

Announcements

- Exam 2, next Thursday, will cover everything up until the end of today.
- Reading: sections 3.1-3.6, 4.1-4.2 of the book.

Review

Select a victim to give us the solution to the isort problem.

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

How would you do this?

```
;; (assert-event (= (isort (list -2 4 5 1)) (list -2 1 4 5)))
;; (assert-event (= (isort nil) nil))
;; (assert-event (= (isort (list "hi" -2 101 -3/2 1)) (list .. -2 .. 1 .. 101)))
```

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

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

```
(defun isort (l)
  (cond ((endp l) nil)
        (t (insert (first l)
                    (isort (rest l))))))
```

Today's Lecture - REPL, Quote, Symbols, If

THE READ-EVAL-PRINT LOOP

ACL2 offers a REPL (read-eval-print loop) where you can evaluate expressions to get results and to test out your functions.

The prompt to the REPL is

```
ACL2 >
```

When you type an expression at the REPL, ACL2:

- reads it,
- evaluates it (that is, computes its value),
- prints the value it has computed, and
- loops, waiting for your next input

Actually, what you see is the `_printed representation_` of the computed value.

For example:

```
5 --> 5
```

where 5 is the (printed) representation of the value 5.

```
(+ 10 5) --> 15
```

where 15 is the representation of 15, the result of evaluating (+ 10 5)

```
(cons 1 (cons 2 nil)) --> (1 2)
```

Expression (cons 1 (cons 2 nil)) evaluates to a list of two elements (1 and 2), which is represented as (1 2).

[note - this representation for lists is different than what you saw in DrScheme's REPL]

Let's quickly review representations.

REVIEW OF REPRESENTATIONS

All values have a representation, which are used to represent the value to the user (for instance, when printed by the read-eval-print loop).

Let's review what we know about the representation of values.

Booleans are easy. The representation of the Boolean value T is just T, while the representation of the Boolean value NIL is just NIL.

Numbers are already more interesting. We saw that there are several kinds of numbers: naturals, integers, rationals, and complex rationals. The representation of a number depends on the kind of number it is.

An integer is represented by -2,-1,0,1,2,... A rational number that is not an integer is represented by a fraction, e.g. 4/5. A complex number is represented by a special notation, e.g. #c(1 2)

Again, there are many ways of denoting the same number and ACL2 prints out a "canonical" representation, which is intended to be the "simplest" representation. That's why 10/2 prints as 5. Similarly, 33/22 prints as 3/2 and #c(1 0) prints as 1.

Another way of saying this is that there are many ways to denote the same number. As we know, 22/11, 2, #b10, all denote 2.

What about other atomic data?

A string value such as "hello" is represented as "hello".

A character value such as #\a is represented as #\a.

REPRESENTATION FOR ATOMIC DATA: for the atomic data we've seen, the representation of a value is exactly the same as the expression we have to write to produce that value (modulo simplification in the case of numbers).

Cons pairs are even more interesting. We saw that `(cons 1 2)`, which returns a pair, represents that the pair as `(1 . 2)`. Similarly, the true list `(cons 1 (cons 2 (cons 3 nil)))`, or equivalently `(list 1 2 3)`, gets represented as the pair `(1 . (2 . (3 . nil)))`.

Here, the representation of a cons pair is genuinely different from how the expression we have to write to produce that cons pair.

As we saw, the representation of cons pairs can be simplified somewhat using a couple of rules.

In particular, the cons pair represented by `(1 . (2 . 3))` can also be represented by `(1 2 . 3)`.

Similarly, the true list represented by `(1 . (2 . (3 . nil)))` can also be represented by `(1 . (2 3))` or `(1 2 . (3 . nil))` or `(1 2 . (3))` or `(1 2 3)`.

Again, ACL2 will print the canonical, or simplest representation - in the examples above, these are `(1 2 . 3)` and `(1 2 3)`. But all the representations in each example represent *the same value*.

REPRESENTATION FOR CONS PAIRS: the representation of a cons pair is different from the expression we have to write to produce that cons pair.

QUOTE

The above probably sounds overly pedantic to you. As it should.

The only reason why we care about this is that there is a mechanism to write a value by essentially saying "give me the value whose representation is this".

This mechanism is called *quote*.

To be more precise, the expression

```
(quote some-representation)
```

yields as a value the value whose representation is "some-representation". (When there are many choices, it will choose the simplest value with that representation.)

Some examples:

What is the value whose representation is T? Easy, T. Thus, `(quote T)` will return the value T.

What is the value whose representation is 5? Easy, 5. (Actually, there are many, 10/2, 20/4, #c(5 0), but 5 is the simplest such.) So (quote 5) will return the value 5.

