Tema 3. PROGRAMACION EN ADA

3.1 INTRODUCCION

PROGRAMA EJEMPLO

```
with Text_IO; -- especificación de contexto
procedure CUENTA is
   NUMERO_DE_ITERACIONES: Integer := 1;    -- definición de una variable
   package Int_IO is new Text_IO.Integer_IO(Integer); -- package local
                                                      --para enteros
begin
   Text_IO.PUT("N° de iteraciones:");
   Int_IO.GET(NUMERO_DE_ITERACIONES);
   Text_IO.NEW_LINE;
   for INDICE in 1..NUMERO_DE_ITERACIONES loop
      Text_IO.PUT("Iteración ");
      Int_IO.PUT(INDICE,4);
      Text_IO.NEW_LINE;
   end loop;
end CUENTA;
```

Los comentarios se denotan por un guión doble (--) y terminan al final de la línea.

El programa principal siempre es un subprograma procedure sin argumentos. Un subprograma es una de las unidades de programa en Ada.

La especificación de contexto indica otras unidades de programa necesarias para la presente unidad. Este procedure usa el paquete Text_IO para soporte de operaciones I/O. Text_IO es un package (otro tipo de unidad) Ada que contiene muchas rutinas para I/O de caracteres, cadenas, enteros, etc.

NUMERO_DE_ITERACIONES es una variable de tipo Integer. El tipo predefinido Integer está definido en el package Standard, y este package se incluye automáticamente en todos los programas Ada, de modo que no necesita ser especificado en una cláusula with.

Int_IO es un package creado en este programa a partir de un package genérico llamado Integer_IO, el cual está contenido en Text_IO. Una unidad genérica es la tercera clase de unidad de programa en Ada.

3.1.1 UNIDADES DE PROGRAMA

Un fichero fuente en Ada contiene una o más unidades de programa. Un programa en Ada está compuesto por una o más unidades de programa. Una unidad de programa es un subprograma (procedure o function), un paquete (package), una tarea (task), una unidad genérica o una unidad protegida (Ada95). Cada unidad de programa se divide en dos partes: **especificación**, que define su interface con el exterior, y **cuerpo**, que contiene el código de la unidad de programa.

- Un subprograma expresa una acción secuencial.
- Un package es una colección de recursos computacionales relacionados. Puede incluir tipos de datos, declaración de variables, otras unidades de programa, etc. Son fundamentales en Ada.
- Una task es una acción a ejecutar en paralelo con otras tareas; base de la programación concurrente en Ada.
- Las unidades genéricas son componentes software reutilizables, que contienen definiciones de algoritmos independientes de los datos. Es una implementación especial de un subprograma o de un package.
- Los objetos protegidos disponen de datos y de operaciones de acceso a esos datos, y aseguran que dichas operaciones no actuarán sobre los datos concurrentemente.

Las unidades de programa pueden aparecer anidadas dentro de otras unidades. Algunas pueden ser compiladas por separado.

Unidades de compilación: unidades Ada que pueden ser compiladas separadamente. Una unidad de compilación puede ser la especificación de un package, la especificación de un subprograma, la especificación de una unidad genérica, y cada uno de los cuerpos correspondientes (las tareas y los objetos protegidos no son unidades de compilación). Cuando se compila un fichero fuente en Ada, el compilador coloca las unidades de compilación resultantes en una biblioteca (APUL, Ada Program Unit Library). Una vez en la APUL, una unidad puede ser utilizada por cualquier otra unidad que haga referencia a ella en una cláusula with. Al encontrar el with, el compilador busca en la APUL la información que describe la interfaz de la unidad referenciada, y chequea si la unidad que se está compilando utiliza dicho interface correctamente.

Todos los compiladores Ada proporcionan una APUL común, que contiene, entre otros, los packages Standard, Text_IO y System.

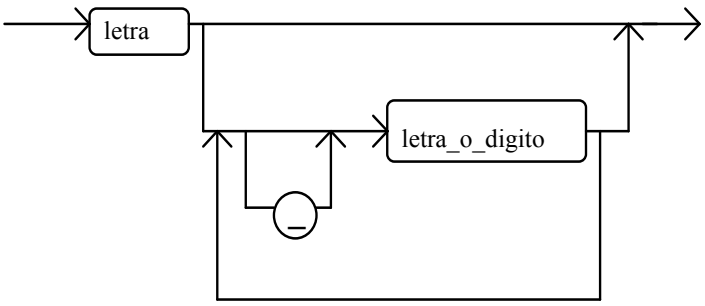
3.2 ESTILO LÉXICO

Un programa en Ada se escribe como una secuencia de líneas de texto que pueden contener los siguientes caracteres:

- Letras mayúsculas A..Z
- Letras minúsculas a..z
- Dígitos 0..9
- Caracteres especiales " ' ( ) * + , - . : ; < = > _ | &
- El carácter espacio

Las unidades léxicas del lenguaje se componen a partir del conjunto de caracteres. Consisten en: identificadores, literales numéricas, literales carácter, cadenas (strings), delimitadores y comentarios.

a) **Identificadores**



Ada no impone límite a la longitud de los identificadores definidos por el usuario, siempre que quepan en una línea. No hace distinción entre mayúsculas y minúsculas. Algunos identificadores son palabras reservadas (if, procedure, end,...).

b) **Literales numéricas**

Representan números reales o enteros (tipo anónimo universal integer). Pueden ser expresados en cualquier base, de 2 a 16. Por defecto, base 10.

Números enteros: sin punto decimal.

Números reales: con punto decimal, y al menos un dígito a cada lado del punto.

Si el número consta de muchos dígitos, para facilitar su lectura por el usuario puede ser dividido en grupos, insertando caracteres de subrayado, ignorados por el compilador.

Tanto enteros como reales pueden tener exponente. Este toma la forma E (o e) seguida por un entero con o sin signo. El exponente no puede ser negativo en el caso de un entero.

Ejemplos:

```
98.4 = 9.84E1 = 984.0e-1 = 0.984E+2
1900 = 19E2   = 190e1    = 1900E+0
3.1415926536 = 3.1415_9265_36
2#1101# = 1101_2 = 13
16#F.FF#E2 = 16#FFF# =  (15x16^0+15x16^-1+15x16^-2)x16^2 = 4095
```

Una literal numérica no puede ser negativa. Una forma tal como -3 consiste en una literal precedida por el operador unario menos.

c) **Literales carácter**

Uno de los caracteres ASCII encerrado entre apóstrofes. 'A', 'x'

d) **Cadenas**

Secuencias de cero o más caracteres encerrados entre comillas.

"esto es un string"

e) **Delimitadores**

Su función depende del contexto.

Simples: ' ( ) * + , . / : ; < = > & |

Compuestos: => .. ** := /= >= <= << >> <>

f) **Comentarios**

Comienzan con dos guiones (--) y terminan al final de línea.

-- esta línea es un comentario

3.3 TIPOS ESCALARES

Ada requiere que cualquier entidad sea declarada antes de ser usada. Estas entidades u objetos pueden ser variables (su valor puede cambiar durante la ejecución del programa) o constantes (conservan el valor inicial durante toda su existencia). Una declaración asocia al objeto con un tipo o subtipo dado.

Un tipo está caracterizado por:

- Un conjunto de valores.
- Un conjunto de operaciones aplicables a los objetos de ese tipo.

En Ada hay cuatro clases de tipos primitivos:

- Escalares: objetos con un solo componente.
- Compuestos: objetos que pueden tener más de un componente.
- Access: proporcionan acceso (referencian) a otros objetos.

- Private: no conocidos para el usuario.

Los tipos escalares se dividen a su vez en tipos enteros, reales y enumerados. Tipos enteros y enumerados pueden ser también llamados tipos discretos.

3.3.1 Declaraciones y asignaciones

Declaración de variables:

IDENTIFICADOR {,IDENTIFICADOR} : TIPO [:= VALOR_INICIAL];

```
I,J : INTEGER;

K,L : INTEGER := 10;
```

Asignación de valores a variables:

IDENTIFICADOR := EXPRESION_SIMPLE;

```
I:=36;

J:=I+K;
```

Declaración de constantes:

IDENTIFICADOR : **constant** [TIPO] := VALOR;

Si el tipo de valor es numérico, es posible omitirlo en la declaración de la constante.

```
PI: constant := 3.1415926;
```

Si no se omite el tipo, el valor puede ser cualquier expresión, y se evalúa en la ejecución.

Ej: `M: constant INTEGER:= I*J;`

Si se omite el tipo, el valor ha de ser una expresión estática, y se evalúa en tiempo de compilación.

3.3.2 Bloques

Un bloque en Ada consiste en:

1. Declaración de objetos locales.
2. Secuencia de sentencias.
3. Colección de excepciones.

declare

<parte declarativa>

begin

<sentencias>

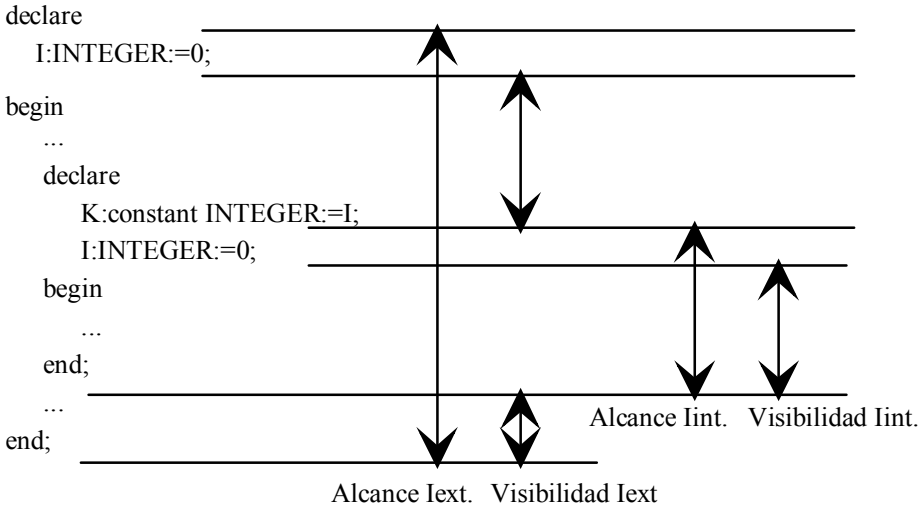
[**exception**

<excepciones>]

end;

Un bloque puede estar situado, dentro de un programa, en cualquier lugar donde iría una sentencia. Un bloque es en sí mismo una sentencia, y así, cualquier sentencia del bloque puede ser otro bloque.

La parte declarativa hace que los objetos declarados empiecen a existir (almacenamiento temporal). Cuando se llega al *end* del bloque, todos los objetos declarados dejan de existir, y el espacio que ocupaban en la pila de memoria dedicada al almacenamiento de variables locales queda libre.



3.3.3 Tipos numéricos (enteros y reales)

El tipo INTEGER está predefinido (incluido en el *package* STANDARD). Su rango depende de la implementación; en una máquina de 16 bits, el rango irá desde -32768 hasta 32767, y puede ser diferente del rango implementado para una máquina de 32 bits. Algunas implementaciones de Ada contienen los tipos LONG_INTEGER y SHORT_INTEGER.

En cuanto a tipos reales, está predefinido el tipo FLOAT, dependiente también de la máquina.

3.3.3.1 Subtipos del tipo INTEGER

Declaración:

```
subtype NOMBRE is INTEGER range EXP_ENTERA..EXP_ENTERA;
```

El rango no necesita ser estático.

```
subtype DIAS is INTEGER range 1..31;   -- subtipo explícito.
D:DIAS;
X: range 1..31;INTEGER
```

Subtipos predefinidos: NATURAL, POSITIVE

```
subtype NATURAL is INTEGER range 0..INTEGER'LAST;
subtype POSITIVE is INTEGER range 1..INTEGER'LAST;
I:INTEGER;
N:NATURAL;
...
I:=N;   --correcto. Un subtipo no es un tipo distinto.
N:=I;   --correcto. Puede dar CONSTRAINT_ERROR en la ejecución si I<0
```

3.3.3.2 Operadores

+, - unarios	signo
+, - binarios	suma, resta. Los dos operandos han de ser del mismo tipo. El resultado, del tipo de los operandos.
*, /	multiplicación, división. Los dos operandos del mismo tipo. El resultado, del tipo de los operandos. La división entera trunca a cero.
rem	resto de la división entera. Trunca a cero.
mod	módulo. Ambos operandos enteros. Trunca a menos infinito. 7 rem 3 = 1 7 mod 3 = 1 7 rem (-3) = 1 7 mod (-3) = -2 (-7) rem 3 = -1 (-7) mod 3 = 2 (-7) rem (-3) = -1 (-7) mod (-3) = -1
abs	Valor absoluto. El resultado, del tipo del operando.
**	Exponenciación. Si el primer operando es entero, el segundo ha de ser natural. Si el primer operando es real, el segundo puede ser cualquier entero. El resultado, del tipo del primer operando.

Además, se pueden efectuar las operaciones de comparación

= /= < <= > >= (operadores relacionales)

entre dos operandos del mismo tipo. El resultado, booleano.

