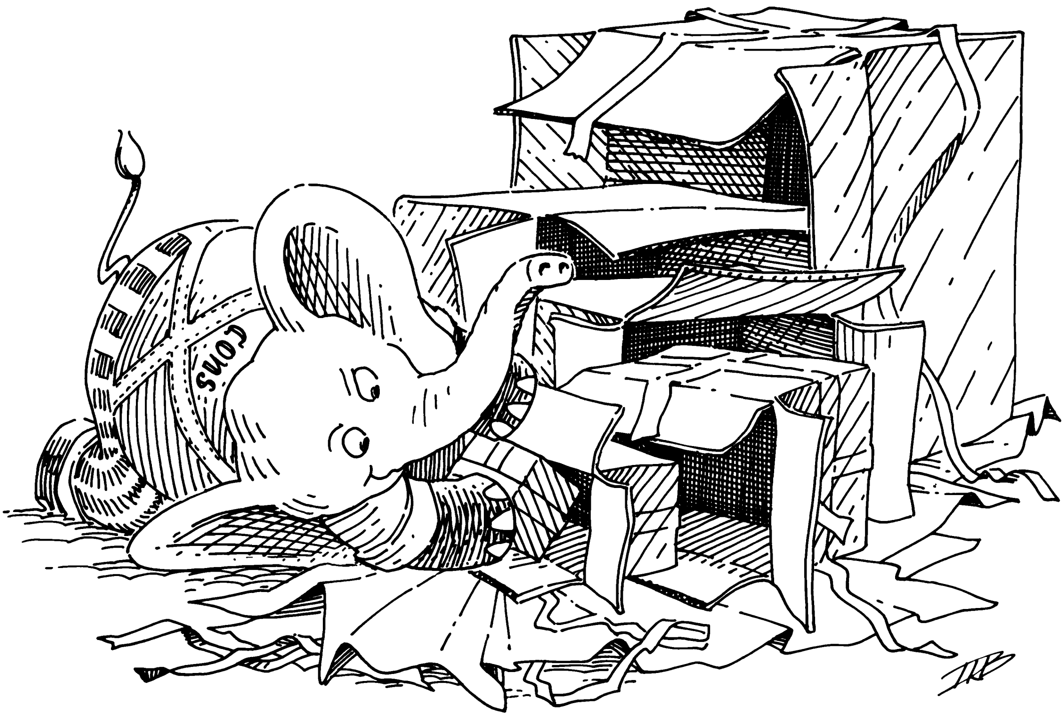
3.
Cons the Magnificent



What is (*rember a lat*) where *a* is mint and *lat* is (lamb chops and mint jelly)	(lamb chops and jelly) "Rember" stands for *rem*ove a mem*ber*.
(*rember a lat*) where *a* is mint and *lat* is (lamb chops and mint flavored mint jelly)	(lamb chops and flavored mint jelly).
(*rember a lat*) where *a* is toast and *lat* is (bacon lettuce and tomato)	(bacon lettuce and tomato).
(*rember a lat*) where *a* is cup and *lat* is (coffee cup tea cup and hick cup)	(coffee tea cup and hick cup).
What does (*rember a lat*) do?	It takes an atom and a lat as its arguments, and makes a new lat with the first occurrence of the atom in the old lat removed.
What steps should we use to do this?	First we will test (*null? lat*)—The First Commandment.
And if (*null? lat*) is true?	Return ().
What do we know if (*null? lat*) is not true?	We know that there must be at least one atom in the lat.
Is there any other question we should ask about the lat?	No. Either a lat is empty or it contains at least one atom.

What do we do if we know that the lat contains at least one atom?

We ask whether *a* is equal to (*car lat*).

How do we ask questions?

By using

```
(cond
  ( ________  ________ )
  ( ________  ________ )).
```

How do we ask if *a* is the same as (*car lat*)

(*eq?* (*car lat*) *a*).

