INTRODUCCIÓN A LA PROGRAMACION EN ADA

Ada, al igual que Pascal y C, es un lenguaje estructurado. Fue desarrollado por iniciativa y bajo la supervisión del Departamento de Defensa (DoD) de EE.UU.

Históricamente, C ha sido la elección preferida para la programación de aplicaciones embebidas, por su independencia del procesador utilizado y lo suficientemente de bajo nivel para permitir al programador manejar el hardware de la máquina. Además puede ser implementado en casi cualquier arquitectura, tiene una performance razonable, es un standard internacional y es familiar a la mayoría de los programadores de sistemas. Sin embargo tiene una serie de desventajas que pueden ser significantes, como por ejemplo la falta de soporte para la concurrencia (multithreading), una omisión importante para los sistemas de tiempo real. Así, el programador requiere usar una interfaz de programación externa o un sistema operativo de tiempo real (SOTR), comprometiendo así la portabilidad. Los lenguajes con modelos de concurrencia integrada pueden lidiar mejor con estos problemas.

Existen dos versiones normalizadas:

– Ada 83 (ISO 8652:1987)

– **Ada 95** (ISO 8652:1995)

La norma (ISO) de Ada 95 define:

– un *núcleo* común para todas las implementaciones

– unos *anexos* especializados para:

» programación de sistemas

» sistemas de tiempo real

» sistemas distribuidos

» sistemas de información

» cálculo numérico

» fiabilidad y seguridad

– Los anexos definen:

» paquetes de biblioteca

» mecanismos de implementación

Las características relevantes de Ada incluyen:

- Un modelo de thread/tarea para expresar actividades concurrentes, exclusión mutua para recursos compartidos, comunicación entre threads o coordinación de tareas y respuesta a eventos asíncronos (incluyendo las interrupciones de hardware).
- Facilidades para asignar prioridades a threads/tareas y para establecer mecanismos de planificación apropiados (por ejemplo, manejo de prioridades).
- Mecanismos para tratar con el tiempo: clocks, time outs, periodicidad.

Como C no soporta características de concurrencia, una solución es usar directamente los servicios de un SOTR para su manejo. Sin embargo, esto limita la programación a correr bajo ese SOTR. En contraste, Ada incluye facilidades para todos estos requerimientos de tiempo real.

Un programa en Ada está compuesto por una o más *unidades de programa*. Estas unidades de programa pueden ser **subprogramas** (que definen algoritmos ejecutables), **packages** (que definen colecciones de entidades), **tareas** (que definen actividades concurrentes), **unidades protegidas** (que definen operaciones para el uso coordinado de datos compartidos entre tareas) o **unidades genéricas** (que definen formas parametrizadas de packages y subprogramas).

Cada *unidad de programa* consiste normalmente de dos partes: una especificación que contiene la información que debe ser visible para las otras unidades y un cuerpo, el cual contiene los detalles de la implementación, los que no necesitan ser visibles a otras unidades. La mayoría de las unidades de programa pueden ser compiladas por separado.

Esta distinción de la especificación y del cuerpo, y la capacidad para compilar unidades por separado, permite que un programa sea diseñado, escrito y testeado como un juego de componentes de software en gran parte independientes.

Un programa en Ada normalmente hará uso de una librería de unidades de programa de utilidad general. El lenguaje proporciona medios que permiten la construcción de librerías propias. Todas las librerías se estructuran de una manera jerárquica; esto permite la descomposición lógica de un subsistema en componentes individuales. El

texto de una unidad de programa compilada por separado del programa debe nombrar las unidades de librería que requiere.

Unidades de programa

Un **subprograma** es la unidad básica para expresar un algoritmo. Hay dos clases de subprogramas: *procedimientos* y *funciones*. Un *procedimiento* es el medio de invocar una serie de acciones. Por ejemplo, puede leer datos, actualizar variables o producir una cierta salida. Puede tener parámetros, para proporcionar medios controlados de pasar la información entre el procedimiento y el punto de la llamada. Una *función* es el medio de invocar el cómputo de un valor. Es similar a un procedimiento, pero además devolverá un resultado.

Un **paquete (package)** es la unidad básica para definir una colección de entidades lógicamente relacionadas. Consiste de un archivo fuente que almacena ciertos comandos para realizar algunas operaciones como imprimir texto, resolver funciones matemáticas, etc. Por ejemplo, un paquete se puede utilizar para definir un juego de declaraciones y operaciones asociadas. Las porciones de un paquete pueden ocultarse del usuario, permitiendo así el acceso solamente a las características lógicas expresadas por la especificación del paquete.

Un *package* es una forma elegante de encapsular código y datos que interactúan juntos en una única unidad y puede actuar en una variedad de formas.

El **package** en Ada se divide en 2 partes distintas: la *especificación* y la *implementación*.

La *especificación* define que hace el *paquete* pero no como realiza su implementación. Esto es usado por el compilador de Ada para chequear y reforzar el uso correcto del package por el programador.

La *implementación* define los métodos que son invocados cuando un mensaje es enviado al objeto.

Nota: *Los packages son parecidos a los archivos de cabecera en C (headers). Pero... **¡Ojo!,** parecido no es igual. Los headers son bloques de código insertados por el Preprocesador de C. Un paquete es una unidad separada de compilación.*

Las unidades de **subprograma** y de **paquete** se pueden compilar por separado y arregladas en jerarquías de unidades del padre e hijo dando un control fino sobre la visibilidad de las características lógicas y de su implementación detallada.

