

#### 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.
  - It will be up this evening.
  - It will be due on Wednesday.
- Lab sessions will also include a review of the material.

#### Review

---

Review: Last time we started looking at the basic data types in ACL2.

What are the booleans? `t`, `nil`.

What's the recognizer for booleans? `booleanp`

What is (numerator 8/12)?

What is the recognizer for integers? `(integerp)` naturals? `(natp)`.

Does ACL2 have inexact numbers? No.

The ACL2 language: Totality

---

Today: ACL2 is a language of *\*total\** functions. What this means is that *\*every\** function is defined for every input. We'll see what this means and how to design ACL2 programs.

Let's start by trying this in to errors.

#### Syntax Errors

---

Just like in Scheme, not all parenthesized expressions are ACL2 expressions.

For example:

`(10 + 20)` (should be `(+ 10 20)`)

This is not a legal expression.

Another example:

`(define (f x) x)`

should be

`(defun f(x) x)`

#### Runtime errors

---

However, there are expressions such as `(/ x y)` that are "legal expressions" and yet, for some values, Scheme will give an error.

For example `(/ 1 0)`

In contrast to Scheme, all "legal" ACL2 expressions *\*have\** a value (i.e., do not result in an error). We sometimes say that

ACL2 is "total" or "untyped" to mean that all ACL2 functions accept anything as input. Put another way, the domain of every function in ACL2 is the whole of the ACL2 universe.

A "nice" consequence is that no errors (like the above) ever occur. However, designing programs for a total language requires a mental adjustment.

We'll see how to make that adjustment today.

Totality  
-----

Again, don't fight totality. Accept it.

OK?

ACL2 functions are total; their domain is the ACL2 universe.

Let us look at an example.

Let us define the factorial function, !.

```
;; !: natural -> natural
;; computes the factorial of its input, e.g., if the input is x,
;; (! x) 1 if x is 0, else it is x (x-1) (x-2) ... 1.

;; (assert-event (= (! 0) 1))
;; (assert-event (= (! 1) 1))
;; (assert-event (= (! 5) 120))

;; (assert-event (= (! -1) 1))

; I don't expect them to come up the above test, so add it later
```

The function we will define is:

```
(defun !(x)
  (cond ((= x 0) 1)
        (t (* x (! (1- x))))))
```

but they don't know about zp, so they will instead probably check if x=0, which will be bad; non-termination.

Is this OK in Scheme (checking for (= x 0) in the base case)? Maybe, because the contract says that the input should be a natural number, and if the contract is satisfied, then we don't have a problem.

But, in ACL2, your functions are? *\*total\**!, meaning that they have to work given any input.

For example, what happens if x=-1? Try it and show them how to interrupt ACL2.

How do we deal with this?

Design guideline 1:

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

So, let's add:

```
;; (assert-event (= (! -1) ??))
;; (assert-event (= (! 2/3) ??))
```

This principle doesn't give guidance on how to define ! outside the intended domain, which is why we have ??'s for now.

Let's address that next.

Design guideline 2:

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

Let's expand this to make clear what is meant.

Let's start by looking at idioms for "testing for 0".

An idiom is an expression that solves a commonly occurring programming problem.

If the expected domain is the natural numbers, then one idea is to *coerce* non-naturals to naturals. There are functions to do this in ACL2. For example,

```
(nfix x) = x if x is a natural number and 0 otherwise.
(ifix x) = x if x is an integer and 0 otherwise.
(rfix x) = x if x is a rational number and 0 otherwise.
```

So, the test above should have been

```
(equal (nfix x) 0)
```

Why? consider the cases: natural number, not natural number.

Since this happens so often, there are idioms for testing for 0:

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

where nfix is defined as:

```
(cond ((natp x) x)
      (t 0))
```

and ifix is defined as:

```
(cond ((integerp x) x)
      (t 0))
```

Let's apply design principle 2.

We are testing atomic data in the cond, so let's recognize and handle data outside the intended domain with the proper idiom, which is? zp

```
(defun !(x)
  (cond ((zp x) 1)
        (t (* x (! (1- x))))))
```

This idea of coercing input outside the intended domain to some element of the intended domain comes up over and over.

So, now let's replace the "???" in our asserts with the right answer, which is? 1.

Note that the above definition is equivalent to the following definition:

```
(defun !(x)
  (cond ((not (natp x)) 1)
        ((= x 0) 1)
        (t (* x (! (1- x))))))
```

But, notice how much nicer the code using the `zp` idiom is. Using the right idiom allowed us to to write code that:

- works for *any* input
- looks like code we were writing before: same structure
- provided totality while minimizing the number of changes (violence) done to the original code

Design check 1:

After designing a program, perform a totality check.

This includes checking not only your code, but your entire design, including the examples.

One more comment: Note the asymmetry between the domain and range of a function. When we say that the domain of `!` is `natp`, then we really mean that `!` should behave according to the contract on *all* natural numbers given as input. But, when we say that the range of `!` is `natp`, then we don't mean (or require) that every natural number can be the output of `!`. For example, `!` can *never* return 3. Our only obligation is to show that whatever `!` returns is a natural number.

Let's design another program.

Let's design a program to construct pascal's triangle.

```

      1
     1 1
    1 2 1
   1 3 3 1
  1 4 6 4 1
 1 5 10 10 5 1
1 6 15 20 15 6 1
...

```

```
;; pascal: natural, natural -> positive natural
;; computes the i,j th entry of Pascal's triangle, e.g.,
;; if the input is i, j, then we compute the entry in the
;; ith row, jth column of Pascal's triangle, where rows
;; and column numbers start at 0.
```

```
;; (assert-event (= (pascal 0 0) 1))
;; (assert-event (= (pascal 2 0) 1))
;; (assert-event (= (pascal 5 2) 10))
```

```
;; (assert-event (= (pascal -1 2) ??))
;; (assert-event (= (pascal 2 -1) ??))
;; (assert-event (= (pascal -1 -1) ??))
```

```
(defun pascal (i j)
  (cond ((zp i) 1)
        ((zp j) 1)
        ((= j i) 1)
        (t (+ (pascal (- i 1) (- j 1))
              (pascal (- i 1) j)))))
```

What's the relationship between `pascal` and `!`?

Well,  $\text{pascal } i \ j$  is the binomial coefficient  $\binom{i}{j}$  ( $i$  choose  $j$ ). You've seen this in discrete math. It is the number of ways to choose  $j$  out of  $i$  elements.

And  $\binom{i}{j}$  is just  $i!/(j! (i-j)!)$ .

It is also the case that  $\binom{i}{j} = \binom{i-1}{j-1} + \binom{i-1}{j}$ . Why? Well, pick one of the  $i$  elements. Either you can choose it or not. If you choose it, then what remains is to choose the  $j-1$  of  $i-1$  elements  $\binom{i-1}{j-1}$ . If you don't then what remains is to choose  $j$  of  $i-1$  elements  $\binom{i-1}{j}$ , so that is  $\binom{i-1}{j-1} + \binom{i-1}{j}$ .

so, we could have added more test cases, e.g.,

```
(assert-event (= (pascal 6 2)
                  (/ (! 6) (* (! 2) (! (- 6 2)))))))
```