Abstracción de procedimientos

José A. Alonso y María J. Hidalgo

Ciencias de la Computación e Inteligencia Artificial

Universidad de Sevilla

Procedimientos como argumentos y valores

```
;;; (suma1 '(2 4 5))  =>  (3 5 6)
(define suma1
  (lambda (l)
    (if (null? l)
        ()
        (cons (+ 1 (car l)) (suma1 (cdr l))))))

;;; (raiz-l '(4 9 16))  =>  (2.0 3.0 4.0)
(define raiz-l
  (lambda (l)
    (if (null? l)
        ()
        (cons (sqrt (car l)) (raiz-l (cdr l))))))

;;; (mult10 '(1 3 6))  =>  (10 30 60)
(define mult10
  (lambda (l)
    (if (null? l)
        ()
        (cons (* 10 (car l)) (mult10 (cdr l))))))
```

Procedimientos como argumentos y valores

- **Abstracción:** `n-map`

```
;;; (n-map sqrt '(4 9 16))                  =>  (2.0 3.0 4.0)
;;; (n-map (lambda (x) (+ 1 x)) '(2 4 5))  =>  (3 5 6)
(define n-map
  (lambda (proc l)
    (if (null? l)
        ()
        (cons (proc (car l))
              (n-map proc (cdr l))))))
```

Procedimientos como argumentos y valores

- **El procedimiento** `map`

```
(map sqrt '(4 9 16 25))                    =>  (2 3 4 5)
(map (lambda (n) (expt n n)) '(1 2 3))     =>  (1 4 27)
(map car '((a b) (1 2) (c d)))             =>  (a 1 c)
(map cadr '((a b) (1 2) (c d)))            =>  (b 2 d)

(map + '(1 1 1) '(2 4 5) '(0 7 2))         =>  (3 12 8)
(map * '(1 1 1) '(2 4 5 6) '(0 7))         =>  (0 28)
(map list '(a b c) '(1 2 3))               =>  ((a 1) (b 2) (c 3))
```

Procedimientos como argumentos y valores

- **El procedimiento `for-each`:**

```
> (for-each display '("Hola" " " "¿Como te llamas?" "\n"))
Hola ¿Como te llamas?
#<unspecified>
> (for-each (lambda (c) (display c) (newline)) '("ab" "cd"))
ab
cd
#<unspecified>
> (map (lambda (c) (display c) (newline)) '("ab" "cd"))
ab
cd
(#<unspecified> #<unspecified>)
```

Procedimientos como argumentos y valores

- **El procedimiento `apply`**

```
(apply + '(1 2 3))  =>  6
(apply max '(1 2))  =>  2
```

- **Procedimientos de aridad variable.**

```
;;; (suma 2 3 5)    =>  10
;;; (suma 2 3 5 7)  =>  17
(define suma
  (lambda l
    (apply + l)))
```

Procedimientos como argumentos y valores

```
;;; (cuenta-pares-1 '(1 2 4 6))  =>  3
(define cuenta-pares-1
  (lambda (l)
    (if (null? l)
        0
        (+ (if (even? (car l))
               1
               0)
           (cuenta-pares-1 (cdr l))))))

;;; (cuenta-pares-2 1 2 4 6)    =>  3
;;; (cuenta-pares-2 1 2 4 6 8)  =>  4
(define cuenta-pares-2
  (lambda args
    (cuenta-pares-1 args)))
```

Procedimientos como argumentos y valores

```
;;; (cuenta-pares-3 1 2 4 6)    =>  3
;;; (cuenta-pares-3 1 2 4 6 8)  =>  4
(define cuenta-pares-3
  (lambda args
    (if (null? args)
        0
        (+ (if (even? (car args))
               1
               0)
           (apply cuenta-pares-3 (cdr args))))))
```

Procedimientos como argumentos y valores

- **Composición de procedimientos**

```
;;; ((composicion sqrt abs) -4)          =>  2.0
;;; (define mas1 (lambda (x) (+ 1 x)))   =>  #<unspecified>
;;; (define h (composicion sqrt mas1))   =>  #<unspecified>
;;; (h 8)                                =>  3.0
;;; (define k (composicion mas1 sqrt))   =>  #<unspecified>
;;; (k 8)                                =>  3.82842712474619
(define composicion
   (lambda (f g)
     (lambda (x)
        (f (g x)))))
```

