The Seasoned Schemer

Daniel P. Friedman and Matthias Felleisen

Foreword and Afterword by Guy L. Steele Jr.

[redacted] (Contents)

Foreword

If you give someone a fish, he can eat for a day.
If you teach someone to fish, he can eat for a lifetime.

This familiar proverb applies also to data structures in programming languages.

If you have read *The Little Lisper* (recently revised and retitled: *The Little Schemer*), the predecessor to this book, you know that lists of things are at the heart of Lisp. Indeed, "LISP" originally stood for "LISt Processing." By the same token, I suppose that the C programming language could have been called CHAP (for "CHAracter Processing") and Fortran could have been FLOP (for "FLOating-point Processing").

Now C without characters or Fortran without its floating-point numbers would be almost unthinkable. They would be completely different languages, perhaps almost useless. What about Lisp without lists? Well, Lisp has not only lists but functions that perform computations. And we have learned, slowly and sometimes laboriously over the years, that while lists are the heart of Lisp, functions are the soul.

Lisp must, of course, have lists; yet functions are enough. Dan and Matthias will show you the way. *The Little Lisper* was truly a feast; but, as you will see, there is more to life than food.

Have you eaten? Very good. Now you are prepared for the real journey.

Come, learn to fish!

—Guy L. Steele Jr.

Preface

To celebrate the twentieth anniversary of Scheme we revised *The Little LISPer* a third time, gave it the more accurate title *The Little Schemer*, and wrote a sequel: *The Seasoned Schemer*.

The goal of this book is to teach the reader to think about the nature of computation. Our first task is to decide which language to use to communicate this concept. There are three obvious choices: a natural language, such as English; formal mathematics; or a programming language. Natural languages are ambiguous, imprecise, and sometimes awkwardly verbose. These are all virtues for general communication, but something of a drawback for communicating concisely as precise a concept as the power of recursion, the subtlety of control, and the true role of state. The language of mathematics is the opposite of natural language: it can express powerful formal ideas with only a few symbols. We could, for example, describe the semantic content of this book in less than a page of mathematics, but conveying how to harness the power of functions in the presence of state and control is nearly impossible. The marriage of technology and mathematics presents us with a third, almost ideal choice: a programming language. Programming languages seem the best way to convey the nature of computation. They share with mathematics the ability to give a formal meaning to a set of symbols. But unlike mathematics, programming languages can be directly experienced—you can take the programs in this book, observe their behavior, modify them, and experience the effect of these modifications.

Perhaps the best programming language for teaching about the nature of computation is Scheme. Scheme is symbolic and numeric—the programmer does not have to make an explicit mapping between the symbols and numerals of his own language and the representations in the computer. Scheme is primarily a functional language, but it also provides assignment, set!, and a powerful control operator, letcc (or call-with-current-continuation), so that programmers can explicitly characterize the change of state. Since our only concerns are the principles of computation, our treatment is limited to the whys and wherefores of just a few language constructs: car, cdr, cons, eq?, atom?, null?, zero?, add1, sub1, number?, lambda, cond, define, or, and, quote, letrec, letcc (or call-with-current-continuation), let, set!, and if. Our language is an *idealized* Scheme.

The Little Schemer and *The Seasoned Schemer* will not directly introduce you to the practical world of programming, but a mastery of the concepts in these books provides a start toward understanding the nature of computation.

Acknowledgments

We particularly want to thank Bob Filman for contributing to the TeXery and Dorai Sitaram for his incredibly clever Scheme program SLaTeX. Kent Dybvig's Chez Scheme made programming in Scheme a most pleasant experience. We gratefully acknowledge criticisms and suggestions from Steve Breeser, Eugene Byon, Corky Cartwright, Richard Cobbe, David Combs, Kent Dybvig, Rob Friedman, Gustavo Gomez-Espinoza-Martinez, Dmitri Gusev, Chris Haynes, Erik Hilsdale, Eugene Kohlbecker, Shriram Krishnamurthi, Julia Lawall, Shinnder Lee, Collin McCurdy, Suzanne Menzel, John Nienart, Jon Rossie, David Roth, Jonathan Sobel, George Springer, Guy Steele, John David Stone, Vikram Subramaniam, Perry Wagle, Mitch Wand, Peter Weingartner, Melissa Wingard-Phillips, Beata Winnicka, and John Zuckerman.

Hints for the Reader

Do not rush through this book. Read carefully; valuable hints are scattered throughout the text. Do not read the book in fewer than five sittings. Read systematically. If you do not ***fully*** understand one chapter, you will understand the next one even less. The questions are ordered by increasing difficulty; it will be hard to answer later ones if you cannot solve the earlier ones.

The book is a dialogue between you and us about interesting examples of Scheme programs. Try the examples while you read. Schemes and Lisps are readily available. While there are minor syntactic variations between different implementations (primarily the spelling of particular names and the domain of specific functions), Scheme is basically the same throughout the world. To work with Scheme, you will need to define `atom?`, `sub1`, and `add1`, which we introduced in *The Little Schemer*:

```
(define atom?
  (lambda (x)
    (and (not (pair? x)) (not (null? x)))))
```

Those readers who have read *The Little LISPer* need to understand that the empty list, `()`, is no longer an atom. To find out whether your Scheme has the correct definition of `atom?`, try `(atom? (quote ()))` and make sure it returns `#f`. To work with Lisp, you will also have to add the function `atom?`:

```
(defun atom? (x)
  (not (listp x)))
```

Moreover, you may need to modify the programs slightly. Typically, the material requires only a few changes. Suggestions about how to try the programs in the book are provided in the framenotes. Framenotes preceded by "S:" concern Scheme, those by "L:" concern Common Lisp. The framenotes in this book, especially those concerning Common Lisp, assume knowledge of the framenotes in *The Little Schemer* or of the basics of Common Lisp.

We do not give any formal definitions in this book. We believe that you can form your own definitions and will thus remember them and understand them better than if we had written each one for you. But be sure you know and understand the Commandments thoroughly before passing them by. The key to programming is recognizing patterns in data and processes. The *Commandments* highlight the patterns. Early in the book, some concepts are narrowed for simplicity; later, they are expanded and qualified. You should also know that, while everything in the book is Scheme (chapter 19 is not Lisp), the language incorporates more than needs to be covered in a text on the nature of computation.

We use a few notational conventions throughout the text, primarily changes in typeface for different classes of symbols. Variables and the names of primitive operations are in *italic*. Basic data, including numbers and representations of truth and falsehood, is set in sans serif. Keywords, i.e., **letrec**, **letcc**, **let**, **if**, **set!**, **define**, **lambda**, **cond**, **else**, **and**, **or**, and **quote** are in **boldface**. When you try the programs, you may ignore the typefaces but not the related framenotes. To highlight this role of typefaces, the programs in framenotes are completely set in a `typewriter` face. The typeface distinctions can be safely ignored until chapter 20, where we treat programs as data.

Finally, Webster defines "punctuation" as the act of punctuating; specifically, the act, practice, or system of using standardized marks in writing and printing to separate sentences or sentence elements or to make the meaning clearer. We have taken this definition literally and have abandoned some familiar uses of punctuation in order to make the meaning clearer. Specifically, we have dropped the use of punctuation in the left-hand column whenever the item that precedes such punctuation is a term in our programming language.

Once again, food appears in many of our examples, and we are no more health conscious than we were before. We hope the food provides you with a little distraction and keeps you from reading too much of the book at one sitting.

Ready to start?	Good luck!

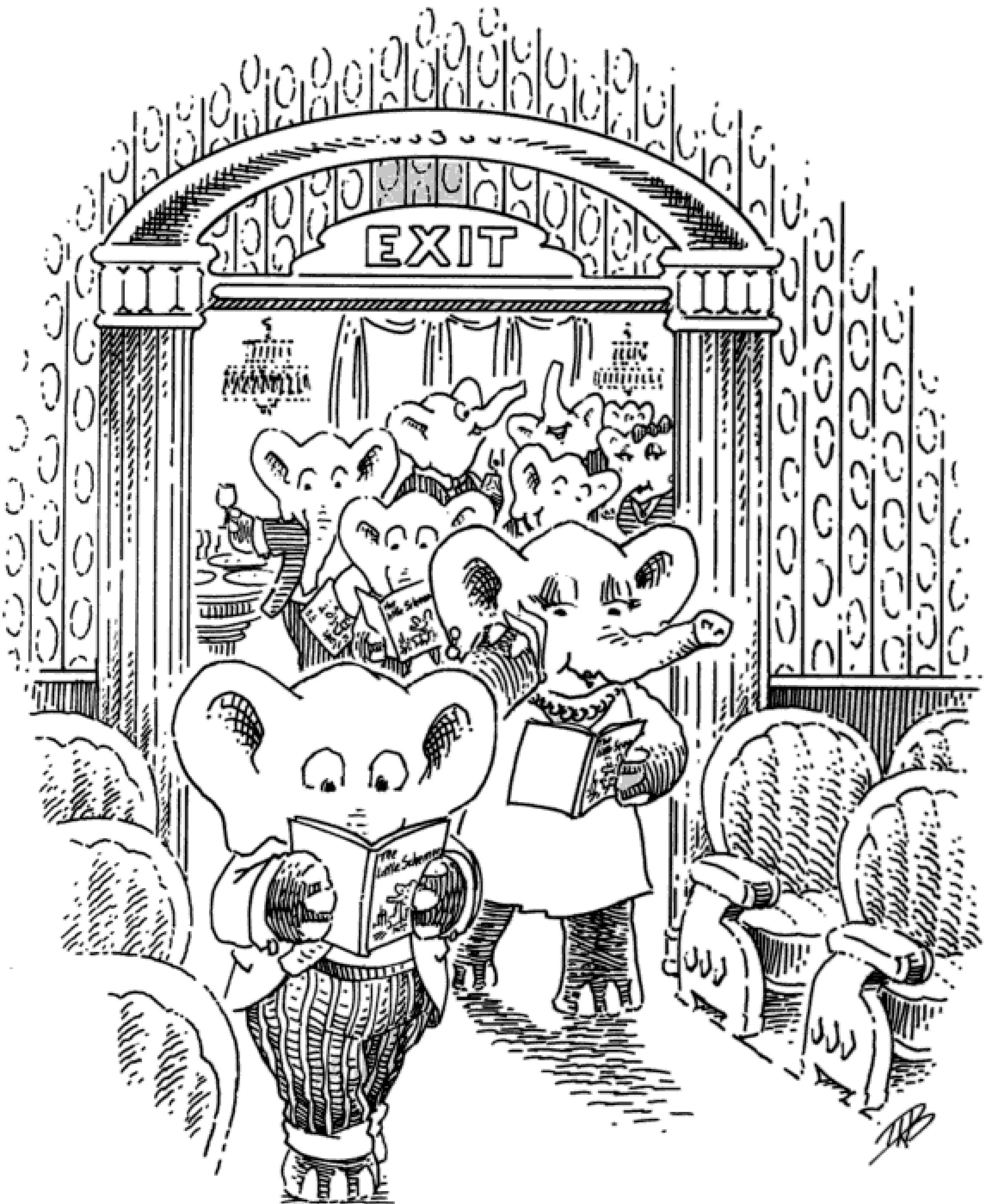
We hope you will enjoy the challenges waiting for you on the following pages.

Bon appétit!

Daniel P. Friedman
Matthias Felleisen

The Seasoned Schemer

11. Welcome Back to the Show



Welcome back.

It's a pleasure.

Have you read *The Little LISPer*?[1]

[1] Or *The Little Schemer*.

#f.

Are you sure you haven't read *The Little LISPer*?

Well, ...

Do you know about Lambda the Ultimate?

#t.

Are you sure you have read that much of *The Little LISPer*?

Absolutely.[1]

[1] If you are familiar with recursion and know that functions are values, you may continue anyway.

Are you acquainted with *member?*

```
(define member?
  (lambda (a lat)
    (cond
      ((null? lat) #f)
      (else (or (eq? a (car lat))
                (member? a (cdr lat)))))))
```

Sure, *member?* is a good friend.

What is the value of (*member? a lat*)
where *a* is sardines
and
lat is (Italian sardines spaghetti parsley)

#t, but this is not interesting.

What is the value of (*two-in-a-row? lat*)
where
lat is (Italian sardines spaghetti parsley)

#f.

Are *two-in-a-row?* and *member?* related?

Yes, both visit each element of a list of atoms up to some point. One checks whether an atom is in a list, the other checks whether any atom occurs twice in a row.

What is the value of (*two-in-a-row? lat*) where
lat is (Italian sardines sardines spaghetti parsley)

#t.

What is the value of (*two-in-a-row? lat*) where
lat is (Italian sardines more sardines spaghetti)

#f.

