2012 | 2013

Especificación del Lenguaje nADA

Procesadores de Lenguajes

Daniel Arbelo Cabrera
Alberto Manuel Mireles Suárez
David Guillermo Morales Sáez
Eduardo Quesada Díaz
María del Carmen Sánchez Medrano

Universidad de Las Palmas de Gran Canaria

Índice

Introducción **3**

Características del lenguaje **3**

Identificadores **3**
Literales **3**
Operadores **3**
Variables **5**
Tipos de datos construidos **6**
Ristras **6**
Comentarios **7**
Procedimientos y funciones **7**
Procedimientos 7
Funciones 8
Paso de parámetros 9
Preprocesador **10**
Ficheros **10**
Estructuras de control **10**

Anexos **11**

File_Type **11**

Introducción

A lo largo de este documento se definen las características del lenguaje de programación nADA, un subconjunto del lenguaje ADA.
Hemos decidido emplear un lenguaje basado en ADA puesto que ya hemos trabajado con él y estamos familiarizados con el mismo. A su vez tiene la parte de declaraciones bien definida y es un lenguaje fuertemente tipado, lo cual garantiza que se trabajará con tipos compatibles. Finalmente, podemos añadir que la sintaxis del lenguaje ADA es muy similar a Pascal, que es un lenguaje apropiado para escribir un compilador.

Características del lenguaje

Identificadores

Los identificadores estarán formados por los caracteres alfanuméricos del alfabeto inglés, empezando siempre por una letra y de tamaño máximo 20 caracteres.

Literales

Admitimos tres tipos de literales: números enteros, en punto flotante y caracteres. Los rangos de los valores son los siguientes:

Tipo de dato	Valor mínimo	Valor máximo
Entero	$-2^{15} + 1$	$2^{15} - 1$
Punto flotante		
Caracteres	0	126

Operadores

Los operadores matemáticos básicos permitidos son los siguientes:

- Suma

La suma está representada con el símbolo + y consiste en combinar dos o más números del mismo tipo.

```
(Literal) + (Literal)
```

- Resta

La resta es la operación inversa a la suma en la que dada cierta cantidad se elimina una parte de ella, siendo el resultado de esta operación la diferencia o resta. Se usará el símbolo – para esta operación.

```
(Literal) - (Literal)
```

- Multiplicación

La multiplicación es una operación matemética que consiste en sumar un número tantas veces como indicas otro número. Se emplea el símbolo * para realizar esta operación.

```
(Literal) * (Literal)
```

- División

La división es una operación aritmética de descomposición que consiste en averiguar cuántas veces un número (divisor) está contenido en otro número (dividendo). El símbolo / representa esta operación.

```
(Literal) / (Literal)
```

- Igualdad

Se basa en la comprobación de que el término a la derecha del símbolo "=" tiene el mismo valor numérico del que está a la izquierda, siendo ambos términos del mismo tipo (entero o punto flotante). Así mismo el valor devuelto en dicha comprobación es de tipo booleano, siendo de valor "`true`" en caso de que la comparación sea cierta y "`false`" en caso de que sea negativa.

- Mayor que / Menor que

Con estos símbolos se comprueba que el valor a la derecha del comprobante condicional tiene un mayor o menor valor numérico (según el símbolo utilizado) que el elemento que esté a la izquierda del condicional, que como en el caso de la igualdad solo acepta elementos de tipo entero o punto flotante. Como ocurre en la igualdad, el valor que indica dicha comprobación es un booleano, en caso de que sea "`true`" significa que el valor a la derecha cumple la condición comparado con el valor a la izquierda, y en caso de que se devuelva "`false`" indicará que no cumple dicha condición. La operación mayor que está representada por el símbolo “>”, mientras que menor que utiliza el símbolo “<”.

- Mayor igual que / Menor igual que

En este caso nos encontramos ante una amalgama entre los operadores anteriores, donde comprobamos que el elemento a la izquierda del condicional cumpla dos condiciones: que sea igual o menor o mayor (según el caso que corresponda). El tipo de datos que acepta son enteros o punto flotante y el valor que devuelve esta comparación es como en los casos anteriores, de tipo booleano: "`true`" si se cumple y "`false`" si no se cumple. Para mayor o igual que se emplea ">=", mientras que para menor o igual que se utiliza "<=".

