WEEK 9 STUDIO

AGENDA

- Mutable variables
- Environment model
- Studio Questions

VARIABLES: IMMUTABLE VS MUTABLE

Think of variables as containers that holds a value inside. This value can be anything.

```
const a = 1;
const b = "a string";
const c = true;
const d = pair(1, null);
const e = [1, 2, 3];
```

VARIABLES: IMMUTABLE VS MUTABLE

However, immutable variables such as `const` can only hold one value. This variable cannot be reassgined to another value.

```
const a = 1;
a = true; // error
```

VARIABLES: IMMUTABLE VS MUTABLE

Mutable variables such as `let` on the other hand, can be reassigned to another value.

```
let a = 1;
a = 2; // ok
a = true; //ok
a = [1, 2, 3]; //ok
a = pair(1, null); //ok
a; // pair(1, null)
```

Function parameters are now mutable variables as well

WHAT IS HAPPENING HERE?

```
const a = [];
a[1] = true; // is this allowed?
a = [1, 2]; // is this allowed?
```

WHAT IS HAPPENING HERE?

```
let a = [];
a[1] = true; // is this allowed?
a = [1, 2]; // is this allowed?
```

KEY POINTS FOR ENVIRONMENT MODEL

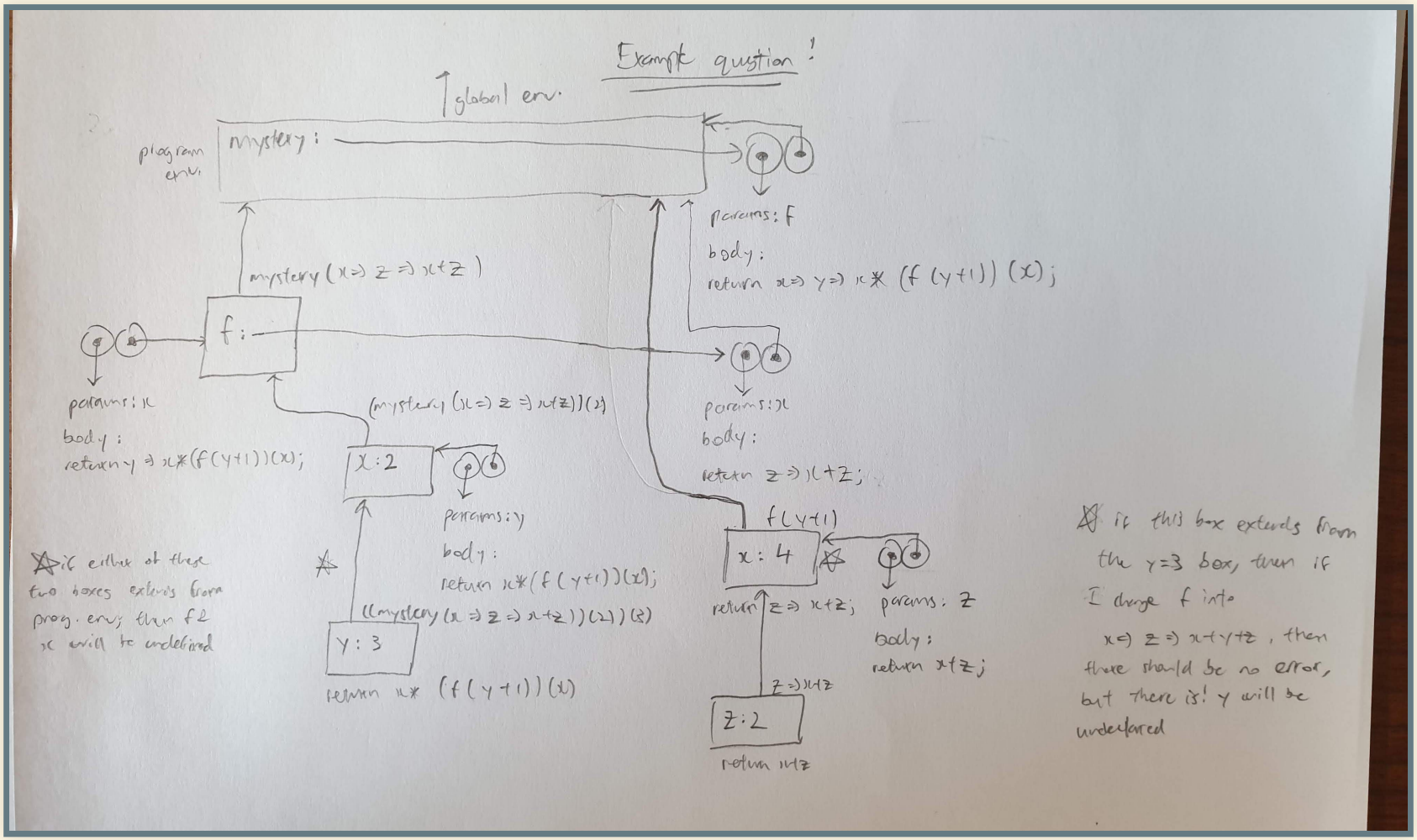
- If else blocks are **also considered blocks** - need to draw a new frame for the block
- Primitive values are written directly next to the name. Do not draw an arrow pointing to a primitive value!
- For the function object, draw out the dots more clearly
- Don't need to draw the frame for primitive functions.
- Refer to this to find out which functions are primitive and which are not.
- READ LECTURE SLIDES 8, ALL THE RULES ARE THERE!

