# LISP Programming Assignment [100 pts] Due Date: March 26rd, 2015

1. [20 pts] Implement a recursive function that computes the **GCD** of two positive integers, *m* and *n* using two different algorithms. Include C/Java-like code in your LISP comments to describe the algorithms applied.
   1. Implement ***GCD\_E*** using Euclid’s algorithm. Assume *m* >= *n* > 0. Use the LISP ***rem*** function (remainder).
   2. Implement ***GCD\_D*** using Dijkstra’s algorithm. Here *m* may be <, >, or = to *n*.

*Note 1*: There is already a LISP function, GCD, which you can use only to verify the correctness of your implementation.

*Note 2*: **You CANNOT use loops** (a recursive approach is required)

*Note 3*: The two functions must be named as stated above (GCD\_E and GCD\_D)

*Note 4*: For GCD\_E, you may use the “rem” function which returns the remainder of the division of two integer numbers. E.g. (rem 5 3) returns 2.

*Note 5*: For GCD\_D you cannot use any other function than “cond”, “=”, “>” and “-“ (subtraction) functions.

*Note 6*: You should not use SETQ anywhere in your function implementation.

1. [10 pts] Implement a recursive function that checks if a number is a prime; the function returns true (T) if N is a prime, and false (NIL) otherwise. You may have to define an additional *helper* function – this one is going to be the actual recursive function. (You may use the recursive definition below):

**isPrime(N) = isPrime2(N, N-1)**

**isPrime2(N, 1) = true**

**isPrime2(N, D) = if D divides N 🡪 false**

**else isPrime2(N, D-1)**

*Note 1*: The main function to implement must be named “**isPrime**”. The recursive helper function can be named anything you want.

*Note 2:* You cannot use loops. A recursive method is required for the helper function.

1. [10 pts] Implement a recursive function named “**isPalindrome**” that returns true (***t***) if a string (represented as a flat list of atoms) is a **palindrome** and false (***nil***) otherwise. See example below:

**> (ispalindrome '(a b c))**

**NIL**

**> (ispalindrome '(a b c b a))**

**T**

**> (ispalindrome '(a b c c b a))**

**T**

**> (ispalindrome '(a c b a))**

**NIL**

*Note* 1: Do not use the “reverse” or the “butlast” functions.

*Note 2:* You may use the “last” function which returns the last element of a list, but (careful!) the result will be in a *list* format. E.g.: (last ‘(1 2 3 4)) will return the **list**: (4).

*Note 3 (****Hint):*** Implement a *helper* function that returns the original list without the last element; equivalent to what the built-in function “butlast” does.

*Note 4:* No loops and no setq in the implementation.

1. [30 pts] Implement a recursive function named “**traverseTree**” that traverses a *binary tree* structure, represented as a multi-level list, and returns the list of nodes in the order they were visited, depending on the traversal type: pre-, post-, and in-order. The function should have two arguments: the **tree** (list) and the traversal **order**. The order argument must be a *string* that takes one of the following values “**pre**”, “**post**”, and “**in**”. Any other value will result in an error message displayed in the console See the example below on how to represent the tree as a multi-level list and what the function is expected to return:

Assume the **tree** structure representation as a multi-level list:

(root (left) (right))

Then, the tree in the figure to the left would be then represented as:

(1 (2 (4 () ()) (5 () ())) (3 (6 (7 () (8 () ())) ()) ()))

Assume the list l is the tree structure above. Then the second row in the table below shows the function call for the various traversal algorithms, while the third row shows the expected result.

|  |  |  |
| --- | --- | --- |
| *Pre-order* | *In-order* | *Post-order* |
| (traverseTree l “pre”) | (traverseTree l “in”) | (traverseTree l “post”) |
| (1 2 4 5 3 6 7 8) | (4 2 5 1 7 8 6 3) | (4 5 2 8 7 6 3 1) |

1. [30 pts] Implement a set of recursive functions that perform the following sorting algorithms on a flat list of integer atoms. ***Note***: the elements in the list may not be unique. I.e. the input list may look like: (5 3 4 3 3 7 2 5) 🡪 sorted = (2 3 3 3 4 5 5 7).
   1. [15 pts] Insertion Sort
   2. [15 pts] Merge Sort

How to submit your homework:

Use the instructions from Homework #1. Name your LISP source file “**hw3.lsp**”. Include the output of your program as well (use the “dribble” command) – name the output “**hw3.txt**”.

Make sure to include comments (preceded by a semicolon) in your LISP source file with your name and various implementation details where necessary and/or required.

**IMPORTANT**: Load and test all your functions together (not as you incrementally add new functions to your source file). **Make sure the file LOADS properly** (and hence compiles) on a Windows machine, using GNU CLISP 2.49. You may use other LISP IDEs to develop the code, but eventually your source file (.lsp) must load in the LISP application we used in class. I WILL NOT DEBUG YOUR CODE! At least submit code that compiles.

Do NOT include test list variables in your source file! Only the function definitions.

Do add comments to separate the functions implemented for each problem and to document your code/approach/logic.

Name your main functions as indicated in each of the problems. Consider the problem specifications as your interfaces.

Do not use SETQ anywhere in your implementation (in any of the functions you define).

Do not use the LOOP function at all. Do not use *any* iterative function (while, until, unless …)

Do not overwrite the functions you define in the source file. Remember the top-down evaluation of all functions defined in a single source file.

Do not introduce unwarranted assumptions about the functions, the input, or the output. If in doubt, ask the professor.

Do not introduce or require additional functions to be called to “set up” your input variables or to display the output. The output should be displayed automatically as a result of it being the last evaluated item in the main function.