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

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

Is this close to what *two-in-a-row?* should look like?

```
(define two-in-a-row?
  (lambda (lat)
    (cond
      ((null? lat) ...)
      (else ...
            (two-in-a-row? (cdr lat))
            ...))))
```

That looks fine. The dots in the first line should be replaced by #f.

What should we do with the dots in the second line?

We know that there is at least one element in *lat*. We must find out whether the next element in *lat*, if there is one, is identical to this element.

Doesn't this sound like we need a function to do this? Define it.

```
(define is-first?
  (lambda (a lat)
    (cond
      ((null? lat) #f)
      (else (eq? (car lat) a)))))
```

Can we now complete the definition of *two-in-a-row?*

Yes, now we have all the pieces and we just need to put them together:

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

There is a different way to accomplish the same task.

We have seen this before: most functions can be defined in more than one way.

What does *two-in-a-row?* do when *is-first?* returns #f

It continues to search for two atoms in a row in the rest of *lat*.

Is it true that (*is-first? a lat*) may respond with #f for two different situations?

Yes, it returns #f when *lat* is empty or when the first element in the list is different from *a*.

In which of the two cases does it make sense for *two-in-a-row?* to continue the search?

In the second one only, because the rest of the list is not empty.

Should we change the definitions of *two-in-a-row?* and *is-first?* in such a way that *two-in-a-row?* leaves the decision of whether continuing the search is useful to the revised version of *is-first?*

That's an interesting idea.

Here is a revised version of *two-in-a-row?*

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

Can you define the function *is-first-b?* which is like *is-first?* but uses *two-in-a-row?* only when it is useful to resume the search?

That's easy. If *lat* is empty, the value of (*is-first-b? a lat*) is #f. If *lat* is non-empty and if (*eq?* (*car lat*) *a*) is not true, it determines the value of (*two-in-a-row? lat*).

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

Why do we determine the value of (*two-in-a-row? lat*) in *is-first-b?*

If *lat* contains at least one atom and if the atom is not the same as *a*, we must search for two atoms in a row in *lat*. And that's the job of *two-in-a-row?*.

When *is-first-b?* determines the value of (*two-in-a-row? lat*) what does *two-in-a-row?* actually do?

Since *lat* is not empty, it will request the value of (*is-first-b?* (*car lat*) (*cdr lat*)).

Does this mean we could write a function like *is-first-b?* that doesn't use *two-in-a-row?* at all?

Yes, we could. The new function would recur directly instead of through *two-in-a-row?*.

Let's use the name *two-in-a-row-b?* for the new version of *is-first-b?*

That sounds like a good name.

How would *two-in-a-row-b?* recur?

With (*two-in-a-row-b?* (*car lat*) (*cdr lat*)), because that's the way *two-in-a-row?* used *is-first-b?*, and *two-in-a-row-b?* is used in its place now.

So what is *a* when we are asked to determine the value of (*two-in-a-row-b? a lat*)

It is the atom that precedes the atoms in *lat* in the original list.

Can you fill in the dots in the following definition of *two-in-a-row-b?*

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

That's easy. It is just like *is-first?* except that we know what to do when (*car lat*) is not equal to *preceding*:

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

What is the natural recursion in *two-in-a-row-b?*

The natural recursion is (*two-in-a-row-b?* (*car lat*) (*cdr lat*)).

Is this unusual?

Definitely: both arguments change even though the function asks questions about its second argument only.

Why does the first argument to *two-in-a-row-b?* change all the time?

As the name of the argument says, the first argument is always the atom that precedes the current *lat* in the list of atoms that *two-in-a-row?* received.

Now that we have *two-in-a-row-b?* can you define *two-in-a-row?* a final time?

Trivial:

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

Let's see one more time how *two-in-a-row?* works.

Okay.

(*two-in-a-row? lat*)
where
lat is (b d e i i a g)

This looks like a good example. Since *lat* is not empty, we need the value of
(*two-in-a-row-b? preceding lat*)
where *preceding* is b
and
lat is (d e i i a g)

(*null? lat*)
where
lat is (d e i i a g)

#f.

(*eq?* (*car lat*) *preceding*)
where *preceding* is b
and
lat is (d e i i a g)

#f,
because d is not b.

And now?

Next we need to determine the value of (*two-in-a-row-b? preceding lat*) where *preceding* is d
and
lat is (e i i a g).

Does it stop here?

No, it doesn't. After determining that *lat* is not empty and that (*eq?* (*car lat*) *preceding*) is not true, we must determine the value of
(*two-in-a-row-b? preceding lat*)
where *preceding* is e
and
lat is (i i a g).

Enough?

Not yet. We also need to determine the value of (*two-in-a-row-b? preceding lat*)
where *preceding* is i
and
lat is (i a g).

And?

Now (*eq?* (*car lat*) *preceding*) is true because *preceding* is i
and
lat is (i a g).

So what is the value of (*two-in-a-row? lat*) where
lat is (b d e i i a g)

#t.

Do we now understand how *two-in-a-row?* works?

Yes, this is clear.

What is the value of (*sum-of-prefixes tup*) where
tup is (2 1 9 17 0)

(2 3 12 29 29).

(*sum-of-prefixes tup*) where
tup is (1 1 1 1 1)

(1 2 3 4 5).

Should we try our usual strategy again?

We could. The function visits the elements of a tup, so it should follow the pattern for such functions:

```
(define sum-of-prefixes
  (lambda (tup)
    (cond
      ((null? tup) ...)
      (else ...
            (sum-of-prefixes (cdr tup))
            ...))))
```

What is a good replacement for the dots in the first line?

The first line is easy again. We must replace the dots with (**quote** ()), because we are building a list.

Then how about the second line?

The second line is the hard part.

Why?

The answer should be the sum of all the numbers that we have seen so far *cons*ed onto the natural recursion.

Let's do it!

The function does not know what all these numbers are. So we can't form the sum of the prefix.

How do we get around this?

The trick that we just saw should help.

Which trick?

Well, *two-in-a-row-b?* receives two arguments and one tells it something about the other.

What does *two-in-a-row-b?*'s first argument say about the second argument.

Easy: the first argument, *preceding*, always occurs just before the second argument, *lat*, in the original list.

So how does this help us with *sum-of-prefixes*

We could define *sum-of-prefixes-b*, which receives *tup* and the sum of all the numbers that precede *tup* in the tup that *sum-of-prefixes* received.

Let's do it!

```
(define sum-of-prefixes-b
  (lambda (sonssf tup)
    (cond
      ((null? tup) (quote ()))
      (else (cons (o+ sonssf (car tup))
                  (sum-of-prefixes-b
                    (o+ sonssf (car tup))
                    (cdr tup)))))))
```

Isn't *sonssf* a strange name?

It is an abbreviation. Expanding it helps a lot: *sum of numbers seen so far.*

What is the value of
(*sum-of-prefixes-b sonssf tup*)
where *sonssf* is 0
and
tup is (1 1 1)

Since *tup* is not empty, we need to determine the value of
(*cons* 1 (*sum-of-prefixes-b* 1 *tup*))
where
tup is (1 1).

And what do we do now?

We *cons* 2 onto the value of
(*sum-of-prefixes-b* 2 *tup*)
where
tup is (1).

Next?

We need to remember to *cons* the value 3 onto (*sum-of-prefixes-b* 3 *tup*)
where
tup is ().

What is left to do?

We need to:
a. cons 3 onto ()
b. cons 2 onto the result of a
c. cons 1 onto the result of b.
And then we are done.

Is *sonssf* a good name?

Yes, every natural recursion with *sum-of-prefixes-b* uses the sum of all the numbers preceding *tup*.

Define *sum-of-prefixes* using *sum-of-prefixes-b*

Obviously the first sum for *sonssf* must be 0:

```
(define sum-of-prefixes
  (lambda (tup)
    (sum-of-prefixes-b 0 tup)))
```

The Eleventh Commandment

Use additional arguments when a function needs to know what other arguments to the function have been like so far.

Do you remember what a tup is?

A tup is a list of numbers.

Is (1 1 1 3 4 2 1 1 9 2) a tup?

Yes, because it is a list of numbers.

What is the value of (*scramble tup*)
where
tup is (1 1 1 3 4 2 1 1 9 2)

(1 1 1 1 1 4 1 1 1 9).

(*scramble tup*)
where
tup is (1 2 3 4 5 6 7 8 9)

(1 1 1 1 1 1 1 1 1).

(*scramble tup*)
where
tup is (1 2 3 1 2 3 4 1 8 2 10)

(1 1 1 1 1 1 1 1 2 8 2).

Have you figured out what it does yet?

It's okay if you haven't. It's kind of crazy. Here's our explanation:
"The function *scramble* takes a non-empty tup in which no number is greater than its own index, and returns a tup of the same length. Each number in the argument is treated as a backward index from its own position to a point earlier in the tup. The result at each position is found by counting backward from the current position according to this index."

If *l* is (1 1 1 3 4 2 1 1 9 2)
what is the prefix of (4 2 1 1 9 2) in *l*

(1 1 1 3 4),
because the prefix contains the first element, too.

And if *l* is (1 1 1 3 4 2 1 1 9 2)
what is the prefix of (2 1 1 9 2) in *l*

(1 1 1 3 4 2).

Is it true that (*scramble tup*) must know something about the prefix for every element of *tup*

We said that it needs to know the entire prefix of each element so that it can use the first element of *tup* as a backward index to *pick* the corresponding number from this prefix.

Does this mean we have to define another function that does most of the work for *scramble*

Yes, because *scramble* needs to collect information about the prefix of each element in the same manner as *sum-of-prefixes*.

What is the difference between *scramble* and *sum-of-prefixes*

The former needs to know the actual prefix, the latter needs to know the sum of the numbers in the prefix.

What is (*pick n lat*)
where *n* is 4
and
lat is (4 3 1 1 1)

1.

What is (*pick n lat*)
where *n* is 2
and
lat is (2 4 3 1 1 1)

4.

Do you remember *pick* from chapter 4?

If you do, have an ice cream. If you don't, here it is:

```
(define pick
  (lambda (n lat)
    (cond
      ((one? n) (car lat))
      (else (pick (sub1 n) (cdr lat))))))
```

Here is *scramble-b*

```
(define scramble-b
  (lambda (tup rev-pre)
    (cond
      ((null? tup) (quote ()))
      (else
        (cons (pick (car tup)
                    (cons (car tup) rev-pre))
          (scramble-b (cdr tup)
            (cons (car tup) rev-pre)))))))
```

How do we get *scramble-b* started?

A better question is: how does it work?

What does *rev-pre* abbreviate?

That is always the key to these functions. Apparently, *rev-pre* stands for reversed prefix.

If
tup is (1 1 1 3 4 2 1 1 9 2)
and
rev-pre is ()
what is the reversed prefix of
(*cdr tup*)

It is the result of *cons*ing (*car tup*) onto *rev-pre*: (1).

If
tup is (2 1 1 9 2)
and
rev-pre is (4 3 1 1 1)
what is the reversed prefix of
(1 1 9 2)
which is (*cdr tup*)

Since (*car tup*) is 2, it is
(2 4 3 1 1 1).

Do we need to know what
rev-pre is when
tup is ()

No, because we know the result of
(*scramble tup rev-pre*)
when *tup* is the empty list.

How does *scramble-b* work?

The function *scramble-b* receives a tup and the reverse of its proper prefix. If the tup is empty, it returns the empty list. Otherwise, it constructs the reverse of the complete prefix and uses the first element of tup as a backward index into this list. It then processes the rest of the tup and *cons*es the two results together.

How does *scramble* get *scramble-b* started?

Now, it's no big deal. We just give *scramble-b* the tup and the empty list, which represents the reverse of the proper prefix of the tup:

```
(define scramble
  (lambda (tup)
    (scramble-b tup (quote ()))))
```

Let's try it.

That's a good idea.

The function *scramble* is an unusual example. You may want to work through it a few more times before we have a snack.

Okay.

Tea time.

Don't eat too much. Leave some room for dinner.

12. Take Cover

What is (*multirember a lat*)
where *a* is tuna
and
lat is (shrimp salad tuna salad and tuna)

(shrimp salad salad and),
but we already knew that from chapter 3.

Does *a* change as *multirember* traverses *lat*

No, *a* always stands for tuna.

Well, wouldn't it be better if we did not have to remind *multirember* for every natural recursion that *a* still stands for tuna

Yes, it sure would be a big help in reading such functions, especially if several things don't change.

That's right. Do you think the following definition of *multirember* is correct?

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

Whew, the *Y* combinator in the middle looks difficult.

