Loro - Un Sistema de Programación Didáctico
http://loro.sf.net

Especificación del lenguaje

Versión 0.9.2 – noviembre 25 de 2002
Carlos A. Rueda

1 Introducción

Loro es un lenguaje de programación de carácter didáctico con revisión estricta de tipos y reciclaje automático de memoria. Su soporte central es el diseño por contrato aplicado a la programación tanto imperativa como orientada a objetos. Este documento ofrece una definición descriptiva del lenguaje. Se asumen conocimientos previos de algún lenguaje de programación como Java o C++. Aunque no se utiliza un estilo estrictamente formal, tampoco se trata de un documento dirigido al lector principiante en programación.

Siendo parte integral de un proyecto abierto y en desarrollo, esta especificación no debe tomarse como definitiva. De hecho, el lenguaje está abierto a los ajustes que sean pertinentes para atender, hasta donde sea posible, las diferentes propuestas y revisiones que vayan surgiendo. Para obtener información actualizada al respecto, favor visitar la página del proyecto.

El apéndice (7.13) explica la notación utilizada e incluye la gramática concreta del lenguaje.

1.1 Aspectos léxicos

En general, el conjunto de caracteres de Loro es el conjunto Unicode. Las letras mayúsculas y minúsculas se toman como distintas en el lenguaje.

1.1.1 Espacios en blanco

El espacio simple, tabulador, y el cambio de línea corresponden a los *espacios en blanco* (*espacios*, para abreviar) cuya función es la de decidir la separación de los componentes léxicos (1.1.3) del lenguaje. Por lo menos un espacio es necesario para separar dos palabras clave o identificadores adyacentes, o entre una palabra clave o identificador y un número, pero no se requiere entre un número y una palabra clave dado que esto no representa ninguna ambigüedad. Los delimitadores y operadores no requieren espacios para separarlos de sus elementos adyacentes a ambos lados. Excepto en literales cadena, ningún componente léxico contiene espacios en blanco.

1.1.2 Comentarios

Un comentario es cualquier texto (secuencia de caracteres) delimitado según dos posibilidades:

- Por la doble diagonal `//` que se encuentre más a la izquierda en una línea y el final de esa línea;
- Por los caracteres `/*` y los caracteres `*/` aunque se abarquen varias líneas. No se permite el anidamiento.

Los comentarios pueden aparecer en cualquier punto en donde pueda venir un componente léxico y no requieren espacios en blanco a ningún lado para delimitarse. Estos comentarios son ignorados completamente durante compilación.

1.1.3 Componentes léxicos

Los componentes léxicos son las palabras clave, constantes literales, identificadores, operadores y delimitadores.

1.1.3.1 Palabras clave

Las palabras claves del lenguaje se muestran a continuación (nótese que las versiones sin tilde también están reservadas):

`booleano`	`cadena`	`caracter`	`entero`
`real`	`algoritmo`	`bajando`	`caso`
`ciclo`	`cierto`	`clase`	`constante`
`constructor`	`continúe`	`crear`	`desde`
`descripción`	`en (*)`	`entonces`	`entrada`
`es_instancia_de`	`especificación`	`éste (*)`	`estrategia`
`existe`	`extiende`	`falso`	`fin`
`global (*)`	`haga`	`hasta`	`implementación`
`inicio`	`implementa (*)`	`interface (*)`	`mientras`
`nada`	`nulo`	`paquete`	`para`
`para_todo`	`paso`	`pos`	`pre`
`repita`	`retorne`	`salida`	`según`
`si`	`si_no`	`si_no_si`	`súper (*)`
`termine`	`utiliza`	`como`	`operación (*)`
`método (*)`	`objeto`	`implementa`	`lance`
`intente`	`atrape`	`siempre`	

(*) No usados aún.

1.1.3.2 Constantes literales

Las constantes literales pueden ser números enteros, números reales, constantes booleanas, caracteres, cadenas y el literal `nulo`.

- Un *entero literal* contiene sólo dígitos y su valor debe estar entre 0 y 2^{31}-1.
- Un *real literal* es una secuencia de dígitos y, opcionalmente, un exponente; si no hay exponente, es obligatorio que aparezca un punto decimal dentro de la secuencia de dígitos inicial (de tal manera que separe las partes entera y decimal del número). El exponente se indica con la letra “e” o “E” y una secuencia de dígitos posiblemente prefijada con un signo más o un signo menos. El valor real será representado según el formato IEEE 754 de 64 bits. Nótese que no hay literales numéricos negativos: los valores negativos se manejan en un nivel superior al léxico.
- Sólo hay dos *booleanos literales* indicados por las palabras reservadas **`cierto`** y **`falso`**.
- Un *caracter literal* es un caracter Unicode encerrado entre comillas simples (`'`). El caracter puede estar escrito explícitamente o representado por una secuencia de escape (ver más abajo).

- Una *cadena literal* es una secuencia de caracteres Unicode encerrada entre comillas dobles (″). Cada caracter puede estar escrito explícitamente o representado por una secuencia de escape (ver más abajo). Puede incluir explícitamente los caracteres de cambio de línea y de tabulación. No hay una longitud límite preestablecida para las cadenas literales. Una cadena literal puede aparecer en donde se espera cualquier expresión, siendo su tipo cadena. Para hacer la diferenciación necesaria en otros usos para cadenas literales, se tienen las siguientes variantes:
    - Un *texto de documentación* es similar a una cadena literal excepto que se encierra en pares de comillas sencillas (`''`). Como su nombre lo indica, estas cadenas son utilizadas para documentar diversos elementos en un programa.
    - Un *texto de implementación* es similar a una cadena literal excepto que se encierra entre los símbolos `{%` y `%}`. Estas cadenas se utilizan para indicar lenguajes diferentes a Loro en la codificación de algoritmos. (8)
- El *literal nulo* es la palabra reservada **`nulo`**. Denota la referencia nula aplicable para arreglos, objetos y algoritmos.

Secuencias de escape
Las siguientes secuencias de escape permiten indicar caracteres especiales dentro de los literales caracter y cadena:

Secuencia	*Descripción*
`\n`	cambio de línea
`\r`	retroceso a inicio de línea
`\t`	Tabulador
`\b`	Retroceso
`\f`	cambio de página
`\\`	diagonal invertida
`\'`	comilla sencilla
`\″`	comilla doble
`\`*`ddd`*	caracter octal (cada *d* es un dígito octal)
`\u`*`dddd`*	caracter Unicode (cada *d* es un dígito hexadecimal)

1.1.3.3 Identificadores

Un identificador es una secuencia de letras y posiblemente dígitos comenzando con una letra, excluyendo las palabras clave (1.1.3.1). Por letra se entiende cualquiera de los caracteres Unicode correspondientes a letra, y también los caracteres `$` y `_` (“subguión”). No hay una longitud límite preestablecida para los identificadores. Un identificador también puede finalizarse con una o más comillas simples, en cuyo caso se interpreta como una variable semántica (3.2.1).

