NOTE I compiled all the versions of the questions from what I could find. I have no way of knowing how many of each will be included especially with the new online versions

KEY: Ctrl+Click on each link to move to the section

FILL IN QUESTIONS

Write down the result of evaluating each of the following Lisp expressions. [Be sure to write parentheses where they should be and nowhere else: You will receive ***no credit*** if the correct answer is (X) and you write X, or if the answer is Y and you write (Y).]

Examples: (_____) '(1 4 3)) = 4 **A correct answer is: second**

(cdr_____) = (1 2 3) **A correct answer is: '(7 1 2 3) || An incorrect answer is: (7 1 2 3)**

a) (car '(P Q)) P
b) (cdr '(P Q)) (Q)
c) (cadr '(P Q)) Q
d) (list '(P Q R) 'S) ((P Q R) S)
e) (append '(P Q R) (list 'S)) (P Q R S)
f) (cons '(P Q R) (list 'S)) ((P Q R) S)
g) (car '(B)) = B
h) (cadr '(A B)) = B
i) (car '((B D) E)) = (B D)
j) (cons 'B '(D)) = (B D)
k) (cdr '(A B D)) = (B D)
l) (append '(B) '(D)) = (B D)
m) (cadr '(A (B D))) = (B D)
n) (list 'B 'D) = (B D)
o) (mapcar #'list '(B D)) = ((B) (D))
p) (mapcar #'car '((B) (D))) = (B D)
q) (mapcar #'cons '(2 4) '((B B) (D D))) = ((2 B B) (4 D D))
r) (mapcar #'+ '(2 4 6 8) '(100 100 100 100)) = (102 104 106 108)
s) (list 'A 'B 'C 'D) (A B C D)
t) (cons '(A B) '(C D)) ((A B) C D)
u) (list '(A B) '(C D)) ((A B) (C D))
v) (list '(A B C D)) ((A B C D))
w) (append '(A B) '(C D)) (A B C D)
x) (member 5 '(3 4 5 6)) (5 6)
y) (or (member 5 '(3 4 5 6)) (member 2 '(1 2 3))) (5 6)
z) (mapcar #'list '(1 2 3) '(4 5 6) '(A B C)) ((1 4 A) (2 5 B) (3 6 C))

aa) (mapcar (lambda (x) (cdr x)) '((A B C D) (E F G) (H I) (J))) ((B C D) (F G) (I) NIL)
bb) (mapcar #'list '(a b c) '(1 2 3) '(d e f)) ((A 1 D) (B 2 E) (C 3 F))
cc) (apply #'mapcar #'list '((A B C) (1 2 3))) ((A 1) (B 2) (C 3))
dd) (mapcar #'cons '(P Q R) '((A 6) (B 9) (C))) ((P A 6) (Q B 9) (R C))
ee) (mapcar #'car '((A B C D) (5 6 7 8) (10))) (A 5 10)

Suppose A is a variable whose value is 10 – e.g., because (SETF A 10) has just been evaluated. Write down the values of the following 2 expressions:

a) (LET ((A 12)
(B A)
B)
Answer: 10

b) (LET* ((A 12)
(B A)
B)
Answer: 12

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) (2 B)
b) (caar x) 2
c) (cadar x) B
d) (cons (list (+ (caar x) 1) (cadar x)) (cdr x)) ((3 B) (4 A) (1 C))
e) (cons (list 1 (car l)) x) ((1 D) (2 B) (4 A) (1 C))

This question is about lambda expressions

a) Write down the values of the following Common Lisp expression:
(mapcar (lambda (y) (+ y 9)) '(1 2 3 4)) **Answer:** (10 11 12 13)
b) Complete the following alternative definition of the function INC-LIST-2:
(This refers to the function where we add 2 to each element of the list)
(defun inc-list-2 (L n) (mapcar (lambda (y) (+ y n) **) L))**

What is the value of the Lisp expression (and T T T 2)?

Answer: 2

What is the value of the Lisp expression (and T T NIL (/ 4 0))?

Answer: NIL

Here is an infix expression: a-b\(12-sqrt(c-a)). As usual, -- has a lower precedence than \.

a) Rewrite this expression in *prefix* notation. Treat **sqrt** as an operator that takes 1 argument.
`- a \ b – 12 sqrt – c a`
b) Rewrite this expression in *postfix* notation. Treat **sqrt** as an operator that takes 1 argument.
`a b 12 c a – sqrt - \ -`

MULTIPLE CHOICE QUESTIONS

Just one of the following five definitions defines a Lisp function SWITCH such that the value of (SWITCH '(P Q)) is (Q P). Which one? Circle the correct answer

a) (defun switch(L) (list (cdr L) (car L))
b) (defun switch(L) (cons (cdr L) (car L)))
c) (defun switch(L) (append (cdr L) (car L)))
d) (defun switch(L) (list (cadr L) (car L)))
e) (defun switch(L) (cons (cadr L) (car L)))

Suppose the value of variable X is a ***list of length 2*** and the value of variable Y is a ***list of length 3***. For each of (i), (ii), (iii), and (iv) below, ***circle the choice*** that correctly describes the value of the corresponding expression. (**Note**: A "list of length *n*" means a list that has exactly *n* elements, each of which may itself be a list. For example, (Q (R (S T) U) (V W V W Q) Q) is a list of length 4.)

i. The value of (append X Y)
   a. Is a list of length 2
   b. Is a list of length 3
   c. Is a list of length 4
   d. Is a list of length 5
   e. Is an atom

ii. The value of (list X Y)
   a. Is a list of length 2
   b. Is a list of length 3
   c. Is a list of length 4
   d. Is a list of length 5
   e. Is an atom

iii. The value of (cons X Y)
   a. Is a list of length 2
   b. Is a list of length 3
   c. Is a list of length 4
   d. Is a list of length 5
   e. Is an atom

iv. The value of (mapcar #'list Y)
   a. Is a list of length 2
   b. Is a list of length 3
   c. Is a list of length 4
   d. Is a list of length 5
   e. Is an atom

Consider the problem of defining a Lisp function SAFE-AVG that takes 2 arguments and that returns the *average* (i.e., the mean) of those 2 arguments if they are numbers, but returns NIL if either argument is not a number. Here are examples of how the function should behave:

(safe-avg 3 7) → 5 (safe-avg 2 'dog) → NIL (safe-avg 'one 'two) → NIL

Here are four possible definitions of SAFE-AVG (some or all of which may be wrong):

Hint: Recall that evaluation of (+ x y) produces an error if either argument is not a number!

a) (defun safe-avg (m n) (and (numberp m) (numberp n) (/ (+ m n) 2)))
b) (defun safe-avg (m n) (and (numberp n) (numberp n) (/ (+ m n) 2)))
c) (defun safe-avg (m n) (and (/ (+ m n) 2) (numberp m) (numberp n)))
d) (defun safe-avg (m n) (and (numberp m) (/ (+ m n) 2) (numberp n)))

Write down the label (**A**, **B**, **C**, or **D**) of ***each correct*** definition of SAFE-AVG; if none of the definitions is correct, write ALL WRONG. **ANSWER:** A and B

Consider the following four statements about high-level and assembly languages

S1: In general, many more kinds of error and inconsistency in the code can be automatically detected before or during program execution when we program in typical high-level language than when we program in an assembly language.