What would be the value of (*rember a lat*) if *a* were the same as (*car lat*)

(*cdr lat*).

What do we do if *a* is not the same as (*car lat*)

We want to keep (*car lat*), but also find out if *a* is somewhere in the rest of the lat.

How do we remove the first occurrence of *a* in the rest of *lat*

(*rember a* (*cdr lat*)).

Is there any other question we should ask?

No.

Now, let's write down what we have so far:

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

What is the value of (*rember a lat*) where
a is bacon
and
lat is (bacon lettuce and tomato)

(lettuce and tomato).
Hint: Write down the function *rember* and its arguments, and refer to them as you go through the next sequence of questions.

Now, let's see if this function works. What is the first question?	(*null? lat*).
What do we do now?	Move to the next line and ask the next question.
else	Yes.
What next?	Ask the next question.
(*eq?* (*car lat*) a)	Yes, so the value is (*cdr lat*). In this case, it is the list (lettuce and tomato).
Is this the correct value?	Yes, because it is the original list without the atom bacon.
But did we really use a good example?	Who knows? But the proof of the pudding is in the eating, so let's try another example.
What does *rember* do?	It takes an atom and a lat as its arguments, and makes a new lat with the first occurrence of the atom in the old lat removed.
What do we do now?	We compare each atom of the lat with the atom *a*, and if the comparison fails we build a list that begins with the atom we just compared.
What is the value of (*rember a lat*) where *a* is and and *lat* is (bacon lettuce and tomato)	(bacon lettuce tomato).

Let us see if our function *rember* works. What is the first question asked by *rember*	(*null? lat*).
What do we do now?	Move to the next line, and ask the next question.
else	Okay, so ask the next question.
(*eq?* (*car lat*) a)	No, so move to the next line.
What is the meaning of (**else** (*rember a* (*cdr lat*)))	**else** asks if **else** is true—as it always is—and the rest of the line says to recur with *a* and (*cdr lat*), where *a* is and and (*cdr lat*) is (lettuce and tomato).
(*null? lat*)	No, so move to the next line.
else	Sure.
(*eq?* (*car lat*) *a*)	No, so move to the next line.
What is the meaning of (*rember a* (*cdr lat*))	Recur where *a* is and and (*cdr lat*) is (and tomato).
(*null? lat*)	No, so move to the next line, and ask the next question.
else	Of course.

(*eq?* (*car lat*) *a*)

Yes.

So what is the result?

(*cdr lat*)—(tomato).

Is this correct?

No, since (tomato) is not the list (bacon lettuce and tomato) with just *a*—and—removed.

What did we do wrong?

We dropped and, but we also lost all the atoms preceding and.

How can we keep from losing the atoms bacon and lettuce

We use Cons the Magnificent. Remember *cons*, from chapter 1?

The Second Commandment

Use *cons* to build lists.

Let's see what happens when we use *cons*

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

What is the value of (*rember a lat*) where *a* is and
and
lat is (bacon lettuce and tomato)

(bacon lettuce tomato).
Hint: Make a copy of this function with *cons* and the arguments *a* and *lat* so you can refer to it for the following questions.

What is the first question?	(*null? lat*).
What do we do now?	Ask the next question.
else	Yes, of course.
(*eq?* (*car lat*) *a*)	No, so move to the next line.
What is the meaning of (*cons* (*car lat*) (*rember a* (*cdr lat*))) where *a* is and and *lat* is (bacon lettuce and tomato)	It says to *cons* the *car* of *lat*—bacon—onto the value of (*rember a* (*cdr lat*)). But since we don't know the value of (*rember a* (*cdr lat*)) yet, we must find it before we can *cons* (*car lat*) onto it.
What is the meaning of (*rember a* (*cdr lat*))	This refers to the function with *lat* replaced by (*cdr lat*)—(lettuce and tomato).
(*null? lat*)	No, so move to the next line.
else	Yes, ask the next question.
(*eq?* (*car lat*) *a*)	No, so move to the next line.
What is the meaning of (*cons* (*car lat*) (*rember a* (*cdr lat*)))	It says to *cons* the *car* of *lat*—lettuce—onto the value of (*rember a* (*cdr lat*)). But since we don't know the value of (*rember a* (*cdr lat*)) yet, we must find it before we can *cons* (*car lat*) onto it.

