CS 61A Fall, 2009 Final exam solutions

1. List mutation vs. variable binding

```
> (define x '(1))
> (define y '(2))
> (define z '(3))
> (set! y z)
> (set! x y)
> (set! z '(4))
> x
```

SET! doesn't mutate any pairs; it just changes the binding of a variable. So, the first SET! changes Y to (3); the next changes X to (3); and the third changes Z to (4). This last one has no effect on X and Y; they remember their /value/, not whatever expression gave rise to the value.

Answer: (3)

```
> (define mylist (list 2 4 6 8))
> (define (magic! ls)
      (set-car! (cdr ls) 100)
      (set! ls (cons 1 ls))
      (set-car! (cdr ls) 4))
> (magic! mylist)
okay
> mylist
```

This time we /do/ mutate pairs. When we call MAGIC!, the variable LS is initially bound to the value of MYLIST, namely (2 4 6 8). The first SET-CAR! changes the car of (4 6 8) (which is (CDR LS)) to 100, so LS is now (2 100 6 8) and so is MYLIST, which is bound to the same pair. The SET! gives LS the value (1 2 100 6 8), /without/ changing MYSLIST. (CDR LS) is (2 100 6 8), so SET-CAR!ing that makes LS (1 4 100 6 8). But that first element isn't part of MYLIST. So we get

Answer: (4 100 6 8)

Scoring: 2 points each, all or nothing.

2. Dynamic vs. lexical scope

```
> (define y 6)
> (define z 3)
> (define (foo x)
    (set! y (+ x y z))
    y)
> (define (bar z)
    (foo 10))
> (bar (foo 7))
```

The environment diagram is almost the same regardless of scope rule, so let's
start by drawing it:

The bindings Y=6 and Z=3 are added to the global frame G.

The third DEFINE has an implicit lambda, which makes the procedure

  P1: params (x), body (set! ...) y, env G

and adds the binding FOO=P1 to G.

The fourth DEFINE similarly makes

  P2: params (z), body (foo 10), env G

and adds the binding BAR=P2 to G.

So far, everything has happened in the global environment, because we haven't
called any procedures. Now we have to start being careful about scope.

To evaluate (bar (foo y)) we first evaluate the subexpressions (with current
environment G). The value of BAR is P2. (FOO 7) is a procedure call, which
we evaluate in the same way:

FOO (in G) is P1; 7 is self-evaluating. Now we call P1 with argument 7.
To do that, we make a frame in which the formal parameter [X] is bound to
the actual argument value [7]. Which frame does it extend? In lexical
scope, it extends the environment in which it was created, namely G. In
dynamic scope, it extends the current environment, which is also G. So, for
this procedure call it doesn't matter which rule we're following.

  E1: X=7, extends G

Now, with E1 as current environment, we evaluate the body of P1, which has
two expressions. The first is (SET! Y (+ X Y Z)). We start by evaluating
the procedure call (+ X Y Z). We find these bindings:

      + = addition primitive, in G
      X = 7, in E1
      Y = 6, in G
      Z = 3, in G

So the result of the addition primitive is 7+6+3 = 16. Now we do the SET!,
which modifies the binding of Y, in G, to 16.

  G: Y=16, Z=3, FOO=P1, BAR=P2

The second expression in the body of P1 is Y. We find the global binding Y=16, so (FOO 7) returns the value 16.

Now we can call (BAR 16). The current environment is again G. BAR is P2.
We make a frame with the binding Z=16. As before, this frame has to extend
G regardless of scope rule:

E2: Z=16, extends G

With E2 as current environment, we evaluate the body of P2, which is (FOO 10).
So we make a frame with the binding X=10. What environment should it extend?
The current environment is E2, but procedure P2 was created in G. So this is
where we have a choice:

E3-lex: X=10, extends G
E3-dyn: X=10, extends E2

This choice will affect the result, because the two environments have conflicting bindings for Z, which is not bound in E3 itself.

With E3 as current environment, we evaluate the body of P1. First we have
to evaluate (+ X Y Z), so we look for bindings for the subexpressions:

+ = addition primitive, in G
X = 10, in E3
Y = 16, in G
lexical: Z = 3, in G
dynamic: Z = 16, in E2

So the value of (+ X Y Z) is 10+16+3 = 29 in lexical scope, or 10+16+16 = 42 in dynamic scope.

