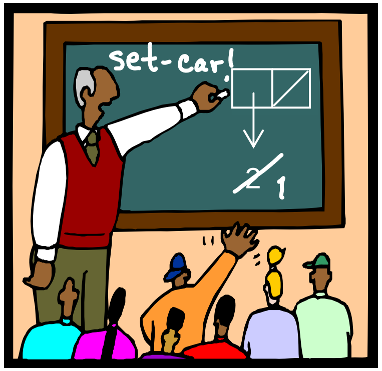
6.001 recitation 3/21/07

- set-car! and set-cdr!
- ring problems
- more set-car!, set-cdr! problems



Dr. Kimberle Koile

compound data mutation

constructor:

`(cons x y)`	**creates a new pair**

selectors:

`(car p)`	**returns car part of pair**
`(cdr p)`	**returns cdr part of pair**

mutators:

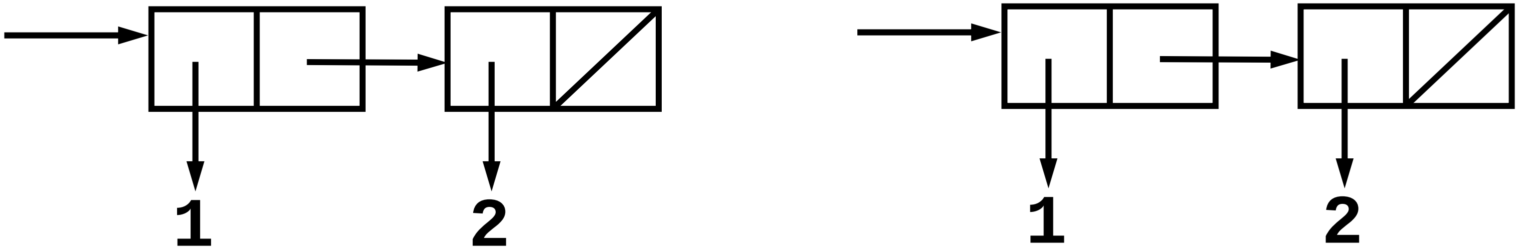
`(set-car! p new-x)`	**changes car pointer in pair**
`(set-cdr! p new-y)`	**changes cdr pointer in pair**
`; Pair,anytype -> undef`	**-- side-effect only!**

sharing, equivalence, and identity

How can we tell if two things are equivalent?

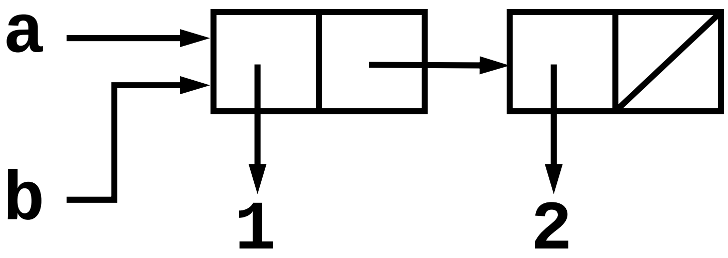
-- What do you mean by "equivalent"?

1. **The *same object*: test with** `eq?`
   ```
   (eq? a b) ==> #t
   ```
2. **Objects that *"look" the same*: test with** `equal?`
   ```
   (equal? (list 1 2) (list 1 2)) ==> #t
   (eq? (list 1 2) (list 1 2)) ==> #f
   ```



example 1: pair/list mutation

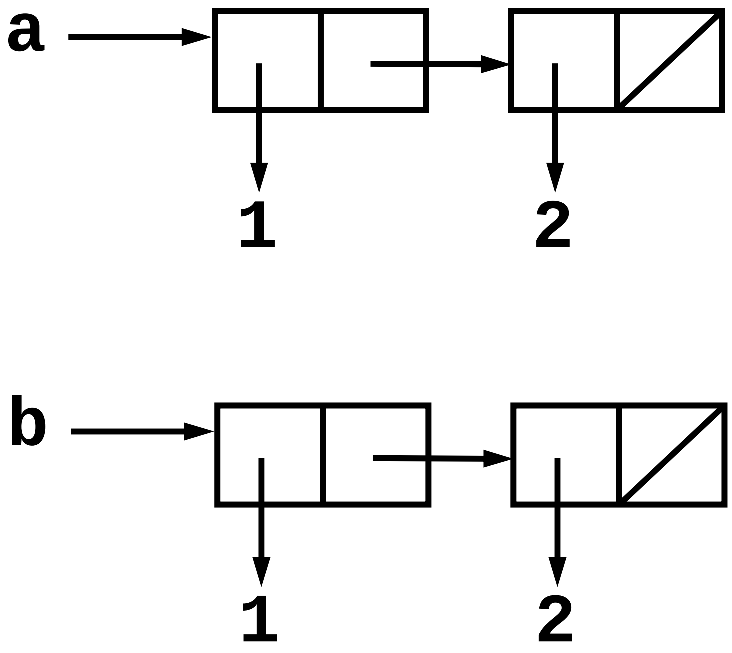
```
(define a (list 1 2))
(define b a)
 a ==> (1 2)
 b ==> (1 2)
```



```
(set-car! a 10)
```

