dit
UPM

Lenguajes de programación para sistemas de tiempo real
Introducción a Ada

Juan Antonio de la Puente
DIT/UPM

Objetivos

- Haremos un repaso de las características básicas de un lenguaje de programación: **Ada 95**
- Veremos un subconjunto mínimo que permite realizar programas en pequeña escala
- Suponemos que se conoce algún lenguaje estructurado (como Pascal o Modula)

Índice

Lenguajes de programación

- Un lenguaje de programación de sistemas de tiempo real debe facilitar la realización de sistemas
  - concurrentes,
  - fiables,
  - puntuales.
- Las características más importantes para la realización de programas en pequeña escala son
  - léxico
  - tipos de datos y objetos
  - instrucciones
  - subprogramas
  - estructura de programas

Ada

- Es un lenguaje imperativo, descendiente de Pascal
  - estructura en bloques
  - fuertemente tipado
- Es un lenguaje pensado para realizar sistemas empotrados de gran dimensión
- Dos versiones normalizadas
  - Ada 83 (ISO 8652:1987)
  - **Ada 95** (ISO 8652:1995)
- Desarrollado por iniciativa y bajo la supervisión del **DoD** (*Department of Defense*) de los EE.UU.

Núcleo y anexos

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

  (no añaden sintaxis ni vocabulario al lenguaje)

Índice

Léxico

- Los *identificadores* pueden contener letras, dígitos o el carácter '_'

  Ejemplos: `Sensor  Tiempo_de_Activación`

- No hay distinción entre mayúsculas y minúsculas
- Hay *palabras reservadas* que no se pueden usar como identificadores
- Los números pueden contener el carácter '_'