What is this function?

```
(define ???
  ((lambda (le)
     ((lambda (f) (f f))
      (lambda (f)
        (le (lambda (x) ((f f) x))))))
   (lambda (length)
     (lambda (l)
       (cond
         ((null? l) 0)
         (else
           (add1 (length (cdr l)))))))))
```

It is the function *length* in the style of chapter 9, using *Y*.

```
(define length
  (Y (lambda (length)
       (lambda (l)
         (cond
           ((null? l) 0)
           (else
             (add1 (length (cdr l)))))))))
```

And what is special about it?

We do not use (**define** ...) to make *length* recursive. Using *Y* on a function that looks like *length* creates the recursive function.

So is *Y* a special version of (**define** ...)

Yes, that's right. But we also agreed that the definition with (**define** ...) is easier to read than the definition with *Y*.

That's right. And we therefore have another way to write this kind of definition.

```
(define multirember
  (lambda (a lat)
    ((letrec[1]
       ((mr (lambda (lat)
              (cond
                ((null? lat) (quote ()))
                ((eq? a (car lat))
                 (mr (cdr lat)))
                (else
                  (cons (car lat)
                    (mr (cdr lat))))))))
       mr)
     lat)))
```

[1] L: (labels ((mr (lat) ...)) (function mr))

But if all we want is a recursive function *mr*, why don't we use this?

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

```
(define multirember
  (lambda (a lat)
    (mr lat)))
```

Because (**define** ...) does not work here.

Why not?

The definition of *mr* refers to *a* which stands for the atom that *multirember* needs to remove from *lat*

Okay, that's true, though obviously *a* refers to the first name in the definition of the function *multirember*.

Do you remember that names don't matter?

Yes, we said in chapter 9 that all names are equal. We can even change the names, as long as we do it consistently.

Correct. If we don't like *lat*, we can use *a-lat* in the definition of *multirember* as long as we also re-name all occurrences of *lat* in the body of the (**lambda** ...).

Yes, we could have used the following definition and nothing would have changed:

```
(define multirember
  (lambda (a a-lat)
    (mr a-lat)))
```

Correct again. And this means we should also be able to use *b* instead of *a* because

```
(define id
  (lambda (a)
    a))
```

is the same as

```
(define id
  (lambda (b)
    b))
```

Yet if we used *b* in the definition of *multirember*

```
(define multirember
  (lambda (b a-lat)
    (mr a-lat)))
```

the *a* in *mr* would no longer make any sense.

Yes: the name *a* makes sense only inside the definition of *multirember*. In general, the names for a function's arguments make sense only inside of (**lambda** ...).

Okay, that explains things.

And that is precisely why we need (**letrec** ...) What do you think is the purpose of the nested box?

It separates the two parts of a (**letrec** ...): the naming part, which is the nested box, and the value part, which is *mr*.

Is the nested box important otherwise?

No, the nested box is merely an annotation that we use to help distinguish the two parts of (**letrec** ...). Once we get accustomed to the shape of (**letrec** ...), we will stop drawing the inner box.

What do we use the naming part for?	The naming part defines a recursive function though unlike **define**d functions; a function defined in the naming part of (**letrec** ...) knows all the arguments of all the surrounding (**lambda** ...) expressions.
And the value part?	It tells us what the result of the (**letrec** ...) is. It may refer to the named recursive function.
Does this mean that (**letrec** ((*mr* ...)) *mr*) defines and returns a recursive function?	Precisely. Isn't that a lot of parentheses for saying just that?
Yes, but they are important.	Okay, let's go on.
What is the value of ((**letrec** ((*mr* ...)) *mr*) *lat*)	It is the result of applying the recursive function *mr* to *lat*.
What is the value of (*multirember a lat*) where *a* is pie and *lat* is (apple custard pie linzer pie torte)	(apple custard linzer torte), but we already knew this.
How can we determine this value?	That's more interesting.
The first line in the definition of *multirember* is no longer (**cond** ...) but ((**letrec** ((*mr* ...)) *mr*) *lat*) What does this mean?	We said that it defines the recursive function *mr* and applies it to *lat*.
What is the first line in *mr*	It is something we are quite familiar with: (**cond** ...). We now ask questions the way we did in chapter 2.

What is the first question?

(*null? lat*), which is false.

And the next question?

(*eq?* (*car lat*) *a*), which is false.

Why?

Because *a* still stands for pie, and (*car lat*) is apple.

That's correct: *mr* always knows about *a* which doesn't change while we look through *lat*

Yes.

Is it as if *multirember* had defined a function mr_{pie} and had used it on *lat*

```
(define mr_pie
  (lambda (lat)
    (cond
      ((null? lat) (quote ()))
      ((eq? (car lat) (quote pie))
       (mr_pie (cdr lat)))
      (else (cons (car lat)
              (mr_pie (cdr lat)))))))
```

Correct, and the good thing is that no other function can refer to mr_{pie}.

Why is **define** underlined?

We use (**define** ...) to express that the underlined definition does not actually exist, but imagining it helps our understanding.

Is it all clear now?

This is easy as apple pie.

Would it make any difference if we changed the definition a little bit more like this?

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

The difference between this and the previous definition isn't that big.
(Look at the third and last lines.)

The first line in (**lambda** (*a lat*) ...) is now of the shape
(**letrec** ((*mr* ...)) (*mr lat*))

Yes, so *multirember* first defines the recursive function *mr* that knows about *a*.

And then?

The value part of (**letrec** ...) uses *mr* on *lat*, so from here things proceed as before.

That's correct. Isn't (**letrec** ...) easy as pie?

We prefer (linzer torte).

Is it clear now what (**letrec** ...) does?

Yes, and it is better than *Y*.

The Twelfth Commandment

Use (letrec ...) to remove arguments that do not change for recursive applications.

How does *rember* relate to *multirember*

The function *rember* removes the first occurrence of some given atom in a list of atoms; *multirember* removes all occurrences.

Can *rember* also remove numbers from a list of numbers or S-expressions from a list of S-expressions?

Not really, but in *The Little Schemer* we defined the function *rember-f*, which given the right argument could create those functions:

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

Give a name to the function returned by
  (*rember-f* *test?*)
where
  test? is *eq?*

```
(define rember-eq? (rember-f test?))
```

where
  test? is *eq?*.

Is *rember-eq?* really *rember*

It is, but hold on tight; we will see more of this in a moment.

Can you define the function *multirember-f* which relates to *multirember* in the same way *rember-f* relates to *rember*

That is not difficult:

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

Explain in your words what *multirember-f* does.

Here are ours:
"The function *multirember-f* accepts a function *test?* and returns a new function. Let us call this latter function *m-f*. The function *m-f* takes an atom *a* and a list of atoms *lat* and traverses the latter. Any atom *b* in *lat* for which (*test? b a*) is true, is removed."

Is it true that during this traversal the result of (*multirember-f test?*) is always the same?

Yes, it is always the function for which we just used the name *m-f*.

Perhaps *multirember-f* should name it *m-f*

Could we use (**letrec** ...) for this purpose?

Yes, we could define *multirember-f* with (**letrec** ...) so that we don't need to re-determine the value of
(*multirember-f test?*)

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

Is this a new use of (**letrec** ...)?

No, it still just defines a recursive function and returns it.

True enough.

What is the value of (*multirember-f test?*) where
test? is *eq?*

It is the function *multirember*:

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

Did you notice that no (**lambda** ...) surrounds the (**letrec** ...)

It looks odd, but it is correct!

Could we have used another name for the function named in (**letrec** ...)

Yes, *mr* is *multirember*.

Is this another way of writing the definition?

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

Yes, this defines the same function.

Since (**letrec** ...) defines a recursive function and since (**define** ...) pairs up names with values, we could eliminate (**letrec** ...) here, right?

Yes, we could and we would get back our old friend *multirember*.

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

Here is *member?* again:

```
(define member?
  (lambda (a lat)
    (cond
      ((null? lat) #f)
      ((eq? (car lat) a) #t)
      (else (member? a (cdr lat))))))
```

So?

What is the value of (*member? a lat*) where *a* is ice and *lat* is (salad greens with pears brie cheese frozen yogurt)

#f, ice cream is good, too.

Is it true that *a* remains the same for all natural recursions while we determine this value?

Yes, *a* is always ice. Should we use The Twelfth Commandment?

Yes, here is one way of using (**letrec** ...) with this function:

```
(define member?
  (lambda (a lat)
    ((letrec
       ((yes? (lambda (l)
                (cond
                  ((null? l) #f)
                  ((eq? (car l) a) #t)
                  (else (yes? (cdr l)))))))
       yes?)
     lat)))
```

Do you also like this version?

Here is an alternative:

```
(define member?
  (lambda (a lat)
    (letrec
      ((yes? (lambda (l)
               (cond
                 ((null? l) #f)
                 ((eq? (car l) a) #t)
                 (else (yes? (cdr l)))))))
      (yes? lat))))
```

Did you notice that we no longer use nested boxes for (**letrec** ...)

Yes. We are now used to the shape of (**letrec** ...) and won't confuse the naming part with the value part anymore.

Do these lists represent sets?
(tomatoes and macaroni)
(macaroni and cheese)

Yes, they are sets because no atom occurs twice in these lists.

Do you remember what (*union set1 set2*) is where
set1 is (tomatoes and macaroni casserole)
and
set2 is (macaroni and cheese)

(tomatoes casserole macaroni and cheese).

Write *union*

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

Is it true that the value of *set2* always stays the same when determining the value of (*union set1 set2*)

Yes,
because *union* is like *rember* and *member?* in that it takes two arguments but only changes one when recurring.

Is it true that we can rewrite *union* in the same way as we rewrote *rember*

Yes, and it is easy now.

```
(define union
  (lambda (set1 set2)
    (letrec
      ((U (lambda (set)
              (cond
                ((null? set) set2)
                ((member? (car set) set2)
                 (U (cdr set)))
                (else (cons (car set)
                            (U (cdr set))))))))
      (U set1))))
```

Could we also have written it like this?

```
(define union
  (lambda (set1 set2)
    (letrec
      ((A (lambda (set)
              (cond
                ((null? set) set2)
                ((member? (car set) set2)
                 (A (cdr set)))
                (else (cons (car set)
                            (A (cdr set))))))))
      (A set1))))
```

Yes.

Correct: *A* is just a name like *U*
Does it matter what name we use?

Absolutely not, but choose names that matter to you and everyone else who wants to enjoy your definitions.

So why do we choose the name *U*

To keep the boxes from getting too wide, we use single letter names within (**letrec** ...) for such minor functions.

Can you think of a better name for *U*

This should be an old shoe by now.

Now, does it work?

It should.

Explain in your words how the new version of *union* works.

Our words:
"First, we define another function *U* that *cdr*s down *set*, *cons*ing up all elements that are not a member of *set2*. Eventually *U* will *cons* all these elements onto *set2*. Second, *union* applies *U* to *set1*."

How does *U* know about *set2*

Since *U* is defined using (**letrec** ...) inside of *union*, it knows about all the things that *union* knows about.

And does it have to pass around *set2*

No, it does not.

How does *U* know about *member?*

Everyone knows the function *member?*.

Does it mean that the definition of *union* depends on the definition of *member?*

It does, but *member?* works, so this is no problem.

Suppose we had defined *member?* as follows.

```
(define member?
  (lambda (lat a)
    (cond
      ((null? lat) #f)
      ((eq? (car lat) a) #t)
      (else (member? (cdr lat) a)))))
```

But this would confuse *union*!

Why?

Because this *member?* takes its arguments in a different order.

What changed?	Now *member?* takes a list first and an atom second.
Does *member?* work?	It works in that we can still use this new definition of *member?* to find out whether or not some atom is in a list.
But?	With this new definition, *union* will no longer work.
Oh?	Yes, because *union* assumes that *member?* takes its arguments in a certain order.
Perhaps we should avoid this.	How?
Well, (**letrec** ...) can define more than just a single function.	Nobody said so.
Didn't you notice the extra pair of parentheses around the function definitions in (**letrec** ...)	Yes.
With (**letrec** ...) we can define more than just one function by putting more than one function definition between the extra pair of parentheses.	This could help with *union*.

Here is a skeleton:

```
(define union
  (lambda (set1 set2)
    (letrec
      ...
      (U set1))))
```

Fill in the dots.

```
...
((U (lambda (set)
      (cond
        ((null? set) set2)
        ((member? (car set) set2)
         (U (cdr set)))
        (else (cons (car set)
                (U (cdr set)))))))
 (member?
   (lambda (a lat)
     (cond
       ((null? lat) #f)
       ((eq? (car lat) a) #t)
       (else (member? a (cdr lat)))))))
...
```

The Thirteenth Commandment

Use (letrec ...) to hide and to protect functions.

Could we also have written this?

```
(define union
  (lambda (set1 set2)
    (letrec
      ((U (lambda (set)
            (cond
              ((null? set) set2)
              ((M? (car set) set2)
               (U (cdr set)))
              (else (cons (car set)
                      (U (cdr set)))))))
       (M? (lambda (a lat)
             (cond
               ((null? lat) #f)
               ((eq? (car lat) a) #t)
               (else
                 (M? a (cdr lat)))))))
      (U set1))))
```

Presumably.

