

# Problemas de Procesadores de Lenguaje

Juan Antonio Pérez Ortiz, japerez[en]dlsi.ua.es

Departament de Llenguatges i Sistemes Informàtics  
Universitat d'Alacant

Diciembre de 2006\*

Esta obra está bajo una licencia Reconocimiento-NoComercial 2.5 Spain de Creative Commons. Para ver una copia de esta licencia, visite <http://creativecommons.org/licenses/by-nc/2.5/es/> o envíe una carta a Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.

Este documento recopila principalmente los problemas de los exámenes de las asignaturas Compiladores 1, Compiladores 2 (plan de estudios 1993) y Procesadores de Lenguaje (plan de estudios 2001) de Ingeniería en Informática de la Universidad de Alicante realizados entre los años 2002 y 2006. Los problemas se ajustan también a los contenidos del libro “Diseño de compiladores” (A. Garrido Alenda, J. M. Iñesta Quereda, F. Moreno Seco y J. A. Pérez Ortiz, 2002, Publicaciones de la Universidad de Alicante). Los enunciados de los problemas podrían contener algunas erratas, que se irán corrigiendo en revisiones posteriores del documento.

## Análisis léxico

✳ **1** Dada la especificación léxica siguiente, diseña el autómata finito (o el diagrama de transiciones) necesario para analizar léxicamente la entrada.

EXPRESIÓN REGULAR	ELEMENTO LÉXICO
*	<b>todos</b>
*.*	<b>conextension</b>
/	<b>raiz</b>
.	<b>actual</b>

Después indica claramente la segmentación en *tokens* que realizaría tu analizador léxico sobre la siguiente cadena de entrada: ./\*.\*.\*

\* Versiones anteriores de este documento: febrero de 2006.

Indica, asimismo, qué caracteres son devueltos a la entrada (si los hay) por el analizador léxico antes de retornar cada uno de los *tokens*.

\* **2** Comenta brevemente en qué consiste el criterio de la subcadena más larga, que se suele seguir en la mayoría de analizadores léxicos.

\* **3** Dada la especificación léxica siguiente, diseña el autómata finito necesario para analizar léxicamente la entrada.

EXPRESIÓN REGULAR	ELEMENTO LÉXICO
<>	<b>distinto</b>
<	<b>menor</b>
>	<b>mayor</b>
+	<b>opsuma</b>
-	<b>opsuma</b>
++	<b>incremento</b>
--	<b>decremento</b>
->	<b>desref</b>
--->	<b>dobleref</b>
:=	<b>asig</b>

Después indica claramente la segmentación en *tokens* que realizaría tu analizador léxico sobre la siguiente cadena de entrada, donde el símbolo `␣` representa un espacio en blanco y el símbolo `↵` representa un salto de línea:

```
␣<<>++++>␣----->↵
:=:-␣----->
```

Indica, asimismo, qué caracteres son devueltos a la entrada (si los hay) por el analizador léxico antes de retornar cada uno de los *tokens*.

\* **4** Dada la especificación léxica siguiente, diseña el autómata finito necesario para analizar léxicamente la entrada.

EXPRESIÓN REGULAR	ELEMENTO LÉXICO
>	<b>mayor</b>
+ -	<b>masmenos</b>
+	<b>opsuma</b>
-	<b>opsuma</b>
++	<b>incremento</b>
--	<b>decremento</b>
->	<b>desref</b>
--->	<b>dobleref</b>

Después, indica claramente la segmentación en *tokens* que realizaría un analizador léxico sobre la siguiente cadena de entrada:

>+-+--->>+--->

## Sintaxis

\* **5** Dada la siguiente gramática, construye un esquema de traducción dirigida por la sintaxis (ETDS) que cuente e imprima el número de símbolos **a** que aparecen en la entrada *justo a continuación* de un símbolo **e**.

$$\begin{aligned}
 S &\longrightarrow AB \\
 A &\longrightarrow \mathbf{a} e A \\
 A &\longrightarrow \mathbf{a} \mathbf{a} \mathbf{b} A \\
 A &\longrightarrow \epsilon \\
 B &\longrightarrow \mathbf{d} e \mathbf{a} e \mathbf{a} \\
 B &\longrightarrow \mathbf{d} e \mathbf{a} e
 \end{aligned}$$

Por ejemplo, ante una entrada como *aeaeaabdeaea*, el ETDS debe imprimir el valor 4.

Toda la información debe pasarse a través de atributos y no es posible utilizar ninguna variable global. Debes indicar claramente, además, de qué tipo es cada uno de los atributos que utilizas y cuál es su cometido.

\* **6** Demuestra que la siguiente gramática es ambigua y diseña una gramática no ambigua equivalente:

$$\begin{aligned}
 S &\longrightarrow TV \\
 T &\longrightarrow \mathbf{t} \mid \epsilon \\
 V &\longrightarrow \mathbf{v} \mid \mathbf{t} \mid \epsilon
 \end{aligned}$$

\* **7** Diseña una gramática no ambigua para el lenguaje de las cadenas balanceadas de paréntesis en las que el número de paréntesis de apertura (o de clausura) es impar. La cadena vacía ha de pertenecer al lenguaje. Ejemplos de entradas válidas para la gramática son:

((()))      ()      (((((((()())))))

\* **8** Demuestra que la siguiente gramática es ambigua y diseña una gramática no ambigua equivalente:

$$\begin{aligned} S &\longrightarrow T V S \mid \mathbf{s} \\ T &\longrightarrow \mathbf{t} \mid \epsilon \\ V &\longrightarrow \mathbf{v} \mid \epsilon \end{aligned}$$

## Análisis sintáctico descendente

\* **9** Demuestra que la siguiente gramática es LL(1) y construye su tabla de análisis sintáctico LL(1).

$$\begin{aligned} S &\longrightarrow \otimes \square S \mid B \\ B &\longrightarrow CD \\ C &\longrightarrow \doteq C \mid \epsilon \\ D &\longrightarrow \approx D \mid \blacktriangleleft \mid \epsilon \end{aligned}$$

Después, escribe las funciones de un analizador descendente recursivo asociadas a los símbolos  $S$  y  $C$ . Indica claramente qué hace cualquier función auxiliar que utilices.

\* **10** Da un ejemplo de gramática no ambigua que sea LL(3), pero ni LL(1) ni LL(2).

\* **11** Diseña una gramática no ambigua que permita reconocer expresiones con los operadores binarios y unarios que aparecen en el cuadro siguiente; los operadores aparecen ordenados de mayor a menor precedencia.

OPERADORES	ARIDAD	ASOCIATIVIDAD
#	unario	—
$\leftrightarrow$ $\leftarrow$	binario	por la derecha
$\hat{\ } \oplus$	unario	—
*	binario	por la izquierda

Los operandos pueden ser números enteros o identificadores y se permiten los paréntesis. Los operadores unarios # y  $\hat{\ }$  preceden al operando al que se aplican; se permiten secuencias de más de uno de estos operadores. Por otro lado, solo se permite un operador  $\oplus$  a continuación del operando correspondiente. Un ejemplo de expresión válida para la gramática sería la siguiente:

$$\hat{\ }^5 \star a \leftarrow b \leftarrow (10 \star 15 \oplus)$$

Después, escribe en pseudocódigo las funciones de un analizador descendente recursivo asociadas a los no terminales de tu gramática encargados de los operadores *binarios*; calcula los conjuntos de predicción necesarios para que el pseudocódigo sea lo más completo posible.

❄ **12** Demuestra que la siguiente gramática es LL(1) y construye su tabla de análisis sintáctico LL(1).

$$\begin{aligned} A &\longrightarrow BC \\ B &\longrightarrow \mathbf{b} \mathbf{c} B \mid \epsilon \\ C &\longrightarrow \mathbf{c} C \mid \epsilon \end{aligned}$$

Después haz la traza del análisis de un analizador descendente basado en la tabla obtenida anteriormente de las dos siguientes cadenas de entrada: "c" y  $\epsilon$ . Indica claramente la configuración de la pila, la entrada restante y la acción realizada en cada iteración del algoritmo de análisis sintáctico.

❄ **13** Demuestra que la siguiente gramática es LL(1) y construye su tabla de análisis sintáctico LL(1).

$$\begin{aligned} S &\longrightarrow \mathbf{a} \\ S &\longrightarrow \mathbf{b} S \mathbf{c} S \mathbf{b} \end{aligned}$$

❄ **14** Dada la siguiente gramática con  $S$  como símbolo inicial:

$$\begin{aligned} S &\longrightarrow \mathbf{a} A \mathbf{a} \mid \epsilon \\ A &\longrightarrow \mathbf{a} \mathbf{b} S \mid \mathbf{c} \end{aligned}$$

determina si es LL(1) y justifica tu respuesta.

❄ **15** Calcula los conjuntos de predicción de las reglas de la siguiente gramática e indica por qué la gramática es LL(1).

$$\begin{aligned} S &\longrightarrow X \mid \ominus \pm S \\ X &\longrightarrow T Y \\ Y &\longrightarrow \star \mid \boxtimes Y \mid \epsilon \\ T &\longrightarrow \cap T \mid \epsilon \end{aligned}$$

Después, escribe las funciones de un analizador descendente recursivo asociadas a los símbolos  $X$  e  $Y$ . Indica claramente qué hace cualquier función auxiliar que utilices.

❄ **16** Demuestra que la siguiente gramática es ambigua. Después indica si es LL(1). Finalmente, diseña una gramática no ambigua equivalente.

$$\begin{aligned} S &\longrightarrow B D \mid \epsilon \\ B &\longrightarrow \mathbf{b} \mid \mathbf{d} \mid \epsilon \\ D &\longrightarrow D \mathbf{d} \mid \epsilon \end{aligned}$$

❄ **17** Demuestra que la siguiente gramática no es LL(1). Después, estudia sus reglas para descubrir alguna de las posibles causas.

$$\begin{aligned} S &\longrightarrow B C D \\ B &\longrightarrow C D \mid D C \\ C &\longrightarrow \mathbf{c} C \mid \epsilon \\ D &\longrightarrow D \mathbf{d} \mid \mathbf{a} \mid \epsilon \end{aligned}$$

❄ **18** Da un ejemplo de gramática que sea LL(2), pero no LL(1).

❄ **19** Calcula los conjuntos de predicción de las reglas de la siguiente gramática. A la vista de lo anterior, indica si la gramática es LL(1).

$$\begin{aligned}
 S &\longrightarrow E D C \\
 E &\longrightarrow \epsilon \mid A e \\
 A &\longrightarrow \mathbf{a} \mid \epsilon \\
 D &\longrightarrow B \mid \mathbf{bb} \\
 C &\longrightarrow \epsilon \mid \mathbf{c} \\
 B &\longrightarrow \epsilon
 \end{aligned}$$

✳ **20** Da un ejemplo de gramática no ambigua que no sea  $LL(k)$  para ningún  $k \leq 2$  y que sea  $LL(k)$  para todo  $k \geq 4$ .

✳ **21** Demuestra que la siguiente gramática es  $LL(1)$  y construye su tabla de análisis sintáctico  $LL(1)$ .

$$\begin{aligned}
 S &\longrightarrow B \\
 B &\longrightarrow C D \\
 C &\longrightarrow \mathbf{c} C \mid \epsilon \\
 D &\longrightarrow \mathbf{a}
 \end{aligned}$$

Después haz la traza del análisis de la cadena de entrada  $\mathbf{ca}$  con un analizador descendente basado en la tabla obtenida anteriormente. Indica claramente la configuración de la pila, la entrada restante y la acción realizada en cada iteración del algoritmo de análisis sintáctico.

