Using the substitution model, show the evaluation of the following Scheme expression in detail. You should show all the evaluation steps, except that you may evaluate all the subexpressions of a combination which are \textit{variable lookups} or \textit{primitives} in one step, e.g. 
\((+ 5 3 8) \Rightarrow\)
\[
\begin{array}{c}
\text{plus}\ 5\ 3\ 8
\end{array}
\]

\(((\text{lambda} (n) (\text{if} (> n 3) + *)) 8) 4 2)\)
\[
\begin{array}{c}
(((\text{proc} (n) (\text{if} (> n 3) \ldots)) 8) 4 2)
\end{array}
\]
\[
\begin{array}{c}
\text{proc} (n) (\text{if} (> n 3) \ldots) \\
8 \\
4 2
\end{array}
\]
\[
\begin{array}{c}
\text{if} (> 8 3) * \\
\Rightarrow #t
\end{array}
\]
\[
\begin{array}{c}
\Rightarrow +
\end{array}
\]
\[
\begin{array}{c}
\Rightarrow \text{add}
\end{array}
\]
\[
\begin{array}{c}
\text{add} 4 2 \\
\Rightarrow 6
\end{array}
\]
2. Consider the following function \( t \):

\[
\text{define } t \\
\quad (\text{lambda } ((f \ \text{<function>})) \\
\quad \quad (\text{lambda } (x) \\
\quad \quad \quad (f (f (f x)))))
\]

For each of the following expressions, give the value that results from evaluating the expression, and justify your answer. You need not do the whole evaluation formally using the substitution model, but should give a (brief) explanation why your answer is correct.

There are three parts to this problem.

(a) \(((t \ \text{add1}) \ 0)\)

Evaluating \((t \ \text{add1})\) gives:

\[(\text{lambda } (x) (\text{add1 } (\text{add1 } (\text{add1 } x))))\]

This function applies \text{add1} three times to \(x\), so it might be called \text{add3}, i.e., a function that adds 3 to its argument. Therefore,

\[ ((t \ \text{add1}) \ 0) \Rightarrow (\text{add3 } 0) \Rightarrow 3 \]

(b) \(((t \ (t \ \text{add1})) \ 0)\)

We saw in part (a) that \((t \ \text{add1})\) was equivalent to a function which adds 3 to its argument, which we called \text{add3}. Substituting this into \((t \ (t \ \text{add1}))\) gives \((t \ \text{add3})\), or

\[(\text{lambda } (x) (\text{add3 } (\text{add3 } (\text{add3 } x)))) \ 0)\]

Thus, \((t \ (t \ \text{add1}))\) is a function which adds 3 to its argument three times—that is, it adds 9 to its argument. As before, you can imagine this is equivalent to an \text{add9} function. Applying \text{add9} to 0 gives 9.

(c) \(((\text{t t} \ \text{add1}) \ 0)\)

The trick here is to figure out what \(((\text{t t} \ \text{add1})\) is. Let \( P \) denote the following proc object:

\[
\text{proc } (f) \ (\text{lambda } (x) (f (f (f x))))
\]

Then,

\[ (t \ t) \Rightarrow \\
\quad [ \ [P \ P] ] \Rightarrow \\
\quad (\text{lambda } (x) ( [P \ (P \ (P \ x))]))
\]

Thus, you can see that \(((t \ t) \ \text{add1})\) evaluates to:

\[
( P \ (P \ (P \ \text{add1})))
\]

2
From part (b), we learned that \( (p (p \text{add1})) \) is add9. Substituting into this expression leaves us to evaluate \( (p \text{add9}) \). But this is just:

\[
\text{(lambda (x) (add9 (add9 (add9 x)))) 0}
\]

otherwise known as 27.
3. The transpose of a matrix $M$ is the matrix $M^T$ obtained by flipping $M$ about its main diagonal, so that its rows become columns, and its columns become rows. Formally, this means that if $a_{i,j}$ denotes the element of $M$ in row $i$ and column $j$, then the transpose takes $a_{i,j}$ to $a_{j,i}$.

For instance, the transpose of the matrix:

$$
M = \begin{pmatrix}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & 9 \\
10 & 11 & 12
\end{pmatrix}
$$

is the matrix:

$$
M^T = \begin{pmatrix}
1 & 4 & 7 & 10 \\
2 & 5 & 8 & 11 \\
3 & 6 & 9 & 12
\end{pmatrix}
$$

One way to represent a matrix in Scheme is as a list of rows, in which each row is a list of elements. In this representation, the first matrix above would be represented:

```
((1 2 3) (4 5 6) (7 8 9) (10 11 12))
```

and its transpose would be:

```
((1 4 7 10) (2 5 8 11) (3 6 9 12))
```

Note that the first list of the transpose consists of the first element of each list in the original matrix; the second list of the transpose consists of the second element of each list in the original matrix, etc.

