CSc 335 Class 14 March 14 2024

Abbreviation for The Little Schemer — see the syllabus.

TLS — go ahead and start working through Chapters 1-6
A&S — see Chapter 2

S-expressions

An inductive definition in BNF (Backus-Naur form) for the class of S-exps is

empty list

sexp ::= atom | ( ) | (sexp ... sexp)

"or"

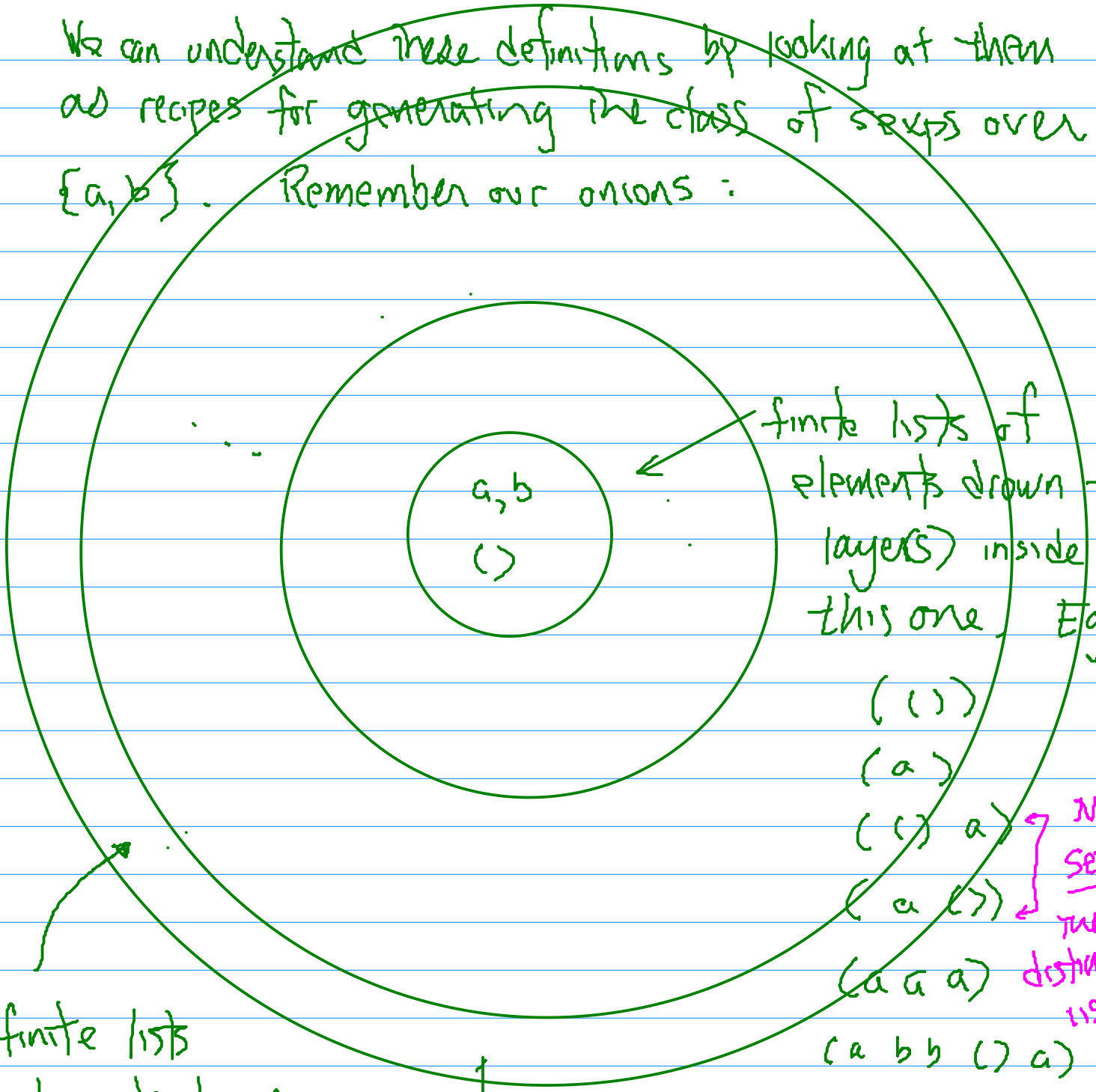
a finite non-empty list of Sexps

for now we will limit the class of atoms to just {a, b}.

