

LISP

- Contenido
 - Programación funcional y recursiva
 - Ejemplos
 - Organización de memoria
 - Funciones anónimas
 - Variables. Alcances

1

Programación recursiva frente a iterativa

- Programación recursiva
 - » Implementación intuitiva
 - » La traducción de la solución recursiva de un problema (caso base y caso recursivo) a código LISP es prácticamente inmediata
 - » Útil cuando hay varios niveles de anidamiento
 - La solución para un nivel es válida para el resto.
 - » La interpretación y comprensión del código puede ser compleja.
 - Es importante llegar a un compromiso entre la claridad de la programación y la eficiencia en la misma.
- Programación iterativa
 - » Con mapcar
 - » Con bucles (dolist, dotimes, etc.)
 - Desaconsejado

2

Ejemplos de recursividad, (1)

- Contar los átomos de cualquier expresión LISP:

```
> (defun cuenta-atomos (expr)
  (cond ((null expr) 0)
        ((atom expr) 1)
        (t (+ (cuenta-atomos (first expr))
              (cuenta-atomos (rest expr))))))

> (cuenta-atomos '(a (b c) ((d e) f)))
6
```

3

Ejemplos de recursividad, (2)

- Número de sublistas de una lista

```
>(defun sublistas (expresion)
  (cond ((or (null expresion) (atom expresion))
        0)
        (t (+ (if (atom (first expresion)) 0 1)
              (sublistas (first expresion))
              (sublistas (rest expresion))))))

> (sublistas '(a (b c) ((d e) f))) >>>
3
```

4

Ejemplos de recursividad, (3)

- **Producto escalar:**
 - » (producto '(2 3) '(4 5)) >>> 23
 - » $2 \times 4 + 3 \times 5 = 23$
- **Versiones válidas:**
 - » **Versión recursiva:**

```
> (defun producto (vector1 vector2)
  (if (or (null vector1) (null vector2))
      0
      (+ (* (first vector1) (first vector2))
         (producto (rest vector1) (rest vector2)))))
```
 - » **Versión con mapcar**

```
>(defun producto (vector1 vector2)
  (apply #'+ (mapcar #'* vector1 vector2)))
```

5

Ejemplos de recursividad, (4)

- **Versión iterativa no recomendada del producto escalar**

```
> (defun producto (vector1 vector2)
  (let ((suma 0))
    (dotimes (i (length vector1))
      (setf suma
              (+ suma (* (nth i vector1)
                          (nth i vector2))))))
  suma))
```

6

Ejemplos de recursividad, (5)

- Profundidad máxima de listas anidadas

```
> (defun profundidad-maxima (expresion)
  (cond ((null expresion) 0)
        ((atom expresion) 1)
        (t (+ 1
              (apply #'max
                     (mapcar #'profundidad-maxima
                             expresion))))))

>>> PROFUNDIDAD-MAXIMA
> (profundidad-maxima '(1))
>>>> 2
> (profundidad-maxima '((1 (2 (3))) 4))
>>>> 5
```

7

Ejemplos de recursividad, (6)

- Ejemplo del cálculo de la potencia de un número optimizada (poco amigable)

```
> (defun potencia (x n)
  ;; "Optimizacion calculo potencia"
  (cond ((= n 0) 1)
        ((evenp n) (expt (potencia x (/ n 2)) 2)
          )
        (t (* x (potencia x (- n 1))))))

> (potencia 2 3)
8
```