ENVIRONMENT MODEL

```
function mystery(f) {
    return x => y => x * (f(y + 1))(x);
}

const m = ((mystery(x => z => x + z))(2))(3);
m;
```

ENVIRONMENT MODEL

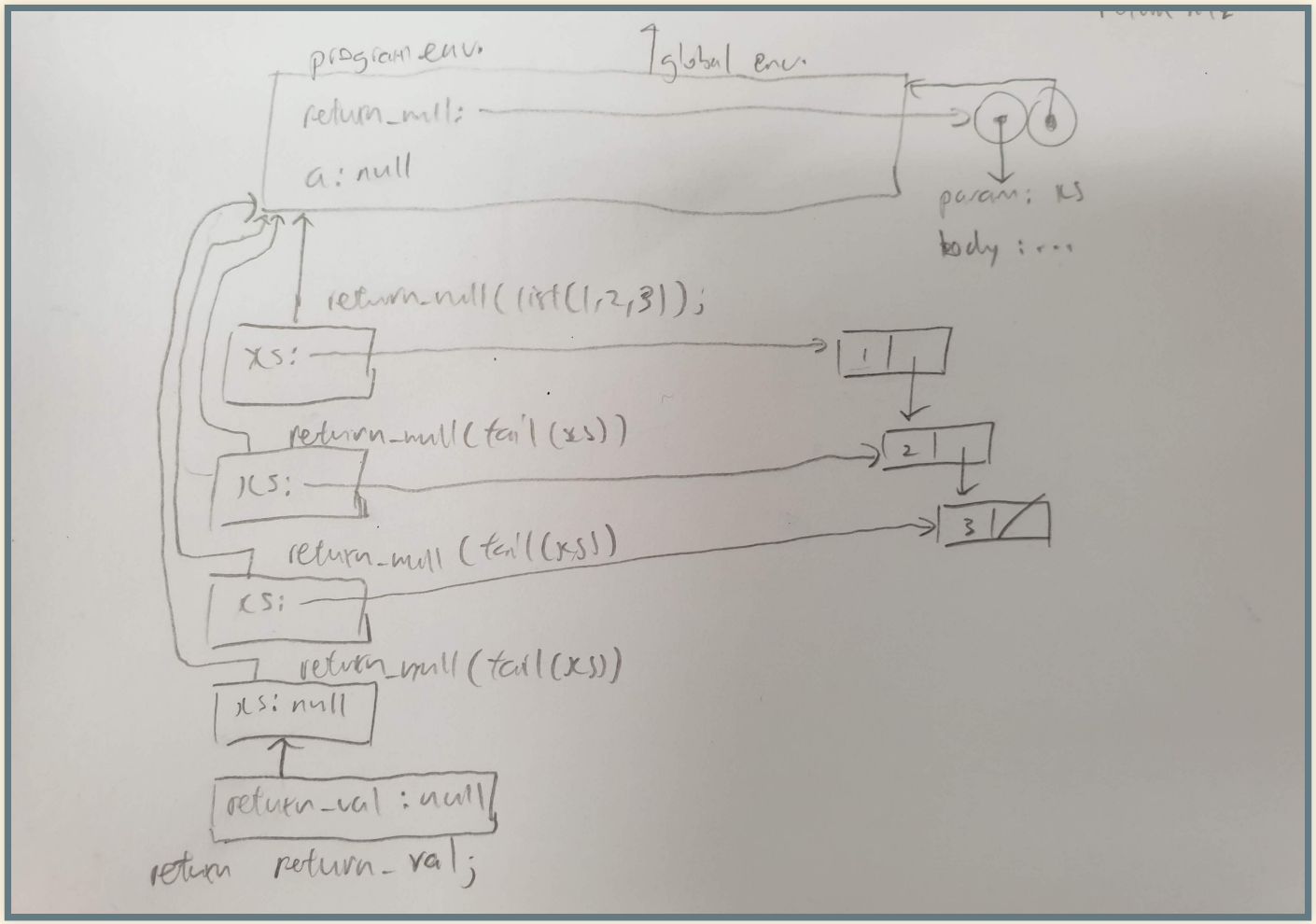


ENVIRONMENT MODEL

```
function return_null(xs) {
    if (is_null(xs)) {
        const return_val = xs;
        return return_val;
    } else {
        return return_null(tail(xs));
    }
}

const a = return_null(list(1, 2, 3));
a;
```

ENVIRONMENT MODEL



ATTENDANCE

STUDIO

QUESTION 2

Version 1 is given in the mission

```
// Version 2
function d_reverse(xs) {
    function helper(result, ys) {
        if (is_null(ys)) {
            return result;
        } else {
            const rest = tail(ys);
            set_tail(ys, result);
            return helper(ys, rest);
        }
    }
    return helper(null, xs);
}
```

QUESTION 3

3. **[In-class only]** The task for this question will be given by your Avenger during your Studio session.

   Write the function `d_filter` that takes as arguments a one-argument predicate function `pred` and a list `xs`, and returns a list that contains only those elements for which `pred` returns `true`. Your function must not create any new pair, and the result list must only be made of existing pairs in `xs`. Your function must not modify the head of any of the existing pairs.

```
function d_filter(pred, xs) {
    // ???
}
```

   Example call:

```
const L = list(1, 2, 3, 4, 5, 6, 7, 8, 9, 11);
d_filter(x => x % 2 === 0, L);  // returns [2, [4, [6, [8, null]]]]
L;  // What is L now?
```

QUESTION 3

```
function d_filter(pred, xs) {
    if (is_null(xs)) {
        return xs;
    } else if (pred(head(xs))) {
        set_tail(xs, d_filter(pred, tail(xs)));
        return xs;
    } else {
        return d_filter(pred, tail(xs));
    }
}
```

