1. Sintaxis abstracta

En la siguiente tabla se presentan los constructores de los nodos que conforman el árbol de sintaxis abstracta. En la descripción de estos, los tipos de la función constructora que aparecen entre paréntesis sólo hacen referencia a los argumentos de dicha función y **nunca** al tipo de retorno:

Nombre	Especificación	Descripción
Programa		
prog	LD × LI → Prog	Crea un programa a partir de una lista de declaraciones (LD) y una lista de instrucciones (LI)
Declaraciones		
Listas		
no_decs	→ LD	Crea una lista de declaraciones vacía
varias_decs	LD × D → LD	Concatena una declaración (D) por el final de una lista de declaraciones (LD) y devuelve la lista resultante
Básicas		
var	String × Tipo → D	Crea una declaración de una variable a partir de su cadena identificadora (String) y su tipo (Tipo)
type	String × Tipo → D	Crea una declaración de un tipo definido por el programador a partir de su cadena identificadora (String) y su tipo (Tipo)
proc	String × LF × LD × LI → D	Crea una declaración de un procedimiento dado su nombre, o cadena identificadora (String), la lista de parámetros (formales, LF), y la lista de identificadores (LD) y de instrucciones (LI) que presenta el cuerpo del procedimiento
Parámetros formales		
pvalor	String × Tipo → F	Construye un parámetro formal pasado por valor, dado su nombre, o cadena identificadora (String), y su tipo (Tipo)
pvar	String × Tipo → F	Crea un parámetro formal pasado por variable, dado su nombre, o cadena identificadora (String), y su tipo (Tipo)
no_params	→ LF	Crea una lista de parámetros formales vacía

varios_params	LF × F → LF	Concatena un parámetro formal (F) por el final de una lista de parámetros formales (LF) y devuelve la lista resultante
Tipos		
Básicos		
int	→ Tipo	Devuelve el tipo de los números enteros
bool	→ Tipo	Devuelve el tipo de los valores lógicos (booleanos)
real	→ Tipo	Devuelve el tipo de los números reales
string	→ Tipo	Devuelve el tipo de las cadenas de caracteres
Renombrados		
ref	String → Tipo	Devuelve el resultado de renombrar un tipo, dando el nuevo nombre (cadena identificadora, String) de dicho tipo
Compuestos		
array	Tipo × String → Tipo	Devuelve el tipo de un array dado su tamaño, proporcionado por un literal (String), y el tipo de cada uno de sus elementos (tipo base, Tipo)
registro	LC → Tipo	Devuelve el tipo de un registro dada la lista de sus campos de datos (LC)
puntero	Tipo → Tipo	Devuelve el tipo de un puntero dado su tipo base, el tipo del dato al que apunta (Tipo)
Campos de datos		
campo	String × Tipo → C	Construye un campo de datos dado su nombre, o cadena identificadora (String), y su tipo (Tipo)
un_campo	C → LC	Crea una lista que contiene un solo campo de datos, dado por (C)
varios_campos	LC × C → LC	Concatena un campo de datos (C) por el final de una lista de campos (LC) y devuelve la lista resultante
Instrucciones		
Listas		
no_ins	→ LI	Crea una lista de instrucciones vacía
varias_ins	LI × I → LI	Concatena una instrucción (I) por el final de una lista de instrucciones (LI) y devuelve la lista resultante

Básicas		
asig	E × E → I	Crea una instrucción de asignación dadas las expresiones izquierda y derecha (E) de la misma
invoc	E × LE → I	Crea una instrucción de invocación dada la expresión (E) que identifica la función y una lista de parámetros reales (proporcionada por una lista de expresiones, LE)
Bloques		
seq	LD × LI → I	Crea una instrucción de bloque dada la lista de declaraciones (LD) y de instrucciones (LI) del cuerpo de dicho bloque
if-then	E × LI → I	Genera un bloque condicional simple dada la condición (proporcionada por una expresión, E) y la lista de instrucciones (LI) del cuerpo de este
if-then-else	E × LI × LI → I	Genera un bloque condicional compuesto dada la condición (proporcionada por una expresión, E) y la lista de instrucciones (LI) de la rama principal y la alternativa, respectivamente, de dicho bloque
while	E × LI → I	Crea un bucle condicional dada la condición de ejecución (proporcionada por una expresión, E) y la lista de instrucciones (LI) del cuerpo del bucle
E/S estándar		
read	E → I	Crea una instrucción de lectura dada la expresión (E) que identifica la variable donde se almacenará el resultado de esta lectura
write	E → I	Crea una instrucción de escritura de un dato que viene dado por la expresión (E) que lo identifica
nl	→ I	Crea una instrucción que introduce un salto de línea por la salida estándar
Memoria dinámica		
new	E → I	Crea una instrucción de reserva de memoria dinámica dada la expresión (E) que identifica la variable que apunta a la nueva región reservada
delete	E → I	Crea una instrucción de liberación de memoria dinámica dada la expresión (E) que identifica la variable que apunta a la región que va a ser liberada
Expresiones		
Listas		

no_exp	→ LE	Crea una lista de expresiones vacía
varias_exp	LE × E → LE	Concatena una expresión (E) por el final de una lista de expresiones (LE) y devuelve la lista resultante
Básicas		
entero	String → E	Devuelve la expresión de un número entero dado el literal (String) que lo representa
decimal	String → E	Devuelve la expresión de un número real (i. e.: un número con parte decimal) dado el literal (String) que lo representa
cadena	String → E	Devuelve la expresión de una cadena de caracteres dado el literal que la representa
ident	String → E	Devuelve la expresión de un nombre de variable (i. e.: su identificador) dado el literal (String) que lo representa
Constantes		
true	→ E	Devuelve la expresión del valor lógico “verdadero”
false	→ E	Devuelve la expresión del valor lógico “falso”
null	→ E	Devuelve la expresión del valor nulo
Operadores relacionales		
eq	E × E → E	Crea la expresión de una comprobación de igualdad entre dos términos dadas sus expresiones (E)
ne	E × E → E	Crea la expresión de una comprobación de desigualdad entre dos términos dadas sus expresiones (E)
le	E × E → E	Crea la expresión de una comprobación de la condición “menor o igual que” entre dos términos dadas sus expresiones (E)
ge	E × E → E	Crea la expresión de una comprobación de la condición “mayor o igual que” entre dos términos dadas sus expresiones (E)
lt	E × E → E	Crea la expresión de una comprobación de la condición “menor que” entre dos términos dadas sus expresiones (E)
gt	E × E → E	Crea la expresión de una comprobación de la condición “mayor que” entre dos términos dadas sus expresiones (E)
Operadores lógicos		

