Problemas de Procesadores de Lenguaje

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

Departament de Llenguatges i Sistemes Informàtics

Universitat d'Alacant

Junio de 2007*

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. Algunos de los problemas relacionados con la generación de código CIL son una adaptación de problemas anteriores en los que debía generarse código `m2r` (un lenguaje objeto usado en las asignaturas mencionadas hasta el curso 2006–2007). 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
*	**todos**
.	**conextension**
/	**raiz**
.	**actual**

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; diciembre 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
`<>`	**distinto**
`<`	**menor**
`>`	**mayor**
`+`	**opsuma**
`-`	**opsuma**
`++`	**incremento**
`--`	**decremento**
`->`	**desref**
`--->`	**dobleref**
`:=`	**asig**

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
`>`	**mayor**
`+-`	**masmenos**
`+`	**opsuma**
`-`	**opsuma**
`++`	**incremento**
`--`	**decremento**
`->`	**desref**
`--->`	**dobleref**

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**.

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

Por ejemplo, ante una entrada como *ae**a**e**a**abde**a**e**a***, 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:

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

❄ **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:

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

Análisis sintáctico descendente

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

$$\begin{array}{lcl} S & \longrightarrow & \otimes \boxdot S \mid B \\ B & \longrightarrow & C\ D \\ C & \longrightarrow & \triangleq C \mid \epsilon \\ D & \longrightarrow & \approx D \mid \blacktriangleleft \mid \epsilon \end{array}$$

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

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

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

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

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

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

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

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

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

$$\begin{array}{lcl} S & \longrightarrow & X \mid \ominus \pm S \\ X & \longrightarrow & T\ Y \\ Y & \longrightarrow & \star \mid \bowtie Y \mid \epsilon \\ T & \longrightarrow & \cap\ T \mid \epsilon \end{array}$$

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.

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

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

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

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

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

❄ **18** 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).

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

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

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

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

Después haz la traza del análisis de la cadena de entrada **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.

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

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

Análisis sintáctico ascendente

❄ **23** Dada la siguiente gramática:

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

Diseña el autómata aceptor de prefijos viables y haz la traza del análisis ascendente de las cadenas "**b**" y "ϵ". Utiliza la siguiente plantilla para hacer la traza:

Pila	Entrada	Acción
⋮	⋮	⋮

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

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

❄ **26** ¿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.

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

$$\begin{array}{lll} E & \longrightarrow & \mathbf{id} \\ E & \longrightarrow & \mathbf{id}\ (E\,) \\ E & \longrightarrow & E + \mathbf{id} \end{array}$$

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?

❄ **28** ¿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.

❄ **29** 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.

❄ **30** 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.

❄ **31** 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.

❄ **32** 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	r5	1	2
1					acepta		
2				d5	r2		
3				d6			
4	d3	d4	r5	r5	r5		7
5	d3	d4	r5	r5	r5	8	
6	d3	d4	r5	r5	r5		9
7			d10				
8					r1		
9			r4	r4	r4		
10			r3	r3	r3		

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

$$
\begin{array}{lllll}
(1) & S & \longrightarrow & L\ \mathbf{,}\ S \\
(2) & S & \longrightarrow & L \\
(3) & L & \longrightarrow & \mathbf{(}\ L\ \mathbf{)} \\
(4) & L & \longrightarrow & \mathbf{a}\ \mathbf{,}\ L \\
(5) & L & \longrightarrow & \epsilon
\end{array}
$$

❄ **33** ¿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?

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

❄ **35** Dada la siguiente gramática:

$$
\begin{array}{llll}
1) & E & \longrightarrow & \mathbf{id} \\
2) & E & \longrightarrow & \mathbf{id}\ (E\,) \\
3) & E & \longrightarrow & E + \mathbf{id}
\end{array}
$$

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
⋮	⋮	⋮

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

❄ **37** 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.

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

$$\begin{array}{lcl} S & \longrightarrow & T\,\mathbf{a}\,T\,\mathbf{b} \\ S & \longrightarrow & V\,\mathbf{b}\,V\,\mathbf{a} \\ T & \longrightarrow & \epsilon \\ V & \longrightarrow & \epsilon \end{array}$$

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 calculas las tablas de análisis sintáctico.

Traducción dirigida por la sintaxis

❄ **39** 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.

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

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.

