GROUP QUESTION: Environment Diagrams

(define x 10)

adds the binding x=10 to the global frame G.

(define (foo x) ...)

creates procedure P1 with
        params: (x)
        body: (if ...)
        env: G
and then creates the binding foo=P1 in the global frame G. (By the way, I'm writing "foo=P1" because I'm doing this in text format. When drawing an actual diagram, you want an arrow from the name foo to the procedure bubbles! No need to give procedures names like P1 at all.)

Nothing else happens during this definition -- the body of P1 is not evaluated at this time.

(foo 4)

creates frame E1 with the binding x=4, extending G.

With E1 as the current environment, evaluate the body, which is the IF expression. Since x=4 in this environment, the OR is false, so we evaluate the LET. This creates a procedure P2:
        params: (y z)
        body: (set! ...) ...
        env: E1
and then immediately invokes the procedure. First it has to evaluate the argument expressions 8 and (/ X 2). The first is self-evaluating. For the second, we find the binding x=4 in current environment E1, and divide that by 2, giving 2. (/ is a primitive, so there's no need to add anything to the diagram for this call.) So now we make a new frame E2 with bindings y=8, z=2, extending E1 (because that's where the right bubble of P2 points).

With E2 as current environment, evaluate the body of the LET, which consists of three expressions:

(set! y (lambda (z) (* z z)))

The LAMBDA expression creates procedure P3:
        params: (z)
        body: (* z z)
        env: E2
and then changes the binding of y in E2 from 8 to that procedure (P3). It does not call the procedure! So it does not compute (* z z)!

(set! x 14)

Since the current environment is E2, and there is no binding for x in that frame, we look at the frame it extends, namely E1. There we do find a binding x=4, and we replace the 4 with 14. We don't change the global binding x=10!

(foo z)

We look for bindings for FOO (value P1, found in G) and Z (value 2, found in E2), then call P1 with argument 2. This creates a new frame E3, with x=2, **extending G** because that's where the right bubble of P1 points.

With E3 as current environment, we evaluate the body of P1, the IF expression. This time, since x=2, the OR returns true, and the procedure returns the value of x, which is 2.

Scoring: Almost all the groups got this right. The only really common mistake was to use dynamic scope, in which E3 extends E2. This got 3 points.

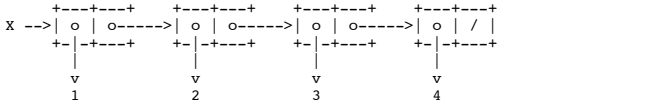
5 correct.
3 dynamic scope; slight right-bubble error (e.g. P2 -> E2).
2 change global x to 14; really bad right-bubble error; change y to 4.
1 multiple right-bubble errors; new frame for SET!.
0 really bad.

There were a handful of unique errors, most getting 1 or 2 points.

QUESTION 1: What Will Scheme Print?

```
> (let ((x (list 1 2 3 4)))
    (set-cdr! (cddr x) (car x))
    x)
```

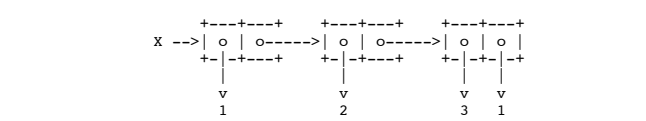
Let's start by drawing the original X:



Next we find (cddr x); that's the pair whose car is 3.

Now we find (car x); that's just the number 1.

Finally, we set the cdr of that third pair to 1.



And write the answer, a three-element improper list whose last cdr is 1:

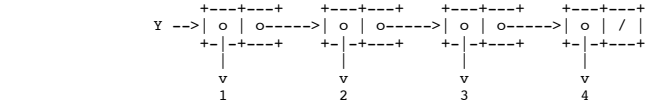
```
(1 2 3 . 1)
```

The most common mistake in the diagram was to make the last cdr point to the entire first pair, creating a circular list. The most common mistake in the printed output was forgetting the dot before the final cdr, followed by adding an extra set of parentheses: (1 2 (3 . 1)). (This would be a three-element list whose third element is the pair (3 . 1).)

If you drew the first car and the last cdr pointing to the same 1, that was also marked as correct. Remember, all words that print the same are automatically EQ?, so it doesn't matter so much if we write them twice.

```
> (let ((y (list 1 2 3 4)))
    (set-car! (cddr y) (cddddr y))
    y)
```

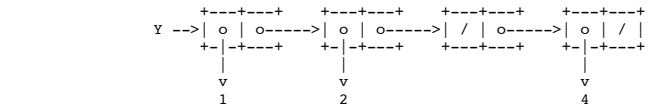
The original Y is the same as the original X:



Again, we find (cddr y), which is the pair whose car is 3.

