Ada 95

- Historia, capacidades
- Elementos léxicos
- Tipos de datos y variables
- Secuencias de control
- Unidades de programa
- Entrada / Salida
- Proceso Paralelo
- Unidades genéricas
- Excepciones
- Programación en tiempo real

Historia

- Concurso del Dep. Def. de E.E.U.U.
  - 1976. Conjunto de requerimientos. Como base se utilizo Pascal, PL/1 y Algol
  - 1977. Revisión del documento. 17 propuestas que se seleccionaron 4, todas basadas en Pascal
  - 1979. Se revisan los diseños de los finalistas y se selecciona el ganador: Honeywell Bull.
  - 1980. Test y evaluación que produjeron algunos cambios hasta publicar el LRM
- Se estandarizó por el ANSI en 1983 (ISO 1987)
- Una revisión posterior llevó al Ada 9X que concluyó en el Ada 95
- Tutorials: http://www.adahome.com/Tutorials/
- LRM: http://www.adahome.com/rm95/

Capacidades del Ada 95

- Definición de paquetes (módulos) con tipos, objetos y operadores relacionados
- Paquetes genéricos
- Manejo de excepciones
- Programación concurrente. Tareas
- Control sobre la representación de los datos
- Librerías predefinidas con infinidad de funcionalidades
- Soporta programación orientada a objetos
- Interfaces para programación mixta con otros lenguajes

Elementos léxicos

- El primer programa

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

- Identificadores

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

- Literales numéricos

```
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"
```

Elementos léxicos (cont)

- Caracteres y cadenas
  - Los caracteres simples se encierran entre (‘)
  - Las cadenas se encierran entre (“)
  - Los caracteres de control por su código
- Sentencias
  - Acciónes:
    - Llamada a un subprograma
    - Asignación con el símbolo ‘:=‘
  - Ejecución condicional y bucles
  - Terminan con el carácter ‘;’
  - Los comentarios comienzan por ‘--’

Tipos de datos y variables

- Tipos nuevos o derivados de ya existentes
- Definición:
  - Comienza por la palabra reservada `type`
  - Sigue el identificador del tipo
  - La palabra reservada `is`
  - La definición del tipo terminada con ‘;’
- Las variables se definen con el carácter ‘:’ seguida del identificador del tipo
- Ejemplo:

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

Datos simples

- Integer
  - Rango: –32767 .. 32767
  - Operadores “+”, “-”, “*”, “/”, “**”
- Float
- Boolean
  - Valores; True y False
  - Operadores: “and”, “or”, “xor”, “not”
- Character y Whide_Character
- Comparación: “=”, “<” , “<=” , “>” , “>=” , “/=”
- Ada permite especificar los tipos (rango y precisión):

```
type Mes is range 1..12;
type Voltaje is delta 0.1 range 0.0 .. 10.0;
type Peso is digits 10;
```


Tipos enumerados

- Definición

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

- Propiedades
  - Relaciones de comparación
  - Primero y último
  - Anterior y siguiente
  - Obtener el ordinal de un valor
  - Obtener el valor a partir del ordinal
  - Convertir a string y viceversa

```
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”
```

Rangos y subrangos

- En Ada los tipos diferentes son totalmente independientes, y no se pueden mezclar en las expresiones y paso de parámetros
- Se pueden definir tipos compatibles con otros definidos utilizando la palabra `subtype`

```
type Dia is (Lun, Mar, Mie, Jue, Vie, Sab, Dom);
type Columna is range 1..72;
type Fila is range 1..24;

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

Datos compuestos

- Tablas: Colección de tipos iguales
- Estructuras: Colección de tipos distintos

Tablas

- Se definen con la palabra `array` y uno o varios tipos enumerados o rangos que se utilizan como índices

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

- 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 (0,0).
```

Tablas (cont.)

