

# Lenguaje Ada

MiniManual



## Sintaxis y Semántica del lenguaje Ada. Ejemplos

Prf. Maite Urretavizcaya

Dpt. de Lenguajes y Sistemas Informáticos

Facultad de Informática. San Sebastián

UPV-EHU

### INDICE

<b>1. INTRODUCCIÓN .....</b>	<b>3</b>
<b>2. TIPOS DE DATOS BÁSICOS .....</b>	<b>7</b>
2.1. INTEGER, FLOAT, CHARACTER, BOOLEAN Y STRING .....	9
2.2. ATRIBUTOS DE TIPOS .....	18
2.3. CONVERSIÓN Y COMPATIBILIDADES DE TIPOS .....	20
<b>3. ESTRUCTURA GENERAL DE UN PROGRAMA EN ADA .....</b>	<b>22</b>
3.1. COMENTARIOS .....	23
3.2. CLÁUSULAS DE CONTEXTO .....	24
3.3. ASIGNACIÓN .....	25
<b>4. ENTRADA Y SALIDA .....</b>	<b>27</b>
4.1. ENTRADA Y SALIDA ESTÁNDAR .....	30
4.1.1. Salida por pantalla .....	32
4.1.2. Entrada desde teclado .....	34
4.2. ENTRADA Y SALIDA NO ESTÁNDAR- FICHEROS DE TEXTO .....	35
<b>5. ESTRUCTURAS DE CONTROL .....</b>	<b>41</b>
5.1. ESTRUCTURAS CONDICIONALES: IF Y CASE .....	41
5.2. ESTRUCTURAS ITERATIVAS: LOOP, WHILE Y FOR .....	44
<b>6. SUBPROGRAMAS .....</b>	<b>51</b>
6.1. PROCEDIMIENTOS Y FUNCIONES .....	51
6.2. SUBPROGRAMAS Y CLÁUSULAS DE CONTEXTO .....	55
6.3. VALORES POR DEFECTO Y SOBRECARGA .....	57
<b>7. DEFINICIÓN DE TIPOS .....</b>	<b>58</b>
7.1. TIPO ENUMERADO .....	58
7.2. SUBTIPOS DE RANGO .....	59
7.3. REGISTROS: RECORD .....	59
7.4. VECTORES Y MATRICES: ARRAY .....	60

## 1. Introducción

Promovido por el Dpto. de defensa de USA. Características:

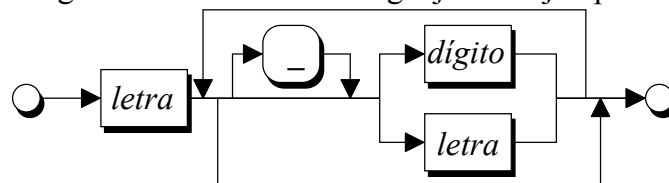
- ◆ Fuertemente tipado
- ◆ Programación imperativa (tradicional)
- ◆ Programación orientada a objetos
- ◆ Programación concurrente y distribuida

Su nombre se debe a Augusta **Ada** Byron, condesa de Lovelace, (1815-1852)

- ♣ Matemática e hija de Lord Byron
- ♣ Trabajó con Babbage y su «Máquina analítica» (ordenador mecánico).
- ♣ Desarrollaron las ideas básicas de la programación
- ♣ Considerada la primera programadora.

### Identificadores. Diagrama sintáctico

Sirve para describir reglas sintácticas de un lenguaje. Por ejemplo un identificador en Ada:



### Ejemplos de identificadores Ada:

Obtener_Datos	Devolver_resultados
contador	X1
Indice_2	

En Ada, los identificadores que sólo se diferencien en las mayúsculas y minúsculas se consideran el mismo:

Suma = suma = SUMA = SuMa

La longitud máxima del identificador la define cada compilador concreto de Ada.

## Gramática BNF (Backus-Naur Form)

Es otra forma de describir la sintaxis. Por ejemplo el identificador Ada:

Identificador ::= Letra { [ \_ ] Letra\_o\_Dígito }

Letra\_o\_Dígito ::= Letra | Dígito

Dígito ::= 0|1|2|3|4|5|6|7|8|9

Letra ::= A|B|C|...|Y|Z|a|...|x|y|z

En BFN hay símbolos que se expanden como Letra, Letra\_o\_dígito y Dígito. Otros son símbolos terminales como `_`, `A`, ..., `Z`, `a`, ..., `z`, `0` ..., `9`

[ ... ] Lo que aparece entre corchetes es opcional (aparece 0 ó 1 vez)

{ ... } Lo que aparece entre llaves puede repetirse (aparece 0, 1 ó más veces)

| Separa las posibles alternativas (elegir una de ellas)

## Palabras reservadas en Ada

Identificadores reservados por el lenguaje. Tienen un significado especial y no pueden usarse para nombrar variables, tipos, etc.

Palabras reservadas de Ada:

<b>abort</b>	<b>declare</b>	<b>generic</b>	<b>of</b>	<b>rem</b>	<b>type</b>
<b>abs</b>	<b>delay</b>	<b>goto</b>	<b>or</b>	<b>renames</b>	
<b>accept</b>	<b>delta</b>		<b>others</b>	<b>requeue</b>	<b>until</b>
<b>access</b>	<b>digits</b>	<b>if</b>	<b>out</b>	<b>return</b>	<b>use</b>
<b>all</b>	<b>do</b>	<b>in</b>		<b>reverse</b>	
<b>and</b>		<b>is</b>	<b>package</b>		<b>when</b>
<b>array</b>	<b>else</b>		<b>pragma</b>	<b>select</b>	<b>while</b>
<b>at</b>	<b>elsif</b>	<b>limited</b>	<b>private</b>	<b>separate</b>	<b>with</b>
	<b>end</b>	<b>loop</b>	<b>procedure</b>	<b>subtype</b>	
<b>begin</b>	<b>entry</b>		<b>protected</b>		<b>xor</b>
<b>body</b>	<b>exit</b>	<b>mod</b>		<b>tagged</b>	
	<b>exception</b>	<b>new</b>	<b>raise</b>	<b>task</b>	
<b>case</b>	<b>for</b>	<b>not</b>	<b>range</b>	<b>terminate</b>	
<b>constant</b>	<b>function</b>	<b>null</b>	<b>record</b>	<b>then</b>	

## 2. Tipos de datos básicos

### Definiciones

Los **tipos escalares** se clasifican en: tipos *enumerados* (Entre ellos los caracteres y booleanos. Se hablará más de los enumerados en el capítulo 4), tipos *enteros* y tipos *reales*. Todos los tipos escalares admiten los operadores relacionales

**Ejemplos:** integer (enteros), float (reales), character (carácter) y boolean (cierto o falso)

Los **tipos discretos** (u ordinales) son tipos de datos escalares donde cada valor (excepto el primero) tiene un único predecesor y (excepto el último) un único sucesor. Los tipos enumerados y los tipos enteros pertenecen a la categoría de tipos *discretos* u *ordinales* que se caracterizan porque a cada valor le corresponde una posición representada por un número natural.

