Especificación y ejemplos de la sintaxis del lenguaje Napol

David Mallasén e Iván Prada

Resumen

En este documento vamos a recoger la especificación de la sintaxis de nuestro lenguaje Napol, así como algunos ejemplos de código.

Índice

1. Identificadores y ámbitos de definición

Los comentarios serán de bloque y sin anidamiento. Empezarán y terminarán por #. Todos los finales de línea deberán terminar con punto y coma "**;**".

Para la declaración de variables seguiremos la sintaxis: **tipoVariable** nombre. Para los arrays utilizaremos la notación de [**tam**], donde *tam* será el tamaño de esa dimensión. Podremos declarar arrays de varias dimensiones separando los tamaños por dos puntos **":"**. Para indicar un array simple escribiremos **tipo[tam] nombre**, para arrays multidimensionales: **tipo[tam1:tam2:...:tamN] nombre**. Los índices de los arrays irán de 0 hasta $tam - 1$. No se podrán declarar varias variables en una misma instrucción.

```
int x;          # Declaramos una variable entera x #

bool[5] xs;     # Creamos un array de booleanos de tamaño 5 #
int[2:7:5] ys;  # Creamos un array de enteros de 3 dimensiones
                de tamaños 2, 7 y 5 #
```

Para la construcción de funciones se usarán las palabras reservadas **fun**, **returns** y **return** de la siguiente manera:

```
fun nombreFuncion(tipo1 argumento1; ... ; tipoN argumentoN;) returns tipoDevuelto {
    # instrucciones #
    return valor;   # Con valor de tipo tipoDevuelto #
}
```

Donde **fun** indica el comienzo de la declaración de una nueva función, los parámetros de la función vienen entre paréntesis y terminados cada uno de ellos con punto y coma (como si fueran declaraciones de variables) y el tipo del valor devuelto viene después de **returns**. Usaremos **return** en el cuerpo de la función para terminar y devolver el valor indicado (ya sea con un valor explícito, con una variable o una expresión en general). Los arrays se pasarán siempre por referencia. No se pueden devolver arrays. Las variables simples se pasarán siempre por valor, en caso de necesitarse por referencia el programador podrá crear un array de una dimensión formado por un elemento para hacer uso del paso por referencia.

Veamos un ejemplo en el que crearemos una función que reciba dos parámetros, sume y devuelva el resultado:

```
fun suma(int x1; int[3:4] x2;) returns int{
    int aux;
    aux = x1 + x2[0:0];
    return aux;
}

fun resta(int x1; int[3:4] x2;) returns int{
    return x1 - x2[0:0];
}
```

Dentro del bloque de código de la función podemos tanto crear variables (de ámbito local a la función) como asignarlas y usarlas. Los bloques anidados de código irán entre llaves **"{"** y **"}"**. El ámbito de los argumentos será el mismo que el del cuerpo de la función.

Para la construcción de procedimientos se usará la palabra reservada **proc** de la siguiente manera:

```
proc nombreFuncion(tipo1 argumento1; ... ; tipoN argumentoN;) {
    # instrucciones #
}
```

El tratamiento de los procedimientos será análogo al de las funciones pero sin devolver un valor.

2. Tipos

Tendremos tipos predefinidos de enteros y booleanos a los que llamamos **int** y **bool** respectivamente (y que serán los tipos de nuestras variables, además de los arrays formados por ellos). Como operadores infijos:

- **Enteros:** + (suma), - (resta), * (producto), / (división entera).
- **Booleanos:** == (igualdad), <= (menor o igual), >= (mayor o igual), < (menor), > (mayor), != (distinto), && (and), || (or), ! (not).

La prioridad de los operadores, de más a menos prioritarios, será la siguiente:

1. Los paréntesis de agrupación.
2. El operador unario ! asociativo.

3. Los operadores booleanos >,<,>=,<= sin asociatividad, los cuales reciben dos enteros y devuelven un booleano.

4. Los operadores == , ! =, / y ∗ con asociatividad a izquierdas.

5. + y − con asociatividad a izquierdas.

6. && y || con asociatividad a derechas, junto a − que es no asociativo.

Equivalencia estructural de tipos: dos variables tendrán el mismo tipo si fueron declaradas con el mismo tipo.

