



# CS1 Recitation

Week 9

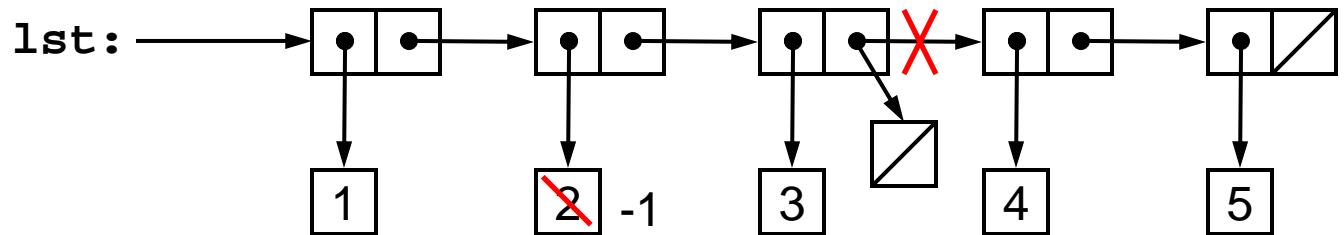
# set-car! and set-cdr!

- Allow you to change the contents of existing cons cells
- Take two arguments:
  - An expression that evaluates to a cons cell
  - A value to store into the cons cell
- set-car! and set-cdr! do not return a value

# set-car! and set-cdr! examples

## ■ Examples:

```
(define lst '(1 2 3 4 5))
```



```
(set-car! (cdr lst) -1)
```

```
lst
```

```
→ (1 -1 3 4 5)
```

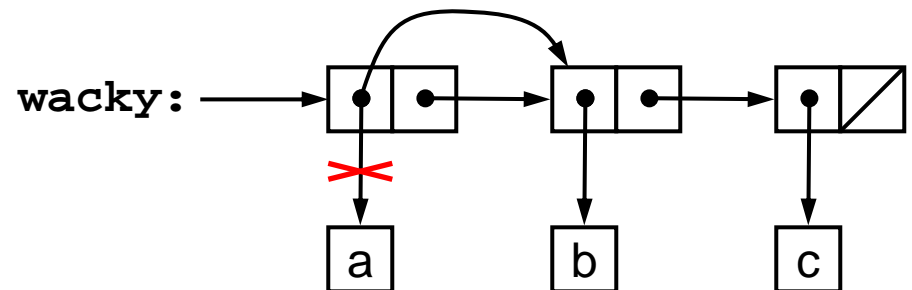
```
(set-cdr! (cddr lst) '())
```

```
lst
```

```
→ (1 -1 3)
```

# Wacky Lists

- Can create bizarre lists with `set-car!` and `set-cdr!`
  - `(define wacky '(a b c))`
  - `wacky`
  - $\rightarrow$  `(a b c)`
  - `(set-car! wacky (cdr wacky))`
  - `wacky`
  - $\rightarrow$  `((b c) b c)`



# Crazy Lists

- Things can get really crazy:

- (define crazy '(a b c))

- crazy

- → (a b c)

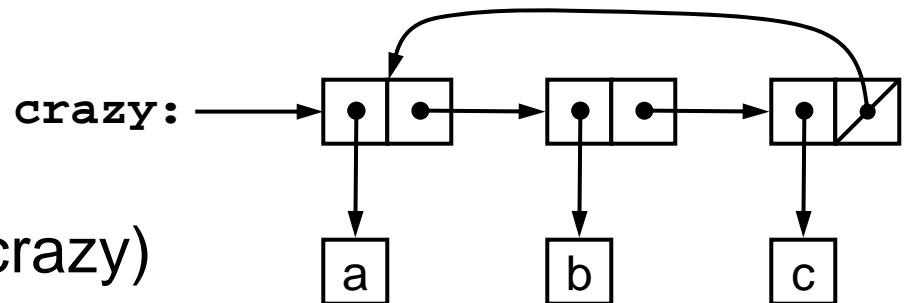
- (set-cdr! (cddr crazy) crazy)

- crazy

- → #0=(a b c . #0#)

- DrScheme detects the cycle and prints an appropriate version

- “#0 is this value. Where #0# appears, it refers back to itself.”