❄ **40** 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**.

$$
\begin{array}{lcl}
X & \longrightarrow & S \\
S & \longrightarrow & A\ B \\
S & \longrightarrow & A\ C \\
A & \longrightarrow & \mathbf{a}\ \mathbf{e}\ A \\
A & \longrightarrow & \mathbf{a}\ \mathbf{d}\ \mathbf{a}\ A \\
A & \longrightarrow & \epsilon \\
A & \longrightarrow & \mathbf{e}\ \mathbf{a} \\
B & \longrightarrow & \mathbf{d}\ \mathbf{e}\ \mathbf{a}\ \mathbf{d}\ \mathbf{a} \\
B & \longrightarrow & \mathbf{d}\ \mathbf{e}\ \mathbf{a}\ \mathbf{e} \\
C & \longrightarrow & D\ \mathbf{d} \\
D & \longrightarrow & \mathbf{a} \\
D & \longrightarrow & \epsilon
\end{array}
$$

Por ejemplo, ante una entrada como aeae**a**dae**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.

❄ **41** 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.

❄ **42** 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).

$$
\begin{array}{lcl}
S & \longrightarrow & L \\
L & \longrightarrow & [\ T\ ] \\
L & \longrightarrow & [\ ] \\
T & \longrightarrow & F\ \textbf{coma}\ T \\
T & \longrightarrow & F \\
F & \longrightarrow & \textbf{a} \\
F & \longrightarrow & \textbf{b} \\
F & \longrightarrow & L
\end{array}
$$

Por ejemplo, ante una entrada como

[a, [b, b, a, [b]], [ ]]

el ETDS debe generar

[**0**, a, [**1**, b, b, a, [**2**, b, **1**], **4**], [**1**, **0**], **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.

❄ **43** 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 ⊕ 3, postfija (1 ⋆ 5) ⊕ 4
```

será

```
⋆ 1 ⊕ 2 3, 1 5 ⋆ 4 ⊕
```

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
$\star$	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.

❄ **44** 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:

a) aquellas subexpresiones rodeadas por **dos** o más parejas de paréntesis

b) 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.

❄ **45** 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?

❄ **46** La siguiente gramática define el lenguaje de los números romanos menores de 100:

$$
\begin{array}{lcl}
S & \longrightarrow & \mathit{Decenas}\ \mathit{Unidades} \\
\mathit{Decenas} & \longrightarrow & \mathit{Dec} \\
\mathit{Decenas} & \longrightarrow & \mathbf{XL} \\
\mathit{Decenas} & \longrightarrow & \mathbf{L}\ \mathit{Dec} \\
\mathit{Decenas} & \longrightarrow & \mathbf{XC} \\
\mathit{Dec} & \longrightarrow & \mathit{Dec}\ \mathbf{X} \\
\mathit{Dec} & \longrightarrow & \epsilon \\
\mathit{Unidades} & \longrightarrow & \mathit{Unid} \\
\mathit{Unidades} & \longrightarrow & \mathbf{IV} \\
\mathit{Unidades} & \longrightarrow & \mathbf{V}\ \mathit{Unid} \\
\mathit{Unidades} & \longrightarrow & \mathbf{IX} \\
\mathit{Unid} & \longrightarrow & \mathit{Unid}\ \mathbf{I} \\
\mathit{Unid} & \longrightarrow & \epsilon
\end{array}
$$

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 **X** en *Dec* y de **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 **VIIII** 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),...

❄ **47** 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)`.

$$
\begin{array}{lcl}
E & \longrightarrow & E\ \textbf{opsum}\ T \\
E & \longrightarrow & T \\
T & \longrightarrow & \textbf{id} \\
T & \longrightarrow & \textbf{entero} \\
T & \longrightarrow & (\ E\ )
\end{array}
$$

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.

❄ **48** 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.

❄ **49** 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`.

$$\begin{array}{lcl} S & \longrightarrow & D \\ D & \longrightarrow & D\ B \mid B \\ B & \longrightarrow & \textbf{cero} \mid \textbf{uno} \end{array}$$

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

$$\begin{array}{lcl} S & \longrightarrow & D \\ D & \longrightarrow & B\ D \mid B \\ B & \longrightarrow & \textbf{cero} \mid \textbf{uno} \end{array}$$

❄ **51** 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 || b) && !c` será `and(or(a,b),not(c))` y la traducción de `!!!!!(a && !!b && c)` será `not(and(a,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.

❄ **52** 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.

$$
\begin{array}{lcl}
S & \longrightarrow & C \\
C & \longrightarrow & \textbf{class id \{ } B\ V \textbf{ \}} \\
B & \longrightarrow & \textbf{public :} P \mid \epsilon \\
V & \longrightarrow & \textbf{private :} P \mid \epsilon \\
P & \longrightarrow & D\ P \mid \epsilon \\
D & \longrightarrow & T \textbf{ id (} T \textbf{ id } L \textbf{)} \mid C \\
L & \longrightarrow & \textbf{,} T \textbf{ id } L \mid \epsilon \\
T & \longrightarrow & \textbf{int} \mid \textbf{float}
\end{array}
$$

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:

```
clase A {
  público:
    A::f1 (entero x real -> entero)
  privado:
    clase A::B {
      privado:
        A::B::f2 (real x real x real -> real)
        clase 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.

