Announcements

* Advice: Whenever you make a mistake, analyse it and learn from it.

Here is some basic information on ACL2 Language that you will also find in the book, I didnt cover this but you should go over it.

ACL2 Values

All data objects in ACL2 can be categorized as follows: (you don't have to remember all of these)

(Description)	(Predicate/recognizer)
* Atomic data ("atoms") include:	ATOM or ENDP
* ACL2 Numbers are either:	ACL2-NUMBERP
* Rationals include:	RATIONALP
* Integers	INTEGERP
* Complex rationals	COMPLEX-RATIONALP
* Symbols include:	SYMBOLP
* Booleans	BOOLEANP
* Keywords	KEYWORDP
* Strings	STRINGP
* Characters	CHARACTERP
* Cons pairs (compound data)	CONSP

Also, the only way to build compound data is with Cons pairs. We will discuss this in more detail soon.

Here are the categories you should know:

(Description)	(Predicate)	(Examples)
* Atoms	ATOM or ENDP	

```
  include:
  * Rationals      RATIONALP        3/4  -20/3
    include:
    * Integers     INTEGERP         -7  0  523
  * Symbols        SYMBOLP          'green  'two
    include:
    * Booleans     BOOLEANP         t  nil   (exhaustive)
  * Strings        STRINGP          "hi"   "Who, me?"
  * Characters     CHARACTERP       #\A  #\q
* Cons pairs       CONSP            '(1 . 2)
```

Booleans

ACL2 and Common Lisp have two special constant symbols that are used, among other things, as boolean values:

```
  t     stands for "true"
  nil   stands for "false" (and has other uses as you have seen today)
```

All of the predicate functions above return booleans. For example,

```
  (booleanp t)   = t      (It is true that t is a boolean)
  (booleanp nil) = t      (It is true that nil is a boolean)
  (booleanp 7)   = nil    (It is false that 7 is a boolean)
```

t and nil are also symbols, which are atoms:

```
  (symbolp t)   = t      (It is true that t is a symbol)
  (symbolp nil) = t      (It is true that nil is a symbol)
  (atom t)      = t      (It is true that t is an atom)
  (consp nil)   = nil    (It is false that nil is a cons pair)
```

Another function that always returns a boolean is EQUAL, which tests data objects for equality:

```
  (equal nil nil)  = t
  (equal nil t)    = nil
  (equal T t)      = t      (Case is ignored when reading symbols)
```

A few functions that are useful on booleans are NOT, AND, and OR:

```
  (not t)              = nil
  (not nil)            = t
```

```
  (not (equal nil t))  = t
  (and t t)            = t
  (and nil t)          = nil
  (or nil t)           = t
  (or nil nil)         = nil
```

AND and OR are special in that they can take any number of parameters. AND returns "true" iff all of its parameters are "true", and OR returns "true" iff at least one of its parameters is "true":

```
  (and (equal nil nil)
       (equal t t)
       (not (equal t nil)))  = t
  (and t)                    = t
  (and nil)                  = nil
  (and)                      = t    (All (zero) parameters are "true")
  (or)                       = nil  (No parameter is "true")
  (or t)                     = t
  (or nil)                   = nil
  (or (equal nil t)
      (equal t t)
      (not (equal t t)))     = t
```

Numbers

As mentioned, all ACL2 numbers are exact, and their size is (in theory) unbounded. This makes ACL2 numbers behave as they do in mathematics, but we will stick with just rational numbers--those that can be represented as one integer divided by another.

But first we consider those rationals with a denominator of 1: the integers. There are many ways to write integers in ACL2--all refering to the same integers--but the standard way is fine for us:

```
  (integerp -5)    = t
  (rationalp 37)   = t
  (equal 42 042)   = t
```

We can compare them:

```
  (< 4 5)    = t
  (<= 4 -5)  = nil
  (> 0 -5)   = t
```

```
  (>= 7 7)   = t
```

We can also do some standard arithmetic:

```
  (+ 4 3)   = 7
  (- 3 5)   = -2
  (* 2 -5)  = -10
  (/ 10 5)  = 2
```

These functions are special, however, because they can different numbers of parameters. For example, + and * can take any number of parameters:

```
  (+ 5)      = 5
  (+ 3 2 1)  = 6
  (* -2)     = -2
  (* 1 2 3)  = 6
  (+)        = 0   (Additive identity, the sum of zero numbers)
  (*)        = 1   (Multiplicative identity, the product of zero numbers)
```

- and / can take one or two parameters. Given one argument, they perform negation and multiplicative inversion respectively:

```
  (- 5)    = -5
  (- -10)  = 10
  (- 0)    = 0
  (/ 1)    = 1
  (/ 5)    = 1/5
  (/ -2/3) = -3/2
```

And that brings us to non-integer rationals. Equality among rationals is mathematical equality:

```
  (equal 7/9 14/18)  = t
  (equal 8/2 4)      = t
```

