Lenguajes de Programación

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

Estructuras de Diseño y Sistema de Tipos

- Las reglas aritméticas para los tipos de punto fijo son más complicadas que las de los enteros y los números de punto flotante.

- Esto es particularmente cierto para la multiplicación y la división.

Estructuras de Diseño y Sistema de Tipos

- Por ejemplo:
- Si VF es un tipo de punto fijo F y VI es un entero, entonces:

VF:=VF*VI; o VF:=VI*VF; son operaciones legales, porque ambas son de tipo F

Estructuras de Diseño y Sistema de Tipos

- Sin embargo, si VG es una variable de punto fijo de cualquier tipo (incluyendo F), entonces:

VF:=VF*VG;

es ilegal, porque `VF*VG´ se considera de tipo `universal fijo´, el cual es un número de punto fijo de precisión máxima.

Estructuras de Diseño y Sistema de Tipos

- Para que la asignación que mostramos anteriormente sea legal, se requiere de una conversión explícita de tipo:

```
VF:=F(VF*VG);
```

Estructuras de Diseño y Sistema de Tipos

- La división sigue reglas similares (en cuanto a lo poco intuitivas), pero éstas son realmente una consecuencia prácticamente inevitable de la aritmética de punto fijo.

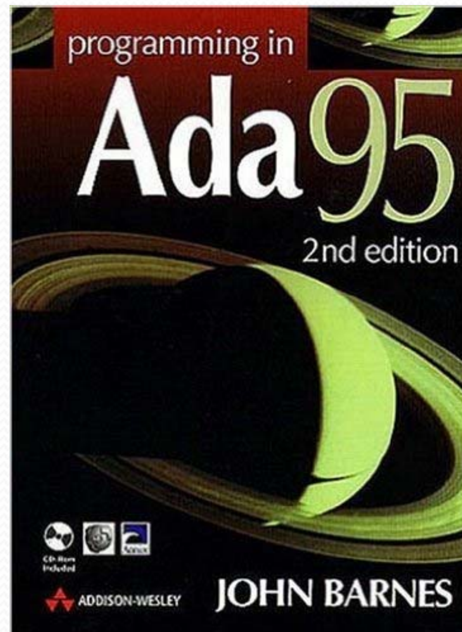
Constructores basados en los de Pascal

- Los constructores de Pascal se adoptan también en Ada.
- Esto incluye enumeraciones, arreglos, registros y apuntadores (a los que se les llama “accesos”).
- Sin embargo, Ada intenta resolver los problemas asociados con algunos de ellos en Pascal.

Constructores basados en los de Pascal

- Por ejemplo, vimos antes el problema con los registros “variantes” en Pascal.
- El problema surgía al cambiar el discriminante de un registro variante sin inicializar los campos correspondientes.

Constructores basados en los de Pascal



- Ada resuelve este problema estableciendo que el discriminante puede cambiarse solamente mediante la asignación de un valor completo a un registro (o sea, asignando un valor a todos los campos de un registro).

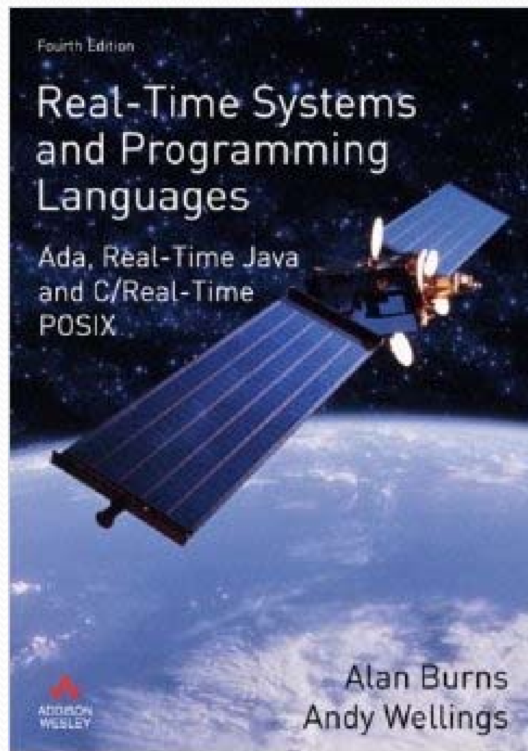
Constructores basados en los de Pascal

- En todas las demás situaciones, el discriminante es tratado como una constante.
- Esto asegura que los campos que corresponden al valor del discriminante estén siempre inicializados.

Equivalencia de Nombres

- El sistema de tipos de Ada es más fuerte que el de Pascal porque impone un uso más consistente de la equivalencia de nombres.
- Ada usa una forma muy pura de la equivalencia de nombres, pero agrega los conceptos de subtipo y tipo derivado, a fin de hacer al sistema de tipos más conveniente y fácil de usar.

Equivalencia de Nombres



- Recordemos que bajo la equivalencia de nombres, dos objetos se consideran del mismo tipo si sus tipos tienen el mismo nombre.

