1 Introducción. Primer contacto con NUEZO

Nuestro lenguaje de programación se llama NUEZO. Para crearlo, nos hemos inspirado en `C++`, `Java` y `Python`, intentando recopilar lo que más nos gustaba de cada uno. Hemos intentado crear una sintaxis intuitiva y accesible.

A la función principal (`main`) la hemos llamado `engine`, porque impulsa el código. Será la primera en llamarse al ejecutar el programa y devolverá un número para que pueda indicar si ha ido todo bien durante la ejecución (0 en caso de exito). Deberá estar siempre definida y no recibirá ningún parámetro.

Al implementar una función, debes especificar primero el tipo del valor que devuelve o, en caso de no devolver nada (`void`) debes escribir `silent`, como si la función no "hablara" de vuelta. Después tienes que escribir el nombre de la función y los argumentos que recibe entre paréntesis. Para cada parámetro, hay que especificar su tipo y su nombre, al igual que en `C++`. Además, se debe añadir el símbolo `&` detrás del tipo del argumento en el caso de que este se desee pasar por referencia.

En el caso de una función cuyo tipo no sea `silent`, no es obligatorio utilizar el valor devuelto, pudiendo aprovechar los cambios que realice la función descartando su resultado.

En cuanto a las instrucciones de retorno, se hacen con la palabra reservada `return` y se podrá ponerle un valor para que retorne (en caso de ser una función que retorna) o no poner nada (en caso de ser de tipo `silent`). Se podrá poner tantas instrucciones de retorno como se desee, dentro de bucles, condicionales, etc (de manera análoga a su funcionamiento normal en lenguajes como `Java`, terminando con la ejecución de la función).

Dentro de una función se podrá hacer una llamada a ella misma, es decir, está permitida la recursión.

Posteriormente veremos ejemplos de programas en NUEZO.

```
type funID (type1 n1 , type2 & n2 , ...  , typeN nN){
    «function body»
}
```

2 Especificación del lenguaje

2.1 Comentarios

Para comentar algo en nuestro lenguaje, hará falta "abrir" el comentario con << y cerrarlo con >>. Esto será así, independientemente de la extensión del comentario o de si es multilínea.

```
«Esto es un comentario»
«Esto no es un comentario
«
Esto es un
comentario multilínea
»
```

2.2 Tipos básicos de variables

En nuestro lenguaje, los enteros se llamarán "`znum`" y los reales "`rnum`", como referencia a sus conjuntos $\mathbb{Z}$ y $\mathbb{R}$. Más adelante especificamos mejor la relación entre enteros y reales en `NUEZO`.

Los booleanos se llamarán "`state`" y podrán tener dos valores: `on` (equivalente a `true`) u `off` (`false`).

```
znum n; «esto es un entero»
rnum r; «esto es un real»
state b; «esto es un booleano»
```

2.3 Declaración de variables

Para declarar variables hay que poner el tipo y luego el nombre. Si se quiere asignar un valor, habrá que usar "`:=`".
Se podrán declarar variables globales constantes, no se permitirá cambiar su valor durante la ejecución de un programa ni ser pasadas por referencia a funciones. Para ello se debe introducir `define` justo delante del tipo de la variable.

```
znum var1;
znum var2 := 7;
define state var3 := off; «var3 es una constante»
```

2.4 Arrays

Para declarar arrays habrá que poner el tipo de las variables del array seguido de la palabra "`chain`", que será una palabra reservada. Después se indicará el tamaño entre paréntesis y el nombre del array. En el caso de declarar arrays de tipo compuesto (por ejemplo otro array), se debe introducir este entre paréntesis.

Si se desea un array de varias dimensiones, se indicarán los tamaños para cada una de ellas separados por comas.

Para inicializar un array, se hará lo mismo que con las variables simples, es decir, usar `":="` y poner entre llaves los valores (si se desea inicializar cada uno con un valor distinto) o entre corchetes (si se desea inicializar todos al mismo valor).

Para acceder a la posición i-ésima de un array se escribirá el nombre del array seguido de `[i]`. Si se trata de arrays multidimensionales, la instrucción mencionada devolverá otro array con una dimensión menor. Los accesos a este array deberán hacerse repitiendo el procedimiento sobre el resultado (el acceso a la posición `"i,j"` no se realiza con `"[i,j]"`, se deben realizar las operaciones por separado, es decir, `"[i][j]"`).

