CSE1729: Introduction to Programming

Mutable data: Objects, Streams, and Environment Semantics

Greg Johnson

Our story thus far...

- We have focused, so far, on a programming paradigm called: ***Functional Programming***.

- When you build a program in a functional style, computation proceeds as the result of function application.

- In particular: *binding a variable only happens when a function is called*.

- This means (the golden rule of a functional language):

> Once a variable is bound, its value (in a particular environment) never changes

The virtues of functional programming

- Functional programming is predictable and relatively easy to analyze. Why?

- Variable values "never change."

- Function definitions never change: Once a function has been defined (in a particular environment), applying it multiple times to a particular value will *always result in the same value*--it behaves like a mathematical function.

Mutable data is powerful

- However, the notion of changing the value of a variable can have some powerful implications.

- In particular (and perhaps most importantly), it can naturally model some important and intuitive computational frameworks.

- In SCHEME, we introduce a new *destructive* ingredient that allows us to change the value of a variable in an environment.

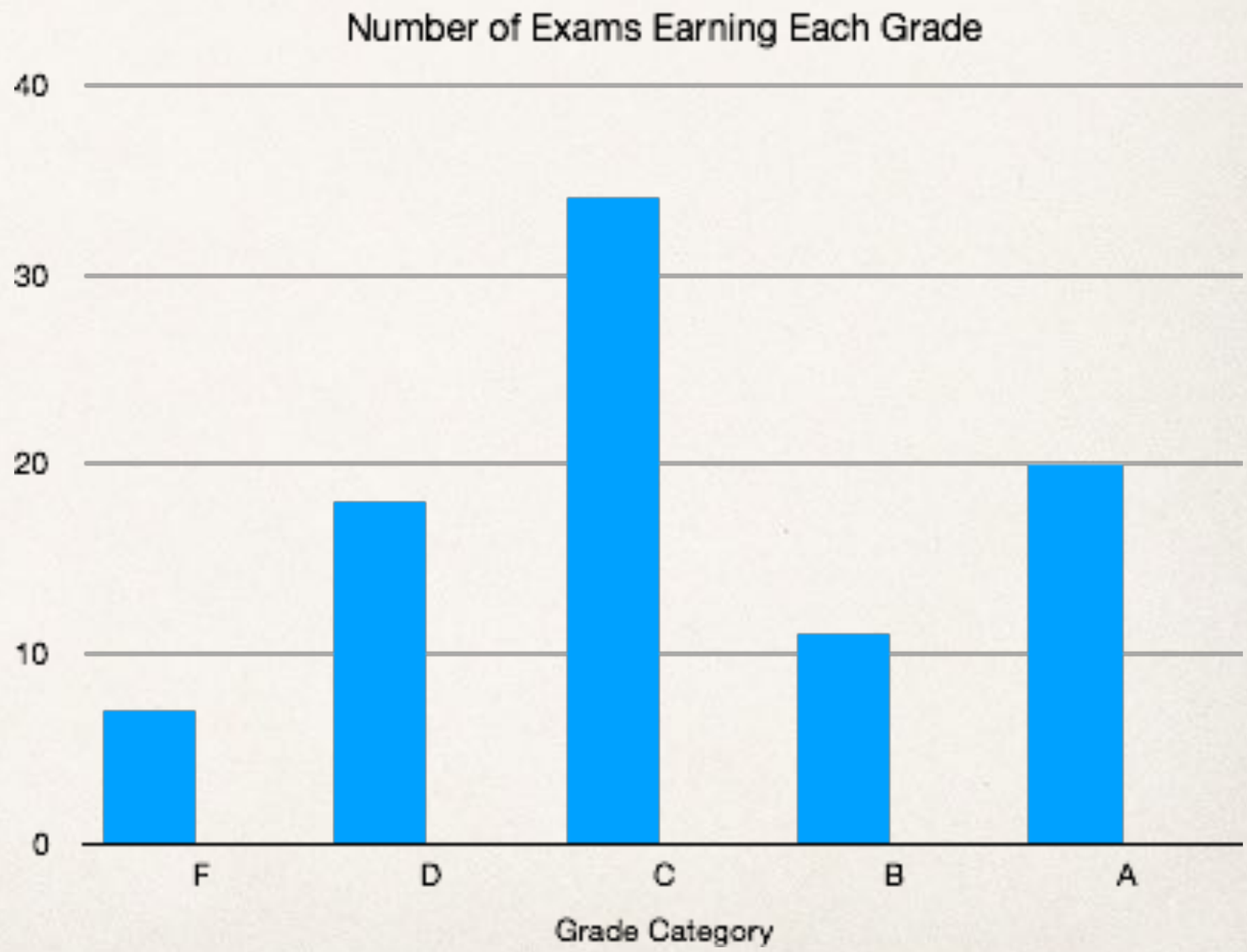
With great power...
comes great responsibility

- Once we forsake purely functional programming, program dynamics become more...

Complex

Prelim 2 Statistics

- Mean: 20.75
- Standard Deviation: ~6.7
- A: >= 27
- B: >= 24.09
- C: >= 17.41
- D: >=10.72
- F: < 10.72



Motivation

- Recall the queue datatype. Problem: **enqueue**/**dequeue** in constant time.

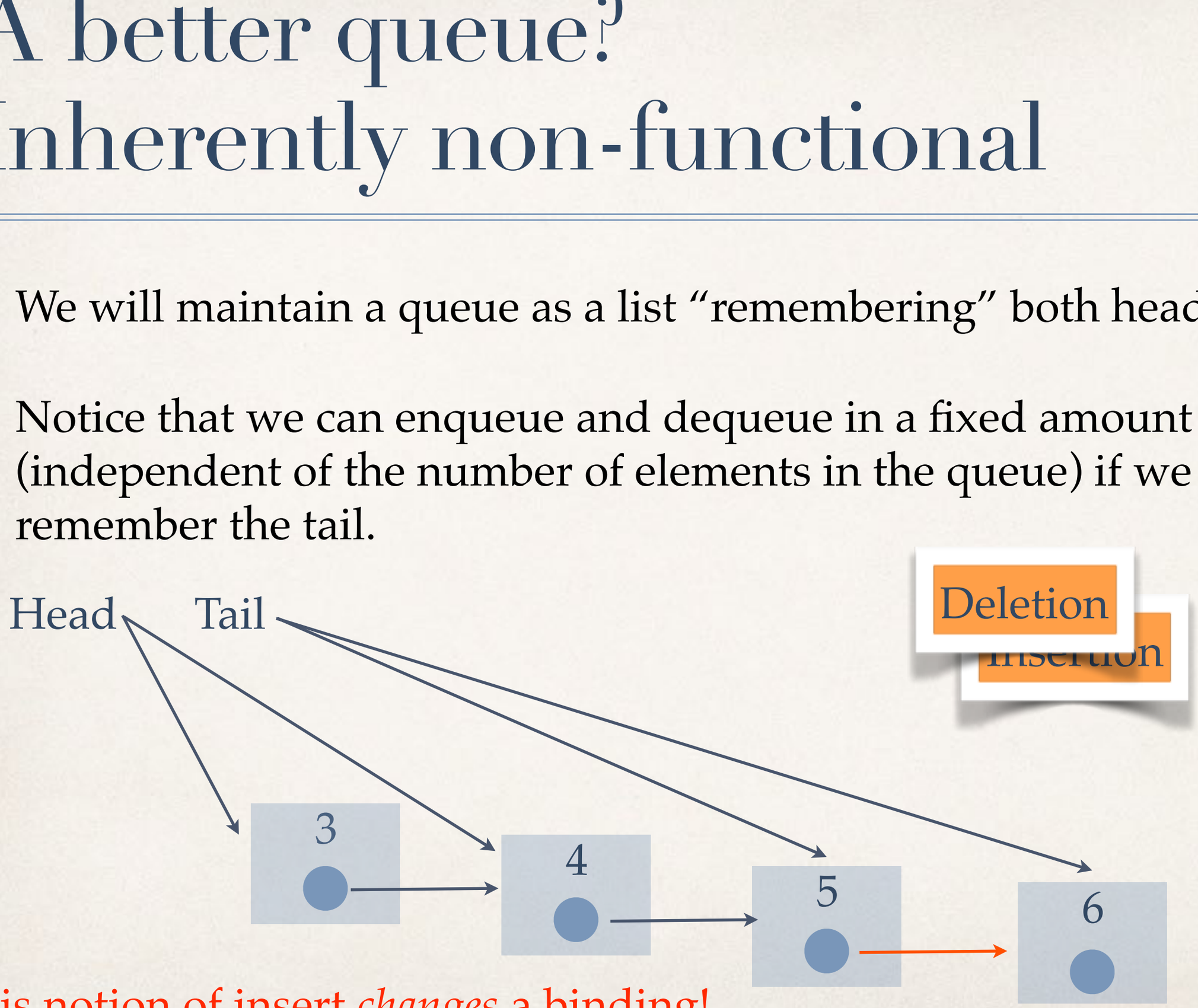
- Idea: If we "knew" where the end of the queue was, we could avoid traversing the entire structure during an enqueue.

- Idea: represent a queue as a pair which maintains the front and back of the queue simultaneously.

- Then...

A better queue?
Inherently non-functional

- We will maintain a queue as a list "remembering" both head and tail.
- Notice that we can enqueue and dequeue in a fixed amount of time (independent of the number of elements in the queue) if we remember the tail.



This notion of insert *changes* a binding!

Basic mechanics of mutable data

- Before discussing mutation of structured data...
  - We'll return briefly to simple data types
  - We'll introduce the notion of object, a principled way to control the complexity of mutable data.
  - Then....finally...we'll return to structured data and give an efficient implementation of a heap.

The `set!` function

- The `set!` changes the value of a variable in the assignment in which it is called.
- A simple example:

```
> (define a 3)
> a
3
> (set! a 4)
> a
4
> (set! a 5)
> a
5
```

a:3

a:4

a:5

The `sequence/begin` function

- The sequence function

  `(begin <expr1> <expr2> ... <exprk>)`

  carries out a sequence of function calls, returning the value of the last call.

- In purely functional programming, *there would be no point to such an operation* (since the return value would always simply be the value of `<exprk>`), but now that we have *side-effects*, sequences can be a valuable tool.

Warning: Your textbook calls this command: `sequence`.

What's the big deal?

- Our programming enterprise will be most successful if we build computational models that reflect the "intuitive, real-world" structure of the objects we're modeling.

- When we want to model objects that *change over time*, we have a choice to make:
  - Model them functionally, as a sequence of "different" objects.
  - Model them via mutation, as single object that is changing.
- Consider sorting: philosophically, are we creating a new list, that is sorted, or massaging the old list into sorted order? It has far reaching consequences for efficiency, intelligibility, modularity, ...

A simple example

- A simple example: maintaining balance information in a bank account:

```
> (define balance 100)
> (define (withdraw f)
    (if (> f balance)
        "Insufficient funds!"
        (begin (set! balance
                     (- balance f))
               balance)))

> (withdraw 40)
60
> (withdraw 40)
20
> (withdraw 40)
"Insufficient funds!"
```

A SCHEME string

A problem with this implementation

```
> (define balance 100)
> (define (withdraw f)
    (if (> f balance)
        "Insufficient funds!"
        (begin (set! balance (- balance f))
               balance)))
```

- Anyone can now change the bank balance...

```
(set! balance 1000000000)
```

> We'd like an abstraction barrier, so that the only way to change the balance is via the official `withdraw` function

Abstraction barriers

Expose only what must be exposed:

- Protect the internals from accidental (or malicious) tampering.
- Protect the user.

Hiding the balance: recall *environment definition semantics*

- Recall lexical scope rules: *the valu*
  determined by the environment in w

The value of x in the body of `add-5` is determined by the environment in which `add-5` was defined.

- To warm up:

```
> (define (incrementer x)
    (lambda (z) (+ x z)))
> (define add-5 (incrementer 5))
> (add-5 10)
15
> (define x 50)
> (add-5 10)
15
>
```

x:5

`(lambda (z) (+ x z))`

`add-5:`

x:50

`incrementer`

Hiding the balance

- The basic definition

```
(define (new-account initial-balance)
  (let ((balance initial-balance))
    (lambda (withdrawal)
      (if (> withdrawal balance)
          "Insufficient funds"
          (begin
            (set! balance (- balance withdrawal))
          balance)))))
```

This function is returned; it has access to balance

- Usage

```
> (define my-acct-w (new-account 100))
> (my-acct-w 30)
70
> (my-acct-w 30)
40
```

Note nonfunctional behavior!

Who sees what?

```
(define (new-account initial-balance)
  (let ((balance initial-balance))
    (lambda (withdrawal)
      (if (> withdrawal balance)
          "Insufficient funds"
          (begin
            (set! balance (- balance withdrawal))
            balance)))))
```

balance

This function is passed out of the environment; note that it refers to `balance`.

```
(define my-acct-w (new-account 100))
```

`balance` does not exist in this environment, but `my-acct-w` can change it!

This is important...let's do it again

- Consider the code fragment:

```
(define (make-hidden-value initial-value)
(let ((value initial-value))
  (define (mult-by x) (begin (set! value (* value x))
                             value))
  (define (add-to y)  (begin (set! value (+ value y))
                             value))
  (cons mult-by add-to)))
```

- What does it do? It creates an environment with a variable `value`. It passes back two functions (as a pair) which can operate on `value`.

In action...

```
(define (make-hidden-value initial-value)
  (let ((value initial-value))
    (define (mult-by x) (begin (set! value (* value x))
                               value))
    (define (add-to y)  (begin (set! value (+ value y))
                               value))
    (cons mult-by add-to)))
```

value: 1244024

This environment is persistent!

Let's follow it during an execution:

```
> (define function-pair (make-hidden-value 100))
> ((car function-pair) 2)
200
> ((car function-pair) 2)
400
> ((cdr function-pair) 2)
402
> ((cdr function-pair) 2)
404
```

function-pair: $(m \; . \; a)$

```
(define (mult-by x)
  (begin (set! value (* value x))
         value))
```

```
(define (add-to y)
  (begin (set! value (+ value y))
         value))
```

Objects

- In our bank account example, the only way to change the balance is through the `withdrawal` function. Thus, while there is destructive assignment, it is carried out behind an abstraction barrier.

- This fits into a paradigm called *object-oriented* programming:

  - Data objects are permitted private "state" information that can change over time, but

  - All interaction with these objects are carried out by specifically-crafted functions; no other functions have access to the state.

A short aside: tokens in SCHEME

- In SCHEME, you can work with "token" values

```
> 'green
green
> (eq? 'green 'Bach)
#f
> (eq? 'green 'green)
#t
```

- No operations are defined on tokens, except for equality.

A more sophisticated bank account example

- The idea: As before, we will establish an environment in which `balance` lives. Functions will be defined in this environment with access to the balance, and passed out of the environment.

```
(define (new-account initial-balance)
  (let ((balance initial-balance))
    (lambda (method x)
(cond ((eq? method 'deposit)
       (begin
         (set! balance
               (+ balance x))
         balance))
      ((eq? method 'withdrawal)
       (if (> x balance)
           "Insufficient funds"
           (begin
             (set! balance
                   (- balance x))
             balance)))))))
```

Putting this to work

- Once the new-account object creator is defined...

```
> (define my-acct (new-account 200))
> (my-acct 'deposit 250)
450
> (my-acct 'withdrawal 300)
150
> balance
. . reference to undefined identifier: balance
>
```

- The function `my-acct` is a *dispatcher;* it can provide different functionality depending on the first parameter.

- *Note that balance is undefined in this environment*.

Refining the solution: a first order dispatcher

- We can slightly improve the solution by allowing the dispatcher to return the appropriate function (rather than applying it itself). Methods can now have varying number of arguments.

```
(define (new-account initial-balance)
  (let ((balance initial-balance))
    (define (deposit f)
      (begin
        (set! balance
              (+ balance f))
        balance))
    (define (withdraw f)
      (if (> f balance)
          "Insufficient funds"
          (begin
            (set! balance
                  (- balance f))
            balance)))
    (define (bal-inq) balance)
    (lambda (method)
      (cond ((eq? method 'deposit) deposit)
            ((eq? method 'withdraw) withdraw)
            ((eq? method 'balance-inquire) bal-inq)))))
```

The methods

The dispatcher

Using the first order dispatcher

- Note that the dispatcher now returns the requested *function*.

```
> (define my-account (new-account 500))
> (my-account 'deposit)
#<procedure:deposit>
> ((my-account 'deposit) 200)
700
> ((my-account 'withdraw) 250)
450
> ((my-account 'balance-inquire))
450
```

Why the double parentheses?

Objects, in general

- Implemented in SCHEME by defining an object creator.
  - The creator builds an environment containing the private state variables of the object.
  - It returns a function (which references these local variables in its body). The function can be a dispatcher, as in the last example.
- In general, these functions with access to the private state variables are called *methods* for this object.

Another example: The Set ADT implemented as an object

```
(define (set-object)
  (let ((S-list '()))
    (define (is-member? x)
      (define (member-iter remnant)
        (cond ((null? remnant) #f)
              ((eq? x (car remnant)) #t)
              (else (member-iter (cdr remnant)))))
      (member-iter S-list))
    (define (insert x)
      (set! S-list (cons x S-list)))
    (define (empty) (eq? S-list '()))
    (lambda (method)
      (cond ((eq? method 'empty) empty)
            ((eq? method 'member) is-member?
            ((eq? method 'insert) insert)))
```

A private variable

The methods; as defined they have access to value

The dispatcher; returned to caller

This Set object, in action

```
> (define S (set-object))
> (s 'empty)
#<procedure:empty>
> ((S 'empty))
#t
> ((S 'insert) 0)
> ((S 'member) 0)
#t
> ((S 'member) 1)
#f
> ((S 'insert) 1)
> ((S 'member) 1)
#t
```

This call creates an environment in which `S-value` lives

`S-value` does not exist in *this* environment

However, these methods can change it!

Observe how the object structure protects the implementation!

- Note that the object structure protects the implementation of the ADT.
- In fact, you could substitute an alternate implementation of this Set ADT (by redefining the private state variable that represents the set and redefining the methods). No one would know!
- In fact, assuming it is correct, it would be absolutely indistinguishable.
- How lovely! The object infrastructure allows us to mimic the satisfying abstraction layer of an ADT.

A complex number object

```
(define (make-complex x y)
  (define (square z) (* z z))
  (define (length)
    (sqrt (+ (square x)
             (square y))))
  (define (real-part) x)
  (define (imag-part) y)
  (define (conjugate) (make-complex x (- y)))
  (define (add c) (make-complex (+ x ((c 'real-part)))
                                (+ y ((c 'imag-part)))))
  (lambda (method)
    (cond ((eq? method 'real-part) real-part)
          ((eq? method 'imag-part) imag-part)
          ((eq? method 'length) length)
          ((eq? method 'conjugate) conjug
          ((eq? method 'add) add))))
```

x, y are private

Methods with access to x and y

Methods passed back by a dispatcher

Examples

```
> (define a (make-complex 4 5))
> ((a 'imag-part))
5
```

Why all the parens?

```
> (define b ((a 'conjugate))

> ((b 'imag-part))
-5
> ((((b 'add) (make-complex 5 6)) 'length))
9.055385138137417
>
```

`Conjugate` returns a new object

`add` takes an object as a argument

Interesting problem: Modeling sum by objects

- Note that some methods return objects!
- How do you model addition of complex numbers?
  - In the previous example, we sent one complex to the method of another, and used accessor methods.
  - Alternatively, you could write a function that takes two complexes (as objects) and adds them. Note that this typically requires only "read access" to the internal variables.
- In this case, the object structure does not protect destructive assignment, but it does hide implementation.

Recall the hailstone sequence: secret Russian cold war weapon

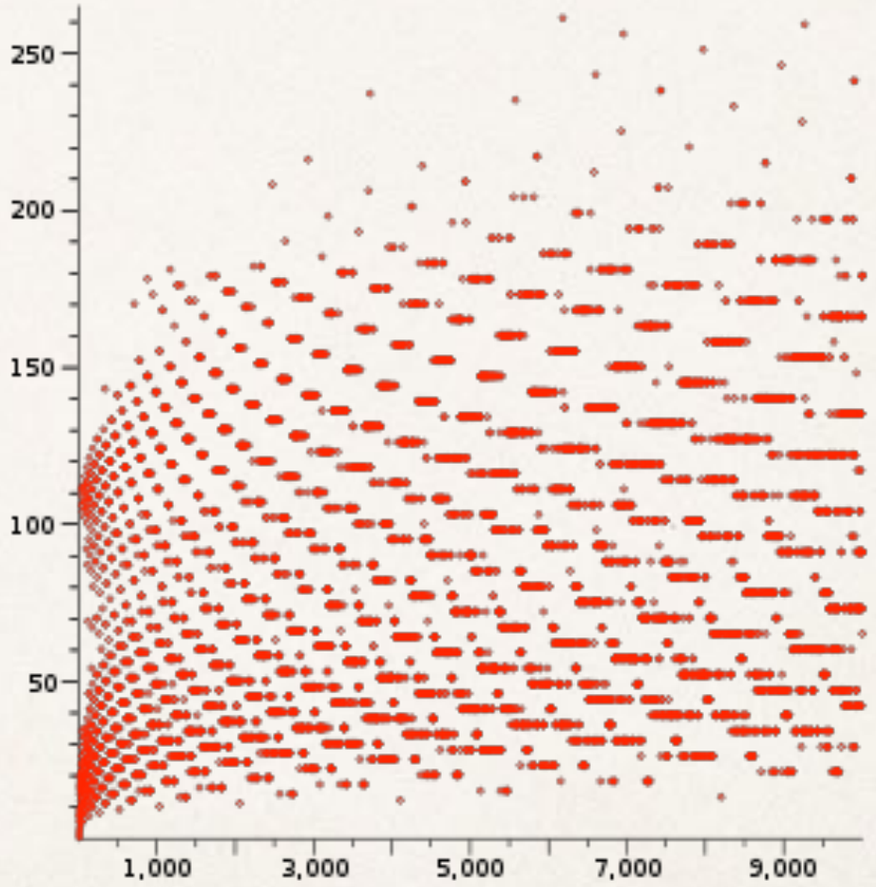
- The Hailstone function:

$$f(n) = \begin{cases} n/2 & \text{if } n \text{ even,} \\ 3n+1 & \text{if } n \text{ odd.} \end{cases}$$

- **Collatz conjecture**. Start with any positive integer x.
  Consider the sequence:

  x, f(x), f(f(x)), ...

  Eventually it reaches 1.



Stopping time

Modeling sequences via mutable data

```
(define (hailstone x)
  (define (next)
    (if (eq? (modulo x 2) 0)
        (begin (set! x (/ x 2)) x)
        (begin (set! x (+ (* 3 x) 1)) x)))
  next)

(define hseq (hailstone 100))
```

Environment contains x

Passes `next` back

- Models the hailstone sequence we discussed before. The environment produced by `hailstone` remembers the current x value.
- The next function updates x and returns the new value.

The hailstone sequence in hidden mutable action

```
> (define hseq (hailstone 25))
> (hseq)
76
> (hseq)
38
> (hseq)
19
> (hseq)
58
> (hseq)
29
> (hseq)
88
> (hseq)
44
> (hseq)
22
> (hseq)
11
> (hseq)
34
> (hseq)
17
> (hseq)
52
> (hseq)
26
> (hseq)
13
> (hseq)
40
> (hseq)
20
> (hseq)
10
> (hseq)
5
> (hseq)
16
> (hseq)
8
> (hseq)
4
> (hseq)
2
> (hseq)
1
>
```

Environment diagrams: The now and future life of a binding

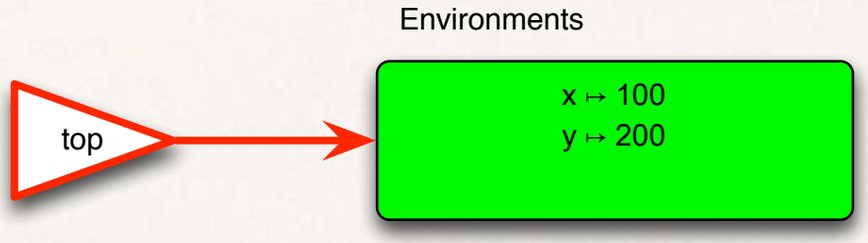
- **Environment inheritance**: When an environment is formed, it inherits all bindings from the environment in which it is defined. However, local bindings shadow bindings they inherit.