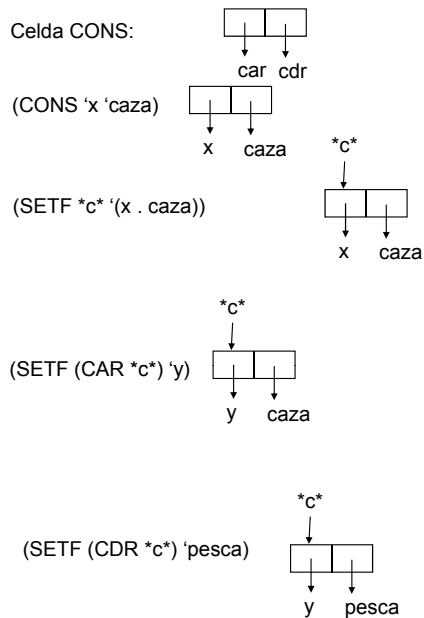
8

Organización de la memoria (1)

- Celdas CONS
 - » Formadas por 2 punteros: CAR y CDR
 - » La función CONS crea una celda CONS a partir de memoria libre
 - > (CONS arg1 arg2)
 - arg1: inicializa la celda CAR
 - arg2: inicializa la celda CDR
 - > (CONS 'x 'caza)
 - (x . caza)
- La notación par-punto es la utilizada por LISP para indicar los elementos apuntados por el CAR y el CDR de una celda CONS. Cuando CDR apunta a NIL, LISP simplifica esta notación no mostrando dicho elemento apuntado.
- MODIFICACIÓN DE CELDAS CONS
 - » Con SETF
 - arg1: dirección de puntero en memoria
 - arg2: la nueva dirección que deseamos que contenga arg1
 - > (SETF *c* '(x . caza))
 - (x . caza)
 - > (SETF (CAR *c*) 'y)
 - (y . caza)
 - > (SETF (CDR *c*) 'pesca)
 - (y . pesca)
 - > *c*
 - (y . pesca)

9

Organización de la memoria (2)



Organización de la memoria (3)

- Disgregación de celdas CONS
 - » Asignando NIL al CDR de la/s celda/s que deseamos disgregar
 - » Creando nuevas celdas CONS con SETF

```
>(SETF *d* `(A B C))
(A B C)
>(SETF *d2* (CDR *d*))
(B C)
>*d*
(A B C)
>(SETF (CDR *d*) nil)
nil
>(SETF *d3* (CDR *d2*))
(C)
>(SETF (CDR *d2*) nil)
nil
```

- cada vez que el CDR de una celda se apunta a NIL, el resto de celdas CONS que estaban apuntadas por dicha celda quedan ocupando memoria y no se pueden reutilizar.
 - » Tratar de optimizar las asignación de memoria evitando el uso abusivo de variables globales
 - » Recuperar la memoria usada e inservible (garbage collection)

11

Organización de la memoria (4)

- Garbage Collection
 - » Analizamos el siguiente ejemplo:

```
>(SETF *e* `(dato1 dato2 dato3))
(dato1 dato2 dato3)
>(SETF *e* `(dato4 dato5 dato6))
(dato4 dato5 dato6)
>*e*
(dato4 dato5 dato6)
```
- La memoria utilizada en la primera asignación de SETF (una serie de datos) queda ligada a la lista inicial y no es accesible
- Se denomina *basura* (*garbage*) a los objetos usados que ocupan memoria y no pueden volver a ser utilizados.
En el caso anterior, son basura:

```
(dato1 dato2 dato3)
```

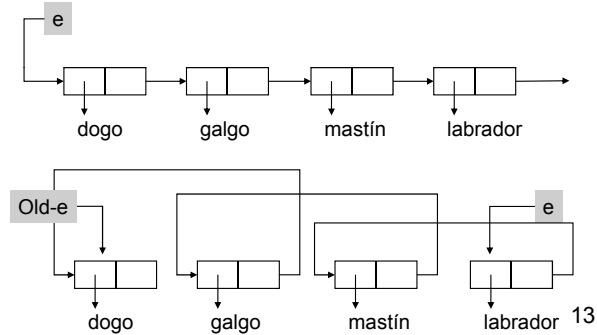
12

Operaciones destructivas

- Consideremos el siguiente ejemplo

```
>(SETF e `(dogo galgo mastín labrador))
(dogo galgo mastín labrador)
>(SETF old-e e)
(dogo galgo mastín labrador)
>(SETF e (NREVERSE old-e))
(labrador mastín galgo dogo)
>e
(labrador mastín galgo dogo)
>old-e
(dogo)
```

