Question 1 (5 points):

What will Scheme print in response to the following expressions? If an expression produces an error message, you may just write “error”; you don’t have to provide the exact text of the message. If the value of an expression is a procedure, just write “procedure”; you don’t have to show the form in which Scheme prints procedures.

\[
\text{(every (lambda (x) (/ x 2))}
\]
\[
\text{(keep even?)}
\]
\[
\text{(every (lambda (x) (* x x))}
\]
\[
\text{'(2 3 4 5))}
\]
\[
\text{(map (lambda (x) (se x x))}
\]
\[
\text{'(a b c))}
\]
\[
\text{(every (lambda (x) (se x x))}
\]
\[
\text{'(a b c))}
\]
\[
\text{(map (lambda (x)}
\]
\[
\text{(let ((x (+ x 1))}
\]
\[
\text{(y x))}
\]
\[
\text{(* x y))})}
\]
\[
\text{'(2 5))}
\]
\[
\text{(cddadr '(a b c d e) (f g h l j) (k l m n o))}
\]

Question 2 (4 points):

What will Scheme print in response to the following expressions? If an expression produces an error message, you may just write “error”; you don’t have to provide the exact text of the message. Also, draw a box and pointer diagram for the value produced by each expression.

\[
\text{(list (list 2 (cons 3 5))})
\]

\[
\text{(append '(5 6 (cons 1 '(2 3)))}
\]
Question 3 (3 points):

(deftn (trick x y) (* y y))

What is the result of (trick (/ 1 0) 5)

a. In normal order

b. In applicative order

(deftn (inc x) (+ x 1))

c. True or False: In evaluating (inc (inc (+ 3 2))), + gets called more in normal than in applicative order

________True ______False

Question 4 (4 points):

(a) what is the order of growth in time of foo below, in terms of n, its argument? (Hint: if n is odd, so is n - 2.) also, does foo generate an iterative or a recursive process?

(deftn (foo n)

   (cond ((= n 1) 1)
         ((even? n) (foo (+ n 1)))
         (else (foo (- n 2)))))

________Θ(n) ______Θ(n^2) ______Θ(2^n) ______Not enough information to know

_______Iterative _______Recursive

(b) What is the order of growth in time of count-to below? Note: using se with a sentence as its first argument and a word as its second argument takes time proportional to the length of the sentence.

(deftn (count-to N)

   (if (= N 1) '(1)

         (se (count-to (- N 1)) N)))

________Θ(n) ______Θ(n^2) ______Θ(2^n) ______Not enough information to know

_______Iterative _______Recursive
Question 5 (8 points):

For this question (both parts), use only higher order procedures, not recursion, even in helper procedures!

(a) Define a procedure is-pig-latin? That takes a sentence as its argument, and determines whether or not it is a pig-latin sentence. In other words, if every word in the sentence ends with ay, the procedure returns #t, and otherwise it returns #f.

(b) Define a procedure latin-change that given a sentence as argument, changes all words that end in ay to end in ey.
Question 6 (8 points):

Write a procedure range that takes three arguments, a sentence sent and two words from and to. It looks for a range of words within sent that start with from and end with to:

> (range '(being for the benefit of mister kite) 'for 'of)

(for the benefit of)

If the from and to words occur more than once, just return the first range found:

> (range '(party of the first part sells party of the second part this car)

 'party 'part)

(party of the first part)

The return value is a sentence that starts with the from word and ends with the to word.

The from and to words may appear any number of times in the sentence, but you should just use the first occurrence of the from word, and the first occurrence of the to word that comes after the from word. You may not assume that the to word will appear after the from word does:

> (range '(come to my party) 'party 'part)

( )

Hint: It’s okay for you to examine each word in the sentence more than once. Write helper procedures.
Question 7 (7 points):

This question is about the iterative-improve procedure from exercise 1.46, page 78:

(define (iterative-improve good-enough? Improve)
  (define (help x)
    (if (good-enough? x)
      x
      (help (improve x))))
  help)

(a) Fill in the blanks in the following definition of piglatin:

(define (piglatin wd)
  (word ((iterative-improve

                                    _______________________________________________________
                                    _______________________________________________________
                                    _______________________________________________________

    wd)

  'ay))

(b) Another candidate for rewriting in terms of iterative-improve would be the repeated function from exercise 1.43, page 77. But in fact this doesn’t work, because the “good enough” condition depends not on the current argument value, but on the number of times the function has been called. Write a function count-iterative-improve that’s like iterative-improve except that each of its two argument functions takes two arguments: the current guess, and the number of times the improve function has been called so far. Here’s how it will be used:

(define (repeated fn num)
  (count-iterative-improve (lambda (x cnt) (= cnt num))
    (lambda x cnt) (fn x))))