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

$$\begin{array}{lcl} S & \longrightarrow & L \\ L & \longrightarrow & [\ T\ ] \\ L & \longrightarrow & [\ ] \\ T & \longrightarrow & F\ \mathbf{,}\ T \\ T & \longrightarrow & F \\ F & \longrightarrow & \mathbf{entero} \\ F & \longrightarrow & L \end{array}$$

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.

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

$$\begin{array}{lcl} S & \longrightarrow & A\ \{\ L.h := A.s\ \}\ L\ \mathbf{pyc}\ \{\ R.k := A.s\ \}\ R \\ & & \{\ S.h := L.s \,||\, R.s\ \} \\ A & \longrightarrow & \mathbf{def}\ \{\ A.s := \mathbf{def}\ .lexema\ \} \\ A & \longrightarrow & \epsilon\,\{\ A.s := \text{“ ”}\ \} \\ L & \longrightarrow & \{\ L_1.h := L.h\ \}\ L_1\ \mathbf{coma\ id} \\ & & \{\ L.s := L_1.s \,||\, \text{“,”} \,||\, L.h \,||\, \mathbf{id}.lexema\ \} \\ L & \longrightarrow & \mathbf{id}\ \{\ L.s := L.h \,||\, \mathbf{id}.lexema\ \} \\ R & \longrightarrow & \{\ R_1.k := R.k\ \}\ R_1\ \mathbf{pyc\ id} \\ & & \{\ R.s := R_1.s \,||\, \text{“;”} \,||\, R.k \,||\, \mathbf{id}.lexema\ \} \\ R & \longrightarrow & \mathbf{var}\ \{\ L.h := R.k\ \}\ L\ \{\ R.s := L.s\ \} \end{array}$$

❄ **55** Traduce a notación de `yacc` el siguiente ETDS:

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

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

$$
\begin{array}{lcl}
S & \longrightarrow & \textbf{prop (entero , } L \textbf{)} \\
L & \longrightarrow & [\ T\ ] \\
L & \longrightarrow & [\ ] \\
T & \longrightarrow & F\textbf{ , } T \\
T & \longrightarrow & F \\
F & \longrightarrow & \textbf{entero} \\
F & \longrightarrow & L
\end{array}
$$

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:

```
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.

❄ **57** 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**.

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

Por ejemplo, ante una entrada como a**ba**b**c**cbc, 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.

❄ **58** La siguiente gramática genera expresiones infijas de sumas y multiplicaciones de constantes enteras y variables.

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

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.

❄ **59** La siguiente gramática genera expresiones infijas de sumas y multiplicaciones de constantes enteras y variables.

$$
\begin{array}{lcl}
X & \longrightarrow & E \\
E & \longrightarrow & E + T \\
E & \longrightarrow & T \\
T & \longrightarrow & T * F \\
T & \longrightarrow & F \\
F & \longrightarrow & \textbf{id} \\
F & \longrightarrow & \textbf{entero}
\end{array}
$$

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.

❄ **60** Los operadores del lenguaje de programación orientado a objetos Smalltalk no siguen las reglas habituales de asociatividad y precendencia; 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

❄ **61** Este no es un problema en sí, sino una explicación común a varios de los problemas de esta sección, que se presenta aquí para evitar la repetición y aumentar la claridad.

En algunos de los problemas siguientes se remite a este para encontrar un ejemplo sobre la forma de rellenar las estructuras de datos del compilador en el caso de lenguajes que permitiría usar *arrays* y objetos intercalados (*arrays*

de objetos, objetos con campos de tipo *array*, etc.); este tipo de lenguajes permite acceder en el programa que aparece a continuación (desde dentro de un método de la clase `E`) a variables como `a[5][4].a2[3][2].a1[1][0]= 19`; para este ejemplo el código generado podría ser:

```
ldarg 0
ldfld D[] E::a
...apila el desplazamiento de [5][4]
ldelem.ref
ldfld C[] D::a2
...apila el desplazamiento de [3][2]
ldelem.ref
ldfld int32[] C::a1
...apila el desplazamiento de [1][0]
ldc.i4 19
stelem.i4
```

La instrucción `ldelem.ref` carga la referencia a un objeto perteneciente a un array de objetos.

Otros problemas remiten a este para aspectos más sencillos, como, por ejemplo, el método `f` y sus declaraciones locales.

A las declaraciones de un programa fuente como el siguiente:

```
class C {

  public int i1;
  public int a1[20][30];

  ...
}

class D {

  public int i2;
  public C c2;
  public C a2[40][50];

  ...
}

class E {
```

```
  public D d;
  public D a[60][70];

  public method f () {

    int x[20][40][60];
    int y[80];

    ...
  }

  ...
}
```

le corresponderían las siguientes tablas; la instantánea pertenece al momento en el que el compilador está compilando el final del método `f` de la clase `E`:

Tabla de símbolos global, TSG

Nombre	Tipo	Posición	Visibilidad	Tipo de símbolo
C	5			nombre de clase
D	8			nombre de clase
E	11			nombre de clase

Tabla de tipos

Núm.	Tipo	Tabla de símbolos	Tamaño / Par.	Tipo base
0	entero			
...	...			
5	clase	TS1		
6	array		30	0
7	array		20	6
8	clase	TS2		
9	array		50	5
10	array		40	9
11	clase	TS3		
12	array		70	8
13	array		60	12
14	método		0	
15	array		60	0
16	array		40	15
17	array		20	16
18	array		80	0

Tabla de símbolos TS1

Nombre de la clase asociada: C				
Nombre	Tipo	Posición	Visibilidad	Tipo de símbolo
i1	0		público	campo
a1	7		público	campo

Tabla de símbolos TS2

Nombre de la clase asociada: D				
Nombre	Tipo	Posición	Visibilidad	Tipo de símbolo
i2	0		público	campo
c2	5		público	campo
a2	10		público	campo

Tabla de símbolos TS3

Nombre de la clase asociada: E				
Nombre	Tipo	Posición	Visibilidad	Tipo de símbolo
d	8		público	campo
a	13		público	campo
f	14		público	
x	17	0		local
y	18	1		local

Como ves, los *arrays* se almacenan en la tabla de tipos indicando el tamaño D de cada una de sus dimensiones; los componentes válidos en cada dimensión van desde 0 hasta $D-1$. Además, para cada clase se almacena en la tabla de tipos una referencia a la tabla de símbolos correspondiente.

❄ **62** Considera el siguiente fragmento de la especificación sintáctica de un lenguaje:

$$
\begin{array}{lcl}
I & \longrightarrow & \textbf{for (}\ E\ \textbf{;}\ E\ \textbf{;}\ E\ \textbf{)}\ \textit{Bloque} \\
E & \longrightarrow & R = E \\
E & \longrightarrow & T \\
T & \longrightarrow & T + F \\
T & \longrightarrow & F \\
F & \longrightarrow & \textbf{cteint} \\
F & \longrightarrow & \textbf{ctebool} \\
F & \longrightarrow & R \\
F & \longrightarrow & (\ E\ ) \\
R & \longrightarrow & \textbf{id}\ D \\
D & \longrightarrow & D\ [\ E\ ] \\
D & \longrightarrow & \epsilon
\end{array}
$$

Diseña un esquema de traducción dirigida por la sintaxis (ETDS) que genere el código CIL adecuado para la máquina virtual de .NET 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 simples son los enteros y los booleanos, que son totalmente incompatibles; además, solo es posible sumar valores enteros.

b. Como se observa en la gramática, la asignación es una expresión más.

c. Puedes asumir que el no terminal *Bloque* tiene un atributo sintetizado *Bloque.cod* con el codigo objeto generado para el cuerpo del bucle.

d. 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`.

e. El ETDS ha de realizar en un atributo los cálculos que determinen el tamaño máximo de la pila (necesarios para la directiva `maxstack`) para el código generado.

f. Para simplificar, considera que el lenguaje fuente exige que todas las instrucciones aparezcan dentro de los métodos (no hay, por lo tanto, un programa principal independiente de cualquier clase).

g. Asume que la tabla de símbolos actual es accesible a través de la referencia global `TSA`.

h. En el problema 61 se muestra un ejemplo de las estructuras de datos mantenidas por el compilador para un programa escrito en el lenguaje fuente de este problema. 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.

i. Para simplificar, considera que todas las variables son locales (no has de considerar, por tanto, ni parámetros ni atributos). Concéntrate, por tanto, únicamente en el método `f` del ejemplo del problema 61 y obvia el resto. Ignora también la posibilidad de declarar *arrays* de objetos.

j. 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`.

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

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

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

a. En este lenguaje el único tipo de datos simple son los enteros.

b. No es necesario que tu ETDS realice los cálculos que determinen el tamaño máximo de la pila (necesarios para la directiva `maxstack`).

c. Para simplificar, considera que el lenguaje fuente exige que todas las instrucciones aparezcan dentro de los métodos (no hay por lo tanto un programa principal independiente de cualquier clase).

d. Asume que la tabla de símbolos actual es accesible a través de la referencia global `TSA`.

