Conceptos Avanzados en Lenguajes de Programación

Tipos de Datos

<2016-08-29 lun>

Contents

Introducción

Tipos de Datos

- Hemos desarrollado una noción intuitiva de tipo de dato; ¿Que hay detras de la intuición?
  - Conjunto de valores de un "dominio" (la aproximación funcional)
  - Estructura interna de un manojo de datos, descripto al nivel de un conjunto pequeño de tipos fundamentales (aproximación estructural)
  - Clase de equivalencia de objetos (aproximación del implementador)
  - Conjunto de operaciones bien-definidas que pueden ser aplicadas a objetos de ese tipo (aproximación de abstracción)

Tipos de Datos

- Utilidad
  - Contexto implícito
  - Chequeo de tipos
    - Asegura que ciertas operaciones erróneas no ocurran
    - aunque no puede prevenir todas
  - polimorfismo surge cuando el compilador encuentra que no necesita saber ciertas cosas

Tipos de Datos

- **Fuertemente Tipado** se ha vuelto un término popular
  - como *programación estructurada*
  - informalmente, significa que el lenguaje previene al programador de aplicar operaciones a los datos que no son apropiados
- **Tipado Estático** significa que el compilador puede realizar todos los chequeos en tiempo de compilación.
- Ejemplos
  - Common Lisp is fuertemente tipado pero no **tipado estaticamente**
  - Ada es estáticamente tipado
  - Pascal es casi estáticamente tipado
  - Java es fuertemente tipado, con una mezcla no trivial de cosas que pueden ser chequeadas estaticamente y cosas que tienen que ser chequeadas dinámicamente.

Tipos de Datos

- Simples
  - Primitivos: integer, float, char, enum
  - Definidos por el Usuario
- Compuestos
  - Arreglos
    - strings
  - Arreglos asociativos
  - Registros
  - Union
  - conjuntos
  - listas
  - punteros
  - archivos

Sistema de Tipos

- Un **Sistema de Tipos** tiene reglas para
  - equivalencia de tipos (¿cuándo los tipos de dos valores son el mismo?)
  - compatibilidad de tipos (¿cuándo puede el valor de un tipo A ser usado en un contexto donde se espera el tipo B?)
  - inferencia de tipos (¿Cuál es el tipo de una expresión, dado el tipo de los operandos?)

Chequeo de Tipos

- Dos Aproximaciones: *equivalencia estructural* y *equivalencia por nombre*
  - La equivalencia por nombre esta basado en las declaraciones
  - La equivalencia estructural esta basada en la noción de significado detrás de esas declaraciones
  - Equivalencia por nombre es mas preferida hoy en dia.

Estructural vs. por Nombre

- a veces es preferible estructural

```
TYPE stack_element = INTEGER; (* or whatever type the user prefers *)
MODULE stack;
IMPORT stack_element;
EXPORT push, pop;
...
PROCEDURE push(elem : stack_element);
...
PROCEDURE pop() : stack_element;
...
```

- otras veces por nombre

```
TYPE celsius_temp = REAL;
fahrenheit_temp = REAL;
VAR c : celsius_temp;
    f : fahrenheit_temp;
```

```
BEGIN (* alias_types *)
    c := 100.0;
    f := c;                 (* this should probably be an error *)
```

Chequeo de Tipos: Coerción

Chequeo de Tipos: Coerción

- Coerción
  - Cuando una expresión es usada en un contexto donde un tipo diferente se espera, uno normalmente obtiene un error.
  - Pero, y en esta situación?:

```
var a : integer; b, c : real;
...

c := a + b;
```

- Muchos Lenguajes lo permiten.
- Puede ser basado solo en los tipos de los operandos (Fortran)

Chequeo de Tipos: Coerción

- Coerción
  - **C** usa mucha coerción, pero con reglas simples:
    - todos los `float` en expresiones se vuelven `double`
    - `short int` y `char` se vuelven `int` en las expresiones
    - Si es necesario, la precisión es removida cuando se asigna a lado izquierdo de la asignación.

