

Announcements

- . Every ordered the book?
- . ACL2s: download it now.
- . Labs: register, signup sheets on WVH 328.
- . The first lab is on *Friday*. The point of the lab will be to go over the ACL2s installation, so bring your laptops, if you want help with the installation.

Today: The ACL2 language, Basic Data Types

We are going to start by introducing the ACL2 language and doing it in a way that highlights differences with Scheme, and in a way that reinforces and reviews what you learned in 211.

The ACL2 language can be seen as a "pure" variant of LISP. Scheme is also a variant of LISP, so the languages are very similar, but they have important differences.

Let's start with the basic data types.

Booleans

t: true
nil: false

There is a recognizer for booleans: `booleanp`.

It takes as input anything.
It returns t (true) if its argument is t or nil.
It returns nil (false) otherwise.

```
(booleanp t) -> t  
(booleanp nil) -> t  
(booleanp 0) -> nil
```

An example of a function that always returns a boolean is `equal`:

```
(equal a b) -> t      if a and b are equal  
             -> nil   otherwise
```

You can also write `(= a b)`

We have the standard boolean connectives: `and`, `or`, `not`, `implies`, and `iff`.

Numbers and arithmetic

Numbers are atomic data types in ACL2. They include:

integers, rationals, and complex rationals, but *no* inexact numbers.

Numbers are mostly written in base 10, but other bases can also be used. For example:

1023 may also be written as 001023, +1023, #b1111111111, #x3ff, #o1777.

All of these numbers are equal; meaning ACL2 cannot distinguish them; they denote the same thing.

For example

```
(= x y), so x, y from above
```

```
returns t (true)
```

There is a built-in function that recognizes integers. It is `integerp`, so `(integerp #x3ff)` returns `t (true)`.

Rationals are either integers or are written in the form of fractions. Here are some examples of rationals: `\ptt{0}`, `\ptt{-77}`, `\ptt{123}`, `\ptt{1/3}`, `\ptt{22/7}`.

There is a predicate to recognize rationals, so `(rationalp 1023)` is `t` (because all rationals are integers). Similarly `(rationalp 129/3)`, but also `(integerp 129/3)`.

As before, there are many ways to represent rationals, e.g., 14 can be represented as 28/2 and `#xe` and `#x1c/2` and `#x1c0/20`.

So, rationals have a numerator and a denominator. And, fractions are reduced internally to lowest terms. You can access these as follows:

```
(numerator 12)
(denominator 12)
```

```
(numerator 3/2)
(denominator 3/2)
```

```
(numerator 123/3)
(denominator 123/3)
```

```
(numerator #x1c0/20)
(denominator #x1c0/20)
```

The complex number $3+5i$ is written `#c(3 5)`. The real and imaginary parts of a complex number are always rationals. `#c(3 0)` is 3. `complex-rationalp` is a recognizer for complex rationals.

```
(complex-rationalp #c(3 5))
(complex-rationalp #c(3 0))
```

A number satisfies `complex-rationalp` iff it doesn't satisfy `rationalp`.

```
(thm (implies (acl2-numberp x)
              (iff (complex-rationalp x)
                    (not (rationalp x)))))
```

So, what about the following:

```
(complex-rationalp 3)
(complex-rationalp 3/2)
(complex-rationalp #c(3 0))
(complex-rationalp #c(3 2))
```

Is there an integer that is also a complex-rational? No.

We won't have much to say about complex rationals.

In contrast to Scheme, all numbers are represented exactly. There are no inexact numbers in aCL2. (Aside: you can represent floating point numbers with rationals and people do that, e.g., AMD does this to verify their floating point units, but these numbers are not built in to ACL2.)

```
(expt 2 10)
(expt 2 50)
(expt 2 (expt 2 10))
(expt 2 (expt 2 50))
```

ACL2 has the standard built-in "functions" +, *, -, /:

```
(+ 12 3/4)
(* 12 4)
(/ 2 18)
(- 12 2)
```

Also, note

```
(- 3)
(+ 1 2 3 4)
(+ )
(* 1 2 3 4)
(* )
(/ 1 2 3)
(/ 3)
(- 1 2)
```

So, + and * can take any number of arguments.
- and / can take 1 or 2 arguments.

You can compare numbers with =, <, <=, >, >=, etc.

```
(< 22/11 3)
```

In fact, >, >=, <= can all be defined in terms of <. How?

```
(> a b) iff (< b a)
(<= a b) iff (not (< b a))
(>= a b) iff (not (< a b))
```

In fact, that is how >, >=, and <= are defined in ACL2.