Asignaciones

- Las asignaciones no crean nuevos vínculos, solo cambian los valores de los ya existentes. Esto se realiza con el método set!
  - define abcde '(a b c d e))
  - abcde ⇒ (a b c d e)
  - (set! abcde (cdr abcde))
  - abcde ⇒ (b c d e)

Asignaciones

```
(define quadratic-formula
   (lambda (a b c)
      (let ((root1 0) (root2 0) (minusb 0) (radical 0)
            (divisor 0))
         (set! minusb (- 0 b))
         (set! radical (sqrt (- (* b b) (* 4 (* a c)))))
         (set! divisor (* 2 a))
         (set! root1 (/ (+ minusb radical) divisor))
         (set! root2 (/ (- minusb radical) divisor))
         (cons root1 root2))))
```

0 = ax2 + bx + c.

Asignaciones

```
(define quadratic-formula
  (lambda (a b c)
    (let ((minusb (- 0 b))
          (radical (sqrt (- (* b b) (* 4 (* a c)))))
          (divisor (* 2 a)))
      (let ((root1 (/ (+ minusb radical) divisor))
            (root2 (/ (- minusb radical) divisor)))
        (cons root1 root2)))))
```

Asignaciones

- Let*: permite realizar asignaciones secuenciales, donde la definición de las variables internas pueden ver a las variables externas.
  - (let* ((x (* 5.0 5.0))
    (y (- x (* 4.0 4.0))))
    (sqrt y)) ⇒ 3.0

Recursividad

- la recursividad se produce cuando un procedimiento se llama a si mismo.
  - Ej:
    - (define length
      (lambda (ls)
        (if (null? ls)
            0
            (+ (length (cdr ls)) 1))))
    - (length '()) ⇒ 0
    - (length '(a)) ⇒ 1
    - (length '(a b)) ⇒ 2

Recursividad

- ¿es posible hacer un let-bound recursivo?
  - EJ:

```
(let ((sum (lambda (ls)
                (if (null? ls)
                    0
                    (+ (car ls) (sum (cdr ls)))))))
  (sum '(1 2 3 4 5)))
```

- NO!!!!
- sum solo existe en el cuerpo del LET y no en la definición de las variables

Recursividad

- *letrec*: al igual que *let* permite definir un conjunto de pares variable-valor y un conjunto de sentencias que las referencian.
- A diferencia de *let*, las variables son vistas en la cabecera también.
  - EJ:

```
(letrec ((sum (lambda (ls)
                (if (null? ls)
                0
                (+ (car ls) (sum (cdr ls)))))))
  (sum '(1 2 3 4 5)))
```

Secuenciamiento

- (begin exp1 exp2 ...) : retorna el valor de la última expresión.
- (define x 3)

```
(begin
        (set! x (+ x 1))
        (+ x x)) ⇒ 8
```

Secuenciamiento

- (define swap-pair!
  (lambda (x)
    (let ((temp (car x)))
      (set-car! x (cdr x))
      (set-cdr! x temp)
      x)))
- (swap-pair! (cons 'a 'b)) ⇒ (b . a)

Secuenciamiento

- (define swap-pair
  (lambda (x)
  (cons (cdr x) (car x))
)
)

Vectores

- un vector es una secuencia de objetos separados por un blanco y precedidos por un # o con la siguiente sintaxis:
  - (vector obj ...)
- Ejemplos:
  - #(a b c) → vector de elementos a, b y c
  - (vector) ⇒ #()
  - (vector 'a 'b 'c) ⇒ #(a b c)

Vectores

- (make-vector n)
  (make-vector n obj): retornan un vector de n posiciones. Si se provee *obj* se llenaran las posiciones con *obj*, en caso contrario permanecerán como *indefinido*
- *(make-vector 0) ⇒ #()*
- *(make-vector 0 'a) ⇒ #()*
- *(make-vector 5 'a) ⇒ #(a a a a a)*

Vectores

- (vector-length vector) : retorna la cantidad de elementos de un vector.
  - (vector-length '#()) ⇒ 0
  - (vector-length '#(a b c)) ⇒ 3
  - (vector-length (vector 1 2 3 4)) ⇒ 4
  - (vector-length (make-vector 300)) ⇒ 300

Vectores

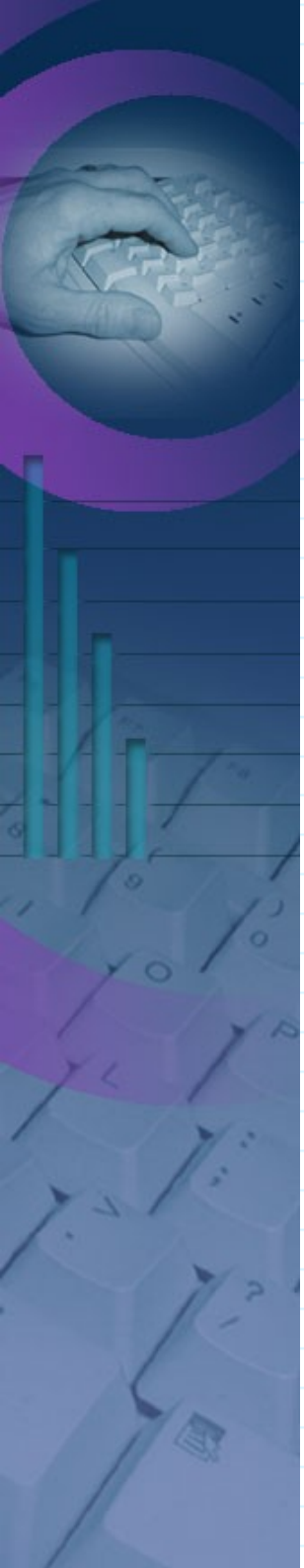
- (vector-ref vector n) : retorna la enésima posición de un vector
  - (vector-ref '#(a b c) 0) ⇒ a
  - (vector-ref '#(a b c) 1) ⇒ b
  - (vector-ref '#(x y z w) 3) ⇒ w

Vectores

- (vector-set! vector n obj): establece el valor de la enésima posición del vector a *obj*
  - *(let ((v (vector 'a 'b 'c 'd 'e)))*
    (vector-set! v 2 'x)
    v) ⇒ #(a b x d e)

Vectores

- (vector-fill! vector obj) reemplaza cada elemento del vector *obj*
- (vector->list vector) devuelve una lista a partir de un vector
- (list->vector list) convierte una lista en vector



Programación Funcional

continuará...