CS61A Spring 2008 Midterm 1 solutions

Note: If your individual score is I points (out of 40) and your group score is G points (out of 11), your overall score will be computed as

```
(max I (+ (* 0.9083333 I) (* 0.3333 G)))
```

The theory behind this formula is that 29 of your 40 individual points are not matched by anything in the group exam, and for the 11 points that are matched, we want to weight the exams as 2/3 individual, 1/3 group. So that gives (+ (* 29/40 I) (* 2/3 (* 11/40 I)) (* 1/3 G)), except that if your individual score is better than the combined score by this formula, we'll keep your individual score. But this correction is done at the end of the semester in the final grade-reporting program; GLOOKUP isn't smart enough to report anything but a fixed weighting of points.

1. What will Scheme print?

```
(every (lambda (x) (/ x 2))
       (keep even?
             (every (lambda (x) (* x x))
             '(2 3 4 5))))
```

Answer: (2 8)

The inner EVERY computes the square of each number, giving (4 9 16 25). The KEEP keeps the even ones, giving (4 16). Then the outer EVERY divides each of those numbers by 2.

```
(map (lambda (x) (se x x)) '(a b c))
```

Answer: ((a a) (b b) (c c))

MAP computes the given function for each element of its data list (A B C), and returns a list in which each element is the result of one of those function calls. For a three-element argument list, MAP always returns a three-element result list.

By the way, this isn't a data abstraction violation. Yes, since we're using MAP, we're thinking of (A B C) as a list, not as a sentence, but its elements are still words, so it's okay to call SENTENCE on them. The result is a /list of sentences/.

```
(every (lambda (x) (se x x)) '(a b c))
```

Answer: (A A B B C C)

The difference between this problem and the one before is that the overall result is put together using SENTENCE as the combiner, instead of using CONS to make a list. SENTENCE flattens the result, because it appends the sentences (A A), (B B), and (C C) into one big sentence.

```
(map (lambda (x)
       (let ((x (+ x 1))
             (y x))
         (* x y)))
     '(2 5))
```

Answer: (6 30)

Of course nobody would really write an expression like this in which the name X is used for two different variables. But we wanted to see if you understand how LET works. So, the MAP calls the LAMBDA procedure twice, once with X=2 and once with X=5. Here's what happens when I substitute 2 for X in all the places that use the X from the LAMBDA expression, rather than the X from the LET:

```
(let ((x (+ 2 1))
      (y 2))
  (* x y))
```

So, inside the LET body, X=3 but Y=2. The key point is that the expressions that provide values for the LET binding values are all evaluated /before/ the let bindings happen. In particular, the X that provides a value for Y is the LAMBDA's X, not the LET's X.

Similarly, for the X=5 case, the LET variables have the value X=6, Y=5.

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

Answer: (h i j)

The key point here is to understand that CDDADR is /not/ an abbreviation for "first do CDR then do CDR then do CAR then do CDR," which would give the wrong result (L M N O). Rather, CDDADR means "the CDR of the CDR of the CAR of the CDR," so we get

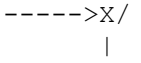
```
(cdr '((a b c d e) (f g h i j) (k l m n o))
       ==> ((f g h i j) (k l m n o))
(car '((f g h i j) (k l m n o)))  ==>  (f g h i j)
(cdr '(f g h i j))  ==>  (g h i j)
(cdr '(g h i j))  ==>  (h i j)
```

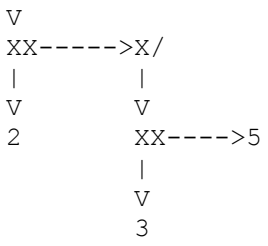
Scoring: One point each.

2. Box and pointers.

```
(list (list 2 (cons 3 5)))
```

Answer: ((2 (3 . 5)))

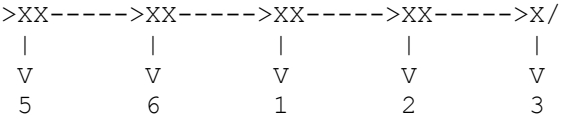




It's easiest to think about this from the inside out. (CONS 3 5) makes the bottommost pair, with car 3 and cdr 5. Then the inner LIST, which has two arguments, therefore makes a list of length 2: the number 2, and the pair that the CONS made. Finally, the outer LIST call has only one argument, so it makes a list of length 1, which is to say, a pair whose car is the element and whose cdr is the empty list.

```
(append '(5 6) (cons 1 '(2 3)))
```

Answer: (5 6 1 2 3)



The CONS sticks one new element, namely 1, in front of the list (2 3), so it returns (1 2 3). So the APPEND is appending two simple lists of numbers, no sublists as elements, so its result is also a simple list of numbers.

