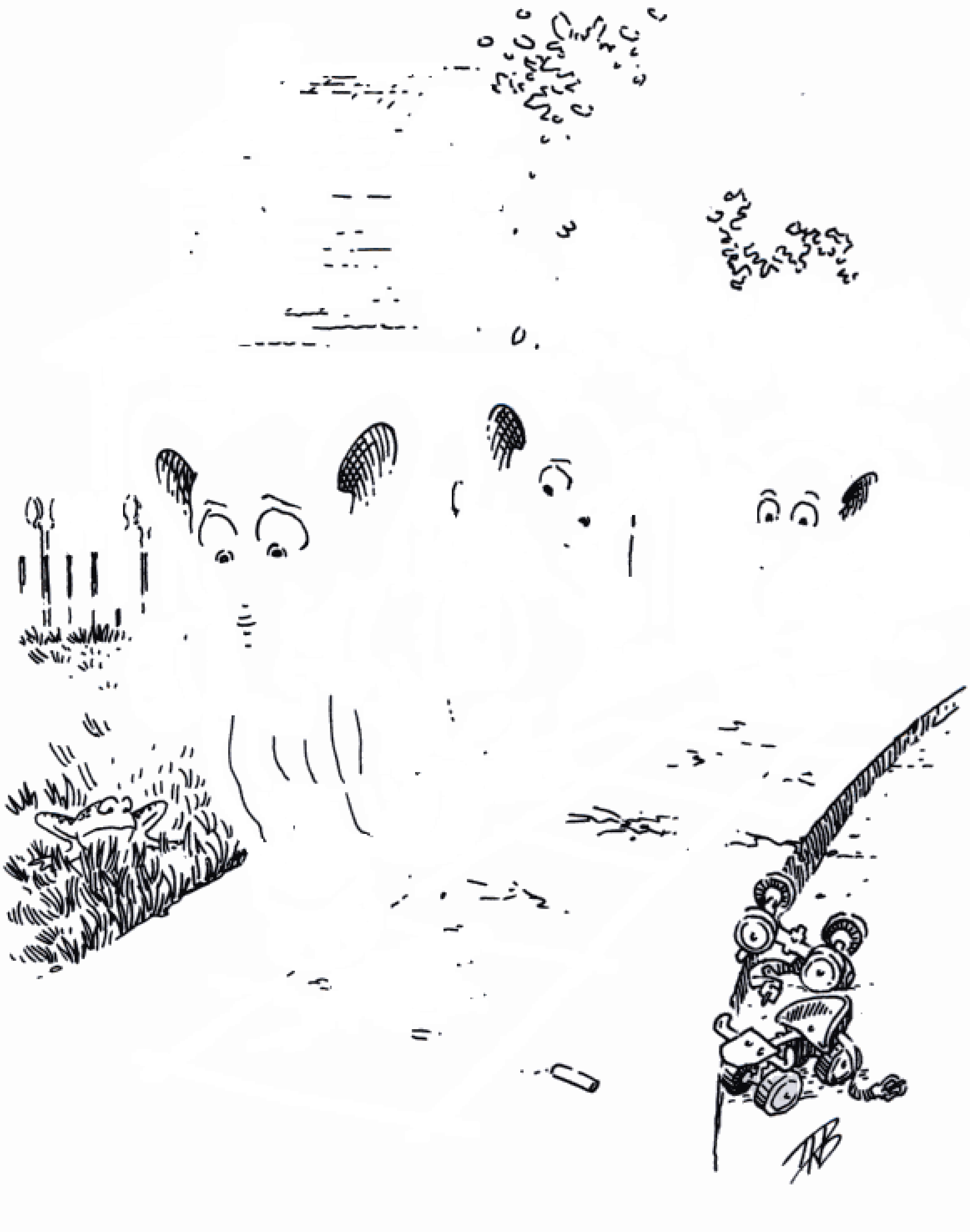
13.
Hop, Skip, and Jump



What is the value of (*intersect set1 set2*) where
set1 is (tomatoes and macaroni)
and
set2 is (macaroni and cheese)

(and macaroni).

