1. Write procedure \(\text{remberSecond a lat}\) which removes the second occurrence of atom \(a\) in \(\text{lat}\). So \(\text{remberSecond a '(a b c a b c a)}\) returns \(\text{(a b c b c a)}\) and 
\(\text{remberSecond a '(a b o b)}\) returns \(\text{(a b o b)}\)

\[
\text{(define rember}\n\begin{align*}
\text{(lambda (a lat)} \n\text{(cond} \n& \quad [(null? lat) null] \n& \quad [(eq? a (car lat)) (cdr lat)] \n& \quad [else (cons (car lat) (rember a (cdr lat))))] \n\text{))})\n\end{align*}
\]

\[
\text{(define remberSecond}\n\begin{align*}
\text{(lambda (a lat)} \n\text{(cond} \n& \quad [(null? lat) null] \n& \quad [(eq? a (car lat)) (cons a (rember a (cdr lat)))] \n& \quad [else (rember2 a (cdr lat)))] \n\text{))})\n\end{align*}
\]

2. Use \(\text{fold}\) to write \(\text{(count a lat)}\) which returns the number of times atom \(a\) occurs in \(\text{lat}\). 
\(\text{(count a '(a b r a c a d a b r a) ) returns 5}\)

\[
\text{(define count}\n\begin{align*}
\text{(lambda (a lat)} \n\text{(fold (lambda (x y) (if (eq? x a) (+ 1 y) y)) 0 lat))})\n\end{align*}
\]

3. Now use \(\text{map}\) and \(\text{apply}\) to write \(\text{(count* a L)}\) which returns the number of times atom \(a\) occurs in general list \(\text{L}\). 
\(\text{(count* a '(a b r (a c) a (d (a b (r a)))))) returns 5.}\)

\[
\text{(define count*}\n\begin{align*}
\text{(lambda (a L)} \n\text{(cond} \n& \quad [(null? L) 0] \n& \quad [(atom? L) (if (eq? a L) 1 0)] \n& \quad [else (apply + (map (lambda (t) (count* a t)) L)))] \n\text{))})\n\end{align*}
\]

Note that you need to have a case that handles recursing down to atoms, because when you map function \(f\) onto a \(\text{lat}\) like ‘(1 2 3) the arguments that are given to \(f\) are atoms.
4. Suppose we have a list of pairs of atoms and numbers representing people and their ages, such as ‘((mary 18) (tom 16) (bob 62) (sylvia 23)) Give a Scheme function (oldest age-list) that returns the atom representing the oldest person in the list. (oldest ‘((mary 18) (tom 16) (bob 62) (sylvia 23)) should return ‘bob. The easiest way to do this is to first find the pair with the largest age, such as ‘(bob 62), then taking the car of that to get the person with the oldest age.

(define oldest
  (lambda (age-list)
    (car (oldest-pair age-list)))))

(define oldest-pair
  (lambda (age-list)
    (fold (lambda (x y) (if (> (cadr x) (cadr y)) x y)) '(a -1) age-list)))

5. An environment associates values with symbols; give it a symbol and it returns the value bound to that symbol. I want to represent environments with procedures, so if I have symbol x bound to 3 in env, then (env ‘x) returns 3. If y is not bound to anything in env then (env ‘y) returns ‘error. Write procedure (extend-env syms vals old-env) that extends old-env with bindings of syms to vals and returns the new environment. For example if we create new-env as

(define new-env (extend-env ‘(x y z) ‘(1 2 3) old-env))
then (new-env ‘x) returns 1, (new-env ‘y) returns 2 and (new-env ‘z’) returns 3.

There are lots of ways to do this. I think of this as a sequence of environment extensions, one for each entry of syms and vals:

(define extend-env
  (lambda (syms vals old-env)
    (let ((extend
          (lambda (s v old)
            (lambda (t)
              (if (eq? t s) v (old t))))))
      (cond
       [(null? syms) old-env]
       [else (extend (car syms) (car vals) (extend-env (cdr syms) (cdr vals) old-env))])))

One way or another, you need to make a function that takes a symbol and looks up its value.
6. Explain in one or two sentences what it means for a procedure to have state. Give an example of a procedure with state and an example of one without state.

A procedure has state if it can return different values over time for the same arguments. This means that what it returns depends on values in its environment that can change.

Here is a procedure without state: `(lambda (x) (+ x 1))` No matter how many times you evaluate the result of giving this argument 10, it always returns 11.

Here is a procedure with state:

```
(define state
  (let ([count 0])
    (lambda ()
      (set! count (+ 1 count))
      count)))
```

This returns the number of times it has been called.

7. We said the Y-combinator is

```
(define YC
  (lambda (a)
    (lambda (F)
      (F F))
    (lambda (f)
      (a (lambda (x) ((f f) x)))))
```

This may be the wackiest definition you have ever seen. Explain in one or two sentences what the significance of the Y-combinator is.

The Y-Combinator creates recursion. It allows us to make recursive but unnamed lambda expressions, so we can get recursive functions without adding to the global environment.