GUIA DE REFERENCIA BASICA
Ada 2005

C treats you like a consenting adult,
Pascal treats you like a naughty child,
Ada treats you like a criminal

Bárbara Álvarez Torres
Diego Alonso Cáceres

Edita: Universidad Politécnica de Cartagena
Primera Edición, Enero 2008

ISBN : 978-84-95781-92-5

TABLA DE CONTENIDOS DEL CURSO

1. Introducción a Ada 2005 1
2. Elementos Léxicos Básicos 5
   2.1. Elementos léxicos, separadores y delimitadores 5
   2.2. Comentarios 5
   2.3. Identificadores 5
   2.4. Palabras reservadas 6
   2.5. Literales 6
3. Estructura de un Programa. Unidades 9
   3.1. Unidades de programa, compilación y librería 9
   3.2. Bloques 11
   3.3. Variables y constantes 12
   3.4. Declaraciones. Reglas de ámbito y visibilidad 12
4. Tipos de Datos: Clasificación 15
   4.1. Tipos enumerados 16
      4.1.1. Tipo carácter 17
      4.1.2. Tipo boolean 17
   4.2. Tipos enteros 18
   4.3. Tipos reales 20
   4.4. Algunos atributos aplicables a los tipos escalares 21
   4.5. Tipos derivados y subtipos 24
   4.6. Compatibilidad y conversión de tipos 25
5. Expresiones y Operadores 27
   5.1. Operadores lógicos 27
      5.1.1. Operadores lógicos sobre tipos modulares 28
      5.1.2. Operadores relacionales 28
   5.2. Operadores aritméticos 29
   5.3. Sobrecarga de operadores 30
6. Sentencias de Control 31
   6.1. Selección entre alternativas lógicas 31
   6.2. Múltiples alternativas en tipos enumerados 32
   6.3. Repetición controlada por contador 33
   6.4. Repetición controlada por condición lógica 34
7. Tipos Estructurados 37
   7.1. Estructuras homogéneas. Arrays 37
      7.1.1. Literales array 38
      7.1.2. "Trozos" de array 39
      7.1.3. Atributos aplicables a los arrays 39
      7.1.4. Arrays no restringidos 39
      7.1.5. El tipo predefinido String 40
   7.2. Estructuras heterogéneas. Records 40
      7.2.1. Literales record 41
      7.2.2. Record variante 42

8. Subprogramas 43
8.1. Definición, declaración y uso 43
8.1.1. Declaración de un subprograma 46
8.1.2. Definición separada (separate) 46
8.2. Parámetros de los subprogramas 47
8.2.1. Clases de parámetros 47
8.2.2. Correspondencia entre parámetros reales y formales 47
8.2.3. Parámetros por omisión 48
9. Paquetes 49
9.1. Tipos privados y limitados 49
9.2. Definición y uso 49
9.2.1. Especificación 50
9.2.2. Cuerpo 51
9.3. Herencia de paquetes 53
10. Unidades Genéricas 55
10.1. Parámetros formales genéricos 55
10.2. Instanciación de unidades genéricas 56
10.3. Subprogramas genéricos 56
10.4. Paquetes genéricos 58
11. Excepciones 63
11.1. Declaración 63
11.2. Lanzamiento 64
11.3. Manejo 64
12. Concurrencia en Ada 67
12.1. Tiempo real y tareas periódicas 70
12.2. Terminación de tareas 72
13. Comunicación entre Tareas 75
13.1. Comunicación síncrona: cita extendida 75
13.1.1. Espera selectiva en el lado del servidor 77
13.1.2. Espera selectiva en el lado del cliente 79
13.1.3. Familia de entrys 80
13.1.4. Atributos de las tareas 81
13.2. Comunicación asíncrona: tipos protegidos 81
13.2.1. Entradas protegidas y sincronización condicional 82
13.3. Acceso atómico a variables 83

1. INTRODUCCIÓN A ADA 2005

Durante los años 1960 y 1970, el Departamento de Defensa de los Estados Unidos (*DoD*) estaba usando más de 450 lenguajes de programación en sus sistemas. Muchos de éstos eran lenguajes que habían sido diseñados específicamente para un proyecto en concreto. Pronto se encontraron con cientos de líneas de código escritas en lenguajes que ya casi nadie sabía o que se habían olvidado. A mediados de 1970 el *DoD* decidió poner fin a esta "crisis del software". Sacaron un concurso para adoptar el diseño de un lenguaje que pudieran usar para todo: tanto para el desarrollo algorítmico como para sistemas con requisitos de tiempo-real. Así nació el lenguaje Ada, que debe su nombre a Lady Ada Lovelace, hija del poeta Lord Byron y ayudante del matemático Charles Babbage, que inventó la máquina analítica.

Hasta el momento existen tres versiones de Ada: *A da 83* (ISO 8652:1987), *A da 95* (ISO 8652:1995) y *A da 2005* (ISO/ IEC 8526:AMD1:2007). Estas normas definen un núcleo común para todas las implementaciones y unos anexos especializados (obligatorios u optativos) para programación de sistemas, sistemas de tiempo real, sistemas distribuidos, etc. Dichos anexos definen paquetes de biblioteca y mecanismos de implementación, pero no añaden sintaxis ni vocabulario al lenguaje. Además estos anexos dejan la puerta abierta a compiladores con más o menos características, pero manteniendo común el núcleo del lenguaje.

Ada es un lenguaje de programación de propósito general avanzado y moderno, quizá un poco adelantado a su época (muchas de sus características ha sido adoptadas por lenguajes más extendidos y tardíos, como C++ y Java). Ada ha sido el primer lenguaje estandarizado por un comité internacional, con compiladores certificados y validados, y el primero diseñado para fomentar el uso de principios de diseño software ampliamente reconocidos. Gracias a las características que se exponen a continuación, Ada reduce el coste de desarrollo, verificación, depuración y mantenimiento durante el ciclo de vida del producto software.

- **Portable**: el código desarrollado en Ada es portable entre sistemas, ya que existen una gran cantidad de compiladores certificados y Ada es el primer lenguaje internacionalmente validado.
- **Modular**: la forma de organizar el código en Ada permite que éste pueda ser planificado, desarrollado, compilado y probado por separado, con lo que se favorece el desarrollo en equipo y la obtención del producto final.
- **Reutilizable**: el concepto de *paquete* Ada permite desarrollar código que puede ser utilizado y cambiado si afectar al resto del programa. A su vez, mediante el uso de *genéricos* se pueden desarrollar algoritmos que trabajen sobre muy diferentes tipos de datos.
- **Fiable**: la característica de *tipado fuerte* de Ada permite detectar rápidamente errores. El mecanismo de *excepciones* facilita el desarrollo de aplicaciones tolerantes a fallos, que pueden reaccionar ante errores. Además, las características *multitarea* del lenguaje evitan tener que usar directamente las llamadas al sistema operativo para soportar concurrencia.

- **Mantenible**: la estructuración del código así como su facilidad de lectura (a costa de escribir más) facilitan la tarea de revisión y mejora del código. Gracias a la modularidad se pueden actualizar paquetes sin afectar a otras partes del programa.

Entre los sistemas que utilizan Ada destacan:

- Exploración geofísica, procesamiento de datos y análisis químico.
- Sistemas bancarios de pago, comunicación entre bancos, sistemas de control de stock.
- Centralitas de control de teléfonos móviles y aplicaciones de telecomunicación.
- Gran cantidad de sistemas de aviónica: control de tráfico aéreo, detección y guía de aviones, simuladores, etc.
- Gran cantidad de los sistemas desarrollados por la NASA para los transbordadores y estaciones espaciales: robótica, sistemas médicos, sistemas de control, etc.
- Sistemas militares estratégicos.

En resumen, Ada se ha diseñado con las siguientes metas en mente:

- Ingeniería del software: programación a gran escala (*paquetes*).
- Legibilidad en vez de rápida escritura.
- Estándar riguroso para que sea portable.
- Semántica precisa.
- Diseñado para ser concurrente.
- Tipado fuerte. Una naranja no es una manzana. No depures: compila.
- Manejo de excepciones: por si Murphy.
- Orientación a objetos: herencia y polimorfismo.
- Genéricos: la forma más sencilla de reutilización.
- Buena compatibilidad con otros lenguajes: C, Fortran y Cobol.

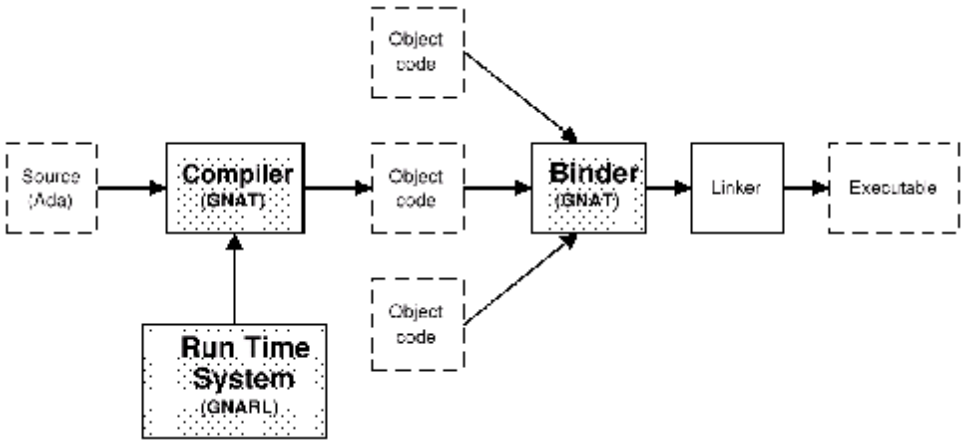
Otro aspecto sorprendente de Ada es que el compilador y las librerías (paquetes) que se proporcionan también están desarrollados en Ada y se suministra el código fuente. De esta forma cualquiera puede consultar las funciones disponibles sin más que buscarlas en el código fuente. También es un excelente ejemplo del que aprender cómo programar.

La estructura del sistema de compilación Ada usado, llamado GNAT, está formado por cuatro partes principales (tal y como muestra la figura), que son:

- ***El compilador***: genera el código objeto de la aplicación.
- **El run-time**: capa para hacer portable el código entre sistemas y que proporciona las características de concurrencia y chequeo de tipos, entre otras cosas.
- **El binder**: verifica la consistencia del código y determina el orden de ensamblaje.

- **El linker**: genera el ejecutable final enlazando el código ya compilado (código objeto) del programa con las librerías del sistema, el run-time y otras librerías suministradas por el desarrollador.

El run-time de Ada está formado a su vez por dos capas, *GNARL* y *GNULL*. *GNARL* proporciona una primera capa de abstracción para manejar los aspectos de lenguajes (sobre todo las tareas) independiente de la plataforma de ejecución. La implementación específica para la plataforma la proporciona la capa *GNULL*. De esta forma es posible tener un lenguaje cuyo código fuente es independiente, hasta cierto punto, de la plataforma de ejecución.



Esta guía recoge las características básicas del lenguaje de programación Ada que el alumno necesita para realizar las prácticas de la asignatura y para comprender como el lenguaje aporta soluciones a muchos de los problemas de los sistemas de tiempo real. Se presupone que el alumno conoce algún lenguaje de programación estructurada como por ejemplo el lenguaje C. Para un conocimiento más amplio de los mecanismos que ofrece el lenguaje Ada deberá remitirse al Manual de Referencia del Lenguaje.

2. ELEMENTOS LÉXICOS BÁSICOS

El código fuente (texto) de un programa en Ada se distribuye en un conjunto de unidades de compilación, cada una de las cuales está formada por una secuencia de *elementos léxicos*. Cada elemento léxico está formado por una secuencia de caracteres, que pertenece a uno de estos tipos: un *identificador*, un *delimitador*, una *palabra reservada*, un *literal numérico*, un *literal carácter*, un *literal "porción"* o un *comentario*. Todas las implementaciones de Ada garantizan que una línea puede tener hasta 200 caracteres y, en consecuencia, que se pueden tener elementos léxicos de 200 caracteres. Un línea de código termina siempre con el delimitador punto y coma (;).

2.1. ELEMENTOS LÉXICOS, SEPARADORES Y DELIMITADORES

Entre dos elementos léxicos consecutivos puede haber cualquier número de *separadores*, pero debe haber al menos uno si estos elementos léxicos son identificadores, palabras reservadas o literales numéricos. El papel de separador lo cumple el carácter espacio (salvo dentro de un comentario), el carácter de tabulación (salvo dentro de un comentario) y el final de línea. Al principio y al final de una unidad de compilación puede haber cualquier número de separadores.

Un **delimitador** es uno de los siguientes caracteres:

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

También son delimitadores (compuestos) las siguientes parejas de caracteres:

```
=>  ..  **  :=  /=  >=  <=  <<  >>  <>
```

2.2. COMENTARIOS

Un comentario se puede poner en cualquier punto del programa, empieza con dos guiones (--) y termina con el final de la línea en que se encuentre.

```
--esto es un comentario
end Un_Procedimiento; --esto es otro
```

2.3. IDENTIFICADORES

Los identificadores son nombres que designan **entidades** del programa (variables, constantes, subprogramas, etc). En Ada, un identificador se forma comenzando con una letra y siguiendo con una combinación de letras, dígitos y el carácter guión bajo (_). Ada no diferencia entre letras mayúsculas y minúsculas (piense por qué). Son ejemplos de identificadores:

```
Nombre_Empleado
Raíz_2
Este_Es_Un_Identificador_Muy_Largo_Pero_Correcto
2_Por_4  -- ilegal: empieza por un número
```

Como en el ejemplo, es usual que los identificadores compuestos de varias palabras se formen juntando las palabras con guión bajo y con las letras iniciales en mayúscula.

2.4. PALABRAS RESERVADAS

Las palabras reservadas son aquellas que tienen definido un significado especial en el lenguaje y no pueden usarse para otro propósito (por ejemplo, no sirven como identificadores). La siguiente tabla muestra las 69 palabras reservadas del lenguaje.

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

2.5. LITERALES

Un literal es la representación de un valor mediante alguna notación adecuada. En Ada se distinguen: literales *numéricos*, literales *carácter*, literales *porción* y el literal ***null***. El literal ***null*** cumple una doble función: valor específico de los tipos "**access**" (punteros) y "sentencia vacía".

Los literales numéricos pueden ser enteros o reales. La forma más básica de un literal numérico es una secuencia de dígitos, si es entero, o una secuencia de dígitos que incluye un punto, si es real.

```
12 –– número entero '12'
12.83 –- número real '12.83'
```

Existe además la posibilidad de incluir el carácter guión bajo (_), añadir un exponente (notación científica) que especifica la potencia de '10' o expresar valores en distintas bases.

```
1_987        -- entero 1987
1E3          -- entero 1000
```

```
5.0E-1          -- real 0.5
2# 1111000#     -- entero 120, en base 2
16# 1C#         -- entero 28, en base hexadecimal
```

Los literales numéricos pertenecen a los llamados *tipos universales*, lo que significa que son compatibles con cualquier tipo entero o en coma flotante. Más adelante, cuando se comente que Ada es un *lenguaje fuertemente tipado* se volverá sobre este punto.

Un literal carácter se forma poniendo un carácter entre apóstrofes:

```
'a', 'b', 'c',...
```

Un literal porción (cadena de caracteres o *String*) se forma con una secuencia de caracteres encerrados entre comillas. Dos comillas sin ningún carácter en medio representan la *porción nula*.

```
"ejemplo de cadena de caracteres"
```

3. ESTRUCTURA DE UN PROGRAMA. UNIDADES

3.1. UNIDADES DE PROGRAMA, COMPILACIÓN Y LIBRERÍA

Un programa en Ada se estructura como un conjunto de **unidades de programa,** que se unen entre sí hasta formar el programa final. Estas unidades pueden ser de los siguientes tipos:

1. **Subprograma**, hay dos clases (procedimientos y funciones) y sirven para definir algoritmos. Se estudiarán en el tema 8.
2. **Paquete**, sirve para agrupar unidades de programa relacionadas y hacen posible la encapsulación y reutilización. Se estudiarán en el tema 9.
3. **Tarea**, define acciones que pueden ejecutarse en paralelo con otras. Se estudiarán en el tema 12.
4. **Tipo protegido**, proporciona exclusión mutua cuando se comparten datos entre tareas. Se estudiarán en el tema 13.
5. **Unidad genérica**, son subprogramas o paquetes con parámetros generalizados que definen componentes reusables. Se estudiarán en el tema 10.

Las unidades de programa pueden anidarse unas dentro de otras, pero sólo los *subprogramas*, *paquetes* (especificación e implementación) y *unidades genéricas* pueden ocupar el nivel más externo de anidamiento en el código y formar una **unidad de compilación**. Una unidad de compilación es un componente de un programa que puede compilarse por separado. Se distinguen dos clases de unidades de compilación: las *unidades de librería*, que son componentes independientes que pueden usarse en distintos programas y las *subunidades*, que se encuentran lógicamente incluidas en las anteriores, pero que se compilan a parte. Un *subprograma principal* inicia y gobierna la ejecución de un programa y desde él comienza la ejecución del resto de unidades que lo componen. Generalmente, el subprograma principal es un procedimiento sin parámetros.

Una unidad de compilación típica consta de dos partes: (1) una *dáusula de contexto* que sirve para especificar las unidades de librería que se necesitan (no hace falta si no se necesita ninguna) y (2) una unidad de compilación (subprograma, paquete o unidad genérica). El siguiente ejemplo muestra una unidad de compilación que proporciona un procedimiento sin parámetros llamado "*Hola",* que puede ser utilizado como subprograma principal o como unidad de librería que forma parte de un programa más complejo.

Ejemplo 1: ejecute el siguiente programa

```
with Ada.Text_Io;       --Clausula de contexto
use Ada.Text_Io;

procedure Hola is       --Elemento de libreria (subprograma)
begin
```

```
   Put_Line("Hola mundo");
end Hola;
```

Ejemplo 2: ejecute el siguiente programa

```
wITh aDa.TeXt_IO;USE AdA.tExT_Io;proCedURe HoLa1 IS bEgIN
PuT_LINe("Hola mundo comprimido");eND Hola1;
```

OBSERVE como el correcto uso de las mayúsculas y minúsculas y de la tabulación facilita o complica enormemente tanto la escritura del código como su posterior repaso en busca de errores. No hablemos de qué pasa cuando semanas después volvemos a repasar el código y ¡pretendemos entender algo de lo que hicimos! Por tanto, es conveniente que cuando programe se acostumbre tanto a indentar correctamente el código (dependiendo de su profundidad) como a comentarlo (sí, es muy pesado, pero cuando lo repase le parecerá que no pudo haber invertido el tiempo en nada mejor). En cuanto adopte estas convenciones verá como mejora la legibilidad, la detección de errores y podrá saber qué hacía ese código que escribió hace meses.
También es conveniente que se acostumbre a elegir nombres significativos para variables y subprogramas, aunque suponga escribir más (el *DoD* se dio cuenta de ello hace tiempo). Lo agradecerá cuando repase código. Usar sustantivos para nombrar las variables, verbos para los subprogramas y definir los tipos con el prefijo "T_", los vectores (array) con "AT_", los record con "RT_", las tareas con "TT_", los tipos protegidos con "PT_" y los paquetes con "Pack_".

La cláusula de contexto, si aparece, consta obligatoriamente de la palabra reservada ***with***, que es la que especifica las unidades de librería que se necesitan. Las librerías a usar se separan por comas (,). En el ejemplo, la cláusula ***with*** especifica que se va a necesitar la unidad de librería "*Ada.Text_Io",* que es la que define el procedimiento "*Put_Line",* para escribir un saludo en la salida estándar. También existe el procedimiento "*Put* ", que no provoca salto de línea al final, y "*New_Line*", que provoca un salto de línea (y acepta como parámetro el número de saltos de línea a realizar). Opcionalmente, se puede emplear una cláusula ***use*** para especificar unidades de librería que se van a usar con frecuencia, de manera que no sea necesario indicar el nombre de la unidad cada vez que se use un elemento de la misma.

Ejemplo 3: ejecute el siguiente programa

```
with Ada.Text_Io;    --Clausula de contexto
procedure Hola2 is   --Elemento de libreria (subprograma)
begin
   Ada.Text_Io.Put_Line("Hola mundo mas largo");
end Hola2;
```

La cláusula ***use*** ahorra trabajo y hace que los programas parezcan menos densos. La cláusula ***use*** puede ponerse también en cualquier lugar entre las declaraciones de la

unidad de programa, aunque ello reduce el alcance de su efecto (ver apartado 3.4 "Reglas de Ámbito y Visibilidad"). Observe en el siguiente ejemplo con qué facilidad se puede reutilizar el código ya escrito.

Ejemplo 4: ejecute el siguiente programa

```
with Hola;              --Clausula de contexto
procedure Hola_Hola is  --Elemento de libreria (subprograma)
begin
   Hola;              -- No hacen falta los parentesis
   Hola;
   Hola;
end Hola_Hola;
```

Ejercicio 1: modifique el ejemplo anterior para que utilice el código de Hola y Hola2.

3.2. BLOQUES

Un bloque es una *sentencia compuesta*, formada por una secuencia de sentencias agrupadas entre las palabras delimitadoras ***begin*** y ***end***, con una parte opcional para realizar declaraciones locales (***declare***). Además, los bloques pueden ser identificados con una etiqueta opcional.

```
[etiqueta:]
[declare
  declaraciones locales;]
begin
  sentencias;
end;
```

Un bloque puede ponerse en cualquier sitio donde pueda ponerse una sentencia simple, pero siempre tiene que tener por encima alguna de las unidades de programa. Ejemplos de bloques:

```
--bloque sin declaraciones locales
begin
  Put_Line("Hola");
end;

--bloque con declaraciones locales
declare
  Aux : Integer;  -- variable sólo existe dentro del bloque
begin
  Aux := i;       -- i,j se suponen declaradas en
  i := j;         -- un ámbito más externo
```

```
   j := Aux;
end;
```

3.3. VARIABLES Y CONSTANTES

Una *variable* es un dato con un *nombre*, un *tipo* y un *valor* asociado que puede modificarse libremente durante la ejecución de la parte de un programa en la que es visible (ámbito). Una declaración de variables consta de: (1) una lista de nombres de las variables que se declaran separadas por comas (,) y terminada con dos puntos (:), (2) el tipo común de las variables y, opcionalmente, (3) la asignación de un valor inicial (:=).

```
X: Float;
Y, Z: Integer:= 0;
```

Una *constante* es un dato con un *nombre*, un *tipo* y un *valor* asociado que no puede modificarse una vez definido. Una declaración de constantes es igual que una declaración de variables excepto en que se precede al tipo con la palabra **constant**.

```
C, K : constant Integer := 0;
```

Ada contempla además una clase de constantes que sirve para asociar un nombre para identificar un valor numérico y que no requiere la especificación de un tipo (se prefiere a la declaración anterior).

```
Pi : constant := 3.1416;
```

3.4. DECLARACIONES. REGLAS DE ÁMBITO Y VISIBILIDAD

Una *declaración* asocia un nombre con una entidad del programa. Las declaraciones se escriben en una sección de declaraciones de una unidad de programa, separadas de las sentencias ejecutables de la misma. Todas terminan en punto y coma (;).

