

Announcements

- Exam 1 graded.
 - Average is in the high 80's.
 - If you got less than an 80, you should be concerned.
 - You should work hard on homeworks and visit office hours.
 - You can see your grades on blackboard.
- Remember HWK2 is up. You have to provide functionality for your social networking Web site, by manipulating your friend-of-relation.
 - 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

Design guideline 3:

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

Today's Lecture

Define (insert n l) to insert an integer immediately before the first element of l that is $\geq n$

What's the type of insert?

```
;; insert: integerp true-listp -> true-listp
```

No, we want a list of integers. But do we have a recognizer for list of integers? No. So, let's define that.

```
;; loip: allp -> booleanp
;; (loip l) returns t iff l is a true-list of of integers
```

```
;; (assert-event (= (loip nil) t))
;; (assert-event (= (loip (list 1 2)) t))
;; (assert-event (= (loip (cons 1 2)) nil))
;; (assert-event (= (loip 4) nil))
```

If you look at the examples, note that we are visiting a linear list, so let's use the recursive scheme we've seen.

```
(defun loip (l)
  (cond ((endp l)
        ...)
        (t ... (first l) ...
              (loip ... (rest l) ...) ...)))
```

```
(defun loip (l)
  (cond ((endp l)
        (equal l nil))
        (t (and (integerp (first l))
                 (loip (rest l))))))
```

```
;; insert: integerp loop -> true-listp
;; (insert n l) inserts integer n immediately before the
;; first element of l that is >= n
```

Aside: what if I said $>n$, instead?

Aside: what should `(insert 4 '(1 2 3))` return?

Not clear. Our contract doesn't say what should happen in this case. Maybe it should? After all, we wouldn't want to return `(4 1 2 3)`, right? How do we fix the contract?

```
;; If no such element exists, n is inserted at the end of the
;; list.

;; (assert-event (= (insert 3 (list 0 1 2 6 8)) (list 0 1 2 3 6 8)))
;; (assert-event (= (insert 1 nil) (list 1)))
;; (assert-event (= (insert 2 (list 2 1)) (list 2 2 1)))
;; (assert-event (= (insert "hi" (list 1 2)) ??))
;; (assert-event (= (insert (list 1) (list -1 2)) ??))
```

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

```
(defun insert (n l)
  (cond ((endp l)
        ...)
        (t ... (first l) ...
            (insert ... (rest l) ...) ...)))

(defun insert (n l)
  (cond ((endp l) (list n))
        (t (cond ((>= (first l) n) (cons n l))
                  (t (cons (first l)
                           (insert n (rest l))))))))
```

Can we say more about the expected type of `insert`? For example, if the `len` of `l` is `k`, what is the `len` of `(insert n l)`? In fact, it should be `k+1`, so we can say:

```
;; insert: integerp loop -> loop
;; (= (len (insert n l)) (+ 1 (len l)))
```

This is a "dependent type" because the type of `insert` (`loop` of `len k+1`) now depends of the input (the `k` is the `len` of the input).

Again, we are working towards proving theorems, and in fact, we will be able to prove that our type comments are theorems:

```
(thm (implies (and (integerp n)
                  (loop l) )
            (loop (insert n l))))

(thm (= (len (insert n l))
        (+ 1 (len l))))
```

For next time, go off and define insertion sort.

```
;; isort: loop -> loop
;; (isort l) sorts l in increasing order
```

I'll pick a victim who will be asked to provide a solution.