LENGUAJE Ñ_ MATEMÁTICO

Procesadores de Lenguaje

LAURA DEL PINO DÍAZ– GRADO EN INGENIERÍA INFORMÁTICA – MENCIÓN EN COMPUTACIÓN

ÍNDICE

Tabla de contenido

INTRODUCCIÓN

En este documento definiremos la creación de un nuevo lenguaje de programación denominado Ñ matemático, derivado del lenguaje Ñ presentado anteriormente con el compañero. Este lenguaje estará escrito en español y tendrá una base la cual estará formada por funcionalidades de diferentes lenguajes de programación, entre los que destacan MATLAB, Java y Korn Shell.

Por ello en la siguiente sección modelaremos como va hacer nuestro lenguaje de programación con el objetivo de poder crear nuestro compilador.

DEFINICIÓN DEL LENGUAJE Ñ_MATEMÁTICO

TIPOS DE DATOS

En esta sección definiremos el modelo de nuestro lenguaje. El lenguaje Ñ estará formado por tres tipos de datos:

- **REAL**. Este tipo lo utilizaremos para codificar cualquier número.
- **VECTOR**. Este tipo lo utilizaremos para almacenar un conjunto de números en un espacio de memoria contigua y que podremos referenciar mediante un nombre. El índice de la primera posición es 0.

Tipos	Ejemplos
REAL	5, -1 , 3.05
VECTOR	[1,-2,3.05,-50.10],[H,o,l,a]

DECLARACIÓN DE VARIABLES REALES

Para declarar una variable de tipo "REAL" se puede utilizar las siguientes gramáticas:

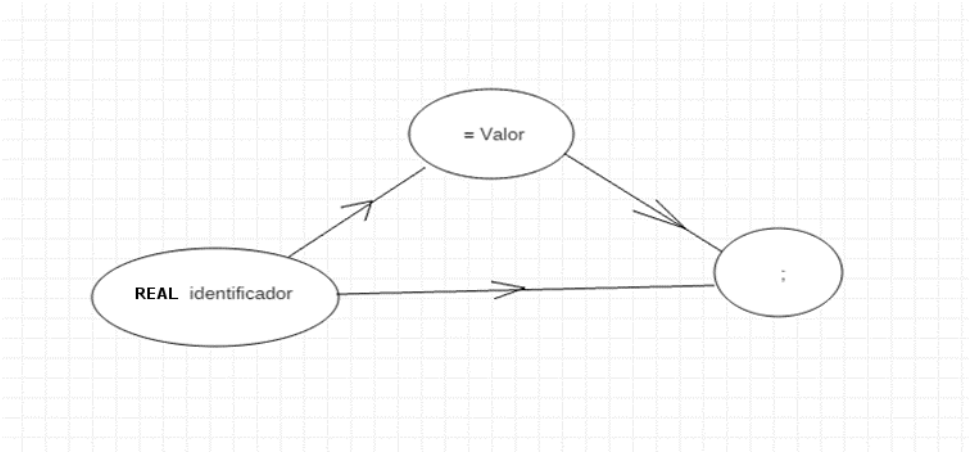

Ilustración 1: declaración de una variable del tipo REAL

Para declarar una variable de tipo "VECTOR" se puede utilizar las siguientes gramáticas:

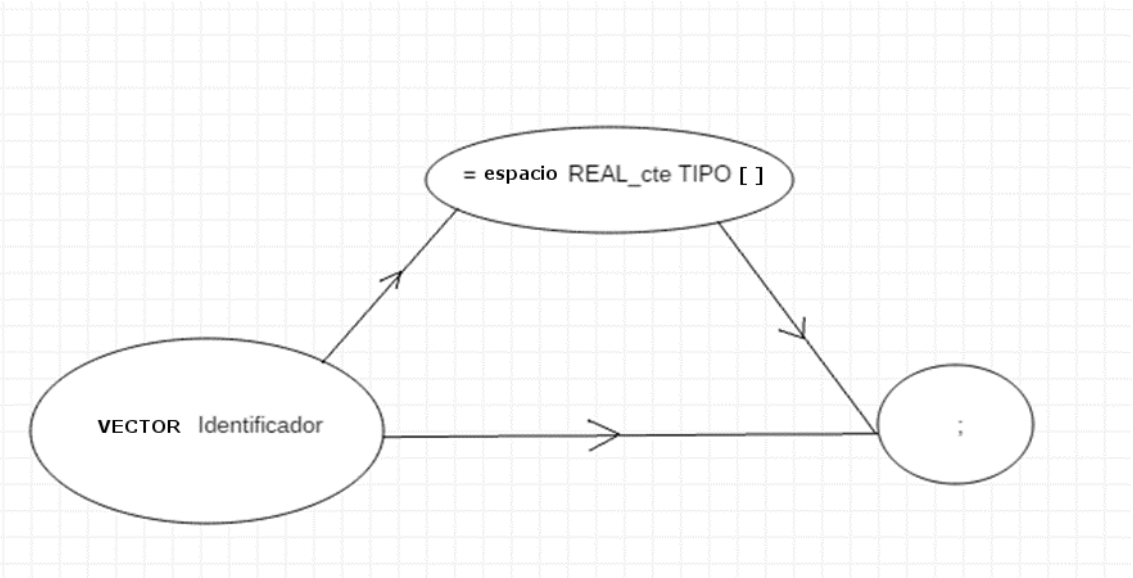

Ilustración 3: Declaración de una variable del tipo VECTOR.

DECLARACIÓN DE VECTORES Y MATRICES

En este lenguaje podremos declarar vector de reales y matrices de dos dimensiones. Todos los tipos de vectores nombrados los podremos declarar y posteriormente "rellenar" con datos. También podremos declarar un vector y la vez indicar que conjunto de datos contiene ese vector. Ambas formas se realizan de la siguiente manera:

- Vector de reales:
  1. **VECTOR** numeros = **espacio** 5 ;
  2. **VECTOR** numeros = **[**1,2,3,4,5**]**;

- Vector de vectores:
  1. **VECTOR** matriz = **espacio** [4] **[**3**]**;
  2. **VECTOR** matriz = **[**1,2,4,6:1,4,5,7:7,8,9,10**]**;

Los vectores carecen de crecimiento dinámico, es decir, el espacio es reservado estáticamente en la declaración.

Para acceder al valor de una posición de un vector de "REAL" hay que tener en cuenta que va desde la posición 0 hasta el tamaño del vector -1, por tanto el primer valor estará almacenado en la posición 0 y así sucesivamente. Por ejemplo de la siguiente manera accederíamos a la posición dos del vector, por tanto, en la variable numero se almacenaría un 3.

```
VECTOR numeros = [1,2,3];

REAL numero = numeros [2]
```

Los vectores de vectores los cuales se utilizan para almacenar matrices tienen una manera distinta de declararse y de acceder a sus posiciones. Cuando declaramos un vector de vectores podremos utilizar las dos formas descritas anteriormente. La que puede traer mayor complicación de entender es la primera por ello la vamos a describir brevemente: **VECTOR** matriz = **espacio [**4**][**3**].** Esto significa que nos vamos a crear un vector con cuatro vectores dentro donde cada vector va a tener tres elementos. Los elementos de los vectores dentro del vector deben ser del mismo tipo de dato. Por ello cada vector dentro del vector será una fila de la matriz, por tanto, el primer vector es la primera fila, el segundo vector es la segunda fila y así sucesivamente. En cuanto al acceso de los vectores dentro del vector se realizarán de la siguiente manera:

```
VECTOR datos  = espacio  [5] [3];

valores[1][2] = 5;
```

Declaramos un vector denominado "datos "con 5 vectores de 3 elementos cada uno. Tras ello asignamos al primer vector dentro de "datos" en la posición 2 el valor 5, por tanto, el primer corchete lo utilizamos para indicar el vector y el segundo corchete para indicar la posición dentro de ese vector.