### **Integer** (números enteros)

números positivos y negativos que no incluyen parte fraccionaria. Constan de signo y dígitos.

Ejemplos: 3 -15 0 3942 +227 4.32

### **Float** (números reales)

números con parte entera y parte fraccionaria.

Ejemplos: 3.1415 0.0 -1.0 8.7E+12 9.2E-10

### **Character** (caracteres)

caracteres alfanuméricos: letras, dígitos y símbolos especiales.

Ejemplos: 'A' 'a' '3' '\$' '?'

### **Boolean** (booleanos)

dos valores posibles: False y True (falso y cierto).

### **String**

cadena de caracteres alfanuméricos (cadena de elementos de tipo character).

Ejemplos: "Facultad" "3,14" "4&/%kk ..."

## 2.1. Integer, Float, Character, Boolean y String

### Tipo Integer Números enteros

**Dominio:** [Integer'first, Integer'last] = [-2147483648, 2147483647]

#### Operaciones

- Operadores relacionales (Or)  
{=, /=, <, >, <=, >=}  
Or: Integer x Integer → Boolean
- Operadores aritméticos binarios: Oab  
{+, -, \*, /, rem, mod}  
Oab: Integer x Integer → Integer
- Operadores aritméticos unarios: Oau  
{+, -, abs}  
Oau: Integer → Integer

#### Subtipo: natural y positive

**Subtipo** es un tipo definido basándose en otro tipo, por ejemplo restringiendo el dominio de valores.

**natural** incluye los enteros desde el 0 al máximo entero representable.

**positive** incluye desde el 1 al máximo entero representable.

#### Cálculo del resto

Habitualmente lo utilizaremos para obtener el resto de dos números positivos. En esos casos los valores obtenidos por Rem y mod son equivalentes. Es decir, **Rem**(a nder) y **Mod** calculan el resto de la división entero cuando los valores de los operandos son positivos.

$$12 \text{ rem } 5 = 2$$

$$12 \text{ mod } 5 = 2$$

$$A = (A/B) * B + (A \text{ rem } B)$$

(A rem B) tiene el mismo signo que A y valor absoluto < B

$$A = B * N + (A \text{ mod } B) \text{ siendo } N \text{ algún entero}$$

(A mod B) tiene el mismo signo que B y valor absoluto < B

Si un operando es negativo el resultado difiere:

$$12/-5 = -2 \quad 12 \text{ rem } -5 = 2 \quad 12 \text{ mod } -5 = -3$$

$$14/-5 = -2 \quad 14 \text{ rem } -5 = 4 \quad 14 \text{ mod } -5 = -1$$

$$-12/5 = -2 \quad -12 \text{ rem } 5 = -2 \quad -12 \text{ mod } 5 = 3$$

$$-14/5 = -2 \quad -14 \text{ rem } 5 = 4 \quad -14 \text{ mod } 5 = 1$$

## Tipo Float

Números reales

### Dominio

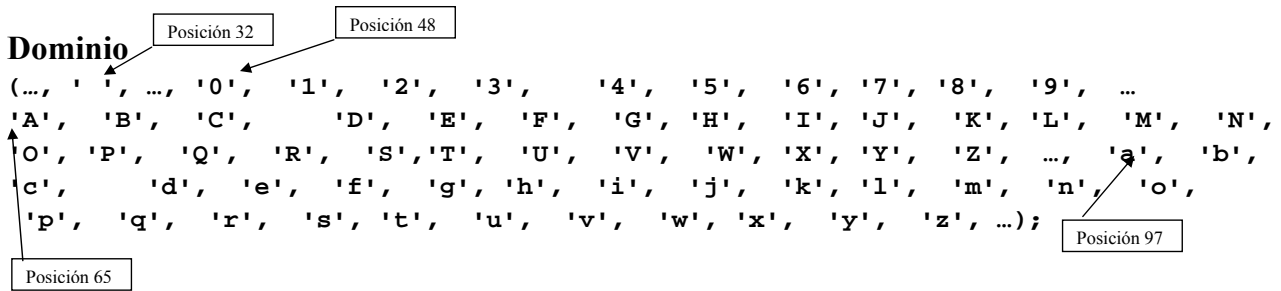
[Float'first, Float'last] = [-3.40282E+38, 3.40282E+38]

### Operaciones

- Operadores relacionales (Or)  
{ =, /=, <, <=, >, >= }  
Or: Float x Float  $\rightarrow$  Boolean
- Operadores aritméticos unarios: Oau  
{ +, -, abs }  
Oau: Float  $\rightarrow$  Float
- Operadores aritméticos binarios: Oab  
{ +, -, \*, / }  
Oab: Float x Float  $\rightarrow$  Float
- Exponencial, Exp { \*\* }  
Exp: Float x **Integer**  $\rightarrow$  Float

## Tipo Character

Conjunto de caracteres alfanuméricos incluidos en el juego de caracteres del ordenador (256 caracteres).



### Operaciones

#### - Operadores relacionales (Or)

{=, /=, <, <=, >, >=}

Or: Character x Character → Boolean

Las comparaciones se realizan en función del orden en el que se encuentran los elementos en el dominio (alfabético para las letras, pero menores las mayúsculas, etc)

## Tipo Boolean

Valores lógicos

**Dominio:** (False, True)

### Operaciones

#### - Operadores relacionales (Or)

{=, /=, <, <=, >, >=}

Or: Boolean x Boolean → Boolean

#### - Operadores lógicos (Ol)

{and, or, xor, not}

Ol: Boolean x Boolean → Boolean

Cortocircuitos (short cuts) **and then** y **or else**:

dan idéntico resultado que **and** y **or**. La diferencia radica en el modo de evaluación.

- **expresion1 and expresion2** --Se evalúan las dos expresiones. No se sabe cuál de ellas se evalúa antes, quizás en paralelo.
- **expresion1 and then expresion2** --Primero se evalúa expresión1. Si es cierta se evalúa expresión2. Pero si es falsa, se devuelve falso **sin evaluar** expresión 2.
- **expresion1 or else expresion2** -- Primero se evalúa expresión1. Si es cierta no se evalúa expresión2  
-- (devuelve cierto).

## Tipo string

Un objeto de este tipo contiene una secuencia de caracteres.