Al igual que en la mayoría de lenguajes de programación, el primer elemento de este se encuentra en la posición 0. Por lo que, un array de n elementos no tendrá ninguno en la posición n.

```
znum chain (3) arr1; «array simple de 3 elementos»
znum chain (4) arr2 := {1,5,6,7};
znum chain (8) arr3 := [5]; «inicializas todas las posiciones a 5»
znum chain (4,5) arr4; «array de dos dimensiones (4x5)»
(znum chain (4)) chain (5) arr4; «array de dos dimensiones (4x5)»
```

2.5 Structs

Para declarar structs habrá que poner la palabra reservada `"block"`. Después se especificará el nombre del struct y se pondrá entre llaves los campos del struct. Al igual que en `C++`, si queremos inicializar un campo se hará igual que cuando declaramos variables simples.

También es posible darle valor a todos los campos de un struct a la vez, en ese caso se hará con los valores de todos los campos separados por comas y entre llaves.

Para acceder a un campo de dicho struct, se hace con `"."`. Esto sirve tanto para leer el valor como para modificarlo.

```
block st1 {znum n, state b := on};
st1 var;
var := {3, off};
var.n := 3;
znum m = st1.n;
```

2.6 Alias

Para renombrar un tipo se utilizarán las palabras reservadas "`rename`" seguida del tipo a renombrar y de "`as`" y el nuevo nombre. No hace falta poner "`;`" al final de la instrucción.
Al igual que en el caso de los arrays, si queremos asignar un nombre a un tipo compuesto, este debe introducirse entre paréntesis.

```
rename znum as edad
rename (znum chain (3)) as list3
```

2.7 Punteros

Para declarar un puntero se usa la palabra "`dir`". Primero, se pondrá el tipo de variable a apuntar. Si se desea que este sea un tipo compuesto, se debe introducir entre paréntesis.

```
znum dir pointer;
```

Para inicializar dicho puntero, se hará igual que en `C++`, es decir, con el operador de asignación "`:=`", la palabra reservada "`new`" y el tipo a apuntar. En caso de ser un array, hará falta también especificar el tamaño. Realmente, un array es, por naturaleza, un puntero a su primer elemento.

```
pointer := new znum;
```

Para inicializar el valor al que apunta, se usará "`()`". Para acceder a su valor, se utilizará el operador prefijo "`_`" precediendo al nombre de la variable.

```
pointer := new znum (4);
znum x := _pointer;
«En pointer se guarda la dirección de memoria y en x se guarda su valor»
```

2.8 Operadores

Los operadores infijos utilizados en las operaciones aritméticas y booleanas serán los usuales `(+,-,*,==,!=,...)`, ya que son bastante intuitivos, a excepción de los mencionados a continuación.

En NUEZO, la '/' es la representante de la división real. En el caso de la división entera utilizaremos el operador '`//`' para distinguirlo de la anterior.

Para los operadores booleanos and y or utilizaremos una notación incluso más intuitiva, con los operadores '`and`' y '`or`' respectivamente.

Para representar números negativos se hará uso del operador unario prefijo '`-`', como en la notación habitual.
Para acceder a la dirección en la que se almacena una variable se utilizará el operador unario prefijo '`&`', al igual que en `C++`.

```
znum x := 1;znum y := 2;
znum z1 := x-y; «En z se guardará el valor -1»
znum z2 := x//y; «En z se guardará el valor 0»
rnum z3 := x/y; «En z se guardará el valor 0.5»
```

Para acceder a los elementos de un array, indicaremos el nombre de este, seguido del operador '`[]`' con el índice del elemento al que queremos acceder.