In traditional math form, this definition might be given as follows:

> The class of Sexps over {a, b} is the least class containing the atoms a and b, as well as the empty list, which is closed under the operation of forming finite lists.

We can understand these definitions by looking at them as recipes for generating the class of sexps over {a, b}. Remember our onions:



The class sexp contains the class list. Since these BNF definitions are a big deal going forward,

Let's look at an exercise, for practice. The question is: is

list ::= atom | () | (atom ... atom) , *

where atom ::= a | b

a "good" definition of the class of lists over {a,b}?

What do we mean by "good"? There are two criteria;

→ soundness (ie - is every object produced by the recipe actually a list over {a,b}?)

→ completeness (ie - is every list over {a,b} produced by this recipe?)

I think you can see - intuitively - that * is not complete: There appears to be no way of generating - eg - any list which contains non-empty sublists.

strongly suggested: prove - by structural induction - that (*) does not generate lists with non-empty sublists.

Hint: how can the ith be applied? The key is to recognize that some list

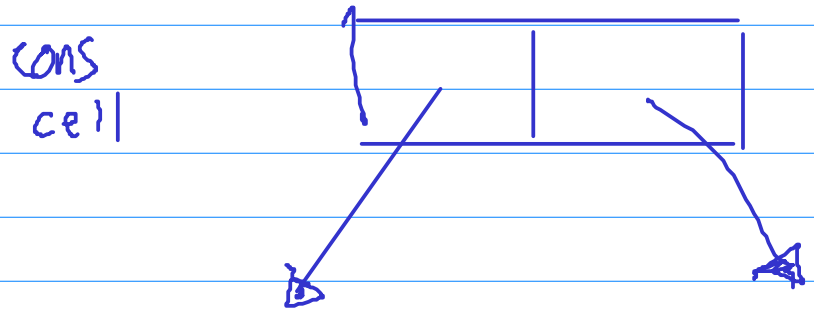
$(e_1\ e_2 \cdots e_k)$

is generated by * only if e_1 and e_2 and $\cdots e_k$ are generated by *.

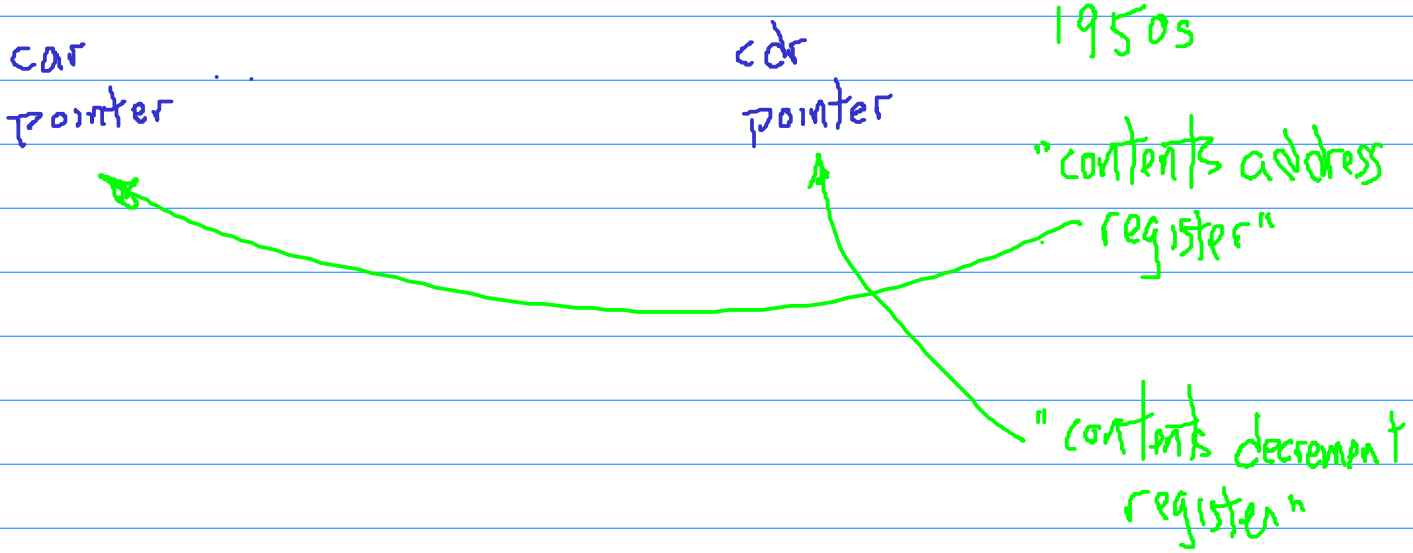
What about soundness? It certainly appears that every object generated by * is a list over $\{a, b\}$. But how could we prove this? It's very helpful to use a pictorial representation of lists - at this point, we will focus entirely on Scheme lists.