Is *intersect* an old acquaintance?

Yes, we have known *intersect* for as long as we have known *union*.

Write *intersect*

Sure, here we go:

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

What would this definition look like if we hadn't forgotten The Twelfth Commandment?

```
(define intersect
  (lambda (set1 set2)
    (letrec
      ((I (lambda (set)
            (cond
              ((null? set) (quote ()))
              ((member? (car set) set2)
               (cons (car set)
                 (I (cdr set))))
              (else (I (cdr set)))))))
      (I set1))))
```

Do you also recall *intersectall*

Isn't that the function that *intersects* a list of sets?

```
(define intersectall
  (lambda (lset)
    (cond
      ((null? (cdr lset)) (car lset))
      (else (intersect (car lset)
              (intersectall (cdr lset)))))))
```

Why don't we ask (*null? lset*)

There is no need to ask this question because *The Little Schemer* assumes that the list of sets for *intersectall* is not empty.

How could we write a version of *intersectall* that makes no assumptions about the list of sets?

That's easy: We ask (*null? lset*) and then just use the two **cond**-lines from the earlier *intersectall*:

```
(define intersectall
  (lambda (lset)
    (cond
      ((null? lset) (quote ()))
      ((null? (cdr lset)) (car lset))
      (else (intersect (car lset)
              (intersectall
                (cdr lset)))))))
```

Are you sure that this definition is okay?

Yes? No?

Are there two base cases for just one argument?

No, the first question is just to make sure that *lset* is not empty before the function goes through the list of sets.

But once we know it isn't empty we never have to ask the question again.

Correct, because *intersectall* does not recur when it knows that the *cdr* of the list is empty.

What should we do then?

Ask the question once and use the old version of *intersectall* if the list is not empty.

And how would you do this?

Could we use another function?

Where do we place the function?

Should we use (**letrec** ...)?

Yes, the new version of *intersectall* could hide the old one inside a (**letrec** ...)

```
(define intersectall
  (lambda (lset)
    (letrec
      ((intersectall
         (lambda (lset)
           (cond
             ((null? (cdr lset))
              (car lset))
             (else (intersect (car lset)
                     (intersectall
                       (cdr lset))))))))
      (cond
        ((null? lset) (quote ()))
        (else (intersectall lset))))))
```

Could we have used *A* as the name of the function that we defined with (**letrec** ...)

Sure, *intersectall* is just a better name, though a bit long for these boxes.

```
(define intersectall
  (lambda (lset)
    (letrec
      ((A (lambda (lset)
            (cond
              ((null? (cdr lset))
               (car lset))
              (else (intersect (car lset)
                      (A (cdr lset))))))))
      (cond
        ((null? lset) (quote ()))
        (else (A lset))))))
```

Great! We are pleased to see that you are comfortable with (**letrec** ...).

One more time: we can use whatever name we want for such a minor function if nobody else relies on it.

Yes, because (**letrec** ...) hides definitions, and the names matter only inside of (**letrec** ...).

Is this similar to (**lambda** $(x\ y)$ M)

Yes, it is. The names x and y matter only inside of M, whatever M is. And in

(**letrec** $((x\ F)\ (y\ G))$ M)

the names x and y matter only inside of F, G, and M, whatever F, G, and M are.

Why do we ask (*null? lset*) before we use *A*	The question (*null? lset*) is not a part of *A*. Once we know that the list of sets is non-empty, we need to check for only the list containing a single set.
What is (*intersectall lset*) where *lset* is ((3 mangos and) (3 kiwis and) (3 hamburgers))	(3).
What is (*intersectall lset*) where *lset* is ((3 steaks and) (no food and) (three baked potatoes) (3 diet hamburgers))	().
What is (*intersectall lset*) where *lset* is ((3 mangoes and) () (3 diet hamburgers))	().
Why is this?	The intersection of (3 mangos and), (), and (3 diet hamburgers) is the empty set.
Why is this?	When there is an empty set in the list of sets, (*intersectall lset*) returns the empty set.
But this does not show how *intersectall* determines that the intersection is empty.	No, it doesn't. Instead, it keeps *intersect*ing the empty set with some set until the list of sets is exhausted.
Wouldn't it be better if *intersectall* didn't have to *intersect* each set with the empty set and if it could instead say "This is it: the result is () and that's all there is to it."	That would be an improvement. It could save us a lot of work if we need to determine the result of (*intersect lset*).