e. En el problema 61 se muestra un ejemplo de las estructuras de datos mantenidas por el compilador para un programa escrito en el lenguaje fuente de este problema. 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.

f. Para simplificar, considera que todas las variables son campos (no has de considerar, por tanto, ni parámetros ni locales). Puedes obviar, por tanto, las variables locales del método `f` del ejemplo del problema 61.

g. 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.

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

❄ **65** Dada una declaración de un *array* como la siguiente:

$$\text{int a}[D_1][D_2][D_3]$$

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:

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

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*:

$$
\begin{array}{lcl}
I & \longrightarrow & \textbf{print}\ E \\
I & \longrightarrow & \textbf{read}\ R \\
R & \longrightarrow & \textbf{id}\ A \\
A & \longrightarrow & [\ E\ ]\ A \\
A & \longrightarrow & [\ E\ ] \\
E & \longrightarrow & R \\
E & \longrightarrow & \textbf{entero}
\end{array}
$$

Apartado A. Diseña un esquema de traducción dirigida por la sintaxis (ETDS) que genere el código CIL adecuado para la máquina virtual de .NET 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 CIL adecuado usando la nueva forma iterativa.

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

a. En este lenguaje el único tipo de datos simple son los enteros.

b. Las instrucciones **print** y **read** son, respectivamente, las de escritura y lectura.

c. No es necesario que tu ETDS realice los cálculos que determinen el tamaño máximo de la pila (necesarios para la directiva `maxstack`).

d. Para simplificar, considera que el lenguaje fuente exige que todas las instrucciones aparezcan dentro de los métodos (no hay por lo tanto un programa principal independiente de cualquier clase).

e. Asume que la tabla de símbolos actual es accesible a través de la referencia global `TSA`.

f. En el problema 61 se muestra un ejemplo de las estructuras de datos mantenidas por el compilador para un programa escrito en el lenguaje fuente de este problema. 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.

g. Para simplificar, considera que todas las variables son locales (no has de considerar, por tanto, ni parámetros ni atributos). Concéntrate, por tanto, únicamente en el método `f` del ejemplo del problema 61 y obvia el resto. Ignora también la posibilidad de declarar *arrays* de objetos.

h. 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). Indica, además, de qué tipo es cada uno de los atributos que utilizas y cuál es su cometido.

❄ **66** Considera el siguiente fragmento de la especificación sintáctica de un lenguaje:

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

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

a. En este lenguaje el único tipo de datos simple son los enteros.

b. Las instrucciones **print** y **read** son, respectivamente, las de escritura y lectura.

c. No es necesario que tu ETDS realice los cálculos que determinen el tamaño máximo de la pila (necesarios para la directiva `maxstack`).

d. Para simplificar, considera que el lenguaje fuente exige que todas las instrucciones aparezcan dentro de los métodos (no hay por lo tanto un programa principal independiente de cualquier clase).

e. Asume que la tabla de símbolos actual es accesible a través de la referencia global `TSA`.

f. En el problema 61 se muestra un ejemplo de las estructuras de datos mantenidas por el compilador para un programa escrito en el lenguaje fuente de este problema. 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.

g. Para simplificar, considera que todas las variables son campos (no has de considerar, por tanto, ni parámetros ni locales). Puedes obviar, por tanto, las variables locales del método `f` del ejemplo del problema 61.

h. 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). Indica, además, de qué tipo es cada uno de los atributos que utilizas y cuál es su cometido.

❄ **67** Explica con cierto detalle (indicando incluso algunas acciones del ETDS) cómo modificarías lo visto en clase para permitir 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?

❄ **68** Considera el siguiente fragmento de la especificación sintáctica de un lenguaje:

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

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

a. En este lenguaje el único tipo de datos simple son los enteros.

b. No es necesario que tu ETDS realice los cálculos que determinen el tamaño máximo de la pila (necesarios para la directiva `maxstack`).

c. Para simplificar, considera que el lenguaje fuente exige que todas las instrucciones aparezcan dentro de los métodos (no hay por lo tanto un programa principal independiente de cualquier clase).

d. Asume que la tabla de símbolos actual es accesible a través de la referencia global `TSA`.

e. En el problema 61 se muestra un ejemplo de las estructuras de datos mantenidas por el compilador para un programa escrito en el lenguaje fuente de este problema. 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.

f. Para simplificar, considera que todas las variables son campos (no has de considerar, por tanto, ni parámetros ni locales). Puedes obviar, por tanto, las variables locales del método `f` del ejemplo del problema 61.

