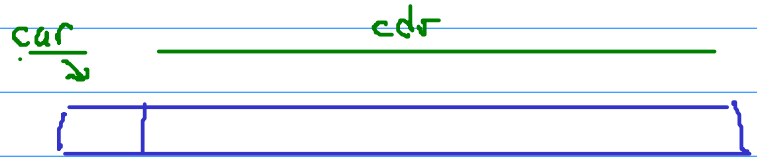
```
;; list ::= () | (cons atom list) | (cons list list)
;;  lat ::= () | (cons atom lat)
```

① We want a function member? which inputs a lat and an atom and which checks whether the atom is included in the lat.

lat

a

(Give a recursive development)



Our Divide & conquer strategy is simply to cdr down the input list:

- we process the car explicitly
- we let the recursive call "magically" process the cdr
- we combine these results to solve the problem

Focus on * - ie - on the recursive call

We expect the recursive call to completely solve the problem for the cdr.

We need to write down precisely what

this means. For this problem:

> The recursive call returns #t when the input atom occurs in the cdr, and #f otherwise.

Since a occurs in inlat precisely when it is either (car inlat) or it occurs in (cdr inlat), we see how to write the code

```
(define (member? a inlat)
  (cond ((null? inlat) #f)
        (else (or (eq? a (car inlat))
                  (member? a (cdr inlat))))))
```

What now does the inductive pf look like?

IH: recursive call works if a is an atom and (cdr inlat) is a lat

Of course the pgm hasn't changed a — so a is an atom here if it was originally.

And we know from the BNF data description that (cdr lat) is a lat. (Note that because inlat is not empty, we know as well that the

application of cdr will not throw an error.)

we already gave the basis and induction steps already.

Aside

Is this really recursive?

Syntactically, it is recursive, with or serving as the guard. But in reality - given the standard left-to-right short-circuit implementation of or - an efficient compiler would likely cast it as a tail recursion.

Consider (or x y z) → returns #t if x is #t, with no need to eval y or z. And if x is #f, it simply moves on to y — no need to remember the value of x

But if or happens not to be implemented this way - say if it evals right to left — or could indeed use the stack.

(2) Next let's consider the problem of removing an element from a lat.

a

lat

cdr-down as our divide & conquer strategy

termination is implicit in cdr-down

What do we do in this case to process the car?

Two cases: either (car lat) is the element to be removed, or it's not.

If it is - Then ... well, wait - the spec seems incomplete! Are we to
(e) remove every occurrence of a, or just the (f) first occurrence of a?

For (e), we need to remove all occurrences from the cdr as well as removing the car.

For (f), we could just return the cdr

If it is not, then we have to retain it.

So it seems we have a more or less complete design idea: pseudo-code, termination, etc.

Lets have draft code for (e)

```
(define (rember* a lat)
  (cond ((null? lat) '())
        (else (if (not (eq? a (car lat)))
                  (cons (car lat)
                        (rember* a (cdr lat)))

                  (rember* a (cdr lat)))
        )))
```

I am content to talk through the induction argument – so I can emphasize the tentative nature of the draft code. The role of the induction is to serve as a check on the code you currently have.

If an error is exposed, then one revisits

the earlier steps. Another way of exposing errors is to run a few tests. In fact, it makes sense to run the tests BEFORE taking the time to write out the induction.

There's another flaw with the spec! It doesn't say...

> why not just return the empty list each time? certainly that would strip out all occurrences of a!

that the non-a elements have to be retained in the same order they were originally.

What about option f?

```
(define (rember a lat)
  (cond ((null? lat) '())
        (else (if (not (eq? (car lat) a))
                  (cons (car lat)
                        (rember a (cdr lat)))
                  (cdr lat)))))
```

The inductive argument must include certification for the claim that this code removes the first occurrence of a, and no others.

Again - I leave for you the important tasks of testing and writing out the induction.