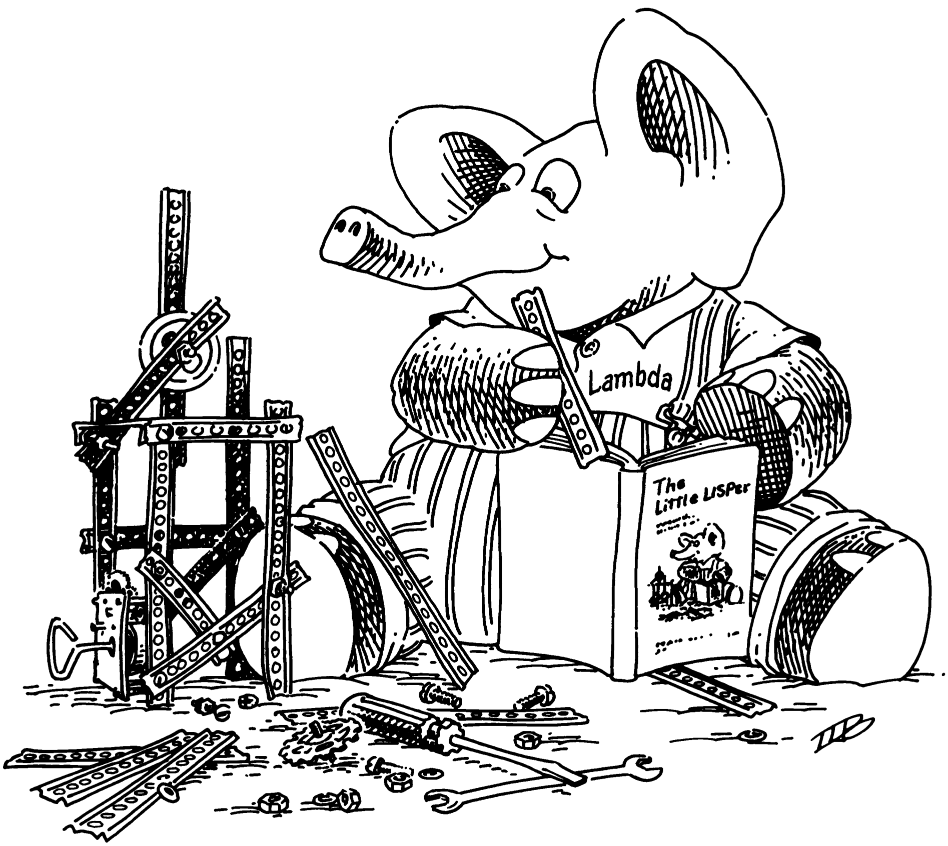
8. Lambda the Ultimate



Remember what we did in *rember* and *insertL* at the end of chapter 5?	We replaced *eq?* with *equal?*
Can you write a function *rember-f* that would use either *eq?* or *equal?*	No, because we have not yet told you how.
How can you make *rember* remove the first a from (b c a)	By passing a and (b c a) as arguments to *rember*.
How can you make *rember* remove the first c from (b c a)	By passing c and (b c a) as arguments to *rember*.
How can you make *rember-f* use *equal?* instead of *eq?*	By passing *equal?* as an argument to *rember-f*.
What is (*rember-f test? a l*) where *test?* is =[1] *a* is 5 and *l* is (6 2 5 3)	(6 2 3).

[1] L: `(rember-f (function =) 5 '(6 2 5 3))`, but there is more.

What is (*rember-f test? a l*) where *test?* is *eq?* *a* is jelly and *l* is (jelly beans are good)	(beans are good).
And what is (*rember-f test? a l*) where *test?* is *equal?* *a* is (pop corn) and *l* is (lemonade (pop corn) and (cake))	(lemonade and (cake)).

Try to write *rember-f*

```
(define rember-f
  (lambda (test? a l)
    (cond
      ((null? l) (quote ()))
      (else (cond
              ((test? (car l) a)[1] (cdr l))
              (else (cons (car l)
                      (rember-f test? a
                        (cdr l)))))))))
```

This is good!

[1] L: (funcall test? (car l) a). Use funcall when invoking a function argument or a function that has not been defuned.

What about the short version?

