Lenguajes de Programación

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

Sentencia para Casos

- Es muy común al escribir programas, el descomponer un problema en dos o más subproblemas. Esto es, un problema se suele dividir en casos que se manejan de manera distinta.

- Por ejemplo, en el registro variante que vimos antes para los aviones, podríamos dividir un vuelo en tres casos, de acuerdo a su estatus: “inAir”, “onGround” o “atTerminal”. Esto se maneja con la sentencia **case** de Pascal.

Sentencia para Casos

- Otros lenguajes tienen mecanismos similares para manejar estos casos. Vimos, por ejemplo, el caso de FORTRAN, que usa el GOTO calculado. Por ejemplo:

```
      GOTO (100, 250, 250, 400), I
100 . . . S1 . . .
      GOTO 500
250 . . . S23 . . .
      GOTO 500
400 . . . S4 . . .
500 . . . resto del programa . . .
```

En este caso, si I=1, se ejecuta la sentencia S_1, si I=2 o 3, se ejecuta la sentencia S_{23} y si I=4, se ejecuta la sentencia S_4.

Sentencia para Casos

- Esto es muy eficiente porque el GOTO calculado se compila como una tabla de saltos. Sin embargo, esta sintaxis no es fácil de interpretar y el flujo de control no resulta del todo obvio. Esto viola el **Principio Estructural**.

Principio Estructural

La estructura estática del programa debiera corresponder, de manera simple, con la estructura dinámica de los cálculos correspondientes.

Sentencia para Casos

- Varios lenguajes de programación han intentado proporcionar alternativas más estructuradas al GOTO calculado de FORTRAN y al **switch** de Algol, que vimos anteriormente.

- Estas estructuras se suelen basar en el **if-then-else**, que descompone un problema en dos casos, dependiendo de la condición. A este tipo de sentencias se les denomina de casos (**case**) y usualmente lucen como se muestra en el acetato siguiente.

Sentencia para Casos

case <expresión> **of**

<sentencia>,

<sentencia>,

⋮

<sentencia>

end case;

Sentencia para Casos

- Esto se interpreta de la manera siguiente: Si el valor de la <expresión> es 1, entonces se ejecuta la primera <sentencia>; si es 2, se ejecuta la segunda <sentencia> y así sucesivamente.

- Al igual que la sentencia **if-then-else**, la sentencia **case** tiene una sola entrada desde la sentencia anterior y una sola salida hacia la sentencia siguiente.

Sentencia para Casos

- Nótese, sin embargo, que este constructor tiene algunos problemas.

- El primero de ellos se relaciona con la escalabilidad. Si tenemos muchos casos (p.ej., tener 50 o 100 casos no es inusual en programas comerciales), resultará difícil para el usuario entender qué sentencias corresponden a cada caso.

- El uso de comentarios puede ayudar, pero a veces los programadores realizan cambios al código sin preocuparse en cambiar los comentarios, lo cual puede conducir a confusiones posteriores.

Sentencia para Casos

- Otro problema es que resulta difícil manejar dos o más casos con el mismo código. Por ejemplo, si quisiéramos usar la sentencia **case** para implementar los casos que vimos antes con el GOTO calculado, tendríamos:

case I **of**
begin . . . S_1 . . . **end**,
begin . . . S_{23} . . . **end**,
begin . . . S_{23} . . . **end**,
begin . . . S_4 . . . **end**
end case

Sentencia para Casos

- Nótese que tuvimos que repetir las sentencias S_{23} para los casos 2 y 3. Esto viola el **Principio de Abstracción** y hace al programa más difícil de leer, más difícil de escribir y más difícil de mantener.

- Como no resulta obvio que el código para los casos 2 y 3 es el mismo, podría ocurrir que se corrige un error en uno de estos dos casos, pero no en el otro.

- Esto podría resolverse factorizando el código de S_{23} y colocándolo en un procedimiento que se invocaría en los casos 2 y 3. Sin embargo, esto complicará el código.

Sentencia para Casos

- Una solución a estos problemas la proporciona la sentencia **case** con etiquetas, que tiene Pascal y que fue propuesta originalmente por C.A.R. Hoare.