- This happens anytime a function is called with an argument whose name coincides with a name in the environment in which the function is defined.

```
(define x 100)
(define y 200)
(define (f x) (+ x y))
(f 8)
```

```
208
```

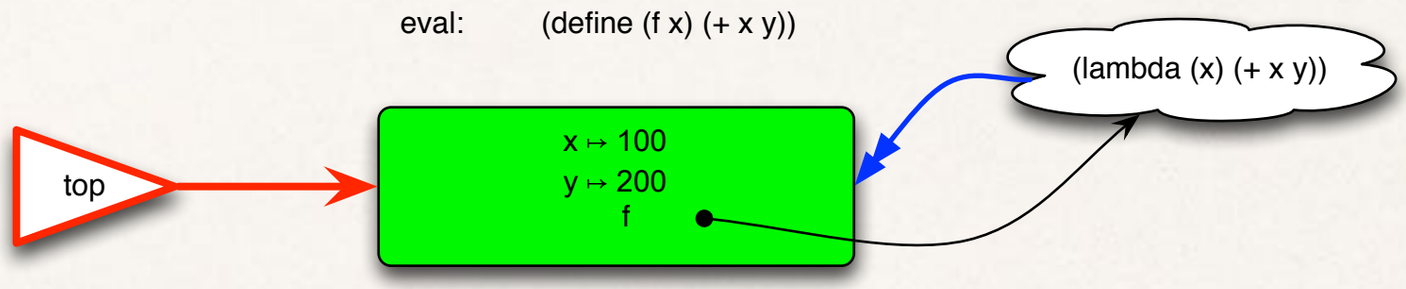
Environment diagrams: In Picture




```
(define x 100)
(define y 200)
(define (f x) (+ x y))
(f 8)
```

208

Environment diagrams: In Picture

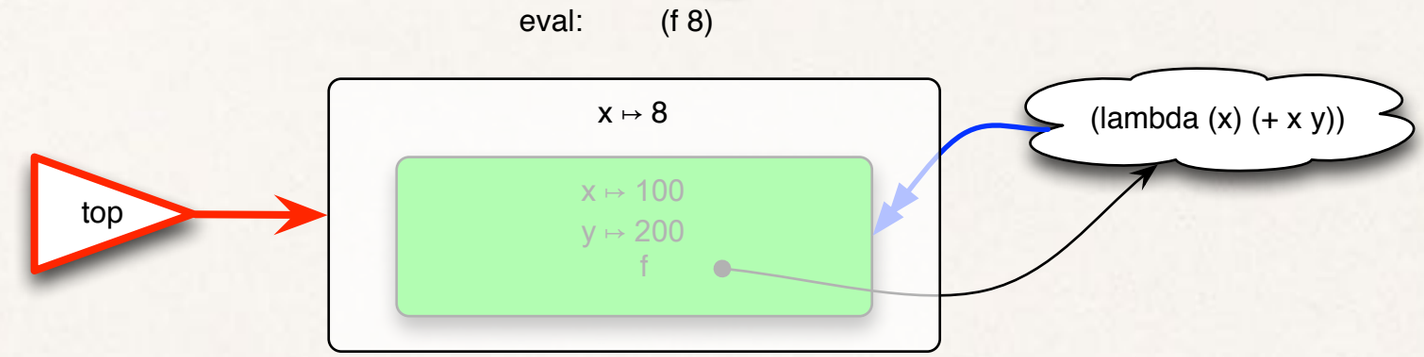



```
(define x 100)
(define y 200)
(define (f x) (+ x y))
(f 8)
```

208

Environment diagrams: In Picture

This "white" environment inherits bindings from its parent



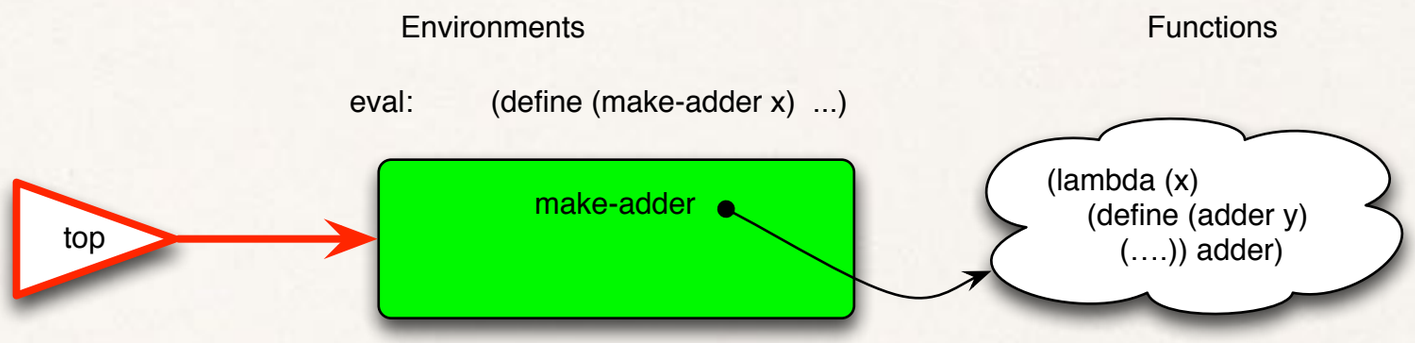
New binding of x *shadows* the inherited one

Function captures defining environment (*blue arrow*)

```
(define x 100)
(define y 200)
(define (f x) (+ x y))
(f 8)
```

208

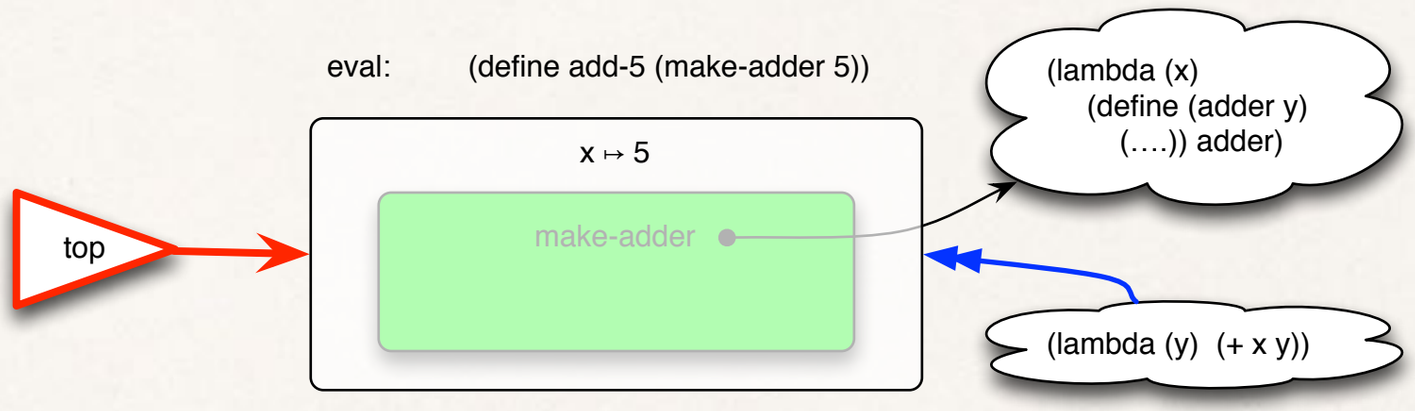
Environment diagrams: In Picture



make-adder added to "top" environment

```
(define (make-adder x)
  (define (adder y)(+ x y))
  adder)
(define add-5 (make-adder 5))
(add-5 100)
105
(define add-6 (make-adder 6))
(add-6 100)
106
```

Environment diagrams: In Picture



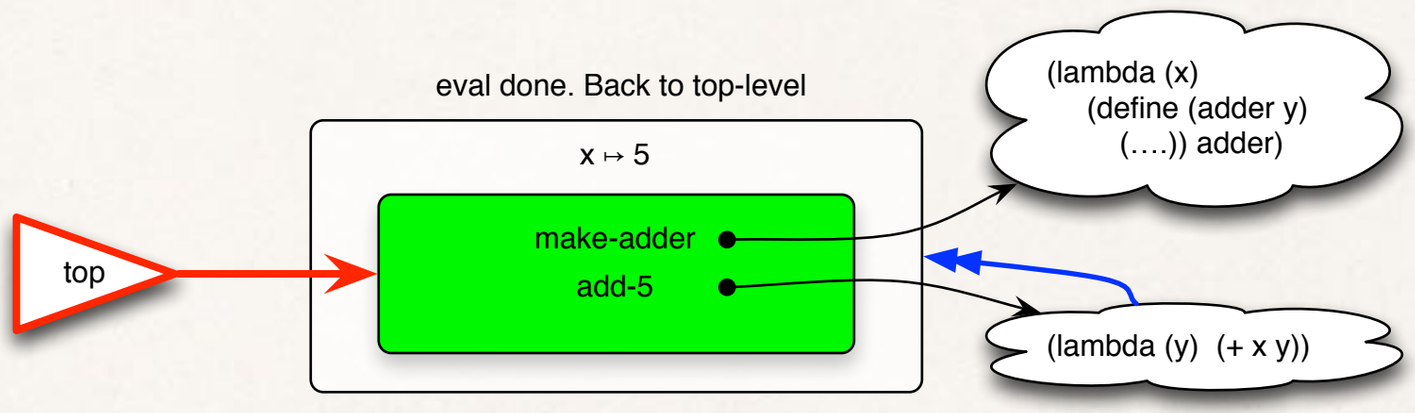
make-adder calls create new env. with x=5

adder lambda created in that new env.

adder lambda *refers to* defining "white" env.

```
(define (make-adder x)
  (define (adder y)(+ x y))
  adder)
(define add-5 (make-adder 5))
(add-5 100)
105
(define add-6 (make-adder 6))
(add-6 100)
106
```

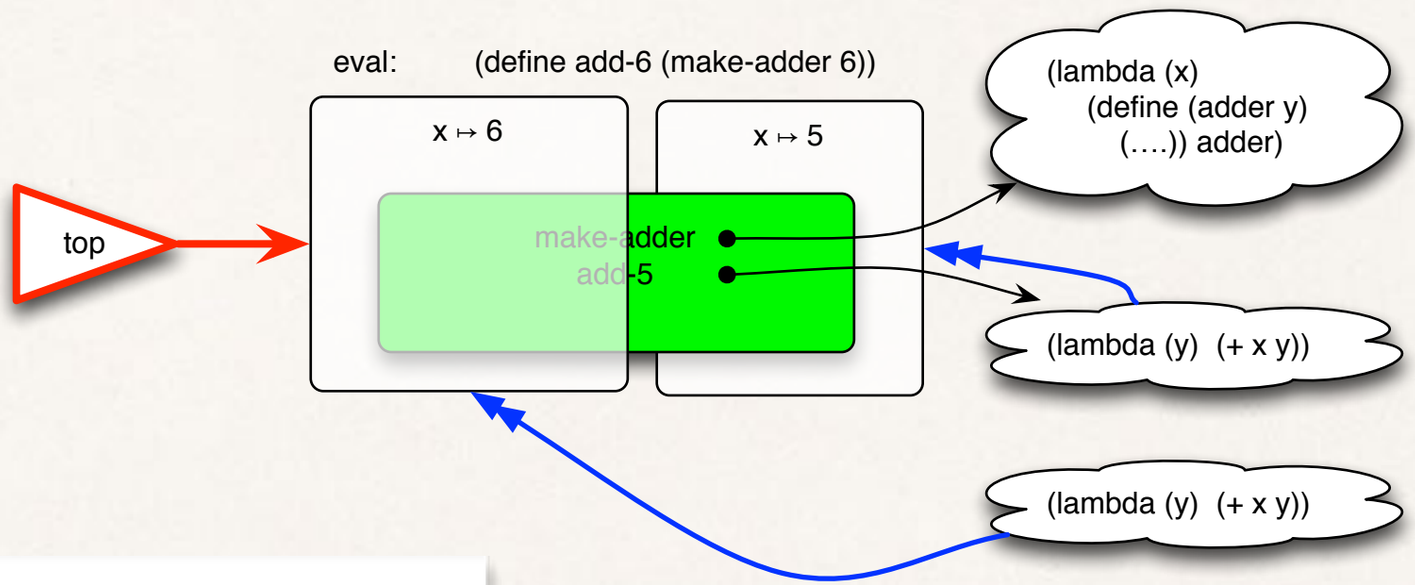
Environment diagrams: In Picture



back top “top” env. Bind add-5 to returned lambda!

```
(define (make-adder x)
  (define (adder y)(+ x y))
  adder)
(define add-5 (make-adder 5))
(add-5 100)
105
(define add-6 (make-adder 6))
(add-6 100)
106
```

Environment diagrams: In Picture



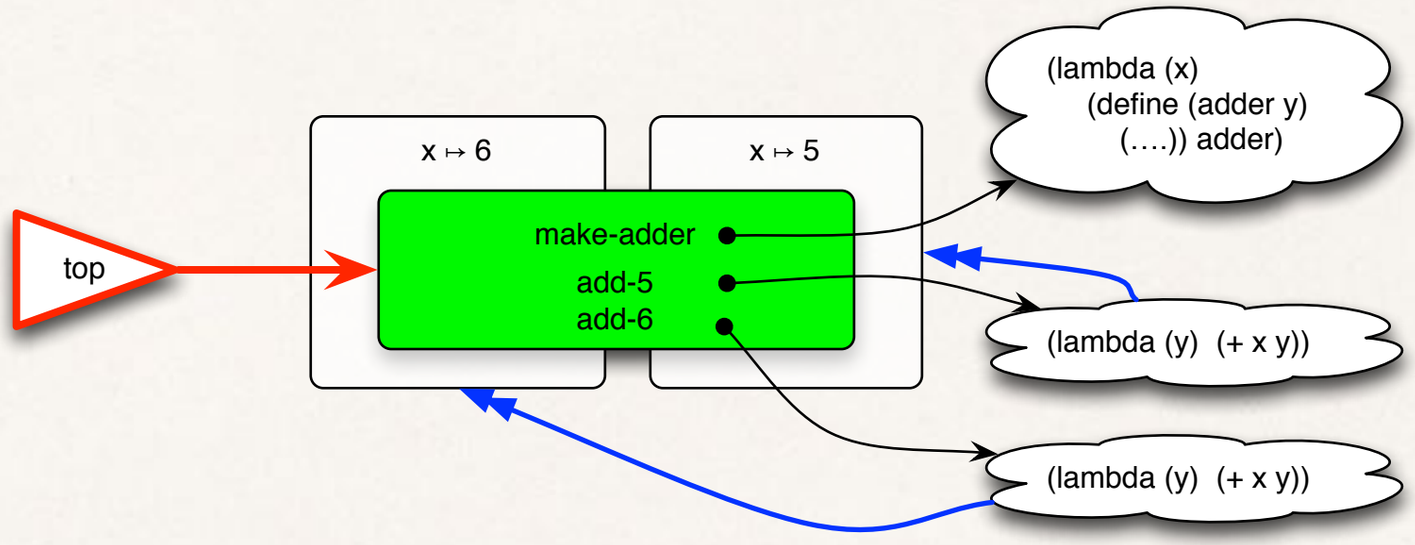
[Skip (add-5 100) for now]

maker-adder called with 6
Create new env with x=6

Define lambda in new env!

```
(define (make-adder x)
  (define (adder y)(+ x y))
  adder)
(define add-5 (make-adder 5))
(add-5 100)
105
(define add-6 (make-adder 6))
(add-6 100)
106
```

Environment diagrams: In Picture

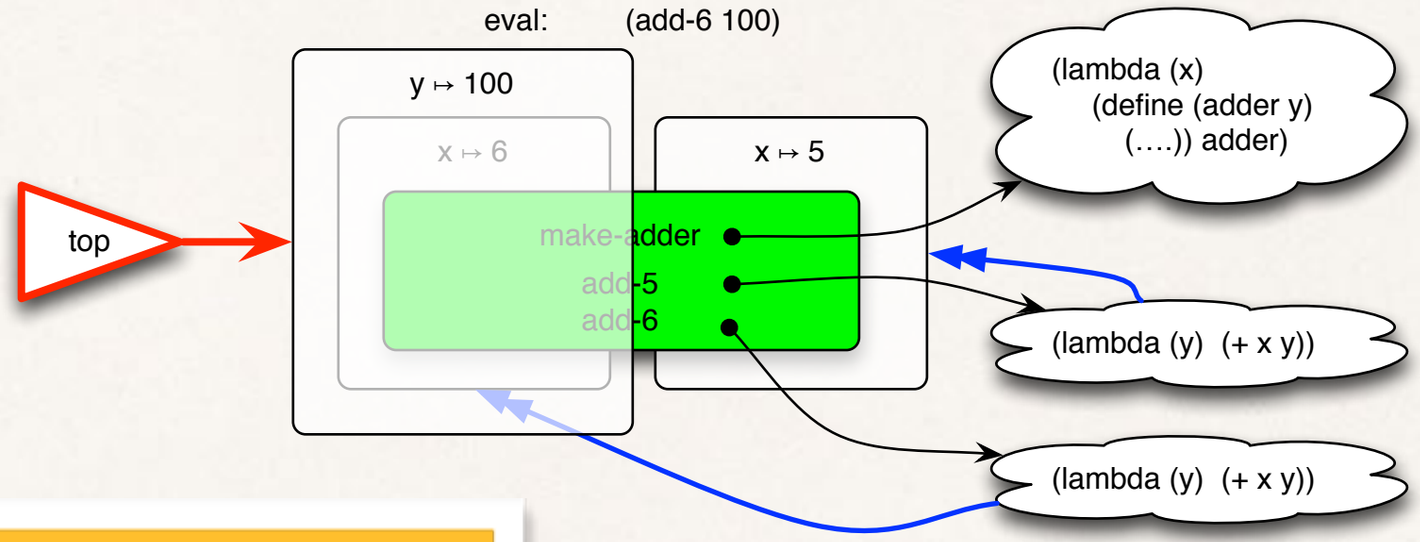


Return to “top” env.
Bind add-6 to returned lambda

Note how new lambda refers to “hidden env”

```
(define (make-adder x)
  (define (adder y)(+ x y))
  adder)
(define add-5 (make-adder 5))
(add-5 100)
105
(define add-6 (make-adder 6))
(add-6 100)
106
```

Environment diagrams: In Picture



Final eval of (add-6 100)

Create new env with y=100

Note how “x” will be picked up from env of lambda

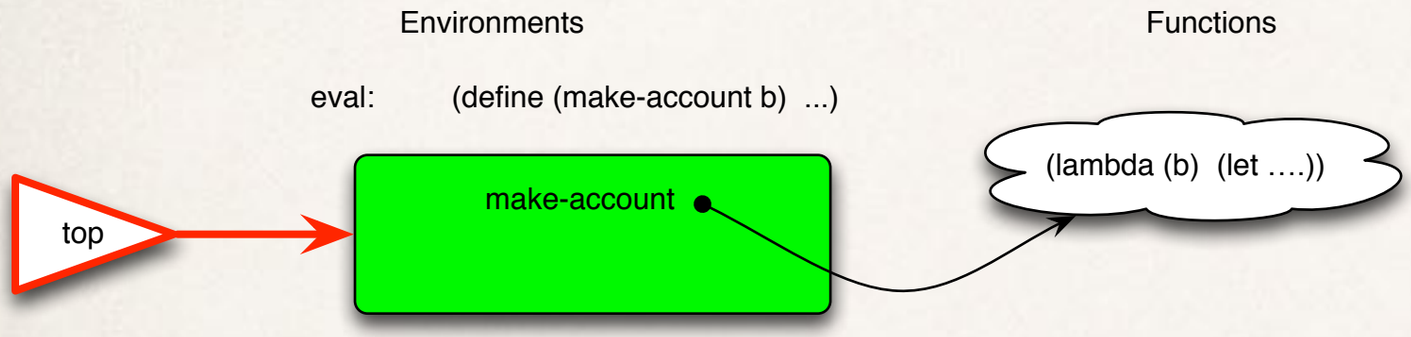
```
(define (make-adder x)
  (define (adder y)(+ x y))
  adder)
(define add-5 (make-adder 5))
(add-5 100)
105
(define add-6 (make-adder 6))
(add-6 100)
106
```

Simple Object Example

```
(define (make-account b)
  (let ((balance b))
    (define (get) balance)
    (define (withdraw x) (set! balance (- balance x)))
    (define (deposit  x) (set! balance (+ balance x)))
    (define (dispatch m)
      (cond ((eq? m 'get) get)
            ((eq? m 'withdraw) withdraw)
            ((eq? m 'deposit) deposit)))
    dispatch))

(define a (make-account 100))
((a 'deposit) 50)
```

Simple Object Example

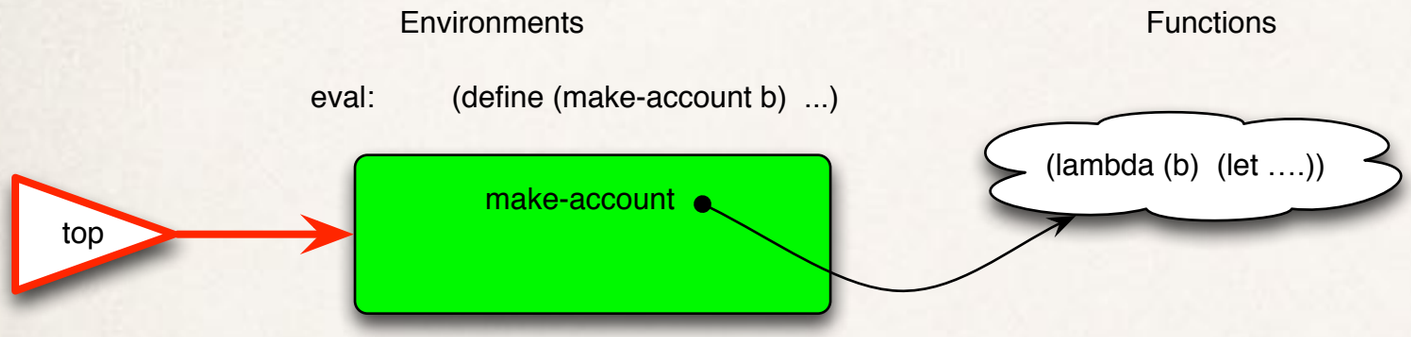


define object creator

```
(define (make-account b)
  (let ((balance b))
    (define (get) balance)
    (define (withdraw x) (set! balance (- balance x)))
    (define (deposit  x) (set! balance (+ balance x)))
    (define (dispatch m)
      (cond ((eq? m 'get) get)
            ((eq? m 'withdraw) withdraw)
            ((eq? m 'deposit) deposit)))
    dispatch))

(define a (make-account 100))
((a 'deposit) 50)
```

Simple Object Example

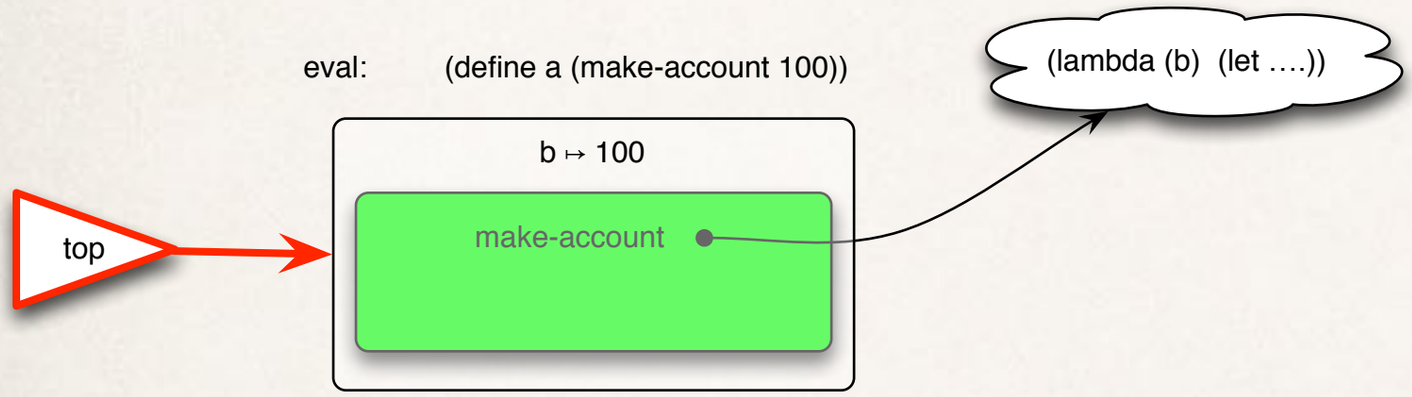


call object creator

```
(define (make-account b)
  (let ((balance b))
    (define (get) balance)
    (define (withdraw x) (set! balance (- balance x)))
    (define (deposit  x) (set! balance (+ balance x)))
    (define (dispatch m)
      (cond ((eq? m 'get) get)
            ((eq? m 'withdraw) withdraw)
            ((eq? m 'deposit) deposit)))
    dispatch))

(define a (make-account 100))
((a 'deposit) 50)
```

Simple Object Example

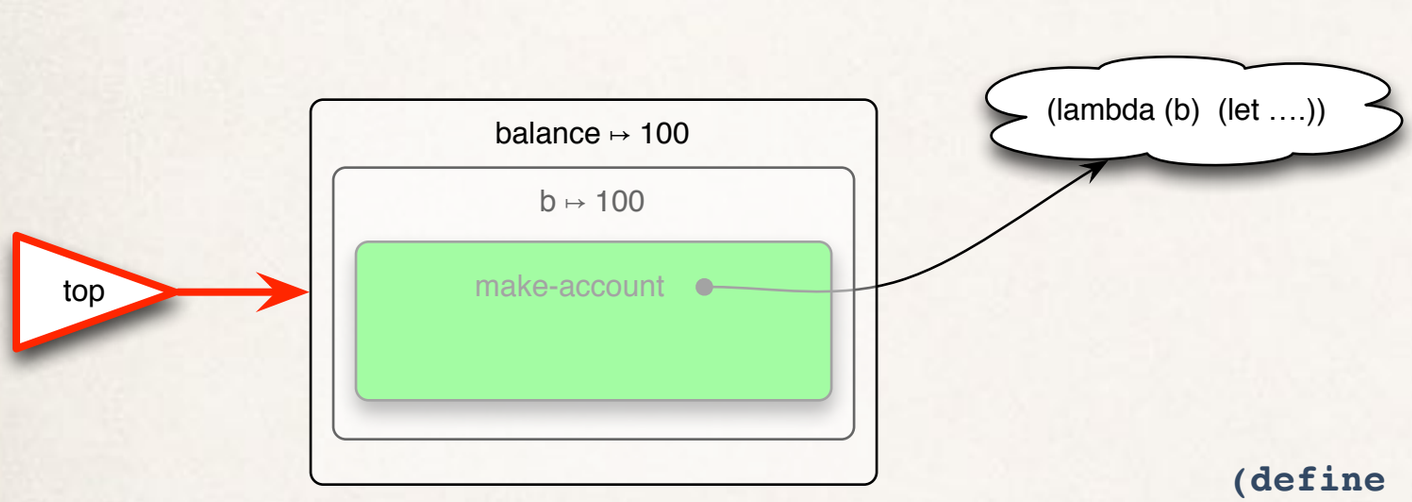


call create env with b=100

```
(define (make-account b)
  (let ((balance b))
    (define (get) balance)
    (define (withdraw x) (set! balance (- balance x)))
    (define (deposit  x) (set! balance (+ balance x)))
    (define (dispatch m)
      (cond ((eq? m 'get) get)
            ((eq? m 'withdraw) withdraw)
            ((eq? m 'deposit) deposit)))
    dispatch))

(define a (make-account 100))
((a 'deposit) 50)
```

Simple Object Example

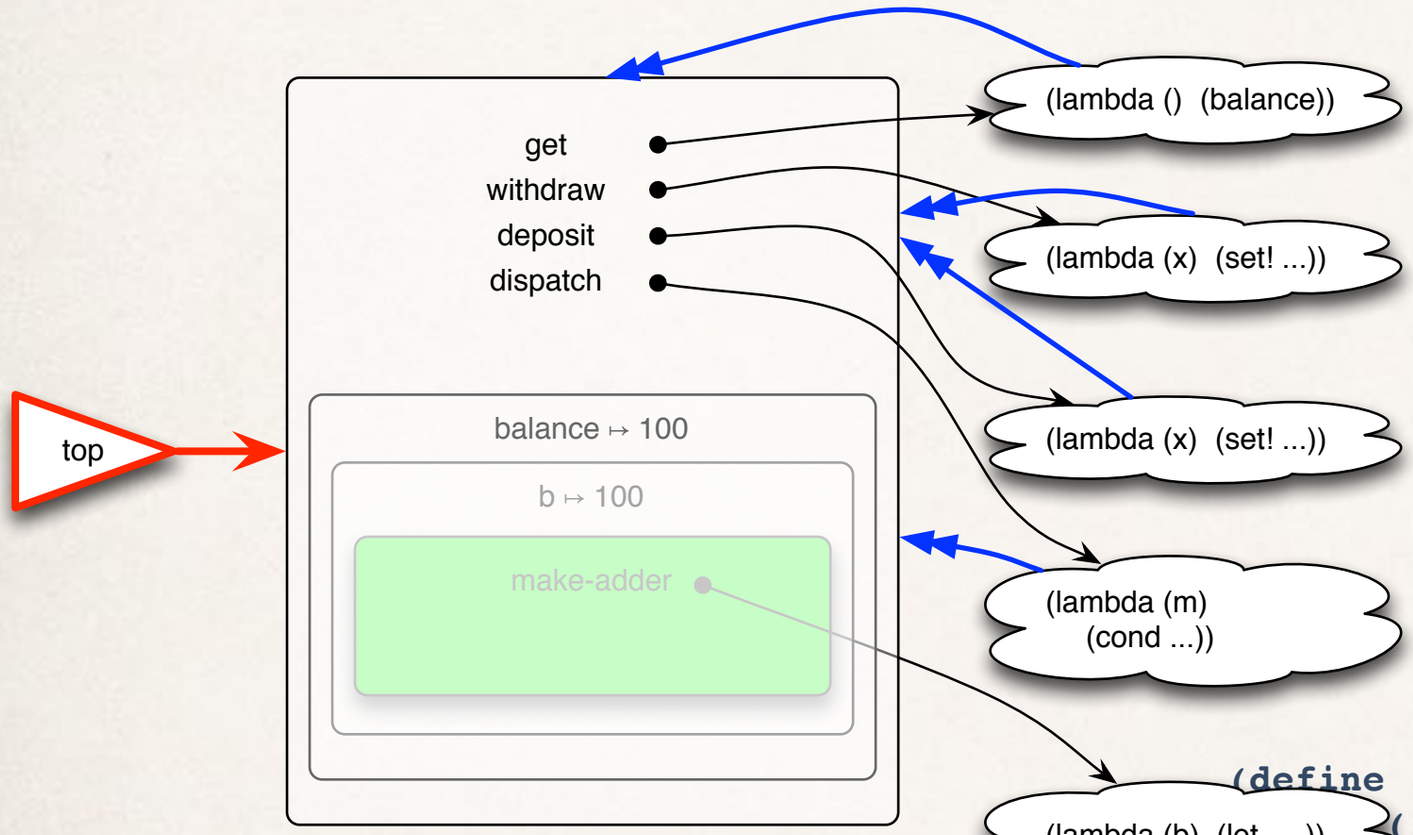


let create nested env
with balance=100

```
(define (make-account b)
  (let ((balance b))
    (define (get) balance)
    (define (withdraw x) (set! balance (- balance x)))
    (define (deposit  x) (set! balance (+ balance x)))
    (define (dispatch m)
      (cond ((eq? m 'get) get)
            ((eq? m 'withdraw) withdraw)
            ((eq? m 'deposit) deposit)))
    dispatch))

(define a (make-account 100))
((a 'deposit) 50)
```

