El Lenguaje Nogo

Manuel Vilares Ferro

Departamento de Computación. Facultad de Informática
Universidad de La Coruña
Campus de Elviña, s/n. 15071 A Coruña
E-mail: vilares@udc.es
Url: http://www.dc.fi.udc.es/~vilares

El Lenguaje Nogo

Este apéndice describe el lenguaje de programación NOGO diseñado por el autor de esta memoria para servir de base práctica a la asignatura de Compiladores. El lenguaje NOGO es parte integrante de la familia de lenguajes imperativos. Como descripción primaria, podemos decir que la sintaxis y la semántica de NOGO constituyen un subconjunto de las presentadas por ADA.

0.1 Introducción

En esta sección pasamos a describir someramente la estructura general de NOGO como la de un lenguaje algorítmico organizado en bloques, así como la notación empleada para su posterior descripción formal, y un primer acercamiento a las posibilidades de cálculo de sus compiladores a partir de una sencilla clasificación de los errores del lenguaje.

Presentación del lenguaje. Un programa NOGO está constituido por dos partes esenciales, una descripción de las acciones a realizar, y una descripción de los datos manipulados por estas acciones. Las acciones se describen en NOGO mediante instrucciones y los datos mediante declaraciones.

Los datos se representan por los valores de sus variables. Cada variable que aparece en una instrucción debe haber sido previamente introducida mediante la correspondiente declaración, la cual le asocia un identificador y un tipo. El tipo define esencialmente el conjunto de valores que pueden ser asignados a la variable, siendo los tipos escalares predefinidos los siguientes: `integer`, `real`, `boolean` y `character`. Es posible, además, imponer restricciones de tipo intervalo a la clase `integer`.

Los tipos compuestos se definen dando la descripción de cada uno de sus componentes, diferenciándose por la estructuración de sus componentes. En el caso de los tableros, todos los componentes son del mismo tipo, pudiendo ser seleccionados según el valor de un índice. Los registros se estucturan en campos, no necesariamente del mismo tipo, que se seleccionan a partir de un identificador.

La instrucción de asignación permite asignar un valor calculado por una expresión, a una variable. Las instrucciones condicionales, `if` y `case`, y los diferentes tipos de bucles permiten el control de la ejecución del programa.

La estructuración de un programa en varias unidades se obtiene por aplicación del concepto de bloque, lo que permite la declaración local de variables, de procedures y de funciones. Ambos, procedures y funciones, pueden tener parámetros pasados por valor o por referencia. Las unidades de programación introducidas pueden imbricarse, dando a NOGO una estructura de bloques donde aplicar las reglas clásicas de alcance y visibilidad. Un programa NOGO es una procedure.

Notación. La gramática del lenguaje se describe mediante la utilización de una variante de la notación BNF. En particular:

- Las palabras con caracteres en minúscula denotan las categorías sintácticas, por ejemplo, `operador_aditivo`.
- Las palabras con caracteres en negrita denotan las palabras reservadas, por ejemplo, **begin**.
- Los corchetes delimitan los elementos opcionales, por ejemplo, **end** `[ identificador ]`.
- Las llaves delimitan un elemento repetido. Dicho elemento puede aparecer un número finito de veces, incluido cero. De este modo, una lista de identificadores se define por:

$$\text{identificadores} \rightarrow \text{identificador } \{ \text{ identificador } \}$$

- Una barra vertical separa dos alternativas, por ejemplo,

$$\begin{array}{lcl} \text{letra\_o\_cifra} & \rightarrow & \text{letra} \\ & & \mid \text{cifra} \end{array}$$

Además, las reglas de la gramática que describen las construcciones estructuradas se presentan de una forma que corresponde al sangrado recomendado del texto. Podemos considerar, por ejemplo, el caso de una instrucción `if`:

instrucción_if → **if** expresión **then**
instrucciones
else
instrucciones
end if;

Clasificación de los errores. El lenguaje reconoce tres categorías de errores:

- Los errores que deben ser detectados en el momento de la compilación por cualquier compilador NOGO. Estos se corresponden a cualquier violación de reglas diferente a las señaladas en los dos puntos que siguen.
- Los errores que deben ser detectados en el momento de la ejecución. Es lo que denominamos *excepciones* y provocan la terminación del programa NOGO.
- Finalmente, el lenguaje especifica ciertas reglas que deben ser respetadas por los programas, aunque los compiladores NOGO no estén obligados a verificar su cumplimiento. Para los errores de esta categoría, nuestro manual de referencia utiliza la palabra *erróneo*. Si un programa erróneo se ejecuta, su efecto es imprevisible.

0.2 Elementos lexicales

Describimos el conjunto de categorías lexicales que conforman NOGO.

Conjunto de caracteres. Todas las construcciones léxicas del lenguaje pueden representarse mediante un conjunto de caracteres que podemos dividir en la forma siguiente:

- Las letras mayúsculas.
- Las letras minúsculas.
- Las cifras.
- Los caracteres especiales: " ' ( ) * + , - . : ; < > !
- El carácter espacio en blanco.

Una letra minúscula es equivalente a la letra mayúscula correspondiente, salvo en una cadena de caracteres, o cuando esta letra representa a un carácter literal.

Unidades lexicales y convenciones de espaciamiento. Un programa es una secuencia de unidades lexicales. La división de la secuencia en líneas, y el espaciamiento entre las unidades lexicales no afectan para nada a la semántica del programa NOGO. Las unidades lexicales son los identificadores, los literales numéricos, los literales caracteres, las cadenas de caracteres, los delimitadores y los comentarios. Un delimitador es un carácter especial de entre los siguientes:

' ( ) * + , - . ; < = > |

o uno de los símbolos compuestos de la lista

=> := ** .. <> >= <=

Las unidades lexicales adyacentes pueden estar separadas por espacios, por un carácter especial o por un cambio de línea. Un identificador o un literal numérico deben estar separados de este modo de un identificador adyacente. No puede haber espacios en el interior de una unidad lexical, salvo en el caso de las cadenas de caracteres, los comentarios y del carácter literal espacio en blanco.

