Hannah Zhang
Simply Scheme: 17.1, 17.2, 17.8 - 17.12, 17.14

```
; prompt
; comments
; answers
; question
```

17.1
What will scheme print out

```
> (car '(Rod Chris Colin Hugh Paul))
'Rod
> (cadr '(Rod Chris Colin Hugh Paul))
'Chris
> (cdr '(Rod Chris Colin Hugh Paul))
'(Chris Colin Hugh Paul)
> (car 'Rod)
Error: a list is not given
> (cons '(Rod Argent) '(Chris White))
'((Rod Argent) Chris White))
> (append '(Rod Argent) '(Chris White))
'(Rod Argent Chris White)
> (list '(Rod Argent) '(Chris White))
'((Rod Argent) (Chris White))
> (caadr '((Rod Argent) (Chris White)
           (Colin Blunstone) (Hugh Grundy) (Paul Atkinson)))
'Chris
> (assoc 'Colin '((Rod Argent) (Chris White)
            (Colin Blunstone) (Hugh Grundy) (Paul Atkinson)))
'(Colin Blunstone)
> (assoc 'Argent '((Rod Argent) (Chris White)
             (Colin Blunstone) (Hugh Grundy) (Paul Atkinson)))
#f
```

17.2

Write a procedure that returns the sample result

```
; ignore the dot

> (f1 '(a b c) '(d e f))
((B C D))
(define (f1 x y)
  (list (append (cdr x) (car y))))

> (f2 '(a b c) '(d e f))
((B C) E)
(define (f2 x y)
  (cons (cdr x) (cadr y)))

> (f3 '(a b c) '(d e f))
(A B C A B C)
(define (f3 x y)
  (append x x))

> (f4 '(a b c) '(d e f))
((A D) (B C E F))
; (A D) is cons since append only works with two lists
(define (f4 x y)
  (list (cons (car x) (car y)) (append (cdr x) (cdr y))))
```

17.8

Write member

```
; the difference between member? and member is that member must
take a list as the second argument and instead of returning #t,
it returns the portion of the list starting with the true
element

(define (member2 wd list)
  (if (null? list)
      #f
      (if (equal? wd (car list))
          list
```

```
          (member2 wd (cdr list)))))
```

17.9

Write list-ref

```
; similar to item except list-ref takes a list and then a number
instead of the other way around and it counts from 0 instead of
1

(define (list-ref2 list num)
  (if (null? list)
      #f
      (if (= num 0)
          (car list)
          (list-ref2 (cdr list) (- num 1)))))
```

17.10

Write length

```
; length is equivalent to count except it only takes lists
```

; this version creates a variable that is constantly updated	; this version is recursive (from last year)

(define (length2 list)
  (define (iter list x)
  (if (null? list)
  x
  (iter (cdr list) (+ x 1))))
  (iter list 0))
	(define (length3 list)
	  (if (null? list)
	  0
	  (+ 1 (length3 (cdr list)))))

17.11

Write before-in-list?, which takes a list and two elements of the list. It should return #t if the second argument appears in the list argument before the third argument:

```
(define (before-in-list? list x y)
  (if (null? list)
      #f
      (cond ((equal? x (car list)) #t)
```

```
            ((equal? y (car list)) #f)
            (else (before-in-list? (cdr list) x y)))))
```

17.12

Write a procedure called flatten. It should return a sentence without any sublists.

Mr. Paley helpers

- (f '()) → '()
- (f '((1 2) 3 4)
  - Car: '(1 2)
  - Cdr: '(3 4)
  - Append + recursion
- (f '(1 2 3))
  - Car: 1
  - Cdr: '(2 3)
  - Cons + recursion

```
; first condition is if it is empty
; second condition is if the first of the list is a list
```
- if so, append the car to the cdr
- then, call flatten to check for multiple nested lists

```
; else is if the car of list is not a list (already flattened)
```
- then we call flatten on cdr to flatten the next part
- when that is all done, call cons to add the first already flattened parts to the newly flattened end parts

```
; i append the first one to the rest, then start all over from the beginning
(define (flatten x)
  (cond ((null? x) '())
        ((list? (car x)) (flatten (append (car x) (cdr x))))
        (else (cons (car x) (flatten (cdr x))))))

; mr paley way
; the difference is that instead of append and then flatten, you flatten both the car and cdr and then append

; mr paley flatten the first part, then the butfirst, and then append both
```

```
(define (flatten2 lyst)
  (cond ((null? lyst) '())
        ((list? (car lyst)) (append (flatten2 (car lyst))
(flatten2 (cdr lyst))))
        (else (cons (car lyst) (flatten2 (cdr lyst))))))
```

17.14

Write a procedure branch that takes as arguments a list of numbers and a nested list structure. It should be the list-of-lists equivalent of item, like this:

```
> (branch '(3) '((a b) (c d) (e f) (g h)))
(E F)
> (branch '(3 2) '((a b) (c d) (e f) (g h)))
F
> (branch '(2 3 1 2) '((a b) ((c d) (e f) ((g h) (i j)) k) (l
m)))
H

; in the else, it already evaluates the first one, it then moves
on to the next with cdr

(define (branch num lyst)
  (cond ((null? lyst) '())
        ((null? num) lyst)
        ((= (length num) 1) (list-ref lyst (- (car num) 1)))
        (else (branch (cdr num) (list-ref lyst (- (car num)
1))))))

; traced version for reference

>(branch '(2 3 1 2) '((a b) ((c d) (e f) ((g h) (i j)) k) (l
m)))
>(branch '(3 1 2) '((c d) (e f) ((g h) (i j)) k))
>(branch '(1 2) '((g h) (i j)))
>(branch '(2) '(g h))
<'h
```