Toda declaración tiene un ámbito (conjunto de sentencias) en el que es visible el nombre declarado y se puede usar para tener acceso a la misma. Generalmente, el ámbito de una declaración se extiende hacia delante y hacia adentro: una declaración hecha en un subprograma o en un bloque es visible en todo el resto del mismo subprograma o bloque a partir del punto en que aparece, incluyendo los subprogramas o bloques anidados en él, salvo que incluyan una declaración con el mismo nombre (una declaración oculta cualquier otra del mismo nombre hecha en un bloque más externo). Las variables ocultadas pueden ser accedidas si los bloques han sido nombrados con etiquetas.

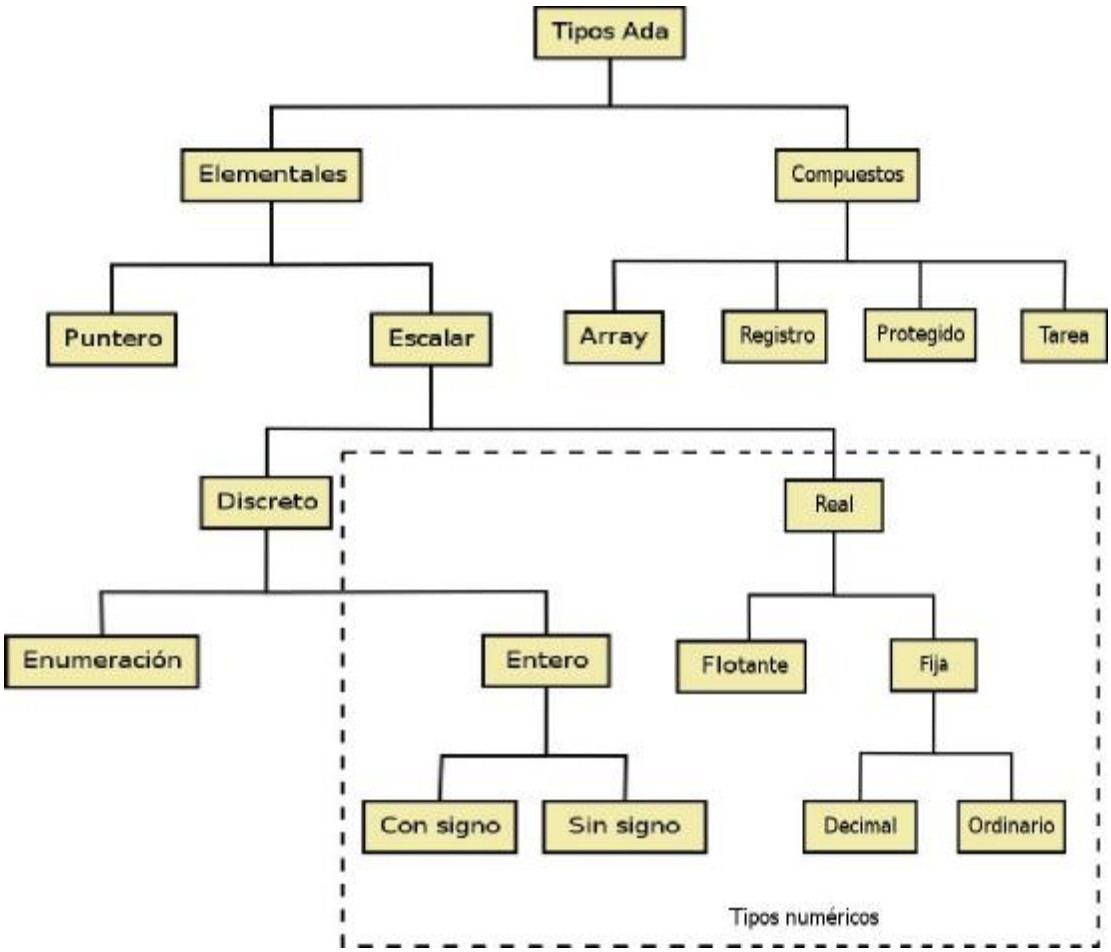
```
Exterior:
declare
   ...
   I,J,K : Float; -- empieza el ámbito de I,J,K tipo Float
begin
   …
   Interior:
   declare
      I,L: Integer; -- empieza el ámbito de I,L tipo Integer
                    -- se oculta I del bloque Exterior
      Put(Exterior.I); -- ahora si se ve
      ...
   begin
      ...
   end; -- fin del ámbito de I,L de tipo Integer

   -- se reanuda el ámbito de I de tipo Float
   ...
end; -- fin del ámbito de I,J,K de tipo Float
```

4. TIPOS DE DATOS: CLASIFICACIÓN

Un tipo de datos es un conjunto de valores *más* un conjunto de operaciones primitivas asociadas a ese conjunto de datos. El lenguaje proporciona tipos predefinidos junto con mecanismos para que el usuario pueda definir nuevos tipos y subtipos. La declaración de un tipo requiere la especificación de un nombre y la definición (descripción) del mismo; la forma general de una declaración de tipo es:

```
type Nombre_Tipo is Definición_Tipo;
```

La siguiente figura muestra una clasificación de los tipos de datos de Ada:



Este primer capítulo sobre tipos se va a centrar únicamente en los tipos elementales del lenguaje: los tipos escalares. Más adelante se verán todos los tipos compuestos.

Como muestra la figura, los tipos escalares se clasifican en: tipos *enumerados*, tipos *enteros* y tipos *reales*. 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 que se representa por un número entero. Los tipos enteros y reales se clasifican también

dentro de los tipos *numéricos*. Todos los tipos escalares admiten los operadores lógicos de relación. Los tipos numéricos admiten, además, diversos operadores aritméticos.

A la hora de representar (elegir el número de bits) tipos *numéricos* en un computador aparecen dos problemas fundamentales: (1) el rango máximo puede estar restringido, debido a la existencia de hardware especializado para realizar las operaciones, con lo que existe un compromiso entre capacidad de representación, espacio ocupado y velocidad de procesamiento; (2) debido a la propia naturaleza de los números (sobre todo los reales) no es posible representar con precisión todos los números existentes. Estas características causan muchos problemas cuando se quiere portar un programa entre máquinas diferentes.

4.1. TIPOS ENUMERADOS

Los tipos enumerados se definen enumerando la lista de valores literales del tipo (un valor literal enumerado puede ser un literal carácter o una secuencia de caracteres válida como identificador).

```
type T_Dias is
    (Lunes,Martes,Miercoles,Jueves,Viernes,Sabado,Domingo);
type T_Vocales is ('a','e','i','o','u');
type T_Color is (Rojo,Azul,Amarillo);
```

El orden y la posición de los valores de un tipo enumerado vienen determinados por el orden en que se enumeran (al primer valor de la lista le corresponde el valor cero al utilizar el atributo Pos, consultar tabla 2, apartado 4.4). Tipos enumerados distintos pueden definirse con literales iguales. Cuando el uso de estos literales pueda resultar ambiguo se deberán cualificar con el nombre del tipo, como se muestra en el siguiente ejemplo, utilizando el carácter apóstrofe tras el nombre del tipo:

```
type T_Semaforo is (Rojo,Amarillo,Verde);
A : T_Color := Rojo;
A in T_Color'(Rojo)..T_Color'(Amarillo) --cualificacion[1]
```

Se pueden definir subtipos de un tipo enumerado especificando un *rango* de valores, utilizando para ello la palabra reservada ".." (dos puntos seguidos).

```
subtype T_Laborables is T_Dias range Lunes . .Viernes;
```

Por último, la entrada/ salida de valores de un tipo enumerado se realiza con el paquete genérico (consultar capítulo 10) *Enumeration_IO,* incluido dentro del paquete *Ada.Text_IO*.

[1] El operador **in** es de pertenencia a un rango de valores (se verá en 5.1.2 "Operadores relacionales").

Ejemplo 5: ejecute el siguiente programa

```
with Ada.Text_IO;

procedure Dias_De_La_Semana is
  type T_Dia is
    (lunes,martes,miercoles,jueves,viernes,sabado,domingo);
  package Pack_Dia_IO is new Ada.Text_IO.Enumeration_IO
    (T_Dia);
  use Pack_Dia_IO;
  Dia : T_Dia := lunes;
begin
  Ada.Text_IO.Put("Valor inicial: ");
  Ada.Text_IO.Put(Dia);
  Ada.Text_IO.New_Line;
  Ada.Text_IO.Put("Introduce el dia de la semana: ");
  Get(Dia);
  Ada.Text_IO.New_Line;
  Ada.Text_IO.Put("El dia introducido es: ");
  Put(Dia);
end Dias_De_La_Semana;
```

El procedimiento "Get" sirve para realizar la entrada de datos al programa y está definido en todos los paquetes genéricos que sirven para realizar las operaciones de entrada/ salida que veremos a lo largo del presente manual.

Los paquetes genéricos de entrada/ salida de Ada definen los procedimientos "Get" y "Put" que ya se han visto. Para utilizarlos hay que crear una instancia con el tipo de datos adecuado, para lo cual se utiliza el operador **new**.

Ejercicio 2: haga un programa similar al ejemplo 5 pero usando el tipo T_Color.

4.1.1. Tipo carácter

Un tipo carácter es cualquier tipo enumerado que tenga al menos un valor literal que sea un literal carácter, es decir, que esté delimitado por apóstrofes. Está predefinido el tipo *Character* que comprende los 256 caracteres del conjunto de caracteres del lenguaje. La entrada/ salida de caracteres se realiza mediante el paquete *Ada.Text_IO*.

```
type T_Vocales is ('a','e','i','o','u');
type T_Cinco_Numeros is ('1','2','3','4','5');
```

4.1.2. Tipo boolean

Existe predefinido en el paquete estándar un tipo enumerado llamado *Boolean* con los valores *True* y *False*. La entrada/ salida con este tipo se realiza como en el caso de los tipos enumerados (paquete genérico *Ada.Text_Io.Enumeration_IO*).

4.2. TIPOS ENTEROS

Los tipos enteros sirven para representar cantidades numéricas exactas, sin parte fraccionaria. Por su propia naturaleza, los tipos enteros se clasifican como tipo discreto, ya que sólo pueden tomar un conjunto determinado de valores. Por tanto, muchos aspectos que se comentan en este apartado son extensibles al resto de tipos discretos, ya sean predefinidos (como *Boolean* o *Character*) o definidos por el usuario. Los tipos enteros se dividen en dos grandes grupos:

1. **Enteros con signo**. En este tipo se almacenan los valores enteros que se manejan diariamente. Ada predefine los siguientes tipos de enteros con signo (en orden creciente del rango de valores): *Short_Integer*, *Integer*, *Long_Integer* y *Long_Long_Integer*. Además también define dos subtipos del tipo *Integer* : *Natural* (0 .. *Integer'Last*) y *Positive* (1 .. *Integer'Last*). Para el tipo *Integer*, además, define también el paquete no genérico *Ada.Integer_Text_Io* para realizar las operaciones de entrada/ salida. Para los restantes tipos (y para los definidos por el usuario) hay que instanciar el paquete genérico *Ada.Text_Io.Integer_Io* con el tipo de dato correspondiente. Existen dos posibilidades para definir un nuevo tipo de entero:

   - Definirlo como un tipo derivado (**new**) de *Integer* y restringirle el rango de posibles valores que las variables del tipo pueden adoptar:

```
with Ada.Text_Io;
type T_Sig_Byte is new Integer range 0..255;
package Pack_Sig_Byte_IO is new
        Ada.Text_Io.Integer_Io(T_Sig_Byte);
```

   - Definirlo como un tipo nuevo, especificando sólo el rango:

```
use Ada.Text_Io;
type T_Sig_Byte is range 0..255;
package Pack_Sig_Byte_IO is new
      Integer_IO(T_Sig_Byte);
```

   La diferencia entre ambas declaraciones es muy sutil y depende realmente del compilador. Se supone que la segunda reserva el espacio necesario para almacenar el rango de valores, con lo que se minimiza el uso de memoria. Por el contrario, puede que algunas operaciones intermedias con el tipo se salgan de este rango, aunque el resultado final esté dentro de él.

```
with Ada.Text_Io;

procedure Nuevos_Tipos_Enteros is
  type T_Integer_A is new Integer range 0..100;
  package Pack_Integer_A_Io is new
        Ada.Text_Io.Integer_Io(T_Integer_A);
  use Pack_Integer_A_Io;
```

```
   Na1, Nb1, Nc1 : T_Integer_A := 70;
   type T_Integer_B is range 0..100;
   package Pack_Integer_B_Io is new
         Ada.Text_Io.Integer_Io(T_Integer_B);
   use Pack_Integer_B_Io;
   Na2, Nb2, Nc2 : T_Integer_B := 70;
begin
   Nc1 := (Na1+Nb1)/ Na1;
   Ada.Text_Io.Put("Usando 'new Integer' => ");
   Put(Nc1);
   Nc2 := (Na2+Nb2)/ Na2; -- desborda! CONSTRAINT_ERROR
   Ada.Text_Io.Put("Sin usar 'new Integer' => ");
   Put(Nc2);
end Nuevos_Tipos_Enteros;
```

2. **Enteros modulares**. Los enteros modulares (definidos con **mod**) utilizan aritmética modular: si A y B son variables de un tipo modular (con módulo M), pueden tomar valores comprendidos entre 0 y M-1, y una expresión como "A+B" representa el "resto de la división (A+B)/ M". El reloj, por ejemplo, utiliza aritmética modular 12 o 24 para las horas. El módulo es siempre positivo. Ada no predefine ningún tipo modular y las operaciones de entrada/ salida se realizan mediante el paquete genérico *Modular_Io*.

```
type T_Mod_Byte is mod 256;
          --tipo entero modular, valores de 0 a 255
package Pack_Mod_Byte_IO is new
      Ada.Text_Io.Modular_Io(T_Mod_Byte);
```

Ejercicio 3: Escriba un programa que le pida que introduzca su edad y muestre por pantalla el mensaje "Tienes xxx años".

El procedimiento *Put* para enteros con signo y enteros modulares admite un parámetro para controlar el número de espacios en blanco que deja antes de imprimirlos. Existen dos posibilidades de fijar este parámetro:

```
with Ada.Text_Io;
use Ada.Text_Io;
procedure Solucion is
   type T_Mod_Byte is mod 256;
   --tipo entero modular (con módulo 256, valores de 0 a 255)
   package Mod_Byte_IO is new Modular_IO(T_Mod_Byte);
   A : T_Mod_Byte := 9;
begin
   Mod_Byte_Io.Put(A, Width=>0); New_line;   --1ª forma
   Mod_Byte_Io.Put(A, 0); New_Line;          --1ª forma bis
   Mod_Byte_Io.Default_Width := 0;           --2ª forma
   Mod_Byte_Io.Put(A);             --definido para siempre
end Solucion;
```

4.3. TIPOS REALES

Los tipos reales son tipos aproximados (a diferencia de los enteros) que introducen problemas de precisión y de redondeo cuando se realizan operaciones con ellos, puesto que representan, en muchos casos, valores tan solo "aproximados" (pruebe a representar en papel 1/ 3). Este problema no es exclusivo de Ada, sino que aparece en todos los lenguajes de programación. Los tipos reales se dividen en:

1. **Reales en coma flotante**, en los que se define un error *relativo*. Ada predefine el tipo *Float, Short_Float, Long_Float y Long_Long_Float* y un conjunto de paquetes no genéricos para realizar la entrada/ salida (*Ada.Float_Text_Io, Ada.Short_Float_Text_Io*, etc). La definición de nuevos tipos en coma flotante se indica mediante la palabra reservada **digits** seguida del número de dígitos que se van a utilizar para representar el número (en notación científica excluyendo el exponente) y, opcionalmente, el rango. Para poder realizar operaciones de entrada/ salida se utiliza el paquete genérico *Ada.Text_Io.Float_Io*.

```
declare
  type T_Real is digits 3;  --no distingue 3.111 de 3.119
  type T_Real_Restr is digits 3 range 0.0..1.0;
  A : T_Real := 456_056_223.321;
  B : T_Real_Restr := 0.001;
begin
  Put(A); New_Line;  --falta declarar paquetes IO
  A:= 456_856_223.321;
  Put(A); New_Line;
  A := 0.000_000_123_3;
  Put(A); New_Line;
  A := 0.000_000_123_9;
  Put(A); New_Line;
  B := 2.003;        -- error, fuera de rango
end;
```

Ejercicio 4: complete el código anterior declarando el paquete para realizar la entrada/ salida y compruebe su ejecución.

2. **Reales en coma fija**, en los que se define un error *absoluto*. Los reales en coma fija se dividen a su vez en reales en coma fija *ordinarios* y reales en coma fija *decimal*. Para ambos tipos se debe especificar el límite absoluto del error (**delta**) como número real. En el caso de un tipo en coma fijo ordinario (de ahora en adelante coma fija) se tiene que especificar también el rango (**range**). La entrada/ salida para números en coma fija se realiza con el paquete genérico *Ada.Text_Io.Fixed_Io*.

```
type T_Real_Fijo is delta 0.125 range -100.0 .. 100.0;
--tipo fijo ordinario con precision de 0.125 en su rango
```

Cuando el **delta** (valor absoluto del error) es potencia de 10 el tipo se tiene que definir en *coma fija decimal*. Para definir un tipo en coma fija decimal se tiene que especificar el **delta** pero también el número de dígitos de precisión (mediante **digits**), aunque el rango en este caso es opcional. La entrada/ salida para un tipo en coma fija decimal se realiza con el paquete genérico *Ada.Text_Io.Decimal_Io*.

```
type T_Precio_en_Euros is delta 0.01 digits 6;
        --tipo fijo decimal -9999.99 a 9999.99
```

4.4. ALGUNOS ATRIBUTOS APLICABLES A LOS TIPOS ESCALARES

Los atributos de un tipo son operaciones especiales que permiten averiguar propiedades del mismo. Estos atributos se invocan directamente sobre el tipo de dato: si T es un tipo escalar y At un atributo del tipo que acepta un parámetro P, el atributo se invoca escribiendo T'At(P). A continuación se muestran unas tablas que contienen la definición de algunos de los atributos aplicables a los distintos tipos de datos definidos en Ada.

Tabla 1: *Atributos definidos para los tipos escalares*	
T 'First	Devuelve el valor más pequeño del tipo T.
T 'Last	Devuelve el valor más grande del tipo T.
T 'Range	Devuelve el rango T'First..T'Last.
T 'Succ(V)	Siendo V un valor de tipo T, devuelve el valor que le sigue (en los tipos reales depende de la representación). T'Succ(T'Last) no pertenece al tipo T.
T 'Pred(V)	Siendo V un valor de tipo T, devuelve el valor que le precede (en los tipos reales depende de la representación). T'Pred(T'First) no pertenece al tipo T.
T 'Max(V1, V2)	Perteneciendo tanto V1 como V2 al tipo, devuelve el mayor de los dos.
T 'Min(V1, V2)	Perteneciendo tanto V1 como V2 al tipo, devuelve el menor de los dos.
T 'Image(V)	Siendo V un valor de tipo T, devuelve un *string* con la representación literal de V.
T 'Width	Devuelve el número máximo de caracteres que requerirá un literal de tipo T devuelto por T'Image(V).
T 'Value(S)	Siendo S una porción con la representación literal de un valor de tipo T, devuelve el valor correspondiente.

Tabla 2: *Atributos definidos para los tipos discretos*	
T 'Pos(V)	Siendo V un valor de tipo T, devuelve la posición de V en el rango de valores de T.
T 'Val(P)	Siendo P un valor entero, devuelve el valor de T que le corresponde, según la definición del tipo enumerado (de no existir se produce un *Constraint_Error*).

Tabla 3: *Atributos definidos para los tipos modulares*	
T 'Mod(V)	Aplica el operador **mod** al valor V que se le pasa como parámetro.
T 'Modulus	Devuelve el valor del modulo, sólo aplicable a enteros modulares.

Resumiendo, los tipos enumerados (es decir, enumerados, carácter y booleano) y los tipos enteros (con signo) tienen definidos los atributos mostrados en las tablas 1 y 2, mientras que los tipos modulares tienen definidos los atributos mostrados en las tablas 1, 2 y 3.

Ejercicio 5: escriba un programa que defina un nuevo tipo entero y un tipo derivado de *Integer* y muestre por pantalla los atributos para los dos tipos definidos y para el tipo *Integer*.

Ejemplo 6: el fichero planetas.adb contiene un ejemplo completo del uso de los atributos de los tipos enumerados. Compile y ejecute el código.

Tabla 4: Atributos definidos para los tipos en coma flotante	
T 'Digits	Devuelve la precisión del tipo (relativa).

Tabla 5: Atributos definidos para los tipos en coma fija ordinarios	
T 'Small	Devuelve el valor más pequeño del tipo (primera potencia de 2 menor que delta).
T 'Delta	Devuelve la precisión del tipo (absoluta).
T 'Fore	Número de dígitos necesario para representar el número ANTES de la coma.
T 'Aft	Número de dígitos necesario para representar el número DESPUES de la coma.

Tabla 6: Atributos definidos para los tipos en coma fija decimal	
T 'Digits	Devuelve la precisión del tipo (relativa).
T 'Scale	Devuelve el número de dígitos tras la coma decimal.
T 'Round(X)	Redondea el número en coma fija decimal.

Resumiendo, los tipos de datos en coma flotante tienen definidos los atributos descritos en las tablas 1 y 4. Los tipos de datos en coma fija ordinarios definen los atributos de las tablas 1 y 5, mientras que a los tipos en coma fija decimal se aplican los atributos de las tablas 1, 5 y 6.

