Bindings & Scopes

• Names can be bound to values by introducing a nested scope

• let takes two or more arguments:
  – The first argument is a list of pairs
    • In each pair, the first element is the name, while the second is the value/expression
  – Remaining arguments are evaluated in order
  – The value of the construct as a whole is the value of the final argument
  – E.g. (let ((a 3)) a)

let Examples

• E.g., (let ((a 3)
  (b 4)
  (square (lambda (x) (* x x)))
  (plus +))
  (sqrt (plus (square a) (square b)))))

• The scope of the bindings produced by let is its second and following arguments
let Examples

• E.g., (let ((a 3))
  (let ((a 4)
    (b a))
  (+ a b))) => ?

• b takes the value of the outer a, because the defined names are visible “all at once” at the end of the declaration list

let* Example

• let* makes sure that names become available “one at a time”
• E.g., (let*((x 1) (y (+ x 1)))
  (+ x y)) => ?
Functions

• **quote: identity function**
  - When the function is given a parameter, it simply returns the parameter
  - E.g., (quote A) => A
    (quote (A B C)) => (A B C)

• The common abbreviation of quote is apostrophe (’)
  - E.g., ’a => a
    ’(A B C) => (A B C)

List Functions

• **car: returns the first element of a given list**
  - E.g., (car ’(A B C)) => A
    (car ’((A B) C D)) => (A B)
    (car ’A) => ?
    (car ’(A)) => ?
    (car ’()) => ?
List Functions

• **cdr**: returns the remainder of a given list after its car has been removed
  – E.g., (cdr '(A B C)) => (B C)
  – (cdr '((A B) C D)) => (C D)
  – (cdr 'A) => ?
  – (cdr '(A)) => ?
  – (cdr '()) => ?

List Functions

• **cons**: concatenates an element with a list
• **cons** builds a list from its two arguments
  – The first can be either an atom or a list
  – The second is usually a list
  – E.g., (cons 'A '()) => (A)
  – (cons 'A '(B C)) => (A B C)
  – (cons '(A B)) => ?
  – (cons '(A B) '(C D)) => ?
  – How to compose a list (A B C) from A, B, and C?
List Functions

• Note that cons can take two atoms as parameters, and return a dotted pair
  – E.g., (cons 'A 'B) => (A . B)
  – The dotted pair indicates that this cell contains two atoms, instead of
    an atom + a pointer
    or
    a pointer + a pointer

More Predicate Functions

• The following returns #t if the symbolic atom is of the indicated type, and #f otherwise
  – E.g., (symbol? 'a) => #t
     (symbol? '()) => #f
  – E.g., (number? '55) => #t
     (number? 55) => #t
     (number? '(a)) => #f
  – E.g., (list? '(a)) => #t
  – E.g., (null? '()) => #t
More Predicate Functions

• eq? returns true if two objects are equal through pointer comparison
  – Guaranteed to work on symbols
  – E.g., (eq? 'A 'A) => #T
    (eq? 'A '(A B)) => #F

• equal? recursively compares two objects to determine if they are equal
  – The objects can be symbols, atoms, numbers, and lists

How do we implement equal?

(define (atom? atm)
  (cond
   ((list? atm) (null? atm)) (else #T)
   )
  )

(define (equal? lis1 lis2)
  (cond
   ((atom? lis1) (eq? lis1 lis2))
   ((atom? lis2) #F)
   ((equal? (car lis1) (car lis2))
    (else #F)
   )
  )
)
More Examples

(define (member? atm lis)
  (cond
   ((null? lis) #f)
   ((eq? atm (car lis)) #t)
   (else (member? atm (cdr lis))))
)

(define (append lis1 lis2)
  (cond
   ((null? lis1) lis2)
   (else (cons (car lis1)
                (append(cdr lis1) lis2))))
)

What is returned for the following function?
(member? 'b '(a (b c)))

Is lis2 appended to lis1, or lis1 prepended to lis2?

An example: apply-to-all function

(define (mapcar fctn lis)
  (cond
   ((null? lis) '())
   (else (cons (fctn (car lis))
               (mapcar fctn (cdr lis))))
  )
)