✳ **22** Dada la siguiente gramática:

$$\begin{aligned}
 1) \quad S &\longrightarrow A \mathbf{b} C \\
 2) \quad S &\longrightarrow \epsilon \\
 3) \quad A &\longrightarrow \epsilon \\
 4) \quad C &\longrightarrow \epsilon
 \end{aligned}$$

Diseña el autómata aceptor de prefijos viables y haz la traza del análisis ascendente de las cadenas “ $\mathbf{b}$ ” y “ $\epsilon$ ”. Utiliza la siguiente plantilla para hacer la traza:

PILA	ENTRADA	ACCIÓN
⋮	⋮	⋮

✳ **23** Dada la siguiente gramática:

- 1)  $S \rightarrow A \mathbf{b} C$
- 2)  $S \rightarrow \epsilon$
- 3)  $A \rightarrow \epsilon$
- 4)  $C \rightarrow \epsilon$

Diseña el autómata aceptor de prefijos viables y haz la traza del análisis ascendente de las cadenas “**b**” y “ $\epsilon$ ”. Utiliza la siguiente plantilla para hacer la traza:

PILA	ENTRADA	ACCIÓN
⋮	⋮	⋮

✳ **24** Da un ejemplo de gramática SLR que no sea LL(2). Justifica tu respuesta.

✳ **25** Explica por qué el método de análisis sintáctico descendente predictivo no puede trabajar con gramáticas recursivas por la izquierda.

## Análisis sintáctico ascendente

✳ **26** Argumenta la verdad o falsedad de la siguiente afirmación: si una gramática presenta recursividad por la derecha, entonces no es LALR(1).

✳ **27** Argumenta la verdad o falsedad de la siguiente afirmación: existen gramáticas no ambiguas para las que cualquier método LR produce conflictos.

✳ **28** ¿Es posible que el autómata aceptor de prefijos viables resultante de aplicar el algoritmo de construcción de tales autómatas visto en clase sea indeterminista? Razona tu respuesta.

✳ **29** Construye la colección canónica de elementos LR(0) (o el autómata aceptor de prefijos) de la siguiente gramática.

- $$\begin{aligned}
 E &\rightarrow \mathbf{id} \\
 E &\rightarrow \mathbf{id} (E) \\
 E &\rightarrow E + \mathbf{id}
 \end{aligned}$$

Utiliza después el método SLR para construir las tablas de análisis sintáctico de un analizador LR. A la vista de lo anterior, ¿es SLR(1) la gramática?

✳ **30** ¿Puede el estado inicial del autómata aceptor de prefijos aparecer en alguna celda de la tabla VE\_A de un analizador SLR? Justifica tu respuesta.

✳ **31** Argumenta la verdad o falsedad de la siguiente afirmación: un buen sistema de recuperación de errores sintácticos no necesita tener en cuenta las características específicas del tipo de analizador sintáctico con el que se vaya a utilizar.

✳ **32** Da un ejemplo del conjunto de *items* asociados a un estado del autómata aceptor de prefijos de un analizador SLR en el que se produzca un conflicto reducción/reducción.

✳ **33** Diseña una gramática no ambigua *de un solo elemento no terminal* (variable) para el lenguaje de las cadenas balanceadas de paréntesis y corchetes. La cadena vacía no ha de pertenecer al lenguaje. Ejemplos de entradas válidas para la gramática son:

((([()])) [ ] [[()]]

Después, obtén las tablas SLR de un analizador sintáctico ascendente para tu gramática e indica si esta es o no SLR.

✳ **34** Dada la siguiente tabla de análisis sintáctico SLR asociada a la gramática que se muestra posteriormente, haz la traza del análisis de la cadena de entrada  $(( ), (a, a))$ .

Indica claramente la configuración de la pila, la entrada restante y la acción realizada en cada iteración del algoritmo de análisis sintáctico.

Estado	Acción					Ir a	
	a	(	)	,	\$	S	L
0	d3	d4	r5		r5	1	2
1					acepta		
2				d5	r2		
3				d6			
4	d3	d4	r5		r5		7
5	d3	d4	r5		r5	8	
6	d3	d4	r5		r5		9
7			d10				
8					r1		
9			r4		r4		
10			r3		r3		

La gramática para la que se ha obtenido la tabla de análisis sintáctico anterior es la que sigue:

- (1)  $S \rightarrow L, S$
- (2)  $S \rightarrow L$
- (3)  $L \rightarrow ( L )$
- (4)  $L \rightarrow a, L$
- (5)  $L \rightarrow \epsilon$

\* **35** ¿Qué le ocurre al analizador SLR resultante si al diseñar el autómata aceptor de prefijos viables duplicamos un estado ya existente? ¿Y si olvidamos un estado?

\* **36** ¿Es posible encontrar una fila de la tabla ACCIÓN de un analizador SLR en la que todas las celdas indiquen error? Razona tu respuesta.

\* **37** Dada la siguiente gramática:

- 1)  $E \rightarrow \mathbf{id}$
- 2)  $E \rightarrow \mathbf{id} ( E )$
- 3)  $E \rightarrow E + \mathbf{id}$

Haz la traza del análisis ascendente de la cadena “**id ( id + id )**”. *Pista:* será más sencillo si usas la pila para almacenar directamente símbolos de la gramática, en lugar de números de estado (es posible resolver este problema sin diseñar el autómata aceptor de prefijos viables).

Utiliza la siguiente plantilla para hacer la traza:

PILA	ENTRADA	ACCIÓN
⋮	⋮	⋮

✳ **38** Argumenta la verdad o falsedad de la siguiente afirmación: si una gramática presenta factores comunes por la izquierda, entonces no es LALR(1).

✳ **39** Argumenta la verdad o falsedad de la siguiente afirmación: existen gramáticas que no son ni ambiguas ni recursivas por la izquierda para las que cualquier método LL( $k$ ) con  $k \leq 10$  produce conflictos.

✳ **40** Explica por qué la siguiente gramática es LL(1) pero no SLR.

$$\begin{aligned}
 S &\longrightarrow T \mathbf{a} T \mathbf{b} \\
 S &\longrightarrow V \mathbf{b} V \mathbf{a} \\
 T &\longrightarrow \epsilon \\
 V &\longrightarrow \epsilon
 \end{aligned}$$

*Nota:* debes justificar por qué es así; es decir, no basta con demostrar que la gramática es LL(1) y no es SLR; de hecho, no es necesario siquiera que calcules las tablas de análisis sintáctico.

## Traducción dirigida por la sintaxis

✳ **41** Dada la siguiente gramática que permite declarar distintos tipos de variables, construye un ETDS que realice una traducción como la indicada en los ejemplos de más abajo.

$$\begin{aligned}
 S &\longrightarrow T L ; \\
 L &\longrightarrow V, L \mid V \\
 V &\longrightarrow V [ \mathbf{entero} ] \mid E \\
 E &\longrightarrow * E \mid \mathbf{id} \\
 T &\longrightarrow \mathbf{int} \mid \mathbf{float} \mid \mathbf{char}
 \end{aligned}$$

Por ejemplo, ante una entrada como:

```
int a, *p;
```

el ETDS debe generar:

La variable `a` es un entero.

La variable `p` es un puntero a un entero.

Por otro lado, ante la siguiente entrada:

```
char tabla[1];
```

el ETDS generará:

La variable `tabla` es un array 1-dimensional con 1 celda en total; cada celda es un carácter.

Y, finalmente, ante una entrada como:

```
float **n[2][3][7];
```

la salida del ETDS debe ser:

La variable `n` es un array 3-dimensional con 42 celdas en total; cada celda es un puntero a un puntero a un real.

Toda la información debe pasarse a través de atributos y no es posible utilizar ninguna variable global. Debes indicar claramente, además, de qué tipo es cada uno de los atributos que utilizas y cuál es su cometido.

✱ **42** Dada la siguiente gramática, construye un esquema de traducción dirigida por la sintaxis (ETDS) que cuente e imprima el número de símbolos **a** que aparecen en la entrada justo *a continuación* de un símbolo **e** e inmediatamente *antes* de un símbolo **d**.

$$\begin{array}{l}
X \longrightarrow S \\
S \longrightarrow AB \\
S \longrightarrow AC \\
A \longrightarrow a e A \\
A \longrightarrow a d a A \\
A \longrightarrow \epsilon \\
A \longrightarrow e a \\
B \longrightarrow d e a d a \\
B \longrightarrow d e a e \\
C \longrightarrow D d \\
D \longrightarrow a \\
D \longrightarrow \epsilon
\end{array}$$

Por ejemplo, ante una entrada como `a e a e a d a e a d`, el ETDS debe imprimir el valor 2; la impresión se debe hacer en la regla del no terminal  $X$ .

El ETDS no puede utilizar ninguna variable global: toda la información debe pasarse a través de atributos. Indica claramente de qué tipo es cada uno de los atributos que utilices y cuál es su cometido.

✱ **43** Diseña la gramática que caracteriza la sintaxis del lenguaje fuente mostrado en el siguiente ejemplo, que permite declarar clases (posiblemente anidadas) y sus métodos. Después, escribe un esquema de traducción dirigida por la sintaxis (ETDS) que genere la traducción adecuada, según se indica a continuación.

Por ejemplo, ante una entrada (que ha de ser reconocida por tu gramática) como:

```

class A {
  public:
    class B {
      private:
        class C {
          public:
            real g (int j);
            real h (real s);
        };
        int m ();
    };
  private:
    int f (int i, real r);
    class D {};
};

```

el ETDS ha de producir la siguiente traducción:

```

clase A::B::pública
clase B::C::privada
método C::g::público devuelve real, recibe 1 entero y 0 reales
método C::h::público devuelve real, recibe 0 enteros y 1 real
método B::m::privado devuelve entero, recibe 0 enteros y 0 reales
método A::f::privado devuelve entero, recibe 1 entero y 1 real
clase A::D::privada

```

Date cuenta de que cada nombre de clase o método lleva como prefijo el nombre de la clase a la que pertenece.

Toda la información debe pasarse a través de atributos y no es posible utilizar ninguna variable global. Debes indicar claramente, además, de qué tipo es cada uno de los atributos que utilizas y cuál es su cometido.

✳ **44** Dada la siguiente gramática que permite definir listas anidadas, construye un esquema de traducción dirigida por la sintaxis (ETDS) que aumente cada una de las listas que aparezcan en la entrada con dos elementos: uno al comienzo de la lista que indique su nivel de anidamiento (comenzando desde 0 para la lista exterior) y otro al final de la lista que indique su longitud (antes de la inserción de los nuevos elementos).

$$\begin{aligned}
 S &\longrightarrow L \\
 L &\longrightarrow [ T ] \\
 L &\longrightarrow [ ] \\
 T &\longrightarrow F \text{ coma } T \\
 T &\longrightarrow F \\
 F &\longrightarrow \mathbf{a} \\
 F &\longrightarrow \mathbf{b} \\
 F &\longrightarrow L
 \end{aligned}$$

Por ejemplo, ante una entrada como

$$[ \mathbf{a}, [ \mathbf{b}, \mathbf{b}, \mathbf{a}, [ \mathbf{b} ] ], [ ] ]$$

el ETDS debe generar

$$[ \mathbf{0}, \mathbf{a}, [ \mathbf{1}, \mathbf{b}, \mathbf{b}, \mathbf{a}, [ \mathbf{2}, \mathbf{b}, \mathbf{1} ], \mathbf{4} ], [ \mathbf{1}, \mathbf{0} ], \mathbf{3} ]$$

El ETDS no puede utilizar ninguna variable global: toda la información debe pasarse a través de atributos. Indica claramente de qué tipo es cada uno de los atributos que utilizas y cuál es su cometido.

