

Announcements

- Exams back on Wednesday.
- HWK 2 is up. Some notes:
 - you **have to** work in pairs.
 - if you get stuck, get help right away.
 - it might be challenging, so make sure you stay on top of it.

Review

Review: Last time we started looking at recursive schemes.

```
;; len: allp -> natp
;; returns the length of a list, i.e., is 0 for atoms
;; else it counts the length of the spine of the list
```

```
;; (assert-event (= (len nil) 0))
;; (assert-event (= (len (list 1 2)) 2))
;; (assert-event (= (len (list 1 2 3)) 3))
;; (assert-event (= (len 5) ??))
```

```
(defun len (x)
  (cond ((atom x) 0)
        (t (+ 1 (len (rest x))))))
```

Today: Recursive Schemes, part 2

Wait. Maybe the domain for len should be true-listp?
What changes are required? Just change the comment describing the type to:

```
;; len: true-listp -> natp
```

But, what design guideline gives us guidance on what to do with non-atomic data that is not of the appropriate type? The analogy with what we have been doing until now would be: create an idiom for testing for an empty list that coerces all elements of the universe to true-lists. Such a function might be:

```
(defun emptyp (l)
  (cond ((true-listp l) 1)
        (t nil)))
```

That's not what we did. For example, what is (len (cons 1 2))? 1. What if we used emptyp instead of atom? 0.

Why is the use of emptyp not advisable? Well, stylistically it looks OK: just introduce a new idiom and replace atom by emptyp. So that does not violate the design principle that I mentioned last time: minimize the violence to the code. But, it does severely change the computational behavior. For example, with atom, what is the complexity of len? linear. What about with emptyp? quadratic. That's horrible.

So, we have:

Design guideline 3:

When defining a function that recurs on a true list, treat the final cdr of the list as nil.

Example: define a function that determines whether e is an

element of a linear list.

Use the design recipe.

```
;; mem: allp true-listp -> booleanp
;; (mem e l) returns t if e is an element of l and nil otherwise

;; (assert-event (= (mem 3 nil) nil))
;; (assert-event (= (mem 2 (list 1 2)) t))
;; (assert-event (= (mem 0 (list 1 2 3)) nil))
;; (assert-event (= (mem 0 (list (list 0 1) (list 1 2)) nil)))
;; (assert-event (= (mem (list 1) (list (list 2) 1 2 (list 1))))))
```

Since we are visiting a linear list, let's use the recursive scheme we've seen:

```
(defun mem (e l)
  (cond ((endp l)
        ...)
        (t ... (first l) ...
              (mem ... (rest l) ...) ...)))
```

```
(defun mem (e l)
  (cond ((endp l) nil)
        ((= e (first l)) t)
        (t (mem e (rest l)))))
```