and	E × E → E	Crea la expresión de una conjunción lógica a partir de las expresiones (E) de sus dos operandos
or	E × E → E	Crea la expresión de una disyunción lógica a partir de las expresiones (E) de sus dos operandos
not	E → E	Crea la expresión de una negación lógica a partir de la expresión (E) de su único operando
Operadores aritméticos		
suma	E × E → E	Crea la expresión de una suma a partir de las expresiones (E) de sus dos sumandos
resta	E × E → E	Crea la expresión de una resta a partir de las expresiones (E) de sus dos operandos
neg	E → E	Crea la expresión de un inverso aditivo dada la expresión (E) del operando al cual se aplica esta operación
mul	E × E → E	Crea la expresión de una multiplicación a partir de las expresiones (E) de sus dos factores
div	E × E → E	Crea la expresión del cociente de una división a partir de las expresiones (E) de sus dos operandos
mod	E × E → E	Crea la expresión del módulo (i. e.: el resto) de una división a partir de las expresiones (E) de sus dos operandos
Acceso a memoria		
indx	E × E → E	Crea la expresión de una indexación de arrays, dada la expresión (E) del identificador del array y la del índice
acc	E × String → E	Crea la expresión del acceso a un campo de datos (proporcionando su cadena identificadora, String) de un registro identificado por una expresión (E) dada
indir	E → E	Crea la expresión de una indirección, dada la expresión (E) del puntero a desreferenciar

2. Vinculación

```
global ts <- tabla_vacía() // Tabla de símbolos

// Métodos auxiliares

recolecta(id, nodo):
      si está_en(ts, id):
```

```
            error id_duplicado
      si no:
            añade(ts, id, nodo)

// Programa

vincula(prog(Ds, Is)):
      vincula1(Ds)
      vincula2(Ds)
      vincula(Is)

// Listas de declaraciones

vincula1(no_decs()): vacío

vincula1(varias_decs(Ds, D)):
      vincula1(Ds)
      vincula1(D)

vincula2(no_decs()): vacío

vincula2(varias_decs(Ds, D)):
      vincula2(Ds)
      vincula2(D)

// Declaraciones básicas

vincula1(var(id, T)):
      vincula1(T)
      recolecta(id, $)

vincula2(var(id, T)):
      vincula2(T)

vincula1(type(id, T)):
      vincula1(T)
      recolecta(id, $)

vincula2(type(id, T)):
      vincula2(T)

vincula(proc(id, PFs, Ds, Is)):
      recolecta(id, $)
      abre_nivel(ts)
      vincula1(PFs)
      vincula1(Ds)
```

```
      vincula2(PFs)
      vincula2(Ds)
      vincula(Is)
      cierra_nivel(ts)

// Parámetros formales

vincula1(no_params()): vacío

vincula1(varios_params(PFs, PF)):
      vincula1(PFs)
      vincula1(PF)

vincula2(no_params()): vacío

vincula2(varios_params(PFs, PF)):
      vincula2(PFs)
      vincula2(PF)

vincula1(pvalor(id, T)):
      vincula1(T)
      recolecta(id, $)

vincula2(pvalor(id, T)):
      vincula2(T)

vincula1(pvar(id, T)):
      vincula1(T)
      recolecta(id, $)

vincula2(pvar(id, T)):
      vincula2(T)

// Tipos básicos

vincula1(int()): vacío
vincula2(int()): vacío

vincula1(bool()): vacío
vincula2(bool()): vacío

vincula1(real()): vacío
vincula2(real()): vacío

vincula1(string()): vacío
vincula2(string()): vacío
```

```
// Tipos renombrados

vincula1(ref(id)):
      si está_en(ts, id):
            $.vínculo <- valor_de(ts, id)
      si no:
            error id_no_declarado

vincula1(array(T, tam)):
      vincula1(T)

vincula2(array(T, tam)):
      vincula2(T)

vincula1(puntero(T)):
      si T ≠ ref(_):
            vincula1(T)

vincula2(puntero(T)):
      si T = ref(id):
            si está_en(ts, id):
                  T.vínculo <- valor_de(ts, id)
            si no:
                  error id_no_declarado
      si no:
            vincula2(T)

// Campos de datos

vincula1(registro(Cs)):
      vincula1(Cs)

vincula1(un_campo(C)):
      vincula1(C)

vincula1(varios_campos(Cs, C)):
      vincula1(Cs)
      vincula1(C)

vincula1(campo(id, T)):
      vincula1(T)

vincula2(registro(Cs)):
      vincula2(Cs)
```

```
vincula2(un_campo(C)):
      vincula2(C)

vincula2(varios_campos(Cs, C)):
      vincula2(Cs)
      vincula2(C)

vincula2(campo(id, T)):
      vincula2(T)

// Listas de instrucciones

vincula(no_ins()): vacío

vincula(varias_ins(Is, I)):
      vincula(Is)
      vincula(I)

// Instrucciones básicas

vincula(asig(Ei, Ed)):
      vincula(Ei)
      vincula(Ed)

vincula(invoc(Eid, PRs)):
      vincula(Eid)
      vincula(PRs)

// Instrucciones de bloque

vincula(seq(Ds, Is)):
      abre_nivel(ts)
      vincula1(Ds)
      vincula2(Ds)
      vincula(Is)
      cierra_nivel(ts)

vincula(if-then(E, Is)):
      vincula(E)
      vincula(Is)

vincula(if-then-else(E, Is, Ie)):
      vincula(E)
      vincula(Is)
      vincula(Ie)
```

```
vincula(while(E, Is)):
      vincula(E)
      vincula(Is)

// Gestión de la E/S estándar

vincula(read(E)):
      vincula(E)

vincula(write(E)):
      vincula(E)

vincula(nl()): vacío

// Gestión de la memoria dinámica

vincula(new(E)):
      vincula(E)

vincula(delete(E)):
      vincula(E)

// Listas de expresiones

vincula(no_exps()): vacío

vincula(varias_exps(Es, E)):
      vincula(Es)
      vincula(E)

// Expresiones básicas

vincula(entero(lit)): vacío

vincula(real(lit)): vacío

vincula(cadena(lit)): vacío

vincula(ident(lit)):
      si está_en(ts, lit):
            $.vínculo <- valor_de(ts, lit)
      si no:
            error id_no_declarado

// Expresiones constantes
```

```
vincula(true()): vacío

vincula(false()): vacío

vincula(null()): vacío

// Operadores relacionales

vincula(eq(E1, E2)):
      vincula(E1)
      vincula(E2)

vincula(ne(E1, E2)):
      vincula(E1)
      vincula(E2)

vincula(le(E1, E2)):
      vincula(E1)
      vincula(E2)

vincula(ge(E1, E2)):
      vincula(E1)
      vincula(E2)

vincula(gt(E1, E2)):
      vincula(E1)
      vincula(E2)

vincula(lt(E1, E2)):
      vincula(E1)
      vincula(E2)

// Operadores lógicos

vincula(and(op1, op2)):
      vincula(op1)
      vincula(op2)

vincula(or(op1, op2)):
      vincula(op1)
      vincula(op2)

vincula(not(op)):
      vincula(op)

// Operadores aritméticos
```

```
vincula(suma(op1, op2)):
      vincula(op1)
      vincula(op2)

vincula(resta(op1, op2)):
      vincula(op1)
      vincula(op2)

vincula(neg(op)):
      vincula(op)

vincula(mul(op1, op2)):
      vincula(op1)
      vincula(op2)

vincula(div(op1, op2)):
      vincula(op1)
      vincula(op2)

vincula(mod(op1, op2)):
      vincula(op1)
      vincula(op2)

// Operadores posfijos de acceso a memoria

vincula(indx(Eb, Ei)):
      vincula(Eb)
      vincula(Ei)

vincula(acc(reg, id)):
      vincula(reg)

vincula(indir(Et)):
      vincula(Et)
```

