Características del lenguaje

El lenguaje que se nos ha asignado cuenta de los siguientes componentes:

- Según el formato de PASCAL
- Palabras reservadas en inglés
- Listas
- Funciones
- for (formato PASCAL)

Descripción del lenguaje en EBNF

```
                  <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_subprog> <bloque>

<Declar_de_variables_locales> ::= <Marca_ini_declar_variables>
                                  <Variables_locales>
                                  <Marca_fin_declar_variables>
                                |

 <Marca_ini_declar_variables> ::= VAR

 <Marca_fin_declar_variables> ::= ENDVAR

          <Cabecera_programa> ::= PROGRAM <identificador>

           <Inicio_de_bloque> ::= BEGIN

              <Fin_de_bloque> ::= END

          <Variables_locales> ::= <Variables_locales> <Cuerpo_declar_variables>
                                |  <Cuerpo_declar_variables>

    <Cuerpo_declar_variables> ::= <Cuerpo_declar_variables> <lista_identificadores> :
```

```
<tipo_dato>;
                                |

      <lista_identificadores> ::= <lista_identificadores>, <identificador>
                                |  <identificador>


                  <tipo_dato> ::= <tipo_dato_A>
                                |  <tipo_dato_B>

                <tipo_dato_A> ::= INTEGER
                                |  REAL
                                |  CHAR
                                |  BOOLEAN

                <tipo_dato_B> ::= LIST OF <tipo_dato_A>

           <Cabecera_subprog> ::= FUNCTION <identificador> (<Variables_locales>) :
<tipo_dato>;
                                   <bloque>;

           <sentencia_return> ::= RETURN <expresion>

                 <Sentencias> ::= <Sentencias> ; <Sentencia>
                                |  <Sentencia>

                  <Sentencia> ::= <bloque>
                                |  <sentencia_asignacion>
                                |  <sentencia_if>
                                |  <sentencia_while>
                                |  <sentencia_for>
                                |  <sentencia_entrada>
                                |  <sentencia_salida>
                                |  <sentencia_return>

       <sentencia_asignacion> ::= <identificador> := <expresion>

                <sentencia_if> ::= <alternativa_simple>
                                |  <alternativa_doble>

         <alternativa_simple> ::= IF <expresion> THEN <Sentencia>

          <alternativa_doble> ::= IF <expresion> THEN <Sentencia> ELSE <Sentencia>

            <sentencia_while> ::= WHILE <expresion> DO <Sentencia>

              <sentencia_for> ::= FOR <sentencia_asignacion> TO <expresion> DO
<Sentencia>
```

```
          <sentencia_entrada> ::= <nomb_entrada> <lista_variables>

           <sentencia_salida> ::= <nomb_salida> <lista_expresiones_o_cadena>

                  <expresion> ::= ( <expresion> )
                                | <op_unario> <expresion>
                                | <expresion> <op_binario> <expresion>
                                | <expresion> <op_ter2> <expresion>
                                | <expresion> <op_ter1> <expresion> <op_ter2> <expresion>
                                | <identificador>
                                | <constante>
                                | <agregado>
                                | <funcion>


              <nomb_entrada> ::= SCAN

               <nomb_salida> ::= PRINT

            <lista_variables> ::= <identificador>, <lista_variables>
                                | <identificador>

<lista_expresiones_o_cadena> ::= <lista_expresiones_o_cadena> <exp_cad>
                                |

                    <exp_cad> ::= <expresion>
                                | <cadena>

           <lista_expresiones> ::= <expresion>, <lista_expresiones>
                                | <expresion>

                   <agregado> ::= [<lista_constantes>]

            <lista_constantes> ::= <constante> <lista_constantes>
                                | <constante>

                     <funcion> ::= <identificador> (<lista_expresiones>)

                   <op_unario> ::= #
                                | ?
                                | >>
                                | <<
                                | $
                                | +
                                | -
                                | NOT

                  <op_binario> ::= +
                                | *
                                | -
```

```
                    |  /
                    |  --
                    |  **
                    |  %
                    |  @
                    |  AND
                    |  OR
                    |  =
                    |  <>
                    |  >
                    |  <
                    |  <=
                    |  >=

      <op_ter1> ::= ++
      <op_ter2> ::= @


   <constante> ::= '[a-z][0-9][A-Z]'
                 |  [0-9]+
                 |  [0-9]+.[0-9]+
                 |  TRUE
                 |  FALSE

<identificador> ::= [a-Z]([a-Z][0-9])*

       <cadena> ::= "([a-Z][0-9])*"
```

Definición de la semántica en lenguaje natural

1. Introducción

Vamos a realizar una descripción básica de nuestro lenguaje de programación basado en el lenguaje Pascal y añadiendo algunas modificaciones.