  Ejemplos: `123 150_000 3.141_592_654 1.7E-5`

- Los comentarios empiezan con "--" y van hasta el final de la línea

Índice

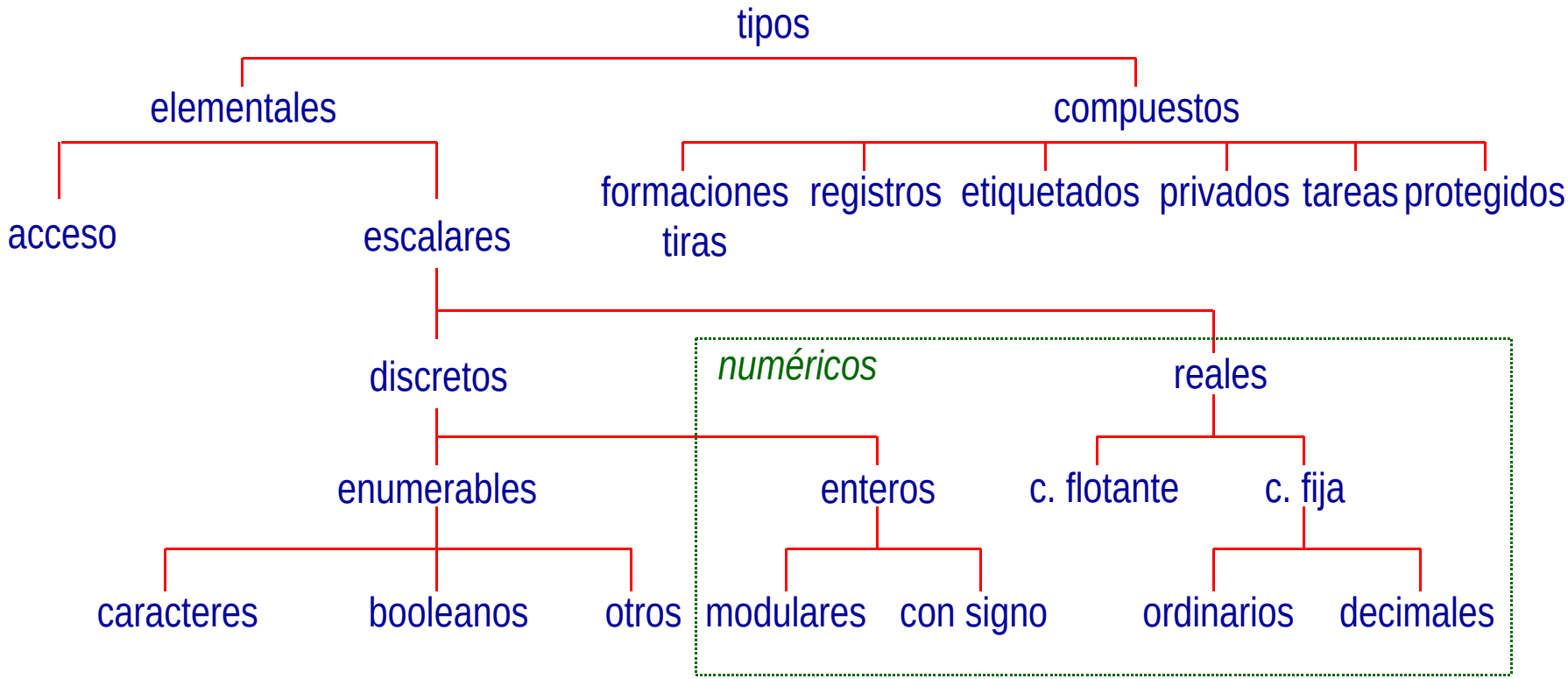
Tipos de datos

- Un *tipo de datos* es un conjunto de *valores* con un conjunto de *operaciones primitivas* asociadas

- Tipado fuerte
  - No se pueden usar valores de un tipo en operaciones de otro tipo sin efectuar una *conversión de tipo* explícita
  - Las operaciones dan siempre resultados del tipo correcto

- Una *clase* es la unión de varios tipos con características comunes

Clasificación de los tipos de datos



Tipos discretos

- Enumerables

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

- Enteros
  - Con signo

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

  - Modulares

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

Tipos reales

- 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;
```

Ejemplos

```
type Index is range 1 .. 100;                 -- entero
type Length is digits 5 range 0.0 .. 100.0; -- coma flotante

First, Last : Index;
Front, Side : Length;

Last := First + 15;                  -- correcto
Side := 2.5*Front;                   -- correcto
Side := 2*Front;                     -- incorrecto
Side := Front + 2*First;             -- incorrecto
Side := Front + 2.0*Length(First); -- correcto
```

Tipos compuestos (1)

- Formaciones

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

Elementos:

```
V : Voltages;

V(5) := 1.0;
V := (1.0, 0.0, 0.0, 0.0, 2.5,
      0.0, 0.0, 0.0, 0.0, 0.0);
V := (1 => 1.0, 5 => 2.5, others => 0.0);
```

Tipos compuestos (2)

- Registros

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

Elementos:

```
X : State

X.Reference := 0.0;
X := (Automatic, 0.0);
X := (Operating_Mode => Automatic,
      Reference      => 0.0);
```

Tipos acceso

- Los objetos de tipos acceso designan valores de otros tipos

```
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;
```

  Las variables de acceso se inician a **`null`** si no se dice nada

- Acceso al objeto dinámico:

```
S.Operating_Mode := Manual;
S.all := (Operating_Mode => Manual, Reference =>
  0.0);
S     := new State := (Manual, 0.0);
```

  – No se puede operar con los valores de los tipos acceso
  – No pueden quedar punteros “colgando”

Declaraciones

- Asocian nombres con definiciones de
  - tipos

    ```
    type Real is digits 8;
    ```
  - objetos

    ```
    X : Real := 0.0;
    J : constant Complex := (0.0, -1.0);
    ```
  - números

    ```
    Pi : constant := 3.141_592_654;
    ```
  - subprogramas
  - otras entidades
- ¡Ojo! Todas terminan en ';'

Elaboración

- Las declaraciones se colocan en *zonas declarativas*
- Al entrar en la zona declarativa se *elaboran*
  - Puede ser al arrancar el programa o durante la ejecución
- La elaboración consiste en la creación de la entidad declarada, seguida por la realización de las operaciones iniciales asociadas a la misma
  - Ejemplos:
    - » iniciación del valor de una variable
    - » Asignación de memoria para crear un objeto dinámico

Índice

Instrucciones

- Simples
  - Asignación
  - Llamada a procedimiento
  - Nula
- Compuestas
  - Secuencia
  - Bloque
  - Selección
  - Iteración
- ¡Ojo! Todas las instrucciones terminan con ';'

Instrucciones simples

- Asignación

```
U := 2.0*V(5) + U0;
```

  - El tipo de la expresión debe ser el mismo que el de la variable
  - No hay conversión implícita de tipos

- Llamada a procedimiento

```
Get(V);
```

- Instrucción nula

```
null;
```

Secuencia

```
Get(V);
U := 2.0*V(65) + U0;
```

Sintaxis:

```
secuencia ::= instrucción {instrucción}
```

¡Ojo! No se puede omitir ';' en la última instrucción

Bloque

```
declare
  V : Voltages              -- variable local al bloque
begin
  Get(V);
  U := 2.0*V(65) + U0;
end;                        -- V deja de existir aquí
```

Sintaxis:

```
bloque ::= [identificador :]
             [declare
                {declaración}] -- parte declarativa
              begin
                secuencia
              end [identificador];
```

Selección

```
if T <= 100.0 then
  P := Max_Power;
elsif T >= 180.0 then
  P := 0.0;
else
  P := Control(R,t);
end if;
```

Sintaxis

```
selección-if ::=
  if expresión-booleana then
     secuencia
  {elsif expresión-booleana then
     secuencia}
  [else
     secuencia]
   end if;
```

Selección por casos

```
case Day is
  when Monday                => Start_Week;
  when Tuesday .. Thursday => Continue_Work;
  when Friday  .. Saturday => End_Week;
  when others                => Relax;
end case;
```

Sintaxis

```
selección-case ::=
  case expresión-discreta is
    alternativa
   {alternativa}
  end case;
alternativa ::= when lista => secuencia
lista       ::= opción {|opción}
opción      ::= expresión |intervalo|others
```

¡Ojo! Hay que cubrir todos los valores del tipo discreto

Iteración (1)

Hay tres clases de bucles

```
for I in 1..10 loop
  Get(V(I));
end loop;
```

¡Ojo! El índice del bucle **for** solo está definido en el cuerpo del bucle

```
while T <= 50.0 loop
  T := Interpolatio(T);
end loop;
```

```
loop
  Get(T);
  P := Control(R,T);
  Put(T);
end loop;
```

Iteración (2)

Sintaxis

```
bucle ::= [identificador :]
            [esquema-de-iteración] loop
               secuencia
             end loop [identificador];

esquema-de-iteración ::=
   for índice in [reverse] intervalo-discreto
  |while expresión-booleana
```

Iteración (3)

Se puede salir del bucle con una instrucción **exit**

```
I := 1;
loop
  Get(U);
  exit when U > 80.0;
  V(I) := U;
  I := I+1;
end loop;
```

Sintaxis

```
exit[identificador] [when expresión-booleana];
```

- La ejecución continúa inmediatamente después del bucle
- Si hay bucles anidados, se sale del interior, o del que tiene el identificador que se indica

Transferencia de control

Sintaxis

```
goto etiqueta;
etiqueta ::= <<identificador>>
instrucción-etiquetada ::= etiqueta instrucción
```

- Hay limitaciones para aumentar la seguridad
  - La instrucción etiquetada tiene que estar en la misma secuencia y con el mismo grado de anidamiento que la de transferencia
  - No se puede salir de un subprograma ni de otras entidades de programa

Índice

Subprogramas

- Hay dos tipos de subprogramas
  - **procedimiento**: abstraccción 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

Declaración de subprograma (1)

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;
```

Declaración de subprograma (2)

Sintaxis

```
declaración-de-subprograma ::=
  especificación-de-subprograma;

especificación-de-subprograma ::=
   procedure nombre [parámetros-formales]
  |function  nombre [parámetros-formales] return
  tipo

parámetros formales ::= (definición{, definición})
definición ::= lista : modo tipo [::= valor-por-
  defecto]
lista ::= identificador{, identificador}
modo ::= [in] | out | in out
valor-por-defecto ::= expresión
```

Declaración de subprograma (3)

- 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
- Puede haber subprogramas *homónimos*
  - Se distinguen por sus parámetros formales (y en el caso de las funciones por el tipo del resultado)
  - El compilador genera la llamada al subprograma adecuado

Cuerpo de subprograma (1)

- Se coloca en una zona declarativa

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

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

Cuerpo de subprograma (2)

- Sintaxis

```
cuerpo-de-subprograma ::=
  especificación-de-subprograma is
    {declaración}
  begin
    secuencia;
  end nombre;
```

  - El cuerpo empieza con una zona declarativa donde se declaran entidades locales
  - El cuerpo de una función debe contener al menos una instrucción **return**

Llamada a subprograma (1)

Es una instrucción simple

```
Increment(X,2);                          -- asociación
  posicional
Increment(Value => X, Step => 2); -- asociación
  nombrada
Increment(X);                            -- Step => 1
  (defecto)

W := 2*Minimum(U,V);
```

Puede formar parte de cualquier secuencia

Llamada a subprograma (2)

Sintaxis

```
llamada-a-subprograma ::=
  nombre; | nombre parámteros-reales;
parámetros-reales ::= (asociación{, asociación})
asociación ::= [nombre-formal =>] parámetro-real
parámetro-real ::= expresión | nombre-de-variable
```