- Operadores binarios

Los operadores binarios que contemplamos son `and`, `or` y `not`. El operador `and` realiza la función booleana de producto lógico, la cual da verdadero a la salida si las dos entradas valen verdadero; el operador `or` realiza la suma lógica, ofreciendo un valor verdadero a la salida siempre que alguno de sus argumentos sea verdadero; y el operador `not` realiza la negación lógica, convirtiendo un valor de entrada en su opuesto (verdadero en falso y falso en verdadero).

Variables

Las variables se declararán después de la línea de `procedure` o `function` y antes del `begin` del programa. Una declaración de variables consta de: una lista de nombres de las variables que se declaran separadas por comas (",") y terminada con dos puntos (":"),el tipo de las variables y, opcionalmente, la asignación de un valor inicial.
La asignación de valores varía según donde se haga, si es en la misma declaración de la variable una vez elegido el tipo de la misma irá precedido de dos puntos y un igual (":=") y a continuación el valor que se le desea asignar. En caso de que se le vaya a dar un valor en medio de la ejecución del código será simplemente el nombre de la variable seguida de los dos puntos y el igual (":=") y posteriormente el valor a asignar.
Un procedimiento solamente será capaz de acceder a variables declaradas en su mismo nivel, por tanto no podrá acceder a variables que se hayan declarado en niveles superiores o inferiores.

```
procedure un_procedimiento (x : out integer) is
--declaración de variables
     x : integer := 1;
begin
     -- código
     x := 3;
     -- código
end un_procedimiento;
```

Tipos de datos construidos

Los tipos de datos construidos que utilizaremos en nADA son dos:

- Vectores
- Enumerados

Los vectores son un conjunto de elementos ordenados. Pueden ser de cualquier tipo, tanto simple como compuesto (podemos tener un vector de vectores) pero dentro de un vector, todos los elementos han de ser del mismo tipo. Se pude acceder a cada elemento en particular a través de un índice, el cual se indica poniendo entre paréntesis el mismo después del nombre de la variable, siempre que se encuentre dentro del rango de valores especificado en la declaración del vector. Su declaración se lleva a cabo con la palabra reservada "`array`", seguido del rango de valores encapsulado por paréntesis y el tipo de dato que será cada elemento del vector. El rango se define con el valor inicial seguido de dos puntos y el valor final del rango. Para poder usar los vectores se deberá crear por tanto el tipo del vector, y luego se podrá declarar una variable de dicho tipo. A continuación declaramos un tipo de vector de 10 elementos de tipo entero:

```
type Array_Entero is array (1..10) of Integer;
vector : Array_Entero;
```

Los tipos enumerados son un tipo de variable que solo puede tomar uno de los valores definidos en el tipo enumerado. Es decir, al crear el tipo se deben indicar los valores que podrá tomar una variable de ese tipo. Por ejemplo, si queremos un tipo de dato enumerado con el fin de consultar el color de un semáforo, sabemos que esa variable tomará tan solo 3 valores, por lo que podemos crear el siguiente tipo enumerado:

```
type enumerado is (ROJO, AMBAR, VERDE);
semaforo : enumerado;
```

De esta manera, la variable "`semaforo`" solo podrá tomar los tres valores establecidos, y facilita el manejo de estructuras de selección, ya que se pueden emplear valores cuyo nombre sea significativo para el programador.

Ristras

Las ristras son un tipo de dato que conforman un conjunto de caracteres encapsulados por el símbolo ' " '. Tendrán como máximo un tamaño de 255 y un último carácter que represente el fin de ristra ('`\0`'). Para poder hacer uso de este tipo de variable se ha de incluir la librería `Strings`. Las ristras pueden declararse de la siguiente manera:

```
ristra : String;
```

Comentarios

Se permitirán los comentarios de línea que vendrán precedidos por "--". Esto indica que los caracteres que se encuentren a partir de este símbolo y hasta el final de línea son comentarios. A su vez no permitiremos los comentarios de bloque.