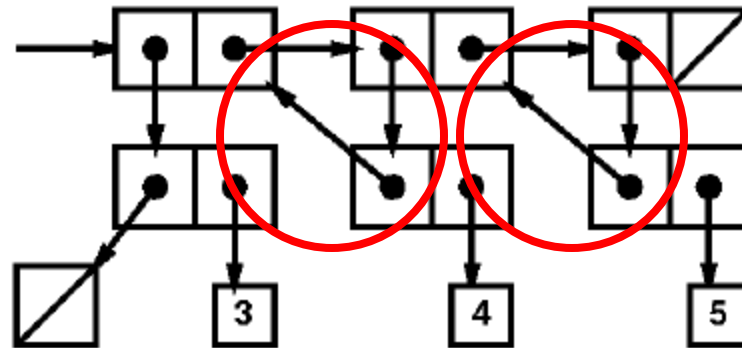
# set-car! and set-cdr!

- set-car! and set-cdr! are not special forms
  - These are normal Scheme functions
    - First argument can be any expression that evaluates to a cons pair
  - set! *is* a special form...
    - First argument *must* be a name (a symbol)
- Scheme does not provide set-cadr!, set-caddr!, ...
  - Only set-car! and set-cdr!
- Very common mistake seen on final exams...

# From Part A:

- Given a diagram, write Scheme code to create the structure

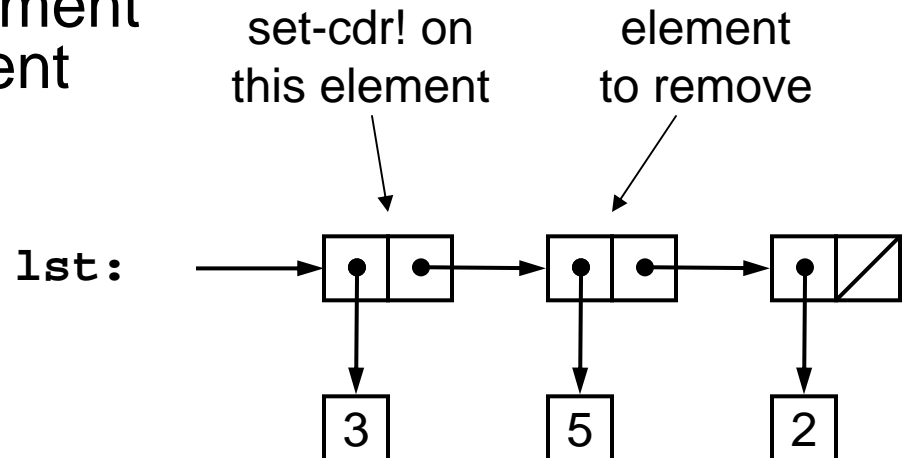
- Example:



- Remember: pointers to a cons cell point to the entire cons cell
  - ...not just the car or the cdr of the cell.

# List Mutation

- List mutation in Scheme can be a bit tricky
  - Each element only points to next element
- To remove an element, must have previous element
- Example: (define lst '(3 5 2))
  - Want to remove element with value 5
  - Must set-cdr! on first element to remove second element



# Removing Elements from a List

- Clearly, removing first element is a special case
  - Don't have a "previous element" to deal with
- Functions that traverse a list and destructively remove elements usually follow this pattern:
  - Outer function handles case of when first node matches
    - No previous node to work with
  - An internal helper function handles nodes with a "previous node"
    - Constraint: previous node definitely doesn't match!

# Remove All Matching Elements

- Write a function that destructively removes all elements that match a particular predicate
  - In class, only removed the first element that matches
  - Signature: (remove-if! a-list remove?)
    - a-list is the list to process
    - remove? is a predicate specifying what elements to remove
- Form of solution:
  - Outer function takes care of first element
    - First element doesn't have a "previous element"
  - An internal helper function will take care of elements with a "previous element"

# Removing the First Element(s)

- Start working on the first-node case:

```
(define (remove-if! a-list remove?)  
  (define (remove-aux previous) ...) ;; fill in this part later  
  (cond ((null? a-list) a-list)      ;; base case  
        ((remove? (car a-list))      ;; need to remove first element  
         (remove-if! (cdr a-list) remove?))  
        (else (remove-aux a-list)    ;; want to keep first element  

```

- Important detail:

- May need to remove multiple elements from start of list!
- (define lst '(1 1 2 3 2))
- (set! lst (remove-if! lst (lambda (x) (= x 1))))

# Removing Other Elements

- Define `remove-aux` now

  - `(define (remove-if! a-list remove?)`

  - `(define (remove-aux previous)`

  - `...`

- What do we know about `previous`?