3.3.4 Tipos enumerados

El programador declara explícitamente el conjunto de valores de un tipo (conjunto ordenado):

```
type NOMBRE_DIAS is (LUN,MAR,MIE,JUE,VIE,SAB,DOM);
type COLOR is (ROJO, AMARILLO, VERDE);
```

subtipos:

```
subtype LABORABLES is NOMBRE_DIAS range LUN..VIE;
```

Tipos enumerados predefinidos:

```
type BOOLEAN is (FALSE,TRUE);
type CHARACTER is ... --caracteres ASCII
```

3.3.4.1 Operadores booleanos: not, and, or, xor, and then, or else

A	B	not A	A and B	A or B	A xor B
F	F	T	F	F	T
F	T	T	F	T	T
T	F	F	F	T	T
T	T	F	T	T	F

Tienen menor precedencia que los operadores relacionales. *and*, *or* y *xor* tienen igual precedencia, menor que *not*. Si se usan mezclados en una expresión, es necesario el uso de paréntesis:

```
D:=A and B or C;   -- ilegal
D:=(A and B) or C;
```

Con *and* y *or*, los operandos se evalúan sin ser especificado el orden. Con *and then* y *or else*, se evalúa el primer operando. El operando de la derecha sólo se evalúa si es necesario.

A and then B Si A es FALSE, B no se evalúa (FALSE)

A or else B Si A es TRUE, B no se evalúa (TRUE)

Prioridad (de mayor a menor):

**	abs	not			
*	/	mod	rem		
+	-	unarios			
&	+	-	binarios		
=	/=	<	<=	>	>=
in	not in				
and	or	xor			
and then			or else		

3.3.5 Atributos

```
type NOMBRE_DIAS is (LUN,MAR,MIE,JUE,VIE,SAB,DOM);
NOMBRE_DIAS'FIRST       = LUN
NATURAL'FIRST               = 0
NOMBRE_DIAS'LAST        = DOM
NOMBRE_DIAS'SUCC(MAR)   = MIE
NOMBRE_DIAS'PRED(VIE)   = JUE
NOMBRE_DIAS'POS(DOM)    = 6
NOMBRE_DIAS'VAL(0)      = LUN
POSITIVE'POS(1)             = 0
NOMBRE_DIAS'IMAGE(LUN)  ="LUN"
NOMBRE_DIAS'VALUE("LUN")    = LUN
```

3.3.6 Tipos derivados

Un tipo derivado es un nuevo tipo, similar a uno ya existente. Hereda las operaciones aplicables al padre.

```
type MANZANAS is new INTEGER range 0..INTEGER'LAST;
type PERAS is new INTEGER range 0..INTEGER'LAST;
M:MANZANAS;
P:PERAS;
I:INTEGER;
R:REAL;
...
M:=P;          --ILEGAL. No se pueden mezclar tipos distintos.
M:=MANZANAS(P);    --LEGAL. Conversión de tipos.
I:=INTEGER(R);
```

El programador puede (se aconseja) definir sus propios tipos enteros y reales, para conseguir una mayor portabilidad en sus programas:

```
type ENTERO is range -1E6..1E6;
```

El tipo base del que deriva puede ser INTEGER en una máquina, y LONG_INTEGER en otra máquina. El compilador se encargará de seleccionar la representación subyacente adecuada. El rango debe ser estático, puesto que la elección del tipo base la efectúa el compilador.

```
type REAL is digits 7;  --nº de dígitos significativos
type MASS is digits 6 range 0.0..3.0;
```

Al igual que con los enteros, el compilador se encargará de elegir el tipo base real adecuado (FLOAT, LONG_FLOAT, SHORT_FLOAT).

3.4 ESTRUCTURAS DE CONTROL

3.4.1 Sentencias if

```
if EXP_BOOLEANA then
     ...
end if;
```

```
if EXP_BOOLEANA then
     ...
else
     ...
end if;
```

```
if EXP_BOOLEANA then
     ...
else
     if EXP_BOOLEANA2 then
          ...
     else
          ...
     end if;
end if;
```

≡

```
if EXP_BOOLEANA then
     ...
elsif EXP_BOOLEANA2 then
          ...
else
     ...
end if;
```

3.4.2 Sentencia case

```
case EXP_DISCRETA is
     SENTENCIA_ALTERNATIVA
   { SENTENCIA_ALTERNATIVA }
end case;
SENTENCIA_ALTERNATIVA:
when ELECCION {|ELECCION} => SECUENCIA_DE_SENTENCIAS
ELECCION:
EXP_SIMPLE | RANGO_DISCRETO | others
RANGO_DISCRETO:
TIPO_O_SUBTIPO_DISCRETO | RANGO
TIPO_O_SUBTIPO_DISCRETO:
NOMBRE [range RANGO]
RANGO:
EXP_SIMPLE..EXP_SIMPLE
```

Ejemplo. Suponiendo los siguientes tipos:

```
type DIAS is (LUN,MAR,MIE,JUE,VI,SAB,DOM);
subtype LABORABLES is DIAS range LUN..VIE;
```

las siguientes SENTENCIAS_ALTERNATIVAS serán válidas (no simultáneamente):

```
when LABORABLES =>   ...   -- subtipo discreto
```

```
when LUN..MIE =>     ...   -- rango
when LUN|JUE|SAB =>  ...   -- exp. simples
when LUN..MIE|DOM => ...   -- rango³exp. simple
```

- Cada valor posible de la expresión que sucede a **case** debe estar cubierto una (y sólo una) vez en las sentencias alternativas.
- Todos los valores y rangos que suceden a **when** deben ser estáticos.
- Si se utiliza **others**, debe ser la última sentencia alternativa.

Ejemplo: Escribir un bloque que calcule qué día es mañana, dadas las siguientes declaraciones, y el día de hoy.

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Maniana is
  type NOMBRE_MES is (ENE,FEB,MAR,ABR,MAY,JUN,JUL,AGO,SEP,OCT,NOV,DIC);
  N_MES : NOMBRE_MES;
  DIA : INTEGER range 1..31;
  MES : INTEGER range 1..12;
  ANIO : INTEGER range 1901..2099;
  package IIO is new Integer_IO(INTEGER); use IIO;

begin
   put("dia (1..31):");
   get(DIA);
   put_line(" ");
   put("mes (1..12):");
   get(MES);
   N_MES:=NOMBRE_MES'VAL(MES-1);
   put_line(" ");
   put("año:");
   get(ANIO);
   put_line(" ");

   declare
      DIAS_MES : INTEGER range 1..31;
   begin
      case N_MES is
         when ABR|JUN|SEP|NOV => DIAS_MES:=30;
         when FEB=> if (ANIO rem 400 = 0) or else ((ANIO rem 4=0)
                    and (ANIO rem 100 /= 0)) then
                        DIAS_MES:=29;
                    else
                        DIAS_MES:=28;
                    end if;
         when others => DIAS_MES:=31;
      end case;
      if DIA>DIAS_MES then
         put_line(NOMBRE_MES'IMAGE(N_MES) & " tiene " &
                  INTEGER'IMAGE(DIAS_MES) & " dias");
         raise constraint_error;
      elsif DIA<DIAS_MES then
         DIA:=DIA+1;
      else
         DIA:=1;
         if N_MES /= DIC then
            N_MES:=NOMBRE_MES'SUCC(N_MES);
         else
            N_MES:=ENE;
            ANIO:=ANIO+1;
         end if;
```

```
         end if;
         put_line("MAÑANA ES...");
         put(DIA);
         put(" " & NOMBRE_MES'IMAGE(N_MES) & " ");
         put(ANIO);
         put_line(" ");
      end;
   end Maniana;
```

3.4.3 Bucles

```
loop          -- bucle infinito
   ...
end loop;
```

```
loop                                loop
   ...                                 ...
   if EXP_BOOLEANA then      ≡         exit when EXP_BOOLEANA;
      exit;                            ...
   end if;                          end loop;
   ...
end loop;
```

```
while EXP_BOOLEANA loop
     ...
end loop;
```

```
for VARIABLE_DISCRETA in RANGO_DISCRETO loop
     ...
end loop;
```

- La variable discreta no tiene que ser declarada previamente. Se declara implícitamente, del tipo del rango, y deja de existir al salir del bucle.
- El rango no puede ser cambiado dentro del bucle.
- Si los límites del rango son literales enteras, se asume tipo INTEGER.

Ejemplos:

```
for I in 1..10 loop
   ...
end loop;

for D in DIAS loop
   ...
end loop;

for M in ENE..JUN loop
   ...
end loop;

   for i in 1 .. 3 loop
      for i in 5 .. 7 loop
         put(i);
      end loop;
      new_line;
      put(i);
      new_line;
   end loop;
```

Salida de bucles anidados:

```
BUCLE:
for I in 1..N loop
     for J in 1..M loop
          ...
          if EXP_BOOLEANA then
               ...
               exit BUCLE;
          end if;
     end loop;
end loop BUCLE;
```

Bucles en orden descendente:

```
for I in reverse 1..10 loop
     ...
end loop;

for I in -1..10 loop  -- ILEGAL: -1 no es una literal numérica
```

Solución:

MENOS_UNO: constant:=-1;

for I in MENOS_UNO..10 loop

3.5 TIPOS COMPUESTOS

- Describen objetos que constan de varios componentes.
- Tipos array: colección de componentes homogéneos.
- Tipos record: colección de componentes heterogéneos.

3.5.1 ARRAYS

Un array consiste en un grupo de componentes, todos del mismo tipo. Un componente particular del array se selecciona dando el nombre del array y uno o más índices, de valores discretos. El número de índices indica la dimensión del array. Ada no impone límite al número de índices.

Definición de tipo array:

```
type NOMBRE_DEL_ARRAY is array (TIPO_INDICE {, TIPO_INDICE}) of
                                                    TIPO_COMPONENTES;
```

```
Ejemplos:
type MATRIZ_ENTERA is array(1..3,1..8) of INTEGER;
subtype INDEX is INTEGER range 1..10;
type VECTOR is array(INDEX) of INTEGER;
V:VECTOR;
M:MATRIZ_ENTERA;
HORAS_TRABAJO: array(DIAS) of REAL;
...
for I in 1..3 loop          ≡         M:=((0,0,0,0,0,0,0,0),
     for J in 1..8 loop                    (0,0,0,0,0,0,0,0),
          M(I,J):=0;                       (0,0,0,0,0,0,0,0));
     end loop;
end loop;
for I in INDEX loop         ≡         V:=(1,1,1,1,1,1,1,1,1,1);
     V(I):=1;
end loop;
```

```
for D in LABORABLES loop
     HORAS_TRABAJO(D):=8.0;
end loop;
HORAS_TRABAJO(SAB):=0.0;
HORAS_TRABAJO(DOM):=0.0;
```

Otras asignaciones válidas:

```
V:=(1..10 => 1);
V:=(INDEX => 1);
M:=(1..3 => (1..8 => 0));
HORAS_TRABAJO:=(LUN..VIE =>8.0, SAB|DOM => 0.0);

-Atributos relativos a los índices:
V'FIRST = 1
V'LAST  = 10
V'LENGTH = 10
V'RANGE = 1..10
M'FIRST(1) = 1
M'LAST(1) = 3
M'LAST(2) = 8
M'RANGE(2) = 1..8
```

En los arrays vistos hasta ahora, los límites de los índices fueron fijados en la declaración. Ada permite también la declaración de tipos array con rango de índices sin definir hasta la declaración de los objetos array particulares. Así, podremos tener dos objetos array del mismo tipo y con diferentes límites en los índices.

```
type VECTOR is array(INTEGER range <>) of BOOLEAN;
V1: VECTOR(1..5);
V2: VECTOR(1..10):=(1..10 => FALSE);
N:INTEGER;
...

declare
   V3:VECTOR(1..N); --rango no estático. Si N<1, sin componentes
```

Se puede hacer una asignación de un array a otro array, o una comparación igualdad/desigualdad, si ambos tienen igual número de dimensiones e igual número de componentes en cada dimensión. Lo siguiente será válido:

```
U:VECTOR(1..4);
V:VECTOR(8..11);
...
U:=V;
if U/=V then ...
```

3.5.2 STRINGS

En el package STANDARD, Ada incluye un tipo array predefinido, llamado STRING, que es un array monodimensional cuyos componentes son de tipo CHARACTER:

```
subtype POSITIVE is INTEGER range 1..INTEGER'LAST;
type STRING is array(POSITIVE range<>) of CHARACTER;
```

Ejemplos de declaraciones y asignaciones:

```
NAME:STRING(1..10);
G:constant STRING:="HOLA";   ≡   G:constant STRING:=('H','O','L','A');
```

En este caso, la longitud del string G viene determinada por su valor inicial.

