TIPOIDEN.error = not existeTipo(TIPOIDEN.tsph, iden.lexema)
  TIPOIDEN.size = damePropsTs(TIPOIDEN.tsph, iden.lexema).size

TIPOIDEN ≡
  {record}
  {/{}}
  CAMPOS.pendH = TIPOIDEN.pendH
  {CAMPOS}

```

```

    {/}}
    TIPOIDEN.props = <t:rec> ++ < campos: CAMPOS.props>
    TIPOIDEN.error = CAMPOS.error
    TIPOIDEN.pend = CAMPOS.pend

TIPOIDEN ≡
{pointer}
TIPOIDEN_1.pendH =TIPOIDEN_0.pendH
{TIPOIDEN}
TIPOIDEN_0.props = <t:puntero> ++ <tbase: ++ TIPOIDEN_1.props>
TIPOIDEN.size = 1
si TIPOIDEN_0.error and TIPOIDEN_1.tipo = ref and
    not existeID(TIPOIDEN_0.tsph, TIPOIDEN_1.props.id)
entonces
    TIPOIDEN_0.pend = añadePendiente(TIPOPENDIENTE_1.pend,
        TIPOIDEN_1.props.id)
    TIPOIDEN_0.error = FALSE
si no
    TIPOIDEN_0.pend = TIPOPENDIENTE_1.pend
Fin si

TIPOIDEN ≡
{array}
{[]}
{natural}
{[]}
{of}
TIPOIDEN_1.pendH =TIPOIDEN_0.pendH
{TIPOIDEN}
TIPOIDEN_0.props = <t:array> ++ <nelem: natural.valor> ++
    <tbase: TIPOIDEN_1.props>
TIPOIDEN_0.error = TIPOIDEN_1.error
TIPOIDEN_0.size = TIPOIDEN_1.size * natural.lexema
TIPOIDEN_0.pend =TIPOIDEN_1.pend

CAMPOS ≡
CAMPO.pendH = CAMPOS.pendH
CAMPO.tsph = CAMPOS.tsph
{CAMPO}
RCAMPOS.pendH = CAMPO.pend
RCAMPOS.tsph = CAMPOS.tsph
{RCAMPOS}
CAMPOS.props = CAMPO.props ++ RCAMPOS.props
CAMPOS.error = CAMPO.erro or RCAMPOS.error
CAMPOS.size = CAMPO.size + RCAMPOS.size
CAMPOS.pend = RCAMPOS.pend

RCAMPOS ≡
{;}
CAMPO.tsph = RCAMPOS_0.tsph
CAMPO.pendH = RCAMPOS_0.pendH
{CAMPO}
RCAMPOS_1.pendH = CAMPO.pend
RCAMPOS_1.tsph = RCAMPOS_0.tsph
{RCAMPOS}
RCAMPOS_0.props = CAMPO.props ++ RCAMPOS_1.props
RCAMPOS_0.error = CAMPO.erro or RCAMPOS_1.error
RCAMPOS_0.size = CAMPO.size + RCAMPOS_1.size
RCAMPOS_0.pend = RCAMPOS_1.pend

```

```

RCAMPOS ≡ lambda
  RCAMPOS.props = "cadena vacia"
  RCAMPOS.error = FALSE
  RCAMPOS.size = 0

CAMPO ≡ {iden} : {TIPOIDEN}
  TIPOIDEN.tsph = CAMPO.tsph
  TIPOIDEN.pendh = CAMPO.pend
  {TIPOIDEN}
  CAMPO.props = <id: iden.lexema> ++ <t: _TIPOIDEN.props.t>
  CAMPO.error = existeID(CAMPO.tsph, iden.lexema) or
    TIPOIDEN.error
  CAMPO.size = TIPOIDEN.size

DEC ≡

  DECTIP.tsph = DEC.tsph
  {DECTIP}
  DEC.decSize = 0
  DEC.dir = DEC.dirH
  DEC.etq = DEC.etqh
  DEC.cod = "cadena vacia"
  DEC.clase = "tipo"
  DEC.lexema = DECTIP.lexema
  DEC.props = DECTIP.props
  DEC.errorDec = DECTIP.error

DECTIP ≡
  {tipo}
  {iden}
  {=}
  TIPOIDEN.tsph = DECTIP.tsph
  {TIPOIDEN}
  DECTIP.lexema = iden.lexema
  DECTIP.props = TIPOIDEN.props
  DECTIP.error = existeID(DECTIP.tsph, iden.lexema) or
    TIPOIDEN.error

DEC ≡
  DECPROC.etqh = DEC.etqh
  DECPROC.tsph=DEC.tsph
  DECPROC.nivelH = DEC.nivelH
  DECPROC.callPendH = DEC.callPend
  {DECPROC}
  DEC.callPend = DECPROC.callPend
  DEC.clase = "procedimiento"
  DEC.lexema = DECPROC.lexema
  DEC.props = DECPROC.props
  DEC.errorDec = DECPROC.error
  DEC.etq = DECPROC.etq
  DEC.cod = DECPROC.cod
  DEC.nivel = DECPROC.nivel

DECPROC ≡
  {proc}
  {iden}
  DPARAMS.tsph = creaTS(DECPROC.tsph)
  DPARAMS.dirH = 2
  DPARAMS.nivelh= DECPROC.nivel+1

```

```

{DPARAMS}
PBLOQUE.nivelh=DECPROC.nivel+1
PBLOQUE.tsph = añadeID(DPARAMS.tsp, DECPROC.lexema,
    DECPROC.props, DECPROC.nivel+1)
PBLOQUE.dirH = DPARAMS.dir
PBLOQUE.etqh = DECPROC.etqh
PBLOQUE.callPendH = DECPROC.callPendH
{PBLOQUE}
DECPROC.callPend = PBLOQUE.callPend
DECPROC.lexema = iden.lexema
DECPROC.clase = "procedimiento"
DECPROC.params = <t:proc>+<params:DPARAMS.params>
DECPROC.cod = PBLOQUE.cod

```

```

PBLOQUE ≡
{forward}
PBLOQUE.ts = PBLOQUE.tsh
PBLOQUE.error = FALSE
PBLOQUE.etq = PBLOQUE.etqh
PBLOQUE.forward = TRUE
PBLOQUE.callPendH = PBLOQUE.callPend

```

```

PBLOQUE ≡
{{}}
DECS.etqh = PBLOQUE.etqh
DECS.tsph=PBLOQUE.tsph
DECS.nivel=PBLOQUE.nivel
DECS.dirH = PBLQUE.dirH
{DECS}
{&}
SENT.etqh = DECS.etq + longPrologo
SENT.tsh= DECS.tsp
{SENTS}
PBLOQUE.ts=DECS.tsp
PBLOQUE.error = SENT.errorSent or DECS.errorDec
PBLOQUE.etq = SENT.etq + longEpligo
PBLOQUE.forward = FALSE
PBLOQUE.cod = prologo() || DECS.cod || SENTS.cod || epilgo()
PBLOQUE.callPend = resolverPend(PBLOQUE.tsph, SENTS.callPend)
{}}

```

```

PBLOQUE ≡
{{}} {&}
SENTS.etqh = PBLOQUE.etqh +longPrologo
SENTS.ts= PBLOQUE.tsph
SENTS.callPendH = PBLOQUE.callPendH
{SENTS}
PBLQUE.error = SENT.errorSent
PBLOQUE.etq = SENT.etq + longEpligo
PBLOQUE.forward = FALSE
PBLOQUE.cod = prologo() || SENTS.cod || epilgo()
PBLOQUE.callPend = resolverPend(PBLOQUE.tsph, SENTS.callPend)
{}}

```

```

DPARAMS ≡
{()}
LISTAPARAMS .dirH = DPARAMS.dirH
LISTAPARAMS.tsph= DPARAMS.tsph
LISTAPARAMS.nivelh = DPARAMS.nivelh

```



```

    {LISTAPARAMS}
    DPARAMS.size = LISTAPARAMS.size
    DPARAMS.param=LISTAPARAMS.param
    DPARAMS.tsp = LISTAPARAMS.tsp
    DPARAMS.error = LISTAPARAMS.error
    {}

DPARAMS ≡
    DPARAMS.size = 0
    DPARAMS.props = {}
    DPARAMS.tsp= DPARAMS.tsph
    DPARAMS.error = FALSE

LISTAPARAMS ::=
    PARAM.dirH = LISTAPARAMS.dirH
    {PARAM}      RLISTAPARAMS.dirH = PARAM.dir
    RLISTAPARAMS.paramH = PARAM.param
    RLISTAPARAMS.nivelH = LISTAPARAMS.nivelH
    si not PARAM.error entonces
        RLISTAPARAMS.tsph = añadeID(LISTAPARAMS.tsph,
            PARAM.lexema,PARAM.props, PARAM.clase,
            LISTAPARAMS.nivelh, LISTAPARAMS.dirH)
        RLISTAPARAMS.dirH = LISTAPARAMS.dirH + PARAM.size
    fin si
    {RLISTAPARAMS}
    LISTAPARAMS.error = RLISTAPARAMS.error or PARAM.error
    LISTAPARAMS.dir = RLISTAPARAMS.dir
    LISTAPARAMS.param = RLISTAPARAMS.param

RLISTAPARAMS ::=
    {,}
    PARAM.dirH = RLISTAPARAMS_0.dirH
    {PARAM}
    RLISTAPARAMS_1.dirH = PARAM.dir
    RLISTAPARAMS_1.paramH = RLISTAPARAMS_0.paramH ++ PARAM.param
    RLISTAPARAMS_1.nivelH = RLISTAPARAMS_0.nivelH
    RLISTAPARAMS_0.error = RLISTAPARAMS_1.error or PARAM.error
    si not PARAM.error entonces
        RLISTAPARAMS_1.tsph = añadeID(RLISTAPARAMS_0.tsph,
            PARAM.lexema,PARAM.props, PARAM.clase,
            RLISTAPARAMS_0.nivelh, LISTAPARAMS.dirH)
        RLISTAPARAMS_1.dirH = RLISTAPARAMS_0.dirH + PARAM.size
    fin si
    {RLISTAPARAMS}
    RLISTAPARAMS_0.error = RLISTAPARAMS_1.error or PARAM.error
    RLISTAPARAMS_0.dir = RLISTAPARAMS_1.dir
    RLISTAPARAMS_0.param = RLISTAPARAMS_1.param