We change the (global) binding of Y to either 29 or 42, and return that new value. So we have:

Answer: Lexical = 29, Dynamic = 42.

For the diagram, draw it as above, using E3-dyn.

Scoring:

1 point each for the two answers; 1 point total if you get them backwards.
No points for arithmetic errors etc.

2 points for the diagram. 1 point for lexical provided that you got the two

answers backwards, otherwise 0. 0 points for binding parameters to argument
expressions instead of to argument values. OK if your procedures have no right bubble (because it's unnecessary in dynamic scope).

3. Order of growth.

(a) In the worst case, you try all the other keys before you find the correct key, for a total of N trials, so this is Theta(N).

(b) After you open the outer box (worst case N trials), you know that /another/ key opens the inner box, so in the worst case you make N-1 trials, for a total worst case of 2N-1, which is still Theta(N).

(c) For the safe deposit box, you can't first find the left-lock key and then
find the right-lock key, because you won't know you have the correct left-lock
key until you are simultaneously trying the correct right-lock key. So you
have to try every pairing of keys, which means N possibilities for the left
key /times/ (not plus) N-1 possibilities for the right key. So in the worst
case you make N(N-1) = N^2 - N trials, which is Theta(N^2).

[Some of the TAs thought that it would be N(N-1)/2, which is the number of
combinations of N things taken 2 at a time. But we need permutations, not
combinations, because it matters which is the left key and which is the right key. If you don't know about permutations and combinations, ignore this paragraph, because N(N-1)/2 would still be Theta(N^2).]

Scoring: 2 points each, all or nothing. It's ok if you gave an exact formula such as 2N-1 for part (b), as long as your formula has the correct
order of growth. We graded this one leniently, as if the entire formula had a Theta() around it.

4. Applicative vs. normal order.

"Applicative order guarantees that arguments to a procedure are evaluated before the procedure body." This is TRUE -- it's the definition of applicative order. (The alternative is normal order, in which unevaluated
actual argument expressions are given to the procedure.)

"Normal order guarantees that arguments to a procedure are evaluated from left
to right." This is FALSE. Normal order delays argument evaluation until the

values are needed by a primitive called inside the procedure body, but it could turn out, for example, that the leftmost argument is /never/ evaluated,
as in the procedure (lambda (x y) (* y y)).

Scoring: 2 points each, all or nothing.

5. Miscellaneous short answer.

A list is a /PAIR/ whose /CDR/ is /A LIST/, or the empty list. This is the official definition of "list." We also gave some credit to solutions of the form "a list is a sequence of pairs whose final CDR is null", though
not as correct (a vector whose elements are pairs would also match this definition).

A list is more efficient than a vector when...

a) Removing items from the front of the sequence.

> YES. Changing the size of a vector requires copying the entire vector (other than the element you're removing). But removing the first element of a list is just CDR, which is constant time.

b) Replacing values at the front of the sequence.

> NO. Both SET-CAR! and VECTOR-SET! are constant time operations. (Replacing a value somewhere other than the front of the sequence takes longer for a list, because you have to call CDR repeatedly to find the element you want to change.)

c) Printing out every element.

> NO. This is Theta(N) time both for vectors and for lists. Since you want to print every element, there's no way to jump directly into the middle, which would give vectors the advantage.

d) Never; vectors are always more efficient.

> NO -- Operations that change the length of the sequence are more efficient for lists because there is no recopying.

A client-server connection's three-way handshake ensures...

a) The network is reliable.

> NO. It does ensure that (some part of) the network is working right now, but there's no guarantee that it will keep working until we're finished with this application.

b) Both sides can send and receive messages.

YES. It has to be a three-way handshake because after the first two messages have been sent and received, the originating host (typically the client) knows that it can both send and receive to/from the other host (typically the server), but the second host knows only that it can receive from the first; it doesn't know that the first host got its answer until it receives the third message.

c) The connection will be terminated when the client quits.

NO. This one is completely irrelevant!

d) None of the above.

NO, because (b) was correct.

The argument to a continuation is...

a) The previous expression evaluated by the interpreter.
b) The expression to evaluate next.

NO. A continuation is a procedure, and arguments to procedures are always values, not expressions!

/The continuation itself/ represents the expression to evaluate next, but it isn't the same thing as the expression.

c) The value computed by a procedure call.

