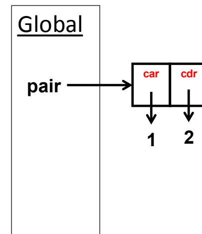
CS61A Lecture 18

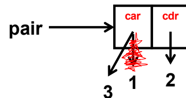
2011-07-20

Colleen Lewis

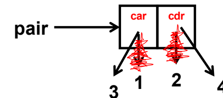
```
STk> (define pair (cons 1 2))
```



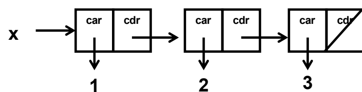
```
STk> (define pair (cons 1 2))
STk> (set-car! pair 3)
```



```
STk> (define pair (cons 1 2))
STk> (set-car! pair 3)
STk> (set-cdr! pair 4)
```



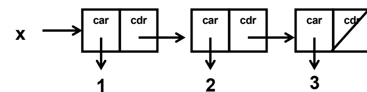
```
STk> (define x (list 1 2 3))
STk> (set-cdr! x 4)
```



What does scheme print?

A) Error
B) (1 . 4)
C) (1 4)
D) (4 2 3)
E) other

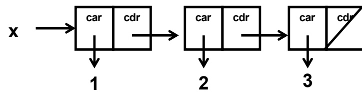
```
STk> (define y (list 1 2 3))
STk> (set-cdr! (cdr x) 4)
```



What does scheme print?

A) Error
B) (1 . 4)
C) (1 2 . 4)
D) (1 2 4)
E) (1 2 3 4)

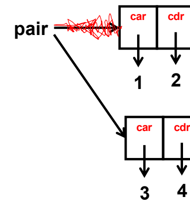
```
STk> (define z (list 1 2 3))
STk> (set! (cdr z) 5)
```



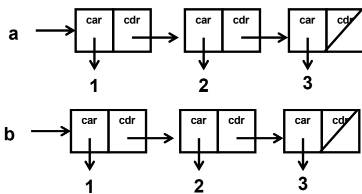
A) Error
B) (1 . 5)
C) (1 2 . 5)
D) (1 2 5)
E) (1 2 3 5)

What does scheme print?

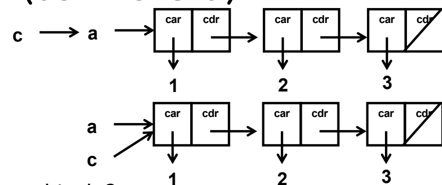
```
STk> (define pair (cons 1 2))
STk> (set! pair (cons 3 4))
```




```
STk> (define a (list 1 2 3))
STk> (define b (list 1 2 3))
```




```
STk> (define a (list 1 2 3))
STk> (define b (list 1 2 3))
STk> (define c a)
```



What does this do?

A) Top picture
B) Bottom picture
C) Other

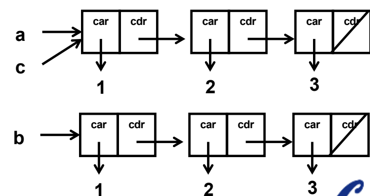
Eq? versus Equal?

- Equal?
  - Checks if two things print the same in Scheme.
- Eq?
  - Checks if two things are LITERALLY the same/identical

```
STk> (define a (list 1 2 3))
STk> (define b (list 1 2 3))
STk> (define c a)
STk> (equal? a b)
```

What does this do?

A) #t
B) #f
C) ???



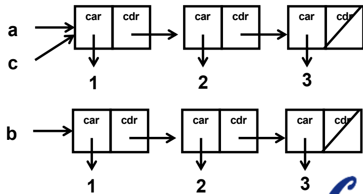
```
STk> (define a (list 1 2 3))
STk> (define b (list 1 2 3))
STk> (define c a)
STk> (eq? a b)
```

What does this do?

A) #t

B) #f

C) ???



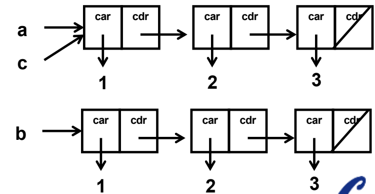

```
STk> (define a (list 1 2 3))
STk> (define b (list 1 2 3))
STk> (define c a)
STk> (equal? a c)
```

What does this do?

A) #t

B) #f

C) ???



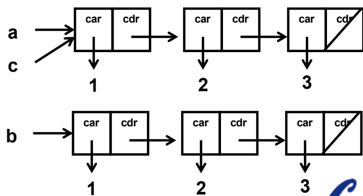

```
STk> (define a (list 1 2 3))
STk> (define b (list 1 2 3))
STk> (define c a)
STk> (eq? a c)
```

What does this do?

