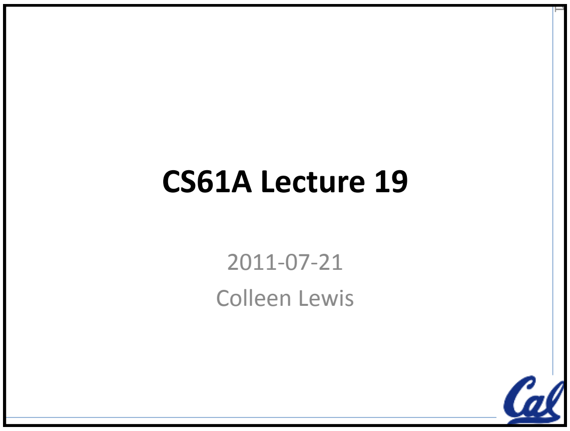
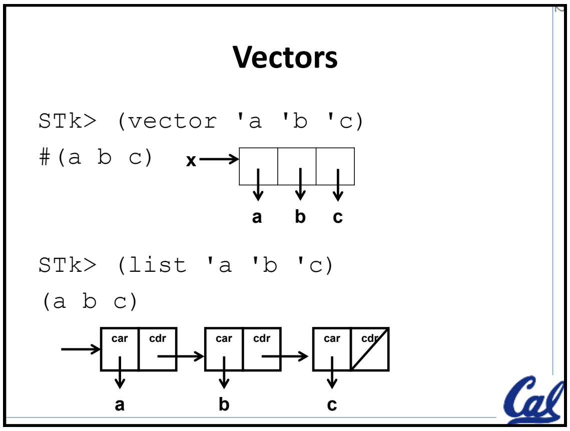


CS61A Lecture 19

2011-07-21

Colleen Lewis



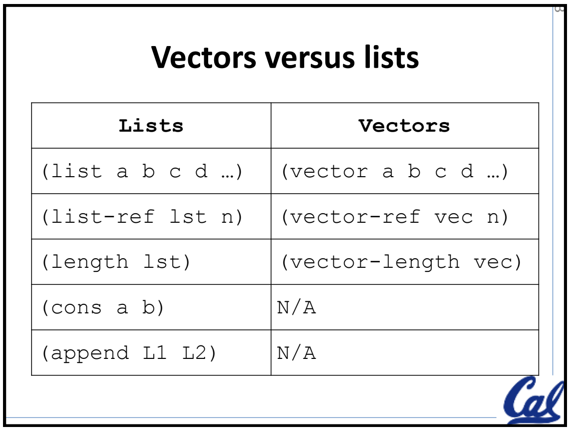
Vectors

```
STk> (vector 'a 'b 'c)
#(a b c)
```

x
a
b
c

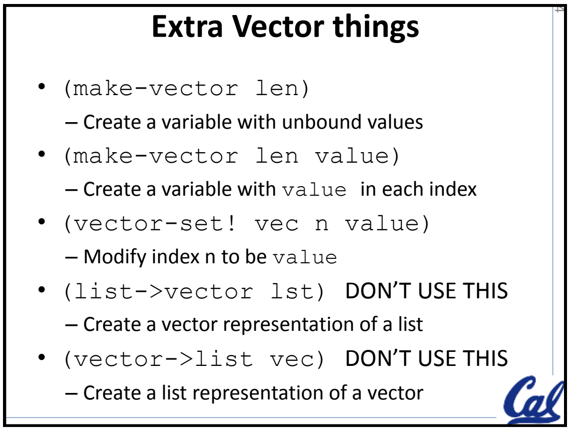
```
STk> (list 'a 'b 'c)
(a b c)
```

car
cdr
car
cdr
car
cdr
a
b
c



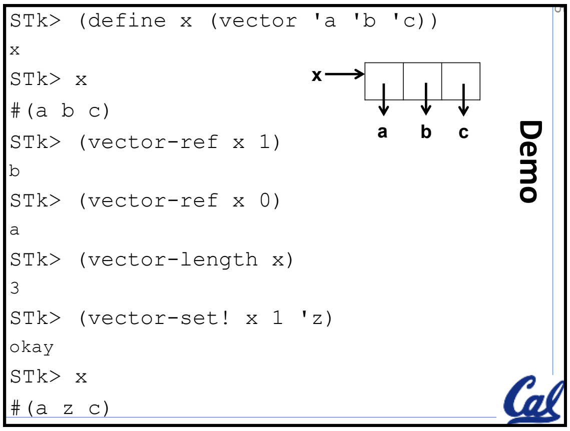
Vectors versus lists

Lists	Vectors
(list a b c d …)	(vector a b c d …)
(list-ref lst n)	(vector-ref vec n)
(length lst)	(vector-length vec)
(cons a b)	N/A
(append L1 L2)	N/A