**Dominio** (string de n posiciones)

Todas las posibles secuencias de n caracteres (tipo character). (En cualquier orden y con repeticiones)

### Operaciones

- Operador Substring  
nombre (6..9) = "Mari"  
nombre (1..n+2) "Juan"  
    (supuesto que n = 2)  
nombre (2..1) = ""  
    (string vacío)  
nombre (1..1) = "J"
- Operador Concatenación  
nombre = "Juan  
"&"Maria"  
nombre = "Juan Mari" &  
'a'
- Operador selección  
carácter  
nombre(6) = 'M'  
nombre(1) ≠ "J"
- Operadores relacionales (Or)  
{=, /=, <, <=, >, >=}  
Or: String x String → Boolean
- La ordenación sigue las pautas del tipo character.
- Se pueden comparar strings de distinta longitud

- Ejemplos,

```
"hipopotamo" < "pato"  
"PC" /= "pc"  
"pata" < "patata"
```

```
nombre: string(1..4); -- declaracion  
...  
nombre:="c";  
nombre:="cacofonia";
```

El compilador puede avisar.  
CONSTRAINT\_ERROR en ejecución.



## Precedencia de operadores

El orden de precedencia, de mayor a menor, donde las operaciones de una misma línea tienen igual precedencia es

+	**	abs	not						
	*	/	mod	rem					
	+	-							(operadores
	+	-	&						
	=	/=	<	<=	>	>=	in	not	
-	↓	and	or	xor					

En una expresión se evalúan primero los operadores con mayor precedencia.

La evaluación de operadores de igual precedencia se realiza de izquierda a derecha.

Para cambiar el orden de precedencia se utilizan paréntesis.

Comparar las siguientes expresiones booleanas

**not(5=3 or 4<6)**

**not 5=3 or 4<6**

## 2.2. Atributos de tipos

Son operaciones presentes en Ada que proporcionan información sobre un determinado tipo de dato.

- **Para tipos escalares** (first y last)

Los tipos escalares son aquellos cuyos valores tienen un orden establecido.

Ejemplos: Integer, Float, Character, Boolean (False < True)

Integer'first,	Integer'last
Float'first,	Float'last
Character'first,	Character'last

- **Atributos para tipos discretos** (pred, succ, val y pos)

Los tipos discretos (u ordinales) son tipos de datos escalares donde cada valor (excepto el primero) tiene un único predecesor y (excepto el último) un único sucesor.

Ejemplos: Integer, Character, Boolean.

Character'pred('B')='A'	Integer'pred(25)=24	Boolean'pred(True)=False
Character'succ('A')='B'	Integer'succ(24)=25	Boolean'succ(False)=True
Character'pos('B')= 66	Integer'pos(25)=25	Boolean'pos(False)=0
Character'val(66)='B'	Integer'val(25)=25	B:=True; Boolean'pos(B)=1
	Integer'pos(-25)=-25	Boolean'val(0)=False

### Otros atributos (image, value, min, max, ...)

integer'image(45)=" 45"

integer'image(-2)="-2"

float'image(2.0)="2.0"

devuelve el string que contiene el valor del tipo

integer'value("2")=2

float'value("2.0")=2.0

operación inversa a image. Obtiene un valor del tipo que corresponde a la representación dada en forma de string

integer'min(10,2)=2

integer'min(2,10)=2

integer'max(10,5)=10

dados dos valores devuelven el mínimo y máximo respectivamente

Sólo para flota (rounding, truncation, floor, ceiling) :

float'rounding(2.5)=3.0

float'rounding(2.4)=2.0

redondea los decimales

float'floor(2.1)=2.0

mayor entero (tipo float) que no es mayor que 2.1

float'truncation(2.6)=2.0

trunca los decimales

float'ceiling(2.1)=3.0

menor entero (tipo float) que no es menor que 2.1

## 2.3. Conversión y Compatibilidades de tipos

### Conversión de tipo

Existen una función predefinida para cada (sub)tipo: Float, Integer, Natural, ...

#### Ejemplos

Float(1)=1.0

Natural(0.5E-2)=0

Integer(-3.49999)=-3

Integer(3.5)=4

Integer(3.49999)=3

Integer(-2.55555)=-2

La conversión de entero a real es trivial.

La de real a entero es por redondeo:

Integer(N) = Parte entera de (N±0.5)

### Compatibilidad de tipos

Comprobación automática de rango que consiste en la detección de la asignación de un valor fuera del rango de una variable.

#### Ejemplos:

Sueldo\_Eufrasio: positive;

-- declaración

Sueldo\_Eufrasio := 0;

--Error en compilación

...

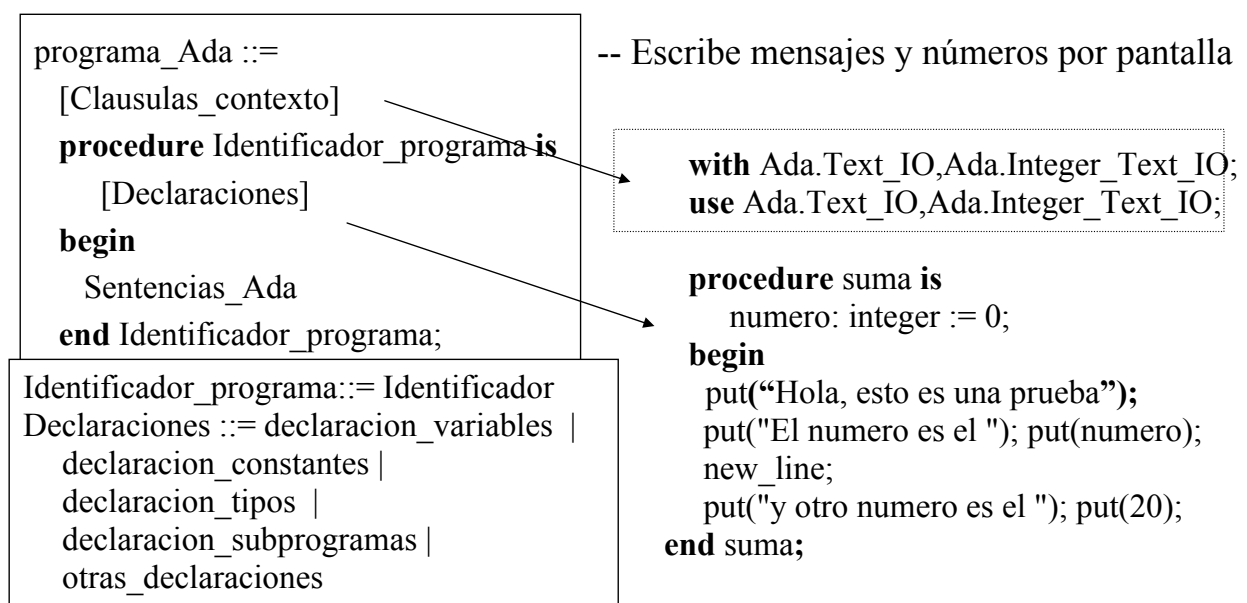
## Compatibilidades de tipos en Ada

- A) Tienen el mismo identificador de tipo.
- B) El dominio del tipo de uno está incluido en el dominio del tipo del otro.
- C) Ambos son subtipos del mismo tipo discreto (ejemplo: positive con natural, subtipos de integer, que es un tipo discreto)

En cada asignación, el compilador comprueba si variable y expresión tienen tipos compatibles. Puede suceder que variable y expresión sean compatibles, pero dar en ejecución un error CONSTRAINT\_ERROR:

```
n: natural:=0; p: positive; -- declaración
p:=n; -- CONSTRAINT_ERROR en ejecución
```

## 3. Estructura general de un programa en Ada



### 3.1. Comentarios

Información incorporada en el código del programa cuya única finalidad es ser útil en la comprensión del mismo.