g. 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.

h. 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.

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). Indica, además, de qué tipo es cada uno de los atributos que utilizas y cuál es su cometido.

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

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

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

Por ejemplo, a una declaración como:

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

le corresponderían 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:

a. En este lenguaje el único tipo de dato simple es el entero.

b. No es necesario que tu ETDS realice los cálculos que determinen el tamaño máximo de la pila (necesarios para la directiva `maxstack`).

c. Asume que la tabla de símbolos actual es accesible a través de la referencia global `TSA`.

d. En el problema 61 se muestra un ejemplo de las estructuras de datos mantenidas por el compilador para un programa escrito en un lenguaje fuente similar al de este problema.

e. Para simplificar, considera que todas las variables declaradas son locales (no has de considerar, por tanto, ni parámetros ni atributos). Concéntrate, por tanto, únicamente en el método `f` del ejemplo del problema 61 y obvia el resto. Ignora también la posibilidad de declarar *arrays* de objetos.

f. 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

g. 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.

h. 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.

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.

❄ **70** Considera el siguiente fragmento de la especificación sintáctica de un lenguaje:

$$
\begin{array}{lcl}
I & \longrightarrow & R = R \\
R & \longrightarrow & \textbf{id}\ B \\
B & \longrightarrow & \epsilon \\
B & \longrightarrow & A \\
A & \longrightarrow & [\ E\ ]\ A \\
A & \longrightarrow & [\ E\ ] \\
E & \longrightarrow & R \\
E & \longrightarrow & \textbf{entero}
\end{array}
$$

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

a. En este lenguaje el único tipo de datos simple son los enteros.

b. 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.

c. No es necesario que tu ETDS realice los cálculos que determinen el tamaño máximo de la pila (necesarios para la directiva `maxstack`).

d. Para simplificar, considera que el lenguaje fuente exige que todas las instrucciones aparezcan dentro de los métodos (no hay por lo tanto un programa principal independiente de cualquier clase).

e. Asume que la tabla de símbolos actual es accesible a través de la referencia global `TSA`.

f. En el problema 61 se muestra un ejemplo de las estructuras de datos mantenidas por el compilador para un programa escrito en el lenguaje fuente de este problema (salvo por lo comentado a continuación respecto a la tabla de tipos). 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.

g. Para simplificar, considera que todas las variables son locales (no has de considerar, por tanto, ni parámetros ni atributos). Concéntrate, por tanto, únicamente en el método `f` del ejemplo del problema 61 y obvia el resto. Ignora también la posibilidad de declarar *arrays* de objetos.

h. También para simplificar la solución al problema, asume que la inserción de la información en la tabla de tipos 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	Posición
a	3	0
b	5	1
c	6	2
d	3	3
e	1	4
f	9	5

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`.

i. 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.

j. 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). Indica, además, de qué tipo es cada uno de los atributos que utilizas y cuál es su cometido.

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

S	$\longrightarrow$	*I*
I	$\longrightarrow$	**for** *id* **:=** *E* **to** *E* **step** *Signo* **entero** *Bloque* **next** *id* **;**
I	$\longrightarrow$	**break if** *E* **;**
I	$\longrightarrow$	**continue if** *E* **;**
I	$\longrightarrow$	**print** *E*
Bloque	$\longrightarrow$	**{** *Sec* **}**
Sec	$\longrightarrow$	*Sec I*
Sec	$\longrightarrow$	*I*
Signo	$\longrightarrow$	**opsum**
Signo	$\longrightarrow$	ϵ

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

a. En este lenguaje el único tipo de datos simple son los enteros.

b. 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.

c. El código generado para el bucle `for` ha de seguir la estructura de la figura 1.

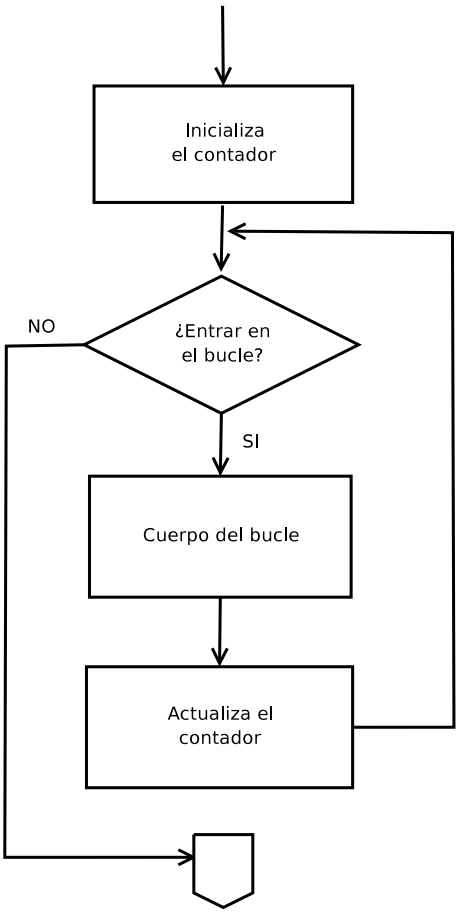

Figura 1: Diagrama de flujo para la instrucción `for`.

d. 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 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. El ETDS ha de usar algún atributo para realizar los cálculos que determinen el tamaño máximo de la pila (necesarios para la directiva `maxstack`) para el código generado.

g. 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 generado y el espacio de pila necesario para 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).

h. 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.

i. 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.

