ADA 95

Manual de programación

9 – Abril - 2002

Sistemas Informáticos de Tiempo Real
Francisco Pastor Gomis

1. Introducción

1.1 Historia

El lenguaje de programación Ada se desarrolló como resultado de un concurso propuesto por el Departamento de Defensa de Estados Unidos. Este surgió como consecuencia del creciente coste en la realización de sistemas en el área del software empotrado el cual aumentó rápidamente. Posteriormente el coste se disparó, no solamente en las fases de diseño e implementación de nuevos sistemas, sino también en la fase de mantenimiento de sistemas existentes.

Se llegó a la conclusión que un factor importante causante de este elevado coste era la falta de estandarización. De echo, se llegaron a contabilizar cerca de 350 lenguajes usados de forma activa en el área de sistemas empotrados. Esto llevó a la decisión de buscar un único lenguaje de programación estándar.

Como primer paso, en 1976 se desarrolló un conjunto de requerimientos que debería cumplir un lenguaje de programación estándar para tiempo real en forma documento al que se llamó Tinman. Se llevó a cabo una evaluación de los lenguajes de programación existentes pero ninguno cumplía totalmente estos requerimientos. Los lenguajes Pascal, PL/1 y Algol 68 se consideraron que eran una buena base para conseguir el nuevo estándar por su uso extendido y buena estructuración.

La segunda fase comenzó en enero de 1977 con la revisión del documento Tinman para conseguir un conjunto de requerimientos de lenguaje más detallado. Se obtuvo un nuevo documento llamado Ironman. Se invitó a una serie de contratistas a diseñar el nuevo lenguaje basado en Pascal, PL/1 o Algol 68. Se recibieron 17 propuestas de las que se seleccionaron 4 para que completaran el diseño final. Los 4 lenguajes estaban basados en el Pascal y correspondían a las compañías Softech, Intemetrics, ambas de Massachussets, SRI International de California y Honeywell afiliada a Cii Honeywell Bull de Francia. Hacia marzo de 1978 los candidatos se redujeron a Intemetrics y Cii Honeywell Bul.

En la tercera fase, los diseños de los dos finalistas se revisaron para conseguir un nuevo documento de requerimientos llamado Steelman. Este se publicó en marzo de 1979 y, finalmente en mayo de 1979 se selección el lenguaje de la compañía francesa que se convertiría en el lenguaje Ada.

La cuarta y última fase consistió en un ejercicio intensivo de Test y Evaluación que llevaron a varios cambios en el lenguaje para finalmente, en Julio de 1980, publicarse el manual de referencia de Ada.

1.2 Donde y como se estandarizó Ada

Ada se estandarizó originalmente por el comité ANSI en 1983 (ISO publicó un estándar equivalente en 1987). Ada se revisó posteriormente para añadir nuevas funcionalidades. Esta revisión se llamó ‘Ada 9X’ o ‘Ada 95’. Los proveedores de Ada actualizaron posteriormente sus compiladores para añadirle las nuevas características de Ada 95. Este documento cubre las funcionalidades de Ada 95, pero no se distingue cuales son nuevas en Ada 95.

Ada se definió oficialmente en el manual de referencia del lenguaje (LRM). El LRM está disponible completamente en línea en un documento con formato de hipertexto. No obstante, el LRM no pretende ser un tutorial y puede resultar difícil de comprender si no se está familiarizado

con Ada. En este documento se harán referencias a menudo al LRM para que se pueda consultar si se está interesado en conocer con mayor detalle un tema determinado.

Ada no fue diseñado por un comité. El diseño original de Ada fue el ganador de un concurso del diseño del lenguaje; el equipo ganador fue liderado por Jean Ichbiah. La revisión de 1995 (Ada 95) se desarrolló por un pequeño equipo liderado por Tucker Taft. En ambos casos, el diseño sufrió un periodo de comentarios públicos donde los diseñadores respondían a los comentarios recibidos.

Existe en la Web un nodo con gran cantidad de información sobre Ada. Este es:

`www.adahome.com`

En este existe un tutorial llamado Lovelace bastante didáctico sobre Ada en

`http://www.adahome.com/Tutorials/Lovelace/lovelace.htm`

También se puede encontrar el LRM de Ada95 en hipertexto:

`http://www.adahome.com/rm95`

1.3 ¿Cuales son las posibilidades de Ada?

1. Se pueden definir paquetes (módulos) de tipos, objetos y operaciones relacionados.

2. Los paquetes y los tipos pueden ser genéricos (parametrizados a través de una plantilla) para facilitar la creación de componentes reusables.

3. Los errores se pueden señalizar como excepciones y manejarlos explícitamente. Muchos errores graves (tal como desbordamiento en los cálculos e índices incorrectos en las tablas) son capturados automáticamente y manejados a través del mecanismo de excepciones, mejorando la fiabilidad de los programas.

4. Se pueden crear tareas (múltiples hilos concurrentes) que se comuniquen entre ellas. Esto ofrece unas posibilidades que no están soportadas de manera estándar por muchos otros lenguajes.

5. La representación de los datos se puede controlar de forma precisa para soportar la programación de distintos sistemas.

6. Se incluye una librería predefinida; esta ofrece Entrada / Salida, manejo de cadenas, funciones numéricas, un interfaz de línea de comandos, un generador de números aleatorios, y muchas cosas más.

7. Soporta la programación orientada a objetos (esta es una característica nueva introducida con el Ada 95). De hecho, Ada 95 fue el primer estándar de lenguaje de programación orientada a objetos.

8. Incluye interfaces para otros lenguajes (tal como C, Fortran y COBOL). Estos interfaces han sufrido importantes mejoras con el Ada 95. Al menos un compilador de Ada (de Intemetrics) puede generar código para una máquina virtual de Java (J-code), de esta forma, los programadores pueden usar Ada para desarrollar applets y aplicaciones Java.

2. Elementos léxicos

Ada es un lenguaje de formato libre, al igual que el Pascal y el C. Las líneas se pueden partir en cualquier lugar y los espacios se pueden situar en el lugar que se prefiera para separar los elementos léxicos. Sus identificadores o palabras reservadas se escriben una a continuación de otra y deben estar separadas por uno o más espacios, tabuladores y / o saltos de línea.

Ada no es sensible a las mayúsculas y minúsculas, excepto en el valor de las cadenas alfanuméricas, las cuales deben encerrarse entre comillas dobles. El mismo identificador con mayúsculas o minúsculas es equivalente.

En este sentido, Ada es un lenguaje que permite una gran libertad, pero para facilitar la lectura de los programas es aconsejable seguir un criterio consistente en el uso de los distintos tipos de identificadores y de las mayúsculas.

2.1 El primer programa

La versión en Ada del programa típico “Hello world” podria ser el siguiente

```
--
-- El primer programa tipico
--
with Text_IO;
procedure Hello is
begin
 Text_IO.Put_Line ("Hello WORLD!");
end Hello;
```

Se puede observar que la sintaxis es muy similar al Pascal. En el caso del Ada un programa comienza con la palabra `‘procedure’` y debe terminar con `‘end;’`.

La sentencia `‘with’` se utiliza para compilar el programa junto con un paquete de funciones que implementan las funciones de E/S sobre un terminal.

2.2 Identificadores

Ada requiere definir nombres para los procedimientos, paquetes y muchas otras construcciones. Estos nombres se denominan identificadores. Algunos ejemplos de identificadores son: “Hola”, “Disparar torpedo” y “X12”. Los identificadores deben comenzar con una letra, aunque después de la letra inicial se pueden añadir tanto dígitos como subrayados. Tal coo se indicó en la sección anterior, las mayúsculas y minúsculas se consideran equivalentes.

La sintaxis que siguen los identificadores en formato BNF es:

```
identifier ::= letter { [ "_" ] letter_or_digit }
letter_or_digit ::= letter | digit
```

Todos los caracteres de un identificador son significativos, y el compilador Ada debe soportar longitudes de líneas e identificadores de al menos 200 caracteres. Es de esperar que un programador no llegue a utilizar esta cantidad, pero l idea es que se ofrezca una gran flexibilidad.

Una consecuencia de esta sintaxis es que los subrayados nunca deben aparecen juntos. Esto se define así de forma intencionada debido a que en algunas impresoras dos subrayados consecutivos no se distinguen de uno solo. Los subrayados tampoco pueden aparecer al principio o al final de un identificador.

Se puede usar una única letra como identificadores, pero no conviene abusar de esto. Si un programa utiliza identificadores de una sola letra será muy complicado descifrarlo más adelante. Es mejor usar identificadores que indiquen claramente lo que guardan o lo que hacen. Tampoco conviene utilizar las letras “L” y “O” como identificadores, aunque Ada lo permite, debido a que la letra minúscula “l” se puede confundir fácilmente con el dígito 1 “1” y la letra mayúscula “O” puede ser indistinguible con el dígito cero “0” en algunos sistemas.

2.3 Literales numéricos

Un “literal numérico” es un número que se incluye en el código fuente de un programa. Hay dos tipos de literales numéricos, los literales reales y los literales enteros. Un literal real incluye el punto (“.”) mientras que un literal entero no. Ejemplos de literales enteros son “2”, “400” y “-7”. Ejemplos de literales reales son “2.0”, “400.0” y “3.14159”.

La notación tradicional de los exponentes (tal como 1.0E9) se permiten en los literales numéricos. Los exponentes se permiten incluso en los literales enteros, aunque en este caso el exponente no puede ser negativo.

Para facilitar la lectura de los números grandes, se permiten los subrayados en medio de un literal numérico.. Por ejemplo, es legal el valor “1_000_000”. El uso es similar al de la coma en los Estados Unidos o el punto en Europa. No se permiten dos subrayados consecutivos, un número no puede termina en subrayado y los subrayados no alteran el valor del literal numérico.

Una característica útil de Ada es la habilidad para expresar literales en otras bases entre 2 y 16 (el lenguaje C tiene una capacidad menor). Estos literales se llaman literales base. Para crear un literal base, se debe escribir primero la base deseada, a continuación el signo “#”, el número expresado en la base indicada y, finalmente, otro signo “#”. Por ejemplo, “2#1001_1000# es un número en base igual a 128+16+8 = 152.

Para mayor exactitud, la sintaxis BNF para los literales numéricos es:

```
numeric_literal ::= decimal_literal | based_literal
decimal_literal ::= numeral [ . numeral ] [ exponent ]
numeral ::= digit { digit | "_" }
exponent ::= "E" [ "+" | "-" ] numeral
based_literal ::= base "#" based_numeral "#" [ exponent ]
base ::= numeral
based_numeral ::= extended_digit { extended_digit | "_" }
extended_digit ::= digit | "A" | "B" | "C" | "D" | "E" | "F"
```

2.4 Caracteres y cadenas

En ocasiones se necesita un literal formado por un único carácter. Un carácter se representa como literal encerrándolo entre comillas simples (‘). Por ejemplo, ‘a’ representa la letra minúscula A. Esto es así incluso para le carácter comilla simple, que se representa como ‘’’.