Scoring: One point per printed answer, one point per box and pointer diagram.

3. Normal and applicative order.

```
(define (trick x y) (* y y))
(trick (/ 1 0) 5)
```

Normal order: The actual argument /expressions/ are substituted into the procedure body, so the expression (/ 1 0) would be substituted for X, except that there aren't any Xs in the body! So this will never be evaluated, and the answer is just (* 5 5), or 25.

Applicative order: The actual argument /values/ are computed before we call the procedure, so we try to evaluate (/ 1 0) and we get an ERROR.

```
(define (inc x) (+ x 1))
(inc (inc (+ 3 2)))
```

Normal order: In the /outer/ call to INC, the expression (INC (+ 3 2)) is substituted into the body of INC, giving the expression

```
(+ (INC (+ 3 2)) 1)
```

Since + is a primitive, we evaluate its arguments, including the inner call to INC. Since INC /isn't/ a primitive, we substitute the actual argument expression (+ 3 2) into the body, giving

```
(+ (+ 3 2) 1)
```

Those two calls to + are made, and then the outer call to + is made, for a total of 3.

Applicative order: Before we can do the outer INC, we have to evaluate its argument expression (INC (+ 3 2)). Before we can do this inner INC, we have to evaluate /its/ argument expression, so we do the first + call, (+ 2 3) = 5. Substituting 5 for X in the body of INC gives (+ 5 1). This is the second call to +, giving 6. We substitute 6 for X in the body of INC for the outer call, which gives us (+ 6 1) for the third + call.

Therefore, both ways involve three calls to +, so the answer is FALSE.

Normal order can involve more calls than applicative if the same parameter is used twice in the body of a procedure, such as (* X X). But that isn't the situation here.

Scoring: One point each.

4. Order of growth, iterative vs. recursive.

(a)

```
(define (foo n)
  (cond ((= n 1) 1)
        ((even? n) (foo (+ n 1)))
        (else (foo (- n 2)))))
```

Each call to FOO does a bunch of constant-time things (=, EVEN?, + or -) and also makes a recursive call. So this is Theta(N) constant-time calls, which means the whole thing is Theta(N).

You have to be careful about the number of calls. If, for example, the recursive calls had been something like (FOO (/ N 2)), then there would be only Theta(log N) calls. The (FOO (- N 2)), if we followed that branch all the time, would mean a total of N/2 calls (because the argument gets smaller by 2 each time, not just by 1 as in the usual examples), but N/2 is Theta(N). The (FOO (+ N 1)) looks scary, because if that branch is taken all the time, then the argument keeps getting /bigger/, and the procedure never finishes at all! But this can only happen if N is even, and in the next call N will be odd, and it'll stay odd in each of the following calls until finally N=1.

There are two recursive calls to FOO, but only one of them happens each time through the procedure, so this isn't the Theta(2^N) situation. Also, each of those recursive calls is the /entire/ action of a COND clause, so they're both tail calls, and so this generates an ITERATIVE process. (If the COND confuses you, imagine that it's two nested IFs:

```
        (if (= n 1)
            1
            (if (even? n)
                (foo (+ n 1))
                (foo (- n 2))))
```

The inner IF is a tail of the outer IF, and both calls to FOO are tails of the inner IF.

(b)

```
(define (count-to N)
  (if (= N 1) '(1)
      (se (count-to (- N 1)) N)))
```

As the note in the question tells us, each call to COUNT-TO includes a call to SE that takes time Theta(N). There are N of these recursive calls. So the whole thing takes time Theta(N^2).

The recursive call to COUNT-TO is part of the expression

```
        (SE (count-to (- n 1)) N)
```

so it's not a tail call; it's embedded in the call to SE, which happens after the COUNT-TO expression is evaluated to provide its first argument. So this one generates a RECURSIVE process.

