WHAT DOES SCHEME PRINT?
Write down what Scheme will show if you type these expressions into the interpreter.

1. (let ((x 3)) (lambda (y) (+ x y)))
   #[closure argslst=...]

2. ((lambda (x) (let ((+ -) x)) (+ 3 2))
   5

3. (and or (not #f) (not not) 2)
   #f

4. ((word ‘but ‘first) ‘hello)
   Error

BOXES AND POINTERS
Write down what the list looks like and draw the box and pointer diagrams.

1. (cons (list 1 3) (append (list (cons 2 3)) (list 4)))

```
---+---  ---+---  ---+---
-->| . | -|--|-------| . | -|--| . | / |
+-|--|---+  ---+|---+  ---+|---+  4
| . | -|--| . | / |
+-|--|---+  ---+|---+  ---+|---+

1 3 3 2 3

(1 3) (2 . 3) 4)
```

2. (list (append (list 3) (cons 4 '(i))))

```
---+---
-->| . | / |
+-|--|---+
| . | / |
+-|--|---+

3 4

(3 4)
```
3. (cons (list 2 4) (list 3 6))

```
  +-------+ +-------+ +-------+
--->| . | -|--->| . | / | 3
+-|-------+ +|--->| . | 6
  v
 +-------+ +-------+
| . | -|--->| . | / |
+-|-------+ +|--->| . | 2
  v  v

((2 4) 3 6)
```

4. (cons (cons 3 1) (list))

```
  +-------+
--->| . | / | 2
+-|-------+
  v
 +-------+
| . | . | 3
+-|------|
  v  v

((3 . 1))
```

5. (define x '(1 (2 3)))

a. draw x.

```
x--->| . | -|--->| . | / | 3
+-|-------+ +|--->| . | 1
  v
 +-------+ +-------+
| . | -|--->| . | / |
+-|-------+ +|--->| . | 2
  v  v

b. what does (cdr x) return?

>(cdr x)
((2 3))
```

**ORDERS OF GROWTH**

1. Suppose a procedure foo requires time $\Theta(n)$ and a procedure bar requires time $\Theta(\log n)$. Also, foo
returns \( n \) and \( \bar{a} \) returns \( \log n \). What time do the following procedure calls require?

a. \((\ast (\text{foo } n) (\text{foo } n))\)

\(\Theta(n)\)

b. \((\text{foo } (\text{bar } n))\)

\(\Theta(\log(n))\)

c. \((\text{bar } (\text{foo } n))\)

\(\Theta(n)\)

2a. Write a procedure \((\text{fib } n)\) to calculate the \( n \)th Fibonacci number. Use a recursive process. What is the order of growth? The \( n \)th Fibonacci number is given by \( F(n) = F(n-2) + F(n-1) \).

\[
\begin{align*}
(\text{define } (\text{fib } n) \quad & \\
& (\text{if } (\text{or } (= n 1) (= n 2)) \\
& 1 \\
& (+ (\text{fib } (- n 1)) (\text{fib } (- n 2))))
\end{align*}
\]

The recursive version of fib has an order of growth of \(\Theta(2^n)\).

b. Now rewrite \(\text{fib}\) using an iterative process. What is the order of growth? Is this better or worse than the version in part a?

\[
\begin{align*}
(\text{define } (\text{fib } n) \quad & \\
& (\text{define } (\text{fib}-\text{iter } n \ \text{previous } \text{current}) \\
& (\text{cond } (\text{or } (= n 1) \text{ previous}) \\
& (\text{or } (= n 2) \text{ current}) \\
& (\text{else } (\text{fib}-\text{iter } (- n 1) \text{ current } (+ \text{ previous } \text{current})))))) \\
& (\text{fib}-\text{iter } n \ 1 \ 1))
\end{align*}
\]

The iterative definition of \(\text{fib}\) has an order of growth of \(\Theta(n)\), since it makes \(n\) recursive calls to \(\text{fib-iter}\). This is clearly better than the recursive version in part a.

3. What does the following code produce in applicative order? Normal order?

\[
\begin{align*}
(\text{define } (\text{iwontstop } n) (\text{iwontstop } (- n 1))) \\
(\text{define } (\text{makemenormal } x \ y) (\text{if } (> y 0) \ y \ x)) \\
(\text{makemenormal } (\text{iwontstop } 3) \ 5)
\end{align*}
\]

**Applicative order:** infinite loop

**Normal order:** 5

**LISTS**

1. This exercise will have you implement mergesort, a sorting algorithm.
a. Given two lists of numbers, write a procedure called merge that returns a list in which the two lists of numbers are “merged” into increasing order. So, for example, (merge (list 1 3 4 6) (list 3 5 7 8)) returns the list (1 3 4 5 6 7 8), while (merge (list 1 2 3 4) (list 5 6 7 8)) returns (list 1 2 3 4 5 6 7 8). You should assume that the lists are already in increasing order.