1.1.3.4 Operadores

Los operadores denotados con símbolos (no palabras clave) son los siguientes:

```
:=          ?:          <=>          =>
```

```
||          &&          |           ^
&           =           !=          <
>           <=          >=          <<
>>          >>>         +           -
*           /           %           !
~           #           @
```

1.1.3.5 Delimitadores

Los siguientes son componentes léxicos (no palabras clave) utilizados como delimitadores.

```
:           ;           ,           .
(           )           [           ]
{           }           ->
```

2 Fuentes

Un *documento fuente* (*fuente*, para abreviar) se refiere al conjunto de elementos (especificaciones, algoritmos, clases) (5) que pueden aparecer dentro de un programa fuente sometido a compilación. En este documento se utiliza el término *unidad de compilación* para cada uno de tales elementos.[1] Un documento fuente tiene el siguiente esquema:

```
[ paquete ]
( utiliza )*
( especificación | algoritmo | clase )+
```

Las unidades quedarán asociadas al paquete indicado si aparece, o bien al paquete *anónimo* si no se indica explícitamente un paquete (2.1). La sintaxis para indicar el paquete correspondiente es:

```
paquete id ( :: id )* ;
```

Todas las unidades del mismo paquete tienen accesibilidad inmediata entre sí. Unidades en otros paquetes deben referenciarse incluyendo el nombre de sus paquetes correspondientes. Los elementos `utiliza` dan una facilidad para abreviar la referencia a unidades ubicadas en otros paquetes. La sintaxis general es:

```
utiliza ( especificación | algoritmo | clase ) id ( :: id )* ;
```

2.1 Paquetes

En Loro, como en otros lenguajes, se utiliza un mecanismo de paquetes para establecer *espacios de nombres* para las unidades compiladas. En este sentido, cada unidad "reside" lógicamente en un cierto paquete. Salvo por el paquete *anónimo* (que no tiene

[1] Es pertinente aclarar que algunos otros lenguajes se utiliza el término *unidad de compilación* a lo que aquí se denomina *documento fuente*.

un nombre explícito), todo paquete se identifica con un nombre particular. La sintaxis para un nombre de paquete es:

id (:: *id*)*

es decir, una lista de identificadores simples separados por "::". El nombre completo de una cierta unidad se establece indicando el nombre del paquete en que reside, el separador ("::") y el nombre de la unidad como tal. Cada tipo de unidad establece un espacio de nombres diferente. Por esta razón, los elementos *utiliza*, dentro de un documento fuente, incluyen la indicación del tipo de unidad correspondiente.

3 Declaraciones

Todos los identificadores que aparezcan en una unidad de compilación (5) tienen que introducirse mediante *declaraciones* correspondientes. Una declaración puede ser *explícita* o *implícita*.

3.1 Declaraciones explícitas

En esta categoría están las *definiciones de unidades*, las *declaraciones de variables* y las *etiquetas*.

3.1.1 Definición de unidades

La definición de una unidad (5) establece una declaración explícita para el nombre correspondiente. En el caso de clases y algoritmos, la vigencia de este nombre comienza justo a continuación del encabezado principal correspondiente con el fin de permitir la recursión. Esto no aplica para las especificaciones.

Para permitir la recursión mutua (y también por conveniencia), es posible utilizar un identificador para referir a una unidad (es decir, especificación, algoritmo o clase) que se define más adelante dentro de un mismo fuente. (Una implementación podría extender esta facilidad para abarcar múltiples fuentes.)

3.1.2 Declaración de variables

Una variable siempre tiene que declararse explícitamente y antes de cualquier uso. Toda declaración de variable involucra una asociación de un *tipo* de dato (4) para el identificador de la variable, y también una descripción verbal. Las declaraciones de variables siempre tienen una vigencia local dentro del *ámbito* en que se hace la declaración. Esta vigencia comienza justo a continuación de la declaración y va hasta el final del ámbito en que se hace la declaración. El ámbito está determinado por cada construcción que permita declaraciones de variables como se explica en otras partes de este documento. Dentro de este ámbito no puede declararse otra variable con el mismo nombre.

3.1.3 Etiquetas

Una *etiqueta* es un identificador para una acción de iteración (6.3). Se declaran a través de la sintaxis correspondiente a cada caso de iteración con la forma básica:

/ *id* **/**

Las acciones **`termine`** (6.5) y **`continúe`** (6.6), que operan sobre iteraciones, pueden cualificarse agregando el identificador de la iteración sobre la que deben actuar. Por defecto, estas acciones actúan sobre la iteración más inmediata. A través del mecanismo de las etiquetas se hace posible que actúen sobre una iteración no inmediata en un anidamiento de iteraciones.

No se permite que dos iteraciones anidadas (una dentro del cuerpo de acciones de la otra) tengan la misma etiqueta.

3.2 Declaraciones implícitas

Ciertos identificadores son introducidos a través de declaraciones implícitas. En este caso se encuentran las *variables semánticas* y ciertos atributos para arreglos y cadenas.

3.2.1 Variables semánticas

Una *variable semántica* (o *lógica*) representa un cierto valor fijo en un momento dado dentro de la ejecución de un programa.

Este mecanismo, que permite *capturar* el valor que tiene una expresión en un momento dado, puede utilizarse para relacionar valores *antes* y *después* de un cierto fragmento de código, lo que es particularmente útil en la escritura de pre y poscondiciones.

Este valor fijo puede corresponder también a una función matemática que, o bien no se quiere expresar, o bien no se puede expresar utilizando la semántica del mismo lenguaje. Por ejemplo, la función matemática que toma un argumento entero positivo *n* y produce el *n*-ésimo número primo podría representarse con una variable semántica.

Una variable semántica sólo puede aparecer dentro del contexto de las afirmaciones (6.7) y no requiere declaración explícita previa como las otras variables regulares del lenguaje. Para dar soporte adecuado a esto, la primera aparición de una variable lógica debe ser en una expresión de *igualdad*: `expresión = x'`. Para efectos de compilación, el tipo de `x'` se deduce de manera automática puesto que toma exactamente el tipo de la expresión. En ejecución, el valor de `x'` será el que resulte de la evaluación de la expresión. En consecuencia, la expresión no puede contener variables semánticas que no hayan sido definidas previamente (en particular, si tanto `x'` como `y'` son variables sin definir previamente, entonces la expresión `x' = y'` será inválida).

