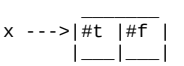
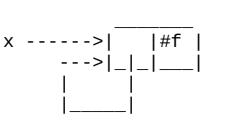
CS61A Midterm 3, Spring 2011

1. Box-and-Pointer Diagrams (2 points)

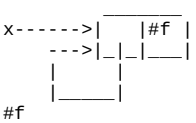
```
> (define x (cons #t #f))
```



```
> (if (car x)
      (set-car! x x)
      #f)
okay
```



```
;; uses the same X as above
> (if (cdr x)
      (set-cdr! x x)
      #f)
```



```
#f
```

Grading:
+1 both What Will Scheme Prints are correct
+1 final box and pointer diagram is correct

You did not receive a point for the box and pointer diagram if you made your left arrow point at X. (Arrows in Scheme can only point to values, not variables!) If you made the assumption that the third expression was evaluated with an unmodified x, you received points if you had a correct box and pointer diagram for the second expression. However, if you made that assumption but you were clearly modifying your box and pointer diagram from the second expression and you modified it incorrectly (e.g. set the cdr to be the pair), then you did not receive the point for the box and pointer diagram.

A common mistake was saying that Scheme would infinite loop after the second expression. However, SET-CAR! does not return the pair it is setting; it returns OKAY.

2. Client-Server (3 points)

First, the correct answer:

Alice	---------->	Server	----------> Bob
Source:		Alice	Alice
Destination:		Bob	Bob
Action:		send-msg	receive-msg
Data:		hello	hello

The most common wrong answer had the "source" and "destination" fields showing which computer is sending and receiving each request (i.e. "Alice/server" and "server/Bob"). This can't actually be how the IM client works, though: how would Bob know who a message came from if the source is just "server"?

However, some might argue that the low-level packets that make up a message would have a source of "Alice" and a destination of "server". This can't be the case; if it were, the action would have to be something more low-level than "send-msg", and the data would have to specify the final destination "Bob". Since the data is just "hello", you know we're talking about an IM request, not a low-level IP packet. Even so, it is a meaningful answer, so no points were taken off for this mistake.

The next mistake was forgetting that the request from a client to the server has a different action than a request from the server to a client, because they're doing different things. You didn't have to get the exact names of the messages correct, just the idea that the first request asks the server to "send" and the second asks Bob to "receive".

Finally, a fair number of answers had the first source as WHOAMI. Remember, the question asked for the *values* of each field in the requests, so WHOAMI will have Alice's user name, which is the value of the variable WHOAMI on Alice's computer.

Scoring:
+1 Source and destination fields go from Alice to Bob.
   (so Alice -> Server / Server -> Bob is okay)
+1 First action is SEND (or some close variant/synonym).
+1 Second action is RECEIVE (or some close variant/synonym).

3. Environment Diagram (6 points)

```
Number of lambda bubbles: 3
    - L1:
params:
body: (lambda (value)
                (lambda (new)
                  (set! value (list new))))
        Defining Environment: Global
    - L2:
params: value
body: (lambda (new)
                (set! value (list new)))
        Defining Environment: E1
    - L3:
params: new
body: (set! value (list new))
        Defining Environment: E2

Number of frames: 4
    - Global Environment
mystery : L1
enigma  : L3
    - E1 [extends Global]
    - E2 [extends E1]
value : (hello)
    - E3 [extends E2]
new : hello

This question gave a lot of students trouble - I suspect it may be due
to the line:
> (define enigma ((mystery) '()))
But let's step through this line-by-line.

> (define (mystery)
    (lambda (value)
      (lambda (new)
        (set! value (list new)) )))

Current Environment: Global
    Here, we are defining a procedure MYSTERY that takes no arguments.
    So here, we create a new lambda bubble L1 for mystery.
    It might help if you rewrite the definition out as:
        (define mystery
          (lambda ()
            (lambda (value) ...)))
since this is what's really happening.

> (define enigma ((mystery) '()))

Current Environment: Global
    First, we invoke MYSTERY. MYSTERY points to the lambda L1, so we
create a new environment E1 that extends Global.

Current Environment: E1 [extends Global]
    The body of L1 says to return (lambda (value) ...). So here, we
create a new lambda L2 for the return value of (mystery):

Current Environment: Global
    We just found out that (mystery) returned L2. Now, we have to evaluate
(L2 '()), since (mystery) returned L2.
    L2 is a lambda that 'points' to E1, so we create a new environment E2 that
extends E1.
Current Environment: E2 [extends E1]
    We bind the variable VALUE to the value () (since we passed in () as
an argument). Now, we evaluate the body of L2:
        (lambda (new)
          (set! value (list new)))
    So, we create a new lambda bubble L3.
Current Environment: Global
    Lambda L3 is the return value of the expression ((mystery) '()), so now we
can bind ENIGMA to L3 in the Global environment.

Finally, we can evaluate the last line:

> (enigma 'hello)

    This is invoking lambda L3 with argument HELLO - since L3's defining
environment is E2, we create a new environment E3 that extends E2.
Current Environment: E3 [extends E2]
    We bind the variable NEW to the value HELLO (since we passed in HELLO as
an argument). Now, we run the body of L3:
        (set! value (list new))
    We find the variable VALUE in E2, so we change it to point to the list
containing the word HELLO. So, we have VALUE point to the box-and-pointer
diagram representing the list (HELLO) (since in the directions we specified
to draw lists as box-and-pointer diagrams).

Grading Rubric:
```

