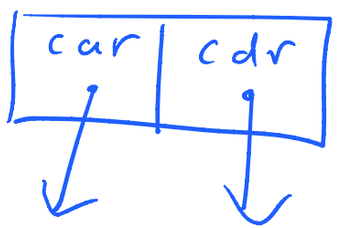
Scheme Lists

October 22, 2016 10:13 PM

Scheme lists are implemented as singly-linked lists, where each node of the list is a cons cell, i.e. an object with 2 pointers



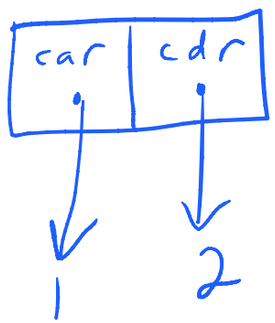
cons cell

also called a pair in Scheme

There is both a visual box notation,

a Scheme notation using dotted pairs.

e.g.



(2 . 3)

Scheme dotted pair notation

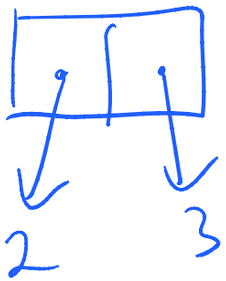
box diagram

(car '(2 . 3)) → 2

(cdr '(2 . 3)) → 3

eg (cons 2 3) creates a new cons cell with car 2 and cdr 3

with car 2 and cdr 3



(2 . 3)

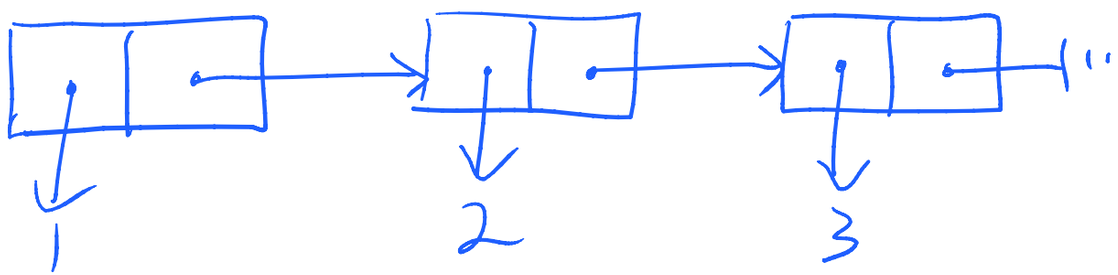
eg Lists are made from cons cells

(1 2 3) = (cons 1 (cons 2 (cons 3 '())))

= (cons 1 (cons 2 (3 . '())))

= (cons 1 (2 . (3 . '())))

