

# Homework 4

Due Sunday, March 8 by 11:59pm

Handout 8  
CSCI 334: Spring 2020

This homework is a *pair programming* assignment. This means that you are required to work with a partner for this lab.

You may choose a lab partner or you may ask to have one assigned. Please inform me of your choice by email by 11:59pm on Monday, March 2.

## Turn-In Instructions

Your work for this lab should be turned in using your assigned GitHub repository.

If you discuss this assignment with anyone but your assigned partner (e.g, a study group), please be sure to include their names in a `collaborators.txt` file in your repository. Note that this assignment should otherwise be anonymized: neither your name nor your partner's name should appear in any of the files submitted with this assignment except the file `collaborators.txt`.

Programming solutions should be typed up and submitted using your partner repository. For example, if your GitHub username is `dbarowy` and you are working with a partner whose username is `wjannen`, look for a repository called `cs334hw4-dbarowy-wjannen` (usernames will be in alphabetical order). Be sure that your work is *committed* and *pushed* to your repository by the due date.

Lisp files should have a `.lisp` suffix. For example, the pair programming problem **Q1b** should appear as the file `q1b.lisp`. Your code should be documented using comments. Comment lines start with a `;`.

**Honor code:** You may collaborate with your partner on all aspects of this assignment, including discussing the problems, work on paper, and writing code. You may also work with students in the class other than your partner, but you may only discuss the problems at a high level (e.g., what the problems mean or general approaches to problems that do not involve code).

This assignment is due on Sunday, March 8 by 11:59pm.

## Reading

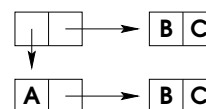
1. **(Required)** "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I," from the course packet.
2. **(Required)** LISP Notes.

## Problems

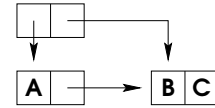
### Q1. (10 points) ..... Cons Cell Representations

- (a) (2 points) Draw the list structure created by evaluating `(cons 'A (cons 'B 'C))`. Include this diagram as an image in your repository.

- (b) (4 points) Write a Pure Lisp expression that will result in the representation to the right, with no sharing of the (B . C) cell. In the comments, explain why your expression produces this structure.



- (c) (4 points) Write a Pure Lisp expression that will result in the representation to the right, with sharing of the (B . C) cell. In the comments, 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`).

## Q2. (10 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.

## Q3. (10 points) ..... Reverse

Write a function `rev` that takes one argument. If the argument is an atom it remains unchanged. Otherwise, the function returns the elements of the list in reverse order:

```
* (rev nil)
nil

* (rev 'A)
A

* (rev '(A (B C) D))
(D (B C) A)

* (rev '((A B) (C D)))
((C D) (A B))
```

## Q4. (10 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 process 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`.

## Q5. (20 points) ..... Working with Structured Data

This part works with the following database of students and grades:

```
;; Define a variable holding the data:
* (defvar grades '((Riley (90.0 33.3))
                    (Jessie (100.0 85.0 97.0))
                    (Quinn (70.0 100.0))))
```

First, write a function `lookup` that returns the grades for a specific student:

```
* (lookup 'Riley grades)
```

```
(90.0 33.3)
```

It should return nil if no one matches.

Now, write a function `averages` that returns the list of student average scores:

```
* (averages grades)
```

```
((RILEY 61.65) (JESSIE 94.0) (QUINN 85.0))
```

You may wish to write a helper function to process one student record (ie, write a function such that `(student-avg '(Riley (90.0 33.3)))` returns `(RILEY 61.65)`, and possibly another helper to sum up a list of numbers). As with `censor` in the previous part, the function `averages` function is most elegantly expressing via a mapping operation (rather than recursion).

We will now sort the averages using one additional Lisp primitive: `sort`. Before doing that, we need a way to compare student averages. Write a method `compare-students` that takes two “student/average” lists and returns true if the first has a lower average and nil otherwise:

```
* (compare-students '(RILEY 61.65) '(JESSIE 94.0))
t
```

```
* (compare-students '(JESSIE 94.0) '(RILEY 61.65))
nil
```

To tie it all together, you should now be able to write:

```
(sort (averages grades) #'compare-students)
```

to obtain

```
((RILEY 61.65) (QUINN 85.0) (JESSIE 94.0))
```

## Q6. (20 points) ..... Deep Reverse

Write a function `deep-rev` that performs a “deep” reverse. Unlike `rev`, `deep-rev` not only reverses the elements in a list, but also deep-reverses every list inside that list.

```
* (deep-rev 'A)
A

* (deep-rev nil)
NIL

* (deep-rev '(A (B C) D))
(D (C B) A)

* (deep-rev '(1 2 ((3 4) 5)))
((5 (4 3)) 2 1)
```

I have defined `deep-rev` on atoms as I did with `rev`.

