```
CS 61A          Fall 1998       Midterm #1 solutions

0.  Name, etc.

Some people did not read and/or sign the box on the front that said
READ AND SIGN THIS in big bold letters.  Other than that, I was pleased
that this semester's students did better than usual at knowing which
sections you're in.

1.  What will Scheme print?

> (3 + 5)
ERROR

        Almost everyone remembered that in Scheme the procedure comes first,
        not between the arguments.  This expression is attempting to call
        the number 3 as if it were a procedure.

> (butfirst (first (butfirst '(i want to tell you))))
ANT

        (bf '(i want to tell you))    --> (want to tell you)
        (first '(want to tell you))   --> want
        (bf 'want)                    --> ant

        One common wrong answer was (ANT) -- a sentence containing
        the word ANT.  Words and sentences are two different kinds of
        data.

        Another wrong answer was 'ANT -- the correct word with a
        quotation mark in front of it.  I think you think that the
        quotation mark means "this is a word" or some such thing;
        but it really means "don't evaluate this expression."  You
        type quotation marks in to Scheme, not the other way around.

        Finally, many people said () -- the empty sentence.  I'm not
        sure why; perhaps they thought that
                (first '(want to tell you)) --> (want)   [WRONG]
        for which the butfirst would indeed be empty.

> ((lambda (a b c) (b a c)) 3 + 5)
8

        Several people thought this was an error.  Perhaps they were
        confused because the body of the procedure, (B A C), looks
        like the formal parameter list, (A B C), and so they thought
        that 3 + 5 was the body.  But actually 3, +, and 5 are the
        actual argument expressions corresponding to the formal
        parameters A, B, and C respectively.  So, substituting the
        arguments for the parameters in the body (B A C) gives (+ 3 5)
        and the value of that expression is 8.

        A few people said this returned a procedure.  But the lambda
        expression is enclosed within a larger expression -- there are
        two sets of parentheses around it -- and so the procedure that
        lambda creates is then invoked, not returned.

> (let ((a +)
        (b (a 3 4)))
    (a b b))
ERROR

        The main point here is that the expression (a 3 4), which is
        supposed to provide a value for B, is evaluated *outside* the
        scope of the LET, so that A is *not* bound to the + procedure
        in that scope.  The error is that A is unbound during the
        attempt to evaluate (a 3 4).

        The most common wrong answer was 14, which is what would be the
        result if B were bound to 7 during the evaluation of (A B B).
        This answer *would* be correct if the LET were a LET* instead.
        Then the expression (a 3 4) would be evaluated with A bound
        to the + procedure.

> (cond ((<4 7) 5)
        ((> 4 2) 6)
        (else 3))
5

  <4 7) is true, so COND returns 5 and never looks at the other
        clauses at all.  Most people got this, but one interesting wrong
        answer was
                        5 6
        as if COND could return two values.  (I'm guessing this person
        thinks that COND *prints* things, in which case it could
        certainly print more than one thing -- but there's no printing
        in functional programming.  Every expression returns exactly one
        value.

> (se ('tell 'me 'why))
ERROR

        The error is that the inner parenthesized expression tries to invoke
        the word TELL as if it were a procedure.  People who said anything
        other than error probably forgot that parentheses in Scheme always
        mean to invoke the first thing inside them as a procedure, unlike
        the math notation f(x) in which the procedure (or function) is given
```

        outside of the parentheses.

Scoring: 1/2 point each, rounded down.

2. Applicative and normal order.

The "2 points" was a typo; this question was worth one point, as shown on the grid on the front cover of the exam.

The expression (* (counter) (counter)) has the value 2 under both applicative and normal order.

Under applicative order, all subexpressions are evaluated before * is called. There are two subexpressions of the form (counter), each of which is evaluated separately. The first evaluation returns 1; the second returns 2. (It doesn't matter whether they are evaluated left to right or right to left, since 1 times 2 = 2 times 1.)

Under normal order, you might think that things would get more complicated, but since * is a primitive (as the problem statement says), its arguments are evaluated before it is invoked even under normal order. It's only arguments to user-defined procedures that are handled differently.

But even if we did something like this:

```
        (define (times a b)
          (* a b))

        (times (counter) (counter))
```

it wouldn't change the result. Normal order *would* affect the call to TIMES, so that instead of substituting 1 and 2 for A and B, we'd substitute the expression (COUNTER) for both A and B in the body of TIMES. The result of that substitution would be

```
        (* (counter) (counter))
```

which is the same thing we started with.