- Tablas sin rango
  - El rango queda sin definir, utilizando el símbolo “<>”
  - El rango se define al declarar la variable

```
type cadena is array (integer range <>) of character;
str: cadena(1..100);
```

- Strings de longitud fija

```
S: String(1..10);
A: String := “String con valor”; -- String de longitud 16
```

  - Asignar un valor: `S := “Ejemplo  “`
  - Extraer un carácter: `S(3)` o modificarlo `S(3):= ‘s’`
  - Concatenar dos strings con el operador “&”
  - Pasar como parámetros a un procedimiento
- String de longitud variable: `Bounded_string` y `Unbounded_string`

Estructuras

- La declaración del tipo comienza con `record` y termina con `end record`

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

- Se pueden dar valores iniciales tanto al declarar el tipo como las variables

```
type Complex is
  record
    Real_Part, Imaginary_Part : Float := 0.0;
  end record;
X: Complex;
X.Real_part := 1.0;
```

Estructuras parametrizadas

- En el nombre del tipo se puede indicar un parámetro y utilizarlo en la definición
- El parámetro puede tener un valor por defecto
- El valor del parámetro se indica al declarar las variables

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

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


Estructuras alternativas

- Son estructuras variables cuyos campos dependen de un campo o selector
- El selector se usa en una construcción tipo case

```
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 );
```

Estructuras dinámicas

- Los datos `access` (punteros) permite crear estructuras que pueden modificar su tamaño y el número de elementos

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

- Con el valor `null` no hacen referencia a ninguna variable
- Admiten operadores “=” y “/=”
- Las estructuras dinámicas se crean utilizando campos de tipo `access`
- Las variables nuevas se crean con `new`
- La referencia se obtiene con el atributo `‘Access`
- El acceso a los campos de una estructura apuntada por un access se hace como a los campos de una variable
- Con el campo `.all` se hace referencia a toda la estructura

Estructuras dinámicas (cont.)

```
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;

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

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

Estructuras dinámicas (cont.)

- Para poder asignar la referencia de una variable a otra de tipo acceso hay que utilizar el atributo `'Access` sobre la variable
- Para que un `access` pueda recibir la referencia de una variable:
  - El access se debe definir con `all`
  - La variable se debe definir con `aliased`

```
...
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;
...
```


Inicialización y constante

- Se pueden asignar un valor a las variables en su declaración (también datos compuestos y tablas)
- También se pueden definir valores iniciales en la declaración del tipo

```
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 );
```

- Se puede declarar una variable como constante y una constante sin tipo

```
PI: constant Float := 3.1415926;

V_MAX: constant := 1_000;
```

Secuencias de control

- Permiten romper la linealidad del programa
- Básicamente son ejecución condicional y bucles
- A diferencia que Pascal y C van siempre parentizadas
- Ejecución condicional
  - Construcción If
  - Construcción Case
- Bucles
  - Bucles Simples
  - Bucles con condición
  - Bucles Iterativos

Construcción If

- Se evalúa un código u otro dependiendo de una expresión booleana
- La construcción “elsif” evita múltiples “end if” finales

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

Construcción Case

- Se ejecutan varias alternativas a partir de una expresión con resultado escalar
- Se debe indicar todos los valores posibles, o usar “others”

```
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"
```

```
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;
```


Bucles simples y con condición

- Los bucles simples son bucles infinitos, de los que se puede salir con la sentencia “exit” o “exit when”

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

- Los bucles con condición incluyen una condición con la sentencia “while” que se evalúa antes de cada iteración

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

Bucles iterativos

- Utilizan un índice local que varia entre un rango de valores: `for` *`index`* `in .. do`
- Los valores se pueden recorren en sentido inverso utilizando `in reverse`

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


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


Unidades de programa

- Un programa se compone de una o mas unidades de programa
  - Subprograma: Secuencia de instrucciones que sigue un algoritmo
  - Paquete: Colección de entidades
  - Tarea: Unidad de ejecución que se ejecuta en paralelo con el resto de unidades de ejecución
  - Unidad protegida: Coordina datos compartidos entre distintas unidades de ejecución
  - Unidad genérica: Soporta la ejecución de código reusable
- Los paquetes son las unidades de programación más importantes

Declaración y Cuerpo

- Las unidades de programa consisten en dos partes
  - Declaración: Contiene la información visible por otras unidades de programa (.ads)
  - Cuerpo: Detalles de la implementación que no necesita ser visible (.adb)
- Las dos partes se guardan en ficheros separados, salvo en dos casos:
  - En los subprogramas no es obligatoria la declaración separada
  - Algunos paquetes pueden no tener detalles de implementación
- La separación en dos partes facilita la descomposición del diseño y las pruebas

Subprogramas (declaración)

- Pueden ser procedimientos o funciones
- Las funciones se distinguen en que devuelven un valor y los parámetros sólo se pasan por valor