## Q7. (20 points) ..... Recursive Definitions

Not all recursive programs take the same amount of time to run. Consider, for instance, the following function that raises a number to a power:

```
(defun power (base exp)
  (cond ((eq exp 1) base)
        (t (* base (power base (- exp 1))))))
```

A call to `(power base e)` takes  $e - 1$  multiplication operations for any  $e \geq 1$ . You could prove this time bound by induction on  $e$ :

**Theorem:** A call to `(power b e)`, where  $e \geq 1$ , takes at most  $e - 1$  multiplications.

- *Base case:*  $e = 1$ . `(power b 1)` returns `b` and performs zero multiplications because  $b^1 = b$ .
- *Inductive hypothesis:* For all  $k < e$ , `(power b k)` takes at most  $k - 1$  multiplications.
- *Proof for  $e > 1$ :* Since  $e$  is greater than 1, the “else” branch of `cond` is taken, which
  - (a) performs one multiply operation, and
  - (b) then recursively calls `(power b (- e 1))`.

By induction, we know that the recursive call performs at most  $(e - 1) - 1 = e - 2$  multiplications. Because the result is multiplied by the base, there up to  $e - 2 + 1 = e - 1$  multiplications. Therefore, `power` performs at most  $e - 1$  multiplications.

Multiplication operations are typically very slow relative to other math operations on a computer. Fortunately, there are other means of exponentiation that use fewer multiplications and lead to more efficient algorithms. Consider the following definition of exponentiation:

$$\begin{aligned} b^1 &= b \\ b^e &= (b^{(e/2)})^2 \text{ if } e \text{ is even} \\ b^e &= b * (b^{e-1}) \text{ if } e \text{ is odd} \end{aligned}$$

- (a) Write a Lisp function `fastexp` to calculate  $b^e$  for any  $e \geq 1$  according to these rules. You will find it easiest to first write a helper function to square an integer, and you may wish to use the library function `(mod x y)`, which returns the integer remainder of  $x$  when divided by  $y$ .

- (b) Show that the program you implemented is indeed faster than the original by determining a bound on the number of multiplication operations required to compute (**fastexp base e**). Prove that bound is correct by induction (as in the example proof above), and then compare it to the bound of  $e-1$  from the first algorithm. Include this proof as a comment in your code. Multiline comments are delineated with `#|` and `|#`, as in: `#| ... |#`

Hint: for **fastexp**, it may be easiest to think about the number of multiplications required when exponent  $e$  is  $2^k$  for some  $k$ . Determine the number of multiplies needed for exponents of this form and then use that to reason about an upper bound for the others.

The following property of the log function may be useful in your proof:

$$\log_b(m) + \log_b(n) = \log_b(mn)$$

For example,  $1 + \log_2(n) = \log_2(2) + \log_2(n) = \log_2(2n)$ .

## Q8. (5 points) ..... Bonus Problem

- (a) Using Pure Lisp, implement a binary search tree, where each tree node is a list that stores a number, a left subtree, and a right subtree. The empty subtree should be represented as `nil`. Implement the following functions:
- insert**: Given a tree `t`, inserts a number, returning a new tree.
  - lookup**: Given a tree `t` and a number `n`, returns `T` if `n` is in the tree, otherwise `nil`.
- (b) How many cons cells are created during **insert**?
- (c) If you were to use features from Impure Lisp instead and a slightly different definition of **insert**, do you think that you could reduce the number of cons cells created during insertion? Why or why not?

## Q9. (0 points) ..... Optional Feedback

How was this assignment on a scale of  $\lambda f.\lambda x.x$  to  $\lambda f.\lambda x.f(f(f(x)))$ ?

Do you have any additional comments or feedback that you would like me to know?

Please supply your answer as a `feedback.txt` file.