Fundamentos de Programación con Ada

Javier Miranda, Francisco Guerra
Luis Hernández

Universidad de Las Palmas de Gran Canaria
Islas Canarias
España

jmiranda@iuma.ulpgc.es
fguerra@iuma.ulpgc.es
lhdez@iuma.ulpgc.es

3 de octubre de 2002

Índice general

Capítulo 1

Introducción

Ada es un lenguaje moderno diseñado para facilitar la escritura de aplicaciones de tiempo real y la programación de sistemas grandes[1] (programas de varios millones de líneas). Lo hemos elegido para impartir esta asignatura porque te va a permitir expresar de una manera amigable complejos conceptos de programación. Además, Ada tiene muchas características que ayudan a prevenir y detectar rápidamente los errores de programación.

1.1. Breve historia de Ada

El nombre del lenguaje fue puesto en recuerdo de Augusta Ada Byron, condesa de Lovelace e hija del poeta Lord Byron. Ada trabajó con Charles Babbage en su calculadora mecánica (que se considera el primer ordenador de la historia) y ha sido considerada la primera programadora de la historia.

Ada fue desarrollado porque el departamento de defensa de Estados Unidos descubrió que ninguno de los lenguajes existentes era apropiado para el control de tiempo real de sistemas empotrados grandes (un sistema empotrado es un sistema en el que el ordenador es parte indispensable de él). Por ello, en 1977 creó una convocatoria para ver quien desarrollaba el lenguaje que mejor se adaptaba a sus requisitos. Tras una preselección, quedaron como finalistas cuatro lenguajes, que fueron denominados Azul, Rojo, Amarillo y Verde para mantener el anonimato de sus diseñadores. Finalmente el ganador fue el lenguaje denominado Verde, que

[1] Esto se conoce técnicamente como "Programación a gran escala" (en inglés "*programming in the large*"

había desarrollado la compañía francesa Honeywell Bull bajo la dirección de *Jean Ichbiah*. Esta versión del lenguaje pasó a ser estándar ISO en 1983. Por esta razón es conocida como **Ada 83**.

Posteriormente se realizó una revisión en el lenguaje con el objetivo de incorporar en él características que facilitan el desarrollo de aplicaciones modernas. Esta nueva revisión incorpora, entre otras cosas, el soporte necesario para programación orientada a objetos y sistemas distribuidos. Además, mantiene la compatibilidad con la versión anterior (podemos compilar con la nueva versión las aplicaciones que habíamos hecho con la antigua que siguen funcionando igual) y es estándar desde Enero de 1995, por lo que es conocida como **Ada 95**.

A lo largo de este curso sólo utilizaremos Ada 83, dejándo Ada 95 para cursos más avanzados.

1.2. Nuestro primer programa Ada

Vamos a comenzar escribiendo un programa que simplemente escribe *Hola!* en la pantalla.

```
--  Autor: Javier Miranda
--  Fecha de la ultima modificacion: 5/Septiembre/2000
--  Descripcion: Este es un programa ejemplo que
--               escribe "Hola" en la pantalla.
with Text_IO;
procedure Hola is
begin
   Text_IO.Put_Line("Hola !"); --  Escribo en pantalla
end Hola;
```

Utilicemos este sencillo ejemplo para conocer un poco el lenguaje. Lo primero que encontramos es un comentario. Los comentarios son notas aclaratorias en las que explicamos algo a quien lea el programa. Ada te permite poner un comentario en cualquier línea de tu programa. Los comentarios de Ada se comienzan mediante dos guiones consecutivos `--` y finalizan automáticamente al final de la línea.

Obviamente, si necesitas poner un comentario que abarque varias líneas, deberás indicar el comienzo de comentario en cada una de las líneas. En este primer programa, el primer comentario abarca varias líneas y se utiliza para decir quién escribió el programa, cuando fue la última vez que lo modificó y una breve descripción de lo que hace el programa. Un buen programador debe poner siempre

este tipo de comentarios en todos sus programas.

Igual que escribimos nuestros libros y documentos mediante frases, los programas se escriben también mediante frases. La principal diferencia es que todas las frases que escribimos en los libros las terminamos con un punto, mientras que las frases de Ada se terminan siempre con un punto y coma (;). También, igual que las frases de nuestros libros pueden contener dentro otras subordinadas, las frases escritas con Ada pueden contener otras frases Ada. En este ejemplo encontramos las siguientes frases:

```
with .... ;
procedure .... is begin ..... end .... ;
Text_IO.Put_Line (".....");
```

Las estructura de las dos primeras frases ha sido fijada mediante varias palabras reservadas de Ada. Las palabras reservadas son palabras especiales de Ada que utilizamos para construir nuestros programas. Ada no diferencia entre mayúsculas y minúsculas. Le da igual que escribamos *PROCEDURE*, *PROCedure*, o cualquier otra combinación. Sin embargo, un buen programador de Ada debe seguir siempre unas normas básicas de estilo. Nosotros escribiremos siempre en minúsculas las palabras propias del lenguaje Ada (conocidas como *palabras reservadas*). Las palabras reservadas utilizadas en este ejemplo son **with**, **procedure**, **is**, **begin** y **end**. En el apartado 1.3 veremos cómo decirle a Ada que compruebe que nuestro programa está escrito con buen estilo.

La frase `with ... ;` está acompañada de la palabra *Text_IO*. Esta frase se utiliza para decirle a Ada que queremos utilizar en nuestro programa la biblioteca que se llama *Text_IO* (es una biblioteca que viene incluida en el lenguaje Ada). Una biblioteca es una colección de pequeños programas que han sido escritos por otras personas y que podemos reutilizar para escribir nuestros programas. En este curso vamos a utilizar la biblioteca *Text_IO* para escribir en pantalla, para leer desde el teclado y para guardar datos en el disco.

A continuación comienza realmente el código de nuestro programa con la frase `procedure .... is ... begin ... end;` Esta frase es fundamental, ya que es la que fija la estructura del programa. Después de la palabra **procedure** se debe poner el nombre del programa (cualquier secuencia de letras y números, siempre que comience por una letra). Entre la palabra **is** y la palabra **begin** se colocan los objetos que va a utilizar nuestro programa. Como en este ejemplo no necesitamos utilizar ningún objeto no ponemos nada. Finalmente, entre la palabra **begin** y la palabra **end** pondremos las acciones que realiza nuestro programa.

Una característica importante de todos los lenguajes de programación modernos es que, al escribir el código, nos permiten introducir tantos espacios en blanco

como queramos para sangrar nuestros programas. Fijate en la línea que empieza con la palabra *Text_IO*. Si no hubiésemos sangrado el programa, sería así:

```
begin
Text_IO.Put_Line("Hola !");
end Hola;
```

Cuando conozcas todo lo que puedes escribir dentro del código del procedimiento comprenderás mucho mejor lo importante que es sangrar muy bien tus programas para mejorar su lectura. Debes tener siempre en cuenta que los programas se leen muchisimas veces (puede que un programa que hagas hoy necesites reutilizarlo dentro de dos años y ya no recuerdes bien lo que hace), por lo que debes aprovechar todos los recursos que proporcione el lenguaje para mejorar su legibilidad. Lo habitual es añadir tres espacios en blanco cada vez que quieras aumentar el sangrado del programa.

En este ejemplo, el código de nuestro programa es una única frase. La frase `Text_IO.Put_Line("....");`. Con esta frase estamos diciendo que queremos utilizar *Put_Line*, que está en la biblioteca *Text_IO*. Dentro de las comillas ponemos el texto que queremos que escriba el programa en la pantalla de nuestro ordenador.

1.3. Uso del compilador: GNAT

Para escribir nuestros programas Ada podemos utilizar cualquier editor de textos. Sin embargo, escribir nuestros programas no es suficiente. Ada es un **lenguaje de programación de alto nivel** (un lenguaje de programación con gran potencia expresiva y fácil de comprender por las personas), mientras que el lenguaje de nuestro ordenador es un **lenguaje de programación de bajo nivel** (un lenguaje muy cercano al lenguaje de la máquina y, por tanto, difícil de comprender por las personas). Para facilitar la traducción de nuestros programas Ada de un lenguaje a otro utilizaremos un compilador. El **compilador** es un programa que lee nuestro programa Ada y genera un fichero ejecutable. El compilador que vamos a utilizar se llama GNAT (acrónimo de *GNU New York University Ada Translator*). Para compilar debemos dar la siguiente órden desde el Sistema Operativo:

```
$ gnatmake -gnatg hola.adb
```

Con esta órden estamos diciendo: “GNAT, construye mi programa hola”. También podemos compilar utilizando la órden `gnatmake -gnatg hola`, porque

GNAT siempre busca nuestro programa Ada en un fichero que termine con la extensión **.adb**. Con el indicador `-gnatg` estamos pidiendo al compilador que compruebe el estilo de nuestro programa. Esto significa que considerará un error que el programa no esté escrito con buen estilo. Las normas básicas que debemos seguir para evitar errores de estilo son:

1. Sangrar siempre utilizando un número de espacios múltiplo de 3.
2. Añadir 2 espacios después del aviso de comienzo de comentario.

```
--  Linea de comentario correcta.
-- Linea de comentario con mal estilo.
```

3. Añadir 1 espacio alrededor de cada operador ($+, -, *, /$).

```
3+4*2        --  Operadores mal sangrados
3 + 4 * 2    --  Operadores bien sangrados
```

GNAT analiza el programa que hemos escrito y busca posibles fallos. Hay dos tipos de fallos:

- **Aviso** (*warning*). Es un fallo leve. Se utiliza para indicar que el fallo, aunque leve, puede a veces ser grave.
- **Error**. Es un fallo grave. En este caso el compilador no genera ningún fichero ejecutable, obligándonos a revisar el código para corregirlo.

En ambos casos GNAT nos indica el número de línea donde está el error y el carácter dentro de esa línea. Por ejemplo:

```
ejemplo.adb:6:11: warning: file name does not match unit name
```

Con este mensaje GNAT nos está diciendo que hemos cometido un error leve que está en la línea 6, carácter 11 del fichero "ejemplo.adb". Nos está diciendo que, como el programa se llama "hola" (ya que es el nombre que pusimos al procedimiento), el nombre del fichero debe ser "hola.adb" (en minúsculas).

Si el programa no tiene errores, cuando GNAT finaliza su trabajo genera un fichero con el mismo nombre, pero sin ninguna extensión[2]. En nuestro caso genera el fichero **hola**.

[2]En caso de que estemos compilando sobre Windows, genera un fichero con la extensión ".EXE".

Para ejecutar el programa sólamente es necesario decírselo al sistema operativo especificando su nombre. Por ejemplo, en Linux:

```
$ ./hola
```

1.4. Identificadores

Los identificadores son los nombres que damos a nuestros programas y a todos los objetos que coloquemos dentro de ellos. Deben comenzar siempre por una letra, seguida de todas las letras, números y subrayados que queramos. Todo subrayado debe siempre ir seguido de una letra o un número ya que, como Ada no permite que los identificadores tengan espacios, utilizamos el subrayado como un separador para facilitar la lectura de los identificadores. Por ejemplo: *Radio_Circulo, Total_Acumulado.*

Los identificadores pueden ser tan largos como queramos (el máximo depende del compilador que utilicemos).

Las palabras reservadas de Ada no pueden utilizarse como identificadores y son las siguientes:

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

En general, si elegimos buenos nombres para nuestros identificadores (nombres largos que dejen bien claro el contenido del identificador) evitamos muchos comentarios innecesarios. Por ejemplo, en vez de escribir:

```
E := 19;      --  La variable E representa la edad
              --  y la iniciamos con el valor 19.
```

...es muchisimo mejor escribir:

```
Edad := 19;
```

Una ventaja importantísima de poner buenos identificadores es que, si posteriormente decidimos cambiar el valor 19 por 20, en el primer caso tendremos que acordarnos de corregir también el comentario para evitar que se quede obsoleto.

1.5. Resumen

En este capítulo nos hemos acercado a Ada: un lenguaje moderno para programar todo tipo de aplicaciones. Hemos visto cómo debemos estructurar nuestros programas, cómo poner comentarios, y cómo elegir buenos identificadores. También hemos visto que el compilador es una herramienta que traduce nuestro programa en un fichero ejecutable y hemos aprendido a utilizar GNAT para compilar nuestros programas.

En el próximo capítulo comenzaremos a ver el lenguaje Ada con más detalle. Veremos qué son las variables, cómo se declaran y utilizan, qué son los tipos de datos, cómo podemos crear subtipos y tipos derivados y qué son los atributos.

Capítulo 2

Declaraciones

Ada tiene unos tipos de datos básicos que podemos utilizar en nuestros programas. Además estos tipos básicos, Ada nos permite añadir nuevos tipos de datos. En este capítulo veremos cómo declarar constantes y variables, cómo definir nuevos tipos de datos, cómo derivar un nuevo tipo de dato a partir de otro ya existente y cómo consultar información de los tipos de datos (a través de lo que se conoce en Ada como atributos).

2.1. Constantes y variables

Cada vez que queramos recordar algo, deberemos decirle a Ada que necesitamos utilizar la memoria del ordenador. La variable es el objeto básico de Ada para almacenar información en la memoria del ordenador. Para evitar errores de programación, cada variable tiene asociado un identificador (un nombre) y un tipo de dato (entero, natural, carácter, etc.) que indica el tipo de información que puede guardar. Hay dos tipos de datos que podemos guardar: datos constantes (datos que no cambian de valor durante toda la ejecución del programa) y datos variables (datos que pueden cambiar de valor durante la ejecución del programa).

Ambos tipos de datos deben declararse en el mismo sitio; entre las palabras **is** y **begin**.

```
procedure ... is
   --  Aqui deben colocarse las declaraciones
   --  de constantes y variables.
   ...
begin
   --  Aqui pueden utilizarse las constantes y
   --  variables que hayamos declarado entre las
   --  palabras is y begin.
   ...
end ... ;
```

Si entre las palabras *begin* y *end* queremos volver a declarar más variables utilizamos una frase declare. El siguiente ejemplo contiene un *declare*. Obviamente, podemos poner todos los declare que necesitemos.

```
procedure ... is
   --  Aqui deben colocarse las declaraciones
   --  de constantes y variables.
   ...
begin
   --  Aqui pueden utilizarse las constantes y
   --  variables que hayamos declarado entre las
   --  palabras is y begin.
   ...
   ...
   declare
      --  Aqui declaro mas variables.
      ...
   begin
      --  El codigo que va aqui puede utilizar todas
      --  las variables que he declarado. Las del
      --  procedimiento y tambien las del declare.
      ...
   end;
   ...
   ...
end ... ;
```

En los siguientes apartados veremos cuales son los tipos de datos básicos de Ada, cómo le decimos a Ada que un dato es constante o variable y cómo podemos utilizarlo.

2.1.1. Formato de las declaraciones

Todas las declaraciones de constantes y variables se hacen con la misma frase:

```
Nombre_Variable : Tipo_De_Dato;
```

Primero debemos dar un nombre a nuestra constante o variable; a continuación debemos poner el carácter dos puntos (:) seguido del tipo de dato de la constante o variable. Para terminar debemos poner siempre el carácter punto y coma (;).

Ada también nos permite declarar varias variables del mismo tipo de una sóla vez. Para ello sólo tenemos que poner una lista con los nombres de las variables separados mediante comas. Por ejemplo:

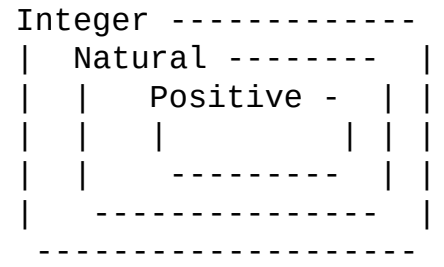
```
Total, Contador, Maximo : Integer;
```

2.1.2. Tipos de datos básicos

Ada tiene seis tipos de datos básicos:

- **Integer**: Para cualquier número entero positivo o negativo.
- **Natural**: Más restrictivo que el anterior. Solamente permite el cero y los números positivos.
- **Positive**: Más restrictivo aún que el anterior. Sólamente permite los números positivos.

  En realidad, como veremos en el apartado 2.2.2, *Natural* y *Positive* son dos subtipos de *Integer* que limitan su rango de valores.



  Podriamos preguntarnos: ¿Para qué hacen falta tres tipos de datos diferentes si puedo trabajar siempre con *Integer* (ya que engloba a los tres)?. Ada tiene los tres tipos de datos para facilitar la detección de errores en nuestros programas. Por ejemplo, si estoy haciendo un programa que me pide elegir un número entre el 1 y el 10, utilizaré una variable de tipo Positive (con lo que Ada comprueba que nunca se trabaje con números negativos ni con el cero). Sin embargo, si necesito guardar la edad de una persona (en años),

puedo utilizar una variable de tipo Natural; y si estoy haciendo un programa de contabilidad puedo utilizar variables de tipo Integer. La correcta elección del tipo de dato permite que Ada nos ayude a escribir programas correctos.

- **Float**: Para almacenar cualquier número real en formato *coma flotante*. Los números en coma flotante debe tener como mínimo un digito antes y después del punto decimal. Por ejemplo, el número *1.0*

  Ada nos permite utilizar la notación científica. De esta forma *1.3E-3* es lo mismo que *0.0013* y *12E3* es lo mismo que 12000.

- **Character**: Para almacenar una única letra (un carácter).

- **String**: Para almacenar una frase completa. Como Ada no sabe cual es el tamaño máximo que queremos permitir, debemos decirselo en el momento de declarar la variable. Por ejemplo, con la siguiente frase decimos a Ada que queremos una variable de tipo String que sirva para guardar como máximo 40 caracteres.

```
Nombre_Completo : String (1 .. 40);
```

2.1.3. Constantes

Para declarar constantes, justo antes de especificar el tipo de dato debemos utilizar la palabra reservada *constant*. Por ejemplo:

```
procedure Ejemplo_Declaraciones is
   Edad          : constant Natural := 18;
   Primera_Letra : constant Character := Á';
   Pi            : constant Float := 3.141592654;
   Curso         : constant String (1 .. 7) := "Primero";
begin
   ...
end Ejemplo_Declaraciones;
```

En las constantes String, no es necesario que digamos su tamaño. En este caso Ada analiza la declaración y asigna a la constante el tamaño que le corresponde. En el siguiente ejemplo, Ada asigna automáticamente el tamaño 10 a la constante.

```
Siglas_Escuela : constant String := "E.T.S.I.T.";
```

Ada analiza todas las declaraciones en orden (de arriba a abajo). Esto nos permite reutilizar las constantes para declarar otras. Por ejemplo:

```
Pi     : constant Float := 3.141592654;
Dos_Pi : constant Float := 2.0 * Pi;
```

En las constantes numéricas, a diferencia de los identificadores, el subrayado no es significativo. De esta forma 12345 y 12_345 son equivalentes. Esto nos permite escribir los números de forma que se puedan leer con mayor claridad. Por ejemplo, podemos escribir 23_595_324_456, en vez de escribir 23595324456.

Por último, conviene saber que Ada nos permite expresar los números en cualquier base entre 2 y 16. Para ello sólo tenemos que poner el número entre dos caracteres # precedidos por la base elegida. Por ejemplo el número 64151 se puede expresar en base 16, base 2 y base 8 de la siguiente forma:

```
16#FC03#     2#1111_1100_0000_0011#      8#176003#
```

2.1.4. Variables

Las variables son objetos cuya información puede cambiar durante la ejecución del programa. La forma de declarar variables es muy parecida a la de constantes. La principal diferencia es que, obviamente, no debemos añadir la palabra constant. Por ejemplo:

```
with Text_IO;
procedure Ejemplo_Variables is
   Edad     : Natural;
   Letra    : Character;
   Linea    : String (1 .. 80);
   Longitud : Natural;
begin
   ...
end Ejemplo_Variables;
```

2.1.5. Uso combinado de constantes y variables

Como es lógico, en nuestros programas podemos utilizar todas las declaraciones de variables y constantes que queramos. Veamos un ejemplo de declaración y uso de variables y constantes.

```
1: procedure Ejemplo_Declaraciones is
2:    Numero, Total    : Integer;
```

```
 3:    Indice           : Integer := 30;
 4:    Resultado, Valor : Float;
 5:    Factor           : constant Positive := 1000;
 6: begin
 7:    Numero    := Indice + 10;
 8:    Resultado := 0.0;
 9:    Valor     := Resultado + Indice;
                    -- ILEGAL. No puedo mezclar tipos.
10:    Valor     := Resultado + Float (Indice);
11:    Total     := Integer (Resultado) + Indice;
12:    Factor    := 10;
                    -- ILEGAL. FACTOR es una constante.
13: end Ejemplo_Declaraciones;
```

Veamos en detalle este ejemplo. Comencemos con la sección de declaraciones. En la línea 2 declaramos dos variables enteras (Numero y Total); en la línea 3 declaramos una variable más (Indice) y la inicializamos al valor 30. En la línea 4 declaramos dos variables de tipo flotante y en la línea 5 declaramos una constante positiva.

Pasemos ahora a las sentencias. En las líneas 7 y 8 vemos dos frases de asignación sencillas (una con enteros y otra con flotante). En la línea 9 el compilador nos dará un error porque Ada no nos permite mezclar de forma accidental datos de diferentes tipos. La línea 10 muestra cómo debe hacerse esta mezcla de tipos de forma intencionada (realizando una conversión de tipo). La línea 11 muestra otra posible conversión de tipos. Sin embargo, debemos recordar que cuando convertimos un dato de tipo *Float* a *Integer*, el compilador realiza un redondeo (por ejemplo, sin convertimos a entero el número 3.4 obtenemos el 3, pero si convertimos a entero el número 3.6 obtenemos el 4). Finalmente la línea 12 es ilegal porque no podemos almacenar un resultado en una constante.

2.2. Tipos

En este apartado vamos a saber cómo podemos crear nuevos tipos de datos. Comenzaremos por los tipos enumerados (tipos de datos en los que debemos dar a Ada una lista con todos los valores posibles). Después veremos cómo crear subtipos y tipos derivados a partir de tipos ya existentes.

Antes de comenzar debes tener en cuenta que, siempre que crees nuevos tipos de datos, es recomendable le des un nombre que comience por la letra “T” seguida de un subrallado. Por ejemplo, *T_Colores*, *T_Persona*, etc. De esta forma consigues dos ventajas importantes:

- Es muy fácil encontrar dónde están declarados tus tipos.
- Puedes utilizar el mismo nombre del tipo (quitando el prefijo "T_") para declarar tus variables. Por ejemplo, *Colores : T_Colores*.

2.2.1. Tipo enumerado

Ada nos permite especificar todos los posibles valores de un tipo. A este nuevo tipo se le llama tipo enumerado. Por ejemplo:

```
 1:  procedure Uso_De_Colores is
 2:     type T_Colores is (Rojo, Amarillo, Verde, Azul);
 3:     Color: T_Colores;
 4:  begin
 5:     Color := Rojo;
 6:     ...
 7:     if Color = Rojo then
 8:        ...
 9:     end if;
10:   end Uso_De_Colores;
```

En la línea 2 definimos un nuevo tipo de datos. Se llama *T_Colores* y sus posibles valores son *Rojo, Amarillo, Verde* y *Azul*. En la línea 4 declaramos la variable *Color*, que puede almacenar cualquiera de los 4 valores del tipo *T_Colores*. En la línea 5 guardamos en la variable *Color* el valor *Rojo* y en la línea 7 consultamos el valor (mediante una frase *if*, que veremos en detalle en el capítulo 5).

La principal ventaja de crear tipos de datos propios para nuestro programa es que Ada nos facilita la detección de errores. Por ejemplo, el programa anterior es equivalente al siguiente:

```
 1:  procedure Uso_De_Colores is
 2:     Rojo     : constant Positive := 0;
 3:     Amarillo : constant Positive := 1;
 4:     Verde    : constant Positive := 2;
 5:     Azul     : constant Positive := 3;
 6:     Color    : Positive;
 7:  begin
 8:     Color := Rojo;
 9:     ...
10:     if Color = Rojo then
11:        ...
12:     end if;
13:   end Uso_De_Colores;
```

Sin embargo, este programa es peor que el anterior porque en la línea 8 podriamos asignar a *Color* el valor 12 y Ada no podría ayudarnos a detectar el error (el 12 es un número positivo y *Color* es una variable que almacena valores positivos, por lo que la operación es correcta). Además, este programa permite sumar, restar, multiplicar y dividir colores. ¿ Tiene eso algún sentido ? (la suma y la resta quizás si, pero el resto de las operaciones no sabemos qué pueden significar).

Los valores enumerados no tienen que ser siempre identificadores. También pueden ser caracteres. Por ejemplo:

```
type T_Digito_Par is ('0', '2', '4', '6', '8');
```

Lo que no permite Ada es mezclar caracteres e identificadores en el mismo tipo enumerado. Por ejemplo:

```
type T_Mezcla is (Grande, Pequeno, 'X', '9');   --  MAL !!!
```

Si dos tipos de datos diferentes tienen elementos cuyo nombre coincide, el uso de este elemento es ambiguo (Ada no sabe a cual de ellos nos referimos). Por ejemplo:

```
type T_Colores is (Rojo, Amarillo, Naranja, Verde, Azul);
type T_Frutas  is (Fresa, Naranja, Manzana);

--  Cuando utilicemos el elemento Naranja, podemos
--   referirnos a un color o a una fruta.
```

Para resolver la ambiguedad debemos decir a Ada a qué enumerado nos referimos. Esto se conoce como cualificar el elemento con el tipo de dato al que pertenece. Por ejemplo:

```
T_Colores'(Naranja) --  Para referirnos al color.
T_Fruta'(Naranja)   --  Para referirnos a la fruta.
```

Como Ada asocia un valor numérico (el orden) a cada uno de los valores posibles de un tipo enumerado, podemos comparar los valores de un enumerado mediante los operadores $<$, $>$, etc. Por ejemplo:

```
Rojo > Amarillo
   --  Falso, porque la posicion de rojo
   --  en nuestra declaracion no es posterior
   --  a la de amarillo.
```

2.2.2. Subtipos

Además de los tipos básicos (*Integer, Natural, Positive, Float, Character, String*), Ada nos permite crear subtipos que limiten el rango de valores de nuestras variables. Por ejemplo:

```
subtype T_Dia is Positive range 1 .. 31;
Dia : T_Dia;

type T_Colores is (Rojo, Amarillo, Verde, Naranja, Azul);
Color: T_Colores;

subtype T_Color_Semaforo is Rojo .. Verde;
Semaforo: T_Color_Semaforo;

subtype T_Probabilidad is Float range 0.0 .. 1.0;
Probabilidad : T_Probabilidad;
```

Cada vez que guardemos un valor en una variable, Ada comprueba que el valor está en el rango permitido para su tipo. Si no lo está, eleva la excepción **Constraint_Error**. Las exepciones son los errores que genera Ada cuando al ejecutar el programa detecta algún fallo en el programa. En el capítulo 8 veremos cómo podemos capturar las excepciones desde nuestros programas Ada.

2.2.3. Tipos derivados

Para prevenir la mezcla accidental de variables, Ada nos permite crear tipos derivados. A diferencia de los subtipos, los tipos derivados son nuevos tipos. Los tipos derivados se declaran utilizando la palabra reservada **new** seguida de un tipo existente (por ejemplo, *Integer*). El nuevo tipo hereda las operaciones del tipo base (suma, resta, etc.). Por ejemplo:

```
 1:  procedure Tipos_Derivados is
 2:     type T_Numero_Manzanas is new Natural;
 3:     type T_Numero_Naranjas is new Natural range 0 .. 100;
 4:
 5:     Num_Manzanas: T_Numero_Manzanas;
 6:     Num_Naranjas: T_Numero_Naranjas;
 7:     Total       : Integer;
 8:  begin
 9:     Num_Manzanas := Num_Naranjas;            -- Ilegal
10:     Num_Manzanas := Total;                   -- Ilegal
11:     Num_Manzanas := T_Numero_Manzanas(Total);
12:     Num_Manzanas := T_Numero_Manzanas(Num_Naranjas);
13:  end Tipos_Derivados;
```

Como vemos en este ejemplo, el tipo derivado impide mezclar tipos accidentalmente (líneas 9 y 10). En el fondo, en este ejemplo todos los datos son números enteros, pero hemos dicho a Ada que queremos diferenciar conceptualmente unos de otros para no mezclar las manzanas con las naranjas. Si queremos mezclar variables de tipos derivados diferentes Ada hacerlo nos permite mezclarlos haciendo una conversión de tipo (líneas 11 y 12). Esta mezcla sólo es posible cuando ambos tipos han sido derivados de un mismo tipo (en este ejemplo todos provienen de *Integer*, ya que *Natural* es un subtipo de los enteros).