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

How does (*rember-f test? a l*) act where *test?* is *eq?*

(*rember-f test? a l*) where *test?* is *eq?*, acts like *rember*.

And what about (*rember-f test? a l*) where *test?* is *equal?*

This is just *rember* with *eq?* replaced by *equal?*.

Now we have four functions that do almost the same thing.

Yes:
  rember with =
  rember with *equal?*
  rember with *eq?*
and
  rember-f.

And *rember-f* can behave like all the others.	Let's generate all versions with *rember-f*.
What kind of values can functions return?	Lists and atoms.
What about functions themselves?	Yes, but you probably did not know that yet.
Can you say what (**lambda** (*a l*) ...) is?	(**lambda** (*a l*) ...) is a function of two arguments, *a* and *l*.
Now what is (**lambda** (*a*) (**lambda** (*x*) (*eq? x a*)))	It is a function that, when passed an argument *a*, returns the function (**lambda** (*x*) (*eq? x a*)) where *a* is just that argument.
Is this called "Curry-ing?"	Thank you, Moses Schönfinkel (1889–1942).
It is not called "Schönfinkel-ing."	Thank you, Haskell B. Curry (1900–1982).
Using (**define** ...) give the preceding function a name.	(**define** *eq?-c*[1] (**lambda** (*a*) (**lambda** (*x*) (*eq? x a*)))) This is our choice.
What is (*eq?-c k*) where *k* is salad	Its value is a function that takes *x* as an argument and tests whether it is *eq?* to salad.

```
(define eq?-c
  (lambda (a)
    (lambda (x)
      (eq? x a))))
```

[1] L:
```
(defun eq?-c (a)
  (function
    (lambda (x)
      (eq x a))))
```

So let's give it a name using (**define** ...)

Okay.

```
(define¹ eq?-salad (eq?-c k))
```

where *k* is salad

[1] L: `(setq eq?-salad (eq?-c 'salad))`.
Use `setq` to define a function that can be `funcalled`.

What is (*eq?-salad* *y*)[1]
where *y* is salad

#t.

[1] L: `(funcall eq?-salad y)`, since `eq?-salad` has not been `defuned`.

And what is (*eq?-salad* *y*)
where *y* is tuna

#f.

Do we need to give a name to *eq?-salad*

No, we may just as well ask
((*eq?-c* *x*) *y*)[1]
where
x is salad
and
y is tuna.

[1] L: `(funcall (eq?-c x) y)`, since `(eq?-c x)` is a function that has not been `defuned`.

Now rewrite *rember-f* as a function of one argument *test?* that returns an argument like *rember* with *eq?* replaced by *test?*

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

is a good start.

Describe in your own words the result of
  (*rember-f test?*)
where
  test? is *eq?*

It is a function that takes two arguments, *a* and *l*. It compares the elements of the list with *a*, and the first one that is *eq?* to *a* is removed.

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?*.

What is (*rember-eq? a l*)
where *a* is tuna
and
  l is (tuna salad is good)

(salad is good).

Did we need to give the name *rember-eq?* to the function (*rember-f test?*)
where
  test? is *eq?*

No, we could have written
  ((*rember-f test?*) *a l*)
where
  test? is *eq?*
  a is tuna
and
  l is (tuna salad is good).

Now, complete the line
  (*cons* (*car l*) ...)
in *rember-f* so that *rember-f* works.

```
(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)))))))))
```

What is ((*rember-f eq?*) *a l*)
where *a* is tuna
and
  l is (shrimp salad and tuna salad)

(shrimp salad and salad).

What is ((*rember-f* *eq?*) *a* *l*)
where *a* is eq?
and
l is (equal? eq? eqan? eqlist? eqpair?)[1]

[1] Did you notice the difference between eq? and *eq?* Remember that the former is the atom and the latter is the function.

(equal? eqan? eqlist? eqpair?).

And now transform *insertL* to *insertL-f* the same way we have transformed *rember* into *rember-f*

```
(define insertL-f
  (lambda (test?)
    (lambda (new old l)
      (cond
        ((null? l) (quote ()))
        ((test? (car l) old)
         (cons new (cons old (cdr l))))
        (else (cons (car l)
                    ((insertL-f test?) new old
                     (cdr l)))))))))
```

And, just for the exercise, do it to *insertR*

```
(define insertR-f
  (lambda (test?)
    (lambda (new old l)
      (cond
        ((null? l) (quote ()))
        ((test? (car l) old)
         (cons old (cons new (cdr l))))
        (else (cons (car l)
                    ((insertR-f test?) new old
                     (cdr l)))))))))
```

Are *insertR* and *insertL* similar?