These pictorial reps are called box and pointer diagrams - you can find these discussed in Ch 2 of A&S.

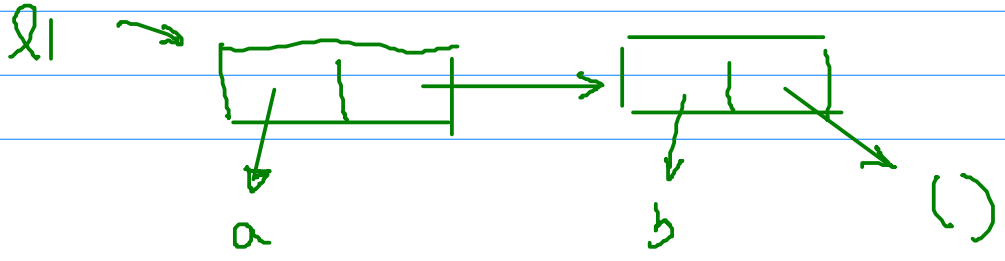
The basic building block of a box and pointer diagram is the cons cell. A cons cell is an assembly of two addresses (pointers)



Traditional terms from the original Lisp - late 1950s



A scheme list can be assembled using cons, which is a primitive. For example, the list (a b), box & pointed as



can be defined

```
(define l1 (cons 'a (cons 'b '())))
```

Aside on quotation

'a is shorthand for (quote a)

→ quote and ' are primitives

→ Their role is to block evaluation: if you enter a (without a quote, and without having previously defined it), you'll get an unbound variable error. But 'a just returns the symbol — a.

In other words, quote gives Scheme a means for dealing with symbolic data —

When working through TLS, you will need to add quote marks ahead of their symbols

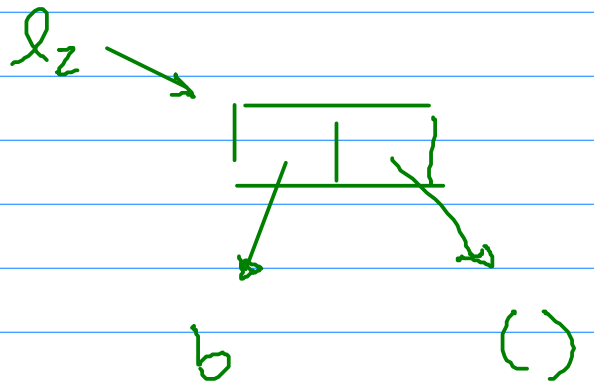
---

Back to cons ...

(define l1 (cons 'a (cons 'b '())))

If we take this apart – say

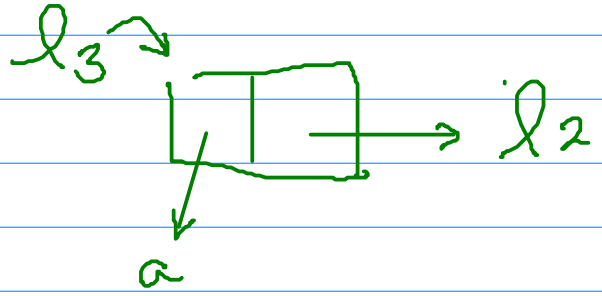
(define l2 (cons 'b '()))