Simple Object Example

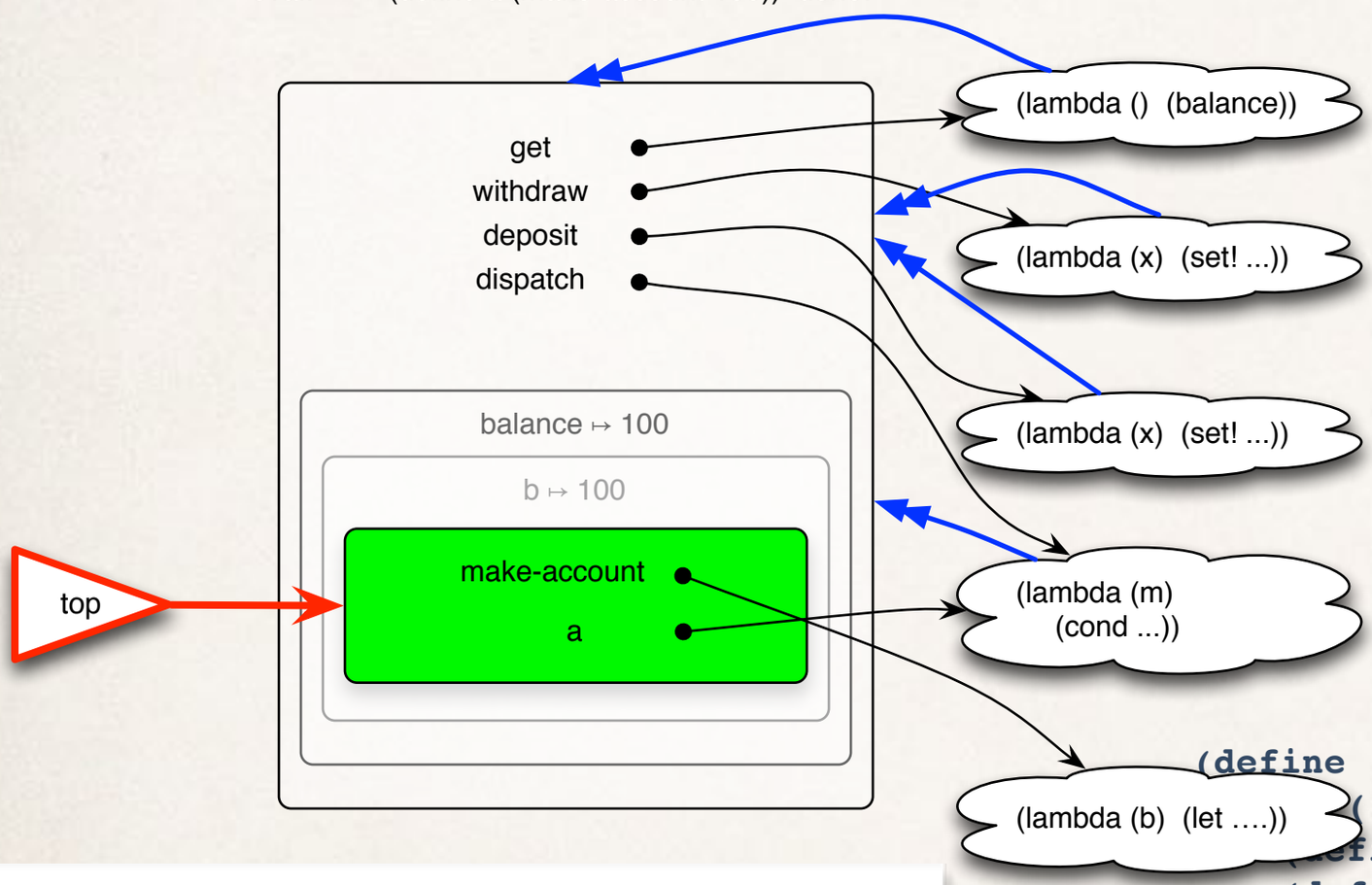


Method defined in nested env!

```
(define (make-account b)
  ((balance b))
  (define (get) balance)
  (define (withdraw x) (set! balance (- balance x)))
  (define (deposit  x) (set! balance (+ balance x)))
  (define (dispatch m)
    (cond ((eq? m 'get) get)
          ((eq? m 'withdraw) withdraw)
          ((eq? m 'deposit) deposit)))
  dispatch))

(define a (make-account 100))
((a 'deposit) 50)
```

Simple Object Example



Dispatcher returned & bound to ‘a’

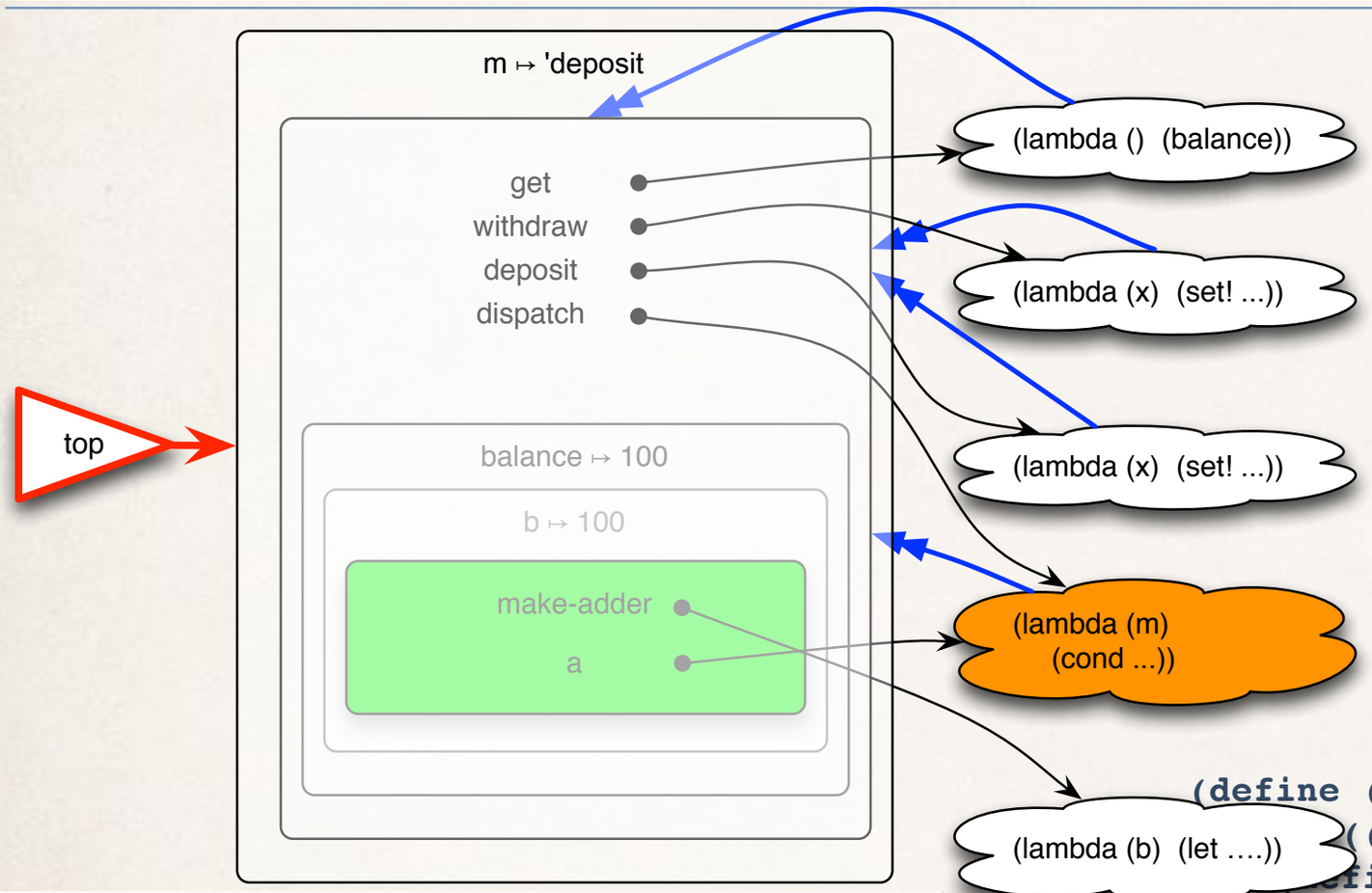
Dispatcher refers to defining env

```
(define (make-account b)
  ((balance b))
   (define (get) balance)
  (define (withdraw x) (set! balance (- balance x)))
  (define (deposit  x) (set! balance (+ balance x)))
  (define (dispatch m)
    (cond ((eq? m 'get) get)
          ((eq? m 'withdraw) withdraw)
          ((eq? m 'deposit) deposit)))
  dispatch))

(define a (make-account 100))
((a 'deposit) 50)
```

Simple Object Example

eval: ((a 'deposit) 50)

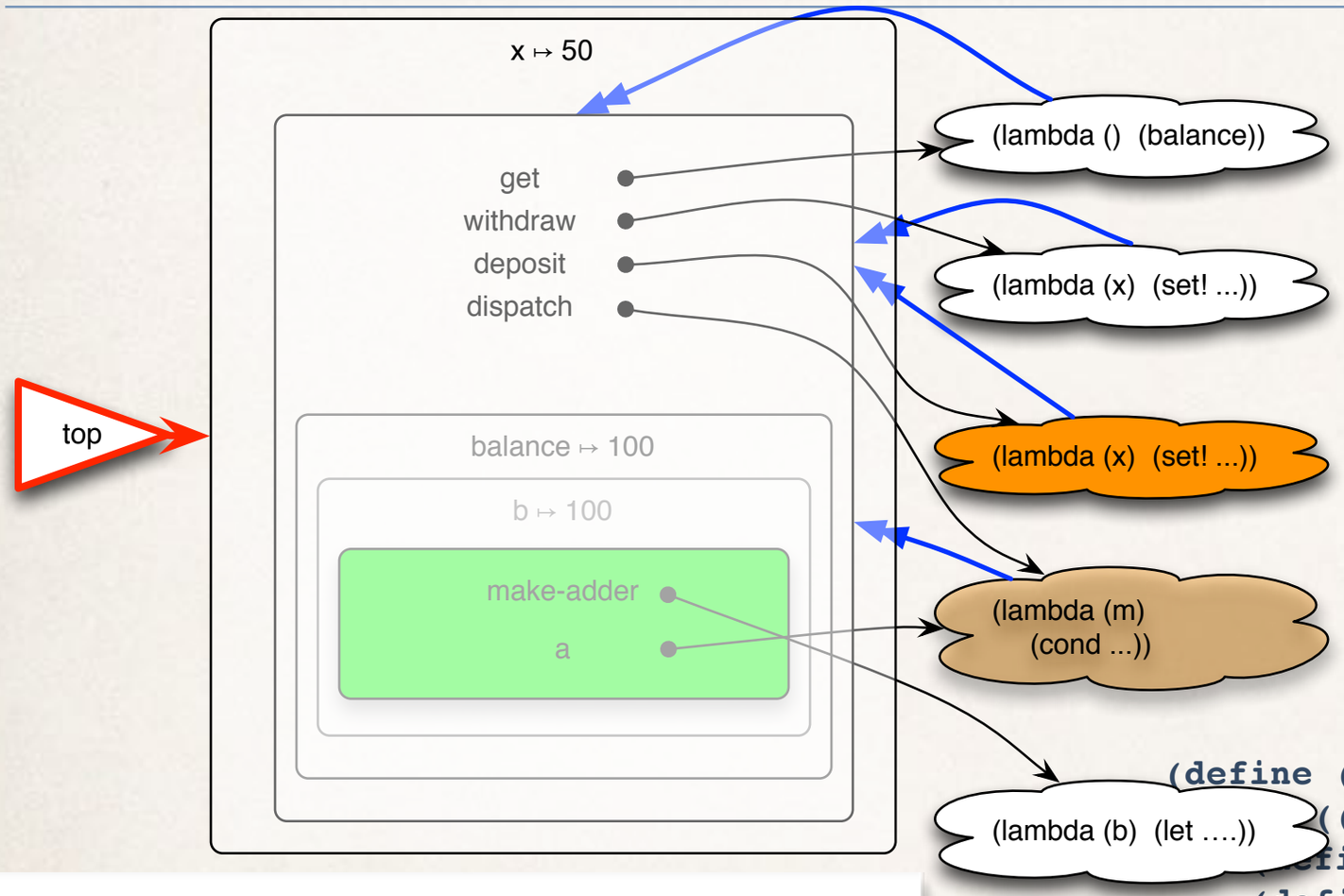


Dispatcher called with ‘deposit

```
(define (make-account b)
  ((balance b))
  (define (get) balance)
  (define (withdraw x) (set! balance (- balance x)))
  (define (deposit  x) (set! balance (+ balance x)))
  (define (dispatch m)
    (cond ((eq? m 'get) get)
          ((eq? m 'withdraw) withdraw)
          ((eq? m 'deposit) deposit)))
  dispatch))

(define a (make-account 100))
((a 'deposit) 50)
```

Simple Object Example

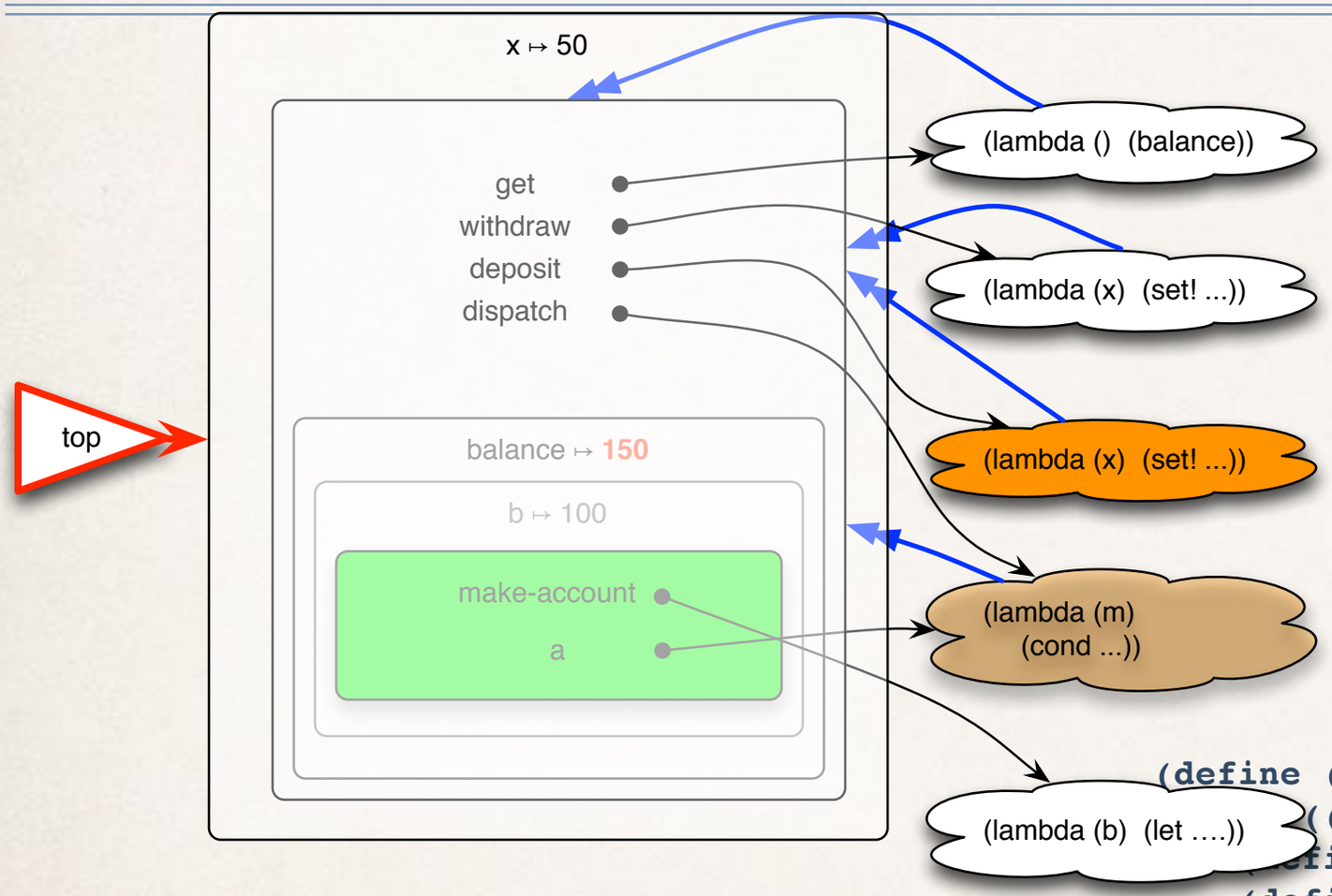


Returned method called with x=50

```
(define (make-account b)
  ((balance b))
   (define (get) balance)
  (define (withdraw x) (set! balance (- balance x)))
  (define (deposit  x) (set! balance (+ balance x)))
  (define (dispatch m)
    (cond ((eq? m 'get) get)
          ((eq? m 'withdraw) withdraw)
          ((eq? m 'deposit) deposit)))
  dispatch))

(define a (make-account 100))
((a 'deposit) 50)
```

Simple Object Example

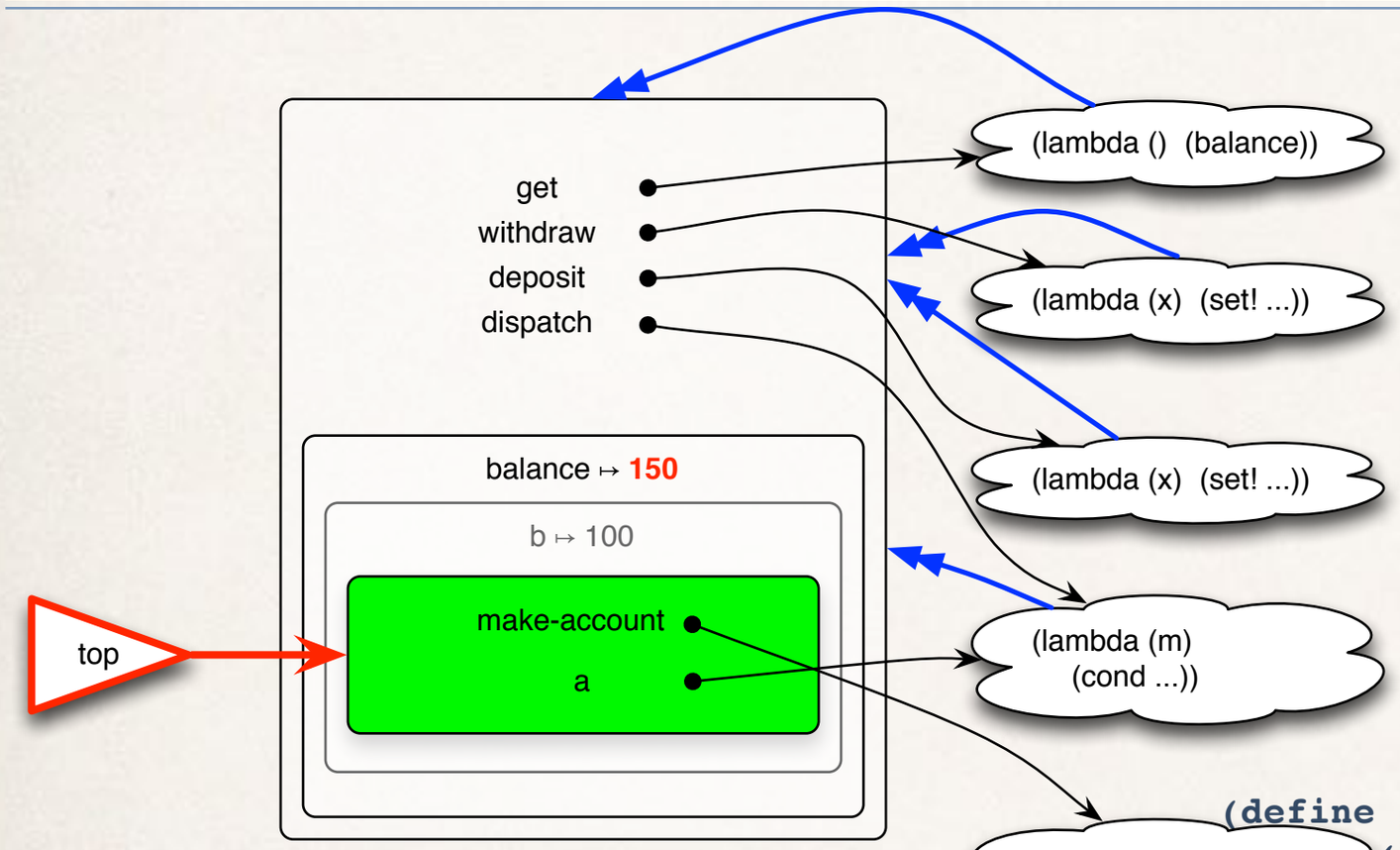


nested env modified!

```
(define (make-account b)
  ((balance b))
   (define (get) balance)
   (define (withdraw x) (set! balance (- balance x)))
   (define (deposit  x) (set! balance (+ balance x)))
   (define (dispatch m)
     (cond ((eq? m 'get) get)
           ((eq? m 'withdraw) withdraw)
           ((eq? m 'deposit) deposit)))
   dispatch))

(define a (make-account 100))
((a 'deposit) 50)
```

Simple Object Example



Return to top-level.
Change is "hidden inside"

```
(define (make-account b)
  ((balance b))
   (define (get) balance)
   (define (withdraw x) (set! balance (- balance x)))
   (define (deposit  x) (set! balance (+ balance x)))
   (define (dispatch m)
     (cond ((eq? m 'get) get)
           ((eq? m 'withdraw) withdraw)
           ((eq? m 'deposit) deposit)))
   dispatch))

(define a (make-account 100))
((a 'deposit) 50)
```

Mutable structured data

- Recall the Stack ADT. It model a stack of objects: you can examine the top object, remove (pop) it, or push on a new object.
  - empty?(S): returns true if the stack is empty,
  - top(S): returns the top object in the stack, without removing it
  - pop(S): removes the top element of the stack.
  - push(x, S): pushes the element x onto the stack.

A stack object that implements this ADT

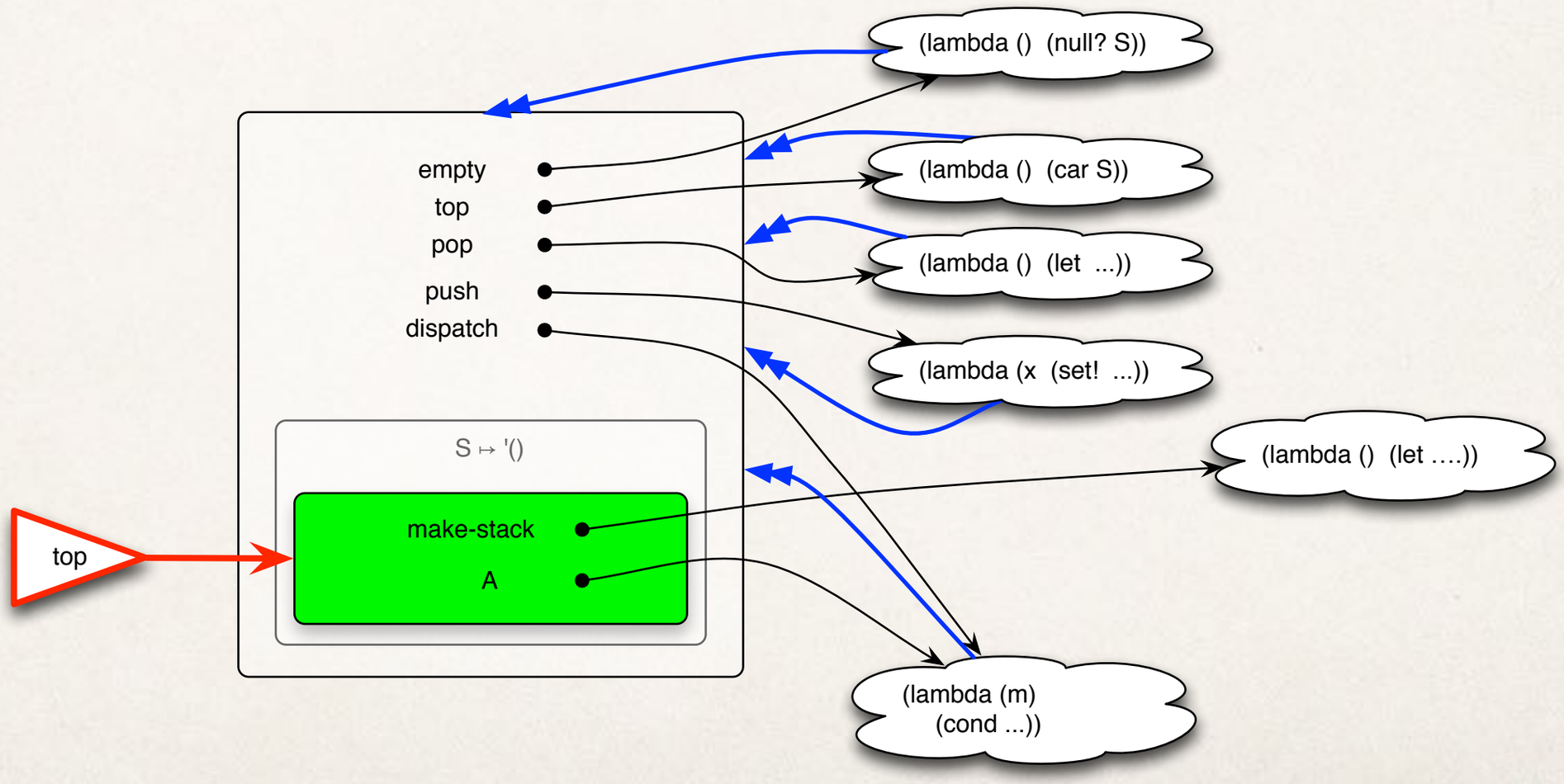
```
(define (make-stack)
  (let ((S '()))
    (define (empty?) (null? S))
    (define (top) (car S))
(define (pop) (let ((top (car S)))
                (begin (set! S (cdr S))
                       top)))
(define (push x) (set! S (cons x S)))
(define (dispatcher method)
  (cond ((eq? method 'top) top)
        ((eq? method 'pop) pop)
        ((eq? method 'push) push)
        ((eq? method 'empty) empty?)))
dispatcher))
```

What complicates pop?

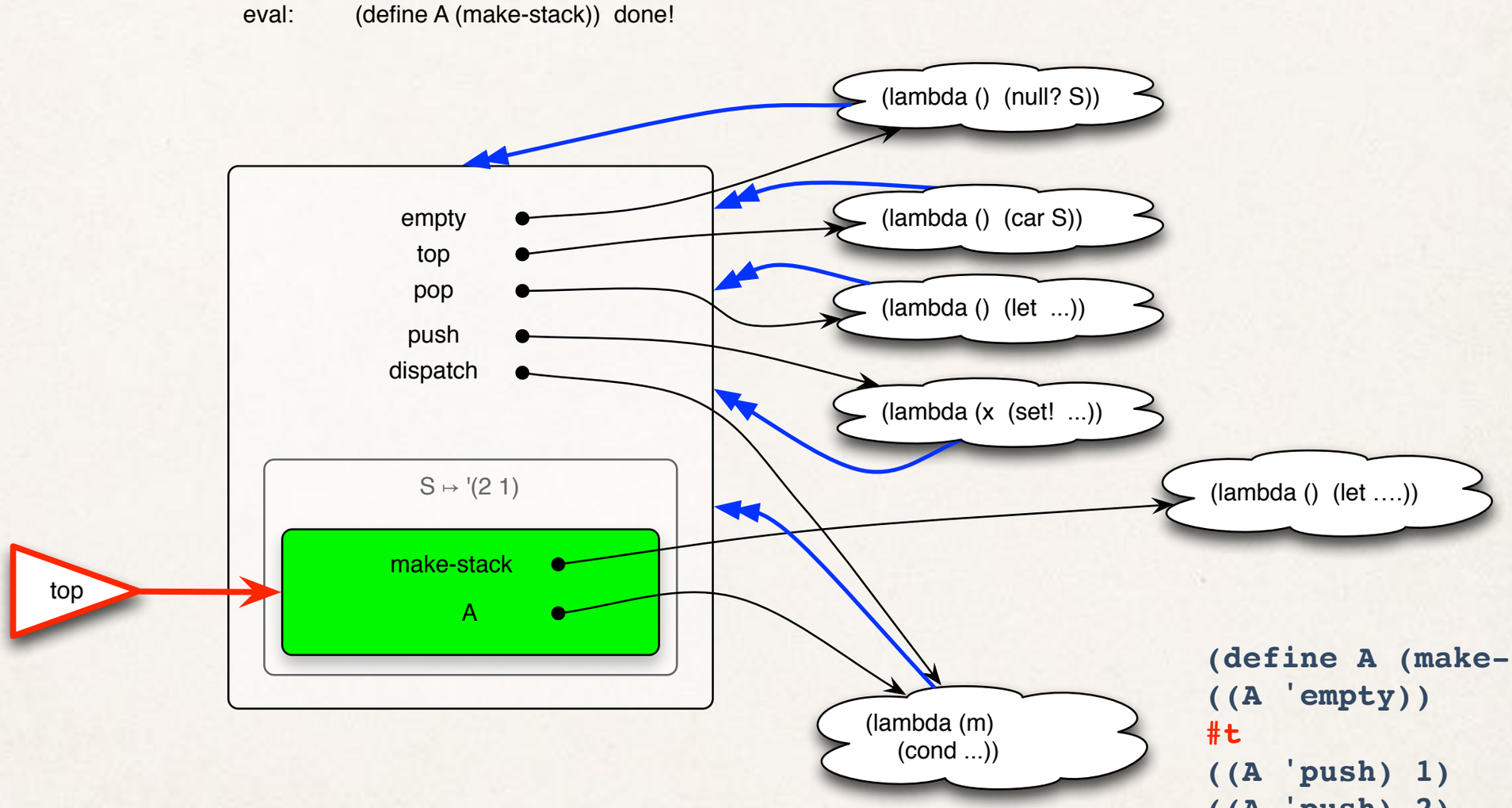
Stack Picture

Stack created from make-stack

eval: (define A (make-stack)) done!