Nótese que la expresión booleana `exp1 = x'`, para *introducir* la variable semántica `x'`, captura la noción de que existe un valor `x'` tal que la expresión dada `exp1` es igual a `x'`; por lo tanto, siempre se evaluará con resultado cierto. Complementariamente, si `x'` ya se encuentra definida previamente, entonces `exp2 = x'` podrá resultar en cierto o en falso dependiendo de si los valores correspondientes son iguales en el momento de la evaluación.

3.2.2 Atributos automáticos

En el caso de arreglos y cadenas, los siguientes identificadores se declaran implícitamente para denotar atributos:

Identificador	*Tipo*	*Descripción*
`inf`	`entero`	Límite inferior en rango de indización
`sup`	`entero`	Límite superior en rango de indización
`longitud`	`entero`	Tamaño total del arreglo o cadena

Si `x` es un arreglo o cadena, siempre se cumple que `x.longitud = x.sup - x.inf + 1`. Ver 7.6 y 7.10.

4 Tipos

4.1 Tipos en Loro

Loro ofrece los siguientes tipos de datos:

- Cuatro tipos primitivos de datos atómicos:
  - enteros
  - reales
  - booleanos
  - caracteres
- El tipo arreglo
- Un tipo especial de arreglo para cadenas de caracteres
- El tipo algoritmo
- El tipo clase

4.1.1 Tipo entero

El tipo entero es el conjunto de enteros representados en complemento a dos con precisión de 32 bits. La palabra reservada **`entero`** denota este tipo.

4.1.2 Tipo real

El tipo real denota números reales en representación de punto flotante con precisión según el formato IEEE 754 de 64 bits. La palabra reservada **`real`** denota este tipo.

4.1.3 Tipo booleano

El tipo booleano sólo admite uno de dos valores posibles: `cierto` o `falso`. La palabra reservada **`booleano`** denota este tipo.

4.1.4 Tipo caracter

El tipo caracter es el conjunto de los caracteres Unicode. Estos códigos tienen un orden establecido por lo que todos los operadores de comparación pueden ser utilizados y algunos aritméticos. La palabra reservada **`caracter`** denota este tipo.

4.1.5 Tipo arreglo

Un arreglo es una secuencia ordenada de elementos o datos del mismo tipo. La sintaxis `[]`***t*** , donde ***t*** es un tipo, denota el tipo “arreglo de elementos del tipo ***t***”. Los elementos del arreglo son accedidos mediante indización con expresiones de tipo entero. Se produce un error de ejecución si se utiliza un índice que no está en el rango válido de indización para el arreglo. El tamaño del arreglo y el rango de indización se establecen en el momento de la creación del arreglo (7.6).

4.1.6 Tipo cadena
Una cadena es una secuencia ordenada de caracteres (como un arreglo) con un tratamiento especial:

- Las cadenas pueden concatenarse.
- Las cadenas son inmutables (no pueden modificarse una vez creadas).
- El operador de igualdad verifica equivalencia del contenido de los argumentos (en el caso de los arreglos sólo se verifica igualdad de referencias).
- Toda expresión en Loro tiene una versión tipo cadena (7.12).

La palabra reservada **`cadena`** denota este tipo.

4.1.7 Tipo clase
Una clase establece una estructura posiblemente heterogénea de datos con respecto a la cual pueden crearse diferentes instancias o particularizaciones (objetos) correspondientes. El tipo denota entonces al conjunto de todas las instancias de la clase. En 5.3 se explica la forma para definir clases.

4.1.8 Tipo algoritmo
Una especificación (5.1) define el tipo correspondiente al conjunto de algoritmos que satisfacen dicha especificación. La forma en que se denota este tipo es:

`algoritmo para` *`nombreEspecificación`*

Es posible no referir a ninguna especificación en particular. En este caso, denominado *algoritmo genérico*, el tipo se denota simplemente:

`algoritmo`

que viene a denotar el conjunto de todos los algoritmos (sin importar qué especificaciones satisfacen).

4.2 Compatibilidad de tipo
En Loro, cada expresión tiene un único tipo y debe haber compatibilidad de tipo en las asignaciones e invocaciones. La igualdad de tipos siempre implica compatibilidad. Para otros casos se tienen las siguientes reglas:

(i) Hay compatibilidad cuando se utiliza un entero en donde se espera un real;
(ii) Hay compatibilidad cuando se utiliza un objeto de la clase X en donde se espera un objeto de una superclase de X;
(iii) Hay compatibilidad cuando se utiliza el literal nulo en donde se espera un arreglo, una cadena o un objeto de cualquier clase.
(iv) Hay compatibilidad cuando se utiliza un algoritmo para cualquier especificación en donde se espera un algoritmo genérico.

5 Unidades
Una *unidad* en Loro es cada elemento del más alto nivel que puede compilarse integralmente. Se tienen tres posibles unidades: *especificación*, *algoritmo*, y *clase*.

Las dos primeras, *especificación* y *algoritmo*, dan soporte al concepto de *proceso*, y la tercera, *clase*, sirve de base para la orientación a objetos. El concepto de proceso permite enfatizar la solución de problemas como una relación entrada/salida de datos,

bajo condicionamientos de responsabilidad tanto para quien usa externamente el proceso, como para quien lo implementa internamente.

5.1 Especificación

La especificación de un proceso tiene como fin establecer de manera precisa sus entradas y salidas. Esto significa incluir información sobre tipos, descripciones verbales, así como sus pre y poscondiciones. Su sintaxis general es:

```
especificación nombreEspecificación ( declaraciones )
               [ -> declaración ]
  descripción literalCadena
  entrada descripciones
  salida descripciones
  pre { afirmación }
  pos { afirmación }
fin especificación
```

El prototipo o encabezado de la especificación muestra el esquema del proceso comenzando con el nombre dado para el mismo. A continuación, entre paréntesis, vienen las declaraciones para las entradas. Todas las entradas son manejadas como *parámetros por valor*, así que ningún parámetro real podrá ser modificado al interior de una implementación para el proceso (algoritmo). Nótese sin embargo que los arreglos y objetos son en realidad *referencias* a los correspondientes contenidos, por lo que dichos contenidos sí podrían ser modificados. En estos casos, la palabra clave **`constante`**, a continuación del tipo en una declaración de un parámetro de entrada, impide que dicho parámetro aparezca recibiendo una asignación hacia alguno de sus elementos.

Solamente si el proceso produce un resultado, se escribe una flecha `->` y se hace las declaración correspondiente.

En las declaraciones se establecen los tipos para las entradas y/o salidas. La vigencia de estas declaraciones va hasta el final de la especificación.