S2: High-level language code is, in general, more concise and easier to write correctly than equivalent assembly code.

S3: High-level language code is, in general, considerably easier to read and understand, and hence to debug and modify, than equivalent assembly code.

S4: High-level language code will, in general, run much faster than equivalent assembly code, assuming that in both cases the code has been carefully written by expert programmers to run as fast as possible.

Exactly which of **S1**, **S2**, **S3**, and **S4** is/are ***true***? If none, write "NONE"; otherwise, write down the label (**S1**, **S2**, **S3**, or **S4**) of ***each*** true statement. **ANSWER:** S1, S2, S3

Just one of the following five statements about prefix and postfix notation is ***false***. Circle the ***false*** statement.

a) Prefix notation is like Lisp notation, but does ***not*** require parentheses.
b) In postfix notation, the operator that should be applied ***last*** is at the ***end***.
c) In prefix notation, the operator that should be applied ***last*** is at the ***beginning***.
d) Postfix notation allows operators whose ***arity is greater than*** 2.
e) In prefix and postfix notation, the meaning of an expression depends on the ***precedence classes*** to which different operators have been assigned

What is the value of the Lisp expression (second '((1 2 3) (4 5)))?

a) 2
b) (4 5)
c) 5
d) ((4 5))
e) (2 3)

What is the value of the Lisp expression (cdr '((1 2 3) (4 5)))?

a) 2
b) (4 5)

c) 5
d) ((4 5))
e) (2 3)

Consider the following Lisp expression:

(mapcar (lambda (x) (cons '(7) x)) '((A) (B C) (D)))

What is the value of this expression? Circle the answers:

a) (((7) A) ((7) B C) ((7) D))
b) (((7) (A)) ((7) (B C)) ((7) (D)))
c) ((7) (A) (B C) (D))
d) (7 (A) (B C) (D))

Suppose a Lisp function F is defined as follows

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

Which 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.

Suppose a Lisp function F is defined as follows

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

Which 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.

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))

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))

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))

Just one of the following 5 statements is ***false***. Circle the ***false*** statement.

a) Operator precedence classes may be used with infix notations but are ***not*** used with postfix and prefix notations
b) If # belongs to a ***higher*** precedence class than ^, and the precedence class of ^ is ***left***-associative, then the ^ operator between z and w is the root of the abstract syntax tree of the infix expression x # y ^ z ^ w.
c) If # belongs to a ***higher*** precedence class than ^, and the precedence class of ^ is ***right***-associative, then the ^ operator between y and z is the root of the abstract syntax tree of the infix expression x # y ^ z ^ w.
d) If # belongs to a ***lower*** precedence class than ^, then # is the root of the abstract syntax tree of the infix expression x # y ^ z ^ w
e) If # belongs to a ***higher*** precedence class than ^, then # ***must*** be applied ***first*** when evaluating the infix expression x ^ y ^ z # w

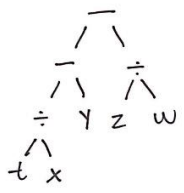
Just one of the following 5 statements is ***false***. Circle the ***false*** statement.

a) Since the Java binary operators "/" and "*" belong to the same (***left***-associative) precedence class, and Java's prefix unary operator "--" has ***higher*** precedence class than "/" and "*", the operator "--" must be applied ***first*** (i.e., must be applied ***before*** the operators "/" and "*" are applied) when the Java expression y / 2 / * --y is evaluated.
b) Operator precedence classes may be used with infix notations but are ***not*** used with postfix and prefix notations.
c) If # belongs to a ***higher*** precedence class than ^, and the precedence class of ^ is ***left***-associative, then the ^ operator between z and w is the operator applied last when evaluating the infix expression x # y ^ z ^ w.

d) If # belongs to a ***higher*** precedence class than ^, and the precedence class of ^ is ***right***-associative, then the ^ operator between y and z is the operator applied last when evaluating the infix expression x # y ^ z ^ w.
e) If # belongs to a ***lower*** precedence class than ^, then # is the operator that is applied last when evaluating the infix expression x # y ^ z ^ w.

When reading the infix notation expression in this question you should assume that, as usual, the ÷ operator belongs to a ***higher*** precedence class than the – operator, and the precedence classes that contain ÷ and – are ***left***-associative.

a) Which one of the following is the correct way to write (t – x ÷ y – z) ÷ w in ***prefix*** notation.
   a. ÷ - t - ÷ y x z w
   b. ÷ - - t ÷ x y z w
   c. ÷ - - t ÷ w x y z
   d. - ÷ t - ÷ y x z w
   e. - ÷ - t ÷ x y z w
b) Which one of the following is the correct way to write (t – x ÷ y – z) ÷ w in ***postfix*** notation.
   a. t x y ÷ - z – w ÷
   b. t x y - ÷ z – w ÷
   c. t x y ÷ - w – z ÷
   d. t x y - ÷ w – z ÷
   e. t x - y ÷ z – w ÷
c) Which one of the following is the correct way to write t ÷ x ÷ y – z - w in ***prefix*** notation.
   a. ÷ - - ÷ t x y z w
   b. ÷ - - ÷ t w x y z
   c. - ÷ - ÷ t x y z w
   d. - - ÷ ÷ t x y z w
   e. - ÷ - ÷ x y t z w
d) Which one of the following is the correct way to write t ÷ x ÷ y – z - w in ***postfix*** notation.
   a. t x ÷ y ÷ w - z -
   b. t x - y ÷ w ÷ z -
   c. t x ÷ y - z ÷ w -
   d. t x ÷ y z ÷ - w -
   e. t x ÷ y ÷ z - w -
e) On the right, draw an ***abstract syntax tree*** of the following expression:
   `t ÷ x - y - z ÷ w`



What is the value of the Lisp expression (list (cons ‘(A) ‘(B)) (append ‘(F) ‘(G)))? Circle the answer:

a) (A B F G)
b) (((A) B) (F G))
c) (((A) B) (F) (G))

d) (((A) B) ((F) (G)))
e) ((A B) ((F) G))

