CS 2800: Logic and Computation Fall 2010

16 September 2010

Lecture Outline:

- ACL2 Basic Data Types
- Expressions
- Evaluation

1 ACL2 Basic Data Types

ACL2 supports 5 disjoint kinds of data objects:

- numbers: `0, -123, 22/7`
- characters:

  `#\A, #\a, #\$, #\Space`
- strings: `"This is a string."`
- symbols: `nil, x, address, len`
- conses: `(1 2 3 4), (a b c), ((a 1) (b 2))`

Here is an informal description of the logical construction of the numbers. The non-negative integers (i.e., the naturals) are built from 0 by the successor function.

Nat := 0 | (Succ Nat) where Succ can be defined as (+ 1 *arg*) The negative integers are built from the naturals by the negation function. The rationals are built from the integers by division. The complex numbers are built from pairs of rationals.

Two commonly used symbols are `t` and `nil`. These are the Boolean symbols and are used to denote true and false in ACL2. However, all ACL2 primitive conditional and propositional operators test against `nil`. Any object other than nil may serve as an indicator of truth. When we say some test is true we mean that the value returned is not nil. The symbol `t` is just a convenient choice.

Conses are objects constructed using the 'cons' operation:
Cons := (cons All All)
where All is any ACL2 data object. Remember that non-empty Lists are Conses, but not all Conses are Lists,
i.e., Set of All non-empty Lists $\in$ Set of all Conses.
True-List := nil | (cons All True-List) Note that `nil` is a symbol, and has overloaded meaning, it stands for the boolean false, but also for the empty true-list and is sometimes written as (). Therefore the empty list is a symbol, not a cons.

The first four datatypes are also called atoms. Structurally complex objects(non-atoms) are constructed using the `cons` operation. So one can broadly say the ACL2 Universe has only atoms and conses.

2 Expressions

ACL2 programs are composed of *expressions*, also called terms.

In CS2500, you learned about the "man in the machine": Understand the language so well, that you can, step by step, evaluate any expression you are given.

We now introduce the core of the language, mainly **simple expressions**:

A *simple expression* is one of:

- a variable symbol
- a constant symbol

- a constant expression, or
- application of function symbol of n arguments to n simple expressions $a_1, \ldots, a_n$,written as $(f \quad a_1 \ldots a_n)$

Comments may be written where whitespace is allowed. A comment begins with a semi-colon and ends with the end of the line. Here is an example expression.

```
(if (equal date '(august 14 1989))          ; Comments are written
     "Happy birthday, ACL2!"                ; like this.
  nil)
```

This expression contains the variable symbol `date`, the constant symbol `nil`, two constant expressions (a list and a string), and two function applications (of the function symbols `if` and `equal`).

3 Evaluation

What does an expression mean? Its meaning is given by its value. What is the value of an expression? It is the ACL2 Object it **evaluates** to. The **value** of an expression depends upon the history of the session in which it occurs. *e.g.*, whether an expression can be evaluated or not depends on which functions have been defined. We leave the history implicit in our discussion here.

Given a history (assignment of ACL2 objects to its variable symbols and all functions defined in the history), an expression can be evaluated. The value is **always** an ACL2 object. In this discussion, we let the history be implicit. But you can think of the history as everything before the TODO LINE in the ACL2s editor, you remember the grey area, everything in the grey area is in the history of current session.

Let `eval` stand for the ACL2 runtime procedure which evaluates expressions: Value(*expression*) = (`eval` *expression*) Note: `eval` is a runtime procedure, used by ACL2 to evaluate expressions, its not available to the users directly. Consider the session editor window(in ACL2s) the *eval window*, where you can ask ACL2 to `eval` any input expression for you.

A variable symbol is a symbol other than a constant symbol (defined below). For example, x, entry, and address-alist are variable symbols. Variable symbols are given values by the (implicit) assignment.