What is the meaning of (*rember a* (*cdr lat*))	This refers to the function with *lat* replaced by (*cdr lat*), that is, (and tomato).
(*null? lat*)	No, so ask the next question.
else	Still.
(*eq?* (*car lat*) *a*)	Yes.
What is the value of the line ((*eq?* (*car lat*) *a*) (*cdr lat*))	(*cdr lat*)—(tomato).
Are we finished?	Certainly not! We know what (*rember a lat*) is when *lat* is (and tomato), but we don't yet know what it is when *lat* is (lettuce and tomato) or (bacon lettuce and tomato).
We now have a value for (*rember a* (*cdr lat*)) where *a* is and and (*cdr lat*) is (and tomato) This value is (tomato) What next?	Recall that we wanted to *cons* lettuce onto the value of (*rember a* (*cdr lat*)) where *a* was and and (*cdr lat*) was (and tomato). Now that we have this value, which is (tomato), we can *cons* lettuce onto it.
What is the result when we *cons* lettuce onto (tomato)	(lettuce tomato).
What does (lettuce tomato) represent?	It represents the value of (*cons* (*car lat*) (*rember a* (*cdr lat*))), when *lat* is (lettuce and tomato) and (*rember a* (*cdr lat*)) is (tomato).

Are we finished yet?

Not quite. So far we know what (*rember a lat*) is when
lat is (lettuce and tomato),
but we don't yet know what it is when
lat is (bacon lettuce and tomato).

We now have a value for (*rember a* (*cdr lat*)) where *a* is and
and
(*cdr lat*) is (lettuce and tomato)
This value is (lettuce tomato)
This is not the final value, so what must we do again?

Recall that, at one time, we wanted to *cons* bacon onto the value of (*rember a* (*cdr lat*)), where
a was and
and
(*cdr lat*) was (lettuce and tomato).
Now that we have this value, which is
(lettuce tomato),
we can *cons* bacon onto it.

What is the result when we *cons* bacon onto
(lettuce tomato)

(bacon lettuce tomato).

What does (bacon lettuce tomato) represent?†

† Lunch?

It represents the value of
(*cons* (*car lat*)
(*rember a* (*cdr lat*))),
when
lat is (bacon lettuce and tomato)
and
(*rember a* (*cdr lat*)) is (lettuce tomato).

Are we finished yet?

Yes.

Can you put in your own words how we determined the final value
(bacon lettuce tomato)

In our words:
"The function *rember* checked each atom of the lat, one at a time, to see if it was the same as the atom and. If the *car* was not the same as the atom, we saved it to be *cons*ed to the final value later. When *rember* found the atom and, it dropped it, and *cons*ed the previous atoms back onto the rest of the lat."

Can you rewrite *rember* so that it reflects the above description?

Yes, we can simplify it.

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

Do you think this is simpler?

Functions like *rember* can always be simplified in this manner.

So why don't we simplify right away?

Because then a function's structure does not coincide with its argument's structure.

Let's see if the new *rember* is the same as the old one. What is the value of the application
(*rember a lat*)
where *a* is and
and
lat is (bacon lettuce and tomato)

(bacon lettuce tomato).
Hint: Write down the function *rember* and its arguments and refer to them as you go through the next sequence of questions.

(*null? lat*)

No.

(*eq?* (*car lat*) *a*)

No.

else

Yes, so the value is
(*cons* (*car lat*)
(*rember a* (*cdr lat*))).