In Lisp, which of the following 5 expressions is the only correct way to write the expression ‘ ‘(A) using QUOTE rather than the ‘ character? Circle the correct answer:

a) (QUOTE QUOTE A)
b) (QUOTE (QUOTE A))
c) (QUOTE (QUOTE A)))
d) (QUOTE ((QUOTE A))))
e) ((QUOTE (QUOTE A))))

Suppose Rare Lisp is a language which is the same as Common Lisp except that, whereas arguments of function calls in Common Lisp are evaluated in left-to-right order, arguments of function calls in Rare Lisp may be evaluated in any order. (In respect Rare Lisp would be like Scheme and C++, whereas Common Lisp is like Java.) What happens when the expression

```
(+    (setf x 2)    x    (setf x 4))
```

is evaluated by a Rare Lisp interpreter? Circle the correct answer:

a) If the arguments of + are evaluated in left-to-right order then the value 8 will be returned; if the arguments are evaluated in right-to-left order, then the value 10 will be returned.
b) If the arguments of + are evaluated in left-to-right order then the value 10 will be returned; if the arguments are evaluated in right-to-left order, then the value 8 will be returned.
c) An evaluation error will occur regardless of the order in which the arguments of + are evaluated.
d) The value 8 will be returned regardless of the order in which the arguments of + are evaluated.
e) The value 10 will be returned regardless of the order in which the arguments of + are evaluated.

Suppose you have just entered the following three expressions at successive Clisp > prompts (with no spaces before and after * and + in 8*7 and 8+7):

```
(setf 8*7 5)

(defun 8+7 (x y) (+ x y))

(defun 8*7 () 2)
```

If you now enter the expression `(8+7 (* 8 7) (8+7 (8*7) 8*7))` at the Clisp > prompt what value will be printed by Clisp? Circle the correct answer:

a) 61
b) 63
c) 114
d) 115
e) 168

What is the value of the expression **(mapcar (lambda (x) (* x x 5)) ‘(0 1 2 3))**?

a) (0 5 20 45)
b) 0
c) (0 1 2 3)
d) (0 1 1 5)

e) NIL

CREATE FROM LIST

****NOTE**** *There are different versions of the following answers that are also correct*

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 ***that 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.

Answer (append (list (cadar e) (cadadr e)) (list (cdaddr e))))

Answer (list (cadar e) (cadr (caadr e)) (cdaddr e))

(Suppose the Lisp variable E has been given a value as follows:

(setf E '((A B 92 93 94 95 96) -1 (W 19 29 39 49 59)))

Write a LISP expression ***that does <u>not</u> involve any numbers*** and ***does <u>not</u> use*** ' (or QUOTE), and that evaluates to a list **((93 94 95 96) (-1 19 29 39 49 59))**.

Answer (list (cdddr (car e)) (append (cons (cadr e) nil) (cdr (caddr e))))

Suppose the Lisp variable E has been given a value as follows:

(setf E '((A B 92 93 94 95 96) (19 29 39 49 59) -1))

Write a LISP expression ***that does <u>not</u> involve any numbers*** and ***does <u>not</u> use*** ' (or QUOTE), and that evaluates to a list **(A (19 29 39 49 59) (-1 92 93 94 95 96))**.

Your expression may contain the variable E and may call any of Lisp's built-in functions.

Answer (append (list (caar e) (second e)) (list (append (cddr e) (cddar e))))

Suppose the Lisp variable X has been given a value as follows:

(setf X '(1 (7 17 27 37) 2 (90 91 92 93)))

Write a LISP expression ***that does <u>not</u> involve any numbers***, but which evaluates to the list

((90 91 92 93 7 17 27 37) (1 2))

Your expression may contain the variable X and calls of any built-in Lisp functions. Some examples of built-in functions are: FIRST/CAR, REST/CDR, SECOND, THIRD, FOURTH, CONS, APPEND, LIST

Answer (list (append (cadddr x) (cadr x)) (list (car x) (caddr x)))

Suppose that e is a Lisp variable whose value is (A (B (C Y) Z) X). Note that (car (car (cdr e))) is an expression that involves ***only*** e, car, and cdr, and whose value is the symbol B.

NOTE This means don't combine (car (car e)) into (caar e)

i. Write an expression that involves ***only*** e, car, and cdr, and whose value is symbol X. If you prefer, you may use first and rest instead of car and cdr.
(car (cdr (cdr e)))

ii. Write an expression that involves ***only*** e, car, and cdr, and whose value is symbol Y. If you prefer, you may use first and rest instead of car and cdr.
(car (cdr (car (cdr (car (cdr e))))))

iii. Write an expression that involves ***only*** e, car, and cdr, and whose value is symbol Z. If you prefer, you may use first and rest instead of car and cdr.
(car (cdr (cdr (car (cdr e)))))

Suppose that e is a Lisp variable whose value is (A B X ((C D) Y) (Z D)). Note that (car (cdr e)) is an expression that involves ***only*** e, car, and cdr, and whose value is the symbol B.

i. Write an expression that involves ***only*** e, car, and cdr, and whose value is symbol X. If you prefer, you may use first and rest instead of car and cdr.
(car (cdr (cdr e)))

ii. Write an expression that involves ***only*** e, car, and cdr, and whose value is symbol Y. If you prefer, you may use first and rest instead of car and cdr.
(car (cdr (car (cdr (cdr (cdr e))))))

iii. Write an expression that involves ***only*** e, car, and cdr, and whose value is symbol Z. If you prefer, you may use first and rest instead of car and cdr.
(car (car (cdr (cdr (cdr (cdr e))))))

Suppose the Lisp variable E has the value ((A) ((X) (Y (Z X)))) – e.g., suppose we have just entered (setf E '((A) ((X) (Y (Z X))))) at a clisp prompt. Then the expression (car (car (car (cdr E)))) = (first (first (second E))) is an expression that involves only E, car, and cdr, and which evaluates to the symbol X.

i. Write another expression that involves only E, car, and cdr, but which evaluates to the symbol Y. [**Note**: You must write an expression that uses no function other than car and cdr; if you first write an expression that uses some other function (such as second or cadr), then you must also write an equivalent expression that uses no function other than car and cdr.]
(car (car (cdr (car (cdr e)))))

ii. Write an expression that involves ***only*** e, car, and cdr, but which evaluates to the symbol Z. [The above **Note** applies here as well.]
(car (car (cdr (car (cdr (car (cdr e)))))))

Suppose the Lisp variable E has been given a value as follows:

(setf E '(((9 19 29) 39 49 59 69 79) ((111 222) (90 91 92 93 94))))

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

((90 91 92 93 94) (111 222 39 49 59 69 79))

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

Answer (list (cadadr e) (append (caadr e) (cdar e)))

Suppose the Lisp variable X has been given a value as follows:

(setf X '(88 ((11 21 31 41)) (90 91 92 93)))

