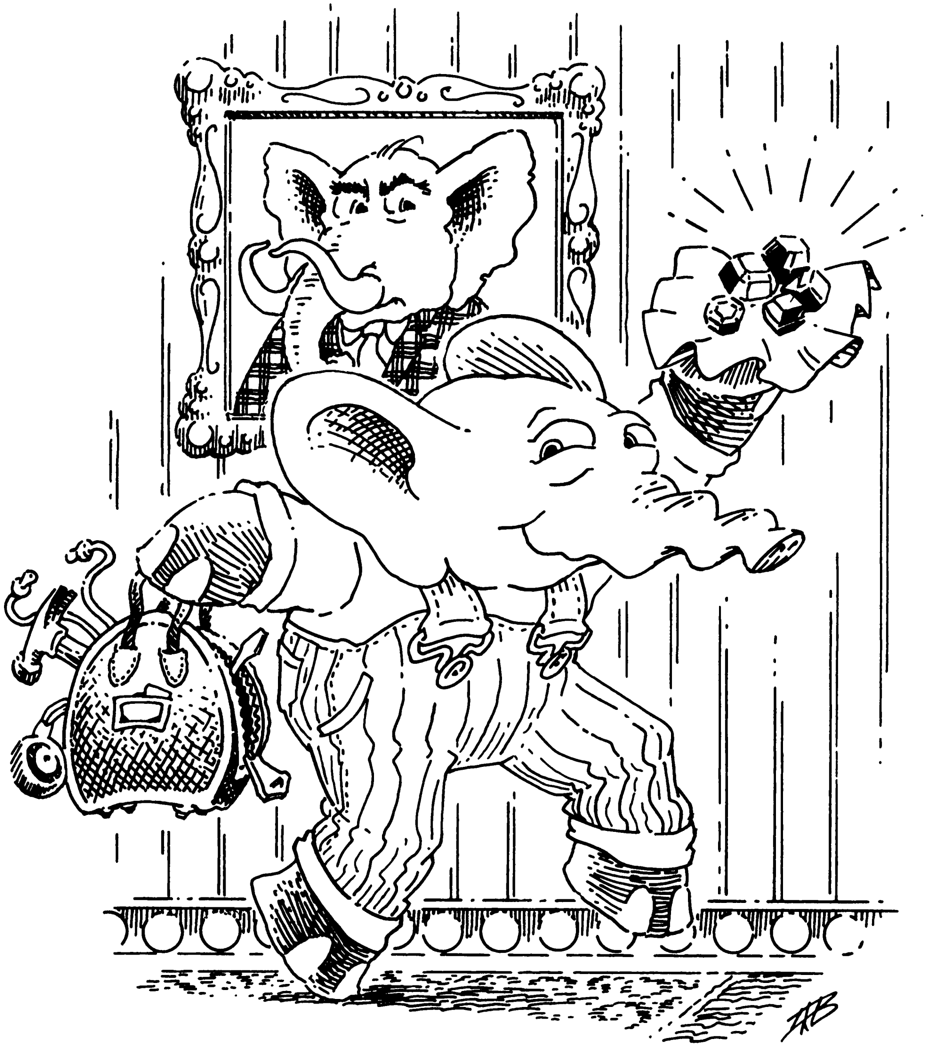
19.
Absconding with the Jewels



We see you have arrived here.

Let's continue.

What is the value of (*deep* 6)

((((((pizza)))))).

Here is *deep* again.

```
(define deep
  (lambda (m)
    (cond
      ((zero? m) (quote pizza))
      (else (cons (deep (sub1 m))
              (quote ()))))))
```

Yes, this is our friend.

How did you determine the value of (*deep* 6)

The value is determined by answering the single question asked by *deep*.

What is the question asked by *deep*

The question is (*zero? m*). If *deep*'s argument is zero, the value of (*deep m*) is pizza. If it is not, we need to determine the value of (*deep* (*sub1 m*)) and *cons* its value onto the null list.

What is the answer to (*zero?* 5)

Why are we doing this? We practiced this kind of thing in chapter 2.

So do you remember these questions?

Sure do.

When (*deep* 0) returns the value pizza, how many *cons* steps do we have to pick up to find out what the value of (*deep* 6) is?

Six.

And they are?

Simple,

we need to:

1. cons the pizza onto ()
2. cons the result of 1 onto ()
3. cons the result of 2 onto ()
4. cons the result of 3 onto ()
5. cons the result of 4 onto ()
6. cons the result of 5 onto ().

