SECTION 1

RECURSION

in recursion we just call the function in on itself till a condition is met, this results in the program firstly processing *ALL* the data and then computing one by one

```
(define (factorial n)
    (if (= n 1)
        1
        (* n (factorial (- n 1)))))

;(factorial 6)
;(* 6 (factorial 5))
;(* 6 (* 5 (factorial 4)))
;(* 6 (* 5 (* 4 (factorial 3))))
;(* 6 (* 5 (* 4 (* 3 (factorial 2)))))
;(* 6 (* 5 (* 4 (* 3 (* 2 (factorial 1)))))) <-- accumulates all the processes
;(* 6 (* 5 (* 4 (* 3 (* 2 1)))))
;(* 6 (* 5 (* 4 (* 3 2))))
;(* 6 (* 5 (* 4 6)))
;(* 6 (* 5 24))
;(* 6 120)
;720
```

ITERATION

Iteration means going forward as opposed to the recursion's going backwards at first, to the very lowest level, and then going forward calculating the result on the way back up, you *change* the value of the variable at every step:

```
(define (factorial n)
  (define (iter product counter)
    (if (> counter n)
        product
        (iter (* counter product) (+ counter 1))))
  (iter 1 1))

;(fact 4)
;(iter 1  1  4)
;(iter 1 2 4)
;(iter 2 3 4)
;(iter 6 4 4)
;(iter 24 5 4)
;24
```

Product	Input	Counter
1	4	1
1	4	2
2	4	3
6	4	4
24	4	5

DIFF BETWEEN THEM

One of the primary ways to differentiate between an iterative and recursive process is to imagine what'd happen if you turned the computer off, then resumed the current computation.
In a recursive process we can never know how deep into the recursion we are.
In the factorial example, this means that we can't return the right value.
In an iterative process, all state information is contained by the process.

Exercise 1

What is the result printed by the interpreter in response to each expression?

```
10
;10
(+ 5 3 4)
;12
(- 9 1)
;8
(/ 6 2)
;3
(+ (* 2 4) (- 4 6))
;6

(define a 3)
(define b (+ a 1))
(+ a b (* a b))
;(+ 3 (+ 3 1) (* 3 (+ 3 1)))
;(+ 3 4 12)
;(19)

(= a b)
;FALSE (amogus)
(if (and (> b a) (< b (* a b)))
b
a)
;(if (and true true) b a)
;(if true b a)
;(b)
;4

(cond ((= a 4) 6)
      ((= b 4) (+ 6 7 a))
      (else 25))
;(+ 6 7 3)
;16
```

```
(+ 2 (if (> b a) b a))
;(+ 2 (if true b a))
;(+ 2 4)
;(+ 6)

(* (cond ((> a b) a)
     ((< a b) b)
     (else -1))
   (+ a 1))
;(* 4 (+ 3 1))
;(* 4 4)
;16
```

Exercise 2

Translate the following expression into prefix

$$\frac{5+4+(2-(3-(6+\frac{4}{5})))}{3(6-2)(2-7)}.$$

```
(/ (+ 5 4 (- 2 (- 3 (+ 6 (/ 4 5))))) (* 3 (- 6 2) (- 2 7)))
; -37/150
```

Exercise 3

Define a procedure that takes three numbers as arguments and returns the sum of the squares of the two larger numbers

```
(define (f1 a b c)
  (if (< a b)
      (if (< a c)
          (+ (* b b) (* c c))
          (+ (* b b) (* a a)))
      (if (< b c)
          (+ (* a a) (* c c))
          (+ (* a a) (* b b)))))
```

Exercise 4

Observe that our model of evaluation allows for combinations whose operators are compound expres- sions. Use this observation to describe the behavior of the following procedure:

```
(define (a-plus-abs-b a b)
  ((if (> b 0) + -) a b))
```

```
;this is just a+|b| by performing a+b if b>0, else a-b
(a-plus-abs-b 3 4)
;(7)
(a-plus-abs-b 3 -4)
```

Exercise 5

Ben Bitdiddle has invented a test to determine whether the interpreter he is faced with is using applicative-order evaluation or normal-order evaluation. He defines the following two procedures:

```
(define (p) (p))
(define (test x y) (if (= x 0) 0 y))
```

Then he evaluates the expression `(test 0 (p))` What behavior will Ben observe with an interpreter that uses applicative-order evaluation? What behavior will he observe with an interpreter that uses normal-order evaluation? Explain your answer. (Assume that the evaluation rule for the special form if is the same whether the interpreter is using normal or applicative order: The predicate expression is evaluated first, and the result determines whether to evaluate the consequent or the alternative expression.

evaluating (p) will result in an infinite loop for applicative order because internal reduction happens **BEFORE** the values are evaluated, like:

```
(test 0 (p))
(if (= 0 0) 0 (p)) ;<---- here the program will go for an inifnite loop
because it can't evaluate `(p)`
```

for normal order eval this will happen:

```
(test 0 (p))
(if (= 0 0)
    0       ;<--- here the program will stop
    (p))    ;<--- the program will actually never reach this phase
```

Exercise 6

Alyssa P. Hacker doesn't see why if needs to be provided as a special form. "Why can't I just define it as an ordinary procedure in terms of cond?" she asks. Alyssa's friend Eva Lu Ator claims this can indeed be done, and she defines a new version of if:

```
(define (new-if predicate then-clause else-clause)
  (cond (predicate then-clause)
        (else else-clause)))
```

Eva demonstrates the program for Alyssa:

```
(new-if (= 2 3) 0 5)
5

(new-if (= 1 1) 0 5)
0
```

Delighted, Alyssa uses new-if to rewrite the square-root program:

```
(define (sqrt-iter guess x)
  (new-if (good-enough? guess x)
          guess
          (sqrt-iter (improve guess x) x)))
```

What happens when Alyssa attempts to use this to compute square roots? Explain.

in the old `(if cond A B)` the B condition would only be evaluated if cond was false BUT in the new `(new-if cond A B)` as per the definition the arguement B needs to be evaluated regardless of if the cond was false because when a function is called the first thing that Scheme does with the argument list is evaluating every single arguement

do you remembah? the scheme uses applicative-order for function in septembah?

Exercise 7

The good-enough? test used in computing square roots will not be very effective for finding the square roots of very small numbers. Also, in real computers, arithmetic operations are almost always performed with limited precision. This makes our test inadequate for very large numbers. Explain these statements, with examples showing how the test fails for small and large numbers. An alternative strategy for implementing good-enough? is to watch how guess changes from one iteration to the next and to stop when the change is a very small fraction of the guess. Design a square-root procedure that uses this kind of end test. Does this work better for small and large numbers?

if y is our guess for $\sqrt{x}$ then, $y = \sqrt{x} + \delta y$ and $|y^2 - x| > \varepsilon$, then for small values:

$$\varepsilon = |y^2 - x|$$
$$\varepsilon = |\left(\sqrt{x} + \delta y\right)^2 - x|$$
$$\varepsilon = |(x + 2\delta y)\sqrt{x} - x|$$
$$\varepsilon = 2|\delta y|\sqrt{x}$$

```
(define (average x y) (/ (+ x y) 2))

(define (improve guess x) (average guess (/ x guess)))

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

(define (abs x)
  (cond ((= x 0) 0)
        ((< x 0) (-x)
        ((> x 0) (x)))))

(define (good-enough? guess x)
(< (abs (- (square guess) x)) 0.001))

(define (sqrt-iter guess x)
  (if (good-enough? guess x) guess
      (sqrt-iter (improve guess x) x)))

(define (my-sqrt x)
  (sqrt-iter 1.0 x))

; Given the last and 2nd last guesses, check if the fractional change is good
enough.
; I'm choosing the formula |guessL-guessLL|/guessL < 0.001
(define (good-enough2? guessL guessLL)
  (< (/ (abs (- guessL guessLL)) guessL) 0.01))

(define (sqrt-iter2 guessL guessLL x)
  (if (good-enough2? guessL guessLL)
      guessL
      (sqrt-iter2 (improve guessL x) guessL x)))

(define (my-sqrt2 x)
  (sqrt-iter2 1.0 1.1 x))
```

Exercise 8

Newton's method for cube roots is based on the fact that if y is an approximation to the cube root of x, then a better approximation is given by the value

$(x/y^2 + 2y)/3$

Use this formula to implement a cube-root procedure analogous to the square-root procedure.

```
(define (improve y x)
  (/ (+ (/ x (* y y)) (* 2 y)) 3))

; Fractional good-enough like in ex 1-7
(define (good-enough? guessL guessLL)
  (< (/ (abs (- guessL guessLL)) guessL) 0.001))

(define (curt-iter guessL guessLL x)
  (if (good-enough? guessL guessLL)
      guessL
      (curt-iter (improve guessL x) guessL x)))

(define (my-curt x)
  (curt-iter 1.0 1.1 x))

; 1.4422495703074083
(my-curt 3)
```

Exercise 9

Each of the following two procedures defines a method for adding two positive integers in terms of the procedures inc, which increments its argument by 1, and dec, which decrements its argument by 1.

```
(define (+ a b)
  (if (= a 0)
      b
      (inc (+ (dec a) b))))

(define (+ a b)
  (if (= a 0)
      b
      (+ (dec a) (inc b))))
```

Using the substitution model, illustrate the process generated by each procedure in evaluating `(+ 4 5)`. Are these processes iterative or recursive?

```
(define (+ a b)
  (if (= a 0)
      b
      (inc (+ (dec a) b))))
;recursive process because it doesn't change the value of the variable
;(+ 4 5)
;(inc (+ 3 5))
;(inc (inc (+ 2 5)))
;(inc (inc (inc (+ 1 5))))
;(inc (inc (inc (inc (+ 0 5)))))
;(inc (inc (inc (inc 5))))
;(inc (inc (inc 6)))
;(inc (inc 7))
;(inc 8)
;9

(define (+ a b)
  (if (= a 0)
      b
      (+ (dec a) (inc b))))
;iterative because it changes the value of the variable at every step
;(+ 4 5)
;(+ 3 6)
;(+ 2 7)
;(+ 1 8)
;(+ 0 9)
;9
```

Exercise 10

The following procedure computes a mathematical function called Ackermann's function.

```
(define (A x y)
  (cond ((= y 0) 0)
        ((= x 0) (* 2 y))
        ((= y 1) 2)
        (else (A (- x 1)
                 (A x (- y 1))))))
```

What are the values of the following expressions?

```
(A 1 10)
(A 2 4)
(A 3 3)
```

Consider the following procedures, where A is the procedure defined above:

```
(define (f n) (A 0 n))
(define (g n) (A 1 n))
(define (h n) (A 2 n))
(define (k n) (* 5 n n))
```

Give concise mathematical definitions for the functions computed by the procedures f, g, and h for positive integer values of n. For example, `(k n)` computes $5n^2$.

```
(A 1 10)
;1024
(A 2 4)
;65536
(A 3 3)
;65536

(define (f n) (A 0 n))
;this is just x 0 always so it is just a function that doubles the arguement
i.e n*2
(define (g n) (A 1 n))
;(A 1 0) is 0
;(A 1 1) is 2
;(A 1 2) is (* 2 2) = 4
;(A 1 3) is (* 2 (* 2 2)) = 8
;(A 1 n) is 2^n
(define (h n) (A 2 n))
;(A 2 0) is 0
;(A 2 1) is 2
;(A 2 2) is 4
;(A 2 3) is 16
;this is (A 1 (A 2 (- n 1)))
;or square of (A 2 (- n 1))
;so this is just square of square of square of square and so on
```

