1. In a certain language expressions are written in infix notation. The language has binary operators and a unary prefix operator, whose precedence classes are as follows:

	binary operators	unary prefix operator	associativity
Class 1:	#	@	left
Class 2:	%	[none]	left
Class 3:	$ ^	[none]	right

Class 1 operators have ***highest*** precedence and class 3 operators have ***lowest*** precedence.

(i) Here are three expressions in this language; in each case, ***circle the operator that is applied last***:

(a1) : w ^ v # u **[0.5 pt.]**

(a2): y % @ z % x **[0.5 pt.]**

(a3) : w ^ v # u $ @ (y % @ z % x) **[0.5 pt.]**

(ii) Draw an abstract syntax tree of ***each*** of the above expressions (a1), (a2), and (a3). **[2 pts.]**

Tree of (a1) Tree of (a2) Tree of (a3)

(iii) Rewrite the first expression of part (i) (i.e., the expression (a1)) in ***prefix*** notation.

_____ **[0.5 pt.]**

(iv) Rewrite the second expression of part (i) (i.e., the expression (a2)) in ***prefix*** notation.

_____ **[0.5 pt.]**

(v) Rewrite the third expression of part (i) (i.e., the expression (a3)) in ***prefix*** notation.

_____ **[0.5 pt.]**

2.[**1 pt.**] What is the value of the Lisp expression `(cons '(+ 3 4) '(+ 3 4))`? Circle the correct answer:

(a) `((+ 3 4) + 3 4)` (b) `(+ 3 4 + 3 4)` (c) `(7 7)`
(d) `(7 + 3 4)` (e) `((+ 3 4) (+ 3 4))`

3.[**1 pt.**] What is the value of the Lisp expression `(append '(+ 3 4) '(+ 3 4))`? Circle the correct answer:

(a) `((+ 3 4) + 3 4)` (b) `(+ 3 4 + 3 4)` (c) `(7 7)`
(d) `(7 + 3 4)` (e) `((+ 3 4) (+ 3 4))`

.[**1 pt.**] What is the value of the Lisp expression `(list '(+ 3 4) '(+ 3 4))`? Circle the correct answer:

(a) `((+ 3 4) + 3 4)` (b) `(+ 3 4 + 3 4)` (c) `(7 7)`
(d) `(7 + 3 4)` (e) `((+ 3 4) (+ 3 4))`

.[**4 pts.**] Suppose the Lisp variable E has been given a value as follows:

```
(setf  E
     '((-1 -2) ((90 91) 92 93 94 95 96 97 98) (9 19 29 39 49 59 69 79 89)))
```

Write a LISP expression ***which does not involve any numbers***, but which evaluates to the list

```
(-2 91 (19 29 39 49 59 69 79 89))
```

Your expression may contain the variable E and any Lisp function calls.

6.(i)[**1 pt.**] Suppose a Lisp function `F` is defined as follows:

```
(defun f (u)
  (let* ((x 3)
         (y (+ x u)))
    (+ x y u)))
```

Which one of the following is a ***correct*** statement about the value of the expression `(F 4)`? Circle the correct answer:

(a) Its value is 12. (b) Its value is 13. (c) Its value is 14. (d) Its value is 15.
(e) Its value cannot be determined without more information.

(ii)[**1 pt.**] Suppose a Lisp function `G` is defined as follows:

```
(defun g (u)
  (let ((x 3)
        (y (+ x u)))
    (+ x y u)))
```

Which one of the following is a ***correct*** statement about the value of the expression `(G 4)`? Circle the correct answer:

(a) Its value is 12. (b) Its value is 13. (c) Its value is 14. (d) Its value is 15.
(e) Its value cannot be determined without more information.

