14. Let There Be Names

Do you remember the function *leftmost*

Is it the function that extracts the leftmost atom from a list of S-expressions?

Yes, and here is the definition:

```
(define leftmost
  (lambda (l)
    (cond
      ((atom? (car l)) (car l))
      (else (leftmost (car l))))))
```

Okay.

What is the value of (*leftmost l*)
where
  l is (((a) b) (c d))

a, of course.

And what is the value of (*leftmost l*)
where
  l is (((a) ()) () (e))

It's still a.

How about this: (*leftmost l*)
where
  l is (((() a) ()))

It should still be a, but there is actually no answer.

Why is it not a

In chapter 5, we said that the function *leftmost* finds the leftmost atom in a non-empty list of S-expressions that does not contain the empty list.

Didn't we just determine (*leftmost l*) where the list *l* contained an empty list?

Yes, we did: *l* was (((a) ()) () (e)).

Shouldn't we be able to define a version of *leftmost* that does not restrict the shape of its argument?

We definitely should.

Which atom can occur in the leftmost position of a list of S-expressions?

Every atom may occur as the leftmost atom of a list of S-expressions, including #f.

Then how do we indicate that some argument for the unrestricted version of *leftmost* does not contain an atom?

In that case, *leftmost* must return a non-atom.

What should it return?

It could return a list.

Does it matter which list it returns?

No, but () is the simplest list.

Is this a good start?

```
(define leftmost
  (lambda (l)
    (cond
      ((null? l) (quote ()))
      ((atom? (car l)) (car l))
      (else ...
            (leftmost (car l))
            ...))))
```

Yes. By adding the first line, *leftmost* now looks like a real *-function.

How do we determine the value of
  (*leftmost* *l*)
where
  l is (((() a) ()))

Using the new definition of *leftmost*, we quickly determine that *l* isn't empty and doesn't contain an atom in the *car* position. So we recur with (*leftmost* *l*)
where
  l is ((() a) ()).

What happens when we recur?

We ask the same questions, we get the same answers, and we recur with (*leftmost* *l*)
where
  l is (() a).

And then?

Then we recur with (*leftmost* *l*)
where
  l is ().

What is the value of (*leftmost* (**quote** ()))

It is (), which means that we haven't found a yet.

What do we need to do?

We also need to recur with the *cdr* of the list, if we can't find an atom in the *car*.

How do we determine whether
 (*leftmost* (*car l*))
found an atom?

We ask (*atom?* (*leftmost* (*car l*))), because *leftmost* only returns an atom if its argument contains one.

And when (*atom?* (*leftmost* (*car l*))) is true?

Then we know what the leftmost atom is.

And how do we say it?

Easy: (*leftmost* (*car l*)).

But if (*atom?* (*leftmost* (*car l*))) is false?

Then we continue to look for an atom in the *cdr* of *l*.

Define *leftmost*

```
(define leftmost
  (lambda (l)
    (cond
      ((null? l) (quote ()))
      ((atom? (car l)) (car l))
      (else (cond
              ((atom? (leftmost (car l)))
               (leftmost (car l)))
              (else (leftmost (cdr l))))))))
```

(*leftmost l*)
where
 l is (((a) b) (c d))

a.

(*leftmost l*)
where
 l is (((a) ()) () (e))

a.

(*leftmost l*)
where
l is (((() a) ()))

a, as it should be.

Does the repetition of (*leftmost* (*car l*)) seem wrong?

Yes, we have to read the same expression twice to understand the function. It is almost like passing along the same argument to a recursive function.

Isn't it?

We could try to use (**letrec** ...) to get rid of such unwanted repetitions.

Right, but does (**letrec** ...) give names to arbitrary things?

Well, we have only used it for functions, but shouldn't it work for other expressions too?

We choose to use (**let** ...) instead. It is like (**letrec** ...) but it is used for exactly what we need to do now.

To give a name to a repeated expression?