Chequeo de Tipos: Coerción

- De hecho, las reglas de coerción son una relajación del chequeo de tipos
  - Nuevas opiniones lo consideran una mala idea
  - Lenguajes como Modula-2 y Ada no permiten coerción
  - C++, sin embargo lo usa en extremo

Chequeo de Tipos: Coerción

- Es importante entender la diferencia entre:
  - **Conversión de Tipos** que es *explícito* y
  - **Coerción de Tipos** que es *implícito*
  - para las conversiones a veces se usa la palabra *cast* (por C)

Arreglos

- Los Arreglos son el tipo compuesto mas importante en los lenguajes de alto nivel. Es una agrupación de elementos (usualmente) homogeneos en la cual los elementos individuales son son identificados por su posición en la agrupación relativo a su primer elemento.

Cuestiones de Diseño de Arreglos

- ¿Cuales tipos son legales para ser subíndices?
- ¿Es chequeado que el subíndice cumpla el rango definido?
- ¿Cuándo se liga el rango de subíndices?
- ¿Cuándo tiene lugar el alojamiento de espacio?
- ¿Cual es el número máximo de subíndices?
- ¿Pueden los arreglos ser inicializados?
- ¿Se pueden definir porciones (slices) de arreglos?

Accediendo a los elementos del Arreglo

- Es una función desde subíndices a elementos `array_name(index_value_list)` → `an element`
- Sintaxis
  - FORTRAN, PL/I, Ada usan *paréntesis*
    - Ada explícitamente usa paréntesis para mostrar uniformidad entre referencia de arreglos y llamadas a función porque ambas mapean resultados
  - La mayoría de los otros lenguajes usan *corchetes*

Tipos de los subíndices de los arreglos

- FORTRAN, C: solo enteros (integer)
- PASCAL: cualquier tipo ordinal (integer, boolean, char, enumeration)
- Ada: Enteros y enumeración (incluídos char y booleanos)
- Java: solo tipos enteros
- C, C++, Perl, y Fortran no especifican chequeo de rango
- Java, ML, C#, especifican chequeo de rango

Categoría de Arreglos

- Estático: rango de subíndices son ligados estáticamente y el alojamiento de memoria es estático (antes del tiempo de ejecución)
  - ventaja: eficiencia (no hay alojamiento dinámico)
- (stack)dinámico Fijo: los subíndices son ligados estáticamente, pero el alojamiento es hecho en tiempo de declaración
  - ventaja: eficiencia de espacio
- (stack)dinámico: rangos de subíndices son ligados dinámicamente y el almacenamiento es dinámico (hecho en tiempo de ejecución)
- (heap)dinámico Fijo: el almacenamiento es ligado dinámicamente pero fijo después del alojamiento.
- (heap)dinámico: la ligadura de los subíndices y el almacenamiento es dinámico y puede cambiar
  - ventaja: flexibilidad (los arreglos pueden crecer o disminuir durante la ejecución del programa)

Categoría de Arreglos

- Los arreglos de **C** y **C++** que incluyen el modificador `static` son *Estáticos*
- Los arreglos de **C** y **C++** sin el modificador `static` son *(stack)dinámicos Fijos*

- Los arreglos de **Ada** pueden ser *(stack)dinámicos*
- **C** y **C++** proveen arreglos *(heap)dinámicos Fijos* (**C#** con sus `ArrayList`)
- **Perl** y **JavaScript** soporta arreglos *(heap)dinámicos.*

