CSCE476/876 **Fall 2021**

Homework 2: Programming Assignment (Lisp Version)
Missionaries & Cannibals

Assigned on: Friday, September 10^{th}, 2021

Due: Monday, September 20^{th}, 2021

Contents

The goal of this assignment is to give students wishing to learn and use Common LISP an introduction to the language and to bolster understanding of the Farmer's Dilemma problem by extending its solution to the Missionaries and Cannibals puzzle. This homework is structured as follows:

- Section 1 gives an overview of using and navigating Emacs and using Emacs to write Lisp code. (0 points)
- Section 2 offers a few exercises that will likely be helpful when implementing your solution to the Missionaries and Cannibals problem. (Bonus 25 points)
- Section 3 introduces Allen's time relations. (20 points, Lisp bonus 4 points)
- Section 4 is the Missionaries and Cannibals puzzle itself. (60 points, Lisp bonus 6 points)

1 Getting started with Emacs and Common Lisp

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.

1. Carefully follow the instructions provided during recitation for setting up your environment, then conscientiously go through the Emacs tutorial:
   http://csce.unl.edu/∼choueiry/emacs-tutorial.txt

2. Check out the key-stroke accelerators provided in
   `http://cse.unl.edu/∼choueiry/emacs-lisp.html` .

   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`)

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`.
- `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)) ⇒ (1 4 6)
  ```

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. The stepper of ACL works best on compiled code, **and** when you stick to 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, just type: `:step`.

6. Use the time and space profiler of Composer to improve your code. Use the Lisp function `time` to evaluate the cost of your code (time and space). You may want to make sure to do the right DECLARATIONS for optimizing your code for speed (check a Lisp manual), etc.

7. 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 TA and RAs for help.

2 Useful functions

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 in a given problem, put them all in the same file.

2.1 Find (1 point)

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.

2.2 List iteration (1 point)

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.

2.3 Exify (2 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))))`.

2.4 Count occurrences (3 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.

2.5 Dot Product (3 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
```

2.6 X-product (5 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))`
Note: The terminology used above (i.e., dot, x-, Cartesian product) is *not* a strict one.

