CS61A Solutions to mt1, Spring 2010

1. What will Scheme print?

```
> (keep (lambda (wd) (= (count wd) 3)) '(for no one))
(for one)
```

This was a reasonably straightforward expression; it looks at each word in the input sentence and keeps the ones with three letters.

As with many other examples in CS61A, "For No One" is the name of a Beatles song, from the album _Revolver_.

```
> (define (f x) (lambda () (se 'hello x)))
> (first (f 'brian))
ERROR
```

This expression was designed to be a bit tricky, but also to highlight how important it is to know what the domain and range of your functions are, especially when they are higher-order functions!

Let's read it carefully. "Define F to be a procedure of one argument, X, whose /result/ is [a procedure of no arguments whose result is the sentence made by combining the word HELLO with the value of X]." In other words, the value of (f 'brian) is going to be a procedure. And we certainly can't apply FIRST to procedures!

A lot of people put HELLO as the answer. Remember that Scheme does exactly what you tell it to, not what you may have actually meant!

```
> (define (g x y) (x (se 'hello y)))
> (g first 'satish)
hello
```

Similar to the previous problem, but this one actually does things

correctly. Again, let's look at the definition of G very carefully:
"Define G to be a procedure of two arguments, X and Y, whose result is [the result of applying X to the sentence made by combining the word HELLO with the value of Y]".

When we call G, X gets bound to the FIRST procedure and Y to the word SATISH. We then call FIRST on the sentence (HELLO SATISH), and get the word HELLO.

```
> (every (* 2) '(1 2 3 4))
ERROR
```

The point of this question was to remember that EVERY takes a /procedure/ as its first argument, not an expression. (You can't pass around expressions in Scheme.) Some people really want this to work, and give the sentence (2 4 6 8) as a result. It doesn't, and doesn't really make sense if you remember the definition of EVERY:

```
(define (every fn sent)
  (if (empty? sent)
      '()
      (se (fn (first sent)) (every fn (bf sent))) ))
```

We call FN on the first item in the sentence. What's FN? Well, it had better be a procedure!

A few people said (2 2 2 2) as their result, presumably because (* 2) evaluates to 2. But the /number/ 2 is still very different than a /procedure returning 2/, and you still get an error when you try to apply 2 to (first sent).

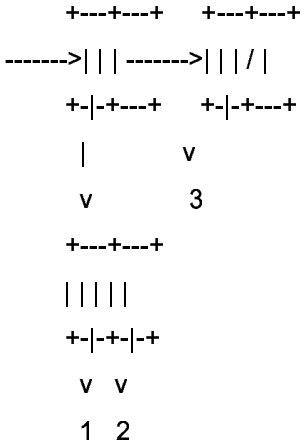
```
> (cadr '((a b c) d e f))
d
```

This was a pretty simple list question; the main test was whether or

not you remembered that CADR means "the CAR of the CDR" and not the other way around. You also had to notice that the first element of the input list was a sublist (a b c), and so the CDR of the whole list came out to (d e f). From there the CAR is clearly D.

2. Box-and-pointer diagrams

```
> (cons (cons 1 2) (cons 3 '()))
((1 . 2) 3)
```



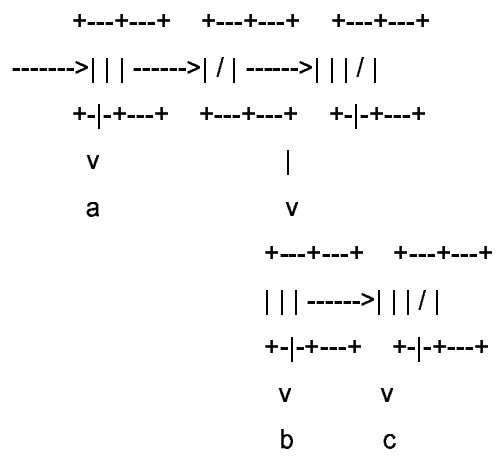
The easiest way to do this problem was to be very mechanical. For the expression, you start with the two inner CONSes and get (1 . 2) and (3 . ()), the latter of which is the same as (3). Then you CONS /those/ together and get ((1 . 2) . (3)), which is ((1 . 2) 3). Many people missed this and put ((1 . 2) (3)) instead.

