1. Recall that `mceval.scm` tests true or false using the `true?` and `false?` procedure:

   ```
   (define (true? x) (not (eq? x false)))
   (define (false? x) (eq? x false))
   ```

   Suppose we type the following definition into MCE:

   ```
   MCE> (define true false)
   ```

   What would be returned by the following expression:

   ```
   MCE> (if (= 2 2) 3 4)
   A: 3
   B: 4
   C: ERROR
   ```

2. Suppose we type the following into `mc-eval`:

   ```
   MCE> (define `x (* x x))
   ```

   This expression evaluates without error. What would be returned by the following expressions?

   ```
   MCE> quote
   A: ERROR: unbound variable quote
   B: (compound-procedure quote ((* x x)) <procedure-env>)

   MCE> (quote 10)
   A: 10
   B: 100
   ```

3. Write a procedure `useless-square` that performs the squaring function correctly the first time you call it, and thereafter only returns what it returned the first time. For example,
   ```
   > (useless-square 5)
   25
   > (useless-square 10)
   25
   > (useless-square 3)
   25
   ```

4. What does the following sequence of expressions return:
   ```
   > (define foo 100)
   > (define (set-fooer! y) (set! foo y))
   > (define (bar x)
       (let ((foo 10))
          (set-fooer! 20)
          foo))
   > (bar 1)
   ```

   LEXICAL SCOPING: _____ DYNAMIC SCOPING: _____
   ```
   > foo
   ```

   LEXICAL SCOPING: _____ DYNAMIC SCOPING: _____

5. **For each evaluator, will the following expression return a value or cause an error? Circle VALUE or ERROR for each.**

```
> (let ((a 3)
        (b a))
    (+ a 4))
```

VALUE   ERROR   The MCE
VALUE   ERROR   Analyzing evaluator
VALUE   ERROR   Lazy evaluator
VALUE   ERROR   The MCE with dynamic scope

For each evaluator, will the following expression return a value or cause an error? Circle VALUE or ERROR for each.

```
> (let ((a 3)
        (b a))
    (+ a b)) ;; this line different from the one above
```

VALUE   ERROR   The MCE
VALUE   ERROR   Analyzing evaluator
VALUE   ERROR   Lazy evaluator
VALUE   ERROR   The MCE with dynamic scope

6. **Which of the following interactions will execute faster or the same in the analyzing evaluator than in the original metacircular evaluator? Circle FASTER or SAME for each.**

```
> (define (gauss-recur n) ;; sum of #s from 1 to n
    (if (= n 1)
        1
        (+ n (gauss-recur (- n 1)))))
> (gauss-recur 1000)
```

Analyzing will be: FASTER   SAME

```
> (define (gauss n) ;; sum of #s from 1 to n
    (/ (* (+ n 1) n) 2)
> (gauss 1000)
```

Analyzing will be: FASTER   SAME

7. **Rotating lists is fun, so let's do some. For the following, assume that only the rule `append` has been defined, as in the lecture:**

   a. **Implement a rule `rotate-forward` so that:**

   ```
   (rotate-forward (1 2 3 4) ?what) ==> ?what = (2 3 4 1).
   ```

   That is, the second list is the first list with the first element attached to the end instead. Assume the list is non-empty.

b. Let's get both sides of the story. We'd like a rule `rotate-backward` so that

```
(rotate-backward (1 2 3 4) ?what) ==> ?what = (4 1 2 3)
```

That is, the second list is the first list with the last element attached to the front instead. You may define other helper rules if you'd like.

8. **Write a procedure, `(tree-member? x tree)` that takes in an element and a general tree, and returns `#t` if `x` is part of `tree`, and `#f` otherwise.**

9. **Write a procedure, (num-sum exp) that takes in a valid Scheme expression, and returns the sum of all numbers that occurs in that expression. For example,**
```
(num-sum`(if (= 2 3) (lambda(x) (+ x 3)) 10)) ==> 18
```

10. This is an attempt to write `(remove-first! ls x)`, which destructive removes the first instance of x:
```
(define (remove-first! ls x)
    (cond ((null? ls) '())
          ((eq? (car ls) x) (set! ls (cdr ls)))
          ((eq? (cadr ls) x) (set-cdr! ls (cddr ls)))
          (else (remove-first! (cdr ls) x))))
```

Suppose `(define L '(a b c))`. What happens to `L` when we do:

a. `(remove-first! L 'a)`

   `L ==>` _____

b. `(remove-first! L 'b)`

   `L ==>` _____

c. `(remove-first! L 'z)`

   `L ==>` _____

11. Fill in the blanks:
```
(define foo
    (let ((L (list 1)))
        (lambda (a)
            (let ((M (cons a L)))
                (lambda (b)
                    (set! L (cons (+ a 1) L))
                    (set! a (+ a 2))
                    (set-car! (cdr M) (+ 3 (cadr M)))
                    (set! b (+ b 4))
                    (list a b L M)))))))
```

```
Stk> (define f (foo 1))
Stk> (f 2)


_____


Stk> (define g (foo 2))
Stk> (g 1)


_____


Stk> (f 2)


_____


Stk> (g 1)


_____
```

12. **There are just too many things to do and too much to keep track of during midterms week. We'd like to implement a system that helps us keep track of what we need to do. To that end, we propose a `job` class. A `job` object is a certain amount of work required; you can do some amounts of work until the work is done. A job also has an automatically assigned id. A job is "done" when there's no more work to do for that job. We also want to keep track of number of jobs we've created. We'd like to interact this way:**

```
STk> (define job1 (instantiate job '(take out trash) 10))
job1
STk> (define job2 (instantiate job '(eat dinner) 5))
job2
STk> (define job3 (instantiate job '(study for cs61a) 100000))
job3
STk> (ask job1 'id)
0
STk> (ask job2 'id)
1
STk> (ask job2 'work-to-do)
5
STk> (ask job1 'do-work 10)
okay
STk> (ask job1 'work-to-do)
0
STk> (ask job1 'done?)
#t
STk> (ask job1 'do-work 10)
(but you are already done!)
STk> (ask job 'number-of-jobs)
3

(define-class (job description work-to-do)
```