(Stack) Dinámicos fijos

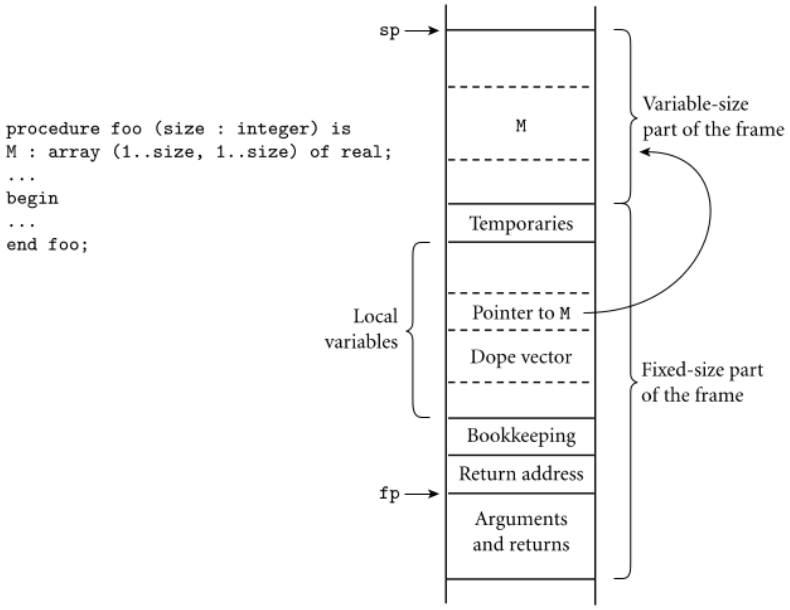


Figure 7.7 **Allocation in Ada of local arrays whose shape is bound at elaboration time.** Here `M` is a square two-dimensional array whose width is determined by a parameter passed to `foo` at run time. The compiler arranges for a pointer to `M` to reside at a static offset from the frame pointer. `M` cannot be placed among the other local variables because it would prevent those higher in the frame from having static offsets. Additional variable-size arrays are easily

Arreglos

- Elementos Contiguos
  - Dirigido por Columnas - solo en **Fortrand**
  - Dirigido por filas
    - usada por el resto de lenguajes
    - hace que el `array [a..b,c..d]` sea igual a `array [a..b] of array [c..d]`

Arreglos

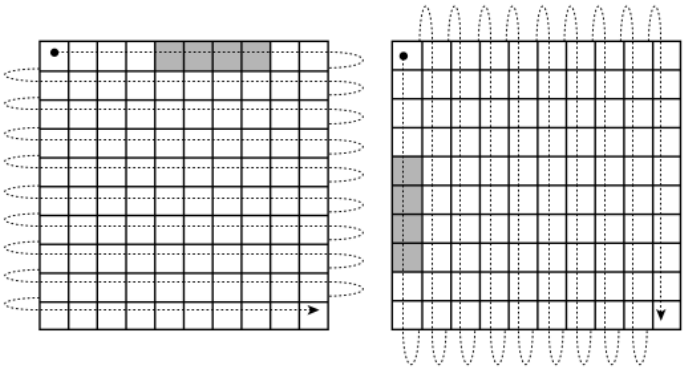


Figure 7.9 **Row- and column-major memory layout for two-dimensional arrays.** In row-major order, the elements of a row are contiguous in memory; in column-major order, the elements of a column are contiguous. The second cache line of each array is shaded, on the assumption that each element is an eight-byte floating-point number, that cache lines are 32 bytes long (a common size), and that the array begins at a cache line boundary. If the array is indexed from A[0,0] to A[9,9], then in the row-major case elements A[0,4] through A[0,7] share a cache line; in the column-major case elements A[4,0] through A[7,0] share a cache line.

Arreglos

- **Dos estrategias para arreglos**
  - Elementos continuos
  - punteros de filas
- **Punteros de Filas**
  - una opcion en **C**
  - permite a las filas colocarse en cualquier parte de la memoria
  - bueno para matrices cuando las filas son de diferente longitud
    - ejemplo arreglo de strings
  - requiere espacio para los punteros

Arreglos

```
char days[][10] = {
    "Sunday", "Monday", "Tuesday",
    "Wednesday", "Thursday",
    "Friday", "Saturday"
};
...
days[2][3] == 's';  /* in Tuesday */
```

```
char *days[] = {
    "Sunday", "Monday", "Tuesday",
    "Wednesday", "Thursday",
    "Friday", "Saturday"
};
...
days[2][3] == 's';  /* in Tuesday */
```

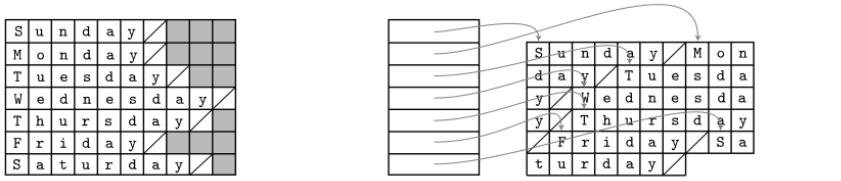


Figure 7.10 **Contiguous array allocation v. row pointers in C.** The declaration on the left is a true two-dimensional array. The slashed boxes are `NUL` bytes; the shaded areas are holes. The declaration on the right is an array of pointers to arrays