Parametrización

- **Parametrización de la suma**

```
;;; (suma 2 3)      =>  5
;;; ((suma-p 2) 3)  =>  5
;;; ((suma-p 4) 3)  =>  7

(define suma
  (lambda (m n)
    (+ m n)))

(define suma-p
   (lambda (n)
      (lambda (m)
         (suma m n))))
```

Parametrización

- **Parametrización del redondeo**

```
;;; (redondea 3.14159 3)  =>  3.142
(define redondea
  (lambda (x n)
    (let ((factor (expt 10 n)))
      (/ (round (* x factor)) factor))))

;;; ((redondea-p 3) 3.14159)  =>  3.142
;;; ((redondea-p 2) 3.14159)  =>  3.14
(define redondea-p
  (lambda (n)
    (lambda (x)
      (let ((factor (expt 10 n)))
        (/ (round (* x factor)) factor)))))
```

Abstracción de recursión plana

```
;;; (suma '(2 4 6))  =>  12
(define suma
  (letrec
      ((suma-aux
        (lambda (l)
          (if (null? l)
              0
              (+ (car l) (suma-aux (cdr l)))))))
    suma-aux))

;;; (producto '(2 5 6))  =>  60
(define producto
  (letrec
      ((producto-aux
        (lambda (l)
          (if (null? l)
              1
              (* (car l) (producto-aux (cdr l)))))))
    producto-aux))
```

Abstracción de recursión plana

```
;;; ((pertenece?-p 2) '(1 2 3))  =>  #t
(define pertenece?-p
  (lambda (x)
    (letrec
        ((pertenece?-p-aux
          (lambda (l)
            (if (null? l)
                #f
                (or (equal? (car l) x)
                    (pertenece?-p-aux (cdr l)))))))
      pertenece?-p-aux)))
```

Abstracción de recursión plana

```
;;; ((map-p sqrt) '(1 4 9))  =>  (1.0 2.0 3.0)
(define map-p
  (lambda (proc)
    (letrec
        ((map-p-aux
          (lambda (l)
            (if (null? l)
                ()
                (cons (proc (car l)) (map-p-aux (cdr l)))))))
      map-p-aux)))
```

Abstracción de recursión plana

- **Procedimiento** `recursion-plana`

```
(define recursion-plana
  (lambda (semilla proc-plano)
    (letrec
        ((recursion-plana-aux
          (lambda (l)
            (if (null? l)
                semilla
                (proc-plano (car l)
                            (recursion-plana-aux (cdr l)))))))
      recursion-plana-aux)))
```

Abstracción de recursión plana

- **Redefiniciones con** `recursion-plana`

```
;;; (suma '(2 4 6))  =>  12
(define suma (recursion-plana 0 +))

;;; (producto '(2 5 6))  =>  60
(define producto (recursion-plana 1 *))

;;; ((pertenece?-p 2) '(1 2 3))  =>  #t
;;; ((pertenece?-p 4) '(1 2 3))  =>  #f
(define pertenece-p?
  (lambda (x)
    (recursion-plana #f
                     (lambda (a b)
                       (or (equal? x a) b)))))
```

Abstracción de recursión plana

```
;;; ((map-p sqrt) '(1 4 9))  =>  (1.0 2.0 3.0)
(define map-p
  (lambda (proc)
    (recursion-plana ()
                     (lambda (x y)
                       (cons (proc x) y)))))

;;; (inversa '(a b c))  =>  (c b a)
(define inversa
  (recursion-plana
   ()
   (lambda (x y) (append y (list x)))))
```

Abstracción de recursión profunda

- **Ejemplos:**

```
;;; (suma-total '((1 2) 3))  =>  6
(define suma-total
  (letrec
      ((suma-total-aux
        (lambda (l)
          (if (null? l)
              0
              (let ((a (car l)))
                (if (list? a)
                    (+ (suma-total-aux a) (suma-total-aux (cdr l)))
                    (+ a (suma-total-aux (cdr l)))))))))
    suma-total-aux))
```