Well, there actually is a way to say such things.

There is?

Yes, we haven't shown you (**letcc** ...) yet.

Why haven't we mentioned it before?

Because we did not need it until now.

How would *intersectall* use (**letcc** ...)?

That's simple. Here we go:

```
(define intersectall
  (lambda (lset)
    (letcc[1] hop
      (letrec
        ((A (lambda (lset)
              (cond
                ((null? (car lset))
                 (hop (quote ()))[2])
                ((null? (cdr lset))
                 (car lset))
                (else
                  (intersect (car lset)
                    (A (cdr lset))))))))
        (cond
          ((null? lset) (quote ()))
          (else (A lset)))))))
```

[1] L: (catch 'hop ...)

[2] L: (throw 'hop (quote ()))

Alonzo Church (1903–1995) would have written:

```
(define intersectall
  (lambda (lset)
    (call-with-current-continuation[1]
      (lambda (hop)
        (letrec
          ((A (lambda (lset)
                (cond
                  ((null? (car lset))
                   (hop (quote ())))
                  ((null? (cdr lset))
                   (car lset))
                  (else
                    (intersect (car lset)
                      (A (cdr lset))))))))
          (cond
            ((null? lset) (quote ()))
            (else (A lset))))))))
```

[1] S: This is Scheme.

Doesn't this look easy?

We prefer the (**letcc** ...) version. It only has two new lines.

Yes, we added one line at the beginning and one **cond**-line inside the minor function *A*

It really looks like *three* lines.

A line in a (**cond** ...) is one line, even if we need more than one line to write it down. How do you like the first new line?

The first line with (**letcc** ... looks pretty mysterious.

But the first **cond**-line in *A* should be obvious: we ask one extra question
(*null?* (*car lset*))
and if it is true, *A* uses ***hop*** as if it were a function.

Correct: *A* will ***hop*** to the right place. How does this ***hop***ping work?

Now that is a different question. We could just try and see.

Why don't we try it with an example?

What is the value of (*intersectall lset*)
where
lset is ((3 mangoes and)
()
(3 diet hamburgers))

Yes, that is a good example. We want to know how things work when one of the sets is empty.

So how do we determine the answer for (*intersectall lset*)

Well, the first thing in *intersectall* is
(**letcc** *hop* ...
which looks mysterious.

Since we don't know what this line does, it is probably best to ignore it for the time being. What next?

We ask (*null? lset*), which in this case is not true.

And so we go on and ...

... determine the value of (*A lset*) where *lset* is the list of sets.

What is the next question?

(*null?* (*car lset*)).

Is this true?

No, (*car lset*) is the set
(3 mangos and).

Is this why we ask (*null?* (*cdr lset*))

Yes, and it is not true either.

---

else

Of course.

And now we recur?

Yes, we remember that (*car lset*) is (3 mangos and), and that we must *intersect* this set with the result of (*A* (*cdr lset*)).

How do we determine the value of (*A lset*) where
  lset is (()
      (3 diet hamburgers))

We ask (*null?* (*car lset*)).

Which is true.

And now we need to know the value of (*hop* (**quote** ())).

Recall that we wanted to *intersect* the set (3 mangos and) with the result of the natural recursion?

Yes.

And that there is
  (**letcc** *hop* ...
which we ignored earlier?

Yes, and (*hop* (**quote** ())) seems to have something to do with this line.

It does. The two lines are like a compass needle and the North Pole. The North Pole attracts one end of a compass needle, regardless of where in the world we are.

What does that mean?

It basically means:
 "Forget what we had remembered to do
  after leaving behind (**letcc** *hop*
  and before encountering (*hop M*) And then
  act as if we were to determine the value of
   (**letcc** *hop M*) whatever *M* is."

