Lenguajes de Programación

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

Ciclos

- El ciclo **for** puede usarse como una extensión convencional del ciclo **DO** en FORTRAN:

```
for i:=1 step 2 until NM do
        interior[i]:=exterior[NM-i];
```

Ciclos

- También puede trabajar como un **while** de Pascal:

```
for NewGuess:=Improve(OldGuess)
        while abs(NewGuess-OldGuess)>TOL
do OldGuess:=NewGuess;
```

Ciclos

- Lo que corresponde al siguiente ciclo en Pascal:

```
NewGuess:=Improve(OldGuess);
while abs(NewGuess-OldGuess) > TOL do
        OldGuess:=NewGuess;
        NewGuess:=Improve(OldGuess);
end;
```

Ciclos

- También pueden usarse valores explícitos:

for days:=31,

if mod(year, 4)=0 **then** 29 **else** 28,

31, 30, 31, 30, 31, 31, 30, 31, 30, 31 **do** . . .

Ciclos

- El segmento anterior de código hace que la variable “days” tome los valores del número de días de cada mes y permite decidir si febrero tendrá 28 ó 29 días (dependiendo de si el año es bisiesto o no).

Ciclos

- Pero el **for** permite cosas bastante extrañas como la siguiente:

```
for i:= 3, 7,
        11 step 1 until 16,
        i/2 while i ≥ 1,
        2 step i until 32
do print (i)
```

Ciclos

- Este programa imprime los números siguientes:

```
3 7 11 12 13 14 15 16 8 4 2 1 2 4 8 16 32
```

Ciclos

- Un problema más serio con el **for** de ALGOL-60 es que permite que cualquier expresión dentro de la lista actual de elementos del ciclo sea re-evaluada en cada iteración.

- Esto significa que, bajo ciertas condiciones, el programador puede no tener idea del número de iteraciones que realizará un ciclo.

Ciclos

- Lo que es peor es que aunque no se requiera de este tipo de cosas, el compilador las hará siempre.
- Esto significa que el costo del ciclo será siempre el mismo sin importar lo que hagamos, ya que los parámetros de un ciclo son siempre re-evaluados.

Ciclos

- Esto viola el **Principio de los Costos Locales**, el cual dice:

> Los diseñadores de un lenguaje deben evitar mecanismos cuyo costo se distribuya a través de todos los programas, sin importar si dicho mecanismo se usa o no.

Ciclos

- Resulta difícil imaginar las razones por las cuales un programador podría requerir una estructura de control tan elaborada como ésta.

- Por ello, en la tercera generación de los lenguajes de programación se regresó a la simplicidad, haciendo uso de estructuras de control menos elaboradas y con reglas más claras.

Sentencias Anidadas

- Los diseñadores de ALGOL se percataron de que debía permitirse a todas las estructuras de control el gobernar un número arbitrario de sentencias.
- Así fue como nació el concepto de sentencias delimitadoras (*bracketing*) en ALGOL-58.

Sentencias Anidadas

- Esto quiere decir que cada estructura de control (p.ej., **if-then**) era considerada como un paréntesis que abre, el cual debía tener un paréntesis que cerrara (p.ej., **end if**).

- Más adelante, durante el diseño del ALGOL-60, y como resultado en gran medida de ver las descripciones de ALGOL-58 en BNF, se percataron de que era suficiente con tener un solo constructor delimitador para todos los casos.

Sentencias Anidadas

- Para delimitar el alcance de todas las sentencias, decidieron usar **begin..end**.
- Así se definió un tipo de sentencia especial, llamada *sentencia compuesta*, la cual podía agrupar cualquier número de sentencias y convertirlas en una sola.

Sentencias Anidadas

- Por ejemplo:

```
begin
        sentencia 1;
        sentencia 2;
            .
            .
            .
        sentencia n
end
```

Sentencias Anidadas

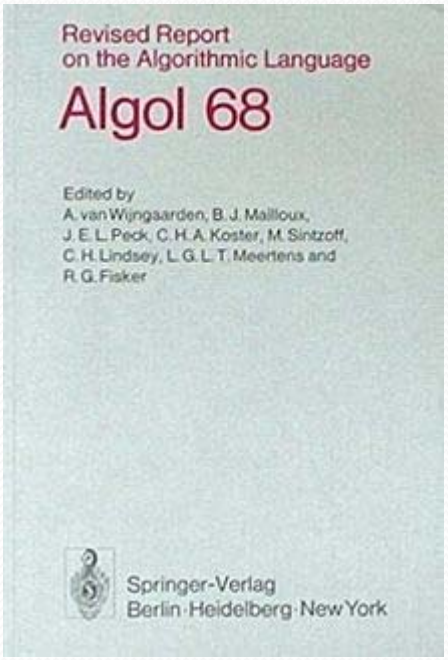
- En el ejemplo anterior, todo lo que está dentro del **begin. .end** se considera una sola sentencia, y se puede usar como tal en cualquier parte de nuestro programa.

Sentencias Anidadas

- El problema es que el **begin. .end** ya se utiliza en ALGOL-60 para delimitar bloques, los cuales definen entornos anidados.

- Este doble uso del **begin. .end** viola el **Principio de Ortogonalidad**, porque dos mecanismos independientes (el agrupamiento de sentencias y la definición de un entorno) son efectuados por el mismo constructor.