- No afectan a la ejecución del programa.
- Comienza con dos guiones y termina al final de la línea.
- Se pueden colocar en cualquier línea.

```
Sentencias_Ada ::= Sentencia_Ada
  {Sentencia_Ada | Comentario}
Sentencia_Ada ::= Sentencia_del_programa |
  Sentencia_el_programa Comentario

Sentencia_del_programa ::= Asignacion |
  OperacionE/S | Sentencia_condicional |
  Sentencia_iterativa

Comentario ::= -- {Cualquier_carácter}
```

Ejemplos:

```
-- Cálculo del total
Total := Parcial1 + Parcial2 + Parcial3;

-- Un comentario largo puede dividirse
-- en varias líneas
```

### 3.2. Cláusulas de contexto

Permiten usar en el programa operaciones definidas en otros lugares.

Es necesario incluir cláusulas para utilizar cualquier **operación de entrada o salida**

```
with Ada.Text_IO; use Ada.Text_IO; -- (1)
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO; -- (2)
with Ada.Float_Text_IO; use Ada.Float_Text_IO; -- (3)
procedure ejemplo is
  --declaraciones
begin
  -- Sentencias del programa:
  ...
end ejemplo;
```

**(1)** son las cláusulas necesarias para `string` y `character`. Además de **(1)** se necesita **(2)** para `integer` y sus subtipos y **(3)** para `float`.

### 3.3. Asignación

Asignacion ::= Identificador := Expresion;

Es preciso que haya compatibilidad de tipos → Tipo (variable)=Tipo (expresión)

#### Declaración

Sentencia que asocia un identificador a una variable, constante, tipo, etc. El identificador permite hacer referencia a ese elemento.

Antes de utilizar cualquier identificador hay que declararlo

Declaracion\_variables ::= Identificadores\_var : Identificador\_Tipo [:= Expresion];

Declaración\_constante ::= Identificadores\_const : **constant** Identificador\_Tipo:= Expresión;

Identificadores\_var ::= Identificadores

Identificadores\_const ::= Identificadores

Identificadores ::= Identificador {, Identificador}

Ejemplos: observar los literales

- **Declaración de variables:**

- Sin inicialización:  
Num, Contador, i: integer;  
Linea: string(1..80);
  - Con inicialización:  
Ordenado: boolean := false;  
Maximo: integer := 50;  
Ultimo\_caracter: character := '!';

- **Declaración de constantes:**

- Entidad: **constant** string(1..4) := "ACME";  
Minimo: **constant** := -8;  
Pi: constant Float := 3.1416;  
Dos\_Pi: **constant** Float := 2.0\*Pi;

## 4. Entrada y salida

### Tipos de memoria:

#### Principal

- En variables de programas.
- Los datos almacenados se pierden al apagar el ordenador.
- El programa accede a las variables de modo directo.

#### Secundaria (por ejemplo discos)

- En ficheros.
- Los datos almacenados en ficheros permanecen aunque se apague el ordenador.
- El programa puede **acceder** a datos de ficheros mediante operaciones de **lectura** sobre variables.
- El programa puede **guardar** datos en ficheros mediante operaciones de **escritura**.

### Memoria principal

- **Variables:** tipos numérico, booleano, caracter, string, enumerado, rango, registro o tabla.
- **Asignación** de valores: con “:=”
- **Consulta** de valores: al usar los valores de variables

```
Total:= Parcial * 1.16;
```

**Los valores contenidos en las variables de un programa sólo permanecen durante su ejecución.**

### Memoria secundaria. Datos permanentes

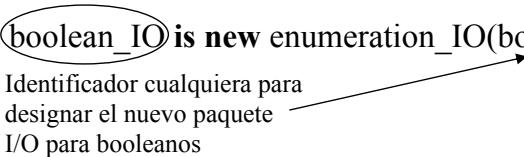
- Lecturas con `get` desde teclado (dispositivo de entrada estándar) o fichero de entrada
- Escrituras con `put` en pantalla (dispositivo de salida estándar) o fichero de salida

Los datos almacenados en ficheros pueden leerse tantas veces como se precise y desde diferentes programas.

El primer paso para poder realizar cualquier operación de lectura o escritura es incluir las cláusulas de contexto adecuadas.

- Tipo Character y String:  
`with Ada.Text_IO; use Ada.Text_IO;`
- Tipo Integer:  
`with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;`
- Tipo Float  
`with Ada.Float_Text_IO; use Ada.Float_Text_IO; -- operaciones para reales`
- Para tipo boolean y cualquier otro tipo enumerado (ver pág. 31). Boolean es un tipo especial de la clase de los enumerados.  
`with Ada.Text_IO; -- dentro de Ada.Text_IO está definido enumeration_IO`  
  
`procedure ...`  
  
`package boolean_IO is new enumeration_IO(boolean); use boolean_IO;`  

Identificador cualquiera para designar el nuevo paquete I/O para booleanos



#### 4.1. Entrada y salida estándar

La entrada estándar en Ada es la proporcionada desde el teclado. La salida estándar se dirige al monitor(pantalla).

Las operaciones básicas son las siguientes:

Lectura	Escritura	Comprobar si hay más datos
<b>get</b> (entero)	<b>put</b> (entero) <b>put</b> (entero, n° dígitos)	<b>end_of_file</b>
<b>get</b> (real)	<b>put</b> (real) <b>put</b> (real, n° dígitos_parte_entera, n° dígitos_parte_decimal, exponencial_E)	
<b>get</b> (caracter)	<b>put</b> (caracter)	
<b>get</b> (un_string) <b>get_line</b> (un_string, natural) -- lee un string -- de tamaño del string o hasta salto de línea	<b>put</b> (un_string) <b>put_line</b> (un_string) -- escribe y -- salta línea	

Se pueden leer/escribir en/desde variables de tipo enumerado. Igual que desde teclado/sobre pantalla:

```
with Ada.Text_IO; use Ada.Text_IO;
```

```
procedure program1 is
```

```
  type t_color is (rojo, azul, amarillo);
```

```
  package color_ES is new Enumeration_IO(t_color);
```

```
  use color_ES;
```

```
  C: t_color;
```

```
begin
```

```
  get(C);  -- valor de tipo enumerado
```

```
  put(C);  -- valor de tipo enumerado
```

```
...
```

El tipo **t\_color** es enumerado

```
with Ada.Text_IO; use Ada.Text_IO;
```

```
procedure programa2 is
```

```
  package boolean_ES is new Enumeration_IO(boolean);
```

```
  use boolean_ES;
```

```
  Es_Cierto: boolean;
```

```
begin
```

```
  get(Es_Cierto);  -- tipo boolean
```

```
  put(Es_Cierto);  -- tipo boolean
```

```
...
```

El tipo **boolean** es enumerado para E/S

Hay operaciones **get** y **put** diferentes para cada tipo básico de datos. Ada es capaz de distinguir (por el tipo del objeto) qué operación concreta ha de usar (igual que ocurre con las operaciones de relación: <, >, <=, >=, =, y /=, o con las operaciones aritméticas: +, -, \*, /). Este aspecto se denomina *sobrecarga* de operadores.