2.7 Cartesian Product (5 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))
```

2.8 Data Structures in LISP (5 points)

- Using `defstruct` create the data type `person`, with fields for a person's name, age, and list of pointers the structures of the siblings of the person.
- Create structures for Bob age 21, Susan age 18, and Frank age 16, who are all siblings.
- Use the `print` function to display the information about the people. Study what happens.
- The problem, if you notice it, is the print function of the data structure. Each symbol in lisp has a print function, which displays some information when the symbol is evaluated. We will be discussing the solution to this problem in recitation, however, you may want to start investigating how to modify the print function of defstruct, which can be easily done.

3 Allen's Time Relations (20 points, Lisp bonus 4 points)

This section of the homework deals with time intervals, which are the building blocks for temporal reasoning. For more background on the subject, goto pages 448 and 449 of AIMA. Figure 1 introduces all 13 possible qualitative relationships that may exist between two intervals. These relations are called Allen relations for qualitative temporal reasoning after James Allen who identified them in his seminal paper [**?**].

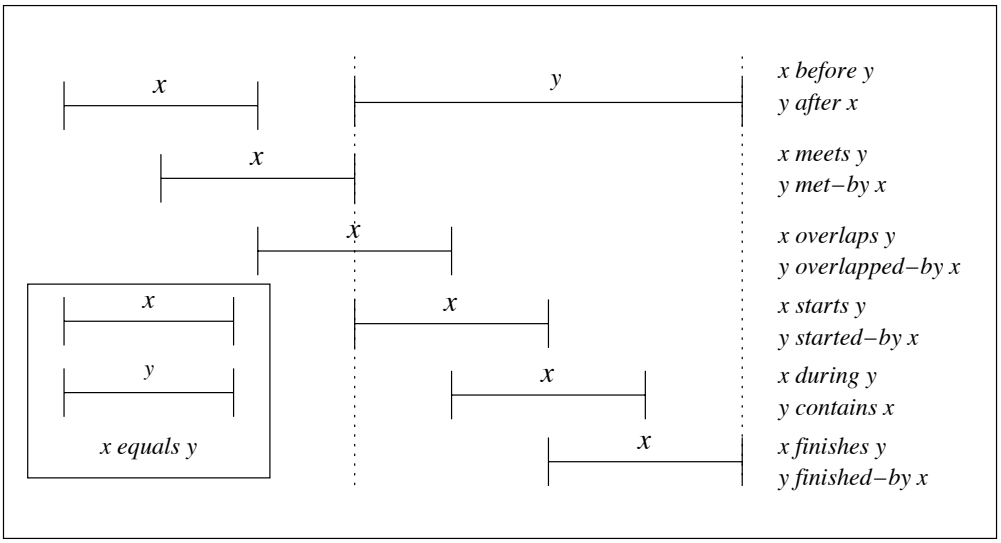

Figure 1: Predicates on time intervals

You are asked to implement CLOS (Common Lisp Object System) objects to represent the intervals and methods to determine whether or not the predicates hold.

1. Implement, using `defclass`, a data type `time-point` that has one slot, which is an integer (representing seconds).

2. Implement, using `defclass`, two data types `begint` and `endt`, as subclasses of `time-point`.

3. Implement, using `defclass`, a data type `interval` that has the following slots: `task-name`, `begint`, and `endt`, where `begint` and `endt` are of the type time-points.

4. Write methods `Start` and `End` that take an instance of `interval` and return the data members `begint` and `endt`, respectively.

5. Write the methods that implement the predicates listed below and illustrated in Figure 1, which take as input two objects of type interval and return whether or not each of the following predicates holds.

- *Meet(i,j)* $\Leftrightarrow$ *End(i) = Start(j)*
- *Before(i,j)* $\Leftrightarrow$ *End(i)* $<$ *Start(j)*
- *After(i,j)* $\Leftrightarrow$ *Before(j,i)*
- *During(i, j)* $\Leftrightarrow$ *Start(j)* $\leq$ *Start(i)* $\bigwedge$ *End(i)* $\leq$ *End(j)*
- *Overlap(i,j)* $\Leftrightarrow$ $\exists$ *k During(k,i)* $\bigwedge$ *During(k, j)*
- *Equals(i,j)* $\Leftrightarrow$ *Start(i) = Start(j)* $\bigwedge$ *End(i) = End(j)*
- *Finishes(i,j)* $\Leftrightarrow$ *End(i) = End(j)*
- *Contains(i,j)* $\Leftrightarrow$ *Start(i)* $<$ *Start(j)* $\bigwedge$ *End(i)* $>$ *End(j)*

The last two exercises in this homework are meant to introduce you to structures and classes. Generally speaking, structures are much lighter data objects than classes are. This is because classes require the definition of a many initialization methods (as described in the beautiful book *The Art of the Metaobject Protocol* of Kiczales). Classes are powerful, but heavy, so do use them only when you really need them.

4 Missionaries & Cannibals puzzle (60 points, Lisp bonus 6 points)

The puzzle is defined as follows: Three missionaries and three cannibals must cross a river using a boat that can carry at most two people at a time. All six people start on one bank of the river, and the goal is to find a series of boat rides that results in everyone on the second bank, with the following restrictions:

- The boat cannot cross the river with no one on board
- The cannibals on either bank cannot outnumber the missionaries on that bank (lest the cannibals eat the missionaries)

Implement a solution to the Missionaries & Cannibals puzzle. Include a file called 'readme.txt' that briefly describes how your program works and describes the functions you used to get the solution, as well as the solution to the puzzle. Your program should solve this puzzle using recursion, and display each of the boat rides used to solve the puzzle when run. Turn in (via handin) your solution in a file called *missionaries.lisp*.

Hints:

- For reference, you can use the Lisp code for the farmer's dilemma, which has been made available on the website of the course under the section 'Recitation.'
- Study this code and run it in ACL. 'Trace' the main functions until you understand how they work.

- Then, using the code for the Farmer's dilemma, write the Lisp code for solving the Missionaries and Cannibals puzzle.