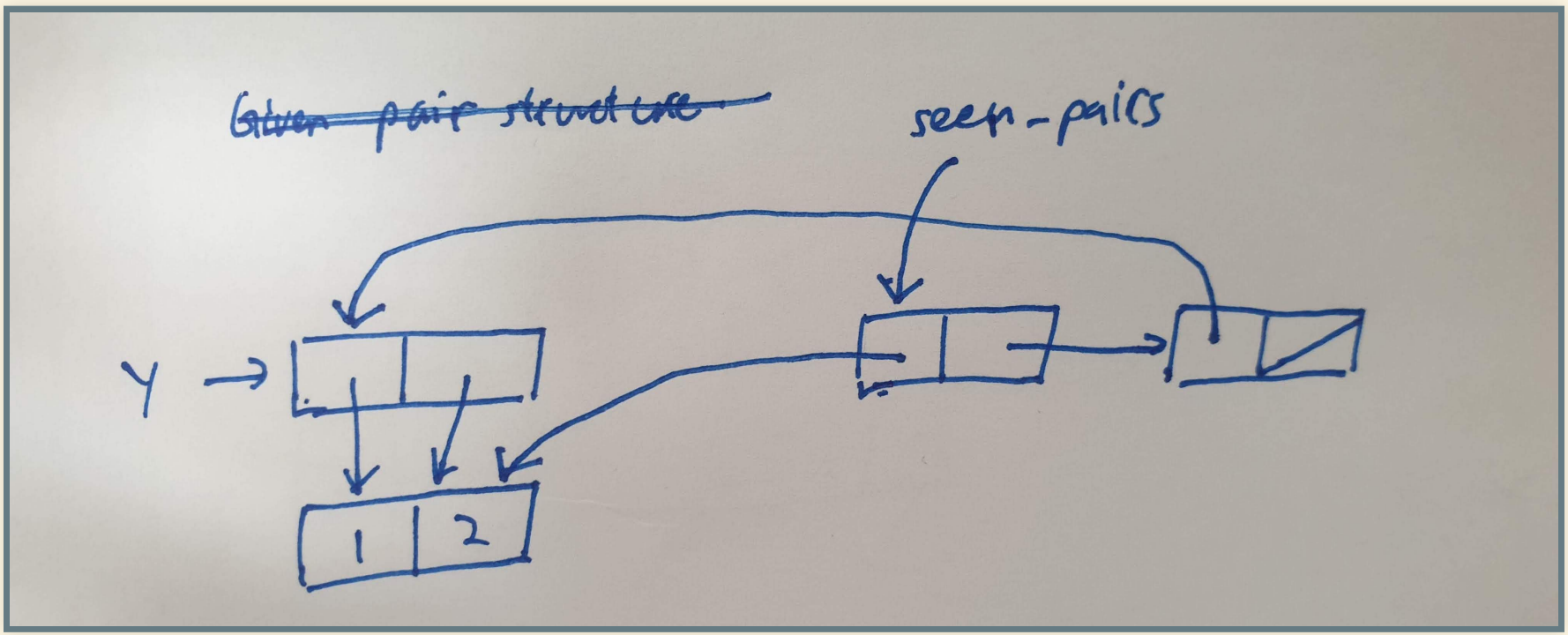
QUESTION 4

- Idea for count pairs is to keep track of the pairs that we have seen so far in a list, and then the length of this list would give us the number of unique pairs in the given pair structure.
- Each time you traverse down the pair structure through `head` and `tail`, if you see a new pair, store it in the list.
- This means that we also need something to check if we have seen the current pair before
- `member` sounds like a good choice for this since we have already stored seen pairs in a list.

