Lenguaje BABAD

José María Borrás , Ismael Sánchez , Javier Sáez

Descripción del lenguaje

La sintaxis de nuestro lenguaje está inspirada en el lenguaje **C**. Esto significa que tomaremos las reglas sintácticas usadas por este lenguaje como referencia para las instrucciones del lenguaje nuevo, respetando en todo momento los requerimientos impuestos al lenguaje. Usaremos el lenguaje **castellano** para las palabras reservadas de nuestro lenguaje. Además, requeriremos que las declaraciones estén enmarcadas.

Vamos a implementar un tipo concreto de estructura de datos en nuestro lenguaje: serán los **arrays 1D y 2D**. No se permitirá la declaración de variables de manera recursiva o mezclada, es decir, no se permitirán arrays 1D de arrays 2D. Este tipo de datos tendrá las operaciones siguientes:

- Acceso a elemento
- Producto
- Suma y resta elemento a elemento
- Producto externo (producto de un array por un escalar)
- Producto de matrices:

  $$C = A \times B$$

  sabiendo que si $A \in \mathcal{M}_{p\times q}$ y $B \in \mathcal{M}_{r\times s}$, es necesario que $q = r$ y el resultado será una matriz $C \in \mathcal{M}_{p\times s}$. Adicionalmente, contemplaremos las constantes de tipo array.

Esta estructura solo puede tener elementos de los tipos básicos definidos (entero, real, carácter o booleano).La sintaxis del lenguaje permitirá construir expresiones complejas con las operaciones definidas.

Nuestro lenguaje también dispondrá de subprogramas. En concreto existirá la posibilidad de crear **funciones** externas al programa principal. Por último, presentaremos una estructura de control: los **bucles do-until**.

La estructura sintáctica de un programa en nuestro lenguaje será:

```
<Programa> ::= <Cabecera_programa> <bloque>
```

Se podrán definir bloques en cualquier parte, al igual que en el lenguaje de referencia. La estructura sintáctica del bloque es la siguiente:

```
<bloque> ::= <Inicio_de_bloque>
             <Declar_de_variables_locales>
             <Declar_de_subprogs>
             <Sentencias>
             <Fin_de_bloque>
```

Una sentencia cualquiera podrá ser un bloque, por lo que podremos anidar bloques y subprogramas. Los argumentos se pasarán por valor y no se permitirán declaraciones fuera de los bloques. Estas declaraciones deben ir entre marcas de declaración (inicio y fin).

Al igual que en el programa, la estructura sintáctica de un subprograma es:

```
<Declar_subprog> ::= <Cabecera_subprograma> <bloque>
```

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> ::= ini_var
<Marca_fin_declar_variables> ::= fin_var

<Cabecera_programa> ::= principal
<Inicio_de_bloque> ::= {
<Fin_de_bloque> ::= }
<Variables_locales> ::= <Variables_locales> <Cuerpo_declar_variables>
                        |  <Cuerpo_declar_variables>

<Cuerpo_declar_variables> ::= <tipo_basico><lista_identificador> ;
<Cabecera_subprograma> ::= <tipo_basico> <ident_array> (<lista_parametros>)

<Sentencias> ::= <Sentencias> <Sentencia>
                 |  <Sentencia>
<Sentencia> ::= <bloque>
                |  <sentencia_asignacion>
                |  <sentencia_if>
                |  <sentencia_do_until>
                |  <sentencia_entrada>
                |  <sentencia_salida>
                |  <sentencia_return>


<sentencia_asignacion> ::= <array_ident> = <expresion> ;
<sentencia_if> ::= si (<expresion>) <sentencia>
                 | si (<expresion>) <sentencia> si_no <sentencia>
<sentencia_do_until> ::= hacer <bloque> hasta (<expresion>)
<sentencia_entrada> ::= <nomb_entrada> <lista_identificador> ;
<nomb_entrada> ::= entrada >>
<sentencia_salida> ::= <nomb_salida> <lista_expresiones_o_cadena> ;
<nomb_salida> ::= salida <<
<sentencia_return> ::= retorno <expresion> ;


<expresion> ::= ( <expresion> )
                |  <op_unario> <expresion>
                |  <expresion> <op_binario> <expresion>
```

```
                |  <array_ident>
                |  <constante>
                |  <funcion>

<corchetes_digitos>::= [<lista_digitos>]
                     | [<lista_digitos],<corchetes_digitos>
<corchetes_matriz>::= [<corchetes_digitos>]
<lista_digitos>::= <digito>,<lista_digitos>
                 | <digito>
<op_unario> ::= !
              | ++
              | --
              | +
              | -
<op_binario> ::=  +
                | -
                | *
                | /
                | ==
                | !=
                | <
                | >
                | <=
                | >=
                | &&
<constante> ::= < const_entero> | <const_real> | <const_booleano> | <const_caracter>
| <const_array>