Yes, (**let** ...) also has a naming part and a value part, just like (**letrec** ...) We use the latter to name the values of expressions.

Okay, so far it looks like (**letrec** ...). Do we use the value part to determine the result with the help of these names?

As we said, it looks like (**letrec** ...) but it gives names to the values of expressions.

How can we use it to name expressions?

We name the values of expressions, but ignoring this detail, we can sketch the new definition:

```
(define leftmost
  (lambda (l)
    (cond
      ((null? l) (quote ()))
      ((atom? (car l)) (car l))
      (else ...))))
```

Can you complete this definition?

How about?

```
...
(let[1] ((a (leftmost (car l))))
  (cond
    ((atom? a) a)
    (else (leftmost (cdr l)))))
...
```

[1] Like (**and** ...), (**let** ...) is an abbreviation:
(**let** ((x_1 α_1) ... (x_n α_n)) β ...)
= ((**lambda** (x_1 ... x_n) β ...) α_1 ... α_n)

Isn't this much easier to read?

Yes, it is.

What is the value of (*rember1** *a* *l*)
where *a* is salad
and
l is ((Swedish rye)
(French (mustard salad turkey))
salad)

((Swedish rye)
(French (mustard turkey))
salad).

(*rember1** *a* *l*)
where *a* is meat
and
l is ((pasta meat)
pasta
(noodles meat sauce)
meat tomatoes)

((pasta)
pasta
(noodles meat sauce)
meat tomatoes).

Take a close look at *rember1**

```
(define rember1*
  (lambda (a l)
    (cond
      ((null? l) (quote ()))
      ((atom? (car l))
       (cond
         ((eq? (car l) a) (cdr l))
         (else (cons (car l)
                     (rember1* a (cdr l))))))
      (else
        (cond
          ((eqlist?
             (rember1* a (car l))
             (car l))
           (cons (car l)
             (rember1* a (cdr l))))
          (else (cons (rember1* a (car l))
                      (cdr l))))))))
```

Fix *rember1** using The Twelfth Commandment.

It even has the same expressions underlined.

```
(define rember1*
  (lambda (a l)
    (letrec
      ((R (lambda (l)
            (cond
              ((null? l) (quote ()))
              ((atom? (car l))
               (cond
                 ((eq? (car l) a) (cdr l))
                 (else (cons (car l)
                             (R (cdr l))))))
              (else
                (cond
                  ((eqlist?
                     (R (car l))
                     (car l))
                   (cons (car l)
                     (R (cdr l))))
                  (else (cons (R (car l))
                              (cdr l)))))))))
      (R l))))
```

What does (*rember1* a l*) do?

It removes the leftmost occurrence of *a* in *l*.

Can you describe how *rember1** works?

Here is our description:
"The function *rember1** goes through the list of S-expressions. When there is a list in the *car*, it attempts to remove *a* from the *car*. If the *car* remains the same, *a* is not in the *car*, and *rember1** must continue. When *rember1** finds an atom in the list, and the atom is equal to *a*, it is removed."

Why do we use *eqlist?* instead of *eq* to compare (*R* (*car l*)) with (*car l*)

Because *eq?* compares atoms, and *eqlist?* compares lists.

Is *rember1** related to *leftmost*

Yes, the two functions use the same trick: *leftmost* attempts to find an atom in (*car l*) when (*car l*) is a list. If it doesn't find one, it continues its search; otherwise, that atom is the result.

Do the underlined instances of (*R* (*car l*)) seem wrong?

They certainly must seem wrong to anyone who reads the definition. We should remove them.

Here is a sketch of a definition of *rember1** that uses (**let** ...)

```
(define rember1*
  (lambda (a l)
    (letrec
      ((R (lambda (l)
            (cond
              ((null? l) (quote ()))
              ((atom? (car l))
               (cond
                 ((eq? (car l) a) (cdr l))
                 (else (cons (car l)
                             (R (cdr l))))))
              (else ...)))))
      (R l))))
```

Here is the rest of the minor function *R*

```
...
(let ((av (R (car l))))
  (cond
    ((eqlist? (car l) av)
     (cons (car l) (R (cdr l))))
    (else (cons av (cdr l)))))
...
```

That's precisely what we had in mind.