The value we're using is (cddddr y). This tripped a few people up, but let's see how it works:
   Y is the pair whose car is 1.
   (cdr y) is the pair whose car is 2.
   (cdr (cdr y)) is the pair whose car is 3.
   (cdr (cdr (cdr y))) is the pair whose car is 4.
   (cdr (cdr (cdr (cdr y)))) is the empty list.

So we're setting the CAR of the third pair to the empty list:



And our printed form is still a four-element list:

```
(1 2 () 4)
```

By /far/ the most common problem was not knowing how to represent the empty list in the box-and-pointer diagram. We accepted an arrow pointing to a single box with a slash, as well as a pair of parentheses, but the way shown above is correct (and makes sense; why should the diagram treat cars and cdrs differently?).

The most common /wrong/ answer for this was to try to show an arrow that pointed to "the cdr of the fourth pair". But you can't point to half a pair!

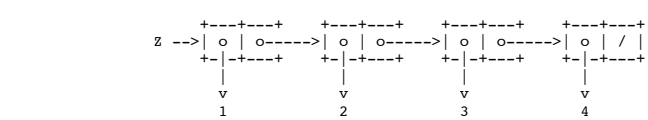
Another common mistake was to write the empty list as '(). But that would be an /expression/, and a list is made up of /values/. Writing '() in a box-and-pointer diagram where you mean "empty list" is like writing (+ 2 3) where you mean 5. Hopefully that will help you keep them straight!

The two most common mistakes in the printed form were forgetting the empty list (1 2 4) and thinking that the list had to end there (1 2). Quoting the empty list here (1 2 '() 4) was also incorrect; STk /never/ prints single quotes!*

*except if they're part of a string, like in proj3.

```
> (let ((z (list 1 2 3 4)))
    (set-car! (cddr z) z)
    z)
```

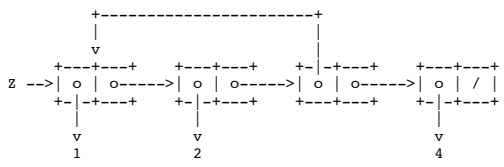
Once again, we start with our plain four-element list:



Once again, we're still looking at the pair whose car is 3.

This time, the second argument is Z itself, which means the pair whose car is 1. Remember, when you deal with mutation, you work at the level of the arrows in the diagram; Z is just that pair, not the whole list.

And we're setting the CAR of the third pair to the first pair:



This creates a self-referential list, which means when you try to print it
you're going to have problems...!

    (1 2 (1 2 (1 2 ...

This one was tricky to grade for the print form. The intent of the instructions
was for you to recognize the loop and just write "Error" or "Loop"; both of
these were considered correct even though the /code/ did not contain a loop.
However, if you attempted to write what STk would actually print, you had to get
it right! That meant
 - open parentheses in the proper place, and
 - no 4s or closing parentheses

The most common mistake in the diagram was to point to Z instead of to the first
pair. Remember, Z is a variable in an environment frame; it's not a value! If
we changed Z later, it wouldn't change this car. *Arrows only point to values.*

The most common mistake in the print form was to forget that since the loop
happened in the car, STk would print an open paren. This gave answers like this:
(1 2 1 2 1 2 ...

Scoring:
+1pt if the diagram is correct
+1pt if the written form is correct
+1pt if neither are correct but the two match (unless you wrote "Error")

-1pt if you forgot the start arrow (once for the entire problem)

QUESTION 2: Streams

```
> (define (madness x y)
    (if (even? x)
        (- x y)
        y))
> (define boss (cons-stream 1 (stream-map madness boss truck)))
> (define truck (cons-stream 3 (stream-map + boss truck)))
```

The easiest way to do this question was by columns:

```
    boss   ???  ???  ???  ???  ???
    truck  ???  ???  ???  ???  ???
```

First fill in the elements that were given:

```
    boss    1   ???  ???  ???  ???
    truck   3   ???  ???  ???  ???
```

Next, take the sum of the elements to get the next elemet of TRUCK. For BOSS,
see if the first element is even (it's not), and then just take the first
element of TRUCK.

```
    boss    1    3   ???  ???  ???
    truck   3    4   ???  ???  ???
```

Repeat.

```
    boss    1    3    4   ???  ???
    truck   3    4    7   ???  ???
```

Here's the tricky case: the current element of BOSS is now even, so we subtract
the current element of TRUCK. (We still add the two to get the next element of
TRUCK.)

```
    boss    1    3    4   -3   ???
    truck   3    4    7   11   ???
```

-3 is odd, so we just go back to what we were doing before.

```
    boss    1    3    4   -3   11
    truck   3    4    7   11    8
```

Most people got this question correct.

