Functional Programming

Chapter 11

Functional Programming

- No side effects
  - output of a program is a mathematical function of the inputs
  - no internal state, no side effects
- Recursion and composition
  - effects achieved by applying functions: recursion, composition
- First-class functions:
  - can be passed as a parameter
  - can be returned from a subroutine
  - can be assigned in a variable
  - (more strictly) can be computed at run time

Functional Programming

- Polymorphism
  - Functions can be applied to general class of arguments
- Lists
  - Natural recursive definition
  - List = head + tail (list)
- Homogeneity
  - program is a list – can be manipulated the same as data
- Garbage collection
  - heap allocation for dynamically allocated data
  - unlimited extent

Functional vs Imperative

- Advantages
- No side effects
    - predictable behavior
- *Referential transparency*
    - Expressions are independent of evaluation order
- *Equational reasoning*
    - Expressions equivalent at some point in time are equivalent at *any* point in time

Functional vs Imperative

- Disadvantages
- *Trivial update problem*
  - Every result is a new object instead of a modification of an existing one
- Data structures different from lists more difficult to handle
  - multidimensional arrays
  - dictionaries
  - in-place mutation
- The trivial update problem is not an inherent weakness of functional programming
  - The implementation could detect whether an old version of a structure will never be used again and update in place

Scheme

- Originally developed in 1975
- Initially very small
- Now is a complete general-purpose language
- Still derived from a smalls set of key concepts

- Lexically scoped
- Functions are first class values
- Implicit storage management

Scheme vs λ-calculus

- Scheme syntax very similar with λ-calculus

- Examples:
  - λ-calculus

    $\lambda x.x$

    $(\lambda x.x * x)\ 4 \Rightarrow_\beta 16$

  - Scheme

```
(lambda (x) x)
((lambda (x) (* x x)) 4)   ⟹   16
```

prefix notation

Scheme: Interpreter

- Interacting with the interpreter

```
"hello" ⇒ "hello"
42 ⇒ 42
22/7 ⇒ 3 1/7
3.1415 ⇒ 3.1415
+ ⇒ #<procedure:+>
(+ 5 3) ⇒ 8
'(+ 5 3) ⇒ (+ 5 3)
'(a b c d) ⇒ '(a b c d)
'(2 3) ⇒ '(2 3)
(2 3) ⇒ error; 2 is not procedure
```

(+ 1 2 3 4) => 10
(*) => 1 <= bool values
(+) => 0

procedure

<= making it not a procedure

Scheme: Elements

- Identifiers
  - cannot start with a character that may start a number:
    `digit`, `+`, `-`, `.`
  - case is important
- Numbers: integers: `-1234`; ratios: `1/2`; floating-point: `1.3`, `1e23`; complex numbers: `1.3 - 2.7i`
- List constants: `'(a b c d)`
- Empty list: `'()`
- Procedure applications: `(+ (* 3 5) 12)`
- Boolean values: `#t` (true), `#f` (false)
  - Any object different from `#f` is true

Scheme: Elements

- Vectors

  ```
  #(this is a vector of symbols)
  ```

- Strings

  ```
  "this is a string”
  ```

- Characters

  ```
  #\a, #\b , #\c
  ```

- Comments:
  - `; ... end_of_line`
  - `#| ... |#`

Scheme: Functions

- Variable definitions

```
(define a 23) a  ⇒  23
```

- Function applications

```
(+ 20 10) ⇒ 30
(+ 1/4 6/3) ⇒ 9/4
(* (* 2/5 5/6) 3) ⇒ 1
```

Scheme: Functions

- Defining a function

```
(define (square x) (* x x))
(square 5) ⇒ 25
```

- Anonymous functions

```
 (lambda (x) (* x x))
((lambda (x) (* x x)) 5) ⇒ 25
```

- Named functions

```
(define square (lambda (x) (* x x)))
(square 5) ⇒ 25
```

Scheme: Quoting

- (`quote` *obj*) or
- `'`*obj*
  - tells Scheme *not* to evaluate

```
(quote (1 2 3 4 5)) ⇒ (1 2 3 4 5)
(quote (+ 3 4)) ⇒ (+ 3 4)
(quote +) ⇒ +
+ ⇒ #<procedure:+>
'(1 2 3 4 5) ⇒ (1 2 3 4 5)
'(+ (* 3 10) 4) ⇒ (+ (* 3 10) 4)
'2 ⇒ 2                ; unnecessary
2 ⇒ 2
'"hi" ⇒ "hi           ; unnecessary
"hi" ⇒ "hi"
```

Scheme: Lists

- (car *list*)
  - gives the first element
- (cdr *list*)
  - gives the list without the first element

← this single quote make it not a production

```
(car '(a b c)) ⇒ a
(cdr '(a b c)) ⇒ (b c)
(car (cdr '(a b c))) ⇒ b
```

if the quote is missed, it would calculate (a b c) and sent error msg.