```
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
```

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

Subprogramas (cuerpo)

- Se compone de una parte declarativa y de una secuencia de instrucciones
- La parte declarativa contiene tipos, variable y/o subprogramas

```
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] ";"
```

- En las funciones el valor que se devuelve se indica con la sentencia `return`, que provoca la salida de la función
- Los parámetros pueden inicializarse con un valor por defecto

Subprogramas (cont.)

```
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;
```

- Se pueden definir subprogramas con el mismo nombre pero con distintos parámetros (sobrecarga)
- Se pueden definir funciones que correspondan con operadores usando el nombre del operador entre “”

```
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;
```

Paquetes

- Proporcionan una unidad básica en la descomposición de programas
- Facilita el desarrollo en equipo
- Para acceder a la parte declarativa de un paquete, se debe referenciar con la cláusula `with` al principio de la unidad
- A los identificadores de un paquete se accede de la forma `Paquete.identificador`
- Usando la cláusula `use` se puede omitir el nombre del paquete, siempre que no haya ambigüedad

Paquetes (declaración)

- Puede contener declaraciones de variables, tipos y subprogramas (ficheros .ads)
- Se puede limitar el uso de algunos datos con la cláusula `private` y `limited private` para ocultar los detalles de su implementación

```
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" ";"
```

Paquetes (cuerpo)

- Contiene los detalles de implementación de los subprogramas definidos en la parte declarativa
- Puede contener declaraciones de otros elementos que sólo se vayan a utilizar dentro del paquete
- Opcionalmente puede contener código de inicialización

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


Unidades de Compilación

- Contiene la parte declarativa o el cuerpo de una unidad de programa, precedidas por las cláusulas de contexto `with` o `use` que necesite
- Un programa está formado por varias unidades de compilación. Con las cláusulas de contexto `with` se resuelven las dependencias
- Conviene poner una unidad de programa por unidad de compilación

```
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
```

Entrada / Salida básica

- Se implementa en el paquete `Text_IO`
- Soportan operaciones sobre ficheros de texto con el tipo `File_Type`
- Antes de usar un fichero debe ser creado o abierto con los procedimientos `Create` o `Open`
- Están definidos `Current_Input` y `Current_Output`

```
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);
```


Entrada / Salida (ejemplo)

```
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;
```

Otras funciones

```
procedure Get(Item : out String);
procedure Get_Line(Item : out String; Last : out Natural);
procedure Put(Item : in  String);
procedure Put_Line(Item : in  String);
Procedure New_Line(Spacing : in Positive_Count := 1);
Procedure Skip_Line (Spacing : in Positive_Count := 1);
Function End_Of_Line return Boolean;
Function End_Of_File return Boolean;
Function Line return Positive_Count;
```

Entrada / Salida (ejemplo)

Programa de ejemplo que copia solo las 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;
```


Proceso paralelo

- Ada permite declarar tareas concurrente, con un comportamiento similar a los threads
- Ofrece unos mecanismos de comunicación y sincronización:
  - Esperar a que termine una tarea
  - Enviar un mensaje a una tarea
  - Utilizar objetos protegidos
  - Comunicarse usando variables globales
- El programa principal funciona como una tarea más.
- A diferencia que en los procedimiento, en las tareas se debe definir la especificación y el cuerpo

