# Homework 5

Due Thursday, October 18 by 10pm

Handout 7
CSCI 334: Fall 2018

## Turn-In Instructions

For this assignment, create a source file for each problem. For example, the source code for question 1 should be called "`q1.lisp`". You should be able to execute this file by running the command "`clisp < q1.lisp`".

Turn in your work using the Github repository assigned to you. The name of the Github repository will have the form `cs334hw5_<your user name>`. For example, my repository would be `cs334hw5_dbarowy`. You should have received an invite to commit in the repository in your email. If you did not receive this email, please contact me right away!

**Honor code:** You may collaborate with one or more people on this assignment, but you may not write code together. All submitted work must be your own original work. If you work with a partner, please submit a `collaborators.txt` file that includes their names.

This assignment is due on Thursday, October 18 by 10pm.

## Reading

1. **(Required)** "Lisp Cheat Sheet"
2. **(Required)** "Recursive Functions of Symbolic Expressions and Their Computation by Machine"

## Problems

### Q1. (*0 points*) ..... Using Lisp

For this problem, use the Lisp interpreter `clisp` on the Unix machines in the computer lab. Typing `clisp` will start up the Lisp read-eval-print loop (REPL). This is a good way to experiment with code. Type `(quit)` to quit the REPL.

When developing a solution to a homework problem, however, I recommend putting your Lisp code into a file and then running the interpreter on that file. This mimics the way the graders will be running your code.

To run the program in the file "`example.lisp`", type

```
clisp < example.lisp
```

at the command line. The interpreter will read, evaluate, and print the result of expressions in the file, in order. For example, suppose "`example.lisp`" contains the following:

```
; square a number
(defun square (x) (* x x))

(square 4)
(square (square 3))

(quit)
```

Evaluating this file produces the following output:

```
SQUARE
16
81
Bye.
```

Lisp evaluates the function declaration for `square`, evaluates the two expressions that apply `square` to the given arguments, and then quits. It is important that the program ends with `(quit)` so that `clisp` will exit and return you to the Unix shell. If your program contains an error (or you forget the `(quit)` expression), `clisp` will print an error message and then wait for you to type some input. Type in "`(quit)`" to exit the interpreter, or type "`^D`" (Control-D) to return to the REPL.

The dialect of Lisp we use is similar to what is described in the book, with a few notable exceptions. McCarthy's paper on Lisp mostly uses a never-implemented dialect of Lisp called "M-expressions". The dialect we use today, "S-expressions," are described in the section titled "Representation of S-Functions by S-Expressions." The simplest Lisp reference, however, is the Lisp cheat sheet page in the "Handouts" section of the course webpage which provides a complete list of the Lisp operations needed for this assignment. To really challenge yourself to think functionally, try using higher-order functions (ie, `mapcar` and `apply`) whenever possible.

The following simple examples may help you start thinking as a Lisp programmer.

(a) What is the value of the following expressions? Try to work them out yourself, and verify your answers on the computer:

i. `(car '(inky clyde blinky pinky))`
ii. `(cons 'inky (cdr '(clyde blinky pinky)))`
iii. `(car (car (cdr '(inky (blinky pinky) clyde))))`
iv. `(cons (+ 1 2) (cdr '(+ 1 2)))`
v. `(mapcar #'(lambda (x) (/ x 2)) '(1 3 5 9))`
vi. `(mapcar #'(lambda (x) (car x)) '((inky 3) (blinky 1) (clyde 33)))`
vii. `(mapcar #'(lambda (x) (cdr x)) '((inky 3) (blinky 1) (clyde 33)))`

(b) Write a function called `list-len` that returns the length of a list. Do not use the built-in `length` function in your solution.

```
* (list-len (cons 1 (cons 2 (cons 3 (cons 4 nil)))))

4
* (list-len '(A B (C D)))

3
```

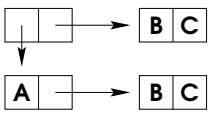
(c) Write a function `double` that doubles every element in a list of numbers. Write this two different ways— first use recursion over lists and then use `mapcar`.

```
* (double '(1 2 3))
(2 4 6)
```

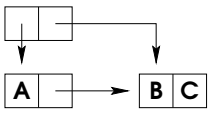
## Q2. (*0 points*) ..... Cons Cell Representations

(a) (*2 points*) Draw the list structure created by evaluating `(cons 'A (cons 'B 'C))`.

(b) (*4 points*) Write a Pure Lisp expression that will result in this representation, with no sharing of the `(B . C)` cell. Explain why your expression produces this structure.



(c) (*4 points*) Write a Pure Lisp expression that will result in this representation, with sharing of the `(B . C)` cell. Explain why your expression produces this structure.




While writing your expressions, use only these Lisp constructs: `lambda` abstraction, function application, the atoms `'A 'B 'C`, and the basic list functions (`cons, car, cdr, atom, eq`). Assume a simple-minded Lisp implementation that does not try to do any clever detection of common subexpressions or advanced memory allocation optimizations.

## Q3. (*0 points*) ..... Recursive List Manipulation

Write a function `merge-list` that takes two lists and joins them together into one large list by alternating elements from the original lists. If one list is longer, the extra part is appended onto the end of the merged list. The following examples demonstrate how to merge the lists together:

```
* (merge-list '(1 2 3) nil)
(1 2 3)

* (merge-list nil '(1 2 3))
(1 2 3)

* (merge-list '(1 2 3) '(A B C))
(1 A 2 B 3 C)

* (merge-list '(1 2) '(A B C D))
(1 A 2 B C D)

* (merge-list '((1 2) (3 4)) '(A B))
((1 2) A (3 4) B)
```

Before writing the function, you should start by identifying the base cases (there are more than one) and the recursive case.

## Q4. (*0 points*) ..... Mapping Functions

Write a function `censor-word` that takes a word as an argument and returns either the word or `XXXX` if the word is a "bad" word:

```
* (censor-word 'lisp)
lisp

* (censor-word 'midterm)
XXXX
```

The lisp expression `(member word '(extension algorithms graphics AI midterm))` evaluates to true if `word` is in the given list.

Use this function to write a `censor` function that replaces all the bad words in a sentence:

```
* (censor '(I NEED AN EXTENSION BECAUSE I HAD AN AI MIDTERM))
(I NEED AN XXXX BECAUSE I HAD A XXXX XXXX)

* (censor '(I LIKE PROGRAMMING LANGUAGES MORE THAN GRAPHICS OR ALGORITHMS))
(I LIKE PROGRAMMING LANGUAGES MORE THAN XXXX OR XXXX)
```

Operations like this that must processes every element in a structure are typically written using mapping functions in a functional language like Lisp. In some ways, mapping functions are the functional programming equivalent of a "for loop", and they are now found in main-stream languages like Python, Ruby, and even Java. Use a map function in your definition of `censor`.