Going beyond lists of atoms — how to carry out inductions?

Why not contrive inducting on list length? The problem is brought out by an observation and an example:

observation: if we want to process a list of lists, then it will be necessary to process the sublists as well as any top-level atoms

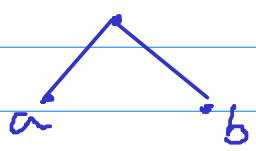
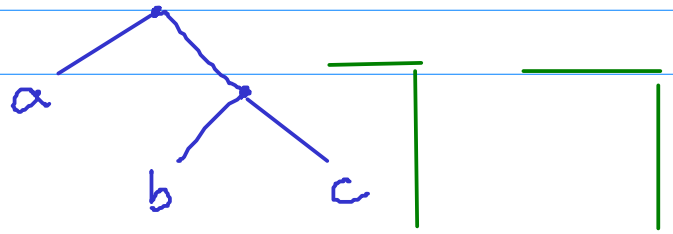
example: (1 (2 3 4 5 6) 7)

The entire list has length 3, but the cadr has length 5. If we were to try inducting on length, and if we wanted to use the IH to claim that our program works on all components of the top-level input, we run into trouble because the chosen induction metric is larger for the cadr component than for the original input.

The idea is to find a new induction metric. We will use "tree complexity", which we need to define.

Tree? What do trees have to do with lists of lists?

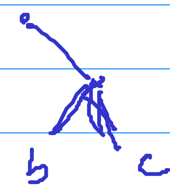
It turns out that there is a natural correspondence between these:

S-exp	$\xrightarrow{g}$	tree
a	$\longrightarrow$	
()	$\longrightarrow$	
(a)	$\longrightarrow$	
(a b)	$\longrightarrow$	
(a (b c))	$\longrightarrow$	

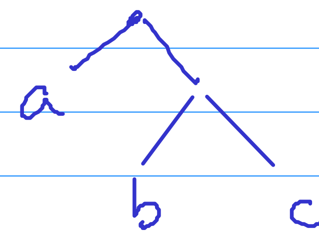
Add a link to reflect parens

The ops of car, cdr and cons can be defined on trees also

(car '(a (b c))) $\xrightarrow{\mathcal{J}}$ • a ← Tree-car

(cdr '(a (b c))) $\xrightarrow{\mathcal{J}}$  ← Tree-cdr

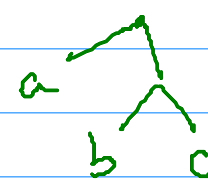
= ((b c)) — " (b c)

(cons 'a '(b c)) $\xrightarrow{\mathcal{J}}$ 

So you see that we can carry out these operations either in s-exp space or in tree space

eg (a (b c)) —car→ a

$$\begin{array}{ccc} \text{s-exp} & \xrightarrow{\quad \mathcal{J} \quad} & \text{Tree} \\ \text{car} \downarrow & & \downarrow \text{Tree-car} \\ \text{s-exp} & \xrightarrow{\quad \mathcal{J} \quad} & \text{Tree} \end{array}$$

 —Tree-car→ • a

Similarly for cdr, and for cons. [Aside: $\mathcal{J}$ is a functor]

The reason I'm showing you this?
Sometimes it is easier (simpler, more clean) to think in terms of trees when we're designing code — and it is necessary to know that this is 'mathematically' justified.

Questions

(1) What has happened to '() in the tree diagram for '(a)?

(cons 'a '()) in tree space is

(Tree-cons . a .) is

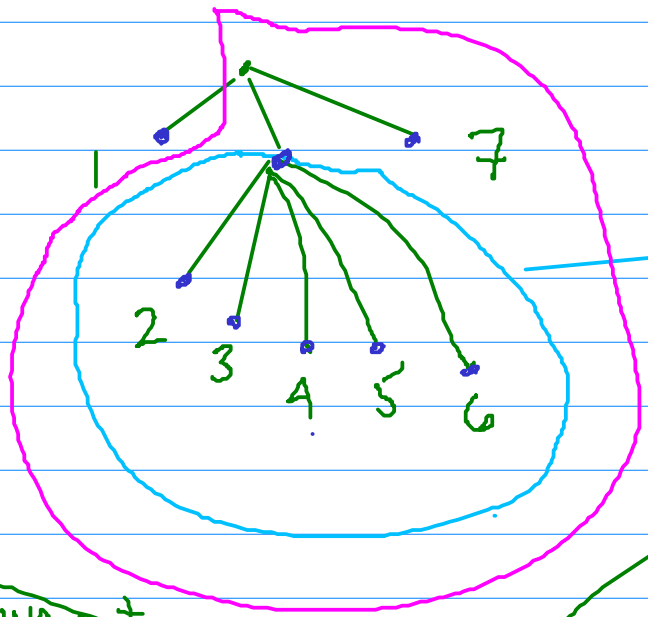
(2) What does this have to do with induction?

It gives us a useful metric: we can induct on the "size" of the tree corresponding to the sexp.

