MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001-- Structure and Interpretation of Computer Programs
Fall Semester, 1998, Quiz II

Be sure to write your name all all pages of this quiz.

Print Your Name: Ben Bitdiddle

Your Recitation Instructor: John Von Neumann

Your Tutor: John McCarthy

Please write clear and concise answers to the questions in the spaces provided in this booklet. You may use scratch paper if you need, but the spaces we provide are the only places we will look at when grading. Note that your solutions, particularly to programming problems, may be judged not only on whether they work or not, but also on clarity and ease of understanding.

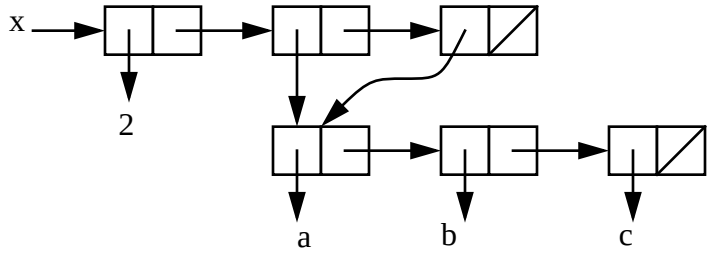
Any comments you would like to make on this quiz:

Comments from graders:

Problem	Value	Grade	Grader
1	25		
2	24		
3	15		
4	15		
5	21		
Total	100		

Problem 1 (25 points)

Consider the following list structure:



Part a

What is the printed representation in Scheme for the value of the variable `x`?

`ANS: (2 (a b c) (a b c))`

Part b

The list structure for `x` shown above can be created with the following Scheme code:

```
(define x
  (let ((y <exp1>))
    (<exp2>)))
```

Complete the code above by providing `<exp1>:`

and `<exp2>`:

Overall code:

```
(define x
  (let ((y (list 'a 'b 'c)))
    (list 2 y y)))
```

Part c

The same list structure for `x` can alternatively be created with the following Scheme code:

```
(define x (list 2 '(a b c) 3))
(set-car! <exp3> <exp4>)
```

Complete the code above by providing `<exp3>`:

```
(cddr x)
```

and `<exp4>`:

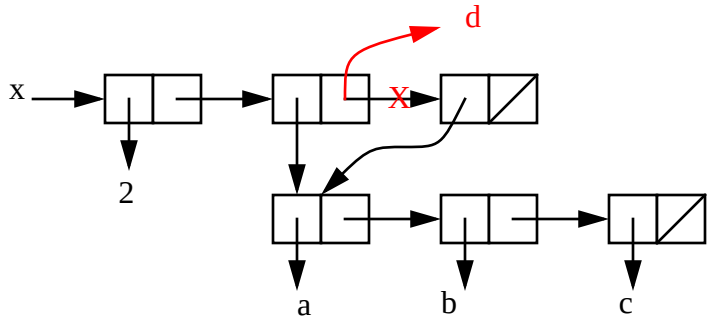
```
(caddr x)
```

Part d

Show how the list structure for `x` in the figure above changes when the following expression is evaluated:

```
(set-cdr! (cdr x) 'd)
```

You should change the drawing in the figure provided at the beginning of this problem. If a pointer changes, draw an "X" through that pointer and draw in any new pointers or values.:

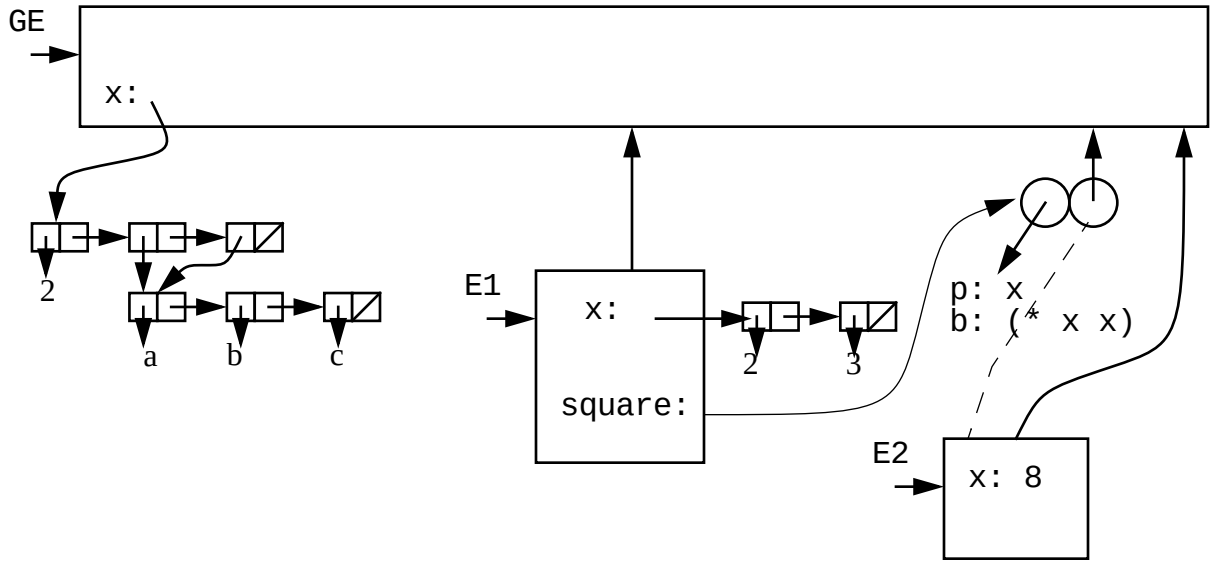


Part e

You are given a global environment as shown (with the same x data structure as in Part a). Draw the environment diagram that results from the evaluation of the following expression (evaluated in the global environment):

```
(let ((x (list 3 (car x)))
      (square (lambda (x) (* x x))))
  (square (+ 5 (car x))))
```

You can assume that `list`, `car`, `+` and `*` are all primitive procedures that already exist in the global environment.



The trickiest part of this problem is to recognize that the (lambda (x) (* x x)) is evaluated in the Global Environment outside the let expression, and so the environment pointer of the square procedure also points to the GE (as do the frame resulting from the application of that procedure to the actual argument 8).

Problem 2 (24 points)

The vote counting for yesterday's election is stalled because a critical part of the following program for tallying the votes has not yet been completed. Your job is to help finish this system.

To begin, the following "tally" data structure has been specified to hold the current count of votes:

```
(tally
  (<name1> <#votes>)
  (<name2> <#votes>)  ;; bug fix - just one paren
  ...
  (<name3> <#votes>))  ;; bug fix -- just two parens
```

The system must support procedures to get the number of votes for any particular candidate, to add a vote to the tally, and to determine the winner of the election. These are illustrated by the following sequence of expressions (you will note that the voter turnout was exceedingly low!):

```
(define governor (list 'tally))    ; initial empty tally

(add-vote 'scott governor)
(add-vote 'paul governor)
(add-vote 'scott governor)
(get-votes 'paul governor) ==> 1
(get-votes 'scott governor) ==> 2
(get-votes 'fred governor) ==> 0
(winner governor) ==> scott
governor ==> (tally (paul 1) (scott 2))
```

Part a

First, write the procedure `get-votes` that takes two arguments as above -- the name of a `candidate`, and a `tally` -- and returns the number of votes for that candidate. If the candidate's name does not appear in the tally, that indicates that they did not receive any votes.

```
(define (assq key alist)
  (cond ((null? alist) '())
        ((eq? key (caar alist)) (car alist))
        ((else (assq key (cdr alist))))))

(define (get-votes candidate tally)
  (let ((item (assq candidate (cdr tally))))
    (if (null? item)
        0
        (cadr item))))
```

Part b

Next, write the procedure `add-vote` that also takes two arguments as above -- the name of a `candidate`, and a `tally` -- and changes the `tally` to include the new or additional vote for that candidate.

```
(define (add-vote candidate tally)
  (let ((item (assq candidate (cdr tally))))
    (if (null? item)
        (set-cdr! tally (cons (list candidate 1)
                              (cdr tally)))
        (set-car! (cdr item) (+ 1 (cadr item))))
    'vote-added))
```

Part c

Finally, write the procedure `winner` which takes one argument -- the `tally` -- and returns the name of the winner (the candidate with the most votes). If the election is a tie, a list of all candidates with the largest number of votes should be returned.

```
(define (winner tally)
  (define (helper alist lead-candidate lead-votes)
    (cond ((null? alist) lead-candidate)
          (else
            (let ((candidate (caar alist))
                  (votes (cadar alist)))
              (cond ((> votes lead-votes)
                     (helper (cdr alist) candidate votes))
                    ((= votes lead-votes)
                     (helper (cdr alist)
                             (if (pair? lead-candidate)
                                 (cons candidate lead-candidate)
                                 (list candidate lead-candidate))
                             votes))
                    (else (helper (cdr alist)
                                  lead-candidate
                                  lead-votes)))))))
  (helper (cdr tally) '() 0))
```

Problem 3 (15 points)

