CS61A SUMMER 2010
MIDTERM 2 REVIEW SESSION

George Wang, Jonathan Kotker, Seshadri Mahalingam, Steven Tang, Eric Tzeng

---

1. What will the Scheme interpreter print in response to the last expression in the following sequence of expressions? Also, draw a box and pointer diagram for the final result.

```
(define x '(a b c))
(define y '(1 2 3))
(set-cdr! (cddr x) (cddr y))
(set-car! (cdr y) (cdr x))
(set-car! x y)
x
```

---

2. Fill the blank with a mutation instruction so that the final result is as shown. Be sure to draw a box and pointer diagram. Use list mutation only. **Do not allocate any new pairs!**

```
(define z '(1 2 3 4))
(set-car! (cdr z) (cdddr z))
```

_____

```
(set-car! z (cdr z))

z
(((4) 3) (4) 3)
```

3. Consider a class representing characters from Nintendo's popular video game *Super Smash Brothers*. Here's an example of how it works:

```
; Instantiate Zelda.
STk> (define zelda (instantiate smash-character 2))
zelda

; Instantiate Jigglypuff.
STk> (define jigglypuff (instantiate smash-character 1))
jigglypuff

; Zelda starts out with 0 damage.
STk> (ask zelda 'damage)
0

; Jigglypuff attacks Zelda for 14 damage.
STk> (ask jigglypuff 'attack zelda 14)
okay

; Zelda attacks Jigglypuff for 20.
STk> (ask zelda 'attack jigglypuff 20)
okay

; Jigglypuff attacks again, killing Zelda.
STk> (ask jigglypuff 'attack zelda 18)
(i died >_<)

; Zelda's damage is reset to 0.
STk> (ask zelda 'damage)
0

; Zelda attacks, kills Jigglypuff, and wins.
STk> (ask zelda 'attack jigglypuff 15)
(i lost x_x)
```

In this simple version, assume that each character is given the number of lives at instantiation. Also, each character keeps track of its own damage, initially at 0. When one character attacks another, the victim's damage count increases by the given damage amount. For simplicity's sake, let's assume that a character dies when its damage is 30 or greater, at which a life is lost and the damage count is reset to 0. Once a character runs out of lives, it loses. Write the smash-character class.

4. (Midterm 3 Spring 2009) The following is an implementation of an object class in plain Scheme:

```
(define make-foo
   (let ((a 3))
     (lambda (b)
       (let ((c 4))
          (define (d e)
            (+ a e))
          (define (f g)
            (if (eq? g 'h)
                d
                (error "huh?")))
          f))))
```

Indicate which symbol in the code above corresponds to each OOP concept. Note: One of the questions has two answers (two symbols that match the concept), and one of the symbols is the answer to two of the questions!

Class variable:	a	b	c	d	e	f	g	h
Instance variable:	a	b	c	d	e	f	g	h
Instantiation variable:	a	b	c	d	e	f	g	h
Message:	a	b	c	d	e	f	g	h
Method:	a	b	c	d	e	f	g	h
Dispatch procedure:	a	b	c	d	e	f	g	h
Method argument:	a	b	c	d	e	f	g	h
Instance:	a	b	c	d	e	f	g	h

5. What is the environment diagram that arises due to the following sequence of calls?

```
(define foo
  (let ((f (lambda (z) (+ z 2))))
    (let ((z (lambda () (f 4)))
          (y 3))
      (lambda (x)
        (set! y (lambda (z) (x z)))
        (y (z))))))

(foo 1+)
```

6. (Midterm 3, Spring 2005) Write a procedure **link-first!** that takes a non-empty list as its argument and destructively links together all elements in the list with the same value as the car of the list. For example,

```
> (define ls '(1 2 3 2 1 1 2 1))
> (link-first! ls)
> ls
(1 1 1 1)
```

you can assume list elements are numbers as in this example (so don't worry about sublists), but don't use word/sentence procedures. **Your procedure must not create new pairs!**

7. We are going to implement a new special form called **cases** in the metacircular evaluator. It has the following general form:

   (cases <variable> (<list of choices> <action>)
                     (<list of choices> <action>) …)

   An example of its usage:

```
(cases x
       ((2 3 4) (+ x 1))
       ((7 8 9) (+ x 2))
       (else (+ x 3)))
```

   The idea is that, depending on the list storing the current value of "x", the corresponding action will be taken. (The action can be a list of expressions.)

   Identify which portions of the metacircular evaluator need to be modified to account for this new special form, and implement the necessary procedures to make this special form work. You may find that the process will be easier if you create (and use!) appropriate selectors and constructors.