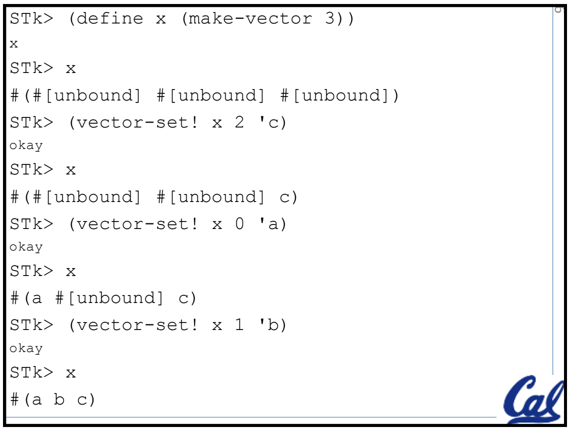


Extra Vector things

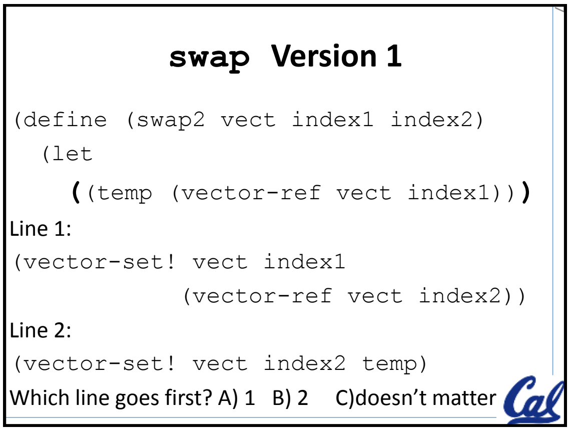
- `(make-vector len)`
  - Create a variable with unbound values
- `(make-vector len value)`
  - Create a variable with `value` in each index
- `(vector-set! vec n value)`
  - Modify index n to be `value`
- `(list->vector lst)` DON'T USE THIS
  - Create a vector representation of a list
- `(vector->list vec)` DON'T USE THIS
  - Create a list representation of a vector



```
STk> (define x (vector 'a 'b 'c))
x
STk> x
#(a b c)
STk> (vector-ref x 1)
b
STk> (vector-ref x 0)
a
STk> (vector-length x)
3
STk> (vector-set! x 1 'z)
okay
STk> x
#(a z c)
```



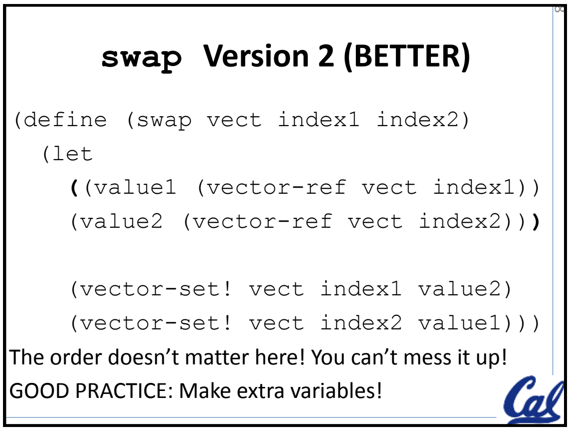
```
STk> (define x (make-vector 3))
x
STk> x
#(#[unbound] #[unbound] #[unbound])
STk> (vector-set! x 2 'c)
okay
STk> x
#(#[unbound] #[unbound] c)
STk> (vector-set! x 0 'a)
okay
STk> x
#(a #[unbound] c)
STk> (vector-set! x 1 'b)
okay
STk> x
#(a b c)
```



swap Version 1

```
(define (swap2 vect index1 index2)
  (let
    ((temp (vector-ref vect index1)))
Line 1:
(vector-set! vect index1
              (vector-ref vect index2))
Line 2:
(vector-set! vect index2 temp)
```