Procedimientos y funciones

Dentro de nuestro lenguaje nADA tenemos definidas dos tipos de estructuras, los procedimientos y las funciones. Dentro de esta primera, los procedimientos, se encuentra la estructura principal de un programa, lo que en lenguaje C conoceríamos como la función `main`.

Procedimientos

Poseen una estructura que comienza con la palabra reservada 'procedure', seguido del `nombre del procedimiento` y entre paréntesis las variables que posea el procedimiento, separas por ",":

```
Nombre de variable : tipo de variable
```

Seguidamente, encontraremos la palabra reservada `'is'`, que marca el inicio de la declaración de las variables internas del proceso, así como las declaraciones de los subprogramas, procedimientos o funciones, que vayan a ser llamados.

Tras la declaración, se debe encontrar la palabra reservada `'begin'`, con la cual comienza el código de ejecución del procedimiento. Este mismo terminará cuando se encuentre la palabra reservada `'end'`, seguido del `nombre del procedimiento` y finalizando con un `';'`.
Hay que matizar, que si estamos en el procedimiento del programa principal este no posee variables de entrada, salida o entrada-salida, simplemente después del nombre del procedimiento le sigue la palabra reservada `is`. También hay que decir que si un programa quiere utilizar una variable o llamar a un subprograma, es estrictamente necesario que se encuentre declarado en entre el `is-begin`. Un ejemplo del uso de procedimientos sería:

```
procedure main is
-- Declaración de variables
    A: integer;
    B: integer;
    C: integer;
-- Procedimiento suma
   procedure suma (A1: in integer, B1: in integer, C1: out
integer) is
   begin
     C1 := A1+B1;
   end suma;
```

```
begin
      A := 1;
      B := 3;
      suma(A,B,C);
end
```

Donde tenemos el programa principal `main`, en el que después del `is` se realiza una declaración de variables, `A,B,C` de tipo entero, y luego se crea un subprocedimiento `suma`, el cual inserta en la variable de salida `C1`, el resultado de la suma de `A1+B1`. Este subprocedimiento es llamado por el programa principal, `suma (A,B,C)`, donde `C` obtendrá el resultado de sumar, `A+B` es decir `1+3`.
Hay que tener en cuenta, que cualquier procedimiento, sea el principal o no, puede tener definido en el bloque `is-begin` tantas funciones y procedimiento como sean necesarios, para la ejecución de la tarea a realizar.

Funciones

Una función consiste en un conjunto de instrucciones con un objetivo concreto que puede ser ejecutada desde otra función o procedimiento. Se diferencian de los procedimientos porque, a diferencia de éstos, las funciones han de devolver un único resultado, y sus parámetros (en caso de tenerlos) serán solo de entrada. Además, una función puede llamarse a sí misma dentro de su código, generando recursividad.

En nuestro lenguaje, la declaración de una función comenzará por la palabra reservada `function`, tras lo cual se indicará el nombre de la función y, entre paréntesis, los parámetros con su tipo de datos asociado y separados por comas (como las funciones sólo admiten parámetros de entrada, no se permite añadir la palabra reservada in junto a éstos). Luego se especifica el tipo de datos que devolverá la función, escribiendo `return [tipo de dato]`, y la palabra reservada `is`. Dentro de un bloque `begin … end [nombre de la función]` se incluirá el código de la misma. Para devolver el valor desde la misma función, se utilizará la sentencia `return`. Como una función prototipo puede considerarse la siguiente:

```
function [nombre de la función] (parámetro : tipo,
parámetro : tipo , …) return [tipo de dato] is
begin
      instrucción;
      instrucción;
      …
      instrucción;
      return [dato];
end [nombre de la función];
```

Un ejemplo para una función en nADA puede ser el siguiente:

```
function Minimo (A : Integer, B : Integer) return
Integer is
begin
     if A <= B then
          return A;
     else
          return B;
     end if;
end Minimo;
```

Paso de parámetros

Dependiendo del modo en el que se le sea pasada una expresión o variable a una función ésta será tratada de una manera distinta.

- `in`: Si la variable o expresión en cuestión es solo de entrada se hará una copia de dicha variable que existirá mientras dicha función todavía exista, y una vez acabe, la copia será destruida.