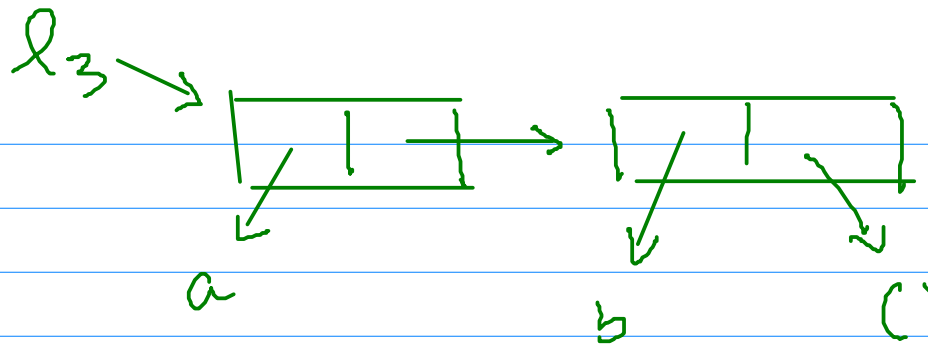
and then

(define l3 (cons 'a l2))

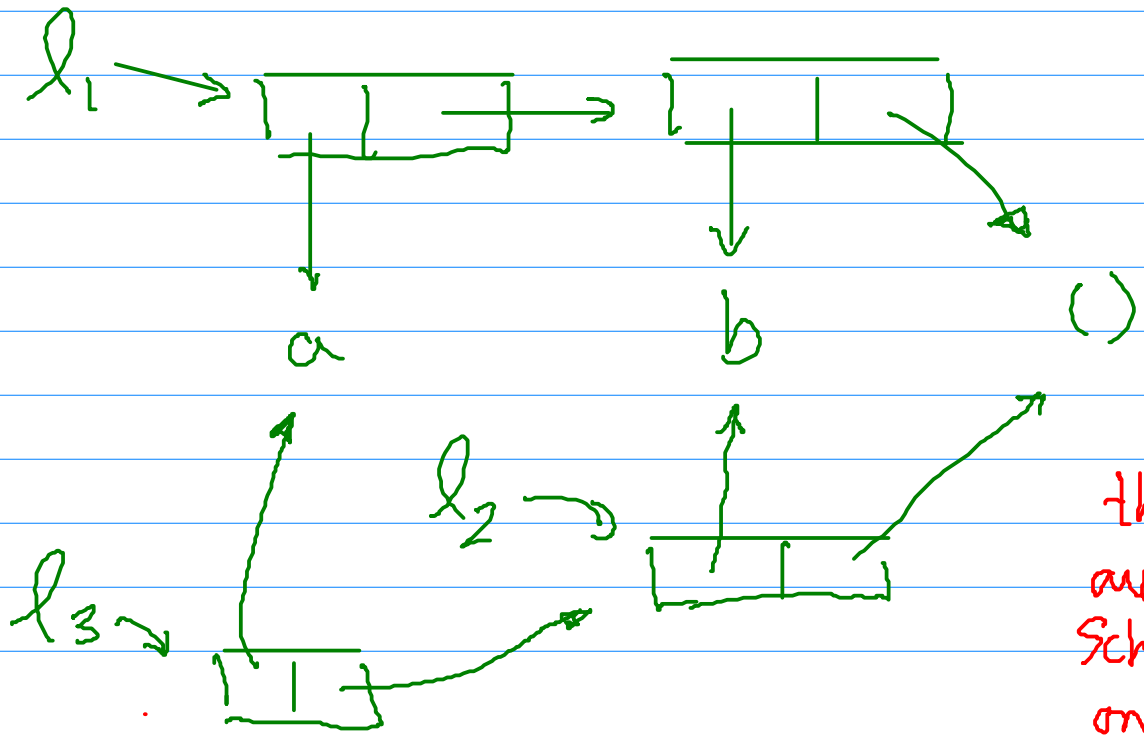
We get



which is



You can ask here: are l_1 and l_3 "the same"? The answer may be a bit surprising, because cons allocates new memory it is called. In fact, following the sequence of definitions just shown, we would have



atoms and the empty list are unique in Scheme: "There's only one letter b and only one ()"

Scheme has two equality tests of particular interest to us — equal? and eq?