    (define (merge ls1 ls2)
      (cond ((null? ls1) ls2)
            ((null? ls2) ls1)
            ((< (car ls1) (car ls2))
             (cons (car ls1) (merge (cdr ls1) ls2)))
            (else (cons (car ls2) (merge ls1 (cdr ls2))))))

b. Given a list of numbers, write a procedure called sublist that also takes in two arguments – start and end – and returns the sublist that starts at position start and ends at position end. Assume that the list indices start from 0. For example, (sublist (list 2 3 4 5) 1 3) should return the list (3 4 5).

    (define (sublist ls start end)
      (if (> start 0)
          (sublist (cdr ls) (- start 1) (- end 1))
          (if (= end 0)
              (list (car ls))
              (cons (car ls) (sublist (cdr ls) start (- end 1))))))

c. We will now implement the mergesort algorithm to sort a list of numbers into increasing order. The algorithm works as follows:
   i. If a list is of length zero or one, then the list is already sorted.
   ii. Otherwise, we separate the list into two smaller, equally-sized lists, sort the smaller lists, and merge the two sorted lists.

Implement the procedure called mergesort that takes in a list of numbers and sorts the list using the mergesort algorithm.

    (define (merge-sort ls)
      (if (or (= (length ls) 0)
               (= (length ls) 1))
          ls
          (let ((m (/ (length ls) 2)))
            (merge (merge-sort (sublist ls 0 (- m 1)))
                   (merge-sort (sublist ls m (- (length ls) 1))))))

NOTE: We are assuming the list argument has a length that is a power of 2 (in other words, we can halve its length repeatedly).

**DATA ABSTRACTION**

Let’s implement a very simple representation of Pokemon. A Pokemon’s attributes will simply contain three fields, defined in the following way:

    (define (pokemon type level experience) (list type level experience))

We wish to be able to reference a Pokemon’s attributes, but we want to do so in a meaningful way.
a. Write the selectors for type, level, and experience. For example, a Pokemon’s type would be defined thus: (define type car).

(define type car)
(define level cadr)
(define experience caddr)

b. Now we wish to be able to make our Pokemon battle each other:
First, if one Pokemon is at least five levels above the other, it automatically wins. Next, if the Pokemon are within five levels of each other, the super-effective type wins. Finally, if neither of the above is true, whoever has more experience wins. The procedure pokemon-battle should return the winner, given two Pokemon poke1 and poke2. You may assume that the procedure super-effective is written. It takes two types and returns true if the first is super-effective against the second. Remember to respect the abstraction!

(define (pokemon-battle poke1 poke2)
  (cond
    ((> (- (level poke1) (level poke2)) 4) poke1)
    ((> (- (level poke2) (level poke1)) 4) poke2)
    ((super-effective (type poke1) (type poke2)) poke1)
    ((super-effective (type poke2) (type poke1)) poke2)
    ((> (experience poke1) (experience poke2)) poke1)
    (else poke2)))

c. Now suppose that for some weird reason, we decided to change the representation of Pokemon attributes to the following:
(define (pokemon type level experience) (list (cons level experience) type))
Rewrite the selectors so that pokemon-battle still works as intended.

(define level caar)
(define experience cdar)
(define type cadr)

HIGHER ORDER FUNCTIONS

1. Write sentfn, a procedure that takes an arithmetic function and a list of sentences of numbers and returns a new list of sentences that is the result of calling the function on each number in each sentence. For example:
   > (sentfn square ‘((2 5) (3 1 6)))
   ((4 25) (9 1 36))

   Use higher order functions, not recursion, and respect the abstraction!

   (define (sentfn fn sent-list)
     (map (lambda (sent) (every fn sent)) sent-list))

2. sum is a procedure that takes as an argument a sentence and returns the sum of all the numbers in that sentence and the letter count of the words in the sentence.
ex: (sum ‘(i can do it 9 times)) = 22
(sum ’(20 percent cooler)) = 33

a. Write sum using recursion. Do not use higher order functions.

Iterative:

(define (sum sent)
  (define (sum-iter sent result)
    (cond ((empty? sent) result)
      ((number? (first sent))
        (sum-iter (bf sent) (+ result (first sent))))
      (else (sum-iter (bf sent) (+ result (count (first sent)))))))
  (sum-iter sent 0))

Recursive:

(define (sum sent)
  (cond ((empty? sent) 0)
    ((number? (first sent))
      (+ (first sent) (sum (bf sent))))
    (else (+ (count (first sent)) (sum (bf sent))))))

b. Write sum using higher order functions. Do not use recursion.

(define (sum sent)
  (accumulate + (every (lambda (x) (if (number? x) x (count x)))) sent))