# Lenguajes de Programación

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

# Aspectos de Diseño

- "**eq**" regresa "t" (cierto) si "x" es igual a "y", y "nil" (falso) de lo contrario.
- Algunas versiones de LISP proporcionan tipos adicionales de átomos, tales como las cadenas.
- En estos casos, se proporcionan también operaciones especiales para manipular estos objetos.

# Aspectos de Diseño

- Ejemplos:

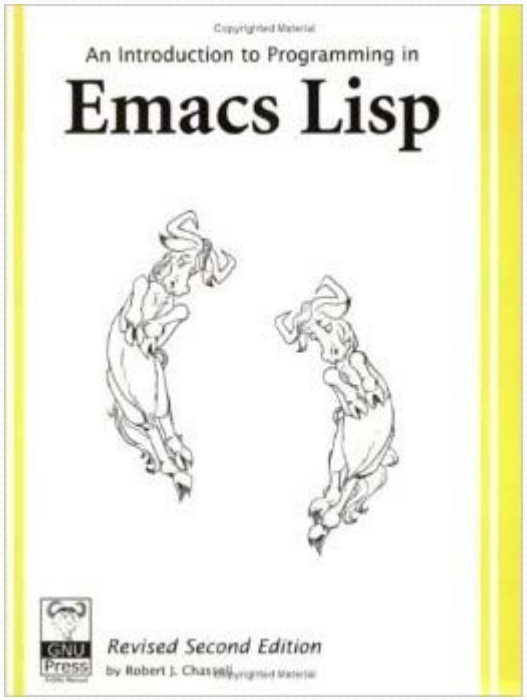
  string=, string<, string>, string <=, string>=,

  make-string, string-upcase, string-downcase, etc.

# Aspectos de Diseño

- El método de estructuración de datos que proporciona LISP es llamado "**list**".
- Las listas se escriben en forma de expresiones-S, rodeando con paréntesis los elementos de las mismas, los cuales se separan mediante espacios en blanco.
- Las listas pueden tener cero, uno o más elementos, lo que sigue el **Principio Cero-Uno-Infinito**.

# Aspectos de Diseño




- Los elementos de las listas pueden a su vez ser listas también, de forma que este mismo principio se satisface para el nivel de anidamiento de las listas.

# Aspectos de Diseño

- Por razones históricas, se considera a la lista vacía ‘() como equivalente al átomo **nil**. Es decir:

**(eq '() nil)**

**(null '())**

- regresan, en ambos casos, cierto.

# Aspectos de Diseño

- Por esta razón, a la lista vacía se le suele denominar “null list”.
- Con la excepción de la lista vacía, todas las listas son no atómicas (es decir, no son átomos).
- Algunas veces se les denomina *valores de datos compuestos*.

# Aspectos de Diseño

- Puede averiguarse si algo es un átomo usando el predicado "**atom**":

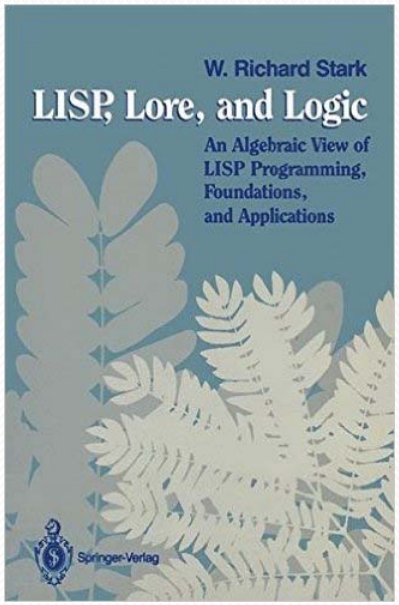
**(atom 'c)**

t

**(atom '(()) )**

nil

# Aspectos de Diseño




- Puesto que las listas son el método principal para estructurar datos en LISP, el lenguaje proporciona también una serie de constructores (es decir, operaciones usadas para construir listas) y selectores (o sea, operaciones usadas para separar listas).

# Aspectos de Diseño

- LISP tiene un constructor (**cons**) y dos selectores (**car** y **cdr**).

- El primer elemento de una lista se selecciona usando "car":

**(car '(mi nombre es Carlos))**

mi

# Aspectos de Diseño

- El primer elemento de una lista puede ser un átomo o una lista, y "**car**" lo retornará, sin importar su naturaleza:

**(car '((mi) (nombre) (es) (Carlos)))**

'(mi)

# Aspectos de Diseño

- Debe advertirse que el primer argumento de "**car**" debe ser una lista no vacía (de lo contrario, no podría tener un primer elemento y LISP retornaría un mensaje de error).

- Además, "**car**" puede regresar un átomo o una lista, dependiendo del argumento que tenga el primer elemento de la lista a la que se aplique.

# Aspectos de Diseño

- La función "**cdr**" regresa toda la lista, excepto su primer elemento.

- Por ejemplo:

**(cdr '(mi nombre es Carlos))**

'(nombre es Carlos)

# Aspectos de Diseño

- También "**cdr**" requiere que su argumento sea una lista no vacía (de lo contrario no podríamos remover su primer elemento, y LISP proporcionaría un mensaje de error).

- Sin embargo, a diferencia de "**car**", "**cdr**" SIEMPRE regresa una lista, aunque ésta podría ser la lista vacía en caso de que el argumento tenga un solo elemento.

# Aspectos de Diseño

- Ejemplo:

**(cdr '(Carlos))**

'()

# Aspectos de Diseño

- Es importante hacer notar que "**car**" y "**cdr**" son funciones puras. Es decir, no modifican su argumento.
- Puede considerarse que durante su operación utilizan una copia de la lista que se les pasa como argumento.
- Por ejemplo, "**cdr**" no borra realmente el primer elemento de una lista sino que regresa una lista idéntica a su argumento, si bien ésta carece del primer elemento.

# Aspectos de Diseño

- El único constructor de LISP, "**cons**", agrega un nuevo elemento al inicio de una lista.

- Por ejemplo:

**(cons 'mi '(nombre es Carlos))**

'(mi nombre es Carlos)

# Aspectos de Diseño

- Advierta que "**cons**" es realmente el inverso de "**car**" y "**cdr**":

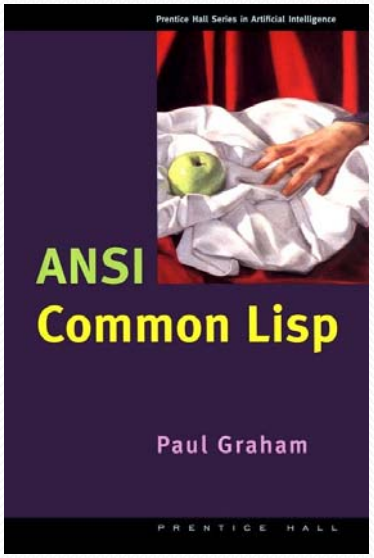
| | |
|---|---|
| **(car '(my name is Carlos))** | 'my |
| **(cdr '(my name is Carlos))** | '(name is Carlos) |
| **(cons 'my '(name is Carlos))** | '(my name is Carlos) |

# Aspectos de Diseño

- Por lo tanto, cualquier lista que podamos construir, también la podemos separar y viceversa.

- Al igual que "**car**" y "**cdr**", "**cons**" es una función pura.

# Aspectos de Diseño




- Esto significa que realmente no agrega nuevos elementos al inicio de su segundo argumento, sino que actúa como si hubiese construido una lista completamente nueva cuyo primer elemento es el argumento pasado a "**cons**" y el resto de esos elementos se copian de su segundo argumento.

# Aspectos de Diseño

- Las listas suelen construirse recursivamente. Si hacemos:

**(cons '(a b) '(c d))**

- el resultado es:

'((a b) c d))

# Aspectos de Diseño

- Esto es debido a que el primer elemento que se agregó es una lista y no un átomo.

