Lenguajes de Programación

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

Mecanismos de Paso de Parámetros

- En clases anteriores, hemos visto diferentes mecanismos de paso de parámetros y analizamos sus ventajas y desventajas.

- Ada proporciona tres modos de paso de parámetros que reflejan la forma en la que el programador pretende usar el parámetro que se pasa: entrada, salida o ambos: entrada y salida. Estos modos se indican con las palabras reservadas '**in**', '**out**' e '**in out**', las cuales se colocan antes del parámetro en cuestión. Si no se coloca ninguna de estas 3 palabras reservadas, se supone el modo '**in**'.

Mecanismos de Paso de Parámetros

- Los parámetros en modo '**in**' se usan para transmitir información hacia un procedimiento, pero no para recibir ninguna información de éste. Este modo es idéntico al paso de parámetros por constante que vimos antes.

- Puesto que este tipo de parámetros se usan solo para entrada, el lenguaje no permite alterarlos. Por tanto, realizar una asignación a un parámetro en modo '**in**' es ilegal.

Mecanismos de Paso de Parámetros

- Recordemos que en Pascal el uso de paso por constante dejaba al compilador el elegir si se pasaría el valor de un parámetro o su dirección.

- Ada no va tan lejos, y especifica que los parámetros escalares siempre se copiarán (o sea, se pasarán por valor).

- Para los tipos *compuestos* (p.ej., arreglos y registros) el compilador puede elegir copiar el valor (si es pequeño) o pasar su dirección.

Mecanismos de Paso de Parámetros

- Los parámetros en modo '**out**' son lo opuesto de los parámetros en modo '**in**'. Se usan para transmitir resultados desde un subprograma.

- Por lo tanto, estos parámetros se consideran sólo de escritura. Dentro de un subprograma, pueden usarse como un destino, pero no como una fuente.

- Obviamente, el parámetro que se pasa como de tipo '**out**' debe ser algo que pueda almacenarse (esto es, una variable de algún tipo).

Mecanismos de Paso de Parámetros

- Los parámetros en modo '**out**' son muy eficientes: análogamente a los parámetros en modo '**in**', los parámetros escalares en modo '**out**', se copian y los parámetros compuestos pueden ser pasados por referencia o copiados.

- El modo restante de paso de parámetros es '**in out**'. Este modo se usa para parámetros que se fungirán tanto como una fuente, así como un destino dentro del subprograma.

Mecanismos de Paso de Parámetros

- Como en este caso, el parámetro es potencialmente un destino, debe tratarse de una variable de algún tipo, igual que en el caso de los parámetros de tipo '**out**'.

- Se usan los mismos métodos de implementación para los parámetros '**in out**': Para valores escalares, los valores se copian al invocarse y al retornarse. Es decir, corresponden básicamente al paso por valor-resultado.

- Los parámetros compuestos pueden pasarse por referencia o por valor resultado (esto lo elige el compilador).

Mecanismos de Paso de Parámetros

- Ada parece haber resuelto los problemas que vimos anteriormente en los mecanismos de paso de parámetros de FORTRAN, Algol-60 y Pascal.

- Ada proporciona una solución más ortogonal al problema de pasar parámetros, al separar las funciones independientes que se involucran en esta tarea.

Mecanismos de Paso de Parámetros

- Dichas funciones son:

- 1) Cómo se usará el parámetro (o sea, entrada, salida, o entrada y salida).

- 2) Cómo se implementará la transmisión del parámetro (o sea, paso por referencia, paso por valor o paso por valor-resultado).

Mecanismos de Paso de Parámetros

- El primero es un asunto de índole “lógica”.
- Esto es, afecta el comportamiento de entrada-salida del programa.
- El segundo es un asunto de “desempeño”.
- Es decir, afecta la eficiencia del programa.

Mecanismos de Paso de Parámetros

- Ada permite al programador resolver el primer asunto (especificando el comportamiento deseado para un cierto parámetro) y reserva al compilador el derecho de resolver el segundo.

Mecanismos de Paso de Parámetros

- Nótese que con las reglas que establece para el paso de parámetros, Ada está mezclando ligeramente otras dos cosas que debieran ser ortogonales: el asunto de pasar por referencia o por valor y el tipo de parámetro que se pase.

