ASSIGNMENT 5: LISP

Due: February 26, 2014

Overview

The purpose of this assignment is for you to gain some experience designing and implementing LISP programs. This assignment explores only a few of the many interesting LISP features.

This assignment is broken into five parts. The first part is a fairly straightforward LISP warmup. The second part illustrates a standard technique of car-cdr recursion. The third part involves writing your own version of the standard LISP function mapcar. The fourth part involves writing a function that rewrites LISP expressions. More specifically, this part takes as input a LISP expression and produces as output another, possibly different LISP expression. The modified expression will be semantically equivalent to the original, but it will use cond's rather than if's. The fifth part checks the results of the fourth part.

N.B., you are restricted as to which LISP functions you are allowed to use for various parts of this assignment. See "Notes" for details.

Part 1: The Double Functions

Write the following three functions.

double-recursive (*y*)

double-iterative (*y*)

double-mapcar (*y*)

each of these returns the list consisting of each element of list *y* in a list has been replaced by two copies of the elements. For example,

(double-recursive '(1 2 3))

returns

(1 1 2 2 3 3)

If *y* is not a list, then nil is returned.

These double functions work only at the top-level; For example,

(double-recursive '(1 (2 3) 4))

returns

(1 1 (2 3) (2 3) 4 4)

Note how the top-level elements are duplicated, but the elements of nested lists are not.

double-recursive is to be written *recursively*, *double-iterative* is to be written *iteratively* using either 'go' or 'do', and *double-mapcar* is to be written using 'mapcar'. Hint: for *double-mapcar*, use the "apply-append trick" (see text).

Part 2: The Recursive Double Function

Write the function

rdouble (*y*)

On non-nested lists, it behaves like the double functions above. On nested lists, however, it returns a list of duplicates of the inner elements. For example,

(rdouble '(1 (2 3) 4))

returns

(1 1 (2 2 3 3) 4 4)

rdouble is to be written recursively (no iteration is allowed). Hint: use car-cdr recursion.

Part 3: The my-mapcar Function

Write the function

my-mapcar (*fun l1 l2*)

it behaves exactly as mapcar using *fun* with argument lists *l1* and *l2*. Assume that *my-mapcar* is invoked only with the two argument lists and that *fun* can be evaluated on two arguments. Write *my*-mapcar recursively. Use no iteration or mapping functions!

Part 4: The Rewrite Function

Write the function

rewrite (*x*)

returns an expression in which each if has been replaced by an equivalent cond. *x* is a valid LISP expression. Assume each if is properly formed, i.e., contains a then part and possibly an else part. Examples:

Expression	*Returns*
(rewrite '(* 44 2))	(* 44 2)
(rewrite '(if 3 4 5))	(cond (3 4) (t 5))
(rewrite '(list (if (= 8 8) 'y) (if (= 8 7) 'no)))	(list (cond ((= 8 8) 'y)) (cond ((= 8 7) 'no)))

Part 5: The Check Function

The function

check (*x*)

returns a list of three values. The second element is the result of evaluating expression *x*. The third element is the result of evaluating the result of (rewrite *x*). The first element is t or nil corresponding to whether or not the value of the second element is equal to that of the third element. Note that if your rewrite function (and check function!) is correct, the first element of each list that check returns will be "t". Assume that *x* contains no variables.

Notes

- The command to use Common LISP is “clisp -q”. clisp is available on CSIF systems.
- Some editors provide specific support to simplify editing LISP functions, for example, emacs’s LISP mode, vi’s -l option, and nedit’s and jot’s parenthesis matching.
- Appendix A of LISPcraft summarizes LISP’s built-in functions. Each function is explained briefly. You will find this a very useful reference as you write and debug your program.
- The test program is provided in the "given" on the class webpage. It exercises the functions that you write; hence, there is no test data. If "test.l" is the name of the test file in your directory, then, within LISP, you need only type “(load "test.l")”. This file defines the test functions

  test-double, *test-rdouble*, *test-rewrite*, and *test-check*.

  Each of these functions exercises the corresponding functions that you are to write. For example, to test your double functions simply type

  (test-double)

  In addition, the function *test* simply invokes each of the above test functions. These test functions use additional helper functions. For example,

  (test-double-recursive)

  tests only *double-recursive*. See the test file for additional helper functions. Do not use any of the test function names as a name of one of your functions. Note that each of these functions returns “t” when complete, so there will be an extra line of output.
- We’re also providing a “batch mode” test script, which you should find very helpful.
- “Correct” output will also be given. Your output must match the “correct” output. By “match”, we mean match exactly character by character, including blanks, on each line; also, do not add or omit any blank lines. (For this program, since LISP is doing all the output, that shouldn’t be hard.) Be sure to read and follow relevant comments in the test program. The only exceptions are the “Dribble ... started ...” and “Dribble ... finished ...” lines, which will, of course, not match. In any case, it is up to you to verify the correctness of your output.
- You may define additional helper functions that your main functions use. Be sure, though, to name the main functions as specified since the test program uses those names.
- Coding restrictions.

  See the LISP functions webpage under HW5 on the webpage. As noted there, use only PURE functions for all functions you write, except:

  double-iterative: PURE, BASICITERATIVE, prog, return, setq
  double-mapcar: PURE, MAPPING

  (OK, I’ll be explicit: Don’t even think about using MAPPING for my-mapcar!)

  Use no global variables.

- To define your own initial LISP environment, place an init.lsp file in the directory in which you execute clisp and run clisp via "clisp -q -i init.lsp". For example, you will probably want to put the command

  (setq *print-case* :downcase)

  in that file.

- When developing your program, you might find it easier to test your functions first interactively before using the test program. You might also find trace feature (LISPcraft section 11.5) or print functions (including the format function) useful in debugging your functions.

- Your code's execution on CSIF must not be grossly inefficient: your code must complete all the provided tests in at most 10 seconds (which is very generous). If your code is taking longer, then you are likely doing much needless recomputing. Using "let" expressions will likely help you solve that problem.

- Grading will be divided as follows.

Percentage	Function(s)
30	Part 1: The Double Functions
25	Part 2: The Recursive Double Function
10	Part 3: The my-mapcar Function
30	Part 4: The Rewrite Function
5	Part 5: The Check Function

- A few points to help the novice LISP programmer:

  - Watch your use of "(", ")", "'", and "`". Be sure to quote things that need to be quoted, e.g., the name of the file in load.

  - To see how LISP reads in your function, use pretty printing. For example, (pprint (symbol-function 'foo)) will print out the definition of function foo, using indentation to show nesting. This is useful to locate logically incorrect nesting due to, e.g., wrong parenthesizing.

  - If you cause an error, Common LISP places you into a mode in which debugging can be performed (LISPcraft section 11.2). To exit any level, except the top level, type ":q". To exit the top level, type "^D". See the class handout for an example.

- See the webpage for exact details of what to turn in, the provided source files, etc. As usual, you must develop this program in "part order": No credit will be given for one part if the previous part is not entirely working.