- `in out`, `out`: Si la variable es de entrada y salida, o salida, será pasada por referencia para poder modificar su valor y al final de la función, devolverlo y cambiar el valor original por el actualizado en el programa que realizó la llamada original a la función. No se admiten expresiones en este tipo de parámetros.

A la hora de declarar una variable dentro de un procedimiento ésta puede ser de 3 tipos:

- `in`: la variable o expresión es de entrada y su valor es usado en el procedimiento, puede ser actualizado dentro del mismo pero no es devuelto, por tanto en el procedimiento que lo llamó mantendrá el mismo valor con el cual fue pasado.

- `out`: la variable se declara de salida y al terminar la ejecución del procedimiento su valor se retorna hacia el procedimiento que llamó a dicho procedimiento.

- `in out`: la variable en cuestión es de entrada y salida a la vez, por tanto su valor se le pasa al procedimiento, dentro del mismo es modificado y cuando termina, se devuelve el valor actualizado al procedimiento que llamó al procedimiento. Este tipo de variables se devuelve como valor resultado

Preprocesador

Para el uso de funciones del sistema, se utiliza la directiva '`with`'. Esta directiva indica qué ficheros externos al programa se han de cargar para utilizar las funciones y clases definidas en el fichero. Para declarar qué librerías se desean utilizar, se ha de seguir el siguiente patrón:

```
with fichero, fichero, ... , fichero;
```

donde los ficheros están separados por comas ('`,`') y el último fichero va seguido de un punto y coma ('`;`').

Ficheros

Para la utilización de ficheros, se ha definido en el fichero `Text_Io` la clase `File_Type`. Este tipo representa la referencia del fichero una vez cargado en memoria, ya que consideramos que existen una serie de ficheros ya abiertos, como la pantalla o el teclado, a los cuales accedemos a partir de un vector. El tipo `File_Type` nos indica el índice en el vector de ficheros.

Existen tres tipos de modo de apertura y uso de ficheros:

- Modo Lectura (`Out_File`)
- Modo Escritura (`In_File`)
- Modo Escritura al final del fichero (`Append_File`)

Finalmente, hay que indicar que para poder utilizar un fichero, primero hay que abrirlo, utilizando la función `Open` y hay que cerrarlo una vez usado con la función `Close`. Para la lectura de datos de un fichero, se utilizará la familia de funciones `get()` y, para la escritura, se utilizará la familia de funciones `put()`, además de permitir la comprobación de fin de línea y fin de fichero.

Estructuras de control

Este lenguaje dispondrá de las siguientes estructuras de control:

- Estructura condicional
- Estructura de selección
- Estructura de repetición

Como estructura condicional utilizaremos el bloque `if – then – else – end if`, estableciendo una condición y ejecutándose las instrucciones pertinentes en función de la misma. Por ejemplo:

```
if(condición) then
     instrucciones
else
     instrucciones
end if;
```

Para la estructura de selección, utilizaremos la instrucción `case – is – when – end case`, estableciendo la variable a evaluar y escogiendo la entrada correspondiente. Por ejemplo:

```
case variable is
     when alternativa => instrucciones;
     when others => instrucciones;
end case;
```

Para la estructura de repetición, se empleará la instrucción `while – loop – end loop`, definiendo la condición de entrada del bucle justo después de la palabra reservada "`while`". Por ejemplo:

```
while (condición) loop
     instrucciones
end loop;
```

Anexos

File_Type

Se trata de una librería que nos permite realizar operaciones sobre ficheros. A su vez es un tipo de variable que representa el fichero (descriptor del fichero) el cual se representará por un número entero. Las operaciones que se permiten con los ficheros son:

- `Open(Descriptor,File_Type,"ruta")`: Se emplea para volcar en memoria el fichero y hace que el descriptor haga referencia al fichero.
- `Close(Descriptor)`: Libera el descriptor y cierra el fichero.
- `Get(Descriptor, carácter)`: Lee un carácter del fichero y lo vuelca en el segundo parámetro que es de salida.
- `Put(Descriptor, carácter)`: Escribe un carácter en el fichero, en la posición en la que se encuentra el índice del fichero.
- `End_Of_File(Descriptor)`: Devuelve un booleano que indica si se ha alcanzado el final del fichero.