A continuación, vienen las descripciones, tanto para la especificación en general, como para cada entrada y/o salida. Para terminar, se escriben una precondición (en caso de haber entradas), y una poscondición (en caso de haber salida). Cada condición es una afirmación, esto es, o bien una expresión de tipo booleano, o bien un literal cadena. En caso de una expresión, ésta puede contener variables semánticas (3.2.1).

Una especificación sólo se reconoce por su nombre completo (con paquete).

5.2 Algoritmo

Un algoritmo es una cierta codificación de acciones que soluciona una especificación dada. La definición de un algoritmo se hace utilizando la siguiente sintaxis general:

```
algoritmo nombreAlgoritmo para prototipoEspecificación
  descripción literalCadena
inicio
  acciones
```

```
fin algoritmo
```

Un algoritmo tiene un cierto nombre y debe indicar el prototipo de la especificación que soluciona. El prototipo se refiere a todo el encabezado de la especificación correspondiente, esto es: el nombre y las declaraciones completas de las entradas y las salidas. La vigencia de estas declaraciones va hasta el final del algoritmo.

A continuación viene una descripción para el algoritmo mediante una literal cadena. Para terminar, vienen las acciones (6) que deben realizarse para llevar a cabo el algoritmo.

Una variante de la construcción algoritmo permite hace su codificación en un lenguaje distinto a Loro. Ver ¿?

5.3 Clase

Una clase establece una estructura posiblemente heterogénea de datos con respecto a la cual pueden crearse diferentes instancias o particularizaciones (objetos) de dicha clase. La definición de una clase se hace mediante la siguiente sintaxis general:

```
clase nombreClase
  descripción literalCadena
  declaraciones
  constructores
fin clase
```

A continuación de un nombre para identificar la clase y de una descripción del propósito de la clase, vienen las declaraciones que establecen la composición de su estructura, y los posibles constructores.

5.3.1 Atributos

Cada declaración involucra el nombre para el atributo (*id*), el tipo (*tipo*) y una cadena literal de descripción (*doc*). Puede tener una de las siguientes formas:

id : *tipo* : *doc*
: Forma básica indicando tipo y documentación. No hay inicialización explícita para el atributo.

id : *tipo* := *expresión* : *doc*
: Declaración con inicialización dada.

id : *tipo* **constante** = *expresión* : *doc*
: Declaración que establece carácter de constante al atributo, por lo que debe darse una expresión para asignar un valor.

La vigencia de cada declaración comienza justo a continuación de la declaración y va hasta el final de la clase.

5.3.2 Constructores
Un constructor permite la creación e inicialización de una instancia para la clase. Su sintaxis general es:

```
constructor ( declaraciones )
  descripción literalCadena
  entrada descripciones
  pre { afirmación }
  pos { afirmación }
  estrategia literalCadena
inicio
  acciones
fin constructor
```

Se pueden tener tantos constructores como se requieran. Ya que no hay nombres particulares para los constructores de una clase, estos se diferencian por los tipos dados en las declaraciones de sus parámetros. La vigencia de estas declaraciones va hasta el final del constructor.

Basta con entrar a la ejecución de las acciones dentro del cuerpo del constructor para que la instancia esté creada e inicializada. Esta inicialización corresponde a las posibles asignaciones de valores iniciales que pueden acompañar la declaración de los atributos (5.3.1), o bien, los valores por defecto según el tipo. Los valores por defecto se muestran en la siguiente tabla.

Tipo atributo	*Valor por defecto*
`entero`	`0`
`real`	`0.0`
`booleano`	`falso`
`caracter`	`'\000'`
`cadena`	`nulo`
algoritmo	`nulo`
arreglo	`nulo`
objeto	`nulo`

Si no se indica ningún constructor en la definición de una clase, se provee uno automáticamente. Este constructor no recibe ninguna entrada y opera dejando inicializado el nuevo objeto según como se ha explicado.

6 Acciones

Por *acción* se entiende básicamente una instrucción ejecutable. Una acción puede ser simple (**termine**, **retorne**, por ejemplo), o puede ser compuesta en el sentido de que incluye a su vez otras acciones, lo que se denota dentro de su sintaxis como una o más apariciones de *`acciones`* (por ejemplo, la acción **si**).

Una acción también puede ser una declaración de variable (3.1.2), cuya vigencia queda limitada al segmento de *`acciones`* en que aparezca. En otras palabras, si un segmento de *`acciones`* incluye una declaración de una variable `x`, entonces la vigencia de `x`

comenzará justo a continuación de su declaración hasta el final del correspondiente segmento.

6.1 Decisión

```
si expresión entonces acciones
( si_no_si expresión entonces acciones )*
[ si_no acciones ]
fin si
```

El tipo de cada expresión debe ser booleano. La primera expresión que resulte cierta hace que se ejecutan las acciones a continuación del respectivo **entonces**. Si ninguna de las expresiones resulta cierta, se ejecutan las acciones a continuación del **si_no**, en caso que aparezca.

6.2 Decisión múltiple

```
según expresión haga
  ( caso expresión : acciones [fin caso] )*
  [ si_no : acciones [fin caso] ]
fin según
```

El tipo de la expresión principal debe ser numérico enumerativo (entero, caracter). Si alguna expresión de los casos indicados es igual a la expresión principal, se ejecutan las acciones dadas a continuación de los dos puntos hasta que se encuentre un **fin caso** o el **fin según**. Las expresiones para los casos deben ser constantes y diferentes entre sí.

6.3 Iteraciones

Se tienen las construcciones de iteración *ciclo*, *mientras*, *para* y *repita*.

```
ciclo [ /id/ ]
   acciones
fin ciclo
```

El cuerpo de acciones se ejecutará repetidamente. La única manera de terminar el ciclo es con una acción **termine** dentro de las acciones. El identificador, si aparece, sirvirá de etiqueta para esta iteración. (Ver acciones **termine** y **continúe**.)

```
mientras [ /id/ ] expresión haga
   acciones
fin mientras
```

El cuerpo de acciones se ejecutará repetidamente en tanto que la expresión, que debe ser de tipo booleano, sea cierta. Si la expresión es falsa al llegar a esta iteración, las acciones no se ejecutan en absoluto. Una acción **termine** puede terminar esta iteración prematuramente. El identificador, si aparece, sirvirá de etiqueta para esta iteración. (Ver acciones **termine** y **continúe**.)