Las cadenas, o tiras de caracteres, se representan encerradas entre comillas dobles (“). Para incluir una comilla doble en una cadena se debe escribir dos veces (“”). Por ejemplo, “Hola” y “El dice “”Hola”””” son cadenas válidas. Una cadena vacía se escribe como “”.

Se verá más adelante como representar los caracteres de control, pero por ahora indicaremos que los caracteres de escape al estilo C no se admiten. Estos se pueden utilizar también en Ada pero de forma distinta y, debido a la forma de trabajar de Ada, no son utilizados tan a menudo como en C.

2.5 Sentencias

Una sentencia en Ada corresponde a una acción, que puede ser la llamada a un subprograma o la asignación del resultado de una expresión en una variable (separando el destino y la expresión por el símbolo `':='`).

También se pueden definir sentencias mas complejas, con ejecución condicional o en forma de bucle.

En Ada las sentencias terminan con ';' (como en C), a diferencia del Pascal que se utiliza para separar sentencias. La diferencia fundamental es que en Ada, antes del `end` debe haber un ';', al contrario que en Pascal.

En un programa se pueden incluir comentarios que den claridad anteponiendo una pareja de guiones: '–'.

3. Tipos de datos y variables

Ada ofrece muchas posibilidades a la hora de definir tipos de datos. Permite definir nuevos tipos de datos así como tipos derivados de otros ya existentes.

Como ya se verá en otros capítulos, también es posible definir operadores sobre tipos de datos nuevos, o redefinir los operadores ya existentes.

A un tipo determinado se le puede asignar un nombre para poderlo utilizar en declaraciones posteriores. La forma de declarar un tipo es utilizando la escribiendo palabra reservada '`type`' el nuevo nombre del tipo, la palabra reservada '`is`', la definición del tipo y finalmente el carácter ';'.

Para crear una variable de un tipo determinado se debe indicar el nombre de la variable, el carácter ':' y a continuación el nombre del tipo terminado en ';'.

Así, la declaración de una variable `I` para utilizarla como un contador entero podría ser:

```
type Contador is new Integer;
I: Contador;
```

Para introducir un valor en una variable se utiliza el operador '`:=`'. Así, por ejemplo:

```
I := 1;
```

En los tipos de datos distinguiremos entre datos simples y compuestos. Los datos simples podrán contener un valor de un tipo determinado, mientras que los datos compuestos estarán formados por una colección de datos simples o compuestos.

3.1 Datos simples

El dato simple más elemental es el tipo `Integer`.

El tipo `Integer` lo utiliza Ada cuando no es importante el rango del valor que se va a guardar. Se garantiza que, al menos, puede contener un valor entre -32767 y 32767, pero estos límites depende realmente de la máquina y del compilador.

Las operaciones disponibles con las variables de tipo entero son las normales: "+" (suma), "–" (resta), "*" (multiplicación), "/" (división entera) y '**' (exponente), con la precedencia normal en estas operaciones.

También se dispone de los operadores normales de comparación: "=", "<",">","<=",">=" y "/=" para el opuesto de "=". Estos operadores dan como resultado un valor de tipo booleano.

El otro tipo de dato básico que se puede usar en operaciones aritméticas es el tipo `Float`. Este tipo admite valores numéricos reales y, al igual que el tipo `Integer`, el rango de valores que soporta depende del hardware y del compilador. Para el tipo `Float` se dispone de los mismos operadores aritméticos y de comparación que para el tipo `Integer`.

Una característica importante en Ada es que se puede definir el rango de valores que puede contener un tipo numérico, y la precisión en los tipos reales. Por ejemplo podemos definir los nuevos tipos:

```
type Mes is range 1..12;

type Voltaje is delta 0.1 range 0.0 .. 10.0;
type Peso is digits 10;
```

Un programa que se desee hacer transportable y que no dependa del hardware ni del compilador, debe utilizar este tipo de definiciones, el lugar de usar directamente los tipos `Integer` y `Float`.

En Ada, al contrario que en otros lenguajes como el C o el BASIC, el tipo booleano es un tipo distinto del entero llamado `Boolean`. Este tipo posee los valores `True` y `False`.

Las operaciones básicas para el tipo `Boolean` son: `and`, `or`, `xor` y `not`.

El último tipo de dato básico es el `Character`. Los valores constantes de este tipo se Expresan en Ada con un único carácter entre comillas simples, como por ejemplo: `'a'`.

Un carácter representa uno de los 256 caracteres posible en el conjunto de caracteres ``Latin-1", que es un super conjunto del conjunto de caracteres ASCII (también llamado ISO 646). Ada 95 también define el tipo `Whide_Character`, (conjunto de caracteres completo ISO 10646) para cuando es necesario manejar caracteres no incluidos en el ISO 646, como por ejemplo caracteres chinos o arábicos.

3.2 Tipos enumerados

En Ada se pueden definir tipos de datos en los el programador indica los valores que puede tomar. Por ejemplo, podemos definir un tipo para representar un día de la semana de la forma:

```
type Dia is (Lun, Mar, Mie, Jue, Vie, Sab, Dom);
```

Una propiedad importante en los tipos enumerados es que sus valores están ordenados. Esto permite definir:

- Establecer relaciones de comparación. Se puede usar los operadores “=”, “<”, etc.
- Están definidos los valores primero y último.
- Dado un valor, se puede obtener el anterior y el siguiente, salvo en el primero y el último respectivamente.
- Dado un valor, se puede obtener el lugar que ocupa en la relación de valores
- Dado un ordinal, se puede obtener su valor asociado para un tipo.

Para manejar estas propiedades Ada define los llamados atributos de un tipo.

Algunos atributos que define Ada son: `First, Last, Succ, Pred, Pos, Val, Value, Imatge`

Los atributos se utilizan separándolos del nombre del tipo con una comilla simple. Así, algunas expresiones verdaderas en el tipo `Dia` definido anteriormente son:

```
Dia’First = Lun
Dia’Last = Dom
Dia’Succ(Mar) = Mie
```

```
Dia'Pred(Mar) = Lun
Dia'Pos(Vie) = 4
Dia'Val(4) = Vie
Dia'Value("Vie") = Vie
Dia'Image(Vie) = "VIE"
```

3.3 Rangos y subrangos

Como ya hemos visto, podemos definir un tipo de datos especificando un rango de valores:

```
type Columna is range 1..72;
type Fila is range 1..24;
```

Una diferencia importante entre Ada y algunos lenguajes es que Ada considera distintas declaraciones de tipos como tipos diferentes, de manera que valores de un tipo no pueden mezclarse valores del otro tipo en una misma expresión, como por ejemplo, en el caso de los tipos `Columna` y `Fila`. Esto es útil para evitar errores pero, en ocasiones, puede interesar que no sea así.

Si queremos definir dos tipos cuyos valores queremos que sean compatibles, ambos se deben declaran como subtipos de un tipo común. En este caso los valores de los subtipos se considerarán como del tipo utilizado para generar el subtipo, y esto hace que se puedan mezclar.

```
subtype Dias_laborables is Dias range Lun..Vie;
subtype Dias_festivos is Dias range Sab..Dom;
```

Un subtipo es, simplemente, otro nombre de un tipo ya existente que puede tener algunas restricciones añadidas.

Los subtipos se pueden declarar en el mismo lugar que se pueden declarar los tipos.

3.4 Datos compuestos

Los datos compuestos están formados por una colección de otras datos, simples o a su vez compuestos.

Existen dos formas de crear datos compuestos: las tablas y las estructuras. Las tablas contienen una serie de datos todos del mismo tipo, mientras que las estructuras pueden estar formadas por datos de tipos distintos.

Para acceder a los distintos datos de una tabla se accede por medio de una índice o lista de índices en el caso de tablas multidimensionales.

Para acceder a los datos de una estructura se utiliza un nombre que se le da a cada uno de ellos.

3.4.1 Tablas

En Ada las tablas se definen utilizando la palabra reservada `array` y uno o varios tipos enumerados o rangos que se utilizarán como índices para acceder al elemento.

La descripción BNF es:

```
tipo_array::= "array" "("  tipo_enumerado
      {  "," tipo_enumerado }  ")" "of" tipo_base ";"
```

Para hacer referencia a un elemento de una tabla se debe indicar entre paréntesis, después de la variable, el índice o índices que correspondan.

Veamos algunos ejemplos:

```
-- Sample array type definitions:
  type Table is array(1 .. 100) of Integer; -- 100 Integers.
  type Schedule is array(Day) of Boolean; -- Seven Booleans, one per Day
  type Grid is array(-100 .. 100, -100 .. 100) of Float; -- 40401 Floats.

 -- Sample variable declarations:
  Products_On_Hand : Table;   -- This variable has 100 Integers.
  Work_Schedule : Schedule;
  Temperature : Grid;

 -- And sample uses:
  Products_On_Hand(1) := 20;  -- We have 20 of Product number 1
  Work_Schedule(Sunday) := False; -- We don't work on Sunday.
  Temperature(0,0) := 100.0;  -- Set temperature to 100.0 at grid point (0,0).
```

3.4.1.1 Tablas sin rango

Se puede definir un tipo tabla sin especificar su rango, de forma que éste se indique en la declaración de la variable. Para ello se indica el rango con los caracteres '<>' que indica que está por especificar.

Veamos un ejemplo:

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

str: cadena(1..100);
```

3.4.2 Strings

Una string es una secuencia de caracteres. En Ada 83 esta definido el tipo string que permite manejar secuencias de caracteres con una longitud fija.

La forma de declarar una string es:

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

O asignando un valor en la declaración:

```
A: String := "String con valor"; -- String de longitud 16
```

El tipo `String` es simplemente un tabla de caracteres. Aunque es un tipo muy sencillo, se pueden hacer varias operaciones con ellas como son:

- Asignarle un valor: `S := "Ejemplo   "`
- Extraer un carácter: `S(3)` o modificarlo: `S(8) = 's';`
- Extraer una subcadena: `S(3..5)` (strings de longitud 3)
- Concatenar dos strings con el operador "&": `"Un " & S` (strings de longitud 13)

- Pasarlo como parámetro a una función.

Las strings pueden pasarse como parámetros a procedimientos sin especificar el tamaño de la string. En cada caso se tomará el tamaño de la string en la llamada. Aunque es usual que el primer elemento sea el índice 1, no se tiene porque asumir esto. Para manejar los componentes de la string conviene utilizar los atributos: `S'First` para el primer índice, `S'Last` para el último y `S'Length` para el tamaño, donde `S` es el nombre del parámetro.

El tipo string plantea el inconveniente que al tener la longitud fija, strings con distinta longitud se consideran tipos distintos y, por tanto, al asignar un valor a una string ambas partes de la asignación deben tener la misma longitud.

Para resolver este problema en Ada 95 hay definidos dos nuevos tipos de strings:

`Bounded_string`: String de longitud variable hasta un tamaño máximo.

`Unbounded_string`: String de longitud variable sin limitación de tamaño.

3.4.3 Estructuras

Las estructuras definen tipos de datos que contienen una serie de campos, cada uno de un tipo de datos.

La forma de definir este tipo de datos es declarando la lista de campos, asociando a cada campo un nombre y un tipo. La declaración comienza con la palabra reservada `record` y termina con `end record`.

Veamos un ejemplo