- Sin embargo, probablemente lo que el programador quería hacer era generar la lista ‘(a b c d)

# Aspectos de Diseño

- Para hacer eso, necesitaríamos una función llamada "**append**":

**(append '(a b) '(c d))**

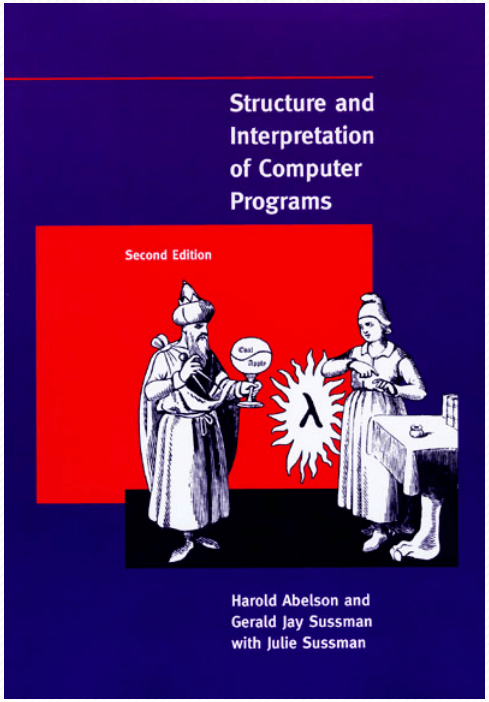
'(a b c d)

# Aspectos de Diseño

- La definición de "**append**" es muy sencilla:

```
(defun append (list1 list2)
  (cond
        ((null list1) list2)
        (t (cons (car list1) (append (cdr list1) list2)) )) )
```

# Aspectos de Diseño




- Esta solución es muy compacta y elegante; hace uso de recursividad de las primitivas para manejo de listas que vimos anteriormente.

# Aspectos de Diseño

- El condicional proporcionado por LISP es una de sus contribuciones más importantes a los lenguajes de programación modernos.
- Recordemos que el otro único lenguaje de programación importante de esa época era FORTRAN, y éste no tenía expresiones condicionales.

# Aspectos de Diseño

- Los matemáticos han reconocido por muchos años el valor de las expresiones condicionales y frecuentemente las han usado.

- Por ejemplo:

$$signum(x) = \begin{cases} 1, & \text{if } x > 0 \\ 0, & \text{if } x = 0 \\ -1, & \text{if } x < 0 \end{cases}$$

# Aspectos de Diseño

- En LISP, esta función se define como:

```
(defun signum(x)
(cond ((> x 0)          1)
        ((zerop x) 0)
        ((< x 0) -1)) )
```

# Aspectos de Diseño

- La razón por la que LISP usa (aparentemente) más paréntesis de los que realmente se necesitan es porque éstos se usan para definir entornos y para delimitar condiciones.

- El condicional podría ser un poco más legible si no requiriera paréntesis, pero entonces tendría que complicarse el intérprete.

# Aspectos de Diseño

- La iteración en LISP puro se efectúa a través de llamadas recursivas.

- Consideremos la función “suma-todo” que suma todos los elementos de una lista:

**(add-all '(1 2 3 4 5))**

15

# Aspectos de Diseño

- Puede definirse esta función usando recursividad:

```
(defun suma-todo (a)
  (cond ((null a) 0)
        (t (+ (car a) (suma-todo (cdr a)) )) ))
```

- Esta función es muy general y trabaja con cualquier número de argumentos.

# Aspectos de Diseño

- La recursividad y la iteración son, teóricamente, equivalentes.
- La única diferencia real entre ellas es la eficiencia: las iteraciones tienden a ser más eficientes que la recursividad.
- La razón es que la recursividad requiere de dos pasos: la construcción de una pila (invocaciones) y su remoción (regreso tras determinar el caso base).

# Aspectos de Diseño

- Sin embargo, si nos preocupa la eficiencia, podemos simular iteraciones usando “recursividad de cola”:

```
(defun suma-todo-it (a acc)
   (cond ((null a) acc)
         (t (suma-todo-it (cdr a) (+ acc (car a)) )) ))
```

# Aspectos de Diseño

- Su uso es ahora el siguiente:

**(suma-todo-it '(1 2 3 4 5) 0)**

15

# Aspectos de Diseño

- Aunque también puede preservarse la sintaxis previa:

```
(defun suma-todo (a)
  (suma-todo-it (a 0)))
```

# Aspectos de Diseño

- Los conectivos lógicos se evalúan condicionalmente.
- Las operaciones Booleanas tradicionales son válidas en LISP:

**(and $x_1$, $x_2$, $x_3$, . . ., $x_n$)**

**(or $x_1$, $x_2$, $x_3$, . . . , $x_n$)**

# Reglas de Entorno Dinámico

- LISP usa reglas de entorno dinámico.

- Es decir, que una función se llama en el ambiente de su invocador.

```
(defun raiz-aux (d)
        (list (/ ( - d b) (* 2 a))
(/ (+ d b) (* 2 a))))
```

# Reglas de Entorno Dinámico

```
(defun raices (a b c)
        (raiz-aux (sqrt (- (expt b 2) (* 4 a c)))))
```

- Aquí, “a” y “b” son accesibles a “raiz-aux” debido a que se usan reglas de entorno dinámico.
- De lo contrario, se habrían tenido que pasar como argumentos.

# Reglas de Entorno Dinámico

- Lo mismo aplica a "**let**": no sería de mucha utilidad si no heredara acceso a los nombres asociados a su entorno circundante.

- Sin embargo, las reglas de entorno dinámico tienen sus problemas.

# Reglas de Entorno Dinámico

- Ejemplo:

  (**set** ‘val 2)

  2

  (**defun** twice (func val) (func (func val)))

- Si hacemos: (twice ‘add1 5) ==> 7

# Reglas de Entorno Dinámico

- Sin embargo:

  (twice '(**lambda** (x) (* val x)) 3)

- regresa 27 en vez de 12.

# Reglas de Entorno Dinámico

- El problema es que “val” re-escribe la definición previa.
- Una forma de resolver este problema es usando un nombre distinto para “val” en cualquiera de sus dos instancias.
- De cualquier forma, LISP viola el **Principio de Ocultamiento de Información**.

# Reglas de Entorno Dinámico

- Este problema se descubrió desde los inicios del LISP y se conoce como el problema del “funarg”, o “functional argument”.
- Se trata simplemente de la diferencia entre usar reglas de entorno dinámico o estático.

# Reglas de Entorno Dinámico

- Sin embargo, LISP proporciona una solución alternativa que evita recurrir a reglas de entorno estático.

- Existe una forma especial llamada "**function**", la cual asocia una expresión lambda a su ambiente de definición.

# Reglas de Entorno Dinámico

- Ejemplo:

  (twice (function (lambda (x) (* val x))) 3)

  12

- Esto da la respuesta correcta, porque "**function**" ha asociado la expresión lambda con su ambiente de definición.

# Reglas de Entorno Dinámico

- O sea, que "**function**" introduce reglas de entorno estático en el lenguaje.
- Sin embargo, esto introduce una excepción al lenguaje y viola el **Principio de Regularidad**.
- Una opción más viable habría sido usar reglas de entorno estático (como en Scheme).

# Expresiones Lambda

- Las expresiones lambda son funciones anónimas (o sea funciones a las que no se les ha asociado un nombre).