- NREVERSE destruye la lista `old-e`, reasignando punteros con el fin de no generar basura. Al concluir, ambas listas comparten la última celda



Funciones anónimas

- Orígenes

- » Deriva de la notación usada por Whitehead y Russell de los *Principia Mathematica*

- » Aplicada por vez primera por Alonzo Church en 1941 en su definición del *cálculo lambda* del que deriva LISP.

- $\hat{x}(x + x)$ (evolución de x -circunflejo)

- $\Lambda x(x + x)$ (evolución de $\hat{}$ a Λ)

- $\lambda x(x + x)$ (evolución de Λ a λ)

- $(\text{lambda } (x) (+ x x))$ (McCarthy 1958)

- Utilización

- » Mediante la notación $\#'$

- » `(funcall #'(lambda (x) (+ x x)))`

Ejemplos de aplicación

- **Problema:** "Dada la lista (1 2 3), obtener sus cuadrados"

- **Solución 1**

```
> (defun n-square (n) (expt n 2))
> (defun square-iterate (list)
  (if (null list)
      nil
      (cons (n-square (car list))
            (square-iterate (cdr list)))))
```

```
> (square-iterate '(1 2 3))
(1 4 9)
```

- **Solución 2**

```
> (mapcar #'(lambda (x)
             (expt x 2)) '(1 2 3))
```

```
(1 4 9)
```

- **Solución 3**

```
> (mapcar #'n-square '(1 2 3))
```

```
(1 4 9)
```

15

Evaluación

- **Mediante**

- » **Eval:** en desuso

```
>(eval '(+ 1 2 3 4))
10
```

- » **Funcall:** se aplica sobre una lista de argumentos simples

```
>(funcall #' + 1 2 3 4)
10
```

- » **Apply:** se aplica sobre una lista de uno o más argumentos el último de los cuales debe ser una lista

```
>(apply #' + 1 2 '(3 4))
10
```

```
>(apply #' + '(1 2 3 4))
10
```

16

Alcance léxico (*lexical scope*)

- Este término hace referencia al conjunto de reglas necesarias para determinar, sin ambigüedad, la ligadura asociada a las variables utilizadas en un fragmento de código.

- Un símbolo hace referencia a la variable que posee dicho nombre en el contexto en el que dicho símbolo aparece

```
> (let ((x 10)) (defun foo () x))
> (let ((x 20)) (foo))
10
```

- Cada vez que se efectúa una ligadura entre variables se determina un *cierre léxico* para la misma. Dicha variable es *visible* dentro del cierre léxico que le corresponda.

```
> (defun main (z)
  (ejemplo-de-alcance z 3))

> (defun ejemplo-de-alcance (x y)
  (append (let ((x (car x)))
            (list x y)))
          (list x z)))
```

z es léxicamente invisible

17

Tipos de variables

- A la vista de los alcances léxico y dinámico comentados anteriormente diferenciamos los siguientes tipos de variables:

- Locales

- » Su alcance es el del contexto en que se definen
- » implícitamente poseen alcance léxico

- Globales

- » Se definen con `setf` (usar `defparameter` en archivos de código fuente)
- » Son visibles en cualquier lugar
- » Por convenio se denotan entre asteriscos

- Libres

- » Se denomina a aquellas variables que se definen fuera de la función que las referencia (x en el ejemplo)

```
> (setf fn (let ((i 3)
                #'(lambda (x) (+ x i))))
> (funcall fn 2)
5
```

- Se pueden declarar (`declare` para local, `declare` para global), tipos (`type`) de variables con propósitos de eficiencia de compilación (ver Graham 13.3).

18