✳ **45** Diseña una gramática y un ETDS que permita traducir expre-

siones con enteros de notación infija a postfija o prefija según una palabra reservada que las precede. La entrada será un conjunto de expresiones a traducir separadas por comas. Por ejemplo, la traducción de

prefija  $1 * 2 \oplus 3$ , postfija  $(1 * 5) \oplus 4$

será

$* 1 \oplus 2 3, 1 5 * 4 \oplus$

Los operadores de las expresiones son los operadores binarios que aparecen en el cuadro siguiente con la asociatividad que allí se indica; los operadores aparecen ordenados de mayor a menor precedencia.

OPERADOR	ASOCIATIVIDAD
$\oplus$	por la derecha
$*$	por la izquierda

Como se ve en el ejemplo anterior, es posible también el uso de paréntesis en las expresiones. Toda la información debe pasarse a través de atributos: no es posible utilizar ninguna variable global. Indica claramente de qué tipo es cada uno de los atributos que utilizas y cuál es su cometido.

✱ **46** Diseña una gramática que permita escribir listas de expresiones aritméticas separadas por punto y coma, como se muestra a continuación. Después, escribe un esquema de traducción dirigida por la sintaxis (ETDS) que simplifique determinadas subexpresiones, según se indica más abajo.

Los operadores de las expresiones son los que aparecen en el cuadro siguiente, ordenados de mayor a menor precedencia, con la asociatividad y tipo que allí se indica; el operador unario debe aparecer detrás del operando correspondiente. Los operandos serán constantes de tipo entero.

OPERADOR	TIPO	ASOCIATIVIDAD
$\otimes$	binario	por la derecha
!	unario	—

El ETDS debe simplificar los dos siguientes casos en los que aparecen paréntesis redundantes:

- aquellas subexpresiones rodeadas por **dos** o más parejas de paréntesis
- aquellas subexpresiones formadas únicamente por un entero rodeado por **una** o más parejas de paréntesis

Por ejemplo, ante una entrada como:

$$3 \otimes (((5 \otimes 3)!! \otimes 5)) ; (((11))) \otimes (3) ; 32$$

la traducción resultante ha de ser:

$$3 \otimes ((5 \otimes 3)!! \otimes 5) ; 11 \otimes 3 ; 32$$

Date cuenta de que solo se han de eliminar los paréntesis redundantes que aparezcan en uno de los dos casos anteriores; cualquier otro uso de los paréntesis (como aquel en el que la asociatividad de un operador garantiza el mismo orden de evaluación que el impuesto por los paréntesis) debe respetarse, aunque pueda parecer merecedor de simplificación.

Toda la información debe pasarse a través de atributos y no es posible utilizar ninguna variable global. Debes indicar claramente, además, de qué tipo es cada uno de los atributos que utilizas y cuál es su cometido.

✱ **47** En una *definición dirigida por la sintaxis con atributos por la izquierda* (o, en general, en una *gramática de atributos por la izquierda*) la información de los atributos siempre fluye de abajo a arriba, de arriba a abajo o de izquierda a derecha, pero *nunca* de derecha a izquierda. ¿Qué consecuencia importante tiene esta propiedad?

✱ **48** La siguiente gramática define el lenguaje de los números romanos menores de 100:

$S$	$\longrightarrow$	$Decenas\ Unidades$
$Decenas$	$\longrightarrow$	$Dec$
$Decenas$	$\longrightarrow$	$\mathbf{XL}$
$Decenas$	$\longrightarrow$	$\mathbf{L}\ Dec$
$Decenas$	$\longrightarrow$	$\mathbf{XC}$
$Dec$	$\longrightarrow$	$Dec\ \mathbf{X}$
$Dec$	$\longrightarrow$	$\epsilon$
$Unidades$	$\longrightarrow$	$Unid$
$Unidades$	$\longrightarrow$	$\mathbf{IV}$
$Unidades$	$\longrightarrow$	$\mathbf{V}\ Unid$
$Unidades$	$\longrightarrow$	$\mathbf{IX}$
$Unid$	$\longrightarrow$	$Unid\ \mathbf{I}$
$Unid$	$\longrightarrow$	$\epsilon$

Escribe un esquema de traducción dirigida por la sintaxis (ETDS), basado en la gramática anterior sin modificar, que permita obtener en un atributo sintetizado de  $S$  el valor decimal del número romano definido. Además, el ETDS ha de restringir el número de  $\mathbf{X}$  en  $Dec$  y de  $\mathbf{I}$  en  $Unid$  a 3, es de-

cir, que si en la entrada aparecen más de 3 veces consecutivas uno de estos símbolos se ha de emitir el error semántico oportuno.

Por ejemplo, para una cadena de entrada como **XXI** la traducción obtenida a de ser 21 y para la cadena de entrada **VIII** se ha de obtener un error semántico.

Toda la información debe pasarse a través de atributos y no es posible utilizar ninguna variable global. Debes indicar claramente, además, de qué tipo es cada uno de los atributos que utilizas y cuál es su cometido.

*Nota:* recuerda algunos números romanos: I (1), II (2), III (3), IV (4), V (5), VI (6), IX (9), X(10), XXXV (35), XL (40), LI (51), LVII (57), XCIX (99),...

\* **49** La siguiente gramática genera expresiones infijas de sumas y restas de constantes enteras y variables. Diseña un ETDS que las traduzca a notación prefija parentizada preoperando, cuando sea posible y la asociatividad del operador lo permita, las constantes enteras que aparezcan secuencialmente en la entrada.

Por ejemplo, ante una entrada como  $3-a+4+(7-1)$ , la traducción ha de ser `suma(suma(resta(3,a),4),6)`; además, la traducción de  $5-1+a$  ha de ser `suma(4,a)` y la traducción de  $a+5-1$  ha de ser `resta(suma(a,5),1)`.

$$\begin{aligned} E &\longrightarrow E \text{ opsum } T \\ E &\longrightarrow T \\ T &\longrightarrow \text{id} \\ T &\longrightarrow \text{entero} \\ T &\longrightarrow ( E ) \end{aligned}$$

Toda la información debe pasarse a través de atributos y no es posible utilizar ninguna variable global. Debes indicar claramente, además, de qué tipo es cada uno de los atributos que utilizas y cuál es su cometido.

\* **50** Elimina la recursividad por la izquierda y los factores comunes por la izquierda de la gramática del problema anterior. Después rediseña el ETDS del problema anterior sobre la nueva gramática.

\* **51** La siguiente gramática genera números binarios. Diseña un ETDS que obtenga en un atributo sintetizado del símbolo inicial el número binario correspondiente a desplazar una posición a la izquierda los dígitos de la entrada; el dígito más significativo (el de más a la izquierda) ha de pasar a la posición menos significativa. Por ejemplo, ante una entrada como

110, la traducción ha de ser 101.

$$\begin{aligned} S &\longrightarrow D \\ D &\longrightarrow DB \mid B \\ B &\longrightarrow \mathbf{cero} \mid \mathbf{uno} \end{aligned}$$

Toda la información debe pasarse a través de atributos y no es posible utilizar ninguna variable global. Debes indicar claramente, además, de qué tipo es cada uno de los atributos que utilizas y cuál es su cometido.

❄ **52** Repite el ejercicio anterior, pero considerando esta vez la siguiente gramática recursiva por la derecha:

$$\begin{aligned} S &\longrightarrow D \\ D &\longrightarrow BD \mid B \\ B &\longrightarrow \mathbf{cero} \mid \mathbf{uno} \end{aligned}$$

❄ **53** Diseña una gramática y un esquema de traducción dirigida por la sintaxis (ETDS) que permita traducir expresiones booleanas cuyos operandos son identificadores y que, además, realice la simplificación de expresiones en las que aparecen varios operadores de negación seguidos. Así, por ejemplo, la traducción de  $(a \mid\mid b) \&\& !c$  será  $\text{and}(\text{or}(a,b), \text{not}(c))$  y la traducción de  $!!!!(a \&\& !!b \&\& c)$  será  $\text{not}(\text{and}(a, \text{and}(b,c)))$ .

Los operadores de las expresiones son los operadores binarios que aparecen en el cuadro siguiente con la asociatividad y ariedad que allí se indica; los operadores aparecen ordenados de mayor a menor precedencia.

OPERADOR	ARIEDAD	ASOCIATIVIDAD
!	unario	–
&&	binario	por la derecha
	binario	por la izquierda

Como se ve en los ejemplos, es posible también el uso de paréntesis en las expresiones de entrada. Nótese que solo se debe simplificar el uso consecutivo de operadores de negación.

Toda la información debe pasarse a través de atributos y no es posible utilizar ninguna variable global. Debes indicar claramente, además, de qué tipo es cada uno de los atributos que utilizas y cuál es su cometido.

❄ **54** Dada la siguiente gramática que permite declarar clases anidadas

y sus métodos para un determinado lenguaje orientado a objetos, construye un ETDS que traduzca a una notación como la indicada en el ejemplo de más abajo.

$$\begin{aligned}
 S &\longrightarrow C \\
 C &\longrightarrow \mathbf{class\ id\ \{ B\ V\}} \\
 B &\longrightarrow \mathbf{public : P\ | \ \epsilon} \\
 V &\longrightarrow \mathbf{private : P\ | \ \epsilon} \\
 P &\longrightarrow D\ P\ | \ \epsilon \\
 D &\longrightarrow T\ \mathbf{id}\ ( T\ \mathbf{id}\ L )\ | \ C \\
 L &\longrightarrow ,\ T\ \mathbf{id}\ L\ | \ \epsilon \\
 T &\longrightarrow \mathbf{int\ | \ float}
 \end{aligned}$$

Por ejemplo, ante una entrada como:

```

class A {
  public:
    int f1(int n,float s)
  private:
    class B {
      private:
        float f2 (float r,float s,float t)
        class C {}
    }
}

```

el ETDS debe generar:

```

class A {
  público:
    A::f1 (entero x real -> entero)
  privado:
    class A::B {
      privado:
        A::B::f2 (real x real x real -> real)
        class A::B::C {}
    }
}

```

Toda la información debe pasarse a través de atributos y no es posible utilizar ninguna variable global. Debes indicar claramente, además, de qué tipo es cada uno de los atributos que utilizas y cuál es su cometido.

✳ **55** La siguiente gramática define un lenguaje de listas (eventualmente anidadas) de números enteros:

$$\begin{aligned}
 S &\longrightarrow L \\
 L &\longrightarrow [ T ] \\
 L &\longrightarrow [ ] \\
 T &\longrightarrow F, T \\
 T &\longrightarrow F \\
 F &\longrightarrow \text{entero} \\
 F &\longrightarrow L
 \end{aligned}$$

Escribe un esquema de traducción dirigida por la sintaxis (ETDS), basado en la gramática anterior sin modificar, que intercambie el primer y el último elemento de cualquiera de las listas de la entrada. Por ejemplo, para una cadena de entrada como:

$$[ [ 1, 2, 3, 4, [ 5 ] ], [ ], [ 6, 7 ], [ 8, 9, 10 ] ]$$

la traducción resultante ha de ser:

$$[ [ 10, 9, 8 ], [ ], [ 7, 6 ], [ [ 5 ], 2, 3, 4, 1 ] ]$$

Toda la información debe pasarse a través de atributos y no es posible utilizar ninguna variable global. Debes indicar claramente, además, de qué tipo es cada uno de los atributos que utilizas y cuál es su cometido.