RLISTAPARAMS ::=
    RLISTAPARAMS.error = FALSE
    RLISTAPARAMS.dir = RLISTAPARAMS.dirH
    RLISTAPARAMS.param = RLISTAPARAMS.paramH

PARAM ≡
    {var}
    {iden}
    {:}
    TIPOIDEN.tsph = PARAM.tsph
    {TIPOIDEN}
    PARAM.clase = "p_variable"
    PARAM.lexema = iden.lexema

```

```

PARAM.props = <t: TIPOIDEN.props>
PARAM.param = <modo : variable> <t: TIPOIDEN.props.t>
PARAM.size = 1

PARAM ≡
{iden}
{:}
TIPOIDEN.tsph = PARAM.tsph
{TIPOIDEN}
PARAM.id= iden.lexema
PARAM.clase = "variable"
PARAM.props = <t: TIPOIDEN.props.t>
PARAM.param = <modo:valor, t: TIPOIDEN.props>
PARAM.error = existeId(PARAM.tsph, iden.lexema) or TIPOIDEN.error
PARAM.size = TIPOIDEN.size

SENTS ::= SENT RSENTS
SENT.etqh = RSENTS.etq
SENT.tsh = SENTS.tsh
SENT.callPendH = SENTS_0.callPendh
{SENT}
RSENTS.callPendH = SENT.callPend
RSENTS.tsh = SENTS.tsh
RSENTS.etqh = SENTS.etqh
{RSENT}
SENTS.etq = SENT.etq
SENTS.errorSent = SENT.errorSent OR RSENTS.errorSent
SENTS.cod = SENT.cod || RSENTS.cod
SENTS.callPend = RSENTS.callPend

RSENTS ::=
{;}
SENT.etqh = RSENTS_0.etq
SENT.tsh = SENTS.tsh
SENT.callPendH = RSENTES_0.callPendh
{SENT}
RSENTS_1.callPendH = SENT.callPend
RSENTS_1.etqh = SENT.etq
RSENTS_1.tsh = RSENTS_0.tsh
{RSENTS}
RSENTS_0.etq = RSENT.etqh
RSENTS_0.errorSent = SENT.errorSent OR RSENTS_0.errorSent
RSENTS_0.cod = SENT.cod || RSENTS_1.cod
RSENTS_0.callPend = RSENTS_1.callPend

RSENTS ::=
RSENTS.callPend = RSENTES.callPendh
RSENTS.etq = RSENTS.etqh
RSENTS.errorSent = FALSE
RSENTS.cod = "cadena vacia"

SENT ::=
SWRITE.tsh = SENT.tsh
SWRITE.etqh = SENT.etqh
{SWRITE}
SENT.errorSent = SWRITE.errorSent
SENT.etq = SWRITE.etq
SENT.cod = SWRITE.cod

SENT ::=
SREAD.tsh = SENT.tsh

```

```

    SREAD.etqh = SENT.etqh
    {SREAD}
    SENT.errorSent = SREAD.errorSent
    SENT.etq = SREAD.etq
    SENT.cod = SREAD.cod

SENT ::=
    SBLOQUE.tsh = SENT.tsh
    SBLOQUE.etqh = SENT.etqh
    {SBLOQUE}
    SENT.errorSent = SBLOQUE.errorSent
    SENT.etq = SBLOQUE.etq
    SENT.cod = SBLOQUE.cod

SENT ::=
    SIF.tsh = SENT.tsh
    SIF.etqh = SENT.etqh
    {SIF}
    SENT.errorSent = SIF.errorSent
    SENT.etq = SIF.etq
    SENT.cod = SIF.cod

SENT ::=
    SWHILE.tsh = SENT.tsh
    SWHILE.etqh = SENT.etqh
    {SWHILE}
    SENT.errorSent = SWHILE.errorSent
    SENT.etq = SWHILE.etq
    SENT.cod = SWHILE.cod

SENT ::=
    SFOR.tsh = SENT.tsh
    SFOR.etqh = SENT.etqh
    {SFOR}
    SENT.errorSent = SFOR.errorSent
    SENT.etq = SFOR.etq
    SENT.cod = SFOR.cod

SWRITE ::=
    {out}
    {()}
    EXP.tsh = SWRITE.tsh
    EXP.etqh = SWRITE.etqh
    {EXP}
    SWRITE.errorSent = (EXP.tipo = tError)
SI SWRITE.errorSent
    ENTONCES
        SWRITE.cod = "Cadena vacía"
    SI NO
        SWRITE.cod = EXP.cod || escribir
    FIN SI
    SWRITE.etq = EXP.etq + 1
    {}

SREAD ::=
    {in}
    {()}
    {iden}
    SREAD.errorSent = (NOT existeVar(SREAD.tsh, iden.lexema))
SI SREAD.errorSent
    ENTONCES

```

```

        SREAD.cod = "Cadena vacía"
SI NO
    SREAD.cod = leer ||
        desapila_dir(damePropiedadesTS(SREAD.tsh
                                        , iden.lexema).dirProp)
FIN SI
SREAD.etq = 2
{}

SBLOQUE ::=
    {}
    SENTS.tsh = SBLOQUE.tsh
    SENTS.etqh = SBLOQUE.etqh
    {SENTS}
    SBLOQUE.errorSent = SENTS.errorSent
    SBLOQUE.etq = SENTS.etq
    SBLOQUE.cod = SENTS.cod
    {}

SIF ::=
    {if}
    EXP.etqh = SIF.etqh
    EXP.tsh = SIF.tsh
    {EXP}
    {then}
    SENT.tsh = SIF.tsh
    SENT.etqh = EXP.etq + 1
    {SENT}
    PELSE.tsh = SIF.tsh
    PELSE.etqh = SENT.etq + 1
    {PELSE}
    SIF.errorSent = (EXP.tipo <> tBool) OR SENT.errorSent OR
        PELSE.errorSent
    SIF.cod = EXP.cod || ir-f(SENT.etq + 1) || SENT.cod ||
        ir-a(PELSE.etq) || PELSE.cod
    SIF.etq = PELSE.etq

PELSE ::=
    {else}
    SENT.tsh = PELSE.tsh
    SENT.etqh = PELSE.etqh
    {SENT}
    PELSE.errorSent = SENT.errorSent
    PELSE.etq = SENT.etq
    PELSE.cod = SENT.cod

PELSE ::= lambdaλ
    PELSE.errorSent = FALSE
    PELSE.cod = λ
    PELSE.etq = PELSE.etqh

SWHILE ::=
    {while}
    EXP.tsh = SWHILE.tsh
    EXP.etqh = SWHILE.etqh
    {EXP}
    {do}
    SENT.etqh = EXP.etq + 1
    SENT.etq = EXP.etq
    {SENT}

```

```

SWHILE.errorSent = (EXP.tipo <> tBool) OR SENT.errorSent
SWHILE.cod = EXP.cod || ir-f(SENT.etq + 1) || SENT.cod
           || ir-a(SWHILE.etqh)
SWHILE.etq = SENT.etq + 1

SFOR ::=
{for}
MEM.tsh = SFOR.tsh
MEM.etqh = SFOR.etqh
{MEM}
EXP_0.tsh = SFOR.tsh
EXP_0.etqh = MEM.etq
{EXP}
{to}
EXP_1.tsh = SFOR.tsh
EXP_1.etqh = EXP_0.etq + 2
{EXP}
{do}
SENT.tsh = SFOR.tsh
SENT.etqh = EXP_1.etq + 2
{SENT}
SFOR.errorSent = SENT.errorSent
           OR (NOT existeID(SFOR.tsh, iden.lexema))
           OR (NOT ((EXP_0.tipo = tNat) OR (EXP_0.tipo = tInt)))
           OR (NOT ((EXP_1.tipo = tNat) OR (EXP_1.tipo = tInt)))
           OR (NOT esCompatibleAsig?(dameTipoTS(SFOR.tsh,
           iden.lexema), EXP_0.tipo))
           OR (NOT esCompatibleAsig?(dameTipoTS(SFOR.tsh,
           iden.lexema), EXP_1.tipo))
SFOR.cod = EXP_0.cod
           || desapila_dir(damePropiedadesTS(SFOR.tsh,
           iden.lexema).dirProp)
           || apila_dir(damePropiedadesTS(SFOR.tsh,
           iden.lexema).dirProp)
           || EXP_1.cod || menorIgual || ir-f(SENT.etq + 5) || SENT.cod
           || apila(damePropiedadesTS(SFOR.tsh, iden.lexema).dirProp)
           || apila(1) || suma
           || desapila_dir(damePropiedadesTS(SFOR.tsh,
           iden.lexema).dirProp)
           || ir-a(SFOR.etqh)
SFOR.etq = SENT.etq + 5

SNEW ≡
{new}
MEM.etqh = SNEW.etqh
MEM.tsh = SNEW.tsh
{MEM}
Si not Mem.tipo = puntero entonces
           SNEW.errorSent = TRUE
Si no
           si MEM.tipo.tBase = ref entonces
               Tam = dameProps(SNEW.tsh, MEM.tipo.tBase.id).tam
           Si no
               Tam = dameProps(SNEW.tsh, MEM.tipo.tBase).tam
           SNEW.cod = Mem.cod || new(tam)
           Desapila_ind()
           SNEW.etq = MEM.etq + 2

