Lists are made with pairs!

STk> (define a (list 1 2 3 4))

Make the Empty List the cdr

STk> (define b (list 1 2))

Make the Empty List the car

Dots
Dots

```
STk> (cons 1 '())
(1 . ())
STk> (cons 1 (cons 2 '()))
(1 . (2 . ()))
(1 2)
```

Practice Removing Dots

```
(cons
  (cons 1 '())
  (cons
    2
    (cons
      (cons 3 4)
      (cons 5 '()))))
```

Accessing Elements

Using `car` and `cdr`

```
STk> (define x (cons 2 '()))
x
STk> x
(2)
STk> (car x)
2
STk> (cdr x)
()
```

How do you get the 2?

```
STk> (define a (list 1 2 3 4))
a
A) (car (cdr a))
B) (cdr (car a))
C) (cdr (car (car a)))
D) (car (cdr (cdr a)))
E) (cdr (car (car a)))
```

The Empty List w/ `car` & `cdr`

```
STk> (define x (cons 2 '()))
x
STk> x
(2)
STk> (car x)
2
STk> (cdr x)
()
```

How do you get the 3?

```
STk> (define a (list 1 2 (list 3 4) 5))
a
A) (car (cdr (cdr a)))
B) (cdr (cdr (car a)))
C) (cdr (car (cdr (car a))))
D) (car (cdr (car (cdr a))))
E) ???
```
We don’t need no stinkin’ pairs

(define (cons x y)
  (lambda (which)
    (cond
      ((equal? which 'car) x)
      ((equal? which 'cdr) y)
      (else (error "Bad message" which))))))

(define (car pair) (define (cdr pair)
  (pair 'car)) (pair 'cdr))

Try It!

• Try to use this new cons!

Does it work the same way as before?
A) Yes
B) No
C) I don’t know

List Methods

list

• Takes any number of arguments and puts them in a list

append

• Takes two lists and turns them into one
• Both arguments MUST be lists

Examples
- (append '(cat) '(dog))  => '(cat dog)
- (append '(cat) '())  => '(cat)
- (append '() '(dog))  => '(dog)
- (append '(cat) '())  => '(cat ()
- (append '() '())  => '() () dog)
The truth about `append`

```
STk> (define a (list 1 2))
STk> (define b (list 3 4 5))
STk> (define c (append a b))
```

Cons

- Takes two arguments
- If the second arg is a list
  - Makes the first arg the car of the new list
  - Makes the second arg the cdr of the new list

```
(cons 'cat '(dog)) → '(cat dog)
(cons '(cat) '(dog)) → '(cat) (dog)
(cons '() '(dog)) → '(() dog)
(cons '(cat) 'dog) → '((cat) . dog)
```

Data Abstraction Goals

- To talk about things using meaning not how it is represented in the computer
- To be able to change how it is represented in the computer without people who use our program caring

Very Happy Code 😊

```
(define (total hand)
  (if (empty? hand)
      0
      (+ (rank (last hand))
          (total (remaining-cards hand)))))
```

Data Abstraction Violation (D.A.V.)

Use the right one or it is a DAV

<table>
<thead>
<tr>
<th>sentence &amp; word stuff</th>
<th>list stuff</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>first</code></td>
<td><code>car</code></td>
</tr>
<tr>
<td><code>butfirst</code></td>
<td><code>cdr</code></td>
</tr>
<tr>
<td><code>last</code></td>
<td><code>∅</code></td>
</tr>
<tr>
<td><code>butlast</code></td>
<td><code>∅</code></td>
</tr>
</tbody>
</table>
Use the right one or it is a DAV

<table>
<thead>
<tr>
<th>sentence &amp; word stuff</th>
<th>list stuff</th>
</tr>
</thead>
<tbody>
<tr>
<td>empty?</td>
<td>null?</td>
</tr>
<tr>
<td>sentence?</td>
<td>list?</td>
</tr>
<tr>
<td>item</td>
<td>list-ref</td>
</tr>
<tr>
<td>sentence</td>
<td>append, cons, list</td>
</tr>
</tbody>
</table>

Implement se2

(define (se2 a b)
  (cond
    ((and (word? a) (word? b)) (list a b))
    ((word? a) (cons a b))
    ((word? b) (append a (list b)))
    (else (append a b))))

Modify this to work with lists (without sub-lists)

(define (square-sent sent)
  (if (empty? sent)
      sent
      (sentence
        (square (first sent))
        (butfirst sent)))))

map2 (like every)

(define (map2 fn lst)
  (if (null? lst)
      lst
      (cons (fn (car lst))
           (map2 fn (cdr lst)))))

STk> (map2 square '(1 2 3 4))
(1 4 9 16)

The real map

(map procedure list1 list2...)

- procedure
  - a procedure that takes in some # of arguments
- Some # of lists
  - The number of lists MUST match the number of arguments that the procedure takes

map

(define (add-2-nums x y)
  (+ x y))

(map add-2-nums '(1 2 3)
  '(4 5 6)
  (5 7 9))
map

(define (add-3(nums x y z))
  (+ x y z))

(map add-3 nums '(1 2 3)
  (4 5 6)
  (7 8 9))

\(\Rightarrow\) '12 15 18

Solutions

Solution

Implement se2

(define (se2 a b)
  (cond
   ((and (word a) (word b))
    (list a b))
   ((word? a)
    (cons a b))
   ((word b)
    (append a (list b))))
   (else
    (append a b)))))

Modify this to work with lists
(without sub-lists)

(define (square-sent sent)
  (if (null? sent)
      sent
      (cons
        (square (car sent))
        (cdr sent)))))