Universidad Centroamericana
"José Simeón Cañas"
Depto. Electrónica e
Informática

# Armus

Lenguaje de Programación
Orientado a Objetos

## Documento Técnico

Manual de Referencia y Ayuda

**Materia:** Compiladores

**Catedrático:** Lic. José Enmanuel Amaya Araujo

**Desarrollado por grupo ACL:**

- Néstor Santiago Aldana Rodríguez
- Bárbara Stefany Aparicio Bermúdez
- Katherine Stephanie Cabrera Blanco
- Diana Marcela López Rosales
- Carlos Miguel López Loarca

UCA | Diciembre 2016

# Índice

# **A**. Descripción del diseño del lenguaje y de sus posibles aplicaciones.

El Lenguaje de programación se basa en la ejecución de algoritmos en pseudo-código en el lenguaje español, lo cual fortalece el práctico entendimiento en las personas que se inician en el ámbito de la programación, y además brinda una de una herramienta más clara para el aprendizaje del paradigma de programación orientada a objetos.

**Origen del lenguaje Armus**

El nombre del lenguaje que se encuentra en la etapa de diseño e identificación lexicográfica ha sido nombrado como **Armus**, dicho nombre tiene origen en el nombre de una estrella llamada Eta Capricorni[1] (también conocida como Armus) es una estrella binaria en la constelación de Capricornio.

**Estructura y sintaxis**

A continuación, se define toda la estructura y sintaxis para la ejecución de algoritmos.

- Paradigma del lenguaje: Programación Orientada a Objetos (POO)
- Lenguaje sensitivo a mayúsculas y minúsculas (case sensitive).

<table>
<tr><td>Manejo de múltiples archivos de código fuente</td><td colspan="2">El lenguaje soporta la inclusión de múltiples archivos, en uno o más archivos que conforman el código fuente completo del programa escrito en el lenguaje Armus.<br><br>El parámetro de configuración MAX_FILE define el número máximo de archivos que pueden existir conteniendo todo el código fuente del programa.<br><br>El parámetro de configuración MAX_NAME_FILE define el número máximo de caracteres que puede tener el nombre o ruta del archivo a incluir.<br><br>Estructura:<br>incluir <ruta o nombre del archivo a ser incluido>;<br><br>Ejemplos:<br>incluir “/home/mike/Documentos/proyecto/Principal.acl”;<br>incluir “../Documentos/proyecto/Fuentes.acl”;<br>incluir “Calculos.acl”;<br><br>NOTA: La ruta puede ser absoluta o relativa en Linux exclusivamente.</td></tr>
<tr><td>Manejo de variables</td><td>Estructura (dentro de métodos):<br><tipo de dato> <variable alfanumérica>;<br><br>Ejemplos de sintaxis:<br>entero n1;<br>byte num, contador, i;<br>real temp;</td><td>Estructura (dentro de clases):<br>publica | privada <tipo> <variable><br><br>Ejemplos de sintaxis:<br>publica entero n2;<br>privada cadena cad = “carlos”;<br>publica real r1 = 3.1416;</td></tr>
<tr><td>Tipos de datos</td><td colspan="2">1. vacio (equivalente a void)<br>2. booleano (verdadero o falso)</td></tr>
</table>

[1] https://es.wikipedia.org/wiki/Eta_Capricorni

| | |
|---|---|
| | 3. **cadena** (cadena de caracteres)<br>4. **caracter** (equivalente a char), 2 Bytes<br>5. **byte** (-127 a 128), 1 Byte<br>6. **entero** (equivalente a int), 4 Bytes<br>7. **real** (equivalente float), 4 Bytes<br>8. **Objeto** (equivalente a Object)<br>9. **Archivo** (equivalente a File) |
| **Instrucciones de Selección** | **si**[<condición>]{<br>  <sentencias>;<br>}**sino**{<br>  <sentencias>;<br>} |
| | **si**[<condición>, <valor verdadero>, <valor falso>];<br><br>**NOTA:** la opción verdadera y falsa no se pueden omitir y deben ser valores. |
| | **probar**[<condición>]{<br>  **caso 1**: <sentencia1>;<br>  <sentencia2>;<br>  **romper**;<br>  **caso 2**: <sentencia1>;<br>  <sentencia2>;<br>  **romper**;<br>  **defecto**: <sentencia1>;<br>  <sentencia2>;<br>  **romper**;<br>} |
| **Instrucciones Iterativas** | **mientras**[<condición>]{<br><br>}  /* el equivalente a while */ |
| | **para**[<valor inicial>, <condición>, <incremento>]{<br><br>}  /* el equivalente a for */ |
| | **hacer**{<br><br>}**mientras**[<condición>];<br><br>/* el equivalente a do-while */ |
| | **paraCada**[<tipo> <variable>, <arreglo/lista>]{<br><br>} /* el equivalente a for-each */ |
| **Instrucciones de entrada y salida** | Sintaxis de entrada:<br>**Sistema.obtenerEntero**[<var>]; //lee un entero y lo almacena en var<br><br>**Sistema.obtenerReal**[<var>]; //lee un núm. real y lo almacena en var<br><br>**Sistema.obtenerCadena**[<var>]; //lee una cadena y lo almacena en var<br><br>**Sistema.obtenerCaracter**[<var>]; //lee un carácter y lo almacena en var |

| | |
|---|---|
| | Sintaxis de salida:<br>**Sistema.mostrar**[“hola”];<br><br>**Sistema.mostrar**[valor]; //valor puede ser cualquier tipo |
| **Sub-Algoritmos** | **publica** <función>[<parámetros entrada>] <tipo salida> {<br>/* puede no tener parámetros de entrada<br>y el tipo de salida puede ser vacío */<br>**retornar** <valor de retorno>;<br>}<br><br>**privada** <función>[<parámetros entrada>] <tipo salida> {<br>/* puede no tener parámetros de entrada<br>o el tipo de salida puede ser vacío */<br>**retornar** <valor de retorno>;<br>}<br><br>**NOTA:** Los procedimientos pueden o no recibir parámetros de entrada (opcional), pero NO retornan nada.<br><br>**publica** <procedimiento>[<parámetros entrada>]{<br><br>}<br><br>**privada** <procedimiento>[<parámetros entrada>]{<br><br>}<br><br>**NOTA 2:** En ambos casos se utilizan sobre objetos.<br><br><objeto>.<función>[<parámetros entrada>] <tipo salida>;<br><br><objeto>.<procedimiento>[<parámetros entrada>];<br><br><objeto>.<procedimiento>[];<br><br>Ejemplo:<br>Alumno.obtenerNombre[]; |
| **Paso de parámetros por valor y por referencia** | Por valor<br><br><objeto>.<función>[11, 24];<br>..<br><función>[entero n1, entero n2]{<br><br>}<br><br>Por referencia<br><br><objeto>.<función>[~<parámetro>];<br>..<br>entero n1 = 11;<br><objeto>.<función>[~n1]<br>..<br><función>[entero *n]{<br>n = 24;<br>}<br>.. |

Opcional

Opcional

Si el parámetro es enviado por referencia, se antepone el símbolo ~

Si el parámetro es recibido por referencia en la definición de la función se coloca * antes del nombre del parámetro.

| | |
|---|---|
| | Sistema.mostrar[n1]; //muestra: 24<br><br>**NOTA:** Se ha pasado por la referencia el parámetro enviado a la función, es decir, que todo lo que se cambie al parámetro dentro de la función también se cambiará en el parámetro original (no se hace una copia del parámetro). |
| **Manejo de Arreglos** | **Arreglo<**tipo**>** <objeto arreglo>;<br><br><objeto arreglo>.**agregar[**<elemento1>**];**<br><objeto arreglo>.**agregar[**<elemento2>**];**<br><br><objeto arreglo>.**obtener[**<elemento1>**];**<br><objeto arreglo>.**obtener[**<elemento2>**];**<br><br><objeto arreglo>.**cuantos[];**<br><objeto arreglo>.**quitar[**<elemento>**];**<br><br><u>Ejemplo:</u><br>**Arreglo<**cadena**>** Lista;<br>Lista.**agregar**[“lunes”];<br>Lista.**agregar**[“martes”];<br>**paraCada**[cadena dia, Lista]{ Sistema.**mostrar**[dia]; } |
| **Manejo Básico de Archivos** (apertura y volcado) | **Archivo** <objeto>;<br><br><objeto>.**abrir[**<ruta o nombre del archivo>**];**<br><br><objeto>.**leerLinea[];** /*lee solo una línea del archivo*/<br><br><objeto>.**volcado[];** /*obtiene todo el contenido del archivo*/<br><br><objeto>.**cerrar[];** /*cierra el archivo*/ |
| **Funciones Predefinidas** | **concatenar[**<cadena1>, <cadena2>**];**<br><br>**parteEntera[**<número real>**];** /*extrae la parte entera de un número real*/<br><br>**comparar[**<cadena1>,<cadena2>**];** /*devuelve 1 si son iguales*/<br><br>**mayor[**<num1>, <num2>**];** /*devuelve el mayor de ambos números */<br><br>**menor[**<num1>, <num2>**];** /*devuelve el menor de ambos números */<br><br>**esPar[**<numero>**];** /*devuelve 1 si es par, 0 si no lo es*/<br><br>**decimalBin[**<número entero>**];** /*transforma de decimal a binario*/<br><br>**potencia[**<número>**,** <exponente>**];** /* ejemplo: $24^8$ */<br><br>**absoluto[**<número>**];** /*obtiene el valor absoluto de un numero*/<br><br>**modulo[**<entero1>,<entero2>**];** /*obtiene el residuo de una división*/<br><br>**longitudCadena[**<cadena>**];** /*devuelve la longitud de una cadena*/<br><br>**NOTA:** todas están definidas en el objeto **Sistema** |

**Operadores Aritméticos y Lógicos**

Lógicos

| Operador | | Descripción |
|---|---|---|
| == | | Igual |
| < | | menor |
| <= | | Menor o igual |
| > | | Mayor |
| >= | | Mayor o igual |
| <> | != | Distinto |
| && | | Operador Y |
| \|\| | | Operador O |
| ! | | Negación |

Aritméticos

| Operador | Descripción |
|---|---|
| + | Suma |
| - | Resta |
| * | Multiplicación |
| / | División |

Orden de Prioridad

| Operador | Descripción |
|---|---|
| () | Paréntesis |
| * , / | Multiplicación, División |
| + , - | Suma, Resta |
| ==,<>,>,<,>=,<= | Igual, Distinto, Mayor, Menor, Mayor o igual, Menor o igual |

**Comentarios**

```
// comentario de una sola línea

/*
 Comentario para párrafos.
*/
```

## B. Palabras Reservadas

```
entero          caso            retornar        menor
byte            defecto         Arreglo         esPar
real            romper          agregar         decimalBin
vacio           mientras        obtener         pontencia
booleano        para            cuantos         absoluto
cadena          hacer           quitar          modulo
caracter        paraCada        abrir           longitudCadena
Objeto          Sistema         leerLinea       clase
Archivo         obtenerEntero   volcado         incluir
si              obtenerReal     cerrar          verdadero
sino            obtenerCadena   concatenar      false
probar          obtenerCaracter parteEntera     obtenerBooleano
publica         mostrar         comparar        mayor
privada         local           principal
```

## C. Expresión regular para los identificadores

- VARIABLES

```
identificador = (letra + símbolo)(letra + digito + símbolo)*
```

Donde:

```
letra = {a, b, .., ñ, .., z, A, B, .., Ñ, .., Z, á, é, í, ó, ú, Á, É, Í, Ó, Ú}
digito = {0,1,2,3,4,5,6,7,8,9}
símbolo = {“_”}
```

- CLASES *(Sub-conjunto de identificador de variable)*

```
identificador = letraMa(letra + digito + símbolo)*
```

Donde:

```
letraMa = {A, B, C, .., Z}
letra = {a, b, .., z, A, B, .., Z}
digito = {0,1,2,3,4,5,6,7,8,9}
símbolo = {“_”}
```

## D. Expresiones regulares para números enteros y números reales

- NÚMEROS ENTEROS & HEXADECIMALES

Un número entero puede ser definido como un numero Hexadecimal, Ejemplo: **#**A32FF (anteponiendo el símbolo **#**)

```
entero = digito(digito)* + #(letra + digito)*
```

Donde:

```
digito = {0,1,2,3,4,5,6,7,8,9}
letra = {A, B, C, D, E, F} (números Hexadecimal 10, 11, 12, 13, 14, 15)
digito = {0,1,2,3,4,5,6,7,8,9}
```

- NÚMEROS REALES

```
real = entero.entero
```

## E. Expresiones regulares para cadenas y caracteres

- CADENA

```
cadena = “(letra + digito + símbolo)*”
```

Donde:

```
letra = {a, b, .., ñ, .., z, A, B, .., Ñ, .., Z, á, é, í, ó, ú, Á, É, Í, Ó, Ú}
digito = {0,1,2,3,4,5,6,7,8,9}
símbolo = {x / x es cualquier carácter diferente de letra, digito o “}
```

- CARÁCTER

```
caracter = ‘(letra + digito + símbolo)’
```

Donde:

```
letra = {a, b, .., ñ, .., z, A, B, .., Ñ, .., Z, á, é, í, ó, ú, Á, É, Í, Ó, Ú}
digito = {0,1,2,3,4,5,6,7,8,9}
símbolo = {x / x es cualquier carácter diferente de letra, digito o ‘}
```

## F. Lista de operadores y caracteres especiales

Lógicos

| Operador | | Descripción |
|---|---|---|
| == | | Igual |
| < | | Menor |
| <= | | Menor o igual |
| > | | Mayor |
| >= | | Mayor o igual |
| <> | != | Distinto |
| && | | Operador Y |
| \|\| | | Operador O |
| ! | | Negación |

Aritméticos

| Operador | Descripción |
|---|---|
| + | Suma |
| - | Resta |
| * | Multiplicación |
| / | División |

Caracteres Especiales

| Carácter | Descripción |
|---|---|
| ~ | Referencia |
| . | Punto |
| # | Inicio de Número Hexadecimal |
| ; | Punto y coma |
| , | Coma |
| : | Dos puntos |
| = | Asignación |
| { | Llave de apertura |
| } | Llave de cierre |
| [ | Corchete de apertura |
| ] | Corchete de cierre |
| ( | Paréntesis de apertura |
| ) | Paréntesis de cierre |

**NOTA**: En la composición de cadenas se agregan además todos los caracteres de la tabla ASCII

## G. Forma de construcción de comentarios en el lenguaje

Para la utilización de los comentarios se hará de igual forma que en lenguaje C y Java, para los comentarios de una sola línea se utiliza **//** , para los comentarios de párrafos **/* */**

Ejemplos:

```
// comentario de una sola línea

/*
 Comentario para párrafos.
*/
```

## H. Un ejemplo de programa para escribir “Hola mundo” en el lenguaje

Manejo de ámbito de variables y métodos: **pública** y **privada**

```
publica clase <nombre_clase>{
     privada entero variable;
     privada metodo[]{
          entero var;
          var = 1985;
     }
}
```

Ejemplo (hola mundo):

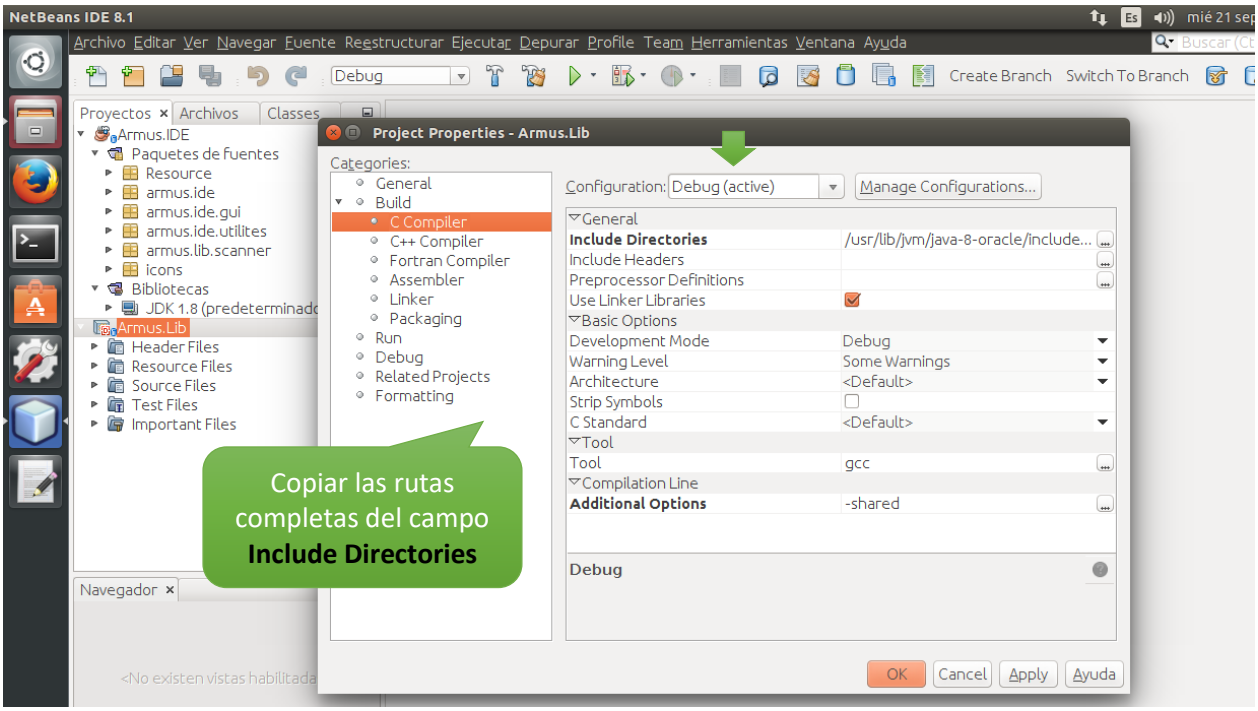
```
publica clase Hola{

     publica principal []{
          Sistema.mostrar[“Hola mundo”];
     }
     /*comentario*/
}
```

