Procedimientos localmente definidos

José A. Alonso y María J. Hidalgo

Ciencias de la Computación e Inteligencia Artificial

Universidad de Sevilla

Entornos. Procedimientos `let` y `letrec`.

- **Concepto de entorno: lista de pares (variable . valor).**
- **Entornos globales:**
  - **Entorno global inicial (del sistema).**
  - **Entorno global del usuario.**
- **Entornos locales:**
  - **Ejemplo:** `((lambda (x y) (+ x y z)) 3 4)`.
  - **Variables ligadas localmente:** `x`, `y`
  - **Variables libres:** `z`.
  - **Campo de una variable.**

Entornos. Procedimientos `let` y `letrec`.

- **Ejemplos de evaluación:**

```
> (define a 1)
#<unspecified>
> (define z '(4))
#<unspecified>
> ((lambda (f y) (f a (f y z))) cons 3)
(1 3 4)
> (define f list)
#<unspecified>
> ((lambda (f y) (f a (f y z))) cons 3)
(1 3 4)
> (f 5 6)
(5 6)
> ((lambda (x)
     ((lambda (y) (- x y)) 15)) 20)
5
```

El procedimiento `let`

- **Ejemplo:**

```
(define a 5)            =>  #<unspecified>
(+ a 1)                 =>  6
(let ((a 3)) (+ a 1))   =>  4
(+ a 1)                 =>  6
```

- **Ejemplos:**

```
> (let ((x 1))
    (let ((x (+ x 1)))
      (+ x 3)))
5
> (let ((x 2) (y 3))
    (let ((x (lambda (x) (+ x y)))
          (y (+ x 4)))
      (list (x y) (x 5))))
(9 8)
```

El procedimiento `let`

- **Sintaxis y semántica:**

```
(let ((variable-1 valor-1)...(variable-n valor-n))
  expresión-1
  ...
  expresión-m)
```

- **Relación entre `let` y `lambda`:**

```
(let ((variable-1 valor-1)...(variable-n valor-n))
  expresión-1
  ...
  expresión-m)

((lambda (variable-1 ... variable-n)
   expresión-1
   ...
   expresión-m)
 valor-1 ... valor-n)
```

El procedimiento `let`

- **Ejemplo:**

```
;;; (raices 1 -5 6)  =>  (3.0 2.0)
(define raices
  (lambda (a b c)
    (let ((aux1 (- b))
          (aux2 (sqrt (- (* b b)
                         (* 4 a c))))
          (aux3 (* 2 a)))
      (list (/ (+ aux1 aux2) aux3)
            (/ (- aux1 aux2) aux3)))))
```

El procedimiento `let`

- **Ejemplo:**

```
;;; (quita-mas-izda 'a '(a b a))         =>  (b a)
;;; (quita-mas-izda 'a '((c a) a b a))   =>  ((c) a b a)
(define quita-mas-izda
  (lambda (x l)
    (cond
     ((null? l) '())
     ((equal? (car l) x) (cdr l))
     ((pair? (car l))
      (let ((aux (quita-mas-izda x (car l))))
        (cons aux
              (if (equal? (car l) aux)
                  (quita-mas-izda x (cdr l))
                  (cdr l)))))
     (else (cons (car l)
                 (quita-mas-izda x (cdr l)))))))
```

El procedimiento `letrec`

- **Ejemplo**

```
> (let ((fact (lambda (n)
                (if (zero? n)
                    1
                    (* n (fact (- n 1)))))))
    (fact 4))

ERROR: unbound variable:  fact
; in expression: (... fact (- n 1))
; in scope:
;   (n)
> (letrec ((fact (lambda (n)
                   (if (zero? n)
                       1
                       (* n (fact (- n 1)))))))
    (fact 4))
24
```

El procedimiento `letrec`

- **Sintaxis y semántica:**

```
(letrec ((variable-1 valor-1)...(variable-n valor-n))
  expresión-1
  ...
  expresión-m)
```

- **Ejemplo:**

```
;;; (fact-it 4)  =>  24
(define fact-it
  (lambda (n)
    (letrec ((fact-it-aux
              (lambda (k ac)
                (if (zero? k)
                    ac
                    (fact-it-aux (- k 1) (* k ac))))))
      (fact-it-aux n 1))))
```

El procedimiento `letrec`

- **Ejemplo:**

```
;;; (inversa-it '(a b c))  =>  (c b a)
(define inversa-it
  (lambda (l)
    (letrec
        ((inversa-it-aux
          (lambda (ls ac)
            (if (null? ls)
                ac
                (inversa-it-aux (cdr ls)
                                (cons (car ls) ac))))))
      (inversa-it-aux l '()))))
```