Equivalencia de Nombres

- Por ejemplo:

type PERSONA **is record** ID, EDAD: **INTEGER; end record**;
type COCHE **is record** ID, EDAD: **INTEGER; end record**;

X:PERSONA;
Y:COCHE;

Equivalencia de Nombres

- Las variables X & Y tienen tipos diferentes bajo equivalencia de nombres, porque sus tipos tienen nombres diferentes (PERSONA y COCHE).

Equivalencia de Nombres

- Ada lleva este principio todavía más lejos, pues dice que incluso si:

  X: **record** ID, EDAD: INTEGER; **end record**;
  Y: **record** ID, EDAD: INTEGER; **end record**;

- X & Y se siguen considerando diferentes.

Equivalencia de Nombres

- El razonamiento detrás de esta decisión es que si el programador está definiendo dos veces el mismo tipo es porque tiene la intención de usarlo para cosas diferentes.

- El compilador, por lo tanto, asignará etiquetas anónimas a cada declaración de tipo y por ende los tipos reales de X & Y serán internamente diferentes.

Equivalencia de Nombres

- La contraparte de equivalencia de nombres (la equivalencia estructural) es normalmente más difícil de implementar .
- También es difícil obtener un consenso respecto a cómo definir adecuadamente la equivalencia estructural, ya que hay demasiadas situaciones distintas en las que puede no resultar claro cómo proceder.

Equivalencia de Nombres

Ada

The Language For A Complex World

- Una de las razones por las que la equivalencia de nombres no ha sido adoptada por otros lenguajes de programación es que suele resultar demasiado restrictiva.

Subtipos

N: INTEGER;
type INDICE **is range** 1..100;
I: INDICE;

- Puesto que INDICE e INTEGER tienen nombres diferentes, la equivalencia de nombres pura los considerará como tipos diferentes.

Subtipos

- Por lo tanto, sería ilegal hacer “N:=I”, lo cual resulta bastante extraño, considerando que el rango de INDICE es un subconjunto de los enteros.

- Las cosas empeoran si consideramos que no es posible escribir algo como “I+1”, porque la suma está definida sólo para enteros y no para el tipo INDICE.

- Este es un problema serio.

Subtipos

- Ada resuelve esta situación estableciendo que un rango tal como 1..100 define un subtipo de algún tipo base.

- De tal forma, I+1 es una operación legal porque este rango es un subtipo de INTEGER.

Subtipos

- Los subtipos también son compatibles con el tipo base de otros subtipos del mismo tipo base.
- De tal forma, asignaciones tales como “N:=I” e “I:=N” son legales (estas equivalencias se checan en tiempo de ejecución).

Subtipos

- Para complementar la definición implícita de subtipo, hay también una forma de definir subtipos en Ada:

subtype INDICE **is INTEGER range** 1..100;

- INDICE en este caso es un subtipo de INTEGER y, por lo tanto, hereda todas las operaciones y propiedades del tipo base.

Subtipos

- Es interesante advertir que INTEGER debe especificarse en este caso (declaración del subtipo), pero no puede especificarse en la declaración de tipo.
- Esto es una irregularidad de la sintaxis de Ada que el programador debe recordar. No resulta claro por qué podríamos necesitar dos formas diferentes de hacer lo mismo en el lenguaje.

- Ada proporciona otro mecanismo de declaración de tipos, llamado “tipos derivados”. Por ejemplo:

type PORCENTAJE **is new INTEGER range** 0. . 100;

Tipos Derivados

- Esto define un nuevo tipo llamado PORCENTAJE, el cual es diferente de INTEGER, INDICE y cualquier otro tipo.

- En particular, no es posible asignar una variable de tipo PORCENTAJE a una variable de tipo INDICE o viceversa.

Tipos Derivados

- Esto tiene sentido, ya que no queremos usar estas dos variables de manera intercambiable ya que significan cosas diferentes.

- Lo que hace diferente a un tipo derivado es que hereda todas las operaciones, funciones y demás atributos (internos o definidos por el usuario) del tipo del cual se derivó.

Tipos Derivados

- Esto permite al usuario definir un nuevo tipo que es diferente (desde un punto de vista abstracto) del tipo del cual se derivó.

- Sin embargo, a la vez permite seguir usando todas las operaciones definidas previamente para el tipo original.

Tipos Derivados

- De hecho, es posible convertir explícitamente entre tipos derivados y sus tipos padre.
- Por ejemplo, PORCENTAJE(N), convierte un valor entero N a un valor de tipo PORCENTAJE.

Restricciones

- El constructor de subrangos de Pascal ha sido reemplazado en Ada por un mecanismo más general: la *restricción (constraint)*.

- La restricción es un mecanismo que restringe el conjunto permisible de datos de un tipo, sin restringir las operaciones aplicables a dichos datos.