What is the meaning of
(*cons* (*car lat*)
(*rember a* (*cdr lat*)))

This says to refer to the function *rember* but with the argument *lat* replaced by (*cdr lat*), and that after we arrive at a value for (*rember a* (*cdr lat*)) we must *cons* (*car lat*)—bacon—onto it.

(*null? lat*)	No.
(*eq?* (*car lat*) *a*)	No.
else	Yes, so the value is (*cons* (*car lat*) (*rember a* (*cdr lat*))).
What is the meaning of (*cons* (*car lat*) (*rember a* (*cdr lat*)))	This says we recur using the function *rember*, with the argument *lat* replaced by (*cdr lat*), and that after we arrive at a value for (*rember a* (*cdr lat*)), we must *cons* (*car lat*)—lettuce—onto it.
(*null? lat*)	No.
(*eq?* (*car lat*) *a*)	Yes.
What is the value of the line ((*eq?* (*car lat*) *a*) (*cdr lat*))	It is (*cdr lat*)—(tomato).
Now what?	Now *cons* (*car lat*)—lettuce—onto (tomato).
Now what?	Now *cons* (*car lat*)--bacon—onto (lettuce tomato).
Now that we have completed *rember* try this example: (*rember a lat*) where *a* is sauce and *lat* is (soy sauce and tomato sauce)	(*rember a lat*) is (soy and tomato sauce).

What is (*firsts* *l*) where *l* is ((apple peach pumpkin) (plum pear cherry) (grape raisin pea) (bean carrot eggplant))	(apple plum grape bean).
What is (*firsts* *l*) where *l* is ((a b) (c d) (e f))	(a c e).
What is (*firsts* *l*) where *l* is ()	().
What is (*firsts* *l*) where *l* is ((five plums) (four) (eleven green oranges))	(five four eleven).
What is (*firsts* *l*) where *l* is (((five plums) four) (eleven green oranges) ((no) more))	((five plums) eleven (no)).
In your own words, what does (*firsts* *l*) do?	We tried the following: "The function *firsts* takes one argument, a list, which is either a null list or contains only non-empty lists. It builds another list composed of the first S-expression of each internal list."

See if you can write the function *firsts*
Remember the Commandments!

This much is easy:

```
(define firsts
  (lambda (l)
    (cond
      ((null? l) ...)
      (else (cons ... (firsts (cdr l)))))))
```

Why
(**define** *firsts*
(**lambda** (*l*)
...))

Because we always state the function name, (**lambda**, and the argument(s) of the function.

Why (**cond** ...)

Because we need to ask questions about the actual arguments.

Why ((*null? l*) ...)

The First Commandment.

Why (**else**

Because we only have two questions to ask about the list *l*: either it is the null list, or it contains at least one non-empty list.

Why (**else**

See above. And because the last question is always **else**.

Why (*cons*

Because we are building a list—The Second Commandment.

Why (*firsts* (*cdr l*))

Because we can only look at one S-expression at a time. To look at the rest, we must recur.

Why)))

Because these are the matching parentheses for (**cond**, (**lambda**, and (**define**, and they always appear at the end of a function definition.

Keeping in mind the definition of (*firsts l*) what is a typical element of the value of (*firsts l*) where *l* is ((a b) (c d) (e f))	a.
What is another typical element?	c, or even e.
Consider the function *seconds* What would be a typical element of the value of (*seconds l*) where *l* is ((a b) (c d) (e f))	b, d, or f.
How do we describe a typical element for (*firsts l*)	As the *car* of an element of *l*—(*car* (*car l*)). See chapter 1.
When we find a typical element of (*firsts l*) what do we do with it?	*cons* it onto the recursion—(*firsts* (*cdr l*)).

The Third Commandment

When building a list, describe the first typical element, and then *cons* it onto the natural recursion.

