# Lenguajes de Programación

Dr. Carlos A. Coello Coello
Departamento de Computación
CINVESTAV-IPN
ccoello@cs.cinvestav.mx

# Paquetes

- Un usuario obtiene acceso a la parte pública de un paquete usando la declaración "**use**":

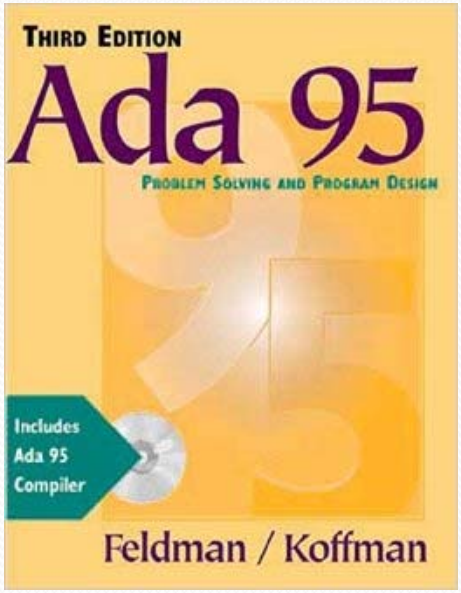
```
declare
        use COMPLEX_TYPE;
        X, Y : COMPLEX;
        Z : COMPLEX := 1.5+2.5*I;
begin
        X := 2.5 + 3.5*I;
        Y := X + Z;
        Z := RE(Z) + IM(X)*I;
        if X = Y then X := Y + Z;
        else X := Y*Z; end if;
 end;
```

# Paquetes

- La declaración "**use**" hace que todos los nombres públicos de un paquete sean visibles a través del bloque en el cual aparece.

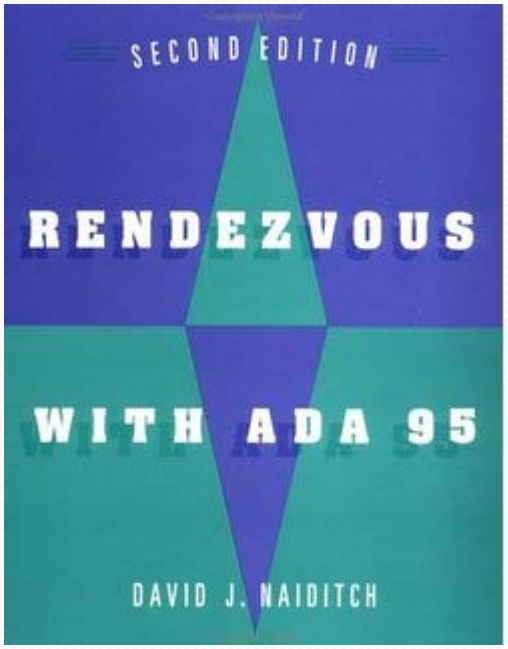
- Podemos ver que esto permite usar todos los tipos, funciones, constantes y demás, como si fuesen parte interna del lenguaje.

# Paquetes




- De hecho, vale la pena mencionar que Ada está implementado de tal forma que considera que los tipos "internos" se encuentran definidos en paquetes que son usados automáticamente (a través de "**use**") por todos los programadores.

# Paquetes




- Por lo tanto, el acceso a nombres es por **consentimiento mutuo**: El implementador del paquete determina qué atributos serán públicos y el usuario del paquete decide si importa los atributos de un paquete en particular.

# Paquetes

- De hecho, es posible efectuar un mayor control en Ada, puesto que podemos seleccionar sólo un cierto nombre dentro de un cierto paquete usando una notación similar a la de Pascal para acceder a un registro.

- Por ejemplo: COMPLEX_TYPE.I

# Paquetes

- Ada también proporciona control sobre la visibilidad durante la compilación separada.

- Normalmente, sólo los identificadores internos son visibles a un módulo compilado por separado, pero si el módulo es precedido por "**with** <lista de nombres>", entonces los identificadores públicos de los módulos nombrados también se vuelven visibles.

# Paquetes

- Los paquetes pueden ser usados como **bibliotecas**.
- De hecho, en Ada, una biblioteca de funciones puede verse simplemente como un paquete que no contiene estructuras de datos.

# Paquetes

- Por ejemplo:

```
package PLOT is
   type POINT is record X, Y : FLOAT; end record;
   procedure MOVE_TO (LOCATION : POINT);
   procedure LINE (FROM, TO : POINT);
   procedure CIRCLE (CENTER : POINT; RADIUS : FLOAT);
   procedure FIT (DATA : array (INTEGER range <>) of
POINT);
   function WHERE return POINT;
end PLOT;
```

# Paquetes

- Una vez realizada la definición del paquete PLOT, si deseamos invocar sus funciones de graficación, lo único que tenemos que hacer es incluir un '**use** PLOT' en nuestro subprograma.

- Evidentemente, la parte privada del paquete puede incluir las definiciones de las constantes y los subprogramas que se necesitan para la implementación, pero que queremos ocultar al usuario. Por tanto, puede verse cómo una biblioteca en Ada es realmente un paquete que no contiene ninguna estructura de datos.

# Paquetes

- Otro uso interesante de los paquetes es para compartir áreas de datos. Consideremos el siguiente ejemplo:

```
package COMMUNICATION is
    IN_PTR, OUT_PTR : INTEGER range 0..99:=0;
    BUFFER : array (0..99) of CHARACTER := (0..99 = > ‘’);
end COMMUNICATION;
```

NOTA: La declaración de BUFFER hace que sea un arreglo inicializado con caracteres en blanco (‘’).