3. Comprobación de tipos

```
// Métodos auxiliares

nodo_error(nodo):
      nodo.tipo <- error
      error aviso_error

ref!(T):
```

```
      mientras T = ref(_):
            T <- T.vínculo.tipo
      devuelve T

es_num(int): devuelve cierto
es_num(real): devuelve cierto
es_num(_): devuelve falso

es_ref(puntero(_)): devuelve cierto
es_ref(null): devuelve cierto
es_ref(_): devuelve falso

es_desig(id(_)): devuelve cierto
es_desig(indx(_,_)): devuelve cierto
es_desig(acc(_,_)): devuelve cierto
es_desig(indir(_)): devuelve cierto
es_desig(_): devuelve falso

tipos_compatibles(T1, T2):
      dic ts <- tabla_vacía()
      tipos_compatibles(T1, T2, ts)

tipos_compatibles(T1, T2, ts): sea R1 = ref!(T1), R2 = ref!(T2) en:

      // Compatibilidad de tipos renombrados

      par par_ref = (R1, R2)
      si está_en(ts, par_ref):
            devuelve valor_de(ts, par_ref)
      si no:
            añade(ts, ref_par, tipos_compatibles(T1, T2, ts))

      // Compatibilidad de otros tipos

      si es_num(R1) y es_num(R2):
            devuelve cierto
      si no si R1 = bool y R2 = bool:
            devuelve cierto
      si no si R1 = string y R2 = string:
            devuelve cierto
      si no si R1 = array(T1',_) y R2 = array(T2',_):
            devuelve tipos_compatibles(T1', T2', ts)
      si no si R1 = record(Cs1) y R2 = record(Cs2):
            si num_campos(Cs1) = num_campos(Cs2):
                  añade(tc, R1)
                  devuelve campos_compatibles(Cs1, Cs2, ts)
```

```
            si no:
                  devuelve falso
      si no si R1 = puntero(_) y T2 = null:
            devuelve cierto
      si no si R1 = puntero(T1') y R2 = puntero(T2'):
            devuelve tipos_compatibles(T1', T2', ts)
      si no:
            devuelve falso

num_campos(un_campo(C)): devuelve 1
num_campos(varios_campos(Cs,_)): devuelve 1 + num_campos(Cs)

campos_compatibles(un_campo(C1), un_campo(C2), ts):
      sea C1 = campo(_,T1), C2 = campo(_,T2) en:
            devuelve tipos_compatibles(T1, T2, ts)

campos_compatibles(varios_campos(Cs1, C1), varios_campos(Cs2, C2), tc):
      sea C1 = campo(_,T1), C2 = campo(_,T2) en:
            si está_en(tc, T2):
                  devuelve cierto
            si no si tipos_compatibles(T1, T2, ts):
                  devuelve campos_compatibles(Cs1, Cs2, tc)
            si no:
                  devuelve falso

ambos_ok(T1, T2, nodo)
      si T1 = ok y T2 = ok:
            nodo.tipo <- ok
      si no:
            nodo.tipo <- error

tipo_base(T, nodo):
      tipo(T)
      nodo.tipo <- T.tipo

// Programa

tipo(prog(Ds, Is)):
      tipo(Ds)
      tipo(Is)
      ambos_ok(Ds, Is, $)

// Listas de declaraciones

tipo(no_decs()): $.tipo <- ok
```

```
tipo(varias_decs(Ds, D)):
      tipo(Ds)
      tipo(D)
      ambos_ok(Ds.tipo, D.tipo, $)

// Declaraciones básicas

tipo(var(id, T)):
      tipo_base(T, $)

tipo(type(id, T)):
      tipo_base(T, $)

tipo(proc(id, PFs, Ds, Is)):
      tipo(PFs)
      tipo(Ds)
      ambos_ok(PFs.tipo, Ds.tipo, $)
      si $.tipo = ok:
            tipo(Is)
            comprueba_ins(Is, $)

// Parámetros formales

tipo(no_params()): $.tipo <- ok

tipo(varios_params(PFs, PF)):
      tipo(PFs)
      tipo(PF)
      ambos_ok(PFs.tipo, PF.tipo, $)

tipo(pvalor(id, T)):
      tipo_base(T, $)

tipo(pvar(id, T)):
      tipo_base(T, $)

// Tipos básicos

tipo(int()): $.tipo <- int

tipo(bool()): $.tipo <- bool

tipo(real()): $.tipo <- real

tipo(string()): $.tipo <- string
```

```
// Tipos renombrados

tipo(ref(id)):
      si $.vínculo = type(_,T):
            tipo_base(T, $)
      si no:
            nodo_error($)

// Tipos compuestos

tipo(array(T, tam)):
      si positivo(tam): // La cadena representa un número positivo
            tipo_base(T, $)
      si no:
            nodo_error($)

tipo(puntero(T)):
      tipo_base(T, $)

// Campos de datos

tipo(registro(Cs)):
      cjto tc <- cjto_vacío()
      para cada C = campo(id, T) en Cs:
            si está_en(tc, id):
                  $.error <- error
                  error campo_duplicado
            tipo(T)
            si T.tipo = error:
                  nodo_error($)
            añade(tc, id)

// Listas de instrucciones

tipo(no_ins()): $.tipo <- ok

tipo(varias_ins(Is, I)):
      tipo(Is)
      tipo(I)
      ambos_ok(Is.tipo, I.tipo, $)

comprueba_ins(no_ins(), nodo):
      nodo.tipo <- ok

comprueba_ins(varias_ins(Is, I), nodo):
```

```
      si I.tipo = ok:
            comprueba_ins(Is, nodo)
      si no:
            nodo_error(nodo)

// Instrucciones básicas

tipo(asig(Ei, Ed)):
      tipo(Ei)
      tipo(Ed)
      si Ei.tipo = error o Ed.tipo = error:
            $.tipo <- error
      si no si tipos_compatibles(Ei.tipo, Ed.tipo) y es_desig(Ei):
            $.tipo <- ok
      si no:
            nodo_error($)

tipo(invoc(Eid, PRs)):
      si Eid = ident(_): sea id = Eid.vínculo en:
            si id.vinculo = proc(_,PFs,_,_):
                  tipo(PRs)
                  comprueba_params(PFs, PRs, $)
                  si $.tipo = error:
                        error aviso_error
      si no:
            nodo_error($)

comprueba_params(no_params(), no_exps(), nodo):
      nodo.tipo <- ok

comprueba_params(no_params(), varias_exps(_,_), nodo):
      nodo.tipo <- error

comprueba_params(varios_params(_,_), no_exps(), nodo):
      nodo.tipo <- error

comprueba_params(varios_params(PFs, PF), varias_exps(PRs, PR), nodo):
      si PF = pvar(_,_) y no es_desig(PR) o no tipos_compatibles(PF, PR):
            nodo.tipo <- error
      comprueba_params(PFs, PRs, nodo)

// Instrucciones de bloque

tipo(seq(Ds, Is)):
      tipo(Ds)
      comprueba_ins(Is, $)
```

```
      ambos_ok(Ds.tipo, Is.tipo, $)

tipo(if-then(E, Is)):
      tipo(E)
      si E.tipo = error:
            $.tipo <- error
      si no si ref!(E.tipo) = bool:
            tipo(Is)
            comprueba_ins(Is, $)
      si no:
            nodo_error($)

tipo(if-then-else(E, Is, Ie)):
      tipo(E)
      si E.tipo = error:
            $.tipo <- error
      si no si ref!(E.tipo) = bool:
            tipo(Is)
            comprueba_ins(Is, $)
            si $.tipo = ok:
                  tipo(Ie)
                  comprueba_ins(Ie, $)
      si no:
            nodo_error($)

tipo(while(E, Is)):
      tipo(E)
      si E.tipo = error:
            $.tipo <- error
      si no si ref!(E.tipo) = bool:
            tipo(Is)
            comprueba_ins(Is, $)
      si no:
            nodo_error($)

// Gestión de la E/S estándar

tipo(read(E)):
      tipo(E)
      si E.tipo = error:
            $.tipo <- error
      si no si es_desig(E) y ref!(E.tipo) en { int, real, string }:
            $.tipo <- ok
      si no:
            nodo_error($)
```

```
tipo(write(E)):
      tipo(E)
      si E.tipo = error:
            $.tipo <- error
      si no si ref!(E.tipo) en { int, real, bool, string }:
            $.tipo <- ok
      si no:
            nodo_error($)

tipo(nl()): $.tipo <- ok

// Gestión de la memoria dinámica

tipo(new(E)):
      tipo(E)
      si E.tipo = error:
            $.tipo <- error
      si no si ref!(E.tipo) = puntero(_):
            $.tipo <- ok
      si no:
            nodo_error($)

tipo(delete(E)):
      tipo(E)
      si E.tipo = error:
            $.tipo <- error
      si no si ref!(E.tipo) = puntero(_):
            $.tipo <- ok
      si no:
            nodo_error($)

// Listas de expresiones

tipo(no_exps()): $.tipo <- ok

tipo(varias_exps(Es, E)):
      tipo(Es)
      tipo(E)
      ambos_ok(Es.tipo, E.tipo)

// Expresiones básicas

tipo(entero(lit)): $.tipo <- int

tipo(real(lit)): $.tipo <- real
```

```
tipo(cadena(lit)): $.tipo <- string

