Mutaciones y programación imperativa

José A. Alonso y María J. Hidalgo

Ciencias de la Computación e Inteligencia Artificial

Universidad de Sevilla

Asignaciones y estados

- **El procedimiento `set!`**

```
(define a 3)                          =>  #<unspecified>
a                                     =>  3
(set! a 5)                            =>  #<unspecified>
a                                     =>  5
(define f (lambda (x) (+ a x)))       =>  #<unspecified>
(f 2)                                 =>  7
(set! f (lambda (x) (* a x)))         =>  #<unspecified>
(f 2)                                 =>  10
```

- **Conceptos:**
  - **Variables de estados**
  - **Estado**
  - **Programa imperativo**

Implementación del tipo de dato "pila"

- **Procedimientos básicos de "pila":**

```
(define pila ())     ; pila es la variable de estado

(define pila-vacia?
  (lambda ()
    (null? pila)))

(define apila!
  (lambda (a)
    (set! pila (cons a pila))))

(define desapila!
  (lambda ()
    (if (pila-vacia?)
        (error "desapila!: La pila está vacía.")
        (set! pila (cdr pila)))))
```

Implementación del tipo de dato "pila"

```
(define cima
  (lambda ()
    (if (pila-vacia?)
        (error "cima: la pila está vacía.")
        (car pila))))

(define escribe-pila
  (lambda ()
    (display "CIMA: ")
    (for-each (lambda (x) (display x) (display " "))
              pila)
    (newline)))
```

Implementación del tipo de dato "pila"

- **Cálculo con pilas:**

```
(escribe-pila)  =>  CIMA: #<unspecified>
(apila! 'a)     =>  #<unspecified>
(apila! 'b)     =>  #<unspecified>
(apila! 'c)     =>  #<unspecified>
(escribe-pila)  =>  CIMA: c b a #<unspecified>
(cima)          =>  c
(escribe-pila)  =>  CIMA: c b a #<unspecified>
(desapila!)     =>  #<unspecified>
(escribe-pila)  =>  CIMA: b a #<unspecified>
(pila-vacia?)   =>  #f
(desapila!)     =>  #<unspecified>
(desapila!)     =>  #<unspecified>
(pila-vacia?)   =>  #t
(cima)          =>  ERROR: cima: la pila está vacía.
(desapila!)     =>  ERROR: desapila!: La pila está vacía.
```

Recursión profunda en estilo imperativo

```
;;; (cuenta-pcn '((-3 0) 1 (9) -2 (6 4)))  =>  (4 1 2)
(define cuenta-pcn
  (lambda (l)
    (let ((p 0) (c 0) (n 0))   ;p, c y n son las variables de estado
      (letrec
          ((bucle
            (lambda (ls)
              (if (not (null? ls))
                  (let ((x (car ls)))
                    (cond ((list? x) (bucle x))
                          ((positive? x) (set! p (+ p 1)))
                          ((zero? x) (set! c (+ c 1)))
                          (else (set! n (+ n 1))))
                    (bucle (cdr ls)))))))
        (bucle l))
      (list p c n))))
```

Recursión profunda en estilo imperativo

```
;;; (cuenta-pcn '((-3 0) 1 (9) -2 (6 4)))  =>  (4 1 2)
(define cuenta-pcn
  (lambda (l)
    (let ((p 0) (c 0) (n 0))   ;p, c y n son las variables de estado
      (let bucle ((ls l))
        (if (not (null? ls))
            (let ((x (car ls)))
              (cond ((list? x) (bucle x))
                    ((positive? x) (set! p (+ p 1)))
                    ((zero? x) (set! c (+ c 1)))
                    (else (set! n (+ n 1))))
              (bucle (cdr ls)))))
      (list p c n))))
```

Recursión profunda en estilo imperativo

```
;;; (cuenta-pcn '((-3 0) 1 (9) -2 (6 4)))  =>  (4 1 2)
(define cuenta-pcn
  (lambda (l)
    (let ((p 0) (c 0) (n 0))   ;p, c y n son las variables de estado
      (do ((aux l (cdr aux)))
          ((null? aux) (list p c n))
        (let ((x (car aux)))
          (cond ((list? x)
                 (let ((y (cuenta-pcn x)))
                   (set! p (+ p (car y)))
                   (set! c (+ c (cadr y)))
                   (set! n (+ n (caddr y)))))
                ((positive? x) (set! p (+ p 1)))
                ((zero? x) (set! c (+ c 1)))
                (else (set! n (+ n 1)))))))))
```