```
NAME:="JOSE" & " PEREZ";  -- NAME = "JOSE PEREZ";
NAME(1..4):="PEPE";             -- NAME = "PEPE PEREZ";
```

3.5.3 RECORDS

Un registro (*record*) es un objeto compuesto que consta de componentes con nombre, los cuales pueden ser de diferentes tipos:

```
type FECHA is
   record
     DIA:INTEGER range 1..31;
     MES:NOMBRE_MES;
     ANIO:POSITIVE;
   end record;
D,E,F : FECHA;
...
D:=(3,ABR,1976);   -- asignación posicional
E.DIA:=4;
E.MES:=MAR;
E.ANIO:=1980;
F:=(MES=>JUL, DIA=>16, ANIO=>1994);  -- asignación con nombre
```

En una asignación con nombre, no es necesario guardar el orden dado en la declaración.

Es posible dar valores por defecto a algunos o a todos los componentes en la declaración del tipo. Así:

```
type INTEGER_VECTOR is array(INTEGER range <>) of INTEGER;
MAX:constant:=100;
type STACK is
   record
     S:INTEGER_VECTOR(1..MAX):=(1..MAX => 0);
     TOP:INTEGER range 0..MAX:=0;
   end record;
```

- Registros variantes

```
type CLASE_SENSOR is (HUMEDAD, PRESION, TEMPERATURA);
type SENSOR(TIPO:CLASE_SENSOR) is
   record
     SITUACION:STRING(1..20);
     case TIPO is
        when HUMEDAD => HUM : NATURAL;
        when PRESION => PRES: REAL;
        when TEMPERATURA => TEMP : INTEGER;
     end case;
   end record;

S:SENSOR(PRESION);
...
S.SITUACION:="SISTEMA A";
S.PRES:=723.5);
```

El valor dado al campo que gobierna el registro variante debe ser estático, de modo que el compilador pueda chequear la validez del registro.

Ejercicios:

1. Declarar un objeto tridimensional, de nombre PARALELEPIPEDO, con un rango de índices posible entre 1 y 20, que almacene valores enteros. Escribir un bloque que llene las caras externas de un cubo NxNxN con 1, y el interior con 0.

Solución.

```
with Ada.Text_IO; use Ada.Text_IO;
procedure CUBOS is
```

```
      subtype INDICE is INTEGER range 1..20;
      type PARALELEPIPEDO is array(INDICE range<>,INDICE range <>,
                                   INDICE range<>) of INTEGER;

      N:INDICE;
      package IIO is new Integer_IO(INTEGER); use IIO;
   begin
      put("Lado del cubo: ");
      get(N);
      put_line(" ");

      declare
         subtype INDCUBO is INDICE range INDICE'FIRST..N;
         CUBO: PARALELEPIPEDO(INDCUBO,INDCUBO,INDCUBO);
      begin
         for I in INDCUBO loop
            for J in INDCUBO loop
               for K in INDCUBO loop
                  if (I=INDCUBO'FIRST) or (I=INDCUBO'LAST)
                     or (J=INDCUBO'FIRST) or (J=INDCUBO'LAST)
                     or (K=INDCUBO'FIRST) or (K=INDCUBO'LAST) then
                        CUBO(I,J,K):=1;
                  else
                        CUBO(I,J,K):=0;
                  end if;
               end loop;
            end loop;
         end loop;

         -- Comprobación. Pintamos el cubo por capas
         for I in INDCUBO loop
            new_line(2);
            put(" capa ");
            put(I);
            new_line;
            for J in INDCUBO loop
                new_line;
                for K in INDCUBO loop
                   put(CUBO(I,J,K));
                   put(" ");
                end loop;
             end loop;
          end loop;
       end;
   end CUBOS;
```

2. Definir un tipo COMPLEJO. Declarar 4 variables C1, C2, CSUM y CPROD de tipo COMPLEJO y escribir las sentencias para asignar a CSUM la suma y a CPROD el producto de C1 y C2.

3.6 SUBPROGRAMAS

Los subprogramas son las unidades ejecutables básicas de los programas en Ada. Dos formas: procedures y funciones. Las llamadas a funciones son componentes de expresiones; las funciones devuelven un valor que forma parte de la expresión. Las llamadas a procedures son sentencias autónomas. Un procedure está formado por una serie de sentencias que definen una acción no elemental.

Los subprogramas, al igual que *packages* y *tasks*, pueden ser divididos en una parte de especificación y un cuerpo. La especificación define la interfaz entre el subprograma y el mundo exterior. Consta del nombre del

subprograma y cero, uno o más parámetros con sus tipos. En el caso de funciones, la especificación también incluye el tipo del valor que devuelve.

Los parámetros que aparecen en la llamada al subprograma se denominan parámetros reales; dentro del subprograma, son llamados parámetros formales. Los parámetros pueden ser pasados en una de tres maneras:

- **in** El subprograma usa el valor del parámetro real, valor que no puede modificar.
- **out** El subprograma crea un valor y no usa el valor del parámetro real.
- **in out** El subprograma usa el valor del parámetro real, y puede asignarle un nuevo valor.

Por defecto, funciona el modo in. En el caso de funciones, los parámetros sólo pueden ser pasados con el modo in, eliminando así los posibles efectos laterales.

3.6.1 Funciones

Cuerpo de una función:

```
function NOMBRE[PARAMETRO{,PARAMETRO}] return TIPO is
[PARTE DECLARATIVA]
begin
     {SENTENCIA}
     return EXPRESION;
end NOMBRE;
```

```
function FACTORIAL(N:POSITIVE) return POSITIVE is
begin
     if N=1 then
          return 1;
     else
          return N*FACTORIAL(N-1);  -- llamada recursiva
     end if;
end FACTORIAL;
```

Cada llamada a una función reserva un nuevo espacio para los objetos declarados en ella y para los parámetros. Estas nuevas copias desaparecen al abandonar la función.

```
type REAL is digits 7;
type VECTOR is array(INTEGER range <>) of REAL;
...
function SUMA(A:VECTOR) return REAL is
--parámetro formal no restringido
     RESULT:REAL:=0.0;
begin
     for I in A'RANGE loop
          RESULT:=RESULT+A(I);
     end loop;
     return RESULT;
end SUMA;
V:VECTOR(1..4):=(1.0,2.0,3.0,4.0);
S:REAL;
...
S:=SUMA(V);
```

El nombre de la función puede ser un identificador o bien uno de los siguientes símbolos, llamados designadores de función:

and	or	xor	=	<	<=	>	>=	+	-	&	abs
not	/	*	mod	rem	**						

En este caso, la función define un nuevo significado para el operador involucrado. Se sobrecarga el operador.

```
function "*"(A,B:VECTOR) return REAL is
     RESULT:REAL:=0.0;
begin
     if (A'FIRST/=B'FIRST) or (A'LAST/=B'LAST) then
          raise CONSTRAINT_ERROR;
     end if;
     for I in A'RANGE loop
          RESULT:=RESULT+A(I)*B(I);
     end loop;
     return RESULT;
end "*";

V:VECTOR(1..3):=(1.0,3.0,5.0);
W:VECTOR(1..3):=(2.0,4.0,6.0);
P:REAL:=8.0;
Q:REAL:=6.0;
R,S:REAL;
...
R:=V*W;      -- o bien R:="*"(V,W);
S:=P*Q;
```

El tipo de operandos indica el significado del operador *. En el primer caso, multiplicar dos vectores. En el segundo, multiplicar dos números reales.

3.6.2 Procedures

El cuerpo de un procedure es similar al de una función, con las siguientes diferencias:

- El procedure comienza con la palabra **procedure**.
- Su nombre debe ser un identificador.
- No devuelve un resultado.
- Los parámetros pueden ser pasados en modo in, out o in out. En los dos últimos casos, el parámetro real debe ser una variable.

Cuerpo de un procedure:

```
procedure CUADRATICA(A,B,C: in FLOAT; R1,R2: out FLOAT;
                     OK: out BOOLEAN) is
     Z:FLOAT;
begin
     Z:=B**2-4.0*A*C;
     if Z<0.0 then
          OK:=FALSE;
          R1:=0.0;
          R2:=0.0;
          return;
     end if;
     OK:=TRUE;
     R1:=(-B+SQRT(Z))/(2.0*A);
     R2:=(-B-SQRT(Z))/(2.0*A);
end CUADRATICA;
```

3.6.3 Especificaciones

Ejemplos de especificaciones:

```
procedure ROTACION (PUNTO: in out COORDENADAS; ANGULO: in RADIANES);
function COS(ANGULO:RADIANES) return FLOAT;
```

Un subprograma puede no incluir la parte de especificación, puesto que la cabecera del cuerpo incluye una parte de *interface* (que debe ser idéntica a la especificación, si ésta existe).

La utilización de especificaciones, aunque no siempre necesaria, es útil para mejorar la legibilidad de los subprogramas. Es buena costumbre agrupar las especificaciones de todos los subprogramas en la cabecera de la parte declarativa de un programa, a modo de resumen de los cuerpos de los subprogramas a utilizar.

La especificación es necesaria cuando dos suprogramas se llaman uno a otro, puesto que una entidad ha de ser declarada antes de ser usada.

```
procedure F(...);          -- especificación de F
procedure G(...) is        -- cuerpo de G
begin
     ...
     F(...);
     ...
end G;

procedure F(...) is        -- cuerpo de F
begin
     ...
     G(...);
     ...
end F;
```

Los subprogramas pueden ser definidos en cualquier parte donde una declaración sea apropiada: parte declarativa de un bloque, cuerpo de otro subprograma, etc. El alcance del subprograma abarca desde el punto en que es definido hasta el final de la parte en que está definido.

3.6.4 Llamadas a subprogramas

- Notación posicional
  `CUADRATICA(-2.0,3.0,4.0,X1,X2,STATUS);`
- Por asociación de parámetros
  `CUADRATICA(A=>-2.0, OK=>STATUS, B=>3.0, C=>4.0, R1=>X1,R2=>X2);`

  No tiene que conservarse el mismo orden de los parámetros formales.

3.6.5 Subprogramas con sobrecarga

Un mismo nombre de subprograma puede ser utilizado para representar dos operaciones diferentes, de la misma forma que vimos con los operadores (función "*", por ejemplo).

```
procedure SET(LISTING: in BOOLEAN);
procedure SET(PIXEL:in COLOR;FRAME:in out BUFFER);
procedure SET(PRIORITY: in POSITIVE);
procedure SET(ADDRESS: in NATURAL);
```

Una llamada al procedure SET es ambigua si el compilador no puede identificar exactamente a qué especificaciones se refiere. El compilador examina el tipo y orden de los parámetros reales, el nombre de los parámetros formales y, en el caso de funciones, el tipo de resultado, para resolver una llamada a un subprograma con sobrecarga.

Las siguientes llamadas no serán ambiguas:

```
SET(BLUE, MY_BUFFER);
SET(TRUE);
```

Sin embargo, SET(1234) es ambigua. Para eliminar la ambigüedad, podemos hacer dos cosas:

- Utilizar el nombre del parámetro formal:

```
SET(ADDRESS=>1234);
```

o bien

- Utilizar el tipo de parámetro:

```
SET(NATURAL'(1234));
```

Un apunte final:

Un subprograma no puede ser llamado en la elaboración de una parte declarativa si su cuerpo aparece después de esa llamada:

```
function A return INTEGER;
I:INTEGER:=A;       -- Ilegal
...
function A return INTEGER is
...
```

Ejercicios:

1. Ejemplo de un programa con procedures y arrays. El procedure invierte los elementos del array.

```
with Text_IO; use Text_IO;
procedure VECTORES is

type VECTOR is array(INTEGER range <>) of INTEGER;
V:VECTOR(1..8);

package IIO is new Integer_IO(INTEGER); use IIO;

procedure REV(X:in out VECTOR) is
     AUX:INTEGER;
begin
     for I in X'first..X'last/2 loop
          AUX:=X(I);
          X(I):=X(X'LAST-I+1);
          X(X'LAST-I+1):=AUX;
     end loop;
end REV;

begin
     for I in V'RANGE loop       --leer vector
          put("elemento " & INTEGER'IMAGE(I) & ":");
          get(V(I));
     end loop;
     REV(V);
     for I in V'RANGE loop       --escribir vector invertido
          put(V(I));
          put_line(" ");
     end loop;
end VECTORES;
```

2. Escribir una función HACER_UNIDAD(N) que devuelva una matriz unidad real NxN. Escribir un programa ejemplo que utilice la función.

3. Escribir funciones "*" y "+" para multiplicar y sumar dos valores complejos. Escribir procedures para leer y escribir complejos.

```
with Text_IO; use Text_IO;
procedure NUMEROS_COMPLEJOS is

type COMPLEJOS is
```

```
      record
         RE,IM:FLOAT;
      end record;

   C1,C2 : COMPLEJOS;
   package FIO is new Float_IO(FLOAT); use FIO;

   function "+"(A,B:COMPLEJOS) return COMPLEJOS is
   begin
      return(A.RE+B.RE,A.IM+B.IM);
   end "+";

   function "*"(A,B:COMPLEJOS) return COMPLEJOS is
   begin
      return(A.RE*B.RE-A.IM*B.IM,A.RE*B.IM+A.IM*B.RE);
   end "*";

   procedure LEEC(A: out COMPLEJOS) is

   begin
      put("Componente real: ");
      get(A.RE);
      new_line;
      put("Componente imaginaria: ");
      get(A.IM);
   end LEEC;

   procedure ESCRIBEC(A: in COMPLEJOS) is

   begin
      new_line;
      put("(");
      put(A.RE);
      put(",");
      put(A.IM);
      put_line(")");
   end ESCRIBEC;

   begin     --NUMEROS_COMPLEJOS
      LEEC(C1);
      LEEC(C2);
      ESCRIBEC(C1+C2);
      ESCRIBEC(C1*C2);
   end NUMEROS_COMPLEJOS;
```

4. Escribir una función "<" para comparar dos valores de tipo FECHA.