SDEL ≡
{dispose}
MEM.etqh = SDEL.etqh

```

```

MEM.tsh = SDEL.tsh
{mem}
Si not Mem.tipo = puntero entonces
    SDEL.errorSent = TRUE
Si no
    si MEM.tipo.tBase = ref entonces
        Tam = dameProps(SDEL.tsh, MEM.tipo.tBase.id).tam
    Si no
        Tam = dameProps(SDEL.tsh, MEM.tipo.tBase).tam
    Fin si
    SNEW.cod = Mem.cod || del(tam)
    SNEW.etq = MEM.etq + 1
Fin si

```

```

SENT ::=
    {iden}
    RSENT.tsh = MEM.tsh
    RSENT.tipoh = dameTipo(SENT.tsh, iden.lexema)
    RSENT.tipo = RMEM.tipo
    RSENT.iden = iden.lexema
    RSENT.etqh = SENT.etqh
    RSENT.callPendh= SENT.callPendh
    {RSENT}
    SENT.etq = RSENT.etq
    SENT.errorSent = RSENT.errorSent
    SENT.cod = RSENT.cod
    SENT.callPend = RSENT.callPend

RSENT ::=
    {RMEM}
    {:=}
    RMEM.tipoh = RSENT.tipoh
    RMEM.etqh = EXP .etqh +1
    RMEM.tsh = RSENT.tsh
    Si damePropsTS(RSENT.tsh, RSENT.iden).clase = pvar entonces
        codTemp = apila_ind()
        nCodTemp = 1
    si no
        nCodTemp = 0
    fin si
    EXP.etqh = RSENT.etqh
    {EXP}
    RSENT.errorSent = errorTipos(RMEM.tipo, EXP.tipo)
    si not RSENT.errorSent and EXP.modos = val entonces
        RSENT.cod= codTemp
        || EXP.cod
        || apila(damePropiedadesTS(RSENT.tsh, iden.lexema).dir)
        || RMEM.cod
        || desapila_ind()
        //copia mueve lo que hay en lo 2° a lo 1°

Si not si RSENT.errorSent and EXP.modos = var entonces
    size = dameSize(RSENT.tsh, RMEM.tipo)
    RSENT.cod= codTemp
    || EXP.cod
    || apila(damePropiedadesTS(RSENT.tsh, iden.lexema).dir)
    || RMEM.cod

```

```

|| mueve(size)
|| copia mueve lo que hay en lo 2° a lo 1°
Fin si
RSENT.etq = RMEM.etq+2+nCodTemp

RSENT ::=
PPARAMS.nivel = RSENT.nivel
PPARAMS.tsh = RSENT.tsh
PPARAMS.procName = RSENT.iden
{PPARAMS}
RSENT.errorSent = PPARAMS.error
dirProc = dameDir(PPARAMS.tsh, RSENT.iden)
RSENT.cod = apila-ret(RSENT.etqh) ||
|| PPARAMS.cod
|| ir_a(dirProc) \\ la dirección esta en la pila
|| // codigo de postllamada
si dirProc = -1 entonces
RSENT.callPend = añadeCallPend(RSENT.callPendH, RSENT.iden,
PPARAMS.etq + 1)
RSENT.etq = PPARAMS.etq + longApilaRet + 1

EXP ::=
EXP1.tsh = EXP.tsh
EXP1.etqh = EXP.etqh
{EXP1}
REXP.etqh = EXP1.etq
REXP.tsh = EXP.tsh
REXP.tipoH = EXP1.tipo
REXP.modoh = EXP1.modoh
REXP.codH = EXP1.cod
{REXP}
EXP.modoh = REXP.modoh
EXP.tipo = REXP.tipo
EXP.cod = REXP.cod

REXP ::= OP0 EXP1
EXP.modoh = val
nIns = 0
Ins = "cadena vacia"
SI (REXP.tipoH = tError)
ENTONCES
EXP.cod = "Cadena vacía"
SI NO
Si REXP.modoh = var
nIns = nIns+1
Ins = Ins || REXP.codH || apila_ind()
Si no
Ins = Ins || REXP.codH
Fin si
{OP}
EXP1.tsh = REXP.tsh
EXP1.etqh = REXP.etqh + nIns
{EXP1}
nIns = 0
Si EXP1.modoh = var
nIns = nIns +1
Ins = Ins || EXP1.cod || apila_ind()
Si no
Ins = Ins || EXP1.cod
Fin si

```

```

        EXP.cod = Ins || OP0.op
        nIns = nIns +1
        REXP.etq = EXP1.etq +nIns
        REXP.etq = EXP1.etq + 1
        REXP.tipo = dameTipo(REXP.tipoH, EXP1.tipo, OP0.op)
Fin si

REXP ::=
    REXP.modoh = REXP.modoh
    REXP1.etqh = REXP1.etqh
    REXP.cod = REXP.codH
    REXP.tipo = REXP.tipoH

EXP1 ::=
    EXP2.tsh = EXP1.tsh
    EXP2.etqh = EXP1.etqh
    {EXP2}
    REXP1.modoh = EXP2.modoh
    REXP1.etqh = EXP2.etq
    REXP1.tsh = EXP1.tsh
    SI (EXP2.tipo = tError)
    ENTONCES
        EXP1.tipo = tError
        EXP1.cod = "Cadena vacía"
    SI NO
        {REXP1}
        REXP1.tipoH = EXP2.tipo
        REXP1.codH = EXP2.cod
        EXP1.tipo = REXP1.tipo
        EXP1.cod = REXP1.cod
        EXP1.modoh = REXP1.modoh
    FIN SI

REXP1 ::=
    {OP1}
    REXP1.modoh = val
    nIns = 0
    Ins = "cadena vacia"
    SI (REXP1_0.tipoH = tError)
    ENTONCES
        REXP1_0.cod = "Cadena vacía"
    SI NO
        Si REXP1_0.modoh = var
            nIns = nIns+1
            Ins = Ins || REXP1_0.codH || apila_ind()
        Si no
            Ins = Ins || REXP1_0.codH
        Fin si
        si (OP1.op = oLogica)
            Ins = Ins + 3
            Ins = Ins || copia || ir_v(EXP2.etq) || desalipa
        Sin si
            EXP2.tsh = REXP1_0.tsh
        EXP2.etqh = REXP1_0.etqh + nIns
        {EXP2}
        nIns = 0
        Si EXP2.modoh = var
            nIns = nIns+1
            Ins = Ins || EXP2.cod || apila_ind()
        Si no

```



```

        Ins = Ins || EXP2.cod
    Fin si
    si not (OP1.op = oLogica)
        nIns = nIns +1
        Ins = Ins || Op1.op
    Fin si
    REXP1_1.etqh = EXP2.etq
    REXP1_1.tsh = EXP2_0.tsh
    REXP1_1.tipoH = dameTipo(REXP1_0.tipoH, EXP2.tipo, OP1.op)
    REXP1_1.modoh = val
    REXP1_1.codH = Ins
    REXP1_1.etqH = EXP2.etq + nIns
    {REXP1}
FIN SI
REXP1_0.cod = REXP1_1.cod
REXP1_0.etq = REXP1_1.etq

REXP1 ::=
    REXP1.modoh = REXP1.modoh
    REXP1.etq = REXP1.etqh
    REXP1.tipo = REXP1.tipoH
    REXP1.cod = REXP1.codH

EXP2 ::=
    EXP3.tsh = EXP2.tsh
    EXP3.etqh = EXP2.etqh
    {EXP3}
    REXP2.modoh = EXP3.modoh
    REXP2.etqh = EXP3.etq
    REXP2.tsh = EXP2.tsh
    SI (EXP3.tipo = tError)
    ENTONCES
        EXP2.tipo = tError
        EXP2.cod = "Cadena vacía"
    SI NO
        {REXP2}
        REXP2.tipoH = EXP3.tipo
        REXP2.codH = EXP3.cod
        EXP2.tipo = REXP2.tipo
        EXP2.cod = REXP2.cod
        EXP2.modoh = REXP2.modoh
    FIN SI

REXP2 ::=
    {OP2}
    REXP2.modoh = val
    nIns = 0
    Ins = "cadena vacia"
    SI (REXP2_0.tipoH = tError)
    ENTONCES
        REXP2_0.cod = "Cadena vacía"
    SI NO
        Si REXP2_0.modoh = var
            nIns = nIns+1
            Ins = Ins || REXP2_0.codH || apila_ind()
        Si no
            Ins = Ins || REXP2_0.codH
    Fin si

```

```

    si (OP2.op = yLogica)
        Ins = Ins || ir_f(EXP3.etq+1)
        Ins = Ins + 1
    Sin si
    EXP3.tsh = REXP2_0.tsh
    EXP3.etqh = REXP2_0.etqh + nIns
    {EXP3}
    nIns = 0
    Si EXP3.modo = var
        nIns = nIns+1
        Ins = Ins || EXP3.cod || apila_ind()
    Si no
        Ins = Ins || EXP3.cod
    Fin si
    si not (OP2.op = yLogica)
        nIns = nIns +1
        Ins = Ins || Op2.op
    Si no
        Ins = Ins || Ir_a(EXP3.etq+2) || apila(FALSE)
        nIns = nIns +2
    Fin si
    REXP2_1.etqh = EXP3.etq
    REXP2_1.tsh = EXP3_0.tsh
    REXP2_1.tipoH = dameTipo(REXP2_0.tipoH, EXP3.tipo, OP2.op)
    REXP2_1.modoH = val
    REXP2_1.codH = Ins
    REXP2_1.etqH = EXP3.etq + nIns
    {REXP2}
FIN SI
REXP2_0.cod = REXP2_1.cod
REXP2_0.etq = REXP2_1.etq