Identificadores. Los identificadores se utilizan como nombres, pero también como palabras reservadas. Un carácter subrayado aislado puede incluirse en un identificador. Todos los caracteres, incluyendo el subrayado, son significativos:

identificador → letra { [_] letra_o_cifra }
letra_o_cifra → letra
| cifra
letra → letra_mayúscula
| letra_minúscula

Los identificadores que no difieren salvo por la utilización de minúsculas y de mayúsculas son considerados idénticos.

Ejemplo:

CUENTA	X	leer_símbolo	María
PAGINA_3	Y1	SinIva	Leer_Página_Siguiente

□

Literales numéricos. Hay dos clases de literales numéricos: Los literales enteros y los reales.

literal_numérico → entero [. entero] [exponente]
entero → cifra { [_] cifra }
exponente → E [+] entero | E - entero

Un carácter subrayado aislado puede ser insertado entra las cifras adyacentes de un número entero, pero este tipo de carácter no es significativo.

La notación decimal convencional también se contempla en NOGO. Los reales se distinguen por la presencia de un punto. Un exponente indica la potencia de diez por la que el número precedente debe ser multiplicado para obtener el valor representado. Un literal entero puede tener exponente, que debe ser positivo o nulo. El exponente puede indicarse mediante `E` o bien `e`.

Ejemplo:

12	0	123_456	1E6	←	literales enteros
12.0	0.0	0.456	3.14159_26	←	literales reales
1.34E-12	1.0E+06			←	literales reales con exponente

□

Literales caracteres. Un literal carácter está formado por la inserción de un carácter entre dos apóstrofes.

Ejemplo:

'A' '*' ''' ' '

□

Cadenas de caracteres. Un literal cadena de caracteres es una secuencia de cero o varios caracteres insertados entre comillas

cadena_de_caracteres → " { carácter } "

de forma tal que una cadena de caracteres arbitraria puede ser representada. En particular, todo carácter " incluido debe ser doblado.

Ejemplo:

""		←	una cadena vacía
"A"	" "	""""	← tres cadenas de longitud uno

□

Comentarios. Un comentario comienza por dos literales -, y se termina con un fin de línea. No puede aparecer mas que después de una unidad lexical, o al principio o fin de un programa. Los comentarios no tienen ningún efecto sobre la semántica del programa.

Ejemplo:

```
-       Esto es un comentario
end;    - El fin de un bucle

-       Un comentario sobre
-       dos líneas
```

□

Palabras reservadas. Los identificadores cuya lista reflejamos en este apartado, son palabras reservadas sin una significación especial para NOGO. Los identificadores declarados no pueden coincidir con las palabras reservadas.

and	**array**	**begin**	**case**	**constant**	**declare**
do	**else**	**end**	**exit**	**for**	**function**
if	**in**	**is**	**loop**	**mod**	**not**
null	**of**	**or**	**others**	**out**	**procedure**
range	**record**	**return**	**reverse**	**subtype**	**then**
type	**when**	**while**			

0.3 Declaraciones y tipos

El lenguaje define varias formas de entidades nombradas. Una entidad nombrada puede ser una constante, una variable, un campo de un registro, un índice de bucle, un subprograma, un bucle nombrado, un parámetro de subprograma, o un tipo.

Una declaración asocia un identificador con una entidad declarada. Cada identificador debe ser explícitamente declarado antes de ser utilizado, con la excepción de los índices e identificadores de bucles, que lo son implícitamente. Consideramos la siguiente sintaxis:

```
declaraciones  →  { declaración }
declaración    →  declaración_de_objeto
                | declaración_de_tipo
                | declaración_de_subtipo
                | declaración_de_subprograma
```

Una declaración permite declarar una o varias entidades.

El proceso por el cual una declaración surge efecto a nivel de la semántica del programa se denomina *elaboración de la declaración*. Este proceso desencadena la siguiente sucesión de acciones:

- El identificador es introducido en el punto de su primera aparición. A partir de ese momento impone sus reglas de visibilidad y ámbito sobre otros identificadores previamente declarados.
- Podemos comenzar entonces la elaboración de la entidad declarada. En todos los casos, salvo para los subprogramas, un identificador no puede ser utilizado como nombre para otra entidad hasta el momento en que la fase de elaboración haya finalizado.

  Un identificador de subprograma puede utilizarse como nombre de la entidad correspondiente

La región de texto en la cual un identificador tiene efecto se denomina *ámbito del identificador*. Esta región comienza siempre en el lugar donde el identificador declarado es introducido. Las declaraciones de objetos, variables y constantes, así como de tipos pasamos a describirlas inmediatamente. El resto de declaraciones aparece más adelante.

Declaraciones de objetos. Un objeto es una entidad que contiene un valor de un tipo dado. Los objetos se introducen mediante declaraciones del tipo:

declaración_de_objeto	→	identificadores : [**constant**] indicación_de_subtipo [:= expresión] ;
		| identificadores : tipo_tablero ;
identificadores	→	identificador {, identificador }

Una declaración de objeto introduce uno o varios objetos de un tipo que es definido por su nombre, con una eventual limitación por intervalo, esto es, de tipo tablero.

Una declaración de objeto puede incluir una expresión especificando el valor inicial de los objetos declarados, con la condición de que la inicialización sea compatible. En este caso, ésta es evaluada en primer lugar y su valor afectado a todos los objetos declarados. Este valor debe satisfacer las limitaciones de intervalo que pudieran estar presentes en la declaración.

Un objeto es constante si la palabra reservada **constant** aparece en la declaración del objeto. El valor inicial de una constante no puede ser modificado con posterioridad.