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

(11 (11 21 31 41) (88 90 91 92 93))

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

Answer (cons (caaadr e) (list (caadr e) (cons (car e) (caddr e))))

LISP FUNCTIONS ***either fill in or write completely || scheme to lisp translation***

Without using COND, IF, WHEN, UNLESS, or **CASE**, complete the following definition of a Lisp function GT-INT that has the following properties: GT-INT takes 2 arguments; it returns T if both arguments are integers and the first argument is greater than the second. In all other cases, GT-INT returns NIL. **Hint**: You may want to use the function INTEGERP. Examples:

(GT-INT 9 3) → T (GT-INT 3 3) → NIL (GT-INT 9.1 3.2) → NIL (GT-INT '(9) 3) → NIL

Reminder: *You must write this function <u>without</u> using* COND, IF, WHEN, UNLESS, or **CASE!**

(defun gt-int (m n)

(and (integer m) (integer n) (> m n)))

<u>Without</u> using APPLY, complete the following definition of a LISP function SUM such that if L is a list of numbers then (SUM L) returns the sum of the elements of L.

Examples: (SUM ()) → 0

(SUM '(2 3 6 4 6)) → 21

(SUM '(5 2 3 6 4 6)) → 26

(defun sum (L)

(if (null L)

```
        0
        (let ((x (sum (cdr L))))
            (+ (car L) x))))
```

ALTERNATIVE from different exam

```
(defun sum (L)
    (cond ((null L) 0 )
        (t (+ (car L) (sum (cdr L)))
```

Use APPLY and #'+ to complete the following alternative definition of the function SUM. Recall that (sum '(a b c d)) = (+ a b c d) for any numbers a, b, c ... d. Your definition must not be recursive.

```
(defun sum (L)  (apply #'+ L))
```

Suppose Lisp functions SPLIT-LIST and MERGE-LISTS with the following properties ***have <u>already</u> been defined***:

If L is a list, then (SPLIT-LIST l) returns a ***list of 2 lists***, in which the first list consists of the 1st, 3rd, 5th, … elements of L, and the second list consists of the 2nd, 4th, 6th, … elements of L.

Example: (SPLIT-LIST '(4 5 3 1 6 8 7 0 3)) returns ((4 3 6 7 3) (5 1 8 0))

If each of L1 and L2 is a list of real numbers in ascending order, then (MERGE-LISTS L1 L2) returns the list of numbers in ascending order that is obtained by merging L1 and L2.

Example: (MERGE-LISTS '(3 3 4 6 7) '(0 1 5 8)) returns (0 1 3 3 4 5 6 7 8).

Fill in the 4 gaps in the following recursive Lisp function MSORT so that if L is a nonempty list of real numbers then (MSORT L) returns a list consisting of the elements of L in ascending order/

Example: (MSORT '(7)) should return (7)

(MSORT '(4 5 3 1 6 8 7 0 3)) should return (0 1 3 3 4 5 6 7 8).

```
(defun msort (L)
    (cond ((endp (cdr L)) L)
        (t (let* (SL (split-list L))
            (X (msort (car SL)))
            (Y (msort (cadr SL) )))
        (merge-lists x y )))))
```

In this question the term ***set*** means a list in which ***no element occurs more than once.*** Complete the following definition of a Common Lisp function SET-UNION such that if each of S1 and S2 is a ***set*** of atoms then (SET-UNION S1 S2) returns a ***set*** of all atoms that occur in at least one of S1 and S2. (***No atom should occur more than once in the returned result.***) Examples:

1. (SET-UNION () ‘(A X 7 F G)) => (A X 7 F G)
2. (SET-UNION ‘(Y P A B) ‘(A X 7 F G)) => (Y P B A X 7 F G)
3. (SET-UNION ‘(7 Y P A B) ‘(A X 7 F G)) => (Y P B A X 7 F G)
4. (SET-UNION ‘(3 Y P A B) ‘(A X 7 F G)) => (3 Y P B A X 7 F G)

```
(defun set-union (L1 L2)
    (if (endp L1)
        L2                                          : base case
        (let ((X (set-union (cdr L1) (L2)))
            (if (member (car L1) L2)
                X                                   : see example 3
                (cons (car L1) X) ))))              : see example 4
```

Complete the following definition of a recursive Lisp function SPLIT-NUMS such that if N *is a non-negative integer* then (SPLIT-NUMS N) returns a list of two lists: The first of the two lists consists of the ***odd*** integers between 0 and N in descending order, and the second list consists of the ***even*** integers between 0 and N in descending order. Examples:

(SPLIT-NUMS 0) => (NIL (0))

(SPLIT-NUMS 7) => ((7 5 3 1) (6 4 2 0))

(SPLIT-NUMS 8) => ((7 5 3 1) (8 6 4 2 0))

(SPLIT-NUMS 9) => ((9 7 5 3 1) (8 6 4 2 0))

```
(defun split-nums (n)
    (if (zerop n)
        ‘(nil (0))
        (let ((x (split-nums (- n 1))))
            (if (evenp n)
                (list (car x) (cons n (cadr x))
                (list (cons n (car x)) (cadr x))))))
```

Write a Common Lisp function OUR-MAPCAR such that if f is a function of *one* argument and L is a list then (our-mapcar f L) returns the same list as (mapcar f L). **Examples:**

(our-mapcar #'evenp NIL) => NIL

(our-mapcar #'evenp '(7 5 4)) => (NIL NIL T)

(our-mapcar #'evenp '(2 7 5 4)) => (T NIL NIL T)

(our-mapcar #'evenp '(3 7 5 4)) => (NIL NIL NIL T)

Hint: A Scheme version of this function is defined at the top of p. 400 of Sethi (p. 16 of the course reader). However, you must write your function in Common Lisp.