And if *deep*'s task had been to make a mozzarella pizza, what steps would we have had to do then?

We just use mozzarella and do whatever we needed to do before:

1. *cons* the mozzarella onto ()
2. *cons* the result of 1 onto ()
3. *cons* the result of 2 onto ()
4. *cons* the result of 3 onto ()
5. *cons* the result of 4 onto ()
6. *cons* the result of 5 onto ().

How about a Neapolitan?

Perhaps we should just define the function *six-layers* and use it to create the pizzas we want:

```
(define six-layers
  (lambda (p)
    (cons
      (cons
        (cons
          (cons
            (cons
              (cons p (quote ()))
              (quote ()))
            (quote ()))
          (quote ()))
        (quote ()))
      (quote ()))))
```

But what if we had started with (*deep* 4)

Then we would have had to define *four-layers* to create these special pizzas.

Define *four-layers*

```
(define four-layers
  (lambda (p)
    (cons
      (cons
        (cons
          (cons p (quote ()))
          (quote ()))
        (quote ()))
      (quote ()))))
```

And how about 1000 layers?

Well, we would need to define the function *thousand-layers*. Somehow we seem to define a function that does exactly what is left to do when *deep*'s argument has become 0.

Yes, that's what we have done.

Isn't there an easier way to do this?

Yes, we can remember this kind of function with a (**set!** ...)

Do you mean something like this?

```
(define deepB
  (lambda (m)
    (cond
      ((zero? m)
       ... (set! toppings ...) ...)
      (else (cons (deepB (sub1 m))
                  (quote ()))))))
```

That is what we mean.

But what do we put where the dots are?

We are about to show you.

And how do we make sure the function still returns *pizza* afterward?

One step at a time. Do you remember (**letcc** ...) from chapter 13?

Yes.

That will help.

You mean what we saw isn't all there is to it?

Not even half.

Okay. Let's see more.

That's what we shall do. Here is a first layer:

```
(define toppings)
```

```
(define deepB
  (lambda (m)
    (cond
      ((zero? m)
       (letcc jump
         (set! toppings jump)
         (quote pizza)))
      (else (cons (deepB (sub1 m))
              (quote ()))))))
```

This use of (**letcc** ...) is different from anything we have seen before.[1]

[1] L: This is impossible in Lisp, but Scheme can do it.

How is it different?

To begin with, the value part of (**letcc** ...) has two parts.

Have we seen this before?

Yes, (**let** ...) and (**letrec** ...) sometimes have more than one expression in the value part.

What else is different about (**letcc** ...)

We don't seem to use *jump* the way we used *hop* in chapter 13.

True. What does *deepB* do with *jump*

It seems to be remembering *jump* in *toppings*.

What could it mean to "remember *jump*"?

We don't even know what *jump* is.

What was *deep* when we asked for the value of (*deep* 9)

Easy: *deep* was the name of the function that we defined at the beginning of the chapter.

So what was *hop* when we asked for the value of (*hop* (**quote** ())) in chapter 13?	We said it was a compass needle. Could *hop* also be a function?
What would be the value of (*deepB* 6)	No problem: ((((((pizza)))))).
And what else would have happened?	We would have remembered *jump*, which appears to be some form of function, in *toppings*.
So what is (*six-layers* (**quote** mozzarella))	((((((mozzarella)))))).
What would be the value of (*toppings* *e*) where *e* is mozzarella	Yes, it would be ((((((mozzarella)))))).
And what about (*toppings* *e*) where *e* is cake	((((((cake)))))).
(*toppings* (**quote** pizza)) would be ((((((pizza)))))) right?	After mozzarella on cake, nothing's a surprise anymore.
Just wait and see.	Why?
Let's add another layer to the cake.	Easy as pie: just *cons* the result onto the null list.
Like this: (*cons* (*toppings* *m*) (**quote** ())) where *m* is cake	That should work, shouldn't it?
You couldn't possibly have known!	It doesn't. Its value would be ((((((cake)))))).

Let's add three slices to the mozzarella:

```
(cons
  (cons
    (cons (toppings (quote mozzarella))
      (quote ()))
    (quote ()))
  (quote ()))
```

((((((mozzarella)))))), same as above. Except that we get mozzarella pizza instead of cake.

Can you explain what happens?

We haven't told you yet, but here is the explanation:

"Whenever we use (*toppings* *m*) it forgets everything surrounding it and adds exactly six layers of parentheses."