Los objetos que no son constantes reciben el nombre de variables. El valor de una variable se define después de su elaboración, a menos que no haya una expresión de inicialización explícita. Un programa que depende de uno de tales valores indefinidos es erróneo.

Ejemplo:

```
-- ejemplos de declaración de variables

A, B, C : INTEGER;
ENGAÑO: BOOLEAN := FALSE;

-- ejemplos de declaración de constantes

X_23 : constant REAL := 25.4234;
N : constant INTEGER range 0 .. 100 := M / 2;
```

□

Declaraciones de tipos. Un tipo caracteriza un conjunto de valores y un conjunto de operaciones aplicables a estos valores. Existen dos clases de tipos:

- Los tipos escalares, en los que los valores no poseen más que un componente. Es el caso de los enteros, reales o booleanos.
- Los tipos compuestos, cuyo valor se vertebra en varios componentes. Es el caso de los tableros y de los tipos registros.

El conjunto de los valores de un tipo entero predefinido INTEGER puede ser restringido sin cambiar el conjunto de operaciones aplicables. Tales restricciones definen un subtipo del tipo entero. Es posible definir un subtipo de un subtipo.

declaración_de_tipo	→	**type** nombre_de_tipo **is** tipo_compuesto ;
declaración_de_subtipo	→	**subtype** nombre_de_tipo **is** indicación_de_subtipo ;
indicación_de_subtipo	→	nombre_de_tipo [restricción]
nombre_de_tipo	→	identificador
restricción	→	**range** expresión .. expresión

La elaboración de una definición de tipo produce un tipo diferente, mientras que la elaboración de un subtipo añade simplemente las restricciones suplementarias que deseamos considerar.

Cuando una restricción es considerada en una declaración de subtipo, ésta debe ser compatible con cualquier otra restricción previmante considerada. El programa NOGO se para ante la excepción CONSTRAINT_ERROR, si esta condición no se verifica.

Ejemplo:

```
type   TABLERO   is array(1 .. 10) of INTEGER;
type   PERSONA   is record
                   NOMBRE: CADENA;
                   EDAD: INTEGER range 0 .. 100;
                 end record ;
```

□

Ejemplo:

```
subtype   A   is INTEGER range 0 .. 100;
subtype   B   is A range 2 .. 10;
```

□

Tipos escalares. Los tipos escalares INTEGER, REAL, BOOLEAN y CHARACTER están predefinidos en NOGO. Pasamos a describirlos brevemente:

- El tipo INTEGER corresponde al conjunto de los enteros. Cada implementación del lenguaje NOGO deberá especificar este dominio, que será simétrico en relación al cero, con la posible excepción de un valor negativo suplementario.
- El tipo REAL es un tipo de coma flotante donde el margen de error se especifica de forma relativa fijando el número mínimo de cifras decimales a ocho.
- El tipo BOOLEAN contiene los valores FALSE y TRUE. La evaluación de una condición da un resultado de este tipo.
- El tipo CHARACTER describe los caracteres.

Ejemplo:

```
MÁS : constant CHARACTER := '+' ;
```

□

Tipos compuestos. La sintaxis para la declaración de tipos compuestos es la dada por las reglas siguientes:

```
tipo_compuesto  →  tipo_tablero
                |  tipo_registro
```

donde un tablero es un objeto compuesto formado por un conjunto de componentes del mismo tipo. Un componente de un tablero se designa por un índice en un intervalo dado.

```
tipo_tablero  →  array ( expresión .. expresión ) of indicación_de_subtipo
```

Las dos expresiones, que deben ser constantes enteras, indican los límites del intervalo de definición del tablero. Este tendrá, necesariamente, una sola dimensión.

Ejemplo:

```
A, B : array ( 2 .. 20 ) of INTEGER ;
```

□

En particular, las cadenas de caracteres son representadas en NOGO mediante la declaración siguiente:

```
type CADENA is array ( 1 .. MAX ) of CHARACTER ;
```

En relación a los tipos registro, se trata de un objeto compuesto formado por campos nombrados que pueden pertenecer a tipos diferentes:

tipo_registro	→	**record**
		lista_de_componentes
		end record
lista_de_componentes	→	componentes { componentes }
componentes	→	identificadores : indicación_de_subtipo ;
		| identificadores : tipo_tablero ;

La elaboración de un tipo registro consiste en la elaboración de las declaraciones de sus diferentes campos, en el mismo orden en el que aparecen. Una declaración de componentes define uno o varios componentes del mismo tipo. Este tipo no puede ser otro mas que un escalar o un tipo tablero.

Ejemplo:

```
FECHA :   record
          DÍA : INTEGER range 1 .. 31 ;
          MES : INTEGER range 1 .. 12 ;
          AÑO : INTEGER range 0 .. 4000 ;
          end record ;

LÍNEA :   record
          ÍNDICE : INTEGER range 1 .. TALLA ;
          TAMPÓN : array ( 1 .. TALLA ) of CHARACTER ;
          end record ;
```

□

Nombres. Los nombres pueden denotar cualquier entidad declarada, esto es, objetos, tipos y subtipos. Pueden igualmente ser nombres de bucles o representar los componentes de un objeto compuesto. Finalmente, un nombre puede representar el resultado devuelto por una llamada de función. La sintaxis correspondiente es la siguiente:

nombre	→	identificador
		| componente_indexado
		| componente_seleccionado
		| llamada_de_función

Las llamadas de función se describen más tarde en este apéndice. Los otros nombres pasamos a introducirlos de inmediato.

Componentes indexados. Un componente indexado denota un elemento en un tablero:

componente_indexado → nombre (expresión)

El nombre denota un objeto de tipo tablero o la llamada a una función que devuelve un tablero. La expresión especifica el valor del índice para este elemento del tablero. Si la evaluación de esta expresión devuelve un valor fuera de los límites permitidos, el programa se para sobre la excepción CONSTRAINT_ERROR.