Good.

The Fifteenth Commandment

(*preliminary version*)

Use (let ...) to name the values of repeated expressions.

Let's do some more **let**ting.

Good idea.

What should we try?

Any ideas?

We could try it on *depth**

What is *depth**?

Oh, that's right. We haven't told you yet. Here it is.

```
(define depth*
  (lambda (l)
    (cond
      ((null? l) 1)
      ((atom? (car l))
       (depth* (cdr l)))
      (else
        (cond
          ((> (depth* (cdr l))
              (add1 (depth* (car l))))
           (depth* (cdr l)))
          (else
            (add1 (depth* (car l)))))))))
```

It looks like a normal *-function.

Let's try an example. Determine the value of
(*depth* l*)
where
  l is ((pickled) peppers (peppers pickled))

2.

Here is another one: (*depth** *l*)
where
l is
```
(margarine
    ((bitter butter)
     (makes)
     (batter (bitter)))
    butter)
```

4.

And here is a truly good example: (*depth** *l*)
where
l is (c (b (a b) a) a)

Still no problem: 3
But it is missing food.

Now let's go back and do what we actually wanted to do.

Yes, we should try to use (**let** ...).

What should we use (**let** ...) for?

We determine the value of (*depth** (*car l*)) and the value of (*depth** (*cdr l*)) at two different places.

Do you mean that these repeated uses of *depth** look like good opportunities for naming the values of expressions?

Yes, they do.

Let's see what the new function looks like.

How about this one?

```
(define depth*
  (lambda (l)
    (let ((a (add1 (depth* (car l))))
          (d (depth* (cdr l))))
      (cond
        ((null? l) 1)
        ((atom? (car l)) d)
        (else (cond
                ((> d a) d)
                (else a)))))))
```

Should we try some examples?

It should be correct. Using (**let** ...) is straightforward.

Let's try it anyway. What is the value of
(depth l)*
where

```
l is (()
      ((bitter butter)
       (makes)
       (batter (bitter)))
      butter)
```

It should be 4. We did something like this before.

Let's do this slowly.

First, we ask *(null? l)*, which is false.

Not quite. We need to name the values of *(add1 (depth* (car l)))* and *(depth* (cdr l))* first!

That's true, but what is there to it? The names are *a* and *d*.

But first we need the values!

That's true. The first expression for which we need to determine the value is
(add1 (depth (car l)))*
where

```
l is (()
      ((bitter butter)
       (makes)
       (batter (bitter)))
      butter).
```

How do we do that?

We use *depth** and check whether the argument is *null?*, which is true now.

Not so fast: don't forget to name the values!

Whew: we need to determine the value of *(add1 (depth* (car l)))* where *l* is ().

And what is the value?

There is no value: see The Law of Car.

Can you explain in your words what happened?

Here are our words:
"A (**let** ...) first determines the values of the named expressions. Then it associates a name with each value and determines the value of the expression in the value part. Since the value of the named expression in our example depends on the value of (*car l*) before we know whether or not *l* is empty, this *depth** is incorrect."

Here is *depth** again.

```
(define depth*
  (lambda (l)
    (cond
      ((null? l) 1)
      ((atom? (car l))
       (depth* (cdr l)))
      (else
        (cond
          ((> (depth* (cdr l))
              (add1 (depth* (car l))))
           (depth* (cdr l)))
          (else
            (add1 (depth* (car l)))))))))
```

Use (**let** ...) for the last **cond**-line.

```
(define depth*
  (lambda (l)
    (cond
      ((null? l) 1)
      ((atom? (car l))
       (depth* (cdr l)))
      (else
        (let ((a (add1 (depth* (car l))))
              (d (depth* (cdr l))))
          (cond
            ((> d a) d)
            (else a))))))))
```

Why does this version of *depth** work?

