

C SCI 316: Lisp Assignment 4

To be submitted no later than: Thursday, October 28.* NOTE: If euclid unexpectedly goes down after 6 p.m. on this due date, there will be *no* extension. Try to submit no later than noon on the due date, and much sooner if possible: To give yourself more time to do reading assignments and work on the next programming assignment (Lisp Assignment 5, which you will receive no later than Monday, October 25), I recommend you complete this assignment well before the due date.

Submissions after the due date will be accepted as *late* submissions until a deadline that will be announced later. See p. 3 of the 1st-day announcements document for information regarding late-submission penalties.

Program in a functional style, without using SETF or DO. Follow the indentation and spacing rules at: <https://euclid.cs.qc.cuny.edu/316/indentation-and-spacing-guidelines-for-Lisp-Assignments.pdf>

Submission instructions are given on page 5.

In these problems, the behavior of the functions you are asked to complete or write is specified only when the arguments have certain explicitly stated properties—e.g., in problem A the behavior of MY-SUM is specified only if its argument is a nonempty list of numbers, and in problem 1 the behavior of SUM is specified only if its argument is a (possibly empty) list of numbers. When functions' arguments do *not* have the stated properties (e.g., if the argument of MY-SUM is NIL, or if an argument of the function SET-UNION of problem 10 is a list in which some element occurs more than once), the functions' behavior is *unspecified*: **Your functions are allowed to return any result or to produce an evaluation error in such cases!**

When evaluation of a function call has produced an infinite loop, you can often abort execution by typing Ctrl-C. At a Break> error prompt, typing `backtrace` will print, in reverse order, the sequence of all function calls that are in progress.

In this document the term *list* should be understood to mean *proper* list.

SECTION 1 (Nonrecursive Preliminary Problems)

The 7 problems in this section (A – G) do not carry direct credit, but are intended to help you solve problems 1 – 7 in Section 2. There *may* be exam questions of a similar nature to A – G.

Your solutions to problems A – G *must not be recursive*. You can test your solutions to these problems on *venus*[†] or *euclid*: Functions SUM, NEG-NUMS, INC-LIST-2, INSERT, ISORT, SPLIT-LIST, and PARTITION with the properties stated in A – G are predefined when you start clisp using `c1` on *venus*[†] or *euclid*. When a function has 2 cases, test your code in both cases!

- A. SUM is a function that is already defined on *venus* and *euclid*; if *L* is *any list of numbers* then (SUM L) returns the sum of the elements of L. [Thus (SUM ()) returns 0.] Complete the following definition of a function MY-SUM *without making further calls of* SUM and without calling MY-SUM recursively, in such a way that if L is any *nonempty* list of numbers then (MY-SUM L) is equal to (SUM L).

```
(defun my-sum (L)
  (let ((X (sum (cdr L))))
    _____ ))
```

*If you have difficulty with these problems, you are encouraged to see me during my office hours. Questions about these problems that are e-mailed to me **will not be answered until after the due date.**

[†]This assumes you executed the `/home/faculty/ykong/316setup` command on *venus* before you did Lisp Assignment 1 (in accordance with the instructions for Assignment 1).

- B. NEG-NUMS is a function that is already defined on venus and euclid; if *L is any list of real numbers* then (NEG-NUMS L) returns a new list that consists of the negative elements of L. For example:
 (NEG-NUMS '(-1 0 -8 2 0 8 -1 -8 2 8 4 -3 0)) => (-1 -8 -1 -8 -3).

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

```
(defun my-neg-nums (L)
  (let ((X (neg-nums (cdr L))))
    _____
  ) )
```

There are two cases: (car L) may or may not be negative.