Ejemplo:

```
MI_TABLA (19) ;
PÁGINA (I+J)
PANTALLA (I) (J)
```

□

Componentes seleccionados. Un componente seleccionado denota un campo de un registro. La sintaxis es la siguiente:

componente_seleccionado → nombre . identificador

donde el nombre denota un registro, o bien una función que devuelve como valor un registro. El identificador especifica el componente de este registro.

Ejemplo:

CITA . DÍA
COCHE (I) . PROPIETARIO
PERSONA (I) . FECHA_DE_NACIMIENTO . DÍA

□

Literales. Un literal denota un valor explícito de un tipo dado. La sintaxis es:

literal → literal_numérico
| literal_carácter
| cadena_de_caracteres

Ejemplo:

3.154_54	–	un literal
1_789	–	un literal entero
'A'	–	un literal carácter
"Un texto"	–	un literal cadena de caracteres

□

Expresiones. Una expresión define el cálculo de un valor. La sintaxis es la que sigue:

expresión	→	relación { operador_lógico relación }
relación	→	expresión_simple [operador_relacional expresión_simple]
expresión_simple	→	[operador_unario] término { operador_aditivo término }
término	→	factor { operador_multiplicativo factor }
factor	→	primario [** primario]
primario	→	literal
		| nombre
		| llamada_de_función
		| (expresión)

donde cada primario posee un tipo y un valor. El tipo de una expresión depende sólo del tipo de sus elementos y de los operadores que son aplicados. Las reglas que definen los posibles tipos de operandos son ilustrados en el ejemplo que sigue.

Ejemplo:

4.0	–	literal real
VOLUMEN	–	valor de una variable
DIRECCIÓN (3). CIUDAD	–	valor del campo de un registro
SINUS (X)	–	llamada de una función
(X + 4)	–	expresión aritmética

□

Operadores. Los operadores se dividen en seis clases, cuya descripción facilitamos ahora por orden de prioridad creciente:

```
operador_lógico          →  and
                            | or
operador_relacional      →  =
                            | /=
                            | <
                            | <=
                            | >
                            | >=
operador_aditivo         →  +
                            | -
operador_unario          →  +
                            | -
                            | not
operador_multiplicativo  →  *
                            | /
                            | mod
operador_exponencial     →  **
```

donde todos los operandos de un factor, término o expresión simple son evaluados en un orden indefinido antes de la aplicación de la operación correspondiente. Para un término, una expresión simple, una relación o una expresión, los operadores de mayor prioridad se aplican en primer lugar. Para una sucesión de operadores de la misma prioridad, éstos se aplican de izquierda a derecha, salvo en el caso de que los paréntesis modifiquen explícitamente estas prioridades.

Ejemplo:

not SOLEADO **and** CALUROSO	–	idéntico a (not SOLEADO) or CALUROSO
X > 4.0 **and** Y > 0.0	–	idéntico a (X > 4.0) and (Y > 0.0)
- 4.0 * A ** 2	–	idéntico a -(4.0 * (A ** 2))
Y ** (-3)	–	los paréntesis son necesarios
A + (B + C)	–	evalúa B+C antes de sumar A

□

Operadores lógicos. Los operadores lógicos son aplicables a los valores de tipo BOOLEAN.

Operador	Operación	Operandos	Resultado
and	conjunción	BOOLEAN	BOOLEAN
or	disyunción	BOOLEAN	BOOLEAN

Ejemplo:

SOLEADO **and** CALUROSO

□

Operadores relacionales. Los operadores relacionales poseen operandos del mismo tipo, entero, real o carácter, y devuelven valores de tipo BOOLEAN.

Ejemplo:

X /= Y
'A' < 'B' **and** 'B' < 'C' – TRUE

□

Operadores aditivos. En lo que concierne a los tipos numéricos, el programa NOGO se para sobre la excepción NUMERIC_ERROR, en el caso de que la operación no pueda ejecutarse directamente.

Operador	Operación	Operandos	Resultado
+	adición	mismo tipo numérico	mismo tipo numérico
or	sustracción	mismo tipo numérico	mismo tipo numérico

Ejemplo:

Z + 0.1 – Z debe ser de tipo real

□

Operadores unarios. Para los tipos numéricos, el programa NOGO se para sobre la excepción NUMERIC_ERROR, en el caso de que la operación no pueda ejecutarse de forma correcta.

Operador	Operación	Operandos	Resultado
+	identidad	tipo numérico	mismo tipo numérico
-	negación	tipo numérico	mismo tipo numérico
not	negación lógica	BOOLEAN	BOOLEAN

Operadores multiplicativos. Los operadores * y / para los enteros y reales, y el operador **mod** para los enteros, devuelven un resultado del mismo tipo que los operandos:

Operador	Operación	Operandos	Resultado
*	multiplicación	entero	entero
		real	real
mod	módulo	entero	entero

El programa NOGO se para sobre la excepción NUMERIC_ERROR, en el caso de que la operación no pueda ser ejecutada de forma correcta.

Operador exponencial. La exponenciación de un entero por un exponente negativo supone la terminación del programa NOGO sobre la excepción CONSTRAINT_ERROR. La exponenciación supone también la terminación del programa sobre la excepción NUMERIC_ERROR, si el resultado no puede ser calculado de forma correcta.

Operador	Operación	Operando der.	Operando izq.	Resultado
**	exponenciación	entero	entero ≥ 0	entero
		real	entero	real

0.4 Instrucciones

Pasamos a describir en este apartado las reglas generales aplicables al conjunto de instrucciones del lenguaje NOGO. Algunas instrucciones específicas, tales como las llamadas a subprogramas se describen más tarde en este apéndice.