Finalmente resaltar que no se pueden redimensionar el tamaño de los vectores por tanto siempre tendrán el mismo tamaño con el que se definió.

OPERADORES ARITMÉTICOS

En este lenguaje dispondremos de un conjunto de operadores bastante grandes, lo que nos permitirá realizar prácticamente cualquier operación de manera directa. Dispondremos de operadores aritméticos, relacionales y lógicos.

Los operadores aritméticos los utilizaremos para realizar operaciones de tipo numérico por tanto serán utilizados por el tipo “REAL” o un “VECTOR con un conjunto de números.

Operador	Descripción	Ejemplo
+	Suma	5 + 2 = 7
-	Resta	5-3 = 2
*	Producto	3 * 5 = 15
/	División	5/2.5 = 2

Para cambios de signo de variables se recomienda el uso de una construcción del tipo ‘0-variable’.

OPERADORES RELACIONALES

También tendremos operadores relaciones los cuales utilizaremos para realizar una comparación numérico o de caracteres entre dos operandos. Exactamente los caracteres utilizarán únicamente los operadores de igual o distinto y solo podrán ser dos caracteres los que se comparen por tanto no se podrán comparar números y caracteres al mismo tiempo. Además no son iguales dos caracteres iguales pero uno en mayúscula y otro en minúscula. Tampoco se podrá utilizar el vector completo como operando pero si los elementos que componen dicho vector. El resultado de la evaluación de cualquier operador relacional será 1 o 0, 1 si se cumple esa comparación y 0 sino se cumple.

Operador	Descripción	Ejemplo/Resultado
==	Igual	5 == 5 /1
!=	Distinto	‘a’ != ‘b’ /0
>	Mayor	5 > 0 /1
<	Menor	5 < 4 /0
>=	Mayor o igual	5 >= 2 / 1
<=	Menor o igual	10 <= 5 / 0

OPERADORES LÓGICOS

Los operadores lógicos se aplican en operaciones relacionales con el objetivo de poder evaluar varias operaciones relacionales simultáneamente. En el caso de que se cumpla la condición el resultado será un 1, en caso contrario será un 0.

Operador	Descripción	Ejemplo/Resultado
&&	AND	(5 == 5 && 'A' == 'a') / 0
||	OR	(2 !=1 || 10.5 >= 2) / 1
!	NOT	!(2 == 1)/1

OPERADORES Y OPERACIONES

El orden en el que se evaluarán los operadores aritméticos es el siguiente de izquierda a derecha (todos aquellos que están en la misma columna pertenecen al mismo nivel, si hay que evaluar varios de la misma columna se sigue el orden de aparición de izquierda a derecha). Por tanto la prioridad de los operadores aritméticos es primero los paréntesis y posteriormente los operadores que se encuentran en el primer nivel, segundo nivel y tercer nivel.

Primer Nivel	Segundo Nivel	Tercer Nivel
./	.+	.%
/	+	%
.*	.-	
*	-	

El orden en el que se evaluarán los operadores relacionales es el siguiente (siguiendo las mismas reglas que en el caso anterior):

Primer nivel	Segundo nivel	Tercer nivel
>=	>	==
<=	<	!=

En el caso de los operadores lógicos se sigue su orden de aparición.

Ante el caso de que haya operadores de todos los tipos mencionados anteriormente se evaluarán los operadores aritméticos en primer lugar, seguidos por los relacionales y en último lugar los lógicos.

Todas las operaciones anteriores se pueden realizar con las siguientes combinaciones de tipos en el orden en el que aparecen a continuación:

- Escalar <operación> Escalar
- Escalar <operación> Vector
- Vector <operación> Vector
- Matriz <operación> Matriz
- Vector <operación> Matriz

SENTENCIAS CONDICIONALES

En el lenguaje Ñ dispondremos de dos sentencias condicionales, la sentencia si-sino para condiciones simples y la sentencia dependeDe para condiciones múltiples.