YES. The continuation uses this value to continue (that's why it's called a "continuation") the computation.

Note, this says "computed by" rather than "returned by" because in continuation passing style, no procedure ever actually returns; it just tail-calls the next continuation.

d) An environment.

NO. Environments are not first class in Scheme.

Scoring: 2 points each, all or nothing.

6. Concurrency.

```
> (define s (make-serializer))
> ((s (lambda ()
        (parallel-execute
```

```
          (lambda () (set! z (- x y)))
          (lambda () (set! y (+ y x)))
          (lambda () (set! x 10)) ))))
```

NO. This serializes the PARALLEL-EXECUTE itself, but does not protect the
threads within the PARALLEL-EXECUTE from each other.

```
> (define s (make-serializer))
> (parallel-execute
   (lambda () (set! z (- x y)))
   (s (lambda () (set! y (+ y x))))
   (s (lambda () (set! x 10))) )
```

NO. This one is really hard to think about without listing all the cases.
It seems as if it could be safe, since the unprotected thread is the only one that modifies Z, and because it looks at X and Y once each, it can't have the sort of error in which (* X X) gets two different values of X when
another thread modifies X in the middle.

First of all, what are the possible correct results? "Correct" means "could happen given some sequential ordering of the threads." With three threads (let's call them A, B, and C), there are six possible orderings. Suppose the initial values are X=100, Y=200, Z=400.

```
ABC:  Z = 100-200 = -100.  Y = 200+100 = 300.   X = 10.
ACB:  Z = 100-200 = -100.  X = 10.              Y = 200+10 = 210.
BAC:  Y = 100+200 = 300.   Z = 100-300 = -200.  X = 10.
BCA:  Y = 100+200 = 300.   X = 10.              Z = 10-300 = -290.
CAB:  X = 10.              Z = 10-200 = -190.   Y = 200+10 = 210.
CBA:  X = 10.              Y = 200+10 = 210.    Z = 10-210 = -200.
```

Now, what can go wrong because A isn't serialized? It has to be that something else interrupts A. Let's look at this sequence of events:

A loads X=100 into a register.
C sets X to 10. (It runs to completion.)
B sets Y to 200+10=210. (It also runs to completion.)
A continues, loading Y=210 into a register, then setting Z to 100-210=-110.

(Note, even though X=10 at this point, thread A had already loaded X=100 into its local register.)

The problem is subtle. X=100 is a possible legal value. Y=210 is a possible
legal value. It's just not legally possible for /both/ of those to be true
/at the same time/, and that's why Z gets an impossible value.

```
> (define ser-x (make-serializer))
> (define ser-y (make-serializer))
> (parallel-execute
   (ser-x (ser-y (lambda () (set! z (- x y)))))
   (ser-y (ser-x (lambda () (set! y (+ y x)))))
   (ser-x (lambda () (set! x 10))) )
```

NO. The first two threads can deadlock, because they acquire the two serializers in the opposite order from each other.

```
> (define s (make-serializer))
> (parallel-execute
   (s (lambda () (set! z (- x y))))
   (s (lambda () (set! y (+ y x))))
   (s (lambda () (set! x 10))) )
```

YES. This is inefficient; since all threads are protected with the same serializer, it doesn't allow any parallelism at all. But it's safe.

Scoring: One point each, all or nothing.

7. MapReduce.

This is a little like the wordcount example, in that we turn the lines of the input stream into key-value pairs in which the key is a word from the text of a file. In the case of wordcount, we just want to add up how many times the word is used, so all the values in the key-value pairs we generate are 1. But this time we want to remember which file this word appears in, so the value will be the filename.

The other complexity is that we only care about certain words, and if the word isn't one of those, we don't generate a key-value pair for it.

```
(define (search keywords)
  (define (search-mapper line-pair)
    (map (lambda (wd) (make-kv-pair wd (kv-key line-pair)))
         (filter (lambda (wd) (member wd keywords))
                 (kv-value line-pair) )))
  (mapreduce search-mapper cons '() input-stream))
```

The nice thing about MapReduce is that it will handle the "sorted by keyword" aspect for us. The problem deliberately didn't specify the exact format of the output, so a solution using CONS-STREAM as the reducer would also be accepted.