6 points: "Perfect"
    - Completely Correct.
5 points: Trivial Mistakes
    - Forgetting arrow heads
    - Reversing arrow heads
    - Forgetting to draw (HELLO) as a box-pointer diagram
4 points: "The Idea"
    - The correct number of lambda bubbles and environments
    - Still a valid environment diagram (i.e things point to 'sensical'
things.
    - Forgot to set! VALUE from () to (HELLO)
    - may be missing a binding
2 points: "An Idea"
    - "Locally correct" environment (in other words, all arrows point to
      "sensical" things for an environment diagram).
0 points:
    - Any environment pointing to a lambda bubble (instead of an
environment).
    - No lambda bubbles
    - Variables without bindings.

4. Streams (6 points)

The problem was asking for a stream, where each element was a procedure that
would take a list as argument, and mimic the functionality of car/cadr/caddr/
etc. More strictly (since cdddddr, ddddddr, etc. does not exist), it would
return a stream of procedures that had the functionality of (lambda (lst) (car
lst), (lambda (lst) (car (cdrlst))), (lambda (list) (car (cdr (cdrlst)))),
etc.

There were multiple ways to do this both correctly and incorrectly. A list of
sample correct solutions follow in order of code elegance:

(a) one stream-map, where each element of GETTERS is the composition of CDR and
the previous element in GETTERS.

```
(define getters
  (cons-stream car (stream-map (lambda (fn) (compose fncdr)) getters)))
```

where compose was defined in SICP as:

```
(define (compose f g)
  (lambda (x)
    (f (g x))))
```

(b) one stream-map, where the compose was expanded in the stream-map

```
(define getters
  (cons-stream car (stream-map
                     (lambda (fn) (lambda (lst) (fn (cdrlst))))
getters)))
```

(c) two stream-maps, one to generatercdr/cddr/cdddr/etc. (call it cdrs), and
one within getters, which adds a car to each element of the first stream.

```
(definecdrs
  (cons-streamcdr (stream-map
                     (lambda (fn) (compose cdrfn))
cdrs)))
(define getters
  (cons-stream car (stream-map
                     (lambda (fn) (compose car fn))
cdrs)))
```

(Although this solution uses COMPOSE for clarity, it was more common to see it
expanded out.)