Restricciones

- En otras palabras, una restricción define un subtipo de un cierto tipo base. Un ejemplo son las restricciones de rango:

INTEGER **range** 1. .100;

CHARACTER **range** 'A'. .'Z';

Restricciones

- Las restricciones tienen los mismos costos y beneficios que los subrangos en Pascal, pero son más generales porque permiten el uso de expresiones que pueden ser evaluadas en tiempo de ejecución.
- En estos casos, se requiere efectuar cierto chequeo en tiempo de ejecución que, de otra manera, se efectuaría en tiempo de compilación.

Restricciones

- Otro ejemplo de restricciones son las de "precisión":

FLOAT **digits** 10 **range** -1e6. . 1e6;
DOLLARS **delta** 1 **range** 1..10;

- Un tercer tipo de restricción son las "discriminantes".

Restricciones

- Si PERSONA es un registro variante cuyo discriminante puede tomar dos valores posibles (MASCULINO, FEMENINO), entonces PERSONA(MASCULINO) es un ejemplo de restricción discriminante.

- Este mecanismo restringe el conjunto de valores posibles que puede adoptar PERSONA a sólo dos (MASCULINO y FEMENINO).

Restricciones

- El cuarto tipo de restricción es el índice. Para escribir una rutina de propósito general que sume los elementos de un arreglo de números reales, podemos definir el tipo VECTOR:

type VECTOR **is array** (INTEGER **range** <>) of FLOAT;

Restricciones

- Esta declaración significa que cada objeto de tipo VECTOR es un arreglo de FLOATs cuyo índice es algún subrango de INTEGER.

- Por lo tanto, para declarar un objeto de tipo VECTOR, este rango debe ser especificado:

VALS: VECTOR(1..100);

DIAS: VECTOR(1..366);

Restricciones

- Ada evita los problemas de Pascal en el indizado de arreglos al permitir que los programadores usen un tipo no restringido en la especificación del parámetro de un procedimiento:

function SUMA (V: VECTOR) **return** FLOAT **is** . . .

- El sistema de tipos de Ada permitirá que tanto VALS como DIAS se puedan pasar como argumento a SUMA, porque se considera a ambos como del mismo tipo (no definido).

- Dentro de SUMA es posible acceder estos parámetros ocultos usando V'FIRST y V'LAST.

Restricciones

Por tanto, el ciclo que se requiere para sumar los elementos del arreglo es el siguiente:

```
for I in V'FIRST . . V'LAST loop
    TOTAL:=TOTAL+V(I);
end loop;
```

- También es posible declarar variables de tipo VECTOR, en cuyo caso los límites correspondientes a VECTOR tienen que almacenarse junto con su contenido.

Enumeraciones

- Las siguientes enumeraciones serían consideradas ilegales en Pascal debido a que no se permiten traslapes:

```
type PRIMARIO is (ROJO, AZUL, VERDE);
type SEMAFORO is (ROJO, AMARILLO, VERDE);
```

- Sin embargo, en Ada, estas enumeraciones son perfectamente válidas.

Enumeraciones

- Esto pareciera introducir ambigüedades en nuestro programa, puesto que ROJO ahora tiene dos significados diferentes.
- Ada resuelve este problema asociando ROJO con la enumeración correspondiente a la declaración de tipo realizada.
- Por ejemplo, si C se declara como de tipo PRIMARIO, entonces C:=ROJO deja de ser ambiguo.

Enumeraciones

- Puesto que hay unas cuantas situaciones en las cuales podría originarse una ambigüedad (cuando se usan procedimientos sobrecargados), y el tipo no puede ser inferido del contexto, entonces el compilador requerirá en ciertos casos que el programador especifique a qué declaración de tipo se está refiriendo.

Enumeraciones

- Ejemplo:
- PRIMARIO(ROJO) o SEMAFORO(ROJO)
- Una de las razones para permitir enumeraciones sobrecargadas es que podemos preservar el símil con los lenguajes humanos en los que tales ambigüedades suelen utilizarse.

Enumeraciones

- En los lenguajes humanos una misma palabra tiene varios significados (o sea, está sobrecargada) dependiendo del contexto en la cual se use (p.ej., traje, del verbo traer y traje, la vestimenta).
- Otra razón es que Ada considera a los conjuntos de caracteres como enumeraciones.

Enumeraciones

- Por ejemplo:

type CHARSET **is** (‘A’, ‘B’, . . . ,’9’, ‘+’, ‘-’, ‘;”,.’);

- Puesto que el mismo caracter (p.ej., ‘A’ o ‘B’) puede aparecer en varios conjuntos diferentes de caracteres, la sobrecarga de las enumeraciones se asume de manera implícita en Ada.

Enumeraciones

- ¿Están de acuerdo con esta solución?
- ¿Que solución alternativa propondrían a este problema?