What do we mean by size? Some combination of height and width?

Let's look at our example

'(1 (2 3 4 5 6) 7)



In this case the height of the cdr is less than that of the entire tree.

Aside: how would we get the 3? cdr → (2 3)
cdr → (3)

Aside: (1 2 3)
cdr → (2 3)
car → 2

cm → 3

But height doesn't work for the cdr — the cdr has the same height as the entire tree.

Well — what about the number of nodes? Clearly smaller for each component than for the original.

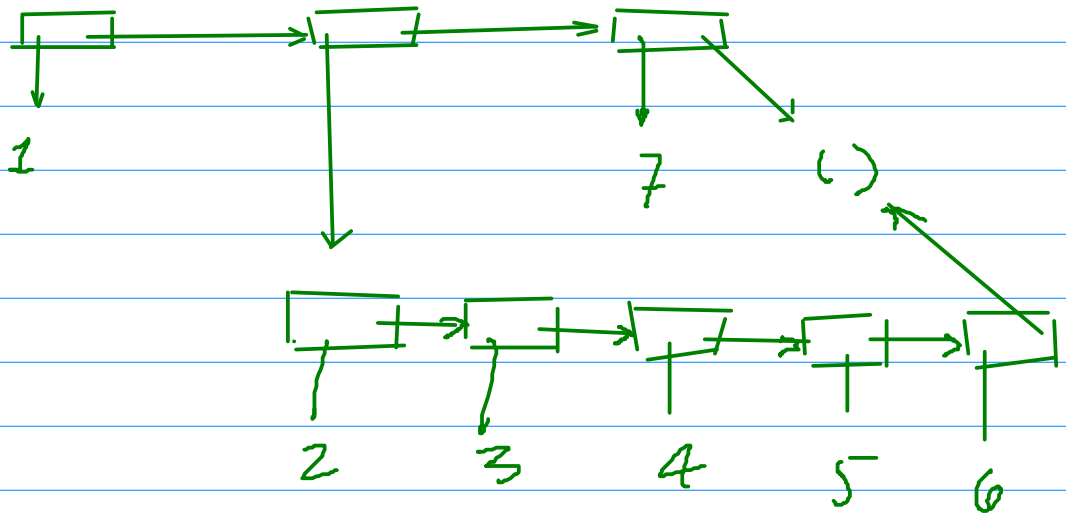
What about the number of leaves? Doesn't work:

(one leaf to one leaf)

Counting nodes is easy in tree space. Since we claim to have a correspondence between trees and sexps, we should be able to figure out what to count in sexp space as well.

What about counting the number of cons cells? Two questions: ① Is this the same as counting nodes? and ② does it work?

Again consider '(1 (2 3 4 5 6) 7) :



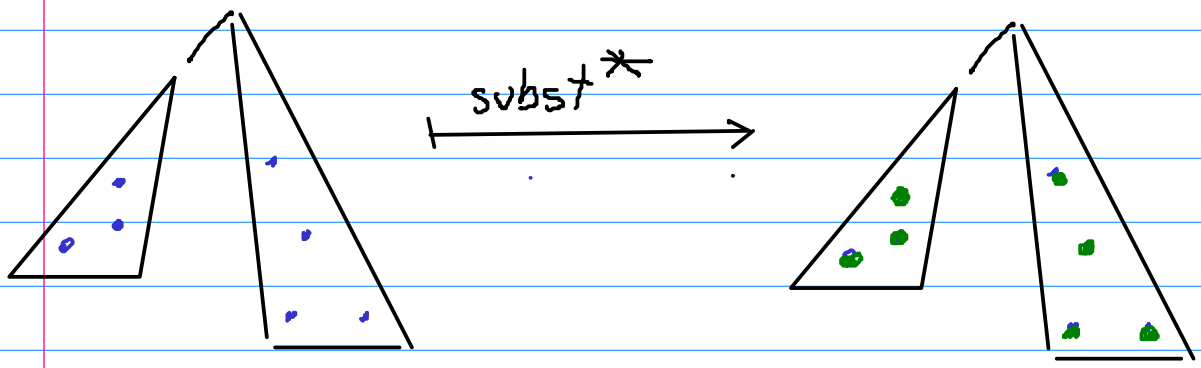
8 cons cells vs 9 nodes - so the answer to 1 is NO (but won't it always be off by 1, for the empty list?)

What about question 2? Looks good

Suggested Exercises ① Is it true that any well-formed component of a list has fewer cons-cells than the list itself? ② is the claim true? If so, then counting cons cells is "the same" as counting tree nodes.

Let's look now at an example of a program which works for general sexp inputs — I direct your attention to lecture9.scm (and to TLS)

We want a program subst* which replaces each occurrence of an atom old by an atom new, in the input tree



We use Tree-recursion — a standard pattern in which one issues recursive calls on both the car and the cdr, after dealing with the basis case(s)

That is: the divide and conquer decision is that we will process the car and the cdr separately.

Will pick up here next time.