Comparación de procedimientos

```
(define f
  (let ((a 0))
    (lambda (x)
      (set! a (+ a x))
      a)))

;;; (f 10)  =>  10
;;; (f 10)  =>  20
;;; (f 20)  =>  40

(define g
  (lambda (x)
    (let ((a 0))
      (set! a (+ a x))
      a)))

;;; (g 10)  =>  10
;;; (g 10)  =>  10
;;; (g 20)  =>  20
```

Memorización

- **Ejemplo de procesamiento de una tabla**

```
> (define tabla '((1 1) (2 4) (3 9)))
#<unspecified>
> (procesa 2 tabla
           (lambda (x) (+ 10 (cadr x)))
           (lambda () 'fallo))
14
> (procesa 5 tabla
           (lambda (x) (+ 10 (cadr x)))
           (lambda () 'fallo))
fallo
```

Memorización

- **El procedimiento** `procesa`

```
(define procesa
  (lambda (elto tabla proc-exito proc-fracaso)
    (letrec
        ((aux
          (lambda (l)
            (cond ((null? l) (proc-fracaso))
                  ((equal? (caar l) elto) (proc-exito (car l)))
                  (else (aux (cdr l)))))))
      (aux tabla))))
```

- **Redefinición de** `assoc`

```
;;; (assoc 2 '((1 1) (2 4) (3 6)))  =>  (2 4)
;;; (assoc 5 '((1 1) (2 4) (3 6)))  =>  #f
(define n-assoc
  (lambda (elto tabla)
    (procesa elto
             tabla
             (lambda (x) x)
             (lambda () #f))))
```

Memorización

- **Procedimiento de memorización**

```
(define memoriza
  (lambda (proc)
    (let ((tabla '()))
      (lambda (arg)
        (procesa arg tabla
                 (lambda (par) (cdr par))
                 (lambda ()
                   (let ((val (proc arg)))
                     (set! tabla (cons (cons arg val) tabla))
                     val)))))))
```

Memorización

- **Aplicación de la memorización a la sucesión de Fibonacci**

```
(define fib
  (lambda (n)
    (if (< n 2)
        n
        (+ (fib (- n 1)) (fib (- n 2))))))

(define fib-con-memoria
  (memoriza fib))

> (fib 20)
;Evaluation took 690 mSec (190 in gc) 87570 cells work, 33 bytes other
6765
> (fib-con-memoria 20)
;Evaluation took 670 mSec (150 in gc) 87607 cells work, 33 bytes other
6765
> (fib-con-memoria 20)
;Evaluation took 0 mSec (0 in gc) 29 cells work, 33 bytes other
6765
```

Memorización

```
(define fib-memorizado
  (memoriza (lambda (n)
              (if (< n 2)
                  n
                  (+ (fib-memorizado (- n 1))
                     (fib-memorizado (- n 2)))))))
```

```
> (fib-memorizado 20)
;Evaluation took 10 mSec (0 in gc) 1208 cells work, 33 bytes other
6765
> (fib-memorizado 20)
;Evaluation took 0 mSec (0 in gc) 29 cells work, 33 bytes other
6765
> (fib-memorizado 30)
;Evaluation took 10 mSec (0 in gc) 1479 cells work, 33 bytes other
832040
```

Definición imperativa de member?

```
;;; (n-member? 'b '(a b c))  =>  #t
;;; (n-member? 'd '(a b c))  =>  #f
(define n-member?
  (lambda (x l)
    (letrec ((ir-a (lambda (etiqueta) (etiqueta)))
             (comienzo (lambda ()
                         (cond ((null? l) #f)
                               ((equal? (car l) x) #t)
                               (else (ir-a reducir)))))
             (reducir (lambda ()
                        (set! l (cdr l))
                        (ir-a comienzo))))
      (ir-a comienzo))))
```