La sentencia "si-sino" (o si-soloSi-sino) nos permite decidir entre dos posibles opciones excluyentes. La expresión que acompaña al "si", la cual tiene que ser una operación relacional o lógica, produce al ser evaluada un 0 o 1 dependiendo de si se cumple o no. En el caso que se cumpla, se ejecutarán el conjunto de sentencias que se encuentra dentro del bloque "si". En caso contrario se ejecutarán las sentencias que se encuentran dentro del bloque "sino" si está ya que dicho bloque es opcional. También es posible tener sentencias "soloSi" seguido de una condición, que permite evaluar más condiciones, no solo la condición que va precedida del "si". Al igual que el bloque "sino", el bloque "soloSi" es opcional.

La sintaxis es la siguiente y como dijimos anteriormente la parte del "sino" es opcional. Dentro de la sintaxis veremos la palabra sentencias la cual no referimos a ninguna, una o más de una sentencia. Esta palabra (sentencias) será utilizada posteriormente para la explicación de otras sintaxis y tendrá el mismo significado.

```
si ( expresión){
    sentencias;
}soloSi(expresión){
    sentencias;
}sino{
    sentencias;
}
```

La sentencia “dependeDe” (en otros lenguajes switch) nos permite decidir entre múltiples opciones excluyentes. El funcionamiento es muy sencillo se evalúa la expresión y en caso de coincidir el valor de la expresión con el valor de una de las ramas “caso” se ejecuta el conjunto de sentencias que sigue hasta el final del “dependeDe” o hasta encontrarse la sentencia “fin” (break en otros lenguajes). Tanto el valor de la expresión como el valor de las ramas debe ser un entero, en caso de que no sea un entero se producirá un error de compilación. Lo normal es que después de cada “caso” o rama haya un “fin” ya que si no se ejecutarían sentencias que no corresponde con dicho “caso”. Además hay un “caso” denominado “porDefecto” el cual es opcional y se ejecuta si no se encuentra coincidencia entre el resultado de la expresión y los casos que se encuentra en el “dependeDe”.

Existe la restricción debe que debe aparecer primero los “caso” y después el “porDefecto”. Además todas las opciones deben ser diferentes.

```
dependeDe (expresion){
    caso valor1:
        sentencias;
    fin;
    caso valorK:
        sentencias;
    fin;
    porDefecto:
        sentencias;
}
```

BUCLE MIENTRAS

Este tipo de bucle nos permite ejecutar un conjunto de sentencias repetitivamente mientras se cumpla una determinada condición. Una característica del bucle "mientras" es que se ejecuta 0 o N veces ya que si la condición del bucle no se cumple no se entra a ejecutar las sentencias.

Por ello su funcionamiento es muy sencillo, se evalúa la expresión (formada por operadores lógicos y relacionales) que está en "mientras" y si como resultado se obtiene un 1, se ejecutan el conjunto de sentencias que se encuentran en el interior del "mientras". Esto se repetirá hasta que la evaluación de la expresión de un 0, por tanto, se deje de cumplir la condición.

La sintaxis de "mientras" es la siguiente:

```
mientras (expresión) {
    sentencias;
}
```

BUCLE PARA

El bucle "para" en su versión completa se utiliza para ejecutar un conjunto de sentencias un número determinado de veces el cual se fija en el interior del bucle.

La sintaxis del "para" es la siguiente:

```
para (asign1;exp1; asign2) {
    sentencias;
}
```

Pueden omitirse cualquier de las dos asignaciones y la expresión del bucle for, pero los puntos y coma deben permanecer. Tanto asign1 como asign2 son asignaciones

mientras exp1 es una expresión condicional. En el caso de que no exista exp1 se considera que la condición es siempre cierta.

La asign1 se utiliza para inicializar la variable que controla el bucle, en su defecto se asumirá que la variable ya está creada en los bloques que contienen al bucle para, con exp1 controlamos la permanencia en el bucle y con asign2 realizamos modificaciones sobre la variable que controla el bucle para poder llegar a salir de éste.