Tareas

```
--
-- 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;
```

- Ada permite declarar tipos de tareas

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

- El tipo podría tener un discriminante que se definiría al declarar la variable, a modo de parámetro para la tarea

Comunicación y sincronización entre tareas

- Las tareas se pueden comunicar por un mecanismo de colas de mensajes asociados a la tarea receptora
- La estructura de los mensajes de cada cola se definen como los parámetros de un procedimiento con una sentencia `entry` en la parte declarativa
- La lectura de los mensajes se hace con una sentencia `accept` en la parte del cuerpo
- En envío es con cita extendida, lo que facilita la comunicación bidireccional usando parámetros por referencia
- Si no se definen parámetros en el `entry` el mecanismo actúa sólo como sincronización

Comunicación y sincronización (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;
```


Comunicación y sincronización (cont.)

- Propiedades
  - Sincronización: Mecanismo de cita extendida
  - Comunicación: Intercambio de información
  - Exclusión mutua: Solo se trata un mensaje de la cola a la vez
- Espera selectiva
  - Permite realizar varios `accept` simultáneamente
  - Opcionalmente se puede indicar un time-out con un `or delay` en lugar del `or accept` o un `else` para cuando no hay mensajes en espera

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

Espera selectiva condicional

- Se pueden indicar condiciones en las alternativas del `select` para que no sean tratados esos mensajes
- Siempre debe haber al menos un `accept` abierto

```
Dos_citas: boolean;
...

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


Tipos protegidos

- Son tipos con sus datos protegidos ante el acceso concurrente
- Son una forma avanzada de semáforos y monitores
- Sólo se puede acceder a los datos a través de funciones protegidas:
  - Funciones, con acceso de solo lectura
  - Procedimientos, con acceso de lectura / escritura
  - Entradas, equivalente a los procedimientos pero con una barrera para permitir el acceso
- Son mecanismos muy eficientes. Las operaciones deben ser cortas
- Se pueden usar como instancias únicas o declararlos como tipos

Tipos protegidos (ejemplo)

```
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;

Control : Mutex;

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


Unidades genéricas

- Las unidades genéricas permiten realizar subprogramas o paquetes con elementos que realizan operaciones con elementos (tipos o valores) que no están determinados
- Estos tipos o elementos se definen cuando se va a utilizar la unidad genérica creando una unidad de programa concreta
- Si en la unidad genérica se utiliza alguna operación sobre el tipo abstracto, esta debe estar definida
- Un ejemplo de subprograma genérico podría ser uno que intercambiara el contenido de dos variables

```
-- 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;
```

Unidades genéricas (cont.)

- Para crear una unidad genérica se ha de escribir utilizando un número reducido de nombres de tipo genéricos, declarándolos como `private`
- La lista de nombres se llaman parámetros formales
- Aunque la declaración es similar, no se deben confundir los tipos genéricos con los tipos privados
- Para utilizar una unidad genéricas se debe crear una nueva a partir de la versión genérica, definiendo lo que serán los parámetros formales (instanciar)
- Cuando se instancia un genérico el paso de los tipos se hace de la misma forma que en una llamada normal a un procedimiento

Unidades genéricas (ejemplo)

```
-- 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;

procedure Swap is new Generic_Swap(Integer);

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

Parámetros formales

- En la declaración de los parámetros formales se pueden incluir varios tipos de elementos
  - Variables de cualquier tipo: Objetos formales

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

  - Cualquier tipo: Tipos formales

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

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

  - Paquetes que son instancias de otros paquetes: Paquetes formales

Tipos y objetos abstractos

- Los objetos abstractos GADO permiten instanciar objetos a partir de un paquete genérico
- En los tipos abstractos GADT al instanciar el paquete se crea un tipo abstracto que permite obtener los objetos
- 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

Ejemplo GADO

```
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;

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;
```