Suppose we had started with (*deepB* **4**)

Then *toppings* would be like the function *four-layers* but it would still forget.

That means

```
(cons
  (cons
    (cons (toppings (quote mozzarella))
      (quote ()))
    (quote ()))
  (quote ()))
```

would be ((((mozzarella))))

Yes!

The Twentieth Commandment

When thinking about a value created with (letcc ...), write down the function that is equivalent but does not forget. Then, when you use it, remember to forget.

What would be the value of

```
(cons (toppings (quote cake))
  (toppings (quote cake)))
```

((((cake)))), no?

Yes, *toppings* would forget everything. What would be the value of

```
(cons (toppings (quote cake))
  (cons (toppings (quote mozzarella))
    (cons (toppings (quote pizza))
      (quote ()))))
```

((((cake)))).[1]

[1] S: Here, the value of the first argument is determined before the second one, but in Scheme the order of evaluation in an application is intentionally unspecified.

Yes! When we use a value made with (**letcc** ...) it forgets everything around it.

Just as the commandment says.

Does this mean that we can never *cons* anything onto *toppings*

Yes, never!

Let's try anyway. Here is a relative of *deep*:

```
(define deep&co
  (lambda (m k)
    (cond
      ((zero? m) (k (quote pizza)))
      (else
        (deep&co (sub1 m)
          (lambda (x)
            (k (cons x (quote ())))))))))
```

This is a version of *deep* that uses a collector. It has been a long time since we saw collectors in chapter 8.

Yes, but collectors are useful here, too.

That's good to know.

How could we determine the value of
(*deep* 6)
using *deep&co*

The second argument of *deep&co* must be a function that returns pizza when given pizza.

Which function does that?

(**lambda** (*x*) *x*).

What is the value of
(*deep&co* 0 (**lambda** (*x*) *x*))

pizza.

And what is the value of (*deep&co* 6 (**lambda** (*x*) *x*))	((((((pizza)))))).
(*deep&co* 2 (**lambda** (*x*) *x*))	((pizza)), of course.

And how do we get there?	We ask (*zero?* 2), which isn't true, and then determine the value of
	(*deep&co* 1
	  (**lambda** (*x*)
	    (*k* (*cons x*
	        (**quote** ())))))
	where
	k is (**lambda** (*x*) *x*).

How do we do that?	We check whether the first argument is 0 again, and since it still isn't, we recur with
	(*deep&co* 0
	  (**lambda** (*x*)
	    (*k* (*cons x*
	        (**quote** ())))))
	where
	k is (**lambda** (*x*)
	      (*k2* (*cons x*
	          (**quote** ()))))
	and
	k2 is (**lambda** (*x*) *x*).

Is there a better way to describe the collector?	Yes, it is equivalent to *two-layers*.

```
(define two-layers
  (lambda (p)
    (cons
      (cons p (quote ()))
      (quote ()))))
```

Why?

We can replace *k2* with (**lambda** (x) x), which shows that *k* is the same as

```
(lambda (x)
  (cons x (quote ()))).
```

And then we can replace *k* with this new function.

Are we done now?

Yes, we just use *two-layers* on pizza because the first argument is 0, and doing so gives ((pizza)).

What is the last collector when we determine the value of (*deep&co* 6 (**lambda** (x) x))

When the first argument for *deep&co* finally reaches 0, the collector is the same function as *six-layers*.

And what is the last collector when we determine the value of
(*deep&co* 4 (**lambda** (x) x))

four-layers.

And now take a close look at the function *deep&coB*

```
(define deep&coB
  (lambda (m k)
    (cond
      ((zero? m)
       (let ()
         (set! toppings k)
         (k (quote pizza))))
      (else
        (deep&coB (sub1 m)
          (lambda (x)
            (k (cons x (quote ())))))))))
```

This function remembers the collector in *toppings*.

What is *toppings* after we determine the value of (*deep&coB* 2 (**lambda** (x) x))

It is
(**lambda** (x)
  (k (*cons* x
      (**quote** ()))))
where
k is (**lambda** (x)
      (*k2* (*cons* x
          (**quote** ()))))
and
k2 is (**lambda** (x) x).

So what is it?

It is *two-layers*.

And what is *toppings* after we determine the value of (*deep&coB* 6 (**lambda** (x) x))

It is equivalent to *six-layers*.

What is the value of
(*deep&coB* 4 (**lambda** (x) x))

((((pizza)))).

What is *toppings*

It is just like *four-layers*.

