6.090 IAP '05 - Homework 4

Assigned Thursday January 13th.
Due 10am Friday January 14th.

Submit a print-out of your .scm file. Should you encounter difficulty printing, the file may be emailed to the 6.090 staff.

Problem 1: Syntactic Sugar

1. Desugar the following expressions:

```
(define (foo x)
  (+ x 5))
(let ((x 1))
  x)
(let ((foo (= x 1))
      (bar 7))
  (if foo
      bar
      #f))
(define (weird x y z)     ; this one's odd
  (lambda (foo)
    (+ x y z foo)))
```

2. Evaluate the following expressions (first guess, then check with MITScheme).

```
(define x 5)
(define (y) (+ 7 7))
(let ((x 3))
  (+ x x))
(let ((x (y))
      (y 7))
  (if (> x 3)
      7
      y)
(let ((mit 12))
  (let ((is (+ mit 1)))
    (let ((hard (- is 7)))
```

```
        (+ mit is hard))))
```

Problem 2: Recursive and Iterative processes

1. **Exploration**: Evaluate the following two definitions in MITScheme.

   ```
   (define (remainder x y)
     (if (< x y)
         x
         (remainder (- x y) y)))
   (define (fact n)
     (if (= n 0)
         1
         (* n (fact (- n 1)))))
   ```

   Then evaluate `(fact 10)` with the Stepper (use M-s instead of C-x,C-e). Hold down the space bar and watch how the computation unfolds. How would you describe the text as a whole as it does the evaluation? How indented is it? What does the end (you can use M-> to go to the end of a buffer) look like? Is there a point at which the process of evaluation changes? Keep in mind that the stepper indents two spaces when it is trying to evaluate a subexpression.

   Pop back to scheme and evaluate `(remainder 30 3)` with the Stepper (again, M-s). Hold down the space bar again. How is this computation different? What does the end look like?

   Explain these differences with reference to recursive vs iterative processes. (You need not submit the *Stepper* buffers you produce)

2. The following are two different implementations of `slow-add`, a procedure that adds two numbers which using any arithmetic procedures other than inc and dec:

   ```
   (define (slow-add1 a b)
     (if (= a 0)
         b
         (inc (slow-add1 (dec a) b))))
   (define (slow-add2 a b)
     (if (= a 0)
         b
         (slow-add2 (dec a) (inc b))))
   ```

   For each procedure, indicate whether it gives rise to a recursive or iterative process, and why. Then test it with the stepper to verify your hypothesis (you need not submit the *Stepper* buffer that you produce).

3. Here is the transformation of `fact` from a recursive to an iterative process that we did in class:

```
; recursive fact
(define (fact n)
  (if (= n 0)
      1
      (* n (fact (- n 1)))))
; iterative fact
(define (fact n)
  (fact-helper n 1))
; helper for iterative version
(define (fact-helper n answer)
  (if (= n 0)
      answer
      (fact-helper (- n 1) (* n answer))))
```

   We also wrote `quotient` in class:

```
(define (quotient x y)
  (if (< x y)
      0
      (+ 1 (quotient (- x y) y))))
```

   Rewrite `quotient` to give rise to an iterative process by following the pattern we used for `fact`. Test your resulting procedure to make sure it works like the original. Then verify that it is iterative by using the Stepper.

Problem 3: Lists

In MITScheme, `#f` is the empty-list. Thus, `nil` evaluates to `#f`. So if you're trying to write build a list that prints out like (3 () 4): a list of three elements, with the empty list as the second element, and it prints (3 #f 4), you've written the right thing!

1. Write the box-and-pointer for the given expressions (you'll find this irritating to submit electronically)

```
(cons (cons 1 nil) (cons 2 nil))
(list (list (list 1) 2) 3)
(cons nil nil)
(append (list 3 2) (cons 1 nil))
```

2. Write expression whose values print out like the following:

```
(7)
("this" "is" "yummy")
((()))
(("apples" 3) ("oranges" 2))
```

3. Here is the `length` procedure we wrote in class:

```
Plan:  Base case: empty-list -> length is 0
       Recursive: length whole lst = 1 + length rest of lst
(define (length lst)
  (if (null? lst)
      0
      (+ 1 (length (cdr lst)))))
```

   Write a procedure called `sum-list` which takes in a list of numbers and outputs their sum.

```
(sum-list (list 1 2 3))
;Value: 6
(sum-list (list 7))
;Value: 7
(sum-list nil)
;Value: 0
```

4. **Extra Bonus Problem:** Write a procedure `seven-on-the-end` which takes in a list and returns a new list with 7 on the end.

```
(seven-on-the-end nil)
;Value: (7)
(seven-on-the-end (list 4))
;Value: (4 7)
(seven-on-the-end (list 4 7 5 3))
;Value: (4 7 5 3 7)
```

   Tackle this problem by figuring out the base case, then the one-off base case (ie where the first recursive call results in the base case).