In fact, ACL2 automatically puts rationals in lowest terms, as shown when printing them out or accessing the numerator or denominator:

```
  -4/6          = -2/3
  20/4          = 5
  (/ -6 -4)     = 3/2
  (/ 15/6 1/2)  = 5
  (numerator -4/6)   = -2
  (numerator 22/8)   = 11
  (numerator -5)     = -5
  (denominator -4/6) = 3
```

```
  (denominator 22/8) = 4
  (denominator -5)   = 1
  (denominator 0)    = 1
```

Arithmetic on rationals works as expected:

```
  (+ 2/3 3/5)   = 19/15
  (+ 3/4 5/6)   = 19/12
  (* 2/3 3/5)   = 6/15
  (* 3/4 5/6)   = 5/8
```

Which raises an interesting question: what is (/ 1 0) ? We know in mathematics that anything divided by zero is undefined. So does ACL2 throw an error or what?

Introduction to Totality

ACL2 functions are total, which means every function call returns a value. This is one of the fundamental design choices in the Boyer-Moore theorem prover, ACL2. For now, you can consider this choice to keep some things simple by eliminating exceptional cases. For example, ACL2 defines anything divided by 0 to be 0. Consequently, division in ACL2 always returns a number, and we don't need to figure out anything about the input to come to that conclusion. (More on this stuff comes later in the course.)

```
  (/ 1 0)   = 0
  (/ 0 0)   = 0
```

Also for totality, arithmetic functions treat non-numbers as 0. Here are some examples:

```
  (+ 5 nil)    = 5
  (* t 12)     = 0
  (- nil)      = 0
  (/ 5 nil)    = 0
  (< nil 5)    = t
  (> nil 5)    = nil
  (< -5 nil)   = t
```

Totality also means there are no "type errors" in which a function was expecting one type of input and got another. The only thing we have to get right is the number of parameters, and that's an easy

check for ACL2:

```
 ACL2 >(booleanp 1 2)

 ACL2 Error in TOP-LEVEL:  BOOLEANP takes 1 argument but in the call
 (BOOLEANP 1 2) it is given 2 arguments.   The formal parameters list
 for BOOLEANP is (X).

 ACL2 >
```

Back to Numbers, with Function Definition

Here are some extra functions pertaining to numbers:

```
  NATP - predicate for natural numbers  (0, 1, 2, ...)
  POSP - predicate for positive naturals (1, 2, 3, ...)
  ZP   - (not (posp x))
```

We will see how useful ZP is in our first function definition. Let us define the factorial function, call it FACT. Here are examples of how factorial is written and defined in mathematics:

```
  5! = 5 * 4 * 3 * 2 * 1
  2! = 2 * 1
  1! = 1
  0! = 1
```

So for any integer greater than 0, the factorial is that integer times the factorial of one less than the integer. (Recursion!) Here's a first try at a definition:

```
  ; fact: nat -> nat
  (defun fact (x)
  "Computes the factorial function"
    (if (= x 0)
      1
      (* x (fact (- x 1)))))
```

An IF looks like

```
  (if  <test>  <true_part>  <false_part>)
```

If the test is true, then it evaluates to the true_part, otherwise the false part.

Our definition works on natural numbers, but is it total? Does it return a value on all inputs?

The answer is No. If we give it a negative number (or indeed, a non-number) it does not terminate:

```
(fact -1)
=
(* -1 (* -2 (* -3 ...
```

A solution is to treat everything outside the intended domain, the naturals, as a base case, 0. So the new base case is 0 or anything not a natural number. If only we had a function that captured all of that...

```
(defun fact (x)
  (if (zp x)
    1
    (* x (fact (- x 1)))))
```

Although we specified the contract, its a comment, ACL2 does not know about it, for
now we will make a distinction between :
- INTENDED DOMAIN : specified in the contract, for e.g in fact it was "nat"
- ACTUAL DOMAIN : For now it is the whole ACL2 Universe

ACL2 Language Errors

We saw last time that there aren't any runtime ``type'' errors; ACL2 functions are untyped and total. But it's not true that anything we right down gives us an answer. There are many *static* errors, in which ACL2 rejects an expression or definition before it tries to execute it.

If we place something in the function position of an expression which is not a function symbol, then we get an error:

```
ACL2 > (10 20 30)

ACL2 Error in TOP-LEVEL:  Function applications in ACL2 must begin
with a symbol or LAMBDA expression.  (10 20 30) is not of this form.

ACL2 >
```

Functions cannot be used as values:

```
  ACL2 > (+ endp consp)

  ACL2 Error in TOP-LEVEL:  Global variables, such as CONSP and ENDP,
  are not allowed. ...

  ACL2 >
```

And you must give functions the correct number of arguments. Unlike advanced Scheme, in which this check (and some of the previous) are made a run time, ACL2 makes this check before executing anything:

```
  ACL2 > (if t 42 (booleanp 1 2))

  ACL2 Error in TOP-LEVEL:  BOOLEANP takes 1 argument but in the call
  (BOOLEANP 1 2) it is given 2 arguments.   The formal parameters list
  for BOOLEANP is (X).

  ACL2 >
```

Note that in that expression (booleanp 1 2) would not be evaluated. (By the way! IF is a special function in that it only evalutes either the <true_part> or the <false_part>, never both. That is pretty important to termination of recursive functions.)