Does this mean that the final collector is related to the function that is equivalent to the one created with (**letcc** ...) in *deepB*

Yes, it is a shadow of the value that (**letcc** ...) creates.

What would be the value of
(*cons* (*toppings* (**quote** cake))
  (*toppings* (**quote** cake)))

(((((cake)))) (((cake)))), not ((((cake)))).

Yes, this version of *toppings* would not forget everything. What would be the value of
(*cons* (*toppings* (**quote** cake))
  (*cons* (*toppings* (**quote** mozzarella))
    (*cons* (*toppings* (**quote** pizza))
      (**quote** ()))))

(((((cake)))) ((((mozzarella)))) ((((pizza))))).

Beware of shadows!

That's correct: shadows are close to the real thing, but we should not forget the difference between them and the real thing.

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

Sure, we defined it in chapter 11.

What is the value of (*two-in-a-row? lat*) where
lat is (mozzarella cake mozzarella)

#f.

What is the value of (*two-in-a-row? lat*) where
lat is (mozzarella mozzarella pizza)

#t.

Here is our original definition of *two-in-a-row?*

```
(define two-in-a-row?
  (lambda (lat)
    (cond
      ((null? lat) #f)
      (else (two-in-a-row-b? (car lat)
              (cdr lat))))))
```

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

Sure, and here is the better version from chapter 12:

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

Explain what *two-in-a-row?* does.

Easy,
it determines whether any atom occurs twice in a row in a list of atoms.

What is the value of (*two-in-a-row*? l*) where *l* is ((mozzarella) (cake) mozzarella)	Are we going to think about "stars"?
Yes. What is the value of (*two-in-a-row*? l*) where *l* is ((mozzarella) (cake) mozzarella)	#f.
What is the value of (*two-in-a-row*? l*) where *l* is ((potato) (chips ((with) fish) (fish)))	#t.
What is the value of (*two-in-a-row*? l*) where *l* is ((potato) (chips ((with) fish) (chips)))	#f.
What is the value of (*two-in-a-row*? l*) where *l* is ((potato) (chips (chips (with) fish)))	#t.
Can you explain what *two-in-a-row*?* does?	Here are our words: "The function *two-in-a-row*?* processes a list of S-expressions and checks whether any atom occurs twice in a row, regardless of parentheses."
What would be the value of (*walk l*) where *l* is ((potato) (chips (chips (with))) fish)	We haven't seen *walk* yet.

Here is the definition of *walk*

```
(define leave)
```

```
(define walk
  (lambda (l)
    (cond
      ((null? l) (quote ()))
      ((atom? (car l))
       (leave (car l)))
      (else
        (let ()
          (walk (car l))
          (walk (cdr l)))))))
```

Have we seen something like this before?

Yes, *walk* is the minor function *lm* in *leftmost*.

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

And what does *lm* do?

It searches a list of S-expressions from left to right for the first atom and then gives this atom to a value created by (**letcc** ...).

So, what would be the value of (*walk l*) where
l is ((potato) (chips (chips (with))) fish)

If *leave* is a magnetic needle like *skip*, *walk* uses it on the leftmost atom.

Does this mean *walk* is like *leftmost* if we put the right kind of value into *leave*

Yes!

What would be the value of (*start-it l*) where
l is ((potato) (chips (chips (with))) fish)
and the definition for *start-it* is

```
(define start-it
  (lambda (l)
    (letcc here
      (set! leave here)
      (walk l))))
```

Okay, now *leave* would be a needle!

Why?	Because *start-it* first sets up a North Pole and then remembers it in *leave*. When we finally get to (*leave* (*car l*)), *leave* is a needle that is attracted to the North Pole in *start-it*.
What would be the value of *leave*	It would be a function that does whatever is left to do after the value of (*start-it l*) is determined.
And what would be the value of (*start-it l*)	It would be potato.
Can you explain how to determine the value of (*start-it l*)	Your words could be: "The function *start-it* sets up a North Pole in *here*, remembers it in *leave*, and then determines the value of (*walk l*). The function *walk* crawls over *l* from left to right until it finds an atom and then uses *leave* to return that atom as the value of (*start-it l*)."
Write the function *waddle* which is like *walk* except for two small things.	What things?

First, if (*leave* (*car l*)) ever has a value, *waddle* should look at the elements in (*cdr l*)	That's easy: we just add (*waddle* (*cdr l*)) after (*leave* (*car l*)), ordering the two steps using (**let** () ...):
	(**let** ()
	  (*leave* (*car l*))
	  (*waddle* (*cdr l*)))
	But why would we want to do this? We know that *leave* always forgets.

