Recursión dirigida por los datos

José A. Alonso y María J. Hidalgo

Ciencias de la Computación e Inteligencia Artificial

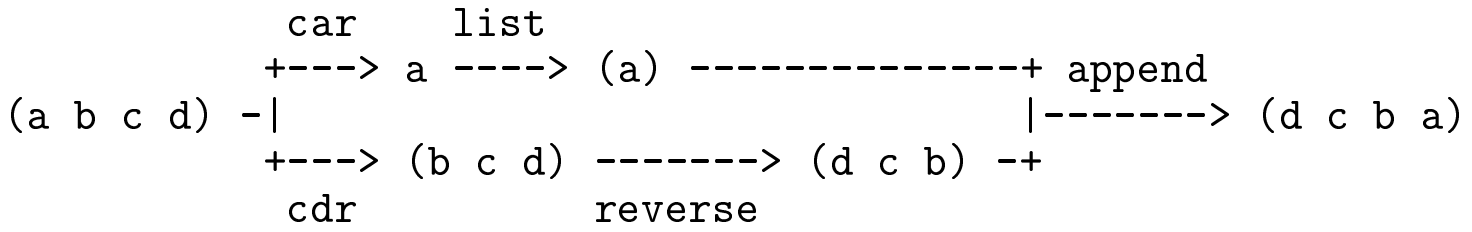
Universidad de Sevilla

Recursión plana: `reverse`

- **Ejemplos:**

```
(reverse '(a b c d))    =>  (d c b a)
(reverse '(a (b c) d))  =>  (d (b c) a)
(reverse '())           =>  ()
```

- **Esquema:**



- **Procedimiento:**

```
(define n-reverse
  (lambda (l)
    (if (null? l)
        ()
        (append (n-reverse (cdr l)) (list (car l))))))
```

Recursión plana: `reverse`

- **Cálculo con traza:**

```
> (trace n-reverse)
#<unspecified>
> (n-reverse '(a b c))
"CALLED" n-reverse (a b c)
 "CALLED" n-reverse (b c)
  "CALLED" n-reverse (c)
   "CALLED" n-reverse ()
   "RETURNED" n-reverse ()
  "RETURNED" n-reverse (c)
 "RETURNED" n-reverse (c b)
"RETURNED" n-reverse (c b a)
(c b a)
> (untrace)
(n-reverse)
```

Recursión plana: `reverse`

- **Cálculo con traza de `n-reverse` y `append`:**

```
> (trace n-reverse append)
#<unspecified>
> (n-reverse '(a b))
"CALLED" n-reverse (a b)
 "CALLED" n-reverse (b)
  "CALLED" n-reverse ()
  "RETURNED" n-reverse ()
  "CALLED" append () (b)
  "RETURNED" append (b)
 "RETURNED" n-reverse (b)
 "CALLED" append (b) (a)
 "RETURNED" append (b a)
"RETURNED" n-reverse (b a)
(b a)
> (untrace)
(append n-reverse)
```

Recursión cruzada: par? y impar?

- **Los predicados de paridad:**

```
(par? 6)     =>  #t
(impar? 6)   =>  #f
```

- **Procedimientos:**

```
(define par?
  (lambda (n)
    (if (= n 0)
        #t
        (impar? (- n 1)))))

(define impar?
  (lambda (n)
    (if (= n 0)
        #f
        (par? (- n 1)))))
```

Recursión cruzada: par? y impar?

- **Cálculo con traza:**

```
> (trace par? impar?)
#<unspecified>
> (par? 4)
"CALLED" par? 4
 "CALLED" impar? 3
  "CALLED" par? 2
   "CALLED" impar? 1
    "CALLED" par? 0
    "RETURNED" par? #t
   "RETURNED" impar? #t
  "RETURNED" par? #t
 "RETURNED" impar? #t
"RETURNED" par? #t
#t
> (untrace)
(impar? par?)
```

Recursión cruzada: par? y impar?

- **Esquema:**

```
(par? 4) => (impar? 3) => (par? 2) => (impar? 1) => (par? 0) => #t
(impar? 4) => (par? 3) => (impar? 2) => (par? 1) => (impar? 0) => #f
```

- **Procedimiento más eficiente:**

```
(define par-2?
  (lambda (n)
    (zero? (remainder n 2))))

(define impar-2?
  (lambda (n)
    (not (par-2? n))))
```

- **Procedimientos primitivos:** even? y odd?