Are we happy now?

Well, almost.

Almost?

The definition of *member?* inside of *union* ignores The Twelfth Commandment.

It does?

Yes, the recursive call to *member?* passes along the parameter *a*.

And its value does not change?

No, it doesn't!

So we can write something like this?

```
(define union
  (lambda (set1 set2)
    (letrec
      ((U (lambda (set)
            (cond
              ((null? set) set2)
              ((M? (car set) set2)
               (U (cdr set)))
              (else (cons (car set)
                      (U (cdr set)))))))
       (M? ...))
      (U set1))))
```

Yes, and here is how we fill in the dots:

```
...
(lambda (a lat)
  (letrec
    ((N? (lambda (lat)
           (cond
             ((null? lat) #f)
             ((eq? (car lat) a) #t)
             (else (N? (cdr lat)))))))
    (N? lat)))
...
```

Now we are happy, right?

Yes!

Did you notice that *set2* is not an argument of *U*

It doesn't have to be because *union* knows about *set2* and *U* is inside of *union*.

Do we know enough about *union* now?

Yes, we do!

Do we deserve a break now?

We deserve dinner or something equally substantial.

True, but hold the dessert.

Why?

We need to protect a few more functions.

Which ones?

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

Sure, it is the function that checks whether some atom occurs twice in a row in some list. It is a perfect candidate for protection.

Yes, it is. Can you explain why?

Here are our words:
"Auxiliary functions like *two-in-a-row-b?* are always used on specific values that make sense for the functions we want to define. To make sure that these minor functions always receive the correct values, we hide such functions where they belong."

So how do we hide *two-in-a-row-b?*

The same way we hide other functions:

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

Does the minor function *W* need to know the argument *lat* of *two-in-a-row?*

No, *W* also takes *lat* as an argument.

Is it then okay to hide *two-in-a-row-b?* like this:

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

Yes, it is a perfectly safe way to protect the minor function *W*. It is still not visible to anybody but *two-in-a-row?* and works perfectly.

Good, let's look at another pair of functions.

Let's guess: it's *sum-of-prefixes-b* and *sum-of-prefixes*.

Protect *sum-of-prefixes-b*

```
(define sum-of-prefixes
  (lambda (tup)
    (letrec
      ((S (lambda (sss tup)
            (cond
              ((null? tup) (quote ()))
              (else
                (cons (+ sss (car tup))
                  (S (+ sss (car tup))
                     (cdr tup))))))))
      (S 0 tup))))
```

Is *S* similar to *W* in that it does not rely on *sum-of-prefixes*'s argument?

It is. We can also hide it without putting it inside (**lambda** ...) but we don't need to practice that anymore.

We should also protect *scramble-b*. Here is the skeleton:

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

Fill in the dots.

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

Can we define *scramble* using the following skeleton?

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

Yes, but can't this wait?

Yes, it can. Now it *is* time for dessert.

How about black currant sorbet?

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 *intersects* 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 *hops* 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.

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)
    (try¹ 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.

15.
The Difference Between Men and Boys....

What is the value of

```
(define x
  (cons (quote chicago)
    (cons (quote pizza)
      (quote ()))))
```

The definitions we have seen so far don't have values. But from now on we will sometimes have to talk about the values of definitions, too.

What does the name *x* refer to?

(chicago pizza).

What is the value of

```
(set!¹ x (quote gone))
```

It doesn't have a value, but the effect is as if we had just written:

```
(define x (quote gone))
```

[1] L: **setq**, pronounced "set queue"
S: Pronounced "set bang."

Did you notice that **define** is underlined?

We have seen it before. It means that we never actually write this definition. We merely imagine it. But it does replace the two boxes on the left.

What does the name *x* refer to?

gone.

What is the value of

```
(set! x (quote skins))
```

Remember this doesn't have a value.

Is (**set!** ...) just like (**define** ...)

Yes, mostly.
A (**set!** ...) expression always looks like (**define** ...). The second item is always a name, the last one is always an expression.

And what is *x* now?

It refers to skins.

What is the value of (*gourmet y*)
where *y* is onion
and
gourmet is

```
(define gourmet
  (lambda (food)
    (cons food
      (cons x (quote ())))))
```

Which *x* do you want?

Now what does *x* refer to?

It still refers to skins.

So what is the value of (*cons x* (**quote** ()))

(skins).

What is the value of
(*gourmet* (**quote** onion))

(onion skins).

```
(set! x (quote rings))
```

It is as if we had written:

```
(define x (quote rings))
```

and as if we had never had any definition of *x* before.

What is the value of (*gourmet y*)
where *y* is onion

Which value of *x* do you want?

And now, what is *x*

It refers to rings.

What is the value of
(*gourmet* (**quote** onion))

It is (onion rings), since *x* is now rings.

Look at this:

```
(define gourmand
  (lambda (food)
    (set! x food)
    (cons food
      (cons x
        (quote ())))))
```

What about it?

Is anything unusual?

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

What are they?

The first one is
(**set!** *x* *food*).
The one after that is
(*cons* *food*
  (*cons* *x*
    (**quote** ()))).

Have we seen something like this before?

Yes, we just saw a (**let** ...) with two expressions in the value part at the end of the previous chapter.

So what do you think is the value of
(*gourmand* (**quote** potato))

It is probably the value of the second expression, just as in a (**let** ...) with two expressions.

And that is?

A good guess is (potato potato).

That is correct!

It also means that the value of *x* is potato.

Yes! And how did that happen?

The first expression
  (**set!** x *food*)
means that the definition of x changed. It is as if we had written:

```
(define x (quote potato))
```

and as if we had never had any definition of x before.

Why?

Because *food* is potato.

What is the value of x now?

It is still potato.

What is the value of (*gourmand* w) where w is rice

Now it is easy: (rice rice).

And what is the value of x now?

rice, of course.

Does *gourmand* remember what food it saw last?

Yes, x always refers to the last food that *gourmand* ate.

Can you write *dinerR* which is like *diner* but also remembers which food it ate last?

```
(define diner
  (lambda (food)
    (cons (quote milkshake)
      (cons food
        (quote ())))))
```

No problem. We can use the same trick.

```
(define dinerR
  (lambda (food)
    (set! x food)
    (cons (quote milkshake)
      (cons food
        (quote ())))))
```

What is the value of (*dinerR* (**quote** onion))

(milkshake onion).

What does x refer to now?

onion.

What is the value of (*dinerR* (**quote** pecanpie))	(milkshake pecanpie).
And now what does *x* refer to?	pecanpie.
Which do you prefer?	Milkshake and pecan pie.
What is the value of (*gourmand* (**quote** onion))	We have done this before: (onion onion).
But, what happened to *x*	It now refers to onion.
What food did *dinerR* eat last?	Not onion.
How did that happen?	Both *dinerR* and *gourmand* use *x* to remember the food they saw last.
Should we have chosen a different name when we wrote *dinerR*	Yes, we should have chosen a new name.
Like what?	*y*.
But what would have happened if *gourmand* had used *y* to remember the food it saw last?	Well, wouldn't we have the same problem again?
Yes, but don't worry: there is a way to avoid this conflict of names.	There must be, because we should be able to get around such coincidences!
Here is a new function:	It looks like *gourmand*.

```
(define omnivore
  (let ((x (quote minestrone)))
    (lambda (food)
      (set! x food)
      (cons food
        (cons x
          (quote ()))))))
```

True, but not quite. What is the big difference?

Didn't you see the **(let ...)** that surrounds the **(lambda ...)**? Here it is:

```
(let ((x (quote minestrone)))
  (lambda (food)
    ...)).
```

What is the little difference?

The names.

What is the value of

```
(define omnivore
  (let ((x (quote minestrone)))
    (lambda (food)
      (set! x food)
      (cons food
        (cons x
          (quote ()))))))
```

We learned that **(let ...)** names the value of expressions.

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

minestrone.

And what is the value part of the **(let ...)**

The value part of this **(let ...)** is a function.

What value does *omnivore* stand for?

We do not know.

That is correct. We need to determine its value.

We have never done this before.

So the definition of *omnivore* is almost like writing two definitions:

```
(define x (quote minestrone))
```

```
(define omnivore
  (lambda (food)
    (set! x food)
    (cons food
      (cons x
        (quote ())))))
```

But it really is this:

```
(define x1 (quote minestrone))
```

```
(define omnivore
  (lambda (food)
    (set! x1 food)
    (cons food
      (cons x1
        (quote ())))))
```

Did you notice that **define** is underlined?	Yes, that's old hat by now.
Did you see the underlined name?	Yes, and that is something new.
What is $\underline{x}_1$	$\underline{x}_1$ is an imaginary name.
Has $\underline{x}_1$ ever been used before with (<u>**define**</u> ...)	No, it has not. And it never, ever will be used with (<u>**define**</u> ...) again.
What does $\underline{x}_1$ refer to?	It stands for minestrone.
So, what is $\underline{x}_1$'s value?	No answer; it is imaginary.
What is the value of *omnivore*	Now it is a function.
What is the value of (*omnivore* z) where z is bouillabaisse	It looks like it is (bouillabaisse bouillabaisse).
What is $\underline{x}_1$'s value?	No answer.
Right?	Always no answer for imaginary names. We just keep in mind what they represent.
What does $\underline{x}_1$ refer to?	It now stands for bouillabaisse.
And why?	After determining the value of (*omnivore* z) where z is bouillabaisse, $\underline{x}_1$ has changed. It is as if we had written: (<u>**define**</u> $\underline{x}_1$ (**quote** bouillabaisse)) and as if we had never had a definition of $\underline{x}_1$ before.

Determining the value of (*omnivore z*) is just like finding the value of (*gourmand z*)

What is the difference?

There is no answer for $\underline{x}_1$

Unlike x, $\underline{x}_1$ is an imaginary name. We must remember what value it represents, because we cannot find out!

The Sixteenth Commandment

Use (set! ...) only with names defined in (let ...)s.

Take a really close look at this:

```
(define gobbler
  (let ((x (quote minestrone)))
    (lambda (food)
      (set! x food)
      (cons food
        (cons x
          (quote ()))))))
```

This looks like *omnivore*.

Not quite. What is the little difference?

The names.

Is there a big difference?

No!

What is the value of

```
(define gobbler
  (let ((x (quote minestrone)))
    (lambda (food)
      (set! x food)
      (cons food
        (cons x
          (quote ()))))))
```

```
(define x2 (quote minestrone))
```

```
(define gobbler
  (lambda (food)
    (set! x2 food)
    (cons food
      (cons x2
        (quote ())))))
```

What is $\underline{x}_2$	$\underline{x}_2$ is another imaginary name.
Has $\underline{x}_2$ ever been used before with (**define** ...)	No, and it never, ever will be used with (**define** ...) again.
What does $\underline{x}_2$ refer to?	It stands for minestrone.
What does $\underline{x}_1$ refer to?	It still stands for bouillabaisse.
So, what is $\underline{x}_2$'s value?	No answer, because $\underline{x}_2$ is imaginary.
What is the value of *gobbler*	It is a function.
What is the value of (*gobbler* *z*) where *z* is gumbo	It is (gumbo gumbo).
Now, what is $\underline{x}_2$'s value?	No answer. Ever!
What does $\underline{x}_2$ refer to?	It now stands for gumbo.
And why?	After determining the value of the definition, the definition of $\underline{x}_2$ has changed. It is as if we had written: (**define** $\underline{x}_2$ (**quote** gumbo)) and as if we had never had a value for $\underline{x}_2$ before.
Determining the value of (*gobbler* *z*) is just like finding the value of (*omnivore* *z*)	What is the difference?

Here is the function *glutton*

```
(define food (quote none))
```

```
(define glutton
  (lambda (x)
    (set! food x)
    (cons (quote more)
      (cons x
        (cons (quote more)
          (cons x
            (quote ())))))))
```

Explain in your words what it does.

As you know, we use our words: "When given a food item, say onion, it builds a list that demands a double portion of this item,

(more onion more onion)

in our example, and also remembers the food item in *food*."

Why does the definition of *glutton* disobey The Seventeenth Commandment?

Recall that we occasionally ignore commandments, because it helps to explain things.

What is the value of
(*glutton* (**quote** garlic))

(more garlic more garlic).

What does *food* refer to

garlic.

Do you remember what x refers to?

onion. In case you forgot, x refers to what *gourmand* or *dinerR* ate last.

Who saw the onion

gourmand.

Can you write the function *chez-nous*, which swaps what x and *food* refer to?

If so, have a snack and join us later for the main meal.

How can *chez-nous* change *food* to what x refers to?

(**set!** *food* x).