Compare with:

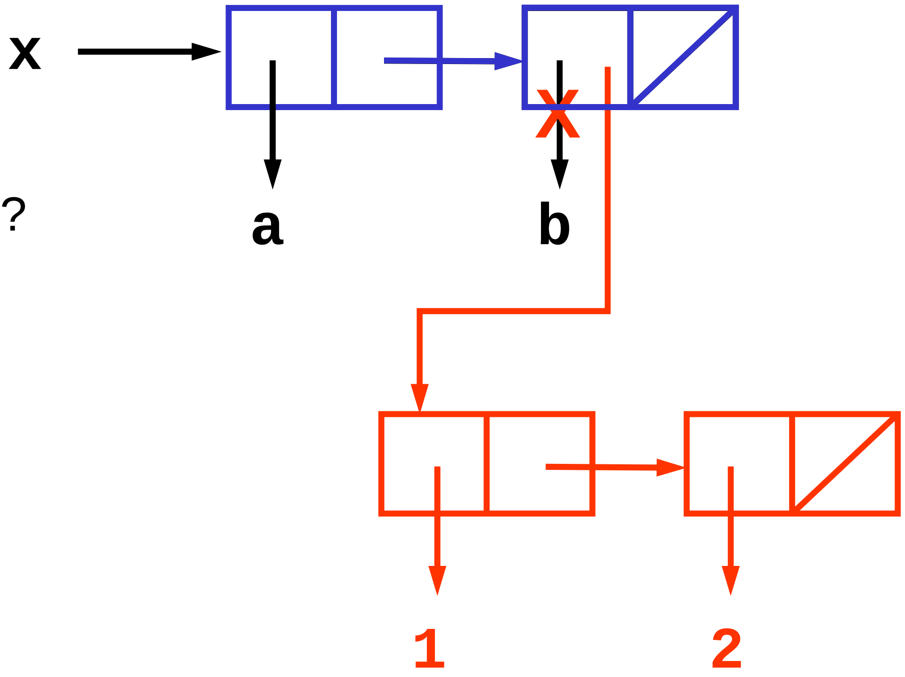
```
(define a (list 1 2))
(define b (list 1 2))
```



```
(set-car! a 10)
```

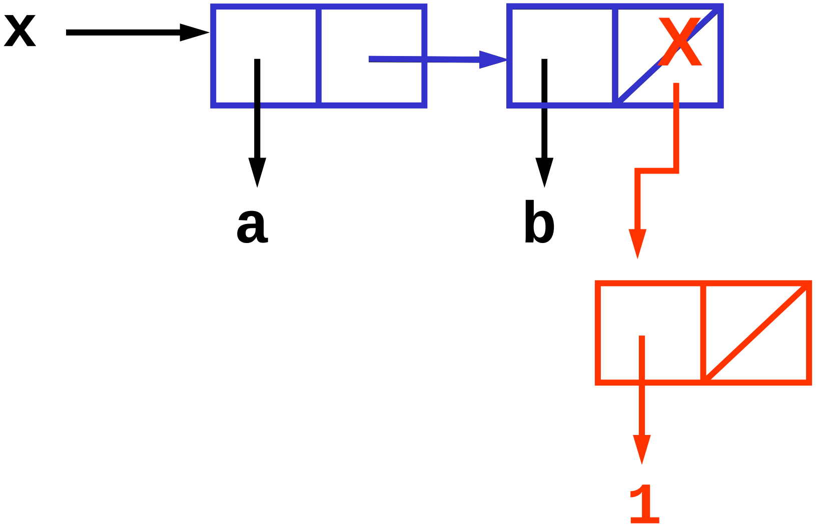
example 2: pair/list mutation

```
(define x (list 'a 'b))
```



How is x mutated to achieve the result at right?