En resumen:

- Los subtipos se utilizan para restringir rangos.
- Los tipos derivados se utilizan para prevenir la mezcla accidental de objetos.

2.2.4. Ajuste de la precisión de números en coma flotante

Ada nos permite fijar la precisión de nuestras variables de punto fijo. Para hacerlo creamos un nuevo tipo utilizando además la palabra reservada **delta** y un rango. Por ejemplo:

```
type T_Voltaje is delta 0.01 range -20.0 .. 20.0;
```

Esto garantiza que las variables de tipo *T_Voltaje* tendrán una precisión de al menos 1/100. Si pedimos una precisión que no puede garantizar el compilador, nos avisará cuando compilemos el programa.

2.3. Atributos

Todos los tipos de datos de Ada tienen asociados unos atributos. Llamamos atributos a unas funciones especiales de Ada que proporcionan información de los tipos de datos y realizar algunas conversiones de tipos de datos. La forma de utilizarlos es poner un apóstrofe después del nombre del tipo y el nombre del atributo que queremos consultar. Veámoslo en los siguientes apartados.

First, Last

Para saber cual es el primer y el último valor de un determinado tipo de dato utilizamos los atributos **First** y **Last**.

```
Maximo := Natural'Last;
Minimo := Integer'First;
```

Succ, Pred

Los atributos **Succ** y **Pred** proporcionan el predecesor y sucesor de un determinado valor. Si intentamos obtener el sucesor del último valor o el predecesor del primer valor, Ada eleva la excepción **Constraint_Error**.

Como hemos visto los tipos enumerados resultan especialmente útiles para crear tablas. Para hacerlo hacemos que cada valor de la tabla sea un elemento del tipo enumerado. Por ejemplo:

```
type T_Colores is (Rojo, Amarillo, Naranja, Verde, Azul);
type T_Frutas  is (Fresa, Naranja, Manzana);

T_Colores'Succ (Amarillo) --  El sucesor de Amarillo es
                          --  Naranja.
T_Frutas'Pred (Azul)      --  El anterior a Azul es Verde
```

Si nos acostumbramos a utilizar estos atributos con los tipos enumerados conseguimos que nuestros programas pueda recorrer fácilmente todos los elementos de la tabla incluso cuando, en el futuro, cambiemos el número de elementos de la tabla. Para saber si hemos llegado al principio o al final de la lista, compararemos el valor del siguiente elemento o el anterior con el valor que proporcionen los atributos *First* y *Last*.

Pos, Val

El atributo **Pos** nos dice la posición de un elemento de tipo enumerado en la declaración del tipo. Para realizar el trabajo inverso utilizaremos el atributo **Val**, que nos da el elemento que está en la posición que digamos. (Debemos tener siempre en cuenta que Ada comienza a contar la posición de los elementos del enumerado desde cero). Por ejemplo:

```
T_Colores'Pos (Rojo)     --  vale 0
T_Colores'Pos (Amarillo) --  vale 1

T_Colores'Val (3)        --  vale Verde
T_Colores'Val (4)        --  vale Azul
```

De forma similar,

```
Character'Pos(Á') vale 65
Character'Pos('B') vale 66
....
```

Esto es debido a que la posición del carácter Á' dentro de la tabla ASCII (el enumerado Character) es 65.

Image, Value

Igual que *Pos* y *Val* convierten desde y hacia enteros, los atributos **Image** y **Value** convierten desde y hacia String. Por ejemplo:

```
type T_Colores is (Rojo, Amarillo, Naranja, Verde, Azul);

--  T_Colores'Value ("Rojo") es el elemento Rojo
--  T_ColoresÍmage (Rojo)   es la string "Rojo"

--  Integer'Value ("123") es el entero 123
--  IntegerÍmage (123)   es la string "123"
```

En caso de que no pueda realizarse la conversión, Ada eleva la excepción **Constraint_Error**.

El uso más frecuente de estos atributos es simplificar la escritura de números en pantalla. Por ejemplo, para escribir números sin necesidad de crear ejemplares del paquete genérico *Text_IO.Integer_IO* podemos utilizar el atributo *Image*.

```
with Text_IO;
procedure Ejemplo_Images is
   Contador : Natural := 12;
   Total    : Float := 312.4;
begin
   Text_IO.Put (IntegerÍmage (Contador));
   Text_IO.Put (FLoatÍmage (Total));
end Ejemplo_Images;
```

2.4. Resumen

En este capítulo hemos visto cómo se declaran constantes y variables en Ada. Además hemos conocido los tipos enumerados: cómo se declaran y utilizan y cómo se leen desde teclado y escriben en pantalla variables de tipos enumerados. También hemos conocido los principales atributos de Ada, que nos permiten escribir programas que recorran los tipos enumerados de forma general (sin necesidad de conocer a priori los elementos exactos del enumerado).

También hemos conocido los subtipos (que se utilizan para restringir el rango de nuestras variables numéricas) y los tipos derivados (que nos permiten crear nuevos tipos de datos utilizando otros ya existentes).

En el siguiente veremos cual es el soporte básico que proporciona Ada para realizar operaciones matemáticas.

First	Primer elemento.
Last	Último elemento.
Pred	Anterior.
Succ	Siguiente.
Val	Valor.
Pos	Posición.
Image	Conversión a string.
Value	Conversión a tipo discreto.

Cuadro 2.1: Resumen de atributos

Capítulo 3

Operadores básicos

En este capítulo aprenderemos a realizar calculos aritméticos y lógicos básicos con Ada.

3.1. Operadores aritméticos

Los lenguajes de programación permiten utilizar los cuatro operadores aritméticos básicos (+, -,*, /). En Ada podemos utilizarlos con cualquier constante o variable de tipo entero o flotante. Podemos mezclar variables de subtipos (ya que son variables del mismo tipo, cuya única diferencia es el margen de valores), pero no podemos mezclar directamente variables de tipos diferentes. Por ejemplo:

```
procedure Ejemplo_Operadores is
   Numero : Natural := 3;
   Total  : Integer := 0;
   Maximo : Float;
begin
   Total  := Total + 4 * Numero;   --  Correcto
   Maximo := 3 * Total;            --  ILEGAL !!
end Ejemplo_Operadores;
```

La primera expresión es correcta ya que Ada puede realizar el producto de la constante 4 por el contenido de la variable natural *Número* (obteniendo como resultado el número natural 12) y sumarlo con el contenido de la variable entera total, ya que el natural es un subtipo de los enteros. Sin embargo, el resultado de multiplicar *Total* por la constante entera 3 no podemos guardarla en la variable

Máximo porque es de tipo *Float*. Si queremos guardar el contenido de una variable entera en una variable *float* (o viceversa) debemos utilizar la conversión de tipos. Para convertir un tipo numérico en otro tipo numérico debemos anteponer el nombre del tipo resultante y poner entre paréntesis la variable (o expresión) que queremos convertir. Por ejemplo:

```
Maximo := Float (3 * Total);    --  Correcto.
```

El operador numérico **abs** devuelve el valor absoluto de un valor de tipo **Integer** o **Float**[1]

Cuando realizamos la división entre dos enteros, el operador / solamente proporciona la parte entera del cociente. Por ejemplo, `9 / 4` es 2. Los operadores **mod** (módulo) y **rem** (remainder) proporcionan el resto de la división. Por ejemplo, `9 mod 4` es 1. Si los dos datos son positivos *mod* y *rem* producen el mismo resultado, pero con valores negativos *mod* da el signo del denominador y *rem* da el signo del numerador.

El operador de exponenciación se expresa mediante dos asteriscos consecutivos (******). Requiere una base de tipo *Integer* o *Float* y un exponente de tipo *Integer* y retorna un valor del mismo tipo que la base. Por ejemplo:

```
procedure Ejemplo_Exponenciacion is
   Numero_1 : Natural;
   Numero_2 : Float;
begin
   Numero_1 := 3 ** 3;
   Numero_2 := 3.0 ** 3;
end Ejemplo_Exponenciacion;
```

Al ejecutar este programa en la variable *Numero_1* se guarda el valor natural 9 y en la variable *Número_2* el valor 9,0.

3.2. Operadores relacionales

Los operadores relacionales de Ada devuelven un valor de tipo **Boolean**. Son los siguientes:

[1]Como es un operador unario, podemos omitir los paréntesis y escribir B := abs A.

```
<  menor que           >  mayor que           =  igual a
<= menor o igual que   >= mayor o igual que   /= no igual a
```

Por ejemplo:

```
procedure Ejemplo_Relacionales is
   Resultado : Boolean;
   Numero_1  : Positive := 23;
begin
   Resultado := 12 > 3;         --  True
   Resultado := Numero_1 = 4;   --  Falso
end Ejemplo_Relacionales;
```

3.3. Operadores lógicos

Los operadores lógicos te permiten realizar operaciones con las constantes y variables de tipo lógico (*Boolean*). Los operadores lógicos de Ada son **and** (y), **or** (o), **xor** (o exclusivo) y **not** (negación). A continuación se muestra la tabla de verdad de cada uno.

```
  X     Y    |   and     or    xor              X    |  not
----- -----  |  -----  -----  -----           ------ | -----
False False  |  False  False  False           False  | True
False True   |  False  True   True            True   | False
True  False  |  False  True   True
True  True   |  True   True   False
```

3.4. Operador concatenación

Cuando quieras concatenar dos cadenas de caracteres puedes utilizar el operador **&**. Por ejemplo:

```
"Esto es" & "un ejemplo"
```

. . . es equivalente a escribir

```
"Esto es un ejemplo"
```

Generalemente este operador se utiliza cuando necesitamos escribir un mensaje muy largo en pantalla. Por ejemplo:

```
Text_IO.Put_Line ("Esto es un ejemplo de un mensaje "
                  & "largo que ocupa varias lineas en "
                  & "el programa. Sin embargo, como todo "
                  & "el mensaje se vuelca en pantalla "
                  & "mediante un unico Put_Line, saldra "
                  & "todo en la misma linea");
```

Otro uso frecuente del operador concatenación es para combinar la escritura de texto y números. Para hacerlo necesitamos utilizar el atributo **Image** (visto en el apartado 2.3). Por ejemplo:

```
Text_IO.Put_Line ("Contador vale "
                  & IntegerÍmage (Contador));
```

3.5. Evaluación óptima de expresiones

Al utilizar los operadores lógicos **or** y **and**, a veces no es necesario evaluar toda la expresión para saber el resultado definitivo. Por ejemplo:

```
if (Funcion_1 (X) > 0) or (Funcion_2 (X) < 13) then
   -----;
   -----;  (bloque de codigo)
   -----;
end if;
```

Si analizamos la tabla de verdad del operador lógico *A or B* (apartado 3.3) vemos que si A es cierto, el resultado ya es definitivamente cierto. Sin embargo, Ada no sabe si nosotros queremos que evalúe B (aunque acabamos de ver que, generalmente, no es necesario). Para decirle a Ada que queremos hacer una evaluación óptima de expresiones lógicas, en vez de utilizar el operador *or*, debes utilizar el operador **or else**. Por ejemplo:

```
if D /= 0.0 or else N / D >= 10.0 then
   -----;
   -----;  (bloque de codigo)
   -----;
end if;
```

Ada garantiza que la expresión a la izquierda de **or else** se evaluará antes que la de la derecha. Si esta expresión es cierta (*True*), la expresión completa será cierta (por lo que Ada no realiza la evaluación de la expresión a la derecha); si es falsa, entonces es necesario evaluar la expresión de la derecha. Aprovechándo ésta característica podemos asegurar que el código anterior nunca realizará una división por cero.

El operador complementario de *or else* es **and then**. En este caso si la expresión a la izquierda de *and then* es falsa, Ada detiene la evaluación del resto de la expresión porque sabe que el resultado es falso; si es cierta, Ada tendrá que evaluar la expresión de la derecha.

3.6. Resumen

En este capítulo hemos conocido los operadores de Ada para calcular operaciones aritméticas y lógicas. En el siguiente capítulo veremos cómo se realiza la entrada/salida de datos en nuestros programas Ada.

Capítulo 4

Entrada/Salida

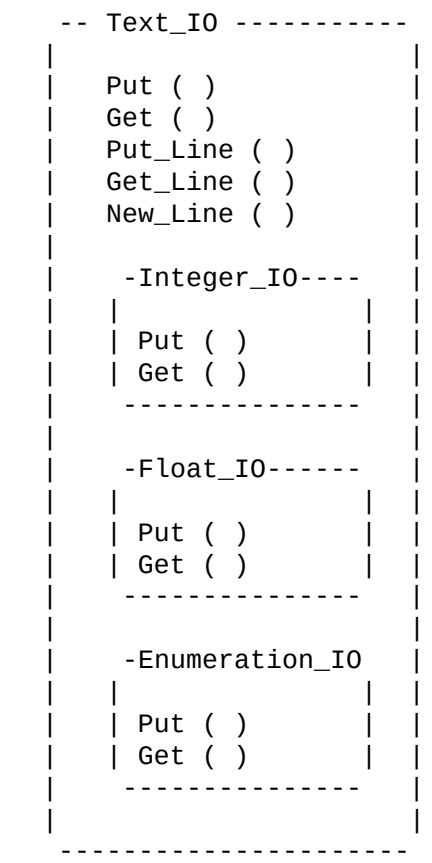
El ordenador necesita algún mecanismo que le permita comunicarse con el exterior. Por eso tiene dos dispositivos básicos que facilitan la entrada y salida de información: el teclado y la pantalla.

Todos los lenguajes de programación proporcionan alguna forma de leer información del teclado y escribir información en la pantalla. En este capítulo veremos cómo se hace con Ada. En Ada todo el control básico del teclado y la pantalla se realiza mediante la biblioteca **Text_IO**, que pasamos a describir.

4.1. Estructura de Text_IO

La biblioteca Text_IO está estructurada en dos niveles. En el primer nivel, el nivel llamado **Text_IO**, están los procedimientos encargados de leer una única letra (un carácter) y los procedimientos encargados de leer y escribir frases completas.

En el segundo nivel hay tres partes: una llamada **Integer_IO**, que contiene los procedimientos para leer y escribir números enteros, otra llamada **Float_IO**, para leer y escribir números en coma flotante y otra llamada **Enumeration_IO**, para leer y escribir valores de tipos enumerados.



Integer_IO, Float_IO y Enumeration_IO son paquetes genéricos Ada. Un paquete genérico Ada es un paquete especial de Ada que nos permite crear diferentes versiones (que llamaremos ejemplares) que permitan trabajar con diferentes rangos y tipos de datos. Por ejemplo, podemos crear un ejemplar del genérico *Text_IO.Integer_IO* para realizar la entrada/salida de números enteros entre 1 y 15, otro para números entre 70 y 120, y otro para números entre -12 y -999. Al realizar la lectura con cada uno de estos paquetes Ada se encarga automáticamente de comprobar que los datos que leemos están en el rango especificado. En general, en esta asignatura utilizaremos *Integer_IO* y *Float_IO* para leer y escribir cualquier número entero (Integer) o cualquier número en coma flotante (Float).

En los siguientes apartados veremos cómo debemos utilizar Text_IO para leer y escribir diferentes tipos de datos.

4.2. Letras

Para escribir y leer una letra utilizaremos directamente el primer nivel de *Text_IO*.

4.2.1. Escribir una letra

Para escribir una única letra utilizaremos **Text_IO.Put**. Por ejemplo:

```
with Text_IO;
procedure Escribir_Letra is
begin
   Text_IO.Put (Á');
end Escribir_Letra;
```

4.2.2. Leer una letra

Para leer una letra debemos declarar una variable de tipo *Character* para que Ada guarde la letra leida. Por ejemplo:

```
with Text_IO;
procedure Leer_Letra is
   Letra : Character;
begin
   Text_IO.Get (Letra);
end Leer_Letra;
```

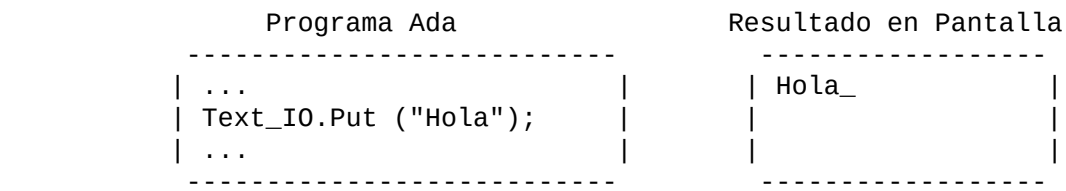
4.3. Texto

Para escribir y leer texto utilizamos también el primer nivel de Text_IO. En los siguientes apartados veremos cómo leer y escribir una frase.

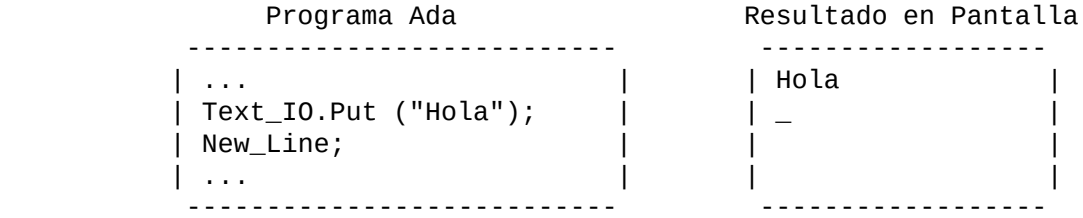
4.3.1. Escribir una frase

Para escribir una frase completa en pantalla tenemos dos posibilidades:

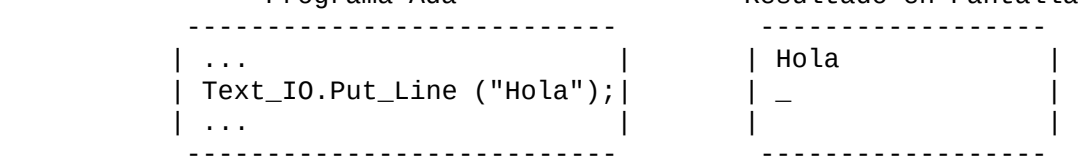
- **Text_IO.Put**(). Escribe en pantalla una frase. Cuando termina de escribir deja el cursor justo a continuación de la frase escrita. Por ejemplo:



La siguiente frase que escribamos en pantalla saldrá justo a continuación de la frase que acabamos de escribir. Si queremos que el cursor de escritura salte al principio de la línea siguiente tendremos que utilizar **Text_IO.New_Line;**. Por ejemplo:



- **Text_IO.Put_Line**(). Escribe en pantalla una frase. Al finalizar deja el cursor al principio de la línea siguiente a la escrita (por tanto es equivalente a llamar a Text_IO.Put y justo a continuación llamar a Text_IO.New_Line). Por ejemplo:



La siguiente frase que escribamos en pantalla saldrá justo en la línea siguiente a la que acabamos de escribir.

4.3.2. Leer una frase

Para realizar cualquier lectura de datos es necesario especificar el nombre de una variable. Para leer texto la variable debe ser de tipo **String**. Tenemos dos formas de leer una frase:

1. **Text_IO.Get**(). Lee del teclado una frase hasta que se llene la variable especificada. Por ejemplo:

```
with Text_IO;
procedure Lee_Frase_1 is
   Linea   : String (1 .. 20);
begin
   Text_IO.Put ("Escribe algo:");
   Text_IO.Get (Linea);
   Text_IO.New_Line;
   Text_IO.Put_Line (Linea);
end Lee_Frase_1;
```

En este ejemplo Ada escribe en pantalla la frase “Escribe algo:” y espera (mediante *Text_IO.Get*) hasta que terminemos de introducir 20 caracteres. Cuando hayamos escrito los 20 caracteres, Ada salta a la línea siguiente (porque hemos puesto *Text_IO.New_Line*) y a continuación escribe en pantalla todo lo que leyó y salta a la línea siguiente (porque le dijimos que hiciese *Put_Line*).

2. **Text_IO.Get_Line**(). Lee del teclado una frase hasta que pulsemos la tecla RETURN o la tecla ENTER. Como ahora el número de caracteres que le damos a Ada no tiene que coincidir con el tamaño de la variable, para utilizar *Text_IO.Get_Line* debemos utilizar dos variables. En la primera Ada nos da los caracteres que lee desde el teclado; en la segunda nos dice cuantos caracteres son. Por ejemplo:

```
with Text_IO;
procedure Lee_Frase_2 is
   Linea    : String (1 .. 20);
   Longitud : Natural;
begin
   Text_IO.Put ("Escribe algo:");
   Text_IO.Get_Line (Linea, Longitud);
   Text_IO.New_Line;
   Text_IO.Put_Line (Linea (1 .. Longitud));
end Lee_Frase_2;
```

Igual que en el ejemplo anterior, Ada comienza escribiendo en pantalla la frase “Escribe algo:”, pero ahora espera (mediante *Text_IO.Get_Line*) hasta que terminemos de introducir los caracteres que queramos (como máximo 20). Cuando hayamos terminado deberemos pulsar RETURN. A continuación Ada salta a la línea siguiente (porque hemos puesto *Text_IO.New_Line* y escribe en pantalla todo lo que leyó. En este caso, a diferencia del ejemplo anterior, debemos decir que de los 20 caracteres de la línea, sólamente queremos escribir en pantalla los que hay hasta la longitud de lo que se leyó. El resto no ha sido utilizado por Ada y, por tanto, no debemos utilizarlo tampoco nosotros.

4.4. Números enteros

Para escribir números enteros en pantalla o para leerlos desde el teclado utilizaremos el segundo nivel de Text_IO: **Text_IO.Integer_IO**. Como este segundo nivel es un paquete genérico (un paquete que aún no está completamente definido y que podemos adaptar a nuestras necesidades), lo primero que debemos hacer es crear un ejemplar del paquete. Para crear un ejemplar del paquete debemos añadir la siguiente frase dentro de nuestro programa:

```
package .... is new Text_IO.Integer_IO (....);
```

Entre los primeros puntos suspensivos ponemos el nombre que queremos dar al ejemplar y entre los segundos puntos suspensivos ponemos el tipo de dato con el que queremos que trabaje este ejemplar.

En el siguiente ejemplo, decimos a Ada que queremos crear tres versiones diferentes (tres ejemplares) de *Text_IO.Integer_IO*.

```
procedure Ejemplo is
   package Int_IO is new Text_IO.Integer_IO (Integer);
   package Nat_IO is new Text_IO.Integer_IO (Natural);
   package Pos_IO is new Text_IO.Integer_IO (Positive);
begin
   ...
end;
```

Hemos dicho a Ada que queremos que la primera se llame *Int_IO*, la segunda *Nat_IO* y la tercera *Pos_IO*. La primera es una versión que sirve para leer y escribir cualquier número entero (Integer es cualquier número entero positivo o negativo). La segunda es una versión más restrictiva; sólo sirve para escribir números naturales (*Natural* son solamente el cero y los números positivos). La tercera es más restrictiva aún. Sólo sirve para escribir números positivos (*Positive* son solamente los números mayores que cero).

4.4.1. Escritura de números enteros

Como hemos visto, para escribir números enteros debemos primero crear un ejemplar de *Text_IO.Integer_IO*. A continuación llamar al procedimiento *Put* utilizando como prefijo el nombre del ejemplar donde está. Por ejemplo:

```
with Text_IO;
procedure Escribir_Entero is
   package Int_IO is new Text_IO.Integer_IO (Integer);
begin
   Int_IO.Put (2 + 2);
end Escribir_Entero;
```

Debemos fijarnos bien en que para escribir un número en pantalla debemos llamar al *Put* del ejemplar que acabamos de crear (Int_IO). Si intentamos llamar al *Put* de *Text_IO.Integer_IO* el compilador nos dirá que hemos cometido un error porque es un paquete genérico, no un ejemplar. (Lógicamente, si intentamos escribir un número utilizando directamente el primer nivel de *Text_IO*, el compilador nos dirá que hemos cometido un error porque no es un carácter). Sólamente podemos escribir enteros utilizando los ejemplares que nosotros creemos en nuestro programa.

4.4.2. Lectura de números enteros

La lectura de números enteros es similar a la escritura. Las únicas diferencias son:

- Que debemos dar a Ada una variable para que almacene el número que lea.
- Que en vez de utilizar Put debemos utilizar Get.

El siguiente ejemplo nos pide que escribamos un número entero y escribe en pantalla el número que hemos escrito.

```
with Text_IO;
procedure Leer_Entero is
   package Int_IO is new Text_IO.Integer_IO (Integer);
   Numero : Integer;
begin
   Text_IO.Put ("Dame un numero entero: ");
   Int_IO.Get (Numero);
   Text_IO.Put ("Escribiste el numero");
   Int_IO.Put (Numero);
end Leer_Entero;
```

4.5. Números reales

Escribir y leer números reales es similar a escribir y leer números enteros. Las únicas diferencias son:

- Se debe utilizar el segundo nivel de *Text_IO* que se llama *Text_IO.Float_IO*.
- Para guardar en memoria el número leido deberemos utilizar una variable que sea de tipo *Float*.

4.5.1. Escritura de números reales

El siguiente ejemplo muestra cómo podemos escribir números reales con Ada.

```
with Text_IO;
procedure Escribir_Real is
   package Float_IO is new Text_IO.Float_IO (Float);
begin
   Float_IO.Put (3.1416E2);
end Escribir_Real;
```

4.5.2. Lectura de números reales

El siguiente ejemplo realiza la lectura de un número en coma flotante y escribe en pantalla el número leido.

```
with Text_IO;
procedure Leer_Real is
   package Float_IO is new Text_IO.Float_IO (Float);
   Numero : Float;
begin
   Text_IO.Put ("Dame un numero real: ");
   Float_IO.Get (Numero);
   Text_IO.Put ("Escribiste ");
   Float_IO.Put (Numero);
end Leer_Real;
```

4.6. Tipos enumerados

Para facilitar la lectura y escritura de tipos enumerados Ada proporciona el paquete genérico *Text_IO.Enumeration_IO*. Este paquete se utiliza exactamente igual que los paquetes *Text_IO.Integer_IO y Text_IO.Float_IO*. Por ejemplo:

```
 1: with Text_IO;
 2: procedure Ejemplo is
 3:    type T_Colores is (Rojo, Amarillo, Verde, Azul);
 4:    package Color_IO is new Text_IO.Enumeration_IO (T_Colores);
 5:
 6:    Color : T_Colores;
 7: begin
 8:    Color := Rojo;
 9:    ...
10:    Color_IO.Get (Color);
11:     ...
12:    Color_IO.Put (Color);
13:  end Ejemplo;
```

En la línea 3 creamos el tipo enumerado *T_Color*, y en la línea 4 el ejemplar del paquete *Text_IO.Enumeration_IO* indicándole a Ada que se va a llamar *Color_IO*. Cada vez que queramos leer desde teclado un enumerado de este tipo utilizamos los procedimientos *Get* (línea 10) y Put (línea 12).

4.7. Resumen

En este capítulo hemos aprendido a decirle a Ada que escriba y lea letras, frases y números. Para leer y escribir caracteres se utiliza directamente el primer nivel de Text_IO; para leer y escribir números debemos utilizar los paquetes genéricos del segundo nivel de *Text_IO*, que son *Text_IO.Integer_IO* y *Text_IO.Float_IO*.

Text_IO permite realizar muchas más cosas que las vistas en este capítulo. En el a’pendice B de este libro están todas las operaciones que puedes hacer con él.

Capítulo 5

Control de flujo

5.1. Introducción

Todos los lenguajes de programación modernos tienen unas estructuras de control básicas que nos permiten expresar complejos algorítmos. Estas estructuras de control pueden clasificarse en tres grandes grupos: secuencial, condicional y repetitiva. En este capítulo conoceremos las estructuras de control de Ada.

5.2. Estructura secuencial

La estructura secuencial es la que hemos estado utilizando en los capítulos anteriores. Ada ejecuta el código en el mismo órden en que está escrito. Por ejemplo:

```
procedure Ejemplo_Secuencia is
    Nombre   : String (1 .. 20);
    Longitud : Natural;
begin
    Text_IO.Put_Line ("Dime tu nombre: ");
    Text_IO.Get_Line (Nombre, Longitud);
    Text_IO.Put_Line ("Hola "
                & Nombre (1 .. Longitud));
end Ejemplo_Secuencia;
```

Ada primero escribe en pantalla la frase *"Dime tu nombre:"*. Después espera a que escribamos nuestro nombre y, justo a continuación, escribe en pantalla la palabra *"Hola"* seguida de nuestro nombre. Vemos que Ada está ejecutando el código siguiendo la secuencia que nosotros hemos puesto en nuestro programa.