If both (*null? l*) and (*atom?* (*car l*)) are false, (*car l*) and (*cdr l*) are both lists, and it is okay to use *depth** on both lists.

Would we have needed to determine (*depth** (*car l*)) and (*depth** (*cdr l*)) twice if we hadn't introduced names for their values?

We would have had to determine the value of one of the expressions twice if we hadn't used (**let** ...), depending on whether the depth of the *car* is greater than the depth of the *cdr*.

Would we have needed to determine (*leftmost* (*car l*)) twice if we hadn't introduced a name for its value?

Yes.

Would we have needed to determine (*rember1** (*car l*)) twice if we hadn't introduced a name for its value?

Yes.

How should we use (**let** ...) in ***depth**** if we want to use it right after finding out whether or not *l* is empty?

After we know that (*null? l*) is false, we only know that (*cdr l*) is a list; (*car l*) might still be an atom. And because of that, we should introduce a name for only the value of (***depth**** (*cdr l*)) and not for (***depth**** (*car l*)).

Let's do it! Here is an outline.

```
(define depth*
  (lambda (l)
    (cond
      ((null? l) 1)
      (else ...))))
```

Fill in the dots.

```
...
(let ((d (depth* (cdr l))))
  (cond
    ((atom? (car l)) d)
    (else
      (cond
        ((> d (add1 (depth* (car l)))) d)
        (else (add1 (depth* (car l))))))))
...
```

And when can we use (**let** ...) for the repeated expression (***add1*** (***depth**** (*car l*)))

```
(define depth*
  (lambda (l)
    (cond
      ((null? l) 1)
      (else ...))))
```

Fill in the dots again.

When we know that (*car l*) is not an atom:

```
...
(let ((d (depth* (cdr l))))
  (cond
    ((atom? (car l)) d)
    (else
      (let ((a (add1 (depth* (car l)))))
        (cond
          ((> d a) d)
          (else a))))))
...
```

Would we have needed to determine (***depth**** (*cdr l*)) twice if we hadn't introduced a name for its value?

No. If the first element of *l* is an atom, (***depth**** (*cdr l*)) is evaluated only once.

If it doesn't help to name the value of (***depth**** (*cdr l*)) we should check whether the new version of ***depth**** is easier to read.

Not really. The three nested **conds** hide what kinds of data the function sees.

So which version of *depth** is our favorite version?

```
(define depth*
  (lambda (l)
    (cond
      ((null? l) 1)
      ((atom? (car l))
       (depth* (cdr l)))
      (else
        (let ((a (add1 (depth* (car l))))
              (d (depth* (cdr l))))
          (cond
            ((> d a) d)
            (else a))))))))
```

The Fifteenth Commandment

(*revised version*)

Use (let ...) to name the values of repeated expressions in a function definition if they may be evaluated twice for one and the same use of the function.

This definition of *depth** looks quite short.

And it does the right thing in the right way.

It does, but this is actually unimportant.

Why?

Because we just wanted to practice **let**ting things be the way they are supposed to be.

Oh, yes. And we sure did.

Can we make *depth** more enjoyable?

Can we?

We can. How do you like this variation?

```
(define depth*
  (lambda (l)
    (cond
      ((null? l) 1)
      ((atom? (car l))
       (depth* (cdr l)))
      (else
        (let ((a (add1 (depth* (car l))))
              (d (depth* (cdr l))))
          (if (> d a) d a))))))
```

This looks even simpler, but what does (**if** ...) do?

The same as (**cond** ...)
Better, (**if** ...) asks only one question and provides two answers: if the question is true, it selects the first answer; otherwise, it selects the second answer.

That's clever. We should have known about this before.[1]

[1] Like (**and** ...), (**if** ...) can be abbreviated:
(**if** α β γ) = (**cond** (α β) (**else** γ))

There is a time and place for everything.

