Review of Scheme

* Prefix notation: (E1 E2 ... En)
  - (+ 2 3); (* 3 4)
* User-defined functions
  - (define (square x) (* x x))
  - (define square (lambda (x) (* x x)))
* Booleans: #t, #f
* Conditionals
  - (if P E1 E2)
  - (cond (P1 E1) (P2 E2) ... (Pn En)
    (else E_{n+1}))
* Higher-order functions
  - Take other functions as arugments and return functions as results
  - (define (twice f x) (f (f x)))
  - (define (twiceV2 f)
    (lambda (x) (f (f x))))
* Introducing local names
  - (let ((x1 E1) ... (xn En)) E)
  - (let* ((x1 E1) ... (xn En)) E)
  - (letrec ((x1 E1) ... (xn En)) E)

* lists
  - quoting mechanism
    * (quote (1 2 3))
    * '(1 2 3)
  - lists: a series of zero or more items
    * '(); '(it seems that);
    * '((it) (seems that))
  - basic operations
    * (null? l): returns #t iff l is the null list
    * (car l): returns the first element in the list
    * (cdr l) returns the rest of l without the first element
    * if l is '((it) (seems that))
      - then (car l) is '(it)
      - then (cdr l) is '((seems that))
      - then (car (car l)) is "it"
    * (cons h t): returns a list whose car is h and whose cdr is t
      - (cons 'a '(b c d)) -> '(a b c d)

List manipulation functions

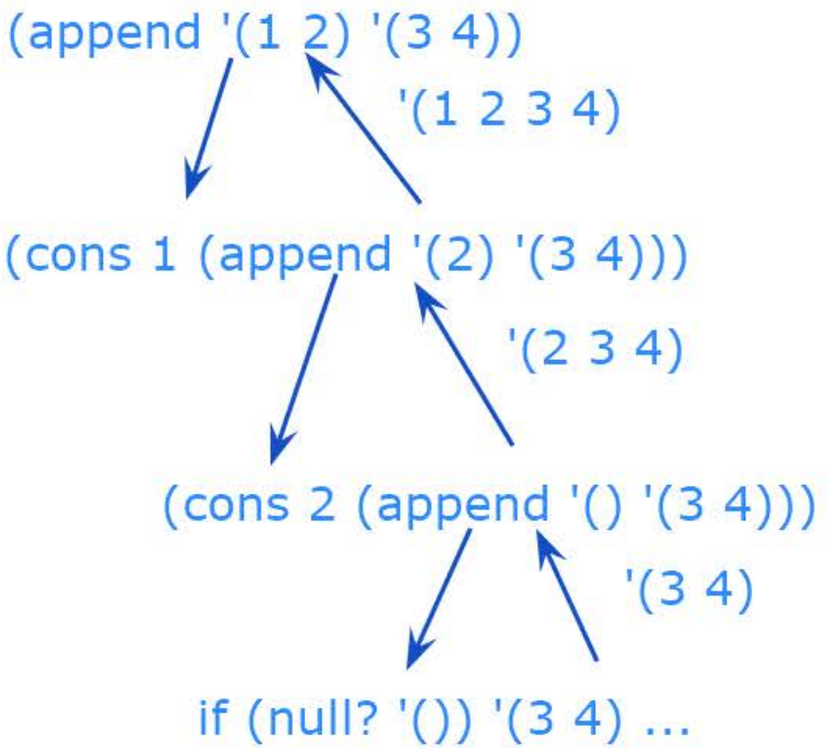
* compute the length of a list
  - (length l): return the length of list l

```
(define (length l)
    (if (null? l) 0
        (+ 1  (length (cdr l)))))
```

  - case analysis and recursion
    * two cases: (1) when l is empty; (2) when l is not empty, which means it has a head and tail
    * when l is not empty, do recursion on the tail of the list and compute the final result based on the recursive call