Sentencias Anidadas



- Otro problema con esto es que un programador podría olvidar incluir el **begin. .end** y con ello se altera sustancialmente el significado de la sentencia (p.ej., dentro de un ciclo).

Sentencias Anidadas

- Ejemplo:

```
for i:=1 step 1 until N do
        LeeArreglo(valor);
        Datos[i]:=if valor < 0 then -valor else valor;
for i:=1 step . . .
```

Sentencias Anidadas

- Esa es la razón por la que resulta mejor práctica usar siempre **begin. .end**, aunque se coloque una sola sentencia entre estos delimitadores.
- Pero hay otro problema. Si existen muchas sentencias anidadas (p.ej., muchos ciclos anidados) resulta confuso poder distinguir qué **end** corresponde a qué **begin**.
- ¿Qué mejoras puede sugerir a los delimitadores para evitar este problema?

Entorno estático vs. dinámico

- Existen 2 formas principales de implementar las reglas de entorno (*scoping rules*) en los lenguajes estructurados: entorno estático y entorno dinámico.

- Si se usa un **entorno estático**, un procedimiento es invocado en el ambiente en que fue definido. En otras palabras, el significado de las sentencias y expresiones bajo un entorno estático se determinan por la estructura estática del programa.

Entorno estático vs. dinámico

- ALGOL-60, Pascal, C, Scheme y la mayoría de los lenguajes de programación utilizan reglas de entorno estático.

- Si se usa un **entorno dinámico**, un procedimiento es invocado en el ambiente de su invocador. En otras palabras, el significado de las sentencias y expresiones bajo un entorno dinámico se determinan por la estructura dinámica de los procesos computacionales que evolucionan con respecto al tiempo.

Entorno estático vs. dinámico

- Las primeras implementaciones de LISP usaban reglas de entorno dinámico.
- En general, los investigadores de lenguajes de programación tienden a favorecer las reglas de entorno estático, ya que son más fáciles y claras de entender para los programadores.

Entorno estático vs. dinámico

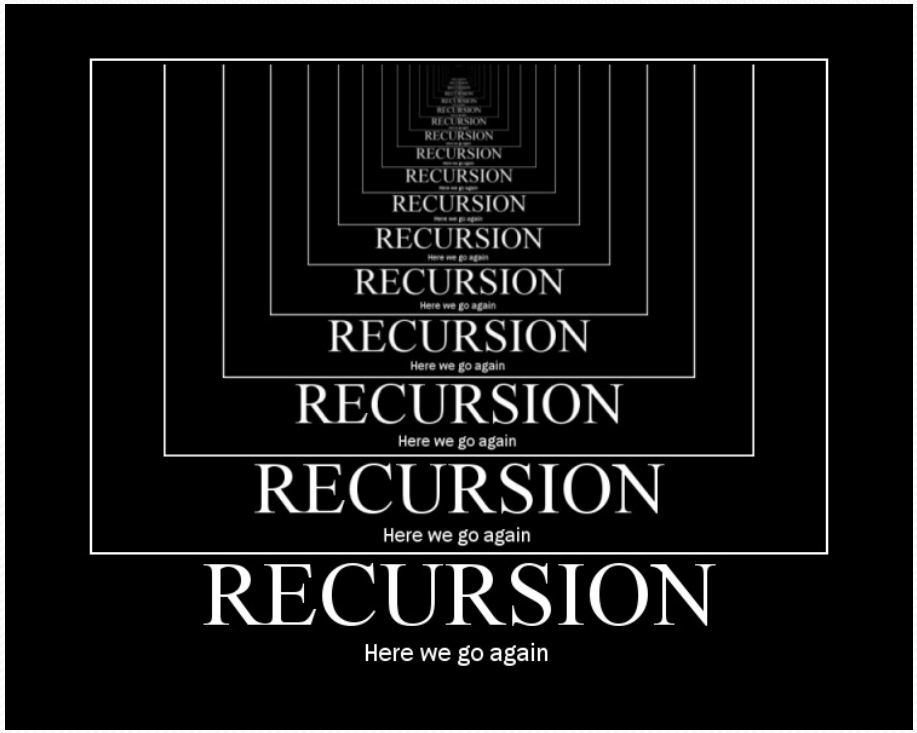
- Además, se cree que las reglas de entorno estático conducen a una programación más confiable que las reglas de entorno dinámico.

- Sin embargo, hay algunos defensores de las reglas de entorno dinámico, a pesar de que el mismo McCarthy (autor de LISP) llegó a decir que este mecanismo se originó accidentalmente durante el diseño de LISP.

Recursividad

- ALGOL-60 permitía procedimientos recursivos (o sea, procedimientos que se llamaban a sí mismos).
- Las definiciones recursivas son muy comunes en computación y en matemáticas, por lo que ésta es una contribución muy importante del lenguaje.

Recursividad



- La recursividad permite un mayor nivel de abstracción en lo que a diseño de algoritmos se refiere, e incluso abrió la posibilidad de crear un paradigma completamente nuevo (la programación funcional).