Ejemplo 7: compile y ejecute el siguiente programa

```
with Ada.Text_IO, Ada.Integer_Text_IO, Ada.Float_Text_IO;
use Ada.Text_IO;
procedure Coma_Fija is
   type T_Fijo_Grande is delta 2.567_765
            range 0.0 .. 12_345.0;
   -- el sistema usa "un small propio" que es la primera
   -- potencia de 2 menor que delta. Si queremos que coincida
   -- con delta se lo podemos decir al sistema mediante la
   -- siguiente clausula de representacion
   for T_Fijo_Grande'Small use 2.567_765;

   type T_Fijo_Pequenyo is delta 0.003_765
          range 0.0 .. 12_345.0;
   for T_Fijo_Pequenyo'Small use 0.003_765;

   type T_Fijo_Decimal is delta 0.001 digits 9;
   package Fijo_Decimal_Io is new
         Ada.Text_Io.Decimal_IO (T_Fijo_Decimal);
   use Fijo_Decimal_Io;
begin
   Put ("GRANDE => Small nuevo: ");
   Ada.Float_Text_Io.Put (T_Fijo_Grande'Small);
   New_Line;
   Put ("GRANDE => Delta: ");
   Ada.Float_Text_Io.Put (T_Fijo_Grande'Delta);
   New_Line;
   Put ("GRANDE => Fore: ");
   Ada.Integer_Text_Io.Put (T_Fijo_Grande'Fore);
   New_Line;
   Put ("GRANDE => After: ");
   Ada.Integer_Text_Io.Put (T_Fijo_Grande'Aft);
   New_Line(2);
   Put ("PEQUENYO => Small: ");
   Ada.Float_Text_Io.Put (T_Fijo_Pequenyo'Small);
   New_Line;
   Put ("PEQUENYO => Delta: ");
   Ada.Float_Text_Io.Put (T_Fijo_Pequenyo'Delta);
   New_Line;
   Put ("PEQUENYO => Fore: ");
   Ada.Integer_Text_Io.Put (T_Fijo_Pequenyo'Fore);
   New_Line;
   Put ("PEQUENYO => After: ");
   Ada.Integer_Text_Io.Put (T_Fijo_Pequenyo'Aft);
   New_Line(2);
   Put ("DECIMAL => Digits: ");
   Ada.Integer_Text_IO.Put (T_Fijo_Decimal'Digits);
   New_Line;
```

```
   Put ("DECIMAL => Scale: ");
   Ada.Integer_Text_Io.Put (T_Fijo_Decimal'Scale);
   New_Line;
   Put ("DECIMAL => Round (45.0026): ");
   Fijo_Decimal_IO.Put (T_Fijo_Decimal'Round (45.0026));
   New_Line;
   Put ("DECIMAL => Round (45.0024): ");
   Fijo_Decimal_IO.Put (T_Fijo_Decimal'Round (45.0024));
   New_Line;
end Coma_Fija;
```

4.5. TIPOS DERIVADOS Y SUBTIPOS

Un tipo se dice que es derivado si en su definición se utiliza la palabra **new** en referencia a otro tipo (como el tipo *T_Sig_Byte*, derivado de *Integer*, visto en un ejemplo anterior); un tipo derivado obtiene sus características del que deriva (su tipo "padre"), aunque realmente se está creando un nuevo tipo en el lenguaje (que es incompatible con el de su "padre", como se mostrará más adelante). Ejemplos de declaraciones de tipos derivados son:

```
type T_Entero is new Integer;
type T_Entero_Corto is new Entero range 0..1000;
```

A veces es conveniente hacer una declaración incompleta de tipo que permita disponer de un nombre de tipo que en realidad se definirá con posterioridad en la misma región del programa.

```
type T_Un_Tipo; --sólo se declara un nombre
                --la definición del tipo requiere
                --una nueva declaración
```

Un subtipo define un subconjunto restringido de valores de un tipo base con atributos específicos, aunque siguen formando parte del tipo y por tanto se pueden seguir haciendo operaciones entre ellos (ya se adelantó una definición de subtipo en el apartado de definición de tipos enumerados). Se declara utilizando la palabra **subtype** en vez de **type**; en la definición hay que indicar cuál es el tipo base del subtipo.

```
subtype T_Byte is Integer range 0..255;
```

Una regla que puede servir para decidir cuándo declarar un subtipo o cuándo definir uno nuevo (con **new**) es la siguiente: se debe definir un subtipo cuando se quiera aprovechar la característica de chequeo de rango que ofrece Ada y se quieren mezclar valores del tipo y del subtipo sin tener que estar haciendo conversiones de tipos continuamente.

Ejemplo 8: vuelva a compilar y ejecutar *planetas.adb* y compruebe el comportamiento de los subtipos.

4.6. COMPATIBILIDAD Y CONVERSIÓN DE TIPOS

Ada es un lenguaje *fuertemente tipado*, lo cual implica que no se pueden usar valores de un tipo en operaciones de otro tipo sin efectuar una conversión explícita de datos, dando así las operaciones siempre resultados del tipo correcto. Así, en general no se pueden mezclar valores de tipos diferentes aún cuando pudieran parecer equivalentes (ni siquiera de un tipo derivado con **new** y su tipo padre). Esta regla no incluye a los subtipos: los valores de un subtipo son valores del tipo base, es decir las operaciones con valores de distintos subtipos de un mismo tipo están permitidas. Como ejemplo vea el siguiente código:

```
type T_Manzanas is new Integer;
type T_Naranjas is new Integer;
Naranjas : T_Naranjas := 5;
Manzanas : T_Manzanas := 8;
Manzanas + Naranjas; –-error, tipos incompatibles
```

Declare nuevos tipos cuando las variables de su programa no tengan ninguna relación entre sí aunque tomen valores compatibles (recuerde el tocino y la velocidad). Esta capacidad del lenguaje permite evitar fallos semánticos, de difícil identificación.

Se pueden realizar conversiones explícitas entre tipos relacionados utilizando el tipo destino como si fuera el nombre de una función con un parámetro que es una expresión del tipo origen de la conversión.

```
X: Float;
Y: Integer := 5;
X := Float(Y); --se asigna a X el valor real 5.0
```

Cuando se realiza una conversión de tipos se está devolviendo un valor compatible para el tipo al que se convierte, de acuerdo a las reglas de conversión del lenguaje. En ningún caso se modifica el tipo de la variable original.

Tenga especial cuidado al realizar una conversión entre tipos. Ada sólo informa cuando los tipos son incompatibles, pero es deber del programador asegurarse de que no se va a perder información al realizar el cambio y de que éste tiene sentido:

```
type T_Manzanas is new Integer;
type T_Naranjas is new Integer;
Naranjas : T_Naranjas := 5;
Manzanas : T_Manzanas := 8;
Manzanas + T_Manzanas(Naranjas);
          –-compila, pero no tiene sentido
```

Por último, resta por mencionar los *tipos universales*. En el apartado sobre literales numéricos se dijo que éstos pertenecen a los tipos universales. Como se ha visto, Ada es un lenguaje fuertemente tipado, por lo que no se pueden realizar operaciones entre

variables de distinto tipo si antes no son convertidas. Entonces, explique el siguiente código:

```
type T_Raro is new Integer;  -- nuevo tipo
A : T_Raro := 8;             -- declaracion inicial
A := A + 6;   -- ¿son compatibles? ¿es correcta la asignacion?
```

La explicación es que los tipos universales (los literales numéricos) son compatibles con todos los tipos, nuevos o derivados, del lenguaje. De esta forma la operación sigue siendo correcta.

Ejercicio 6: dada la siguiente declaración de tipos y variables, compruebe si las expresiones que se indican a continuación son correctas o no. Para ello deberá incluir en el programa los paquetes correctos para poder mostrar los resultados.

```
type Index is range 1..100;
type Length is digits 5 range 0.0..100.0;
First, Last: Index;
Front, Side: Length;
                    -- ¿son correctas?:
Last := First + 15;
Side := 2.5 * Front;
Side := 2 * Front;
Side := Front + 2* First;
Side := Front + 2.0*Length(First);
```

5. EXPRESIONES Y OPERADORES

Una expresión es una combinación de operadores y operandos de cuya evaluación se obtiene un valor. Los operandos pueden ser nombres que denoten variables o constantes, funciones, literales de cualquier tipo adecuado de acuerdo con los operadores u otras expresiones más simples. La evaluación de una expresión da lugar a un valor de algún tipo. Una expresión se dice que es del tipo de su resultado. Ejemplos de expresiones:

```
a + 5*b
(a >= 0) and ((b+5) > 10)
-a * 2 + b
-b + sqrt (b**2 - 4*a*c)
```

Las expresiones se evalúan de acuerdo con la precedencia de los operadores. Ante una secuencia de operadores de igual precedencia, la evaluación se realiza según el orden de escritura, de izquierda a derecha. El orden de evaluación puede modificarse usando paréntesis. Todas las operaciones están definidas en el lenguaje como funciones, como ya se mostrará en la lección de subprogramas.

Ada agrupa los operadores en 6 categorías, de menor a mayor precedencia. Los operadores binarios se usan en formato infijo (<*operando_izquierdo*> <*operador*> <*operando_derecho*>), como en "*a + b*". Los operadores unarios se usan en formato prefijo (<*operador*> <*operando*>), como en *"-5"*.

5.1. OPERADORES LÓGICOS

Están predefinidos los siguientes operadores lógicos **not**, **and**, **or** y **xor**. Su significado es el convencional:

A	B	(A and B)	(A or B)	(A xor B)
True	True	True	True	False
True	False	False	True	True
False	True	False	True	True
False	False	False	False	False

Existen versiones de los operadores **and** y **or**, llamadas **and then** y **or else** que tienen el mismo significado, pero realizan una "evaluación en cortocircuito", que consiste en que evalúan siempre primero el operando izquierdo y, si el valor de éste es suficiente para determinar el resultado, no evalúan el operando derecho. Estos operadores de cortocircuito han de usarse con cuidado, ya que aunque consiguen que el programa se ejecute más rápido, puede que no se ejecute el código de segundo o del resto operadores (se pueden escribir varios **and then** e incluso mezclarlos con **or else**).

5.1.1. Operadores lógicos sobre tipos modulares

Con los tipos modulares (**mod**) los operadores lógicos (**and, or, xor, not**) no producen un resultado del tipo Boolean, sino que realizan las operaciones binarias bit a bit.

Ejemplo 9: complete y compile el siguiente código y compruebe su ejecución.

```
declare
   type T_Byte is mod 2#11111111#; -- de 0 a 255
   package Byte_Io is new Modular_Io (T_Byte);
   use Byte_Io;
   B : T_Byte := 2#11010110#; --214
   C : T_Byte := 2#10000111#; --135
begin
   Put("El valor de 'B' en decimal: ");Put(B);New_Line;
   Put("El valor de 'C' en decimal: ");Put(C);New_Line;
   Put("'B' and 'C': ");Put(B and C);New_Line;
   Put("          binario: ");
   Put(B and C, Base=>2);New_Line;
   Put("'B' or 'C': ");Put(B or C);New_Line;
   Put("          binario: ");
   Put(B or C, Base=>2);New_Line;
   Put("'B' xor 'C': ");Put(B xor C);New_Line;
   Put("          binario: ");
   Put(B xor C, Base=>2);New_Line;
   Put("not 'B' = 255-B: ");
   Put(not B);Put(" = ");Put(254-B); -- 256 NO esta incluido
end;
```

5.1.2. Operadores relacionales

Los operadores relacionales devuelven un resultado de tipo Boolean. Dentro de este grupo están predefinidos los operadores de igualdad "=" y desigualdad "/ =" y los operadores de comparación "<", "<=", ">" y ">=".

Existe también un operador de pertenencia (**in**, **not in**) que determina si un valor pertenece a un rango:

```
if (Char in 'A'..'Z') or (Char in 'a'..'z') then
   Put_Line ("El caracter es una letra");
end if;

if Char not in '0'..'9' then
   Put_Line ("El caracter NO es un numero");
end if;
```

5.2. OPERADORES ARITMÉTICOS

Los operadores de adición predefinidos para cualquier tipo numérico son "+" y "-". Ejemplo de uso en expresiones (sean A, B y C de tipo T):

```
C := A + B;
A := A – B;
```

Los operadores unarios de adición predefinidos para cualquier tipo numérico, *T*, son la identidad "+" y la negación "-". Ejemplo de uso en expresiones (sean A y B de tipo T):

```
B:= -A;
```

Los operadores de multiplicación *"*"* y división *"/"* están predefinidos para diversas combinaciones de enteros y reales. Ejemplo de uso en expresiones (sean A, B y C del mismo tipo entero o real):

```
C := A * B;
```

Por último se introducirán dos operadores relacionados: el operador módulo (**mod**) para trabajar en aritmética modular y el operador resto de la división entera (**rem**). Ejemplo de uso en expresiones (sean A, B y C de tipo T):

```
C := A rem B;      -- tiene el signo de 'A'
C := A mod B;      -- tiene el signo de 'B'
```

Ejercicio 7: escriba un programa que compruebe el resultado de aplicar los operadores *mod* y *rem* cuando los operandos son positivos y negativos (hay cuatro opciones).

La aritmética modular cumple las siguientes igualdades:

$A \bmod B = (A + k*B) \bmod B$, para todo k y $A = k*B + (A \bmod B)$, para algún k

Los operadores de máxima prioridad en el lenguaje son:

- El operador de cálculo del valor absoluto **abs**, definido para cualquier tipo numérico.
- El operador de negación lógica **not**, definido para cualquier tipo booleano.
- El operador de exponente "**". El número base puede ser de los tipos entero o en coma flotante, pero el exponente ha de ser un numero entero (positivo o negativo).

5.3. SOBRECARGA DE OPERADORES

Ada permite que el programador sobrecargue los operadores del lenguaje, esto es, que pueda redefinirlos dándoles nuevos significados. Para sobrecargar un operador, simplemente hay que definir una función cuyo nombre sea el operador entre comillas y que tenga los parámetros adecuados (se verá un ejemplo más adelante cuando se introduzcan los subprogramas).

6. SENTENCIAS DE CONTROL

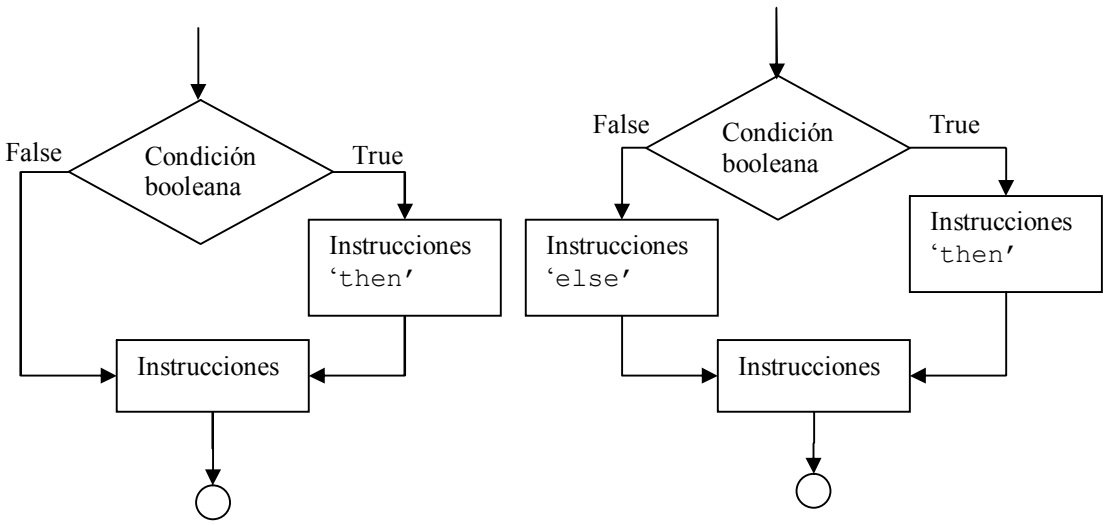
Las sentencias de control son sentencias compuestas que controlan la ejecución del grupo de sentencias que delimitan.

6.1. SELECCIÓN ENTRE ALTERNATIVAS LÓGICAS

La selección entre dos posibles alternativas de ejecución se realiza mediante la evaluación de una condición formulada por una expresión *Booleana* (recuerde el tipado fuerte).

```
if condición_booleana then
  ...          -- bloque de instrucciones
[elsif
  ... ]        -- bloque de instrucciones
[elsif
  ... ]        -- bloque de instrucciones
[else
  ... ]        -- bloque de instrucciones
end if;
```

Esta construcción se corresponde con los siguientes diagramas de flujo:



Ejemplo 10: ejecute el siguiente programa

```
with Ada.Text_IO;
use Ada.Text_IO;
procedure Actividad_Diaria is
   type T_Dias is
```

```
    (lunes,martes,miercoles,jueves,viernes,sabado,domingo);
  package Pack_T_Dias_IO is new Enumeration_IO (T_Dias);
  use Pack_T_Dias_IO;
  subtype T_Finde is T_Dias range sabado..domingo;
  subtype T_Dias_Laborables is T_Dias range lunes..viernes;
  Dia : T_Dias := lunes;
begin
  Put("Que dia es hoy? ");
  Get(Dia);
  if Dia = miercoles then
     Put_Line("Estudiar Ada");
  elsif Dia in T_Finde then
     Put_Line("Tiempo libre!");
  else
     Put_Line("Ir a clase");
  end if;
end Actividad_Diaria;
```

6.2. MÚLTIPLES ALTERNATIVAS EN TIPOS ENUMERADOS

La selección entre múltiples alternativas de ejecución se realiza mediante la evaluación de un selector.

```
case selector is
  when alternativa => ...   -- bloque de instrucciones
  when alternativa => ...   -- bloque de instrucciones
  ...
  when others => ...        -- bloque de instrucciones
end case;
```

La variable que se utiliza como *selector* tiene que pertenecer a uno de los tipos discretos, ya sea un tipo enumerado o un tipo entero. Las *alternativas* pueden ser uno o varios valores, o rangos, del tipo del *selector* separados por "| " (equivale al operador OR). Los valores no pueden repetirse entre dos cláusulas ***when***. En el caso de que las cláusulas ***when*** no cubran todos los posibles valores del tipo del selector, es necesario incluir la cláusula ***others*** para los valores no contemplados. En caso de aparecer, ***others*** va al final del ***case***. Ada es un lenguaje seguro y obliga a que todos los valores de un tipo enumerado aparezcan en el **case**.

```
case Mes is
  when 1 .. 2 |  12 => Put("Invierno");
  when 3 .. 5 => Put("Primavera");
  when 6 .. 8 => Put("Verano");
  when 9 .. 11 => Put("Otoño");
  when others => Put("¿En qué planeta estás?");
end case;
```

Ejercicio 8: modifique el ejemplo 10 para que utilice la sentencia *case*.

6.3. REPETICIÓN CONTROLADA POR CONTADOR

Hace que un conjunto de sentencias se ejecute un número de veces determinado a priori; la cuenta de las ejecuciones realizadas se lleva a cabo mediante una variable de control del bucle que va tomando los sucesivos valores de un rango ordinal.

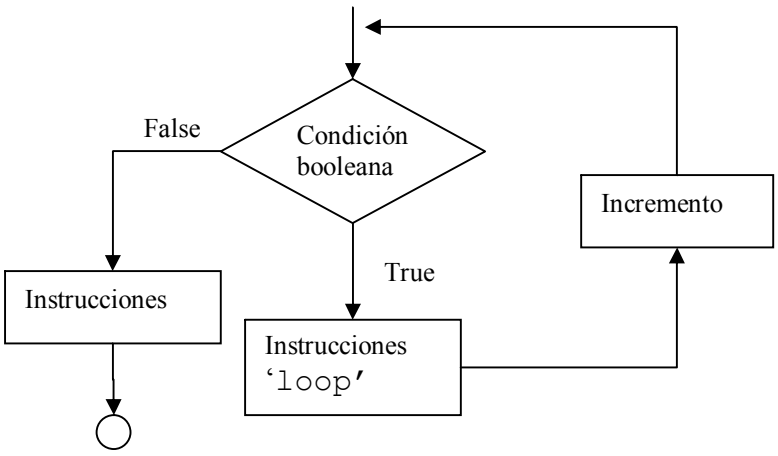
```
[etiqueta:]              -- optativo
for Num in 1..5 loop
   Put(Num);             -- escribe 1 2 3 4 5
end loop [etiqueta];
```

El parámetro de control no se declara (menos trabajo para el programador), adopta automáticamente el tipo del rango que debe recorrer, es local al bucle (por tanto, oculta cualquier otra declaración exterior con el mismo nombre), no puede modificarse y deja de existir al acabar el bucle.

El rango puede recorrerse también en orden inverso utilizando la palabra ***reverse***.

```
[etiqueta:]
for Num in reverse 1..5 loop
   Put(Num); -- escribe 5 4 3 2 1
end loop [etiqueta];
```

El flujo de control del bucle **for** se muestra en el siguiente diagrama:



Ejemplo 11: ejecute el siguiente programa

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

```
   type T_Mod is mod 8;
   package Pack_Mod_IO is new Modular_IO(T_Mod);
begin
   for Repe in 1..5 loop
      for Modulo in T_Mod'range loop
         Pack_Mod_IO.Put(Modulo);
         Put(" ");
      end loop;
      New_Line;
   end loop;
end Modular;
```

Ejercicio 9: escriba un programa que muestre todos los valores del tipo enumerado T_Dias con un *for*.

6.4. REPETICIÓN CONTROLADA POR CONDICIÓN LÓGICA

Ada sólo ofrece explícitamente la versión con control previo, es decir, la condición siempre se evalúa antes de entrar en el bucle (no define bucle *do-while*).

```
[etiqueta:]
while condición_booleana loop
   ... -- bloque de instrucciones
end loop [etiqueta];
```

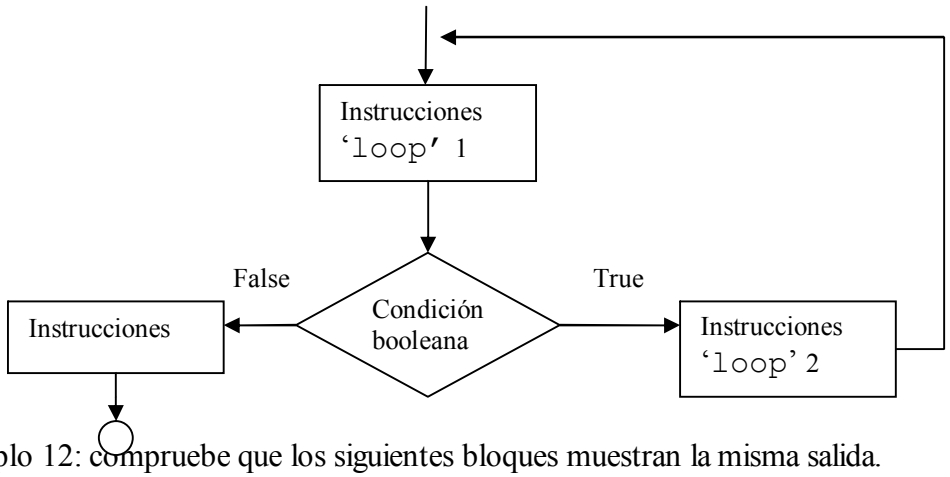
Sin embargo, ofrece un bucle sin esquema de iteración prestablecido que puede utilizarse para simular cualquier otro. Su forma básica es un bucle infinito:

```
[etiqueta:]
loop
   ...
end loop [etiqueta];
```

De un bucle como éste (de hecho de cualquier bucle) se puede salir usando una sentencia ***exit***. Si tiene bucles anidados se sale del bucle interior. En caso de que se coloque una etiqueta se puede salir de cualquier bucle, independientemente del nivel de anidamiento.

```
[etiqueta:]
loop
   ...
   exit [etiqueta] when condición_booleana;
   ...
end loop [etiqueta];
```

El diagrama de flujo genérico para los bucles de repetición se muestra en el siguiente esquema. En él puede verse como, jugando con la presencia de los bloques de instrucciones 1 y 2, se puede obtener cualquier esquema de repetición deseado.



Ejemplo 12: compruebe que los siguientes bloques muestran la misma salida.

```
declare
   type T_Cuenta_Atras is range 1..10;
begin
   for_loop:
   for I in reverse T_Cuenta_Atras loop
      Put(I);       --hay que definir un paquete...
      delay 1.0;
   end loop for_loop;
   New_Line(2);
   Put("IGNICION!!!!");
end;
```

```
declare
   type T_Cuenta_Atras is range 1..10;
   I : T_Cuenta_Atras := 10;
begin
   while (I > 1) loop
      Put(I);       --hay que definir un paquete...
      I := I – 1;
      delay 1.0;
   end loop;
   Put(I);
   New_Line(2);
   Put("IGNICION!!!!");
end;
```

```
declare
   type T_Cuenta_Atras is range 1..10;
   I : T_Cuenta_Atras := 10;
begin
   ca_loop: loop
      Put(I);       --hay que definir un paquete...
      exit ca_loop when (I = 1);
      I := I – 1;
      delay 1.0;
   end loop ca_loop;
   New_Line(2);
   Put("IGNICION!!!!");
end;
```

Ejercicio 10: escriba un programa que solicite al usuario la entrada de caracteres por teclado y que continúe hasta que el carácter introducido sea un número. Imprima el número de caracteres pulsado hasta acabar.