REXP2 ::=
    REXP2.modo = REXP2.modoH
    REXP2.etq = REXP2.etqh
    REXP2.tipo = REXP2.tipoH
    REXP2.cod = REXP2.codH

EXP3 ::=
    EXP4.tsh = REXP3.tsh = EXP3.tsh
    EXP4.etqh = REXP3.etqh
    {EXP4}
    REXP3.modoH = EXP4.modo
    REXP3.etqh = EXP4.etq
    SI (EXP4.tipo = tError) ENTONCES
        EXP3.tipo = tError
        EXP3.cod = "Cadena vacía"
    SI NO
        REXP3.codH = EXP4.cod
        REXP3.tipoH = EXP4.tipo
        {REXP3}
        EXP3.cod = REXP3.cod
        EXP3.tipo = REXP3.tipo
        EXP3.modo = REXP3.modo
    FIN SI

REXP3 ::=
    {OP3}
    EXP3.tsh = REXP3.tsh
    EXP3.etqh = REXP3.etqh

```

```

SI (REXP3.tipoH = tError)
    REXP3.cod = "Cadena vacía"
SI NO
    nIns = 0
    Ins = "cadena vacia"
    Si REXP3.modoh = var
        nIns = nIns+1
        Ins = Ins || REXP3.codH || apila_ind()
    Si no
        Ins = Ins || REXP3.codH
    Fin si
    EXP3.etqh = REXP3.etq + nIns
    nIns = 0
    {EXP3}
    Si EXP3.modoh = var
        nIns = nIns +1
        Ins = Ins || EXP3.cod || apila_ind()
    Si no
        Ins = Ins || EXP3.cod
    Fin si
    REXP3.cod = Ins || OP3.op
    nIns = nIns +1
FIN SI
REXP3.etq = EXP3.etq + nIns
REXP3.modoh = val
REXP3.etq = EXP3.etq + 1
REXP3.tipo = dameTipo(REXP3.tipoH, EXP3.tipo, OP3.op)

REXP3 ::=
    REXP3.modoh = REXP3.modoh
    REXP3.etq = REXP3.etqh
    REXP3.cod = REXP3.codH
    REXP3.tipo = REXP3.tipoH

EXP4 ::=
    {OP4_1}
    TERM.tsh = EXP4.tsh
    TERM.etqh = EXP4.etqh
    {TERM}
    EXP4.modoh = val
    EXP4.tipo = dameTipo(TERM.tipo, OP4_1.op)
    SI (EXP4.tipo = tError)
    ENTONCES
        EXP4.cod = "Cadena vacía"
    SI NO
        si TERM.modoh = var
            EXP4.cod = TERM.cod || apila_ind() || OP4_1.op
            EXP4.etq = TERM.etq + 2
        Si no
            EXP4.cod = TERM.cod || OP4_1.cod
            EXP4.etq = TERM.etq + 1
        FIN SI
    FIN SI

EXP4 ::=
    {}
    EXP4.modoh = val
    EXP4.tipo = dameTipo(TERM.tipo, "|")
    TERM.tsh = EXP4.tsh
    SI (EXP4.tipo = tError)
    ENTONCES

```

```

        EXP4.cod = "Cadena vacía"
SI NO
    {TERM}
    si TERM.modo = var
        EXP4.cod = TERM.cod || apila_ind() || valorAbs
        EXP4.etq = TERM.etq + 2
    Si no
        EXP4.cod = TERM.cod || valorAbs
        EXP4.etq = TERM.etq + 1
    FIN SI
FIN SI
TERM.etqh = EXP4.etqh
{||}

EXP4 ::=
    TERM.tsh = EXP4.tsh
    TERM.etqh = EXP4.etqh
    {TERM}
    EXP4.modo = TERM.modo
    EXP4.tipo = TERM.tipo
    EXP4.etq = TERM.etq
    EXP4.cod = TERM.cod

TERM ::=
    {boolean}
    TERM.modo = val
    TERM.tipo = tBool
    TERM.cod = apila(valorDe(boolean.lexema))
    TERM.etq = TERM.etqh + 1

TERM ::=
    {cadCaracteres}
    TERM.modo = val
    TERM.tipo = tChar
    TERM.cod = apila(valorDe(cadCaracteres.lexema))
    TERM.etq = TERM.etqh + 1

TERM ::=
    {natural}
    TERM.modo = val
    TERM.tipo = tNat
    TERM.cod = apila(valorDe(natural.lexema))
    TERM.etq = TERM.etqh + 1

TERM ::=
    {entero}
    TERM.modo = val
    TERM.tipo = tInt
    TERM.cod = apila(valorDe(entero.lexema))
    TERM.etq = TERM.etqh + 1

TERM ::=
    {real}
    TERM.modo = val
    TERM.tipo = tFloat
    TERM.cod = apila(valorDe(real.lexema))
    TERM.etq = TERM.etqh + 1

TERM ::=
    { ( ) }
    EXP.etqh = TERM.etqh

```

```

EXP.tsh = TERM.tsh
{EXP}
TERM.moda = EXP.moda
TERM.tipo = EXP.tipo
TERM.etq = EXP.etq
TERM.cod = EXP.cod
{}}

MEM = {iden} {RMEM}
RMEM.tsh = MEM.tsh
RMEM.etqh = MEM.etqh
RMEM.tipoh = damePropiedadesTS(MEM.tsh, iden.lexema).tipo
Si damePropsTS(MEM.tsh, iden.lexema).clase = pvar entonces
    RMEM.etqh = MEM.etqh + 2
    {RMEM}
    MEM.cod = apila(damePropiedadesTS(MEM.tsh, iden.lexema).dir)
        || apila_ind()
        || RMEM.cod
si no
    RMEM.etqh = MEM.etqh + 1
    {RMEM}
    MEM.cod = apila(damePropiedadesTS(MEM.tsh, iden.lexema).dir)
        || RMEM.cod
fin si
MEM.etq = RMEM.etq
MEM.tipo = RMEM.tipo

RMEM = λ
RMEM.tipo = RMEM.tipoh
RMEM.cod = apila(RMEM.nivel+1)
    || suma // no necesitamos poner el +2 porque
    // la direccion en la TS comienza en 2
    || apila_ind()
RMEM.etq = RMEM.etqh + 3
Si no
    RMEM.cod = apila(RMEM.nivel+1)
    || suma
    RMEM.etq = RMEM.etqh + 2
fin si

RMEM =
{->} RMEM_1.tsh = RMEM_0.tsh
RMEM_1.tipoh = tipoBase(RMEM_0.tsh, RMEM_0.tipoh)
{RMEM}
RMEM_0.tipo = RMEM_1.tipo
RMEM_0.cod = Apila_ind() || RMEM_1.cod

RMEM = {[ ] {EXP} { ] } RMEM
{ [ ]
EXP.tsh = RMEM.tsh
EXP.etqh = RMEM_0.etqh
{EXP}
RMEM_1.etqh = RMEM_0.etqh
RMEM_1.tsh = RMEM_0.tsh
{ ] }
Si RMEM_0.tipo = tArray and EXP.tipo = tEntero entonces
    RMEM_0.tipo = RMEM_1.tipo
    Size = damePropiedadesTS(RMEM.tsh, RMEM.tipoh).size
    RMEM_1.tipoh = tipoBase(RMEM_0.tsh, RMEM_0.tipoh)
    RMEM_1.etqh = EXP.etq + 3
    {RMEM}

```

```

        RMEM_0.etq = RMEM_1.etq
        RMEM_0.cod = EXP.cod || Apila(size) || multiplica || suma
            || RMEM_1.cod
        RMEM_0.etq = RMEM_1.etq
Si no
    RMEM.tipo = tError

OP0 ::=
    {<}
    OP0.op = menor

OP0 ::=
    {>}
    OP0.op = mayor

OP0 ::=
    {<=}
    OP0.op = menorIgual

OP0 ::=
    {>=}
    OP0.op = mayorIgual

OP0 ::=
    {=}
    OP0.op = distinto

OP0 ::=
    {=/=}
    OP0.op = distinto

OP1 ::=
    {+}
    OP1.op = suma

OP1 ::=
    {-}
    OP1.op = resta

OP1 ::=
    {or}
    OP1.op = oLogica

OP2 ::=
    {*}
    OP2.op = multiplicacion

OP2 ::=
    {/}
    OP2.op = division

OP2 ::=
    {%}
    OP2.op = resto

OP2 ::= and
    {and}
    OP2.op = yLogica

OP3 ::= <<

```

```

        {<<}
        OP3.op = despIzq

OP3 ::= >>
        {>>}
        OP3.op = despDer

OP4_1 ::= not
        {not}
        OP4_1.op = negLogica

OP4_1 ::= -unario
        {-unario}
        OP4_1.op = negArit

OP4_1 ::=
        {(nat) }
        OP4_1.op = castNat

OP4_1 ::=
        {(int) }
        OP4_1.op = castInt

OP4_1 ::=
        {(char) }
        OP4_1.op = castChar

OP4_1 ::=
        {(float) }
        OP4_1.op = castFloat

```

9 Esquema de traducción orientado al traductor predictivo – recursivo

9.1 Variables globales

Las variables globales utilizadas son tres

-
- Pend: Tipos o funciones que están pendientes de declarar
- callPend: Tipos o funciones que están pendientes de parchear
- Cod: donde esta el codigo maquina generado
- Nivel: nivel donde se encuentra el analizador

9.2 Nuevas operaciones y transformación de ecuaciones semánticas

- emit(codigo, token), emite una parte de codigo maquina.
- emit(codigo) emite un codigo maquina
- damToken(tToken). Devuelve un token de un tipo pasado.
- dameToken(tToken, value)
- token(), devuelve el token en que se cuenta en analizador.
- consume(), consume un token y lo devuelve
- consume(String: token) consume un token, en caso de que el token no sea el pasado por parametro termina la ejecución del programa.

El código en este caso en vez de concatenarse se emite una operación que se añade al último de el conjunto de operaciones.

Las operaciones se han cambiado a unos códigos correspondientes a la operación que

9.3 Esquema de traducción