Which line goes first? A) 1 B) 2 C)doesn't matter



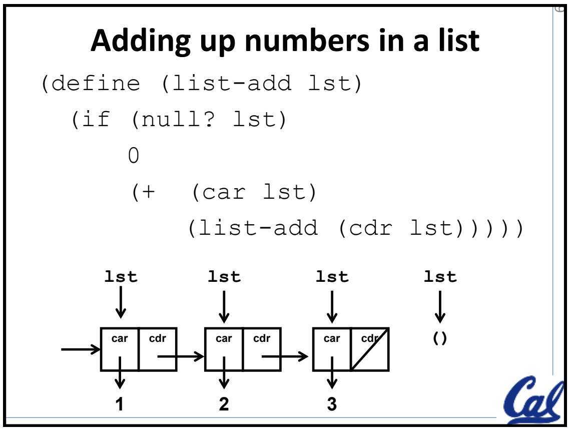
swap Version 2 (BETTER)

```
(define (swap vect index1 index2)
  (let
    ((value1 (vector-ref vect index1))
    (value2 (vector-ref vect index2)))

    (vector-set! vect index1 value2)
    (vector-set! vect index2 value1)))
```

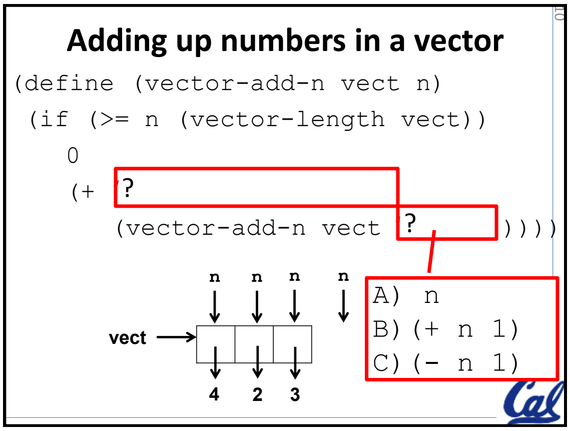
The order doesn't matter here! You can't mess it up!

GOOD PRACTICE: Make extra variables!



Adding up numbers in a list

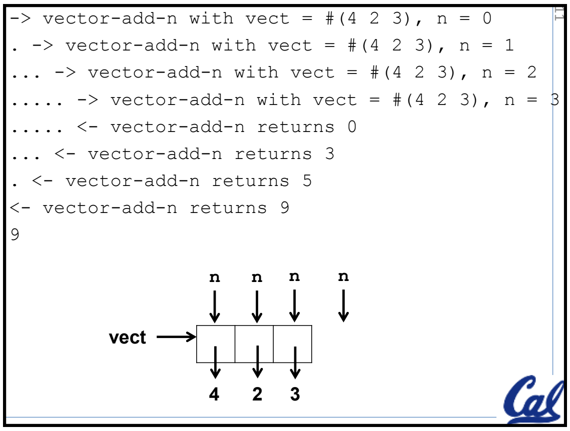
```
  (define (list-add lst)
    (if (null? lst)
        0
        (+  (car lst)
            (list-add (cdr lst)))))
```



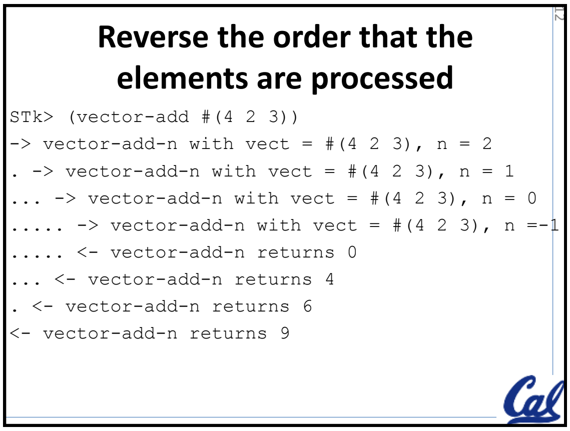
Adding up numbers in a vector

```
(define (vector-add-n vect n)
 (if (>= n (vector-length vect))
    0
    (+ ?
       (vector-add-n vect ? ))))
```