```
   type FECHA is
      record
        DIA:INTEGER range 1..31;
        MES:NOMBRE_MES;
        ANIO:INTEGER range 0..3000;
      end record;
```

3.7 PACKAGES

Un package consiste en una colección de entidades o recursos computacionales relacionados . El package encapsula esos recursos. Consta de dos partes: especificación y cuerpo.

La parte especificativa es la parte visible del package. Especifica qué contiene el package y cómo ha de ser usado. Es la interfaz entre el package y el usuario. El cuerpo del package consiste en la implementación de los recursos que aporta el mismo, detalles ocultos para el usuario.

La estructura del package soporta los principios de modularidad, abstracción, localización y ocultamiento de información.

Uno de los problemas con los lenguajes estructurados tradicionales, tales como PASCAL, es que no ofrecen un suficiente control de la visibilidad. Supongamos por ejemplo una pila representada por un array, con una variable que refiere al elemento tope, un procedure PUSH para añadir un elemento, y una función POP para borrar un elemento. En PASCAL:

```
program PRUEBA;
const
     MAX=100;
var
     S:ARRAY[1..MAX] of INTEGER;
     TOP:0..MAX;

procedure PUSH(X:INTEGER);
begin
     TOP:=TOP+1;
     S[TOP]:=X;
end;

function POP:INTEGER;
begin
     TOP:=TOP-1;
     POP:=S[TOP+1];
end;

begin
     TOP:=0;
     ...
end;
```

No hay forma de utilizar los subprogramas PUSH y POP sin dar acceso a las variables S y TOP. La implementación de la pila es visible para el usuario.

En Ada, el uso de packages permite ocultar los detalles, y dar acceso solamente a aquello que debe ser visible:

```
package PILA is         -- especificación
     procedure PUSH(X:INTEGER);
     function POP return INTEGER;
end PILA;

package body PILA is    -- cuerpo
     MAX: constant:=100;
     S:ARRAY(1..MAX) of INTEGER;
     TOP: INTEGER range 0..MAX;
     procedure PUSH(X:INTEGER) is
     begin
          TOP:=TOP+1;
          S(TOP):=X;
     end PUSH;

     function POP return INTEGER is
     begin
          TOP:=TOP-1;
          return S(TOP+1);
```

```
     end POP;
begin        -- Inicialización. Opcional. En este ejemplo, se podría
   TOP:=0;   -- omitir si declaramos TOP:INTEGER range 0..MAX:=0;
end PILA;
```

La especificación de un package no puede contener cuerpos de subprogramas, obviamente.

Un package puede ser declarado en cualquier parte declarativa, tal como en un bloque, subprograma, u otro package. El package existe hasta el final del alcance donde es declarado.

```
declare                              declare
     package PILA is                      package PILA is
          ...                             ...
     package body PILA is                 package body PILA is
          ...                             ...
begin                          ≡         use PILA;
     ...                               begin
     PILA.PUSH(M);                     ...
     ...                                    PUSH(M);
     N:=PILA.POP;                      ...
     ...                                    N:=POP;
end;                                   ...
                                       end;
```

Aplicaciones recomendadas para packages:

a) Colección de declaraciones. Agrupamiento de objetos y tipos, con la consiguiente mejora en el mantenimiento.

```
package METRIC_EARTH_CONSTANTS is
     EQUATORIAL_RADIUS: constant:=6378.145;          -- Km
     GRAVITATION_CONSTANT: constant:=3.986_012e5;    -- Km^3/s^2
     SPEED_UNIT: constant:=7.905_368_28;             -- Km/s
     TIME_UNIT: constant:=806.811_874_4;             -- s
end METRIC_EARTH_CONSTANTS;
```

No se necesita cuerpo para este package.

b) Grupos de unidades de programas relacionadas.

```
package FUNCIONES_TRASCENDENTALES is
     function COS(ANG:FLOAT) return FLOAT;
     function SIN(ANG:FLOAT) return FLOAT;
     function TAN(ANG:FLOAT) return FLOAT;
end  FUNCIONES_TRASCENDENTALES;
package body  FUNCIONES_TRASCENDENTALES is
     ...
```

3.8 UNIDADES DE BIBLIOTECA

En cualquier sistema *software* es conveniente, y muchas veces necesario, hacer que cada unidad sea independiente del resto. Puede haber diferentes grupos trabajando en distintas partes del sistema; los test del *software* mejoran si las unidades individuales están fisicamente separadas, pudiéndose así estudiar cada una aisladamente. Para ello, es necesario que cada unidad pueda ser compilada por separado.

Ada permite procesar el texto de un programa en una o más compilaciones. Una compilación consta de una o más unidades de compilación. Una unidad de compilación puede ser: una especificación de un package o de un subprograma; el cuerpo de un package o de un subprograma (unidades secundarias), una unidad genérica (a estudiar en un tema posterior) o una subunidad. La unidad principal de un programa ha de ser un subprograma. Las unidades de compilación, una vez compiladas pasan a formar parte de una **biblioteca de programa**, y reciben el nombre de unidades de biblioteca. Cada unidad ha de tener un nombre único. Además de producir el código objeto

correspondiente a la unidad, el compilador también coloca en la biblioteca la información que describe la interfaz que la unidad presenta al exterior.

Si tenemos alguna unidad de biblioteca compilada previamente, otra unidad puede aplicar una cláusula **with** para tener visibilidad de la misma.

```
with PILA;
procedure PRINCIPAL is
use PILA;
M,N:INTEGER;
begin
     ...
     PUSH(M);
     ...
     N:=POP;
     ...
end PRINCIPAL;
```

Al encontrar la cláusula with, el compilador obtiene de la biblioteca la información que describe la interfaz dado por la unidad.

Desarrollo *Bottom-Up* (de abajo a arriba)

La metodología de diseño llamada *Bottom-Up* permite la creación de unidades que pueden ser utilizadas por muchos módulos. En Ada, las unidades de biblioteca se utilizan para implementar esta metodología; se crean packages y subprogramas que proporcionan las facilidades primitivas necesarias, y a partir de ellas se construyen programas.

3.8.1 Subunidades

Desarrollo *Top-Down* (de arriba a abajo)

La metodología de diseño *Top-Down* implica comenzar en los niveles más altos de abstracción, y a partir de ellos, descomponer el sistema en niveles más primitivos. En Ada, se utilizan las subunidades para soportar esta metodología. Cuando descomponemos un sistema, podemos diseñar la especificación de un subprograma, package o task, y posponer la implementación del cuerpo correspondiente. Considerar el ejemplo siguiente:

```
procedure MAIN is
     package TRANSFORM is
          procedure DECYPHER(MESSAGE: in out STRING);
          procedure ENCODE(MESSAGE: in out STRING);
     end TRANSFORM;
     package body TRANSFORM is separate;
     procedure REPORT(MESSAGE: in STRING) is separate;
begin
     -- body de MAIN
end MAIN;
```

Podemos compilar por separado los cuerpos del package y del subprograma, utilizando la forma siguiente:

```
separate(MAIN)-- identifica la unidad padre
package body TRANSFORM is ...

separate(MAIN)
procedure REPORT(MESSAGE: in STRING) is ...
```

Si una subunidad tiene incluída otra subunidad, la cláusula separate debe incluir el nombre completo del padre. Si, por ejemplo, TRANSFORM tiene una subunidad, la cláusula será:

```
separate(MAIN.TRANSFORM)
```

Puesto que cada subunidad debe incluir explícitamente el nombre del padre, esta forma de diseño no es apropiada en el caso en que varias unidades de programa necesiten utilizar las facilidades que aporta la subunidad.

3.8.2 ORDEN DE COMPILACION Y RECOMPILACION

Cuando un sistema está construído a partir de varias unidades de biblioteca, las dependencias existentes entre ellas exigen un determinado orden de compilación. Básicamente, la regla es que una unidad dada debe ser compilada antes de que sea visible para otra unidad.

- La especificación de un subprograma, package o task debe ser compilada antes que el cuerpo.
- Una unidad debe ser compilada antes que sus subunidades.
- Si una unidad llama a otra con una cláusula with, la unidad llamada debe ser compilada primero.

3.8.2.1 Reglas de recompilación

Si se modifica	Se debe recompilar
Especificación	La especificación, el cuerpo, sus subunidades, y todas las unidades que dependen de la misma (que la mencionan en una cláusula with)
Cuerpo	El cuerpo y sus subunidades
Subunidad	La subunidad y sus subunidades

Ejercicios:

1. Escribir un package NUMEROS_COMPLEJOS que haga visible:

El tipo COMPLEJO

Una constante i=√-1

Funciones +, -, *, / que actúen sobre valores de tipo COMPLEJO

Procedimientos para leer y escribir valores de tipo COMPLEJO

3.9 TIPOS PRIVATE

Hemos visto cómo la estructura package nos permite ocultar al usuario la implementación de objetos. Vamos a ver en este capítulo la definición de tipos private, que nos permite ocultar al usuario los detalles de la construcción de un tipo.

En el package NUMEROS_COMPLEJOS, ya escrito, hemos definido un tipo COMPLEJO, una constante I, y determinadas operaciones con datos de tipo COMPLEJO. El problema que esta estructura puede presentar es que el usuario puede hacer uso del hecho de que los números complejos están definidos en representación cartesiana. En lugar de utilizar el operador complejo "*", el usuario puede escribir sentencias como: C.IM:=C.IM+1.0;, en lugar de algo más abstracto como: C:=C+I;.

Podemos prevenir el uso del conocimiento de la representación interna del tipo. Considerar la siguiente definición:

```
package NUMEROS_COMPLEJOS is
type COMPLEJO is private;
     I:constant COMPLEJO;
     function "+"(X:COMPLEJO) return COMPLEJO;
     function "-"(X:COMPLEJO) return COMPLEJO;
     function "*"(X,Y:COMPLEJO) return COMPLEJO;
     function "/"(X,Y:COMPLEJO) return COMPLEJO;
     function CONST(R,I:FLOAT) return COMPLEJO;
     function RE_PART(X:COMPLEJO) return FLOAT;
     function IM_PART(X:COMPLEJO) return FLOAT;

private
     type COMPLEJO is
        record
          RE,IM:FLOAT;
        end record;
     I: constant COMPLEJO:=(0.0,1.0);
end;
```

La parte de la especificación que precede a la palabra reservada private es la parte visible del package, y da la información disponible desde fuera del package. Fuera del package se conoce que existe un tipo llamado COMPLEJO, pero no se conoce nada acerca de su implementación. Las únicas operaciones disponibles para elementos de tipo COMPLEJO son: asignación, =, /=, y los subprogramas especificados en el package.

Después de la palabra private se esconden los detalles de los tipos declarados como private, y se asignan los valores iniciales a las constantes de esos tipos.

Un tipo privado puede ser implementado en cualquier forma consistente con las operaciones visibles para el usuario, sin necesidad de que el usuario tenga que reformar sus programas. Para nuestro ejemplo, el tipo COMPLEJO podría implementarse como un array de dos componentes, o también en coordenadas polares. Podría ser necesario cambiar las implementaciones de los cuerpos de los subprogramas, en el cuerpo del package, pero la especificación del package permanecería igual (excepto la parte private).

Según esta nueva definición del tipo COMPLEJO, una sentencia del programa del usuario, tal como C:=(1.0,1.0); (o bien C.RE:=1.0; C.IM:=1.0;) debería ser sustituída por C:=CONST(1.0,1.0);

Otro ejemplo de uso de tipos private:

```
package MANAGER is
   type PASSWORD is private;
   function GET return PASSWORD;
   function IS_VALID(P:PASSWORD) return BOOLEAN;
   private
      type PASSWORD is range 0..7000;
end MANAGER;
```

```
...

use MANAGER;
MI_PASSWORD,TU_PASSWORD: PASSWORD;

begin
   ...
   MI_PASSWORD:=GET;
   TU_PASSWORD:=MI_PASSWORD;  -- permitido
   TU_PASSWORD:=8;  -- no permitido: el tipo PASSWORD está oculto⇒
                -- no sabemos qué valores se le pueden asociar.
```

Ejercicios.

1. Completar el package RACIONALES, cuya especificación es:

```
package RACIONALES is
-- Representación y manipulación de números racionales

  type RACIONAL is private;

--Operadores
  function "/" (X : INTEGER; Y : POSITIVE) return RACIONAL;
  -- constructor: retorna un nº racional simplificado
  function NUMER (R : RACIONAL) return INTEGER;
  function DENOM (R : RACIONAL) return POSITIVE;
-- selectores: retornan el numerador y el denominador de un nº racional R

  procedure GET (ITEM : out RACIONAL);
  procedure PUT (ITEM : in RACIONAL);

  function "+"(R1,R2 : RACIONAL) return RACIONAL;
  function "-"(R1,R2 : RACIONAL) return RACIONAL;
  function "*"(R1,R2 : RACIONAL) return RACIONAL;
  function "/"(R1,R2 : RACIONAL) return RACIONAL;

  function "<" (R1,R2 : RACIONAL) return BOOLEAN;
  function ">" (R1,R2 : RACIONAL) return BOOLEAN;
  function "<="(R1,R2 : RACIONAL) return BOOLEAN;
  function ">="(R1,R2 : RACIONAL) return BOOLEAN;

private
  type RACIONAL is
    record
      NUMERADOR     : INTEGER := 0;
      DENOMINADOR : POSITIVE:= 1;
    end record; -- Racional
end RACIONALES;
```

Solución:

```
with Text_IO; use Text_IO;
package body RACIONALES is
   package IIO is new Integer_IO(INTEGER); use IIO;
-- local function MCD

  function MCD(M: POSITIVE; N: POSITIVE) return POSITIVE is
  -- MCD  de M y N, mediante el algoritmo de Euclides
    R : NATURAL;
    TEMPM: POSITIVE;
```

```
    TEMPN: POSITIVE;
  begin -- MCD
    TEMPM := M;
    TEMPN := N;
    R := TEMPM rem TEMPN;

    while R /= 0 loop
      TEMPM := TEMPN;
      TEMPN := R;
      R := TEMPM rem TEMPN;
    end loop;
    return TEMPN;
  end MCD;

  -- Operadores

  function "/" (X : INTEGER; Y : POSITIVE) return RACIONAL is
    G: POSITIVE;
  begin
    if X = 0 then
      return (NUMERADOR => 0, DENOMINADOR => 1);
    end if;
    G := MCD(abs X, Y);
    return (NUMERADOR => X/G, DENOMINADOR => Y/G);
  end "/";

  function NUMER (R : RACIONAL) return INTEGER is
  begin
    return R.NUMERADOR;
  end NUMER;

  function DENOM (R : RACIONAL) return POSITIVE is
  begin
    return R.DENOMINADOR;
  end DENOM;

  procedure GET (ITEM : out RACIONAL) is
    N: INTEGER;
    D: POSITIVE;
  begin -- Get
    Put("Numerador: ");
    Get(N);
    Put("Denominador: ");
    Get(D);
    ITEM := N/D;
  end GET;

  procedure PUT (ITEM : in RACIONAL) is
  begin -- PUT
    Put(Item => NUMER(ITEM), Width => 1);
    Put(Item => '/');
    Put(Item => DENOM(ITEM), Width => 1);
  end PUT;

  function "+"(R1,R2 : RACIONAL) return RACIONAL is
    N: INTEGER;
    D: POSITIVE;
  begin
    N := NUMER(R1) * DENOM(R2) + NUMER(R2) * DENOM(R1);
    D := DENOM(R1) * DENOM(R2);
```

```
    return N/D;
  end "+";


  function "*"(R1,R2 : RACIONAL) return RACIONAL is
    N: INTEGER;
    D: POSITIVE;
  begin
    N := NUMER(R1) * NUMER(R2);
    D := DENOM(R1) * DENOM(R2);
    return N/D;
  end "*";

  function "-"(R1,R2 : RACIONAL) return RACIONAL is
    N : INTEGER;
    D : POSITIVE;
  begin
    N := NUMER(R1) * DENOM(R2) - NUMER(R2) * DENOM(R1);
    D := DENOM(R1) * DENOM(R2);
    return N/D;
  end "-";

  function "/"(R1,R2 : RACIONAL) return RACIONAL is
    N : INTEGER;
    D : POSITIVE;
  begin
    N := NUMER(R1) * DENOM(R2);
    D := NUMER(R2) * DENOM(R1);
    return N/D;
  end "/";

  function "<" (R1,R2 : RACIONAL) return BOOLEAN is
  begin
    return NUMER(R1) * DENOM(R2) < NUMER(R2) * DENOM(R1);
  end "<";

  function ">" (R1,R2 : RACIONAL) return BOOLEAN is
  begin
    return NUMER(R1) * DENOM(R2) > NUMER(R2) * DENOM(R1);
  end ">";

  function "<=" (R1,R2 : RACIONAL) return BOOLEAN is
  begin
    return NUMER(R1) * DENOM(R2) <= NUMER(R2) * DENOM(R1);
  end "<=";

  function ">=" (R1,R2 : RACIONAL) return BOOLEAN is
  begin
    return NUMER(R1) * DENOM(R2) >= NUMER(R2) * DENOM(R1);
  end ">=";

end RACIONALES;

with RACIONALES, Text_IO;
use RACIONALES, Text_IO;
procedure TESTRACIONAL is
  A,B,C,D,E,F : RACIONAL;
begin
  A := 1/3;
  B := (-2)/4;
```

```
   PUT("Número A: ");
   PUT(A);
   PUT("Número B: ");
   PUT(B);
   PUT_LINE("Introducir número C");
   GET(C);
   PUT_LINE("Introducir número D");
   GET(D);
   E := A + B;
   PUT("E = A + B = ");
   PUT(E);
   NEW_LINE;
   F := C * D;
   PUT("F = C * D = ");
   PUT(F);
   NEW_LINE;
   PUT(Item => "A + E * F = ");
   PUT(Item => A + E * F);
   NEW_LINE;
   if C<D then
      PUT(C);
      PUT(" MENOR QUE ");
      PUT(D);
   end if;

end TESTRACIONAL;
```

3.9.1 TIPOS LIMITED PRIVATE

Las operaciones aplicable a tipos private pueden quedar restringidas solamente a las especificadas en la parte visible del package. Esto se consigue declarando el tipo como **limited private**. En este caso, las sentencias de asignación y las de comparación =, /= predefinidas ya no estarán disponibles para el usuario del package.

Ejercicios.

2. Completar el package PILA, cuya especificación es:

```
package PILA is
   type STACK is limited private;
   procedure PUSH(P:in out STACK; X:in INTEGER);
   procedure POP(P:in out STACK; X: out INTEGER);
   function VACIA(P:STACK) return BOOLEAN;
   function LLENA(P:STACK) return BOOLEAN;
   procedure PUT(P:in STACK);
   function LONG(P:STACK) return INTEGER;

   private
      MAX: constant:=100;
      type INTEGER_VECTOR is array(INTEGER range <>) of INTEGER;
      type STACK is
         record
            S: INTEGER_VECTOR(1..MAX);
            TOP: INTEGER range 0..MAX := 0;
         end record;
end PILA;
```

3.10 TIPO ACCESS

Objetos de este tipo toman valores que proporcionan acceso a otros objetos:

```
type BUFFER is
   record
      MESSAGE: string(1..10);
      PRIORITY: POSITIVE;
   end record;

type BUFFER_POINTER is access BUFFER;
MY_P, YOUR_P, THEIR_P : BUFFER_POINTER;
```

Inicialmente, las tres variables de BUFFER_POINTER toman el valor **null**, el cual representa el hecho de que actualmente no apuntan a nada. Este es el único caso en que Ada define un valor implícito.

```
MY_P := new BUFFER;
YOUR_P := new BUFFER'("----------",1);
THEIR_P := YOUR_P;
```

Las dos primeras sentencias crean un nuevo objeto de tipo BUFFER, y sus direcciones son asignadas a las variables MY_P y YOUR_P. Con esta última, asignamos explícitamente valores a los componentes del record. Si posteriormente se ejecuta la sentencia YOUR_P := null; el objeto al que apuntaba ya no será accesible. Además, estos objetos dinámicos serán destruídos una vez salgamos del alcance donde los objetos access están declarados.

THEIR_P apuntará al mismo objeto que YOUR_P.

YOUR_P.all := MY_P.all => todos los elementos del objeto señalado por YOUR_P tomarán los valores de los elementos del BUFFER señalado por MY_P. Pero YOUR_P /= MY_P, pues apuntan a distintos objetos.

Uno de los usos más comunes para objetos access es el de implementación de listas ligadas:

```
type CELL;
type LINK is access CELL;
type CELL is
   record
      VALUE:INTEGER;
      NEXT:LINK;
   end record;

L, N : LINK;
...
L := new CELL'(VALUE => 1, NEXT => null);
for I in 1..5 loop
   N := new CELL'(VALUE=>I*10, NEXT=>L);
   L:=N;
end loop;
```

Ejercicios.

1. Modificar el package PILA de modo que el tipo STACK esté implementado utilizando el tipo access.
2. Completar el package cuya parte visible es:

```
package COLA is
   type QUEUE is limited private;
   procedure PONER(Q: in out QUEUE; X: in ITEM);
   procedure QUITAR(Q: in out QUEUE; X: out ITEM);
   function VACIA(Q:QUEUE) return BOOLEAN;
   function LONG(Q:QUEUE) return INTEGER;
   private ....
end COLA;
```

3.11 UNIDADES GENERICAS

Supongamos que necesitamos un subprograma para ordenar valores de un tipo dado (INTEGER, por ejemplo). Tal subprograma podría tener la siguiente especificación:

```
type INT_ARRAY is array(natural range <>) of INTEGER;
procedure SORT(MY_ARRAY: in out INT_ARRAY);
```

Para ordenar un array de elementos de otro tipo debemos crear otro subprograma (sobrecargar el procedure SORT) debido a las reglas de tipología fuerte del Ada. Nos veremos forzados a escribir:

```
type REAL_ARRAY is array(natural range <>) of FLOAT;
procedure SORT(MY_ARRAY: in out REAL_ARRAY);
```

Sería muy deseable disponer de unidades de programa 'patrón', que sean escritas una vez y posteriormente ajustadas a las necesidades de tipo particulares. Para ello, Ada permite la creación de packages y subprogramas genéricos. Son unidades patrón, no utilizables directamente. Se deben crear ejemplos particulares, que pueden ser usados como unidades ordinarias.

3.11.1 Definición

Para crear una unidad genérica, podemos tomar una especificación de package o subprograma, y añadirle un prefijo, llamado parte genérica, que define los parámetros genéricos.

Por ejemplo, podemos crear el siguiente subprograma que intercambia dos valores enteros:

```
procedure SWAP(UNO,DOS: in out INTEGER);
   AUX : INTEGER;
begin
   AUX:=UNO;
   UNO:=DOS;
   DOS:=AUX;
end SWAP;
```

Si esta aplicación es necesaria para intercambiar otro tipo de elementos, como el algoritmo sería el mismo no es necesario escribir un nuevo subprograma para cada situación:

```
generic
   type ELEMENT is private;
procedure GEN_SWAP(UNO,DOS: in out ELEMENT);
El cuerpo será:
procedure GEN_SWAP(UNO,DOS: in out ELEMENT) is
   AUX : ELEMENT;
begin
    AUX:=UNO;
    UNO:=DOS;
   DOS:=AUX;
end GEN_SWAP;
```

Tanto la especificación de la unidad genérica como el cuerpo deben ser compilados antes de su uso.

3.11.2 Uso

Debemos dar un identificador que nombre la unidad ejemplo. Debemos también asociar un parámetro real a cada parámetro genérico. Así, para la unidad GEN_SWAP, tendremos:

```
procedure INT_SWAP is new GEN_SWAP(INTEGER);
procedure CHAR_SWAP is new GEN_SWAP(CHARACTER);
...
C1,C2:CHARACTER;
...
```

```
CHAR_SWAP(C1,C2);
```

La particularización de una unidad genérica puede realizarse en cualquier lugar donde puede ser declarado un package o un subprograma.

3.11.3 Parámetros genéricos

3.11.3.1 Tipos genéricos

El parámetro real asociado al parámetro genérico debe ser compatible con el mismo.

Parámetro formal	Parámetro real
type GENERAL is limited private;	Cualquier tipo
type ELEMENT is private;	Cualquier tipo que permita asignación y test igualdad (desigualdad)
type ELEMENT is (<>)	Tipo discreto
type ELEMENT is range <>	Tipo entero
type ELEMENT is digits <>	Tipo punto flotante
type ELEMENT is delta <>	Tipo punto fijo
type PUNT is access OBJECTO	Tipo access que designe al mismo tipo de objeto
type MATRIZ is array (INDICE) of ELEMENTO	Array con igual nº de dimensiones, tipos de índices y tipo de componentes

Ejemplos:

```
generic
   type T is (<>);
function NEXT(X: T) return T;
function NEXT(X: T) return T is
begin
   if X=T'LAST then
      return T'FIRST;
   else
      return T'SUCC(X);
   end if;
end NEXT;
with NEXT;
procedure TEST is
   type DIAS is (LUN,MAR,MIE,JUE,VIE,SAB,DOM);
   function MANIANA is new NEXT(DIAS);
   subtype DIGITO is INTEGER range 0..9;
   function SIG_DIG is new NEXT(DIGITO);
   ...
```

UNCHECKED_DEALLOCATION: subprograma genérico predefinido:

```
generic
   type OBJECT is limited private;
   type NAME is access OBJECT;
procedure UNCHECKED_DEALLOCATION(X: in out NAME);


with UNCHECKED_DEALLOCATION;
...
type CELL;
type LIST is access CELL;
type CELL is
```

```
      record
         NUM: INTEGER;
         SIG: LIST;
      end record;

   procedure DISPOSE is new UNCHECKED_DEALLOCATION(CELL,LIST);
   ...
   L : LIST;
   ...
      DISPOSE(L);
   ...
```

El package TEXT_IO incluye cuatro unidades genéricas:

```
   generic
      type NUM is range <>;
   package INTEGER_IO is ...

   generic
      type NUM is digits <>;
   package FLOAT_IO is ...

   generic
      type NUM is delta <>;
   package FIXED_IO is ...

   generic
      type ENUM is (<>);
   package ENUMERATION_IO is ...
```

La particularización de estas unidades nos permitirá leer y escribir valores de tipos enteros, punto flotante, punto fijo y de tipos enumerados, respectivamente.