With The Third Commandment, we can now fill in more of the function *firsts* What does the last line look like now?	$(\textbf{else}\ (\textit{cons}\ \underbrace{(\textit{car}\ (\textit{car}\ l))}_{\text{typical element}}\ \underbrace{(\textit{firsts}\ (\textit{cdr}\ l))}_{\text{natural recursion}}\ )).$

What does (*firsts l*) do

```
(define firsts
  (lambda (l)
    (cond
      ((null? l) ...)
      (else (cons (car (car l))
                  (firsts (cdr l)))))))
```

where *l* is ((a b) (c d) (e f))

Nothing yet. We are still missing one important ingredient in our recipe. The first line ((*null? l*) ...) needs a value for the case where *l* is the null list. We can, however, proceed without it for now.

(*null? l*) where *l* is ((a b) (c d) (e f))

No, so move to the next line.

What is the meaning of
(*cons* (*car* (*car l*))
(*firsts* (*cdr l*)))

It saves (*car* (*car l*)) to *cons* onto (*firsts* (*cdr l*)). To find (*firsts* (*cdr l*)), we refer to the function with the new argument (*cdr l*).

(*null? l*) where *l* is ((c d) (e f))

No, so move to the next line.

What is the meaning of
(*cons* (*car* (*car l*))
(*firsts* (*cdr l*)))

Save (*car* (*car l*)), and recur with (*firsts* (*cdr l*)).

(*null? l*) where *l* is ((e f))

No, so move to the next line.

What is the meaning of
(*cons* (*car* (*car l*))
(*firsts* (*cdr l*)))

Save (*car* (*car l*)), and recur with (*firsts* (*cdr l*)).

(*null? l*)

Yes.

Now, what is the value of the line
((*null? l*) ...)

There is no value; something is missing.

What do we need to *cons* atoms onto?	A list. Remember The Law of Cons.
For the purpose of *cons*ing, what value can we give when (*null? l*) is true?	Since the final value must be a list, we cannot use #t or #f. Let's try (**quote** ()).
With () as a value, we now have three *cons* steps to go back and pick up. We need to: I. either 1. *cons* e onto () 2. *cons* c onto the value of 1 3. *cons* a onto the value of 2 II. or 1. *cons* a onto the value of 2 2. *cons* c onto the value of 3 3. *cons* e onto () III. or *cons* a onto the *cons* of c onto the *cons* of e onto () In any case, what is the value of (*firsts l*)	(a c e).
With which of the three alternatives do you feel most comfortable?	Correct! Now you should use that one.
What is (*insertR new old lat*) where *new* is topping *old* is fudge and *lat* is (ice cream with fudge for dessert)	(ice cream with fudge topping for dessert).
(*insertR new old lat*) where *new* is jalapeño *old* is and and *lat* is (tacos tamales and salsa)	(tacos tamales and jalapeño salsa).

(*insertR new old lat*) where *new* is e *old* is d and *lat* is (a b c d f g d h)	(a b c d e f g d h).
In your own words, what does (*insertR new old lat*) do?	In our words: "It takes three arguments: the atoms *new* and *old*, and a lat. The function *insertR* builds a lat with *new* inserted to the right of the first occurrence of *old*."
See if you can write the first three lines of the function *insertR*	(**define** *insertR* (**lambda** (*new old lat*) (**cond** ...)))
Which argument changes when we recur with *insertR*	*lat*, because we can only look at one of its atoms at a time.
How many questions can we ask about the lat?	Two. A lat is either the null list or a non-empty list of atoms.
Which questions do we ask?	First, we ask (*null? lat*). Second, we ask **else**, because **else** is always the last question.
What do we know if (*null? lat*) is not true?	We know that *lat* has at least one element.
Which questions do we ask about the first element?	First, we ask (*eq?* (*car lat*) *old*). Then we ask **else**, because there are no other interesting cases.

Now see if you can write the whole function *insertR*

```
(define insertR
  (lambda (new old lat)
    (cond
      ( ________  ________ )
      (else
        (cond
          ( ________  ________ )
          ( ________  ________ ))))))
```

Here is our first attempt.