# Paquetes

- En este caso, todo procedimiento que necesite usar esta área de datos simplemente necesita declararla mediante "**use**".

- Por ejemplo, usando el código que mostramos en el acetato anterior, podemos usar dos procedimientos P y Q para comunicarse entre sí mediante "**use**".

# Paquetes

```
procedure P is
        use COMMUNICATION;
begin
                ⋮
        BUFFER(IN_PTR) := NEXT;
        IN_PTR := (IN_PTR+1) mod 100;
                ⋮
end P;

procedure Q is
        use COMMUNICATION;
begin
                ⋮
        C := BUFFER(OUT_PTR);
                ⋮
end Q;
```

# Paquetes

- Esto es similar al uso de COMMON en FORTRAN, pero evita el problema de las definiciones traslapadas que vimos antes, así como los problemas de “aliasing” que ocurrían en FORTRAN.

- En este caso, sólo aquellos subprogramas que necesiten acceder a BUFFER, podrán hacerlo aplicando **use** a COMMUNICATION.

# Paquetes

- Los paquetes pueden usarse de forma similar a las clases en un lenguaje orientado a objetos.

- Es decir, que con un paquete podemos encapsular una estructura de datos y luego proporcionar una interfaz independiente de la representación para accederla.

# Paquetes

- Por ejemplo, podríamos crear una pila y después proporcionar acceso a sus funciones para hacer POP y PUSH de elementos de o hacia ella.

- Los detalles de la implementación, sin embargo, permanecerían ocultos.

# Paquetes

Por ejemplo, consideremos la siguiente definición de una pila:

```
package STACK1 is
        procedure PUSH (X : in INTEGER);
        procedure POP (X : out INTEGER);
        function EMPTY return BOOLEAN;
        function FULL return BOOLEAN;
        STACK_ERROR : exception;
end STACK1;
```

# Paquetes

- La decisión respecto a cómo implementar la pila (p.ej., usando un arreglo) se oculta en el paquete.

- Por ejemplo, en el acetato siguiente mostramos una posible implementación de una pila.

```
package body STACK1 is
  ST : array (1..100) of INTEGER;
  TOP : INTEGER range 0..100 := 0;

  procedure PUSH (X : in INTEGER) is
  begin
    if FULL() then raise STACK_ERROR;
    else
       TOP := TOP +1; ST(TOP) := X;
    end if;
  end PUSH;

  procedure POP (X : out INTEGER) is
  begin
      if EMPTY() then raise STACK_ERROR;
      else
         X := ST(TOP);
         TOP := TOP -1;
      end if;
  end POP;

function EMPTY return BOOLEAN is
begin return TOP = 0; end;

function FULL return BOOLEAN is
begin  return TOP = 100; end;

end STACK1;
```

# Paquetes

- Una vez que este paquete ha sido implementado, puede usarse igual que antes. Por ejemplo:

```
declare
        use STACK1;
        I, N : INTEGER;
begin
          ⋮
        PUSH(I);
        POP(N);
          ⋮
if EMPTY() then PUSH(N); end if;
          ⋮
end;
```

# Paquetes

- También puede adoptarse la notación "punto" para seleccionar un atributo público de STACK1 sin necesidad de adoptar "**use**". Por ejemplo:

```
STACK1.PUSH(I);
STACK1.POP(N);
if STACK1.EMPTY() then STACK1.PUSH(N); end if;
```

  - Esto permite a los usuarios del paquete ser tan selectivos como requieran acerca de los nombres a los que se accede desde STACK1. Nuevamente, el acceso es por mutuo consentimiento.

# Paquetes Genéricos


- Mejor aún, Ada permite la definición de *paquetes genéricos*, los cuales son similares a los "templates" de C++, ya que proporcionan una interfaz más general a un cierto tipo de estructura de datos sin importar el tipo base usado en nuestro código.

# Paquetes Genéricos

- Ejemplo:

```
generic
package STACK is
        procecure PUSH (X : in INTEGER);
        procedure POP (X : out INTEGER);
        function EMPTY return BOOLEAN;
        function FULL return BOOLEAN;
        STACK_ERROR: exception;
end STACK;
```

# Paquetes Genéricos

- Esta definición luce exactamente igual que la que vimos anteriormente para un paquete, excepto por el uso de la palabra "**generic**".

- Supongamos ahora que queremos dos pilas a las que llamaremos STACK1 y STACK2. Podemos solicitar su creación usando:

**package** STACK1 **is new** STACK;
**package** STACK2 **is new** STACK;

# Paquetes Genéricos

- Posteriormente, podemos usar estas dos instancias de STACK usando la notación de punto. Por ejemplo:

STACK1.PUSH(I);

STACK2.POP(N);

**if** STACK2.EMPTY() **then** STACK1.PUSH(N); **end if**;

Nótese que no es posible utilizar "**use**" para construir pilas, ni para operadores como PUSH(I), porque esto daría pie a ambigüedades (no sabemos a cuál de las 2 pilas se refiere el operador).

