Structure and Interpretation of Computer Programs
October 2000

Quiz 4
Lisp, the Universe, and Everything

Before starting, please write your name in the first blank below, and *optionally* a guess as to how well you think you will do in the second. Start the quiz only when instructed to do so. You may use any written resources you wish, but you may not consult another student, nor use a computer, nor a calculator. You will have three hours to finish this quiz, at which point please close the document, and *optionally* re-assess your anticipated grade in the third blank below. Please give your tests to the staff as you leave. Your actual grade will not be affected by your self-assessment, nor by opting out of self-assessment.

Name: *Solutions*

Optional, expected grade (percent correct) before taking quiz: _____

Optional, expected grade (percent correct) after taking quiz: _____

Problem	*Score*	*Value*
1		10
2		10
3		10
4		10
5		15
6		10
7		20
8		15
Total		100

Problem 1: 10 points Examine the functions `enigma`, `mystery`, and `conundrum` given below.

```
(define (enigma m)
  (m 2))

(define (mystery p q)
  (lambda (z) (p (enigma q) z)))

(define (conundrum a)
  (lambda (x) (mystery x a)))
```

What will be the result of evaluating the following Scheme expressions? Write "procedure" if a procedure object would be returned, and write "error" if an error would be generated. Otherwise, write the value resulting from evaluating the expression. Assume that our familiar definition

```
(define (square x) (* x x))
```

has been already evaluated.

```
(enigma inc)
;Value: 3
```

```
(enigma 'square)
;The object square is not applicable.
;Type D to debug error, Q to quit back to REP loop: q
;Quit!
```

```
(mystery list square)
;Value: #[compound-procedure 6]
```

```
((mystery list square) 3)
;Value: (4 3)
```

```
(((conundrum square) list) 2)
;Value: (4 2)
```

Problem 2: 10 points **(A)** Assume that we have defined a variable `x` which contains a list structure. Later we use `set!` to assign a completely new value to `x`. Under what circumstances is it safe to garbage collect the list structure from old value of `x`.

If nothing else points to the old value of `x`*, or any substructure of* `x`*, it will be safe to reclaim the space.*

(B) Given the following diagram of memory locations where `root` is `P2`,

0	1	2	3	4	5	6	7
N5	P4	P3	N4	P7	N3	P2	P0
E0	N2	P7	P0	E0	E0	P4	P4

how many free cells will there be after garbage collection?

3

(C) What is one advantage of the stop-and-copy garbage collection method over mark-and-sweep?

Increases locality.
Only looks at good cells.

(D) What is one advantage of the mark-and-sweep garbage collection method over stop-and-copy?

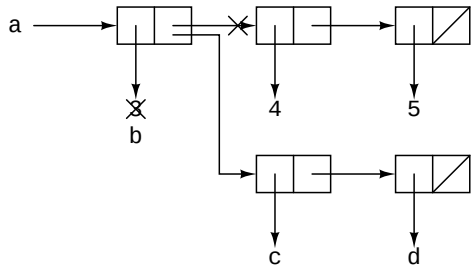
More use of available memory resources.

Problem 3: 10 points **(A)** Assume that we have evaluated the following four expressions.

```
(define a '(1 2 3))
(set! a '(3 4 5))
(set-car! a 'b)
(set-cdr! a '(c d))
```

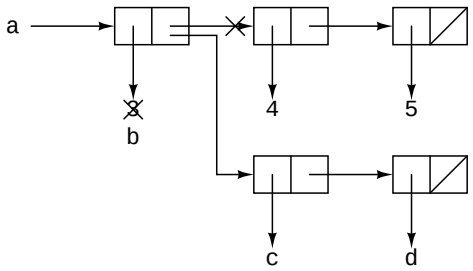
What will be the result of evaluating `a` ?

(b c d)



Note: The initial list structure containing 1, 2, and 3 is not shown on this diagram.

(B) Assume a structure corresponding to the box and pointer diagram below has been bound to the variable `s`. Write a 1-line expression which when evaluated will replace the symbol `b3` with the symbol `x`.



1-line expression: *(set-car! (caadr s) 'x)*

Problem 4: 10 points Ben Bitdiddle, ever curious, finds an obscure implementation of Scheme where `delay` is *not* memoized (sometimes called "Really Repulsively Reluctant and Slow Scheme" or R^{3}S Scheme). He executes the following snippets of code as a series of experiments. His function `(display-stream s n)` prints out the first `n` elements of the stream `s`.

```
(define *ones-count* 0)
(define ones (cons-stream 1 (begin
                              (set! *ones-count* (inc *ones-count*))
                              ones)))
(display-stream ones 10)
```

(A) What will the value of `*ones-count*` be in R^{3}S Scheme?