Scoring:
+0.5pt for getting the first element of each stream
+1pt for getting the -3 (the only time the subtraction applied)
+0.5pt for every other element.

As a special case, if you forgot to include the first two elements, but got
everything else, you got 4 points.

QUESTION 3: OOP, Below the Line

As soon as we define make-thingo, we recognize that make-thingo is bound to the

```
procedure
        (lambda (baz) (let .... (else (flopp zot))))))))

Thus, if we were to try:
        (define my-thing (make-thingo 'sally))
we should observe that my-thing has baz bound to sally.  Thus, baz is similar to
an instantiation variable (E).

Also, since we know that make-thingo is bound inside the environment created by
(let (foo 'yakko)) ...), then we know that my-thing has access to foo at the
time of instantiation, so foo is analogous to a class variable (B).

As soon as my-thing is defined, garply gets bound to wakko, so garply is an
instance variable (D).

floop must be the parent (A) of a make-thingo object, since floop is bound to a
call to make-bear and we see that at the bottom of the cond, we call
(floop zot), which is analogous to passing a message to a parent object if the object doesn't have a method to handle the message.

Finally, if we were to try (my-thing 'yes) or (my-thing 'no), zot will get bound
to this message, so zot represents the message (F).

Final answers:
foo:    B
baz:    E
garply: D
floop:  A
zot:    F
xyzzy:  C

Grading: Half point for each correct answer.



QUESTION 4: List Mutation

An important thing to note for this question was that there is no way to
write duplicate-elements! without creating new pairs. For instance, a three
element list only contains three pairs, but duplicate-elements! needs to
mutate the list into a six element list, which contains six pairs. This means
that, unlike most mutation questions, you were actually /required/ to use
cons for this question.

Probably the simplest way to do this question was to create a new pair with
the same car and cdr as the first pair in this list. This is as simple as
calling (cons (car lst) (cdr lst)). From there, it's just a matter of making
the appropriate call to set-cdr! to add this newly created list into the
list structure, and recursing:

(define (duplicate-elements! lst)
  (if (null? lst)
      â€¯()
      (let ((new-pair (cons (car lst) (cdr lst))))
        (set-cdr! lst new-pair)
        (duplicate-elements! (cdr new-pair)))))

Another clever solution that we saw fairly often was one that used
interleave!. If you can make a copy of the original list, then you can use
interleave! to create a single list containing each element twice. By far
the easiest way to copy a list was to just use append:

(define (duplicate-elements! lst)
  (interleave! lst (append lst â€¯())))

There were a surprising number of solutions that involved set-car!. Since
we were creating a flat list structure in this problem, there was no reason
to call set-car!, and many attempts to do so resulted in nested list
structure.

Scoring:
6: perfect

5: trivial mistakes
- Using the return value of duplicate-elements! in a set-cdr! without
  making duplicate-elements! return anything, as long as everything else was
  right
- Forgetting to handle null lists as input
- Calling list/append instead of cons
- Creating the right structure, but in the wrong place (i.e. the original
  argument no longer points to the start of the list)
- Attempting to do more than the question asked, but all parts involving the
  problem statement are correct

4: the idea
- Making recursive calls on the cdr rather than the cddr (which usually
  resulted in an infinite loop)
- Calling set-cdr! on the wrong argument (such as calling (set-cdr! (cdr ls)
  â€¦) instead of (set-cdr! lst ...) )
- Recursing on the cadre instead of the cddr, but only once
- Using the return value of set-cdr! in another call to set-cdr! (most
  students who did this expected it to return the list rather than just okay)

3: messing up the idea
- Correctly generates most of the list, but creates a circular structure in
  the base case
- Attempting to add one pair to the front of the list using set!, but
  otherwise duplicating every other element correctly (so that the final
  result is all elements duplicated except the first one)

2: an idea
- Uses (car lst) as if it were the first pair of the list
```

```
- Creates new pairs and attempts to set-cdr! them, but has problems in doing
  so (such as creating circular lists or overwriting necessary values)
- Generates nesting structures (e.g. deep lists or a list of pairs)
- Correctly duplicates the first element without overwriting any other