# Paquetes Genéricos

- Los paquetes son instanciados en una forma muy similar a los procedimientos, pero con una diferencia notable:

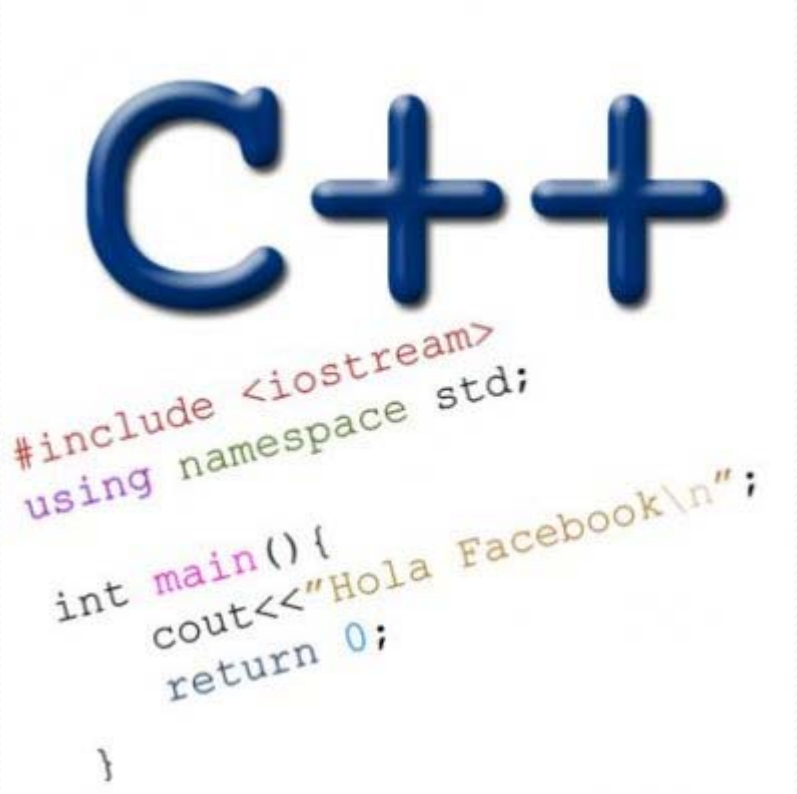
- Los procedimientos pueden ser instanciados *dinámicamente*, mientras que los paquetes sólo pueden ser instanciados *estáticamente*.

# Paquetes Genéricos

- Alguien podría argumentar que ya que los procedimientos pueden ser instanciados dinámicamente, que entonces podríamos colocar un paquete dentro de un procedimiento para superar esta limitante.

- Aunque esto es válido, sería algo incorrecto (en el sentido más puro) el decir en este caso que los paquetes en Ada pueden instanciarse dinámicamente.

# Paquetes Genéricos




- Como mencionamos antes, hay muchas similitudes entre los paquetes de Ada y las clases de los lenguajes orientados a objetos como C++.

# Paquetes Genéricos

- ¿Qué hay acerca de su instanciación?
- ¿Pueden instanciarse dinámicamente las clases?
- Simula-67 y Smalltalk permiten la instanciación dinámica de clases, pero, como veremos más adelante, el problema principal con esto es que amerita un nuevo esquema para manejo de memoria (la recolección de basura).

# Paquetes Genéricos

- Existen más funciones genéricas disponibles en Ada.
- En el paquete que vimos antes para implementar una pila (STACK1), la pila estaba limitada a 100 elementos.
- Ahora supongamos que en vez de dos pilas del mismo tamaño, necesitamos que STACK1 tenga 100 elementos y que necesitamos que STACK2 tenga 64 elementos.

# Paquetes Genéricos

- ¿Cómo podemos hacer esto? Parecería que tendríamos que copiar el código de STACK1 y reemplazar todas las ocurrencias de 100 por 64.

- Evidentemente, esto sería muy ineficiente y dañaría la vulnerabilidad, la legibilidad y el mantenimiento del código.

- Los procedimientos resuelven este problema mediante los parámetros. Ada resuelve este problema al permitir parámetros en un paquete genérico. Veamos un ejemplo.

# Paquetes Genéricos

```
generic
        LENGTH : NATURAL := 100;
package STACK is
        procedure PUSH (X : in INTEGER);
        procedure POP (X : out INTEGER);
        function EMPTY return BOOLEAN;
        function FULL return BOOLEAN;
        STACK_ERROR : exception;
end STACK;
```

# Paquetes Genéricos

- Nótese que en este caso, a LENGTH se le asignó un valor de 100. Este será el valor que tomará si no se especifica. El cuerpo del paquete se altera reemplazando cada ocurrencia de 100 por LENGTH como se muestra a continuación:

```
package body STACK is
    ST : array (1..LENGTH) of INTEGER;
    TOP : INTEGER range 0..LENGTH := 0;
     ... el resto de las definiciones ...
     ... pero con 100 reemplazado por LENGTH ...
end stack;
```

# Paquetes Genéricos

- Cuando una pila es instanciada, este parámetro debe ligarse a un número natural. Por ejemplo:

**package** STACK1 **is new** STACK(100);
**package** STACK2 **is new** STACK(64);

Como el tamaño de la pila por omisión es 100, también podemos usar:

**package** STACK1 **is new** STACK;

# Paquetes Genéricos

- Los paquetes genéricos pueden tener cualquier número de parámetros de cualquier tipo, igual que los procedimientos.
- También pueden tener varios tipos de parámetros que no pueden usarse con los procedimientos.
- Supongamos que necesitamos usar una pila, STACK3, que contenga sólo caracteres.