- Por ejemplo:

  (**lambda** (x) (cons val x)

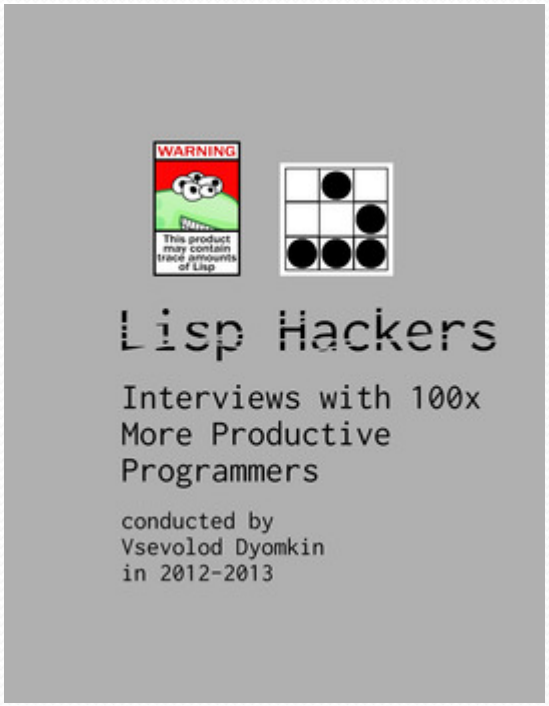
- representa a la función de **x** cuyo valor es '(**cons** val x)'.

# Expresiones Lambda

- Esto es equivalente a:

(**defun** consval(x) (cons val x))

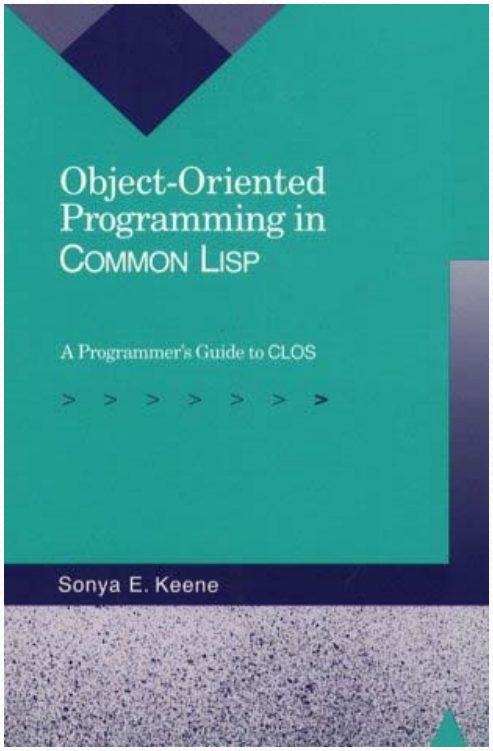
# Expresiones Lambda




- Las expresiones lambda son valores que pueden ser manipulados como cualquier otra lista en LISP.

- En particular, se les puede pasar como parámetros.

# Expresiones Lambda




- Los argumentos de lambda pueden ser constantes numéricas o simbólicas, variables, listas o aplicaciones de funciones.

# Expresiones Lambda

- Nótese sin embargo que una expresión lambda no es evaluable.
- Las únicas expresiones que pueden evaluarse en LISP son los átomos y las aplicaciones de funciones.
- El propósito de una expresión lambda es aplicarse a ciertos argumentos, pero no puede ser evaluada por sí misma.

# Expresiones Lambda

- Esto viola el **Principio de Ortogonalidad**, porque las expresiones lambda tienen significado sólo en conexión con las expresiones que aplican funciones, pero no tienen significado por sí mismas.

# Expresiones Lambda

- Los funcionales pueden reemplazar a las expresiones lambda.

- Por ejemplo:

```
(defun bu (f x)
        (function (lambda (y) (f x y))))
```

- convierte una función binaria en una unaria.

# Expresiones Lambda

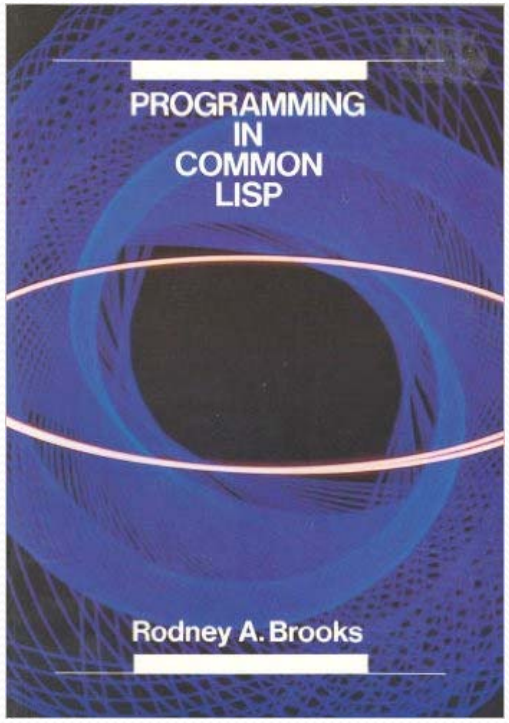
- Nótese que:

  (bu ‘cons val)

- es el equivalente de:

  (**lambda** (x) (**cons** val x))

# Expresiones Lambda




- Aunque las expresiones lambda en sí no son ciudadanos de primera clase en LISP, si combinación con "**function**" puede comportarse de manera casi idéntica.

# Paso de Parámetros

- En LISP los parámetros se pasan mediante un mecanismo llamado “compartición” (*sharing*):

- El mismo objeto que es asociado y/o designado por el i-ésimo parámetro que se pasa, se asocia, tras la invocación a la función, al i-ésimo parámetro del prototipo de la función.

# Objetos de Datos

- Los principales objetos de datos son los “átomos”, los cuales se organizan en listas.
- Los átomos son números o símbolos.
- Un símbolo es una constante alfanumérica que se asocia con algún objeto.

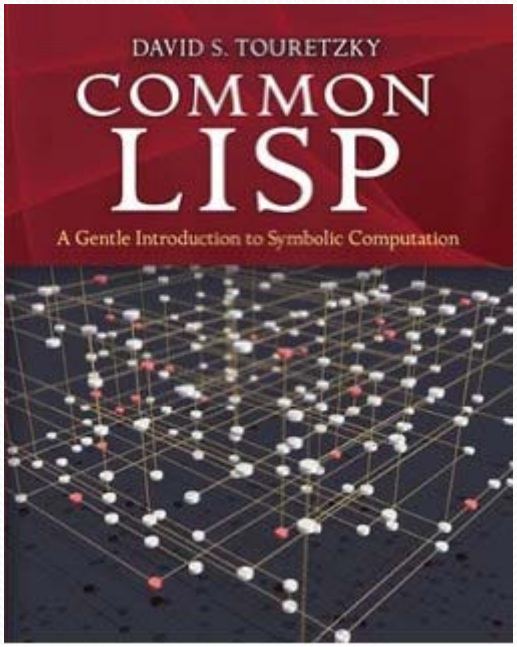
# Selectores

- "**car**" y "**cdr**" nos permiten crear y manipular estructuras de datos (selectores):

  (1) Cuyo tamaño no puede ser determinado antes del tiempo de ejecución

  (2) cuyos elementos son miembros de un grupo diverso sin reglas especiales que apliquen a todos ellos.

# Selectores




- Cada elemento se identifica mediante un símbolo único a él, y no mediante un símbolo que (como los números) es “predecible” dado el objeto a ser representado.

# Quote

- Construimos símbolos usando el operador "**quote**":

(**quote** foo) => ('foo)

- Esto significa "regresa el objeto precedido por "**quote**" sin evaluarlo".

# Modo Imperativo

- LISP tiene una forma "**progn**" que evalúa una secuencia de expresiones a fin de regresar el valor de la última de ellas.
- "**progn**" es la versión en LISP de la sentencia compuesta de Algol.
- Sin embargo, a diferencia de ella, sólo puede contener constantes, aplicaciones o saltos; LISP no proporciona ningún mecanismo análogo al bloque de Algol.

# Ciclos

- Las versiones modernas de LISP introducen constructores para ciclos de diversos tipos, muchos de los cuales suelen estar modelados a partir de Algol 60.

- Sin embargo, al contar con los condicionales y la recursividad, estos constructores para ciclos son realmente innecesarios.

# Ambientes Denominativos

- Uno de los rasgos distintivos de LISP es que para entenderlo, es necesario comprender muy bien sus mecanismos de asociación entre nombres y valores ("*binding*").