SCHEME VERSION

```
(define (map f x)
    (cond ((null? X) '())
        (else (cons (f (car x)) (map f (cdr x)))) ))
```

LISP ANSWER

```
(defun our-mapcar (f L)
    (cond ((endp L) '())
        (t (cons (funcall f (car L)) (our-mapcar f (cdr L))))))
```

Write a Common Lisp function 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 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 at the top of p. 397 of Sethi (p. 13 of the course reader). ***However, you must write your function in Common Lisp.***

SCHEME VERSION

```
(define (remove-if f x)
    (cond ((null? x) '())
        ((f (car x)) (remove-if f (cdr x)))
        (else (cons (car x) (remove-if f (cdr x)))) ))
```

LISP ANSWER

```
(defun remove-if (f L)
```

```
(cond ((endp L) nil)
      ((funcall f (car L)) (remove-if f (cdr L)))
      (t (cons (car L) (remove-if f (cdr L))))))
```

Complete the following definition of a Lisp function SAFE-AVG that takes 2 arguments and returns the *average* (i.e., the mean) of those two 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. **Example:**

(SAFE-AVG 2.0 6.4) => 4.2 (SAFE-ARGS 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)
      (if (and (numberp m) (numberp n))
            (/ (+ m n) 2)
            ‘BAD-ARGS))
```

Define a Common Lisp function MEMBER-2 such that if K is a symbol and X is a list, then (member-2 K X) returns T if the symbol is an element of the list, and returns NIL if the symbol is not an element of the list. **Examples:** (member-2 ‘A ‘(C A B C D)) => T (member-2 ‘A ‘(B C D)) => NIL

Hint: A Scheme analog of this function is defined at the bottom of p. 22 / p. 417 of the course reader, though there should be an opening parenthesis before define. However, you must write a Common Lisp function. For example, else has no predefined meaning in Common Lisp, and so your function must ***not*** use a COND clause of the form (else …)

SCHEME VERSION

```
define (member? k x)
      (cond ((null? x) #f)
            ((eq? k (car x)) #t)
            (else (member? k (cdr x))) ))
```

LISP ANSWER

```
(defun member-2 (K X)
      (cond ((endp x) NIL)
            ((equal k (car x)) t)
            (t (member-2 k (cdr x)))))
```

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 2 non-base cases – x may be *greater than or equal to* the first element of *l* (as in example B), or *x* may be *less than* the first element of *l* (as in example D).

Example:

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))
          ((>= x (car L)) (cons x L))
          (t (let ((N (insert-d x (cdr L))))
               (cons (car L) N))))))
```

Define a Common Lisp function `less` such that if *k* is a real number and *x* is a list of real numbers then (less *k x*) returns a list that can be obtained from the list *x* by removing all the numbers that are > *k*. Thus `(less 5 '(7 3 5 2 1 9 7 4))` should return `(3 2 1 4)`.

Hint: A Scheme analog of this function is defined on p. 23 / p. 418 (in Exercise 10.3). However, you must write a Common Lisp function. For example, `else` has no predefined meaning in Common Lisp, and so your function must ***not*** use a COND clause of the form `(else …)`.

SCHEME VERSION

```
define (less k x)
    (cond ((null? x) '())
          ((<  (car x) k) (cons (car x) (less k (cdr x))))
          (else (less k (cdr x))) ))
```

LISP ANSWER

```
(defun less (K X)
    (cond ((endp x) '())
          ((< (car X) K) (cons (car X) (less K (cdr X))))
          (t (less K (cdr X)))))
```

Complete the following definition of a Lisp function SWITCH such that if L is any two-element list, then (switch L) returns a list consisting of the same two elements, but in the opposite order.

Example: (switch '(A B)) should return (B A).

```
(defun switch (L) (list (cadr L) (car L)))
```

Complete the following definition of a Lisp function ROTATE-LEFT such that if L is any nonempty list, then (rotate-left L) returns a new list of the same length whose last element is the first element of L, and who's k^{th} element is the k+1st element of L if $1 \leq k \leq$ the length of L.

Example: (ROTATE-LEFT '(A B C D E F G)) should return (B C D E F G A).

```
(defun rotate-left (L) (append (cdr L) (list (car L))))
```

Complete the following definition of a Lisp function MONTH->INTEGER which takes as argument a symbol that should be the name of a month, and which returns the number of the month.

Examples: (MONTH->INTEGER 'MAY) => 5 (MONTH->INTEGER 'JULY) => 7

If the argument is not a symbol, or if it is a symbol that is not the name of a month, then your function must return the symbol ERR.

Examples: (MONTH->INTEGER '(MAY)) => ERR (MONTH->INTEGER 'J) => ERR

```
(defun month->integer (m)
    (cond   ((eq m 'january)  1)
            ((eq m 'febuary)  2)
            ((eq m 'march)    3)
            ((eq m 'april)    4)
            ((eq m 'may)      5)
            ((eq m 'june)     6)
            ((eq m 'july)     7)
            ((eq m 'august)   8)
            ((eq m 'september) 9)
            ((eq m 'october)  10)
            ((eq m 'november) 11)
            ((eq m 'december) 12)
            (t 'ERR)))
```

Without using MAPCAR, write a Lisp function INC-LIST-2 such that if L is a list of numbers and N is a number then (INC-LIST-2 L N) returns a list, of the *same length* as L, in which each element is equal to (N + the corresponding element of L). Examples:

(INC-LIST-2	()	7)	=>	NIL
(INC-LIST-2	‘(1 20 3 9)	7)	=>	(8 27 10 16)
(INC-LIST-2	‘(4 1 20 3 9)	7)	=>	(11 8 27 10 16)

```
(defun inc-list-2 (L n)
        (if (null L)
                nil
                (let ((x (inc-list-2 (cdr L) n)))
                        (if (car L) (append (list (+ (car L) n)) x) nil))))
```

Suppose you have ***already written*** a Common Lisp function INSERT such that, if N is any real number and L is any list of real numbers in ***ascending*** order, then (INSERT N L) returns the list of numbers in ***ascending*** order that is obtained by placing N in an appropriate position in L. Thus (INSERT 4 ‘(0 1 3 7 7 8)) => (0 1 3 4 7 7 8) and (INSERT -1 ‘(0 1 3 7 7 8)) => (-1 0 1 3 7 7 8). Use INSERT to complete the following definition of a function ISORT such that, if L is a list of real numbers, then (ISORT l) is a list consisting of the elements of L in ***ascending*** order. [You are ***not*** expected to define the function INSERT.]

Examples: (ISORT ()) => NIL

(ISORT ‘(3 7 0 8 1 7)) => (0 1 3 7 7 8)

(ISORT ‘(4 3 7 0 8 1 7)) => (0 1 3 4 7 7 8)

```
(defun isort (L)
        (if (endp L)
                nil
                (let ((x (isort (cdr L))))
                        (insert (car L) x) )))
```

Show how the definition of **isort** in the previous problem can be written without using LET.

```
(defun isort (L)
        (if (endp L)
                nil
                (insert (car L) (isort (cdr L)))))
```

Complete the following definition of a Lisp function MULTIPLE-MEMBER that takes 2 arguments and behaves as follows whenever the 1st arguments is a symbol or number and the 2nd argument is a list; if the symbol or number occurs ***at least twice*** in the list, then MULTIPLE-MEMBER returns the part of the list that begins with the ***2nd*** occurrence of that atom; otherwise MULTIPLE0MEMBER returns NIL. You may use the built-in function MEMBER.