# Paquetes Genéricos

- Parecería como que nuevamente tendríamos que copiar la definición que mostramos antes de una pila, reemplazando INTEGER por CHARACTER.

- Esto es, evidentemente, indeseable, por las mismas razones que mencionamos antes. Se sacrificaría la vulnerabilidad y legibilidad del código y se complicaría el mantenimiento al mismo.

# Paquetes Genéricos

- Para resolver este problema, en Ada se pueden definir pilas independientes del tipo:

```
generic
        LENGTH : NATURAL := 100;
        type ELEMENT is private;
  package STACK is
        procedure PUSH (X : in ELEMENT);
        procedure POP (X : out ELEMENT);
        function EMPTY return BOOLEAN;
        function FULL return BOOLEAN;
        STACK_ERROR : exception;
  end STACK;
```

# Paquetes Genéricos

- El tipo de parámetro se define como **private** porque actúa como un tipo privado.
- Las únicas operaciones disponibles sobre este tipo son la asignación y las comparaciones de igualdad.
- La implementación de STACK se muestra en el acetato siguiente.

```
package body STACK is
   ST : array (1..LENGTH) of ELEMENT;
   TOP : INTEGER range 0..LENGTH := 0;

   procedure PUSH (X : out ELEMENT) is
   begin
        ... como antes ...
   end PUSH;

   procedure POP (X : out ELEMENT) is
   begin
        ... como antes ...
   end POP;

   function EMPTY return BOOLEAN is
   begin return N = 0; end;

   function FULL return BOOLEAN is
   begin return N = LENGTH;  end;

end STACK;
```

# Paquetes Genéricos

- Dadas estas definiciones, los siguientes son ejemplos de instanciaciones de pilas generales:

  **package** STACK1 **is new** STACK (100, INTEGER);

  **package** STACK3 **is new** STACK (256, CHARACTER);

# Paquetes Genéricos

- El primer parámetro define el tamaño de la pila y el segundo el tipo base de sus elementos.

- Estas pilas pueden usarse con la notación punto que vimos antes. Por ejemplo: ‘STACK1.POP(N)’ o ‘STACK3.PUSH(‘A’)’.

# Paquetes Genéricos

- También se pueden acceder mediante "**use**":

```
declare
        use STACK1;
        use STACK3;
        I, N : INTEER;
        C, D: CHARACTER;
  begin
        PUSH(I);
        PUSH(C);
        POP(N);
        POP(D);
        if STACK1.EMPTY() then ....
        if STACK3.FULL() then ...
  end;
```

# Paquetes Genéricos

- En este caso, se permite el uso de ambas pilas, porque el contexto determina qué procedimiento es el que estamos tratando de utilizar.
- Por ejemplo, 'PUSH(I)' no es ambiguo, porque I es un INTEGER y sólo hay un procedimiento PUSH que opera sobre un parámetro de tipo INTEGER.
- Nótese que las funciones EMPTY y FULL deben accederse usando un operador punto, porque no toman ningún argumento.

# Paquetes Genéricos

- Los paquetes genéricos suenan como una gran idea, pero su flexibilidad no es gratuita.
- De hecho, en este caso, el costo es tal vez demasiado alto, debido sobre todo a las muchas opciones que nos proporcionan.
- De tal forma, es muy difícil poder generar código eficiente para poder manejar paquetes genéricos.

# Paquetes Genéricos

- Cuando se crean varias estructuras de datos genéricas, es posible compartir código, pero eso hará que se requieran más chequeos en tiempo de ejecución.

- Esto se complica cuando se usan paquetes con parámetros, pues en este caso, podemos tener, por ejemplo, una pila de enteros y otra de caracteres, por lo cual compartir código se vuelve mucho más difícil.

# Representaciones Internas y Externas

- En los ejemplos que hemos visto de paquetes, hemos manejado la representación de las estructuras de datos de dos formas diferentes.

- En nuestro ejemplo del paquete STACK, los procedimientos para manipular la pila eran “parte” de cada pila, por lo que usamos cosas como “STACK1.POP(N)”. A esto se le llama una **representación interna**, porque los operadores de una cierta estructura de datos se encuentran, conceptualmente, adentro de cada instancia de dicha estructura de datos.

# Representaciones Internas y Externas

- El otro enfoque es el que usamos en el paquete COMPLEX: Los operadores estaban en un paquete que manejaba todos los objetos complejos, y escribíamos cosas como “RE(Z)”.

- A esto se le conoce como una representación externa.

- Frecuentemente, una misma estructura de datos se puede representar de manera externa o interna. Veamos, por ejemplo, una representación externa de una pila.

# Representaciones Internas y Externas

```
package STACK_TYPE is
        type STACK is private;
        procedure NEW_STACK (out S : STACK);
        procedure PUSH (in out S : STACK; in X : INTEGER);
        procedure POP (in out S : STACK; out X : INTEGER);
        function EMPTY (S : STACK) return BOOLEAN;
        function FULL (S : STACK) return BOOLEAN;

private
        type STACK is record
          ST : array (1..100) of INTEGER;
          TOP : INTEGER range 0..100 := 0;
          end record;

end STACK_TYPE;
```

# Representaciones Internas y Externas

- Ahora podemos instanciar nuestras pilas de la manera siguiente:

```
declare
        use STACK_TYPE;
        STACK1 : STACK;
        STACK2 : STACK;
begin
        NEW_STACK (STACK1);
        NEW_STACK (STACK2);
        PUSH (STACK1, I);
        POP (STACK2, N);
        if EMPTY (STACK1) then PUSH (STACK1, N); end if;
   end;
```

NOTA: No se puede declarar una pila como una constante, debido a la necesidad de inicializarla con NEW_STACK.