7. TIPOS ESTRUCTURADOS

Un tipo estructurado define una agrupación de variables de un tipo más simple o de otro tipo estructurado; normalmente se distingue entre tipos estructurados *homogéneos* y tipos estructurados *heterogéneos*. Un tipo estructurado es *limitado* (ver capítulo 9) si se declara como tal o si alguno de sus elementos es de tipo *limitado*.

7.1. ESTRUCTURAS HOMOGÉNEAS. ARRAYS

Los *arrays* son estructuras homogéneas constituidas por un conjunto ordenado de elementos del mismo tipo, a los que se puede acceder individualmente indicando la posición que ocupan en el *array*, expresada como un conjunto de índices discretos. Un *array* puede ser uni o multidimensional; determinar la posición de un elemento requiere un índice por cada dimensión. Cuando se declara un *array* es necesario indicar el tipo de los elementos y el rango de variación de los índices de cada dimensión; si el *array* tiene más de una dimensión, se declara separando los rangos de cada dimensión mediante comas (,).

El acceso a un elemento del *array* se hace poniendo el nombre de la variable *array* y colocando su índice (o índices separados por comas) entre paréntesis justo detrás. Los operadores asignación (:=) e igualdad (=) se comportan de forma ligeramente diferente cuando se aplican entre variables de tipo array. Siempre que las variables sean del mismo tipo array, la asignación copia todos los valores de una variable a la otra, mientras que la igualdad comprueba si todas las posiciones de los dos arrays contienen los mismos valores.

```
type AT_Vector is array (1..4) of Integer;
V: AT_Vector; --array unidimensional
   --elementos V(1),V(2),V(3) y V(4) del tipo Integer

M: array(1..2,Character'('A')..Character'('B')) of Float;    --array bidimensional
--elementos M(1,'A'),M(1,'B'),M(2,'A') y M(2,'B') tipo Float
M(1,'A') := 0; --fila 1, columna 'A'

VM: array(1..2) of AT_Vector; --array unidimensional
  --array de arrays
VM(1)(3) := 0; --fila 1, fila 3 (es un array de arrays)
```

Observe que para declarar los arrays M y VM no se ha empleado un tipo de datos definido previamente. Las variables declaradas así (sin definir antes un tipo) se denominan variables *anónimas*. Ada permite definir variables anónimas solo en *arrays*, *tareas* (capítulo 12) y *variables protegidas* (capítulo 13).

7.1.1. Literales array

Son agregados de valores del tipo de los elementos del *array* que se pueden utilizar para inicializar variables o constantes de tipo *array*, asignarlos a variables de tipo array o emplearlos en otras operaciones que involucren arrays. Un agregado posicional de un array unidimensional se forma enumerando los valores según su posición:

```
V: array(1..5) of Integer := (12,3,45,6,23);
    -- V(1) = 12, V(2) = 3, V(3) = 45, V(4) = 6 y V(5) = 23
V(2..4) := (others => 8); -- agregado
    -- V(1) = 12, V(2) = 8, V(3) = 8, V(4) = 8 y V(5) = 23
```

Se puede utilizar una cláusula **others** para dar un mismo valor a las últimas posiciones del *array*.

```
V := (12,3,45, others => 0);
    -- V(1) = 12, V(2) = 3, V(3) = 45, V(4) = 0 y V(5) = 0
V := (others => 0); --todas las posiciones a cero.
```

En *arrays* multidimensional se enumeran los elementos agrupándolos por dimensiones:

```
M: array(1..3,1..2) of Character :=
          (('a','b'),('c','d'),('e','f'));
...
M := (('a','b'),('c', others => 'd'), others => ('e','f'));
```

También se pueden definir agregados por asociación nominal, indicando para cada valor la posición a que corresponde, en cuyo caso se pueden enumerar salteados y desordenados:

```
V := (4, 3 => 15, 5 => 8, others => 0);
   --V(1) = 4, V(2) = 0, V(3) = 15, V(4) = 0 y V(5) = 8
M := (3 => ('e','f'), 1 => ('a','b'), others => ('c','d'));
```

Ejercicio 11: defina un tipo array de 5 elementos enteros, declare una variable de ese tipo e inicialícela (1,2,3,4,5). Utilice un bucle "**for**" para mostrar por pantalla sus valores. Modifique posteriormente el programa para que pregunte al usuario los valores a asignar al *array* y luego lo vuelva a mostrar. Rehágalo para que utilice un bucle "**while**".

Otros ejemplos de declaración de arrays:

```
type T_Dia is (lunes,martes,miercoles,jueves,viernes,
          sabado,domingo);
type AT_Dias is array (T_Dia) of Float; ;
type AT_Fin is array (T_Dia range sabado..domingo) of Float;
Array_Cte : constant array (2..7) := (others=>6);
```

7.1.2. "Trozos" de array

Se puede seleccionar un rango en particular de un array sustituyendo el índice por un rango. Esto permite realizar operaciones de forma más cómoda así como asignar partes de un array a otro array (siempre que sean compatibles).

```
V: array(1..5) of Integer := (12,3,45,6,8);
      -- V(3..4) corresponde a los valores (45,6)
```

También está definido el operador concatenación (&), que permite unir trozos de *array* (siempre que contengan elementos del mismo tipo) para formar otros mayores. Este operador es de gran utilidad sobre todo en los *Strings* (que se explican más adelante).

7.1.3. Atributos aplicables a los arrays

Todos los *arrays* tienen los atributos: *First, Last, Length* y *Range*, si A es un *array* y N un valor entero positivo menor o igual que el número de dimensiones del *array*, se tiene que:

- **A'First**: límite inferior del rango del primer índice de A.
- **A'First(N):** límite inferior del rango del N-ésimo índice de A. A'First(1) = A'First
- **A'Last**: límite superior del rango del primer índice de A.
- **A'Last(N):** límite superior del rango del N-ésimo índice de A. A'Last(1) = A'Last
- **A'Range**: equivalente al rango A'First..A'Last.
- **A'Range(N):** equivalente al rango A'First(N)..A'Last(N). A'Range(1) = A'Range
- **A'Length**: número de valores del primer índice. Equivale a A'Last – A'First
- **A'Length(N):** número de valores del índice N. A'Length(1) = A'Length

El uso de los atributos permite escribir algoritmos aplicables independientemente del rango de las distintas dimensiones de un *array*.

Ejercicio 12: rescriba el ejercicio anterior para que no dependa del tamaño de array elegido. Pruébelo cambiando el tamaño del array.

Ejercicio 13: escriba un programa que, dado un array de un número variable de números aleatorios, lo ordene y luego lo muestre por pantalla.

7.1.4. Arrays no restringidos

Se pueden declarar *arrays* no-restringidos (*unconstraint arrays*) especificando el tipo de los índices pero no su rango. Existen para poder pasar *arrays* como parámetros a subprogramas (como se muestra en el siguiente capítulo) sin que el rango tenga que ser fijo en el subprograma. No olvide que Ada es un lenguaje de tipado fuerte.

```
type AT_Libre is array(Integer range <>) of Float;
```

No se pueden declarar variables de un tipo no-restringido: se debe definir primero un subtipo restringido o especificar directamente los rangos de los índices en la propia declaración de una variable.

```
subtype AT_Restringido is AT_Libre(1..50);
A: AT_Restringido;
B: AT_Libre(1..10);
```

7.1.5. El tipo predefinido String

Ada predefine el tipo *String* en el paquete *Standard* como un *array* no-restringido de caracteres. Debido a su naturaleza de *array*, la cadena de caracteres que alberga un *String* tiene que ocupar todas sus posiciones, aunque sea con espacios en blanco, lo cual dificulta un poco su uso.

```
type String is array (Positive range <>) of Character ;
Nombre : String (1..10); --hay que restringirlo
Nombre := "Ada       ";  --exactamente 10 caracteres ☹
Put(Nombre & "en honor a Lady" & Nombre); --concatenar
```

Los *Strings* son los principales usuarios de la capacidad de trocear un array y del operador concatenación (como ya se ha mostrado). El operador concatenación permite también unir caracteres con *Strings*. Para el tipo *String* están también definidos los siguientes operadores lógicos: =, / =, <, <=, >, >=.

Otros paquetes de utilidad para usar Strings son *Ada.Strings.Fixed* y *Ada.Strings.Search* (para usar con el tipo *String*). Los paquetes *Ada.Strings.Bounded* y *Ada.Strings.Unbounded* usan otras definiciones de cadena de caracteres, más útiles que *String* para algunos casos. Ver manual de referencia del lenguaje para más información.

7.2. ESTRUCTURAS HETEROGÉNEAS. RECORDS

Los **record** son estructuras heterogéneas: agregados de elementos (*campos*) de distintos tipos, que se mantienen bajo un nombre común porque están estrechamente relacionados entre sí, como por ejemplo los campos de una agenda (nombre, apellido, fecha nacimiento, teléfono, etc.). Un record se define con la palabra **record**, seguida de la declaración de sus campos y **end record**.

```
type RT_Record [discriminante] is record
   lista elementos;
end record;

type RT_Complejo is record
   Real, Imag: Float;
end record;
```

Se pueden especificar valores iniciales para los campos de un *record* en la propia definición del tipo.

```
type RT_Complejo is record
   Real, Imag: Float := 0.0;
end record;
```

El acceso a los campos individuales de una variable *record* se consigue poniendo un punto (.) detrás del nombre de la variable y a continuación el nombre del campo al que se quiere acceder. Un *record* puede contener variables de otro tipo record y para acceder a ellas se necesita utilizar un punto (.) más.

```
X, Y, Z: RT_Complejo;
...
X.Real := 1.0;
X.Imag := 1.0;
Y.Real := X.Real;
Y.Imag := X.Imag;
Z := Y;          --X, Y, Z tienen el mismo valor
```

Los *records* se pueden definir también con un *discriminante*, lo que permite asignar valores iniciales o por defecto a algún campo. Este discriminante ha de ser un tipo discreto.

```
type RT_Complejo2 (Re,Im : Integer := 0) is record
   Real : Float := Float(Re)/ 10.0;
   Imag : Float := Float(Im)*2.0;
end record;
```

7.2.1. Literales record

Se pueden formar literales **record** para usarlos en inicializaciones, asignaciones y otras operaciones de record de dos formas:

1. Como agregado *posicional*, especificando los valores de todos los campos en el orden adecuado y entre paréntesis, por ejemplo `X := (3.5, 7.1);`
2. Como agregado *nominal*, especificando los nombres de los campos (con =>) junto con los valores, por ejemplo `X := (real => 3.5, imag => 7.1);` Cuando se usa un agregado nominal los campos se pueden enumerar en cualquier orden, pero siempre hay que enumerarlos todos.

En ambos casos, se puede utilizar la palabra reservada **others** para inicializar los campos del **record** que no hayan sido explícitamente inicializados, ya que Ada requiere que el usuario los inicialice correctamente.

Ejercicio 14: desarrolle un programa para utilizarlo como agenda personal. Para ello defina un tipo record con los campos Nombre (*String* 1..3, iniciales del nombre), Edad (rango 1..120), Telefono_Casa (rango 95000000..988000000), Dia_Nacimiento (1..31), Mes_Nacimiento (tipo enumerado) y Año_Nacimiento (1910..2200) y

escriba un programa que solicite los datos por entrada y luego los muestre por pantalla. Posteriormente defina un nuevo record *Empleado* que use el record anterior y añada dos campos: sueldo y fecha de contrato, además de solicitar estos datos por teclado y luego mostrarlos.

7.2.2. Record variante

Se pueden definir tipos **record** que puedan tener campos diferentes en función del valor del discriminante.

```
type T_Figura is (Rectángulo, Círculo);
type RT_Polígono(Forma : T_Figura) is record
                        --'Forma' es el discriminante
   Pos_X, Pos_Y: Float;
   case Forma is
      when Rectángulo => Base, Altura: Float;
      when Círculo => Radio: Float;
   end case;
end record;
```

Cuando se declaren variables de este tipo hay que especificar obligatoriamente el valor del campo discriminante (excepto en la declaración de parámetros formales); una vez hecha la declaración no se puede cambiar.

```
R : RT_Polígono(Rectángulo);
```

8. SUBPROGRAMAS

Un subprograma es un procedimiento (**procedure**) o una función (**function**). La diferencia entre un procedimiento y una función es que el primero sólo indica la ejecución de una secuencia de instrucciones, en función de unos parámetros, mientras que la segunda representa un valor que se genera como resultado de su ejecución (diferencia más bien filosófica, aunque impone ciertas restricciones como ya se mostrará).

8.1. DEFINICIÓN, DECLARACIÓN Y USO

La definición de un subprograma consta de dos elementos:

1. **Cabecera**, donde se fija su clase, su nombre y se describen sus *parámetros formales*. Los parámetros formales van entre paréntesis (si no tiene parámetros no se ponen los paréntesis) y declaran las variables locales (existen sólo dentro del bloque declarativo del subprograma) que necesita el subprograma para realizar su cometido. Por último se añade la palabra **is**. En el caso de las funciones, la cabecera acaba con **return** <*tipo de dato*> **is.** La cabecera especifica todo lo que alguien que quiere usar el subprograma necesita saber sobre cómo invocarlo.
2. **Bloque declarativo,** como los que se han visto a lo largo del curso (**declare** para las variables, tipos y, lo que es nuevo, para la definición anidada de otros subprogramas) y el bloque de sentencias ejecutables del subprograma, delimitado por las palabras reservadas **begin** y **end.** Si el subprograma es una función, entre las sentencias ejecutables debe incluirse al menos una sentencia de retorno (**return**) con una expresión que indique el valor a devolver a quien la llamó (y del mismo tipo que el declarado en la cabecera!); si la ejecución de una función alcanza el final sin encontrar una sentencia de retorno, se produce un error (se eleva la excepción *Program_Error*). En los procedimientos puede usarse la sentencia **return** (sin parámetro) para retornar, aunque puede omitirse cuando el único punto de retorno se encuentra al final, como en el siguiente ejemplo.

```
procedure Intercambiar(A,B: in out Integer) is
   C: Integer;
begin
   C := A;
   A := B;
   B := C;
   --'return' automatico
end Intercambiar;

function Media(A,B: Float) return Float is
begin
   return (A + B) /  2.0; -- 'return' obligatorio
end Media;
```

Los subprogramas se invocan por su nombre y colocando, entre paréntesis, los valores de los parámetros que se le tienen que pasar (según muestra la cabecera). Los valores en concreto que se pasan se llaman *parámetros reales*. En caso de que no tengan parámetros, se invocan sin colocar los paréntesis. La llamada a un procedimiento constituye una sentencia (y puede ir en su propia línea), mientras que la llamada a una función sólo puede hacerse como parte de una expresión (porque devuelven un valor):

```
Intercambiar(X,Y);
Z := Media(L,M);
return Media(L,M);
if Media(L,M) > 5.0 then ...
```

Un subprograma puede constituir por sí mismo una unidad de librería o estar anidado dentro de una unidad mayor. Si un subprograma está anidado en una unidad mayor, la definición del subprograma debe escribirse en la sección de declaraciones de esa unidad.

```
procedure Principal is
   ...
   procedure Intercambiar(A,B: in out Integer) is
      ...
   begin
      ...
   end Intercambiar;
   ...
begin
   ...
end Principal;
```

Las declaraciones locales (tanto de variables locales como de otros subprogramas) están sujetas a las reglas de ámbito generales de Ada.

Ejemplo 13: estudie, compile y ejecute el fichero *esfera1.adb*.

Ejercicio 15: escriba un programa que, a partir de los datos contenidos en un array (generado aleatoriamente), defina dos funciones para obtener la media y la desviación típica y un procedimiento para imprimir el array (use arrays no-restringidos como parámetros).

En un mismo ámbito se pueden tener varios subprogramas con el mismo nombre, siempre que se diferencien en los parámetros o en el tipo del resultado (si son funciones). Esto se conoce como *sobrecarga de subprogramas*.

```
procedure Intercambiar (A, B :  in out Integer) is
   ...
begin
   ...
end Intercambiar;
```

```
procedure Intercambiar (A, B : in out Float) is
   ...
begin
   ...
end Intercambiar;
```

Se pueden usar funciones para sobrecargar los operadores del lenguaje, otorgándoles nuevos significados. La sobrecarga de operadores es una clase de sobrecarga de subprogramas. Para sobrecargar un operador, simplemente hay que definir una función cuyo nombre sea el operador entre comillas y que tenga los parámetros adecuados. Por ejemplo, dado el siguiente tipo:

```
type Complejo is record
   PReal, PImag: Float;
end record;
```

El siguiente fragmento de código muestra como se puede sobrecargar el operador de suma ("+") para utilizarlo con el fin de sumar números complejos:

```
function "+"(A, B : in Complejo) return Complejo is
   Suma: Complejo;
begin
   Suma.PReal := A.PReal + B.PReal;
   Suma.PImag := A.PImag + B.PImag;
   return Suma;
end "+";
```

Una vez definida esta función, se puede aplicar el operador entre variables del tipo definido en los parámetros, devolviendo un valor del tipo definido como resultado.

```
...
S, X, Y: Complejo;
...
S := X + Y;
```

Sólo se pueden redefinir los operadores del lenguaje (no se pueden inventar operadores) y manteniendo siempre su cardinalidad (los binarios se redefinirán con funciones de dos parámetros y los unarios con funciones de un parámetro). La lista completa de operadores que se pueden sobrecargar es: ****, *, / , +, –, mod, rem, abs, not, and, or, xor, =, <, <=, >, >=, &**. El operador desigualdad (/ =) no puede ser sobrecargado en caso de que el operador igualdad (=) devuelva un resultado de tipo *Boolean* (ya que en este caso '/ =' es 'not =').

Ejercicio 16: defina todas las funciones básicas para poder realizar operaciones con números complejos: "+" "-" "*" y "/ ". Escriba un programa de prueba.

8.1.1. Declaración de un subprograma

En determinadas circunstancias se precisa escribir una declaración de un subprograma separada de su definición. La declaración de un subprograma es como su cabecera, pero terminada en ";" en vez de con la palabra "**is**", para expresar que lo que se está dando es una vista de un subprograma cuya definición se halla en otro lugar.

```
procedure Intercambiar (A, B : in out Integer);
function Media (A, B : Float) return Float;
```

8.1.2. Definición separada (separate)

La definición de un subprograma que está anidado en otra unidad puede extraerse de la misma para compilarse por separado, dejándolo indicado en la unidad matriz mediante una declaración con cláusula **separate**. A efectos de reglas de ámbito es como si la definición del subprograma estuviera donde está dicha declaración. El subprograma separado debe indicar quién es su unidad matriz.

```
procedure Ejemplo_Separado is
   ...
   -- Indicación de que Intercambia se desarrollará aparte
   procedure Intercambia(A,B: in out Integer) is separate;
   ...
begin
   ...
end Ejemplo_Separado;


-- El código siguiente va en un fichero nuevo
-- de nombre <nombre matriz>-<nombre subprograma>.adb
-- ejemplo_separado-intercambia.adb en este caso
separate(Ejemplo_Separado)
procedure Intercambia(A,B: in out Integer) is
   ...
begin
   ...
end Intercambia;
```

Otra ventaja que tiene la separación de subprogramas (además de evitar programas muy extensos) es que cada uno de los ficheros separados puede tener sus propias cláusulas **with**. Las unidades separadas se compilan siempre después de compilar la unidad matriz. Esta separación de subprogramas puede realizarse hasta cualquier nivel de profundidad.

Ejemplo 14: estudie, compile y ejecute el fichero *esfera2.adb*.

8.2. PARÁMETROS DE LOS SUBPROGRAMAS

El número, tipo y clase de los parámetros formales se declara en la *lista de parámetros formales* que se pone justo después del nombre del subprograma en la cabecera o declaración del mismo.

La lista de parámetros formales se delimita por paréntesis y se compone de una o varias sub-listas de parámetros del mismo tipo, separadas por punto y coma (;). Cada sub-lista está formada por una secuencia de nombres separados por comas (,), seguida de dos puntos (":") a continuación de los cuales se pone el tipo de los parámetros de la secuencia:

```
procedure Muchos_Parámetros(A, B : in Integer; C : out Float;
                            L, M : in out Integer);
```

8.2.1. Clases de parámetros

En función del sentido en el que se transfiere la información, los parámetros pueden ser:

1. De entrada (**in**). Del invocado al invocante. Valor por defecto si no se especifica otra cosa. Las funciones sólo admiten este modificador al tipo de parámetros. Los parámetros de entrada no se pueden modificar en el subprograma, que los considera constantes.
2. De salida (**out**). Del invocante al invocado. Sólo admitido por los procedimientos. Son parámetros que se van a usar para devolver información, y por tanto no tiene sentido que el procedimiento invocado los lea (contienen basura). Mediante el uso de este parámetro, los procedimientos pueden devolver muchos valores.
3. De entrada/ salida (**in out**). En ambos sentidos. Sólo admitido por los procedimientos. En el procedimiento invocado tiene sentido leer la variable antes de modificarla (contiene un valor útil).

8.2.2. Correspondencia entre parámetros reales y formales

Cuando se llama a un subprograma, hay que especificar los parámetros reales sobre los que se quiere que actúe. Los parámetros reales ocuparán en la ejecución el lugar que en la declaración ocupan los parámetros formales correspondientes (los de la cabecera). La correspondencia entre parámetros reales y formales se establece normalmente por posición, lo que quiere decir que el primer parámetro real corresponde al primero formal, el segundo al segundo y así sucesivamente. En el siguiente ejemplo se llama al procedimiento "*Intercambia*" haciendo corresponder el parámetro real X al formal A, y el real Y al formal B:

```
...
Intercambiar(X,Y); -- A toma el valor de X y B el de Y
...
```

La correspondencia entre parámetros formales y reales también puede hacerse por nombre (notación nominal): <*nombre del parámetro formal*> => <*nombre del parámetro real*>.

Cuando se usa esta notación no se utiliza la notación posicional, es decir, da igual el orden:

```
...
Intercambiar (B=>Y, A=>X); -- A toma el valor de X y B el de Y
...
```

A un parámetro formal de entrada se le puede hacer corresponder como parámetro real cualquier expresión del tipo adecuado; a los parámetros formales de salida o de entrada/ salida sólo les pueden corresponder parámetros reales que sean variables.

8.2.3. Parámetros por omisión

Es posible especificar un valor por defecto para los parámetros de entrada asignándolo detrás del nombre del tipo.

```
procedure Defecto(A,B: in Integer := 0; C: in Float:= 0.0);
```

Si un parámetro formal tiene un valor por defecto no es necesario pasar ningún parámetro real, pero si después hay parámetros no omitidos, habrá que utilizar la notación nominal.

```
Defecto(X,Y);
   -- A toma el valor de X, B el de Y y C toma el valor 0.0
Defecto(X);
     -- A toma el valor de X, B el valor 0 y C el valor 0.0
Defecto(X,C => Z);
      -- A toma el valor de X, B el valor 0 y C el de Z
```

9. PAQUETES

Un paquete (**package**) es una unidad de compilación que agrupa un conjunto de entidades relacionadas. Su aplicación más corriente es para realizar la declaración de un tipo de datos junto con las operaciones primitivas para su manejo. De esta forma el tipo se puede utilizar desde el exterior manteniendo ocultos los detalles de funcionamiento. Y lo que es más importante, se evita que el programador pueda utilizar mal o modificar el código original. En otras palabras, un paquete es un mecanismo de encapsulamiento y ocultación de información especialmente útil para definir tipos abstractos de datos. Un paquete se puede utilizar también para agrupar una serie de subprogramas, como por ejemplo una librería matemática.