Sentencia para Casos

- La sintaxis de la sentencia **case** en Pascal es la siguiente:

case <expresión> **of**

<etiqueta(s)> : **begin** . . . <sentencia> . . . **end**;
<etiqueta(s)> : **begin** . . . <sentencia> . . . **end**;
.
.
.
<etiqueta(s)> : **begin** . . . <sentencia> . . . **end**
end

Sentencia para Casos

- Esta sentencia proporciona un mecanismo seguro que está auto-documentado (sigue el **Principio de Etiquetamiento**).

Principio de Etiquetamiento

Evite secuencias arbitrarias que tengan más de unos pocos elementos de longitud; no requiera que el usuario sepa la posición absoluta de un elemento en una lista. En vez de eso, asocie una etiqueta representativa con cada elemento y permita que los elementos se coloquen en cualquier orden.

Sentencia para Casos

- La sentencia **case** de Pascal no depende del programador el documentar adecuadamente cada uno de los casos posibles, pues eso se hace con las etiquetas. El ejemplo anterior quedaría de la forma siguiente:

case I **of**

1: **begin** . . . S_1 . . . **end**;

2,3: **begin** . . . S_{23} . . . **end**;

4: **begin** . . . S_4 . . . **end**;

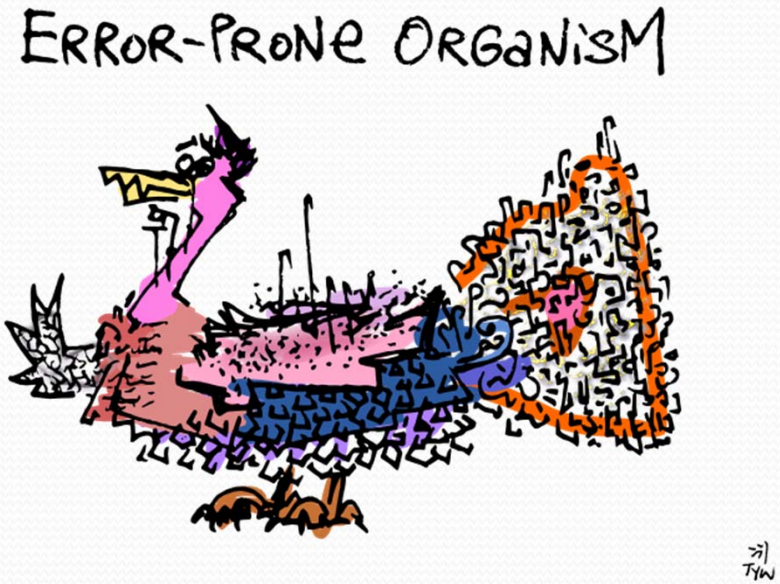
end

Sentencia para Casos

- La sentencia **case** de Pascal también es más flexible que su contraparte sin etiquetas. Por ejemplo, permite el uso de etiquetas alfanuméricas para denominar los distintos casos. Por ejemplo:

```
case NextFlight.status of
        inAir:        . . . caso para vuelos en el aire . . .
        onGround:     . . . caso para vuelos en tierra . . .
        atTerminal:   . . . caso para vuelos en terminal . . .
end
```

Sentencia para Casos



- Esto evita al programador el procedimiento (propenso a errores) de tener que mapear manualmente estos casos a valores enteros y documenta cada uno de los casos proporcionados.

Sentencia para Casos

- La sentencia **case** de Pascal también es muy eficiente.
- Puesto que el compilador puede determina el tipo de la expresión usada en la sentencia **case**, sabe los valores posibles que tendrá esta expresión. Por tanto, puede usar esta información para construir una tabla de saltos del tamaño apropiado.
- Algunos compiladores incluso van más lejos y generan una tabla de saltos, una tabla hash, una búsqueda binaria de una tabla ordenada, una búsqueda secuencial de una tabla lineal o una serie de **if**s, dependiendo del número de valores, sus rangos y su agrupamiento.

Mecanismos de paso de parámetros

- Pascal eliminó el paso de parámetros por nombre y sólo permite el **paso por valor** y el **paso por referencia** (que reemplaza al paso por nombre de Algol).

- El propósito del **paso por referencia** es permitir que un procedimiento modifique los parámetros que se le pasan. En otras palabras, los parámetros los procedimientos de referencia son de salida (o de entrada-salida).

Mecanismos de paso de parámetros

- El problema del paso por referencia en FORTRAN era que un procedimiento podía realizar alternaciones de parámetros que no tenían ningún sentido (por ejemplo, podían alterar constantes).

- Pascal resuelve este problema, porque el programador debe especificar en la declaración de un procedimiento si un parámetro se pasará por valor o por referencia.

