CS 61A Fall 2009 Midterm 1 solutions

1. What will Scheme print?

```
(every (lambda (x) (se x x))
       (keep (lambda (x) (even? (count x)))
           '(and your bird can sing)))
```

Answer: (your your bird bird sing sing)

The function used as argument to KEEP selects words with an even number of letters, and so the KEEP returns (your bird sing). Then EVERY repeats each word twice. Because EVERY uses SENTENCE as its combiner, the results are flattened into a sentence. If we'd used MAP instead of EVERY, the result would have been ((your your) (bird bird) (sing sing)).

```
((lambda (x y) (x (y 3)))
 (lambda (x) (* x x))
 (lambda (x) (+ x 6)))
```

Answer: 81

By substitution, this is equivalent to

```
   ((lambda (x) (* x x)) ((lambda (x) (+ x 6)) 3))
    --------------------  --------------------
      subst for X             subst for Y
```

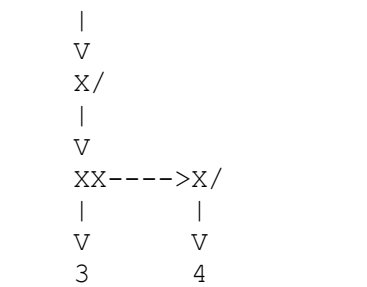
The inner procedure call ((lambda (x) (+ x 6)) 3) returns 9. So the outer procedure call returns (* 9 9) = 81.

Scoring: One point each. If you put quotation marks in your answers, you lost only one point on questions 1 and 2 for that even if you did it more than once.

2. Box and pointer diagrams.

```
(list (list (cons 3 (list 4))))
```

Answer: (((3 4)))

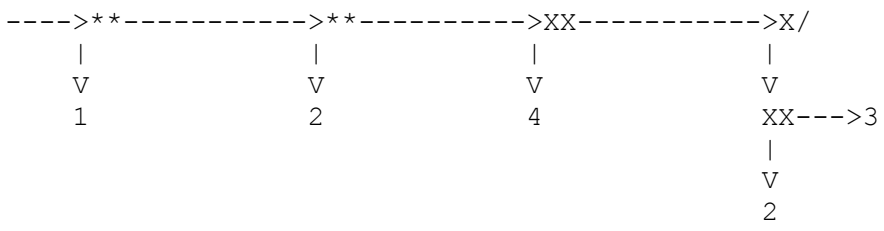


The two troublesome things here are understanding how to represent a one-element list (namely, as a single pair whose CAR is the one element and whose CDR is the empty list), and understanding that CONS just makes one pair, so it sticks the new element 3 at the front of the list (4), giving (3 4). So (LIST (CONS ...)) gives the one-element list ((3 4)), and (LIST (LIST (CONS ...))) gives the one-element list (((3 4))).

A tempting wrong answer is (((3 . (4)))), thinking that anything created by CONS will print with a dot in the middle. But Scheme doesn't remember how a pair is created, and it never prints a dot followed by an open parenthesis.

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

Answer: (1 2 4 (2 . 3))



The call to CONS makes the bottom pair (2 . 3); the call to LIST makes the two-element list (4 (2 . 3)). Dotted pairs can be elements of lists! (And that doesn't make the overall list improper.)

The pairs marked as ** instead of XX are /copies of/ the spine of the list created by (LIST 1 2), since APPEND copies the spines of all but its last argument. In this case that doesn't matter, because we don't use the original list for anything after the APPEND is done.

Scoring: One point per print form, one point per box and pointer diagram. One common mistake was not putting the start arrow to the first box in the diagram.

3. Normal and applicative order.

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

(square (+ 2 3))
```

Normal order: The actual argument /expression/ (+ 2 3) is substituted into
the body of SQUARE:

```
       (* (+ 2 3) (+ 2 3))
```

So + is called twice.

Applicative order: This is the way Scheme really does it; we start evaluating
the call to SQUARE by /evaluating/ its actual argument expression; computing
(+ 2 3) gives the value 5. Then we substitute the actual argument /value/
into the body of SQUARE:

```
       (* 5 5)
```

So + is called just once.

Scoring: One point each.

4. Orders of growth.

```
(define (mystery n)
  (cond ((< n 0) 0)
      ((odd? n) (+ 2 (mystery (- n 2))))
      (else (+ 1 (mystery (- n 1))))))
```

You might have to think a little to know /exactly/ how many recursive calls
are made; the crucial point is that the argument can be even only once, so
it's N/2 calls, or maybe (N/2)+1. But without reasoning that out, it's clear
that /at worst/ we make N recursive calls, so no matter what, this is Theta(N).

The procedure generates a recursive process; each recursive call to MYSTERY is
enclosed in a larger expression (+ 1 (MYSTERY ...)) or (+ 2 (MYSTERY ...)), so
there's still work to do after that recursive call returns.