Recursión profunda: cuenta-atomos

- **El predicado** `atom?`**:**

```
(atom? 'abc)      =>  #t
(atom? 23.5)      =>  #t
(atom? '(a b c))  =>  #f
(atom? ())        =>  #t
```

- **Procedimiento** `atom?`**:**

```
(define atom?
  (lambda (a)
    (not (pair? a))))
```

Recursión profunda: cuenta-atomos

- **Ejemplos de** cuenta-atomos**:**

```
(cuenta-atomos '(a d (e f) g))           =>  5
(cuenta-atomos '(8 (a b) (-1 (xy z))))   =>  6
(cuenta-atomos '(()))                    =>  1
(cuenta-atomos '())                      =>  0
```

- **Esquema:**

```
1.- (cuenta-atomos '()) => 0
2.- (cuenta-atomos '(a (c d) e)) =>
      => (+ 1 (cuenta-atomos '((c d) e))) =>
      => 4
3.- (cuenta-atomos '((a b) c d)) =>
      => (+ (cuenta-atomos '(a b)) (cuenta-atomos '(c d))) =>
      => 4
```

Recursión profunda: `cuenta-atomos`

- **Procedimiento** `cuenta-atomos`:

```
(define cuenta-atomos
 (lambda (l)
   (cond
    ((null? l) 0)                                  ; 1
    ((atom? (car l))                               ; 2
     (+ 1 (cuenta-atomos (cdr l))))
    (else (+ (cuenta-atomos (car l))               ; 3
             (cuenta-atomos (cdr l)))))))
```

Recursión profunda: cuenta-atomos

- **Cálculo con traza:**

```
> (cuenta-atomos '(a (c d) e))
"CALLED" cuenta-atomos (a (c d) e)
 "CALLED" cuenta-atomos ((c d) e)
  "CALLED" cuenta-atomos (c d)
   "CALLED" cuenta-atomos (d)
    "CALLED" cuenta-atomos ()
    "RETURNED" cuenta-atomos 0
   "RETURNED" cuenta-atomos 1
  "RETURNED" cuenta-atomos 2
  "CALLED" cuenta-atomos (e)
   "CALLED" cuenta-atomos ()
   "RETURNED" cuenta-atomos 0
  "RETURNED" cuenta-atomos 1
 "RETURNED" cuenta-atomos 3
"RETURNED" cuenta-atomos 4
4
```

Recursión profunda: `cuenta-no-listas`

- **Ejemplos:**

```
(cuenta-atomos '(a (() d) e))     =>  4
(cuenta-no-listas '(a (() d) e))  =>  3
```

- **Procedimiento:**

```
(define cuenta-no-listas
  (lambda (l)
    (cond
     ((null? l) 0)
     ((not (list? (car l)))
      (+ 1 (cuenta-no-listas (cdr l))))
     (else (+ (cuenta-no-listas (car l))
              (cuenta-no-listas (cdr l)))))))
```

Recursión profunda: `intercambia-total`

- **Ejemplos:**

```
(intercambia-total 'a 'b '(a b (c b) a))    =>  (b a (c a) b)
(intercambia-total 1 2 '((1 2) 3 1 (2 4)))  =>  ((2 1) 3 2 (1 4))
```

- **Esquemas:**

```
1.- (intercambia-total 1 2 '())  =>  ()
2.- (intercambia-total 1 2 '(1 3 2 1)) =>
      => (cons 2 (intercambia-total 1 2 '(3 2 1)))
      => (2 3 1 2)
3.- (intercambia-total 1 2 '(2 3 2 1)) =>
      => (cons 1 (intercambia-total 1 2 '(3 2 1)))
      => (1 3 1 2)
4.- (intercambia-total 1 2 '((a 1) 1 3 2)) =>
      => (cons (intercambia-total 1 2 '(a 1))
               (intercambia-total 1 2 '(1 3 2)))
      => ((a 2) 2 3 1)
```

Recursión profunda: intercambia-total

```
5.- (intercambia-total 1 2 '(a 1 3 2)) =>
      => (cons 'a (intercambia-total 1 2 '(1 3 2)))
      => (a 2 3 1)
```

