Hannah Zhang

2.17, 2.18, 2.20-2.27

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

2.17

Define a procedure last-pair that returns the list that contains only the last element of a given (nonempty) list

```
; check if the list is empty and if so, return the last element
; if not empty, keep checking until it is empty
(define (last-pair lyst)
  (if (empty? (cdr lyst))
      lyst
      (last-pair (cdr lyst))))
```

2.18

Define a procedure reverse that takes a list as an argument and returns a list of the same elements in reverse order

```
; recursion at the front, moving from right to left
(define (reverse lyst)
  (if (empty? lyst)
      '()
      (append (reverse (cdr lyst)) (list (car lyst)))))

; this is what I wrote first
(define (reverse1 lyst)
  (if (empty? lyst)
      '()
      (list (reverse1 (cdr lyst)) (car lyst))))
; which returns
'((((((() 25) 16) 9) 4) 1)
; this is incorrect since when you use list on a list and a number, it creates sub lists
; instead, we can make both of them lists and append to remove sublists
```

```
; iteratively just for fun
(define (reverse2 lystt)
  (define (iter lyst result)
    (if (empty? lyst)
        result
        (iter (cdr lyst) (append (list (car lyst)) result))))
  (iter lystt '()))
```

2.20

Write a procedure same-parity that takes one or more integers and returns a list of all the arguments that have the same even-odd parity as the first argument

```
; takes two or more arguments
(define (f x y . z) x)
; takes zero or more argument
(define (g . w) w)
; in this case, same-parity takes one or more arguments, therefore the input is x . y
; note: using filter makes it much easier
; filter takes a predicate and a list, returns elements of the list which the predicate is true
; adds the first input (x) to the filtered other arguments (y)
(define (same-parity x . y)
  (if (odd? x)
      (cons x (filter odd? y))
      (cons x (filter even? y))))
```

2.21

The procedure square-list takes a list of numbers as argument and returns a list of the squares of those numbers. Complete both of them by filling in the missing expressions.

```
; this version involves cons each term to the rest using recursion
(define (square-list1 items)
  (if (null? items)
      null
      (cons (* (car items) (car items)) (square-list1 (cdr items)))))
; this uses map to apply a function to every argument in a given
```

```
(define (square-list2 items)
  (map (lambda (x) (* x x)) items))
```

2.22

```
; why doesn't this work?
(define (square-list items)
  (define (iter things answer)
    (if (null? things)
        answer
        (iter (cdr things)
              (cons (square (car things))
                    answer))))
  (iter items nil))
```

It returns everything in reverse order because while doing recursion, she cons the newly squared term to the left side of the existing list which makes it backwards.

```
; why doesn't this work either?
(define (square-list items)
  (define (iter things answer)
    (if (null? things)
        answer
        (iter (cdr things)
              (cons answer
                    (square (car things))))))
  (iter items nil))
```

This doesn't work because he uses cons on a list and a single number. This creates multiple sublists while the answer calls for just one list.

```
; for fun, write the correct version
; correct the two errors
```

- append two lists, instead of cons a list and a number
- append the already existing answer and add on the newly calculated square as a list

```
(define (square-list5 items)
  (define (iter things answer)
    (if (null? things)
        answer
```

```
        (iter (cdr things) (append answer (list (square (car
things)))))))
  (iter items null))
```

2.23

The procedure for-each is similar to map. Except, it does not return anything. We must call print to see the results. Give an implementation of for-each.

```
; this procedure must return nothing when called
; it only returns something when print is called
(define (for-each1 func lystt)
  (define (iter lyst result)
    (if (null? lyst)
        (void)
        (iter (cdr lyst) (append result (list (func (car
lyst)))))))
  (iter lystt null))
; we did this by using (void) as a return statement once
everything is evaluated.
; when print is used, the function is printed out each time
until the list/input is empty. then it returns void.
; in normal situations, the whole function is evaluated and at
the end it returns what it has computed. however, because the
last line is (void), everything it has calculated is ignored and
void is returned
```

2.24

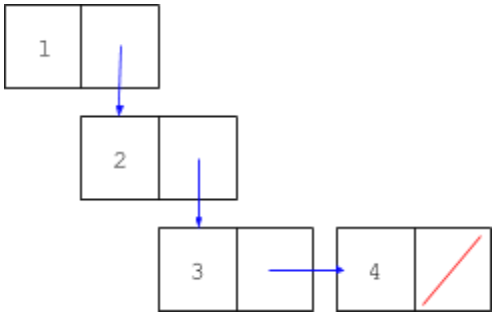
Evaluate (list 1 (list 2 (list 3 4))). Give the result from the interpreter, box-and-pointer structure, and a tree.

Result from interpreter

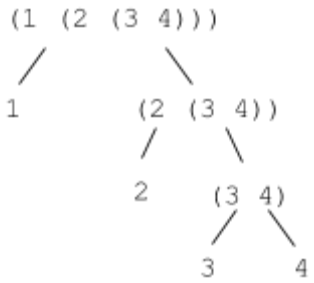
- '(1 (2 (3 4)))

Box-and-pointer structure

- Remember, list makes as many pairs as the num of arguments
  - A list has to end with an empty list
- One pair with 3 and 4 is a result of cons
  - A pair does not end with an empty list
  - Cons always returns one pair
- (list 3 4) is all in one row & last box is empty
- (list 2 (list 3 4) → two rows