- Esto es porque se está diciendo explícitamente que los parámetros escalares siempre se pasarán por valor.

Mecanismos de Paso de Parámetros

- ¿Cómo decide el compilador si un cierto parámetro se debe pasar por valor o por referencia? Esto es fácil de analizar.

- Supongamos que un parámetro compuesto ocupa “s” palabras de memoria y que cada componente del parámetro ocupa una palabra de memoria.

Mecanismos de Paso de Parámetros

- Después, supongamos que durante la ejecución de un subprograma, los componentes de un parámetro se acceden “n” veces.

- Podemos calcular C, que es el costo de pasar el parámetro por valor, y R que es el costo de pasarlo por referencia.

Mecanismos de Paso de Parámetros

$$C=2s+n$$

$$R=2n+1$$

- Cuando el parámetro es copiado, se requieren 2s referencias de memoria para transmitirlo y una referencia de memoria para cada uno de los “n” accesos posteriores a los componentes.

Mecanismos de Paso de Parámetros

- Si el parámetro se pasa por referencia, entonces se requiere una referencia de memoria para transmitirlo (suponiendo que una dirección de memoria cabe en una unidad de almacenamiento) y se requieren dos referencias de memoria por cada uno de los “n” accesos posteriores a los componentes (suponiendo que el compilador no optimice este proceso).

Mecanismos de Paso de Parámetros

- Ahora podemos determinar las condiciones bajo las cuales resulta menos costoso pasar un parámetro por referencia que por valor.

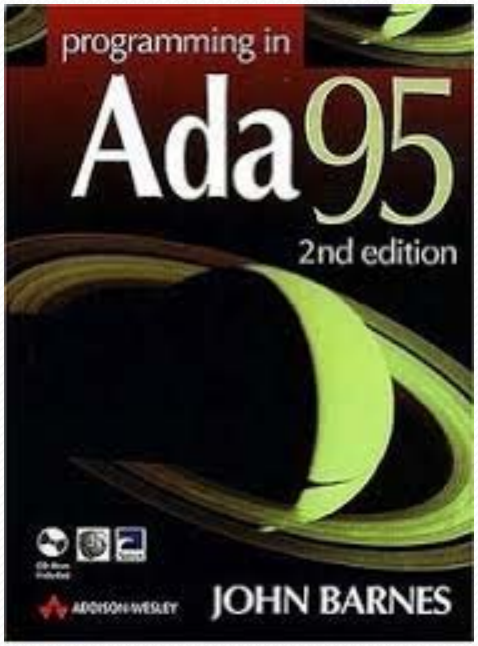
Mecanismos de Paso de Parámetros

- R<C cuando 2n+1 < 2s+n
- Por lo tanto, R<C cuando n<2s-1
- Desafortunadamente, “n” depende del comportamiento dinámico del programa.

Mecanismos de Paso de Parámetros

- Por lo tanto, es usualmente imposible para un compilador poder determinar su tamaño.
- En consecuencia, la decisión de si usar paso de parámetros por valor o referencia debe basarse en algunas premisas razonables.

Mecanismos de Paso de Parámetros



- Por ejemplo, si suponemos que el parámetro es un arreglo y que cada elemento del arreglo es referenciado una vez, entonces n=s (si cada elemento ocupa una unidad de almacenamiento), C=3n y R<C.

Mecanismos de Paso de Parámetros

- De tal forma, el paso por referencia es menos costoso en este ejemplo.
- Advierta que R>C cuando n>2s-1.
- Por lo tanto, copiar es menos costoso si, en promedio, cada elemento del arreglo es accedido al menos dos veces.

Parámetros independientes de la posición y por omisión

- Cuando desarrollamos bibliotecas de funciones, es común tener situaciones en las que tenemos un gran número de parámetros.

- En este caso, el programador que las usa tienen que memorizar sus posiciones particulares para poder accederlas correctamente.