5.3. Estructuras condicionales

Aunque la estructura secuencial permite escribir algorítmos sencillos, no es suficiente para escribir programas completos. Las estructuras condicionales permiten que Ada evalúe alguna condición y en función del resultado obtenido ejecute un fragmento de código determinado. Ada tiene dos estructuras condicionales: **if** y **case**.

5.3.1. *if*

La forma más sencilla de esta estructura es la siguiente:

```
if Condicion then
   -----;
   -----;  (bloque de codigo)
   -----;
end if;
```

La condición es una expresión aritmética o lógica cuyo resultado es un valor lógico (*True* o *False*). Si al evaluar la condición el resultado es verdadero, Ada ejecuta el bloque de código; si es falso, se lo salta. Dentro del bloque de código podemos poner cualquiera de las estructuras que veremos en este capítulo (incluyendo más estructuras *if*). Por ejemplo:

```
if Condicion_1 then
   -----;
   if Condicion_2 then
      -----;
      -----;
   end if;
   -----;
end if;
```

Si queremos que nuestro algorítmo trate los dos casos (cuando la condición es cierta y cuando es falsa), debemos decirselo a Ada de la siguiente forma:

```
if Condicion then
   -----;
   -----;  (bloque 1 de codigo)
   -----;
else
```

```
    -----;
    -----;  (bloque 2 de codigo)
    -----;
end if;
```

Si al evaluar la condición el resultado es verdadero, Ada ejecuta el primer bloque de código; si es falso, ejecuta el segundo bloque de código. Igual que en el caso anterior, cada bloque puede contener cualquiera de las estructuras que veremos en este capítulo.

Si lo que queremos hacer es tener una serie de preguntas diferentes y que Ada sólo ejecute uno de los bloques de código, deberemos decirselo de la siguiente forma:

```
if Condicion_1 then
    -----;
    -----;   (bloque 1 de codigo)
    -----;
elsif Condicion_2 then
    -----;
    -----;   (bloque 2 de codigo)
    -----;
.
.
.
elsif Condicion_N then
    -----;
    -----;   (bloque N de codigo)
    -----;
else
    -----;
    -----;   (bloque N+1 de codigo)
    -----;
end if;
```

Ada evalúa la primera condición. Si es cierta, sólo ejecuta el primer bloque de código. Si es falsa, evalúa la segunda condición. Si es cierta sólo ejecuta el segundo bloque de código y así sucesivamente. Si no es cierta ninguna de las condiciones, Ada ejecuta el bloque de código del *else*.

Como hemos visto Ada tiene tres tipos de estructura *if*: la simple, la doble y la múltiple. Obviamente, con la estructura simple se pueden crear todas las demás, y con la doble se puede crear la múltiple (Ada proporciona las tres para facilitar la escritura de nuestros programas). Por ejemplo, los siguientes fragmentos de código son equivalentes:

```
if A >= B and C = A + D then
```

```
if A >= B and C = A + D then
```

```
   -----;
   -----;
   -----;
elsif G = H + P then
   -----;
elsif Q > R or S <= T then
   -----;
   -----;
else
   -----;
end if;
```

<=====>

```
   -----;
   -----;
   -----;
else
   if G = H + P then
      -----;
   else
      if Q > R or S <= T then
         -----;
         -----;
      else
         -----;
      end if;
   end if;
end if;
```

5.3.2. *case*

Si necesitamos utilizar una estructura *if* múltiple en que la condición es siempre la misma (la única diferencia es el valor que debe cumplirse en cada caso) es mejor utilizar la sentencia **case**.

```
case Expresion is
   when ... =>
        -----
        -----  (bloque 1 de codigo)
        -----
   when ... =>
        .
        .
        .
   when ... =>
        -----
        -----  (bloque N de codigo)
        -----
   when others =>
        -----
        -----  (bloque N+1 de codigo)
        -----
end case;
```

La expresión que decide el salto de la sentencia **case** debe ser de tipo discreto (un entero o un tipo enumerado). Dentro del *case* se deben gestionar TODOS los posibles casos. Si no se desea realizar ninguna acción en un determinado caso se debe utilizar la sentencia especial **null**. Para especificar que en el resto de los casos se realice una determinada acción se utiliza **when others** (como vemos, su función es similar al *else* de la estructura *if* múltiple).

Cuando queremos asociar el mismo fragmento de código a varios casos podemos:

- Utilizar la barra vertical y especificar cada caso.
- Especificar un rango de valores diciendo el valor inicial y el valor final, separados por dos puntos consecutivos (por ejemplo, 1 .. 3).

En el siguiente ejemplo suponemos que *Letra* es una variable de tipo *Character*. Si el valor del carácter es '*', se ejecutará el primer bloque; si es # o $, se ejecutará el segundo bloque; si es un dígito se ejecuta el tercer bloque; si es una letra (mayúscula o minúscula) se ejecuta el cuarto bloque. Finalmente, si es cualquier otro carácter, se ejecuta el quinto bloque de código.

```
case Letra is
   when '*' =>
      -----;
      -----;
   when '#' | '$' =>
      -----;
      -----;
      -----;
   when '0' .. '9' =>
      -----;
   when Á'..'Z' | á'..'z' =>
      -----;
      -----;
      -----;
   when others =>
      -----;
      -----;
end case;
```

Para entender bien este ejemplo veamos a continuación su equivalente utilizando el *if* múltiple.

```
case Letra is
   when '*' =>
      -----;
      -----;
   when '#' | '$' =>
      -----;
      -----;
      -----;
   when '0' .. '9' =>
```

```
if Letra = '*' then
   -----;
   -----;
elsif Letra = '#' or Letra = '$' then
   -----;
   -----;
   -----;
elsif Letra in '0' .. '9' then
```

```
      -----;
   when Á'..'Z' | á'..'z' =>

      -----;
      -----;
      -----;
   when others =>
      -----;
      -----;
end case;
```

```
      -----;
elsif Letra in Á'..'Z'
        or Letra in á'..'z' then
      -----;
      -----;
      -----;
  else
      -----;
      -----;
  end if;
```

En el último *elsif* (ejemplo de la derecha) hemos utilizado el operador **in**. Este operador returna *True* cuando el valor de una variable está en el rango que se especifica justo a continuación de la palabra **in**.

5.4. Estructuras repetitivas

Además de las estructuras condicionales, necesitamos estructuras que nos permitan decir a Ada que queremos repetir un determinado fragmento de código. Ada tiene tres estructuras repetitivas: **loop**, **while** y **for**.

5.4.1. *loop*

La estructura **loop** nos permite repetir un bloque de código infinitas veces. Se expresa así:

```
loop
      -----;
      -----;
end loop;
```

Si queremos que, cuando se cumpla una determinada condición, Ada salga del bucle, debemos utilizar una frase **exit** indicando la condición de salida. También podemos hacerlo con un bucle *if* y una frase *exit*. Por ejemplo, los dos fragmentos de código siguientes son equivalentes:

```
loop                                loop
   -----;                              -----;
   exit when Total > 300;              if Total > 300 then
   -----;                                 exit;
end loop;                              end if;
                                       -----;
                                    end loop;
```

En caso de que tengamos varios bucles anidados (uno dentro de otro), al decirle a Ada *exit* le estamos indicando que debe salir del bucle donde está el *exit* (el bucle más interno). Por ejemplo:

```
loop
     -----;
     loop
        -----
        -----
        exit when Condicion_1;
     end loop;
     -----;
     exit when Condicion_2;
     -----;
end loop;
```

El primer *exit* le está diciendo a Ada que cuando se cumpla su condición salga del bucle más pequeño. El segundo *exit* le está diciendo que, cuando se cumpla la segunda condición salga del bucle más grande.

La ejecución de este programa sería así. Ada comienza a ejecutar el bucle grande. Tras ejecutar la primera línea del bucle grande, comienza a ejecutar el bucle pequeño, y no sale de él hasta que se cumpla *Condición_1*. Cuando Ada sale del bucle pequeño, continua ejecutando el resto de las líneas del bucle grande hasta llegar al *exit* del bucle grande. Si su condición es cierta, Ada sale del bucle grande (no ejecuta el resto de líneas); si es falsa, ejecuta el resto del líneas y vuelve a ejecutar el bucle grande desde el principio.

Como ejemplo de uso, veamos cómo podemos hacer el menú de un programa.

```
with Text_IO;
procedure Ejemplo_Menu is
   Opcion : Character;
begin
    loop

       Text_IO.Put_Line ("Menu");
       Text_IO.Put_Line ("1. Entrar datos");
```

```
            Text_IO.Put_Line ("2. Mostrar datos");
            Text_IO.Put_Line ("3. Calcular");

            loop
               Text_IO.Get (Opcion);
               exit when Opcion in '1'..'3';
               Text_IO.Put_Line ("Opcion incorrecta");
            end loop;

            case Opcion is
               when '1' =>
                     ...
                     ...
               when '2' =>
                     ...
               when '3' =>
                     ...
                     ...
               when others =>
                     null;
            end case;

         end loop;
      end Ejemplo_Menu;
```

5.4.2. *while*

Si necesitamos un bucle en el que, justo antes de comenzar cada vuelta, Ada compruebe si debe ejecutar el código del bucle, en vez de *loop* debemos utilizar *while*. Obviamente el bucle *while* es equivalente a un bucle *loop* que tenga una frase *exit* justo al principio. La principal ventaja del bucle *while* es que facilita la lectura del programa. Por ejemplo, los dos fragmentos siguientes son equivalentes:

```
while Condicion loop
   -----;
   -----;
end loop;
```

```
loop
   exit when Condicion;
   -----;
   -----:
end loop;
```

Fijate bien que este bucle puede no ejecutarse nunca (porque si al comenzar la condición es falsa, Ada se salta todo el bucle).

5.4.3. *for*

Si tenemos que repetir el bucle un número exacto de veces (que conocemos antes de empezar el bucle), debemos utilizar el bucle **for**. Al bucle *for* debemos

asociarle una variable. Esta variable la crea automáticamente cuando comienza a ejecutar el bucle y la utiliza para llevar la cuenta de cuantas veces ha hecho el bucle. Podemos decirle a Ada que cuente hacia adelante o hacia detrás.

```
for Contador in 1 .. 10 loop
    -----;
    -----;
end loop;
```

```
for Indice in reverse 25 .. 50 loop
   -----;
   -----;
end loop;
```

Ada crea la variable *Contador* y la utiliza para llevar la cuenta del número de veces que repite el código que hemos puesto dentro del bucle. En el primer ejemplo, como le hemos dicho que cuente hacia adelante, llevará la cuenta como 1, 2, 3, . . . , 10. En el segundo ejemplo Ada crea la variable *Indice*, pero esta vez cuenta al revés (50, 49, . . . , 25).

Ada sabe que la variable que lleva la cuenta del bucle es una variable especial que podemos leer, pero nunca modificar. Sólo puede modificarla Ada incrementándola o decrementándola al repetir el bucle *for*.

El indice del bucle puede ser cualquier tipo discreto (enteros o enumerados), pero no de tipo flotante. En caso de ambiguedad, debemos ayudar a Ada a saber cual es el tipo de dato que debe utilizar para realizar la cuenta. Por ejemplo, si tenemos el tipo:

```
type T_Colores is (Rojo, Amarillo, Naranja, Verde, Azul);
type T_Semaforo is (Rojo, Amarillo, Verde);
```

. . . podemos escribir:

```
for Indice in Rojo .. Azul loop
    ...
end loop;
```

. . . ya que Ada sabe que el único tipo enumerado que tiene el valor *Azul* es el tipo *T_Colores* y, por tanto, deduce que la variable Color debe ser de tipo T_Colores. Sin embargo, si escribimos:

```
for Indice in Rojo .. Verde loop
    ...
end loop;
```

...Ada no sabe si *Color* debe ser de tipo *T_Colores* o de tipo *T_Semaforo*. Para resolver esta ambiguedad debemos especificar el tipo de una de las siguientes formas:

```
--  METODO 1
------------
    for Color in T_Colores'(Rojo) .. Verde loop
        ...
    end loop;

--  METODO 2
------------
    for Color in T_Colores range Rojo .. Verde loop
        ...
    end loop;
```

5.4.4. Elección del bucle más apropiado

Una regla general para elegir el bucle más apropiado es intentar utilizar primero el bucle *for*. Si no conseguimos escribir nuestro algorítmo con él, intentamos utilizar el bucle *while*. Finalmente, si no podemos utilizar ninguno de los anteriores, utilizaremos el bucle más general: el *loop*.

```
for   --->   while  -->   loop
```

5.5. Resumen

En este capítulo hemos conocido todas las estructuras de control de Ada. La estructura secuencial nos permite especificar el órden en que debe Ada ejecutar el algorítmo; la estructura condicional nos permite decirle que debe elegir sólo una de las opciones que le damos, y la estructura repetitiva nos permite decirle que debe repetir un trozo del algorítmo.

Con lo que hemos visto hasta ahora ya podemos escribir cualquier programa. Sin embargo, si escribimos todo nuestro programa dentro de un único procedimiento llega un momento en que es muy difícil detectar fallos en nuestros algorítmos. Para mejorar esta situación todos los lenguajes de programación modernos permiten descomponer nuestros programas en subprogramas (procedimientos y funciones). Esto lo veremos en el siguiente capítulo.

Capítulo 6

Subprogramas

En los capítulos anteriores hemos escrito nuestros ejemplos utilizando siempre un único procedimiento (el procedimiento que contiene el programa principal). Sin embargo, Ada nos permite descomponer nuestros programas en subprogramas (procedimientos y funciones).

En este capítulo veremos qué diferencia hay entre ambos tipos de subprogramas, cómo elegir buenos nombres para ellos, cómo declarar variables dentro de los subprogramas, las reglas que fija Ada para establecer la comunicación entre los subprogramas y cómo realizar la traza de nuestros programas Ada.

6.1. Procedimientos y funciones

La principal diferencia que encontramos entre un procedimiento y una función es la forma en que se utilizan. Para llamar a un procedimiento solamente damos su nombre. Sin embargo, como toda función devuelve siempre un valor, debemos almacenar el resultado que devuelve en una variable o utilizar este valor en alguna expresión aritmética o lógica. Por ejemplo:

```
 1: procedure Ejemplo_1 is
 2:
 3:    procedure Menu is
 4:    begin
 5:       ...
 6:       ...
 7:       return;
 8:    end Menu;
 9:
10:
11:  begin
12:     Menu;
13:  end Ejemplo_1;
```

```
 1: procedure Ejemplo_2 is
 2:
 3:    function Maximo
 4:       return Positive is
 4:    begin
 5:       ...
 6:       return Positive'Last;
 7:    end Maximo;
 8:
 9:    Valor : Positive;
10: begin
11:    Valor := Maximo;
12: end Ejemplo_2;
```

El ejemplo de la izquierda funciona de la siguiente forma: Ada comienza la ejecución del código del programa principal (que en este caso se llama Ejemplo_1). Entre el *begin* y el *end* sólamente hay una llamada a *Menu*. Esto significa que Ada debe dejar pendiente la ejecución del programa principal y ejecutar el procedimiento *Menu*. Ada ejecuta el procedimiento *Menu* hasta que encuentre la palabra **return** o llegue al **end Menu**. En ambos casos da por finalizado el procedimiento y continúa ejecutando el procedimiento que tiene el programa principal. En caso de que *Menu* llame a otros procedimientos, Ada ejecuta esos procedimientos y, cada vez que finalice la ejecución de uno, volverá al anterior para completar su ejecución.

Como vemos, trabajar con subprogramas tiene enormes ventajas. Podemos dar un nombre a un fragmento de código y llamarlo cada vez que lo necesitemos (evitando así repetir el mismo código dentro del programa). De esta forma, si decidimos cambiar alguna de las acciones de este código sólo necesitaremos hacer el cambio una vez, mientras que si hemos repetido el código muchas veces tendremos que buscar todos los trozos repetidos y cambiarlos todos exactamente de la misma forma (desde que se nos olvide corregir alguno de los trozos estaremos introduciendo errores en nuestro programa que serán cada vez más difíciles de detectar).

La ejecución de la función del segundo ejemplo es similar. La principal diferencia es que en su declaración debemos decir el tipo de dato que devuelve (línea 3) y que en el punto de la llamada tendremos que utilizar una asignación (porque el resultado de la función hay que guardarlo en alguna variable). Tiene como ventaja que podemos utilizar directamente el resultado de la función para evaluar cualquier expresión. Por ejemplo:

```
Valor := Maximo - 456;
```

Para devolver el valor de la función, el cuerpo de la función debe tener obligatoriamente la palabra **return** seguida del valor que se quiere devolver (en este caso se devuelve *Positive'Last*, línea 6).

6.1.1. Elección del nombre del subprograma

La correcta elección del nombre de un procedimiento o de una función sigue debe seguir las siguientes reglas básicas.

- El nombre de una función debe ser un sustantivo (ya que debe nombrar el valor que devuelve la función). Por ejemplo:

  ```
  function Suma ...
  ```

  Esto facilita la lectura del código en los puntos de llamada. Por ejemplo:

  ```
  Total := Total + Suma (Numero, Valor);
  ```

- El nombre de un procedimiento debe ser un verbo. Por ejemplo:

  ```
  procedure Sumar ...
  ```

  De nuevo, este criterio facilita la lectura del código en los puntos de llamada, ya que los procedimientos no se llaman en medio de cualquier expresión. Por ejemplo:

  ```
  Sumar (Numero, Valor);
  ```

La elección entre procedimiento o función depende de si queremos llamar al subprograma dentro de cualquier expresión aritmética o lógica (en este caso sería una función) o si, por el contrario, queremos que la llamada se realice de forma exclusiva (un procedimiento). También influyen otros aspectos del lenguaje como el *modo* de los parámetros, que veremos en el apartado 6.2.

6.1.2. Variables locales

Los subprogramas pueden contener declaraciones locales entre las palabras **is** y **begin**. Por ejemplo:

```
with Text_IO;
procedure Ejemplo is

   procedure Presentacion is
      Num_Veces : Positive := 4;
      Letra     : Character := '*';
   begin
      for I in 1 .. Num_Veces loop
         Text_IO.Put (Letra);
      end loop;
      Text_IO.New_Line;
   end Presentacion;

   Calculo : Natural;
   Numero  : Float;
begin
   Presentacion;
   Calculo := ... ;
   . . .;
end Ejemplo;
```

En este ejemplo el procedimiento principal (*Ejemplo*) tiene dos variables: *Total*, de tipo *Natural* y *Número*, de tipo *Float*. El procedimiento *Presentación* tiene también dos variables: *Num_Veces*, de tipo *Positive* y *Letra*, de tipo *Character*.

Este ejemplo funciona de la siguiente forma. Cuando se comienza a ejecutar el programa *Hola*, Ada crea las variables del procedimiento principal (*Total* y *Número*). A continuación Ada comienza a ejecutar el cuerpo del procedimiento principal, que llama al procedimiento *Presentación*. Antes de comenzar a ejecutar *Presentación* Ada crea sus variables (*Num_Veces* y *Letra*) y las inicia con los valores que hemos fijado en el programa (4 y *). A continuación ejecuta todas las acciones de *Presentación* y, cuando finaliza su ejecución, destruye *Num_Veces* y *Letra* (porque, ya no hacen falta). En caso de que posteriormente se vuelva a llamar al procedimiento *Hola*, Ada volverá a crear las variables y a asignar a cada una su valor inicial.

En resumen, cada vez que se llama a un procedimiento (o a una función) Ada realiza los siguientes pasos:

1. Crea las variables locales del procedimiento (o función).
2. Ejecuta todas las sentencias que hay entre las palabras *begin* y *end* (hasta llegar al *end* o ejecutar una órden *return*).
3. Destruye todas las variables locales del procedimiento (o función).

Por tanto, debemos siempre recordar que las variables son siempre locales al procedimiento que las crea. Si se llama de nuevo al procedimiento, se vuelven a

crear. No mantienen el valor que tenían al finalizar la llamada anterior.

6.1.3. Ambitos y visibilidad

Ada establece unas normas rígidas que regulan el acceso a todas las variables de nuestro programa. Estas normas se pueden simplificar en dos:

1. Todo subprograma tiene visibles sus variables y las variables de todos los subprogramas que lo engloban.
2. Si en este recorrido hay varias variables con el mismo nombre (identificador), tendremos acceso a la primera variable visible (avanzando desde el subprograma hacia los subprogramas que lo engloban).

Por ejemplo, si tenemos un programa con la siguiente estructura:

```
procedure P1 is
   V1 : Integer;

   function P2
      return ...
   is
      V2 : Integer;

      procedure P3 is ...
          V1 : Float;
      begin
         ...
      end P3;

      function P4
         return ...
      is
         V4 : Integer;
      begin
         ...
      end P4;

   begin
      ...
   end P2;

begin
   ...
end P1;
```

El procedimiento P3 puede utilizar su variable (V1, de tipo *Float*) y la variable de P2 (V2, de tipo *Integer*). Sin embargo, no podrá acceder nunca a la variable V1

del procedimiento principal P1 (ya que su propia variable, al llamarse igual, le impide ver la de P1). La función P4 puede ver las variables V4, V2 y la V1 del programa principal (no puede ver la V1 de P3 porque P4 no está dentro de P3).

A la hora de realizar las llamadas a los subprogramas, Ada mantiene unas reglas similares:

- Todo subprograma puede llamar a cualquiera de los subprogramas que lo engloban (incluido él mismo). Por ejemplo, P3 puede llamar a P3, a P2 y a P1.
- También puede llamar a todos los subprogramas de su mismo nivel que hayan sido declarados antes que él. Por ejemplo, P4 puede llamar a P3, pero P3 no puede llamar a P4.
- Obviamente, también puede llamar a los que estén declarados dentro de él. Pero si éstos tienen a su vez otros subprogramas dentro de ellos, a éstos no pueden llamarlos. Por ejemplo, P1 puede llamar a P2 (porque está declarado justo dentro de su ámbito), pero no puede llamar a P3 (porque está declarado dentro de P2).

La segunda regla puede parecer demasiado estricta. Mis programas pueden necesitar que los procedimientos del mismo nivel se llamen entre si. Si queremos que ambos puedan llamarse mutuamente tenemos que añadir una declaración de P4 antes de P3. Esto se hace añadiendo una copia de la cabecera del procedimiento justo antes de P3. Por ejemplo:

```
procedure P1 is
   V1 : Integer;

   function P2 is
      V2 : Integer;

      procedure P4;         --  Declaracion nueva

      procedure P3 is ...
          V1 : Float;
      begin
         ...
         P4;        --  Ahora P3 puede llamar a P4
         ...
      end P3;

      procedure P4 is ...
         V4 : Integer;
      begin
         ...
```

```
        ...
    end P4;

begin
    ...
end P2;

begin
    ...
end P1;
```

De esta forma, como Ada lee el programa de arriba a abajo, cuando comience a leer la función P2 sabrá que más adelante va a encontrarse un procedimiento que se llama P4. Por tanto, cuando vea que P3 llama a P4, lo dará por correcto (porque sabe que está en el programa). Para disponer siempre de esta posibilidad cuando compilemos con la opción de comprobación de estilos deberemos declarar todos los procedimientos antes de poner su código.

6.2. Parámetros

Para facilitar la comunicación entre los subprogramas Ada nos permite declarar parámetros.

```
procedure Ejemplo is

   procedure Sumar (Numero_1  : . . . ;
                    Numero_2  : . . . ;
                    Resultado : . . .  ) is
      --  Los parametros especificados en la cabecera
      --  de los subprogramas se llaman 'FORMALES'
   begin
      . . .
   end Sumar;
   . . .
begin
   Sumar ( , , , );
   --  Los parametros que se pasan en la llamada se
   --  llaman parametros 'REALES'.
end Ejemplo;
```

Cuando queramos utilizar parámetros deberemos añadir unos paréntesis después del nombre del subprograma. En el siguiente apartado veremos qué debemos poner en los puntos suspensivos.

6.2.1. Modo de los parámetros

Los parámetros se utilizan para transmitir información entre los procedimientos. Por tanto, debemos decir si el parámetro contiene información que entra en el subprograma (es un parámetro de entrada), que sale del procedimiento (es un parámetro de salida) o información que entra, se modifica y sale del procedimiento (es un parámetro de entrada y salida). Para especificar el modo de cada parámetro se utilizan las palabras **in**, **out** e **in out**. Justo a continuación del modo debemos especificar el tipo del parámetro (que puede ser cualquiera de los tipos básicos de Ada, o cualquier tipo que hayamos creado nosotros). Por ejemplo, si completamos el código de nuestro ejemplo anterior, obtenemos:

```
procedure Ejemplo is

   procedure Sumar
      (Numero_1  : in Integer;
       Numero_2  : in Integer;
       Resultado : out Integer)
   is
   begin
      Resultado := Numero_1 + Numero_2;
   end Sumar;

   Dato_1    : Integer := 12;
   Dato_2    : Integer := 10;
   Resultado : Integer;
begin
   Sumar (Dato_1, Dato_2, Resultado);
   --  Aqui resultado contiene el valor 22.
end Ejemplo;
```

Cuando se realiza la llamada al procedimiento el contenido de los parámetro de entrada se inicia con el valor de los datos especificados en la llamada (en este caso los valores de *Dato_1* y *Dato_2* se copian sobre los parámetros *Numero_1* y *Numero_2*). El código de este procedimiento suma el contenido del primer parámetro de entrada al contenido del segundo parámetro de entrada ($Numero\_1 + Numero\_2$) y almacena el resultado en el parámetro de salida (*Resultado*). Cuando finaliza la llamada, el contenido del parámetro de salida (*Resultado*) se copia sobre la variable utilizada en la llamada. En este caso, como la variable del procedimiento principal se llama también resultado, estaríamos copiando el contenido del parámetro *Resultado* del procedimiento *Sumar* sobre la variable *Resultado* del procedimiento principal.

Como hemos visto en ejemplos anteriores, en el caso de las funciones su especificación contiene también el tipo del resultado. Por ejemplo, si el procedimiento anterior lo escribimos como una función el código sería:

```
procedure Ejemplo is

   function Suma
      (Numero_1  : in Integer;
       Numero_2  : in Integer)
      return Integer is
   begin
      return Numero_1 + Numero_2;
   end Sumar;

   Dato_1    : Integer := 12;
   Dato_2    : Integer := 10;
   Resultado : Integer;
begin
   Resultado := Suma (Dato_1, Dato_2);
   --  Aqui resultado contiene el valor 22.
end Ejemplo;
```

Como las funciones sólo pueden devolver un único resultado, Ada no permite que tengan parámetros en modo **out** o **in out**.

Si no decimos el modo de algún parámetro, Ada presupone siempre que es de modo **in**. Por ejemplo, las siguientes son equivalentes:

```
procedure Sumar
   (Numero_1  : in Integer;
    Numero_2  : in Integer;
    Resultado : out Integer);
```

```
procedure Sumar
   (Numero_1  : Integer;
    Numero_2  : Integer;
    Resultado : out Integer);
```

Cuando varios parámetros son del mismo tipo, podemos ponerlos en una lista separados por comas. Por ejemplo:

```
procedure Sumar
   (Numero_1, Numero_2 : in Integer;
    Resultado          : out Integer);
```

Resumiendo, la especificación de un procedimiento o función proporciona el nombre, modo y tipo de cada uno de los parámetros. Si se omite el modo del parámetro, se presupone que es **in**). Las funciones sólo pueden tener parámetros en modo **in**.

6.2.2. Parámetros por omisión

Ada permite asociar un valor por omisión a cada uno de los parámetros de entrada (**in**). Estos valores se utilizan como valor inicial de los parámetros en caso de que en la llamada no se pase ningún valor. Por ejemplo:

```
procedure Escribir
   (Dato  : in Integer;
    Ancho : in Integer := 6;
    Base  : in Integer := 10);
```

Esto significa que cuando se realice una llamada a *Escribir* los parámetros *Ancho* y *Base* son opcionales. Si se omite *Ancho* se presupone el valor 6 y si se omite *Base* se presupone el valor 10.

Suponiendo que J es un entero, todas las siguientes llamadas son válidas:

```
Escribir (J);
Escribir (J, Ancho => 4);
Escribir (J, Base  => 16);
Escribir (J, 4, 16);
Escribir (J, Ancho => 4, Base => 16);
Escribir (Dato => J, Base => 16, Ancho => 4);
```

En la primera llamada solamente se pasa el valor del primer parámetro (*Dato*). Por tanto, al realizar la llamada Ada se encarga automáticamente de iniciar el parámetro *Ancho* con el valor 6 y el parámetro *Base* con el valor 10. En la segunda se está pasando el valor del primer y el último parámetro. El resto de los ejemplos son fáciles de entender.