Only the middle piece is a bit different.

Can you write a function *insert-g* that would insert either at the left or at the right?

If you can, get yourself some coffee cake and relax! Otherwise, don't give up. You'll see it in a minute.

Which pieces differ?

The second lines differ from each other. In *insertL* it is:

((*eq?* (*car l*) *old*)
 (*cons new* (*cons old* (*cdr l*)))),

but in *insertR* it is:

((*eq?* (*car l*) *old*)
 (*cons old* (*cons new* (*cdr l*)))).

Put the difference in words!

We say:
"The two functions *cons old* and *new* in a different order onto the *cdr* of the list *l*."

So how can we get rid of the difference?

You probably guessed it: by passing in a function that expresses the appropriate *cons*ing.

Define a function *seqL* that
1. takes three arguments, and
2. *cons*es the first argument onto the result of *cons*ing the second argument onto the third argument.

```
(define seqL
  (lambda (new old l)
    (cons new (cons old l))))
```

What is:

```
(define seqR
  (lambda (new old l)
    (cons old (cons new l))))
```

A function that
1. takes three arguments, and
2. *cons*es the second argument onto the result of *cons*ing the first argument onto the third argument.

Do you know why we wrote these functions?

Because they express what the two differing lines in *insertL* and *insertR* express.

Try to write the function *insert-g* of one argument *seq*
which returns *insertL*
where *seq* is *seqL*
and
which returns *insertR*
where *seq* is *seqR*

```
(define insert-g
  (lambda (seq)
    (lambda (new old l)
      (cond
        ((null? l) (quote ()))
        ((eq? (car l) old)
         (seq new old (cdr l)))
        (else (cons (car l)
                ((insert-g seq) new old
                 (cdr l)))))))
```

Now define *insertL* with *insert-g*

```
(define insertL (insert-g seqL))
```

And *insertR*.

```
(define insertR (insert-g seqR))
```

Is there something unusual about these two definitions?

Yes. Earlier we would probably have written
(**define** *insertL* (*insert-g seq*))
where
seq is *seqL*
and
(**define** *insertR* (*insert-g seq*))
where
seq is *seqR*.
But, using "where" is unnecessary when you pass functions as arguments.

Is it necessary to give names to *seqL* and *seqR*

Not really. We could have passed their definitions instead.

Define *insertL* again with *insert-g*
Do not pass in *seqL* this time.

```
(define insertL
  (insert-g
    (lambda (new old l)
      (cons new (cons old l)))))
```

Is this better?

Yes, because you do not need to remember as many names. You can
(*rember func-name* "your-mind")
where *func-name* is *seqL*.

Do you remember the definition of *subst*

Here is one.

```
(define subst
  (lambda (new old l)
    (cond
      ((null? l) (quote ()))
      ((eq? (car l) old)
       (cons new (cdr l)))
      (else (cons (car l)
              (subst new old (cdr l)))))))
```

Does this look familiar?

Yes, it looks like *insertL* or *insertR*. Just the answer of the second **cond**-line is different.

Define a function like *seqL* or *seqR* for *subst*

What do you think about this?

```
(define seqS
  (lambda (new old l)
    (cons new l)))
```

And now define *subst* using *insert-g*

```
(define subst (insert-g seqS))
```

And what do you think *yyy* is

```
(define yyy
  (lambda (a l)
    ((insert-g seqrem) #f a l)))
```

where

```
(define seqrem
  (lambda (new old l)
    l))
```

Surprise! It is our old friend *rember*

Hint: Step through the evaluation of
(*yyy a l*)
where
a is sausage
and
l is (pizza with sausage and bacon).
What role does #f play?

What you have just seen is the power of abstraction.

The Ninth Commandment

Abstract common patterns with a new function.

Have we seen similar functions before?

Yes, we have even seen functions with similar lines.