Instrucciones simples y compuestas. Listas de instrucciones. Una instrucción puede ser simple o compuesta. En el primer caso, no contiene otra instrucción, mientras que las compuestas si. La sintaxis es la siguiente:

instrucciones	→	instrucción { instrucción }
instrucción	→	instrucción_simple | instrucción_compuesta
instrucción_simple	→	instrucción_vacía | instrucción_de_asignación | instrucción_exit | instrucción_return | llamada_de_procedure
instrucción_compuesta	→	instrucción_if | instrucción_case | instrucción_loop | instrucción_bloque
instrucción_vacía	→	**null**

donde la ejecución de la instrucción **null** no tiene otro efecto que pasar a la acción siguiente.

Instrucción de asignación. Una asignación reemplaza el valor actual de una variable por un nuevo valor especificado por una expresión. La variable y la expresión deben ser del mismo tipo.

instrucción_de_asignación → nombre := expresión

Después de la ejecución de una asignación, la expresión de la parte derecha, así como toda expresión utilizada en la especificación de la variable, son evaluadas. El valor de la expresión debe satisfacer las restricciones de intervalo que eventualmente se le puedan aplicar a la variable.

Ejemplo:

```
I, J: INTEGER range 1 .. 10 ;
K : INTEGER range 1 .. 20 ;
I := J ; – verificación de tipo superflua
K := J ; – verificación de tipo superflua
J := K ; – verificación de tipo necesaria
```

□

El lenguaje no define si la evaluación de la expresión de la parte derecha precede, sigue o se realiza en paralelo a la de las expresiones que se sitúan en la especificación de la variable. Un programa que realize hipótesis al respecto es erróneo.

En lo que se refiere a la asignación de variables de tipo tablero, los dos tableros implicados deben ser del mismo tipo y poseer los mismos bornes. También es posible asignar un literal cadena de caracteres a un tablero. En el caso de que la cadena posea una longitud inferior al número de elementos del tablero, el resto de dicho tablero será completado por caracteres de espacio en blanco. Si la cadena es de longitud superior al tablero, el programa se para sobre la excepción CONSTRAINT_ERROR.

Ejemplo:

```
UNA_CADENA := “HOLA” ;
```

□

La asignación de una variable registro es ilegal.

Instrucción if. Una instrucción **if** selecciona la ejecución de una secuencia de instrucciones según sea el valor de una expresión de tipo BOOLEAN que representa la condición de control. La sintaxis es la siguiente:

```
instrucción_if  →  if          expresión then
                               instrucciones
                   [else
                               instrucciones ]
                   end if ;
```

En el caso de que la instrucción tenga por valor TRUE, se ejecuta la primera lista de instrucciones. En otro caso, es la segunda lista la que se ejecuta, si la parte **else** está presente.

Ejemplo:

```
if      MES = 12 and DÍA = 31 then
        MES := 1;
        DÍA := 1;
        AÑO := AÑO + 1;
end if;

if      SANGRADO then
        VERIFICAR_MARGEN_DERECHO;
        DECALAR_POR_LA_IZQUIERDA;
else
        CAMBIO_DE_LÍNEA
end if;
```

□

Instrucción case. Una instrucción de este tipo selecciona la ejecución de una instrucción de entre un conjunto dado. La condición de control viene dada por una expresión entera. La sintaxis es la que sigue:

```
instrucción_case  →  case      expresión is
                               { when entrada { | entrada } => instrucciones }
                     end       case ;

entrada           →  expresión
                     | expresión  ..  expresión
                     | others
```

donde cada alternativa viene precedida por una lista de entradas que especifica los valores para los que una alternativa dada será seleccionada. En cualquier caso, el tipo de la expresión que sigue a **case** debe ser un subtipo entero. Las expresiones de entrada deben ser constantes enteras. La entrada **others** no puede aparecer más que una sola vez como única, y última, alternativa. La ejecución de una instrucción **case** se traducirá en la ejecución de una sola de las alternativas presentes.

Ejemplo:

```
type DÍA is INTEGER range 1 .. 7 ;
LUNES : DÍA := 1 ;
...
DOMINGO : DÍA := 7 ;
...
case HOY is
      when LUNES .. JUEVES      => TRABAJO ;
      when VIERNES              => TRABAJO; DEPORTE ;
      when others               => null;
end case ;
```

□

Instrucción loop. Una instrucción **loop** especifica que una lista de instrucciones debe ser ejecutada cero o más veces. La sintaxis es la que sigue:

instrucción_loop	→	[identificador_de_bucle :] [cláusula_de_iteración] bucle_de_base [identificador_de_bucle] ;
identificador_de_bucle	→	identificador
cláusula_de_iteración	→	**for** índice_de_bucle **in** [**reverse**] expresión .. expresión | **while** expresión
bucle_de_base	→	**loop** instrucciones **end loop**
índice_de_bucle	→	identificador

Si un identificador de bucle aparece en una instrucción **loop**, éste debe ser dado al principio y al final. Un identificador de bucle se declara de forma implícita al final de la parte declarativa del bloque o subprograma más interno que contiene al bucle en cuestión. Cuando este bloque no tiene parte declarativa, una parte declarativa ficticia es generada de forma automática por el compilador.

Una instrucción **loop** sin cláusula de iteración especifica una ejecución repetida del bucle de base. Este bucle puede ser terminado por la ejecución de una instrucción **exit** o **return**.

En el caso de considerar una instrucción **loop** con una cláusula de iteración **while**, que debe ser de tipo BOOLEAN, ésta se evalúa y verifica antes de cada ejecución del bucle de base. Si la condición establecida es TRUE, la secuencia de instrucciones en el bucle de base se ejecuta, en otro caso el bucle termina.

