

# Lenguaje BABAA

Luis Antonio Ortega Andrés  
Guillermo Galindo Ortuño  
Johanna Capote Robayna  
Antonio Coín Castro

## 1 Descripción del lenguaje

La sintaxis de nuestro lenguaje está inspirada en el lenguaje **C**. Por ello, tomaremos como referencia las reglas sintácticas de este para las instrucciones del nuevo lenguaje.

Para las palabras reservadas usaremos el idioma **castellano**.

En nuestro lenguaje implementaremos los **arrays 1D y 2D** como estructura de datos, que solo pueden tener elementos **del mismo** tipo básico en su interior: entero, carácter, real y booleano. Este tipo de datos tendrán las siguientes operaciones:

- Acceso a elemento.
- Producto.
- Suma y resta elemento a elemento.
- Producto externo (producto de un array por un escalar).
- Producto de matrices (teniendo en cuenta que para multiplicar dos matrices, las dimensiones de estas deben ser las adecuadas).

El tipo de subprograma que contendrá nuestro lenguaje son **funciones**, es decir, supondremos que los subprogramas siempre devuelven un valor.

Por último, el lenguaje incluirá una estructura de control tipo **switch**.

## 2 Descripción formal de la sintaxis del lenguaje usando BNF

```
<Programa> ::= <Cabecera_programa> <Bloque>
<Bloque> ::= <Inicio_de_bloque>
           <Declar_de_variables_locales>
           <Declar_de_subprogs>
           <Sentencias>
           <Fin_de_bloque>
<Declar_de_subprogs> ::= <Declar_de_subprogs> <Declar_subprog>
                        |
<Declar_subprog> ::= <Cabecera_subprograma> <Bloque>
<Declar_de_variables_locales> ::= <Marca_ini_declar_variables>
                                   <Variables_locales>
                                   <Marca_fin_declar_variables>
```

<Marca_ini_declar_variables>	::= var
<Marca_fin_declar_variables>	::= finvar
<Cabecera_programa>	::= programa()
<Inicio_de_bloque>	::= {
<Fin_de_bloque>	::= }
<Variables_locales>	::= <Variables_locales> <Cuerpo_declar_variable>;
	<Cuerpo_declar_variable>;
<Cuerpo_declar_variable>	::= <Tipo> <Lista_id>
<Acceso_array>	::= [<Expresion>]
	[<Expresion>, <Expresion>]
<Identificador_comp>	::= <Identificador>
	<Identificador><Acceso_array>
<Acceso_array_cte_>	::= [<Natural>]
	[<Natural>, <Natural>]
<Identificador_comp_cte>	::= <Identificador>
	<Identificador><Acceso_array_cte>
<Cabecera_subprog>	::= <Tipo_comp> <Identificador>(<Lista_argumentos>)
<Lista_argumentos>	::= <Argumentos>
<Argumentos>	::= <Argumento> , <Argumentos>
	<Argumento>
<Argumento>	::= <Tipo> <Identificador_comp_cte>
<Booleano>	::= verdadero   falso
<Digito>	::= 0   1   2   3   4   5   6   7   8   9
<Natural>	::= <Digito> <Natural>
	<Digito>
<Real>	::= <Natural>.<Natural>
<Caracter>	::= a   ...   z   A   ...   Z   _
<Alfanum>	::= <Caracter>
	<Natural>
<Caracter_ascii>	::= (Cualquier carácter ASCII menos las comillas (' o '''))
	\"
	\'
<Cadena>	::= <Cadena> <Caracter_ascii>
	<Caracter_ascii>
<Cadena_const>	::= "<Cadena>"
<Constante>	::= <Natural>
	<Real>
	<Booleano>
	'<Caracter_ascii>'
<Tipo>	::= entero
	real
	buleano
	caracter
<Tipo_comp>	::= <Tipo>
	<Tipo><Acceso_array>
<Identificador>	::= <Identificador> <Alfanum>