6.2.3. Parámetros de la línea de órdenes

El único procedimiento que no puede tener nunca parámetros es el procedimiento del programa principal (ya que es un procedimiento especial que da el nombre al programa completo y lo llamamos desde el sistema operativo). Sin embargo, nosotros estamos acostumbrados a llamar al editor de textos dandole el nombre del archivo que queremos editar. ¿Cómo se hace esto desde Ada?

Este caso especial se resuelve mediante otra biblioteca de Ada. La biblioteca *Ada.Command_Line*. Esta biblioteca tiene dos funciones: *Argument_Count*, que nos dice el número de parámetros que nos han pasado y *Argument*, que nos da cada uno de los parámetros. Por ejemplo, si queremos escribir en pantalla todos los parámetros que nos pasen podemos hacerlo así:

```
with Text_IO;
with Ada.Command_Line;
procedure Parametros is
begin
```

```
    for I in 1 .. Ada.Command_Line.Argument_Count loop
       Text_IO.Put_Line( Ada.Command_Line.Argument (I));
    end loop;
end Parametros;
```

6.3. Llamada a un subprograma

Ada permite dos formas de pasar los parámetros en la llamada a los subprogramas.

1. **Notación posicional**: Separando los parámetros mediante comas:

```
Sumar (Valor_1, Valor_2, Resultado);
```

2. **Notación nombrada**: Nombrando los parámetros.

```
Sumar (Numero_1  => Valor_1,
       Numero_2  => Valor_2,
       Resultado => Resultado);
```

Aunque la notación nombrada es más larga, es muchísimo más clara que la posicional. Es bueno que te acostumbres a utilizarla para facilitar la lectura y revisión de tus programas.

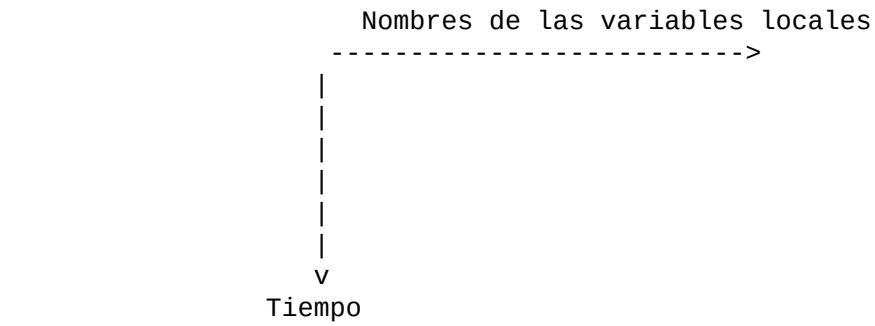
6.3.1. Recursividad

Como hemos visto en el apartado anterior, cada vez que se llama a un procedimiento Ada crea las variables del procedimiento. Esto ocurre incluso cuando el procedimiento se llama a sí mismo. Cuando un procedimiento se llama a sí mismo, decimos que el procedimiento es recursivo. Por ejemplo, la siguiente función resuelve el calculo del factorial de forma recursiva.

```
function Factorial (N : Natural) return Natural is
begin
    if N = 0 then
        return 1;
    else
        return N * Factorial (N - 1);
    end if;
end Factorial;
```

6.4. Traza

Llamamos *traza* de un programa a la representación gráfica del estado del programa en función del tiempo. La principal utilidad de la traza es comprobar que el código hace exactamente lo que queremos haga. Para realizar la traza de un programa debemos crear una tabla de la siguiente forma:



Los nombres de las variables deben respetar el orden de las declaraciones de nuestro programa. Por supuesto, el tiempo representado no es un tiempo real. En cada paso de tiempo solamente debemos poner el nuevo valor de las variables que ha modificado nuestro programa. Veamoslo a continuación con más detalle.

6.4.1. Traza sin subprogramas

Como primer ejemplo, veamos cómo se genera la traza de un programa que sólo tiene el procedimiento principal.

```
1: procedure Ejemplo_Traza is
2:    Numero_1 : Integer  := -22;
3:    Numero_2 : Positive := 45;
4:    Letra    : Character;
5: begin
6:    Letra    := Á';
7:    Numero_1 := 2 * Numero_1;
8:    Numero_2 := Character'Pos (Letra);
9: end Ejemplo_Traza;
```

La traza sería:

```
          Ejemplo_Traza

 Numero_1    Numero_2       Letra
(Integer)   (Positive)   (Character)
 ----------------------------------
|  -22          45                 |
|----------------------------------|
|                             'A'  |
|----------------------------------|
|  -44                             |
|----------------------------------|
|               65                 |
 ----------------------------------
```

En la primera linea (centrado con respecto a las líneas siguientes) ponemos el nombre del procedimiento. En la línea siguiente, respetando el orden en que hemos puesto las variables en el programa, ponemos los nombres de las variables de izquierda a derecha (*Numero_1, Numero_2* y *Letra*). En la siguiente línea ponemos (entre paréntesis) el tipo de cada variable. Estamos preparados para comenzar la traza.

Pasamos una línea horizontal y ponemos el valor inicial de todas las variables (los valores iniciales que están en las líneas 2 y 3 del programa). En caso de que no tengan valor inicial (como la línea 4) no ponemos nada (dejamos su casilla vacía). En las siguientes líneas sólamente colocamos los nuevos valores que va guardando Ada en cada una de las variables (los que va guardando en las asignaciones de las líneas 6, 7 y 8 de este programa). Si una línea Ada no modifica ninguna variable, nos la saltamos (porque no es representativa para la traza del programa).

6.4.2. Traza con subprogramas

Generalmente todos nuestros programas estarán compuestos de subprogramas. Para hacer la traza deberemos hacer lo siguiente:

1. Cuando se realiza la llamada debemos añadir (siempre por la derecha de la traza) todos los parámetros del subprograma y poner el valor inicial de los parámetros de entrada. El valor inicial será el valor pasado en la llamada o su valor por omisión (en caso de que exista).

2. En la línea siguiente debemos añadir todas las variables locales del subprograma.

3. A continuación, igual que en la traza sin subprogramas, en las líneas siguientes colocamos los nuevos valores que va guardando Ada en cada una de las variables.

4. Cuando se termina la ejecución del subprograma debemos copiar el valor de los parámetros de salida en las variables utilizadas en la llamada.

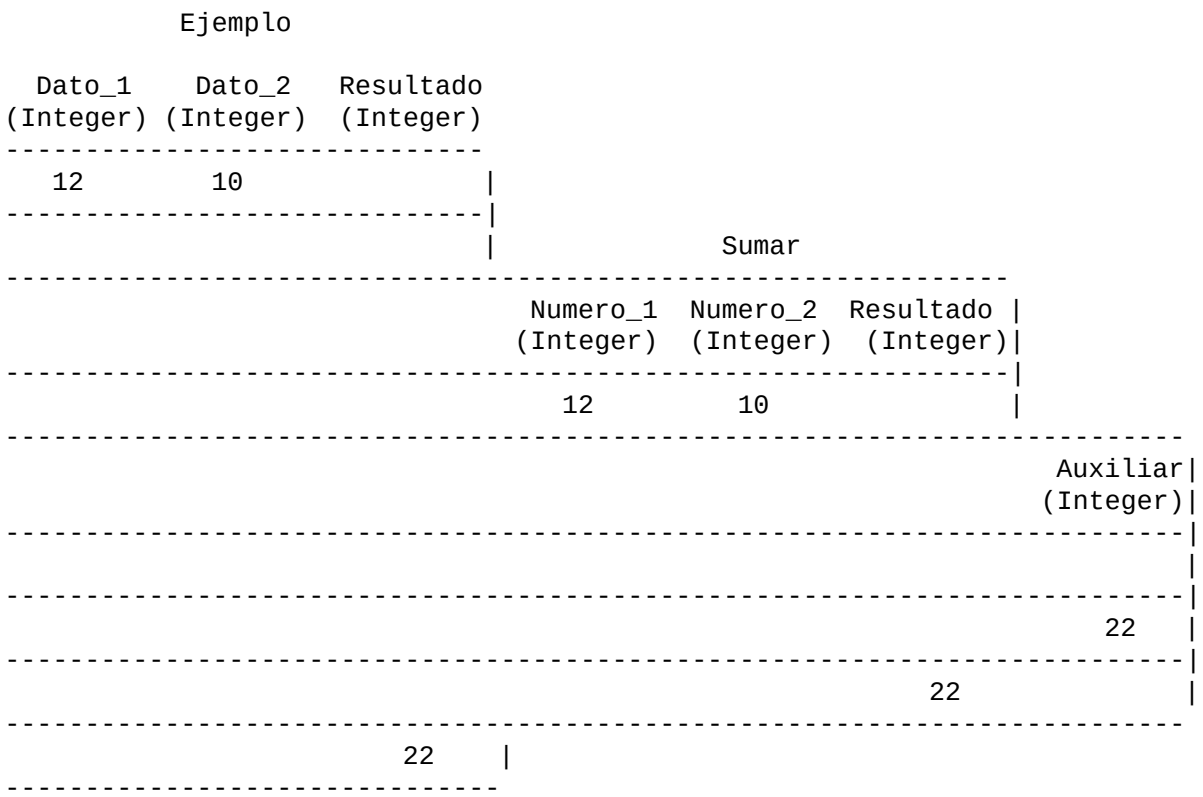
Si el subprograma llama a otro subprograma, realizamos de nuevo la misma secuencia de pasos.

Como ejemplo vamos a ver a continuación la traza del siguiente ejemplo:

```
procedure Ejemplo is

   procedure Sumar
      (Numero_1  : in Integer;
       Numero_2  : in Integer;
       Resultado : out Integer)
   is
      Auxiliar  : Integer;
   begin
      Auxiliar  := Numero_1 + Numero_2;
      Resultado := Auxiliar;
   end Sumar;

   Dato_1    : Integer := 12;
   Dato_2    : Integer := 10;
   Resultado : Integer;
begin
   Sumar (Dato_1, Dato_2, Resultado);
   --  Aqui resultado contiene el valor 22.
end Ejemplo;
```



Debemos tener mucho cuidado en representar correctamente el margen derecho de nuestra traza, ya que cuando cerramos este margen (hacia la izquierda) estamos representando que Ada elimina las variables del subprograma.

6.5. Resumen

En este capítulo hemos visto los subprogramas de Ada (procedimientos y funciones): las diferencias que hay entre los procedimientos y las funciones, la forma de pasar y recibir información mediante los parámetros, y la forma de realizar la traza de nuestros programas. En el siguiente capítulo profundizaremos un poco más en las estructuras de datos. Veremos cómo podemos crear estructuras compuestas a partir de los tipos de datos básicos de Ada.

Capítulo 7

Tipos compuestos

En este capítulo completaremos nuestro conocimiento de tipos de datos para este primer curso. Todos los lenguajes de programación modernos permiten crear tipos compuestos (tipos de datos compuestos por varios tipos sencillos). Existen dos tipos: heterogéneos y homogéneos. En Ada los heterogéneos se llaman registros y los homogéneos se llaman formaciones (en inglés *arrays*). Veamos cada uno de ellos.

7.1. Registro

Un registro es una estructura de datos heterogénea que nos permite agrupar declaraciones y referirnos a ellas como un todo. Por ejemplo:

```
type T_Mes is (Enero, Febrero, Marzo, Abril, Mayo, Junio, Julio,
               Agosto, Septiembre, Octubre, Noviembre, Diciembre);
subtype T_Dia is Positive range 1 .. 31;
type T_Fecha is
      record
          Dia : T_Dia;
          Mes : T_Mes;
          Año : Integer;
      end record;

Aniversario: T_Fecha;
```

En este ejemplo, la variable *Aniversario* tiene tres partes llamadas campos: *Aniversario.Dia*, *Aniversario.Mes* y *Aniversario.Año*. Los campos de un registro

pueden ser de cualquier tipo, incluyendo otros registros, y pueden utilizarse por separado. Por ejemplo:

```
Aniversario.Dia := 18;
Aniversario.Mes := Febrero;
Aniversario.Año := 2000;
```

También podemos iniciar todos los elementos del registro de una sola vez. Para ello Ada nos permite utilizar las dos posibilidades que teníamos en las llamadas a los subprogramas. La notación posicional y la notación nombrada. En el siguiente ejemplo asignamos los tres campos de una sola vez utilizando la notación posicional:

```
Aniversario := (18, Febrero, 2000);
```

El objeto a la izquierda de la asignación es de tipo *T_Fecha* y el objeto a la derecha es un agregado con notación posicional (debido a que las tres partes están en el mismo orden que los tres campos de la definición del registro).

La notación nombrada es mucho más clara y nos permite especificar las partes del agregado en cualquier orden. Por ejemplo:

```
Aniversario := (Mes => Febrero, Dia => 18, Año => 2000);
```

Esta notación facilita la lectura de los programas. También podemos combinar ambos tipos de notación dentro de un mismo agregado. Para ello comenzamos siempre con la notación posicional y desde que especificamos un elemento mediante notación nombrada ya tenemos que especificar el resto mediante notación nombrada. Por ejemplo:

```
Aniversario := (18, Año => 2000, Mes => Febrero);
```

7.1.1. Valores por omisión

Igual que los parámetros de los subprogramas, los registros también pueden tener valores por omisión. Por ejemplo:

```
type T_Mes is (Enero, Febrero, Marzo, Abril, Mayo, Junio, Julio,
               Agosto, Septiembre, Octubre, Noviembre, Diciembre);
subtype T_Dia is Positive range 1 .. 31;
```

```
type T_Fecha is
      record
          Dia : T_Dia   := 1;
          Mes : T_Mes   := Enero;
          Año : Integer := 2000;
      end record;

Aniversario: T_Fecha;
```

En este caso, Ada inicializa el campo *Aniversario.Año* al valor *2000* desde el instante en que crea la variable. Sin embargo no podemos escribir:

```
Aniversario := (18, Febrero);   -- Ilegal
```

...porque tenemos que especificar todos los campos.

7.1.2. Registro con variantes

La definición de un registro puede tener, simultáneamente, varios formatos posibles que llamamos variantes. Por ejemplo, si vamos a crear una ficha médica para los pacientes de un médico, deberemos diferenciar la información que almacenamos cuando el paciente es un hombre o una mujer. En caso de que también haya campos comunes (por ejemplo la edad), Ada nos obliga a colocarlos al principio (las variantes van siempre al final del registro). Por ejemplo:

```
type T_Sexo is (Hombre, Mujer);
type Persona(Sexo : T_Sexo) is
      record
        Edad : Natural := 0;
        case Sexo is
            when Varon =>
                 Problemas_De_Prostata: Boolean := False;
            when Mujer =>
                 Problemas_De_Ovarios : Boolean := False;
                 Numero_De_Embarazos  : Natural := 0;
        end case;
      end record;
Ficha_Medica: T_Persona;
```

En este ejemplo hemos dicho a Ada que todas las personas tienen un campo común con la *Edad*. Además hemos dicho a Ada que si es un hombre, el registro tiene un campo de tipo lógico para almacenar si ha tenido alguna vez problemas de próstata, mientras que si es una mujer, queremos que el registro tenga dos campos para recordar si ha tenido problemas en los ovarios y el número de embarazos.

Las variables de este tipo de registro se declaran y se inicializan igual que el resto de los registros. Por ejemplo:

```
Antonio : Persona(Sexo => Hombre) := (Sexo => Hombre,
                                      Edad => 21,
                                      Problemas_De_Prostata => False);
Isabel  : Persona(Sexo => Mujer)  := (Sexo => Mujer,
                                      Edad => 18,
                                      Problemas_De_Ovarios => False,
                                      Numero_De_Embarazos  => 0);
```

Si intentamos acceder al campo *Antonio.Numero_De_Embarazos* o al campo *Isabel.Problemas_De_Prostata* Ada eleva la excepción **Constraint_Error**.

7.2. Formación (*array*)

Una formación es un conjunto de datos homogéneos. Para identificar cada uno de los elementos de la formación utilizamos un índice. Llamamos formaciones unidimensionales a las que tienen un único índice y multidimensionales a las que tienen varios índices. Comencemos viendo las unidimensionales.

7.2.1. Formación unidimensional

Para declarar una formación unidimensional debemos especificar:

1. El tipo y el rango del índice de la formación.
2. El tipo de los elementos de la formación.

El índice puede ser de cualquier tipo discreto (entero o enumerado) y los elementos de la formación pueden ser de cualquier tipo (incluyendo registros y otras formaciones). Por ejemplo:

```
type T_Tabla_1 is array (1 .. 10) of Integer;
   --                    ^^^^^^^     ^^^^^^^
   --  Tabla con 10 elementos de tipo entero.

type T_Tabla_2 is array (-10 .. 10) of Float;
```

```
   --                    ^^^^^^^^^     ^^^^^
   --  Tabla con 21 elementos (se incluye el cero) de tipo Float.

type T_Tabla_3 is array (Positive range 2001 .. 2015) of Float;
   --                    ^^^^^^^^ ^^^^^
   --  Tabla con 15 elementos de tipo Float. Hemos avisado a Ada
   --  que el indice va a ser de tipo Positive.

type T_Colores is (Rojo, Amarillo, Naranja, Verde, Azul);
type T_Tabla_4 is array (T_Colores range Rojo .. Verde) of Natural;
--                    ^^^^^^^^^
--  Tabla con 4 elementos de tipo Natural. El indice es un color.

type T_Tabla_5 is array (1 .. 4) of T_Tabla_4;
--  Tabla con 4 elementos de tipo T_Tabla_4. Para acceder a
--  cada uno de los elementos tendremos que indicar los dos
--  indices. Por ejemplo:
--     T : T_Tabla_5;
--     Para acceder al elemento Rojo del primer elemento
--     debo decir T (1) (Rojo) := 12;
```

7.2.2. Formación multidimensional

Las formaciones no están limitadas a tener un único índice; pueden tener tantos índices (dimensiones) como necesitemos (y todos los índices pueden ser de tipos discretos diferentes). Por ejemplo:

```
type T_Tabla is array
          (Integer   range -10 .. -1,
           T_Colores range Rojo .. Azul,
           T_Mes     range Febrero .. Junio) of T_Fecha;
```

7.2.3. Inicialización

Obviamente las formaciones pueden inicializarse mediante múltiples asignaciones individuales, o mediante bucles. Sin embargo, Ada también permite que las formaciones puedan inicializarse y asignarse mediante agregados (pudiendo utilizarse tango la notación posicional como la nombrada). Por ejemplo:

```
type T_Vector5 is array(1 .. 5) of Float;

A : constant T_Vector5 := (2.0, 4.0, 8.0, 16.0, 32.0);
```

```
B : constant T_Vector5 := (1 => 2.0,  2 => 4.0,
                           3 => 8.0,  4 => 16.0,
                           5 => 32.0);
```

El agregado debe especificar todos los elementos de la formación. Si hay muchos elementos que deben tener el mismo valor inicial, podemos decirle a Ada los que son diferentes y utilizar la palabra reservada *others* para dar el valor del resto. Por ejemplo:

```
type T_Vector500 is array(1 .. 500) of Float;

V1 : T_Vector500 := (others => 0.0);
W  : T_Vector500 := T_Vector500'(10 => 1.3,
                                 15 => -30.7,
                                 others => 0.0);
```

También podemos especificar varios elementos del array mediante la barra vertical. Por ejemplo:

```
type T_Tabla is array (1 .. 15) of Boolean;
Mi_Tabla : T_Tabla;
...
Mi_Tabla := T_Tabla'(1 | 3 | 5 | 7 | 9 => True,  others => False);
```

También podemos especificar rangos. Por ejemplo, para inicializar a 1 los elementos 1, 6, 7, 8, 9 10 y el resto a cero:

```
type T_Tabla is array(1 .. 15) of Natural;

Tabla_1 : T_Tabla :=T_Tabla'(1 | 6 .. 10 => 1,  others => 0);
Tabla_2 : T_Tabla := (1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0);
```

En el caso de las formaciones multidimensionales podemos utilizar agregados anidados para su inicialización. Por ejemplo:

```
Matriz : array(0 .. 9, 1 .. 5) of Float := (others => (others => 1.0));.

type T_Cuadrado is array (1 .. 10, 1 .. 10) of Integer;
...
Cuadrado : T_Cuadrado := T_Cuadrado'(4 => (5 => 1, others => 0),
                                     others  => (others => 0)
                                    );
```

7.2.4. Copia

Para copiar los elementos de una formación en otra podemos:

- Copiar cada uno de los elementos mediante un bucle.
- Asignar directamente el contenido de una formación a la otra (siempre que las dos tengan exactamente el mismo tamaño y sus elementos sean del mismo tipo).

Por ejemplo, si tenemos las siguientes declaraciones:

```
type T_Vector100 is array (1 .. 100) of Float;
type T_Vector300 is array (1 .. 300) of Float;

Vector_1, Vector_2, Vector_3 : T_Vector100;
Vector_4, Vector_5           : T_Vector300;
```

Vector_1, *Vector_2 y Vector_3* son del tipo *T_Vector100* y por tanto podemos escribir *Vector_1 := Vector_3* para copiar toda la formación mediante una única asignación. De forma similar podemos escribir *Vector_5 := Vector_4*, pero no *Vector_4 := Vector_1*.

Sin embargo, no es necesario que los índices coincidan (sólo deben coindicir el número de elementos y sus tipos). Por ejemplo:

```
declare
    type T_Vector is array(Integer range <>) of Float;
    Vector_1 : T_Vector(1 .. 5);
    Vector_2 : T_Vector(2 .. 6) := (others => 0.0);

    Frase_1 : String(1 .. 5);
    Frase_2 : String(2 .. 6) := (others => ' ');
 begin
    Vector_1 := Vector_2;
    Frase_1  := Frase_2;
    Frase_1  := "Hola Ada";       -- Constraint_Error
    Frase_1 (1 .. 8) := "Hola Ada";
    Frase_1  := "Hola";           -- Constraint_Error
    Vector_1 := (1.0, 2.0, 3.0);  -- Constraint_Error
  end;
```

7.2.5. Porción de una formación unidimensional

Una porción de un formación es una parte de la formación y se expresa mediante un rango[1]. Por ejemplo:

```
Tabla  : array (1 .. 10) of Integer :=
                          (1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
Frase  : String (1 .. 8) := "Hola Ada";

Tabla (1 .. 3);   -- 1,2,3
Tabla (6 .. 8);   -- 6,7,8
Frase (6 .. 8);  -- "Ada"
Frase (3 .. 4);  -- "la"
```

También podemos escribir:

```
S(1 .. 10):= S(2 .. 11);
A(1 .. 3) := A(4 .. 6);
```

. . . porque las longitudes coinciden a ambos lados del operador de asignación.

Si el valor inicial es mayor que el valor final tenemos una porción nula (y su longitud es cero). Por ejemplo:

```
 S (2 .. 1); -- porcion nula
```

7.2.6. Operadores para formaciones unidimensionales

El operador **&** concatena dos formaciones siempre que sean del mismo tipo (incluyendo dos *String*). También permite concatenar un elemento con una formación. Por ejemplo:

```
procedure Ejemplo_Concatenacion is
   Letra_1, Letra_2   : Character := '*';
   Frase_2            : String (1 .. 2);
   Frase_3            : String (1 .. 3) := (others => ' ');
   Frase_5            : String (1 .. 5);

   type T_Vector is array (Integer range <>) of Float;
   Numero_1, Numero_2 : Float := 1.2;
   Vector_2           : T_Vector(1 .. 2);
```

[1] Algunos lenguajes utilizan el término *substring* para referirse a una porción de una *String*, pero en Ada podemos tomar una porción de cualquier tipo de formación, no sólo de strings.

```
   Vector_3           : T_Vector(1 .. 3) := (others => 0.0);
   Vector_5           : T_Vector(1 .. 5);
begin
   Frase_1 := Letra_1 & Letra_2;
   Frase_5 := Frase_2 & Frase_3;
   Frase_3 := Letra_1 & Frase_2;
   Frase_3 := Frase_2 & Letra_1;;

   Vector_2 := Numero_1 & Numero_2;
   Vector_5 := Vector_2 & Vector_3;
   Vector_3 := Numero_1 & Vector_2;
   Vector_3 := Vector_2 & Numero_1;
end Ejemplo_Concatenacion;
```

Si tenemos formaciones unidimiensionales de *Boolean* podemos utilizar los operadores *and*, *or*, *xor* y *not* definidos para el tipo *Boolean*. El resultado final es que Ada realiza la operación elemento a elemento. De esta forma podemos simular conjuntos. Por ejemplo:

```
declare
   type T_Tabla is array (1 .. 3) of Boolean;
   Tabla_1   : T_Tabla := (True, False, False);
   Tabla_2   : T_Tabla := (True, True,  False);
   Resultado : T_Tabla;
begin
   Resultado := Tabla_1 or Tabla_2;  --  (True, True, False)
   Resultado := Tabla_1 and Tabla_2; --  (True, False, False)
   Resultado := Tabla_1 xor Tabla_2; --  (False, True, False)
end;
```

Igual que ocurre con los registros, puedes utilizar los operadores $=$ y $/=$ para comparar dos formaciones del mismo tipo. Los registros son iguales si cada uno de sus campos es igual; las formaciones son iguales si cada uno de sus elementos es igual. Las formaciones de diferente longitud son siempre diferentes.

También podemos utilizar los operadores $<$, $>$. En este caso Ada comienza a comparar los elementos uno a uno. Mientras ambos elementos sean iguales Ada continua avanzando. Desde que no sean iguales utiliza ese resultado como el valor definitivo de la comparación. Por ejemplo:

```
Frase_1 : String(1 .. 10) := "Para todos";
Frase_2 : String(1 .. 7)  := "Para mi";

--  Frase_1 > Frase_2 porque 't' > 'm' en la
--  definicion Ada del tipo Character
```

Ada comienza a comparar las dos frases. Compara la primera letra de cada frase. Como son iguales, avanza a la siguiente. Así hasta que encuentre alguna

diferente (en este caso la 't' y la 'm'). Si todas las letras son iguales (por ejemplo, "Juan" y "Juan Francisco"), Ada considera que es mayor la más grande (ya que es el órden que se utiliza para ordenar alfabéticamente).

7.2.7. Atributos *First, Last* y *Length*

En las formaciones podemos utilizar los atributos **First** y **Last** para saber cual es su primer y último índice válido. Por ejemplo:

```
type T_Vector is array(30 .. 33) of Integer;
Mi_Vector : T_Vector;
...
Mi_Vector( T_Vector'First ) := 1000;    -- Mi_Vector(30) := 1000;
Mi_Vector( Mi_Vector'First) := 1000;    -- Mi_Vector(30) := 1000;

Mi_Vector( T_Vector'Last )  := 3000;    -- Mi_Vector(33) := 3000;
Mi_Vector( Mi_Vector'Last)  := 3000;    -- Mi_Vector(33) := 3000;
```

El atributo **Range** es una abreviación del rango *First .. Last*. Se suele utilizar para recorrer todos los elementos de una formación. Por ejemplo:

```
for I in Mi_Vector'range loop
    ...
end loop;
```

En este ejemplo *T_Vector'Range* y *Mi_Vector'Range* representan el rango *30 .. 33*.

También podemos utilizar el atributo **Length** para conocer el número de elementos de una formación. En el ejemplo anterior *T_Vector'Length* y *Mi_Vector'Length* valen 4.

Cuando tenemos formaciones multidimensionales debemos decir entre paréntesisa cual de los índices nos referimos. Por ejemplo:

```
type T_Tabla is array (Positive range 1 .. 10,
                       Integer range -34 .. 6) of Character;

--  T_Tabla'First (1) vale 1
--  T_Tabla'Last  (1) vale 10

--  T_Tabla'First (2) vale -34
--  T_Tabla'Last  (2) vale 6
```

7.2.8. Formaciones irrestringidas

Otra forma de declarar formaciones consiste en no especificar el rango del índice (mediante el símbolo <>). Por ejemplo:

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

D, E, F : T_Vector (1 .. 100);
G, H    : T_Vector (1 .. 300);
```

Como vemos, en la frase que declara el tipo *T_Vector* no hemos especificado el rango del índice (no hemos dicho aún el rango del índice). Por tanto, cuando declaremos variables de este tipo deberemos siempre especificar el rango.

Si uno de los indices de la formación está limitado, el resto debe estar también limitado. Por ejemplo, no podemos declarar:

```
type T_Rectangulo is
     array(1 .. 5,
           Integer range <>) of Float;  -- ilegal
```

7.2.9. Cadenas de caracteres (Strings)

A diferencia de otros lenguajes de programación, las cadenas de Ada no son un tipo de dato especial. Ada tiene una declaración implícita[2] del tipo **String** que es de la siguiente forma:

```
type String is array(Positive range <>) of Character;
```

Esto significa que podemos declarar:

```
Cadena_1 : String(1 .. 5);
Cadena_2 : constant String := "Hola Ada";
```

...pero no podemos declarar:

[2]Una declaración implícita es una declaración que realiza automáticamente Ada antes de comenzar a analizar cualquier programa. Por tanto, esta declaración está siempre presente en todos los programas escritos con Ada.