If you experiment with this stuff, you will find

(equal? l1 l3) = #t

[same logical structure]

(eq? l1 l3) = #f

[physically different addresses]

Going a bit further, you'll see that

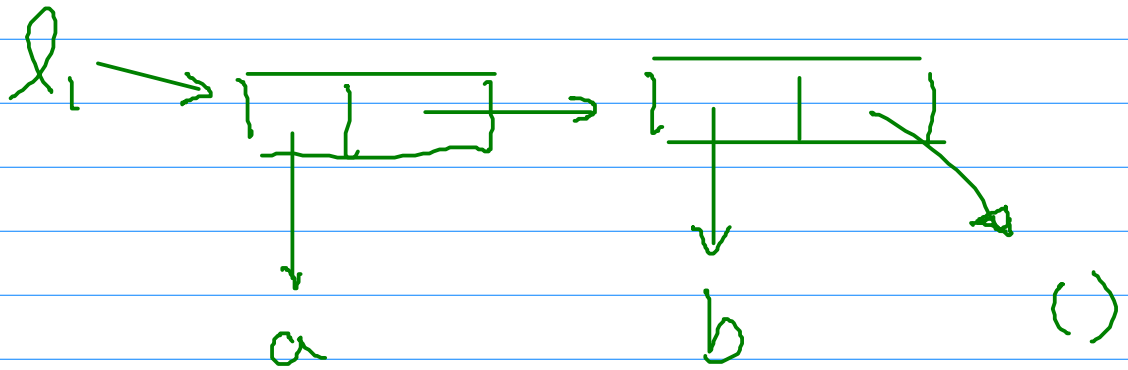
(eq? (car l1) (car l3)) = #t

↑ how to eval? Assuming that the value of l1 is a cons cell, (car l1) returns the value pointed at by the car pointer of that cell.

For datatypes, we will have constructors and selectors (and maybe classifiers as well)

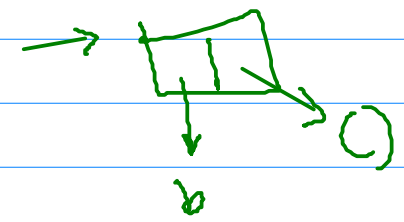
For the pair structures in scheme, the constructor is cons and the selectors are car and cdr. Not all pair structures are lists! (will come back)

We've seen that we can extract a as (car l_1)



How about extracting b? We'd get b via function composition:

(cdr l_1) = (b), ie

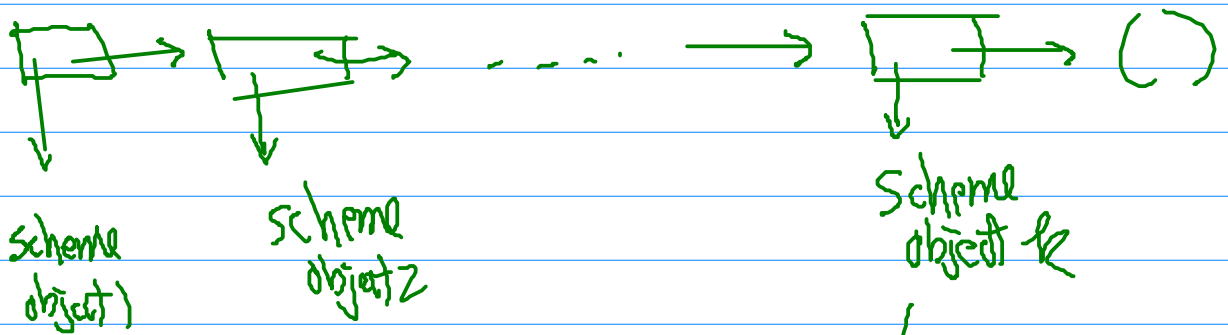


(cdr (cdr l1)) = b

This is an atom.

This is a list

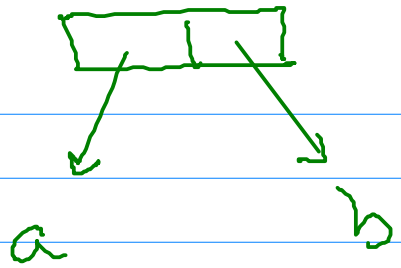
A scheme list – in terms of B&P diagrams – is any cons-cell structure with a "flat backbone" whose rightmost cdr is ()



which displays as

(so1 so2 ... sok)

On the other hand, (cons 'a 'b) is the structure



which displays as (a . b) — it is a dotted pair, not a list.