Construcción del lenguaje ñ

LAURA DEL PINO DÍAZ Y AITOR DEL PINO SAAVEDRA HERNÁNDEZ – GRADO EN INGENIERÍA INFORMÁTICA – MENCIÓN EN COMPUTACIÓN

Procesadores de Lenguaje

ÍNDICE

|  |  |
| --- | --- |
| Introducción | 2 |
| Definición del lenguaje Ñ | **3** |
|  |  |
|  |  |

INTRODUCCIÓN

En este documento definiremos la creación de un nuevo lenguaje de programación denominado Ñ. 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 en C.

DEFINICIÓN DEL LENGUAJE Ñ

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.
* **PALABRA**. Este tipo nos va a permitir poder representar lo caracteres del código ASCII.
* **VECTOR**. Este tipo lo utilizaremos para almacenar un conjunto de caracteres o 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 |
| **LETRA** | ‘A’,’b’,’5’,’-‘,’\*’ |
| **VECTOR** | [1,-2,3.05,-50.10],[H,o,l,a] |

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

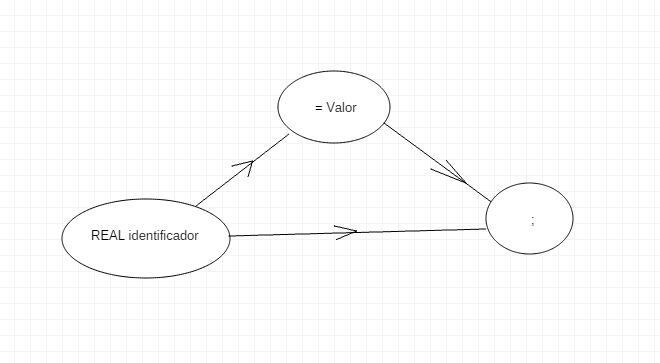


Ilustración 1: declaración de una variable del tipo REAL

Para declarar una variable de tipo “LETRA” se puede utilizar las siguientes gramáticas:

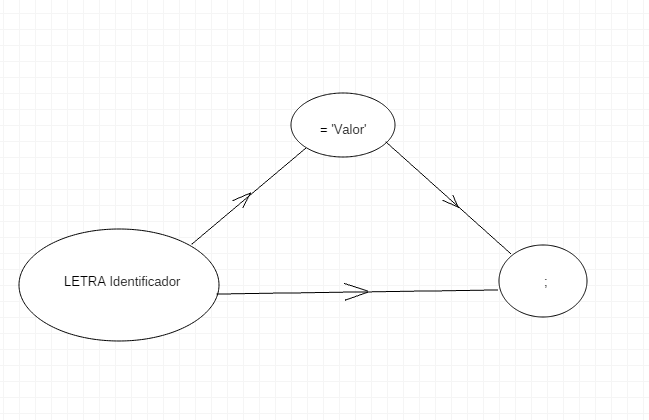


Ilustración 2: Declaración de una variable del tipo LETRA.

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

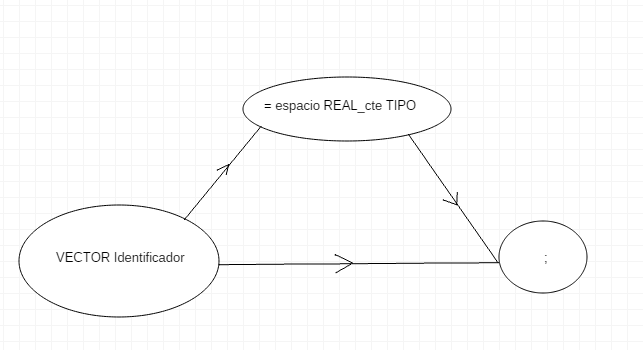


Ilustración 3: Declaración de una variable del tipo VECTOR.

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.

OPERADORES ARITMÉTICOS

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 |
| .+ | Suma punto | [1,2] .+ [2,3] = [3,5] |
| - | Resta | 5-3 = 2 |
| .- | Resta punto | [1,3.5,2] .– [1,0,-3] = [0,3.5,-1] |
| \* | Producto | 3 \* 5 = 15 |
| .\* | Producto punto | [1,2,5].\*[1,2,3] = [1,4,15] |
| / | División | 5/2.5 = 2 |
| ./ | División punto | [10,15,30] ./ [2,3,10] = [5,5,3] |
| % | Módulo | 4%2 = 0 |
| .% | Modulo punto | [4,3,1] .% [4,3,1] = [0,0,0] |

Las operaciones de módulo y de módulo punto deben realizarse con números enteros. En el caso de utilizarse números con decimales se redondearán con el objetivo de obtener números enteros. Por ello si la primera cifra decimal es mayor o igual que cinco se sumará uno a la cifra anterior sino se mantendrá dicha cifra anterior.

