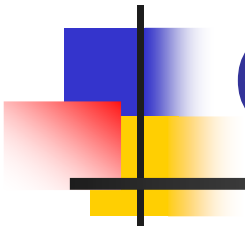
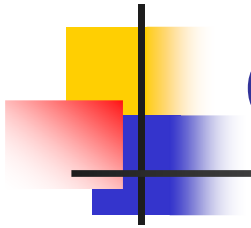


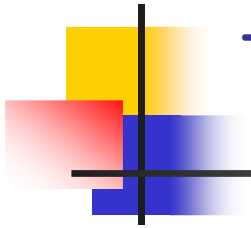
Other built-in Operators

Shorthand for nested car, cdr



- Scheme provides shorthands for expressions consisting of successive application of car and cdr
  - (car (cdr x)) -> (cadr x)
  - (cdr (car x)) -> (cdar x)
  - (car (cdr (car x))) -> (cadar x)

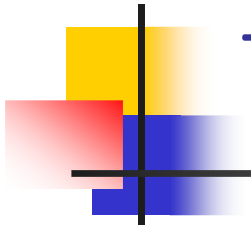
expression	shorthand	Value
x	x	((it seems that) you (like me)
(car x)	(car x)	(it seems that)
(car (car x))	(caar x)	it
(cdr (car x))	(cdar x)	(seems that)
(cdr x)	(cdr x)	(you (like) me)
(car (cdr x))	(cadr x)	you
(cdr (cdr x))	(cddr x)	((like) me)



The let construct

- (let ((x$_1$ E$_1$) (x$_2$ E$_2$) ... (x$_k$ E$_k$)) F)
  - E$_1$, E$_2$, ... , E$_k$ are all evaluated
  - Then F is evaluated with x$_i$
  - the result is the value of F
- The let constructor
  - Allows subexpressions to be named
  - Can be used to factor out common subexpression
    - ex) (let ((three-sq (square 3))) (+ three-sq three-sq))

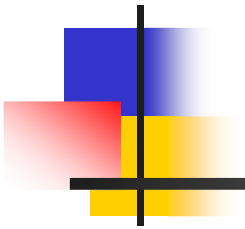
      ; (+ (square 3) (square 3)) = 18



The let construct

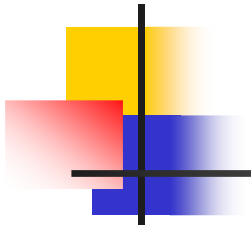
- The sequential variant of the let
  (let* ((x_1 E_1) (x_2 E_2) ... (x_k E_k)) F)
  - Binds x_i to the value of E_i before E_{i+1}
  - ex) (define x 0)

```
(let (( x 2 ) ( y x )) y )          ; 0
(let* (( x 2 ) ( y x )) y )         ; 2
```



10.3 List Manipulation

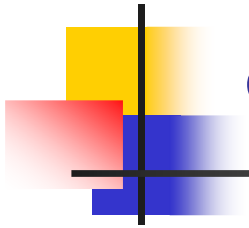
Getting the length of a list : `length`



- E.g.`(length '(1 2 3))` → `3`
- Length of an empty list) = 0
  - **`(length '()) ≡ 0`**
- Length of a nonempty list `(cons a y)` = length (y) + 1
  - `(length (cons a y))≡ (+ 1 length y))`
  - Let `(cons a y)` → `x`, `(cdr x)` → `y`
    `(length x) ≡ (+ 1 (length (cdr x)))`

```
(define (length x)
     (cond ((null? x) 0)
           (else (+ 1 (length (cdr x))))))
```

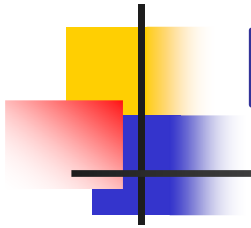
Appending two lists : append



- e.g.
  - (append '() '(a b c d)) → (a b c d)
  - (append '(a b c) '(d)) → (a b c d)
  - (cons 'a (append '(b c) '(d))) → (a b c d)
- **(append '() z)) ≡ z**
- (append (cons a y) z) ≡ (cons a (append y z))
  → (append x z) ≡ (cons (car x) (append (cdr x) y)))))

```
(define (append x y)
      (cond ((null? x) y)
            (else (cons (car x) (append (cdr x) y)))))
```

Getting a flattened form of a list : `flatten`



- E.g.
  - `((a) ((b b)) (((c c c )))` → `(a b b c c c)`
- Use Scheme function `pair?`
  - Test whether its argument is a list
  - E.g. `(pair? 1)` → `#f`, `(pair? '(1))` → `#t`

```
(define (flatten x)
  (cond ((null? x) '())
        ((not (pair? x)) (list x))
        (else (append (flatten (car x) )
                      (flatten (cdr x))))))
```

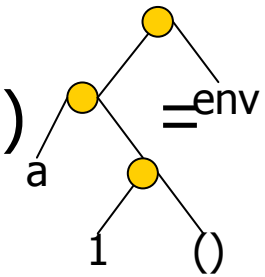
Mapping a function across list elements : **map**

- E.g.
  - `(map square '(1 2 3 4 5))` → `(1 4 9 16 25)`
    - `(define (square x) (* x x))`
  - `(map car '((a 1) (b 2) (c 3) (d 4)))` → `(a b c d)`

```
(define (map f x)
  (cond ((null? x) '())
        (else (cons (f (car x))
                    ( map f (cdr x))))))
```

Association lists(1)

- A list of pairs
  - e.g. `((a 1) (b 2) (c 3) … )`
- `bind :` returns an association list with a new binding for a key
  - e.g. `(bind 'a 1 env)`




```
(define (bind key value env)
  (cons (list key value) env))
```

Association lists(2)

- `bind-all` : binds `keys` to `values`
  - e.g. `(bind-all (a b c) (1 2 3) 'env)`
    → `((a 1) (b 1) (c 3) env)`

```
(define (bind-all keys values env)
  (append (map list keys values) env)
```

- `assoc` : extracts the 1st binding for a variable from association list(built-in function)
  - e.g. `(assoc 'a '((a 1) (b 2) (a 3)))` → `(a 1)`
    `(assoc 'b '((a 1) (b 2) (a 3)))` →`(b 2)`
  - No binding found → false