CSc 335 Class 15 March 19 2024

Some more on cons-cell structures. Lets start with an example.

Suppose we have the list l1

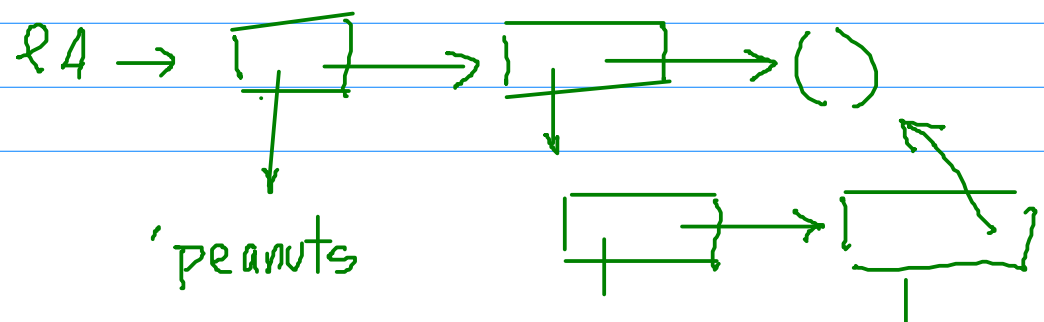
(I (had (bread and (peanuts (and yoghurt))))) (for (lunch)))

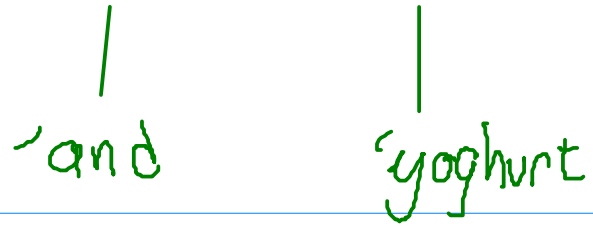
(1) How could this list be defined? How can components of the list be accessed? What is the scheme representation of this list?

(1) It would be error-prone to try to enter the list directly, all at once, from the keyboard. Better to do it one component at a time:

```
(define l4 (list 'peanuts
                 (list 'and 'yoghurt)))
```

sets up (in memory)





Although list (the primitive constructor) is convenient, there are other ways of building l4:

```
(define l4  (cons 'peanuts
                  (list (list 'and
                              'yoghurt))))
```

and

```
(define l4  (cons 'peanuts
                  (cons (list 'and 'yoghurt)
                        '())
                  ))
```

and

```
(define l4 (cons 'peanuts
                 (cons
                   (cons 'and (cons 'yoghurt '()))
                   '()))))
```

This last form exposes the underlying structure of l4, as shown above. We can use the

B&P diagram to understand how to access components of l4. Suppose, for example, that we wish to extract the component 'and using the datatype selectors car and cdr. Following pointer arrows in the B&P diagram, we see

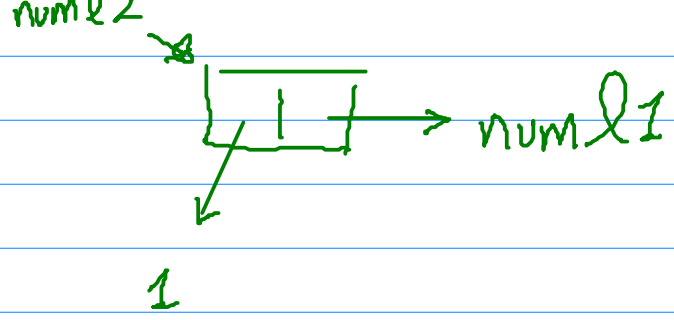
(car (car (cdr l4))) = 'and

Shorthand: (caadr l4)