9.1. TIPOS PRIVADOS Y LIMITADOS

Una declaración de *tipo privado* es una en la que como descripción se utiliza la palabra reservada **private**. La declaración proporciona una vista restringida de un tipo, que está definido en otra parte del código. Sólo pueden definirse tipos privados dentro de un paquete.

```
type T_Privado is private;
type T_Limitado is limited private;
```

En el ámbito en el que es visible un tipo privado sólo admite el operador de asignación (:=) y los operadores relacionales de igualdad (=, /=). Los tipos **limited private** no definen ninguno de estos operadores (los tiene que definir el programador, si quiere).

El tipo de dato que se define en un paquete (junto con sus operaciones asociadas) suele ser generalmente de este tipo. De este modo se consigue "encapsular" el contenido del tipo, no permitiendo que nadie ajeno al paquete pueda tener acceso a su implementación. Esto se hace así para evitar que, por error o a propósito, se manipule el contenido del tipo erróneamente.

Un tipo es limitado (y por tanto no tiene operaciones predefinidas) si está declarado como **limited private**, si es derivado de uno limitado, si es una tarea (**task**) o si es un tipo protegido (**protected type**). Los tipos compuestos son limitados si tienen algún elemento limitado.

9.2. DEFINICIÓN Y USO

Un paquete se divide generalmente en dos partes: *especificación* e *implementación*, que se sitúan en ficheros diferentes con las extensiones respectivas "*.ads" y "*.adb". La *especificación* dicta el aspecto exterior del paquete y cumple la misma misión que la cabecera en los subprogramas: informar al usuario del paquete de qué subprogramas y de qué tipos de datos dispone. Es decir, es lo que utilizan los usuarios del paquete. La

implementación de un paquete sólo incumbe (y es diseñada) por el desarrollador del mismo.

El lenguaje Ada define cuatro paquetes base: *System* (adaptación a la arquitectura de la máquina en concreto), *Standard* (definiciones básicas del lenguaje, usado por defecto), *Interfaces* (funciones para comunicarse con otros lenguajes) y *Ada* (a partir de aquí se definen el resto de paquetes de utilidad). Consulte el manual de referencia del lenguaje para obtener una descripción de estos paquetes. Salvo estos nombres, el programador puede usar cualquier otro para sus propios paquetes.

9.2.1. Especificación

La estructura de la especificación de un paquete es la siguiente:

```
package <nombre_de_paquete> is
   <elementos públicos>
[private]
   <elementos privados>
end <nombre_de_paquete>;
```

La parte anterior a la palabra *private* constituye la interfaz pública de los subprogramas y tipos de datos (servicios al fin y al cabo) ofertados por el paquete, que son accesibles a quien lo use con **with**. La parte que sigue constituye la parte privada de la especificación, en la que se especifica la estructura de datos con que se implementa un tipo ofrecido en la interfaz u otros elementos similares, como constantes de dicho tipo. En esta parte todos los tipos declarados como **private** o **limited private** tienen que definirse completamente (el compilador necesita saber cuánto espacio reservar). La palabra *private* puede omitirse de la especificación, en cuyo caso se entiende que la parte privada de la especificación es nula.

La especificación de un paquete puede contener únicamente la definición de tipos y constantes. En este caso no es necesario que el paquete tenga implementación.

Ejemplo 15: busque el fichero *a-ngelfu.ads (Ada.Numerics.Generic_Elementary_Functions)* y edítelo. En él encontrará la definición de un paquete matemático para tipos en coma flotante. No se preocupe por las cosas que no entienda. Observe como sigue con la filosofía de "definir un tipo de dato y las operaciones que pueden realizarse con él", solo que en este caso el tipo de dato ya estaba predefinido.

Ejemplo 16: estudie, compile y ejecute el ejemplo *esfera3*. Este ejemplo muestra cómo reorganizar el código ya visto en *esfera* y *esfera2* en un paquete.

La siguiente especificación es de un paquete llamado *Pack_Pila_De_Enteros*, que ofrece un tipo llamado *Pila* y cuatro operaciones primitivas para manejar pilas de valores de tipo *Integer*:

```
package Pack_Pila_De_Enteros is
  type T_Pila is limited private; --¿Qué significa esto?
  procedure Push(P : in out T_Pila; E : in Integer);
```

```
   procedure Pop(P : in out T_Pila; E: out Integer);
   function Esta_Vacia(P : T_Pila) return Boolean;
   function Esta_Llena(P : T_Pila) return Boolean;
   procedure Imprimir(P : in out T_Pila);
private
   type T_Pila is record
      Pila: array (20..50) of Integer;  -- ¿por qué falla?
      Num_Elementos : Integer := 0;
   end record;
end Pack_Pila_De_Enteros;
```

Ejercicio 17: mejore la especificación anterior. Preste especial atención a la definición del tipo T_Pila. Ada no permite la definición de variables cuyo tipo es anónimo en records.

9.2.2. Cuerpo

La especificación de un paquete necesita un cuerpo si contiene declaraciones que requieran completarse (por ejemplo, la declaración de una función requiere su implementación). La estructura de un **package body** es:

```
package body <nombre_de_paquete> is
   <desarrollo del paquete>
end <nombre_de_paquete>;
```

Todas las declaraciones hechas en la especificación del paquete se pueden usar en la implementación (**body**) aunque estén en la parte **private**; pero las declaraciones hechas en el **package body** sólo se pueden usar en dentro del cuerpo y nunca desde fuera.

Ejemplo 17: busque el fichero de implementación del paquete del ejemplo anterior, *a-ngelfu.adb,* y edítelo para ver cómo es el código. Este paquete utiliza las librerías de C.

A continuación se muestra el **package body** del paquete *"Pack_Pila_De_Enteros"*:

```
package body Pack_Pila_De_Enteros is
   procedure Push(P: in out T_Pila; E: in Integer) is
   begin
      --sentencias
   end Push;

   procedure Pop(P : in out T_Pila; E: out Integer) is
   begin
     --sentencias
   end Pop;

   function Esta_Vacia(P: T_Pila) return Boolean is
   begin
     --sentencias
   end Esta_Vacia;
```

```
  function Esta_Llena(P: T_Pila) return Boolean is
  begin
    --sentencias
  end Esta_Llena;

  procedure Imprimir(P: in out T_Pila) is
  begin
    --sentencias
  end Imprimir;
end Pack_Pila_De_Enteros;
```

Una vez compiladas la especificación y la implementación de un paquete, se puede hacer uso del mismo en otras unidades, simplemente incluyendo el nombre del paquete en sus cláusulas de contexto.

```
with Text_Io, Ada.Integer_Text_Io, Pack_Pila_De_Enteros;
use  Text_Io, Ada.Integer_Text_Io, Pack_Pila_De_Enteros;
procedure Prueba is
  P: T_Pila;
  Valor : Integer;
begin
  for I in 1..5 loop
    if not Esta_Llena (P) then
      Push(P,I);
    end if;
  end loop;
  while not Esta_Vacia(P) loop
    Pop(P, Valor);
    Put(Valor);
    New_Line;
  end loop;
end Prueba;
```

Se recuerda en esta parte que el ámbito de validez de una declaración (válida para cualquier elemento del lenguaje) se extiende desde el punto en que es declarada hasta el punto en que acaba el bloque en que está definida.

Ejercicio 18: complete el cuerpo del paquete *Pila_De_Enteros* y escriba un programa que lo use (como el anterior). Convierta el bucle **for** en un bucle **while**.

Modificando ligeramente el ejemplo de la pila, se podría realizar un paquete que proporcionara al usuario una única pila, compartida entre el resto de unidades de compilación:

```
package Pack_Pila_Unica_De_Enteros is
  procedure Push(E: in Integer);
  procedure Pop (E: out Integer);
```

```
   function Esta_Llena return Boolean;
   function Esta_Vacía return Boolean;
   procedure Imprimir;
end Pack_Pila_Unica_De_Enteros;
```

La declaración del tipo "Pila" debe colocarse en el cuerpo del paquete (**package body**), en donde se encuentra protegida de acceso por parte del exterior. Observe que también se han tenido que modificar ligeramente los parámetros de los subprogramas que forman el paquete.

Ejercicio 19: escriba el cuerpo del paquete y un procedimiento de prueba para esta nueva especificación del paquete *Pila_De_Enteros*.

9.3. HERENCIA DE PAQUETES

Se pueden derivar paquetes a partir de otros existentes, creando lo que se llama un paquete "hijo". Normalmente, lo que se pretende conseguir es añadir una nueva funcionalidad sin tener que modificar el paquete original (y sin tener que volver a recompilarlo). Es como si el código del paquete hijo estuviera escrito dentro del paquete padre, pero que se hubiera sacado a otro fichero. El siguiente ejemplo crea un paquete hijo de *Pack_Pila_De_Enteros* con el fin de proporcionar una operación que permita obtener una copia de una pila. Además, se define un procedimiento para vaciar una pila:

```
package Pack_Pila_De_Enteros.Copiable is
   procedure Vaciar (P: in out T_Pila);
   procedure Copiar (Pe: in T_Pila; Ps: out T_Pila);
                  --no lleva parte 'private'!!
end Pack_Pila_De_Enteros.Copiable;
```

Un paquete hijo tiene acceso a todo el código declarado en la especificación de su padre sin necesidad de incluirlo con **with**. Esto incluye la parte **private** del padre pero no al cuerpo del padre, que sigue permaneciendo oculto al hijo. De ahí que a veces se coloquen subprogramas dentro de la parte privada de la especificación de un paquete: para que los puedan invocar los hijos.

Una unidad que requiera toda la funcionalidad de los dos paquetes (padre e hijo) necesitará incluirlos a ambos con **with**.

Ejercicio 20: complete la definición de *Pack_Pila_De_Enteros.Copiable* y escriba un procedimiento de prueba que defina dos pilas de este tipo. Copie el contenido de una pila a la otra y luego vacíe las dos.

10. UNIDADES GENÉRICAS

Ada permite crear unidades genéricas, es decir, con parámetros que se pueden concretar para diferentes instancias de la unidad. De esta manera se facilita la reutilización de un código que se escribe una sola vez. Para ello basta anteceder la declaración de la unidad correspondiente con la palabra **generic**; los parámetros formales de la unidad se sitúan en la zona comprendida entre la palabra **generic** y el comienzo de la declaración de la unidad. Sólo los subprogramas y los paquetes admiten esta parametrización.

```
generic
   <parámetros formales genéricos>
<unidad genérica> --subprograma o paquete
```

10.1. PARÁMETROS FORMALES GENÉRICOS

Los parámetros formales genéricos pueden ser: variables, tipos o subprogramas. Al igual que los parámetros formales de los subprogramas, algunas clases de parámetros formales genéricos pueden tener un valor por defecto, en cuyo caso se podrá omitir el parámetro al instanciar la unidad. Los parámetros formales genéricos se declaran de distinta forma según su naturaleza:

1. **Objetos formales.** Se declaran normalmente: *<nombre>: <tipo> := <valor por defecto>*, siendo el valor por defecto opcional.

```
Valor1: Integer;
Valor2: Positive := 1;
```

2. **Tipos formales privados.**

```
type Tipo1 is private;
type Tipo2 is limited private;
```

3. **Tipos formales escalares.**

```
type T_Discreto is (<>);
type T_Entero is range <>;
type T_Modular is mod <>;
type T_Real_Float is digits <>;
type T_Real_Fijo is delta <>;
type T_Real_Decimal is delta <> digits <>;
```

4. **Tipos formales array.** Hay que especificar el tipo de los índices y el tipo de los elementos, que suelen ser a su vez, aunque no tienen porqué, parámetros formales genéricos declarados previamente:

```
type T_Elemento is private;
type T_Indice is (<>);
type Vector is array (T_Indice range <>) of T_Elemento;
```

5. **Subprogramas.** Se utiliza la palabra **with** precediendo a la definición de la cabecera del subprograma que se espera:

```
type T_Elemento is private;
with procedure Acción(X : in T_Elemento);
```

Se puede especificar un nombre por defecto para el subprograma pasado por parámetro, utilizando "*is <nombre>*":

```
with procedure Acción (X : in T_Elemento) is Escribir;
```

Si se quiere que el nombre por defecto sea el mismo que el del parámetro formal, se pondrá como nombre "<>":

```
with procedure Acción (X : in T_Elemento) is <>;
```

6. **Paquetes.** Se utiliza la palabra **with** precediendo a la instanciación del paquete que queremos como parámetro formal.

```
with package <Nombre_Instancia> is new  <Nombre_Genérico> (<>);
```

10.2. INSTANCIACIÓN DE UNIDADES GENÉRICAS

Las unidades genéricas representan una plantilla mediante la cual se indica al compilador cómo poder crear (en distintas situaciones configuradas por los parámetros) unidades no genéricas que responden al mismo funcionamiento general que la unidad genérica. De esta forma, las unidades genéricas no se usan directamente, sino que se tienen que instanciar previamente. Para crear una instancia de una unidad genérica hay que especificar un nombre para la unidad no genérica que el compilador va a construir y los parámetros reales genéricos a utilizar en el lugar de los formales:

```
<tipo unidad> <nombre de la instancia> is new <nombre unidad
        genérica> (<parámetros reales genéricos>);

package Ada.Integer_Text_Io is new
        Ada.Text_Io.Integer_Io(Integer);
```

10.3. SUBPROGRAMAS GENÉRICOS

En muchas situaciones un mismo problema se plantea en contextos diferentes, de forma que el algoritmo que lo resuelve es el mismo, salvo por los detalles propios de cada contexto; por ejemplo, si se quiere ordenar un array, las acciones que habrá que realizar

son las mismas independientemente del tipo de los elementos del array (siempre que sean ordenables), de su número, e incluso de la operación que se utilice para compararlos.

Ejercicio 21: escriba un programa para ordenar un array (4..10) de Integer y mostrarlo por pantalla. Posteriormente, escriba un programa para ordenar un array (30..40) de Integer y mostrarlo por pantalla. Tras éste, escriba un programa para ordenar un array (4..10) de Float y mostrarlo por pantalla. Por último, escriba un programa para ordenar un array (30..40) de Float y mostrarlo por pantalla. Se parecen mucho, hay algunas cosas que se pueden generalizar, pero no se puede reutilizar el mismo código cada vez.

Un subprograma genérico es un subprograma que se escribe dejando todos estos detalles como *parámetros formales genéricos*, a concretar cuando haga falta, de tal manera que se podrá luego utilizar en distintas situaciones sin tener que rescribir cada vez el algoritmo completo. En C se llaman plantillas (*templates*), quizá un nombre más acertado. Sin embargo un genérico no es ejecutable, ya que no es más que una plantilla, en la que se establecen los huecos. Cuando se instancia, rellenando los huecos, es cuando de verdad se obtiene código ejecutable. El siguiente ejemplo ilustra la forma de hacerlo.

```
generic
   type Tipo is private;
   with function "<"(X,Y: Tipo) return Boolean;
   type Indice is (<>);
   type Lista is array(Indice range <>) of Tipo;
procedure Ordenar(L: in out Lista);

procedure Ordenar(L: in out Lista) is
   Menor: Indice;
   Aux : Tipo;
begin
   for I in L'First..Indice'Pred(L'Last) loop
      Menor := I;
      for J in Indice'Succ(I)..L'Last loop
         if L(J) < L(Menor) then
            Menor := J;
         end if;
      end loop;
      if Menor / = I then
         Aux := L(I);
         L(I) := L(Menor);
         L(Menor) := Aux;
      end if;
   end loop;
   return;
end Ordenar;
```

Una vez que se tiene escrito el subprograma genérico, se pueden declarar instancias del mismo con diferentes parámetros reales genéricos, lo que hará que el compilador cree

los subprogramas correspondientes. Para ilustrarlo, se proporciona un procedimiento llamado *"Prueba"* en cuya cláusula de contexto se incluye la unidad *"Ordenar"*. El procedimiento *"Prueba"* declara dos instancias del procedimiento *"Ordenar"*, aplicadas a arrays de elementos de tipo *Integer* y rango *Integer* (o *Docena*), que se diferencian sólo en la operación de ordenación. Cuando el compilador elabore estas declaraciones, se dispondrá de dos procedimientos no genéricos (*"Ordena_Vector_Ascendente"* y *"Ordena_Vector_Descendente"*) que podrán ser utilizados, normalmente, igual que si se hubiesen escrito explícitamente.

```
with Ada.Text_Io,Ordenar;
use Ada.Text_Io;
procedure Prueba is
   subtype Docena is Integer range 1..12;
   type Vector is array(Integer range <>) of Integer;
   procedure Ordena_Vector_Ascendente is new
            Ordenar(Integer,"<",Integer,Vector);
   procedure Ordena_Vector_Descendente is new
            Ordenar(Integer,">",Docena,Vector);
   V: Vector(Docena);
   ...
begin
    Ordena_Vector_Ascendente(V);
    ...
end Prueba;
```

Para hacernos una idea de los que sucede, imagine que al instanciar un subprograma genérico el compilador escribe internamente de nuevo el código del genérico, creando un nuevo subprograma, completando los huecos dejados por los parámetros genéricos (virtuales) con los parámetros reales que se pasan. Como puede deducirse, el mecanismo es similar al de sustitución de parámetros al invocar subprogramas

Ejercicio 22: escriba un programa único para ordenar los arrays del ejercicio anterior haciendo uso de un procedimiento genérico "ordenar".

10.4. PAQUETES GENÉRICOS

Un paquete genérico es el que posee parámetros que, al ser fijados, permite crear instancias no genéricas aplicables en situaciones diferentes, reutilizando el código sin tener que rescribirlo. Por ejemplo, en el caso de las pilas de enteros del ejemplo previo, es evidente que tanto la estructura de datos del tipo *"Pila"*, como los algoritmos de manipulación que la acompañan, no dependen en realidad de que los elementos apilados sean de tipo *Integer*. En consecuencia, se puede escribir un paquete genérico parametrizando el tipo de los elementos, de forma que luego se puedan crear instancias de pilas que acomoden distintos tipos de elementos.

Especificación: se añade la lista de parámetros genéricos formales y se sustituyen todas las apariciones de *Integer* por *T_Elemento*, el parámetro virtual o genérico del paquete:

```
generic
   type T_Elemento is private;
   type T_Indice is mod <>;
package Pack_Pila_Generica is
   type T_Pila is limited private;
   procedure Push(P: in out T_Pila;E: in T_Elemento);
   procedure Pop(P: in out T_Pila; E : out T_Elemento);
   function Esta_Vacia(P: T_Pila) return Boolean;
   function Esta_Llena(P: T_Pila) return Boolean;
private
   Tam_Pila : Integer := T_Indice'Modulus;
   type At_Pila is array (T_Indice) of T_Elemento;
   type T_Pila is record
      Pila: At_Pila;
      Num_Elementos : T_Indice := 0;
      Primero : T_Indice := T_Indice'First;
      Ultimo : T_Indice := T_Indice'First;
   end record;
end Pack_Pila_Generica;
```

Implementación: bastaría con sustituir todas las apariciones de *Integer* por *T_Elemento*.

Utilización: el nuevo paquete, *"Pack_Pila_Generica"*, aparece en la sentencia **with** de la cláusula de contexto, pero no así en la *use*, puesto que al ser genérico es sólo una plantilla a partir de la cual definir instancias y no es en sí mismo una unidad utilizable directamente. La definición de una instancia se hace especificando un nombre para el paquete no genérico al que dará lugar y los valores reales correspondientes a los parámetros (al menos de aquellos que no tengan un valor por defecto):

```
with Ada.Text_Io, Ada.Integer_Text_Io, Pack_Pila_Generica;
use Ada.Text_Io, Ada.Integer_Text_Io;

procedure Prueba_Pila_Generica is
   type T_Indice_Array is mod 20;
   package Pack_Pila_De_Enteros is new Pack_Pila_Generica
      (T_Elemento => Integer, T_Indice => T_Indice_Array);
   use Pack_Pila_De_Enteros;
   Pila: Pack_Pila_De_Enteros.T_Pila;
   Elem : Integer;
begin
   for I in 1..5 loop
      Push(Pila,I);
   end loop;
   while not Esta_Vacia(Pila) loop
      Pop(Pila, Elem);
      Put(Elem);
      New_Line;
   end loop;
end Prueba_Pila_Generica;
```

Ejercicio 23: complete la definición de *Pack_Pila_Generica* y escriba un procedimiento de prueba que defina dos instancias de la pila (una para *Integer* y otra para *Float*). No haga uso de la claúsula *use* interna.

En cuanto a la herencia de paquetes genéricos, un paquete hijo de un paquete genérico tiene obligatoriamente que ser genérico (aunque no precise parámetros, sí los precisa su padre). Pero de un paquete no genérico sí que se puede derivar un paquete genérico (y uno no genérico, claro).

Para instanciar un paquete hijo genérico primero se tiene que haber creado una instancia del paquete padre (siempre que éste sea genérico). Posteriormente se instancia el paquete hijo genérico a partir de la instancia concreta anterior del paquete padre.

```
generic
    -- aunque no tiene parámetros genericos
package Pack_Pila_Generica.Copiable is
  procedure Vaciar (P: in out T_Pila);
  procedure Copiar (Pe: in T_Pila; Ps: out T_Pila);
                -- ¡no lleva parte 'private'!
end Pack_Pila_Generica.Copiable;
```

La forma de instanciar correctamente un paquete hijo genérico es la siguiente:

```
with Ada.Text_Io, Ada.Integer_Text_Io, Pack_Pila_Generica,
       Pack_Pila_Generica.Copiable;
use Ada.Text_Io, Ada.Integer_Text_Io;

procedure Prueba_Pila_Generica_Copiable is
  type T_Indice_Array is mod 20;
  package Pack_Pila_De_Enteros is new Pack_Pila_Generica
    (T_Elemento => Integer, T_Indice => T_Indice_Array);
  use Pack_Pila_De_Enteros;
  package Pack_Pila_De_Enteros_Copiable is new
        Pack_Pila_De_Enteros.Copiable;
   -- observe cómo se utiliza la instancia del paquete padre
  use Pack_Pila_De_Enteros_Copiable;
  Pila: Pack_Pila_De_Enteros.T_Pila;
  Elem : Integer;
  Pila2: Pack_Pila_De_Enteros.T_Pila;
begin
  for I in 1..5 loop
    Push(Pila,I);
  end loop;
  Copiar (Pila, Pila2);
  while not Esta_Vacia(Pila) loop
    Pop (Pila, Elem);
    Put(Elem);
    New_Line;
```

```
   end loop;
end Prueba_Pila_Generica_Copiable;
```

Ejercicio 24: complete la definición de *Pack_Pila_Generica.Copiable* y escriba un procedimiento de prueba para probarlo con una pila de *Character*. No haga uso de la claúsula *use* interna.

