CS 61A Fall 2005 Midterm #1 solutions

Note: If your individual score is I points (out of 40) and your group score is G points (out of 12), your overall score will be computed as

```
(max I (+ (* 0.91666667 I) (* 0.3333 G)))
```

The theory behind this formula is that 20 of your 40 individual points are not matched by anything in the group exam, and for the 10 points that are matched, we want to weight the exams as 2/3 individual, 1/3 group. So that gives (+ (* 30/40 I) (* 2/3 (* 10/40 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?

Scoring: One point each, all or nothing, except that we deducted at most one point for quotation marks before/around any number of otherwise correct answers. *You* type quotation marks in front of sentences you want Scheme to treat as data rather than as procedure calls, but THE QUOTATION MARKS ARE NOT PART OF THE VALUES, AND SCHEME DOESN'T PRINT THEM.

1)
```
> (keep (lambda (x) (or (even? x) (< (count x) 3)))
        '(1 12 123))
(1 12)
```

(EVEN? X) is true for 12; (< (COUNT X) 3) is true for 1 and 12. So the (OR ...) is true for 1 and 12.

2)
```
> (se '(procedures are) (first 'class))
(procedures are c)
```

(FIRST 'CLASS) is its first letter, C.

3)
```
> (every (* x x) '(4 5 6))
ERROR (unbound symbol x)
```

The answer (16 25 36) would be correct if the question were
```
(every (LAMBDA (X) (* x x)) '(4 5 6))
```
but in the actual question, the first argument to EVERY isn't a function; it's an attempt to compute the square of some particular number called X, and there isn't any such variable.

4)
```
> (every first (keep even? '(23 48 12 87 6)))
(4 1 6)
```

The (KEEP ...) part returns (48 12 6), and so (EVERY ...) returns the first digits of each of those numbers. It's okay that 6 has only one digit; that just means that its last is the same as its first. And it's okay that 1 isn't even; 12 is even, and the KEEP testing happens before we take FIRST of anything.

5)
```
> (word (first '(wish you)) (bf '(were here)))
ERROR (Invalid argument to WORD: (HERE).)
```

(BF '(WERE HERE)) is the sentences (HERE), not the word HERE. That would be (LAST '(WERE HERE)). So this expression gives the WORD procedure a sentence as its second argument.

6)
```
> (cond ('comfortably 'numb) (hey you) (else money))
numb
```

The first COND clause is ('comfortably 'numb). Since anything other than #F is considered true in Scheme, the word COMFORTABLY is true, and so the expression 'NUMB is evaluated to provide the return value for the COND. The remaining clauses are not evaluated, so it doesn't matter than HEY is an unbound variable.

2. Applicative vs. Normal Order

Applicative: 5. Normal: 2. Actual Scheme: 5.

To our surprise, this was the question everyone seemed to find difficult.

In applicative order, FIRST the argument expressions are evaluated, THEN the procedure is called. So the first step is to evaluate

```
(* 2 2)
(* 3 3)
(funky #f (* 4 4) (* 5 5))
```

The first two of these are the first two invocations of *. The third is itself a user-defined procedure call, so its argument expressions are evaluated:

```
#f
(* 4 4)
(* 5 5)
```

The first of these is self-evaluating; the others are invocations three and four of *. Now we substitute actual argument values into the body of FUNKY (for the inner invocation):

```
(if #f 16 (* 25 25))
```

IF is a special form, so its second and third argument expressions are not necessarily evaluated. In this case, the first one is false, so the third one is evaluated; this is the fifth call to *.

Now we're ready to substitute argument values into the body of FUNKY for the outer invocation:

```
(if 4 9 (* 625 625))
```

Since anything other than #F is considered true, 4 is true, and therefore the 9 is evaluated and the (* 625 625) is not evaluated. So the value of the entire expression is 9, and there were five multiplications.