Ejemplo:

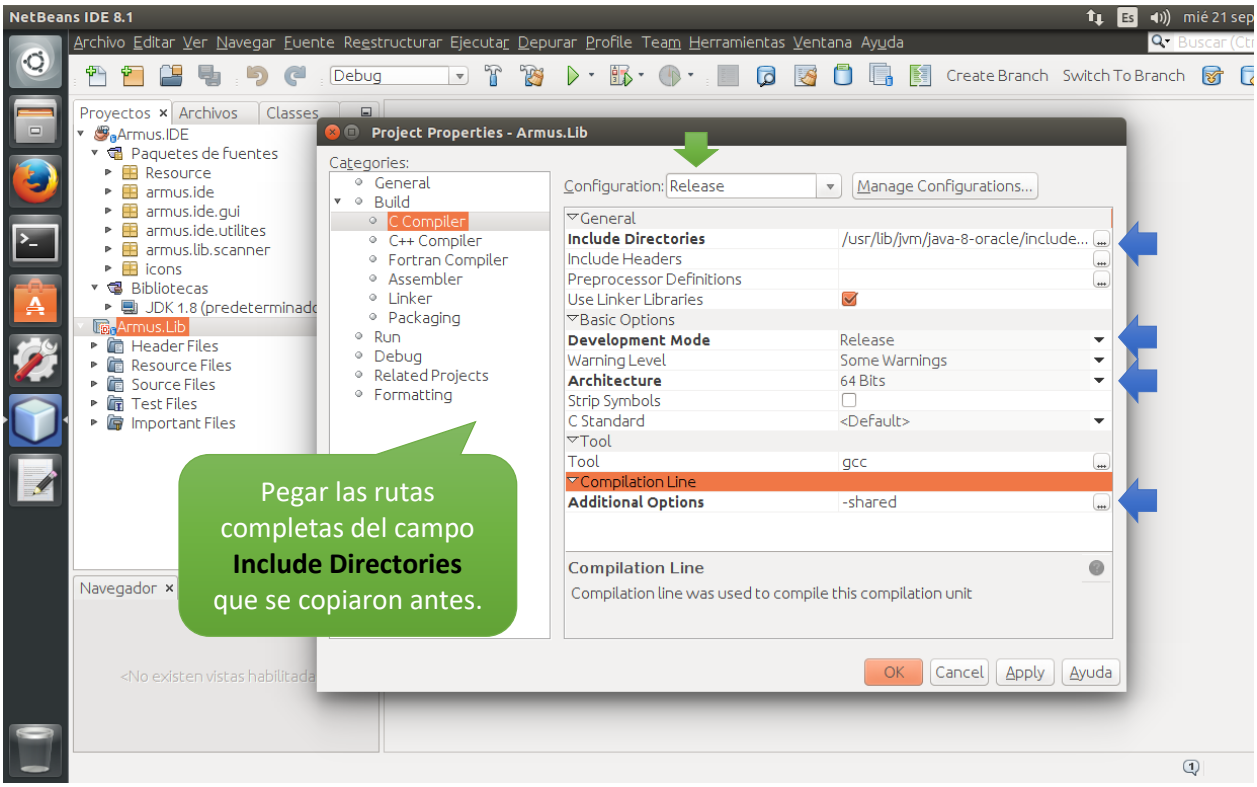
```
publica clase Prueba {
     publica calculo3x5[] entero { //retorna un valor de tipo entero
          entero m = 0;
          m = 3 * 5;
          retornar m;
     }
     publica principal[]{
          cadena cad; //declaración de variable de tipo cadena
          entero n; //declaración de variable de tipo entero
          Sistema.mostrar[“Ingrese su nombre\n”];
          Sistema.obtenerCadena[cad]; //lee desde el teclado una cadena de caracteres
          si[Sistema.comparar[cad, “carlos”]{
               Sistema.mostrar[“son iguales\n”];
          }sino{
               probar[cad]{
                    caso 1: Sistema.mostrar[“son diferentes\n”];
                            romper;
                    caso 2: cad = “nestor”;
                            romper;
          }
          n = Sistema.calculo3x5[];
     }
     /*comentario*/
}
```

# I. Configuración para la generación de la librería de JNI para el IDE y Analizador Lexicográfico en Netbeans sobre Linux Ubuntu 14.04

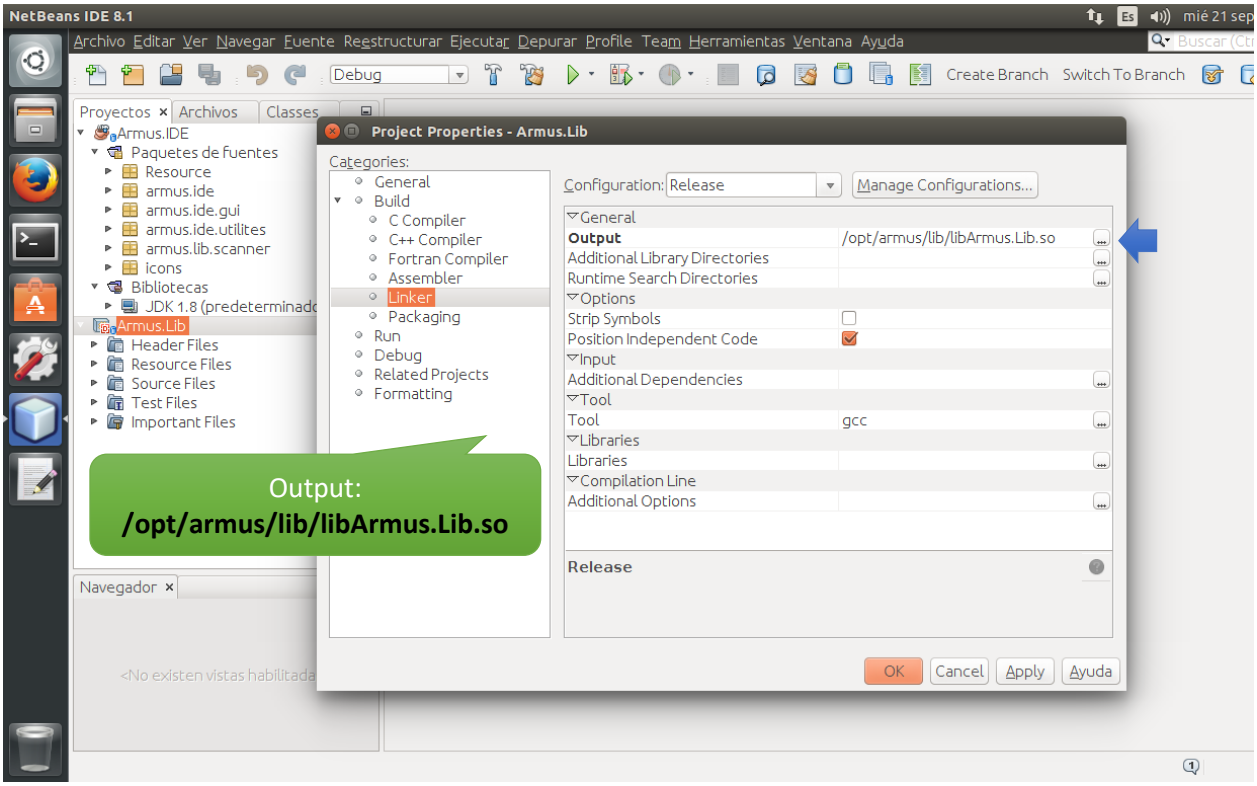
Abrimos **Netbeans** y hacemos clic derecho sobre el proyecto **Armus.Lib**, seleccionamos **Propiedades** y en la opción de **C Compiler** cambiamos en la parte de arriba Debug (active) por **Release**.



Una vez realizado ese cambio colocamos la siguiente configuración en los parámetros mostrados.




Ahora hacemos clic en la opción de **Linker** y añadimos la ruta en el campo **Output**.



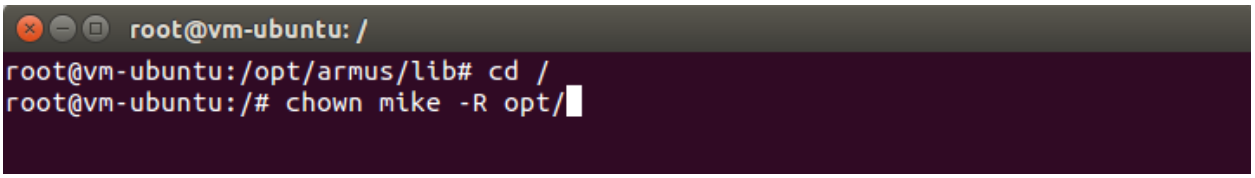
Dicho directorio debe ser creado previamente con los permisos del usuario que ejecutará y generará la librería en Netbeans, para ello, abrimos una terminal (o consola) en Linux y escribimos los siguientes comandos:

| | |
|---|---|
| `sudo su` | Accedemos como usuario root |
| `cd opt/` | Entramos al directorio opt |
| `mkdir armus/` | Creamos la carpeta armus |
| `mkdir lib/` | Creamos la carpeta lib |

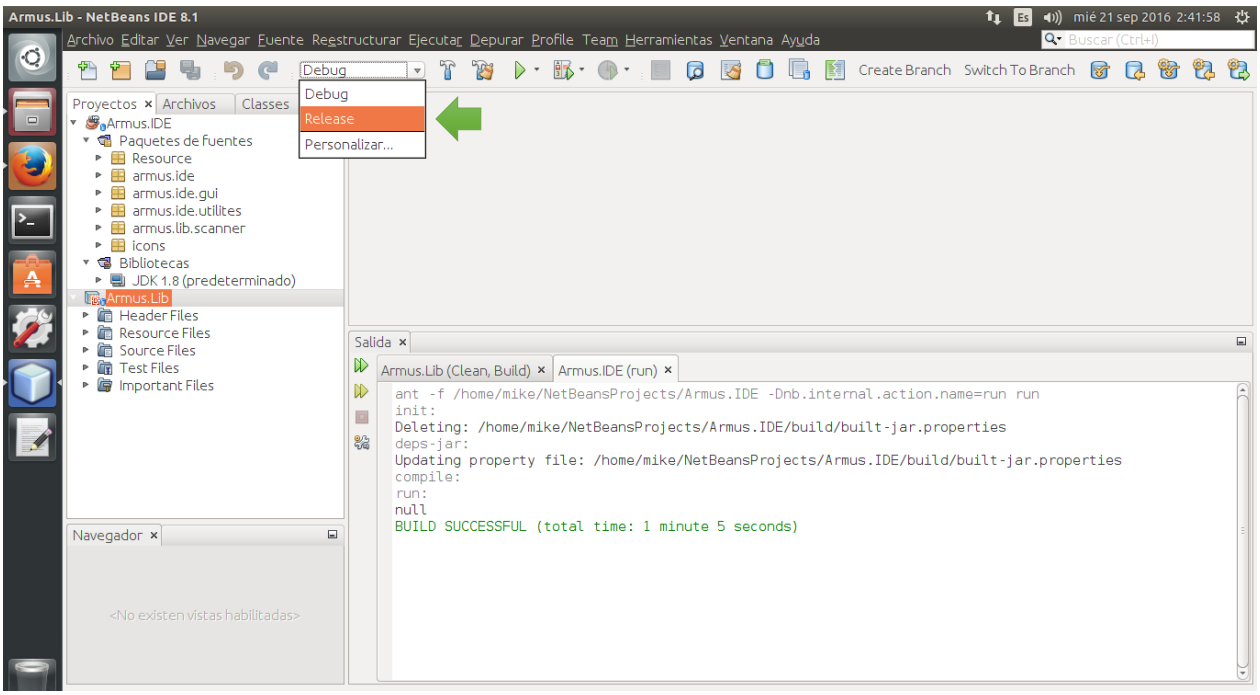
El archivo **libArmus.Lib.so** siempre debe estar en esta ruta para que el programa IDE y Analizador Léxico funcionen correctamente.


Ahora creamos los permisos para el usuario **mike** en dicha carpeta:

| | |
|---|---|
| `chown mike -R opt/` | Otorga permisos al usuario **mike** recursivamente en el directorio **opt** |




Ahora sí, vamos nuevamente a Netbeans y en la parte superior cambiamos de Debug a Release, ejecutamos el proyecto y automáticamente la librería se creará con la configuración y ruta establecidas.



## J. Lista de Tokens utilizados

Existe una lista de los tokens utilizados, que está definida en el código fuente de la siguiente manera:

**char *token_string[] = {**"nulo", "comentario", "numeroReal", "numeroEntero", "ident", "mas", "menos", "por", "barra", "llaveI", "llaveF", "parentI", "parentF", "corcheteI", "corcheteF", "punto", "coma", "puntoycoma", "asignacion", "mei", "mai", "myr", "mnr", "igl", "nig", "negacion", "ytok", "otok", "referencia", "enteroTok", "byteTok", "realTok", "vacioTok", "booleanoTok", "cadenaTok", "caracterTok", "objetoTok", "archivoTok", "siTok", "sinoTok", "probarTok", "casoTok", "defectoTok", "romperTok", "mientrasTok", "paraTok", "hacerTok", "paracadaTok", "sistemaTok", "obtenerEnteroTok", "obtenerRealTok", "obtenerCadenaTok", "obtenerCaracterTok", "mostrarTok", "publicaTok", "privadaTok", "retornarTok", "arregloTok", "agregarTok", "obtenerTok", "cuantosTok", "quitarTok", "abrirTok", "leerLineaTok", "volcadoTok", "cerrarTok", "concatenarTok", "parteEnteraTok", "compararTok", "mayorTok", "menorTok", "esParTok", "decimalBinTok", "potenciaTok", "absolutoTok", "moduloTok", "longitudCadenaTok", "claseTok", "incluirTok", "obtenerBooleanoTok", "falsoTok", "verdaderoTok", "datoCadena", "datoCaracter", "localTok", "principalTok", "dosPuntos"**};**

## K. Diagramas de Sintaxis

Diagrama general de un programa escrito en lenguaje Armus:

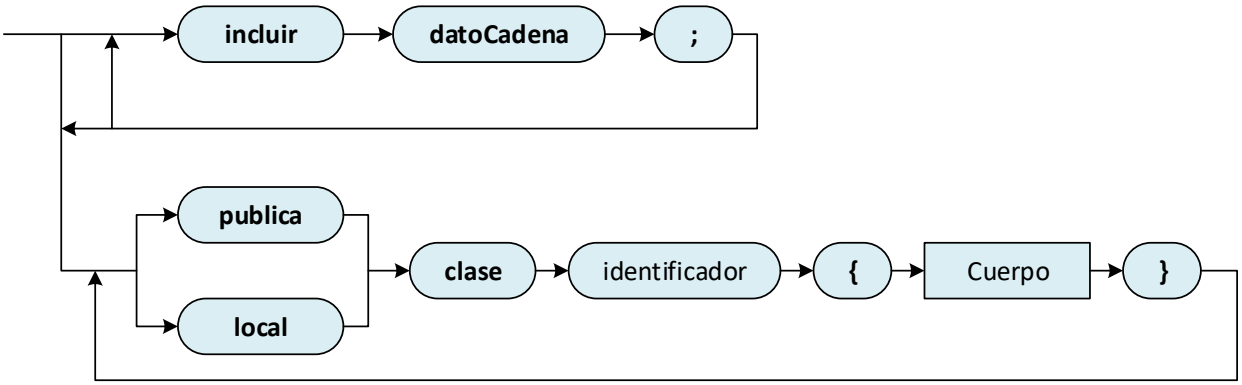



**Figura 01**. Diagrama de sintaxis de la estructura general de un algoritmo.

Las clases **públicas** son utilizadas por cualquiera, las clases **locales** solo pueden ser accedidas desde el mismo archivo

Pueden existir múltiples clases en el mismo archivo del programa

Cuerpo:

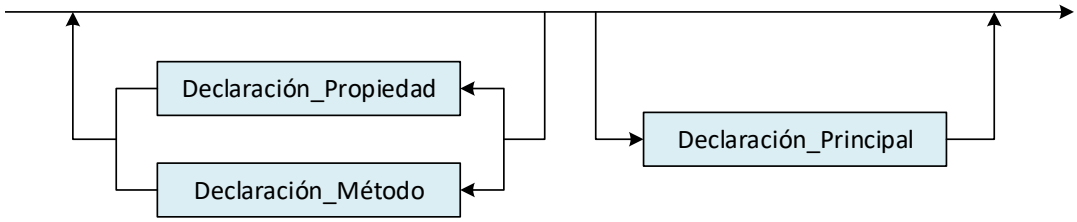



**Figura 02**. Diagrama de sintaxis para la estructura interna de una clase.

Declaración_Propiedad:

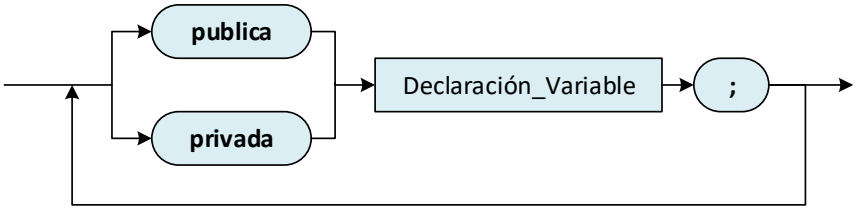



**Figura 03**. Diagrama de sintaxis para la declaración de una propiedad de la clase.

Declaración_Variable:

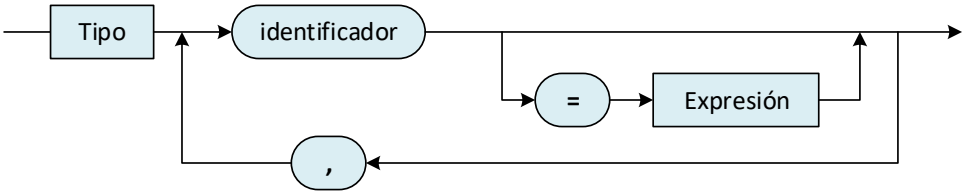



**Figura 04**. Diagrama de sintaxis para la declaración de una variable.

Expresión:

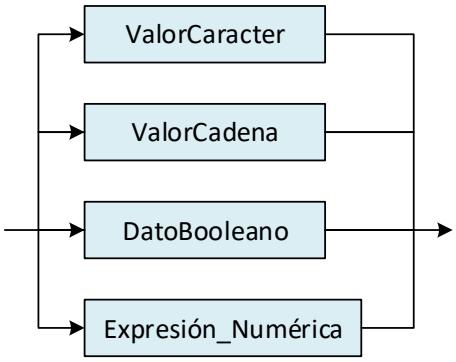



**Figura 05**. Diagrama de sintaxis que muestra las diferentes expresiones que se pueden utilizar. ValorCaracter y ValorCadena corresponden a cada una de las posibilidades que pueden generar un valor de dicho tipo dentro de una expresión.

Tipo:

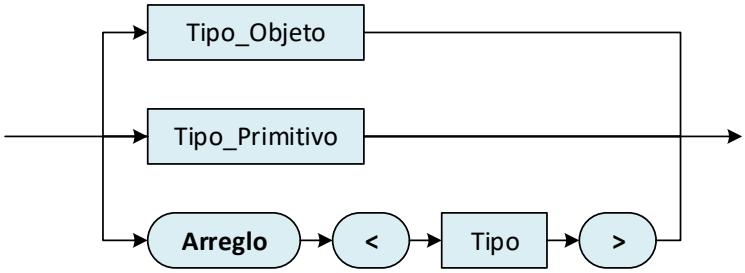



**Figura 06**. Diagrama de sintaxis que muestra las tres clasificaciones de tipo que pueden existir.

Tipo_Objeto:

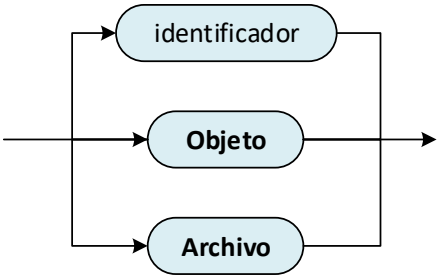



**Figura 07**. Diagrama de sintaxis que muestra los tipos objetos en el lenguaje Armus, donde identificador es una clase creada por el usuario.

Tipo_Primitivo:

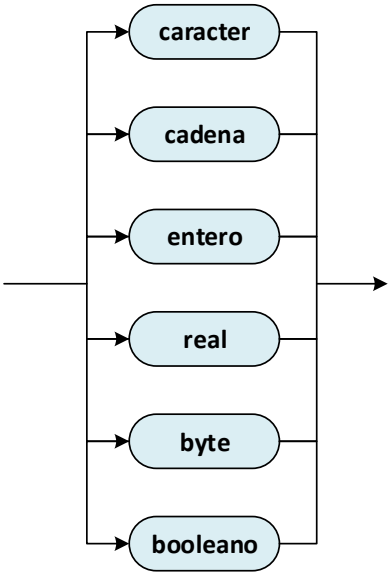



**Figura 08**. Diagrama de sintaxis que muestra los tipos primitivos (no son objetos) en el lenguaje Armus.

DatoBooleano:


**Figura 09**. Diagrama de sintaxis para el tipo de dato booleano, solo tiene las palabras reservadas verdadero y falso.

Declaración_Método:

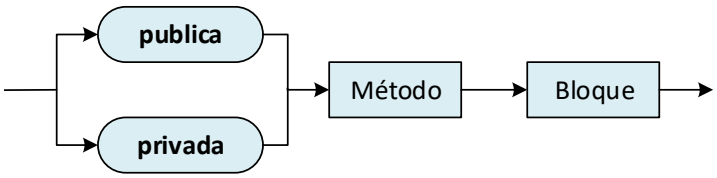



**Figura 10**. Diagrama de sintaxis que muestra la forma de declaración de un método.

Método:

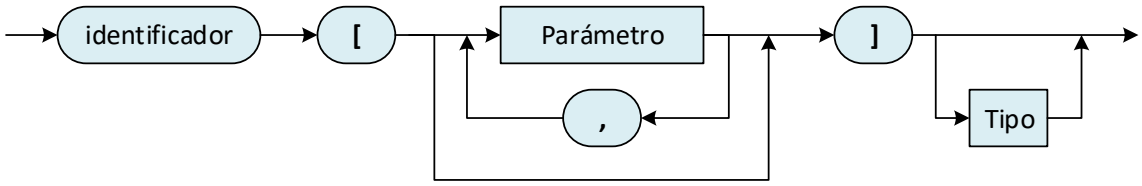



**Figura 11.** Diagrama de sintaxis que muestra la estructura sintáctica de un método, donde identificador es el nombre del método, además de que puede o no tener parámetros de entrada y si posee tipo de valor de retorno entonces es una función, sino es un procedimiento.

Parámetro:

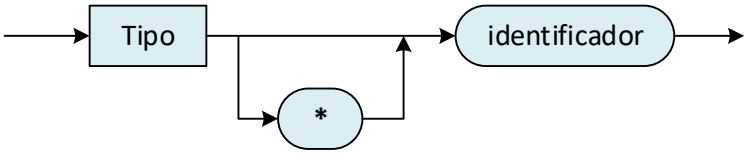



**Figura 12**. Diagrama de sintaxis para definir un parámetro en el lenguaje Armus, el * se utiliza opcionalmente para indicar que el parámetro será usado por referencia.

Declaración_Principal:

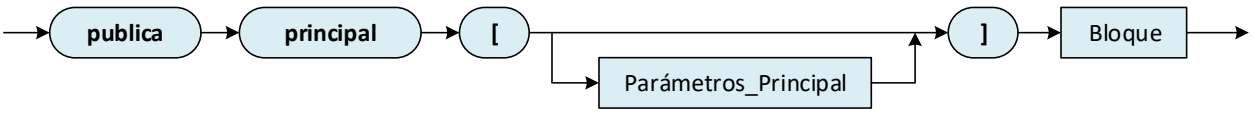



**Figura 13**. Diagrama de sintaxis para definir la estructura sintáctica del método principal (main).

Parámetros_Principal:

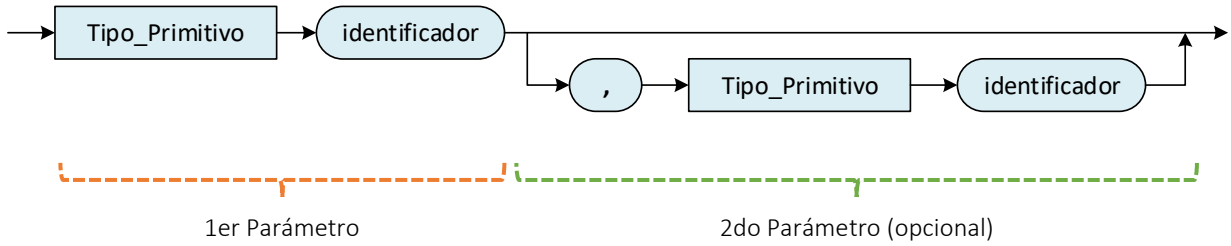



**Figura 14**. Diagrama de sintaxis de la utilización de parámetros en el método principal, tiene un máximo de dos parámetros.

Llamada_Método:

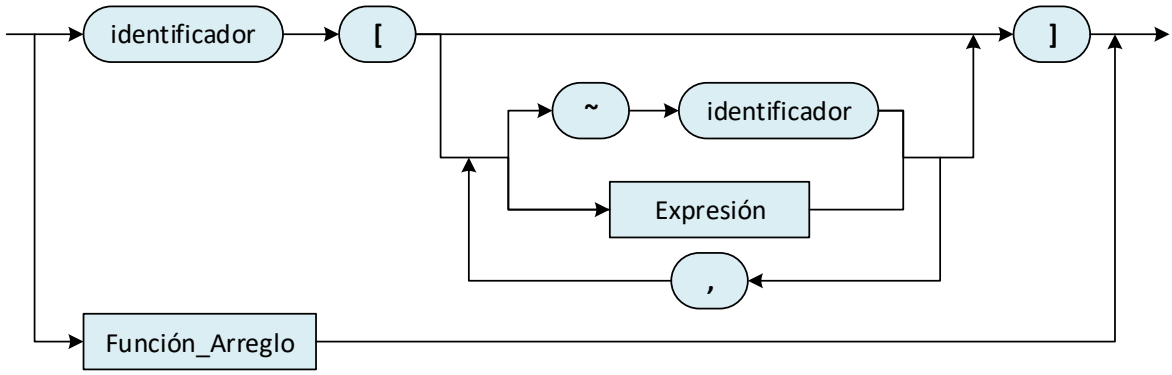



**Figura 15**. Diagrama de sintaxis de la estructura de la llamada a un método desde un objeto clase. El símbolo **~** antes de un identificador hace referencia a que este será enviado por referencia.

Expresión_Numérica:

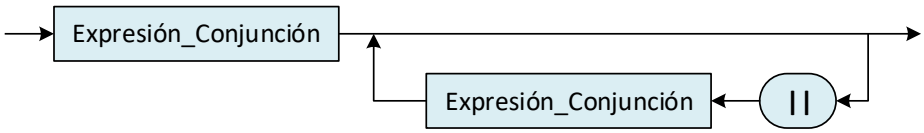



**Figura 16**. Diagrama de sintaxis para una expresión numérica, la doble barra (||) es equivalente a un OR.

Expresión_Conjunción:

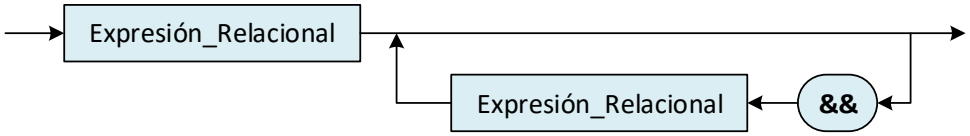



**Figura 17**. Diagrama de sintaxis para una expresión de conjunción, la doble ampersand (&&) es equivalente a un AND (lógico).

Expresión_Relacional:

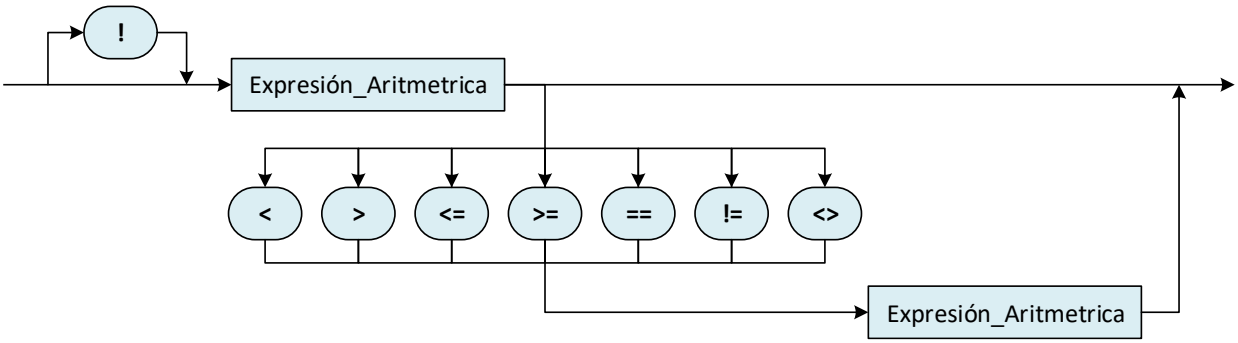



**Figura 18**. Diagrama de sintaxis para una expresión relacional, en la cual se tiene los operadores lógicos que soporta el lenguaje Armus, incluida la negación.

Expresión_Aritmetrica:

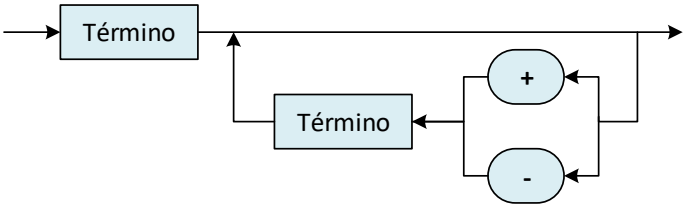



**Figura 19**. Diagrama de sintaxis de expresión aritmética, donde puede haber uno o más términos sumados o restados.

Término:

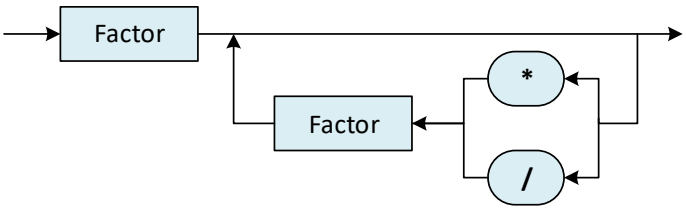



**Figura 20**. Diagrama de sintaxis de termino, la cual tiene uno o más factores multiplicados o divididos.

Factor:

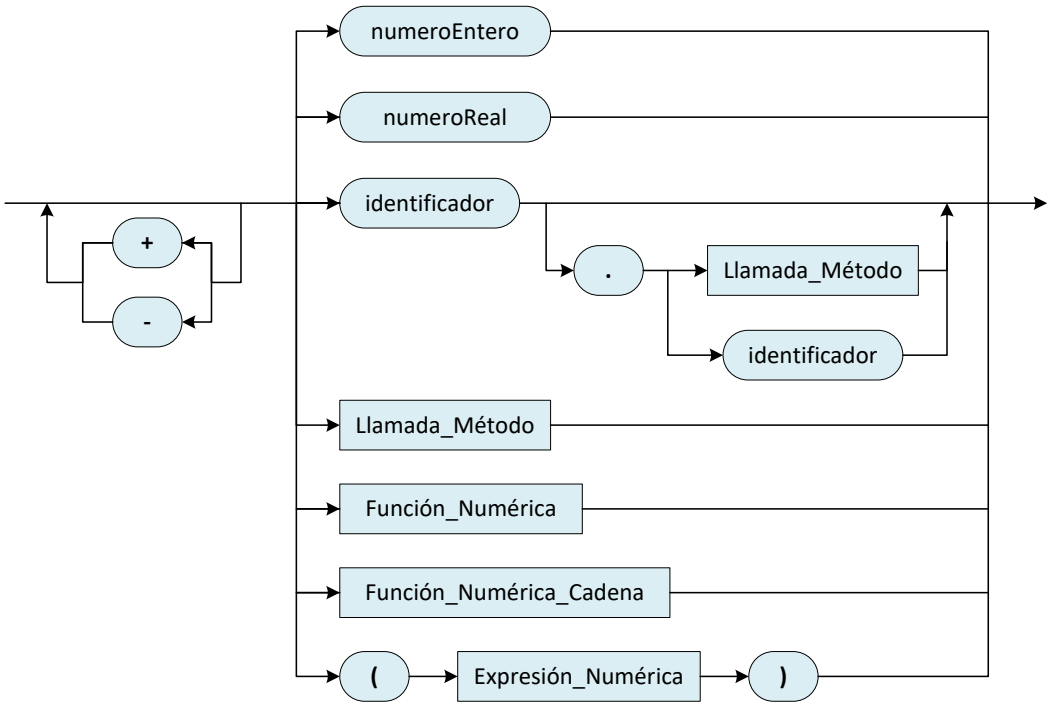



**Figura 21**. Diagrama de sintaxis de factor, el cual puede tener múltiples posibilidades, numero entero, número real, variables, llamadas a métodos o atributos, funciones que devuelven valores o una expresión numérica.

