Some Basic ACL2-Lisp Functions

ACL2 Lecture 3

Warren A. Hunt, Jr.
hunt@cs.utexas.edu

Computer Science Department
University of Texas
2317 Speedway, M/S D9500
Austin, TX 78712-0233

February, 2026

REV, Accumulators, Complexity, and Tracing

In this lecture, we investigate some basic list and tree processing functions.

- Discussion of REV
- `TRACE$` of REV
- Using accumulators; tail recursion
- We define a way to sum the leaves of a tree with integer leaves.
- We define an ACL2 predicate that recognizes trees with integer leaves.
- We introduce `FLATTEN`, a function to flatten a tree into a list.
- We postulate flattening a tree without using `APP`.
- We will use `TRACE$` to animate our definitions.

REMINDER: Using examples, we are introducing functional programming.

The lack of side effects provides opportunities for analysis. Much of this course concerns the pursuit of such opportunities.

List REVerse

We can use APP to create a REVerse function.

Consider:

```
(defun app (x y)
  (if (atom x)
      y
    (cons (car x)
          (app (cdr x) y))))


(defun rev (x)
  (if (atom x)
      nil
    (app (rev (cdr x))
         (list (car x)))))
```

Does REV return a TRUE-LISTP?

Remark: in running text, we often write ACL2 terms in upper case because Lisp *up-cases* everything.

TRACE of REVerse

Let's *animate* the `REV` function.

```
(trace$ REV)
```

It appears linear in its performance, but what about `APP`?

```
(trace$ APP)
```

Does `REV` return a `TRUE-LISTP`?

Does reversing a list twice produce return the original input?

```
(equal (rev (rev x)) x)
```

Deeper Question: Should we trace `CONS`?

Using Accumulators; Tail Recursion

We see that REV may be expensive to evaluate.

What is the complexity of REV?

Let's consider the use of an accumulator.

```
(defun rev2-help (x acc)
  (if (atom x)
      acc
    (rev2-help
      ;; Trim off first element
      (cdr x)
      ;; Extend the accumulator
      (cons (car x) acc))))

(defun rev2 (x)
  (rev2-help x nil))
```

Let's trace things to see what happens.

Tree Copy

So far, our definitions have concerned only right-associated trees (lists).

Consider:

```
(defun tree-copy (x)
  ;; If no pair
  (if (atom x)
      ;; return the atom
      x
    ;; otherwise, pair a copy of
    (cons
      ;; the TREE-COPY of the left subtree, and
      (tree-copy (car x))
      ;; the TREE-COPY of the right subtree
      (tree-copy (cdr x)))))
```

Note that we *move* both left and right. Is this OK?

Does this proposed definition meet the **Principle of Structural Recursion** requirements?

A Tree with Integer Leaves

Can we define a function that recognizes a tree containing just integers?

Will we allow *empty* trees?

```
(defun tree-integerp (x)
  ;; If pair recognized
  (if (atom x)
      ;; do we have an integer?
      (integerp x)
    ;; otherwise, check
    (and
      ;; the left subtree, and
      (tree-integerp (car x))
      ;; the right subtree
      (tree-integerp (cdr x)))))
```

Are we under utilizing CONS? How many CONSes are needed to *hold n* atoms?

FLATTEN a Tree

Can we define a function that, from left-to-right, takes each tip of a tree and creates a list?

```
(flatten (cons (cons 1 2) (cons 3 4)))
==>
(1 2 3 4)
```

We want a list with all tree tips. So, let's append the left and right subtrees together.

What do we need to do?

Count the Number of Tips

Given the idiomatic use of right-associated CONS tress to represent collections of items, we define a way to count the number of items.

```
(defun len (x)
  ;; If pair recognized
  (if (consp x)
      ;; then, increment and continue
      (+ 1 (len (cdr x)))
    ;; otherwise, return zero
    0))
```

Does this definition work properly? What about this next function?

```
(defun count-tips (x)
  ;; If atom recognized
  (if (atom x)
      ;; One tip
      1
    ;; otherwise, sum tips in the left and right subtrees
    (+ (count-tips (car x))
       (count-tips (cdr x)))))
```

Appending Two Lists

Often, we want to combine the elements of one list with another.

Our lists are ordered, so we have to make a decision as to how we wish to combine two lists. For now, we just *attach* one list to the *front* of a second list.

```
(defun not (x) (if x NIL T)
(defun atom (x) (not (consp x)))
(defun app (x y)
  ;; If pair recognized
  (if (atom x)
      ;; then, just return Y
      y
    ;; otherwise, make a new pair
    (cons
      ;; of the first item in X
      (car x)
      ;; and APP of the rest of X with Y
      (app (cdr x) y))))
```

To define a recursive function, we have to demonstrate that something gets smaller with each recursive call.

So, what gets smaller?

Associativity of APP

When we have defined something, we often wish to consider properties of the functions we have defined.

For example, is APP associative?

```
(equal (app (app x y) z)
       (app x (app y z)))
```

Let's run some simulations and see what happens.

Can we establish this relationship once and for all?

Yes, by using the **Principle of Structural Induction**, we can prove that APP is associative.

Consideration of Structural Induction

Given the definition

```
(defun f (x)
  (if (consp x)
      (f (cdr x))
      t))
```

can you prove the theorem `(equal (f x) t)` using the logical machinery we have described above?

ACL2 supports inductive proofs. Its Induction Principle is quite general and involves the notion of the ordinals and well-foundedness.

For now, we will use a much simpler principle.

A substitution σ is a `car`/`cdr` *substitution* on `x` if the binding (image) of `x` under σ is a `car`/`cdr` nest around `x`.

The other bindings of σ are unrestricted. For example, σ = {`x` $\leftarrow$ `(car x)`, `y` $\leftarrow$ `(cons (cdr x) y)`} is a `car`/`cdr` substitution on `x`.

Principle of Structural Induction

Principle of Structural Induction: Let ψ be the term representing a conjecture. ψ may be proved by selecting an "induction" variable `x`, selecting a set of `car`/`cdr` substitutions on `x` $\sigma_1, \ldots, \sigma_n$, and by proving the following subgoals:

```
Base Case:
(implies (not (consp x))
         ψ)
```

and

```
Induction Step:
(implies (and (consp x)           ; test
              ψ/σ_1                ; induction hypothesis 1
              ⋮
              ψ/σ_n)               ; induction hypothesis n
         ψ)                        ; induction conclusion
```

Here is an example Induction Step.

```
(implies (and (consp x)
              ψ/{x ← (car x), y ← (app x y)}
              ψ/{x ← (cdr (cdr x)), y ← (cons x y)}
              ψ/{x ← (cdr (cdr x)), y ← y})
         ψ)
```