But how do we forget something?

Easy: we do not do it.

You mean we do not *intersect* the set (3 mangos and) with the result of the natural recursion?

Yes. And even better, when we need to determine the value of something that looks like
  (**letcc** *hop* (**quote** ()))
we actually know its answer.

The answer should be (), shouldn't it?

Yes, it is ()

That's what we wanted.

And it is what we got.

Amazing! We did not do any *intersect*ing at all.

That's right: we said *hop* and arrived at the right place with the result.

This is neat. Let's *hop* some more!

The Fourteenth Commandment

Use (letcc ...) to return values abruptly and promptly.

How about determining the value of
(*intersectall lset*)
where
  lset is ((3 steaks and)
      (no food and)
      (three baked potatoes)
      (3 diet hamburgers))

We ignore (**letcc** *hop*.

And then?

We determine the value of (*A lset*) because *lset* is not empty.

What do we ask next?	(*null?* (*car lset*)), which is false.
And next?	(*null?* (*cdr lset*)), which is false.

And next?	We remember to *intersect* (3 steaks and)
	with the result of the natural recursion:
	  (*A* (*cdr lset*))
	where
	  lset is ((3 steaks and)
	          (no food and)
	          (three baked potatoes)
	          (3 diet hamburgers)).

What happens now?	We ask the same questions as above and find
	out that we need to *intersect* the set
	(no food and) with the result of (*A lset*)
	where
	  lset is ((three baked potatoes)
	          (3 diet hamburgers)).

And afterward?	We ask the same questions as above and find out that we need to *intersect* the set (three baked potatoes) with the result of (*A lset*) where *lset* is ((3 diet hamburgers)).
And then?	We ask (*null?* (*car lset*)), which is false.
And then?	We ask (*null?* (*cdr lset*)), which is true.
And so we know what the value of (*A lset*) is where *lset* is ((3 diet hamburgers))	Yes, it is (3 diet hamburgers).

Are we done now?

No! With (3 diet hamburgers) as the value, we now have three *intersects* to go back and pick up.
We need to:
a. *intersect* (three baked potatotes) with (3 diet hamburgers);
b. *intersect* (no food and) with the value of a;
c. *intersect* (3 steaks and) with the value of b.
And then, at the end, we must not forget about (**letcc** *hop*.

Yes, so what is (*intersect set1 set2*)
where
set1 is (three baked potatoes)
and
set2 is (3 diet hamburgers)

().

So are we done?

No, we need to *intersect* this set with (no food and).

Yes, so what is (*intersect set1 set2*)
where
set1 is (no food and)
and
set2 is ()

().

So are we done now?

No, we still need to *intersect* this set with (3 steaks and).

But this is also empty.

Yes, it is.

So are we done?

Almost, but there is still the mysterious
(**letcc** *hop*
that we ignored initially.

Oh, yes. We must now determine the value of (**letcc** *hop* (**quote** ()))	That's correct. But what does this line do now that we did not use *hop*?
Nothing.	What do you mean, nothing?
When we need to determine the value of (**letcc** *hop* (**quote** ())) there is nothing left to do. We know the value.	You mean, it is () again?
Yes, it is () again.	That's simple.
Isn't it?	Except that we needed to *intersect* the empty set several times with a set before we could say that the result of *intersectall* was the empty set.
Is it a mistake of *intersectall*	Yes, and it is also a mistake of *intersect*.
In what sense?	We could have defined *intersect* so that it would not do anything when its second argument is the empty set.
Why its second argument?	When *set1* is finally empty, it could be because it is always empty or because *intersect* has looked at all of its arguments. But when *set2* is empty, *intersect* should not look at any elements in *set1* at all; it knows the result!

Should we have defined *intersect* with an extra question about *set2*

```
(define intersect
  (lambda (set1 set2)
    (letrec
      ((I (lambda (set1)
            (cond
              ((null? set1) (quote ()))
              ((member? (car set1)
                 set2)
               (cons (car set1)
                 (I (cdr set1))))
              (else (I (cdr set1)))))))
      (cond
        ((null? set2) (quote ()))
        (else (I set1))))))
```

Yes, that helps a bit.

