Logic and Computation	**Lectures 7**
CSU 290 Spring 2009 (Pucella)	**Thursdayy, Jan 22, 2009**

Lists and Cons Structures

The aim today is to finish up the last bit of programming from ACL2 that we need, before turning to logic, our real goal.

This lecture will be somewhat post-modern: we are going to deconstruct lists. We are going to understand how lists arise in most programming languages.

Recall that when I talked about lists, I called them true lists, which naturally indicate that there might be such things as non-true lists. And indeed, there are.

The key is to examine what the `cons` operation is doing. The way we've been using cons is always with its second argument being a true list, including `nil`. But recall that ACL2 requires functions to be total, meaning that `cons` must return a value even if given something which is not a true list as a second argument. Thus, in particular, `(cons 1 2)` must denote some value.

In general, `(cons x y)` actually creates a *cons cell*, which we can represent as

```
+---+---+
|   |   |
+-/-+-\-+
 /     \
x       y
```

Thus, `(cons 1 2)` yields

```
+---+---+
|   |   |
+-/-+-\-+
 /     \
1       2
```

`(cons 1 (cons 2 3))` yields

```
+---+---+
|   |   |
+-/-+-\-+
 /     \
1     +---+---+
      |   |   |
      +-/-+-\-+
       /     \
      2       3
```

`(cons (cons 1 2) (cons 3 4))` yields:

```
      +---+---+
      |   |   |
      +-/-+-\-+
       /     \
+---+---+  +---+---+
|   |   |  |   |   |
+-/-+-\-+  +-/-+-\-+
 /     \    /     \
1       2  3       4
```

We call a structure made up of a cons cell at the root a *cons structure*. The recognizer `consp` is used to recognize a cons structure: `(consp x)` returns `t` if `x` is a cons structure, and `nil` otherwise.

Operations `car` and `cdr` are used to access the first component and second component of a cons cell. Thus, `(car (cons 1 (cons 2 3)))` returns `1`, and `(cdr (cons 1 (cons 2 3)))` returns the cons structure `(cons 2 3)`.

This is all nice and well, but what does this have to do with true lists? Well, true lists are just cons structures with a special form. Consider the true list `(cons 1 nil)`; as a cons structure, it looks like this:

```
+---+---+
|   |   |
+-/-+-\-+
 /     \
1      nil
```

The true list `(cons 1 (cons 2 nil))`, as a cons structure, looks like this:

```
+---+---+
|   |   |
+-/-+-\-+
 /     \
1     +---+---+
      |   |   |
      +-/-+-\-+
       /     \
      2      nil
```

The true list `(cons 1 (cons 2 (cons 3 nil)))` , as a cons structure, looks like this:

```
+---+---+
|   |   |
+-/-+-\-+
 /     \
1     +---+---+
      |   |   |
      +-/-+-\-+
       /     \
      2     +---+---+
            |   |   |
            +-/-+-\-+
             /     \
            3      nil
```

Do you see a pattern here? A true list has the property that if you repeatedly follow the second component of the cons cell that forms the root of the structure, you eventually end up with a `nil`.

The cons cells you reach following the second component of each cons cell starting from the root is called the *spine* of the cons structure, so an alternate characterization of a true list is a cons structure whose spine ends with a `nil`. The elements of the true lists can be read off the spine, by looking at the first component of each cons cell of the spine.

So a cons structure that is not a true list (i.e., whose spine does not end with a `nil`) is one of those non-true lists—also sometimes called an improper list.

The important thing is that when we design recursive functions over true lists *using the design template we've given you*, you are automatically taking care of improper lists and returning a value when they are given as input. The key is that `(endp x)` returns `t` exactly when `x` is *not* a cons structure.[1] What any function does on an improper lists depends on the function definition.

[1] In fact, `(endp x)` is defined to be `(not (consp x))`.

For example, if you try `(length (cons 1 (cons 2 3)))`, you can check that it computes `(+ 1 (+ 1 (length 3)))`, and since `(length 3)` returns 0, `(length (cons 1 (cons 2 3)))` returns 2.

As an exercise, check what `app` does if you give it an improper list as input, for instance, `(app (cons 1 (cons 2 3)) '(4 5 6))`.

This approach of defining true lists as a special case of a cons structure can be used in any programming language that has a notion of pairs (because, you've realized by now, a cons cell is just a pair).

To conclude this lecture, let's talk about how ACL2 actually represents cons structures, since it will not actually draw the kind of diagrams that I wrote above.

Intuitively, the cons cell

```
+---+---+
|   |   |
+-/-+-\-+
 /     \
x       y
```

will be represented as `(x . y)` by ACL2.

Thus, `(cons 1 2)` returns `(1 . 2)`, `(cons 1 (cons 2 3))` returns `(1 . (2 . 3))`, and so on. You can use the `'` notation just like for true lists, if you like: `'((1 . 2) . 3)` is the same as writing `(cons (cons 1 2) 3)`.

Actually, the result is displayed in a slightly optimized way, so that if you construct a true list, the result gets displayed like a true list, without any dots.

Here is how you can "compute" how ACL2 will display a cons structure. Think of it as applying the following rewrite rules repeatedly until none applies any more:

(1) (a $\cdots$ b . `nil`) gets rewritten as (a $\cdots$ b);

(2) (a $\cdots$ b . (x $\cdots$ y)) gets rewritten as (a $\cdots$ b x $\cdots$ y);

(3) (a $\cdots$ b . (x $\cdots$ y . z)) gets rewritten as (a $\cdots$ b x $\cdots$ y . `z`).

Thus, you can check that

- `(cons 1 (cons 2 3))` would be displayed as `(1 . (2 . 3))` which by (3) gets rewritten as `(1 2 . 3)` which is what actually gets displayed;
- `(cons 1 (cons 2 nil))`—note that this is a true list— would be displayed as `(1 . (2 . nil))` which by (3) gets rewritten as `(1 2 . nil)` which by (1) gets rewritten as `(1 2)`, agreeing with the usual representation of true lists.