```
Cadena_3 : String;     -- Ilegal
```

...porque es un tipo irrestringido (y debemos fijar su rango al declarar la variable).

Es importante que nos demos cuenta de que, al no ser un tipo especial, todos los mecanismos que proporciona Ada para tratar formaciones podemos utilizarlos con las variables de tipo *string*. Por ejemplo:

```
declare
   Saludo : String(1 .. 4);
begin
   Saludo := ('H', ó', 'l', á');
   Saludo := "Hola";
end;
```

Ambas asignaciones son equivalentes; la primera notación es más formal (de acuerdo con lo visto hasta ahora), pero Ada permite abreviar utilizando la segunda notación (entre comillas).

Si queremos que una *String* contenga las comillas, debemos duplicarlas. Por ejemplo:

```
Text_IO.Put_Line("Fichero: ""datos.dat"" no encontrado.");

Al ejecutarla obtendremos en pantalla el mensaje:

             Fichero "datos.dat" no encontrado.
```

Los programadores novatos a veces confunden un *Character* con una *String* de longitud 1. Si escribimos:

```
Frase : String(1 .. 10);
```

...*Frase (1)* es un *Character* y *Frase (1 .. 1)* es una *String* de longitud 1 (de la misma manera que `'X'` es una constante de tipo *Character* y `"X"` es una constante de tipo *String* de longitud 1). Por tanto, podemos escribir:

```
Frase (1) := 'X';
Frase (1 .. 1) := "X";
```

...pero NO podemos mezclar tipos y escribir:

```
Frase (1) := "X";      --  Los tipos no coinciden.
Frase (1 .. 1):= 'X';  --  Los tipos no coinciden.
```

Afortunadamente la biblioteca *Text_IO* tiene procedimientos *Put* y *Get* para el tipo *Character* y procedimientos *Put* y *Get* para el tipo String. Esto nos permite escribir en pantalla una *String* de las siguientes formas:

```
Text_IO.Put ( Frase (1 .. 1) );
Text_IO.Put ( Frase (1) );
```

Sin embargo, los procedimientos *Put_Line* y *Get_Line* sólo permiten constantes y variables de tipo *String*.

7.3. Resumen

Hemos visto cómo podemos crear estructuras de datos complejas a partir de los tipos de datos básicos de Ada. Distinguimos dos tipos de estructuras de datos: heterogéneas y homogéneas. Las heterogéneas se construyen mediante los registros. Las homogéneas se construyen mediante las formaciones. Hemos aprendido a crear el tipo de dato asociado, a declarar sus variables y a utilizarlas.

En el siguiente capítulo conoceremos brevemente las excepciones de Ada, que son el mecanismo que utiliza Ada para notificarnos errores durante la ejecución de nuestros programas.

Capítulo 8

Excepciones

Cuando ocurre un error durante la ejecución del programa decimos que ha ocurrido una excepción. Ada nos permite capturar las excepciones y ejecutar un bloque especial de código denominado **manejador de excepciones**.

Al igual que en los tipos de datos, Ada proporciona algunas excepciones predefinidas y nos permite definir nuevas excepciones.

8.1. Excepciones predefinidas

Las cinco excepciones predefinidas de Ada son:

1. **Constraint_Error**: Esta excepción se eleva cuando:

   - Hay un índice fuera de rango (por ejemplo, intentar acceder al elemento 31 de una formación de 30 elementos).

     ```
     declare
       type T_Tabla is array (1 .. 30) of Natural;
       Tabla : T_Tabla;
     begin
       Tabla (31) := 0;  -- Contraint_Error
       Tabla (0)  := 0;  -- Constraint_Error
       ...
     end;
     ```

   - Se asigna a una variable un valor fuera del rango permitido. Por ejemplo, se intenta asignar una valor negativo a un positivo.

```
declare
  Numero : Positive;
begin
  Numero := 0;   -- Contraint_Error
  Numero := -3;  -- Constraint_Error
  ...
end;
```

- Se utiliza mal un atributo.

```
declare
   Numero : Integer;
begin
   Numero := Integer'Value (''12X3''); -- Constraint_Error
   ...
end declare;
```

  En este caso hemos pedido a Ada que nos de el valor numérico de un número que no existe. Por eso Ada nos eleva la excepción.

- Se asigna una formación a otra con diferente tamaño.

```
Saludo : String(1 .. 5) := "Hola";  --  Constraint_Error
```

  Saludo tiene un tamaño de 5 letras y le hemos asignado una frase que sólo tiene 4 letras.

2. **Numeric_Error**: Se eleva cuando se produce un desbordamiento en una operación aritmética o una división por cero.

3. **Program_Error**: Se eleva cuando se ha producido algún error de programación que no pudo ser detectado en tiempo de compilación. Por ejemplo, cuando una función termina sin ejecutar ninguna sentencia *return*.

4. **Storage_Error**: Se eleva cuando se hay algún problema de falta de memoria. Por ejemplo, cuando se ha creado un procedimiento recursivo que no finaliza nunca o cuando creamos una tabla demasiado grande para la memoria de nuestro ordenador.

5. **Tasking_Error**: Veremos en detalle esta excepción en cursos posteriores.

8.2. Declaración de excepciones

A parte de las excepciones predefinidas, también puedes declarar tus propias excepciones. Para hacerlo, debes utilizar el tipo de dato **exception**. Por ejemplo:

```
Dato_Erroneo : exception;
```

Cuando detectes el error y quieras que Ada lo considere como un error debes **elevar la excepción**. Para esto debes utilizar la frase **raise**. Por ejemplo:

```
if Dato > 500 then
   raise Dato_Erroneo;
end if;
```

Cuando elevas una excepción Ada detiene la ejecución del programa y muestra en pantalla el error (el nombre de la excepción).

También podemos capturar las excepciones y hacer que cuando se detecte un error Ada ejecute algún fragmento de código alternativo (por ejemplo, volver a pedir el número). Esto se hace mediante un *manejador de excepciones*.

8.3. Manejador de excepciones

Para capturar las excepciones utilizamos los *manejadores de excepción*. Para ello utilizamos la palabra reservada **exception** en medio de cualquier bloque *begin-end* y una estructura muy parecida a la de la sentencia *case*. Veamos como ejemplo un procedimiento que contiene un manejador para las excepciones predefinidas **Constraint_Error** y **Numeric_Error** y para la excepción *Error* definida por nosotros.

```
with Text_IO;
procedure Ejemplo is
   package Int_IO is new Text_IO.Integer_IO (Integer);
   Numero: Integer;
   Error : exception;
begin
   loop
      Text_IO.Put ("Dame un numero positivo: ");
      Int_IO.Get (Numero);
      if Numero <= 0 then
         raise Error;
      end if;
      Text_IO.Put("El cuadrado es ... ");
      Int_IO.Put (Numero*Numero);
          -- Al hacer la multiplicacion Ada eleva la excepcion
          -- Constraint_Error or Numeric_Error si el resultado
          -- es demasiado grande.
   end loop;
exception
```

```
    when Constraint_Error | Numeric_Error =>
       Text_IO.Put (" ... Demasiado grande.");
    when Error =>
       Text_IO.Put ("Te dije POSITIVO !!");
end Ejemplo;
```

Cuando se eleva una excepción Ada detiene la ejecución normal del programa y salta a ejecutar el manejador de excepciones. Si no lo hay, finaliza la ejecución del procedimiento, retorna al procedimiento que haya realizado la llamada (si lo hubo) y vuelve a buscar el manejador de excepciones. Este proceso se repite hasta que encuentre un manejador de excepciones. Si no encuentra ninguno, finaliza la ejecución de todo el programa y escribe en pantalla un error diciendonos cual fue la excepción que se elevó.

Al igual que ocurre en la sentencia *case*, podemos utilizar la barra vertical para expresar que varias excepciones tienen asociadas el mismo manejador. También podemos utilizar la expresión *when others* para expresar que en el caso de que sea cualquier otra excepción ejecute un determinado manejador.

Dentro del manejador de excepciones podemos reelevar la excepción. Para ello simplemente utilizamos la sentencia *raise* sin especificar ninguna excepción. Por ejemplo:

```
when others =>
       Text_IO.Put_Line("Error desconocido.");
       raise;
```

8.4. Resumen

En este breve capítulo hemos conocido qué son las excepciones de Ada, cuales son las excepciones predefinidas de Ada, cómo declarar nuevas excepciones, cómo se elevan y cómo se capturan.

En el siguiente capítulo veremos, también brevemente, cómo son las bibliotecas de Ada.

Capítulo 9

Paquetes

Los paquetes nos permiten agrupar declaraciones que estén lógicamente relacionadas (Por ejemplo, el paquete *Text_IO* contiene todos los procedimientos y funciones necesarios para escribir y leer datos en pantalla y en ficheros).

Todos los paquetes constan de dos partes: una parte de especificación, que sólo contiene declaraciones de constantes, tipos de datos, variables y subprogramas, y una parte de cuerpo, que contiene el código de todos los subprogramas del paquete.

9.1. Especificación

La especificación más simple de un paquete se indica mediante la siguiente frase.

```
package  . . . . .  is
    . . .
end . . . . . ;
```

Después de la palabra **package** se pone el nombre del paquete. Este nombre debe repetirse al final (justo después de la palabra **end**). Entre las palabras **is** y **end** se ponen todas las declaraciones de constantes, tipos y subprogramas del paquete que deben ver todos los que utilicen este paquete. Estas declaraciones son la interfaz del paquete. Por ejemplo:

```
package Calculo is
   Valor_Maximo : constant Positive := 12345;

   procedure Duplicar(Numero   : in Integer;
                      Respuesta : out Integer);
   function Doble (Numero : in Integer) return Integer;
end Calculo;
```

En esta especificación hemos puesto la declaración de una constante, un procedimiento y una función. El código de estos subprogramas se pone en el cuerpo del paquete.

En GNAT los ficheros que contienen la especificación de los paquetes deben terminar siempre con la extensión **.ads**. Por ejemplo, la especificación de este paquete debe ponerse en un fichero con el nombre *calculo.ads*.

9.2. Cuerpo

El cuerpo del paquete debe contener el código de todos los procedimientos y funciones declarados en la especificación del paquete. Además puede opcionalmente tener más declaraciones de constantes, tipos, variables y subprogramas y un código de inicialización (que se coloca justo al final del paquete dentro de un bloque *begin-end*). Por ejemplo:

```
package body Calculo is
   Tabla: array(1..100) of Integer;

   procedure Duplicar(Numero   : in Integer;
                      Respuesta: out Integer) is
   begin
      Respuesta := 2 * Numero;
   end Duplicar;

   function Doble(Numero : in Integer)
                     return Integer is
   begin
      return Tabla(Numero);
   end Doble;

begin
   -- Codigo de inicializacion del paquete
   for i in 1..10 loop
       Tabla(I):=2*I;
   end loop;
end Calculo;
```

Las declaraciones hechas dentro del cuerpo del paquete (por ejemplo *Tabla*) sólamente están disponibles dentro del cuerpo del paquete (mientras que las declaraciones hechas en la especificación del paquete pueden utilizarse desde todos los programas que lo utilicen). Esta característica nos permite modificar la implementación de una aplicación sin modificar su interfaz. Por ejemplo, podría ocurrirsenos otro algorítmo para estos subprogramas que no utilizase ninguna tabla. Con sólo cambiar el cuerpo del paquete, conseguimos cambiar la biblioteca sin que afecte a sus usuarios.

En GNAT los cuerpos de los paquetes se colocan en ficheros con extensión **.adb**. Por tanto, este ejemplo debe estar en el fichero *calculo.adb*.

9.3. Uso de paquetes (*with*)

Como ya sabemos, el programa que quiere utilizar este paquete debe indicarlo en su cabecera mediante una sentencia *with*. Por ejemplo:

```
with Calculo;
procedure Ejemplo is
   I, J : INTEGER := 10;
begin
   Calculo.Duplicar(Numero => I, Respuesta => J);
   J := Calculo.Doble(I);
end Ejemplo;
```

9.4. Resumen

En este capítulo hemos visto una breve descripción de qué son los paquetes y las partes que lo componen. En el siguiente capítulo aprenderemos a crear, modificar y borrar ficheros desde nuestros programas Ada.

Capítulo 10

Ficheros

10.1. Excepciones asociadas a ficheros

Ada eleva excepciones cuando intentamos hacer alguna operación incorrecta con ficheros:

- Eleva la excepción **Status_Error** si intentamos leer o escribir en un fichero que está cerrado.
- Eleva **Mode_Error** cuando intentamos leer de un fichero que fue abierto o creado para escribir en él (un fichero abierto o creado con el modo *Out_File*), o cuando intentamos escribir en un fichero que fue abierto para leer de él (fue abierto con el modo *In_File*).
- Eleva *Name_Error* cuando intentamos abrir un fichero que no existe, o cuando intentamos crear un fichero con un nombre que no es válido en el sistema operativo en el que estamos trabajando.
- Eleva *End_Error* cuando intentamos continuar leyendo información después de haber llegado al final del fichero.

10.2. Ficheros de texto

Hasta ahora hemos utilizado *Text_IO* para leer y escribir texto utilizando el teclado y la pantalla, pero también puede utilizarse para leer y escribir en disco ficheros de texto.

La especificación del paquete *Text_IO* proporciona un tipo de dato denominado **File_Type** con el que podemos declarar variables de tipo fichero. Por cada fichero que utilicemos, deberemos crear una variable de este tipo. Por ejemplo:

```
with Text_IO;
procedure Ejemplo_Fichero is
   Fichero : Text_IO.File_Type;
begin
   ...
end Ejemplo_Fichero;
```

Para trabajar con ficheros lo primero que debemos hacer es asociar a cada variable su fichero. Esto se hace con los procedimientos **Create** y **Open**. La diferencia fundamental entre los dos es que debemos utilizar *Create* cuando el fichero no existe aún (y queremos que Ada lo cree) y utilizaremos *Open* cuando el fichero ya existe en nuestro disco. Por ejemplo:

```
with Text_IO;
procedure Ejemplo_Fichero is
   Fichero : Text_IO.File_Type;
begin
   Text_IO.Create (Fichero, Text_IO.Out_File, "saludo.txt");
   ...
end Ejemplo_Fichero;
```

Si le decimos a Ada que cree un fichero y el fichero ya existía, Ada borra el fichero y crea uno nuevo con el mismo nombre.

Como vemos en este ejemplo, al asociar el nombre del fichero tenemos que decir si la información va a salir desde el programa Ada hacia el fichero (**out**) o si la información va a entrar en el programa Ada (**in**) porque el programa va a leerla.

Una vez asociado el nombre del fichero, ya podemos utilizar la variable para escribir o leer información del fichero. En *Text_IO* existen dos versiones de todos los procedimientos de escritura y lectura que conocemos (**New_Line, Put, Get, Get_Line**, etc.). Una versión accede directamente al teclado y la pantalla (es la versión que hemos utilizado hasta ahora); la otra versión es para utilizarla con ficheros. Esta segunda versión siempre recibe como primer parámetro una variable de tipo *File_Type* (no el nombre del fichero), para saber en qué fichero debe leer o escribir.

```
procedure Get (File : in File_Type; Item : out Character);
procedure Get (Item : out Character);
```

```
procedure Put (File : in File_Type; Item : in Character);
procedure Put (Item : in Character);

procedure Get (File : in File_Type; Item : out String);
procedure Get (Item : out String);

procedure Put (File : in File_Type; Item : in String);
procedure Put (Item : in String);

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

procedure Get_Line
  (Item : out String;
   Last : out Natural);

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

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

Como ejemplo de uso, el siguiente programa Ada escribe varias líneas en un fichero de texto. Como el fichero es de texto, podemos leerlo con cualquier editor de textos.

```
with Text_IO;
procedure Ejemplo_Fichero is
   Fichero : Text_IO.File_Type;
begin
   Text_IO.Create (Fichero, Text_IO.Out_File, "saludo.txt");
   Text_IO.Put_Line ("Este es un ejemplo de escritura");
   Text_IO.Put_Line ("en un fichero desde un programa Ada.");
   Text_IO.Close (Fichero);
end Ejemplo_Fichero;
```

Al terminar de utilizar un fichero debo siempre cerrar el fichero (mediante **Close**).

Si quiero leer todo el contenido de un fichero de texto, justo antes de leer cada línea, tengo que preguntarle a Ada si he llegado ya al final del fichero (ya que Ada es quien único sabe si ya lo ha leido todo o no). Para hacer esto utilizaremos la función **End_Of_File**. Como ejemplo, el siguiente programa Ada lee todas las líneas de un fichero de texto y las escribe en pantalla.

```
with Text_IO;
```

```
procedure Ver_Fichero is
   Fichero  : Text_IO.File_Type;
   Linea    : String (1 .. 80);
   Longitud : Natural;
begin
   Text_IO.Open (Fichero, Text_IO.Out_File, "saludo.txt");

   while not Text_IO.End_Of_File (Fichero) loop
      Text_IO.Get_Line (Fichero, Linea, Longitud);
      Text_IO.Put_Line (Linea (1 .. Longitud));
   end loop;

   Text_IO.Close (Fichero);
end Ver_Fichero;
```

En este ejemplo vemos que podemos combinar fácilmente escritura y lectura en ficheros con lectura y escritura en pantalla. El *Get_Line* está leyendo del fichero, mientras que el *Put_Line* está escribiendo en pantalla.

10.2.1. Números y enumerados

Los paquetes genéricos *Integer_IO*, *Float_IO* y *Enumeration_IO* también tienen dos versiones de los procedimientos *Put* y *Get*. Una para acceder a teclado y pantalla (que es la que hemos utilizado hasta ahora); la otra para leer o escribir en ficheros. Igual que en los procedimientos de *Text_IO*, para utilizar la versión de ficheros debemos pasar como primer parámetro una variable de tipo *Text_IO.File_Type*. Por ejemplo:

```
with Text_IO;
procedure Ejemplo_Fichero is
   package Int_IO is new Text_IO.Integer_IO (Integer);

   Fichero_Numeros : Text_IO.File_Type;
begin
   Text_IO.Create (Fichero_Numeros, Text_IO.Out_File, "numeros.txt");

   for Contador in 1 .. 10 loop
       Int_IO.Put (Fichero_Numeros, Contador);
   end loop;

   Text_IO.Close (Fichero_Numeros);
end Ejemplo_Fichero;
```

10.2.2. Ejemplo: Copia de un fichero de texto

Como ejemplo de uso de *Text_IO*, veamos un programa que nos permite copiar un fichero de texto.

```
with Text_IO;
procedure Copiar_Fichero is
   Nombre          : String (1..20);
   Longitud_Nombre : Natural;
   Origen, Destino : Text_IO.File_Type;
   Linea           : String (1 .. 120);
   Longitud        : Natural;
begin
   Text_IO.Put("Nombre del fichero a copiar: ");
   Text_IO.Get_Line(Nombre, Longitud_Nombre);

   Text_IO.Open( File => Origen,
                 Mode => Text_IO.In_File,
                 Name => Nombre(1..Longitud_Nombre) );

   Text_IO.Put("Nombre del nuevo fichero: ");
   Text_IO.Get_Line(Nombre, Longitud_Nombre);

   Text_IO.Create (File => Destino,
                   Mode => Text_IO.Out_File,
                   Name => Nombre(1 .. Longitud_Nombre) );

   while not Text_IO.End_Of_File(Origen) loop
         Text_IO.Get_Line (Origen, Linea, Longitud);
         Text_IO.Put_Line (Destino, Linea(1..Longitud));
   end loop;

   Text_IO.Close (Origen);
   Text_IO.Close (Destino);
end Copiar_Fichero;
```

10.3. Ficheros de acceso secuencial y de acceso directo

Text_IO sirve para crear, leer y escribir ficheros de texto. Sin embargo, Ada también proporciona los paquetes genéricos **Sequential_IO** y **Direct_IO** que pueden crear, leer y escribir ficheros con cualquier formato. Este tipo de ficheros no puede verse fácilmente en pantalla (ya que en pantalla lo que vemos es sólamente texto). Cada uno tiene su propio tipo *File_Type* para crear ficheros. Por ejemplo:

```
with Sequential_IO;
procedure Ejemplo is
   package Seq_IO is new Sequential_IO (Integer);
   Fichero : Seq_IO.File_Type;
begin
   ...
end Ejemplo;
```

Igual que *Text_IO*, ambos paquetes tienen procedimientos para crear, abrir, cerrar ficheros. También tienen procedimientos para leer y escribir información en el fichero, pero se llaman *Read* y *Write* (en vez de *Get* y *Put*). Por ejemplo:

```
with Sequential_IO;
procedure Ejemplo is

   package Seq_IO is new Sequential_IO (Integer);

   Fichero : Seq_IO.File_Type;

begin

   Seq_IO.Create
      (File => Fichero,
       Mode => Text_IO.Out_File,
       Name => "Numeros.dat");

   for Contador in 1 .. 100 loop
      Seq_IO.Write (Fichero, Contador);
   end loop;

   Seq_IO.Close (Fichero);

end Ejemplo;
```

Sequential_IO simpre escribe o lee la información en posiciones consecutivas del fichero, mientras que *Direct_IO* puede escribir o leer en cualquier posición del fichero. Para ello, los procedimientos *Read* y *Write* de *Direct_IO* tienen un parámetro adicional que dice en qué posición del fichero queremos leer o escribir.

```
with Direct_IO;
procedure Ejemplo is

   package Dir_IO is new Sequential_IO (Integer);
   use type Dir_IO.Positive_Count;
   --  Con la frase "use type" le estoy diciendo a Ada
   --  que para sumar, restar multiplicar y dividir
   --  numeros de tipo Positive_Count utilice las
   --  funciones "+", "-", "*", "/" que estan en el
   --  paquete Dir_IO (en vez de utilizar la suma, resta,
```

```
      --  multiplicacion y division de numeros enteros).

      Fichero  : Dir_IO.File_Type;
      Posicion : Dir_IO.Positive_Count;

   begin

      Dir_IO.Create
         (File => Fichero,
          Mode => Text_IO.Out_File,
          Name => "Numeros.dat");

      Posicion := 10;
      Dir_IO.Write (Fichero, 500);

      Posicion := Posicion - 5;
      Dir_IO.Write (Fichero, 250);

      Dir_IO.Close (Fichero);

   end Ejemplo;
```

En este ejemplo hemos creado un fichero para escribir números, pero sólamente escribimos dos números. Primero escribimos el número 500 en la posición 10 del fichero (por tanto Ada deja espacio para los 9 primeros números). A continuación escribimos el número 250 en la posición 5 y cerramos el fichero (como no hemos escrito nada en las posiciones 1, 2, 3, 4, 6, 7, 8, 9 lo que hay es “basura”; lo normal es que un programa controle bien lo que está escribiendo para no cerrar el fichero dejando basura dentro de él).

Ambos paquetes proporcionan todos los modos de apertura de fichero que proporciona *Text_IO* (*In_File* y *Out_File*). Sin embargo, cada uno permite un modo más:

- *Sequential_IO* proporciona un tercer modo que nos permite abrir el fichero para añadir información en él (*Append_File*).
- *Direct_IO* proporciona un tercer modo que nos permite abrir el fichero para leer y escribir información en él (*Inout_File*).

10.4. Resumen

En este capítulo hemos visto cómo podemos crear, leer y escribir ficheros. En el siguiente capítulo tienes una colección de ejercicios que debes resolver utilizando Ada.

Apéndice A

Ejercicios

A.1. Identificadores y tipos de datos.

1. Indica qué identificadores son correctos en Ada.

```
MI_COCHE     Casa_3      "Contador"    Numero1

indice 1     Elemento&3  Procedure     %Total

34_Final     Codigo-1    Total_Suma     'x'
```

2. Indica de qué tipo de dato es cada uno de los siguientes valores Ada:

```
167    167.0   'x'  "True"  16.4e5  16e5

  7    '7'     "7"  False   1_000   0.0_5
```

A.2. Expresiones

1. Supongamos que en un programa hemos hecho las siguientes declaraciones:

```
I: Integer := 2;
J: Integer := 3;
X: Float   := 4.0;
Y: Float   := 5.0;
```

   Evalúa las siguientes expresiones e indica el tipo de dato de cada resultado.

```
(a) I + J          (b) I + 5          (c) 2 + 3

(d)  X - 1.5       (e) 2.0 * 2.5      (f) Y / X

(g) J / I          (h) 14 rem 4       (i) J mod I

(j) Integer(X)**I  (k) Y ** (-1)      (l) I ** J

(m) I + J * 2      (n) X * Y ** 2     (o) abs X-Y

(p) X / Y * 2.0    (q) 2.0 * FLOAT(J)
```

2. Evalúa las siguientes expresiones lógicas:

```
(a) True and 10 > 8
(b) 5.0 >= 10.3 or á' > 'b'
(c) 3 not in 1..7
(d) I /= 0 and 14/I > 3  -- Supongamos que I vale 3
(e) 3 > 3 or "hola" /= "HOLA"
```

3. Escribe un programa que reciba desde teclado la siguiente información de un coche: Número de registro, kilometraje al principio y al final del año pasado, y total de litros de gasolina consumidos ese año. A partir de esta información el programa debe generar un informe con la siguiente información:

   - Número de registro:
   - Kilómetros recorridos el año pasado:
   - Combustible consumido:
   - Litros por kilómetro:

4. Escribe un programa que lea del teclado una cantidad en pesetas y nos diga el equivalente en euros.

5. Escribe un programa que lea del teclado una cantidad en euros y nos diga el equivalente en pesetas.

6. Escribe un programa que lea del teclado un peso en libras y onzas y nos diga el equivalente en kg. NOTA: 1 libra son 0.4536 Kg y hay 16 onzas en cada libra.

7. Escribe un programa que lea del teclado un peso en Kg y nos diga el equivalente en libras y onzas.

8. En Europa el consumo de un coche se mide en litros por kilómetro, mientras que en Inglaterra se mide en millas por galón. Escribe un programa que reciba una cantidad en litros por kilómetro y genere su equivalente en millas por galón. NOTA: 1 milla son 1109 Km; 1 galón son 3,785 litros.

9. Escribe un programa que calcule el volumen y el área de una esfera.

$$Volumen = \frac{4\pi r^3}{3} \qquad Area = 4\pi r^2$$

10. Escribe un programa que lea del teclado la longitud de dos lados de un triángulo rectángulo y nos diga el valor de la hipotenusa.

    NOTA: Para poder utilizar la función que calcula la raíz cuadrada de un número, que se llama SQRT, tenemos que utilizar un paquete genérico que se llama Numerics.Generic_Elementary_Function. La forma de utilizarlo es similar a Integer_IO. Por ejemplo:

```
with Ada.Numerics.Generic_Elementary_Functions;
procedure Ejemplo is
    package Math is new
            Ada.Numerics.Generic_Elementary_Functions(Float);
    use Math;
begin
    -- Ya puedo utilizar SQRT, SIN, COS, etc.
end Ejemplo;
```

11. Escribe un programa que reciba desde teclado los coeficientes de una ecuación de segundo grado y calcule las raíces.

12. Escribe un programa que reciba desde teclado las coordenadas de dos puntos (X_1, Y_1), (X_2, Y_2) y calcule la distancia entre ellos.

$$Distancia = \sqrt{(X_1 - X_2)^2 + (Y_1 - Y_2)^2}$$

13. Escribe un programa que lea de teclado una letra en minúscula y nos la escriba en mayúscula.

14. Supongamos que la variable T contiene un carácter entre '0' y '9'. Escribe una sentencia que realice la conversión de T a un entero en el rango 0..9 y lo asigne a una variable de tipo entero.

A.3. Sentencias

A.3.1. if

1. Escribe un programa que lea dos números desde el teclado y nos diga cual de los números es el menor.

2. Escribe un programa que lea del teclado dos caracteres y nos los escriba ordenados (siguiendo el orden del estándar ASCII).

3. Escribe un programa que lea del teclado tres nombres y nos los escriba ordenados de menor a mayor (siguiendo el orden del estándar ASCII).

