1. What will Scheme print?

```
> (define (poof x)
    (lambda (y) (+ x y)) )
> (define bam 10)
> ((poof bam) 4)
```

__14__

    A better name for POOF would be MAKE-ADDER; (po
of bam) returns a function we
    might call +10. Then we pass 4 to that function
 and get 14.

```
> (define swoosh
    (lambda (num)
      (lambda (y) (+ y num)) ))
> (define kablooie 10)
> ((swoosh kablooie) 4)
```

__14__

    The trick here is that it's pretty much exactly
 the same problem as the
    previous one. SWOOSH is still basically MAKE-AD
DER; we just wrote the
    lambda out explicitly here.

```
> (define (mystery sent)
    (if (empty? sent)
        '()
        (se (first sent) (mystery sent)) ))
> (mystery '(you cant always get what you want))
```

__Infinite loop__

    In this case, you had to notice that the recurs
ive call to MYSTERY doesn't
    actually make SENT any shorter. Since the base
case is when the sentence is

empty, we'll never reach it!

```
> (define garply 5)
> (and (> garply 0) (/ 20 garply))
```

__4__

This may have been the most-missed problem on the entire exam. The way AND works is it goes through each argument and sees if it's false. If so, it just stops there. Otherwise, it returns the value of the /last/ argument.

We're pretty sure this came up in the book or lab, but it threw a number of people. Many people put #t as the result, but since any non-#f value counts as true, AND can get away with the behavior it actually has. Some people said this was an error, presumably because (/ 20 garply) wasn't #t or #f. Again, in Scheme any value other than #f counts as true!

```
> (define brrrm '((we need to) (go (deeper))))
> (cdar brrrm)
```

__(need to)__

The most common mistake here was to read this as (cadr brrrm) instead of (cdar brrrm). Remember, the As and Ds follow the same order as nesting CARs and CDRs: (cdr (car brrrm)).

```
> (car brrrm)
(we need to)
> (cdr (car brrrm))
(need to)
```

Scoring: 1pt each, all or nothing.

2. Fill-in-the-Blanks / Box-and-Pointer Diagrams

```
> (LIST 'a '(b . c) #f)
(a (b . c) #f)

   +---+---+   +---+---+   +---+---+
-->| o | o---->| o | o---->| o | / |
   +-|-+---+   +-|-+---+   +-|-+---+
     |           |           |
     v           v           v
     a         +---+---+    #f
               | o | o |
               +-|-+-|-+
                 |   |
                 v   v
                 b   c
```

    This wasn't too hard to pick a procedure for, since CONS only takes two
    arguments and APPEND only takes lists. The box-and-pointer diagram wasn't
    too hard either: the list has three elements, so we draw three "spine"
    pairs. Then we fill in the elements. A and #f are easy. (b . c) is a single
    pair, which is also not too bad.

```
> (CONS '(a) '(b))
((a) b)

   +---+---+   +---+---+
-->| o | o---->| o | / |
   +-|-+---+   +-|-+---+
     |           |
     v           v
```

```
   +---+---+     b
   | o | / |
   +-|-+---+
     |
     v
     a
```

This was a little trickier, but the other list functions wouldn't have helped here:

```
> (list '(a) '(b))
((a) (b))
> (append '(a) '(b))
(a b)
```

What does CONS do? It just makes one new pair whose car is the first argument and whose cdr is the second argument. What's the car of the result? (a). What's the cdr? (b). Hey...that looks good!

There were two good ways to do this diagram. The first is to follow the same algorithm as last time: draw the spine pairs for the list, then fill in the elements. The second is to use the fact that you're doing a CONS: you can draw the diagrams for the lists (a) and (b), then make a new pair whose car is (a) and whose cdr is (b).

```
> (APPEND (LIST #t #f) '(maybe dunno))
(#t #f maybe dunno)

   +---+---+   +---+---+   +---+---+   +---+---+
-->| o | o---->| o | o---->| o | o---->| o | / |
   +-|-+---+   +-|-+---+   +-|-+---+   +-|-+---+
     |           |           |           |
     v           v           v           v
```

```
    #t          #f          maybe       dunno

    The box-and-pointer diagram was pretty straightforward; it's just a four-
    element list. Like the first problem in this set, we have a list that's not
    all words or all booleans (true-falses).

    The functions were a little trickier to figure out; it was easier if you
    looked at the outside one first. We're trying to put two elements in front
    of the list (maybe dunno). CONS won't do it; it attaches a /single/ element
    to the front of a list. LIST won't do it, since it makes a /new/ list out
    of its arguments (in this case, we'd get a two-element list). APPEND,
    though, might just do what we want.

    If we do use APPEND, we know both arguments have to be lists, and we know
    they'll be appended together. So we can just use LIST as the inside
    procedure to call.


Scoring:
+1pt for each correct procedure call
+1pt for each correct box-and-pointer diagram
-1pt if any start arrows are missing
-1pt if any arrowheads are missing
-1pt if values in the box-and-pointer diagram are quoted.
     They're values, not expressions!



3. Orders of Growth (BAR and FOO)

(define (bar n)
  (define (baz n)
```

```
    (if (= n 0)
        0
        (+ n (baz (- n 1))) ))
  (* (baz n) (baz n)) )
```

