CS 61A Spring 2009 Midterm 1 solutions

1. What will Scheme print?

```
(every (lambda (x) (se x x))
       (map (lambda (x) (* x x)) '(1 2 3)))
```

Answer: (1 1 4 4 9 9)

The result of the MAP call is (1 4 9), and the EVERY call makes a sentence in which every number appears twice. The main point of this problem is that if we'd used MAP instead of EVERY, the answer would have been ((1 1) (4 4) (9 9)), but EVERY uses SENTENCE to combine the individual results, and SENTENCE flattens out its argument sentences.

```
(keep (lambda (x) x) (keep even? '(1 2 3 4 5 6 7)))
```

Answer: (2 4 6)

The inner KEEP call keeps even numbers, so it returns (2 4 6). The interesting part of the question is the outer KEEP, because its first argument isn't a conventional predicate function; it's the identity function. But in Scheme everything other than #F counts as true, so KEEP keeps all the numbers in the sentence.

```
(let ((first last) (last first))
  (last (first '(for no one))))
```

Answer: o

The main point of this question is that LET doesn't carry out the specified bindings one by one, but rather all at once. So the binding (LAST FIRST) binds the name LAST to /the original, global/ value of FIRST, not to the rebound FIRST set up by the (FIRST LAST) binding. Thus, the body of the LET returns the first letter of the last word of the argument sentence.

```
(cadadr '((a b c d e) (f g h i j) (k l m n o)))
```

Answer: g

The quick way to do this problem is to recognize that CADR means "the second element," and CADADR means the CADR of the CADR, so

it returns the second word [G] in the second sublist [(F G H I J)]
of the argument. If you didn't think of that, you can work out the answer by doing the CARs and CDRs from the inside out, i.e., starting with the /rightmost/ A or D in the name -- in this case, a D, for CDR:

```
(cdr '((a b c d e) (f g h i j) (k l m n o)))  ==>
                ((f g h i j) (k l m n o))
(car '((f g h i j) (k l m n o))  ==>  (f g h i j)
(cdr '(f g h i j))  ==>  (g h i j)
(car '(g h i j))  ==>  g
```

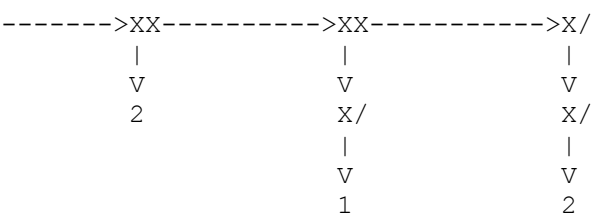
If you did the CARs and CDRs from left to right, you'd have tried to take the CDR of the word B and gotten an error.

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.

```
(append (cons '(1) '(2)) (list '(1) '(2)))
```

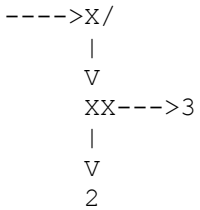
Answer: ((1) 2 (1) (2))



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 (1) at the front of the list (2), giving ((1) 2). The pair shown as ** above is /a copy of/ the one made by the CONS call, since APPEND copies the spine of all but its last argument.

```
(list (cons 2 3))
```

Answer: ((2 . 3))



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

Scoring: One point per print form, one point per box and pointer diagram.

3. Normal and applicative order.

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

(define (foo x y)
        (+ (square x) y))

(foo (square 1) (square 2))
```

Normal order: The actual argument /expressions/ (SQUARE 1) and (SQUARE 2) are substituted into the body of FOO:

```
        (+ (square (square 1)) (square 2))
```

+ is a primitive, so its argument expressions must be evaluated before Scheme can add the numbers. To evaluate (SQUARE (SQUARE 1)), we substitute the actual argument /expression/ (SQUARE 1) into the body of SQUARE (this is the first invocation of SQUARE):