Una unidad **tarea** es la unidad básica para definir una secuencia de acciones que se puedan ejecutar concurrentemente con las de otras tareas. Tales tareas pueden implementarse en multicomputadoras, multiprocesadores, o alternando la ejecución en un solo procesador. Una tarea puede definir una sola tarea a ejecutarse o un tipo tarea permitiendo la creación de cualquier número de tareas similares.

Una **unidad protegida** es la unidad básica para definir las operaciones protegidas para el uso coordinado de los datos compartidos entre tareas. La exclusión mutua es provista automáticamente, y pueden definirse protocolos más elaborados que manejan concurrencia. Una operación protegida puede ser un **subprograma** o un **entry**. Un **entry protegido** especifica una expresión booleana (una barrera del entry o **entry barrier**) que debe ser verdadera antes de que el cuerpo del entry se ejecute. Una unidad protegida puede definir un solo objeto protegido o un tipo protegido permitiendo la creación de varios objetos similares.

Todos los programas en Ada comparten una estructura básica:

```
with Package_Name; use Package_Name;

procedure Nombre_del_programa is
      Variable : {tipo_de_variable (Integer, Float, etc.)};

begin
      Sentencia_1;
      Sentencia_2;
end Nombre_del_programa;
```

El **`procedure`** (procedimiento) es básicamente el nombre del programa.

Donde dice: Variable : tipo_de_variable es una declaración de variable.

Por ejemplo, podría ser:

```
Variable : Integer;
```

Si fuera un valor decimal de punto flotante sería:

```
Variable : Float;
```

La declaración **`begin`** inicia la ejecución de las sentencias del programa.

Las líneas `Sentencia_1;` y `Sentencia_2;` no representan nada en particular en ADA. En un programa real, serían reemplazadas con comandos que realicen alguna acción.

La sentencia **`end`** `Nombre_del_Programa;` finaliza la ejecución del programa.

Ejemplo: programa que imprime "Hola Mundo":

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

  -- no se requieren variables aquí en este caso

begin
  Put ("Hola Mundo!");
end Hola_Mundo;
```

Un concepto importante en Ada es la idea de encapsular juntos ítems para formar un *package* que puede reutilizarse en otros programas. El **package** "Text_IO" contiene las funciones para operaciones de entrada/salida en Ada y son usadas por los programas del usuario que requieren ingresar o imprimir texto. La razón por la cual dice “Ada.” antes de "Text_IO." es que ADA puede tener interfaz con otros lenguajes diferentes, por ejemplo con C o COBOL. Como puede haber librerías con miles de packages de los cuales elegir, se necesita especificar qué librería se está usando.

El nombre del procedimiento es “Hola_Mundo”. Las declaraciones "with" y "use" invocan el package Text_IO dentro del programa de forma tal que sus funciones puedan ser usadas. Para correr el programa debería ser guardado en un archivo con un nombre, por ejemplo *hola_ mundo.ada*. En el caso de usar el compilador de GNAT debiera ser *hola_mundo.adb*.

En Ada, un *procedure* puede ser una unidad de programa autocontenido que puede compilarse independientemente. En el ejemplo, el procedimiento “Hola_Mundo” forma un programa completo que puede ser compilado y ejecutado por un compilador apropiado.

Como el programa solo imprime un par de palabras en pantalla, no requiere declaración de variables. En este caso se insertó un comentario (esto se hace a través de un doble guión (--) antes del texto “no se requieren variables en este caso”).

La declaración `Put ("Hola Mundo!");` contiene el comando que imprime el texto que está entre los paréntesis en la pantalla.

Una variante de este programa se muestra a continuación. Se pide al usuario su primer nombre que luego se imprime en pantalla junto con otro mensaje de texto:

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Ingresar_Nombre is
  Nombre   : String (1..80);
  Longitud : Integer;
begin
  Put ("Ingrese su primer nombre> ");
  Get_Line (Nombre, Longitud);
  New_Line;
  Put ("Hola ");
  Put (Nombre (1..Longitud));
  Put (", esperamos que disfrutes aprendiendo Ada!");
end Ingresar_Nombre;
```

La variable `Nombre` es el nombre ingresado por el usuario y la indicación de tipo `String (1..80)` indica que `Nombre` es una cadena de hasta 80 caracteres de largo (definido por la variable `Length`).

Al ejecutar el programa, éste imprimirá en pantalla "Ingrese su primer nombre >”, esperando aceptar una entrada ingresada por el usuario. Luego que el usuario ingresa su nombre y presiona Enter, el programa saltará una línea (`New_Line;`) e imprimirá “Hola [nombre del usuario], esperamos que disfrutes aprendiendo Ada!”.

Declaraciones y sentencias

El cuerpo de una unidad del programa contiene generalmente dos partes: una parte declarativa, que define las entidades lógicas que se utilizarán en la unidad del programa, y una secuencia de declaraciones, que define la ejecución de la unidad de programa.

La parte declarativa asocia nombres a las entidades declaradas. Por ejemplo, un nombre puede denotar un tipo, una constante, una variable, o una excepción. Una parte declarativa también introduce los nombres y los parámetros de otros subprogramas jerarquizados, los paquetes (**packages)**, las unidades de tarea (**tasks**), las unidades protegidas, y las unidades genéricas que se utilizarán en la unidad de programa.