Inicialización de Arreglos

- Algunos Lenguajes permiten inicialización en el tiempo de alojamiento.
  - ejemplo de **C**, **C++**, **Java**, **C#**
    - `int list [] = {4, 5, 7, 83}`
  - cadena de caracteres en **C** y **C++**
    - `char name [] = "freddie";`
  - Array of strings en **C** and **C++**
    - `char *names [] = {"bob", "jake", "Joe"};`
  - **Java**
    - `String[] names = {"Bob", "Jake", "Joe"};`

Operaciones de Arreglos

- **APL** provee el mas poderoso conjunto de operadores para procesar vectores y matrices y operaciones unarias (por ejemplo revertir elementos de una columna)
- **Ada** permite asignación de arreglos y concatenación
- **Fortran** provee operaciones *elementales* a causa de que son entre pares de elementos del arreglo
  - Por ejemplo, el operador + entre dos arreglos resulta en un arreglo con la suma de los pares de elementos de los dos arreglos.

Arreglos

- Ejemplo `A : array [L1..U1] of array [L2::U2] of array [L3..U3] of elem;`
  - $D1 = U1 - L1 + 1$
  - $D2 = U2 - L2 + 1$
  - $D3 = U3 - L3 + 1$
  - $S3 =$ tamaño de `elem`
  - $S2 = D3 * S3$
  - $S1 = D2 * S2$

$A(i, j, k) =$ `address of A` $+(i * S1) + (j * S2) + (k * S3) - [(L1 * S1) + (L2 * S2) + (L3 * S3)]$

Slices

- Una *porción* (slice) de un arreglo es una subestructura de un arreglo; un mecanismo de referenciación.
- Los *Slices* son útilies en lenguages que tienen operaciones sobre arreglos (APL, FORTRAN etc).

Slices

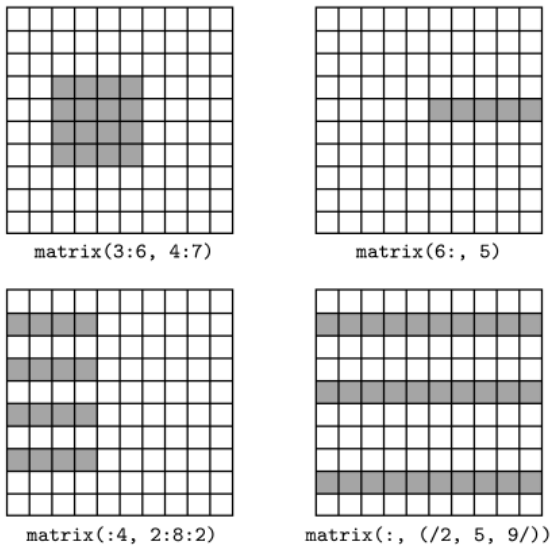
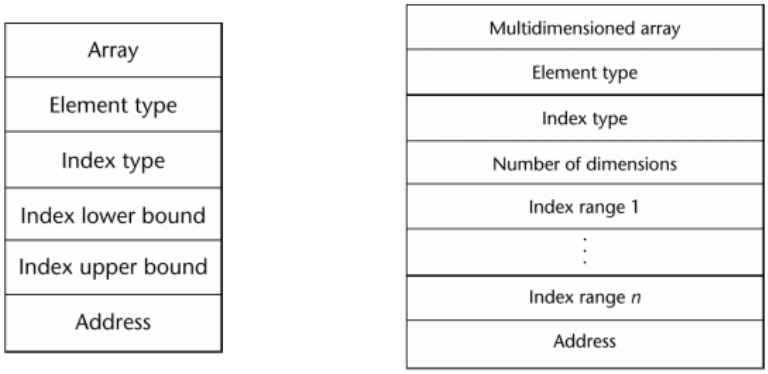