Abstracción de recursión profunda

```
;;; ((filtra-total-p number?) '((a 2) b 4))  =>  ((2) 4)
(define filtra-total-p
  (lambda (pred)
    (letrec
        ((filtra-total-p-aux
          (lambda (l)
            (if (null? l)
                ()
                (let ((a (car l)))
                  (if (list? a)
                      (cons (filtra-total-p-aux a)
                            (filtra-total-p-aux (cdr l)))
                      (if (pred a)
                          (cons a (filtra-total-p-aux (cdr l)))
                          (filtra-total-p-aux (cdr l)))))))))
      filtra-total-p-aux)))
```

Abstracción de recursión profunda

- **Patrón** `recursion-profunda`

```
(define recursion-profunda
  (lambda (semilla proc-elto proc-lista)
    (letrec
        ((aux
          (lambda (l)
            (if (null? l)
                semilla
                (let ((a (car l)))
                  (if (list? a)
                      (proc-lista (aux a)
                                  (aux (cdr l)))
                      (proc-elto a
                                 (aux (cdr l)))))))))
      aux)))
```

Abstracción de recursión profunda

- **Redefiniciones con** `recursion-profunda`

```
;;; (suma-total '((1 2) 3))  =>  6
(define suma-total (recursion-profunda 0 + +))

;;; ((filtra-total-p number?) '((a 2) b 4))  =>  ((2) 4)
(define filtra-total-p
  (lambda (pred)
    (recursion-profunda
     ()                      ; semilla
     (lambda (a ls)          ; (proc-elto a (aux (cdr l)))
       (if (pred a)
           (cons a ls)
           ls))
     cons)))                 ; (proc-lista (aux a) (aux (cdr l)))
```

Abstracción de recursión profunda

```
;;; ((filtra-total-p number?) '((a 2) b 4))  =>  ((2) 4)
(define filtra-total-p
  (lambda (pred)
    (letrec
        ((filtra-total-p-aux
          (lambda (l)
            (if (null? l)
                ()
                (let ((x (car l))
                      (y (filtra-total-p-aux (cdr l))))
                  (if (pred x)
                      (cons x y)
                      (if (list? x)
                          (cons (filtra-total-p-aux x) y)
                          y)))))))
      filtra-total-p-aux)))
```

Abstracción de recursión profunda

```
;;; ((intercambia-total 1 2) '((1 2) 3 1 (2 4)))  =>  ((2 1) 3 2 (1 4))
(define intercambia-total
  (lambda (a b)
    (letrec
        ((intercambia-total-aux
          (lambda (l)
            (if (null? l)
                ()
                (let ((x (car l))
                      (y (intercambia-total-aux (cdr l))))
                  (if (or (equal? a x) (equal? b x))
                      (if (equal? a x)
                          (cons b y)
                          (cons a y))
                      (if (list? x)
                          (cons (intercambia-total-aux x) y)
                          (cons x y))))))))
      intercambia-total-aux)))
```

Abstracción de recursión profunda

- **Patrón de** `recursion-profunda-y-test`

```
(define recursion-profunda-y-test
  (lambda (semilla test p-test p-elto p-lista)
    (letrec
        ((aux
          (lambda (l)
            (if (null? l)
                semilla
                (let ((x (car l))
                      (y (aux (cdr l))))
                  (if (test x)
                      (p-test x y)
                      (if (list? x)
                          (p-lista (aux x) y)
                          (p-elto x y))))))))
      aux)))
```

Abstracción de recursión profunda

- **Redefiniciones con** `recursion-profunda-y-test`

```
;;; ((filtra-total-p number?) '((a 2) b 4))  =>  ((2) 4)
(define filtra-total-p
  (lambda (pred)
    (recursion-profunda-y-test
     ()                      ; semilla
     pred                    ; (test x)
     cons                    ; (p-test x y)
     (lambda (x y) y)        ; (p-elto x y)
     cons)))                 ; (p-lista (aux x) y)
```

Abstracción de recursión profunda

```
;;; ((intercambia-total 1 2) '((1 2) 3 1 (2 4)))  =>  ((2 1) 3 2 (1 4))
(define intercambia-total
  (lambda (a b)
    (recursion-profunda-y-test
     ()                                  ; semilla
     (lambda (x)                         ; (test x)
       (or (equal? a x) (equal? b x)))
     (lambda (x y)                       ; (p-test x y)
       (if (equal? a x)
           (cons b y)
           (cons a y)))
     cons                                ; (p-elto x y)
     cons)))                             ; (p-lista (aux x) y)
```