CS 61A   Spring, 2009   Final exam solutions

1. Applicative/normal order.

```
(define (if-false x y)
  (if x 'true y))

(if-false (/ 33 0) (/ 33 1))
```

In APPLICATIVE order, both of the actual argument expressions are evaluated
before the function is called, so the expression (/ 33 0) is evaluated and
gives an ERROR.

In NORMAL order, the argument expressions are not evaluated, but are
substituted into the body:

```
        (if (/ 33 0) 'true (/ 33 1))
```

The IF expression has to evaluate its first argument in order to know which
of the other two it should choose. So (/ 33 0) is still evaluated, and we
still get an ERROR, although it comes a little later in the process.

```
(if-false (/ 33 1) (/ 33 0))
```

In APPLICATIVE order, both argument expressions are evaluated right away, as
in the first example, so we again get an ERROR.

In NORMAL order, the argument expressions are substituted into the body:

```
        (if (/ 33 1) 'true (/ 33 0))
```

The first argument to IF is evaluated; its value is 33, which is considered
true in Scheme, so IF evaluates its second argument and returns the word TRUE.
(No quotation mark!) The expression (/ 33 0) is never evaluated, so there is
no error.

Scoring: One point per answer.

2. Higher order functions.

In order for this to work even with the possibility of negative numbers,
you can't use a fixed base case for the accumulation with MAX. Instead you

have to find the first number in the list, if any, and use that as the
base
case for the accumulation:

```
(define (maxnum lst)
  (let ((nums (filter number? lst)))
    (if (null? nums)
        #f
        (accumulate max (car nums) (cdr nums)))))
```

The four-point positive-number-only solution would be this:

```
(define (maxnum lst)   ; not perfect but at least elegant :-)
  (accumulate max 0 (filter number? lst)))
```

Another correct solution avoids the problem of the base case for
ACCUMULATE
by using APPLY instead. This works only if you first ensure that there
is
at least one number; you can't apply MAX to no arguments:

```
(define (maxnum lst)
  (let ((nums (filter number? lst)))
    (if (null? nums)
        #f
        (apply max nums))))
```

Why do we need APPLY here? Why not just (MAX NUMS)? The MAX primitive
does
not take a list as argument. It takes one or more /numbers/ as
argument(s).

Many people misunderstood what the problem is about negative numbers.
It's
not that negative numbers are excluded from the domain of MAX! People
who
took pains to reinvent their own MAX didn't benefit from doing so
(although
it didn't hurt either). The problem is mathematical; there is no
identity
element for the MAX function, unlike + (identity 0) and * (identity 1).
This
is still true even if you write it yourself. Similarly, explicitly
looking
for a minus sign as the FIRST of a number is unnecessary and unhelpful.

Because the argument is a list that might include sublists, and not a
sentence, it would be a data abstraction violation to use KEEP, FIRST,
etc.

Some people, instead of using FILTER, said

```
        (map (lambda (elt) (if (number? elt) elt '())) lst)
```

and then hoped that the result would be the empty list if there are no
numbers. Alas, what you get in that case is a list full of empty
lists, like

this: (() () () ()). What you wanted was EVERY, except that that's a data
abstraction violation. You could use FLATMAP with a little extra work:

```
        (flatmap (lambda (elt) (if (number? elt) (LIST elt) '())) lst)
```

But why not just use FILTER? It's the tool designed for this purpose.

Many people, because the question explicitly mentioned lists as elements,
thought it was necessary to say

```
        (filter number? (filter atom? lst))
```

or equivalent things. The type predicates, such as NUMBER?, take any value
as an argument; they just return #F if the type isn't what they want.

You never have to say (lambda (x) (f x)), e.g.,

```
        (filter (lambda (elt) (number? elt)) lst)
```

This works, but get out of the habit.

Scoring:

```
5  correct.
4  positive numbers only.
3  data abstraction violation.
2  dies if no numbers in list.
2  says (max a-list).
1  close but computes the wrong function.
0  not even close.
0  uses recursion.
```

3. Orders of growth.

```
(define (common? ls1 ls2)
  (cond ((null? ls1) #f)
        ((null? ls2) #f)
        ((equal? (car ls1) (car ls2)) #t)
        (else (or (common? ls1 (cdr ls2))
                  (common? (cdr ls1) ls2)))))
```

Each call to COMMON? gives rise to two recursive calls, similar to
COUNT-CHANGE or PASCAL, and so this is Theta(2^n) time.