Part a

Assume that you have the following definition for the stream of fibonacci numbers:

```
(define fibs
  (cons-stream 0
               (cons-stream 1
                            (add-streams fibs
                                         (stream-cdr fibs)))))
```

You come across the following definitions for streams `s1` and `s2`:

```
(define s1
        (stream-map square
                    (stream-filter odd? fibs)))

(define s2
        (stream-filter odd?
                       (stream-map square fibs)))
```

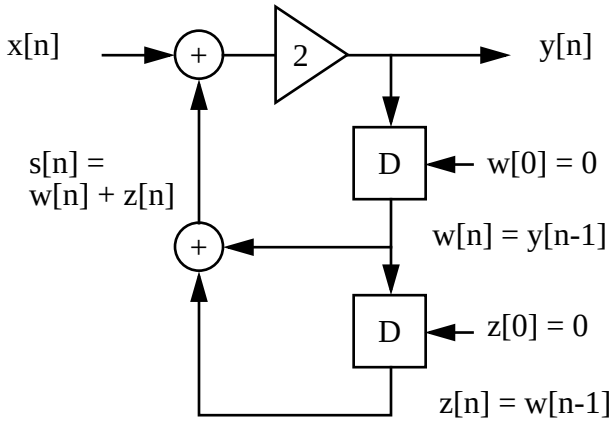
You wonder if `s1` and `s2` generate the same stream. To figure this out, you decide to compute what the first 5 elements of each are. Fill in the following table:

fibs	0	1	1	2	3	5	8
s1	1	1	9	25	169		
s2	1	1	9	25	169		

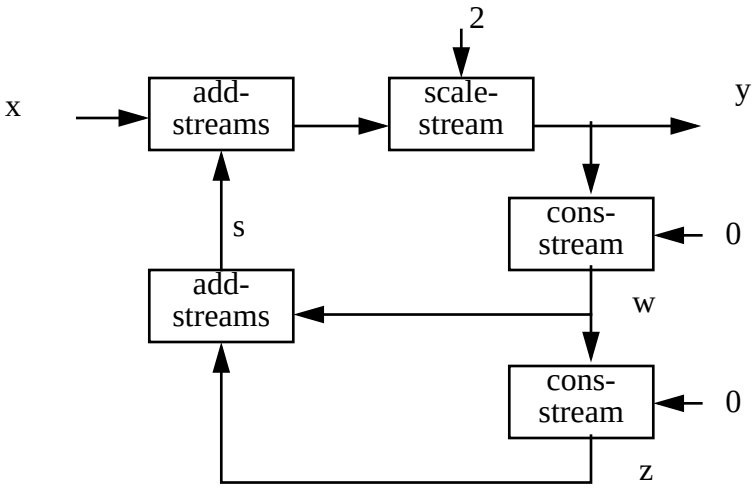
Standard definitions for the various stream procedures (e.g. `stream-filter`, `stream-map`) are provided at the end of the quiz handout in case you need them.

Part b

Streams are often applied to model signal processing systems. For example, the following diagram illustrates a simple digital signal processing (DSP) system that works on an input stream `x` (often denoted `x[n]` where `n` is the discrete time index) and produces an output stream `y`.



This same system can be represented in a more familiar Scheme streams processing framework as shown in the following figure:



The stream figure corresponds to the following scheme expressions for this DSP system:

```
(define w (cons-stream 0 y))
(define z (cons-stream 0 w))
(define s (add-streams w z))
(define y (scale-stream 2 (add-streams x s)))
```

These definitions are recursive (`y` depends on `s` which depends on `w` which depends on `y`!), and it is the delayed evaluation capability of streams that allows us to create such recursive data structures as these which often occur in digital signal processing. Standard definitions for the various stream procedures (e.g. `scale-stream`, `add-streams`) are provided at the end of the quiz handout in case you need them.

Complete the following table for the first few elements of the contents of these streams:

x	1	0	0	0
w	0	2	4	12
z	0	0	2	4
s	0	2	6	16
y	2	4	12	32

Problem 4 (15 points)