Do you remember *value* from chapter 6?

```
(define value
  (lambda (nexp)
    (cond
      ((atom? nexp) nexp)
      ((eq? (operator nexp)
            (quote +))
       (+ (value (1st-sub-exp nexp))
          (value (2nd-sub-exp nexp))))
      ((eq? (operator nexp)
            (quote ×))
       (× (value (1st-sub-exp nexp))
          (value (2nd-sub-exp nexp))))
      (else
        (↑ (value (1st-sub-exp nexp))
           (value (2nd-sub-exp nexp)))))))
```

Do you see the similarities?

The last three answers are the same except for the +, ×, and ↑.

Can you write the function *atom-to-function* which:
1. Takes one argument *x* and
2. returns the function +
   if (*eq? x* (**quote** +))
   returns the function ×
   if (*eq? x* (**quote** ×)) and
   returns the function ↑
   otherwise?

```
(define atom-to-function
  (lambda (x)
    (cond
      ((eq? x (quote +)) + )
      ((eq? x (quote ×)) ×)
      (else ↑))))
```

What is (*atom-to-function* (*operator nexp*)) where
nexp is (+ 5 3)

The function +, not the atom +.

Can you use *atom-to-function* to rewrite *value* with only two **cond**-lines?

Of course.

```
(define value
  (lambda (nexp)
    (cond
      ((atom? nexp) nexp)
      (else
        ((atom-to-function
           (operator nexp))
         (value (1st-sub-exp nexp))
         (value (2nd-sub-exp nexp)))))))
```

Is this quite a bit shorter than the first version?

Yes, but that's okay. We haven't changed its meaning.

Time for an apple?

One a day keeps the doctor away.

Here is *multirember* again.

```
(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)))))))
```

Write *multirember-f*

No problem.

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

What is ((*multirember-f test?*) *a lat*) where
test? is *eq?*
a is tuna
and
lat is (shrimp salad tuna salad and tuna)

(shrimp salad salad and).

Wasn't that easy?	Yes.
Define *multirember-eq?* using *multirember-f*	(**define** *multirember-eq?* (*multirember-f test?*)) where *test?* is *eq?*.
Do we really need to tell *multirember-f* about tuna	As *multirember-f* visits all the elements in *lat*, it always looks for tuna.
Does *test?* change as *multirember-f* goes through *lat*	No, *test?* always stands for *eq?*, just as *a* always stands for tuna.
Can we combine *a* and *test?*	Well, *test?* could be a function of just one argument and could compare that argument to tuna.
How would it do that?	The new *test?* takes one argument and compares it to tuna.

Here is one way to write this function.
 	
(**define** *eq?-tuna*
  (*eq?-c k*))
 	
where *k* is tuna
Can you think of a different way of writing this function?
	Yes, and here is a different way:
	
	(**define** *eq?-tuna*
	  (*eq?-c* (**quote** tuna)))

Have you ever seen definitions that contain atoms?	Yes, 0, (**quote** ×), (**quote** +), and many more.

Perhaps we should now write *multiremberT* which is similar to *multirember-f* Instead of taking *test?* and returning a function, *multiremberT* takes a function like *eq?-tuna* and a lat and then does its work.

This is not really difficult.

```
(define multiremberT
  (lambda (test? lat)
    (cond
      ((null? lat) (quote ()))
      ((test? (car lat))
       (multiremberT test? (cdr lat)))
      (else (cons (car lat)
                  (multiremberT test?
                    (cdr lat)))))))
```

What is (*multiremberT test? lat*)
where
test? is *eq?-tuna*
and
lat is (shrimp salad tuna salad and tuna)

(shrimp salad salad and).

Is this easy?

It's not bad.

How about this?

```
(define multirember&co
  (lambda (a lat col)
    (cond
      ((null? lat)
       (col (quote ()) (quote ())))
      ((eq? (car lat) a)
       (multirember&co a
         (cdr lat)
         (lambda (newlat seen)
           (col newlat
             (cons (car lat) seen)))))
      (else
        (multirember&co a
          (cdr lat)
          (lambda (newlat seen)
            (col (cons (car lat) newlat)
              seen)))))))
```

Now that looks really complicated!

Here is something simpler:

```
(define a-friend
  (lambda (x y)
    (null? y)))
```

Yes, it is simpler. It is a function that takes two arguments and asks whether the second one is the empty list. It ignores its first argument.