How can the function change *x* to what *food* refers to?

(**set!** *x* *food*).

How many arguments does *chez-nous* take?

None!

Is this the right way of putting it all together in one definition?

```
(define chez-nous
  (lambda ()
    (set! food x)
    (set! x food)))
```

It is worth a try, but we should check whether it works.

What does *food* refer to?

garlic.

What does *x* refer to?

onion.

What is the value of (*chez-nous*)

Now, what does *food* refer to

onion.

Now, what does *x* refer to?

onion.

Did you look closely at the last answer?

We hope so.

Why is the value of *x* still onion

After changing *food* to the value that *x* stands for, *chez-nous* changes *x* to what *food* refers to.

And what does *food* refer to?

onion.

The Eighteenth Commandment

Use (set! *x* ...) only when the value that *x* refers to is no longer needed.

How could we save the value in *food* so that it is still around when we need to change *x*

With (**let** ...).

Explain!

Here is our attempt:
"(**let** ...) names values. If *chez-nous* first names the value in *food*, we have two ways to refer to its value. And we can use the name in (**let** ...) to put this value into *x*."

Like this?

```
(define chez-nous
  (lambda ()
    (let ((a food))
      (set! food x)
      (set! x a))))
```

Yes, exactly like that.

What is the value of
(*glutton* (**quote** garlic))

(more garlic more garlic).

What does *food* refer to?

garlic.

What is the value of
(*gourmand* (**quote** potato))

(potato potato).

What does *x* refer to?

potato.

What is the value of (*chez-nous*)

And *food* refers to ...	potato.
But this time, x refers to ...	garlic.
See you later!	Bye for now.
Don't you want anything to eat?	No, that was enough garlic for one day.
If you want something full of garlic, try skordalia.	Perhaps someday.

SKORDALIA

To make 3 cups:

6 cloves to 1 head garlic, peeled

2 cups mashed potatoes (approximately 4 medium potatoes)

4 or more large slices of French- or Italian-type bread,
crusts removed, soaked in water, and squeezed dry

1/2 to 3/4 cup olive oil

1/3 to 1/2 cup white vinegar

Pinch of salt

Pound the garlic cloves in a large wooden mortar with a pestle until thoroughly mashed. Continue pounding while adding the potatoes and bread very gradually, beating until the mixture resembles a paste. Slowly add the oil, alternating with the vinegar, beating thoroughly after each addition until well absorbed. Add salt, taste for seasoning, and beat until the sauce is very thick and smooth, adding more vinegar or soaked squeezed bread, if necessary. Then scoop into a serving bowl. Cover and refrigerate until ready to use. Use as a dip for beets, zucchini, and eggplant.

THE FOOD OF GREECE
Vilma Liacours Chentiles
Avenel Books, New York, 1975

Here are *sweet-tooth* and *last*

```
(define sweet-tooth
  (lambda (food)
    (cons food
      (cons (quote cake)
        (quote ())))))
```

```
(define last (quote angelfood))
```

More food: did you exercise after your snack?

What is the value of (*sweet-tooth* *x*) where *x* is chocolate

(chocolate cake).

What does *last* refer to?

angelfood.

What is the value of (*sweet-tooth* *x*) where *x* is fruit

(fruit cake).

Now, what does *last* refer to?

Still angelfood.

Can you write the function *sweet-toothL* which returns the same value as *sweet-tooth* and which, in addition, changes *last* so that it refers to the last *food* that *sweet-toothL* has seen?

We have used this trick twice before. Here we go:

```
(define sweet-toothL
  (lambda (food)
    (set! last food)
    (cons food
      (cons (quote cake)
        (quote ())))))
```

What is the value of
(*sweet-toothL* (**quote** chocolate))

(chocolate cake).

And the value of *last* is ...

chocolate.

What is the value of (*sweet-toothL* (**quote** fruit))	(fruit cake).
And *last*	It refers to fruit.
Isn't this easy?	Easy as pie!
Find the value of (*sweet-toothL* x) where x is cheese	It is (cheese cake).
What is the value of (*sweet-toothL* (**quote** carrot))	(carrot cake).
Do you still remember the ingredients that went into *sweet-toothL*	There was chocolate, fruit, cheese, and carrot.
How did you put this list together?	By quickly glancing over the last few questions and answers.
But couldn't you just as easily have memorized the list as you were reading the questions?	Of course, but why?
Can you write a function *sweet-toothR* that returns the same results as *sweet-toothL* but also memorizes the list of ingredients as they are passed to the function?	Yes, you can. Here's a hint. (**define** *ingredients* (**quote** ()))
What is that hint about?	This is the name that refers to the list of ingredients that *sweet-toothR* has seen.
One more hint: The Second Commandment.	Is this the commandment about using *cons* to build lists?

Did we forget about The Sixteenth Commandment?

Sometimes it is easier to explain things when we ignore the commandments. We will use names introduced by (**let** ...) next time we use (**set!** ...).

What is the value of (*deep* 3)

No, it is not a pizza. It is (((pizza))).

What is the value of (*deep* 7)

Don't get the pizza yet. But, yes, it is (((((((pizza))))))).

What is the value of (*deep* 0)

Let's guess: pizza.

Good guess.

This is easy: no toppings, plain pizza.

Is this *deep*

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

It would give the right answers.

Do you remember the value of (*deep* 3)

It is (((pizza))), isn't it?

How did you determine the answer?

Well, *deep* checks whether its argument is 0, which it is not, and then it recurs.

Did you have to go through all of this to determine the answer?

No, the answer is easy to remember.

Is it easy to write the function *deepR* which returns the same answers as *deep* but remembers all the numbers it has seen?

This is trivial by now:

```
(define Ns (quote ()))
```

```
(define deepR
  (lambda (n)
    (set! Ns (cons n Ns))
    (deep n)))
```

Great! Can we also extend *deepR* to remember all the results?

This should be easy, too:

```
(define Rs (quote ()))
```

```
(define Ns (quote ()))
```

```
(define deepR
  (lambda (n)
    (set! Rs (cons (deep n) Rs))
    (set! Ns (cons n Ns))
    (deep n)))
```

Wait! Did we forget a commandment?

The Fifteenth: we say (*deep* *n*) twice.

Then rewrite it.

```
(define deepR
  (lambda (n)
    (let ((result (deep n)))
      (set! Rs (cons result Rs))
      (set! Ns (cons n Ns))
      result)))
```

Does it work?

Let's see.

What is the value of (*deepR* 3)

(((pizza))).

What does *Ns* refer to?	(3).
And *Rs*	((((pizza)))).
Let's do this again. What is the value of (*deepR* 5)	(((((pizza))))).
Ns refers to ...	(5 3).
And *Rs* to ...	((((((pizza))))) (((pizza)))).

The Nineteenth Commandment

Use (set! ...) to remember valuable things between two distinct uses of a function.

Do it again with 3	But we just did. It is (((pizza))).
Now, what does *Ns* refer to?	(3 5 3).
How about *Rs*	((((pizza))) (((((pizza))))) (((pizza)))).
We didn't have to do this, did we?	No, we already knew the result. And we could have just looked inside *Ns* and *Rs*, if we really couldn't remember it.

How should we have done this?

Ns contains 3. So we could have found the value (((pizza))) without using *deep*.

Where do we find (((pizza)))

In *Rs*.

What is the value of (*find* 3 *Ns Rs*)

(((pizza))).

What is the value of (*find* 5 *Ns Rs*)

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

What is the value of (*find* 7 *Ns Rs*)

No answer, since 7 does not occur in *Ns*.

Write the function *find*
In addition to *Ns* and *Rs* it takes a number *n* which is guaranteed to occur in *Ns* and returns the value in the corresponding position of *Rs*

```
(define find
  (lambda (n Ns Rs)
    (letrec
      ((A (lambda (ns rs)
             (cond
               ((= (car ns) n) (car rs))
               (else
                 (A (cdr ns) (cdr rs)))))))
      (A Ns Rs))))
```

We are happy to see that you are truly comfortable with (**letrec** ...)

No problem.

Use *find* to write the function *deepM* which is like *deepR* but avoids unnecessary *cons*ing onto *Ns*

No problem, just use (**if** ...):

```
(define deepM
  (lambda (n)
    (if (member? n Ns)
      (find n Ns Rs)
      (deepR n))))
```

What is *Ns*

(3 5 3).

And *Rs*

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

Now that we have *deepM* should we remove the duplicates from *Ns* and *Rs*

How could we possibly do this?

You forgot: we have (**set!** ...)

```
(set! Ns (cdr Ns))
```

```
(set! Rs (cdr Rs))
```

What is *Ns* now?

(5 3).

And how about *Rs*

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

Is *deepM* simple enough?

Sure looks simple.

Do we need to waste the name *deepR*

No, the function *deepR* is not recursive.

And *deepR* is used in only one place.

That's correct.

So we can write *deepM* without using *deepR*

```
(define deepM
  (lambda (n)
    (if (member? n Ns)
        (find n Ns Rs)
        (let ((result (deep n)))
          (set! Rs (cons result Rs))
          (set! Ns (cons n Ns))
          result))))
```

This is another form of simplifying.

Which is why we did it after the function was correct.

If we now ask one more time what the value of (*deepM* 3) is

... then we use *find* to determine the result.

Ready? What is the value of (*deepM* 6)

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

Good, but how did we get there?

We used *deepM* and *deep*, which *cons*ed onto *Ns* and *Rs*.

But, isn't (*deep* 6) the same as (*cons* (*deep* 5) (**quote** ()))

What kind of question is this?

When we find (*deep* 6) we also determine the value of (*deep* 5)

Which we can already find in *Rs*.

That's right.

Should we try to help *deep* by changing the recursion in *deep* from (*deep* (*sub1* *m*)) to (*deepM* (*sub1* *m*))?

Do it.

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

What is the value of (*deepM* 9)

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

What is *Ns* now?

(9 8 7 6 5 3).

Where did the 7 and 8 come from?

The function *deep* asks for (*deepM* 8).

And that is why 8 is in the list.

(*deepM* 8) requires the value of (*deepM* 7).

Is this it?

Yes, because (*deepM* 6) already knows the answer.

Can we eat the pizza now?

No, because *deepM* still disobeys The Sixteenth Commandment.

That's true. The names in (**set!** *Ns* ...) and (**set!** *Rs* ...) are not introduced by (**let** ...)

It is easy to do that.

Here it is:

```
(define deepM
  (let ((Rs (quote ()))
        (Ns (quote ())))
    (lambda (n)
      (if (member? n Ns)
          (find n Ns Rs)
          (let ((result (deep n)))
            (set! Rs (cons result Rs))
            (set! Ns (cons n Ns))
            result)))))
```

What is the value of this definition?

Two imaginary names and *deepM*.

```
(define Rs1 (quote ()))
```

```
(define Ns1 (quote ()))
```

```
(define deepM
  (lambda (n)
    (if (member? n Ns1)
        (find n Ns1 Rs1)
        (let ((result (deep n)))
          (set! Rs1 (cons result Rs1))
          (set! Ns1 (cons n Ns1))
          result))))
```

What is the value of (*deepM* 16)

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

Why is #f a good answer in that case?

When *find* succeeds, it returns a list, and #f is an atom.

Can we now replace *member?* with *find* since the new version also handles the case when its second argument is empty?

Yes, that's no problem now. If the answer is #f, *Ns* does not contain the number we are looking for. And if the answer is a list, then it does.

Okay, then let's do it.

```
(define deepM
  (let ((Rs (quote ()))
        (Ns (quote ())))
    (lambda (n)
      (if (atom? (find n Ns RS))
          (let ((result (deep n)))
            (set! Rs (cons result Rs))
            (set! Ns (cons n Ns))
            result)
          (find n Ns Rs)))))
```

That's one way of doing it. But if we follow The Fifteenth Commandment, the function looks even better.

```
(define deepM
  (let ((Rs (quote ()))
        (Ns (quote ())))
    (lambda (n)
      (let ((exists (find n Ns RS)))
        (if (atom? exists)
            (let ((result (deep n)))
              (set! Rs (cons result Rs))
              (set! Ns (cons n Ns))
              result)
            exists)))))
```


Take a deep breath or a deep pizza, now.

Do you remember *length*

Sure:

```
(define length
  (lambda (l)
    (cond
      ((null? l) 0)
      (else (add1 (length (cdr l)))))))
```

Is this a good solution?

```
(define length
  (let ((h (lambda (l) 0)))
    (set! h
      (L (lambda (arg) (h arg))))
    h))
```

Yes, except that (**lambda** (*arg*) (*h arg*)) seems to be a long way of saying *h*.

Why can we write
(**lambda** (*arg*) (*h arg*))

Because *h* is a function of one argument.

Does *h* always refer to
(**lambda** (*l*) 0)

No, it is changed to the value of
(*L* (**lambda** (*arg*) (*h arg*))).