Semánticamente la siguiente es una forma equivalente para la iteración *mientras*:

```
      ciclo [ /id/ ]
         termine si ! expresión ;
         acciones
      fin ciclo
```

```
para [ /id/ ] id := expresión [bajando] [paso expresión]
              hasta expresión haga
   acciones
fin para
```

Los tipos de las expresiones deben ser aritméticos y compatibles con el tipo declarado para el identificador. Esta construcción asignará en cada ciclo un valor de incremento (o de decremento si aparece **bajando**) a la variable, comenzando con el valor de la primera expresión. Este incremento es igual al valor de la expresión dada en la parte **paso**, y será unitario por defecto. Se ejecutarán las acciones dadas hasta que el valor de la variable esté por encima (o por debajo si aparece **bajando**) del valor dado por la última expresión. Es posible hacer la declaración del identificador dentro de esta misma construcción:

```
para id: tipo := expresión ...
```

en cuyo caso, el alcance (vigencia) del identificador se restringe solamente al cuerpo de la iteración. Una acción **termine** puede terminar esta iteración prematuramente. El identificador, si aparece, servirá de etiqueta para esta iteración. (Ver acciones **termine** y **continúe**.)

```
repita [ /id/ ]
   acciones
hasta expresión
```

Las acciones se ejecutan siempre una primera vez y se seguirán ejecutando en tanto no se haga cierta la expresión, que debe ser de tipo booleano. Una acción **termine** puede terminar esta iteración prematuramente. El identificador, si aparece, servirá de etiqueta para esta iteración. (Ver acciones **termine** y **continúe**.)

Semánticamente la siguiente es una forma equivalente para la iteración *repita*:

```
ciclo [ /id/ ]
   acciones
   termine si expresión ;
fin ciclo
```

6.4 Retorno de valor

```
retorne [expresión]
```

Esta acción permite retornar un valor de salida en un algoritmo y al mismo tiempo completar su ejecución. El tipo de la expresión debe ser compatible con el tipo especificado para la salida del proceso respectivo.

6.5 Terminación directa de una iteración

termine [/id/] [**si** *expresión*]

Hace que el flujo de ejecución pase a la acción inmediatamente después del delimitador de la iteración correspondiente. Esta iteración es la referida por el identificador dado o es el más inmediato por defecto. En caso que aparezca la expresión, esta terminación esta condicionada a que el valor de esta expresión sea cierta.

6.6 Continuación en una iteración

continúe [/id/] [**si** *expresión*]

Hace que el flujo de ejecución pase inmediatamente a la primera acción en el cuerpo de la iteración correspondiente. Esta iteración es la referida por el identificador dado o es el más inmediato por defecto. En caso que aparezca la expresión, esta continuación esta condicionada a que el valor de esta expresión sea cierta. En el caso de una iteración **para**, siempre se efectúa el incremento o decremento de la variable.

6.7 Lanzamiento de una excepción

lance *expresión*

Genera una excepción. Esto rompe el flujo normal de ejecución provocando un des-anidamiento de secciones **intente** y/o marcos de la pila de ejecución hasta que se alcance un acción **atrape** compatible con el tipo del valor lanzado. Una excepción no atrapada en este proceso termina la ejecución correspondiente.

6.8 Intento de acciones

```
intente acciones
( atrape ( declaración ) acciones )*
[ siempre acciones ]
fin intente
```

Esta construcción permite atrapar una excepción no atrapada en la ejecución de alguna de las acciones a continuación de la palabra **intente**. El tipo del valor generado en la excepción es confrontado con los tipos de las declaraciones en los elementos **atrape**. La primera declaración que resulte compatible en tipo determina el segmento de acciones a ejecutar como parte de la atención a la excepción generada. El identificador de la declaración tiene asignado el valor lanzado. Pueden indicarse 0 ó más secciones **atrape**. La sección **siempre**, que es opcional, indica el segmento de acciones a ejecutar independientemente de si se presenta una excepción o no. Si dentro de la ejecución de la sección **siempre** se genera una excepción no atrapada, ésta se convierte en la excepción que continuará propagándose.

Por lo menos debe indicarse una sección **atrape** o una sección **siempre** en un intento de acciones.

6.9 Afirmación

{ *afirmación* **}**

La afirmación puede ser una expresión de tipo booleano o puede ser un texto literal de documentación. En el caso de una expresión booleana, la ejecución del programa se detendrá si su evaluación resulta ser falsa. En caso de ser un texto de documentación, la afirmación se asume cierta inmediatamente por lo que no hay ningún efecto en la ejecución.
Si se trata de una expresión booleana, ésta puede contener variables semánticas (3.2.1) y los operadores binarios => (implicación), y <=> (equivalencia) (7.3); también puede haber cuantificación de variables (7.13).

7 Expresiones

7.1 Asignación

variable := *expresión*
Esta expresión resulta igual a la evaluación de la expresión de la derecha promovida al tipo declarado para la variable, siempre y cuando haya compatibilidad (4.2). Una asignación tiene el efecto secundario de asignar a la variable el valor resultante de evaluar la expresión. El operador de asignación := tiene asociatividad por la derecha.

7.2 Condicional

expresión ? *expresión* : *expresión*

La primera expresión debe ser booleana; si ésta resulta cierta, toda la expresión condicional toma el valor de la segunda expresión; si no, toma el valor de la tercera expresión. Las dos expresiones alternativas deben tener el mismo tipo.

7.3 Operación binaria (expresión, expresión)

Sintaxis:

expresión op expresión

Los operadores binarios sobre (*expresión*, *expresión*) se muestran a continuación. Las líneas separan las distintas precedencias en orden de menor a mayor. Todos estos operadores asocian por la izquierda excepto el de asignación que asocia por la derecha.

Operador	*Descripción*
:=	Asignación
<=>	equivalencia lógica
=>	implicación lógica
||	O lógico
&&	Y lógico
|	O aritmético
^	O exclusivo
&	Y aritmético
=	Igualdad

!=	No igualdad
<	menor que
>	mayor que
<=	menor que o igual a
>=	mayor que o igual a
<<	desplazamiento de bits a la izquierda
>>	desplazamiento de bits a la derecha con signo
>>>	desplazamiento de bits a la derecha
+	suma, concatenación de cadenas
−	Resta
*	Multiplicación
/	División
%	módulo división entera

En el caso de los operadores lógicos =>, ||, &&, si el resultado de la evaluación de la expresión izquierda determina el resultado de toda la operación, entonces no se hace la evaluación de la expresión derecha.

7.4 Operación binaria (expresión, tipo)

Sintaxis:

expresión op tipo

Los operadores binarios sobre (*expresión*, *tipo*) se muestran a continuación.

Operador	*Descripción*
implementa	Determina si la expresión es un algoritmo para la especificación indicada en el tipo.
como	Conversión de la expresión al tipo. Ver 7.12.
es_instancia_de	Determina si la expresión es un objeto de la clase indicada en el tipo.