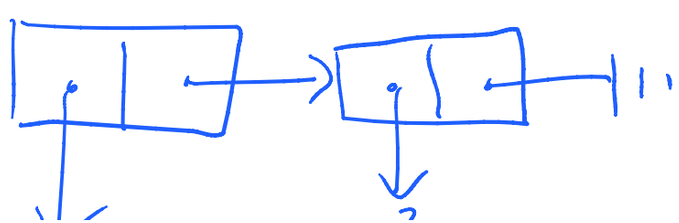
$= (1 . (2 . (3 . '())))$

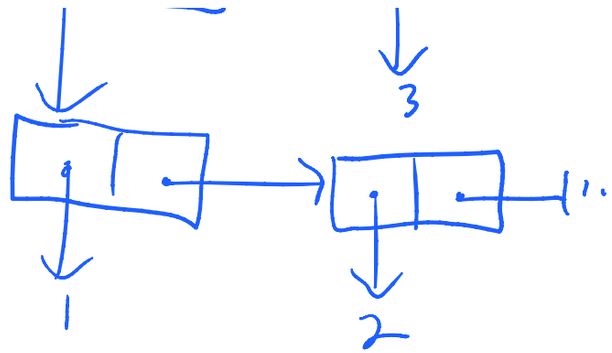


Box diagram

e.g. $((1\ 2)\ 3) = ((1\ 2) . 3)$

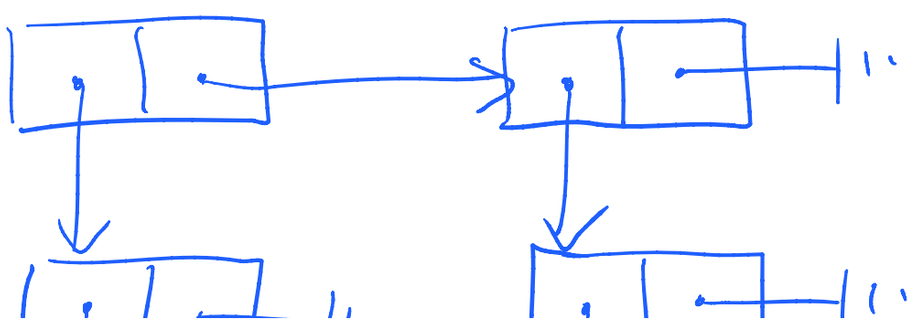
$= ((1 . (2 . '())) . 3)$

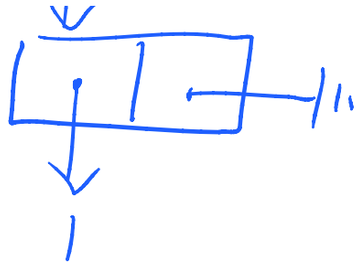


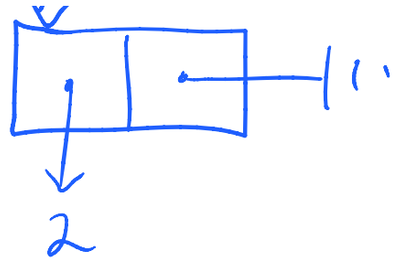


eg $((1)(2)) = ((1).((2).\,'()))$

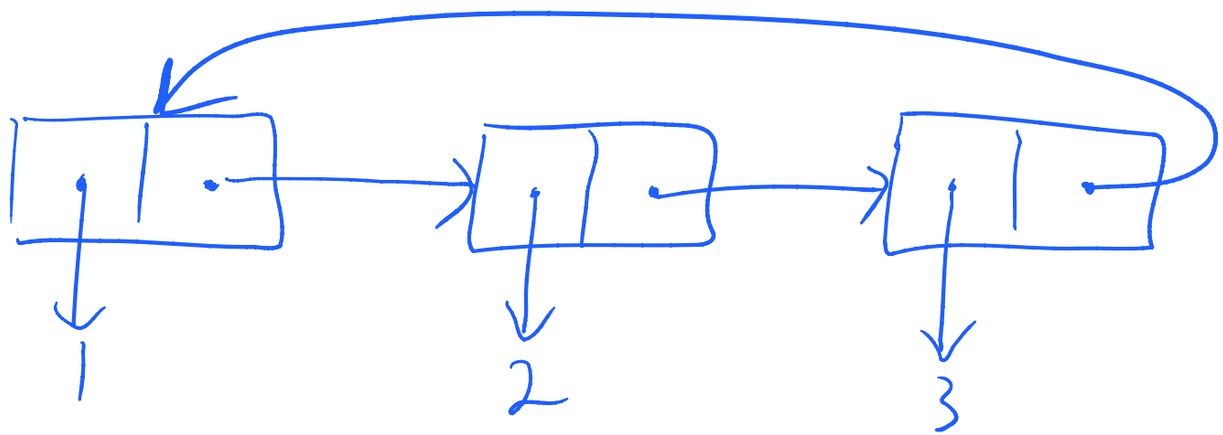
$= ((1.\,'())\,.\,((2.\,'())\,.\,'()))$







eg Cyclic data structures are possible:



In Scheme, you need to use modification functions like (set-car! pair obj)