Recursividad

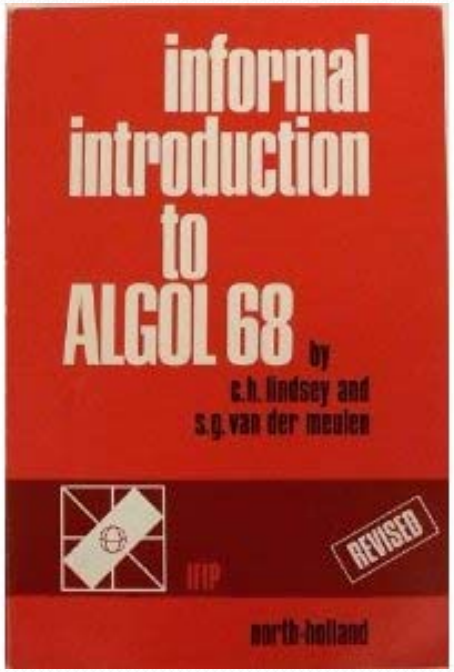
- Un ejemplo de recursividad en ALGOL:

```
integer procedure fac(n);
  value n; integer n;
  fac:=if n = 0 then 1 else n * fac(n-1);
```

Recursividad

- La naturaleza recursiva de los programas en ALGOL-60 requería de un nuevo modelo de ejecución (no podemos traducir el código línea por línea a ensamblador) dado que no hay recursividad en lenguaje máquina.
- La pregunta es entonces: ¿cómo traducimos la recursividad a lenguaje máquina?
- La solución más simple es convertir los programas recursivos a programas iterativos usando una pila.

Recursividad

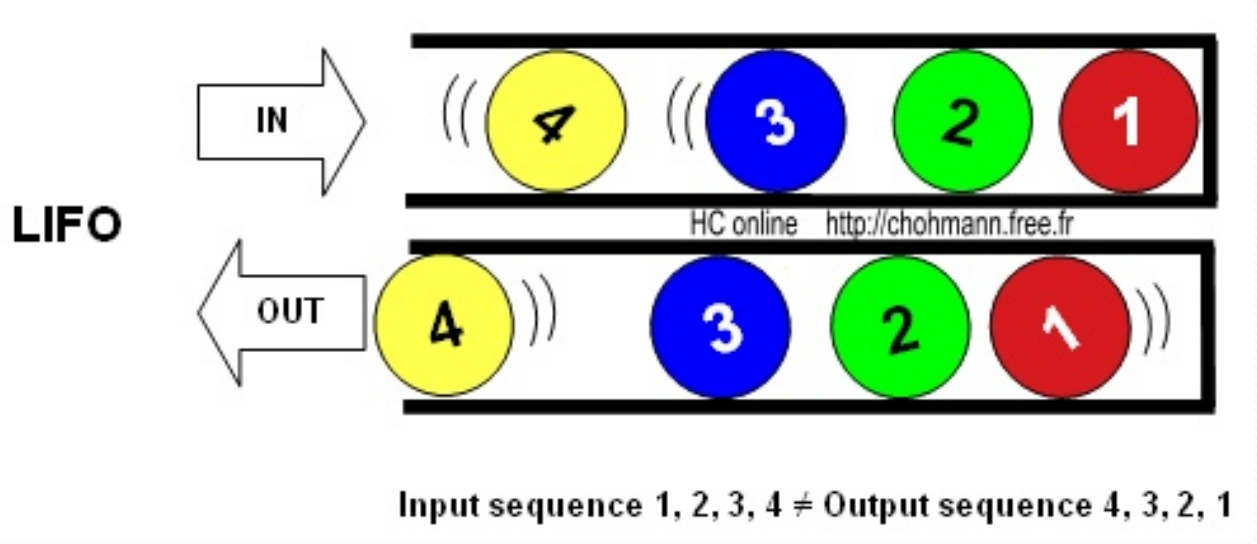


- FORTRAN usaba una pila para evaluar sus expresiones aritméticas, así que podemos simplemente usarla ahora en tiempo de ejecución para las llamadas a procedimientos.

Recursividad

- La pila debe tener espacio para almacenar lo siguiente:
  - Parámetros
  - Variables locales
  - Dirección de retorno
  - Áreas para almacenar el estado actual cuando se llame a otro subprograma

Recursividad



- Puesto que no sabemos cuántos necesitaremos por cada subprograma, tenemos que asignar un nuevo registro de activación por cada llamada recursiva y tenemos que liberarlo por cada regreso de una invocación (LIFO, o sea, el último en entrar es el primero en salir).

Recursividad

- Por lo tanto, se requiere de una pila de registros de activación.

- Puesto que el tamaño de los arreglos locales se determina al entrar a un bloque, el tamaño del registro de activación puede variar de una invocación a otra.

Recursividad

- Puesto que se permiten definiciones locales en los bloques, y dado que el tamaño de un arreglo puede ser distinto cada vez que se entra a un bloque, necesitamos un registro de activación para cada bloque.

- Sin embargo, en este caso (de los arreglos) no se requiere espacio para parámetros ni para la dirección de regreso.

Recursividad

- Los arreglos de ALGOL-60 son semi-dinámicos porque aunque sus límites se definen en tiempo de ejecución, una vez que se determina un cierto tamaño para ellos, éste no puede cambiar.

- ALGOL-68 permitía arreglos dinámicos, ya que éstos podían crecer o disminuir de tamaño después de su definición inicial. Debe mencionarse, sin embargo, que este tipo de arreglos es mucho más difícil de implementar usando pilas.

Recursividad

- Además, los arreglos dinámicos no son vitales en un lenguaje de programación (puede manejarse memoria dinámicamente de maneras más eficientes).

- Por tanto, el tipo de arreglos con que contaba ALGOL-60 era un buen compromiso de diseño.