11. EXCEPCIONES

En la ejecución de un programa pueden darse muchas situaciones inesperadas: dispositivos que fallan, datos que entran incorrectamente, valores fuera del rango esperado, etc. Incorporar todas las posibilidades de fallo y casos extraños en la lógica de un algoritmo puede entorpecer su legibilidad y su eficiencia; no tenerlas en cuenta produce que los programas aborten de forma incontrolada y puedan generarse daños en la información manejada.

Los mecanismos de control de excepciones permiten contemplar este tipo de problemas separándolos de lo que es el funcionamiento "normal" del algoritmo; una *excepción* es una manifestación de un tipo de error e identifica la ocurrencia de una situación anormal. En el estándar de Ada están definidas las siguientes excepciones básicas:

Constraint_error	Ocurre cuando se intenta asignar a una variable un valor no válido o cuando se intenta acceder a una posición de un *array* fuera del rango permitido.
Program_error	Ocurre en situaciones extrañas cuando parte de un programa no es accesible o cuando se alcanza el **end** de una función sin encontrar un **return**.
Storage_error	Ocurre cuando se agota la memoria disponible.
Tasking_error	Está relacionado con errores en programas que utilicen programación concurrente.

Además, un gran número de librerías del lenguaje definen sus propias excepciones específicas y Ada permite también al usuario definir sus propias excepciones.

El mecanismo para manejar excepciones debe utilizarse únicamente en situaciones realmente excepcionales y para tratar errores inesperados. El programador debe intentar anticiparse a las posibles causas de fallo para tratarlas correctamente, dejando las excepciones para casos no previstos. El tratamiento de excepciones, además, no es gratuito sino que penaliza la velocidad ejecución del programa.

11.1. DECLARACIÓN

El programador puede declarar sus propias excepciones para identificar problemas particulares que puedan surgir en sus programas. Las declaración está sujeta a las reglas generales de declaración y consta de: una lista de identificadores, dos puntos (":") y la palabra **exception**.

```
Error1, Error2: exception; --se declaran dos excepciones
```

11.2. LANZAMIENTO

Al hecho de activar una excepción cuando ocurre la situación para la que está prevista se le conoce como "lanzar" la excepción; básicamente esto consiste en interrumpir la ejecución "normal" del programa e intentar localizar una sección de código capaz de hacerse cargo de la excepción y resolverla. Las excepciones predefinidas se lanzan automáticamente cuando ocurre la situación para la que se han previsto; se pueden también lanzar las excepciones predefinidas o, más probablemente, las declaradas en el programa usando **raise**.

```
raise Error1; --lanza la excepción Error1;
```

Suele ser bastante común el uso del *relanzamiento* de excepciones: el bloque en que se produce realiza un primer tratamiento y, dependiendo de su naturaleza, decide volverla a lanzar para que el código que lo invocó realice también su propio tratamiento. Y así sucesivamente.

11.3. MANEJO

Un *manejador de excepciones* es un conjunto de instrucciones destinadas a ejecutarse en respuesta al lanzamiento de una excepción. Los manejadores de excepciones empiezan con una cláusula **when** seguida de la lista de excepciones a las que responde unidas por el operador "| " (or) y a continuación las instrucciones a ejecutar.

```
when Error1 |  Error2 =>
  --aquí se ponen las instrucciones a ejecutar
  --cuando ocurra la excepción Error1 o la Error2
  ...
```

Los manejadores se agrupan en una sección situada al final de cualquier bloque **begin**...**end** que contenga el algoritmo cuyas excepciones se quiere controlar; esta sección se abre con la palabra **exception**.

```
begin
  --Instrucciones "normales" en las que puede haber
  --problemas
  ...
exception
  --manejadores de excepciones
  when nombre_de_la_excepción =>
    -- instrucciones a ejecutar cuando ocurre la excepción
end;
```

Durante la ejecución de un algoritmo, cuando se lanza una excepción el control es transferido a la sección de excepciones situada al final del bloque **begin**...**end** en el que se engloba y allí se busca un manejador que tenga en su lista la excepción lanzada y se ejecuta. Si no se encuentra ningún manejador que tenga a la excepción en su lista o si el

bloque **begin**...**end** no tiene sección de tratamiento de excepciones, la búsqueda del manejador continúa en el siguiente bloque activo más externo (aquel que inició la ejecución del actual). Mientras no se encuentre un manejador apropiado la búsqueda continúa hasta alcanzar el procedimiento principal; si en éste tampoco se encuentra un manejador acorde a la excepción lanzada, se aborta la ejecución del programa. Cuando un programa aborta se imprime en pantalla información acerca de qué excepción ha ocurrido y de dónde se ha producido, por si pudiera resultar de alguna utilidad.

Como ya se ha dicho, cuando se produce una excepción en cualquier punto dentro de un bloque, la ejecución salta directamente a la zona de tratamiento de excepciones. A partir de ahí el código sigue ejecutándose normalmente, es decir, una vez tratada la excepción NO se vuelve al punto en que saltó. La ejecución continúa normalmente *desde el final* del bloque **begin**...**end** en que se produjo. De ahí la utilidad de anidar bloques de declaración: para que el programa no acabe en caso de que se produzca una excepción.

Por último, se puede añadir a los manejadores de excepciones una última alternativa cuyas instrucciones se ejecutarán si la excepción elevada no es tratada en ninguno de los manejadores anteriores:

```
begin
  --Instrucciones "normales" en las que puede haber
  --problemas
exception                    --manejadores de excepciones
  when nombre_de_la_excepción =>
    -- instrucciones a ejecutar cuando ocurre la excepción
  when others =>
    -- instrucciones a ejecutar para cualquier excepción
    -- que se eleva y no es tratada en algun manejador
end;
```

Ejemplo 18: En el ejemplo se muestra cómo se controla una excepción de forma que el programa se recupere y continúe ejecutándose. Para ello se ha declarado una excepción *Divide_por_cero* y dos funciones *Divide1* y *Divide2*. *Divide1* ilustra cómo, tras capturar una excepción y hacer algunas operaciones, se puede relanzar para que un bloque más externo continúe su manejo. *Divide2* ilustra cómo se lanza una excepción definida por el programador.

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Excepciones is
  Divide_Por_Cero: exception; -- declaración de excepción
  Res : Float;
  function Divide1(X, Y: in Integer) return Float is
    R: Float;
  begin
    R := Float(X/ Y);
    return R;
  exception
    when Constraint_error =>
      Put_Line("El divisor es cero");
```

```
    raise; -- se relanza Constraint_error
  end Divide1;
  function Divide2(X, Y: in Integer) return Float is
    R: Float;
  begin
    if Y = 0 then raise Divide_Por_Cero;
    else
      R := Float(X)/ Float(Y);
      return R;
    end if;
  end Divide2; ;
begin
  Put_Line("DIVIDE 1");
  Res := Divide1(4,0);
  Put_Line("DIVIDE 2"); -- ¡Esto no se ejecuta!
  Res := Divide2(4,0);
exception
  when Constraint_error| Divide_Por_Cero =>
      Put_Line("No se puede dividir por cero");
  when others =>
      Put_Line(" Error desconocido");
end Excepciones;
```

12. CONCURRENCIA EN ADA

Se dice que un sistema es concurrente cuando se compone de un conjunto de procesos *secuenciales y autónomos* que se ejecutan de forma (*aparentemente*) paralela. Como ejemplo de paralelismo cotidiano considere que los miembros de una familia van a comprar productos para celebrar una comida. La compra de cada producto podría representarse como en el siguiente código mediante llamadas a procedimientos tales como Comprar_Carne, Comprar_Ensalada y Comprar_Vino:

```
procedure Comprar is
begin
  Comprar_Carne;
  Comprar_Ensalada;
  Comprar_Vino;
end Comprar;
```

Sin embargo, esta solución se corresponde con una compra secuencial. Sería mucho más eficiente que cada miembro de la familia se encargara de comprar un producto y se dieran cita en el aparcamiento. Esta solución concurrente podría ser representada por el siguiente código:

```
procedure Comprar is
  task Comprar_Ensalada; -- especificación de una tarea
  task body Comprar_Ensalada is  -- cuerpo de una tarea
  begin
    Comprar_Ensalada;
  end Comprar_Ensalada;

  task Comprar_Vino;     -- especificación de otra tarea
  task body Comprar_Vino is       -- cuerpo de la tarea
  begin
    Comprar_Vino;
  end Comprar_Vino;
begin
   Comprar_Carne; -- se ejecuta en paralelo con las tareas
end Comprar;
```

En el ejemplo las tres compras se ejecutan en paralelo. Los sistemas de tiempo real son concurrentes por naturaleza ya que realizan varias actividades *simultáneamente*, por ello es necesario que su programación se lleve a cabo para que se ejecuten en paralelo. Ada soporta mecanismos de concurrencia, en concreto las actividades concurrentes reciben el nombre de tareas (**task**) y se declaran explícitamente en cualquier zona declarativa (entre el **declare** y el **begin** de un bloque). Las tareas se pueden declarar como tipos *anónimos* (y por tanto son únicos) o como variable de un tipo tarea.
La declaración de una tarea (o de un tipo tarea) tiene dos partes: *especificación* que contiene la interfaz visible de la misma y *cuerpo* que contiene las instrucciones que

ejecuta. En el siguiente ejemplo se muestra la declaración de la especificación y cuerpo de una tarea *A* :

```
procedure Ejemplo is
                    -- principio zona declarativa
  task A; -- especificación de la tarea
  task body A is -- cuerpo de la tarea
     -- declaraciones locales
  begin
     -- secuencia de instrucciones
  end A;
                    -- final zona declarativa
begin
   -- A empieza a ejecutarse concurrentemente aquí con las
   -- instrucciones de Ejemplo, que podría ser 'null;'
end Ejemplo; -- Ejemplo no termina hasta que termina A
```

Ejercicio 25: escriba el procedimiento comprar de forma que cada tarea imprima en pantalla un mensaje diez veces y compruebe su ejecución. ¿Qué sucede? ¿Y si se imprime 500 veces?

Las tareas se empiezan a ejecutar antes que el cuerpo de la parte declarativa donde se declaran (antes del **begin** en el **procedure** Ejemplo). La declaración de una tarea *A* como en el ejemplo anterior es una declaración de tarea anónima, mientras que la declaración de un tipo tarea permite disponer de una plantilla para crear tareas similares. La declaración de un tipo tarea funciona exactamente igual y está sometida a las mismas limitaciones que la declaración de cualquier tipo (por ejemplo, *Integer*). Se recuerda que Ada considera las tareas como tipos limitados, es decir, sin operaciones predefinidas (:=, =, /=).

```
task type TT_Prueba;      -- declaración de un tipo tarea
B,C,D : TT_Prueba;          -- declaración de una tarea
task body TT_Prueba is       -- cuerpo del tipo tarea
  -- secuencia de instrucciones
end TT_Prueba;
```

En la práctica, una tarea deberá realizar alguna actividad indefinidamente mientras dure la ejecución del programa. Para ello, se hace uso de las estructuras de repetición en el cuerpo de la tarea como en el siguiente ejemplo:

```
task body Tarea_Tipica is
begin
  while Continuar loop
     -- secuencia de instrucciones
     -- ojo! Si salta una excepcion la tarea acaba!
  end loop;
end Tarea_Tipica;
```

```
task body Tarea_Tipica_Resistente is
begin
  while Continuar loop
    begin
      -- secuencia de instrucciones
      -- preparada contra excepciones
    exception
      -- tratamiento de excepciones
    end;
  end loop;
end Tarea_Tipica_Resistente;
```

Las tareas y los tipos tarea son unidades de programa, por lo que deben ser declarados en un paquete o subprograma. Ada trata las tareas como si fueran tipos de datos normales, aunque con características especiales. Generalmente, cuando se trata de una tarea única (anónima) y no es necesario que su interfaz sea visible desde otros puntos del programa, se declaran en el cuerpo de un paquete de la siguiente manera:

```
package body Ejemplo is
  task A; -- especificación de la tarea
  task body A is -- cuerpo de la tarea
  -- declaraciones locales
  begin
    -- secuencia de instrucciones
  end A;
begin
    -- 'A' empieza a ejecutarse concurrentemente aquí con las
    -- instrucciones de 'Ejemplo'
end Ejemplo;
```

Si se trata de un tipo tarea probablemente interese que dicho tipo sea visible fuera del paquete, para poder declarar variables de este tipo en otras unidades de programa. En este caso se puede declarar en la especificación del paquete como en el siguiente ejemplo:

```
package Ejemplo is
  task type TT_T;  -- especificación del tipo tarea T
  A: TT_T;         -- se crea una tarea
end Ejemplo;

package body Ejemplo is
  task body TT_T is ...;    -- cuerpo del tipo tarea T
  B: TT_T;    -- aquí también es visible el tipo tarea
end Ejemplo;
```

Las tipos tarea (*task type*), al igual que los record, pueden tener discriminante para proporcionar algunos datos al crear la tarea. Como en el caso del tipo de datos **record**, sólo se permiten los tipos discretos:

```
type T_Servicio is (impresora, web, bbdd, archivo);
task type TT_Type (Serv : T_Servicio);

task body TT_Type is        -- cuerpo del tipo tarea
    Servicio : T_Servicio := Serv;
begin
  -- código que describe el comportamiento de la tarea
end TT_Type;
Tarea_Web : TT_Type (web);
```

Por último, hay que indicar que el procedimiento principal que ejecuta el programa también es considerado como una tarea más del programa. No hay que tomarse el tema de las tareas a la ligera y empezar a usar tareas para cualquier cosa, ya que cuantas más tareas se ejecuten mayor es el tiempo que se pierde realizando los *cambios de contexto* por parte de la CPU (restaurando el estado en que se quedó la tarea).

Ejercicio 26: escriba un paquete con un tipo tarea que admita como discriminante el número de veces que se imprime un mensaje en pantalla y el mensaje a imprimir. Utilice el siguiente tipo para imprimir los mensajes: **type** T_Mensaje **is** (Comprando_Ensalada, Comprando_Vino, Comprando_Carne). Implemente un procedimiento que haga uso de dicho paquete, siguiendo el ejemplo anterior.

12.1. TIEMPO REAL Y TAREAS PERIÓDICAS

Los sistemas de tiempo real requieren la representación de sus requisitos temporales. En general, es necesario poder leer el paso del tiempo, retrasar la ejecución de un proceso, o definir límites temporales para la ocurrencia de un proceso. En Ada el paquete predefinido *Ada.Calendar* proporciona funciones de reloj. Así por ejemplo la función `Clock` devuelve un valor del tiempo del tipo `Time` y el tipo predefinido `Duration` permite representar intervalos de tiempo en segundos. El paquete *Ada.Real_Time* proporciona otras funciones adicionales para el control de tareas con restricciones de tiempo-real.

Hay dos formas de que una tarea retrase su ejecución o espere una cantidad de tiempo a que suceda algo (e.g. llegada de un dato, conexión, etc): espera activa y espera pasiva (suspensión). La espera activa es la más fácil de programar:

```
for Tiempo_Perdido in 1..100_000_000 loop
  null;
end loop;
```

Sin embargo, es la menos efectiva, ya que consume tiempo de CPU (más bien, lo malgasta). Además, la tiempo que espera es difícil de calcular, ya que depende de la velocidad de ejecución de la CPU. Y lo que es peor, mientras la tarea está ejecutando el bucle de espera activa ¡ninguna otra tarea puede entrar en ejecución! Así pues, la espera pasiva o suspensión de una tarea es la forma correcta de hacerlo, aunque implica llamadas al sistema operativo para que proceda a su suspensión (cosa que Ada hace también por el desarrollador).

Con el fin de retrasar la ejecución de una tarea Ada dispone de la instrucción **delay**. Dicha instrucción permite suspender la ejecución de una tarea durante un cierto tiempo. La ejecución se puede suspender durante un intervalo de tiempo *relativo* al instante actual mediante la siguiente instrucción:

```
delay expresión;  -- valor real, como 6.8567
```

donde *expresión* es de tipo `Duration` (nótese que una instrucción **delay** con argumento cero o negativo no produce ningún retardo).

La ejecución de una tarea también se puede suspender hasta que se llegue a un instante determinado de tiempo *absoluto*. Para ello Ada dispone de la instrucción **delay until** que suspende la ejecución de la tarea que la invoca hasta que el valor del reloj sea igual al especificado por la expresión, en este caso de tipo `Time`.

Para representar actividades que se realizan periódicamente, por ejemplo un muestreo del entorno con el que el sistema de tiempo real interacciona, es necesario hacer uso de instrucciones que retrasan la ejecución y del paquete *Ada.Calendar* para representar los requisitos temporales. En la siguiente porción de código se representa una tarea periódica haciendo uso de la instrucción **delay until**:

```
with Ada.Calendar;
use Ada.Calendar;

task body Periodica is
  Periodo: constant Duration:= ...;
  Proxima_Vez: Time:= Clock;
begin
  -- iniciación
  loop
    delay until Proxima_Vez;
    -- acción periódica
    Proxima_Vez := Proxima_Vez + Periodo;
  end loop;
end Periodica;
```

Otra representación de una tarea periódica podría realizarse utilizando un retardo relativo mediante el uso de la sentencia **delay**. Sin embargo, de esta forma no se obtiene el resultado esperado (¿por qué?):

```
with Ada.Calendar;
use Ada.Calendar;

task body Periodica2 is
    Periodo: constant Duration:= ...;
    Proxima_Vez: Time := Clock;
begin
  -- inicializacion
```

```
  loop
    Proxima_Vez := Proxima_Vez + Periodo;
    -- actividad periódica
    delay Proxima_Vez – Clock;
  end loop;
end Periodica2;
```

Ejercicio 27: escriba un programa con tres tareas periódicas (llamadas Tarea_1, Tarea_2 y Tarea_3) con diferentes periodos que muestren en pantalla su nombre y una cuenta atrás (la salida debería salir entremezclada). Compruebe la implementación tanto con un retardo absoluto como relativo.

12.2. TERMINACIÓN DE TAREAS

Una tarea en Ada puede terminar de varias formas. Una de ellas es porque se completa la ejecución de su cuerpo o porque se ha elevado una excepción que no maneja. También las tareas pueden terminar porque otra tarea (o ella misma) la aborta. Cuando se aborta una tarea se abortan también todas las tareas que dependen de ella.

```
abort A;
```

Esta forma de terminación sólo es válida para las tareas que no son anónimas. Las tareas anónimas pueden ser abortadas mediante procedimientos definidos en el paquete de biblioteca *Ada.Task_Identification*. Salvo en las más contadas excepciones, las tareas nunca deben acabarse de esta forma, ya que pueden dejar variables en un estado inconsistente y afectar al resto del programa.

La forma en que las tareas van a acabar su ejecución tiene que ser prevista por el diseñador y es una parte importante del código de una tarea. La forma normal es que se pueda controlar el número de veces que se repite la ejecución del cuerpo de la tarea, de forma que se pueda parar fácilmente. Otra posibilidad es que la tarea tenga un bloque **select** con una opción **terminate** (ver siguiente capítulo).

```
with Ada.Calendar;
use Ada.Calendar;

Continuar : Boolean := True;

procedure Terminar_Tarea is
begin
  Continuar := False;
end Terminar_Tarea;

task body Periodica_Con_Fin is
  Periodo: constant Duration:= ...;
  Proxima_Vez: Time := Clock;
```

```
begin
  -- inicializacion
  while (Continuar) loop
    delay until Proxima_Vez;
    -- acción periódica
    Proxima_Vez := Proxima_Vez + Periodo;
  end loop;
end Periodica_Con_Fin;
```

El atributo booleano *Terminated* se usa para comprobar si la tarea ha terminado y es verdadero cuando la tarea A ha terminado (A'*Terminated*).

13. COMUNICACIÓN ENTRE TAREAS

Aunque las tareas sean entidades independientes que se ejecutan de forma "paralela", sin tener conocimiento (ni necesidad de tenerlo) de otras posibles tareas que existen en el sistema, es habitual que, en determinados momentos, dos o más tareas tengan que sincronizarse en algún punto. A partir de ese punto, cada una sigue evolucionando a su ritmo. Hay dos mecanismos de comunicación entre tareas:

1. Comunicación **síncrona**: las tareas esperan hasta que todas hayan llegado al "punto de reunión". Es similar a una llamada telefónica. Se puede utilizar sólo para sincronizarse (*cita*) o también para realizar cálculos e intercambiar variables (*cita extendida*).
2. Comunicación **asíncrona**: ninguna de las tareas espera a la otra, sino que lanza su mensaje y continúa su ejecución. Es similar a un buzón de correos. Se usa sobre todo para el intercambio de datos de *forma segura* y *sin esperas innecesarias*.

13.1. COMUNICACIÓN SÍNCRONA: CITA EXTENDIDA

Puesto que la comunicación síncrona implica a una tarea invocada (servidor) y una o varias tareas invocantes (clientes), cada una de ellas tiene que definir la parte que le corresponde:

1. La tarea servidor define las entradas de sincronización que se pueden invocar. Las entradas de sincronización (denominadas **entry**) se definen en la especificación de la tarea, independientemente de si ésta es un *task type* o una tarea anónima. La definición de un **entry** es similar a la de un **procedure**. La declaración de parámetros de tipo '**out**' permite a la tarea servidor devolver valores a la tarea cliente. Un **entry** sin parámetros se usa sólo para *sincronización*.

```
task type TT_Pantalla is
  entry Escribir(Char: Character; X,Y: Coordenadas);
end Pantalla;
Display: TT_Pantalla; --declaración de tarea servidor
```

2. La tarea cliente invoca el **entry** (punto de reunión) de otra tarea servidor que está definida en su ámbito, es decir, que sea visible, utilizando la llamada *<nombre_servidor>.<nombre_entry>(parámetros)* en el cuerpo (*task body*) de la tarea cliente:

```
Display.Escribir('A',50,24);
```

3. En la implementación de la tarea servidor (*task body*) se colocan, por último, los puntos en que el servidor se quiere sincronizar con la tarea cliente. Para aceptar una llamada, el servidor coloca en su cuerpo una sentencia **accept** seguida del nombre y los parámetros del **entry** que quiere aceptar. Opcionalmente, un

accept puede incluir un bloque **do..end** con el conjunto de instrucciones a ejecutar mientras dure la cita. Sin un bloque do..end, la cita es sólo de sincronización. El bloque **do..end** permite el uso de **return** para acabar la cita en cualquier punto y la definición de un bloque para capturar excepciones. Una excepción no capturada o relanzada se propaga a ambas tareas.