3. Instrucciones ejecutables

El operador de asignación será el = y para elementos de arrays multidimensionales indicaremos la posición a asignar separando las dimensiones por dos puntos **":"**: **nombre[pos 1:...:pos n]**. Solo se podrán asignar valores a elementos de un array. También sólo se podrán pasar a funciones/procedimientos, o el array completo o un elemento simple de él. No se permitirá el uso de trozos de un array debido al tratamiento de arrays multidimensionales que se les da, así como tampoco se podrá asignar un array a otro (ya que no se ha realizado la parte de opcional de punteros).

```
int x;
int y;
x = 5;
y = x;

bool ok;
bool[5:5] mat;

mat[0:2] = false;
ok = mat[0:2];
```

Tendremos condicionales de una y dos ramas que vendrán identificados con **if** y opcionalmente para la segunda rama con **else**. Indicaremos la condición entre paréntesis a continuación del **if** de la siguiente manera:

```
if(cond) {
    # rama del then #
}
# else opcional #
else {
    # rama del else #
}
```

Los bucles **while** serán de la siguiente forma:

```
while(cond){
    # Instrucciones a ejecutar mientras que cond sea cierta #
}
```

Veamos un ejemplo de uso de **while** que calcula el factorial de 10:

```
int n;
int fact;
fact = 1;
n = 10;
while(n > 1) {
    fact = fact * n;
    n = n - 1;
}
# fact = 10! #
```

Tendremos también un bucle **for** con la siguiente sintaxis:

```
# Directamente poniendo la variable iteradora y usando su valor #
for(variableIt; cond; incr;) {
    # Instrucciones a ejecutar mientras que cond sea cierta #
}

# Asignando un valor a la variable iteradora #
for(variableIt = valor; cond; incr;) {
    # Instrucciones a ejecutar mientras que cond sea cierta #
}
```

Veamos un ejemplo que inicializa todos los valores de una matriz 10×10 de enteros a 0:

```
int[10:10] x;
int i;
int j;

for(i = 0; i < 10; i = i + 1;) {
    j = 0;
    for(j; j < 10; j = j + 1;) {
        x[i:j] = 0;
    }
}
```

Tendremos la equivalencia siguiente entre un **for** y un **while**:

```
# Version con for #
int i;
for(i = 0; i < 10; i = i + 1;) {
    # cuerpo #
}

# Version con while #
int j;
j = 0;
while(j < 10) {
    # cuerpo #
    j = j + 1;
}
```

Las expresiones formadas por constantes e identificadores así como operadores infijos se realizarán siguiendo las prioridades de los operadores ya citados anteriormente.

Las llamadas a funciones se realizarán con la siguiente sintaxis:

```
nombreFuncion(argumentos);
```

Se podrán hacer llamadas a funciones en cualquier lugar donde pueda ir una expresión. El tipo esperado de la expresión debe ser el tipo indicado en el **returns** de la función.

Las llamadas a procedimientos se realizarán usando la palabra reservada **do** con la siguiente sintaxis:

```
do nombreProcedimiento(argumentos);
```

Se podrán hacer llamadas a procedimientos en cualquier lugar donde pueda ir una sentencia.

Un ejemplo ilustrativo:

```
int x;  # El returns de suma es de tipo int #
x = suma(2, 3);
```

```
if(esVerdad()) { # El returns de esVerdad es de tipo bool #
    do hacerAlgo(x); # hacerAlgo es un proc que admite un argumento de tipo int #
}
```

4. Errores

Como mensaje de error se dará una línea con la siguiente estructura:
"ERROR **ETAPA DE ERROR** en fila **Fila** y columna **Columna**. Breve mensaje con el motivo de error."

5. Estructura de un archivo Napol

Los programas de Napol tendrán la siguiente estructura: primero un conjunto de sentencias que podrán ser declaraciones de variables, asignaciones en estas, declaraciones de funciones así como declaraciones a procedimientos. Posteriormente, la última instrucción de nuestro código será una llamada a un procedimiento que hayamos declarado anteriormente (que por decirlo de alguna forma, será el que se comporte como un Main). Los ficheros de Napol han de tener la extensión ".napol".