Practice ED!

- (define x 10)
- (define (foo y)

```
(let ((x 20)
      (f (lambda (z) (y))))
  (f 14)))
```

- (foo (lambda () x))

Mutable Data…

…mutations mwahahaha….

So far…

- So first off, for ADTs we know how to
  - create them (constructors)
  - get info from them (selectors)
- Now it's time to find out how to change them!

Intro to Mutations

- So basically everything in Scheme is represented in pairs.
- So remember cons creates a pointer to a pair, where the car is a pointer to the first element, and the cdr is a pointer to the last…

Pointers…

- So if you've programmed in other languages such as Java & C you know what these are.
- So we have 2 mutators…
  - set-car!
  - set-cdr!

Set-car! & Set-cdr!

- set-car!
  - Does what you think it does…it sets the car of a pair to be a value so….
    (set-car! x y) means to change the car of x to point to y
- set-cdr!
  - It sets the cdr of a pair to be a value
    (set-cdr! x y) means to change the cdr of x to point to y
- *note* usually **'!'** means change in Scheme

Mutators in action!

- (define x (cons 1 2))
  → (1 . 2)
- (define y (list 1 2 3))
  → (1 2 3)
- (set-car! x y)
- x
  → ((1 2 3) . 2)
- (set-cdr! (cddr y) (cdr x))
- y
  → (1 2 3 . 2)

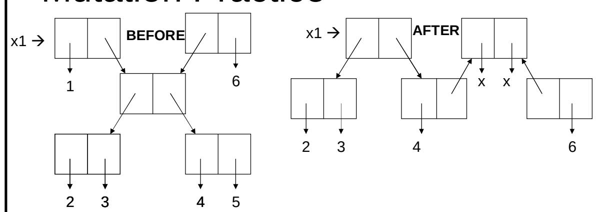
Let's do some…

```
> (define x (list (list 'to 'be)))
> (define y (list 'or 'not))
> (set-cdr! x y)
> x
> y
> (set-cdr! (cdr y) (car x))
> x
> y
```

Answers…

```
> (define x (list (list 'to 'be)))
    x
> (define y (list 'or 'not))
    y
> (set-cdr! x y)
    okay
> x
    ((to be) or not)
> y
    (or not)
> (set-cdr! (cdr y) (car x))
    okay
> x
    ((to be) or not to be)
> y
    (or not to be)
```

Mutation Practice



X-Men…
(set-car! x1 ______)
(set-car! ______ ‘x)
(set-cdr! ______ ______)
(set-cdr! x1 ______)
(set-cdr! ______ ‘x)

Mutation Answer

- X-Men…
  (set-car! x1 (cadr x1))
  (set-car! (cdr x1) ‘x)
  (set-cdr! (cddr x1) (cdr x1))
  (set-cdr! x1 (cddr x1))
  (set-cdr! (cddr x1) ‘x)

Eq? vs. Equal?

- What’s the difference?
  - equal? tests for whether or not two symbols are equal.
  - eq? tests for **pointer** equality.

Eq? vs. Equal?

- Let's take an example…
  - (define x (cons 1 2))
  - (define y (cons 1 2))
  - (eq? x y) → #f
  - (equal? x y) → #t
  - (set-car! x y)
  - (eq? (car x) y) → #t
- Still confused?
  - The EQ? story…
- Make sure you use these two predicates correctly!

Another helpful predicate…

- memq
  - Works like member, but this is for pointer equality.

```
STk> (define x (list 1 2))
okay
STk> (define y (list x x))
okay
STk> (memq 1 x)
(1 2)
STk> (memq y x)
#f
STk> (memq x y)
((1 2) (1 2))
STk> (define z (list 1 2))
okay
STk> (memq x z)
#f
```

Equivalent?

- As you can see we changed the structure of x and y using our mutators.
- Now when we define an ADT we can define a constructor, selectors, and mutators. Many people wonder why the following are not equivalent:
  (set-cdr! x y) equivalent to ? (set! (cdr x) y)

Equivalent?

(set-cdr! x y) equivalent to ? (set! (cdr x) y)

- **NO**, these examples are not equivalent.
- Set! changes values.
- set-car/cdr! changes pointers! Very very different.
- Now lets look at some examples of data structures that use mutation frequently.

Stacks, Trees, & Queues

- **Stacks**
  A last-in first-out queue in which we keep track of pointers to the top element and the next to top element.
- **Trees**
  We already know about trees, but look forward to 61b where you will learn about balanced-trees, tree-rotations, removing and adding elements to all kinds of trees.
- **Queues and Deques**
  A first-in first-out structure that needs to keep track of the first and next element. (A deque is a double-ended queue). (in book if you're interested!)

More Problems!

- Write **remove-dupls!** which takes a list and removes all the duplicate elements of a non-empty list. You may not construct new pairs, ie use **cons** or anything like that.
  - (define x (list 'a 'b 'b 'a))
  - (remove-dupls! x) → [returns something]
  - x → (b a)

Answer: **remove-dupls**

- (define (remove-dupls lst)

```
(define (remove-dupls lst)
    (cond ((null? (cdr lst)) lst)
          ((member (car lst) (cdr lst))
           (set-car! lst (cadr lst))
           (set-cdr! lst (cddr lst))
           (remove-dupls! lst))
          (else (remove-dupls! (cdr lst)))))
```

More Mutations!

- Write **merge!** which takes in two lists and behaves in this manner…
  - (define x (list 1 3 5 7))
  - (define y (list 2 4 6))
  - (merge! x y) → (1 2 3 4 5 6 7)
  - x → (1 2 3 4 5 6 7)
  - y → (2 3 4 5 6 7)

DO NOT ALLOCATE NEW PAIRS!!!

Answer: **merge!**

```
(define (merge! x y)
  (cond ((null? x) y)
        ((null? y) x)
        ((< (car x) (car y))
         (set-cdr! x (merge! (cdr x) y))
         x)
        (else
         (set-cdr! y (merge! x (cdr y)))
         y)))
```

Next Time: Streams & Midterm Review…

...row, row, row your boat…