Ejemplo GADT

```
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;
```

Excepciones

- Una excepción representa en Ada un error grave
- La acción por defecto es detener el programa
- La acción de una excepción se puede modificar declarando un manejador
- Ada tiene un número predefinido de excepciones. La mas común es `Constraint_Error` que ocurre cuando una variable toma un valor no permitido
- Las comprobaciones del lenguaje para lanzar excepciones se pueden eliminar para aumentar la eficiencia
- Algunos paquetes definen sus propias excepciones, como es el caso del paquete `Text_IO`

Declaración de excepciones

- Las excepciones nuevas deben ser declaradas como etiquetas

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

- Para lanzar una excepción se usa una sentencia raise

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

- Los manejadores de excepción se declaran en la parte de manejadores de un bloque

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

Excepciones (cont.)

```
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;
```

- Excepciones predefinidas más comunes

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

Excepciones en `Text_IO`

```
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;
```

Relojes en Ada

- Ada ofrece dos paquete que dan acceso a un reloj y permiten operar con el tiempo
  - Ada.Calendar
  - Ada.Real_Time
- Las operaciones mas 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 paquetes el tiempo actual se obtiene con la función:

```
function Clock return Time;
```


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`
- El tiempo se define en un calendario a partir de los tipos y funciones:

```
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;
```

Paquete calendar (cont.)

```
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;
```


Paquete Real_Time

- Define el tipo `Time_Span` equivalente al `Duration` para representar un intervalo
- 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`
- 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;
```

Paquete Real_Time (cont.)

- 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;
```


Retardos y actividades periódicas

- Ada ofrece la primitiva `delay` para realizar un retardo. El parámetro es del tipo `duration`
- Pero no es adecuado para tareas periódicas pues no se tiene en cuenta el tiempo de ejecución en cada iteración

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

- Este problema se resuelve realizando un retardo absoluto con `delay until` donde el parámetro es del tipo `Time`

Prioridades de tareas

- El lenguaje Ada introduce un comportamiento no determinista en el modelo de concurrencia:
  - El orden en que se ejecutan las tareas
  - Las gestión de las colas de las entries
  - El comportamiento de los objetos protegidos
- 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 en los objetos protegidos

Prioridades en Ada

- Ada permite asignar prioridades estáticas a las tareas y modificar su prioridad durante la ejecución
- En el paquete `System` se definen los siguientes tipos para manejar 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;
```

Prioridades en Ada (cont.)

- La prioridad inicial de una tarea se asigna incluyendo una sentencia `pragma` en su especificación

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

- Para entidades que actúen como manejadores de interrupción se usará:

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

- Si a una tarea no se le asigna prioridad se le da la prioridad `Default_Priority`
- Para modificar la prioridad de una tarea cuando ya ha sido creada se dispone del paquete `Ada.Dynamic_Priorities`
- La política de planificación se puede elegir el siguiente pragma:

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


Prioridades dinámicas

```
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`
- La función `Current_Task` que devuelve el `Task_Id` de la tarea que la llama

Protocolos de bloqueo

- Pretenden evitar la inversión de prioridad
- Ada ofrece el mecanismo de techo de prioridad
  - Se asigna una prioridad a un objeto protegido, mayor que la de todas las tareas que lo utilizan
  - Mientras una tarea lo utiliza, adquiere la prioridad del objeto protegido
- Para utilizar este protocolo se debe utilizar el pragma:

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

- Para definir la prioridad del objeto protegido se utiliza un pragma similar al de las prioridades de las tareas

Protocolos de bloqueo (cont.)

- Si una tarea con prioridad mayor que la definida en un objeto protegido intenta utilizarlo se produce una excepción del tipo `Program_Error`

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

Políticas de colas

- Se puede elegir la política de tratamiento de colas
  - En las barreras de los objetos protegidos
  - En las colas de mensajes de las tareas
  - Entre las distintas colas de una sentencia select
- La políticas de cola se define con el pragma:

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

- Existen dos identificadores de políticas predefinidos:
  - `Fifo_Queuing` (defecto)
  - `Priority_Queuing`