7.[**2 pts.**] Complete the following definition of a Lisp function SAFE-AVG that takes 2 arguments and returns the *average* (i.e., the mean) of those 2 arguments if they are numbers. But if one or both of the arguments is not a number, then the function returns the symbol BAD-ARGS. Examples:

(SAFE-AVG 2.0 6.4) => 4.2 (SAFE-AVG 3 7) => 5

(SAFE-AVG '(23.1) 47.3) => BAD-ARGS (SAFE-AVG 'ONE 'TWO) => BAD-ARGS

Hint: You may want to use the built-in predicate function NUMBERP.

```
(defun safe-avg (m n)
```

8.[**2 pts.**] Write a Common Lisp function ~~OUR~~-REMOVE-IF such that, if f is any predicate that takes one argument and L is a list, then (remove-if f L) returns a list of all the elements of L which do ***not*** satisfy the predicate f. **Examples**:

```
(remove-if  #'oddp  NIL) => NIL
(remove-if  #'oddp       '(3  6  5  4))  =>    (6 4)
(remove-if  #'oddp    '(7  3  6  5  4))  =>    (6 4)
(remove-if  #'oddp    '(2  3  6  5  4))  =>  (2 6 4)
```

Hint: A Scheme version of this function is defined in Fig. 10.5 on p. 397 of Sethi (p. 7 of the course reader). ***However, you must write your function in Common Lisp.***

9.[**2 pts.**] Complete the following definition of a Common Lisp function INSERT-D such that, if *x* is a real number and *l* is a list of real numbers **in *descending* order**, then (INSERT-D *x l*) returns a list of numbers, also in ***descending*** order, that is obtained by inserting *x* in an appropriate position in the list *l*. **Hint**: There are two non-base cases—*x* may be *greater than or equal to* the 1st element of *l* (as in example B below), or *x* may be *less than* the first element of *l* (as in, e.g., example D).

Examples:

```
A. (INSERT-D   8    ( )) => (8)
B. (INSERT-D   9   '(6 4 2 0 -3))   =>  (9 6 4 2 0 -3)
C. (INSERT-D   1    '(4 2 0 -3))   =>    (4 2 1 0 -3)
D. (INSERT-D   1   '(6 4 2 0 -3))   =>  (6 4 2 1 0 -3)
```

```
(defun insert-d (x L)
   (cond ((endp L) (list x))
```

10. Let L be a Lisp variable whose value is: `(D B B A A A A C)`
Let X be a Lisp variable whose value is: `((2 B) (4 A) (1 C))`

Notice that the value of `(cdr X)` is `((4 A) (1 C))`.

Now write down the values of the following Lisp expressions; note that expression (e) involves L as well as X. Be sure to ***write parentheses just where they should be***: You will receive ***no credit*** if the answer is (E) and you write E, or if the answer is E and you write (E).

(a) `(car X)` _____ **[0.5 pt.]** (b) `(caar X)` _____ **[0.5 pt.]** (c) `(cadar X)` _____ **[0.5 pt.]**

(d) `(cons (list (+ (caar X) 1) (cadar X)) (cdr X))`

_____ **[0.5 pt.]**

(e) `(cons (list 1 (car L)) X)`

_____ **[0.5 pt.]**

11. **[2.5 pts.]** Fill in the 3 gaps in the following definition of a Lisp function COUNT-REPETITIONS such that if L is any ***nonempty*** list of atoms then (COUNT-REPETITIONS L) returns a list of pairs that indicate the number of repeated adjacent occurrences of each element of L, as shown in the following examples:

```
(COUNT-REPETITIONS '(W))   =>  ((1 W))
(COUNT-REPETITIONS   '(B B A A A A C))  =>          ((2 B) (4 A) (1 C))
(COUNT-REPETITIONS '(B B B A A A A C))  =>          ((3 B) (4 A) (1 C))
(COUNT-REPETITIONS '(D B B A A A A C))  =>    ((1 D) (2 B) (4 A) (1 C))
```

Hint: The correct answers to parts (d) and (e) of the previous question are relevant to this question!

```
(defun count-repetitions (L)
  (cond
    ((endp (cdr L)) (list (cons _____  L)))
    (t  (let ((X (count-repetitions (cdr L))))
          (if (equal (car L) (cadr L))
              _____
              _____ )))))
```