```
type Date is
  record
    Day   : Integer range 1 .. 31;
    Month : Integer range 1 .. 12;
    Year  : Integer range 1 .. 4000;
  end record;
```

En la declaración del tipo se pueden incluir los valores a los que se asignarán cada uno de los campos en el momento de la declaración de una variable. Así, por ejemplo, se puede definir en tipo complejo de la forma:

```
type Complex is
  record
    Real_Part, Imaginary_Part : Float := 0.0;
  end record;
```

y utilizarlo para declarar variables de este tipo:

```
X: Complex
```

Para modificar el valor de los campos de esta variable hay que escribir a continuación del nombre de la variable, separado por el carácter '.', el nombre del campo que se desee:

```
X.Real_Part := 1.0
```

3.4.3.1 Estructuras parametrizadas

En la definición de una estructura se puede incluir un parámetro que se puede utilizar en la declaración para crear estructuras distintas según el valor del parámetro. Por ejemplo, se puede definir el tipo:

```
type Buffer(TAM: integer) is
  record
    Posicion: integer;
    Datos : array (1..TAM) of character;
end record;
```

y declarar variables dando valores distintos al parámetro:

```
Buf_peque: Buffer(100);
Buf_grande: Buffer(1000);
```

Al parámetro de la estructura también se le llama discriminante.

El parámetro puede tener también un valor por defecto.

3.4.3.2 Estructuras alternativas

Ada permite definir estructuras de datos variables, en los que los campos que forman la estructura dependerá del contenido de un campo selector.

Las estructuras alternativas se definen de forma similar a las estructuras parametrizadas, pero el parámetro se utiliza en una construcción del tipo `case`. En la construcción case deben figurar todos los valores posibles para el tipo del parámetro.

Un ejemplo de estructura alternativa seria:

```
type Atividad is ( Estudia, Trabaja, Paro );
type Estudios is ( Primaria, Secundaria, Universidad );
type Trabajo is ( Empleado, Directivo, Liberar );
type Persona( Labor: Actividad ) is
  record
    nombre: array (1..30) of character;
    edad: integer
    case Labor is
        when Estudia =>
            donde: Estudios;
            nota: Integer;
        when Trabaja =>
            ocupacion: Trabajo;
            sueldo: integer;
        when Paro =>
            null;
    end case;
end record;

Individuo1 : Persona( Labor => Estudia );
Individuo1 : Persona( Labor => Trabaja );
```

Una vez se ha definido el valor del selector este ya no se puede modificar.

Los campos de la parte variable se referencian igual que los de la parte fija.

3.4.4 Estructuras dinámicas

Ada permite la creación de estructuras que modifican su tamaño y el número de elementos que contienen dinámicamente a través de los tipos de datos `access`. Estos tipo de datos es semejante al tipo puntero del C o Pascal.

La especificación de un tipo `access` en formato BNF es:

```
access_object_type_declaration ::= "type" new_type_name "is"
                                    "access" ["all"] type_name ";"
```

Las variables del tipo `access` pueden hacer referencia a una variable o a ninguna, conteniendo el valor `null`. Cuando se crea un tipo `access` Ada la inicializa al valor `null`, salvo que se indique otra cosa.

Se pueden utilizar los operadores '=' y '/=' para comprobar si dos referencias son iguales o distintas, o si son `null`.

Con el tipo `access` tenemos una forma de acceder a variables nuevas, creadas en tiempo de compilación. Esto es especialmente útil si tenemos en cuenta que podemos crear variables de tipo estructura donde alguno de sus campos sea del tipo `access` a una variable del mismo tipo. De esta forma, por cada nueva variable creada tenemos posibilidad de crear otra y crear estructuras de datos sin límite de tamaño, como son las listas, los árboles, los grafos, etc.

La forma de crear una nueva variable es con la palabra reservada `new`. Veamos un ejemplo:

```
type List_Node;  -- An incomplete type declaration.
type List_Node_Access is access List_Node;
type List_Node is
    record
      Data        : Integer;
      Next        : List_Node_Access;  -- Next Node in the list.
    end record;

Current : List_Node_Access;
Root    : List_Node_Access;;


Current := new List_Node;
Root    := new List_Node;
```

Por seguridad, Ada 95 solo permite acceder con el tipo `access` a variables creadas dinámicamente. Si queremos poder acceder a cualquier variable de su tipo hay que incluir el modificador `all` en la definición del tipo.

Para acceder a los campos de la variable a la que apunta un tipo `access` es de la misma forma que si utilizáramos directamente la variable:

```
Current.Data := 1;
Root.Data := 2;
Root.Next := Current;
```

En la última expresión estamos asignando el contenido de la referencia `Current` a la referencia `Root.Next` (no se copia el contenido sino la referencia). Si queremos copiar el contenido hay que usar el modificador `all`.

```
Root.all := Current.all;  -- Sentencia (1).
Root     := Current;      -- Sentencia(2).
```

En la sentencia (1) se copian todos los campos de la variable a la que apunta `Current` en cada uno de los campos de la variable a la que apunta `Root`, mientras que en la sentencia (2) únicamente se copia la referencia.

Si queremos que una variable de tipo `access` obtenga la referencia de una variable de su mismo tipo se deben de cumplir dos cosas. Como ya hemos visto, el tipo `access` debe estar definido con el modificador `all`. Además la variable a la que se va a apuntar debe estar declarada con el modificador `aliased` antes del tipo, para indicar que se podrá acceder a su contenido, no solo con su nombre sino también desde variables `access` asignadas directa o indirectamente a esta variable.

Para poder asignar la referencia de una variable a otra de tipo acceso hay que utilizar el atributo `'Access` sobre la variable. Veamos como se haría la lista anterior de manera que la cabeza de la lista sea una variable global llamada `Head`:

```
...
type List_Node_Access is access all List_Node;
Current : List_Node_Access;
Root    : List_Node_Access;
Head    : aliased List_Node
...
begin
Root := Head'Access;
Current := new List_Node;
...
```

3.5 Inicialización y constantes

En Ada es posible asignar valores a variables en el momento de su creación. La forma de hacerlo es utilizando el operador asignación (`:=`) después del tipo de la variable y el valor que queremos que tome. Por ejemplo:

```
I: Interger := 1;
```

De la misma forma se pueden inicializar datos compuestos, tanto tablas como estructuras:

```
type Mes is ( Enero, Febrero, Marzo, Abril, Mayo, Junio, Julio,
              Agosto, Septiembre, Octubre, Noviembre Diciembre );
DiasMes: array ( Mes ) of integer := ( 31, 28, 31, 30, 31, 30,
                                       31, 31, 30, 31, 30, 31 );


type Complex is
  record
    Real_Part, Imaginary_Part : Float := 0.0;
  end record;
X: Complex := ( 0.0, 0.0 )
Y: Complex := ( Imaginary_Part => 0.0 );
```

Ada permite asignar solo parte de los campos de un dato compuesto, tanto en tablas como en estructuras, indicando el nombre de los datos que queremos asignar. Si no se indican todos los campos se debe utilizar el identificador `others` para indicar un valor por defecto.

Recordemos que en la declaración del tipo de una estructura también podíamos indicar los valores por defecto que podía tomar cada campo. En tal caso si se pueden inicializar solo algunos campos que modificarían el valor que se da al definir la estructura.

Por ejemplo:

```
DiasVerano: DiasMes := ( Julio => 31, Agosto => 31,
                      Septiembre =>30, others => 0 );
Y: Complex := ( Imaginary_Part => 0.0 );
```

Al declarar una variable inicializada podemos indicar al compilador que la trate como una constante. En este caso no está permitido modificar su valor. La forma de hacer esto es utilizar la palabra reservada `constant` antes del tipo. Por ejemplo:

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

La utilización de constantes es importante en aplicaciones empotradas donde el programa reside en una ROM. El compilador sitúa la memoria de las constantes en la zona de ROM en lugar de la zona de memoria RAM, de ahí la importancia de detectar si se intenta modificar esta variable.

Se pueden declarar constantes sin especificar el tipo:

```
V_MAX: constant := 1_000;
```

En este caso el compilador sustituye la constante como el literal del que toma el valor cada vez que se utiliza, sin construir una variable inicializada.

4. Secuencias de control

Las secuencias de control son construcciones que permiten romper la linealidad en la ejecución de un programa.

Básicamente son de dos tipos: la ejecución condicional y los bucles.

A diferencia de otros lenguajes como el Pascal o el C, las construcciones de control van siempre parentizadas, es decir, comienzan con una palabra que abre la construcción y terminan con otra que la cierra. Normalmente la palabra para cerrarla es `end` seguida de la utilizada para abrirla.

4.1 Ejecución condicional

Dentro de la ejecución condicional podemos distinguir dos tipos, la construcción `if` y la construcción `case`. El funcionamiento de estas es el mismo que en otros lenguajes como el Pascal o el C (en el C `switch` en lugar de `case`). Entre el principio y el final de la construcción pueden ir más de una sentencia Ada.

4.2 Construcción `if`

Las construcciones `if` evalúan una expresión con resultado booleano y, según el resultado sea verdadero o falso, ejecutan una secuencia de instrucciones u otra.

La expresión BNF de la secuencia `if` es:

```
construcción_if ::=
  "if" condición "then"
     secuencia_de_instrucciones
  {"elsif" condición "then"
     secuencia_de_instrucciones }
  ["else"
     secuencia_de_instrucciones]
  "end if;"
```

La construcción con `elsif` es equivalente a al utilización de otro `if` dentro de la parte del `else`, pero de esta forma se evitan múltiples `end if` al final de la construcción.

4.3 Construcción `case`

La construcción `case` es una particularización de la construcción `if` cuando se utilizan múltiples `elsif` y las condiciones están relacionadas con distintos valores que puede tener una expresión que de como resultado un valor escalar.

La expresión BNF de la secuencia `case` es:

```
construcción_case ::=
  "case" expression "is"
      case_statement_alternative
     {case_statement_alternative}
  "end case;"

case_statement_alternative ::=
   "when" discrete_choice_list "=>" sequence_of_statements
```

```
discrete_choice_list ::= discrete_choice { | discrete_choice }

discrete_choice ::= expression | discrete_range | "others"
```

Ilustremos esto con un ejemplo:

```
case A is                       -- Execute something depending on A's value:
  when 1          => Fly;       -- if A=1, execute Fly.
  when 3 .. 10    => Put(A);    -- if A is 3 through 10, put it to the screen.
  when 11 | 14    => null;      -- if A is 11 or 14, do nothing.
  when 2 | 20..30 => Swim;      -- if A is 2 or 20 through 30, execute Swim.
  when others     => Complain; -- if A is anything else, execute Complain.
end case;
```

Ada necesita que se especifiquen todos los valores posibles que puede tomar la expresión, o utilizar la parte correspondiente a `others`.

4.4 Bucles

Los bucles facilitan la ejecución repetida de una secuencia de instrucciones. Existen tres tipos de bucles: simples, con condición e iterativos.

Todos estos tipos de bucles comienzan con la palabra `loop` y terminan con `end loop`.

4.4.1 Bucles simples

Los bucles simples serían bucles infinitos sino se pudiera utilizar en su interior una instrucción para salir del bucle. La instrucción para salir de un bucle es `exit` (similar al `break` de C). La variante `exit when` permite salir del bucle cuando sólo si se cumple una determinada condición.

Veamos un ejemplo:

```
A := 1;
loop
  A := A + 1;
  exit when A > 3;
end loop;
```

4.4.2 Bucles con condición

Los bucles con condición son similares a los bucles simples pero con una condición antes del `loop` indicada por la palabra `while`. La condición se evaluará al principio de cada iteración, saliendo del bucle cuando el resultado de la condición sea falso. Veamos esto con un ejemplo:

```
A := 1;
while A <= 3
loop
  A := A + 1;
end loop;
```

4.4.3 Bucles iterativos

Los bucles iterativos son similares a los bucles simples pero con un rango de valores antes del `loop` indicada por la palabras `for` y `in`. Después del `for` se indica el nombre de una variable temporal que será visible únicamente dentro del bucle y que, en cada iteración, tomará uno de los valores de la secuencia que se indicará a continuación del `in`. Veamos un ejemplo:

```
A := 0
for I in 1..10
loop
  A := A + I;
end loop;
```

Es posible ejecutar el bucle comenzando por el último valor de la secuencia y terminando con el primero. Para ello hay que utilizar la palabra `reverse` después del `in`.

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

5. Unidades de Programa

Un programa Ada está compuesta por una o más unidades de programa. Una unidad de programa puede ser:

- *Subprograma*: Define una secuencia de instrucciones según un determinado algoritmo.
- *Paquete*: Define una colección de entidades. Los paquetes son los mecanismos principales de agrupación en Ada. Es algo similar al "module" del lenguaje Modula, o el "unit" del Turbo Pascal (a partir de la versión 6).
- *Tarea*: Define una determina unidad de ejecución que se puede realizar en paralelo con el resto de unidades de ejecución.
- *Unidad protegida*: Coordina datos compartidos entre distintas unidades de ejecución que funcionan en paralelo (no existe en Ada 83).
- *Unidad genérica*: Soporta la construcción de código reusable (similar a las 'Templates' de C++).

Los paquetes son las unidades de programación más importantes en Ada. La mayoría de los programas Ada son un conjunto elevado de paquetes, con un procedimiento como unidad principal que es quien comienza el programa.

5.1 Declaración y cuerpo

Las unidades de programa, incluyendo a los subprogramas y los paquetes, consisten normalmente en dos partes:

- La declaración, que contiene información que será visible desde otras unidades de programa (análogo al contenido de los ficheros ".h" del lenguaje C). La extensión usada en Ada es ".ads".
- El cuerpo, que contiene los detalles de la implementación que no necesita ser visible desde otras unidades de programa (análogo al contenido de los ficheros ".c" del lenguaje C). La extensión usada en Ada es ".adb".

Las dos partes de una unidad de programa usualmente se guardan en ficheros separados. Esta distinción entre declaración y cuerpo permite que un programa pueda ser diseñado, escrito y probado como un gran conjunto de componentes de software independientes.

Hay dos casos especiales que hacen más fácil la programación:

1. Para los subprogramas (procedimientos y funciones), no es obligatoria la declaración separada. Si un subprograma tiene cuerpo pero no tiene declaración, el cuerpo puede servir como su propia declaración.
2. Algunos paquetes pueden no tener detalles de implementación. Por ejemplo, un paquete que contenga únicamente una serie de constantes. De todas formas esto es poco usual, lo normal es que todos los paquetes tengan la parte de declaración y de la implementación.

5.2 Subprogramas

Los subprogramas pueden ser procedimientos y funciones. Lo que distingue a ambos es que las funciones devuelven un valor de un determinado tipo, mientras que los procedimientos no.

Un subprograma se compone de una parte declarativa y de un cuerpo.

En la parte declarativa se describe como se hace uso de este subprograma, mientras que en el cuerpo se describe lo que hace el subprograma.

Los subprogramas pueden tener parámetros que se pueden utilizar para pasar información al subprograma o para devolverla. Las palabras `in` y `out` indican respectivamente el modo en que se utilizan los parámetros (si no se indica nada, el modo por defecto es `in`).

En Ada no está permitido usar parámetros del tipo `out` en funciones. Con esto se evitan posibles efectos laterales en expresiones del tipo:

```
A := B + F(B)
```

En este caso, si la función `F` modificara el parámetro `B`, el resultado dependería de si la suma se evalúa de izquierda a derecha o de derecha a izquierda.

5.2.1 Declaración

La descripción de como declarar un subprograma en BNF es:

```
subprogram_declaration ::= subprogram_specification ";"

subprogram_specification ::= "procedure" procedure_name parameter_profile |
                  "function" procedure_name parameter_profile "return" type

parameter_profile ::= [ "(" parameter_specification
                            { ";" parameter_specification} ")" ]

parameter_specification ::= parameter_name_list ":" mode type
                            [ ":=" default_expression ]

mode ::= [ "in" ] | "out" | "in" "out"

parameter_name_list ::= identifier { "," identifier }

procedure_name ::= identifier
```

Un ejemplo de declaración de procedimiento y de función sería:

```
procedure Average(A, B : in Integer; Result : out Integer);

function Average_Two(A, B : in Integer) return Integer;
```

En los parámetros se puede indicar un valor por defecto añadiendo un valor de inicialización usando ‘`:=`’.

5.2.2 Cuerpo

En el cuerpo de un subprograma se define las operaciones que éste realiza, el uso que hace de los parámetros y, si se trata de una función, el valor que devuelve.

Se compone de una parte declarativa y de un cuerpo o secuencia de instrucciones.

En la parte declarativa se definen los tipos y las variables locales a este subprograma que apoyarán la construcción del cuerpo.

La secuencia de instrucciones es la que se ejecutará cada vez que se llame al subprograma.

La definición BNF del cuerpo de un subprograma es:

```
subprogram_body ::= subprogram_specification "is"
                      declarative_part
                    "begin"
                      sequence_of_statements
                    "end" [designator] ";"

declarative_part ::= { declarative_item }

declarative_item ::= object_declaration | subprogram_body

object_declaration ::= identifier_list : [constant] type [":=" expression] ";"
```

Como podemos ver en esta definición, en la parte declarativa se pueden incluir otro cuerpo de subprograma que será local al subprograma que se está creando.

En el caso de las funciones, el valor que se devuelve se indica con la instrucción `return`, seguida de la expresión resultado. Esta instrucción hace que se salga del subprograma.

Veamos todo esto con un ejemplo:

```
procedure Sum_Squares(A, B : in Integer; Result : out Integer) is

  function Square(X : in Integer) return Integer is
  begin -- this is the beginning of Square
    return X*X;
  end Square;

begin -- this is the beginning of Sum_Squares
  Result := Square(A) + Square(B);
end Sum_Squares;
```

Los parámetros se pueden inicializar a un valor por defecto usando "`:= expresion`".

5.2.3 Sobrecarga de subprogramas

En ocasiones se desea realizar la misma operación conceptual sobre distintos tipos de datos. Ada permite crear subprogramas con el mismo nombre pero con parámetros distintos, considerándolos como subprogramas independientes.

A esto se llama sobrecarga de subprogramas.

En el cuerpo de un subprograma se define las operaciones que éste realiza, el uso que hace de los parámetros y, si se trata de una función, el valor que devuelve.

5.2.4 Sobrecarga de operadores

El lenguaje Ada permite redefinir los operadores que se pueden utilizar sobre tipos de datos incluidos en el compilador, o definir estos operadores sobre nuevos tipos de datos creados por el programador.

Los operadores se definen como funciones donde el nombre es el símbolo utilizado por el operador entre comillas dobles (“”). Los parámetros son los que necesita el operador de los tipos para los que estamos definiendo el operador, y el resultado será del tipo del resultado que queremos que de este operador. Veamos un ejemplo:

```
type Complex is
record
  Real_part: Float;
  Imag_part: Float;
end record;

function “+” (X,Y: Complex) return Complex;
function “-” (X,Y: Complex) return Complex;
function “*” (X,Y: Complex) return Complex;
function “/” (X,Y: Complex) return Complex;
```

5.3 Paquetes

Los paquetes proporcionan una unidad básica en la descomposición de programas y permiten el desarrollo ordenador de programas, tanto pequeños como grandes.

Facilitan también el desarrollo de una aplicación llevada a cabo por un grupo de personas y utilizar paquetes de software similares a las librerías.

5.3.1 Parte declarativa

La parte declarativa de un paquete puede contener declaraciones de variables, de tipos y de subprogramas que serán visibles desde otras unidades de compilación. Se incluye en los ficheros con la extensión “.ads”.

Desde un paquete se puede impedir que quien lo utilice no pueda manejar arbitrariamente ciertos datos. Para ello es posible definir tipos de datos como limitados, utilizando la palabra `private`. En este caso el usuario del paquete solo podrá hacer uso de los datos por medio de los subprogramas que ofrece el paquete y utilizar las operaciones de igualdad del Ada: ‘`=`’ y ‘`/=`’ así como el de asignación ‘`:=`’. Esto último se puede evitar definiendo el tipo como `limited private`.

En el caso de los tipos privados el compilador necesita tener información de como están definidos, para poder declarar variables de este tipo y pasarlos como parámetros. Para ello se define en la parte de la declaración en una sección indicada como privada.

La descripción la parte declarativa de un paquete en el formato BNF es:

```
package_declaration: := "package" package_name "is"
                 { definition_element }
                 [ "private" { type_definition } ]
                 "end" [package_name ] ";"
```

```
definition_element ::= type_definition |
                       variable_declaration |
                       subprogram_definition |
                       private_type_definition

private_type_definition ::= "type" type_name "is" [ "limited" ] "private" ";"
```

5.3.2 Cuerpo

El cuerpo de un paquete cumple el mismo propósito que el cuerpo de un subprograma, en particular contiene los detalles de implementación de los subprogramas definidos en la parte declarativa. Se incluye en los ficheros con la extensión ".adb".

También puede contener declaraciones de otros elementos (tipos, variables o subprogramas) que sólo se vayan a utilizar dentro del paquete.

La descripción del cuerpo de un paquete en el formato BNF es:

```
package_body ::= "package" "body" package_name "is"
                      [ local_variables_and_types ]
                      subprograms_bodies
                 "begin"
                   initiallization_statements
                 "end" [package_name ] ";"
```

5.3.3 Uso de la información visible

Aunque un paquete define un conjunto de información, esta no es directamente visible desde otras unidades de programa, a no ser que hagan referencia a él.

Para poder utilizar los elementos declarados en otra unidad se debe usar la cláusula `with` al principio de la unidad.

Una manera de referencia un indicador declarado en un paquete es con anteponiendo a su nombre el nombre del paquete, separando ambos nombre por un punto:

```
Nombre_paquete.identificador_declarado
```

De forma alternativa, puede hacerse directamente visible en otra unidad del programa toda la información de la parte visible de un paquete usando la cláusula `use` al principio de la unidad.

El uso de las cláusulas `with` y `use` las veremos al tratar las unidades de compilación.

5.4 Unidades de compilación

Una unidad de compilación contiene o bien la parte de la declaración o bien el cuerpo de una unidad de programa, precedidas por las cláusulas de contexto `with` o `use` que necesite. Estas unidades de compilación pueden ser la declaración de un paquete, el cuerpo de un paquete, la declaración de un subprograma o un cuerpo de subprograma, junto con sus cláusulas de contexto.

El compilador de Ada compila colecciones de una o más unidades de compilación, por ello es importante entender lo que son. Para compilar algo, necesariamente debe ser parte de una unidad de compilación correcta.

No es necesario de esta forma la creación de proyectos o ficheros del tipo ‘`makefile`’. Se siguen unas reglas para los nombrar los ficheros que deben contener las Unidades de Compilación.

La sintaxis, simplificada, de una unidad de compilación en el formato BNF es:

```
1.  compilation_unit ::= context_clause library_item
2.  context_clause ::= {context_item}
3.  context_item ::= with_clause | use_clause
4.  with_clause ::= "with" library_unit_name { "," library_unit_name} ";"
5.  use_clause ::= "use" library_unit_name { "," library_unit_name} ";"
6.  library_item ::= package_declaration | package_body |
                     subprogram_declaration | subprogram_body
```

Con la cláusula `with` se está indicando al compilador que se quiere tener acceso a las definiciones del paquete o paquetes que se indican a continuación. La referencia de cada definición veíamos que es de la forma:

```
Nombre_paquete.identificador_ceclarado
```

Si queremos evitar poner en cada definición el prefijo del nombre del paquete, se debe utilizar la cláusula `use`. En este caso, siempre se podrá deshacer una posible ambigüedad utilizando el formato completo anterior.

Aunque la mayoría de los compiladores de Ada permiten escribir más de una unidad de compilación en un módulo, es aconsejable poner las unidades de compilación en módulos separados. El compilador de Ada GNAT requiere que cada unidad de compilación esté en un fichero separado.

6. Entrada / Salida básica.

Hasta ahora hemos hecho referencia al paquete `Text_IO` utilizando las operaciones básicas `Put` y `Get`. Vamos a ver ahora otras posibilidades que ofrece este paquete.

Con el paquete `Text_IO` también se pueden realizar operaciones sobre ficheros, utilizando el tipo llamado `File_Type`. Todas las operaciones con ficheros se realizan sobre objetos de este tipo. El valor por defecto para todas las funciones de entrada (como por ejemplo `Get`) es `Current_Input`, que es del tipo `File_Type`, mientras que el valor por defecto para las funciones de salida (como `Put` y `Put_Line`) es `Current_Output`.

Antes de realizar cualquier operación de entrada/salida sobre un fichero, este debe haber sido creado o abierto. Existen dos operaciones para realizar estas acciones que son `Create` y `Open`. `Open` abre un fichero ya existente, mientras que `Create` crea uno nuevo (eliminando el fichero original si existiera).

Antes de salir del programa se deben cerrar todos los ficheros abiertos. La operación para hacer esto es `Close`.

Las definiciones de estas funciones son:

```
  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);
```

El modo de apertura o creación puede ser `In_File`, `Out_File` o `Append_File`. El parámetro `Form` es opcional y se utiliza, si es necesario, para indicar información específica dependiente del sistema operativo, como por ejemplo los permisos de acceso. Veamos un ejemplo sencillo de creación de un fichero y escritura de texto en él.

```
with Text_IO;
use  Text_IO;

procedure Make_Hi is
  New_File : File_Type;
begin
  Create(New_File, Out_File, "hi");
  Put_Line(New_file, "Hi, this is a test!");
  Close(New_File);
end Make_Hi;
```

El paquete `Text_IO` modeliza los ficheros de texto como una secuencia de líneas, cada uno con cero o más caracteres. Los siguientes subprogramas ayudan a manejar los finales de línea y los finales de fichero:

```
Procedure New_Line
```

`New_Line` finaliza la línea actual y comienza una nueva línea. Puede tomas un parámetro opcional indicando cuantas líneas se deben crear (una por defecto). Se puede especificar también el fichero donde escribir las líneas (por defecto es `Current_Output`).

`Procedure Skip_Line`

`Skip_Line` es el contrario de `New_Line`, sitúa la lectura al principio de la línea siguiente a la línea actual, descartando el texto por leer hasta el final de la línea.

`Function End_Of_Line`

`End_Of_Line` devuelve `True` si la entrada está en el final de la línea, de lo contrario devuelve `False`.

`Function End_Of_File`

`End_Of_File` devuelve `True` si la entrada está en el final dl fichero, de lo contrario devuelve `False`.

`Function Line`

`Line` devuelve el número de línea del fichero que se está leyendo o escribiendo (la primera línea es la 1). Resulta útil cuando se están procesando ciertos datos de entrada y, de repente, aparece algún problema en la entrada.

Al igual que en las funciones `Put` y `Get`, el primer parámetro de estos subprogramas es del tipo `File_Type`. Si se quieren utilizar para ficheros distintos de los de por defecto, se debe pasar en este parámetro. Hay que tener en cuenta que, a diferencia que en otros lenguajes como el C, los subprogramas que no llevan parámetros no deben llevar paréntesis.

Veamos otro programa que utiliza un bucle muy típico en Ada () que visualiza sólo líneas largas:

```
with Ada.Strings.Unbounded, Text_IO, Ustrings;
use  Ada.Strings.Unbounded, Text_IO, Ustrings;

procedure Put_Long is
  -- Print "long" text lines
  Input : Unbounded_String;
begin
  while (not End_Of_File) loop
    Get_Line(Input);
    if Length(Input) > 10 then
      Put_Line(Input);
    end if;
  end loop;
end Put_Long;
```

También se implementan los fichero `Text_IO` correspondientes a los ficheros de entrada / salida estándar. Estos son: `Current_Input`, `Current_Output` y `Current_Error`. También ofrece los procedimientos `Set_Input`, `Set_Output` y `Set_Error` para modificar estos fichero.

Los procedimientos `Put` y `Get` que manejan `string` están sobrecargados con otros equivalentes que ofrecen E/S para otros tipos como son el tipo `Character` y los tipos numéricos

`Integer` y `Float`, estos últimos en los paquetes `Ada.Integer_Text_IO` y `Ada.Flota_Text_IO` respectivamente.

El compilador Ada o el sistema operativo puede retrasar la escritura en un fichero para mejorar la eficiencia utilizando buffer. Si en un momento determinado se desea sincronizar la escritura se debe utilizar el procedimiento `Flush`.

7. Proceso paralelo

El lenguaje Ada soporta programación concurrente (paralela) por medio de las tareas de Ada. Estas tareas se ejecutan concurrentemente y pueden interactuar con cualquier otra utilizando una serie de mecanismos. Estas tareas son muy parecidas a los threads o procesos ligeros.

Las tareas pueden arrancar (activarse) y detenerse (terminar). Existen una serie de mecanismos a través de los cuales las tareas pueden sincronizarse y comunicarse una vez se han activado. Los principales son:

- Una tarea puede esperar a que otra tarea termine. Este es el caso normal al terminar un procedimiento que tiene declaradas tareas locales.
- Una tarea puede enviar un mensaje a otra, esto se llama cita (rendezvous)
- Las tareas pueden utilizar objetos protegidos, los cuales ofrecen acceso de lectura y escritura protegido a los datos. Los objetos protegidos son una novedad de Ada 95.
- Las tareas pueden utilizar una serie de variables globales para comunicarse. Este último método es eficiente pero peligroso, especialmente si no se posee un domino de la programación concurrente. Ada lo permite porque algunas aplicaciones de tiempo real necesitan de esta eficiencia, pero siempre se debe utilizar con precaución.

En realidad, un programa Ada siempre incluye, al menos, una tarea, que es la que forma el programa principal que es puesto en marcha desde el sistema operativo.

7.1 Tareas

Las tareas son unos objetos especiales de Ada que tienen ejecución propia, independiente del proceso que los crea, a diferencia de los procedimiento que necesitan que sean llamados desde alguna parte del programa principal.

A diferencia de los procedimientos, en las tareas necesitan definirse su especificación y su cuerpo. Veamos como se declara una tarea en el siguiente ejemplo:

```
--
-- El primer programa concurrente
--
with Text_IO;
procedure Saludos is
  Task Saludo;
  Task body Saludo is
  begin
    Text_IO.Put_Line ("Buenos dias!");
  end;
begin
  Text_IO.Put_Line ("Hola!");
end Saludos;
```

En este programa en realidad hay dos tareas, la del programa principal que escribirá “Hola!” y la tarea Saludo que escribirá “Buenos días!”, aunque en el programa principal no haya ninguna referencia a la tarea Saludo.

Ada permite declarar tipos de tareas, en lugar de crearlas directamente. En este caso las tareas existirán en el momento que se instancia un objeto de este tipo. Para declarar el tipo basta sustituir en la especifación la palabra `Task` por `Task body`.

Podríamos definir en nuestro programa:

```
Task type Saludo;
Task body Saludo is
...
Varios: array(1..5) of Saludo;
```

En este caso se están creando 5 tareas que escribirán a la vez “Buenos días!”.

Hasta aquí no hay nada que permita la sincronización entre tareas. Para que las tareas puedan sincronizarse y pasarse información se utilizan las sentencias `entry` y `accept`.

La sentencia `entry` se utiliza en la declaración de la tarea. Define un nombre de entrada que puede aceptar la tarea y que, opcionalmente podrá tener parámetros, tanto de entrada como de salida.

El `accept` se utiliza en el cuerpo de la tarea. Define las acciones que debe realizar la tarea cuando otra invoca esta entrada. Cada sentencia `accept` debe tener su correspondiente `entry` en la parte de la declaración, con el mismo nombre y los mismos parámetros.

Veamos un ejemplo:

```
with Text_IO;
procedure Saludos is
  Task Saludo is
    entry start(nombre: string);
  end;
  Task body Saludo is
  yo: string(1..10);
  begin
    accept start(nombre: string) do
      yo := nombre;
    end;
    Text_IO.Put_Line ("Buenos dias " & yo);
  end;
begin
  Text_IO.Put_Line ("Comenzamos");
  Saludo.start(“Juan”);
end Saludos;
```

La sincronización de la tarea que hace el `entry` y la que hace el `accept` es de la siguiente manera:

- La primera tarea que llegue al `entry` o al `accept` queda esperando a la otra. En este consiste la cita (rendezvous). Hay que tener en cuenta que las citas van emparejadas, por el mismo nombre y el mismo tipo de parámetros. Los distintos `entry` de varias tareas a otra que todavía no ha hecho el correspondiente `accept` quedan encoladas por orden de llegada.

- Una vez las dos tareas han coincidido en la cita, pasa a ejecutarse la tarea que ha hecho el `accept`, ejecutando las instrucciones que figuran entre el `do` y el `end` siguientes, en

caso de existir, permaneciendo la otra tarea en espera (esto en necesario pues se pueden utilizar parámetros del tipo `out`).

- Una vez se ha alcanzado el `end`, el sistema decide cual de las dos tareas tomará el control de la CPU (según la prioridad, por ejemplo, o las dos a la vez si hay varios procesadores).

Con el mecanismo de cita tenemos las siguientes propiedades:

- Sincronización. La tarea que llega primero espera a la otra
- Intercambio de información. A través de los parámetros
- Exclusión mutua. Si varias tareas hacen el mismo `entry` simultáneamente, solo se procesa uno a la vez

Puede ocurrir que una tarea no conozca el orden en que va a recibir las citas, en tal caso interesa poder atender varios `accept` indistintamente. Esto se puede hacer con una construcción `select`. Veamos un ejemplo:

```
select
  accept cita1;
or
  accept cita2(i: integer) do
      ...
  end;
or
   delay 5.0;
end select;
```

Al llegar la tarea al `select` procesará cualquiera de las citas separadas por `or`. Después del `end` de un `accept`, y antes del siguiente, se pueden incluir instrucciones que solo se procesarán si es tratada el `accept` que le precede.

Como vemos en el ejemplo, opcionalmente se puede utilizar la sentencia `delay`, en lugar del `accept`, para programar un time-out. Si no se produjera ninguna cita en el tiempo indicado se procesarían las instrucciones que siguen al `delay` y se saldría del `select`.

Otra posibilidad que permite el Ada es activar o desactivar distintas posibilidades de un `select` según unas condiciones. La forma de hacerlo es con la sentencia `when`, seguido de una expresión booleana y el separador '`=>`' antes del `accept`. Podemos modificar el ejemplo anterior para que la `cita2` sólo se procese según el contenido de una variable.

```
Dos_citas: boolean;
...

select
  accept cita1;
or when Dos_citas =>
  accept cita2(i: integer) do
      ...
  end;
or
  delay 5.0;
end select;
```

Por último, la construcción `select` permite utilizar un `else` en lugar del `or delay`. En tal caso, si se llega al `select` y no hay ningún `entry` pendiente, la tarea no se quedará esperando sino que procesará las instrucciones que siguen al `else`. Hay que tener en cuenta que en un mismo `select` no tiene sentido utilizar a la vez el `or delay` y el `else`.

Otra utilidad de los timeouts es controlar el tiempo de ejecución de un determinado cálculo, y si este supera un determinado deadline finalizarlo para realizar un proceso alternativo.

En Ada esto se puede expresar de la forma:

```
select
     delay 0.1;
then abort
     -- sentencias del calculo
end select;
```

En el código anterior, si las sentencias del cálculo no se completan en 100 mseg, se abortaría su ejecución y se pasaría a ejecutar la sentencia siguiente al `end select`.

Otra características que se puede usar en la creación de tareas es definirlas con un discriminante, que hace el papel de un parámetro que se pasa a la tarea en el momento de su declaración (creación).

7.2 Tipos protegidos

Un tipo protegido es un objeto pasivo de protección sobre los datos de forma consistente, incluso cuando varias tareas están accediendo simultáneamente a ellos. Los tipos protegidos son muy eficientes en la forma en que se han implementado en Ada 95. Se pueden considerar como una forma muy avanzada de semáforos y monitores.

Un tipo protegido contiene datos a los que una tarea sólo puede acceder a través de un conjunto de operaciones protegidas definidas por el programador. Hay tres de operaciones protegidas:

1. Funciones protegidas, las cuales ofrecen acceso de solo lectura a datos internos. A las funciones protegidas pueden acceder múltiples tareas simultáneamente.

2. Procedimientos protegidos, los cuales ofrecen acceso de lectura y escritura a los datos internos. Cuando una tarea está ejecutando un procedimiento protegido, ninguna otra tarea puede interactuar con el tipo protegido.

3. Entradas protegidas, las cuales son como los procedimientos protegidos excepto en que añaden una barrera. Una barrera es una expresión booleana que debe valer verdadero para que la tarea que accede a la entrada puede continuar. Si la barrea se evalúa a falso la tarea quedaría bloqueada y se pondría en una cola hasta que la expresión se evaluara a verdadero.

Los tipos protegidos suelen ser muy eficientes, a menudo más que utilizar semáforos directamente. Las operaciones deben ser cortas del tipo poner a un flag, actualizar uno o dos valores, u otras similares. Las operaciones largas incrementan el tiempo de latencia máximo del sistema, (el tiempo que puede necesitar el sistema para responder a una nueva situación) lo cual no suele ser deseable.

Al igual que en las tareas, los tipos protegidos se pueden utilizar como instancias únicas o declararlos como tipos para utilizar múltiples instancias de un mismo tipo, incluso usándolas dentro de estructuras o tablas.

Veamos un ejemplo de como se podrían implementar los mutex utilizando tipos protegidos:

```
protected type Mutex is
  entry Lock;        -- Acquire this resource exclusively.
  procedure Unlock;  -- Release the resource.
private
  Busy : Boolean := False;
end Mutex;

protected body Mutex is
  entry Lock when not Busy is
  begin
    Busy := True;
  end Lock;

  procedure Unlock is
  begin
    Busy := False;
  end Unlock;
end Mutex;
```

Para crear un mutex y utilizarlo bastaría crear una instancia de este tipo:

```
Control : Mutex;

Control.Lock;
Operacion_segura;
Control.Unlock;
```

8. Unidades genéricas

Algunas veces es más útil crear una versión más genérica de un subprograma o paquete y entonces utilizar esta versión genérica para crear subprogramas o paquetes más específicos. La forma en que Ada permite hacer esto son las unidades genéricas (generics), y es equivalente a las plantillas (templates) de C++.

Las unidades genéricas permiten realizar subprogramas o paquetes con elementos que realizan operaciones con elementos (tipos o valores) que no están determinados. El usuario, antes de utilizarlos deberá definir cuales son estos elementos, y podrá utilizar la versión particular del subprograma o paquete sin tener que rescribirlo. Hay que tener en cuenta que si en la definición del genérico se utiliza alguna operación para un elemento abstracto, esta debe estar definida para el tipo concreto con el que se va a utilizar.

La forma más fácil de ver esto es con un ejemplo. Supongamos la función que intercambia dos valores enteros.

```
-- Here's the declaration (specification):
procedure Swap(Left, Right : in out Integer);

-- .. and here's the body:
procedure Swap(Left, Right : in out Integer) is
  Temporary : Integer;
begin
  Temporary := Left;
  Left := Right;
  Right := Temporary;
end Swap;
```

La forma de intercambiar dos valores podría ser idéntica independientemente del tipo de los datos que se están intercambiando. El algoritmo sería el mismo cara cualquier tipo, sea entero, real o string.

Podemos escribir una versión de `Swap` más general, sustituyendo el tipo `Integer` por un tipo genérico. Veamos como sería esta versión genérica:

```
-- Here's the declaration (specification):
generic
  type Element_Type is private;
procedure Generic_Swap(Left, Right : in out Element_Type);

-- .. and here's the body:
procedure Generic_Swap(Left, Right : in out Element_Type) is
  Temporary : Element_Type;
begin
  Temporary := Left;
  Left := Right;
  Right := Temporary;
end Generic_Swap;
```

En general, para crear una versión genérica de un subprograma (o paquete), se ha de escribir utilizando un número reducido de nombres de tipo genéricos, declarándolos como `private`. Antes del subprograma o paquete se debe escribir `generic` y una lista de la información que se quiere que sea genérica. La lista se llama de parámetros formales de un genérico; esta lista es similar a la lista

de parámetros en una declaración de procedimiento. Más tarde se explicará lo que significa la frase ‘ `is private` ’.

Aunque la declaración es similar, no se deben confundir los tipos genéricos con los tipos privados a los que nos referíamos en la parte declarativa de los paquetes. Entonces, en el paquete se debía incluir una parte privada en la que se definían los tipos privados, que no eran visibles desde el exterior del paquete. Aquí es al contrario, los tipos no se definen en el genérico sino en el exterior, y en el cuerpo del genérico se utilizan sin conocer exactamente lo que son (datos abstractos), solamente utilizando operaciones que estos tipos deben tener definidas.

Para utilizar un subprograma genérico (o paquete) tenemos que crear un programa (o paquete) a partir de la versión genérica. Este proceso se llama instanciar, y el resultado se llama una instancia. Por ejemplo, veamos como se crearía el procedimiento `Swap` a partir de su versión genérica `Generic_Swap`:

```
procedure Swap is new Generic_Swap(Integer);
```

Lo mismo sería para otros tipos de datos como:

```
procedure Swap is new Generic_Swap(Float);
procedure Swap is new Generic_Swap(Unbounded_String);
```

Se debe tener en cuenta que cuando se instancia un genérico el paso de los tipos se hace de la misma forma que en una llamada normal a un procedimiento.

8.1 Parámetros formales

En las declaración de los parámetros formales se pueden incluir varios tipos de elementos:

- Valores o variables de cualquier tipo. Estos se llaman “objetos formales”. Por ejemplo el valor de un tamaño máximo.
- Cualquier tipo. Estos se llaman “tipos formales”.
- Paquetes que son instancias de otros paquetes. Estos se llaman “paquetes formales”.

8.1.1 Objetos formales

La forma de declarar un objeto formal con la sintaxis BNF es:

```
formal_object_declaration ::= identifier_list ":" [ "in" | "in out" ]
                                  type_name [ ":=" default_expression ] ";"
```

Un ejemplo sería:

```
Maximum_Size : Integer;
```

Al instanciar un subprograma o paquete genérico se debe dar un valor adecuado para el objeto formal.

8.1.2 Tipos formales

Los tipos formales especifican el nombre de un tipo y que "clase de tipos" están permitidos. Una declaración formal de tipo especifica el "mínimo" o el "peor de los casos" de la clase de tipo que se requiere. El tipo más "mínimo" en Ada se llama tipo `limited private`. Este es el "peor de los casos" porque la palabra `private` significa que es posible que no se conozca nada sobre los detalles de la implementación y `limited` significa que no tiene definidas las operaciones de asignación e igualdad.

Una declaración simplificada de tipo formal tiene la siguiente sintaxis:

```
formal_type_declaration ::= "type" defining_identifier "is"
                            formal_type_definition ";"

formal_type_definition ::= ["tagged"] ["limited"] "private" | "(<>)"
```

Un tipo con el indicador `tagged` significa que se pueden construir tipos nuevos a partir de él en los que se amplia el número de componentes del tipo. Los tipos `tagged` soportan la herencia en el Ada, necesaria para la Programación Orientada a Objetos.

Veamos algunos ejemplos, con su significado:

```
type Item is limited private;  -- Item can be any type.
type Item is private;          -- Item can be any type that has assignment
                               -- (:=) and equal-to (=) operation.
type Item is tagged limited private; -- Item can be any tagged type.
type Item is tagged private;   -- Item can be any tagged type with :=.
type Item is (<>);             -- Item can be any discrete type, including
                               -- Integer and Boolean.
```

8.1.3 Paquetes formales

No se abordan en este manual

8.2 Tipos y objetos abstractos

Veamos un ejemplo de paquete genérico que implementa un stack de un tipo genérico de datos:

```
generic
  Size : Positive;
  type Item is private;
package Generic_Stack is
  procedure Push(E : in  Item);
  procedure Pop (E : out Item);
  Overflow, Underflow : exception;
end Generic_Stack;
```

La implementación de estas definiciones sería:

```
  package body Generic_Stack is
  type Table is array (Positive range <>) of Item;
  Space : Table(1 .. Size);
  Index : Natural := 0;

  procedure Push(E : in Item) is
  begin
```

```
    if Index >= Size then
      raise Overflow;
    end if;
    Index := Index + 1;
    Space(Index) := E;
  end Push;

  procedure Pop(E : out Item) is
  begin
    if Index = 0 then
      raise Underflow;
    end if;
    E := Space(Index);
    Index := Index - 1;
  end Pop;

end Generic_Stack;
```

Con este paquete podemos instanciar stacks de diferentes tipos y tamaños, incluso varios stacks iguales para manejar varios stacks.

Pero esto no es todo lo flexible que uno quisiera. Por ejemplo, no se pueden pasar stacks como parámetros a subprogramas, ni se pueden crear tablas de stacks. Esto es porque el paquete que se instancia no se comporta como un tipo.

A los paquetes genéricos como el anterior se les llama GADO (Generic Abstract Data Object). En cada instancia se está creando un único objetos sobre el que actuan los subprogramas definidos en el paquete.

Se puede conseguir mayor flexibilidad si dentro del paquete genérico definimos un nuevo tipo privado (por ejemplo `Stack`) y cambiamos cada operación para que reciba como parámetro un objeto de este tipo. De esta forma, utilizando el nuevo tipo, nosotros podemos pasar Stacks como parámetro, utilizarlo para crear stacks, y cualquier otra cosa que se puede hacer con los tipos. El ejemplo anterior del stack quedaría de la forma:

```
generic
  Size : Positive;
  type Item is private;
package Generic_Stack is
  type Stack is limited private;
  procedure Push(S : in out Stack; E : in  Item);
  procedure Pop (S : in out Stack; E : out Item);
  Overflow, Underflow : exception;
private
  type Stack is record
    Index: natural := 0;
    Space: array(1 .. Size) of Item;
  end record;
end Generic_Stack;
```

La implementación de los subprogramas se omite, pero serían similares a la versión utilizando el parámetro en lugar de la variable local al paquete.

Este tipo de paquetes genéricos se denominan GADT (Generic Abstract Data Type), pues definen un tipo de dato complejo que utiliza otro tipo de dato abstracto.

Los GADT son más costosos de manejar pues necesitan que se pase en un parámetro el objeto sobre el que va a operar cada subprograma, pero resultan más potentes que los GADO.

Por lo general, se recomienda que se creen GADT en lugar de GADO.

9. Excepciones

Los programas que trabajan en una situación real deben ser capaces de manejar situaciones en las que se producen errores u otras situaciones excepcionales. El lenguaje Ada ofrece facilidades para controlar este tipo de situaciones y facilitar su manejo.

En Ada, una excepción representa un tipo de situación excepcional, normalmente un error grave. Durante el tiempo de ejecución se puede capturar una excepción, lo cual indicará que se ha producido una situación excepcional.

La acción por defecto que se toma cuando se produce una excepción es detener el programa. Normalmente se muestra en nombre de la excepción y el lugar donde se ha producido, aunque esto depende del compilador. La ventaja de un lenguaje que maneje excepciones como el Ada y el C++ es que este comportamiento se puede modificar.

En el caso que no se quiera que se tome la acción por defecto, es necesario indicar que es lo que se quiere que se haga definiendo un manejador de excepción, en el que se indica que excepción se quiere manejar y que se debe hacer cuando se produzca la excepción.

Las excepciones normalmente representan algo que no ocurre normalmente, por tanto, es conveniente reservar su uso para manejar situaciones de error serias. No se debe utilizar para situaciones previstas o esperadas normalmente y nunca se debe hacer un uso abusivo de ellas. Normalmente el lugar donde está definido el manejador está lejos, del lugar en el código donde se ha producido la excepción, y un manejador puede tratar excepciones que se produzcan en lugares muy dispares y situaciones muy distintas, lo que acaba quitando claridad al código. Por lo tanto, las excepciones se deben usar únicamente cuando u subprograma no puede realizar su trabajo por alguna razón.

Las excepciones permiten eliminar del código la infinidad de comprobaciones de error que se debe hacer en cada llamada al sistema o valores fuera de rango en las variables de nuestro programa, siempre en situaciones que normalmente no producen error pero hay que controlar que hacer cuando se produzca ocasionalmente.

El Ada tiene un número predefinido de excepciones que se producen cuando fallan una serie de comprobaciones que están incluidas en el lenguaje. La excepción incluida en el lenguaje que ocurre más frecuentemente es la de error de rango (`Constraint_Error`), la cual se produce cuando una variable o expresión toma un valor fuera del rango permitido para su tipo. Ejemplos de estas son: guardar un valor en una variable demasiado grande o pequeño para su tipo, dividir por cero, o utilizar un índice inválido en una tabla.

Como es de esperar, la realización de estas comprobaciones quitan eficiencia al código, aunque esta es menos de al que uno pueda pensar. De todas formas se puede indicar al compilador que suprima estas comprobaciones, aunque esto se debe hacer sólo cuando el programa ha está bien probado y depurado. Aun así, hay partidarios de que no se prescinda nunca de estas comprobaciones.

Algunos paquete definen sus propias excepciones, por ejemplo, `Text_IO` define la excepción `End_Error` que se produce cuando se intenta hacer un `Get` después de haberse alcanzado el final de fichero, o la excepción `Name_Error` que se produce cuando se intenta abrir un fichero que no existe.

9.1 Declaración de excepciones

El programador puede declarar y utilizar situaciones excepcionales distintas de las incluidas en el compilador. Para poder utilizar una excepción antes debe declararse.

La descripción BNF de una declaración de excepción es:

```
exception_declaration ::= defining_identifier_list ": exception;"
defining_identifier_list ::= identifier { "," identifier }
```

Un ejemplo podría ser:

```
Mi_error : exception;
```

Las declaraciones de excepción normalmente se colocan en la parte de declaración de un paquete (parte visible).

La provocación de una excepción es también muy sencilla. La definición BNF de una llamada a excepción es:

```
raise_statement ::= "raise" [ exception_name ] ";"
```

Y para la excepción del ejemplo anterior podríamos hacer:

```
if I < 1 or i > 10 then
  raise Mi_error;
end if;
```

Podemos apreciar que el nombre de la excepción es opcional. Si no se incluye se provocará una de las excepciones definidas según veremos en el siguiente apartado.

9.2 Uso de excepciones

Para manejar una excepción una vez se ha producido debe definirse un manejador de excepción. Los manejadores de excepción se definen justo antes del `end` que empareja con un `begin`, intercalando la palabra `exception`. En este caso el `end` cerrará tanto el `begin` como los manejadores de excepción.

La forma de declarar los manejadores de excepción es análogo a las construcciones `case`. Una definición BNF simplificada de la declaración de un manejador de excepción es:

```
exception_handler ::= “when” exception_choice { "|"  exception_choice } "=>"
                      sequence_of_statements
exception_choice  ::= exception_name | "others"
```

La palabra `others` se utiliza para definir la acción a tomar en el resto de excepciones que no se han indicado en algún `when`.

Veamos un ejemplo de un procedimiento que utiliza una excepción para abrir un fichero, o crearlo en el caso de no existir.

```
procedure Open_Or_Create(File : in out File_Type;
                         Mode : File_Mode; Name : String) is
begin
  -- Try to open the file. This will raise Name_Error if
```

```
   -- it doesn't exist.
   Open(File, Mode, Name);
exception
   when Name_Error =>
      Create(File, Mode, Name);
end Open_Or_Create;
```

Cuando se produce una excepción Ada abandona lo que estaba haciendo y busca un manejador para excepción que se ha producido en la secuencia de instrucciones donde se ha invocado a `raise`. Una secuencia de instrucciones se entiende como el conjunto de instrucciones que figuran entre `begin` y `end`. Si no encuentra el manejador realiza un retorno desde el subprograma actual e intenta localizar el manejador en el programa llamante (desde el cual ha sido llamado), repitiendo esta operación hasta que lo encuentre o salga del programa (que es la acción por defecto de todas las excepciones).

Dentro de un manejador de excepción se puede realizar cualquier tipo de acción. Si se quisiera llamar a un manejador de la misma excepción de un nivel superior bastaría con utilizar una instrucción `raise` sin el nombre de la excepción.

9.3 Excepciones definidas en el lenguaje más comunes

El Ada se definen una serie de excepciones que se activan automáticamente cuando se produce una determinada situación de error. Las más comunes son:

Nombre	**Algunas causas que la produce**
`constraint_error`	Tipos enumerados, rangos e índices en tablas
`numeric_error`	División por cero
`program_error`	Select con todos los accept cerrados.
`storage_error`	Error en la gestión de memoria
`tasking_error`	Tarea inexistente

Algunos paquetes definen sus propias excepciones, como puede ser el paquete `Text_IO` utiliza las excepciones definidas en el paquete `Ada.IO_Exceptions` definido como:

```
package Ada.IO_Exceptions is

   Status_Error : exception;
   Mode_Error   : exception;
   Name_Error   : exception;
   Use_Error    : exception;
   Device_Error : exception;
   End_Error    : exception;
   Data_Error   : exception;
   Layout_Error : exception;

end Ada.IO_Exceptions;
```

10. Programación en Tiempo Real

10.1 Relojes

En muchas aplicaciones, el programa necesita tener una noción del paso del tiempo. Ada ofrece dos paquete que dan acceso a un reloj y permiten operar con el tiempo. Estos paquetes son:

```
Ada.Calendar
Ada.Real_Time
```

En ambos paquetes se define el tipo privado `Time` y se definen operaciones trabajar con el: Las más importantes son:

- Poner un valor al tiempo
- Leer el contenido de un tiempo
- Aritmética de tiempo (sumar y restar un intervalo)
- Comparación de tiempo

En ambos paquete el tiempo actual se obtiene por la función:.