A) n
B) (+ n 1)
C) (- n 1)

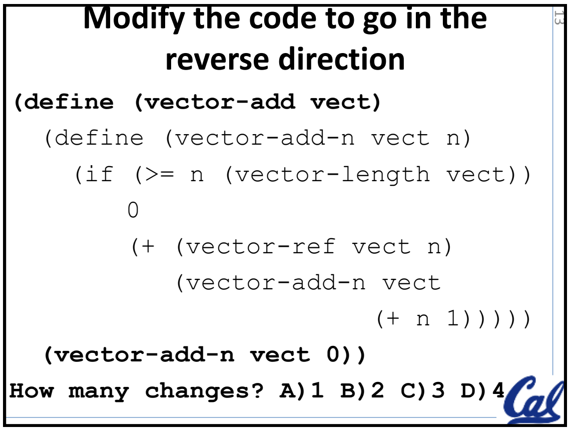


```
-> vector-add-n with vect = #(4 2 3), n = 0
. -> vector-add-n with vect = #(4 2 3), n = 1
... -> vector-add-n with vect = #(4 2 3), n = 2
..... -> vector-add-n with vect = #(4 2 3), n = 3
..... <- vector-add-n returns 0
... <- vector-add-n returns 3
. <- vector-add-n returns 5
<- vector-add-n returns 9
9
```



Reverse the order that the elements are processed

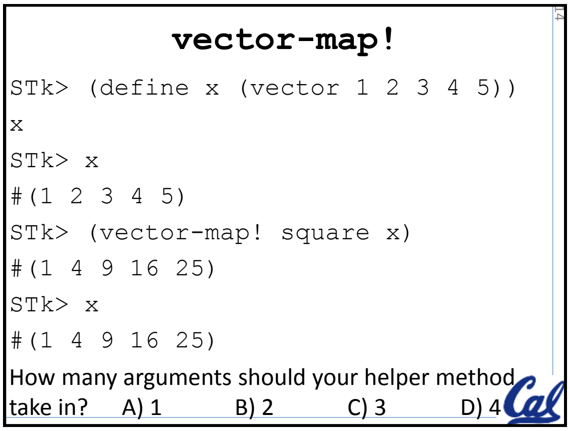
```
STk> (vector-add #(4 2 3))
-> vector-add-n with vect = #(4 2 3), n = 2
. -> vector-add-n with vect = #(4 2 3), n = 1
... -> vector-add-n with vect = #(4 2 3), n = 0
..... -> vector-add-n with vect = #(4 2 3), n =-1
..... <- vector-add-n returns 0
... <- vector-add-n returns 4
. <- vector-add-n returns 6
<- vector-add-n returns 9
```



Modify the code to go in the reverse direction

```
(define (vector-add vect)
  (define (vector-add-n vect n)
    (if (>= n (vector-length vect))
        0
        (+ (vector-ref vect n)
           (vector-add-n vect
                         (+ n 1)))))
  (vector-add-n vect 0))
```

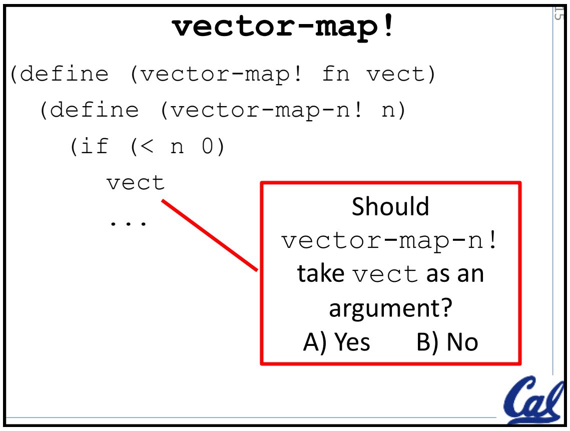
`How many changes? A)1 B)2 C)3 D)4`



vector-map!

```
STk> (define x (vector 1 2 3 4 5))
x
STk> x
#(1 2 3 4 5)
STk> (vector-map! square x)
#(1 4 9 16 25)
STk> x
#(1 4 9 16 25)
```