=> car '(b c)

=> b

- (cons *list*)
  - constructs a list from an element and a list

```
(cons 'a '()) ⇒ (a)
(cons 'a (cons 'b (cons 'c '()))) ⇒ (a b c)
(cons 'a 'b) ⇒ (a . b)   ;improper list
```

Scheme: Lists

- (`list` obj_1 obj_2 ...)
  - constructs (proper) lists; arbitrarily many arguments

```
(list 'a 'b 'c) ⇒ (a b c)
(list) ⇒ ()
(list 'a '(b c)) ⇒ (a (b c))
```

- (`null?` *list*)
  - tests whether a list is empty

```
(null? '()) ⇒ #t
(null? '(a)) ⇒ #f
```

Scheme: Variable binding

and it would make multiple define
& one time

- `(let ((`*var val*`)...)` exp_1 exp_2 ...`)`

visible only — body

var would only be used in the body of let

- each *var* is bound to the value of the corresponding *val*
- returns the value of the final expression
- the body of `let` is the sequence exp_1 exp_2 ...
- each *var* is visible only within the body of `let`
- no order is implied for the evaluation of the expressions *val*

int i=1, j=5; let((i 1)(j 5) body)

Scheme: Variable binding

```
(let ((x 2))                 ;let x be 2 in ...
  (+ x 3)) ⇒ 5
(let ((x 2) (y 3))
  (+ x y)) ⇒ 5
(let ((a (* 4 4)))
  (+ a a)) ⇒ 32
(let ((f +) (x 2) ( y 3))
  (f x y)) ⇒ 5
(let ((+ *))
  (+ 2 5)) ⇒ 10
(+ 2 5) ⇒ 7 ; + unchanged outside previous let
```

2

<- two bindings.

<= let plus sign indicates multiple function.

+ here is only plus sign,

not a multiple.

Scheme: Variable binding

```
(let ((x 1))
  (let ((y (+ x 1)))          ;nested lets
    (+ y y))) ⇒ 4
(let ((x 1))
  (let ((x (+ x 1)))          ;new variable x
    (+ x x))) ⇒ 4
```

a list of bindings.

body of

body

=> (+(+x 1)(+x 1))

=> (+(+1 1)(+1 1))

=> (+ 2 2) => 4.

old x

new x   old x

int x

x

int x

x

Scheme: Variable binding

```
(let ((x1 1))
  (let ((x2 (+ x1 1))) ; indices show bindings
    (+ x2 x2))) ⇒ 4
(let ((x1 1) (y1 10))
 (let ((x2 (+ y1 (* x1 1))))
  (+ x2 (- (let ((x3 (+ x2 y1)) (y2 (* y1 y1)))
   (- y2 x3)) y1))))) ⇒ 80
(let ((sum (lambda (ls)
             (if (null? ls)
                 0
                 (+ (car ls) (sum (cdr ls)))))))
  (sum '(1 2 3 4 5)))
```

error: undefined. the sum here is not visible to the definition, so it cannot be used

body

Scheme: Variable binding

- `(let*` `((`*var val*`)…)` exp_1 exp_2 `…)`
- similar with `let`
- each *val* is within the scope of variables to its left
- the expressions *val* are evaluated from left to right

if using let, it is incorrect.
let* makes all definitions visible

```
(let* ((x 10) (y (- x 4)))
  (* y y)) ⇒ 36
```

```
(let ((x 10) (y (- x 4)))
  (* y y))
```

Scheme: Variable binding

recursive let

- (`letrec` ((*var* *val*)…) exp_1 exp_2 …)
  - each *val* is within the scope of all variables
  - no order is implied for the evaluation of the expressions *val*

```
(letrec ((sum (lambda (ls)
               (if (null? ls)
                   0
                   (+ (car ls) (sum (cdr ls)))))))
  (sum '(1 2 3 4 5))) ⇒ 15
```

let (x y) (y x)

- `let` – for independent variables
- `let*` – linear dependency among variables
- `letrec` – circular dependency among variables

e.g. dependent on itself

Scheme: Variable binding

```
(letrec ((even?  (lambda (x)
         if x = 0 return true   (or (= x 0)
         else odd?(x-1)             (odd?  (- x 1)))))
         (odd?  (lambda (x)
         if x = 0 return false  (and (not (= x 0))
         else even?(x-1)             (even?  (- x 1))))))
  (list (even? 132) (odd? 2)))  ⇒  '(#t, #f)
```

