Hannah Zhang
1.30 - 1.33

; prompt
; comments
; answers
; question

1.34
We define the procedure

```
(define (f g)
  (g 2))
```

Then we have

(f square) → *4*
(f (lambda (z) (* z (+ z 1)))) → *6*

; this procedure takes another procedure as the argument
; if you put square, it will invoke the procedure on 2
; if we put the function z, it will invoke z on 2 where the input z = 2

What happens if we (perversely) ask the interpreter to evaluate the combination (f f)? Explain.

; f takes a procedure as an input but in this case, it is given 2 as an input. Therefore, (f f) can not run since the argument given is not a procedure.

1.37a

Define a procedure cont-frac such that evaluating (cont-frac n d k) computes the value of the *k*-term finite continued fraction.

; infinite continued fraction

For k = 1 → $\frac{1}{1}$ → 1

For k = 2 → $\frac{1}{1+\frac{1}{1}}$ → 1/2

For k = 3 → $\frac{1}{1+\frac{1}{1+\frac{1}{1}}}$ → 2/3

; define a "counter" variable that starts at 1

- If counter is equal to k
  - Terminate the fraction with $\frac{nk}{dk}$
  - K is a set value, counter starts from 1 and increments until it is equal to k
  - Note: n1 means n of counter where n is a function and same with d1
- If counter is not equal to k
  - Continue the recursion and add one to counter
  - $\frac{n1}{d1+\frac{n2}{d2+\frac{n3}{d3}}}$ is given
  - Must find a pattern to continue the recursion
    - $g(3) = \frac{n3}{d3}$
    - $g(2) = \frac{n2}{d2+g(3)}$
    - $g(1) = \frac{n1}{d1+g(2)}$
    - Pattern: $g(x) = \frac{nx}{dx+g(x+1)}$

; using the if/else statements & patterns, write cont-frac

```
(define (cont-frac n d k)
  (define (iter counter)
    (if (= counter k)
        (/ (n k) (d k))
        ; uses counter since it increases every time, is not
always the same
        (/ (n counter) (+ (d counter) (iter (+ counter 1))))))
  (iter 1))
```

; when calling cont-frac, you need to define n and d as functions

```
; in the example given, we use lambda to define n and d while we are calling cont-frac
(cont-frac (lambda (i) 1.0)
           (lambda (i) 1.0)
           k)
; replace k with the amount of times you want recursion to be called
; if you want a different pattern
```

$$\frac{2}{4+\frac{3}{9+\frac{4}{16+\dots}}}$$

```
; change n to i+1 and d to i^2

; writing cont-frac iteratively for practice, use invariant quantity
(define (cont-frac1 n d k)
  (define (iter counter a)
    (if (= counter k)
        a
        (iter (+ counter 1) (/ (n counter) (+ a (d counter))))))
  (iter 1 1))
```

How large must you make k in order to get an approximation that is accurate to 4 decimal places?

- Basically when k plugged in is equal to 1 divided by the golden ratio
- 1/golden ratio = 0.618033
- When k = 11 → 0.618055
- When k = 11, the number is accurate to 4 decimal places of the golden ratio
- Therefore the answer is k = 11

1.38

Write a program that uses your cont-frac procedure to approximate e, based on Euler's expansion.

Sequence: 1 2 1 1 4 1 1 6 1 1 8 1

Pattern: first & third → 1, middle → x * 2

In this case ni always equals 1

di = ???

- d1 = 1
- d2 = 2
- d3 = 1
- d4 = 1
- d5 = 4
- d6 = 1

Pattern: if d+1 is divisible by 3, return ((d+1)/3) * 2; else return 1

Write the function d that will be passed into cont-frac

```
(define (d i)
  (if (= (remainder (+ i 1) 3) 0)
      (* (/ (+ i 1) 3) 2.)
      1))
```

Write n

```
(define (n i) 1)
```

- Call cont-frac with the new functions n and d
- Put a value for k to guess (bigger the guess, more accurate the num)
- The answer is e-2 so add 2 to the whole thing (given by the problem)

```
(define (e k)
  (+ 2 (cont-frac n d k)))
```

and we get

e = 2.718281

1.41

Define a procedure double that takes a procedure of one argument as argument and returns a procedure that applies the original procedure twice.

```
; double takes a procedure
; it could take double or another function
; this function below tries to apply procedure twice
(define (double1 procedure)
  (procedure (procedure)))
; however, procedure does not have any arguments
; we must use lambda

(define (double procedure)
  (lambda (x) (procedure (procedure x))))
; this gives an argument to procedure

(define (inc x)
  (+ x 1))

; (double inc) returns #<procedure>
; somehow we must give an input to (double inc)
; for example if we want to call f, we use (f x)
; same, if we want to call (double inc), we use ((double inc) x)
; give a value to x

((double inc) 5) → 7
((double inc) 6) → 8
; calls inc on x twice
```

What value is returned by
(((double (double double)) inc) 5)

21

; ((double inc) 5) → 7

- this is equal to 2; call in 2 times on 5

; (((double double) inc) 5) → 9

- this is equal to 2^2 which is 4; call inc 4 times on 5

; (((double (double double)) inc) 5) → 21

- This equal to 2^2^2 which is 16; call inc 16 times on 5

1.42

Write compose

```
((compose square inc) 6)
```

49

Invoke f(g(x)) which is equal to (square (inc 6)) in this case

```
(define (inc x)
  (+ x 1))

(define (square x)
  (* x x))

; this is incorrect since we would call (compose square inc 6)
; which is missing a parentheses
; this returns a value
(define (compose1 f g x)
  (f (g x)))

; need to use lambda for x to preserve the parentheses
; this returns a procedure
(define (compose f g)
  (lambda (x) (f (g x))))

; we are trying to return a function, not a value
; e.g. (compose square inc) --> #<procedure>
; if we want a value, we put in a value for x so
; e.g. ((compose square inc) 6) --> 49
```

1.43

((repeated square 2) 5)
625
; square 5 twice → (5 * 5) (5 * 5) or 5^{2^2}

Compose takes the f(g(x))
Repeated takes the function of x, repeated n-times
f(f(f(x)))
f**(f(f(x)))** → bolded part is n - 1

(repeated function n-times)

→ using repeated, it would be

(**(function (repeated function n-times))** x)
; the actual important part is bolded, x is just an input

(function (repeated function (- n-times 1)))

Use n-1 since repeated is called, and then function is already called once so it should be n number of times minus the one time it has already been called

→ using compose, it would be
(compose function (repeated function (- n-times 1)))

Use this pattern to write the function

; use recursion to solve this → must use a base case
; must return a procedure = use lambda or directly return

```
; returns the procedure
(define (repeated function n-times)
 (if (= n-times 1)
     function
     (compose function (repeated function (- n-times 1)))))
```

Or

```
; uses lambda to return a procedure
(define (repeated3 function n-times)
 (if (> n-times 0)
     (compose function (repeated3 function (- n-times 1)))
     (lambda (x) x)))
```

Notes to help me understand this problem

; can use (lambda (x) x)) since we want to return a procedure
; "function" is a procedure
; when you do not evaluate "function", do not need parentheses
; instead it just returns the procedure "f" without evaluation
; if you add parentheses, it evaluates and returns the result of the evaluation
; in conclusion, this function repeated returns another function
; "function" is the input given

; ((repeated square 2) 5)
; "function" is square in this case
; n-times is 2
; 5 is the argument
; since repeated returns a procedure, we must give an argument for it to evaluate
; therefore it will not return a procedure, it will return a value