Estructuras Denominativas

- Las estructuras denominativas principales de Ada están basadas en las de Pascal: constante, variable, tipo, procedimiento y función.
- Sin embargo, casi todas ellas han sido mejoradas en Ada.
- También hay dos tipos nuevos de declaración: tarea y paquete.

Estructuras Denominativas

- Las variables permiten inicialización (a diferencia de Pascal):

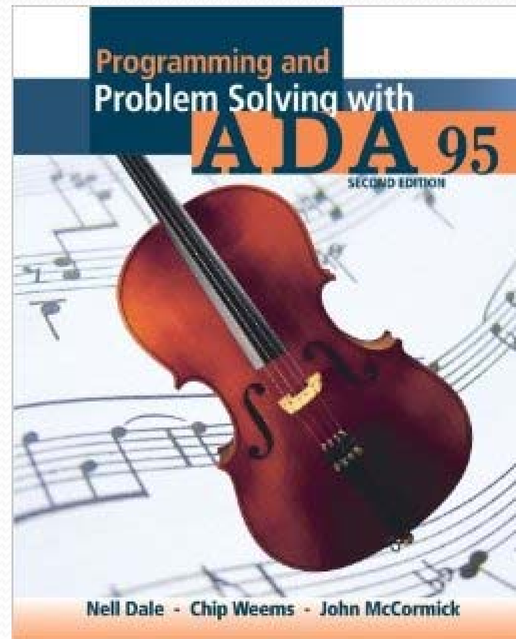
APROXIMACION: FLOAT:=3.5;

- Esto hace que los programas sean más legibles y evita los problemas clásicos que se presentan al no inicializar variables.

Estructuras Denominativas

- El valor inicial que se asigna a una variable en Ada no tiene que ser una constante.
- Ada también permite el uso de expresiones de cualquier complejidad.
- Dichas expresiones se evalúan cuando se entra al bloque o procedimiento en el cual ocurren.

Estructuras Denominativas



- Esto es una ventaja, ya que permite que el mismo constructor pueda usarse para todo tipo de inicialización, independientemente de si el valor es o no una constante.

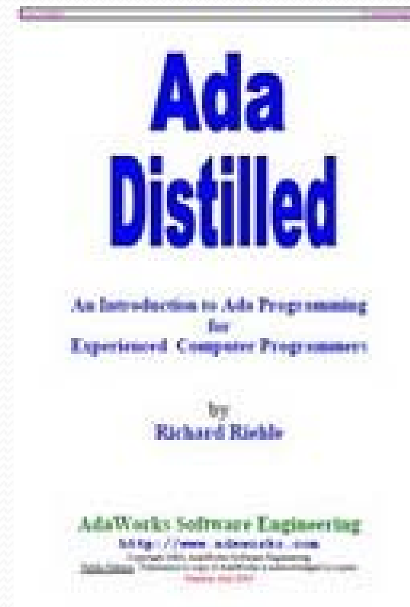
Estructuras Denominativas

- La declaración de constantes es una forma modificada de la declaración de variables:

PIES_POR_MILLA: **constant** INTEGER:=5280;

PI: **constant** := 3.14159_2653589;

Estructuras Denominativas



- Una declaración de constante es interpretada exactamente igual que la declaración de una variable, excepto por el hecho de que el valor no puede cambiarse en este caso después de haberse inicializado al entrar a un cierto entorno.

Estructuras Denominativas

- Por lo tanto, este tipo de declaración es más general que la de Pascal, ya que puede calcularse su valor durante la ejecución de un programa y puede diferir en distintas instancias del entorno.
- Esto es útil y ayuda al mantenimiento del programa.

Estructuras Denominativas

- En Ada también es posible declarar constantes sin proporcionar su tipo:

- PI: **constant**:=3.14159_2653589;

- El tipo puede omitirse si se trata de un número.

Estructuras Denominativas

- Asimismo, puede omitirse el tipo si la expresión del lado derecho “involucra sólo literales, nombres de literales numéricas, llamadas de la función predefinida ABS, literales entre paréntesis, y los operadores aritméticos predefinidos”.

Estructuras Denominativas

- La razón para este mecanismo un tanto inusual es permitir la definición de constantes de tipo "entera universal" y de tipo "real universal".

- Estos tipos tienen la precisión máxima posible y no son normalmente accesibles a los programadores.

Estructuras Denominativas

- El beneficio de este mecanismo es permitir al programador nombrar una constante numérica que sea independiente del tipo y de la precisión.
- El costo de este mecanismo es que debe recordarse la regla antes mencionada, la cual puede resultar poco intuitiva para el programador.

Especificaciones y Definiciones

- Ada soporta el ocultamiento de información dividendo las declaraciones en dos partes:

    1. Una que define la interfaz.
    2. Una que proporciona una implementación.

La mayor parte de las declaraciones en Ada pueden dividirse en estas dos partes.

Especificaciones y Definiciones

- Por ejemplo, para definir una constante, podemos usar:

TAM_MAXIMO: **constant** INTEGER;

Esto especifica que TAM_MAXIMO es el nombre de una constante de tipo INTEGER, pero no define su valor.