tipo(ident(lit)): sea $.vínculo = D en:
      si D = var(id, T) o D = pvalor(id, T) o D = pvar(id, T):
            $.tipo <- T
      si no:
            nodo_error($)

// Expresiones constantes

tipo(true()): $.tipo <- bool

tipo(false()): $.tipo <- bool

tipo(null()): $.tipo <- null

// Operadores relacionales

tipo(eq(E1, E2)):
      tipo(E1)
      tipo(E2)
      sea R1 = ref!(E1.tipo), R2 = ref!(E2.tipo) en:
            si R1.tipo = error o R2.tipo = error:
                  $.tipo <- error
            si no si es_num(R1) y es_num(R2):
                  $.tipo <- bool
            si no si R1 = bool y R2 = bool:
                  $.tipo <- bool
            si no si R1 = string y R2 = string:
                  $.tipo <- bool
            si no si es_ref(R1) y es_ref(R2):
                  $.tipo <- ok
            si no:
                  nodo_error($)

tipo(ne(E1, E2)):
      tipo(E1)
      tipo(E2)
      sea R1 = ref!(E1.tipo), R2 = ref!(E2.tipo) en:
            si R1.tipo = error o R2.tipo = error:
                  $.tipo <- error
            si no si es_num(R1) y es_num(R2):
                  $.tipo <- bool
            si no si R1 = bool y R2 = bool:
                  $.tipo <- bool
            si no si R1 = string y R2 = string:
```

```
                $.tipo <- bool
          si no:
                nodo_error($)

tipo(le(E1, E2)):
      tipo(E1)
      tipo(E2)
      sea R1 = ref!(E1.tipo), R2 = ref!(E2.tipo) en:
            si R1.tipo = error o R2.tipo = error:
                $.tipo <- error
            si no si es_num(R1) y es_num(R2):
                $.tipo <- bool
            si no si R1 = bool y R2 = bool:
                $.tipo <- bool
            si no si R1 = string y R2 = string:
                $.tipo <- bool
            si no:
                nodo_error($)

tipo(ge(E1, E2)):
      tipo(E1)
      tipo(E2)
      sea R1 = ref!(E1.tipo), R2 = ref!(E2.tipo) en:
            si R1.tipo = error o R2.tipo = error:
                $.tipo <- error
            si no si es_num(R1) y es_num(R2):
                $.tipo <- bool
            si no si R1 = bool y R2 = bool:
                $.tipo <- bool
            si no si R1 = string y R2 = string:
                $.tipo <- bool
            si no:
                nodo_error($)

tipo(gt(E1, E2)):
      tipo(E1)
      tipo(E2)
      sea R1 = ref!(E1.tipo), R2 = ref!(E2.tipo) en:
            si R1.tipo = error o R2.tipo = error:
                $.tipo <- error
            si no si es_num(R1) y es_num(R2):
                $.tipo <- bool
            si no si R1 = bool y R2 = bool:
                $.tipo <- bool
            si no si R1 = string y R2 = string:
                $.tipo <- bool
```

```
            si no:
                  nodo_error($)

tipo(lt(E1, E2)):
      tipo(E1)
      tipo(E2)
      sea R1 = ref!(E1.tipo), R2 = ref!(E2.tipo) en:
            si R1.tipo = error o R2.tipo = error:
                  $.tipo <- error
            si no si es_num(R1) y es_num(R2):
                  $.tipo <- bool
            si no si R1 = bool y R2 = bool:
                  $.tipo <- bool
            si no si R1 = string y R2 = string:
                  $.tipo <- bool
            si no:
                  nodo_error($)

// Operadores lógicos

tipo(and(op1, op2)):
      tipo(op1)
      tipo(op2)
      sea R1 = ref!(op1.tipo), R2 = ref!(op2.tipo) en:
            si R1.tipo = error o R2.tipo = error:
                  $.tipo <- error
            si no si R1 = bool y R2 = bool:
                  $.tipo <- bool
            si no:
                  nodo_error($)

tipo(or(op1, op2)):
      tipo(op1)
      tipo(op2)
      sea R1 = ref!(op1.tipo), R2 = ref!(op2.tipo) en:
            si R1.tipo = error o R2.tipo = error:
                  $.tipo <- error
            si no si R1 = bool y R2 = bool:
                  $.tipo <- bool
            si no:
                  nodo_error($)

tipo(not(op)):
      tipo(op)
      sea R1 = ref!(op.tipo) en:
            si R1.tipo = error:
```

```
                $.tipo <- error
            si no si R1 = bool:
                $.tipo <- bool
            si no:
                nodo_error($)

// Operadores aritméticos

tipo(suma(op1, op2)):
      tipo(op1)
      tipo(op2)
      sea R1 = ref!(op1.tipo), R2 = ref!(op2.tipo) en:
            si R1.tipo = error o R2.tipo = error:
                $.tipo <- error
            si no si R1 = int y R2 = int:
                $.tipo <- int
            si no si es_num(R1) y es_num(R2):
                $.tipo <- real
            si no:
                nodo_error($)

tipo(resta(op1, op2)):
      tipo(op1)
      tipo(op2)
      sea R1 = ref!(op1.tipo), R2 = ref!(op2.tipo) en:
            si R1.tipo = error o R2.tipo = error:
                $.tipo <- error
            si no si R1 = int y R2 = int:
                $.tipo <- int
            si no si es_num(R1) y es_num(R2):
                $.tipo <- real
            si no:
                nodo_error($)

tipo(neg(op)):
      tipo(op)
      sea R1 = ref!(op.tipo) en:
            si R1.tipo = error:
                $.tipo <- error
            si no si R1 = int:
                $.tipo <- int
            si no si R1 = real:
                $.tipo <- real
            si no:
                nodo_error($)
```

```
tipo(mul(op1, op2)):
      tipo(op1)
      tipo(op2)
      sea R1 = ref!(op1.tipo), R2 = ref!(op2.tipo) en:
            si R1.tipo = error o R2.tipo = error:
                  $.tipo <- error
            si no si R1 = int y R2 = int:
                  $.tipo <- int
            si no si es_num(R1) y es_num(R2):
                  $.tipo <- real
            si no:
                  nodo_error($)

tipo(div(op1, op2)):
      tipo(op1)
      tipo(op2)
      sea R1 = ref!(op1.tipo), R2 = ref!(op2.tipo) en:
            si R1.tipo = error o R2.tipo = error:
                  $.tipo <- error
            si no si R1 = int y R2 = int:
                  $.tipo <- int
            si no si es_num(R1) y es_num(R2):
                  $.tipo <- real
            si no:
                  nodo_error($)

tipo(mod(op1, op2)):
      tipo(op1)
      tipo(op2)
      sea R1 = ref!(op1.tipo), R2 = ref!(op2.tipo) en:
            si R1.tipo = error o R2.tipo = error:
                  $.tipo <- error
            si no si R1 = int y R2 = int:
                  $.tipo <- int
            si no:
                  nodo_error($)

// Operadores de acceso a memoria

tipo(indx(Eb, Ei)):
      tipo(Eb)
      tipo(Ei)
      sea Ebr = ref!(Eb.tipo), Eir = ref!(Ei.tipo) en:
            si Ebr.tipo = error o Eir.tipo = error:
                  $.tipo <- error
            si no si Ebr = array(T,_) y Eir = int:
```

```
                $.tipo <- T
          si no:
                nodo_error($)

tipo(acc(reg, id)):
      tipo(reg)
      sea Er = ref!(reg.tipo) en:
            si reg.tipo = error o Er.tipo = error:
                $.tipo <- error
            si no si Er = record(Cs):
                comprueba_campos(Cs, id, $)
            si no:
                nodo_error($)

comprueba_campo(campo(cid, T), id, nodo):
      si cid = id:
            tipo_base(T, $)

comprueba_campos(un_campo(C), id, nodo):
      comprueba_campo(C, id, nodo)
      si nodo.tipo = indef:
            nodo_error($)

comprueba_campos(varios_campos(Cs, C), id, nodo):
      comprueba_campo(C, id, nodo)
      si nodo.tipo = indef:
            comprueba_campos(Cs, id, nodo)

tipo(indir(Et)):
      tipo(Et)
      sea Er = ref!(Et.tipo) en:
            si Er.tipo = error:
                $.tipo <- error
            si no si Er = puntero(T):
                tipo_base(T, $)
            si no:
                nodo_error($)
```