Figure 7.6 Array slices (sections) in Fortran 90. Much like the values in the header of an enumeration-controlled loop (Section 6.5.1), $a:b:c$ in a subscript indicates positions a, $a+c$, $a+2c$, ...through b. If a or b is omitted, the corresponding bound of the array is assumed. If c is omitted, `1` is assumed. It is even possible to use negative values of c in order to select

Descriptores en Tiempo de Compilación



Arreglos Asociativos

- Un *arreglo asociativo* es una colección no ordenada de elementos de datos que son indexados por un numero igual de valores llamados *claves*

(keys)

  - claves definidas por el usuario deben ser almacenadas

- Ahora llamados *Diccionarios*
- en **PERL**
  - Nombres comenzando con `%`; literales son delimitados con paréntesis
    * `%hi_temps` = ("Mon" => 77, "Tue" => 79, "Wed" => 65, ...),
  - Para acceder se usan llaves y claves:
    * `%hi_temps{"wed"} = 83;`
  - Los elementos pueden ser removidos con `delete`
    * `delete %hi_temps{"Tue"}`

Strings

- *Strings* son en realidad arreglos de caracteres
- Son frecuentemente casos especiales, para darles flexibilidad (como polimorfismo y tamaño dinámico) que no es disponible para arreglos en general
  - Es mas facil proveer estas cosas para *strings* que para arreglos en general porque los *strings* son de una dimensión y no circulares.

Tipo Registros

- Un registro es un conjunto posiblemente heterogeneo de elementos de datos en el cual los elementos individuales son identificados por su nombre
- Cuestiones de Diseño
  - ¿Cual es la sintaxis para referenciar los campos?
  - ¿Son permitidas las referencias elípticas?

Tipo Registros

- Cobol

```
01 EMPLOYEE-RECORD.
   02 EMPLOYEE-NAME.
      05 FIRST    PICTURE IS x(20).
      05 MIDDLE   PICTURE IS x(10).
      05 LAST     PICTURE IS x(20).
   02 HOURLY-RATE PICTURE IS 99v99.
```

- Ada

```
type Employee_Name_Type is record
   First : String (1..20);
   Middle : String (1..10);
   Last : String (1..20);
end record;
type Employee_Record_Type is record
   Employee_Name: Employee_Name_Type;
   Hourly_Rate: Float;
end record;
Employee_Record: Employee_Record_Type;
```

Registros

- Referencia a los campos
  - **COBOL** `field_name OF record_name_1 OF ... OF record_name_n`
  - Otros (notación con punto) `record_name_1.record_name_2. ... record_name_n.field_name`
- Referencias completamente calificadas: debe incluir todo el camino de nombres de registros.
- Referencia elíptica: permite no especificar nombres intermedios siempre que la referencia sea no ambigua. Ej: `FIRST OF EMP-REC` en **COBOL**

Operaciónes de Registros

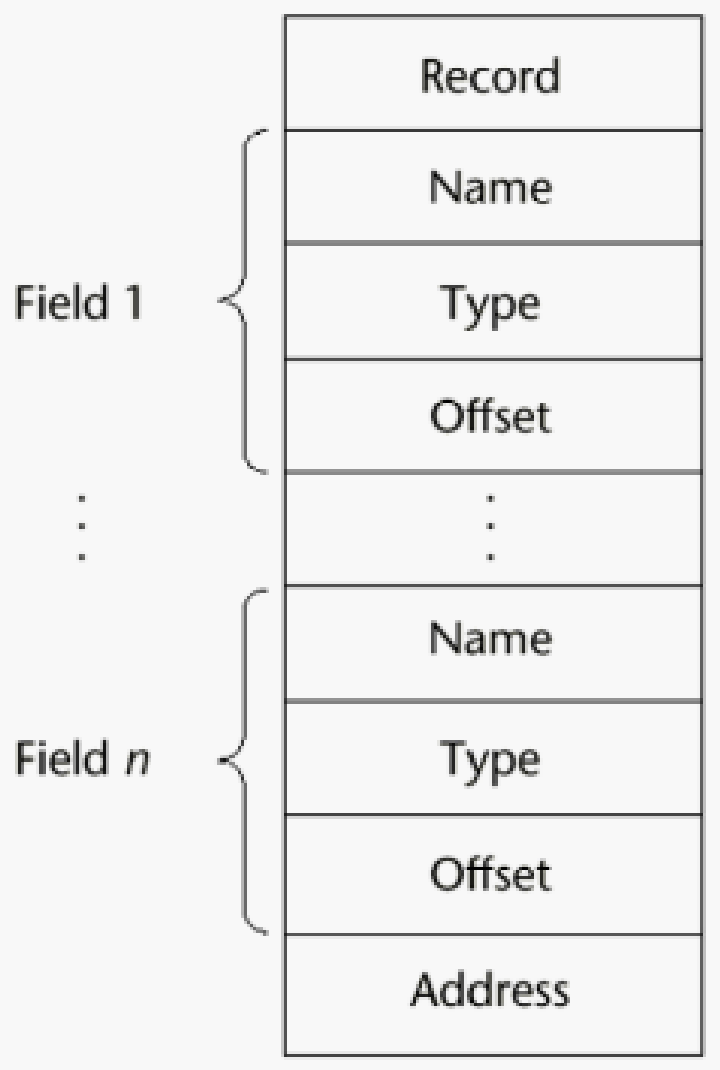
- La asignación es muy común si los tipos son identicos