```
accept Escribir(Char: Character; X,Y: Coordenadas)do
  -- Secuencia de instrucciones para escribir un
  -- carácter en la posición (X,Y)
exception
  -- bloque de tratamiento de excepciones
end Escribir;
```

El cuerpo del **accept** especifica las acciones que se ejecutan cuando se acepta la llamada. La tarea servidora espera en el **accept** a que alguna tarea cliente realice una llamada. Del mismo modo, la tarea que realiza la llamada quedará a la espera hasta que la tarea receptora acepte la cita. Esta forma de comunicación y sincronización entre tareas se basa en un mecanismo de *cita extendida*: el emisor identifica explícitamente al receptor y éste acepta mensajes de cualquier emisor. Mientras están citadas, la tarea invocante (cliente) queda a la espera (suspendida), mientras que la tarea receptora (servidor) ejecuta el código del **accept**. Por tanto, no conviene hacer los bloques **accept** demasiado largos, puesto que la tarea cliente se mantiene suspendida. Una vez que acaba, ambas tareas continúan su marcha normal e independiente una de otra.

Muchos clientes pueden invocar un **entry** de una tarea servidor. En este caso, sólo uno es atendido a la vez, colocándose el resto en una cola a la espera de que el servidor las pueda atender. Para que se lleve a cabo una cita, la tarea receptora debe aceptar la llamada al punto de entrada correspondiente. Para ello debe ponerse al menos una instrucción **accept** por cada entrada y en cualquier lugar del cuerpo de la tarea receptora.

Ejemplo 19: ejecute el siguiente código y compruebe la salida

```
with Ada.Text_IO, Ada.Integer_Text_Io;
use Ada.Text_IO, Ada.Integer_Text_Io;
procedure Ejemplo_Cita is
  task Servidor is --Especificacion
    entry Sincronizacion;
    entry Cita_Extendida (Par : in out Integer);
  end Servidor;
  task body Servidor is --Implementacion
    Nombre : constant String := "SERVIDOR =>";
    Parametro : Integer := 100;
  begin
    Put_Line (Nombre & " Espera 3 seg antes de aceptar");
    delay 5.0;
    accept Sincronizacion;
    New_line(2);
    Put_Line (Nombre & " Espero a que me llamen");
```

```
    accept Sincronizacion do
      Put("--- Ahora no es solo sincronizacion, ");
      Put_Line ("sino cita (aunque sin parametros) ---");
    end Sincronizacion;
    delay 3.0; New_Line(2);
    Put (Nombre & " 'Parametro' vale "); Put (Parametro);
    Put_Line(" antes de la cita"); delay 3.0; New_Line(2);
    accept Cita_Extendida (Par : in out Integer) do
      Put_Line("Estoy intercambiando los valores...");
      declare
        aux : Integer;
      begin
        aux := Par; Par := Parametro;
        Parametro := aux; delay 5.0;
      end;
    end Cita_Extendida;
    delay 3.0; New_Line(2);
    Put (Nombre & " 'Parametro' vale "); Put (Parametro);
    Put_Line(" despues de la cita");
  end Servidor;
  Parametro : Integer := 10;
  Nombre : constant String := "CLIENTE =>";
begin
  Put_Line (Nombre & " Start! Invoco al servidor ya");
  Servidor.Sincronizacion;
  Put_Line (Nombre & " Espero 3 seg antes de llamar");
  delay 5.0;
  Servidor.Sincronizacion;
  delay 3.0;
  Put (Nombre & " 'Parametro vale '"); Put (Parametro);
  Put_Line(" antes de la cita");
  delay 3.0;
  Servidor.Cita_Extendida (Parametro);
  delay 3.0;
  Put (Nombre & " 'Parametro vale '"); Put (Parametro);
  Put_Line(" despues de la cita");
end Ejemplo_Cita;
```

13.1.1. Espera selectiva en el lado del servidor

A menudo no es posible prever en qué orden se van a invocar las entradas de una tarea, sobre todo si se trata de una tarea que acepta llamadas a una o más entradas y ejecuta un servicio para cada una de ellas. En este caso es necesario que la tarea receptora pueda esperar simultáneamente llamadas en varias entradas. Ada ofrece una estructura de control que permite la espera selectiva en varias alternativas mediante la palabra reservada **select**. El bloque **select** se comporta como una estructura secuencial (elige una opción y termina).

```
select
  accept Entrada_1 do
    ...
  end Entrada_1;
  -- aquí puede haber más instrucciones
or
  accept Entrada_2 do
    ...
  end Entrada_2;
  -- puede haber aquí más instrucciones
or
  ...
end Escribir;
```

También es posible que alguna de las alternativas de selección se acepte sólo en determinadas condiciones. Para ello Ada define las *guardas* para los **accept**, en cuyo caso la cita será aceptada si la condición es verdadera. Hay que tener cuidado de no colocar un conjunto de guardas de forma que todas sean falsas y no se pueda elegir ninguna (el servidor quedaría bloqueado). Las condiciones de las guardas sólo se re-evalúan cada vez que se entra en el **select**.

```
select
   when condición => accept Entrada_3 do
      -- sentencias
   end Entrada_3;
   ...
end select;
```

Una instrucción **select** puede tener una parte final **else** que se ejecuta si al llegar al **select** no se puede aceptar inmediatamente ninguna otra alternativa (no hay tareas esperando en ningún **accept**). Esta es la opción a elegir para evitar el bloqueo de la tarea cuando existen muchas guardas.

```
select
  accept Entrada_1;
[or
  accept Entrada_n;]
else
  secuencia_de_instrucciones
end select;
```

Una de las alternativas de un **select** puede incluir una alternativa **terminate**, en cuyo caso no es posible usar el **else**.

```
or
   terminate;
```

La alternativa **terminate** es siempre la última de todas y sólo se ejecuta si todas las tareas de las que depende esta tarea han acabado o están listas para acabar (en su propio **terminate**) y si no hay ningún cliente esperando en ninguno de los otros **accept**. De esta forma todas las tareas acaban a la vez de forma segura.

Por último, se puede especificar un tiempo máximo que el servidor espera a recibir una llamada antes de ejecutar un trozo de código alternativo utilizando "**or delay".**

```
select
  accept Entrada do
    -- secuencia de instrucciones
  end Entrada;
or
  accept Entrada_2;
  delay 1.0; -- no confundir con la siguiente linea
or
  delay tiempo_en_segundos;
  -- acción alternativa
 end select;
```

En el ejemplo, si no se puede aceptar la llamada la tarea que la realiza queda suspendida *tiempo_en_segundos* segundos y ejecuta una acción alternativa en caso de no producirse ninguna cita durante ese tiempo.

Las tres alternativas especiales de un **select** (**else**, **terminate** y **delay**) son mutuamente excluyentes entre sí, por lo que sólo puede aparecer una de ellas.

Ejercicio 28: escriba un programa para modelar el comportamiento de un servidor. El servidor acepta llamadas a dos entradas: imprimir trabajo (tarda 10 segundos) y consultar correo (tarda 4 segundos). Haga uso de **select** para evitar que el servidor esté parado cuando hay otros clientes esperando. Defina un tipo tarea para el cliente (con ID) y lance varios clientes.

13.1.2. Espera selectiva en el lado del cliente

La capacidad de elección de la estructura **select** puede usarse también en la tarea invocante para mejorar su comportamiento, aunque las posibilidades están restringidas al **else** o a la llamada temporizada.

Se puede limitar el tiempo que tarda en aceptarse una llamada mediante una llamada temporizada:

```
select
  -- llamada a la tarea receptora
or
  delay tiempo_en_segundos;
  -- acción alternativa
end select;
```

Se puede realizar otra acción en caso de que el servidor no esté listo para aceptar la llamada:

```
select
  -- llamada a la tarea receptora
else
  -- acción alternativa
end select;
```

A diferencia de lo que se decía para el servidor, en el caso del cliente sólo puede aparecer o bien un **or delay** o bien un **else**. No se pueden mezclar ni pueden aparecer varios **or**.

Por último, falta el **select** *asíncrono*. En esta opción, el cliente espera un tiempo máximo a que el servidor acabe de ejecutar la cita o bien (el cliente) termina la cita y ejecuta un código alternativo.

```
select
  delay tiempo_en_segundos;
  -- codigo alternativo por si se demora el principal
then abort
  -- codigo de la accion principal
end select;
```

Ejercicio 29: modifique el programa del ejercicio anterior para que los clientes esperen dos segundos a que el servidor les atienda. Si no, muestran un mensaje de desacuerdo y se van.

Ejemplo 20: estudie, compile y ejecute los ejemplos que se encuentran en el directorio "barbero". Lea primero el fichero de texto antes de ejecutarlos.

13.1.3. Familia de entrys

Los entry relacionados con el mismo número y tipo de parámetros pueden definirse en lo que se llama una familia de entrys. El concepto es muy similar al de un *array*.

```
task Familia_Entrys is
   entry Entrada (1..6) (X: in out Float);
   entry Normal;
end Familia_Entrys;
..............
accept Entrada (2) (X: in out Float) do
   -- codigo asociado a la entrada
end Entrada;
..............
Familia_Entrys.Entrada(2)(X=>5.678);  --llamada
```

13.1.4. Atributos de las tareas

Ada define los siguientes atributos para aplicar a las tareas:

- El atributo booleano *Callable* se usa para comprobar si se puede llamar todavía a algún **entry** de la tarea (A'*Callable*).
- El atributo *Count* devuelve el número de tareas que están esperando en la cola de un **entry** determinado (Entry_Cualquiera'*Count*). Este atributo solo puede ser usado dentro del cuerpo de la tarea que contiene el **entry**.

13.2. COMUNICACIÓN ASÍNCRONA: TIPOS PROTEGIDOS

Una secuencia de instrucciones que se debe ejecutar de forma indivisible se denomina *sección crítica*. La forma de sincronización que se usa para proteger una sección crítica se llama *exclusión mutua*. Ada ofrece un mecanismo para llevar a cabo este tipo de operaciones: los *tipos protegidos*. Los tipos protegidos se engloban dentro de los tipos limitados (no tienen predefinida ninguna operación, ni siquiera :=, = ni /=).

Un tipo protegido en Ada es un tipo compuesto cuyas operaciones se ejecutan en exclusión mutua. Se pueden declarar tipos protegidos únicos, como ocurría en el caso de las tareas. En ambos casos hay una *especificación* (interfaz visible desde otras unidades de programa) y un *cuerpo* (invisible para los usuarios). Las operaciones se declaran en la parte pública y los detalles de implementación del tipo van en la parte privada, siendo sólo visibles en el cuerpo. El siguiente código muestra un ejemplo de tipo protegido que garantiza la exclusión mutua en el acceso a un número entero compartido entre dos tareas:

```
protected type PT_Entero_Compartido (Val:Integer) is
  function Valor return Integer;
  procedure Cambiar(Nuevo_Valor: Integer);
private
  Dato: Integer := Val;
end PT_Entero_Compartido;
```

Un **procedure** da acceso exclusivo en lectura y escritura a los datos privados mientras que una **function** da acceso concurrente en lectura. Como en el caso de la cita extendida, la tarea que invoca un subprograma de un tipo protegido puede utilizar también **select**. El código de las operaciones va en el cuerpo del tipo protegido:

```
protected body PT_Entero_Compartido is
   function Valor return Integer is
   begin
     return Dato;
   end Valor;

   procedure Cambiar(Nuevo_Valor: Integer)is
   begin
```

```
      Dato:= Nuevo_Valor;
   end Cambiar;
end PT_Entero_Compartido;
```

Es posible declarar variables del tipo protegido en cualquier zona declarativa donde sea visible; al fin y al cabo, se comportan como exactamente igual que un tipo cualquiera, como Integer. En la siguiente porción de código se declaran dos variables del tipo *Entero_Compatido* y se inicializan respectivamente a 0 y 1. La invocación de operaciones siempre debe ir precedida por el nombre de la variable y utiliza la notación *variable.operación* (similar a la de los **record**).

```
declare
  X : PT_Entero_Compartido(0);
  Y : PT_Entero_Compartido(1);
begin
  X.Cambiar(Y.Valor + 1); -- ahora X.Valor es 2
end;
```

13.2.1. Entradas protegidas y sincronización condicional

Para realizar sincronización condicional pueden definirse varios **entry** en un tipo protegido. Una entrada es una operación protegida con una interfaz semejante a la de un procedimiento, pero al que se le ha añadido una barrera *booleana*. Los entrys proporcionan acceso excluyente de lectura y escritura de las variables del tipo protegido.

```
entry E (...) when Bool is ...
```

De esta forma solo cuando la barrera (condición Bool en el ejemplo) es verdadera, la tarea que invoca la entrada puede continuar. En caso contrario, la tarea queda a la espera. Realmente, un '**procedure**' es un '**entry when** True'. El siguiente ejemplo muestra el tipo protegido Buffer con una entrada para almacenar números enteros en el mismo:

```
protected type PT_Buffer is
  entry Push(X: in Integer);
private
 ...
end PT_Buffer;
```

En el cuerpo del tipo protegido se asocia a la entrada una barrera que indicará que si el buffer está lleno no puede almacenarse ningún elemento:

```
protected body PT_Buffer is
  entry Push(X: in Integer) when no_lleno is
  begin
    ...
  end Poner;
end PT_Buffer;
```

Las llamadas a las entradas deben ir siempre cualificadas con el nombre de la variable:

```
Datos: PT_Buffer;
Datos.Poner(3);
```

Dado que los tipos protegidos son unidades de programa, y no de compilación, se tienen que declarar dentro de un bloque.

13.3. ACCESO ATÓMICO A VARIABLES

Ada también permite definir variables cuya lectura y escritura se realiza de forma atómica, sin tener que definir un tipo protegido. Para ello, basta con añadir tras la definición de la variable la siguiente directiva para el compilador:

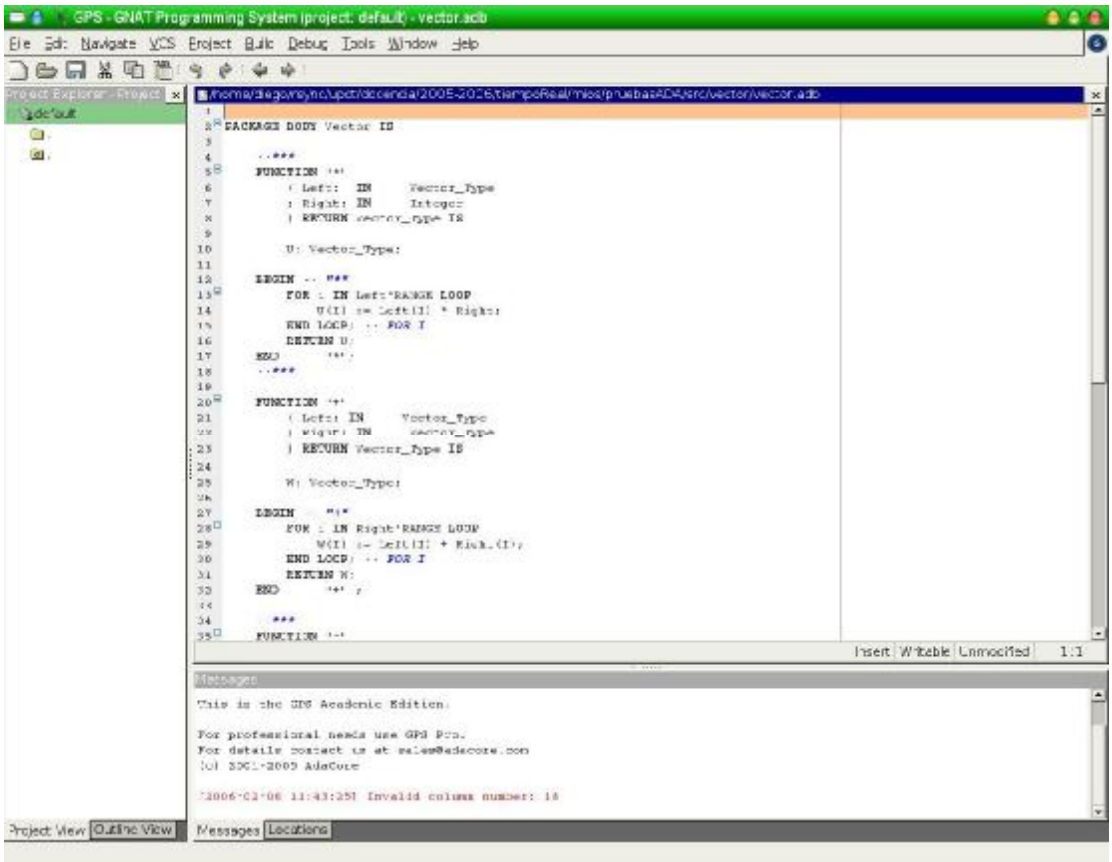
```
pragma Atomic (nombre_vble);
  -- 'pragma' indica una directiva para el compilador
```

GNAT PROGRAMMING SYSTEM (GPS)

GPS es un entorno de desarrollo integrado (IDE) para programar no sólo en Ada sino también en otros lenguajes, como C/ C++. Este entorno de desarrollo tiene, entre otras, las siguientes características:

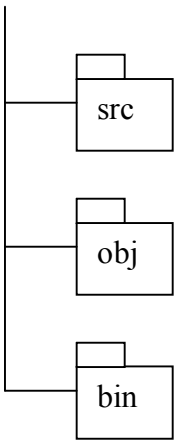
- Edición inteligente y coloración de sintaxis.
- Sistema de ayuda basado en HTML.
- Soporte para el ciclo de desarrollo: compilación/ construcción/ ejecución.
- Navegación inteligente por el código fuente.
- Desarrollo basado en proyecto.
- Generación de grafos de dependencia entre ficheros, árboles de llamada, etc.
- Depurador integrado.
- Integración con el software de control de versiones (CVS, Subversión, etc …)
- Análisis de diferencias entre ficheros.
- Generación automática del código.
- Reformateo de código fuente.

La siguiente figura muestra una captura de pantalla del entorno en funcionamiento:



Toda la información relativa al entorno de desarrollo y al compilador de Ada se encuentra dentro de la carpeta "doc" que se crea al instalar el GPS. Por favor, consulte dicha carpeta para ampliar la información sobre el sistema, ya que el presente documento no es más que una escueta guía para comenzar.

Puesto que GPS utiliza una definición de proyecto para manejar el código fuente, lo normal es comenzar siempre con un proyecto nuevo para realizar las prácticas, aunque cuando no sean muchos ficheros se puede reutilizar el mismo proyecto. Es muy importante ponerse de acuerdo en la estructura de los directorios a utilizar, y se recomienda hacerlo siempre de la siguiente forma (extrapolable a cualquier lenguaje de programación y a cualquier entorno):



Directorio para organizar el código fuente.

Directorio para almacenar el código intermedio

Directorio para almacenar el código ejecutable.

Se recomienda crear un directorio nuevo para cada proyecto con sus subdirectorios src, obj y bin, si no aparecerán todos los ficheros en todos los proyectos, ya no existe una forma explícita de añadirlos al proyecto que queremos.

CÓMO CREAR UN PROYECTO

Para comenzar un nuevo proyecto hay que escoger el menú "Proyect→New" y aparece la siguiente ventana, en la que se nos piden las características del proyecto:

1. Nombre y ruta
2. Lenguaje de programación
3. Software de control de versiones
4. Directorio/ s para el código fuente (con subdirectorios)
5. Directorio para el código objeto y el código ejecutable
6. Fichero/ s principal/ es
7. Esquema de nombrado
8. Opciones para el compilador

De todas las opciones, sólo hay que especificar los puntos 1,4,5 y 6, pero puede que alguna vez se necesite cambiar la 8.

A partir de aquí, cualquier fichero que se cree en la carpeta definida en (4) se añadirá directamente al árbol del proyecto. Se recuerda que se utiliza la extensión "**adb**" para la implementación y "**ads**" para la especificación. Hasta que no se llegue al tema de paquetes todos los ficheros tienen extensión **adb**.

CÓMO COMPILAR Y EJECUTAR UN PROGRAMA

Para crear un nuevo fichero se utiliza el menú "File→New". Es muy importante añadir a la definición del proyecto cuál o cuáles van a ser los ficheros principales (los que proporcionan el punto de entrada). Para ello se selecciona la opción "Proyect→Edit Proyect Properties", que abre la ventana de definición de proyecto. En ella elegimos la pestaña "Main Files" y añadimos el fichero que se va a utilizar como principal. Si no se realiza esta operación el código sólo se compila, pero ni se enlaza ni se construye el ejecutable.

Una vez se han marcado los ficheros principales ya se puede proceder a acabar de escribir el código fuente, a compilarlo y ejecutarlo. Las siguientes opciones se encuentran todas dentro del menú "Build", aunque algunas de ellas dependen de los ficheros que se hayan añadido como principales:

- Se puede construir (compilar+enlazar) todo el código fuente o sólo la parte responsable de crear uno de los ejecutables (Make→All, Make→), tecla **F4**.
- Se puede compilar sólo el fichero que se esté editando actualmente. No se genera ejecutable ni compila ninguna dependencia más (Compile File).
- Se puede ejecutar el código ya construido (Run→).

GPS es un entorno de programación muy completo que tiene muchas opciones para facilitar la vida al programador. Por favor, pierda parte de su tiempo experimentado y leyendo el manual de referencia del entorno. Tampoco pierda la oportunidad de consultar el menú de ayuda ("Help"), ya que en él encontrará toda la información necesaria tanto sobre el compilador de Ada como sobre el propio entorno de programación. En concreto, en el menú Help → GNAT Runtime → Ada se encuentra la especificación de todos los paquetes que forman el lenguaje.

ALGUNOS PAQUETES DE UTILIDAD

En este anexo sólo se van a comentar los paquetes de utilidad que englobados dentro del paquete padre "Ada". De todos los paquetes que proporciona por defecto GNAT, los que se van a utilizar con mayor frecuencia son los siguientes:

- Todos los acabados en Text_IO, ya que son los que permiten realizar la entrada salida de datos por teclado y pantalla:
  - *Ada.Integer_Text_Io* para tipos *Integer*, *Ada.Float_Text_Io* para *Float*, *Ada.Long_Float_Text_Io* para *Long_Float*, etc.
  - Para los tipos "nuevos" (creados con el operador new) y el resto de tipos (enumerados, modulares, reales, etc) los paquetes genéricos correspondientes se encuentran dentro del paquete *Ada.Text_Io*, son paquetes anidados.
- Los paquetes para realizar operaciones matemáticas "Ada.Numerics.X". Dentro de esta familia de paquetes los más importantes son:
  - *Ada.Numerics.Generic_Elementary_Functions* : paquete genérico con las funciones elementales (trigonométricas, raíz cuadrada, logaritmos, etc).
  - *Ada.Numerics.Discrete_Random* y *Ada.Numerics.Float_Random* : paquetes para trabajar con números aleatorios. El primero de ellos es genérico.
- *Ada.Decimal* : proporciona un método para obtener de una sola operación el cociente y el resto de una división.
- *Ada.Calendar* y *Ada.Real_Time* : paquetes para trabajar con unidades temporales (de uso en la implementación de tareas).
- *Ada.Command_Line* : paquete para poder pasarle argumentos a un ejecutable.

El siguiente ejemplo muestra cómo obtener números aleatorios en Ada. Compílelo y ejecútelo.