La secuencia de declaraciones describe una secuencia de las acciones que deben ser realizadas. Las declaraciones se ejecutan en sucesión (a menos que una transferencia del control haga que la ejecución continúe desde otro lugar).

Una declaración de asignación cambia el valor de una variable. Una *llamada del procedimiento* (**procedure call**) invoca la ejecución de un procedimiento después de asociar cualesquiera parámetros reales proporcionados en la llamada a los parámetros formales correspondientes.

Las sentencias **case** e **if** permiten la selección de una secuencia cerrada de declaraciones basadas en el valor de una expresión o en el valor de una condición.

La sentencia **loop** proporciona el mecanismo iterativo básico en ADA. Una sentencia **loop** especifica que una secuencia de declaraciones debe ser ejecutada en varias ocasiones según lo especificado por un esquema de iteración, o hasta que sea encontrada una sentencia **exit**.

Ciertas sentencias se asocian a la ejecución concurrente. Una sentencia **delay** retrasa la ejecución de una tarea por una duración especificada o hasta un tiempo especificado. Un **entry call** se escribe como una llamada del procedimiento; solicita una operación en una tarea o en un objeto protegido, bloqueando al llamador hasta que la operación pueda ser ejecutada. Una tarea llamada puede aceptar un **entry call** ejecutando un **accept**, que especifica las acciones entonces que se realizarán como parte del *rendezvous* con la tarea que realizó la llamada. Un **entry call** en un objeto protegido es procesado cuando la evaluación de la barrera correspondiente (entry barrier) da verdadero, con lo cual el cuerpo del **entry** se ejecuta. La sentencia **requeue** permite la disponer de un número de actividades relacionadas con control preferencial. Una forma de la sentencia **select** permite una espera selectiva para uno de varios rendezvous alternativos. Otras formas de la sentencia **select** permiten llamadas condicionales o sincronizadas del **entry** y la transferencia asincrónica del control en respuesta a un cierto evento que lo acciona.

La ejecución de una unidad del programa puede encontrar situaciones de error en las cuales la ejecución normal del programa no puede continuar. Por ejemplo, un cómputo aritmético puede exceder el valor máximo permitido de un número, o puede hacerse un intento para tener acceso a un componente de un **array** usando un valor de índice incorrecto. Para ocuparse de tales situaciones de error, las declaraciones de una unidad de programa pueden ser textualmente seguidas por manejadores de excepción (**excepcion handlers**) que especifican las acciones que se tomarán cuando se presenta la situación de error. Las excepciones pueden ser elevadas explícitamente por una sentencia **raise**.

Tipos de datos

Cada objeto en ADA tiene un tipo, que caracteriza un sistema de valores y un sistema de operaciones aplicables. Un *tipo de datos* es un conjunto de *valores* con un conjunto de *operaciones primitivas* asociadas.

Las clases principales de tipos son: **tipos elementales** (que abarcan tipos de enumeración, numéricos y de acceso) y **tipos compuestos** (incluyendo tipos de arreglo y de registro).

Un **tipo enumeración** define un conjunto ordenado de literales distintos, por ejemplo una lista de estados o un alfabeto de caracteres. Los booleanos y carácter están predefinidos. Por ej.:

```
Boolean -- predefinido
Character -- predefinido
type Mode is (Manual, Automatic);
```

Los **tipos numéricos** proporcionan medios para realizar cómputos numéricos exactos o aproximados. Los cómputos exactos utilizan los tipos de número entero, que denotan conjuntos de números enteros consecutivos. Los cómputos aproximados utilizan tipos de punto fijo, con límites absolutos en el error, o tipos de punto flotante, con límites relativos en el error. Los tipos numéricos entero, flotante y duración están predefinidos.

Enteros

– Con signo

```
Integer -- predefinido
type Index is range 1 .. 10;
```

– Modulares

```
type Octet is mod 256;
```

– Coma flotante

```
Float -- predefinido
type Length is digits 5 range 0.0 .. 100.0;
```

– Coma fija

– Ordinarios

```
Duration -- predefinido
type Voltage is delta 0.125 range 0.0 .. 5.25;
```

– Decimales

```
type Money is delta 0.01 digits 15;
First, Last : Index;
Front, Side : Length;
```

Los objetos de **tipo acceso** designan valores de otros tipos, por ej.:

```
type State_Reference is access State;
```

Los objetos a los que se accede a través de un tipo acceso de crean dinámicamente:

```
S : State_Reference := new State;
```

Los **tipos compuestos** permiten definiciones de objetos estructurados con componentes relacionados. Incluyen arreglos y registros. Un arreglo (**array**) es un objeto con componentes indexados del mismo tipo. Un registro (**record**) es un objeto con componentes nombrados posiblemente de diversos tipos. Tareas y tipos protegidos son también formas de tipos compuestos. Los tipos **array** y **Wide_String** están predefinidos.

Arreglos

```
type Voltages is array (Index) of Voltage;
type Matrix is array (1 .. 10, 1 .. 10) of Float;
```

Registros

```
type State is
   record
     Operating_Mode : Mode;
      Reference : Voltage;
   end record;
```

Subprogramas

Hay dos tipos de subprogramas

- **procedimiento**: abstracción de acción
- **función**: abstracción de valor

Ambos pueden tener *parámetros.* Un subprograma tiene dos partes

- **especificación**: define la interfaz (nombre y parámetros)
- **cuerpo**: define la acción o el algoritmo que se ejecuta cuando se invoca el subprograma

A veces se puede omitir la especificación. En este caso la interfaz se define al declarar el cuerpo.