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

What is the value of the application
  (*insertR new old lat*)
that we just determined
where
  new is topping
  old is fudge
and
  lat is (ice cream with fudge for dessert)

(ice cream with for dessert).

So far this is the same as *rember*
What do we do in *insertR* when
(*eq?* (*car lat*) *old*) is true?

When (*car lat*) is the same as *old*, we want to insert *new* to the right.

How is this done?

Let's try *cons*ing *new* onto (*cdr lat*).

Now we have

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

Yes.

So what is (*insertR new old lat*) now where
new is topping
old is fudge
and
lat is (ice cream with fudge for dessert)

(ice cream with topping for dessert).

Is this the list we wanted?

No, we have only replaced fudge with topping.

What still needs to be done?

Somehow we need to include the atom that is the same as *old* before the atom *new*.

How can we include *old* before *new*

Try *consing* *old* onto (*cons new* (*cdr lat*)).

Now let's write the rest of the function *insertR*

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

When *new* is topping, *old* is fudge, and *lat* is (ice cream with fudge for dessert), the value of the application, (*insertR new old lat*), is
(ice cream with fudge topping for dessert).
If you got this right, have one.

Now try *insertL*
Hint: *insertL* inserts the atom *new* to the left of the first occurrence of the atom *old* in *lat*

This much is easy, right?

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

Did you think of a different way to do it?

For example,

```
((eq? (car lat) old)
 (cons new (cons old (cdr lat))))
```

could have been

```
((eq? (car lat) old)
 (cons new lat))
```

since (*cons old* (*cdr lat*)) where *old* is *eq?* to (*car lat*) is the same as *lat*.

Now try *subst*
Hint: (*subst new old lat*) replaces the first occurrence of *old* in the *lat* with *new*
For example,
where
new is topping
old is fudge
and
lat is (ice cream with fudge for dessert)
the value is
(ice cream with topping for dessert)
Now you have the idea.

Obviously,

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

This is the same as one of our incorrect attempts at *insertR*.

Go cons a piece of cake onto your mouth.

Now try *subst2*
Hint:
(*subst2 new o1 o2 lat*)
replaces either the first occurrence of *o1* or the first occurrence of *o2* by *new*
For example,
where
new is vanilla
o1 is chocolate
o2 is banana
and
lat is (banana ice cream with chocolate topping)
the value is
(vanilla ice cream with chocolate topping)

```
(define subst2
  (lambda (new o1 o2 lat)
    (cond
      ((null? lat) (quote ()))
      (else (cond
              ((eq? (car lat) o1)
               (cons new (cdr lat)))
              ((eq? (car lat) o2)
               (cons new (cdr lat)))
              (else (cons (car lat)
                      (subst2 new o1 o2
                        (cdr lat)))))))))
```

Did you think of a better way?

Replace the two *eq?* lines about the (*car lat*) by
((**or** (*eq?* (*car lat*) *o1*) (*eq?* (*car lat*) *o2*))
(*cons new* (*cdr lat*))).

If you got the last function, go and repeat the cake-consing.

Do you recall what *rember* does?

The function *rember* looks at each atom of a lat to see if it is the same as the atom *a*. If it is not, *rember* saves the atom and proceeds. When it finds the first occurrence of *a*, it stops and gives the value (*cdr lat*), or the rest of the lat, so that the value returned is the original lat, with only that occurrence of *a* removed.

Write the function *multirember* which gives as its final value the lat with all occurrences of *a* removed.

```
(define multirember
  (lambda (a lat)
    (cond
      ( ________  ________ )
      (else
        (cond
          ( ________  ________ )
          ( ________  ________ ))))))
```

Hint: What do we want as the value when (*eq?* (*car lat*) *a*) is true?
Consider the example
where *a* is cup
and
lat is (coffee cup tea cup and hick cup)

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

After the first occurrence of *a*, we now recur with (*multirember a* (*cdr lat*)), in order to remove the other occurrences.

The value of the application is (coffee tea and hick).