Would it make you happy?

Actually, no.

You are not easily satisfied.

Well, *intersect* would immediately return the correct result but this still does not work right with *intersectall*.

Why not?

When one of the *intersect*s returns () in *intersectall*, we know the result of *intersectall*.

And shouldn't *intersectall* say so?

Yes, absolutely.

Well, we could build in a question that looks at the result of *intersect* and *hop*s if necessary?

But somehow that looks wrong.

Why wrong?

Because *intersect* asks this very same question. We would just duplicate it.

Got it. You mean that we should have a version of *intersect* that *hops* all the way over all the *intersects* in *intersectall*

Yes, that would be great.

We can have this.

Can (**letcc** ...) do this? Can we skip and jump from *intersect*?

Yes, we can use *hop* even in *intersect* if we want to jump.

But how would this work? How can *intersect* know where to *hop* to when its second set is empty?

Try this first: make *intersect* a minor function of *intersectall* using *I* as its name.

```
(define intersectall
  (lambda (lset)
    (letcc hop
      (letrec
        ((A ...)
         (I ...))
        (cond
          ((null? lset) (quote ()))
          (else (A lset)))))))
```

```
...
((A (lambda (lset)
      (cond
        ((null? (car lset))
         (hop (quote ())))
        ((null? (cdr lset))
         (car lset))
        (else (I (car lset)
                 (A (cdr lset)))))))
 (I (lambda (s1 s2)
      (letrec
        ((J (lambda (s1)
              (cond
                ((null? s1) (quote ()))
                ((member? (car s1) s2)
                 (J (cdr s1)))
                (else (cons (car s1)
                            (J (cdr s1))))))))
        (cond
          ((null? s2) (quote ()))
          (else (J s1)))))))
...
```

What can we do with minor functions?

We can do whatever we want with the minor version of *intersect*. As long as it does the right thing, nobody cares because it is protected.

Like what?

We could have it check to see if the second argument is the empty set. If it is, we could use *hop* to return the empty set without further delay.

Did you imagine a change like this:

```
...
(I (lambda (s1 s2)
     (letrec
       ((J (lambda (s1)
             (cond
               ((null? s1) (quote ()))
               ((member? (car s1) s2)
                (J (cdr s1)))
               (else (cons (car s1)
                           (J (cdr s1))))))))
       (cond
         ((null? s2) (hop (quote ())))
         (else (J s1))))))
...
```

Yes.

What is the value of (*intersectall lset*) where
lset is ((3 steaks and)
(no food and)
(three baked potatoes)
(3 diet hamburgers))

We know it is ().

Should we go through the whole thing again?

We could skip the part when *A* looks at all the sets until *lset* is almost empty. It is almost the same as before.

What is different?

Every time we recur we need to remember that we must use the minor function *I* on (*car lset*) and the result of the natural recursion.

So what do we have to do when we reach the end of the recursion?	With (3 diet hamburgers) as the value, we now have three *I*s to go back and pick up. We need to determine the value of a. *I* of (three baked potatotes) and (3 diet hamburgers); b. *I* of (no food and) and the value of a; c. *I* of (3 steaks and) and the value of b.
Are there any alternatives?	Correct: there are none.
Okay, let's go. What is the first question?	(*null? s2*) where *s2* is (3 diet hamburgers).
Which is not true.	No, it is not.
Which means we ask for the minor function *J* inside of *I*	Yes, and we get () because (three baked potatoes) and (3 diet hamburgers) have no common elements.
What is the next thing to do?	We determine the value of (*I s1 s2*) where *s1* is (no food and) and *s2* is ().
What is the first question that we ask now?	(*null? s2*) where *s2* is ().
And then?	We determine the value of (**letcc** *hop* (**quote** ())).

Why?

Because (*hop* (**quote** ())) is like a compass needle and it is attracted to the North Pole where the North Pole is (**letcc** *hop*.

And what is the value of this?