4. Asignación de espacio

```
global ent dir <- 0 // Contador de direcciones
global ent nivel <- 0 // Nivel de anidamiento

// Programa
```

```
asigna_espacio(prog(Ds, Is)):
      asigna_espacio(Ds)
      asigna_espacio(Is)

// Listas de declaraciones

asigna_espacio(no_decs()): vacio

asigna_espacio(varias_decs(Ds, D)):
      asigna_espacio(Ds)
      asigna_espacio(D)

// Declaraciones básicas

asigna_espacio(var(id, T)):
      $.dir <- dir
      $.nivel <- nivel
      asigna_espacio(T)
      dir <- dir + T.tam

asigna_espacio(type(id, T)):
      asigna_espacio(T)

asigna_espacio(proc(id, PFs, Ds, Is)):
      $.dir <- dir
      ent prev_dir <- dir
      nivel <- nivel + 1
      $.nivel <- nivel

      dir <- 0
      asigna_espacio(PFs)
      asigna_espacio(Ds)
      asigna_espacio(Is)

      nivel <- nivel - 1
      dir <- prev_dir

// Parámetros formales

asigna_espacio(no_params()): vacío

asigna_espacio(varios_params(PFs, PF)):
      asigna_espacio(PFs)
      asigna_espacio(PF)

asigna_espacio(pvalor(id, T)):
```

```
      $.dir <- dir
      $.nivel <- nivel
      asigna_espacio(T)
      dir <- dir + T.tam

asigna_espacio(pvar(id, T)):
      $.dir <- dir
      $.nivel <- nivel
      asigna_espacio(T)
      dir <- dir + 1

// Tipos

asigna_espacio(T):
      si T.espacio = indef:
            asigna_espacio1(T)
            asigna_espacio2(T)

// Tipos básicos

asigna_espacio1(int()): $.tam <- 1
asigna_espacio2(int()): vacío

asigna_espacio1(bool()): $.tam <- 1
asigna_espacio2(bool()): vacío

asigna_espacio1(real()): $.tam <- 1
asigna_espacio2(real()): vacío

asigna_espacio1(string()): $.tam <- 1
asigna_espacio2(string()): vacío

// Tipos compuestos

asigna_espacio1(array(T, tam)): sea t = valor_de(tam) en:
      si t ≥ 0:
            asigna_espacio(T)
            $.tam <- T.tam * t
      si no:
            error aviso_error

asigna_espacio2(array(T, tam)): vacío

asigna_espacio1(puntero(T)):
      $.tam <- 1
      si T ≠ ref(_):
```

```
            asigna_espacio1(T)

asigna_espacio2(puntero(T)):
      si T = ref(_):
            asigna_espacio(T)
      si no:
            asigna_espacio2(T)

asigna_espacio1(registro(Cs)):
      $.tam <- 0
      para cada C = campo(_,T) en Cs:
            C.desp <- $.tam // Asignación del desplazamiento
            asigna_espacio1(T)
            $.tam <- $.tam + T.tam // Acumulamos el tamaño de cada campo

asigna_espacio2(registro(Cs)):
      para cada C = campo(_,T) en Cs:
            asigna_espacio2(T)

// Listas de instrucciones

asigna_espacio(no_ins()): vacío

asigna_espacio(varias_ins(Is, I)):
      vincula(Is)
      vincula(I)

// Instrucciones de bloque

asigna_espacio(seq(Ds, Is)):
      asigna_espacio(Ds)
      asigna_espacio(Is)

asigna_espacio(if-then(E, Is)):
      asigna_espacio(Is)

asigna_espacio(if-then-else(E, Is, Ie)):
      asigna_espacio(Is)
      asigna_espacio(Ie)

asigna_espacio(while(E, Is)):
      asigna_espacio(Is)

asigna_espacio(_): vacío // No se asigna espacio para las restantes
```