La especificación se declara en una zona declarativa.

```
procedure Reset;   -- sin parámetros

procedure Increment(Value : in out Integer;
Step : in Integer := 1);

function Minimum (X,Y : Integer) return Integer;
```

Hay tres *modos* de parámetros formales:

- **in:** No se modifica al ejecutar el subprograma. Pueden tener un valor por defecto.
- **out:** El subprograma debe asignar un valor al parámetro
- **in out:** El subprograma modifica el valor del parámetro

Los parámetros de las funciones solo pueden ser de modo **in.**

Los modos no están ligados al mecanismo de paso de parámetros.

Cuerpo del subprograma

Se coloca en una zona declarativa.

```
procedure Increment (Value : in out Integer;
    Step : in Integer := 1) is
    Value := Value + Step;
end Increment;

function Minimum(X,Y : Integer) return Integer is
    if X <= Y then
    return X;
    return Y;
end Minimum;
```

Estructura de programas

Los subprogramas son *unidades de programa:*

– hay otras: paquetes, tareas, y objetos protegidos

– un *paquete* es un módulo que contiene declaraciones de tipos, objetos, subprogramas, y otras unidades de programa.

También son *unidades de compilación:*

– los paquetes también lo son

– un fichero fuente contiene una (o a veces más) unidades de compilación

Los subprogramas y paquetes compilados forman una *biblioteca de compilación.*

Un programa ejecutable se *monta* a partir de una biblioteca que incluya un *procedimiento principal.*

Un programa Ada se compone de:

– un procedimiento principal (normalmente sin parámetros)

– otros subprogramas o paquetes escritos por el programador

– subprogramas y paquetes predefinidos (y precompilados)

Cuando se usan elementos de un paquete hay que importar el paquete con una cláusula **with**

with nombre-de-paquete{, nombre-de-paquete};

La cláusula **use** permite hacer referencia directa a los nombres declarados en los paquetes importados.

use nombre-de-paquete{, nombre-de-paquete};

La cláusula '`with Ada.Text_Io; use Ada.Text_Io;`' vuelve disponibles los contenidos del paquete `Ada.Text_Io` a la unidad de programa siguiente. El package `Ada.Text_Io` contiene definiciones para realizar operaciones de entrada y salida de caracteres y objetos de cadenas. El efecto exacto de estas cláusulas es el siguiente:

- `with Ada.Text_Io;`

  Vuelve disponible a la unidad todos los componentes públicos del package. Sin embargo, cuando los componentes del package son usados en un programa, deben estar predefinidos con el nombre del package.

- `use Ada.Text_Io;`

  Permite que los componentes públicos del package sean usados sin tener que predefinir su nombre con el del package.

Paquetes predefinidos

Operaciones numéricas:

– Ada.Numerics, Ada.Numerics.Generic_Elementary_Functions

Operaciones con caracteres y cadenas:

– Ada.Characters, Ada.Strings, etc.

Entrada y salida:

– Ada.Text_IO, Ada.Integer_Text_IO, Ada.Float_Text_IO, etc.

Interfaces con otros lenguajes

Otros.

Subtipos

Un *subtipo* es un subconjunto de valores de un tipo, definido por una *restricción.* La forma más simple de restricción es un intervalo de valores.

```
subtype Small_Index is Index range 1 .. 5;
subtype Big_Index is Index range 6 .. 10;
subtype Low_Voltage is Voltage range 0.0 .. 2.0;
```

– Hay dos subtipos predefinidos

```
subtype Natural is Integer range 0 .. Integer'Last;
subtype Positive is Integer range
```

Procedimientos y funciones

Los procedimientos y funciones permiten al programador abstraer código en subprogramas que pueden ser reutilizados en diferentes lugares de un programa o aún en otros programas. Sin embargo otros mecanismos como el **package**, permiten mucha mayor flexibilidad para la reutilización de código en los programas.

Una función o procedimiento es un conjunto de una o más *declaraciones* de Ada dentro de un subprograma, el que puede ser llamado y ejecutado desde cualquier parte de un programa.

Las funciones y procedimientos permiten al programador un grado de abstracción para la resolución de problemas. Sin embargo, son más potentes cuando se usan combinados con otros datos relacionados para formar una *clase.*