- **Procedimiento:**

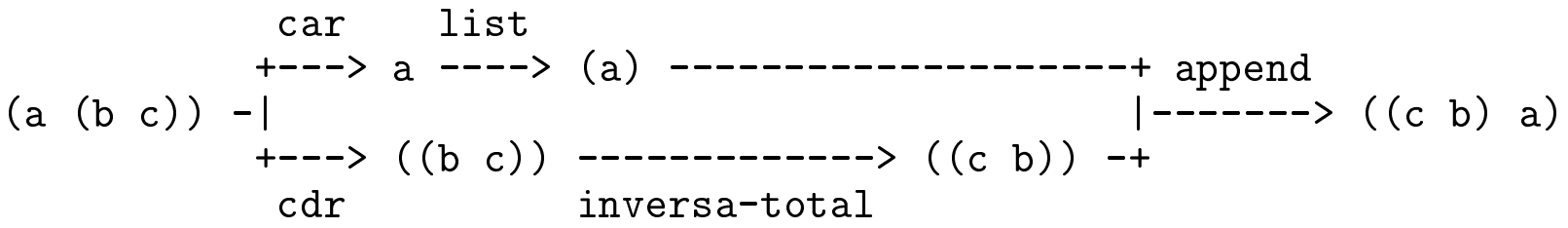
```
(define intercambia-total
  (lambda (x y l)
    (cond
      ((null? l) '())                                          ; 1
      ((equal? x (car l))                                      ; 2
       (cons y (intercambia-total x y (cdr l))))
      ((equal? y (car l))                                      ; 3
       (cons x (intercambia-total x y (cdr l))))
      ((pair? (car l))                                         ; 4
       (cons (intercambia-total x y (car l))
             (intercambia-total x y (cdr l))))
      (else (cons (car l) (intercambia-total x y (cdr l))))))) ; 5
```

Recursión profunda: `inversa-total`

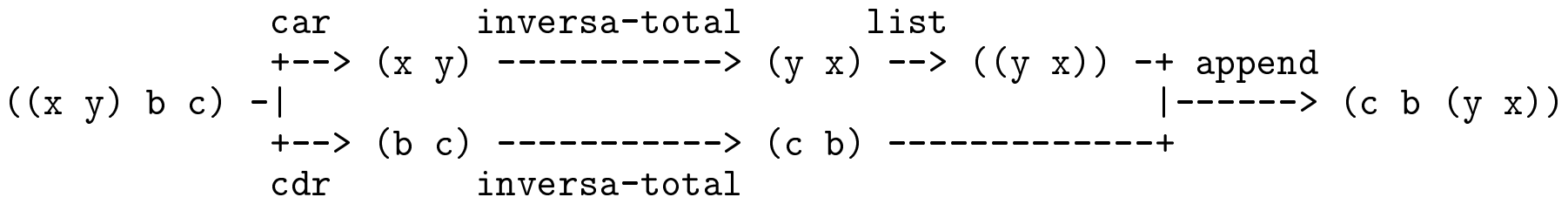
- **Ejemplos:**

```
(inversa-total '(a (b c)))          =>  ((c b) a)
(inversa-total '((1 2) ((3 4) 5)))  =>  ((5 (4 3)) (2 1))
(inversa-total '())                 =>  ()
```

- **Esquemas:**



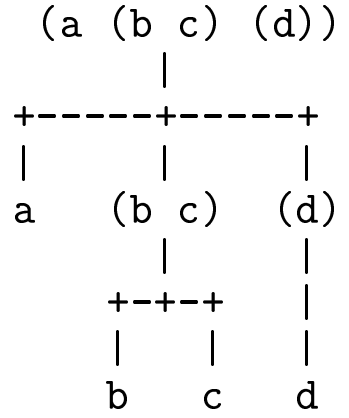
Recursión profunda: `inversa-total`



- **Procedimiento:**

```
(define inversa-total
  (lambda (l)
    (cond
     ((null? l) '())
     ((atom? (car l))
      (append (inversa-total (cdr l)) (list (car l))))
     (else (append (inversa-total (cdr l))
                   (list (inversa-total (car l))))))))
```

Representación de listas mediante árboles



Listas y árboles: profundidad

- **Ejemplos:**

```
(profundidad '(a b c))                => 1
(profundidad '(a (b c) (d)))          => 2
(profundidad '((a (b c)) d (e) f))    => 3
(profundidad 'a)                      => 0
(profundidad '())                     => 0
```

- **Procedimiento:**

```
(define profundidad
  (lambda (expr)
    (if (not (pair? expr))
        0
        (max (+ 1 (profundidad (car expr)))
             (profundidad (cdr expr))))))
```