El controversial GOTO

- La interacción de funciones es uno de los problemas más difíciles de verificar cuando se diseña un lenguaje de programación y no importa qué tan bueno sea nuestro diseño, tarde o temprano acabamos por padecerla en algunas ocasiones.

El controversial GOTO

- Un ejemplo clásico de este problema en ALGOL-60 es el uso de **GOTO**s fuera de bloque:

```
a:begin array X[1:100];
        .
        b:begin array Y[1:100];
        .
                goto exit;
        .
          end;
        .
    exit:
end
```

El controversial GOTO

- En este caso, puesto que la etiqueta “exit” está declarada en el bloque (a), el cual contiene al bloque (b), entonces la etiqueta es visible para la sentencia **goto** del bloque (b).

- Analicemos el efecto de ejecutar esta sentencia **goto**.

El controversial GOTO

- Cuando esto ocurre, habrán registros de activación de ambos bloques (el (a) y el (b)) en la pila.

- Si saliéramos del bloque (b) de manera normal (es decir, a través del **end**), el registro de activación de (b) se borraría, dejando el de (a) en la parte superior de la pila.

El controversial GOTO

- Si salimos del bloque (b) haciendo un salto, se tiene que hacer lo mismo: el registro de activación de (b) debe borrarse de la pila.

- Esto significa que el proceso de ejecución de un **goto** en Algol puede ser mucho más complicado que un simple salto absoluto en lenguaje máquina, puesto que puede requerir el tener que salir a través de varios niveles de bloques.

El controversial GOTO

- El problema se agrava aun más si usamos recursividad. Por ejemplo:

```
begin
        procedure P(n);
          value n; integer n;
          if n=0 then goto out
          else P(n-1);

        P(3);
    out:
    end
```

El controversial GOTO

- ¿Cuántos registros de activación deben borrarse en este caso? Se deben borrar 3 antes de saltar a la etiqueta “out”.

- El problema es que el compilador puede no saber esto con anticipación, ya que la cantidad de registros de activación a borrarse depende de los datos que se estén usando.

El controversial GOTO

- Sin embargo, sabemos que este proceso puede ser muy costoso y se trata realmente de una interacción producida por efecto de la recursividad y el hecho de que se permita usar **goto**s en cualquier parte de un programa, a fin de mantener la regularidad del lenguaje.

El controversial GOTO

- Claro que siempre puede argumentarse que es una mala práctica de programación escribir código como el antes mostrado.

- Sin embargo, dado que existe el potencial para que dicho código se escriba, eso debe considerarse dentro del diseño de un lenguaje de programación.

La sentencia **switch**

- Existe una sentencia para manejo de casos en ALGOL-60 llamada **switch**, la cual es realmente una extensión del **GOTO** calculado de FORTRAN.

La sentencia **switch**

```
begin
                switch rango = bajo, medio, alto, muyalto;          .

                goto rango[i];
        bajo :  . . . Manejar caso para bajo . . .
                goto acabe;
        medio : . . . Manejar caso para medio . . .
                goto acabe;
        alto :  . . . Manejar caso para alto . . .
                goto acabe;
        muyalto :       . . . Manejar caso para muyalto . . .

        acabe : . . .
end
```

La sentencia **switch**

- Esto funciona como si ‘rango’ fuera un arreglo inicializado con las etiquetas ‘bajo’, ‘medio’, ‘alto’ y ‘muyalto’. La sentencia **goto** realiza la transferencia a la etiqueta definida por rango[i].

- El problema es que esta declaración de **switch** es difícil de entender y resulta ser altamente dependiente del uso del **goto**, lo cual es malo.

La sentencia **switch**

- Este problema podría resolverse si se usa un caso por omisión para todas las decisiones (como en C), en vez de usar varios **goto**'s.

- Asimismo, el hecho de que no es necesario hacer converger todos los casos al final de las opciones, podría hacer que los programadores produjeran código todavía más difícil de entender.

La sentencia **switch**

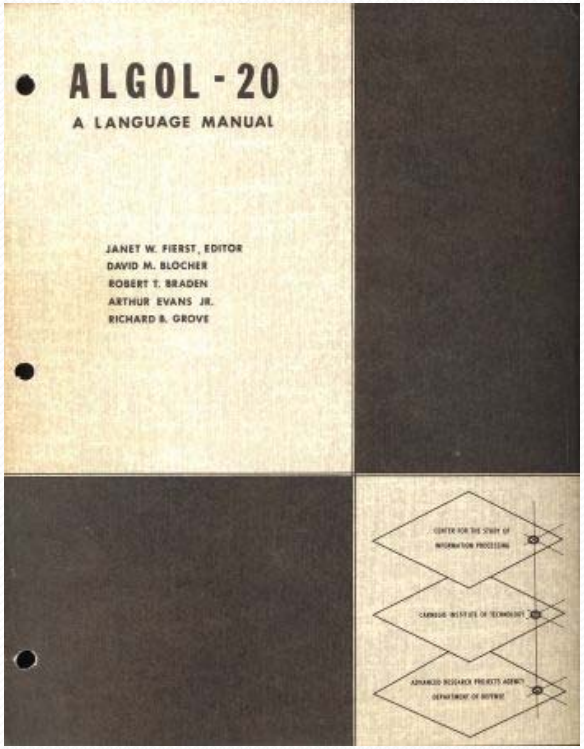
- La sentencia **switch** también es barroca. Considere el siguiente ejemplo:

```
begin
        switch S = L, if i > 0 then M else N, Q;
        goto S[j];
end
```

La sentencia **switch**

- Si j=1, entonces el **goto** nos transfiere a la etiqueta L.
- Si j=3, entonces nos transfiere a la etiqueta Q.
- Si j=2, entonces el **goto** nos transfiere a la etiqueta M o N, dependiendo del valor de i.