A constant symbol is `t`, `nil`, or a symbol declared with defconst(like `*PI-APPROX*` in hwk2). The value of `t` is the ACL2 object t; the value of `nil` is the ACL2 object nil. Symbols declared with defconst have names that start and end with asterisks. The value of such a symbol is some ACL2 object specified by the defconst declaration. For example, `*PI-APPROX*` is a constant symbol in hwk2. Its value is the rational number 22/7.

A constant expression is a number, a character, a string, or a single quote mark (') followed by an ACL2 object. In the last case, we call the constant expression a *quoted constant*. The value of a number, character, or string is itself. The value of a quoted constant is the ACL2 object quoted. Thus, `123` and `"Error"` are constant expressions with themselves as their values. `'Monday` is a quoted constant whose value is the symbol Monday. `'(August 14 1989)` is a quoted constant whose value is a list of three elements, the first of which is the symbol august. Note that `'123` and `'"Error"` are quoted constants and have 123 and "Error", respectively, as their values. More generally, when writing numeric, character, and string data in expressions the quote mark is optional. We generally do not quote numbers, characters, or strings since they evaluate to themselves anyway. But when symbols (other than `t` and `nil`) are used as data in an expression they must be quoted because otherwise they might be confused with variables. That is, the expression `x` is a variable whose value is specified by the context(history). The expression `'x` is a constant expression whose value is the symbol x.
Value(x) = history(x) whereas
Value('x) = the symbol x

Cons objects must be quoted because otherwise they might be confused with function applications. We will return to this point in a moment.

The quoted constant `'`α may also be written `(quote` α`)`. The symbol `quote` is not a function symbol but a special marker indicating that its argument is to be taken literally rather than treated as an expression.

Recall the definition of what a simple expression is, there was only one case which was recursive in its definition, that was the function application, for which the following fact holds:

The value of an expression(function application) is given recursively in terms of the values of the subexpressions(arguments of the function application)

The value of a function application, ($f \quad a_1 \ldots a_n$), under a given variable assignment is described in terms of the values, c_i , of the a_i under that assignment. *i.e.*, Let Value(a_i) = c_i
And lets say `f` is defined as:
(**defun** `f` ($v_1 \ldots v_n$)
    body)

where *body* is a simple expression. Value(`f` $a_1 \ldots a_n$) = Value(*body*) under the assignment $v_i = c_i$.

Some functions are primitive, and they are treated differently, for each primitive function symbol, ACL2 associates a mathematical function that it directly uses to evaluate a function application. In the above example, `equal` is a primitive function symbol of two arguments. `if` is a primitive function symbol of three arguments. The function associated with `if` returns its second argument or third argument depending on its first argument. In particular, if the first is `nil`, the function returns its third argument; otherwise, it returns its second. The value of the simple expression

```
(if (equal date '(august 14 1989))             ; Comments are written
       "Happy birthday, ACL2!"                  ; like this.
   nil)
```

is either the string `"Happy birthday, ACL2!"` or else the symbol `nil`, depending on whether the value assigned to the variable symbol date is the list `(august 14 1989)` or not.

Many novice ACL2 programmers are confused about when to use the single quote mark. If numbers, characters, and strings evaluate to themselves, why not treat list constants similarly? The reason is that list constants can look just like function applications. We now return to the question of why we must write single quote marks before list constants. Reconsider the simple expression above, except remove the single quote mark.

```
(if (equal date (august 14 1989))         ; Comments are written
    "Happy birthday, ACL2!"               ; like this.
   nil)
```

Note that the value of the subexpression `(august 14 1989)` in this expression bears no *a priori* relationship to the similar expression in the earlier example. Here, `(august 14 1989)` is treated as the application of the function `august` to the arguments `14` and `1989` and its value is not necessarily the list object of 3 elements, first being the symbol `august`, the second and third being numbers `14` and `1989`.

Two wonderful facts about ACL2 should be noted here. The first is that every expression is an ACL2 object. For example, we can write a program that operates on the list constant.

```
'(if (equal date (august 14 1989))            ; Comments are written
      "Happy birthday, ACL2!"                 ; like this.
    nil)
```

The second is that function and variable names, we use in expressions are *symbols*, *i.e.*, they satisfy `symbolp` predicate.