```
PROGRAMA_INIT ::=
{
  ts = creaTS()
  Si token() pertenece {&} entonces
    PROGRAMA_1()
  Si no
    Programa_2()
}

PROGRAMA_1() ::=
  pendH = creaPendientes()
  tsph = creaTS()
  nivel = 0
  etqh = 0
  dir = 0
  DECS.callPendH = creaCallPends()
  DECS(nivel, dir, etqh, pendH, callPendH, tsph, ts, etq, pend,
callPend, errorDecs)
  Consume(&)
  SENTs(nivel, ts,etq, pend, callPend,etqSent,errorSent)
```



```

PROGRAMA.error = errorDec OR errorSent OR
    pend.size <> 0 or callPend.size <> 0
emit(stop)

```

```

PROGRAMA_2() ::=
    pend = creaPendientes()
    tsph = creaTS()
    callPend = creaCallPends()
    etq = 0
    nivel = 0
    Consume(&)
    SENTS(nivel, ts, etq, pend, callPend, callPendH, etqSent,
        errorSent)
    PROGRAMA.error = SENTS.errorSent
    emit(stop)

DECS(in nivel, in dirH, in etqh, in pendH, in callPendH, in tsph,
    out ts, out etq, out pend, out callPend, out errorDecs,
    out etq)
DEC(nivel, dirH, etqh, pendH, callPendH, tsph, tsDec, etqDec,
    pendDec, callPendDec, errorDec, clase, forward, lexema,
    props, dirSent)
errorDecs = errorDec
dir = dirH
si not errorDec and clase = "procedimiento" entonces
    si not forward and entonces
        si tsDec.existeID(DEC.lexema) and
            not pendDec.pendiente(lexema) or
            not pendDec.tipoPendiente(lexema)=proc)
            entonces
                errorSents = TRUE
            Si no si pendDec.pendiente(lexema)
                pendSent.eliminaPendiente(lexema)
                errorDecs = tsDec.compruebaCampos(lexema,
                    props, clase, nivel)
                tsSent.actualizaID(lexema, dirSent)
            Si no
                errorDecs = TRUE
            fin si
        si no si tsDec.existeID(lexema) entonces
            errorDecs = TRUE
        Si no // no existe ID y es forward
            tsDec.añadeID(lexema, props, clase, nivel, dirSent)
            pendDec.añadePendientes(lexema, "proc")
        fin si
    si not errorDec // es un tipo o una var
    entonces
        si tsDec.existeID(lexema) entonces
            errorDecs = TRUE
        Si no
            tsDec.añadeID(lexema, props, clase, nivel, dirDec)
            dir++
            Si clase = "tipo"
                and pendDec.existePendientes(lexema)
                and pendDec.tipoPendiente(lexema) = "tipo"
            Entonces
                pendDec.eliminaPendientes(lexema)
            Fin si
        Fin si
    Fin si
Fin si
    si token pertenece {;}

```

```

        RDECS_1(nivel, dir, etqDec, pendDec, callPendDec,
                tsDec, ts, pend, callPend, errorRdecs, etq, dir)
Si no
    RDECS_1(nivel, dir, etqDec, pendDec, callPendDec,
            tsDec, ts, pend, callPend, errorRdecs, etq, dir)
fin si
errorDecs = errorDecs or errorRdecs

RDECS_1(in nivel, in dirH, in etqH, in pendH, in callPendH, in tsH,
        Out ts, out pend, out callPend, out error, out etq, out dir)
DEC(nivel, dirH, etqh, pendH, callPendH, tsph, tsDec, etqDec,
pendDec,
    callPendDec, errorDec, clase, forward, lexema, props,
    dirDec)
error = errorDec
dir = dirH
si not error and clase = "procedimiento" entonces
    si not forward and entonces
        si tsDec.existeID(DEC.lexema) and
            not pendDec.pendiente(lexema) or
            not pendDec.tipoPendiente(lexema)=proc)
        entonces
            errorSents = TRUE
        Si no si pendDec.pendiente(lexema)
            pendSent.eliminaPendiente(lexema)
            error = tsDec.compruebaCampos(lexema,
                props, clase, nivel)
            tsSent.actualizaID(lexema,dirSent)
        Si no
            error = TRUE
        fin si
    si no si tsDec.existeID(lexema) entonces
        errorDecs = TRUE
    Si no // no existe ID y es forward
        tsDec.añadeID(lexema, props, clase, nivel, dirSent)
        pendDec.añadePendientes(lexema,"proc")
    fin si
si not errorDec // es un tipo o una var
entonces
    si tsDec.existeID(lexema) entonces
        error = TRUE
    Si no
        tsDec.añadeID(lexema, props, clase, nivel, dirDec)
        dir++
        Si clase = "tipo"
            and pendDec.existePendientes(lexema)
            and pendDec.tipoPendiente(lexema) = "tipo"
        Entonces
            pendDec.eliminaPendientes(lexema)
        Fin si
    Fin si
Fin si
si token pertenece {;}
    RDECS_1(nivel, dir, etqDec, pendDec, callPendDec,
            tsDec, ts, pend, callPend, errorRdecs, etq, dir)
Si no
    RDECS_1(nivel, dir, etqDec, pendDec, callPendDec,
            tsDec, ts, pend, callPend, errorRdecs, etq, dir)
fin si
error = error or errorRdecs

```

```

RDECS_2(in nivel, in dirH, in etqH, in pendH, in callPendH, in tsH,
        Out ts, out pend, out callPend, out error, out etq,out dir)
    Etq = etqH
    Ped = pendH
    Ts = tsh
    callPend = callPendH
    error = false
    dir = dirH

```

```

DEC(in nivel, in Dir in etqh, in pendH, in callPendH, in tsph,
    Out tsDec, out etqDec, out pendDec, out callPendDec,
    out errorDec, out clase, out forward, out lexema,
    out props, out dir)
si token() pertence {"tipo"} entonces
    DEC-tipo(nivel, dir, etqh, pendH, callPendH, tsph, tsDec,
        etqDec, pendDec, callPendDec, errorDec, clase, forward,
        lexema, props, dir)
si no si token() pertenece {"proc"}
    DEC-proc(nivel, dir, etqh, pendH, callPendH, tsph, tsDec,
        etqDec, pendDec, callPendDec, errorDec, clase, forward,
        lexema, props, dir)
si no
    DEC-var(nivel, dir, etqh, pendH, callPendH, tsph, tsDec,
        etqDec, pendDec, callPendDec, errorDec, clase, forward,
        lexema, props, dirSent)

```

```

DEC-var(in nivel, in dir, in etqh, in pendH, in callPendH, in tsph,
    Out tsDec, out etqDec, out pendDec, out callPendDec,
    out error, out clase, out forward, out lexema,
    out props, out dir)    lexema = consume()
consume(':')
clase = "variable"
forward = false
TIPOIDEN.tsph = DECVAR.tsph
TIPOIDEN.pendH = DECVAR.pendH
{TIPOIDEN}
TIPOIDEN(tsph, pendH, pend,props, error, etq, decSize, tipo)
error = tsph.existeID(tsph, lexema)or error
TIPOIDEN(in tsph, in pendH, out pend,out props, out error, out etq,
    Out decSize, out tipo)
Tipo = consume()
si no Si tipo = "boolean"
    Pend= pendH
    Error = false
    decSize = 1
    props = "<t: boolean>"
si no Si tipo = "caracter"
    Pend= pendH
    Error = false
    decSize = 1
    props = "<t: caracter>"
si no Si tipo = "natural"
    Pend= pendH
    Error = false
    decSize = 1
    props = "<t: natural>"
si no Si tipo = "integer"
    Pend= pendH
    Error = false
    decSize = 1
    props = "<t: integer>"

```

```

si no Si tipo = "float"
    Pend= pendH
    Error = false
    decSize = 1
    props = "<t: float>"
si no Si tipo = "record"
    Campos(pendh, tsph, propsCampos, error, pend, sizeDec)
    Props = "<t:rec>" ++ "<campos: " ++ props.campos++>"
si no Si tipo = "pointer"
    TIPOIDEN(tsph, pendH, pend,out propsPointer, error,
        Etq,sizePointer, tipoPointer )
    props = "<t:puntero>"++"<tbase: " ++ propsPointer ++>"
    pend = pendH
    sizeDec = 1
    si error and tipoPointer = ref
        and not tsph.existeID(propsPointer.id)
    entonces
        pend = añadePendiente(propsPointer.id)
        error = false
    fin si
si no si tipo "array"
    consume(['\'])
    elements = consume('natural')
    consume(']')
    consume('of')
    TIPOIDEN(tsph, pendH, pend,out propsArray, error,
        Etq,sizeArray, tipoArray )
    Props = "<t:array>" ++ "<nelem:" ++elements++>" ++
        "<tbase:" ++props.array>

si no
    Pend= pendH
    Error = tsph.existeTipo(tipo)
    decSize = tsph.dameSize(tipo)
    tipo = ref
    props = "<t:ref>"++ "<id: iden.lexema>"

Campos(in pendh, in tsph, out propsCampos, out error, out pend,
    Out sizeDec)
    Campo(pendH, tsph, pend, sizeCampo, errorCampo)
    RCampos(pendCampo, tsph, propsRCampos, error, pend,sizeCampos)
    SizeDec = sizeCampo + SizeCampos
    Error = error or errorCampo
Rcampos(in pendH, in tsph, out props, out error,out pend,out sizeDecs)
    si token() pertenece {;}
        pend = pendH
        error = FALSE
        sizeDec = 0
        props = ""
    si no
        consume (;)
        Campo(pendH, tsph, pendCampo, sizeCampo, errorCampo)
        RCampos(pendCampo, tsph, propsRCampos, error, pend,
            sizeCampos)
        SizeDec = sizeCampo + SizeCampos
        Error = error or errorCampo
    Fin si
Campo(in pendH, in tsph, out pend, out size, out error)
    lexema = consume()
    TIPOIDEN(tsph, pendH, pend, propsTipo, errorTipo,
        Etq,size, tipo)
    props = "<id: " ++ lexema++ ">" ++ "<t:" ++ propsTipo.t">"

```