En el caso de los operadores lógicos =>, ||, &&, si el resultado de la evaluación de la expresión izquierda determina el resultado de toda la operación, entonces no se hace la evaluación de la expresión derecha.

7.5 Operación unaria

Sintaxis:

op expresión

Los operadores unarios se muestran en la siguiente tabla.

Operador	*Descripción*
!	Negación lógica
+	Más
−	Menos

~	Negación bit a bit
#	Tamaño de un arreglo o cadena
@	Versión cadena

7.6 Creación de arreglo

La forma básica de creación de un arreglo es:

crear [*expresión* .. *expresión* **]** *tipo*

Las dos expresiones establecen el rango de indización para el arreglo. Cada expresión debe ser del tipo entero. Cada elemento del arreglo será del tipo dado. En el caso de un tipo que sea a su vez arreglo, se ofrece la posibilidad de crear en la misma acción los subarreglos correspondientes. Por ejemplo,

crear [1 .. 10] [1 .. 5] *tipo*

crea un arreglo de diez arreglos (de tipo `[]`*tipo*), cada uno de los cuales creado con espacio para cinco elementos del tipo *tipo*. Esta regla es válida para mayores dimensiones.

La expresión:

crear [1 .. 10] [] *tipo*

crea un arreglo de diez arreglos nulos (cada uno de tipo `[]`*tipo*).

La forma general de creación de arreglos es:

crear [*expresión* .. *expresión* **]**
(**[** *expresión* .. *expresión* **]**)* *tipo*

La forma alternativa

crear [*expresión* **]** ...

es una abreviatura para

crear [0 .. *expresión* – 1 **]** ...

7.7 Creación enumerada de arreglo

Un arreglo no vacío también se puede obtener mediante la enumeración explícita de sus elementos:

[*expresión* (**,** *expresión*)* **]**

La primera expresión establece el tipo del arreglo. Si esta primera expresión es de tipo *t*, entonces toda la expresión del arreglo será de tipo `[]`*tipo*. Las expresiones en la

enumeración deben ser compatibles con la primera. Si se indican *n* expresiones, el arreglo tendrá el rango de subindización desde 0 hasta *n* – 1, inclusive.

7.8 Creación de objeto

Las instancias de una clase se crean mediante el siguiente mecanismo:

crear *nombre* [**(** [*expresiones*] **)**]

donde *nombre* indica el nombre de una clase, y las expresiones deben corresponder en tipo con las entradas especificadas para alguno de los constructores de tal clase (5.3.2). El resultado de esta expresión es una nueva instancia de la clase correspondiente.

7.9 Subindización

expresión **[** *expresión* **]**

La primera expresión debe ser de tipo arreglo o de tipo cadena. La segunda expresión debe ser de tipo entero y corresponde al índice. Se produce un error de ejecución si se trata de subindizar un arreglo o cadena por fuera del rango válido respectivo (7.6).

7.10 Referencia a atributo

expresión **.** *id*

La primera expresión debe ser un objeto, un arreglo o una cadena, y el identificador a continuación del punto debe corresponder con el nombre de uno de los atributos de la expresión. Se produce un error de ejecución si el valor de la expresión resulta en la referencia nula.

El tipo de toda la expresión será igual al tipo declarado para el atributo denotado por el identificador. Esta declaración ha sido explícita en el caso que la expresión corresponda a una instancia de una clase, o implícita en el caso de arreglos y cadenas como se define en 3.2.2.

7.11 Invocación

La invocación de un algoritmo se escribe:

nombre **(** [*expresiones*] **)**

donde *nombre* indica el nombre posiblemente completo (con paquete) de un algoritmo, y las expresiones en paréntesis indican los argumentos de entrada. Los tipos de estas expresiones deben ser compatibles con los tipos esperados por el algoritmo de acuerdo con la especificación correspondiente.

7.12 Conversión de tipo

expresión **como** *tipo*

Hace la conversión del tipo de la expresión al tipo indicado. Toda expresión es convertible a su mismo tipo y al tipo cadena.

Tipo expresión	*Convertible a*	*Efecto*
`entero`	`real`	Versión en punto flotante.
	`caracter`	Caracter Unicode correspondiente.
	`booleano`	Cierto sólo si la expresión es diferente de 0.
	`cadena`	Versión cadena del valor.
`real`	`entero`	Redondeo al entero más cercano.
	`cadena`	Versión cadena del valor.
`booleano`	`entero`	1 sólo si la expresión es cierta.
	`cadena`	Versión cadena del valor.
`caracter`	`entero`	Código Unicode correspondiente.
	`cadena`	Versión cadena del valor.
`algoritmo`	`algoritmo para X`	El algoritmo genérico se toma como un algoritmo para una especificación concreta X.
	`cadena`	Descripción del esquema del algoritmo, estilo "`algoritmo nombre(x:tipo)->y:tipo`"
`[]` *tipo*	`cadena`	Valor del arreglo estilo "`[x, y, ...]`"
objeto de clase `X`	objeto de clase `Y` si `Y es` una super- o sub-clase de X	La expresión se toma como instancia de la clase Y.
	`cadena`	Valor del objeto estilo "`{atributo1 = valor1, atributo2 = valor2, ...}`"

7.13 Identificador

id

La evaluación de esta expresión resulta en el valor asignado para la variable de nombre indicado. Este identificador debe haberse declarado explícitamente con anterioridad. Se genera un error en ejecución si la variable no tiene un valor asignado.

7.14 Literal

literal

Esta expresión tiene el valor literal indicado de acuerdo a como se define en 1.1.3.2.

7.15 Cuantificación de variables

La cuantificación de variables sólo puede aparecer dentro de afirmaciones (6.7) y tiene la siguiente sintaxis general:

(**existe** | **para_todo**) *declaraciones* [, *expresión*] : *expresión*

En compilación se verifica la validez de las declaraciones de variables, como también que las expresiones sean de tipo booleano. El tipo de toda la expresión es booleano. En

ejecución no se hace ningún tratamiento especial y toda la expresión se toma inmediatamente como cierta.

8 Algoritmos en otros lenguajes

La definición de un algoritmo admite una variante para permitir su codificación de acciones en otros lenguajes de programación. La sintaxis es:

```
algoritmo nombreAlgoritmo para prototipoEspecificación
  implementación lenguaje código
fin algoritmo
```

Ambos elementos *lenguaje* y *código* son cadenas literales. Estas pueden encerrarse entre comillas dobles o entre los símbolos `{%` y `%}`. El primer elemento, *lenguaje*, indica el lenguaje en que se hace la implementación, y el segundo, *código*, corresponde al código en dicho lenguaje. En fase de compilación no se hace ninguna revisión a los contenidos de dichas cadenas, pero sí en fase de ejecución para llevar a cabo los preparativos necesarios de interfaz entrada/salida para comunicar el entorno de ejecución Loro con el código del lenguaje de implementación. Actualmente se cuenta con dos posibilidades, ambas directamente basadas en el lenguaje Java.