How many arguments should your helper method take in? A) 1 B) 2 C) 3 D) 4

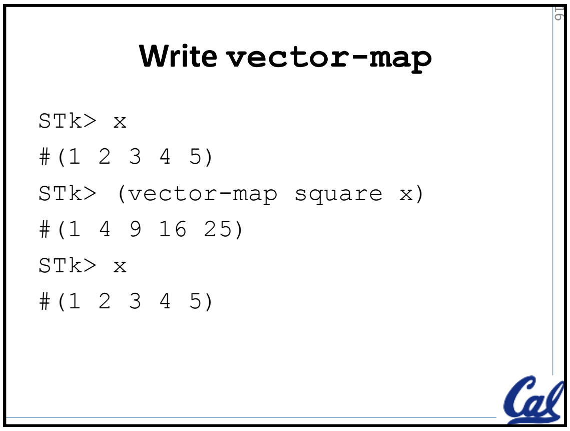


vector-map!

```
(define (vector-map! fn vect)
  (define (vector-map-n! n)
    (if (< n 0)
        vect
        ...
```

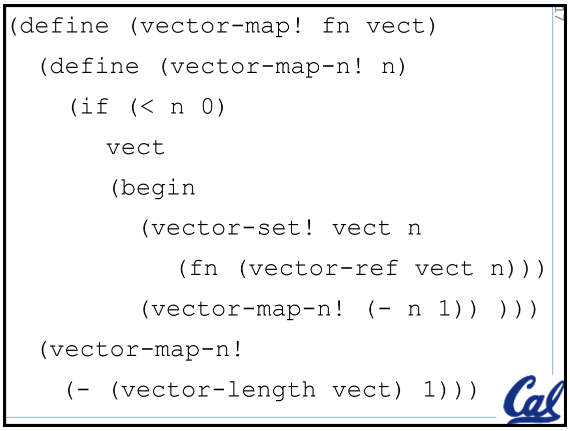
Should `vector-map-n!` take `vect` as an argument?

A) Yes B) No

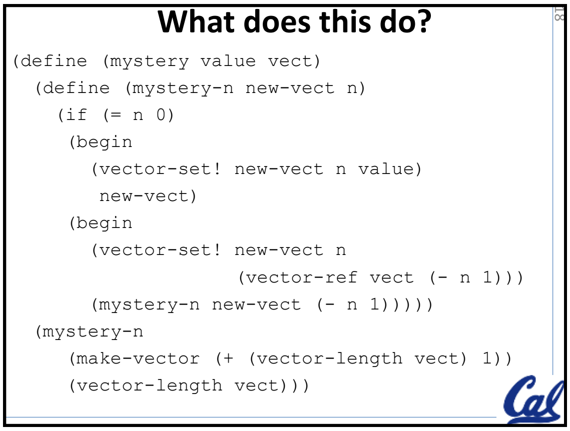


Write vector-map

```
STk> x
#(1 2 3 4 5)
STk> (vector-map square x)
#(1 4 9 16 25)
STk> x
#(1 2 3 4 5)
```



```
(define (vector-map! fn vect)
  (define (vector-map-n! n)
    (if (< n 0)
        vect
        (begin
           (vector-set! vect n
              (fn (vector-ref vect n)))
           (vector-map-n! (- n 1)) )))
  (vector-map-n!
   (- (vector-length vect) 1)))
```



What does this do?

```
(define (mystery value vect)
  (define (mystery-n new-vect n)
    (if (= n 0)
     (begin
       (vector-set! new-vect n value)
        new-vect)
     (begin
       (vector-set! new-vect n
                    (vector-ref vect (- n 1)))
       (mystery-n new-vect (- n 1)))))
  (mystery-n
     (make-vector (+ (vector-length vect) 1))
     (vector-length vect)))
```

What does this do? (cont.)

```
STk> (define x (vector 1 2 3))
x
STk> (mystery 4 x)
A) #(4 1 2 3)
B) #(1 2 3 4)
C) #(4 4 4 4)
D) #(1 1 1 1)
E) Other
STk> x
```

Was x changed? A) Yes B) No

Tradeoffs with runtime

List	Vectors
list-ref Θ(n)	vector-ref Θ(1)
cons Θ(1)	vector-cons Θ(n)

Which one NEEDS a helper procedure?