What is the value of
  (*multirember&co a lat col*)
where
  a is tuna
  lat is (strawberries tuna and swordfish)
and
  col is *a-friend*

This is not simple.

So let's try a friendlier example. What is the value of (*multirember&co a lat col*)
where
  a is tuna
  lat is ()
and
  col is *a-friend*

#t, because *a-friend* is immediately used in the first answer on two empty lists, and *a-friend* makes sure that its second argument is empty.

And what is (*multirember&co a lat col*)
where
  a is tuna
  lat is (tuna)
and
  col is *a-friend*

multirember&co asks
  (*eq?* (*car lat*) (**quote** tuna))
where
  lat is (tuna).
Then it recurs on ().

What are the other arguments that *multirember&co* uses for the natural recursion?

The first one is clearly tuna. The third argument is a new function.

What is the name of the third argument?

col.

Do you know what *col* stands for?

The name *col* is short for "collector."
  A collector is sometimes called a "continuation."

Here is the new collector:

```
(define new-friend
  (lambda (newlat seen)
    (col newlat
      (cons (car lat) seen))))
```

where
  (*car lat*) is tuna
and
  col is *a-friend*
Can you write this definition differently?

Do you mean the new way where we put tuna into the definition?

```
(define new-friend
  (lambda (newlat seen)
    (col newlat
      (cons (quote tuna) seen))))
```

where
  col is *a-friend*.

Can we also replace *col* with *a-friend* in such definitions because *col* is to *a-friend* what (*car lat*) is to tuna

Yes, we can:

```
(define new-friend
  (lambda (newlat seen)
    (a-friend newlat
      (cons (quote tuna) seen))))
```

And now?

multirember&co finds out that (*null? lat*) is true, which means that it uses the collector on two empty lists.

Which collector is this?

It is *new-friend*.

How does *a-friend* differ from *new-friend*

new-friend uses *a-friend* on the empty list and the value of
  (*cons* (**quote** tuna) (**quote** ())).

And what does the old collector do with such arguments?

It answers #f, because its second argument is (tuna), which is not the empty list.

What is the value of
  (*multirember&co a lat a-friend*)
where *a* is tuna
and
  lat is (and tuna)

This time around *multirember&co* recurs with yet another friend.

```
(define latest-friend
  (lambda (newlat seen)
    (a-friend (cons (quote and) newlat)
      seen)))
```

And what is the value of this recursive use of *multirember&co*

#f, since (*a-friend ls1 ls2*)
where
ls1 is (and)
and
ls2 is (tuna)
is #f.

What does (*multirember&co a lat f*) do?

It looks at every atom of the *lat* to see whether it is *eq?* to a. Those atoms that are not are collected in one list *ls1*; the others for which the answer is true are collected in a second list *ls2*. Finally, it determines the value of (*f ls1 ls2*).

Final question: What is the value of
(*multirember&co* (**quote** tuna) *ls col*)
where
ls is (strawberries tuna and swordfish)
and
col is

```
(define last-friend
  (lambda (x y)
    (length x)))
```

3, because *ls* contains three things that are not tuna, and therefore *last-friend* is used on (strawberries and swordfish) and (tuna).

Yes!

It's a strange meal, but we have seen foreign foods before.

The Tenth Commandment

Build functions to collect more than one value at a time.

Here is an old friend.

```
(define multiinsertL
  (lambda (new old lat)
    (cond
      ((null? lat) (quote ()))
      ((eq? (car lat) old)
       (cons new
         (cons old
           (multiinsertL new old
             (cdr lat)))))
      (else (cons (car lat)
              (multiinsertL new old
                (cdr lat)))))))
```

Do you also remember *multiinsertR*

No problem.

```
(define multiinsertR
  (lambda (new old lat)
    (cond
      ((null? lat) (quote ()))
      ((eq? (car lat) old)
       (cons old
         (cons new
           (multiinsertR new old
             (cdr lat)))))
      (else (cons (car lat)
              (multiinsertR new old
                (cdr lat)))))))
```

Now try *multiinsertLR*
Hint: *multiinsertLR* inserts *new* to the left of *oldL* and to the right of *oldR* in *lat* if *oldL* are *oldR* are different.