Scheme: Functions

- (`lambda` *formals* *exp*$_1$ *exp*$_2$ …)

  params body

  - returns a function

lambda x <= not in list

lambda (x) <= in a list

- *formals* can be:
- A *proper list* of variables (*var*$_1$ … *var*$_n$)
  - then exactly *n* parameters must be supplied, and each variable is bound to the corresponding parameter

```
((lambda (x y) (* x (+ x y))) 7 13) ⇒ 140
```

- A *single* variable *x* (not in a list): then *x* is bound to a list containing all actual parameters

```
((lambda x x) 1 2 3) ⇒ (1 2 3)
((lambda x (sum x)) 1 2 3 4) ⇒ 10
```

Scheme: Functions

- An *improper list* terminated with a variable, `(`var_1 … var_n `.` var`)`, then at least n parameters must be supplied and var_1 … var_n will be bound to the first n parameters and var will be bound to a list containing the remaining parameters

```
((lambda (x y . z) (list x y z)) 1 2 3 4)
    ⇒ (1 2 (3 4))
```

two or more argument

x y z: a list

Scheme: Assignments

- `(set! var exp)` imperative
  - assigns a new value to an existing variable
  - this is not a new name binding but new value binding to an existing name

```
(let ((x 3) (y 4))
  (set! x 10)
    (+ x y))  ⇒  14
```

update x to 10

Scheme: Assignments

```
(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))
      (list root1 root2))))
```

→ $ax^2+bx+c=0$

$x = \frac{-b \pm \sqrt{b^2-4ac}}{2a}$

minusb (b) => -b

```
(quadratic-formula 1 -3 2) ⇒ (2 1)
```

Scheme: Assignments

- Can be done without `set!`

```
(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)))
        (list root1 root2)))))

(quadratic-formula 1 -3 2) ⇒ (2 1)
```

Scheme: Assignments

- Cannot be done without `set!`
  - the following version of cons, `cons-new`, counts the number of times it is called in the variable cons-count

```
(define cons-count 0)
(define cons-new
      (let ((old-cons cons))
        (lambda (x y)
          (set! cons-count (+ cons-count 1))
          (old-cons x y))))
(cons-new 'a '(b c))
cons-count ⇒  1
(cons-new 'a (cons-new 'b (cons-new 'c '())))
cons-count ⇒  4
```

Scheme: Sequencing

- `(begin` exp_1 exp_2 ...`)`
  - exp_1 exp_2 ... are evaluated from left to right
  - used for operations causing side effects
  - returns the result of the last expression

Scheme: Sequencing

```
(define quadratic-form
 (lambda (a b c)
  (begin
   (define root1 0) (define root2 0)
   (define minusb 0) (define radical 0) (define
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))
   (list root1 root2))))
(quadratic-form 1 -3 2)  ⇒ '(2 1)
```

Scheme: Conditionals

- (if *test consequent alternative*)
  - returns the value of consequent or alternative depending on test

```
(define abs
   (lambda (x)
      (if (< x 0)    if
          (- 0 x)    true
          x)))       false
(abs 4) ⇒ 4
(abs -5) ⇒ 5
```

Scheme: Conditionals

- `(not` *obj*`)`
  - returns `#t` if obj is false and `#f` otherwise

```
(not #f) ⇒ #t
(not 'a) ⇒ #f
(not 0) ⇒ #f
```

Scheme: Conditionals

- `(and` *exp* ...`)`
  - evaluates its subexpressions from left to right and stops immediately if any expression evaluates to false
  - returns the value of the last expression evaluated

```
(and #f 4 6 'a) ⇒ #f
(and '(a b) 'a 2) ⇒ 2
(let ((x 5))
  (and (> x 2) (< x 4))) ⇒ #f
```

← true

Scheme: Conditionals

- (`or` *exp* ...)
  - evaluates its subexpressions from left to right and stops immediately if any expression evaluates to true
  - returns the value of the last expression evaluated

```
(or #f 4 6 'a) ⇒ 4
(or '(a b) 'a 2) ⇒ (a b)
(let ((x 3))
  (or (< x 2) (> x 4))) ⇒ #f
```