- Un nombre en Pascal, por ejemplo, designa un lugar en particular.

- Asignar un valor a una variable llamada "foobar" significa, en efecto, tomar algún objeto y depositarlo en una caja llamada "foobar".

# Ambientes Denominativos

- Lo que estuviese previamente en la caja de “foobar” desaparecerá.
- En LISP, sin embargo, “asignar un nuevo valor a alguna variable” significa realmente “asociar una nueva etiqueta denominativa a algún objeto”.
- Los nombres no designan lugares; simplemente corresponden a etiquetas denominativas que pueden colocarse a cualquier objeto que elijamos.

# Ambientes Denominativos

- Esto tiene dos consecuencias importantes:
  - Remover una etiqueta denominativa de un objeto no cambia al objeto.
  - Un objeto puede, por lo tanto, tener muchas etiquetas denominativas asociadas a él.

# Ambientes Denominativos

- Por ejemplo, si hacemos en Algol-60:

  foobar:=un objeto rojo;
  bazball:=foobar;

- Si ahora cambiamos el objeto rojo por uno amarillo, eso no afecta a “bazball”, que sigue conteniendo un objeto rojo.

# Ambientes Denominativos

- Pero si ejecutamos instrucciones equivalentes en LISP:

(**setq** foobar un objeto rojo)

(**setq** bazball foobar)

- y cambiamos el objeto rojo por uno amarillo, ahora tanto foobar como bazball están asociados con un objeto amarillo.

# Ambientes Denominativos

- "**setq**" es la expresión para asignaciones en LISP.
- Lo que hace es asociar la etiqueta "foobar" a un objeto rojo; la segunda asignación asocia una etiqueta llamada "bazball" al objeto que ya había sido llamado "foobar".
- De tal forma, el objeto rojo tiene ahora dos etiquetas asociadas a él. Por eso la última sentencia cambió al mismo objeto.

# Ambientes Denominativos

- "**setq**" es una abreviación de "set quote".
- "**setq**" no evalúa su primer argumento; lo trata como si fuese un identificador, que es realmente lo que queremos cuando ejecutamos una asignación.
- El operador primitivo "**set**" evalúa sus dos argumentos. (setq foo 3) es equivalente a (set 'foo 3).

# Ambientes Denominativos

- La comprensión de las relaciones entre nombres y objetos en LISP es importante para entender su estrategia de creación de objetos y su idea de invocación de funciones.
- Cuando corre un programa en LISP, hace que se creen nuevos objetos.
- Estos objetos permanecen “vivos” durante todo el tiempo que se les necesite.

# Ambientes Denominativos

- Cuando un objeto deja de usarse, será reciclado, tarde o temprano, por el recolector de basura del lenguaje. Será hasta entonces que desaparezca.

- En Algol 60, sin embargo, los objetos se crean en puntos bien definidos durante la ejecución de un programa (al entrar a un bloque) y luego desaparecen en puntos bien definidos también (al salir del bloque).

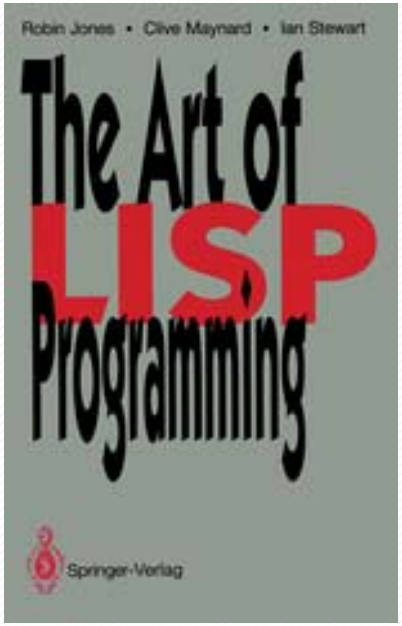
# Argumentos Funcionales

- En LISP, existe una función "**mapcar**" que nos permite aplicar una función a cada elemento de una lista.

- Esta función nos regresa una lista de los resultados:

(**mapcar** 'add1 '(1 5 3 7))
(2 6 4 8)

# Argumentos Funcionales




- Esta es una aplicación del **Principio de Abstracción** y, de hecho, en LISP es posible también definir una función “mapcar” que tome un número variable de argumentos, de forma que trabaje con cualquier función.

# Argumentos Funcionales

- Usando "**mapcar**" es posible escribir meta-funciones similares:

(**filter** 'minusp '(7 -3 8 -1 0))
(-3 -1)

(**reduce** 'plus 0 '(1 2 3 4 5))
15

# Argumentos Funcionales

- Recordemos que las funciones son ciudadanos de primera clase en LISP.
- Una de las ventajas de los argumentos funcionales es que suprimen los detalles de las estructuras de control y de la recursividad.
- Además, también simplifican la combinación de programas ya implementados.

# Argumentos Funcionales

- Por ejemplo, podemos escribir una función que calcule el producto punto de dos listas:

$$u.v = \sum_{i=1}^{n} u_i v_i$$

# Argumentos Funcionales

(**define** ip (u v)
  (**reduce** ‘plus 0 (**mapcar** ‘* u v)))

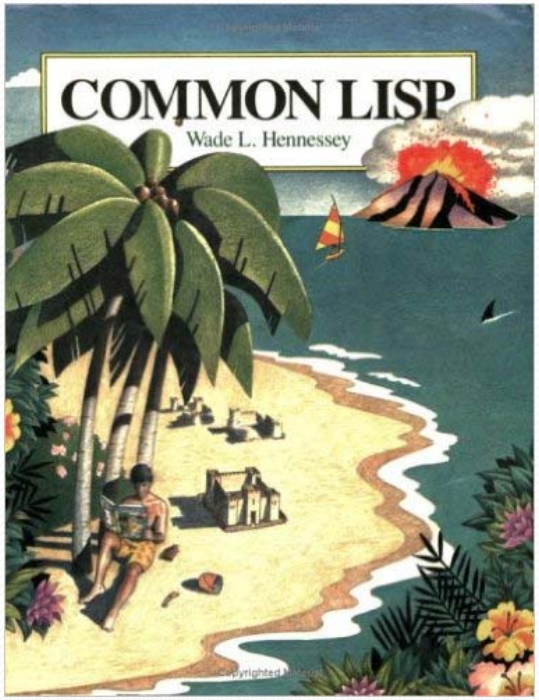
(ip ‘(1 2 3) ‘(-2 3 4)) ==> -2 + 6 + 12 = 16

**define** se usa para agregar nuevas definiciones al intérprete de LISP.

# Programas vs. Listas

- Un programa en LISP es algo que puede ser evaluado, algo que tiene significado.
- Una lista en LISP es meramente una colección de átomos en algún formato en particular.

# Programas vs. Listas




- Algunas listas en LISP pueden verse como programas y pueden ser evaluadas como tales, pero eso no significa que conceptualmente sean la misma cosa.

# Programas vs. Listas

- En términos más generales, un “programa” en LISP es meramente una serie de aplicaciones de funciones.
- Los programas pueden representarse como listas, lo cual proporciona enorme flexibilidad al lenguaje.

# Representación de listas

- El hecho de que los programas en LISP se representen como listas tiene varias ventajas. En particular, hace relativamente simple escribir programas en LISP que lean, pre-procesen, transformen y generen programas en LISP.

