CSCE476/876 **Fall 2023**

Homework 1: Introduction to Lisp

Assigned on: Monday, August 28, 2023.

Due: Friday, September 8, 2023.

Goal. The goal of this homework is that you learn the fundamentals of Lisp and practice writing Lisp code using Emacs. We strongly recommend that you:

- Go through the Emacs tutorial (see below).
- Read Chapter 2, 3, and 4 of the book on Lisp by Winston and Horn (LWH).
- Practice the materials in Sections 1 and 2 below.

Submitting your work. You will be graded on answering the questions in Sections 3 and 4 below. All lisp functions should be done using ACL (preferably in Emacs). Type the answers to the non-programming questions in a `$<$lastname$>$.txt` file and put all your code for the programming questions in a `$<$lastname$>$.lisp` file. Submit both files on webhandin `http://cse.unl.edu/handin` *before* class on Friday, September 8, 2023.

Contents

1 Getting started

You have already seen most of the content of this 'getting started' section during the recitations. We just want to insist and make sure that you have gone through the steps below.

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

   (a) Conscientiously go through the Emacs tutorial:
   `http://cse.unl.edu/~choueiry/emacs-tutorial.txt`

   (b) Review some of the most commonly useful Emacs commands:
   `http://cse.unl.edu/~choueiry/Recitation/emacs.pdf`

   (c) Check out the key-stroke accelerators provided in
   `http://cse.unl.edu/~choueiry/emacs-lisp.html` .

   (d) Practice using Emacs. Open an Emacs buffer, create a file `my-test.lisp`, write a Lisp function, and test it.
   In particular, load a file (`C-x C-f`), check how `TAB` and the `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`)

2. 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 `=`. Then 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)`.

Search and read the documentation on the web.[1] Can you explain why the two functions give a different result?

- 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`.
- `mapcar` is a very useful function that will make the dot-product, x-product, and Cartesian Product very simple to do. `mapcar` is used in the form (literally taken from Guy Steele's *Common Lisp* page171):

  `mapcar` *function* list &rest *more-lists*

  `mapcar` operates on successive elements of the lists. First the function is applied to the `car` of each list, then to the `cadr` of each list, and so on. The value returned by `mapcar` is a list of the results of the successive calls to the function. For example:

  `(mapcar #'abs '(3 -4 2 -5 -6))` ⇒ `(3 4 2 5 6)`
  `(mapcar #'+ '(1 2 3) '(1 2 3))` ⇒ `(2 4 6)`

3. 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 a column, that is, `:`.

4. The stepper of ACL works best on compiled code. Check out the following scenario. First, compile your file and load the compiled filed. Then, type in the `*common-lisp*` buffer in Emacs:

   `:step '<name of the function to step through>` .

   Then type the function call:

   `(<name of the function to step through> <arg1> <arg2> etc.)` .

   To stop the stepper in emacs, just type: `:step`.

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 instructor and TA for help.

[1] `http://www.cs.cmu.edu/afs/cs.cmu.edu/project/ai-repository/ai/html/cltl/clm/node74.html`

2 Basic training

Follow the instructions on the webpage of the class:

1. From the Recitations page, go through "First Steps in Lisp" `https://cse.unl.edu/~choueiry/F23-476-876/Recitation/FirstStepsInLisp.pdf`
2. From the Recitations page, go through "Next Steps in Lisp" `https://cse.unl.edu/~choueiry/F23-476-876/Recitation/SecondStepsInLisp.pdf`
3. From the Recitations page, carefully read "Writing More 'Fluent' Lisp" `https://cse.unl.edu/~choueiry/F23-476-876/Recitation/notes-better-lisp.pdf`

3 Simple exercises (33 points)

Complete the following exercises.