Scheme: Conditionals

switch in C

- (cond *clause*$_1$ *clause*$_2$ …)
  - evaluates the test of each clause until one is found true or all are evaluated

```
(define memv
  (lambda (x ls)
(cond
  ((null?  ls) #f)
  ((eqv?  (car ls) x) ls)
  (else (memv x (cdr ls))))))
(memv 'a '(d a b c)) ⇒ '(a b c)
(memv 'a '(b b c)) ⇒ #f
```

Scheme: Recursion, iteration, mapping

- `(let` *name* `((`*var* *val*`)...)` exp_1 exp_2 ...`)`
  - this is named let
  - it is equivalent with

`((letrec ((`*name* `(lambda (`*var* ...`)` exp_1 exp_2 ...`)))`

name`)`

val ...`)`

Scheme: Recursion, iteration, mapping

```
(define divisors
  (lambda (n)
    (let f ((i 2))
           switch
      (cond
        ((>= i n) '())  checks if n
        ((integer?  (/ n i))  checks if n is dividable by i
         (cons i (f (+ i 1))))
        (else (f (+ i 1)))))))
(divisors 5) ⇒ '()
(divisors 12) ⇒ '(2 3 4 6)
```

Scheme: Recursion, iteration, mapping

- `(do ((`*var val update*`)...) (`*test res* ...`)` *exp* ...`)`
  - variables *var*... are are initially bound to *val*... and rebound on each iteration to *update*...
  - stops when *test* is true and returns the value of the last *res*
  - when *test* is false, it evaluates *exp*..., then *update*...; new bindings for *var*... are created and iteration continues

Scheme: Recursion, iteration, mapping

```
(define factorial
  (lambda (n)
    (do ((i n (- i 1)) (a 1 (* a i)))
      ((zero?  i) a))))
(factorial 0) ⇒ 1
(factorial 1) ⇒ 1
(factorial 5) ⇒ 120
```

Scheme: Recursion, iteration, mapping

```
(define fibonacci
 (lambda (n)
  (if (= n 0) 1
   (do ((i n (- i 1))(a1 1 (+ a1 a2))(a2 0 a1))
                                       ((= i 0) a1)))))
(fibonacci 0) ⇒ 1
(fibonacci 1) ⇒ 1
(fibonacci 2) ⇒ 2
(fibonacci 3) ⇒ 3
(fibonacci 4) ⇒ 5
```

Scheme: Recursion, iteration, mapping

- (`map` *procedure* $list_1$ $list_2$ ...)
  - applies *procedure* to corresponding elements of the lists $list_1$ $list_2$ ... and returns the list of the resulting values
  - procedure must accept as many arguments as there are lists
  - the order is not specified

```
(map abs '(1 -2 3 -4 5 -6)) ⇒ (1 2 3 4 5 6)
(map (lambda (x y) (* x y))
  '(1 2 3 4) '(5 6 7 8)) ⇒ (5 12 21 32)
```

Scheme: Recursion, iteration, mapping

- `(for-each` *procedure* $list_1$ $list_2$ ...`)`
  - similar to map
  - does not create and return a list
  - applications are from left to right

```
(let ((same-count 0))
  (for-each
   (lambda (x y)
     (if (= x y)
         (set! same-count (+ same-count 1))
         '()))
   '(1 2 3 4 5 6) '(2 3 3 4 7 6))
  same-count) ⇒ 3
```

Scheme: Pairs

- `cons` builds a pair (called also *dotted pair*)
- both proper and improper lists can be written in dotted notation
- a list is a chain of pairs ending in the empty list ()

- proper list: `cdr` of the last pair is the empty list
  - *x* is a proper list if there is *n* such that `cdr`^{n}(x) = `'()`
- improper list: `cdr` of the last pair is anything other than ()

list b

```
(cons 'a '(b)) ⇒ '(a b)      ; proper
(cons 'a 'b) ⇒ '(a . b)      ; improper
(cdr (cdr (cdr '(a b c)))) ⇒ '()
(cdr (cdr '(a b . c))) ⇒ 'c
```

b

when the tail is not a list, it is improper.