Scoring: One point per order of growth, one point per iterative/recursive.

5. Using higher-order procedures.

(a) We are trying to find out whether something is true for every word of a sentence. You could try to construct a list of #T and #F values, but this doesn't help -- you'd still have to ask the question "is every element of the list #T?" which is the same kind of question as "is every word piglatin?"

Instead, use KEEP to extract the words for which the condition is true, then see if you found all of them:

```
(define (is-pig-latin? sent)
  (equal? sent (keep (lambda (wd) (equal? (word (last (bl wd)) (last wd))
                                          'ay))
                     sent)))
```

or

```
(define (is-pig-latin? sent)
  (equal? sent (keep (lambda (wd) (and (equal? (last (bl wd)) 'a)
                                       (equal? (last wd) 'y)))
                     sent)))
```

or use a helper procedure:

```
(define (is-pig-latin? sent)
```

```
  (equal? sent (keep (lambda (wd) (equal? (last2 wd) 'ay))
                     sent)))

(define (last2 wd)
  (word (last (bl wd)) (last wd)))
```

Note that all of these check only for a suffix -ay, as stated in the problem, not checking that the first letter of the word is a vowel, although that's another condition for a valid Pig Latin word. It was fine if you made the additional check.

All of the above also assume that every word in the sentence has at least two letters. This is a flaw, and we preferred solutions that checked:

```
(define (last2 wd)
  (if (>= (count wd) 2)
      (word (last (bl wd)) (last wd))
      'foo))   ; or anything other than 'ay
```

but we didn't take points off for omitting this check.

Many solutions used the unnecessary circumlocution

```
        (if .... #t #f)
```

The thing you're testing in the IF is already true or false! You could use it directly, without calling IF. But this isn't an error, just a stylistic weakness, and we didn't take points off for it.

Scoring:

4	correct
2-3	has the idea
1	has an idea
0	other

"Has the idea" means using higher order functions, not recursion, and trying to answer a yes/no question about every word in the sentence.

Here are some specific common wrong answers and their scores:

Uses EVERY instead of KEEP, or otherwise trying to make a sentence of Booleans: at most 2 points. Remember, sentences can only hold words!

Returns true if /any/ word in the sentence is Pig Latin: at most 2 points.

Recursive solution: 0 points.

"Semi-recursive" solution that uses a higher order function but has a recursive helper: at most 1 point.

(b) Here we are asked to transform each word of a sentence. You might have gotten confused because we're only asked to change /some/ of the words, but that just means that in some cases the transformation function should return

the same word:

```
(define (latin-change sent)
  (every (lambda (wd) (if (equal? (last2 wd) 'ay)
                          (word (bl (bl wd)) 'ey)
                          wd))
         sent))
```

As above, there are other ways to check whether the last two letters are AY.

Scoring: Same as part (a). Having the idea means using higher order functions, not recursion, and transforming the words without eliminating any of them (i.e., no KEEP).

Solutions using KEEP: at most 1 point.

Solutions that change every word, even if not Pig Latin: at most 1 point.

Recursive solutions: 0 points.
Semi-recursive: at most 1 point.

6. Writing recursive procedures.

This problem was much easier if you paid attention to the hint than if you didn't. The thing to do is just use helper functions you want as if you've written them, and then write them later:

```
(define (range sent from to)
  (let ((chop (all-after sent from)))
    (if (empty? chop)
        '()
        (if (member? to chop)
            (all-before chop to)
            '()))))

(define (all-after sent wd)
  (cond ((empty? sent) '())
        ((equal? (first sent) wd) sent)
        (else (all-after (bf sent) wd))))

(define (all-before sent wd)
  (cond ((empty? sent) #f)     ; Can't happen because of MEMBER? test
        ((equal? (first sent) wd) (sentence wd))
        (else (sentence (first sent) (all-before (bf sent) wd)))))
```

Looking through the sentence for the TO word (using MEMBER?) and then, only if it's in there, using ALL-BEFORE to find the desired range means that you don't have to worry about what ALL-BEFORE will return if the word isn't found. (Without that test, it wouldn't work just to return the empty sentence when you reach the end of the input sentence, because all those (SENTENCE ...) calls would still return a nonempty sentence, equal to the original argument!)

