Tutorial 4

prepared by Anya Tafliovich
based on material provided by Susan Stevenson

June 11, 2003

```
;; (member e lst)
;; returns #t if e is in lst, () otherwise.
;; e is an atom and lst is a non-nested list.

(define (member e lst)
   (cond ((null? lst)  ())
         ((eq? e (car lst))  #t)
         (else  (member e (cdr lst)))))
```

OR:

```
(define (member e lst)
   (and (not (null? lst))
        (or (eq? e (car lst))
            (member e (cdr lst)))))
```

```
;; (union lst1 lst2)
;; returns a list consisting of all elements in
;; either lst1 or lst2, with no duplicates.
;; lst1 and lst2 are flat lists

(define (union lst1 lst2)
   (cond ((null? lst1)  lst2)
         ((member (car lst1) lst2)
           (union (cdr lst1) lst2))
         (else  (cons (car lst1)
                      (union (cdr lst1) lst2)))))


; NOTE: An alternative solution to the above is to append the
;       lists, then remove the duplicates.

; BUT:  It is less efficient: it means traversing the first list
;       (in append) and then traversing the combined list and comparing each
;       element of the combined list to each element of the rest of the list
;       (to remove duplicates).  In the above solution, you only traverse the
;       first list comparing each element to each element of the second list.
```

```
;; (intersection lst1 lst2)
;; returns a list consisting of all the
;; elements that are in both lst1 and lst2
;; lst1 and lst2 are flat lists

(define (intersection lst1 lst2)
   (cond ((null? lst1)  ())
         ((member (car lst1) lst2)
           (cons (car lst1)
                 (intersection (cdr lst1) lst2)))
         (else  (intersection (cdr lst1) lst2))))



;;Exercise:
;; (difference lst1 lst2)
;; returns a list consisting of all elements
;; that are in lst1 and NOT in lst2.
;; lst1 and lst2 are flat lists
```

Higher-Order Procedures (HOPs)

```
;; (reduce op lst id)
;; applies the binary operator op to the elements of
;; list lst right-associatively, or returns id
;; (the identity element) if lst is empty.
;; Pre: op is a binary procedure,
;;      lst is a list of valid arguments to op,
;;      id is the identity value for op,
;;         i.e., (op x id) => x for all x that are valid arguments to op.


(define (reduce op lst id)
   (cond ((null? lst)  id)
         (else  (op (car lst)
                    (reduce op (cdr lst) id)))))
```

```
;; (intersect-all lst1 lst2)
;; returns a list of elements that corresponding sublists
;; of lst1 and lst2 have in common, with no duplicates.
;; (lst1 and lst2 are lists of lists, with the same number of sublists.)
;;
;;
;;  Example:  If l1 is '((1 2 4 5) (2 4 5) (3 8))
;;               l2 is '((1 3 5)   (2 5)   (1 5 8))
;;
;;               (intersect-all l1 l2) => (1 2 5 8)
;;
;;  Note: order of elements in result doesn't matter.

;; Helper function:
;; return a list consisting of all elements of lst, with no duplicates

(define (rm-dups lst)
   (cond ((null? lst) ())
         ((member (car lst) (cdr lst))
           (rm-dups (cdr lst)))
         (else  (cons (car lst)
                      (rm-dups (cdr lst))))))




(define (intersect-all lst1 lst2)
  (rm-dups (apply append (map intersection lst1 lst2))))
```

```
;; (union-all lst1 lst2)
;; returns a list of elements consisting of the union of corresponding
;; sublists of lst1 and lst2, with no duplicates.
;; (lst1 and lst2 are lists of lists, with the same number of sublists.)
;;
;;   Example:  If l1 is '((1 2 4 5) (2 4 5) (3 8))
;;                l2 is '((1 3 5)   (2 5)   (1 5 8))
;;
;;               (union-all l1 l2) => (1 2 3 4 5 8)
;;
;;   (Note: order of elements in result doesn't matter.

(define (union-all lst1 lst2)
   (rm-dups (apply append (map union lst1 lst2))))


; OR:

(define (union-all lst1 lst2)
  (rm-dups (union (apply append lst1) (apply append lst2))))


; OR:

(define (union-all lst1 lst2)
   (reduce union (map union lst1 lst2) '()))
```

```
;; (in-all-lists e lst)
;; returns #t if e is in every sublist of lst, () otherwise.
;; (lst is a list of lists.)
;; Note: If lst is  empty, then in-all-lists should return ().
;;
;;   Example: (in-all-lists 5 '((1 3 5) (2 5) (1 5 8))) => #t
;;            (in-all-lists 1 '((1 3 5) (2 5) (1 5 8))) => #f
;;            (in-all-lists 1 '()) => #f



(define (in-all-lists e lst)
   (cond ((null? lst) ())
         (else  (in-all-lists-helper e lst))))


(define (in-all-lists-helper e lst)
   (or (null? lst)
       (and (member e (car lst))
            (in-all-lists-helper e (cdr lst)))))



; OR: Using cond and 'and':

(define (in-all-lists-helper e lst)
   (cond ((null? lst) #t)
         (else  (and (member e (car lst))
                     (in-all-lists-helper e (cdr lst))))))


; OR: Using cond only (not so good):

(define (in-all-lists-helper e lst)
   (cond ((null? lst)  #t)
         ((member e (car lst))
           (in-all-lists-helper e (cdr lst))))
         (else ())))
```