5. Repertorio de instrucciones de la máquina P

El repertorio de instrucciones de la máquina P incluye todas las documentadas en el Campus Virtual de la asignatura junto con las instrucciones de apilado de tipos básicos, las cuales tienen la siguiente forma: *apilaX*(*e*), donde *e* es el elemento a añadir a la pila, el cual es de un tipo básico denotado por *X*. Este repertorio también incluye las instrucciones homólogas a los operadores aritméticos, lógicos y de comparación aplicables a expresiones del lenguaje, cuya ejecución consiste en desapilar los operandos requeridos de la pila de operaciones y apilar el resultado de la operación aplicada sobre ellos, siempre y cuando todos estos valores existan (i. e.: el número de elementos de la pila es mayor o igual que la aridad de la operación) y sean del tipo adecuado, según las reglas de tipo del lenguaje.

Adicionamente, empleamos nuevas operaciones que vienen descritas en la siguiente tabla, donde se proporciona una implementación en lenguaje C de cada una de ellas. Dichas implementaciones hacen uso de las siguientes definiciones:

Definiciones del código en C

```
enum TIPO {
	NUM, // Tipo numérico: entero o real
	BOOL, // Tipo booleano
	CHAR, // Tipo carácter
	STRING // Tipo cadena de caracteres
};

struct nodo {
	TIPO tipo;
	void* val // Si tipo = OP, apunta a un entero con el código de operación (opcode)
};

// Devuelve la cima de la pila, apuntada por 'n' tras la ejecución de esta función
void cima(struct nodo* n);

// Devuelve la representación literal del valor del nodo 'n', apuntada por 'str' tras la
// ejecución de esta función
void toString(struct nodo* n, char* str);

// Elimina la cima de la pila, si existe. Si la pila es vacía, esta función tiene ningún
// efecto sobre ella
void desapila();

// Escribe un mensaje de error que incluye la cadena 'mensaje' por la salida de error
// (stderr) y aborta la ejecución del programa
void errorPila(const char* mensaje);
```

Instrucciones	Descripción	Implementación C de ejemplo
`promreal()`	Si hay un valor numérico (i. e.:	`struct nodo top;` `cima(&top);`

	entero o real) en la cima de la pila de operaciones, lo desapila y apila el número real equivalente a dicho valor. Nótese que la operación no tiene ningún efecto si el valor de la cima es un número real.
	
	Si la pila está vacía o la cima no es un valor numérico, la operación falla y se lanza una excepción.
		`if (top.tipo == NUM)`
		`{`
		`  desapila();`
		`  apilareal(top->val);`
		`}`
		`else`
		`  errorPila("Cima no numérica");`

`escanea()`	Lee una línea de texto introducida por la entrada estándar y la añade a la pila de operaciones como cadena de caracteres.	`char c = getc(stdin);`
		`while (!feof(stdin) && c!='\n')`
		`{`
		`    apilachar(c);`
		`    c = getc(stdin);`
		`}`

`imprime()`	Desapila e imprime el valor de la cima de la pila.	`char* str;`
		`struct nodo top;`
		`cima(&top);`
		`toString(&top, str);`
		`printf("%s", str);`
		`desapila();`
		`free(str);`

`endl()`[1]	Imprime un salto de línea en la salida estándar.	`printf("\n");` `fflush(stdout);`

6. Generación de código

```
global cola procs <- cola_vacia()
global ent cte NULL <- -1 // Constante para representar un puntero nulo

// Métodos auxiliares

tam_base(E): sea T = ref!(E.tipo) en:
      devuelve T.tipo.tam

comprueba_nulo(sig):
      linea_código(dup())
      línea_código(apilaint(NULL))
      línea_código(eq)
      línea_código(irf(sig)) // Saltamos a apilaind()
      línea_código(stop)
      error aviso_error

extrae_primero(cola):
```

