Tutorial 4

week of February 5, 2007

1 More Recursion

```
;; (member e lst) checks whther e is a member of list lst
;; Pre: e is an atom and lst is a flat list
;; Return: #t if e is an element of lst, () otherwise.
(define (member e lst)
   (cond ((null? lst)  ())
         ((eq? e (car lst))  #t)
         (else  (member e (cdr lst)))))
```

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

```
;; (union lst1 lst2) returns a list which represents a set union
;; of elements in lst1 and elements in lst2
;; Pre: lst1 and lst2 are flat lists
;; Return: a list consisting of all elements that are in the set union
;;         of elements from lst1 and elements from lst2
(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. However, 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 which represents a set union
;;  of elements in lst1 and elements in lst2
;; Pre: lst1 and lst2 are flat lists
;; Return: a list consisting of all elements that are in the set
;;         intersection of elements from lst1 and elements from lst2
(define (intersection lst1 lst2)
   (cond ((null? lst1)  ())
         ((member (car lst1) lst2)
           (cons (car lst1)
                 (intersection (cdr lst1) lst2)))
         (else  (intersection (cdr lst1) lst2))))
```

EXERCISE: write a procedure (difference lst1 lst2), which, given two flat lists lst1 and lst2, returns a list consisting of all elements that are in lst1 and not in lst2.

2 Higher Order Procedures

1. MAP

```
    (map <procN> <list1> ... <listN>),

where <procN> is an N-ary procedure
                    (a procedure that takes N arguments)
      <list1> is (e11 e12 ... e1M), a list of M elements
      <list2> is (e21 e22 ... e2M), a list of M elements
       .....
      <listN> is (eN1 eN2 ... eNM), a list of M elements

return the following list:

       (r1 r2 ... rM),

where  r1 = (<procN> e11 e21 ... eN1)
       r2 = (<procN> e12 e22 ... eN2)
         .....
       rM = (<procN> e1M e2M ... eNM)
```

For example,

```
1 ]=>  (map null? '( 1 () 2 (3 4) () 5))
;Value 1: (() #t () () #t ())

1 ]=> (map - '(1 2 3 4) '(1 1 1 1))
;Value 2: (0 1 2 3)

1 ]=> (map cons '(1 2 3 4) '((2 3) (2 3) (2 3) (2 3)))
;Value 4: ((1 2 3) (2 2 3) (3 2 3) (4 2 3))
```

2. APPLY

```
    (apply <procN> <list>),

where <procN> is an N-ary procedure
                    (a procedure that takes N arguments)
  and <list> is (arg1 arg2 ... argN), a list of N elements

return the result of evaluating:

      (<procN> arg1 arg2 ... argN)
```

For example,

```
1 ]=> (apply null? '( () ))
;Value: #t

1 ]=> (apply - '(10 5))
;Value: 5

1 ]=> (apply cons '( a (b c d)))
;Value 5: (a b c d)

1 ]=> (apply + '(1 1 1 1 1 1 1))
;Value: 7

1 ]=> (apply append '( (1) (2) (3 4) (5) () ))
;Value 6: (1 2 3 4 5)
```

3 Using HOPs

```
;; (rm-dups lst) returns a list consisting of all elements of lst,
;;  with no duplicates
;; Pre: lst is a flat list
;; Return: a list of all elements of lst, without 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))))))

;; (intersect-all lst1 lst2) returns a list of elements that
;; corresponding sublists of lst1 and lst2 have in common,
;; with no duplicates
;; Pre: lst1 and lst2 are lists of flat 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))
;;
;;             then  (intersect-all l1 l2) is (1 2 5 8)
;;
;;  Note: order of elements in result doesn't matter.
(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
;;  set union of corresponding sublists of lst1 and lst2,
;; with no duplicates
;; Pre: lst1 and lst2 are lists of flat 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))
;;
;;             then (union-all l1 l2) is (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))))

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

```
;; (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) is 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)))))
```

Another solution to the previous problem:

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

```
;; (in-all-lists e lst) checks if e is an element of every
;;  list contained in lst
;; Pre: lst is a list of flat lists
;; Return: #t if e is an element every sublist of lst, () otherwise.
;;         if lst is empty, 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-every-list e lst))))

;; (in-every-list e lst) checks if e is an element of every
;;  list contained in lst
;; Pre: lst is a list of flat lists
;; Return: #t if e is an element every sublist of lst, () otherwise.
;;         if lst is empty, return #t.
(define (in-every-list e lst)
   (or (null? lst)
       (and (member e (car lst))
            (in-every-list e (cdr lst)))))

(define (in-every-listr e lst)
   (cond ((null? lst) #t)
         (else  (and (member e (car lst))
                     (in-every-list e (cdr lst))))))

(define (in-every-list e lst)
   (cond ((null? lst)  #t)
         ((member e (car lst))
           (in-every-list e (cdr lst))))
         (else ())))
```