Scheme: Predicates

- (`boolean?` *obj*)
  - `#t` if *obj* is either `#t` or `#f`; `#f` otherwise
- (`pair?` *obj*)
  - `#t` if *obj* is a pair; `#f` otherwise

```
(pair? '(a b)) ⇒ #t
(pair? '( a . b)) ⇒ #t
(pair? 2) ⇒ #f
(pair? 'a) ⇒ #f
(pair? '(a)) ⇒ #t
(pair? '()) ⇒ #f
```

Scheme: Predicates

- (char? *obj*) - #t if *obj* is a character, else #f
- (string? *obj*) - #t if *obj* is a string, else #f
- (number? *obj*) - #t if *obj* is a number, else #f
- (complex? *obj*) - #t if *obj* is complex, else #f
- (real? *obj*) - #t if *obj* is a real number, else #f
- (integer? *obj*) - #t if *obj* is integer, else #f
- (list? *obj*) - #t if *obj* is a list, else #f
- (vector? *obj*) - #t if *obj* is a vector, else #f
- (symbol? *obj*) - #t if *obj* is a symbol, else #f
- (procedure? *obj*) - #t if *obj* is a function, else #f

Scheme: Input / Output

- `(read)`
  - returns the next object from input
- `(display` *obj*`)`
  - prints *obj*

```
(display "compute the square root of:")
  ⇒ compute the square root of: 2
(sqrt (read))
 ⇒ 1.4142135623730951
```

Scheme: Deep binding

takes no argument

local

```
(define A
 (lambda (i P)
  (let ((B (lambda () (display i) (newline))))
    (cond ((= i 4) (P))
          ((= i 3) (A (+ i 1) P))
          ((= i 2) (A (+ i 1) P))
          ((= i 1) (A (+ i 1) P))
          ((= i 0) (A (+ i 1) B))))))
(define C (lambda () 10))
(A 0 C) ⇒ 0
```

Scheme: Deep binding

```
(define A
 (lambda (i P)
  (let ((B (lambda () (display i) (newline))))
    (cond ((= i 4) (P))
          ((= i 3) (A (+ i 1) P))
          ((= i 2) (A (+ i 1) B))
          ((= i 1) (A (+ i 1) P))
          ((= i 0) (A (+ i 1) B))))))
(define C (lambda () 10))
(A 0 C) ⇒ 2
```

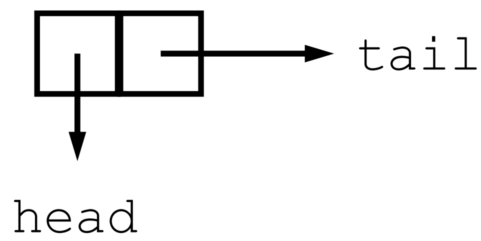
Scheme: Storage allocation for lists

- Lists are constructed with `list` and `cons`
  - `list` is a shorthand version of nested cons functions

```
(list 'apple 'orange 'grape)
       ⇒ '(apple orange grape)
(cons 'apple (cons 'orange (cons 'grape '())))
       ⇒ '(apple orange grape)
```

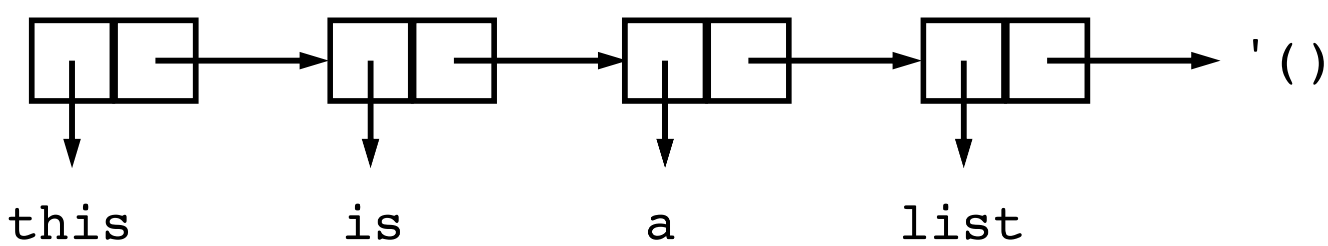
Scheme: Storage allocation for lists

- Memory allocation with `cons`
  - cell with pointers to head (`car`) and tail (`cdr`):



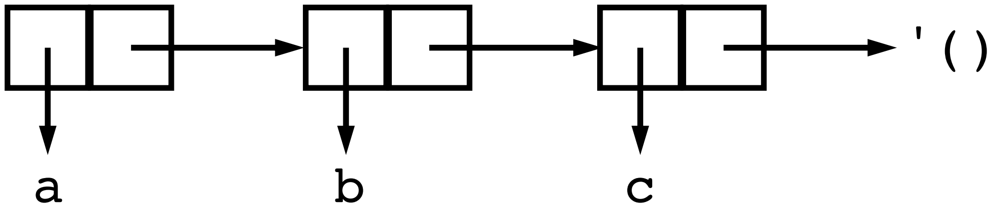
- Example

```
(cons 'this (cons 'is (cons 'a (cons 'list '()))))
```