The Stack object in action




```
(define A (make-stack))
((A 'empty))
#t
((A 'push) 1)
((A 'push) 2)
((A 'top))
2
((A 'pop))
2
((A 'top))
1
```

Stack after the two “push”

Implementing an efficient queue

- Recall the Queue ADT. It models a queue of objects: you can examine the first object, remove it, or enqueue on a new object onto the end.
  - empty?(Q): returns true if the queue is empty, otherwise false
  - front(Q): returns the front object in the queue, without removing it
  - dequeue(Q): removes the top element of the queue.
  - enqueue(x, Q): pushes the element x onto the queue.

Complaint about our previous implementation?

- Efficiency: If implemented in terms of a list, front and dequeue are fast, but placing an object at the end of the queue is costly--you need to traverse the entire list.

- Problem: It seems as though there is just no way to improve upon this with our current technology: lists. Can we use the `cons`tructor in a fancier way?

- To talk about this clearly, we need a more precise picture of how SCHEME maintains pairs.

A detour on Scheme assignment semantics

A more nuanced model of the SCHEME pair

- Previously, we have discussed pairs abstractly: *a box with two slots* into which you can place any SCHEME value.

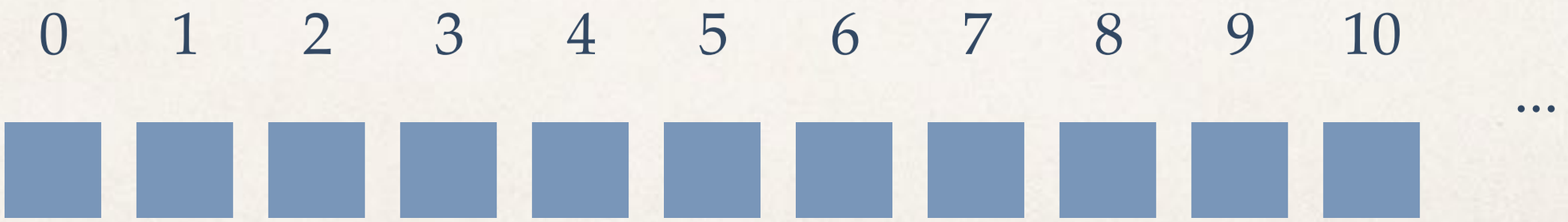
- In fact, SCHEME maintains a pair as a tuple of two *pointers*. This is handy because all pairs then have exactly the same size. In fact, these pointers are just names of memory addresses that contain the objects that are supposed to be **in** the pair.

- Similarly, a SCHEME variable, is really a pointer to the value to which it is assigned

Values and pointers in SCHEME

- You may think of the SCHEME interpreter as maintaining an array of memory cells. A memory cell can hold a primitive SCHEME value: a Boolean, a character, a token, a number, ... (In fact, it is slightly more complex than this: some primitive values may require several adjacent cells.)

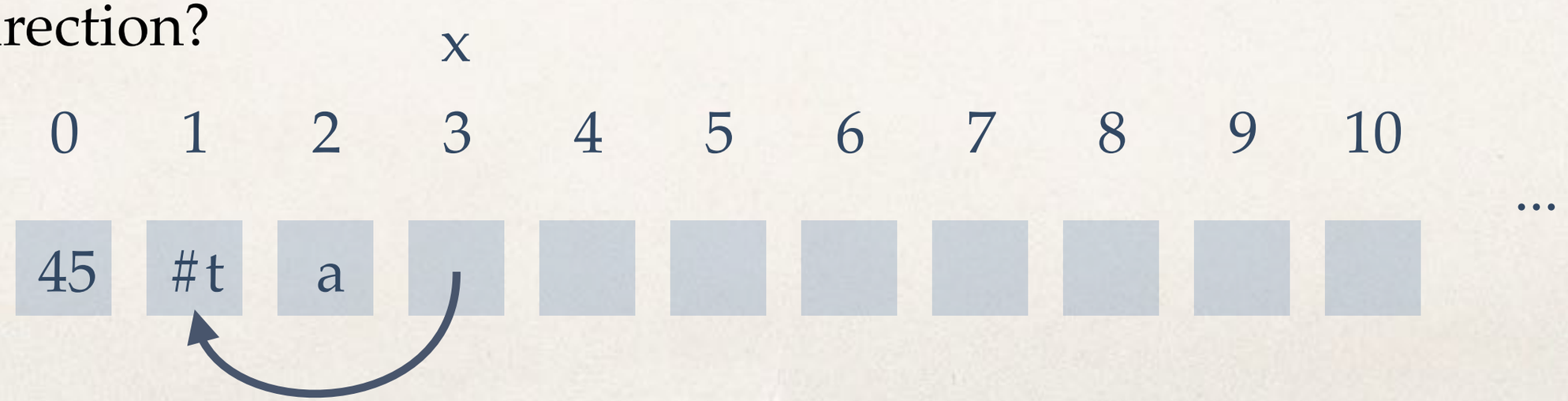
- Each memory cell has a unique *address*--a number that the computer uses to refer to it.



Cells hold values or pointers

- A cell can hold an primitive value.

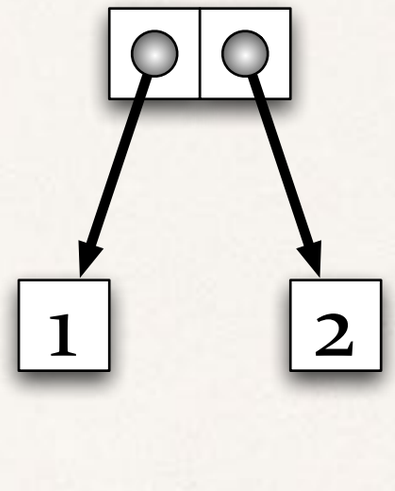
- A SCHEME variable is associated with a cell that holds *a pointer containing the name of another cell that holds its value*. Why the indirection?



Representing variables and pairs

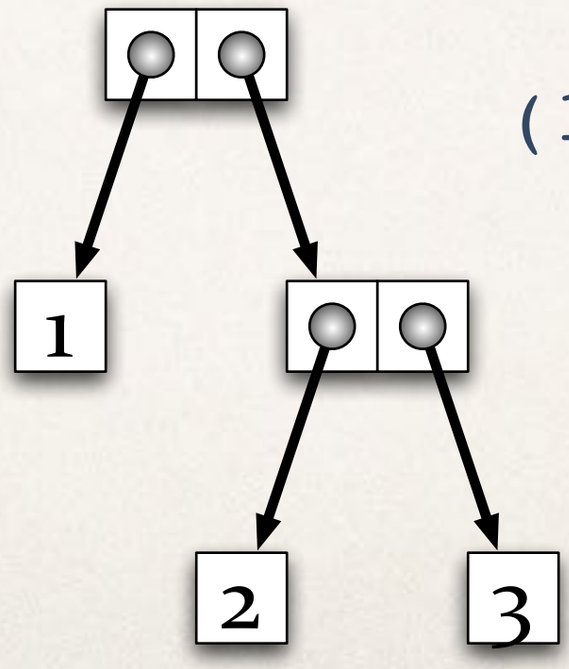
- A mnemonic way to represent a variable in SCHEME, then:

x ⟶ #t




```
(1 . 2)
```

- This same convention is used for pairs. A pair in SCHEME is represented as a tuple of *two pointers*, one pointing to the contents of the car, one to the cdr. Thus you may think of a pair as occupying two adjacent memory locations, each a pointer. Thus...



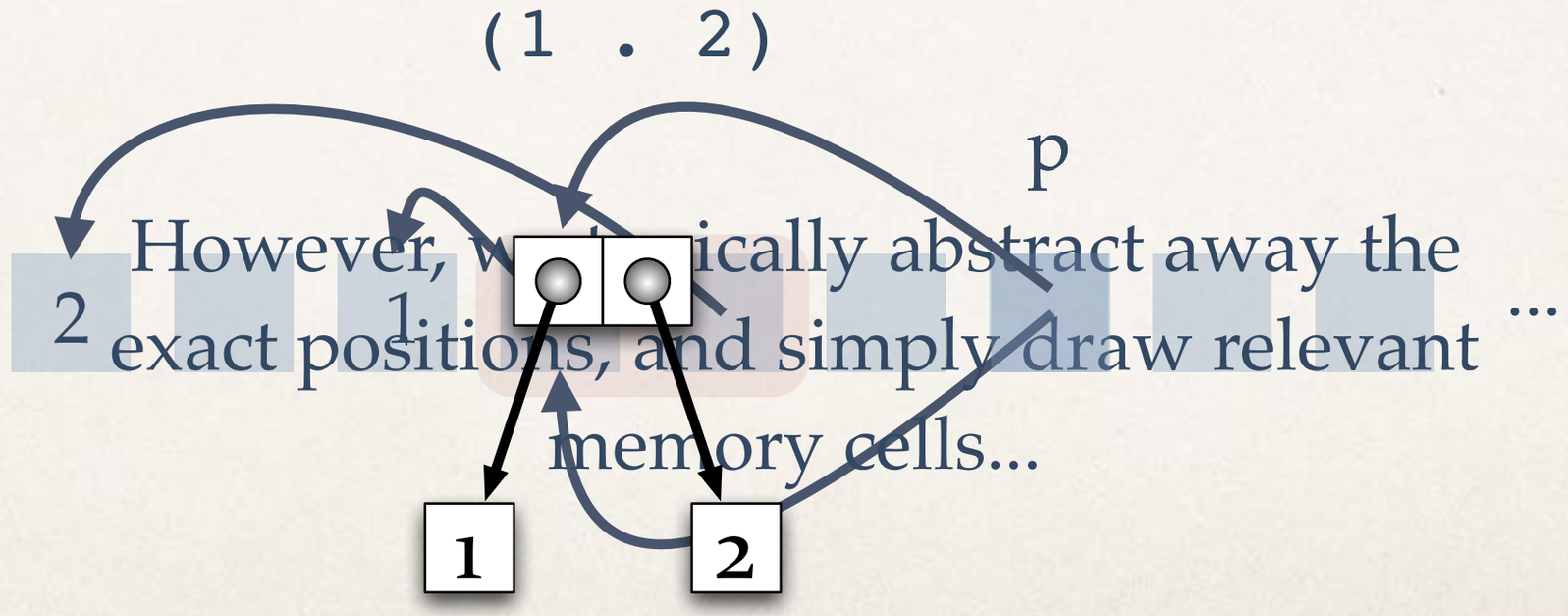

```
(1 . (2 . 3))
```

Pairs as *pairs of pointers*

- Consider the code snippet

```
> (define p (cons 1 2))
```

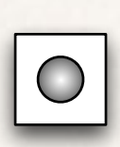
- This definition yields a memory layout something like this:



However, we typically abstract away the exact positions, and simply draw relevant memory cells...

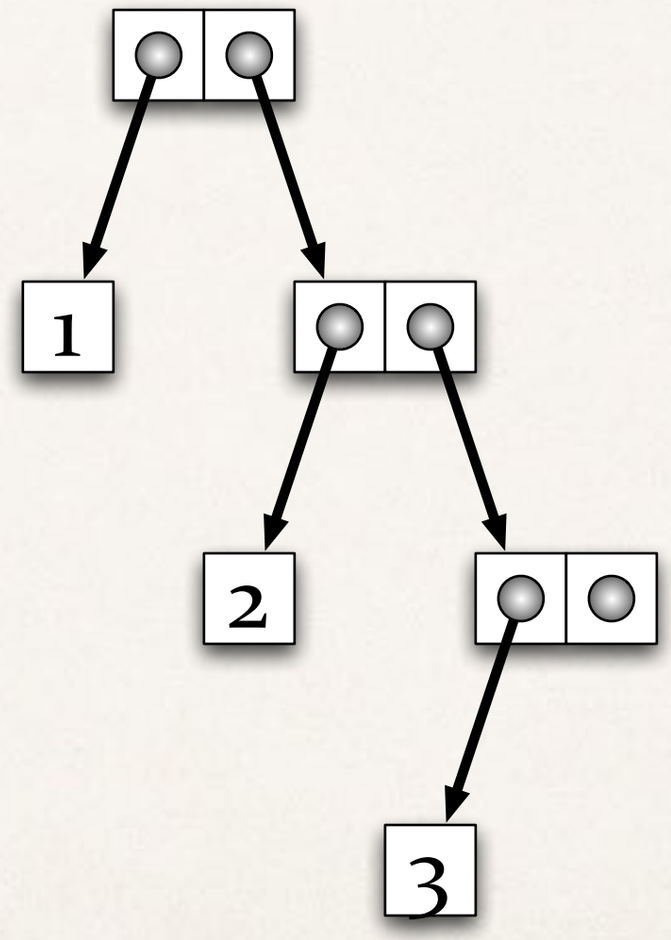
Recall the SCHEME convention for lists

- Scheme reserves a special null pointer.



- Then, lists are represented as nested pairs, where the car points to the first element of the list and the cdr points to the "rest" of the list.

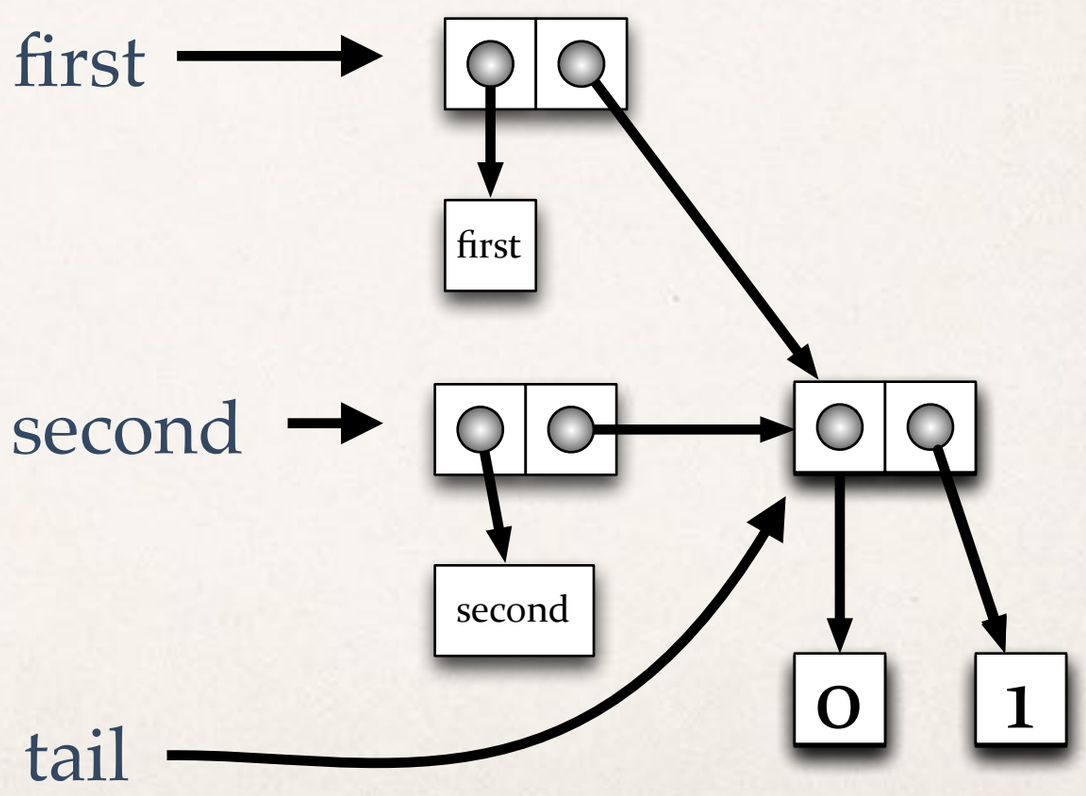
```
(1 . (2 . (3 . '()))
        or
(1 2 3)
```



You can infer the structure of cons(structs) from how they are created

```
> (define tail (cons 0 1))
> (define first (cons 'first tail))
> (define second (cons 'second tail))
> first
(first 0 . 1)
> second
(second 0 . 1)
```

Whoa! Notice that these structures overlap.

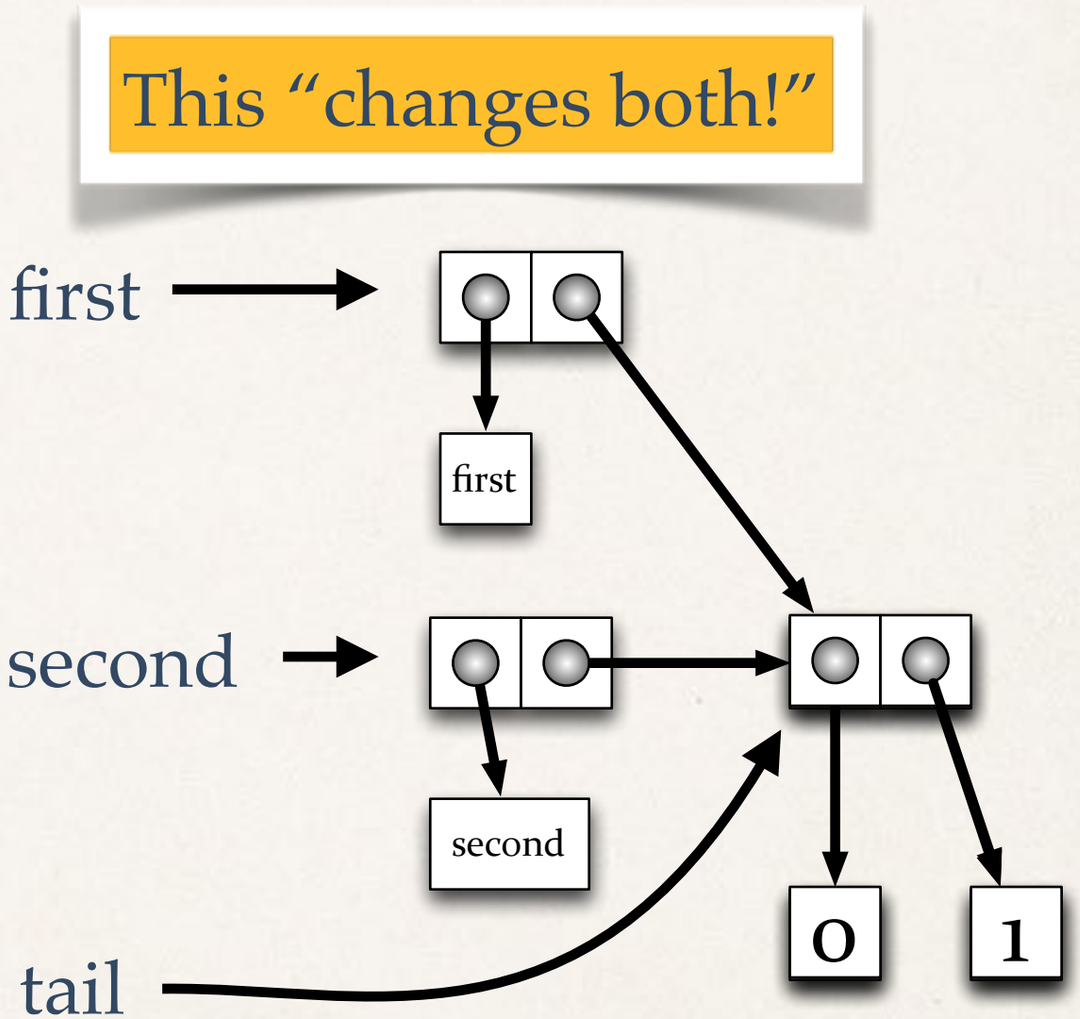


`set!`, `set-car!`, and `set-cdr!`

- You've seen `set!` in action (on a variable). It redirects the pointer so that it points to a new object.

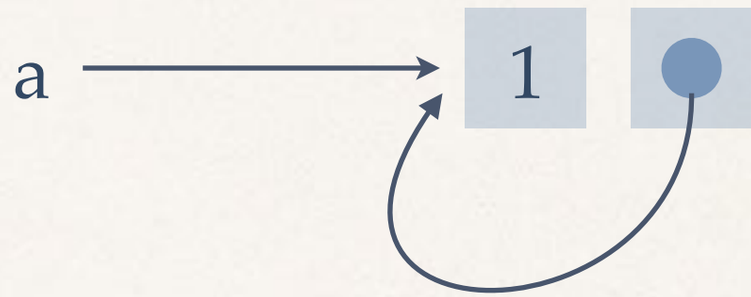
```
> first
(first 0 . 1)
> second
(second 0 . 1)
> (set-car! tail 1)
> first
(first 1 . 1)
> second
(second 1 . 1)
```

- There are analogous operations for destructively setting the car and cdr of a pair: `set-car!` and `set-cdr!`.



This can create (arbitrarily) complex, even circular, structures

```
> (define a (list 1))
> (set-cdr! a a)
> (car a)
1
> (cadr a)
1
> (caddr a)
1
> (caddr a)
1
> (cadddr a)
1
```

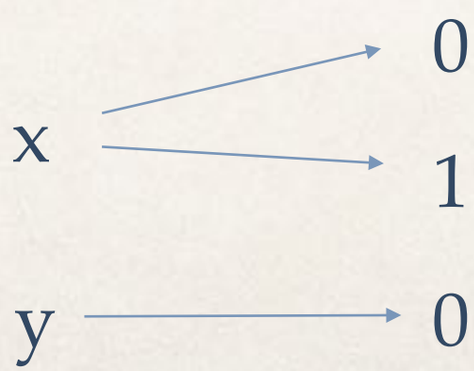


This creates a kind of infinite data structure.

Beware: The semantics of variable assignment depend on the value

- `(define var <expr>)` sets `var` to be the value returned by `<expr>`.

- If the value is an atomic scheme type (number, Boolean, etc.), this is easy to understand…



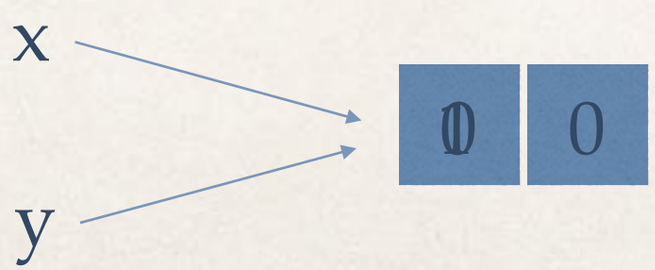

```
> (define x 0)
> (define y x)
> x
0
> y
0
> (set! x 1)
> x
1
> y
0
```

Beware: The semantics of variable assignment depend on the value

- `(define var <expr>)` sets `var` to be the value returned by `<expr>`.

```
> (define x (cons 0 0))
> (define y x)
> x
(0 . 0)
> y
(0 . 0)
> (set-car! x 1)
> x
(1 . 0)
> y
(1 . 0)
```

- If the value is structured, like a pair, this it is referred by a pointer: a new copy is not produced…



Mutable structured data in Scheme and Racket

- Scheme and Racket handle mutable structured data a little differently.
- To obtain the behavior we have discussed in class (and read about in SICP), you'll need to start your Racket buffer with

```
#lang rsr5
```

```
> (define a (cons 2 3))
> (car a)
2
> (set-car! a 4)
> a
(mcons 4 3)
```

Even so, racket will print pairs a little differently...

Mutable pairs in Racket

- The Racket designers mean business...a pair is either *mutable* or *functional* and never the twain shall meet.

- Pairs built with `cons` cannot be destructively assigned.

- A mutable pair is built with `mcons`, accessed with `mcar` and `mcdr`, destructive assignment via `set-mcar!` and `set-mcdr!`.

```
> (define a (mcons 1 2))
> (car a)
. . car: expects argument of type <pair>; given (mcons 1 2)
> (mcar a)
1
> (set-mcar! a 3)
> a
(mcons 3 2)
```

Can't mix and match

Mutable list infrastructure

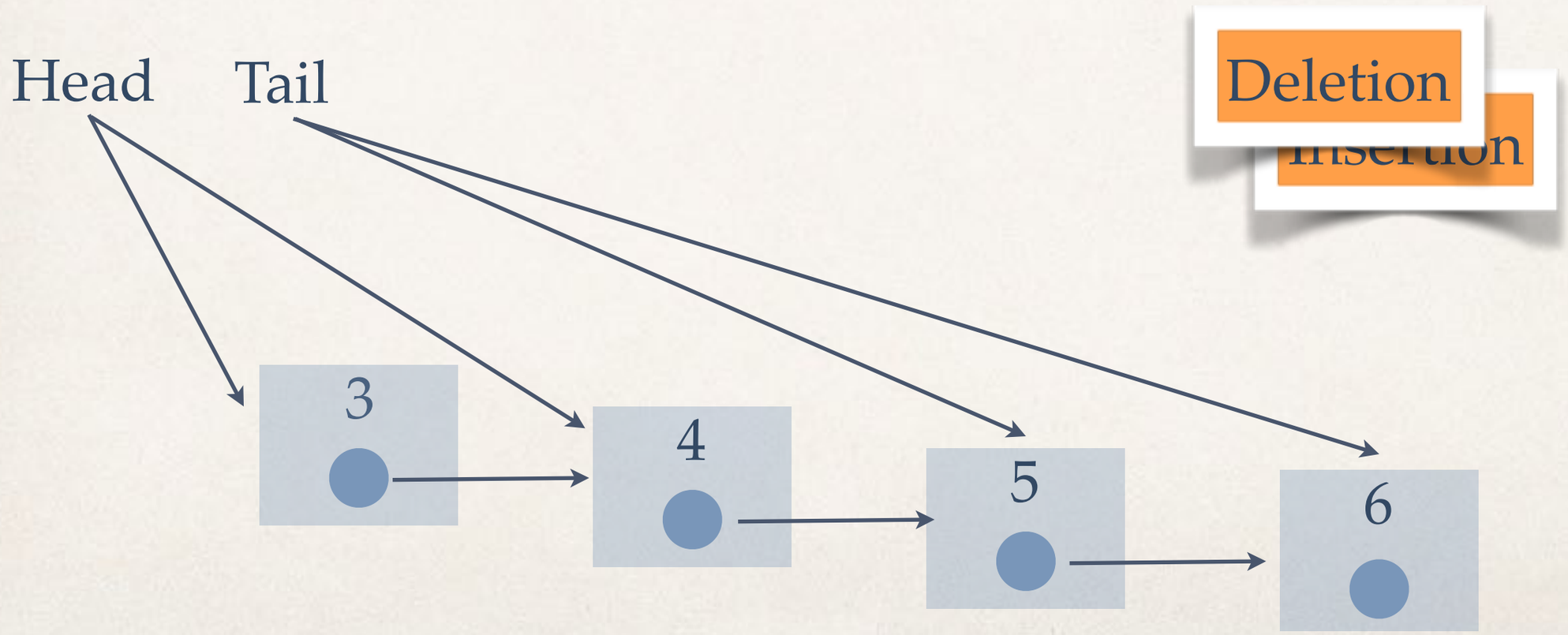
- Racket has infrastructure for mutable lists. As with pairs, lists and mutable lists (built from mutable pairs) must be kept separate.

- The infrastructure is loaded with:

```
> (require compatibility/mlist)
> (mlist 1 2 3 4)
(mcons 1 (mcons 2 (mcons 3 (mcons 4 '()))))
> (null? (mlist 1 2))
#f
> (mappend (mlist 1 2) '())
(mcons 1 (mcons 2 '()))
> (mmap (lambda (x) (* 2 x)) (mlist 1 2 3 4 5))
(mcons 2 (mcons 4 (mcons 6 (mcons 8 (mcons 10 '())))))
```

A more sophisticated queue data structure

- We will maintain a queue as a linked list that "remembers" its tail.
- Notice that we can enqueue and dequeue in a fixed amount of time (independent of number of elements in the queue) if we remember the tail.



Implementing a queue as a linked list in SCHEME

- Each "node" of our "linked list" must remember two things
  - A **value**,
  - **A pointer** to its successor.
- We implement this as a **pair** containing the value and a pointer to the next node.

```
(value . next)
```

- We thus define the accessor functions:

```
(define (value n) (car n))
(define (next n) (cdr n))
```

As for the queue itself, we need to retain two pointers

- To maintain the queue, we must will two pointers: **`head`** and **`tail`**. We will maintain the invariant that **`head`** *points to the front of the queue*; **`tail`** *points to the end of the queue*.

