Problem 3. (20 points) Recall that Scheme `let` construct (which is *not* recursive) expands into `lambda` expressions as follows:

```
(let [(x1 E1)
      (x2 E2)
      ...
      (xn En)]
  E)}
```

abbreviates

```
((lambda (x1 x2 ... xn) E) E1 E2 ... En)
```

Similarly, the `let*` construct expands into `let` expressions as follows:

```
(let* [(x1 E1)
       (x2 E2)
       ...
       (xn En)]
  E)
```

abbreviates

```
(let [(x1 E1)]
  (let [(x2 E2)]
    ...
      (let [(xn En)]
        E)...))
```

The other binding form in the Scheme `let` family is `letrec`; it has the same scoping rules as the Jam recursive let.

For each of the two expressions on the next page, circle each binding occurrence of a variable and draw arrows from each bound occurrence back to the corresponding binding occurence. For example, given the expression

```
(lambda (x) (+ x 1))
```

the correct answer is:

```
(lambda (x) (+ x 1))
```

1.
   ```
   (let*
     [(fib (lambda (n)
             (letrec
               [(fibhelp (lambda (m fn-1 fn-2)
                           (let [(fn (+ fn-1 fn-2))]
                             (if (zero? m)
                                 fn
                                 (fibhelp (sub1 m) fn fn-1)))))]
               (if (< n 2)
                   1
                   (fibhelp (sub1 n) 1 1)))))
      (fib100 (fib 100))]
     (* fib100 fib100))
   ```

2.
   ```
   (let* [(pair (lambda (x y)
                  (let [(x x)
                        (y y)]
                    (lambda (msg)
                      (cond
                        [(eq? msg 'first) x]
                        [(eq? msg 'second) y]
                        [else (error 'pair "illegal method name ~a" msg)])))))
          (pair (pair 1 2))]
      (pair 'first))
   ```