A) #t

B) #f

C) ???



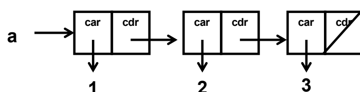
What is a after the call to `sq-list!`?

```
(define (sq-list! lst)
 (if (null? lst)
   lst
   (begin
     (set-car! lst (square (car lst)))
     (sq-list! (cdr lst)))))
STk> (define a (list 1 2 3))
a
STk> (sq-list! a)
```

A) `()` B) `(1 2 3)` C) `(1 4 9)` D) `Error`

What is a after the call to `sq-list!`?

```
(define (sq-list! lst)
 (if (null? lst)
   lst
   (begin
     (set-car! lst (square (car lst)))
     (sq-list! (cdr lst)))))
```




```
STk> (define a (list 1 2 3))
a
STk> (remove-last a)
okay
STk> a
(1 2)
STk> (remove-last a)
okay
STk> a
(1)
STk> (remove-last a)
(cdr of list is null - not possible to remove)
STk> (remove-last '())
(list is null - not possible to remove last)
```

Write `remove-last`. Show solution with:

A) Chalk

B) Emacs

C) PPT

TRACE mystery

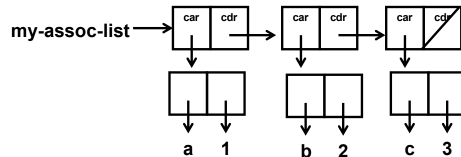
```
(define (mystery lst)
  (if (or (null? lst)
          (null? (cdr lst)))
      'ok
      (let ((x (cdr lst)))
        (set-cdr! lst (cdr x)))))
```


Trace mystery

```
STk> (define a (list 1 2 3 4))
a
STk> (mystery a)
okay
STk> (mystery (cdr a))
okay
STk> a
A.(1 3) B.(1 4) C.(2 3) D.(3 4)
E. Error
```


What does an association list look like?

```
STk> (define my-assoc-list
    '((a . 1) (b . 2) (c . 3)))
```



assoc

```
STk> (define my-assoc-list
    '((a . 1) (b . 2) (c . 3)))
my-assoc-list
STk> (assoc 'a my-assoc-list)
(a . 1)
STk> (assoc 'b my-assoc-list)
(b . 2)
STk> (assoc 'd my-assoc-list)
#f
```

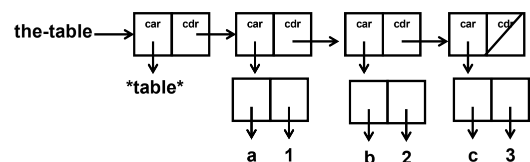

assoc

```
(define (assoc key records)
  (cond
    ((null? records)
          #f)
    ((equal? key [          ])
          (car records))
    (else
        (assoc key (cdr records)))))
```

(____ records) the blank is:
A) car B) caar C) cdr D) cadr

Tables

```
(define the-table (list '*table*))
```



Tables

```
(define (get key)
  (let
    ((record
        (assoc key (cdr the-table))))
     (if [          ]
         #f
         (cdr record))))
```

the blank is:
A. (null? record) B. (record)
C. (not (record)) D. Not poss.
E. (not record)


```
STk> (define the-table (list '*table*))
the-table
STk> (put 'a '1)
ok
STk> the-table
(*table* (a . 1))
STk> (put 'b 2)
ok
STk> the-table
(*table* (b . 2) (a . 1))
```



```
(define (put key value)
  (let ((record (assoc key (cdr the-table))))
    (if (not record)
        (set-cdr!
           [          ]
           [          ]
        (set-cdr!
           [          ]
           [          ]
    'ok)
```


Remove-last SOLUTION

```
(define (remove-last lst)
 (cond
  ((null? lst)
   '(list is null - not possible…))
  ((null? (cdr lst))
   '(cdr of list is null…))
  ((null? (cdr (cdr lst)))
      (set-cdr! lst '()))
  (else (remove-last (cdr lst)))))
```



```
(define (put key value)
  (let ((record (assoc key (cdr the-table))))
    (if (not record)
        (set-cdr!
           the-table
           (cons (cons key value)
                 (cdr the-table)))
        (set-cdr!
           record
           value)))
    'ok)
```


flat-list-filter!

```
(define (flat-list-filter! pred lst)
  (cond
   ((or (null? lst) (null? (cdr lst)))
    lst)
   ((pred (cadr lst))
    (remove-next! lst)
    (flat-list-filter! pred lst))
   (else (flat-list-filter! pred
                            (cdr lst)))))
```