By the way, although the problem says not to worry about the same word appearing more than once in the same file, if we wanted to eliminate duplicates, the reducer is where we'd do it.

Scoring:
2 points for making (word . filename) kv-pairs
2 points for filtering out non-keywords
1 point for returning a /list/ of kv-pairs in mapper
1 point for reducing correctly

We accepted solutions that used a STREAM-FILTER to do the filtering, though
this isn't the best use of MapReduce. Solutions that used multiple calls to
MapReduce, one for each keyword, got a maximum of 5 points, since the output
wasn't sorted by keyword as specified in the problem.

Some solutions had two-expression IFs to do their filtering, but (if #f '())
has a result of OKAY!. These solutions got a maximum of 5 points if there was
a simple value that could be put in for the third expression (such as the empty list) that would make it work, and 4 points if more post-processing would be needed.

8. Data Directed Programming

(a) You're told that each procedure takes the old position as argument and
returns the new position. So this is all you have to do:

```
(put 'robot 'forward (lambda (pos) (+ pos 1)))
(put 'robot 'back (lambda (pos) (- pos 1)))
(put 'robot 'home (lambda (pos) 0))
```

Since the PUT that's provided for you is two-dimensional, I used the word ROBOT as a sort of type tag. But we didn't take off for using a one-dimensional table with just the operation name as key, or for using a LOOKUP/INSERT! table.

It's also okay to define the procedures before putting them in the table:

```
(define (forward pos)
  (+ pos 1))

(put 'robot 'forward forward)
```

But in this case it's important that the key is the quoted operation name and

the value stored in the table is the (not quoted) procedure!

(b) We have to use GET to translate an operation name into a method:

```
(define (final-position init cmds)
  (if (null? cmds)
      init
      (final-position ((get 'robot (car cmds)) init) (cdr cmds))))
```

Scoring:

3 points for each part:

3 correct
2 non-conceptual error (e.g. HOME is the wrong function)
1 sets a global position variable (could be either or both parts)
1 (a) puts a non-function in the table
1 (b) list of procedures instead of command names
0 doesn't use GET/PUT

9. Streams

As usual for interleave problems, it's helpful to draw a zigzag diagram:

```
5  ___       ___       ___       ___       ___

        ___       ___       ___       ___
```

In the dashes on the first line we're going to put elements of my-stream as we learn them, and in the dashes on the second row we'll put 2 times the
elements of my-stream. We already know one element:

```
5  _5_       ___       ___       ___       ___

        _10       ___       ___       ___
```

Now we know the second and third elements:

```
5  _5_       _5_       _10       ___       ___

        _10       _10       _20       ___
```

And now we know the first seven elements, which is much more than we need:

```
5  _5_       _5_       _10       _5_       _10

        _10       _10       _20       _10
```

So the first eight elements are 5 5 10 5 10 10 20 5.

Scoring: The first three elements are worth 1 point total (all must be right), the next three are worth 1 point total, and the last two are worth 1 point each.

Solutions that got the output of INTERLEAVE backwards only lost 1 point total.
The same penalty was assigned to people who displayed the stream starting from the second element, or who squared instead of multiplying by 2.

10. Query sustem

(a) prefix

The examples gave you a big hint about the base case, which is that the empty list is a prefix of any list:

```
(assert! (rule (prefix () ?any)))
```

If the first list isn't empty, then its first element must be the first element of the second list too. And then the same relation must hold for the cdrs of the two lists:

```
(assert! (rule (prefix (?first . ?small) (?first . ?big))
             (prefix ?small ?big)))
```

The use of ?FIRST twice in the rule's conclusion is what requires the first
elements to agree, and the body of the rule is the recursive query about the two CDRs.

Many people weren't comfortable with reusing variables and tried this, which also works:

```
(assert! (rule (prefix (?car-a . ?cdr-a) (?car-b . ?cdr-b))
             (and (same ?car-a ?car-b)
                    (prefix ?cdr-a ?cdr-b))))
```

The trouble came when some people noticed that (prefix ?x ?x) was a valid rule and tried to use PREFIX as SAME. The problem with that is that it doesn't work properly for deep lists: (()) is a prefix of ((1)) that way. We took off a point for this.