Además tendremos un bucle "para" implícito el cual nos va permitir crear vectores con un conjunto determinados de números. Su sintaxis será la siguiente

```
VECTOR  identificador = inicio:inc:fin
```

Inicio nos indica en que número se empieza, inc el incremento que se va a realizar, y fin hasta que número vamos a llegar. Inc se puede obviar y en dicho caso el incremento será de uno en uno.

Este bucle implícito también se puede utilizar para recorrer los vectores de la siguiente forma:

```
para vector[ REAL valor = 0: final]{

     sentencias;

}
```

Donde la etiqueta "final" indica que se va a recorrer todas las posiciones del vector, de esta forma no hay que saber el número de elementos del vector. Además asignamos a una variable "REAL", en nuestro caso "valor", la posición actual del vector en la que nos encontramos.

ESTRUCTURA DE UN PROGRAMA

Para elaborar un programa en Ñ, todo el código se debe definir en el mismo fichero de texto. Este fichero de texto podrá contener únicamente la función de inicio del programa.

La función de "inicio" está definida de la siguiente forma:

```
func inicio{

    sentencias;

    [devuelve];

}
```

La peculiaridad de esta función es que no devuelve ningún tipo de dato, y que recibe por parámetros todos los caracteres de la línea de comandos, debe ser el usuario quien controle donde termina cada uno de los parámetros que se le pasan y de pasarlo al tipo de datos adecuado.

ENTRADA-SALIDA

En Ñ existen las palabras reservadas “lee” y “escribe” que corresponden a artilugios del lenguaje de programación que permiten leer de teclado y escribir en pantalla.

“lee” nos permite introducir por pantalla el valor de una variable. Nos devolverá un vector de caracteres que tendremos que interpretar para obtener el valor numérico, en caso de que lo que pidiésemos al usuario fuese un REAL.

“escribe” nos permite mostrar por pantalla un texto y el valor de una variable, tras mostrarlo se produce un salto de líneas. Por ejemplo si queremos solo mostrar un texto ponemos escribe (“Hola esto es una prueba”). En el caso de querer mostrar un texto más una variable utilizamos el “+”. Por ejemplo escribe (“Prueba” + identificador). Finalmente si queremos mostrar el valor de distintas variables más texto se realiza mediante la concatenación con el “+”. Por ejemplo escribe (“Valor” + identificador + “posicion” + identificador_1).

En el caso de hacer uso de vectores para la lectura de un conjunto de valores se tendrá que declarar el vector como en los otros tipos e introducir los datos de la siguiente manera:

- Si queremos introducir un conjuntos de números debemos separarlos por espacios. De la siguiente manera: 1 2 3 5.4 23.5
- En el caso introducir un conjunto de caracteres se realizará de esta manera sencilla: Hola mundo
- Finalmente para introducir matrices se realizará de distinta forma la cual explicaremos con un ejemplo sencillo.

```
func nulo inicio (VECTOR letras_de_comando){

        VECTOR matriz = espacio [4][3]; //MATRIZ 4x3

        matriz = lee();

}
```

Cuando vayamos a introducir la matriz lo haremos por filas, es decir, introducimos los valores de la primera fila y pulsamos enter, así hasta rellenar la matriz. En nuestro

ejemplo pondríamos los 3 valores de la primera fila y daríamos a enter, así hasta rellenar la matriz. En el caso de no introducir una fila correctamente se producirá un error.

Cuando vayamos a mostrar por pantalla un vector se realizará de la misma manera que los otros tipos de datos, es decir llamando al identificador del vector. En este caso nos mostrará por pantalla todos los datos que contiene el vector ya pueden ser números o caracteres.

IDENTIFICADORES DE VARIABLES

Para determinar el identificador de una variable se puede utilizar cualquier cadena de caracteres que no incluya números al comienzo del mismo y que no coincida con ninguna de las palabras reservadas del lenguaje Ñ ni ningún símbolo especial.

COMENTARIOS

Los comentarios en el lenguaje Ñ se podrán realizar de dos maneras. La primera sería poniendo en una línea el símbolo "#", esto significaría que desde ese punto hasta el final de la línea es un comentario. En los programas de prueba podremos ver varios ejemplos de comentarios.