En el caso de los arrays n-dimensionales, se hará de la misma forma pero indicando los índices de cada dimensión separados por ','. Si en un array n-dimensional accedemos a un elemento introduciendo k índices (obligatoriamente $k \leq n$), se devolverá el array (n-k)-dimensional que se encuentre en la posición correspondiente.

```
znum chain (4) arr := 1,2,3,4;
znum x := arr[0];«En x se guardará el valor 1»
znum chain (2,2,2) arr1 := [1];
znum y := arr1[1][0][1]; «En y se guardará el valor 1»
znum chain z (2,2) := arr1[1];
```

En NUEZO se incluyen unos operadores muy útiles de `C++`, con la notación de operadores infijos binarios seguidos de '`:=`'. Estos son: '`+:=`', '`-:=`', '`*:=`', '`/:=`', '`//:=`' y ' `%:=`'.

En la siguiente tabla se muestran las prioridades de cada operador.

2.9 Escritura y lectura por pantalla

Para leer y escribir por pantalla, usamos funciones intuitivas exclusivas de NUEZO. Si queremos leer enteros, reales o booleanos usaremos `zRead`, `rRead` y `sRead` respectivamente. De manera análoga se escribirán por pantalla (cambiando `Read` por `Write`), con funciones cuyo único argumento será la variable a escribir.

La escritura de booleanos (`state`) se hará de la siguiente manera: si la variable tiene el valor `on` se escribira un `1` y, en caso contrario, un `0`.

Operador	Prioridad	Tipo	Asociatividad
.	0	Binario	No asociativo
&	1	Unario (prefijo)	No asociativo
-	1	Unario (prefijo)	No asociativo
* / // %	2	Binario	Izquierda
+ -	3	Binario	Izquierda
<= >= < >	4	Binario	Izquierda
== !=	5	Binario	Izquierda
!	6	Unario (prefijo)	No asociativo
and	7	Binario	Izquierda
or	8	Binario	Izquierda
:= +:= -:= *:= /:= //:= %:=	9	Binario	No asociativo

```
znum x := zRead();
zWrite(x==0); «Escribirá 1 si x==0 y 0 e.o.c.»
```

2.10 Relación entero - real

Para facilitar la programación, en NUEZO permitimos asignar valores enteros (`znum`) a variables reales (`rnum`). Pero esto no se aplica en el caso simétrico: no es posible asignar un valor real a una variable entera.

Se permite también sumar una variable real con una entera, en cuyo caso el valor devuelto es real. Al igual que con la suma, esto se permite con el resto de operaciones aritméticas. Como es lógico, la operación de división entera siempre devuelve un resultado entero sean cuales sean las variables a las que se aplica. Veamos algunos ejemplos de operaciones permitidas:

```
znum x := 6.0; «Esto no está permitido»
rnum y := 6; «Guarda el valor 6.0 en y»
rum z := 1 + 1.5; «Guarda el valor 2.5 en z»
z *:= 3; «Ahora z vale 7.5»
```

2.11 Condicionales

En NUEZO, como en cualquier otro lenguaje, tenemos instrucciones condicionales. Estas evalúan una expresión de tipo state y, si su valor es `on`, ejecuta una serie de instrucciones; si su valor es `off`, entonces salta al final de estas, sin ejecutarlas.

Para estas instrucciones utilizaremos una notación habitual en los lenguajes de programación (`if-elif-else`). Las palabras reservadas '`elif`' y '`else`' se utilizarán de forma opcional si queremos añadir series de instrucciones que se ejecuten en casos cuando no se cumpla la primera condición.

```
if(cond1){
    «Se ejecuta si se cumple cond1»
} elif(cond2){
    «Se ejecuta si no se cumple cond1 pero sí cond2»
} elif(cond3){
    «Se ejecuta si no se cumplen cond1 ni cond2 pero sí cond3»
} else
    «Se ejecuta si no se cumplen cond1 ni cond2 ni cond3»
}
```

2.12 Bucles

En NUEZO existen los siguientes tres tipos de bucles:

- **Bucle WHILE:** para utilizar este tipo de bucles se debe escribir la palabra reservada `while`, seguida de una expresión de tipo `state` entre paréntesis y el cuerpo del bucle, una serie de instrucciones, entre llaves. El funcionamiento del bucle `while` en NUEZO es similar al que tiene en `C++`. Primero se evalúa la expresión de tipo `state`, si su valor es `on` se ejecuta el cuerpo del bucle y, posteriormente, se vuelve a evaluar la expresión `state`. Este proceso se repite hasta que el valor de la expresión `state` es `off`.

  ```
  while (cond){
      «Cuerpo del bucle while»
  }
  ```

- **Bucle FOR:** los bucles `for` en NUEZO son como los de `C++`. Entre paréntesis se indicarán los tres campos usuales separados de ';'.