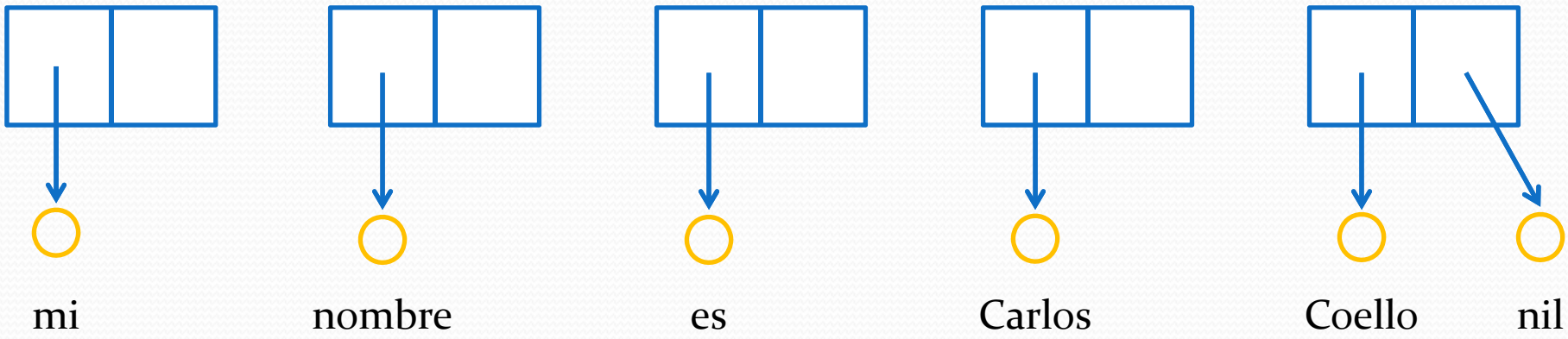
- Esto simplifica mucho el poder escribir un intérprete de LISP en LISP, que fue la motivación original para adoptar esta representación de programas mediante listas.

# Representación de listas

- Más importante aún es el hecho de que el uso de listas para representar programas ha permitido a los programadores en LISP escribir ayudas de programación de alto nivel y herramientas de transformación, en LISP, de manera fácil y conveniente.

- Como consecuencia, existen diversos ambientes avanzados de programación que se han escrito en LISP.

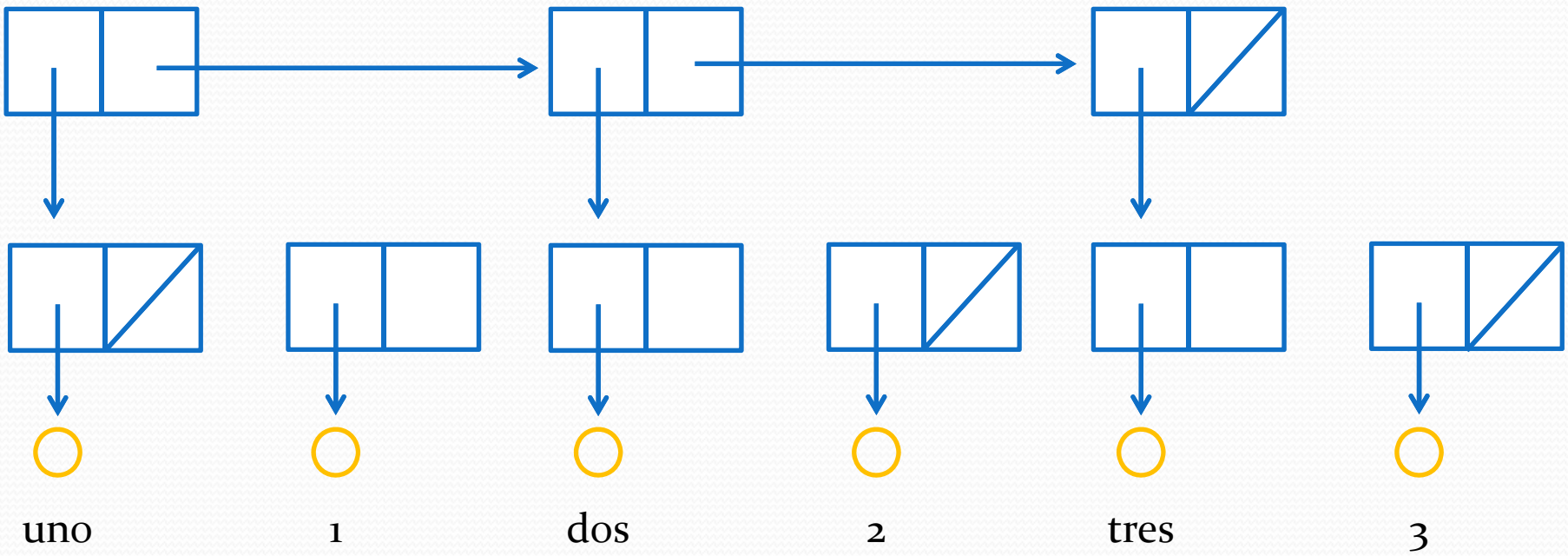
# Representación de listas




(mi nombre es Carlos Coello)

Las listas en LISP se representan usando listas ligadas.

# Representación de listas



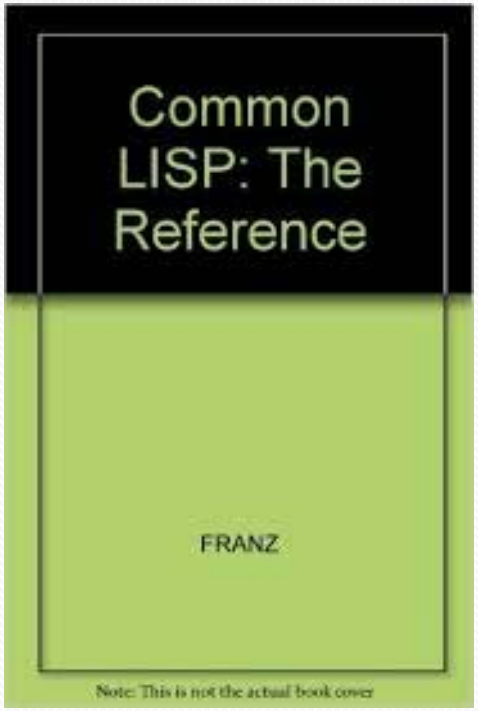

( (uno 1)  (dos 2)  (tres 3) )

# Representación de listas

- Las primitivas de listas son muy simples y eficientes.
- Suponiendo que L apunta al inicio de una lista, el "**car**" está dado por (usando notación de Pascal):

```
A:=L^.left;
```

# Representación de listas




- Así mismo, el "**cdr**" está dado por:

D:=L^.right;

# Representación de listas

Los términos “CAR” y “CDR” son abreviaciones que se utilizaban en la IBM 704 (donde se implementó la versión original de LISP):