El procedimiento `mientras-que`

- **El procedimiento** `mientras-que`

```
;;; > (let ((i 0))
;;;     (mientras-que (lambda () (< i 10))
;;;                   (lambda () (display i) (set! i (+ i 1))))
;;;     (newline))
;;; 0123456789
;;; #<unspecified>
(define mientras-que
  (lambda (pred cuerpo)
    (let bucle ()
      (cond ((pred) (cuerpo) (bucle))))))
```

El procedimiento `mientras-que`

- **Redefinición de `member?` con `mientras-que`**

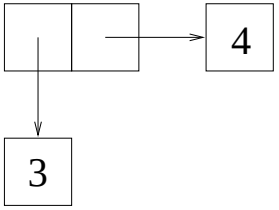
```
;;; (n-member? 'b '(a b c))  =>  #t
;;; (n-member? 'd '(a b c))  =>  #f
(define n-member?
  (lambda (x l)
    (let ((resp #f))
      (mientras-que
       (lambda () (not (or (null? l) resp)))
       (lambda ()
         (if (equal? (car l) x)
             (set! resp #t)
             (set! l (cdr l)))))
      resp)))
```

El procedimiento `mientras-que`

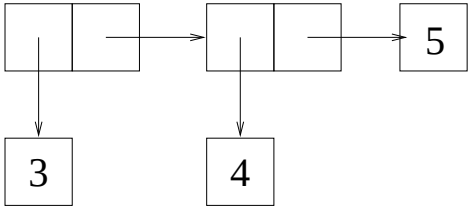
- **Redefinición de `intercambia` con `mientras-que`**

```
;;; > (intercambia 1 2 '(1 2 3 2 1))
;;; > CIMA: 2 1 3 1 2
;;; #<unspecified>
(define intercambia
  (lambda (a b l)
    (let ((aux l)
          (resp '()))
      (mientras-que
       (lambda () (not (null? aux)))
       (lambda ()
         (cond ((equal? (car aux) a) (apila! b))
               ((equal? (car aux) b) (apila! a))
               (else (apila! (car aux))))
         (set! aux (cdr aux))))
      (escribe-pila))))
```

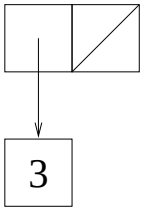
Diagramas caja–puntero



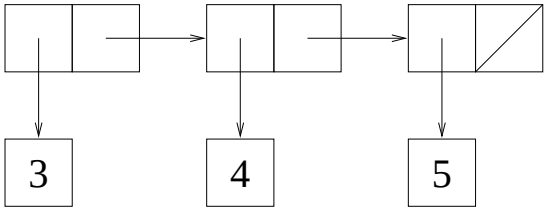
(cons 3 4)



(cons 3 (cons 4 5))



(cons 3 ())



(cons 3 (cons 4 (cons 5 ())))

Procedimientos set-car! y set-cdr!

- **Procedimientos** `set-car!` **y** `set-cdr!`

```
(define x '(1 2 3))  =>  #<unspecified>
x                    =>  (1 2 3)
(set-car! x 0)       =>  #<unspecified>
x                    =>  (0 2 3)
(set-cdr! x '(1))    =>  #<unspecified>
x                    =>  (0 1)
```

- **Procedimiento** `inserta!`

```
;;; (define l '(1 2 4 5 7))  =>  #<unspecified>
;;; l                        =>  (1 2 4 5 7)
;;; (inserta! 3 l)           =>  (1 2 3 4 5 7)
;;; l                        =>  (1 2 3 4 5 7)
(define inserta!
  (lambda (x l)
    (cond ((null? l) (list x))
          ((< x (car l)) (cons x l))
          (else (set-cdr! l (inserta! x (cdr l)))
                l))))
```

Procedimientos `set-car!` y `set-cdr!`

- **El procedimiento** `append!`

```
;;; (define x '(a b))    =>  #<unspecified>
;;; (define y '(c d e))  =>  #<unspecified>
;;; (n-append! x y)      =>  (a b c d e)
;;; x                    =>  (a b c d e)
;;; y                    =>  (c d e)
(define n-append!
  (lambda (x y)
    (cond ((null? x) y)
          ((null? (cdr x)) (set-cdr! x y))
          (else (n-append! (cdr x) y)
                x))))
```