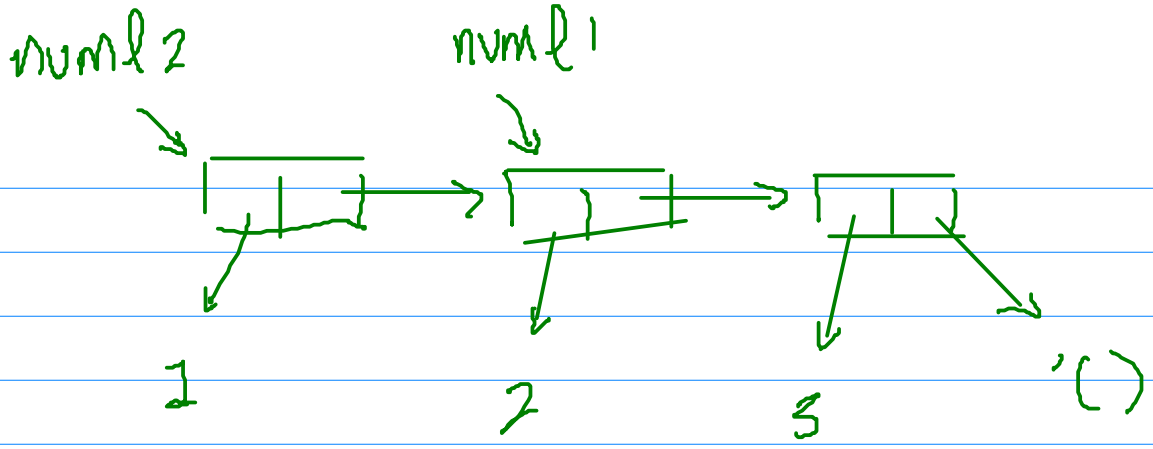
Recall: cons allocates a fresh cons-cell.

If we had (define numl1 (list 2 3))

Then (define numl2 (cons 1 (list 2 3))) returns



written out, this would be



(Remember – no need to quote numbers, as they are self-evaluating)

This is the idea for showing soundness of lst ::= atom | (atom ... atom)

assuming lists are built using cons and recalling that lists are built from flat cons-cell backbones with rightmost cdr

The empty list — you can see that consing an atom to a list of atoms again returns a list

i.e. everything the recipe "produces" is a list

Back to Q4 – to type it in directly, one could use quote

(define Q4 '(peanuts (and yoghurt)))

Note, however, that

(define lla '(peanuts (list 'and 'yoghurt)))

returns

(peanuts (list 'and 'yoghurt))

↑ unevaluated, due to the quote.

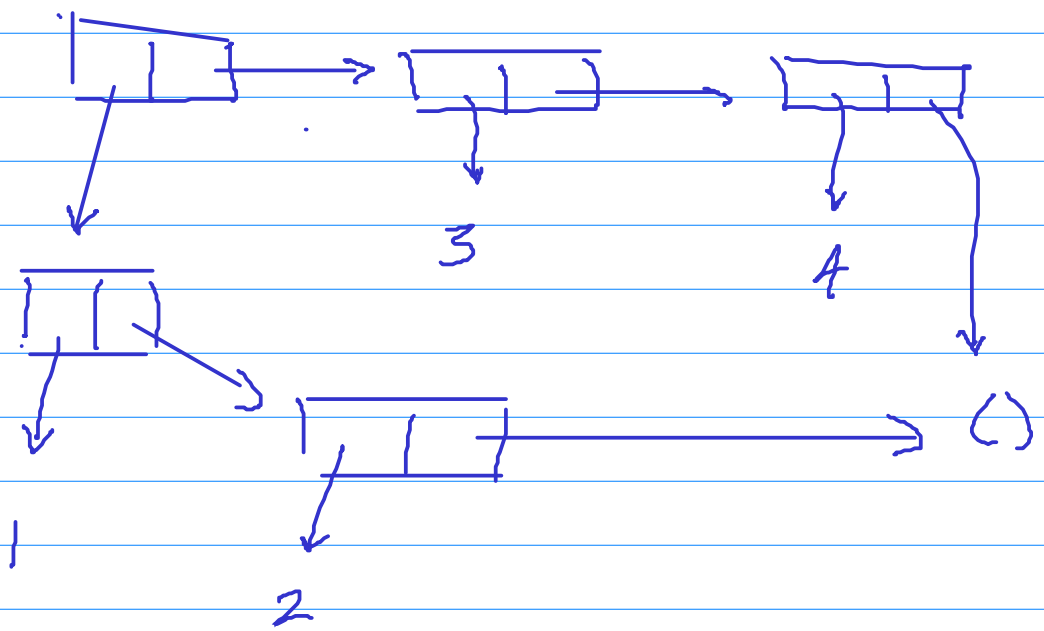
Exercise Figure out additional components of l1, and then how ① define l1, ② draw l1, and ③ access various components of l1.