The EMPTY? test in ALL-BEFORE isn't really necessary, since we're sure that we'll find the TO word before we run out of words to test.

During the grading I was mortified to learn that I'd written an exam question that's easier to grade if written iteratively! An iterative search for the TO word /is/ able to return an empty sentence correctly if it falls off the end of the sentence without finding the TO word:

```
(define (range sent from to)
  (let ((chop (all-after sent from)))
    (if (empty? chop)
        '()
        (all-before chop to '()))))

(define (all-before sent to result)
  (cond ((empty? sent) '())
        ((equal? (first sent) to) (se result to))
        (else (all-before (bf sent) to (se result (first sent))))))
```

But my mortification is partly assuaged by the fact that this solution is quite inefficient, Theta(N^2) time, because using SE to add a word at the /end/ of a sentence takes Theta(N) time.

Scoring:

8	correct
7	trivial mistake
4-6	has the idea
1-3	has an idea
0	other

"Has the idea" means looking for the FROM word, then looking for the TO word after it, and /trying/ to return the words between them. You didn't have the idea if you thought you could use KEEP to eliminate unwanted words, since what makes a word unwanted isn't the word itself, but its position in the sentence relative to the FROM and TO words, which KEEP can't test.

Wrong answer if no TO word in the sentence, or no TO word after FROM word: at most 6 points.

Finds the /last/ TO word instead of the /first/ TO word after FROM word: at most 6 points.

7. Functions that return functions.

(a) Translating a word to Pig Latin may not have seemed like an iterative improvement, just because all the examples in the book were numeric. But it exactly fits the pattern! A guess is "good enough" if it starts with a vowel. If not, we "improve" it by moving the first letter to the end:

```
(define (piglatin wd)
  (word ((iterative-improve
          (LAMBDA (WD) (MEMBER? (FIRST WD) '(A E I O U)))
          (LAMBDA (WD) (WORD (BF WD) (FIRST WD))) )
         wd)
        'ay))
```

We accepted solutions using helpers we'd written for Pig Latin before, such as (LAMBDA (WD) (VOWEL? (FIRST WD))) or PL-DONE?. Note that PL-DONE? is

already a procedure that takes a word as argument, so it doesn't have to be wrapped in a LAMBDA, but VOWEL? doesn't take the whole word as argument, so you can't just say VOWEL? as the first argument to iterative-improve.

Scoring: One point for each of the two functions.

Many people only said things like (VOWEL? (FIRST WD)), without a surrounding LAMBDA expression. This received no points.

(b) We're going to write something that has the same general structure as iterative-improve, but its two argument functions will take two arguments each, and the count in the second argument will be maintained by count-iterative-improve itself:

```
(define (COUNT-iterative-improve good-enough? improve)
  (define (help x CNT)
    (if (good-enough? x CNT)
        x
        (help (improve x CNT) (+ CNT 1))))
  (LAMBDA (X) (help X 0)))
```

The lower case letters in this definition are taken exactly from the original iterative-improve. The CAPITAL letters are the changes: Two arguments to GOOD-ENOUGH?, two arguments to IMPROVE, and increasing the CNT in the recursive call to HELP. Lastly, we can't just return HELP, because it's a function of two arguments, and we want to return a function of just one argument, the initial guess X.

Scoring:

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

"Has the idea" means that the two arguments are functions of X and CNT, and that you increment CNT rather than expecting the argument functions to do it.

CNT starts at 1 instead of 0: no points off.

Returns HELP instead of (LAMBDA (X) (HELP X 0)): at most 3 points.

Mixes up call to IMPROVE with incrementing count, such as
        (help (improve x (+ cnt 1)))
at most 3 points.

Returns (HELP X 0) instead of (LAMBDA (X) (HELP X 0)): at most 2 points.

If you don't like your grade, first check these solutions. If we graded your paper according to the standards shown here, and you think the standards were wrong, too bad -- we're not going to grade you differently from everyone else. If you think your paper was

not graded according to these standards, bring it to Brian or your TA. We will regrade the entire exam carefully; we may find errors that we missed the first time around. :-)

If you believe our solutions are incorrect, we'll be happy to discuss it, but you're probably wrong!