Since there are two recursive calls (both of which are done each time), at
least one of them must be a non-tail call, so this generates a recursive
process. (Actually the situation is a little more complicated because of the
way OR works as a special form. If the first call to COMMON? returns true,

then the second call doesn't happen. Still, OR doesn't /know/ that the first
call is going to return true until it returns, so OR still has to check the
result. And if the first one returns false, then both are evaluated.)

```
(define (common? ls1 ls2)
  (cond ((null? ls1) #f)
        ((member (car ls1) ls2) #t)
        (else (common? (cdr ls1) ls2))))
```

This does one recursive call for each element of the list, so there are
a total of N recursive calls. However, each of those calls involves a call
to MEMBER, which is itself Theta(N) time, so this version of COMMON? is
Theta(N^2) time.

(Some people asked questions during the exam having to do with the difference
between MEMBER and MEMBER?, thinking that the latter would take Theta(N) time
because it's written in Scheme, whereas the former is a real Scheme primitive
and must therefore be constant time. But MEMBER is also Theta(N); just
because it's a primitive doesn't give it a magic way to check the N elements
of the list all at once!)

The recursive call to COMMON? is the last thing the procedure has to do; it
provides the answer directly, with no adding 1 or anything like that, and so
this procedure generates an iterative process.

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

4. Streams.

```
(define my-stream
  (cons-stream 3 (cons-stream 4 (add-streams my-stream
                                             (stream-cdr my-stream)))))
```

The first two elements are obviously 3 and 4. After that we have to add
MY-STREAM and (STREAM-CDR MY-STREAM), so let's look at those:

```
  3     4    _____ _____ _____
  4   _____ _____ _____ _____
```

We can do the leftmost addition; the result is 7, so now we have

```
  3     4     7    _____ _____
  4     7    _____ _____ _____
```

Now we can do the next addition and get 11:

```
  3     4     7    11    _____
  4     7    11   _____ _____
```

The third sum is thus 18, and that gives us the five desired elements:

```
  3     4     7    11    18
```

Scoring: Two points, all or nothing.

5. Tree recursion.

The value of a new datum depends on the results higher up in the tree, so we're going to need a helper procedure that remembers the sum so far.

```
(define (sumpath tree)
  (define (help tree sum)
    (make-tree (+ (datum tree) sum)
               (map (lambda (child) (help child (+ (datum tree) sum)))
                    (children tree))))
  (help tree 0))
```

You can avoid having to add twice either by using a LET:

```
(define (sumpath tree)
  (define (help tree sum)
    (let ((newsum (+ (datum tree) sum)))
      (make-tree newsum
                 (map (lambda (child) (help child newsum))
                      (children tree)))))
  (help tree 0))
```

or, less obviously, by starting the sum with the datum of the root node:

```
(define (sumpath tree)
  (define (help tree sum)
    (make-tree sum
               (map (lambda (child) (help child (+ (datum child) sum)))
                    (children tree))))
  (help tree (datum tree)))
```

The LET solution is most efficient, since the addition inside the MAP call
is actually performed once per child, so that's the one you want to avoid.
But in any case it's just a constant factor improvement.

Scoring:

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

6. Assignment and environments.

I was really, really disappointed at how few people got full credit on what
was supposed to be an easy problem. (This was partly because most people drew
box and pointer diagrams, whereas what would have helped was an environment
diagram!) So, if you got this right, I especially want you as a lab assistant
in the fall, so next semester's students will learn this.

```
(define (swap ls1 ls2)
  (let ((temp ls1))
    (set! ls1 ls2)
    (set! ls2 temp) ) )

STk> (define foo (list 1 2 3))
foo
STk> (define bar (list 'a 'b 'c))
bar
STk> (swap foo bar)
okay
STk> foo
(1 2 3)        ; first answer
STk> bar
(A B C)        ; second answer
```

This is the part that most people got wrong.

Although the intent of the SWAP procedure is clearly to exchange the values
of the variables used as its arguments, it doesn't actually do that. The
two SET! expressions in the procedure exchange the values of the procedure's
local variables LS1 and LS2, but they don't affect any other variables.

