CSE 341, Autumn 2012, Assignment 5
Racket Macros, Prolog Warmup
Due: Monday Nov 5, 10:00pm

15 points total (3 points per question)

Include appropriate unit tests for each of your top-level functions or rules. For Prolog, you can use helper rules as needed.

You can use up to 2 late days for this assignment.

1. Racket macros: the lecture notes and code for `delay` and `force` included functions `my-delay` and `my-force`. Rewrite `my-delay` as a macro, so that the user doesn't have to manually wrap the delayed expression in a lambda. So the syntax for `my-delay` should be just like Racket's `delay`. Note in particular that you can have multiple expressions in the body. For example, this should work:

   ```
   (my-delay (write "hi there ") (+ 3 4))
   ```

   Rewrite the `my-force` function from the lecture notes if necessary so that it works correctly with your `my-delay` macro. (Or perhaps it will be OK as is.) Leave `my-force` as a function in any case, rather than making it a macro.

   For the unit tests for `my-delay` and `my-force`, you want to demonstrate that the expressions aren't evaluated when you construct the delay; that they are evaluated the first time you use `my-force` and that it returns the correct value; and that additional uses of `my-force` continue to return the correct value without re-evaluating the expressions. You can use side effects judiciously here. The tests need to be automated — do things like checking the value of a counter, rather than expecting the user to look at the output and see if something got printed.

2. Another Racket macro: define a macro `my-and` that does exactly the same thing as the built-in Racket `and`. (Hint: see the handouts for macros, in particular the `my-or` example. Remember that `and` works on an indefinite number of expressions, including 0 expressions.) Include suitable unit tests.

3. Prolog warmup: write a Prolog rule `repeat` that succeeds if the second argument is a list with 0 or more occurences of the first argument. For example, these goals should succeed:

   ```
   repeat(squid,[]).
   repeat(squid,[squid,squid,squid,squid]).
   ```

   and this should fail:

   ```
   repeat(squid,[squid,squid,squid,clam]).
   ```

   Include suitable unit tests for your rule (the above goals are enough, but you can add some others if you want).

   In addition, try your goal with a variable for either the first or second argument. You don't need to include any output from this however. Backtrack a few times if there are more answers available. For example try:

   ```
   repeat(clam,Xs).
   repeat(X,[squid,squid,squid]).
   ```

4. Write a Prolog rule `average` that computes the average of a list of numbers. Fail if the list is empty. (You don't need to worry about lists of things that aren't numbers.)

5. Prolog sentences: write a Prolog rule `sentence` that succeeds if the first argument is the definite article (in other words, the atom "the"), the second article is a noun, and the third article is a verb. Define at least four facts about nouns (so that Prolog has at least four possible nouns to use in sentences), and also at least four facts about verbs. For example, your facts might include:

   ```
   noun(octopus).
   verb(swims).
   ```

   Then the goal `sentence(the, octopus, swims).` should succeed.

   Include a few unit tests for your rule.

   In addition, try your goal with variables for all three arguments. Backtrack if there are more answers available. You don't need to turn in all the output for this. *However, in a comment in your code, say how many different answers you found, and explain why Prolog found that number of answers.*

Turnin: Turn in separate files for your Racket macros and Prolog problem. Both should include appropriate unit tests that the TAs can run.