Examples:

(MULTIPLE-MEMBER 'A '(B A B B A C A A B)) => (A C A A B)

(MULTIPLE-MEMBER 'A '(B A B B)) => NIL

(MULTIPLE-MEMBER 'A '(B B D)) => NIL

```
(defun multiple-member (a b)
    (cond
        ((not (listp b)) nil)
        ((not (or (symbolp a) (numberp a))) nil)
        (t (member a (cdr (member a b))))))
```

Write a ***tail-recursive*** Common Lisp function REV such that if X and Z are lists then (REV X Z) returns the list obtained by appending the *reverse* of the list X to the list Z. For example, (REV '(1 2 3 4) '(A B C)) => (4 3 2 1 A B C) and (REV '(1 2 3 4 5) NIL) => (5 4 3 2 1).

There is a Scheme version of this function near the top of p 397 of Sethi (which is on p 13 of your course reader), but *you must translate that definition into Common Lisp*.

SCHEME VERSION

```
(define (rev x z)
    (cond ((null? x) z)
        (else (rev (cdr x) (cons (car x) z))) ))
```

LISP ANSWER

```
(defun rev (x z)
    (cond ((endp x) z)
        (t (rev (cdr x) (cons (car x) z)))))
```

Write a Common Lisp predicate ODD-GT-MILLION that takes one argument, and which returns T if its argument is an ***odd*** integer greater than a million, but returns NIL otherwise. Some of the following built-in predicates might be useful in writing this function: >, <=, EVENP, INTEGERP, ODDP. [You are not required to use any of these functions, and you are free to use other Lisp functions.]

Examples: (ODD-GT-MILLION 2010231.23) => NIL (ODD-GT-MILLION 2010230) => NIL

(ODD-GT-MILLION 11) => NIL (ODD-GT-MILLION 2010231) => T

```
(defun odd-gt-million (n)
    (if (and (integer n) (> n 1000000) (oddp n))
        'T
        NIL))
```

Complete the following definition of a Lisp function REPEATED-ELTS such that, if L is any list of numbers, then (REPEATED-ELTS L) returns a list that contains one element from each "run" of two or more equal numbers, but contains no other elements. For example:

(REPEATED-ELTS '(1 2 3 3 9 2 2 2 1 1 3 1 3 2 2 7 8 3 3 3 3 4)) => (3 2 1 2 3)

Here the argument of REPEATED-ELTS contains 5 "runs" of two or more equal numbers; these 5 runs have been underlined above. Thus REPEATED-ELTS returns a list that consists of one element from each of these 5 runs. Further examples:

(REPEATED-ELTS '(4 9 9 8 7 7)) => (9 7)

(REPEATED-ELTS '(6 4 9 9 8 7 7)) => (9 7)

(REPEATED-ELTS '(2 6 4 9 9 8 7 7)) => (9 7)

(REPEATED-ELTS '(6 6 4 9 9 8 7 7)) => (6 9 7)

```
(defun repeated-elts (L)
    (cond
        ((endp L) nil)
        ((not (equal (car L) (cadr L)))          ; L = (x y . . .) or L = (x)
            (repeated-elts (cdr L)))
        ((not (equal (car L) (caddr L)))         ; L = (x x y . . .) or L = (x x)
            (cons (car L) (repeated-elts (cddr L))))
        (t (repeated-elts (cdr L)))))            ; L = (x x x . . .)
```

Write a Common Lisp function SPLIT-LIST such that if L is a list, then (SPLIT-LIST L) returns a list of two lists, in which the first list consists of the 1st, 3rd, 5th, … elements of L, and the second list consists of the 2nd, 4th, 6th, … elements of L.

Examples:

(split-list ()) => (NIL NIL)

(split-list '(B C D 1 2 3 4 5)) => ((B D 2 4) (C 1 3 5))

(split-list '(A B C D 1 2 3 4 5)) => ((A C 1 3 5) (B D 2 4))

```
(defun split-list (L)
    (if (endp L)
        '(() ())
        (let ((x (split-list (cdr L))))
            (list (cons (car L) (cadr x)) (car x)) )))
```

Using ***neither*** COND ***nor*** IF, define a Lisp function SAFE-DIVIDE with the following properties: SAFE-DIVIDE takes 2 arguments. If *both arguments are numbers* ***and*** *the 2nd argument is* ***not*** *equal to* 0, SAFE-DIVIDE returns the result of using / to divide the 1st argument by the 2nd argument; in all other cases, it returns NIL. **Reminder: Use neither COND nor IF**!

Examples: (safe-divide 4 2) => 2 (safe-divide 4 0) => NIL (safe-divide 'four 2) => NIL

```
(defun safe-divide (a b)
    (and (numberp a) (numberp b) (not (= b 0) (/ a b))
```

MIN-FIRST is a Lisp function that is already defined on Euclid and venus; if L *is any nonempty list of real numbers* then (MIN-FIRST L) returns a list whose CAR is the *minimum* of the numbers in L and whose CDR is the list obtained when the first occurrence of that minimum value is removed from L. (Thus (MIN-FIRST L) is "L with the first occurrence of its minimum value moved to the front if it is not there already".) For example:

(MIN-FIRST '(0 3 4 1)) => (0 3 4 1) (MIN-FIRST '(4 1 0 3 5 0 2)) => (0 4 1 3 5 0 2)

SSORT is another function that is already defined on euclid and venus; if L *is any list of real numbers* then (SSORT L) is a list of those same numbers in ascending order. For example:

(SSORT '(4 1 0 3 5 0 2) => (0 0 1 2 3 4 5) (SSORT '(4 1 3 5 0 2)) => (0 1 2 3 4 5)

Complete the following definition of a function MY-SSORT ***without making further calls*** SSORT and ***without*** calling MY-SSORT recursively, in such a way that if L is any ***nonempty*** list of real numbers then (MY-SSORT L) is equal to (SSORT L).

```
(defun my-ssort (L)
    (let* ((L1 (min-first L))
        (x ((ssort (cdr L1))))
        (cons (car L1) x) ))
```

Would the definition of MY-SSORT in the previous question be correct if let* were changed to let? Justify your answer.

Answer from previous test

No. Because then the parameters would be bound in parallel. Thus (cdr L1) would not yield the desirable value since L1 is not set to (min-first L) before the cdr value is evaluated…

Answer continues but is cut off in image. Overall answer is correct.