If you're not convinced, imagine calling SWAP with quoted arguments:

```
        (swap '(1 2 3) '(a b c))
```

This would run with no errors, but it can hardly change the values of the
constant argument expressions!

```
STk> (define x 1)
STk> (define y x)
STk> (set! x 2)
STk> y
1              ; third answer
```

This is a very simple example of something that some groups seemed to
misunderstand on the environment question of midterm 3. DEFINE binds its
first argument (the variable name) to /the value of/ its second argument.

The value is computed when you do the define; afterward, the variable
"knows" just that value, not the expression from which it was computed.
So it doesn't matter if we change the value of X; that can't affect Y.

Scoring: 2 point each half, all or nothing. (It's not one point for each
blank in the first part because thinking that the two variables end up
equal, which is how people got one of the blanks right, is if anything even
worse than thinking the two values would be swapped.)

7. OOP.

(a) The CONTACT class.

```
(define-class (contact name address phone))
```

That's it! That's the entire definition. Some people added redundant
method definitions, e.g.,

```
        (method (name) name)   ; not needed!
```

but our OOP system provides these methods automatically. We didn't take off
points for it, though, especially since some other OOP languages don't give
you these "getter" methods unless you ask for them.

We did take off, though, for thinking that CELLPHONE should be a parent class
of CONTACT. Is a contact a kind of cellphone? Can a contact make phone
calls? No!

And we took off for

```
        (method (name) (ask self 'name))      ; WRONG!
```

which is an infinite loop!

(b) The SMARTPHONE class.

```
(define-class (smartphone)
  (parent (cellphone))
  (instance-vars (contacts '()))
  (method (add-contact name addr phone)
    (set! contacts
          (cons (instantiate contact name addr phone)
                contacts)))
  (method (call name)
    (let ((found (filter (lambda (c) (equal? (ask c 'name) name))
                         contacts)))
      (if (null? found)
          "Name not found"
          (ask self 'dial (ask (car found) 'phone))))))
```

```
                ; or (USUAL 'DIAL ...)
```

A Smartphone /is/ a kind of cellphone, so this is where the PARENT goes.
Some people left out the PARENT, and instead they said (instantiate cellphone)
every time the CALL method is used. But that's not how it works! Your
smartphone doesn't run off to the cellphone factory to make a new cellphone
every time you want to make a phone call; it has a cell phone chip inside it.
We took off the PARENT point for these solutions but didn't take off
additional points in the CALL method.

At least one student made a cellphone (in an INITIALIZE clause) and put it
in an instance variable, then said (ASK MY-PHONE 'DIAL ...) instead of
(ASK SELF 'DIAL ...). We rejected this solution as not taking advantage of
the OOP paradigm. Although it allows the CALL method to work, this solution
doesn't let you send a DIAL message to the smartphone, and we'd like that to
work also.

Many students confused classes and instances, e.g.,

```
        (parent (dumb-phone))
```
or
```
        (ask smartphone 'dial ...)
```

This is a pretty significant failure to understand the OOP paradigm.

Several people said

```
  (method (add-contact name addr phone)        ; WRONG!
    (instantiate contact name addr phone)
    (set! contacts (cons contact contacts)))
```

Instantiate /returns a value/, the new instance. It doesn't magically
bind the class name to the new instance!

Other students made a list of names instead of a list of contacts. This
makes searching for a name easier, but it makes it impossible for the CALL
method to find the actual phone number!

There are other ways to accomplish what the FILTER does, and you can do a
better job by checking for more than one contact with the given name, but on
61A exams you don't have to check for errors unless specifically told to.

Scoring:

1 point for part (a).

```
1 point for PARENT clause.
1 point for INSTANCE-VARS clause.  (No point for CLASS-VARS.)
2 points for ADD-CONTACT method. There are three parts to this:
    - the SET! CONTACTS
    - (CONS contact CONTACTS)  [not (CONS name CONTACTS)!]
    - the INSTANTIATE
  We gave 1 point for any two of those three.
2 points for CALL method.  There are three parts to this, too:
    - ASK SELF 'DIAL
    - ASK contact 'PHONE       [not (ASK name 'PHONE)!]
    - a correct FILTER
  We gave 1 point for any two of those three.
```

8. Mutation.

On your exam papers, draw actual boxes, not the ASCII makeshifts we use in the solutions!

```
(let ((x (list 'a 'b 'c)))
 (set-cdr! (cdr x) 3)
 x)

(A B . 3)

x ---> XX ---> XX ---> 3
       |       | .
       V       V  .
       A       B   .>X/
                     |
                     V
                     C
```

The dotted line shows the previous (cddr x) before mutation.