Función_Numérica:

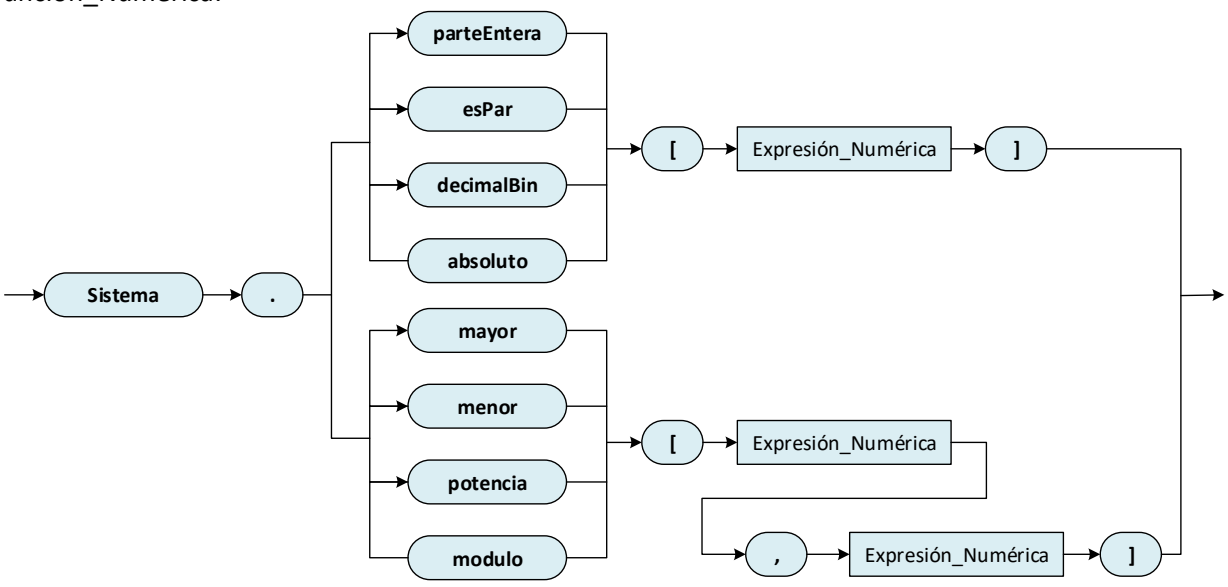



**Figura 22**. Diagrama de sintaxis para función numérica, la cual a partir del objeto Sistema se llaman las funciones predefinidas en el lenguaje.

Función_Numérica_Cadena:

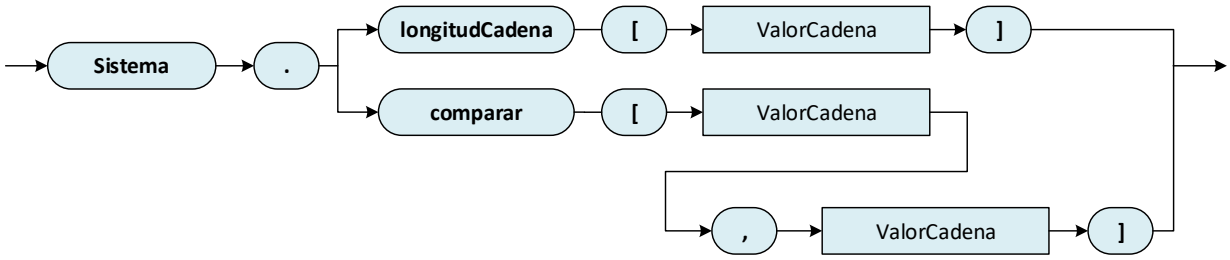



**Figura 23**. Diagrama de sintaxis para las funciones que devuelven un dato numérico recibiendo como parámetros valores de cadena.

ValorCadena:

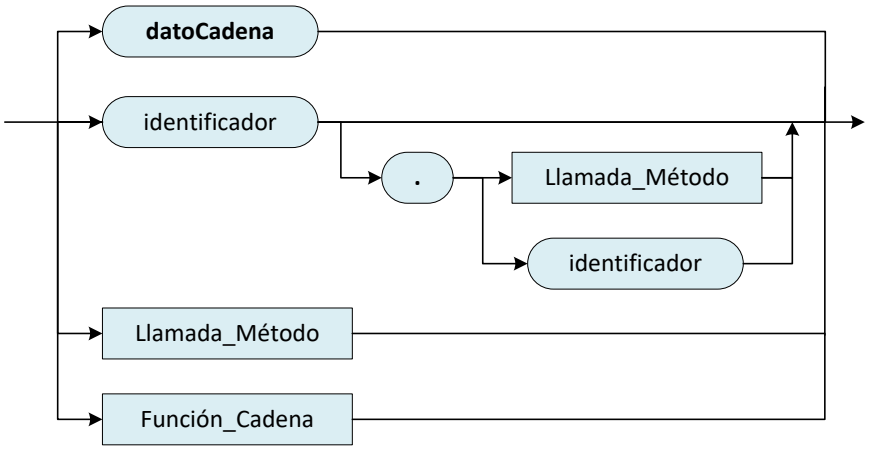



**Figura 24**. Diagrama de sintaxis para los posibles valores que pueden generar una cadena de caracteres.

ValorCaracter:

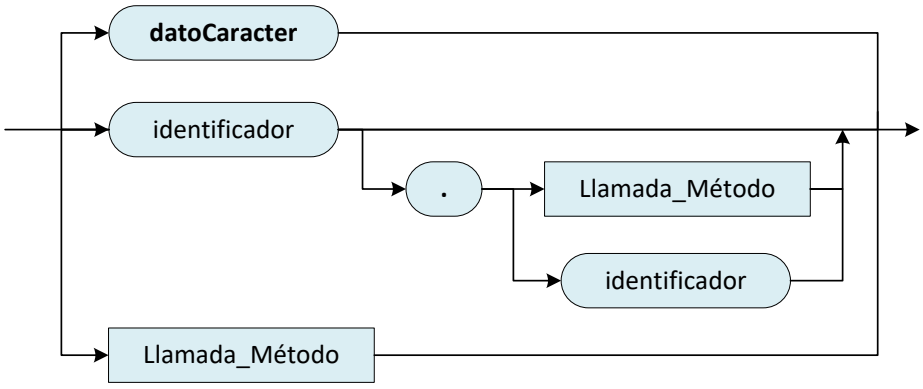



**Figura 25**. Diagrama de sintaxis para los posibles valores que puede generar un carácter.

Función_Cadena:

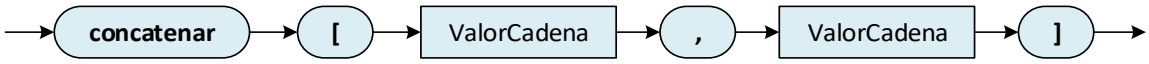



**Figura 26**. Diagrama de sintaxis de la única función que retorna una cadena de caracteres, la cual concatena dos valores de cadena en uno solo.

Bloque:

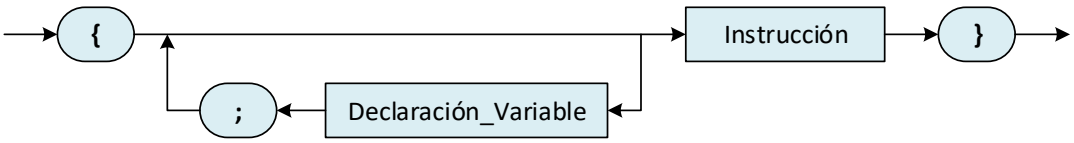



**Figura 27**. Diagrama de sintaxis de bloque, el cual declara o no variables y su contenido tiene Instrucción.

Función_Arreglo:

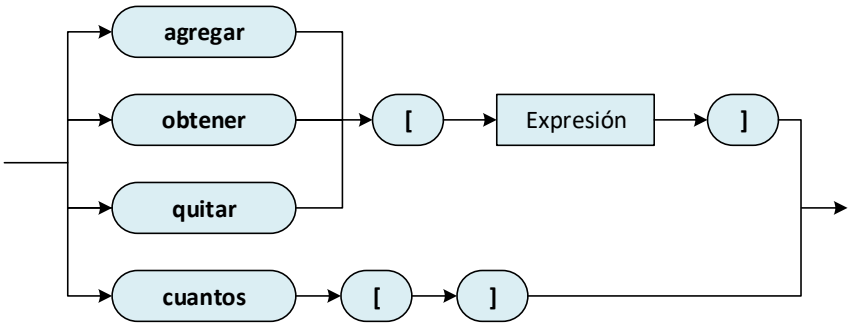



**Figura 28**. Diagrama de sintaxis de las funciones que se manejan en un arreglo, en total son cuatro.

Función_Archivo:

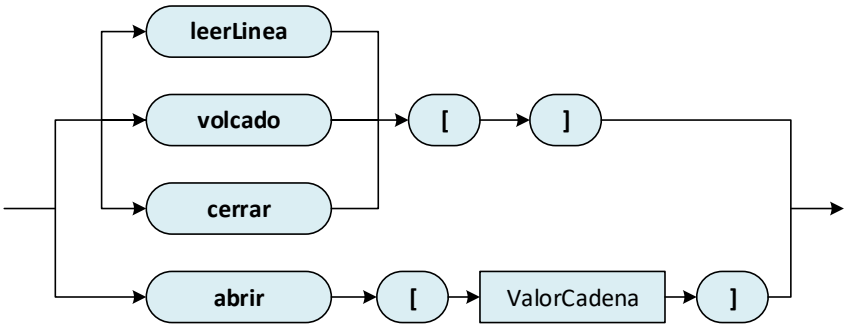



**Figura 29**. Diagrama de sintaxis de las funciones que utilizan para el manejo del objeto Archivo.

Instrucción:

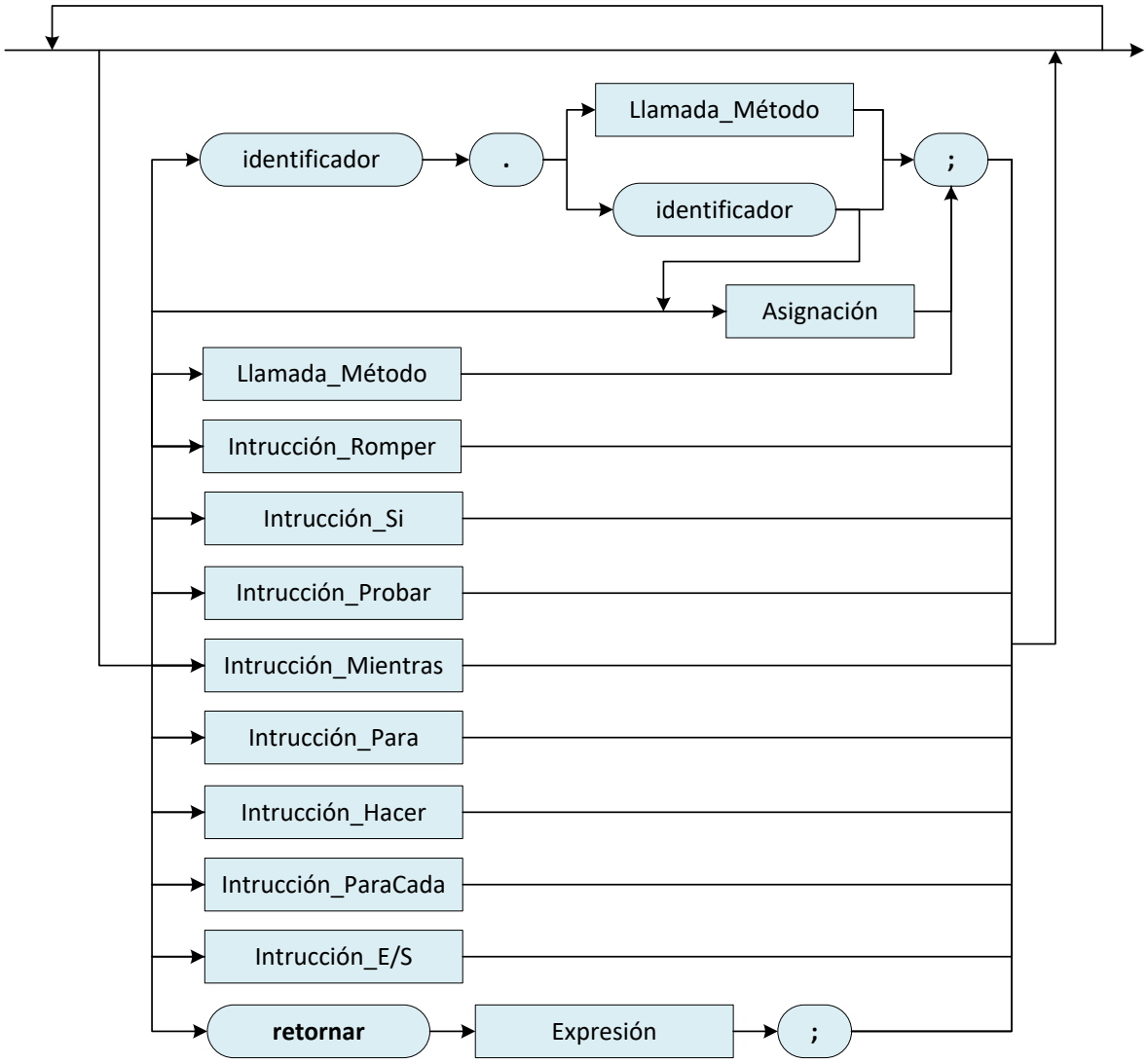



**Figura 30**. Diagrama de sintaxis para instrucción, las cuales pueden ser llamadas una o más veces.

Instrucción_E/S:

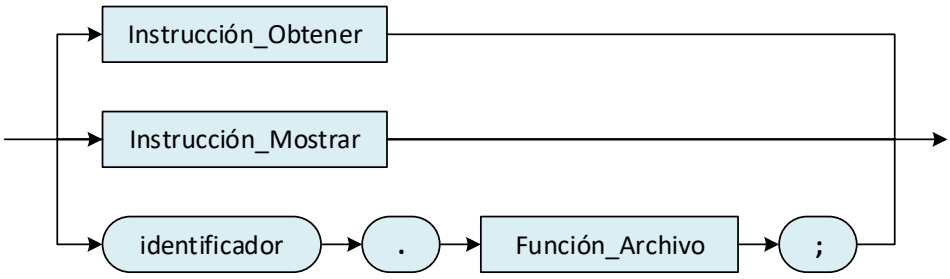



**Figura 31**. Diagrama de sintaxis para las instrucciones de entrada y salida de datos.

Asignación:

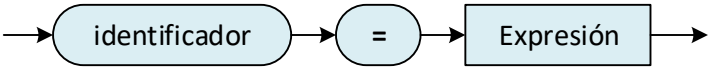



**Figura 32**. Diagrama de sintaxis que muestra la estructura de una asignación en el lenguaje.

Instrucción_Si:

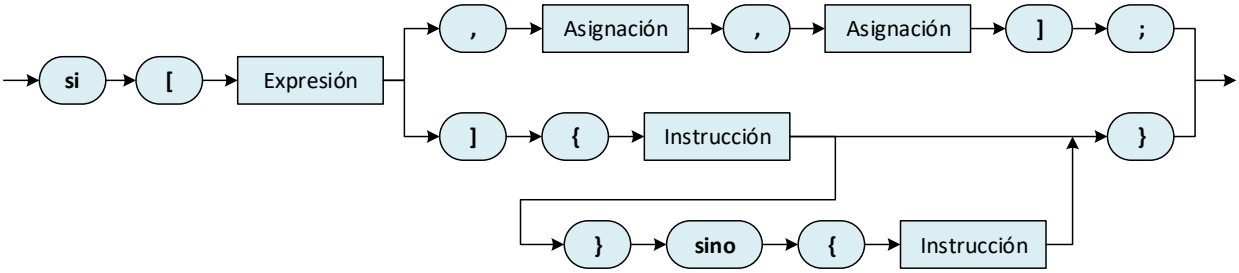



**Figura 33**. Diagrama de sintaxis para la instrucción si (if), la cual puede ser de dos formas.

Instrucción_Mientras:

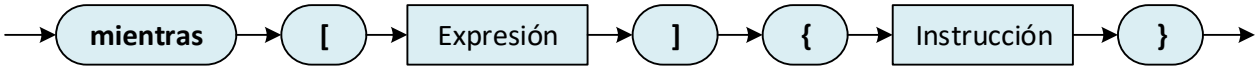



**Figura 34**. Diagrama de sintaxis para la instrucción mientras (while).

Instrucción_Probar:

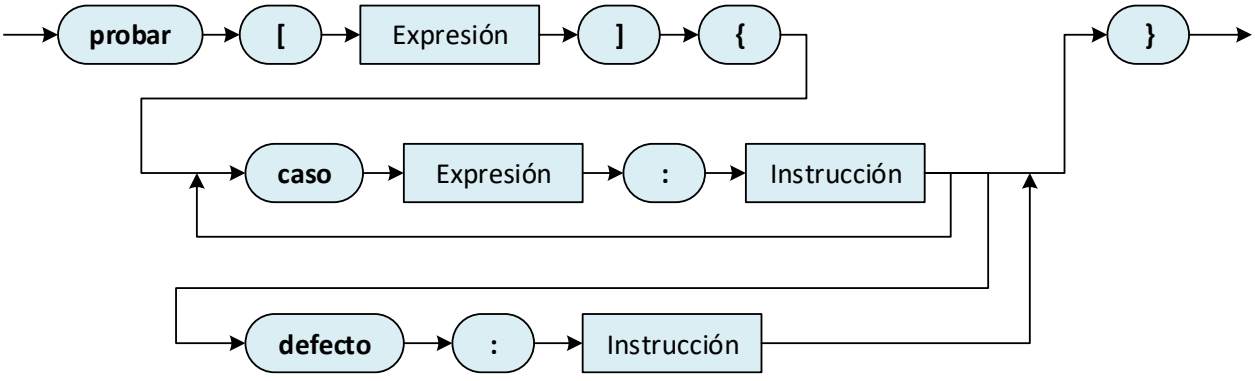



**Figura 35**. Diagrama de sintaxis para la instrucción probar (switch); tiene una opción por defecto que es opcional, pero solo puede ser utilizada una sola vez en dicha instrucción.