  elements, but forgets to recurse

1: messing up an idea
- Putting two copies of the list back to back, rather than duplicating
  individual elements (such as (1 2 3 1 2 3) rather than (1 1 2 2 3 3) )
- Failing to actually change the list, but still calling set-cdr! and using
  recursion
- Not creating new pairs, but overall still attempting to duplicate elements

We gave no points to solutions that didn't keep the original pairs.



QUESTION 5: Metacircular Evaluator

This question seemed to scare a lot of students. The biggest hurdle was figuring
out what needed to change. Recall that we're trying to make both of these work:

>> (define new-odds (lambda nums (filter odd? nums)))
>> (new-odds 1 2 3 4 5)
(1 3 5)

>> (define old-odds (lambda (nums) (filter odd? Nums)))
>> (old-odds '(1 2 3 4 5))
(1 3 5)

Conceptually, we want to bind the formal parameter nums to a list of arguments.
When are formal parameters bound to arguments? In extend-environment. So, the
only necessary changes will be in extend-environment.

; old extend-environment
(define (extend-environment vars vals base-env)
  (if (= (length vars) (length vals))
      (cons (make-frame vars vals) base-env)
      (if (< length vars) (length vals))
          (error "Too many arguments supplied" vars vals)
          (error "Too few arguments supplied" vars vals))))

Previously, extend-environment expected both vars (formal parameters) and vals (arguments) to be lists. For instance, in the (old-odds '(1 2 3 4

(extend-environment (nums) ((1 2 3 4 5)) <global-environment>)

Then, we would create a new frame with nums bound to the list (1 2 3 4 5):
        (cons (make-frame (nums) ((1 2 3 4 5))) base-env)

Now, with a call like (new-odds 1 2 3 4 5), we still want nums to be bound to the list (1 2 3 4 5). However, what's different is now nums isn't

(extend-environment nums (1 2 3 4 5) <global-environment>)

So, now we have to check to see if the formal parameter is an atom â€“ if it is, then we're dealing with the "new" lambda, and we should do some

(define (extend-environment vars vals base-env)
  (if (list? vars)
      (cons (make-frame vars vals) base-env)                    ; same old
      (cons (make-frame (list vars) (list vals)) base-env)))    ; new!

This was the only required change. Note that if you didn't remember to wrap the vals in a list, then what ends up happening is that nums becomes

There were other ways to solve this problem, such as modifying mc-apply or
changing both extend-environment and make-frame, but this is one of the most
compact solutions.

Rubric:
+ 1 point: if the old way still works
  - "old way" being the usual (lambda (nums) (filter odd? nums))
+ 2 points: checking if vars (formal parameters) is a list
  - 1 point if you checked vals instead of vars
  - 1 point if you had the check inside make-frame, but didn't modify
       extend-environment; because the original extend-environment has the
       error-checking for the lengths of vars and vals, extend-environment
       would throw an error before ever getting to make-frame.
+ 2 points: If you list-ed both vars and vals
  - 1 point if you only list-ed vars



QUESTION 6: Vectors

For part A, we looked for something around a one-liner. Any of these would work:

(define (vc-extend! last-in-chain)
  (vector-set! last-in-chain 10 (make-vector 10))
  (vector-set! (vector-ref last-in-chain 10) 10 #f))

(define (vc-extend! last-in-chain)
  (vector-set! last-in-chain 10 (make-vector 10 #f)))

(define (vc-extend! last-in-chain)
  (vector-set! last-in-chain 10 (make-vector-chain)))

Ways to earn points:
+1 Make a vector of size 11
+1 Set the last element to #f
+1 Add the new vector to last-in-chain

We docked a point if you were off by one, or forgot that the return value was
supposed to be OKAY.
```

For part B, we looked for something like this:

```
(define (vc-set! vc index value)
  (if (< index 10)
      (vector-set! vc index value)
      (begin (if (not (vector-ref vc 10))
                 (vc-extend! vc) )
             (vc-set! (vector-ref vc 10) (- index 10) value) )))
```

We awarded points if you either explicitly or implicitly did the following:
+1 tested if the index is less than 10
+1 vector-set! the element properly
+1 checked if the 11th element is false
+1 properly extended the chain if it is false
+1 made recursive calls as needed
+1 remembered to subtract 10 from the index for recursive calls

You lost points if you
-1 assumed that vc-extend! returns a useful value
-1 assumed that vc-ref will return #f for an index out of bounds

If you called vc-extend! whether or not it was necessary, you lost both the "checking" point and the "properly extending" point.