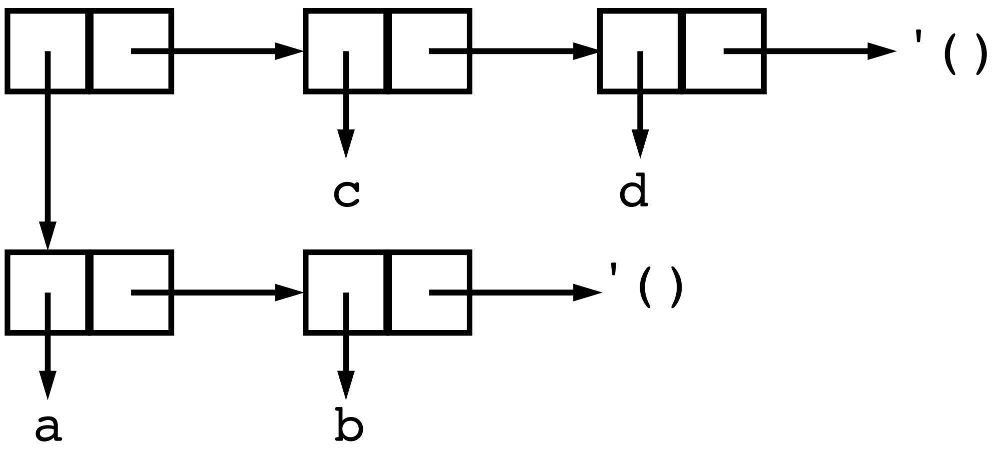


Scheme: Storage allocation for lists

(cons 'a '(b c)) ⇒ '(a b c)



(cons '(a b) '(c d)) ⇒ '((a b) c d)



Scheme: Equality

- `(eq?` obj_1 obj_2`)`
  - returns `#t` if obj_1 and obj_2 are identical, else `#f`
  - implementation as fast as possible
- `(eqv?` obj_1 obj_2`)`
  - returns `#t` if obj_1 and obj_2 are equivalent, else `#f`
  - similar to `eq?` but is guaranteed to return `#t` for two exact numbers, two inexact numbers, or two characters with the same value
- `(equal?` obj_1 obj_2`)`
  - returns `#t` if obj_1 and obj_2 have the same structure and contents, else `#f`

Scheme: Equality

```
(eq? 'a 3) ⇒ #f
(eqv? 'a 3) ⇒ #f
(equal? 'a 3) ⇒ #f

(eq? 'a 'a) ⇒ #t
(eqv? 'a 'a) ⇒ #t
(equal? 'a 'a) ⇒ #t

(eq? #t (null? '())) ⇒ #t
(eqv? #t (null? '())) ⇒ #t
(equal? #t (null? '())) ⇒ #t

(eq? 3.4 (+ 3.0 .4)) ⇒ #f
(eqv? 3.4 (+ 3.0 .4)) ⇒ #t
(equal? 3.4 (+ 3.0 .4)) ⇒ #t
```

eq: comparing same obj in memory

eqv: eq + #t for same primitive values, i.e. int, ...

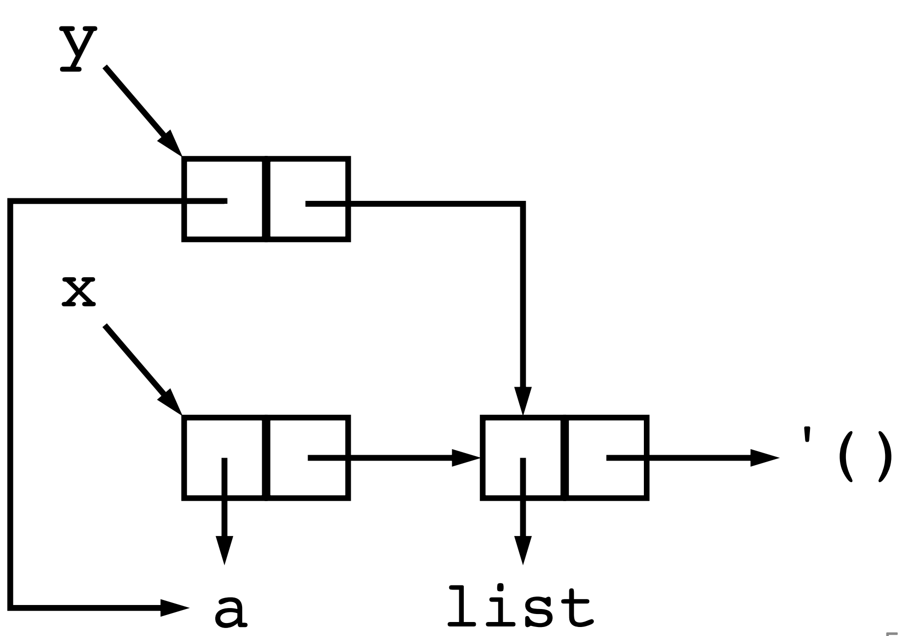
equal: eqv + comparing lists, vectors,

← not identical format

same result value.

