Program the following 13 functions in LISP. Make sure you test them thoroughly. Sample data will be mailed to you. Turn in a listing of your program and the results of applying the test data.

1. Given two sets of atoms \( x \) and \( y \) represented as lists, write functions \( \text{union}[x, y] \), \( \text{intersection}[x, y] \) and \( \text{set_difference}[x, y] \), for their union \( x \cup y \), intersection \( x \cap y \), and set difference \( x \setminus y \), respectively. Use the function \( \text{member}[n, x] \) defined below, which may also be written as \( n \in x \):

\[
\text{member}(x, u) = \begin{cases} 
\text{nil} & \text{if null } u \\
\text{member}(x, \text{cdr } u) & \text{else if car } u \text{ eq } x \\
\text{t} & \text{else}
\end{cases}
\]

For example, \( (A \ B \ C) \cup (B \ C \ D) = (A \ B \ C \ D) \), \( (A \ B \ C) \cap (B \ C \ D) = (B \ C) \), and \( (A \ B \ C) \setminus (B \ C \ D) = (A) \).

Pay attention to getting correct the trivial (i.e., base) cases in which some of the arguments are \( \text{nil} \). In general, it is important to understand clearly the trivial cases of functions.

2. Given an integer \( n \) and a list \( l \) of integers sorted in increasing order, write a function \( \text{merge}[n, l] \) which inserts \( n \) in its proper place in \( l \). For example, \( \text{merge}[3, \ '(2 \ 4)] = (2 \ 3 \ 4) \), and \( \text{merge}[3, \ '(2 \ 3)] = (2 \ 3 \ 3) \).

3. Given two sets of atoms \( x \) and \( y \) represented as ordered lists containing no duplicates, write functions \( \text{union}[x, y] \), \( \text{intersection}[x, y] \) and \( \text{set_difference}[x, y] \) giving the union, intersection, and set difference, respectively, of \( x \) and \( y \); the result is wanted as an ordered list.

Note that computing these functions of unordered lists takes a number of comparisons proportional to the square of the number of elements of a typical list, while for ordered lists, the number of comparisons is proportional to the number of elements.

4. Using \( \text{merge} \), write a function named \( \text{sort}[l] \) that transforms an unordered list \( l \) into an ordered list. Your algorithm should repeatedly invoke the \( \text{merge} \) function starting with an empty list, thereby running in \( O(n^2) \) time for a list of \( n \) elements.

5. Write a predicate \( \text{occur}[a, s] \) to indicate whether an atom \( a \) occurs in a given s-expression \( s \), e.g., \( \text{occur}[B, \ '((A \ B) . C)] = t. \)

6. Write a function \( \text{num_occur}[a, s] \) that indicates how many times an atom \( a \) occurs in an s-expression \( s \), e.g., \( \text{num_occur}[B, \ '((A \ B) . C)] = 1. \)

7. Write a function \( \text{nodups}[s] \) to make a list without duplications of the atoms occurring in an s-expression \( s \), e.g., \( \text{nodups}['((A . B) . (C . A))]' = (A \ B \ C). \)
8. Write a function `multiplicity[s]` that indicates which atoms occur more than once in an s-expression `s`. The result should be in the form of a list of pairs (i.e., an assoc-list), where each pair consists of the atom that occurs more than once and its multiplicity, e.g., `multiplicity['(A . B) . (C . A)] = ((A . 2)).`

9. Write a predicate `multi_occurs_expr[x, y]` that indicates whether or not an s-expression `x` has more than one occurrence of an s-expression `y` as a sub-expression, e.g., `multi_occurs_expr['(A . B) . (C . (A . B)), (A . B)] = t.`