Can you see how *multirember* works?

Possibly not, so we will go through the steps necessary to arrive at the value (coffee tea and hick).

(*null? lat*)

No, so move to the next line.

---

else

Yes.

(*eq?* (*car lat*) *a*)

No, so move to the next line.

What is the meaning of (*cons* (*car lat*) (*multirember a* (*cdr lat*)))

Save (*car lat*)—coffee—to be *cons*ed onto the value of (*multirember a* (*cdr lat*)) later. Now determine (*multirember a* (*cdr lat*)).

(*null? lat*)

No, so move to the next line.

else	Naturally.
(*eq?* (*car lat*) *a*)	Yes, so forget (*car lat*), and determine (*multirember a* (*cdr lat*)).
(*null? lat*)	No, so move to the next line.
else	Yes!
(*eq?* (*car lat*) *a*)	No, so move to the next line.
What is the meaning of (*cons* (*car lat*) (*multirember a* (*cdr lat*)))	Save (*car lat*)—tea—to be *cons*ed onto the value of (*multirember a* (*cdr lat*)) later. Now determine (*multirember a* (*cdr lat*)).
(*null? lat*)	No, so move to the next line.
else	Okay, move on.
(*eq?* (*car lat*) *a*)	Yes, so forget (*car lat*), and determine (*multirember a* (*cdr lat*)).
(*null? lat*)	No, so move to the next line.
(*eq?* (*car lat*) *a*)	No, so move to the next line.
What is the meaning of (*cons* (*car lat*) (*multirember a* (*cdr lat*)))	Save (*car lat*)—and—to be *cons*ed onto the value of (*multirember a* (*cdr lat*)) later. Now determine (*multirember a* (*cdr lat*)).

(*null? lat*)	No, so move to the next line.
(*eq?* (*car lat*) *a*)	No, so move to the next line.
What is the meaning of (*cons* (*car lat*) (*multirember a* (*cdr lat*)))	Save (*car lat*)—hick—to be *cons*ed onto the value of (*multirember a* (*cdr lat*)) later. Now determine (*multirember a* (*cdr lat*)).
(*null? lat*)	No, so move to the next line.
(*eq?* (*car lat*) *a*)	Yes, so forget (*car lat*), and determine (*multirember a* (*cdr lat*)).
(*null? lat*)	Yes, so the value is ().
Are we finished?	No, we still have several *cons*es to pick up.
What do we do next?	We *cons* the most recent (*car lat*) we have—hick—onto ().
What do we do next?	We *cons* and onto (hick).
What do we do next?	We *cons* tea onto (and hick).
What do we do next?	We *cons* coffee onto (tea and hick).
Are we finished now?	Yes.

Now write the function *multiinsertR*

```
(define multiinsertR
  (lambda (new old lat)
    (cond
      ( ________  ________ )
      (else
        (cond
          ( ________  ________ )
          ( ________  ________ ))))))
```

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

It would also be correct to use *old* in place of (*car lat*) because we know that (*eq?* (*car lat*) *old*).

Is this function defined correctly?

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

Not quite. To find out why, go through (*multiinsertL new old lat*) where *new* is fried, *old* is fish and *lat* is (chips and fish or fish and fried).

Was the terminal condition ever reached?

No, because we never get past the first occurrence of *old*.

Now, try to write the function *multiinsertL* again:

```
(define multiinsertL
  (lambda (new old lat)
    (cond
      ( ________  ________ )
      (else
        (cond
          ( ________  ________ )
          ( ________  ________ ))))))
```

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

The Fourth Commandment

(preliminary)

Always change at least one argument while recurring. It must be changed to be closer to termination. The changing argument must be tested in the termination condition: when using *cdr*, test termination with *null?*.

Now write the function *multisubst*

```
(define multisubst
  (lambda (new old lat)
    (cond
      ( ________  ________ )
      (else
        (cond
          ( ________  ________ )
          ( ________  ________ ))))))
```

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