Write a Scheme function `transpose` which takes a matrix, represented as a list of lists, and returns its transpose represented as a list of lists. You may assume that all of the lists in the input matrix have the same number of elements. Feel free to use built-in Scheme functions such as `append`, `filter`, `map`, `foldr`, `reverse`, `member`, etc. if they are useful.

```scheme
;; Transposes a matrix, making rows into columns and vice versa
(define transpose
  (lambda ((m <list>))
    (if (null? (car m)) ; are the rows empty?
      '()
      (cons (map car m) ; make a list of the first column
             (transpose (map cdr m))))))

;; An alternate solution, with a somewhat esoteric application of
;; the higher order 'apply' procedure.
(define transpose-2
  (lambda ((m <list>))
    (apply map list m)))
```
4. **Algorithmic Analysis** Suppose you are given a function (**insert** x L) which, given an object x and a proper list L, returns a new list consisting of the elements of L with x inserted into the correct location according to some ordering relation. Suppose **insert** runs in $O(n)$ time in the worst case, where n is the length of L. Given that this is so, what is the worst-case asymptotic runtime of the following implementation of the **sort** function? Justify your answer.

```
(define sort
  (lambda ((L <lst>))
    (if (null? L)
        ()
        (insert (car L)
            (sort (cdr L))))))
```

This algorithm is “insertion sort”, which is $O(n^2)$ time in the worst case, where n is the length of the list L being sorted.

**Justification:** The **insert** function is called n times, first with a list of length n, then of length n - 1, then n - 2, and so forth down to length 1. Each of these n calls takes $O(n)$ time (by the upper bound given for **insert**), so the overall function takes $O(n^2)$ time.

Another (more detailed) way to look at it is as follows—suppose $T(k)$ denotes the exact runtime of **insert** on a list of length k, in the worst case (meaning, we assume the value we’re inserting is going at the very end of the list). Then, we can write a function $S(n)$ that expresses the runtime of **sort**, in the worst case:

$$S(n) = S(n - 1) + T(n - 1) + c$$

In the base case, $S(0) = c$ for some constant c. If you expand this equation by substitution (making the usual simplifying assumptions about constants) you get:

$$S(n) \leq T(n - 1) + T(n - 2) + \ldots + T(0) + cn$$

Because we know that $T(k) = O(k)$, we know that there is some constant c so that $T(n - 1) \leq cn$ for all “sufficiently large” values of n. We know that $T(n - 1) > T(n - 2) > \ldots > T(0)$, and since we are looking for an upper bound, we’re willing to say:

$$S(n) \leq nT(n - 1) + c \leq n(cn) + c = cn^2 + c = O(n^2)$$

It is sufficient, for the purposes of this exam, to recognize that this function performs a worst-case $O(n)$ algorithm n times.
5. **Short Answer**

(a) Draw a box-and-pointer diagram to illustrate the data structure that would be created by evaluation of the following expression:

\[
\text{(define } L \text{ (cons (cons (cons 1 2) (cons 3 4)) '()))}
\]

(b) Is the structure pointed to by \( L \) in the problem above a *proper* list? Why or why not? 

Yes—the **cdr** of the pair pointed to by \( L \) is the empty list \( '() \). The fact that the **car** of the structure is not a proper list does not affect this question.
(c) For a class example, we implemented rational numbers. To do this, we created a new type `<rat>` as follows:

```
(define-class <rat> ()
  (numer <integer>)
  (denom <integer>))
```

Why is it preferable to define the `<rat>` type in this way, as opposed to simply writing

```
(define <rat> <pair>)
```

then defining `numer` and `denom` as aliases for `car` and `cdr`? 

If we define `<rat>` as equivalent to `<pair>`, then we will be unable to distinguish a pair representing a `<rat>` from a pair used to represent some other kind of structure. That would cause problems if we were using a `<rat>` as a parameter to a generic function, and we wanted that function to also have a method to handle the `<pair>` type in some other way.

In general, it is preferable for unrelated types to be disjoint, that is, each type should be distinguishable in some way from all other distinct types.

(d) Why doesn’t evaluating the following Scheme expression generate a “division by zero” error?

```
(lambda () (/ 1 0))
```

The body of a `lambda` expression is not evaluated until the resulting procedure object is applied. Since we have not applied the procedure in this instance, the code to divide by zero does not get evaluated.
6. For each of the following expressions, provide a definition of `blob` that would cause the expression to evaluate to 23. For instance, if the expression were:

(car blob)

...then one possible answer would be to write:

(define blob (cons 23 0))

There is more than one right answer in each case—please provide only one solution for each expression.