#### 4.1.1. Salida por pantalla

Tipo Character y String

**put**(character);

**put**(string);

**put\_line**(string);

Tipo Integer

**put** (integer);

**put** (integer; width);

Se escribe el número entero representado con una anchura de 11 espacios primer espacio para el signo y el resto para el entero justificado a la derecha o con una anchura de "width" caracteres. Si son necesarios menos caracteres que "width" para representar el entero, se rellena con blancos a la izda como en el caso anterior. Si no es suficiente con "width" caracteres, se usan los que haga falta.

put(4) -- escribe: "        4"

put(4, 0) --escribe: "4"



#### Tipo Float

**put**(float);

**put**(float, fore, aft, exp);

-- fore: natural nº dígitos antes de la coma- parte entera

-- aft: natural nº dígitos después de la coma- precisión decimal

-- exp: natural para expresar el nº de dígitos del exponente  $10^{**exp}$

Se escribe el float representado como un exponencial con "fore" dígitos antes de la coma, "aft" dígitos tras la coma y "exp" dígitos para el exponente. Si "exp" vale 0 no se escribe exponente. Si vale mayor que cero se escribe siempre con "exp" dígitos de exponente. Si el número de dígitos necesario es mayor que el espacio indicado por "fore" se usarán los caracteres necesarios. Si es menor se rellena con espacios a la izda. El signo negativo cuenta como un espacio más

Suponer que P=-12.34; Q=0.00567;

**put**(P); --escribe: -1.23400E+01

**put**(p, 0, 2, 0); --escribe: -12.34

**put**(Q); --escribe: 5.67000E-03

**put**(q, 0, 3, 0); --escribe: 0.006

-- Lo que más usaremos sera reales sin exponencial y con una precision de 2 o 3 decimales.

#### 4.1.2. Entrada desde teclado

Los caracteres de fin de línea y fin de página no pueden ser leídos por los procedimientos **get**, en su lugar, se saltan.

##### Tipo Character y String

**get** (character);

**get** (string); // el número de caracteres que se lean dependerá del nº de caracteres definidos para el string. Si el string está definido para 10 caracteres, entonces tendrá que leer exactamente 10 caracteres.

**get\_line** (string; last);

El string contiene los caracteres de la entrada y "last" indicará cuántos caracteres se han leído.

Si no hay tantos caracteres en la línea como la longitud de item, se devuelve un substring con tantos caracteres como haya en la línea y las restantes posiciones del string no se modifican. Si se llega a un final de línea, se salta al principio de la siguiente línea

##### Tipo Integer/Float

**get** (num);

Operación de fin de datos : **end\_of\_file** devuelve valor true/false;

## 4.2. Entrada y salida NO estándar- ficheros de texto

- **Ficheros de entrada**, la organización de datos y lectura es semejante al *teclado*.
- **Ficheros de salida**, el proceso de escritura es semejante al de la *pantalla*
- Los ficheros se pueden **crear** con el **editor** (herramienta con la que escribimos los programas, como AdaGIDE, Bloc de Notas, Emacs, ... )
- Se pueden **visualizar** con el **editor**.
- Pueden contener valores de uno o varios de los siguientes tipos:  
*Character*, *String*, *numéricos* (*Integer* y cualquier subtipo integer, y *Float*) y Enumerados (*Boolean* y cualquier otro definido por el usuario).
- Además pueden contener 2 tipos caracteres especiales:
  - ✓ fin de línea (ninguno, uno o varios)
  - ✓ fin de fichero (siempre uno)

Para poder realizar una operación de lectura desde un fichero o escritura hacia un fichero son necesarios varios pasos:

- inclusión de las cláusulas de contexto adecuadas, las mismas cláusulas de contexto que para leer de teclado y escribir en pantalla, dependiendo de los tipos de datos contenidos en el fichero.

- declaración de una variable tipo fichero que te permite manipular ficheros  
F: file\_type;
- apertura del fichero para poder hacer las operaciones de lectura/escritura  
**open**(F, [in\_file/out\_file], string-nombre del fichero); -- **in\_file** fichero para lectura  
-- **out\_file** fichero para escritura
- Las operaciones básicas son las siguientes:

Lectura ( <b>get</b> )	Escritura ( <b>put</b> )	Comprobar si hay más datos
<b>get</b> (F,entero)	<b>put</b> (F,entero) <b>put</b> (F,entero, n°_dígitos)	<b>end_of_file</b> (F)
<b>get</b> (F,real)	<b>put</b> (F,real) <b>put</b> (F, real, n°_dígitos_parte_entera, n°_dígitos_parte_decimal, exponencial_E)	
<b>get</b> (F,caracter)	<b>put</b> (F,caracter)	
<b>get</b> (F,un_string) <b>get_line</b> (F, un_string, natural)	<b>put</b> (F,un_string) <b>put_line</b> (F,un_string)	

#### 4.2.1. Lectura desde ficheros de texto

*Variable\_file*: **file\_type**; --Es necesario definir una variable del tipo *file\_type*

**Open**(*Variable\_file*, **In\_file**, *String*);

- *Variable\_file* es de tipo **file\_type**
- **In\_file** significa que es un fichero de “entrada” (se van a leer sus datos)
- *String* es el nombre externo del fichero. El fichero debe existir con ese nombre.
- **Open** abre una conexión entre el programa y el fichero a través de *Variable\_file*.
- Tras ejecutar **Open**, el primer dato del fichero *String* está disponible para su lectura.