La ejecución de una instrucción **loop** con una cláusula de iteración **for** comienza por la elaboración de dicha cláusula, lo que resuelve la declaración del índice del bucle. El identificador de dicho índice es introducido en primer lugar , y luego el intervalo denotado por las dos expresiones es evaluado. El índice de bucle se comporta como una variable de tipo entero, local al bucle de base.

Si el intervalo señalado es nulo, el bucle de base no se ejecuta. En otro caso, la lista de instrucciones del bucle de base pasa a ejecutarse tantas veces como valores enteros se encuentran en el intervalo. Antes de cada iteración, el valor correspondiente del intervalo es asignado al índice del bucle. Estos valores son asignados en orden creciente a menos que la palabra reservada **reverse** esté presente. En este caso, los valores se asignan en orden decreciente.

En el interior del bucle de base, el índice del bucle se comporta como una constante. En consecuencia, éste no puede ser modificado por asignación, ni pasado como parámetro en modo **in out** a una procedure.

Ejemplo:

```
while   OFERTA(N).PRECIO < LÍMITE.PRECIO loop
        REGISTRAR(OFERTA(N).PRECIO) ;
        N := N+1 ;
end     loop ;

for     J in 1 .. N loop
        if TAMPÓN(J) /= ESPACIO then METER(TAMPÓN(J)) ;
        end if ;
end     loop ;
```

□

Bloques. Un bloque introduce una lista de instrucciones opcionalmente precedida por una parte declarativa. La sintaxis es la que sigue:

instrucción_bloque → **declare**
declaraciones
begin
instrucciones
end ;

La ejecución de una instrucción bloque se hace mediante la elaboración de su parte declarativa, seguida por la ejecución de la lista de instrucciones.

Ejemplo:

```
declare
        TEMPORAL: INTEGER ;
begin
        TEMPORAL := V; V := U; U := TEMPORAL ;
end ;
```

□

Instrucción exit. Una instrucción **exit** provoca la terminación del bucle que la engloba. La sintaxis es la que sigue:

instrucción_exit → **exit** [identificador_de_bucle] [**when** expresión] ;

El bucle que termina el **exit** es el más interno conteniendo dicha instrucción, a menos que el **exit** especifique un identificador de bucle. En este último caso, es el bucle referido el que se termina.

Si una instrucción **exit** contiene una cláusula **when**, la expresión, que necesariamente es de tipo BOOLEAN, es evaluada y la terminación no se produce más que si el valor de dicha expresión es TRUE.

Una instrucción **exit** sólo puede aparecer al interior de un bucle.

Ejemplo:

```
for     I in 1 .. MAX loop
        LEER_ELEMENTO(ELEM) ;
        TRATAR(ELEM) ;
        exit when ELEM = ELEMENTO_TERMINAL ;
end     loop

CICLO:
loop
        – instrucciones iniciales
        exit CICLO when ENCONTRAR;
        – instrucciones finales
end     loop CICLO ;
```

□

0.5 Subprogramas

Un subprograma es una unidad ejecutable susceptible de ser llamada desde un programa o subprograma. Hay dos formas de subprogramas en NOGO: Las procedures y las funciones. Una llamada a una procedure es una instrucción, una llamada a una función devuelve un valor.

Declaraciones de subprogramas. Una declaración de subprograma declara, bien una procedure, bien una función. La sintaxis es la que sigue:

declaración_de_subprograma → especificación_de_subprograma
[cuerpo_de_subprograma] ;

especificación_de_subprograma → **procedure** identificador [(parte_formal)]
→ | **function** identificador [(parte_formal)]
return indicación_de_subtipo

parte_formal → [declaración_de_parámetros { ; declaración_de _parámetros }]

declaración_de_parámetros → identificadores : modo indicación_de_subtipo

modo → [**in**]
| **in out**

cuerpo_de_subprograma → **is**
declaraciones
begin
instrucciones
end [identificador] ;

La especificación de un subprograma introduce su identificador y sus parámetros, si los hay. En relación a las funciones, el tipo de valor devuelto debe especificarse igualmente.

En lo que se refiere a la elaboración de una declaración de subprograma, el identificador que define su nombre es introducido en primer lugar. Una vez hecho esto, puede ser utilizado como el nombre del subprograma correspondiente.

En el caso de la elaboración de una declaración de parámetro, los identificadores de la lista son primero introducidos, para luego establecer su modo y tipo.

La llamada a un subprograma provoca su ejecución, especificada por su cuerpo. En primer lugar se ponen en correspondencia los parámetros reales y los formales según las reglas que definiremos más tarde. La parte declarativa del programa es entonces elaborada, y la lista de instrucciones del cuerpo, ejecutada. Tras la terminación de la ejecución se vuelve al lugar de la llamada al subprograma.

Ejemplo:

```
procedure  EMPILAR    ( E: in INTEGER; P: in out array PILA ) is
begin
           if         P.ÍNDICE = P.TALLA then
                      ERROR := TRUE ;
           else
                      P.ÍNDICE := P.ÍNDICE + 1 ;
                      P.TAB(P.ÍNDICE) := E ;
           end if ;
end        EMPILAR ;
```

□

Un subprograma no puede llamarse si no ha sido previamente declarado. En el caso de ser objeto de una recursividad indirecta, es necesario que aparezca primero la especificación de un subprograma sin su cuerpo. La misma declaración deberá aparecer después con el correspondiente cuerpo.

Ejemplo:

```
procedure   P (A: INTEGER) ;

procedure   Q (B: REAL) is
begin
                ...
                P(3) ;
end             Q ;

procedure   P (A: INTEGER) is
begin
                ...
                Q(5.0) ;
end             P ;
```

□

Parámetros formales. Los parámetros de un subprograma se consideran variables locales al mismo. Hay dos modos posibles para un parámetro:

in El parámetro se comporta como una constante local, donde el valor viene dado por el parámetro real correspondiente.

in out El parámetro se comporta como una variable local, donde el valor inicial viene dado por el parámetro real correspondiente. Permite la afectación de dicho parámetro real.