- **Ada** permite comparación de registros
- Los registros de **Ada** pueden ser inicializados con conjunto de literales
- **COBOL** provee `MOVE CORRESPONDING`
  - copia un campo de un registro origen al correspondiente campo en el registro destino.

Comparación con Arreglos

- Tiene un diseño directo y seguro
- Son usados cuando el agrupamiento de datos es heterogeneo
- El acceso es mucho mas rápido que en arreglos porque el acceso a los nombres de los campos es estático

Implementación de Registros



Un desplazamiento de dirección relativo al comienzo del registro es asociado con cada campo.

Tipo Uniones

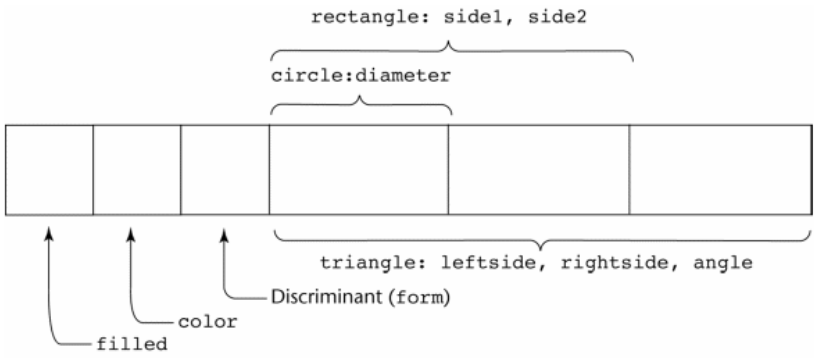
- Una *Union* es un tipo a cuyas variables se les permite almacenar diferentes valores de tipo (estructura) en diferentes tiempos durantes la

ejecución.

- Cuestiones de Diseño
  - ¿Debería requerirse chequeo de tipos?
  - ¿Deberían incluirse como tipos particulares de Registros?
- **Fortran**, **C**, y **C++** provee constructores de *Union* sin soporte para chequeo de tipos se llaman *uniones libres*
- Chequeo de tipos en *Uniones* requieren que se incluya un indicador de tipo llamado *discriminante*
  - soportado por **Ada**

tipo Union de Ada

```
type Shape is (Circle, Triangle, Rectangle);
type Colors is (Red, Green, Blue);
type Figure (Form: Shape) is record
  Filled: Boolean;
  Color: Colors;
  case Form is
      When Circle => Diameter : Float;
      When Triangle =>
             LeftSide, Rightside: Integer;
             Angle: Float;
      when Rectangle => Side1,Side2: Integer;
  end case;
end record;
```



Evaluación de Uniones

- también llamados registros variantes
- Es una construcción potencialmente insegura
  - no permite chequeo de tipos o es muy caro
- **Java** y **C#** no soportan uniones
  - Como reflejo de la creciente preocupación por la seguridad en los lenguajes de programación
- La falta de discriminante (tag) significa que uno no sabe lo que hay almacenado
- La posibilidad de cambiar el discriminante permite acceder a campos erroneamente