		<Caracter>
<Expresion>	::=	(<Expresion>)
		<Identificador_comp>
		<Constante>
		<Op_unario_izquierda> <Expresion>
		<Expresion> <Op_binario> <Expresion>
		<Agregado1D>
		<Agregado2D>
		<Llamada_funcion>
<Agregado1D>	::=	{<Expresiones>}
<Agregado2D>	::=	{<Listas> ; <Expresiones>}
<Listas>	::=	<Listas> ; <Expresiones>
		<Expresiones>
<Expresiones>	::=	<Expresion>, <Expresiones>
		<Expresion>
<Llamada_funcion>	::=	<Identificador>(<Expresiones>)
		<Identificador>()
<Op_unario_izquierda>	::=	!
		+
		-
<Op_binario>	::=	==
		>=
		<=
		!=
		*
		/
		+
		-
		^
		<
		>
		&&
<Sentencias>	::=	<Sentencias> <Sentencia>
<Sentencia>	::=	<Bloque>
		<Sentencia_asignacion>
		<Sentencia_if>
		<Sentencia_while>
		<Sentencia_switch>
		<Sentencia_break>
		<Sentencia_return>
		<Sentencia_entrada>
		<Sentencia_salida>
<Sentencia_asignacion>	::=	<Identificador_comp> = <Expresion>;
<Sentencia_if>	::=	si (<Expresion>) <Sentencia> <Sentencia_else>
<Sentencia_else>	::=	otro <Sentencia>

```

<Sentencia_while>      ::= mientras (<Expresion>) <Sentencia>
<Sentencia_switch>    ::= casos (<Expresion>) <Bloque_switch>
<Bloque_switch>       ::= { <Opciones> }
<Opciones>             ::= <Opciones> <Opcion>
                        | <Opcion> <Opcion_pred>
                        | <Opcion_pred>
<Opcion>              ::= caso <Natural>: <Sentencias>
<Opcion_pred>         ::= predeterminado: <Sentencias>
<Sentencia_break>     ::= roto;
<Sentencia_return>    ::= devolver <Expresion>
<Sentencia_entrada>   ::= entrada <Lista_id>;
<Lista_id>            ::= <Lista_id>, <Identificador_comp>
                        | <Identificador_comp>
<Lista_exp_cad>       ::= <Lista_exp_cad>, <Exp_cad>
                        | <Exp_cad>
<Exp_cad>             ::= <Expresion>
                        | <Cadena>
<Sentencia_salida>    ::= salida <Lista_exp_cad>;

```

### 3 Definición de la semántica en lenguaje natural

El programa comienza con una cabecera inicial y un bloque. La cabecera inicial esta formada por la palabra reservada “programa()”. Por otro lado, el bloque empieza con { y termina con }, y en su interior pueden aparecer variables locales, subprogramas o sentencias.

Las variables locales deben declararse entre unas marcas de inicio y fin, notadas como **var** y **finvar** respectivamente. Una vez dentro de las marcas, podemos declarar variables al estilo de C, e inicializarlas *in situ*:

```

<tipo> <id>;
<tipo> <id> = <expr>;

```

Los subprogramas son siempre funciones que devuelven algo. Constan de una cabecera <tipo> <nombre> (<argumentos>), donde los argumentos pueden ser 0 o más, separados por comas. El cuerpo de los subprogramas vuelve a ser un bloque, lo que permite anidamiento. Se devuelven los datos con la palabra clave **devolver**.

Las sentencias pueden ser un bloque, una expresión o las instrucciones de control (*si, otro, mientras, casos*). Las instrucciones de control contienen una expresion y una sentencia, y su sintaxis es la misma que la de C. Notamos que los casos en el *switch* solo pueden ser números enteros.

Una expresión puede encontrarse entre paréntesis, y puede ir precedida de una operación unaria o combinar dos expresiones con una operación binaria. Además puede ser una constante o un identificador.

Por último, tenemos un tipo especial similar a los arrays de C, que representa un agregado de datos del mismo tipo, ya sea unidimensional o bidimensional, cuya sintaxis es la siguiente:

```

// Declaración
<tipo> <id>[<tamaño>]; // 1-D
<tipo> <id>[<tamaño1>, <tamaño2>]; //2-D

```

```
// Acceso
<id>[<pos>]; // 1-D
<id>[<pos1>, <pos2>]; // 2-D
```

## 4 Tabla de tokens

Nombre	Expresión regular	Código	Atributos
CABECERA	"programa()"	257	
LLAVEIZQ	"{"	258	
LLAVEDCH	"}"	259	
INILOCAL	"var"	260	
FINLOCAL	"finvar"	261	
TIPO	"entero"   "real"   "buleano"   "caracter"	262	0:entero 1:real 2:buleano 3:caracter
IF	"si"	263	
ELSE	"otro"	264	
WHILE	"mientras"	265	
SWITCH	"casos"	266	
CASE	"caso"	267	
BREAK	"roto"	268	
CIN	"entrada"	269	
COUT	"salida"	270	
PREDET	"predeterminado"	271	
ASIG	"="	272	
CORCHIZQ	"["	273	
CORCHDCH	"]"	274	
COMA	","	275	
PYC	";"	276	
PYP	":"	277	
PARIZQ	"("	278	
PARDCH	")"	279	
IDENTIFICADOR	[a-zA-Z_][a-zA-Z0-9_]*	280	
CONSTANTE	-?[0-9]+(.[0-9]+)?   "verdadero"   "falso"   \'^[^\'\\]\'	281	0:num 1:verdadero 2:falso 3:cte_caracter
RETURN	"devolver"	282	
OPBIN	[<>]=?   [=!]=   "*"   "/"   "^"   "  "   "&&"	283	0:> 1:< 2:<= 3:>= 4:== 5:!= 6:* 7:/ 8:^ 9:   10:&&
OPUNARIOIZQ	"!"	284	
MASMENOS	+   -	285	0:+ 1:-
CADENA	\"[^\\"]*\\"	286	