✳ **56** Traduce a notación de `yacc` el siguiente ETDS; intenta utilizar el menor número de marcadores posible.

$$\begin{aligned}
 S &\longrightarrow A \{ L.h := A.s \} L \text{ pyc } \{ R.k := A.s \} R \\
 &\quad \{ S.h := L.s \parallel R.s \} \\
 A &\longrightarrow \text{def } \{ A.s := \text{def.lexema} \} \\
 A &\longrightarrow \epsilon \{ A.s := \text{" "} \} \\
 L &\longrightarrow \{ L_1.h := L.h \} L_1 \text{ coma id} \\
 &\quad \{ L.s := L_1.s \parallel \text{" ,"} \parallel L.h \parallel \text{id.lexema} \} \\
 L &\longrightarrow \text{id} \{ L.s := L.h \parallel \text{id.lexema} \} \\
 R &\longrightarrow \{ R_1.k := R.k \} R_1 \text{ pyc id} \\
 &\quad \{ R.s := R_1.s \parallel \text{" ;"} \parallel R.k \parallel \text{id.lexema} \} \\
 R &\longrightarrow \text{var } \{ L.h := R.k \} L \{ R.s := L.s \}
 \end{aligned}$$

✳ **57** Traduce a notación de `yacc` el siguiente ETDS:

$$\begin{aligned}
D &\longrightarrow T \{ L.h := T.tipo \} L \text{ pyc} \\
L_0 &\longrightarrow \{ L_1.h := L_0.h \} L_1 \text{ coma } \{ V.h := L.h \} V \\
L &\longrightarrow \{ V.h := L.h \} V \\
V &\longrightarrow \text{id} \{ \text{insertaTS}(\text{id} .lexema, V.h) \} \\
T &\longrightarrow \text{int} \{ T.tipo := \text{entero} \} \\
T &\longrightarrow \text{float} \{ T.tipo := \text{real} \}
\end{aligned}$$

✱ **58** La siguiente gramática define la sintaxis del operador **prop** que se aplica a listas (eventualmente anidadas) de números enteros:

$$\begin{aligned}
S &\longrightarrow \text{prop} ( \text{entero} , L ) \\
L &\longrightarrow [ T ] \\
L &\longrightarrow [ ] \\
T &\longrightarrow F , T \\
T &\longrightarrow F \\
F &\longrightarrow \text{entero} \\
F &\longrightarrow L
\end{aligned}$$

El operador **prop** busca de izquierda a derecha en la lista indicada como segundo argumento un elemento que sea igual al entero del primer argumento. Si encuentra este elemento, el resultado del operador es el resto de la lista, incluyendo el elemento encontrado. En otro caso, el resultado es una lista vacía.

Escribe un esquema de traducción dirigida por la sintaxis (ETDS), basado en la gramática anterior sin modificar, que produzca como traducción la correspondiente a la aplicación del operador. La traducción resultante ha de ser una lista bien formada. Por ejemplo, para una cadena de entrada como:

$$\text{prop} ( [[1], [5,6, [3,4,5]], [], 2] , 4 )$$

la traducción obtenida ha de ser:

$$[[[4,5]], [], 2]$$

Toda la información debe pasarse a través de atributos y no es posible utilizar ninguna variable global. Debes indicar claramente, además, de qué tipo es cada uno de los atributos que utilizas y cuál es su cometido.

✱ **59** Dada la siguiente gramática, construye un esquema de traducción dirigida por la sintaxis (ETDS) que cuente e imprima el número de

símbolos **b** que aparecen en la entrada justo *a continuación* de un símbolo **a**.

$$\begin{aligned} S &\longrightarrow \mathbf{a b A} \\ A &\longrightarrow \mathbf{a b c A} \\ A &\longrightarrow \mathbf{c b c A} \\ A &\longrightarrow \epsilon \end{aligned}$$

Por ejemplo, ante una entrada como **ababcbbc**, el ETDS debe imprimir el valor 2; la impresión se debe hacer en la regla del no terminal *S*.

El ETDS no puede utilizar ninguna variable global: toda la información debe pasarse a través de atributos. Indica claramente de qué tipo es cada uno de los atributos que utilices y cuál es su cometido.

✱ **60** La siguiente gramática genera expresiones infijas de sumas y multiplicaciones de constantes enteras y variables.

$$\begin{aligned} X &\longrightarrow E \\ E &\longrightarrow E + T \\ E &\longrightarrow T \\ T &\longrightarrow T * F \\ T &\longrightarrow F \\ F &\longrightarrow \mathbf{id} \\ F &\longrightarrow \mathbf{entero} \\ F &\longrightarrow ( E ) \end{aligned}$$

Diseña un ETDS que, dada una expresión de entrada, imprima el número de sumas y multiplicaciones contenidas en ella.

Por ejemplo, ante una entrada como **3+a\*4\*(7+1+b)**, la traducción ha de ser:

número de sumas: 3; número de multiplicaciones: 2

Toda la información debe pasarse a través de atributos y no es posible utilizar ninguna variable global. Debes indicar claramente, además, de qué tipo es cada uno de los atributos que utilizas y cuál es su cometido.

✱ **61** La siguiente gramática genera expresiones infijas de sumas y multiplicaciones de constantes enteras y variables.

$$\begin{array}{l}
 X \longrightarrow E \\
 E \longrightarrow E + T \\
 E \longrightarrow T \\
 T \longrightarrow T * F \\
 T \longrightarrow F \\
 F \longrightarrow \text{id} \\
 F \longrightarrow \text{entero}
 \end{array}$$

Diseña un ETDS que, dada una expresión de entrada, imprima el número máximo de sumas y multiplicaciones *seguidas* contenidas en ella.

Por ejemplo, ante una entrada como  $3+a*4*7+1+b+h+5*v+m$ , la traducción ha de ser:

número máximo de sumas seguidas: 4; número máximo de multiplicaciones seguidas: 2.

Toda la información debe pasarse a través de atributos y no es posible utilizar ninguna variable global. Debes indicar claramente, además, de qué tipo es cada uno de los atributos que utilizas y cuál es su cometido.

✱ **62** Los operadores del lenguaje de programación orientado a objetos Smalltalk no siguen las reglas habituales de asociatividad y precedencia; muy al contrario, en Smalltalk todas las expresiones se evalúan de izquierda a derecha y este comportamiento se puede alterar mediante el uso adecuado de los paréntesis. Así,  $2+3*3$  se evalúa como 15 y  $2+(3*3)$  como 11.

Diseña una gramática no ambigua para los operadores de suma y multiplicación (*tokens opsum* y *opmul*) que respete este comportamiento. Después, construye los árboles de análisis sintáctico de las dos expresiones anteriores y comprueba sobre ellos que la traducción a notación prefija produce el resultado correcto. *Nota:* no es necesario diseñar ningún ETDS para lo anterior: es suficiente con que “decores” el árbol con la traducción.

## Generación de código

✱ **63** Considera el siguiente fragmento de la especificación sintáctica de un lenguaje:

$$\begin{aligned}
 I &\longrightarrow \mathbf{print} T \\
 I &\longrightarrow \mathbf{read} R \\
 T &\longrightarrow T + F \\
 T &\longrightarrow F \\
 F &\longrightarrow \mathbf{cteint} \\
 F &\longrightarrow \mathbf{id} ( P ) \\
 F &\longrightarrow R \\
 F &\longrightarrow ( T ) \\
 P &\longrightarrow \epsilon \\
 P &\longrightarrow L \\
 L &\longrightarrow L \mathbf{coma} T \\
 L &\longrightarrow T \\
 R &\longrightarrow \mathbf{id} D \\
 D &\longrightarrow [ T ] D \\
 D &\longrightarrow \epsilon
 \end{aligned}$$

Diseña un esquema de traducción dirigida por la sintaxis (ETDS) que genere el código adecuado para la máquina virtual `m2r` y que emita los mensajes de error semántico oportunos. Para ello, has de tener en cuenta los siguientes aspectos:

- En este lenguaje el único tipo de datos simple son los enteros, que ocupan una única posición de memoria.
- Como puedes ver en la gramática, el lenguaje permite usar *arrays* y efectuar llamadas a funciones.
- Como el único tipo de datos del lenguaje son los enteros, es suficiente almacenar en la tabla de símbolos el número de argumentos que tiene cada función y no es necesario introducir ninguna información adicional sobre las funciones en la tabla de tipos.

Los arrays se almacenan en la tabla de tipos indicando el tamaño de cada una de sus dimensiones (el límite inferior es siempre 0).

La inserción de toda esta información en cada una de las tablas se lleva a cabo en otras reglas de la gramática, que no has de considerar.

Así, a un *array* y una función como la del siguiente ejemplo:

```
int s[2][20][200];
int function f(int a, int b, int c);
```

les corresponderían las siguientes tablas:

TABLA DE SÍMBOLOS

NOMBRE	TIPO	DIRECCIÓN	NÚM. PARÁMETROS	ETIQ.
s	3	0	-	-
f	FUNC	-	3	L1

TABLA DE TIPOS

NÚMERO	TIPO	TAMAÑO	TIPO BASE
0	ENTERO		
1	ARRAY	200	0
2	ARRAY	20	1
3	ARRAY	2	2

- d. No se ha de generar código que compruebe en tiempo de ejecución que los índices están dentro del rango válido. Sin embargo, se ha de tener en cuenta el siguiente aspecto importante: si el valor de la expresión (en el no terminal  $T$ ) que aparece entre corchetes es negativo, se ha de interpretar como un acceso desde el final del *array*, es decir, [-1] se refiere a la última posición, [-2] a la penúltima, etc. La comprobación del signo de la expresión se ha de hacer, evidentemente, en tiempo de ejecución.

Señala claramente qué hace cualquier función auxiliar que utilices. Toda la información debe pasarse a través de atributos y no es posible usar ninguna variable global (excepto las correspondientes a la tabla de símbolos y de tipos, y la usada para determinar la posición de las temporales). Indica, además, de qué tipo es cada uno de los atributos que utilizas y cuál es su cometido.

*Nota:* recuerda utilizar la notación de los ETDS y *no* la de yacc.