8.1 BeanShell

Esta es la opción que ahora da más flexibilidad para acceder a todo el catálogo de clases Java disponible. La palabra clave para indicar el lenguaje es `bsh`. Para más detalles sobre el sistema BeanShell, consúltese http://beanshell.org.

La interfaz con Loro se establece de la siguiente manera. Se crea un intérprete BeanShell y se inicializa con las entradas recibidas por el algoritmo de tal manera que se hagan accesibles en el código. Para efectos de la eventual salida producida por el algoritmo, Loro toma el valor resultante de la evaluación del código fuente si éste no es nulo, o bien lee el valor asociado a nombre formal del parámetro de salida.

Suponiendo que existe la especificación `ingresar`, el siguiente ejemplo muestra cómo podría utilizarse BeanShell para hacer una posible implementación.

```
algoritmo para ingresar(mensaje: cadena)-> cad: cadena
implementación "bsh"
{%
      return JOptionPane.showInputDialog(null, mensaje);
%}
fin algoritmo
```

8.2 Método estático Java

En esta opción se indica un cierto método Java estático de implementación de acuerdo con el prototipo del algoritmo. La palabra clase es `java`. En el elemento código se debe indicar el nombre completo de la clase que contiene el mencionado método esperado.

Por ejemplo, suponiendo que existe la especificación `leerCadena`, lo siguiente muestra cómo indicar una implementación basada en método estático Java:

```
algoritmo para leerCadena() -> v: cadena
      implementacion "java" "impl.Lector";
fin algoritmo
```

En ejecución, el sistema Loro carga la clase Java indicada `impl.Lector`, y ejecuta el método estático `impl.Lector.leerCadena`. Para más detalles técnicos, favor consultar la documentación de desarrollo.

9 Apéndice

La Gramática

Se utiliza una notación BNF extendida para definir la gramática del lenguaje. Esta notación se describe a continuación.

Producciones
La gramática se define como una lista de producciones. Cada *producción* se compone de tres partes:

nombreProducción `::=` *formaProducción*

- Un nombre propio para la producción.
- El símbolo `::=` para separar el nombre de la producción de la forma que puede tomar dicha producción.
- La forma sintáctica que puede tener la producción.

Cada *forma sintáctica* para las producciones es una secuencia de elementos, donde cada elemento puede ser:

- El nombre de una producción.
- Un componente léxico.
- Un símbolo especial para facilitar la definición:
  - `|`
    (Barra vertical) Se usa como operador para separar diferentes alternativas de forma.
  - `( )`
    Agrupa una secuencia de elementos con la finalidad de mostrar alternativas internas y/o para la aplicación de uno de los símbolos especiales `|, ?, *, +` (ver a continuación).
  - `( ... )?`
    Significa que lo denotado entre los paréntesis es opcional.
  - `[ ... ]`
    Significa que lo denotado entre los paréntesis es opcional. (Es una manera alternativa para `( ... )?`.)
  - `( ... )*`
    Significa que lo denotado entre los paréntesis puede aparecer cero o más veces.

- `( ... )+`
  Significa que lo denotado entre los paréntesis puede aparecer una o más veces.

Componentes léxicos

Los componentes léxicos se muestran de dos maneras en la gramática:

- Explícitamente encerrados entre comillas dobles cuando sólo se trata de una sóla posibilidad que debe tomarse literalmente (como `"clase"`, `"::"`, `"falso"`).
- A través de un nombre encerrado entre corchetes angulares (< y >) para referir al componente léxico correspondiente, la mayoría de los cuales se definen en 1.1.3, además de `<EOF>` que significa “fin de entrada” y los siguientes para permitir opcionalmente el uso de tildes:

```
<CONTINUE>       ::= "continue" | "continúe"
<DESCRIPCION>    ::= "descripcion" | "descripción"
<ESPECIFICACION> ::= "especificacion" | "especificación"
<IMPLEMENTACION> ::= "implementacion" | "implementación"
<SEGUN>          ::= "segun" | "según"
<SUPER>          ::= "super" | "súper"
<OPERACION>      ::= "operacion" | "operación"
<METODO>         ::= "metodo" | "método"
```

NOTA:
Para facilitar la lectura en el cuerpo del documento, los componentes léxicos correspondientes a palabras reservadas no se muestran entre comillas (ej. `"clase"`), sino en negrilla (**`clase`**). Así mismo, las producciones y algunos componentes léxicos se escriben en cursiva (como *`expresión`*, *`acciones`*, *`id`*, etc.).