- C. INC-LIST-2 is a function that is already defined on venus and euclid; if *L is any 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). For example,

(INC-LIST-2 () 5) => NIL (INC-LIST-2 '(3 2.1 1 7.9) 5) => (8 7.1 6 12.9)

Complete the following definition of a function MY-INC-LIST-2 *without making further calls of* INC-LIST-2 and without calling MY-INC-LIST-2 recursively, in such a way that if L is any *nonempty* list of numbers and N is any number then (MY-INC-LIST-2 L N) is equal to (INC-LIST-2 L N).

```
(defun my-inc-list-2 (L N)
  (let ((X (inc-list-2 (cdr L) N)))
    _____
  ) )
```

- D. INSERT is a function that is already defined on venus and euclid; if N *is any real number* and L *is any list of real numbers in ascending order* then (INSERT N L) returns a list of numbers in ascending order obtained by inserting N in an appropriate position in L. Examples:
 (INSERT 8 ()) => (8) (INSERT 4 '(0 0 1 2 4)) => (0 0 1 2 4 4) (INSERT 4 '(0 0 1 3 3 7 8 8)) => (0 0 1 3 3 4 7 8 8)
 Complete the following definition of a function MY-INSERT *without making further calls of* INSERT and without calling MY-INSERT recursively, in such a way that if N is any real number and L is any *nonempty* list of real numbers in ascending order then (MY-INSERT N L) is equal to (INSERT N L).

```
(defun my-insert (N L)
  (let ((X (insert N (cdr L))))
    _____
  ) )
```

[There are two cases: N may or may not be \leq (car L). In the former case you do not need to use X, so if you move that case outside the LET the function will be more efficient.]

- E. ISORT is a function that is already defined on venus and euclid; if *L is any list of real numbers* then (ISORT L) is a list consisting of the elements of L in ascending order. Complete the following definition of a function MY-ISORT *without making further calls of* ISORT and without calling MY-ISORT recursively, in such a way that if L is any *nonempty* list of real numbers then (MY-ISORT L) is equal to (ISORT L).

```
(defun my-isort (L)
  (let ((X (isort (cdr L))))
    _____
  ) )
```

Hint: You should not have to call any function other than INSERT and CAR.

IMPORTANT: If you have not yet done problems 15 – 20 of Lisp Assignment 2, do those six problems before you work on the next two problems!

- F. SPLIT-LIST is a function that is already defined on venus and euclid; if L is any 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 '(A B C D 1 2 3 4 5)) => ((A C 1 3 5) (B D 2 4)) (SPLIT-LIST '(B C D 1 2 3 4 5)) => ((B D 2 4) (C 1 3 5)) (SPLIT-LIST '(A)) => ((A) NIL)
- Complete the following definition of a function MY-SPLIT-LIST *without making further calls of* SPLIT-LIST and without calling MY-SPLIT-LIST recursively, in such a way that if L is any *nonempty* list then (MY-SPLIT-LIST L) is equal to (SPLIT-LIST L).

```
(defun my-split-list (L)
  (let ((X (split-list (cdr L))))
    _____ ))
```

- G. PARTITION is a function that is already defined on venus and euclid; 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 in L. Examples: (PARTITION '(7 5 3 2 1 5) 1) => (NIL (7 5 3 2 1 5)) (PARTITION '(4 0 5 3 1 2 4 1 4) 4) => ((0 3 1 2 1) (4 5 4 4)) (PARTITION () 9) => (NIL NIL)
- Complete the following definition of a function MY-PARTITION *without making further calls of* PARTITION and without calling MY-PARTITION recursively, in such a way that if L is any *nonempty* list of real numbers and P is a real number then (MY-PARTITION L P) is equal to (PARTITION L P).

```
(defun my-partition (L P)
  (let ((X (partition (cdr L) P)))
    _____ ))
```

There are two cases: (car L) may or may not be less than P.