4. Escribe un programa que nos pida el precio de un artículo de una tienda y el número de artículos de ese tipo que hemos vendido a un cliente. Si el total de la factura es más de 150 euros, el programa hará un descuento de un diez por ciento a la factura.

5. Escribe un programa que lea de teclado una cantidad en euros y nos diga cuantos billetes de 100, 50, 10 y monedas de 2, 1, 0,5, 0,2, 0,1 y 0,01 euro debemos dar para pagar esa cantidad.

6. Utilizando la sentencia *if*, escribe un programa que lea una nota (un número entre 1 y 10) y genere en pantalla la nota correspondiente: suspenso, aprobado, bien, notable, sobresaliente.

A.3.2. case

1. Utilizando la sentencia *case*, escribe un programa que lea un número entre 1 y 12 que representa uno de los meses del año y genere en pantalla un mensaje que nos diga la estación del año asociada.

   NOTA: Para el programa, la primavera abarca los meses de Marzo, Abril y Mayo; el verano, los meses de Junio, Julio y Agosto; el otoño, los meses de Septiembre, Octubre y Noviembre; y el invierno, los meses de Diciembre, Enero y Febrero.

   ```
   Ejemplo:

       Dime un mes (1..12): 12
       Es invierno
   ```

2. Utiliza la sentencia *case* para escribir un programa que lea una nota (un número entre 1 y 10) y genere en pantalla la nota correspondiente: suspenso, aprobado, bien, notable, sobresaliente.

A.3.3. loop

1. Escribe un programa que, mediante dos bucles anidados, genere en pantalla la siguiente figura:

```
+ + + + +
+ + + +
+ + +
+ +
+
```

2. Escribe un programa que nos muestre en pantalla las tablas de multiplicar.

3. Escribe un programa que, utilizando bucles, genere en pantalla la siguiente tabla de multiplicar. NOTA: El límite superior se especifica desde teclado.

```
1   2   3   4   5
2   4   6   8   10
3   6   9   12  15
4   8   12  16  20
5   10  15  20  25
```

4. Escribe un programa que nos muestre en pantalla una tabla con los cuadrados de los doce primeros números.

5. Escribe un programa que muestre en pantalla una tabla con los valores de los cuadrados y cubos de todos los números comprendidos entre N1 y N2. Los valores de N1 y N2 lo lee el programa desde el teclado.

6. Escribe un programa que calcule la suma de todos los números enteros positivos comprendidos entre 1 y N. El valor de N lo lee el programa desde el teclado.

7. Escribe un programa que, mediante bucles, calcule la media de todos los números comprendidos entre 1 y N. El valor de N lo lee el programa desde el teclado.

8. Escribe un programa que calcule la suma de todos los números impares menores que n. El valor de n lo lee el programa desde el teclado. NOTA: Escribe tres versiones del programa; una utilizando sólo bucles *loop*, otro utilizando sólo bucles *while* y otra utilizando sólo bucles *for*.

9. Escribe un programa que reciba desde teclado un cierto número de números reales y escriba en pantalla:

   a) El número menor.

   b) El número mayor.

   c) El valor medio de todos los números.

   Supongamos que este programa lo necesitan tres personas diferentes, y cada una quiere leer los datos de una forma diferente:

   - La primera persona quiere que siempre se lean 10 números.
   - La segunda persona quiere que no se fije esta cantidad. Ella quiere decir al programa (cada vez que lo ejecute) el número de datos que va a introducir.
   - La tercera persona dice que ella no sabe cuantos datos va a introducir. Quiere que el programa le pregunte si quedan más datos.

10. Escribe un programa que calcule el factorial de cualquier número entero positivo.

11. Escribe un programa que nos diga si un número cualquiera es primo o no.

12. Modifica el programa anterior para que calcule todos los números primos que hay entre 1 y el número que digamos nosotros por teclado.

13. Dada la siguiente función matemática: $f(x) = 3x^3 - 5x^2 + 2x - 20$.

    a) Genera una tabla con todos los valores de $f(x)$ existentes en el intervalo entero -10..10.

    b) Genera la tabla con los valores de $f(x)$ asociados a los valores de x comprendidos en el rango de números reales -2..2 en incrementos de 0,1.

14. Escribe un programa que lea N números desde teclado y nos diga cual de ellos es el menor y su posición.

15. Escribe una calculadora de sobremesa que sume, reste, multiplique y divida números de forma acumulativa. Cuando comienza su ejecución la calculadora contiene el valor 0.

```
Ejemplo:

    Resultado = 0
```

```
+12
Resultado = 12
-4
Resultado = 8
*3
Resultado = 24
/4
Resultado = 6
```

16. Un banco nos da un interés del 3,5 por ciento al año. Supongamos que ponemos X euros fijos en el banco. Escribe un programa que calcule cuantos años tendremos que esperar hasta que la cuenta tenga 1000 euros más que cuando la creamos.

17. Escribe un programa que calcule cuál es el número entero más pequeño que comple que:

$$\sum_{i=1}^{k} i^2 > n$$

NOTA: El valor de n se lee desde el teclado.

18. El ayuntamiento de una ciudad ha hecho un estudio de su población y ha obtenido los siguientes resultados:

    - En 1994 tenían 26.000 habitantes.
    - Cada año nace un 0,7 por ciento de personas y muere un 0,6 por ciento.
    - Cada año hay una inmigración de 300 personas y una emigración de 325 personas.

    Escribe un programa que calcule el número de habitantes que habrá en cualquier año.

19. El alquiler de un coche nos cuesta 60 euros por día más 1 euro por cada kilómetro recorrido. Escribe un programa que reciba desde teclado un número de días y un número de kilómetros y nos diga cuanto nos va a costar el alquiler del coche.

20. Escribe un programa que reciba una hora desde el teclado (horas, minutos y segundos) y nos escriba en pantalla el valor de la hora en el siguiente segundo.

```
Ejemplo:
    Hora: 0:3:59
    Siguiente segundo: 0:4:00

    Hora: 2:59:59
    Siguiente segundo: 3:00:00
```

21. Modifica el programa anterior para que el programa se convierta en un reloj. NOTA: Para que el programa Ada espere un segundo debemos utilizar la sentencia *delay 1.0;*.

A.4. Atributos

1. Escribe un programa que muestre en pantalla cual es el entero más pequeño y más grande que es capaz de manejar nuestro compilador de Ada. Haz lo mismo con los números reales.

2. Escribe un programa que muestre en pantalla la tabla de caracteres ASCII de los caracteres que hay entre la posición 65 y la posición 127.

3. Escribe un programa que lea del teclado un caracter y calcule el siguiente caracter (siguiendo el orden de la tabla ASCII).

4. Escribe un programa que lea del teclado dos caracteres y escriba en pantalla todos los caracteres ASCII que están entre ellos (siguiendo el orden de la tabla ASCII).

A.5. Estructuras de datos

A.5.1. Vectores

1. Escribe un programa que lea desde teclado 20 números y los vuelva a escribir en pantalla en orden inverso.

2. Escribe un programa que lea desde teclado 20 números y los vuelva a escribir en pantalla sin repetir los números que estén duplicados. NOTA: No es necesario que los números estén ordenados para que el programa funcione correctamente.

3. Declara un enumerado que contenga los días de la semana. A partir de esta declaración, crea la tabla *Mañana* que utilizaremos para saber cuál es el día siguiente a cualquier día de la semana.

4. Declara una tabla que sirva para traducir números romanos en números enteros positivos. Utilizando esta tabla, escribe un programa que lea un número romano y lo traduzca a un entero positivo.

5. Escribe un programa que reciba un número entero positivo y genere el número romano equivalente.

6. Escribe un programa que calcule y almacene en una tabla los 50 primeros números primos. Para calcular si un número es primo lo dividimos por todos los que tenemos en la tabla; si es primo, lo insertamos en la tabla.

A.5.2. Cadenas de caracteres

1. Escribe un programa que cuente el número de espacios en blanco que hay en una línea de texto.

```
Ejemplo:                1         2         3
                 12345678901234567890123456789012345
    TEXTO    : Estoy   aprendiendo a   programar.
    RESULTADO: 7.
```

2. Escribe un programa que lea un texto y nos indique:

   - Cuántas letras contiene (mayúsculas y minúsculas).
   - Cuántos números contiene.
   - Cuántos caracteres especiales (caracteres que no son letras ni números).

3. Escribe un programa que lea tres líneas de texto y cuente el número de vocales.

4. Escribe un programa que sustituya los espacios en blanco consecutivos por un único espacio.

```
Ejemplo:

    TEXTO    : Estoy   aprendiendo a    programar.
    RESULTADO: Estoy aprendiendo a programar.
```

5. Escribe un programa que centre en pantalla la frase que nosotros introduzcamos desde el teclado.

6. Escribe un programa que reciba desde teclado una línea con un máximo de 80 caracteres y nos escriba en las líneas siguientes cada una de las frases que hemos puesto. NOTA: Suponemos que el separador de frases es el punto.

7. Escribe un programa que lea una fecha en formato americano (mes/día/año) y nos la genere en el formato utilizado en España (día/mes/año) y en el formato estándar ISO (año-mes-día).

```
Ejemplo:

    Introduce la fecha en formato americano: 10/25/98

    25/10/98
    1998-10-25
```

8. Escribe un programa que busque una palabra dentro de una frase y nos indique la posición donde la encontró. Tanto la palabra como la frase la introducimos nosotros desde el teclado.

```
Ejemplo:

    FRASE   : Quiero buscar una palabra en esta frase.
    PALABRA : una
    POSICION: 15
```

9. Modifica el programa anterior para que nos diga cuantas veces aparece una determinada palabra dentro de una frase.

10. Escribe un programa que busque una palabra dentro de una frase y la sustituya por otra palabra.

```
Ejemplo:

    FRASE    : Esta es una posible frase.
    BUSCA    : una
    SUSTITUYE: otra
    RESULTADO: Esta es otra posible frase.
```

11. Escribe un programa que, añadiendo espacios, justifique un texto a un tamaño que especificamos por teclado.

    NOTA: Para justificar debemos calcular el número de espacios que debemos insertar y seguidamente recorrer el texto de izquierda a derecha (o de derecha izquierda) buscando los espacios en blanco y añadiendo un nuevo espacio hasta completar dicha cantidad.

```
Ejemplo:

    TEXTO    : Este es el texto a justificar.
    TAMANO   : 40
                          1         2         3         4
                 12345678901234567890123456789012345678901234567890
    RESULTADO: Este    es   el   texto   a   justificar
```

12. El método simple de cifrado de información consiste en sustituir cada letra del abecedario por otra que está un 3 posiciones por delante de ella. Por ejemplo, la A se sustituye por la C, la B se sustituye por la D, la C por la E y así sucesivamente.

```
Ejemplo:

    Texto original  : JAVIER
    Texto codificado: LCXKGT
```

    a) Escribe un programa que lea un mensaje con un tamaño máximo de 60 caracteres, lo cifre utilizando el método César, y escriba en pantalla el resultado. NOTA: Suponemos que todas las letras del mensaje están siempre en mayúscula; si hay alguna en minúscula se deja sin codificar.

    b) Escribe un programa que lea un mensaje cifrado por el método César, lo descifre y genere en pantalla el mensaje original.

    c) Escribe un programa que lea un mensaje cifrado cuyo desplazamiento desconocemos, y escriba en pantalla todos los posibles mensajes originales.

13. Un texto palíndromo es un texto que se lee igual de izquierda a derecha que de derecha a izquierda. Escribe un programa que lea un texto y nos diga si es palíndromo.

14. Escribe un programa que calcule la frecuencia de todos los caracteres de que consta un mensaje.

15. Los elementos químicos se representan mediante una o dos letras: la primera siempre en mayúscula y la segunda en minúscula (por ejemplo C, Hg, Au). Escribe un programa que calcule el peso molecular de una fórmula química.

    NOTAS:

    - Los valores de los pesos moleculares están en cualquier libro de química.
    - Como no podemos poner en el teclado subíndices, utilizaremos la representación H2O para representar H_2O(2 moléculas de Hidrógeno y una de Oxígeno).

A.5.3. Matrices

1. Escribe un programa que calcule la suma de todos los elementos de la diagonal principal de una matriz cuadrada.

2. Escribe un programa que calcule la suma de todos los elementos de la diagonal secundaria de una matriz cuadrada.

3. Escribe un programa que sume todos los elementos del borde de cualquier matriz.

4. Escribe un programa que busque una submatriz dentro de una matriz. Por ejemplo:

```
Submatriz          Matriz
  1  2          8  7  6  5
  3  4          1  2  4  9
                3  4  5  7
```

   En este caso, el programa debe decir que ha encontrado la submatriz a partir del elemento 2,1 (fila 2, columna 1) de la matriz.

5. Escribe un programa que invierta horizontalmente cualquier matriz. Por ejemplo:

```
  Matriz          Resultado
8  7  6  5       3  4  5  7
1  2  4  9       1  2  4  9
3  4  5  7       8  7  6  5
```

   El contenido de la primera fila ha pasado a ser el contenido de la tercera fila, el contenido de la segunda fila permanece igual y el contenido de la tercera fila ha pasado a ser el contenido de la primera fila.

6. Repite el ejercicios anterior pero haciendo el movimiento en vertical. Por ejemplo:

```
  Matriz          Resultado
8  7  6  5       5  6  7  8
1  2  4  9       9  4  2  1
3  4  5  7       7  5  4  3
```

7. Escribe un programa que mueva todos los elementos de la matriz a la siguiente fila. La primera fila se llena con el contenido de la última fila. Por ejemplo:

```
  Matriz              Resultado
8  7  6  5          3  4  5  7
1  2  4  9          8  7  6  5
3  4  5  7          1  2  4  9
```

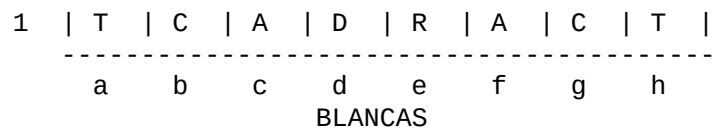
8. Repite el ejercicio anterior pero haciendo el movimiento vertical. Por ejemplo:

```
  Matriz              Resultado
8  7  6  5          5  8  7  6
1  2  4  9          9  1  2  4
3  4  5  7          7  3  4  5
```

9. Utiliza una matriz de caracteres para almacenar todas las letras de cualquier sopa de letras. Escribe un programa que busque una palabra en la sopa de letras. Recuerda que en la sopa de letras cada palabra puede estar en horizontal, vertical o en diagonal.

10. Un tablero de ajedrez tiene 64 cuadros. Las columnas se nombran mediante las letras a,b,c,d,e,f,g y h, y las filas mediante los números del 1 al 8. Las piezas de ajedrez son: reina, rey, alfil, caballo, torre y peón. Las piezas son blancas o negras. Escribe una declaración Ada que describa la posición inicial del tablero, que es la siguiente:

```
                    NEGRAS
   ---------------------------------------
8  | T* | C* | A* | D* | R* | A* | C* | T* |
   ---------------------------------------
7  | P* | P* | P* | P* | P* | P* | P* | P* |
   ---------------------------------------
6  |    |    |    |    |    |    |    |    |   T: Torre
   ---------------------------------------   C: Caballo
5  |    |    |    |    |    |    |    |    |   A: Alfil
   ---------------------------------------   D: Dama
4  |    |    |    |    |    |    |    |    |   R: Rey
   ---------------------------------------   P: Peon
3  |    |    |    |    |    |    |    |    |
   ---------------------------------------   *: Ficha negra
2  | P  | P  | P  | P  | P  | P  | P  | P  |
   ---------------------------------------
```



11. Realiza la declaración de una matriz tridimensional de números enteros. Utilizando esta declaración, escribe un programa que calcule el número de elementos positivos que contiene una matriz tridimensional llena de números enteros.

12. Escribe un programa que lea desde teclado los elementos de una matriz de cualquier tamaño, calcule la suma de cada una de sus filas y columnas almacenando el resultado en dos vectores (uno con la suma de las filas y otro con la suma de las columnas) y finalmente escriba estos vectores en pantalla.

A.6. Procedimientos y funciones

1. Escribe una función que evalúe el signo de un número. La función debe retornar el valor 1 si el número es mayor que cero, 0 si el número es igual a cero y -1 si el número es menor que cero.

2. Escribe un programa que contenga:

   a) Una función que calcule el máximo de dos números enteros.

   b) Una función que recibe un carácter y nos diga si es una letra o no.

   c) Una función que recibe un carácter y nos diga si es una letra o un número.

3. Escribe una función que reciba una letra mayúscula y devuelva su equivalente en minúscula.

4. Escribe la función *mcd()* que calcula el máximo común divisor de dos números enteros positivos utilizando el algoritmo de Euclides. El algoritmo de Euclides es el siguiente:

   a) Realiza la división entera.

   b) Si el resto es cero, el *mcd* es el divisor.

   c) En caso contrario, copia el valor del divisor en el dividendo, el valor del resto en el divisor, y vuelve al primer paso.

5. Escribe una función que, mediante bucles, calcule la potencia n de un número (X^n).

6. Escribe una función que, mediante bucles, calcule el factorial de un número ($X!$).

7. Cuando un lenguaje de programación no tiene las funciones matemáticas, podemos utilizar el desarrollo en serie de McLaurin para calcularlas. Por ejemplo, la función *seno()* podemos evaluarla calculando la siguiente serie:

$$sen(X) = X - \frac{X^3}{3!} + \frac{X^5}{5!} - \frac{X^7}{7!} + \frac{X^9}{9!} + ...$$

   Utilizando las funciones de los dos problemas anteriores (X^n y $X!$), escribe una función que calcule *sen(X)* realizando el desarrollo en serie de McLaurin hasta que el valor de $\frac{X^n}{n!}$ sea menor que 10^{-5}.

8. Modifica el programa anterior para que también evalúe e^x realizando la siguiente serie mientras no se alcance una precisión de 10^{-4}.

$$E(X) = 1 + X + \frac{X^2}{2!} + ... + \frac{X^n}{n!}$$

9. Escribe una función que calcule la suma de los 10 elementos de un vector. NOTA: La función debe trabajar correctamente independientemente del rango del índice del vector.

10. Escribe una función que calcule la suma de todos los elementos de un vector, independientemente del tamaño del vector y del rango del índice del vector.

11. Escribe una función que recibe un vector con números enteros y nos diga cual es la posición del menor.

12. Utilizando la función anterior, escribe un programa que reciba N números, los inserte en una tabla y escriba la tabla en pantalla ordenada de menor a mayor.

13. Escribe una función que sume dos vectores de igual tamaño y rangos cualesquiera, dando como resultado otro vector de igual tamaño.

14. Escribe una calculadora que, mediante funciones y procedimientos, permita sumar, restar, multiplicar y dividir números romanos.

15. Escribe una calculadora que permita sumar, restar y multiplicar (escalar y vectorialmente) vectores.

16. La amplitud de un vector $(V_1, V_2, ..., V_n)$ se calcula aplicando la siguiente fórmula:

$$L = \sqrt{V_1^2 + V_2^2 + ... + V_n^2}$$

Escribe una función que calcule la amplitud de un vector de números reales.

a) Considera que el vector sólo contiene cuatro elementos.

b) Considera que el vector tiene un número arbitrario de elementos.

17. Decimos que dos vectores U y V son ortogonales cuando se verifica que:

$$\sum_{i=1}^{n} u_i * v_i = 0$$

Escribe una función que nos sirva para saber si dos vectores (de cualquier tamaño) son ortogonales.

18. Escribe una función que compruebe si el nombre de un identificador sigue las reglas de Ada.

19. Utilizando la función anterior, escribe un programa que lea de teclado una línea de texto y nos diga cuantas palabras son identificadores Ada válidos.

20. Definimos la rotación de un vector a la derecha como una operación que mueve cada elemento una posición a la derecha y el último lo coloca en la primera posición. Escribe una función que rote un vector un número determinado de posiciones a la derecha.

21. En estadística, la mediana es el valor central de un conjunto de valores ordenados. En caso de que el número de valores sea par, la mediana es el valor medio de los dos valores centrales. Escribe una función que reciba un número arbitrario (y no ordenado) de valores de entrada y retorne la mediana.

22. En estadística, la moda es el valor que más veces se repite en un conjunto de datos. Escribe una función que reciba un número arbitrario de valores de entrada y retorne la moda.

23. Supongamos que nuestro lenguaje de programación no es capaz de trabajar con matrices de rango (1..N, 1..M). Escribe un programa que, mediante funciones, nos permita trabajar con un un vector como si fuese una matriz. Por ejemplo, cuando queremos acceder al elemento (4,4), accedemos al elemento 44 del vector; cuando queremos acceder al elemento (2,3), accedemos al elemento 23, y así sucesivamente.

24. Generiliza la solución del problema anterior para que se pueda trabajar con matrices de cualquier rango.

25. Escribe las declaraciones Ada necesarias para representar la siguiente tabla que indica si un país es vecino de otro:

	Belgica	Italia	Francia
Belgica	no	si	no
Francia	si	no	si
Italia	no	si	no

    Añade algunos países más y escribe la función *Número_De_Vecinos()* que recibe como parámetro un país y una tabla similar a la descrita anteriormente y calcula el número de vecinos de dicho país.

26. Escribe una función que nos diga si una matriz es simétrica.

27. Escribe las funciones *Fila()* y *Columna()* que nos permiten obtener los elementos de una determinada fila o columna de cualquier matriz de números enteros positivos.

28. Escribe una función que calcule el producto escalar de dos vectores.

29. Utilizando las funciones de los dos problemas anteriores, escribe un programa que realice la multiplicación de dos matrices cualesquiera.

30. Escribe un programa que lea un mensaje y nos codifique y decodifique mensajes en Morse. La tabla de codificación Morse es la siguiente:

A .-	H	O ---	U ..-
B -...	I ..	P .--.	V ...-
C -.-.	J .---	Q --.-	W .--
D -..	K -.-	R .-.	X -..-
E .	L .-..	S ...	Y -.--
F ..-.	M --	T -	Z --..
G --.	N -.		

    NOTA: La codificación la debe realizar el procedimiento *Codificar* y la decodificación el procedimiento *Decodificar*.

31. Escribe un programa que, mediante funciones, nos permita sumar, restar, multiplicar y dividir dos fracciones. NOTA: El resultado debe mostrarse en pantalla también en forma de fracción.

32. Una empresa nos ha encargado un programa para llevar la gestión de su inventario. Por cada artículo necesitan que el programa recuerde:

    - La identificación del artículo (un código de cuatro caracteres).
    - Una breve descripción de hasta 60 caracteres.
    - El número de artículos existentes en el almacén.
    - El precio de venta.

    Escribe un programa que lee esta información desde teclado y a continuación espera a que le demos una orden. Las órdenes posibles son:

    - INFO xxxx: Escribir toda la información del artículo xxxx.
    - VENDER xxxx: Registrar que se ha vendido un artículo del producto xxxx.
    - COMPRAR xxxx: Registrar que se ha comprado un artículo del producto xxxx.

    NOTA: Cada una de estas operaciones la debe realizar un procedimiento diferente.

A.6.1. Recursividad

1. Una empresa nos hace la siguiente oferta de trabajo:

    - El primer año nos paga 1000 euros.
    - El resto de los años tendremos un aumento de sueldo anual de 4 por ciento.

    Escribe una función recursiva que calcule el sueldo que tendremos dentro de X años.

2. Una forma de calcular el máximo común divisor de dos números enteros positivos es utilizar la siguiente definición recursiva:

    - $mcd(m, n) = m$, si $m = n$.
    - $mcd(m, n) = mcd(m - n, n)$, si $m > n$.
    - $mcd(m, n) = mcd(m, n - m)$ en el resto de los casos.

    Escribe la función recursiva $mcd()$ que calcula el máximo común divisor de dos números aplicando esta definición.

3. Escribe un procedimiento recursivo que escriba en pantalla todos los dígitos de un número entero positivo en orden inverso. NOTA: Utiliza la división por 10 para extraer cada uno de los dígitos.

```
Ejemplo  : 1234
Resultado: 4321
```

4. Dado un número natural N y una base de numeración B inferior a N, escribe un procedimiento recursivo que escriba en pantalla N en base B.

```
Ejemplo:
     Numero (N): 15
     Base   (B): 2
     Resultado : 1111
```

5. Escribe un procedimiento recursivo que multiplique dos números naturales utilizando la siguiente fórmula:

   - $a * b = a$, si $b = 1$
   - $a * b = a * (b - 1) + a$, si $b > 1$

6. Los coeficientes de un binomio se pueden definir de la siguiente forma:

$$\binom{n}{0} = 1$$

$$\binom{n}{n} = 1$$

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} 0 < k < n$$

   Escribe una función recursiva que calcule el valor de: $\binom{i}{j}$.

A.7. Ficheros

1. Escribe un programa que nos permita guardar en disco la siguiente información de los jugadores de un equipo de fútbol:

   - Nombre.
   - Peso.
   - Estado civil (soltero, casado, divorciado).

2. Escribe un programa que, a partir de la información contenida en el fichero anterior, nos permita crear otro fichero que contiene la misma información y además la edad de cada jugador.

3. Escribe un programa que, a partir del fichero generado en el apartado anterior, nos genere dos ficheros: uno con los jugadores menores de 25 años y otro con los jugadores mayores de 25 años.

4. Escribe un programa que funcione como una agenda electrónica. Cada ficha contiene el nombre de una persona y su teléfono. La agenda electrónica nos debe permitir: añadir una nueva ficha, borrarla, buscarla, escribir en pantalla una ficha o escribirlas todas y ordenarlas por el nombre. La agenda debe mantener la información aunque la apaguemos.

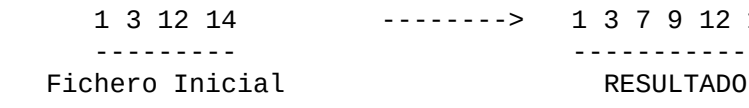
5. Supongamos que tenemos el fichero PERSONAS.DAT con el nombre y la edad de un conjunto de personas. Se pide:

   - Escribir un programa que copie el contenido del fichero en otro fichero llamado COPIA.DAT.
   - Modificar el programa anterior para que nos permita insertar un elemento en el fichero de salida. El programa nos debe preguntar si queremos insertar el elemento por el principio, por el final o en una determinada posición del fichero de salida.
   - Modificar el programa para que el programa genere el fichero de salida EDADES.DAT, que solamente contiene las edades de todas las personas.
   - Escribir un programa que lea el fichero PERSONAS.DAT y le sume 1 a la edad de todas las personas.

6. Escribe un programa que copie cualquier fichero en otro.

7. Escribe un programa que calcule la longitud de un fichero.

8. Escribe un programa que inserte ordenadamente dos números en un fichero.



9. Escribe un programa que cuente el número de líneas, palabras y caracteres que hay en cualquier fichero de texto.

10. Escribe un programa que nos pase a mayúsculas todo el contenido de un fichero de texto.

11. Escribe un programa que calcule la frecuencia de todas las palabras que hay en un fichero de texto.

12. Escribe un programa que muestre en pantalla el contenido de un fichero. Cada vez que se llene la pantalla (se escriben en pantalla 23 líneas) el programa debe esperar a que se pulse una tecla.

13. Escribe un programa que nos pida una palabra y el nombre de un fichero y nos escriba en pantalla todas las líneas del fichero que contienen esa palabra.

14. Para llevar el control de las notas de un examen nos han encargado escribir un programa que almacene en un fichero la siguiente información:

    - Código del alumno.
    - La nota obtenida en cada una de las 10 preguntas.

    Además, debemos escribir otro programa que nos diga el código del alumno que sacó la mejor nota en el examen.

15. Escribe el programa juego del *el ahorcado*. El jugador debe adivinar una palabra que está almacenada en un fichero. Para ello dice una letra y el programa le enseña en pantalla los caracteres que ha acertado. El juego termina cuando el jugador acierta la palabra completa o falla un determinado número de intentos.

```
Ejemplo:
   ****
   Adivina una letra: O
   O*O*
   Adivina un letra: M
   <<<< NO HAY NINGUNA M. TE PERMITO 3 FALLOS MAS >>>>
   Adivina una letra: L
   OLO*
   Adivina una letra: R
   OLOR
   >>>>>> GANASTE
```

16. Modifica todos los ejercicios de la sección 7.2 para que utilicen ficheros de acceso directo (en vez de secuenciales).

Apéndice B

Text_IO

Este apéndice contiene la interfaz completa de los paquetes de que consta la jerarquía de paquetes de Text_IO.

