**CSCE476/876** **Spring 2003**

# Homework 3: Programming Assignment Using Emacs and Common Lisp

**Assigned on:** Wednesday February 5, 2003.

**Due:** Monday February 24, 2003

---

This programming assignment will help you get familiarized with Common Lisp by demonstrating a few simple programs and asking you to write a few more. For each of the problems, create a separate lisp file. Name them problem1.lisp, problem2.lisp, and so on. Store all of your work on a given problem in the same file. When required to define several functions, put them all in the same file.

## Getting started

Emacs is more than a simple (and powerful) editor: it provides you with a terrific environment for running a Common Lisp interpretor. Emacs may seem a little confusing at the beginning, but your efforts will quickly pay off.

1. Carefully read the instructions for setting up your environment provided in
   http://cse.unl.edu/čhoueiry/S03-476-876/linux-acl.html
   then conscientiously go through the Emacs tutorial:
   http://csce.unl.edu /čhoueiry/S03-476-876/emacs-tutorial.txt

2. Check out the key-stroke accelerators provided in
   http://cse.unl.edu/čhoueiry/S03-476-876/emacs-lisp.html .

   Open an Emacs buffer, create a file my-test.lisp, write a Lisp function, and test them.
   In particular, load a file (C-x C-f), check how TAB and Space bar achieve completion of commands and file names, interrupt a command (C-g), delete a line in a buffer (C-k), move forward and backward in the buffer (C-f, C-b, M-f, M-b, etc.), save the modifications in the buffer to the file (C-x C-f), check the message in the mini buffer), kill an open buffer (C-x k)

3. Start a Lisp interpreter in Emacs by typing M-x fi:common-lisp (check out completion with the space bar by typing M-x fi:com<space-bar>). Answer yes by typing <return> to all questions asked in the mini-buffer (until you learn to do otherwise). Now you should have a prompt sign of the Lisp interpreter. This is a loop that reads whatever you type in and evaluates it as a Lisp expression as soon as you hit the carriage return. Practice your knowledge of Emacs and interactions with the Lisp interpretor by executing all the instructions in Chapter 2, 3, and 4 of LWH. In particular,

- Test the functions car, cdr, cadr, cdar, first, length which operate on a list.
- Test cons, append and list and note the differences between them with respect to their input and output.
- Test push, pop, pushnew, delete and remove and note whether or not they are destructive.
- Test unary predicates atom, listp, consp, null, evenp, oddp, etc. on atoms, numbers, lists, NIL and T as input.
- Test the binary predicate =. The test eq, eql and equal. For instance, define: (setf ls1 '(a b c)) and (setf ls2 '(b c)). Now, Test: (eq (cdr ls1) ls2) and (equal (cdr ls1) ls2). What do you conclude?
- Read about and test the constructs if, when, cond, do, do*, dolist, dotimes, mapcar, find, reduce (my absolute favorite), some, every,
- Read about and test the functions on sets (as lists): intersection, union, set-difference, member, subseteq, adjoin.

4. Save some of the functions you have written in the file my-test.lisp. Exit Lisp by typing :exit in the Lisp interpreter and start Lisp again typing M-x fi:com<space-bar>. You can load the functions you have written in my-test.lisp in the Lisp environment by typing in your lisp buffer:

   (load "<path>/my-test.lisp")

   Emacs provides also some quick commands: :ld ~/<path>/my-test.lisp. To have a list of all the abbreviated commands provided by emacs, type in your Lisp buffer help. Note that all abbreviated commands start with :.

5. Exit Lisp with :ex and quit emacs C-x C-c.

Now, it is time to jump into the fire! Do not hesitate to ask the GTAs for help.

# 1 Squaring (5 points)

This problem asks you to define a very simple function. Write the function (sqr n) that returns the square of the argument n. For example, (sqr 3) should return 9 and (sqr 1.414) should return 1.9993961.

# 2 Exponentiate (5 points)

Write the function (power n m) that raises and number n to an integer power m. For example, (power 3 2) should return 9.

## 3 Fourth list element (5 points)

Common Lisp has a built-in command `fourth` that returns the fourth item in a list. For example `(fourth '(a b c d e f g))` returns `D`. Write the function `(my-fourth ls)` that performs the same operation on list `ls`. You can define this function recursively or iteratively, as you desire, but don't simply call the built-in function. Since this is meant to duplicate the functionality of `fourth`, the function call `(my-fourth '(a b c d e f g))` should also return `D`.

## 4 The `cond` conditional (10 points)

Review the syntax of the `cond` conditional operator. You will use it in this problem to handle a three case situation. Write a function `(sign n)` that will return `-` if the argument is less than zero, `0` if the argument is equal to zero, or `+` if the argument is greater than zero. For example, `(sign -91)` will return `-` and `(sign 31337)` will return `+`. Use `cond` to test for which value to return.