   For convenience, the metacircular evaluator has been reproduced below:

```
(define (input-loop)
  (display "=61A=> ")
  (flush)
  (let ((input (read)))
    (if (equal? input 'exit)
        (print "Au Revoir!")
        (begin
          (print (mc-eval input the-global-env))
          (input-loop)))))

(define (mc-eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        ((if-exp? exp)
         (if (not (eq? (mc-eval (cadr exp) env) 'nay))
             (mc-eval (caddr exp) env)
             (mc-eval (cadddr exp) env)))
        ((begin-exp? exp)
         (eval-sequence (cdr exp) env))
        ((quote-exp? exp) (cadr exp))
        ((set-exp? exp)
             (set-variable-value! (cadr exp)
                                  (mc-eval (caddr exp) env)
                                  env))
```

```
        ((definition? exp)
         (if (list? (cadr exp))
             (mc-eval (define->lambda exp) env)
             (define-variable!
                            (cadr exp)
                            (mc-eval (caddr exp) env) env)))

        ((lambda-exp? exp) (make-procedure (cadr exp) (cddr exp) env))
        ((list? exp) (mc-apply (mc-eval (car exp) env)
                               (map (lambda (arg-exp)
                                      (mc-eval arg-exp env)) (cdr exp))))
        (else (error "UNKNOWN expression"))))

(define (mc-apply fn args)
  (cond ((lambda-proc? fn)
          (eval-sequence (body fn)
                         (extend-environment
                             (params fn)
                             args
                             (env fn))))
        (else (do-magic fn args))))

;;;;;;;;;;;;;;;;;;;;;;
;; Procedure ADT ;;;;;
;;;;;;;;;;;;;;;;;;;;;;

(define (make-procedure params body env)
  (list 'procedure params body env))

(define (params p)
  (cadr p))

(define (body p)
  (caddr p))

(define (env p)
  (cadddr p))

(define (lambda-proc? p)
  (and (list? p)
       (eq? (car p) 'procedure)))

;;;;;;;;;;;;;;;;;;;;;;;
;; Helper Procedures ;;;
;;;;;;;;;;;;;;;;;;;;;;;
(define (quote-exp? exp)
  (eq? (car exp) 'quote))

(define (define->lambda exp)
  (list 'define (caadr exp)
        (append (list 'lambda (cdadr exp)) (cddr exp))))

(define (set-exp? exp)
  (eq? (car exp) 'set!))
```

```
(define (lambda-exp? exp)
  (eq? (car exp) 'lambda))


(define (begin-exp? exp)
  (eq? (car exp) 'begin))


(define (eval-sequence exps env)
  (cond ((null? (cdr exps)) (mc-eval (car exps) env))
        (else
          (mc-eval (car exps) env)
          (eval-sequence (cdr exps) env))))


(define (if-exp? exp)
  (and (list? exp)
       (eq? (car exp) 'if)))


(define (boolean? exp)
  (or (eq? exp 'aye)
      (eq? exp 'nay)))


(define (do-magic fn args)
  (apply fn args))


(define (definition? exp)
  (eq? (car exp) 'define))

(define (variable? exp)
  (symbol? exp))

(define (self-evaluating? exp)
  (or (number? exp)
      (boolean? exp)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Additional Primitives ;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (yell wd)
  (word wd '!!))

(define (square num)
  (* num num))

(define (factorial num)
  (if (= num 0) 1
      (* num (factorial (- num 1)))))

(define (new-null? ls)
  (if (null? ls)
      'aye
```

```
      'nay))

(define (new-= num1 num2)
  (if (= num1 num2)
      'aye
      'nay))

(define (new-< num1 num2)
  (if (< num1 num2)
      'aye
      'nay))

;;;;;;;;;;;;;;;;;;;;;;;;
;; Environment Related ;;;
;;;;;;;;;;;;;;;;;;;;;;;;

(define (extend-environment vars vals base-env)
  (cons (cons vars vals) base-env))

(define (define-variable! var val env)
  (define first-frame (car env))
  (define (scan vars vals)
    (cond ((null? vars)
           (set-car! first-frame (cons var (car first-frame)))
           (set-cdr! first-frame (cons val (cdr first-frame))))
          ((eq? var (car vars))
           (set-car! vals val))
          (else
           (scan (cdr vars) (cdr vals)))))
  (scan (car first-frame) (cdr first-frame))
  var)

(define the-global-frame
  (cons (list '+ '- '/ '* 'car 'cdr 'cons 'null? 'nil 'yell 'square 'factorial
              '=   '<    'list)
        (list + - / * car cdr cons new-null? nil yell square factorial
              new-= new-<  list)))


(define the-global-env
  (cons the-global-frame nil))


(define (set-variable-value! var val env)
  (define first-frame (car env))
  (define (scan vars vals)
    (cond ((null? vars)
               (if (eq? env the-global-env)
                   (error "Unbound Variable")
                   (set-variable-value! var val (cdr env))))
          ((eq? var (car vars)) (set-car! vals val))
          (else (scan (cdr vars) (cdr vals)))))
  (scan (car first-frame)
        (cdr first-frame)))
```

```
(define (lookup-variable-value var env)
  (define first-frame (car env))
  (define (scan vars vals)
    (cond ((null? vars)
               (if (eq? env the-global-env)
                   (error "Unbound Variable")
                   (lookup-variable-value var (cdr env))))
          ((eq? var (car vars)) (car vals))
          (else (scan (cdr vars) (cdr vals)))))
  (scan (car first-frame)
        (cdr first-frame)))

(input-loop)
```

8. Write a procedure vector-remove! that, given a vector and an element that may or may not be in the vector, returns a new vector with the occurrences of that element removed. The size of the new vector should match exactly the number of elements that are not the ones removed. You may find the count procedure helpful, which, given a vector and an element, returns the number of occurrences of that element in that vector. **Do not use vector->list or list->vector in this problem.** Use no data aggregates other than vectors.

```
(define (count v i num)
  (if (< num (vector-length v))
    (if (= i (vector-ref v num))
        (+ 1 (count v i (+ num 1)))
        (count v i (+ num 1)))
    0))

STk> (vector-remove! #(1 2 3 1) 1)
#(2 3)
STk> (vector-remove! #(1 2 3) 4)
#(1 2 3)
```