La Gramática

La gramática concreta de Loro es la siguiente.
NOTA: Tabla generada de manera automática con base en la implementación disponible a la fecha. Incluye algunos elementos en desarrollo que no se encuentran aún completamente implementados o documentados.

```
     fuente ::= ( paquete )? ( utiliza ";" )* ( unidad )+ <EOF>
     unidad ::= ( especificacion | algoritmo | clase | interface_ )
 interface_ ::= "interface" tid ( "extiende" tnombre ( "," tnombre )* )?
                ( <DESCRIPCION> )? tdoc ( especificacion )* "fin"
                "interface"
    paquete ::= "paquete" tnombre ";"
    utiliza ::= "utiliza" ( <ESPECIFICACION> | "algoritmo" | "clase" )
                tnombre
      clase ::= ( "clase" | "objeto" ) tid ( "extiende" tnombre )? (
                "para" tnombre ( "," tnombre )* )? ( <DESCRIPCION> )?
                tdoc ( declDescs )? ( constructor )* ( algoritmo )* "fin"
                ( "clase" | "objeto" )
 declDescs ::= declDesc ";" ( declDesc ";" )*
  declDesc ::= tid ":" ntipo ( ( ":=" expresion | "constante" ( "="
```

```
                    expresion )? ) )? ( ":" )? tdoc
      constructor ::= "constructor" "(" ( declaraciones )? ")" ( <DESCRIPCION>
                    )? tdoc ( "entrada" ( "nada" | descripciones ) )? ( "pre"
                    afirmacion )? ( "pos" afirmacion )? "estrategia" tdoc
                    "inicio" acciones "fin" "constructor"
    especificacion ::= ( <ESPECIFICACION> | <OPERACION> ) tid "(" (
                    declaraciones )? ")" ( "->" declaraciones )? (
                    <DESCRIPCION> )? tdoc ( "entrada" ( "nada" |
                    descripciones ) )? ( "salida" ( "nada" | descripciones )
                    )? ( "pre" afirmacion )? ( "pos" afirmacion )? "fin" (
                    <ESPECIFICACION> | <OPERACION> )
     descripciones ::= ( descripcion )+
       descripcion ::= tid ":" tdoc ( ";" )?
         algoritmo ::= ( "algoritmo" | <METODO> ) ( tid )? "para" tnombre "(" (
                    declaraciones )? ")" ( "->" declaraciones )? ( (
                    <DESCRIPCION> | "estrategia" )? tdoc "inicio" acciones |
                    <IMPLEMENTACION> timpl timpl ( ";" )? ) "fin" (
                    "algoritmo" | <METODO> )
     declaraciones ::= declaraciones1Tipo ( "," declaraciones1Tipo )*
       declaracion ::= tid ":" ntipo
declaraciones1Tipo ::= tids ":" ntipo ( ( ":=" expresion | "constante" ( "="
                    expresion )? ) )?
  declaracion1Tipo ::= tids ":" ntipo ( ( ":=" expresion | "constante" ( "="
                    expresion )? ) )?
             ntipo ::= ( ntipobasico | "[" "]" ntipo | "interface" tnombre |
                    "algoritmo" ( "para" tnombre )? | ( "clase" )? tnombre )
       ntipobasico ::= ( "entero" | "booleano" | "caracter" | "real" | "cadena"
                    )
          acciones ::= ( accion ";" )*
            accion ::= ( declaracion1Tipo | expresion | decision |
                    decisionMultiple | iteracion | retorne | intente | lance
                    | afirmacion | termine | _continue )
accionesInterprete ::= ( accionInterprete ( ";" accionInterprete )* ( ";" )? )?
                    <EOF>
  accionInterprete ::= ( accion | utiliza )
         iteracion ::= ( mientras | para | repita | ciclo )
        afirmacion ::= "{" ( expresion | <TEXT_DOC> ) "}"
          decision ::= "si" expresion "entonces" acciones ( decision_si_no_si )*
                    ( "si_no" acciones )? "fin" "si"
 decision_si_no_si ::= "si_no_si" expresion "entonces" acciones
  decisionMultiple ::= <SEGUN> expresion "haga" ( "caso" expresion ":" acciones
                    ( "fin" "caso" ( ";" )? )? )* ( "si_no" ( ":" )? acciones
                    ( "fin" "caso" ( ";" )? )? )? "fin" <SEGUN>
          mientras ::= "mientras" ( "/" tid "/" )? expresion "haga" acciones
                    "fin" "mientras"
              para ::= "para" ( "/" tid "/" )? ( declaracion | tid ) ( "desde" |
                    ":=" ) expresion ( "bajando" )? ( "paso" expresion )?
                    "hasta" expresion "haga" acciones "fin" "para"
            repita ::= "repita" ( "/" tid "/" )? acciones "hasta" expresion
             ciclo ::= "ciclo" ( "/" tid "/" )? acciones "fin" "ciclo"
           retorne ::= "retorne" ( expresiones )?
           termine ::= "termine" ( "/" tid "/" )? ( "si" expresion )?
         _continue ::= <CONTINUE> ( "/" tid "/" )? ( "si" expresion )?
           intente ::= "intente" acciones ( "atrape" "(" declaracion ")"
                    acciones ( "fin" "atrape" )? )* ( "siempre" acciones )?
```

```
                    "fin" "intente"
          lance ::= "lance" expresion
     expresiones ::= expresion ( "," expresion )*
       expresion ::= e_cond ( ":=" expresion )?
          e_cond ::= e_implic ( "?" expresion ":" e_cond )?
        e_implic ::= e_o ( ( "<=>" e_o | "=>" e_o ) )*
             e_o ::= e_y ( "||" e_y )*
             e_y ::= e_oarit ( "&&" e_oarit )*
          e_oarit ::= e_oexc ( "|" e_oexc )*
           e_oexc ::= e_yarit ( "^" e_yarit )*
          e_yarit ::= e_igual ( "&" e_igual )*
          e_igual ::= e_instancia ( "=" e_instancia | "!=" e_instancia )*
     e_instancia ::= e_rel ( ( <IMPLEMENTA> ntipo | <COMO> ntipo |
                     "es_instancia_de" ntipo ) )?
           e_rel ::= e_corr ( "<" e_corr | ">" e_corr | "<=" e_corr | ">="
                     e_corr )*
          e_corr ::= e_sum ( "<<" e_sum | ">>" e_sum | ">>>" e_sum )*
           e_sum ::= e_mul ( "+" e_mul | "-" e_mul )*
           e_mul ::= e_unaria ( "*" e_unaria | "/" e_unaria | "%" e_unaria )*
       e_unaria ::= ( "!" e_unaria | "-" e_unaria | "+" e_unaria | "~"
                     e_unaria | "#" e_unaria | "@" e_unaria | e_primaria | (
                     "existe" | "para_todo" ) declaraciones ( "," expresion )?
                     ":" e_unaria )
      e_primaria ::= e_prefijoPrimaria ( ( "." tid ) | ( "[" expresion "]" ) |
                     ( "(" ( expresiones )? ")" ) )*
e_prefijoPrimaria ::= ( e_const | "[" expresion ( "," expresion )* ( "," )? "]"
                     | e_nombre | "(" expresion ")" | e_crear )
         e_crear ::= "crear" ( "[" expresion ( ".." expresion )? "]"
                     tamanoArreglo | tnombre ( "(" ( expresiones )? ")" )? )
   tamanoArreglo ::= ( ntipo | "[" expresion ( ".." expresion )? "]"
                     tamanoArreglo )
         e_const ::= ( e_literalEntero | e_literalBooleano | e_literalReal |
                     e_literalCaracter | e_literalCadena | e_literalNulo )
   e_literalNulo ::= "nulo"
 e_literalEntero ::= <LITERAL_ENTERO>
   e_literalReal ::= <LITERAL_REAL>
e_literalBooleano ::= ( "cierto" | "falso" | "pre" )
 e_literalCadena ::= <LITERAL_CADENA>
e_literalCaracter ::= <LITERAL_CARACTER>
            e_id ::= tid
        e_nombre ::= tid ( "::" tid )*
             tid ::= <ID>
          tidEOF ::= tid <EOF>
            tids ::= tid ( "," tid )*
         tnombre ::= tid ( "::" tid )*
      tnombreEOF ::= tnombre <EOF>
           timpl ::= ( <IMPL> | <LITERAL_CADENA> )
            tdoc ::= ( <TEXT_DOC> | <LITERAL_CADENA> )
```