(d) An iterative approach (yuck!) Note that this solution is very similar to the
other solutions, but does not fully take advantage of the power of stream-map.
This uses a helper function that returns the correct number of composed cdrs,
depending on the number passed in, and then uses a stream-map to add the car.

```
(define ones
  (cons-stream 1 ones))
(define integers
  (cons-stream 0 (stream-map + ones integers)))
(define (getters-helper n)
  (if (= n 0)
      (lambda (lst) lst)
      (lambda (lst) (cdr ((getters-helper (- n 1)) lst)))))

(definecdrs (stream-map getters-helper integers))
(define getters (stream-map (lambda (fn) (compose car fn)) cdrs))
```

Most incorrect solutions involved either:
(a) a poor understanding of what arguments car/cdr can take
(b) wrong return values
(c) a poor understanding of how to construct a stream
The problems was asking for a stream, whose elements were *procedures*.
Furthermore, each procedure had to be a procedure that took one argument (a

list). Other common mistakes include creating an incorrect stream (car cdrcdr cdr ...) or forgetting one of the elements (e.g. (car caddrcadddr ...)). A listing of some common wrong solutions and the reason for their wrongness follow:

(a) By far, the most common wrong answer involved nesting cars/cdrs, where CDR or CAR was offered as an argument to another CDR/CAR. The procedures CDR and CAR take lists, not other functions as arguments!

```
(define getters
  (cons-stream
car
    (stream-map
      (lambda (fn) (fncdr))  ; the fn will be a car/cdr/etc. it cannot take
                              ; acdr as an argument! use *compose* (see sol 1)
getters)))
```

(b) Another common wrong answer was to return a stream of lists of cars/cdrs. This is not what the problem was asking for.

```
(define getters
  (cons-stream
    (list car)                      ; wrong type of value
    (stream-map
      (lambda (fn) (cons fncdr))   ; wrong type of value
getters)))
```

(c) Another common wrong answer, similar to the last wrong answer, was to return a stream of words car/cadr/caddr/cadddr, etc. Not only is this the wrong return value, but there are no such procedures defined past 4 Ds. (There was an extra-for-experts problem way back to return such a procedure given its name.)

Scoring:
6 Perfect
5 Trivial mistake: syntax error, etc.
4 The idea: infinite stream of elements that resemble car/cadr/caddr/etc. in the proper order, including: improperly composed cars/cdrs, lists of cars/cadr/etc., or words car/cadr/etc.)
  - If improperly composed cars/cdrs were returned, adding the function "compose" (see solution 1 above) should make the program run-the program has no other errors.
  - If other wrong values were returned, the program should still run without error-the only error is the type of the values in the returned stream.
2 An idea: any valid infinite stream.
0 Does not return a stream or used list-ref.

5. List (Pair) Mutation (8 points)

Question 5
(8 Points)

The problem was to correct ALL sublists so that each ends in a null (so they are proper lists). We start with some correct solutions.

A correct solution would be the following:

```
(define (deep-fix! ls)
  (cond ((atom? ls) ls)                          ; atom? catches anything that
                                                 ; is not a list or pair.

        ((not (pair? (cdrls)))                  ; Make sure we fix the cdr of
         (set-car! ls (deep-fix! (carls)))      ; any pair that is not a list.
         (set-cdr! ls '())
ls)
        (else                                    ; Otherwise, handle both the
         (set-car! ls (deep-fix! (carls)))      ; car and the next pair in the
         (deep-fix! (cdrls))                    ; current list
ls)))
```

A more direct, and common, correct solution was the following:

```
(define (deep-fix! ls)
  (cond ((null? ls) '())
        ((not (pair? (cdrls)))
         (set-car! ls (if (pair? (carls))
                          (deep-fix! (carls))
                          (carls)))
         (set-cdr! ls '())
ls)
        (else
         (set-car! ls (if (pair? (carls))
                          (deep-fix! (carls))
                          (carls)))
         (deep-fix! ls)
ls)))
```

This, of course, could be broken up into simpler chunks like the following solution:

```
(define (deep-fix! ls)
  (fix-car! ls)
  (fix-cdr! ls)
ls))

(define (fix-car! ls)
  (if (pair? (carls))
      (deep-fix! (carls))
      'okay))

(define (fix-cdr! ls)
  (if (not (pair? ls))
      (set-cdr! ls '())
      (deep-fix! (cdrls))))
```

It's worth noting that we aren't using LIST? for the checks, we're using PAIR? This is because we want the predicate to be true for things like (5 . 47). However, we really didn't take off points for that sort of mistake.

Also note that you don't actually need to use SET-CAR!. If the car is an atom, it will stay the same. If it's a pair, the pair might change, but it won't be replaced by an atom or another pair. You only had to make sure the overall return value from DEEP-FIX! was the original input pair.