2. Estructura de un programa

Objetivo	Definir un nuevo programa
Formato	`PROGRAM <identificador> <bloque>`
Descripción	Para comenzar un nuevo programa debemos indicarlo mediante la palabra clave `PROGRAM`, seguido de un identificador con el nombre que queramos asignarle. Por último, indicaremos el bloque de código que lo compondrá.
Ejemplo	**`PROGRAM`** `ejemplo` `BEGIN`

	`` VAR a, b : Integer; ENDVAR FUNCTION Sumar (n1, n2 : Integer) : Integer; BEGIN return n1 + n2; END a := 1; b := 2; return Sumar (a, b); END ```

a. Encabezamiento

Objetivo	Definir el encabezamiento del programa
Formato	`program <identificador>`
Descripción	En la cabecera del programa se especifican el nombre y parámetros del programa. Es meramente informativa y se sitúa en la primera línea del programa encabezada por la palabra reservada program.
Ejemplo	**`PROGRAM`** `ejemplo;`

b. Definición de un bloque

Objetivo	Definir un nuevo bloque de código
Formato	``` BEGIN [VAR Declaración de variables locales ENDVAR] [Declaración de subprogramas] Sentencias del programa END ```
Descripción	Para indicar la declaración de un nuevo bloque de código, utilizamos la palabra clave `BEGIN`. A continuación, y de manera opcional, pasamos a la declaración de variables locales y subprogramas. Una vez realizadas las declaraciones, introducimos

	el conjunto de sentencias a ejecutar. Por último, cerramos el bloque con la palabra clave `END`.
Ejemplo	**BEGIN** VAR mi_var : Integer; ENDVAR mi_var := 1; return mi_var; **END**

c. Declaración de variables locales

Objetivo	Declarar una o varias variables
Formato	<identificador> (, <identificador>)* : <tipo_dato>;
Descripción	Mediante esta sentencia se pueden declarar una o más variables de diferentes tipos usando los siguientes tipos de datos: Integer : Define una variable de tipo entero Char : Define una variable de tipo carácter Real : Define una variable de tipo real Boolean : Define una variable de tipo lógica
Ejemplo	numero : Integer; letra1, letra2 : Char;

d. Declaración de subprogramas

Objetivo	Declarar un subprograma
Formato	FUNCTION <identificador> ([Declaración de variables]) : <tipo_dato>; <bloque>;
Descripción	Para declarar un subprograma, especificamos su cabecera mediante la

	palabra clave `FUNCTION`, seguida del nombre del subprograma. A continuación especificamos las variables que servirán como argumentos, y por último, el tipo de dato devuelto. A continuación, proporcionamos el bloque de instrucciones que compondrá e subprograma.
Ejemplo	`FUNCTION funcion_ej (a, b : Integer; c : Char) : Integer;` `BEGIN` `  VAR` `  res : Integer;` `  ENDVAR` `  print [c];` `  res = a + b;` `  return res;` `END`

3. **Tipos de datos**
    a. **Integer (Números enteros)**
    b. **Char (Caracteres)**
    c. **Real (Números reales)**
    d. **Boolean (Valores lógicos: TRUE y FALSE)**
    e. **List (explicado en otro apartado)**