(a) (let* ((grog blob) (blob grog)) (+ blob grog 1))
    (define blob 11)

(b) (foldr + 0 (map (lambda (n) (- n 1)) blob))
    (define blob (list 24))

(c) (cadr (filter blob ’(5 18 21 23 37)))
    (define blob (lambda (n) (> n 18)))

(d) ((blob blob) blob)
    (define blob (lambda (x) (lambda (y) 23)))

(e) (+ 3 (* 4 (length (cdr (blob)))))
    (define blob (lambda () (list 1 2 3 4 5 6)))
7. A graph $G = (V, E)$ is a mathematical structure consisting of a set $V$ of vertices and a set $E$ of edges connecting those vertices. For instance, here is a graph with four vertices and eight edges:

![Graph Diagram]

In Problem Set 3, we represented graphs by explicitly keeping a list of vertices, and a separate list of edges (edges, you recall, were represented by structures of the form $⟨u, v⟩$, with $u$ being the vertex where the edge begins and $v$ being the vertex where the edge ends).

A different way to represent a graph is using a data structure known as an adjacency list. An adjacency list (as the name suggests) is a list, which has one element for each vertex in the graph. The element of the adjacency list corresponding to a particular vertex $u$ is itself a list, containing the names of all the vertices $u$ is adjacent to in the graph. Vertex $u$ is adjacent to vertex $v$ if there is an edge from $u$ to $v$.

For instance, the entry for vertex $a$ in the above example graph would be the list $(b, c, d)$, since those are the three vertices to which edges go starting from $a$.

Suppose you have been given a $<graph>$ class implemented using an adjacency list. Each vertex is represented as a $<symbol>$. You are given an accessor $(neighbors \ vtx \ G)$ which, given a graph $G$ and a vertex name $vtx$ returns a list of the vertices adjacent to $vtx$. Furthermore, you are given an accessor $(vertices \ G)$ which returns a list of the vertices of graph $G$.

Using these abstractions, write a function $(inbound \ vtx \ G)$ which, given a vertex name $vtx$ and a graph $G$, returns a list (possibly empty) of all the vertices from which there is an edge leading to $vtx$ in the graph.

You should feel free to use built-in Scheme functions such as append, filter, map, foldr, reverse, member, etc. if they are useful to you.

\[
\text{(define inbound (lambda ((vtx <symbol>) (G <graph>))}
\text{\hspace{1em} (filter (lambda (v)
\text{\hspace{3em} (memq vtx (neighbors v G)))
\text{\hspace{3em} (vertices G))))})
\]

⇐

9
8. Write a function \((k\text{-subsets } L \ k)\), which takes a proper list \(L\) and an integer \(k \geq 0\). It should return a list of all the lists of \(k\) elements that can be formed from the objects in \(L\). For example,

\[
(k\text{-subsets } '(a b c d) 2)
\]

...should return:

\[
((a b) (a c) (a d) (b c) (b d) (c d))
\]

Note that your function does not have to return the \(k\)-lists in the same order as is given here. \(\Leftarrow\)

This problem is related to recursively computing binomial coefficients. You are given a list \(L\) from which you want to compute a sublist of \(k\) elements. If \(k = 0\), then the only sublist to consider is the empty list \('()\), so that is the only list we need to return:

\[
(k\text{-subsets } '(a b c d) 0) \Rightarrow (())
\]

Similarly, if \(k\) equals the length of \(L\), then the whole list \(L\) is the only list we need to return:

\[
(k\text{-subsets } '(a b c d) 4) \Rightarrow ((a b c d))
\]

These are the base cases. So now, let us consider the recursive case, where \(0 < k < |L|\). Any list of \(k\) elements drawn from \(L\) either contains the first element of \(L\), or it does not. If it does, then it must contain \(k - 1\) elements drawn from the rest of \(L\). If not, it must contain \(k\) elements drawn from the rest of \(L\).

Since both are possible, we will construct both possibilities—the \(k\)-subsets drawn from \((\text{cdr } L)\), and the \((k - 1)\)-subsets drawn from \((\text{cdr } L)\) with \((\text{car } L)\) added to them. Here is how this might be written in Scheme:

\[
\text{(define k-subsets}
\]

\[
\text{(lambda ((L <list>) (k <integer>))}
\]

\[
\text{(cond ((= k 0)) ;; Base case, lists of length 0}
\]

\[
\text{((= k (length L)))}
\]

\[
\text{((list L))) ;; Base case, the whole list}
\]

\[
\text{(else}
\]

\[
\text{(append (map (lambda (x))}
\]

\[
\text{(cons (car L) x))}
\]

\[
\text{(k-subsets (cdr L) (- k 1)))}
\]

\[
\text{(k-subsets (cdr L) k))))))}
\]

\text{)}\)