- Los parámetros formales de modo **in** se asocian a expresiones
- Los de modo **out** o **in out** se asocian a variables
- Con asociación nombrada los parámetros pueden ir en cualquier orden

Índice

Estructura de programas (1)

- 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) unidad 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*

Estructura de programas (2)

- 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};
```

Paquetes predefinidos

- Operaciones numéricas
  - Ada.Numerics, Ada.Numerics.Generic_Elementary_Functions
- Operaciones con caracteres y tiras
  - 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

Entrada y salida

- Ada.Text_IO

```
procedure Get      (Item : out String);
procedure Put      (Item : in  String);
procedure Put_Line (Item : in  String);
procedure New_Line;
```

- Ada.Integer_Text_IO

```
procedure Get (Item : out Integer);
procedure Put (Item : in  Integer);
```

- Ada.Float_Text_IO

```
procedure Get (Item : out Float);
procedure Put (Item : in  Float);
```

Ejemplo

```
with Ada.Text_IO;          use Ada.Text_IO;
procedure Hello is
begin
  Put_Line("Hello!");
end Hello;
```

Compilación con GNAT

- Se parte de un fichero `hello.adb`. Para compilar se hace:

```
$ gcc -c hello.adb
```

  - resultado: ficheros `hello.o y hello.ali`
- Montaje y enlace :

```
$ gnatbind hello.ali
$ gnatlink hello.ali
```

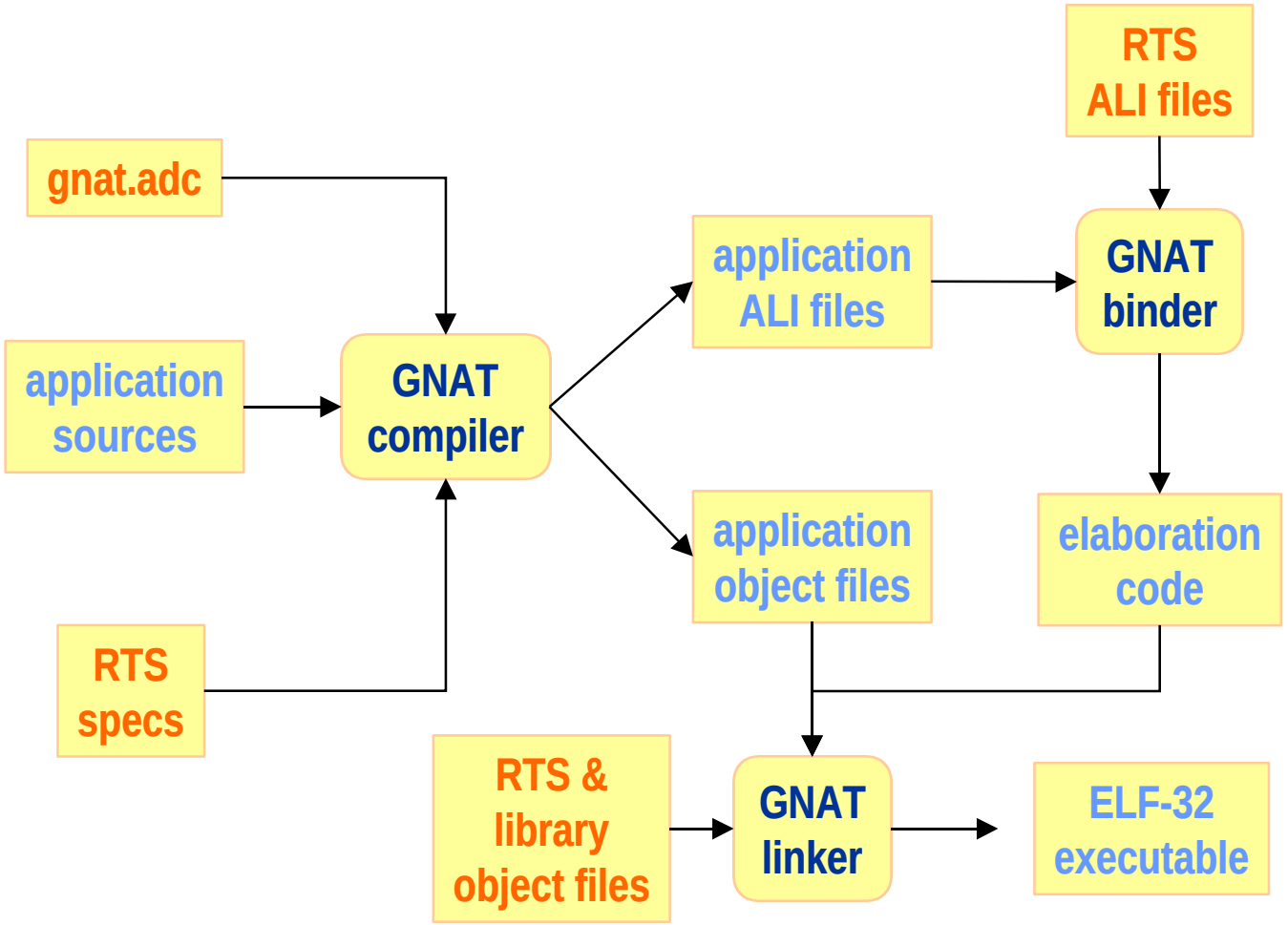
  - resultado: fichero `hello` (linux) o `hello.exe` (Windows)
- Se puede hacer todo de una vez:

```
$ gnatmake hello.adb
```

  - compila todo lo que haga falta, no sólo `hello.adb`
- Para ejecutarlo se hace (en linux)

```
$ ./hello
```

Compilación y montaje



Índice

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 1 .. Integer'Last;
```

- Las operaciones con valores de distintos subtipos de un mismo tipo están permitidas
- Las restricciones se comprueban al compilar o al ejecutar

Ejemplos

```
A : Small_Index := 1;
B : Big_Index;
C : Index;

A := 3;      -- correcto
A := 6;      -- error
A := B;      -- error
A := C;      -- error si C > 5
A := A + 1; -- error si A > 4
```

Tipos derivados

Un tipo derivado es una copia de otro tipo, cuyos valores y operaciones *hereda*

```
type Surface is new Float;
```

– Se puede imponer una restricción en el conjunto de valores

```
type Length  is new Float range 0.0 .. 100.0;
type Width   is new Float range 0.0 ..  20.0;
S : Surface; L : Length; W : Width; -- tipos distintos
```

– Los valores no son compatibles, pero se pueden convertir

```
S := L*W;                              --  incorrecto
S := Surface(Float(L)*Float(W));       --  correcto
```

– Las restricciones se comprueban al compilar o al ejecutar

Formaciones irrestringidas

- Se pueden declarar tipos formación con un intervalo de índices indefinido (formaciones irrestringidas)

```
type Measurements is
  array (Positive range <>) of Voltage;
```

  - El tipo **String** está predefinido:

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

- Al declarar objetos hay que restringir los índices

```
M : Measurements;                    -- incorrecto
M : Measurements (1 .. 10);          -- correcto
M : Measurements := (1..10 => 0.0);  -- correcto
S : String (1 .. 25);                -- correcto
S : String := "ejemplo";             -- correcto
```

Formaciones dinámicas

- También se puede declarar formaciones con intervalos dinámicos (que se evalúan al elaborar la declaración)

```
type Measurements is
  array (1 .. No_Of_Inputs) of Voltage;
```

- Las expresiones que aparecen en los límites tiene que estar definidas en el momento de la elaboración (pero no necesariamente al compilar)

Registros con discriminantes

- Un discriminante es un componente de un registro que permite parametrizar los objetos del tipo

```
type Variable is (Temperature, Pressure);
type Measurement (Kind: Variable) is
  record
    Value : Voltage;
  end record;

T : Measurement(Temperature);
P : Measurement := (Kind => Pressure, Value => 2.5);
```

- El discriminante tiene que ser de un tipo discreto
- No se puede cambiar una vez asignado

Registros con variantes

- Se puede usar un discriminante para declarar tipos registro con partes variantes

```
type Measurement (Kind: Variable) is
  record
    Value : Voltage;
    case Kind is
      when Temperature => Too_High  : Boolean := False;
      when Pressure    => Sensor_Id : Sensor_Range;
    end case;
  end record;
T : Measurement(Temperature);
P : Measurement := (Pressure, 2.5, 4);
T.Too_High := True;
Id := P.Sensor_Id;
```

Conversión de tipos

- Se pueden convertir valores de un tipo a otro
  - entre tipos numéricos
  - entre formaciones con la misma estructura y tipo de elementos
  - entre registros con los mismos discriminantes y tipo de componentes
  - entre un tipo y sus derivados (pero no éstos entre sí)
- El nombre del tipo al que se convierte se usa como si fuera una función

```
I : Integer; S : Surface;
I := Integer(S);
S := Surface(Float(L)*Float(W));
```

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 Language Reference Manual*
- *Ada 95 Rationale*
- John Barnes – *Programming in Ada 95*
- *GNAT User's Guide*