```

        error = tsph.existeSubID(tipo,lexema) or errorTipo
DEC-tipo(in nivel, in dirH, in etqh, in pendH, in callPendH, in tsph,
        Out ts, out etqDec, out pendDec, out callPendDec,
        Out errorDec, out clase, forward, out lexema,
        Out props, out dir)
Consume('tipo')
Lexema = consume()
TIPOIDEN(tsph, pendH, pend, propsTipo, errorTipo,
        sizeTipo, subtipo)
ts = tsph
etqh = etq
pendCall = callPendH
Clase = "tipo"
forward = false
dir = dirH
props = "<t:ref> <tBase: " "subTipo">"
error = errorTipo or tsph.existeID(lexema)

DEC-proc(in nivel, in dirH, in etqh, in pendH, in callPendH, in tsph,
        Out ts, out etq, out pendDec, out callPend,
        Out error, out clase, out forward, out lexema,
        Out props, out dir)
Consume('proc')
Clase = "procedimientos"
Lexema = consume()
nTsph = tsph.clone()
subDir = 2
subNivel = nivel +1
DPARAMS(subnivel, nTsph, subDir, pendH, pendParams, nTsp,
        dirParams, paramsProps, errorParams)
Props = "<t:proc><params:"++paramsProps">"
PBLOQUE(subnivel, nTsp, dirParams, etqh, callPendH, callPend,
        pendParams,forward, errorBloque, etq)
Error = errorBloque or errorParams
        // el error de existir se gestiona en este
        // caso al añadir

PBLOQUE(in nivel, in tsph,in dirH, in etqh, in callPendH, in pendH
        out callPend, out forward, out error, out etq)
Si token() pertenece {forward}
        Forward = TRUE
        callPend = callPendH
        etq = etqH
        error = FALSE
si no
        consume ('{'')
        forward = FALSE
        si token() pertenece {&}
                prologo()
                SENTS(nivel, tsph, etqh + longPrologo, pend,
                        callPendH, callPend,etqSent)
                        errorSent)
                epilogo()
                etq = EtqDec + longEpligo
                error = errorSent or errorDecs
                callPend.resolverPend(DECS.ts)
                consume ('}')

        si no
                prologo()
                DECS(nivel, dirH, etqh + longPrologo, in pendH,

```

```

        callPendH, tsph, ts, etqDec, pend, callPend,
        errorDecs, etq)
    SENTS(nivel, ts, etqDec, pend, callPendH,
        callPend, etqSent)
        errorSent)
    epilogo()
    etq = EtqDec + longEpligo
    error = errorSent or errorDecs
    callPend.resolverPend(DECS.ts)
    consume (``)
fin si

DPARAMS(in nivel , in tsph , in dirh, in pendH, out pend, out tsp, out
    dir, Out props, out error)
Si token () pertence {} entonces
    Consume({})
    LISTAPARAMS(dirh, tsph, nivel, pendh , pend, size, props,
tsp, error, dir)
    Consume({})
Si no
    Dir = dirH
    Error = false
    Props = ""
    Tsp = tsph
Fin si
LISTAPARAMS(in dirh, in tsph, in nivel in pendh, out pend, out size,
out props,
    Out tsp, out error, out dir)
PARAM(tsph, error, lexema, propsParam, clase, sizeParam)
Si not error entonces
    Tsph.añadeID(lexema, propsParam, clase, nivel)
    RLISTAPARAMS(dirH, tsph, nivel, pendParam, pend,
        sizeParams, propsParams, tsp, error, dir)
    props = propsParam ++ PropsParams
    Dir = dirh+paramSize
    size = sizeParams + sizeParam
fin si

RLISTAPARAMS(in dirh, in tsph, in nivel, in pendH, out pend, out size,
    out props, Out tsp, out error, out dir)
Si token() pertenece {,}
    Error = false
    Dir = dirH
    Props = ""
Si no
    PARAM(tsph, pendH, pendParam, error, lexema, propsParam,
        clase, sizeParam)
    Si not error entonces
        Tsph.añadeID(lexema, propsParam, clase, nivel, dirh)
        RLISTAPARAMS(dirH, tsph, nivel, pendParam, pend,
            sizeParams, propsParams, tsp, error, dir)
        props = propsParam ++ PropsParams
        Dir = dirh+paramSize
        size = sizeParams + sizeParam
    fin si
fin si

PARAM(in tsph, pendh, out pend, out error, out lexema, out props,
    out clase, out param, Out sizeParam)
Si token() pertence {var} entonces
    Consume(var)

```

```

Lexema = consume()
Clase = "p variable"
Consume(':')
TIPOIDEN(tsph, pendH, pend, propsTipo,errorTipo, size,
        out tipo)
sizeParam = 1
error = errorTipo or tsph.existeID(lexema)
Props = "<t: " ++ tipo ++ ">"
Param = <modo:variable, t: TIPOIDEN.props>
Si no
Lexema = consume()
Clase = "variable"
Consume(':')
TIPOIDEN(tsph, pendH, pend, propsTipo,errorTipo, sizeParam,
        out tipo)
error = errorTipo or tsph.existeID(lexema)
Props = "<t: " ++ tipo ++ ">"
Param = <modo:valor, t: TIPOIDEN.props>
Fin si

```

```

SENTS(in nivel, in ts,in etqh, in callPendH, callPend, out etq,
      out error)
SENT(nivel, ts, etqh, callPendH, callPendSent, etqSent,
     errorSent)
SENTS(nivel, ts, etqh, callPendSent, callPend, etq, errorRSENTS)
Error = errorSent or errorRSENTS

```

```

SENTS(in nivel, in ts,in etqh, in callPendH, callPend, out etq,
      out error)
si token() pertenece {;}
    error = false
    etq = etqh
    callPend = callPendH
si no
    consume (;)
    SENT(nivel, ts, etqh, callPendH, callPendSent, etqSent,
         errorSent)
    SENTS(nivel, ts, etqh, callPendSent, callPend, etq,
          errorRSENTS)
    Error = errorSent or errorRSENTS
Fin si

```

```

SENT(in nivel, in ts, in etqh, in callPendH, out callPend, out etq,
     Out erro)
si token() pertenece {in}
    SREAD(etqh, tsh, etq, error)
si token() pertences {out}
    SWRITE(etqh, tsh, etq, error)
si token() pertences {{} encontes
    SBLOQUE(etqh, tsh, callPendH, callPend, etq, error)
si token() pertences {if} encontes
    SIF(etqh, tsh, callPendH, callPend, etq, error)
si no si token() pertences {for} encontes
    SFOR(etqh, tsh, callPendH, callPend, etq, error)
si no si token() pertences {while} encontes
    SWHILE(out errorSent1, etq, etqh, tsh)
si no si token() pertences {new} encontes
    SNEW(etqh, tsh, etq, error)
si no si token() pertences {dispose} encontes

```

```

        SDEL(etqh, tsh, etq, error)
    si no
        RSENT(etqh, tsh, callPend, consume(),etq, error, callPendH)
    fin si
    errorSent = errorSent1
}

SWRITE (in etqh, in tsh, out etq, out error)::=
    consume('out');// out
    EXP(etqh, tsh, etqExp, tipo, modo)
    si tipo = tError
    entonces
        vaciaCod()                error = TRUE
    si no
        emit(escribir)
        etq = etqexp + 1
    fin si
    consume('\') // )
}

SREAD (in etqh, in tsh, out etq, out error)::= {
    consume('in') // in
    lexema = consume()
    error = not ts.existeID(lexema)
    si error
    entonces
        vaciaCod()
    si no
        emit(leer)
        emit(desapila_dir,tsh.damePropsTS(lexema).dir)
        etq = etqh + 2          consume ('\') // (
    fin si
}

SIF(in etqh, in tsh, in callPendH, out callPend, out etq, out error)
    consume('if')
    EXP(etqh, tsh, etqExp, tipo, modo)
    Consume('then')
    emnit(ir_f,-1)// lugar = etqexp <- al final de sent o +1
    SENT(nivel, ts, etqh, callPendH, callPendSent, etqSent,
        errorSent)
    if token() pertenece {else} entonces
        emit(ir_a(?)) // lugar etqsent<-al final solo si hay else
        parchea(etqexp, etqsent+1) // +1 por el ir_a
        PELSE(etqh, tsh, callPendSent, callPendElse, etq,
            errorElse)
        parchea(etqsent, etqelse)
        etq = etqelse
    si no
        parchea(etqexp, etqsent) // no hay +1 porque no hay ir_a
        etq = etqsent
        errorElse = FALSE
    fin si
    si tipo = tError or anotherErrorSent or errorElse
    entonces
        errorSent = TRUE
    si no
}

PELSE (in etqh, in tsh, in callPendH, out callPend, out etq,out error)
    consume('else')

```



```

    SENT(etqh, tsh, callPendH, callPend, etq, error)
}

SWHILE(in etqh, in tsh, in callPendH, out callPend, out etq, out error)
    consume('while')
    EXP(etqh, tsh, etqExp, tipo, modo)
    ir_f(?) // pos = etqexp, dest = fin SWHILE
    consume('do')
    SENT(nivel, ts, etqExp+1, callPendH, callPendSent, etqSent,
        errorSent)
        // +1 x ir_f
    ir_a(etqh) // dest init WHILE
    etq = etqsent +1 // +1 x ir_a
    parchea (etqexp, etq)
    si tipo = tError || anotherErrorSent || errorElse
    entonces
        error = TRUE
    si no