Answer: Theta(N)

Our first step is going to be figuring out the run time of BAZ. After all, since
BAR calls BAZ, it's going to change our answer.

First off, other than the recursive call, how long does a single call to BAZ
take? We do =, -, and +, but no other functions. All of these take constant time
(with reasonably sized numbers, anyway), as does the IF. So one time "through"
BAZ takes constant time.

Now we have to see how many times BAZ will be called. The base case is when N is
zero. If N isn't zero, how many recursive calls does it take before it is? Well,
we subtract one from N each time, so it'll take N times. Multiply that by the
time for one call to BAZ (constant), and we get Theta(N) time for BAZ.

Now we can jump back out to BAR. What do we have to do to evaluate the body?

```
1. Compute the first (baz n)    Theta(N) time
2. Compute the second (baz n)   Theta(N) time
3. Multiply the two results     Theta(1) time
```

We do all three of those, so we can add up the times and get Theta(2N+1).
But we don't care about constant factors or "small" terms, so that's still
Theta(N). So BAR also takes linear time.

There were two "too-hasty" answers here. The first

was people who said this
would be Theta(N^2) time, or "quadratic" time. Reme
mber, just because it's
doing multiplication doesn't mean you multiply the
times. If it was subtracting
the two values, you wouldn't subtract the times.

The second mistake was to see the two calls to BAZ
and mistake them for
recursive calls, like for computing numbers in Pasc
al's triangle. When you make
two recursive calls, and those recursive calls make
two recursive calls, you
often end up with Theta(2^N) time. This is "tree re
cursion", cause you get the
same sort of tree shape we drew in class. (These ar
e the sort of problems where
when your computer gets twice as fast you can compu
te one more item.) But since
these /weren't/ recursive calls, they're just part
of "how long does one time
through BAR take?".

```
(define (foo n)
  (if (= (remainder n 7) 0)
      n
      (foo (- n 1))))
```

Answer: Theta(1)

This was admittedly a tricky question, but you've s
een it before! Let's follow
the same steps. First, how long does one time throu
gh FOO take? REMAINDER, =,
and - all take constant time, as does IF, so one ti
me through is still constant.

Now, what's the base case? When N is a multiple of
7 (that's what it means for
the remainder of n/7 to be 0). If N /isn't/ a multi
ple of 7, how many recursive

calls will it take before it is? We subtract one from N each time, so it can
take /at most six/ recursive calls before we reach the base case.

So our /worst/ case is 6 recursive calls, times constant time for each call,
which is overall still constant time. Theta(1). Really!

The point of this question (both parts) was that trying to pattern-match
orders-of-growth problems ("it subtracts 1 from n, so it's linear") doesn't
always work. You really have to say "how long for one time through? how many
times through?" to get the right answer.

Scoring: 2pts each, all or nothing.

4. Orders of Growth (MAP)

We have a list of N sentences, each with N words, called PARAGRAPH. How long
does it take to run (map count paragraph)?

Answer: Theta(N^2)

Even without the code for MAP, we know it's going to go through each item in
the list only once. And COUNT takes Theta(N) time, where N is the number of
words in a sentence. So we have to add up all the COUNT calls made by MAP;
there are N of them (one per sentence in the paragraph). This comes out to
Theta(N^2).

How long would it take to run (map selection-sort p

aragraph)? Selection sort is
a Theta(N^2) algorithm.

Answer: Theta(N^3)

    This time, we know running SELECTION-SORT on on
e sentence will take
    Theta(N^2) time. And we have to run it on N sen
tences. So we add up all the
    N^2s and get Theta(N^3).