  En el primero, se inicializa la variable cuyo valor cambiará cada iteración (o se pone una expresión cualquiera, por ejemplo, `"i:=0"`, en cuyo caso se entiende que la variable `i` ya estaba creada de antes y quieres empezar el bucle asignándole el valor 0). En el segundo,

se indica la condición de parada. El último de ellos será la instrucción que queramos que se ejecute al final de cada vuelta, relacionada con la variable del primer campo.

```
for (znum i:=0 ; i<10 ; i+:=1){
    «Cuerpo del for que da 10 vueltas»
}
```

- **Bucle FOR EACH:** inspirándonos en `Java`, en NUEZO tenemos el bucle `for each`, es decir, un bucle que facilita la iteración de una lista de elementos. Su sintaxis es muy intuitiva y sólo funciona si la variable a iterar es de tipo `chain`. Se guardará cada uno de sus elementos en una variable local auxiliar, que cambiará cada iteración.

  Para separar el nombre de la variable auxiliar del de la lista, se usa la palabra reservada `in`.

```
for each (znum elem in list){
    «En elem se guarda cada elemento de la lista chain.
    El bucle acaba cuando ya hemos iterado la lista entera»
}
```

Dentro de cualquier tipo de bucles pueden aparecer condicionales o incluso otros bucles. Permitiendo por ejemplo dos bucles for each anidados para recorrer todos los elementos de un array de dos dimensiones.

2.13 Expresiones del lenguaje

Las expresiones que aparecen en el lenguaje NUEZO son las siguientes:

- **Constantes:** constan de los números, ya sean `znum` o `rnum` y los valores del tipo `state (on, off)`.
- **Variables:** identificadores de las variables creadas por el usuario, tengan valor asignado o no. Estos pueden corresponder a un tipo simple o a un `block`.
- **Operadores infijos:** los ya mencionados que pueden utilizarse en expresiones aritméticas o `state`.
- **Llamadas a función:** consistirá del identificador de la función a la que se pretende llamar, seguida de los argumentos que esta necesita entre paréntesis y separados por ’`,`’.

```
funID(arg1,arg2,...,argN);
```

3 Ejemplos de códigos

Veamos una serie de ejemplos de programas y funciones escritas en NUEZO, con la intención de familiarizarnos más con el lenguaje y ver sus analogías con el resto.

3.1 Algoritmo de Euclides para la división

En este ejemplo, se quiere calcular la división 21/4 con el algoritmo de Euclides. Utilizamos una función auxilar *division* que recibe cuatro argumentos y la calcula.

Hacemos uso de dos punteros (*quotient* y *reminder*), así como un bucle `while`.

```
silent division (znum dividend, znum divisor, znum & quotient, znum &
reminder){
    quotient := 0;
    remainder := dividend;

    while (remainder >= divisor){
         remainder -:= divisor;
         quotient +:= 1;
    }
}

znum engine(){
    znum dividend := 21;
    znum divisor := 4;
    znum quotient;
    znum reminder;

    division (dividend, divisor, quotient, reminder);
    «Se guardan los resultados en quotient y reminder»
    zWrite(quotient);
    return 0;
}
```

3.2 Lectura/escritura

En este ejemplo se leerá un número entero por consola. Si este es par, se escribirá el mismo número y, en caso contrario, su opuesto.

```
silent readWrite() {
    znum x := zRead();
    if(x%2 == 0){
        zWrite(x);
    } else{
        zWrite(-x);
    }
}

znum engine(){
    readWrite();
    return 0;
}
```

3.3 Máximo de una lista

En este último ejemplo se programa una función que calcula el máximo de una lista, usando el bucle `for each`. Para simplificar, suponemos que todos los números de la lista son mayores o iguales que 0.

```
znum maxi(znum chain list) {
    znum x := -1;
    for each (znum elem in list){
        if (elem > x){
            x := elem;
        }
    }
    return x;
}
```