* (append l1 l2): return the concatenation of l1 and l2
  - (append '(1 2) '(3 4)) --> '(1 2 3 4)

```
(define (append l1 l2)
    (if (null? l1) l2
        (cons (car l1) (append (cdr l1) l2))))
```



'(1 2 3 4)
the same as (cons 1 (cons 2 (cons 3 (cons 4 '()))))

* Project 2; due on the 28th
* Reminder about the interaction
  - use the chat room to provide your answers
  - Q&A
  - canvas to do clicker questions
* Review about scheme
  - List processing: null?, car, cdr, cons
  - Case analysis and recursion
  - (define (length l)
      (if (null? l) 0 (+ 1 (length (cdr l)))))
  - (define (append l1 l2)
      (if (null? l1) l2
        (cons (car l1) (append (cdr l1) l2))))

* Exercise: define a listSum, which takes a list of numbers and returns the sum of the number; if the input list is the empty list, return 0

```
(define (listSum l)
    (if (null? l) 0
        (+ (car l) (listSum (cdr l)))))
```

* the member function: (member? a l) returns #t iff a is a member of l
  - e.g., (member? 3 '(1 2 3)) --> #t
    (member? 4 '(1 2 3)) --> #f
    (member? '(a) '((a) (b) (c))) --> #t
    (member? 'a '((a) (b) (c))) --> #f

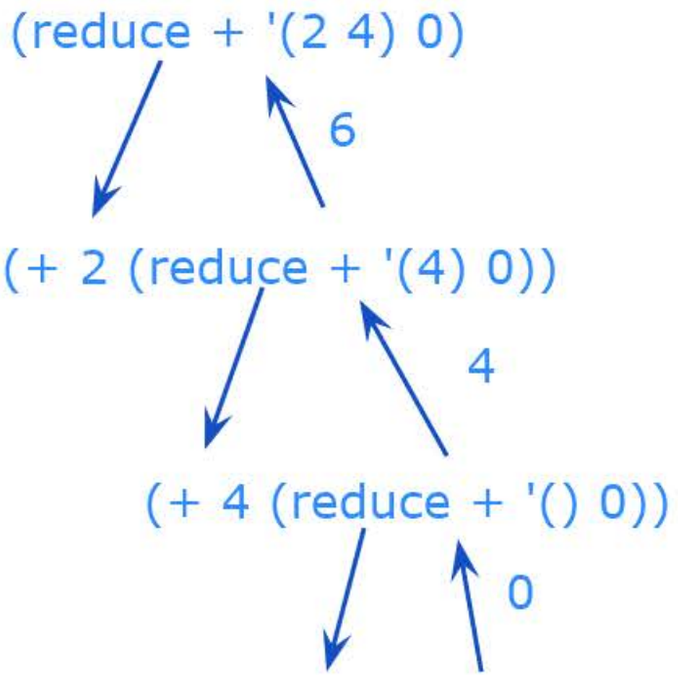
```
(define (member? a l)
    (cond ((null? l) #f)
          ((equal? a (car l)) #t)
          (else (member? a (cdr l)))))
```

* the map function: (map f l) returns a new list whose elements are the result of calling f on elements in l
  - (map plusOne '(1 2 3)) --> '(2 3 4)
  - (map square '(1 2 3)) --> '(1 4 9)

```
(define (map f l)
    (if (null? l) '()
        (cons (f (car l))  (map f (cdr l)))))
```

* reduce: to reduce a list of items into a single item according to some function
  - (reduce f l ini)
  - (reduce + '(2 4 6) 0) --> 0+2+4+6 = 12
  - (reduce + '() 0) --> 0
  - (reduce * '(2 4 6) 1) --> 1*2*4*6 = 48

```
(define (reduce f l ini)
     (if (null? l) ini
         (f (car l) (reduce f (cdr l) ini))))
```