The box-and-pointer diagram worked the same way -- remember, each CONS creates exactly one new pair! So you get the bottom pair and the right pair, then put them together with one more pair.

Scoring: 1pt for the value, 2pts for the box-and-pointer diagrams.
-1pt if you didn't have a start arrow

Given the list (a () (b c)), draw the box-and-pointer diagram for it, then

write an expression that creates it, using only quoted symbols, the empty list, and the CONS procedure.



This is the sort of problem you want to do "top-down" (or "outside-in"). The outside list has three elements, so you should start by drawing a "spine" of three connected pairs, where the last pair's CDR is the empty list.

The first element is the word A; that's easy.

The second element is an empty list. We allowed you to use either a slash, a pair of parens (), or even a separate slashed box to represent the empty list, but you had to be consistent! The empty list is the same thing whether it's marking the end of a list (CDR) or is an element of a list (CAR).

The last element is itself a list, this time with two elements. Again, you should start by drawing a two-pair spine, then fill in the element values afterwards.

A common mistake was to quote A, B, and C in your box-and-pointer diagram. This is INCORRECT! A box-and-pointer diagram represents /values/, not expressions, and the value of the *word* B is just B.

(cons 'a (cons '() (cons (cons 'b (cons 'c '())) '())))

Building the expression was mostly just an exercise in being careful; one CONS per box in the diagram, and make sure each CONS has a CAR and CDR specified. This time, you /had/ to quote the words A, B, and C, because they're being used in an expression. We didn't take off any points for not quoting the empty list, because STk is OK with that. But it's not actually in the Scheme standard.

If you messed up on your diagram, but wrote an expression that matched it, we gave you full credit for the expression.

Answers that used a quoted list or another list constructor besides CONS received no credit.

Scoring: 1pt for the box-and-pointer diagram, 2pts for the expression.
-1pt for not quoting the words A, B, and C in the expression.
-1pt if you didn't have a start arrow, unless you already lost a point for (2a)

3. Data Abstraction

Assume FOO is a /list/ of /sentences/. Write an expression that returns the final /letter/ of the first /word/ in the first /sentence/ of the list FOO.

(last (first (car foo)))

This was a pretty straightforward data abstraction question. Even though other answers would give you the "right" value (such as (last (caar foo))), that wasn't the point of the question! Such answers received no credit.

Scoring: 2pts, all or nothing.

Assume BAR is a /pair/ of /2D-points/. Write an expression to compute the slope of the line through the points. You must use the abstraction with constructor MAKE-POINT and selector X-COORD and Y-COORD.

```
(/ (- (y-coord (cdr bar)) (y-coord (car bar)))
   (- (x-coord (cdr bar)) (x-coord (car bar))) )
```

This one was a little trickier. There were two issues that tripped people up here. The first was the word "pair", which some people took to mean "two-element list". Whenever we use the word "pair" in Scheme, we always mean "the thing returned by CONS, a single box in a box-and-pointer diagram". Answers that used CADR to get the second point received no credit.

The second issue was with mentioning MAKE-POINT. We were trying to be clear about using the familiar data abstraction that had already come up in lab, but some students took it to mean "you must use MAKE-POINT", which is /not/ what we intended! Most of these attempts went awry, many to pass BAR as a lone argument to MAKE-POINT.

Solutions that created copies of the existing points and correctly used the selectors were given full credit, since it required /more/ work to stay within the abstraction that way.

Some students decided to write their answer as a procedure with a parameter BAR. This was fine.

A few answers used X-COORD and Y-COORD to extract the two points from BAR (then correctly using them on each point). This doesn't make any sense! A pair does not have x- and y- coordinates; it has a car and a cdr. The whole point of data abstraction is to use different language to talk about different kinds of data, /not/ that you should never use CAR and CDR.

Scoring: 2pts for a correct expression that respected data abstraction.
1pt for a solution that respected data abstraction but got the slope wrong (such as putting the difference of x-coordinates in the numerator).
0pts for a solution that violated data abstraction.