Back to *depth**.

One more thing. What is a good name for

```
(lambda (n m)
  (if (> n m) n m))
```

max,
because the function selects the larger of two numbers.

Here is how to use *max* to simplify *depth**

```
(define depth*
  (lambda (l)
    (cond
      ((null? l) 1)
      ((atom? (car l))
       (depth* (cdr l)))
      (else
        (let ((a (add1 (depth* (car l))))
              (d (depth* (cdr l))))
          (max a d))))))
```

Can we rewrite it without (**let** ((*a* ...)) ...)

Yes, no problem.

```
(define depth*
  (lambda (l)
    (cond
      ((null? l) 1)
      ((atom? (car l))
       (depth* (cdr l)))
      (else (max
              (add1 (depth* (car l)))
              (depth* (cdr l)))))))
```

Here is another chance to practice **let**ting: do it for the protected version of *scramble* from chapter 12:

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

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

How do you like *scramble* now?

It's perfect now.

Go have a bacon, lettuce, and tomato sandwich. And don't forget to let the lettuce dry.

Try it with mustard or mayonnaise.

Did that sandwich strengthen you?

We hope so.

Do you recall *leftmost*

Sure, we talked about it at the beginning of this chapter.

```
(define leftmost
  (lambda (l)
    (cond
      ((null? l) (quote ()))
      ((atom? (car l)) (car l))
      (else
        (let ((a (leftmost (car l))))
          (cond
            ((atom? a) a)
            (else (leftmost (cdr l)))))))))
```

What is (*leftmost* *l*)
where
  l is (((a)) b (c))

It is a.

And how do we determine this?	We have done this before.
So how do we do it?	We quickly determine that *l* isn't empty and doesn't contain an atom in the *car* position. So we recur with (*leftmost l*) where *l* is ((a)).
What do we do next?	We quickly determine that *l* isn't empty and doesn't contain an atom in the *car* position. So we recur with (*leftmost l*) where *l* is (a).
And now?	Now (*car l*) is a, so we are done.
Are we really done?	Well, we have the value for (*leftmost l*) where *l* is (a).
What do we do with this value?	We name it *a* and check whether it is an atom. Since it is an atom, we are done.
Are we really, really done?	Still not quite, but we have the value for (*leftmost l*) where *l* is ((a)).
And what do we do with this value?	We name it *a* again and check whether it is an atom. Since it is an atom, we are done.
So, are we done now?	No. We need to name a one more time, check that it is an atom one more time, and then we're completely done.

Have we been here before?

Yes, we have. When we discussed ***intersectall***, we also discovered that we really had the final answer long before we could say so.

And what did we do then?

We used (**letcc** ...).

Here is a new definition of *leftmost*

```
(define leftmost
  (lambda (l)
    (letcc skip
      (lm l skip))))
```

```
(define lm
  (lambda (l out)
    (cond
      ((null? l) (quote ()))
      ((atom? (car l)) (out (car l)))
      (else (let ()[1]
              (lm (car l) out)
              (lm (cdr l) out))))))
```

[1] L: **progn** also works.
S: **begin** also works.

Wow!

Did you notice the unusual (**let** ...)

Yes, the (**let** ...) contains two expressions in the value part.

What are they?

The first one is
  (*lm* (*car* *l*) *out*).
The one after that is
  (*lm* (*cdr* *l*) *out*).

And what do you think it means to have two expressions in the value part of a (**let** ...)

Here are our thoughts:
"When a (**let** ...) has two expressions in its value part, we must first determine the value of the first expression. If it has one, we ignore it and determine the value of the second expression[1]."

[1] This is also true of (**letrec** ...) and (**letcc** ...).

What is (*leftmost l*)
where
l is (((a)) b (c))

It should be a.

And how do we determine this?

We will have to use the new definition of *leftmost*.

Does this mean we start with (**letcc** *skip* ...)