Especificaciones y Definiciones

- Una constante "diferida" de este tipo normalmente se usaría como parte de la especificación de un paquete.

- De esta forma, un paquete puede proporcionar una constante sin definir su valor como parte de su interfaz. Esta constante puede ser "implementada", esto es, se le puede dar un valor, usando una definición convencional:

TAM_MAXIMO: **constant** INTEGER := 256;

- Esta definición, normalmente aparecería en la parte privada de un paquete.

Subprogramas

- Puesto que los subprogramas (procedimientos y funciones) forman la mayor parte de la interfaz a un paquete, las especificaciones de subprogramas son muy importantes. Por ejemplo, la especificación siguiente de una interfaz:

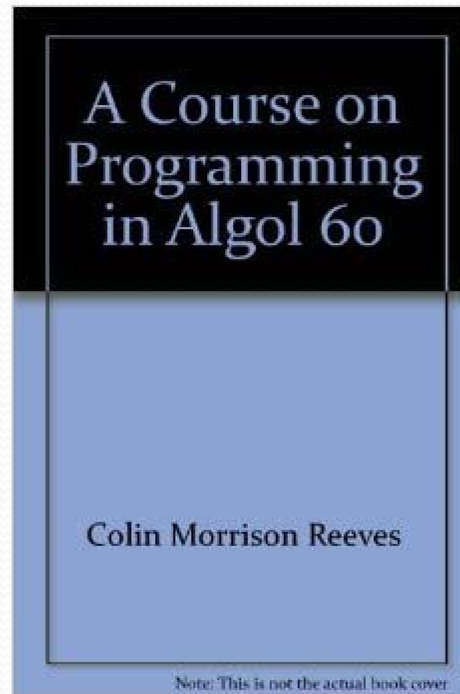
```
procedure BINSEARCH (T: MESA; VISTO: FLOAT;
                     out LUGAR: INTEGER; out
                     ENCONTRADO: BOOLEAN);
```

Dice al compilador que BINSEARCH es un procedimiento con cuatro parámetros.

Subprogramas

- Los primeros dos son parámetros de entrada de tipo MESA y FLOAT, respectivamente. El tercero y cuarto son parámetros de salida de tipo INTEGER y BOOLEAN, respectivamente.
- Esta es la interfaz entre el procedimiento BINSEARCH y sus invocadores, esto es, la información que deben conocer tanto los usuarios como el programador.
- Una especificación de este tipo, usualmente aparecería como parte de la especificación de la interfaz de un paquete.

Problemas con la Estructura de Bloques



- A principios de los 1970s, varios investigadores del área de lenguajes de programación comenzaron a cuestionarse la estructura de bloques de Algol-60 y otros lenguajes de segunda y tercera generación.

Problemas con la Estructura de Bloques

- Mientras que se aceptaba que estos lenguajes tenían muchas características positivas, también se les culpaba por los problemas que ocurrían al escribir programas de gran tamaño.

- Algunos de estos problemas fueron descritos por Wulf y Shaw (de la Universidad Carnegie-Mellon, en Estados Unidos) en un artículo publicado en 1973 con el título “Global Variable Considered Harmful”.

Problemas con la Estructura de Bloques

- En este artículo, Wulf y Shaw identificaron cuatro problemas con la estructura de bloques:

- Efectos colaterales
- Acceso indiscriminado
- Vulnerabilidad
- Definiciones no traslapables

- A continuación, describiremos cada uno de ellos.

Problemas con la Estructura de Bloques

- **Efectos colaterales:** Este término se refiere al cambio efectuado a una variable no local por parte de un procedimiento o una función.

- El problema se origina por el acceso oculto a una variable (o sea, por usar variables visibles desde otros procedimientos).

Problemas con la Estructura de Bloques

- Por ejemplo, supongamos que tenemos el siguiente procedimiento en Algol-60:

```
integer procedure Max(x,y);     integer x,y;
  begin
        suma := suma + 1;
        Max := if x>y then x else y;
  end;
```

Problemas con la Estructura de Bloques

- Claramente, este procedimiento devuelve el máximo de entre 2 enteros que se le pasan como argumentos. Sin embargo, tiene también el efecto colateral de alterar el valor de 'suma', que se supone que está definida en un bloque exterior.

- Podemos suponer que la intención del programador es llevar un contador del número de veces que se invoca el procedimiento Max.

Problemas con la Estructura de Bloques

- ¿Cuál es el problema con esto?

- El problema es que hace difícil determinar el efecto que producirá la invocación a un procedimiento. Por ejemplo, si tenemos:

```
longitud := Max(necesito,solicito);
```

Es obvio que esta invocación a Max involucra a las variables 'necesito', 'solicito' y 'longitud'.