* Review about list processing
  - Basics ops: null?, car, cdr, cons
  - Case analysis and recursion
  - (define (member? a lst)
      (cond ((null? lst) #f)
            ((equal? a (car lst)) #t)
            (else (member? a (cdr lst)))))
  - (map f l): returns a new list whose elements are the results of calling f on the elements of l
    * (map square '(2 3 4)) --> '(4 9 16)
  - (reduce f l ini): reduces the list l into a single value based on the ini value and f
    * (reduce + '(2 3 4) 0) --> 2+3+4+0=9

* Hadoop Example of using MapReduce
  - n webpages: (w1 w2 ... wn)
  - goal: want to compute the number of webpages that contain the "virus"
  - step 1:
    (map f '(w1 w2 ... wn))
    * f takes a webpage and returns 1 if the page contains the keyword; otherwise, return 0
    * let lst = (map f '(w1 w2 ... wn))
      - e.g., '(1 0 1 ... 1)
  - step 2: (reduce + lst 0)
  - easily parallelized
    * step 1: divide the list of pages into two halves: (w1 ... wk); (w_{k+1} ... wn)
      - server 1: l1 = (map f '(w1 ... wk))
      - server 2: l2 = (map f '(w_{k+1}, ... wn)
      - lst = (append l1 l2)
        - (cons l1 l2) wrong
    * step 2: partition lst into l1 and l2
      - server 1: n1 = (reduce + l1 0)
      - server 2: n2 = (reduce + l2 0)
      - n1+n2 would be the final result

* Association lists
  - an association list is just a list of pairs
    - e.g., '((a 1) (b 2) (c 3))
  - a data structure that maps from keys to values
  - in compilers, this can be used to implement symbol tables: keys are symbols and values are bindings for the symbols

- two operations
  * (bind k v al): construct a new association list with new entry (k v) as well as the old entries in al
  * (lookup k al): perform a lookup in al to see whether there is an entry with key k; if there is, return the key; otherwise, return #f
- (bind k v al)
  - e.g., (bind 'd 4 '((a 1) (b 2) (c 3)))
    --> '((d 4) (a 1) (b 2) (c 3))
  - design choice
    * what happens if k is already in al?
      (bind 'a 10 '((a 1) (b 2) (c 3)))
    * choice 1
      - remove the old entry and add the new entry
      - '((a 10) (b 2) (c 3))
    * choice 2
      - keep the old entry and add the new entry to the front; would work as long as the look up operation always do the lookup from the beginning
      - '((a 10) (a 1) (b 2) (c 3))
      - easier to program in function languages; the choice we are going to use

```
(define (bind k v al)
    (cons (list k v) al))

(define (lookup k al)
    (cond ((null? al) #f)
          ((equal? k (caar al))
           (car al))
          (else (lookup k (cdr al)))))
```

* Currying and uncurrying
  - how to write multi-parameter functions?
  - uncurried style: the function takes all parameters at once
    * (define (twice f x) (f (f x)))
    * (define (add n1 n2) (+ n1 n2))
  - curried style: the function takes one parameter at a time
    * (define (twiceV2 f)
        (lambda (x) (f (f x))))
    * (define (addN n1)
        (lambda (n2) (+ n1 n2)))

- In general,
  * uncurried style:
    (lambda (x1 ... xn) e)
  * curried style:
    (lambda (x1)
      (lambda (x2)
        ...
          (lambda (xn) e) ...))
- uncurried style: it requires all arguments provides when the function gets called
  * (add 3 4) fine; (add 3) error
- curried style: caller can do partial evaluation, providing one argument at a time
  * (addN 3) fine
- curried style provides more flexibility
  (map (add 3) '(1 2 3)) error
  (map (addN 3) '(1 2 3)) work and produce '(4 5 6)