La sentencia **switch**



- Este es un abuso de la notación y podría llevarnos fácilmente a confusiones e interpretaciones erróneas del código.

La sentencia **switch**

- Otra cosa interesante de mencionar es que si se incluye una sentencia del tipo:

goto S[k]

- y ‘k’ está fuera de rango (o sea, es menor de 1, o es mayor que el número de opciones contenidas en S), el efecto que se produce es que se pasa el control a la siguiente sentencia.

La sentencia **switch**

- En otras palabras, un **switch** fuera de rango se interpreta como un fracaso en la selección de opciones (un *fall through*).
- Esto se debe a que no contamos con un caso por omisión.
- Qué implicaciones tiene esto con respecto al **Principio de Seguridad**?

Mecanismos de paso de parámetros

- Paso por valor
- Paso por referencia
- Paso por nombre

Mecanismos de paso de parámetros

- **Paso por valor:** Al momento de la invocación, se pasa una copia del parámetro real al subprograma.
- Por ende, cualquier cambio que éste sufra dentro del subprograma no será propagado al invocador.
- En otras palabras, ningún cambio al parámetro afecta al invocador.

Mecanismos de paso de parámetros

- **Paso por referencia:** Se pasa un apuntador a la dirección de memoria del valor que sirve de parámetro al subprograma invocado.

- La referencia no cambiará, pero el valor que contiene sí puede hacerlo.

- En otras palabras, cualquier cambio al parámetro en el subprograma, afectará al invocador.

Mecanismos de paso de parámetros

- **Paso por nombre:** Se pasa una liga simbólica (llamada "thunk" en inglés) del invocador al invocado.

- Aunque esta liga no puede variar, cada vez que se evalúa puede regresar una referencia diferente y por tanto, un valor diferente.

Mecanismos de paso de parámetros

- En su afán por intentar resolver los problemas de FORTRAN, ALGOL-60 proporcionaba 2 mecanismos para paso de parámetros: por valor y por nombre.

- El paso de parámetros por nombre es un mecanismo inusual en los lenguajes de programación modernos. La idea era proporcionar un mecanismo similar al paso por referencia.

Paso por nombre

- Sin embargo, el paso por nombre no funciona igual que el paso por referencia.
- A continuación mostraremos un ejemplo de cómo difieren los 2 mecanismos entre sí.

Paso por nombre

```
SUBROUTINE S(EL, K)
K=2
EL=0
RETURN
END
```

Veamos el siguiente segmento de código (en FORTRAN)

```
A(1) = A(2) = 1
I=1
CALL S(A(I), I)
```

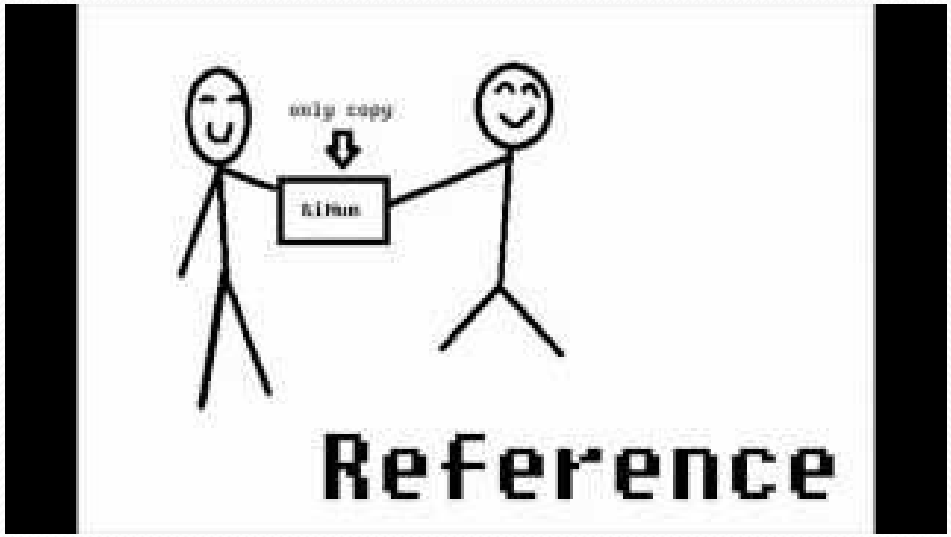
Paso por nombre

- Si pasamos estos parámetros por referencia (como hace FORTRAN), al regresar de la invocación de S, tendremos:

A(1)=0

I = 2

Paso por nombre



- Es importante hacer notar cómo el hecho de haber cambiado el valor de 'I' no tuvo efecto en 'A(I)', puesto que la referencia a 'A(I)' se calculó en el momento de la invocación y no se recalcula al regreso de la subrutina.