✱ **64** Considera el siguiente fragmento de la especificación sintáctica de un lenguaje:

$$\begin{aligned}
 I &\longrightarrow \mathbf{for} ( E ; E ; E ) \text{ Bloque} \\
 E &\longrightarrow R = E \\
 E &\longrightarrow T \\
 T &\longrightarrow T + F \\
 T &\longrightarrow F \\
 F &\longrightarrow \mathbf{cteint} \\
 F &\longrightarrow \mathbf{ctebool} \\
 F &\longrightarrow R \\
 F &\longrightarrow ( E ) \\
 R &\longrightarrow \mathbf{id} D \\
 D &\longrightarrow D [ E ] \\
 D &\longrightarrow \epsilon
 \end{aligned}$$

Diseña un esquema de traducción dirigida por la sintaxis (ETDS) que genere el código adecuado para la máquina virtual `m2r` y que emita los mensajes de error semántico oportunos. Para ello, has de tener en cuenta los siguientes aspectos:

- a. En este lenguaje los tipos enteros y los booleanos son totalmente incompatibles; además, solo es posible sumar valores enteros.
- b. Tanto los datos de tipo booleano como los de tipo entero ocupan una única posición de memoria.
- c. Como se observa en la gramática, la asignación es una expresión más.
- d. Puedes asumir que el no terminal *Bloque* tiene un atributo sintetizado *Bloque.cod* con el código objeto generado para el cuerpo del bucle.
- e. El bucle **for** se comporta de forma similar al de C, aunque en este caso la segunda expresión ha de ser de tipo booleano (la primera y la tercera, sin embargo, pueden ser de cualquier tipo): la primera expresión sirve de inicialización; la segunda es una condición que determina si se ha de ejecutar el cuerpo del bucle; y la tercera es una expresión de actualización que se ejecuta al final de cada pasada del bucle. La mecánica de ejecución del bucle es equivalente a la de una instrucción de tipo **while**.
- f. Los arrays se almacenan en la tabla de tipos indicando el límite inferior y superior de cada una de sus dimensiones; este almacenamiento se lleva a cabo en otras reglas de la gramática, que no has de considerar. Así, el tipo asociado a un *array* declarado como

```
int s[1..2][10..20][100..200];
```

sería el número 4 de la siguiente tabla de tipos:

NÚMERO	TIPO	LÍM. INF.	LÍM SUP.	TIPO BASE
0	ENTERO			
1	BOOLEANO			
2	ARRAY	100	200	0
3	ARRAY	10	20	2
4	ARRAY	1	2	3

- g. No se ha de generar código que compruebe en tiempo de ejecución que los índices están dentro del rango válido.

Señala claramente qué hace cualquier función auxiliar que utilices. Toda la información debe pasarse a través de atributos y no es posible usar ninguna variable global. Indica, además, de qué tipo es cada uno de los atributos que utilizas y cuál es su cometido.

*Nota:* recuerda utilizar la notación de los ETDS y *no* la de yacc.

✳ **65** Considera el siguiente fragmento de la especificación sintáctica de un lenguaje:

$$\begin{aligned}
 I &\longrightarrow \mathbf{print} E \\
 I &\longrightarrow \mathbf{read} L \\
 L &\longrightarrow L \mathbf{coma} R \\
 L &\longrightarrow R \\
 R &\longrightarrow \mathbf{id} A \\
 R &\longrightarrow R \mathbf{punto id} A \\
 A &\longrightarrow A [ E ] \\
 A &\longrightarrow \epsilon \\
 E &\longrightarrow E \mathbf{opsum} T \\
 E &\longrightarrow T \\
 T &\longrightarrow R \\
 T &\longrightarrow \mathbf{entero}
 \end{aligned}$$

Diseña un esquema de traducción dirigida por la sintaxis (ETDS) que genere el código adecuado para la máquina virtual `m2r` y que emita los mensajes de error semántico oportunos. Para ello, has de tener en cuenta los siguientes aspectos:

- En este lenguaje el único tipo de datos simple son los enteros, que ocupan **cuatro** posiciones de memoria.
- Como puedes ver en la gramática, el lenguaje permite usar *arrays* y registros intercalados (*arrays* de registros, registros con campos de tipo *array*, etc.). Por ejemplo, el lenguaje fuente permite variables como `c[3].d[j][k-1].e`.
- Aunque la instrucción de escritura (**print**) está diseñada para imprimir un único valor, la de lectura (**read**) permite leer con una sola instrucción (en el mismo orden en que aparecen en el programa fuente) los valores de más de una variable.
- Los arrays se almacenan en la tabla de tipos indicando el límite inferior y superior de cada una de sus dimensiones. Además, para cada registro se almacena en la tabla de tipos una referencia a la tabla de símbolos

en la que se especifican los campos correspondientes. La inserción de toda esta información en cada una de las tablas se lleva a cabo en otras reglas de la gramática, que no has de considerar.

Así, a una declaración como esta:

```
struct {int a; int b;} s[1..2][10..20][100..200];
```

le corresponderían las siguientes tablas:

TABLA DE SÍMBOLOS GLOBAL

NOMBRE	TIPO	DIRECCIÓN
s	4	0

TABLA DE SÍMBOLOS TS1

NOMBRE	TIPO	DIRECCIÓN
a	0	0
b	0	2

TABLA DE TIPOS

NÚMERO	TIPO	LÍM. INF.	LÍM SUP.	TIPO BASE	TAM.
0	ENTERO				
1	REGISTRO			TS1	4
2	ARRAY	100	200	1	
3	ARRAY	10	20	2	
4	ARRAY	1	2	3	

El campo TAM indica el tamaño total de los registros.

Señala claramente qué hace cualquier función auxiliar que utilices. Toda la información debe pasarse a través de atributos y no es posible usar ninguna variable global (excepto las correspondientes a la tabla de símbolos y de tipos, y la usada para determinar la posición de las temporales). Indica, además, de qué tipo es cada uno de los atributos que utilizas y cuál es su cometido.

*Nota:* recuerda utilizar la notación de los ETDS y *no* la de yacc.

\* **66** Explica con cierto detalle (indicando incluso las acciones del ETDS) cómo modificarías lo visto en clase para la generación de código m2r para *arrays* de forma que se realizara en tiempo de ejecución la comprobación de que los índices están dentro del rango permitido. ¿Cómo se detectaría si se cumple esa condición? ¿Cómo se emitiría el mensaje de error?

\* **67** Dada una declaración de un *array* como la siguiente:

```
int a[D1][D2][D3]
```

donde  $D_i$  es el tamaño de la  $i$ -ésima dimensión del *array*, una posible expresión para el cálculo de la dirección de memoria asociada al componente  $a[i][j][k]$  es:

$$\begin{aligned} \text{Dirección base}(a) &+ i \times (D_2 \times D_3 \times \text{Tamaño}(\text{int})) \\ &+ j \times (D_3 \times \text{Tamaño}(\text{int})) \\ &+ k \times \text{Tamaño}(\text{int}) \end{aligned}$$

Considera el siguiente fragmento de la especificación sintáctica de un lenguaje en el que se incluye la parte relacionada con el acceso a un componente de un *array*:

$$\begin{aligned} I &\longrightarrow \mathbf{print} E \\ I &\longrightarrow \mathbf{read} R \\ R &\longrightarrow \mathbf{id} A \\ A &\longrightarrow [ E ] A \\ A &\longrightarrow [ E ] \\ E &\longrightarrow R \\ E &\longrightarrow \mathbf{entero} \end{aligned}$$

**Apartado A.** Diseña un esquema de traducción dirigida por la sintaxis (ETDS) que genere el código adecuado para la máquina virtual `m2r` usando la expresión anterior para determinar la dirección en memoria de los datos, y que emita los mensajes de error semántico oportunos.

**Apartado B.** Indica una forma iterativa equivalente a la expresión anterior en la que en cada paso se use únicamente la información correspondiente a una de las dimensiones y el resultado del paso anterior. Demuestra que ambas formas de calcular la dirección en memoria son equivalentes (*pista*: transforma, para ello, la forma iterativa en la expresión de más arriba). A continuación, vuelve a diseñar un ETDS que genere el código adecuado para `m2r` usando la nueva forma iterativa.

Para los dos apartados anteriores, has de tener en cuenta los siguientes aspectos:

- En este lenguaje el único tipo de datos simple son los enteros, que ocupan **una** posición de memoria.
- Las instrucciones **print** y **read** son, respectivamente, las de escritura y lectura.
- No es necesario realizar el reciclado de las posiciones de memoria temporales.

- d. Los arrays se almacenan en la tabla de tipos indicando el tamaño  $D$  de cada una de sus dimensiones; los índices de cada dimensión van de 0 a  $D-1$ . La inserción de esta información en la tabla de tipos se lleva a cabo en otras reglas de la gramática, que no has de considerar.

Así, a una declaración como esta:

```
int s[2][20][200];
```

le corresponderían las siguientes tablas:

TABLA DE SÍMBOLOS

NOMBRE	TIPO	DIRECCIÓN
s	3	100

TABLA DE TIPOS

NÚMERO	TIPO	TAMAÑO	TIPO BASE
0	ENTERO	1	
1	ARRAY	200	0
2	ARRAY	20	1
3	ARRAY	2	2

Asume que el lenguaje fuente no permite la declaración de funciones. No se ha de generar código que compruebe en tiempo de ejecución que los índices están dentro del rango válido.

Señala claramente qué hace cualquier función auxiliar que utilices. Toda la información debe pasarse a través de atributos y no es posible usar ninguna variable global (excepto las correspondientes a la tabla de símbolos y de tipos, y la usada para determinar la posición de las temporales). Indica, además, de qué tipo es cada uno de los atributos que utilizas y cuál es su cometido.

*Nota:* recuerda utilizar la notación de los ETDS y *no* la de yacc.

❄ **68** Explica con cierto detalle (indicando incluso algunas acciones del ETDS) cómo modificarías lo visto en clase sobre la generación de código m2r para considerar la posibilidad de declarar y usar punteros a tipos simples al estilo de C:

```
int a;
int *p;    /* p es un puntero a entero */
a= 12;
p= &a;    /* p apunta a la dirección de la variable a */
print *p; /* imprime 12 */
```

Algunas preguntas que pueden orientar tu respuesta: ¿qué se almacenaría en las tablas de símbolos y de tipos?, ¿cómo se usaría esa información?, ¿cuál es el código objeto que se generaría para un código como el del ejemplo anterior?

\* **69** Considera el siguiente fragmento de la especificación sintáctica de un lenguaje:

$$\begin{array}{l}
 I \longrightarrow \mathbf{print} E \\
 I \longrightarrow \mathbf{read} R \\
 R \longrightarrow \mathbf{id} B \\
 R \longrightarrow \mathbf{id} B \mathbf{punto} R \\
 B \longrightarrow \epsilon \\
 B \longrightarrow A \\
 A \longrightarrow A [ E ] \\
 A \longrightarrow [ E ] \\
 E \longrightarrow E \mathbf{opsum} T \\
 E \longrightarrow T \\
 T \longrightarrow R \\
 T \longrightarrow \mathbf{entero}
 \end{array}$$

Diseña un esquema de traducción dirigida por la sintaxis (ETDS) que genere el código adecuado para la máquina virtual `m2r` y que emita los mensajes de error semántico oportunos. Para ello, has de tener en cuenta los siguientes aspectos:

- En este lenguaje el único tipo de datos simple son los enteros, que ocupan **una** posición de memoria.
- Como puedes ver en la gramática, el lenguaje permite usar *arrays* y registros intercalados (*arrays* de registros, registros con campos de tipo *array*, etc.). Por ejemplo, el lenguaje fuente permite variables como `c[3].d[j][k-1].e`.
- Las instrucciones **print** y **read** son, respectivamente, las de escritura y lectura.
- No es necesario realizar el reciclado de las posiciones de memoria temporales.
- Los arrays se almacenan en la tabla de tipos indicando el límite inferior y superior de cada una de sus dimensiones. Además, para cada registro se almacena en la tabla de tipos una referencia a la tabla de símbolos en la que se especifican los campos correspondientes. La inserción de

toda esta información en cada una de las tablas se lleva a cabo en otras reglas de la gramática, que no has de considerar.

