12.
Take Cover

What is (*multirember a lat*)
where *a* is tuna
and
lat is (shrimp salad tuna salad and tuna)

(shrimp salad salad and),
but we already knew that from chapter 3.

Does *a* change as *multirember* traverses *lat*

No, *a* always stands for tuna.

Well, wouldn't it be better if we did not have to remind *multirember* for every natural recursion that *a* still stands for tuna

Yes, it sure would be a big help in reading such functions, especially if several things don't change.

That's right. Do you think the following definition of *multirember* is correct?

```
(define multirember
  (lambda (a lat)
    ((Y (lambda (mr)
          (lambda (lat)
            (cond
              ((null? lat) (quote ()))
              ((eq? a (car lat))
               (mr (cdr lat)))
              (else (cons (car lat)
                          (mr (cdr lat))))))))
     lat)))
```

Whew, the *Y* combinator in the middle looks difficult.

What is this function?

```
(define ???
  ((lambda (le)
     ((lambda (f) (f f))
      (lambda (f)
        (le (lambda (x) ((f f) x))))))
   (lambda (length)
     (lambda (l)
       (cond
         ((null? l) 0)
         (else
           (add1 (length (cdr l)))))))))
```

It is the function *length* in the style of chapter 9, using *Y*.

```
(define length
  (Y (lambda (length)
       (lambda (l)
         (cond
           ((null? l) 0)
           (else
             (add1 (length (cdr l)))))))))
```

And what is special about it?

We do not use (**define** ...) to make *length* recursive. Using Y on a function that looks like *length* creates the recursive function.

So is Y a special version of (**define** ...)

Yes, that's right. But we also agreed that the definition with (**define** ...) is easier to read than the definition with Y.

That's right. And we therefore have another way to write this kind of definition.

```
(define multirember
  (lambda (a lat)
    ((letrec[1]
       ((mr (lambda (lat)
              (cond
                ((null? lat) (quote ()))
                ((eq? a (car lat))
                 (mr (cdr lat)))
                (else
                  (cons (car lat)
                    (mr (cdr lat))))))))
        mr)
      lat)))
```

But if all we want is a recursive function *mr*, why don't we use this?

```
(define mr
  (lambda (lat)
    (cond
      ((null? lat) (quote ()))
      ((eq? a (car lat))
       (mr (cdr lat)))
      (else
        (cons (car lat)
          (mr (cdr lat)))))))
```

```
(define multirember
  (lambda (a lat)
    (mr lat)))
```

[1] L: `(labels ((mr (lat) ...)) (function mr))`

Because (**define** ...) does not work here.

Why not?

The definition of *mr* refers to *a* which stands for the atom that *multirember* needs to remove from *lat*

Okay, that's true, though obviously *a* refers to the first name in the definition of the function *multirember*.

Do you remember that names don't matter?

Yes, we said in chapter 9 that all names are equal. We can even change the names, as long as we do it consistently.

Correct. If we don't like *lat*, we can use *a-lat* in the definition of *multirember* as long as we also re-name all occurrences of *lat* in the body of the (**lambda** ...).

Yes, we could have used the following definition and nothing would have changed:

```
(define multirember
  (lambda (a a-lat)
    (mr a-lat)))
```

Correct again. And this means we should also be able to use *b* instead of *a* because

```
(define id
  (lambda (a)
    a))
```

is the same as

```
(define id
  (lambda (b)
    b))
```

Yet if we used *b* in the definition of *multirember*

```
(define multirember
  (lambda (b a-lat)
    (mr a-lat)))
```

the *a* in *mr* would no longer make any sense.

Yes: the name *a* makes sense only inside the definition of *multirember*. In general, the names for a function's arguments make sense only inside of (**lambda** ...).

Okay, that explains things.

And that is precisely why we need
(**letrec** ...)
What do you think is the purpose of the nested box?

It separates the two parts of a (**letrec** ...): the naming part, which is the nested box, and the value part, which is *mr*.

Is the nested box important otherwise?

No, the nested box is merely an annotation that we use to help distinguish the two parts of (**letrec** ...). Once we get accustomed to the shape of (**letrec** ...), we will stop drawing the inner box.

What do we use the naming part for?