# Representaciones Internas y Externas

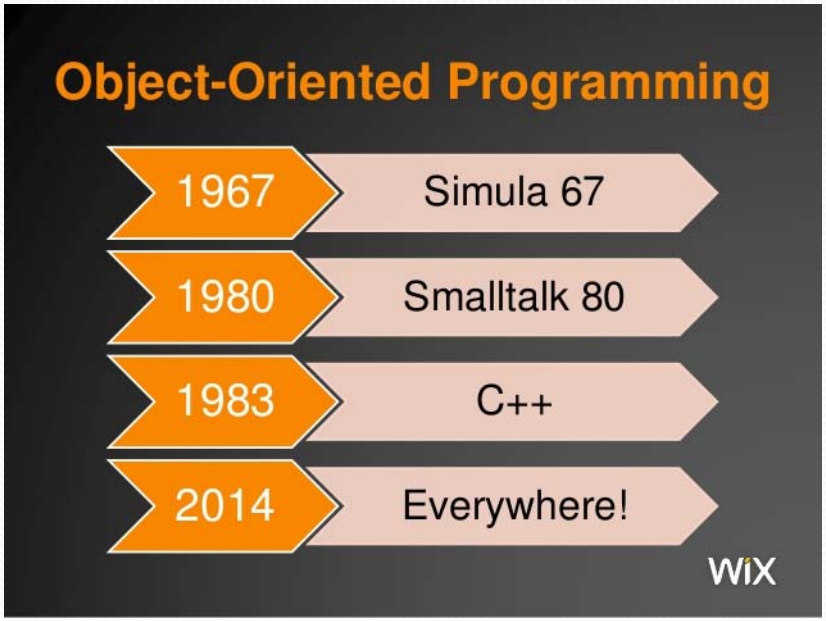
- Existen varias diferencias entre las representaciones internas y las externas.
- En Ada, las representaciones externas son más generales y flexibles.
- Por ejemplo, usando una definición externa de una pila, podemos pasarla como parámetro, asignarla, hacerla elemento de otra estructura de datos, etc.

# Representaciones Internas y Externas

- Por ejemplo, si no sabemos exactamente cuántas pilas necesitaremos, podemos declarar un arreglo o una lista ligada de STACKs e inicializarlas (vía NEW_STACK) con tantos elementos como se requiera.

- Por tanto, el número de instancias de una representación externa puede determinarse dinámicamente, mientras que para una representación interna, el número de instancias está limitado por el número de instanciaciones genéricas que el programador escriba.

# Representaciones Internas y Externas




- Simula-67 y Smalltalk usan representaciones internas para todas sus estructuras de datos (llamadas clases), pero les permiten ser instanciadas dinámicamente. Ese es un enfoque más orientado a objetos que el adoptado por Ada.

# Sobrecarga de Procedimientos

- Hemos visto que los elementos de una enumeración pueden estar sobrecargados en Ada.
- También vimos que los operadores nativos del lenguaje (p.ej., +, -, *, /) también pueden pueden sobrecargarse (como lo hicimos con nuestra definición de números complejos).
- En Ada, también pueden sobrecargarse los procedimientos y esto suele ocurrir sobre todo cuando se usan paquetes genéricos.

# Sobrecarga de Procedimientos

- Supongamos que INT_STACK_TYPE y CHAR_STACK_TYPE son paquetes que implementan pilas representadas externamente de enteros y caracteres, respectivamente. Un ejemplo de su uso es:

```
declare
   S1 : INT_STACK_TYPE.STACK;
   S2 : CHAR_STACK_TYPE.STACK;
begin
   INT_STACK_TYPE.PUSH(S1,5);
   CHAR_STACK_TYPE.PUSH(S2,'A');
end;
```

# Sobrecarga de Procedimientos

- Como resulta un tanto inconveniente tener que usar los prefijos INT_STACK_TYPE o CHAR_STACK_TYPE cada vez que usemos PUSH o POP, podemos emplear "**use**":

```
declare
   S1 : INT_STACK_TYPE.STACK;
   S2 : CHAR_STACK_TYPE.STACK;
   use INT_STACK_TYPE;
   use CHAR_STACK_TYPE;
begin
   PUSH (S1,5);
   PUSH (S2,'A');
end;
```

# Sobrecarga de Procedimientos

- Después de las dos declaraciones de “use”, tenemos disponibles dos declaraciones de cada uno de los operadores de las pilas (PUSH, POP, EMPTY, FULL), uno para la pila de enteros y otro para la pila de caracteres.

- Estos se distinguen por contexto, igual que los operadores sobrecargados. Por ejemplo, PUSH(S1,5) debe referirse a la pila de enteros.

# Sobrecarga de Procedimientos

- Debe resultar evidente en este caso que el compilador debe efectuar un proceso de identificación de operadores para los nombres de los procedimientos.

- Este proceso depende de los argumentos del subprograma y, si se trata de una función, depende también de su contexto de uso.

# Sobrecarga de Procedimientos

- Por ejemplo, en la expresión:

```
Z := F(G(X,Y));
```

El procedimiento F a ser utilizado depende tanto del tipo de Z (o sea, el contexto de F) como del tipo que regresa G (o sea, el argumento de F).

