6.090 - Building Programming Experience
IAP 2006
Problem Set 4 Solutions

1. Desugar the following expressions:

   - ```
     (define (foo x)
       (+ x 5))
     ```
     desugars to:
     ```
     (define foo (lambda (x) (+ x 5)))
     ```
   - ```
     (let ((x 1))
       x)
     ```
     desugars to:
     ```
     ((lambda (x) x) 1)
     ```
   - ```
     (let ((foo (= x 1))
            (bar 7))
       (if foo
           bar
           #f))
     ```
     desugars to:
     ```
     ((lambda (foo bar)
       (if foo
           bar
           #f))
      (= x 1)
      7)
     ```
   - ```
     (define (weird x y z)
       (lambda (foo)
         (+ x y z foo)))
     ```
     desugars to:
     ```
     (define weird
       (lambda (x y z)
         (lambda (foo)
           (+ x y z foo))))
     ```

2. Evaluate the following expressions.

```
> (define x 5)
; no result

> (define (y) (+ 7 7))
; no result

> (let ((x 3))
    (+ x x))
6

> (let ((x (y))
```

```
        (y 7))
    (if (> x 3)
        7
        y))
7

> (let ((mit 12))
    (let ((is (+ mit 1)))
      (let ((hard (- is 7)))
        (+ mit is hard))))
31
```

3. `slow-add` iteratively and recursively.
   You may have found it necessary to write the `inc` and `dec` methods:

   ```
   (define (inc x) (+ x 1))
   (define (dec x) (- x 1))

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

   `slow-add1` is a **recursive** process, so it keeps deferring the `inc` operation. Each time `slow-add1` is called, it has to remember to come back and do an `inc` and it indents another 2 spaces in the stepper. At the end, it reaches the bottom of the recursion and is able to do all the `inc`'s.

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

   `slow-add2` is an **iterative** process. It doesn't have any deferred operations and thus stays at the same indentation level the whole time.

4. Recursive `quotient`:

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

   Iterative version of `quotient`:

   ```
   (define (quotient x y)
     (quotient-helper x y 0))

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

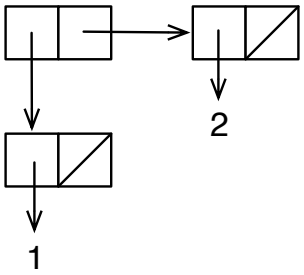

Figure 1: `(cons (cons 1 null) (cons 2 null))`

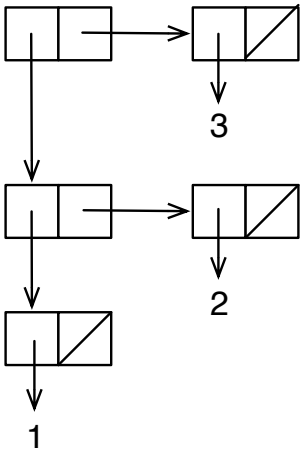

Figure 2: `(list (list (list 1) 2) 3)`

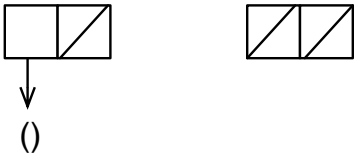

Figure 3: 2 options for `(cons null null)`

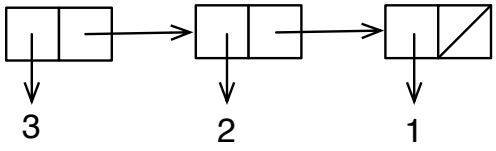

Figure 4: `(append (list 3 2) (cons 1 null))`

5. List Diagrams: see Figures 1, 2, 3, 4

6. Write expressions whose values print out like the following:

```
> (list 7)
(7)

> (list "this" "is" "yummy")
("this" "is" "yummy")

> (list (list null))
((()))
> (list (list (list)))
((()))
> (list (cons null null))
((()))

> (list (list "apples" 3) (list "oranges" 2))
(("apples" 3) ("oranges" 2))
```