Yes, and as before we ignore it for a while. We just don't forget that we have a North Pole called *skip*.

So what do we do?

We determine the value of (*lm l out*)
where
out is *skip*, the needle of a compass.

Next?

We quickly determine that *l* isn't empty and doesn't contain an atom in the *car* position. So we recur with (*lm l out*)
where
l is ((a))
and
out is *skip*, the needle of a compass.
And we also must remember that we will need to determine the value of (*lm l out*)
where
l is (b (c))
and
out is *skip*.

What do we do next?

We quickly determine that *l* isn't empty and doesn't contain an atom in the *car* position. So we recur with (*lm l out*)
where
  l is (a)
and
  out is *skip*, the needle of a compass.
And we also must remember that we will need to determine the value of
  (*lm l out*)
where
  l is ()
and
  out is still *skip*.

What exactly are we remembering right now?

We will need to determine the values of
  (*lm l out*)
where
  l is ()
and
  out is *skip*, the needle of a compass
as well as (*lm l out*)
where
  l is (b (c))
and
  out is *skip*, the needle of a compass.

Don't we have an atom in *car* of *l* now?

We do. And that means we need to understand
  (*out* (*car l*))
where
  l is (a)
and
  out is *skip*, the needle of a compass.

What does that mean?

We need to forget all the things we remembered to do and resume our work with
  (**letcc** *skip a*)
where *a* is a.

Are we done?

Yes, we have found the final value, a, and nothing else is left to do.

Isn't this peaceful?

Yes, it is. We never need to ask again whether a is an atom.

True or false: *lm* is only useful in conjunction with *leftmost*

Yes, that's true. We shouldn't forget The Thirteenth Commandment when we use The Fourteenth.

Here is one way to hide *lm*

```
(define leftmost
  (letrec
    ((lm (lambda (l out)
           (cond
             ((null? l) (quote ()))
             ((atom? (car l))
              (out (car l)))
             (else
               (let ()
                 (lm (car l) out)
                 (lm (cdr l) out)))))))
    (lambda (l)
      (letcc skip
        (lm l skip)))))
```

Can you think of another?

In chapter 12 we usually moved the minor function out of a (**lambda** ...)'s value part, but we can also move it in:

```
(define leftmost
  (lambda (l)
    (letrec
      ((lm (lambda (l out)
             (cond
               ((null? l) (quote ()))
               ((atom? (car l))
                (out (car l)))
               (else
                 (let ()
                   (lm (car l) out)
                   (lm (cdr l) out)))))))
      (letcc skip
        (lm l skip)))))
```

Correct! Better yet: we can move the (**letrec** ...) into the value part of the (**letcc** ...)

```
(define leftmost
  (lambda (l)
    (letcc skip
      (letrec (...)
        (lm l skip)))))
```

Can you complete the definition?

```
...
(lm (lambda (l out)
      (cond
        ((null? l) (quote ()))
        ((atom? (car l))
         (out (car l)))
        (else (let ()
                (lm (car l) out)
                (lm (cdr l) out))))))
...
```

This suggests that we should also use The Twelfth Commandment.

Why?

The second argument of *lm* is always going to refer to *skip*.

So?

When an argument stays the same and when we have a name for it in the surroundings of the function definition, we can drop it.

Rename *out* to *skip*

```
(define leftmost
  (lambda (l)
    (letcc skip
      (letrec (...)
        (lm l skip)))))
```

Yes, all names are equal.

```
...
(lm (lambda (l skip)
      (cond
        ((null? l) (quote ()))
        ((atom? (car l))
         (skip (car l)))
        (else
          (let ()
            (lm (car l) skip)
            (lm (cdr l) skip))))))
...
```

Can we now drop *skip* as an argument to *lm*

It is always the same argument, and the name *skip* is defined in the surroundings of the (**letrec** ...) so that everything works:

```
(define leftmost
  (lambda (l)
    (letcc skip
      (letrec
        ((lm (lambda (l)
               (cond
                 ((null? l) (quote ()))
                 ((atom? (car l))
                  (skip (car l)))
                 (else
                   (let ()
                     (lm (car l))
                     (lm (cdr l))))))))
        (lm l)))))
```

Can you explain how the new *leftmost* works?