Exercise 11

A function f is defined by the rule that $f(n) = n$ if $n < 3$ and $f(n) = f(n-1) + 2f(n-2) + 3f(n-3)$ if $n \geq 3$. Write a procedure that computes f by means of a recursive process. Write a procedure that computes f by means of an iterative process.

recursive is pretty easy

```
(define (f n)
  (if (< n 3)
      n
      (+ (f (- n 1)) (* (f (- n 2)) 2) (* (f (- n 3)) 3))))
```

for iterative. We need three pieces of state to track the current, the last and the values of f. (that is, $f(n-1)$, $f(n-2)$ and $f(n-3)$.)

```
(define (f n)
  (f-it-helper 2 1 0 n))

(define (iterator a b c counter)
  (if (= counter 0)
      c
      (iterator (+ a (* 2 b) (* 3 c))
                a
                b
                (- counter 1))))
;think like this
;f(0) = 0  |
;f(1) = 1  | all known values
;f(2) = 2  |

;f(3) = f(2) + 2f(1) + 3f(0)
;f(4) = f(3) + 2f(2) + 3f(1)
;f(5) = f(4) + 2f(3) + 3f(2)
;f(6) = f(5) + 2f(4) + 3f(3)
```

Exercise 12

Write a recursive procedure that computes elements of Pascal's triangle.

```
(define (pascal row col)
  (cond ((= col 0) 1)
        ((= col row) 1)
        (else (+ (pascal (- row 1) col) (pascal (- row 1) (- col 1))))))
;idk how to display it :P
```

Exercise 13

Prove that $\text{fib}(n)$ is the closest integer to $\varphi^n/\sqrt{5}$ where $\varphi = \left(1 + \sqrt{5}\right)/2$.

Hint: Let $\psi = \left(1\sqrt{5}\right)/2$. Use induction and the definition of the Fibonacci numbers to prove that $\text{Fib}(n) = (\varphi^n - \psi^n)/\sqrt{5}$.

TODO

Exercise 14

Draw the tree illustrating the process generated by the count-change procedure of 1.2.2 in making change for 11 cents. What are the orders of growth of the space and number of steps used by this process as the amount to be changed increases?

```
;the code was this
(define (count-change amount)
   (cc amount 5))
(define (cc amount kinds-of-coins)
   (cond ((= amount 0) 1)
         ((or (< amount 0) (= kinds-of-coins 0)) 0)
         (else (+ (cc amount
                      (- kinds-of-coins 1))
                  (cc (- amount
                         (first-denomination kinds-of-coins))
                      kinds-of-coins)))))
(define (first-denomination kinds-of-coins)
   (cond ((= kinds-of-coins 1) 1)
         ((= kinds-of-coins 2) 5)
         ((= kinds-of-coins 3) 10)
         ((= kinds-of-coins 4) 25)
         ((= kinds-of-coins 5) 50)))
(count-change 11)
```

$$\sum_{i=a}^{b} \frac{1}{i(i+2)}$$

Exercise 15

The sine of an angle (specified in radians) can be computed by making use of the approximation $\sin(x) = x$ if x is sufficiently small, and the trigonometric identity to reduce the size of the argument of sin. (For purposes of this exercise an angle is considered "sufficiently small" if its magnitude is not greater than 0.1 radians.) These ideas are incorporated in the following procedures:
we are using the identity $\sin(3x) = 3\sin(x) - 4\sin^3(x)$

```
(define (cube x) (* x x x))
(define (p x)
  (- (* 3 x) (* 4 (cube x))))
(define (sine angle)
  (if (not (> (abs angle) 0.1))
      angle
      (p (sine (/ angle 3.0)))))
```

- How many times is the procedure p applied when (sine 12.15) is evaluated?
- What is the order of growth in space and number of steps (as a function ofa) used by the process generated by the sine procedure when (sine a) is evaluated?

- the procedure p is evaluated 5 times

$$\frac{12.15}{3^n} = 0.1$$

$$n = \frac{\log\left(\frac{12.5}{0.1}\right)}{\log(3)} \approx 4.4$$

```
;also the output is this
(sine 12.5)
-0.060813768577286265
```

- the order of growth of steps is just $\Theta(\log(a))$ and similar in space because there is no recursion

Exercise 16

Design a procedure that evolves an iterative exponentiation process that uses successive squaring and uses a logarithmic number of steps, as does fast-expt.

(Hint: Using the observation that $b^{(n/2)^2} = \left(b^2\right)^{n/2}$ keep, along with the exponent n and the base b, an additional state variable a, and define the state transformation in such a way that the product ab^n is unchanged from state to state. At the beginning of the process a is taken to be 1, and the answer is given by the value of a at the end of the process. In general, the technique of defining an invariant quantity that remains unchanged from state to state is a powerful way to think about the design of iterative algorithms.)

```
(define (exponent base n)
  (helperiter 1 base n))
(define (helperiter acc base n)
  (cond ((= n 0) acc)
        ((= (remainder n 2) 0) (helperiter acc (* base base) (/ n 2))) ;if the
exponent is a multiple of 2 then do b^n = (b*b)^(n/2)
        (helperiter (* acc base) base (- n 1))))
```

Exercise 17

The exponentiation algorithms in this section are based on performing exponentiation by means of repeated multiplication. In a similar way, one can perform integer multiplication by means of repeated addition. The following multiplication procedure (in which it is assumed that our language can only add, not multiply) is analogous to the expt procedure:

```
(define (* a b)
  (if (= b 0) 0
      (+ a (* a (- b 1)))))
```

this algorithm takes a number of steps that is linear in b. Now suppose we include, together with addition, operations double, which doubles an integer, and halve, which divides an (even) integer by 2. Using these, design a multiplication procedure analogous to fast-expt that uses a logarithmic number of steps.

we can do something like $ab = 2ab/2$ for even b (because we are given the procedures *double* and *half*)
and the previous same procedure for odd b

```
(define (even? n)
  (= (remainder n 2) 0))

(define (* a b)
  (cond ((= b 0) 0)
        ((even? b) (* (double a) (half 2)))
        (else (+ a (* a (- b 1))))))
```

Exercise 18

Using the results of Exercise 1.16 and Exercise 1.17, devise a procedure that generates an iterative process for multiplying two integers in terms of adding, doubling, and halving and uses a logarithmic number of steps.

we know that in iterative processes we need an accumulator a counter and some number of other arguements for operations, so let's make one of the multiplication arguements as the counter, something like

```
(define (multiply a b)
  (define (iterator acc a b)
    (cond ((= b 0) acc)
          ((even? b) (iterator acc (double a) (half b)))
          (else (iterator (+ acc a) a (- b 1)))))
  (iterator 0 a b))
```

Exercise 19

There is a clever algorithm for computing the Fibonacci numbers in a logarithmic number of steps. Recall the transformation of the state variables a and b in the fib-iter process of 1.2.2:

$$a \leftarrow a + b$$

$$b \leftarrow a$$

Call this transformation T, and observe that applying T over and over again n times, starting with 1 and 0, produces the pair $\text{Fib}(n+1)$ and $\text{Fib}(n)$. In other words, the Fibonacci numbers are produced by applying T^n, the n^{th} power of the transformation T, starting with the pair $(1, 0)$.

Now consider T to be the special case of $p = 0$ and $q = 1$ in a family of transformations T_{pq} where T_{pq} transforms the pair (a, b) according to:

$$a \leftarrow bq + aq + ap$$

$$b \leftarrow bp + aq$$

Show that if we apply such a transformation T_{pq} twice, the effect is the same as using a single transformation $T_{p'q'}$ of the same form, and compute p' and q' in terms of p and q. This gives us an explicit way to square these transformations, and thus we can compute T^n using successive squaring.

```
(define (fib n)
  (fib-iter 1 0 0 1 n))
(define (fib-iter a b p q count)
  (cond ((= count 0) b)
    ((even? count)
     (fib-iter a
               b
               ⟨??⟩ ; compute p′
               ⟨??⟩ ; compute q′
               (/ count 2)))
    (else (fib-iter (+ (* b q) (* a q) (* a p))
                    (+ (* b p) (* a q))
                    p
                    q
                    (- count 1)))))
```

let us say there is a matrix T_{pq}, then according to the question this matrix *transforms* the matrix (in question called pair) $\begin{pmatrix} a \\ b \end{pmatrix}$ as follows:

$$T_{pq} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} bq + aq + ap \\ bp + aq \end{bmatrix}$$

$$T_{pq} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} a(p+q) + bq \\ aq + bp \end{bmatrix}$$

$$T_{pq} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} a \\ b \end{bmatrix} \begin{bmatrix} p+q & q \\ q & p \end{bmatrix}$$

$$T_{pq} = \begin{bmatrix} p+q & q \\ q & p \end{bmatrix}$$

and according to question we need to show that T_{pq}^2 (T_{pq} used 2 times) is equivalent to some $T_{p'q'}$, so:

$$T_{pq}^2 = \begin{bmatrix} p+q & q \\ q & p \end{bmatrix}^2$$

$$T_{pq}^2 = \begin{bmatrix} p^2 + 2pq + 2q^2 & 2pq + q^2 \\ 2pq + q^2 & q^2 + p^2 \end{bmatrix}$$

$$T_{pq}^2 = T_{(2pq+q^2)(q^2+p^2)}$$

now just shove it in the code and *noice*

```
(define (fib n)
  (fib-iter 1 0 0 1 n))
(define (fib-iter a b p q count)
  (cond ((= count 0) b)
    ((even? count)
     (fib-iter a
               b
               (+ (* 2 p q) (* q q)) ; compute p'
               (+ (* q q) (* p p)) ; compute q'
               (/ count 2)))
    (else (fib-iter (+ (* b q) (* a q) (* a p))
                    (+ (* b p) (* a q))
                    p
                    q
                    (- count 1)))))
```

Exercise 20

> The process that a procedure generates is of course dependent on the rules used by the interpreter. As an example, consider the iterative gcd procedure given above. Suppose we were to interpret this procedure using normal-order evaluation, as discussed in 1.1.5. (The normal-order-evaluation rule for if is described in Exercise 1.5.) Using the substitution method (for normal order), illustrate the process generated in evaluating (gcd 206 40) and indicate the remainder operations that are actually performed. How many remainder operations are actually performed in the normal-order evaluation of (gcd 206 40)? In the applicative-order evaluation?

