# Homework 1

Due 14 February

## ━━━ Reading ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━

1. **(Required)** Mitchell, Chapter 3.

2. **(As Needed)** The Lisp Tutorial from the "Links" web page, as needed for the programming questions.

3. **(Optional)** J. McCarthy, Recursive functions of symbolic expressions and their computation by machine, *Comm. ACM* 3,4 (1960) 184–195. You can find a link to this on the cs334 web site. The most relevant sections are 1, 2 and 4; you can also skim the other sections if you like.

## ━━━ Problems ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━

1. (*40 points*) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . Lisp Programming

**Unix Accounts**   We will be working on the Unix lab computers throughout the semester. If you have not used these machines before or don't remember your password, please see Mary or me to obtain a password and verify that you can log in.

The only two applications you will need for this assignment are `xterm` and `emacs`. Both are accessible from the "Applications" menu under "Accessories". If you are not familiar with either of these, I can point you to some useful resources.

**Lisp Programming**   For this problem, use the lisp interpreter on the Unix machines in the computer lab.

You can use the lisp interpreter interactively by running the command `clisp`, which will enter the lisp read-eval-print loop. However, I recommend putting your lisp code into a file and then running the interpreter on that file.

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.
```

It evaluates the function declaration for "square", evaluates the two expressions containing square, and then quits. It is important that the program ends with (quit) so that the lisp interpreter will exit and return you to the Unix shell. If your program contains an error (or you forget the (quit) expression), the lisp interpreter will print an error message and then wait for you to input an expression to evaluate. Just type in "(quit)" at that point to exit the interpreter, or type "^D" (Control-D) to return to the read-eval-print loop.

The dialect of lisp we use is similar to what is described in the book, with a few notable exceptions. See the Lisp notes page on the handouts website for a complete list of the Lisp operations that we have discussed. You should not need anything beyond what is listed there. Try to using higher-order functions (ie, mapcar and apply) where possible.

(a) **Self Check**: The following simple examples may help you start thinking as a Lisp programmer. You do not need to hand in anything for this part.

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)))

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
```

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)
```

(b) **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 0) 1)
        (t (* base (power base (- exp 1))))))
```

There are more efficient means of exponentiation. First write a Lisp function that squares an integer, and then using this function, design a Lisp function `fastexp` which calculates $b^e$ for any $e \geq 0$ by the rule:

$$
\begin{aligned}
b^0 &= 1 \\
b^e &= (b^{(e/2)})^2 \quad \text{if } e \text{ is even} \\
b^e &= b * (b^{e-1}) \quad \text{if } e \text{ is odd}
\end{aligned}
$$

The function (`mod x y`) returns the remainder of x when divided by y. Show that the program you implemented is indeed faster than the original by comparing the number of multiplications that must be done for some exponent $e$ in the first and second algorithms. (Hint: it is easiest to calculate the number of multiplications if the exponent, $e$, is of the form $2^k$ for the second algorithm. Give the answer for exponents of this form and then try to give an upper bound for the others.)

(c) **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.

(d) **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))
```

(e) **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 A 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)
```

(f) **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). Do NOT write `averages` recursively– use a mapping operation.

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))
```

(g) **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`.

(h) **Optional Lisp for the Parenthetically Inclined**

This part is optional. Implement a binary search tree where each node is a list storing a number, a left child, and a right child. Write operations to insert a number, and to check if a number is in the tree. Write this in Pure Lisp. How many cons cells are created on an insert operation? Is there any way to reduce this number? Why or why not? Modify the code to use Impure Lisp features for insertion. How many cons cells are created for insert operations in the new implementation?

Make one file containing the function definitions for the above questions, as well as expressions to demonstrate that the functions work properly. Your code should be documented (comment lines start with ";") and include your name. Include a printout in what you hand in, and submit an electronic copy with the command `turnin -c 334 foo.lisp`, where `foo.lisp` is the name of your source file. You may submit the file more than once if you find a mistake or wish to change what you submitted the first time.

**2.** (*20 points*) ..................................... Conditional Expressions in Lisp

Mitchell, Problem 3.2

**3.** (*10 points*) .......................................... Cons Cell Representations

Mitchell, Problem 3.1

**4.** (*10 points*) ......................................................... Detecting Errors

Evaluation of a Lisp expression can either terminate normally (and return a value), terminate abnormally with an error, or run forever. Some examples of expressions that terminate with an error are (/ 3 0), division by 0; (car 'a), taking the car of an atom; and (+ 3 "a"), adding a string to a number. The Lisp system detects these errors, terminates evaluation, and prints a message to the screen. Suppose that you work at a software company that builds word processing software in Impure Lisp (It's been done: Emacs!). Your boss wants to handle errors in Lisp programs without terminating the computation, but doesn't know how, so your boss asks you to . . .

(a) (*5 points*) . . . implement a Lisp construct (error? E) that detects whether an expression E will cause an error. More specifically, your boss wants evaluation of (error? E) to halt with value *true* if evaluation of E terminates in error and halt with value *false* otherwise. Explain why it is not possible to implement the error? construct as part of the Lisp environment.

(b) (*5 points*) ...implement a Lisp construct (guarded E) that either executes E and returns its value, or if E would halt with an error, returns $0$ without performing any side effects. This could be used to try to evaluate E and if an error would occur, just use $0$ instead. For example,

```
(+ (guarded E) E')   ; just E' if E halts with an error; E+E' otherwise
```

will have the value of E$'$ if evaluation of E would halt in error, and the value of E+E$'$ otherwise. How might you implement the guarded construct? What difficulties might you encounter? Notice that unlike (error? E), evaluation of (guarded E) does not need to halt if evaluation of E does not halt.

**5**. (*10 points*) ................................................ Definition of Garbage

Mitchell, Problem 3.5

**6**. (*15 points*) ............................ Concurrency in Lisp (Bonus Question)

Mitchell, Problem 3.8