Parámetros independientes de la posición y por omisión

- Esta tarea es muy tediosa y propensa a errores. Por ejemplo:

procedure DRAW_AXES(X_ORIGIN, Y_ORIGIN : COORD;
X_SCALE, Y_SCALE : FLOAT; X_SPACING, Y_SPACING : NATURAL;
X_LOGARITHMIC, Y_LOGARITHMIC : BOOLEAN;
X_LABELS, Y_LABELS : BOOLEAN; FULL_GRID : BOOLEAN);

Parámetros independientes de la posición y por omisión

- Una invocación a DRAW_AXES podría verse de la forma siguiente:

- DRAW_AXES (500, 500, 1.0, 0.5, 10, 10,
  FALSE, TRUE, TRUE, TRUE, FALSE);

Resulta muy difícil decir los significados de estos parámetros a partir de esta invocación, y ciertamente se debe recurrir a la definición del procedimiento para averiguar esto.

Parámetros independientes de la posición y por omisión

- Inspirándose en una idea propuesta originalmente en Sistemas Operativos, Ada permite el uso de parámetros independientes de su posición.

- Bajo este esquema, cada parámetro es identificado por su nombre, pero la posición en la cual se le coloca es irrelevante. Veamos un ejemplo con el procedimiento DRAW_AXES que definimos anteriormente.

Parámetros independientes de la posición y por omisión

```
DRAW_AXES(X_ORIGIN => 500, Y_ORIGIN => 500,
X_SPACING => 10, Y_SPACING => 10, FULL_GRID =>
   FALSE,
X_SCALE => 1.0, Y_SCALE => 0.5,
X_LABEL => TRUE, Y_LABEL => TRUE,
X_LOGARITHMIC => FALSE, Y_LOGARITHMIC => TRUE);
```

Parámetros independientes de la posición y por omisión

- Esto es más legible y mucho menos propenso a errores que la versión en la que los parámetros dependen de su posición:

  DRAW_AXES (500, 500, 1.0, 0.5, 10, 10, FALSE, TRUE, TRUE, TRUE, FALSE);

Parámetros independientes de la posición y por omisión

- Los parámetros independientes de su posición siguen el **Principio de Etiquetamiento**.

Evite secuencias arbitrarias de más de unos cuantos elementos de largo; no requiera que el programador sepa la posición absoluta de un elemento en una lista. En vez de eso, asocie una etiqueta significativa con cada elemento y deje que los elementos sean colocados en cualquier orden.

Parámetros independientes de la posición y por omisión

- Adicionalmente, Ada permite el uso de parámetros por omisión.

- Este mecanismo permite al usuario asignar valores únicamente a aquellos parámetros que necesitan ser definidos de manera diferente de algún valor base definido en la declaración del procedimiento.

Parámetros independientes de la posición y por omisión

- La motivación para este mecanismo es simple. En un intento por ser general, DRAW_AXES proporciona muchas opciones diferentes.
- Sin embargo, varias de estas opciones raramente se usarán.
- Por ejemplo, la mayor parte de los usuarios no querrán una rejilla completa o ejes logarítmicos.

Parámetros independientes de la posición y por omisión

- Desafortunadamente, los usuarios deben especificar todas las opciones, aunque sea para desactivarlas.
- Esto es una violación al **Principio de los Costos Localizados**, que dice que los usuarios no debieran pagar por lo que no usan.
- Los parámetros por omisión son una solución a este problema, pues pueden omitirse parámetros que, en la mayor parte de los casos, no se requieren. Veamos un ejemplo en el acetato siguiente.

Parámetros independientes de la posición y por omisión

```
procedure DRAW_AXES (X_ORIGIN, Y_ORIGIN : COORD
   := 0;
   X_SCALE , Y_SCALE : REAL := 1.0;
   X_SPACING, Y_SPACING : NATURAL := 1;
   X_LABEL, Y_LABEL : BOOLEAN := TRUE;
   X_LOGARITHMIC, Y_LOGARITHMIC : BOOLEAN :=
   FALSE;
   FULL_GRID : BOOLEAN := FALSE );
```

Parámetros independientes de la posición y por omisión

- Esto nos permite escribir las invocaciones a DRAW_AXES de forma mucho más compacta:

```
DRAW_AXES (500, 500, Y_SCALE => 0.5,
Y_LOGARITHMIC => TRUE,
                X_SPACING => 10, Y_SPACING => 10);
```

- Nótese en este ejemplo que estos dos mecanismos (los parámetros independientes de su posición y los parámetros por omisión) pueden mezclarse en la misma aplicación.