- for normal order evaluation it will be

```
;i AM NOT WRITING THAT WTF it will take ages bro
```

- for applicative order of evaluation

```
; ok fuck this i saw everywhere and everyone has a different answer? wtf?
```

Exercise 21

> Use the smallest-divisor procedure to find the smallest divisor of each of the following numbers: 199, 1999, 19999.

```
(define (smallest-divisor n) (find-divisor n 2))

(define (divides? a b)
  (= (remainder b a) 0))

(define (find-divisor n test-divisor)
  (cond ((> (* test-divisor test-divisor) n) n)
        ((divides? test-divisor n) test-divisor)
        (else (find-divisor n (+ test-divisor 1)))))

(smallest-divisor 199)
;199
(smallest-divisor 1999)
;;1999
(smallest-divisor 7)
;7
```

Exercise 22

Most Lisp implementations include a primitive called runtime that returns an integer that specifies the amount of time the system has been running (measured, for example, in microseconds). The following timed-prime-test procedure, when called with an integer prints and checks to see if is prime. If is prime, the procedure prints three asterisks followed by the amount of time used in performing the test.

```
(define (timed-prime-test n)
  (newline)
  (display n)
  (start-prime-test n (runtime)))

(define (start-prime-test n start-time)
  (if (prime? n)
      (report-prime (- (runtime)
                       start-time))))
(define (report-prime elapsed-time)
  (display " *** ")
  (display elapsed-time))
```

Using this procedure, write a procedure search-for-primes that checks the primality of consecutive odd integers in a specified range. Use your procedure to find the three smallest primes larger than 1000; larger than 10,000; larger than 100,000; larger than 1,000,000. Note the time needed to test each prime. Since the testing algorithm has order of growth of $\Theta(\sqrt{n})$ you should expect that testing for primes around 10,000 should take about $\sqrt{10}$ times as long as testing for primes around 1000. Do your timing data bear this out? How well do the data for 100,000 and 1,000,000 support the $\Theta(\sqrt{n})$ prediction? Is your result compatible with the notion that programs on your machine run in time proportional to the number of steps required for the computation?

there actually isn't a runtime primitive in racket *(because i couldn't get scheme to work on my machine)*, BUT there is *current-milliseconds* which i *think* does the same stuff?

```
(define (smallest-divisor n)
  (find-divisor n 2))

(define (find-divisor n test-divisor)
  (cond ((> (* test-divisor test-divisor) n) n)
        ((divides? test-divisor n) test-divisor)
        (else (find-divisor n (+ test-divisor 1)))))

(define (divides? a b)
  (= (remainder b a) 0))

(define (prime? n)
  (= n (smallest-divisor n)))
```

```
(define (timed-prime-test n start-time)
  (cond ((prime? n)

         (display n)
         (display "is a prime | found in ")
         (display (- (current-milliseconds) start-time))
         (newline)
         true)
        (else false)))

(define (primes-larger-than n x)
  (cond ((= x 0) false)
        (else (cond ((timed-prime-test n (current-milliseconds)) (primes-
larger-than (+ n 1) (- x 1)))
                    (else (primes-larger-than (+ n 1) x))))))

(primes-larger-than 100000000 3)
;100000007 is a prime | found in 0
;100000037 is a prime | found in 0
;100000039 is a prime | found in 0
;#f

(primes-larger-than 1000000000 3)
;1000000007 is a prime | found in 0
;1000000009 is a prime | found in 0
;1000000021 is a prime | found in 0
;#f

(primes-larger-than 10000000000 3)
;10000000019 is a prime | found in 0
;10000000033 is a prime | found in 0
;10000000061 is a prime | found in 0
;#f
```

ummmmm so this is awkward, uhhhhhhh my machine is *too powerful* for the current-milliseconds to be counted so it just returns 0 for everytime welp……fuck

Exercise 23

The smallest-divisor procedure shown at the start of this section does lots of needless testing: After it checks to see if the number is divisible by 2 there is no point in checking to see if it is divisible by any larger even numbers. This suggests that the values used for test-divisor should not be 2, 3, 4, 5, 6, ..., but rather 2, 3, 5, 7, 9, ... To implement this change, define a procedure next that returns 3 if its input is equal to 2 and otherwise returns its input plus 2. Modify the smallest-divisor procedure to use `(next test-divisor)` instead of `(+ test-divisor 1)`. With timed-prime-test incorporating this modified version of smallest-divisor, run the test for each of the 12 primes found in Exercise 1.22. Since this modification halves the number of test steps, you should expect it to run about twice as fast. Is this expectation confirmed? If not, what is the observed ratio of the speeds of the two algorithms, and how do you explain the fact that it is different from 2?

```
#lang racket

(define (smallest-divisor n)
  (find-divisor n 2))

(define (find-divisor n test-divisor)
  (cond ((> (* test-divisor test-divisor) n) n)
        ((divides? test-divisor n) test-divisor)
        (else (find-divisor n (nextdiv test-divisor)))))

(define (divides? a b)
  (= (remainder b a) 0))

(define (prime? n)
  (= n (smallest-divisor n)))

(define (nextdiv test-divisor)
  (cond ((= test-divisor 2) 3)
        (else (+ test-divisor 2))))

(define (timed-prime-test n start-time)
  (cond ((prime? n)
         (display n)
         (display "is a prime | found in ")
         (display (- (current-milliseconds) start-time))
         (newline)
         true)
        (else false)))

(define (primes-larger-than n x)
  (cond ((= x 0) false)
        (else (cond ((timed-prime-test n (current-milliseconds))
                     (primes-larger-than (+ n 1) (- x 1)))
                    (else (primes-larger-than (+ n 1) x))))))
```

```
(primes-larger-than 100000000 3)
;100000007 is a prime | found in 0
;100000037 is a prime | found in 0
;100000039 is a prime | found in 0
;#f

(primes-larger-than 1000000000 3)
;1000000007 is a prime | found in 0
;1000000009 is a prime | found in 0
;1000000021 is a prime | found in 0
;#f

(primes-larger-than 10000000000 3)
;10000000019 is a prime | found in 0
;10000000033 is a prime | found in 0
;10000000061 is a prime | found in 0
;#f
```

uhhhhhh yeah same thing :P

Exercise 24

Modify the timed-prime-test procedure of Exercise 1.22 to use fast-prime? (the Fermat method), and test each of the 12 primes you found in that exercise. Since the Fermat test has $\Theta(\log(n))$ growth, how would you expect the time to test primes near 1,000,000 to compare with the time needed to test primes near 1000? Do your data bear this out? Can you explain any discrepancy you find?

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

(define (expmod base exp m)
  (cond ((= exp 0) 1)
        ((even? exp)
         (remainder (square (expmod base (/ exp 2) m)) m))
        (else
          (remainder (* base (expmod base (- exp 1) m)) m))))

(define (fermat-test n)
  (define (try-it a)
    (= (expmod a n n) a))
  (try-it (+ 1 (random (- n 1)))))

(define (fast-prime? n times)
  (cond ((= times 0) true)
        ((fermat-test n) (fast-prime? n (- times 1)))
```

```
        (else false)))

(define (timed-prime-test n start-time)
  (cond ((fast-prime? n 12)
         (display n)
         (display "is a prime | found in ")
         (display (- (current-milliseconds) start-time))
         (newline)
         #t)
        (else #f)))

(define (primes-larger-than n x)
  (cond ((= x 0) false)
        (else (cond ((timed-prime-test n (current-milliseconds))
                     (primes-larger-than (+ n 1) (- x 1)))
                    (else (primes-larger-than (+ n 1) x))))))

(primes-larger-than 100000 3)
(display "\n\n")
(primes-larger-than 1000 3)

;1000003is a prime | found in 180
;1000033is a prime | found in 186
;1000037is a prime | found in 186
;#f
;
;
;1009is a prime | found in 0
;1013is a prime | found in 0
;1019is a prime | found in 0
;#f
```

hory shit, racket is so slow that it actually gave me a time

Exercise 25

Alyssa P. Hacker complains that we went to a lot of extra work in writing expmod. After all, she says, since we already know how to compute exponentials, we could have simply written

```
(define (expmod base exp m)
  (remainder (fast-expt base exp) m))
```

Is she correct? Would this procedure serve as well for our fast prime tester? Explain.

no clue about time complexity but this sht will not fly for space for sure, cuz fast-expt makes recursive calls to itself and so does remainder, which means the remainder will be called for EACH call to fast-expt (atleast the way we have defined it till now)

Exercise 26

Louis Reasoner is having great difficulty doing Exercise 1.24. His fast-prime? test seems to run more slowly than his `prime?` test. Louis calls his friend Eva Lu Ator over to help. When they examine Louis's code, they find that he has rewritten the expmod procedure to use an explicit multiplication, rather than calling square:

```
(define (expmod base exp m)
  (cond ((= exp 0) 1)
        ((even? exp)
         (remainder
          (* (expmod base (/ exp 2) m) (expmod base (/ exp 2) m))
          m))
        (else
         (remainder
          (* base
             (expmod base (- exp 1) m))
          m))))
```

"I don't see what difference that could make," says Louis. "I do." says Eva. "By writing the procedure like that, you have transformed the $\Theta(\log n)$ process into a $\Theta(n)$ process." Explain.

if we use `(square (expmod base (/ exp 2) m))` then firstly we would've done then (expmod base (/ exp 2) m) and then (square <result> <result>) so we would've evaluated (expmod) ONCE every cycle

but if we do `(* (expmod base (/ exp 2) m) (expmod base (/ exp 2) m))` then we need to calculate (expmod) TWICE which means that for every

Exercise 27

Demonstrate that the Carmichael numbers listed in Footnote 47 really do fool the Fermat test. That is, write a procedure that takes an integer n and tests whether a^n is congruent to a modulo n for every $a < n$, and try your procedure on the given Carmichael numbers.

Numbers that fool the Fermat test are called Carmichael numbers, and li̇̈le is known about them other than that they are extremely rare. ǥere are 255 Carmichael numbers below 100,000,000. ǥe smallest few are 561, 1105, 1729, 2465, 2821, and 6601. In testing primality of very large numbers chosen at random, the chance of stumbling upon a value that fools the Fermat test is less than the chance that cosmic radiation will cause the computer to make an error in carrying out a "correct" algorithm. Considering an algorithm to be inadequate for the first reason but not for the second illustrates the difference between mathematics and engineering.

Exercise 28

One variant of the Fermat test that cannot be fooled is called the Miller-Rabin test (Miller 1976; Rabin 1980). This starts from an alternate form of Fermat's Little Theorem, which states that if n is a prime number and a is any positive integer less than n, then $a^{n-1} \equiv 1(\mathrm{mod})n$. To test the primality of a number n by the Miller-Rabin test, we pick a random number $altn$ and find $a^{n-1}(\mathrm{mod})n$ using the expmod procedure. However, whenever we perform the squaring step in expmod, we check to see if we have discovered a "nontrivial square root of 1 modulo n ," that is, a number not equal to 1 or $n-1$ whose square is equal to 1 modulo n. It is possible to prove that if such a nontrivial square root of 1 exists, then n is not prime. It is also possible to prove that if n is an odd number that is not prime, then, for at least half the numbers $a<n$, computing a^{n-1} in this way will reveal a nontrivial square root of 1 modulo n. (This is why the Miller-Rabin test cannot be fooled.) Modify the expmod procedure to signal if it discovers a nontrivial square root of 1, and use this to implement the Miller-Rabin test with a procedure analogous to fermat-test. Check your procedure by testing various known primes and non-primes. Hint: One convenient way to make expmod signal is to have it return 0.

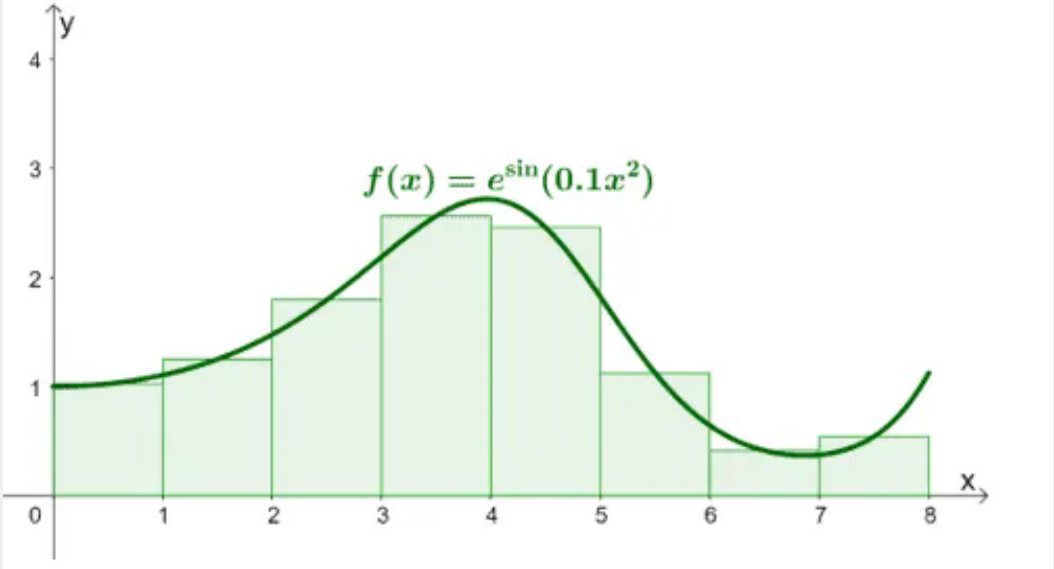
TODO

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

$$\sum_{k=a}^{b} f(n) = f(a) + ... + f(b)$$

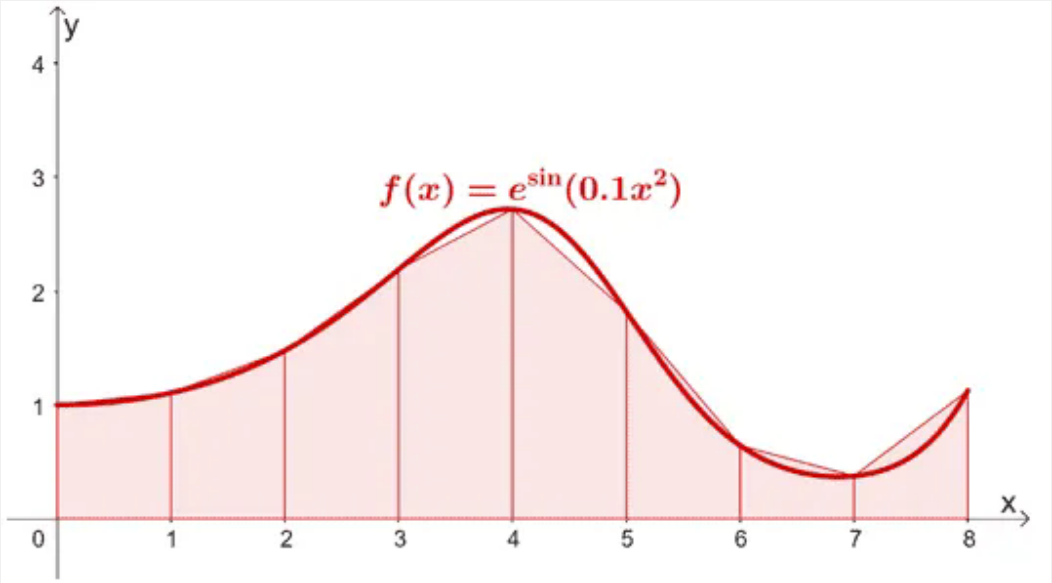
MIDPOINT THEOREM

$$\int_a^b f(x) = dx\left[f\left(a + \frac{dx}{2}\right) + f\left(a + dx + \frac{dx}{2}\right) + f\left(a + 2dx + \frac{dx}{2}\right)\right]$$



TRAPEZOIDAL THEOREM

$$\int_z^b f(x) = \frac{dx}{2}[f(a) + 2(f(a + dx) + f(a + 2dx) + f(a + 3dx) + f(a + 4dx)...)f(b)]$$



Exercise 29

Simpson's Rule is a more accurate method of numerical integration than the method illustrated above. Using Simpson's Rule, the integral of a function f between a and b is approximated as:

$$\left(\frac{h}{3}\right)(y_0 + 4y_1 + 2y_2 + 4y_3 + 2y_4 + \cdots + 2y_{n-2} + 4y_{n-1} + y_n)$$

where $h = (b - a)/n$, for some even integer n, and $y_k = f(a + kh)$. (Increasing n increases the accuracy of the approximation.) Define a procedure that takes as arguments f , a, b, and n and returns the value of the integral, computed using Simpson's Rule. Use your procedure to integrate cube between 0 and 1 (with $n = 100$ and $n = 1000$), and compare the results to those of the integral procedure shown above.

```
(define (integral f a b n)

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

  (define (h) (/ (- b a) n))

  (define (y k)
    (f (+ a (* k (h)))))

  (define (element k)
    (cond ((= k 0) (* 1 (y k)))
          ((= k n) (* 1 (y k)))
          ((odd? k) (* 4 (y k)))
          (else (* 2 (y k)))))

  (define (sus n)
    (+ n 1))

  (* (/ (h) 3) (sum element 0 sus n)))

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

(integral square 0 1 100.00)
;0.33333333333333337
(integral square 0 1 1000.00)
;0.33333333333333326
```

HOLY SHIT THIS TOOK ME A WHOLE DAY TO DO, i have no idea about calc 1 i guess ;-; also just discovered that you need double decimal digits after the number to get accurate decimal places :P

Exercise 30

ǫe sum procedure above generates a linear recursion. ǫe procedure can be rewriǖen so that the sum is performed iteratively. Show how to do this by filling in the missing expressions in the following definition:

```
(define (sum term a next b)
  (define (iter a result)
    (if (??)
        (??)
        (iter (??)(??))))
  (iter (??)(??)))
```

```
(define (sum term a next b)
  (define (iter a result)
    (if (> a b)
        result
        (iter (next a) (+ result (term a)))))
  (iter a 0))
```

Exercise 31

- ǫe sum procedure is only the simplest of a vast number of similar abstractions that can be captured as higher-order procedures. Write an analogous procedure called product that returns the product of the values of a function at points over a given range. Show how to define factorial in terms of product. Also use product to compute approximations to π using the formula.

$$\frac{\pi}{4} = \frac{2.4.4.6.6.8...}{3.3.5.5.7.7...}$$

- If your product procedure generates a recursive process, write one that generates an iterative process. If it generates an iterative process, write one that generates a recursive process.

- ```
  (define (products func a next b)
    (if (> a b) 1
        (* (func a) (products func (next a) next b))))

  ; defining factorial
  (define (factorial n)
  ```
  ```

```
  (define (next n) (+ n 1))
  (define (element k) k)
  (products element 1 next n))

(factorial 3)
```

i am simplifying to find pi

$$\frac{\pi}{4} = \frac{2.4.4.6.6.8...}{3.3.5.5.7.7...}$$
$$\frac{\pi}{2} = \frac{2.2.4.4.6.6.8...}{3.3.5.5.7.7...}$$
$$\frac{\pi}{2} = \frac{2^2.4^2.6^2.8^2...}{3^2.5^2.7^2...}$$

```
(define (next-term x) (+ x 2))

(define (element x) (* x x))

(define (pie precision)
  (* 2 (/ (product element 2 next-term precision)
          (product element 3 next-term (- precision 1)))))

(pie 100)
; 315
(pie 1000)
; 3143
(pie 10000)
; 31417
(pie 100000)
; 314160
; pretty good i guess?
```

- ```
  (define (products-iter func a next b)
    (define (helperiter counter result)
      (if (> counter b) result
          (helperiter (next counter) (* result (func counter)))))
    (helperiter a 1))
  ```
  ```

Exercise 32

1. Show that sum and product (Exercise 1.31) are both special cases of a still more general notion called accumulate that combines a collection of terms, using some general accumulation function:

```
(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 and show how sum and product can both be defined as simple calls to accumulate.

2. If your accumulate procedure generates a recursive process, write one that generates an iterative process. If it generates an iterative process, write one that generates a recursive process.

1.
```
(define (accumulate combiner null-value term a next b)
  (cond ((> a b) null-value)
        (else (combiner (term a)
                        (accumulate combiner null-value term (next a) next
b)))))
```

2.
```
(define (accumulate-iter combiner null-value term a next b)
  (define (helperiter counter result)
    (if (> counter b)
        result
        (helperiter (next counter) (combiner result (term counter)))))
  (helperiter a null-value))
```

for the definitions of sum and product

```
(define (sum term a next b)
  (accumulate + 0 term a next b))

(define (product term a next b)
  (accumulate * 1 term a next b))
```

Exercise 33

You can obtain an even more general version of accumulate (Exercise 1.32) by introducing the notion of a filter on the terms to be combined. That is, combine only those terms derived from values in the range that satisfy a specified condition. The resulting filtered-accumulate abstraction takes the same arguments as accumulate, together with an additional predicate of one argument that specifies the filter. Write filtered-accumulate as a procedure. Show how to express the following using filtered-accumulate:

1. the sum of the squares of the prime numbers in the interval a to b (assuming that you have a prime? predicate already written)
2. the product of all the positive integers less than n that are relatively prime to n (i.e., all positive integers $i < n$ such that $\mathrm{GCD}(i, n) = 1$).

i don't even understand wtf this question is asking me to do tbh ;-; what is a filter :'D

Exercise 34

Suppose we define the procedure

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

Then we have

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

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

```
application: not a procedure;
 expected a procedure that can be applied to arguments
  given: 2
  context...:
```

this results in syntax error because

```
(f f)
(f 2)
(2 2) ; <----this is wrong syntax
```

Exercise 35

> Show that the golden ratio φ (Section 1.2.2) is a fixed point of the transformation $x \mapsto 1 + 1/x$ and use this fact to compute φ by means of the `fixed-point` procedure.

wishfully thinking that φ IS really the fixed point of $x \mapsto 1 + 1/x$ then:

$$1 + \frac{1}{x} = x$$
$$x + 1 = x^2$$
$$x^2 - x - 1 = 0$$
$$x = \frac{1 \pm \left(\sqrt{5}\right)}{2}$$

then

```
  (display "value of phi")
  (/ (+ 1.0 (sqrt 5.0)) 2.0)

(define tolerance 0.00001)

(define (fixed-point f first-guess)
  (define (close-enough? v1 v2)
    (< (abs (- v1 v2))
       tolerance))
  (define (try guess)
    (let ((next (f guess)))
      (if (close-enough? guess next)
          next
          (try next))))
  (try first-guess))

(define (phicalc x)
  (+ 1.00 (/ 1.00 x)))

(display "value of phi using phicalc")
(fixed-point phicalc 1.5)
; value of phi: 1.618033988749895
; value of phi using phicalc: 1.6180327868852458
```