```
        (* (square 1) (square 1))
```

Each of the actual argument expressions to *, namely (SQUARE 1) and (SQUARE 1) again, must be evaluated; these are the second and third invocations of SQUARE. Finally, we evalute (SQUARE 2), the second argument to +; that's the fourth and last invocation. So the answer is 4.

Applicative order: This is the way Scheme really does it; we start evaluating the call to FOO by /evaluating/ its actual argument expressions; these are the first and second invocations of SQUARE, with arguments 1 and 2. Then we substitute the actual argument /values/ into the body of FOO:

```
        (+ (square 1) 4)
```

Evaluating the body after substitution gives us the third and final invocation of SQUARE. So this answer is 3.

Scoring: Two points for each.

4. Orders of growth.

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

This was slightly more complicated than the usual sequential recursion because sometimes the recursive call reduced the argument by 1 and sometimes
it reduced it by 2. But that doesn't affect the order of growth.

Suppose /every/ recursive call reduced the argument by 1. Then clearly the
procedure's running time would be Theta(N).

Suppose /every/ recursive call reduced the argument by 2. Then you might
be tempted to say the running time is Theta(N/2), and that would be a correct answer, but Theta(N/2)=Theta(N), because constant factors don't matter in Theta notation.

The actual procedure does some of each. Does that change the order of growth? No. As it turns out, the (EVEN? N) test can succeed at most once, since if N is even then N-1 is odd, and every recursive call after the first one will reduce the odd argument value by 2, keeping it odd. But in any case the number of invocations would be somewhere between N/2 and N, so the procedure takes Theta(N) time.

To complete the analysis we'd have to ensure that all the primitives (EVEN?, <, +, and -) take constant time, but they do.

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.

Scoring: Two points for the order of growth; two points for recursive vs. iterative process.

5. Data abstraction.

```
(define (who-sang song)
  (define (helper song songs)
    (cond ((NULL? songs) #f)
```

```
          ((equal? song (TITLE (CAR songs)))
           (ARTIST (CAR songs)))
          (else (helper song (CDR songs)))))
  (helper song great-songs))
```

The variable SONGS is a list of songs, not a song. Everything whose abstract
type is "list of such-and-such" is a /list/, not a /such-and-such/, and so
it's appropriate to use the constructors and selectors for lists -- in this
program, CAR and CDR, in the contexts (CAR SONGS) and (CDR SONGS).

The value returned by (CAR SONGS) is a /song/, not a list, and so we should
use the selectors TITLE and ARTIST with it, not CAR and CDR.

(At the exam, someone complained rightly that SONG isn't a good formal parameter for this procedure, since the actual argument is a /title/ rather
than a /song/. That's a good criticism, but the examples should have made
it clear what we intended. And, by the way, TITLE would be an even worse
choice, since that would prevent us from using the selector TITLE inside the procedure! Something like SONG-TITLE would be best.)

Scoring: One point off for each data abstraction violation; each of the six capitalized words in the solution above has to be correct.

6. Recursion.

There are several possible approaches to this problem, but the cleanest result comes from treating finding the beginning of the omitted string and
actually removing the desired words as two different tasks, and writing a
helper function for the second of them:

```
(define (remove-n sent target n)
  (cond ((empty? sent) '())
        ((equal? (first sent) target)
         (nth-butfirst n sent))
        (else (sentence (first sent)
                        (remove-n (butfirst sent) target n)))))

(define (nth-butfirst n sent)
  (cond ((empty? sent) '())    ;; optional
        ((= n 0) sent)
        (else (nth-butfirst (- n 1) (butfirst sent)))))
```

The first COND clause in NTH-BUTFIRST is optional because we said you could
assume that if the target word is found, there are enough words remaining

in the sentence to satisfy the count N. So the second base case (= N 0) should be reached before the first one.

Note that after the desired words have been removed, there is no need to continue BUTFIRSTing down the sentence looking for the target word, since
you've been told it will only appear once.

Many people tried to do the finding and the removing in one procedure. This can be done if you use a helper with an extra argument to keep track
of whether you're in the finding part of the process or the removing part:

```
(define (remove-n sent target n)
  (define (helper sent n removing-flag)
    (cond ((null? sent) '())
          ((= n 0) sent)
          (removing-flag
           (helper (bf sent) (- n 1) #t))
          ((equal? (first sent) target)
           (helper (bf sent) (- n 1) #t))
          (else (sentence (first sent)
                          (helper (bf sent) n #f)))))
  (helper sent n #f))
```

Some people succeeded at this, but others tried to do it without something
like the REMOVING-FLAG variable. Instead, they did the EQUAL? test again
for each of the words that should have been removed, and so they didn't remove those words.

Some people who tried it without an NTH-BUTFIRST procedure also chose an iterative process, adding a RESULT-SO-FAR variable to which the words before
the target word were added. This can work, especially since the sentence
abstraction makes it easy to add each new word at the right end of the RESULT-SO-FAR, but (as you now understand since you know how sentences are
represented) adding words at the right instead of at the left causes the resulting program to take Theta(N^2) time instead of Theta(N) time. We don't take off points for inefficiencies in this course (unless a question
specifically asks about orders of growth), but it seems perverse to use the
more complicated iterative style in order to make the program run /slower/!

Scoring:

8 correct
7 trivial error
5 has the idea
2 has an idea
0 other

To count as having the idea, a solution must use recursion rather than higher order functions, and must have a structure that first looks for the target word and then counts down the words to be removed. (Exception:
an otherwise correct solution that uses (REPEATED BF N) to do the removal
but makes no other use of higher order functions gets 5 points.)

7. Higher order functions.

(a) The crucial idea here is that we need two nested calls to EVERY, one to choose a first letter, and one to choose a second letter:

```
(define (cross wd)
  (every (lambda (letter1)
           (every (lambda (letter2)
                    (word letter1 letter2))
                  wd))
         wd))
```

There's really no reasonable alternative to this solution, except that the
EVERY call to choose letter1 could be on the inside. That would change the
order in which the two-letter words appear in the result, but we accepted it.

Replacing the inner EVERY with MAP would be a data abstraction violation,
but would work. But replacing the /outer/ EVERY with MAP /would not/ work;
it would produce the result ((AA AB AC) (BA BB BC) (CA CB CC)) for the example invocation in the exam. We need the flattening (appending) effect
of EVERY. But we treated any use of MAP instead of EVERY as "having the idea" (4 points).

Some people seemed not to believe that EVERY's domain includes words as well as sentences. A few people have asked me, since the exam, when you learned that. I think this question betrays a lack of faith in abstraction,
or at least in the word/sentence abstraction. EVERY is built on top of that abstraction; specifically, that means that it uses the word/sentence
selectors FIRST, BF, etc., and you know that /those/ work both for words and
for sentences. Since you wrote EVERY yourselves (in the week 2 homework),
you know how EVERY is built on top of FIRST and friends. And you know that
it's "the word/sentence abstraction" because it's supposed to enable you to
think about words and sentences almost interchangeably. (The only exception

is that it wouldn't make sense for the WORD constructor to take a sentence
as an argument.)

The result of not thinking EVERY would work on words was that some students
felt the need to "explode" the words: to turn ABC into (A B C). The easiest
way to do that is (EVERY (LAMBDA (X) X) 'ABC), but you can't do that if you
don't think you can use EVERY on words. This led people to want to use
ACCUMULATE, which raises the interesting question of whether or not the
domain of ACCUMULATE includes words.

Sadly, this simple question has a complicated answer. SICP has two different
procedures named ACCUMULATE: the Chapter 1 (ACCUMULATE TERM A NEXT B) whose
domain is numbers, and the Chapter 2 (ACCUMULATE COMBINER BASE-CASE LST) whose
domain is lists (not sentences, let alone words, since SICP doesn't know about
the Berkeley word/sentence abstraction). Further complicating things, CS 3
uses a (ACCUMULATE COMBINER WD-OR-SENT) version. The Berkeley startup file
for STk defines a version of ACCUMULATE that sees how many arguments you gave
it; if two arguments, it has the CS 3 word/sentence domain, but if three
arguments, it has the SICP list domain.

Uncertainty on this point led some students to write recursive helpers just
to explode a word. We didn't take points off for this if it was the only
use of recursion, nor did we penalize any assumption about the domain of
ACCUMULATE.

(Of course there's no possibility of confusion about the /range/ of
ACCUMULATE, since that's determined by the combiner function you provide
as one of its arguments.)

Some people, even people with correct solutions, surrounded one or both
of the EVERY invocations with an invocation of SENTENCE:

```
(se (every ...))
```

We think that they intended this as a sort of type declaration: take the
result from EVERY and consider it a sentence. But we're not sure. In any
case, this isn't an error, but since EVERY already returns a sentence it
doesn't do anything, and it isn't needed as part of data abstraction
either.

Many people understood that two nested calls to EVERY were needed, but
wrote them this way:

```
        (every (lambda ...) (every ... wd))   ; wrong
```

instead of this way:

```
        (every (LAMBDA ... (EVERY ... WD)) wd); right
```

The incorrect version would take each letter of the word, compute some function of that letter, then compute some other function of that result.
Ultimately it would give a result of length N, for an N-letter word. The
correct version, in which the function computed for each letter computes another function /also for each letter/, gives a result of length N^2, as desired. That's a very significant difference, and we counted this error as "an idea" rather than "the idea."

A few people had one correct call to EVERY, but used recursion instead of
the other EVERY call. We considered this too as "an idea."

A few people knew that they had to avoid recursive calls, but wrote code making it clear that they were trying to follow a recursive pattern, doing
everything but the recursive call itself, things like

```
        (se (cross-helper (first wd)) (bf wd)); huh?
```

when we're pretty sure they were really thinking

```
        (se (cross-helper (first wd)) (cross (bf wd)))       ;
recursive
```

Any solution that said (FIRST WD) in it was automatically considered suspicious; the only allowable context for this was the EXPLODE function as mentioned earlier.

Scoring:

6 correct
5 trivial mistake
4 has the idea
2 has an idea
0 other

(b) The point of this part was for you to see that it asks you to generalize
your answer to part (a). So the structure is very similar, except that it's
surrounded by a wrapper that accepts FN as its argument:

```
(define (make-crosser fn)
  (lambda (wd)
    (every (lambda (letter1)
             (every (lambda (letter2)
                      (FN letter1 letter2))
                    wd))
```

```
          wd)))
```

The capitalized FN is the only change in the body of the inner function, compared to the body of CROSS in part (a).

In scoring part (b), we focused mainly on the first two lines in the definition above: MAKE-CROSSER takes a function as argument and returns another function; that latter function takes a word as its argument. If you got that right, and then in the body of that inner function repeated the same mistake you made in part (a), you weren't penalized a second time for the same mistake. (We did check, though, to make sure that you changed your inner function to call FN instead of calling WORD.)

It turns out there's another way to write MAKE-CROSSER: instead of copying
code from CROSS, you can /invoke/ CROSS and then further process that result:

```
(define (make-crosser fn)
  (lambda (wd)
    (every (lambda (wrd) (fn (first wrd) (last wrd)))
           (cross wd))))
```

Notice that WD is the parameter to the outer lambda, and WRD the parameter
to the inner lambda. WRD represents one of the N^2 words generated by CROSS.

A few people who had the idea of writing it that way forgot that FN expects
two separate one-letter arguments, so they said

```
        (every FN (cross wd))
```

or the equivalent

```
        (every (lambda (wrd) (fn wrd))
               (cross wd))
```

We counted this as having the idea.

Scoring:

4 correct
3 trivial mistake
2 has the idea
1 has an idea
0 other