Punteros y Tipos Recursivos

Tipo Punteros

- Los Punteros sirven para dos propósitos:
  - acceso eficiente (y a veces intuitivo) a objetos muy elaborados (como en **C**)
  - creación dinámica de estructuras ligadas, en conjunción con administración de memoria *heap*
- Varios lenguajes (e.g. **Pascal**) restringen los punteros para acceder a cosas en el *heap*
- Los punteros son usados en un modo *por valor* de las variables
  - No se necesitan como modo *por referencia*

Punteros y Tipos Recursivos

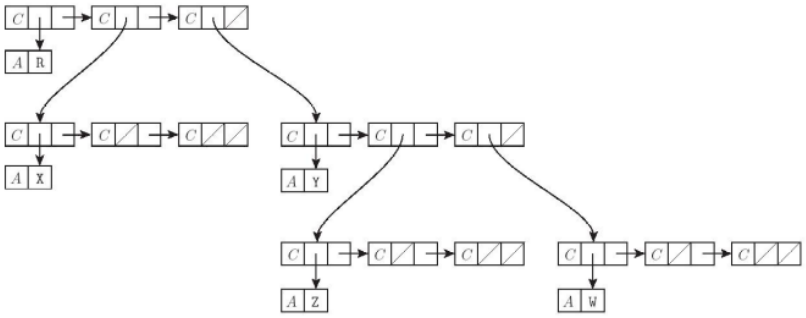


Figure 7.13: **Implementation of a tree in Lisp.** A diagonal slash through a box indicates a `nil` pointer. The *C* and *A* tags serve to distinguish the two kinds of memory blocks: cons cells and blocks containing atoms.

Punteros y Tipos Recursivos

- **C** punteros y arreglos
  - `int *a == int a[]`
  - ~int **a == int *a[]
- Las equivalencias no siempre ocurren
  - Especificamente, una declaración aloja un arreglo si se especifica un tamaño para la primera dimensión
  - en caso contrario aloja un puntero
    - `int **a, int *a[]` puntero a puntero a int
    - `int *a[n]`, arreglo de n elementos de punteros
    - `int a[n][m]` arreglo de dos dimensiones

Punteros y Tipos Recursivos

- El compilador tiene que ser capaz de establecer el tamaño de las cosas apuntadas por los punteros
  - Por lo tanto las siguientes no son válidas:
    - `int a[][]` mal
    - `int (*a)[]` mal

- regla de declaración de **C**: lee a la derecha tanto como puede (sujeto a paréntesis), luego a la izquierda, y luego sube de nivel y repite.
  - `int *a[n]` arreglo de n elementos de punteros a enteros
  - `int (*a)[n]` puntero a un arreglo de n elementos de enteros

Punteros y Tipos Recursivos

- Los problemas con punteros *cogados* se deben a:
  - desalojo explícito de objetos del *heap*
    - solo en lenguajes que tienen explícito desalojo
  - desalojo implícito de objetos elaborados
- Dos mecanismos de implementación para atrapar punteros *colgados*
  - *Tombstones* lapidas
  - *Locks and Keys* llaves y cerraduras

Listas

- Una *Lista* es definida recursivamente ya sea como una lista vacía o un par consistente de un objeto (que puede ser una lista o un átomo) y otra lista (mas corta)
  - Las *Listas* son ideales para programar en lenguajes lógicos y funcionales
    - En **Lisp** de hecho un programa *es* una lista, y puede extenderse a si mismo para construir una lista y ejecutarla
  - Las *Listas* pueden usarse en programas imperativos.

Archivos y Entrada/Salida

- Entrada/Salida (E/S) facilita al programa a comunicarse con el mundo externo
  - E/S interactiva y E/S con archivos
- Interactivo generalmente implica comunicación con usuarios humanos y dispositivos físicos

- Archivos generalmente se refieren a almacenamiento fuera de linea implementado por el sistema operativo.
- Archivos pueden ser categorizados en:
  - Temporarios
  - Persistentes