Notes

This homework has three types of problems:

Self Check: You are strongly encouraged to think about and work through these questions, but you will not submit answers to them.

Problems: You will turn in answers to these questions.

Pair Programming: This part involves writing Lisp code. You are required to work with a partner on it. You are welcome to choose your own partner, but I can also assist in matching pairs — simply send me email and I will pair you with someone else also looking for a partner. Please do not let finding a partner go until the last minute.

We will be working on the Unix lab computers throughout the semester. The only applications you will need for this assignment are terminal and emacs. To find these after logging in, click on the top icon — a purple spiral — in the tool bar on the left edge of the screen, and then type the name of the application you would like to find into the search box. If you are not familiar with either of these, I can point you to some useful resources.

I encourage you to work in the Unix lab whenever you like, but also keep in mind that you can ssh to our computers from anywhere on campus. Their names are listed on the department's web page, and you can also connect to either of our dedicated computing server, lohani.cs.williams.edu.

Reading

1. (Required) Mitchell, Chapter 3.

2. (As Needed) The Lisp Tutorial from the “Resources” 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.

What To Turn In

Please hand in work in two pieces, one for the Problems and one for the Pair Programming:

Problems: Turn in handwritten or typed answers by the beginning of lecture. Be sure that your work: 1) is stapled, 2) is legible, with clearly marked answers and in the correct order, and 3) includes a list of any students you discussed the problems with.

Pair Programming: Turn in a printout of your code separately from your problem answers. Only one of each pair needs to hand in a printout, but be sure both names are at the top of the file. Turn in an electronic copy as well using the instructions at the end of the programming questions.
S1. (0 points) ........................................... Cons Cell Representations

Mitchell, Problem 3.1

S2. (0 points) .................................................. Using Lisp

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 using higher-order
functions (ie, `mapcar` and `apply`) where 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 (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)

Problems

Q1. (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) …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 (1) halt with value true if evaluation of E would terminate in error, and (2) halt with value false otherwise. Explain why it is not possible to implement the error? construct as part of the Lisp environment.

(b) …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.

Q2. (20 points) Conditional Expressions in Lisp

Mitchell, Problem 3.2
Mitchell, Problem 3.5

Pair Programming

Before starting on the programming, you will need to clone your git repository for this assignment and set it up to share with your partner. The GitLab tutorial on the CS 334 Resources page contains more details about using GitLab and git, but the basic steps are the following. Any of the class staff can help you get things set up if you have not used git before or run into trouble.

You will be working with a partner. Only one of you needs to do the following before working on the Lisp code.

- Go to [https://evolene.cs.williams.edu](https://evolene.cs.williams.edu) (This server is only available on campus.)
- Log in with your CS Unix username and password.
- You should see a project repository named “cs334-s19-hw1-UNIX_ID”.
- Clone that repository to your directory. You will use a command like the following.
  ```
  git clone https://evolene.cs.williams.edu/freund/cs334-s19-hw1-UNIX_ID.git
  ```
  Be sure to clone with HTTPS, and not SSH.
- The repository contains a file in which you will write the following short programming exercises. To push your changes to your GitLab repository, you’ll first need to commit any edits you’ve made:
  ```
  git commit -m "description of changes" -a
  ```
  and then push them:
  ```
  git push
  ```
- You will need to give your partner access to your hw1 repository. To do this, follow the directions in Section 6 of the GitLab tutorial on the class Resources web page. Your repository will now appear in your partner’s account on evolene, and your partner can clone and push changes to it as well.

Once these steps are complete, your partner should log in to evolene and clone the repository in the way described above. Your partner will have also have a repository names “cs334-s19-hw1-PARTNER_UNIX_ID” — you won’t need to clone or modify this in any way. Our submission system will identify partners and figure out which repository was used. The other one will just be ignored.

The only other git command you will likely need is `git pull`, which updates your local copy with whatever changes have been pushed to GitLab since you last pulled.

Please complete the following questions in the `hw1.lisp` file.

(a) **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:

```lisp
(defun power (base exp)
  (cond ((eq exp 1) base)
        (t (* base (power base (- exp 1))))))
```
A call to \((\text{power base } e)\) takes \(e - 1\) multiplication operations for any \(e \geq 1\). You could prove this time bound by induction on \(e\):

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

- **Base case:** \(e = 1\). \((\text{power b } 1)\) returns \(b\) and performs no multiplications, and \(e - 1 = 0\).
- **Ind. hyp:** For all \(k < e\), \((\text{power b } k)\) takes at most \(k - 1\) multiplications.
- **Prove for \(e\):** Since \(e\) is greater than 1, the “else” branch is taken, which calls \((\text{power b } (- e 1))\). The induction hypothesis shows that the recursive call uses \((e - 1) - 1 = e - 2\) multiplications. The result is then multiplied by the base, yielding a total of \(e - 2 + 1 = e - 1\) 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:

\[
\begin{align*}
 b^1 & = b \\
 b^e & = (b^{e/2})^2 \text{ if } e \text{ is even} \\
 b^e & = b \times (b^{e-1}) \text{ if } e \text{ is odd}
\end{align*}
\]

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 \(y\).

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 \((\text{fastexp base } e)\). 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, \(1 + \log_2(n) = \log_2(2) + \log_2(n) = \log_2(2n)\).

(b) **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.

(c) **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))

(d) **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)

* `(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`.

(e) **Working with Structured Data**
This part works with the following database of students and grades:

```lisp
;; 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:
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:

```lisp
* (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:

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

to obtain

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

(f) **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.

```lisp
* (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`.

(g) **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?

**What To Turn In.**

- Your code should be documented (comment lines start with “;”) and include the ID numbers of both partners at the top.
• One of each pair should turn in a printout of that file separate from the answers to the written problems.

• Commit and push all changes to your GitLab repository.

• Verify your commits by navigating to your lab repository in a web browser and examining the version that is stored there.

• The shared repository you are using is either your own or your partners. The other one will be unused. There is no need to do anything to that repository. Our submission scripts will ignore unused repositories and look only at the ones with completed solutions.