1. What will Scheme output? Draw box-and-pointer diagrams to help determine this.

(a) (cons (cons 1 nil) (cons 2 (cons (cons 3 (cons 4 5)) (cons 6 nil))))

(b) (cons (cons (car '(1 2 3)) (list 2 3 4)) (cons 2 3))

(c) (define a 4)
   ((lambda (x y) (+ a)) 1 2)

(d) ((lambda (x y z) (y x)) 2 / 2)

(e) ((lambda (x) (x x)) (lambda (y) 4))

(f) (define boom1 (/ 1 0))

(g) boom1

(h) (define boom2 (lambda () (/ 1 0)))

(i) (boom2)

(j) How can we rewrite boom2 without using the lambda operator?
2. What will Scheme output?
   (a) \( \text{if } 0 \ (\div 1 0) \ 1 \) \\

   (b) \( \text{and } 1 \ #f \ (\div 1 0) \) \\

   (c) \( \text{and } 1 \ 2 \ 3 \) \\

   (d) \( \text{or } #f \ #f \ 0 \ #f \ (\div 1 0) \) \\

   (e) \( \text{or } #f \ #f \ (\div 1 0) \ 3 \ 4 \) \\

   (f) \( \text{and } (\text{and}) \ (\text{or}) \)

3. \textbf{let} is a special form in Scheme which allows you to create local bindings. Consider the example

   \( \text{let } ((x 1)) \ (+ x 1) \) \\

   Here, we assign \( x \) to 1, and then evaluate the expression \((x + 1)\) using that binding, returning 2. However, outside of this expression, \( x \) would not be bound to anything.

   Each \textbf{let} special form has a corresponding lambda equivalent. The equivalent lambda expression for the above example is

   \( ((\lambda (x) (+ x 1)) 1) \)

   The following line of code does not work. Why? Write the lambda equivalent of the \textbf{let} expressions.

   \( \text{let } ((\text{foo } 3) \ \\
       \text{bar } (+ \text{foo } 2)) \ \\
       (+ \text{foo} \text{bar}) \)
2. Scoping

1. What is the difference between dynamic and lexical scoping?

2. What would this print using lexical scoping? What would it print using dynamic scoping?
   a = 2
   def foo():
       a = 10
       return lambda x: x + a
   bar = foo()
   bar(10)

3. How would you modify an environment diagram to represent dynamic scoping?
3 Code-Writing

1. Implement \texttt{waldo}. \texttt{waldo} returns \#t if the symbol \texttt{waldo} is in a list. You may assume that the list passed in is well-formed.

\begin{verbatim}
scm> (waldo '(1 4 waldo))
#t
scm> (waldo '())
#f
scm> (waldo '(1 4 9))
#f
\end{verbatim}

\begin{verbatim}
(define (waldo lst)
  )
\end{verbatim}
2. **Extra challenge:** Define `waldo` so that it returns the index of the list where the symbol `waldo` was found (if `waldo` is not in the list, return `#f`).

```scheme
scm> (waldo '(1 4 waldo))
2
scm> (waldo '())
#f
scm> (waldo '(1 4 9))
#f

(define (waldo lst)
  ...)
```
3. **(Optional)** The quicksort sorting algorithm is an efficient and commonly used algorithm to order the elements of a list. We choose one element of the list to be the pivot element and partition the remaining elements into two lists: one of elements less than the pivot and one of elements greater than the pivot. We recursively sort the two lists, which gives us a sorted list of all the elements less than the pivot and all the elements greater than the pivot, which we can then combine with the pivot for a completely sorted list.

Implement `quicksort` in Scheme. Choose the first element of the list as the pivot. You may assume that all elements are distinct. Hint: you may want to use a helper function.

You may additionally want to use the built-in `append` function, which takes in two lists and returns a new list containing the elements of the first list followed by the elements of the second list. You can also use `filter` procedure, which takes in a one-argument function and a list and returns a new list containing only the elements of the original list for which the function returns true, although it is not required.

```
scm> (quicksort (list 5 2 4 3 12 7))
(2 3 4 5 7 12)
```