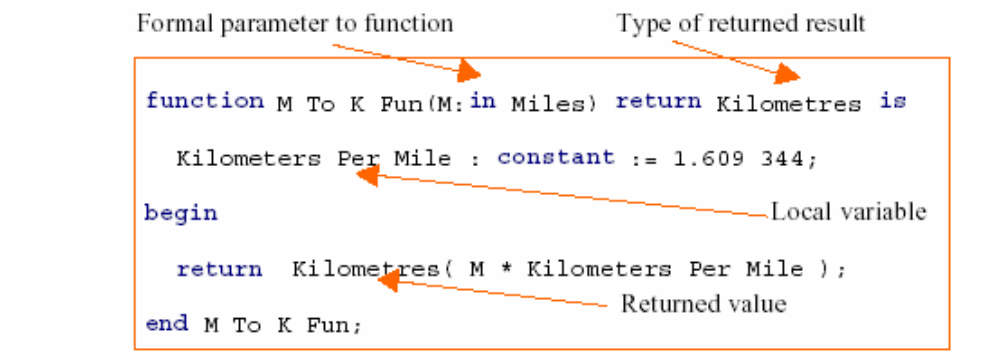
Funciones

Una función es un subprograma que transforma sus valores de entrada en valores de salida. Por ejemplo una función para convertir una distancia en millas a kilómetros sería:

```
type Miles      is digits 8 range 0.0 .. 25_000.0;
type Kilometres is digits 8 range 0.0 .. 50_000.0;

function M_To_K_Fun(M:in Miles) return Kilometres is
  Kilometres_Per_Mile : constant := 1.609_344;
begin
  return  Kilometres( M * Kilometres_Per_Mile );
end M_To_K_Fun;
```

Los componentes principales de esta función son:



La función `M_To_K_Fun` es usada en el siguiente programa, que imprimirá una tabla de conversión de millas a kilómetros. En Ada, una función o procedimiento puede declararse en la sección de declaración de un procedimiento o de una función.

Usando esta técnica, los tipos para `Miles` y Kilometres pueden hacerse visibles para la función `M_To_K_Fun` y para el cuerpo principal del código que implementa la impresión de la tabla de conversión.

```
with Ada.Text_Io, Ada.Float_Text_Io;
use  Ada.Text_Io, Ada.Float_Text_Io;
procedure Main is
  type Miles      is digits 8 range 0.0 .. 25_000.0;
  type Kilometres is digits 8 range 0.0 .. 50_000.0;

  function M_To_K_Fun(M:in Miles) return Kilometres is
    Kilometres_Per_Mile : constant := 1.609_344;
  begin
    return  Kilometres( M * Kilometres_Per_Mile );
  end M_To_K_Fun;

  No Miles : Miles;

begin
  Put("Miles  Kilometres"); New_Line;
  No Miles := 0.0;
  while No_Miles <= 10.0 loop
    Put( Float(No_Miles), Aft=>2, Exp=>0 ); Put("    ");
    Put( Float( M To K Fun( No Miles ) ), Aft=>2, Exp=>0 );
    New_Line;
    No Miles := No Miles + 1.0;
  end loop;
end Main;
```

Luego de compilado y ejecutado, el programa imprimirá los siguientes resultados:

```
Miles  Kilometres
 0.00      0.00
 1.00      1.61
 2.00      3.22
 3.00      4.83
 4.00      6.44
 5.00      8.05
 6.00      9.66
 7.00     11.27
 8.00     12.87
 9.00     14.48
10.00     16.09
```

Variables locales

Cuando una variable es declarada dentro de una función, su tiempo de vida es el de la función. Al comenzar la ejecución de la función, automáticamente se crea espacio para las variables locales en la forma de una pila. Al finalizar la función, el espacio creado para las variables locales es devuelto al sistema.

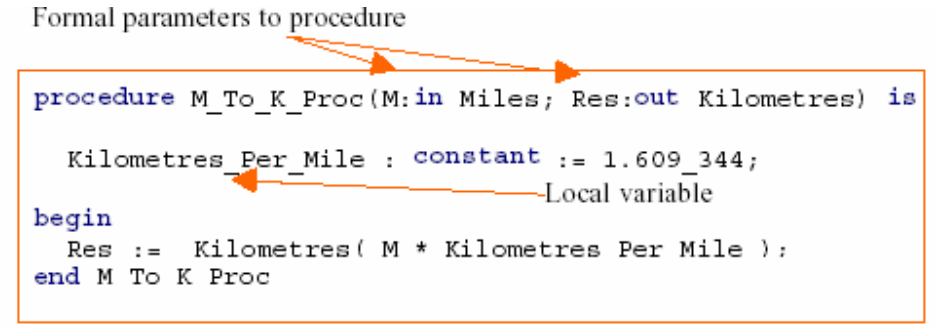
Procedimientos

Son unidades de programa que, a diferencia de una función, **no retornan un resultado**. Si alguna información debe ser retornada, lo hace escribiendo a un parámetro que se ha declarado con el modo **out**.

Escribiendo a ese parámetro del procedimiento, se actualiza el valor del parámetro pasado al procedimiento.

```
type Miles      is digits 8 range 0.0 .. 25_000.0;
type Kilometres is digits 8 range 0.0 .. 50_000.0;
procedure M_To_K_Proc(M:in Miles; Res:out Kilometres) is
  Kilometres_Per_Mile : constant := 1.609_344;
begin
  Res :=  Kilometres( M * Kilometres_Per_Mile );
end M_To_K_Proc;
```

Los componentes principales de este procedimiento son:



Este procedimiento puede ser usado en un programa de la siguiente manera:

```
with Ada.Text_Io, Ada.Float_Text_Io;
use  Ada.Text_Io, Ada.Float_Text_Io;
procedure Main is
  type Miles      is digits 8 range 0.0 .. 25_000.0;
  type Kilometres is digits 8 range 0.0 .. 50_000.0;

  procedure M_To_K_Proc(M:in Miles; Res:out Kilometres) is
    Kilometres_Per_Mile : constant := 1.609_344;
  begin
    Res :=  Kilometres( M * Kilometres_Per_Mile );
  end M_To_K_Proc;

  No_Miles : Miles;
  No_Km    : Kilometres;

begin
  Put("Miles  Kilometres"); New_Line;
  No_Miles := 0.0;
  while No_Miles <= 10.0 loop
    Put( Float(No_Miles), Aft=>2, Exp=>0 ); Put("     ");
    M_To_K_Proc( No_Miles, No_Km );
    Put( Float( No_Km ), Aft=>2, Exp=>0 );
    New_Line;
    No_Miles := No_Miles + 1.0;
  end loop;
end Main;
```

Luego de ejecutado este programa producirá la misma salida como el anterior que usaba una función para convertir millas a kilómetros.