Listas y árboles: aplanar

- **Ejemplos:**

```
(aplanar '((a ((b c))) d ((e) f)))  =>  (a b c d e f)
(aplanar '(a (b ()) ()))            =>  (a b () ())
(aplanar '())                       =>  ()
```

- **Procedimiento:**

```
(define aplanar
  (lambda (l)
    (cond
     ((null? l) '())
     ((not (pair? (car l)))
      (cons (car l) (aplanar (cdr l))))
     (else (append (aplanar (car l))
                   (aplanar (cdr l)))))))
```

Recursión e iteración: factorial

- **Procedimiento recursivo del factorial:**

```
(define fact
  (lambda (n)
    (if (= n 0)
        1
        (* n (fact (- n 1))))))
```

- **Proceso recursivo del factorial:**

```
(fact 3)
= (* 3 (fact 2))
= (* 3 (* 2 (fact 1)))
= (* 3 (* 2 (* 1 (fact 0))))
= (* 3 (* 2 (* 1 1)))
= (* 3 (* 2 1))
= (* 3 2)
= 6
```

Recursión e iteración: factorial

- **Tabla del factorial iterativo:**

n	4	3	2	1	0
acumulador	1	4	12	24	24

- **Procedimiento `fact-it`:**

```
(define fact-it
  (lambda (n)
    (fact-it-aux n 1)))
```

Recursión e iteración: factorial

```
(define fact-it-aux
   (lambda (n acumulador)
     (if (zero? n)
         acumulador
         (fact-it-aux (- n 1) (* n acumulador)))))
```

- **Proceso iterativo con** `fact-it`

```
(fact-it 3) =
= (fact-it-aux 3 1)
= (fact-it-aux 2 3)
= (fact-it-aux 1 6)
= (fact-it-aux 0 6)
= 6
```

Recursión e iteración: n-reverse

- **Definición del n-reverse recursivo:**

```
(define n-reverse
  (lambda (l)
    (if (null? l)
        ()
        (append (n-reverse (cdr l)) (list (car l))))))
```

- **Cálculo con el n-reverse recursivo:**

```
(n-reverse '(a b c)) =
= (append (n-reverse '(b c)) '(a))
= (append (append (n-reverse '(c)) '(b)) '(a))
= (append (append (append (n-reverse ()) '(c)) '(b)) '(a))
= (append (append (append () '(c)) '(b)) '(a))
= (append (append '(c) '(b)) '(a))
= (append '(c b) '(a))
= (c b a)
```

Recursión e iteración: `n-reverse`

- **Tabla iterativa de la inversa:**

```
+----------+----------+
|  l       |acumulador|
+----------+----------+
|  (a b c) |       () |
|    (b c) |      (a) |
|      (c) |    (b a) |
|       () |  (c b a) |
+----------+----------+
```

- **Procedimiento `n-reverse-it`:**

```
(define n-reverse-it
  (lambda (l)
    (n-reverse-it-aux l '())))
```

Recursión e iteración: n-reverse

```
(define n-reverse-it-aux
  (lambda (l acumulador)
    (if (null? l)
        acumulador
        (n-reverse-it-aux (cdr l) (cons (car l) acumulador)))))
```

- **Cálculo con n-reverse-it:**

```
(n-reverse-it '(a b c)) =
= (n-reverse-it-aux '(a b c) '())
= (n-reverse-it-aux '(b c)   '(a))
= (n-reverse-it-aux '(c)     '(b a))
= (n-reverse-it-aux '()      '(c b a))
= (c b a)
```

Recursión e iteración: n-reverse

- **Comparación de** `n-reverse` **y** `n-reverse-it`**:**

```
;;; (crea-lista 5)  =>  (5 4 3 2 1)
(define crea-lista
  (lambda (n)
    (if (zero? n)
        ()
        (cons n (crea-lista (- n 1))))))

> (define l-1000 (crea-lista 1000))
;Evaluation took 30 mSec (0 in gc) 5017 cells work, 40 bytes other
#<unspecified>
> (n-reverse l-1000)
;Evaluation took 2630 mSec (1570 in gc) 506507 cells work, 31 bytes other
(1 2 3 ... 998 999 1000)
> (n-reverse-it l-1000)
;Evaluation took 30 mSec (0 in gc) 6012 cells work, 31 bytes other
(1 2 3 ... 998 999 1000)
```