There is also a solution that uses APPEND-TO-FORM, the relation for appending
from the book:

```
(assert! (rule (prefix ?pre ?full)
               (append-to-form ?pre ?something ?full)))
```

(b) sublist

The big hint is that you use PREFIX to help you define SUBLIST! In fact, that gives you the base case for this relation:

```
(assert! (rule (sublist ?a ?b)
             (prefix ?a ?b)))
```

If the first list isn't a prefix of the second list, then it has to be a sublist of the cdr of the second list:

```
(assert! (rule (sublist ?sub (?first . ?rest))
             (sublist ?sub ?rest)))
```

The two rules can be combined into one using OR, but you have to be careful. If you just say

```
(assert! (rule (sublist ?sub (?first . ?rest))           ; almost right
             (or (prefix ?sub (?first . ?rest))
               (sublist ?sub ?rest))))
```

this rule works only if the second list is non-empty, since its conclusion
requires that list to have a car and a cdr. So you need to add one special case:

```
(assert! (sublist () ()))
```

So you still have two assertions, and this version is, I think, harder to read. So I'd be inclined not to use OR in this problem. We weren't strict
about this, though, since this time it only eliminates that single case. (If you got this wrong in PREFIX, it meant that one list wasn't a prefix of
itself. But for SUBLIST, this case ends up being handled by PREFIX.)

Many people slipped up by using PREFIX for the "recursive" call instead of
SUBLIST; this fails to check the rest of the list. Others tried to make sure
the first elements /didn't/ match; this resulted in (1 2 3) not being a sublist of (1 1 2 3). The last common mistake was trying to merge the PREFIX
rules in as SUBLIST rules. This would work for sub-/sequences/, but fell down
for SUBLIST: (1 3 5) would then be a sublist of (1 2 3 4 5).

As with PREFIX, there is also a direct solution using APPEND-TO-FORM:

```
(assert! (rule (sublist ?sub ?full)
               (and (append-to-form ?something ?sub ?prefix)
                    (append-to-form ?prefix ?something-else ?full))))
```

Scoring: 4 points each half

4 correct
3 correct except for empty lists
2 misses significant cases but gets something right
1 severe logic errors
0 not logic programs (composition of functions, or just incoherent)

Answers with spurious parentheses around the arguments to a relation got a
maximum of 3 points.

11. Tree programming

If DEPTH is zero, then there is just one tree in the answer. But the answer still has to be a /forest/ with that one tree in it.

If DEPTH is greater than zero, then we have to combine the results of applying LEVEL to all this tree's children, with DEPTH reduced by 1. What does "combine" mean? Each result is a /forest/, i.e., a list of trees. We want the overall result to be a forest, too, not a list of forests. So we have to APPEND the results we get.

```
(define (level depth Tree)
  (if (= depth 0)
      (list Tree)   ; a forest with one Tree in it.
      (apply append     ; or use REDUCE instead of APPLY
                 (map (lambda (child) (level (- depth 1) child))
                          (children Tree)))))
```

That's it! Of course you can write a FOREST-LEVEL instead of calling MAP,
but it's not necessary.

Scoring:

8 correct.
7 trivial error, or base case wrong.
6 tries to flatten but does it wrong.
4 returns list of forests.
4 returns list of data.
4 works on binary trees.
3 serious domain errors, e.g., (level (- depth 1) (children tree)).
3 works on deep lists (correctly).
1 some kind of functional tree recursion.

0 even worse.

12. Metacircular evaluator.

This is a special form, so it's the job of EVAL, not APPLY. We'll add a COND clause to MC-EVAL, like this:

```
(define (mc-eval exp env)
  (cond ...
        ((while? exp) (eval-while exp env))
        ...))

(define (while? exp)
  (tagged-list? 'while exp))

(define while-test cadr)     ; selectors for while expression
(define while-actions cddr)
```

Now we have to do what WHILE is supposed to do: Evaluate the test expression. If it's false, we're done. Otherwise, evaluate the actions, and then do the same thing again.

```
(define (eval-while exp env)
  (if (true? (mc-eval (while-test exp) env))
      (begin (eval-sequence (while-actions exp) env)
             (eval-while exp env))
      'okay))
```

That's all! Luckily we had EVAL-SEQUENCE available to evaluate all the actions in one step.

Scoring:

2 points for detecting and parsing WHILE expressions
3 points for the logic of testing and acting
3 points for recursive evaluation of subexpressions