Because of our second change.	And that is?

Second, before determining the value of
(leave (car l))
the function *waddle* should remember in *fill* what is left to do.

This is similar to what we did with *deepB*.

```
(define fill)
```

```
(define waddle
  (lambda (l)
    (cond
      ((null? l) (quote ()))
      ((atom? (car l))
       (let ()
         (letcc rest
           (set! fill rest)
           (leave (car l)))
         (waddle (cdr l))))
      (else (let ()
              (waddle (car l))
              (waddle (cdr l)))))))
```

Is it now possible that *(leave (car l))* yields a value?

No, not really! But something similar may occur: if *fill* is ever used, it will restart *waddle*.

One step at a time! We need to learn to walk before we run! What would be the value of
(start-it2 l)
where
l is ((donuts)
(cheerios (cheerios (spaghettios)))
donuts)
and

```
(define start-it2
  (lambda (l)
    (letcc here
      (set! leave here)
      (waddle l))))
```

donuts,
of course.

But?

In addition, *waddle* would remember *rest* in *fill*.

What is *rest*	It is a needle, just as *jump* in *deepB*.
Didn't we say that *jump* would be like a function?	Yes, it would have been like a function, but when used, it would have also forgotten what to do afterward.
What kind of function does *rest* correspond to?	If *rest* is to *waddle* what *jump* is to *deepB*, the function ignores its argument and then it acts like *waddle* for the rest of the list until it encounters the next atom.
Why does this function ignore its argument?	Because the new North Pole creates a function that remembers the rest of what *waddle* has to do after (**letcc** ...) produces a value. Since the value of the first expression in the body of (**let** () ...) is ignored, the function throws away the value of the argument.
What does the function do afterward?	It looks for the first atom in the rest of the list and then uses *leave* on it. It also remembers what is left to do.
What is the rest of the list?	Since *l* is ((donuts) (cheerios (cheerios (spaghettios))) donuts), the rest of the list without the first atom is (() (cheerios (cheerios (spaghettios))) donuts).

Can you define the function that corresponds to *rest*

No problem:

```
(define rest1
  (lambda (x)
    (waddle l1)))
```

where
l1 is (()
(cheerios (cheerios (spaghettios)))
donuts).

Was this really no problem?

Well, *x* is never used but that's no problem.

What would be the value of
(*get-next* (**quote** go))
where

```
(define get-next
  (lambda (x)
    (letcc here-again
      (set! leave here-again)
      (fill (quote go)))))
```

The value would be cheerios.

Why?

Because *fill* is like *rest1*, except that it forgets what to do. Since (*rest1* (**quote** go)) would eventually determine the value of (*leave* (**quote** cheerios)), and since *leave* is just the North Pole *here-again*, the result of (*get-next* (**quote** go)) would be just cheerios.

And what else would have happened?

Well, *fill* would now remember a new needle.

And what would this needle correspond to?

It would have corresponded to a function like *rest1*, except that the rest of the list would have been smaller.

Define this function.

```
(define rest2
  (lambda (x)
    (waddle l2)))
```

where
l2 is (((cheerios (spaghettios)))
donuts).

Does *get-next* deserve its name?

Yes, it sets up a new North Pole for *fill* to return the next atom to.

What else does it do?

Just before *fill* determines the next atom in the list of S-expressions that was given to *start-it2*, it changes itself so that it can resume the search for the next atom when used again.

Does this mean that the value of
(*get-next* (**quote** go))
would be cheerios again?

Yes, if after determining the first value of (*get-next* (**quote** go)) we asked for the value again, we would again receive cheerios, because the original list
was ((donuts)
(cheerios (cheerios (spaghettios)))
donuts).

And if we were to determine the value of (*get-next*[1] (**quote** go)) a third time, what would we get?

spaghettios,
because the next atom in the list is spaghettios.

[1] This is not a mathematical function.

Let's imagine we asked
(*get-next* (**quote** go))
for a fourth time.

donuts.

Last time: (*get-next* (**quote** go))

Wow!