Our explanation is:
"The function *leftmost* sets up a North Pole in *skip* and then determines the value of (*lm l*). The function *lm* looks at every atom in *l* from left to right until it finds an atom and then uses *skip* to return this atom abruptly and promptly."

(This would be a good time to count Duane's elephants.)

Didn't we say that *leftmost* and *rember1** are related?

Yes, we did.

Is *rember1** also a function that finds the final result yet checks many times that it did?

No, in that regard *rember1** is quite different. Every time it finds that the *car* of a list is a list, it works through the *car* and checks right afterwards with *eqlist?* whether anything changed.

Does *rember1** know when it failed to accomplish anything?

It does: every time it encounters the empty list, it failed to find the atom that is supposed to be removed.

Can we help *rember1** by using a compass needle when it finds the empty list?

With the help of a North Pole and a compass needle, we could abruptly and promptly signal that the list in the *car* of a list did not contain the interesting atom.

Here is a sketch of the function *rm* which takes advantage of this idea:

```
(define rm
  (lambda (a l oh)
    (cond
      ((null? l) (oh (quote no)))
      ((atom? (car l))
       (if (eq? (car l) a)
           (cdr l)
           (cons (car l)
             (rm a (cdr l) oh))))
      (else ...
            (letcc oh
              (rm a (car l) oh))
            ...))))
```

What does the function do when it encounters a list in (*car l*)

It sets up a North Pole and then recurs on the *car* also using the corresponding compass needle. When it finds an empty list, it uses the needle to get back to a place where it should explore the *cdr* of a list.

What kind of value does
  (**letcc** *oh*
    (*rm a* (*car l*) *oh*))
yield when (*car l*) does not contain *a*

The atom no.

And what kind of value do we get when the *car* of *l* contains *a*

A list with the first occurrence of *a* removed.

Then what do we need to check next?

We need to ask whether or not this value is an atom:

```
(atom?
  (letcc oh
    (rm a (car l) oh))).
```

And then?

If it is an atom, *rm* must try to remove an occurrence of *a* in (*cdr l*).

How do we try to remove the leftmost occurrence of *a* in (*cdr l*)

Easy: with (*rm a* (*cdr l*) *oh*).

Is this the only thing we have to do?

No, we must not forget to add on the unaltered (*car l*) when we succeed. We can do this with a simple *cons*:

```
(cons (car l) (rm a (cdr l) oh)).
```

And if (**letcc** *oh* ...)'s value is not an atom?

Then it is a list, which means that *rm* succeeded in removing the first occurrence of *a* from (*car l*).

How do we build the result in this case?

We *cons* the very value that

```
(letcc oh
  (rm a (car l) oh))
```

produced onto (*cdr l*), which does not change.

Which compass needle do we use to reconstruct this value?

We don't need one because we know *rm* will succeed in removing an atom.

Does this mean we can use (*rm a* (*car l*) 0)

Yes, any value will do, and 0 is a simple argument.

Let's do that!

Here is a better version of *rm*:

```
(define rm
  (lambda (a l oh)
    (cond
      ((null? l) (oh (quote no)))
      ((atom? (car l))
       (if (eq? (car l) a)
           (cdr l)
           (cons (car l)
             (rm a (cdr l) oh))))
      (else
        (if (atom?
              (letcc oh
                (rm a (car l) oh)))
            (cons (car l)
              (rm a (cdr l) oh))
            (cons (rm a (car l) 0)
              (cdr l)))))))
```

How can we use *rm*

We need to set up a North Pole first.

Why?

If the list does not contain the atom we want to remove, we must be able to say no.

