Logic Programming

1. Write a rule for car of list. For example, (car (1 2 3 4) ?x) would have ?x bound to 1.
   (rule (car (?car . ?cdr) ?car))

2. Write a rule for cdr of list. For example, (cdr (1 2 3) ?y) would have ?y bound to (2 3).
   (rule (cdr (?car . ?cdr) ?cdr))

3. Define a rule for member, so that (member 4 (1 2 3 4 5)) would be satisfied, and (member 3 (4 5 6)) would not, and (member 3 (1 2 (3 4) 5)) would not.
   (rule (member ?item (?item . ?cdr)))
   (rule (member ?item (?car . ?cdr)) (member ?item ?cdr))

4. Define a rule for deep-member, so that (deep-member 3 (1 2 (3 4) 5)) would be satisfied as well.
   (rule (deep-member ?item (?item . ?cdr)))
   (rule (deep-member ?item (?car . ?cdr)) (deep-member ?item ?car))
   (rule (deep-member ?item (?car . ?cdr)) (deep-member ?item ?cdr))

5. Define a rule for reverse, so that (reverse (1 2 3) (3 2 1)) would be satisfied. Assume you have a rule for append

   (rule (reverse () ()))
   (rule (reverse (?car . ?cdr) ?reversed-ls)
       (and (reverse ?cdr ?r-cdr)
            (append ?r-cdr (?car) ?reversed-ls)))

6. Write the rule interleave so that (interleave (1 2 3) (a b c d) ?what) would bind ?what to (1 a 2 b 3 c d).

   (rule (interleave ?ls () ?ls))
   (rule (interleave () ?ls ?ls))
   (rule (interleave (?car . ?cdr) ?ls2 (?car . ?r-cdr))
       (interleave ?ls2 ?cdr ?r-cdr))

Final Review

1. Write a higher-order function, all-pairs, that takes two lists and return a list of all possible pairs of elements from the two argument lists. DO NOT use recursion. For example:

   > (all-pairs '(1 2) '(2 3 4 5))
   ((1 2) (1 3) (1 4) (1 5) (2 2) (2 3) (2 4) (2 5))

   (define (all-pairs ls1 ls2)
       (accumulate append (map (lambda (x) (map (lambda (y) (list x y)) ls1)) ls 2)))

2. Define the stream of even powers of 2, without defining procedures (no lambdas). You may use the stream ones as a building block.

   (define powers
       (cons-stream 1 (stream-map * fours powers)))
   (define fours (stream-add ones ones ones ones))

3. Write a procedure, (smallest-containing-tree tree x y) that takes in a general tree and two elements x and y, and returns the smallest subtree of tree containing both x and y. If tree does not contain x and y, return false. You can use tree-member?.

   (define (smallest-containing-tree tree x y)
       (if (and (tree-member? tree x)
                 (tree-member? tree y))
(define (smallest-containing-tree-of-forest forest x y)
  (if (null? forest)
      #f
      (or (smallest-containing-tree (car forest) x y)
          (smallest-containing-tree-of-forest (cdr forest) x y))))

4. Draw the environment diagram and figure out what the last expression returns

(define p
  (let ((x #f))
    (set! x 1)
    (lambda (n)
      (lambda ()
        (set! n (+ n 1))
        (set! x (+ x n))
        x))))

(define m 3)
(define p1 (p m))
(define p2 (p m))
(p1)
(p2)
(p1)

14

5. Analyze the following code and write down all possible answers:

(define x 10)
(parallel-execute (lambda () (set! x (+ 5 x)) (set! x (* x 3)))
  (lambda () (if (> x 16)
                 (set! x 100)
                 (set! x (- x 20))))))

100, -45, -15, 45, -10, -5, -30

6. Write a procedure, (num-sum exp) that takes in a valid Scheme expression, and returns the sum of all numbers that occurs in that expression. For example,

(num-sum (if (= 2 3) (lambda(x) (+ x 3)) 10)) => 18

(define (num-sum exp)
  (cond ((number? exp) exp)
        ((atom? exp) 0)
(else (+ (num-sum (car exp))
        (num-sum (cdr exp))))))

7. Define a procedure (trimmed ls) that takes in a list and returns the same list without the first and last element. So,

(trimmed (1 2 3 4 5 6)) ==> (2 3 4 5)
(trimmed ()) ==> ()
(trimmed (1)) ==> ()
(trimmed (1 2)) ==> ()

(define (trimmed ls)
    (cond ((<= (length ls) 2) ())
          (else (reverse (cdr (reverse (cdr ls)))))
          ))

8. Define a procedure (coolize ls) that does this:

STk> (coolize (1 2 3 4 5))
(1 (2 (3) 4) 5)
STk> (coolize (a b c d e f))
(a (b (c () d) e) f)

You may assume you are given a procedure (last ls) that returns the last element of the list. If you want even more practice, define this procedure. You can also use trimmed.

(define (last ls)
    (cond ((null? (cdr ls)) (car ls))
          (else (last (cdr ls)))))

(define (coolize ls)
    (cond ((null? ls) ())
          ((null? (cdr ls)) ls)
          (else (list (car ls)
                      (coolize (trimmed ls))
                      (last ls))))))