Wow, what?	Since donuts is the very last atom in *l*, *waddle* finally reaches (*null? l*) where *l* is ().
And then?	Well, the final value is ().
What is so bad about that?	If we had done all of what we intended to do, we would be back where we originally asked what the value of (*start-it2 l*) would be where *l* was ((donuts) (cheerios (cheerios (spaghettios))) donuts).
And from there on?	Heaven knows what would happen. Perhaps it was a good thing that we always asked "what would be the value of" instead of "what is the value of."
Why would it get back to *start-it2*	Once the original input list to *waddle* is completely exhausted, it returns a value without using any needle. In turn, *start-it2* returns this value, too.
What should happen instead?	If *get-next* really deserves its name, it should return (), so that we know that the list is completely exhausted.
But didn't we say that *get-next* deserved its name?	We did and it does most of the time. Indeed, with the exception of the very last case, when the original input list is exhausted, *get-next* works exactly as expected.
Does this mean that *start-it2* would deserve the name *get-first*	No, it wouldn't. It does get the first atom, but later it also returns () when everything is over.

Is it also true that *waddle* doesn't use *leave* to return ()	Yes, it is.
And is it true that using (*leave* (**quote** ())) after the list is exhausted would help things?	Yes, it would: if *leave* were used, then *get-next* would return () eventually, and we would know that the list was exhausted.
Does *get-first* deserve its name:	Yes!

```
(define get-first
  (lambda (l)
    (letcc here
      (set! leave here)
      (waddle l)
      (leave (quote ())))))
```

Does (*get-first* *l*) return () when *l* doesn't contain an atom?	Yes!
And does *get-next* deserve its name?	Yes!
Does (*get-next* (**quote** go)) return () when the latest argument of *get-first* didn't contain an atom?	Yes!
(*get-first* *l*) where *l* is (donut)	donut.
(*get-next* (**quote** go))	().
What would (*get-first* *l*) be where *l* was (fish (chips))	fish.

What would be (*get-next* (**quote** go))

chips.

What would be (*get-next* (**quote** go))

().

Are there any more atoms to look at?

No!

What would (*get-first* *l*) be
where
l is (fish (chips) chips)

fish.

What would be (*get-next* (**quote** go))

chips.

What would be (*get-next* (**quote** go))

chips.

Is it true that chips occurs twice in a row in (fish (chips) chips)

Yes, it does! And by using *get-first* and *get-next*, we can find out!

Should we define *two-in-a-row*?* like this:

```
(define two-in-a-row*?
  (lambda (l)
    (let ((fst (get-first l)))
      (if (atom? fst)
          (two-in-a-row-b*? fst)
          #f))))
```

Yes, and here is *two-in-a-row-b*?*:

```
(define two-in-a-row-b*?
  (lambda (a)
    (let ((n (get-next (quote go))))
      (if (atom? n)
          (or (eq? n a)
              (two-in-a-row-b*? n))
          #f))))
```

Why does *two-in-a-row*?* check whether *fst* is an atom?

Returning (), a non-atom, is *get-first*'s way of saying that there is no atom in *l*.

Why does *two-in-a-row-b*?* not take the list as an argument?

Because *get-next* knows how to get the rest of the atoms, without being told about *l*.

Why does *two-in-a-row-b*?* check whether *n* is an atom?

Returning (), a non-atom, is *get-next*'s way of saying that there are no more atoms in *l*.

Didn't we forget The Thirteenth Commandment?

That's easy to fix, and since *get-first* is only used once, we can get rid of it, too:

```
(define two-in-a-row*?
  (letrec
    ((T? (lambda (a)
           (let ((n (get-next 0)))
             (if (atom? n)
                 (or (eq? n a)
                     (T? n))
                 #f))))
     (get-next
       (lambda (x)
         (letcc here-again
           (set! leave here-again)
           (fill (quote go)))))
     (fill (lambda (x) x))
     (waddle
       (lambda (l)
         (cond
           ((null? l) (quote ()))
           ((atom? (car l))
            (let ()
              (letcc rest
                (set! fill rest)
                (leave (car l)))
              (waddle (cdr l))))
           (else (let ()
                   (waddle (car l))
                   (waddle (cdr l)))))))
     (leave (lambda (x) x)))
    (lambda (l)
      (let ((fst (letcc here
                   (set! leave here)
                   (waddle l)
                   (leave (quote ())))))
        (if (atom? fst) (T? fst) #f)))))
```

Isn't this a large definition?	Yes, it is. It was a good idea to develop it in several steps.
And what's (*two-in-a-row*? l*) where *l* is (((food) ()) (((food))))	#t.
Are you hungry yet?	Very!
Okay, let's hurry then. This is only an appetizer anyway.	What's next?
Real food.	Let's have a banquet.
Hold on!	Why?
Don't forget your banquet, but we still need to do something.	What?

Hop, Skip, and Jump!