The naming part defines a recursive function though unlike **defined** functions; a function defined in the naming part of (**letrec** ...) knows all the arguments of all the surrounding (**lambda** ...) expressions.

And the value part?

It tells us what the result of the (**letrec** ...) is. It may refer to the named recursive function.

Does this mean that
(**letrec** ((*mr* ...)) *mr*)
defines and returns a recursive function?

Precisely. Isn't that a lot of parentheses for saying just that?

Yes, but they are important.

Okay, let's go on.

What is the value of
((**letrec** ((*mr* ...)) *mr*) *lat*)

It is the result of applying the recursive function *mr* to *lat*.

What is the value of (*multirember a lat*)
where *a* is pie
and
lat is (apple custard pie linzer pie torte)

(apple custard linzer torte),
but we already knew this.

How can we determine this value?

That's more interesting.

The first line in the definition of *multirember* is no longer (**cond** ...) but
((**letrec** ((*mr* ...)) *mr*) *lat*)
What does this mean?

We said that it defines the recursive function *mr* and applies it to *lat*.

What is the first line in *mr*

It is something we are quite familiar with: (**cond** ...). We now ask questions the way we did in chapter 2.

What is the first question?

(*null? lat*), which is false.

And the next question?

(*eq?* (*car lat*) *a*), which is false.

Why?

Because *a* still stands for pie, and (*car lat*) is apple.

That's correct: *mr* always knows about *a* which doesn't change while we look through *lat*

Yes.

Is it as if *multirember* had defined a function mr_{pie} and had used it on *lat*

```
(define mr_pie
  (lambda (lat)
    (cond
      ((null? lat) (quote ()))
      ((eq? (car lat) (quote pie))
       (mr_pie (cdr lat)))
      (else (cons (car lat)
                  (mr_pie (cdr lat)))))))
```

Correct, and the good thing is that no other function can refer to mr_{pie}.

Why is **define** underlined?

We use (**define** ...) to express that the underlined definition does not actually exist, but imagining it helps our understanding.

Is it all clear now?

This is easy as apple pie.

Would it make any difference if we changed the definition a little bit more like this?

```
(define multirember
  (lambda (a lat)
    (letrec
      ((mr (lambda (lat)
             (cond
               ((null? lat) (quote ()))
               ((eq? a (car lat))
                (mr (cdr lat)))
               (else
                 (cons (car lat)
                   (mr (cdr lat))))))))
       (mr lat))))
```

The difference between this and the previous definition isn't that big.

(Look at the third and last lines.)

The first line in (**lambda** (*a lat*) ...) is now of the shape

(**letrec** ((*mr* ...)) (*mr lat*))

Yes, so *multirember* first defines the recursive function *mr* that knows about *a*.

And then?

The value part of (**letrec** ...) uses *mr* on *lat*, so from here things proceed as before.

That's correct. Isn't (**letrec** ...) easy as pie?

We prefer (linzer torte).

Is it clear now what (**letrec** ...) does?

Yes, and it is better than *Y*.

The Twelfth Commandment

Use (letrec ...) to remove arguments that do not change for recursive applications.

How does *rember* relate to *multirember*

The function *rember* removes the first occurrence of some given atom in a list of atoms; *multirember* removes all occurrences.

Can *rember* also remove numbers from a list of numbers or S-expressions from a list of S-expressions?

Not really, but in *The Little Schemer* we defined the function *rember-f*, which given the right argument could create those functions:

```
(define rember-f
  (lambda (test?)
    (lambda (a l)
      (cond
        ((null? l) (quote ()))
        ((test? (car l) a) (cdr l))
        (else (cons (car l)
                    ((rember-f test?) a
                     (cdr l))))))))
```

Give a name to the function returned by
  (*rember-f test?*)
where
  test? is *eq?*

```
(define rember-eq? (rember-f test?))
```

where
  test? is *eq?*.

Is *rember-eq?* really *rember*

It is, but hold on tight; we will see more of this in a moment.

Can you define the function *multirember-f* which relates to *multirember* in the same way *rember-f* relates to *rember*

That is not difficult:

```
(define multirember-f
  (lambda (test?)
    (lambda (a lat)
      (cond
        ((null? lat) (quote ()))
        ((test? (car lat) a)
         ((multirember-f test?) a
          (cdr lat)))
        (else (cons (car lat)
                    ((multirember-f test?) a
                     (cdr lat))))))))
```

Explain in your words what *multirember-f* does.