The most common error was to say that the answer would be 1 under applicative order. People who said that were confusing this problem with a different one, more like what I did in lecture:

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

        (square (counter))
```

For *that* problem, the answer would be 1 under applicative order and 2 under normal order. But this is the opposite of the situation you were given.

Normal order can turn one subexpression into multiple copies (which are then separately evaluated), but it can't turn two expressions, even if identical, into a single expression to be evaluated once.

Interestingly, almost as many people said that the answer would be 1 under *normal* order. My guess is that they were thinking of the square problem, but instead of actually working through that problem, they just remembered that normal order is the one that gives the weird answer. :-)

Scoring: 1 point, all or nothing.

3. Log base two.

Since this was the problem on the entrance test, and since I solved it in lecture in three different programming languages including Scheme, everyone should have gotten this perfectly correct! Alas, that didn't happen.

(a) recursive process

```
(define (lg n)
  (if (= n 1)
      0
      (+ (lg (floor (/ n 2))) 1)))
```

Please notice that this is *exactly* what the problem statement told you to write. It told you that lg(1) is 0. And for other values, it told you that lg(n) = lg(floor(n/2))+1. Many people asked during the exam "what does floor mean," but *you don't have to know* what floor means! If you just follow instructions, and take lg(floor(n/2))+1, your program will be correct. (But you should know anyway: floor(x) is the largest integer less than or equal to x.)

(b) iterative process

I didn't do this in lecture, so you had a bit of an excuse for not getting it. Here's what we wanted:

```
(define (lg n)
  (define (helper n result)
    (if (= n 1)
        result
        (helper (floor (/ n 2)) (+ result 1))))
  (helper n 0))
```

Again, we asked you to use the algorithm described in the problem statement, not invent some other algorithm.

Scoring: Each half was worth two points, assigned as follows:

```
        2:  correct AND has the desired structure
        1:  correct OR has the desired structure
        0:  neither
```

"Correct" means that your program gives the right answer and generates a recursive or iterative process, as appropriate. "Has the desired structure" means that you implemented (perhaps wrongly) the algorithm you were told to implement.

There weren't any interesting wrong answers to part (a). For part (b), several people tried to write the following iterative program, which uses a different algorithm from the one we told you:

```
(define (lg n)                 ;; NOT TO SPEC
  (define (helper product count)
    (if (> product n)
        (- count 1)
        (helper (* product 2) (+ count 1))))
  (helper 1 0))
```

We gave this one point, although in fact everyone who tried it got it wrong because they didn't work out the need to go one extra iteration to handle both exact powers of 2 and numbers that aren't powers of 2.

The most common one-point answer for part (b) was to write the helper procedure but not write a wrapper procedure that invokes it. Some people called their helper procedure LG and wrote little comments about what values they wanted the user to give for the extra arguments, but that's not following the spec -- LG takes one argument.

A fairly common one-point mistake was to get the iterative program right except for the base case, returning 0 instead of RESULT. Common in both (a) and (b) was to leave out FLOOR altogether, for one point.

The most common zero-point solution for (b) was to call LG inside the call to helper, something like this:

```
        (helper (lg (floor (/ n 2))) (+ result 1))
```

The resulting process is not only recursive but also Theta(N^2).

A less common zero-point solution was an iterative solution that takes N iterative steps instead of lg(N) iterative steps, thereby getting the wrong answer.

A general comment about grading programming questions: We ignored missing or extra close parentheses, but did pay attention to missing or extra open parentheses, since they might indicate that you didn't know whether or not to invoke a procedure.

```
4.  constant-fn? and make-constant-checker

Part (a):

(define (constant-fn? fn sent)
  (cond ((empty? sent) #t)      ; we didn't penalize for omitting this
        ((empty? (bf sent)) #t)
        ((equal? (fn (first sent))
                 (fn (first (bf sent))))
         (constant-fn? fn (bf sent)))
        (else #f)))

Scoring:  This part was worth three points, as follows:

        3: correct
        2: has the idea (must have domain and range correct)
        1: has an idea
        0: other (including any solution with specific cases built in
                        such as SQRT or (lambda (x) 4) etc)

This is a straightforward higher-order procedure, and everyone should have
gotten it.  Most people did pretty well, with two common minor errors.

One common error was to check for (empty? sent) as the base case, but not
check for (empty? (bf sent)).  Really you should check for both, as above,
because the specification for the program says that the second argument
is "a sentence of numbers," not a *nonempty* sentence of numbers.  But
leaving out the second test is particularly bad, because it means that
the program will blow up for any argument sentence.  Since the third
clause asks for (first (bf sent)), the non-base case requires at least
two numbers in the sentence.

[A clever solution that a few people used to avoid this problem was to
say (equal? (fn (first sent)) (fn (last sent))).  Since even a one-word
sentence has both a first word and a last word, these people could
check (empty? sent) as their base case.]