También podremos realizar sumas, restas, multiplicaciones y divisiones de un valor por un vector, de manera que el resultado será realizar la operación de ese valor por todos los valores que componen el vector o viceversa, por tanto se cumple la propiedad conmutativa. Por ejemplo: 5 + [1,2,3] = [6,7,8] , [1,4,5] + 5 = [6,9,10].

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 |

PRECEDENCIA DE OPERADORES

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.

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 (en el lenguaje Ñ si-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.

La sintaxis es la siguiente y como dijimos anteriormente la parte del else 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;

}**sino**{

sentencias;

}

sentencia-k;

}else{

sentencia-k;

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 switch.

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

**dependiendoDe** (expresion){

**caso** valor1:

sentencias;

**fin**;

**caso** valorK:

sentencias;

**fin**;

**porDefecto:**

sentencias;

}

break;

}

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 while. 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 ninguna o tantas funciones como se desee además de la función de inicio del programa. Todas las funciones definidas por el usuario serán definidas antes que la función de inicio.

Así pues para definir una función se utilizará la siguiente estructura:

**func** tipo\_a\_devolver identificador (**[**TIPO **[salida]** identificador\_variable , TIPO **[salida]** identificador\_variable2 ...**]**){

sentencias;

**devuelve** **[**identificador\_variableX**]**;

}

Donde “tipo\_a\_devolver” indica el tipo de dato que va retornar la función, identificador es el nombre que define el usuario para la función, TIPO es el tipo de variable que recibe de los definidos anteriormente, “salida” indica si la variable es de salida (por lo que se está pasando su referencia) o en caso de que la etiqueta “salida”esté ausente se pasa el valor solamente.

Tras ello se ejecutan todas las sentencias definidas por el usuario dentro de ese bloque. Como norma general se tiene que especificar el tipo de dato devuelto, en caso de no querer devolver ningún tipo de datos se utiliza la etiqueta “void”en el “tipo\_a\_devolver”. Además tenemos la etiqueta “devuelve”la cual se utiliza para indicar el dato de la variable que vamos a devolver. El dato que se encuentra dentro de esta variable tiene que ser del mismo tipo que se ha puesto en el “tipo\_a\_devolver”. También la etiqueta “devuelve” puede aparecer varias veces en una función como por ejemplo dentro de una sentencia condicional y al final de una función. Lo único restrictivo es que si devolvemos un tipo de dato en una función debe haber alguna sentencia “devuelve”que se ejecute siempre. Por último indicar que si se utiliza la etiqueta “void”la etiqueta “devuelve”se utiliza para finalizar la ejecución de la función en casos concretos y no es obligatorio utilizarlo. En dicho caso como no vamos a devolver ningún tipo de dato solo se pone “devuelve”.

La función de “inicio” está definida de la siguiente forma:

**func** **void inicio**(**VECTOR** letras\_del\_comando){

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.

Para llamar desde la función de inicio a las funciones definidas anteriormente, tenemos que utilizar la siguiente construcción:

[TIPO] [identificador = ] identificador\_de\_función ([lista\_de\_parámetros])

Cuando llamemos a las funciones tendremos que pasarle el mismo número de parámetros y del mismo tipo que cuando se definió.

Las variables que se utilicen en un programa en Ñ serán todas locales, para facilitar la lectura y comprensión del código. Para aquellas variables que en otros lenguajes se consideran “variables globales” animamos al programador a seguir el patrón de ingeniería del software denominado “inyección de dependencia” mediante el uso del paso por referencia.

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. Por ejemplo REAL edad = lee (). Siempre debemos indicar el tipo de datos que vamos a introducir por pantalla.

“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).

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.

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** **void inicio**(**VECTOR** letras\_del\_comando){

escribir ( “Introducir una edad”);

**REAL** edad = lee();

si (edad >= 18){

escribir (“Eres mayor de edad”)

}else{

escribir(“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** **void inicio**(**VECTOR** letras\_del\_comando){

**REAL** numero = 2;

dependiendoDe (numero){

caso 1:

escribir ( “Hay un uno”);

fin;

caso 2 :

escribir ( “Hay un dos”);

fin;

porDefecto**:**

escribir (“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**  **void inicio**(**VECTOR** letras\_del\_comando){

**VECTOR** numeros = 2: 2:10

para numeros[ **REAL** posicion = 0: final]{

escribir (“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 while 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 función.

**func**  **void inicio**(**VECTOR** letras\_del\_comando){

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

escribir (“Valor del contador ” + contador );

}

contador = 0;

while (contador != 10){

escribir (“Valor del contador ” contador);

}

}