Mecanismos de paso de parámetros

- Si un parámetro se pasa por referencia, entonces el compilador se asegura de que el valor donde se almacenará lo que regrese del procedimiento es algo que tiene sentido (o sea, una variable, el elemento de un arreglo o el campo de un registro).

- Por tanto, este mecanismo es *seguro*.

Mecanismos de paso de parámetros

- Los parámetros pasados por referencia también son *eficientes*, porque sólo se pasa la dirección de memoria a los mismos, sin importar su tamaño.

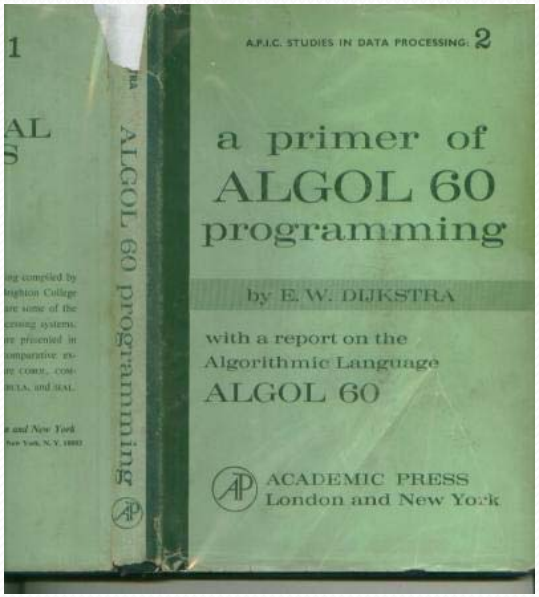
- En contraste, los parámetros pasados por nombre son costosos, porque debe invocarse una liga simbólica (*thunk*) cada vez que se usa el parámetro.

Mecanismos de paso de parámetros

- El **paso por valor** en Pascal es exactamente igual que en Algol: se pasa una copia del parámetro, el cual se usa como una variable local en el procedimiento correspondiente.

- Este mecanismo se usa para parámetros de entrada, puesto que previene la modificación del valor que se pasa al procedimiento.

Mecanismos de paso de parámetros



- Este mecanismo tiene los mismos problemas de eficiencia que el paso por valor de Algol, puesto que si se pasa un arreglo o cualquier otra estructura de datos grande, ésta tendrá que ser copiada.

Mecanismos de paso de parámetros

- El Reporte de Pascal original no especificaba que los únicos mecanismos de paso de parámetros eran el **paso por valor** y el **paso por referencia**.

- El reporte decía que existían dos mecanismos de paso de parámetros: el **paso por referencia** y el **paso por constante**.

Mecanismos de paso de parámetros

- Un parámetro pasado por **constante** es similar a uno pasado por valor, pues ambos se usan como parámetros de entrada.

- Sin embargo, el uso de **paso por constante** es más eficiente que el **paso por valor**, como veremos a continuación.

Mecanismos de paso de parámetros

- Un parámetro pasado por **constante** se considera constante en el cuerpo del procedimiento a donde se pasa. Por tanto, no es legal usar el parámetro como el destino de una asignación.

- También es ilegal usar este valor en cualquier otro contexto donde se use como destino. En otras palabras, el compilador protege la seguridad de los parámetros pasados por constante.

Mecanismos de paso de parámetros

- Puesto que el compilador impide que un parámetro pasado por constante se altere, el compilador puede decidir si pasar el valor del parámetro o su dirección de memoria (lo que resulte más eficiente).

- Por ejemplo, para valores pequeños, tales como enteros, caracteres y elementos de una enumeración, el compilador puede copiar el valor del parámetro directamente (o sea, lo pasaría por valor).

Mecanismos de paso de parámetros

- Para valores grandes, como un arreglo o un registro de gran tamaño, el compilador sólo pasará la dirección de memoria del parámetro.
- Esto ahorra tiempo y el espacio extra que se requiere al copiar el parámetro usando el paso por valor.
- Tampoco hay riesgo de que el parámetro se altere, porque el compilador lo impide.

Mecanismos de paso de parámetros

- Como puede verse, el paso de parámetros por constante tiene la seguridad del paso por valor y la eficiencia del paso por referencia. La pregunta es entonces: ¿por qué se eliminó este mecanismo de paso de parámetros de Pascal?