In normal order, FIRST the actual argument *expressions* are substituted into the body, then the result is evaluated. So the *outer* call to FUNKY gives us the expression

```
(if (* 2 2) (* 3 3) (* (funky #f (* 4 4) (* 5 5))
                       (funky #f (* 4 4) (* 5 5))))
```

IF starts by evaluating its first argument; this is the first call to *. The result, 4, is true, and so IF evaluates its second argument expression. This is the second and last multiplication! The answer, 4, is the same value we got with applicative order, but with fewer multiplications.

Actual Scheme uses applicative order! We have no idea how so many people came up with an answer to this part that was different from both of their other answers.

Scoring: One point each.

3. Iterative vs. Recursive Process.

The first procedure generates an iterative process; the second generates a recursive process.

In the first case, even though the recursive call is part of a larger expression, that expression is an IF special form, so (supposing NUM is greater than zero) IF evaluates the recursive call as the last thing it has to do; the result of the recursive call is the result of the IF. (And the inner IF is the result of the outer IF, so the same reasoning applies to that one.)

In the second case, the *second* recursive call is a tail call (the last task to be done), but the *first* one isn't. After the first call returns its value, OR has to decide whether it's true or false, and if false, go on to the second recursive call. So this is a recursive process.

Scoring: Two points each.

4. Invariants

The answer is C: min(m,n).

Consider an example:

invocation	m+n	n-m	min(m,n)	max(m,n)
(mystery 3 6)	9	3	3	6
(mystery 3 5)	8	2	3	5
(mystery 3 4)	7	1	3	4
(mystery 3 3)	6	0	3	3

The only one of these whose value doesn't change is min(m,n).

Scoring: Three points, all or nothing. No credit for more than one answer checked.

5. Order of Growth.

Answers: F, T, T.

Theta(N) and Theta(2N) are the same! Constant factors don't matter in figuring out the order of growth -- both of these are linear functions. During the exam, people kept asking questions like "Can we assume the computers get faster?" It doesn't matter whether computers get faster or don't get faster; even on the same computer, a process that takes time N and a process that takes time 2N are the same order of growth.

Theta(N) is less than Theta(N^2), though. No matter what the constant factors might be, once N is large enough, that difference will outweigh the constant factors.

Similarly, Theta(1), which means that the program takes a fixed amount of time regardless of the input size, is less than Theta(N). No matter how long the fixed amount of time is, for large enough N a linear-time program will take longer than that.

Scoring: One point each.

6. NO-DUPLICATES?

```
(define (no-duplicates? sent)
  (cond ((empty? sent) #t)
        ((member? (first sent) (bf sent)) #f)
        (else (no-duplicates? (bf sent)))))
```

I think this problem is easier to solve with recursion than with higher order functions, even though it does have a KEEP-like flavor, because the criterion for keeping a word involves the other words in the sentence, not just the word itself. But it's possible to do it with higher order functions:

```
(define (no-duplicates? sent)
  (empty? (keep (lambda (wd)
                  (not (empty? (bf (keep (lambda (wd2) (equal? wd2 wd))
                                         sent)))))
                sent)))
```

Some people invented helper procedures to do some of the subtasks, either instead of the inner KEEP call in this version, or because they forgot about the MEMBER? procedure.

In order to write the recursive version above, a crucial idea you need is that you only have to check, for each word, whether it has any duplicates that come *after* it in the sentence, Why? If the word has a duplicate *before* it in the sentence, then this word would have been caught as a duplicate of that other word, when we checked the other word, which we do first. As soon as any duplicate is found, we can stop checking. (That's another advantage to the recursive version; using KEEP will check all of the words even after a duplicate for one of them has been found.)

Most people got this right. One common small error was to use a one-word

sentence as the base case, instead of the empty sentence -- the spec says "takes a sentence as its argument," not "takes a nonempty sentence." Some people also got true and false backwards. We took off one point for each of these.

Scoring:

6 correct
5 trivial error, see above
4 has the idea
2 has an idea
0 other

7. MAKE-CUSTOMIZED-EVERY.

The tricky thing here was to keep track of the domains and ranges of the various procedures involved: MAKE-CUSTOMIZED-EVERY itself, its argument PRED, its return value (which I'll call CUSTOM in these notes), and CUSTOM's argument procedure (which I'll call FN).