The second error was to misunderstand the domains of some primitives.
The most common version was to call the = primitive instead of EQUAL?.
But = works only if its arguments are numbers.  Of the four examples
shown in the problem statement, the first two use functions that return
numbers, but the last two use a function that returns #T or #F.
A related problem was that some people tried to write the program in
two pieces, one of which applies FN to each number in SENT, and the other
of which checks whether the results are all equal.  But they mostly tried
to do this using SENTENCE to combine the results, or equivalently, using
the higher-order function EVERY, which itself uses SENTENCE.  But the
domain of SENTENCE is words and sentences, not, for example, #T or #F.

Those programs had the idea.  The most common "has an idea" solution was
```

to try to make this an EVERY-style recursion, sort of like this:

```
        (equal? (fn (first sent))
                (constant-fn? fn (bf sent)))    ; WRONG
```

Some people wrote solutions using CONS, CAR, CDR, MAP, etc. As you know
by now, these are data abstraction violations, since you were told to
accept "a sentence of numbers" as argument, not a list of numbers. But
we didn't take off for it.

For part (b) there was really only one good solution:

```
(define (make-constant-checker fn)
  (lambda (sent) (constant-fn? fn sent)))
```

I'm pleased that most students did, in fact, find this solution. Many
others had correct solutions that involved rewriting constant-fn?, but
that misses one of the key ideas of computer science, code reuse. Or,
in the words of General Giap, "Use what you have to get what you need."

It may be worth showing a solution that effectively rewrites constant-fn?:

```
(define (make-constant-checker fn)
  (lambda (sent)
    (cond ((empty? sent) #t)
          ((empty? (bf sent)) #t)
          ((equal? (fn (first sent))
                   (fn (first (bf sent))))
           ((make-constant-checker fn) (bf sent)))      ;; see below
          (else #f))))
```

The interesting thing in this solution is the next to the last line. The
two open parentheses are *not* the typical beginning of a COND clause;
they are both regular procedure-calling parentheses. You use a recursive
call to make-constant-checker, with the same argument, to get a function
that you then apply to (bf sent). It's worthwhile thinking about this case
in connection with the general rule that a problem must get smaller in
a recursive call. Something is getting smaller, but not exactly in the
recursive call!

Scoring: Same three-point scale as part (a), except that we didn't take
off points again for repeating the same mistake in (b) that you made in (a).

5. Degree of a polynomial.

I really like this problem. It's an example of constructing a program that
does something significant out of small, easy pieces.

You didn't really have to understand the mathematics to solve the problem,
but you *should* understand the math -- you should have learned it in high

school, and if not, you should learn it now!

Suppose you've collected some data in an experiment, and you'd like to find a formula that fits the data. If you have N values, you can always find a polynomial function of degree at most N that fits exactly. In this program, we start with the values of f(x) corresponding to x=1, x=2, etc. We want to find the simplest polynomial function that will fit, and the first step is to find the degree of that polynomial -- the highest power of x that's needed. If all the values are the same, as in a case like (5 5 5 5 5), then the degree is zero; the function is the constant polynomial f(x)=5. If the *differences* between values is constant, as in (5 7 9 11 13), then the polynomial is linear, in this case f(x)=2x+3. If the differences between differences is constant, then the function is quadratic. And so on. Taking differences between equally-spaced values is sort of like taking the derivative of the function; that's why this works.

(a) all-equal?

The really cool solution to this problem is

```
(define all-equal? (make-constant-checker (lambda (x) x)))
```

Please read that carefully if it's not what you did. Make-constant-checker returns a function. We want all-equal? to be such a function. There's no need to say

```
(define (all-equal? nums)
  ((make-constant-checker (lambda (x) x)) nums))
```

which was the more common solution. Train yourself to think like the first version!

The function used as argument to make-constant-checker doesn't have to be the identity function; it can be any one-to-one function. But it can't be =, which is a function of two arguments.

Scoring: One point for either correct solution. The problem requires that you use make-constant-checker in the implementation!

(b) differences

This is a straightforward recursion.

```
(define (differences nums)
  (if (empty? (bf nums))
      '()
      (se (- (first (bf nums)) (first nums))
          (differences (bf nums)))))
```

We didn't take off for getting the arguments to - backwards, even though it's wrong, because in this application it doesn't matter. But it *does* matter if you take the absolute value of the differences; that will lead to getting the degree of some polynomials wrong. There is no linear function that fits the data (1 2 1 2 1 2)!

We also didn't take off for making the base case a sentence of length 2, again because a sentence of length 1 won't come up in this context. But you really should believe in empty sentences as legitimate return values.

Scoring: 2 if correct, 1 if it has some merit, 0 otherwise.

(c) degree

This is the fun part:

```
(define (degree nums)
  (if (all-equal? nums)
      0
      (+ 1 (degree (differences nums)))))
```

Most people who wrote a recursive-process solution got it right. Many people tried to write an iterative-process solution; some succeeded, but a lot didn't. This is a great example of a problem for which your life will be much easier if you learn to love recursion!

Scoring: Same as (b). A correct solution must use the procedures from parts (a) and (b).