}

// SFOR ::= for MEM = EXP to EXP do SENT
SFOR (in etqh, in tsh, in callPendH, out callPend, out etq, out error)
    Consume(for)
    Lexema = consume()
    consume ('=')
    MEM(etqh, tsh, out etq, out error, tipoMem, dirMem)
    EXP(etqh, tsh, etqExp1, tipoExp, modoExp)
    Si not dameTipo(tipoMem, TipoExp, oAsign) = entero entonces
        Error = true
    Si no
        Si modoExp = var entonces
            Emit(Apila_ind)
            etqExp1 = etqExp1 + 1
        fin si
        emit(desapila_dir, dirMem)
        // hasta aquí la asinacion etqExp1+1
        Consume (to)
        EXP(etqExp1, tsh, etqExp2, tipoExp2, modoExp2)
        Si not dameTipo(tipoMem, TipoExp, oMenor) = tBoolean entonces
            Error = true
        Si no
            Si modoExp2 = var entonces
                Emit(Apila_ind)
                etqExp2 = etqExp2 + 1
            fin si
            //Hasta aquí el previo de la comparación
            //se saltará a aquí etqExp2
            Emit(duplica)
            Emit(apila_dir, dirMem)
            Emit(menorIgual)
            Emit(ir_f, -1) // etq = etqExp + 3, salta a etq del for
            SENT(nivel, tsh, etqExp2+4, callPendH, callPendSent,
                etqSent, errorSent)
            emit(ir_a, etqExp2)
            // aquí hay que retornar el salto
            // quitamos el valor de comparacion
            Parchea(etqExp2 + 3, etqSent)
            Emit(desalipa)
        Fin si
    Fin si
SNEW(in etqh, in tsh, out etq, out error)

```

```

Consume('new')
MEM(etqh, tsh, etqMem, error, tipoMem, dirMem)
Si not mem.tipo = puntero or error entonces
    Error = true
Si no
    Emit(eNew, damePropTs(tipoMem).size)
    Desapila_ind()
    Etq = etqMem+2

SDEL(in etqh, in tsh, out etq, out error)
Consume('new')
MEM(etqh, tsh, etqMem, error, tipoMem, dirMem)
Si not mem.tipo = puntero or error entonces
    Error = true
Si no
    Emit(dispose, damePropTs(tipoMem).size)
    Etq = etqMem+1

RSENT(in etqh, in tsh, in callPendH, in lexema, out etq, out error, out
    callPendH, out dir)
si not tsh.existeID(lexema) or tsh.tipoID(lexema) = tipo
entonces
    error = TRUE
si no Si token() pertence {} and tsh.tipoID(lexema) = proc
entonces
    consume()
    dirProc = tsh.dameDir(RSENT.iden)
    emite(apila-ret, RSENT.etqh)
    emite(PPARAMS.cod)
    emite(ir_a, dirProc) \\ la dirección esta en la pila
    si dirProc = -1 entonces
        callPend = añadeCallPend(lexema, PPARAMS.etqh + 2)
    LPARAMS(etqh, tsh, etq, lexema, error)
    Consume()
si no si tsh.tipoID(lexema) = proc

    dirH = tsph.damePropsTs(lexema).dir
    RMEM(etqh, tsh, etqH, tipoH, dirH, error, tipoMem, dirMem)
    Consume(:=)
    EXP(etqh, tsh, etqExp1, tipoExp, modoExp)
    Emit(apila(dirMem))
    Apila(memDir)
    Si modoExp = val
        Emit(apila_ind)
        Etq = etqExp +1
    Si no
        Size = tsh.damePropsTs(lexema).size
        Emit(copia, size)
    Fin si
Si no
    Error = TRUE

LPARAMS(in etqh, in tsh, out etq, out lexema, out error)
EXP(etqh, tsh, etqExp, tipoExp, modoExp, errorExp)
nParamsH = 1
procName = lexema
si and (modoExp = val or tsh.parametroPorValor(lexema, nParamsH))
entonces
    paramSizeH = 1
    emite (apila, 0)
    emite (apila_ind)

```

```

        emite (apila,2)
        emite (suma)
        emite (apila_ind)
        etqPre = etqExp + 5
    si no
        paramSizeH = tsh.dameSize(EXP.tipo)
        emite (apila,0)
        emite (apila_ind)
        emite (apila,2)
        emite (suma)
        emite (copia,size)
        etqPre = etqExp + 5
    RLPARAMS(etqh, tsh, etq, lexema, error)
    Error = tsh.comparaParamFunc(lexema, modo, tipo)
        or tipoExp = tError or RLPARAMS.error

RPARAMS(in etqh,in tsh,out etq,in lexema,out error)
    Si token() pertenece {,}
        Etq = etqh
        Error = false
    Si no
        EXP(etqh, tsh, etqExp, tipoExp, modoExp,errorExp)
        nParamsH = 1
        procName = lexema
        si and (modoExp = val or
            tsh.parametroPorValor(lexema,nParamsH))
        entonces
            paramSizeH = 1
            emite (apila,0)
            emite (apila_ind)
            emite (apila,2)
            emite (suma)
            emite (apila_ind)
            etqPre = etqExp + 5
        si no
            paramSizeH = tsh.dameSize(EXP.tipo)
            emite (apila,0)
            emite (apila_ind)
            emite (apila,2)
            emite (suma)
            emite (copia,size)
            etqPre = etqExp + 5
        RLPARAMS(etqh, tsh, etq, lexema, error)
        Error = tsh.comparaParamFunc(lexema, modo, tipo)
            or tipoExp = tError or RLPARAMS.error
    fin si

EXP (out tipo)::= {
    EXP1(out tipol)
    tipoH =tipol;
    // XXX codH = EXP1.cod}
    si tipol = tError
    entonces
        tipo = tError
    si no
        si token pertenece {< > <= >= \=} // fin de instrucción
        entonces
            REXP_1(in tipoH, out tipo2);
            si tipo2 = tError;
            entonces
                tipo = tError;

```

```

                vaciaCod();
            si no
                tipo = tBool;
            fin si
        si no
            REXP_2();
            tipo = tipoH
        fin si
    fin si
}

REXP_1(in tipoH, out tipo) ::= {
    OP0(out op)
    EXP1(out tipol)
    tipo = dameTipo(tipoH,tipo,op)
    si tipo = tError
    entonces
        vaciaCod()
    si no
        emit(op)
    fin si
}

REXP_2() ::= {
    // no hacemos nada el tipo heredado lo hacemos arriba para evitar el
    // paso de parametros y el cod es global
}

EXP1 (out tipo)::= {
    EXP2 (out tipol)
    si token pertenece { + - or} // fin de instrucción
    entonces
        REXP1_1(in tipoH, out tipo2)
        si tipol = tError or tipo2 = tError
        entonces
            tipo = tError
            vaciaCod()
        si no
            tipo = tipo2
        fin si
    si no
        REXP1_2()
        tipo = tipol
    fin si
}

REXP1_1 (in tipoH, out tipo)::= {
    OP1 (out op)
    EXP2 (out tipol)
    tipoH1 = dameTipo(tipoH, tipol,op)
    si tipoH1 = tError
    entonces
        vaciaCod()
    si no
        emit(op)
    fin si

    si token pertenece { + - or } // fin de instrucción
    entonces
        REXP1_1(in tipoH1, out tipo2)
        tipo = tipo2

```

```

        si no
            REXP1_2()
            tipo = tipoH1
        fin si
    }

REXP1_2 ::= {
}

EXP2 (out tipo)::= {
    EXP3 (out tipol)
    si token pertenece {and * / %} // fin de instrucción
    entonces
        REXP2_1(in tipoH, out tipo2)
        si tipol = tError or tipo2 = tError
        entonces
            tipo = tError
            vaciaCod()
        si no
            tipo = tipo2
    si no
        REXP2_2()
        tipo = tipol
    fin si
fin si
}

REXP2_1 (in tipoH, out tipo)::= {
    OP2 (out op)
    EXP3 (out tipol)
    tipoH1 = dameTipo(tipoH, tipol,op)
    si tipoH1 = tError
    entonces
        vaciaCod()
    si no
        emit(op)
    fin si

    si token pertenece {and * / %} // fin de instrucción
    entonces
        REXP2_1(in tipoH1, out tipo2)
        tipo = tipo2
    si no
        REXP2_2()
        tipo = tipoH1

    fin si
}

REXP2_2(){
}

EXP3(out tipo) ::= {
    si token() pertenecea {}
    entonces
        EXP4_2(out tipol)
    si token () perteneces {(float) (int) (nat) |valor| (char)}
        EXP4_1(out tipol)
    si no
        EXP4_3(out tipol)
    fin si
}

```

```

si tipoH = tError
entonces
    tipo = tError
    vaciaCod()
si no
    tipoH = tipo1
    si token pertenece { << >> } // fin de instrucción
    entonces
        REXP3_2()
        tipo = tipo1
    si no
        REXP3_1(in tipoH, out tipo2)
        si tipo2 = tError
        entonces
            tipo = tError
        si no
            tipo = tNat
        fin si
    fin si
fin si
}

REXP3_1(in tipoH, out tipo)::= {
    OP3 EXP3
    OP3 (out op)
    EXP3(out tipo1)
    tipo = dameTipo(tipo1, tipoH, op)
    si tipo = tError
    entonces
        vaciaCod()
    si no
        emit(op)
    fin si
}

REXP3_2 ::= {
}

EXP4_1 (out tipo)::= {
    OP4_1(out op)
    TERM(out tipo1)
    tipo = dameTipo(tipo, op)
    si tipo = tError
    entonces
        vaciaCod()
    si no
        emit(op)
    fin si
}