Finally, note that in this version DEEP-FIX! doesn't check for an empty list. This is because the question said that DEEP-FIX! takes a pair as input, and the empty list is not a pair. FIX-CAR! and FIX-CDR! then only call DEEP-FIX! on pairs. Including a call to NULL?, however, (or ATOM?) wouldn't break anything, and might be necessary depending on how you write your recursive calls.

The most common wrong answers looked something like these:

```
; WRONG ANSWER
(define (deep-fix! ls)
 (cond ((null? ls) '())
       ((not (pair? (cdrls)))
        (set-cdr! ls '())                      ; FORGOT TO CHECK THE CAR
ls)
       ((pair? (carls))
        (set-car! ls (deep-fix! (carls)))
        (deep-fix! (cdrls))
ls)
       (else (deep-fix! (cdrls)) ls)))

; WRONG ANSWER - This seems to act like you can do more than one cond clause
;                in one call?
(define (deep-fix! ls)
  (cond ((null? ls) '())
        ((not (pair? (cdrls)))
         (set-cdr! ls '())
ls)
        ((pair? (carls))
         (set-car! ls (deep-fix! (carls)))
ls)
        (else (deep-fix! ls) ls)))
```

Here's the rubric we used:

8 Points - "Perfect"

7 Points - "Trivial Mistakes"
    * Only worked for lists of numbers.
    * Forgot to return the list.

5 Points - "The Idea"
    * Recurses on the car and cdr + uses set-cdr! on what 'appears' to be the last pair in the list
    * There was no use of set!

4 Points
    * Used a solution involving MAP that would have worked if it were a FOR-EACH (this was a very small edge case)

3 Points - "An Idea"
    EITHER:
    * Appeared to have the right idea but does something wrong not mentioned for the higher scores.
    * Did deep recursion correctly but didn't call set-cdr! properly.
    * Used set! on the last cdr: ((atom? ls) (set! ls '()))
    OR:
    * Does correct set-cdr!, but forgot to recurse for some case.
    * Destroyed the list

1 Point
* set-cdr!'s the first pair of a bad list and does not perform deep recursion correctly.

6. Vectors (8 points)

The standard correct solution searches for an item in the vector between
position i and j (inclusively).

```
(define (member-in-range? vec item i j)
  (cond ((> i j) #f)
          ((equal? item (vector-ref vec i)) #t)
          (else (member-in-range? vec item (+ i 1) j))))
```

OR

```
(define (member-in-range? vec item i j)
  (and (not (> i j))
       (or (equal? item (vector-ref vec i))
              (member-in-range? vec item (+ i 1) j))))
```

These solutions are all very similar and they get full credit (along with their
close variations).

We also gave full points for the following solution that create a list with the
the vector elements in range and use MEMBER to determine if the list contains
the item we are searching for. This is not a good way to do vector operations,
though!

```
(define (member-in-range? vec item start end)
  (define (helper i)
    (if (> i end)
          '()
          (cons (vector-ref vec i) (helper (+ i 1)))))
  (member item (helper start)))
```

Some students wrote some less efficient solutions that checks the whole vector instead of just the elements in range.

```
(define (member-in-range? vec item start end)
  (define (helper i)
    (cond ((= i (vector-length vec)) #f)
            ((and (>= i start) (<= i end) (= (vector-ref vec i) item)) #t)
            (else (helper (+ i 1)))))
  (helper 0))
```

This solution checks the whole vector with this more complicated solution.

A lot of student made the trivial mistake of checking for (= start end) instead
of (> start end). This will lead to the exclusion of the element at the last
index in the range.

A uncommon mistake that will cost student more points:

```
(define (member-in-range? vec item start end)
  (cond ((> start end) #t)
          ((not (= (vector-ref vec start) item)) #f)
          ((= (vector-ref vec start) item) #t)
          (else (member-in-range? vec item (+ 1 start) end))))
```

This solution has the main idea but stops short at the first element that
doesn't match the item.

Solutions that get 0 include:
- try to use CAR, CDR on vectors
- use of (non-existent) VECTOR-MEMBER? or VECTOR-FILTER?

8 - perfect
7 - trivial mistake
4 - the idea
0 - ....