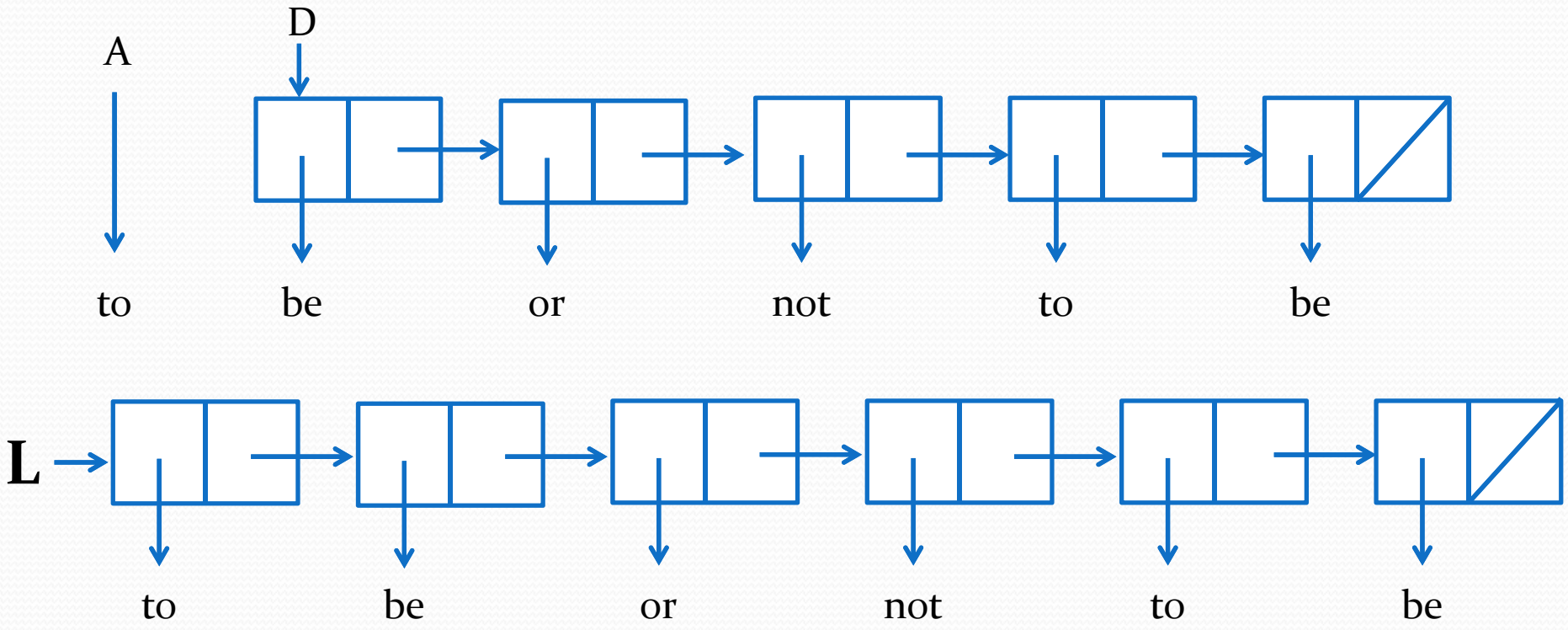
- CAR = *Contents of the Address Register*
- CDR = *Contents of the Decrement Register*

# Representación de listas

- Puesto que "**cons**" es el inverso de "**car**" y "**cdr**", debe resultar claro cómo se implementa usando listas ligadas:

(ver acetato siguiente)

# Representación de Listas



# Representación de listas

- El resultado de hacer el "**cons**" de dos listas A y D debe ser un apuntador a una celda cuyo lado izquierdo sea A y cuyo lado derecho sea D.

- En otras palabras, todo lo que tenemos que hacer es poner A y D en las mitades izquierda y derecha de una celda.

# Representación de listas

- La operación "**cons**" requiere un paso de asignación de almacenamiento.
- Debe obtenerse una celda nueva del "heap" (o sea, del área de memoria disponible).

# Representación de listas

- En Pascal se hace "**cons**" usando:

```
new(L);
L^.left :=A;
L^.right:=D;
```

# Funciones Puras

- El “aliasing” no suele ser un problema preocupante en LISP a pesar de que las sublistas pueden compartirse.
- La razón es que “**car**”, “**cdr**” y “**cons**” son funciones puras.
- Esto es, no tienen efectos colaterales sobre las listas que ya fueron creadas previamente.

# Funciones Puras

- "**cdr**" no borra un elemento de una lista, pues la lista original se mantiene intacta tras la aplicación de esta función.

- Lo mismo ocurre con "**cons**", que no agrega elementos a una lista sino que genera una nueva lista con los argumentos que se le pasan.

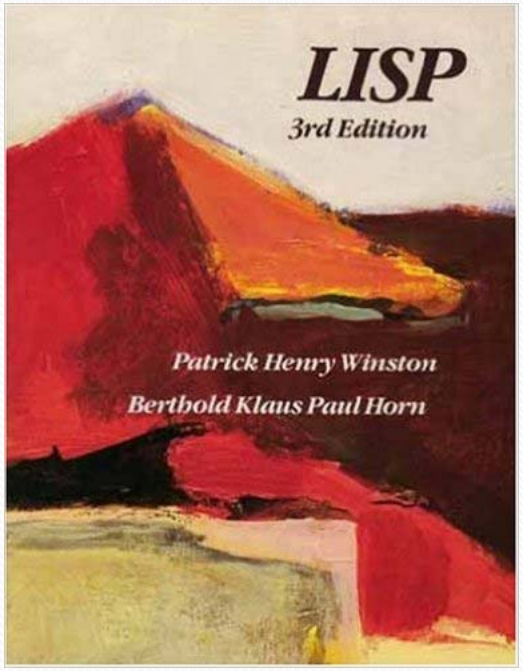
# Funciones Puras

- El resultado de esto es que la compartición de sublistas es invisible al usuario.
- Esta compartición incrementa la eficiencia del programa sin cambiar su significado.
- Sin embargo, en algunos casos sí es posible modificar una lista pasada como argumento y para ello existen los denominados “mutadores”.

# Funciones Puras

- LISP cuenta con algunas pocas funciones imperativas que pueden crear efectos colaterales.
- Por ejemplo:
- **rplaca**: se usa para alterar la mitad izquierda de una celda.
- **rplacd**: se usa para alterar la mitad derecha de una celda.

# Funciones Puras




- Este tipo de funciones imperativas deben usarse con mucho cuidado porque el resultado de su aplicación puede ser impredecible (p.ej., podrían cambiarse listas que aparentemente no guardan relación con la operación realizada).

# Listas Asociativas

- Una lista asociativa permite almacenar información en listas de listas que hacen las veces de campos de una base de datos.

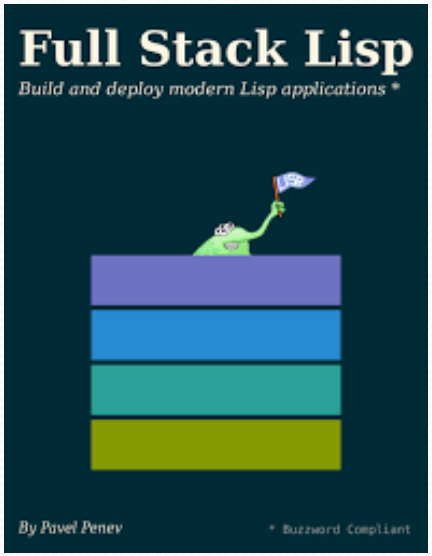
- En otras palabras, una lista asociativa es el equivalente de un "**record**" en un lenguaje procedural.

# Listas Asociativas

- Ejemplo:

(nombre (Carlos Coello))
(categoria (CINVESTAV 3-F))
(fecha-contrato (1 enero 2001)))

# Listas Asociativas




- El orden de la información en una lista asociativa es irrelevante.

- Se puede acceder a la información de manera asociativa usando "**assoc**".

# Listas Asociativas

- Por ejemplo:

(set ‘DS ‘((nombre (Carlos Coello)) (categoria (CINVESTAV 3-F) (fecha-contrato (1 enero 2001)))

(assoc ‘fecha-contrato DS)

(1 enero 2001)

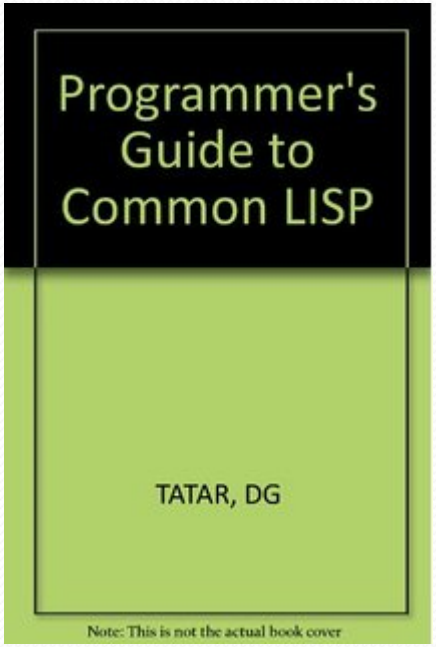
# Asociaciones Locales

(**let** (($n_1$ $e_1$) . . . ($n_m$ $e_m$)) E)

'Let' se usa para efectuar asociaciones locales (variables locales).