Instrucción_Romper:

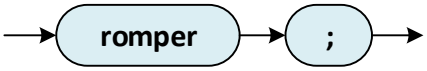



**Figura 36**. Diagrama de sintaxis para la instrucción romper, la cual simplemente debe de tener la palabra reservada romper (break), seguido de un punto y coma.

Instrucción_Para:

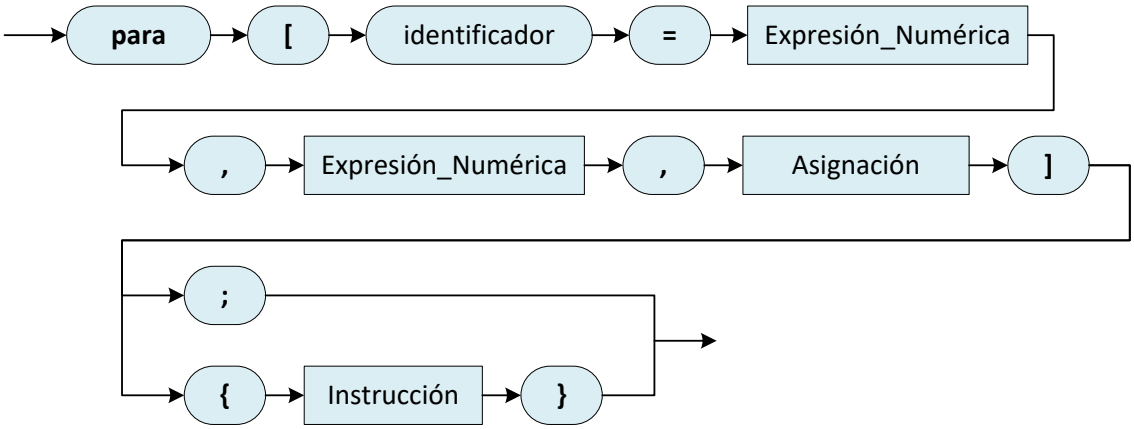



**Figura 37**. Diagrama de sintaxis para la instrucción para (for), la cual puede tener instrucciones o tener un punto y coma al final.

Instrucción_Hacer:

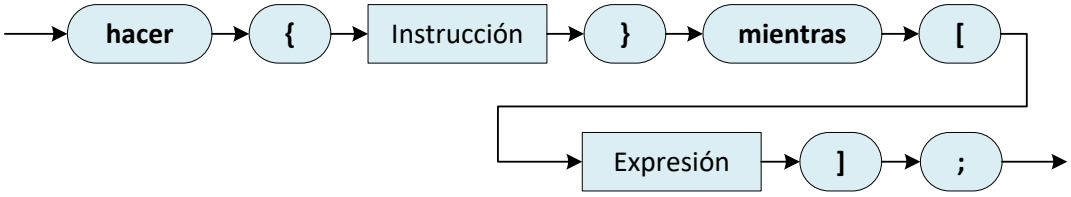



**Figura 38**. Diagrama de sintaxis para la instrucción hacer-mientras (do-while).

Instrucción_ParaCada:

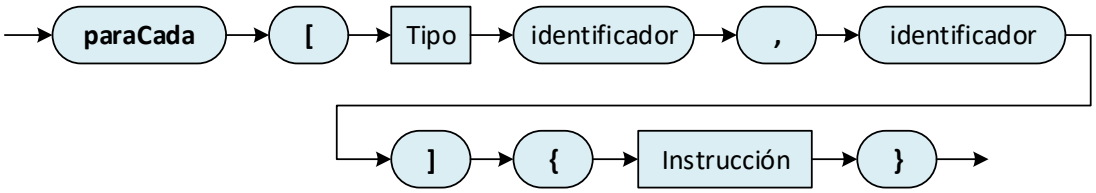



**Figura 39**. Diagrama de sintaxis para la instrucción paraCada (for-each).

Ejemplo:

```
Arreglo<cadena> Lista;
Lista.agregar[“lunes”];
Lista.agregar[“martes”];
paraCada[cadena dia, Lista]{
      Sistema.mostrar[dia];
}
```

Instrucción_Obtener:

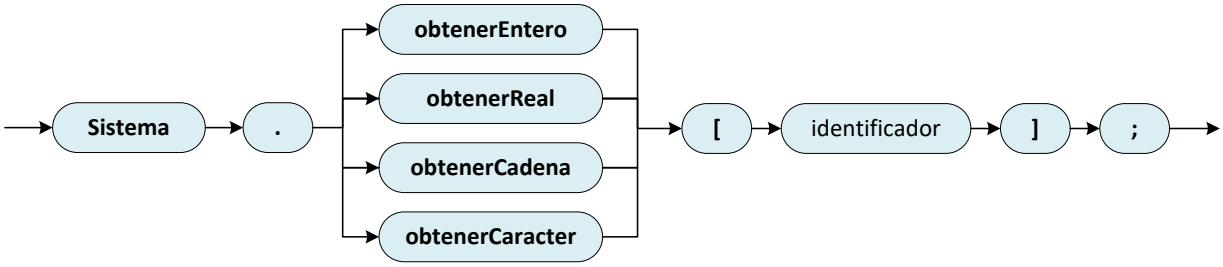



**Figura 40**. Diagrama de sintaxis para las instrucciones obtener del objeto sistema.

Instrucción_Mostrar:

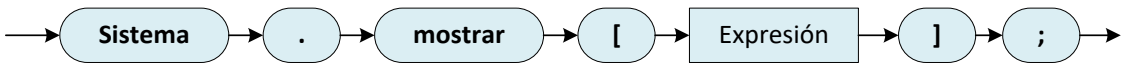



**Figura 41**. Diagrama de sintaxis para la instrucción mostrar.

## L. Gramática Formal

**Programa** → **IncluirArchivo DefinicionClase**

**IncluirArchivo** → ε **|** incluirTok datoCadena puntoycoma **IncluirArchivo**

**DefinicionClase** → **IdentClass** claseTok ident llaveI **Cuerpo** llaveF **MasDefinicionClase**

**IdentClass** → publicaTok **|** localTok

**MasDefinicionClase** → ε **|** **DefinicionClase**

**Cuerpo** → **Declar MasDeclar Declarar**

**Declar** → ε **|** **DedProp** **|** **DedMet**

**MasDeclar** → ε **|** **Declar MasDeclar**

**Declarar** → ε **|** **DeclararPrincipal**

**DedProp** → **IdentProp Dedvar** puntoycoma

**IdentProp** → publicaTok **|** privadaTok

**DedMet** → **IdentProp Metodo Bloque**

**Dedvar** → **Tipo Dedvariable MasDedvar**

**Dedvariable** → ident **Dedvari**

**Dedvari** → ε **|** asignacion **Expresion**

**MasDedvar** → ε **|** coma **Dedvariable MasDedvar**

**Tipo** → **TipoObj** **|** **TipoPrim** **|** arregloTok mnr **Tipo** myr

**TipoObj** → ident **|** objetoTok **|** archivoTok

**TipoPrim** → caracterTok **|** cadenaTok **|** enteroTok **|** realTok **|** byteTok **|** boolenoTok

**Expresion** → **ValorCaracter** **|** **ValorCadena** **|** **DatoBool** | **ExpresNum**

**IdentLlamada** → ident **|** ident punto **LlamadaMet** **|** ident punto ident

**DatoBool** → verdaderoTok **|** falsoTok

**Metodo** → ident corcheteI corcheteF **Tipo** **|** ident corcheteI **ListParam** corcheteF

**ListParam** → **Parametro MasListParam**

**MasListParam** → ε **|** coma **ListParam**

**Parametro** → **Tipo Param** ident

**Param** → ε **|** por

**DeclararPrincipal** → publicaTok principalTok corcheteI **LlevaParamPrin** corcheteF **Bloque**

**LlevaParamPrin** → ε **|** **ParamPrin**

**ParamPrin** → **TipoPrimIdent ParamPrinUno**

**ParamPrinUno** → ε **|** coma **TipoPrimIdent**

**TipoPrimIdent** → **TipoPrim** ident

**LlamadaMet** → **FuncionArreglo** **|** ident corcheteI **LlamadaMetUno** corcheteF

**LlamadaMetUno** → ε **|** **IdentExpres MasIdentExpres**

**IdentExpres** → referencia ident **|** **Expresion**

**MasIdentExpres** → ε **|** coma **IdentExpres**

**ExpresNum** → **ExpresCon MasExpresNum**

**MasExpresNum** → ε **|** otok **ExpresCon**

**ExpresCon** → **ExpresRel MasExpresCon**

**MasExpresCon** → ε **|** ytok **ExpresRel**

**ExpresRel** → **Negacion ExpresAri XexpresRel**

**Negacion** → ε **|** negacion

**XexpresRel** → ε **|** **Simbolo ExpresAri**

**Simbolo** → mnr **|** myr **|** mei **|** mai **|** igl **|** nig

**ExpresAri** → **Termino MasExpresAri**

**Signo** → mas **|** menos

**MasExpresAri** → ε **|** **Signo Termino**

**Termino** → **Factor MasTermino**

**MasTermino** → ε **|** **Oper Factor**

**Oper** → por **|** barra

**Factor** → **Sing Factores**

**Sing** → ε **|** **Signo**

**Factores** → numeroEntero **|** numeroReal **|** **IdentLlamada** **|** **LlamadaMet** **|** **FuncNum** **|** **FuncNumCad** **|** parentI **ExpresNum** parentF

**FuncNum** → sistemaTok punto **Fnum** parentI **ExpresNum MasExpressNum** parentF

**Fnum** → **ParaNum** **|** **ParaSin**

**ParaNum** → parteEnteraTok **|** esParTok **|** decimalBinTok **|** absolutoTok

**MasExpressNum** → ε **|** coma **ExpresNum**

**ParaSin** → mayorTok **|** menorTok **|** potenciaTok **|** moduloTok

**FuncNumCad** → sistemaTok punto **Conlog** corcheteI **ValorCadena MasFuncNumCad** corcheteF

**MasFuncNumCad** → ε **|** coma **ValorCadena**

**Conlong** → compararTok **|** longitudCadenaTok

**ValorCaracter** → datoCaracter **|** **IdentLlamada** **|** **LlamadaMet**

**ValorCadena** → datoCadena **|** **IdentLlamada** **|** **LlamadaMet** **|** **FuncCad**

**FuncCad** → concatenarTok corcheteI **ValorCadena** coma **ValorCadena** corcheteF

**Bloque** → llaveI **MasBloque Instruccion** llaveF

**MasBloque** → ε **|** **Dedvar** puntoycoma **MasBloque**

**FuncionArreglo** → **Funcarr** corcheteI **Expresion** corcheteF **|** cuantosTok corcheteI corcheteF

**Funcarr** → agregarTok **|** obtenerTok **|** quitarTok

**FuncionArc** → abrirTok corcheteI **ValorCadena** corcheteF **|** **Funcarc** corcheteI corcheteF

**Funcarc** → leerLineaTok **|** volcadoTok **|** cerrarTok

**Instruccion** → ε **|** **MasInstruccion Instruccion**

**MasInstruccion** → **IdentLlamada** puntoycoma **|** **LlamadaMet** puntoycoma **|** **Asignacion** puntoycoma **|** romperTok puntoycoma **|** **InstSi** **|** **InstProbar** **|** **InstMientras** **|** **InstPara** **|** **InstHacer** **|** **InstParaCad** **|** **InstES** **|** retornarTok **Expresion** puntoycoma

**InstES** → **InstObtener** **|** **InstMostrar** **|** ident punto **FuncionArc** puntoycoma

**Asignacion** → ident asignacion **Expresion**

**InstSi** → siTok corcheteI **Expresion** **InstSiUno**

**InstSiUno** → coma **Asignacion** coma **Asignacion** corcheteF puntoycoma **|** corcheteF llaveI **Instruccion** **InstSiDos** llaveF

**InstSiDos** → ε **|** llaveF sinoTok llaveI **Instruccion**

**InstMientras** → mientrasTok corcheteI **Expresion** corcheteF llaveI **Instruccion** llaveF

**InstProbar** → probarTok corcheteI **Expresion** corcheteF llaveI casoTok **Expresion** dospuntos **Instruccion** **MasInstProbar** **InstProbarUno** llaveF

**InstProbarUno** → ε **|** defectoTok dospuntos **Instruccion**

**MasInstProbar** → ε **|** casoTok **Expresion** dospuntos **Instruccion**

**InstPara** → paraTok corcheteI ident asignacion **ExpresNum** coma **ExpresNum** coma **ExpresNum** corcheteF **InstParaUno**

**InstParaUno** → puntoycoma **|** llaveI **Instruccion** llaveF

**InstHacer** → hacerTok llaveI **Instruccion** llaveF mientrasTok corcheteI **ExpresNum** corcheteF puntoycoma

**InstParaCad** → paracadaTok corcheteI **Tipo** ident coma ident corcheteF llaveI **Instruccion** llaveF

**InstObtener** → sistemaTok punto **Obtener** corcheteI ident corcheteF puntoycoma

**Obtener** → obtenerEnteroTok **|** obtenerRealTok **|** obtenerCadenaTok **|** obtenerCaracterTok

**InstMostrar** → sistemaTok punto mostrarTok corcheteI **Expresion** **InstMostrarUno** corcheteF puntoycoma

**InstMostrarUno** → ε **|** coma **Expresion**

## **M**. Aplicación de la Regla de Primero

Prim(**Programa**) = {incluirTok} U {}

Prim(**IncluirArchivo**) = {incluirTok} U {}

Prim(**DefinicionClase**) = {publicaTok, localTok} U {}

Prim(**IdentClass**) = {publicaTok, localTok} U {}

Prim(**MasDefinicionClase**) = {publicaTok, localTok} U {}

Prim(**Cuerpo**) = Prim(Declar) U Prim(MasDeclar) U Prim(Declarar) = {publicaTok, privadaTok} U {}

Prim(**Declar**) = Prim(DedProp) U Prim(DedMet) U {} = {publicaTok, privadaTok} U {}

Prim(**MasDeclar**) = {publicaTok, privadaTok} U {}

Prim(**Declarar**) = Prim(DedPrim) U {} = {publicaTok} U {}

Prim(**DedProp**) = {publicaTok, privadaTok}

Prim(**IdentProp**) = {publicaTok, privadaTok}

Prim(**DetMet**) = {publicaTok, privadaTok}

Prim(**Dedvar**) = {ident, objetoTok, archivoTok, caracterTok, cadenaTok, enteroTok, realTok, byteTok, booleanoTok, arregloTok}

Prim(**Dedvariable**) = {ident}

Prim(**Dedvari**) = {asignacion} U {}

Prim(**MasDedvar**) = {coma} U {}

Prim(**Tipo**) = Prim(TipoObj) U Prim(TipoPrim) U {arregloTok} = {ident, objetoTok, archivoTok, caracterTok, cadenaTok, enteroTok, realTok, byteTok, booleanoTok, arregloTok}

Prim(**TipoObj**) = {ident, objetoTok, archivoTok}

Prim(**TipoPrim**) = {caracterTok, cadenaTok, enteroTok, realTok, byteTok, booleanoTok}

Prim(**Expresion**) = Prim(ValorCaracter) U Prim(ValorCadena) U Prim(DatoBool) U Prim(ExpresNum) = {datoCaracter, ident, agregarTok, obtenerTok, quitarTok, cuantosTok, datoCadena, concatenarTok, verdaderoTok, falsoTok, negacion, mas, menos, numeroEntero, numeroReal, parentI, sistemaTok} U {}