  - `previous` is not null

  - `(remove? previous)` is false

# Removing Other Elements (2)

- Define remove-aux now:

```
(define (remove-if! a-list remove?)  
  (define (remove-aux previous)  
    (let ((current (cdr previous))) ;; actually working with current  
      (cond ((null? current)  
             'done) ;; dummy value to show we're done  
            ((remove? (car current)) ;; remove current node  
             (set-cdr! previous (cdr current))  
             (remove-aux previous))  
            (else (remove-aux current)))))) ;; keep current node  
  ... )
```

- Again, may need to remove multiple elements in a row

# Using remove-if!

- Try it out:

- (define a '(1 1 2 2 3))

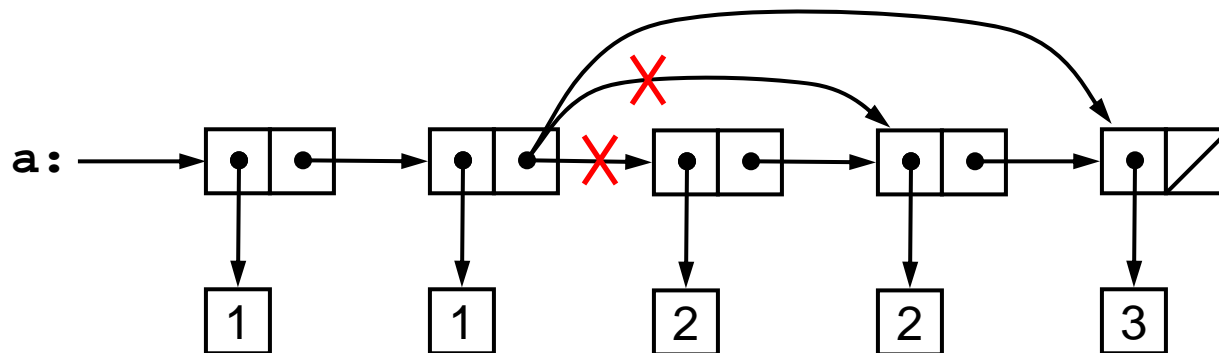
- (remove-if! a (lambda (x) (= x 2)))

- → (1 1 3)

- a

- → (1 1 3)

- remove-if! uses set-cdr! to change a in-place



# Using remove-if! (2)

- Try removing elements at the front:
  - `(define b '(1 1 2 2 3))`
  - `(remove-if! b (lambda (x) (= x 1)))`
  - `→ (2 2 3)`
  - `b`
  - `→ (1 1 2 2 3)`
- Can't use `set-cdr!` to remove elements at the front...
- And, `remove-if!` can't really use `set!`
  - `remove-if!` has its own frame containing a-list as a binding
  - Can't write this as a generic function, without using it like this:
    - `(define b (remove-if! b (lambda (x) (= x 1))))`
    - `(set! b (remove-if! b (lambda (x) (= x 1))))`

# Using remove-if! Properly

- Since remove-if! can't use set-cdr! to remove first elements, need to use remove-if! like this:
  - (set! my-list  
    (remove-if! my-list my-pred?))
  - (define my-list  
    (remove-if! my-list my-pred?))
  - This is why remove-if! also returns the final result
- If you *know* the first element will never be removed, you could use it like this:
  - (remove-if! my-list my-pred?)

# eq? and equal?

- eq? returns #t if its arguments are the same identical object
- equal? returns #t if its arguments have the same values
  - the same structure and contents
- Avoid using eq? for numbers
  - Use = for numbers if possible
  - eq? intended for symbols, cons cells, etc.

# eq? and equal? examples

- `(define a '(1 2 (3 4) 5))`
- `(define b '(1 2 (3 4) 5))`
- `(define c a)`
- `(eq? a b)`
  - `→ #f` ;; not the same object
- `(eq? a c)`
  - `→ #t` ;; the same identical object
- `(equal? a b)`
  - `→ #t` ;; same structure and contents
- `(equal? a c)`
  - `→ #t` ;; same structure and contents (obvious)

# Part B: memq function

- (memq value a-list)
  - Searches a-list for value
  - If value appears in list, returns the list from where the value appears
    - i.e. returns the first cons cell in the list whose car is the specified value
  - If value doesn't appear in list, returns #f
- Uses eq? for comparing value to list contents
  - ...hence the name memq
  - Another function member uses equal? for testing

# memq function

- Use memq to find out if a value appears in a list
- e.g. find a name in a list of names
  - (define (name-in-list? name name-list)  
 (pair? (memq name name-list)))
  - If name is found, memq returns the list from that point forward
    - i.e. a cons cell whose car is the specified name
  - Otherwise, memq returns #f, which isn't a cons cell
  - Example:
    - (name-in-list? 'bob '(al bob carl dave)) → #t
    - (memq 'bob '(al bob carl dave)) → (bob carl dave)