Problemas con la Estructura de Bloques

- Estas variables son claramente parte de la interfaz al procedimiento Max.
- No es posible imaginar que este procedimiento modifica a la variable 'suma', sin ver la implementación del procedimiento.
- Esto genera problemas potenciales de mantenimiento y depuración del código.

Problemas con la Estructura de Bloques

- Consideremos ahora la siguiente invocación:

```
suma := 10;
longitud := Max(suma, 10);
```

El procedimiento Max modificará uno de los parámetros que se le pasan (suma). El valor exacto que regresa Max depende de si Max incrementa 'suma' antes o después de evaluar sus parámetros.

Problemas con la Estructura de Bloques

- Así mismo, el valor de 'longitud' será 11 si los parámetros se pasan por nombre o 10 si se pasan por valor.

- Todos estos problemas ocurren porque la variable global 'suma' está oculta de la interfaz a Max.

- En realidad, 'suma' es un parámetro tanto de entrada como de salida, pero no aparece en la lista de parámetros.

Problemas con la Estructura de Bloques

- FORTRAN permitía efectos colaterales cuando se usaba parámetros de salida (todos los parámetros se pasaban por referencia) y con los bloques COMMON.
- Sin embargo, si no se usaba COMMON, podían evitarse los efectos colaterales prácticamente por completo.
- Desafortunadamente, los efectos colaterales son una consecuencia natural de la estructura de bloques.

Problemas con la Estructura de Bloques

- Esto se debe a que el anidamiento dentro de un bloque implica que todas las variables declaradas en ese bloque son visibles y, por tanto, alterables.
- De tal forma, podemos resumir el problema de los efectos colaterales diciendo que éstos resultan del acceso oculto a una variable.

Problemas con la Estructura de Bloques

- **<u>Acceso indiscriminado</u>:** Es la incapacidad de impedir el acceso a una cierta variable.
- No hay forma de arreglar las declaraciones en un lenguaje estructurado de tal forma que no ocurra el acceso indiscriminado.

Problemas con la Estructura de Bloques

- Ejemplo:

```
begin
   integer array S[1:100];
   integer TOP;

   procedure Push(x);  integer x;
      begin    TOP := TOP + 1;   S[TOP] := x; end;

   procedure  Pop(x);  integer x;
       begin    Pop := S[TOP];   TOP := TOP – 1;  end;

   TOP := 0;

   … uso de Push y Pop …
end
```

Problemas con la Estructura de Bloques

- La variable S, que representa la pila, debe declararse en el mismo bloque que Push y Pop, a fin de que pueda ser visible a estos dos procedimientos.

- Sin embargo, debido a esta razón, un usuario puede usar o alterar inadvertidamente el valor de S a través de Push y Pop.

Problemas con la Estructura de Bloques

- El acceso indiscriminado crea un problema de mantenimiento, ya que una cierta estructura podría ser usada por cualquier procedimiento dentro de su mismo entorno.
- Por tanto, cambiar dicha variable implica modificar (o al menos) checar lo demás.

Problemas con la Estructura de Bloques

- **Vulnerabilidad:** Un segmento de programa no puede preservar acceso a una variable bajo ciertas circunstancias.

- El problema básico es que pueden interponerse nuevas declaraciones entre la definición y el uso de una variable.

Problemas con la Estructura de Bloques

- Supongamos que tenemos un programa muy grande en Algol que tiene la siguiente estructura:

```
begin
   integer x;
   ... muchas líneas de código ...
   begin
   ... muchas líneas de código ...
   ... x := x+1; ...
   ..........................................
   end;
   .......
end;
```

Problemas con la Estructura de Bloques

- Supongamos que hay tantas líneas de código entre la definición y el uso de 'x', que éste llena muchas páginas.

- Supongamos ahora que en el proceso de dar mantenimiento a este programa, decidimos que necesitamos una nueva variable local en el bloque interno. Decidimos llamar a 'x' a esta variable, sin percatarnos de que ese nombre ya está siendo utilizado en ese bloque.

Problemas con la Estructura de Bloques

- El resultado de esta modificación es el siguiente:

```
begin
   integer x;
   .... muchas líneas de código ....
begin real x;  comment NUEVA DECLARACION;
   .... muchas líneas de código ....
   ... x := x+1; ...
   ..............................................
   end;
   .......
end;
```

Problemas con la Estructura de Bloques

- Podemos ver claramente que el acceso a la declaración externa de ‘x’ ha sido bloqueado, y que ahora la sentencia ‘x := x+1’ se refiere a la nueva variable x.
- La nueva declaración de x ha sido interpuesta entre la definición original de ‘x’ y su uso.
- En resumen, vulnerabilidad significa que un segmento de programa no puede preservar el acceso a una variable.

Problemas con la Estructura de Bloques

- **Definiciones no Traslapadas:** Esto significa que no podemos controlar el acceso compartido a variables.
- Supongamos que tenemos un cierto conjunto de procedimientos P1 y P2, los cuales requieren de un arreglo DA a través del cual se comunican (o sea, lo comparten), y otro conjunto de procedimientos P2, P3 y P4 requieren un arreglo DB.