- To check emptiness:

```
(define (empty?) (null? head))
```

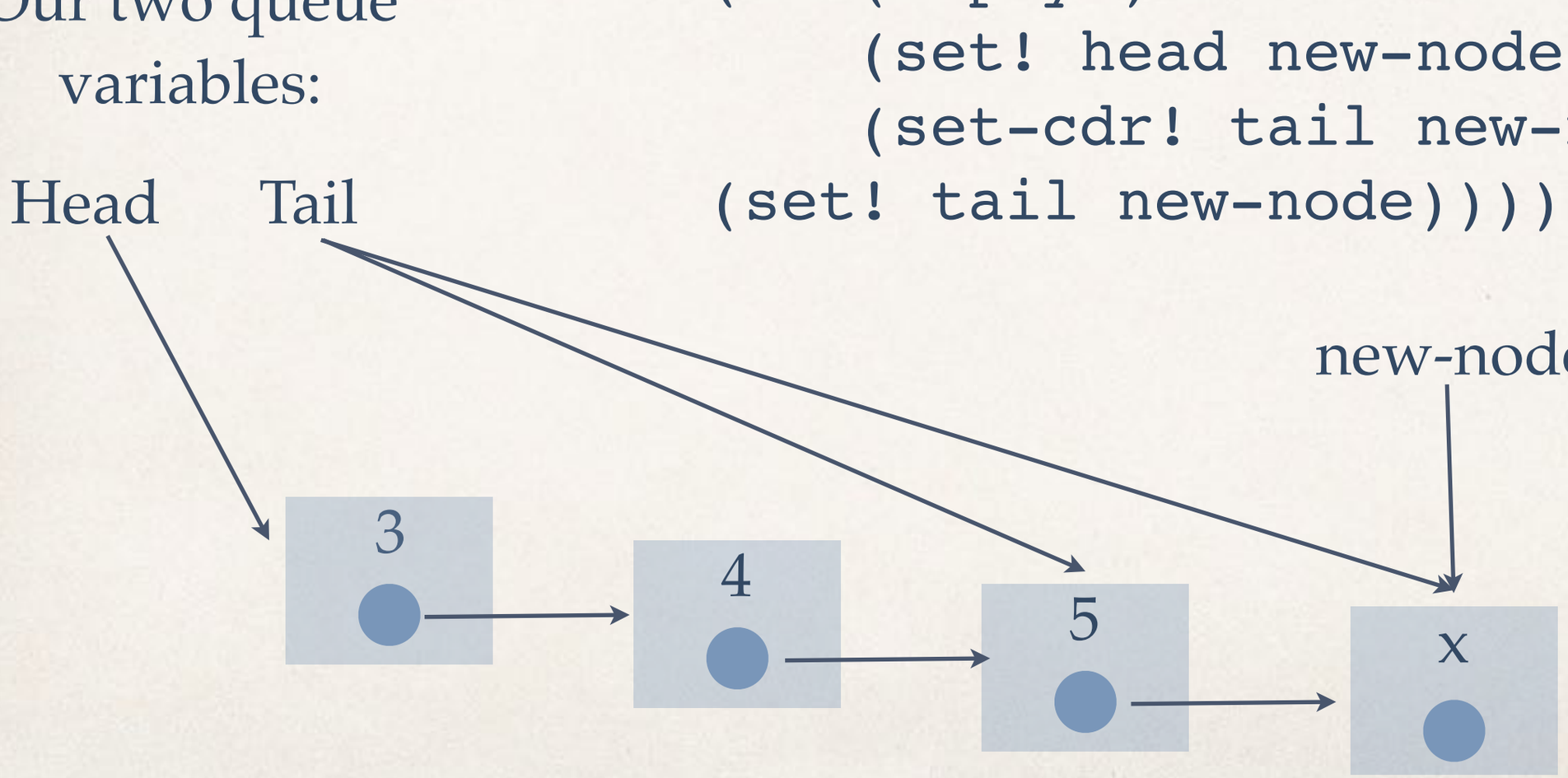
- To return the value at the head:

```
(define (front) (value head))
```

Insertion and deletion via pointer redirection: **Enqueue**

```
(define (enqueue x)
  (let ((new-node (cons x '())))
    (begin
      (if (empty?)
          (set! head new-node)
          (set-cdr! tail new-node))
      (set! tail new-node))))
```

Our two queue variables:



Beware! **Variables** and **Values** have new distinctions…

- This code does not work if we remove the let and replace each `new-node` with its "value."

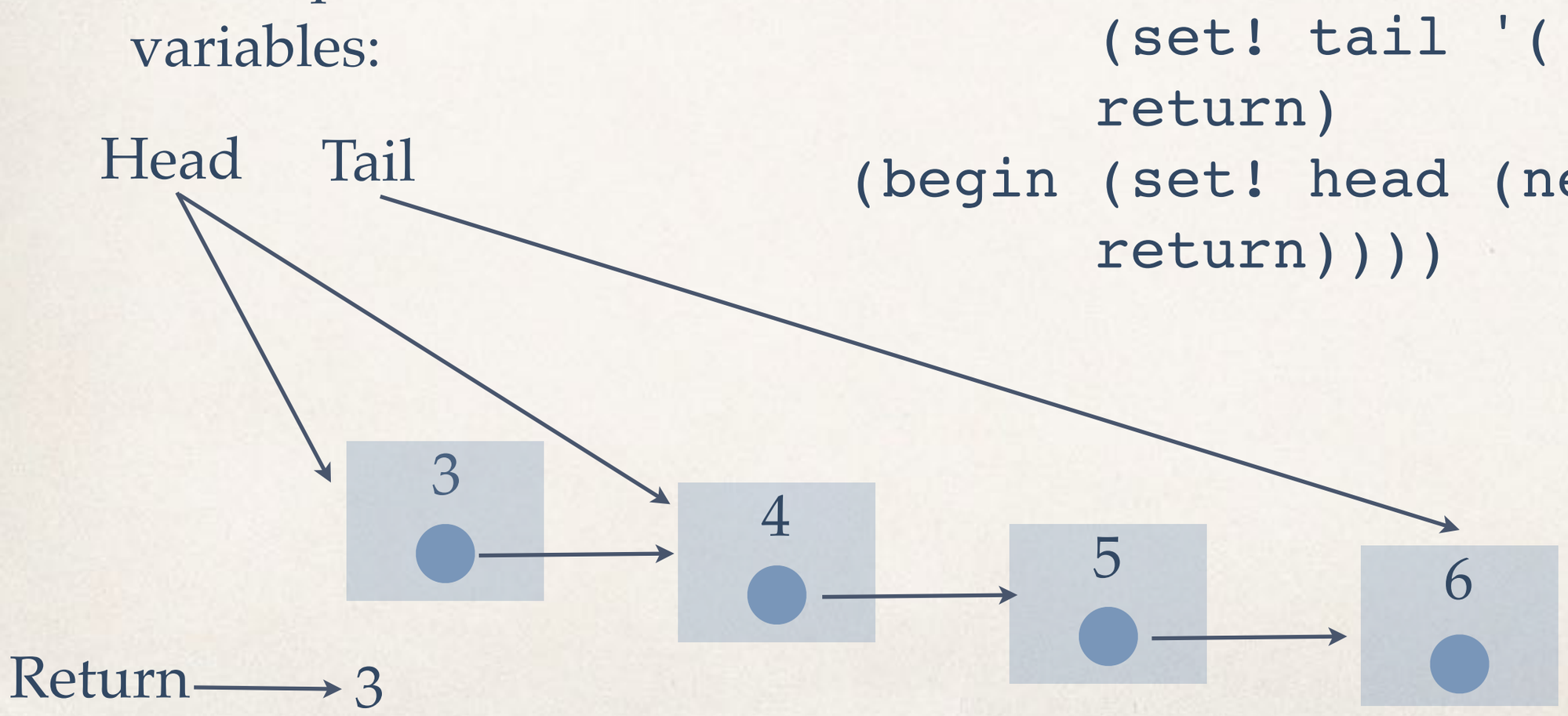
```
(define (enqueue x)
  (let ((new-node (cons x '())))
    (begin
      (if (empty?)
          (set! head new-node)
          (set-cdr! tail new-node))
      (set! tail new-node))))
```

Important point: `cons` builds a new pair in memory. Now tail points to a different "new" pair (not to the last element of the chain)!

Insertion and deletion via pointer redirection: **Dequeue**

```
(define (dequeue)
  (let ((return (value head)))
    (if (eq? head tail)
        (begin (set! head '())
               (set! tail '())
               return)
        (begin (set! head (next head))
               return))))
```

Our two queue variables:



To protect the internals, we wrap these into an object

```
(define (make-queue)
  (let ((head '())
        (tail '()))
    (define (value n) (car n))
    (define (next n)  (cdr n))
    (define (empty?) (null? head))
    (define (front) (value head))
    (define (enqueue x)
      (let ((new-node (cons x '())))
        (begin
          (if (empty?)
              (set! head new-node)
              (set-cdr! tail new-node))
          (set! tail new-node))))
(define (dequeue)
  (let ((return (value head)))
    (if (eq? head tail)
        (begin (set! head '())
               (set! tail '())
               return)
        (begin (set! head (next head))
               return))))
(define (dispatcher method)
  (cond ((eq? method 'empty) empty?)
        ((eq? method 'enqueue) enqueue)
        ((eq? method 'dequeue) dequeue)
        ((eq? method 'front) front)))
  dispatcher))
```

Private data

Private functions

Methods

The dispatcher, returned

The queue, in action

```
> (define q (make-queue))
> ((q 'enqueue) 5)
> ((q 'enqueue) 6)
> ((q 'enqueue) 7)
> ((q 'dequeue))
5
> ((q 'enqueue) 8)
> ((q 'dequeue))
6
> ((q 'dequeue))
7
> ((q 'dequeue))
8
> ((q 'empty))
#t
```

Destructive tree maintenance

- Maintaining a binary search tree. As with the trees we considered earlier in the semester, each node maintains: a **value**, and **two pointers**. We use the convention:

```
(value . (left . right))
```

- Then we introduce the accessor functions:

```
(define (value n) (car n))
(define (pointers n) (cdr n))
(define (right n) (cdr (pointers n)))
(define (left n) (car (pointers n)))
```

The code for testing membership is unchanged...

- To test membership (the tree is in the global variable `bst`):

```
(define (element? x)
  (define (search n)
    (cond ((null? n) #f)
          ((eq? x (value n)) #t)
          ((< x (value n)) (search (left n)))
          ((> x (value n)) (search (right n)))))
  (search bst))
```

Why the embedded function?

Insertion will require that we manufacture a new node

- Making a new node.

```
(define (make-tree value left right)
  (cons value (cons left right)))
```

- When we insert, we will create a new node with a `let` statement, and stitch it into the tree.

Insertion requires some new considerations

```
(define (insert x)
  (let ((new-node (make-tree x '() '())))
    (define (insert-internal n)
      (cond ((>= x (value n))
             (if (null? (right n))
                 (set-cdr! (pointers n) new-node)
                 (insert-internal (right n))))
            ((< x (value n))
             (if (null? (left n))
                 (set-car! (pointers n) new-node)
                 (insert-internal (left n))))))
    (if (null? bst)
        (set! bst new-node)
        (insert-internal bst))))
```

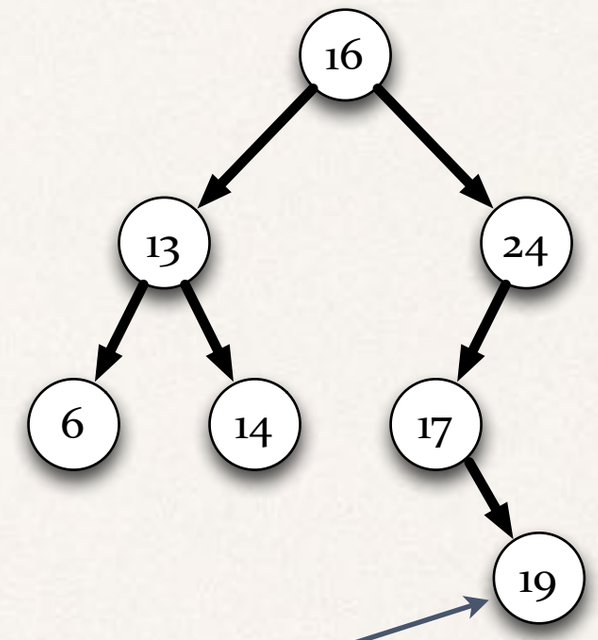
A new node is created

...to an internal node.

The node is stitched to the root or...

Insertion in a picture...

- make-tree constructs new node (assigned to `new-node`)
- `insert-internal` traverses `bst` to determine desired position for new node.
- `new-node` is stitched in with destructive assignment.



Bundling this into an object protects the internals

```
(define (make-set)
  (let ((bst '()))
    (define (value n) (car n))
    (define (pointers n) (cdr n))
    (define (right n) (cdr (pointers n)))
    (define (left n) (car (pointers n)))
    (define (make-tree value left right)
      (cons value (cons left right)))
    (define (empty?) (null? bst))
    (define (element? x)
      (define (search n)
        (cond ((null? n) #f)
              ((eq? x (value n)) #t)
              ((< x (value n)) (search (left n)))
              ((> x (value n)) (search (right n)))))
      (search bst))
(define (insert x)
  (let ((new-node (make-tree x '() '())))
    (define (insert-internal n)
      (cond ((>= x (value n))
             (if (null? (right n))
                 (set-cdr! (pointers n) new-node)
                 (insert-internal (right n))))
            ((< x (value n))
             (if (null? (left n))
                 (set-car! (pointers n) new-node)
                 (insert-internal (left n))))))
    (if (null? bst)
        (set! bst new-node)
        (insert-internal bst))))
```

The private data and functions

The methods

```
(define (list-elements)
  (define (extract-sorted n)
    (if (null? n)
        '()
        (append (extract-sorted (left n))
                (list (value n))
                (extract-sorted (right n)))))
  (extract-sorted bst))
(define (dispatcher method)
  (cond ((eq? method 'insert) insert)
        ((eq? method 'element?) element?)
        ((eq? method 'list-elements) list-elements)
        ((eq? method 'empty?) empty?)))
dispatcher))
```

The dispatcher, passed back

Streams

- A stream is an abstract data type that supports: **empty?**, **head**, and **rest**. You may imagine a stream as representing a sequence of elements so that:
  - **empty?** determines if the sequence is empty,
  - **head** returns the next element of the stream (leaving it at the front of the stream),
  - **rest** returns the remainder of the stream (after removing the first element).
- HEY! **null?**, **car**, and **cdr** do exactly this with a SCHEME list!

Note that streams are "functional"

Indeed, SCHEME lists implement streams

- What's the point making this an abstract data type? Are there other implementations that might be valuable?

- Yes! Recall our "objectified" version of the `hailstone` sequence: this yields a infinite "stream" of numbers; *you can't do that with a list*. Can we perfect it, so that it really adheres to the stream notion?

```
(define (hailstone x)
  (define (next)
    (if (eq? (modulo x 2) 0)
        (begin (set! x (/ x 2)) x)
        (begin (set! x (+ (* 3 x) 1)) x)))
  next)
```

Take 1: Well, we can adapt this so that it provides head & advance

- We can adapt to return the `head` and (destructively) `advance`, but this doesn't yield the stream semantics: in a stream, `rest` should return a new stream object!

```
(define (hailstone x)
  (define (advance)
    (if (eq? (modulo x 2) 0)
        (set! x (/ x 2))
        (set! x (+ (* 3 x) 1))))
  (define (head) x)
  (define (dispatcher method)
    (cond ((eq? method 'advance) advance)
          ((eq? method 'head) head)))
  dispatcher)
```

Take 2: A functional object, that returns objects

`rest` returns a new hailstone object!

```
(define (hailstone x)
  (define (rest)
    (if (eq? (modulo x 2) 0)
        (hailstone (/ x 2))
        (hailstone (+ (* 3 x) 1))))
(define (head) x)
(define (dispatcher method)
  (cond ((eq? method 'rest) rest)
        ((eq? method 'head) head)))
dispatcher)
```

This stream object, in action

```
(define h-stream (hailstone 100))
((h-stream 'head))
> 100
(define rest-of-stream ((h-stream 'rest)))
((rest-of-stream 'head))
> 50
(define and-more-of-it ((rest-of-stream 'rest)))
((and-more-of-it 'head))
> 25
```

Behaves like an infinite list!

Streams by explicit delayed evaluation

- We can strip away some of the complexity here, and recapture this idea in a more abbreviated fashion.

- The idea: We will implement a stream as a *pair*,
  - The `car` of the pair is the head of the stream.
  - The `cdr` of the pair is a function which, when called, returns the pair representing the rest of the stream.

- Then...

```
(define (head s) (car s))
(define (rest s) ((cdr s)))
```

Why the extra function call?

The stream of naturals 0, 1, 2, 3, ...

- We can naturally define the stream of integers starting at k:

```
(define (integer-stream x)
  (cons x
        (lambda () (integer-stream (+ 1 x)))))
```

- Then:

Acts like an infinite list of numbers

```
(head (integer-stream 0))
> 0
(head (rest (integer-stream 0)))
> 1
(head (rest (rest (integer-stream 0))))
> 2
```

Why does this need to be so complicated?

- What would happen if we just defined

```
(define (integer-stream x)
  (cons x
        (integer-stream (+ 1 x))))
```

- Then the stream constructor would never terminate. *The lambda is a way to **delay** evaluation of the rest of the stream until we need it.*

- To remind us why the lambda is there, we may define

```
(define (delay something)
  (lambda () something))
```

`Delay` puts off evaluation until...it is `forced`.

- Then

```
> (define (delay something)
    (lambda () something))
> (delay 1)
#<procedure:...ream-objects.rkt:47:26>
> ((delay 1))
1
```

- If we define `(define (force c) c)`, then

```
> (delay 1)
#<procedure:...ream-objects.rkt:47:26>
> (force (delay 1))
1
```

Remember the convention for a stream…

- In our convention, a stream is a *pair*.
- The `car` is the first element of the stream;
- The `cdr` is a *function* which, when evaluated with no arguments, returns the rest of the stream (a new pair).

```
(first-element . (lambda () …))
```

The stream of naturals from k

- With this new terminology:

```
(define (head s) (car s)
(define (rest s) (force (cdr s)))
  (define (integer-stream x)
    (cons x (delay (integer-stream (+ 1 x))))))
```

- Then:

```
(head (integer-stream 21))
> 21
(head (rest (integer-stream 21)))
> 22
(head (rest (rest (integer-stream 21))))
> 23
```

But that is not all...

- Streams have another important function: **efficiency**. Recall the problem of generating a list of the primes up to k. One intuitive may to mentally model this problem:
  - generate a list of the numbers from 2 through k.
  - "filter" them, expunging the composites and keeping the primes.
- This is straightforward:

Integer lists and filtering

- Lists of integers, from a to b:

```
> (define (integers a b)
    (if (< b a) '()
        (cons a (integers (+ a 1) b))))
> (integers 3 6)
(3 4 5 6)
```

- Filtering:

```
> (define (filter f elements)
    (cond ((null? elements) '())
          ((f (car elements))
              (cons (car elements)
              (filter f (cdr elements))))
          (else (filter f (cdr elements)))))
> (filter odd? (integers 3 6))
(3 5)
```

Then, to generate the primes...

- If `prime?` tests for primality, we can obtain the list of the first k primes by this natural process:

```
(define (primes-upto k)
  (filter prime? (integers 2 k)))
```

- Problems?
  - In order to even work with the car of this list, we have to first generate the whole thing (both the list of integers and, then, the filtered list).
  - If, e.g., it turned out that we only needed the first few of them, we would have wasted all the work of generating the rest.

A stream of primes on demand

- We can use the stream infrastructure to yield a natural stream of primes:

```
(define (primes-from k)
  (if (prime? k)
      (cons k (delay (primes-from (+ k 1))))
      (primes-from (+ k 1))))
```

A `delay`: we've generated what we needed

No `delay` here: we need to keep working

- Then…

```
> (head (primes-from 4))
5
> (head (rest (rest (primes-from 11))))
17
```

Transforming a stream with map

- Recall our notion of map, which applies a function to each element of a list. We develop an analogue for streams. (As above, we maintain a stream as a pair containing the head, and a delayed rest; the empty stream is denoted `'()`)

```
(define (head s) (car s))
(define (rest s) (force (cdr s)))

(define (s-map f s)
  (if (null? s)
      '()
      (cons (f (car s))
            (delay (s-map f (rest s))))))
```

Compare with map for a list!

Mapping the integers to…the squares

- This gives an extremely pleasing notion of implicitly acting on an *infinite* stream:

```
(define (square x) (* x x))
(define (int-stream k)
  (cons k (delay (int-stream (+ 1 k)))))

(define (square-stream k)
  (s-map square (int-stream k)))
```

- For example:

```
> (square-stream 100)
(10000 . #<promise>)
> (rest (square-stream 100))
(10201 . #<promise>)
> (rest (rest (square-stream 100)))
(10404 . #<promise>)
```

And now...we must have: The Fibonacci stream

- This gives a natural way to capture the Fibonacci numbers as an infinite stream:

```
(define (fib-stream current next)
  (cons current
        (delay (fib-stream next (+ next current)))))
(define fibs (fib-stream 0 1))
> fibs
(0 . #<promise>)
> (head (rest (rest (rest fibs))))
2
> (head (rest (rest (rest (rest (rest fibs))))))
3
> (head (rest (rest (rest (rest (rest (rest fibs)))))))
5
```

Where is the information required to build the rest of the stream?

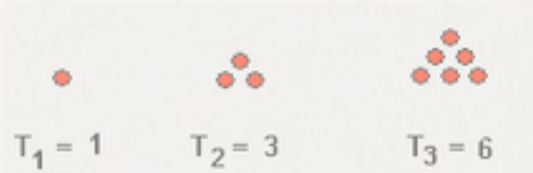
```
(define (fib-stream current next)
  (cons current
        (delay (fib-stream next (+ next current)))))
(define fibs (fib-stream 0 1))
```

Note that the `delay` (equivalent to the `lambda` below) refers to variables in its definition environment. This environment carries the information forward

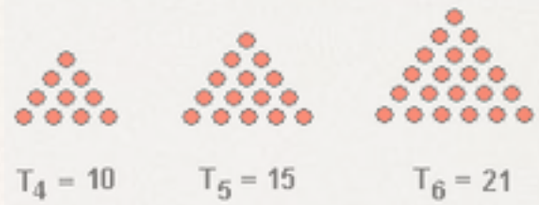
```
(define (fib-stream current next)
  (cons current
        (lambda () (fib-stream next (+ next current)))))
(define fibs (fib-stream 0 1))
```

The Triangular numbers

- Numbers of the form 1, 1+2, 1+2+3, ... are called triangular. Why?



- We can construct the stream of triangular numbers from the stream of integers. In fact, we define a generic `triangulate` that operates on numeric streams.



Triangulate operates on a stream

```
(define (triangulate stream)
  (if (null? stream)
      '()
      (cons (car stream)
            (delay (s-map (lambda (x) (+ x (car stream)))
                          (triangulate (rest stream)))))))

(define tris (triangulate (integers 1)))
```

The Sieve of Eratosthenes

- A classical method for producing the primes.
  - cross off all multiples of 2...
  - cross of all multiples of 3...
  - ...
- Suffices to cross off prime multiples

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100
101	102	103	104	105	106	107	108	109	110
111	112	113	114	115	116	117	118	119	120

Prime numbers

A stream sieve function

- Removes all multiples of k from a stream of numbers

```
(define (sieve k numbers)
  (cond ((null? numbers) '())
        ((divides? k (car numbers))
         (sieve k (rest numbers)))
        (else (cons
                (car numbers)
                (delay (sieve k (rest numbers)))))))
```

- Thus `(sieve 2 (integers 0))` yields the odds!

Implementing the prime stream via the sieve

- Then, the sieve of Eratosthenes:

```
(define (prime-stream sieved)
  (cons (car sieved)
        (delay (prime-stream
                (sieve (car sieved)
                       (rest sieved))))))
(define primes (prime-stream (integers 2)))
```

- How does this work?

1. Start with the stream of integers.
2. Peel off the first prime in the stream.
3. Sieve multiples of this from the rest of the stream.
4. Pass this stream along to the rest, where this will be repeated.

The prime stream, in action

- Then, in action:

```
> primes
(2 . #<promise>)
> (rest primes)
(3 . #<promise>)
> (rest (rest primes))
(5 . #<promise>)
> (rest (rest (rest primes)))
(7 . #<promise>)
```

- What stream is being passed from `rest` to `rest`?

Vectors

- The data structures we have built, so far, are assembled by pointer manipulation on the basic pair structure.

- Scheme also provides a method for maintaining a "k-slotted" object: the *vector*.

- A vector has a fixed length that is determined when it is constructed. The constructor takes a single numeric argument—the number of slots:

```
(make-vector k)          (vector 'a 'b 'c)
```

- Once a vector is created, its *arity* (number of slots) never changes.

Assigning vector slots

- Assigning a vector "slot":

```
(vector-set! v k value)
```

- Retrieving a vector "slot":

```
(vector-ref v k)
```

- Slots are numbered:
  0 through k-1

```
> (define v (make-vector 4))
> (vector-set! v 0 'a)
> (vector-set! v 1 'b)
> (vector-set! v 2 'c)
> (vector-set! v 3 'd)
> v
#(a b c d)
> (vector-ref v 3)
d
```

Vectors vs. lists

- Vectors have a fixed length, determined at definition time. However, SCHEME can retrieve an arbitrary element of a vector in a "single time step" (in particular, it does not depend on which element it retrieves).

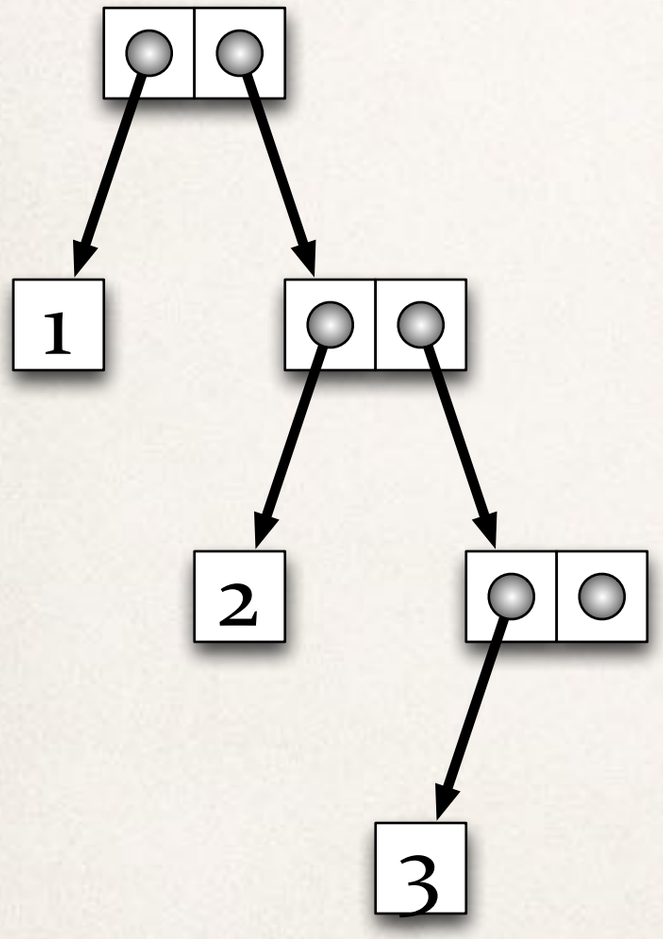
- Lists can grow and shrink during a computation, but retrieving the *k*th element of a list requires stepping through the list.

- There is a significant difference in implementation. These are laid out very differently in memory.

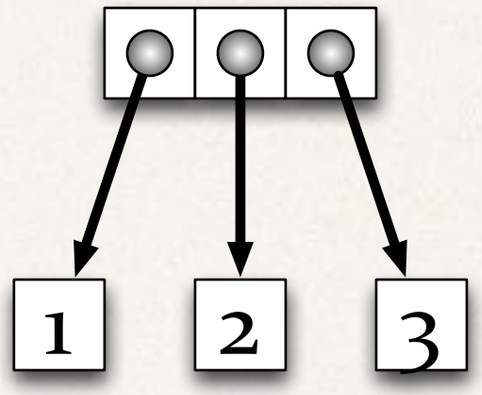
- Naming conventions: Many other programming languages call these *arrays*.