Now suppose the function SSORT used in the previous question is ***not*** already defined. Show how you can define a recursive Lisp function SSORT with the same properties.

```
(defun SSORT (L)
    (if (endp L)
        '()
        (let* ((L1 (min-first L))
               (x (ssort (cdr L1))))
               (cons (car L1) x))))
```

Without using COND, IF, WHEN, UNLESS, or CASE, complete the following definitions of a Lisp function SAME-SIGN-INT that has the following properties: SAME-SIGN-INT takes two arguments. It returns T if both arguments are integers and they are either both positive, or both negative, or both equal to zero. In *all other cases* it returns NIL. **Hint**: The predicate INTEGERP can be used to test whether an argument is an integer. **Reminder**: Do ***not*** use COND, IF, WHEN, UNLESS, or CASE! **Examples**:

(SAME-SIGN-INT 'CAT 'DOG) => NIL

(SAME-SIGN-INT 0 2) => NIL

(SAME-SIGN-INT -3 9) => NIL

(SAME-SIGN-INT -3 -9) => T

(SAME-SIGN-INT 0 0) => T

(SAME-SIGN-INT 4 7) => T

```
(defun same-sign-int (x y)
    (and (integer x) (integerp y)
        (or (and (zerop x) (zerop y))
            (and (> 0 x) (> 0 y))
            (and (< 0 x) (< 0 y)))))
```

Complete the definition below of a Lisp function QUADRATIC-1 that has the following properties: QUADRATIC-1 takes 3 arguments, which are assumed to be numbers. If the values of the 1st, 2nd, and 3rd arguments are respectively A, B, and C, then QUADRATIC-1 returns the value of the following expression:

$$\frac{-B + \sqrt{B^2 - 4ac}}{2a}$$

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

I know the quadratic function uses +/- but for this question, he specifically put +

```
(defun quadratic-1 (a b c)
    (if (and (numberp a) (numberp b) (numberp c))
        (/ (+ (-B) (sqrt (- (* b b) (* 4 a c)))) (* 2 a))
        'ERROR))
```

Complete the following definition of a Lisp function PARTITION such that if L is a list of real numbers and P is a real number then (PARTITION L P) returns a list whose CAR is a list of the elements of L that are strictly less than P, and whose CADR is a list of the other elements of L. Each element of L must appear in the CAR or CADR of (PARTITION L P), and should appear there just as many times as it appears in L. Examples:

```
(partition          () 5)                    =>          (NIL NIL)
(partition      '(0 11 8 4 9 6 11 8 10) 9)   =>   ((0 8 4 6 8) (11 9 11 10))
(partition   '(15 0 11 8 4 9 6 11 8 10) 9)   =>   ((0 8 4 6 8) (11 9 11 10))
(partition    '(3 0 11 8 4 9 6 11 8 10) 9)   =>   ((0 8 4 6 8) (11 9 11 10))
(defun partition (L P)
    (if (endp L)
        '(() ())
        (let ((x (partition (cdr L) P)))
            (if (< (car L) P)
                (list (cons (car L) (car x)) (cadr x))
                (list (car x) (cons (car L) (cadr x)) ))))))
```

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))
(defun count-repetitions (L)
    (cond
```

```
((endp (cdr L)) (list (cons 1 L)))
(t      (let ((x (count-repetitions (cdr L))))
              (if (equal (car L) (cadr L))
                  (cons (list (+ (caar x) 1) (cadar x)) (cdr x))
                  (cons (list 1 (car L)) x) )))
```

ABSTRACT EXPRESSIONS **Includes AST, infix, postfix, etc. **

Infix expressions in a certain language use the following three precedence classes:

	Postfix unary operators	**Prefix** unary operators	**Binary** operators	
Class 1	* ^	&	@	***Left***-associative
Class 2		~	$	***Left***-associative
Class 3			#	***Right***-associative

Class *i* operators have higher precedence rank than class *i*+1 operators – ***class 1 is the highest precedence class, and class 3 is the lowest precedence class.***

i. Using the above information, ***circle*** the operator that should be applied last in the following expression:

x #$_{3R}$ (&$_{1L}$ y ^$_{1L}$ #$_{3R}$ z) $$_{2L}$ w #$_{3R}$ u *$_{1L}$

[To help you; subscripts have been attached to each operator to indicate its precedence class and whether that class is left-associative (L) or right-associative (R), but *this information can also be obtained from the above table*]

ii. Using the above information, ***circle*** the operator that should be applied last in the following expression:

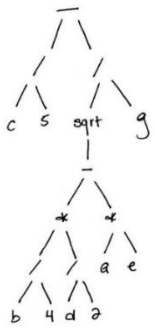
&$_{1L}$ y ^$_{1L}$ $$_{2L}$ z $$_{2L}$ w $$_{2L}$ u

iii. Using the above information, ***circle*** the operator that should be applied last in the following expression:

~$_{2L}$ y $$_{2L}$ z $$_{2L}$ w #$_{3R}$ @$_{1L}$

Draw an ***abstract syntax tree*** of the following expression – treat sqrt as an operator with once argument:

```
c / 5 – sqrt ((b / 4) * (d / 2) – a * e) / g
```



Rewrite the expression in ***prefix*** notation

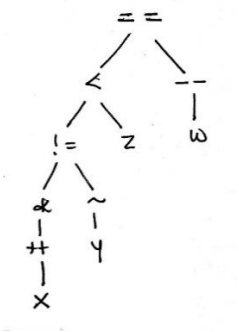
```
- / c 5 / sqrt - * / b 4 / d 2 * a e g
```

In a certain language (a simple subset of C++) expressions are written in infix notation with the following operator precedence classes:

Class 1	++	--	(unary postfix operators)	Left-associative
Class 2	*	~	(unary prefix operators)	Right-associative
Class 3	<	>	(binary operators)	Left-associative
Class 4	==	!=	(binary operators)	Left-associative

Class 1 is the highest precedence class and class 4 is the lowest precedence class.

Draw an abstract syntax tree of the expression `(* x ++ != ~ y) < z == w –`



In certain languages, expressions are written <u>infix notation</u>. The language has binary operators and a unary prefix operator, whose precedence classes are as follows:

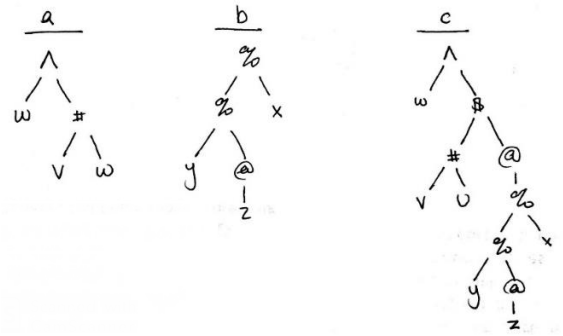
	binary operators	unary prefix operators	associativity
Class 1	#	@	Left
Class 2	%	[none]	Left
Class 3	$ ^	[none]	right

Class 1 operators have the ***<u>highest</u>*** precedence and class 3 operators have the ***<u>lowest</u>*** precedence

i) Here are 3 expressions in this language; in each case, ***circle the operator that is <u>applied last:</u>***