What is the value of
(**lambda** (*arg*) (*h arg*))

We don't know because it depends on *h*.

How many times does the value of *h* change?

Once.

What is the value of
(*L* (**lambda** (*arg*) (*h arg*)))

It is a function:

```
(lambda (l)
  (cond
    ((null? l) 0)
    (else (add1
            ((lambda (arg) (h arg))
             (cdr l)))))).
```

What is the value of

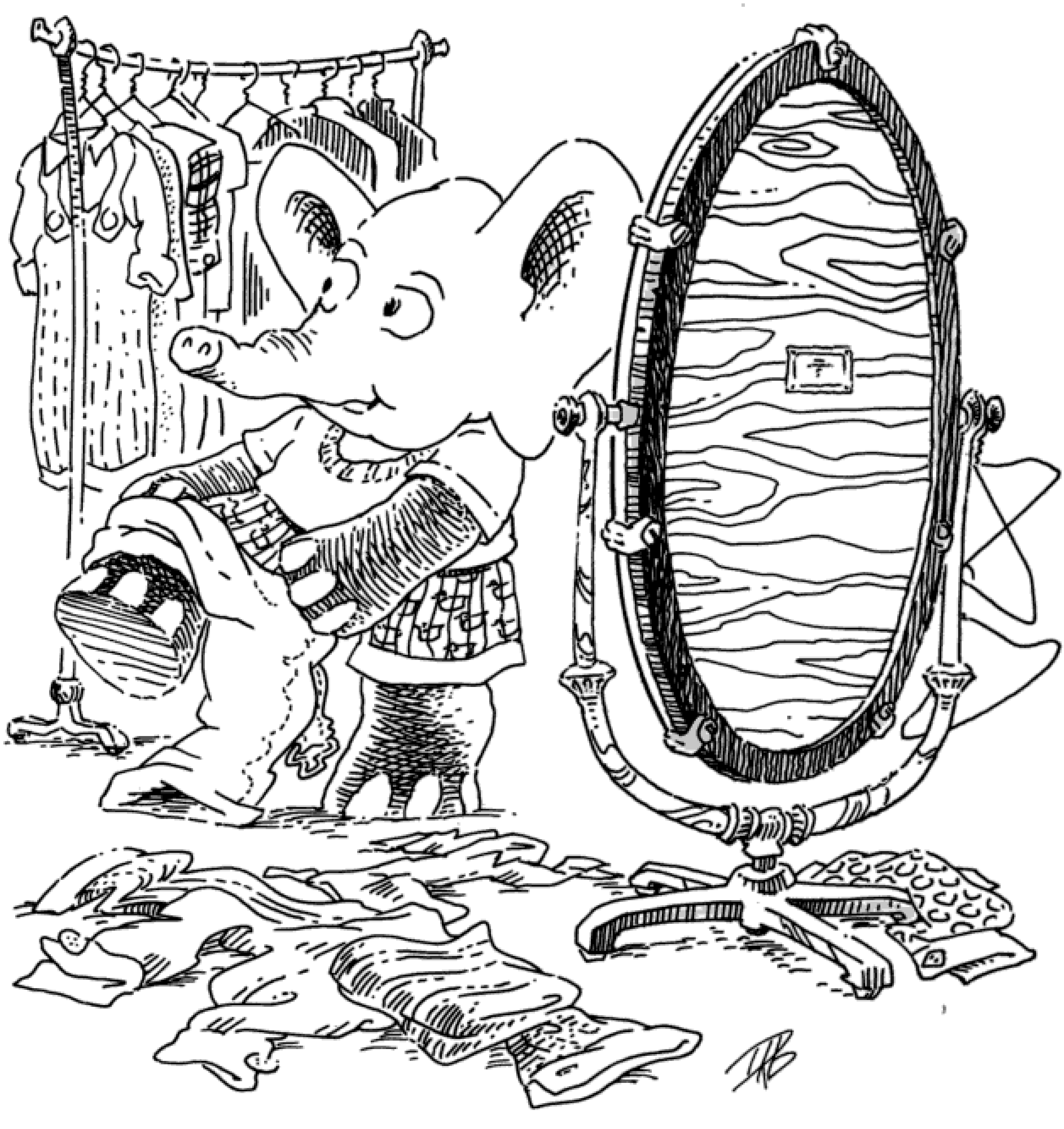
```
(lambda (l)
  (cond
    ((null? l) 0)
    (else (add1
            ((lambda (arg) (h arg))
             (cdr l))))))
```

We don't know because *h* changes. Indeed, it changes and becomes this function.

And then?

Then the value of *h* is the recursive function *length*.

17. We Change, Therefore We Are!



Is there a (**set!** *m* ...) in the value part of (**let** ((*m* *n*)) ...)

No. Are you asking whether we should unname again?

We could, couldn't we?

Yes, because now a name is replaced by a name.

Do it again!

```
(define deepM
  (let ((Rs (quote ()))
        (Ns (quote ())))
    (lambda (n)
      (let ((exists (find n Ns RS)))
        (if (atom? exists)
            (let ((result ...))
              (set! Rs (cons result Rs))
              (set! Ns (cons n Ns))
              result)
            exists)))))
```

```
...
(if (zero? n)
    (quote pizza)
    (cons (deepM (sub1 n))
      (quote ())))
...
```

Wouldn't you like to know how much help *deepM* gives?

What does that mean?

Once upon a time, we wrote *deepM* to remember what values *deep* had for given numbers.

Oh, yes.

How many *cons*es does *deep* use to build pizza

None.

How many *cons*es does *deep* use to build (((((pizza)))))

Five, one for each topping.

How many *cons*es does *deep* use to build (((pizza)))

Three.

How many *conses* does *deep* use to build pizza with a thousand toppings?

1000.

How many *conses* does *deep* use to build all possible pizzas with at most a thousand toppings?

That's a big number:
the *conses* of (*deep* 1000), and
the *conses* of (*deep* 999), and
. . . , and
the *conses* of (*deep* 0).

You mean 500,500?

Yes, thank you, Carl F. Gauss (1777–1855).

Yes, there is an easy way to determine this number, but we will show you the hard way. It is far more exciting.

Okay.

Guess what it is?

Can we write a function that determines it for us?

Yes, we can write the function *consC* which returns the same value as *cons* and counts how many times it sees arguments.

This is no different from writing *deepR* except that we use *add1* to build a number rather than *cons* to build a list.

```
(define consC
  (let ((N 0))
    (lambda (x y)
      (set! N (add1 N))
      (cons x y))))
```

Don't forget the imaginary name.

```
(define N1 0)
```

```
(define consC
  (lambda (x y)
    (set! N1 (add1 N1))
    (cons x y)))
```

Could we use this function to determine 500,500?

Sure, no problem.

How?

We just need to use *consC* instead of *cons* in the definition of *deep*:

```
(define deep
  (lambda (m)
    (if (zero? m)
        (quote pizza)
        (consC (deep (sub1 m))
          (quote ())))))
```

Wasn't this exciting?

Well, not really.

So let's see whether this new *deep* counts *conses*

How about determining the value of (*deep* 5)?

That is easy; we shouldn't bother. What is the value of $\underline{N}_1$

We don't know, it is imaginary.

But that's how we count *conses*

How could we possibly see something that is imaginary?

Here is one way.

```
(define counter)
```

```
(define consC
  (let ((N 0))
    (set! counter
      (lambda ()
        N))
    (lambda (x y)
      (set! N (add1 N))
      (cons x y))))
```

Is this as if we had written:

```
(define N_2 0)
```

```
(define counter
  (lambda ()
    N_2))
```

```
(define consC
  (lambda (x y)
    (set! N_2 (add1 N_2))
    (cons x y)))
```

But?

It changed $\underline{N}_3$ to 0.

What is the value of (*supercounter f*) where *f* is *deep*

500500.

Is this what we expected?

Yes!

It is time to see how many *cons*es are used for (*deepM* 5)

Don't we need to modify its definition so that it uses *consC*?

Of course! What are you waiting for?

```
(define deepM
  (let ((Rs (quote ()))
        (Ns (quote ())))
    (lambda (n)
      (let ((exists (find n Ns RS)))
        (if (atom? exists)
            (let ((result
                   (if (zero? n)
                       (quote pizza)
                       (consC
                         (deepM (sub1 n))
                         (quote ())))))
              (set! Rs (cons result Rs))
              (set! Ns (cons n Ns))
              result)
            exists)))))
```

How many *cons*es does *deepM* use to build (((((pizza)))))

Probably five?

What is the value of (*counter*)

500505.

Yes!

Yes, but it means we forgot to initialize with *set-counter*.

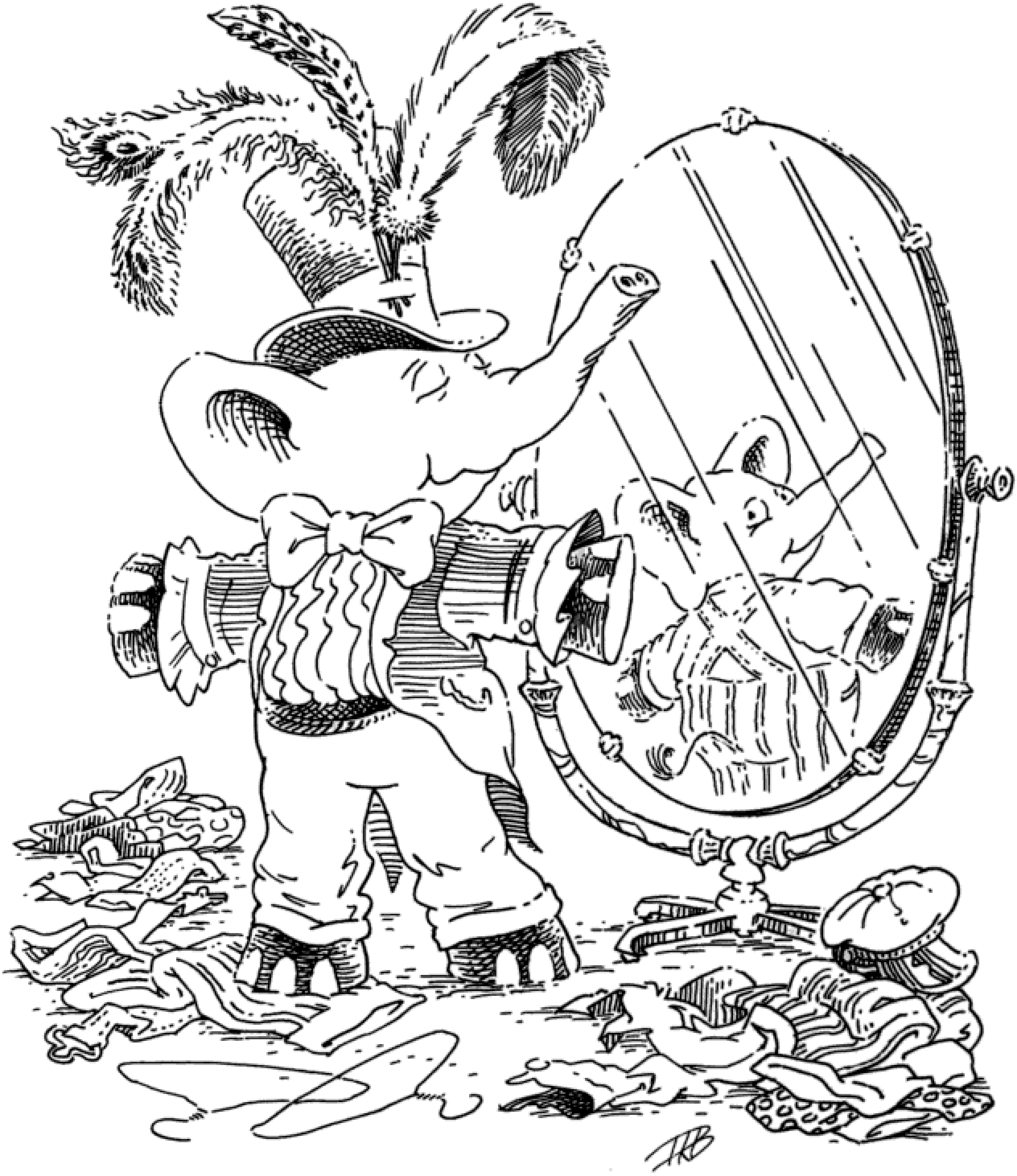
What is the value of (*set-counter* 0)	
How many *conses* does *deepM* use to build (((((pizza)))))	Five.
What is the value of (*counter*)	5.
What is the value of (*deep* 7)	(((((((pizza))))))).
What is the value of (*counter*)	Obvious: 7.
Didn't we need to *set-counter* to 0	No, we wanted to count the number of *conses* that were needed to build (*deepM* 5) and (*deepM* 7).
Why isn't this 12	Because that was the point of *deepM*.
What is (*supercounter f*) where *f* is *deepM*	Don't we need to initialize?
No. What is (*supercounter f*) where *f* is *deepM*	1000.
How many more *conses* does *deep* use to return the same value as *deepM*	499,500.
"A LISP programmer knows the value of everything but the cost of nothing."	Thank you, Alan J. Perlis (1922–1990).