## 5 Even numbers (10 points)

Common Lisp has built-in functions that can be used to test whether a value is even or odd. These functions are called `evenp` and `oddp`. Both function take a single integer argument. Experiment with them to see what they do. Write a function `(all-even list)` that will take a list of integers and return a list containing only the even integers. For example

```
(all-evenp '(1 2 3 4 5 6 7 8 9 10))
```

should return `(2 3 6 8 10)`. This can easily be done by using a loop to iterate across the list and using the `evenp` function to decide whether or not to save the current element.

## 6 Member (10 points)

Common Lisp has a built-in function called `member`, which is called with the syntax

```
(member element list)
```

and will return `nil` if the `element` is not found in the `list`. If, on the other hand, the element is found in the list, the function will return a portion of the list, starting with the first occurrence of the element. For example, `(member 'b '(a b c d))` will return `(B C D)`. Also, observe that `(member 'b '(a b c a b c))` returns `(B C A B C)`. Experiment with the function, to be certain that you understand what it does.

1. Write a function `(my-member-cond element list)` that duplicates the functionality of the built-in `member` function. Implement the function using `cond` and a recursive call.

2. Write a function (my-member-do element list) that duplicates the functionality of the built-in member function. Implement the function iteratively, using the do primitive (see page 117 in your Lisp textbook).

# 7 Find (10 points)

Common Lisp has a built-in function called find, which is called with the syntax

(find element list)

and will return nil if the element is not found in the list. If, on the other hand, the element is found in the list, the function will simply return that element. For example, (find 'b '(a b c d)) will return B. Observe that (find 'b '(a b c a b c)) also returns B. Modify the my-member- functions that you wrote for the above problem to duplicate the built-in find function. This is a very simple task.

1. Create a function (my-find-cond element list) that uses recursion.

2. Create a function (my-find-do element list) that uses iteration.

# 8 List iteration (Total 20 points, 5 points each)

The goal of this exercise is to make you use various constructs of Common Lisp to iterate over the elements of list. You are asked to write a function double-xx that takes as input a list of numbers such as '(3 22 5.2 34) and returns a list of "doubled-up" numbers '(6 44 10.4 68).

1. Write double-mapcar using mapcar.

2. Write double-dolist using dolist.

3. Write double-do using do.

4. Write double-recursive using cond and recursive calls.

# 9 Exify (10 points)

Write a *recursive* function exify that takes a list as input and returns a list in which all non-nil elements are replaced by the atom X.
Test it first on: (exify '(1 hello 3 foo 0 nil bar)).
It should return: (X X X X X NIL X).
Then test it on: (exify '(1 (hello (3 nil (foo)) 0 (nil)) (((bar))))).
It should return: (X (X (X NIL (X)) X (NIL)) (((X)))).

## 10 Count occurrences (10 points)

Write a *recursive* function `count-anywhere` that takes an atom and an arbitrary nested list as input and counts the number of times the atom occurs anywhere within the list. Example `(count-anywhere 'a '(a (b (a) (c a)) a))` returns 4.

## 11 Dot Product (5 points)

Write a function that computes the dot product of two sequences of numbers represented as lists. Assume that the two lists given as input have the same length. The dot product is computed by multiplying the corresponding elements and then adding up the resulting product. Example:

```
(dot-product '(10 20) '(3 4))) = 110
(dot-product '(1 2 4 5) '(3 4 3 4)) = 43
```

## 12 X-product (25 points)

Write a function that takes a function name and two lists and returns the x-product defined by applying the function on the elements of the lists at the same position. Example:
`(x-product #'+ '(1 2 3) '(10 20 30))` returns `(11 12 13 21 22 23 31 32 33)` and
`(x-product #'list '(1 2 3) '(a b c))`
returns `((1 A) (2 A) (3 A) (1 B) (2 B) (3 B) (1 C) (2 C) (3 C))`

## 13 Bonus: Cartesian Product (40 points)

Write a function that takes a list of *any* number of lists and return the Cartesian product:
`(K-product '((a b c) (1 2 3)))`
returns: `((A 1) (A 2) (A 3) (B 1) (B 2) (B 3) (C 1) (C 2) (C 3))` and
`(k-product '((a b) (1 2 3) (x y)))`
returns:
```
((A 1 X) (A 1 Y) (A 2 X) (A 2 Y) (A 3 X)  (A 3 Y)
 (B 1 X) (B 1 Y) (B 2 X) (B 2 Y) (B 3 X) (B 3 Y))
```

# Notes:

- The terminology used above (i.e., dot, x-, Cartesian product) is *not* a strict one.
- Use the time and space profiler of Composer to improve your code. Use the Lisp function "time" to evaluate the cost of your code. You may want to make sure to do the right DECLARATIONS for optimizing your code for speed (check a Lisp manual), etc.