Annotated solutions

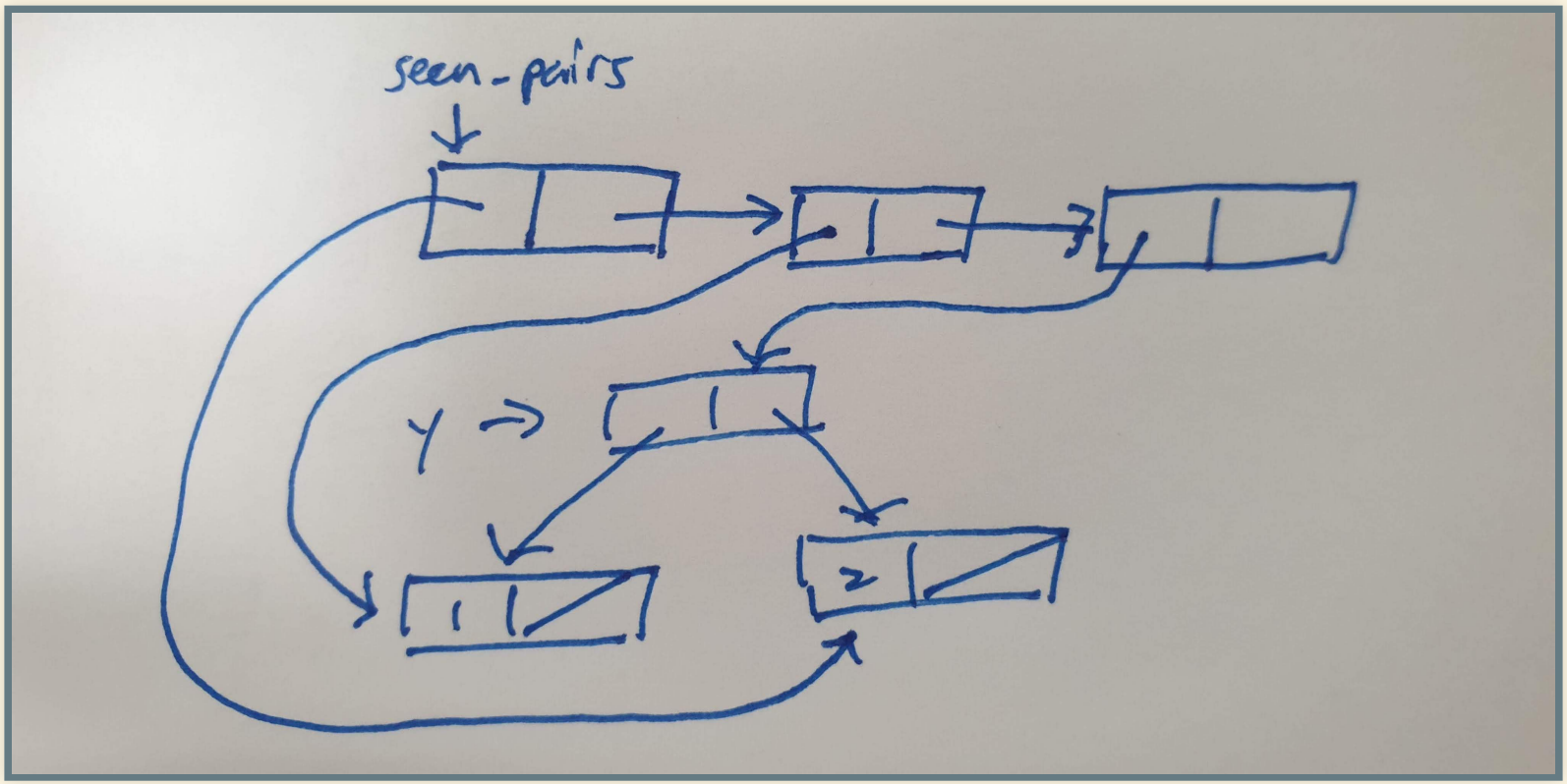
QUESTION 4

If `y` looks like this, this is what `seen_pairs` look like:



QUESTION 4

If `y` looks like this, this is what `seen_pairs` look like:



FILTER GOING MENTAL

Draw the environment model for the following program:

```
function filter(pred, xs) {
    return is_null(xs)
        ? xs
        : pred(head(xs))
            ? pair(head(xs),
                   filter(pred, tail(xs)))
            : filter(pred, tail(xs));
}

filter(x => x % 2 === 0, list(1, 2));
```