B.1. Text_IO

```
package Text_IO is

   type File_Type is limited private;
   type File_Mode is (In_File, Out_File, Append_File);

   type Count is range 0 .. System.Parameters.Count_Max;

   subtype Positive_Count is Count range 1 .. Count'Last;

   ----------------------
   -- File Management --
   ----------------------

   procedure Create
     (File : in out File_Type;
      Mode : in File_Mode := Out_File;
      Name : in String := "";
      Form : in String := "");
   --  Si el fichero no existe, lo crea.
   --  Si el fichero ya existe, lo borra y crea uno nuevo.
   --  Si quieres crear el fichero en un determinado directorio
   --  debes dar en el parametro "Name" el camino completo.
   --  Por ejemplo: Name => "/tmp/Numeros.dat"
   --  El campo "Form" no se suele utilizar.

   procedure Open
     (File : in out File_Type;
```

```
   Mode : in File_Mode;
   Name : in String;
   Form : in String := "");
--  Abre un fichero que ya existe. Los parametros "Name" y
--  "Form" son identicos a los de "Create".

procedure Close  (File : in out File_Type);
--  Cierra un fichero y elimina el vinculo entre la variable
--  fichero y el nombre del fichero (por lo que puedo
--  reutilizar la variable fichero).

procedure Delete (File : in out File_Type);
--  Borra un fichero y elimina el vinculo entre la variable
--  fichero y el nombre del fichero.

procedure Reset  (File : in out File_Type; Mode : in File_Mode);
--  Reabre el fichero con el modo especificado.

procedure Reset  (File : in out File_Type);
--  Reabre el fichero con el mismo modo que tiene actualmente.

function Mode (File : in File_Type) return File_Mode;
--  Consulta el modo en que esta abierto el fichero.

function Name (File : in File_Type) return String;
--  Consulta el nombre del fichero.

function Form (File : in File_Type) return String;
--  Consulta el formato (si se dio alguno) asociado al fichero.

function Is_Open (File : in File_Type) return Boolean;
--  Me dice si la variable fichero tiene asociado algún
--  fichero abierto o si ya lo he cerrado.

-------------------------------------
-- Column, Line, and Page Control --
-------------------------------------

procedure New_Line (File : in File_Type; Spacing : in Positive_Count := 1);
procedure New_Line (Spacing : in Positive_Count := 1);

procedure Skip_Line (File : in File_Type; Spacing : in Positive_Count := 1);
procedure Skip_Line (Spacing : in Positive_Count := 1);

function End_Of_File (File : in File_Type) return Boolean;
function End_Of_File return Boolean;

-----------------------------
-- Characters Input-Output --
-----------------------------

procedure Get (File : in File_Type; Item : out Character);
procedure Get (Item : out Character);

procedure Put (File : in File_Type; Item : in Character);
procedure Put (Item : in Character);
```

```
procedure Look_Ahead
  (File        : in File_Type;
   Item        : out Character;
   End_Of_Line : out Boolean);
--  Nos dice cual es la siguiente letra que hay en el fichero,
--  pero no la quita del buffer de entrada.

procedure Look_Ahead
  (Item        : out Character;
   End_Of_Line : out Boolean);
--  Nos dice cual es la siguiente tecla que se ha pulsado,
--  pero no la quita del buffer de entrada.

procedure Get_Immediate
  (File : in File_Type;
   Item : out Character);

procedure Get_Immediate
  (Item : out Character);

procedure Get_Immediate
  (File      : in File_Type;
   Item      : out Character;
   Available : out Boolean);

procedure Get_Immediate
  (Item      : out Character;
   Available : out Boolean);
--  No dice si se ha pulsado alguna tecla (Available)
-   y que tecla es la pulsada (Item).

--------------------------
-- Strings Input-Output --
--------------------------

procedure Get (File : in File_Type; Item : out String);
procedure Get (Item : out String);

procedure Put (File : in File_Type; Item : in String);
procedure Put (Item : in String);

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

procedure Get_Line
  (Item : out String;
   Last : out Natural);

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

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

```
   --  Exceptions

   Status_Error : exception;
   Mode_Error   : exception;
   Name_Error   : exception;
   Use_Error    : exception;
   Device_Error : exception;
   End_Error    : exception;
   Data_Error   : exception;
   Layout_Error : exception;

private
   . . .
end Text_IO;
```

B.2. Integer_IO

```
generic
   type Num is range <>;

package Integer_IO is

   Default_Width : Field := Num'Width;
   Default_Base  : Number_Base := 10;

   procedure Get
     (File  : in File_Type;
      Item  : out Num;
      Width : in Field := 0);

   procedure Get
     (Item  : out Num;
      Width : in Field := 0);

   procedure Put
     (File  : in File_Type;
      Item  : in Num;
      Width : in Field := Default_Width;
      Base  : in Number_Base := Default_Base);

   procedure Put
     (Item  : in Num;
      Width : in Field := Default_Width;
      Base  : in Number_Base := Default_Base);

   procedure Get
     (From : in String;
      Item : out Num;
      Last : out Positive);

   procedure Put
     (To   : out String;
      Item : in Num;
```

```
      Base : in Number_Base := Default_Base);

private
   pragma Inline (Get);
   pragma Inline (Put);

end Integer_IO;
```

B.3. Float_IO

```
generic
   type Num is digits <>;

package Float_IO is

   Default_Fore : Field := 2;
   Default_Aft  : Field := Num'digits - 1;
   Default_Exp  : Field := 3;

   procedure Get
     (File  : in File_Type;
      Item  : out Num;
      Width : in Field := 0);

   procedure Get
     (Item  : out Num;
      Width : in Field := 0);

   procedure Put
     (File : in File_Type;
      Item : in Num;
      Fore : in Field := Default_Fore;
      Aft  : in Field := Default_Aft;
      Exp  : in Field := Default_Exp);

   procedure Put
     (Item : in Num;
      Fore : in Field := Default_Fore;
      Aft  : in Field := Default_Aft;
      Exp  : in Field := Default_Exp);

   procedure Get
     (From : in String;
      Item : out Num;
      Last : out Positive);

   procedure Put
     (To   : out String;
      Item : in Num;
      Aft  : in Field := Default_Aft;
      Exp  : in Field := Default_Exp);

end Float_IO;
```

B.4. Enumeration_IO

```
generic
   type Enum is (<>);

package Enumeration_IO is

   Default_Width : Field := 0;
   Default_Setting : Type_Set := Upper_Case;

   procedure Get (File : in File_Type; Item : out Enum);
   procedure Get (Item : out Enum);

   procedure Put
     (File  : in File_Type;
      Item  : in Enum;
      Width : in Field := Default_Width;
      Set   : in Type_Set := Default_Setting);

   procedure Put
     (Item  : in Enum;
      Width : in Field := Default_Width;
      Set   : in Type_Set := Default_Setting);

   procedure Get
     (From : in String;
      Item : out Enum;
      Last : out positive);

   procedure Put
     (To   : out String;
      Item : in Enum;
      Set  : in Type_Set := Default_Setting);

end Enumeration_IO;
```

Apéndice C

Sequential_IO

```
generic
   type Element_Type (<>) is private;

package Sequential_IO is

   type File_Type is limited private;

   type File_Mode is (In_File, Out_File, Append_File);

   ----------------------
   -- File management --
   ----------------------

   procedure Create
     (File : in out File_Type;
      Mode : in File_Mode := Out_File;
      Name : in String := "";
      Form : in String := "");

   procedure Open
     (File : in out File_Type;
      Mode : in File_Mode;
      Name : in String;
      Form : in String := "");

   procedure Close  (File : in out File_Type);
   procedure Delete (File : in out File_Type);
   procedure Reset  (File : in out File_Type; Mode : in File_Mode);
   procedure Reset  (File : in out File_Type);

   function Mode    (File : in File_Type) return File_Mode;
   function Name    (File : in File_Type) return String;
   function Form    (File : in File_Type) return String;

   function Is_Open (File : in File_Type) return Boolean;
```

```
   ---------------------------------
   -- Input and output operations --
   ---------------------------------

   procedure Read  (File : in File_Type; Item : out Element_Type);
   procedure Write (File : in File_Type; Item : in Element_Type);

   function End_Of_File (File : in File_Type) return Boolean;

   ----------------
   -- Exceptions --
   ----------------

   Status_Error : exception;
   Mode_Error   : exception;
   Name_Error   : exception;
   Use_Error    : exception;
   Device_Error : exception;
   End_Error    : exception;
   Data_Error   : exception;

private
   . . .
end Sequential_IO;
```

Apéndice D

Direct_IO

```
generic
   type Element_Type is private;
package Direct_IO is

   type File_Type is limited private;
   type File_Mode is (In_File, Inout_File, Out_File);
   type Count is new System.Direct_IO.Count;

   subtype Positive_Count is Count range 1 .. Count'Last;

   ----------------------
   -- File Management --
   ----------------------

   procedure Create
     (File : in out File_Type;
      Mode : in File_Mode := Inout_File;
      Name : in String := "";
      Form : in String := "");

   procedure Open
     (File : in out File_Type;
      Mode : in File_Mode;
      Name : in String;
      Form : in String := "");

   procedure Close  (File : in out File_Type);
   procedure Delete (File : in out File_Type);
   procedure Reset  (File : in out File_Type; Mode : in File_Mode);
   procedure Reset  (File : in out File_Type);

   function Mode (File : in File_Type) return File_Mode;
   function Name (File : in File_Type) return String;
   function Form (File : in File_Type) return String;

   function Is_Open (File : in File_Type) return Boolean;
```

```
   ---------------------------------
   -- Input and Output Operations --
   ---------------------------------

   procedure Read
     (File : in File_Type;
      Item : out Element_Type;
      From : in Positive_Count);

   procedure Read
     (File : in File_Type;
      Item : out Element_Type);

   procedure Write
     (File : in File_Type;
      Item : in Element_Type;
      To   : in Positive_Count);

   procedure Write
     (File : in File_Type;
      Item : in Element_Type);

   procedure Set_Index (File : in File_Type; To : in Positive_Count);

   function Index (File : in File_Type) return Positive_Count;
   function Size  (File : in File_Type) return Count;

   function End_Of_File (File : in File_Type) return Boolean;

   ----------------
   -- Exceptions --
   ----------------

   Status_Error : exception;
   Mode_Error   : exception;
   Name_Error   : exception;
   Use_Error    : exception;
   Device_Error : exception;
   End_Error    : exception;
   Data_Error   : exception;

private
   . . .
end Direct_IO;
```

Apéndice E

Funciones matemáticas

```
generic
   type Float_Type is digits <>;

package Ada.Numerics.Generic_Elementary_Functions is
pragma Pure (Generic_Elementary_Functions);

   function Sqrt    (X           : Float_Type'Base) return Float_Type'Base;
   function Log     (X           : Float_Type'Base) return Float_Type'Base;
   function Log     (X, Base     : Float_Type'Base) return Float_Type'Base;
   function Exp     (X           : Float_Type'Base) return Float_Type'Base;
   function "**"    (Left, Right : Float_Type'Base) return Float_Type'Base;

   function Sin     (X           : Float_Type'Base) return Float_Type'Base;
   function Sin     (X, Cycle    : Float_Type'Base) return Float_Type'Base;
   function Cos     (X           : Float_Type'Base) return Float_Type'Base;
   function Cos     (X, Cycle    : Float_Type'Base) return Float_Type'Base;
   function Tan     (X           : Float_Type'Base) return Float_Type'Base;
   function Tan     (X, Cycle    : Float_Type'Base) return Float_Type'Base;
   function Cot     (X           : Float_Type'Base) return Float_Type'Base;
   function Cot     (X, Cycle    : Float_Type'Base) return Float_Type'Base;

   function Arcsin  (X           : Float_Type'Base) return Float_Type'Base;
   function Arcsin  (X, Cycle    : Float_Type'Base) return Float_Type'Base;
   function Arccos  (X           : Float_Type'Base) return Float_Type'Base;
   function Arccos  (X, Cycle    : Float_Type'Base) return Float_Type'Base;

   function Arctan
     (Y   : Float_Type'Base;
      X   : Float_Type'Base := 1.0)
     return Float_Type'Base;

   function Arctan
     (Y     : Float_Type'Base;
      X     : Float_Type'Base := 1.0;
      Cycle : Float_Type'Base)
      return  Float_Type'Base;
```

```
   function Arccot
     (X   : Float_Type'Base;
      Y   : Float_Type'Base := 1.0)
     return Float_Type'Base;

   function Arccot
     (X     : Float_Type'Base;
      Y     : Float_Type'Base := 1.0;
      Cycle : Float_Type'Base)
     return   Float_Type'Base;

   function Sinh    (X : Float_Type'Base) return Float_Type'Base;
   function Cosh    (X : Float_Type'Base) return Float_Type'Base;
   function Tanh    (X : Float_Type'Base) return Float_Type'Base;
   function Coth    (X : Float_Type'Base) return Float_Type'Base;
   function Arcsinh (X : Float_Type'Base) return Float_Type'Base;
   function Arccosh (X : Float_Type'Base) return Float_Type'Base;
   function Arctanh (X : Float_Type'Base) return Float_Type'Base;
   function Arccoth (X : Float_Type'Base) return Float_Type'Base;

end Ada.Numerics.Generic_Elementary_Functions;
```

E.1. Ejemplo de uso

```
with Ada.Numerics.Generic_Elementary_Functions;
with Text_IO;
procedure Demo is
   package Float_IO is new Text_IO.Float_IO (Float);
   package Math is new Ada.Numerics.Generic_Elementary_Functions (Float);
   use Math;
begin
   Float_IO.Put (Sqrt (9.0));
end Demo;
```

Apéndice F

Linux

Linux es el sistema operativo que vas a utilizar en el laboratorio para realizar las prácticas de esta asignatura. Llamamos **sistema operativo** a un conjunto de programas que facilitan el uso del ordenador.

En la actualidad los sistemas operativos más conocidos son MS-DOS, Windows, UNIX, Linux y Mac-OS. La diferencia fundamental entre Linux y el resto es que Linux es totalmente gratuito. Puedes conseguirlo en Internet o en revistas especializadas.

Al hablar de Linux los expertos dicen: *"No sabía si instalar Red-Hat o Suse. Al final he instalado Mandrake porque me han dicho que más fácil de instalar, aunque Debian es más completa y es la que realmente es completamente gratuita"*. ¿Qué quieren decir? ¿Acaso hay varios tipos de Linux?.

Como todos los sistemas operativos Linux consta de dos partes. El núcleo, que es el programa que comienza a ejecutarse desde que enciendo el ordenador hasta que lo apago, y el resto de programas (editores, compiladores, calculadoras, programas de juegos, etc.) que vienen con todos los sistemas operativos. Lo que ocurre es que, como Linux es gratuito y todos sus programas son gratuitos, algunas personas y empresas se están encargando de elegir los programas que más se utilizan, añadir los programas que faciliten su instalación y poner todo ésto dentro de uno o varios CD. Para facilitar su identificación, cada una de estas colecciones tiene un nombre y los más conocidos son: *Red-Hat, Suse, Mandrake* y *Debian*.

F.1. Breve historia de Linux

En 1991, con 23 años, un estudiante de informática de la Universidad de Helsinki (Finlandia) llamado *Linus Torvalds* se propone como entretenimiento hacer un sistema operativo que se comporte exactamente igual al sistema operativo UNIX, pero que funcione sobre cualquier ordenador compatible PC. Posteriormente Linus tuvo que poner como requisito mínimo que el ordenador tuviese un procesador i386, ya que los ordenadores con CPU más antiguas no facilitaban el desarrollo de un sistema operativo compatible con UNIX.

Un factor decisivo para el desarrollo y aceptación de Linux va a ser la gran expansión de Internet. Internet facilitó el trabajo en equipo de todos los que quisieron colaborar con Linus y fueron aportando todos los programas que vienen con UNIX. Linus no pretendía crear todos los programas que vienen con UNIX. Su objetivo fundamental era crear un núcleo del S.O. que fuera totalmente compatible con el de UNIX y que permitiera ejecutar todos los programas gratuitos compatibles UNIX desarrollados por la Free Software Foundation (fundada por *Richard Stallman*) que vienen con licencia GNU[1]. Esta licencia impide poner precio a los programas donados a la comunidad científica por sus propietarios (programas *libres*) y obliga a que si se escriben nuevos programas utilizando código de programas libres, estos sean también libres.

Para crear su núcleo, Linus se inspiró en Minix, una versión reducida de UNIX desarrollada por el profesor *Andy Tanenbaum* para que sus alumnos pudieran conocer y experimentar con el código de un sistema operativo real.

Linus escribió un pequeño núcleo que tenía lo necesario para leer y escribir ficheros en un disquette. Estamos a finales de Agosto de 1991 y Linus ya tiene la versión $0{,}01$. Como no era muy agradable de usar y no hacia gran cosa, no lo anunció. Le puso como nombre Linux, que es un acrónimo en inglés de “Linus UNIX” (el UNIX de Linus).

El 5 de octubre de 1991, Linus anuncia la primera versión “oficial” de Linux, la $0{,}02$. Esta versión ya podía ejecutar dos herramientas básicas de GNU: el intérprete de órdenes (`bash`) y el compilador de C (`gcc`). Linux no tenía aún nada sobre soporte a usuarios, distribuciones, documentación ni nada parecido (aún hoy la comunidad de Linux trata estos asuntos de forma secundaria; lo primero sigue siendo el desarrollo del kernel).

Linus siguió trabajando hasta que Linux llegó a ser un producto realmente útil. Dió los fuentes de Linux para que cualquiera pudiese leerlo, modificarlo y

[1] Acrónimo recursivo de la frase en inglés “GNU is Not UNIX”.

mejorarlo. Seguía siendo la versión $0{,}02$ pero ya ejecutaba muchas aplicaciones GNU (`bash`, `gcc`, `gnu-make`, `gnu-sed`, `compress`, etc.)

Tras la versión $0{,}03$, Linus salto a la versión $0{,}10$, al tiempo que más gente empezaba a participar en su desarrollo. Después de numerosas revisiones, alcanzó la versión $0{,}95$, reflejando la esperanza de tener lista muy pronto una versión estable (generalmente, la versión $1{,}0$ de los programas es la primera teóricamente completa y sin errores). Esto sucedía en marzo de 1992. Año y medio después, en diciembre del 93, nacía Linux 1.0.

Hoy Linux es ya un clónico de UNIX completo y hay muchas personas escribiendo programas para Linux. Incluso las empresas están empezando a escribir programas para Linux ya que el nivel de aceptación que ha tenido es enorme. ¿Quién iba a imaginar que este "pequeño" clónico de UNIX creado por un estudiante iba a convertirse en un estándar mundial para los ordenadores personales?.

F.2. Uso de Linux

En este apartado veremos como ejemplo una sesión normal con Linux. Veremos cómo se realiza la entrada en el sistema, el cambio de contraseña,

F.2.1. Entrada en el sistema

Unix es un sistema operativo **multiusuario**. Esto quiere decir que, a diferencia de MS-DOS o Windows-98, varias personas pueden estar trabajando simultáneamente con un mismo ordenador. Por tanto, para comenzar a trabajar con Linux debemos identificarnos (decir quienes somos y demostrarlo). Linux se presenta de la siguiente forma:

```
Debian GNU/Linux

ladyada login:_
```

Nos está diciendo que la distribución que se ha instalado en esta máquina es *Debian* y que la máquina se llama *ladyada*. Con la palabra **login** Linux está esperando a que digamos quienes somos. Vamos a entrar en Linux. Supongamos que se nos ha asignado como login la palabra *alumno200*. Deberemos escribir *alumno200* y pulsar a continuación la tecla RETURN o la tecla ENTER (en tu caso, debes utilizar el nombre de usuario que se te ha asignado).

```
Debian GNU/Linux
```

```
ladyada login: alumno200
Password:
```

Con la palabra **Password** Linux quiere que demostremos que somos realmente la persona que hemos dicho. Para comprobarlo nos pide que le digamos nuestra contraseña secreta (en el siguiente apartado veremos cómo podemos modificar nuestra contraseña secreta). Escribimos nuestra contraseña secreta (que al ser secreta Linux la lee, pero no la enseña en la pantalla). Si nos equivocamos, Linux nos dice que la contraseña no es correcta y nos pide que volvamos a decir quienes somos. Si la contraseña es correcta nos dará en pantalla un mensaje parecido al siguiente:

```
Last login: Thu Oct  5 23:08:49 2000 on tty2
Linux ladyada 2.2.5 #34 Fri Mar 17 20:31:35 WET 2000 i686 unknown

Copyright (C) 1993-1999 Software in the Public Interest, and others

Most of the programs included with the Debian GNU system are
freely redistributable; the exact distribution terms for each
program should be described in the individual files
in /usr/share/doc/*/copyright or /usr/doc/*/copyright.

Debian GNU comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.

$ _
```

! Enhorabuena ! Ya estas dentro de Linux.

F.2.2. Cambiando la palabra de paso

Para asegurar que ninguna otra persona que trabaje en el mismo ordenador pueda enviar un mail o modificar, borrar o copiar nuestra información entrado en Linux con nuestra identidad, debemos poner una buena contraseña. ¿Qué quiero decir con una buena contraseña? Una que sea difícil de adivinar a cualquiera que pretenda entrar en tu cuenta. No pongas nunca como contraseña:

- El número de tu carnet de identidad (desde que alguien lo sepa ya puede entrar en tu cuenta).
- Tu fecha de nacimiento (también está en el carnet de identidad).

- El nombre de tu novia (es fácil de saber).
- Ninguna palabra que esté en un diccionario (hay programas que prueban automáticamente con todas las palabras del diccionario hasta conseguir entrar).

Como regla general debes poner una contraseña que sea una combinación de letras, números y símbolos especiales del teclado (por ejemplo $ =, (, #, etc.). Para que te hagas una idea, fíjate algunas combinaciones buenas que podemos hacer con la palabra *Jose*, el número 52 y el símbolo #.

```
52#Jose        5#2Jose        Jo#52#se        5#Jose#2
```

Las contraseñas deben cambiarse cada cierto tiempo. Cuando nos identificamos introduciendo la contraseña, alguien puede ver algunas de las teclas que hemos pulsado. Cuantas más veces escribamos nuestra contraseña, mayor probabilidad hay de que alguien la descubra. Para cambiar nuestra contraseña tenemos que pedirselo a Linux. Para esto le decimos a Linux que queremos ejecutar la orden **passwd**.

```
$ passwd
Changing password for alumno200
(current) UNIX password:_
```

Lo primero que nos pide Linux es que le demos la contraseña que tenemos puesta hasta ahora. Lo pide como medida de seguiridad ya que podría ocurrir que nos levantemos del ordenador un momento y que algún gracioso intente aprovechar para cambiarnos la contraseña. Después de darle la contraseña correcta, Linux nos pide dos veces consecutivas la nueva contraseña (lo pide dos veces por si acaso hemos pulsado mal alguna tecla y ponemos una contraseña diferente de la que realmente queríamos poner).

```
$ passwd
Changing password for alumno200
(current) UNIX password:
Enter new UNIX password:
Retype new UNIX password:
```

A partir de este instante, la próxima vez que entremos en el sistema deberemos dar a Linux la nueva contraseña. Si nos olvidamos nuestra contraseña, debemos pedir al encargado del laboratorio que nos la cambie. Utilizando una identificación

especial, puede actualizar nuestra contraseña, aunque ni siquiera él puede consultar la que teníamos antes. Esto nos permite utilizar la misma contraseña en diferentes ordenadores y que ni siquiera los instaladores o gestores de esos ordenadores puedan conocerla.

F.2.3. Saliendo del sistema

Cuando terminamos nuestro trabajo debemos siempre decirle a Linux que nos vamos (para que Linux deje todos nuestros documentos bien guardados). Esto lo hacemos diciéndole **logout** o **exit**.

```
$ logout
```

Esta orden también evita que otras personas puedan enviar un mail o modificar, borrar o copiar nuestra información, utilizando el terminal donde estabamos nosotros.

F.2.4. Ficheros

Cuando salimos del sistema, Linux olvida los datos con los que hemos estado trabajando (dedica su memoria al resto de usuarios que continuen trabajando con él). Si queremos guardar datos en nuestra cuenta para continuar trabajando con ellos la próxima vez que entremos en Linux, debemos utilizar ficheros, también llamados archivos.

Un fichero es un conjunto de información al que se le ha asignado un nombre. Por ejemplo, un mensaje de correo es un fichero y un programa que puede ser ejecutado también. El nombre de fichero lo ponemos nosotros y, en Linux, puede contener mayúsculas y minúsculas. Por ejemplo, *apuntes.txt* y *Apuntes.txt* son dos ficheros diferentes. Además, el nombre puede ser tan largo como queramos (el tamaño máximo son 256 caracteres, pero no resulta nada cómodo trabajar con nombres de ficheros tan largos).

Aunque a Linux le da igual el nombre que demos a nuestros ficheros, algunos programas tienen unas reglas estrictas. Por ejemplo, el compilador de Ada nos obliga a que los ficheros que contienen programas Ada tengan un nombre que termine en **.adb**. Algunos editores nos obligan a que los ficheros que se editan con ellos terminen en *.txt* o *.doc*.

F.2.5. Directorios

Para organizar nuestro trabajo necesitamos algún mecanismo que nos permita agrupar nuestros ficheros según su contenido. Para ello utilizamos los directorios. Un directorio puede ser considerado como una "carpeta" que contiene muchos ficheros diferentes. Igual que los ficheros, los directorios también tienen nombre con el que los podemos identificar.

Las órdenes básicas para trabajar con directorios son:

1. **pwd**: Nos dice en qué directorio estamos trabajando.
2. **mkdir** (*Make Directory*): Crea un directorio.
3. **rmdir** (*Remove Directory*): Borra un directorio. Linux sólo permite borrar un directorio cuando está vacio.
4. **cd** (*Change Directory*): Cambiar de directorio. Si queremos subir al directorio anterior debemos poner `cd ..`. Si llamamos a *cd* sin decirle ningún nombre de directorio nos lleva de nuevo al directorio inicial (el directorio en que nos deja Linux cada vez que entramos en nuestra cuenta). Esto también ocurre si ponemos `cd ~`.
5. **ls** Ver el contenido del directorio.

Los directorios mantienen una estructura de árbol. Esto quiere decir que los directorios pueden, a su vez, contener otros directorios. Por ejemplo, supongamos que queremos crear la siguiente estructura de directorios en nuestra cuenta:

```
~/Documentos
 /Practicas_Ada/Cuestionario
               /Calculadora
               /Matrices
               /Juego
 /Utilidades
```

La secuencia de órdenes que tenemos que dar es la siguiente:

```
$ mkdir Documentos
$ mkdir Practicas_Ada
$ cd Practicas_Ada
$ mkdir Cuestionario
$ mkdir Calculadora
$ mkdir Matrices
```

```
$ mkdir Juego
$ cd ..
$ mkdir Utilidades
```

Para ver la estructura de directorios podemos utilizar la orden **du** (*Disk Usage*). Por ejemplo:

```
$ du
1       ./Documentos
1       ./Practicas_Ada/Cuestionario
1       ./Practicas_Ada/Calculadora
1       ./Practicas_Ada/Matrices
1       ./Practicas_Ada/Juego
5       ./Practicas_Ada
1       ./Utilidades
12      .
```

Ejercicio: Realiza la secuencia de órdenes que consideres necesaria para borrar todos los directorios que acabamos de crear.

Ahora ya sabes cómo moverte por los directorios, pero el simple movimiento por el árbol de directorios es poco útil. Necesitamos alguna forma de ver el nombre de todos los ficheros que contiene cada directorio. Para esto utilizamos la orden **ls**. Por ejemplo, si queremos ver todos los ficheros que hay en la raiz del disco duro de nuestro ordenador hacemos:

```
$ ls /
1     boot   etc     initrd      mnt   sbin  var
amnt  cdrom  floppy  lib         proc  tmp   var.old
bin   dev    home    lost+found  root  usr   vmlinuz
```

Esto solamente nos dice el nombre de los ficheros y directorios, pero no es fácil diferenciar cual es cada uno. Si queremos diferenciarlos lo pedimos de la siguiente forma:

```
$ ls -F /
1      boot/   etc/     initrd/      mnt/   sbin/  var@
amnt/  cdrom/  floppy/  lib/         proc/  tmp@   var.old/
bin/   dev/    home/    lost+found/  root/  usr/   vmlinuz
```

Linux nos marca los directorios añadiendo el carácter “/” al final de su nombre. También nos marca con el carácter “” los enlaces simbólicos a otros ficheros

(los enlaces son nombres de ficheros cuyo contenido está en otro fichero y, probablemente también en otro directorio).

La orden "`ls -F`" puede también añadir al final el carácter "*". Esto indica que es un fichero ejecutable. Si "`ls -F`" no añade nada, entonces es un fichero normal, es decir no es ni un directorio ni un ejecutable.

Como hemos visto, generalmente cada orden UNIX puede tomar una serie de opciones definidas en forma de argumentos. Estas opciones, también llamadas indicadores, usualmente comienzan con el carácter "-", como vimos antes con `ls -F`.

También podemos movernos directamente a un directorio dando su nombre completo (por ejemplo, "`cd /usr/bin`"). A veces podemos encontrarnos con el mensaje de error *"Permission denied"*. Esto simplemente es debido a cuestiones de seguridad del UNIX (estamos intentando acceder a un directorio o fichero que es de otro usuario y no tenemos permiso para leerlo). Para poder movernos o ver el contenido de un directorio, el propietario debe haberte permitido hacerlo.

F.2.6. Operaciones con ficheros

Linux nos permite crear ficheros (mediante el editor, el compilador, o el programa que vayamos a utilizar), copiarlos (con la orden **cp**), moverlos a otro directorio (con la orden **mv**) y borrarlos (mediante la orden **rm**, que viene de la palabra *remove*).