But we know the value of food!

((((((((((((((((more pizza))))))))))))))))
(((((((((((((((more pizza)))))))))))))))
((((((((((((((more pizza))))))))))))))
(((((((((((((more pizza)))))))))))))
((((((((((((more pizza))))))))))))
(((((((((((more pizza)))))))))))
((((((((((more pizza))))))))))
(((((((((more pizza)))))))))
((((((((more pizza))))))))
(((((((more pizza)))))))
((((((more pizza))))))
(((((more pizza)))))
((((more pizza))))
(((more pizza)))
((more pizza))
(more pizza)
more pizza)

What is the value of (counter)	5.
What is the value of (*set-counter* 0)	
(*rember1*C2* *a* *l*) where *a* is noodles and *l* is ((food) more (food))	((food) more (food)), because this list does not contain noodles.
And what is the value of (*counter*)	5, because *rember1*C2* needs five *consC*s to rebuild the list ((food) more (food)).
What food are you in the mood for now?	Find a good restaurant that specializes in it and dine there tonight.

18. We Change, Therefore We Are the Same!



What is the value of (*lots* 3)

(egg egg egg).

What is the value of (*lots* 5)

(egg egg egg egg egg).

What is the value of (*lots* 12)

(egg egg egg egg egg egg
egg egg egg egg egg egg).

What is the value of (*lenkth* (*lots* 3))

3.

What is the value of (*lenkth* (*lots* 5))

5.

What is the value of (*lenkth* (*lots* 15))

15.

Here is *lots*

```
(define lots
  (lambda (m)
    (cond
      ((zero? m) (quote ()))
      (else (kons[1] (quote egg)
              (lots (sub1 m)))))))
```

[1] L, S: This is like cons.

And this is *lenkth*:

```
(define lenkth
  (lambda (l)
    (cond
      ((null? l) 0)
      (else (add1 (lenkth (kdr[1] l)))))))
```

[1] L, S: This is like cdr.

How can we create a list of four eggs from (*lots* 3)

How about (*kons* (**quote egg**) (*lots* 3))?

Can we add an egg at the other end of the list?

Of course we can.

```
(define add-at-end
  (lambda (l)
    (cond
      ((null? (kdr l))
       (konsC (kar[1] l)
         (kons (quote egg)
           (quote ()))))
      (else (konsC (kar l)
              (add-at-end (kdr l)))))))
```

[1] L, S: This is like car.

Why do we ask (*null?* (*kdr* *l*))

Because we promise not to use *add-at-end* with non-empty lists.

What is a non-empty list?

A non-empty list is always created with *kons*. Its tail may be the empty list though.

What is *konsC*

konsC is to *consC* what *kons* is to *cons*.

What is the value of (*add-at-end* (*lots* 3))

(egg egg egg egg).

How many *konsC*es did we use?

The value of (*kounter*) is 3.

Can we add an egg at the end without making any new *kons*es except for the last one?

That would be a surprise!

Here is one way.

Are there any others?

```
(define add-at-end-too
  (lambda (l)
    (letrec
      ((A (lambda (ls)
            (cond
              ((null? (kdr ls))
               (set-kdr[1] ls
                 (kons (quote egg)
                   (quote ()))))
              (else (A (kdr ls)))))))
      (A l)
      l)))
```

[1] L: This is like rplacd.
S: This is like set-cdr!.

Sure there are, but we are not interested in them.

Okay.

What is the value of (*set-kounter* 0)

What is the value of (*kounter*)

0.

What is the value of
(*add-at-end-too* (*lots* 3))

(egg egg egg egg).

How many *konsC*es did *add-at-end-too* use?

Can we count them?

What if we told you that the value of (*kounter*) is 0

That's what it should be because *add-at-end-too* never uses *konsC* so the value of (*kounter*) should not change.

Do you remember *cons*

It is magnificent.

Recall *zub1* *edd1* and *sero?* from *The Little Schemer*. We can approximate *cons* in a similar way:

```
(define kons
  (lambda (kar kdr)
    (lambda (selector)
      (selector kar kdr))))
```

Write *kar* and *kdr*

```
(define kar
  (lambda (c)
    (c (lambda (a d) a))))
```

```
(define kdr
  (lambda (c)
    (c (lambda (a d) d))))
```

Suppose we had given you the definition of *bons*

```
(define bons
  (lambda (kar)
    (let ((kdr (quote ())))
      (lambda (selector)
        (selector
          (lambda (x) (set! kdr x))
          kar
          kdr)))))
```

Write *kar* and *kdr*

They are not too different from the previous definitions of *kar* and *kdr*.

```
(define kar
  (lambda (c)
    (c (lambda (s a d) a))))
```

```
(define kdr
  (lambda (c)
    (c (lambda (s a d) d))))
```

How can *bons* act like *kons*

Are we about to find out?

What is the value of (*bons* *e*) where *e* is egg

It is a function that is almost like (*kons* *e* *f*) where *f* is the empty list.

What is different?

When we determine the value of (*bons* (**quote** egg)), we also make a new imaginary name, $\underline{kdr}_1$. And the value that this imaginary name refers to can change over time.

How can we change the value that $\underline{kdr}_1$ refers to?

We could write a function that is almost like *kar* or *kdr*. This function could use the function (**lambda** (*x*) (**set!** $\underline{kdr}_1$ *x*)).

What is a good name for this function?

A good name is *set-kdr* and here is its definition.

```
(define set-kdr
  (lambda (c x)
    ((c (lambda (s a d) s)) x)))
```

Can we use *set-kdr* and *bons* to define *kons*

It's a little tricky but *bons* creates *kons*-like things whose *kdr* can be changed with *set-kdr*.

Let's do it!

Okay, this should do it:

```
(define kons
  (lambda (a d)
    (let ((c (bons a)))
      (set-kdr c d)
      c)))
```

Is *kons* a shadow of *cons*

It is.

Is *kons* different from *cons*

It certainly is. But don't forget that chapter 6 said: Beware of shadows.

Did we make any *kons*es when we added an egg to the end of the list?

Only for the new egg.

What is the value of

```
(define dozen (lots 12))
```

To find out, we must determine the value of (*lots* 12).

How many *kons*es did we use?

12.

What is the value of

```
(define bakers-dozen (add-at-end dozen))
```

To find out, we must determine the value of (*add-at-end dozen*).

Does that mean that the *konses* in *bakers-dozen* are the same as the first twelve in *bakers-dozen-again*

Absolutely not!

Does that mean that the *konses* in *dozen* are still the same as the first twelve in *bakers-dozen-too*

It sure does!

What is the value of
(*eklist? bakers-dozen bakers-dozen-too*)
where

```
(define eklist?
  (lambda (ls1 ls2)
    (cond
      ((null? ls1) (null? ls2))
      ((null? ls2) #f)
      (else
        (and (eq? (kar ls1) (kar ls2))
          (eklist? (kdr ls1) (kdr ls2)))))))
```

#t.

What does "the same" mean?

That is a deep philosophical question. Thank you, Gottfried W. Leibniz (1646–1716).

There is a new idea of "sameness" once we introduce (**set!** ...)

And that is?

Two *konses* are the same if changing one changes the other.

What does that mean?

How can we change a *kons*

We defined *set-kdr* so that we could add a new egg at the end of the list *without* additional *konses*.

Suppose we changed the first *kons* in *dozen*. Would it cause a change in the first *kons* of *bakers-dozen*

No.

Suppose again we changed the first *kons* in *dozen*. Would it cause a change in the first *kons* of *bakers-dozen-too*

Yes!

Time to define this notion of same.

```
(define same?
  (lambda (c1 c2)
    (let ((t1 (kdr c1))
          (t2 (kdr c2)))
      (set-kdr c1 1)
      (set-kdr c2 2)
      (let ((v (= (kdr c1) (kdr c2))))
        (set-kdr c1 t1)
        (set-kdr c2 t2)
        v))))
```

Thank you, Gerald J. Sussman and Guy L. Steele Jr.

What is the value of
(*same? bakers-dozen bakers-dozen-too*)

#t.

Why?

The function *same?* temporarily changes the *kdr*s of two *kons*es. Then, if changing the second *kons* also affects the first *kons*, the two must be the same.

Could you explain this again?

If someone overate and you have a stomach ache, you are the one who ate too much.

How many imaginary names are used to determine the value of

```
(same?
  (kons (quote egg) (quote ()))
  (kons (quote egg) (quote ())))
```

Two. One for the first *kons* and one for the second.

What is its value?

#f.

How did *same?* determine the answer?

The function first names the values of the *kdr*s. Then it changes them to different numbers. The answer is finally determined by comparing the values of the two *kdr*s. Finally, the *set-kdr*s change the respective *kdr*s so that they refer to their original values.

Here is the function *last-kons*

```
(define last-kons
  (lambda (ls)
    (cond
      ((null? (kdr ls)) ls)
      (else (last-kons (kdr ls))))))
```

Describe what it does.

The function *last-kons* returns the last *kons* in a non-empty *kons*-list.

```
(define long (lots 12))
```

Fine.

What does *long* refer to?

(egg egg egg egg egg egg
egg egg egg egg egg egg).

What would be the value of
(*set-kdr* (*last-kons long*) *long*)

Did you notice the subjunctive mood?

And then, what would be the value of
(*lenkth long*)

No answer.

What is the value of
(*set-kdr* (*last-kons long*) (*kdr* (*kdr long*)))

What is the value of
(*lenkth long*)

Still no answer.

Here is the function *finite-lenkth* which returns its argument's length, if it has one. If the argument doesn't have a length, the function returns false.

```
(define finite-lenkth
  (lambda (p)
    (letcc infinite
      (letrec
        ((C (lambda (p q)
               (cond
                 ((same? p q)
                  (infinite #f))
                 ((null? q) 0)
                 ((null? (kdr q)) 1)
                 (else
                   (+ (C (sl p) (qk q))
                      2)))))
          (qk (lambda (x) (kdr (kdr x))))
          (sl (lambda (x) (kdr x))))
        (cond
          ((null? p) 0)
          (else
            (add1 (C p (kdr p)))))))))
```

Bon appétit.

Guy's Favorite Pie

```
(define mongo
  (kons (quote pie)
    (kons (quote à)
      (kons (quote la)
        (kons (quote mode)
          (quote ()))))))
(set-kdr (kdr (kdr (kdr mongo))) (kdr mongo))
```

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.

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?

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:

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

Yes!

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.

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)))))
```

Do you remember tables from chapter 10?

A table is something that pairs names with values.

How did we represent tables?

We used lists and entries.

Could a table be anything else?

Yes, a function. A table acts like a function, because it pairs names with values, in the same way that functions pair arguments with results.

So let's use functions to make tables. Here is a way to make an empty table:

```
(define the-empty-table
  (lambda (name)
    ...))
```

Don't fill in the dots!

In *The Little Schemer* we used (*car* (**quote** ())).

What does that do?

It breaks The Law of Car.

If a table is a function, how can we extract whatever is associated with a name?

We apply the table to the name.

Write the function *lookup* that does that.

```
(define lookup
  (lambda (table name)
    (table name)))
```

Can you explain how *extend* works?

```
(define extend
  (lambda (name1 value table)
    (lambda (name2)
      (cond
        ((eq? name2 name1) value)
        (else (table name2))))))
```

Here are our words:
"It takes a name and a value together with a table and returns a table. The new table first compares its argument with the name. If they are identical, the value is returned. Otherwise, the new table returns whatever the old table returns."

What is the value of

```
(define x 3)
```

No answer.

What is (*value e*)
where
e is (define x 3)

No answer.

What is *value*

The name is familiar from chapter 10. But, the function *value* there does not handle (**define** ...).

So the new *value* might be defined like this.

```
(define value
  (lambda (e)
    ...
    (cond
      ((define? e) (*define e))
      (else (the-meaning e))) ... ))
```

Yes, this might do for a while. And don't bother filling in the dots, now. We will do that later.

Should we continue with (**letcc** ...) now?

Oh no!

Okay, we'll wait until later.

Whew!

Do we need *define?*

We don't need to define it now, because it is easy, but here it is anyway.