Paso por nombre

- Veamos ahora la misma subrutina en ALGOL-60:

```
procedure S(el,k)
        integer el,k;
        begin
          k:=2;
          el:=0;
        end;
```

Paso por nombre

- Y ahora ejecutamos el siguiente segmento de código:

```
a[1]:=a[2]:=1;
i:=1;
S (a[i],i);
```

Paso por nombre

- El resultado en este caso es el siguiente:

i=2;

a[2]:=0;

Paso por nombre

- Esto es porque lo que se hace es pasar una liga simbólica que une ‘el’ con ‘a[i]’.
- Por tanto, cada vez que se hace una referencia a ‘el’ dentro de la subrutina, el valor de a[i] cambia.
- Asimismo, si se modifica el valor de ‘k’ (ligado simbólicamente a ‘i’), se cambia el valor de ‘a[i]’, ya que la sustitución se hace al regreso de la subrutina.

Paso por nombre

- Cuando se pasa un parámetro por nombre, el parámetro que se transmite del invocador al invocado es realmente sustituido textualmente por el parámetro original en todas sus apariciones en el subprograma.

- Esto es muy diferente de todos los demás mecanismos de paso de parámetros que se usan hoy en día.

Paso por nombre

- Al pasar un parámetro por nombre, se le liga a un método de acceso al momento de la invocación del subprograma.

- Sin embargo, la asociación real a un valor o una dirección de memoria se retrasa hasta que se asigna o se hace referencia al parámetro original del subprograma.

Paso por nombre

- El objetivo de esta asociación tardía de valores es, al menos en teoría, proporcionar una mayor flexibilidad.
- De hecho, pueden lograrse cosas muy interesantes con este mecanismo de paso de parámetros.

Paso por nombre

- Tal vez el ejemplo más famoso de esto sea el propuesto en 1960 por Jørn Jensen, del *Regnecentralen* en Dinamarca, y que desde entonces se conoce como el "dispositivo de Jensen" (*Jensen's device*).

Paso por nombre

- La idea de Jensen fue pasar una expresión y una o más variables que aparecieran en esa expresión como parámetros a un subprograma.

- Cualquier cambio a uno de los parámetros en el subprograma cambiaría los valores de ocurrencias posteriores del parámetro original y, en consecuencia, de la expresión ligada a él.

Paso por nombre

```
real procedure SUM (ADDER, INDEX, LENGTH);
        value LENGTH;
        real ADDER;
        integer INDEX, LENGTH;
        begin
        real TEMPSUM;
        TEMPSUM := 0.0;
        for INDEX := 1 step 1 until LENGTH do
                TEMPSUM := TEMPSUM + ADDER;
        SUM := TEMPSUM
        end;
```

Paso por nombre

- Si ‘A’ es un escalar, entonces la invocación:

SUM (A, I, 100)

simplemente produce el valor 100*A, al sumar A cien veces a una variable inicialmente puesta a cero.

Paso por nombre

- Supongamos ahora que 'A' es un arreglo de 100 reales y usamos:

SUM (A[I], I, 100)

- Ahora, lo que hace SUM es sumar los elementos del arreglo.

Paso por nombre

- Esto se logra porque el ciclo se convierte en:

```
for I:=1 step 1 until 100 do
        TEMPSUM := TEMPSUM + A[I]
```

Paso por nombre

- El mismo programa nos puede producir la suma de los cuadrados de los elementos del arreglo si usamos:

```
SUM (A[I]*A[I], I, 100)
```

Paso por nombre

- El producto punto de los dos vectores contenidos en los arreglos **A** y **B** (cada uno de longitud 100), puede obtenerse con:

```
SUM (A[I]*B[I], I, 100)
```

Paso por nombre

- Estos diferentes usos del procedimiento SUM ilustran el alto grado de flexibilidad que proporciona el paso de parámetros por nombre.

- Con este mecanismo de paso de parámetros, un mismo procedimiento puede ser utilizado para una amplia variedad de propósitos diferentes.

Paso por nombre

- Sin embargo, el precio que debe pagarse por esta flexibilidad reside en 2 áreas: legibilidad y velocidad de ejecución.

- Los procedimientos que usan este mecanismo resultan difíciles de entender y algunas veces no resulta fácil saber siquiera lo que producirá un cierto procedimiento.

Paso por nombre

- Si se pasa un escalar por nombre, este mecanismo es equivalente al paso por referencia.
- Si se pasa una expresión constante por nombre, el mecanismo es equivalente a paso por valor.

Paso por nombre

- Sin embargo, si se pasa un arreglo por nombre, no hay ningún otro mecanismo de paso de parámetros que resulte equivalente a éste.
- Esto se debe a que el valor del subíndice puede cambiar durante la ejecución, entre las diferentes veces que se haga referencia a él.
- Esto hará que diferentes apariciones del parámetro original del subprograma invocado se refieran a diferentes elementos del arreglo.

Paso por nombre

- Si se pasa por nombre una expresión que contenga una variable, tampoco hay ningún otro mecanismo equivalente a éste.

- Esto es porque la expresión es evaluada por cada referencia al parámetro original en el momento en que se haga uso de dicha referencia.

Paso por nombre

```
procedure BIGSUB;
        integer GLOBAL;
        integer array LIST [1:2];
        procedure SUB (PARAM);
                integer PARAM;
                begin
                PARAM := 3;
                GLOBAL := GLOBAL+1;
                PARAM := 5
                end;
        begin
        LIST[1] := 2;
        LIST[2] := 2;
        GLOBAL := 1;
        SUB (LIST[GLOBAL])
        end;
```

Paso por nombre

- Después de correr este programa, obtenemos los valores siguientes:

  LIST[1]:=3
  LIST[2]:=5
  GLOBAL:=2

Paso por nombre

- En términos de costo, el paso por nombre se implementa mediante procedimientos que no toman parámetros o con segmentos de código residentes en memoria en tiempo de ejecución.