1. **Is the following an atom, list, both, or neither?** **(5 points)**
   (a) `ATOM`
   (b) `(rest '(1 2 3))`
   (c) `)(`
   (d) `(rest '())`
   (e) `()`
2. **Math** Evaluate the following functions. **(3 points)**
   (a) `(/ (+ 5 7) (- 1 4))`
   (b) `(+ (* 3 4) (* 5 (first '(0 1 2))))`
   (c) `(max 3 (min 5 2 8))`
3. **First/Rest** **(8 points)**
   Get to the number 3 in each of the lists below by using some sequence of the Lisp functions first and rest. You may also use compound car/cdr's.
   Example: `(setf x '(1 2 3 4))`
   Answer: `(first (rest (rest x)))` or `(caddr x)`
   (a) `(setf x '(1 (2 (3 (4)))))`
   (b) `(setf x '((1 2) (3 4)))`
   (c) `(setf x '(1 (2 3) 4))`
   (d) `(setf x '((((1) 2) 3) 4))`

4. **Cond** **(4 points)**
   Define a function `ESTIMATE-SIZE` that takes a positive number n as an argument and returns SMALL if $n < 10$, MEDIUM if $10 \leq n < 50$, and BIG if $n \geq 50$

5. **NEGATIVE-P** **(2 points)**
   Define a predicate function `NEGATIVE-P` that takes a number as its argument and returns `t` if the number is negative. If the number is null or positive, it should return `nil`.

6. **X5** **(2 points)**
   Define a function `X5` that takes one argument, tests whether the argument is a number, and returns the number multiplied by 5 otherwise returns the argument itself.

7. **Using `mapcar`** **(6 points)**
   Find an on-line manual of Lisp, such as:
   `http://www-2.cs.cmu.edu/afs/cs.cmu.edu/project/ai-repository/ai/html/cltl/cltl2.html`
   `http://www.franz.com/support/documentation/6.2/ansicl/ansicl.htm`
   and study the definition and use of the function `mapcar`. Using `mapcar`, write a short function `MAPIPULATE` that takes a list of numbers and multiplies each negative number by 5, leaving every non-negative number as is, and return the modified list.

8. Describe in words how the following function operates: **(3 points)**

```
(defun foo(s)
  (cond ((null s) 1)
        ((atom s) 0)
        (t (max (+ (foo (first s)) 1)
                (foo (rest s))))))
```

4 Programming (ugrads: 72 points, grads: 102 points)

1. **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`. (Note: You are expected to write `power` yourself, you may not use `expr` or similar functions.)

2. **Even numbers** **(5 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 4 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. (Note: You are expected to write `all-evenp` yourself, you may not use `remove-if-not` or similar functions.)

3. **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 `(what-is n)` that will return `ATOM` if the argument is an atom, `LIST` if the argument is a list, or `NUMBER` if the argument is a number. For example, `(what-is -91)` will return `NUMBER` and `(whatis '(1 2 3))` will return `LIST`. Use `cond` to test for which value to return.

4. **Learn to use `reduce`** **(10 points)**
   Find an on-line manual of Lisp, such as:
   `http://www-2.cs.cmu.edu/afs/cs.cmu.edu/project/ai-repository/ai/html/cltl/cltl2.html`
   `http://www.franz.com/support/documentation/6.2/ansicl/ansicl.htm`
   and study the definition and use of the function `reduce`. This is a particularly elegant and powerful construct (instructor's favorite). Using `reduce`, write a very short function, `average-reduce`, that takes a list of numbers and returns the value of their average.

5. **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.

   (a) 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.

   (b) 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).

6. **Find** **(6 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 Homework 2 to duplicate the built-in `find` function. This is a very simple task.

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

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

7. **List iteration** **(Total 8 points, 2 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)`.

(a) Write `double-mapcar` using `mapcar`.

(b) Write `double-dolist` using `dolist`.

(c) Write `double-do` using `do`.

(d) Write `double-recursive` using `cond` and recursive calls.

8. **Exify** **(8 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))))`.

9. **Count occurrences** **(5 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.

10. **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
```

11. **X-product** **(15 points)**
Mandatory for grad students, bonus for undergrads.

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))`

Note: The terminology used above (i.e., dot, x-, Cartesian product) is *not* a strict one.

12. **Cartesian Product** **15 points)**
Mandatory for grad students, bonus for undergrads.

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))`