4. **Operadores y expresiones**
    a. **Aritméticos**

Nombre	Tipo	Lexema	Operandos	Ejemplo	Resultado
Suma	binario	+	dos valores de tipo entero o real	3 + 3 4.0 + 3.4	Devuelve la suma de los dos valores
Resta	binario	-	dos valores de tipo entero o real	3 - 3 4.0 - 3.4	Devuelve la resta de los dos valores
Multiplicación	binario	*	dos valores de tipo entero o real	4 * 2 5.0 * 2	Devuelve la multiplicación de los dos valores
División	binario	/	dos valores de tipo entero o	4 / 2 5.0 / 2	Devuelve la división de el

			real		primer valor entre el segundo
Módulo	binario	%	dos valores de tipo entero o real	4 % 2 5.0 % 2	Devuelve el resto entero de realizar la división de el primer valor entre el segundo

b. Lógicos

Nombre	Tipo	Lexema	Operandos	Ejemplo	Resultado
AND	binario	AND	dos valores booleanos	TRUE AND FALSE	Devuelve el resultado (TRUE o FALSE) de realizar la operación lógica AND
OR	binario	OR	dos valores booleanos	TRUE OR FALSE	Devuelve el resultado (TRUE o FALSE) de realizar la operación lógica OR
NOT	unario	NOT	valor booleano	NOT TRUE	Devuelve el resultado (TRUE o FALSE) de realizar la operación lógica NOT

Igual	binario	=	dos valores del mismo tipo	3 = 3 3.5 = 3.5 'a' = 'b' [1, 2] = [2, 1]	Devuelve el resultado (TRUE o FALSE) de realizar la comparación de igualdad
Distinto	binario	<>	dos valores del mismo tipo	3 <> 3 3.5 <> 3.5 'a' <> 'b' [1, 2] <> [2, 1]	Devuelve el resultado (TRUE o FALSE) de realizar la comparación de diferencia
Mayor estricto	binario	>	dos valores de tipo entero o real	1 > 10	Devuelve el resultado (TRUE o FALSE) de realizar la operación lógica ">"
Menor estricto	binario	<	dos valores de tipo entero o real	6 < 2	Devuelve el resultado (TRUE o FALSE) de realizar la operación lógica "<"
Mayor o igual	binario	>=	dos valores de tipo entero o real	3 >= 5	Devuelve el resultado (TRUE o FALSE) de realizar la operación lógica ">="
Menor o igual	binario	<=	dos valores de tipo entero o real	3 <= 5	Devuelve el resultado (TRUE o FALSE) de realizar la operación lógica "<="

5. Operadores de entrada y salida

Objetivo	Obtener datos de entrada a través del teclado

Formato	**SCAN** <nombre_variable>;
Descripción	Entrada de datos del programa mediante el uso del teclado. Se captará todo carácter o número introducido hasta pulsar la tecla return.
Ejemplo	**SCAN** x;

Objetivo	Imprimir variables o una cadena por pantalla
Formato	**PRINT** <lista_expresiones_o_cadena>;
Descripción	Es la salida del sistema. Nos permite imprimir por pantalla una lista de expresiones o cualquier cadena de caracteres.
Ejemplo	**PRINT** "Hola mundo!"; x := 10; **PRINT** x;

6. Estructuras de control

a. IF

Objetivo	Introducir una instrucción condicional
Formato	**IF (** <expresión_lógica> **)** **THEN** <sentencia> Con la opción else: **IF (** <expresión_lógica> **)** **THEN** <sentencia_o_bloque_1> **ELSE** <sentencia_o_bloque_2>

Descripción	La expresión que sigue al IF representa la condición a evaluar. Si la condició es TRUE se ejecuta la parte THEN, si es FALSE se ejecuta la parte ELSE o ninguna parte si no la hay. Notar que después de THEN y ELSE no debe escribirse el punto y coma. Si se escribe precediendo a ELSE, se terminará allí la sentencia IF y se producirá un error de compilación.
Ejemplo	`IF seguir` `  THEN` `  BEGIN` `  IF (B <> 1)` `  THEN` `  C := 1` `  ELSE` `  D := 1` `  END;`