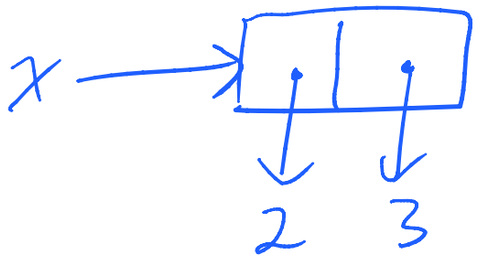
functions like (set-car! pair obj)

or

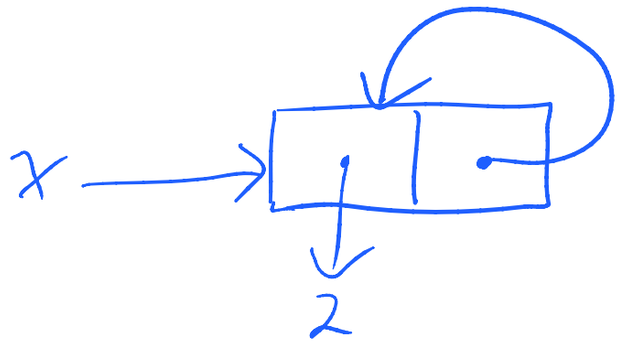
(set-cdr! pair obj)

scheme functions ending with ! are functions that modify (mutate) lists. Functions that don't end with a ! do not modify lists.

eg (define x (cons 2 3))

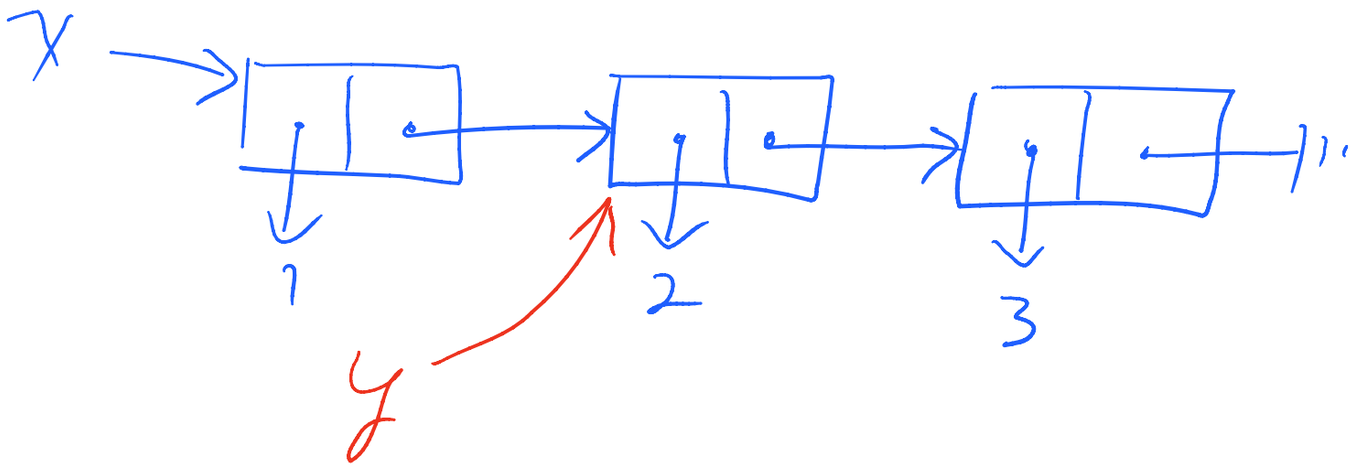



```
(set-cdr! x x)
```



Careful! Cyclic data structures cause ordinary Scheme functions to loop forever!