```
with Ada.Text_IO, Ada.Float_Text_IO,
  Ada.Numerics.Discrete_Random, Ada.Numerics.Float_Random;
use Ada.Text_IO;

procedure Main is
  type Dias_Semana is (Lunes,Martes,Miercoles,Jueves,
      Viernes,Sabado,Domingo);
  package Dias_Io is new Enumeration_IO (Dias_Semana);
  package Dia_Random_Pack is new
         Ada.Numerics.Discrete_Random (Dias_Semana);
  Discrete_Generator : Dia_Random_Pack.Generator;
  Dias_temp          : Dias_Semana;
  package Float_Random_Pack renames
                  Ada.Numerics.Float_Random;
  package Float_Io renames Ada.Float_Text_IO;
  Float_Generator : Float_Random_Pack.Generator;
```

```
   float_Temp      : Float;
begin
   for i in  1 .. 5 loop
      Dias_temp  := Dia_Random_Pack.Random
                    (Discrete_Generator);
      float_Temp := Float_Random_Pack.Random
                    (Float_Generator);
      Put ("El dia aleatorio es: ");
      Dias_Io.Put (Dias_temp); New_Line;
      Put ("El float aleatorio es: ");
      Float_Io.Put (float_Temp); New_Line(3);
   end loop;
end Main;
```

Ada REFERENCE CARD

bold	Ada keyword	*italic*	Ada 95
[ ]	Optional term	{ }	Repeatable
|	Alternative	...	Identical term

ATTRIBUTES

S - subtype E - entry declaration or exception
T - task X - object
P - program A - discriminated type or array
R - record D - library-level declaration

P'Access Access to subprogram

X'Access Access to object

X'Address Address of the first of the storage elements allocated to object, program unit, or label

S'*Adjacent* Adjacent machine number of argument towards the second floating point argument.

S'Aft The number of decimal digits needed after the decimal point to accommodate the delta

X'*Alignment* Alignment of object

S'Base Denotes the base unconstrained subtype

S'*Bit_Order* Record subtype bit ordering (type System.Bit_Order)

P'*Body_Version* Version of the compilation unit that contains the body

T'Callable True when the task denoted by T is callable

E'*Caller* Value of the type Task_ID that identifies the task whose call is now being serviced

S'*Ceiling* Smallest (most negative) integral value greater than or equal to argument

S'*Class* Subtype of the class-wide type

X'*Component_Size* Size in bits of components of the array subtype or object

S'*Compose* Combine fraction and integer arguments into a floating point subtype

A'Constrained True if discriminated type denotes a constant, a value, or a constrained variable

S'*Copy_Sign* Result whose magnitude is that of float Value and whose sign is that of Sign

E'Count Number of calls presently queued on the entry

S'*Definite* True if the actual subtype of a a formal indefinite subtype is definite

S'Delta The delta (universal_real) of the fixed point subtype

S'*Denorm* True if every value expressible in canonical form with an an exponent of T'Machine_Emin

S'Digits Number of digits of the decimal fixed point subtype

S'Digits Number of decimal mantissa digits for floating point subtype

S'*Exponent* Normalized exponent of the floating point argument

S'*External_Tag* An external string representation of the tagged type

A'First(N) Lower bound of N-th index of [constrained] array type

A'First Lower bound of first index of [constrained] array type

S'First Lower bound of the range of scalar subtype

R.C'First_Bit Bit offset, from the start of the first of the storage elements occupied by C, of the first bit occupied by C

S'*Floor* Largest integral value less than or equal to the argument

S'Fore Minimum number of characters needed before the decimal point

S'*Fraction* Decompose floating point argument into fractional part

E'*Identity* Unique identity of the exception

T'*Identity* Value of type Task_ID identifying the task

S'Image Image of the value of argument as a String

S'*Input* Reads and returns one value from the Stream argument

A'Last(N) Upper bound of N-th index range of [constrained] array type

A'Last Upper bound of first index range of [constrained] array type

S'Last Upper bound of the range of scalar subtype

R.C'Last_Bit Bit offset, from the start of the first of the storage elements occupied by C, of the last bit occupied by C

S'*Leading_Part* The leading part of floating point value with number of radix digits given by second argument

A'Length(N) Number of values of the N-th index range of [constrained] array type

A'Length Number of values of the first index range of [constrained] array type

S'*Machine* Machine representation of floating point argument

S'Machine_Emax Largest (most positive) value of floating point exponent

S'Machine_Emin Smallest (most negative) value of floating point exponent

S'Machine_Mantissa Number of digits in machine representation of mantissa

S'Machine_Overflows True if numeric overflow detected for fixed or floating point

S'Machine_Radix Radix of machine representation of the fixed or floating point

S'Machine_Rounds True if rounding is performed on inexact results of the fixed or floating point

S'*Max* The greater of the values of the two scalar arguments

S'*Max_Size_In_Storage_Elements* Maximum value for Size_In_Storage_Elements that will be requested via Allocate

S'*Min* The lesser of the values of the two scalar arguments

S'*Model* Model number of floating point type

S'*Model_Emin* Model number version of S''Machine_Emin

S'*Model_Epsilon* Absolute difference between the model number 1.0 and the next model number above for subtype.

S'*Model_Mantissa* Model number version of S''Machine_Mantissa

S'*Model_Small* Smallest positive model number of subtype

S'*Modulus* The modulus (universal_integer) of the modular subtype

S'*Output* Writes the value of Item to Stream, including any bounds or discriminants

D'*Partition_ID* Identifies the partition in which D was elaborated

S'Pos Position of the value of the discrete subtype argument

R.C'Position Same as R.C'Address - R'Address for component C

S'Pred Predecessor of the argument

A'Range(N) Equivalent to the range A'First(N) .. A'Last(N)

A'Range Equivalent to the range A'First .. A'Last

S'Range Equivalent to the range S'First .. S'Last

S'*Read* Reads the value of Item from the Stream argument

S'*Remainder* Remainder after dividing the first floating point argument by its second.

S'*Round* Fixed-point value obtained by rounding X (away from 0, if X is midway between two values)

S'*Rounding* Floating-point integral value nearest to X, rounding away from zero if X lies exactly halfway between two integers

S'Safe_First The lower bound of the safe range

S'Safe_Last The upper bound of the safe range

S'*Scale* Position of the fixed-point relative to the rightmost significant digits of values of subtype

S'*Scaling* Scaling by a power of the hardware radix.

S'*Signed_Zeros* True if positive and negative signed zeros are representable

S'Size Size in bits of objects instantiated from subtype

X'Size Size in bits of the representation of the object

S'Small Small of the fixed-point type

S'*Storage_Pool* Storage pool of the access subtype

S'*Storage_Size* Number of storage elements reserved for the storage pool

T'Storage_Size	Number of storage elements reserved for the task
S'Succ	Successor of the argument
S[X]'*Tag*	The tag (type Tag) of the [class-wide] tagged type
T'Terminated	True if the task denoted by T is terminated
S'*Truncation*	The value Ceiling(X) when X is negative, else Floor(X)
S'*Unbiased_Rounding*	Integral value nearest to X, rounding toward the even integer if X lies exactly halfway between two integers.
X'*Unchecked_Access*	Same as X'Access but lacks accessibility rules/checks
S'Val	Value of the discrete subtype whose position number equals the value of argument
X'*Valid*	True if and only if the scalar object denoted by X is normal and has a valid representation
S'Value	Returns a value of the subtype given an image of the value as a String argument
P'*Version*	The version of the compilation unit that contains the declaration
S'*Wide_Image*	Image of the value of argument as a Wide_String
S'*Wide_Value*	Returns a value given an image of the value as a Wide_String argument
S'*Wide_Width*	Maximum length of Wide_String returned by S'Image
S'Width	Maximum length of String returned by S'Image
S'*Write*	Writes the value of Item to Stream argument

PRAGMAS

pragma *All_Calls_Remote*[(library_unit_name)];
pragma *Asynchronous*(local_name);
pragma *Atomic*(local_name);
pragma *Atomic_Components*(array_local_name);
pragma *Attach_Handler*(handler_name, expression);
pragma Controlled(first_subtype_local_name);
pragma *Convention*([Convention =>] convention_identifier,
[Entity =>] local_name);
pragma *Discard_Names*[([On =>] local_name)];
pragma Elaborate(library_unit_name{, ...});
pragma *Elaborate_All*(library_unit_name{, ...});
pragma *Elaborate_Body*[(library_unit_name)];
pragma *Export*([Convention =>] convention_identifier,
[Entity =>] local_name [,
[External_Name =>] string_expression] [,
[Link_Name =>] string_expression]);
pragma *Import*([Convention =>] convention_identifier,
[Entity =>] local_name [,
[External_Name =>] string_expression] [,
[Link_Name =>] string_expression]);
pragma Inline(name {, ...});
pragma *Inspection_Point*[(object_name {, ...})];
pragma *Interrupt_Handler*(handler_name);
pragma *Interrupt_Priority*[(expression)];
pragma *Linker_Options*(string_expression);
pragma List(identifier);
pragma *Locking_Policy*(policy_identifier);
pragma *Normalize_Scalars*;
pragma Optimize(identifier);
pragma Pack(first_subtype_local_name);
pragma Page;
pragma *Preelaborate*[(library_unit_name)];
pragma Priority(expression);
pragma *Pure*[(library_unit_name)];
pragma *Queuing_Policy*(policy_identifier);
pragma *Remote_Call_Interface*[(library_unit_name)];
pragma *Remote_Types*[(library_unit_name)];
pragma *Restrictions*(restriction{, ...});
pragma *Reviewable*;
pragma *Shared_Passive*[(library_unit_name)];
pragma *Storage_Size*(expression);
pragma Suppress(identifier [, [On =>] name]);
pragma *Task_Dispatching_Policy*(policy_identifier);
pragma *Volatile*(local_name);
pragma *Volatile_Components*(array_local_name);

STANDARD LIBRARY

package Standard

Boolean	True or False
Integer	Implementation defined
Natural	Integers >= 0
Positive	Integers > 0
Float	Implementation defined
Character	*8*-bit ASCII
Wide_Character	16-bit ISO 10646
String	Array of Characters
Duration	Time
Constraint_Error	Predefined Exception
Program_Error	Predefined Exception
Storage_Error	Predefined Exception
Tasking_Error	Predefined Exception

package *Ada*

- *Asynchronous_Task_Control*
- *Calendar*
- *Characters*
  - *Handling*
  - *Latin_1*
- *Command_Line*
- *Decimal*
- *Direct_IO*
- *Dynamic_Priorities*
- *Exceptions*
- *Finalization*
- *Float_Text_IO*
- *Integer_Text_IO*
- *Interrupts*
  - *Names*
  - *IO_Exceptions*
- *Numerics*
  - *Complex_Elementary_Functions*
  - *Complex_Types*
  - *Discrete_Random*
  - *Elementary_Functions*
  - *Float_Random*
  - *Generic_Complex_Elementary_Functions*
  - *Generic_Complex_Types*
  - *Generic_Elementary_Functions*
- *Real_Time*
- *Sequential_IO*
- *Storage_IO*
- *Streams*
  - *Stream_IO*
- *Strings*
  - *Bounded*
  - *Fixed*
  - *Maps*
    - *Constants*
  - *Unbounded*
  - *Wide_Bounded*
  - *Wide_Fixed*
  - *Wide_Maps*
    - *Wide_Constants*
  - *Wide_Unbounded*
- *Synchronous_Task_Control*
- *Tags*
- *Task_Attributes*
- *Task_Identification*
- *Text_IO*
  - *Complex_IO*
  - *Text_Streams, etc*
- *Unchecked_Conversion*
- *Unchecked_Deallocation*
- *Wide_Text_IO*
  - *Complex_IO*
  - *Text_Streams, etc*

package *Interfaces*

- *C*
  - *Pointers*
  - *Strings*
- *COBOL*
- *Fortran*

package *System*

- *Address_To_Access_Conversions*
- *Machine_Code*
- *RPC*
- *Storage_Elements*
- *Storage_Pools*

Ada SYNTAX CARD

bold	Ada keyword	*italic*	Ada 95
[ ]	Optional term	{ }	Repeatable
|	Alternative	\ \	Choose one
...	Identical term	::=	Expansion term

LIBRARY

```
← COMPILATION_UNIT ::=
  {with library_unit_name {,...}; | use_clause} library_item
| {with library_unit_name {,...}; | use_clause} separate (parent_name)
     \subprogram_body|package_body|task_body|protected_body\

← USE_CLAUSE ::=
  use pack_name {,...};              | use type subtype_name {,...};

← LIBRARY_ITEM ::=
  [private] subprogram_spec;     | [private] package_spec;
| [private] generic {generic_formals|use_clause} subprogram_spec;
| [private] generic {generic_formals|use_clause} package_spec;
| [private] package [parent_name.]id is new gen_pack_name
            [generic_actuals];
| [private] procedure [parent_name.]id is new gen_proc_name
            [generic_actuals];
| [private] function [parent_name.]id|op is new gen_func_name
            [generic_actuals];
| subprogram_body;                | package_body;
| [private] package [parent_name.]id renames pack_name;
| [private] generic package [parent_name.]id renames
            gen_pack_name;
| [private] generic procedure [parent_name.]id renames
            gen_proc_name;
| [private] generic function  [parent_name.]id renames
            gen_func_name;
| [private] subprogram_spec renames callable_entity_name;
```

DECLARATIONS

```
← BASIC_DECLARATION ::=
  type id is ( \id|'character'\ {,...});
| type id is mod static_expr;
| type id is digits static_expr [rangestatic_s_expr .. static_s_expr];
| type id is [delta static_expr] range static_s_expr .. static_s_expr;
| type id  is delta static_expr digits static_expr [range
            static_s_expr .. static_s_expr];
| type id [discrim] is [abstract] new subtype_id [with record list
            end record];
| type id [discrim] is [abstract] new subtype_id [with null record];
| type id is array_type_defn;
| type id [discrim] is [[abstract] tagged] [limited] record list
            end record;
| type id [discrim] is [abstract] tagged [limited] null record;
| type id is access [all | constant] subtype_id;
| type id is access [protected] procedure [formals];
| type id is access [protected] function [formals] return
            subtype_name;
| task type id [discrim] [is
   {entry id [(discrete_range)] [formals]; | rep_clause}
  [ private {entry id [(discrete_range)] [formals]; | rep_clause}]
  end [id]];
| protected type id [discrim] is
   { subprogram_spec | entry id [(discrete_range)] [formals]; | rep_clause }
  [ private { subprogram_spec | entry id [(discrete_range)] [formals]; |
            id {,id} : [aliased] subtype_id [:= expr]; | rep_clause } ]
  end [id];
| type id [(<>)|discrim];
| type id [(<>)|discrim] is [[abstract] tagged] [limited] private;
| type id [(<>)|discrim] is [abstract] new ancestor_subtype_id with private;
| subtype id is subtype_id;
| id {,id} : [aliased] [constant] subtype_id [:= expr];
| id {,id} : [aliased] [constant] array_type_defn [:= expr];
| task id [is
   {entry id [(discrete_range)] [formals]; | rep_clause}
  [ private {entry id [(discrete_range)] [formals]; | rep_clause}]
  end [id]];
| protected id is
   { subprogram_spec | entry id [(discrete_range)] [formals]; | rep_clause }
  [ private { subprogram_spec | entry id [(discrete_range)] [formals]; |
            id {,id} : [aliased] subtype_id [:= expr]; | rep_clause } ]
  end [id];
| id {,id} : constant := static_expr;
| subprogram_spec [is abstract];
| package_spec;
| id : subtype_name renames object_name;
| id : exception renames exception_name;
| package id renames pack_name;
| subprogram_spec renames callable_entity_name;
| generic package id renames gen_pack_name;
| generic procedure id renames gen_proc_name;
| generic function id renames gen_func_name;
| id {,id} : exception;
| generic {generic_formals|use_clause} subprogram_spec;
| generic {generic_formals|use_clause} package_spec;
| package id is new gen_pack_name [generic_actuals];
| procedure id is new gen_proc_name [generic_actuals];
| function id|op is new gen_func_name [generic_actuals];

← SUBTYPE_ID ::=
  subtype_name
| subtype_name range name'Range[(static_expr)]
| subtype_name range s_expr .. s_expr
| subtype_name [digits|delta] static_expr [range name'Range[(static_expr)]
| subtype_name [digits|delta] static_expr [range s_expr .. s_expr]
| subtype_name (discrete_range {,...})
| subtype_name ([selector_name {|...} =>] expr {,...})

← ARRAY_TYPE_DEFN ::=
  array(subtype_name range <> {,...}) of [aliased] subtype_id
| array(discrete_range {,...}) of [aliased] subtype_id

← DISCRETE_RANGE ::=
  discrete_subtype_id | name'Range[(static_expr)] | s_expr .. s_expr

← DISCRIM ::=
  (id {,id} : [access] subtype_name [:= expr] {; ...})

← LIST ::=
  id {,id} : [aliased] subtype_id [:= expr]; | rep_clause {...}
| {id {,id} : [aliased] subtype_id [:= expr]; | rep_clause {...}}
  case name is
   when \expr|discrete_range|others\ {| ...} => list
   {...}
  end case;
| null;

← DECLARATIVE_ITEM ::=
  basic_declarative_item
| subprogram_body | package_body | task_body | protected_body
| subprogram_spec is separate; | package body id is separate;
| task body id is separate; | protected body id is separate;

← BASIC_DECLARATIVE_ITEM ::=
  basic_declaration | rep_clause | use_clause

← SUBPROGRAM_SPEC ::=
  procedure [parent_name.]id [formals]
| function [parent_name.]id|op [formals] return subtype_name

← FORMALS ::=
  ( id {,id} : [in | in out | out | access] subtype_name [:= expr] {; ...})

← SUBPROGRAM_BODY ::=
  subprogram_spec is
   {declarative_item}
  begin handled_statements
  end [id];

← PACKAGE_SPEC ::=
  package [parent_name.]id is
   {basic_declarative_item}
  [private {basic_declarative_item}]
  end [[parent_name.]id];

← PACKAGE_BODY ::=
  package body [parent_name.]id is
   {declarative_item}
  [begin handled_statements]
  end [[parent_name.]id];

← TASK_BODY ::=
  task body id is
   {declarative_item}
  begin
   handled_statements
  end [id];

← PROTECTED_BODY ::=
  protected body id is
   { subprogram_spec | subprogram_body |
     entry id1 [(for id2 in discrete_range)] [formals] when bool_expr is
      {declarative_item}
     begin handled_statements
     end [id1]; |
```

```
   rep_clause }
 end [id];

← GENERIC_FORMALS ::=
  id {,id} : [in] subtype_name [:= expr];
| type id[(<>)|discrim] is [[abstract] tagged] [limited] private;
| type id[(<>)|discrim] is [abstract] new subtype_name [with private];
| type id is (<>);
| type id is range <>;
| type id is mod <>;
| type id is digits <>;
| type id is delta <> [digits <>];
| type id is array_type_defn;
| type id is access [all | constant] subtype_id;
| type id is access [protected] procedure [formals];
| type id is access [protected] function [formals]
            return subtype_name;
| with subprogram_spec [is \name|<>\];
| with package id is new gen_pack_name \(<>)|[generic_actuals]\;

← GENERIC_ACTUALS ::=
  ([selector_name =>] \expr|var_name|subprog_name|entry_name|
                      subtype_name|pack_inst_name\ {,...})
```

STATEMENTS, EXPRESSIONS

```
← NAME ::=
  id          | op                    | name.all
| name(expr {,...})                   | name(discrete_range)
| name.selector_name                  | name'attribute_designator
| subtype_name(expr|name)             | 'character'
| func_name [([selector_name =>] \expr|var_name\ {,...})]

← SELECTOR_NAME ::=
  id | 'character' | op

← ATTRIBUTE_DESIGNATOR ::=
  id[(static_expr)] | Access | Delta | Digits

← AGGREGATE ::=
  array_aggregate
| ([\expr|subtype_name\ with] [selector_name {|...} =>
            | others => ] expr {,...})
| ([\expr|subtype_name\ with] null record)

← ARRAY_AGGREGATE ::=
  (expr, expr {,...})                 | (expr {,...}, others => expr)
| (\expr|discrete_range|others\ {|...} => expr {,...})

← EXPR ::=
  relation {xor relation}
| relation {and relation}             | relation {and then relation}
| relation {or relation}              | relation {or else relation}

← RELATION ::=
  s_expr [\= | /= | < | <= | > | >=\  s_expr]
| s_expr [not] in name'Range[(static_expr)]
| s_expr [not] in s_expr .. s_expr
| s_expr [not] in subtype_name

← S_EXPR ::=
  [+|-] term {\+|-|&\ term}

← TERM ::=
  factor {\*|/|mod|rem\ factor}

← FACTOR ::=
  primary [** primary] | abs primary | not primary

← PRIMARY ::=
  numeric_literal | null | string_literal | aggregate | name
| subtype_name'(expr) | subtype_name'aggregate | new subtype_id
| new subtype_name'(expr) | new subtype_name'aggregate | (expr)

← STATEMENT ::=
  [<<label>>] program_statement

← PROGRAM_STATEMENT ::=
| var_name := expr;                  | exit [loop_name] [when bool_expr];
| goto label;                        | null;
| return [expr];                     | entry_call_statement
| delay_statement                    | requeue entry_name [with abort];
| abort task_name {,...};           | raise [exception_name];
| subtype_name'(expr);               | subtype_name'aggregate;
| proc_name [([selector_name =>] \expr|var_name\ {,...})];
| if bool_expr then
    statement {...}
   {elsif bool_expr then statement {...}}
   [else statement {...}]
   end if;
| case expr is
    when \expr|discrete_range|others\ {| ...} => statement {...}
    {...}
   end case;
| [id:] [while bool_expr | for id in [reverse] discrete_range] loop
    statement {...}
   end loop [id];
| [id:] [declare {declarative_item}]
   begin handled_statements
   end [id];
| accept id [(expr)] [formals] [do handled_statements end [id]];
| select_statement

← HANDLED_STATEMENTS ::=
    statement {...}
   [exception
    when [id:] \exception_name|others\ {| ...} => statement {...}
    {...}]

← ENTRY_CALL_STATEMENT ::=
  entry_name [([selector_name =>] \expr|var_name\ {,...})];

← DELAY_STATEMENT ::=
  delay [until] delay_expr;

← SELECT_STATEMENT ::=
  select
   [when bool_expr =>]
     accept id [(expr)] [formals] [do handled_statements end [id]];
     [statement {...}]
   | delay_statement [statement {...}]
   | terminate;
  { or
   [when bool_expr =>]
     accept id [(expr)] [formals] [do handled_statements end [id]];
     [statement  {...}]
   | delay_statement [statement {...}]
   | terminate; }
  [ else statement {...} ]
  end select;
| select
   entry_call_statement [statement {...}]
   or delay_statement [statement {...}]
  end select;
| select
   entry_call_statement [statement {...}]
  else statement {...}
  end select;
| select
   \entry_call_statement|delay_statement\ [statement {...}]
  then abort  statement {...}
  end select;
```

REPRESENTATION

```
← REP_CLAUSE ::=
  for local_name'attribute_designator use expr;
| for local_name'attribute_designator use name;
| for first_subtype_local_name use array_aggregate;
| for first_subtype_local_name use record [at mod static_expr;]
   {component_name at static_expr range static_s_expr..static_s_expr;}
  end record;
| for \id|op\ use at expr;

← LOCAL_NAME ::=
  id['attribute_designator] | op['attribute_designator] | library_unit_name
```

LEXICAL

```
id                ::= identifier_letter {[underline] letter_or_digit}
letter_or_digit   ::= identifier_letter | digit
numeric_literal   ::= numeral [.numeral] [exponent] | numeral
                      #base [.base] # [exponent]
numeral           ::= digit {[underline] digit}
exponent          ::= E [[+]|-] numeral
base              ::= extended_digit {[underline] extended_digit}
extended_digit    ::= digit | A | B | C | D | E | F
string_literal    ::= "{"" | non_quote_character}"
comment           ::= --
op                ::= "<" | ">" | "=" | "&" | "<=" | ">=" | "+" | "/" | "*"
pragma            ::= pragma id [([id =>] \name|expr\ {,...})];
```