Tareas

Las tareas en Ada son un mecanismo que permite que varios "threads" o "hilos" de ejecución se ejecuten en un programa simultáneamente. Esto facilita la construcción de programas de tiempo real que pueden procesar mensajes generados por fuentes múltiples de una manera ordenada.

Un programa puede tener secciones de código que se ejecuten concurrentemente mientras no tengan interacción o dependencia entre sí.

Por ejemplo, el cálculo del factorial de un número entero y la determinación de si un número es primo, pueden hacerse concurrentemente como threads de ejecución separados. Esto se puede implementar mediante un **tipo tarea** dentro de un *package*. Una vez creada una instancia (o hilo) de la tarea, ésta se ejecutará como un thread individual.

Un hilo es una unidad básica de utilización de la CPU y consiste en un contador de programa, un juego de registros y un espacio de pila. Comparte con sus hilos pares la sección de código, la sección de datos y recursos del sistema operativo como archivos abiertos y señales, lo que se denomina colectivamente una tarea. Al igual que los procesos, los hilos comparten la CPU, y sólo hay un hilo activo (en ejecución) en un instante dado. Un hilo dentro de un proceso se ejecuta secuencialmente, y cada hilo tiene su propia pila y contador de programa. Un proceso tradicional es igual a una tarea con un solo hilo. Una tarea no hace nada si no hay hilos en ella y un hilo debe estar en una y sólo una tarea.

La conmutación de la CPU entre hilos pares y la creación de hilos es económica respecto de la conmutación de contexto entre procesos. Aunque una conmutación de contexto de hilos también requiere un cambio de conjunto de registros, no hay que realizar operaciones relacionadas con la gestión de memoria. Al igual que cualquier entorno de procesamiento paralelo, el multihilado de un proceso puede introducir problemas de control de concurrencia que requieren el uso de secciones críticas o cerraduras.

La comunicación entre los threads en ejecución se realiza a través del mecanismo llamado **entry,** que permite realizar un "rendezvous" (encuentro) entre dos threads concurrentes en ejecución. En el rendezvous, la información puede ser intercambiada entre las tareas.

A continuación se muestra la especificación para dos *packages*. El primero define una tarea para calcular el factorial de un número positivo y el segundo determina si un número positivo es o no es primo.

```
package Pack_Factorial is
  task type Task_Factorial is                --Specification
    entry Start( F:in Positive );            --Rendezvous
    entry Finish( Result:out Positive );     --Rendezvous
  end Task_Factorial;
end Pack_Factorial;
```

```
package Pack_Is_A_Prime is
  task type Task_Is_Prime is                    --Specification
    entry Start( P:in Positive );               --Rendezvous
    entry Finish( Result:out Boolean );         --Rendezvous
  end Task_Is_Prime;
end Pack_Is_A_Prime;
```

Nota: El *rendezvous* Start se usa para pasar datos a la tarea y el *rendezvous* Finish se usa para retornar el resultado.

Una tarea es creada usando el mecanismo normal de elaboración de Ada. Para crear una instancia de la tarea Task_Factorial se utiliza la siguiente declaración:

```
Thread_1 : Task_Factorial;
```

La tarea comenzará ejecutándose como un thread independiente tan pronto como se ingrese al bloque delimitado por la declaración. Un rendezvous que le pase el número 5 a esta tarea en ejecución se escribirá como:

```
Thread_1.Start(5); --Start factorial calculation
```

Nota: Esto puede pensarse como el envío del mensaje Start con un parámetro de 5 a la tarea Thread_1.

Las tareas descriptas pueden usarse de la siguiente manera:

```
with Ada.Text_Io, Ada.Integer_Text_Io,
     Pack_Factorial, Pack_Is_A_Prime;
use Ada.Text_Io, Ada.Integer_Text_Io,
     Pack_Factorial, Pack_Is_A_Prime;
procedure Main is
     Thread_1 : Task_Factorial;
     Thread_2 : Task_Factorial;
     Thread_3 : Task_Is_Prime;
     Factorial: Positive;
     Prime : Boolean;
begin
     Thread_1.Start(5);            --Start factorial calculation
     Thread_2.Start(7);             --Start factorial calculation
     Thread_3.Start(97);            --Start is_prime calculation
     Put("Factorial 5 is ");
     Thread_1.Finish( Factorial );  --Obtain result
     Put( Factorial ); New_Line;
     Put("Factorial 7 is ");
     Thread_2.Finish( Factorial );  --Obtain result
     Put( Factorial ); New_Line;
     Put("97 is a prime is ");
     Thread_3.Finish( Prime );      --Obtain result
     if Prime then                  --
           Put("True");             -- and print
     else
           Put("False");
     end if;
     New_Line;
end Main;
```

Nota: las tareas comienzan su ejecución tan pronto como el **begin** del bloque en el cual fueron creadas comienza. El rendezvous Start se usa para controlar este comportamiento caprichoso.