EXP4_2 (out tipo)::= {
    consume('|')
    TERM(out tipo1)
    tipo = dameTipo(tipo, '|')
    si tipo = tError
    entonces
        vaciaCod()
    si no
        emit(valor_absolut)
    fin si
    consume('|')
}

```

```

}

EXP4_3 (out tipo) ::= {
    TERM(out tipol)
    tipo = tipol
}

TERM(out tipo) ::= {
    si token () pertenece {booleanvalue}
    entonces
        TERM_1(out tipol)
    si token () pertenece {character}
    entonces
        TERM_2(out tipol)
    si token () pertenece {natural}
    entonces
        TERM_3(out tipol)
    si token () pertenece {entero}
    entonces
        TERM_4(out tipol)
    si token () pertenece {real}
    entonces
        TERM_5(out tipol)
    si token () pertenece {identificador}
    entonces
        TERM_6(out tipol)
    si token () pertenece {'('}
    entonces
        TERM_7(out tipol)
    fin si
    tipo = tipol
}

TERM_1(out tipo) ::= {
    var boolean = consume();
    tipo = tBool;
    emit(apila,dameToken(tipo,valorDe(boolean.lexema))
}

TERM_2(out tipo) ::= {
    var caracter= consume();
    tipo = tChar;
    emit(apila,dameToken(tipo,valorDe(caracter.lexema))
}

TERM_3(out tipo) ::= {
    var natural = consume()
    tipo = tNat
    emit(apila,dameToken(tipo,valorDe(natural.lexema))
}

TERM_4(out tipo) ::= {
    var entero = consume()
    tipo = tInt;
    emit(apila,dameToken(tipo,valorDe(entero.lexema))
}

TERM_5(out tipo) ::=
    {}
    var real = consume();
    tipo = tFloat

```

```

        emit(apila,dameToken(tipo,valorDe(real.lexema))
    }

    TERM_6 (out tipo) ::= {
        iden(out lexema)
        si (not existeID(ts,lexema)
        entonces
            tipo = tError
            vaciaCod()
        si no
            tipo = dameTipo(ts,lexema)

        emit(apila_dir,dameToken(entero,valorDe(damePropiedadesTS(ts,lex
ema).dirProp))
        fin si
    }

    TERM_7(out tipo) ::= {
        consume('(')
        (
            EXP(out tipo1)
            tipo = tipo1
            consume(')')
        )
    }

    OP0 (out op) ::= {
        si token pertenece {'<'}
        entonces
            OP0_1(op)
        si token pertenece {'>'}
        entonces
            OP0_2(op)
        si token pertenece {'<='}
        entonces
            OP0_3(op)
        si token pertenece {'>='}
        entonces
            OP0_4(op)
        si token pertenece {'='}
        entonces
            OP0_5(op)
        si token pertenece {'=\='}
        entonces
            OP0_6(op)
    }

    OP0_1 (out op) ::= {
        consume('<')
        op = menor
    }

    OP0_2 (out op) ::= {
        consume('>')
        op = mayor
    }

    OP0_3 (out op) ::= {
        consume('<=')
        op = menorIgual
    }

```



```

OP0_4 (out op)::= {
    consume('>=')
    op = mayorIgual
}

OP0_6 (out op)::= {
    consume('!=')
    op = distinto
}

OP1 (out op)::={
    si token pertenece {'+'}
    entonces
        OP1_1(op)
    si token pertenece {'-'}
    entonces
        OP1_2(op)
    si token pertenece {'or'}
    entonces
        OP1_3(op)
}

OP1_1 (out op)::= {
    consume('+')
    op = suma
}

OP1_2 (out op)::= {
    consume('-')
    op = resta
}

OP
1_3 (out op)::= {
    consume('or')
    op = oLogica
}

OP2 (our op)::={
    si token pertenece {'*'}
    entonces
        OP2_1(op)
    si token pertenece {'/' }
    entonces
        OP2_2(op)
    si token pertenece {'%'}
    entonces
        OP2_3(op)
    si token pertenece {'and'}
    entonces
        OP2_4(op)
}

OP2_1 (out op)::= {
    consume('*')
    op = multiplicacion
}

OP2_2 (out op)::= {

```

```

        consume('/')
        op = division
    }

OP2_3 (out op)::= {
    consume('%')
    op = resto
}

OP2_4 (out op)::= {
    consume('and')
    op = yLogica
}

OP3 (out op)::={
    si token pertenece {'<<'}
    entonces
        OP3_1(op)
    si token pertenece {'>>'}
    entonces
        OP3_2(op)
}

OP3_1 (out op)::= {
    consume('<<')
    op = despIzq
}

OP3_2 (out op)::= {
    consume('>>')
    op = despDer
}

OP4_1 (out op)::= {
    si token pertenece {' not'}
    entonces
        OP4_1_1(op)
    si token pertenece {'-unario'}
    entonces
        OP4_1_2(op)
    si token pertenece {' castNat'}
    entonces
        OP4_1_3(op)
    si token pertenece {' castInt'}
    entonces
        OP4_1_4(op)
    si token pertenece {'castChar '}
    entonces
        OP4_1_5(op)
    si token pertenece {' castFloat'}
    entonces
        OP4_1_6(op)
}

OP4_1_1 (out op)::={
    consume('not')
    op = negLogica
}

OP4_1_2 (out op)::= {

```

```
        consume('-unario')
        op = negArit
    }

    OP4_1_3(out op) ::= {
        consume('(nat)')
        op = castNat
    }

    OP4_1_4 (out op) ::= {
        consume('(int)')
        op = castInt
    }

    OP4_1_5 (out op) ::= {
        consume('(char)')
        op = castChar
    }

    OP4_1_6 (out op) ::= {
        consume('(float)')
        op = castFloat
    }
```

10 Formato de representación del código P

El formato de representación del fichero es un fichero binario codificado con bytecode creado por nosotros.

Las instrucciones el primer byte representara la operación los sucesivos bytes seran operandos de la operación si la necesita. Es decir escribir no necesita operadores porque los coge de la pila, sin embargo la operación de apila dir tendra como segundo operando la dirección.

La codificación es de la siguiente forma

```
public static final byte MENOR = 0X00;
public static final byte MAYOR = 0X01;
public static final byte MENORIGUAL = 0X02;
public static final byte MAYORIGUAL = 0X03;
public static final byte DISTINTO = 0X04;
public static final byte SUMA = 0X05;
public static final byte RESTA = 0X06;
public static final byte PRODUCTO = 0X07;
public static final byte DIVISION = 0X08;
public static final byte MODULO = 0X09;
public static final byte YLOGICO = 0X10;
public static final byte OLOGICO = 0X11;
public static final byte NOLOGICO = 0X12;
public static final byte SIGNO = 0X13;
public static final byte DESPLAZAMIENTOIZQUIERDA = 0X14;
public static final byte DESPLAZAMIENTODERECHA = 0X15;
public static final byte CASTNAT = 0X16;
public static final byte CASTINT = 0X17;
public static final byte CASTCHAR = 0X18;
public static final byte CASTFLOAT = 0X19;
public static final byte APILA = 0X20;
public static final byte APILA_DIR = 0X21;
public static final byte DESAPILA = 0X22;
public static final byte DESAPILA_DIR_BOOLEAN = 0X40;
public static final byte DESAPILA_DIR_INTEGER = 0X41;
public static final byte DESAPILA_DIR_NATURAL = 0X42;
public static final byte DESAPILA_DIR_FLOAT = 0X43;
public static final byte DESAPILA_DIR_CHAR = 0X44;
public static final byte LEER = 0X24;
public static final byte ESCRIBIR = 0X25;
public static final byte VALOR_ABSOLUTO = 0X26;
```

Las operaciones que necesitan espacio adicional para la codificación son Apila, ApilaDir y DesapilaDir

11 Notas sobre la implementación

11.1 Descripción de archivos

Los Documentos. Además del fichero de texto se adjunta la imagen que representa a la maquina de estados.

- PLG-Grupo12-Memoria.rtf
- Autómata.jpg

El source esta dividido de la siguiente forma

Analizador léxico:

Esta en el paquete analizadorLexico. Consta de tres elementos

- ALexico.java: Es el cuerpo del analizador léxico, la implementación del automata
- Token.java: Los objetos de esta clase representan los tokens leídos por el analizador léxico.
- tToken; Contiene información sobre todos los tokens

Analizador Sintactico y traductor: (No acabado el sintactico)

Está en el paquete analizadorSintactico. Engloba en analizador sintactico y emite los codigos para el código maquina. Los archivos que tiene

- ASintactico: contiene el cuerpo del analizador sintáctico. Llama al analizador léxico y después a la parte del sintáctico, por último crea el fichero objeto para el intérprete.
- Emit: contiene la gestión de traducción de las operaciones máquina desde el traductor. Almacenando toda la información en vector que posteriormente es salvada en un fichero. Este fichero se encarga de la traducción y crea el fichero objeto.

Interprete

Paquete interprete y subpaquetes instruccionesMV y tipos.

Paquete interprete:

- Interprete.java: se encuentra el cuerpo del intérprete. Lee un fichero pasado por parámetro. Como segundo parámetro puede escribirse la palabra trazar si se quiere poner en modo traza.

Paquete instruccionesMV:

Paquete tipos contiene todos los tipos **primitivos** de la maquina virtual, asi como una clase abstracta que deben implementar todos. Tienen un procedimiento toBytes y fromBytes para la serialización.

Otras notas sobre la implementación que se consideren pertinentes (por ejemplo: diagramas de clase UML describiendo la arquitectura del sistema).

Qué se ha conseguido y qué se ha dejado pendiente para más adelante.

Libros, artículos y otras fuentes de información utilizadas (por ejemplo páginas web).

Sólo si alguno fuese necesario...

[illegible]

* * * * *

* * * * *

**

*