And this one?



set-car! and set-cdr! problems

For the given expressions:
(a) Draw the box and pointer diagram corresponding to the list or pair structure
(b) Write what Scheme prints out after evaluating the last expression in the sequence

1. (define x (cons 7 (list 8 9)))
   (set-car! (cdr x) 10)

a. box and pointer diagram for x

b. printed result for x

set-car! and set-cdr! problems

For the given expressions:
(a) Draw the box and pointer diagram corresponding to the list or pair structure
(b) Write what Scheme prints out after evaluating the last expression in the sequence

```
2.   (define y  ’(7))
     (define z  (let ((x (list ’a ’(b c)   (car y))))
                      (set-car! y (cdr x))
                      (set-cdr! x (car (cdr x)))
                      x))
     z
```

a. box and pointer diagram for x, y and z

b. printed result for z

more set-car! and set-cdr! problems

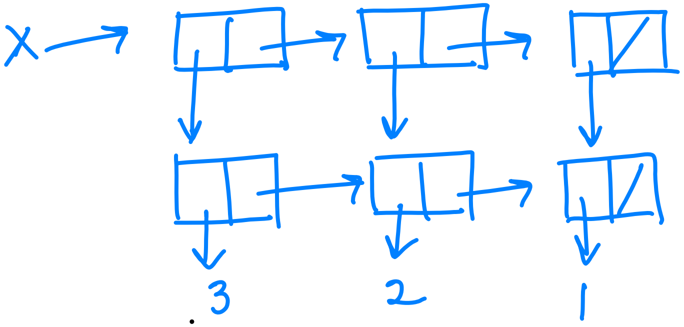
For the box & pointer diagram:

(a) Write what Scheme prints out for the structure (if it can)

(b) Write a Scheme expression that makes the structure (if an error, describe it)

(c) Draw the structure that results from the mutation, and its printed representation.

3.



a. x =>

b. Scheme expression:

c. mutation: (set-car! (cdr (second x)) 4)

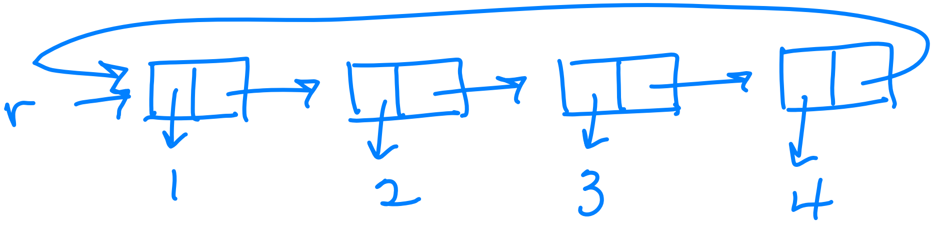
x =>

ring problems

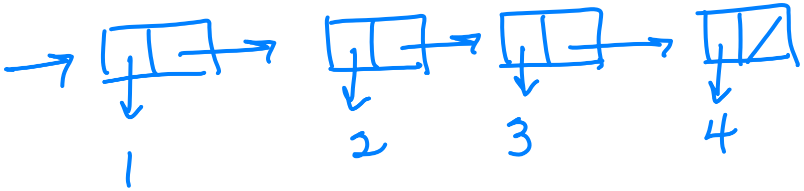
Rings are circular structures similar to lists.

If we define a ring r: (define r (make-ring ‘(1 2 3 4)))

the following are true: (nth 0 r) => 1 (nth 1 r) => 2 … (nth 4 r) => 1



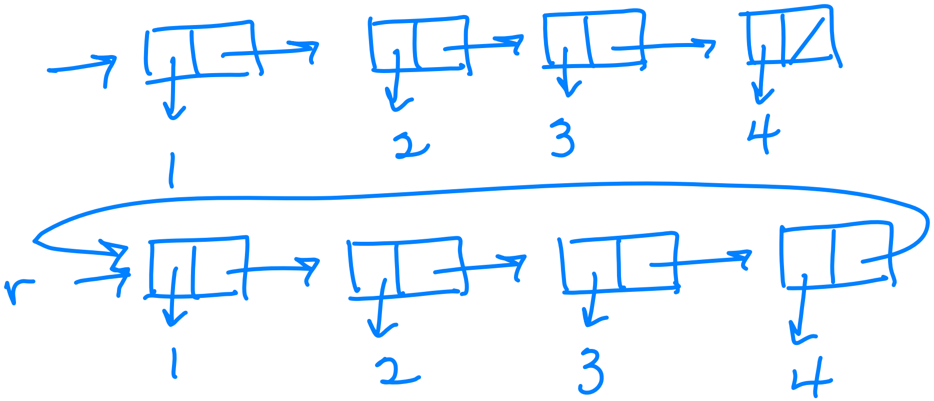
In order to make a ring, we need a procedure last-pair which returns the last pair in its argument:
(last-pair (list 1 2 3 4))=> (4)