Por otra parte, G podría ser un procedimiento sobrecargado, en cuyo caso su significado dependerá de sus argumentos (X, Y) y de su contexto de uso (el argumento que requiere F).

# Sobrecarga de Procedimientos

- Para que un programa en Ada sea considerado correcto, debe haber una selección única para F y G que satisfaga las restricciones antes mencionadas.
- Si no existe ninguna, entonces el programa no tiene sentido.
- Si existe más de una, entonces el programa es ambiguo.

# Sobrecarga de Procedimientos


- Con esto puede verse que si la sobrecarga se usa de manera excesiva, puede volverse muy difícil para el compilador (e incluso para un humano) determinar el significado de una expresión en Ada.

# Sobrecarga de Procedimientos

- Esto puede complicarse todavía más si consideramos que en Ada podemos usar parámetros con posiciones independientes, como veremos más adelante.

- La identificación de operadores suele hacerse propagando la información de tipos hacia arriba y hacia debajo de los árboles de expresiones en varias pasadas.

# Sobrecarga de Procedimientos

- Sin embargo, el número exacto de pasadas que se requieren ha sido tema de investigación durante varios años.
- Los procedimientos sobrecargados, sin embargo, no son tan poco comunes como uno pudiera pensar.
- De hecho, se motiva a los programadores en Ada a que sobrecarguen los nombres de los procedimientos.

# Sobrecarga de Procedimientos

- En efecto, la misma definición de Ada pareciera motivar el uso de procedimientos sobrecargados.
- Hay ejemplos en el lenguaje, tales como el comando GET, que tiene más de 14 significados diferentes (uno por cada tipo interno y por cada enumeración).

# Estructuras de Control

- Ada tiene un conjunto mucho más rico de estructuras de control:

1. Un condicional
2. Un iterador (definido e indefinido)
3. Una sentencia para casos
4. Subprogramas
5. Una sentencia **goto**
6. Facilidades para manejo de excepciones
7. Facilidades para programación concurrente

# Estructuras de Control

- Ada proporciona sólo una sentencia para iteraciones, llamada "**loop**", la cual maneja iteraciones definidas, indefinidas e incluso infinitas. Aunque esta sentencia es muy general y resuelve algunos de los problemas de Algol y Pascal, también es muy barroca. Su sintaxis es:

**loop** <secuencia de sentencias> **end loop**

- Este es un ejemplo de un iterador infinito (o sea, la secuencia de sentencias se ejecutan para siempre en este caso).

# Estructuras de Control

- Como los ciclos infinitos no son muy frecuentes en programación, Ada proporciona una sentencia "**exit**" para salirse de un ciclo infinito. La ejecución de esta sentencia en cualquier parte del cuerpo de un ciclo, hará que el ciclo se aborte y el control continúa después del "**end loop**".

- Por ejemplo, el acetato siguiente muestra el código para buscar una llave K en un arreglo A.

# Estructuras de Control

```
I:=A'FIRST;
loop
        if I > A'LAST then exit; end if;
        if A(I) = K then exit; end if;
        I := I+1;
end loop;
```

# Estructuras de Control

- Podemos usar cualquier número de sentencias "**exit**" que queramos y éstos pueden estar embebidos a cualquier profundidad dentro del ciclo.

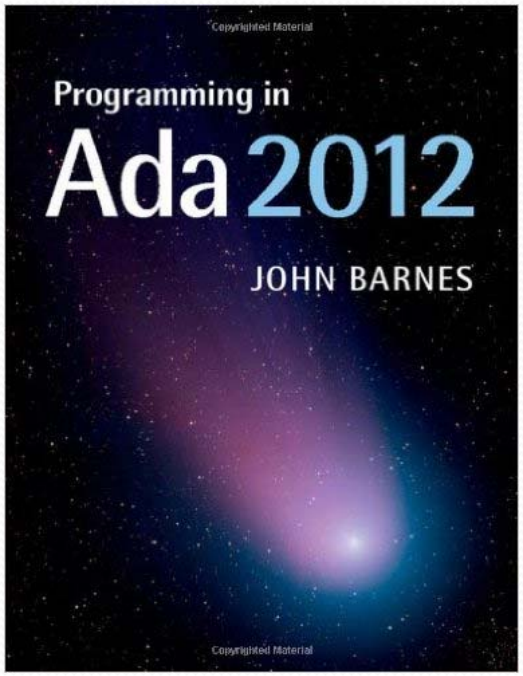
- Debe resultar evidente que el uso de "**exit**" permite implementar ciclos con la decisión en medio en Ada.

# Estructuras de Control

- Adicionalmente, usando etiquetas, cualquier sentencia "**exit**" puede salir de cualquier número de niveles de anidamiento e incluso de bloques (aunque no de subprogramas).

- Puede verse que "**exit**" tiene de alguna forma, las características de un "**goto**" no local.

# Estructuras de Control




- Puesto que las sentencias "**exit**" son con mucha frecuencia el consecuente de una sentencia "**if**", Ada proporciona un tipo de abreviación especial para manejar este caso.

# Estructuras de Control

- Usándola, el ciclo anterior puede re-escribirse como:

```
I := A'FIRST;
loop
        exit when I > A'LAST;
        exit when A(I) = K;
        I := I+1;
end loop;
```

# Estructuras de Control

- El propósito de esta abreviación es no sólo ahorrarse unos cuantos caracteres, sino también marcar claramente las condiciones de terminación de un ciclo.

