QUESTION 1: What Will Scheme Print?

What will the Scheme interpreter print in response to each of the following expressions? If any expression results in an error or infinite loop, just write "Error".

```
> (define (foo x)
    (define count (+ x count))
count)
> (define count 0)
> (foo 3)
_____________
> (foo 4)
_____________
>count
_____________
```

Solution and Explanation:

```
> (foo 3)
[Answer] 3
> (foo 4)
[Answer] 4
>count
[Answer] 0
```

The body of the procedure FOO creates a new variable count that is defined in foo's environment -- thus, each call to (foo ...) never modifies the variable COUNT that is defined in the global environment.Hence, why COUNT is still 0.

Rubric: 1 point each, no partial credit.

QUESTION 2: Pairs and Lists

a) Give an example of something that is a pair but not a list. If there is no such thing, write "impossible".

The simplest answer is something like (a . b). Note that this is a pair, but not a list, because a list is a series of pairs where the last pair in the series has a cdr of null.

b) Give an example of something that is a list but not a pair. If there is no such thing, write "impossible".

This one was a little tricky -- the empty list (i.e. null, or ()) was the correct answer.

c) What will the Scheme interpreter print in response to the second expression below? If it results in an error or infinite loop, just write "Error".

```
> (define my-stream (cons-stream 1 my-stream))
> (list (pair? my-stream) (list? my-stream))
```

(#t #f) was the correct solution. A stream is a pair, because a stream is a pair whose car is a value (the stream-car), and whose cdr is a promise to compute the rest of the stream.

A stream is not a list, however -- the cdr of my-stream is a promise. In order for my-stream to be a list, it would either have to have a cdr of null, or point to a list -- a chain of pairs that eventually points to a pair whose cdr is null.

Note that (list? my-stream) doesn't infinitely keep "recursing" down the infinite stream of 1s -- the predicate LIST? won't force the promise, it only checks to see if the cdr is null, or if it points to another pair.

Rubric: 1 point for (a), 1 point for (b), 2 points for (c):
Part c:
(#t #f)   - 2 pts
(#t . #f) - 1 pt
(#f #f)   - 1 pt
(#t #t)   - 1 pt
all others - 0 pts

QUESTION 3: Orders of Growth

```
(define (pinky lst)
  (if (null? lst)
```

```
    0
    (+ (accumulate + 0 lst) (pinky (cdrlst)))))
```

Theta(N^2), because for each element in the list, we call ACCUMULATE, which takes Theta(N) time. So, since there are N elements in the list, and for each one we call a procedure that takes Theta(N) time, we get N^2.

(Some people might notice that because the list gets one element shorter each time, the ACCUMULATE will go faster and faster. This turns out to give us half of N^2. But we don't care about constant factors when talking about orders of growth, so the answer is still Theta(N^2))

```
(define (brain lst)
  (if (null? lst)
      0
      (+ (+ (car lst) (brain (cdrlst)))
            (brain (cdrlst)) )))
```

Theta(2^N), because each call of BRAIN spawns two more calls of BRAIN, so the number of calls grows exponentially as the size of the list grows. If your list is of length N, there will be 2^N calls of BRAIN.

2 points each, all or nothing.

QUESTION 4: Environments

Consider the following code:

```
STk> (define (foo x)
       (let ((x 7) (f (lambda (z) (+ z x))))
       (set! x 3)
       (f 10) ))
STk> (foo 9)
```

Running it will create the following environment diagram. Complete the diagram by adding in five missing arrows and the result of the SET!, then in the result of the final expression.

E2 points to G, both procedures, E1, and E3 point to E2. The SET! changes the x in E3 to point to 3. The result of the final expression is 19.