```
   ...
   with IO_EXCEPTIONS;
   package TEXT_IO is
      type FILE_TYPE is limited private;
      type FILE_MODE is (IN_FILE, OUT_FILE);

      type COUNT is range 0 .. implementation_defined;
      subtype POSITIVE_COUNT is COUNT range 1 .. COUNT'LAST;
      UNBOUNDED: constant COUNT:= 0; -- line and page length

      subtype FIELD is INTEGER range 0.. implementation_defined;
      subtype NUMBER_BASE is INTEGER range 2..16;
      type TYPE_SET is (LOWER CASE, UPPER_CASE);

      -- File management

      -- CREATE, OPEN, CLOSE, DELETE, RESET, MODE, NAME,
      -- FORM and IS_OPEN as for SEQUENTIAL_IO

      procedure CREATE(FILE: in out FILE_TYPE; MODE: in FILE_MODE:=OUT_FILE;
                       NAME: in STRING:=""; FORM: in STRING:="");
      procedure OPEN(FILE: in out FILE_TYPE; MODE: in FILE_MODE;
                     NAME: in STRING; FORM: in STRING:="");
      procedure CLOSE(FILE: in out FILE_TYPE);
      procedure DELETE(FILE: in out FILE_TYPE);
      procedure RESET(FILE: in out FILE_TYPE);
      function MODE(FILE: in FILE_TYPE) return FILE_MODE;
      function NAME(FILE: in FILE_TYPE) return STRING;
      function FORM(FILE: in FILE_TYPE) return STRING;
```

```
        function IS_OPEN(FILE: in FILE_TYPE) return BOOLEAN;

     ------------------------------------------------------
        -- Control of default input, output and error files --
        ------------------------------------------------------

        procedure Set_Input  (File : in File_Type);
        procedure Set_Output (File : in File_Type);
        procedure Set_Error  (File : in File_Type);

        function Standard_Input  return File_Type;
        function Standard_Output return File_Type;
        function Standard_Error  return File_Type;

        function Current_Input  return File_Type;
        function Current_Output return File_Type;
        function Current_Error  return File_Type;

     -- Specification of line and page lengths

        procedure SET_LINE_LENGTH(TO: in COUNT);
        procedure SET_PAGE_LENGTH(TO: in COUNT);
        function LINE_LENGTH return COUNT;
        function PAGE_LENGTH return COUNT;

        -- also with FILE parameter

        -- Column, line and page control

        procedure NEW_LINE(SPACING: in POSITIVE_COUNT:= 1);
        procedure SKIP_LINE(SPACING: in POSITIVE_COUNT:= 1);
        function END_OF_LINE return BOOLEAN;
        procedure NEW_PAGE;
        procedure SKIP_PAGE;
        function END_OF_PAGE return BOOLEAN;
        function END_OF_FILE return BOOLEAN;
        procedure SET_COL(TO: in POSITIVE_COUNT);
        procedure SET_LINE(TO: in POSITIVE_COUNT);
        function COL return POSITIVE_COUNT;
        function LINE return POSITIVE_COUNT;
        function PAGE return POSITIVE_COUNT;

        -- also with FILE parameter

        -- Character input-output

        procedure GET(FILE: in FILE_TYPE; ITEM: out CHARACTER);
        procedure GET(ITEM: out CHARACTER);
        procedure PUT(FILE: in FILE_TYPE; ITEM: in CHARACTER);
        procedure PUT(ITEM: in CHARACTER);

        -- String input-output

        procedure Get (File : in File_Type; Item : out String);
        procedure Get (Item : out String);
        procedure Put (File : in File_Type; Item : in String);
        procedure Put (Item : in String);
        procedure  Get_Line(File:in  File_Type;  Item:out  String;  Last:out
Natural);
        procedure Get_Line(Item : out String; Last : out Natural);
```

```
        procedure Put_Line(File : in File_Type; Item : in String);
        procedure Put_Line(Item : in String);

     -- Generic package for input-output of integer types

        generic
           type NUM is range <>;
        package INTEGER_IO is
           DEFAULT_WIDTH: FIELD:= NUM'WIDTH;
           DEFAULT_BASE: NUMBER_BASE:= 10;

           procedure GET(ITEM: out NUM; WIDTH: in FIELD:= 0);
           procedure PUT(ITEM: in NUM; WIDTH: in FIELD:= DEFAULT_WIDTH;
                         BASE: in NUMBER_BASE:= DEFAULT_BASE);
           procedure GET(FROM: in STRING; ITEM: out NUM; LAST: out POSITIVE);
           procedure PUT(TO: out STRING; ITEM: in NUM;
                         BASE: in NUMBER_BASE:= DEFAULT_BASE);
        end INTEGER_IO;

         -- Generic packages for input-output of real types

        generic
           type NUM is digits <>;
        package FLOAT_IO is
           DEFAULT_FORE: FIELD:= 2;
           DEFAULT_AFT: FIELD:= NUM'DIGITS-1;
           DEFAULT_EXP: FIELD:= 3;

           procedure GET(ITEM: out NUM; WIDTH: in FIELD:= 0);
           procedure PUT(ITEM: in NUM; FORE: in FIELD:= DEFAULT_FORE;
                     AFT:   in   FIELD:=   DEFAULT_AFT;   EXP:   in   FIELD:=
DEFAULT_EXP);
           procedure GET(FROM: in STRING; ITEM: out NUM; LAST: out POSITIVE);
           procedure   PUT(TO:out   STRING;   ITEM:in   NUM;   AFT:in   FIELD:=
DEFAULT_AFT;
                         EXP: in FIELD:= DEFAULT_EXP);
         end FLOAT_IO;

        generic
           type NUM is delta <>;
        package FIXED IO is
           DEFAULT_FORE: FIELD:= NUM'FORE;
           DEFAULT_AFT: FIELD:= NUM'AFT;
           DEFAULT_EXP: FIELD:= 0;
           -- then as for FLOAT_IO
        end FIXED_IO;

        -- Generic package for input-output of enumeration types

        generic
           type ENUM is (<>);
        package ENUMERATION_IO is
           DEFAULT_WIDTH: FIELD:= 0;
           DEFAULT_SETTING: TYPE_SET:= UPPER CASE;

           procedure GET(ITEM: out ENUM);
           procedure PUT(ITEM: in ENUM; WIDTH: in FIELD:= DEFAULT_WIDTH;
                         SET: in TYPE_SET:= DEFAULT_SETTING);
           procedure GET(FROM: in STRING; ITEM: out ENUM; LAST: out POSITIVE);
           procedure PUT(TO: out STRING; ITEM: out ENUM;
```

```
                    SET: In TYPE_SET:= DEFAULT_SETTING);
      end ENUMERATION_IO;

      -- Exceptions

      STATUS_ERROR: exception renames IO_EXCEPTIONS.STATUS_ERROR;
      . . .
      LAYOUT_ERROR: exceptlon renames IO_EXCEPTIONS.LAYOUT_ERROR;

      private
         -- implementation-dependent;
   end TEXT_IO;
```

Ejemplos

```
   package IIO:
   with TEXT_IO; use TEXT_IO;
   package IIO is new INTEGER_IO(INTEGER);

   package FIO:
   with TEXT_IO; use TEXT_IO;
   package FIO is new FLOAT_IO(FLOAT);

   with TEXT_IO; use TEXT_IO;
   procedure COLORS is

     type ENGLISH_COLORS is
        (WHITE, BLACK, RED, PURPLE, BLUE, GREEN, YELLOW, ORANGE);

     type FRENCH_COLORS is
        (BLANC, NOIR, ROUGE, POURPRE, BLEU, VERT, JAUNE, ORANGE);

     package ENGLISH_COLOR_IO is new ENUMERATION_IO (ENUM =>
ENGLISH_COLORS);
     package FRENCH_COLOR_IO is new ENUMERATION_IO (ENUM => FRENCH_COLORS);

     ENG_COLOR : ENGLISH_COLORS;
     FR_COLOR  : FRENCH_COLORS;
     POSITION  : NATURAL;

   begin
     PUT (ITEM => "PLEASE ENTER AN ENGLISH COLOR > ");
     ENGLISH_COLOR_IO.GET (ITEM => ENG_COLOR);
     POSITION := ENGLISH_COLORS'POS(ENG_COLOR);
     FR_COLOR  := FRENCH_COLORS'VAL(POSITION);
     PUT (ITEM => "THE FRENCH COLOR IS ");
     FRENCH_COLOR_IO.PUT (ITEM => FR_COLOR, SET => TEXT_IO.LOWER_CASE);
     NEW_LINE;
   end COLORS;
```

3.11.3.2 Valores y objetos genéricos

Ada también permite la definición de valores y objetos como parámetros formales genéricos. La declaración de estos parámetros es similar a la declaración de variables, con la adición de la palabra reservada **in**para un valor, o **in out** para un objeto:

```
   generic
      ROWS: in INTEGER:=24;
      COLUMNS: in INTEGER:=80;
   package TERMINAL is ...
   ...
```

```
package MICRO_TERMINAL is new TERMINAL(24,40);
package NORMAL_TERMINAL is new TERMINAL;
```

3.11.4 Subprogramas como parámetros

Un parámetro genérico puede ser también un subprograma.

Ejemplo:

```
-- procedure specification for GenericSwapSort
GENERIC -- procedure specification for GenericSwapSort

  -- here are all the generic formal parameters
  TYPE ElementType IS PRIVATE;  -- any nonlimited type will do
  TYPE IndexType   IS (<>);     -- any discrete type for index
  TYPE ListType    IS ARRAY (IndexType RANGE <>) OF ElementType;
  WITH FUNCTION Compare (Left, Right : ElementType) RETURN Boolean;
PROCEDURE GenericSwapSort(List: IN OUT ListType);

-- procedure body for GenericSwapSort
PROCEDURE GenericSwapSort(List: IN OUT ListType) IS
E : ElementType;

BEGIN -- GenericSwapSort

  FOR PositionToFill IN List'First..List'Last LOOP
    FOR ItemToCompare IN PositionToFill..List'Last LOOP
      IF Compare(List(ItemToCompare),  List(PositionToFill)) THEN
        E:=List(PositionToFill);
        List(PositionToFill) := List(ItemToCompare);
        List(ItemToCompare) := E;
      END IF;
    END LOOP;
  END LOOP;
END GenericSwapSort;

WITH Text_IO, GenericSwapSort;
use Text_IO;
PROCEDURE TestSort IS

  TYPE    Color IS (Red, Orange, Yellow, Green, Blue, Violet);
  TYPE    CrayonVector IS ARRAY(Color) of Natural;
  package IIO is new Integer_IO(Integer); use IIO;
  package FIO is new Float_IO(Float); use FIO;
  PACKAGE Color_IO IS NEW Text_IO.Enumeration_IO(Enum => Color);
  use Color_IO;
  SUBTYPE Index IS Integer RANGE 1..10;
  TYPE    FloatVector IS ARRAY(Index RANGE <>) OF Float;
  V1 : FloatVector(1..10);
  V2 : CrayonVector;
  -- local procedures to display the contents of a vector

  PROCEDURE DisplayCrayonVector (V: CrayonVector) IS
  BEGIN
    FOR Count IN V'First..V'Last LOOP
      Put (Item=>Count,    Width=>6);
      Put(Item=>V(Count), Width=>4);
      New_Line;
    END LOOP;
    New_Line;
```

```
    END DisplayCrayonVector;

    PROCEDURE DisplayFloatVector (V: FloatVector) IS
    BEGIN
      FOR Count IN V'First..V'Last LOOP
        Put(Item=>V(Count), Fore=>4, Aft=>2, Exp=>0);
      END LOOP;
      New_Line;
    END DisplayFloatVector;

    -- two instances of GenericSwapSort for Float vectors;
    -- the first sorts in increasing order, the second in decreasing order

    PROCEDURE SortUpFloat IS NEW GenericSwapSort
      (ElementType => Float,
       IndexType   => Index,
       ListType    => FloatVector,
       Compare     => "<");

    PROCEDURE SortDownFloat IS NEW GenericSwapSort
      (ElementType => Float,
       IndexType   => Index,
       ListType    => FloatVector,
       Compare     => ">");

    -- two instances of GenericSwapSort for Float vectors;
    -- the first sorts in increasing order, the second in decreasing order

    PROCEDURE SortUpCrayon IS NEW GenericSwapSort
      (ElementType => Natural,
       IndexType   => Color,
       ListType    => CrayonVector,
       Compare     => "<");

    PROCEDURE SortDownCrayon IS NEW GenericSwapSort
      (ElementType => Natural,
       IndexType   => Color,
       ListType    => CrayonVector,
       Compare     => ">");