This tests basic mutation, and also the printed representation of an improper list. A common wrong answer is (a (b . 3)); another is (a b 3).
People who said (a (b . 3)) mostly had correct diagrams; people who said (a b 3) mostly had diagrams agreeing with that mistake. The problem seems
to be confusing, e.g., (cdr x) [a pair] and (cadr x) [the symbol B].
There's no such thing as a pointer to half of a pair; it points to the whole
pair, or it points to the same thing the arrow from one half points to.

```
(let ((x (list 'a 'b 'c)))
 (set-car! x (caddr x))
 (set-car! (cddr x) (car x))
 x)

(C B C)

       A
       ^
       :
```

```
x ---> XX ---> XX ---> X/
       |       |       |
       V       V       V
       C       B       C
```

The dotted line shows the previous (car x).

This tests that the expression (car x) in the second SET-CAR! refers to
the new value, not the original value. Some people, again confusing
(caddr x) with (cddr x), thought that this generated an infinite loop,
with the third pair pointing to the [left half of the] first pair.
Other people had the right print form, but had the new arrow in the
diagram
pointing to the third pair instead of to C.

```
(let ((x (list 'a 'b 'c)))
 (set! (car x) (caddr x))
 x)
```

ERROR

This tests that SET! is a special form whose first argument must be a
symbol
(the name of a variable).

If you drew a diagram along with saying ERROR, we ignored it; if you
said
the print form was (A B C) you didn't get any points, even though that's
the value of X, because it isn't the value of the expression, which
doesn't
have a value.

Some people thought that the arguments to SET! were both evaluated, and
that
the expression was therefore effectively (set! a c). This is completely
wrong; the first argument must be a symbol, not an expression whose
value
is a symbol. People who wrote this and also said ERROR did get credit,
because it wouldn't be fair to take the points away, given that some of
the
people who just said ERROR without an explanation thought the same
thing.
But people who didn't say ERROR didn't get credit.

By the way, the error message is not "unbound variable" as many people
suggested; "(CAR X)" is not a variable [a symbol] at all, bound or
unbound.

Scoring: One point per print form, one per diagram, except two points
for
the ERROR in the third expression.

9. Concurrency.

```
(define x 100)
(define y 10)
(define s (make-serializer))
(define t (make-serializer))
(parallel-execute (s (lambda () (set! x (+ x y))))
                  (t (lambda () (set! y (* x y)))))
(list x y)
```

The correct answers are the ones you'd get if the two SET! expressions
were evaluated in either sequential order. If the first one is done
first,
then the answer is (110 1100) because Y is (* 110 10). If the second
one
is done first, then the answer is (1100 1000) because X is (+ 100 1000).

Since the two threads use two different serializers, they are not
protected
against each other, so incorrect results are possible. In this case,
the only
reason for an incorrect result is that both threads read X and Y before
either
thread changes its variable. In that case we get the answer (110 1000).
A common wrong answer was "none," which was my first guess too because
the
two threads modify different variables, but in fact once a thread has
examined
the values of variables it has to finish its work in order to avoid
potential
problems, even if there is no conflict about modifying the same
variable.

(To my surprise, several people got these two answers reversed, marking
the
two correct answers as incorrect, and the incorrect answer as correct,
losing
two points. "Correct" in this context means "consistent with some
sequential
evaluation." See, for example, the second "note" in section 4.1.3 of
the
Scheme standard, page 79 of reader vol. 2.)

No deadlock is possible, since each thread uses only one serializer.

```
(define x 100)
(define y 10)
(define s (make-serializer))
(define t (make-serializer))
(parallel-execute (s (t (lambda () (set! x (+ x y)))))
                  (s (t (lambda () (set! x (* x y)))))) ; x this time!
(list x y)
```

Correct answers: If the first SET! is done first, it sets X to 110, and
then the second SET! sets it to 1100, so the answer is (1100 10). If
the
second SET! is done first, it sets X to 1000, and then the first SET!
sets

it to 1010, so the answer is (1010 10).

Incorrect answers: None are possible, because both threads use the same serializers. (Using two of them is redundant; even though two variables are
involved, one serializer would protect the threads against each other, and
there are no other threads to worry about.) We accepted solutions in which
people listed answers that would have been possible without serializers but
also said explicitly that these wouldn't occur because serializers are used.