Defining FOO creates a procedure that points to the global environment. Calling FOO creates the environment E2 that points to where FOO points to -- the global environment. LET creates a procedure and then calls it, so the LET creates a procedure bubble that points to E2, since that's where we are currently. We evaluate the arguments first. X is 7, and F is a procedure that points to E2, since that's where we are currently. Now, calling the LET procedure creates an environment E3 that points to where the LET procedure points to -- E2. Now we're in E3. We set x to be 3 in E3, and we call F on 10. This creates an environment E1 that points to where F points to -- E2. Now we're in E1, and we add 10 + 9 to get 19.

1 point for each arrow, 1 point for the set!, 1 point for 19.

QUESTION 5: Scoping

Given the following code:

```
(define (foo x y)
  (bar 1 y))
(define (bar a b)
  (* x y))
(foo 3 4)
```

What is the result of the final expression using lexical scope?
[Answer] ERROR

What is the result of the final expression using dynamic scope?
[Answer] 12

Which does Scheme use?
[Answer] Lexical scope

Both FOO and BAR are procedures defined from the global environment -- thus, their lambdas "point to" the global environment.

In lexical scope, when FOO calls (bar 1 y), the newly-created environment for BAR extends the global environment. There is no variable x or y in the global

environment, so there is an unbound variable error for x, y.

In dynamic scope, when FOO calls (bar 1 y), the newly-created environment for BAR instead extends FOO's environment (the caller's environment). Thus, BAR can resolve the variables x and y as the argument variables for FOO, thus ultimately returning 12.

Scheme does indeed use Lexical scope. On the other hand, Logo uses Dynamic scope.

Rubric: 1 point for each blank. Alternately, you could have earned one point for the answers (12, error, dynamic), as it just means you mixed up the meanings of dynamic and lexical.

QUESTION 6: Client-Server Programming and Therac-25

(a) Ben Bitdiddle is able to send chat messages to Alyssa P. Hacker, but he isn't getting the replies she sends. Which of the following is most likely the problem?

- His client didn't set a callback.
- His client didn't finish the three-way handshake.
- The server isn't sending out updated client-lists when clients log on.
- The server isn't forwarding chat messages to their destinations.

The correct answer is "his client didn't set a callback". This would keep the client from processing any incoming messages after completing the three-way handshake, but wouldn't affect outgoing messages.

If the client didn't finish the three-way handshake, Ben wouldn't have been able to receive /or/ send messages.

Even if the server didn't update the client list, Ben should still be able to receive chat messages.

If the server didn't forward messages to their destinations, Alyssa wouldn't have been able to receive the messages Ben sent.

(b) Which of the following fixes for the Therac-25 would have prevented the most accidents?

- Add hardware interlocks, like the earlier Therac-20.
- Log any software-detected abnormalities.
- Ask operators to confirm high radiation doses.
- Use serializers to prevent concurrency issues.

The correct answer is to add hardware interlocks. With these, it would not have been possible for the machine to have produced an out-of-alignment or overpowered beam.

Logging abnormalities would have helped to figure out what went wrong, but it wouldn't have prevented any of the accidents directly.

The software already asked operators to confirm many operations they put in. Adding another confirmation probably would have simply trained people to press "Yes" once more. The user interface was already a source of complaints and unreliability; even optimistically, this would not have prevented as many accidents as hardware interlocks.

"Use serializers to prevent concurrency issues" is a relatively unspecific solution. Even if all of the concurrency issues with the Therac-25 were fixed, however, there were still user interface issues that could have caused accidents.

Grading: 2 points each, all or nothing.

QUESTION 7: Concurrency

a)
N Incorrect Results
Y Deadlock
N Inefficiency (missed opportunity for parallelism)

Explanation:
There are no incorrect results -- the first expression can only execute if it has acquired both cereal-1 and cereal-2. Similarly, the second expression

can only execute if it has acquired both cereal-2 and cereal-1. Since a serializer can only be held by one expression, each expression is guaranteed to be atomic (assuming no deadlock happens).

