

Announcements

- Next Thursday, we have exam1.
- It will cover everything we until the end of this week.
- To help you prepare, check the Web page for HW1.
- Lab sessions will also include a review of the material and homework.

Review

Review: Last time we focussed on the totality of ACL2.

Design guideline 1:

Test your programs with values that are outside of their intended domain.

Design guideline 2:

Atomic data outside the intended domain should be recognized and handled in a non-recursive way with the proper idiom.

Idiom	Intended domain	Completion
(zp x)	(natp x)	(= (nfix x) 0)
(zip x)	(integerp x)	(= (ifix x) 0)

Design check 1:

After designing a program, perform a totality check.

The ACL2 language: cons and lists

Conses are ordered pairs of objects. They are by far the most commonly used objects in ACL2.

Conses are sometimes called ``lists,`` ``cons pairs,`` ``dotted pairs,`` or ``binary trees.``

There are many ways to write a given list. This should not be surprising. Consider the fact that 123, 000123, 246/2, and #b1111011 are all ways to write down the same numeric constant and stylistic considerations determine which form you use. So too with list constants.

Any two objects may be put together into a cons pair. The cons pair containing the integer 1 and the integer 2 might be written as <1, 2> or drawn as follows.

$$\begin{array}{c} \diagup \quad \diagdown \\ 1 \quad 2 \end{array}$$

In ACL2 it is written as (1 . 2) and is created with (cons 1 2).

This is a single cons object in ACL2 with two integer objects as constituents. The left-hand constituent is called the car of the pair. The right-hand constituent is called the cdr. Note: car is equal to first and cdr is equal to rest. I don't care which you use.

Note: conses are recognized by consp. Also, conses are different from the atomic data we've seen until now, e.g., numbers. It is a theorem that:

(thm (or (atom x) (consp x)))

```
(thm (implies (atom x) (not (consp x))))
(thm (implies (not (atom x)) (consp x)))
```

Let's look at more examples. <1, <2, 3> may be drawn as:

```
  /\
 1 /\
 2 3
```

and can be written in ACL2 as (1 . (2 . 3))

What about (1 . (2 . (3 . nil)))

What tree corresponds to this?

Does every binary tree correspond to a cons? Yes.

The notation we are using to write list constants is called **dot notation**. It is a straightforward translation of the familiar Cartesian coordinate notation in which parentheses replace the brackets and a dot (which must be surrounded by whitespace) replaces the comma.

ACL2 has two syntactic rules that allow us to write cons pairs in a variety of ways. The first rule provides a special way to write trees like (1 . nil). This tree may be written as (1). That is, if a cons pair has the symbol nil as its cdr, you can drop the *'dot'* and the nil when you write it.

Examples:

```
(1 . ((2 . nil) . (3 . nil)))
```

==>

```
(1 . ((2) . (3)))
```

The second rule provides a special way to write a cons pair that contains another cons pair in the cdr: you may drop the dot and the balanced pair of parentheses following it. Thus (x . (...)) may be written as (x ...). For example, (1 . ((2) . (3))) simplifies to (1 (2) 3).

Another example: (1 . (2 . (3 . nil))) may be written as:

- (1 . (2 . (3)))
- (1 . (2 3))
- (1 2 3)
- (1 2 . (3 . nil))
- etc.

Binary trees that terminate in nil on the right-most branch, such as the tree above are called **true lists**. By extension, the symbol nil is also called a (true) list or, in particular, the **empty list**, and is sometimes written (). Note that the empty list is equal to nil.

The **elements** of a list are the successive cars along the *'cdr chain.'* That is, the elements are the car, the car of the cdr, the car of the cdr of the cdr, etc. The elements of the list (1 2 3) = (1 . (2 . (3 . nil))) are 1, 2, and 3, in that order. There are no elements in the empty list.

Combinations of car and cdr are so common that a naming scheme exists. For example, the cadr of a list is the car of the cdr

(second = cadr) and the caddr is the car of the cdr of the cdr, (third = caddr), etc.

If we let x denote the tree (1 2 3), then the car of x is 1, the cdr of x is (2 3), the cadr of x is 2, the caddr of x is 3, and the caddr of x is nil.

Of course, the elements of a list may be conses. Consider the tree:

Draw the tree for ((0 . 1) (1 . 2) (2 . 3)). Its car is the cons pair (0 . 1), a pair whose car is 0.

Lists such as the above are so common they have a name. They are called association lists or alists. The list above associates the "key" 0 with the "value" 1, the "key" 1 with the "value" 2, etc.

Note: alists can be used to represent functions over a finite domain.

Later we show functions for manipulating alists.

Here is another alist.

```
(("NAME" . ((("FIRST" . "John")
              ("LAST" . "Smith"))
 ("AGE" . 35)
 ("HEIGHT-IN-METERS" . 24/13)
 ("ADDRESS" . ((("STREET" . "1404 Elm Street")
                 ("CITY" . "San Lajitas")
                 ("STATE" . "TX")
                 ("ZIP" . 79834))))))
```

Observe that some of the ``values`` are themselves alists. Whitespace may be inserted into the display of a list to improve appearance and readability, as long as it does not destroy the parsing of the atoms. When the elements of a linear list cannot be displayed on one line, many programmers ``stack`` the elements one under the other. This is called ``pretty printing`` and involves questions of aesthetics.

Totality: the notions of car and cdr are defined also on atomic data: the car and cdr of any atom is nil.

Beware: this does not mean that nil is produced by consing nil onto nil! Which is?

For each list below, how many elements are in the list?
How many distinct elements?

- (6 +6 -6 #b110 -12/2)
- (NIL () (NIL) (()))
- ((1 2) (1 2) (1 . 2) (1 2 . NIL))

What is the length of the longest branch in each of the following binary trees? Also, for each tree, write down the list of atoms in the leaves, from leftmost to rightmost. Does either tree contain the same atom twice as a leaf? It sometimes helps to draw the tree.

```
((1 2 . 3) . 4)                                    -> 3
((6 7 8) 1 2 (3 . 4) 5)                       -> 5
```

Since lists are so common, we have support for constructing them,

using list:

```
(list 1 2 3 4)
```

=

```
(cons 1 (cons 2 (cons 3 (cons 4 nil))))
```