```
(define (bar n)
  (if (or (> n 10) (< n 0))
      n
      (bar (+ n 1))))
```

This one was a little tricky; you had to read the problem carefully. If N is

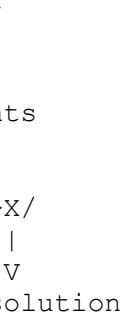
more than 10, the procedure just returns N right away, without doing any recursion. And if N is less than 10, there can be only a limited number of recursive calls, at most 10 of them. So this has a constant upper bound on the amount of time required, so it's Theta(1) despite the appearance of a linear recursion, and despite the unusual counting up rather than down in the recursive call.

The recursive call is the outermost procedure call in one branch of the IF, so it's the last thing the procedure has to do -- a tail call. Therefore this generates an iterative process.

Scoring: One point for each order of growth; one point for each recursive vs. iterative process.

5. Data abstraction.

(a) It might help to draw a box and pointer diagram for the result of calling the constructor:



QUESTION is the CAR of the pair marked **, which is in turn the CAR of the entire structure. So we want the QUESTION selector to be

```
(define question caar)
```

SOLUTION is the second element (the CADR) of that sublist, so its the CADR of the CAR of the whole thing:

```
(define solution cadar)
```

Finally, POINTS is the second element of the big list:

```
(define points cadr)
```

It's fine if you said, e.g.,

```
(define (points problem)
  (cadr problem))
```

The most common mistake was not realizing that cdr /always/ returns a list,
so many people put cdr instead of cadr for points.

Scoring: One point each.

(b) Add up the POINTS of a list of problems.

```
(define (total-points problems)
  (accumulate + 0 (map points problems)))
```

Or, to do it the harder way,

```
(define (total-points problems)
  (if (null? problems)
      0
      (+ (points (car problems))
       (total-points (cdr problems)))))
```

Note that PROBLEMS is a list of problems, not a problem, so we use CAR, CDR,
etc. to look inside it. But (CAR PROBLEMS) is a Problem, so it would be a
data abstraction violation to use CAR and friends on it; we use the selector
POINTS that was defined for that abstract data type.

Scoring:

4 correct.
3 Missing base case or (null? (cdr problems)).
2 works but violates data abstraction.
0 doesn't work.

(c) How to change representation?

Nothing else changes! That's the whole idea of data abstraction.

Scoring: Two points, all or nothing.

6. Recursion vs. higher order functions

(a) recursion

```
(define (appearances sent wd)
  (cond ((empty? sent) 0)
      ((equal? (first sent) wd)
```

```
       (+ 1 (appearances (bf sent) wd)))
      (else (appearances (bf sent) wd))))
```

This is a KEEP-like structure with three COND clauses, because we're checking whether each word of the sentence satisfies some condition, but we just update a count rather than collecting a sentence of matching words.

(b) higher order functions

This can be done in different ways. The first version makes an intermediate result that's a sentence of copies of the desired word, such as (LOVE LOVE):

```
(define (appearances sent wd)
  (count (keep (lambda (w) (equal? w wd)) sent)))
```

The second approach makes an intermediate result that's a sentence of zeros (for non-matches) and ones (for matches), such as (0 1 0 0 0 0 1 0):

```
(define (appearances sent wd)
  (accumulate + 0 (every (lambda (w) (if (equal? w wd) 1 0)) sent)))
```

Many students thought that this question asked for an iterative process version of the solution. Iterative processes are NOT higher order functions.

Scoring: 6 points for each half, on this scale:

6 correct
5 trivial error / DAV / member? instead of equal?
3-4 more significant error, but has the idea
1-2 has an idea
0 other

7. HTMLcheck

One small detail that can be troublesome here is that if an end instruction is seen when there are no pending start instructions, the program is going to want to compare the end instruction against the nonexistent matching start instruction, and end up trying to take the cdr of the empty list. To avoid that, instead of putting in a special check for it, the program below starts

out its list of currently active start instructions with something that can't
match any start instruction, namely the empty list (but any non-word would
do). So if the first thing seen is an end instruction, the program compares
its first word with the empty list that it finds in (car current). It then
correctly returns #F instead of blowing up. But you could just test for an
empty list of active start instructions instead.

```
(define (htmlcheck text)
  (define (help text current)
    (cond ((null? text) (null? (cdr current)))
        ((list? text)
         (if (equal? (first (caar text)) '/)
             (if (equal? (bf (caar text)) (car current))
               (help (cdr text) (cdr current))
               #f)
             (help (cdr text) (cons (caar text) current))))
        (else (help (cdr text) current))))
  (help text '(()) ))
```

The first COND clause says that when we run out of text, the list CURRENT
should contain only the one element, the empty list, that it started
with.

The second COND clause handles HTML instructions. If (CAR TEXT) is a
list,
i.e., an HTML instruction, then (CAAR TEXT) is the first word of that
instruction. We want to know if the first letter of that word is a
slash,
in which case it's an end instruction. In a case like this I find the IF
nested inside a COND clause clearer than having clauses of the form

```
   ((and (list? (car text)) (equal? ...)) ...)
   ((and (list? (car text)) (not (equal? ...)) ...)
```

but there are many correct ways to organize it, as long as all the
possible
cases are considered.

One common mistake was that students checked only that it was a
palindrome,
such as ((A) (B) (/B) (/A)), whereas ((A) (/A) (B) (/B)) is also a valid
text.
These solutions got a maximum of 3 points. Other students only retained
the
last open tag and didn't maintain a list of open tags.

Scoring:
6 completely correct
5 trivial error; didn't check for an end instruction before a start one.
4 uses the list correctly
3 palindrome; has some list of tags

2 has an idea
1 has an idea
0 other