A.A. Yes, the `helper` function is tail-recursive. If a call to `helper` makes a recursive call to `helper`, then the return value of the first call is simply the return value of the recursive call. No additional work is done after the recursive call. That's tail recursion.

A.B. (Instead of drawing the frames, I will describe them verbally.)

First, there is the global frame, which includes values for the symbols `null?`, `cdr`, `+`, `helper`, and `list-length`.

Second, there is a frame for the call to `list-length`. Its parent is the global frame. It contains the symbol `lyst` bound to the value `(1 2 3)`.

Third, there is one (and only one) frame for calls to `helper`. (If Guile didn't optimize tail recursion, then there would be four frames for calls to `helper`.) Its parent is the global frame. It contains values for the symbols `lyst` and `n`. On the first call, the values are `(1 2 3)` and `0`. On the second call, they get replaced by `(2 3)` and `1`. On the third call, they get replaced by `(3)` and `2`. On the fourth call, they get replaced by `()` and `3`.

B.A. I'm mainly interested in this definition of the `struct` that describes instances of `Array`:

```
typedef struct Array Array;
struct Array {
    Object super;
    int length;
    void **data;
};
```

Later, the body of the class builder function explicitly refers to the superclass:

```
    ArrayClass.superclass = &ObjectClass;
```

B.B. Here are my four lines of code. (Bonus question: Why should the `retain` be done before the `release`, as opposed to the other way around?)

```
    obj^retain();
    void *old = array->data[i->value];
    old^release();
    array->data[i->value] = obj;
```

C.A. The `new` function does three things before calling `initialize`. First, it allocates memory to hold the instance. Second, it sets the instance's class pointer. Third, it initalizes the instance's reference count to 1.

C.B. A method such as `initialize` can be called only on an object that knows its class. So the allocation and setting of the class pointer must happen before `initialize`. Initializing the reference count could happen in `initialize`, but it makes slightly more sense to have it happen in `new`, where the rest of `Object`'s state is initialized, and so that subclassers don't have to remember to do it.

D. In our OOC code, the `Integer`, `Double`, and `String` classes all respond to the `add` method. Is it because they all descend from `Object` and `Object` has an `add` method? If so, then that would be an example of polymorphism by inheritance/subtype. But no, `Object` does not have an `add` method. Instead, we send these objects `add` messages just because we happen to know that they can respond to them. That's duck typing.

E.A. It evaluates every subexpression in `(func ...)`, finding that `func` itself evaluates to a closure. It creates a new frame, whose parent is the frame stored in the closure. In that new frame, it binds the formal parameters stored in the closure to the values of the `...` subexpressions. Then it evaluates the body stored in the closure with respect to the new frame. (Unlike E.B, a local frame is created.)

E.B. It evaluates every subexpression in `(+ ...)`, finding that `+` itself evaluates to a primitive function. It invokes that primitive function on the list of values of the `...` subexpressions. (Unlike E.A, no local frame is created.)

E.C. It recognizes that `if` is a keyword. It evaluates the first subexpression after the `if`. If that subexpression is true, then it evaluates the second subexpression after `if` and returns that value; otherwise, it evaluates the third subexpression after `if` (if any) and returns that value. (Unlike E.A and E.B, not all arguments are evaluated.)

F. The code works correctly, but it is slow. The `invoke` function has to search 26 classes back in the inheritance tree, to find the implementation of `retain`. Given how frequently `retain` is going to be called, probably, repeating this search again and again is a huge amount of wasted work. (And the same goes for `release` and `autorelease`.)

G. It seems that CPython is trying to benefit from the advantages of (automatic) reference counting while also working around its one enormous defect. The big advantage of reference counting is that it reclaims resources quickly, with none of the latency spikes typical of garbage collection. Its enormous defect is that it can't resolve cyclical references. So the garbage collector periodically comes through, to clean up unneeded cyclically referenced objects, that the reference counting has failed to reclaim.