Here are ours:
"The function *multirember-f* accepts a function *test?* and returns a new function. Let us call this latter function *m-f*. The function *m-f* takes an atom *a* and a list of atoms *lat* and traverses the latter. Any atom *b* in *lat* for which (*test? b a*) is true, is removed."

Is it true that during this traversal the result of (*multirember-f test?*) is always the same?

Yes, it is always the function for which we just used the name *m-f*.

Perhaps *multirember-f* should name it *m-f*

Could we use (**letrec** ...) for this purpose?

Yes, we could define *multirember-f* with (**letrec** ...) so that we don't need to re-determine the value of
(*multirember-f test?*)

```
(define multirember-f
  (lambda (test?)
    (letrec
      ((m-f
         (lambda (a lat)
           (cond
             ((null? lat) (quote ()))
             ((test? (car lat) a)
              (m-f a (cdr lat)))
             (else
               (cons (car lat)
                     (m-f a (cdr lat))))))))
      m-f)))
```

Is this a new use of (**letrec** ...)?

No, it still just defines a recursive function and returns it.

True enough.

What is the value of (*multirember-f test?*) where
test? is *eq?*

It is the function *multirember*:

```
(define multirember
  (letrec
    ((mr (lambda (a lat)
           (cond
             ((null? lat) (quote ()))
             ((eq? (car lat) a)
              (mr a (cdr lat)))
             (else
               (cons (car lat)
                 (mr a (cdr lat))))))))
      mr))
```

Did you notice that no (**lambda** ...) surrounds the (**letrec** ...)

It looks odd, but it is correct!

Could we have used another name for the function named in (**letrec** ...)

Yes, *mr* is *multirember*.

Is this another way of writing the definition?

Yes, this defines the same function.

```
(define multirember
  (letrec
    ((multirember
       (lambda (a lat)
         (cond
           ((null? lat) (quote ()))
           ((eq? (car lat) a)
            (multirember a (cdr lat)))
           (else
             (cons (car lat)
               (multirember a
                 (cdr lat))))))))
      multirember))
```

Since (**letrec** ...) defines a recursive function and since (**define** ...) pairs up names with values, we could eliminate (**letrec** ...) here, right?

Yes, we could and we would get back our old friend *multirember*.

```
(define multirember
  (lambda (a lat)
    (cond
      ((null? lat) (quote ()))
      ((eq? (car lat) a)
       (multirember a (cdr lat)))
      (else
        (cons (car lat)
          (multirember a (cdr lat)))))))
```

Here is *member?* again:

```
(define member?
  (lambda (a lat)
    (cond
      ((null? lat) #f)
      ((eq? (car lat) a) #t)
      (else (member? a (cdr lat))))))
```

So?

What is the value of (*member? a lat*)
where *a* is ice
and
lat is (salad greens with pears brie cheese
frozen yogurt)

#f,
ice cream is good, too.

Is it true that *a* remains the same for all natural recursions while we determine this value?

Yes, *a* is always ice. Should we use The Twelfth Commandment?

Yes, here is one way of using (**letrec** ...) with this function:

```
(define member?
  (lambda (a lat)
    ((letrec
       ((yes? (lambda (l)
                (cond
                  ((null? l) #f)
                  ((eq? (car l) a) #t)
                  (else (yes? (cdr l)))))))
       yes?)
     lat)))
```

Do you also like this version?

Here is an alternative:

```
(define member?
  (lambda (a lat)
    (letrec
      ((yes? (lambda (l)
               (cond
                 ((null? l) #f)
                 ((eq? (car l) a) #t)
                 (else (yes? (cdr l)))))))
      (yes? lat)))))
```

Did you notice that we no longer use nested boxes for (**letrec** ...)

Yes. We are now used to the shape of (**letrec** ...) and won't confuse the naming part with the value part anymore.

Do these lists represent sets?
(tomatoes and macaroni)
(macaroni and cheese)

Yes, they are sets because no atom occurs twice in these lists.