<funcion> ::= <identificador>(<lista_expr>)
            | <identificador> ()

<tipo_basico> ::= entero
                | booleano
                | real
                | caracter

<lista_identificador> ::= <lista_identificador> , <ident_array>
                    | <ident_array>


<idarray>::= <identificador>[<numero>] | <identificador>[<numero>][<numero>]
<ident_array>::=<identificador> | <idarray>
<array>::= <identificador>[<expresion>]
          | <identificador>[<expresion>][<expresion>]
<array_ident>::= <array> | <identificador>

<lista_parametros> ::= <lista_parametros> , <tipo_basico> <ident_array>
                     | <tipo_basico> <ident_array>


<num>::= <num><digito>
        | <digito>
```

```
<digito>::= 0 | 1 | ... | 9
<identificador>::= <identificador><alfanumerico>
                 | <letra>
                 | <identificador>_
                 | _<letra>

<letra>::= a | ... | z
<alfanumerico>::= <alfanumerico> <letra>
                 | <alfanumerico> <digito>
                 | <letra>
                 | <digito>



<lista_expresiones_o_cadena>::= <lista_expresiones_o_cadena> <expresiones_cadena>;
                              | < expresiones_cadena>
<expresiones_cadena>::= <expresion>
                      | <cadena>

<cadena> ::= "cadena de caracteres"
<const_entero>::= <num>
<const_real>::= <num>.<num>
<const_caracter>::= 'caracter'
<const_boolena> ::= verdadero | falso

<lista_expr>::= <lista_expr>, <expresion> | <expresion>
<const_array>:: = [<lista_expr>]
```

Definición de la semántica en lenguaje natural

El programa está compuesto por una cabecera de programa y un bloque.

- La *cabecera* estará denotada por la palabra **principal***
- En el *bloque* tendremos varias partes:
    1. Inicio de bloque: Se iniciará con una palabra, en nuestro caso: **{**
    2. Declaración de variables locales: se declararán las variables locales entre dos marcas de declaración, la de inicio que será: **ini_var** y la de fin: **fin_var**
    3. Declaración de subprogramas: el subprograma (que en nuestro caso es una función)se compone de una cabecera, que será: **funcion**, y de otro bloque interno.
    4. Sentencias, que se componen recursivamente de más sentencias. Una sentencia puede ser un bloque, una asignación entre variables, un condicional, un bucle, una entrada o salida, una llamada a una función, una expresión o una sentencia *return*, que marcaremos como **retorno**. Una **expresión** podrá ser un operador unario, binario, un identificador, una constante , una función o una variable. Podrá estar (o no) entre paréntesis, y terminarán siempre en un **;**

  Si tenemos una lista de parámetros para una función, esta deberá ir entre paréntesis y escrita de la forma: **tipo nombre_variable**, y separadas por comas. El **tipo**, como sabemos podrá ser *entero,real,booleano,caracter*. Si queremos indicar que una variable es un vector, se hará escribiendo detrás de su nombre **[“dimensión”]**. Si queremos que sea una matriz, escribiremos detrás del nombre **[<“dimensión i”] [“dimensión j”]**

Identificación de los tokens

Identificación de las palabras

Las palabras identificadas son las siguientes:

{	=	>	verdadero	ini_var	entero
}	+	<	falso	fin_var	caracter
;	-	>=	principal	hasta	booleano
(	*	<=	si	retorno	real
)	/	==	si_no	hacer	entrada »
,	;	!	salida «	cadena	constante entera
++	–	!=	&&		constante real
[	]				

Identificación de los tokens

Token	Código	Palabra	Atributo
PRINCIPAL	256	principal	
INI_BLOQUE	257	{	
FIN_BLOQUE	258	}	
INI_EXPR	259	(	
INI_TAM	260	[	
FIN_TAM	261	]	
FIN_EXPR	262	)	
COMA	263	,	
PTCOMA	264	;	
OP_BINARIO	265	+	0: * 1: / 2: == 3: != 4: >= 5: >= 6: > 7: < 8: &&
VERDADERO	266	verdadero	
FALSO	267	falso	
TIPO_BASICO	268	entero booleano Caracter real	0: entero 1: booleano 2: caracter 3: real
SI	269	si	
SI_NO	270	si_no	
INI_VAR	271	ini_var	
FIN_VAR	272	fin_var	
FUNCION	273	funcion	
RETORNO	274	return	
HACER	275	hacer	
HASTA	276	hasta	
OP_UNARIO	277	& ! + -	0: & 1: ! 2: + 3: - 4:++ 5:–
IGUAL	278	=	
CONST_ENT	279	[0-9]+	
CONST_R	280	[0-9]+.[0-9]+	
CONST_CAR	281	^[a-zA-Z0-9_]	
ENTRADA	282	entrada »	
SALIDA	283	salida »	
CADENA	284	^[a-zA-Z0-9_]+	