Parámetros independientes de la posición y por omisión

- Esta flexibilidad adicional, sin embargo, tiene algunos costos extra, puesto que se incrementa fácilmente la complejidad del lenguaje.

- Además, hay un problema más serio de interacción de funciones asociado con estos mecanismos cuando se usa sobrecarga de procedimientos.

- Este problema no resulta fácil de detectar.

Parámetros independientes de la posición y por omisión

- Recordemos que un procedimiento sobrecargado puede tener varios significados en un contexto.
- El compilador determina el significado con base en el tipo de sus parámetros y en el contexto de uso del subprograma.
- Consideremos el ejemplo que se muestra en el siguiente acetato.

Parámetros independientes de la posición y por omisión

```
procedure P (X : INTEGER; Y : BOOLEAN := FALSE);

procedure P (X : INTEGER; Y : INTEGER := 0);
```

Parámetros independientes de la posición y por omisión

- El procedimiento P está sobrecargado porque tiene dos significados posibles.

- Las reglas para la identificación de operadores nos dicen que P(9,TRUE) es una llamada al primer procedimiento y que P(5,8) es una llamada al segundo procedimiento.

Parámetros independientes de la posición y por omisión

- Advierta sin embargo que hemos proporcionado un valor por omisión para Y en las declaraciones de ambos procedimientos.
- La pregunta es: ¿cuál es el significado de P(3)?
- Podría ser cualquier de los dos procedimientos, ya que omitimos el único parámetro que los distingue.

Parámetros independientes de la posición y por omisión

- De hecho, dado que P(3) es una invocación ambigua, Ada no permite las dos declaraciones procedurales antes indicadas.

- Un conjunto de declaraciones de subprogramas es ilegal si introduce el potencial de invocaciones ambiguas.

Parámetros independientes de la posición y por omisión

- Aunque esto tiene sentido, en la práctica no siempre es tan obvio ver que dos declaraciones procedurales son potencialmente ambiguas.
- Consideremos el ejemplo que se muestra en el acetato siguiente.

Parámetros independientes de la posición y por omisión

```
type PRIMARY is (RED, BLUE, GREEN);
type STOP_LIGHT is (RED, YELLOW, GREEN);

procedure SWITCH (COLOR : PRIMARY; X:FLOAT;
Y:FLOAT);

procedure SWITCH (LIGHT : STOP_LIGHT; Y:FLOAT;
X:FLOAT);
```

Parámetros independientes de la posición y por omisión

- Estas dos declaraciones lucen muy diferentes, y no tienen parámetros por omisión.

- Sin embargo, la invocación:

```
SWITCH (RED, X=> 0.0, Y=> 0.0);
```

es ambigua y, por tanto, las declaraciones son ilegales.

Parámetros independientes de la posición y por omisión

- Esta ambigüedad es el efecto de la interacción de dos sobrecargas: una en las enumeraciones y otra en los procedimientos.

Parámetros independientes de la posición y por omisión

- Esto ilustra el nivel de complejidad que puede producirse en Ada como consecuencia de la interacción de funciones. De hecho, las reglas que especifican el tipo de sobrecargas que son permisibles son un poco más complicadas de lo que hemos visto.

- Claramente, un programa que haga uso excesivo de las sobrecargas combinadas con los parámetros independientes de posición y por omisión, resultan difíciles de entender para un humano y son también difíciles de compilar.

Concurrencia

- Cuando queremos efectuar una tarea, suele ser más eficiente y más conveniente, realizar varias cosas al mismo tiempo.
- Por ejemplo, una persona puede leer un mapa mientras otra conduce un auto.
- No tendría mucho sentido terminar de conducir cuando comienza a leerse el mapa, y tampoco sería muy eficiente leer el mapa en su totalidad, antes de comenzar a conducir.

Concurrencia

- Esto mismo aplica a ciertos programas. Por lo tanto, Ada proporciona un mecanismo que permite a un programa hacer más de una cosa a la vez.

- Consideremos un ejemplo. Supongamos que tenemos un sistema pequeño e independiente de procesamiento de textos que permite a los usuarios imprimir un archivo mientras editan otro.