LOS PROGRAMAS DE PRUEBA ESTÁN A LA ESPERA DE ACTUALIZACIÓN PUESTO QUE SE ESCRIBIERON PARA EL LENGUAJE DE PROGRAMACIÓN Ñ

PROGRAMAS DE PRUEBA

En este apartado expondremos una serie de programas de prueba con el objetivo de entender el lenguaje Ñ a la perfección. Además realizaremos una micro aplicación en lenguaje Java y la "traduciremos" al lenguaje Ñ.

PROGRAMA DE DECLARACIONES DE VARIABLES Y OPERADORES

En este programa se demostrará como realizar todos los tipos de declaraciones e inicializaciones de funciones junto con un subconjunto de operadores así como se utilizan los comentarios.

```
func nulo inicio (VECTOR letras_del_comando){

        REAL num;

        num = 1 || 0 ; # num = 1

        num = 0 – num ; #num = -1

        VECTOR  vec = [2,3,4,5];

        VECTOR vec2 = vec * num; #vec2=[-2,-3,-4,-5]

        VECTOR vec3 = vec2 .*vec1; #vec3=[-4,-9,-16,-25]

        VECTOR vec4 = espacio 4 LETRA; #vec4=['\0','\0','\0','\0']

        LETRA nula = vec4[0]; #nula= '\0'

        LETRA hache = 'h';

        vect4[0] = hache; #vect4['h' ,'\0','\0','\0']

        vect4[1:final] = ['o','l','a']; #vect4 ['h','o','l','a']

}
```

PROGRAMA PARA LEER DE TECLADO Y ESCRIBIR EN PANTALLA

En este programa se mostrará cómo se lee un vector de teclado y como se muestra por pantalla.

```
func nulo inicio (VECTOR letras_de_comando){

        VECTOR teclado = lee();

        escribe(teclado);

}
```

Este ha sido un caso sencillo en donde no se ha interpretado lo que se ha leído del teclado sino que se ha pasado directamente el vector a la salida. Si quisiéramos hacer una interpretación de lo que se lee en un número "REAL" deberíamos optar por una solución como la que se muestra en la sección "PROGRAMAS DE LLAMADAS A FUNCIONES" que se encuentra entre las últimas secciones de programas de ejemplo.

Pero pongamos que el valor que nos interesa mostrar en pantalla no es un simple vector de caracteres como en el ejemplo anterior sino que es el resultado del producto punto del programa de ejemplo de declaración de variables. El código sería el siguiente:

```
func nulo inicio(VECTOR letras_del_comando){

        REAL num;

        num = 1 || 0 ; # num = 1

        escribe(num);

        num = 0 – num ; #num = -1

        escribe(num);

        VECTOR  vec = [2,3,4,5];

        VECTOR vec2 = vec * num; #vec2=[-2,-3,-4,-5]

        VECTOR vec3 = vec2 .*vec1; #vec3=[-4,-9,-16,-25]

        escribe(vec3);

}
```

He colado a propósito las líneas de donde se muestra por pantalla el valor de la variable “num” para ilustrar que no solo se pueden escribir por pantalla vectores sino que también se pueden escribir valores de variables.

Lo más importante de estos dos ejemplos es demostrar que se puede pasar cualquier tipo de parámetros a la función “escribe” que imprimirá su valor, mientras que la función “lee” solo devolverá un vector de caracteres.

PROGRAMAS DE SENTENCIAS CONDICIONALES

En este primer programa de sentencias condicionales veremos cuando se ejecuta el bloque se sentencias dentro del si o las del sino en el caso de que hubiera. En este caso leemos por consola una edad e indicamos si con esa edad eres mayor de edad o no.

```
func  nulo inicio(VECTOR letras_del_comando){
   # Programa que indica si una persona es mayor de edad.
   REAL edad = 18;
   si ( edad >= 18){
       escribe (“Eres mayor de edad”)
    }sino{
       escribe(“No eres mayor de edad”)
    }
}
```