This is a way of combining the two functions.

```
(define multiinsertLR
  (lambda (new oldL oldR lat)
    (cond
      ((null? lat) (quote ()))
      ((eq? (car lat) oldL)
       (cons new
         (cons oldL
           (multiinsertLR new oldL oldR
             (cdr lat)))))
      ((eq? (car lat) oldR)
       (cons oldR
         (cons new
           (multiinsertLR new oldL oldR
             (cdr lat)))))
      (else
        (cons (car lat)
          (multiinsertLR new oldL oldR
            (cdr lat)))))))
```

The function *multiinsertLR&co* is to *multiinsertLR* what *multirember&co* is to *multirember*

Does this mean that *multiinsertLR&co* takes one more argument than *multiinsertLR*?

Yes, and what kind of argument is it?

It is a collector function.

When *multiinsertLR&co* is done, it will use *col* on the new lat, on the number of *left* insertions, and the number of *right* insertions. Can you write an outline of *multiinsertLR&co*

Sure, it is just like *multiinsertLR*.

```
(define multiinsertLR&co
  (lambda (new oldL oldR lat col)
    (cond
      ((null? lat)
       (col (quote ()) 0 0))
      ((eq? (car lat) oldL)
       (multiinsertLR&co new oldL oldR
         (cdr lat)
         (lambda (newlat L R)
           ...)))
      ((eq? (car lat) oldR)
       (multiinsertLR&co new oldL oldR
         (cdr lat)
         (lambda (newlat L R)
           ...)))
      (else
        (multiinsertLR&co new oldL oldR
          (cdr lat)
          (lambda (newlat L R)
            ...))))))
```

Why is *col* used on (**quote** ()) 0 and 0 when (*null? lat*) is true?

The empty *lat* contains neither *oldL* nor *oldR*. And this means that 0 occurrences of *oldL* and 0 occurrences of *oldR* are found and that *multiinsertLR* will return () when *lat* is empty.

So what is the value of

```
(multiinsertLR&co
  (quote cranberries)
  (quote fish)
  (quote chips)
  (quote ())
  col)
```

It is the value of (*col* (**quote** ()) 0 0), which we cannot determine because we don't know what *col* is.

Is it true that *multiinsertLR&co* will use the new collector on three arguments when (*car lat*) is equal to neither *oldL* nor *oldR*

Yes, the first is the lat that *multiinsertLR* would have produced for (*cdr lat*), *oldL*, and *oldR*. The second and third are the number of insertions that occurred to the left and right of *oldL* and *oldR*, respectively.

Is it true that *multiinsertLR&co* then uses the function *col* on (*cons* (*car lat*) *newlat*) because it copies the list unless an *oldL* or an *oldR* appears?

Yes, it is true, so we know what the new collector for the last case is:

```
(lambda (newlat L R)
  (col (cons (car lat) newlat) L R)).
```

Why are *col*'s second and third arguments just *L* and *R*

If (*car lat*) is neither *oldL* nor *oldR*, we do not need to insert any new elements. So, *L* and *R* are the correct results for both (*cdr lat*) and all of *lat*.

Here is what we have so far. And we have even thrown in an extra collector:

```
(define multiinsertLR&co
  (lambda (new oldL oldR lat col)
    (cond
      ((null? lat)
       (col (quote ()) 0 0))
      ((eq? (car lat) oldL)
       (multiinsertLR&co new oldL oldR
         (cdr lat)
         (lambda (newlat L R)
           (col (cons new
                  (cons oldL newlat))
             (add1 L) R))))
      ((eq? (car lat) oldR)
       (multiinsertLR&co new oldL oldR
         (cdr lat)
         (lambda (newlat L R)
         ...)))
      (else
        (multiinsertLR&co new oldL oldR
          (cdr lat)
          (lambda (newlat L R)
            (col (cons (car lat) newlat)
              L R))))))))
```

Can you fill in the dots?

The incomplete collector is similar to the extra collector. Instead of adding one to *L*, it adds one to *R*, and instead of *cons*ing *new* onto *cons*ing *oldL* onto *newlat*, it *cons*es *oldR* onto the result of *cons*ing *new* onto *newlat*.

So can you fill in the dots?