().

Done.

Huh? Done?

Yes, all done.

That's quite a feast.

Satisfied?

Yes, pretty much.

Do you want to go hop, skip, and jump around the park before we consume some more food?

That's not a bad idea.

Perhaps it will clear up your mind.

And use up some calories.

Can you write *rember* with (**letrec** . . .)

Sure can:

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

What is the value of
(*rember-beyond-first a lat*)
where *a* is roots
and
lat is (noodles
spaghetti spätzle bean-thread
roots
potatoes yam
others
rice)

(noodles spaghetti spätzle bean-thread).

And (*rember-beyond-first* (**quote** others) *lat*)
where
lat is (noodles
spaghetti spätzle bean-thread
roots
potatoes yam
others
rice)

(noodles
spaghetti spätzle bean-thread
roots
potatoes yam).

And (*rember-beyond-first a lat*)
where *a* is sweetthing
and
lat is (noodles
spaghetti spätzle bean-thread
roots
potatoes yam
others
rice)

(noodles
spaghetti spätzle bean-thread
roots
potatoes yam
others
rice).

And
  (*rember-beyond-first* (**quote** desserts) *lat*)
where
  lat is (cookies
    chocolate mints
      caramel delight ginger snaps
    desserts
    chocolate mousse
    vanilla ice cream
    German chocolate cake
    more desserts
    gingerbreadman chocolate
      chip brownies)

(cookies
 chocolate mints
   caramel delight ginger snaps).

Can you describe in one sentence what *rember-beyond-first* does?

As always, here are our words:
"The function *rember-beyond-first* takes an atom *a* and a lat and, if *a* occurs in the lat, removes all atoms from the lat beyond and including the first occurrence of *a*."

Is this *rember-beyond-first*

```
(define rember-beyond-first
  (lambda (a lat)
    (letrec
      ((R (lambda (lat)
            (cond
              ((null? lat) (quote ()))
              ((eq? (car lat) a)
               (quote ()))
              (else (cons (car lat)
                          (R (cdr lat))))))))
      (R lat))))
```

Yes, this is it. And it differs from *rember* in only one answer.

What is the value of (*rember-upto-last a lat*)
where *a* is roots
and
lat is (noodles
spaghetti spätzle bean-thread
roots
potatoes yam
others
rice)

(potatoes yam
others
rice).

And (*rember-upto-last a lat*)
where *a* is sweetthing
and
lat is (noodles
spaghetti spätzle bean-thread
roots
potatoes yam
others
rice)

(noodles
spaghetti spätzle bean-thread
roots
potatoes yam
others
rice).

Yes, and what is (*rember-upto-last a lat*)
where *a* is cookies
and
lat is (cookies
chocolate mints
caramel delight ginger snaps
desserts
chocolate mousse
vanilla ice cream
German chocolate cake
more cookies
gingerbreadman chocolate
chip brownies)

(gingerbreadman chocolate
chip brownies).

Can you describe in two sentences what *rember-upto-last* does?

Here are our two sentences:
"The function *rember-upto-last* takes an atom *a* and a lat and removes all the atoms from the lat up to and including the last occurrence of *a*. If there are no occurrences of *a*, *rember-upto-last* returns the lat."

Does this sound like yet another version of *rember*	Yes, it does.
How would you change the function *R* in *rember* or *rember-beyond-first* to get *rember-upto-last*	Both functions are the same except that upon discovering the atom *a*, the new version would not stop looking at elements in *lat* but would also throw away everything it had seen so far.
You mean it would forget some computation that it had remembered somewhere?	Yes, it would.
Does this sound like *intersectall*	It sounds like it: it knows that the first few atoms do not contribute to the final result. But then again it sounds different, too.
Different in what sense?	The function *intersectall* knows what the result is; *rember-upto-last* knows which pieces of the list are *not* in the result.
But does it know where it can find the result?	The result is the *rember-upto-last* of the rest of the list.
Suppose *rember-upto-last* sees the atom *a* should it forget the pending computations, and should it restart the process of searching through the rest of the list?	Yes, it should.
We can do this.	You mean we could use (**letcc** ...) to do this, too?
Yes.	How would it continue searching, but ignore the atoms that are waiting to be *cons*ed onto the result?