No deadlock is possible, even though two serializers are used, because
both threads use them in the same order.

Scoring: One point for each pair of correct answers, one point for each set of
incorrect answers, one point for each deadlock. Exception: If you listed all
three possible answers as correct in the first problem, and no other answers,
you lose only one point, not two. Extra answers lose the point.

10. Analyzing evaluator.

```
(define (mystery n)
  (if (< n 1)
      1
      (* n (mystery (/ n 2)))))

> (mystery 100)
```

Analyzing will be FASTER. Since the procedure is called more than once,
it benefits from doing the analysis in advance.

```
(define (f x)
  (* (+ x 30) 7))

(define (g x)
  (* x x))

(define (h x)
  (/ (+ (* x 2) 10) 2))

(define (fgh n)
  (f (g (h n))))

> (fgh 100)
```

Analyzing will be THE SAME SPEED. There are lots of procedure calls, but
only one to each procedure, so there's no repetition of the analysis to be
avoided by pre-analyzing.

Scoring: One point each.

11. Lazy evaluator.

```
(define (darrentron a b c)
    (if a (* a b) c))

> (darrentron (* 1 2) (* 3 4) (* 1 2))
```

The procedure call makes thunks of the three argument expressions. In the
following, we'll denote them as THUNK1[* 1 2], THUNK2[* 3 4], and
THUNK3[* 1 2]. Note that, even though two of the actual argument expressions
are equal, Scheme doesn't recognize them as the same; each becomes a separate
thunk.

Substituting these thunks into the body gives

```
        (if THUNK1[* 1 2] (* THUNK1[* 1 2] THUNK2[* 3 4]) THUNK3[* 1 2])
```

The IF expression has to force THUNK1 to decide whether to evaluate its
second argument or its third argument. The value is 2. The situation now
depends on whether or not promises are memoized:

```
memoized:      (if 2 (* THUNK1[2] THUNK2[* 3 4]) THUNK3[* 1 2])
not memoized:  (if 2 (* THUNK1[* 1 2] THUNK2[* 3 4]) THUNK3[* 1 2])
```

Here the notation THUNK1[2] means that this thunk remembers the value from
having been forced, so it won't call * if forced again.

Since 2 counts as true, IF will now evaluate its second argument. This is a
call to a primitive, so its arguments will be forced, resulting in the second
call to * if memoized, or the second and third calls to * if not memoized.
In either case we now have

```
        (if 2 (* 2 12) THUNK3[* 1 2])
```

This computes and returns the answer 24. THUNK3 is never forced. This
computation is the third call to * if thunks are memoized, or the fourth
call to * if they're not memoized.

So the answers are, 3 for memoized, 4 for not memoized.

Scoring: One point each.

12. Nondeterministic evaluator

(a) FIRST-N is written just as it would normally be, except that if the list is too short it /fails/ instead of giving an error:

```
(define (first-n lst num)
  (cond ((= num 0) '())                ; This clause has to come first!
        ((null? lst) (AMB))    ; This is how you express failure.
        (else (cons (car lst) (first-n (cdr lst) (- num 1))))))
```

The capitalized (AMB) above is the essence of this part of the problem.
Returning #F or FAIL or any other such value means that you don't understand
how to use the nondeterministic evaluator.

The problem says that the second argument is a nonnegative integer. In a real
programming project you'd check for an argument not in the domain, but that
isn't required for 61A exams unless specifically required in the problem.

Scoring: The general idea is that each half of this problem is worth three
points, on the scale

```
        3  correct or trivial error
        2  has the idea
        1  has an idea
        0  other
```

Here are the details for part (a):

```
3 - perfect
    off-by-one errors
    using list instead of cons
    returns reversed list

2 - not using amb in base case
    not checking if there are enough elements in the list
    using amb in recursive step with one of its branches proper and the other
      wrong/broken
    no base case for return list

1 - returning one element at a time (AMB as combiner)
    bad recursion, but checking that num >= (length ls)

0 - not returning anything!
```

(b) SUBLIST-OF-LENGTH works by noticing that any consecutive sublist either
includes the first element, in which case it's the one that first-n returns,
or doesn't, in which case it's part of a recursive call for (cdr lst).

```
(define (sublist-of-length lst num)
  (if (null? lst)
      (first-n lst num)       ; will return () for num=0, else fail
      (amb (first-n lst num)
           (sublist-of-length (cdr lst) num))))
```

Note that we don't subtract 1 from NUM in the recursive call! Unlike the
recursive call in FIRST-N, this one has to return the entire desired list,
not all but one element of it. (That goes along with the fact that the
"combiner" is AMB rather than CONS.)