- A) `(list->vector lst)`
  - Create a vector representation of a list
- B) `(vector->list vec)`
  - Create a list representation of a vector
- C) Neither
- D) Both
  - DON'T USE these in the homework. If the point is to learn to do things with vectors, we don't want you to change them to lists.

vect-->list SOLUTION

```
(define (vect-->list vect)
```

How many arguments does your helper method take?

a) 0

b) 1

c) 2

d) 3

e) N/A

list-->vector SOLUTION

```
(define (list-->vector lst)
```

Does your n (index variable) go up or down?

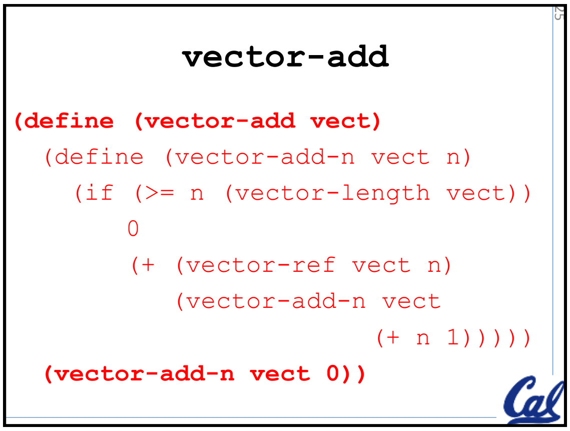
a) Up

b) Down

c) N/A

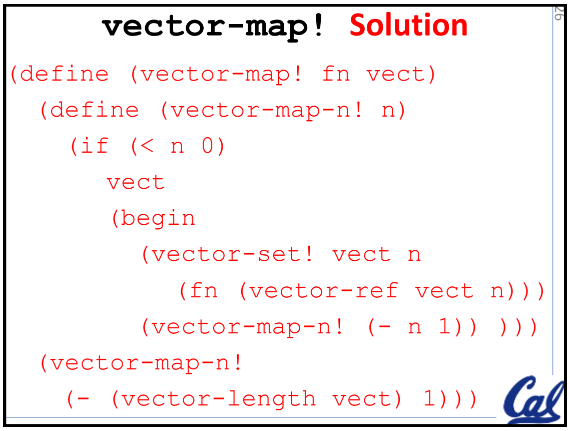
Modify the code to go in the reverse direction

```
(define (vector-add vect)
  (define (vector-add-n vect n)
    (if ~~(>= n (vector-length vect))~~)
        0 (< n 0)
        (+ (vector-ref vect n)
           (vector-add-n vect
                         (~~+~~ n 1)))))
  (vector-add-n vect ~~0~~))      –
How many ch(vector-length vect)
```



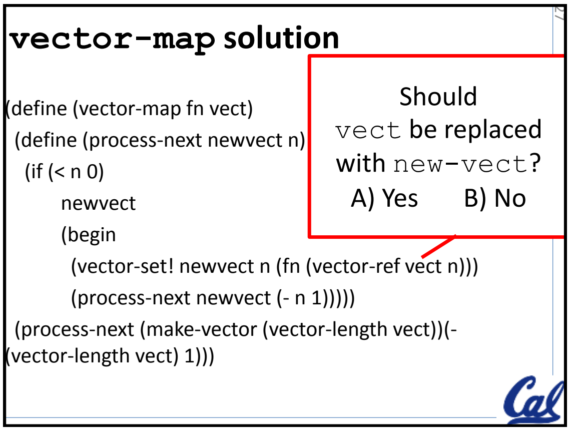
vector-add

```
(define (vector-add vect)
  (define (vector-add-n vect n)
    (if (>= n (vector-length vect))
        0
        (+ (vector-ref vect n)
           (vector-add-n vect
                         (+ n 1)))))
  (vector-add-n vect 0))
```



vector-map! Solution

```
(define (vector-map! fn vect)
  (define (vector-map-n! n)
    (if (< n 0)
        vect
        (begin
          (vector-set! vect n
              (fn (vector-ref vect n)))
          (vector-map-n! (- n 1)) )))
  (vector-map-n!
    (- (vector-length vect) 1)))
```