9

The function `display-stream` *will* `stream-cdr` *down* `ones`*, printing the first 10 values. The first value will be* `(stream-car ones)` *which is not delayed and returns* `1` *without modifying* `*ones-count*`*. The second value will be* `(stream-car (stream-cdr ones))`*, the third will be* `(stream-car (stream-cdr (stream-cdr ones)))`*, and so forth. Each time* `stream-cdr` *is applied to* `ones`*, the mutation statement will be evaluated and the value of* `*ones-count*` *incremented (the* `cdr` *of* `ones` *is a promise—a lambda expression of no arguments—which, when* `force`*d will evaluate the body of the* `begin` *statement). Thus, first the* `set!` *expression will be evaluated, mutating the value of* `*ones-count*`*, and next the value of* `ones` *will be returned. Therefore, printing the first term will not modify the value of *ones-count*, but printing every other term will, and the result will be nine.*

(B) Then, he evaluates the same code in MIT Scheme. What value does he get for `*ones-count*`?

1

In a sense, the MIT Scheme analysis is much simpler. Every time the `cdr` *of* `ones` *is evaluated, a check is made to see if the zero-argument lambda function which comprises the promise has been previously executed, returning the memoized result when it has. This will be every time but the first, and therefore, the incrementing operation on* `*ones-count*` *will only happen once. All additional times that it would be executed the memoized result of the promise will be returned instead, bypassing the execution of the* `set!` *statement.*

Problem 5: 15 points **(A)** Write a procedure called `map-two-streams` which takes a procedure of two arguments and two streams. It will return the stream that results from applying the procedure to corresponding elements in the two streams. For example, `(map-two-streams + ones integers)` would return the stream `2 3 4 5 6 7 8 ...`.

```
(define (map-two-streams proc s1 s2)
  (cons-stream (proc (stream-car s1)
                     (stream-car s2))
               (map-two-streams (stream-cdr s1)
                                (stream-cdr s2))))
```

(B) Use `map-two-streams` to write a procedure called `min-two-streams`, which takes two streams and returns a stream consisting of the smaller of the two elements at each stream position. You may use the `min` primitive from MIT Scheme.

```
(define (min-two-streams s1 s2)
  (map-two-streams min s1 s2))
```

Problem 6: 10 points Write a Scheme procedure of one argument `n` which returns the same value as the value computed in `s` by the register machine below. Call your procedure `bob`. **Do not use side effects in your answer.**

```
(controller
  START
    ;; assume N initialized
    (assign s (const 0))
  LABEL1
    (test (op =) (reg n) (const 0))
    (branch (label done))
    (assign s (op +) (reg s) (reg n))
    (assign n (op -) (reg n) (const 1))
    (goto label1)
  DONE
  )
```

```
(define (bob n)
  (define (bob-iter n s)
    (if (= n 0)
        s
        (bob-iter (- n 1) (+ s n))))
  (bob-iter n 0))
```

This procedure does not check for `n` *being negative, but then again, neither does the register machine above!*

Problem 7: 20 points **(A)** If you were writing an interpreter for a new computer language, would it be better to model it on the metacircular evaluator or the analyze evaluator? Why? Write at most 3 sentences.

Metacircular	Analyze
easier to implement *easier to develop and debug*	*better performance*

(B) Why is compiled code more efficient than interpreted code? Again, write at most 3 sentences.

Syntactic analysis is only done once.
Compilers can optimize generated code.

Problem 7(continued) **(C)** We want to add a new `adu:when` special form to our metacircular evaluator. The new special form has the following behavior: `(adu:when` ⟨*pred*⟩ ⟨e_1⟩ ⟨e_2⟩ `...` ⟨e_n⟩`)` tests the predicate ⟨*pred*⟩ and executes the expressions ⟨e_1⟩ through ⟨e_n⟩ if and only if the predicate returned a true value. The value returned by the entire expression is the value returned by ⟨e_n⟩ if it was evaluated, and `adu:false` otherwise. Add `adu:when` to `mc-eval`, which is given on the last page of this exam as a convenient tear-off sheet. Tell us where you would insert your code using the line numbers on the tear-off sheet, *e.g.* by writing "after line X." Assume that the appropriate predicate and selectors for `when` expressions exist: `when?`, `when-predicate`, and `when-exps`.

```
;; after line 21
((when? exp) (eval-when exp env))


;;; ... and then somewhere else ...
(define (eval-when exp env)
 (if (eq-adu-true? (mc-eval (when-predicate exp) env))
     (eval-sequence (when-exps exp) env)
     (eval 'adu:false env)))
```

(D) Now we want to add `adu:not` as a primitive to our system. **Briefly** discuss where in the code you would do this, and what approach you would take.