Exercise 36

Modify `fixed-point` so that it prints the sequence of approximations it generates, using the `newline` and `display` primitives shown in Exercise 1.22. Then find a solution to $x^x = 1000$ by finding a fixed point of $x \mapsto \log(1000)/\log(x)$. (Use Scheme's primitive `log` procedure, which computes natural logarithms.) Compare the number of steps this takes with and without average damping. (Note that you cannot start `fixed-point` with a guess of 1, as this would cause division by $\log(1) = 0$.)

```
(define tolerance 0.00001)

(define (fixed-point f first-guess)
  (define (close-enough? v1 v2)
    (< (abs (- v1 v2))
       tolerance))
  (define (try guess)
    (let ((next (f guess)))
      (display "try: ") (display next) (newline)
      (if (close-enough? guess next)
          next
          (try next))))
  (try first-guess))

(define (func x)
  (/ (log 1000) (log x)))

(define (damp x)
  (/ (+ (func x) x) 2))

(fixed-point (lambda (x) (damp (func x))) 2)
; $ racket test.rkt
; try: 6.485128247251651
; try: 4.490141163115099
; try: 4.563162341921608
; try: 4.5546831412819255
; try: 4.555631480818134
; try: 4.555524951832057
; try: 4.555536912620918
; try: 4.555535569622013
; 4.555535569622013
; $ racket test.rkt | wc -l
; 9
```

```
(define tolerance 0.00001)

(define (fixed-point f first-guess)
  (define (close-enough? v1 v2)
    (< (abs (- v1 v2))
       tolerance))
```

```
  (define (try guess)
    (let ((next (f guess)))
      (display "try: ") (display next) (newline)
      (if (close-enough? guess next)
          next
          (try next))))
  (try first-guess))

(define (func x)
  (/ (log 1000) (log x)))

(define (damp x)
  (/ (+ (func x) x) 2))

(fixed-point func 2)

; $ racket test.rkt
; try: 9.965784284662087
; try: 3.004472209841214
; try: 6.279195757507157
; try: 3.759850702401539
; try: 5.215843784925895
; try: 4.182207192401397
; try: 4.8277650983445906
; try: 4.387593384662677
; try: 4.671250085763899
; try: 4.481403616895052
; try: 4.6053657460929
; try: 4.5230849678718865
; try: 4.577114682047341
; try: 4.541382480151454
; try: 4.564903245230833
; try: 4.549372679303342
; try: 4.559606491913287
; try: 4.552853875788271
; try: 4.557305529748263
; try: 4.554369064436181
; try: 4.556305311532999
; try: 4.555028263573554
; try: 4.555870396702851
; try: 4.555315001192079
; try: 4.5556812635433275
; try: 4.555439715736846
; try: 4.555599009998291
; try: 4.555493957531389
; try: 4.555563237292884
; try: 4.555517548417651
; try: 4.555547679306398
; try: 4.555527808516254
; try: 4.555540912917957
```

```
; try: 4.555532270803653
; 4.555532270803653
;
; $ racket test.rkt | wc -l
; 35
```

it takes

$$35 - 9$$

more steps without avg damping

Exercise 37

1. An infinite continued fraction is an expression of the form

$$\cfrac{N_1}{D_1 + \cfrac{N_2}{D_2 + \cfrac{N_3}{D_3 + \ldots}}}$$

As an example, one can show that the infinite continued fraction expansion with the N_i and the D_i all equal to 1 produces $1/\varphi$, where φ is the golden ratio (described in 1.2.2). One way to approximate an infinite continued fraction is to truncate the expansion after a given number of terms. Such a truncation–a so-called k-term finite continued fraction–has the form

$$\cfrac{N_1}{D_1 + \cfrac{N_2}{\ldots + \cfrac{N_k}{D_k}}}$$

Suppose that `n` and `d` are procedures of one argument (the term index i) that return the N_i and D_i of the terms of the continued fraction. Define a procedure `cont-frac` such that evaluating `(cont-frac n d k)` computes the value of the k-term finite continued fraction. Check your procedure by approximating $1/\varphi$ using

```
(cont-frac (lambda (i) 1.0) (lambda (i) 1.0) k)
```

for successive values of `k`. How large must you make `k` in order to get an approximation that is accurate to 4 decimal places?

2. If your `cont-frac` procedure generates a recursive process, write one that generates an iterative process. If it generates an iterative process, write one that generates a recursive process.

```
; initial was iterative
(define (cont-frac n d k)
  (define (helperiter acc x)
    (if (= x 0)
        acc
```

```
        (helperiter (/ (n x) (+ (d x) acc)) (- x 1))))
  (helperiter 0 k))

(cont-frac (lambda (x) 1.0) (lambda (x) 1.0) 100)
(/ 1.00 (/ (+ 1.00 (sqrt 5.00)) 2.00))

; now for recursive it is shrimple
```

Exercise 38

In 1737, the Swiss mathematician Leonhard Euler published a memoir De Fractionibus Continuis, which included a continued fraction expansion for $e - 2$, where e is the base of the natural logarithms. In this fraction, the N_i are all 1, and the D_i are successively:

`1, 2, 1, 1, 4, 1, 1, 6, 1, 1, 8, ...`

Write a program that uses your `cont-frac` procedure from Exercise 1.37 to approximate e, based on Euler's expansion.

```
(define (den k)
  (if (= (remainder k 3) 2)
      (+ 2.00 (* 2.00 (quotient k 3)))
      1.00))

(define (num k) 1)

(define (approx-e k)
  (+ 2.00 (cont-frac num den k)))

(approx-e 10)
;2.7182817182817183
(approx-e 100)
;2.7182818284590455
```

first try 😎 [me when the modular arithmetic]

Exercise 39

A continued fraction representation of the tangent function was published in 1770 by the German mathematician J.H. Lambert:

$$\tan x = \cfrac{x}{1 - \cfrac{x^2}{3 - \frac{x^2}{5 - \cdots}}}$$

where x is in radians. Define a procedure `(tan-cf x k)` that computes an approximation to the tangent function based on Lambert's formula. `k` specifies the number of terms to compute, as in Exercise 1.37.

```
(define (tan-sus x k)
  (define (den n)
    (- (* n 2) 1))
  (define (num n)
    (if (= n 1) x
        (* -1 x x)))
  (cont-frac num den k))

(tan 3.14)
-0.000592653659180776
(tan-sus 3.14 10)
-0.0005926555452557651
```

huh.....pretty good....i guess? idk?

Exercise 40

Define a procedure `cubic` that can be used together with the `newtons-method` procedure in expressions of the form

```
(newtons-method (cubic a b c) 1)
```

to approximate zeros of the cubic $x^3 + ax^2 + bx + c$.

```
(define (abs x)
  (cond ((> x 0) x)
        ((< x 0) (* -1 x))
        (else 0)))

(define (fixed-point f first-guess)
  (define (close-enough? v1 v2)
    (< (abs (- v1 v2))
       tolerance))
  (define (try guess)
    (let ((next (f guess)))
      (if (close-enough? guess next)
          next
          (try next))))
  (try first-guess))

(define (newton-transform g)
  (lambda (x) (- x (/ (g x) ((deriv g) x)))))

(define (deriv g)
  (define dx 0.001)
  (lambda (x) (/ (- (g (+ x dx)) (g x)) dx)))

(define (newtons-method g guess)
  (fixed-point (newton-transform g) guess))
```

```
(define (cubic a b c)
  (lambda (x) (+ (* x x x) (* a (* x x)) (* b x) c)))

(newtons-method (cubic -3 1 1) 1)
; 1
```

Exercise 41

Define a procedure `double` that takes a procedure of one argument as argument and returns a procedure that applies the original procedure twice. For example, if `inc` is a procedure that adds 1 to its argument, then `(double inc)` should be a procedure that adds 2. What value is returned by

```
(((double (double double)) inc) 5)
```

`first try 😎 hehehe`

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

(define (dooble x)
  (lambda (s) (x (x s))))

((dooble inc) 9)
```

for `(((dooble (dooble dooble)) inc) 5)`

```
(dooble dooble)
; (lambda (s) (dooble (dooble s)))
(dooble (dooble dooble))
; (lambda (y) ((lambda (s) (dooble (dooble s))) ((lambda (s) (dooble (dooble
s))) y)))
; (lambda (y) ((lambda (s) (dooble (dooble s))) (dooble (dooble y))))
; (lambda (y) (dooble (dooble (dooble (dooble y)))))
((dooble (dooble dooble)) inc)
; (lambda (y) (dooble (dooble (dooble (dooble y)))) inc)
; (dooble (dooble (dooble (dooble inc))))
; ^(2*)   ^(2*)   ^(2*)   ^(2*inc)
; (2*2*2*2*inc)
; (16*inc)
; (+ 16)
(((dooble (dooble dooble)) inc) 5)
; (+ 16 5)
```

Exercise 42

Let f and g be two one-argument functions. The composition f after g is defined to be the function $x \mapsto f(g(x))$. Define a procedure `compose` that implements composition. For example, if `inc` is a procedure that adds 1 to its argument,

```
((compose square inc) 6)
49
```

shrimple

```
(define (compose a b)
  (lambda (s) (a (b s))))

((compose square inc) 5)
; 36
```

Exercise 43

If f is a numerical function and n is a positive integer, then we can form the n^{th} repeated application of f, which is defined to be the function whose value at x is $f(f(\ldots(f(x))\ldots))$. For example, if f is the function $x \mapsto x + 1$, then the n^{th} repeated application of f is the function $x \mapsto x + n$. If f is the operation of squaring a number, then the n^{th} repeated application of f is the function that raises its argument to the 2^n-th power. Write a procedure that takes as inputs a procedure that computes f and a positive integer n and returns the procedure that computes the n^{th} repeated application of f. Your procedure should be able to be used as follows:

```
((repeated square 2) 5)
625
```

Hint: You may find it convenient to use `compose` from Exercise 1.42.

```
(define (repeated f n)
  (cond ((= n 1) f)
        (else (compose (repeated f (- n 1)) f))))

((repeated square 2) 5)
; 625
```

Exercise 44

The idea of smoothing a function is an important concept in signal processing. If f is a function and dx is some small number, then the smoothed version of f is the function whose value at a point x is the average of $f(x - \mathrm{dx})$, $f(x)$, and $f(x + \mathrm{dx})$. Write a procedure `smooth` that takes as input a procedure that computes f and returns a procedure that computes the smoothed f. It is sometimes valuable to repeatedly smooth a function (that is, smooth the smoothed function, and so on) to obtain the n-fold smoothed function. Show how to generate the n-fold smoothed function of any given function using `smooth` and `repeated` from Exercise 1.43.