Representation

List of 3 values



Vector of 3 values



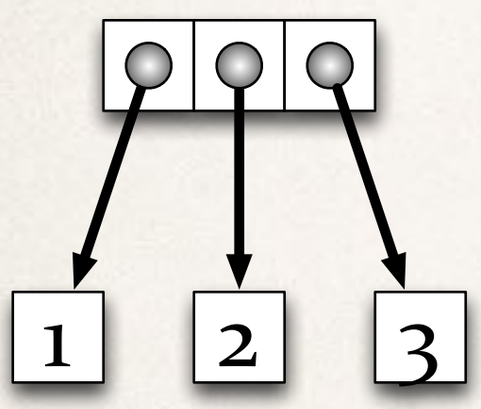
Each entry has the same size

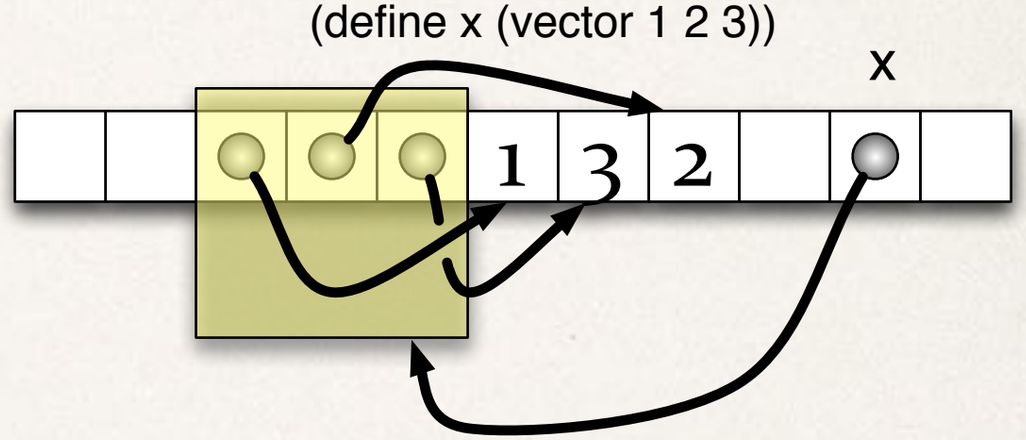
Therefore…

Easy to find k^{th} entry!

In Memory

(define x (vector 1 2 3))

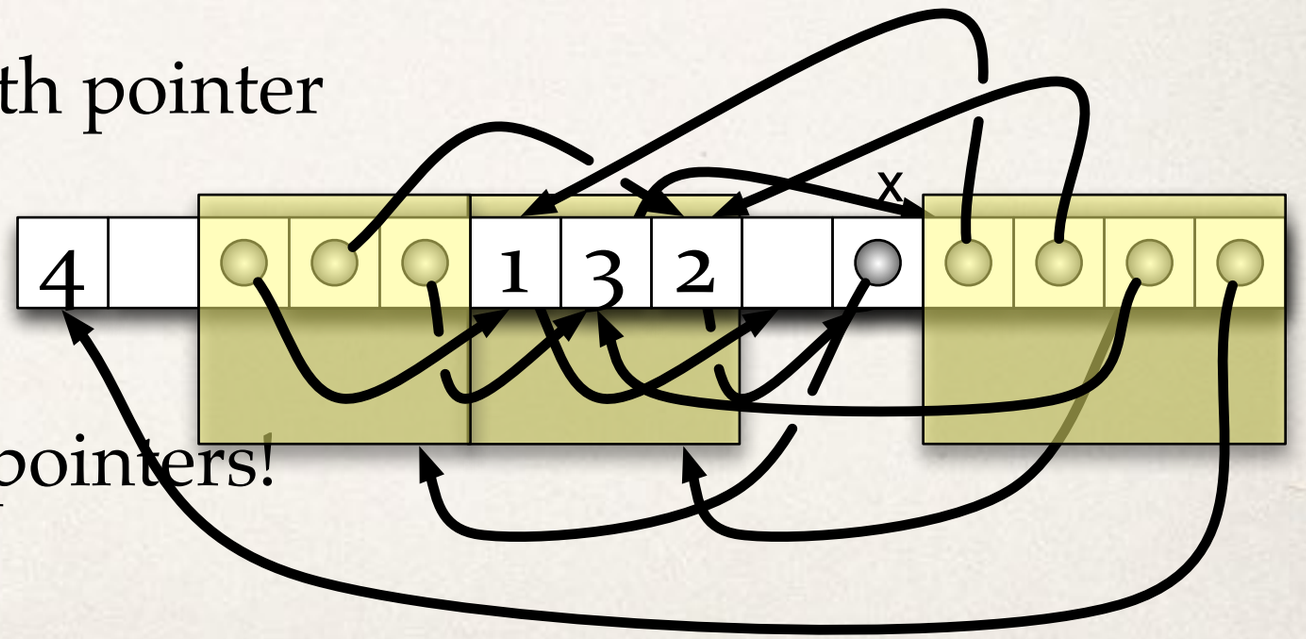




vector-set! is quite similar to set-car! and set-cdr!
It modifies the k^{th} pointer

Vectors *cannot* grow!

- Why?
- Consider adding a fourth entry (e.g., value 4) in a vector of size 3…
- Recall the in-memory representation!
  - You would have to add a fourth pointer
  - But there is no room!
- You would have to copy all the pointers!
  - *Not* built-in in SCHEME.



Adding two vectors...requires an iterator

```
(define (vector-add v1 v2)
  (let ((result (make-vector (vector-length v1))))
    (define (add-iterator k)
      (if (< k (vector-length v1))
          (begin (vector-set! result k
                              (+ (vector-ref v1 k)
                                 (vector-ref v2 k)))
                 (add-iterator (+ k 1)))
          result))
    (add-iterator 0)))
```

Make the new vector

Define the iterator

Run the iterator, starting at zero

One of our goals, as programmers, is to write programs that clearly reflect the structure of the algorithms they carry out. Does this one?

Scheme provides a built-in iterator: do

- The do command. A built-in iterator with syntax:

```
(do ((<var1> <init1> <step1>)
     ...)
    (<test> <return-expression>)
    <command> ...)
```

- **Semantics**: <init$_i$> are evaluated, and variables are bound to these values. Then the iteration phase begins:

- Iteration phase: <test> is evaluated. If true, <return-expression> is evaluated and returned as the value. If false, <command> is evaluated (for side-effects), each <step$_i$> is evaluated, and the iteration is repeated.

Adding vectors with the built-in iterator

- As before, we construct a new vector of the right size.
- Then the iterator "steps" through the vectors, setting `result` for each relevant `i`.

```
(define (vector-add v1 v2)
  (let ((result (make-vector (vector-length v1))))
    (do ((i 0 (+ i 1)))
      ((>= i (vector-length v1)) result)
      (vector-set! result i
                   (+ (vector-ref v1 i)
                      (vector-ref v2 i))))))
```

Parameter passing: *Call by Value* vs. *Call by Reference*

- Once you begin to work with mutable data, you need to answer an important question about your programming language: its *calling convention*:

- **Call by value**: When a function is called, a **new copy** of the argument is sent to the function. (In particular, if the function destructively changes is parameter, no one can tell from the calling environment.)

- **Call by reference**: When a function is called, a **pointer to the argument** is passed to the function. Thus changes propagate back to the calling environment.

Scheme's calling convention:

- All atomic datatypes (integers, Booleans, keywords) are passed **by value**.

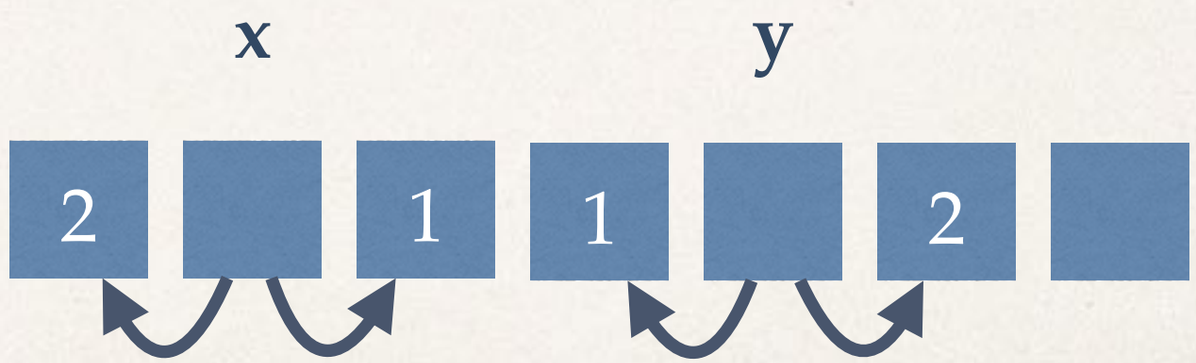
- All structured datatypes (pairs and—by extension—lists, vectors) are passed **by reference**.

- Note that this perfectly mirrors the conventions we have discussed for the rules for variable assignment: assignment to atomic datatypes is done by copy (that is, by value); assignment to structured datatypes is done by reference.

- NOTE: If you only write functional programming, you could never notice the difference.

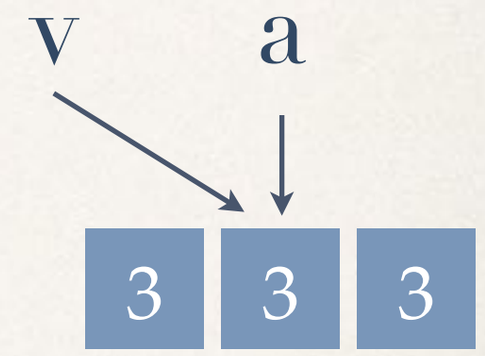
Atomic data: Call by value.

```
> (define x 1)
> x
1
> (define (increment y) (set! y (+ y 1)))
> (increment x)
> x
1
> (set! x (+ x 1))
> x
2
```

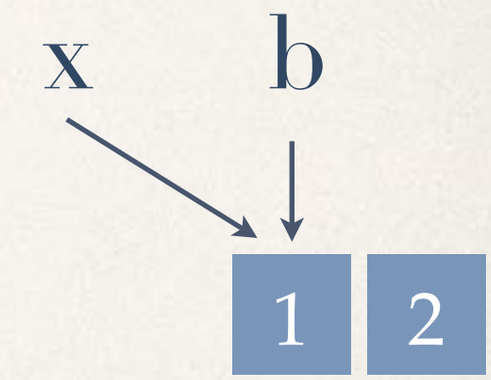


Structured data: Call by reference

```
> (define a (make-vector 3))
> (vector-fill! a 3)
> a
#(3 3 3)
```



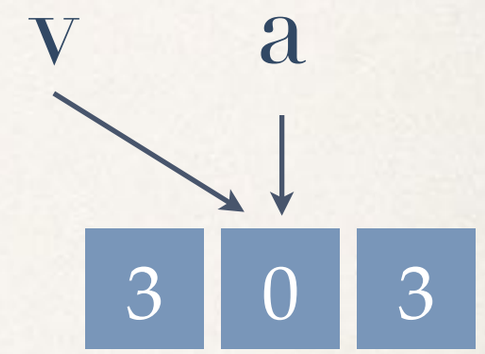

```
> (define (zero-1 v) (vector-set! v 1 0))
> (zero-1 a)
> a
#(3 0 3)
> (define b (cons 1 2))
> (define (disturb x) (set-car! x 3))
> (disturb b)
> b
(3 . 2)
```



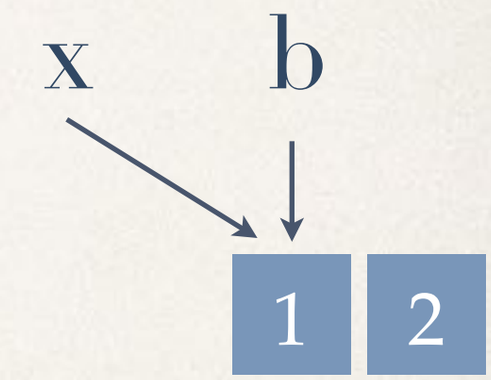
`vector-set!`, `set-car!` changes the structure to which a variable points

Structured data: Call by reference

```
> (define a (make-vector 3))
> (vector-fill! a 3)
> a
#(3 3 3)
```



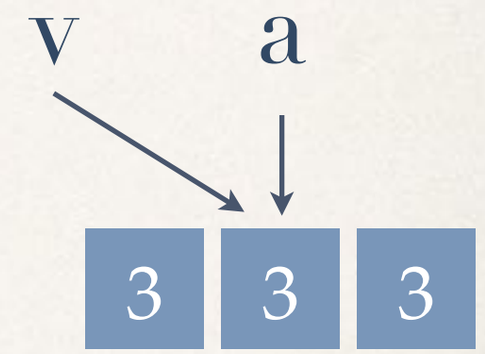

```
> (define (zero-1 v) (vector-set! v 1 0))
> (zero-1 a)
> a
#(3 0 3)
> (define b (cons 1 2))
> (define (disturb x) (set-car! x 3))
> (disturb b)
> b
(3 . 2)
```



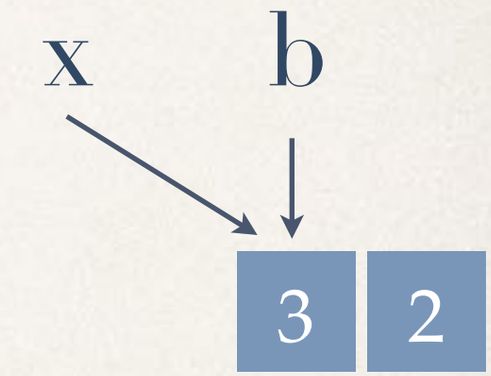
`vector-set!`, `set-car!` changes the structure to which a variable points

Further examples with structured data

```
> (define a (make-vector 3))
> (vector-fill! a 3)
> a
#(3 3 3)
```




```
> (define (zero-1 v) (vector-set! v 1 0))
> (zero-1 a)
> a
#(3 0 3)
> (define b (cons 1 2))
> (define (disturb x) (set-car! x 3))
> (disturb b)
> b
(3 . 2)
```



`vector-set!`, `set-car!` changes the structure to which a variable points

Accumulating a vector sum (functionally)

- A functional version...no destructive assignment.
- How are the intermediate values retained?

```
(define (vector-sum v)
  (define (sum-accumulate i)
    (if (>= i (vector-length v))
        0
        (+ (vector-ref v i)
           (sum-accumulate (+ i 1)))))
  (sum-accumulate 0))

> (vector-sum (vector 1 2 3 4 5))
15
```

Accumulating a vector sum (destructively)

- Using `do`: loops through the array. For each index, it destructively updates result.

All action in a single environment

```
(define (vector-sum-d v)
  (let ((result 0))
    (do ((index 0 (+ index 1)))
      ((>= index (vector-length v)) result)
      (set! result
            (+ result (vector-ref v index))))))
```

```
> (vector-sum-d (vector 1 2 3 4 5))
15
```

Inner Product

- Simple idea
  - Sum of products of identical components.
  - Both vectors have the same length

$$\sum_{i=0}^{n-1} a_i \cdot b_i$$

```
(define (vector-product v1 v2)
 (let ((sp 0))
    (do ((i 0 (+ i 1)))
      ((>= i (vector-length v1)) sp)
      (set! sp (+ sp (* (vector-ref v1 i)
                        (vector-ref v2 i)))))))

> (vector-product (vector 1 2 3) (vector 4 5 6))
32
```

Computing Sum of Squares?

- Easy enough!

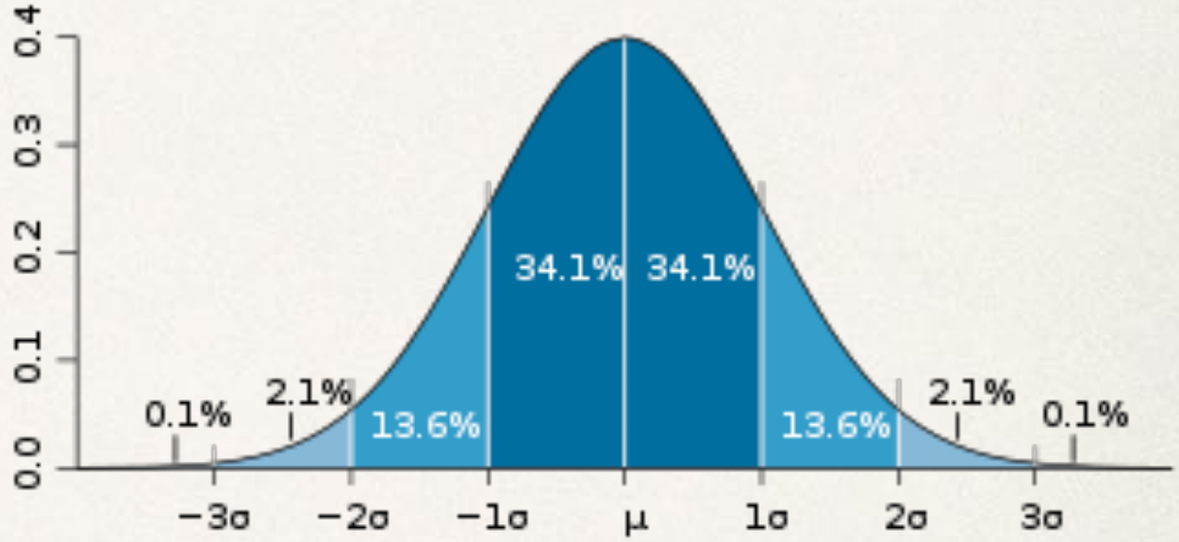
```
(define (vector-product v1 v2)
  (let ((sp 0))
    (do ((i 0 (+ i 1)))
      ((>= i (vector-length v1)) sp)
      (set! sp (+ sp (* (vector-ref v1 i)
                        (vector-ref v2 i)))))))

(define (sum-square v) (vector-product v v))
```

Standard deviation

- Is a measure of the "spread" of a distribution
- For a sample of size n.

$$\sigma = \sqrt{\frac{1}{n-1} \cdot \left(\sum_{i=0}^{n-1} (a_i - \mu)^2 \right)}$$



- With the average defined as:

$$\mu = \frac{\sum_{i=0}^{n-1} a_i}{n}$$

What is a good plan?

The code….

```
(define (vector-average v)
 (/ (vector-sum v) (vector-length v)))
```

```
(define (vector-stdev v)
   (let* ((mu   (vector-average v))
          (muv  (make-vector (vector-length v) (- mu)))
          (terms   (vector-add v muv))
          (ssq     (sum-square terms))
          )
     (sqrt (/ ssq (- (vector-length v) 1)))))
```

autofill with value

Note

- Most of the examples so far….
  - Use vectors in a “read-only” style
- However…
  - We benefit from the fast access to the i^{th} entry

Maintaining Sets in Vectors (a.k.a. Arrays)

- We've invested a lot of time discussing the Set ADT and various implementations.

- Let's consider implementing the SET ADT using a vector.

- Because vectors have fixed size, we will simply *assume our sets never exceed a maximum size*:

  - this simplifies the implementation, but

  - it can lead to unexpected "overflow" errors. **Programmer beware!**

  - There are ways to manage this, which we will briefly discuss.

First implementation: Unordered sequence in a vector

- Initialize a vector with `max` slots.
- Maintain the set in the first `size` slots of the vector:
  - Insert: Place in an empty slot at position `size`.
  - Member?: Search through the first `size` slots.

To insert, add here…

Efficiency (complexity) of these operations?

- Insert is fast. `vector-set!` at a particular location.
- Testing membership is slow. Must search through the entire set.
- Also: Must be careful about overflows. Important to handle the case where `size = max`.
- The implementation…

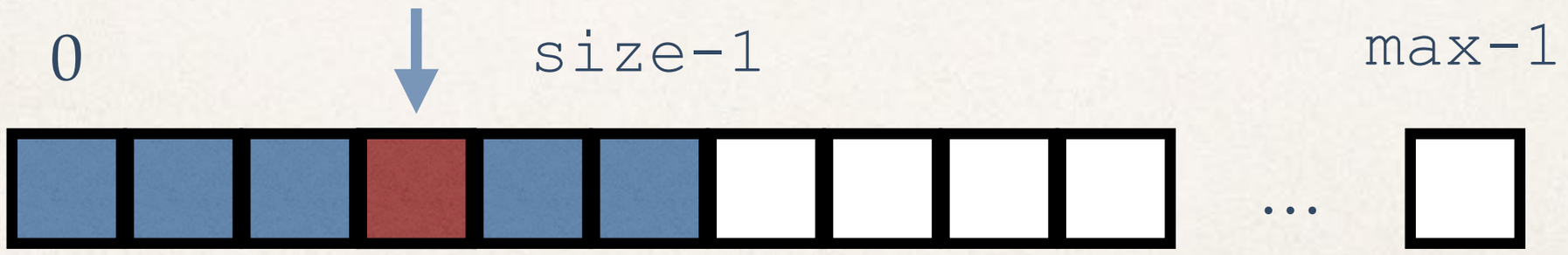
The implementation as a Scheme object

```
(define (set-as-vec max)
  (let ((set (make-vector max))
        (size 0))
    (define (insert x)
      (if (= size max)
          'overflow
          (begin
            (vector-set! set size x)
            (set! size (+ 1 size))
            'success)))
    (define (member x)
      (define (search-from i)
        (and (< i size)
             (or (= x (vector-ref set i))
                 (search-from (+ i 1)))))
      (search-from 0))
    (define (empty) (= size 0))
    (define (dispatcher method)
      (cond ((eq? method 'empty) empty)
            ((eq? method 'insert) insert)
            ((eq? method 'member) member)))
    dispatcher))
```

- `max` is set when object is built.
- the vector and the (set) size are internal variables
- note that `insert` returns either `success` or `overflow`

Alternative implementation: **Sorted** sequence in a vector

- Initialize a vector with `max` slots.
- Maintain the set in the first `size` slots of the vector, **in sorted order**:
- Insert: Determine where to add the element; shift the remaining elements forward one slot.
- Member?: Use **binary search**—very fast!



Maintain the set in the first `size` slots of the vector, **in sorted order**:

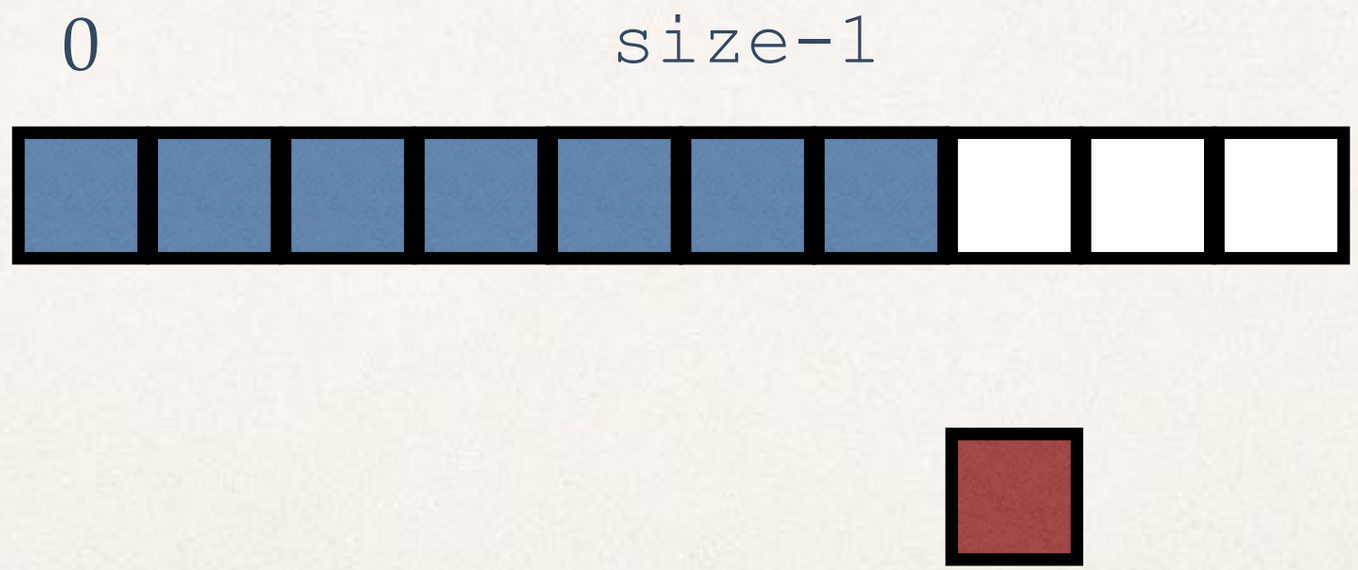
Insert: Determine where to add the element; shift the remaining elements forward one slot.

Member?: Use **binary search**—very fast!

Now, we know exactly where a new element should be placed…

Insertion details…

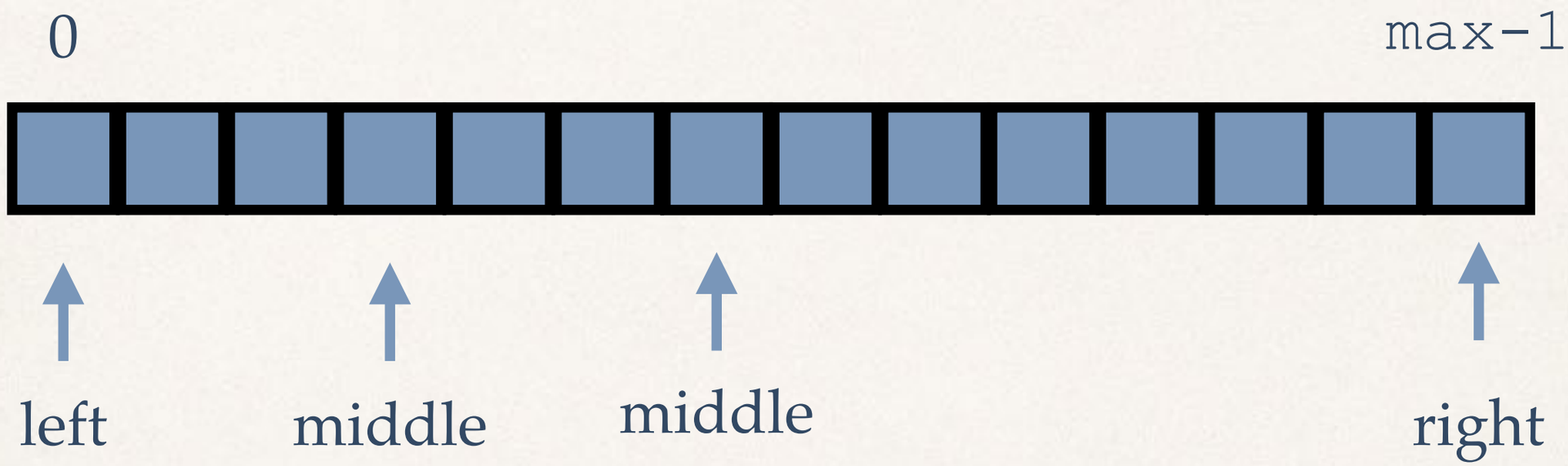
- The simplest implementation searches and shifts at the same time…



Binary search…

- Fast search for an element x in a sorted sequence…if only one element in the sequence, it's easy!
- Otherwise: examine the *middle* element…if $x = middle$, you found it!
  - If $x < middle$, you only need to consider the first half—recurse!
  - If $x > middle$, you only need to consider the second half—recurse!
- Often implemented by maintaining two positions in the vector, *left* and *right*, which determine the **portion of the vector that remains to be searched**.
- During each step the distance between left and right will be reduced by 50%.

A binary search example



x > middle?

left & right point to the limits of the current portion of the vector under consideration.

Efficiency (complexity) of these operations?