Esta es esencialmente una relación cliente-servidor entre el programa principal, el cliente, que requiere un servicio de la tarea servidor.

Reuniendo todo

```
package Pack_Factorial is
    task type Task_Factorial is --Specification
        entry Start( F:in Positive ); --Rendezvous
        entry Finish( Result:out Positive ); --Rendezvous
    end Task_Factorial;
end Pack_Factorial;

package Pack_Is_A_Prime is
    task type Task_Is_Prime is --Specification
        entry Start( P:in Positive ); --Rendezvous
        entry Finish( Result:out Boolean ); --Rendezvous
    end Task_Is_Prime;
end Pack_Is_A_Prime;

with Ada.Text_Io, Ada.Integer_Text_Io,
    Pack_Factorial, Pack_Is_A_Prime;
use Ada.Text_Io, Ada.Integer_Text_Io,
    Pack_Factorial, Pack_Is_A_Prime;
procedure Main is
    Thread_1 : Task_Factorial;
    Thread_2 : Task_Factorial;
    Thread_3 : Task_Is_Prime;
    Factorial: Positive;
    Prime : Boolean;
begin
    Thread_1.Start(5); --Start factorial calculation
    Thread_2.Start(7); --Start factorial calculation
    Thread_3.Start(97); --Start is_prime calculation

    Put("Factorial 5 is ");
    Thread_1.Finish( Factorial ); --Obtain result
    Put( Factorial ); New_Line;

    Put("Factorial 7 is ");
    Thread_2.Finish( Factorial ); --Obtain result
    Put( Factorial ); New_Line;

    Put("97 is a prime is ");
    Thread_3.Finish( Prime ); --Obtain result
    if Prime then --
        Put("True"); -- and print
else
        Put("False");
    end if;
    New_Line;
end Main;
```

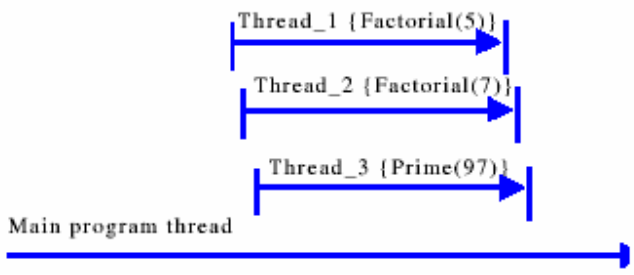
```
package body Pack_Factorial is
    task body Task_Factorial is --Implementation
        Factorial : Positive;
        Answer : Positive := 1;
begin
        accept Start( F:in Positive ) do --Factorial
            Factorial := F;
        end Start;
        for I in 2 .. Factorial loop --Calculate
            Answer := Answer * I;
        end loop;
        accept Finish( Result:out Positive ) do --Return answer
            Result := Answer;
        end Finish;
    end Task_Factorial;
end Pack_Factorial;

package body Pack_Is_A_Prime is
    task body Task_Is_Prime is --Implementation
        Prime : Positive;
        Answer: Boolean := True;
    begin
        accept Start( P:in Positive ) do --Factorial
            Prime := P;
        end Start;
        for I in 2 .. Prime-1 loop --Calculate
            if Prime rem I = 0 then
                Answer := False; exit;
            end if;
        end loop;
        accept Finish( Result:out Boolean ) do --Return answer
            Result := Answer;
        end Finish;
    end Task_Is_Prime;
end Pack_Is_A_Prime;
```

Al ejecutarse el programa se generan los siguientes resultados:

```
Factorial 5 is 120
Factorial 7 is 5040
97 is a prime is True
```

La ejecución del programa anterior puede verse como 3 threads activos:



Una vez iniciado, cada uno de los threads se ejecutarán concurrentemente hasta que el rendezvous Finish es encontrado.

Nota: La presente implementación de la concurrencia dependerá de la arquitectura subyacente, el software y el hardware de la plataforma donde el programa se ejecuta.

Tarea rendezvous

El mecanismo rendezvous se usa para:

- sincronización de 2 threads individuales a fin de intercambiar información.
- sincronización de la ejecución de dos threads.

Un rendezvous es generado por una tarea a través de una declaración **`entry`** y la otra tarea realizando un llamado a este **`entry`**. Por ejemplo, el código para que un rendezvous pase un número `Positive` a la tarea objeto `thread_1` sería:

Programa principal (cliente) que genera el thread 1	**Cuerpo de la tarea Thread_1 (server)**

`Thread_1.Start(5);`	`accept Start(F:in Positive) do`
	`  Factorial := F;`
	end Start;

Para producir este efecto, uno de los threads de control será suspendido hasta que el otro thread lo capture. Entonces en el rendezvous (encuentro), el dato, en este caso el número 5, es transferido entre las tareas. El código entre **do** y **end** es ejecutado con la tarea cliente suspendida. Después que el código entre **do** y **end** ha sido ejecutado ambas tareas prosiguen su ejecución independiente.