Prim(**IdentLlamada**) = {ident}

Prim(**DatoBool**) = {verdaderoTok, falsoTok}

Prim(**Metodo**) = {ident}

Prim(**ListParam**) = {ident, objetoTok, archivoTok, caracterTok, cadenaTok, enteroTok, realTok, byteTok, booleanoTok, arregloTok}

Prim(**MasListParam**) = {coma} U {}

Prim(**Parametro**) = {ident, objetoTok, archivoTok, caracterTok, cadenaTok, enteroTok, realTok, byteTok, booleanoTok, arregloTok}

Prim(**Param**) = {por} U {}

Prim(**DeclararPrincipal**) = {publicaTok}

Prim(**LlevaParamPrin**) = Prim(ParamPrin) U {} = {caracterTok, cadenaTok, enteroTok, realTok, byteTok, booleanoTok} U {}

Prim(**ParamPrin**) = Prim(TipoPrimIdent) = Prim(TipoPrim) = {caracterTok, cadenaTok, enteroTok, realTok, byteTok, booleanoTok}

Prim(**ParamPrinUno**) = {coma} U {}

Prim(**TipoPrimIdent**) = Prim(TipoPrim) = {caracterTok, cadenaTok, enteroTok, realTok, byteTok, booleanoTok}

Prim(**LLamadaMet**) = Prim(FuncionArreglo) U {ident} = {ident, agregarTok, obtenerTok, quitarTok, cuantosTok}

Prim(**LlamadaMetUno**) = Prim(IdentExpres) U {} = {referencia, datoCaracter, ident, datoCadena, concatenarTok, verdaderoTok, falsoTok, negación} U {}

Prim(**IdentExpres**) = {referencia} U Prim(Expresion) = {referencia, datoCaracter, ident, datoCadena, concatenarTok, verdaderoTok, falsoTok, negacion} U {}

Prim(**MasIdentExpres**) = {coma} U {}

Prim(**ExpresNum**) = Prim(ExpresCon) = Prim(ExpresRel) = Prim(Negacion) = {negacion} U Prim(ExpresAri)
= {negacion} U Prim(Termino)
= {negacion} U Prim(Factor)
= {negacion, mas, menos, numeroEntero, numeroReal, parentI, ident} U Prim(LlamadaMet) U Prim(FuncNum) U Prim(FuncNumCad)
= {negacion, mas, menos, numeroEntero, numeroReal, parentI, ident, agregarTok, obtenerTok, quitarTok, cuantosTok, sistemaTok}

Prim(**MasExpresNum**) = {oTok} U {}

Prim(**ExpresCon**) = Prim(ExpresRel) = Prim(Negacion) = {negacion, mas, menos, numeroEntero, numeroReal, parentI, ident, agregarTok, obtenerTok, quitarTok, cuantosTok, sistemaTok} U {}

Prim(**MasExpresCon**) = {yTok} U {}

Prim(**ExpresRel**) = Prim(Negacion) = {negacion, mas, menos, numeroEntero, numeroReal, parentI, ident, agregarTok, obtenerTok, quitarTok, cuantosTok, sistemaTok} U {}

Prim(**Negacion**) = {negacion} U {}

Prim(**XexpresRel**) = Prim(Simbolo) U {} = {mnr, myr, mei, mai, igl, nig} U {}

Prim(**Simbolo**) = {mnr, myr, mei, mai, igl, nig}

Prim(**ExpresAri**) = Prim(Termino) = Prim(Factor) = Prim(Sing) = Prim(Signo) U {} = {mas, menos, numeroEntero, numeroReal, parentI, ident, agregarTok, obtenerTok, quitarTok, cuantosTok, sistemaTok} U {}

Prim(**Signo**) = {mas, menos}

Prim(**MasExpresAri**) = Prim(Signo) U {} = {mas, menos} U {}

Prim(**Termino**) = Prim(Factor) U Prim(Sing) = Prim(Factor) U Prim(Signo) U {} = {mas, menos, numeroEntero, numeroReal, parentI, ident, agregarTok, obtenerTok, quitarTok, cuantosTok, sistemaTok} U {}

Prim(**MasTermino**) = Prim(Oper) U {} = {por, barra} U {}

Prim(**Oper**) = {por, barra}

Prim(**Factor**) = Prim(Sing) U Prim(Factores) U {}
= Prim(Signo) U Prim(Factores) U {}
= {mas, menos, numeroEntero, numeroReal, parentI, ident, agregarTok, obtenerTok, quitarTok, cuantosTok, sistemaTok} U {}

Prim(**Sing**) = Prim(Signo) U {} = {mas, menos} U {}

Prim(**Factores**) = {numeroEntero, numeroReal, parentI} U Prim(IdentLlamada) U Prim(LlamadaMet) U Prim(FuncNum) U Prim(FuncNumCad) = { numeroEntero, numeroReal, parentI, ident, agregarTok, obtenerTok, quitarTok, cuantosTok, sistemaTok}

Prim(**FuncNum**) = {sistemaTok}

Prim(**Fnum**) = Prim(ParaNum) U Prim(ParaSin) = {parteEnteraTok, esParTok, decimalBinTok, absolutoTok, mayorTok, menorTok, potenciaTok, moduloTok}

Prim(**ParaNum**) = {parteEnteraTok, esParTok, decimalBinTok, absolutoTok}

Prim(**MasExpressNum**) = {coma} U {}

Prim(**ParaSin**) = {mayorTok, menorTok, potenciaTok, moduloTok}

Prim(**FuncNumCad**) = {sistemaTok}

Prim(**MasFuncNumCad**) = {coma} U {}

Prim(**Conlong**) = {compararTok, longitudCadenaTok}

Prim(**ValorCaracter**) = {datoCaracter} U Prim(IdentLlamada) U Prim(LLamadaMet) = {datoCaracter, ident, agregarTok, obtenerTok, quitarTok, cuantosTok}

Prim(**ValorCadena**) = {datoCadena} U Prim(IdentLlamada) U Prim(LLamadaMet) U Prim(FuncCad) = {datoCadena, ident, agregarTok, obtenerTok, quitarTok, cuantosTok, concatenarTok}

Prim(**FuncCad**) = {concatenarTok}

Prim(**Bloque**) = {llaveI}

Prim(**MasBloque**) = Prim(Dedvar) U {} = {ident, objetoTok, archivoTok, caracterTok, cadenaTok, enteroTok, realTok, byteTok, booleanoTok, arregloTok} U {}

Prim(**FuncionArreglo**) = Prim(Funcarr) U {cuantosTok} = {agregarTok, obtenerTok, quitarTok, cuantosTok}

Prim(**Funcarr**) = {agregarTok, obtenerTok, quitarTok}

Prim(**FuncionArc**) = {abrirTok} U Prim(Funcarc) = {abrirTok, leerLineaTok, volcadoTok, cerrarTok}

Prim(**Funcarc**) = {leerLineaTok, volcadoTok, cerrarTok}

Prim(**Instruccion**) = Prim(MasInstruccion) U {} = {romperTok, retornarTok, ident, agregarTok, obtenerTok, quitarTok, cuantosTok, siTok, probarTok, mientrasTok, paraTok, hacerTok, paracadaTok, sistemaTok} U {}

Prim(**MasInstruccion**) = {romperTok, retornarTok} U Prim(IdentLlamada) U Prim(LLamadaMet) U Prim(Asignacion) U Prim(InstSi) U Prim(InstProbar) U Prim(InstMientras) U Prim(InstPara) U Prim(InstHacer) U Prim(InstParaCad) U Prim(InstES) = {romperTok, retornarTok, ident, agregarTok, obtenerTok, quitarTok, cuantosTok, siTok, probarTok, mientrasTok, paraTok, hacerTok, paracadaTok, sistemaTok}

Prim(**InstES**) = Prim(InstObtener) U Prim(InstMostrar) U {ident} = {sistemaTok, ident}

Prim(**Asignacion**) = {ident}

Prim(**InstSi**) = {siTok}

Prim(**InstSiUno**) = {coma, corcheteF}

Prim(**InstSiDos**) = {llaveF} U {}

Prim(**InstMientras**) = {mientrasTok}

Prim(**InstProbar**) = {probarTok}

Prim(**InstProbarUno**) = {defectoTok} U {}

Prim(**MasInstProbar**) = {casoTok} U {}

Prim(**InstPara**) = {paraTok}

Prim(**InstParaUno**) = {llaveI, puntoycoma}

Prim(**InstHacer**) = {hacerTok}

Prim(**InstParaCad**) = {paracadaTok}

Prim(**InstObtener**) = {sistemaTok}

Prim(**Obtener**) = {obtenerEnteroTok, obtenerRealTok, obtenerCadenaTok, obtenerCaracterTok}

Prim(**InstMostrar**) = {sistemaTok}

Prim(**InstMostrarUno**) = {coma} U {}

# **N**. Aplicación de las Reglas de Primero y Siguiente

**Definición de gramática LL(1)**
Para que una gramática independiente de contexto cumpla con ser una gramática LL1 se debe la siguiente regla:

Dada una Gramática libre de contexto G(CFG) y sus producciones de la forma:
$\zeta \rightarrow \zeta_1 \mid \zeta_2 \mid \cdots \mid \zeta_n$ se debe cumplir que:

$$\mathrm{Prim}(\zeta_1) \cap \mathrm{Prim}(\zeta_2) \cdots \cap \mathrm{Prim}(\zeta_n) = \emptyset$$

Luego, si se puede derivar a ε (palabra vacía) de un símbolo no terminal X, se requiere además que:

$$\mathrm{Prim}(\mathrm{X}) \cap \mathrm{Sig}(\mathrm{X}) = \emptyset$$

A continuación, se presenta las evaluaciones matemáticas de las reglas de Primero y Siguiente, utilizadas para comprobar si la gramática que genera al Lenguaje **Armus** es LL(1):

**********

**Programa** → **IncluirArchivo DefinicionClase**
**IncluirArchivo** → ε **|** incluirTok datoCadena puntoycoma **IncluirArchivo**

Regla 01
Prim(IncluirArchivo1) ∩ Prim(incluirArchivo2) = {} ∩ {incluirTok} = ∅

Regla 02
Prim(IncluirArchivo) ∩ Sig(IncluirArchivo) = Prim(IncluirArchivo) ∩ Prim(DefinicionClase) = {incluirTok} ∩ {publicaTok, localTok} = ∅

Por tanto, la gramática CUMPLE con ser LL(1) en la producción "**IncluirArchivo**".

**********

**DefinicionClase** → **IdentClass** claseTok ident llaveI **Cuerpo** llaveF **MasDefinicionClase**
**IdentClass** → publicaTok **|** localTok
**MasDefinicionClase** → ε **|** **DefinicionClase**

Regla 01
Prim(MasDefinicionClase1) ∩ Prim(MasDefinicionClase2) = {} ∩ {publicaTok, localTok} = ∅

Regla 02
Prim(MasDefinicionClase) ∩ Sig(MasDefinicionClase) = {publicaTok, localTok} ∩ {} = ∅

Por tanto, la gramática CUMPLE con ser LL(1) en la producción "**MasDefinicionClase**".

**********

**Cuerpo** → **Declar MasDeclar Declarar**
**Declar** → ε **|** **DedProp** **|** **DedMet**
**DedProp** → **IdentProp Dedvar** puntoycoma

**DedMet** → **IdentProp Metodo Bloque**
**IdentProp** → publicaTok **|** privadaTok

Regla 01
Prim(Declar1) ∩ Prim(Declar2) ∩ Prim(Declar3)
= {} ∩ Prim(DedProp) ∩ Prim(DedMet)
= {} ∩ Prim(IdentProp) ∩ Prim(IdentProp)
= {} ∩ {publicaTok, privadaTok} ∩ {publicaTok, privadaTok}
= {publicaTok, privadaTok} ≠ Ø

Por tanto, la gramática NO CUMPLE con ser LL(1) en la producción "**Declar**".
Semántica: Se lee el siguiente token para identificar si es un método o atributo (propiedad)

**********

**Cuerpo** → **Declar MasDeclar Declarar**
**Declar** → ε **|** **DedProp** **|** **DedMet**
**MasDeclar** → ε **|** **Declar MasDeclar**
**DedProp** → **IdentProp Dedvar** puntoycoma
**DedMet** → **IdentProp Metodo Bloque**
**IdentProp** → publicaTok **|** privadaTok
**Declarar** → ε **|** **DeclararPrincipal**
**DeclararPrincipal** → publicaTok principalTok corcheteI **LlevaParamPrin** corcheteF **Bloque**

Regla 01
Prim(MasDeclar1) ∩ Prim(MasDeclar2)
= {} ∩ Prim(Declar)
= {} ∩ {publicaTok, privadaTok} ∩ {}
= Ø

Regla 02
Prim(MasDeclar) ∩ Sig(MasDeclar)
= {publicaTok, privadaTok} ∩ Prim(Declarar)
= {publicaTok, privadaTok} ∩ {publicaTok} U {}
= {publicaTok} ≠ Ø

Por tanto, la gramática NO CUMPLE con ser LL(1) en la producción "**MasDeclar**".
Semántica: Se lee el siguiente token para identificar si es un método o atributo (propiedad)

**********

**Cuerpo** → **Declar MasDeclar Declarar**
**Declarar** → ε **|** **DeclararPrincipal**
**DeclararPrincipal** → publicaTok principalTok corcheteI **LlevaParamPrin** corcheteF **Bloque**

Regla 01
Prim(Declarar1) ∩ Prim(Declarar2)
= {} ∩ Prim(DeclararPrincipal)
= {} ∩ {publicaTok} = Ø

Regla 02
Prim(Declarar) ∩ Sig(Declarar) = {publicaTok} ∩ {} = ∅

Por tanto, la gramática CUMPLE con ser LL(1) en la producción "**Declarar**".

**********

**Dedvariable** → ident **Dedvari**
**Dedvari** → ε **|** asignacion **Expresion**

Regla 01
Prim(Dedvari1) ∩ Prim(Dedvar2) = {} ∩ {asignacion} = ∅

Regla 02
Prim(Dedvari) ∩ Sig(Dedvari) = {asignacion} ∩ {} = ∅

Por tanto, la gramática CUMPLE con ser LL(1) en la producción "**Dedvari**".

**********

**Dedvar** → **Tipo Dedvariable MasDedvar**
**MasDedvar** → ε **|** coma **Dedvariable MasDedvar**

Regla 01
Prim(MasDedvar1) ∩ Prim(MasDedvar2) = {} ∩ {coma} = ∅

Regla 02
Prim(MasDedvar) ∩ Sig(MasDedvar) = {coma} ∩ {} = ∅

Por tanto, la gramática CUMPLE con ser LL(1) en la producción "**MasDedvar**".

**********

**Metodo** → ident corcheteI corcheteF **Tipo** **|** ident corcheteI **ListParam** corcheteF
**ListParam** → **Parametro MasListParam**
**MasListParam** → ε **|** coma **ListParam**

Regla 01
Prim(MasListParam1) ∩ Prim(MasListParam2) = {} ∩ {coma} = ∅

Regla 02
Prim(MasListParam) ∩ Sig(MasListParam) = {coma} ∩ {corcheteF} = ∅

Por tanto, la gramática CUMPLE con ser LL(1) en la producción "**MasListParam**".

**********

**Parametro** → **Tipo Param** ident
**Param** → ε **|** por

Regla 01
Prim(Param1) ∩ Prim(Param2) = {} ∩ {por} = Ø

Regla 02
Prim(Param) ∩ Sig(Param) = {por} ∩ {ident} = Ø

Por tanto, la gramática CUMPLE con ser LL(1) en la producción "**Param**".

**********

**DeclararPrincipal** → publicaTok principalTok corcheteI **LlevaParamPrin** corcheteF **Bloque**
**LlevaParamPrin** → ε **|** **ParamPrin**
**ParamPrin** → **TipoPrimIdent ParamPrinUno**
**TipoPrimIdent** → **TipoPrim** ident
**TipoPrim** → caracterTok **|** cadenaTok **|** enteroTok **|** realTok **|** byteTok **|** boolenoTok

Regla 01
Prim(LlevaParamPrin1) ∩ Prim(LlevaParamPrin2)
= {} ∩ Prim(ParamPrin)
= {} ∩ {caracterTok, cadenaTok, enteroTok, realTok, byteTok, boolenoTok} = Ø

Regla 02
Prim(LlevaParamPrin) ∩ Sig(LlevaParamPrin)
= {caracterTok, cadenaTok, enteroTok, realTok, byteTok, boolenoTok} ∩ {corcheteF} = Ø

Por tanto, la gramática CUMPLE con ser LL(1) en la producción "**LlevaParamPrin**".

**********

**ParamPrin** → **TipoPrimIdent ParamPrinUno**
**ParamPrinUno** → ε **|** coma **TipoPrimIdent**

Regla 01
Prim(ParamPrinUno1) ∩ Prim(ParamPrinUno2) = {} ∩ {coma} = Ø

Regla 02
Prim(ParamPrinUno) ∩ Sig(ParamPrinUno) = {coma} ∩ {} = Ø

Por tanto, la gramática CUMPLE con ser LL(1) en la producción "**ParamPrinUno**".