Scoring: If your answer to part (a) computed the wrong function, but you use
that incorrect result correctly in part (b), you don't lose points twice.

```
3 - perfect
    correctly returning *all* sublists (not just consecutive)
    no base case on ls
    no (cdr ls) on the recursive call

2 - not putting first-n inside amb, only putting recursive call in amb
      (would only have to shift first-n into amb; anything more drastic
      to correct that would be a 1)
    calling require on first-n but not putting first-n inside amb
    using (car lst) instead of (first-n lst num)
    using (apply amb ...) for what would be a correct solution
      if APPLY AMB worked)
    thinking amb returns a list of alternatives

1 - answer embedded in the code, but confused about amb
      (MUST HAVE RECURSIVE CALL somewhere)
    bad combiner (i.e. not amb)

0 - not recursive
    computes wrong function
```

13. Mapreduce.

(a) A crucial thing to remember is that the way you count things in
mapreduce is to have the mapper produce key-value pairs with 1 as the
value, and then add them. But in this case, besides counting the input
kv-pairs, we also want to count the words. So the mapper makes values
that are lists of two numbers, one of which is the number of words in this
input kv-pair and the other of which is just 1.

```
(define countstream
  (mapreduce (lambda (kv-pair)
               (list (make-kv-pair (kv-key kv-pair)
                                   (list (count (kv-value kvp)) 1))))
             (lambda (old new)
               (map + old new))
             '(0 0)
             "rock-band-songs"))
```

If you didn't think to use MAP in the reducer, you could say

```
        (list (+ (car old) (car new))
              (+ (cadr old) (cadr new)))
```

but you should get in the habit of using higher order functions!

(b) In order to take advantage of the sorting by key that mapreduce does,
we have to swap keys and values, so the average phrase length is the /key/
of a kv-pair, and the song title is the /value/.

To get the average phrase length, we divide the total number of words in
the song by the number of phrases in the song. This is why two mapreduce
calls are needed; you can't take an average one new value at a time, averaging
the average-so-far with the new value. You have to do all the adding first,
then do one division (per song) in the second pass.

```
(mapreduce (lambda (kv-pair)
             (list (make-kv-pair (apply / (kv-value kv-pair))
                                 (kv-key kv-pair))))
           cons
           '()
           countstream)
```

Again, if you didn't think to use APPLY to do the division, you could say

```
        (/ (car (kv-value kv-pair)) (cadr (kv-value kv-pair)))
```

but isn't it more elegant if you're thinking in terms of higher order functions?

Since we didn't say what to do about ties (two songs with the same average),
you have a lot of flexibility about the reducer. I used CONS above, on the
theory that there won't be too many songs tied for a single average, so they
can all fit in memory. But you could use CONS-STREAM instead, or you could
even decide to choose one of the songs arbitrarily by using

```
        (lambda (new old) new)
or
        (lambda (new old) old)
```

as the reducer. Just be sure to pick a base case that matches your reducer.

Some people wanted to use something like < or MIN as the reducer. This is

presumably because we're looking for the lowest average. But it's a
pretty
severe misunderstanding of the group-reducing nature of mapreduce. Each
reducing process only sees the kv-pairs for songs with the same average!
And the values, which are what your reducer function sees, are song
titles,
not numbers. Alternatively, if you don't swap the keys and values,
then each
reducer will only see exactly one average, because part (a) gave us a
stream
in which there's only one kv-pair per song. Using MIN would make sense
if
we were doing a single non-parallel reduce of the results from part
(a), but
that's not what the problem asked you to do.

Scoring: 3 points for each part, on this scale:

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

Having the idea means both seeming to understand how mapreduce works and
seeming to understand how we intend to use mapreduce to solve this
problem,
but getting details wrong. It's hard to be more specific because the
solutions to this problem varied widely.