Problemas con la Estructura de Bloques

- Entonces tenemos que declarar a DA y DB en un nivel desde el cual sean visibles a P1, P2,P3 y P4, a pesar de que algunos de estos procedimientos no necesitan saber de alguno de estos dos arreglos.
- Esto crea problemas de mantenimiento y de seguridad, además de hacer más ineficiente el uso de memoria si los arreglos son muy grandes.

Problemas con la Estructura de Bloques

```
begin
    array DA[...];
    array DB[...];
    procedure P1;      ... ;
    procedure P2;      ... ;
    procedure P3;      ... ;
    procedure P4;      ... ;
    ...
end
```

Alternativas para resolver estos problemas de acuerdo a Wulf y Shaw

- 1) **El mecanismo por omisión no debiera ser el extender el entorno de una variable a los bloques más internos :** Los lenguajes estructurados tradicionalmente tienen un acceso hereditario implícito a las variables de los bloques en los cuales éstas se encuentran inmersos.

Alternativas para resolver estos problemas de acuerdo a Wulf y Shaw

- Esta es precisamente la fuente de los efectos colaterales, el acceso indiscriminado y la vulnerabilidad, aunque el eliminar este mecanismo no garantiza la solución total de todos estos problemas.

Alternativas para resolver estos problemas de acuerdo a Wulf y Shaw

- 2) **El derecho a acceder a un nombre debe ser del consentimiento mutuo del creador y de quien accede al nombre:** Quien crea un cierto nombre, debe ser capaz de definir quién puede accederlo y quien no.

- Esto eliminaría el acceso indiscriminado, la vulnerabilidad y las definiciones no traslapadas.

Alternativas para resolver estos problemas de acuerdo a Wulf y Shaw

- 3) **Los derechos de acceder a una estructura y los de acceder a sus subestructuras debieran estar separados:** Esto significa que el hecho de que podamos accder a una cierta estructura (p.ej., una pila o una cola de espera) no debiera implicar que tenemos acceso al mecanismo que la implementa.

Alternativas para resolver estos problemas de acuerdo a Wulf y Shaw

- Por ejemplo, no debiéramos tener acceso al arreglo o a la lista ligada usada para implementar la pila o la cola de espera.

- Este problema está asociado con el acceso indiscriminado.

Alternativas para resolver estos problemas de acuerdo a Wulf y Shaw

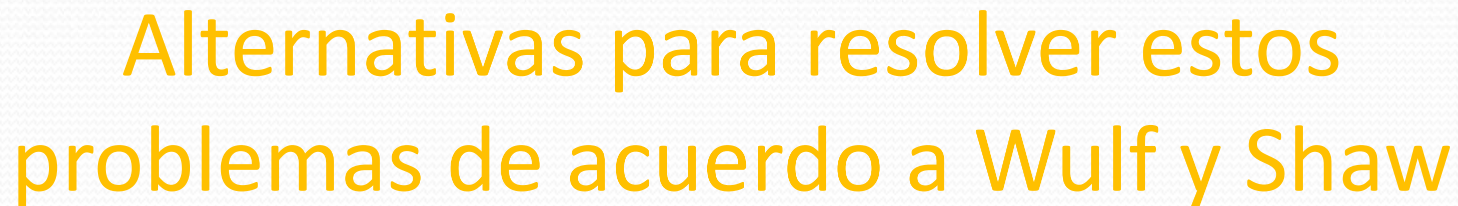
- 4) **Debiera ser posible distinguir entre diferentes tipos de acceso:** Por ejemplo, algunos usuarios debieran tener sólo acceso de lectura a una cierta estructura de datos, mientras otros podrían tener acceso de lectura y escritura.

- Esto ayudaría a resolver los problemas de los efectos colaterales y la vulnerabilidad.

Alternativas para resolver estos problemas de acuerdo a Wulf y Shaw

- 5) **La declaración de definición, el acceso a un nombre y la asignación de memoria debieran ser procesos separados:** En la mayor parte de los lenguajes estructurados estas tres funciones están conectadas muy íntimamente.