  BEGIN -- TestSort

    V1 := (0.7, 1.5, 6.9, -3.2, 0.0, 5.1, 2.0, 7.3, 2.2, -5.9);
    New_Line;
    Put_Line(Item=> "Testing GenericSwapSort for float vectors");
    Put_Line(Item=> "Here is the vector before sorting.");
    DisplayFloatVector(V => V1);
    New_Line;
    SortUpFloat(List => V1);
    Put_Line(Item=> "Here is the vector after upward sorting.");
    DisplayFloatVector(V => V1);
    New_Line;

    SortDownFloat(List => V1);
    Put_Line(Item=> "Here is the vector after downward sorting.");
    DisplayFloatVector(V => V1);
    New_Line;

    V2 := (0, 100, 23, 27, 15, 94);
    New_Line;
```

```
  Put_Line(Item=> "Testing GenericSwapSort for crayon vectors");

  Put_Line(Item=> "Here is the vector before sorting.");
  DisplayCrayonVector(V => V2);
  New_Line;

  SortUpCrayon(List => V2);
  Put_Line(Item=> "Here is the vector after upward sorting.");
  DisplayCrayonVector(V => V2);
  New_Line;

  SortDownCrayon(List => V2);
  Put_Line(Item=> "Here is the vector after downward sorting.");
  DisplayCrayonVector(V => V2);
END TestSort;
```

Ejemplo: colas genéricas

```
generic
   type ITEM is private;
package GCOLAS is
   type QUEUE is limited private;
   procedure PONER(Q:in out QUEUE; X: in ITEM);
   procedure QUITAR(Q:in out QUEUE; X: out ITEM);
   function VACIA(Q:QUEUE) return BOOLEAN;
   function LONG(Q:QUEUE) return INTEGER;

private
   type CELL;
   type LINK is access CELL;
   type CELL is
     record
        VALOR: ITEM;
        NEXT: LINK;
     end record;
   type QUEUE is
     record
        CAB,FIN:LINK;
        CONT:INTEGER:=0;
     end record;
end GCOLAS;

with Text_IO; use Text_IO;
package body GCOLAS is

   procedure PONER(Q:in out QUEUE; X: in ITEM) is
      L:LINK;
   begin
      L:=new CELL'(X,null);
      if Q.CONT=0 then
         Q.CAB:=L;
         Q.FIN:=L;
      else
         Q.FIN.NEXT:=L;
         Q.FIN:=L;
      end if;
      Q.CONT:=Q.CONT+1;
   end PONER;

   procedure QUITAR(Q:in out QUEUE; X: out ITEM) is
   begin
```

```
         if Q.CONT=0 then
            PUT_LINE("COLA VACIA");
            return;
         end if;
         X:=Q.CAB.VALOR;
         Q.CAB:=Q.CAB.NEXT;
         Q.CONT:=Q.CONT-1;
      end QUITAR;

      function VACIA(Q:QUEUE) return BOOLEAN is
      begin
         return Q.CONT=0;
      end VACIA;

      function LONG(Q:QUEUE) return INTEGER is
      begin
         return Q.CONT;
      end LONG;
   end GCOLAS;

   with Text_IO; use Text_IO;
   with GCOLAS;

   procedure TESTGCOL is

   package INT_COLA is new GCOLAS(INTEGER); use INT_COLA;
   package IIO is new Integer_IO(INTEGER); use IIO;
   Q: QUEUE;
   N: INTEGER;

   begin
      for I in 1..10 loop
         PUT("NUMERO ENTERO: ");
         GET(N);
         PONER(Q,N);
      end loop;
      NEW_LINE;
      PUT("LONGITUD DE LA COLA: ");
      PUT(LONG(Q));
      NEW_LINE;
      for I in 1..5 loop
         QUITAR(Q,N);
         PUT("ELEMENTO ");
         PUT(I);
         PUT(" EXTRAIDO = ");
         PUT(N);
         NEW_LINE;
      end loop;
      PUT("LONGITUD DE LA COLA: ");
      PUT(LONG(Q));
   end TESTGCOL;
```

3.12 Programación orientada a objetos (OOP) en Ada95

Vamos a presentar en este capítulo una breve descripción de los mecanismos que Ada95 proporciona para la programación orientada a objetos. Comenzaremos con un repaso informal de la terminología orientada a objetos (OO), y pasaremos luego a analizar la implementación en Ada95 de los principales aspectos de la OOP.

3.12.1 Terminología OO

- Objeto.- Entidad que tiene una estructura interna (datos) a la que se accede mediante mediante el envío de mensajes.
- Mensaje.- Petición de que el objeto ejecute uno de sus métodos.
- Método.- Conjunto de acciones que acceden o manipulan la estructura del objeto. El detalle de estas acciones está oculto para el usuario del objeto.
- Encapsulamiento.- Agrupación de datos y métodos en una unidad, de forma que el acceso a los datos sólo se puede realizar mediante mensajes, o llamadas a los métodos.
- Clase.- Nombre colectivo de todos los objetos que comparten estructura y comportamiento.
- Extensión de tipos.- Capacidad para definir un tipo como una extensión de otro (tipo padre).
- Herencia.- Capacidad de un tipo para heredar las operaciones primitivas (métodos) de su tipo padre, para modificar dichas operaciones y para añadir otras nuevas.
- Polimorfismo.- Capacidad para distinguir, en tiempo de ejecución, el tipo específico de un objeto de otros tipos relacionados, y en particular capacidad para seleccionar una operación según el tipo específico.

En Ada, una clase (según la terminología OO) puede ser representada con la ayuda de un package. La especificación del package define la estructura y los mensajes que pueden ser enviados a un objeto de la estructura dada. Si la estructura se define como private, se asegura el encapsulamiento.

Ejemplo:

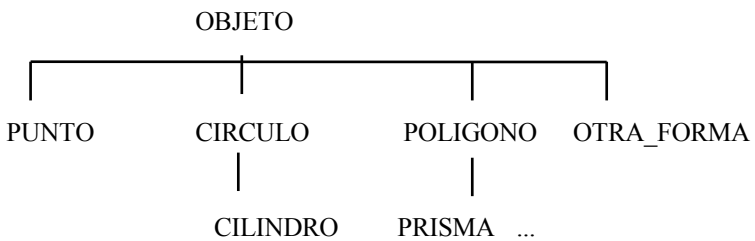
```
-- package utilizado en el tema de Concurrencia

package BUFFERS is
  type BUFFER is private;
  procedure APPEND(D:in Integer; B: in out BUFFER);
  procedure TAKE(D:out Integer; B: in out BUFFER);
  -- APPEND y TAKE son los mensajes que pueden ser enviados a un objeto BUFFER

private
  ...
end BUFFERS;
  ...
  B1: BUFFER;   -- objeto
```

El cuerpo del package contendrá el código de los métodos.

3.12.2 Herencia y extensión de tipos

Estas facultades se consiguen en Ada95 con los tipos **tagged record**.

Supongamos que queremos manipular varias formas de objetos geométricos, según la siguiente jerarquía



Todos los objetos tendrán una posición dada por sus coordenadas X e Y. Podemos declarar la raiz de la jerarquía como:

```
type OBJETO is tagged
record
   X_COORD: Float;
   Y_COORD: Float;
end record;
```

La palabra reservada tagged indica que este tipo puede ser extendido. Los demás objetos geométricos derivarán de este tipo OBJETO.

Por ejemplo, podemos tener:

```
type CIRCULO is new OBJETO with
record
   RADIO: Float;
end record;
```

CIRCULO tendrá tres componentes: X_COORD e Y_COORD, heredados, y RADIO, añadido explícitamente.

A veces es conveniente derivar un tipo nuevo sin añadir nuevos componentes:

```
type PUNTO is new OBJETO with null record;
```

También podemos hacer que un tipo private sea tagged:

```
type NOMBRE is tagged private;
-- permite extensión, pero no conocer sus detalles
...


private
   type NOMBRE is tagged
   record
      ...
   end record;
```

y hacer que un tipo derive de otro y mantenga ocultos los componentes adicionales:

```
type OTRA_FORMA is new OBJETO with private;
...
private
   type OTRA_FORMA is new OBJETO with
   record
      -- resto de componentes
   end record;
```

Las operaciones primitivas de un tipo son aquellas declaradas en la misma especificación de package que el tipo, y que tienen parámetros o retornan resultados de ese tipo. Los tipos derivados heredan dichas operaciones.

Podemos tener el siguiente package:

```
package OBJETOS is
      type OBJETO is tagged
      record
         X_COORD: Float;
         Y_COORD: Float;
      end record;
      function DISTANCIA(O: in OBJETO) return Float;
      function AREA(O: in OBJETO) return Float;
      -- el tipo OBJETO no tiene área, pero es interesante que la función
      -- esté definida para que pueda ser heredada por sus descendientes
end record;

package body OBJETOS is

      function DISTANCIA(O: in OBJETO) return Float is
      begin
           return SQRT(O.X_COORD**2+O.Y_COORD**2); -- distancia al origen
      end DISTANCIA;
      function AREA(O: in OBJETO) return Float is
      begin
           return 0.0;
      end AREA;
end OBJETOS;
```

El tipo CIRCULO heredará ambas funciones. Como la función AREA heredada no es apropiada, declararemos:

```
function AREA(O: in CIRCULO) return Float is
begin
     return PI*O.RADIO**2;
end AREA;
```

La implementación de una función heredada puede ser modificada, como vemos.

```
with OBJETOS; use OBJETOS;
package FIGURAS is
      type PUNTO is new OBJETO with null record;
      type CIRCULO is new OBJETO with
      record
         RADIO: Float;
      end record;
      function AREA(O: in CIRCULO) return Float;
      type CILINDRO is new CIRCULO with
      record
         ALTURA: Float;
      end record;
      function AREA(C: in CILINDRO) return Float;
      type OTRA_FORMA is new OBJETO with private;
      function AREA(F: in FORMA) return Float;
private
      type OTRA_FORMA is new OBJETO with
      record
         ...
      end record;
end FIGURAS;
```

La conversión de tipos es permitida hacia la raiz de la jerarquía. Podemos escribir:

```
      O: OBJETO := (1.0,0.5);
      C: CIRCULO := (0.0,1.0,5.5);
      CIL: CILINDRO := (0.0,0.0,6.5,10.0);
O2: OBJETO := OBJETO(CIL);        -- O2:=(0.0,0.0)
      C:=CIRCULO(CIL);                 -- C:=(0.0,0.0,6.5);
```

La función AREA de CILINDRO podrá tener el código:

```
      function AREA(C: in CILINDRO) return FLOAT is
      begin
         return 2.0*AREA(CIRCULO(C))+2.0*PI*C.RADIO*C.ALTURA;
      end AREA;
```

También podemos dar un valor a un descendiente por extensión:

```
      C:= (O with 4.2);                      -- C:=(1.0,0.5,4.2)
      CIL := (C with 7.0);             -- CIL:=(1.0,0.5,4.2,7.0)
      P: PUNTO :=(O with null record); -- P:=(1.0,0.5)

      F1,F2: Float;

      F1 := AREA(P);                   -- F1:= 0.0
      F1 := AREA(C);                   -- F1:= PI*4.2**2
      F2 := AREA(PUNTO(C));            -- F2:= 0.0
```

Ejemplo:

Dado el package siguiente:

```
      package RESERVAS_AVION is
         type RESERVA is tagged private;
         procedure HACER_RESERVA(R: in out RESERVA);

      private
         type FECHA is
         record
            D,M,A: Positive;
         end record;
         type RESERVA is taggged
         record
            NUM_VUELO: Integer;
            FECHA_VUELO: FECHA;
            NUM_ASIENTO: String(1..3):="   ";
         end record;

      end RESERVAS_AVION;
```

a) Escribir un cuerpo apropiado para el package

b) Definir un nuevo package que incluya dos extensiones del tipo RESERVA:

-RESERVA_BASICA, que no añade ningún componente

-RESERVA_CLASE1, que añade el componente MENU, que puede tomar como valor VEGETARIANO, CARNE o PESCADO, y el componente ASIENTO, que puede tomar como valor PASILLO o VENTANA.

Escribir un nuevo procedure HACER_RESERVA para RESERVA_CLASE1, que utilice el procedure heredado.

3.12.3 Polimorfismo

Hemos visto en el apartado anterior cómo declarar una jerarquía de tipos derivados de un tipo OBJETO. Todos los tipos derivados de un ascendiente común heredan los componentes y las operaciones del mismo. En el caso de OBJETO, todos sus descendientes heredan X_COORD, Y_COORD y las funciones AREA y DISTANCIA. Parece normal entonces que podamos utilizar cualquiera de estas operaciones para un elemento de un tipo de la jerarquía sin saber de antemano de qué tipo se trata. Esto implica que el tipo de un objeto al que se manda un mensaje no es conocido en tiempo de compilación.

Asociado a cada tipo tagged T hay un tipo T'CLASS que comprende todos los tipos de la familia a partir de T. Por ejemplo, OBJETO'CLASS incluirá los siguientes tipos: OBJETO, PUNTO, OTRA_FORMA, CIRCULO, CILINDRO. CIRCULO'CLASS incluye CIRCULO y CILINDRO. Todas las operaciones de OBJETO'CLASS serán aplicables a CIRCULO'CLASS, pero no viceversa.

Una dificultad que presenta la flexibilidad dada por estos tipos polimórficos (tipos ***wide class***) es que no podemos saber cuánto espacio va a ocupar un objeto del tipo, puesto que puede ser un objeto extendido. Así, aunque un objeto pueda ser declarado de un tipo polimórfico (X:NOMBRE'CLASS) debe ser inicializado con un valor de un tipo especificado perteneciente a la clase dada:

```
C: CIRCULO := (0.0, 0.0, 5.0);
X: OBJETO'CLASS := C;
```

Parece por tanto que esta declaración no es muy útil. Sin embargo, un parámetro formal SI puede ser un tipo polimórfico, y el parámetro real puede ser de cualquier tipo de la clase correspondiente.