[1] Diseño de la instrucción extraída de la norma ISO del lenguaje C++ (ISO/IEC 14882).

```
      p <- primero(cola)
      desencolar(cola)
      devuelve p

// Programa

genera_código(prog(Ds, Is)):
      genera_código(Is)
      línea_código(stop())
      recolecta_procs(Ds)
      mientras no es_vacia():
            genera_código(extrae_primero(procs))

recolecta_procs(no_decs()): vacío

recolecta_procs(varias_decs(Ds, D)):
      encola_procs(Ds)
      si D = proc(_,_,_,_):
            encola(procs, D)

// Declaraciones básicas

genera_código(proc(id, PFs, Ds, Is)):
      genera_código(Is)
      línea_código(desactiva($.nivel, $.tam))
      línea_código(irind())

// Listas de instrucciones

genera_código(no_ins()): vacío

genera_código(varias_ins(Is, I)):
      genera_código(Is)
      genera_código(I)

// Instrucciones básicas

genera_código(asig(Ei, Ed)):
      genera_código(Ei)
      genera_código(Ed)
      si Ei.tipo = real y Ed.tipo = int:
            si es_desig(Ed):
                  línea_código(apilaind())
            línea_código(promreal())
            línea_código(desapilaind())
      si no si es_desig(Ed): sea T = Ei.tipo en:
```

```
            línea_código(mueve(T.tam))
      si no:
            línea_código(desapilaind())

genera_código(invoc(Eid, PRs)):
      sea id = Eid.vinculo, proc = id.vinculo = proc(_,PFs,_,_) en:
            línea_código(activa($.vínculo.nivel, $.vínculo.tam, $.sig))
            paso_params(PFs, PRs)
            línea_código(desapilad(proc.nivel))
            línea_código(ira(proc.inicio))

paso_params(no_params(), no_exps()): vacío

paso_params(varios_params(PFs, PF), varias_exps(PRs, PR)):
      paso_param(PF, PR)
      paso_params(PFs, PRs)

paso_param(PF, PR):
      línea_código(dup())
      línea_código(apilaint(PF.dir))
      línea_código(suma())
      genera_código(PR)
      si PF = pvalor(_,T):
            si PF.tipo = real y PR.tipo = int:
                  // Promocionamos el entero a real
                  línea_código(promreal())
            si no si es_desig(PR):
                  línea_código(mueve(T.tam))
            si no:
                  línea_código(desapilaind())
      si no:
            línea_código(desapilaind())

// Instrucciones de bloque

genera_código(seq(Ds, Is)):
      genera_código(Is)

genera_código(if-then(E, Is)):
      genera_código(E)
      si es_desig(E):
            línea_código(apilaind())
      línea_código(irf($.sig))
      genera_código(Is)

genera_código(if-then-else(E, Is, Ie)):
```

```
      genera_código(E)
      si es_desig(E):
            línea_código(apilaind())
      línea_código(irf(Ie.inicio))
      genera_código(Is)
      línea_código(ir($.sig))
      genera_código(Ie)

genera_código(while(E, Is)):
      genera_código(E)
      si es_desig(E):
            línea_código(apilaind())
      línea_código(irf($.sig))
      genera_código(Is)
      línea_código(ir($.inicio))

// Gestión de la E/S estándar

genera_código(read(E)):
      genera_código(E)
      línea_código(scan(E))
      línea_código(desapilaind())

genera_código(write(E)):
      genera_código(E)
      si es_desig(E):
            línea_código(apilaind())
      línea_código(print())

genera_código(nl()): línea_código(endl())

// Gestión de la memoria dinámica

genera_código(new(E)):
      genera_código(E)
      línea_código(alloc(tam_base(E)))
      línea_código(desapilaind())

genera_código(delete(E)):
      genera_código(E)
      línea_código(apilaind())
      comprobar_nulo($, $.sig - 1)
      línea_código(dealloc(tam_base(E)))

// Listas de expresiones
```

```
genera_código(no_exps()): vacío

genera_código(varias_exps(Es, E)):
      genera_código(Es)
      genera_código(E)

// Expresiones básicas

genera_código(entero(lit)): línea_código(apilaint(lit))

genera_código(real(lit)): línea_código(apilareal(lit))

genera_código(cadena(lit)): línea_código(apilastring(lit))

genera_código(ident(lit)):
      si $.vínculo.nivel = 0:
            línea_código(apilaint($.vínculo.dir))
      si no:
            línea_código(apilad($.vínculo.nivel))
            línea_código(apilaint($.vínculo.dir))
            línea_código(suma())
            si $.vínculo = pvar(_,_):
                  línea_código(apilaind())

// Expresiones constantes

genera_código(true()): línea_código(apilabool(true))

genera_código(false()): línea_código(apilabool(false))

genera_código(null()): línea_código(apilaint(NULL))

// Operadores relacionales

genera_código(eq(E1, E2)): op_binaria(eq, E1, E2, $)

genera_código(ne(E1, E2)): op_binaria(ne, E1, E2, $)

genera_código(le(E1, E2)): op_binaria(le, E1, E2, $)

genera_código(lt(E1, E2)): op_binaria(lt, E1, E2, $)

genera_código(ge(E1, E2)): op_binaria(ge, E1, E2, $)

genera_código(gt(E1, E2)): op_binaria(gt, E1, E2, $)
```

```
// Operadores lógicos

genera_código(and(op1, op2)): op_binaria(and, op1, op2, $)

genera_código(or(op1, op2)): op_binaria(or, op1, op2, $)

genera_código(not(op)): op_unaria(not, op)

// Operadores aritméticos

genera_código(suma(op1, op2)): op_binaria(suma, op1, op2, $)

genera_código(resta(op1, op2)): op_binaria(resta, op1, op2, $)

genera_código(mul(op1, op2)): op_binaria(mul, op1, op2, $)

genera_código(div(op1, op2)): op_binaria(div, op1, op2, $)

genera_código(mod(op1, op2)): op_binaria(mod, op1, op2, $)

genera_código(neg(op)): op_unaria(neg, op)

op_unaria(op, E):
      genera_código(E)
      si es_desig(E):
            línea_código(apilaind())
      línea_código(op)

op_binaria(op, E1, E2, nodo):
      sea T = ref!(nodo.tipo), R1 = ref!(E1.tipo), R2 = ref!(E2.tipo) en:
            genera_código(E1)
            si es_desig(E1):
                  línea_código(apilaind())
            si T = real y R1 = int:
                  línea_código(promreal())
            genera_código(E2)
            si es_desig(E2):
                  línea_código(apilaind())
            si T = real y R2 = int:
                  línea_código(promreal())
            línea_código(op)

// Operadores de acceso a memoria

genera_código(indx(Eb, Ei)):
      genera_código(Eb)
```

```
      genera_código(Ei)
      si es_desig(Ei):
            línea_código(apilaind())
      línea_código(apilaint(tam_base(Eb)))
      línea_código(mul())
      línea_código(suma())

despl(no_campos(), id): devuelve 0

despl(varios_campos(Cs, C), id): sea C = campo(cid,_) en:
      si cid = id:
            devuelve C.desp
      si no:
            devuelve despl(Cs, id)

genera_código(acc(reg, id)): sea ref!(reg) = record(Cs) en:
      genera_código(reg)
      línea_código(apilaint(despl(reg, id)))
      línea_código(suma())

genera_código(indir(Et)):
      genera_código(Et)
      comprueba_nulo($.sig - 1)
      línea_código(apilaind())

genera_código(_): vacío // No se genera código para las restantes
```