**Get**(*Variable\_file*, *Var*); o **Get\_line**(*Variable\_file*, *String\_Var*, *Var\_long*);

- *Variable\_file* es de tipo **file\_type**.
- Se ha tenido que ejecutar antes **Open** con *Variable\_type*
- *Var* es una variable donde se va a leer el siguiente valor del fichero ligado a *Variable\_file*

**Skip\_line**(*Variable\_file*); -- Saltar para leer desde la siguiente línea

**End\_of\_File**(*Variable\_file*); -- cierto si no quedan más datos que leer

**End\_of\_Line**(*Variable\_file*);-- cierto si no quedan datos por leer en la línea actual

**Close**(*Variable\_file*);

#### 4.2.2. Escritura sobre ficheros de texto

*Variable\_file*: **file\_type**;

- Es necesario definir una variable del tipo predefinido *file\_type*

**Create**(*Variable\_file*, **Out\_file**, *String*); -- crea un nuevo fichero

- *Variable\_file* es de tipo **file\_type**
- **Out\_file** significa que es un fichero de “salida” (se van a escribir sobre él datos)
- *String* es el nombre del fichero en el sistema
- No debe existir un fichero de nombre *String*
- **Create** crea un fichero de nombre *String* y establece una conexión entre el programa y el fichero a través de *Variable\_file*

**Open**(*Variable\_file*, **opción**, *String*);

- Es como **Create**, pero debe existir un fichero de nombre *String*
- **Opción**:
  - **Out\_file** Borra el contenido del fichero *String* y establece una conexión entre el programa y el fichero a través de *Variable\_file*
  - **Append\_file**, No borra el contenido del fichero *String* y las escrituras se realizan a partir del último dato del fichero

**Put**(*Variable\_file*, *Expresión*); -- escribe resultado de expresión donde este el cursor

- *Variable\_file* es de tipo **file\_type**
- Se ha tenido que ejecutar antes **Create** con *Variable\_type* o bien **Open-Out\_file**
- El valor resultante de evaluar la *Expresión* se escribe en el fichero ligado a *Variable\_file*

**Put\_line**(*Variable\_file*, *Valor\_String*); -- escribe string y salta de línea

**Close**(*Variable\_file*);

- Es preciso que exista una conexión entre el programa y un fichero a través de *Variable\_file* (hecha con **Create** o **Open**)
- Cierra dicha conexión entre programa y fichero
- Tras ejecutar **Close** ya no se pueden realizar más escrituras usando *Variable\_file*

Se pueden usar todas las instrucciones de escritura utilizadas para la salida estándar, añadiendo la variable *file\_type* como primer parámetro:

**New\_line**(*Variable\_file*); -- Pone el cursor en la siguiente línea

## Errores típicos de ejecución.

Durante el uso de ficheros se pueden producir los siguientes errores:

- *Status\_Error*: cuando se intenta utilizar un fichero que no ha sido abierto (con **Open** o **Create**), o bien cuando se intenta abrir un fichero que ya estaba abierto
- *Mode\_Error*: cuando se intenta hacer una operación inconsistente con el estado del fichero (p. ej., escribir en un fichero abierto para lectura)
- *Name\_Error*: error producido por **Open** o **Create** cuando el nombre del fichero es incorrecto (p. ej., cuando se quiere abrir para lectura un fichero que no existe)
- *End\_Error*: producido cuando se intenta leer más allá del final del fichero
- *Data\_Error*: error producido al intentar leer un valor de cierto tipo y encontrar a la entrada un valor de distinto tipo (p. ej., al intentar leer un entero y encontrarse "abc" a la entrada)

## 5. Estructuras de Control

### 5.1. Estructuras condicionales: if y case.

```
Sentencia_if ::=  
    if Condicion then  
        Sentencias  
    { elsif Condicion then  
        Sentencias }  
    [ else  
        Sentencias ]  
    end if;
```

```
comprar: boolean:=False;  
precio_unidad, cant: integer; --declaraciones
```

```
if cant > 2000 then  
    if precio_unidad <= 50000  
        then comprarlo := True;  
    else put("precio muy caro"); new_line;  
    end if;  
else put("cantidad insuficiente");  
end if; -- Estructuras "if" anidadas
```

```
edad: integer;  
coeficiente: float; --declaraciones  
...  
if edad > 60 then coeficiente := 0.6;  
    elsif edad > 40 then coeficiente := 0.75;  
    elsif edad > 30 then coeficiente := 1.0;  
    else coeficiente := 0.6; -- edad <= 30  
end if;  
put("Coeficiente asignado es MIBOR +");  
put(coeficiente);  
new_line;
```

### Estructura "case": selección múltiple

```
Sentencia_Case ::=  
    case Expresion_selector is  
        when Alternativa =>  
            Sentencias  
    { when Alternativa =>  
        Sentencias }  
    [ when others =>  
        Sentencias ]  
    end case;  
Alternativa ::= Valores { | Valores}  
Valores ::= Literal | Constante |  
    Rango_valores | Expresión_sin_variables
```

```
operador: character; op1, op2: integer;  
--declaraciones
```

```
...  
get(op1); get(operador); get(op2);  
case operador is  
    when '+' => put(op1 + op2);  
    when '-' => put(op1 - op2);  
    when '*' => put(op1 * op2);  
    when '/' => put(op1 / op2);  
end case;
```

La expresión (selector) debe corresponder a un tipo discreto (por ejemplo, no puede ser Float).

## Estructura "case": ejemplos

```
N_letras, N_digitos, N_simbolos: natural:= 0; c: character;  --declaraciones
```

```
...
```

```
get(c);
```

```
case c is
```

```
  when 'a'..'z' | 'A'..'Z' =>      N_letras:= N_letras + 1;
```

```
  when '0'..'9' =>      N_digitos:= N_digitos + 1;
```

```
  when others =>      N_simbolos:= N_simbolos + 1;
```

```
end case;
```

Una estructura **if-elsif...-else** no siempre se puede sustituir por **case** ya que la expresión del **case** (selector) debe ser de tipo discreto.

## 5.2. Estructuras iterativas: *while*, *loop* y *for*

### "while" ("mientras")

```
Sentencia_while ::=
  while Condicion loop
    Sentencias_del_while
  end loop;
```

```
Sentencias_del_while ::=
  Sentencia_del_Programa
```

Primero se evalúa la condición. Si es falsa el **while** acaba. Si es cierta, se ejecutan las secuencias del **while** y se vuelve a evaluar la condición. Este proceso se repite hasta que la condición (de continuación) se evalúe a falso.

Ejemplo: Contar cuántos negativos hay en la entrada antes de la aparición del primer 0.

```
Numero ← 1
Contador ← 0
mientras numero /= 0 y
    hay números por leer
    leer Número
    Si Número < 0
        Contador ← Contador + 1
Fin
```

```
numero, contador: integer; --declaraciones
...
numero := 1 -- valor distinto de cero
contador := 0
while numero /= 0 and then not end_of_file loop
    get(numero);
    if numero < 0 then
        contador:=contador + 1;
    end if;
end loop;
```

## “loop” (“repetir”)

```
Sentencia_loop ::=
    loop
        Sentencias_del_loop
    end loop

Sentencias_del_loop ::=
    Sentencia |
    exit when condicion;
```

Las sentencias del loop se ejecutan repetidas veces hasta que la condición de **exit when** sea cierta (condición de parada).

**exit when** pueden estar dentro de estructuras condicionales, siempre que estén dentro del ámbito de la estructura **loop**

### Ejemplo: implementación

Contar cuántos números de los introducidos por teclado son negativos:

Contador ← 0

**Repetir**

**Salir si** no hay números por leer  
leer Número

**Si** Número < 0

Contador ← Contador + 1

**Fin**

Contador, Numero: integer; --declaraciones

...

