Some Basic ACL2-Lisp Functions

ACL2 Lecture 2

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

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

January, 2025

Lists, Concatenation, Reversing, and Tracing

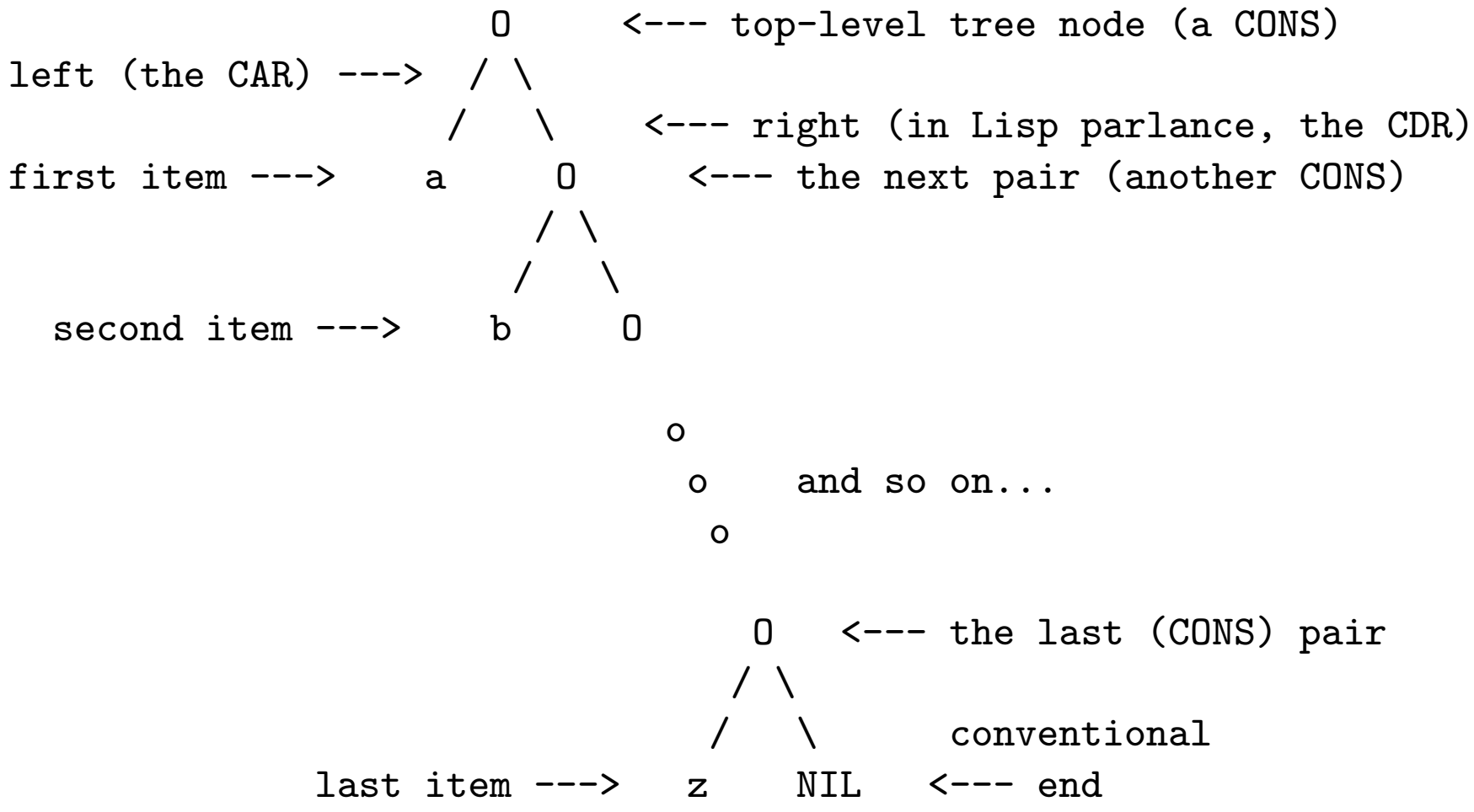
In this lecture, we investigate some basic list processing functions.

- We define an ACL2 predicate that recognizes a list integers.
- We specify the `APP` concatenation function.
- We introduce `REV`, a function to reverse a list.
- We use `TRACE$` to investigate the `APP` and `REV` function.

At the end, we provide some challenge problems.

Aggregating a Collection of ACL2 Objects

We will use lists to create collections of ACL2 objects.



where each item (a, b, ..., z) are the elements.

A Proper List

As we have only constructor (pairing) function, `CONS`, we are obliged to use it for all of our data structures.

That is, all ACL2 data objects are nothing more than `CONS` trees with atoms at the tips.

A well-formed, right-associated list (another Lisp idiom) is often called a proper list, but is known in ACL2 as `TRUE-LISTP`.

```
(defun true-listp (x)
  ;; If pair recognized
  (if (consp x)
      ;; then, check the right branch
      (true-listp (cdr x))
    ;; otherwise, require  NIL
    (eq x nil)))
```

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

The LENgth of a List

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

That is, all ACL2 data objects are nothing more than CONS trees with atoms at the tips.

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

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?

A Gentle Introduction to ACL2

Moore's *A Gentle Introduction to ACL2 Programming* can now be investigated.

One thing to note is Moore's use of the `COND` macro.

```
  :trans (cond ((atom x) 0)
               ((natp x) (- x))
               ((symbolp x) (cons x x))
               ((characterp x) (char-code x))
               (t x))
==>
  (IF (ATOM X)
      '0
      (IF (NATP X)
          (UNARY-- X)
          (IF (SYMBOLP X)
              (CONS X X)
              (IF (CHARACTERP X)
                  (CHAR-CODE X)
                X))))
```

Can you REVerse a List with APP

Here is a challenge problem: Using APP as a helper function, can you define a function that will reverse a list?

If so, what will its time complexity be?

Can we use `TRACE$` to help figure this out?