7. Etiquetado

```
global cola procs <- cola_vacia()
global ent tag <- 0

// Programa

etiqueta(prog(Ds, Is)):
      $.inicio <- tag
      etiqueta(Is)
      tag <- tag + 1
      recolecta_procs(Ds)
      mientras no es_vacia():
            etiqueta(extrae_primero(procs))
      $.sig <- tag

encola_procs(no_decs()): vacío
```

```
encola_procs(varias_decs(Ds, D)):
      encola_procs(Ds)
      si D = proc(_,_,_,_):
            encola(proc, D)

// Declaraciones básicas

etiquetado(proc(id, PFs, Ds, Is)):
      $.inicio <- tag
      etiquetado(Is)
      tag <- tag + 2
      $.sig <- tag

// Listas de instrucciones

etiqueta(no_ins()):
      $.inicio <- tag
      $.sig <- tag

etiqueta(varias_ins(Is, I)):
      $.inicio <- tag
      etiqueta(Is)
      etiqueta(I)
      $.sig <- tag

// Instrucciones básicas

etiqueta(asig(Ei, Ed)):
      $.inicio <- tag
      etiqueta(Ei)
      etiqueta(Ed)
      si Ei.vínculo.tipo = real y Ed.vínculo.tipo = int:
            si es_desig(Ed):
                  tag <- tag + 1
            tag <- tag + 2
      si no:
            tag <- tag + 1
      $.sig <- tag

etiqueta(invoc(Eid, PRs)):
      sea id = Eid.vinculo, id.vinculo = proc(_,PFs,_,_) en:
            $.inicio <- tag
            tag <- tag + 1
            paso_params(PFs, PRs)
            tag <- tag + 2
            $.sig <- tag
```

```
paso_params(no_params(), no_exps()): vacío

paso_params(varios_params(PFs, PF), varias_exps(PRs, PR)):
      paso_param(PF, PR)
      paso_params(PFs, PRs)

paso_param(PF, PR):
      tag <- tag + 3
      etiqueta(PR)
      si PF = pvalor(_,T):
            si PF.tipo = real y PR.tipo = int:
                  tag <- tag + 1
            tag <- tag + 1
      si no:
            tag <- tag + 1

// Instrucciones de bloque

etiqueta(seq(Ds, Is)):
      $.inicio <- tag
      etiqueta(Is)
      $.sig <- tag

etiqueta(if-then(E, Is)):
      $.inicio <- tag
      etiqueta(E)
      si es_desig(E):
            tag <- tag + 1
      tag <- tag + 1
      etiqueta(Is)
      $.sig <- tag

etiqueta(if-then-else(E, Is, Ie)):
      $.inicio <- tag
      etiqueta(E)
      si es_desig(E):
            tag <- tag + 1
      tag <- tag + 1
      etiqueta(Is)
      tag <- tag + 1
      etiqueta(Ie)
      $.sig <- tag

etiqueta(while(E, Is)):
      $.inicio <- tag
```

```
      etiqueta(E)
      si es_desig(E):
            tag <- tag + 1
      tag <- tag + 1
      etiqueta(Is)
      tag <- tag + 1
      $.sig <- tag

// Gestión de la E/S estándar

etiqueta(read(E)):
      $.inicio <- tag
      etiqueta(E)
      tag <- tag + 2
      $.sig <- tag

etiqueta(write(E)):
      $.inicio <- tag
      etiqueta(E)
      si es_desig(E):
            tag <- tag + 1
      tag <- tag + 1
      $.sig <- tag

etiqueta(nl()):
      $.inicio <- tag
      tag <- tag + 1
      $.sig <- tag

// Gestión de la memoria dinámica

etiqueta(new(E)):
      $.inicio <- tag
      etiqueta(E)
      tag <- tag + 2
      $.sig <- tag

etiqueta(delete(E)):
      $.inicio <- tag
      etiqueta(E)
      tag <- tag + 7
      $.sig <- tag

// Listas de expresiones

etiqueta(no_exps()):
```

```
      $.inicio <- tag
      $.sig <- tag

etiqueta(varias_exps(Es, E)):
      $.inicio <- tag
      etiqueta(Es)
      etiqueta(E)
      $.sig <- tag

// Expresiones básicas

etiqueta(entero(lit)):
      $.inicio <- tag
      tag <- tag + 1
      $.sig <- tag

etiqueta(real(lit)):
      $.inicio <- tag
      tag <- tag + 1
      $.sig <- tag

etiqueta(cadena(lit)):
      $.inicio <- tag
      tag <- tag + 1
      $.sig <- tag

etiqueta(ident(lit)):
      $.inicio <- tag
      si $.vínculo.nivel = 0:
            tag <- tag + 1
      si no:
            tag <- tag + 3
            si $.vínculo = pvar(_,_):
                  tag <- tag + 1
      $.sig <- tag

// Expresiones constantes

etiqueta(true()):
      $.inicio <- tag
      tag <- tag + 1
      $.sig <- tag

etiqueta(false()):
      $.inicio <- tag
      tag <- tag + 1
```

```
      $.sig <- tag

etiqueta(null()):
      $.inicio <- tag
      tag <- tag + 1
      $.sig <- tag

// Operadores relacionales

etiqueta(eq(E1, E2)): op_binaria(eq, E1, E2, $)

etiqueta(ne(E1, E2)): op_binaria(ne, E1, E2, $)

etiqueta(le(E1, E2)): op_binaria(le, E1, E2, $)

etiqueta(lt(E1, E2)): op_binaria(lt, E1, E2, $)

etiqueta(ge(E1, E2)): op_binaria(ge, E1, E2, $)

etiqueta(gt(E1, E2)): op_binaria(gt, E1, E2, $)

// Operadores lógicos

etiqueta(and(op1, op2)): op_binaria(and, op1, op2, $)

etiqueta(or(op1, op2)): op_binaria(or, op1, op2, $)

etiqueta(not(op)): op_unaria(not, op, $)

// Operadores aritméticos

etiqueta(suma(op1, op2)): op_binaria(suma, op1, op2, $)

etiqueta(resta(op1, op2)): op_binaria(resta, op1, op2, $)

etiqueta(mul(op1, op2)): op_binaria(mul, op1, op2, $)

etiqueta(div(op1, op2)): op_binaria(div, op1, op2, $)

etiqueta(mod(op1, op2)): op_binaria(mod, op1, op2, $)

etiqueta(neg(op)): op_unaria(neg, op, $)

op_binaria(op, E1, E2, nodo):
      sea T = ref!(nodo.tipo), R1 = ref!(E1.tipo), R2 = ref!(E2.tipo) en:
            nodo.inicio <- tag
```

```
            etiqueta(E1)
            si es_desig(E1):
                  tag <- tag + 1
            si T = real y R1 = int:
                  tag <- tag + 1
            etiqueta(E2)
            si es_desig(E2):
                  tag <- tag + 1
            si T = real y R2 = int:
                  tag <- tag + 1
            tag <- tag + 1
      nodo.sig <- tag

op_unaria(op, E, nodo)
      nodo.inicio <- tag
      etiqueta(E)
      si es_desig(E):
            tag <- tag + 1
      tag <- tag + 1
      nodo.sig <- tag

etiqueta(indx(Eb, Ei)):
      $.inicio <- tag
      etiqueta(Eb)
      etiqueta(Ei)
      si es_desig(Ei):
            tag <- tag + 1
      tag <- tag + 3
      $.sig <- tag

etiqueta(acc(reg, id)):
      $.inicio <- tag
      etiqueta(reg)
      tag <- tag + 2
      $.sig <- tag

etiqueta(indir(Et)):
      $.inicio <- tag
      etiqueta(Et)
      tag <- tag + 5
      $.sig <- tag

etiqueta(_): vacío
```