```
(define dx 0.0001)
(define (smooth f)
  (lambda (x) (/ (+ (f (+ x dx))
                    (f x)
                    (f (- x dx)))
                 3)))
((smooth square) 2)
; 4.000000006666666
(((repeated smooth 5) square) 2)
; 4.000000033333335
```

Exercise 45

We saw in 1.3.3 that attempting to compute square roots by naively finding a fixed point of $y \mapsto x/y$ does not converge, and that this can be fixed by average damping. The same method works for finding cube roots as fixed points of the average-damped $y \mapsto x/y^2$. Unfortunately, the process does not work for fourth roots—a single average damp is not enough to make a fixed-point search for $y \mapsto x/y^3$ converge. On the other hand, if we average damp twice (i.e., use the average damp of the average damp of $y \mapsto x/y^3$) the fixed-point search does converge. Do some experiments to determine how many average damps are required to compute n^{th} roots as a fixed-point search based upon repeated average damping of $y \mapsto x/y^{\{n-1\}}$. Use this to implement a simple procedure for computing n^{th} roots using `fixed-point`, `average-damp`, and the `repeated` procedure of Exercise 1.43. Assume that any arithmetic operations you need are available as primitives.

i don't even understand wtf i need to do TODO

Exercise 46

> Several of the numerical methods described in this chapter are instances of an extremely general computational strategy known as iterative improvement. Iterative improvement says that, to compute something, we start with an initial guess for the answer, test if the guess is good enough, and otherwise improve the guess and continue the process using the improved guess as the new guess. Write a procedure `iterative-improve` that takes two procedures as arguments: a method for telling whether a guess is good enough and a method for improving a guess. `Iterative-improve` should return as its value a procedure that takes a guess as argument and keeps improving the guess until it is good enough. Rewrite the `sqrt` procedure of 1.1.7 and the `fixed-point` procedure of 1.3.3 in terms of `iterative-improve`.

```
(define (iter-improve good-enough? improve)
  (define (iterating x)
    (if (good-enough? x)
        x
        (iterating (improve x))))
  iterating)

(define (average x y)
  (/ (+ x y) 2))

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

(define (squareroot x)

  (define (improving y)
    (average y (/ x y)))

  (define (good? s)
    (< (abs (- (square s) x)) 0.01))

  ((iter-improve good? improving) x))

(squareroot 2.00)
; 1.4166666666666665

(define (fixed-point f guess)
  (define (close? s)
    (< (abs (- (f s) s)) 0.01))
  ((iter-improve close? f) guess))

(fixed-point sin 1.0)
```

btw remember that the fixed point of a function is just the point at which $f(x) = x$ or the point at which $y = x$ intersects $y = f(x)$

SECTION 2

say gex

Exercise 1

Define a better version of `make-rat` that handles both positive and negative arguments. `Make-rat` should normalize the sign so that if the rational number is positive, both the numerator and denominator are positive, and if the rational number is negative, only the numerator is negative.

idk how……..but uhhhhhh……..this *automatically* works in racket with…….definition as is

```
(define (make-rat n d)
  (cond ((< d 0) (make-rat (- n) (- d)))
        (else (let ((g (gcd n d)))
                (cons (/ n g)
                      (/ d g))))))

(make-rat 4 12)
;'(1 . 3)
(make-rat 4 -12)
;'(1 . -3)
(make-rat -4 12)
;'(1 . -3)
(make-rat -4 -12)
;'(1 . 3)
```

Exercise 2

Consider the problem of representing line segments in a plane. Each segment is represented as a pair of points: a starting point and an ending point. Define a constructor `make-segment` and selectors `start-segment` and `end-segment` that define the representation of segments in terms of points. Furthermore, a point can be represented as a pair of numbers: the x coordinate and the y coordinate. Accordingly, specify a constructor `make-point` and selectors `x-point` and `y-point` that define this representation. Finally, using your selectors and constructors, define a procedure `midpoint-segment` that takes a line segment as argument and returns its midpoint (the point whose coordinates are the average of the coordinates of the endpoints). To try your procedures, you'll need a way to print points:

```
(define (print-point p)
  (newline)
  (display "(")
  (display (x-point p))
  (display ",")
  (display (y-point p))
  (display ")"))
```

```
(define (make-point x y)
  (cons x y))
(define (x-point p)
  (car p))
(define (y-point p)
  (cdr p))

(define (make-segment p1 p2)
  (cons p1 p2))
(define (start-segment s)
  (car s))
(define (end-segment s)
  (cdr s))

(define (midpoint-segment s)
  (define (mid-x x1 x2)
    (/ (+ x1 x2) 2))
  (define (mid-y y1 y2)
    (/ (+ y1 y2) 2))
  (define x1 (car (car s)))
  (define x2 (car (cdr s)))
  (define y1 (cdr (car s)))
  (define y2 (cdr (cdr s)))
  (cons (mid-x x1 x2) (mid-y y1 y2)))


(define (print-point p)
  (newline)
  (display "[")
  (display (x-point p))
  (display ",")
  (display (y-point p))
  (display "]"))

(print-point (midpoint-segment (make-segment
                                  (make-point 0 0)
                                  (make-point 2 2))))
; [1,1]
```

Exercise 3

Implement a representation for rectangles in a plane. (Hint: You may want to make use of Exercise 2.2.) In terms of your constructors and selectors, create procedures that compute the perimeter and the area of a given rectangle. Now implement a different representation for rectangles. Can you design your system with suitable abstraction barriers, so that the same perimeter and area procedures will work using either representation?

ok so this is basically just asking us to make a rectangle given two points
god this is tedious

```
(define (make-point x y)
  (cons x y))
(define (x-point p)
  (car p))
(define (y-point p)
  (cdr p))

(define (make-rect p1 p2)
  (define (height p1 p2)
    (abs (- (y-point p1) (y-point p2))))
  (define (width p1 p2)
    (abs (- (x-point p1) (x-point p2))))
  (cons (width p1 p2) (height p1 p2)))

(define (area r)
  (* (car r) (cdr r)))

(define (perimeter r)
  (* (abs (+ (car r) (cdr r))) 2))


(area (make-rect (make-point 0 0) (make-point 3 3)))
;9
(perimeter (make-rect (make-point 0 0) (make-point 3 3)))
;12
```

ehhh idk? is this ok?

now.....how tf are they asking me to make *implement a different representation for rectangles* -w-

Exercise 4

Here is an alternative procedural representation of pairs. For this representation, verify that `(car (cons x y))` yields `x` for any objects `x` and `y`.

```
(define (cons x y)
  (lambda (m) (m x y)))

(define (car z)
  (z (lambda (p q) p)))
```

What is the corresponding definition of `cdr`? (Hint: To verify that this works, make use of the substitution model of 1.1.5.)

```
(define (cons x y)
  (lambda (m) (m x y)))

(define (car z)
  (z (lambda (p q) p)))

(car (cons 3 4))
; 3

(define (cdr z)
  (z (lambda (p q) q)))

(cdr (cons 3 4))
; 4
```

first try ezz

Exercise 5

Show that we can represent pairs of nonnegative integers using only numbers and arithmetic operations if we represent the pair a and b as the integer that is the product $2^a 3^b$. Give the corresponding definitions of the procedures `cons`, `car`, and `cdr`.

ok for this one i am totally blank tbh TODO

Exercise 6

In case representing pairs as procedures wasn't mind-boggling enough, consider that, in a language that can manipulate procedures, we can get by without numbers (at least insofar as nonnegative integers are concerned) by implementing 0 and the operation of adding 1 as

```
(define zero (lambda (f) (lambda (x) x)))

(define (add-1 n)
  (lambda (f) (lambda (x) (f ((n f) x)))))
```

This representation is known as Church numerals, after its inventor, Alonzo Church, the logician who invented the λ-calculus.

Define `one` and `two` directly (not in terms of `zero` and `add-1`). (Hint: Use substitution to evaluate `(add-1 zero)`). Give a direct definition of the addition procedure `+` (not in terms of repeated application of `add-1`).

TODO

Exercise 7

Alyssa's program is incomplete because she has not specified the implementation of the interval abstraction. Here is a definition of the interval constructor:

```
(define (make-interval a b) (cons a b))
```

Define selectors `upper-bound` and `lower-bound` to complete the implementation.

```
(define (upper-bound i) (car i))
(define (lower-bound i) (cdr i))
```

Exercise 8

Using reasoning analogous to Alyssa's, describe how the difference of two intervals may be computed. Define a corresponding subtraction procedure, called `sub-interval`.

```
(define (sub-interval x y)
  (make-interval (- (lower-bound x) (upper-bound y))
                 (- (upper-bound x) (lower-bound y))))
```

Exercise 9

The width of an interval is half of the difference between its upper and lower bounds. The width is a measure of the uncertainty of the number specified by the interval. For some arithmetic operations the width of the result of combining two intervals is a function only of the widths of the argument intervals, whereas for others the width of the combination is not a function of the widths of the argument intervals. Show that the width of the sum (or difference) of two intervals is a function only of the widths of the intervals being added (or subtracted). Give examples to show that this is not true for multiplication or division.

optional

Exercise 10

Ben Bitdiddle, an expert systems programmer, looks over Alyssa's shoulder and comments that it is not clear what it means to divide by an interval that spans zero. Modify Alyssa's code to check for this condition and to signal an error if it occurs.

optional

Exercise 11

In passing, Ben also cryptically comments: 'By testing the signs of the endpoints of the intervals, it is possible to break `mul-interval` into nine cases, only one of which requires more than two multiplications.' Rewrite this procedure using Ben's suggestion.

optional

Exercise 12

Define a constructor `make-center-percent` that takes a center and a percentage tolerance and produces the desired interval. You must also define a selector `percent` that produces the percentage tolerance for a given interval. The `center` selector is the same as the one shown above.

optional

Exercise 13

Show that under the assumption of small percentage tolerances there is a simple formula for the approximate percentage tolerance of the product of two intervals in terms of the tolerances of the factors. You may simplify the problem by assuming that all numbers are positive.

optional

Exercise 14

Demonstrate that Lem is right. Investigate the behavior of the system on a variety of arithmetic expressions. Make some intervals A and B, and use them in computing the expressions A/A and A/B. You will get the most insight by using intervals whose width is a small percentage of the center value. Examine the results of the computation in center-percent form (see Exercise 2.12).

optional

Exercise 15

Eva Lu Ator, another user, has also noticed the different intervals computed by different but algebraically equivalent expressions. She says that a formula to compute with intervals using Alyssa's system will produce tighter error bounds if it can be written in such a form that no variable that represents an uncertain number is repeated. Thus, she says, `par2` is a better program for parallel resistances than `par1`. Is she right? Why?

optional

Exercise 16

Explain, in general, why equivalent algebraic expressions may lead to different answers. Can you devise an interval-arithmetic package that does not have this shortcoming, or is this task impossible? (Warning: This problem is very difficult.)

optional

Exercise 17

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

```
(last-pair (list 23 72 149 34))
(34)
```

```
(define (last-pair x)
  (if (null? (cdr x))
      x
      (last-pair (cdr x))))

(last-pair (list 1 23 4 5))
; '(5)
```

Exercise 18

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

```
(reverse (list 1 4 9 16 25))
(25 16 9 4 1)
```

THE PROBLEM!!! THE ONE AND ONLY!!! THE INTERVIEW PROBLEM!!!!