Así, a una declaración como esta:

```
struct {int a; int b;} s[1..2][10..20][100..200];
```

le corresponderían las siguientes tablas:

TABLA DE SÍMBOLOS GLOBAL

NOMBRE	TIPO	DIRECCIÓN
s	4	0

TABLA DE SÍMBOLOS TS1

NOMBRE	TIPO	DIRECCIÓN
a	0	0
b	0	1

TABLA DE TIPOS

NÚMERO	TIPO	LÍM. INF.	LÍM SUP.	TIPO BASE	TAM.
0	ENTERO				1
1	REGISTRO			TS1	2
2	ARRAY	100	200	1	
3	ARRAY	10	20	2	
4	ARRAY	1	2	3	

El campo TAM indica el tamaño total de los registros.

Asume que el lenguaje fuente no permite la declaración de funciones. No se ha de generar código que compruebe en tiempo de ejecución que los índices están dentro del rango válido.

Señala claramente qué hace cualquier función auxiliar que utilices. Toda la información debe pasarse a través de atributos y no es posible usar ninguna variable global (excepto las correspondientes a la tabla de símbolos y de tipos, y la usada para determinar la posición de las temporales). Indica, además, de qué tipo es cada uno de los atributos que utilizas y cuál es su cometido.

*Nota:* recuerda utilizar la notación de los ETDS y *no* la de yacc.

\* **70** Explica con cierto detalle (indicando incluso algunas acciones del ETDS) cómo modificarías lo visto en clase para la generación de código m2r para considerar la posibilidad de definir tipos de datos al estilo de C:

```
typedef int bonoloto[7];

bonoloto a,b; /* a y b son arrays de 7 enteros */
```

¿Cómo se modificaría la gestión de las tablas de símbolos y de tipos?  
 ¿Cómo se comprobaría si dos tipos de datos son compatibles en, por ejemplo, una asignación?

✱ **71** Considera el siguiente fragmento de la especificación sintáctica de un lenguaje:

$$\begin{array}{l}
 I \longrightarrow \mathbf{print} R \\
 I \longrightarrow \mathbf{read} L \\
 L \longrightarrow L \mathbf{coma} R \\
 L \longrightarrow R \\
 R \longrightarrow \mathbf{id} A \\
 R \longrightarrow R \mathbf{punto id} A \\
 A \longrightarrow [\mathbf{entero}] A \\
 A \longrightarrow \epsilon
 \end{array}$$

Diseña un esquema de traducción dirigida por la sintaxis (ETDS) que genere el código adecuado para la máquina virtual `m2r` y que emita los mensajes de error semántico oportunos. Para ello, has de tener en cuenta los siguientes aspectos:

- En este lenguaje el único tipo de datos simple son los enteros, que ocupan **dos** posiciones de memoria.
- Como puedes ver en la gramática, el lenguaje permite usar *arrays* y registros intercalados (*arrays* de registros, registros con campos de tipo *array*, etc.). Por ejemplo, el lenguaje fuente permite variables como `c[3].d[5][7].e`.
- Los componentes de los *arrays* se indizan exclusivamente mediante constantes enteras; esto puede simplificar bastante la generación de código ligada a la regla de los corchetes.
- Aunque la instrucción de escritura (**print**) está diseñada para imprimir un único valor, la de lectura (**read**) permite leer con una sola instrucción (en el mismo orden en que aparecen en el programa fuente) los valores de más de una variable.
- Los *arrays* se almacenan en la tabla de tipos indicando el límite inferior y superior de cada una de sus dimensiones. Además, para cada registro

se almacena en la tabla de tipos una referencia a la tabla de símbolos en la que se especifican los campos correspondientes. La inserción de toda esta información en cada una de las tablas se lleva a cabo en otras reglas de la gramática, que no has de considerar.

Así, a una declaración como esta:

```
struct {int a; int b;} s[1..2][10..20][100..200];
```

le corresponderían las siguientes tablas:

TABLA DE SÍMBOLOS GLOBAL

NOMBRE	TIPO	DIRECCIÓN
s	4	0

TABLA DE SÍMBOLOS TS1

NOMBRE	TIPO	DIRECCIÓN
a	0	0
b	0	2

TABLA DE TIPOS

NÚMERO	TIPO	LÍM. INF.	LÍM SUP.	TIPO BASE	TAM.
0	ENTERO				
1	REGISTRO			TS1	4
2	ARRAY	100	200	1	
3	ARRAY	10	20	2	
4	ARRAY	1	2	3	

El campo TAM indica el tamaño total de los registros.

Señala claramente qué hace cualquier función auxiliar que utilices. Toda la información debe pasarse a través de atributos y no es posible usar ninguna variable global (excepto las correspondientes a la tabla de símbolos y de tipos, y la usada para determinar la posición de las temporales). Indica, además, de qué tipo es cada uno de los atributos que utilizas y cuál es su cometido.

*Nota:* recuerda utilizar la notación de los ETDS y *no* la de yacc.

\* **72** Considera el siguiente fragmento de la especificación sintáctica de un lenguaje, que corresponden a la declaración e inicialización de *arrays*:

$$\begin{aligned}
 Decl &\longrightarrow \textit{Tipo id } A := V ; \\
 A &\longrightarrow [ \textit{entero} ] A \\
 A &\longrightarrow \epsilon \\
 V &\longrightarrow \{L\} \\
 V &\longrightarrow \{E\} \\
 L &\longrightarrow L, V \\
 L &\longrightarrow V \\
 E &\longrightarrow \epsilon \\
 E &\longrightarrow I \\
 E &\longrightarrow \textit{entero} , E \\
 E &\longrightarrow \textit{entero} \\
 T &\longrightarrow \textit{int}
 \end{aligned}$$

Diseña un esquema de traducción dirigida por la sintaxis (ETDS) que rellene la tabla de símbolos y la tabla de tipos adecuadamente (según se indica a continuación) y que genere el código adecuado para inicializar los valores de los componentes del *array* (también como se indica a continuación).

Por ejemplo, ante una declaración como:

```
int s[2][3][2] = {{{1,2},{3,4},{}} , {{5,6},{},{8}}};
```

le corresponderían las siguientes tablas:

TABLA DE SÍMBOLOS

NOMBRE	TIPO	DIRECCIÓN
s	4	0

TABLA DE TIPOS

NÚMERO	TIPO	TIPO BASE	TAMAÑO
0	ENTERO		1
1	ARRAY	0	2
2	ARRAY	1	3
3	ARRAY	2	2

y la siguiente inicialización (con una visión lineal del *array*):

1	2	3	4	0	0	5	6	0	0	8	0
---	---	---	---	---	---	---	---	---	---	---	---

Además, has de tener en cuenta los siguientes aspectos:

- En este lenguaje el único tipo de dato simple es el entero, que ocupa una posición de memoria.

- b. Los *arrays* se almacenan en la tabla de tipos indicando el tamaño de cada dimensión  $D$ ; los componentes del array van de 0 a  $D - 1$  (como en C).
- c. A la hora de generar código, asume que el lenguaje fuente no permite la declaración de funciones ni de registros.
- d. No es necesario indicar todos los componentes de cualquiera de las dimensiones; los valores omitidos corresponden siempre a los de “más a la derecha” y se han de inicializar a cero. Por ejemplo, la declaración:

```
int s[1][2][3][2] = {{{{10}}}};
```

genera la siguiente inicialización en memoria:

10	0	0	0	0	0	0	0	0	0	0	0	0
----	---	---	---	---	---	---	---	---	---	---	---	---

- e. Matizando lo anterior, una lista de enteros solo puede aparecer en el nivel más interno (el asociado a la última dimensión); en otro caso, se ha de emitir un error semántico. Igualmente, solo pueden aparecer llaves vacías `{ }` en el nivel más interno y también debe emitirse un error semántico si lo hace en otro nivel.
- f. Si se indican más valores en la lista de valores de inicialización de una de las dimensiones que las impuestas por el tamaño de dicha dimensión, se ha de emitir un error semántico.
- g. *Pista:* una posible forma de generar el código para la inicialización es almacenar en el registro `B` la dirección de inicio del *array* e ir llevando un contador en el que se indique la posición relativa en la que guardar el siguiente componente. *Nota:* puedes asumir que el repertorio de instrucciones de `m2r` incluye el modo de direccionamiento `@B+A`, aunque muchas de las posibles soluciones del problema no lo necesitan.

Señala claramente qué hace cualquier función auxiliar que utilices. Toda la información debe pasarse a través de atributos y no es posible usar ninguna variable global (excepto las correspondientes a la tabla de símbolos y de tipos). Indica, además, de qué tipo es cada uno de los atributos que utilizas y cuál es su cometido.

*Nota:* recuerda utilizar la notación de los ETDS y *no* la de `yacc`.

✱ **73** Considera el siguiente fragmento de la especificación sintáctica de un lenguaje:

$$\begin{aligned}
 I &\longrightarrow R = R \\
 R &\longrightarrow \mathbf{id} B \\
 B &\longrightarrow \epsilon \\
 B &\longrightarrow A \\
 A &\longrightarrow [ E ] A \\
 A &\longrightarrow [ E ] \\
 E &\longrightarrow R \\
 E &\longrightarrow \mathbf{entero}
 \end{aligned}$$

Diseña un esquema de traducción dirigida por la sintaxis (ETDS) que genere el código adecuado para la máquina virtual `m2r` y que emita los mensajes de error semántico oportunos. Para ello, has de tener en cuenta los siguientes aspectos:

- En este lenguaje el único tipo de datos simple son los enteros, que ocupan **una** posición de memoria.
- Como puedes ver en la gramática, el lenguaje permite usar *arrays*. A diferencia de lo visto en las clases de teoría (y en la práctica) donde solo se permitía la asignación de componentes particulares del *array* (de tipo simple), este lenguaje permite realizar asignaciones de *arrays* completos o de *subarrays*, siempre que tengan el mismo número de dimensiones con el mismo tamaño en cada una de ellas. Por ejemplo, dadas las siguientes declaraciones de arrays:

```

int a[5][7][11];
int b[5][13][11];
int c[3][7][11];
int d[5][7][11];
int e[11];
int f[5][11][7];

```

se permiten las asignaciones `a=d`, `a[1]=c[2]`, `a[3][6]=b[3][8]`, `d[1][2]=e` o `a[3][6][1]=e[7]`; pero no se permite `a=b` por ser *arrays* de tamaños distintos (pese a tener el mismo número de dimensiones), o `a[3]=b[4]` o `a[2]=f[2]` por ser *subarrays* también de tamaños distintos en alguna de las dimensiones.

- No es necesario realizar el reciclado de las posiciones de memoria temporales.
- Los índices válidos a la hora de acceder a los componentes de un *array* están entre 0 y el tamaño correspondiente a cada dimensión menos uno (como en el lenguaje C).

- e. Los arrays se almacenan en la tabla de tipos indicando el tamaño y el tipo base de cada una de sus dimensiones. La inserción de esta información en cada una de las tablas se lleva a cabo en otras reglas de la gramática, que no has de considerar.
- f. Para simplificar la solución al problema, asume que la inserción de esta información se realiza reutilizando, siempre que sea posible, la información de tipos ya existentes en la tabla de tipos para *arrays* anteriores. Así, a una serie de declaraciones como la anterior le corresponderían las siguientes tablas:

TABLA DE SÍMBOLOS GLOBAL

NOMBRE	TIPO	DIRECCIÓN
a	3	0
b	5	385
c	6	1100
d	3	1331
e	1	1716
f	9	1727

TABLA DE TIPOS

NÚMERO	TIPO	TAMAÑO	TIPO BASE
0	ENTERO	1	
1	ARRAY	11	0
2	ARRAY	7	1
3	ARRAY	5	2
4	ARRAY	13	1
5	ARRAY	5	4
6	ARRAY	3	2
7	ARRAY	7	0
8	ARRAY	11	7
9	ARRAY	5	8

Como ves, la reutilización de los tipos se traduce en, por ejemplo, la adición de una única fila para el caso del *array* tridimensional c de manera que se reutilizan las filas 1 y 2 insertadas, en principio, para el *array* a.