```
function Clock return Time;
```

10.1.1 Paquete Calendar

El paquete `Calendar` trabaja con el tipo estándar `Duration`, que está definido como un `float` con un `delta` dependiente de la implementación. La precisión de este tipo viene dada por el valor `Duration'Small` y no debe ser mayor de 20 milisegundos.

El tiempo se define en un calendario a partir de los tipos:

```
subtype Year_Number  is Integer range 1901 .. 2099;
subtype Month_Number is Integer range 1 .. 12;
subtype Day_Number   is Integer range 1 .. 31;

subtype Day_Duration is Duration range 0.0 .. 86_400.0;
```

Las funciones para dar un valor al tiempo y leer su valor son:

```
function Year    (Date : Time) return Year_Number;
function Month   (Date : Time) return Month_Number;
function Day     (Date : Time) return Day_Number;
function Seconds (Date : Time) return Day_Duration;

procedure Split
  (Date    : Time;
   Year    : out Year_Number;
   Month   : out Month_Number;
   Day     : out Day_Number;
   Seconds : out Day_Duration);

function Time_Of
  (Year    : Year_Number;
   Month   : Month_Number;
   Day     : Day_Number;
```

```
            Seconds : Day_Duration := 0.0)
            return    Time;
```

10.1.2 Paquete Real_Time

El paquete `Real_Time`, además del tipo `Time`, define el tipo privado `Time_Span` en lugar del tipo `Duration` para representar un intervalo de tiempo. La unidad de tiempo más pequeña que se puede representar viene dada por la constante `Time_Unit`. El tiempo avanza en unidades de la constante `Tick`, el cual no debe ser mayor de 1 milisegundo.

Se ofrece una representación del tiempo monotónica del tiempo, equivalente al tiempo físico que se observa en un entorno. El valor del tiempo se puede considerar a partir de del valor de un entero I y un origen de tiempo, de forma que el tiempo actual vendrá dado por la expresión:

$$E + (I+1) * Time\_Unit$$

Los intervalos de tiempo representados por `Time_Span` equivaldría a valores de la expresión:

$$I * Time\_Unit$$

Para inicializar y leer el tiempo se dispone de las funciones:

```
type Seconds_Count is new integer range -integer'Last .. integer'Last;

procedure Split (T : Time; SC : out Seconds_Count; TS : out Time_Span);
function Time_Of (SC : Seconds_Count; TS : Time_Span) return Time;
```

Los valores del tipo `Seconds_Count` representa un número de segundos transcurridos desde el origen del tiempo

Para manejar los intervalos de tiempo se usan las funciones:

```
function To_Duration  (TS : Time_Span) return Duration;
function To_Time_Span (D : Duration)   return Time_Span;

function Nanoseconds  (NS : integer) return Time_Span;
function Microseconds (US : integer) return Time_Span;
function Milliseconds (MS : integer) return Time_Span;
```

10.1.3 Retardos y actividades periódicas

Además de tener acceso al tiempo, las tareas deben poder detenerse o retardar su ejecución durante un tiempo, de forma que se sincronicen con el paso del tiempo.

Una forma de retardar la ejecución sería por medio de un bucle haciendo una espera ocupada:

```
Start := Clock;

loop
    exit when Clock - Start > 10.0;  -- O To_Time_Span(10.0)
end loop;
```

Ada ofrece una primitiva para realizar un retardo, permitiendo que la tarea no quede consumiendo CPU hasta que finalice el retardo.

```
delay 10.0
```

Si deseamos que una tarea sea periódica se puede escribir utilizando una primitiva `delay` de la forma:

```
task body T is
begin
    loop
        Action;
        delay 10.0;
    end loop;
end;
```

En esta representación aparece el problema de que el periodo se ve incrementado por el tiempo de ejecución de la acción y el tiempo durante el cual la tarea es expulsada de la CPU en cada bucle.

Una mejora de la tarea periódica sería utilizando una variable con el tiempo absoluto de la siguiente activación:

```
task body T is
    Interval: constant Duration := 10.0;
    Next_time: Time;
begin
    Next_Time := Clock + Interval;
    loop
        Action;
        delay Next_Time - Clock;
        Next_Time := Next_Time + Interval;
    end loop;
end;
```

Esto solo reduce parcialmente el problema. Cabe la posibilidad, de que la tarea sea expulsada de la CPU justo después de la expresión `Next_Time - Clock` y antes de ejecutarse el `delay`. En este caso la siguiente activación se vería demorada en tiempo que la tarea permaneciera expulsada.

Ada resuelve este problema ofreciendo una directiva para realizar un retardo hasta un tiempo absoluto, en lugar de un retardo relativo: `delay until`. La tarea anterior se modificaría de la forma:

```
        delay until Next_Time;
```

10.2 Prioridades de tareas

Para asegurar el buen comportamiento de un sistema de tiempo real se beben eliminar las posibles indeterminaciones en la ejecución del programa de control.

El lenguaje Ada introduce un comportamiento no determinista en los siguientes aspectos:

- El orden en que se ejecutan las tareas: el orden en que las tareas utilizan la CPU
- Las gestión de las colas de las entries: cuando tienen lugar más de una alternativa.
- El comportamiento de los objetos protegidos: cuando varias tareas coinciden en su utilización y los problemas de inversión de prioridad.

Cuando hay más de una tarea lista para ser ejecutada, el orden de ejecución se puede determinar asignando una prioridad única a cada tarea.

La prioridad de la tarea se puede utilizar también para ordenar las colas de las entries y determinar el orden en que se procesan en una sentencia `select` y en los objetos protegidos.

Ada permite tanto asignar prioridades estáticas a las tareas y modificar su prioridad en tiempo de ejecución.

En el paquete `System` se definen los siguientes tipos para manejar las prioridades:

```
subtype Any_Priority is Integer
  range 0 .. Standard'Max_Interrupt_Priority;

subtype Priority is Any_Priority
  range 0 .. Standard'Max_Priority;

subtype Interrupt_Priority is Any_Priority
  range Standard'Max_Priority + 1 ..
Standard'Max_Interrupt_Priority;

Default_Priority : constant Priority :=
  (Priority'First + Priority'Last) / 2;
```

Una implementación debe soportar, al menos, 30 valores para la prioridad y un nivel distinto para la prioridad de interrupción.

La prioridad inicial de una tarea se asigna incluyendo una sentencia `pragma` en su especificación. Por ejemplo:

```
Task Controler is
    pragma Priority(10);
end Controler;
```

Para entidades que actúan como manejadores de interrupción se define un `pragma` especial:

```
pragma Interrupt_Priority(Expresion);
```

Se puede omitir la expresión. En tal caso se toma el mayor valor posible:

```
pragma Interrupt_Priority;
```

Si a una tarea se le asigna una prioridad de interrupción entonces no será interrumpida por interrupciones de un nivel inferior o igual al definido.

Una prioridad asignada utilizando uno de estos `pragma` se llama prioridad base. Una tarea puede tener una prioridad activa mayor, tal como veremos en las políticas de bloqueo de los objetos protegidos.

Se puede modificar la prioridad del programa principal incluyendo un `pragma` en su declaración. Si esta no es así se el sistema asigna la prioridad por defecto: `Default_Priority`.

Cualquier otra tarea que no incluya en su especificación un `pragma` de prioridad, toma la prioridad de la tarea que la crea.

Para modificar la prioridad de una tarea cuando ya ha sido creada se dispone del paquete `Ada.Dynamic_Priorities`, cuya especificación es la siguiente:

```
with System;
with Ada.Task_Identification;

package Ada.Dynamic_Priorities is

   procedure Set_Priority
     (Priority : System.Any_Priority;
      T        : Ada.Task_Identification.Task_Id :=
                    Ada.Task_Identification.Current_Task);

   function Get_Priority
     (T        : Ada.Task_Identification.Task_Id :=
                    Ada.Task_Identification.Current_Task)
     return System.Any_Priority;

end Ada.Dynamic_Priorities;
```

El identificador de una tarea se puede obtener a partir de la variable con la que se define la tarea `T` y el atributo `Identity`: `T'Identity.`

Además, en el paquete `Ada.Task_Identification` existe la función `Current_Task` que devuelve el `Task_Id` de la tarea que la llama.

Para elegir la política de planificación se puede usar el siguiente pragma:

```
pragma Task_Dispatching_policy(Policy_Identifier);
```

Un valor posible para la política de planificación es Fifo_withng_Prioprity, en el que las tareas con la misma prioridad se ejecutan en orden de activación.

10.3 Protocolos de bloqueo

Cuando se producen bloqueos entre tareas para asegurar la exclusión mutua en el uso de algún recursos se puede producir inversión de prioridad, según se ve en la primera práctica de la asignatura.

Ada ofrece un mecanismo para evitar la inversión de prioridad utilizando el protocolo de herencia de techo de prioridad. Este consiste en asignar una prioridad a un objeto protegido, de manera que mientras una tarea está ejecutando una función (procedimiento o entrada) protegida hereda la prioridad del objeto protegido, recuperando su prioridad base al finalizar esta ejecución.

Para poder utilizar el protocolo del techo de prioridad el programa Ada debe incluir el siguiente `pragma`:

```
pragma Locking_Policy(Ceiling_Locking);
```

Para definir la prioridad techo de un objeto protegido se utiliza, al igual que en las tareas, un `pragma` en su especificación:

```
pragma Priority(Expresion);
```

Un ejemplo de especificación de objeto protegido sería:

```
protected type Mutex is
    pragma Priority(28);
    entry Lock;        -- Acquire this resource exclusively.
    procedure Unlock;  -- Release the resource.
  private
    Busy : Boolean := False;
  end Mutex;
```

Si una tarea con prioridad mayor que la definida en un objeto protegido intenta utilizarlo se produce una excepción del tipo `Program_Error`. Esto es así porque en el protocolo de herencia de prioridad la prioridad del objeto protegido debe ser, al menos, la mayor de las prioridades de las tareas que lo utilizan.

10.4 Política de colas

Se puede elegir la política de tratamiento de colas para los objetos protegidos y las tareas, y el comportamiento de la política de selección para entries abiertos en un objeto protegido y en una sentencia `select`. Para esto se utiliza el `pragma:`

```
pragma Queuing_Policy(Identificador_de_politica);
```

Existen dos identificadores de políticas predefinidos. Estos son `Fifo_Queuing` y `Priority_Queuing`. Una implementación puede definir otras políticas. La política por defecto es `Fifo_Queuing`.

Las políticas de planificación se comportan como cabe esperar por el nombre. Solo cabe destacar que la selección entre distintas colas en la política `Fifo_Queuing` es arbitraria.

Una tarea se pone en la cola ordenada por prioridad según su prioridad activa. Si esta cambia no se modifica la posición en la cola. Ahora bien, si se modifica la prioridad base de la tarea mientras está suspendida, se retira de la cola y se reordena según la nueva prioridad (pero detrás de las tareas con la misma prioridad).

11. Indice