Do you remember what (*union set1 set2*) is where
set1 is (tomatoes and macaroni casserole)
and
set2 is (macaroni and cheese)

(tomatoes casserole macaroni and cheese).

Write *union*

```
(define union
  (lambda (set1 set2)
    (cond
      ((null? set1) set2)
      ((member? (car set1) set2)
       (union (cdr set1) set2))
      (else (cons (car set1)
                  (union (cdr set1) set2))))))
```

Is it true that the value of *set2* always stays the same when determining the value of (*union set1 set2*)

Yes,
because *union* is like *rember* and *member?* in that it takes two arguments but only changes one when recurring.

Is it true that we can rewrite *union* in the same way as we rewrote *rember*

Yes, and it is easy now.

```
(define union
  (lambda (set1 set2)
    (letrec
      ((U (lambda (set)
            (cond
              ((null? set) set2)
              ((member? (car set) set2)
               (U (cdr set)))
              (else (cons (car set)
                          (U (cdr set))))))))
       (U set1)))))
```

Could we also have written it like this?

Yes.

```
(define union
  (lambda (set1 set2)
    (letrec
      ((A (lambda (set)
            (cond
              ((null? set) set2)
              ((member? (car set) set2)
               (A (cdr set)))
              (else (cons (car set)
                          (A (cdr set))))))))
       (A set1)))))
```

Correct: *A* is just a name like *U*
Does it matter what name we use?

Absolutely not, but choose names that matter to you and everyone else who wants to enjoy your definitions.

So why do we choose the name *U*

To keep the boxes from getting too wide, we use single letter names within (**letrec** ...) for such minor functions.

Can you think of a better name for U

This should be an old shoe by now.

Now, does it work?

It should.

Explain in your words how the new version of *union* works.

Our words:
"First, we define another function U that *cdr*s down *set*, *cons*ing up all elements that are not a member of *set2*. Eventually U will *cons* all these elements onto *set2*. Second, *union* applies U to *set1*."

How does U know about *set2*

Since U is defined using (**letrec** ...) inside of *union*, it knows about all the things that *union* knows about.

And does it have to pass around *set2*

No, it does not.

How does U know about *member?*

Everyone knows the function *member?*.

Does it mean that the definition of *union* depends on the definition of *member?*

It does, but *member?* works, so this is no problem.

Suppose we had defined *member?* as follows.

```
(define member?
  (lambda (lat a)
    (cond
      ((null? lat) #f)
      ((eq? (car lat) a) #t)
      (else (member? (cdr lat) a)))))
```

But this would confuse *union*!

Why?

Because this *member?* takes its arguments in a different order.

What changed?	Now *member?* takes a list first and an atom second.
Does *member?* work?	It works in that we can still use this new definition of *member?* to find out whether or not some atom is in a list.
But?	With this new definition, *union* will no longer work.
Oh?	Yes, because *union* assumes that *member?* takes its arguments in a certain order.
Perhaps we should avoid this.	How?
Well, (**letrec** ...) can define more than just a single function.	Nobody said so.
Didn't you notice the extra pair of parentheses around the function definitions in (**letrec** ...)	Yes.
With (**letrec** ...) we can define more than just one function by putting more than one function definition between the extra pair of parentheses.	This could help with *union*.

Here is a skeleton:

```
(define union
  (lambda (set1 set2)
    (letrec
      ...
      (U set1))))
```

Fill in the dots.

```
...
((U (lambda (set)
      (cond
        ((null? set) set2)
        ((member? (car set) set2)
         (U (cdr set)))
        (else (cons (car set)
                    (U (cdr set)))))))
 (member?
   (lambda (a lat)
     (cond
       ((null? lat) #f)
       ((eq? (car lat) a) #t)
       (else (member? a (cdr lat)))))))
...
```

The Thirteenth Commandment

Use (letrec ...) to hide and to protect functions.

Could we also have written this?

```
(define union
  (lambda (set1 set2)
    (letrec
      ((U (lambda (set)
            (cond
              ((null? set) set2)
              ((M? (car set) set2)
               (U (cdr set)))
              (else (cons (car set)
                          (U (cdr set)))))))
       (M? (lambda (a lat)
             (cond
               ((null? lat) #f)
               ((eq? (car lat) a) #t)
               (else
                 (M? a (cdr lat)))))))
      (U set1))))
```

Presumably.