**********

**LlamadaMet** → **FuncionArreglo** **|** ident corcheteI **LlamadaMetUno** corcheteF
**LlamadaMetUno** → ε **|** **IdentExpres MasIdentExpres**

Regla 01
Prim(LlamadaMetUno1) ∩ Prim(LlamadaMetUno2)
= {} ∩ Prim(IdentExpres)
= {} ∩ {referencia, datoCaracter, ident, datoCadena, concatenarTok, verdaderoTok, falsoTok, negacion}
= Ø

Regla 02
Prim(LlamadaMetUno) ∩ Sig(LlamadaMetUno)
= {referencia, datoCaracter, ident, datoCadena, concatenarTok, verdaderoTok, falsoTok, negacion} ∩ {corcheteF}
= Ø

Por tanto, la gramática CUMPLE con ser LL(1) en la producción "**LlamadaMetUno**".

**********

**LlamadaMetUno** → ε **|** **IdentExpres MasIdentExpres**
**MasIdentExpres** → ε **|** coma **IdentExpres**

Regla 01
Prim(MasIdentExpres1) ∩ Prim(MasIdentExpres2) = {} ∩ {coma} = Ø

Regla 02
Prim(MasIdentExpres) ∩ Sig(MasIdentExpres) = {coma} ∩ {} = Ø

Por tanto, la gramática CUMPLE con ser LL(1) en la producción "**MasIdentExpres**".

**********

**ExpresNum** → **ExpresCon MasExpresNum**
**MasExpresNum** → ε **|** otok **ExpresCon**

Regla 01
Prim(MasExpresNum1) ∩ Prim(MasExpresNum2) = {} ∩ {otok} = Ø

Regla 02
Prim(MasExpresNum) ∩ Sig(MasExpresNum) = {otok} ∩ {} = Ø

Por tanto, la gramática CUMPLE con ser LL(1) en la producción "**MasExpresNum**".

**********

**ExpresCon** → **ExpresRel MasExpresCon**
**MasExpresCon** → ε **|** ytok **ExpresRel**

Regla 01
Prim(MasExpresCon1) ∩ Prim(MasExpresCon2) = {} ∩ {ytok} = Ø

Regla 02
Prim(MasExpresCon) ∩ Sig(MasExpresCon) = {ytok} ∩ {} = Ø

Por tanto, la gramática CUMPLE con ser LL(1) en la producción "**MasExpresCon**".

**********

**ExpresRel** → **Negacion ExpresAri XexpresRel**
**Negacion** → ε **|** negacion

Regla 01
Prim(Negacion1) ∩ Prim(Negacion2) = {} ∩ {negacion} = Ø

Regla 02
Prim(Negacion) ∩ Sig(Negacion)
= {negacion} ∩ Prim(ExpresAri)
= {negacion} ∩ {mas, menos} U {}
= Ø

Por tanto, la gramática CUMPLE con ser LL(1) en la producción "**Negacion**".

**********

**ExpresRel** → **Negacion ExpresAri XexpresRel**
**XexpresRel** → ε **|** **Simbolo ExpresAri**
**Simbolo** → mnr **|** myr **|** mei **|** mai **|** igl **|** nig

Regla 01
Prim(XexpresRel1) ∩ Prim(XexpresRel2)
= {} ∩ Prim(Simbolo)
= {} ∩ {mnr, myr, mei, mai, igl, nig}
= Ø

Regla 02
Prim(XexpresRel) ∩ Sig(XexpresRel)
= {mnr, myr, mei, mai, igl, nig} ∩ {}
= Ø

Por tanto, la gramática CUMPLE con ser LL(1) en la producción "**XexpresRel**".

**********

**ExpresAri** → **Termino MasExpresAri**
**MasExpresAri** → ε **|** **Signo Termino**
**Signo** → mas **|** menos

Regla 01
Prim(MasExpresAri1) ∩ Prim(MasExpresAri1)
= {} ∩ Prim(Signo)
= {} ∩ {mas, menos}
= Ø

Regla 02
Prim(MasExpresAri) ∩ Sig(MasExpresAri) = {mas, menos} ∩ {} = Ø

Por tanto, la gramática CUMPLE con ser LL(1) en la producción "**MasExpresAri**".

**********

**Termino** → **Factor MasTermino**
**MasTermino** → ε **|** **Oper Factor**

**Oper** → por **|** barra

Regla 01
Prim(MasTermino1) ∩ Prim(MasTermino2)
= {} ∩ Prim(Oper)
= {} ∩ {por, barra}
= Ø

Regla 02
Prim(MasTermino) ∩ Sig(MasTermino) = {por, barra} ∩ {} = Ø

Por tanto, la gramática CUMPLE con ser LL(1) en la producción "**MasTermino**".

**********

**Factor** → **Sing Factores**
**Sing** → ε **|** **Signo**
**Signo** → mas **|** menos

Regla 01
Prim(Sing1) ∩ Prim(Sing2)
= {} ∩ Prim(Signo)
= {} ∩ {mas, menos}
= Ø

Regla 02
Prim(Sing) ∩ Sig(Sing)
= {mas, menos} ∩ Prim(Factores)
= {mas, menos} ∩ {numeroEntero, numeroReal, parentI, ident, sistemaTok}
= Ø

Por tanto, la gramática CUMPLE con ser LL(1) en la producción "**Sing**".

**********

**FuncNum** → sistemaTok punto **Fnum** parentI **ExpresNum MasExpressNum** parentF
**MasExpressNum** → ε **|** coma **ExpresNum**

Regla 01
Prim(MasExpressNum1) ∩ Prim(MasExpressNum2) = {} ∩ {coma} = Ø

Regla 02
Prim(MasExpressNum) ∩ Sig(MasExpressNum) = {coma} ∩ {parentF} = Ø

Por tanto, la gramática CUMPLE con ser LL(1) en la producción "**MasExpressNum**".

**********

**FuncNumCad** → sistemaTok punto **Conlog** corcheteI **ValorCadena MasFuncNumCad** corcheteF
**MasFuncNumCad** → ε **|** coma **ValorCadena**

Regla 01
Prim(MasFuncNumCad1) ∩ Prim(MasFuncNumCad2) = {} ∩ {coma} = Ø

Regla 02
Prim(MasFuncNumCad) ∩ Sig(MasFuncNumCad) = {coma} ∩ {corcheteF} = Ø

Por tanto, la gramática CUMPLE con ser LL(1) en la producción "**MasFuncNumCad**".

**********

**Bloque** → llaveI **MasBloque Instruccion** llaveF
**MasBloque** → ε **|** **Dedvar** puntoycoma **MasBloque**

Regla 01
Prim(MasBloque1) ∩ Prim(MasBloque2)
= {} ∩ Prim(Dedvar)
= {} ∩ {ident, objetoTok, archivoTok, caracterTok, cadenaTok, enteroTok, realTok, byteTok, booleanoTok, arregloTok}
= Ø

Regla 02
Prim(MasBloque) ∩ Sig(MasBloque)
= {ident, objetoTok, archivoTok, caracterTok, cadenaTok, enteroTok, realTok, byteTok, booleanoTok, arregloTok} ∩ Prim(Instruccion)
= {ident, objetoTok, archivoTok, caracterTok, cadenaTok, enteroTok, realTok, byteTok, booleanoTok, arregloTok} ∩ {romperTok, retornarTok, ident, agregarTok, obtenerTok, quitarTok, cuantosTok, siTok, probarTok, mientrasTok, paraTok, hacerTok, paracadaTok, sistemaTok} U {}
= {ident} ≠ Ø

Por tanto, la gramática NO CUMPLE con ser LL(1) en la producción "**MasBloque**".

**********

**Instruccion** → ε **|** **MasInstruccion Instruccion**
**MasInstruccion** → **IdentLlamada** puntoycoma **|** **LlamadaMet** puntoycoma **|** **Asignacion** puntoycoma **|** romperTok puntoycoma **|** **InstSi** **|** **InstProbar** **|** **InstMientras** **|** **InstPara** **|** **InstHacer** **|** **InstParaCad** **|** **InstES** **|** retornarTok **Expresion** puntoycoma

Regla 01
Prim(Instruccion1) ∩ Prim(Instruccion2)
= {} ∩ Prim(MasInstruccion)
= {} ∩ {romperTok, retornarTok, ident, agregarTok, obtenerTok, quitarTok, cuantosTok, siTok, probarTok, mientrasTok, paraTok, hacerTok, paracadaTok, sistemaTok}
= Ø

PERO:
Prim(IdentLlamada) = {ident}
Prim(LlamadaMet) = {ident, agregarTok, obtenerTok, quitarTok, cuantosTok}
Prim(Asignacion) = {ident}
Prim(InstES) = {sistemaTok, ident}

Regla 01
Prim(IdentLlamada) ∩ Prim(LlamadaMet) ∩ Prim(Asignacion) ∩ Prim(InstES) = {ident} ≠ Ø

Por tanto, la gramática NO CUMPLE con ser LL(1) en la producción "**Instruccion**".
Semántica: Si se encuentra un identificador, se verifica si es un método, un objeto, un objeto de tipo archivo o una variable, para decidir si se trata de una llamada a un método, un objeto a utilizar, un objeto archivo para llamada de método o una asignación a una variable.

**********

**Expresion** → **ValorCaracter** | **ValorCadena** | **DatoBool** | **ExpresNum**

Prim(ValorCaracter) = {datoCaracter, ident, agregarTok, obtenerTok, quitarTok, cuantosTok}
Prim(ValorCadena) = {datoCadena, ident, agregarTok, obtenerTok, quitarTok, cuantosTok, concatenarTok}
Prim(DatoBool) = {verdaderoTok, falsoTok}
Prim(ExpresNum) = {negacion, mas, menos, numeroEntero, numeroReal, parentI, ident, agregarTok, obtenerTok, quitarTok, cuantosTok, sistemaTok}

Regla 01
Prim(ValorCaracter) ∩ Prim(ValorCadena) ∩ Prim(DatoBool) ∩ Prim(ExpresNum)
= {ident, agregarTok, obtenerTok, quitarTok, cuantosTok} ≠ Ø

Por tanto, la gramática NO CUMPLE con ser LL(1) en la producción "**Expresion**".
Semántica: Si se encuentra un identificador la compilación seguirá si se cumple lo siguiente:

a) Si el identificador es una variable entera o real entonces seguirá por el camino de Expresión_Numerica.
b) Si el identificador es una función que retorna un valor entero o real entonces seguirá por el camino de Expresión_Numerica.
c) Si el identificador es una variable de tipo carácter o una función que retorna un valor de tipo carácter entonces seguirá por el camino de ValorCaracter.
d) Si el identificador es una variable de tipo cadena o una función que retorna un valor de tipo cadena entonces seguirá por el camino de ValorCadena.

Si se encuentra una palabra reservada agregar, obtener, quitar, cuantos, entonces se seguirá por el camino de Función_Arreglo.

**********

**InstSiUno** → coma **Asignacion** coma **Asignacion** corcheteF puntoycoma **|** corcheteF llaveI **Instruccion** **InstSiDos** llaveF
**InstSiDos** → ε **|** llaveF sinoTok llaveI **Instruccion**

Regla 01
Prim(InstSiDos1) ∩ Prim(InstSiDos2) = {} ∩ {llaveF} = Ø

Regla 02
Prim(InstSiDos) ∩ Sig(InstSiDos) = {llaveF} ∩ {llaveF} ≠ Ø

Por tanto, la gramática NO CUMPLE con ser LL(1) en la producción "**InstSiDos**".

Semántica: se deberá leer el siguiente token, si es un “sino” se debe realizar la producción InstSiDos sino no y continua.

**********

**InstProbar** → probarTok corcheteI **Expresion** corcheteF llaveI casoTok **Expresion** dospuntos **Instruccion** **MasInstProbar InstProbarUno** llaveF
**InstProbarUno** → ε **|** defectoTok dospuntos **Instruccion**

Regla 01
Prim(InstProbarUno1) ∩ Prim(InstProbarUno2) = {} ∩ {defectoTok} = Ø

Regla 02
Prim(InstProbarUno) ∩ Sig(InstProbarUno) = {defectoTok} ∩ {llaveF} = Ø

Por tanto, la gramática CUMPLE con ser LL(1) en la producción “**InstProbarUno**”.

**********

**InstProbar** → probarTok corcheteI **Expresion** corcheteF llaveI casoTok **Expresion** dospuntos **Instruccion** **MasInstProbar InstProbarUno** llaveF
**InstProbarUno** → ε **|** defectoTok dospuntos **Instruccion**
**MasInstProbar** → ε **|** casoTok **Expresion** dospuntos **Instruccion**

Regla 01
Prim(MasInstProbar1) ∩ Prim(MasInstProbar1) = {} ∩ {casoTok} = Ø

Regla 02
Prim(MasInstProbar) ∩ Sig(MasInstProbar)
= {casoTok} ∩ Prim(InstProbarUno)
= {casoTok} ∩ {defectoTok} U {}
= Ø

Por tanto, la gramática CUMPLE con ser LL(1) en la producción “**MasInstProbar**”.

**********

**InstMostrar** → sistemaTok punto mostrarTok corcheteI **Expresion InstMostrarUno** corcheteF puntoycoma
**InstMostrarUno** → ε **|** coma **Expresion**

Regla 01
Prim(InstMostrarUno1) ∩ Prim(InstMostrarUno2) = {} ∩ {coma} = Ø

Regla 02
Prim(InstMostrarUno) ∩ Sig(InstMostrarUno) = {coma} ∩ {corcheteF} = Ø

Por tanto, la gramática CUMPLE con ser LL(1) en la producción “**InstMostrarUno**”.

**********

**Factor** → **Sing Factores**

**Sing** → ε **|** **Signo**
**Factores** → numeroEntero **|** numeroReal **|** **IdentLlamada** **|** **LlamadaMet** **|** **FuncNum** **|** **FuncNumCad** **|** parentI **ExpresNum** parentF

Prim(IdentLlamada) = {ident}
Prim(LlamadaMet) = {ident, agregarTok, obtenerTok, quitarTok, cuantosTok}
Prim(FuncNum) = {sistemaTok}
Prim(FuncNumCad) = {sistemaTok}

Regla 01
Prim(IdentLlamada) ∩ Prim(LlamadaMet) = {ident} ≠ ∅

Prim(FuncNum) ∩ Prim(FuncNumCad) = {sistemaTok} ≠ ∅

Por tanto, la gramática NO CUMPLE con ser LL(1) en la producción "**Factores**".
Semántica: Si se encuentra un identificador la compilación seguirá si se cumple lo siguiente:

a) Si el identificador es una variable entera o real entonces seguirá por el camino de Declaración_Variable (atributo).
b) Si el identificador es un método entonces seguirá por el camino de Llamada_Metodo.

Si es la palabra reservada "Sistema" se adelantará dos tokens hasta determinar a cuál función hace referencia, si es del camino Funcion_Numerica o Funcion_Numerica_Cadena.

**********

**ValorCaracter** → datoCaracter **|** **IdentLlamada** **|** **LlamadaMet**

Regla 01
Prim(IdentLlamada) ∩ Prim(LlamadaMet) = {ident} ≠ ∅

Por tanto, la gramática NO CUMPLE con ser LL(1) en la producción "**ValorCaracter**".
Semántica: Si se encuentra un identificador la compilación seguirá si se cumple lo siguiente:

a) Si el identificador es una variable de tipo carácter entonces seguirá por el camino de Declaración_Variable (atributo).
b) Si el identificador es una función que retorna un carácter entonces seguirá por el camino de Llamada_Metodo.
c) Si el identificador es cualquier otra cosa entonces se emitirá un error.

**********

**ValorCadena** → datoCadena **|** **IdentLlamada** **|** **LlamadaMet** **|** **FuncCad**

Regla 01
Prim(IdentLlamada) ∩ Prim(LlamadaMet) = {ident} ≠ ∅

Por tanto, la gramática NO CUMPLE con ser LL(1) en la producción "**ValorCaracter**".
Semántica: Si se encuentra un identificador la compilación seguirá si se cumple lo siguiente:

a) Si el identificador es una variable de tipo cadena entonces seguirá por el camino de Declaración_Variable (atributo).
b) Si el identificador es una función que retorna una cadena entonces seguirá por el camino de Llamada_Metodo.

Si es la palabra reservada “concatenar” entonces seguirá por el camino de Función_Cadena.

**********

## O. Reglas Semánticas Extras Aplicadas

En la producción **Bloque**.

- Si el bloque pertenece a un método de tipo función debe existir al menos un “retornar” con su respectiva expresión.
- Si el bloque pertenece a un método de tipo procedimiento no puede existir un “retornar”.

En la producción **Llamada_Metodo**.

- Verificar si el identificador se encuentra declarado como un método.
- Verificar que la cantidad de parámetros del método sea igual a la definida en su declaración
- Verificar que el valor de los parámetros corresponda al tipo con el que fueron declarados. (Validación realizada en tiempo de ejecución).

En la producción **Asignacion**.

- Luego de evaluar Expresión se debe comprobar que exista una compatibilidad de tipos (Validación realizada en tiempo de ejecución).