En Scheme, '**let***' se usa para realizar asociaciones locales que requieren recursividad.

# Common LISP



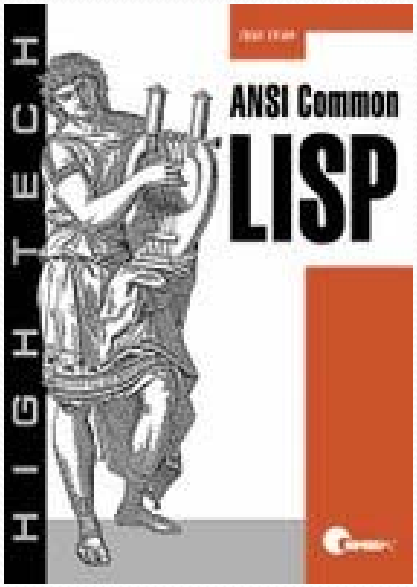

- En un intento por tener las ventajas de las reglas de entorno estático y mantener compatibilidad con versiones más viejas de LISP, el *Common LISP* adoptó una combinación de las reglas de entorno dinámico y de las reglas de entorno estático.

# Common LISP

- Por ejemplo, los parámetros de una función y las variables (establecidas mediante "**set**" o mediante "**let**") normalmente operan con reglas de entorno estático.

- Sin embargo, es posible dotar a las variables de reglas de entorno dinámico si se declaran usando "**special**".

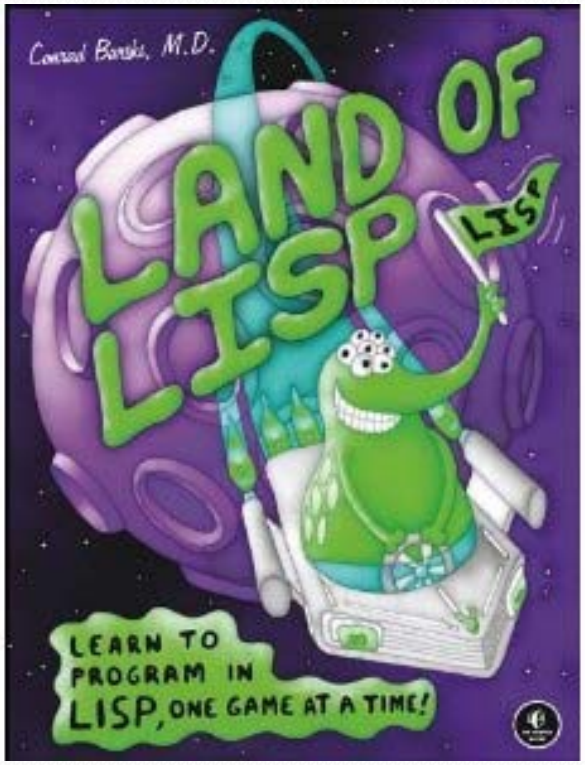
# Common LISP




- Las funciones establecidas mediante "**defun**" son globales.

- Por lo tanto, da igual si se usan con ellas reglas de entorno estático o reglas de entorno dinámico.

# Common LISP



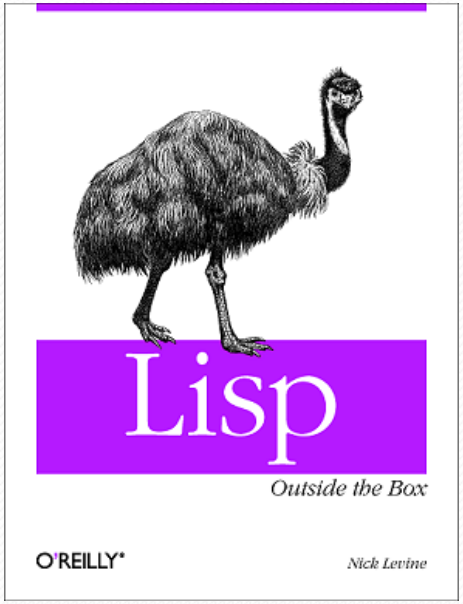

- Otros constructores de LISP, tales como "**catch**" (usado para manejar excepciones) tienen también reglas de entorno dinámico.

# Borrado Explícito

- Cuando se usa la asignación de memoria dinámica, el programador debe liberar una cierta celda, moviéndola al área de almacenamiento disponible, cuando ésta ya no se requiera.

- Por ejemplo, en Pascal, se usa '**dispose**' para esta tarea. Si hacemos '**dispose**(p)', la celda a la que apunta 'p' se regresa al área de almacenamiento disponible.

# Borrado Explícito




- Esto normalmente se logra ligando la celda a una free-list, donde el asignador de almacenamiento puede encontrarla, a fin de satisfacer alguna solicitud posterior de asignación de almacenamiento (‘new’).

# Problemas con el Borrado Explícito

- Requiere que los programadores trabajen más duro. Deben llevar un seguimiento cuidadoso de cada celda, así como de todas las listas en las que ésta participa (las listas pueden ser compartidas).

- También deben determinar el momento en que una celda puede ser liberada.

# Problemas con el Borrado Explícito

- Obviamente, es mucho mejor hacer que una computadora se haga cargo de los detalles del manejo de almacenamiento, ya que de esa forma los programadores pueden concentrarse en cosas mucho más importantes.

- El borrado explícito viola la seguridad del sistema de programación. El problema es que podríamos regresar al almacenamiento disponible una celda que sigue siendo referenciada por varias otras listas.

# Problemas con el Borrado Explícito

- Las listas que hagan referencia a esta celda tendrán "apuntadores colgados" (*dangling pointers*), o sea, apuntadores que no apuntan a una celda asignada.

- Las referencias colgadas (*dangling references*) producen errores misteriosos e impredecibles.

# Soluciones al Problema de los Apuntadores Colgados

- **Tombstones** (lápidas): Esta solución fue propuesta por Lomet en 1975. La idea es hacer que todas las variables dinámicas incluyan una celda especial, llamada "lápida" (*tombstone*) que es en sí misma un apuntador a la variable dinámica.

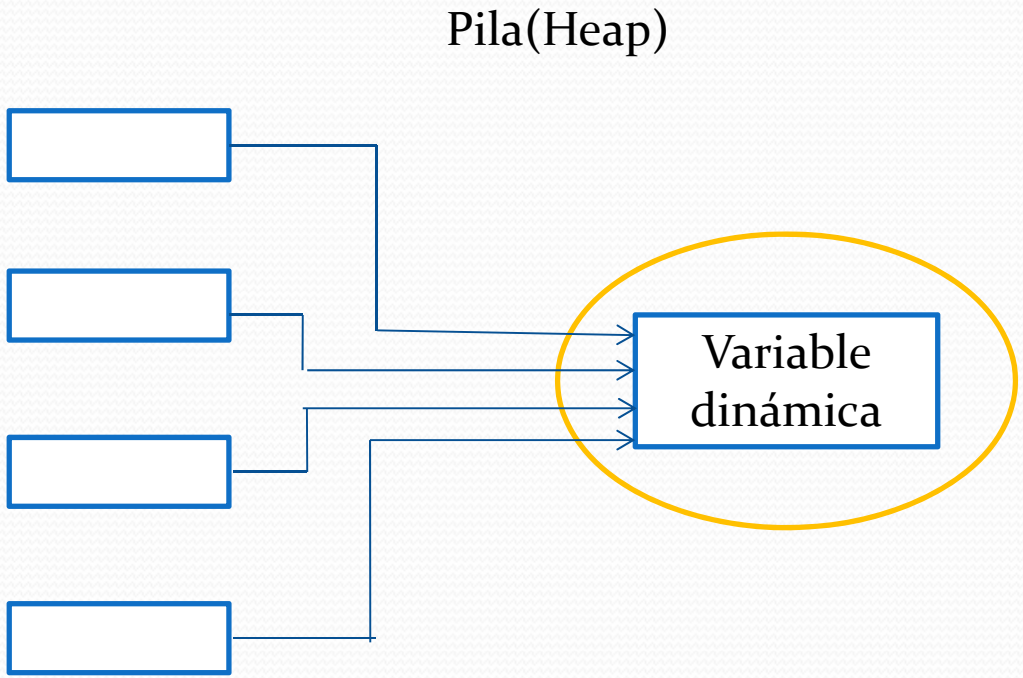
- El apuntador realmente apunta sólo a las lápidas y nunca a las variables dinámicas. Cuando una variable dinámica es liberada, la lápida permanece pero se pone en NIL, indicando que la variable dinámica ya no existe.