Deadlock can occur -- consider the following scenario:

exp1 (the first expression) acquires cereal-1
exp2 acquires cereal-2

exp1 tries to acquire cereal-2, but it can't, so it waits for exp2 to release cereal-2

exp2 tries to acquire cereal-1, but it can't, so it waits for exp1 to release cereal-1

DEADLOCK

Finally, there's no missed opportunity for parallelism. Since both expressions are mutating the shared variable z, we must guard access to z with a serializer. There's no way to guarantee "correct" solutions if we ran exp1 and exp2 concurrently with no protection.

b)
N Incorrect Results
N Deadlock
Y Inefficiency (missed opportunity for parallelism)

Explanation:
There are no incorrect results -- each expression exp1, exp2, exp3 can only run if they have both acquired the serializers x-serializer and y-serializer. Since a serializer can only be held by one expression at a given time, each expression is guaranteed to run without being interrupted by a different expression.

There is no possibility for deadlock. Unlike in part (a), each expression acquiresserializers "in order" - so, there will never be a "circular waiting" scenario.

Finally, there is missed opportunity for parallelism. Notice that the first two expressions exp1, exp2 only use the variable x, while the third expression exp3 only uses the variable y. So, exp3 is independent from exp1 and exp2 -- thus, we could have exp3 run during the execution of either exp1 or exp2 and still get correct results. However, the code currently does not allow exp3 to run during the execution of exp1 or exp2 -- this is inefficient.

Rubric: 1 point for each answer (no partial credit).

QUESTION 8: Lazy Evaluator

After evaluating these expressions, what is the value of count?
3

What is the value of binky? (If any expression generates an error or infinite loop, just write "Error". If either value is a promise, just write "Promise").

Promise/Thunk

Explanation:
Ultimately, the line (define binky (akbar (jeff))) binds "binky" to the result of (akbar (jeff)). Since (jeff) is the argument to the compound-procedure AKBAR, it gets delayed. For clarity, let's refer to the delayed (jeff) as <promise (jeff)>.

Then, we enter the body of AKBAR. We do (set! count 3), then we set! hoho to point to <promise (jeff)>, then we return HOHO (which is <promise (jeff)>). Finally, BINKY points to the return value of AKBAR, which is <promise (jeff)>.

So, COUNT is 3, and BINKY is <promise (jeff)>. Because (jeff) was never invoked (i.e. its promise was never forced), the body of JEFF is never executed.

Rubric:
3, promise (or 3, thunk) - 4 pts
5, 42   - 2 pts
3, 42   - 1 pt
0, promise (or 0, thunk) - 1 pt
all others   - 0 pts

QUESTION 9: MapReduce
(a)
```
; A correct solution:
(mapreduce count-words max 0 "/shakespeare")
```

Here, the key concept is to realize that the reducer is called with two arguments at a time, which in this case are two line-counts. Since we want the longest line from each play, we can simply find the "max" line length, so we need to change the reducer to MAX.

Another very common solution is to explicitly make a new reducer, as following:

```
(define (greatest x y)
  (if (> x y)
x
      y))
```

This works basically the same as MAX, but was a little extra work.

Grading:
3 pts - Correct

2 pts - The Idea
- changing the reducer to the > function (this does not return a number!)
- assuming that the reducer function takes in two /key-value pairs/
- changing the base case incorrectly (i.e. '())

1 pt - An Idea
- Assuming the reducer was a one argument function that took in the entire list of values
- Incorrectly thinking that the reducer takes in the entire line from the play
- Calculating the longest word in each play

0 pts - Writing code that does not work, or not showing progress towards the correct solution

(b)
```
; A correct solution
(define (count-thou input)
  (list (make-kv-pair (kv-key input)
                      (length (filter (lambda (x) (eq? x 'thou))
                                      (kv-value input))) )))
(map-reduce count-thou + 0 "/shakespeare")
```

count-thou filters every thou in the line from the play, and then finds the length of the remaining line to get the count of thous. The reducer does not need to be changed from the original problem, since all we need to do is add every count of thou together per play.

