CS61A SUMMER 2010
FINAL REVIEW SESSION 1

George Wang, Jonathan Kotker, Seshadri Mahalingam, Steven Tang, Eric Tzeng
Derived from the notes of Chung Wu and Justin Chen,
and past CS61A review sessions (Spring 2007)

QUESTION 1.

We want to replace the root of a tree by shifting certain children up by one level successively and putting the original root at the end of the shifting. Write a procedure **demote-root** that takes a tree and a list of children index numbers as arguments and returns a new tree that has the children at those indices successively moved up by 1, with the root taking the original position of the last child.

For example, if we have the tree:

```
        3      root
       /|\
      1 2 4    level 1
     /|\
    5 6 7      level 2
```

and the list (0 2), we would move the element in the 0th index in level 1 up by one (so the number 1) and the element in the 2nd index in level 2 up by one (so the number 7) and have the root (the number 3) replace the original position of the 7, giving us:

```
        1      root
       /|\
      7 2 4    level 1
     /|\
    5 6 3      level 2
```

Remember to return a NEW tree!

QUESTION 2.

What will Scheme print? If an expression produces an error message, simply write “error”; if the value of an expression is a procedure, simple write “procedure”.

```
(keep (lambda (x) x)
      (every (lambda (y) (if (even? y) #t y))
             ‘(1 3 3 7)))                              ________________

(and or (not #f) (not not) 2)                          ________________
```

QUESTION 3.

What will Scheme print in response to the following expressions? If an expression produces an error message, simply write “error”. **Also, draw a box and pointer diagram for the value produced by each expression.**

```
(cons (list 1 3) (append (list (cons 2 3)) (list 4)))     ________________
```

```
(list (append (list 3) (cons 4 ‘())))                  ________________
```

```
(let ((x (list 1 2 3 4)))
  (set-car! (cddr x) x)
  x)                                                   ________________
```

```
(let ((x (list 1 2 3)))
  (set-car! x (cdr x))
  (set-cdr! (car x) 5)
  x)                                                      _______________
```

```
(let ((x (list 1 2 3)))
  (set-car! x (list 'a 'b 'c))
  (set-car! (cdar x) 'd)
  x)                                                      _______________
```

QUESTION 4.

```
(define (square x) (* x x))
(define (foo x y) (+ x (* y y)))
(foo (* 2 2) (square 3))
```

How many times is * called in:

Normal order _______________   Applicative order _______________

QUESTION 5.

What is the order of growth in time of the following procedure **foo**, in terms of its argument value *n*? Also, does it generate an iterative process or recursive process?

```
(define (foo n)
  (if (even? n)
      (mystery n)
      (mystery (+ n 1))))

(define (mystery x)
  (if ((x < 1) 1)
      ((even? x) (+ 1 (mystery (- x 1))
                      (mystery (- x 2))))
      (else (+ 1 (mystery (- x 2))))))
```

_____ $\Theta(1)$ _____ $\Theta(n)$ _____ $\Theta(n^2)$ _____ $\Theta(2^n)$

QUESTION 6.

Write **sent-fn**, a procedure that takes an arithmetic function and a list of sentences of numbers and returns a new list of sentences that is the result of calling the function each number in each sentence. For example:

```
> (sent-fn square '((2 5) (3 1 6)))
((4 25) (9 1 36))
```

Use higher order functions, not recursion, and respect all relevant data abstractions!

QUESTION 7.

Sometimes when we see lots of parentheses around a single variable, we get confused as to what it's supposed to be doing:

```
((((f))) 1 3)
```

Write a procedure **make-nested** that takes a number **parens** and a procedure **end-with** and returns a procedure that, when called with **parens** number of nested parentheses, will invoke **end-with** after removing itself from the nested parentheses. For the example above, we would use:

```
(define f (make-nested 3 +))
```

which would cause ((((f))) 1 3) to return 4 because there are three nested parens directly around f, and then + is called on 1 and 3.

QUESTION 8.

Here is a class **set-of-numbers** that represents a mutable set of numbers (a set has no duplicates). We would like to implement three methods for this set of numbers:

```
add – adds a number into the set.
      Recall that a set should never have duplicates.
remove – removes a number from the set.
         Has no effect if the set doesn't contain the target number.
remove-all – takes another set-of-numbers as an argument and removes
             all the numbers that appear in the other set of numbers
             from the current set. The other set is unchanged.
```

The return values of these methods are unimportant.

```
(define-class set-of-numbers
    (instance-vars (nums '()))
    ;; finish the implementation here
```

QUESTION 9.
Write a predicate procedure **deep-car?** that takes a symbol and a deep-list (possibly including sublists to any depth) as its arguments. It should return true if and only if the symbol is the car of the list or of some list that's an element, or an element of an element, etc.

```
> (deep-car? 'a '(a b c))
#t
> (deep-car? 'a (b (c a) a d))
#f
> (deep-car? 'a ((x y) (z (a b) c) d))
#t
```