- Esto se programa en Ada definiendo dos tareas disjuntas, llamadas PRINT y EDIT. Veamos el código del acetato siguiente.

```
procedure WORD_PROCESSOR is

   task EDIT; end EDIT;

   task body EDIT is
   begin
      ⋮    -- editar el archivo seleccionado
   end;

   task PRINT; end PRINT;

   task body PRINT is
   begin
      ⋮    --imprimir el archivo seleccionado
   end;

begin
   ⋮   --iniciar tareas y esperar a que se completen
end WORD_PROCESSOR
```

Concurrencia

- En este código entonces un procedimiento, llamado WORD_PROCESSOR, con dos tareas locales, EDIT y PRINT.

- Nótese que una tarea (**task**) se declara de manera muy similar a un paquete (**package**), con una especificación separada y un cuerpo.

- En este caso, no hay nombres públicos, de manera que no hay nada en la especificación.

Concurrencia

- Cuando invocamos a WORD_PROCESSOR, todas las tareas locales se inician automáticamente.
- Esto es, hacemos tres cosas a la vez: Comenzamos ejecutando los cuerpos de WORD_PROCESSOR, EDIT y PRINT.
- Podemos suponer que EDIT hace su trabajo de comunicarse con el usuario y editar el archivo, mientras que PRINT hace su trabajo de enviar otro archivo a la impresora.

Concurrencia

- ¿Qué hace WORD_PROCESSOR?
- En este caso, no mucho. Puesto que el cuerpo del procedimiento está vacío, encontramos inmediatamente el '**end**' e intentamos regresar. Ada, sin embargo, impide que un procedimiento retorne, mientras tenga tareas locales activas.
- Por tanto, WORD_PROCESSOR esperará, mientras EDIT y PRINT sigan ejecutándose.

Concurrencia

- Cuando estas dos tareas terminen (llegando a su '**end**' correspondiente), entonces WORD_PROCESSOR podrá regresar a su invocador.

- ¿Por qué Ada requiere que todas las tareas locales terminen antes de que un procedimiento pueda terminar su ejecución?

- Supongamos que WORD_PROCESSOR tiene algunas variables locales. Dichas variables son visibles a EDIT y PRINT, porque están declaradas en el mismo ambiente.

Concurrencia

- Consideremos ahora lo que ocurre cuando WORD_PROCESSOR termina. Cuando eso ocurre, su registro de activación se borra. Por tanto, cualquier referencia a las variables locales que exista en EDIT y PRINT, ya no tendrán significado. A este problema se le conoce como de las referencias colgadas (*dangling references*).

- Una opción sería preservar el registro de activación de WORD_PROCESSOR hasta que EDIT y PRINT hayan terminado.

Concurrencia

- Sin embargo, esto evitaría usar una simple pila para asignar y liberar los registros de activación.
- La alternativa es retrasar la terminación de WORD_PROCESSOR hasta que esto pueda realizarse de manera segura.
- Esta decisión ilustra el **Principio de Seguridad**.

Concurrencia

- Hemos visto un ejemplo de dos tareas que se ejecutan concurrentemente, pero que no se comunican.

- Sin embargo, esto no suele ocurrir. Regresemos a nuestro ejemplo en el que una persona conduce y otra lee un mapa. Para que estas dos tareas concurrentes resulten útiles, la persona que lee el mapa debe ser capaz de comunicarle instrucciones a quien conduce. De hecho, podría incluso ocurrir que el conductor tenga que detenerse para que quien lee el mapa pueda decidir en dónde darán vuelta.

Concurrencia

- Esto mismo ocurre en programación. Supongamos que tenemos una aplicación que recupera registros de una base de datos, las resume e imprime los resultados.

- Como los procesos de recuperación y de resumir son bastante independientes, podemos implementarlos como tareas que se ejecutan concurrentemente.

- Sin embargo, estas dos tareas deben comunicarse. La tarea SUMMARY le tendrá que decir a la tarea RETRIEVAL qué registro quiere y la tarea RETRIEVAL le tiene que transmitir los registros del proceso SUMMARY cuando se encuentren. El código correspondiente se muestra en el acetato siguiente.

```
procedure DB_SYSTEM is
        task SUMMARY; end SUMMARY;

        task body SUMMARY is
        begin
          ⋮  --genera el resumen
        end;

        task RETRIEVAL;
          entry SEEK (K: KEY);
          entry FETCH (out R: RECD);
        end RETRIEVAL;

        task body RETRIEVAL is
        begin
          ⋮  --buscar registro y regresarlo
        end;

begin
 ⋮    ..esperar a que se completen las tareas locales
end DB_SYSTEM;
```

Concurrencia

- Adviértase que la especificación de RETRIEVAL contiene dos declaraciones de tipo '**entry**'.