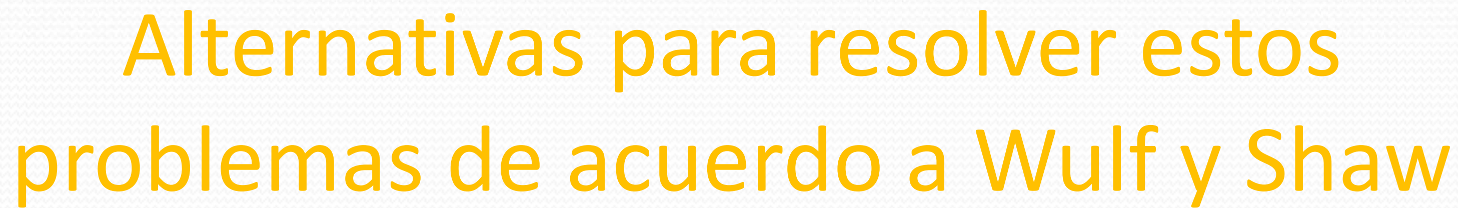
Alternativas para resolver estos problemas de acuerdo a Wulf y Shaw



- Por ejemplo, cuando se declara una variable en Algol, el nombre se define mediante su apariencia en la declaración, el acceso al nombre se determina por el nivel de anidamiento con respecto a los procedimientos existentes y la asignación de memoria y su liberación se determinan con respecto a los puntos de entrada y salida de un bloque.

Alternativas para resolver estos problemas de acuerdo a Wulf y Shaw

- Estas son tres funciones ortogonales (o sea, independientes). En algunos casos, los lenguajes de programación han intentado separar estas funciones.
- Cuando permitimos manejo dinámico de memoria (p.ej., el "**new**" de C o Pascal) por ejemplo, realmente estamos separando estas funciones, puesto que decidimos cuándo asignar y cuándo liberar memoria asociada con una variable, así como el lugar donde se usará su contenido.



Alternativas para resolver estos problemas de acuerdo a Wulf y Shaw

- Aunque Ada sigue siendo un lenguaje estructurado (con las desventajas asociadas a dichos lenguajes), cuenta con mecanismos mucho más sofisticados (p.ej., los paquetes) que permiten eliminar la mayor parte de las desventajas de los bloques.

Principios de Parnas

- Más o menos en la misma época en que Wulf y Shaw realizaban su trabajo, Parnas enunción dos importantes principios relacionados con el ocultamiento de información:
- 1) Debe proporcionarse al usuario toda la información necesaria para usar un módulo correctamente y nada más.
- 2) Debe proporcionar al implementador toda la información necesaria para completar el módulo y nada más.

Principios de Parnas

- De esta forma, el usuario de un módulo no sabrá cómo está implementado el mismo, y no podrá escribir programas que dependan de dicha implementación. Esto hará al módulo más fácil de mantener, puesto que el implementador sabrá exactamente qué cosas pueden cambiar y cuáles no.

- De manera análoga, los implementadores no tienen conocimiento del contexto de uso de su módulo, excepto por aquello que pueden ver en la interfaz. Esto simplifica también el mantenimiento del módulo, porque saben lo que pueden y no pueden cambiar de forma segura.

Paquetes

- Los paquetes de Ada soportan principios de ocultamiento de información antes descritos y permiten controlar el acceso a las declaraciones.
- La declaración de un paquete se divide en dos partes: una especificación de la interfaz y un cuerpo del módulo.

Paquetes

- La *especificación de la interfaz* define la interfaz entre el interior y el exterior del paquete. Por lo tanto, esta es la información del paquete que el usuario debe conocer.

- En consecuencia, el implementador debe saber la forma en la que se usará el paquete para definir una interfaz adecuada.

Paquetes

- El formato de la especificación de un paquete es el siguiente:

package UN_NOMBRE **is**

. . . especificación de nombres públicos. . .

end UN_NOMBRE;

Paquetes

- Un paquete tendrá entonces una parte privada y una pública, como una clase en un lenguaje orientado a objetos.

- Una declaración *privada* será visible de forma que pueda usarse en declaraciones de objetos, especificaciones de parámetros, etc., pero su estructura interna estará oculta de los usuarios del paquete.

Paquetes

- El cuerpo del paquete, el cual es conocido sólo por su implementador, proporciona la definición de cada nombre mencionado en la especificación.
- También puede declarar procedimientos, funciones y tipos locales, así como cualquier otra cosa requerida por la implementación.
- Todas estas declaraciones son privadas.

Paquetes

```
package COMPLEX_TYPE is
   type COMPLEX is private;
   I : constant COMPLEX;
   function + (X, Y : COMPLEX) return COMPLEX;
   function  - (X, Y : COMPLEX) return COMPLEX;
   function *  (X,Y : COMPLEX) return COMPLEX;
   function /  (X,Y : COMPLEX) return COMPLEX;
   function RE (X : COMPLEX) return FLOAT;
   function IM (X : COMPLEX) return FLOAT;
   function +  (X : FLOAT; Y : COMPLEX) return COMPLEX;
   function *  (X : FLOAT; Y : COMPLEX) return COMPLEX;

private
   type COMPLEX is record RE, IM : FLOAT := 0.0; end record;
   I : constant COMPLEX := (0.0, 1.0);
end COMPLEX_TYPE;
```

Paquetes

- El implementador de un paquete puede controlar, mediante la colocación de las declaraciones en la parte pública o privada del paquete, qué nombres puede acceder el usuario de un paquete.
- Cualquier cosa que se coloque en la especificación es pública y, por tanto, potencialmente accesible.