b. WHILE

Objetivo	Introducir una instrucción bucle de tipo while
Formato	**WHILE (** <expresión_lógica> **) DO** <sentencia_o_bloque>
Descripción	Es una sentencia de control iterativa. Repite se sentencia de ejecución mientras se cumpla la condición de nuestra expresión lógica. La sentencia que se repite puede ser compuesta. Notar que la expresión se evalúa antes de ejecutar la sentencia. Si el valor inicial es FALSE, no se ejecutará la sentencia ninguna vez. La sentencia que se repite debe modificar el valor de la expresión, si no resulta un bloque sin salida. Para ejecutar un grupo de sentencias, hay que construir una sentencia compuesta con los delimitadores BEGIN y END.
Ejemplo	`WHILE (i < N) DO` `BEGIN` `  potencia := potencia * x;` `  i := i+1` `END`

c. FOR

Objetivo	Introducir una instrucción bucle de tipo for
Formato	**FOR** <sentencia_asignación> **TO** <expresion> **DO** <sentencia_o_bloque>

Descripción	Permite construir bucles repetitivos controlador por un contador inicializado en la sentencia de asignación. La variable de control, el valor inicial y el valor final deben ser todos del mismo tipo ordinal. Las sentencias que se repiten (rango del bucle) no deber modificar el valor de la variable de control.
Ejemplo	`FOR i := 0 TO 10 DO` `PRINT "HOLA";`

7. Listas

Objetivo	**Crear** una variable de tipo lista
Formato	`<identificador> (,<identificador>)*` **: LIST OF** <tipo_basico>**;**
Descripción	No podemos crear listas recursivas. Solo podemos crear listas de tipos básicos. En la misma definición no podemos inicializar la lista.
Ejemplo	`LIST OF integer x;`

Objetivo	**Asignar** valores a una variable de tipo lista
Formato	**<identificador> := [** `<valor>,(<valores>)*` **];**
Descripción	No podemos crear listas recursivas. Solo podemos crear listas de tipos básicos. En la misma definición no podemos inicializar la lista. Los valores de la lista pueden ser valores, variables o constantes.
Ejemplo	`x1 := [1,2,3];`

Nombre	Lexema	Argumento	Ejemplo	Resultado
avanzar	>>	lista	*l* >>	Avanza el cursor en una posición
retroceder	<<	lista	*l* <<	Retrocede el cursor en una posición
cursor al comienzo de la lista	$	lista	$*l*	Lleva el cursor al comienzo de la lista

Se definen los siguientes operadores, teniendo en cuenta que son formas de expresión y no de sentencia, salvo que se indique como sentencia, es decir, se devuelve un nuevo elemento del tipo base de la lista o una nueva lista modificada tras realizar la operación correspondiente.

Nombre	Tipo	Lexema	Operandos	Ejemplo	Resultado
longitud	unario	#	lista	$\#l$	Devuelve el número de elementos de l
elemento actual	unario	?	lista	$?l$	Devuelve el elemento actual de la lista
elemento posición	binario	@	lista y valor	$l@x$	Devuelve el elemento de la posición x
añadir elemento en una posición	ternario	++ y @	lista, valor y posición	$l++x@z$	Devuelve una copia de l con x añadido en la posición z
borrar elemento en una posición	binario	−−	lista y posicion	$l--x$	Devuelve una copia de l con el elemento en la posición x borrado
borrar lista a partir de una posición	binario	%	lista y posición	$l\%x$	Devuelve una copia de l sin los elementos a partir de la posición x
concatenar listas	binario	**	dos listas	$l_1 ** l_2$	Añade los elementos de l_2 en l_1 y devuelve una copia
suma	binario	+	lista y valor	$l+x$	suma de x con cada elemento
			valor y lista	$x+l$	suma de cada elemento con x
resta	binario	−	lista y valor	$l-x$	resta de x con cada elemento
producto	binario	*	lista y valor	$l*x$	producto de x con cada elemento
			valor y lista	$x*l$	producto de cada elemento con x
división	binario	/	lista y valor	l/x	división de x con cada elemento

Tal y como muestra la tabla anterior, los operadores unarios tienen como argumento una lista de cualquier tipo base. Los operadores binarios manejan la lista de tipo base y, bien un elemento que debe ser del mismo tipo base o bien la posición que debe ser de tipo entero o ambos (operador ternario).
Existen tres operadores unarios para la realización del recorrido de una lista. Dos de ellos hacen desplazamientos, uno de retroceso y otro de avance y el tercero sitúa el cursor al comienzo de la lista.
Por último, la concatenación de listas debe actuar sobre listas que posean el mismo tipo base con independencia del número de elementos que en ellas se alojen.