SECTION 2 (Main Problems)

Your solutions to problems 1 – 13 will count a total of 2% towards your grade if it is computed using rule A. **Note that a working solution to each of problems 1 – 7 can be obtained from a solution to the corresponding one of problems A – G by changing the name of the function MY-FUNC to FUNC and adding appropriate base case code, without changing the LET block.** [The resulting definition of FUNC will be correct because it has the following property: *In all non-base cases where FUNC's recursive call returns the correct result, FUNC returns the same result as MY-FUNC.* Assuming your definition of MY-FUNC is correct, this property implies that in non-base cases FUNC returns the correct result whenever FUNC's recursive call returns the correct result, which in turn implies FUNC never returns an incorrect result if your base case code is correct.] But if you solve a problem this way then **any cases that do not need to use the LET's local variable should be moved out of the LET, and the LET should be eliminated if the value of its local variable is never used more than once.**

Note 1: On euclid and venus, when you LOAD a function definition for any of problems 1 – 7, your function will *replace* the predefined function that has the same name (and Clisp will issue a WARNING that the predefined function has been redefined). It follows, for example, that if your definition of SUM for problem 1 is wrong then, after you LOAD your definition of SUM, your definition of MY-SUM for problem A may stop working even if that definition is correct because MY-SUM calls SUM.

Note 2: You may use ENDP or NULL to test whether a list is empty. Recall that (ENDP L) returns the same result as (NULL L) if the value of L is a list. (But evaluation of (ENDP L) produces an error if the value of L is an atom other than NIL.)

1. Define a recursive function SUM with the properties stated in problem A. Note that whereas NIL is not a valid argument of MY-SUM, NIL is a valid argument of SUM.

2. Define a recursive function NEG-NUMS with the properties stated in problem B. Note that NIL is a valid argument of NEG-NUMS.
3. Define a recursive function INC-LIST-2 with the properties stated in problem C. Note that the first argument of INC-LIST-2 may be NIL.
4. Define a recursive function INSERT with the properties stated in problem D. Note that the second argument of INSERT may be NIL.
5. Define a recursive function ISORT with the properties stated in problem E. **Hint:** In your definition of ISORT you should not have to call any function other than ISORT itself, INSERT, CAR, CDR, and ENDP or NULL. (An IF or COND form is not considered to be a function call, and will be needed.)
6. Define a recursive function SPLIT-LIST with the properties stated in problem F.
7. Define a recursive function PARTITION with the properties stated in problem G.
8. Without using MEMBER, complete the following definition of a recursive function POS such that if *L is a list and E is an element of L* then (POS E L) returns the position of the first occurrence of E in L, but if *L is a list and E is not an element of L* then (POS E L) returns 0.

```
(DEFUN POS (E L)
  (COND ((ENDP L) ... )
        ((EQUAL E (CAR L)) ... )
        (T (LET ((X (POS E (CDR L))))
              ...
            ) ) ) )
```

Examples: (POS 5 '(1 2 5 3 5 5 1 5)) => 3 (POS 'A '(3 2 1)) => 0 (POS '(3 B) '(3 B)) => 0
 (POS '(A B) '((K) (3 R C) A (A B) (K L L) (A B))) => 4 (POS '(3 B) '((3 B))) => 1

9. Define a recursive 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 even integers between 0 and N in descending order, and the other list consists of the odd integers between 0 and N in descending order. Examples: (SPLIT-NUMS 0) => ((0) NIL)
 (SPLIT-NUMS 7) => ((6 4 2 0) (7 5 3 1)) (SPLIT-NUMS 8) => ((8 6 4 2 0) (7 5 3 1))

IMPORTANT: In problems 10 – 13 the term set is used to mean a proper list of numbers and/or symbols in which no atom occurs more than once. You may use MEMBER but *not* the functions UNION, NUNION, REMOVE, DELETE, SET-DIFFERENCE, and SET-EXCLUSIVE-OR.

10. Define a recursive function SET-UNION such that if *s1 and s2 are sets* then (SET-UNION s1 s2) is a set that contains the elements of s1 and the elements of s2, but no other elements. Thus (SET-UNION '(A B C D) '(C E F)) should return a list consisting of the atoms A, B, C, D, E, and F (in any order) in which no atom occurs more than once.
11. Define a recursive function SET-REMOVE such that if *s is a set and x is an atom in s* then (SET-REMOVE x s) is a set that consists of all the elements of s except x, but if *s is a set and x is an atom which is not in s* then (SET-REMOVE x s) returns a set that is equal to s.