QUESTION 10. (Environment Diagrams)

```
1. (define (kons a b)
      (lambda (m)
          (if (eq? m ’kar) a b)))

   (define p (kons (kons 1 2) 3))
```

```
2. (define x 3)
   (define y 4)
   (define foo
      ((lambda (x) (lambda (y) (+ x y)))
       (+ x y)))
   (foo 10)
```

```
3. (define x 17)
   (define f
      (let ((x 4))
          (lambda (y)
             (print x)
             (set! x y)
             y)))
```

QUESTION 11.
(Question 2 of MT3, Spring 1996)
Write **list-rotate!** which takes two arguments, a nonnegative integer n and a list seq. It returns a mutated version of the argument list, in which the first n elements are moved to the end of the list, like this:

```
> (list-rotate! 3 (list ’a ’b ’c ’d ’e ’f ’g))
(d e f g a b c)
```

You may assume that 0 ≤ n < (length seq) without error checking.

Note: **Do not allocate any new pairs in your solution.** Rearrange the existing pairs.

QUESTION 12.
Consider **foo**, a popular name for mystery functions.

```
(define foo
      (let ((L (list 1)))
         (lambda (a) (let ((M (cons a L)))
                        (lambda (b) (set! L (cons (+ a 1) L))
                                    (set! a (+ a 2))
                                    (set-car! (cdr M) (+ 3 (cadr M)))
                                    (set! b (+ b 4))
                                    (list a b L M))))))
```

Fill in the blanks below :

```
> (define f (foo 1))
> (f 2)
____________________________
> (define g (foo 2))
> (g 1)
____________________________
> (f 2)
____________________________
> (g 1)
____________________________
```

QUESTION 13. (Metacircular Evaluator)
1. Recall that mceval.scm tests true or false using the **true?** and **false?** procedure:

```
  (define (true? x) (not (eq? x false)))
  (define (false? x) (eq? x false))
```

Suppose we type the following definition into MCE:

```
  MCE> (define true false)
```

What would be returned by the following expression:

```
  MCE> (if (= 2 2) 3 4)                                  _____________
```

2. Alyssa P. Hacker has noticed that many 61A students lose points by leaving out the empty frames that you get from invoking a procedure with no arguments. She asks all her friends how she can teach the students better so they won't make this mistake.

Ben Bitdiddle says, "Instead of teaching the students better, let's just modify the metacircular evaluator so that it doesn't create frames that would be empty. Then the students will be right." Modify the metacircular evaluator to implement Ben's suggestion.

(a) Is this primarily a change to **eval** or to **apply**?

(b) What specific procedure(s) will you change?

(c) Make the changes on the next page.

(d) Lem E. Tweakit notices that this change is not such a great idea. He thinks the modified metacircular evaluator will interpret certain Scheme programs incorrectly. Under what circumstances would this modification to the evaluator change the meaning of Scheme programs run using the metacircular evaluator? (That is, what kind of program will produce different results using the modified version than using the original one?)

```
(define (mc-eval exp env)
  (cond ((self-evaluating? exp) exp)
     ((variable? exp) (lookup-variable-value exp env))
     ((quoted? exp) (text-of-quotation exp))
     ((assignment? exp) (eval-assignment exp env))
     ((definition? exp) (eval-definition exp env))
     ((if? exp) (eval-if exp env))
     ((lambda? exp) (make-procedure (lambda-parameters exp) (lambda-body exp)
env))
     ((begin? exp) (eval-sequence (begin-actions exp) env))
     ((cond? exp) (mc-eval (cond->if exp) env))
     ((application? exp) (mc-apply (mc-eval (operator exp) env)
                            (list-of-values (operands exp) env)))
     (else
      (error "Unknown expression type -- EVAL" exp))))

(define (mc-apply procedure arguments)
  (cond ((primitive-procedure? procedure)
           (apply-primitive-procedure procedure arguments))
        ((compound-procedure? procedure)
           (eval-sequence (procedure-body procedure)
                          (extend-environment
                            (procedure-parameters procedure)
                            arguments
                            (procedure-environment procedure))))
        (else (error "Unknown procedure type -- APPLY" procedure))))

(define (list-of-values exps env)
  (if (no-operands? exps)
      '()
      (cons (mc-eval (first-operand exps) env)
            (list-of-values (rest-operands exps) env))))

(define (extend-environment vars vals base-env)
  (if (= (length vars) (length vals))
      (cons (make-frame vars vals) base-env)
      (if (< (length vars) (length vals))
            (error "Too many arguments supplied" vars vals)
            (error "Too few arguments supplied" vars vals))))

(define (make-frame variables values)
  (cons variables values))
```