How would you say, "Do this or that to the rest of the list"?

Easy: do this or that to (*cdr lat*).

And how would you say "Ignore something"?

With a line like (*skip* ...), assuming the beginning of the function looks like (**letcc** *skip*.

Well then ...

... if we had a line like
  (**letcc** *skip*
at the beginning of the function, we could say
  (*skip* (*R* (*cdr lat*)))
when necessary.

Yes, again. Can you write the function *rember-upto-last* now?

Yes, this must be it:

```
(define rember-upto-last
  (lambda (a lat)
    (letcc skip
      (letrec
        ((R (lambda (lat)
              (cond
                ((null? lat) (quote ()))
                ((eq? (car lat) a)
                 (skip (R (cdr lat))))
                (else
                  (cons (car lat)
                    (R (cdr lat))))))))
        (R lat)))))
```

Ready for an example?

Yes, let's try the one with the sweet things.

You mean the one
where *a* is cookies
and
lat is (cookies
chocolate mints
caramel delight ginger snaps
desserts
chocolate mousse
vanilla ice cream
German chocolate cake
more cookies
gingerbreadman chocolate
chip brownies)

Yes, that's the one.

No problem. What is the first thing we do?

We see (**letcc** *skip* and ignore it for a while.

Great. And then?

We ask (*null? lat*).

Why?

Because we use *R* to determine the value of (*rember-upto-last a lat*).

And (*null? lat*) is not true.

But (*eq?* (*car lat*) *a*) is true.

Which means we *skip* and actually determine the value of
(**letcc** *skip* (*R* (*cdr lat*)))
where
lat is (cookies
chocolate mints
caramel delight ginger snaps
desserts
chocolate mousse
vanilla ice cream
German chocolate cake
more cookies
gingerbreadman chocolate
chip brownies)

Yes.

What next?

We ask (*null? lat*).

Which is not true.

And neither is (*eq?* (*car lat*) *a*).

So what?

We recur.

How?

We remember to *cons* chocolate onto the result of (*R* (*cdr lat*))
where
lat is (chocolate mints
caramel delight ginger snaps
desserts
chocolate mousse
vanilla ice cream
German chocolate cake
more cookies
gingerbreadman chocolate
chip brownies).

Next?

Well, this goes on for a while.

You mean it drags on and on with this recursion.

Exactly.

Should we gloss over the next steps?

Yes, they're pretty easy.

What should we look at next?

We should remember to *cons* chocolate, mints, caramel, delight, ginger, snaps, desserts, chocolate, mousse, vanilla, ice, cream, German, chocolate, cake, and more onto the result of (*R* (*cdr lat*))
where
lat is (more cookies
gingerbreadman chocolate
chip brownies).
And we must not forget the (**letcc** *skip* ... at the end!

That's right. And what happens then?

Well, right there we ask (*eq?* (*car lat*) *a*)
where
a is cookies
and
lat is (cookies
gingerbreadman chocolate
chip brownies).

Which is true.

Right, and so we should (*skip* (*R* (*cdr lat*))).

Yes, and that works just as before.

You mean we eliminate all the pending *cons*es and determine the value of
(**letcc** *skip* (*R* (*cdr lat*)))
where
lat is (cookies
gingerbreadman chocolate
chip brownies).

Which we do by recursion.

As always.

What do we have to do when we reach the end of the recursion?

We have to *cons* gingerbreadman, chocolate, chip, and brownies onto ().

Which is (gingerbreadman chocolate
chip brownies)

Yes, and then we need to do the (**letcc** *skip* with this value.

But we know how to do that.

Yes, once we have a value,
(**letcc** *skip*
can be ignored completely.

And so the result is?

(gingerbreadman chocolate
chip brownies).

Doesn't all this hopping and skipping and jumping make you tired?

It sure does. We should take a break and have some refreshments now.

Have you taken a tea break yet?
We're taking ours now.