1. Write last-pair.

```
(define (last-pair x)
```

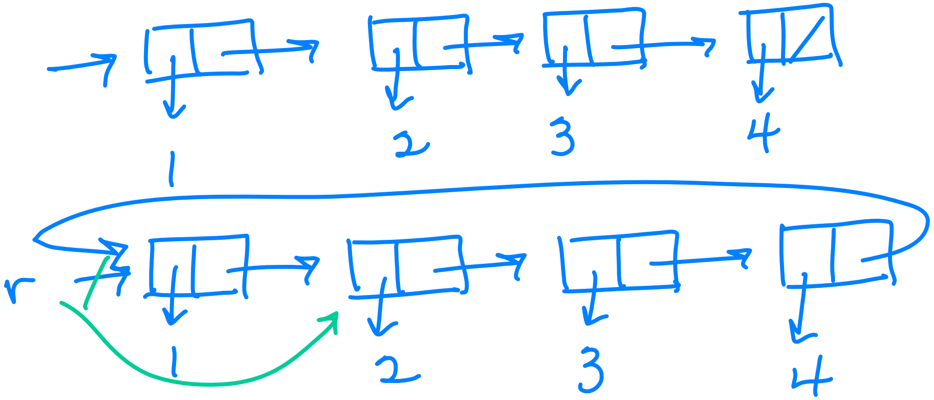
ring problems



2. Write make-ring!, which takes a list and makes a ring out of it..

```
(define (make-ring!  x)








)
```

ring problems

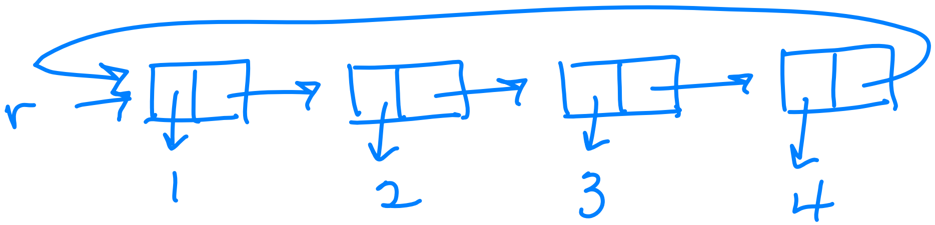


3. Write the procedure rotate-left, which takes a ring and returns a ring that has been rotated one to the left.

```
(define r1 (rotate-left r))
(nth 0 r1) => 2
```

```
(define (rotate-left ring)



)
```

ring problems

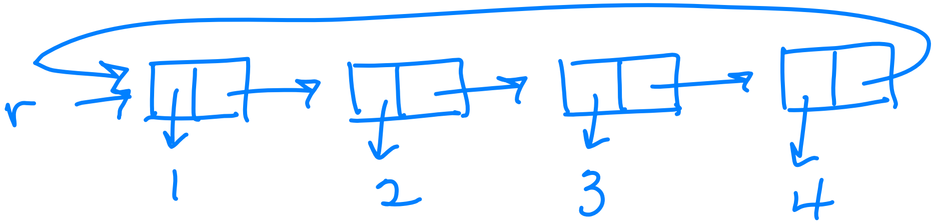


4. What happens if you evaluate (length r) on the above ring?

Write the procedure ring-length, which returns the length of the original list used in constructing the ring. (Hint: Write a helper procedure.)

```
(define (ring-length ring)

)
```

ring problems



5. Rotating a ring to the right is harder than rotating to the left. (Why?) Write the procedure rotate-right. (Hint: You might want to use the procedure repeated, which takes a procedure, a number n, and an argument to the procedure, and repeatedly calls the op on the argument n times.)

```
(define (rotate-right ring)















)
```

more set-car! and set-cdr! problems

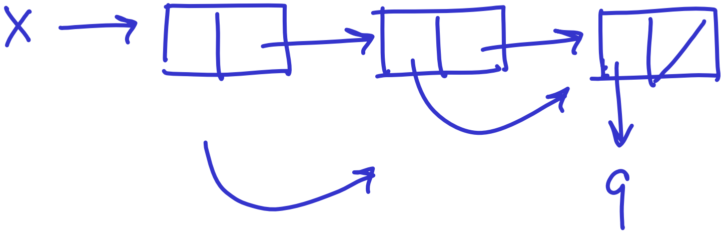
For the box & pointer diagram:

(a) Write what Scheme prints out for the structure (if it can)

(b) Write a Scheme expression that makes the structure(if an error, describe it)

(c) Draw the structure that results from the mutation, and its printed representation.