What is the value of
  (**letcc** *Say* (*rm a l Say*))
where
  a is noodles
and
  l is ((food) more (food))

((food) more (food))
  because this list does not contain noodles.

And how do we determine this?

Since (*car l*) is a list, we set up a new North Pole, called *oh*, and recur with
  (*rm a* (*car l*) *oh*)
where
  a is noodles
and
  l is ((food) more (food)).

Which means?

After one more recursion, using the second **cond**-line, *rm* is used with noodles, the empty list, and the compass needle *oh*. Then it forgets the pending *cons* of food onto the result of the recursion and checks whether no is an atom.

And no is an atom ...

Yes, it is. So we recur with
  (*cons* (*car l*) (*rm a* (*cdr l*) *Say*))
where
  a is noodles
and
  l is ((food) more (food)).

How do we determine the value of
  (*rm a l Say*)
where
  a is noodles
and
  l is (more (food))

We recur with the list ((food)) and, if we get a result, we *cons* more onto it.

How do we determine the value of
(*rm a l Say*)
where
a is noodles
and
l is ((food))

We have done something like this before. We might as well jump to the conclusion.

Okay, so after we fail to remove an atom with
(*rm a l oh*)
where
l is (food)
we try
(*rm a l Say*)
where
a is noodles
and
l is ()

Yes, and now we use
(*Say* (**quote** no)).

And what happens?

We forget that we want to

1. *cons* more onto the result and
2. *cons* (food) onto the result of 1.

Instead we determine the value of
(**letcc** *Say* (**quote** no)).

So we failed.

Yes, we did.

But *rember1** would return the unaltered list, wouldn't it?

No problem:

```
(define rember1*
  (lambda (a l)
    (if (atom? (letcc oh (rm a l oh)))
        l
        (rm a l (quote ())))))
```

Why do we use (*rm a l* (**quote** ()))

Since *rm* will succeed, any value will do, and () is another simple argument.

Didn't we forget to name the values of some expression in *rember1**

```
(define rember1*
  (lambda (a l)
    (let ((new-l (letcc oh (rm a l oh))))
      (if (atom? new-l)
          l
          new-l))))
```

We can also use (**let** ...) in *rm*:

```
(define rm
  (lambda (a l oh)
    (cond
      ((null? l) (oh (quote no)))
      ((atom? (car l))
       (if (eq? (car l) a)
           (cdr l)
           (cons (car l)
             (rm a (cdr l) oh))))
      (else
        (let ((new-car
                (letcc oh
                  (rm a (car l) oh))))
          (if (atom? new-car)
              (cons (car l)
                (rm a (cdr l) oh))
              (cons new-car (cdr l))))))))
```

Do we need to make up a good example for *rember1**

We should, but aren't we late for dinner?

Do we need to protect *rm*

We should, but aren't we late for dinner?

Are you that hungry again?

Try some baba ghanouj followed by moussaka. If that sounds like too much eggplant, escape with a gyro.

Try this hot fudge sundae with coffee ice cream for dessert:

```
(define rember1*
  (lambda (a l)
    (try[1] oh (rm a l oh) l)))
```

It looks sweet, and it works, too.

[1] Like (**and** ...), (**try** ...) is an abbreviation:

```
(try x α β)
=
(letcc success
  (letcc x
    (success α))
  β)
```

The name *success* must not occur in α or β.

And don't forget the whipped cream and the cherry on top.

What do you mean?

We can even simplify *rm* with (**try** ...)

```
(define rm
  (lambda (a l oh)
    (cond
      ((null? l) (oh (quote no)))
      ((atom? (car l))
       (if (eq? (car l) a)
           (cdr l)
           (cons (car l)
             (rm a (cdr l) oh))))
      (else
        (try oh2
          (cons (rm a (car l) oh2)
            (cdr l))
          (cons (car l)
            (rm a (cdr l) oh)))))))
```

Does this version of *rember1** rely on no being an atom?

No.

Was it a fine dessert?

Yes, but now we are *oh* so very full.