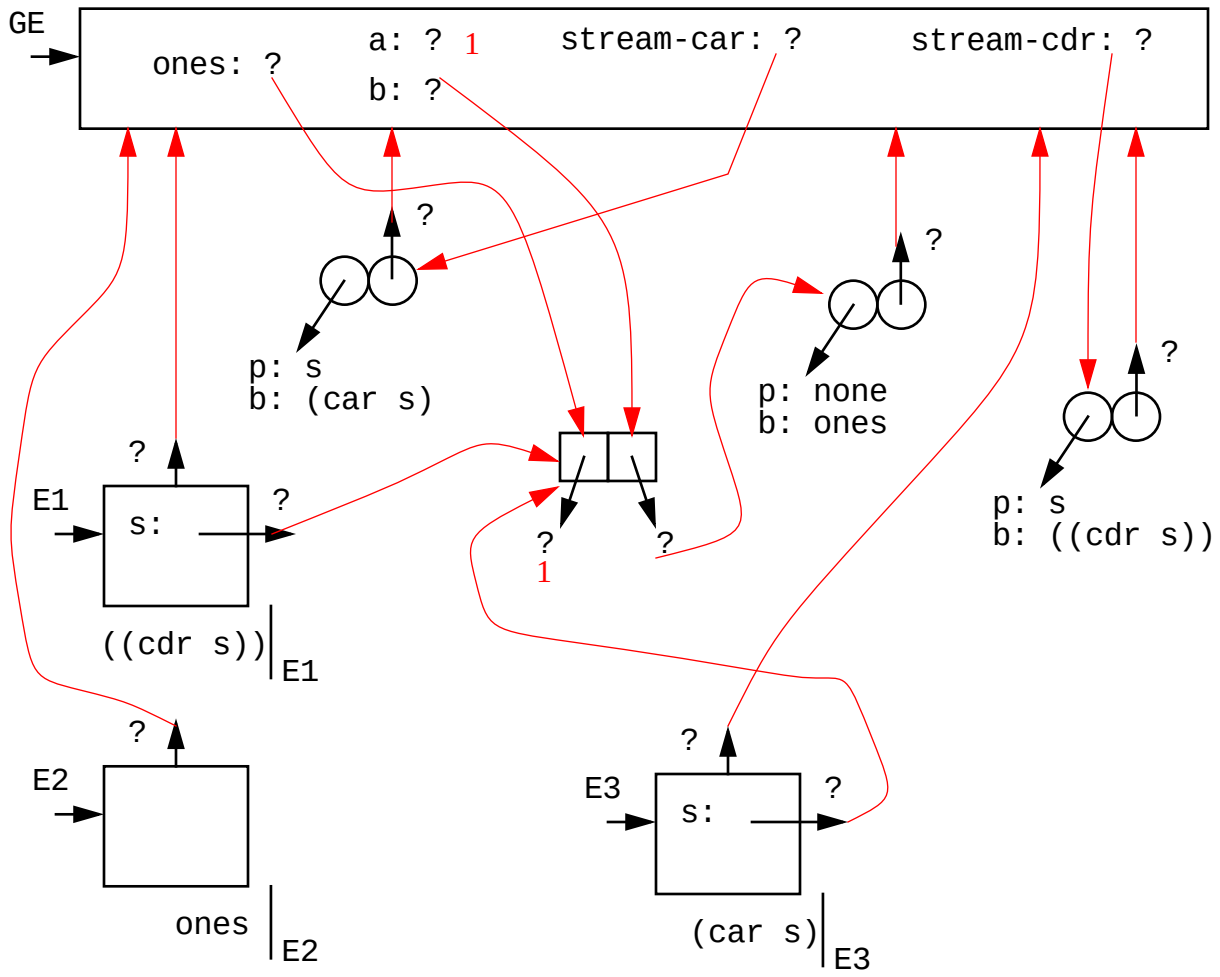
Here we look at one implementation of streams in Scheme, and develop an understanding of this implementation using the environment model. We will implement `(cons-stream x y)` as a special form that is equivalent to `(cons x (lambda () y))` as shown below:

```
(cons-stream x y)   ≡   (cons x (lambda () y))
(define (stream-car s) (car s))
(define (stream-cdr s) ((cdr s)))
```

The following partially completed environment diagram results from the evaluation of the above definitions, followed by the evaluation of the following expressions (in the global environment).

```
(define ones (cons-stream 1 ones))
(define a (stream-car ones))
(define b (stream-cdr ones))
```

Complete the diagram for all parts marked with a question mark, including variable and binding values, environment parts of procedures, enclosing environments for frames, and other data structures. You might want to work through the evaluation of the five define expressions above one at a time to ensure that you complete all 15 question marks in the environment diagram below.