# Soluciones al Problema de los Apuntadores Colgados

- Cualquier referencia a un apuntador que apunte a una lápida puesta en NIL, puede detectarse como un error. Esto evita que un apuntador pueda apuntar a una variable que ha sido liberada de memoria.

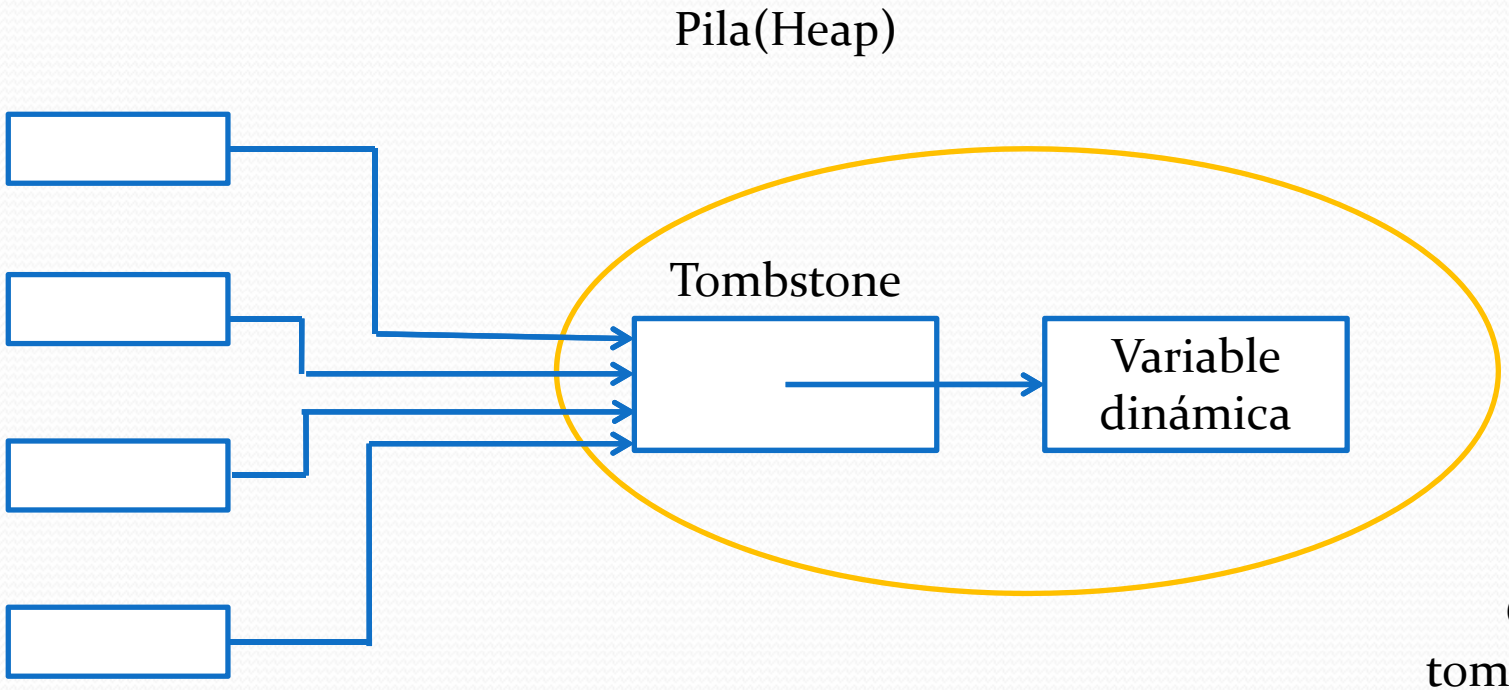
- En los 2 acetatos siguientes veremos el uso de apuntadores sin lápidas (tombstones) y con ellas.

# Soluciones al Problema de los Apuntadores Colgados




Sin tombstones

# Soluciones al Problema de los Apuntadores Colgados



# Soluciones al Problema de los Apuntadores Colgados

- Las lápidas son una solución costosa tanto en tiempo como en espacio.
- Como las lápidas nunca son liberadas, su almacenamiento nunca es reclamado.
- Cada acceso a una variable dinámica a través de una lápida, requiere de un nivel adicional de indirección, lo cual requiere un ciclo adicional de CPU en la mayor parte de las computadoras.

# Soluciones al Problema de los Apuntadores Colgados

- Ningún lenguaje de programación de uso extendido utiliza lápidas. Sin embargo, el sistema de software de la Macintosh utiliza lápidas para detectar apuntadores colgantes y para facilitar la re-asignación de objetos asignados dinámicamente.

# Soluciones al Problema de los Apuntadores Colgados

- **Locks and Keys** (Cerraduras y Llaves): Se usaron en la implementación del UW-Pascal. En este compilador, los valores de los apuntadores se representan como pares ordenados (llave, dirección), donde la llave es un valor entero.

- Las variables dinámicas se representaban como el almacenamiento para la variable más una celda adicional que almacenaba un valor entero que hacía las veces de la llave.

# Soluciones al Problema de los Apuntadores Colgados

- Cuando se asigna una variable dinámica, se crea un valor para su cerradura y se le coloca en la celda disponible para ese propósito, así como en la celda de la llave del apuntador correspondiente.

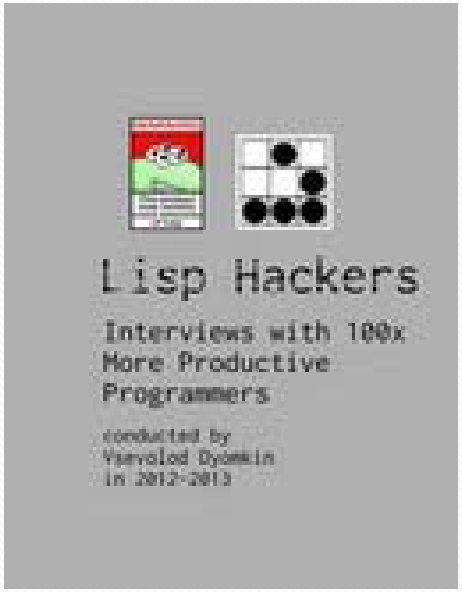
- Cada acceso al apuntador desasignado compara el valor de la llave del apuntador con el de su cerradura en la variable dinámica. Si corresponden, el acceso es legal; de lo contrario, el acceso se trata como un error en tiempo de ejecución.

# Soluciones al Problema de los Apuntadores Colgados

- Cualquier copia del apuntador a otros apuntadores debe incluir el valor de la llave. Por lo tanto, se permite que cualquier número de apuntadores puedan referenciar una variable dinámica dada.

- Cuando una variable dinámica es liberada con "**dispose**", el valor de su cerradura se inicializa a un valor ilegal.

# Soluciones al Problema de los Apuntadores Colgados




- Posteriormente, si un apuntador diferente del especificado en el "**dispose**" es liberado, el valor de su dirección permanecerá intacto, pero su llave no corresponderá con su cerradura, por lo cual el acceso a la variable no se permitirá.

# PROLOG (Antecedentes)

- En términos llanos, la programación lógica se refiere al uso de una notación basada en una lógica formal para comunicar algún proceso a una computadora.

- El cálculo de predicados es la notación usada en los lenguajes que pertenecen al paradigma denominado “programación lógica”.