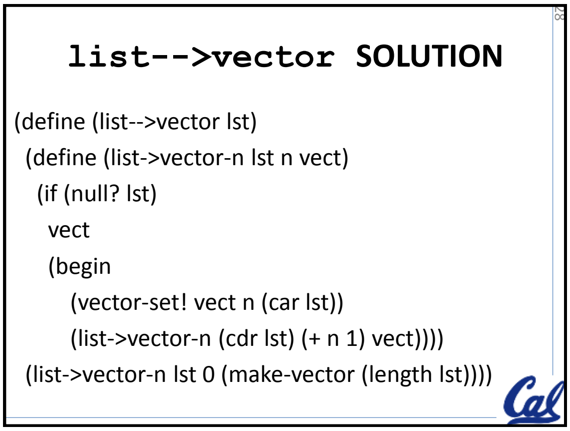


vector-map solution

```
(define (vector-map fn vect)
 (define (process-next newvect n)
  (if (< n 0)
      newvect
      (begin
       (vector-set! newvect n (fn (vector-ref vect n)))
       (process-next newvect (- n 1)))))
 (process-next (make-vector (vector-length vect))(-
(vector-length vect) 1)))
```

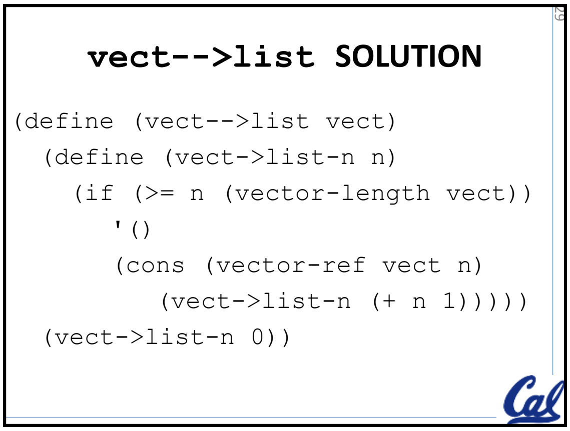
Should vect be replaced with new-vect?

A) Yes B) No



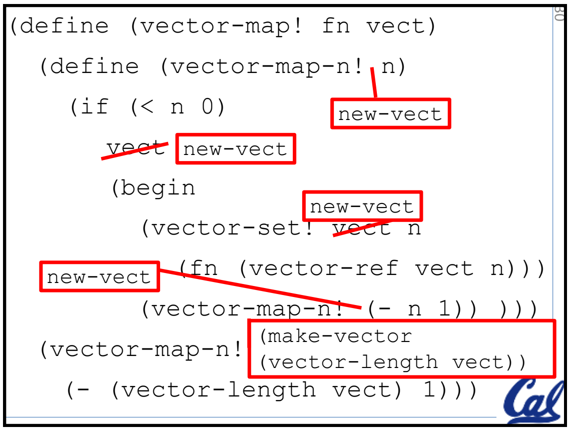
list-->vector SOLUTION

```
(define (list-->vector lst)
 (define (list->vector-n lst n vect)
  (if (null? lst)
   vect
   (begin
     (vector-set! vect n (car lst))
     (list->vector-n (cdr lst) (+ n 1) vect))))
 (list->vector-n lst 0 (make-vector (length lst))))
```



vect-->list SOLUTION

```
(define (vect-->list vect)
  (define (vect->list-n n)
    (if (>= n (vector-length vect))
        '()
        (cons (vector-ref vect n)
              (vect->list-n (+ n 1)))))
  (vect->list-n 0))
```



```
(define (vector-map! fn vect)
  (define (vector-map-n! new-vect n)
    (if (< n 0)
        ~~vect~~ new-vect
        (begin
                            new-vect
          (vector-set! ~~vect~~ n
              (fn (vector-ref vect n)))
          (vector-map-n! new-vect (- n 1)) )))
  (vector-map-n! (make-vector
                  (vector-length vect))
    (- (vector-length vect) 1)))
```