Identificación de tokens

Identificación de lexemas:

identificador	%	,	FOR	BEGIN	<>
constante	@	;	=	END	>

cadena	?	PRINT	RETURN	PROGRAM	<
+	>>	SCAN	FUNCTION	VAR	<=
*	<<	WHILE	LIST OF	ENDVAR	>=
-	$	DO	INTEGER	NOT	‘
/	#	IF	REAL	AND	“
--	(	THEN	CHAR	OR	TO
**	)	ELSE	BOOLEAN	:=	:
++	[	]			

Identificación de lexemas con igual función semántica:

identificador	%	,	FOR	BEGIN	<>
constante	@	;	=	END	>
cadena	?	PRINT	RETURN	PROGRAM	<
+	>>	SCAN	FUNCTION	VAR	<=
*	<<	WHILE	LIST OF	ENDVAR	>=
-	$	DO	INTEGER	NOT	‘
/	#	IF	REAL	AND	“
--	(	THEN	CHAR	OR	TO
**	)	ELSE	BOOLEAN	:=	:
++	[	]			

Nombre	Código	Atributos	Expr. Regular
MASMENOS	256	0: + 1: -	"+" | "-"
OPBIN	257	0: * 1: / 2: -- 3: ** 4: % 6: = 7: AND 8: OR 9: <> 10: > 11: <	"*" | "/" | "--" | "**" | "%" | "=" | "AND" | "OR" | "<>" | ">" | "<" | "<=" | ">="

		12: >= 13: <=	
OPUNA	258	0: ? 1: >>> 2: << 3: $ 4: #	"?" | ">>" | "<<" | "$" | "#"
OPTER1	288		“++”
OPTER2	289		“@”
IDENT	259		[a-Z]+
CONST	260	0: "'[a-z][0-9][A-Z]'" 1: [0-9]+ 2: [0-9]+.[0-9]+ 3: "TRUE" 4: "FALSE"	"'[a-z][0-9][A-Z]'" | [0-9]+ | [0-9]+.[0-9]+ | "TRUE" | "FALSE"
CAD	261		""([a-Z][0-9])*""
PARI	262		"("
PARD	263		")"
CORI	290		“[“
CORD	291		“]”
COMA	264		","
PYC	265		";"
PRINT	268		"PRINT"
SCAN	269		"SCAN"
IF	270		"IF"
THEN	271		"THEN"
ELSE	272		"ELSE"
WHILE	273		"WHILE"
DO	274		"DO"
FOR	275		"FOR"
TO	276		"TO"
TIPOBASICO	277	0: INTEGER 1: REAL 2: CHAR 3: BOOLEAN	"INTEGER" | "REAL" | "CHAR" | "BOOLEAN"
LISTOF	278		"LIST OF"
BEG	279		"BEGIN"
END	280		"END"
VAR	281		"VAR"
ENDVAR	282		"ENDVAR"
PROG	283		"PROGRAM"
FUNCTION	284		"FUNCTION"
RETURN	285		"RETURN"

ASIG	286		":="
DOSPU	287		":"