4. Orders of Growth

```
(define (append a b) ; helper function for INTEGERS-TO
  (if (null? a)
      b
      (cons (car a) (append (cdr a) b)) ))

(define (integers-to n)
  (if (= n 0)
      '()
      (append (integers-to (- n 1)) (list n)) ))
```

INTEGERS-TO is Theta(n^2).

Let's start with APPEND. Every time it calls itself, the first argument becomes one element shorter: (append (cdr a) b). So it will call itself one time for each element, which means it ends up with as many recursive calls as the first list is long. It only does constant-time operations inside the body (besides the recursive call), so we can say APPEND is Theta(length of first argument).

Now, how calls to APPEND are there going to be? Well, INTEGERS-TO subtracts 1 from its argument N with every recursive call, so there are going to be N recursive calls -- each of which performs a call to APPEND. At this point you can already guess that the runtime is going to be Theta(n*n) = Theta(n^2).

If you're being careful, you'll realize that APPEND doesn't always get called with lists of the same length! When N is 10, the first argument to APPEND is (integers-to 9), which has 9 elements. But when N is 3, the first argument is (integers-to 2), which has 2 elements! So, to be very careful we have to add these up:

(n-1) + (n-2) + (n-3) + ... + 2 + 1

If you remember your math, this series comes out to (n^2)/2, which is

still Theta(n^2).

The most common incorrect answer was Theta(n), probably from people who didn't catch that APPEND was Theta(n) itself, or who thought that the fact that the size of the list doesn't stay the same would cancel out the effect.

```
(define (mystery x)
  (cond ((= x 0) 0)
        ((= x 1) 1)
        (else (+ (mystery (- x 1))
                 (mystery (/ x x)) ))))
```

MYSTERY is Theta(n)

This one was tricky! There are two recursive calls, so you'd be tempted to put Theta(2^n). But take a look at them. (mystery (- x 1)) is a standard recursive call; it will take X-1 recursive calls before reaching a base case.

(mystery (/ x x)) is different. Why? Because (/ x x) is always 1, and (mystery 1) is a base case. So that call takes constant time. You can almost think of it as not a recursive call at all; it is, but it doesn't contribute to the order of growth.

Scoring: 2pts each, all or nothing

5. Iterative vs. Recursive Processes

```
(define (slow-multiply a b)
  (define (slow-multiply-internal a b result)
    (cond ((= a 0) result)
          (else (slow-multiply-internal (- a 1) b (+ result b))) ))
  (slow-multiply-internal a b 0))
```

SLOW-MULTIPLY generates an iterative process.

This is a prototypical example of an iterative procedure.
SLOW-MULTIPLY certainly doesn't do any work; it simply calls
SLOW-MULTIPLY-INTERNAL right away and returns its answer.

SLOW-MULTIPLY-INTERNAL has a base case and a recursive case. The base case is just returning a value, so it's not going to mess up a recursive process. The recursive case makes a single recursive call, and moreover, it's a /tail call/, meaning it's the /last/ thing to do in computing SLOW-MULTIPLY-INTERNAL.

A lot of iterative procedures use helper procedures with a RESULT parameter. This way, the base case can just return the RESULT once everything is done, and all the recursive calls can rely on that.

```
(define (faster-multiply a b)
  (cond ((= a 0) 0)
        ((odd? a) (+ (faster-multiply (quotient a 2) (* b 2)) b))
        (else (faster-multiply (quotient a 2) (* b 2))) ))
```

FASTER-MULTIPLY generates a recursive process.

In this case, we don't have any helpers; instead we just have a base case and two recursive cases. The first recursive case calls FASTER-MULTIPLY, then adds B to the result. Because there is more work to do after the recursive call repeats, this is /not/ a tail call, and thus the function does not generate an iterative process.

The second recursive case /is/ a tail call, but that doesn't matter at this point, since the first case already needs to do work after the recursive call finishes.

Scoring: 2pts each, all or nothing.