- g. La asignación de *arrays* o de *subarrays* implica generar código que realice la copia del bloque de datos correspondiente a la referencia de la derecha de la asignación en la zona de memoria asociada a la referencia de la parte izquierda.
- h. Asume que el lenguaje fuente no permite la declaración de funciones. No se ha de generar código que compruebe en tiempo de ejecución que los índices están dentro del rango válido.

Señala claramente qué hace cualquier función auxiliar que utilices. Toda la información debe pasarse a través de atributos y no es posible usar ninguna variable global (excepto las correspondientes a la tabla de símbolos y de tipos, y la usada para determinar la posición de las temporales). Indica, además, de qué tipo es cada uno de los atributos que utilizas y cuál es su cometido.

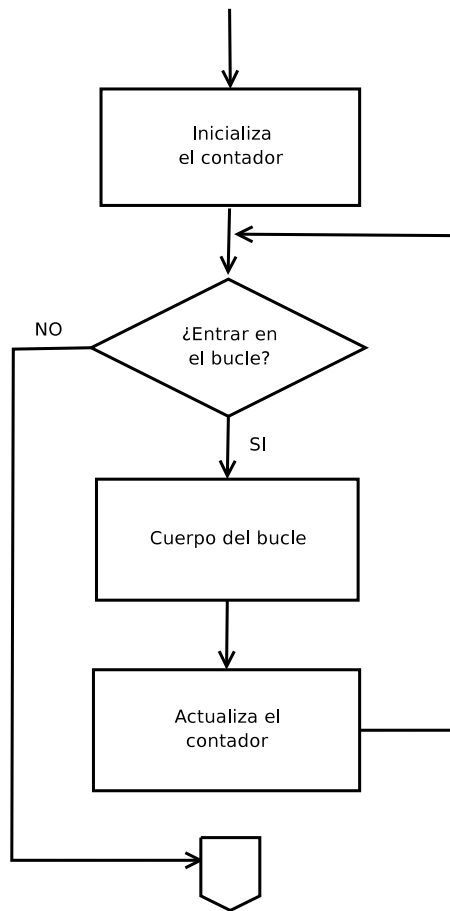
*Nota:* recuerda utilizar la notación de los ETDS y *no* la de yacc.

✱ **74** Considera el siguiente fragmento de la especificación sintáctica de un lenguaje:

$$\begin{array}{ll}
 S & \longrightarrow I \\
 I & \longrightarrow \text{for } id := E \text{ to } E \text{ step } Signo \text{ entero } Bloque \text{ next } id ; \\
 I & \longrightarrow \text{break if } E ; \\
 I & \longrightarrow \text{continue if } E ; \\
 I & \longrightarrow \text{print } E \\
 Bloque & \longrightarrow \{ Sec \} \\
 Sec & \longrightarrow Sec I \\
 Sec & \longrightarrow I \\
 Signo & \longrightarrow \text{opsum} \\
 Signo & \longrightarrow \epsilon
 \end{array}$$

Diseña un esquema de traducción dirigida por la sintaxis (ETDS) que genere el código adecuado para la máquina virtual `m2r` y que emita los mensajes de error semántico oportunos. Para ello, has de tener en cuenta los siguientes aspectos:

- En este lenguaje el único tipo de datos simple son los enteros, que ocupan **una** posición de memoria.
- Como puedes ver en la gramática, esta parte del lenguaje se encarga de los elementos asociados a un bucle tipo `for` en el que se indica el valor inicial del contador, el valor superior y los incrementos (positivos o negativos) del contador tras cada iteración.
- El código generado para el bucle `for` ha de seguir la estructura de la figura 1.
- La instrucción `break` se utiliza para terminar la ejecución del bucle más cercano si la condición indicada por la expresión es cierta. La instrucción `continue` se utiliza para saltar el resto de la iteración actual del bucle más cercano. El bucle no necesariamente termina cuando se



**Figura 1:** Diagrama de flujo para la instrucción `for`.

encuentra la instrucción `continue`; simplemente, no se ejecutan las instrucciones que se encuentran a continuación de ella y se salta directamente a la siguiente iteración.

Por ejemplo, el siguiente bucle imprime los valores 1, 3 y 5:

```
for i:= 1 to 100 step 2 {  
    print i;  
    break if i>4;  
} next i;
```

Por otro lado, el siguiente bucle imprime en orden descendente los números pares entre 1 y 100:

```
for j:= 100 to 1 step -1 {
```

```
    continue if j mod 2 != 0;
    print j;
} next j;
```

Además, el siguiente bucle se ejecuta una sola vez:

```
for j:= 100 to 100 step -1 {
    print j;
} next j;
```

Finalmente, este bucle no se ejecuta una sola vez (sería equivalente a un bucle de la forma `for(i=100;i>=101;i--)` en C):

```
for j:= 100 to 101 step -1 {
    print j;
} next j;
```

- e. La instrucción **print** es la instrucción de escritura.
- f. Asume que las reglas (que no has de considerar) asociadas al no terminal  $E$  (encargado de las expresiones) asignan a  $E$  sendos atributos con el código y la dirección temporal donde se guarda el resultado de la expresión; no es necesario un atributo para el tipo ya que este es siempre entero (en el caso de expresiones booleanas, este entero valdrá cero si la condición es falsa y uno en otro caso).
- g. La variable que acompaña al **next** ha de ser la misma que aparece junto al **for**; en otro caso, se ha de emitir un error semántico.
- h. No es posible usar ni **break** ni **continue** fuera de un bucle **for**; si se detecta esta situación, se ha de emitir un error semántico.

Señala claramente qué hace cualquier función auxiliar que utilices. Toda la información debe pasarse a través de atributos y no es posible usar ninguna variable global (excepto las correspondientes a la tabla de símbolos y de tipos, y la usada para determinar la posición de las temporales). Indica, además, de qué tipo es cada uno de los atributos que utilizas y cuál es su cometido.

*Nota:* recuerda utilizar la notación de los ETDS y *no* la de yacc.

✱ **75** Considera el siguiente fragmento de la especificación sintáctica de un lenguaje:

$$\begin{array}{l}
 I \quad \longrightarrow \quad \mathbf{id} \ R = \mathbf{id} \ R \\
 R \quad \longrightarrow \quad R \ [ \mathbf{entero} ] \\
 R \quad \longrightarrow \quad \epsilon
 \end{array}$$

Diseña un esquema de traducción dirigida por la sintaxis (ETDS) que genere el código adecuado para la máquina virtual `m2r` y que emita los mensajes de error semántico oportunos. Para ello, has de tener en cuenta los siguientes aspectos:

- a. Los tipos de datos simples de este lenguaje son los enteros y los reales, que ocupan **una** posición de memoria. Está permitida la asignación entre tipos distintos, mediante las adecuadas conversiones de tipo.
- b. Esta parte de la gramática tiene que ver con la asignación entre variables simples o componentes de *arrays*.
- c. Date cuenta de que en el interior de los corchetes solo pueden aparecer constantes enteras (y no expresiones arbitrarias). Esto simplifica considerablemente el código objeto generado; el que consideres adecuadamente esta simplificación se tendrá en cuenta a la hora de corregir tu solución. *Pista*: considera qué operaciones pueden realizarse en tiempo de ejecución y cuáles en tiempo de compilación.
- d. No es necesario realizar el reciclado de las posiciones de memoria temporales.
- e. Los índices válidos a la hora de acceder a los componentes de un *array* están entre 0 y el tamaño correspondiente a cada dimensión menos uno (como en el lenguaje C).
- f. Los arrays se almacenan en la tabla de tipos indicando el tamaño y el tipo base de cada una de sus dimensiones. La inserción de esta información en cada una de las tablas se lleva a cabo en otras reglas de la gramática, que no has de considerar.

Así, a las siguientes declaraciones de arrays:

```
int a[7][9];
int e[9];
```

le corresponderían las siguientes tablas:

NOMBRE	TIPO	DIRECCIÓN
a	3	0
e	4	63

TABLA DE TIPOS

NÚMERO	TIPO	TAMAÑO	TIPO BASE
0	ENTERO	1	
1	REAL	1	
2	ARRAY	9	0
3	ARRAY	7	2
4	ARRAY	9	0

- g. Asume que el lenguaje fuente no permite la declaración de funciones.
- h. Es necesario indicar tantos corchetes como dimensiones tenga el *array*; ni más ni menos.
- i. No se ha de generar código que compruebe en tiempo de ejecución que los índices están dentro del rango válido.

Señala claramente qué hace cualquier función auxiliar que utilices. Toda la información debe pasarse a través de atributos y no es posible usar ninguna variable global (excepto las correspondientes a la tabla de símbolos y de tipos, y la usada para determinar la posición de las temporales). Indica, además, de qué tipo es cada uno de los atributos que utilizas y cuál es su cometido.

*Nota:* recuerda utilizar la notación de los ETDS y *no* la de yacc.

\* **76** Considera el siguiente fragmento de la especificación sintáctica de un lenguaje:

$$\begin{aligned}
 I &\longrightarrow \mathbf{print} E \\
 E &\longrightarrow E \mathbf{opsum} T \\
 E &\longrightarrow T \\
 T &\longrightarrow \mathbf{id} \\
 T &\longrightarrow \mathbf{id opinc} \\
 T &\longrightarrow \mathbf{opinc id}
 \end{aligned}$$

Diseña un esquema de traducción dirigida por la sintaxis (ETDS) que genere el código adecuado para la máquina virtual **m2r** y que emita los mensajes de error semántico oportunos. Para ello, has de tener en cuenta los siguientes aspectos:

- a. Los tipos de datos simples de este lenguaje son los enteros y los reales, que ocupan **una** posición de memoria.
- b. Los posibles lexemas asociados al *token* **opinc** son **++** y **--**.

- c. Esta parte del lenguaje se encarga principalmente de las subexpresiones de preincremento, predecremento, postincremento y postdecremento. Estos operadores solo trabajan con variables enteras; debe emitirse un error semántico si la variable es de tipo real.

Cuando el operador precede al operando, el resultado de la subexpresión será el valor obtenido tras realizar la operación; en otro caso, será el valor del operando justo antes de realizarla. El valor resultante de la operación de incremento o decremento deberá almacenarse en ambos casos en la variable que actúa como operando.

El comportamiento es análogo al que se produce en Java con estos operadores, es decir, si la variable entera `a` vale 1, el resultado de la expresión `a++ + a++` ha de ser 3 ( $1 + 2$ ); por tanto, el código del incremento se ha de ejecutar antes de la suma, en este caso. Otro ejemplo: si `a` vale 1, el resultado de la expresión `++a + ++a - a++` es  $2 + 3 - 3 = 2$  y al acabar la evaluación de la expresión la variable `a` valdrá 4.

- d. Asume que el lenguaje no permite trabajar con funciones.