```
cp fichero_antiguo  fichero_nuevo

mv nombre_antiguo   nombre_nuevo
mv fichero          directorio

rm fichero
```

Debes tener en cuenta que si el nombre del fichero nuevo ya existe, `mv` y `cp` sobreescriben sin consultar. Además, como ves hay dos formas de mover ficheros. La primera formas se utiliza para cambiar el nombre de un fichero y la segunda se utiliza para mover un fichero a otro directorio. Ten mucho cuidado cuando muevas un fichero a otro directorio: puede que ya exista en ese directorio un fichero con el mismo nombre y, por tanto, Linux sobreescribe el fichero (con lo que pierdes para siempre el que había antes).

Al borrar ficheros debemos tener mucho cuidado. Los ficheros que borres no puedes recuperrarlos nunca más (son borrados definitivamente). Si quieres decirle a Linux que te pregunte si realmente quieres borrar cada uno de los ficheros (para tener una confirmación adicional) debes utilizar el indicador **-i**. Por ejemplo, supongamos que quieres borrar todos los ficheros de un directorio. Para hacerlo debes utilizar el carácter comodín (que significa *todos los ficheros*), pero además deberías utilizar el indicador **-i** para que Linux te pregunte si realmente quieres borrar cada fichero.

```
$ rm -i *
rm: remove ‘ejemplo.adb’? y
rm: remove ‘prueba.adb’? n
```

En este ejemplo he dicho a Linux que quiero borrar *ejemplo.adb* (porque he respondido con la **y** de *yes*) y no quiero borrar *prueba.adb* (porque he respondido con la **n** de *no*).

F.2.7. Caracteres comodín

Una característica importante de la mayoría de los interpretes de órdenes de Unix (también en MS-DOS y Windows) es la capacidad para referirse a más de un fichero usando caracteres especiales llamados *comodines*. Los principales comodines son:

- El carácter asterisco (*). Significa cualquier secuencia de caracteres. Por ejemplo, como todos los programas Ada terminan con la extensión *.adb*, para ver todos los ficheros Ada que tengo en un directorio tengo que hacer:

  ```
  $ ls *.adb
  ```

- El carácter de cierre de interrogación (?). Significa cualquier carácter individual. Se utiliza cuando tengo varios ficheros que sólo se diferencian en una o varias letras. Por ejemplo, si tengo las prácticas en varios ficheros que se llaman *practica_1, practica_2*, etc. puedo verlos haciendo:

  ```
  $ ls practica_?.adb
  ```

Como ves, los caracteres comodín te permiten referirte a más de un fichero a la vez. Gracias al uso de estos caracteres podemos copiar, mover o borrar muchos ficheros de una vez. Por ejemplo:

```
$ cp /bin/s* .
$ rm *
```

Primero hemos copiado todos los ficheros del directorio `/bin` que comiencen por la letra “s” en el directorio actual (siempre que quiero hacer referencia al directorio donde estoy utilizo un punto). A continuación, como sólo era una prueba, borro todos los ficheros que tengo en mi directorio. Si en el directorio había otros ficheros, también fueron borrados. Por esta razón no olvidemos usar el indicador **-i** cuando no estemos seguros de lo que se podría borrar.

F.2.8. Obteniendo ayuda

Para trabajar con Linux no es necesario que nos sepamos de memoria todas sus órdenes. Podemos preguntarle a Linux cómo se hace cualquier cosa. Para ello debes utilizar el manual de Linux (**man**). Veamos cómo se utiliza.

Supongamos que quiero hacer algo con un directorio (por ejemploi, he olvidado en qué directorio me encuentro) y no sé cómo se hace. Lo primero que hago es preguntarle a Linux que me enseñe todo lo que se puede hacer con directorios. Esto se pregunta mediante la orden `man -k` seguido de la palabra que quiero buscar. Por ejemplo:

```
$ man -k directory | less
```

Con esta orden le estamos diciendo a Linux que busque todas las órdenes que en su descripción tengan la palabra *“directory”*, y con `| less` le estamos diciendo que si se llena la pantalla nos deje avanzar y retroceder por la pantalla con los cursores. Cuando queramos terminar pulsaremos la tecla **q**. Supongamos que al ejecutar esta petición Linux nos escribe en pantalla lo siguiente:

```
chdir (2)        - change working directory
dir (1)          - list directory contents
dirname (1)      - strip non-directory suffix from file name
getcwd (3)       - Get current working directory
getdents (2)     - get directory entries
getwd (3)        - Get current working directory
install-info (8) - create or update entry in Info directory
ls (1)           - list directory contents
mkdir (2)        - create a directory
mknod (2)        - create a directory or special or ordinary file
opendir (3)      - open a directory
pwd (1)          - print name of current/working directory
readdir (2)      - read directory entry
readdir (3)      - read a directory
rmdir (2)        - delete a directory
vdir (1)         - list directory contents
lndir (1x)       - create a shadow directory of symbolic links to a directory
```

```
Cwd (3pm)        - get pathname of current working directory
DirHandle (3pm)  - supply object methods for directory handles
find (1)         - search for files in a directory hierarchy
```

Linux nos está diciendo las órdenes que ha encontrado y que están relacionadas con los directorios. Además, entre paréntesis, nos dice también la sección del manual en la que está su documentación. Revisamos la lista y vemos que la orden que estabamos buscando es *pwd* y está en la sección 1 del manual. Para pedirle a Linux la documentación completa de *pwd* le digo `man seccion palabra`. Por ejemplo:

```
$ man 1 pwd
PWD(1)                          FSF                           PWD(1)

NAME
       pwd - print name of current/working directory

SYNOPSIS
       pwd [OPTION]

DESCRIPTION
       Print  the full filename of the current working directory.

       --help       display this help and exit

       --version    output version information and exit

REPORTING BUGS
       Report bugs to <bug-sh-utils@gnu.org>.

SEE ALSO
       The full documentation for pwd is maintained as a  Texinfo
       manual.   If  the  info and  pwd   programs  are  properly
       installed at your site, the command

              info pwd

       should give you access to the complete manual.

COPYRIGHT
       Copyright C 1999 Free Software Foundation, Inc.
       This is free software; see the source for  copying  condi-
       tions.  There is NO warranty; not even for MERCHANTABILITY
       or FITNESS FOR A PARTICULAR PURPOSE.
```

Las páginas de manual de Linux no son un tutorial de uso. Se utilizan como recordatorio para evitarnos recordar todos los indicadores y órdenes de Linux. Son

una gran fuente de información que nos permiten refrescar la memoria cuando olvidamos algo de una orden que necesitamos utilizar.

Ejercicio Pruebe `man` con las órdenes que ya hemos utilizado hasta ahora.

Apéndice G

Emacs

Emacs es el editor de texto que vamos a utilizar para escribir nuestros programas Ada. En este apéndice conocerás todo lo necesario para que hagas tus prácticas con emacs.

G.1. Uso básico de emacs

Antes de comenzar a trabajar con *emacs* debes saber que si en algún momento te trabas con el teclado, pulsa `Ctrl-g` (pulsa la tecla `Ctrl` y, sin soltarla, pulsa la tecla g). De esta forma *emacs* anula la orden que ibas a darle y te permite seguir trabajando con tu documento. Visto esto, comencemos a utilizar *emacs*.

La forma más sencilla de crear un fichero de texto es llamar a nuestro editor de textos, en este caso *emacs*, dándole el nombre del fichero. Por ejemplo:

```
$ emacs hola.adb
```

Como resultado, verás la pantalla inicial de emacs, que es la siguiente:

```
Buffers Files Tools Edit Search Ada Help
_
```

```
-----Emacs: hola.adb          (Ada)--L1--All-----------------------------
Loading ada-mode...done
```

En la parte alta de la pantalla hay un menú de opciones. Este menú sólo se puede usar cuando utilizamos *emacs* desde un escritorio gráfico. Si quieres utilizar una versión simplificada de este menú, pulsa la tecla `F10` y elige las opciones que esta tecla.

La mayor parte de la pantalla está vacia. Es tu zona de trabajo. Para escribir texto, simplemente utiliza el teclado (incluyendo las teclas para moverte arriba, abajo, izquierda, derecha, página adelante y página atrás). Las dos últimas líneas son especialmente interesantes. La penúltima línea (la que tiene una cadena larga de guiones) se denomina línea de modo. En esta línea *emacs* te dice el nombre del fichero que estás editando, el tipo de fichero (texto, programa Ada, página HTML, etc.), la línea que estás editando y una referencia porcentual de tu posición dentro del documento completo (por ejemplo, 5 %).

Cuando realizas algún cambio en el fichero *emacs* te avisa colocando dos asteriscos en la línea de modo. Por ejemplo:

```
--**-Emacs: hola.adb          (Ada)--L1--All-----------------------------
```

La línea inmediatamente inferior a la línea de modo se denomina minibuffer, o

a veces el área de eco. *Emacs* usa el minibuffer para enviar mensajes al usuario y, cuando es necesario, para leer información que introduce el usuario. En el ejemplo vemos que *emacs* nos está diciendo que, como el fichero termina con la extensión de los ficheros Ada (.adb), acaba de cargar en memoria todo lo necesario para trabajar con programas Ada. Desde que comencemos a escribir algo, *emacs* quita este mensaje y deja el minibuffer vacio.

Si en el ordenador que utilizas para editar tus programas tienes algún problema para moverte con las teclas arriba, abajo, izquierda, derecha (por ejemplo, hay alguna tecla estropeada que no funciona bien), *emacs* tiene reservada una combinación de teclas cuyo resultado es completamente equivalente. Para utilizar esta combinación de teclas debes presionar la tecla de control (`Ctrl`) y sin soltarla pulsar una letra del teclado. Por ejemplo, para conseguir que el cursor se mueva hacia la derecha deberás pulsar `Ctrl-f` (*emacs* eligió la *f* porque es la primera letra de la palabra *forward*). En la siguiente tabla tienes la secuencia equivalente a cada una de las teclas de movimiento del cursor.

`Ctrl-f`	Avanza un carácter.
`Ctrl-b`	Retrocede un carácter.
`Ctrl-n`	Avanza a la siguiente línea.
`Ctrl-p`	Retrocede a la línea anterior.
`Ctrl-v`	Avanza a la siguiente página.
`Esc-v`	Retrocede a la página anterior.

Cuadro G.1: Secuencias equivalentes para movimiento del cursor

NOTA: A diferencia de las secuencias de control vistas hasta ahora (todas las secuencias de control de *emacs* que comienzas pulsando la tecla `Ctrl`), en las secuencias que comienzan con la tecla `Esc` no es necesario mantenerla pulsada todo el rato. Por ejemplo, `Esc-v` significa: Pulsa la tecla `Esc`, suéltala y pulsa la tecla `v`.

Emacs también tiene otras combinaciones de teclas que te serán muy utiles para escribir tus programas. Las principales son:

Cuando editas un fichero con *emacs*, lo primero que hace *emacs* es sacar una copia del fichero y guardarla en su memoria. Por tanto, cuando realizas modificaciones en el fichero, realmente sólo estás modificando la copia que tiene *emacs* (no el fichero). La combinación de teclas `Ctrl-x-s` te permite decidir el instante exacto en que quieres que la copia que tienes en memoria se actualice en el

disco. Esta combinación de teclas quiere decir: pulsa la tecla `Ctrl`; sin soltarla pulsa la tecla x, suelta la x y, sin soltar `Ctrl`, pulsa ahora la tecla s. Aunque al leerlo puede parecer complicado, en realidad es bastante sencillo porque las teclas x y s están en el teclado juntas, por lo que toda la combinación se puede hacer fácilmente con una mano.

`Ctrl-a`	Ir al principio de la línea.
`Ctrl-e`	Ir al final de la línea.
`Ctrl-l`	Redibujar la pantalla y coloca la línea actual en el centro.
`Ctrl-d`	Borrar el carácter actual.
Retroceso	Borra el carácter anterior.
`Ctrl-k`	Borrar el texto hasta el final de la línea.
`Ctrl-x-s`	Guardar el fichero.
`Ctrl-x-c`	Salir de emacs.

Cuadro G.2: Otras combinaciones de teclas de emacs

Como ejercicio, escribe en pantalla nuestro primer ejemplo con Ada hasta que tengas en pantalla el siguiente resultado.

```
Buffers Files Tools Edit Search Ada Help
with Text_IO;
procedure Hola is
begin
   Text_IO.Put_Line (‘‘Hola Ada’’);
end Hola;
```

```
-----Emacs: hola.adb          (Ada)--L4--All-------------------------------
Wrote hola.adb
```

G.1.1. Uso de GNAT desde emacs

Para compilar tus programas con GNAT se utiliza la siguiente combinación de teclas: `Ctrl-c-c`. En el minibuffer *emacs* te preguntará cómo quieres compilar el programa y deberás decirle que utilice *gnatmake*. Por ejemplo, para compilar el ejemplo anterior pulsas `Ctrl-c-c` y en el minibuffer verás:

```
Compile command: make -k
```

Como queremos utilizar *gnatmake* (en vez de *make*), borramos la orden de *make* y escribimos:

```
Compile command: gnatmake -gnatg hola.adb
```

Al pulsar RETURN (o ENTER) *emacs* divide la pantalla en dos, con lo que verás algo parecido a:

```
Buffers Files Tools Edit Search Ada Help
with Text_IO;
procedure Hola is
begin
   Text_IO.Put_Line (‘‘Hola Ada’’);
end Hola;




-----Emacs: hola.adb          (Ada)--L4--All-----------------------------
-cd /tmp/
gnatmake -gnatg hola.adb
gcc -c -gnatg hola.adb
gnatbind -x hola.ali
gnatlink hola.ali

Compilation finished at Sun Oct  8 00:23:27



--**-Emacs: *compilation*      (Compilation:exit [0])--L1--All------------
(No files need saving)
```

Si has cometido errores, entonces, en la ventana de abajo GNAT te dirá todos los errores. Para que *emacs* se coloque automáticamente en cada uno de los errores debes utilizar la siguiente combinación de teclas: `Ctrl-x ‘SPC`. Esto quiere decir, pulsas `Ctrl-x`, sueltas ambas teclas y a continuación pulsas el acento que está en la tecla justo al lado de la tecla P (‘) seguido de la barra de espacio (SPC).

Cuando termines de corregir los errores debes pulsar `Ctrl-x 1` para volver a trabajar con una única ventana.

Para ejecutar tu programa debes pulsar `Ctrl-z`. Con ésto dejas “*dormido*” el editor y en la pantalla verás lo siguiente:

```
[1]+  Stopped                 emacs hola.adb
$ _
```

Acabas de salir temporalmente al sistema operativo (*Linux*) dejando dormido el editor (*emacs* sigue estando en la memoria del ordenador, pero no puede utilizarse). Para ejecutar el programa que acabamos de hacer, simplemente decimos

el nombre del directorio donde está el programa (en este caso está en el directorio actual, que se indica con un punto) seguido del nombre del programa.

```
$ ./hola
```

Al ejecutar el programa verás que Ada escribe en pantalla el mensaje *"Hola Ada"* (es lo único que le has pedido que haga). Para volver al editor debes escribir `fg` (estas pidiendo a Linux que continúe ejecutando el programa que acabas de dejar dormido).

```
$ fg
```

En las dos tablas siguientes se resumen todas las combinaciones de teclas que hemos utilizado para compilar el programa.

`Ctrl-c-c`	Compilar.
`Ctrl-x 'SPC`	Ir al siguiente error.
`Ctrl-x 1`	Volver a trabajar con una ventana.
`Ctrl-z`	Dormir el editor.

Cuadro G.3: Combinaciones de teclas de *emacs* para compilar.

`fg`	Continuar la ejecución del programa dormido.

Cuadro G.4: Órden de Linux para volver al editor.

Un error común es dejar dormido a *emacs* y, en lugar de volver al editor pulsando `Ctrl-z`, volver a pedir a *Linux* que ejecute otra vez a *emacs* con el mismo fichero (`emacs hola.adb`). Esto es realmente peligroso porque cada vez que hacemos esta secuencia de pasos creamos una nueva copia de nuestro fichero en memoria. En algún momento podemos perder las últimas modificaciones del fichero, por ejemplo, salido de la copia que acabamos de actualizar con `Ctrl-x-c`, entrando en una copia anterior con `fg` y volviendo a dar la orden de grabar `Ctrl-x-s`. Además, al irse llenando la memoria del ordenador con muchas copias, en cualquier momento *Linux* se puede quejar diciendo que no tiene memoria. En resumen, cuando salimos de *emacs* pulsando `Ctrl-z`, siempre debemos volver al editor con la orden `fg`.

G.1.2. Cortar, pegar, destruir y tirar

Emacs, como editor de textos, te permite cortar y pegar bloques de texto. A fin de hacer esto, necesita que le indiques dónde comienza y termina el bloque de texto que quieres cortar, copiar o borrar. Para marcar el principio del bloque debes teclear `Ctrl-SPC` (la tecla `Ctrl` seguido de la barra de espacio). En el minibuffer verás el mensaje *"Mark set"* (Marca establecida). A diferencia de otros editores, no verás ninguna marca especial en el documento; tú sabes que la has puesto y eso es lo que importa.

La marca de final de bloque es simplemente el cursor. Cuando dejes de mover el cursor y des la orden de cortar (`Ctrl-w`), *emacs* corta todo el texto desde donde pusiste la marca hasta donde esté el cursor en ese instante. Cuando quieras que *emacs* inserte una copia del trozo que acabas de copiar simplemente pulsas `Ctrl-y`.

Otra forma de cortar un bloque de código es hacerlo línea a línea. Para esto te colocas al principio del trozo que quieres cortar y pulsas `Ctrl-k` repetidamente. Cuando acabes de cortar el trozo, te colocas donde quieras insertar una copia y pulsas `Ctrl-y` (exactamente igual que antes).

Si lo que quieres es sacar una copia de un trozo de texto, pero sin cortarlo, colocas la marca de principio de bloque (`Ctrl-SPC`), te mueves hasta el final del bloque y tecleas `Esc-w`. Para insertar copias de este trozo utilizas también `Ctrl-y` (como en los casos anteriores).

`Ctrl-SPC`	Marcar el principio del bloque.
`Ctrl-w`	Cortar desde la marca hasta la posición actual.
`Esc-w`	Copiar desde la marca hasta la posición actual.
`Ctrl-k`	Cortar desde hasta el final de la línea.
`Ctrl-y`	Pegar el último bloque borrado.
`Esc-x query-replace`	Busca y pregunta antes de sustituir.
`Esc-x replace-string`	Busca y sustitye (sin preguntar).

Cuadro G.5: Secuencias especiales de composición de texto.

G.1.3. Buscar y reemplazar

Podemos buscar el texto avanzando hacia adelante pulsa `Ctrl-s` (busca la palabra desde la posición donde está el cursor hasta el final del documento) y para

buscar hacia atrás pulsa `Ctrl-r`.

Por ejemplo, supongamos que quieres buscar la cadena `pulga` en el siguiente texto.

```
Yo estaba temeroso de quedarnos sin gasolina, cuando mi
pasajero dijo

 ‘‘!Me escuece el brazo! Hay una pulga!’’.
```

Suponiendo que tienes el cursor justo al principio del documento (o por lo menos en algún carácter que esté antes de la primera letra de la palabra que buscas, `pulga`, y tecleas `Ctrl-s`. En el minibuffer *emacs* pondrá el mensaje:

```
I-search:
```

Emacs te está preguntando qué quieres buscar. Tan pronto como escribas la “p”, verás que *Emacs* salta a la primera “p” que haya en el documento (en este ejemplo, la primera “p” es de la palabra `pasajero`). Ahora escribe la “u” de `pulga`, y *emacs* saltará sobre `pulga`, sin haber tenido que escribir la palabra entera. Cuando has encontrado la palabra, puedes abandonar la búsqueda pulsando ENTER, RETURN o cualquiera de las teclas normales de movimiento del cursor.

Si encuentras la palabra, pero no es exactamente la que buscabas (porque la misma palabra está varias veces en tu documento), entonces pulsa `Ctrl-s` de nuevo y *emacs* continuará la búsqueda y colocará el cursor en la siguiente que encuentre.

Emacs también permite sustituir una palabra por otra. Para hacerlo teclea `Esc-x query-replace` y ENTER. Si quieres que *emacs* busque y sustituya una palabra por otra sin preguntar, utiliza `Esc-x replace-string`.

`Ctrl-s`	Buscar hacia adelante.
`Ctrl-r`	Buscar hacia atrás.
`Esc-x query-replace`	Busca y pregunta antes de sustituir.
`Esc-x replace-string`	Busca y sustitye (sin preguntar).

Cuadro G.6: Secuencias especiales de búsqueda y sustitución

G.1.4. La ayuda de emacs

Emacs tiene muchas facilidades de ayuda, tan extensas de hecho, que sólo las comentaremos brevemente. A las facilidades de ayuda más básicas se accede tecleando `Ctrl-h` y luego una única letra. Por ejemplo:

- `Ctrl-h k`: muestra la ayuda sobre una tecla (te pide que presiones una tecla, y entonces te dice lo que hace esa tecla).
- `Ctrl-h t`: abre un breve manual en inglés sobre *emacs*.
- `Ctrl-h Ctrl-h`: te da ayuda sobre la ayuda.
- `Ctrl-h w`: te dice qué combinación de teclas está asociada a una determinada orden de *emacs*.
- `Ctrl-h f`: te dice qué hace una determinada orden de *emacs*.
- `Ctrl-h a`: te dice todas las órdenes de *emacs* que tienen una determinada palabra (cómo la orden *apropos* de Linux).

Siempre que damos una orden a *emacs*, intenta completar el nombre de la órden. Por tanto, cuando busques algo no necesitas saber exactamente cómo se llama en *emacs*. Si piensas que puedes adivinar la palabra con la que podría comenzar, tecleala (o al menos su primera letra) y pulsa `Tab` para ver si se completa. Si no, vuelve atrás e intente otra cosa. Lo mismo ocurre con los nombres de ficheros: aún cuando no puedas recordar del todo el nombre que diste a un fichero al que no has accedido en tres meses, puedes probar y usar la terminación automática para averiguar si estás en lo cierto. Usa la terminación automática como una forma de preguntar, y no sólo como una forma de escribir menos.

Apéndice H

FTP

FTP ("File Transfer Protocol") es el conjunto de programas que se usa en Internet para transferir ficheros entre sistemas. La mayoría de los sistemas UNIX, VMS y MS-DOS de Internet tienen un programa llamado *ftp* que se usa para transferir estos ficheros. Además, nosotros utilizaremos en el laboratorio para copiar nuestros ficheros en un disquette y llevarnos las prácticas a casa y viceversa.

H.1. Uso de ftp

Cuando comienza la ejecución de `ftp` deberás ver algo como:

```
Connected to ladyada
220 ladyada FTPD ready at 15 Dec 1992 08:20:42 EDT
Name    :
Password:
230 Guest login ok, access restrictions apply.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp>
```

Como es lógico, lo primero que hace *ftp* es pedirte tu nombre de usuario y tu contraseña (para ver si eres realmente quien dices ser y permitirte leer y escribir los documentos que hay en la cuenta). Después de dar nuestra clave y contraseña, ya estamos dentro. Nuestro prompt es `ftp>`, y el programa `ftp` está a la espera de órdenes. Para ver el contenido de nuestro directorio utilizamos *ls* o *dir*. También podemos utilizar *dir*. La diferencia fundamental es que *ls* produce un listado corto

(solamente los nombres de los ficheros) y *dir* genera un listado más largo (con el tamaño de los ficheros, fechas de modificación, etc.).

Antes de moverte de un directorio a otro debes pensar dónde quieres cambiar de directorio: en la máquina donde estás escribiendo (*lcd*) o en la máquina a la te has conectado, que en este caso es *ladyada* (*cd*).

Con la orden *help* puedes obtener ayuda de cualquier orden de *ftp*. Por ejemplo, *help cd*.

H.1.1. Traer ficheros

Antes de traer ficheros, debes determinar el tipo de fichero que vas a transferir. En lo que concierne al FTP, los ficheros van en dos formatos: binario y texto. Los ficheros que vas a transferir en el laboratorio son los fuentes de tus prácticas (ficheros ASCII, es decir, de texto). Por tanto, lo primero que debes hacer, antes de traerte ficheros, es avisar a *ftp* que son ficheros ASCII.

```
ftp> ascii
200 Type set to A.
ftp>
```

Ahora ya esta listo para traerte el fichero. Para hacerlo utiliza la orden *get* <*nombre-remoto*> <*nombre-local*>. El nombre remoto es el nombre del fichero en la máquina a la que te has conectado con ftp (en nuestro caso *ladyada*) y el nombre local es el nombre que quieres que tenga el fichero en la máquina dondes estás trabajando (o en el disquete). Si no das ningún nombre local, *ftp* presupone que quieres que tenga exactamente el mismo nombre que en la máquina remota. Por ejemplo, supongamos que quieres traerte el fichero *hola.adb*

```
ftp> get hola.adb
200 PORT command successful.
150 ASCII data connection for README (128.84.181.1,4527) (1433 bytes).
226 ASCII Transfer complete.
local: hola.adb remote: hola.adb
1493 bytes received in 0.03 seconds (49 Kbytes/s)
ftp>
```

Si quieres traer varios ficheros con una única orden, utiliza la orden *mget*. Con esta orden puedes utilizar los caracteres comodín * y ? (igual que en Linux y MS-DOS). Cada vez que vaya a traer un fichero te pedirá que le confirmes si de verdad quieres traerlo o no (ya que al utilizar comodines puede que estes intentando traer

más ficheros de los que realmente querias traer). Si quieres activar o desactivar esta confirmación debes utilizar la orden *prompt*.

H.1.2. Poner ficheros

Cuando lo que quieres es poner ficheros en el directorio remoto en vez traértelos, deberás utilizar la orden *put*. De la misma manera que con *get*, también puedes utilizar *mput* y *prompt*.

H.1.3. Salir de FTP

Para terminar una sesión FTP, sólo tienes que usar la orden *quit*.

Apéndice I

GNU Free Documentation License

Version 1.1, March 2000

Preamble

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of

subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

I.1. Applicability and Definitions

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

I.2. Verbatim Copying

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

I.3. Copying in Quantity

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes lim-

ited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

I.4. Modifications

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with

at least five of the principal authors of the Document (all of its principal authors, if it has less than five).

- State on the Title page the name of the publisher of the Modified Version, as the publisher.
- Preserve all the copyright notices of the Document.
- Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- Include an unaltered copy of this License.
- Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.

- Do not retitle any existing section as “Endorsements” or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version’s license notice. These titles must be distinct from any other section titles.

You may add a section entitled “Endorsements”, provided it contains nothing but endorsements of your Modified Version by various parties – for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

I.5. Combining Documents

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number.

Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled “History” in the various original documents, forming one section entitled “History”; likewise combine any sections entitled “Acknowledgements”, and any sections entitled “Dedications”. You must delete all sections entitled “Endorsements.”

I.6. Collections of Documents

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

I.7. Aggregation With Independent Works

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an “aggregate”, and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document’s Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

I.8. Translation

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

I.9. Termination

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

I.10. Future Revisions of This License

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License .ºr any later version.ªpplies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright © YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST. A copy of the license is included in the section entitled “GNU Free Documentation License”.

If you have no Invariant Sections, write “with no Invariant Sections” instead of saying which ones are invariant. If you have no Front-Cover Texts, write “no Front-Cover Texts” instead of “Front-Cover Texts being LIST”; likewise for Back-Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.