

Announcements

---

- Tomorrow, we have exam1.
- It will cover everything we did until the end of last week.
- HW1 is due today; remember it is OK to work in pairs & recommended.
- Did everyone get email about submission instructions?
- Everyone log on to Blackboard?
- ACL2s: check for updates regularly (see ACL2s Web page on how to do this). Very simple.

Review

---

Review: Last time we focussed on cons and lists.

cons are ordered pairs.

Today we are going to continue with this, in order to give you more experience in defining functions on conses.

Today: Common recursive schemes

---

Today we are going to look at some common recursive schemes. This is in section 3.6 of your book. Read it.

Let's define some functions on lists.

I want to visit true-listp.

I also want us to write down a recognizer for the domain and range.

```
;; true-listp: All -> booleanp
;; checks if its input is a true list, i.e., is nil or a cons
;; that terminates in nil on the right-most branch.

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

What's a good way of giving a data definition for All? This is kind of new.

All: atom | cons All All

With this data definition, we are led to the following template:

```
(defun true-listp (x)
  (cond ((atom x)
         ... )
        (t ....
         (true-listp (car x)) ...
         (true-listp (cdr x)) ... )))
```

Let's fill it in:

```
(defun true-listp (x)
  (cond ((atom x)
         (equal x nil))
        (t (true-listp (cdr x)))))
```

A few comments.

Why did we define All as above?

When we prove theorems, we want to exactly characterize the intended domain, so let's start writing predicates recognizing the domains.

```
(defun allp (x)
  (or (atom x)
      (and (= x (cons (car x) (cdr x)))
            (allp (car x))
            (allp (cdr x))))))
```

What if we defined it as All: atom | consp?

```
(defun allp (x)
  (or (atom x)
      (consp x)))
```

The above is an equivalent definition of allp.

But, remember that every element of the ACL2 universe is an atom or a cons, so yet another equivalent definition is:

```
(defun allp (x) t)
```

Other ways of thinking of All

```
All: t
All: atom | consp
All: atom | cons All All
All: atom | cons (atom | cons All All) All
All: number | character | symbol | string | other_atom | cons All All
All: not_nat | Nat, where Nat: 0 | succ Nat
```

...

We can write recognizers for all of the above and prove that they are equivalent.

The interesting thing is that the above definitions are different in that they lead to different recursive schemes.

In 211, one of your jobs was to construct templates based on \*how\* arguments are used. For example, in true-listp all natural numbers were treated in the same way, so distinguishing between 0 and a positive number, that just wasn't important. So, our template did not make that distinction: it only had one case for atoms.

Another way of looking at this is that the recursive scheme we wind up with is important, so let's just directly look at some recursive schemes:

The first recursive scheme is for visiting the elements of a linear list:

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

```
;; 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))))))
```