- Podemos ver que dichas declaraciones tienen parámetros de manera muy similar a los procedimientos y, de hecho, pueden ser invocados como procedimientos.

Concurrencia

- Por ejemplo, el cuerpo de SUMMARY puede lucir de la forma siguiente:

```
task body SUMMARY is
begin
  ⋮
  SEEK (ID);
  ⋮
  FETCH (NEW_RECD);
  ⋮
end SUMMARY;
```

Concurrencia

- La llamada a SEEK le dice a RETRIEVAL que encuentre el registro en la base de datos y la llamada a FETCH coloca su valor en NEW_RECD.

- Aunque los procedimientos y las entradas (**entries**) son similares de alguna forma, hay diferencias importantes.

- Recordemos cómo funciona una llamada normal a un procedimiento: cuando un subprograma llama a otro, los parámetros se transmiten del invocador al invocado, el invocador es suspendido y se activa al invocado.

Concurrencia

- El invocador permanece suspendido hasta que el invocado regresa. En ese momento, los resultados se transmiten del invocado de vuelta al invocador, se desactiva al invocado y se continúa la ejecución en el invocador (o sea, se le reactiva).

- Cuando una tarea invoca a una entrada (**entry**) en otra, transmite los parámetros de manera muy similar a como se hace con los procedimientos. La diferencia es que una vez que el invocado acepta los parámetros, el invocador continúa ejecutándose: no se suspende.

Concurrencia

- Las dos tareas permanecen activas y continúan ejecutándose concurrentemente.
- Por tanto, la invocación 'SEEK(ID)' se puede ver más adecuadamente como una operación de envío de mensajes en la que el mensaje (ID) se coloca en '**entry** SEEK' donde está disponible para su recuperación (RETRIEVAL).
- De hecho, una entrada (**entry**) es llamada frecuentemente *mailbox* o *message port*.

Concurrencia

- ¿Cómo acepta mensajes una tarea? Esto se logra con la sentencia **accept**, cuya sintaxis es la siguiente:

accept <nombre> (<parámetros>) **do** <sentencias> **end** <nombre>;

Por ejemplo, el cuerpo de RETRIEVAL se vería como se muestra en el acetato siguiente.

Concurrencia

```
task body RETRIEVAL is
begin
  loop
        accept SEEK (K:KEY) do
                RK := K;
        end SEEK;
          ⋮    --buscar el registro RK y colocarlo en RECD_VALUE
        accept FETCH (out R: RECD) do
                R:=RECD_VALUE;
        end FETCH;
    end loop;
end RETRIEVAL;
```

Concurrencia

- Nótese que RETRIEVAL está escrito como un ciclo que buscar alternativamente un registro y regresa su valor.
- La primera sentencia **accept** acepta un mensaje del buzón de SEEK y asocia el parámetro K a este mensaje.
- ¿Qué pasa si RETRIEVAL alcanza esta sentencia **accept** antes de que SUMMARY haya enviando el mensaje?

Concurrencia

- Se suspende a sí mismo, esperando el arribo de un mensaje en el buzón SEEK. Posteriormente, cuando SUMMARY envía el mensaje, será aceptado y ambas tareas procederán concurrentemente.

- De manera similar, si SUMMARY intenta transmitir SEEK(ID) antes de que RETRIEVAL esté listo para aceptarla, SUMMARY será suspendido hasta que RETRIEVAL acepte un mensaje de SEEK.

Concurrencia

- Por lo tanto, un mensaje se transmite realmente cuando las entradas a las que el invocador llama y desde las que el receptor acepta, son las mismas.

- A este mecanismo se le conoce como "**rendez-vous**" en la literatura de Ada.

- Este mecanismo puede nunca llevarse a cabo, en caso de que una de las tareas espere para siempre. A eso se le llama *deadlock*.