Contador:=0;

**loop**

**exit when** end\_of\_file;

get(Numero);

**if** Numero < 0 **then**

Contador:=Contador+1;

**end if;**

**end loop;**

Completa el programa anterior, para que además de escribir por pantalla el valor del contador, tenga la estructura completa de un programa Ada.

### "for" (" para cada")

Sentencia\_for ::=

**for** Variable\_for **in** [reverse] Rango **loop**

Sentencias\_del\_for

**end loop**

Rango ::= valor\_inic .. valor\_final      -- valor\_inic <= valor\_final

| tipo\_rango

Sentencias\_del\_for ::= Sentencias\_del\_Programa

La variable del for se autodeclara (si existía una con igual identificador, permanece intacta cuando acaba el for).

En la primera iteración la variable del **for** toma el primer valor del rango. Se ejecutan tantas iteraciones como valores haya en el rango. Cada nueva iteración la variable del for toma el siguiente valor del rango. La última iteración toma el último valor del rango.



Ejemplo. Escribir los números entre 1 y 100 divisibles por 2 y 5

**Para cada** Num entre 1 y 100

Si resto(Num,2)=0 and resto(Num,5)=0  
escribir Num

**Fin**

**for** n in 1..100 **loop**

if n rem 2 =0 and n rem 5 =0 then  
put(n);

end if;

**end loop;**

### "for" de rango decreciente

**for** i in reverse 1..100 **loop**

put(i); new\_line;

**end loop;**

## Bucles anidados

Ejemplo: Dada una serie de números a la entrada, escribir la tabla de multiplicar de cada número. Por ejemplo si los números leídos son el 1 y el 3 escribir

1 x 1 = 1                      3 x 1 = 3

1 x 2 = 2    y después    3 x 2 = 6

...

...

**Mientras** hay números por leer

Leer Número

**Para cada** Multiplicador entre 1 y 10

escribir Número " x " Multiplicador

escribir " = " Número \* Multiplicador

escribir en nueva línea

**Fin Para**

**Fin**

**while** not end\_of\_file **loop**

get(numero);

**for** Multiplicador in 1..10 **loop**

put(Número); put(" x ");

put(Multiplicador); put(" = ");

put(Número \* Multiplicador);

new\_line;

**end loop;**

**end loop;**

## 6. Subprogramas

Pueden ser funciones o procedimientos

### 6.1. Procedimientos y funciones

#### Procedimientos

```
Procedimiento_Ada ::= [Clausulas_contexto]
procedure Ident [ (parametros_formales_proc) ]
is
    [Declaraciones_locales]
begin
    Sentencias_del_procedimiento
end Ident;
parametros_formales_proc ::=
    id_parametros: modo_paso tipo
    {; id_parametros: modo_paso tipo}
id_parametros ::= id_param {, id_param}
id_parametros ::= identificador
```

```
Declaraciones_locales ::=
    Declaraciones
    ;
Sentencias_del_procedimiento ::=
    sentencias_de_programa
```

### Ejemplos

```
procedure intercambiar (n1, n2: in out integer) is
```

```
-- pos: el valor de n1 es el que tenía n2 a la entrada y viceversa
```

```
    aux: integer := n1;
```

```
begin
```

```
    n1:=n2; n2:=aux;
```

```
end intercambiar;
```

```
procedure caracter_sig (st: in string; c: in character; n: out natural; c_sig: out character) is
```

```
-- pre: el carácter "c" debe aparecer en el string y debe haber otro carácter "c_sig" en "st", después de "c".
```

```
-- pos: devuelve un número que indica la posición de la ocurrencia del carácter en el string. El carácter devuelto es
```

```
-- el que se encuentra en la posición siguiente a n
```

```
begin
```

```
    n:=1;
```

```
    while st(n) /= c loop
```

```
        n:= n+1;
```

```
    end loop;
```

```
    c_sig:=st(n+1);
```

```
end caracter_sig;
```

- Los parámetros in SÓLO se pueden consultar
- Los strings se pueden declarar sin un tamaño prefijado como parámetro formal

## Funciones en Ada

```
funcion_Ada ::=
  [Clausulas_contexto]
function Ident [ (parametros_formales_fun) ] return tipo is
  [Declaraciones_locales]
  begin
    Sentencias_de_la_funcion
  end Ident;
parametros_formales_fun ::=
  id_parametros: [in] tipo
  {; id_parametros: [in] tipo}

id_parametros ::= id_param {, id_param}
Declaraciones_locales ::=
  decl_tipos | decl_constantes |
  decl_variables | decl_subprogramas |
  otras_declaraciones
Sentencias_de_la_funcion ::= sentencia_Ada | return valor;
valor ::= expresion
```

La expresión devuelta con **return** debe ser del mismo tipo que el definido en el **return** de la cabecera

### ejemplos

```
function max (num1, num2: integer) return integer;
-- pos: devuelve el mayor de los números de entrada
begin
  if num1 > num2 then    return num1;
  else return num2;
  end if;
end if;

function buscar_siguiente (st: string; c: character; n: natural) return natural is
-- pre: n<= long(st)
-- pos: devuelve un número que indica la posición del carácter c en el string a partir de la posición n.
-- inclusive. Si no hay ocurrencias del carácter se devuelve el número 0
  posicion: natural;
begin
  posición := n;
  while posicion < st'length and then st(posición) /= c loop
    posición:= posición +1;
  end loop;
  if st(posicion)=c then    return posicion; else    return 0;
  end if;
end buscar_siguiente;
```

## 6.2. Subprogramas y cláusulas de contexto

Un modo de organizar los subprogramas en Ada consiste en ubicar cada subprograma en un fichero y compilar éstos separadamente.

De la compilación sólo puede obtenerse un ejecutable si se trata de un procedimiento sin parámetros. Y en otro caso, se puede utilizar el subprograma en otros subprogramas.

Para ello es preciso incorporar la correspondiente cláusula de contexto.

```
with intercambiar;  
procedure ordenar (n1,n2: in out integer) is  
  -- pos: el primer valor es el menor y el segundo el mayor  
begin  
  if n1 > n2 then  
    intercambiar(n1,n2);  
  end if;  
end ordenar;
```

Con "with intercambiar" decimos al programa o subprograma que utilice un subprograma de nombre intercambiar que ha sido declarado y compilado anteriormente.

### Notación posicional

Un posible uso del procedimiento anterior sería:

```
with ordenar;  
with Ada.Text_io, Ada.Integer_Text_io;  
use Ada.Text_io, Ada.Integer_Text_io;  
procedure pru is  
  -- pre: en la entrada estándar hay dos números enteros  
  -- pos: escribe los números de la entrada ordenados de menor a mayor  
  x,y: integer;  
begin  
  get(x); get(y);  
  ordenar(x,y); -- (*)  
  put(x); put(y); new_line;  
end pru;
```

Con notación posicional la línea --(\*) se escribe de cualquiera de estas dos formas:

```
ordenar(n1=>x, n2=>y);  
ordenar(n2=>y, n1=>x);
```

## 6.3. Sobrecarga

Varios procedimientos y/o funciones pueden tener el mismo identificador.