14. Logic programming.

First let's write it in Scheme. Since it has to do deep substitution,
the
easiest way uses car/cdr recursion:

```
(define (subst old new input)
  (cond ((equal? input old) new)
        ((atom? input) input)  ; this includes the case of an empty list
        (else (cons (subst old new (car input))
                    (subst old new (cdr input))))))
```

You could also write it treemap-style, with a recursive call inside a
MAP,
but this would be less useful since you're not given anything
equivalent to
MAP in the query system.

A good solution has one rule corresponding to each of the three COND
clauses:

```
(assert! (rule (subst ?old ?new ?old ?new)))

(assert! (rule (subst ?old ?new ?other ?other)
               (and (not (same ?other ?old))
                    (not (same ?other (?car . ?cdr))))))
```

```
(assert! (rule (subst ?old ?new (?incar . ?incdr) (?outcar . ?outcdr))
               (and (subst ?old ?new ?incar ?outcar)
                    (subst ?old ?new ?outcar ?outcdr))))
```

In the second rule, we have to rule out the case OTHER=OLD because, unlike
COND clauses, rules are not automatically mutually exclusive; more than one
rule could match a query. We also have to say that OTHER is an atom, not a
pair.

For solutions of this kind, using car/cdr recursion and atoms as base cases,
we graded as follows:

1 point for the rule (subst ?old ?new ?old ?new).

2 points for the (subst ?old ?new ?other ?other) rule if correct; 1 point if
the rule tries to do the right thing but fails (usually because half of the
body is missing).

3 points for the recursive rule, with 1 point for recursion on both car and
cdr, 1 point for avoiding unnecessary additional rules that cause duplicate
solutions, such as

```
        (subst ?old ?new (?old . XXX) (?new . YYY))
```

kinds of rules, and the third point if exactly right.

In addition, we gave a bonus point (but with max=6) for partly-right car/cdr
solutions that are at least valid logic programs: a SUBST relation with four
components, no composition of functions, etc.

A very common kind of wrong solution doesn't do /deep/ substitution, but only
checks elements of the top-level list. The division of points among the three
rules above wasn't workable for those papers, so we used a separate grading
scale with a maximum of 4 points. Here is a typical solution of this kind:

```
(assert! (rule (subst ?old ?new () ())))

(assert! (rule (subst ?old ?new (?old . ?x) (?new . ?y))
               (subst ?old ?new ?x ?y)))

(assert! (rule (subst ?old ?new (?other . ?x) (?other . ?y))
               (and (not (same ?other ?old))
                    (subst ?old ?new ?x ?y))))
```

For solutions like this, we gave 1 point for the base case [or for the real base case required by one of the examples, (subst ?old ?new ?old ?new)];
we gave 1 point for a correct rule for the car=old case; and we gave 2 points
for the car different from old case, 1 point for each half of the AND. We
also gave imperfect non-deep solutions the bonus point for valid logic programs (but with max=4).

15. Metacircular evaluator.

We're adding a special form, so this belongs in MC-EVAL, not in MC-APPLY.

The easy part is recognizing the new form:

```
(define (let!? exp)
  (tagged-list? exp 'let!))
```

We also have to add a COND clause in MC-APPLY to make this check. I found
it easiest to extract the list of bindings at this point, too:

```
(define (mc-eval exp env)
  (cond ...
        ((let!? exp) (make-bindings (cadr exp) env))
        ...))
```

Now for actually doing the work. One subtlety missed by most students is
that we asked for LET!, not LET!* -- that is, all the value expressions
should be evaluated /before/ any bindings are made! So the overall shape
of the solution should be something like this:

```
(define (make-bindings bindings env)
  (let ((vars (map car bindings))
        (vals (map (lambda (binding) (mc-eval (cadr binding) env))
                   bindings)))
    (for-each (lambda (var val)
                (make-binding var val env))
              vars vals)))
```

An alternative to the MAP that creates VALS would be

```
        (list-of-values (map cadr bindings) env)
```

There are two basic ways to approach MAKE-BINDING. One is to use either
LOOKUP-VARIABLE-VALUE or a special version of it to decide between DEFINE and
SET!; the other is to make a new version of SET-VARIABLE-VALUE! that does a

DEFINE instead of giving an error message if no binding is found. I
think the
second is cleaner, but the first was more common, so I'll discuss that
approach first. In either case, it's important to look for an existing
binding anywhere in the environment, not just in the first frame.

Many people just modified LOOKUP-VARIABLE-VALUE to return #F instead of
giving
an error message if the variable is unbound. This is problematic for
two
reasons: it changes the behavior of plain old variable lookup, and you
mustn't
break existing features when you add a feature; also, you can't
distinguish
between an unbound symbol and one whose value happens to be #F.