If it takes time t(N) to run FOO on a sentence of l
ength N, how long would it
take to run (map foo paragraph)?

Answer: Theta(N*t(N))

    This was about noticing the pattern from the pr
evious two parts. In both,
    you're given a procedure whose running time you
 know, and basically asked
    "if I run this N times (one for each sentence i
n the paragraph), how long
    will it take?". Presented this way, it's clear
that running something that
    takes time proportional to t(n), N times, will
take Theta(N*t(n)) time.
    In the first problem, t(n) was just N (or Theta
(N), if you prefer);
    in the second, t(n) was Theta(N^2).

5. Recursion vs. Iteration (SQUARES)

The given procedure is recursive, because after eac
h recursive call returns the
SENTENCE procedure call combines the results. There
fore, before the last
recursive call returns, all this work that combines
 the results is stored and

its execution delayed. This type of recursion forms a triangle shape outwards.
On the other hand, a recursive procedure that generates an iterative process should have a flat stack. In other words, no work is waiting on the recursive call to return.

1 point for checking "recursive" for the given procedure.

The standard answer for writing this procedure iteratively looks like:

```
(define (squares sent)
   (define (helper sent result)
      (if (empty? sent)
          result
          (helper (bf sent) (se sent (square (first sent))))))
   (helper sent '()))
```

This part of the question is out of 3 points. There are couple minor mistakes that could cause 1 point. Many students wrote "(se (square (first sent)) sent)". Note that this approach reverses the order of the result, since the later squares will appear at the beginning of the sentence. Other mistakes includes forgetting to square the first of the sentence, returning an empty sentence instead of result as a base case, DAV for using list procedures, etc. Any procedure that generated a recursive process is given a 0.

3 for perfect solution
2 for minor trivial mistakes, including:
- reversed results
- forgot to square results

- DAV (using list procedures instead of sentence pr
ocedures)
- returned list instead of sentence
- forgot to bf the original sentence
- forgot recursive call

1 for more serious mistakes, including:
- returning only the last element because they didn
't combine the results
  correctly
- combinations of minor trivial mistakes.

0 for implementing square as a procedure (or part o
f the procedure) that
generates a recursive process.

6. Higher Order Procedures (SEPARATE)

There were really two parts to this question. The f
irst part was recognizing
that we are working with lists, meaning that we nee
d to use list procedures like
MAP, CAR, CDR. The second part was recognizing that
, given a sublist, we can get
the name by doing (car sublist) and the age by doin
g (cadr sublist).

The most straight-forward approach is to first get
a list of all the names, then
append that to a list of all the ages:

```
(define (separate lst)
  (append (map car lst) (map cadr lst)))
```

A fairly common mistake was to use sentence instead
 of append, and every instead
of map. Remember, sentence/every are for words/sent
ences, while append/map are
for lists! The same also applies to first/butfirst
versus car/cdr - the former

are for words/sentences, while the latter are for l
ists.

Another fairly common mistake was to use (map cdr l
st) instead of
(map cadr lst). Remember - the cdr of a list return
s another list. We want
(map cadr lst), because (cadr '(ellie 77)) will act
ually return the number 77,
rather than the list '(77).

Several people tried solutions using both MAP and F
ILTER - however, most
attempts weren't successful. The reason why using f
ilter and map was tricky is
because both procedures return lists. The following
 "solution" almost does what
we want:

```
(define (separate lst)
  (append (map (lambda (sublist)
                 (filter (lambda (elem) (not (numbe
r? elem))) sublist))
               lst)
          (map (lambda (sublist)
                 (filter number? sublist))
               lst)))
```

Unfortunately, the result we get would look like:
((kevin) (frederickson) (dug) (ellie) (24) (76) (7)
 (77))
A fix to this would be to then flatten this result
- an application of
accumulate would do nicely:

```
(define (separate lst)
  (accumulate append
              '()
              (append (map (lambda (sublist)
                             (filter (lambda (elem)

                                       (not (number
```

```
? elem)))
                                            sublist))
                                lst)
                          (map (lambda (sublist)
                                  (filter number? sublis
t))
                                lst))))
```

This is lengthy, however - in short, take advantage of patterns in the data when you see them!