```
(define define?
  (lambda (e)
    (cond
      ((atom? e) #f)
      ((atom? (car e))
       (eq? (car e) (quote define)))
      (else #f))))
```

Do we need **define*

Yes, we need it. With (**define** ...), we can add new definitions.

Here is **define*

```
(define global-table
  ... the-empty-table ...)
```

```
(define *define
  (lambda (e)
    (set! global-table
      (extend
        (name-of e)
        (box
          (the-meaning
            (right-side-of e)))
        global-table))))
```

This function looks like one of those functions that remembers its arguments.

Yes, **define* uses *global-table* to remember those values that were **define**d.

The table appears to be empty at first.

Is it empty?

We shall soon find out.

When **define* extends a table with a name and a value, will the name always stand for the *same* value?

No, with (**set!** ...) we can change what a name stands for, as we have often seen.

Is this the reason why **define* puts the value in a *box* before it extends the table?

If we knew what a *box* was, the answer might be yes.

Here is the function that makes *box*es:

```
(define box
  (lambda (it)
    (lambda (sel)
      (sel it (lambda (new)
                (set! it new))))))
```

Does this remind you of something we have discussed before?

It should: *bons* from chapter 18 is a similar function.

Have we seen this before?

Remember *Y!* from chapter 16?

Is it important that we always have the most recent value of *global-table*

Yes, we will soon see why that is.

Here is *meaning*

```
(define meaning
  (lambda (e table)
    ((expression-to-action e)
     e table)))
```

What do you think the function *expression-to-action* does?

It translates *e* to a function that knows what to do with the expression and the table.

Do we need to define *expression-to-action*

No, we have seen it in chapter 10; it is easy; and it can wait until later.

Fine, we will consider it later.

Okay.

Here is the most trivial action.

```
(define *quote
  (lambda (e table)
    (text-of e)))
```

Can you define **identifier*

The function **identifier* is similar to **quote*, but it uses *table* to look up what a given name is paired with.

And what is a name paired with?

A name is paired with a box that contains its current value. So **identifier* must *unbox* the result of looking up the value.

And how does **identifier* look up the value?

It's best to have **identifier* use *lookup*, which finds the box that is paired with the name in the table.

```
(define *identifier
  (lambda (e table)
    (unbox (lookup table e))))
```

Okay one more:

```
(define beglis
  (lambda (es table)
    (cond
      ((null? (cdr es))
       (meaning (car es) table))
      (else ((lambda (val)
               (beglis (cdr es) table))
             (meaning (car es) table))))))
```

Can you define *box-all*

Trivial, with that kind of name:

```
(define box-all
  (lambda (vals)
    (cond
      ((null? vals) (quote ()))
      (else (cons (box (car vals))
                  (box-all (cdr vals)))))))
```

Take a look at *beglis*
What is
  ((**lambda** (*val*) ...)
   (*meaning* (*car es*) *table*))

It is the same as
  (**let** ((*val* (*meaning* (*car es*) *table*)))
   ...)
which first determines the value of (*meaning* (*car es*) *table*) and then the value of the value part.

Why didn't we use (**let** ...)

Our functions will work for all the definitions that we need for them. And they do not need to deal with expressions of the shape (**let** ...) because we know how to do without them.

How do you do without (**let** ...) in
  (**let** ((*x* 1)) (+ *x* 10))

Like this: it's the same as
  ((**lambda** (*x*) (+ *x* 10)) 1).

Do you remember how to do without (**let** ...) in
  (**let** ((*x* 1) (*y* 10)) (+ *x* *y*))

Yes, it's the same as
  ((**lambda** (*x* *y*) (+ *x* *y*)) 1 10).

So what does
  (**let** ((*val* (*meaning* (*car es*) *table*)))
   (*beglis* (*cdr es*) *table*))
do for *beglis*

First, it determines the value of (*meaning* (*car es*) *table*) and names it *val*. And then, it determines the value of (*beglis* (*cdr es*) *table*).

What happens to the value named *val*

Nothing. It is ignored.

Why did we determine a value that is ignored in the end?

Because the values of all but the last expression in the value part of a (**lambda** ...) are ignored.

Can you summarize now what the function *beglis* does for **lambda*

We summarize:
"The function *beglis* determines the values of a list of expressions, one at a time, and returns the value of the last one."

How does **lambda* work?

When given (lambda (x y ...) ...), it returns the function that is in the inner box of **lambda*.

What does that function do?

It takes the values of the arguments and apparently extends *table*, pairing each formal name, x, y, ..., with the corresponding argument value.

Write the function *multi-extend*, which takes a list of names, a list of values, and a table and constructs a new table with *extend*

No problem.

```
(define multi-extend
  (lambda (names values table)
    (cond
      ((null? names) table)
      (else
        (extend (car names) (car values)
          (multi-extend
            (cdr names)
            (cdr values)
            table)))))))
```

Okay, so now that we know how *table* is extended, what happens after the new table is constructed?

The function that represents a (lambda ...) expression uses the resulting table to determine the value of the body of the (lambda ...) expression, which was the first argument to **lambda*.

Which parts of the table can change even though the table stays the same?

Each box that the table remembers for any given name may change its value.

That's how (**set!** ...) works, right?

True.

Write *odd?* and *even?* as recursive functions.

Do you mean this pair of functions?

```
(define odd?
  (lambda (n)
    (cond
      ((zero? n) #f)
      (else (even? (sub1 n))))))
```

```
(define even?
  (lambda (n)
    (cond
      ((zero? n) #t.)
      (else (odd? (sub1 n))))))
```

Yes, what is (*value* *e*)
where
e is

```
(define odd?
  (lambda (n)
    (cond
      ((zero? n) #f)
      (else (even? (sub1 n))))))
```

No answer.

What is (*value* (**quote** odd?))

A function.

Which table does the function use when we ask (*value* *e*)
where
e is (odd? 0)

The function extends *lookup-in-global-table* by pairing n with (a box containing) 0.

And then?

Eventually we get the result: #f.

What kind of function does **application* expect from (*meaning e table*) where
e is car

It will need to be a function that takes all of its arguments in a list and then does the right thing.

How many values should the list contain that (*meaning* (**quote** car) *table*) receives?

Exactly one.

And what kind of value should this be?

The value must be a list. And then we take its *car*.

Define the function that we can use to represent *car*

Let's call it *:car*.

```
(define :car
  (lambda (args-in-a-list)
    (car (car args-in-a-list))))
```

Are there other primitives for which we should have a representation?

Yes, *cdr* is one, and *add1* is another.

We should have a function that makes representations for such functions.

Here is one:

```
(define a-prim
  (lambda (p)
    (lambda (args-in-a-list)
      (p (car args-in-a-list)))))
```

We also need one for functions like *cons* that take two arguments.

No problem: now the argument list must contain exactly two elements, and we just do what is necessary:

```
(define b-prim
  (lambda (p)
    (lambda (args-in-a-list)
      (p (car args-in-a-list)
         (car (cdr args-in-a-list))))))
```

And now we can define **const*

```
(define *const
  (lambda (e table)
    (cond
      ((number? e) e)
      ((eq? e #t) #t)
      ((eq? e #f) #f)
      ((eq? e (quote cons))
       (b-prim cons))
      ((eq? e (quote car))
       (a-prim car))
      ((eq? e (quote cdr))
       (a-prim cdr))
      ((eq? e (quote eq?))
       (b-prim eq?))
      ((eq? e (quote atom?))
       (a-prim atom?))
      ((eq? e (quote null?))
       (a-prim null?))
      ((eq? e (quote zero?))
       (a-prim zero?))
      ((eq? e (quote add1))
       (a-prim add1))
      ((eq? e (quote sub1))
       (a-prim sub1))
      ((eq? e (quote number?))
       (a-prim number?)))))
```

Can you rewrite **const* using (**let** ...)

Where? Why? There are no repeated expressions.

What is (*value e*)
where
e is

```
(define ls
      (cons
        (cons
          (cons 1 (quote ()))
          (quote ()))
        (quote ())))
```

We add ls to *global-table* and rember what it stands for.

What is (*value e*)
where
e is (car (car (car ls)))

1.

How do we determine this value?

It is an application, so we need to find out what car is and the value of the argument.

How do we determine the value of car

We use the function **const*:
(**const* (**quote** car)) tells us.

And that is?

It is the same as (*a-prim car*), which is like *:car*.

How do we determine the value of the argument?

It is an application, so we need to find out what car is and the value of the argument.

(*value* (**quote** car))

We use the function **const*:
(**const* (**quote** car)) tells us.

And?

It is the same as (*a-prim car*), which is like *:car*.

How do we determine the value of the argument?

It is an application, so we need to find out what car is and the value of the argument.

(*value* (**quote** car))

We use the function **const*:
(**const* (**quote** car)) tells us.

How often did we have to figure out the value of (*a-prim car*)

Three times.

Is it the same value every time?

It sure is.

Is this wasteful?

Yes: let's name the value!

Can we really use (**let** ...)

We can: we just saw how to replace it.

Where do we put the (**let** ...)

Around (**cond** ...)?

When would we determine the values in this (**let** ...)

Each time **const* determines the value of **car**.

So this wouldn't help.

Let's put the (**let** ...) outside of (**lambda** ...).

Here is **const* with (**let** ...)

```
(define *const
  (let ((:cons (b-prim cons))
        (:car (a-prim car))
        (:cdr (a-prim cdr))
        (:null? (a-prim null?))
        (:eq? (b-prim eq?))
        (:atom? (a-prim atom?))
        (:number? (a-prim number?))
        (:zero? (a-prim zero?))
        (:add1 (a-prim add1))
        (:sub1 (a-prim sub1))
        (:number? (a-prim number?)))
    (lambda (e table)
      (cond
        ((number? e) e)
        ((eq? e #t) #t)
        ((eq? e #f) #f)
        ((eq? e (quote cons)) :cons)
        ((eq? e (quote car)) :car)
        ((eq? e (quote cdr)) :cdr)
        ((eq? e (quote null?)) :null?)
        ((eq? e (quote eq?)) :eq?)
        ((eq? e (quote atom?)) :atom?)
        ((eq? e (quote zero?)) :zero?)
        ((eq? e (quote add1)) :add1)
        ((eq? e (quote sub1)) :sub1)
        ((eq? e (quote number?))
         :number?)))))
```

Can you rewrite **const* without (**let** ...)

```
(define *const
  ((lambda (:cons :car :cdr :null?
            :eq? :atom?
            :zero? :add1 :sub1 :number?)
     (lambda (e table)
       (cond
         ((number? e) e)
         ((eq? e #t) #t)
         ((eq? e #f) #f)
         ((eq? e (quote cons)) :cons)
         ((eq? e (quote car)) :car)
         ((eq? e (quote cdr)) :cdr)
         ((eq? e (quote null?)) :null?)
         ((eq? e (quote eq?)) :eq?)
         ((eq? e (quote atom?)) :atom?)
         ((eq? e (quote zero?)) :zero?)
         ((eq? e (quote add1)) :add1)
         ((eq? e (quote sub1)) :sub1)
         ((eq? e (quote number?))
          :number?))))
   (b-prim cons)
   (a-prim car)
   (a-prim cdr)
   (a-prim null?)
   (b-prim eq?)
   (a-prim atom?)
   (a-prim zero?)
   (a-prim add1)
   (a-prim sub1)
   (a-prim number?)))
```

The Seasoned Schemer

Daniel P. Friedman and Matthias Felleisen
Drawings by Duane Bibby
Foreword and Afterword by Guy L. Steele Jr.

The notion that "thinking about computing is one of the most exciting things the human mind can do" sets both *The Little Schemer* (formerly known as *The Little LISPer*) and its new companion volume, *The Seasoned Schemer*, apart from other books on LISP. The authors' enthusiasm for their subject is compelling as they present abstract concepts in a humorous and easy-to-grasp fashion. Together, these books will open new doors of thought to anyone who wants to find out what computing is really about.

The Little Schemer introduces computing as an extension of arithmetic and algebra — things that everyone studies in grade school and high school. It introduces programs as recursive functions and briefly discusses the limits of what computers can do. The authors use the programming language Scheme and a menu of interesting foods to illustrate these abstract ideas. *The Seasoned Schemer* introduces the reader to additional dimensions of computing: functions as values, change of state, and exceptional cases.

The Little LISPer has been a popular introduction to LISP for many years. It has appeared in French and Japanese. *The Little Schemer* and *The Seasoned Schemer* are worthy successors and will prove equally popular as textbooks for Scheme courses as well as companion texts for any complete introductory course in Computer Science.

Daniel P. Friedman is Professor in the Computer Science Department, Indiana University, and Matthias Felleisen is Professor in the Computer Science Department, Rice University. Together they have taught courses on computing and programming with Scheme for more than 25 years and published over 100 papers and three books on these topics.

The MIT Press
Massachusetts Institute of Technology
Cambridge, Massachusetts 02142

9 780262 561006 90000

0-262-56100-X
www-mitpress.mit.edu