Este siguiente programa nos muestra cómo se utiliza la sentencia condicional "dependiendoDe". Este programa tiene la función de imprimir pon pantalla si el número almacenado en la variable "numero" es un uno, un dos o es otro número.

```
func  nulo inicio(VECTOR letras_del_comando){
    REAL numero  = 2;
  dependiendoDe (numero){
     caso 1:
          escribe ( "Hay un uno");
          fin;
      caso 2 :
          escribe ( "Hay un dos");
           fin;
      porDefecto:
           escribe ("Este número no es un uno ni un dos");
   }
}
```

PROGRAMA DE BUCLES

En este programa veremos cómo se puede hacer uso de dos bucles que existen en el lenguaje Ñ. Para ello iniciaremos un vector de números con valores del 2 al 10 e incrementando de dos en dos. Tras ello vamos recorriendo el vector e imprimimos por pantalla la posición y el valor de esa posición del vector.

```
func  nulo inicio(VECTOR letras_del_comando){
    VECTOR  numeros = 2: 2:10
    para numeros[ REAL posicion = 0: final]{
        escribe ("En la posición " + posición + "esta el número" + numeros[posición]);
    }
 }
```

En el siguiente programa veremos cómo se puede hacer uso se puede hacer uso del bucle mientras y para. En ambos casos inicializaremos una variable real y la iremos aumentando proporcionalmente hasta alcanzar un valor concreto, por tanto, veremos cómo realizar la misma tarea de diferente forma.

```
func  nulo inicio(VECTOR letras_del_comando){

    para ( REAL contador = 0 ; contador <= 10; contador =  contador + 1){

         escribe (“Valor del contador ” + contador );

     }

     contador = 0;

     mientras (contador != 10){

         escribe (“Valor del contador ” contador);

     }

 }
```

PROGRAMA DE LLAMADAS A FUNCIONES

A continuación ponemos como ejemplo de realización de llamadas a funciones, y un poco como conglomeración de todo el lenguaje Ñ un programa que lo que hace es reinterpretar un vector de caracteres como un real:

```
        func nulo inicio(VECTOR args)  {


VECTOR case1 = ['1'];

VECTOR case2 = ['1','0','3','0'];

VECTOR case3 = ['-3','2'];

VECTOR case4 = ['2','.','1'];

VECTOR case5 = ['-','3','3','5'];



                REAL d;

                d = pl_parseDouble(case1);

                d =pl_parseDouble(case2);

                d =pl_parseDouble(case3);

                d =pl_parseDouble(case4);

                d =pl_parseDouble(case5);


            }}



        func REAL pl_parseDouble(VECTOR case1)  {

                REAL posicion_coma = encuentra_posicion_coma( case1);

                REAL signo_negativo = es_negativo(case1);

                REAL parte_parte_entera = 0;

                REAL parte_decimal = 0;

                REAL index = 0;

                REAL result = 0, aux = 1, aux2 = 0;

               index = signo_negativo;
```

```
       mientras(case1[index] != '\0') { #el símbolo ‘\0’ indica el final del vector
            si(index < posicion_coma) {
                si(index > 0) {
                 result = result * 10;
          }
                result = result + dameValor(case1[index]);
           }sino si(index > posicion_coma) {
                aux = dame_potencia_de_diez(index - (posicion_coma+1));
                aux2 = aux2 + dameValor(case1[index]) * aux;
           }
           index = index + 1;
          }
        parte_parte_entera = result;
        parte_decimal =  (aux2 / dame_potencia_de_diez(index-(posicion_coma+1)));

        si (signo_negativo == 0) {
                devuelve parte_parte_entera + parte_decimal;
        sino  {
                devuelve departe_parte_entera + parte_decimal;  }    }

 func REAL encuentra_posicion_coma(VECTOR case1)  {
        REAL index = 0;
        mientras(case1.[index] != '  0')
         {
           if(case1[index] == '.') return index;
           index = index + 1;
         }
        return 274000000000; #Integer.MAXVALUE
     }

func REAL es_negativo(VECTOR case1)  {
        si (case1[0] ==  '91- '91)  {
        devuelve 1;
```

```
          }sino {
        devuelve 0;
         }
    }


func REAL dame_potencia_de_diedevolverz( REAL i)  {
        REAL returnable = 1;
        para (int j = 0; j < i ; j++)  {
            returnable = returnable * 10;
         }
        devuelve returnable;
    }
```

```
func REAL dameValor(LETRA charAt)  {
      dependiendoDe(charAt) {
          caso '1': devuelve 1;
          caso '2': devuelve 2;
          caso '3': devuelve 3;
          caso '4': devuelve 4;
          caso '5': devuelve 5;
          caso '6': devuelve 6;
          caso '7': devuelve 7;
          caso '8': devuelve 8;
          caso '9': devuelve 9;
          caso '0': devuelve 0;
       }
      devuelve -1;
   }
```

