Variables y Expresiones Let

- La forma sintáctica Let incluye pares de *variables-expresiones* junto con una secuencia de expresiones que representan el cuerpo del Let

(let ((var1 val) [(var2 val)...]) exp1 exp2 ...)

Variables y Expresiones Let

- También se usa para simplificar expresiones
  - (+ (* 4 4) (* 4 4)) ⇒ 32
  - (let ((a (* 4 4)))
    (+ a a)) ⇒ 32
  - (let ((list1 '(a b c)) (list2 '(d e f)))
    (cons (cons (car list1) (car list2))
    (cons (car (cdr list1)) (car (cdr list2)))))
    ⇒ ((a . d) b . e)

Variables y Expresiones Let

- Es posible anidar Lets
  - (let ((a 4) (b -3))
    (let ((a-squared (* a a))
    (b-squared (* b b)))
    (+ a-squared b-squared))) → 25
  - (let ((x 1))
    (let ((x (+ x 1)))
    (display x))) → 2

Expresiones Lambda

- Lambda permite crear un nuevo procedimiento
- Expresión general:
  - (lambda (var ...) exp1 exp2 ...)
  - Ejemplo: (lambda (x) (+ x x)) ⇒ #<procedure>
- Uso:
  - ((lambda (x) (+ x x)) (* 3 4)) ⇒ 24

Expresiones Lambda

- Como los procedimientos son objetos los podemos asignar a variables
  - (let ((double (lambda (x) (+ x x))))
    (list (double (* 3 4))
    (double (/ 99 11))
    (double (- 2 7)))) ⇒ (24 18 -10)
  - (let ((double-cons (lambda (x) (cons x x))))
    (double-cons 'a)) ⇒ (a . a)

Definiciones de alto nivel

- Las definiciones de **Alto Nivel** son vistas desde todos los procedimientos
- se declaran a partir de la cláusula *define*
  - *(define double-any*
    (lambda (f x)
    (f x x)))
  - *(double-any + 10) ⇒ 20*
  - *(double-any cons 'a) ⇒ (a . a)*

Definiciones de Alto Nivel

- se pueden utilizar para cualquier tipo de objetos, no solo procedimientos

  (define sandwich "milanesa-tomate-y-lechuga")
  sandwich ⇒ "milanesa-tomate-y-lechuga"

Definiciones de Alto Nivel

- Scheme provee abreviaturas llamadas cadr y cddr que son composiciones de
  - (car (cdr *list*))
  - (cdr (cdr *list*))
  - (define cadr
    (lambda (x)
    (car (cdr x))))
  - (define cddr
    (lambda (x)
    (cdr (cdr x))))

Expresiones Condicionales

- (if (<test>)
  (<verdad>)
  (<falso>)
)
  - Ej: (define abs
    (lambda (n)
    (if (< n 0)
    (- 0 n)
    n)
    ))

Expresiones Condicionales

- not: devuelve el inverso del parámetro dado
  - (not #t) → #f
  - (not “false”) → #f
  - (not #f) → #t
- or/and: realiza la comparación lógica y devuelve el resultado
  - (or) → #f
  - (or #t #f) → #t
  - (and #t #f) → #f

Expresiones Condicionales

- =, <, >, <=, y >= son todos predicados y responden a preguntas específicas sobre sus argumentos devolviendo un valor de verdad
- los nombres de los predicados normalmente finalizan en ? excepto los anteriores

Expresiones Condicionales

- null? : devuelve #t si el argumento es una lista vacía
  - (null? ()) → #t
  - (null? 'abc) ⇒ #f
  - (null? '(x y z)) ⇒ #f

Expresiones Condicionales

- eqv?: requiere dos argumentos y devuelve #t si son equivalentes
  - (eqv? 'a 'a) ⇒ #t
  - (eqv? 'a 'b) ⇒ #f
  - (eqv? #f #f) ⇒ #t
  - (eqv? #t #t) ⇒ #t
  - (eqv? #f #t) ⇒ #f
  - (eqv? 3 3) ⇒ #t
  - (eqv? 3 2) ⇒ #f

Expresiones Condicionales

- Otros predicados son:
  - pair?
  - symbol?
  - number?
  - string?

Expresiones Condicionales

- cond:permite realizar múltiples test/acciones. Su forma general es la siguiente:
  - (cond (test exp) ... (else exp))
    - Ej:
```
(define sign
    (lambda (n)
        (cond
            ((< n 0) -1)
            ((> n 0) +1)
            (else 0))))
```

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 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