Definición de la gramática abstracta

```
                  <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_subprog> <bloque>

<Declar_de_variables_locales> ::= <Marca_ini_declar_variables>
                                  <Variables_locales>
                                  <Marca_fin_declar_variables>
                                |

 <Marca_ini_declar_variables> ::= VAR

 <Marca_fin_declar_variables> ::= ENDVAR

          <Cabecera_programa> ::= PROGRAM <identificador>

           <Inicio_de_bloque> ::= BEGIN

              <Fin_de_bloque> ::= END

          <Variables_locales> ::= <Variables_locales> <Cuerpo_declar_variables>
                                | <Cuerpo_declar_variables>

    <Cuerpo_declar_variables> ::= <Cuerpo_declar_variables> <lista_identificadores> :
<tipo_dato>;
                                |

      <lista_identificadores> ::= <lista_identificadores>, <identificador>
                                | <identificador>
```

```
                  <tipo_dato> ::= <tipo_dato_A>
                                |  <tipo_dato_B>

                <tipo_dato_A> ::= INTEGER
                                |  REAL
                                |  CHAR
                                |  BOOLEAN

                <tipo_dato_B> ::= LIST OF <tipo_dato_A>

            <Cabecera_subprog> ::= FUNCTION <identificador> (<Variables_locales>) :
<tipo_dato>;
                                   <bloque>;

            <sentencia_return> ::= RETURN <expresion>

                  <Sentencias> ::= <Sentencias> ; <Sentencia>
                                |  <Sentencia>

                   <Sentencia> ::= <bloque>
                                |  <sentencia_asignacion>
                                |  <sentencia_if>
                                |  <sentencia_while>
                                |  <sentencia_for>
                                |  <sentencia_entrada>
                                |  <sentencia_salida>
                                |  <sentencia_return>

         <sentencia_asignacion> ::= <identificador> := <expresion>

                 <sentencia_if> ::= <alternativa_simple>
                                |  <alternativa_doble>

           <alternativa_simple> ::= IF <expresion> THEN <Sentencia>

            <alternativa_doble> ::= IF <expresion> THEN <Sentencia> ELSE <Sentencia>

              <sentencia_while> ::= WHILE <expresion> DO <Sentencia>

                <sentencia_for> ::= FOR <sentencia_asignacion> TO <expresion> DO
<Sentencia>


            <sentencia_entrada> ::= <nomb_entrada> <lista_variables>

             <sentencia_salida> ::= <nomb_salida> <lista_expresiones_o_cadena>

                    <expresion> ::= ( <expresion> )
```

```
                                | <op_unario> <expresion>
                                | <expresion> <op_binario> <expresion>
                                | <expresion> <op_ter2> <expresion>
                                | <expresion> <op_ter1> <expresion> <op_ter2> <expresion>
                                | <identificador>
                                | <constante>
                                | <agregado>
                                | <funcion>


              <nomb_entrada> ::= SCAN

               <nomb_salida> ::= PRINT

            <lista_variables> ::= <identificador>, <lista_variables>
                                | <identificador>

<lista_expresiones_o_cadena> ::= <lista_expresiones_o_cadena> <exp_cad>
                                |

                    <exp_cad> ::= <expresion>
                                | <cadena>

           <lista_expresiones> ::= <expresion>, <lista_expresiones>
                                | <expresion>

                   <agregado> ::= [<lista_constantes>]

            <lista_constantes> ::= <constante> <lista_constantes>
                                | <constante>

                     <funcion> ::= <identificador> (<lista_expresiones>)

                  <op_unario> ::= #
                                | ?
                                | >>
                                | <<
                                | $
                                | +
                                | -
                                | NOT

                 <op_binario> ::= +
                                | *
                                | -
                                | /
                                | --
                                | **
                                | %
                                | @
```

```
                                |  AND
                                |  OR
                                |  =
                                |  <>
                                |  >
                                |  <
                                |  <=
                                |  >=

                     <op_ter1> ::= ++
                     <op_ter2> ::= @


                  <constante> ::= '[a-z][0-9][A-Z]'
                                |  [0-9]+
                                |  [0-9]+.[0-9]+
                                |  TRUE
                                |  FALSE

               <identificador> ::= <

                        <cadena> ::= "([a-Z][0-9])*"
```

```
<BEG> ::= BEGIN
<END> ::= END
<PROG> ::= PROGRAM
<DOSPU> ::= :
<PYC> ::= ;
<COMA> ::= ,
<TIPOBASICO> ::=
<LISTOF> ::= LIST OF
<FUNCTION> ::= FUNCTION
<RETURN> ::= RETURN
<ASIG> ::= :=
<IF> ::= IF
<THEN> ::= THEN
<ELSE> ::= ELSE
<WHILE> ::= WHILE
<DO> ::= DO
<FOR> ::= FOR
<TO> ::= TO
<PARI> ::= (
<PARD> ::= )
<CORI> ::= [
```

<CORD> ::=]