Hannah Zhang
1.30 - 1.33

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

1.30
Rewrite the sum procedure to be iterative

```
; recursive
(define (sum term a next b)
  (if (> a b)
      0
      (+ (term a)
         (sum term (next a) next b))))

; trace
(sum square 1 plus-two 5)
> (+ (square 1) (sum square 3 plus-two 5))
> > (+ (square 3) (sum square 5 plus-two 5))
> > > (+ (square 5) (sum square 7 plus-two 5))
> > > 0
> > 0 + 25
> 0 + 25 + 9
0 + 25 + 9 + 1 → 35


; iterative
(define (sum term a next b)
  (define (iter a result)
    (if <??>
        <??>
        (iter <??> <??>)))
  (iter <??> <??>))

; iterative way with nested functions
(define (sum2 term a next b)
```

```
  (define (iter a result)
    ; just like in recursive sum, check if a is greater than b
    ; can use variables previously defined in enclosing
functions
    (if (> a b)
        ; if it is greater, return result
        result
        ; if not, move onto the next term, add a to the result
        (iter (next a) (+ result (term a)))))
  ; call iter where the user enters 4 variables
  ; variables are evaluated in the nested function
  ; result starts at 0 but keeps on adding up until iter is true
  (iter a 0))


; trace
(sum2 square 1 plus-two 5)
> (iter (plus-two 1) (+ 0 (square 1))
< (iter 3 1)
> (iter (plus-two 3) (+ 1 (square 3))
< (iter 5 10)
> (iter (plus-two 5) (+ 10 (square 5))
< (iter 7 35)
35

; iterative way with an invariant quantity (1 function)
(define (sum3 term a next b result)
  (if (> a b)
      result
      (sum3 term (next a) next b (+ result (term a)))))
```

1.31 (a)

Write three procedures analogous to sum. First write product, then write factorial, then pi with the given formula.

```
write the procedure product, analogous to sum
; recursive & higher-order
(define (product term a next b)
  (if (> a b)
      1
      (* (term a) (product term (next a) next b))))

; iterative version of product (just for practice)
; with an invariant quantity
(define (product2 term a next b result)
  (if (> a b)
      result
      (product2 term (next a) next b (* result (term a)))))
; with a nested function
(define (product3 term a next b)
  (define (iter a result)
    (if (> a b)
        result
        (iter (next a) (* result (term a)))))
  (iter a 1))

define factorial in terms of product
; functions that are used in factorial
(define (same x) x)
(define (plus-one x)
  (+ x 1))

(define (factorial x)
  (product same 1 plus-one x))

use product to compute approximations to pi
; make a table and find a pattern
     when x = 1 → 2/3
     when x = 2 → 4/3
```

```
      when x = 3 → 4/5
      when x = 4 → 6/5
; i found out that there is a pattern when x is even or odd
      odd: (x+1)/(x+2)
      even: (x+2)/(x+1)
; define a helper function that applies the pattern above
(define (odd x)
  (= (remainder x 2) 1))

(define (even x)
  (= (remainder x 2) 0))

(define (pi-helper x)
  (if (even x)
      (/ (+ x 2) (+ x 1))
      (/ (+ x 1) (+ x 2))))
; use the function to write pi (use it as term to determine what
to multiply each one by), x is plus-one each time
(define (plus-one x)
  (+ x 1))

; use 1.0 so the final answer is a decimal
(define (pi x)
  (* 4 (product pi-helper 1.0 plus-one x)))
```

1.32 (iterative process only)

Show that sum and product are both special cases of a still more general notion called accumulate

```
; accumulate applies a given function/operator to all the values
(accumulate combiner null-value term a next b)

Accumulate takes as arguments the same term and range
specifications as sum and product, together with a combiner
procedure (of two arguments) that specifies how the current term
is to be combined with the accumulation of the preceding terms
and a null-value that specifies what base value to use when the
terms run out.

; write accumulate as an iterative process
; combiner is applied to each value
; null-value specify what base value to use when term runs out
     (0 with add/subtract, 1 with multiply/divide)
; term, a, next, b, result are the same with sum/product

; with invariant quantity
(define (accumulate1 combiner null-value term a next b result)
  (if (> a b)
      result
      (accumulate1 combiner null-value term (next a) next b
(combiner result (term a)))))

; with nested function
(define (accumulate2 combiner null-value term a next b)
  (define (iter a result)
    (if (> a b)
        result
        (iter (next a) (combiner result (term a)))))
  (iter a null-value))

; recursively (just for fun)
(define (accumulate3 combiner null-value term a next b)
  (if (> a b)
```

```
      null-value
      (combiner (term a) (accumulate3 combiner null-value term
(next a) next b))))

; show how sum and product can be defined as calls to accumulate
; sum
(define (sum term a next b)
  (accumulate1 + 0 term a next b 0))
; product
(define (product term a next b)
  (accumulate1 * 1 term a next b 1))
```

1.33

Write a more general version of accumulate using filter

Use odd numbers instead of primes for part a

Filter is a predicate. If true, accumulate the term; if not, ignore the term and move on

```
; write accumulate using filter
; iterative with invariant quantity
(define (filtered-accumulate predicate combiner null-value term
a next b result)
  (if (predicate a)
      (if (> a b)
          result
          (filtered-accumulate predicate combiner null-value
term (next a) next b (combiner result (term a))))
  (filtered-accumulate predicate combiner null-value term (next
a) next b result)))

; however, the above function is an infinite loop if filter is
#f
; the function below solves this by checking (> a b) before
checking if filter is #t or #f
(define (filtered-accumulate1 predicate combiner null-value term
a next b result)
  (if (> a b)
      result
      (if (predicate a)
          (filtered-accumulate1 predicate combiner null-value
term (next a) next b (combiner result (term a)))
          (filtered-accumulate1 predicate combiner null-value
term (next a) next b result))))

; iterative with nested function
(define (filtered-accumulate2 predicate combiner null-value term
a next b)
  (define (iter a result)
    (if (> a b)
```

```
        result
        (if (predicate a)
            (iter (next a) (combiner result (term a)))
            (iter (next a) result))))
  (iter a null-value))

; write the sum of squares of odd numbers from a to b using
filtered-accumulate
(define (sum-of-squares a b)
  (filtered-accumulate2 odd? + 0 square a plus-one b))

; write the product of all the positive integers less than n
that are relatively prime to n
; assuming that relatively-prime? exists already
(define (relatively-prime a b)
  (filtered-accumulate2 relatively-prime? * 1 same a plus-one
b))

; filtered-accumulate recursively (just for fun)
(define (filtered-accumulate3 predicate combiner null-value term
a next b)
  (if (> a b)
      null-value
      (if (predicate a)
          (combiner (term a) (filtered-accumulate3 predicate
combiner null-value term (next a) next b))
          (filtered-accumulate3 predicate combiner null-value
term (next a) next b))))
```