j. Para simplificar, considera que todas las variables son locales (no has de considerar, por tanto, ni parámetros ni atributos).

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.

❄ **72** Considera el siguiente fragmento de la especificación sintáctica de un lenguaje:

$$\begin{array}{lcl} I & \longrightarrow & \textbf{id}\ R = \textbf{id}\ R \\ R & \longrightarrow & R\ [\ \textbf{entero}\ ] \\ R & \longrightarrow & \epsilon \end{array}$$

Diseña un esquema de traducción dirigida por la sintaxis (ETDS) que genere el código CIL adecuado para la máquina virtual de .NET 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. 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 que tu ETDS realice los cálculos que determinen el tamaño máximo de la pila (necesarios para la directiva `maxstack`).

e. Para simplificar, considera que el lenguaje fuente exige que todas las instrucciones aparezcan dentro de los métodos (no hay por lo tanto un programa principal independiente de cualquier clase).

f. Asume que la tabla de símbolos actual es accesible a través de la referencia global `TSA`.

g. En el problema 61 se muestra un ejemplo de las estructuras de datos mantenidas por el compilador para un programa escrito en el lenguaje fuente de este problema. 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.

h. Para simplificar, considera que todas las variables son locales (no has de considerar, por tanto, ni parámetros ni atributos). Concéntrate, por tanto, únicamente en el método `f` del ejemplo del problema 61 y obvia el resto. Ignora también la posibilidad de declarar *arrays* de objetos.

i. Es necesario indicar tantos corchetes como dimensiones tenga el *array*; ni más ni menos.

j. 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). 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:

$$
\begin{array}{lcl}
I & \longrightarrow & \textbf{print}\ E \\
E & \longrightarrow & E\ \textbf{opsum}\ T \\
E & \longrightarrow & T \\
T & \longrightarrow & \textbf{id} \\
T & \longrightarrow & \textbf{id opinc} \\
T & \longrightarrow & \textbf{opinc id}
\end{array}
$$

Diseña un esquema de traducción dirigida por la sintaxis (ETDS) que genere el código CIL adecuado para la máquina virtual de la plataforma .NET 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.

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. El ETDS ha de realizar los cálculos que determinen el tamaño máximo de la pila (necesarios para la directiva `maxstack`).

e. Para simplificar, considera que el lenguaje fuente exige que todas las instrucciones aparezcan dentro de los métodos (no hay por lo tanto un programa principal independiente de cualquier clase).

f. Asume que la tabla de símbolos actual es accesible a través de la referencia global `TSA`.

g. Para simplificar, considera que todas las variables son locales (no has de considerar, por tanto, ni parámetros ni atributos).

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.

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

$$
\begin{array}{lcl}
E & \longrightarrow & E \,\textbf{*}\, T \\
E & \longrightarrow & T \\
T & \longrightarrow & |\, F \,| \\
T & \longrightarrow & F \\
F & \longrightarrow & \textbf{id} \\
F & \longrightarrow & \textbf{entero} \\
F & \longrightarrow & \textbf{real} \\
F & \longrightarrow & (\, E \,)
\end{array}
$$

Diseña un esquema de traducción dirigida por la sintaxis (ETDS) que genere el código CIL adecuado para la máquina virtual de .NET 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.

b. Los identificadores pueden corresponder a variables de tipo simple o de tipo *array* (de una o más dimensiones).

c. Esta parte de la gramática caracteriza los operadores de producto (*****) y de *norma* ($|\cdot|$). 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* `a` y `b` toman los valores `a=[3,4]` y `b=[2,3]`, el resultado de `|a|*(a*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 `CIL`, puedes hacerlo, pero has de definir claramente el comportamiento de la instrucción.

j. El ETDS ha de realizar los cálculos que determinen el tamaño máximo de la pila (necesarios para la directiva `maxstack`).

k. Para simplificar, considera que el lenguaje fuente exige que todas las instrucciones aparezcan dentro de los métodos (no hay por lo tanto un programa principal independiente de cualquier clase).

l. Asume que la tabla de símbolos actual es accesible a través de la referencia global `TSA`.

m. En el problema 61 se muestra un ejemplo de las estructuras de datos mantenidas por el compilador para un programa escrito en el lenguaje fuente de este problema. 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.

n. Para simplificar, considera que todas las variables son locales (no has de considerar, por tanto, ni parámetros ni atributos). Concéntrate, por tanto, únicamente en el método `f` del ejemplo del problema 61 y obvia el resto. Ignora también la posibilidad de declarar *arrays* de objetos.

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.

Subprogramas

❄ **75** Indica el código objeto CIL que se generaría para el siguiente programa fuente.

Comenta brevemente cada una de las instrucciones objeto.