Yes, the final collector is

```
(lambda (newlat L R)
  (col (cons oldR (cons new newlat))
    L (add1 R))).
```

What is the value of
(*multiinsertLR&co new oldL oldR lat col*)
where
new is salty
oldL is fish
oldR is chips
and
lat is (chips and fish or fish and chips)

It is the value of (*col newlat* 2 2)
where
newlat is (chips salty and salty fish or salty fish and chips salty).

Is this healthy?

Looks like lots of salt. Perhaps dessert is sweeter.

Do you remember what *-functions are?

Yes, all *-functions work on lists that are either
— empty,
— an atom *cons*ed onto a list, or
— a list *cons*ed onto a list.

Now write the function *evens-only** which removes all odd numbers from a list of nested lists. Here is *even?*

```
(define even?
  (lambda (n)
    (= (× (÷ n 2) 2) n)))
```

Now that we have practiced this way of writing functions, *evens-only** is just an exercise:

```
(define evens-only*
  (lambda (l)
    (cond
      ((null? l) (quote ()))
      ((atom? (car l))
       (cond
         ((even? (car l))
          (cons (car l)
            (evens-only* (cdr l))))
         (else (evens-only* (cdr l)))))
      (else (cons (evens-only* (car l))
              (evens-only* (cdr l)))))))
```

What is the value of (*evens-only* l*)
where
l is ((9 1 2 8) 3 10 ((9 9) 7 6) 2)

((2 8) 10 (() 6) 2).

What is the sum of the odd numbers in *l* where
l is ((9 1 2 8) 3 10 ((9 9) 7 6) 2)

$9+1+3+9+9+7=38$.

What is the product of the even numbers in *l* where
l is ((9 1 2 8) 3 10 ((9 9) 7 6) 2)

$2 \times 8 \times 10 \times 6 \times 2 = 1920$.

Can you write the function *evens-only*&co* It builds a nested list of even numbers by removing the odd ones from its argument and simultaneously multiplies the even numbers and sums up the odd numbers that occur in its argument.

This is full of stars!

Here is an outline. Can you explain what (*evens-only*&co* (*car l*) ...) accomplishes?

```
(define evens-only*&co
  (lambda (l col)
    (cond
      ((null? l)
       (col (quote ()) 1 0))
      ((atom? (car l))
       (cond
         ((even? (car l))
          (evens-only*&co (cdr l)
            (lambda (newl p s)
              (col (cons (car l) newl)
                (× (car l) p) s))))
         (else (evens-only*&co (cdr l)
                 (lambda (newl p s)
                   (col newl
                     p (+ (car l) s)))))))
      (else (evens-only*&co (car l)
              ...)))))
```

It visits every number in the *car* of *l* and collects the list without odd numbers, the product of the even numbers, and the sum of the odd numbers.

What does the function *evens-only*&co* do after visiting all the numbers in (*car l*)

It uses the collector, which we haven't defined yet.

And what does the collector do?

It uses *evens-only*&co* to visit the *cdr* of *l* and to collect the list that is like (*cdr l*), without the odd numbers of course, as well as the product of the even numbers and the sum of the odd numbers.

Does this mean the unknown collector looks roughly like this:

```
(lambda (al ap as)
  (evens-only*&co (cdr l)
    ...))
```

Yes.

And when (*evens-only*&co* (*cdr l*) ...) is done with its job, what happens then?

The yet-to-be-determined collector is used, just as before.

What does the collector for
(*evens-only*&co* (*cdr l*) ...)
do?

It *cons*es together the results for the lists in the *car* and the *cdr* and multiplies and adds the respective products and sums. Then it passes these values to the old collector:

```
(lambda (al ap as)
  (evens-only*&co (cdr l)
    (lambda (dl dp ds)
      (col (cons al dl)
        (× ap dp)
        (+ as ds))))).
```

Does this all make sense now?

Perfect.

What is the value of
(*evens-only*&co l the-last-friend*)
where
l is ((9 1 2 8) 3 10 ((9 9) 7 6) 2) and
the-last-friend is defined as follows:

```
(define the-last-friend
  (lambda (newl product sum)
    (cons sum
      (cons product
        newl))))
```

(38 1920 (2 8) 10 (() 6) 2).

Whew! Is your brain twisted up now?	Go eat a pretzel; don't forget the mustard.