Homework 1

Due 14 February

Handout 6
CSCI 334: Spring 2018

Notes

This homework has three types of problems:

Self Check: You are strongly encouraged to think about and work through these questions, and may do so with a partner. Do 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 for this section. If you have a preferred partner, be sure to email me (dbarowy@cs.williams.edu) with your partner's name **by the evening of Tuesday, Feb 6**. If I have not received an email from you by then, I will assign you a partner.

Turn-In Instructions

You must turn in your work for this assignment in two parts, one for the Problems part and one for the Pair Programming part. You will be assigned two GitHub repositories.

Problems: Problem sets should be typed up using LaTeX and submitted using the `hw1-problems` repository. For example, if your GitHub username is `dbarowy`, then your repository will be called `hw1-problems-dbarowy`. Be sure that your work is committed and pushed to your repository by Wednesday, February 14 at 11:59pm.

If you discuss the problem set with your partner or with a study group, please be sure to include their names in a `collaborators.txt` file in your repository.

Pair Programming: Programming solutions should be typed up and submitted using the `hw1-progs` 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 `hw1-progs-dbarowy-wjannen` (usernames will be in alphabetical order). Be sure that your work is committed and pushed to your repository by Wednesday, February 14 at 11:59pm.

Lisp files should have a "`.lisp`" suffix. For example, the pair programming problem **P1** should appear as the file `p1.lisp`. Your code should be documented using comments. Comment lines start with a "`;`".

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 Bailey to obtain a password and verify that you can log in. Mary has posted her hours that she will be available to set up accounts on the door to the UNIX Lab in TCL 312.**

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. For example, if your username is `bcool`, you can connect to `lohani` on the command line by typing: `ssh bcool@lohani.cs.williams.edu`

Reading

1. **(Required)** Mitchell, Chapter 3. Note: be sure to check the book errata on the course webpage for this chapter.
2. **(As Needed)** The Lisp Tutorial from the "Tutorials" section of the course 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 a PDF on the cs334 web site. The most relevant sections are 1, 2 and 4; you can also skim the other sections if you like.

Self Check

S1. (0 points) .... Cons Cell Representations

Mitchell, Problem 3.1

S2. (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. See the Lisp notes page in the "Tutorials" section of the course webpage for a complete list of the Lisp operations needed for this assignment. 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)
```

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 software using Impure Lisp. Your boss wants to handle errors in Lisp programs without terminating the entire computation, but doesn't know how.

(a) You boss asks you to implement a Lisp construct `(error? E)` that detects whether an expression `E` will cause an error. More precisely, your boss wants evaluation of `(error? E)` to (1) halt with value `T` if evaluation of `E` would terminate in error, and (2) halt with value `nil` otherwise. Explain why it is not possible to implement the `(error? E)` construct.

(b) After you finish explaining why `(error? E)` is impossible, your boss proposes an alternative. Instead, your boss wants you to implement a Lisp construct `(guarded E)` that either executes `E` and returns its value, or if `E` halts with an error, returns 0 without performing any side effects. This could be used to try to evaluate `E`, and if an error occurs, to use 0 instead. For example,

```
(+ (guarded E) E2)
```

will have the value of `E2` if evaluation of `E` halts in error, and the value of `E + E2` otherwise. Observe that unlike `(error? E)`, evaluation of `(guarded E)` does not need to halt if evaluation of `E` does not halt.

i. How might you implement the `guarded` construct?

ii. What difficulties might you encounter?

Q2. (10 points) .... Definition of Garbage

Mitchell, Problem 3.5

Pair Programming

P1. (10 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. Multline 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, $1 + \log_2(n) = \log_2(2) + \log_2(n) = \log_2(2n)$.

P2. (6 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.

P3. (6 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))
```

P4. (6 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 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)
```

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`.

P5. (6 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 elegently 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))
```

P6. (6 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`.

P7. (8 points) .... Challenge 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:

  i. `insert`: Given a tree `t`, inserts a number, returning a new tree.
  ii. `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?