4 for a correct solution
3 for small mistakes like:
- syntax errors
- Data Abstraction Violations (using sentence instead of append, etc)

2 for both a small mistake and a more serious mistake, like:
- using cdr instead of cadr

1 for a correct recursive solution (you had to use higher order procedures!), or
  an unsuccessful use of map+filter.
0 for anything else

Basically, you got one point for having append, map, car, and cadr in the solution. Then, any mistakes deduct from that.

7. Recursion (DOUBLES)

This was a pretty straightforward question; it was just all of the cases that made a problem. Basically, we have to think about the base case (the empty sentence), the recursive case where the first word is a number, and the case where the first word is not a number. Each of these cases is pretty straightforward on its own, so let's put them toget her.

```
(define (doubles sent)
  (cond ((empty? sent) '())
        ((number? (first sent))
         (se (* 2 (first sent))
             (doubles (bf sent)) ))
        (else (se (word (first sent) (first sent))
                  (doubles (bf sent)) ))))
```

For this problem we were very lenient. In particula
r, if you wrote this for the
last case:

```
(se (first sent) (first sent) (double (bf sent)))
```

i.e. forgetting to WORD together the first word to
double it, we only took off
one point. Similarly, if you had this in a helper p
rocedure:

```
((not (number? x)) (word 'x 'x))
```

we also only took off one point. Those quotes reall
y shouldn't be there; what
you'll get back is the word XX. But this is the sor
t of mistake we assumed you'd
be able to catch very quickly if you had an actual
computer. (We probably won't
allow this next time!)

A couple of solutions used WORD?, or their own ALPH
ABET? procedure, to determine
whether a given word is a number or not. The former
won't work, since numbers
are words, and have been since day 1. The latter do
esn't handle words that
/aren't/ made up of alphabet letters, like the excl
amation point in the example.
It also won't handle odd words like NUMB3RS, which
have digits in them but
aren't numbers overall.

6 for a correct solution
5 trivial mistakes, including any of:
- forgetting WORD in the non-number case (described above)
- extra quotes when doubling in the non-number case (described above)
- data abstraction violations (using list procedures on sentences)
3 "The Idea", including ALL of:
- a base case (even if not quite correct)
- using BUTFIRST to go through the input sentence
- trying to separate the number and non-number cases
- using SENTENCE to combine the current element and the recursive result
1 "An Idea", including only some of the "The Idea" criteria
0 for anything else

8. Lost on the Moon...again? (RECOVER)

The easiest way to do this one was to write a separate helper procedure to deal with a single word at a time. This lets us not worry about the whole sentence at all, and in fact we can just use a single higher-order procedure to handle this.

```
(define (fix wd)
  (if (number? wd)
      wd
      (keep (lambda (letter) (not (number? letter)))
            wd)))
```

Many people weren't sure what would happen if we used KEEP on a word. It actually does do what you want. (EVERY, on the other hand, will return a

sentence even if you give it a word as input.) So i
t was also possible to write
your own version of KEEP:

```
(define (throw-out-numbers wd)
  (cond ((empty? wd) "")
        ((number? (first wd)) (throw-out-numbers (b
f wd)))
        (else (word (first wd) (throw-out-numbers (
bf wd)))) ))

(define (fix wd)
  (if (number? wd)
      wd
      (throw-out-numbers wd) ))
```

The tricky part here was actually knowing what the
empty word was, since it only
comes up in passing. The "correct" way to write it
is as an empty /string/,
using double quotes. Many solutions also took advan
tage of the fact that we just
asked if WD was empty, so we can just return WD. Ot
her solutions tried to return
the empty sentence (); despite being a domain/range
 problem, we only took off 1
point for this. (A particularly clever solution was
 (bf 'a), which of course
will be the empty word.)

Anyway! Now that we have FIX, we can just apply it
to every word in the
sentence. RECOVER is straightforward:

```
(define (recover sent)
  (every fix sent) )
```

6 for a correct solution
5 trivial mistakes, including any of the following:

- reversing the result because you wrote an iterati
ve version of RECOVER

- data abstraction violations (using list procedure
s on sentences)
- doing something incorrect in trying to write the
empty word
4 returning a sentence from FIX (or equivalent) ins
tead of a word
3 "The Idea", including ALL of the following:
- trying to separate the numbers from the non-numbe
rs
- for a mixed word, trying to keep only non-numbers

- fixing every word in the input sentence
1 "An Idea", including some but not all of the "The
 Idea" criteria
0 for anything else