- Una posibilidad es que este mecanismo tiene algunos problemas de seguridad, pues permite ciertas formas limitadas de *aliasing*. Por ejemplo, la misma variable puede ser visible de 2 maneras distintas: como no local y como parámetro.

Mecanismos de paso de parámetros

- Consideremos el siguiente segmento de código:

```
type vector = array [1..100] of real;
var A: vector;
      procedure P (x: vector);
      begin
         writeln (x[1]);
         A[1] := 0;
         writeln (x[1]);
       end;

begin
        P(A)
end
```

Mecanismos de paso de parámetros

- En este caso, dentro del procedimiento P, se han usado dos nombres distintos para la misma área de almacenamiento: ‘x’ y ‘A’.

- Puesto que ‘x’ es un parámetro pasado por constante, el compilador elegirá pasarlo por referencia (es un arreglo), lo que quiere decir que se pasará la dirección de ‘A’ a P. Como la asignación que aparece dentro de P modifica el contenido del arreglo, las 2 referencias que se incluyen a x[1] pueden producir valores diferentes, pese a que x[1] no se modificó en ningún momento.

Mecanismos de paso de parámetros

- Desafortunadamente, la alternativa elegida en Pascal (permitir sólo el paso por valor y el paso por referencia), puede motivar a los programadores a pasar algo por referencia simplemente porque es más eficiente (en términos de uso de memoria).

- Esto pone en peligro la seguridad de los programas y causa confusión entre otras personas que lean el código correspondiente.

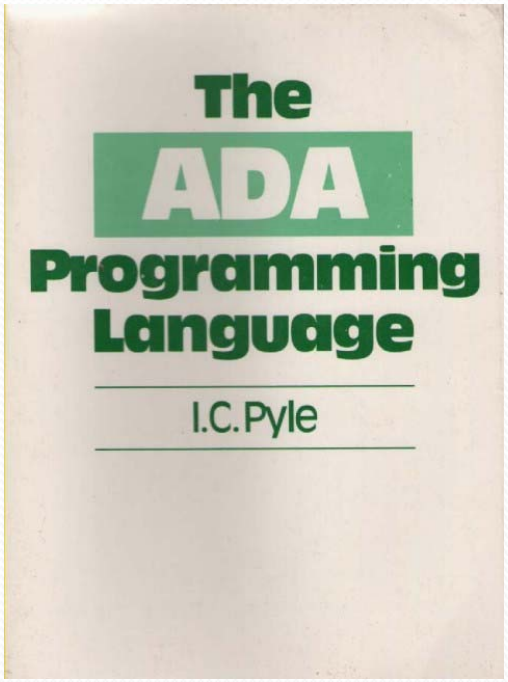
Mecanismos de paso de parámetros

- La raíz de este problema es que se confunden dos cuestiones que son ortogonales:

  1) La decisión de si un parámetro debe ser usado para entrada o para salida.

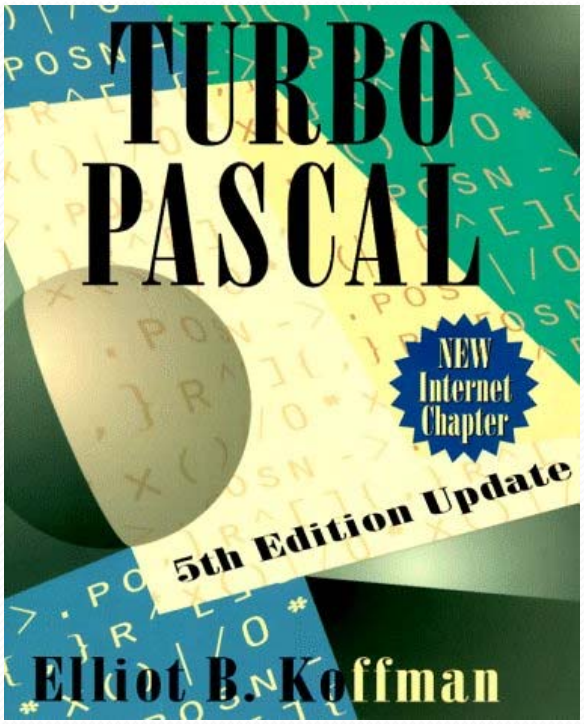
  2) La decisión de si debe copiarse su valor o pasarse la dirección de su valor.

Mecanismos de paso de parámetros



- Lenguajes modernos como Ada han separado mejor estas decisiones regresando a un mecanismo de paso de parámetros similar al **paso por constante**.