A common mistake was to only count the word thou once per line, as follows:
```
; A working, but incorrect solution
(define (count-thou input)
  (list (make-kv-pair (kv-key input))
        (if (member 'thou (kv-value input)) 1 0) ))
```
This solution simply counts the number of *lines* with the word thou, rather than the number of overall occurrences of the word though (there can be multiple in one line!).

Grading:
3 pts - Correct

2 pts
- Only handling one thou per line
- Having the mapper return a key-value pair, rather than a list of kv-pairs
- Making a comparison like (eq? kv-pair 'thou).

1 pt
- Changing the key to the word "thou". This counts the total number of thous in /all/ plays, rather than giving the counts from each play.

0 pts - other

QUESTION 10: Accumulate

Our ideal solution looked very similar to a reducer you might write for a

mapreduce call. Remember that reducers and procedures that you might pass to accumulate work very similarly, so you should have only needed to compare the kv-value of the next pair with the kv-value of the largest so far.

```
(define (lv-helper next so-far)
  (if (> (kv-value next) (kv-value so-far))
next
      so-far))
```

This type of procedure is so common, in fact, that a perfectly acceptable solution was the following:

```
(define lv-helper find-max-reducer) ; as defined on page 353 of reader
                                    ; vol. 2
```

We saw a surprisingly large number of recursive solutions; as a general rule of thumb, procedures such as these don't need to be recursive! This is best explained by looking at the actual procedure body of accumulate:

```
(define (accumulate fn base ls)
  (if (null? ls)
base
      (fn (car ls) (accumulate fn base (cdrls)))))
```

Note that accumulate already does the job of calling its argument recursively; this means that recursive helpers tended to be overly complex and non-functional.

Some solutions also attempted to use a local state variable instead of using a so-far argument, but this didn't work with our definition of largest-value, which was:

```
(define (largest-value assoc-list)
  (accumulate lv-helper (car assoc-list) (cdrassoc-list)))
```

Note that accumulate is called with the first kv-pair as its base case, whereas solutions that attempted local state generally ignored the so-far argument (and by extension, the base case) entirely. We awarded these solutions 2 points.

The grading scale went as follows:

4 points - perfect
 - The way in which ties were handled was irrelevant
 - Returning a new kv-pair rather than the original was okay (though not ideal, due to eq-ness)

3 points - trivial mistakes
 - Reversed the comparator (e.g. > instead of <)
 - Violating data abstraction (e.g. car instead of kv-key)

2 points - the idea
 - Correct domain/range, with some correct comparison, but other serious flaws
 - Not handling the base case (e.g. local state mishaps)
 - Incorrectly violating data abstraction (e.g. cadr instead of kv-value)

1 point - an idea
 - Incorrectly establishing an ADT (e.g. (define value cadr))
 - Domain is a single list of kv-pairs
 - Domain is two numbers (instead of kv-pairs) and range is a number
 - Domain is a kv-pair and a list of kv-pairs
 - Correct domain, incorrect range (such as just returning a number)

0 points
 - No comparisons made between the values
 - Major domain/range issues, such as returning just a key

QUESTION 11: Trees

The function check-children is supposed to return #t if every node in a Tree has a datum equal to its number of children. Fix all errors and data abstraction violations (DAVs) in the following incorrect implementation of check-children.

```
(define (check-children tree)
  (or (= (first tree) (length (cdr tree)))
      (accumulate (lambda (x y) (and x y))
```

```
                #f
                (every check-children
                       (cdr tree) ))))
```

The correct answer is as follows (six changes):

```
(define (check-children tree)
  (AND (= (DATUM tree) (length (CHILDREN tree)))
       (accumulate (lambda (x y) (and x y))
                   #T
                   (MAP check-children
                        (CHILDREN tree) ))))
```

+1pt per error (standard correct values in comments on each error).
-1pt for each "correction" that breaks something
0pt for each "correction" that doesn't help or hurt
0pt for a correction to something broken that doesn't actually fix it

Common errors were missing the OR/#f, or noticing one then not correcting it "correctly". Some students tried to completely rewrite it, which we gave full credit to if and only if it actually would work, otherwise you could miss up to 2 points for screwing it up even more and any points we couldn't allocate for you fixing things.