Este rendezvous entre las dos tareas se ilustra a continuación con un rendezvous entre la tarea del programa principal y una instancia de la tarea Factorial:

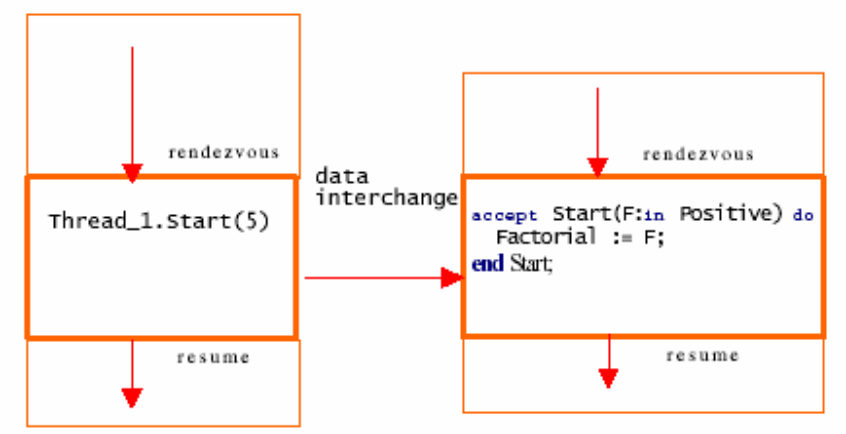


Ilustración de un rendezvous

Otras variantes del rendezvous son:

Variante	Cliente	Tarea servidor
Ninguna información pasada	`Thread_1.Start;`	**accept** `Start;`
Ninguna información pasada pero Thread_1 ejecuta sentencias durante el rendezvous	`Thread_1.Start;`	**accept** `Start` **do** `Statements;` **end** `Start;`

La implementación de la tarea

En el cuerpo del package `Pack_Factorial` mostrado a continuación, la tarea `Task_Factorial` usa dos puntos de rendezvous:

- `Start` para obtener el dato con el cual trabajar.
- `Finish` para entregar el resultado.

Cuando el thread de control de la tarea alcanza el final del cuerpo de la tarea, la tarea finaliza. Cualquier intento de rendezvous con una tarea concluida, generará la excepción `Task_Error`.

```
package body Pack_Factorial is
      task body Task_Factorial is --Implementation
            Factorial : Positive;
            Answer : Positive := 1;
      begin
            accept Start( F:in Positive ) do --Factorial
               Factorial := F;
            end Start;
            for I in 2 .. Factorial loop --Calculate
```

```
               Answer := Answer * I;
            end loop;
            accept Finish( Result:out Positive ) do --Return answer
               Result := Answer;
            end Finish;
      end Task_Factorial;
end Pack_Factorial;
```

Igualmente, la tarea `Task_Is_Prime` en el package `Pack_Is_A_Prime` recibe y entrega datos a otro thread de control.

```
package body Pack_Is_A_Prime is
      task body Task_Is_Prime is                --Implementation
         Prime : Positive;
         Answer: Boolean := True;
      begin
         accept Start( P:in Positive ) do        --Factorial
            Prime := P;
         end Start;
         for I in 2 .. Prime-1 loop              --Calculate
            if Prime rem I = 0 then
               Answer := False; exit;
            end if;
         end loop;
         accept Finish( Result:out Boolean ) do   --Return answer
            Result := Answer;
         end Finish;
      end Task_Is_Prime;
end Pack_Is_A_Prime;
```

Excepciones

El manejo de excepciones tiene por objetivo tratar con errores u otras situaciones excepcionales que se producen durante la ejecución de un programa.

Una excepción representa una clase de situación excepcional; la ocurrencia de tal situación (en tiempo de ejecución) es llamada *ocurrencia de una excepción*. Levantar una excepción consiste en abandonar la ejecución normal del programa así como dirigir la atención al hecho que la situación correspondiente ha motivado. La realización de algunas acciones en respuesta a la elevación de una excepción es llamada *manejo de la excepción*.

Una **declaración de excepción** declara un nombre para una excepción. Una excepción es elevada explícitamente por una cláusula **raise** o por una falla producida en el entorno de ejecución. Cuando una excepción ocurre, el control puede ser transferido a un manejador de excepciones.

Una excepción es una manifestación de un cierto tipo de error:

– cuando se produce un error, se *eleva* la excepción correspondiente
– se abandona la ejecución normal y se pasa a ejecutar un *manejador* asociado a la excepción
– se busca un manejador en el mismo cuerpo o bloque
– si no lo hay, la excepción se *propaga* al nivel superior, bloque exterior o punto de invocación de un subprograma
– si no se encuentra ningún manejador, se termina el programa.

Nombres de excepciones

Se pueden declarar excepciones en cualquier zona declarativa (no son objetos)

```
Sensor_Failure : exception;
```

– Se elevan con una instrucción **raise**:

```
raise Sensor_Failure;
```

Algunas excepciones están predefinidas:

```
Constraint_Error : exception;
Program_Error : exception;
Storage_Error : exception;
Tasking_Error : exception;
```

– Se elevan automáticamente por el entorno de ejecución, o explícitamente con **raise.**

Manejadores de excepciones

Los manejadores se declaran al final de un bloque o cuerpo:

```
begin
...
exception
when Constraint_Error => ...;
when Sensor_Failure => ...;
when Storage_Error | Program_Error => ...;
when others => ...;
end;
```

Un manejador es una secuencia de instrucciones: cuando termina, se devuelve el control al punto de invocación del bloque o cuerpo que falló.

Bibliografía

Ada 95 Reference Manual

Object-oriented Software in Ada 95 Second Edition - Michael A. Smith

GNAT User's Guide