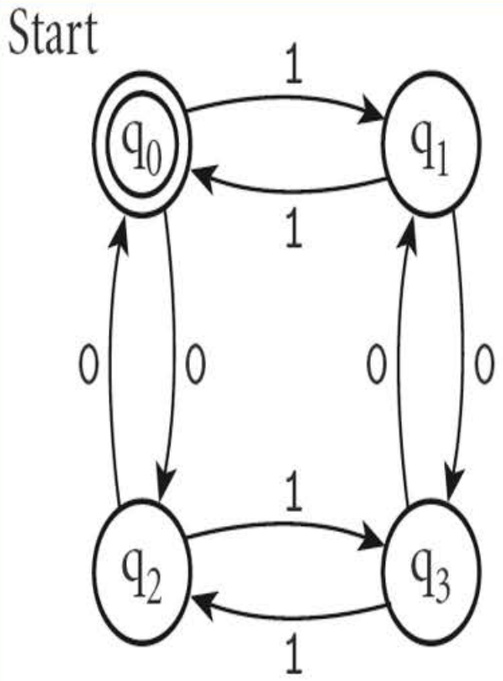
* proj2 due on Sat
* Review on Scheme
  - list processing
    * case analysis and recursion
    * e.g., length, append, member?, map, reduce
  - Association lists
    * a list of pairs of keys and values
    * '((a 1) (b 2) (c 3))
    * (bind k v al): adds a new entry (k v) to al
    * (lookup k al)
      - in standard scheme, it's called assoc
  - currying
    * uncurried function: takes all parameters at once
      - (lambda (x1 ... xn) e)
    * curried function: takes one parameter at a time
      - (lambda (x1)
          (lambda (x2)
            ...
            (lambda (xn) e) ...))

  - uncurried fun:
    (define (add m n) (+ m n))
    * Ocaml, the type for add would be
      int*int -> int
  - curried fun
    (define (addN m)
      (lambda (n) (+ m n)))
    * Ocaml: the type of add N would be
      int -> (int -> int)
  - conversion from an uncurried fun to a curried function
    * let's assume f is a two-argument uncurried fun
    (define (curry2 f)
      (lambda (x) (lambda (y) (f x y))))
    (curry2 add)

    (map (add 2) '(1 2 3 4)) --> error
    (map ((curry add) 2) '(1 2 3 4)) -->
      '(3 4 5 6)
    * Ex: write curry3, which takes an uncurried function f with three parameters and converts f into the curried version

    * (define (curry3 f)
        (lambda (x)
          (lambda (y)
            (lambda (z) (f x y z)))))
  - convert a curried function to a uncurried one
    * (define (uncurry2 f)
        (lambda (x y) ((f x) y)))
    * (define (uncurry3 f)
        (lambda (x y z) (((f x) y) z)))

* An extended example on Scheme programming
  - textbook ch11.3: DFA simulation
  - DFA
    * start state
    * the transition function: takes the current state and the next input character and returns the next state
    * a set of final states



* takes even numbers of zeros and ones:
  - e.g., 0101
* encode the DFA, use a list with three elements
  - first: start state
  - second: transition function as an associate list mapping from a pair of current states and next chars to new states
  - third: a list of final states

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

* Review
  - Currying and uncurrying
    * curried fun: takes one parameter at a time
    * uncurried fun: takes all parameters at once
    * conversion between curried and uncurried funs
  - DFA simulation
    * Encode the DFA as a list: (1) start state; (2) the transition function encoded as an association list; (3) a list of final states
    * move: running the DFA for one step;
      - take the DFA and the next input char and return a new DFA
      - the new DFA have the next state as the new start state and also have the same transition fun and list of final states as the old dfa, when the transition is legal
      - the new DFA has the start state being a special error state

```
(define simulate
    (lambda (dfa input)
        (letrec ((helper
                    (lambda (mvs d2 i)
                        (let ((c (current-state d2)))
                            (if (null? i) (cons c mvs)
                                (helper (cons c mvs) (move d2 (car i)) (cdr i)))))))
            (let ((moves (helper '() dfa input)))
                (reverse (cons (if (is-final? (car moves) dfa) 'accept 'reject)
                    moves))))))
```

* let d1 be the result of (move d 1)