Scheme: Equality

```
(eq? '(a) '(a)) ⇒ #f
(eqv? '(a) '(a)) ⇒ #f
(equal? '(a) '(a)) ⇒ #t

(define x '(a list))
(define y (cons (car x) (cdr x)))
(eq? x y) ⇒ #f
(eqv? x y) ⇒ #f
(equal? x y) ⇒ #t
```



Scheme: List searching

- (`memq` *obj list*)

  (`memv` *obj list*)

  (`member` *obj list*)
  - return the first tail of list whose car is equivalent to *obj* (in the sense of `eq?`, `eqv?`, or `equal?` resp.) or `#f`

```
(memq 'b '(a b c)) ⇒ '(b c)
```

Scheme: List searching

- (assq *obj list*)

  (assv *obj list*)

  (assoc *obj list*)
  - an *association list (alist)* is a proper list whose elements are key-value pairs (*key* . *value*)
  - return the first element of *alist* whose `car` is equivalent to *obj* (in the sense of `eq?`, `eqv?`, or `equal?` resp.) or `#f`

```
(assq 'b '((a . 1) (b . 2))) ⇒ '(b . 2)
(assq 'c '((a . 1) (b . 2))) ⇒ #f
(assq 2/3 '((1/3 . a) (2/3 . b))) ⇒ '(2/3 . b)
(assq 2/3 '((1/3 a) (2/3 b))) ⇒ '(2/3 b)
```

Scheme: Evaluation order

- λ-calculus:
  - applicative order (parameters evaluated before passed)
  - normal order (parameters passed unevaluated)
- Scheme uses applicative order
  - applicative may be faster
  - in general, either one can be faster

Scheme: Evaluation order

- Example: applicative order is faster

```
(double (* 3 4))
⇒ (double 12)
⇒ (+ 12 12)
⇒ 24

(double (* 3 4))
⇒ (+ (* 3 4) (* 3 4)) ⇒ (+ 12 (* 3 4))
⇒ (+ 12 12)
⇒ 24
```

Scheme: Evaluation order

- Example: normal order is faster

```
(define switch (lambda (x a b c)
  (cond ((< x 0) a)
        ((= x 0) b)
        ((> x 0) c))))

(switch -1 (+ 1 2) (+ 2 3) (+ 3 4))
⇒ (switch -1 3 (+ 2 3) (+ 3 4))
⇒ (switch -1 3 5 (+ 3 4))
⇒ (switch -1 3 5 7)
⇒ (cond ((< -1 0) 3)
        ((= -1 0) 5)
        ((> -1 0) 7))
⇒ 3
```

Scheme: Evaluation order

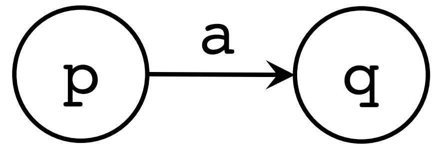
- Example: normal order is faster (cont'd)

```
(switch -1 (+ 1 2) (+ 2 3) (+ 3 4))
⇒ (cond ((< -1 0) (+ 1 2))
        ((= -1 0) (+ 2 3))
        ((> -1 0) (+ 3 4))
⇒ (cond (#t (+ 1 2))
        ((= -1 0) (+ 2 3))
        ((> -1 0) (+ 3 4))
⇒ (+ 1 2)
⇒ 3
```