El procedimiento `letrec`

- **Raiz cuadrada por el método de Newton**

```
;;; (raiz-cuadrada 9)  =>  3.00009155413138
(define raiz-cuadrada
  (lambda (x)
    (raiz-cuadrada-iter 1 x)))

(define raiz-cuadrada-iter
  (lambda (y x)
    (if (aceptable? y x) y
        (raiz-cuadrada-iter (mejora y x) x))))

(define aceptable?
  (lambda (y x) (< (abs (- (* y y) x)) 0.001)))

(define mejora
  (lambda (y x) (/ (+ y (/ x y)) 2)))
```

El procedimiento `letrec`

- **Ejemplo:**

```
(define raiz-cuadrada-mediante-letrec
  (lambda (x)
    (letrec
      ((aceptable?
        (lambda (y x)
          (< (abs (- (* y y) x))
             0.001)))
       (mejora
        (lambda (y x)
          (/ (+ y (/ x y)) 2)))
       (raiz-cuadrada-iter
        (lambda (y x)
          (if (aceptable? y x)
              y
              (raiz-cuadrada-iter (mejora y x) x)))))
      (raiz-cuadrada-iter 1 x))))
```

Test de primalidad

```
;;; (primo? 7)  =>  #t
;;; (primo? 9)  =>  #f
(define primo?
  (lambda (n)
    (and (not (zero? n))
         (= n (menor-divisor n)))))

;;; (menor-divisor 6)  =>  2
;;; (menor-divisor 7)  =>  7
(define menor-divisor
  (lambda (n)
    (menor-divisor-a-partir-de n 2)))
```

Test de primalidad

```
;;; (menor-divisor-a-partir-de 70 3)  =>  5
;;; (menor-divisor-a-partir-de 70 5)  =>  5
;;; (menor-divisor-a-partir-de 70 10)  =>  70
(define menor-divisor-a-partir-de
  (lambda (n d)
    (cond
     ((> (* d d) n) n)
     ((divisible? n d) d)
     (else (menor-divisor-a-partir-de n (+ d 1))))))

;;; (divisible? 6 3)  =>  #t
;;; (divisible? 6 4)  =>  #f
(define divisible?
  (lambda (b a)
    (= (remainder b a) 0)))
```

Test de primalidad

- primo?-con-letrec

```
(define primo?-con-letrec
  (lambda (n)
    (letrec ((menor-divisor
              (lambda (n)
                (menor-divisor-a-partir-de n 2)))
             (menor-divisor-a-partir-de
              (lambda (n d)
                (cond
                 ((> (* d d) n) n)
                 ((divisible? n d) d)
                 (else (menor-divisor-a-partir-de n (+ d 1))))))
             (divisible?
              (lambda (b a) (= (remainder b a) 0))))
      (and (not (zero? n))
           (= n (menor-divisor n))))))
```

Potencias

- **Potencia recursiva:**

```
;;; (potencia-r 2 4)  =>  16
(define potencia-r
  (lambda (b n)
    (if (zero? n) 1
        (* b (potencia-r b (- n 1))))))
```

- **Potencia iterativa:**

```
(define potencia-it
  (lambda (b n)
    (letrec ((potencia-it-aux
              (lambda (b n ac)
                (if (zero? n)
                    ac
                    (potencia-it-aux b (- n 1) (* ac b))))))
      (potencia-it-aux b n 1))))
```

Potencias

- **Potencia recursiva rápida:**

```
(define potencia-r-rapida
  (lambda (b n)
    (cond
      ((zero? n) 1)
      ((even? n)
       (cuadrado (potencia-r-rapida b (/ n 2))))
      (else (* b (potencia-r-rapida b (- n 1)))))))

(define cuadrado
  (lambda (n)
    (* n n)))
```

Potencias

- **Potencia iterativa rápida:**

```
(define potencia-it-rapida
  (lambda (b n)
    (letrec
        ((potencia-it-rapida-aux
          (lambda (b n ac)
            (cond
             ((zero? n) ac)
             ((= n 2)
              (potencia-it-rapida-aux b 0 (* ac (cuadrado b))))
             ((even? n)
              (potencia-it-rapida-aux (cuadrado b) (/ n 2) ac))
             (else
              (potencia-it-rapida-aux b (- n 1) (* ac b)))))))
      (potencia-it-rapida-aux b n 1))))
```

Potencias

- **Comparaciones:**

```
> (potencia-r 2 200)
;Evaluation took 10 mSec (0 in gc) 1183 cells work, 2681 bytes other
1606938044258990275541962092341162602522202993782792835301376
> (potencia-r-rapida 2 200)
;Evaluation took 0 mSec (0 in gc) 92 cells work, 83 bytes other
1606938044258990275541962092341162602522202993782792835301376
> (expt 2 200)
;Evaluation took 0 mSec (0 in gc) 16 cells work, 105 bytes other
1606938044258990275541962092341162602522202993782792835301376
```

Bibliografía

- [Abelson–96]
  Cap. 1: "Building abstractions with procedures".
- [Springer–94]
  Cap. 3: "Locally definded procedures".