Are we happy now?

Well, almost.

Almost?

The definition of *member?* inside of *union* ignores The Twelfth Commandment.

It does?

Yes, the recursive call to *member?* passes along the parameter *a*.

And its value does not change?

No, it doesn't!

So we can write something like this?

```
(define union
  (lambda (set1 set2)
    (letrec
      ((U (lambda (set)
            (cond
              ((null? set) set2)
              ((M? (car set) set2)
               (U (cdr set)))
              (else (cons (car set)
                          (U (cdr set)))))))
       (M? ...))
      (U set1))))
```

Yes, and here is how we fill in the dots:

```
...
(lambda (a lat)
  (letrec
    ((N? (lambda (lat)
           (cond
             ((null? lat) #f)
             ((eq? (car lat) a) #t)
             (else (N? (cdr lat)))))))
    (N? lat)))
...
```

Now we are happy, right?

Yes!

Did you notice that *set2* is not an argument of *U*

It doesn't have to be because *union* knows about *set2* and *U* is inside of *union*.

Do we know enough about *union* now?

Yes, we do!

Do we deserve a break now?

We deserve dinner or something equally substantial.

True, but hold the dessert.

Why?

We need to protect a few more functions.

Which ones?

Do you remember *two-in-a-row?*

Sure, it is the function that checks whether some atom occurs twice in a row in some list. It is a perfect candidate for protection.

Yes, it is. Can you explain why?

Here are our words:

"Auxiliary functions like *two-in-a-row-b?* are always used on specific values that make sense for the functions we want to define. To make sure that these minor functions always receive the correct values, we hide such functions where they belong."

So how do we hide *two-in-a-row-b?*

The same way we hide other functions:

```
(define two-in-a-row?
  (lambda (lat)
    (letrec
      ((W (lambda (a lat)
            (cond
              ((null? lat) #f)
              (else (or (eq? (car lat) a)
                        (W (car lat)
                           (cdr lat))))))))
      (cond
        ((null? lat) #f)
        (else (W (car lat) (cdr lat)))))))
```

Does the minor function *W* need to know the argument *lat* of *two-in-a-row?*

No, *W* also takes *lat* as an argument.

Is it then okay to hide *two-in-a-row-b?* like this:

```
(define two-in-a-row?
  (letrec
    ((W (lambda (a lat)
          (cond
            ((null? lat) #f)
            (else (or (eq? (car lat) a)
                      (W (car lat)
                         (cdr lat))))))))
    (lambda (lat)
      (cond
        ((null? lat) #f)
        (else (W (car lat) (cdr lat))))))))
```

Yes, it is a perfectly safe way to protect the minor function W. It is still not visible to anybody but *two-in-a-row?* and works perfectly.

Good, let's look at another pair of functions.

Let's guess: it's *sum-of-prefixes-b* and *sum-of-prefixes*.

Protect *sum-of-prefixes-b*

```
(define sum-of-prefixes
  (lambda (tup)
    (letrec
      ((S (lambda (sss tup)
            (cond
              ((null? tup) (quote ()))
              (else
                (cons (+ sss (car tup))
                  (S (+ sss (car tup))
                     (cdr tup))))))))
      (S 0 tup))))
```

Is S similar to W in that it does not rely on *sum-of-prefixes*'s argument?

It is. We can also hide it without putting it inside (**lambda** ...) but we don't need to practice that anymore.

We should also protect *scramble-b*. Here is the skeleton:

```
(define scramble
  (lambda (tup)
    (letrec
      ((P ...))
      (P tup (quote ())))))
```

Fill in the dots.

```
...
(lambda (tup rp)
  (cond
    ((null? tup) (quote ()))
    (else (cons (pick (car tup)
                      (cons (car tup) rp))
                (P (cdr tup)
                   (cons (car tup) rp))))))
...
```

Can we define *scramble* using the following skeleton?

```
(define scramble
  (letrec
    ((P ...))
    (lambda (tup)
      (P tup (quote ())))))
```

Yes, but can't this wait?

Yes, it can. Now it *is* time for dessert.

How about black currant sorbet?