- Insert is *slow*. Could require shifting the whole set over.
- Testing membership is *fast*. Binary search only requires $\log_2(size)$ steps, because the active area shrinks by 50% each time.
- Looks like a tradeoff vs. the previous implementation…However, in many circumstances, the *bulk of the set operations are membership queries*. So this is preferable.
- The implementation…

Putting the pieces together…

```
(define (set-as-sorted-vec max)
  (let ((set (make-vector max))
        (size 0))
    (define (member x)
      (define (search left right)
        (and (<= left right)
             (let* ((middle (round (/ (+ left right) 2)))
                    (middle-value (vector-ref set middle)))
               (cond ((= x middle-value) #t)
                     ((< x middle-value) (search left (- middle 1)))
                     ((> x middle-value) (search (+ middle 1) right))))))
      (search 0 (- size 1)))
    (define (insert x)
      (define (shift-and-insert p)
        (if (or (= p 0) (< (vector-ref set (- p 1)) x))
            (vector-set! set p x)
            (begin (vector-set! set p (vector-ref set (- p 1)))
                   (shift-and-insert (- p 1)))))
      (if (< size max)
          (begin (shift-and-insert size)
                 (set! size (+ size 1))
                 'success)
          'overflow))
    (define (empty) (= size 0))
  (define (dispatcher method)
    (cond ((eq? method 'empty) empty)
          ((eq? method 'insert) insert)
          ((eq? method 'member) member)))
  dispatcher))
```

Binary search

Shift & insert

Another look at sorting… with vectors

- We've talked about three sorting algorithms:
  - **Selection sort** (find minimum element, place first, recursively sort the rest of the list), and
  - **Quicksort** (partition around a pivot elements, recursively sort the pieces, and reassemble the full list).
  - **Mergesort** (divide into two halves, recursively sort each half, merge the two sorted lists into one).
- Let's reconsider the sorting challenge with data in a vector.

Selection sort

Find minimum element

Swap with first element

Recursively handle the rest of the array, now we know the smallest element appears first…

Sorting considerations…

- In the "worst case," selection sort might require approximately n^2 operations to sort a sequence of length n. There are faster algorithms (like mergesort, which is approximately $n \log n$).

- However: Selection sort is "**in-place**": you don't need to allocate other vectors to carry out the sorting (it all occurs inside the initial vector). This can be valuable in some practical settings.

- And: Selection sort is also well-behaved when the input vector is **almost sorted**, which can also be useful in some settings.

An implementation…

```
(define (selection-sort numbers)
  (define (swap i j)
    (let ((temp (vector-ref numbers i)))
      (begin (vector-set! numbers i (vector-ref numbers j))
             (vector-set! numbers j temp))))
  (define (index-min i j)
    (if (< (vector-ref numbers i)
           (vector-ref numbers j))
        i j))
  (define (min-range start end)
    (if (= start end) start
        (index-min start (min-range (+ 1 start) end))))
  (define (selection-sort-range start end)
    (if (< start end)
        (begin (swap start (min-range start end))
               (selection-sort-range (+ 1 start) end))
        numbers))
  (selection-sort-range 0
                        (- (vector-length numbers) 1)))
```

Takes vector `numbers` as argument

Code for swap

Given two indices: which refers to a smaller number?

Smallest in a range

Sort

Another famous sorting algorithm: Bubblesort

- …in which large values "bubble up" to the correct positions.
- Roughly: If the vector has length n, repeat the following (n-1) times:
  - Scan the vector, left to right, comparing values in adjacent cells; anytime you discover an out-of-order pair, swap them.
- Bubble sort is also "in-place."

Why does this work?

- It seems clear that after enough "sweeps," the list really ought to be sorted, but it's not exactly obvious why.

- One way to see correctness is that after the first sweep, the largest element has been correctly placed at the end of the vector.

- After the second sweep, the second to largest element is in place.

- Indeed, after k sweeps, the top k elements are correct!

- (By the way, this suggest a minor improvement to the algorithm…)

An implementation of Bubblesort

```
(define (bubblesort numbers)
  (let ((size (vector-length numbers)))
    (define (swap i j)
      (let ((temp (vector-ref numbers i)))
        (begin (vector-set! numbers i
                            (vector-ref numbers j))
               (vector-set! numbers j temp))))
  (define (test-and-swap i)
    (if (> (vector-ref numbers (- i 1))
           (vector-ref numbers i))
      (swap i (- i 1)) '()))
  (do ((sweep 0 (+ sweep 1)))
    ((= sweep size) numbers)
    (do ((position 1 (+ position 1)))
      ((= position size) #t)
      (test-and-swap position)))))
```

Note “nested” loops

Maintaining a heap in a vector?

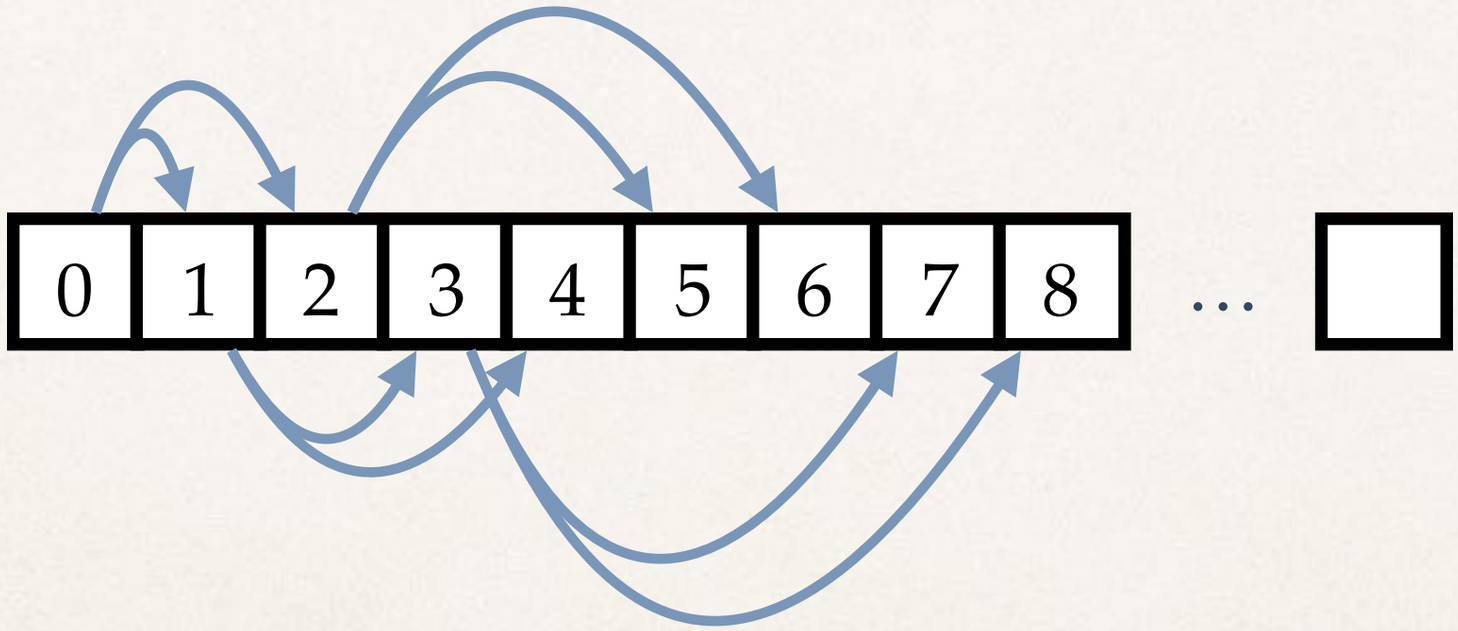
- Heaps seem like an inherently "treelike" construct. It's not entirely obvious how to easily implement a heap and—by extension—Heapsort using a vector.

- However there is a nice convention for this which permits efficient heap operations. In fact, this gives rise to a sorting algorithm that is significantly faster than Selection sort and Bubblesort for large inputs.

- The convention: the heap will live in the prefix of the vector: the "children" of cell t are cells $2t+1$ and $2t+2$.

Maintaining a heap in a vector

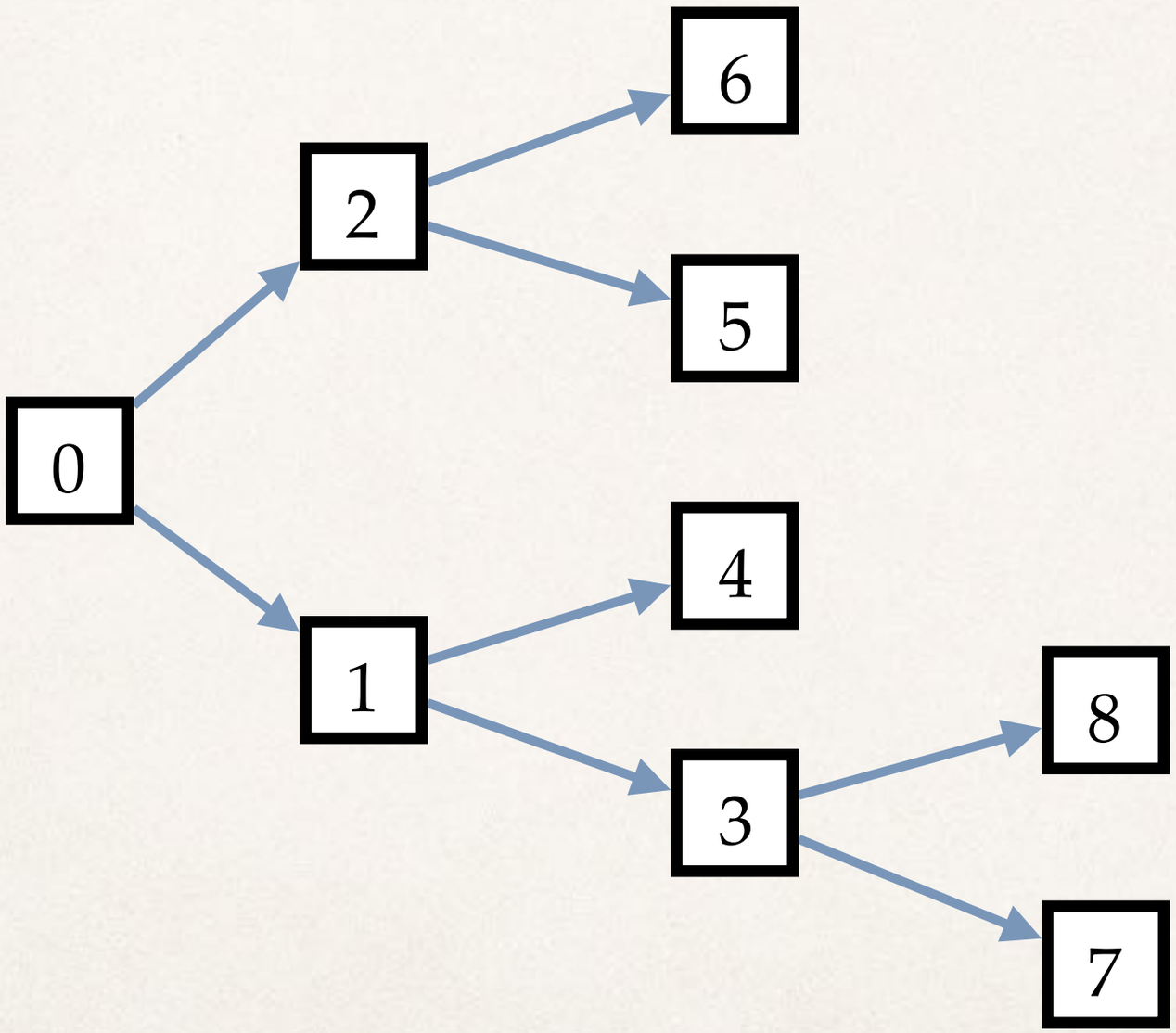
- Every "node" points to two children.
- Every "node" (except 1) has a *unique* parent.



Cell k is the parent of cells $2k+1$ and $2k+2$

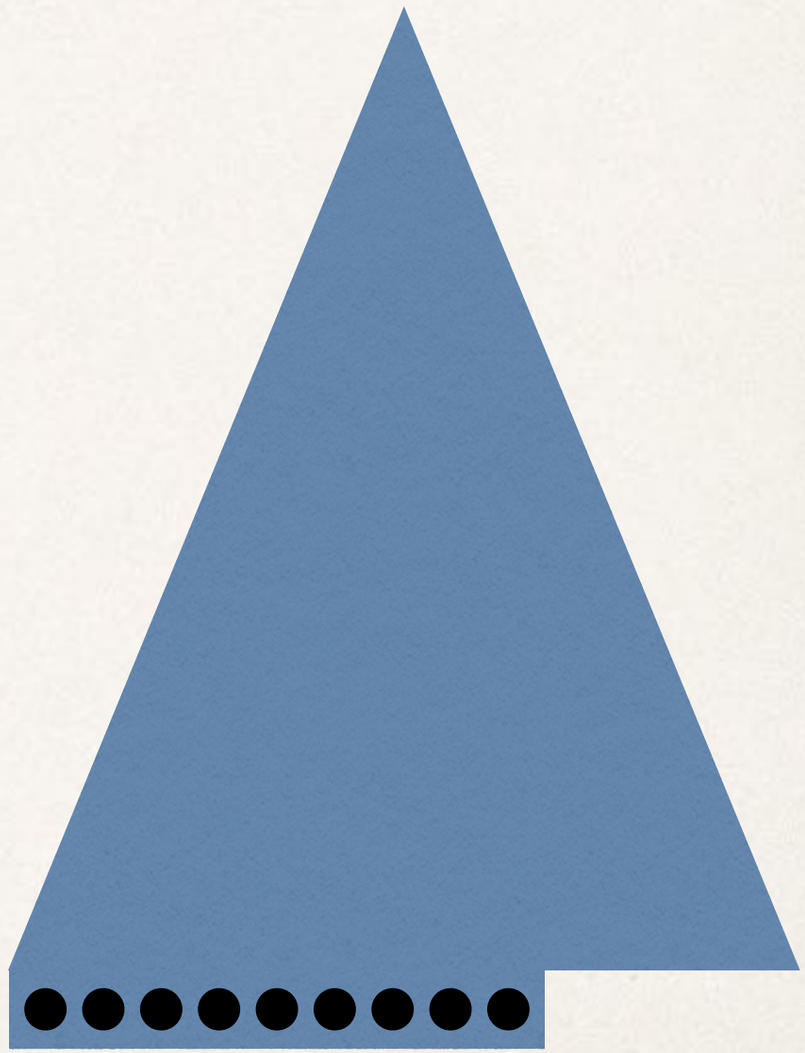
Alternative picture

Alternative picture



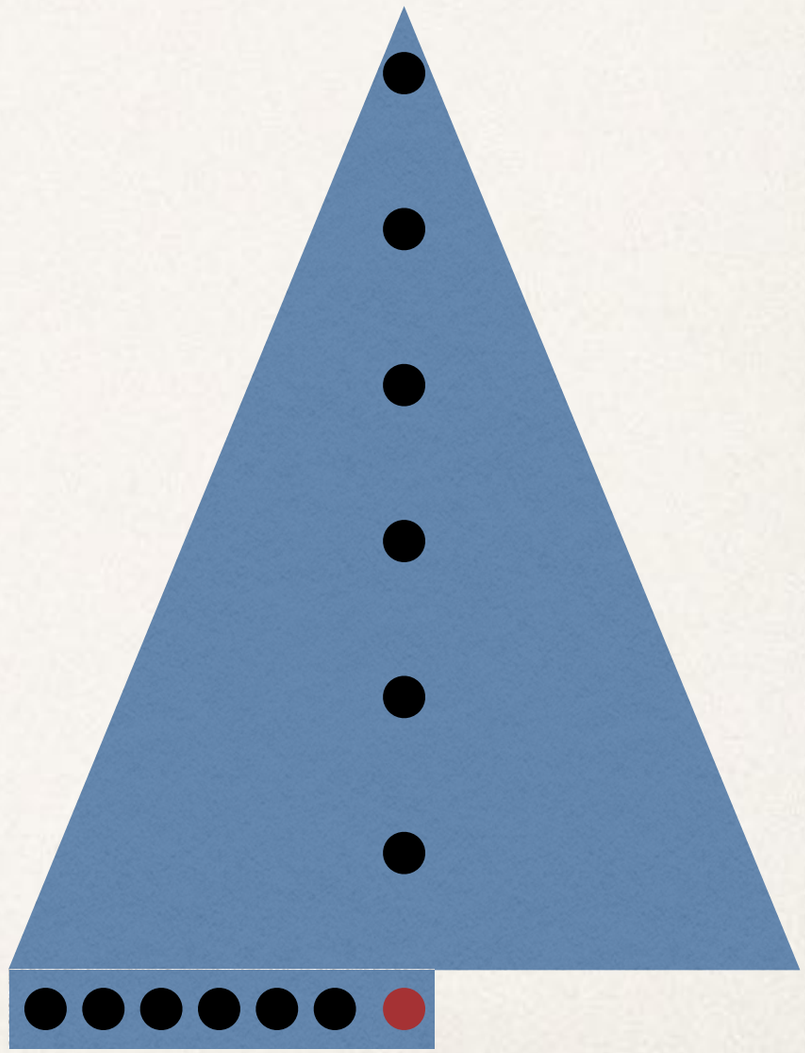
This maintains an implicit, balanced tree

- This implicitly defines a tree.
- If a prefix of the array is filled, the tree is “balanced” and the leaves are filled in “left-to-right” on the bottom level.
- How to maintain a heap that preserves this structure?



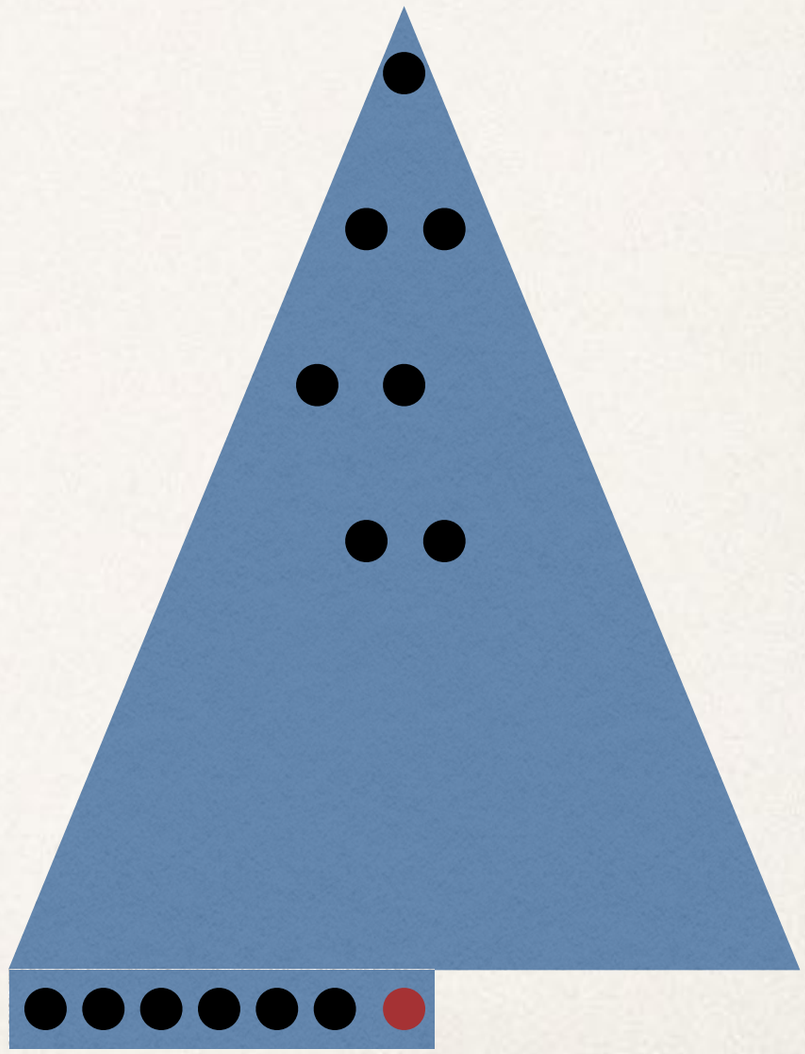
Heap insertion

- New element is added to expand bottom row
- Then, new element is **sifted-up**:
  - Compare with parent, if smaller then swap;
  - Repeat until parent is smaller.
- This results in a heap with the right tree structure



Heap delete-min

- Minimum element removed from root.
- Rightmost child is placed at root.
- Then, root element is **sifted-down**:
  - If smaller child: swap with smallest child;
  - Repeat until smaller than both children.
- This results in a heap with the right tree structure.



The code in Scheme…

- …is a little complicated:
  - Have to be careful about over-/underflows.
  - Have to be careful about whether indexes are “null” or not (that is, whether they lie in the part of the vector associated with the heap).
- The full code:

The full code:

```
(define (heap-as-vector max)
  (let ((heap (make-vector max))
        (size 0))
    (define (read k)     (vector-ref  heap k))
    (define (write! k v) (vector-set! heap k v))
    (define (parent k) (floor (/ (- k 1) 2)))
    (define (left k)  (+ 1 (* 2 k)))
    (define (right k) (+ 2 (* 2 k)))
    (define (non-null k) (< k size))
    (define (swap i j)
      (let ((temp (read i)))
        (begin (write! i (read j))
               (write! j temp))))
    (define (sift-up k)
      (if (or (= k 0)
              (> (read k) (read (parent k))))
          #t
          (begin (swap k (parent k))
                 (sift-up (parent k)))))
    (define (sift-down k)
      (let ((left-smaller (and (non-null (left k))
                               (< (read (left k)) (read k))))
            (right-smaller (and (non-null (right k))
                                (< (read (right k)) (read k)))))
        (cond ((and left-smaller
                    (or (not right-smaller)
                        (<= (read (left k)) (read (right k)))))
               (begin (swap k (left k)) (sift-down (left k))))
              ((and right-smaller
                    (or (not left-smaller)
                        (<= (read (right k)) (read (left k)))))
               (begin (swap k (right k)) (sift-down (right k)))))))
    (define (empty) (= size 0))
    (define (min) (read 0))
    (define (insert x)
      (if (= size max)
          'overflow
          (begin (write! size x)
                 (sift-up size)
                 (set! size (+ 1 size)))))
    (define (remove-min)
      (if (= size 0)
          'underflow
          (begin (write! 0 (read (- size 1)))
                 (set! size (- size 1))
                 (sift-down 0))))
    (define (dispatcher method)
      (cond ((eq? method 'empty) empty)
            ((eq? method 'insert) insert)
            ((eq? method 'min) min)
            ((eq? method 'remove-min) remove-min)))
    dispatcher))
```

Further considerations about sorting in a vector

- In fact…Heapsort can be turned in to an "in-place" algorithm. Expand the active area by one cell at a time, sift-up the new value. (In fact, there is a fancier version that works "from the right.")
- What about other sorting algorithms we have discussed?
  - How to implement Mergesort with vectors? (Can it be in-place?)
  - How to implement Quicksort with vectors? (Can it be in-place?)

Further questions…

- How would you implement a stack using a vector?
- How would you implement a queue using a vector? (This is not entirely straightforward…)
- You want to maintain a set in a vector, but don't want to be constrained by the size of the vector. How, exactly, do you handle things?

Managing a dynamic set in a fixed size vector…

```
(define (set-as-dynamic-vec)
  (let* ((max 1)
         (set (make-vector max))
         (size 0))
    (define (copy v w)
      (do ((t 0 (+ t 1)))
        ((= t (vector-length v)) w)
        (vector-set! w t (vector-ref v t))))
    (define (insert x)
      (if (= size max)
          (let ((newset (make-vector (* 2 max))))
            (begin (copy set newset)
                   (set! set newset)
                   (set! max (* 2 max))
                   (insert x)))
          (begin (vector-set! set size x)
                 (set! size (+ 1 size)))))
```

Start with a vector of size 1

Code to copy

Setup new vector & copy!

```
(define (member x)
  (define (search-from i)
    (and (< i size)
         (or (= x (vector-ref set i))
             (search-from (+ i 1)))))
  (search-from 0))
(define (empty) (= size 0))
(define (dispatcher method)
  (cond ((eq? method 'empty) empty)
        ((eq? method 'insert) insert)
        ((eq? method 'member) member)))
dispatcher))
```

Unchanged

Vector, Matrices and 42.

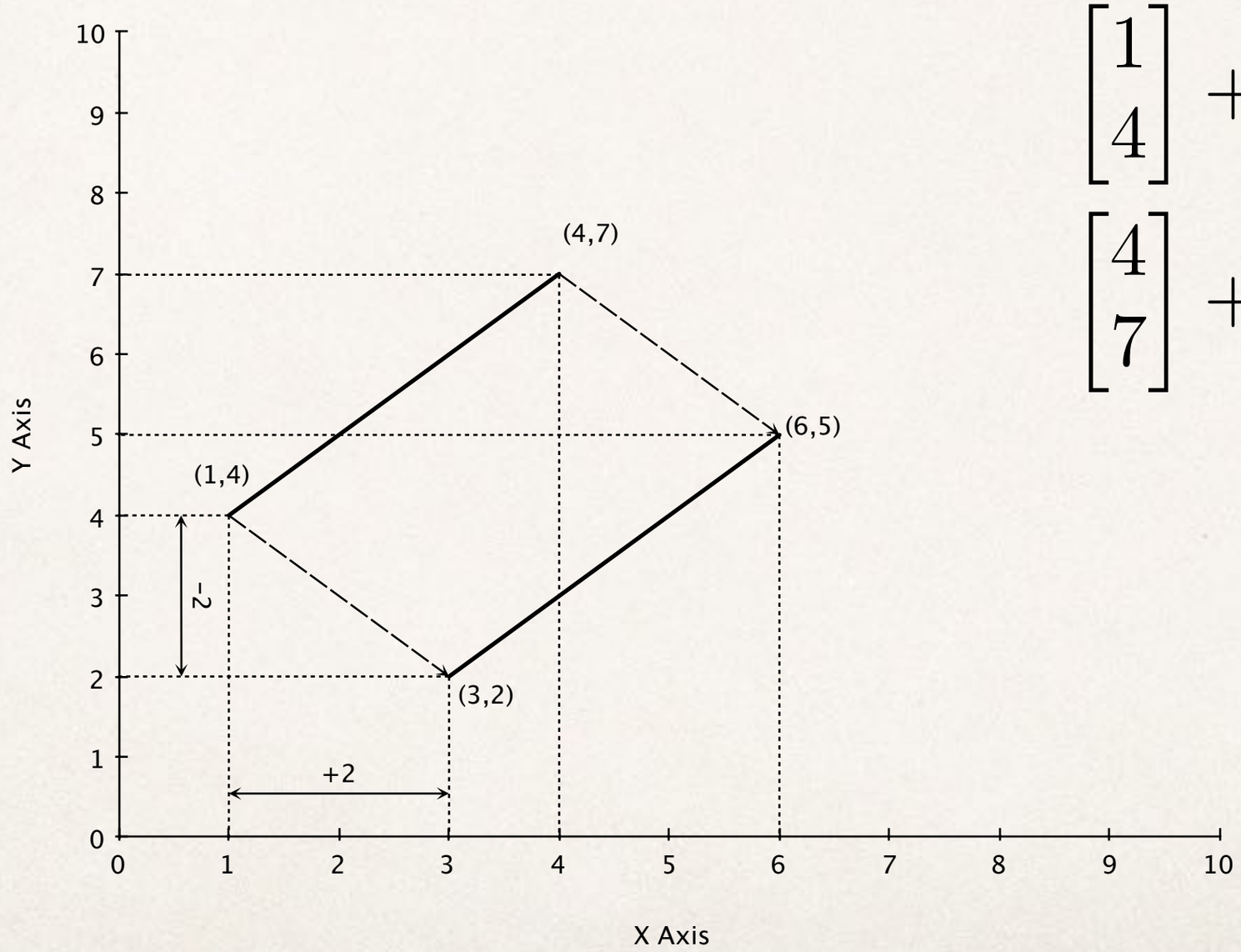
Answer to the Ultimate Question of Life, the Universe, and Everything

- Geometry & Linear Algebra
  (in a handful of Slides!)

Vectors to represent points

Vector of length 2	Point in the plane
Vector of length 3	Point in 3D space
Vector of length 4	Point in space and time?
Longer vectors?	Use your imagination!