<pre>procedure intercambiar(n1,n2: in out integer) is -- pos: el valor de n1 es el que tenía n2 a la entrada y -- viceversa aux: integer:=n1; begin n1:=n2; n2:=n1; end intercambiar;</pre>	<pre>procedure intercambiar(n1,n2: in out character) is -- pos: el valor de n1 es el que tenía n2 a la entrada y --viceversa aux: character:=n1; begin n1:=n2; n2:=n1; end intercambiar;</pre>
---	--

En Ada se pueden definir subprogramas con el mismo nombre, pero diferente cabecera.

En estos casos se dice que el identificador está sobrecargado.

Esto pasa con subprogramas predefinidos que ya conocemos: put, get, +, -, \*, /, ...

## 7. Definición de tipos

### 7.1. Tipo Enumerado

Enumeración de elementos en un orden determinado

**type** id\_tipo **is** descripción

```
Declaracion_tipo_enumerado ::=
    type id_enumerado is (identificadores | caracteres);
caracteres ::= car_ASCII {, car_ASCII}
```

#### Ejemplos

```
type t_dias_semana is (lunes, martes, miercoles,jueves, viernes, sábado,domingo);
type t_colores is (rojo, azul, negro, verde);
type t_digitos_hex is ('0','1','2','3','4','5','6','7','8','9','A','B','C','D','E','F');
```

#### Atributos:

T'pred, T'succ, T'first, T'last, T'pos, T'val, T'image, T'value, T'min, T'max

```
t_dias_semana'pos(lunes) = 0
t_dias_semana'pos(domingo) = 6
```

No se admite, debe existir concordancia:  
col: t\_colores;  
col:= lunes; --error

## 7.2. Subtipos de rango

Subconjunto de elementos de un tipo discreto

```
Declaracion_subtipo_rango ::=  
    subtype id_subtipo is id_tipo range  
        primer_valor .. ultimo_valor
```

```
subtype t_rango_personas is integer range 1..10;
```

Cuando escribimos 1..10 estamos definiendo un rango que es un subtipo sin nombre.

```
subtype t_dias_laborables is t_dias_semana range lunes..viernes;
```

## 7.3. Registros: *record*

```
type t_estudiante is record  
    n_exped: natural;  
    nombre, apellido: string(1..30);  
    curso: natural;  
    grupo: character;  
end record;
```

```
Declaracion_tipo_registro ::=  
    type id_registro is  
        record  
            id_campos: tipo;  
            {id_campos: tipo;}  
        end record;  
    id_registro ::= Identificador  
    id_campos ::= Identificadores
```

Ejercicio: describir un tipo registro para contener los valores correspondientes a una fecha del calendario.

**Agregados de registro:** Sirven para dar valores a una variable de tipo registro:

```
est1: t_estudiante := (1150, "Iñaki  ", "Zuñiga  ", 1, D);
```

Usando identificadores de campos:

```
est2 := (curso=>2, n_exp=>1520, grupo=>'D'  
        apellido=>"Arregi  ", nombre=>"Luis  ");
```

## 7.4. Vectores y matrices: *array*

```
type t_rango_num is integer range 1..10;  
type t_numeros is array(t_rango_num) of integer;  
N: t_numeros;
```

```
type t_matriz is array(1..10, 1..20) of integer;  
A: t_matriz
```

```
Declaracion_tipo_array ::=  
    type ident_tipo is array (rangos) of tipo; |  
        Definicion_tipo_array_sin_restr  
  
    rangos ::= rango {,rango}  
  
    rango ::= primer_valor .. ultimo_valor |  
        subtipo_rango | subtipo_discreto
```

### Atributos:

N'first N'last N'range N'length  $\Leftrightarrow$  t\_numeros'first(N) t\_numeros'last(N) t\_numeros'range  
t\_numeros'length

A'first A'last A'range A'length  $\Leftrightarrow$  A'first(1), A'last(1), A'range(1), A'length(1)

A'first(N), A'last(N), A'length(N)

A'range(N), (En el ej. de la matriz si N=1 es 1..10 y N=2 es 1..20)

### Ejemplo

procedure **put** (v: in l\_numeros) is  
--pos: en la salida estándar están los  
-- números que contiene el vector “v”

**begin**

**for** i **in** rango\_num **loop**

put(v(i));

**end loop**;

**end put**;

function **esta**(v: l\_numeros; n: integer) return  
boolean is

--pos: cierto si n esta en l y falso si no está

i:natural:=v'first;

**begin**

**loop**

**exit when** i<v'last and then v(i)=n

i:= i+1;

**end loop**;

return v(i)=n;

**end esta**;

### Agregados Array

type l\_numeros is array(1..10) of integer;

-- Inicialización

numeros: l\_numeros:= (0,1,0,1,0,1,0,1,0,1);

	1	2	3	4	5	6	7	8	9	10
numeros	0	1	0	1	0	1	0	1	0	1

otros\_num: l\_numeros:=(1=>1, 2=>1, 3=>1, others=>0);

	1	2	3	4	5	6	7	8	9	10
otros_num										

numeros:=(others=>0);

	1	2	3	4	5	6	7	8	9	10
numeros										

type t\_matriz is array(1..3, 1..4) of integer;

matriz: t\_matriz:=((0,0,0,0),(1,1,1,1),(2,2,2,2));

```

subtype t_rango_estud is integer range 1..3;
type t_estudiantes is array (t_rango_estud) of t_estudiante;

estudiantes: t_estudiantes:=
  ((1320,"Juan      ", "Zubia      ",1,D),
   (1789,"Cesar     ", "Albistegui ",2, B),
   (1810,"Carmen    ", "Etxebeste  ",5,A)) ;

```

### Arrays sin restricciones (*unconstrained*)

```

type l_numeros is array(integer range<>) of integer;

```

Para definir parámetros de subprogramas

No es posible definir variables de estos tipos

```

Declaracion_tipo_array_sin_restr ::=
type ident_tipo is
    array (rangos_sin_restr) of tipo;

rangos_sin_restr ::=
    rango_sin_restr {,rango_sin_restr}

rango_sin_restr ::= tipo_discreto range <>

```

El tipo string está predefinido de la siguiente forma:

```

type string is array (positive range<>) of character;

```

Se pueden definir variables string indicando el rango:      nombre: string (1..30);

Lo mismo ocurre con el resto de tipos array sin restricciones:      numeros: t\_numeros(1..10);

- El rango de valores se concreta en el paso de parámetros:

(1) Definición del tipo sin restricciones

```

type t_enteros is array(character range<>) of integer;

```

(2) Descripción de parámetro. Declaración: rango desconocido

```

function esta(l: t_enteros; n: integer) return boolean;

```

(3) Descripción de variable restringiendo el rango

```

lis: t_enteros('A'..'K');

```

(4) Utilización: rango conocido

```

esta(lis);

```