QUESTION 12: Streams

Define the following stream. You may use helper procedures.

```
> (show-stream strm 15)
(1 1 2 1 2 3 1 2 3 4 1 2 3 4 5 ...)
```

First, a correct solution:

```
(define (count-up n)
  (define (helper i)
    (if (= i n)
        (cons-stream i (count-up (+ n 1)))
        (cons-stream i (helper (+ i 1))) ))
  (helper 1) )

(definestrm (count-up 1))
```

This mirrors the structure of the problem pretty well: count up to a certain number, then when you get there, start at the beginning of the next bit. In this solution, N is the number we're counting to before starting over, while I is the number we're on. Many people had solutions like this one, though sometimes they combined COUNT-UP and HELPER into one function.

The most common incorrect answer involved being too smart:

```
; uses STREAM-APPEND and STREAM-ENUMERATE-INTERVAL from SICP
(define (count-up n)
  (stream-append (stream-enumerate-interval 1 n)
                 (count-up (+ n 1)) ))
```

This correctly describes the stream we want to create: an infinite stream that starts with the sequence from 1 to N. The problem is that STREAM-APPEND is not a special form like CONS-STREAM; it doesn't delay its arguments. This means that the recursive call to COUNT-UP happens immediately, causing infinite recursion even before the call to STREAM-APPEND actually happens.

A few people managed to fix this the same way as in the book:

```
(define (count-up n)
  (stream-append-delayed (stream-enumerate-interval 1 n)
                         (delay (count-up (+ n 1))) ))
```

This works, and it's actually much nicer than the first solution above, because it very clearly says what the stream is in the definition. Even with the necessary explicit delay.

Another common family of incorrect answers created streams that looked like this:

```
((1) (1 2) (1 2 3) (1 2 3 4) (1 2 3 4 5) ...)
```

Individual answers differed on whether the inner sequences were lists, streams, or some irregular pair structure. It's possible to flatten this

stream as well and thus get a correct answer:

```
(define (lists-from n)
  (cons-stream (cons-stream n the-empty-stream)
               (stream-map (lambda (sub-stream) (cons-stream n sub-stream))
                           (lists-from (+ n 1)) )))

(define (stream-flatten strm)
  (define (helper sub-stream)
    (if (eq? sub-stream the-empty-stream)
        (stream-flatten (stream-cdrstrm))
        (cons-stream (stream-car sub-stream)
                     (helper (stream-cdr sub-stream)) )))
  (helper (stream-car strm)) )

(definestrm (stream-flatten (lists-from 1)))
```

But a number of people decided to use the STREAM-FLATTEN from the book. This unfortunately doesn't work, even if you remembered to make a stream-of-streams instead of a stream-of-lists, because the book's STREAM-FLATTEN uses INTERLEAVE in order to "flatten" a stream of potentially infinite streams. We considered this a trivial mistake, since this isn't how the regular FLATTEN behaves and it's not what we'd expect.

Grading
6 perfect
5 trivial mistakes
- off by one (missing first/last number in each sub-sequence)
- reversed counting (1 2 1 3 2 1 4 3 2 1 5 4 3 2 1 ...)
- uses one variable for both I and N
- uses STREAM-FLATTEN or STREAM-FLAT-MAP from book (which uses INTERLEAVE)
4 "the idea"
- solutions that would work if STREAM-APPEND delayed its second argument
- uses STREAM-FLATTEN or STREAM-FLAT-MAP from book, but the elements of the top-level stream aren't actually streams
3
- solutions that used CONS-STREAM where the first argument was a stream, but meant to append the two streams
- solutions that give the correct sequence but with nesting ((1) (1 2) (1 2 3) (1 2 3 4) ...). Any sort of nesting is okay.
2 "an idea" (answers varied)
1 correctly makes a stream
0 other

QUESTION 13: Object-Oriented Programming

a)
+1 Declares ANIMAL as a parent correctly.
+1 Use USUAL correctly to call ANIMAL class's EAT method.
+1 SET! EVERYTHING-EATEN to the list constructed in the FISH's EAT method.