- Sin embargo, puesto que "**exit**" puede estar enterrado profundamente en el cuerpo de un ciclo, esta segunda tarea puede ser muy complicada, aún usando esta abreviación.

# Estructuras de Control

- Ada lleva la idea de las abreviaciones todavía más lejos.
- Como la combinación "**exit-when**" aparece con mucha frecuencia al inicio de los ciclos, es permisible que estos ciclos se re-escriban de una forma similar a los ciclos "**while**" de Pascal.

# Estructuras de Control

```
I := A'FIRST;
while I <= A'LAST
loop
        exit when A(I) = K;
        I := I + 1;
end loop;
```

# Estructuras de Control

- Advierta que la frase "**while**" es simplemente un prefijo del ciclo básico.

- De hecho, hay incluso otra abreviación para casos de iteración definida, tales como el que se mostró anteriormente. En este caso se usa "**for**" en vez de "**while**".

# Estructuras de Control

```
for I in A'FIRST . . A'LAST
loop
        exit when A(I) = K;
end loop;
```

# Estructuras de Control

- Desafortunadamente, esto no funciona exactamente como el ciclo "**while**", debido a que la frase "**for**" automáticamente declara la variable de control I, haciéndola, por tanto, local al ciclo.

- Esto significa que fuera del ciclo no será posible determinar si K se encontró o no.

# Estructuras de Control

- Para corregir este problema, debemos usar una variable diferente para controlar el ciclo, de manera que podamos almacenar la localización de K en I.

- Veamos la solución en el acetato siguiente.

# Estructuras de Control

```
for J in A'FIRST . . A'LAST
  loop
        if A(J) = K then
              I := J;
              exit;
        end if;
  end loop;
```

# Estructuras de Control

- Adviértase que en este caso estamos de vuelta usando una sentencia "**exit**" adentro de una sentencia "**if**".
- Finalmente, observe que el código anterior no nos permite determinar si K se encontró o no.
- Esto es porque el flujo de control alcanza el "**end loop**" en ambos casos.
- Para resolver este problema debemos introducir otra variable, como se muestra en el acetato siguiente.

# Estructuras de Control

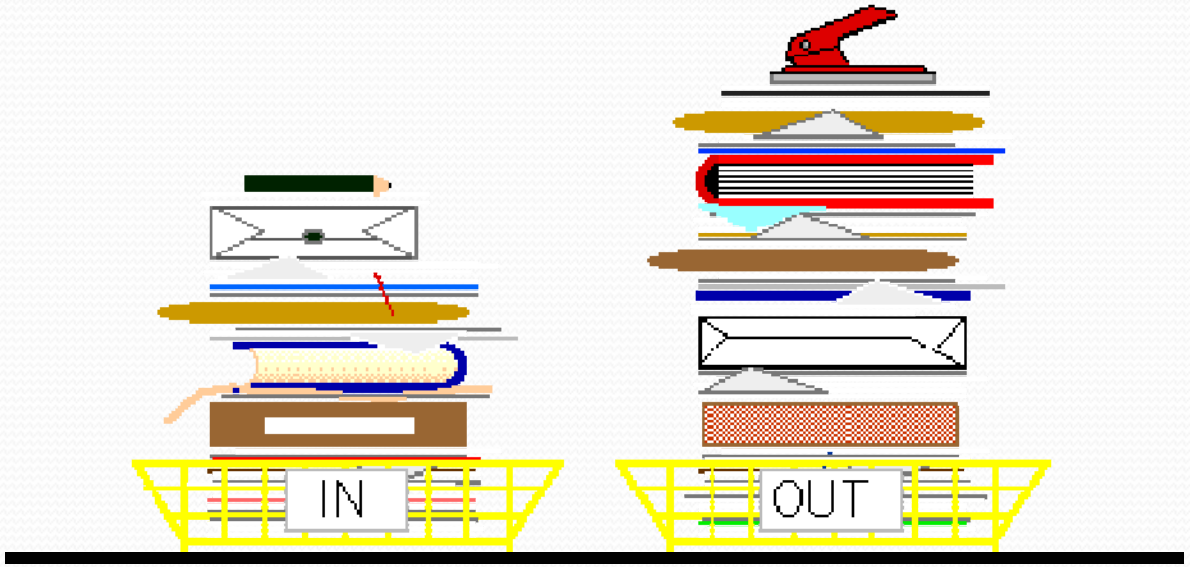
```
declare
   FOUND: BOOLEAN := FALSE;
begin
   for J in A'FIRST..A'LAST
   loop
          if A(J) = K then
            I := J;
            FOUND := TRUE;
            exit;
          end if;
   end loop;

   if FOUND then
          :   --- manejar caso encontrado
      else
          :   --- manejar caso no encontrado
      end if;
end;
```

# El Mecanismo de Excepción

- Puesto que Ada fue diseñado para aplicaciones embebidas (p.ej., sistemas de misiles), sus programas deben actuar razonablemente bien bajo una serie de condiciones distintas.

- Por tanto, en el caso de fallas, es importante que los programas en Ada puedan responder a situaciones excepcionales.

# El Mecanismo de Excepción




- Esto lo lleva a cabo el **mecanismo de excepción** de Ada, el cual nos permite definir situaciones excepcionales, indicar su ocurrencia y responder a ellas.