```
a. w ^ v # u
b. y % @ z % x
c. w ^ v # u $ @ (y % @ z % x)
```

ii) Draw an abstract syntax tree of ***each*** of the above expressions



iii) Rewrite the first expression of part (i) in ***prefix*** notation
`^ w # v u`

iv) Rewrite the second expression of part (i) in ***prefix*** notation
`% % y @ z x`

v) Rewrite the third expression of part (i) in ***prefix*** notation
`^ w $ # v u @ % % y @ z x`

Here is an expression in infix notation: `sqrt (x - 9 ÷ y) – z`. As usual, ÷ has a higher precedence than -. (Regard sqrt as a unary operator that has higher precedence than -.)

a) Rewrite this expression in ***Lisp*** notation
ANSWER `(- (sqrt (- x (÷ 9 y))) z)`

b) Rewrite this expression in ***prefix*** notation
ANSWER `- sqrt - x ÷ 9 y z`

c) Rewrite this expression in ***postfix*** notation
ANSWER `x 9 y ÷ - sqrt z -`

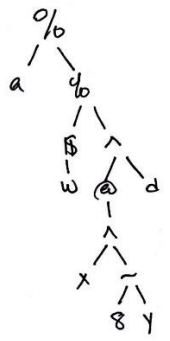
In a certain language, expressions are written in infix notation. The language has binary, prefix unary, and postfix unary operators that belong to the following precedence classes:

	Binary ops	**Prefix unary ops**	**Postfix unary ops**	**Associativity**
1st Class:	# ~	~	[none]	Right
2nd Class:	[none]	[none]	$	Left
3rd Class:	% ^	@	[none]	Right

1st class operators have the ***highest*** precedence and 3rd class operators have the ***lowest*** precedence.

a) Draw the abstract syntax tree of the following expression. **Note**: To help you understand this expression, *each operator has been given subscripts that indicate its precedence class* (1, 2, or 3) *and whether its class is left-associative* (L) *or right-associative* (R), even though this information can be deduced from the above table. For example, the subscripts in $^{}_{3R}$ tell you ^ is in the 3rd class and that class is right-associative.

`a %3R w $2L %3R (@3R x ^3R 8 ~1R y) ^3R d`



Rewrite the expression in ***prefix*** notation.

`% a % $ w ^ @ ^ x ~ 9 y d`

Rewrite the above expression in ***postfix*** notation.

`a w $ x 8 y ~ ^ @ d ^ % %`

Compute the value of the following expression (which is written in prefix notation)

$$+_4\ 3\ -_2\ *_3\ 3\ -_2\ 4\ 2\ 5\ 7\ -_2\ 3\ 1\ 3$$

assuming that $+_4$ denotes the addition operator of arity 4 (e.g., the value of $+_4\ 3\ 2\ 7\ 5$ is 17), $*_3$ denotes the multiplication operator of arity 3 (e.g., the value of $*_3\ 3\ 2\ 7$ is 42), and $-_2$ denotes the subtraction operator of arity 2 (e.g., the value of $-_2\ 7\ 3$ is 4). Circle the correct value:

The value is: a) -19 b) -3 c) 25 d) 29 e) 31

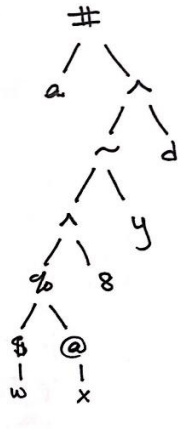
In a certain language, expressions are written in infix syntax. The language has binary, prefix unary, and postfix unary operators that belong to the following precedence classes:

	Binary ops	**Prefix unary ops**	**Postfix unary ops**	**Associativity**
1st Class:	# ~	[none]	[none]	Right
2nd Class:	[none]	@	$	Left
3rd Class:	% ^	[none]	[none]	Left

1st class operators have the ***highest*** precedence and 3rd class operators have the ***lowest*** precedence.

a) Draw the abstract syntax tree of the following expression. [**Note**: The subscripts attached to each operator indicate its precedence class and whether that class is left- or right-associative.]

$$a\ \#_{1R}\ (w\ \$_{2L}\ \%_{3L}\ @_{2L}\ x\ ^{}_{3L}\ 8\ \sim_{1R}\ y)\ ^{}_{1R}\ d$$



Rewrite the expression in ***prefix*** notation.

`# a ^ ~ ^ % $ w @ x 8 y d`

Rewrite the above expression in ***postfix*** notation.

`a w $ x @ % 8 ^ y ~ d ^ #`

Infix expressions in a certain language use the following three precedence classes:

	Binary operators	**Prefix** unary operators	**Associativity**
Class 1	\	~	***right***-associative
Class 2	$	#	***left***-associative
Class 3	& @	none	***right***-associative

Class *i* operators have higher precedence rank than class *i*+1 operators – ***class 1 is the highest precedence class, and class 3 is the lowest precedence class.***

i. Here is an infix expression in the above-mentioned language:

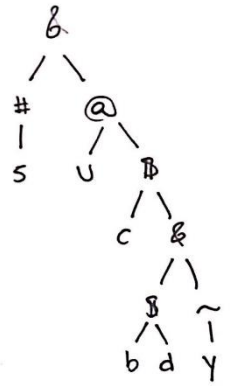
`# 5 / & u @ c $ (6 $ d & ~ y)`

Circle the operator that should be applied ***last*** when this expression is evaluated.

ii. Circle the operator that should be applied ***last*** when the following subexpression of the expression of (i) is evaluated:

`(6 $ d & ~ y)`

iii. Draw an abstract syntax tree of the expression of (i).



iv. Rewrite the expression of (i) in ***prefix*** syntax.

`5 # & u @ c $ b $ d & ~ y`

In a certain language, expressions are written in infix notation. The language has binary and prefix unary operators that belong to the following precedence classes:

	Binary ops	**Prefix ops**	**Associativity**
1st Class:	[none]	@ #	Right
2nd Class:	+ &	[none]	Left
3rd Class:	% ^ $	[none]	Left

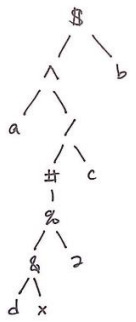
1st class operators have the ***highest*** precedence and 3rd class operators have the ***lowest*** precedence.

Consider the following expression in this language; the subscripts attached to each operator indicate its precedence class and whether that class is left- or right-associative.

$$@_{1R}\ a\ ^{\wedge}_{3L}\ \#_{1R}\ (d\ \&_{2L}\ x\ \%_{3L}\ 2)\ \backslash_{2L}\ c\ \$_{3L}\ b$$

a) Which operator in the above expression should be applied ***last*** when the expression is evaluated?
   `$`

b) Draw an ***abstract syntax tree*** of the expression.



c) Rewrite the expression in *prefix* notation.
   `$ ^ @ a \ # & d x 2 c b`