Write the primitive procedure in Scheme and install in list of primitives, bound to `adu:not`.

Problem 8: 15 points Let us add a `videotape` to our game system from Problem Set 7 (you may ask a staff member for a copy of the code if you do not have one). A `videotape` will be a type of `thing`. Its only method will be `self-destruct`, where it will move to the location `nowhere` and have the owner `nobody`. We could write the following code:

```
(define (make-videotape name location)
  (let ((⟨e1⟩ ⟨e2⟩))
    (lambda (message)
      (cond ((eq? message 'self-destruct)
             ⟨e3⟩)
            (else
             ⟨e4⟩))))))
```

Write the missing sections of code.

The intended answer, which fits nicely into the space provided, does not actually work. Had the instructors been taking the exam, they would not have done very well! The expected answer was as follows:

```
(define (make-videotape name location)
  (let ((videotape (make-thing name location)))
    (lambda (message)
       (cond ((eq? message 'self-destruct)
              (lambda (self)
                 (ask self 'set-place 'nowhere)
                 (ask self 'set-owner 'nobody))
             (else
               (get-method thing message))))))
```

This solution will not actually work because of the following bugs. First, if the videotape is owned, the owner is not informed of having lost possession of it. Similarly, the place where the videotape resides is not informed of the tape having moved, and so the videotape will appear to be in two places at once.

Attempts to correct the first bug by using the `move-to` *method would have not worked because* `things` *do not have the* `move-to` *method. The* `change-place` *function (note: a procedure, not a method) was written for just this special case in the game code.*

The ownership bug is slightly more complicated because we need to see if there is an owner before asking the owner to lose the object. Once that has happened, we can then tell the object to have no owner.

Note that attempts to create ownership by introducing a local `owner` *variable were not necessary, as* `things` *already can be owned, and shadowing that variable is unnecessary. Also, performing mutations locally via* `set!` *expressions suffers from the same inheritance problems that we studied in lecture and section: the right way to avoid this is to use the* `self` *argument from the method procedure.*

The full, correct, solution is on the next page.

Problem 8 (continued) *A very small number of students got the correct solution, or came close. Grading was intentionally lenient.*

```
(define (make-videotape name location)
  (let ((videotape (make-thing name location)))
    (lambda (message)
       (cond ((eq? message 'self-destruct)
              (lambda (self)
                 (if (ask self 'owned)
                     (ask (ask self 'owner) 'lose self))
                 (ask self 'set-owner 'nobody)
                 (change-place self nowhere)))
             (else
               (get-method thing message))))))
```

Convenient Tear-Off Sheet This page contains a copy of the definition of `mc-eval` for use with Problem 7. This is the same definition as appeared in Problem Set 9 (you may ask a staff member for a copy of the remainder of the code from the problem set if you would like).

```
(define (mc-eval exp env)                                                  ;  (1)
  (cond ((self-evaluating? exp)                                            ;  (2)
         exp)                                                              ;  (3)
        ((variable? exp)                                                   ;  (4)
         (lookup-variable-value exp env))                                  ;  (5)
        ((quoted? exp)                                                     ;  (6)
         (text-of-quotation exp))                                          ;  (7)
        ((assignment? exp)                                                 ;  (8)
         (eval-assignment exp env))                                        ;  (9)
        ((definition? exp)                                                 ; (10)
         (eval-definition exp env))                                        ; (11)
        ((if? exp)                                                         ; (12)
         (eval-if exp env))                                                ; (13)
        ((and? exp)                                                        ; (14)
         (eval-and exp env))                                               ; (15)
        ((lambda? exp)                                                     ; (16)
         (make-procedure (lambda-parameters exp) (lambda-body exp) env))   ; (17)
        ((begin? exp)                                                      ; (18)
         (eval-sequence (begin-actions exp) env))                          ; (19)
        ((cond? exp)                                                       ; (20)
         (mc-eval (cond->if exp) env))                                     ; (21)
        ((application? exp)                                                ; (22)
         (mc-apply (mc-eval (operator exp) env)                            ; (23)
                   (list-of-values (operands exp) env)))                   ; (24)
        (else (error "Unknown expression type -- MC-EVAL"))))              ; (25)
```