```
function COSTE_PINTURA(O:OBJETO'CLASS; P: Float) return Float is
begin
   return P*AREA(O);
end COSTE_PINTURA;
```

La función anterior calcula el coste de pintar la superficie de un objeto, siendo P el precio por unidad de superficie. O puede ser de tipo OBJETO, o de cualquier tipo derivado de OBJETO. La función AREA que se va a utilizar no es conocida hasta que el programa se ejecute.

Además de ahorrarnos la duplicación de código:

```
function COSTE_PINTURA(O:CIRCULO;P.Float)...
function COSTE_PINTURA(O:CILINDRO;P:Float)...
...
```

nos permite añadir en el futuro otros tipos descedientes de OBJETO sin necesidad de recompilar el sistema existente.

Se puede chequear si un objeto está dentro de una clase:

```
function PRUEBA(H:OBJETO'CLASS) return ... is
   ...
   if H in CILINDRO'CLASS then
      ...
   elsif H in CIRCULO'CLASS then
      ...
```

```
   else
      ...
   endif;
   ...
```

Por ejemplo, el procedure HACER_RESERVA del package RESERVAS_AVION, utilizado anteriormente, podría reescribirse como:

```
procedure HACER_RESERVA(R: in out RESERVA'CLASS);
```

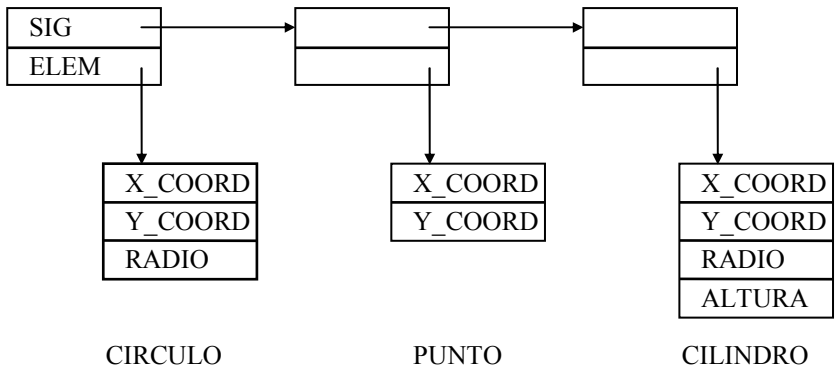
Este procedure podrá ser utilizado por cualquier objeto de tipo RESERVA o de un tipo descendiente de RESERVA:

```
procedure HACER_RESERVA(R: in out RESERVA'CLASS) is
begin
    -- leer R.FECHA_VUELO y R.NUM_VUELO
    if R in RESERVA_CLASE1'CLASS then
       -- leer R.MENU y R.ASIENTO
       -- asignar valor a R.NUM_ASIENTO, dependiendo de R.ASIENTO
    else
       -- leer R.NUM_ASIENTO
    end if;
end HACER_RESERVA;
```

Otra restricción de los tipos *wide class* consiste en que no podemos tener un array de componentes de uno de estos tipos, puesto que los componentes pueden ser de distintos tamaños, haciendo imposible una indexación eficiente.

Como consecuencia de estas restricciones, es muy habitual utilizar tipos access en OOP, puesto que no hay problemas en que un puntero refiera a objetos de diferentes tamaños.

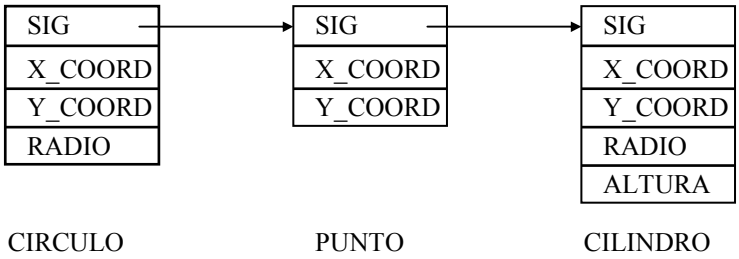
```
type PUNT is access all OBJETO'CLASS;
type CELDA;
type PUNT2 is access CELDA;
type CELDA is
record
   ELEM: PUNT;
   SIG: PUNT2;
end record;
```



Se pueden también construir listas de objetos polimórficos incluyendo el puntero dentro del objeto:

```
type ELEMENTO;
type PUNT_ELEM is access all ELEMENTO'CLASS;
type ELEMENTO is tagged
record
   SIG: PUNT_ELEM;
end record;

type OBJETO is new ELEMENTO with
record
```

```
   X_COORD, Y_COORD: Float;
end record;
```



EJERCICIO

El siguiente package proporciona operaciones estandar en el manejo de objetos polimórficos. Escribir el cuerpo del package.

```
package LISTAS is
   type LISTA is limited private;
   type ELEMENTO is tagged private;
   type PUNT_ELEM is access all ELEMENTO'CLASS;
   procedure PON(L: in out LISTA; E: in PUNT_ELEM);
   procedure SACA(L: in out LISTA; E: out PUNT_ELEM);
   function LONG(L: LISTA) return Integer;
private
   type ELEMENTO is tagged
   record
      SIG: PUNT_ELEM;
   end record;
   type LISTA is
   record
      CONT: Integer := 0;
      CAB: PUNT_ELEM;
   end record;
end LISTAS;
```

3.12.4 Tipos abstractos

Según lo visto hasta ahora, una jerarquía de tipos dispone de un tipo raiz que incluye campos de datos. Por ejemplo, el tipo OBJETO utilizado incluye los campos X_COORD e Y_COORD, y las operaciones AREA y DISTANCIA. Todos los objetos del tipo OBJETO'CLASS heredan dichos campos y operaciones.

A veces es necesario o conveniente definir un tipo raiz no con la idea de declarar objetos de ese tipo, sino con el fin de que sus descendientes hereden determinadas propiedades. Esto puede hacerse con la ayuda de los llamados **tipos abstractos**. No se pueden declarar objetos de un tipo abstracto. Un tipo abstracto puede incluir subprogramas abstractos. Estos subprogramas no tienen cuerpo, y por tanto no pueden ser llamados directamente. Cada tipo derivado del tipo abstracto podrá asignarle el cuerpo apropiado.

EJEMPLO. Vamos a reescribir el ejemplo de reservas de avión.

```
package RESERVAS_AVION is
   type RESERVA is abstract tagged null record;
   -- tipo abstracto sin componentes
   type PUNT_RESERVA is access all RESERVA'CLASS;
   procedure HACER_RESERVA(R: in out RESERVA) is abstract;
end RESERVAS_AVION;
```

Este package no tendrá cuerpo. Podemos desarrollar toda la 'infraestructura' de reservas y añadir luego los tipos normales de reservas, como herederos del anterior:

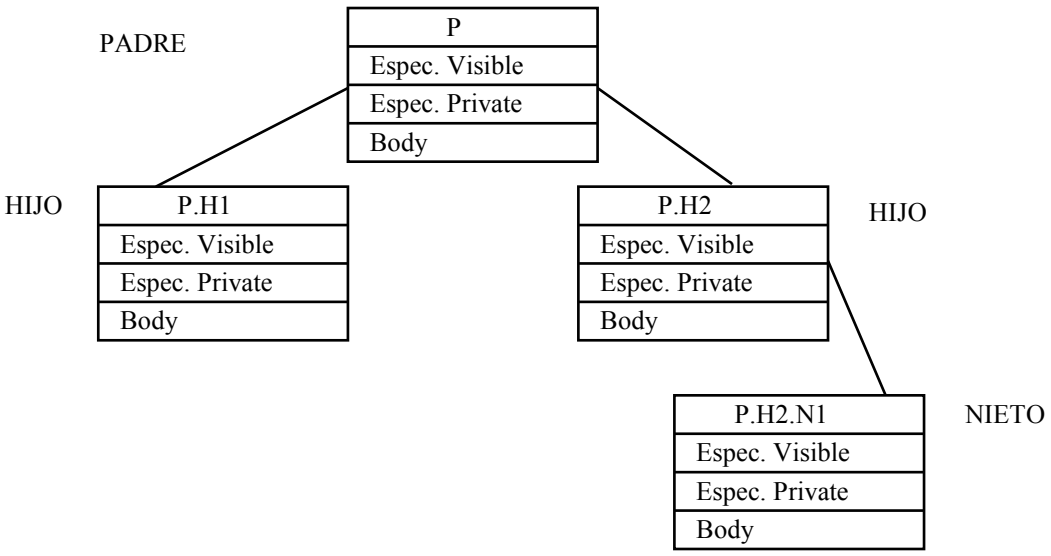
```
with RESERVAS_AVION; use RESERVAS_AVION;
package NUEVAS is
   type RESERVA_BASICA is new RESERVA with private;
   type RESERVA_CLASE1 is new RESERVA_BASICA with private;
```

```
      procedure HACER_RESERVA(R: in out RESERVA_BASICA);
      procedure HACER_RESERVA(R: in out RESERVA_CLASE1);
   private
      type FECHA is
      record
         D,M,A: Positive;
      end record;
      type POSICION is (PASILLO, VENTANA);
      type COMIDA is (VEGETARIANO, CARNE, PESCADO);
      type RESERVA_BASICA is new RESERVA with
      record
         NUM_VUELO: Integer;
         FECHA_VUELO: FECHA;
         NUM_ASIENTO: String(1..3):="   ";
      end record;
      type RESERVA_CLASE1 is new RESERVA_BASICA with
      record
         MENU: COMIDA;
         ASIENTO: POSICION;
      end record;
   end NUEVAS;
```

3.12.5 Jerarquía de librerías

Una librería hija (*child library*) es una forma de añadir código a un package existente sin cambiar el package original. El uso de librerías hijas permite dividir un package grande en componentes más manejables, o extender un package existente sin necesidad de recompilarlo.

Reglas de visibilidad




```
package P.H1 is                    package body P.H1 is
      -- Espec. Visible                -- Body
   private                         end P.H1;
      -- Espec. Private
   end P.H1;
```

Puede acceder a →	P	P.H1	P.H2	P.H2.N1
P.H1 Espec. Visible	Espec. Visible (No necesita clausula with)	•	Espec. Visible si utiliza with P.H2	Espec. Visible si utiliza with P.H2.N1

P.H1 Espec. Private o Body	Espec. Visible Espec. Private	•	Espec. Visible si utiliza with P.H2	Espec. Visible si utiliza with P.H2.N1
P.H2 Espec. Visible	Espec. Visible (No necesita clausula with)	Espec. Visible si utiliza with P.H1	•	No es utilizable
P.H2 Espec. Private o Body	Espec. Visible Espec. Private	Espec. Visible si utiliza with P.H1	•	Espec. visible en el Body de P.H2 si utiliza with P.H2.N1
P.H2.N1 Espec. Visible	Espec. Visible (No necesita clausula with)	Espec. Visible si utiliza with P.H1	Espec. Visible (No necesita clausula with)	•
P.H2.N1 Espec. Private o Body	Espec. Visible Espec. Private	Espec. Visible si utiliza with P.H1	Espec. Visible Espec. Private	•

Unidades hijas private

Un package hijo private (declaración: **private** package NOMBRE is ...) es como un package hijo normal, con dos reglas de visibilidad extra:

- Sólo es visible dentro del subárbol cuya raiz es su padre. Dentro de ese subárbol, no es visible para la especificación de los packages hermanos, o descendientes de los hermanos, aunque sí es visible para los cuerpos de los mismos.

- La parte visible del package private tiene acceso a la parte private de su padre.

Si P.H2 fuera private,

- El cuerpo de P.H1 podría acceder a la espec. visible de P.H2 con la clausula with P.H2.
- Para P.H2.N1 no habría diferencia entre que sea o no private P.H2.
- La espec. visible de P.H2 podría acceder a la espec. private de P.
- Ninguna otra unidad, fuera del subárbol que comienza en P, podría utilizar P.H2.

BIBLIOGRAFÍA

[1] Barnes,J.; 'Programming in Ada95'; ed. Addison-Wesley (1996)

[2] Ada Reference Manual; Intermetrics, Inc. (1995)

[3] Feldman, M.; Ada95: Problem Solving and Program Design; Addison-Wesley (1996)

[4] Booch, G.; Bryan, D.; Software Engineering with Ada; Addison-Wesley (1994)

[5] Burns,A., Wellings,A.; 'Concurrency in Ada'; ed. Cambridge University Press (1995); cap 13

[6] Smith,M.A.; 'Object Oriented Software in Ada95'; Internat. Thomson Computer Press (1996)