We will encounter some other kinds of static errors, and we will consider those in more detail as they become more important.

Recursive Functions and the Design Recipe

We will use the following recipe:
1. Problem Statement and Contract
2. Data Definition
3*. Examples as Tests (Use check=)
4. Function Template
5*. Function Definition
6. Make it Total(but keep it simple!)
3a*. Add tests of the function outside the contract/intended input, to check
    for totality (use CHECK or CHECK=)

Note: The items starred are not written as comments and are lisp code.

(*Problem Statement*)
Find the largest number in a list of natural numbers

```
(*Data Definition*)
What are the 2 datatypes we are talking about?
- nat (primitive type)
- nat-list : nil | (cons nat nat-list)
Since this is very common, we also say
nat-list: (listof nat)

(*Contract*)
max-list: nat-list -> nat

(*Examples as Tests*)
(check= (max-list '(1 2 3 2 1)) 3)
(check= (max-list '(5)) 5)
(check= (max-list nil) 0)
(check= (max-list 5) 0) ;see 3a

(*Function Template*)
(defun max-list (lon)
 (if <BASE CASE>
   <handle base case>
   <do something with (first lon)
       and (max-list (rest lon)) >
       ))

(*Function Definition*)
; MAX-LIST: (listof nat) -> nat
(defun max-list (lon)
"Returns the largest element of the given list"
  (if (endp lon)
    0
    (if (< (max-list (rest lon)) (first lon))
      (first lon)
      (max-list (rest lon)))))
```

Now let us consider some more interesting examples:

```
  (max-list '(-3 -4))
```

What should it return? If we think of the function as returning the largest *number* in the list, the return value seems wrong:

```
  ACL2 p>VALUE (max-list '(-3 -4))
  0
  ACL2 p>
```

The contract calls for a list of natural numbers and we used that assumption when we decided to have the max-list of the empty list be 0. What this example returns is the maximum of -3, -4, and 0, which is 0.

Let's consider a somewhat large example:

```
  (check= (max-list '(0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
                      0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 5)) 5)
```

You might end up waiting a very long time for that to finish. How do we know it will finish? Well, for now, you have to figure that out for yourself. Do all your recursions decrease toward a base case? This one does. So why is it taking so long?

First of all, we aren't really concerned with efficiency in this class, so if you write something like that on a test, you will get full credit. Aside from being inefficient, it is logically a simple definition.

The reason for the inefficiency is that each call can make two recusive calls, the two instances of (max-list (rest lon)). Thus, (max-list '(0 5)) will call (max-list '(5)) twice. (max-list '(0 0 5)) will call (max-list '(0 5)) twice. Double the execution time enough times, and it becomes intractable.

What can we do about this? Well, in this case, the two recursive calls are computing the same value! Can't we reuse this value without asking to recompute it? That is what LET is for:

```
(defun max-list (lon)
  (if (endp lon)
    0
    (let ((m (max-list (rest lon))))
      (if (< m (first lon))
        (first lon)
        m))))
```

The form of a LET is

```
  (let ((<var1> <expr1>)
        (<var2> <expr2>)
        ...)
    <body>)
```

and the meaning is that it executes <expr1>, <expr2>, ... and binds the values returned to <var1>, <var2>, ... respectively in evaluating <body>.

In this case, (max-list (rest lon)) is evaluted and bound to the variable m in the expression (if (< m (first lon)) (first lon) m). I could have chosen
most any variable name instead of m. I just chose m to be short for "max".

But, there is a cleaner solution to this problem, and that is to write a function that takes the maximum of two numbers and use that to implement max-list:

```
; Actually, this is already defined in ACL2:
(defun max (x y)
  (if (> x y)
    x
    y))

; MAX-LIST: (listof nat) -> nat
; Returns the largest element of the given list
(defun max-list (lon)
  (if (endp lon)
    0
    (max (first lon)
         (max-list (rest lon)))))
```

Notice how close to the function template this defintion is!!

Using the auxilliary function has a similar effect as using the LET, because
the result of the recursive call (max-list (cdr lon)) gets bound to the parameter y in MAX and is potentially used twice: once in comparing and possibly once in returning a value. But because it was bound to a variable or parameter, the recursive call is only made once.

Various versions of app (X Y)

```
; Version 1 recurring on the the first list x
; app : TL x TL -> TL
(defun app (X Y)
  (if (endp X) ; base case
    Y
    (cons (first X) (app (rest X) Y))))


; version 2 : recurring on the the first list but working from the end
(defun app (X Y)
  (if (endp X)
    Y
```

```
    (app (but-last X) (cons (last X) Y)))))

; NOTE but-last is different from the inbuilt "butlast" ACL2 function which
has the signature butlast (x n)

; version 3: recurring on second argument
(defun app (X Y)
    (if (endp Y)
      X
      (app (app X (list (first Y))) (rest y))))
```