Since the problem states that "A fish is a kind of animal", the is-a relationship shows that FISH is a child class of the ANIMAL class. Student often got the syntax wrong. A common case was "(parent animal)" instead of "(parent (animal)".

The next part involves the variable LAST-EATEN in the parent class of ANIMAL. With the current code, LAST-EATEN is never updated with the last thing eaten. We need a call using USUAL to make sure the default behavior. (Notice that we can't directly set LAST-EATEN, because it's in the parent class, and the FISH class shouldn't create its own LAST-EATEN, because then the ANIMAL methods might be out of sync with the FISH's LAST-EATEN.)

Lastly, the CONS expression in the current EAT method doesn't do anything. A SET! is required to correctly set the EVERYTHING-EATEN variable.

The USUAL and SET! statement may be switched, since we never specified the output of the EAT method.

b)

fish SHOULD be made a parent of goldfish. Again, goldfish is a fish. The problem says that goldfish behaves exactly like a fish.

animal SHOULD-NOT be made a parent of a goldfish, because animal should already be a parent of the fish class.

An instance variable (or instantiation variable) named owner SHOULD be

included in the goldfish class, because we need a variable to keep track of who the owner is.

An instance variable (or instantiation variable) named last-eaten SHOULD-NOT be included in the goldfish class, because the fish class which is a parent class of goldfish already has this last-eaten variable. Declaring a new "last-eaten" variable doesn't ensure the correct behavior of goldfish's parent classes.

An instance variable (or instantiation variable) named everything-eaten SHOULD-NOT be included in the goldfish class for the same reason as last-eaten not being an instance var.

We SHOULD-NOT have an owner method for the goldfish class. All the method is expected to do is to return the owner variable, which is a property that an instance variable (or instantiation variable) already has.

We SHOULD-NOT have a new version of the eat method, since nothing is changed and we want the goldfish class to behave exactly the same as the fish class.

We SHOULD-NOT have a new version of the worms-eaten method, because goldfish behaves the same in worms-eaten.

0.5 points for each correct marked Y/N.

The total for question 13 was rounded down.

QUESTION 14: Logic Programming

Answer:

```
(rule (every-other () ()))
(rule (every-other (?x) (?x)))
(rule (every-other (?a ?b . ?l-rest) (?a . ?r-rest))
      (every-other ?l-rest ?r-rest) )
```

Clearly, this set of rules needs a recursive case because it must be able to handle lists of arbitrary length. Because we want to get rid of every other element, the recursive case has to look at the list two elements at a time. It keeps the first element, throws out the second, and then recurses on everything else, which is the sublist starting with the third element. Because of the way the recursive case works, we needed two base cases: one for even-length lists, and one for odd-length lists. If the list has an even number of elements, it will reduce the list two elements at a time until the list is empty, at which point the query system finds a match with the empty list case. Otherwise, the list has an odd number of elements, and it goes through the same recursive process, but stops when there is only one element left, in which case it matches the second base case.

Grading:
6 perfect
- no points off if you didn't use assert!. But you did need to use the "rule" keyword (except for the empty list base case)
- accepted base cases for 2 and 1 elements instead of 0 and 1 elements
- unsimplified pair structures also worked:
  (every-other (?a . (?b . ?l-rest)) (?a . ?r-rest))
5 trivial mistakes
- quotes anywhere
- missing question marks in front of variables
- missing one of the two base cases but recursive case is completely correct
- included extra base cases that lead to duplicate results
4 "the idea"
One of the following:
- missing both base cases, but recursive case is completely correct
- at least one base case and recursive case is a little off
2 "an idea"
- missing both base cases and recursive case is a little off
1 wrote a valid query system program
Includes cases where:
- recursive case missing completely
- solving some other unrelated problem
0 didn't write a valid query system program