Vectors & Translation

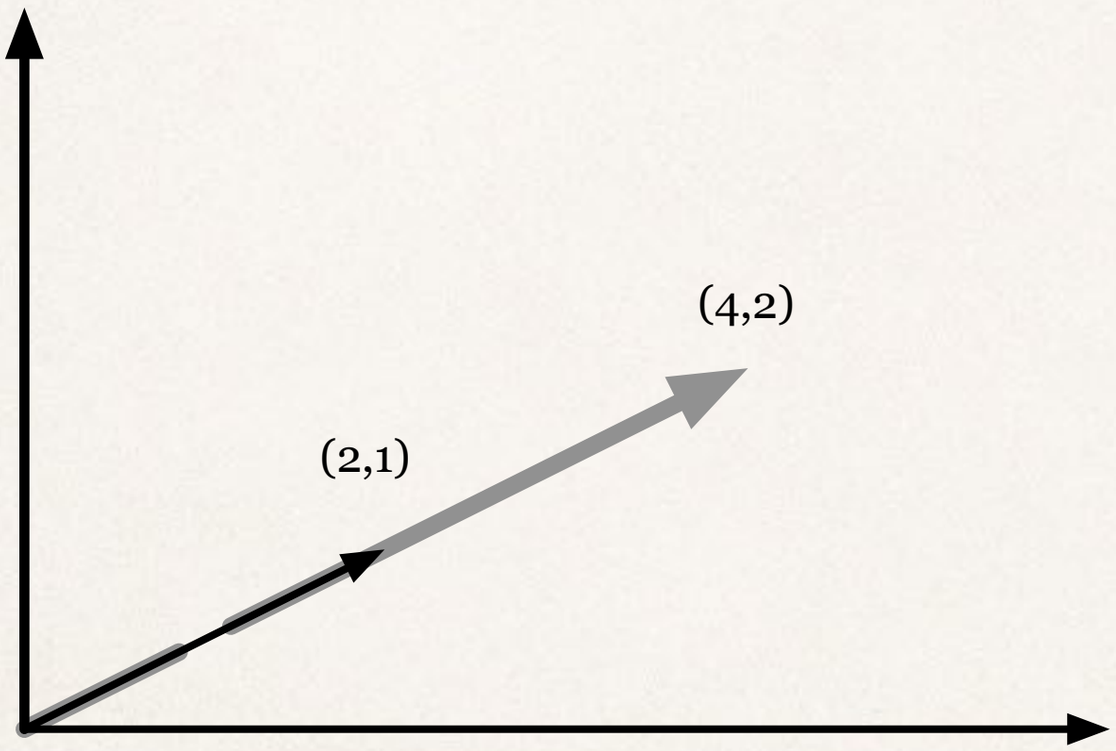


$$\begin{bmatrix}1\\4\end{bmatrix}+\begin{bmatrix}2\\-2\end{bmatrix}=\begin{bmatrix}3\\2\end{bmatrix}$$

$$\begin{bmatrix}4\\7\end{bmatrix}+\begin{bmatrix}2\\-2\end{bmatrix}=\begin{bmatrix}6\\5\end{bmatrix}$$

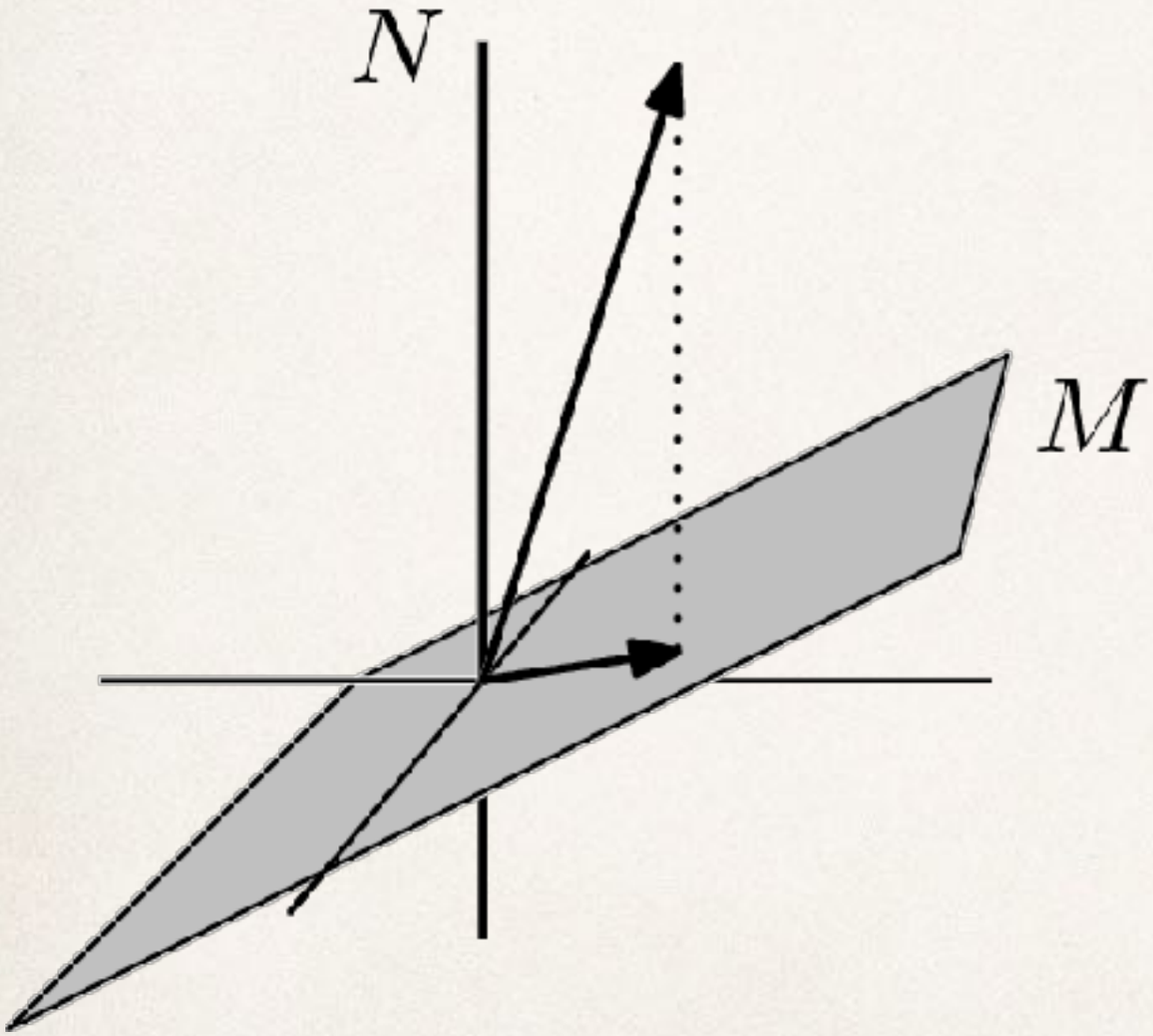
Vector & Scaling

$$2 \cdot \begin{bmatrix} 2 \\ 1 \end{bmatrix} = \begin{bmatrix} 4 \\ 2 \end{bmatrix}$$



Scale each component

Vectors & Projection



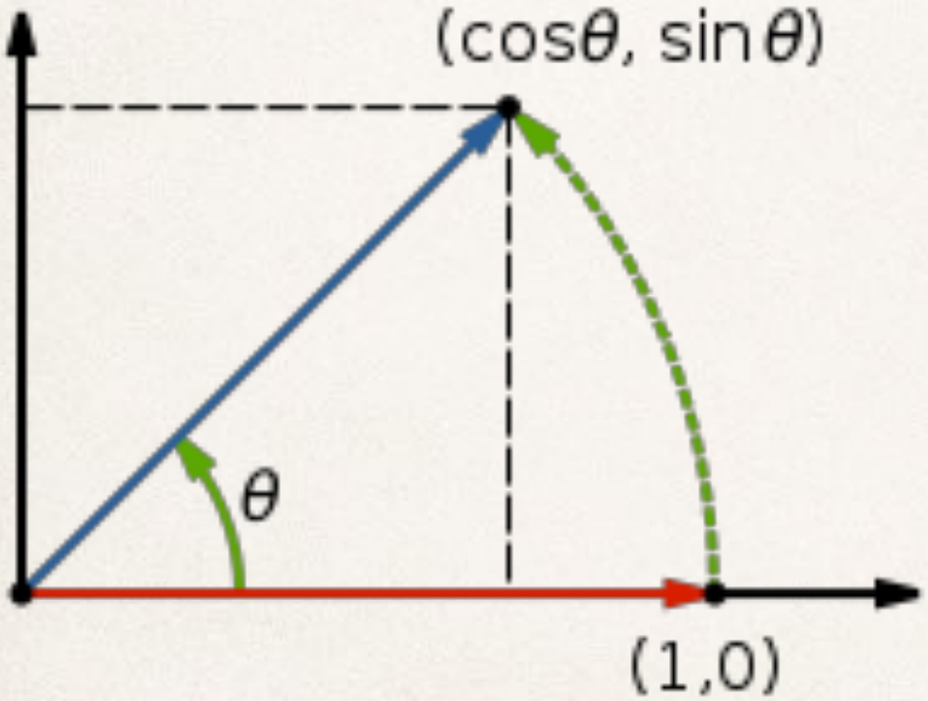
Done by *inner product*

$$\begin{bmatrix} a_1 \\ a_2 \end{bmatrix} \odot \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} = a_1 b_1 + a_2 b_2$$

$$\begin{bmatrix} 1 \\ 2 \end{bmatrix} \odot \begin{bmatrix} 3 \\ 4 \end{bmatrix} = 1 \times 3 + 2 \times 4$$

Vector & Rotation

$$x' = x \cdot \cos(\theta) - y \cdot sin(\theta)$$
$$y' = x \cdot \sin(\theta) + y \cdot cos(\theta)$$



Looks like inner-products!

$$x' = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix}$$

$$y' = \begin{bmatrix} \sin(\theta) & \cos(\theta) \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix}$$

That's a matrix!

Matrix Use ?

- A very useful application: Geometry
- Represent *linear transformations*
  - Translation
  - Scaling
  - Shearing
  - Rotation
- Represent their combination as well!

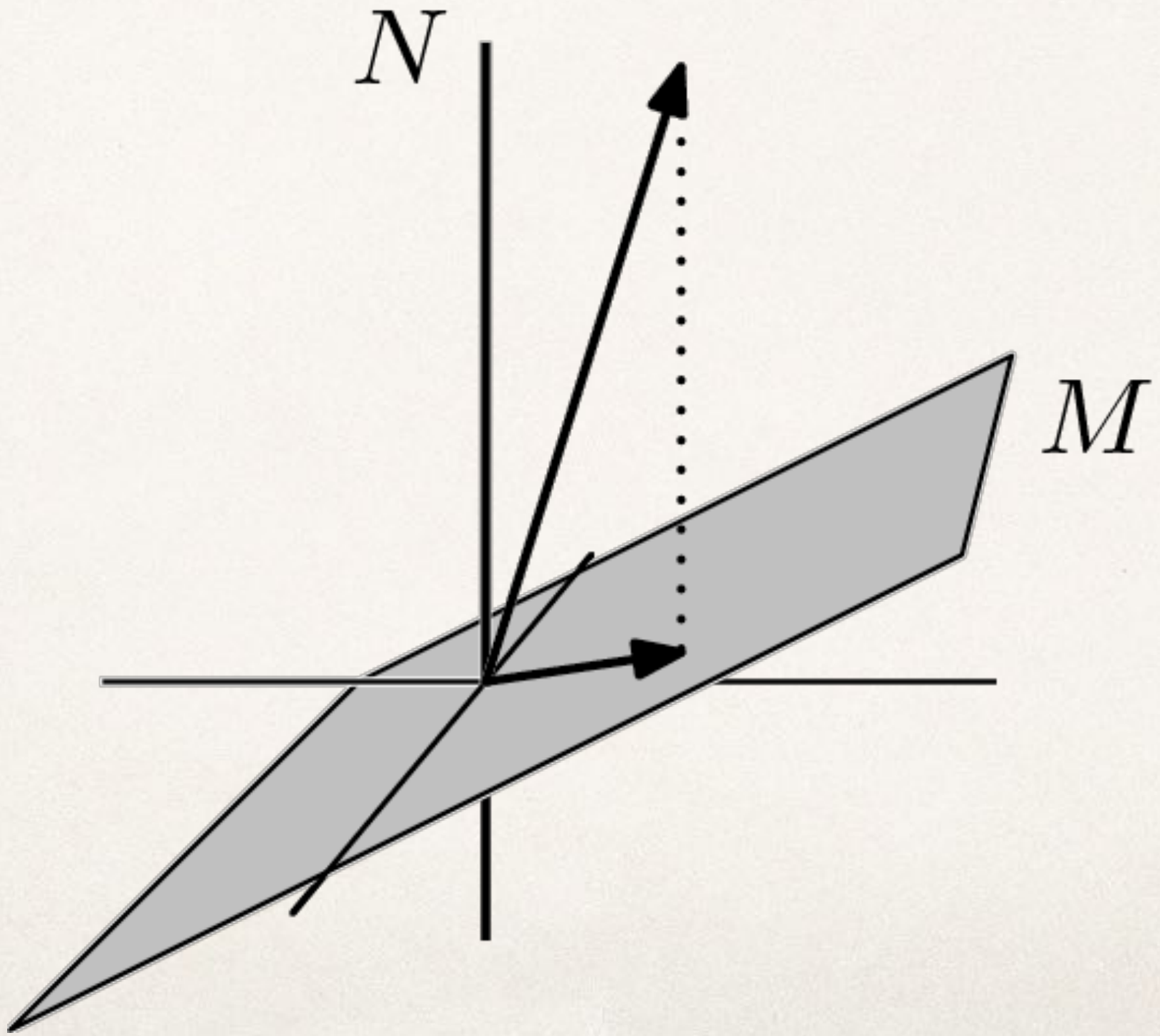
How?
Through composition
(matrix multiplication)

Scaling, Rotation, & Projection are Linear Functions

- Consider projecting a vector into a plane.
  - If you double the length of the vector, this doubles the length of the projection.
- If you project the sum of two vectors, this projects to the sum of the projections.

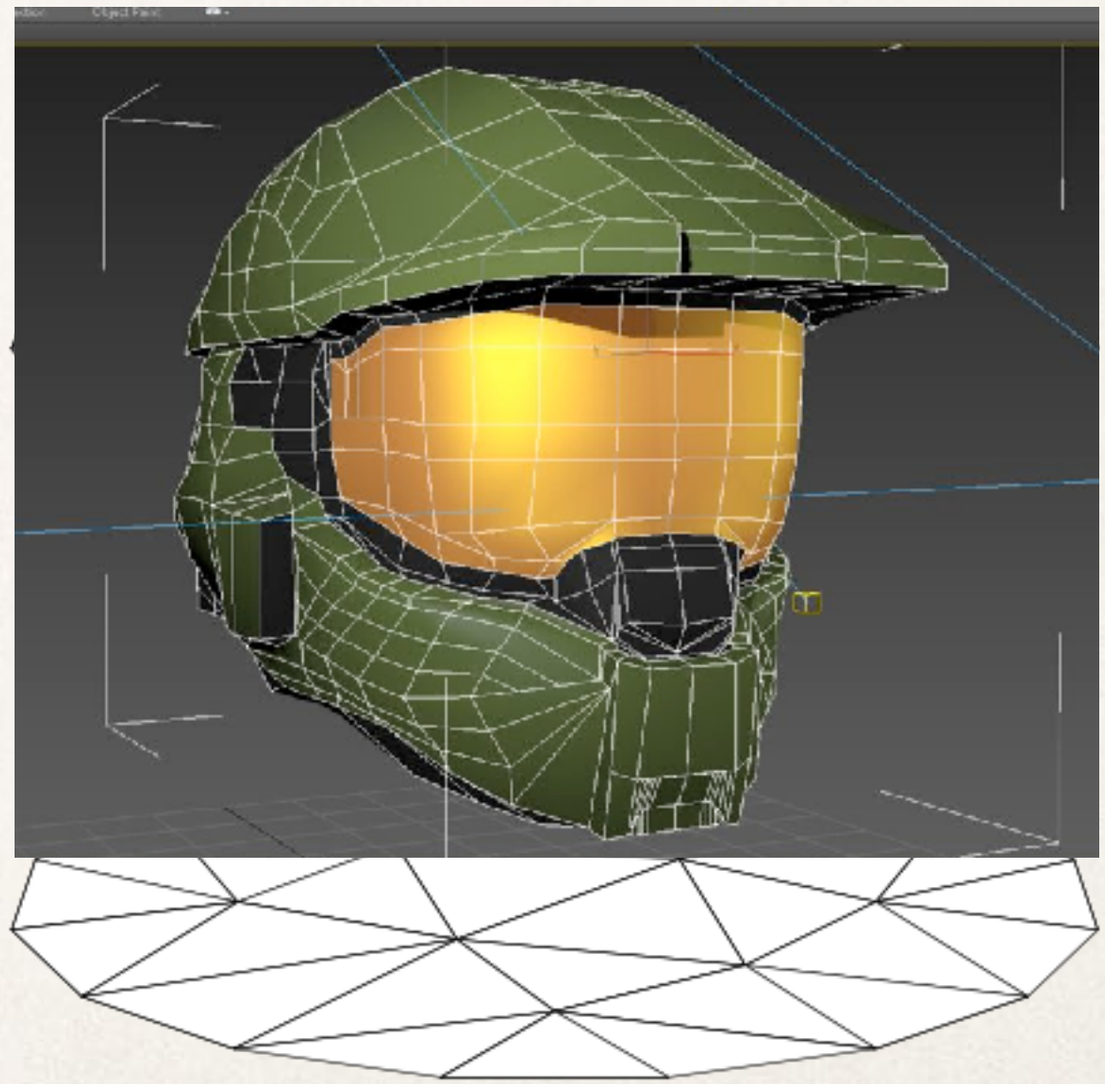
A function with these properties
is a *linear function* (of a vector).

Projection in a picture



Linear maps are fundamental in geometry, graphics, physics, …

- Linear maps can give good local approximations of more complicated objects. (That's what calculus is a all about.)
- **Computation with linear maps is (fairly) easy.**



Linear maps can be expressed… as *matrices*

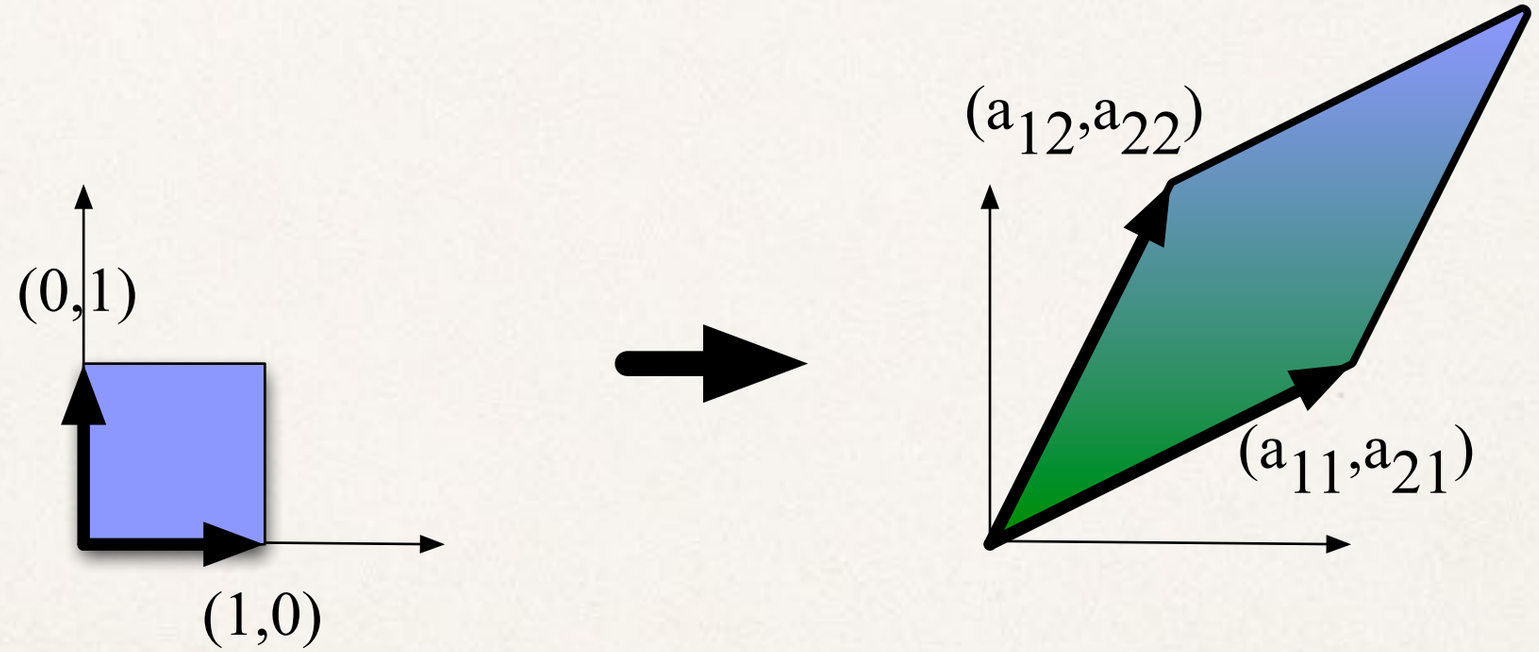
The matrix

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$$

represents the map that takes

$$\begin{bmatrix} 1 \\ 0 \end{bmatrix} \to \begin{bmatrix} a_{11} \\ a_{21} \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} 0 \\ 1 \end{bmatrix} \to \begin{bmatrix} a_{12} \\ a_{22} \end{bmatrix}$$

In pictures…



Applying a linear map to a vector

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} ax + by \\ cx + dy \end{bmatrix}$$

Then

- The identity map:

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

- Scaling everything by 1/2:

$$\begin{bmatrix} 1/2 & 0 \\ 0 & 1/2 \end{bmatrix}$$

- Scaling just the x-axis by 1/2:

$$\begin{bmatrix} 1/2 & 0 \\ 0 & 1 \end{bmatrix}$$

- Projection onto the line spanned by (1,1):

$$\begin{bmatrix} 1/2 & 1/2 \\ 1/2 & 1/2 \end{bmatrix}$$

- Rotation by 30 degrees:

$$\begin{bmatrix} \cos 30^\circ & -\sin 30^\circ \\ \sin 30^\circ & \cos 30^\circ \end{bmatrix}$$

Linear maps can be composed my *matrix multiplication*

- Scale x axis by 1/2 after rotation by 30 degrees.

$$\begin{bmatrix} 1/2 & 0 \\ 0 & 1 \end{bmatrix} . \begin{bmatrix} \cos 30° & -\sin 30° \\ \sin 30° & \cos 30° \end{bmatrix} = \begin{bmatrix} (1/2)\cos 30° & -(1/2)\sin 30° \\ \sin 30° & \cos 30° \end{bmatrix}$$

Matrix Multiplication?

- What it does
  - Computes a *new* matrix
  - Each element of the new matrix is an inner-product

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} * \begin{bmatrix} \alpha & \beta \\ \gamma & \delta \end{bmatrix} = \begin{bmatrix} a \cdot \alpha + b \cdot \gamma & a \cdot \beta + b \cdot \delta \\ c \cdot \alpha + d \cdot \gamma & c \cdot \beta + d \cdot \delta \end{bmatrix}$$

Matrix Multiplication?

- What it does
  - Computes a *new* matrix
  - Each element of the new matrix is an inner-product

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} * \begin{bmatrix} \alpha & \beta \\ \gamma & \delta \end{bmatrix} = \begin{bmatrix} a \cdot \alpha + b \cdot \gamma & a \cdot \beta + b \cdot \delta \\ c \cdot \alpha + d \cdot \gamma & c \cdot \beta + d \cdot \delta \end{bmatrix}$$

Matrix Multiplication?

- What it does
  - Computes a *new* matrix
  - Each element of the new matrix is an inner-product

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} * \begin{bmatrix} \alpha & \beta \\ \gamma & \delta \end{bmatrix} = \begin{bmatrix} a \cdot \alpha + b \cdot \gamma & a \cdot \beta + b \cdot \delta \\ c \cdot \alpha + d \cdot \gamma & c \cdot \beta + d \cdot \delta \end{bmatrix}$$

Matrix Multiplication?

- What it does
  - Computes a *new* matrix
  - Each element of the new matrix is an inner-product

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} * \begin{bmatrix} \alpha & \beta \\ \gamma & \delta \end{bmatrix} = \begin{bmatrix} a \cdot \alpha + b \cdot \gamma & a \cdot \beta + b \cdot \delta \\ c \cdot \alpha + d \cdot \gamma & c \cdot \beta + d \cdot \delta \end{bmatrix}$$

Matrix multiplication… on larger matrices

$$\begin{bmatrix} \vdots \\ R_i \\ \vdots \end{bmatrix} \cdot \begin{bmatrix} \dots & C_j & \dots \end{bmatrix} = \begin{bmatrix} \ddots & \vdots & \ddots \\ \dots & R_i \odot C_j & \dots \\ \ddots & \vdots & \ddots \end{bmatrix}$$

R_i is the ith row

C_j is the jth column

$(\mathrm{i},\mathrm{j})^{\mathrm{th}}$ entry

Interesting Functions to write...

- SCHEME Functions to
  - Create a matrix
  - Fill a matrix
  - Compute the image of a point by a matrix
  - Multiply two matrices

Defining a n x m matrix

```
(define (make-matrix n m)
  (let ((rows (make-vector n '())))
    (do ((i 0 (+ i 1)))
        ((>= i n) rows)
      (vector-set! rows i (make-vector m 0)))))
```

What is inside the matrix ?

A vector of vectors

Multiplying two matrices

```
(define (mat-prod m1 m2)
  (define (matrix-set! m i j v) (vector-set! (vector-ref m i) j v))
  (define (matrix-get m i j)    (vector-ref (vector-ref m i) j))
  (let* ((m1row (vector-length m1))
         (m1col (vector-length (vector-ref m1 0)))
         (m2col (vector-length (vector-ref m2 0)))
       (result (make-matrix m1row m2col)))
(do ((i 0 (+ i 1)))
       ((>= i m1row) result)
       (do ((j 0 (+ j 1)))
           ((>= j m2col) result)
           (matrix-set! result i j
                   (let ((sp 0))
                        (do ((k 0 (+ k 1)))
                         ((>= k m1col) sp)
                         (set! sp (+ sp (* (matrix-get m1 i k)
                                           (matrix-get m2 k j))))))
                  )))))))
```

Language conventions: C, Java, Pascal, ...

- Destructive assignment is common practice in many “procedural” languages.

- While functional programming is possible, they were designed to optimize usage of destructive assignment. In C, e.g., it is not possible to nest function definitions.

Another example: Computing the largest element of a list of integers

- In a single environment, this steps through a list and destructively updates `current-max` if the current `car` is larger. (It assumes all elements are positive.)

```
(define (maximum l)
  (let ((current-max 0))
    (do ((node l (cdr node)))
      ((null? node) current-max)
      (if (> (car node) current-max)
          (set! current-max (car node))))))

>(maximum (list 1 2 3))
3
```

Step through the list

Update the largest

Efficiency of explicit looping versus recursive iteration

- One reason to prefer explicit looping is that it can be carried out with minimal overhead: no new environments are created to change the "iterating variables."

- Scheme handles this by giving built-in support for tail-recursion optimization.

- Recall: a function is tail-recursive if the *value* returned by the function is a recursive call.

```
(define (fact n)
  (if (= n 0) 1
      (* n
         (fact (- n 1)))))
```

Pending

This is not tail recursive

Tail recursive

```
(define (fact-accumulate n a)
  (if (= n 0) a
      (fact-accumulate (- n 1)
                       (* n a))))
```

Tail-recursion Optimization

- In a tail recursive call, no new environment needs to be created (and no state needs to be remembered on behalf of the caller). Instead the arguments are re-bound to their new values, and the body is evaluated with these bindings.

- This means that a tail recursive function can, in principle, be evaluated as quickly as an explicit loop.

- However, writing tail-recursive code takes discipline. It often involved passing state as a variable...as in the previous example.

Continuation passing: a principled method to induce tail recursion

- Instead of: call `f` on `<arg>`, then apply `g` to the result:

```
(g (f <arg>))        e.g.        (* n (fact (- n 1))
```

- We call f^c with `<arg>` and ask it to apply `g` to the value when it is complete. In this case `g` is the *continuation*: what would have been done when `f` returned.

```
(f^c <arg> g)
```

e. g.

The continuation

```
(fact-c (- n 1) (lambda (k) (* k n))
```

Factorial, continuation passing style

- Basic ingredient: function fact-c that computes the factorial of its first argument and then...*applies its second argument to the result*.

Compute factorial of `m`, then apply `c`

```
(define (fact-c m c)
  (if (= m 0) (c 1)
      (fact-c (- m 1)
              (lambda (x) (c (* m x))))))
```

Note! In order to be tail recursive, `fact-c` asks the recursively called `fact-c` to finish the computation, multiplying by `m`. (and, then, applying the continuation `c` fact-c was called with.).

The final product: `factorial` in continuation passing style

```
(define (fact-cps n)
  (define (fact-c m c)
    (if (= m 0) (c 1)
        (fact-c (- m 1)
                (lambda (x) (c (* m x))))))
  (fact-c n (lambda (x) x)))
```

- Note: It's tail recursive. Observe how the continuations "pile-up" in the recursive call. This second argument holds all of the pending operations.
- Note: initially, we simply call `fact-c` with the identity continuation.
- Note: unnamed functions are convenient (but not necessary).