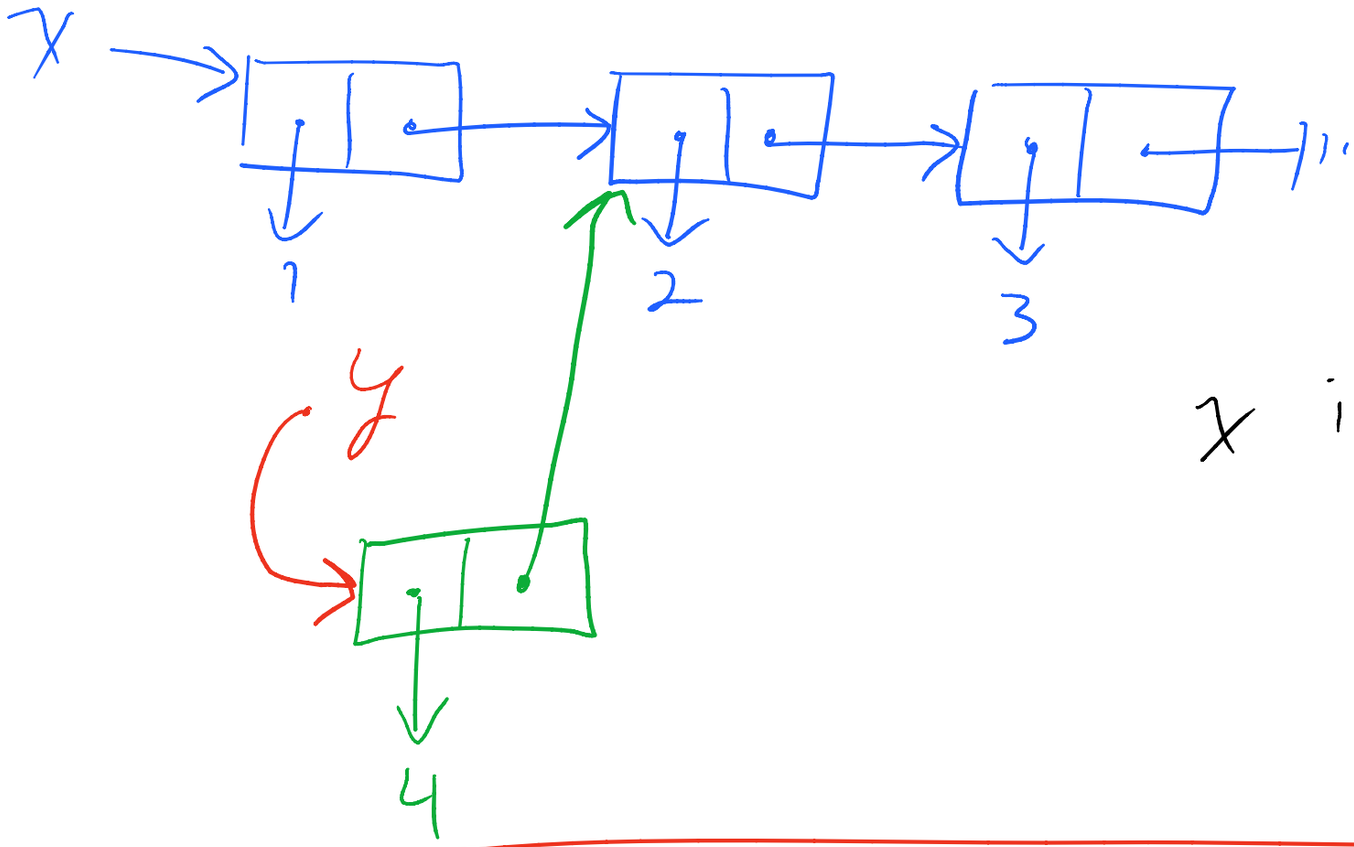
eg (define x '(1 2 3))



x and y share some of the same cells

(define y (cdr x))

(define z (cons 4 y))



x is still (1 2 3)

Unless you use mutating functions that end with !, like set-car! and set-cdr! there is no way to

set-cdr!, there is no way to modify or change existing cons cells.