```
(define (reverse l)
  (if (null? l)
      l
      (append (reverse (cdr l)) (cons (car l) null))))

(reverse (list 1 2 3 4))
; '(4 3 2 1)
```

also an iterative one for good measure btw

```
(define (reverse-iter l)
  (define (helper result l)
    (if (null? l) result
        (helper (cons (car l) result)
                (cdr l))))
  (helper null l))
```

Exercise 19

Consider the change-counting program of 1.2.2. It would be nice to be able to easily change the currency used by the program, so that we could compute the number of ways to change a British pound, for example. As the program is written, the knowledge of the currency is distributed partly into the procedure `first-denomination` and partly into the procedure `count-change` (which knows that there are five kinds of U.S. coins). It would be nicer to be able to supply a list of coins to be used for making change.

We want to rewrite the procedure `cc` so that its second argument is a list of the values of the coins to use rather than an integer specifying which coins to use. We could then have lists that defined each kind of currency:

```
(define us-coins
  (list 50 25 10 5 1))

(define uk-coins
  (list 100 50 20 10 5 2 1 0.5))
```

We could then call `cc` as follows:

```
(cc 100 us-coins)
292
```

To do this will require changing the program `cc` somewhat. It will still have the same form, but it will access its second argument differently, as follows:

```
(define (cc amount coin-values)
  (cond ((= amount 0)
         1)
        ((or (< amount 0)
             (no-more? coin-values))
         0)
        (else
         (+ (cc
             amount
             (except-first-denomination
              coin-values))
            (cc
             (- amount
                (first-denomination
                 coin-values))
             coin-values)))))
```

Define the procedures `first-denomination`, `except-first-denomination` and `no-more?` in terms of primitive operations on list structures. Does the order of the list `coin-values` affect the answer produced by `cc`? Why or why not?

```
(define (no-more? x) (null? x))
(define (except-first-denomination x) (cdr x))
(define (first-denomination x) (car x))
```

shrimple

Exercise 20

The procedures +, *, and `list` take arbitrary numbers of arguments. One way to define such procedures is to use `define` with dotted-tail notation. In a procedure definition, a parameter list that has a dot before the last parameter name indicates that, when the procedure is called, the initial parameters (if any) will have as values the initial arguments, as usual, but the final parameter's value will be a list of any remaining arguments. For instance, given the definition

```
(define (f x y . z) ⟨body⟩)
```

the procedure `f` can be called with two or more arguments. If we evaluate

```
(f 1 2 3 4 5 6)
```

then in the body of `f`, `x` will be 1, `y` will be 2, and `z` will be the list `(3 4 5 6)`. Given the definition

```
(define (g . w) ⟨body⟩)
```

the procedure `g` can be called with zero or more arguments. If we evaluate

```
(g 1 2 3 4 5 6)
```

then in the body of `g`, `w` will be the list `(1 2 3 4 5 6)`.

Use this notation to 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. For example,

```
(same-parity 1 2 3 4 5 6 7)
(1 3 5 7)

(same-parity 2 3 4 5 6 7)
(2 4 6)
```

```
(define (same-parity f . x)
  (cond ((or (null? f) (null? x)) null)
        ((even? f) (find-even x))
        ((odd? f) (find-odd x))))

(define (find-even x)
  (define (help result lst)
    (cond ((null? lst) result)
          ((odd? (car lst)) (help result (cdr lst)))
          ((even? (car lst)) (help (cons (car lst) result) (cdr lst)))))
  (help null x))
```

```
(define (find-odd x)
  (define (help result lst)
    (cond ((null? lst) result)
          ((even? (car lst)) (help result (cdr lst)))
          ((odd? (car lst)) (help (cons (car lst) result) (cdr lst)))))
  (help null x))

(same-parity 8 2 3 4 5 6)
; '(6 4 2)
```

ok so this is giving the solution in reverse, and i know the reason BUT idk how to make append such that it appends a pair and a “non-pair” without messing up the tree

Exercise 21

The procedure `square-list` takes a list of numbers as argument and returns a list of the squares of those numbers.

```
(square-list (list 1 2 3 4))
(1 4 9 16)
```

Here are two different definitions of `square-list`. Complete both of them by filling in the missing expressions:

```
(define (square-list items)
  (if (null? items)
      nil
      (cons ⟨??⟩ ⟨??⟩)))

(define (square-list items)
  (map ⟨??⟩ ⟨??⟩))
```

```
(define (square-list items)
  (if (null? items)
      null
      (cons (* (car items) (car items)) (square-list (cdr items)))))

(define (square-list-2 items)
  (map (lambda (x) (* x x)) items))
```

Exercise 22

Louis Reasoner tries to rewrite the first `square-list` procedure of Exercise 2.21 so that it evolves an iterative process:

```
(define (square-list items)
  (define (iter things answer)
    (if (null? things)
        answer
        (iter (cdr things)
              (cons (square (car things))
                    answer))))
  (iter items nil))
```

Unfortunately, defining `square-list` this way produces the answer list in the reverse order of the one desired. Why?

Louis then tries to fix his bug by interchanging the arguments to `cons`:

```
(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 either. Explain.

ok this is the EXACT issue i had with `Exercise 2.20` and i understand that in the first one it is pretty obvious, you're appending the *new value* to the *old value which already HAS a list of the squares of the previous elements*

in the second one though we get

```
(square-list (list 1 2 3 4))
;'(((((() . 1) . 4) . 9) . 16)
```

so this is clearly doing

```
(cons (cons (cons (cons null 1) 4) 9) 16)
```

Exercise 23

The procedure `for-each` is similar to `map`. It takes as arguments a procedure and a list of elements. However, rather than forming a list of the results, `for-each` just applies the procedure to each of the elements in turn, from left to right. The values returned by applying the procedure to the elements are not used at all—`for-each` is used with procedures that perform an action, such as printing. For example,

```
(for-each
 (lambda (x) (newline) (display x))
 (list 57 321 88))

57
321
88
```

The value returned by the call to `for-each` (not illustrated above) can be something arbitrary, such as true. Give an implementation of `for-each`.

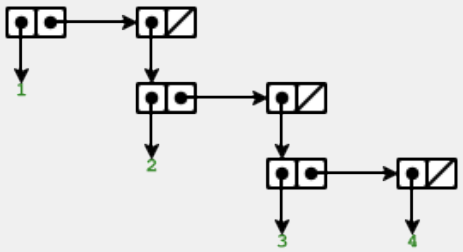
```
(define (for-each proc items)
  (proc (car items))
  (if (null? (cdr items)) true
      (for-each proc (cdr items))))

(for-each (lambda (x) (newline)
                      (display x)
                      (display "->")
                      (display (square x))) (list 1 2 3 4 5))
; 1->1
; 2->4
; 3->9
; 4->16
; 5->25#t
```

Exercise 24

Suppose we evaluate the expression `(list 1 (list 2 (list 3 4)))`. Give the result printed by the interpreter, the corresponding box-and-pointer structure, and the interpretation of this as a tree (as in Figure 2.6).

```
(list 1 (list 2 (list 3 4)))
; (1 (2 (3 4)))
```



Exercise 25

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

```
(1 3 (5 7) 9)
((7))
(1 (2 (3 (4 (5 (6 7))))))
```

```
(define l1 (list 1 3 (list 5 7) 9))
(define l2 (list (list 7)))
(define l3 (list 1 (list 2 (list 3 (list 4 (list 5 (list 6 7)))))))
(car (cdr (car (cdr (cdr l1)))))
(car (car l2))
(car
  (cdr
    (car
      (cdr
        (car
          (cdr                              ; stairs :0
            (car
              (cdr
                (car
                  (cdr
                    (car
                      (cdr l3)))))))))))) ;(what even is this question bro XD)
```

Exercise 26

Suppose we define `x` and `y` to be two lists:

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

What result is printed by the interpreter in response to evaluating each of the following expressions:

```
(append x y)
(cons x y)
(list x y)
```

```
(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))
```

Exercise 27

Modify your `reverse` procedure of Exercise 2.18 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. For example,

```
(define x
  (list (list 1 2) (list 3 4)))

x
((1 2) (3 4))

(reverse x)
((3 4) (1 2))

(deep-reverse x)
((4 3) (2 1))
```

```
(define (reverse l)
  (define (helper result l)
    (if (null? l) result
        (helper (cons (car l) result)
                (cdr l))))
  (helper null l))

(define (deep-reverse x)
  (if (not (pair? x))
      x
      (reverse x)))
```

Exercise 28

Write a procedure `fringe` that takes as argument a tree (represented as a list) and returns a list whose elements are all the leaves of the tree arranged in left-to-right order. For example,

```
(define x
  (list (list 1 2) (list 3 4)))

(fringe x)
(1 2 3 4)

(fringe (list x x))
(1 2 3 4 1 2 3 4)
```

ok so this one is wrong BUT it needs a *tiny* bit of correction

```
(define (fringe lst)
  (cond ((null? lst) null)
```

```
        ((not (pair? lst)) lst)
        (else (append (fringe (car lst))
                      (fringe (cdr lst))))))

(fringe '((1 2) (3 4) 9))
; append: contract violation
```

instead of returning the fringe as is *if not a pair* we should return it by *making it a pair*

```
(define (fringe lst)
  (cond ((null? lst) null)
        ((not (pair? lst)) (cons lst null))
        (else (append (fringe (car lst))
                      (fringe (cdr lst))))))

(fringe '((1 2) (3 4) 9))
; '(1 2 3 4 9)
```

i basically copied the `count-leaves` program in the example

Exercise 29

A binary mobile consists of two branches, a left branch and a right branch. Each branch is a rod of a certain length, from which hangs either a weight or another binary mobile. We can represent a binary mobile using compound data by constructing it from two branches (for example, using `list`):

```
(define (make-mobile left right)
  (list left right))
```

A branch is constructed from a `length` (which must be a number) together with a `structure`, which may be either a number (representing a simple weight) or another mobile:

```
(define (make-branch length structure)
  (list length structure))
```

1. Write the corresponding selectors `left-branch` and `right-branch`, which return the branches of a mobile, and `branch-length` and `branch-structure`, which return the components of a branch.

2. Using your selectors, define a procedure `total-weight` that returns the total weight of a mobile.

3. A mobile is said to be balanced if the torque applied by its top-left branch is equal to that applied by its top-right branch (that is, if the length of the left rod multiplied by the weight hanging from that rod is equal to the corresponding product for the right side) and if each of the submobiles hanging off its branches is balanced. Design a predicate that tests whether a binary mobile is balanced.

4. Suppose we change the representation of mobiles so that the constructors are

```
(define (make-mobile left right)
  (cons left right))

(define (make-branch length structure)
  (cons length structure))
```

How much do you need to change your programs to convert to the new representation?