Another example: if l1 and l2 are lists, what is the value of (cons l1 l2)?

Eg: l1 = '(1 2)

l2 = '(3 4)

(cons l1 l2) = ((1 2) 3 4)



There is another primitive list op called append:

(append l1 l2) = (1 2 3 4)

Lets look at some program developments using lists.

Consider a program which is to input a list, and output #t if the input is a list of atoms, and #f otherwise

Here, we will assume

list ::= ( ) | (cons atom list) | (cons list list)

and

lat ::= ( ) | (cons atom lat)

atom ::= a | b

The first definition describes the input data, and effectively tells us how to structure the program

```
(define (lat l)
  (cond ((null? l) #t)
        ((atom? (car l)) ———— ??)
        (else #f))))
```

recursive call on a shorter list - i.e. (lat (cdr l))

Check out the load primitive to avoid having to type def. of atom? each time you want it!

atom? is NOT a primitive – so we define it

```
(define (atom? x)
  (and (not (null? x))
       (not (pair? x))))
```

Here null? and pair? are primitives

$$(\text{null? } l) = \begin{cases} \#t & \text{if } l \text{ is the empty list} \\ \#f & \text{otherwise} \end{cases}$$

→ (what does scheme do if l is not a list?)

$$(\text{pair? } c) = \begin{cases} \#t & \text{if } c \text{ is a cons-cell} \\ \#f & \text{otherwise} \end{cases}$$

→ (what does scheme do if c is, say, a number?)

```
(define (lat l)
  (cond ((null? l) #t)
        ((atom? (car l)) ———— ??)
        (else #f))))
```

recursive call on a shorter list — i.e. (lat (cdr l))

Why that else-clause? How about

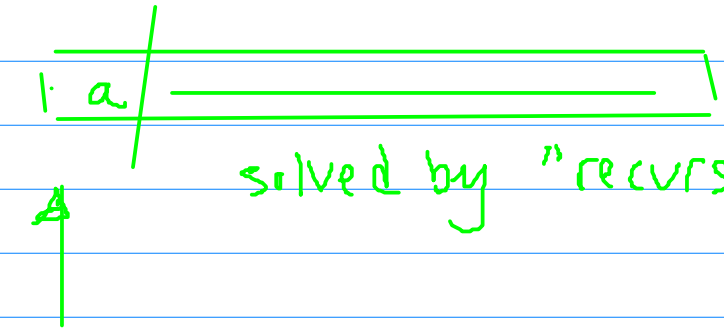
```
(define (lat l)
  (cond ((null? l) #t)
        (else (and (atom? (car l))
                   (lat? (cdr l))))))
```

cleaner. Simpler. More in line with the BNF.

pf by induction on the length of the input
(# elts in the list)

Strategy used: "cdring down the list" —

With this design idea, plus the idea of processing the tail of the list recursively we have our divide and conquer set up:



IH

processed by
checking whether it is an atom

IS

we know that the list is a lat precisely if BOTH the car is an atom and the cdr is a lat

So we use and, which is the deferred operator.