In problems 12 and 13 you may use the function SET-REMOVE from problem 11.

12. Define a recursive function SET-EXCL-UNION such that if *s1 and s2 are sets* then (SET-EXCL-UNION s1 s2) is a set that contains all those atoms that are elements of *exactly one* of s1 and s2, but no other atoms. (SET-EXCL-UNION s1 s2) does *not* contain any atoms that are neither in s1 nor in s2, and also does *not* contain the atoms that are in both of s1 and s2. For example, (SET-EXCL-UNION '(A B C D) '(E C F G A)) should return a list consisting of the atoms B, D, E, F, and G (in any order) in which no atom occurs more than once.

13. Define a recursive function SINGLETONS such that if *e* is any list of numbers and/or symbols then (SINGLETONS *e*) is a set that consists of all the atoms that occur *just once* in *e*.
Examples: (SINGLETONS ()) => NIL (SINGLETONS '(G A B C B)) => (G A C)
(SINGLETONS '(H G A B C B)) => (H G A C) (SINGLETONS '(A G A B C B)) => (G C)
(SINGLETONS '(B G A B C B)) => (G A C) [Hint: When *e* is nonempty, consider the case in which (car *e*) is a member of (cdr *e*), and the case in which (car *e*) is not a member of (cdr *e*).]

How to Submit

Important Note: If euclid unexpectedly goes down after 6 p.m. on the due date, there will be **no extension**. Try to submit no later than noon on the due date, and much sooner if possible.

You may work with up to two other students on these problems. But, **as stated on p. 3 of the first-day announcements document, when two or three students work together each of them must write up his or her solutions individually; no two students should submit identical files.**

Put the function definitions you wrote for problems 1 – 13 in a single file named
your last name in lowercase-4.lsp

This file must include definitions of any helping functions that are used. At the beginning of the file there must be a comment that shows your name and, if you are working with one or two partners, the name(s) of your partner(s).

Within the file, your solution to each problem should be preceded by a comment of the following form, where N is the problem number: `;;; Solution to Problem N`
Your solutions should appear *in the same order as the problems*. If you cannot solve a problem, put a comment of the form `;;; No Solution to Problem N Submitted` where a solution to that problem would have appeared.

To submit your solutions, leave a copy of *your last name in lowercase-4.lsp* in your home directory on euclid *no later than* the due date. (If you are working on another machine and have forgotten how to copy files to euclid, then re-read p. 3 of the Lisp Assignment 3 document.)

After leaving the file *your last name in lowercase-4.lsp* on euclid as explained above, login to your euclid account and test your Lisp functions on euclid: Start Clisp by entering `cl` at the euclid> prompt, enter `(load "your last name in lowercase-4")` at Clisp's > prompt, and then call each of your functions with test arguments. **Note that if euclid's Clisp cannot even LOAD your file without error (i.e., if LOAD gives a Break ...> prompt) then you can expect to receive no credit at all for your submission, even if the only error is a single missing or extra parenthesis!**

Functions that are incorrectly named may receive **no credit** (e.g., if your solution to problem 3 is named INCLIST-2 or INC-LIST2, you may well get **no credit** for that problem).

Do NOT open your submitted file in any editor on euclid after the due date, unless you are (re)submitting a corrected version of your solutions as a late submission! Also do not execute `mv`, `chmod`, or `touch` with your submitted file as an argument after the due date. (However, it is OK to view a submitted file using the `less` file viewer after the due date.)

As mentioned on p. 3 of the first-day announcements document, you are required to keep a backup copy of your submitted file on venus, and another copy elsewhere. You can enter the following two commands on euclid to email a copy of your submitted file to yourself and to put a copy of the file on venus:

```
echo . | mailx -s "copy of submission" -a your last name in lowercase-4.lsp $USER  
scp your last name in lowercase-4.lsp your venus username@149.4.211.180:
```