Señala claramente qué hace cualquier función auxiliar que utilices. Toda la información debe pasarse a través de atributos y no es posible usar ninguna variable global (excepto las correspondientes a la tabla de símbolos y de tipos, y la usada para determinar la posición de las temporales). Indica, además, de qué tipo es cada uno de los atributos que utilizas y cuál es su cometido.

*Nota:* recuerda utilizar la notación de los ETDS y *no* la de yacc.

\* **77** Considera el siguiente fragmento de la especificación sintáctica de un lenguaje:

$$\begin{aligned} E &\longrightarrow E * T \\ E &\longrightarrow T \\ T &\longrightarrow | F | \\ T &\longrightarrow F \\ F &\longrightarrow \mathbf{id} \\ F &\longrightarrow \mathbf{entero} \\ F &\longrightarrow \mathbf{real} \\ F &\longrightarrow ( E ) \end{aligned}$$

Diseña un esquema de traducción dirigida por la sintaxis (ETDS) que genere el código adecuado para la máquina virtual `m2r` y que emita los mensajes de error semántico oportunos. Para ello, has de tener en cuenta los siguientes aspectos:

- a. Los tipos de datos simples de este lenguaje son los enteros y los reales, que ocupan una posición de memoria.
- b. Los identificadores pueden corresponder a variables de tipo simple o de tipo *array*.
- c. Esta parte de la gramática caracteriza los operadores de producto (\*) y de *norma* (|·|). El operador de producto está *sobrecargado* de forma que se puede aplicar tanto a dos *escalares* (en este caso, variables o constantes de tipo entero o real) como a dos vectores (en este caso, *arrays* unidimensionales con el mismo tamaño), pero no a combinaciones de un vector y un escalar. Por otro lado, el operador de norma se puede aplicar únicamente a un operando de tipo *array* unidimensional.
- d. Si no se cumple alguna de las restricciones anteriores, se ha de emitir el oportuno mensaje de error.
- e. El producto escalar de dos vectores se define como:

$$\mathbf{x} \cdot \mathbf{y} = \sum_{i=1}^n x_i y_i,$$

donde  $n$  es la longitud del vector.

- f. La norma de un vector se define como:

$$|\mathbf{x}| = \sqrt{\sum_{i=1}^n x_i^2} = \sqrt{\mathbf{x} \cdot \mathbf{x}}$$

- g. Por ejemplo, si los *arrays*  $\mathbf{a}$  y  $\mathbf{b}$  toman los valores  $\mathbf{a}=[3,4]$  y  $\mathbf{b}=[2,3]$ , el resultado de  $|\mathbf{a}| * (\mathbf{a} * \mathbf{b})$  será  $\sqrt{3^2 + 4^2} \times (3 \times 2 + 4 \times 3) = 5 \times 18 = 90$  (en realidad, 90.0 probablemente).
- h. Como esta parte de la gramática no permite el acceso a *subarrays*, no has de considerar el caso del producto de *subarrays* de dimensión 1.
- i. Si necesitas añadir alguna instrucción *escalar* al repertorio de instrucciones de `m2r`, puedes hacerlo, pero has de definir claramente el comportamiento de la instrucción.
- j. Los *arrays* se almacenan en la tabla de tipos indicando el tamaño y el tipo base de cada una de sus dimensiones. La inserción de esta información en cada una de las tablas se lleva a cabo en otras reglas de la gramática, que no has de considerar.

Así, a las siguientes declaraciones de *arrays*:

```
int a[7][9];
int e[9];
```

le corresponderían las siguientes tablas:

TABLA DE SÍMBOLOS GLOBAL

NOMBRE	TIPO	DIRECCIÓN
a	3	0
e	4	63

TABLA DE TIPOS

NÚMERO	TIPO	TAMAÑO	TIPO BASE
0	ENTERO	1	
1	REAL	1	
2	ARRAY	9	0
3	ARRAY	7	2
4	ARRAY	9	0

- k. Asume que el lenguaje fuente no permite la declaración de funciones.

Señala claramente qué hace cualquier función auxiliar que utilices. Toda la información debe pasarse a través de atributos y no es posible usar ninguna variable global (excepto las correspondientes a la tabla de símbolos y de tipos, y la usada para determinar la posición de las temporales). Indica, además, de qué tipo es cada uno de los atributos que utilizas y cuál es su cometido.

*Nota:* recuerda utilizar la notación de los ETDS y *no* la de yacc.

## Subprogramas

✱ **78** Indica el código objeto `m2r` que se generaría para el siguiente programa fuente, asumiendo un entorno de ejecución basado en pila.

Comenta brevemente cada una de las instrucciones objeto.

```
real a;
function real g (real x,real y) {
    x= y;
    real b;
    b= x*y;
    return b-a*2.0;
}
```

```

program {
  a= 3.14159;
  a= 40+g(50,0.5)+g(80.0,1.0);
}

```

✱ **79** Considera el siguiente fragmento de la especificación sintáctica de un lenguaje:

$$\begin{aligned}
 I &\longrightarrow \mathbf{print} T \\
 I &\longrightarrow \mathbf{read} R \\
 T &\longrightarrow T + F \\
 T &\longrightarrow F \\
 F &\longrightarrow \mathbf{cteint} \\
 F &\longrightarrow \mathbf{id} ( P ) \\
 F &\longrightarrow R \\
 F &\longrightarrow ( T ) \\
 P &\longrightarrow \epsilon \\
 P &\longrightarrow L \\
 L &\longrightarrow L \mathbf{coma} T \\
 L &\longrightarrow T \\
 R &\longrightarrow \mathbf{id} D \\
 D &\longrightarrow D [ T ] \\
 D &\longrightarrow \epsilon
 \end{aligned}$$

Diseña un esquema de traducción dirigida por la sintaxis (ETDS) que genere el código adecuado para la máquina virtual `m2r` y que emita los mensajes de error semántico oportunos. Para ello, has de tener en cuenta los siguientes aspectos:

- En este lenguaje el único tipo de datos simple son los enteros, que ocupan una única posición de memoria.
- Como puedes ver en la gramática, el lenguaje permite usar *arrays* y efectuar llamadas a funciones (bajo un entorno de ejecución basado en pila).
- Como el único tipo de datos del lenguaje son los enteros, es suficiente almacenar en la tabla de símbolos el número de argumentos que tiene cada función y no es necesario introducir ninguna información adicional sobre las funciones en la tabla de tipos.

Los arrays se almacenan en la tabla de tipos indicando el tamaño de cada una de sus dimensiones (el límite inferior es siempre 0).

La inserción de toda esta información en cada una de las tablas se lleva a cabo en otras reglas de la gramática, que no has de considerar.

Así, a un *array* y una función como la del siguiente ejemplo:

```
int s[2][20][200];
int function f (int a, int b, int c);
```

les corresponderían las siguientes tablas:

TABLA DE SÍMBOLOS

NOMBRE	TIPO	DIRECCIÓN	NÚM. PARÁMETROS	ETIQ.	ÁMBITO
s	3	0	–	–	0
f	FUNC	–	3	L1	0

TABLA DE TIPOS

NÚMERO	TIPO	TAMAÑO	TIPO BASE
0	ENTERO		
1	ARRAY	200	0
2	ARRAY	20	1
3	ARRAY	2	2

- d. No se ha de generar código que compruebe en tiempo de ejecución que los índices están dentro del rango válido. Sin embargo, se ha de tener en cuenta el siguiente aspecto importante: si el valor de la expresión (en el no terminal  $T$ ) que aparece entre corchetes es negativo, se ha de interpretar como un acceso desde el final del *array*, es decir, [-1] se refiere a la última posición, [-2] a la penúltima, etc. La comprobación del signo de la expresión se ha de hacer, evidentemente, en tiempo de ejecución.

Señala claramente qué hace cualquier función auxiliar que utilices. Toda la información debe pasarse a través de atributos y no es posible usar ninguna variable global (excepto las correspondientes a la tabla de símbolos y de tipos, y la usada para determinar la posición de las temporales). Indica, además, de qué tipo es cada uno de los atributos que utilizas y cuál es su cometido.

*Nota:* recuerda utilizar la notación de los ETDS y *no* la de yacc.

\* **80** Indica el código objeto `m2r` que se generaría para el siguiente fragmento de programa fuente, asumiendo un entorno de ejecución basado

en pila y que el lenguaje no permite el uso de *arrays* ni de *registros*. La variable “a” es una variable global.

Comenta brevemente cada una de las instrucciones objeto.

```
int a;
function int g (int b) {
    if (a < b)
        return 0;
    a=a-1;
    return a+g(b);
}
```

❄ **81** Para permitir que se invoque una función que aún no se ha definido en el programa fuente muchos lenguajes (como C, por ejemplo) contemplan la declaración del prototipo de una función. Explica cómo implementarías un sistema de este tipo, inspirándote para ello en un lenguaje fuente como el usado en clase, que no permite los prototipos de funciones.

❄ **82** Indica el código objeto `m2r` que se generaría para el siguiente fragmento de programa fuente, asumiendo un entorno de ejecución basado en pila y que el lenguaje no permite el uso de *arrays* ni de *registros*.

Comenta brevemente cada una de las instrucciones objeto.

```
function int fib (int n) {
    if (n > 2)
        return fib(n-1)+fib(n-2);
    return 1;
}
```

❄ **83** Explica con cierto detalle (indicando incluso algunas acciones del ETDS) cómo modificarías lo visto en clase sobre la generación de código `m2r` para considerar la posibilidad de declarar funciones *sobrecargadas* al estilo de C++:

```
int f(int a);
int f(int a,int b);
int f(real a,real b);
```

¿Cómo se modificaría la gestión de las tablas de símbolos y de tipos?  
¿Cómo se decide qué función invocar en una llamada determinada? Dado el

ejemplo anterior, ¿cómo se determina la función a invocar cuando se hace una llamada como `f(3,3.14)`?

❄ **84** Indica el código objeto `m2r` que se generaría para el siguiente fragmento de programa fuente, asumiendo un entorno de ejecución basado en pila y que el lenguaje no permite el uso de *arrays* ni de *registros*.

Comenta brevemente cada una de las instrucciones objeto.

```
int f (int a) {
    return 2*a;
}
program {
    print(f(2));
}
```

❄ **85** Explica con cierto detalle (indicando incluso algunas acciones del ETDS) cómo modificarías lo visto en clase (o en tu práctica) sobre la generación de código `m2r` para funciones para considerar la posibilidad de pasar parámetros *por referencia*. Considera el siguiente ejemplo:

```
function suma (real v, real:ref r) {
    v= r+v;
    r= v;
}

program {
    int a,b,c;
    a=1;
    b=5;
    suma(a,b);
    print a; # imprime 1
    print b; # imprime 6
}
```

En el ejemplo anterior, el parámetro `a` se pasa por valor (igual que en la práctica), por lo que aunque el valor del argumento `v` se cambia dentro de la función, este cambio no repercute en la variable `a` usada en la llamada. Por otro lado, el parámetro `b` se pasa por referencia, por lo que dado que el argumento `r` se cambia en el interior de la función, este cambio se repercute a la variable `b` del programa principal.

¿Cómo se modificaría la gestión de las tablas de símbolos y de tipos?  
¿Cómo se gestionaría la dependencia entre el argumento y el parámetro?  
¿Cuál sería el código generado dentro de una función para el acceso a los parámetros pasados por referencia?

## Más problemas

❄ **86** Aprovechando la experiencia adquirida en la resolución de los problemas anteriores, diseña tus propios problemas y resuélvelos.