Una función sólo permite parámetros en modo **in**.

En el caso de no explicitar el modo de los parámetros de un subprograma, por defecto consideraremos que es de modo **in**. Si un parámetro en modo **in** es un tablero o un registro, ninguno de sus componentes podrá ser modificado mediante asignaciones. Para los parámetros de tipo escalar, al principio de cada llamada, el valor de cada uno de los parámetros reales es copiado en el parámetro formal correspondiente. Una vez finalizada la ejecución, el valor de cada parámetro formal de modo **in out** es vuelto a copiar en el parámetro real correspondiente.

En lo que se refiere a los parámetros de tipo tablero o registro, el parámetro formal puede dar acceso al real correspondiente, durante la ejecución del subprograma.

En el caso de existir caminos de acceso múltiples a un parámetro de los descritos, por ejemplo si se trata de una variable global que a la vez es utilizada como parámetro real, si se realizase una asignación del parámetro real en el subprograma por un acceso que no sea el parámetro real, entonces el valor del parámetro formal debe ser indefinido. Un programa NOGO que utiliza valores indefinidos se considera erróneo.

Llamadas a subprogramas. Una llamada a un subprograma es, bien una llamada a una procedure, bien una llamada a una función. La llamada especifica la correspondencia entre los valores de los parámetros reales y los formales del subprograma. Un parámetro real es bien una variable, bien el valor de una expresión. La sintaxis es la siguiente:

```
llamada_de_procedure  →  identificador [ parámetros_reales ] ;

llamada_de_función    →  identificador parámetros_reales
                         | identificador ( )

parámetros_reales     →  ( expresión { , expresión } )
```

La llamada a una función sin parámetros se hace dando el nombre de la función seguido por dos paréntesis vacíos. La llamada de una procedure sin parámetros se hace dando el nombre de la misma seguido por un punto y coma.

Ejemplo:

```
IR_A_LA_DERECHA ;
BUSCAR ( CADENA, POSICIÓN_ACTUAL, NUEVA_POSICIÓN ;
HORA := CLOCK ( ) ;
```

□

En cualquier caso, la expresión utilizada como parámetro real de modo **in out** debe ser una variable. La expresión utilizada como parámetro real de modo **in** se evalúa antes de la llamada. Si una variable dada como parámetro real de modo **in out** es un componente de un tablero o de un registro, su identidad se establece antes de la llamada.

Para un parámetro de tipo escalar, toda restricción de intervalo sobre el parámetro formal debe ser satisfecha por el valor del parámetro real antes de cada llamada. Si el modo es **in out**, toda restricción de este tipo sobre el parámetro real debe ser satisfecha por el valor del parámetro formal una vez la ejecución haya vuelto al programa de llamada.

Procedures de entrada y salida. Existen dos procedures predefinidas READ y WRITE que permiten leer y escribir la representación textual de un valor de tipo escalar cualquiera sobre el flujo de entrada y el de salida. Es igualmente posible escribir una cadena de caracteres.

Ejemplo:

```
READ(A) ;

WRITE("HOLA") ;

WRITE(X+3) ;

WRITE(SALARIO > 5_000) ;
```

□

Estructura de un subprograma. Un subprograma NOGO es una procedure sin parámetros.

Ejemplo:

```
procedure   HOLA is
begin
            WRITE("BUENOS DÍAS") ;
end         HOLA ;
```

□

0.6 Reglas de visibilidad

Las reglas de visibilidad definen el ámbito de las declaraciones y las reglas que establecen que identificadors son visibles en una zona dada de un programa NOGO.

Definiciones. Una declaración asocia un identificador con una entidad del programa, por ejemplo una variable, un subprograma, un parámetro formal, o un campo de un registro. La región del texto donde la declaración tiene efecto se denomina *ámbito de la declaración*. Esta región comienza allí donde el identificador declarado ha sido introducido.

El mismo identificador puede ser introducido por diferentes declaraciones en el texto de un programa y puede, por tanto, ser asociado a diferentes entidades. Los ámbitos referentes a declaraciones diferentes de un mismo identificador pueden dar lugar a fenómenos de superposición.

Los ámbitos de declaraciones superpuestas pueden obtenerse, por ejemplo, teniendo un mismo nombre de campo para varios registros. De forma habitual estos recubrimientos se producen por imbricación de las unidades del programa considerado, bloques y subprogramas.

La declaración de una entidad por un identificador se dice *visible* en un punto dado del programa cuando su utilización en dicho punto hace referencia a esta entidad.

Ambito de las declaraciones. Las entidades de un programa pueden ser declaradas de formas diversas. Así, en la parte declarativa de un bloque o subprograma, o como parámetro de un subprograma. Una entidad puede ser declarada como el componente de un registro. Un parámetro de bucle es declarado de forma implícita para su uso en la cláusula de iteración. En la misma línea, la declaración de un identificador de bucle es implícita.

El ámbito de cada forma de declaración se define por las reglas que pasamos a presentar. Cuando decimos que el ámbito se extiende a partir de la declaración, ello significa que el ámbito se extiende a partir del punto donde el identificador es introducido. Las reglas son las que siguen:

- El ámbito de una declaración que aparece en la parte declarativa de un bloque, subprograma, o en la parte formal de una especificación de subprograma, se extiende a partir de la declaración hasta el fin del bloque o subprograma.
- El ámbito del campo de un registro se extiende desde la declaración de dicho campo hasta el fin del ámbito del tipo registro.
- El ámbito de un identificador de bucle se extiende a partir de su aparición en la cláusula de iteración hasta el fin del bucle correspondiente.

Visibilidad de los identificadores. Para cada declaración, existe un subconjunto de su ámbito donde la entidad declarada puede ser simplemente nombrada por su identificador. La entidad, su declaración y su identificador son directamente visibles a partir de este subconjunto. En la región del ámbito donde no hay visibilidad directa, un contexto apropiado puede hacer visible la entidad en cuestión. Este contexto puede ser el prefijo de un campo de un registro.