Tree



2.25

Give combinations of cars and cdrs that will pick 7 from each of the following lists

```
(1 3 (5 7) 9)
(car (cdr (car (cdr (cdr list)))))

((7))
(car (car list))

(1 (2 (3 (4 (5 (6 7))))))
; cdr returns a list so we must take the car of it before applying cdr again
(car (cdr (car (cdr (car (cdr (car (cdr (car (cdr (car (cdr list))))))))))))
```

2.26

What result is printed by the interpreter

```
(define x (list 1 2 3))
(define y (list 4 5 6))

(append x y)
'(1 2 3 4 5 6)
```

```
(cons x y)
'((1 2 3) 4 5 6)

(list x y)
'((1 2 3) (4 5 6))
```

2.27

Modify your reverse procedure to produce a deep-reverse procedure that takes a list as argument and returns as its value the list with its elements reversed and with all sublists deep-reversed as well.

```
; same function as before except we must call reverse again on the (car lyst) since it is not fully reversed yet
; however, this process soon gets rid of the list and the argument to reverse is wrong
; therefore we must add a second conditional where if the lyst is not a list, then we just return itself
(define (deep-reverse lyst)
  (cond ((empty? lyst) '())
        ((not (list? lyst)) lyst)
        (else (append (deep-reverse (cdr lyst)) (list (deep-reverse (car lyst)))))))

; iteratively-ish
; we must call deep-reverse2 because if we only use iter, we only reverse the first layer. Therefore we use deep-reverse2 to reverse the next layers.
(define (deep-reverse2 lystt)
  (define (iter lyst result)
    (cond ((empty? lyst) result)
          ((not (list? lyst)) lyst)
          (else (iter (cdr lyst) (append (list (deep-reverse2 (car lyst))) result)))))
  (iter lystt '()))
```

Old SCVAL Door Prize Problem

; code in the process

; comments

; answer

; process of thinking

The Three Factors					Triangular Number		Notation	
TFTN#	**1st**	**2nd**	**3rd**					
1	1	2	3	Π=	6	=	T	3
2	**4**	**5**	**6**	Π=	**120**	=	T	**15**
3	**5**	**6**	**7**	Π=	**210**	=	T	**20**
4	**9**	**10**	**11**	Π=	**990**	=	T	**44**
5	**56**	**57**	**58**	Π=	**185136**	=	T	**608**
6	636	637	638	Π=	258474216	=	T	22736

Notes for understanding

- Notation, first of the three factors, triangular num

My idea

- Create a list for all the possible consecutive sums
- Create a list for all the possible products of three nums
- Compare the numbers in each and if it matches, return it

Process of thinking

- Writing the list for product

```
(define (triplet x)
  (* x (+ x 1) (+ x 2)))
```

The input to x should be 0

```
(define (prod1 x)
  (define (iter num result)
    (if (> num 636)
        result
```

```
        (iter (+ num 1) (cons (triplet num) result))))
  (iter x '()))
```

- This returns a backward list, starting with greatest number

- Writing the list for sum

```
The input to x should be 0
(define (sum1 x)
  (define (iter num list-result total-sum)
    (if (= num 22736)
        list-result
        (iter (+ num 1) (cons (+ num total-sum) list-result) (+
num total-sum))))
  (iter x '() 0))
```

- Again, this returns a backward list

- Reversing both the lists

```
; input starting with 636
(define (prod x)
  (define (iter num result)
    (if (= num 0)
        result
        (iter (- num 1) (cons (triplet num) result))))
  (iter x '()))

; input starting with one
(define (sum x)
  (define (iter num list-result total-sum)
    (if (> num 22736)
        list-result
        (iter (+ num 1) (append list-result (list (+ num
total-sum))) (+ num total-sum))))
  (iter x '() 0))
```

- Original idea: use list-ref and then compare numbers

```
(define (list-of-sums x)
  (list-ref (sum 1) x))

(define (list-of-prod x)
  (list-ref (prod 636) x))
```

```
(define (func2 x y)
  (define (iter n1 n2 result)
    (cond ((or (= n1 22735) (= n2 635)) result)
          ((= (list-of-sums n1) (list-of-prod n2))
           (iter (+ n1 1) (+ n2 1) (cons (list-of-prod n2)
result)))
          ((< (list-of-sums n1) (list-of-prod n2))
           (iter (+ 1 n1) n2 result))
          ((> (list-of-sums n1) (list-of-prod n2))
           (iter n1 (+ 1 n2) result))
          (else result)))
  (iter x y '()))
```

- This takes too long

- Instead, use let to define the computed lists, so we do not have to compute it every time
- Both lists advance at the same time to reduce time
- Return number of the list of sum (highest notation), number of the list of prod (first factor), list-ref of prod (triangular number)

```
(define (func)
  (let ((list-sum (sum 1))
        (list-prod (prod 636)))
    (define (iter n1 n2 result)
      (cond ((or (= n1 22736) (= n2 636)) result)
            ((= (list-ref list-sum n1) (list-ref list-prod n2))
             (iter (+ n1 1) (+ n2 1) (append result (list (list
(+ n1 1) (+ n2 1) (list-ref list-prod n2))))))
            ((< (list-ref list-sum n1) (list-ref list-prod n2))
             (iter (+ n1 1) n2 result))
            ((> (list-ref list-sum n1) (list-ref list-prod n2))
             (iter n1 (+ n2 1) result))
            (else result)))
    (iter 0 0 '()))))
```

Returns: '((3 1 6) (15 4 120) (20 5 210) (44 9 990) (608 56 185136) (22736 636 258474216))

```
; takes about 20 seconds
```