Parámetros Procedurales



- Para restaurar algo de la flexibilidad perdida al omitir el paso de parámetros por nombre, Pascal permite que se pasen procedimientos y funciones como argumentos a otros procedimientos y funciones.

Parámetros Procedurales

- Por ejemplo, supongamos que necesitamos definir la función 'difsq (f, x)', que calcula $f(x^2) - f(-x^2)$para cualquier función real f y un número real x. O sea:

$$difsq\ (sin, \theta) = \ \sin(\theta^2) - \sin(-\theta^2)$$

Según el Reporte Revisado de Pascal, debemos usar:

```
procedure difsq (function f: real; x: real): real;
begin
    difsq := f(x*x) – f(-x*x)
end;
```

Parámetros Procedurales

- Este código luce bastante simple. Sin embargo, hay un problema: el método que usa Pascal para especificar que una función se pasará como parámetro introduce un hueco muy importante en la seguridad del lenguaje.

- Noten que en la definición del procedimiento 'difsq' sólo se especifica que 'f' es una función **real**. Sin embargo, no se dice nada sobre los parámetros de 'f' (ni cantidad, ni tipos).

Parámetros Procedurales

- Esto hace casi imposible para el compilador el poder determinar si una invocación particular de ‘difsq’ es legal o no (recordemos que el mismo procedimiento ‘difsq’ se podría pasar como parámetro a otro procedimiento).

- Esto motivó que el comité que estandarizó Pascal modificara este mecanismo, de tal forma que se tengan que especificar los argumentos de un procedimiento que se pasa como parámetro.

Parámetros Procedurales

- Para el ejemplo anterior, se debe usar:

procedure difsq (**function** f(y: **real**): **real**; x: **real**): **real**;
  begin . . . . **end**;

Esto permite al compilador el poder efectuar un chequeo completo de tipos, lo cual preserva los tipos fuertes de Pascal.

La implementación correcta de 'difsq' se mostrará en clase.

Parámetros Procedurales

- Existen varios costos asociados con la solución que acabamos de ver. Un costo obvio es el incremento que se obtiene en la complejidad del lenguaje.

- Un costo más sutil es la implicación de que los procedimientos son ciudadanos de primera clase.

- Puesto que esta solución permite que los procedimientos se pasen como parámetros, el usuario podría pensar que también se pueden usar en las declaraciones de variables.

Parámetros Procedurales

- Si se permitiera que los procedimientos fueran ciudadanos de primera clase, el lenguaje tendría que definir el significado de usar variables procedurales, arreglos procedurales, archivos procedurales, etc.

- Existen diversos aspectos complejos relacionados con el diseño de un lenguaje de programación y su implementación que se relacionan con estos conceptos.

- Por otro lado, si el lenguaje no permite todas estas opciones, eso significa que los procedimientos son ciudadanos de segunda clase, lo cual los convierte en excepciones, las cuales ya dijimos que incrementan también la complejidad del lenguaje.

Recapitulando

- Recordemos que el objetivo primordial de Pascal era enseñar a programar.
- Esto fue lo que condujo a que su diseño enfatizara aspectos tales como la eficiencia, simplicidad y confiabilidad.
- Pese a sus problemas de diseño, Pascal fue un lenguaje de programación muy exitoso, que se usó para enseñar a programar a varias generaciones de estudiantes universitarios.

Implementación de los Lenguajes Estructurados

- Puesto que la noción de FORTRAN de un **registro de activación** no es adecuada para lenguajes estructurados tales como Algol y Pascal, debemos reanalizar dichos registros de activación, considerando sus propósitos.

- Los **registros de activación**, como su nombre lo indica, llevan el registro del estado de activación de un procedimiento.

Implementación de los Lenguajes Estructurados

- Para conocer el estado de activación de un procedimiento, necesitamos la siguiente información:

  1) El código o algoritmo, que forma el cuerpo del procedimiento.

  2) El lugar en ese código donde se está ejecutando esta activación del procedimiento.

  3) Los valores de todas las variables visibles a esta activación.

Implementación de los Lenguajes Estructurados

- Dado que el código o algoritmo no cambia durante la ejecución del procedimiento, no se tiene que almacenar en el registro de activación.

- Los otros dos elementos antes mencionados pueden variar durante la ejecución de un programa, por lo cual deben ser parte del registro de activación.