TODO (it's easy.....i think)

Exercise 30

Define a procedure `square-tree` analogous to the `square-list` procedure of Exercise 2.21. That is, `square-tree` should behave as follows:

```
(square-tree
(list 1
(list 2 (list 3 4) 5)
(list 6 7)))
(1 (4 (9 16) 25) (36 49))
```

Define `square-tree` both directly (i.e., without using any higher-order procedures) and also by using `map` and recursion.

```
(define (square-tree tree)
  (cond ((null? tree) null)
        ((not (pair? tree)) (square tree))
        (else (cons (square-tree (car tree))
                    (square-tree (cdr tree))))))

(define (square-tree-map tree)
  (define (sub-tree x)
    (if (not (pair? x))
        (square x)
        (square-tree-map x)))
  (map sub-tree tree))

(list 1 (list 2 (list 3 4) 5) (list 6 7))
; '(1 (2 (3 4) 5) (6 7))
(square-tree (list 1 (list 2 (list 3 4) 5) (list 6 7)))
; '(1 (4 (9 16) 25) (36 49))
(square-tree-map (list 1 (list 2 (list 3 4) 5) (list 6 7)))
; '(1 (4 (9 16) 25) (36 49))
```

Exercise 31

Abstract your answer to Exercise 2.30 to produce a procedure `tree-map` with the property that `square-tree` could be defined as

```
(define (square-tree tree)
  (tree-map square tree))
```

```
(define (tree-map proc tree)
  (cond ((null? tree) null)
        ((not (pair? tree)) (proc tree))
        (else (cons (tree-map proc (car tree))
                    (tree-map proc (cdr tree))))))
```

```
(define (sus x)
  (tree-map square x))

(sus (list 1 (list 2 (list 3 4) 5) (list 6 7)))
; '(1 (4 (9 16) 25) (36 49))
```

Exercise 32

We can represent a set as a list of distinct elements, and we can represent the set of all subsets of the set as a list of lists. For example, if the set is `(1 2 3)`, then the set of all subsets is `(() (3) (2) (2 3) (1) (1 3) (1 2) (1 2 3))`. Complete the following definition of a procedure that generates the set of subsets of a set and give a clear explanation of why it works:

```
(define (subsets s)
  (if (null? s)
      (list nil)
      (let ((rest (subsets (cdr s))))
        (append rest (map ⟨??⟩ rest)))))
```

```
(define (subsets s)
  (if (null? s)
      (list null)
      (let ((rest (subsets (cdr s))))
        (append rest (map (lambda (x) (cons (car s) x)) rest)))))

(subsets (list 1 2 3))
; '(() (3) (2) (2 3) (1) (1 3) (1 2) (1 2 3))
```

Exercise 33

Fill in the missing expressions to complete the following definitions of some basic list-manipulation operations as accumulations:

```
(define (map p sequence)
  (accumulate (lambda (x y) ⟨??⟩)
              nil sequence))

(define (append seq1 seq2)
  (accumulate cons ⟨??⟩ ⟨??⟩))

(define (length sequence)
  (accumulate ⟨??⟩ 0 sequence))
```

```
(define (map-test p sequence)
  (accumulate (lambda (acc y) (cons (p y) acc))
```

```
              null sequence))

(define (append seq1 seq2)
  (accumulate cons seq1 seq2))

(define (length sequence)
  (accumulate (lambda (_ acc) (+ acc 1)) 0 sequence))
```

Exercise 34

Evaluating a polynomial in x at a given value of x can be formulated as an accumulation. We evaluate the polynomial

$$a_n x^n + a_{n-1} x^{n-1} + ... + a_1 x + a_0$$

using a well-known algorithm called Horner's rule, which structures the computation as

$$(...(a_n x + a_{n-1})x + ... + a_1)x + a_0$$

In other words, we start with a_n, multiply by x, add a_{n-1}, multiply by x, and so on, until we reach a_0.

Fill in the following template to produce a procedure that evaluates a polynomial using Horner's rule. Assume that the coefficients of the polynomial are arranged in a sequence, from a_0 through a_n.

```
(define
  (horner-eval x coefficient-sequence)
  (accumulate
   (lambda (this-coeff higher-terms)
     (??))
   0
   coefficient-sequence))
```

For example, to compute $1 + 3x + 5x^3 + x^5$ at $x = 2$ you would evaluate

```
(horner-eval 2 (list 1 3 0 5 0 1))
```

```
(define (horner-eval x coefficient-sequence)
  (accumulate
   (lambda (this-coeff higher-terms)
     (+ this-coeff (* higher-terms x)))
   0
   coefficient-sequence))
```

Exercise 35

Redefine `count-leaves` from 2.2.2 as an accumulation:

```
(define (count-leaves t)
  (accumulate ⟨??⟩ ⟨??⟩ (map ⟨??⟩ ⟨??⟩)))
```

```
(define (count-leaves-test t)
  (accumulate (lambda (y x)
                (+ x y))
              0
              (map (lambda (x)
                     (if (not (pair? x))
                         1
                         (count-leaves-test x)))
                   t)))

(count-leaves-test (list (list 1 23) 1 2 3))
; 5
```

Exercise 36

The procedure `accumulate-n` is similar to `accumulate` except that it takes as its third argument a sequence of sequences, which are all assumed to have the same number of elements. It applies the designated accumulation procedure to combine all the first elements of the sequences, all the second elements of the sequences, and so on, and returns a sequence of the results. For instance, if `s` is a sequence containing four sequences, `((1 2 3) (4 5 6) (7 8 9) (10 11 12))`, then the value of `(accumulate-n + 0 s)` should be the sequence `(22 26 30)`. Fill in the missing expressions in the following definition of `accumulate-n`:

```
(define (accumulate-n op init seqs)
  (if (null? (car seqs))
      nil
      (cons (accumulate op init ⟨??⟩)
            (accumulate-n op init ⟨??⟩))))
```

```
(define (accumulate-n op init seqs)
  (if (null? (car seqs))
      null
      (cons (accumulate op init (map car seqs))
            (accumulate-n op init (map cdr seqs)))))
```

this one i didn't think of,
i am still not getting used to the idea of thinking in a higher level of abstraction of map and acc i guess

Exercise 37

Suppose we represent vectors $v = (v_i)$ as sequences of numbers, and matrices $m = (m_{ij})$ as sequences of vectors (the rows of the matrix). For example, the matrix

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 4 & 5 & 6 & 6 \\ 6 & 7 & 8 & 9 \end{pmatrix}$$

is represented as the sequence `((1 2 3 4) (4 5 6 6) (6 7 8 9))`. With this representation, we can use sequence operations to concisely express the basic matrix and vector operations. These operations (which are described in any book on matrix algebra) are the following:

(dot-product v w) returns $\Sigma_i v_i w_i$
(matrix-*-vector m v) returns the vector **t** where $t_i = \Sigma_j m_{ij} w_i$
(matrix-*-vector m n) returns the vector **p** where $p_{ij} = \Sigma_k m_{ik} n_{kj}$
(transpose m) returns the vector **n** where $n_{ij} = m_{ji}$

We can define the dot product as

```
(define (dot-product v w)
  (accumulate + 0 (map * v w)))
```

Fill in the missing expressions in the following procedures for computing the other matrix operations. (The procedure `accumulate-n` is defined in Exercise 2.36.)

```
(define (matrix-*-vector m v)
  (map ⟨??⟩ m))

(define (transpose mat)
  (accumulate-n ⟨??⟩ ⟨??⟩ mat))

(define (matrix-*-matrix m n)
  (let ((cols (transpose n)))
    (map ⟨??⟩ m)))
```

```
(define (dot-product v w)
  (accumulate + 0 (map * v w)))

(define (matrix-vector m v)
  (map (lambda (x) (dot-product x v)) m))

(define (transpose mat)
  (accumulate-n cons null mat))

(define (matrix-matrix m n)
  (let ((cols (transpose n)))
    (map (lambda (row) (matrix-vector cols row)) m)))
```

```
(define m1 (list (list 1 2) (list 3 4)))
(define m2 (list (list 2 0) (list 1 2)))
(matrix-matrix m1 m2)
; '((4 4) (10 8))
```

Exercise 38

The `accumulate` procedure is also known as `fold-right`, because it combines the first element of the sequence with the result of combining all the elements to the right. There is also a `fold-left`, which is similar to `fold-right`, except that it combines elements working in the opposite direction:

```
(define (fold-left op initial sequence)
  (define (iter result rest)
    (if (null? rest)
        result
        (iter (op result (car rest))
              (cdr rest))))
  (iter initial sequence))
```

What are the values of

```
(fold-right / 1 (list 1 2 3))
(fold-left  / 1 (list 1 2 3))
(fold-right list nil (list 1 2 3))
(fold-left  list nil (list 1 2 3))
```

Give a property that `op` should satisfy to guarantee that `fold-right` and `fold-left` will produce the same values for any sequence.

```
(fold-right / 1 (list 1 2 3))
(fold-left  / 1 (list 1 2 3))
(fold-right list nil (list 1 2 3))
(fold-left  list nil (list 1 2 3))
; 3/2
; 1/6
; '(1 (2 (3 ())))
; '((() 1) 2) 3)

(fold-right (lambda (y x)
              (display x)
              (display " dividing ")
              (display y)
              (display " -> ")
              (/ y x)) 1 (list 1 2 3))
(fold-left (lambda (y x)
             (display x)
             (display " dividing ")
             (display y)
```

```
            (display " -> ")
            (/ y x)) 1 (list 1 2 3))
; 1 dividing 3 -> 3 dividing 2 -> 2/3 dividing 1 -> 3/2
; 1 dividing 1 -> 2 dividing 1 -> 3 dividing 1/2 -> 1/6
```

we want it such that the *order* of op does not matter on the result i.e
like in addition $(a + b) + c = a + (b + c)$
or multiplication $(a * b) * c = a * (b * c)$ so we want it such that `(op (op a b) c) == (op a (op b c))`

Exercise 39

Complete the following definitions of `reverse` (Exercise 2.18) in terms of `fold-right` and `fold-left` from Exercise 2.38:

```
(define (reverse sequence)
  (fold-right
   (lambda (x y) ⟨??⟩) nil sequence))

(define (reverse sequence)
  (fold-left
   (lambda (x y) ⟨??⟩) nil sequence))
```

```
(define (reverse-test sequence)
  (accumulate
   (lambda (x y) (append  y (list x))) null sequence))

(define (reverse2-test sequence)
  (fold-left
   (lambda (x y) (append (list y) x)) null sequence))

(reverse-test (list 1 2 3 4))
(reverse2-test (list 1 2 3 4))
; '(4 3 2 1)
; '(4 3 2 1)
```

Exercise 40

Define a procedure `unique-pairs` that, given an integer n, generates the sequence of pairs (i, j) with $(1 < j < (i < n))$. Use `unique-pairs` to simplify the definition of `prime-sum-pairs` given above.

```
(define (unique-pairs n)
  (flatmap (lambda (i)
             (map (lambda (j) (list i j))
```

```
                (enumerate-interval 1 (- i 1))))
          (enumerate-interval 1 n)))

(unique-pairs 4)
; '((2 1) (3 1) (3 2) (4 1) (4 2) (4 3))

(define (prime-sum-pairs n)
  (define (make-pair-sum pair)
    (list (car pair) (cadr pair) (+ (car pair) (cadr pair))))
  (define (prime-sum? pair)
    (prime? (+ (car pair) (cadr pair))))
  (map make-pair-sum
       (filter prime-sum?
               (unique-pairs n))))

(prime-sum-pairs 5)
; '((2 1 3) (3 2 5) (4 1 5) (4 3 7) (5 2 7))
```