Once you have these figured out, the actual code is simple:

```
(define (make-customized-every pred)
  (lambda (fn sent)
    (every (lambda (wd) (if (pred wd) (fn wd) wd))
           sent)))
```

That's the most elegant solution. You can also write it recursively, which in effect means rewriting EVERY, but then you need a way to refer to CUSTOM, which doesn't have a name in the solution above (it's the procedure made by the lambda expression), within the recursion. One solution is an internal definition:

```
(define (make-customized-every pred)
  (define (custom fn sent)
    (cond ((empty? sent) '())
          ((pred (first sent))
           (se (fn (first sent)) (custom fn (bf sent))))
          (else (se (first sent) (custom fn (bf sent))))))
  custom)
```

A common error in this version was to forget the last line, the one that actually returns CUSTOM. The other way was to recreate CUSTOM each time:

```
(define (make-customized-every pred)
  (lambda (fn sent)
    (cond ((empty? sent) '())
          ((pred (first sent))
           (se (fn (first sent))
               ((make-customized-every pred) fn (bf sent))))
          (else (se (first sent)
                    ((make-customized-every pred) fn (bf sent)))))))
```

This is perfectly correct (even though it seems to make a recursive call with the same argument that it's given, so you might have to think about why this isn't an infinite recursion), but a little inefficient because of the repeated calls to make-customized-every.

The most common error, a serious one, was to return a procedure that doesn't take FN and SENT (or equivalents, regardless of name) as arguments, or not to return a procedure at all. We gave such solutions at most three points.

Scoring:
7 correct
6 trivial error
4-5 has the idea
2-3 has an idea
0 other

8. POLY

If you paid attention to the hint, you ended up with this elegant solution:

```
(define (poly coefs)
  (lambda (x)
    (if (empty? coefs)
        0
        (+ (* x ((poly (butlast coefs)) x))
           (last coefs)))))
```

or this slightly faster variant, moving the lambda inside the if (twice):

```
(define (poly coefs)
  (if (empty? coefs)
      (lambda (x) 0)
      (lambda (x)
        (+ (* x ((poly (butlast coefs)) x))
           (last coefs)))))
```

(It's faster because the IF test only happens once, when you create the polynomial procedure, rather than each time it's called.)

In this problem, unlike question 6, it was okay to use a one-term (constant) polynomial, rather than an empty polynomial, as the base case, since the spec says that the argument will be "a sentence of one or more numbers."

It's also possible, although uglier and less efficient, to solve this problem by computing powers of x:

```
(define (poly coefs)
  (lambda (x)
    (if (empty? coefs)
        0
        (+ (* (first coefs) (expt x (- (count coefs) 1)))
           ((poly (butfirst coefs)) x)))))
```

EXPT is a Scheme primitive, but many people defined their own, which is also okay provided you get it right.

Note that in computing the value of a smaller polynomial, you have to take the result of the recursive call to POLY, which is a function, and call that function with X as its argument. That's why all of these solutions say

```
        ((poly <something) x)
```

Thinking that the recursive call to POLY gives a number lost two points.

Returning 1 instead of 0 for an empty polynomial lost one point.

Using the coefficients in the wrong order lost two points.

Defining an internal helper but never calling it (so your outer procedure has no body other than the internal definition) lost two points.

Multiplying all the coefficients by X (instead of by a power of X) lost two points. (Usually this wasn't intentional, but the result of an error in making recursive calls.)

Writing *external* (global) helper procedures that use X or COEFS without taking them as arguments lost three points.

No lambda (not returning a procedure) lost four points.

The very worst error, all too common, was to write a procedure that works only for polynomials with four (or up to four) coefficients. Never write a program that works only for the example(s) shown in the problem! These solutions got zero points.

Scoring: Same formula as question 7; see above for particular cases.

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!