PROGRAMA: CONTEO DE COINCIDENCIAS

Para mostrar la funcionalidad completa del lenguaje Ñ, hemos desarrollado una pequeña aplicación cuya función es la del conteo de coincidencias de una palabra en otra. Para ello hemos desarrollado dos versiones de la aplicación una con el algoritmo iterativo y otro con el algoritmo recursivo. Ambas versiones se desarrollaron y probaron en Java previamente, dichos programas en Java se adjuntan.

```
func nulo inicio (VECTOR comando){
        escribe(“Introducir un texto: ”);
        VECTOR contenedor = lee();
        escribe(“Introducir un patron: ”);
        VECTOR buscado = lee();
        REAL frecuenciaPalabrasRecursiva = frecuencia_recursiva(salida buscado,salida
        contenedor);
```

```
        REAL frecuenciasPalabrasIterativo = frecuencia_iterativa(buscado,contenedor);

        escribe("Numero de coincidencias encontradas con algoritmo recursivo: " +
        frecuenciaPalabrasRecursiva);

        escribe("Numero de coincidencias encontradas con algoritmo iterativo: " +
        frecuenciaPalabrasIterativo);

}
```

#ALGORITMO RECURSIVO

```
func REAL frecuencia_recursiva(VECTOR salida buscado, VECTOR salida contenedor, REAL
ind_buscado, REAL ind_contenedor){

        si(contenedor[ind_contenedor] == '\0'){

                si (buscado[ind_buscado] == '\0'){

                        devuelve 1;

                }

                devuelve 0;

        }

        Si (buscado[ind_buscado] == '\0'){

                devuelve 1 + frecuencia_recursiva(salida buscado, salida contenedor, 0,
        ind_contenedor+1);

        }

        Si(buscado[ind_buscado] != contenedor[ind_contenedor]){

                devuelve 1 + frecuencia_recursiva(salida buscado, salida contenedor, 0,
ind_contenedor+1);

        }

        Si (buscado[ind_buscado] == contenedor[ind_contenedor]){

                devuelve frecuencia_recursiva(salida buscado, salida contenedor, ind_buscado +
1, ind_contenedor+1);



}


func REAL frecuencia (VECTOR salida buscado, VECTOR salida contenedor){

        devuelve frecuencia_recursiva(salida buscado, salida contenedor,0,0);

}
```

#ALGORITMO ITERATIVO

```
func REAL frecuencia_iterativa(VECTOR salida contenido, VECTOR salida conteniente){
```

```
    REAL posicionFrase;
    REAL contadorDePalabras = 0;
    REAL posicionPala
    para conteniente[ posicionFrase = 0: final]{
        para contenido[posicionPalabra = 0:final]{
            si(conteniente[posicionPalabra+posicionFrase] != contenido[posicionPalabra]){
               fin;
            }
         }
         si(contenido[posicionPalabra] == '/0'){
             contadorDePalabras = contadorDePalabras + 1;
         }
      }
      devuelve contadorDePalabras;
}
```

El algoritmo recursivo también nos permite ilustrar el paso de parámetros por referencia para evitar tener variables globales para el conteo ni múltiples copias en la pila para un mismo valor.

•