Just to be a cute, what is the value whose representation is 10/5? Well, 10/5 represents the same number as the representation 2 (they are both representations of the number 2), and since 2 is the simplest value with that representation, (quote 10/5) will return the value 2.

Let's be more interesting. What is the value whose representation is (1 2)? We can think about this a few ways, but let's do it the long way. (1 2) is an alternative representation for (1 . (2 . nil)), which is the result of evaluating (cons 1 (cons 2 nil)). So the value whose representation is (1 2) is the same value that you obtain when evaluating (cons 1 (cons 2 nil)), which is, of course, the true list with elements 1 and 2. So, (quote (1 2)) returns the true list containing elements 1 and 2. In other words, (quote (1 2)) gives you the same value as evaluating (cons 1 (cons 2 nil)).

Similarly, the value whose representation is (1 . 2) is the same value that you get when you evaluate (cons 1 2), and therefore (quote (1 . 2)) evaluates to the same value as (cons 1 2).

So we see that in the case of cons pairs, the (quote ...) notation is a convenient way to describe a value.

(quote some-representation) can be abbreviated 'some-representation.

What does the following evaluate to?

```
ACL2 > '5
5
```

```
ACL2 > '10/2
5
```

```
ACL2 > '33/22
3/2
```

```
ACL2 > 'T
T
```

```
ACL2 > '(1 2 3 4 5)
(1 2 3 4 5)
```

```
ACL2 > '(1 2 . (3 4 5))
(1 2 3 4 5)
```

```
ACL2 > '(1 . (2 . 3))
(1 2 . 3)
```

What about ''1?

It is really (quote (quote 1)) so --> (quote 1), which is printed '1.

So, what is (length ''1)? 2!

SYMBOLS

Time for us to talk about the last form of atomic data, symbols.

Symbols are just that, symbols. For instance, when defining colors, we would like one color to have values such as red, blue, green, etc.

ACL2 also lets us define constants, using symbols.

Constants are symbols that start and end with*, e.g.:

```
(defconst *a* 10)
```

Whenever you evaluate *a* after you've defined it, it will be replaced by its value, 10.

```
ACL2 > *a*
10
```

```
ACL2 > (/ *a* 2)
5
```

While symbols and strings may seem similar, they are very different.

For example, strings evaluate to themselves: `"*a*"`

But, symbols do not.

So how do we get symbols? Well, let's think backwards from the representation. We would like the symbol red to be represented as just red. Similarly, we would like the symbol blue to be represented as just blue. That choice is the most natural. So, following what we saw above, one way to get an expression to give us a symbol whose representation is blue is just to quote the representation, that is, 'blue. Thus, the expression 'blue yields the symbol blue, which is represented as just blue.

Easy, huh?

```
'red --> red
'blue --> blue
'hello-i-am-a-symbol --> hello-i-am-a-symbol
'*a* --> *a*
```

ACL2 provides several functions for dealing with symbols. For example, there is a symbol recognizer.

As with all recognizers, its domain is Allp and its range is booleanp.

```
;; Allp -> boolean
(symbolp x) is a predicate to check if an argument is a symbol.
```

```
ACL2 > (symbolp 'abc)
T
```

```
ACL2 > (symbolp '(1 2))
NIL
```

```
ACL2 > (symbolp 10)
NIL
```

```
ACL2 > (symbolp '10)
NIL
```

You can also compare symbols.

```
ACL2 > (= 'abc 'def)
NIL
```

```
ACL2 > (= 'abc 'abc)
T
```

```
ACL2 > (= 'abc 'abcd)
NIL
```

```
ACL2 > (= 'abc 'ABC)      [ case insensitive ]
T
```

No symbol is equal to a non-symbol.

```
(thm (implies (and (symbolp x)
                   (stringp y))
              (not (= x y))))
```

Introducing: if

ACL2 includes if, which is just if-then-else

For example,

```
(cond ((endp x) nil)
      (t (car x)))
```

is equivalent to:

```
(if (endp x)
    nil
    (car x))
```

You can do this in ACL2 as follows:

```
(thm
 (= (cond ((endp x) nil)
          (t (car x)))
    (if (endp x)
        nil
        (car x))))
```

In fact, cond is just a shorthand for if. It "expands" into an if. You can also see what the cond represents as follows:

```
:trans (cond ((endp x) nil)
             (t (car x)))
```

What about

```
:trans (cond ((endp x) nil)
             ((endp (cdr x)) (car x)))
```

Remember ACL2 is total, so if your cases are not exhaustive, ACL2 will return nil.