6. Higher-order Procedures

Write a procedure VOWEL-COUNTS that takes a sentence and returns a sentence of the number of vowels in each word. (The CS61A TAs like to refer to this as "evoweluation".) Your solution should use only *higher-order procedures* (no recursion)!

```
(define (vowel-counts sent)
  (every (lambda (x) (count (keep vowel? x))) sent) )
```

This was the most compact solution; another common one used two calls to EVERY (one for keeping vowels and one for counting the remaining letters).

Most people got the basic ingredients: KEEPing only vowels in EVERY word, and COUNTing them. Common mistakes included using KEEP where an EVERY was needed or vice versa, or correctly KEEPing the vowels, but then COUNTing the words in the sentence instead of vowels in each word.

People who forgot that they could use COUNT could still get a correct answer with a more roundabout solution like this:

```
(define (vowel-counts sent)
  (every (lambda (wd)
           (accumulate +
                       0
                       (every (lambda (letter)
                                (if (vowel? letter) 1 0) )
                              wd))
           sent))
```

Unfortunately, it was hard to get this right because of the way that EVERY works. Some answers tried to convert each word to a sentence of numbers using EVERY, then use another EVERY to do the accumulation. The trouble is, EVERY returns a /sentence/, so each word's sentence of numbers gets flattened into one big sentence afterwards, losing any

internal structure.

A few answers had expressions like in problem (1d), where there was a bare expression with no lambda. (every (keep vowel? wd) sent). This doesn't make sense; where did WD come from? Remember that KEEP and EVERY (and their list friends FILTER and MAP) take a /procedure/ as their first argument.

Scoring:
8 Perfect
7 Trivial mistakes
4-5 "The idea"
5 Reversed COUNT and EVERY (counted words instead of letters)
5 Egregious paren issues (usually with open parens, not close parens)
3 Used a bare expression instead of a lambda in a call to KEEP or EVERY
1-2 "An idea"

7. Recursion

Write a procedure CONTAINS-PHRASE? that takes two sentences, and returns #t if the first sentence contains the second sentence in order (but not necessarily consecutively).

```
(define (contains-phrase? sent phrase)
  (cond ((empty? phrase) #t)
        ((empty? sent) #f)
        ((equal? (first sent) (first phrase))
         (contains-phrase? (bf sent) (bf phrase)) )
        (else (contains-phrase? (bf sent) phrase)) ))
```

There are two base cases to think about here! First, we only know that the sentence actually does contain the phrase when the entire phrase has been accounted for, which is the first case. And we only know that the phrase is /not/ in the sentence when the phrase has a word that's not in the rest of the sentence at all. Some people used MEMBER? to check this, but it was easier to just wait until SENT ran

out of words.

Getting these base cases backwards actually does give incorrect results! Consider the case where the last word in the phrase is the last word in the main sentence -- both arguments will be empty. We have to be sure to return #t in this case.

The recursive cases are similar; either we've found the first word in the phrase or we haven't. Either way, we're done with the word in the main sentence. Some people tried to be smart and actually found out where the next word /was/, and used either a helper procedure or the library procedure SUBSEQ to chop off all words up to that point. But that wasn't necessary either.

The most common mistake involved the very intelligent idea to KEEP only those words in the main sentence that were part of the phrase, then see if there were the same number of words left over as there were in the phrase, or some variant of this idea. The main problem here is if the main sentence contains duplicates, in which case this procedure falls down, usually on one or both of the following cases:

```
> (contains-phrase? '(let let it be) '(let it be))
#t
> (contains-phrase? '(it it be) '(let it be))
#f
```

Remember, it's not enough to just have the examples working! We try to pick helpful cases, but in the end you still have to solve the problem as posed.

Scoring:
8 Perfect
7 Trivial mistakes
7 Base case mistakes (invalid, missing, or reversed base cases)
5 "The idea"
5 Any solution that worked correctly assuming no duplicates
4 Correct solution for checking if the first sentence /consecutively/

contained the phrase (a harder problem, but not what was asked for)

3 Correct solution for checking if the first sentence contained the words of the phrase in any order (an easier problem)

1-2 "An idea"