# Design Problems

- Last “important” topics in CS1 are:
  - mutable state, set!, set-car!, set-cdr!, etc.
  - message-passing objects, OOP
- OOP allows us to build useful abstractions
- Final exam usually includes a reasonably large design problem
  - Final walks you through the objects to create!
  - You should be used to programs that use several different message-passing objects in the code



# A Simple Design Example

- Want to create a message-passing object to represent a simple bank
  - Bank manages checking accounts
- Operations:
  - Open a checking account, with an initial deposit
  - Check the balance of an account
  - Make deposits and withdrawals on an account
  - Transfer money between accounts
  - Close an account, and receive back the final amount

# Banks and Bank Accounts

- Bank accounts consist of:
  - An account ID
  - A balance
- The bank keeps track of all open accounts
- When a person opens a new account, its ID should be unique! 😊
  - The bank needs to be able to generate new, unique account IDs
- How should we design and implement this?

# Bank Design

- How should we design this?
- A bank message-passing object
  - Store the list of accounts currently open
  - Keep track of the next account ID to assign
- A bank account message-passing object
  - Provide a container for each account's ID and its balance
  - Provide some simple operations on accounts
- Bank object can delegate many operations to the account object

# Bank Operations

- Bank object needs to provide some operations
  - 'open-account initial-amt → returns new account ID
  - 'get-balance acct-id → returns balance
  - 'deposit acct-id amount → returns new balance
  - 'withdraw acct-id amount → returns new balance
    - ...or an error if insufficient funds
  - 'transfer amount from-id to-id
  - 'close-account acct-id → returns final balance
- This is the object's public interface
  - What the object's clients are able to interact with

# More Bank Operations

- Could also use some other operations when implementing the bank
  - When creating a new account, need to generate a new account ID
  - Most operations need to retrieve a particular account object, given its account ID
- Do we want to expose these operations?
  - Code outside the bank object *should not* be able to perform these operations
  - They are private members of the object

# Bank Account Object

- Start with bank account object, to make things easier

```
(define (make-account id balance)
  (lambda (op . args)
    (cond ((eq? op 'get-id) id)
          ((eq? op 'get-balance) balance)
          ((eq? op 'update-balance!)
           (let ((delta (car args)))
             (if (< (+ balance delta) 0)
                 (error "Insufficient funds in account" id)
                 (begin
                    (set! balance (+ balance delta))
                    balance))))))
          (else (error "Unrecognized operation" op))))))
```

# Bank Object

- Bank object needs some state:

- A list of open accounts
- Last account ID issued

- Code:

```
(define (make-bank)
  (define accounts '())
  (define last-account-id 10000)
  ...
)
```

- Could also use a let expression – just creates an additional frame

- (I like define, but do what makes sense to you...)

# Helper Functions


- Declare the bank object's internal functions next
- Code:

```
(define (make-bank)
  (define accounts '())
  (define last-account-id 10000)

  (define (new-account-id)
    (set! last-account-id (+ last-account-id 1))
    last-account-id)

  (define (find-account acct-id) ...)
  (define (remove-account! acct-id) ...)
  ...
)
```

An exercise for  
the student 😊





# Bank Message-Passing Function

- Finally, implement message-passing function using helpers and internal state
- Only the message-passing function is returned
  - Caller can't access helper functions or internal state
  - Bank's message-passing function can still use these internally

# Bank Message-Passing Function

```
(define (make-bank)
```

```
...
```

```
(lambda (op . args)
```

```
  (cond ((eq? op 'open-account)
```

```
    (let ((new-acct (make-account  
                    (new-account-id) (car args))))
```

```
      (set! accounts (cons new-acct accounts))
```

```
      (new-acct 'get-id)))
```

```
  ((eq? op 'get-balance)
```

```
    ;; Note: find-account should report an error on invalid acct ID.
```

```
    (let ((acct (find-account (car args))))
```

```
      (acct 'get-balance)))
```

```
...
```

```
  (else (error "Unrecognized operation" op))))))
```

# More Complicated Operations

- Balance transfer is more interesting

- Three arguments: amount, from-id, to-id
- Use let to simplify argument passing, etc.