- A estos segmentos se les denomina “thunks” (el nombre fue sugerido por P. Z. Ingerman).

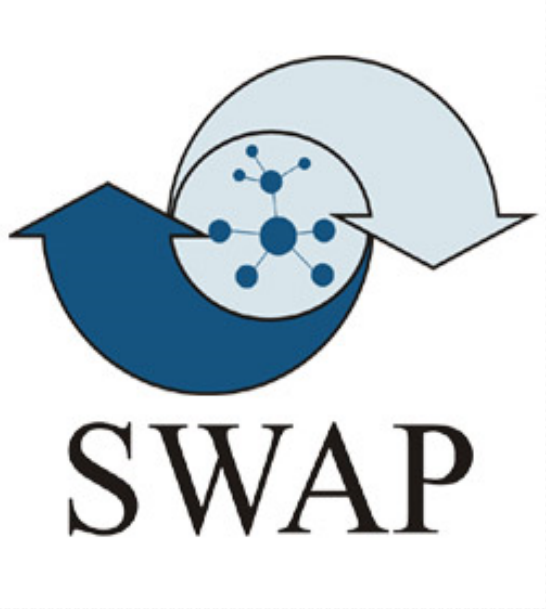
Paso por nombre

- Debe usarse un “thunk” por cada referencia a un parámetro pasado por nombre en el subprograma invocado.
- El “thunk” evalúa la referencia en el ambiente adecuado, o sea en donde se declaró el subprograma (reglas de entorno estáticas), y regresa la dirección del valor pasado como parámetro.

Paso por nombre

- Si el parámetro pasado es una expresión, el código de la referencia debe incluir los datos necesarios para obtener el valor de la celda de memoria cuya dirección fue retornada por el “thunk”.
- Lo anterior deja entrever que este mecanismo de paso de parámetros es más costoso que el simple paso por referencia.
- Esto trae como consecuencia una ejecución más lenta de los programas que lo usan.

Paso por nombre



- Un problema un tanto sorprendente con el paso por nombre es que no puede usarse para implementar una función tan sencilla como es el **SWAP** (intercambiar los valores de 2 variables).

Paso por nombre

```
procedure SWAP (FIRST, SECOND);
        integer FIRST, SECOND;
        begin
        integer TEMP;
        TEMP := FIRST;
        FIRST := SECOND;
        SECOND := TEMP
        end;
```

Paso por nombre

- Esto funciona correctamente si los 2 argumentos que se pasan son escalares, así como en la mayoría de los demás casos.

- Sin embargo, no funciona cuando se pasa el elemento de un arreglo y su subíndice.

Paso por nombre

- Por ejemplo, si hacemos:

SWAP(A[I],I)

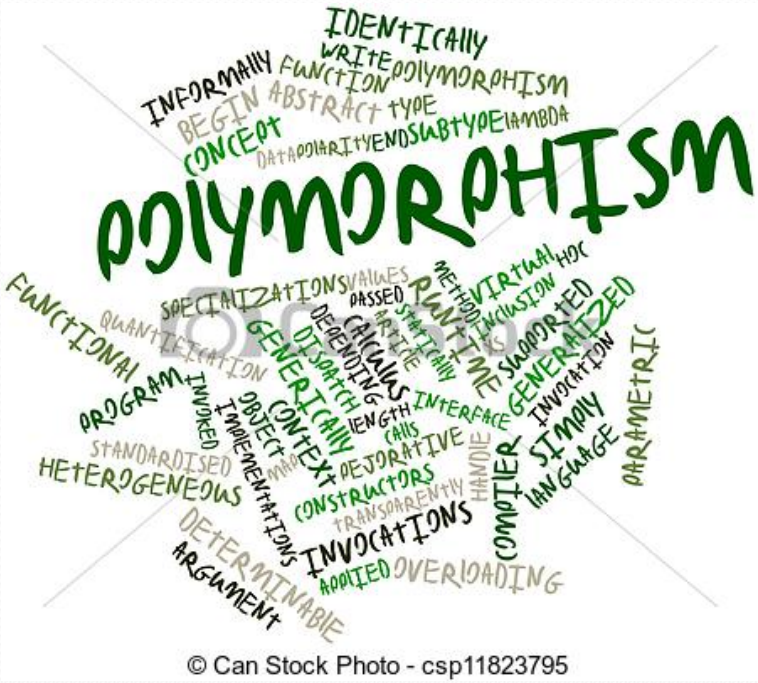
el procedimiento no funciona, ya que el nuevo valor de I será usado para calcular A[I].

Paso por nombre

- Se ha podido demostrar que no hay forma de escribir una función general en ALGOL que realice correctamente el intercambio de valor entre sus 2 parámetros.

- Nótese sin embargo, que el concepto de asociación retrasada en el que se basa el paso de parámetros por nombre no es realmente algo descabellado ni totalmente desacreditado por la comunidad de lenguajes de programación.

Paso por nombre



- El poderoso mecanismo de **polimorfismo**, que es una parte integral de la programación orientada a objetos, es simplemente una asociación retrasada de llamadas a subprogramas o mensajes a objetos.