Una entidad declarada inmediatamente en una entidad se dice *local a dicha unidad*. Una entidad visible en una unidad, pero declarada al exterior de esta se dice *global a esta unidad*.

Para cada forma de declaración y al interior de su ámbito, la región de texto en la que un identificador declarado es visible, a menos que esté superpuesto por otra entidad con el mismo nombre en una unidad imbricada, se define como sigue:

- Un identificador declarado en la parte declarativa de un bloque o subprograma, es visible en este bloque o subprograma.
- El identificador de un campo de un registro es directamente visible al interior de la definición del tipo registro. Fuera de ésta, pero al interior de su ámbito, el componente es visible si incluimos un prefijo con el tipo registro del que es componente.
- El identificador de un parámerto formal de un subprograma es visible en la parte formal donde este parámetro ha sido declarado, y en el cuerpo del subprograma.
- El identificador de un índice de bucle es visible al interior de un bucle.

Ejemplo:

```
procedure   P is
            A: BOOLEAN ;
            B: BOOLEAN ;
            procedure       Q is
                            C: BOOLEAN ;
                            B: BOOLEAN ;    – redeclaración de B
            begin
                            B:= A ;         – B local, A global
            end Q
begin
            A:= B ;                         – A y B declaradas en P
end P ;
```

□

Entorno predefinido. Todos los identificadores predefinidos, esto es, los de tipos como INTEGER, REAL, BOOLEAN y CHARACTER, y los de las procedures de entrada y salida READ y WRITE, se suponen definidos en un entorno situado al nivel más externo del programa.

0.7 Un ejemplo

Para finalizar este apéndice, incluimos un ejemplo de programa NOGO. Se trata de una implementación del algoritmo de ordenación conocido habitualmente como *"heapsort"*, u *ordenación por montículos*.

Como algoritmo perteneciente a lo que conocemos como *métodos de ordenación por selección*, en los que la idea fundamental reposa sobre el principio de seleccionar el elemento más pequeño en una lista para luego ordenar el resto de ésta. Ello se traduce en la búsqueda del elemento minimal, su inclusión en la primera posición de la lista y repetición del proceso sobre la parte de la lista no ordenada, en cuya parte final hemos colocado el elemento que antes se encontraba en el primer lugar. Un ejemplo de ejecución se muestra en la figura 0.1.

El rasgo que singulariza a los distintos métodos de selección está en la estrategia aplicada para guardar, entre dos recorridos de la lista, el resultado de las comparaciones efectuadas. En el caso que nos ocupa se utiliza una estructura de datos denominada *"heap"*, que nosotros traducimos libremente por el término montículo o montón.

Datos	| 101	115	30	63	47	20
Selección						**20**
Datos	**20**	| 115	30	63	47	101
Selección			**30**			
Datos	**20**	**30**	| 115	63	47	101
Selección					**47**	
Datos	**20**	**30**	**47**	| 63	115	101
Selección				**63**		
Datos	**20**	**30**	**47**	**63**	| 115	101
Selección						**101**
Datos	**20**	**30**	**47**	**63**	**101**	**115**

Figure 0.1: Ordenación por selección de la lista $[101, 115, 30, 63, 47, 20]$

En nuestra implementación, hemos considerado dos procedures locales al programa principal. La primera, METER(I, N), busca en el *"heap"* a partir de la posición I, el valor minimal. La segunda, CONSTRUIR_HEAP, realiza la construcción física del *"heap"* ordenado mediante un bucle en I de llamadas a METER(I, N).

La representación de los montones se realiza mediante un árbol binario equilibrado implementado en forma de tablero TABLA(I). Aplicamos las siguientes reglas:

- La raíz viene dada por TABLA(1).
- El nodo TABLA(I/2) es el padre de TABLA(I), para I > 1.
- En caso de existir hijos para el nodo TABLA(I), éstos son TABLA(2*I) y TABLA(2*I+1).
- Si 2*I=N, entonces TABLA(I) sólo posee un hijo, TABLA(N).
- Si I $\geq$ N/2, entonces TABLA(I) es una hoja.

En estas condiciones, el código del programa es el siguiente:

```
procedure ORDENAR_POR_MONTONES is

   N: constant INTEGER := 10;
   TABLA: array(1..N) of INTEGER;
   TEMPORAL: INTEGER;

   procedure METER(I,N: in out INTEGER) is
      TEMPORAL,J: INTEGER;
      TERMINAR: BOOLEAN;
   begin
```

```
      TEMPORAL :=  TABLA(I);
      J :=  2*I;
      TERMINAR :=  FALSE;

      while (J <= N) and not TERMINAR loop
         if (J < N) and (TABLA(J) < TABLA(J+1)) then
            J := J+1;
         end if;
         if TEMPORAL > TABLA(J) then
            TERMINAR := TRUE;
         else
            TABLA(J/2) := TABLA(J);
            J := 2*J;
         end if;
      end loop;

      TABLA(J/2) := TEMPORAL;

   end METER;

   procedure CONSTRUIR_MONTON is
   begin
      for I in reverse 1 .. N/2 loop
         METER(I,N);
      end loop;
   end CONSTRUIR_MONTON;

begin

   for I in 1 .. N loop
      READ(TABLA(I));
   end loop;

   CONSTRUIR_MONTON;

   for I in reverse 1 .. N-1 loop
      TEMPORAL := TABLA(I+1);
      TABLA(I+1) := TABLA(1);
      TABLA(1) := TEMPORAL;

      METER(1,I);
   end loop;

   for I in 1 .. N loop
      WRITE(TABLA(I));
   end loop;

end ORDENAR_POR_MONTONES;
```