Recursión doble: Fibonacci

- **La sucesión de Fibonacci:**

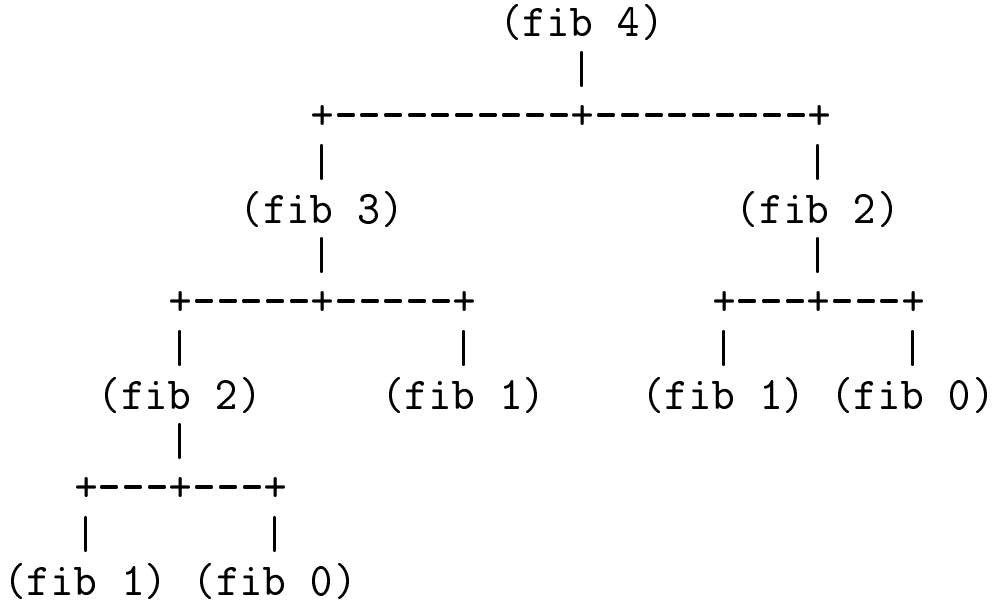
```
0, 1, 1, 2, 3, 5, 8, 13, ...

F(0) = 0,
F(1) = 1,
F(n) = F(n-1) + F(n-2), si n > 1.
```

- **Procedimiento recursivo:**

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

Recursión doble: Fibonacci

- **Arbol de computación:**



Recursión doble: Fibonacci

- **Tabla iterativa para la sucesión de Fibonacci:**

n	6	5	4	3	2	1	0	n-1
acumulador-1	0	1	1	2	3	5	8	acumulador-2
acumulador-2	1	1	2	3	5	8	13	acumulador-1 + acumulador-2

Recursión doble: Fibonacci

- **Procedimiento iterativo de Fibonacci:**

```
(define fib-it
   (lambda (n)
     (if (zero? n)
         0
         (fib-it-aux n 0 1))))

(define fib-it-aux
  (lambda (n acumulador-1 acumulador-2)
    (if (= n 1)
        acumulador-2
        (fib-it-aux (- n 1)
                    acumulador-2
                    (+ acumulador-1 acumulador-2)))))
```

Recursión doble: Fibonacci

- **Cálculo con traza:**

```
> (trace fib-it fib-it-aux)
#<unspecified>
> (fib-it 4)
"CALLED" fib-it 4
 "CALLED" fib-it-aux 4 0 1
  "CALLED" fib-it-aux 3 1 1
   "CALLED" fib-it-aux 2 1 2
    "CALLED" fib-it-aux 1 2 3
    "RETURNED" fib-it-aux 3
   "RETURNED" fib-it-aux 3
  "RETURNED" fib-it-aux 3
 "RETURNED" fib-it-aux 3
"RETURNED" fib-it 3
3
> (untrace)
(fib-it-aux fib-it)
```

Recursión doble: Fibonacci

- **Comparación de los procedimientos recursivos e iterativos:**

```
> (fib 30)
;Evaluation took 195340 mSec (45920 in gc) 10770152 cells work, 33 bytes other
832040
> (fib-it 30)
;Evaluation took 0 mSec (0 in gc) 188 cells work, 33 bytes other
832040
```

Bibliografía

- [Abelson–96]
  Cap. 1.2: "Procedures and processes they generate".
- [Springer–94]
  Cap. 3: "Data driven recursion".