Paso por nombre

- La **evaluación perezosa** (*lazy evaluation*) es otro mecanismo muy poderoso que no es más que una forma de asociación retrasada.

- Básicamente, consiste en evaluar partes de código funcional solamente cuando resulta evidente que dicha evaluación es absolutamente necesaria.

Aspectos Sintácticos de ALGOL-60

- El formato libre adoptado por ALGOL-60 lo hacía independiente de la máquina donde el lenguaje se implementara.
- Esto estaba en contraposición con el formato fijo adoptado en FORTRAN.

Aspectos Sintácticos de ALGOL-60

- Este tipo de formato libre introdujo un nuevo punto de discusión entre los programadores: cómo indentar el código para hacer que se viera más legible.

- Esto originó la creación de guías de diseño y estilo para la programación.

Aspectos Sintácticos de ALGOL-60

- El comité que diseñó ALGOL, eligió crearlo de tal forma que sería independiente de cualquier conjunto de caracteres en particular, debido a que los americanos no lograron ponerse de acuerdo con los europeos.

- El origen de la disputa que impidió estandarizar el conjunto de caracteres adoptado era algo aparentemente trivial: los americanos deseaban adoptar el punto decimal en los números reales y los europeos querían que se adoptara la coma.

Aspectos Sintácticos de ALGOL-60

- Esta disputa condujo a la definición de 3 niveles del lenguaje: un lenguaje de referencia, un lenguaje para publicaciones y varias representaciones en hardware.

- Estos lenguajes diferían en cuanto a sus convenciones léxicas, pero tenían la misma sintaxis.

Aspectos Sintácticos de ALGOL-60

- El **lenguaje de referencia** es el usado en todos los ejemplos contenidos en el reporte de ALGOL.

- Por ejemplo:

$$\mathbf{a[i+1] := (a[i] + pi \times r \uparrow 2)/6.02_{10}23;}$$

Aspectos Sintácticos de ALGOL-60

- El **lenguaje para publicaciones** estaba destinado a ser usado para publicar algoritmos implementados en ALGOL.

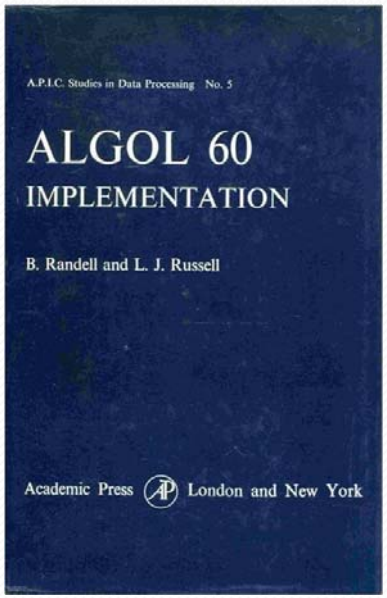
- Se distinguía por permitir varias convenciones léxicas y de impresión (p.ej., letras griegas y subíndices) que facilitarían su legibilidad.

Aspectos Sintácticos de ALGOL-60

- Ejemplo:

$$a_{i+1} \leftarrow \{a_i + \pi \times r^2\} / 6.02 \times 10^{23}$$

Aspectos Sintácticos de ALGOL-60



- Finalmente, se dejó a los implementadores del lenguaje la tarea de definir las **representaciones de ALGOL en hardware** que resultaran más apropiadas para los conjuntos de caracteres y dispositivos de entrada/salida de sus propios sistemas de cómputo.

Aspectos Sintácticos de ALGOL-60

- Ejemplo:

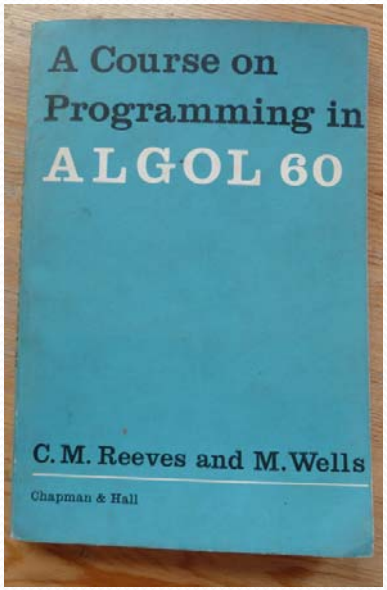
a[i+1] := (a[i] + pi*r^2)/6.02E23;

Aspectos Sintácticos de ALGOL-60

- ALGOL no tenía por tanto el problema de FORTRAN de confundir palabras clave con identificadores.

- Esto se debía a que las palabras clave del lenguaje se identificaban de manera única (por ejemplo, usando caracteres en **negritas**), y por tanto no tenían relación o similitud con respecto a identificadores con nombre parecido.

Aspectos Sintácticos de ALGOL-60



- ALGOL también se adhiere de forma más cercana que FORTRAN al **Principio Cero-Uno-Infinito**.

- También es más elegante y está mejor estructurado.

Aspectos Sintácticos de ALGOL-60

- ALGOL ha sido una de las influencias más importantes a los lenguajes de programación modernos de todo el mundo. Muchas de sus convenciones sintácticas, sus estructuras, su regularidad e incluso su notación, han sido adoptadas por lenguajes estructurados modernos tales como C, Modula-2, Ada, etc.