1.



a. x =>

b. Scheme expression:

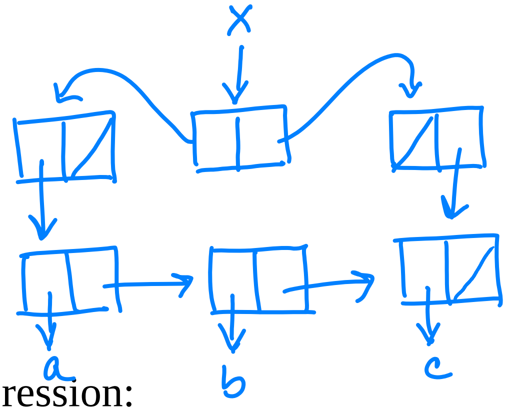
c. mutation: (set-cdr! (car x) '(8))

x =>

more set-car! and set-cdr! problems

For the box & pointer diagram:
(a) Write what Scheme prints out for the structure (if it can)
(b) Write a Scheme expression that makes the structure (if an error, describe it)
(c) Draw the structure that results from the mutation, and its printed representation.

2.



a. x =>

b. Scheme expression:

c. mutation: (set-cdr! (cddr x) (caaar x))

x =>

more set-car! and set-cdr! problems

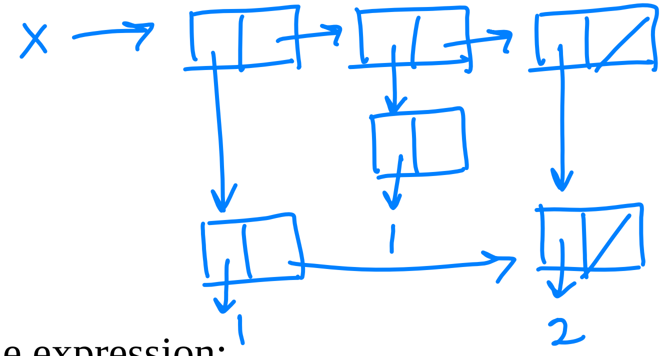
For the box & pointer diagram:

(a) Write what Scheme prints out for the structure (if it can)

(b) Write a Scheme expression that makes the structure (if an error, describe it)

(c) Draw the structure that results from the mutation, and its printed representation.

3.



a. x =>

b. Scheme expression:

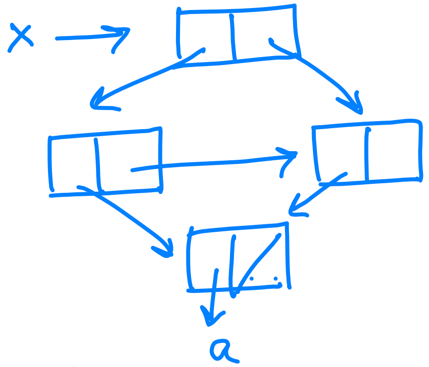
c. mutation: (set-car! (caar x) 3)

x =>

more set-car! and set-cdr! problems

For the box & pointer diagram:
(a) Write what Scheme prints out for the structure (if it can)
(b) Write a Scheme expression that makes the structure (if an error, describe it)
(c) Draw the structure that results from the mutation, and its printed representation.

4.



a. x =>

b. Scheme expression:

c. mutation: (set-cdr! (first x) (second x))

x =>

more set-car! and set-cdr! problems

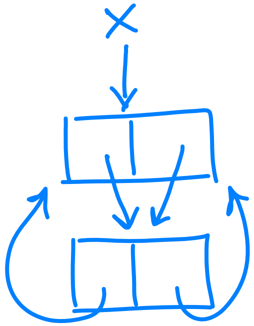
For the box & pointer diagram:

(a) Write what Scheme prints out for the structure (if it can)

(b) Write a Scheme expression that makes the structure (if an error, describe it)

(c) Draw the structure that results from the mutation, and its printed representation.

5.



a. x =>

b. Scheme expression:

c. mutation: (set-car! (cdr x) ’())
(set-cdr! (car x) ’())

x =>

c .)))