...

```
((eq? op transfer)
```

```
(let ((amount (car args)
```

```
      (from-acct (find-account (cadr args)))
```

```
      (to-acct (find-account (caddr args))))
```

```
(from-acct update-balance! (- amount))
```

```
(to-acct update-balance! amount)))
```

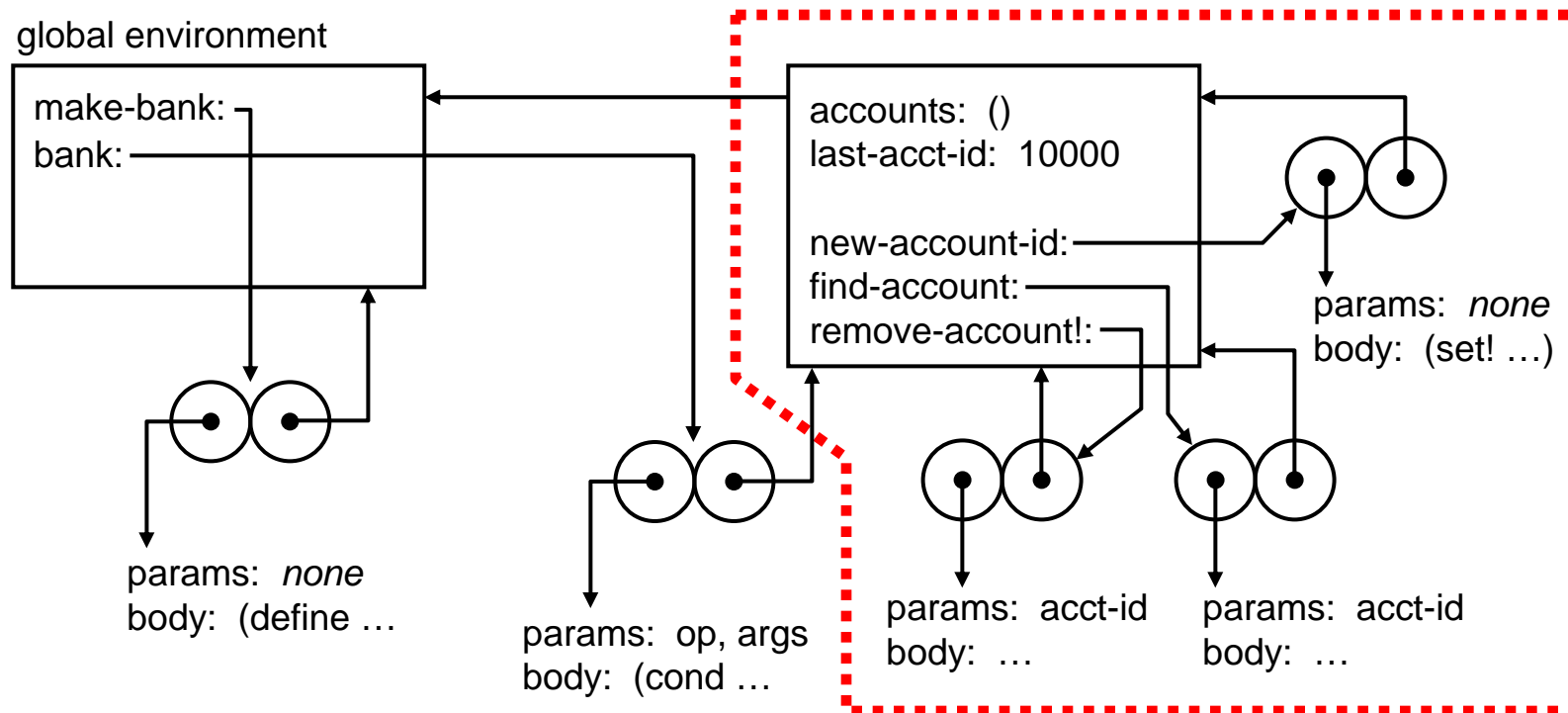
- Again, find-account should report an error if an account ID is not found

- Deduct from “from account” first, to prevent overdraft!

- Account object will flag an “insufficient funds” error on overdraft

# Environment Model Diagram

- (define bank (make-bank))
  - All internal state and helpers are hidden from outside the bank message-passing object



# Using Abstractions

- In our bank example, abstractions are *very* important
- Account object provides simple operations, along with error checking
  - Bank can use operations provided by account
  - Bank doesn't have to worry about error-checking for overdrafts
- Bank message-passing object also uses its own internal helper functions
  - Encapsulates complicated operations, such as getting a new account ID, finding or removing an account
- Implementation of the desired features becomes very simple!