Problem 5 (21 points)

In this problem, we develop a system to perform some generic operations on geometric objects such as circles, squares, and rectangles. We want a system that will enable us to create objects with different dimensions, and then be able to determine such things as the area of the object, or its perimeter. For example, we would like to

```
(define c1 (make-circle 5))         ; radius 5
(define s1 (make-square 3))         ; side of size 3
(define r1 (make-rectangle 2 3))    ; sides of size 2 and 3
(define s2 (make-square 10))

(area s1) ==> 9
(area s2) ==> 100
(area r1) ==> 6
(area c1) ==> 78.5

(perimeter c1) ==> 31.4
(perimeter r1) ==> 10
```

To support this, we will assume that the table `put` and `get` operations are available:

```
(put <op> <type> <procedure>)
(get <op> <type>) ==> <procedure>
```

The following (simplified) `apply-generic` interface is used:

```
(define (apply-generic op object)
  (let ((proc (get op (type-of object))))
    (if proc
        (apply proc (contents object))
        (error "No operation for op"))))
```

With the following type tagging approach:

```
(define (attach-tag object tag)
  (cons tag object))

(define (type-of object)
  (if (pair? object)
      (car object)
      (error "not a tagged object")))

(define (contents object)
  (if (pair? object)
      (cdr object)
      (error "not a tagged object")))
```

Consider the following partially complete implementation of the `rectangle` package.

```
(define (install-rectangle-package)
  ; Internal representation
  (define (make-r width height) ...)
  (define (width-r r) ...)
  (define (height-r r) ...)
  (define (area-r r) ...)
  (define (perimeter-r r) ...)

  ; External interface
  (define (tag r) (attach-tag r 'rectangle))
  (put 'make 'rectangle ...)
  (put 'area 'rectangle ...)
  (put 'perimeter 'rectangle ...)
  'done)

(define (make-rectangle w h)
  (apply (get 'make 'rectangle) w h))
```

Part a

Define the `make-r` constructor procedure, as well as the accessor procedures `width-r` and `height-r`. You have the freedom to choose your internal representation of the rectangle object, but be sure to make your constructors and accessors consistent!

```
(define (make-r width height)
  (cons width height))

(define (width-r r) (car r))

(define (height-r r) (cdr r))
```

The following generic `area` and `perimeter` operations are defined to work with not only circles and squares, but also rectangles:

```
(define (area geometry)
  (apply-generic 'area geometry))

(define (perimeter geometry)
  (apply-generic 'perimeter geometry))
```

Part b

Define the `area-r` procedure (inside the rectangle package) consistent with this generic procedure interface.

```
(define (area-r r)
  (* (height-r r) (width-r r)))
```

Part c

Finish the interfacing of the rectangle package to the generic geometry system, by writing the two `put` expressions below (which are called inside `install-rectangle-package`).

```
(put 'make 'rectangle ???)

ANS: (put 'make 'rectangle (lambda (w h) (tag (make-r w h))))

(put 'area 'rectangle ???)

ANS: (put 'area 'rectangle area-r)
```

Part d

We now want to extend our system to enable mutation of an existing object as in the following example code:

```
(define r2 (make-rectangle 3 4))
(area r2) ==> 12
(resize r2 5 6) ==> RESIZED
(area r2) ==> 30
```

We add the following `resize` generic operation to the system. Note that `resize` operates by side-effect to change the given geometric object. It does not return any new object; rather it just returns the symbol `RESIZED` to indicate success.

```
(define (resize geometry . args)
  (apply (get 'resize (type-of geometry))
         (contents geometry)
         args))
```

Write an internal procedure `resize-r` (which will be added inside the rectangle package) that will work consistently with this generic interface for `resize`:

```
(define (resize-r r new-w new-h)
  (set-car! r new-w)
  (set-cdr! r new-h)
  'resize)
```

Write the appropriate `put` expression to interface the `resize-r` procedure to the generic geometry package (as before, this `put` expression will be called inside `install-rectangle-package`).

```
(put 'resize 'rectangle ???)
```

ANS: `(put 'resize 'rectangle resize-r)`

Stream definitions (in case you need them):

```
(define (stream-map proc stream)
  (if (stream-null? stream)
      the-empty-stream
      (cons-stream (proc (stream-car stream))
                   (stream-map proc (stream-cdr stream)))))

(define (scale-stream c stream)
  (stream-map (lambda (x) (* x c)) stream))

(define (stream-filter pred s)
  (cond ((stream-null? s) s)
        ((pred (stream-car s))
         (cons-stream (stream-car s)
                      (stream-filter pred (stream-cdr s))))
        (else (stream-filter pred (stream-cdr s)))))


(define (add-streams s1 s2)
  (cond ((stream-null? s1) s2)
        ((stream-null? s2) s1)
        (else (cons-stream (+ (stream-car s1) (stream-car s2))
                           (add-streams (stream-cdr s1)
                                        (stream-cdr s2))))))
```