```
class C {

  public real a;

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

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

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

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

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

a. En este lenguaje el único tipo de datos simple son los enteros.

b. Como puedes ver en la gramática, el lenguaje permite usar *arrays* y efectuar llamadas a métodos.

c. 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 método y no es necesario introducir ninguna información adicional sobre los métodos en la tabla de tipos.

d. No es necesario que tu ETDS realice los cálculos que determinen el tamaño máximo de la pila (necesarios para la directiva `maxstack`).

e. Para simplificar, considera que el lenguaje fuente exige que todas las instrucciones aparezcan dentro de los métodos (no hay por lo tanto un programa principal independiente de cualquier clase).

f. Asume que la tabla de símbolos actual es accesible a través de la referencia global `TSA`.

g. En el problema 61 se muestra un ejemplo de las estructuras de datos mantenidas por el compilador para un programa escrito en el lenguaje fuente de este problema. 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.

h. Para simplificar, considera que todas las variables son locales (no has de considerar, por tanto, ni parámetros ni atributos). Concéntrate, por tanto, únicamente en el método `f` del ejemplo del problema 61 y obvia el resto. Ignora también la posibilidad de declarar *arrays* de objetos.

i. Observa que la gramática solo contempla el caso de invocaciones de métodos de instancia dentro de la propia clase.

j. 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). 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** Indica el código objeto CIL que se generaría para el siguiente fragmento de programa fuente.

Comenta brevemente cada una de las instrucciones objeto.

```
class C {
  public int a;
```

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

❄ **78** Para permitir que se invoque un método que aún no se ha definido en el programa fuente sin que el compilador tenga que realizar más de una pasada, se puede hacer que el lenguaje contemple la declaración de prototipos (o signaturas) de métodos. Explica cómo implementarías un sistema de este tipo, inspirándote para ello en un lenguaje fuente como el usado en la asignatura, que no permite dichos prototipos. Céntrate en el caso de los métodos de dentro de una misma clase.

❄ **79** Indica el código objeto CIL que se generaría para el siguiente fragmento de programa fuente.

Comenta brevemente cada una de las instrucciones objeto.

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

❄ **80** Explica con cierto detalle (indicando incluso algunas acciones del ETDS) cómo modificarías lo visto en clase sobre la generación de código CIL para considerar la posibilidad de declarar dentro de una clase métodos *sobrecargados* 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)`?

❄ **81** Indica el código objeto CIL que se generaría para el siguiente fragmento de programa fuente.

Comenta brevemente cada una de las instrucciones objeto.

```
class C {
  public method int f (int a) {
    return 2*a;
  }
}
program {
  C x;
  print(x.f(2));
}
```

Más problemas

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