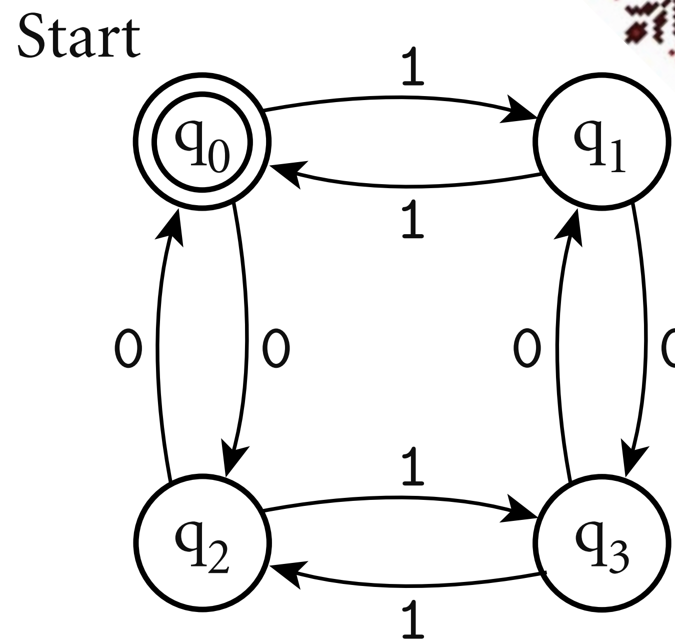
Scheme: Higher-order functions

```
(define mcompose
 (lambda (flist)
  (lambda (x)
   (if (null? (cdr flist))
    ((car flist) x)
    ((car flist) ((mcompose (cdr flist)) x))))))

(define cadr
  (mcompose (list car cdr)))
(cadr '(a b c)) ⇒ ‘b

(define cadaddr
  (mcompose (list car cdr car cdr cdr)))
(cadaddr '(a b (c d))) ⇒ 'd
```

Scheme: DFA simulation

- DFA description:
  - start state
  - transitions: list of pairs
    - `((p a) q)`: 

  - final states




```
(define zero-one-even-dfa
  ‘(q0                                    ; start state
    (((q0 0) q2) ((q0 1) q1)
      ((q1 0) q3) ((q1 1) q0)
      ((q2 0) q0) ((q2 1) q3)
      ((q3 0) q1) ((q3 1) q2))            ; transition fn
    (q0)))                                ; final states
```

Scheme: DFA simulation

- DFA simulation:

```
(simulate
 zero-one-even-dfa    ; machine description
 '(0 1 1 0 1))        ; input string
⇒  '(q0 q2 q3 q2 q0 q1 reject)

(simulate
 zero-one-even-dfa  ; machine description
 '(0 1 0 0 1 0))  ; input string
⇒  '(q0 q2 q3 q1 q3 q2 q0 accept)
```

Scheme: Differentiation

- Symbolic differentiation

$$\frac{d}{dx}(c) = \frac{d}{dx}(y) = 0, \qquad c \text{ a constant}, y \neq x$$

$$\frac{d}{dx}(x) = 1$$

$$\frac{d}{dx}(u+v) = \frac{d}{dx}(u) + \frac{d}{dx}(v), \qquad u, v \text{ functions of } x$$

$$\frac{d}{dx}(u-v) = \frac{d}{dx}(u) - \frac{d}{dx}(v)$$

$$\frac{d}{dx}(uv) = u\frac{d}{dx}(v) + v\frac{d}{dx}(u)$$

$$\frac{d}{dx}\left(\frac{u}{v}\right) = \frac{v\frac{d}{dx}(u) - u\frac{d}{dx}(v)}{v^2}$$

Scheme: Differentiation

```
(define diff
  (lambda (x expr)
    (if (not (pair? expr))
        (if (equal? x expr) 1 0)
        (let ((u (cadr expr))(v (caddr expr)))
          (case (car expr)
            ((+) (list '+ (diff x u) (diff x v)))
            ((-) (list '- (diff x u) (diff x v)))
            ((*) (list '+
                           (list '* u (diff x v))
                           (list '* v (diff x u))))
            ((/) (list '/ (list '-
                                    (list '* v (diff x u))
                                    (list '* u (diff x v)))
                           (list '* v v)))
            )))))
```

Scheme: Differentiation

```
(diff 'x '3) => 0
(diff 'x 'x) => 1
(diff 'x 'y) => 0
(diff 'x '(+ x 2)) => '(+ 1 0)
(diff 'x '(+ x y)) => '(+ 1 0)
(diff 'x '(* 2 x)) => '(+ (* 2 1) (* x 0))
(diff 'x '(/ 1 x)) => '(/ (- (* x 0) (* 1 1)) (* x x))
(diff 'x '(+ (* 2 x) 1)) => '(+ (+ (* 2 1) (* x 0)) 0)
(diff 'x '(/ x (- (* 2 x) (* 1 x))))
=> '(/
      (- (* (- (* 2 x) (* 1 x)) 1) (* x (- (+ (* 2 1)
(* x 0)) (+ (* 1 1) (* x 0)))))
      (* (- (* 2 x) (* 1 x)) (- (* 2 x) (* 1 x))))
```