Homework 1, due midnight Thursday Mar 1

Full electronic submission due 1st, paper version due Friday in TA mailbox (101A Smith).

Working with people in your section (80)

- A&S 1.6
- Code the procedures `append` and `reverse`. Code each one twice: as a linear process recursion and iteratively. Note: an iterative procedure that calls a recursive process helper is no longer iterative; but it may certainly call other iterative procedures.
- Draw (paper submission only) the box-and-pointer structure of

  ```
  (cons 2 (cons (list 3 4) 5))

  (cons (cons 1 2)(cons 1 ()))

  (list (cons 1 ()) (cons 2 ()) ())
  ```

- A&S 1.30, 1.31, 1.32
- Have we discussed enough Scheme in class to write the "real" version of map? If so, write it; otherwise, identify what is needed.

Do not submit these.

Working alone: for submission

You may **talk** with another student and **share ideas** with another student for these problems, but you MAY NOT look at another student's answers or code. You may of course discuss anything with the TA or professor.

Pay special attention to what is required. If the question asks for drawings, process illustrations, answers, descriptions, etc., be sure to provide those things.

1. A&S 1.16
2. Write a definition, in your own words, of "syntactic sugar".
3. If 2^{30} function calls can be processed each second, calculate the amount of time required to perform the tree recursive fib code from class for (fib 70). Report your answer in sensible units. Show all of your calculations.
4. A&S 1.19
5. Write a procedure which uses the map procedure we wrote in class, and displays every element of a list argument. Put two spaces between each item in the list and a newline after the last item.
6. Modify the previous procedure so that it takes an arbitrary number of arguments instead of a list. If one of the items in the list is itself a list, it should appear on its own line.
7. See the quote.txt file in this directory.

   The purpose of this function is to mimic the actions of a simple cybernetic animal that lives in a one-dimensional world of zeros and ones[1]. Ones represent things to eat, and the animal seeks them out and eats them until none are left. An interaction with chomp will look like this:

[1] This problem adapted with permission from Kathy McCoy.

```
> (chomper '(0 1 0 V 1  1  1 1 0 0) )
(0 1 0 > 1 1 1 1 0 0)
(0 1 > 0 1 1 1 1 0 0)
(0 > 0 0 1 1 1 1 0 0)
(0 V 0 0 1 1 1 1 0 0)
(0 < 0 0 1 1 1 1 0 0)
(0 0 < 0 1 1 1 1 0 0)
(0 0 0 < 1 1 1 1 0 0)
(0 0 0 0 < 1 1 1 0 0)
(0 0 0 0 0 < 1 1 0 0)
(0 0 0 0 0 0 < 1 0 0)
(0 0 0 0 0 0 0 < 0 0)
(0 0 0 0 0 0 0 V 0 0)
"All Done!"
> (chomper '(0 0 V 0 0) )
"All Done!"
>
```

Let's notice a couple of things about the Chomper. First, Chomper itself is represented in one of three ways, depending on its current state. If Chomper is not currently moving in a direction, it is represented as a V. If it is moving left, it is represented as a >, and if it is moving right, a <. Thus the list given to the chomper function consists of a series of zeros and ones and one instance of Chomper in one of its three states.

Suppose Chomper is in its V state. Then, if there is any food to its left (i.e., there are any ones to its left) then it will switch to a state in which it is moving left. If there is no food to its left (i.e., there are NO ones to its left) but there is food to its right, it will switch to a state in which it is moving right. If there is no food left for it to consume (when it is in its V state), it will print out a message.

Once Chomper is started in one direction or another, it will continue in that direction as long as there is any food in that direction at all. When food runs out in that direction, Chomper will not move but will switch into its V state. As long as there is food in a particular direction, Chomper will move one place in that direction, converting ones into zeros as they are met. The state of Chomper is printed out with each change to its state.

I used 45 lines, including white space, to write 7 functions for Chomper. I could easily have written 4 or 10 functions, but 7 fit the way I was thinking about it. First I enumerated the cases I could see, then wrote black box names to handle each one. I sketched a high level function in Scheme, then commented it out (why?). Then I wrote each low level function and tested them. As I wrote them I made a couple of changes to the high level code (in particular I could now see that some of the cases were subsumed by others). When the low level functions were all done, I uncommented the controlling function and tested it.

Note: do not use any Scheme functionality we haven't covered so far.