# El Mecanismo de Excepción


- Las excepciones son muy útiles para interceptar errores del sistema, puesto que pueden prevenir casos tales como la división entre cero, overflow, underflow, entrada inválida de datos, etc.

# El Mecanismo de Excepción

- Supongamos que tenemos un subprograma llamado PRODUCER, que está dentro de un programa en Ada es responsable de obtener datos y almacenarlos en una pila para ser procesados posteriormente por otro subprograma llamado CONSUMER.

- Si los datos llegan más rápidamente que lo que se espera, PRODUCER puede intentar colocar más datos en la pila de los que caben en ésta. Esta es una situación excepcional que el programa debe manejar adecuadamente.

# El Mecanismo de Excepción

- Esto ya lo habíamos ilustrado anteriormente, en la definición de STACK1:

```
package STACK1 is
        procedure PUSH (X : in INTEGER);
        procedure POP (X : out INTEGER);
        function EMPTY return BOOLEAN;
        function FULL return BOOLEAN;
        STACK_ERROR : exception;
end STACK1;
```

```
package body STACK1 is
  ST : array (1..100) of INTEGER;
  TOP : INTEGER range 0..100 := 0;

  procedure PUSH (X : in INTEGER) is
  begin
     if FULL() then raise STACK_ERROR;
     else
         TOP := TOP +1; ST(TOP) := X;
     end if;
   end PUSH;

   procedure POP (X : out INTEGER) is
   begin
        if EMPTY() then raise STACK_ERROR;
        else
            X := ST(TOP);
            TOP := TOP -1;
        end if;
   end POP;

function EMPTY return BOOLEAN is
begin return TOP = 0; end;

function FULL return BOOLEAN is
begin  return TOP = 100; end;

end STACK1;
```

# El Mecanismo de Excepción

- En este código consideramos una excepción llamada STACK_ERROR que se activará cuando intentamos hacer una operación PUSH en una pila llena o una operación POP en una pila vacía. Se usa la sentencia **raise** para activar una excepción.

- En virtud de esta implementación, necesitamos ahora un método para manejar esta situación excepcional.

# El Mecanismo de Excepción

- Regresando al ejemplo antes mencionado, supongamos que los datos recientes son más importantes que los datos viejos.

- De tal forma, en este caso, podemos manejar una situación de sobrecarga en la pila, eliminando algunos elementos de la pila para dar cabida a los nuevos datos.

# El Mecanismo de Excepción

```
procedure PRODUCER ( ... );
  use STACK1;
begin
    ⋮
  PUSH (DATA);
    ⋮
  exception
    when STACK_ERROR =>
      declare SCRATCH : INTEGER;
      begin
        for I in 1..3 loop
          if not EMPTY() then POP(SCRATCH); end if;
        end loop;
        PUSH(DATA);
      end;
end PRODUCER;
```

# El Mecanismo de Excepción

- En este código puede verse que si las excepciones tuvieran las mismas reglas de entorno que las usadas en Algol y que las que usan otros identificadores en Ada, entonces la definición de STACK_ERROR en PRODUCER no sería visible en PUSH.

- Esto hace evidente que las excepciones deben seguir reglas diferentes para su interpretación.

# El Mecanismo de Excepción

- Supongamos que PRODUCER no hubiese definido un manejador para STACK_ERROR. ¿Qué habría pasado en este caso si PUSH hubiese generado un STACK_ERROR?

- En Ada, si PRODUCER no cuenta con un manejador de esta excepción, ésta se propaga al invocador de PRODUCER. Si el invocador define el manejador para STACK_ERROR, entonces lo ejecuta ahí. De lo contrario, se sigue propagando la excepción, hasta el procedimiento más externo, en cuyo caso el programa terminará.

# El Mecanismo de Excepción

- Esto nos ilustra la esencia del mecanismo de propagación de excepciones de Ada. Si la excepción está definida en el ambiente local, vamos a su manejador.
- De lo contrario, buscamos dicho manejador en el ambiente de su invocador.
- Continuamos recorriendo la liga dinámica hasta que encontremos un manejador para la excepción.

# El Mecanismo de Excepción

- Esto es por demás interesante, porque en lo referente a excepciones, los subprogramas son invocados en el ambiente del invocador, aunque en todos los demás casos, se invocan en su ambiente de definición.

- Aunque todos los demás nombres en Ada se asocian estáticamente, las excepciones se asocian dinámicamente, lo que hace que las excepciones en Ada sean excepción a las reglas de entorno del lenguaje. Esto viola el Principio Estructural y el Principio de Regularidad.

# El Mecanismo de Excepción

- Cuando se eleva una excepción, debe usarse una rutina que se activa en tiempo de ejecución y cuya labor es recorrer la cadena dinámica buscando un ambiente que contenga el manejador de la excepción activada.

- Durante este recorrido, el registro de activación de cualquier subprograma o bloque que no defina un manejador para la excepción dada, debe ser borrado. De tal forma, podemos ver que una excepción funciona como un "**goto**" no local con un entorno dinámico.

# El Mecanismo de Excepción

- Las implicaciones obvias de esto es que no podemos realmente recuperarnos de una excepción y continuar la ejecución de nuestro programa como en otros lenguajes (p.ej., PL/I).

- Esto se debe a que todos los registros de activación recorridos antes de llegar al manejador de la excepción, fueron borrados durante la búsqueda.