These problems could be solved with two improvements to the plan.
First,
instead of returning #F, return a value that can't accidentally be the
value
of an existing variable. The way to do this is to create a pair (an
STk pair)
just for this purpose:

```
(define unbound-token (cons '() '()))

(define (lookup-variable-value var env)
  ...
  (if (eq? env the-empty-environment)
      UNBOUND-TOKEN            ; was (error ...)
      (let ...))
  ...)
```

Second, you have to change ordinary variable lookup to recognize this
token
and give the error message:

```
(define (mc-eval exp env)
  (cond ...
        ((variable? exp)
         (LET ((RESULT (LOOKUP-VARIABLE-VALUE EXP ENV)))
           (IF (EQ? RESULT UNBOUND-TOKEN)
               (ERROR "UNBOUND VARIABLE" EXP)
               RESULT)))
        ...)
```

Nobody actually succeeded at this, although some students did recognize
that
there's a problem and tried returning an /unlikely/ value for unbound
variables. A much more straightforward solution is to make a not-
quite-copy
of lookup-variable-value that's a Boolean saying whether there's a
binding:

```
(define (binding-exists? var env)
  (define (env-loop env)
    (define (scan vars vals)
      (cond ((null? vars)
             (env-loop (enclosing-environment env)))
            ((eq? var (car vars))
             #T)       ;; was (car vals)
            (else (scan (cdr vars) (cdr vals)))))
    (if (eq? env the-empty-environment)
        #F             ;; was (error ...)
        (let ((frame (first-frame env)))
          (scan (frame-variables frame)
                (frame-values frame)))))
  (env-loop env))
```

Now you can solve the problem easily:

```
(define (make-binding var val env)
  (if (binding-exists? var env)
      (set-variable-value! var val env)
      (define-variable! var val env)))
```

Note, by the way, that although this problem has some similarities with
the
behavior of MAKE in the Logo interpreter -- combining aspects of DEFINE
and
SET! -- it should /not/ make new bindings in the global environment;
this
is still Scheme, and DEFINE adds bindings to the /current/ environment.

My preferred solution is to use an almost-copy of SET-VARIABLE-VALUE!
as the
basis for MAKE-BINDING:

```
(define (make-binding var val TOPENV)      ;; note formal parameter
  (define (env-loop env)
    (define (scan vars vals)
      (cond ((null? vars)
             (env-loop (enclosing-environment env)))
            ((eq? var (car vars))
             (set-car! vals val))
            (else (scan (cdr vars) (cdr vals)))))
    (if (eq? env the-empty-environment)
        (DEFINE-VARIABLE! VAR VAL TOPENV)   ;; was (error ...)
        (let ((frame (first-frame env)))
          (scan (frame-variables frame)
                (frame-values frame)))))
  (env-loop TOPENV))   ;; was ENV
```

I had to change the last formal parameter of MAKE-BINDING to a name
other
than ENV, because in the helper procedure ENV-LOOP, the value of ENV is
the
empty environment (even worse than the global environment!) at the time
we
need the original environment to add a binding to it.

Most solutions were variants of one of these two approaches. Other approaches
were wildly wrong (e.g., modifying MC-APPLY) or level confusions (e.g., doing
a DEFINE or SET! in STk, rather than in the metacircular evaluator's data
structures), and got no points.

Scoring: We started with 6 points and took off for mistakes:

-1 for not checking for LET! in MC-EVAL

-1 for implementing LET!*, alternating evaluations with bindings

-2 for not evaluating the value expressions at all

-1 for not checking (or checking only one frame) for existing bindings

-1 for never modifying an existing binding (or doing it wrong)

-1 for never adding a new binding (or adding it in the wrong place)

-1 if the solution doesn't work for an existing value of #F

-1 if you broke the "unbound variable" message

-1 if you /cause/ an "unbound variable" message for unbound LET! variables
   (i.e., you just use the unmodified lookup-variable-value)

-2 if you think a LET! expression has a body

-2 if you think there's only one binding in a LET! expression

-1 if you misparse the expression in some other way (getting caadadr wrong,
   or calling (eval-let! exp env) in MC-EVAL and then doing a recursive
   call to EVAL-LET! with just a list of bindings, no LET! in front)