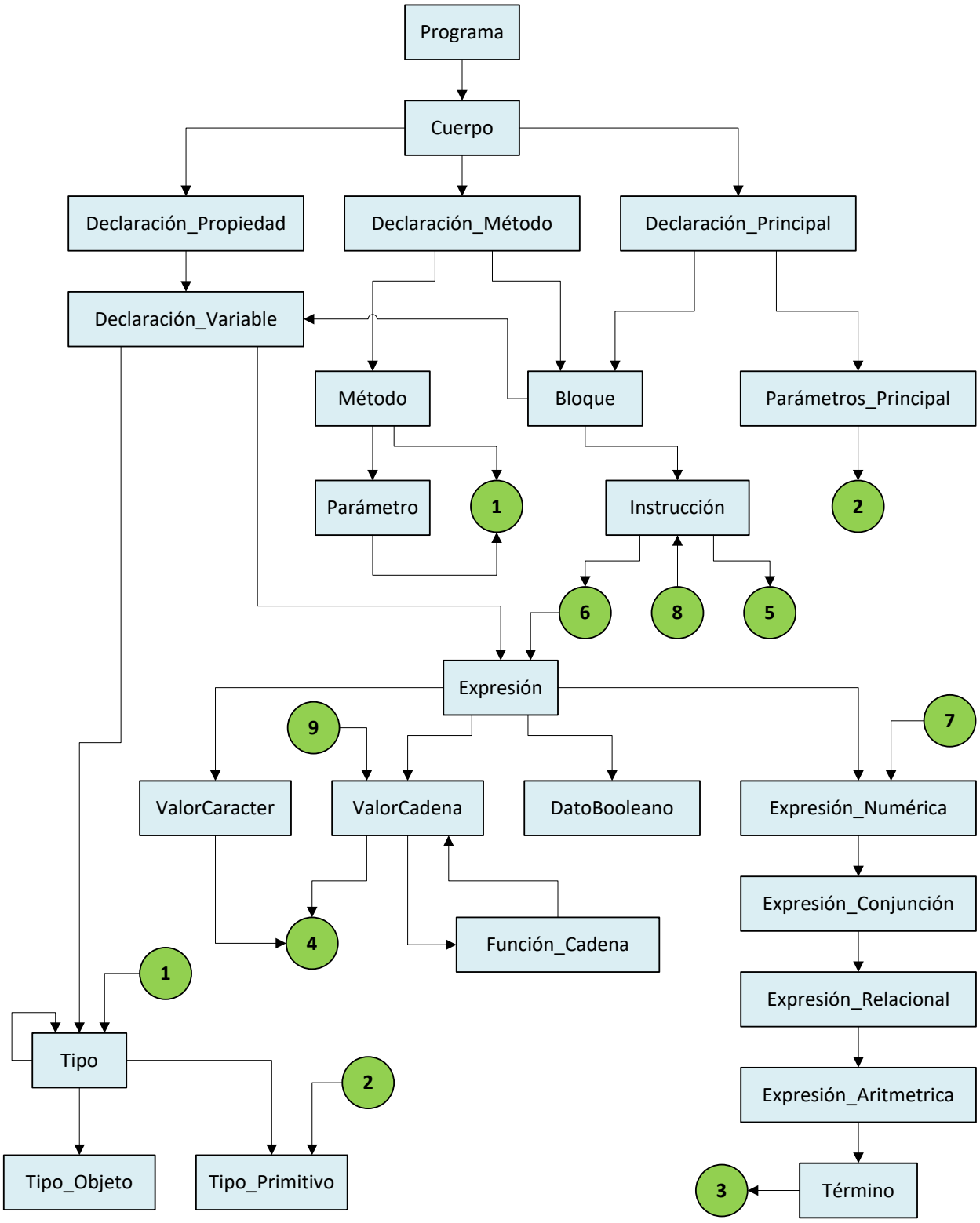
En la producción **Instruccion_Para**.

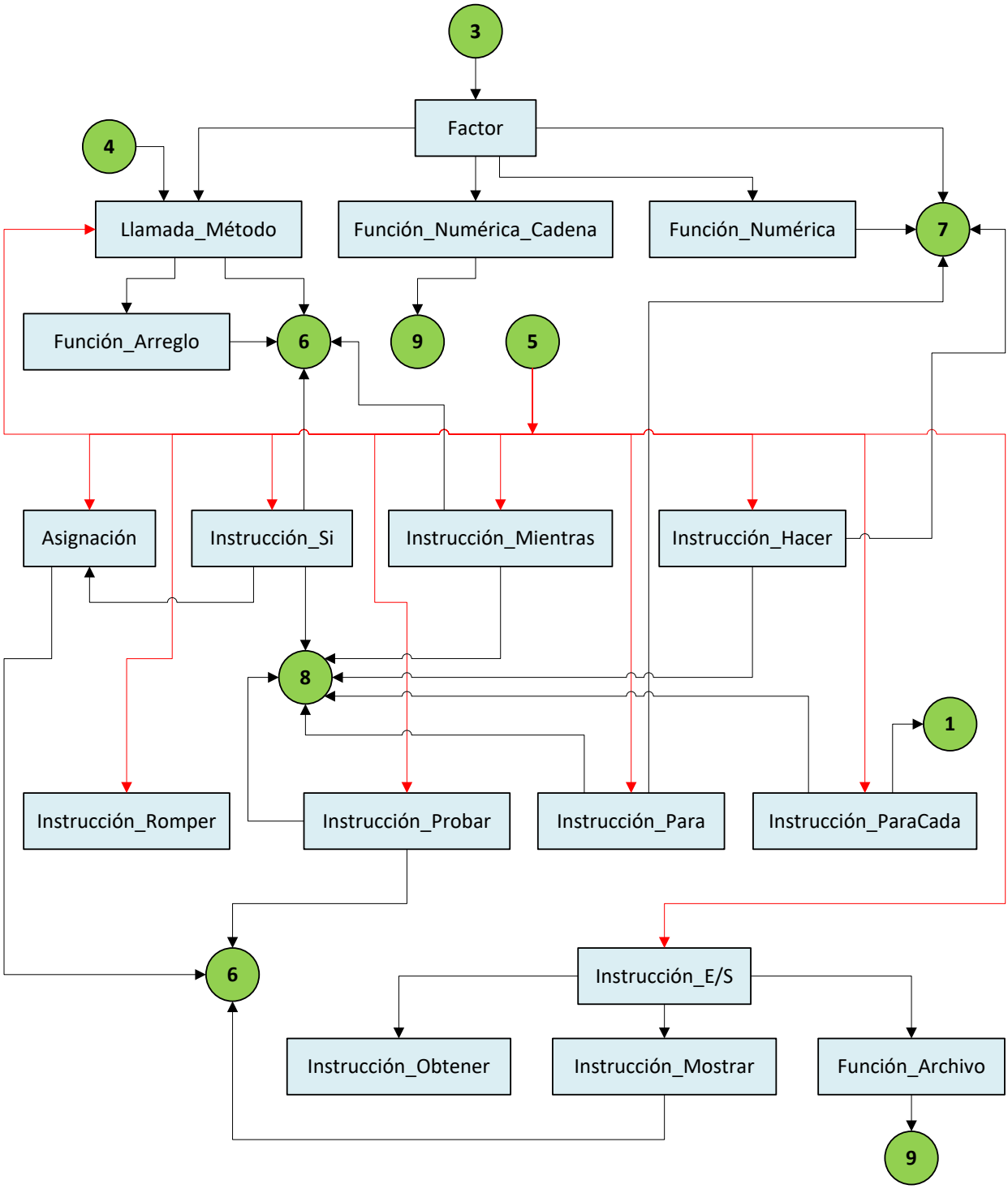
- Variable debe ser un variable entera.
- El valor de Expresion_Numerica debe ser un entero o al menos que exista compatibilidad de tipos.
- (Validación realizada en tiempo de ejecución).
- Para
    - » Limite1: Valor de la primera aparición de Expresion_Numerica.
    - » Limite2: Valor de la segunda aparición de Expresion_Numerica.
    - » Paso: Valor de la tercera aparición de Expresion_Numerica.

  Se debe cumplir una de las siguientes dos reglas:
    a) Si Limite1 < Limite2 entonces Paso > 0.
    b) Si Limite1 > Limite2 entonces Paso < 0.

  (Validación realizada en tiempo de ejecución)

## P. Diagrama de Dependencias




3
Factor
4
Llamada_Método
Función_Numérica_Cadena
Función_Numérica
7
Función_Arreglo
6
9
5
Asignación
Instrucción_Si
Instrucción_Mientras
Instrucción_Hacer
8
1
Instrucción_Romper
Instrucción_Probar
Instrucción_Para
Instrucción_ParaCada
6
Instrucción_E/S
Instrucción_Obtener
Instrucción_Mostrar
Función_Archivo
9

## Q. Estrategias para la estabilización de errores sintácticos y semánticos

- En la función del analizador sintáctico del compilador cada llamada a alguna otra función, se añadirá un parámetro extra que contendrá un conjunto de tokens seguidores denominado toksig, el cual contendrá todas las palabras reservadas o caracteres especiales del lenguaje que se esperan ver después de que la función invocada finalice el análisis sintáctico correspondiente.
  Esto con el fin de que dicho conjunto toksig ante la presencia de un error en la función llamada, se logre realizar un salto de texto (o código) hasta encontrar un token con el que la función (la que realizo inicialmente la llamada de la secuencia) pueda continuar analizando el código fuente.

- En la función del analizador sintáctico del compilador cada llamada a alguna otra función, agregará el parámetro toksig de la función invocada su propio conjunto toksig. Esto para que, al encontrar un error en la función llamada, se podrá también realizar un salto de código (texto) hasta un token con el que la función precedente a la función que realizo la invocación pueda usar.

- Antes de iniciar el código del analizador sintáctico se crea un conjunto toksig de arranque (conjunto inicial), que tenga los elementos que siguen de la primera función del análisis sintáctico, de manera que pueda iniciarse la sincronización encargada de estabilizar la detección de errores dentro del compilador.

- Cuando se encuentra un error se realiza un salto de texto inteligente (buscando el token del conjunto toksig al cual pueda saltar), aprobando que el proceso de compilación pueda continuar y evitando la generación de errores en cascada.

- Como cada archivo *.acl puede contener más de una clase, el conjunto de arranque (conjunto inicial) está compuesto de tres tokens: publicaTok, localTok, claseTok, dichos tokens corresponden a las primeras palabras reservadas que busca en el archivo y que pueden generar errores dentro de su sintaxis; el conjunto toksig para este caso es únicamente el token llaveF.

## R. Estrategia para el Manejo de Código Intermedio (Múltiples Archivos)

Cada archivo **.acl** se crear un archivo de código intermedio con la extensión **.ob** y un archivo extra para guardar una estructura donde estén los valores de cada atributo de la clase **.strc** así:

| **Clase1.acl** | **Clase1.ob** | **Clase1.strc** |
|---|---|---|
| Contiene el código fuente del programa | Contiene el código intermedio del programa (código-p) | Estructura con los valores de cada atributo de la clase. |

Cada **clase** utiliza una plantilla y cada instancia tiene que tener la misma forma. Así por cada clase se genera un código intermedio completo.

| **Persona.acl** | **Persona.ob** | **Persona.strc** |
|---|---|---|
| | | |

| **Main.acl** | **Main.ob** | **Main.strc** |
|---|---|---|
| | | |

# S. Diseño del Código Intermedio

El formato de una instrucción en código intermedio es:

| f | n | d | p |
|---|---|---|---|

Donde:
**f**: es el mnemónico de la instrucción.
**n**: es la diferencia estática de nivel de la variable, para el caso de arreglos representa el nivel de anidamiento existe, pero también depende de f:

- Si **f** es **INO**, y además de tipo Arreglo (d = -2), **n** indica el nivel de anidamiento para dicho arreglo.
- Si **f** es **CAR**, **CRR**, y **n** = 2 indica que cargara desde una pila auxiliar.

**d**: este valor depende del valor de **f**:

- Si **f** es **LIT**, **d** indica el valor literal a cargar.
- Si **f** es **INS**, **d** indica la cantidad de espacios en memoria de la pila.
- Si **f** es **INO**, **d** indica el tipo de objeto o hash de la clase a la que pertenece:
  - -2 → Arreglo
  - -1 → Archivo
  - 0 → Objeto
  - # → Hash de la clase
- Si **f** es **ALM**, **CAR**, **CRR**, **d** indica posición en la pila.
- Si **f** es **OPR**, **d** indica la operación a realizar.

**p**: este valor depende del valor de **f**:

- Si **f** es **INS**, **LIT**, **p** indica el tipo de dato primitivo:
  - 0 → sin tipo de dato
  - 1 → cadena
  - 2 → carácter
  - 3 → entero
  - 4 → real
  - 5 → byte
  - 6 → booleano
- Si **f** es **INO**, y además es de tipo Arreglo (d = -2), **p** indica el tipo de dato contenido en el Arreglo. Es 0 si es de otro tipo de objeto.
- Si **f** es **ALM**, **CAR**, **CRR**, **p** indica la posición de la estructura de valores.
- Si **f** es **SAC**, **SAL**, **p** indica la dirección de salto en el código intermedio.

El código intermedio generado por el compilador del lenguaje Armus cuenta con los siguientes mnemónicos para la construcción de sus instrucciones:

**INS** Instancia o reserva de espacio en la pila.

| f | n | d | p |
|---|---|---|---|
| INS | | | |

**n**: nivel (n = 0 siempre)
**d**: cantidad de espacios en memoria
**p**: tipo de dato primitivo a instanciar

Valores de **p**:

- 0 → sin tipo de dato
- 1 → cadena
- 2 → carácter
- 3 → entero
- 4 → real
- 5 → byte
- 6 → booleano

**INO** Instancia un objeto o reserva espacio en la pila para un objeto.

| f | n | d | p |
|---|---|---|---|
| **INO** | | | |

**n**: anidamiento para Arreglo
**d**: hash de la clase a la que pertenece (ó tipo de objetos).
**p**: tipo de dato contenido en el Arreglo

Valores de **p**:
- -2 → Arreglo
- -1 → Archivo
- 0 → Objeto
- # → Hash de la clase

**ALM** Almacenamiento

| f | n | d | p |
|---|---|---|---|
| **ALM** | | | |

**n**: nivel.
**d**: posición en la pila.
**p**: posición en la estructura de valores.

**LIT** Cargar un valor literal (constante) al tope de la pila.

| f | n | d | p |
|---|---|---|---|
| **LIT** | | | |

**n**: nivel (n = 0 siempre).
**d**: valor a cargar.
**p**: tipo de dato.

Valores de **p**:
- 0 → sin tipo de dato
- 1 → cadena
- 2 → carácter
- 3 → entero
- 4 → real
- 5 → byte
- 6 → booleano

**CAR** Cargar.

| f | n | d | p |
|---|---|---|---|
| **CAR** | | | |

**n**: nivel (n = 2 indica que cargara desde una pila auxiliar)
**d**: posición donde está el valor en la pila.
**p**: posición de la estructura de valores.

**CRR** Cargar por Referencia (equivalente a CAR, pero este mnemónico hace un puntero por referencia, no una copia del valor).

| f | n | d | p |
|---|---|---|---|
| **CRR** | | | |

**n**: nivel (n = 2 indica que cargara desde una pila auxiliar)
**d**: posición donde está el valor en la pila.
**p**: posición de la estructura de valores.

**SAC** Salto condicional.

| f | n | d | p |
|---|---|---|---|
| **SAC** | | | |

**n**: nivel.
**d**: no utilizado (d = 0 siempre).
**p**: dirección de salto en el código objeto.

**SAL** Salto incondicional.

| f | n | d | p |
|---|---|---|---|
| **SAL** | | | |

**n**: nivel.
**d**: no utilizado (d = 0 siempre).
**p**: dirección de salto en el código objeto.

**OPR** Operación.

| f | n | d | p |
|---|---|---|---|
| **OPR** | | | |

**n**: nivel (n = 0 siempre).
**d**: operación a realizar (d = -1, es, fin de atributos).
**p**: no utilizado (p = 0 siempre).

Lista de operaciones:

d = -1 → fin de declaración de atributos de la clase.
d = 0 → fin de un método sin retorno
d = 1 → menos unario ( - )
d = 2 → suma ( + )
d = 3 → resta ( - )
d = 4 → multiplicación ( * )
d = 5 → división ( / )
d = 6 → menor que ( < )
d = 7 → mayor que ( > )
d = 8 → mayor o igual ( >= )
d = 9 → menor o igual ( <= )
d = 10 → igualdad ( == )
d = 11 → diferente de ( <> )
d = 12 → negación ( ! )
d = 13 → OR ( || )
d = 14 → AND ( && )
d = 15 → absoluto
d = 16 →decimal
d = 17 → esPar
d = 18 → parteEntera
d = 19 → modulo
d = 20 → potencia
d = 21 → menor
d = 22 → mayor
d = 23 → longitudCadena
d = 24 → comparar
d = 25 → concatenar
d = 26 → obtenerCaracter
d = 27 → obtenerCadena
d = 28 → obtenerReal
d = 29 → obtenerEntero
d = 30 → mostrar