```
CSU290 Lecture Notes              Lecture 11                 5 Oct 2008

                                                          Peter Dillinger
```

(continuation of work on DINT function)

Recall what the DINT function is supposed to do:

```
; DINT: rational-list -> rational-list
; Returns the "discrete integral" of the input list.  That is, the returned
; list should be the same length and the nth element should be the sum of
; all elements of the input list up to and including the nth element.
; Examples:
;   (dint '(6 5 3 1)) = '(6 11 14 15)
;   (dint   '(5 3 1)) =   '(5  8  9)
;   (dint     '(3 1)) =     '(3  4)
;   (dint       '(1)) =        '(1)
;   (dint        '()) =         '()
```

We saw how we could break down this problem in the traditional way, by recursing on the cdr and using that solution in solving the original problem.

There are other ways we can take apart and solve this problem. One thing to notice is that the value of each element of the returned list depends on all those before it in the original list. In other words, the computation depends on prefixes of the list. On the other hand, ACL2 lists are constructed recursively off of suffixes; i.e. a suffix of a list is a list. This suggests to me that perhaps the problem would be easier if we tackle it backwards--by reversing the input and then reversing the output.

Suppose we write DINT as follows:

```
  (defun dint (x)
    (rev (dint-rev (rev x))))
```

where REV reverses a list and DINT-REV solves the problem with both the input and output reversed.

First let's look at REV:

```
; REV: true-list -> true-list
; Returns a list with elements in reverse order from the given list.
(defun rev (l)
  ?)

(check= (rev '(1 2 3)) '(3 2 1))
(check= (rev '(2 3)) '(3 2))
(check= (rev '(3)) '(3))
(check= (rev '()) '())
```

The contract calls for a true list to be passed in. That data definition suggests we would call REV on the CDR of l and use that in solving the problem on l. If l is, for example, '(1 2 3), the reverse is '(3 2 1), and the CDR is '(2 3), whose reverse is '(3 2). Clearly, if we add the first element to the end of the list returned by reversing the CDR of the list, we get the reverse of the whole thing.

How do we add a single element e to the end of a list l?

```
  (cons l e)
  (cons l (cons e nil))
  (append l e)
  (append l (cons e nil))
```

Which is it? Well if l is ’(3 2) and e is 1, then

```
  (cons ’(3 2) 1)                = ’((3 2) . 1)   Wrong!
  (cons ’(3 2) (cons 1 nil))     = ’((3 2) 1)     Wrong!
  (append ’(3 2) 1)              = ’(3 2 . 1)     Wrong!
  (append ’(3 2) (cons 1 nil))  = ’(3 2 1)       Correct!
```

Another way to write (cons e nil) is (list e).

What about base cases? Let’s examine the trivial cases. The reverse of the empty list is the empty list. The reverse of a list with a single element is that same list. Otherwise, the result will be different.

We could incorporate checks for both of those trivial cases:

```
  (defun rev (l)
    (if (endp l)                       ; use ENDP to check for empty list
      nil
      (if (endp (cdr l))
        l
        (append (rev (cdr l)) (list (car l))))))
```

If we try some examples, we see that we can simplify the function by getting rid of the (endp (cdr l)) case. Without it, each contract input will eventually reduce to the (endp l) case and compute the right answer from there:

```
  (defun rev (l)
    (if (endp l)
      nil
      (append (rev (cdr l)) (list (car l)))))
```

Why do we return nil in the case of (endp l) rather than just returning l? Well, if the input meets our contract of being a true list and (endp l) is true, then l must be nil, and it does not matter whether we return l or nil. If l does not meet our contract, then if we return nil rather than l, then REV will always return a true list, which is nice if we can get that without adding any IFs to the function.

Now to work on DINT-REV:

```
; DINT-REV: rational-list -> rational-list
; Returns the "discrete integral" of the input list, except starting from the
; end of the list.  The returned list should be the same length and the nth
; element should be the sum of all elements of the input list after and
; including the nth element.
;
; Examples:
;   (dint ’(1 3 5 6)) = ’(15 14 11 6)
;   (dint   ’(3 5 6)) =    ’(14 11 6)
;   (dint     ’(5 6)) =       ’(11 6)
;   (dint       ’(6)) =          ’(6)
;   (dint        ’()) =           ’()
```

Now that seems easier to solve than DINT alone was. All we have to do is sum up everything in the list starting at each point and put that on the returned list. I believe we have written a function SUM-LIST which will be useful in summing all the elements of a list.

```
(defun dint-rev (x)
  (if (endp x)
    nil
    (cons (sum-list x)
          (dint-rev (cdr x)))))
```

Accumulator Version

If we think about how we might solve the DINT problem by hand, we would keep track of the sum so far and use that in determining what value to put in the result list. For example:

Sum So Far	Remaining List
0	’(6 5 3 1)
6	’(5 3 1)
11	’(3 1)
14	’(1)
15	’()

And what we want to return is ’(6 11 14 15), so at each step, we should put the next sum on the result list. Let us write a helper for DINT which takes an extra parameter for the sum so far:

```
(defun dint-helper (sofar x)
  (if (endp x)
    nil
    (cons (+ (car x) sofar)
          (dint-helper (+ (car x) sofar)
                       (cdr x)))))
```